FPGA Based Deep Learning Accelerator for RF Applications

A Design Framework

by

Hans den Boer

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday July 13, 2021 at 15:00.

Student number:	4966244	
Thesis committee:	Dr. Ir. J.S.S.M Wong,	TU Delft, supervisor
	Dr.Ir. T.G.R.M. van Leuken	TU Delft
	V. Voogt,	TNO

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Preface

This master thesis marks the end of my master's degree in Computer Engineering at TU Delft. Because of Corona, I did this master thesis project almost completely from home, and that is why I am especially grateful that TNO gave me the opportunity to do this research.

There are several people I would like to thank, namely the people at TNO with whom I have worked and in particular my supervisor at TNO, Vincent Voogt. As well as my supervisor from TU Delft, Stephan Wong. Finally, I would like to thank my family for supporting me throughout my educational career.

Hans den Boer
Den Haag, July 2021

Abstract

Recently, interest in the use of deep learning technology for RF applications has increased. However, many of these studies are focused on developing deep learning models for a particular RF application. Therefore this master thesis focuses on the implementation of these kinds of deep learning models by using FPGAs such that these deep learning models can be used in an FPGA-based Software Defined Radio.

In this master thesis, a custom FPGA accelerator is designed for CNN models using reusable and configurable building blocks. The accelerator employs a streaming architecture and is fully pipelined, such that it accepts new input data every clock cycle. A key design aspect is that all building blocks in the accelerator are designed to be able to work on a portion of its input data. The implication is that the building blocks can produce an output as soon as enough input data is available. As a result, the work that the building blocks have to perform is spread out over time and the memory required for storing data is also reduced. Moreover, the precision of the fixed point parameters and operations is configurable. Therefore there is no limitation of only specifically supporting binary or ternary operations.

This accelerator has been tested for the automatic modulation classification problem. The result is an accelerator that can process real-time data at 600MHz and consume fewer FPGA resources than other similar initiatives. In a direct comparison with hls4ml, the designed custom accelerator achieves 2.4 times higher throughput and 2.3 times lower latency for the identical CNN, while also achieving the same accuracy and significantly lower resource utilization. In addition, the custom accelerator is compared to a ternary neural network FPGA accelerator for modulation classification as proposed by Tridgell et al. The custom accelerator uses 3.3 times fewer LUTs, 9 times fewer FFs, 4 times fewer DSPs, and uses no BRAM, while the accelerator proposed by Tridgell et al. uses 48.5% of the available BRAM in an RFSoC FPGA.

Hans den Boer
Den Haag, July 2021

Acronyms

ADC	Analog-to-Digital Converter.
AI	Artificial Intelligence.
ANN	Artificial Neural Network.
ASIC	Application Specific Integrated Circuit.
BNN	Binary Neural Network.
BRAM	Block Random Access Memory.
CNN	Convolutional Neural Network.
CPU	Central Processing Unit.
CR	Cognitive Radio.
DL	Deep Learning.
DNN	Deep Neural Network.
DSP	Digital Signal Processing/Processor.
FF	Flip-Flop.
FFT	Fast Fourier Transform.
FIFO	First In, First Out.
FPGA	Field-Programmable Gate Array.
FPS	Frames Per Second.
GPU	Graphics Processing Unit.
GRU	Gated Recurrent Unit.
HDL	Hardware Description Language.
HLS	High Level Synthesis.
HPC	High Performance Computing.
LSTM	Long Short-Term Memory.
LUT	Lookup Table.
MACC	Multiply-And-Accumulate.
MLP	Multilayer Perceptron.
QNN	Quantized Neural Network.
ReLU	Rectified Linear Activation Unit.
ResNet	Residual Neural Network.
RF	Radio Frequency.
RFIC	Radio Frequency Integrated Circuits.
RFNoC	Radio Frequency Network on a Chip.
RNN	Recurrent Neural Network.
SDR	Software Defined Radio.
SNR	Signal-to-Noise Ratio.

Glossary

Deep Learning	Deep Learning is a machine learning method based on utilizing Artificial Neural Networks.
II	In hardware design, Initiation Interval (II) is the delay between successive initiations of a function or a loop. For example, a function with an II of 1, means that there is a delay of 1 clock cycle between each initiation of the function.
IP core	Semiconductor intellectual property core is a reusable unit of logic or data that is the intellectual property of one party. IP cores can be used as building blocks in ASICs and FPGAs.
IQ	In-phase component (I) and quadrature component (Q) are used for most modern RF signal generation, modulation and demodulation.
Machine learning at the edge	Machine learning and especially deep learning algorithms are often computationally intensive and therefore these algorithms commonly run on powerful CPU/GPU platforms. However there is a trend to bring these algorithms, albeit in a reduced form, to edge devices. Where edge devices can be a device that is 'close' to the source of the data, such as a (handheld) communication device.

Contents

1 Introduction

1.1. Context of Assignment

Whilst the economic importance of 'being' connected seems to increase without bounds, the expectations of wireless mobile communication technologies seem to increase as well. As a result, the availability of frequency spectrum that can be used for such technologies has become quite an expensive commodity as well. Large corporations are willing to invest a great deal of money to obtain the license to be the sole user of parts of the frequency spectrum. The economically beneficial implications of a highly connected society, as well as the economic interests of the corporations that facilitate this mobile communication infrastructure, have given rise to a political lobby that aims to free up increasingly more parts of the frequency band for auctioning, thus pushing away the current users of such bands.

In practice, however, large parts of the frequency spectrum remain relatively underused despite this demand for more spectrum. One of the reasons for this is due to the rather static spectrum management that is currently employed in most communication protocols. The concept of **Cognitive Radio** (CR), which was first proposed in 1998 by Joseph Mitlola III [1], [2], aims to increase the spectrum efficiency by using a dynamic spectrum management approach. Cognitive radios are typically radios which are capable of sensing their electromagnetic environment and can subsequently select unused parts of the spectrum (in frequency and/or time) for their own transmissions, possibly using transmission parameters (such as the waveform) which fit the selected transmission opportunity. As such the cognitive radio does not only need to be aware of other radio signals, but it also needs to be capable of recognizing those signals for which they are the intended receiver (in an uncoordinated fashion).

Even though the concept of cognitive radio is quite old, it has proven difficult to develop a fully cognitive radio system. One of the most important limitations is the required processing power to achieve the level of spectrum awareness that is required, in real-time. Recently, however, the concept of cognitive radio has received a renewed interest with the advent of **deep learning** (DL) technology. Deep learning is a subclass of artificial intelligence (AI) and is a rapidly developing research field. In deep learning, models based on neural networks are trained, such that they can infer something about unseen data.

In general, the use of deep learning for communication networks is still quite new. Currently, several companies have started to focus on the use of deep learning for improving communication technologies [3], [4], and also in scientific literature a number of papers have been published in this particular sub-field. Despite the newness, the first results are promising and show the potential that deep learning can offer in the (near) future.

Still, the availability of deep learning technology is not sufficient. Because an important aspect of these studies into deep learning for radio systems is that it must be applicable in practice. Most radio systems rely on Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs) for processing. Most deep learning models however run on CPU or GPU platforms, which are typically not desired for digital signal processing, either because of the sequential instruction nature or their power requirements. Therefore, it is useful to execute deep learning models on these communication systems as well.

There are other domains where deep learning is already being applied much more in practice. In particular, the vision/imaging domain is a major stimulant for the development and applicability of deep learning models. A good example of this domain is autonomous cars, in which artificial intelligence enables a car to

be aware of its surroundings and determine appropriate actions.

Recently, hardware is available to bring AI capabilities to edge devices. These devices are primarily designed for image processing applications and are generally not designed for RF radio systems in mind. There are several differences between applications for the RF domain and the imaging domain. One such difference is the possible data rates. To illustrate this difference, a comparison between RF and imaging domain is made for realistic systems. Full HD Cameras (resolution of 1920 x 1080) with a frame rate of 60 FPS, produce more than 124 million pixels per second. Although for autonomous cars the frame rate of cameras may be lower depending on the function of a camera on a car [5]. These data rates can be lower than the data rate of communication systems. Recently, Xilinx released RFSoC wherein an RF transceiver is integrated with an FPGA in a single chip [6]. Depending on the RFSoC type, a maximum data rate of 5 Giga samples per second of IQ data is possible for 8 channels. These data rates completely surpass the data rates in the vision domain. Even a simple RF transceiver such as the the AD9361 chip from Analog Devices, has a maximum sample rate of 61.44 Mega samples per second of IQ data. A high data rate may be a requirement for a deep learning based RF application.

As indicated earlier, the use of deep learning for RF applications requires a means of integrating these models into communication systems. The assignment of this work is therefore to investigate the possibilities for integrated deep learning models on FPGA platforms, such that they can operate as part of the signal processing chain in a software defined radios (SDR). This master thesis project is therefore aimed at facilitating further research into the use of RF deep learning in communications systems.

1.2. About TNO

This Master thesis project is commissioned by TNO[1] (Netherlands Organisation for Applied Scientific Research). The TNO's Electronic Defence department is investigating the use of deep learning for various communication applications. Figure 1.1 provides an overview of how the process of this research is structured.

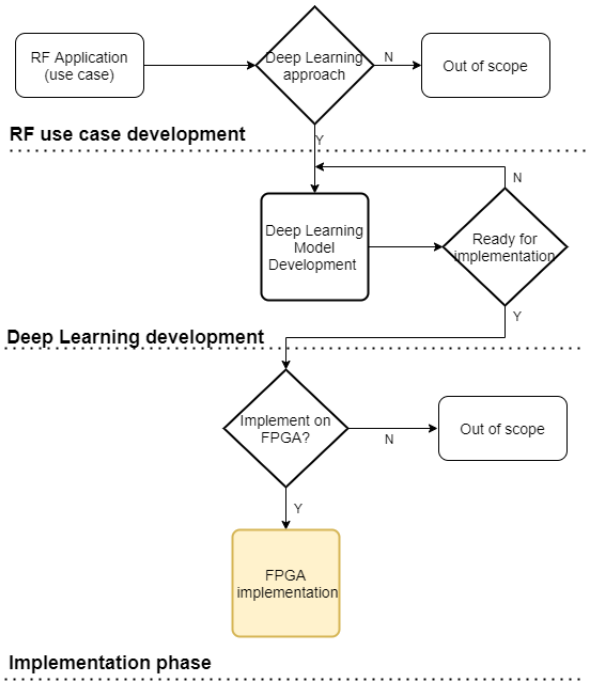


Figure 1.1: Process of RF deep learning research at TNO.

There are three main aspects of the deep learning research performed at TNO. Firstly, there is an RF application for which it needs to be determined whether a deep learning approach is suitable. Secondly, if opted for a deep learning approach, one or more deep learning models are developed. These models must be suitable for implementation. Finally, in the implementation phase, the implementation platform is determined.

[1] https://www.tno.nl/en/

The implementation platform could be a CPU or GPU platform, but in this master thesis, we specifically focus on an FPGA based implementation.

It is important to note that these three aspects are not independent of each other and are not necessarily followed consecutively. For example, the knowledge building for the FPGA implementation can influence how to train a deep learning model and how feasible an RF use case is with current capabilities. Limitations with respect to implementing deep learning models on the FPGA can also limit the possibilities in the 'RF Deep Learning development' phase for use-cases where the FPGA is essential. Therefore it is of importance that an implementation strategy will minimize any limitations in this phase. So far, TNO's research has mainly focused on the first two aspects. And the research in this master thesis is aimed at the implementation phase, to make deep learning technology practically applicable in communication systems.

1.3. Problem statement

In Section 1.1 we explained that the implementation of RF deep learning models on the FPGA can form a requirement for cognitive radio technology. Moreover, the FPGA implementation may have to deal with a very high data rate. Cognitive radio itself is a complex and comprehensive subject, and therefore this master thesis will focus on a subpart of cognitive radio, namely the **modulation classification** use case. Modulation classification is essential to carry out multiple cognitive radio techniques, such as dynamic spectrum access [7]. TNO is already investigating the modulation classification use case, and both academic and industry are also researching this specific topic. For this use case it is desirable to run deep learning models on the FPGA.

The problem statement can be formulated from this:

How can FPGAs be efficiently utilized to implement deep learning models for modulation classification in a Software Defined Radio (SDR)?

This problem statement can be divided into sub research questions. The list below contains sub research questions which provides the skeleton to answer the main research question:

1. What do deep learning models for modulation classification looks like?
    - What are the requirements for these models?
2. What existing solutions are there for deep learning acceleration using FPGAs?
    - Are these solutions sufficiently suitable and compatible with RF deep learning applications? If not, can modifications be made?
    - What characteristics do these existing solutions have?
    - Are these solutions open-source, such that modifications can be made?
3. What requirements are there for FPGA deep learning acceleration?
    - Are there specific characteristics that are important for RF applications?
4. How can FPGAs be used in a broader context within RF applications than just modulation classification?

1.4. Project goals & Methodology

The goal of this thesis is to present an implementation method for deploying RF deep learning models to an FPGA for use in software defined radio systems. To achieve this goal, several sub-goals are defined.

Goal 1: Research deep learning models for modulation classification The first goal is to get insight into RF deep learning models. In this case, specifically the modulation classification use case is being investigated. As explained in the previous section, modulation classification is a good use case to look at as it has already been published in the academic literature.

Goal 2: Evaluate feasibility of currently available solutions There are already available solutions for FPGA deep learning acceleration. The various solutions need to be explored to assess whether they are suitable for RF deep learning applications. A specific aspect that is important, is the extent to which these solutions are applicable in software defined radios. To assess this, certain requirements that may be important for certain RF applications must be taken into account, such as throughput requirements. The modulation classification use case is not specific for high throughput, because the throughput requirements depend on the type of signal. Therefore, this use case can be used to investigate different solutions, which may be suitable for different applications.

Goal 3: Develop implementation method Based on the results of goal 2, the design of an implementation method can be made. If existing solutions prove to be sufficiently suitable for FPGA deep learning acceleration used for applications within the RF domain, then the implementation method will be based on these existing solutions. In the other case, a custom accelerator must be developed.

The design is not specifically intended for the modulation classification use case. This application is used only because this is a clear example application for the RF domain where comparisons are possible. Cognitive radio has many more aspects where deep learning can represent an important role. That is why it is useful to this into account.

Goal 4: Evaluate the implementation method Based on the modulation classification use case, the implementation will be evaluated. Comparisons with existing solutions can be made.

Goal 5: Develop a demonstration of a modulation classifier The main goal of this research is to make deep learning applicable for communication applications. It is useful to actually demonstrate the modulation classification use case in which a deep learning classifier runs on an FPGA based software defined radio platform.

1.5. Overview of the thesis

This thesis is organized as follows:

- **Chapter 2: Background**
  This chapter describes the background of the three main aspects of this thesis. Firstly, the software defined radio concept is introduced, and the role of FPGAs in software defined radios is discussed. Secondly, the required background knowledge for neural networks is explained, especially of importance are the discussed building blocks that are present in common neural networks. Finally, the relevant related work is discussed in deep learning for RF applications as well as FPGA acceleration of neural networks.

- **Chapter 3: Preliminary Experiments**
  In this chapter two different tools (Vitis AI and hls4ml) for neural network FPGA acceleration are tested for applicability for RF applications. These are reviewed in-depth, such as their architecture and their throughput capabilities. The conclusion of this chapter is whether these tools provides a suitable basis for FPGA acceleration of RF deep learning or if it is necessary to develop a custom FPGA acceleration tool.

- **Chapter 4: Implementation**
  This chapter discusses the development of a custom deep learning FPGA accelerator. Firstly, the design goals for this implementation are summarised. Subsequently, the actual design is discussed.

- **Chapter 5: Evaluation**
  The developed custom deep learning FPGA accelerator is tested with deep learning models for modulation classification use case. Subsequently, the results of this custom FPGA accelerator are discussed, and the custom accelerator is compared with already existing solutions.

- **Chapter 6: Demonstration**
  This chapter demonstrates a working modulation classifier deployed to an FPGA in a software defined radio system.

- **Chapter 7: Conclusions**
  Finally, this last chapter contains the conclusions of the entire thesis. Furthermore, the contributions made in this master thesis are summarized and recommendations for further work are given.

2

Background

This chapter aims to provide a basic background with respect to the research field connected to this thesis. Here we distinguish four main research fields: Software Defined Radio (SDR), Neural Networks, Modulation Classification and Field-Programmable Gate Array (FPGA). The latter having a specific focus on the implementation aspects of Deep Learning (DL) models.

This chapter is structured as follows. The first Section 2.1 explains the Software Defined Radio concept. Subsequently, Section 2.2 provides the necessary background on neural networks. Thereafter, Section 2.3 explains the concept of modulation use case and discusses the related work on this topic. Finally, Section 2.4 provides background on FPGA deep learning acceleration and also discusses related work.

2.1. Software Defined Radio

2.1.1. Concept

The concept of SDR aims to implement radio transceiver systems in which as much of the signal processing is handled in the digital domain (digital signal processing (DSP)) and, consequently, with a minimum of radio frequency (RF) hardware components. Contrary to the 'classical' radio concept where all signal processing is handled in RF components (such as filter structures) the SDR concept provides a re-programmable radio system. The result is a highly flexible radio transceiver which can change its transmission and receiver parameters through a simple software update (or reconfiguration of FPGA).

Note, however, that the actual amount of flexibility still depends on the actual hardware used for the transceiver. For example, while ASICs can be used to handle (parts of) the processing in a very efficient way they often offer a much lower degree of re-programmability. As a more flexible alternative DSP microprocessors can be used, which implement specific DSP functions in a very efficient way and offer more re-programmability than ASICs.

On the other hand the hardware must also be capable of performing the actual DSP within the time restrictions of the actual radio application. CPUs for example, which arguably offer the highest degree of re-programmability, are typically very inefficient in handling most DSP functions.

FPGA technology seems to offer the middle ground when it comes to both re-programmability and DSP capabilities. Due to it's inherent ability to handle parallel processing of data especially well, but compared to CPU processing it offers limited reconfigurability during run-time. It is therefore not surprising that FPGA technology can already be found in many telecommunication systems such that mobile network base transceiver stations (BTS).

2.1.2. Developments

The concept of SDR has gained a lot of popularity in recent years. Especially, developments in the RF integrated circuits (RFIC) and FPGAs resulted in SDR systems, which are available for both professional and hobbyist applications. A good overview of these developments of SDR concepts are given in [8]. From this reference three SDR concepts are displayed in Figure 2.1.

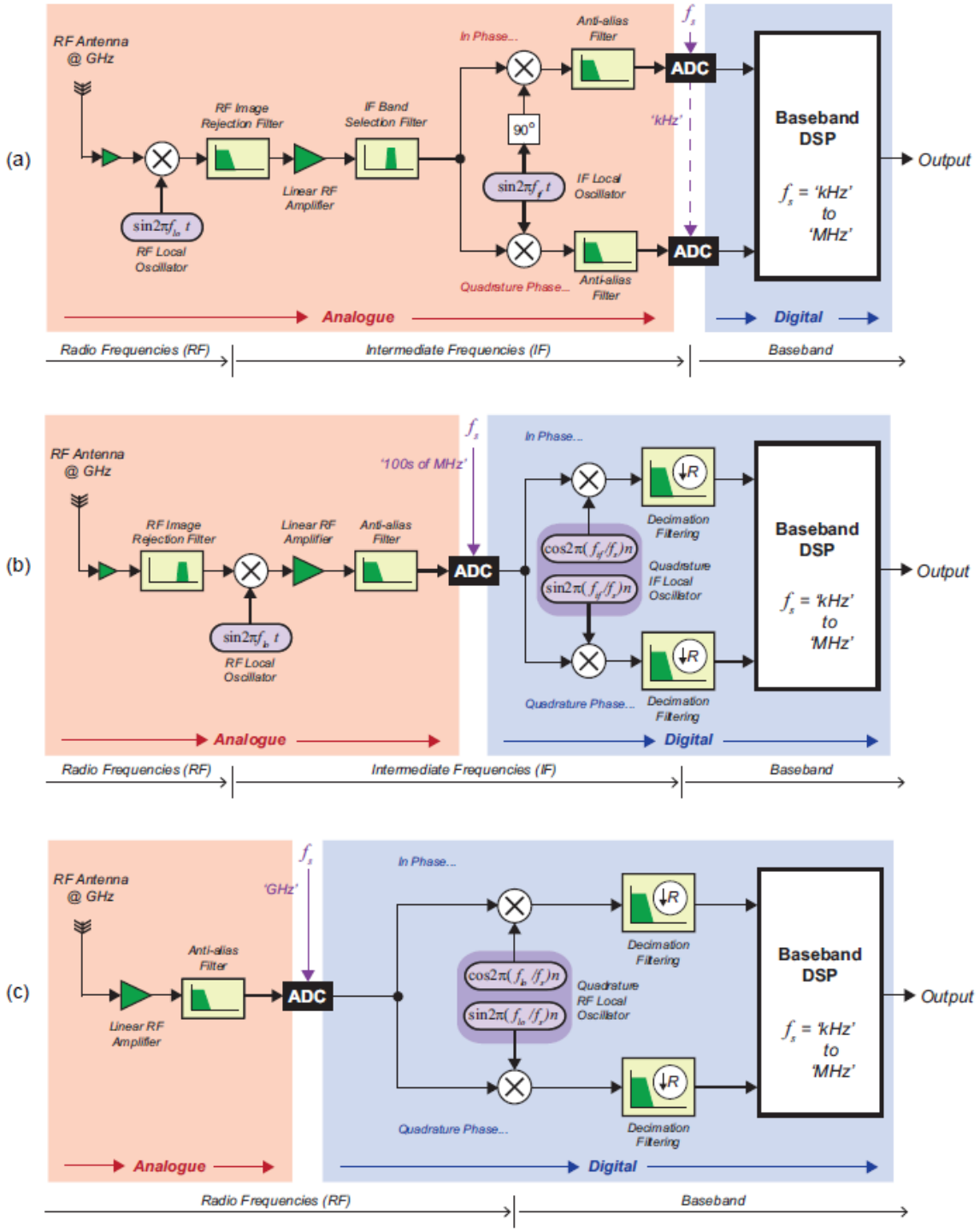


Figure 2.1: Three SDR types (a) baseband SDR, (b) IF SDR, (c) zero-IF SDR. From [8]

- The first SDR type is the baseband SDR, which is the oldest category. This type of SDR is tuned on a certain carrier frequency and subsequently samples the analog data to digital samples using an analog-to-digital-converter (ADC) with a relatively low sample rate (up to a few MS/s). This type of receiver is typically used to receive a single signal with limited bandwidth. As can be seen from the figure this type of SDR still uses considerable amount of RF components. Currently, baseband SDRs are commercially available for under 20 euros, for example in the form of an RTL-SDR [9] (a Digital Video Broadcast - Terrestial (DVB-T) receiver that can also output raw output from the ADC).
- The next SDR type is called the intermediate frequency (IF) SDR. Compared to the baseband SDR, analog to digital conversion takes place at the intermediate frequency level using an ADC with a higher sample rate (few tens to hundreds of MS/s). In these kind of SDRs further mixing and decimation takes place in the digital domain. As a result these types of receivers are capable of simultaneously receiving multiple signals within the same band, faster (digital) frequency re-tuning (no oscillator so no lock time) or other applications such as wideband spectrum monitoring. This type of SDRs are commercially available at manufactures such as Ettus Research, Analog Devices and some smaller manufacturers. Depending on RF specifications (such as tuning range, number of transmit and receive channels, bit depth and sampling rate) these devices can cost anywhere from a few hundred to thousands of euros.
- The last SDR type is the zero-IF SDR. This type of SDR has a minimal amount of RF components and almost all signal processing is performed in the digital domain. The ADC typically has a sample rate in the order of a few gigasamples per second. Some examples are currently available, but are mainly intended for R&D work. One such example is the Xilinx Zynq UltraScale+ RFSoC in the price range from about 10 to 20 thousand euros.

Where most IF SDR equipment still mainly relies on the CPU for processing (the SDR 'hardware' only samples the data and transmits it to a general purpose computer using a USB or Ethernet cable) there is a trend towards implementing more of the DSP functions on the FPGA. The reason for this trend is logical: while having an ADC with a higher sampling rate is beneficial for the re-programmability and use of the SDR platform, it has the downside that more samples per second also means a greater amount of data that needs to be processed. Furthermore, the CPU is not the most scalable platform to implement DSP functions on, hence the utilization of FPGAs for implementing these DSP functions.

As a result new initiatives are being offered to facilitate DSP processing on the FPGA. Ettus Research offers the RF Network on a Chip (RFNoC) framework [10], [11] that is compatible with their newer SDR platforms, whereas Xilinx offers the Vitis framework [12] aimed towards offloading certain functions from the CPU to the FPGA. As an alternative one can also simply implement the DSP processing directly in the FPGA logic. In the end however, all solutions require the DSP processes to be implemented in some hardware description language (HDL), either directly or using a high level synthesis (HLS) solution.

2.2. Neural networks

This section gives a brief overview of various types of neural networks. For a more extensive reference of neural networks refer to [13].

Deep learning is a type of machine learning based on artificial neural networks (ANN). Although the name suggest otherwise, artificial neural network are developed with no or loose connections to neuroscience, according to G. Indiveri et al [14]. Still, ANNs and biological neural networks are similar in that they learn from data (or experience) and can use it to make predictions.

