Accelerating Convolutional Neural Network Inference on Digital Signal Processor

Uula Ollila

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 28.02.2022

Supervisor

Prof. Antti Hannukainen

Advisor

M.Sc. Marko Hassinen

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Uula Ollila	
Title Accelerating Convolutional Neural Network Inference on Digital Signal Processor	
Degree programme Master's Programme in Mathematics and Operations Research	
Major Applied mathematics	**Code of major** SCI3053
Supervisor Prof. Antti Hannukainen	
Advisor M.Sc. Marko Hassinen	
Date 28.02.2022	**Number of pages** 95+2 **Language** English

Abstract

The ongoing deployment of 5G NR is to bring a completely new wave of technology and revolutionize wireless data transfer. The new standard will provide improved data rates, lower latency, better energy efficiency and larger capacity for connected devices. New performance requirements and increased flexibility challenge conventional physical layer (L1) solutions and boost research of new algorithmic approaches. One proposed new approach is deep learning (DL), which in recent studies have shown to provide a feasible alternative for existing algorithms in terms of computational load and accuracy. So far, neural networks (NNs) based solutions are not available in any commercial 5G physical layer product. They employ highly specialized hardware, which poses challenges when efficient NN inference implementations are considered.

Convolutional neural networks (CNNs) are an essential subtype of NNs and have recently been introduced to replace several L1 processing blocks. This work aims to develop a comprehensive framework that enables accelerating CNN inferences on CEVA-XC4500 DSP, a current state of the art in L1 processing. The basic idea of the framework is presented earlier for multilayer perceptrons (MLPs), and this work elaborates the idea for 1-D CNNs. Essential parts of CNNs are covered, including convolution layers, pooling layers and some activation functions. The DSP implementation is optimized with the single instruction, multiple data (SIMD) intrinsics that the DSP offers broad support. Since the DSP is designed for fixed-point arithmetic, the framework includes a procedure for quantizing the network parameters as a part of the framework's offline preprocessing part.

Implementation's real-time performance is evaluated by recording the processor cycle counts of running inference with five different CNN models. The results are recorded in a software simulator that simulates the operation of the DSP in a cycle-accurate manner. The implementation's accuracy and quantization procedure are also evaluated. The results show that the target DSP can accelerate optimized CNN inferences effectively. Compared to the unoptimized reference implementation, speedups ranging from 6.5 to 26.8 were observed. Most performance gains originate from optimizing the convolution layers, typically computationally the heaviest part of CNNs. Clear benefits can also be observed from optimizing pooling layers and activation functions.

Keywords 5G, convolutional neural network, digital signal processor, neural network inference, hardware acceleration, SIMD

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Uula Ollila		
Työn nimi Konvoluutioneuroverkkoinferenssin kiihdyttäminen digitaalisella signaaliprosessorilla		
Koulutusohjelma Matematiikan ja operaatiotutkimuksen maisteriohjelma		
Pääaine Sovellettu matematiikka		**Pääaineen koodi** SCI3053
Työn valvoja Prof. Antti Hannukainen		
Työn ohjaaja FM Marko Hassinen		
Päivämäärä 28.02.2022	**Sivumäärä** 95+2	**Kieli** Englanti

Tiivistelmä

Meneillään oleva 5G NR-standardin käyttöönotto on tuomassa mukanaan täysin uutta teknologiaa, joka on muuttamassa pysyvästi langatonta tiedonsiirtoa. Uusi standardi mahdollistaa nopeamman tiedonsiirron, pienemmän viiveen, paremman energiatehokkuuden ja suuremman kapasiteetin yhdistetyille laitteille. Uudet suorituskykyvaatimukset ja verkkojen joustava rakenne haastavat perinteiset fyysisen kerroksen (L1) ratkaisut ja vauhdittavat uusien algoritmisten lähestymistapojen tutkimusta. Yksi ehdotettu lähestymistapa on syväoppiminen (deep learning, DL), joka on hiljattain osoittautunut käyttökelpoiseksi vaihtoehdoksi nykyisille algoritmeille laskentakuorman ja tarkkuuden näkökulmasta. Toistaiseksi neuroverkkoihin perustuvia ratkaisuja ei ole saatavilla missään kaupallisessa fyysisen kerroksen tuotteessa. Niissä hyödynnetään erikoistuneita sulautettuja järjestelmiä, mikä asettaa tiettyjä haasteita tehokkaiden neuroverkkoinferenssien näkökulmasta.

Konvoluutioneuroverkot (CNN) ovat neuroverkkojen alatyyppi, joita on vastikään ehdotettu korvaajiksi tietyille fyysisen kerroksen prosessointiosille. Tässä työssä pyritään kehittämään kattava ohjelmistokehys, jonka avulla konvoluutioneuroverkkoinferenssejä voidaan kiihdyttää CEVA-XC4500 digitaalisella signaaliprosessorilla (DSP). Ohjelmistokehyksen perusideaa on hyödynnetty aiemmin monikerroksisilla perseptroniverkoilla, ja tässä työssä kehitetään ideaa edelleen soveltumaan yksiulotteisille CNN-malleille. Toteutus optimoidaan hyödyntäen DSP:n laajoja vektorilaskentaominaisuuksia. DSP on suunniteltu kiinteän pisteen aritmetiikkaa varten, joten ohjelmistokehys toteuttaa mallien parametrien kvantisoinnin kiintopisteluvuiksi osana verkon offline-esikäsittelyä.

Toteutuksen suorituskykyä arvioidaan mittaamalla kuluneiden prosessorisyklien määrä viidellä eri CNN-inferenssillä. Mittaukset suoritetaan syklitarkalla ohjelmistosimulaattorilla. Lisäksi tarkastellaan inferenssin kvantisointia ja tarkkuutta. Tulokset osoittavat, että DSP voi kiihdyttää optimoituja inferenssejä hyvin tehokkaasti. Optimoimattomaan referenssitoteutukseen verrattuna havaittiin 6.5-26.8 kertaisia nopeuslisäyksiä. Suurin osa suorituskyvyn kasvusta tapahtuu konvoluutiokerroksissa, joka on tyypillisesti CNN-mallien laskennallisesti raskain osa. Selkeää hyötyä saavutetaan myös optimoimalla koontikerroksia ja aktivaatiofunktioita.

Avainsanat 5G, konvoluutioneuroverkko, digitaalinen signaaliprosessori, neuroverkkoinferenssi, laitteistokiihdyttäminen, SIMD

Preface

I have done easier things in my life than writing this thesis. It was a challenging but, in the end, also so rewarding experience. Sometimes during this process, I felt like I was a living example of the paradox of Achilles and the tortoise. Every time I thought I was close to the finish line, I realized there was always something more to do. I'm so glad that it is now over.

At this point, it is time to express gratitude to everyone who has supported me in my studies. I must thank my supervisor Antti and advisor Marko for all the great advice and feedback in this thesis process. I admire your patience. I also want to thank my co-workers at Nokia for the great discussions during this time. I am grateful to my line managers, Henna Koskenniemi and Mikko Volanen, for providing me with the opportunity to do this thesis as a part of your team. It was so truly amazing to constantly learn new things in such a challenging environment.

Most importantly, I want to thank my family for always supporting me in everything I do. And Unna, I express my endless gratitude to you. It was sensational to share this journey with you. No one else could cheer me up better than you do when it feels that there is no sense in absolutely anything.

Finishing this thesis brings an end to the most memorable part of my life. These years at Otaniemi and Aalto University gave me so much more than I could ever imagine back in 2015 when I jumped that night train at Oulu railway station. Studying has been so much more than just studying for me. I'd like to thank The Guild of Physics, especially boards 2017 and 2019. It is crazy to think about what kind of amazing stuff I could do with you. I also want to thank all my friends I was privileged to get to know during these six and a half years. These people around me are so much more valuable in my life than any degree.