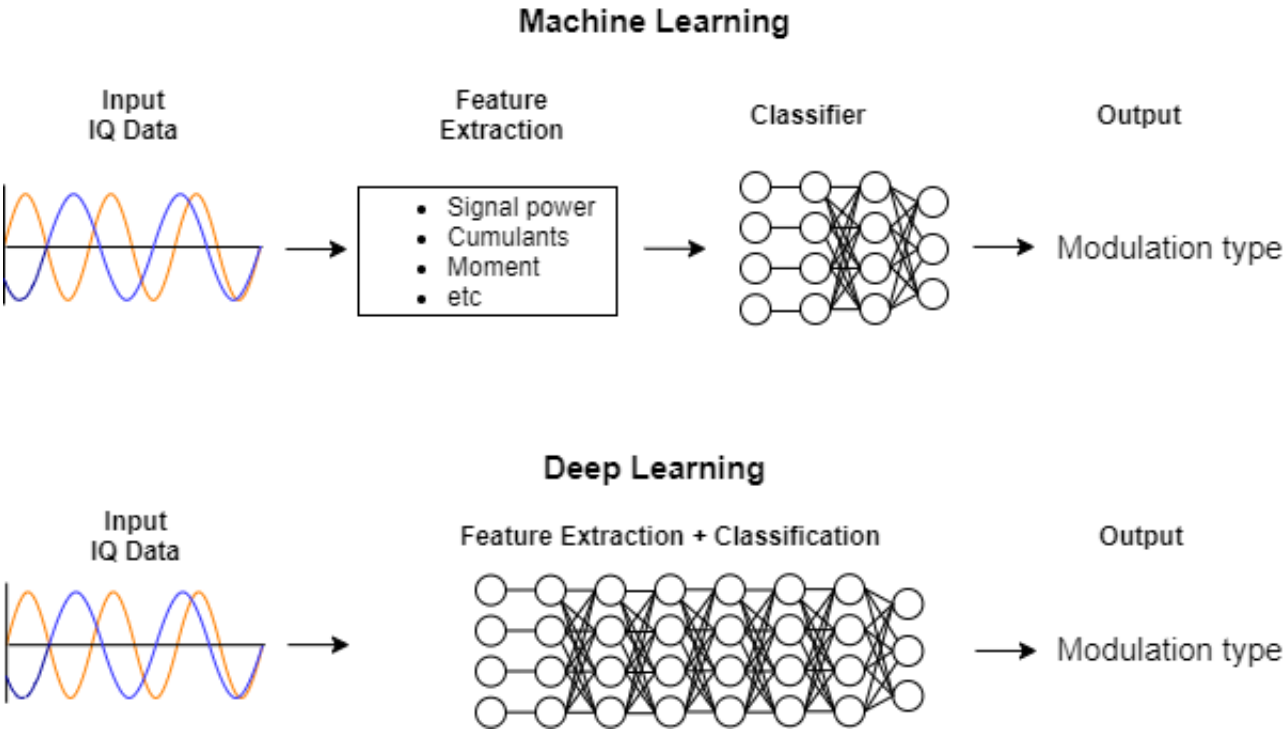


Figure 2.2: Illustration of difference between machine learning and deep learning.

The difference between machine learning and deep learning is illustrated in Figure 2.2. In machine learning features are extracted from raw input data, and these features are used in a classifier to make predictions, also called **inference**. Examples of features in the case of modulation classification prediction are statistical properties, such as moment and cumulants, and instantaneous properties such as signal power [15]. In machine learning it can be difficult to know what features should be extracted [13]. Deep learning attempts to solve this issue by combining the feature extraction and classifier into one model such that the model learns the features autonomously from the training dataset. The implication is that the dataset has a major influence on the quality of the deep learning model.

The next sections will cover the basics of neural networks, such as the training and inference of deep learning models. Furthermore, a number of different artificial neural networks are discussed: feedforward neural networks, convolutional neural networks (CNN), residual neural networks (ResNet), recurrent neural networks (RNN) and quantized neural networks (QNN). Finally, an overview of common building blocks in neural networks is given.

2.2.1. Basics

A basic neural network is depicted in Figure 2.3. This network is called a feed-forward connected neural network (FFNN) or multilayer perceptron (MLP). It is called a feed-forward network, because information flows only in one direction. In this example there are 4 input nodes, 2 hidden layers with each 3 nodes, and 1 output node. The input nodes are simply the input data, for an image this can be the pixel values and for RF data this can be IQ data samples from an ADC. The hidden layers and the output layer are known as fully-connected or dense layers. In a fully connected layer, each node receives the output of all the nodes in the preceding layer. Each node has a set of weights which are multiplied with the inputs to that node. The result

is accumulated and a bias is added, this result is passed to the activation function of the node. An activation function is often used to add non-linearity to the node, otherwise the neural network cannot learn non-linear functions. A summary of common activation functions is provided in Section 2.2.6. Through multiple hidden layers, the input values are transformed to form an output (or multiple outputs).

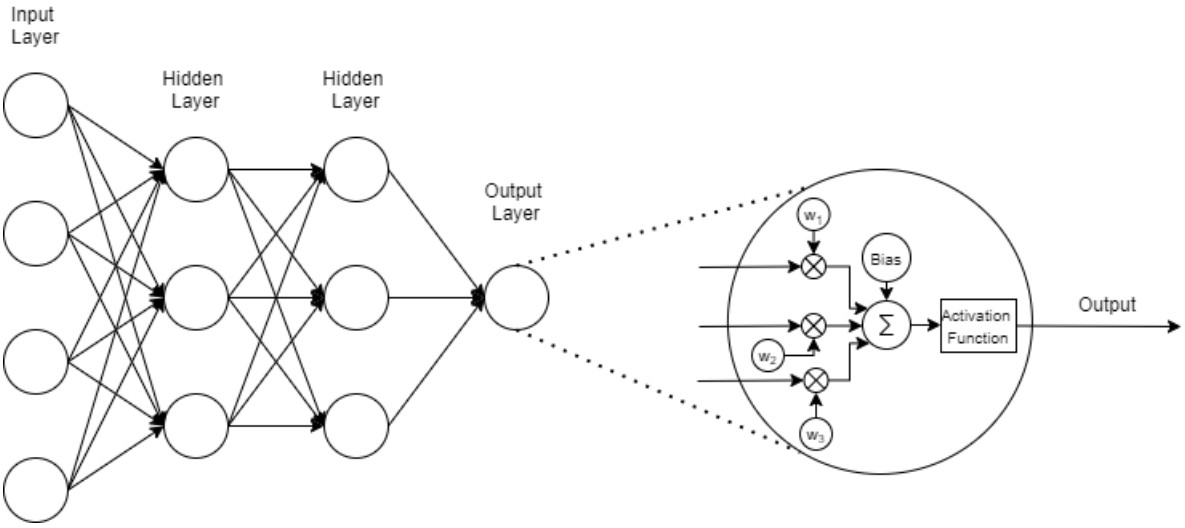


Figure 2.3: Example of a feedforward neural network.

A labeled dataset is used by the neural network in the training phase to learn the parameters, such as the weights and biases. For example, in a dataset for modulation classification, each example contains a label with the modulation type of that particular example. This use of a labeled dataset to construct machine/deep learning model is called supervised learning.

Training a neural network consist of two steps, a forward propagation and a backward propagation. In the forward propagation, the examples from the training dataset are fed to the network. The output of the network is compared to the labels using a loss function. The goal of training is to minimize the loss function. During backward propagation the network is traversed from output to input and the parameters are updated. Training a neural network is done by alternating the forward propagation and backward propagation. The goal of the training of the model is to make predictions about unseen data, in other words data that is not part of the training dataset. Therefore it is important that the model generalizes enough. It is called overfitting when a model only makes correct predictions about the data that is used for the training of the model. It is important that the dataset captures enough of target distribution. In the case of modulation classification RF signals can have the same modulation but different signal to noise ratios (SNR). Therefore, to train a deep learning model to correctly classify the modulation of signals, the dataset should have examples of RF signals with different SNRs. Otherwise, the situation may arise that the model can only properly classify signals with a specific SNR.

Training of deep learning is typically done on computer systems with CPUs and GPUs. Especially the backward propagation is costly in terms of memory. Deep learning acceleration on FPGAs is typically only for the forward propagation, such that the **inference** is done on FPGAs, but training of the model is done on CPU/GPU systems.

2.2.2. Convolutional Neural Networks

The most basic neural network consisting of only fully connected layers is already described in the previous section. Another common neural network type is a convolutional neural network (CNN). Convolutional neural networks are used extensively for computer vision tasks. In computer vision the input data are often 2D signals, such as images and video frames. For time series data such as RF signals, the data are often 1D.

A basic CNN is depicted in Figure 2.4. This figure shows that CNNs can be divided into a feature extraction part and a classification part. The feature extraction is done through convolution layers. The classification is often done by one or multiple fully connected layers which are discussed in Section 2.2.1. It should be noted that the fully connected network also extracts features from the input, but in a CNN the separation between feature extraction and classification parts is more evident.

The 1D convolutional operation in CNNs is shown in Figure 2.5(a). This figure shows a kernel, which contains trainable parameters, that slides over the 1D input samples and performs the convolution operation. This figure also shows that for a kernel size of 3 and a input length of 6 samples, the output length will be 4. This is called the border effect, therefore often padding is added, such that the input and output length are equal, which is shown in Figure 2.5(b). Many neural network libraries actually implement a cross-correlation function, instead of the convolution operation [13]. But this is not a problem, because the difference between convolution and cross-correlation is simply flipping the kernel (weights) [13].

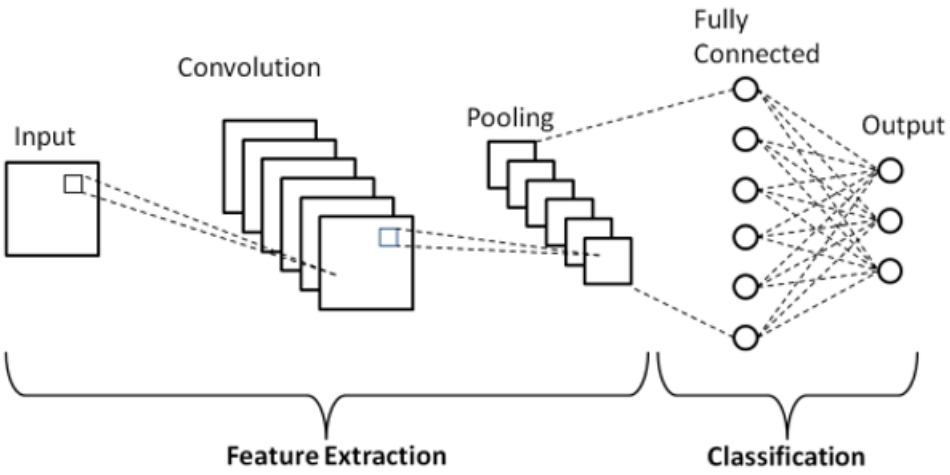
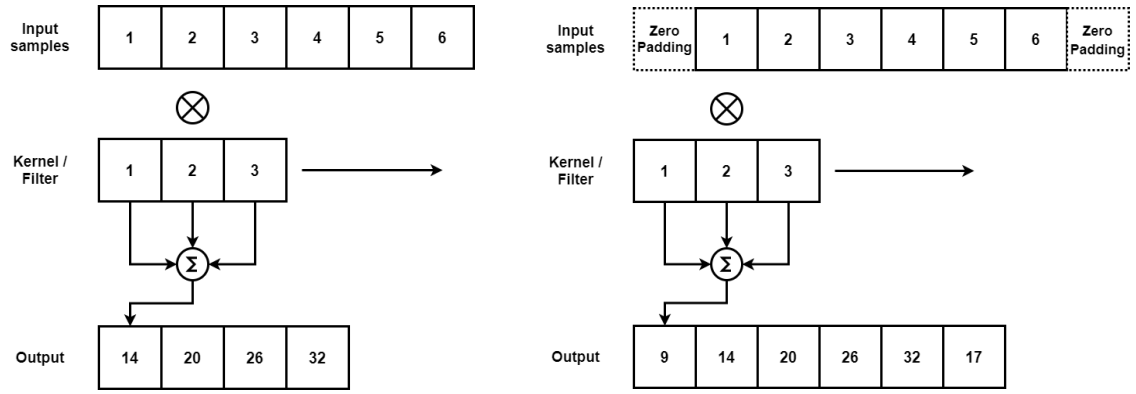


Figure 2.4: Schematic diagram of a basic convolutional neural network (CNN). Diagram from [16].



(a) No padding

(b) Including padding

Figure 2.5: Schematic illustration of 1D convolution operation.

In order to detect numerous features, multiple filters are often applied in a convolutional layer. This is depicted in Figure 2.6. The input of the convolutional layer consists of one or more channels with each the same length. The number of channels is defined by the amount of output channels from the preceding layer, or in the case of the first convolutional layer the number of channels is simply the amount of channels of the input. For example for RF data, this is often I and Q data, thus the input consist of two channels. All the filters are applied to the input channels, and the output of the convolutional layer is often commonly called feature maps. The number of channels in the output is equal to the amount of filters, and if padding is used then the output length is equal to the input length.

A CNN commonly consist of several subsequent layers to form a deep neural network. Apart from the convolutional layer other layers such as pooling layers are also included in a CNN. These other types of layers are further explained in Section 2.2.6.

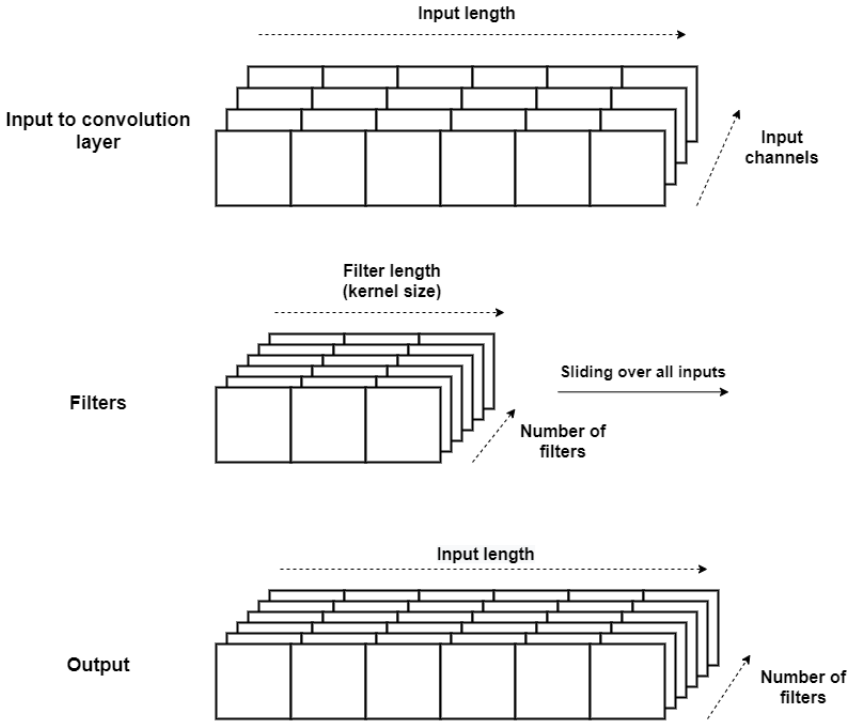


Figure 2.6: Illustration of the 1D convolution layer in a neural network. A convolution layer usually contains several filters, that are applied to the input channels, to determine the output.

2.2.3. Residual Neural Networks

A problem with deep neural networks is that they can be difficult to train. There are two main reasons for this difficulty, the vanishing/exploding gradient problem and the accuracy saturation problem.

- The vanishing/exploding gradient problem can be encountered during training of neural networks. It arises because during the backpropagation the gradients can be become vanishingly small or exploding to large values. Consequently, the weights cannot update to a new value, which may result in that the training of a neural network improves the model very slowly or not at all. The vanishing/exploding problem can largely be addressed by the correct selection of the activation function and the initialization scheme of the weights [17].
- The accuracy saturation problem occurs when increasing the depth of a neural network does not increase the accuracy, but instead the accuracy of the neural network saturates or even deteriorates. This makes it more difficult to train deep networks.

To address both problems residual learning is proposed [18] as a means to ease the training of deep learning models. A shortcut connection (also known as skip connection) is added to the network, such that layers can be skipped. An example of this principle is illustrated in Figure 2.7. In this case the shortcut connection performs the identity function, and the result of the stacked layers are added to this identity function. The identity function does not add extra parameters. The idea behind this shortcut connection is that it allows input data in the network to skip some layers, such that the input of a certain layer is not directly originating from its preceding layer, but instead may originate from a layer several layers preceding it. Since the published paper from 2015, Residual neural networks (also called ResNet) are used extensively to construct increasingly deeper convolutional neural networks. A ResNet based on the ideas from [18] won the ImageNet competition in 2015.

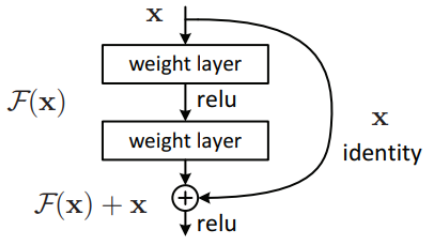


Figure 2.7: Building block for residual learning. Figure from [18]

2.2.4. Recurrent Neural Networks

Recurrent Neural Networks (RNN) are a class of neural networks commonly used in applications where data is sequential [13]. Examples of typical applications are speech-to-text and machine translation where an input sentence in a certain language is translated to a different language. In Figure 2.8 a comparison between a standard RNN and a feed-forward connected neural network (FFNN) is shown. The difference between a FFNN and a RNN is that a RNN has a feedback connection or loop in its hidden unit(s). In this way RNN shares parameters across several time steps. In a RNN the same input may produce different outputs depending on its preceding inputs, while a FFNN does not have this dependency.

A problem with standard RNNs is the gradient vanishing/exploding problem. To address this problem gated RNNs are introduced. These models contain long short-term memory (LSTM) units or gated recurrent units (GRU), and the idea is to create paths through time that have derivatives that neither vanish nor explode, according to [13].

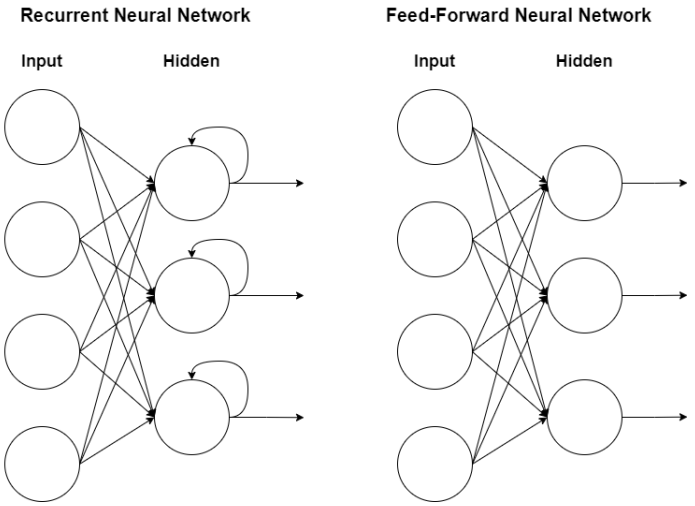


Figure 2.8: Comparison between a Recurrent Neural Network (RNN) and a Feed-Forward Neural Network (FFNN)

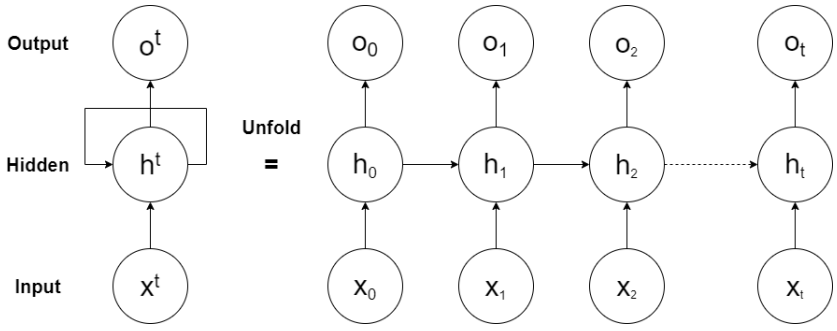


Figure 2.9: Standard Recurrent Neural Network and unfolded RNN.

2.2.5. Quantized Neural Networks

Neural networks are typically trained with single or double precision floating-point format. However, devices like FPGAs are not efficiently able to do floating-point calculations. To efficiently deploy neural networks to edge devices, such as microprocessors and FPGAs, the parameters and weights are commonly quantized. Such a network, called Quantized Neural Network (QNN), reduces the computational complexity and memory requirements, but could lead to a decrease in accuracy. It should be noted that a QNN is not a neural network architecture or topology, but rather the quantization of existing neural networks, such as CNNs.

There are different possible strategies and variants of QNNs. In post-training quantization the neural network is trained with floating point precision, and after training the parameters and weights are quantized. A possible advantage of this method is that training phase does not change, and makes it possible to optimize the model before making the model suitable for deploying in edge devices. Although, a close collaboration between training of the deep learning model and the implementation could also result in a better design. Floating point numbers have a much greater dynamic range than fixed point numbers. As a consequence of training a floating point model, the quantization of learned parameters and operations could be unsuitable or inefficient for the target implementation platform. Also, the weights and output values of different layers in the network could require different datatypes for various layers, which the hardware platform may not (efficiently) support. Instead of post-training quantization of a floating point model, a QNN can also be directly trained using quantized parameters. This is sometimes called quantization-aware training.

Binary/ternary Neural Networks An extreme case of a QNN is using ternary or binary weights, also known as TWN and BNN respectively. For a ternary neural networks, the weights are -1, 0 or 1, and for binary the weights are either -1/0 or 1. The main use of such networks is for FPGA acceleration. According to [19], the primary advantages of BNN over higher precision QNNs are efficient operations, normally the convolution operation requires a K (or K x K for 2D) element multiply-accumulate operation, but in BNN this can be implemented as a bitwise XNOR between two K bit vectors and a bitcount. A second advantage of binarizing weights and feature maps is the greatly reduced memory size. Existing accelerators are typically constrained in performance by a combination of on-chip storage space and off-chip memory bandwidth [19].

2.2.6. Building blocks

In the previous sections FFNNs, CNNs, ResNets and RNNs have been discussed. This section provides a overview of the common building blocks presented in these networks.

Fully Connected Layer A fully connected layer, also known as a dense layer. All the input nodes in a fully connected layer have a connection to its output nodes. The output of a fully connected layer is the result of the multiply and accumulate operation of all the input nodes and the corresponding weights, thus each output node depends on all the inputs nodes. A fully connected layer can be used to construct a feed-forward neural network. Also, one or more fully connected layers are commonly included in a CNN as the last layers to perform classifications based on the extracted features by the convolution layers.

Convolutional Layer A convolutional layer applies filters (also called kernels) to the input of the layer. The output of this layer is commonly called feature maps, since the convolutional layer is often included to detect features in the input. The filter size determine the amount of input values that are involved in the calculation of a output. In Section 2.2.2 a more thorough explanation of the convolutional layer is presented.

Pooling layer A pooling layer produces a summary of a number of inputs. For example, max pooling selects the max value within a defined range of inputs. Pooling layers are often used after a convolutional layer, such that further operations are performed with a summarised features, rather than the exact position of features. This principle is called invariance, because it makes the model more robust against small translations to the input. Besides max pooling, average pooling is also a popular pooling method. The outputs of average pooling are simply composed of the average of a number of nearby inputs.

Activation Function The activation function is a function at the output for a node, it is also known as a transfer function. The most basic activation function is the linear activation function, given by

$$f(x) = x \tag{Eq. 2.1}$$

with x is the input of the activation function and $f(x)$ the output of the node. Thus a linear activation function, simply passes its input to the output. Activation functions are often used to add non linearity to a node, such that the model can have nonlinear relations. A few examples of non linear activation functions:

- **Sigmoid and tanh**
  The sigmoid function or logistic function usually used in neural networks is given by:

$$f(x) = \frac{1}{1+e^{-x}} \tag{Eq. 2.2}$$

  The tanh function is given by:

$$f(x) = \tanh(x) \tag{Eq. 2.3}$$

  The range of the sigmoid is between (0, 1), which makes it useful for a probability output. The range of the tanh function is between (-1, 1). A problem with the sigmoid and tanh function is that the outputs saturate for negative input values to -1 or 0 respectively and saturate to 1 for positive input values. The consequence of this saturation is that in deep neural networks it is difficult to know what direction the parameters should change in order to optimize the cost function [20]. This problem is known as the vanishing gradient problem [20]. Since a trend in deep learning is to increase the depth of the network, sigmoid and tanh activation are not used in deep neural networks.

- **ReLU**
  The Rectified Linear Activation Unit (ReLU) activation function is a popular activation function, it simply clips all the negative values to zero, according to formula 2.4.

$$f(x) = \max(0, x) \tag{Eq. 2.4}$$

  The non linearity is accomplished by mapping all negative values to zero. The function is linear for positive values, which makes the optimization for. A limitation of ReLU, is that it a node becomes insensitive for weight updates, since the output of ReLU is zero for negative input values. This problem is called dying ReLU.

- **ReLU variants**
  There are many variants of the ReLU that attempt to solve the problems with ReLU activation. The Leaky ReLU (LReLU), introduced in 2013, contains the identity function for positive values. Unlike ReLU negative values don't map to zero, but LReLU introduces a negative slope to map negative values. As a result, LReLU keeps the weight update alive during the entire propagation process [20].

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{Eq. 2.5}$$

  The Scaled Exponential Linear Unit (SELU) is another variant. The formula is given by equation 2.6.

$$f(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1)) & \text{if } x \leq 0 \end{cases} \tag{Eq. 2.6}$$

  Where λ and α are fixed parameters, meaning they are not changed during training. When λ is 1 the activation function is called Exponential Linear Unit (ELU). Like ReLU activation it alleviates the vanishing gradient problem by using the identity function for positive values. The main advantage of (S)ELU over ReLU is that it improves the learning characteristics [20]. A disadvantage of (S)ELU is the computational complexity because of the exponential operation.