Y161b, Otakaari 1, Espoo 28.02.2022

Uula Ollila

Contents

Abbreviations

1D	One dimensional
5G NR	Fifth generation new radio
AI	Artificial intelligence
API	Application programming interface
ASIC	Application specific integrated circuit
BP	Backpropagation
BPSK	Binary phase-shift keying
CNN	Convolutional neural network
DAAU	Data address and arithmetic unit
DFT	Discrete Fourier transform
DL	Deep learning
DMA	Direct memory access
DMRS	Demodulation reference signal
DNN	Deep neural network
DSP	Digital signal processor
ECG	Electrocardiogram
eMBB	Enhanced Mobile Broadband
FFT	Fast Fourier transform
FNN	Feedforward neural network
FPGA	Field-programmable gate array
GCU	General computation unit
GEMM	General matrix multiply
GPP	General purpose processor
GPU	Graphics processing unit
IDE	Integrated development environment
IDFT	Inverse discrete Fourier transform
IEEE	Institute of electrical and electronics engineers
ILP	Instruction level parallelism
IR	Intermediate representation
L1	Physical layer
LS	Least squares
LTE	Long term evolution
MAC	Multiply-accumulate operation
MEA	Mean absolute error
MIMO	Multiple input multiple output

ML	Machine learning
MLP	Multilayer perceptron
mMTC	massive Machine Type Communications
MPN	McCulloch-Pitts neuron
MSE	Mean squared error
NMSE	Normalized mean squared error
NN	Neural network
ONNX	Open neural network exchange
OSI	Open Systems Interconnection
PCU	Program control unit
PRACH	Physical random access channel
PUSCH	Physical uplink shared channel
QAM	Quadrature amplitude modulation
QPSK	Quadrature phase-shift keying
ReLU	Rectified Linear Unit
RISC	Reduced instruction set computer
RNN	Recurrent neural network
SGD	Stochastic gradient descend
SIMD	Single instruction, multiple data
SNR	Signal to noise ratio
SoC	System on a chip
TOA	Time of arrival
TPU	Tensor processing unit
URLLC	Ultra-reliable and low latency communications
VA	Vector arithmetic unit
VB	Vector bit manipulation unit
VM	Vector move and pack unit
VCU	Vector computation unit
VLIW	Very long instruction word
VRF	Vector register file

Chapter 1

Introduction

The current deployment of fifth-generation (5G) mobile networks (MN) is due to bring a completely new wave of technology and revolutionize everyday lives. 5G New Radio (NR) will provide improved data rates, lower latency, better energy efficiency and larger capacity for connected devices. The standard is designed to be highly flexible to support different use cases and applications. Compared to previous standards, improvements of 5G NR will not be small but rather a significant leap into a new level in performance [1].

New performance requirements and increased network flexibility pose new engineering challenges by significantly increasing network complexity. It challenges conventional information theory solutions and speeds up research of new algorithmic approaches. Deep learning (DL) and neural networks (NN) are included in the proposed approaches. NNs are already a current state of the art in many fields outside signal processing, such as computer vision and speech recognition [2]. Recent studies show that in terms of computational load and accuracy, neural networks can provide a feasible alternative for existing signal processing algorithms [3]. In the physical layer context, the proposed DL based applications include substituting individual blocks [4, 5, 6] as well as replacing the entire transceiver with autoencoder [7, 8]. Despite active research in simulated environments, DL-based solutions are not yet active in any commercially available 5G base station.

Most of the real-time processing in the physical layer is done using highly specialized hardware typically implemented in a System on Chip (SoC). The SoC can include multiple digital signal processors (DSPs) and application-specific hardware accelerators. Both can be used to accelerate different signal processing algorithms. However, either of them is rarely designed for operating with floating-point tensors, which are the core of almost all modern DL models. Those models are optimally processed using specific DL processing hardware, which is not yet very common in current SoCs operating in millions of base stations for years. Modifying, extending or replacing existing hardware is difficult and expensive; thus, efficient NN implementations for existing hardware can be beneficial. Conventional DSPs are optimized for multiply-accumulate (MAC) operations, which form the core of most signal processing algorithms. Neural networks share this fundamental similarity; the multiplications and additions between inputs and network parameters are at the core of neural network inference.

This work aims to develop a comprehensive framework that enables the efficient accelerating of convolutional neural networks (CNN) inferences on CEVA-XC4500 DSP. The DSP is part of a Nokia ReefShark chipset, which is Nokia's in-house developed SoC for baseband products. The SoC design is based on 3GPP specifications for LTE and 5G NR network standards. ReefShark is delivered as a physical layer processing plug-in unit for the Nokia AirScale baseband module, which serves as a software-defined system module supporting all radio technologies from 2G to 5G.