- **Softmax**
  The softmax function is often used as the activation function of the output nodes, to normalize the output of a network to a probability distribution over predicted output classes whose sums adds up to 1. The softmax function is given by equation 2.7. Where z_i is an element of the input vector z.

$$f(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{Eq. 2.7}$$

Dropout Dropout is a technique for addressing overfitting [21]. The idea of dropout is to randomly set nodes values to zero during training, and thus 'dropped'. A node is dropped with a probability of $1 - p_{\text{keep}}$, where p_{keep} is the probability that the node is present. During inference, the node remains always active, and the weights are multiplied by p_{keep}.

2.3. Modulation Classification

Modulation classification is the recognition of the modulation type of a particular signal. Examples of different modulation types are: On-Off Keying (OOK), Phase-Shift Keying (PSK) and Quadrature Amplitude Modulation (QAM). A modulation classifier can also be a sub-part within a broader function, an example of this is shown in Figure 2.10. In this case the RF spectrum consist of many different signals, and from this spectrum a single signal is filtered, before being presented as input to the classifier. This illustrates that many use cases within the RF domain require different functions to fullfill that use case. This is certainly the case with cognitive radio. Consequently, many of these functions have to run on a software defined radio platform. Hence, many of these functions will be implemented on an FPGA. Therefore, it is useful to run a deep learning model, for example for modulation classification, on the FPGA, because many of other functions are also performed there.

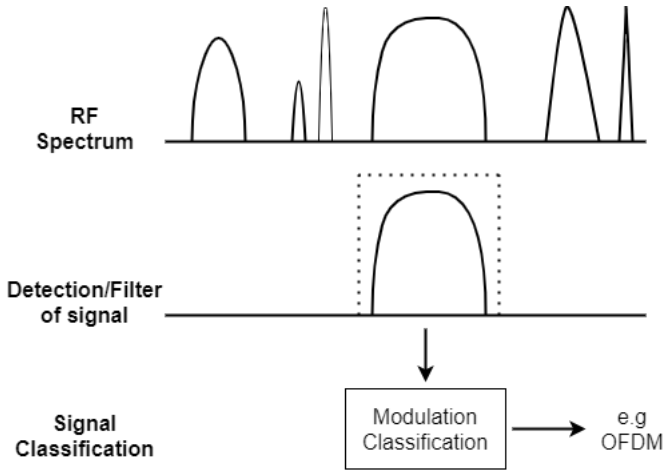


Figure 2.10: Example how modulation classification fits within a broader function.

2.3.1. Related Work

As described in Chapter 1, deep learning is a very popular field, in both academic and industry. Deep learning specifically for RF is relatively new. In 2017 one of the first papers related to deep learning for the physical layer was was published by O'Shea et al [22]. This paper gained academic interest and kicked off more research on deep learning for RF applications and, in particular modulation classification. In 2018 O'Shea et al published a paper [23] where they described the generation of a synthetic dataset containing labeled samples of 24 different analog and digital modulation types. This dataset is called DeepSig: RadioML (version 2018.01A). They trained 3 models using this dataset, a non deep learning model using XGBoost, a convolutional neural network and a residual neural network. The residual neural network performs best. The CNN has 257k trainable parameters, while the ResNet has 236k trainable parameters. In this paper the models are also tested with over-the-air data. The over-the-air dataset is generated by two SDRs, a transmitter and a receiver. The training and evaluation of the models with the over-the-air data are not done real-time, but afterwards on a GPU. Since the original paper by O'Shea et al, more research is done to the neural networks used for modulation classification. Some of these studies are listed below.

MCNet, Huynh-The et al [24] This paper proposes a cost-efficient convolutional neural network (CNN) for modulation classification. The accuracy is significantly higher than the CNN and residual neural network from [23]. For instance, the accuracy of MCNet is an improvement of 12.4% relative to the residual network [23] at a Signal to Noise ratio (SNR) of +10dB. Moreover, the amount of trainable parameters in MCNet is 142k, which is a reduction of approximately 40% compared to residual network from [23].

Automatic Modulation Classification using Recurrent Neural Networks [25] This paper states that CNN based models mainly considers the spatial correlation characteristics of RF signals rather than time domain related characteristics. Therefore, this article considers recurrent neural networks based on gated recurrent units (GRU) for modulation classification. The proposed model has approximately 38k parameters, and the accuracy at high SNR regime is 91%. Which is slightly less than [24]. However, the dataset used for the evaluation of the proposed model, is an older version of the DeepSig dataset, namely version 2016.10a. This version of this dataset contains 11 modulation types and an input vector size of 128 IQ samples. Hence, the results of this research cannot be compared fairly to the results from [23], [24]. Nevertheless, this research shows that RNNs could be a viable alternative to CNN based models for modulation classification.

Gated Recurrent Unit Neural Networks for Automatic Modulation Classification with Resource-Constrained End-Devices [7] This paper focuses on adapting the proposed GRU based RNN from [25], to obtain a network that is more suitable for resource-constrained end-devices. This paper focuses on the memory footprint of the model, which is largely determined by the number of parameters in the network.The adapted network has less parameters, namely approximately 18.4k, while the accuracy is limited by a few percent.

Deep Learning Models for Wireless Signal Classification with Distributed Low-Cost Spectrum Sensors [26] This paper demonstrates that simple LSTM models can achieve good accuracy on the DeepSig dataset, if the input data is formatted as amplitude and phase (polar coordinates) instead of directly feeding the model with IQ samples.

Robust Deep Radio Frequency Spectrum Learning for Future Wireless Communication Systems [27] This research explores "how to develop robust deep learning models that generalizes well on unseen data for different wireless communication scenarios in practice using RF data" [27]. They specifically evaluate the effect of SNR in the development of an RF deep learning model. An example of this effect of SNR on the deep learning model, can be seen in Figure 2.11. This figure shows that the model that is specifically trained for signals with an SNR of 0dB or 10dB is not well suited for signals with different SNR. Therefore, a deep learning model trained with a fixed SNR level cannot be used in many practical scenarios.

In this work deep learning models based on CNNs are used. They compare CNNs and RNNs with comparable accuracy, and came to the conclusion that the training time for CNNs is considerable shorter.

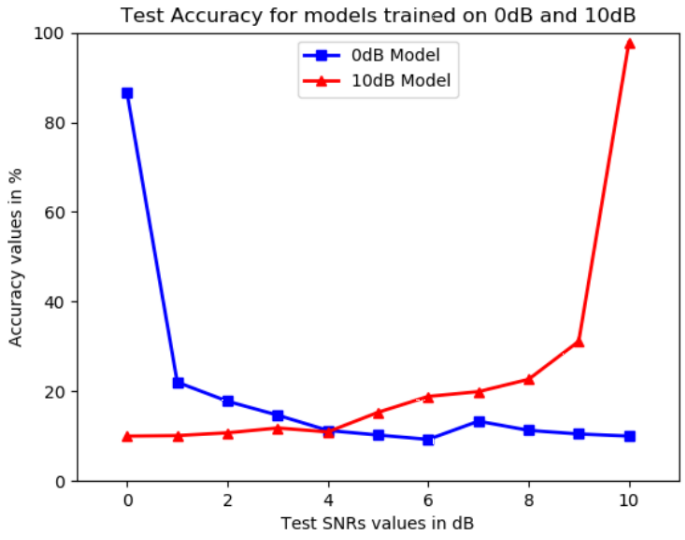


Figure 2.11: Accuracy versus SNR of two DL models trained on dataset with different SNRs (0dB or 10 dB). Figure from [27]

2.3.2. Main findings

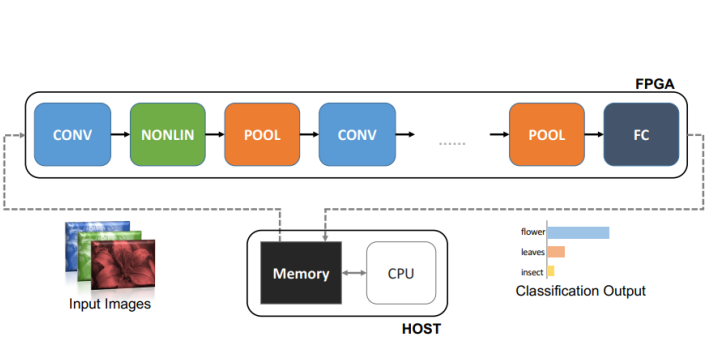
Based on the related research discussed in this thesis, it can be stated that modulation classification using deep learning techniques is still in early research phase, compared to more well developed deep learning research areas such as image classification. Several, different neural networks such as convolutional neural

networks and recurrent neural networks are being investigated. Both CNNs and RNNs show promising results, and there does not seem to be a clear trade-off yet as to whether RNNs or CNNs are more suitable for modulation classification, and what the consequences are of one network over the other. CNNs seems to be used more often for modulation classification, and also may be easier to train. Therefore, this thesis will focus on CNNs from now on.

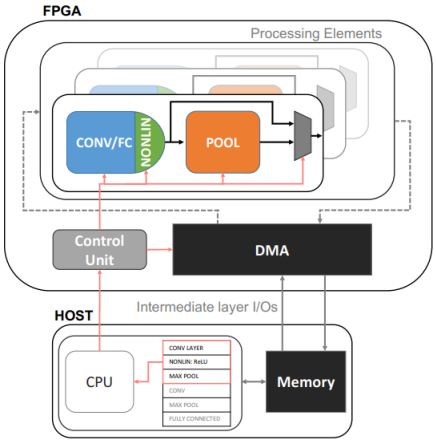
2.4. FPGA Deep Learning Acceleration

Acceleration of deep learning algorithms on FPGAs is gaining a lot of attention in both academic community and industry. Different strategies and frameworks exist for deploying deep learning models to an FPGA. In this section different papers and frameworks are discussed.

A good overview of different CNN-to-FPGA toolflows is given by I. Verieris et al [28]. These tools can be taxomized in two main catagories: streaming architecture and single computation engines. In Figure 2.12 the two categories are depicted.



(a) Example of a streaming architecture accelerator



(b) Example of a single computation engine accelerator

Figure 2.12: Two different FPGA CNN accelerator architectures, as described by I. Verieris et al [28]. Figures are from [28]

Streaming Architectures Typically consist of one distinct hardware block for each layer of the target CNN, where each block is optimized separately to exploit the parallelism in its layer. The data is streamed through the architecture and as a result this design exploits parallelism between layers by means of pipelining. The downside of a streaming architecture is that the hardware is not generic, meaning that the hardware is different for each neural network. An advantage is that the hardware can be specifically optimized for a certain neural network and use case.

Single Computation Engines This design favours flexibility over customization. The control of hardware and scheduling of operations is performed by software. A neural network is broken down into instructions, which can be executed by the computation engine. The fixed architectural template can be scaled to the target FPGA and CNN. The advantage of this design approach is that the architecture is not specific to a certain CNN, which means that the hardware does not have to change for a different network. If the computation engine supports all the operations of a certain CNN, the computation engine can execute the CNN. The downside of this design strategy is that the one-size-fits-all approach can lead to a high variability in the achieved performance accross CNNs with different workloads characteristics [28].

2.4.1. Comparison of toolflows

The compared CNN-to-FPGA toolflows by I. Venieris et al [28] are depicted in Table 2.1. Most of the tools are not publicly available for use (open source or commercial). Only DeepBurning, Haddoc2, DNNWEAVER and FINN are open source projects. Next these tools are summarized.

- **DeepBurning**[29]
  DeepBurning is an end-to-end deep learning acceleration tool. It consist of a library of building blocks which implement certain common neural network operations. Dependent on the neural network and user constraints, DeepBurning generates a custom accelerator, based on a streaming architecture. A fully expanded neural network might not be possible, because of FPGA resource constraints or user specified area constraints. DeepBurning enables spatial and temporal folding, in order to compress the neural network into hardware. At the moment DeepBurning is not completely publicly available. Only a few pre-build examples are available. Furthermore, it is no longer actively maintained (last Github activity in 2019).

- **Haddoc2**[30]
  Haddoc2 is similar to DeepBurning, in that the tool generates a streaming architecture accelerator. Each layer in the target CNN is mapped to a hardware stage, in which each processing entity gets private resources, in order to maximize parallelism. Therefore, Haddoc2 does not require external memory. Haddoc2 does not support time sharing of resources in order to limit the amount of resources that are used. Furthermore, the neural network size that can be accelerated using Haddoc2 is limited, because Haddoc2 does not support partial unrolling [28]. Haddoc2 is not actively maintained anymore (last Github activity in 2018).

- **DNNWEAVER** [31]
  DNNWeaver is based on the computation unit architecture. Therefore, DNNWeaver translates a high-level neural network to its Instruction Set Architecture (ISA). The accelerator is based on parametrised architectural template, such that it can matched to the target neural network and FPGA. DNNWeaver is not actively maintained anymore (last Github activity in 2019).

- **FINN**[32], [33]
  generates a data-driven streaming architecture accelerator. It is mainly targeted at very low precision quantized neural networks, such as binary neural networks (BNN). FINN expresses the BNN compute operations as matrix-vector operations followed by thresholding. The Matrix-Vector-Threshold Unit (MVTU) forms the computational core of accelerator. FINN is actively maintained by Xilinx Labs.

All these tools are targeted at the computer vision domain. This is evidenced by the fact that the operations are only available or optimized for 2D data. From the tools tabulated in Table 2.1, FINN the only tool that is publicly available for use and is still actively maintained.

Tooflow name	Publicly available for use
fpgaConvNet	No
DeepBurning	Yes (https://labfor.github.io/)
Angel-Eye	No
ALAMO	No
Haddoc2	Yes (https://github.com/DreamIP/haddoc2)
DNNWEAVER	Yes (http://dnnweaver.org)
Caffeine	No
AutoCodeGen	No
FINN	Yes (https://xilinx.github.io/finn/)
FP-DNN	No
Snowflake	No
SysArrayAccel	No
FFTCodeGen	No

Table 2.1: Evaluated toolflows in [28]

In addition to the discussed tools in Table 2.1, there are more existing tools and demonstrations of Deep learning acceleration on FPGA, such as Vitis AI and hls4ml.

Vitis AI Vitis AI is generic tool for AI inference, specifically for Xilinx hardware platforms. Unlike FINN which is an experimental framework from Xilinx Research Labs, Vitis AI is an official Xilinx product. Vitis AI is meant to be a generic tool, such that the barrier to use FPGAs for deep learning acceleration is lowered.

Vitis AI is based on a computation unit architecture, therefore the neural network is translated to instructions which can be executed on the computation unit. In Vitis AI this computation unit is called the Deep Learning Processing Unit (DPU). This DPU can be tailored to different applications and FPGAs. Like the already discussed tools, Vitis AI is fully focused on the vision domain.

Hls4ml Hls4ml [34] is developed for high throughput and low latency requirements. The motivation for the development of hls4ml stems from the high data rate involved in the CERN Large Hadron Collider (LHC) for proton-proton collisions. Collisions occur every 25ns and the particle detectors around the LHC ring produce tens of terabtyes of data every second [35]. The trigger to decide whether a discrete collision should be kept for further analysis or discarded has strict latency requirements because of the high data rate. The decisions taken by the trigger are based on deep learning. The accelerator as generated by the hls4ml framework is based on the streaming architecture.

2.4.2. Main findings toolflows

Of all the toolflows discussed in the previous section, only three tools are still actively maintained and publicly available for use. These tools are Vitis AI, hls4ml and FINN.

As explained in Chapter 1 restrictions in the implementation of DL models for FPGA deployment have influence on the RF use case development and DL model development. FINN is mainly targeted at very low precision neural networks, such as binarized neural networks. Therefore, this aspect of FINN is a restriction, which is not desirable in this master thesis project.

Vitis AI and hls4ml appear to be promising tools for deploying CNNs to FPGAs. In Chapter 3, both Vitis AI and hls4ml will be tested to assess their suitability for RF deep learing.

2.4.3. Related work

The amount of demonstrations of modulation classification on FPGA based SDR is very limited. In this section two demonstrations are discussed.

Real-time Automatic Modulation Classification using RFSoC This paper [36] specifically focuses on the real-time aspect of implementing deep learning models using FPGA based radio platforms. The authors train a ternary-weights convolutional neural network that is capable of producing high speed real-time classifications on the signal modulation classification use case. An overview of the FPGA implementation is shown in Figure 2.13.

The IQ samples produced by the RF ADC are directed to the scheduler. The scheduler sends two sets of IQ samples to the CNN accelerator, and the scheduler also receives the predictions upon completion. The IQ sample rate is 500 MHz, and the CNN processes 2 sets of IQ samples with a clock rate of 250 MHz. The prediction is completes when 1024 IQ samples have been provided, after which the next prediction immediately starts. Therefore, the throughput is 488k ($\frac{500MHz}{1024}$) classifications per second, and the latency is approximately 2000 clock cycles at 250 MHz, thus 8 μs.

This paper clearly explains that throughput and latency are two important aspects of a real-time modulation classifier. Therefore, in this master thesis, emphasis is placed on the throughput and latency as such aspects that are essential to an FPGA-based deep learning solution for RF applications. A major drawback of the solution presented in [36] is that only ternary weights are supported. This already constitutes a limitation when developing the RF use case and training deep learning models, and as discussed in Chapter 1, such limitations are undesirable in the context of this master thesis.

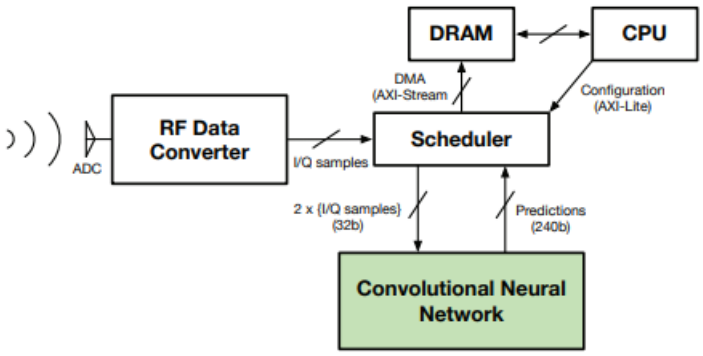


Figure 2.13: Block schematic of FPGA CNN accelerator integrated on FPGA based SDR. Figure from [36].

Implementation of Deep learning-Based Automatic Modulation Classifier on FPGA SDR Platform This paper [37] demonstrates the implementation of a modulation classifier (in this paper called Automatic Modulation Recognition) running on an FPGA based software defined radio system. The authors implement a convolutional autoencoder (CAE) neural network using Vivado HLS. This IP is integrated in an RFNoC block, as shown Figure 2.14. RFNoC enables a modular design, in which functions can be implemented as an RFNoC block. These blocks are connected to a crossbar, so that data can be exchanged between the blocks. RFNoC also enables data to be passed between the FPGA and CPU. GNUradio is used to run the application. In Figure 2.15 the GNUradio flow diagram is shown.

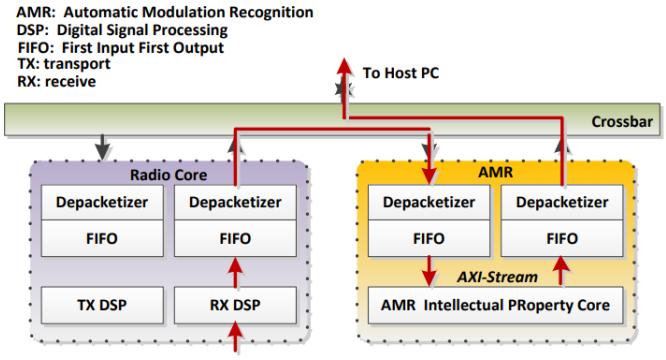


Figure 2.14: Data flow of RFNoC based modulation classification application. RFNoC enables blocks to exchange data between each other by means of a crossbar. In this figure the radio core and the automatic modulation recognition (AMR) are two RFNoC blocks. Figure from [37].

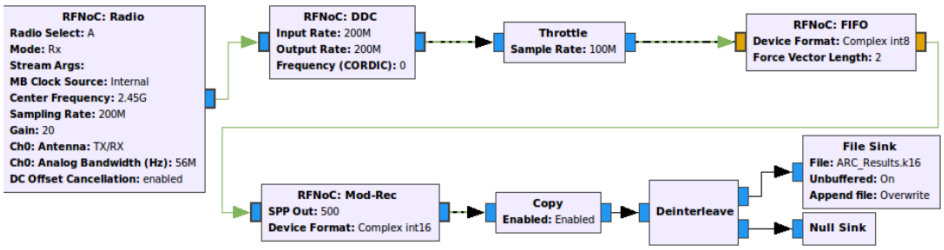


Figure 2.15: GNU radio flow diagram of automatic modulation recognition. The RFNoC blocks can be connected together in order to create an RF application. Figure from [37].

2.5. Conclusion

Existing deep learning models are often based on Convolutional Neural Networks (CNN) or Recurrent Neural Networks (RNN). Both models achieve promising results and there is still no clear consensus as to which models are best suited for modulation classification. However, convolutional neural networks are used more often and may be easier to train. In this thesis CNNs are used for the modulation classification use case.

A reasonable amount of research has already been done into the use of FPGAs for deep learning acceleration, but many of these studies are fully focused on vision tasks. Therefore it is useful to test some of these solutions for RF deep learning applications. There are mainly two different FPGA architectures for deep learning acceleration.

- Streaming architecture in which the data samples are 'streamed' through the neural network. The accelerator is specific to a particular network. This means that the accelerator has to change to run a different network.
- Computation unit architecture which executes instructions. The resulting accelerator does not have to change for different networks, as long as these networks can be converted to compatible instructions.

In the next chapter hls4ml and Vitis AI are evaluated whether they are suitable for RF deep learning. Both hls4ml and Vitis AI are publicly available for use, and also actively maintained. A number of aspects are of particular importance for FPGA based DL acceleration for RF applications. First, the latency and throughput are important because of the communication context. Second, the use of FPGA resources determines whether one or more neural networks can be deployed on an FPGA to run in coexistence with other DSP functions in an FPGA-based SDR.

3

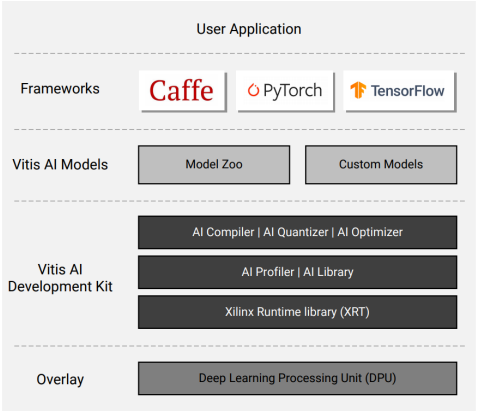
Preliminary Experiments

In the previous chapter two fundamental different FPGA deep learning accelerator architectures were reviewed. Furthermore, several tools were discussed. In this chapter, a more in depth review is performed of two promising tools, Vitis AI [38] and hls4ml [39]. In order to investigate whether these tools are suitable for deploying deep learning models for use in the RF domain on an FPGA, models trained on the DeepSig dataset are used for evaluation.

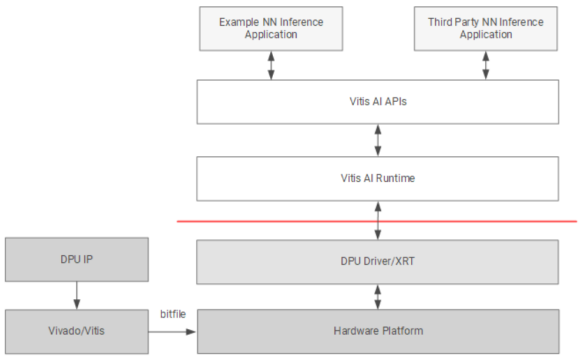
This chapter is structured as follows. Section 3.1 discusses the Vitis AI tool such as the workflow of using this tool. Subsequently, Section 3.2 discusses the hls4ml tool and evaluation of the this tool.

3.1. Vitis AI

The development environment for deploying deep learning models to an FPGA using Vitis AI is depicted in Figure 3.1(a). The first step in the development flow is to train a deep learning model using one of the most common frameworks. There are a number of ready-made models available. The next step is to quantize this floating point model, using the Vitis AI quantizer. Vitis AI uses INT8 datatypes for the quantize step. Subsequently, this quantized model can be compiled into instructions for the Deep Learning Processing Unit (DPU). Next, this model can be deployed to an FPGA, Figure 3.1(b) shows this process. First of all, the DPU IP should be implemented on the hardware platform. The drivers and APIs are all implemented on the host CPU. The API contains a number of C++/Python functions to execute the compiled deep learning model on the DPU.



(a) The Vitis AI stack. Figure form [40].



(b) Vitis AI development flow. Figure from [41]

Figure 3.1

3.1.1. Evaluation

To evaluate the suitability of Vitis AI for RF applications, a ResNet model, based on the model proposed by O'shea in 2018 [23], will be deployed to an FPGA using Vitis AI. The Xilinx ZCU104 evaluation board is used to

perform these experiments. Because DPU in Vitis AI is based on the computation unit architecture, the FPGA only has to be configured once using a bitstream. Vitis AI provides board images in which the FPGA is already configured, and also the complete software stack for the ARM processor is pre-installed. In this case version v2020.1-v1.2.0 is used.

The ResNet model developed for RF data uses 1D operations. Since Vitis AI has been developed for vision applications, Vitis AI is currently only supporting 2D operations. Therefore, this ResNet model has been converted to a 2D model. Running this model using Vitis AI results in approximately 2k classifications per second. Since the deep learning model uses 1024 samples for a prediction, a maximum continuous sample rate of 2 mega samples per second (MSPS) would be sustainable in a real-time application. It depends on the context of a certain use case whether this is sufficient. In any case, there are numerous applications within the RF domain that require a higher throughput. Another smaller CNN with approximately 10k parameters is tested and achieves a throughput of approximately 4k classifications per second. These results from Vitis AI show a consequence of a computational unit architecture, namely being able to support a wide variety of different neural networks, but the achieved performance varies between networks. The experiments indicate that Vitis AI would be mainly useful in applications that require a large neural network, such as ResNet, and where very low latency and/or high throughput is not required. Moreover, Vitis AI is not well suitable for RF data, since Vitis AI uses 8 bit numbers. RF ADCs are often have a resolution of 12 to 16 bit. During these tests, it was observed that the accuracy of a quantized model is significantly lower (a drop of approximately 30%) than the floating point model. This result shows that Vitis AI is currently not suitable for RF deep learning, however given the rapid developments, Vitis AI could become a suitable means for deploying RF deep learning models to an FPGA.

Figure 3.2 shows an example system using Vitis AI. In this case images from a camera are processed by the FPGA, and send to memory by a DMA controller. The data to the DPU is controlled by a CPU, and thus data cannot be feed directly to the DPU. This is a limitation of Vitis AI, and deep learning accelerators based on a computation unit architecture. In an RF application it might be beneficial to being able to directly feed data to a neural network IP, as is possible with a streaming architecture.

It is concluded that Vitis AI does not provide a fully adequate tool for modulation classification (and RF applications in general). In particular, the throughput and latency are limited and the 8-bit data type also forms a limitation, as RF ADCs are often 12 bits or more. However, Vitis AI can be used to accelerator large neural networks as demonstrated in this section with a ResNet model. Therefore Vitis AI can be useful in applications where large neural networks are used and which do not require a high throughput or low latency.

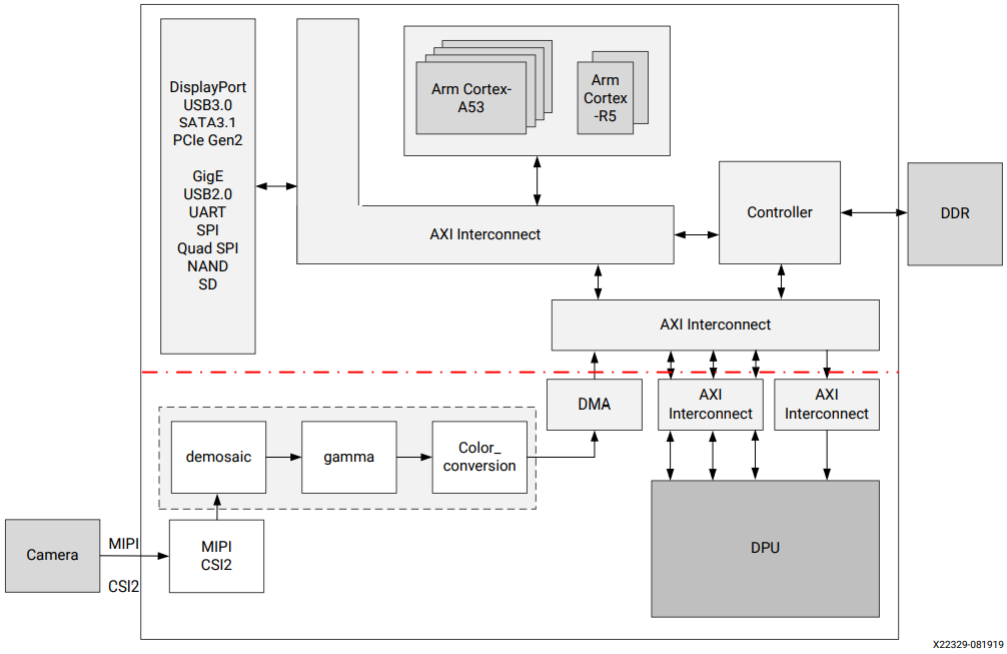


Figure 3.2: Example system with Vitis AI. Figure from [41]

3.2. hls4ml

The workflow of hls4ml is depicted in Figure 3.3. First, a model is trained in one of the most common frameworks. This can be a floating point model, but also a quantized model using the fixed point datatype. After this step, the model can be translated into an HLS project that can be synthesized and implemented to run on an FPGA [34]. The HLS project can be configured, such as the precision of the fixed point data types in various layers of the implemented model.

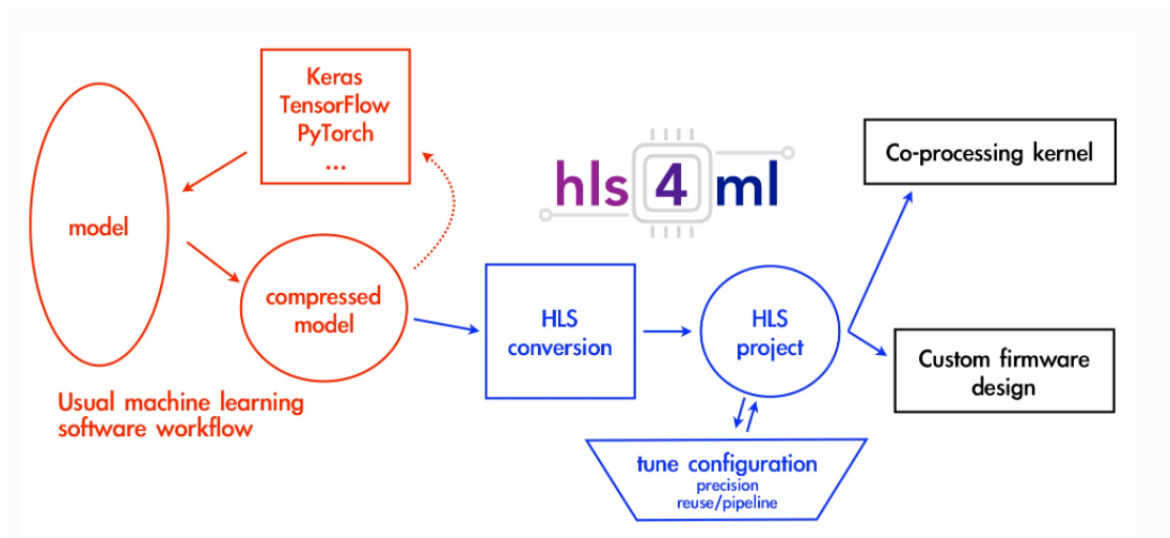


Figure 3.3: Hls4ml development flow. Figure from [34]

3.2.1. Evaluation

Hls4ml uses Vivado HLS to compile and synthesize the HLS code. An attempt has been made to test hls4ml with a CNN, the model used is shown in Table 3.1. This neural network is very small, since it contains only 1692 trainable parameters. However, even such a small network cannot be successfully synthesized using a generated HLS project by hls4ml. This is due to the way hls4ml is designed.

Layer	Output Shape	Parameters
Conv 1d	[1024 4]	28
Max Pooling 1d	[256 4]	
Conv 1d	[256 4]	52
Max Pooling 1d	[64 4]	
Conv 1d	[64 4]	52
Max Pooling 1d	[16 4]	
Dense	24	1560

Table 3.1: Small CNN to test hls4ml. This network consist of 1692 trainable parameters in total.

Listing 3.1: hls4ml top function

```
1 void hls4ml_top_function(input_t input[1024*2], result_t output[24])
```

Listing 3.1 shows the top function of the hls4ml HLS project. All the inputs are passed at once to this top function, in this case the input size is 2048 samples (1024 I and Q samples). The top function is pipelined by default, therefore all the layers in the neural network are pipelined. Consequently, all the loops in the design are automatically unrolled. Since, all the input samples are processed at once, a for-loop iterates over all these input samples to perform the convolution operation. Hence, unrolling all the for-loops results in an

enormous amount of hardware, which causes Vivado HLS to throw an error. Nevertheless, the fully pipelined accelerator as generated by hls4ml, achieves a high throughput and low latency for smaller networks. For example, suppose the hls4ml top function as depicted in 3.1 can accept a new set of inputs every clock cycle, then the throughput is simply equal to the clock frequency the accelerator is running on.

Another limitation of the generated HLS project by hls4ml, is that each layer in the neural network starts sequentially, such that a layer starts executing whenever its previous layer is finished processing all its inputs. The implication is that all the outputs of every layer must be stored, which can use significant memory. To give an example, the first convolution layer from Table 3.1 has 4 filters and 1024 input samples for 2 channels (IQ data). This results in an output shape 1024 by 4, thus in total 4096 values. This can grow even larger if more filters are applied.

In order to make the design successfully synthesize for the model shown in Table 3.1, all the pipeline pragmas as well as most of the partitioning pragmas are removed. This results in an expected throughput in the order of 250 classifications per second. Although the throughput could be increased a bit further by optimizing, it is still a factor 8 lower compared to the throughput of the much larger ResNet model using Vitis AI. However, the fully pipelined streaming design as generated by hls4ml still has potential advantages over Vitis AI, which are summarised below.

- Potentially high throughput and low latency, but this requires the design to be actually synthesizable and also that the design fits on an FPGA in terms of resources.
- Possible to feed data from an ADC into an FPGA and then directly into the CNN accelerator. As can be seen in Figure 3.2, with Vitis AI data can be pre-processed in the FPGA, but afterwards the data is moved to memory by a DMA controller.
- Hls4ml is open source, thus it is possible to make changes to hls4ml and use it as a base for further development. While Vitis AI is not open source and is meant to be a tool which just works and does not allow much design exploration.

Like Vitis AI, hls4ml also does not provide a fully adequate tool for modulation classification (and RF applications in general). The main shortcoming of hls4ml is that it is only suitable for very small neural networks which are not sufficiently capable of modulation classification.

Based on the experiments performed in this chapter it is decided that a custom accelerator will be designed. This custom accelerator should be able to reach a high throughput and low latency, and also be suitable for reasonable sized neural networks capable of modulation classification.

3.3. Conclusion

In this preliminary experiments chapter, two FGPA deep learning accelerators are evaluated for their suitability for RF deep learning. Firstly, Vitis AI is tested and secondly hls4ml is tested. Vitis AI is based on a computation unit architecture, and hls4ml is based on a streaming architecture. The preliminary tests show that Vitis AI can be useful for TNO in certain cases. These are cases where large neural networks are required and a high throughput/low latency is not a requirement. It is important to mention that vitis AI currently only supports 8-bit data input. This forms a limitation for RF deep learning, since RF ADCs are often 12 to 16 bit. Perhaps this will change in the future, because of the rapid developments in FPGA deep learning accelerators.

Tests with hls4ml proves that hls4ml can only be used for very small networks, because of this limitation hls4ml is not useful for TNO. However, its fully pipelined streaming architecture still has advantages, such as a high throughput and low latency. Therefore, a custom CNN FPGA accelerator will be developed based on such a streaming architecture. This accelerator should support larger networks than hls4ml currently supports, and provide a high throughput and low latency accelerator.

4

Implementation

The previous chapters showed that there are various developments ongoing regarding FPGA deep learning acceleration. However, the related research and preliminary experiments have shown that these existing solutions are not fully adequate for FPGA RF deep learning acceleration.

Examples of shortcomings are that certain developments are specifically targeted at the vision domain and are, therefore, not optimized for the RF domain. In this master thesis project we designed an implementation for an FPGA deep learning accelerator, which specifically takes RF applications into account. This implementation should facilitate users to implement deep learning technology in software defined radio systems. And as a result, enable users to do applied research in deep learning for RF use cases.

This chapter is structured as follows. Section 4.1 presents the main design goals of the implementation. Section 4.2 describes the process to deploy a neural network to an FPGA. Section 4.3 describes the actual implementation of the accelerator. Section 4.4 provides the conclusion of this chapter.

4.1. Main design goals and specifications

The first step in designing a custom solution is to determine the main design goals and specifications. These design goals are motivated by the performed experiments in Chapter 3 and also from the workflow presented in Chapter 1.

Streaming architecture As described in the preliminary experiments section, it is beneficial to have an accelerator based on a streaming architecture when the use case requires high throughput. This is because a streaming architecture can be made specifically for a particular network and optimizations are possible to meet specific requirements such as throughput and latency. Another advantage of a streaming architecture is that this architecture does not necessarily require control. This means that the input values of the neural network can directly be presented to an FPGA IP containing the neural network accelerator and the FPGA IP produces an output. This may be advantageous when several pre-and post-processing operations are be required. These operations may be directly related to deep learning, such as the normalization of input values coming from an ADC. In addition, other operations not directly related to deep learning may be needed, such as a channelizer. A channelizer can be used to split the RF spectrum into different bands, such that these bands, which can contain different signals, can be processed separately. By performing all the operations on the FPGA it is possible to design a high throughput accelerator that directly accepts data from an RF ADC, and also pre-and post-processing functions can be directly connected to the FPGA accelerator IP.

In the case of a computation unit architecture, the neural network is implemented by executing a series of instructions. This architecture requires control in the form of scheduling of instructions and data movement. This control is often implemented in CPUs instead of FPGA. In conclusion, an streaming architecture allows for an FPGA only implementation.

Process CNN on a sample-by-sample basis In order meet the high throughput and low latency requirement and simultaneously low resources, it is essential to process a CNN on a sample-by-sample basis. A streaming architecture allows for the design of an accelerator where each layer in the CNN can start execution with only a portion of all the input data. In this master thesis, this principle is called processing on a sample-by-sample

basis. In standard CPU or GPU implementations, the neural network is executed in a sequential way. The subsequent layer only executes when all the outputs are available from the current layer. CPUs/GPUs often execute a neural network by processing a large number of batches at once. In an FPGA design where processing is done on a sample-by-sample basis, each layer can start executing as soon as there is data available. The design philosophy behind this is to spread the work over time as much as possible.

Furthermore, by only storing the minimum values in the neural network that are needed for the current operation, the memory requirements are drastically reduced. This is beneficial for FPGAs since they are limited by on-chip memory and off-chip memory is slow compared to on-chip memory. A convolutional layer produces feature maps as output. The amount of maps produced by a convolutional layer is equal to the number of kernels/filters in that layer. And the size is equal to the input size. In the modulation classification use case, the input to the first convolutional layer has a length of 1024 samples per feature map. The required storage space in FPGAs to store the complete feature maps is significant. This also scales with larger networks with more filters per layer or more layers.

The design must be fully pipelined, such that each clock cycle a new input sample is accepted by the FPGA accelerator. This allows an RF ADC to directly interface with the FPGA accelerator and achieve high throughput and low latency.

Implement generic building blocks of common operations in CNNs CNNs consist of a number of building blocks. The most common building blocks are described in Section 2.2.6. The principle of the building blocks is that it is easy to construct a CNN simply by configuring the building blocks and connecting them. In order to construct a basic CNN, the design should support the following building blocks:

- 1D Convolution Layer
- 1D Pooling Layer
- Dense Layer
- Activation Function

These building blocks are not made specifically for a certain arithmetic precision but instead use fixed point datatype with variable width. This allows the user to choose the precision of the weights and operations. As a result, a trade-off can be made between, for example, resources and accuracy. While binary and ternary neural networks show promising results in research, the limitation of the specific targeting of BNN/TNN is not desired in this master's thesis. Still, the specific support for BNN and TNN can be added at a later stage. It should be noted that fixed point operations naturally support operations with binary or ternary numbers, but are not optimized for this.

Implement using HLS Hardware designs are traditionally described using a hardware description language (HDL), such as VHDL and Verilog. High level Synthesis (HLS) tools increase the abstraction level, which can reduce the development time. Also, making changes in the HLS code is generally easier, for example chaining the degree of parallelism or memory access patterns, can be controlled in Vivado HLS through pragmas. Also, Vivado HLS enables a user to specify a target clock frequency, such that Vivado HLS can automatically insert registers to shorten the critical path. Although the design is primarily aimed at achieving high throughput, HLS also allows for rapid design changes to meet other specifications. As a result, the use of HLS in the workflow makes it possible to tailor the design specifically to certain specifications depending on the requirements of the RF use case or deep learning models. Still, it remains a great effort to truly optimize Vivado HLS code in order to obtain a good design.

4.2. Workflow of implementing neural networks on an FPGA

As described in paragraph 1.2, the workflow of the application of deep learning technology is divided into three aspects which are listed below.

- RF use case development
- Deep learning development
- Implementation

The scope of this thesis is mainly the implementation aspect of the deep learning, but as already mentioned in Section 1.2 these three aspects are not independent of each other. This dependence is also apparent from the workflow of deploying neural networks on an FPGA as depicted in Figure 4.1. The training of a neural network and the implementation of the neural network on an FPGA are part of the same workflow. Thus, these two aspects should be compatible with each other.

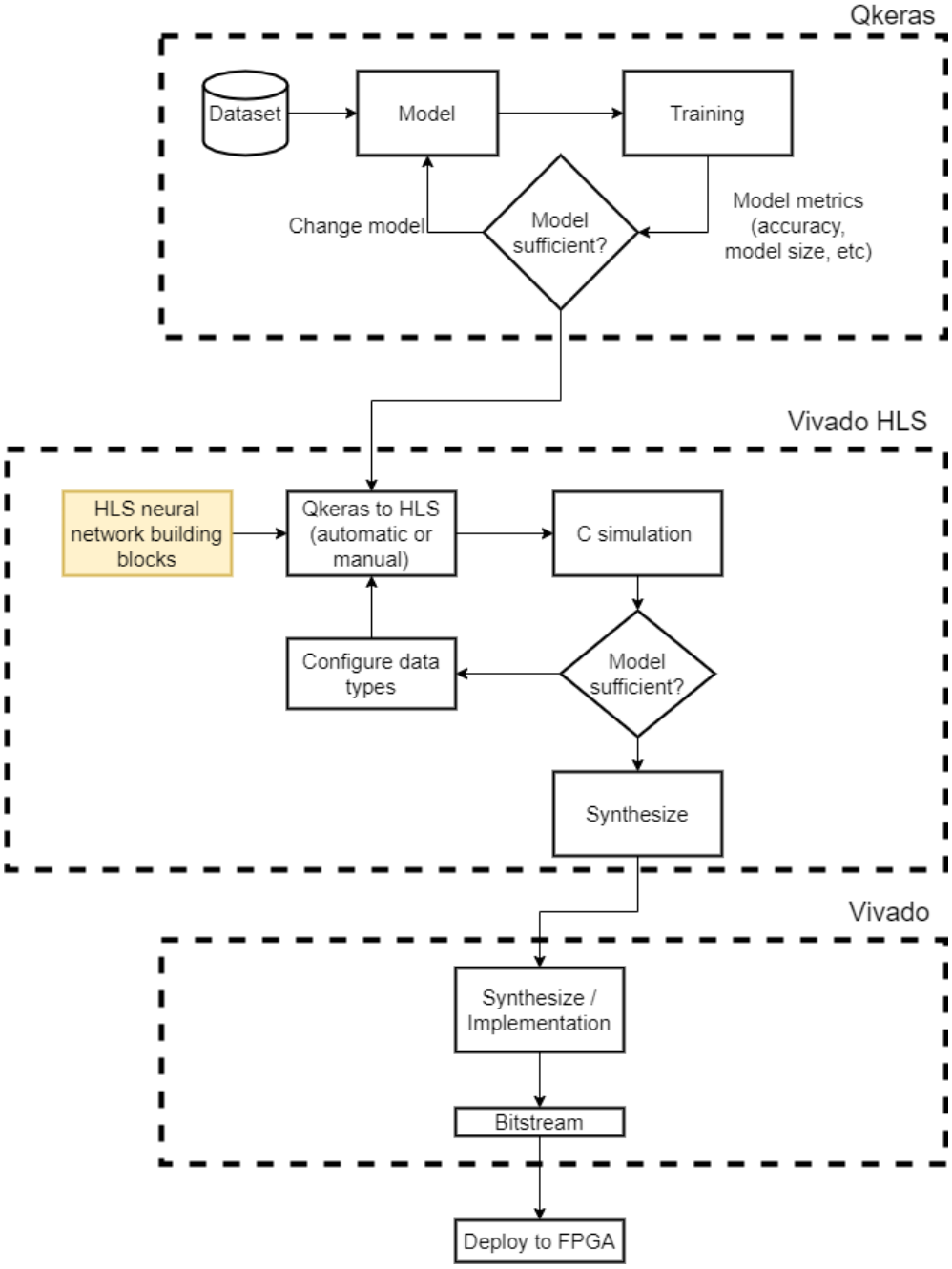


Figure 4.1: High level overview, mapping NN to FPGA

The first step in this workflow is to train the neural network, this can be done on a regular desktop system. As described in Section 4.1 the implementation will support QNN with fixed point datatypes. Qkeras [35] is an extension to Keras [42] to enable quantization aware training of neural networks. A nice property of QKeras is that the various building blocks are a drop in replacement for the floating point building blocks in Keras. This makes it easy to do a direct comparison between the floating point model and fixed point model. Training and evaluation of the model can be done entirely on CPU/GPU platforms. If the model proves to be

sufficiently suitable, the next step is to implement the model on an FPGA.

The Qkeras model is converted to the designed streaming architecture accelerator. In this thesis the process of converting the model to an HLS description is not yet automated. Therefore, the neural network is built manually using the HLS building blocks designed in this master thesis. All the learned parameters, the weights and biases, are automatically converted to a C++ header file which can be inserted in the HLS project.

The data types should also be correctly chosen. The amount of bits for the weights and biases are simply the same as the QKeras model. The number of bits used for the operations can be chosen for each layer. Section 5 elaborates further on the process of selecting this number of bits. Next, the HLS implementation can be simulated using the Vivado HLS C++ simulator. By using the same dataset for both the QKeras model and the FPGA accelerator implementation can be compared. It could be that the QKeras model is used as a reference and that the implementation should get exactly the same results. But it may also be the case that the number of bits are chosen to make a trade off between resources and accuracy (or other classification evaluation metrics). Next, the design can be synthesized using Vivado HLS. Finally, the IP can be integrated in a Vivado project, and after synthesizing and implementation, the resulting bitstream can be used to deploy the accelerator to an FPGA.

4.3. HLS implementation of building blocks

In this section the HLS neural network building blocks are designed. These building blocks are used to construct neural networks by connecting the building blocks together and configure each building block.

4.3.1. 1D Convolution

The 1D convolution operation has been discussed in Section 2.2.2. In that section the direct convolution method is discussed. This method is based on a sliding window, in which multiple filters slide over the input data and at each slide position the multiply-and-accumulate(MACC) operation is performed between the input data values and the corresponding filter values. Apart from the direct convolution, other techniques for calculating convolution are for example FFT based or matrix multiplication. These alternative methods, are mainly targeted at HPC systems with large amount of memory and bandwidth [43], [44] available. Therefore, in this implementation the direct convolution method is used.

Listing 4.1: 1D convolution operation pseudo code

```
for(l=0; l<input length; l++){
    for (n=0; n<number of filters; n++){
        for(k=0; k<number of input_channels; k++){
            for(m=0; m<filter size; m++){
                output[l][n] = output[l][n] + input_sample[l+m][k] * filter[k][n][m]
            }
        }
    }
}
```

In code listing 4.1 pseudo code of the 1D direct convolution operation is shown. The range of the 4 nested for loops follows from Figure 2.6 from Section 2.2.2. The notation is as follows: L is equal to the length of the input to the convolution layer; N is equal to the number of filters; K is equal to the number of input channels and M is equal to the filter/kernel size. The filters slide over the input sequence, and every output value only depends on the input values that overlap with the position of the filters. This means that the outer for-loop can be omitted, and the input values can be provided one by one to the convolution layer. For each output value, M input samples have to be stored. The design of this is shown in Figure 4.2.

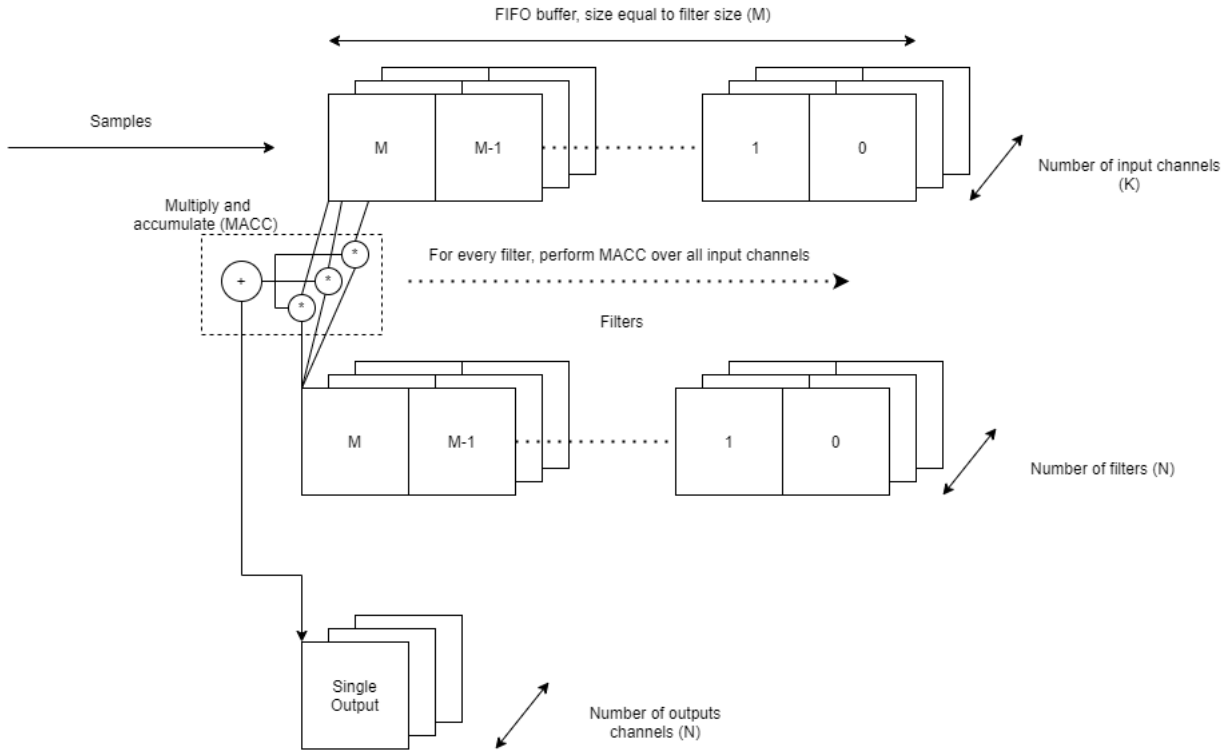


Figure 4.2: 1D convolution operation, streaming architecture

The input values are stored in K FIFO buffers, with each a size equal to the filter size (M). As a result, every time a new input (with K channels) arrives in the FIFO buffer, a single output value can be calculated. And this output value can be directly passed to the next layer. This principle is programmed in HLS C++ code, a simplified version is shown in listing 4.2. This function receives a single input value, the weights and bias; and produces a single output. The FIFO buffer is declared at line 8 as a static type, because the buffer should retain the values after the function is completed. The conv1d_operation function performs the actual convolution operation, which is shown in listing 4.3. The multiplication and accumulation is split into two separate nested loops, as during tests it was found out that Vivado HLS can optimize this design more. Another point of attention is the implementation of the padding. Instead of augmenting the input dataset with padding values, the implementation checks whether the current sample value is at the beginning (left) or end (right) of the input sequence. If so, the FIFO buffers contain input samples from different 'frames'. Where a frame is a sequence of input values that belong together, just like in an image all the pixels belong to 1 frame.

Listing 4.2: 1D convolution streaming pseudo code

```
void conv1d_streaming(din_T (&in)[input_chan],
            dout_T (&out)[num_filters],
            Conv1D_weight_t (&filter)[filt_width][input_chan][num_filters],
            Conv1D_bias_t (&bias)[num_filters],
{
    static  din_T fifo_buffer[input_chan][filt_width] = {0};
    acc_T acc[num_filters];

    for(int i=0; i<num_filters; i++){
          acc[i] = bias[i];
    }

    fifo_buf(fifo_buffer, in);
    conv1d_operation(fifo_buffer, acc, filter);
    write_output(acc, out);
}
```

Listing 4.3: Convolution operation

```
void conv1d_operation(din_T fifo_buffer[input_chan][filt_width],
                      acc_T acc[num_filters],
                      Conv1D_weight_t (&filter)[filt_width][input_chan][num_filters])
{
    mult_T mult[num_filters][input_chan][filt_width];

    for(int n=0; n<num_filters; n++){    //for all filters
    for(int k=0; k < input_chan; k++){ //for all input channels
      for(int m=0; m<filt_width; m++){ //length of the filter

                if(padding left side of input sequence){
          mult[n][k][m] = 0;
        }
        else if(padding right side of input sequence){
          mult[n][k][m] = 0;
        }
        else{
          mult[n][k][m] = fifo_buffer[k][m] * filter[m][k][n];
        }
      }
    }
    }

    for(int n=0; n<num_filters; n++){    //for all filters
    for(int k=0; k < input_chan; k++){ //for all input channels
      for(int m=0; m<filt_width; m++){ //length of the filter
        acc[n] += mult[n][k][m];
      }
    }
  }
}
```

4.3.2. 1D Pooling

The pooling operation is used to reduce the input dimensions. In the same way as the convolution layer, a FIFO buffer is used to store all the samples that are needed to produce a single output, as shown in Figure 4.3. In this case, the FIFO buffer size is equal to the pool size (P). An output is produced every time the FIFO buffer receives P samples, so that the FIFO buffer is completely refreshed with new data. Therefore, the pooling layer performs downsampling. Consequently, the throughput drops by a factor of $\frac{1}{poolsize}$. Depending on the pooling type, in this implementation the max value can be selected from the FIFO buffer, or the average value of all the values in the FIFO buffer is calculated.

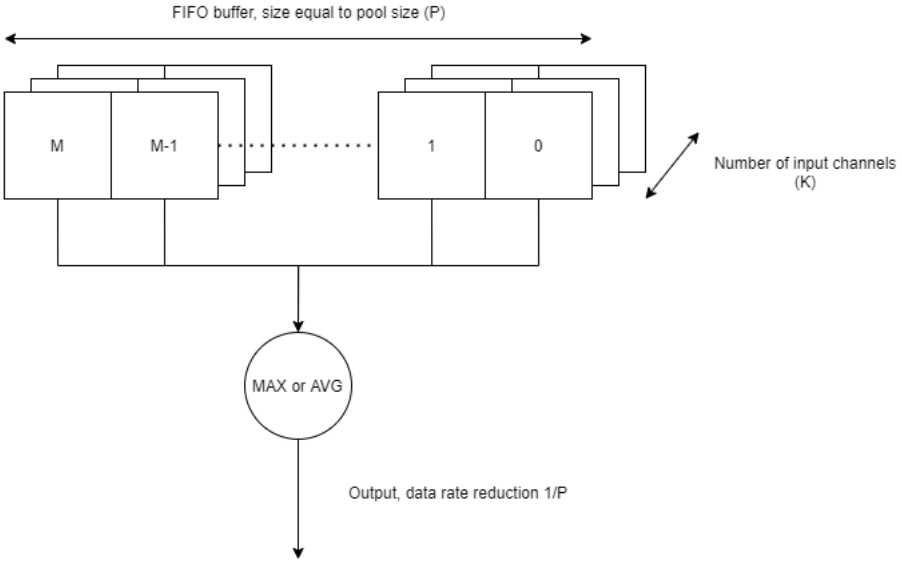


Figure 4.3: 1D pooling operation, streaming architecture

4.3.3. Activation Function

The activation function does not need FIFO buffers, because the output only depends on a single input, as shown in Figure 4.4. A ReLU activation function, can be easily be implemented as shown in listing 4.4. The ReLU activation function can be formulated by $f(x) = \max(0, x)$, therefore negative values are clipped to zero, and positive values are not modified.

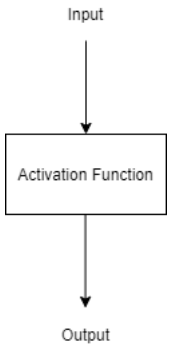


Figure 4.4: Activation function, streaming architecture

Listing 4.4: ReLU activation function pseudo code

```
for(int i=0; i<Number of filters; i++){
    input = in[i];

    if(input < 0){
        out[i] = 0;
    }
    else{
        out[i] = input;
    }
}
```

4.3.4. Dense Layer

In a dense (fully-connected) layer every output node produced by the dense layer is dependent on all the inputs to that layer. Therefore, the complete output is only valid after all the inputs nodes have been processed. As depicted in Figure 4.5 the output of the convolution layer is the input to the dense layer. Often the output of the convolution layer is converted from multidimensional (multiple channels) vector to a one-dimensional vector (nodes). In deep learning terminology this is called to 'flatten' the input.

The subsequent dense layer receives a single output of the preceding convolution layer. Thus dense layer receives only part of all the hidden nodes. Figure 4.6 shows how each part of the input is processed. Hence, every time a series of inputs are provided to the dense layer all the nodes are multiplied with the corresponding weights and accumulated in the output.

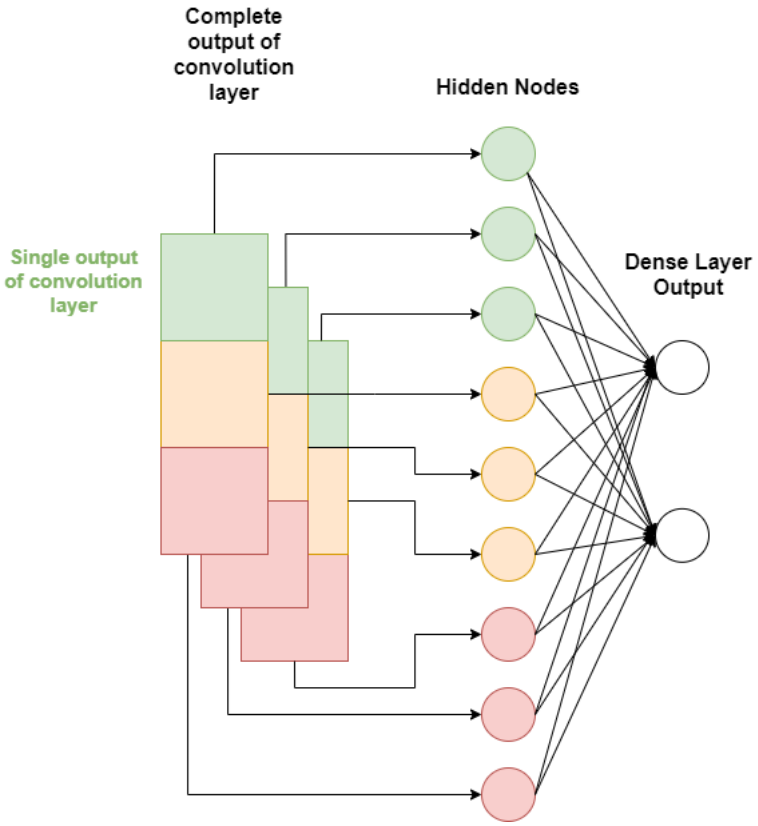


Figure 4.5: The output of a convolution layer is flatten before feeding into a dense layer.

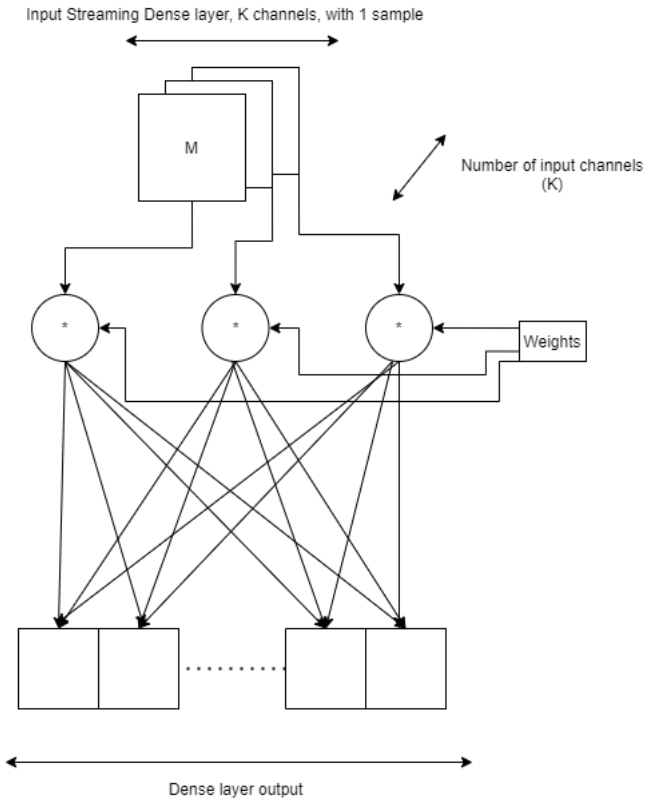


Figure 4.6: Dense operation, streaming architecture

4.3.5. Connecting multiple layers

A neural network can be constructed with the building blocks developed in this master thesis project. Figure 4.7 shows the connection of a convolutions layer to an activation layer, and the connection of the activation layer to the pooling layer. There is only 1 input sample (both I & Q data) provided to the first layer each initiation of the HLS neural network implementation. In this way not all the layers are active each initiation. In a fully pipelined design it is possible to accept a new IQ sample each clock cycle, this means to have an initiation interval (II) of 1. And an II of 2, means that only a new IQ sample can be provided to the design every two clock cycles. Therefore, the initiation interval directly influences the throughput, in classifications per second. This relation is given in equation 4.1.

$$Throughput = \frac{clock\ frequency}{II \cdot input\ length} \tag{Eq. 4.1}$$

Each pooling operation reduces the data rate in the neural network by $\frac{1}{poolsize}$. Figure 4.8 illustrates this principle for a simple neural network. This simple neural network contains 3 pooling layers with a pooling size of 2, such that they reduce the data rate by a factor of 2. In this figure it is indicated when a certain layer has produced an output. The first pooling layer produces an output every 2 samples, because this pooling layer needs two samples to calculate a summary value, such as the maximum or average. Likewise, the third pooling layer produces an output every 8 input samples. In this figure the neural network processes 8 samples in total, such that the dense layer produces an output after 8 samples. In the case that the neural network processes more than 8 samples, for example 16 samples, then the output of the dense layer would be ready after 16 samples.

CNNs can contain multiple dense layers, the first dense layer often receives its inputs from a pooling layer. The implemented pooling layer provides part of the output in a fixed interval (due to the throughput reduction), thus the output the pooling layer is spread over a number of clock cycles. Therefore, this first dense layer can simply directly use the output from the pooling layer, as illustrated in Figure 4.9. Furthermore, this figure shows the insertion of a buffer between two dense layers. This is such that the second dense layer processes its input sequentially, or in other words the output of the first dense layer is spread over multiple initiation intervals. Otherwise, the second the dense layer receives all of its inputs at once, and the processing

of this second dense layer would not be spread over a number of clock cycles. This is only possible if the data rate of the output of the first dense layer is low enough, such that the second dense layer is finished, before a new output is produced by the first dense layer.

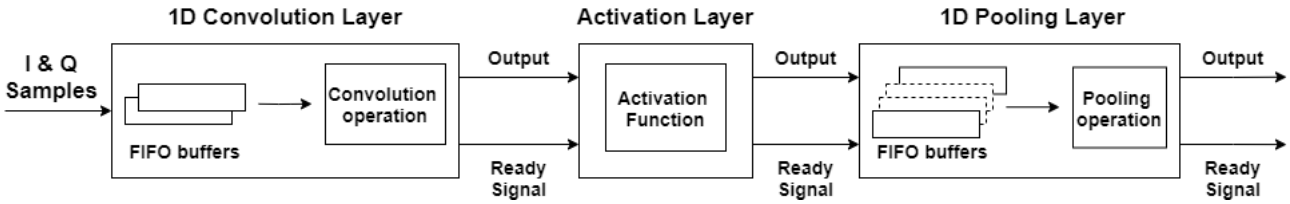


Figure 4.7

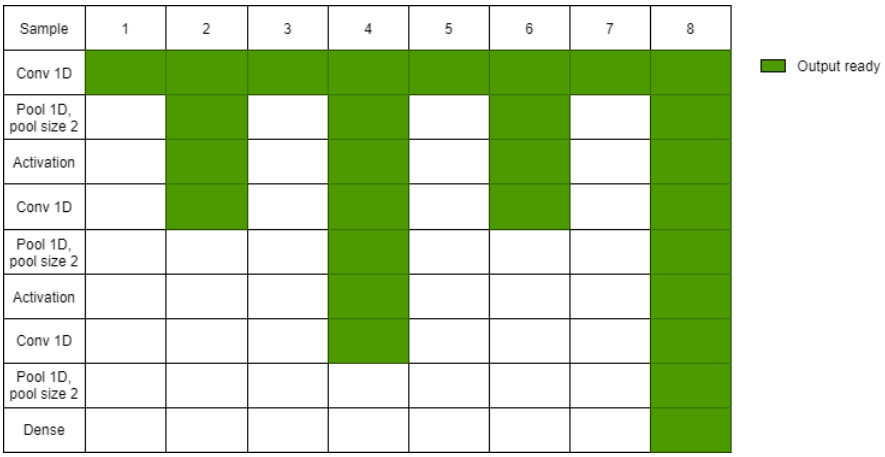


Figure 4.8: Output ready

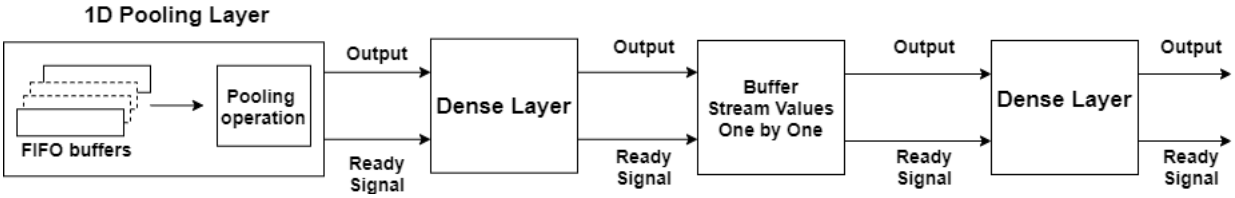


Figure 4.9: Inserting a buffer between multiple Dense layers

4.4. Conclusion

This chapter describes the implementation of a convolutional neural network FPGA accelerator with a streaming architecture. The most common building blocks of a 1D CNN are implemented: 1D convolution layer, activation function, pooling layer and a dense layer. These building blocks are designed using Vivado HLS. To construct a neural network, these building blocks can be configured and connected to each other. All building blocks use fixed point numbers, and the precision of every building block can be configured individually. The quantized neural network itself can be trained using the QKeras framework. Subsequently, this model can be converted to an FPGA accelerator using the developed HLS building blocks. This conversion is not yet done automatically, by means of a tool, instead the building blocks are manually configured and connected in order to construct the network.

The most important aspect of the accelerator is that each building block is not executed sequentially with all its inputs available, but instead each building block processes part of the whole input as soon as it is available. Therefore, the convolutional and pooling layer contains FIFO buffers to buffer all the necessary inputs in order to calculate a single output. Furthermore, the accelerator directly accepts samples from an ADC.

5

Evaluation & Results

In Chapter 4, the design and implementation of a custom CNN FPGA accelerator was described. This chapter covers the evaluation and results of this accelerator. The advantages and disadvantages/limitations are discussed. Furthermore, the results of the custom accelerator are compared with already existing solutions. In Chapter 3, hls4ml is evaluated and during this master thesis project a new version of hls4ml was released. This version is suitable for larger CNN networks, therefore we compare this new version of hls4ml with the results from this master thesis.

This chapter is structured as follows. Section 5.1 presents the metrics used for the evaluation. Section 5.2 describes the test setup and the two CNN models used for evaluation. Section 5.3 describes the process of constructing the HLS implementation of the CNN models, of which an important part is the determination of the required precision of the fixed point parameters and operations. Section 5.4 discusses the results of various FPGA implementations of the two CNN models. In Section 5.5 the process of deployment of the FPGA CNN accelerator is described. Sections 5.6 and 5.7 compare the custom accelerator with other existing solutions. Finally, Section 5.8 provides the conclusion to this chapter.

5.1. Metrics

The custom implementation is quantitatively evaluated according to the following metrics:

Throughput & Latency Throughput is measured in terms of classifications per second. In Chapter 4 the equation for calculating throughput is given by Equation 4.1. The clock frequency and the initiation interval (II) determine the data rate that the accelerator can process (data rate = $\frac{\text{Clock Frequency}}{\text{II}}$). The custom accelerator is designed to achieve an II of 1. Therefore, in order to determine the maximum data rate the accelerator can accept, the maximum clock frequency of the custom accelerator will be investigated.

The number of samples that the neural network uses to make predictions, is a value that is chosen during the training of the deep learning model. This evaluation uses an input sequence of 1024 IQ samples. This amount is common for deep learning modulation classification.

Latency is the time between the moment the FPGA accelerator receives its inputs and when the classification output is available. Like throughput, latency of the custom accelerator depends on the input length.

Resources The amount of FPGA resources that are utilized, there are four main resources present in most FPGAs: LUTs, FFs, BRAM and DSPs. It is important that the number of resources does not get too close to the maximum resources of FPGA, as this makes placement and routing more difficult. Moreover, this affects the maximum achievable clock frequency.

Accuracy The accuracy of the neural network model implemented on the FPGA. Models are often trained on desktop systems using floating point datatypes. Deploying such a model to an FPGA often requires converting the model to fixed point, or even binary and ternary representation. This quantization step could possibly lead to a decrease in accuracy of the model. For both the custom implementation and hls4ml, Qkeras is used to train a neural network using fixed point numbers. Since Qkeras already uses quantized weights and operations, the FPGA implementation should achieve a similar accuracy. In other words the accuracy or

other classification metrics of the Qkeras network can be considered as a reference for the FPGA implementation.

5.2. Test setup

All tests described in this chapter are performed using a Xilinx ZCU104 evaluation board, which contains an Zynq UltraScale+ MPSoC FPGA (part number: xczu7ev-ffvc1156-2-e). In terms of software, the following versions are used:

- Vivado HLS version: 2019.2
- Vivado Version: 2019.2

5.2.1. Evaluation CNN Models

As described in Chapter 3, the first step in deploying a neural network for inference on an FPGA using the custom implementation, is to train a network on the CPU. Qkeras is used to train a network on a custom dataset developed by TNO. This dataset contains 17 different modulation types. Each sequence is 1024 IQ samples long.

The goal of this evaluation is to evaluate whether the custom implementation is suitable for FPGA high throughput deep learning acceleration. To optimize the neural network itself, is not the main scope of the report. However the neural network should be a realistic model in the sense that it can be used for modulation classification. Two neural networks are trained for the evaluation. In Table 5.1a Model A is shown. This network uses 7 bit weights, and consist in total of 3765 trainable parameters. The model A accuracy is 62.37% for the complete TNO validation dataset. Model B is shown in 5.1b, this model is larger and contains 12041 trainable parameters. This models accuracy is 69.82%. This model contains 5 stacks, where each stack is built according to table 5.2. The accuracy obtained by Model A and B are regarded sufficient to consider the models realistic.

(a) Model A

Layer	Output shape	Number of parameters
Convolution 1D	[1024, 16]	112
ReLU	[1024, 16]	
Max Pooling 1D	[512, 16]	
Convolution 1D	[512, 16]	784
ReLU	[512, 16]	
Max Pooling 1D	[256, 16]	
Convolution 1D	[256, 16]	784
ReLU	[256, 16]	
Max Pooling 1D	[128, 16]	
Convolution 1D	[128, 16]	784
ReLU	[128, 16]	
Max Pooling 1D	[32, 4]	
Convolution 1D	[32, 4]	196
ReLU	[32, 4]	
Max Pooling 1D	[16, 4]	
Flatten	[64]	
Dense	[17]	1105

(b) Model B

Layer	Output shape	Number of parameters
Stack	[1024, 16]	1616
Max Pooling 1D	[512, 16]	
Stack	[512, 16]	1840
Max Pooling 1D	[256, 16]	
Stack	[256, 16]	1840
Max Pooling 1D	[128, 16]	
Stack	[128, 16]	1840
Max Pooling 1D	[64, 8]	
Stack	[64, 8]	536
Max Pooling 1D	[32, 8]	
Flatten	[256]	
Dense	[17]	4369

Table 5.1: Two neural networks used for evaluation. Model A has a total of 3765 trainable parameters, and model B has a total of 12041 trainable parameters. Both Model A and B are quantized neural networks constructed using QKeras. All the weights are fixed point numbers between -1 and 1, 1 bit is used for the sign and 6 bits are used for fractional part.

Convolution 1D, kernel size = 1
ReLU
Convolution 1D, kernel size = 3
ReLU
Convolution 1D, kernel size = 3
ReLU

Table 5.2: Stack buildup, as used in Model B.

5.3. Construct CNN FPGA Accelerator

The next step is to implement this neural network, in order to deploy it on an FPGA. There is not yet a tool or script to automatically convert a Qkeras network to the custom implementation's HLS code. Therefore, the implementation is manually constructed using the layer building blocks. The weights and biases in the Qkeras model are saved to a C++ header file, and the header file is imported in the custom implementation.

5.3.1. Specifying Precision

The number of bits that is used for the operations should be specified for every layer. The range of values that are possible in the model can be determined, such that the appropriate number of bits can be chosen. There are three methods considered in this thesis:

1. The minimum and maximum values that can occur in every layer in the neural network can be calculated. Consequently, this method does not take into account the actual values of the input and trainable parameters, but simply only the minimum and maximum value of the fixed point data type they use. Therefore, if the inputs or trainable parameters change, the implementation does not have to change.

2. The needed precision can be determined according to the actual inputs and trainable parameters (weights and biases), as such that this method results in an implementation that is specifically constructed for a certain set of inputs and weights. Consequently, if these inputs or weights change, then the implementation may not be able to correctly represent the values that can occur in the layer.

3. It is also possible to choose the number of bits in such a way that a trade-off is made between accuracy and resources. By specifying less bits, not all values in the neural network will be correctly represented, but it may be possible that the accuracy drop is limited. In other words, the other two methods results in an implementation that is guaranteed to achieve the same accuracy as the QKeras model. While this method makes a trade-off between accuracy and resources.

To construct a baseline for the evaluation, the number of bits in each layer is specified according to method 1. In the next two sections, the required precision is determined based on method 1 and 2.

5.3.2. Method 1: Baseline

All the trainable parameters, inputs and operations are using Vivado HLS and its fixed point datatype. In Figure 5.1, all the configuration options of this Vivado HLS fixed point datatype are illustrated. In the next paragraphs the fixed point configuration for all the separate layers are determined.

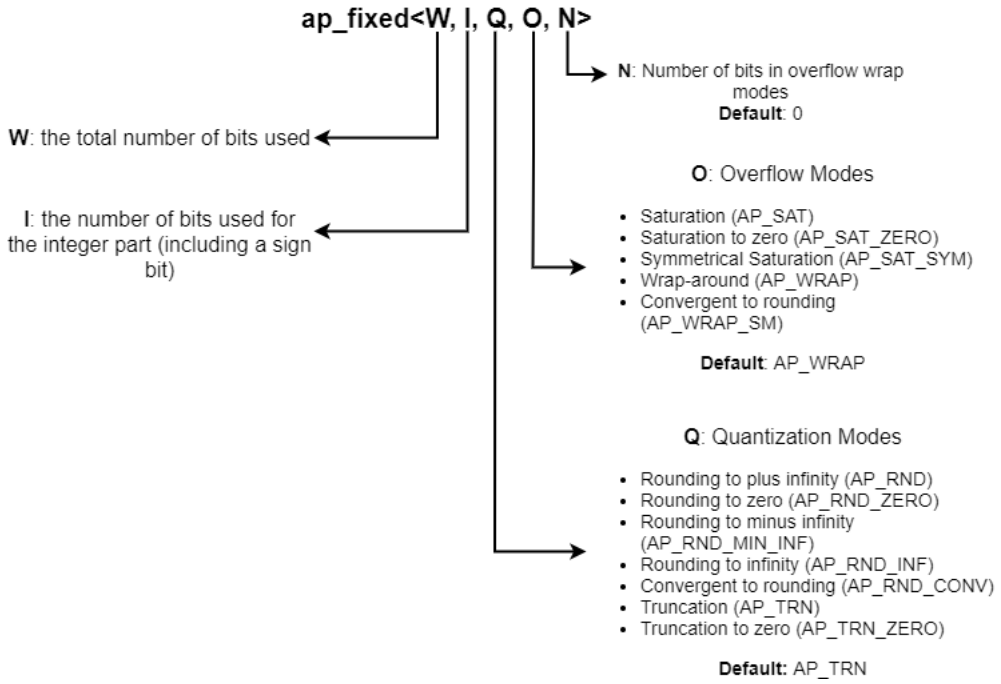


Figure 5.1: Vivado HLS fixed point datatype.

ReLU Layer The number of bits for ReLU layer are specified in the Qkeras model, so the implementation can use the same amount of bits. Furthermore, the ReLU operation in the QKeras model saturates to the maximum value of the fixed point number. Since, both QKeras models A and B use 1 bit for integer part and 6 bits for the fractional part, the maximum value is 0.984375 ($2^{-1}+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}$).

The values for the ReLU activation are clipped in the implementation by means of the Vivado HLS fixed point saturation option (AP_SAT). Another possibility would be to program this clipping behavior in the HLS code of the activation function itself, but this option led to an excessive increase in the resources for the ReLU activation.

The clipping behavior of the QKeras ReLU activate layer ensures that the maximum value cannot continue to increase, as is the case with a floating point model. Therefore, the maximum input value of both the subsequent pooling and convolution layers is always defined.

Max Pooling Layer The max pooling layer does not do any operation which changes the values of the nodes, so it can have the same number of bits as its preceding layer. Which in this case is the ReLU activation function. However, the max pooling layer, don't need the Vivado HLS fixed point saturation option.

Convolution Layer The number of bits for the convolution layer depends on the kernel size, because that determines how many values are multiplied and accumulated. Since, the number of bits of the inputs to the convolution layer and the number of bits for the weights are known. The minimum/maximum values that can occur in this layer can be calculated. The convolution operation is simply the multiplication of these inputs with the corresponding weights and accumulate all the multiplications. The range of the multiply operation is lower than from the accumulate operation. And since the multiplication and accumulation variables are split into two separate variables, the number of bits for these two variables can be specified separately.

In this case of all convolution layer have a kernel size of 3, and the input and weights are 7 bit numbers (1 integer bit, 6 fractional bits). Since, the inputs only use 1 bit the integer part, this is actually just a sign bit. Therefore, the results of the multiplication are all below 1, which means that the multiplication only

needs 1 bit instead of 2 bits. Thus the multiplication operation needs 12 (2*6) fractional bits and 1 integer bit (ap_fixed<13, 1>).

The accumulator sums up all the input channels with the result of the multiplication with the kernels. In the case of 16 input channels, the accumulator sums up 16 channels with each 3 values. Therefore, the maximum value can be calculated with Equation 5.1. In this case the input channels are 16 for the first 4 convolution layers, kernel size is 3, and maximum fixed point value is 0.984375. Hence, the maximum value for the accumulator is 47.5, thus 7 integer bits are needed. The number of fractional bits can be equal to the multiplier, which uses 12 fractional bits.

$$\text{max value accumulator} = \text{input channels} \cdot \text{kernel size} \cdot (\text{maximum fixed point value})^2 + \text{bias} \qquad \text{(Eq. 5.1)}$$

Dense Layer In worst case the dense layer sums all the input nodes containing all the minimum or maximum value. The number of integer bits can specified to be able to accommodate this situation. The dense layer in Model A, receives 64 input nodes, therefore the maximum value is 63. Hence, 7 integer bits would be needed to represent this maximum value.

Input Layer The first convolution layer operates directly on the input data. The other convolution layers operate on data from the preceding max pooling layer. Therefore, the number of bits differs for first convolution layer and should be specified according to the range of the input values.

The samples in the TNO dataset are floating point single-precision number. In Figure 5.2 two different frames (1024 samples) are shown. From this figure it is clear that between frames there is a large difference in amplitude. The minimum and maximum amplitude are approximately -490 en 482 respectively. These floating point numbers must be represented as fixed point numbers. To determine this, the entire dataset is converted to fixed point numbers and the mean absolute error (MAE) is determined with the reference floating point numbers. The result of this is shown in Figure 5.3. Firstly, the amount of integer bits are determined, by varying the number of integer bits and keeping the number of fractional bits constant (15 in this case). Figure 5.3(a) clearly shows that the MAE reaches a plateau from 10 or more integer bits. This result corresponds to the minimum (-490) and maximum (482) values found in the TNO dataset. Secondly, the number of fractional bits can be determined. Figure 5.3(b) shows the MAE for varying amount of fractional bits, and 10 integer bits. From these results, 7 fractional bits (MAE of 0.0039) is deemed a good enough representation.

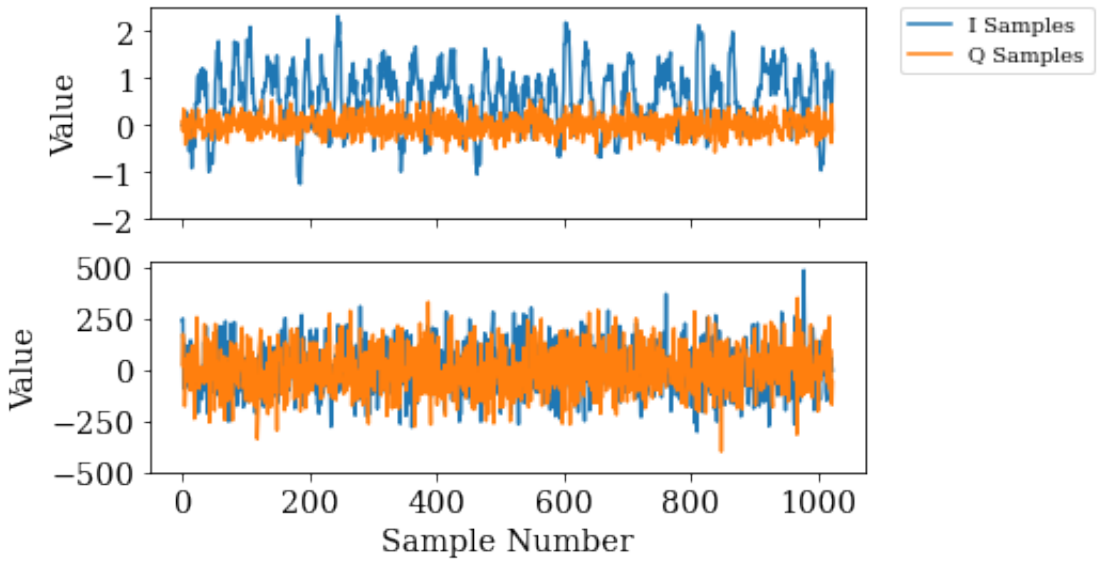
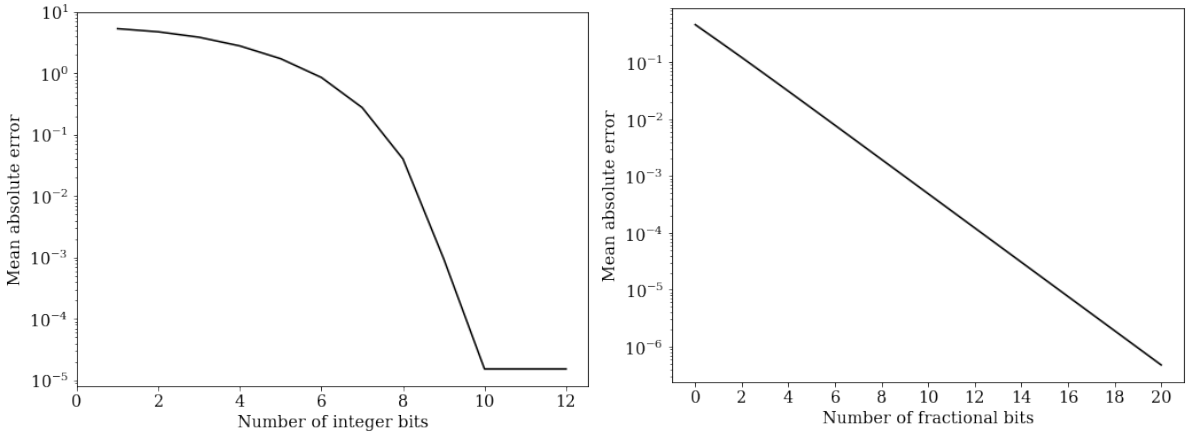


Figure 5.2: IQ samples example from TNO dataset. Note the difference in amplitude between the upper and lower plot.



(a) Integer bits

(b) Fractional bits

Figure 5.3: Input values from dataset converted from floating point to fixed point. The mean absolute error is is shown as a function of the amount of integer and fractional bits.

5.3.3. Method 2: Improvement over baseline

In this section the precision of each layer is determined based on the actual value of the trainable parameters and inputs. The validation dataset, which is also used to determine the accuracy, is provided as input to the model. It is important to realize that this dataset does not contain all the possible inputs. As a consequence, it might be possible that a certain input to the model might result in values that cannot be represented by the implementation. Therefore, it is important to have a validation dataset, that reflects the target distribution as much as possible.

In Table 5.3 the minimum and maximum output values per layer of Model A are tabulated. Likewise, Table 5.4 tabulates these values for Model B. These results are obtained by feeding all the frames of the validation dataset to the model. In appendix A, table 5.3 is shown in more detail where the minimum and maximum values are obtained per 10k frames.

Table 5.3 and 5.4 also show the required fixed point configurations, to correctly represent all the minimum and maximum values. As a result, the implementation with these datatypes, should achieve the same accuracy as the reference QKeras Model A and B.

Layer	Output Value		Required Datatype
	Min	Max	
Input	-490.73	482.97	ap_fixed<17,10>
Conv 1d	-715.91	731.30	ap_fixed<24,11>
ReLU	0.0	0.9844	ap_fixed<7,1, AP_RND, AP_SAT>
Conv 1d	-6.95	10.23	ap_fixed<17,5>
ReLU	0.0	0.9844	ap_fixed<7,1, AP_RND, AP_SAT>
Conv 1d	-3.41	7.73	ap_fixed<16,4>
ReLU	0.0	0.9844	ap_fixed<7,1, AP_RND, AP_SAT>
Conv 1d	-7.78	4.93	ap_fixed<16,4>
ReLU	0.0	0.9844	ap_fixed<7,1, AP_RND, AP_SAT>
Conv 1d	-5.38	4.62	ap_fixed<16,4>
ReLU	0.0	0.9844	ap_fixed<7,1, AP_RND, AP_SAT>

Table 5.3: The minimum and maximum output values per layer in Model A. These values are obtained by using the validation dataset as input.

	Layer	Output Value		Required Datatype
		Min	Max	
	Input	-490.73	482.97	ap_fixed<17,10>
Stack 1	Conv 1d	-735.8	747.25	ap_fixed<24,11>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
	Conv 1d	-5.28	3.97	ap_fixed<16,4>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
	Conv 1d	-3.81	7.35	ap_fixed<16,4>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
Stack 2	Conv 1d	-2.51	2.85	ap_fixed<15,3>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
	Conv 1d	-4.62	4.00	ap_fixed<16,4>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
	Conv 1d	-4.27	2.68	ap_fixed<15,3>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
Stack 3	Conv 1d	-3.32	2.59	ap_fixed<15,3>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
	Conv 1d	-6.08	2.81	ap_fixed<16,4>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
	Conv 1d	-4.55	3.54	ap_fixed<15,3>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
Stack 4	Conv 1d	-2.53	2.57	ap_fixed<15,3>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
	Conv 1d	-4.94	2.72	ap_fixed<15,3>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
	Conv 1d	-4.68	3.15	ap_fixed<15,3>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
Stack 5	Conv 1d	-2.72	2.05	ap_fixed<15,3>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
	Conv 1d	-4.36	4.32	ap_fixed<16,4>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>
	Conv 1d	-1.69	6.75	ap_fixed<16,4>
	ReLU	0.0	0.9844	ap_fixed<7,1,AP_RND,AP_SAT>

Table 5.4: The minimum and maximum output values per layer in Model B. These values are obtained by using the validation dataset as input.

5.4. Results

In the previous section, the process of determining the required precision for all the separate layers was discussed, such that there are multiple implementations of the same model. This section reviews the results of the various implementations. First, the results of Model A are reviewed. The various implementations of Model A are tabulated in Table 5.5. The baseline implementation is obtained by specifying the datatypes according to method 1 from Section 5.3.2. Next, method 2 is used in implementation 2, Finally, implementation 3, 4 and 5 are attempts for making a compromise between accuracy and resources.

Layers		Implementation				
		1. Baseline (Method 1)	2. (Method 2)	3. (Method 3)	4. (Method 3)	5. (Method 3)
Input		ap_fixed<17,10>	ap_fixed<17,10>	ap_fixed<16,10>	ap_fixed<16,10>	ap_fixed<17,10>
Conv 1D	Mult	ap_fixed<23,10>	ap_fixed<23,10>	ap_fixed<22,10>	ap_fixed<22,10>	ap_fixed<23,10>
	Acc	ap_fixed<26,13>	ap_fixed<24,11>	ap_fixed<25,13>	ap_fixed<25,13>	ap_fixed<24,11>
Conv 1D	Mult	ap_fixed<13,1>	ap_fixed<13,1>	ap_fixed<13,1>	ap_fixed<13,1>	ap_fixed<9,1>
	Acc	ap_fixed<20,7>	ap_fixed<17,5>	ap_fixed<16,4>	ap_fixed<14,2, AP_RND, AP_SAT>	ap_fixed<13,5>
Conv 1D	Mult	ap_fixed<13,1>	ap_fixed<13,1>	ap_fixed<13,1>	ap_fixed<13,1>	ap_fixed<9,1>
	Acc	ap_fixed<20,7>	ap_fixed<16,4>	ap_fixed<16,4>	ap_fixed<14,2, AP_RND, AP_SAT>	ap_fixed<12,4>
Conv 1D	Mult	ap_fixed<13,1>	ap_fixed<13,1>	ap_fixed<13,1>	ap_fixed<13,1>	ap_fixed<9,1>
	Acc	ap_fixed<20,7>	ap_fixed<16,4>	ap_fixed<16,4>	ap_fixed<14,2, AP_RND, AP_SAT>	ap_fixed<12,4>
Conv 1D	Mult	ap_fixed<13,1>	ap_fixed<13,1>	ap_fixed<13,1>	ap_fixed<13,1>	ap_fixed<9,1>
	Acc	ap_fixed<18,5>	ap_fixed<16,4>	ap_fixed<16,4>	ap_fixed<14,2, AP_RND, AP_SAT>	ap_fixed<12,4>
Dense		ap_fixed<19,7>	ap_fixed<16,7>	ap_fixed<16,7>	ap_fixed<16,7>	ap_fixed<16,7>

Table 5.5: The various implementations of Model A. This table show the specified datatype for every layer for all the implementations. The baseline is constructed by specifying the datatype for each layer according to method 1.

The results of all the implementations are depicted in Table 5.6. As indicated in Section 5.3.1, the accuracy of the implementations obtained with method 1 and 2, achieve the same accuracy as the Qkeras model. The results from Table 5.6 confirm this statement. Furthermore, from these results it can be concluded that there is a significant difference between the Vivado HLS resource estimate and the actual resources as reported by Vivado. This difference is mainly between the amount of FF and LUTs. The difference in resources between the baseline and implementation 2 quite minor. Implementation 4 results in a bad design, because of the use of the vivado HLS fixed point saturation option for the accumulator. Implementation 5 could potentially be an interesting trade-off between accuracy and resources, although an accuracy drop of almost 3 % is considerable. All these results show implementing larger neural networks on a ZCU104 board is easily possible.

		Implementation				
		1. Baseline (Method 1)	2. (Method 2)	3. (Method 3)	4. (Method 3)	5. (Method 3)
Accuracy		62.37%	62.37%	62.34%	58.63%	59.59%
Vivado HLS resource estimate	BRAM	0	2	2	2	2
	DSP	90	111	111	74	65
	FF	30644	30074	30232	63173	27124
	LUT	108214	99490	99361	278295	99609
Actual Vivado resources	BRAM	0	0	0	0	0
	DSP	111	115	115	133	65
	FF	22972	22407	22564	40576	20580
	LUT	36626	36687	36434	80206	32294

Table 5.6: Model A implementation results

	Layers		Implementation	
			1. Baseline (Method 1)	2. (Method 2)
	Input		ap_fixed<17,10>	ap_fixed<17,10>
Stack 1	Conv 1d	Mult	ap_fixed<23,10>	ap_fixed<23,10>
		Acc	ap_fixed<24,11>	ap_fixed<24,11>
	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<20,7>	ap_fixed<16,4>
	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<20,7>	ap_fixed<16,4>
Stack 2	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<17,5>	ap_fixed<15,3>
	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<20,7>	ap_fixed<16,4>
	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<20,7>	ap_fixed<15,3>
Stack 3	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<17,5>	ap_fixed<15,3>
	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<20,7>	ap_fixed<16,4>
	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<20,7>	ap_fixed<15,3>
Stack 4	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<17,5>	ap_fixed<15,3>
	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<20,7>	ap_fixed<15,3>
	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<20,7>	ap_fixed<15,3>
Stack 5	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<17,5>	ap_fixed<15,3>
	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<20,7>	ap_fixed<16,4>
	Conv 1d	Mult	ap_fixed<13,1>	ap_fixed<13,1>
		Acc	ap_fixed<20,7>	ap_fixed<16,4>
	Dense		ap_fixed<21,9>	ap_fixed<21,9>

Table 5.7: Model B implementations

For Model B, only two implementations are evaluated, which are depicted in Table 5.7. The results of these implementations are shown in Table 5.8. The accuracy is the same as the Qkeras model. As with Model A, there is again a large discrepancy between the resource utilization estimate of vivado HLS and the actual resources in Vivado. In comparison with Model A, Model B is 3.2 times larger in terms of parameters. Furthermore, in terms of resource utilization of the baseline implementation, Model B uses 2.6 times more DSPs, 2.5 times more FFs and 2.7 times more LUTs. These results indicate that it would be possible to implement larger models than Model B using the ZCU104 platform.

		Implementation	
		1. Baseline (Method 1)	2. (Method 2)
Accuracy		69.82%	69.82%
Vivado HLS resource estimate	BRAM	2	2
	DSP	243	300
	FF	77169	77106
	LUT	306153	297276
Actual Vivado resources	BRAM	0 (0%)	0 (0%)
	DSP	293 (17%)	357 (20.7%)
	FF	57346 (12.4%)	55501 (12%)
	LUT	99835 (43.3%)	98414 (42.7%)

Table 5.8: Model B implementation results

5.5. FPGA Deployment

The FPGA IP is integrated in a Vivado project, in order to deploy the CNN accelerator to an FPGA. Figure 5.4 shows the FPGA CNN accelerator IP integrated in a Vivado block diagram. The CNN accelerator consist of AXI stream interfaces for both the input stream and output stream, furthermore there is a debug stream that can be used for debugging purposes. The Stream Input Generator streams the input data to the CNN accelerator. And the Stream Output Capture IP, captures the output of the CNN IP. Both IPs are also developed using Vivado HLS. The input generator has a number of options which can be configured using the AXI4-Lite interface. For example, the input values that are streamed to the CNN IP can be chosen, such as a sequence with all ones or zeroes. The AXI4-Lite interface is controlled in Python by means of the PYNQ framework. All the inputs and outputs of the CNN IP are monitored by using the Xilinx Integrated Logic Analyzer (ILA) IP core. An example of signals monitored by the ILA is shown in Figure 5.5. Here the IP is provided with a known input, in this case all zeros. In this way it can be checked whether the design runs correctly on the FPGA. Since the TNO modulation dataset consist of 17 classes, the output of the CNN IP also consist of 17 values. Figure 5.5 shows it takes 17 clock cycles to stream all the output values. By comparing these FPGA results with the C++ and RTL level simulations, the design is proven to function correctly on the FPGA.

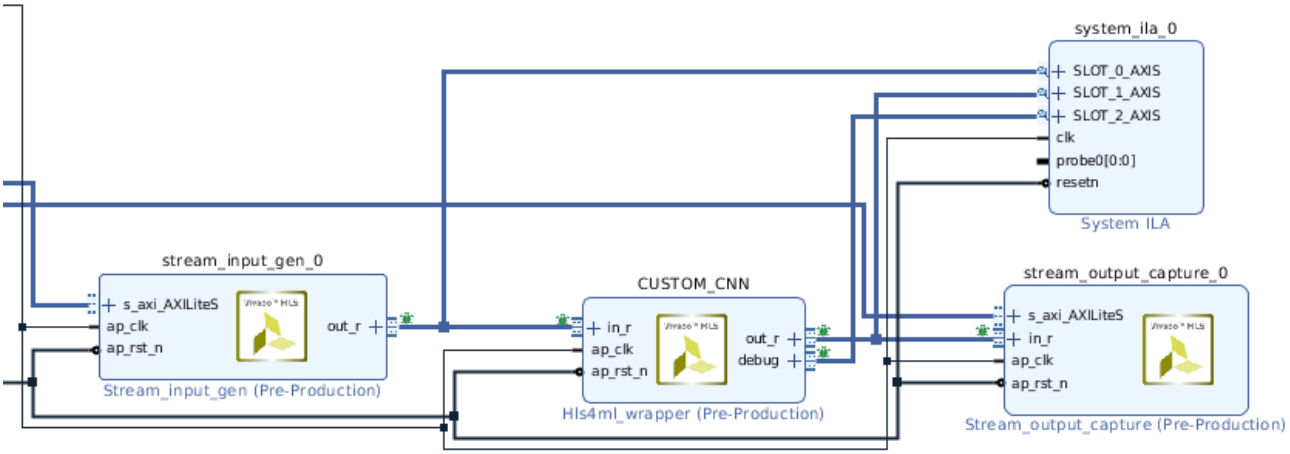


Figure 5.4: Vivado block design of the CNN accelerator.

Figure 5.5: Output of custom CNN FPGA accelerator captured by ILA.

5.5.1. Throughput & Latency Measurement

One of the design objectives, as described in Section 4.1, is to fully pipeline the design. In this way the design can directly interface with an ADC running on the same clock frequency, making a high throughput possible. As already described in Section 4, the throughput of this design is constant and determined by the design itself. Since, the FPGA implementation of Model A runs on a clock of 300MHz, the throughput is almost 293k classifications per second. The latency is measured in Figure 5.6. The input user signal, indicates the start of a new frame. For example in Figure 5.6, the user signal is active at clock cycle 200 and 1224, which correctly corresponds to a frame length of 1024 samples. Furthermore, this figure shows that the output is produced every 1024 clock cycles, in this case cycle 290 and 1314. The output that is produced at cycle 1314 belongs to the frame with the first sample at cycle 200. Therefore, the latency of Model A implementation is $1314 - 200 = 1114$ cycles, which is 3.7 us at 300MHz. Furthermore, Model B runs on a clock frequency of 250MHz which results in a throughput of 244k classifications per second and the latency is 4.6 us.

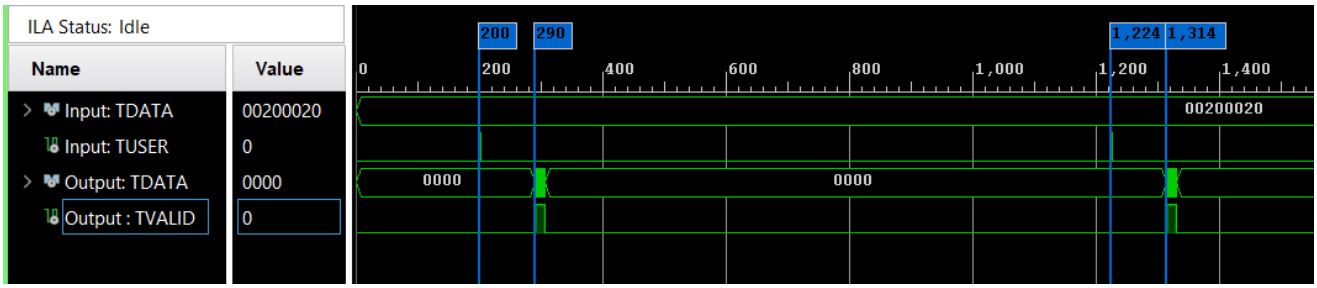


Figure 5.6: Throughput and latency measurement of custom CNN FPGA accelerator of Model A. Measured on the FPGA itself with ILA.

5.6. Comparison Hls4ml

5.6.1. Differences

Hls4ml version v0.5.0 (bartsia) implements some significant changes. These changes are described in reference [45]. The first change is that the AXI Streaming interface is used for the input and output of the accelerator. Secondly, the data between layers is also done in a streaming fashion. At a high level this is the same strategy as the implementation we developed in this master thesis project. However, there are important differences, which are now explained in this section.

Convolution implementation Both the 1D and 2D convolution operation in hls4ml are implemented based on the general matrix multiplication (GEMM) method [45]. This is different to custom implementation developed in this master thesis project, which implements the direction convolution operation. For GEMM different possible algorithms exist, such as im2col and kn2row [46]. These algorithms are mainly targeted at 2D convolution operations. Hls4ml's implementation uses an approach similar to the im2col algorithm, which is illustrated in Figure 5.7 for 2D convolution. In the im2col algorithm a part of the input is copied and unrolled into columns of a new intermediate matrix, called input-patch-matrix [46]. Furthermore, a kernel-patch-matrix is constructed by unrolling all kernels into row vectors and concatenate all the rows from each kernel. Now the convolution operation can be performed by the matrix multiplication between the input-patch-matrix and the kernel-patch-matrix. Important to note is that hls4ml does not build the entire input-patch-matrix, but instead considers one column vector at a time.

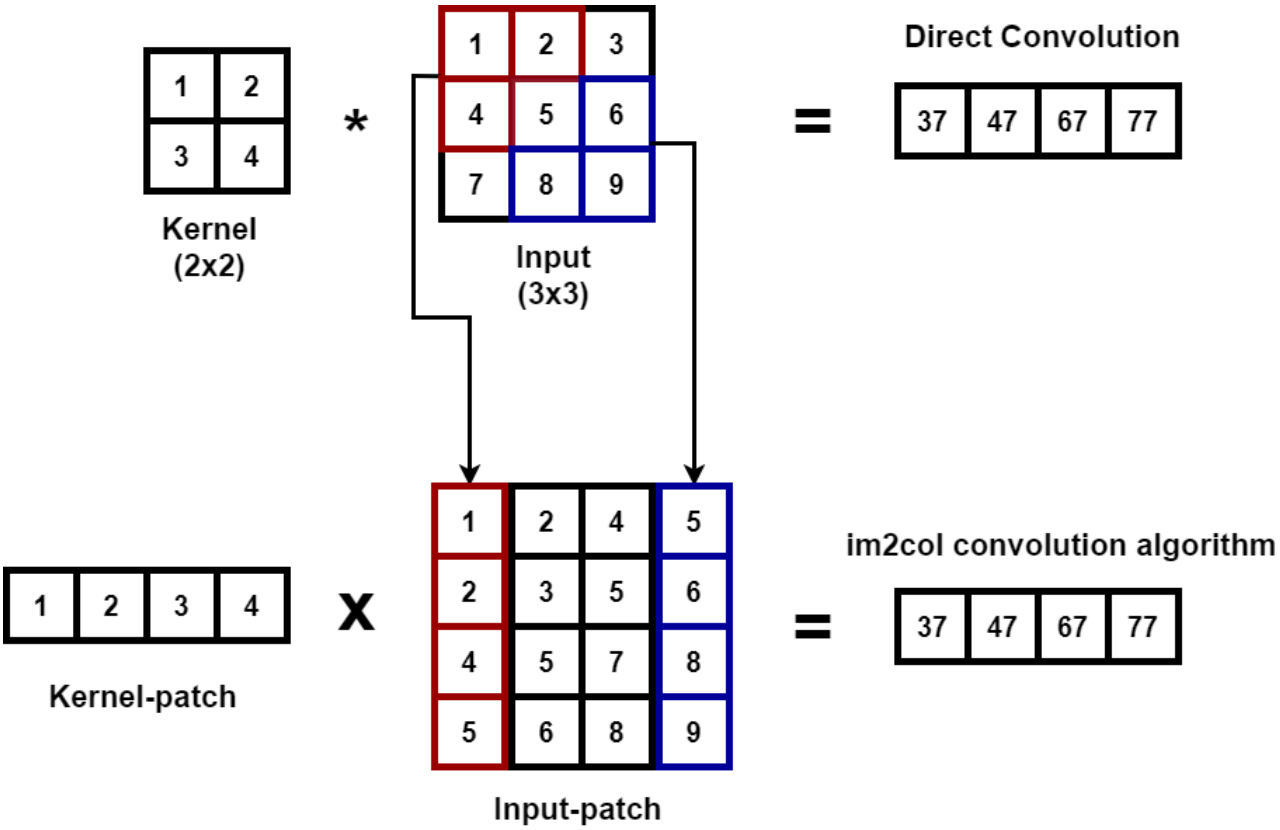


Figure 5.7: The im2col algorithm shown for 2D data, which is used in hls4ml's convolution implementation.

Streaming FIFO buffer implementation Hls4ml inserts FIFO buffers between between the layers in the neural network, such that that input and output values between the layers can be streamed. As an example, the HLS code of the first convolution, activation and pooling operation are depicted in Figure 5.8. All the FIFO buffers between the layers are inserted by Vivado HLS automatically by means of the HLS STREAM pragma. The first function, zeropad1d_cl is meant to add the zero padding to its output stream, therefore the depth of this output FIFO buffer is $1024 + 2$. Apart from the FIFO buffers between the layers, hls4ml also employ buffers inside the convolution and pooling layers itself. For example, Figure 5.9 list part of the conv1d_1d_cl function,

it shows the variable data_window which are 16 FIFO buffer of depth 1024 for the first convolution layer. These FIFO buffers are significantly deeper than in the custom implementation. This is because in hls4ml the FIFO buffer depth is determined according to the input length while in the custom solution the FIFO buffer depth is determined by the filter width. As a result, the FIFO buffer depth is reduced from 1024 to 3. Moreover, hls4ml inserts FIFO buffers between all layers, while the custom solution only inserts FIFO buffers into layers that actually need multiple samples to produce an output, such as the convolution layer and pooling layer. In addition hls4ml inserts layers that are not actually in the QKeras model, such as zeropad1d_cl and linear, which also increases the amount of FIFO buffers.

```
hls::stream<layer24_t> layer24_out("layer24_out");
#pragma HLS STREAM variable=layer24_out depth=1026
nnet::zeropad1d_cl<input_t, layer24_t, config24>(q_conv1d_input, layer24_out); // zp1d_q_conv1d

hls::stream<layer2_t> layer2_out("layer2_out");
#pragma HLS STREAM variable=layer2_out depth=1024
nnet::conv_1d_cl<layer24_t, layer2_t, config2>(layer24_out, layer2_out, w2, b2); // q_conv1d

hls::stream<layer3_t> layer3_out("layer3_out");
#pragma HLS STREAM variable=layer3_out depth=1024
nnet::linear<layer2_t, layer3_t, linear_config3>(layer2_out, layer3_out); // q_conv1d_linear

hls::stream<layer4_t> layer4_out("layer4_out");
#pragma HLS STREAM variable=layer4_out depth=1024
nnet::relu<layer3_t, layer4_t, relu_config4>(layer3_out, layer4_out); // act_1

hls::stream<layer5_t> layer5_out("layer5_out");
#pragma HLS STREAM variable=layer5_out depth=512
nnet::pooling1d_cl<layer4_t, layer5_t, config5>(layer4_out, layer5_out); // max_pooling1d
```

Figure 5.8: Code snippet of hls4ml's implementation of first convolution, relu and pooling layer of Model A.

```
hls::stream<typename data_T::value_type> data_window[CONFIG_T::filt_width * CONFIG_T::n_chan];
const int win_depth = CONFIG_T::out_width;
for (unsigned i_out = 0; i_out < CONFIG_T::filt_width * CONFIG_T::n_chan; i_out++) {
    #pragma HLS STREAM variable=data_window[i_out] depth=win_depth
}
```

Figure 5.9: Code snippet of hls4ml's data_window variable used in the conv1d_1d_cl function.

5.6.2. Comparison

This section describes the comparison between the custom implementation and hls4ml. Since the custom implementation is similar to hls4ml, a direct comparison is possible. By keeping as many parameters as possible the same, a good quantitative comparison can be made. The two neural networks (model A and B) as used in the evaluation of the custom implementation are also in the comparison between the custom implementation and hls4ml.

As described in Section 3.2 hls4ml can automatically convert a deep learning model to an HLS project. Firstly, hls4ml is used to create a HLS project for Model A with the datatypes configured according to implementation 3 in Table 5.5. This hls4ml correctly synthesizes and achieves the same accuracy of 62.34% as the custom accelerator. The RTL simulation of the hls4ml implementation is depicted in Figure 5.10. In this figure two markers are placed to measure the throughput, in this case a single frame (1024 samples) is processed after 2058 cycles. Another point of attention is the amount of input and output streams generated by hls4ml. The hls4ml implementation constructs a separate AXI streaming interface for each input and output channel. Therefore the input consist of 2 AXI streaming interfaces, and the output consist of 17 AXI streaming interfaces. In Figure 5.10 only 2 of the total 17 of such output streams are shown in the simulation. A wrapper is added to the hls4ml IP, such that the input and output can be provided by a single AXI Streaming Interface, in order to integrate the hls4ml IP into Vivado in a similar way as the custom accelerator (Figure 5.4).

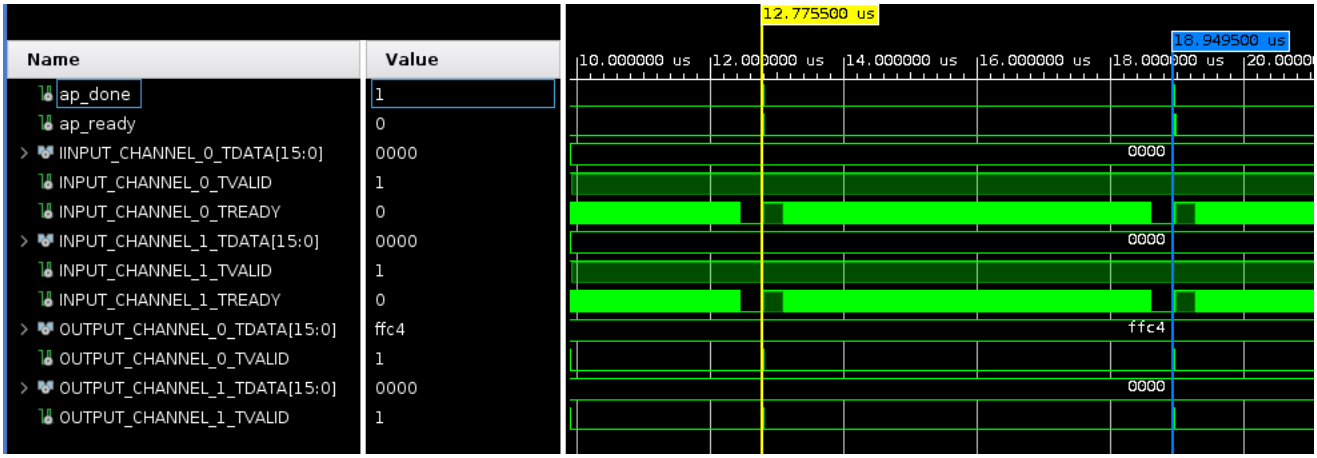


Figure 5.10: Output of CNN accelerator as generated with the hls4ml tool. All the 17 output classes are generated as a separate AXI streaming interface. A wrapper is constructed to use a single AXI streaming interface for the input as well as the output.

The next step is to directly compare hls4ml implementation of Model A and B to the custom implementation. The results of this comparison for Model A are tabulated in table 5.9.

		Implementation			
		Custom Model A	Hsl4ml Model A	Custom Model B	Hls4ml Model B
Accuracy		62.34%	62.34%	69.82%	69.82%
Vivado HLS Resource Estimation	BRAM	2	565	2	1225
	DSP	111	34	300	6
	FF	30232	61139	77106	154620
	LUT	99361	171060	297276	507215
Actual Vivado Resources	BRAM	0 (0%)	218.5 (70.0%)	0 (0%)	Vivado Implementation Fails
	DSP	115 (6.7%)	35 (2.0%)	357 (20.7%)	
	FF	22564 (4.9%)	48617 (10.5%)	55501 (12.0%)	
	LUT	36435 (15.8%)	79317 (34.5%)	98414 (42.7%)	
Vivado Timing	Clock Frequency	300 MHz	250 MHz	250 MHz	
	WNS	0.002 ns	0.081 ns	0.045 ns	
Performance	Throughput	292.97k	121.48k	244.14k	
	Latency	1114 cycles 3.71 us	2183 cycles 8.55 us	1153 cycles 4.61 us	

Table 5.9: Results of direct comparison custom and hls4ml Model A implementation

From these results it can be concluded that the amount of resources of the hls4ml implementation is significantly higher. For example, the hls4ml implementation of Model A has a 70% BRAM utilization, due to the inserted FIFO buffers. Whereas the custom implementation does not use BRAM. This large amount of BRAM forms a limitation for hls4ml to be able to synthesize larger networks. For instance, implementing Model B using hls4ml fails. Besides the lower BRAM utilization in the custom implementation, also the amount of FFs and LUTs are 2.15 times and 2.18 times lower. The only resource that hls4ml uses less of are the DSPs, but in percentage terms this is still a small difference. From these results it can be concluded that the custom implementation has a significantly smaller footprint, while still achieving the same accuracy. Therefore, the

custom solution allows to make various design choices, such as FPGA deployment of larger neural network or the use of an FPGA in a more more efficient way.

In terms of latency and throughput, the custom solution also achieves better results. For Model A, the throughput is 2.4 times higher and the latency is 2.3 times lower. Since a key design goal of hls4ml is low latency, these results are not only interesting for RF use cases, but also for use cases where hls4ml is used.

5.7. Comparison with TNN FPGA accelerator from [36]

As described in Section 2.4 the implementation in [36] is claimed to be the fastest, open-source, FPGA-based modulation classification implementation. In this section we compare the results from this paper with the results obtained in this master thesis project. In [36] various ternary weights CNNs are trained using the RadioML 2018.01A DeepSig dataset [23]. Therefore Model A and Model B are also trained using the same dataset, such that a direct comparison is possible. In [37] the accuracy of the models are specified at +30dB SNR. The results from the comparison are tabulated in 5.10. A number of observations can be made. Firstly, the number of parameters for the ternary weights models are significantly larger. Model B uses almost 53 times less parameters than the largest models from [36]. This difference in parameters is logical since binary weights and ternary weights networks require more parameters compared to a fixed point or floating point model to achieve comparable accuracy. Still, the accuracy of Model B is greater than the accuracy of the best performing ternary weight model TW-128. Next, the resources of the FPGA accelerator implementation are compared. It should be noted that the target device in [36] is the Xilinx Zynq Ultrascale+ RFSoC (part number xczu28dr-ffvg1517-2-e). However, the Zynq UltraScale+ MPSoC as used in this master thesis, employs the same FPGA technology, therefore a fair comparison can be made. Both Model A and B have a lower resource utilization in every aspect compared the ternary weights models. Model B compared to TW-128 uses almost 3.2 times less LUTs, more than 9 times less FFs, 4 times less DSP units and uses no BRAM. Moreover, the implementation of TW-128 could not be successfully implemented in [36], because it failed to route.

Nevertheless, the implementations from [36] can process two input samples at once at a frequency of 250MHz, such that the accelerator can effectively process samples at a frequency of 500MHz. While the Model A processes samples at a frequency of 300MHz and Model B at a frequency of 250MHz. Still the latency of Model A and B is 3.7μs and 4.6 μs respectively, while the ternary weights networks have a latency of 8 μs. In order to increase the throughput of the custom solution, the implementation is modified to also process two input samples in a single clock cycle. Due to time constraints in this master thesis project, this has only been done for Model A. Table 5.10 shows the result of this modification. It can be observed that this adjustment does not have much impact on the number of resources while the data rate that this accelerator can be process in real-time is increased from 300MHz to 600MHz. To the best of our knowledge this modification results in the fastest FPGA-based automatic modulation classification.

In conclusion, in this section we demonstrate that the FPGA implementation of large ternary neural networks (in terms of trainable parameters) does not result in a more efficient design in terms of accuracy, resources, throughput and latency when compared to custom implementation developed in this master thesis which targets the fixed point models. This is an interesting result, as much research is being performed in the literature on binary/ternary neural networks, especially in the context of the FPGA implementations.

Model Name	# parameters	Accuracy +30dB SNR	LUTs	FFs	BRAMs	DSPs
TW-128[1]	636k	81.7%	320k (75.3%)	506k (59.5%)	524 (48.5%)	1431 (33.5%)
TW-96	490k	81.1%	232k (54.7%)	369k (43.4%)	524 (48.5%)	1207 (28.3%)
TW-64	381k	78.7%	124k (29.1%)	217k (25.5%)	524 (48.5%)	1496 (35%)
TW-INCRA-128	636k	80.2%	211k (49.6%)	324k (38.1%)	512.2 (48.3%)	1407 (32.9%)
TW-BA-128	636k	75.9%	234k (55.1%)	333k (39.2%)	523 (48.4%)	1408 (33.0%)
TW-BA-64-512	381k	67.4%	80k (18.8%)	108k (12.7%)	521 (48.2%)	1471 (34.4%)
TW-BA-64	102k	63.1%	58k (13.7%)	92k (10.8%)	76 (7.0%)	704 (16.5%)
Model A	3765	62.2%	36k (8.5%)	23k (2.7%)	0 (0%)	115 (2.7%)
Model B	12041	82.3%	98k (23.0%)	56k (6.6%)	0 (0%)	357 (8.4%)
Model A 2 samples in single clock cycle	3765	62.2%	37k (8.7%)	21k (2.5%)	0 (0%)	127 (3.0%)

Table 5.10: Comparison between results from [36] and the results from this master thesis.[1] Failed to route in [36]

5.8. Conclusion

To evaluate the custom accelerator two CNN models are trained using QKeras on a custom dataset developed by TNO. The first model, Model A, has approximately 3.7k trainable parameters (7-bit fixed point) and achieves an accuracy of 62.37%. The second model, Model B, has approximately 12k trainable parameters (7-bit fixed point) and achieves an accuracy of 69.82%. Both models are implemented using the custom developed building blocks and the FPGA implementation is quantitatively evaluated according to the following metrics: throughput, latency, resources and accuracy. Different methods for determining the required precision of each layer are used to construct multiple implementations. In the baseline implementation of Model A and B, the precision is determined in such a way to guarantee correct results for all the possible inputs. Hence, the accuracy of baseline implementations achieves exactly the same accuracy as the QKeras model running on the CPU. The Model A implementations can run on a clock frequency of 300MHz, which results in a throughput of almost 293k classifications per second and a latency of 3.7μs. And the throughput and latency of implementation of Model B running on 250MHz is 244k and 4.6μs respectively. Furthermore, the resource utilization of the FPGA implementation of Model B, indicate that larger neural networks are possible.

Additionally, the custom accelerator is also compared to the hls4ml [39] tool and the results from paper [36]. During this master thesis project a new version of hls4ml was released which addresses the issue of being able to only run very small CNNs. This version of the hls4ml tool can successfully implement Model A, but the implementation of Model B results in too large resource utilization for the ZCU104 platform. The implementation of hls4ml Model A has a throughput of 121k classifications per second and a latency of 8.6 us. Moreover, the resource utilization of the hls4ml implementation is significantly higher compared the implementation developed in this master thesis. An important difference between the hls4ml and custom implementation is that the custom implementation does not use BRAM, and that the excessive BRAM utilization of hls4ml poses a limit to be able to implement larger networks.

Finally, the custom accelerator is also compared to the FPGA modulation classifier implementation described in paper [36]. The models implemented in this paper are all ternary weights neural networks, which means that the weights can have the value -1, 0 or 1. From the comparison between the ternary weights models and fixed point models A and B, it appears that the ternary weights models from [36] have a significantly higher resource utilization in order to achieve a comparable accuracy. This is a very interesting result, because binary and ternary neural networks are mainly researched in the context of FPGA implementation, because the operations in these networks can be efficiently implemented. Hence, we demonstrate that ternary weights neural networks does not necessarily lead to a better FPGA implementation.

6

Demonstration

Chapter 5 evaluates the custom FPGA CNN accelerator. Part of this evaluation was the integration of the CNN accelerator into an FPGA design, which verified that the design is functionally correct. However, Chapter 5 did not integrate the FPGA CNN accelerator IP into an FPGA based Software Defined Radio (SDR) platform. Therefore, the purpose of this chapter is to demonstrate a deep learning modulation classifier deployed to an FPGA based Software Defined Radio.

6.1. Setup

This demonstration runs on a Xilinx Zynq-7000 SoC ZC706 evaluation platform. This evaluation platform consist of a Zynq-7000 FPGA (part number: XC7Z045-FFG900–2).

To capture the RF signals, an AD-FMCOMMS3-EBZ RF transceiver evaluation board is used. This evaluation board is based on the Analog Devices AD9361 RF transceiver [47]. The RF-ADC has a resolution of 12-bit I and Q data and works at a maximum sample frequency of 61.44MHz, with a maximum bandwidth is 56MHz. The AD-FMCOMMS3-EBZ evaluation board complies with the FPGA Mezzanine Card (FMC) standard [48], which means that it can be connected to the ZC706 board with a standardised connector, which is depicted in Figure 6.1.

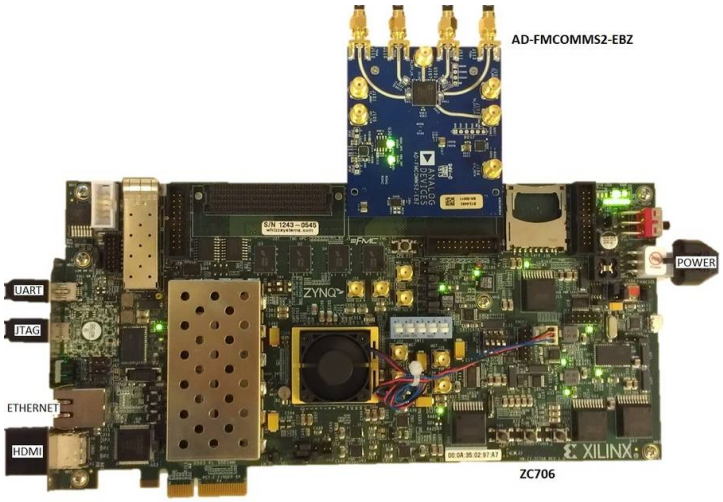


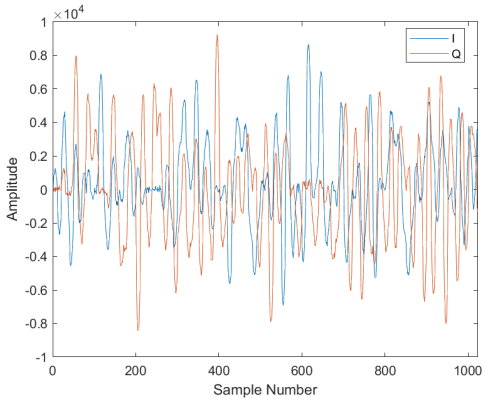
Figure 6.1: FPGA based software defined radio. The ZC706 evaluation platoform is used in combination with the FMCOMMS3 RF transceiver. Figure is from [49] and shows the AD-FMCOMMS2 RF tranceiver instead of the FMCOMMS3.

The samples are transmitted and received by the same FMCOMMS3 and TX and RX are connected by a loopback cable. A Matlab script is used to control the settings of the RF transceiver, such as gain and sample rate, and to transmit the samples and receive the output from the classifier.

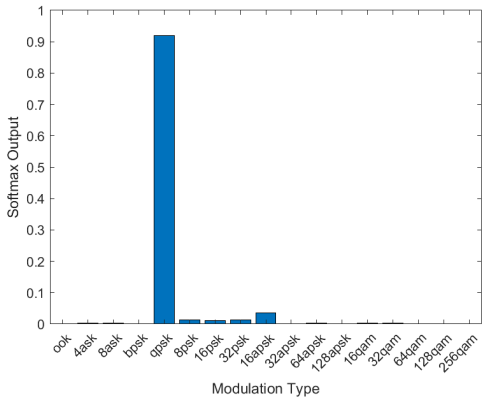
6.2. Results

The FPGA modulation classifier is directly fed with input signals originating from the TNO dataset. For example, Figures 6.2(a) and 6.3(a) depicts the transmitted signal with QPSK modulation and 4ASK modulation respectively. Both signals have a length of 1024 samples. The output of the classifier is depicted in Figure 6.2(b) and Figure 6.3(b). The x-axis lists all the 17 modulation types, and both the QPSK and 4ASK input signals are correctly classified.

Based on the test performed, it has been found that the 12-bit ADC/DAC has a negative influence on the accuracy of the classifier. This is as expected since the FPGA IP has an input of 16 bit which is specifically determined according to the training dataset and therefore by using a 12 bit ADC/DAC information is lost. Unfortunately, in the current demonstration platform, the I and Q signals are frequently interchanged. This behaviour is also present in the provided base application by analog devices [50], which is used as a base when integrating the modulation classifier IP. Hence, due to this behaviour it is not possible to correctly determine the accuracy of the classifier running on this demonstration platform using the entire dataset. However, by capturing the signals that feed into the classifier it is concluded that the classifier is functionally correct. Therefore, the results of this demonstration proves that FPGA CNN accelerator of a model for the modulation classification use case can be used in a SDR system.

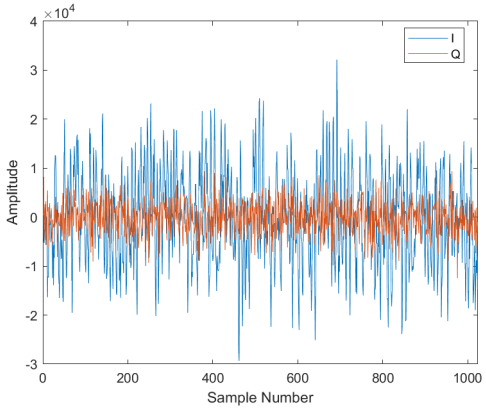


(a) Transmitted QPSK signal.

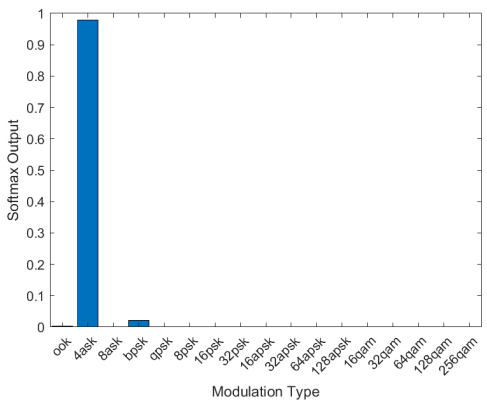


(b) Classifier prediction (Softmax Output).

Figure 6.2: Prediction of FPGA modulation classifier with a QPSK signal.



(a) Transmitted 4ASK signal.



(b) Classifier prediction (Softmax Output).

Figure 6.3: Prediction of FPGA modulation classifier with a 4ASK signal.

6.3. Conclusion

In this chapter a demonstration is given of a functional deep learning modulation classifier running on an FPGA based Software Defined Radio platform. This demonstration proves that a DL model can run on the FPGA in a SDR platform and process RF data in real-time.

7

Conclusion

7.1. Conclusions

In Chapter 1 an introduction of the assignment was provided. In brief, deep learning for RF applications is emerging and the goal is to actually use these deep learning models in communication systems. Because, FPGAs are often used in software defined radio systems, it is useful to be able deploy these models to FPGAs. Since the applications of RF deep learning are endless, this thesis specifically deals with the modulation classification use case.

Next, Chapter 2 provides a more in-depth background study into the research fields involved in this thesis, which are deep learning for modulation classification and FPGA acceleration of deep learning models. Also an introduction to software defined radios is given. The next two paragraphs summarise the most important conclusions about deep learning for modulation classification and deep learning FPGA acceleration.

Deep learning for modulation classification is a new research field and only a limited amount of publications on this topic exist. Existing deep learning models for modulation classification are often based on Convolutional Neural Networks (CNN) or Recurrent Neural Networks (RNN). Both models achieve promising results and there is not yet a clear consensus as to which models are best suited for modulation classification. However, convolutional neural networks are used more often and may be easier to train. Furthermore, TNO already has experience with CNNs for modulation classification. Therefore, in this thesis CNNs are used for the modulation classification use case.

A reasonable amount of research has already been done into the use of FPGAs for deep learning acceleration, but many of these studies are fully focused on vision tasks. Therefore it is useful to test some of these solutions for RF deep learning applications. There are mainly two different FPGA architecture for deep learning acceleration.

- Streaming architecture in which the data samples are 'streamed' through the neural network. The accelerator is specific to a particular network. This means that the accelerator has to change to run a different network.

- Computation unit architecture which executes instructions. The resulting accelerator does not have to change for different networks, as long as these networks can be converted to compatible instructions.

In the evaluation of existing solutions a number of aspects are of particular importance for FPGA based DL acceleration for RF applications. First, the latency and throughput are important because of the communication context. Second, the use of FPGA resources determines whether one or more neural networks can be deployed on an FPGA to run in coexistence with other DSP functions in an FPGA-based SDR.

Chapter 3 discusses the preliminary tests, in which the suitability of existing FPGA deep learning for RF applications is evaluated. Two relatively mature tools for deploying deep learning models to FGPAs are evaluated. Firstly, Vits AI is tested and secondly hls4ml is tested. Vitis AI is based on a computation unit architecture, and hls4ml is based on a streaming architecture. The preliminary tests show that Vitis AI can be useful for TNO in certain cases. These are cases where large neural networks are required and a high throughput/low latency is not a requirement. It is important to mention that vitis AI currently only supports 8-bit data input. This forms a limitation for RF deep learning, since RF ADCs are often 12 to 16 bit. Perhaps this will change in

the future, because of the rapid developments in FPGA deep learning accelerators. Tests with hls4ml proves that hls4ml can only be used for very small networks, because of this limitation hls4ml is not useful for TNO. However, its fully pipelined streaming architecture still has advantages, such as a high throughput and low latency. Therefore, a custom CNN FPGA accelerator is developed based on such a streaming architecture. This accelerator should support larger networks than hls4ml currently supports, and provide a high throughput and low latency accelerator.

Chapter 4 describes the implementation of a CNN FPGA accelerator that is based on a streaming architecture. The most common building blocks of a 1D CNN are implemented namely: 1D convolution layer, activation function, pooling layer and a dense layer. These building blocks are designed using Vivado HLS. To construct a neural network, these building blocks can be configured and connected to each other. All building blocks use fixed point numbers, and the precision of every building block can be configured individually. The quantized neural network itself can be trained using the QKeras framework. Subsequently, this model can be converted to a FPGA accelerator using the developed HLS building blocks. This conversion is not yet done automatically, by means of a tool, instead the building blocks are configured and connected in order to construct the network.

The most important aspect of the accelerator is that each building block is not executed sequentially with all its inputs available, but instead each building block processes part of the whole input as soon as it is available. Therefore, the convolutional and pooling layer contains FIFO buffers to buffer all the necessary inputs in order to calculate a single output. Furthermore, the input of the accelerator is streaming input, such that the samples of an ADC can directly be fed to the accelerator.

In Chapter 5, the custom accelerator is evaluated. In order to evaluate the custom accelerator, two quantized CNN models are trained using QKeras on a custom dataset developed by TNO. The first model, model A, has approximately 3.7k trainable parameters (7-bit fixed point) and achieves an accuracy of 62.37%. The second model, model B, has approximately 12k trainable parameters (7-bit fixed point) and achieves an accuracy of 69.82%. Both models are implemented using the custom developed building blocks and the FPGA implementation is quantitatively evaluated according to the following metrics: throughput, latency, resources and accuracy. Different methods for determining the required precision of each layer are used to construct multiple implementations. In the baseline implementation of model A and B, the precision is determined in such a way to guarantee correct results for all the possible inputs. Hence, the accuracy of baseline implementations achieves exactly the same accuracy as the QKeras model running on the CPU. The model A implementations can run on a clock frequency of 300MHz, which results in a throughput of almost 293k classifications per second and a latency of 3.7μs. And the throughput and latency of implementation of model B running on 250MHz is 244k and 4.6μs respectively. Furthermore, the resource utilization of the FPGA implementation of model B, indicate that larger neural networks are possible.

Additionally, the custom accelerator is also compared to the hls4ml [39] tool and the results from paper [36]. During this master thesis a new version of hls4ml was released which addresses the issue of being able to only run very small CNNs. This version of the hls4ml tool can successfully implement model A, but the implementation of model B results in too large resource utilization for the ZCU104 platform. The implementation of hls4ml model A has a throughput of 121k classifications per second and a latency of 8.6 us. Moreover, the resource utilization of the hls4ml implementation is significantly higher compared the implementation developed in this master thesis. An important difference between the hls4ml and custom implementation is that the custom implementation does not use BRAM, while the excessive BRAM utilization of hls4ml poses a limit to be able to implement larger networks.

Finally, the custom accelerator is also compared to the FPGA modulation classifier implementation described in paper [36]. The models implemented in this paper are all ternary weights neural networks, which means that the weights can have the value -1, 0 or 1. From the comparison between the ternary weights models and fixed point models A and B, it appears that the ternary weights models from [36] have a significantly higher resource utilization in order to achieve a comparable accuracy. This is a very interesting result, because binary and ternary neural networks are mainly researched in the context of FPGA implementation, because the operations in these networks can be efficiently implemented. Hence, we demonstrate that ternary weights neural networks does not necessarily lead to a better FPGA implementation.

Chapter 6 provides a demonstration of the fully functional deep learning modulation classifier running on an FPGA based Software Defined Radio platform. This offers new possibilities for further research into deep learning algorithms for RF applications.

7.2. Main contributions

As described in the introduction in Chapter 1, the intent of this thesis is provide a new research initiative for utilizing FPGAs in RF deep learning applications. This intent translates to the following main research question:

How can FPGAs be efficiently utilized to implement deep learning models for modulation classification in a Software Defined Radio (SDR)?

In this thesis an FPGA CNN accelerator for deep learning has been developed, which makes it possible to deploy deep learning modulation classification models to an FPGA in a Software Defined Radio (SDR).

The main contributions of this thesis are listed below:

- Implementation of a custom FPGA CNN accelerator using reusable and configurable building blocks. The accelerator employs a streaming architecture and is fully pipelined, such that it accepts new input data every clock cycle. A key design aspect is that all building blocks in the accelerator are designed to be able to work on a portion of its input data. The implication is that the building blocks can produce an output as soon as enough input data is available. As a result, the work that the building blocks have to perform is spread out over time and the memory required for storing data is also reduced. Moreover, the precision of the fixed point parameters and operations is configurable. Therefore there is no limitation of only specifically supporting binary or ternary operations.
- To the best of our knowledge, the fastest FPGA-based automatic modulation classification implementation. This implementation is capable of real-time processing data at a frequency of 600MHz, which results in a throughput of almost 586k classifications per second for a frame consisting of 1024 samples.
- Evaluation of custom accelerator using two CNN models, Model A and Model B. These models are implemented using the in this master thesis project developed FPGA CNN accelerator. When compared to similar initiatives that exist also outside the area of RF applications, the implementation strategy developed in this research has shown to yield a more compact FPGA implementation in terms of resources, while achieving a higher throughput, lower latency and higher model accuracy.
  - In comparison with hls4ml, the resulting accelerator achieves 2.4 times higher throughput and 2.3 times lower latency for Model A, while also achieving the same accuracy and significantly lower resource utilization. In particular, the significant BRAM utilization of hls4ml limits the scalability of the accelerator to support larger neural networks. The custom accelerator uses no BRAM for both Model A and Model B. Unlike the custom accelerator, the Model B hls4ml implementation cannot be successfully deployed.
  - The implementation described by Tridgell et al. [36] for implementing a ternary neural network FPGA based automatic modulation classification achieves a significantly higher resource utilization. For example, when compared to the custom Model B implementation the accelerator proposed by Tridgell et al. [36] results in 3.3 times more LUTs, 9 times more FFs, 4 times more DSPs and a significant amount of BRAM. Moreover, the accuracy of the best performing model from [36] is 0.6% lower.
- Demonstration of a functional modulation classifier deployed to an FPGA in a Software Defined Radio (SDR).

7.3. Future Work

We presented a fully functional implementation enabling the deployment of deep learning models to FPGAs for RF applications as well as beyond the RF domain. A number of proposals for future work are provided in the list below, building on the work of this thesis.

- Develop an end-to-end tool, such that the network can be trained in Qkeras on the CPU and the tool automatically generates the HLS implementation of the accelerator. Furthermore, all the configuration options should be easily configurable in the tool itself and the simulation and synthesis results can be automatically generated in order to facilitate the design exploration. The goal should be that deep learning engineers could use FPGAs to accelerate deep learning models, without extensive or specific knowledge of FPGAs. In this way the barrier is lowered to deploy deep learning models to FPGAs.

- Expand the available building blocks to support other types of neural networks such as Residual Neural Networks and Recurrent Neural Networks. Furthermore, the current implementation only supports 1D operations, so the support of 2D allows for applications that require 2D operations.
- The current implementation works especially well for high throughput and/or low latency applications, because of its fully pipelined design. However, this implementation is not suitable for large neural networks in the order of hundreds of thousands or millions of trainable parameters. Therefore, the possibility can be investigated to customize the implementation, so that the implementation can be specifically configured for certain resources, throughput or latency requirements. Hence, a trade-off between performance and resources could be made.
- In Section 5.7 the result of processing multiple input samples per clock cycle has already been discussed. It is future work is to explore this aspect more in-depth. Also related to this is throughput matching, in which the implementation takes into account the differences in throughput that exist in different layers of the neural network. By utilizing throughput matching, it could be possible to construct or configure a building block for each layer in the neural network that support a specific throughput, which ideally results in an optimized implementation in terms of resources.
- Further evaluate the implementation of the custom accelerator. For example the maximum neural network size that can be deployed using the custom FPGA accelerator can be investigated.

Bibliography

[1] J. Mitola and G. Maguire, "Cognitive radio: Making software radios more personal," IEEE Personal Communications, vol. 6, no. 4, pp. 13–18, 1999. DOI: 10.1109/98.788210.

[2] J. Mitola, "Cognitive radio for flexible mobile multimedia communications," in 1999 IEEE International Workshop on Mobile Multimedia Communications (MoMuC'99) (Cat. No.99EX384), 1999, pp. 3–10. DOI: 10.1109/MOMUC.1999.819467.

[3] D. Inc. (2021). "Deepsig," [Online]. Available: https://www.deepsig.ai/ (visited on 06/16/2020).

[4] D. Digital. (2021). "Deepwave digital," [Online]. Available: https://deepwavedigital.com/ (visited on 06/16/2020).

[5] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, and J.-M. Frahm, "Re-thinking cnn frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge," in 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2019, pp. 305–317. DOI: 10.1109/RTAS.2019.00033.

[6] Zynq ultrascale+ rfsoc data sheet: Overview, RFSoC, Rev. 1.12, Xilinx, Apr. 2021.

[7] R. Utrilla, E. Fonseca, A. Araujo, and L. A. Dasilva, "Gated recurrent unit neural networks for automatic modulation classification with resource-constrained end-devices," IEEE Access, vol. 8, pp. 112 783–112 794, 2020. DOI: 10.1109/ACCESS.2020.3002770.

[8] R. Stewart, K. Barlee, D. Atkinson, and L. Crockett, Software Defined Radio using MATLAB & Simulink and the RTL-SDR. Sep. 2015, ISBN: 978-0-9929787-2-3.

[9] Rtl-sdr, Accessed: 02-05-2021. [Online]. Available: https://www.rtl-sdr.com/.

[10] J. Malsbury and M. Ettus, "Simplifying fpga design with a novel network-on-chip architecture," in Proceedings of the Second Workshop on Software Radio Implementation Forum, ser. SRIF '13, Hong Kong, China: Association for Computing Machinery, 2013, pp. 45–52, ISBN: 9781450321815. DOI: 10.1145/2491246.2491251. [Online]. Available: https://doi.org/10.1145/2491246.2491251.

[11] E. Research, Rfnoc, 2019. [Online]. Available: https://kb.ettus.com/RFNoC.

[12] Xilinx, Vitis unified software platform, 2021. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis.html.

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

[14] G. Indiveri, B. Linares-Barranco, R. A. Legenstein, G. Deligeorgis, and T. Prodromakis, "Integration of nanoscale memristor synapses in neuromorphic computing architectures," CoRR, vol. abs/1302.7007, 2013. arXiv: 1302.7007. [Online]. Available: http://arxiv.org/abs/1302.7007.

[15] X. Li, F. Dong, S. Zhang, and W. Guo, "A survey on deep learning techniques in wireless signal recognition," Wireless Communications and Mobile Computing, vol. 2019, pp. 1–12, Feb. 2019. DOI: 10.1155/2019/5629572.

[16] Phung and Rhee, "A high-accuracy model average ensemble of convolutional neural networks for classification of cloud image patches on small datasets," Applied Sciences, vol. 9, p. 4500, Oct. 2019. DOI: 10.3390/app9214500.

[17] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," Journal of Machine Learning Research - Proceedings Track, vol. 9, pp. 249–256, Jan. 2010.

[18] K. He, X. Zhang, S. Ren, and J. Sun, Deep residual learning for image recognition, 2015. arXiv: 1512.03385 [cs.CV].

[19] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable fpgas," in Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '17, Monterey, California, USA: Association for Computing Machinery, 2017, pp. 15–24, ISBN: 9781450343541. DOI: 10.1145/3020078.3021741. [Online]. Available: https://doi.org/10.1145/3020078.3021741.

[20] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, Activation functions: Comparison of trends in practice and research for deep learning, 2018. arXiv: 1811.03378 [cs.LG].

[21] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," J. Mach. Learn. Res., vol. 15, no. 1, pp. 1929–1958, Jan. 2014, ISSN: 1532-4435.

[22] T. O'Shea and J. Hoydis, "An introduction to deep learning for the physical layer," IEEE Transactions on Cognitive Communications and Networking, vol. 3, no. 4, pp. 563–575, 2017. DOI: 10.1109/TCCN.2017.2758370.

[23] T. J. O'Shea, T. Roy, and T. C. Clancy, "Over-the-air deep learning based radio signal classification," IEEE Journal of Selected Topics in Signal Processing, vol. 12, no. 1, pp. 168–179, 2018. DOI: 10.1109/JSTSP.2018.2797022.

[24] T. Huynh-The, C.-H. Hua, Q.-V. Pham, and D.-S. Kim, "Mcnet: An efficient cnn architecture for robust automatic modulation classification," IEEE Communications Letters, vol. 24, no. 4, pp. 811–815, 2020. DOI: 10.1109/LCOMM.2020.2968030.

[25] D. Hong, Z. Zhang, and X. Xu, "Automatic modulation classification using recurrent neural networks," in 2017 3rd IEEE International Conference on Computer and Communications (ICCC), 2017, pp. 695–700. DOI: 10.1109/CompComm.2017.8322633.

[26] S. Rajendran, W. Meert, D. Giustiniano, V. Lenders, and S. Pollin, "Deep learning models for wireless signal classification with distributed low-cost spectrum sensors," IEEE Transactions on Cognitive Communications and Networking, vol. 4, no. 3, pp. 433–445, 2018. DOI: 10.1109/TCCN.2018.2835460.

[27] D. Adesina, J. Bassey, and L. Qian, "Robust deep radio frequency spectrum learning for future wireless communications systems," IEEE Access, vol. 8, pp. 148 528–148 540, 2020. DOI: 10.1109/ACCESS.2020.3015939.

[28] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions," ACM Comput. Surv., vol. 51, no. 3, Jun. 2018, ISSN: 0360-0300. DOI: 10.1145/3186332. [Online]. Available: https://doi.org/10.1145/3186332.

[29] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family," in 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), 2016, pp. 1–6. DOI: 10.1145/2897937.2898002.

[30] K. Abdelouahab, M. Pelcat, J. Serot, C. Bourrasset, and F. Berry, "Tactics to directly map cnn graphs on embedded fpgas," IEEE Embedded Systems Letters, pp. 1–4, 2017, ISSN: 19430663. DOI: 10.1109/LES.2017.2743247. [Online]. Available: http://ieeexplore.ieee.org/document/8015156/.

[31] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on, IEEE, 2016, pp. 1–12.

[32] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '17, ACM, 2017, pp. 65–74.

[33] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," ACM Transactions on Reconfigurable Technology and Systems (TRETS), vol. 11, no. 3, pp. 1–23, 2018.

[34] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and et al., "Fast inference of deep neural networks in fpgas for particle physics," Journal of Instrumentation, vol. 13, no. 07, P07027–P07027, Jul. 2018, ISSN: 1748-0221. DOI: 10.1088/1748-0221/13/07/p07027. [Online]. Available: http://dx.doi.org/10.1088/1748-0221/13/07/P07027.

[35] C. N. C. J. au2, A. Kuusela, S. Li, H. Zhuang, T. Aarrestad, V. Loncar, J. Ngadiuba, M. Pierini, A. A. Pol, and S. Summers, "Automatic deep heterogeneous quantization of deep neural networks for ultra low-area, low-latency inference on the edge at particle colliders," 2020. arXiv: 2006.10159 [physics.ins-det].

[36] S. Tridgell, D. Boland, P. H. Leong, and S. Siddhartha, "Real-time automatic modulation classification," in 2019 International Conference on Field-Programmable Technology (ICFPT), 2019, pp. 299–302. DOI: 10.1109/ICFPT47387.2019.00052.

[37] Z.-L. Tang, S.-M. Li, and L.-J. Yu, "Implementation of deep learning-based automatic modulation classifier on fpga sdr platform," Electronics, vol. 7, no. 7, 2018, ISSN: 2079-9292. DOI: 10.3390/electronics7070122. [Online]. Available: https://www.mdpi.com/2079-9292/7/7/122.

[38] Xilinx, Xilinx/vitis-ai. [Online]. Available: https://github.com/Xilinx/Vitis-AI.

[39] Fastmachinelearning, Fastmachinelearning/hls4ml. [Online]. Available: https://github.com/fastmachinelearning/hls4ml.

[40] Xilinx, Vitis ai user guide v1.3, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/vitis_ai/1_3/ug1414-vitis-ai.pdf.

[41] ——, Vitis ai zynq dpu v3.3 product guide, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_3/pg338-dpu.pdf.

[42] Keras. (2021). "Keras," [Online]. Available: https://keras.io/ (visited on 06/16/2020).

[43] J. Zhang, F. Franchetti, and T. M. Low, High performance zero-memory overhead direct convolutions, 2018. arXiv: 1809.10170 [cs.LG].

[44] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, Fast convolutional nets with fbfft: A gpu performance evaluation, 2015. arXiv: 1412.7580 [cs.LG].

[45] T. Aarrestad, V. Loncar, N. Ghielmetti, M. Pierini, S. Summers, J. Ngadiuba, C. Petersson, H. Linander, Y. Iiyama, G. D. Guglielmo, J. Duarte, P. Harris, D. Rankin, S. Jindariani, K. Pedro, N. Tran, M. Liu, E. Kreinar, Z. Wu, and D. Hoang, Fast convolutional neural networks on fpgas with hls4ml, 2021. arXiv: 2101.05108 [cs.LG].

[46] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2017, pp. 19–24. DOI: 10.1109/ASAP.2017.7995254.

[47] A. Devices, Ad9361 rf agile transceiver, 2016. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/AD9361.pdf.

[48] R. Seelam, "I/o design flexibility with the fpga mezzanine card (fmc)," 2009.

[49] T. Collins. (2020). "Ad-fmcomms3-ebz user guide," [Online]. Available: https://wiki.analog.com/resources/eval/user-guides/ad-fmcomms3-ebz (visited on 06/16/2020).

[50] D. Bogdan. (2021). "Ad-fmc-sdcard for zynq altera soc quick start guide," [Online]. Available: https://wiki.analog.com/resources/tools-software/linux-software/zynq_images (visited on 06/16/2020).

A

Model A Range of Values

	Frames (total of 136k frames)																													
	0 – 10k		10k – 20k		20k – 30k		30k – 40k		40k – 50k		50k – 60k		60k – 70k		70k – 80k		80k – 90k		90k – 100k		100k – 110k		110k – 120k		120k – 130k		130k – 136k	
Layer	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
Input	-77.52	74.4380	-163.0	157.53	-168.0	179.11	- 37.62	40.43	- 37.25	38.79	- 43.59	38.17	-91.96	93.72	- 147.02	134.57	- 278.60	313.41	- 314.71	323.88	- 159.93	169.02	- 239.10	252.65	- 463.38	482.97	- 490.73	476.74		
Conv1d	- 119.66	118.60	- 253.87	232.95	- 246.04	236.34	- 52.60	56.68	- 57.10	53.02	- 61.88	55.11	- 133.16	149.89	- 227.60	224.59	- 448.90	459.19	- 442.59	429.90	- 243.55	238.89	- 365.36	361.20	- 665.31	664.95	- 715.91	731.30		
ReLU	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844		
Conv1d	-5.94	9.13	-6.59	9.76	-6.31	9.85	-5.52	7.00	-5.80	7.49	-5.42	6.80	-6.25	9.34	-6.34	9.78	-6.57	9.91	-6.72	10.23	-6.56	9.65	-6.60	10.03	-6.92	10.06	-6.95	10.09		
ReLU	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844		
Conv1d	-2.98	7.40	-3.08	7.54	-3.20	7.59	-2.79	6.57	-2.78	6.64	-2.81	6.79	-3.03	7.53	-3.39	7.64	-3.29	7.66	-3.32	7.69	-3.38	7.51	-3.30	7.65	-3.30	7.64	-3.41	7.73		
ReLU	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844		
Conv1d	-6.14	4.31	-6.89	4.39	-7.30	4.44	-5.18	4.87	-4.65	4.71	-4.64	4.74	-6.65	4.93	-7.05	4.90	-7.58	4.58	-7.65	4.85	-7.07	4.833	-7.35	4.81	-7.78	4.48	-7.76	4.35		
ReLU	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844		
Conv1d	-3.90	4.41	-4.53	3.53	-4.65	4.08	-3.30	4.53	-3.07	4.62	-3.35	4.57	-4.12	4.55	-4.71	3.91	-5.32	3.32	-5.38	3.76	- 4.9673	3.44	-5.34	3.38	-5.30	2.75	-5.26	2.39		
ReLU	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844	0.0	0.9844		