The basic idea of the framework in this work is not that novel. In [9, 10] accelerating multilayer perceptrons (MLPs) in the same target DSP were thoroughly studied. This work elaborates this idea to support one-dimensional CNNs. This work presents implementations for essential parts of a CNN, including convolution layers, pooling layers and necessary activation functions. The implementation is optimized by vectorizing all components of the inference algorithm using the single instruction, multiple data (SIMD) intrinsics the DSP supports. Since the target DSP is designed essentially for fixed-point arithmetic, the framework also includes a procedure for quantizing the network parameters of the floating-point models into compatible fixed-point formats as a part of the offline preprocessing phase.

The real-time performance of DSP implementation is evaluated by recording the processor cycle counts of running inference with five different CNN models. The cycle counts were recorded in a software simulator, which simulates the operation of the DSP in a cycle-accurate manner. It does not consider all effects that affect the overall performance of the actual SoC, such as data transfer between the main memory and internal memory of the DSP. Thus, the measured cycle counts in this work provide only lower bound estimates for the cycle counts on the SoC hardware.

1.1 Structure

This work is divided into nine Chapters, which are organized as follows. Chapter 2 discusses the wireless physical layer providing the basics of wireless communication systems and introducing the latest approaches to improving physical layer communications with deep learning. Chapter 3 provides the background in neural networks and deep learning. It also looks into modern deep learning frameworks, compilers and methods for quantization. Chapter 4 introduces convolution as a mathematical operation and provides background on convolutional neural networks (CNNs) and the key concepts related to them. Chapter 5 covers the fundamental concepts of digital signal processor (DSP) design and discusses the features and constraints of the target SoC. The focus is mainly on the DSP for which the implementation is designed. Chapter 6 presents in detail the implemented inference framework, including the offline preprocessing and the real-time DSP inference. Chapter 7 introduces the experimental setup and metrics that are applied in the evaluation of the performance and the correctness of the DSP inference. The obtained results are presented and analyzed in Chapter 8. Finally, the work is concluded in Chapter 9.

Chapter 2

Physical layer in wireless communication

The Open Systems Interconnection (OSI) model is a conceptual model that characterizes communications over a network between computer systems. It aims to provide an interoperative way to describe communication between diverse computer systems using standardized protocols. The OSI model is widely used as it visualizes how networks operate and helps isolate and troubleshoot network problems. The model partitions data flow in a communication system into seven abstraction layers. Each intermediate abstraction layer serves certain functionalities to the above layer and utilizes functionalities provided by the layer below it [11].

The first and lowest of the seven layers in the OSI model is the physical layer (Layer 1 or L1). L1 enables data transmission over a physical medium, such as optical signals in a fibre or radio waves over the air. The physical layer defines the means of transmitting a raw bitstream over a physical connection between devices in a network by providing an electrical, mechanical, and procedural interface to the transmission medium. As it is a fundamental layer under all higher-level layers, the physical layer has a crucial role in establishing reliable communication and achieving proper transmission rates.

The neural network inference framework developed in this work objects to accelerating inferences related to wireless L1 processing in 5G mobile networks. The target hardware is a state of the art solution for physical layer processing in 5G base station products. This Chapter provides a background on the wireless physical layer to understand the context of the possible use cases. In the first Section, the basic principles of wireless communication are discussed. The second Section introduces some recent applications and advantages of exploiting deep learning to improve physical layer processing. After that, two use cases are introduced shortly, for which deep learning solutions have been developed in previous works. These cases are realistic examples in which the inference framework developed in this work could be used in the physical layer of 5G mobile networks.

2.1 Wireless communication

In modern society, many digital systems rely on different wireless communication methods. In all wireless systems, the basic idea is that the data is transmitted from one location to another over a wireless medium, typically air. A conventional wireless communication system consists of a transmitter, a radio channel, and a receiver in the basic form. Transmitter sends the data over a transmission medium, i.e. the channel, and the receiver receives the transmitted signal and tries to reconstruct the original content of the data. The data can be information from analogue voice signals to bitstreams describing files on digital systems.

2.1.1 Channel model

The transmitter and the receiver communicate over a radio channel in wireless communication. The transmitted signal arrives at the receiver through only one path in the ideal case. However, this is rarely the case in the real communication environment. The transmitted signal is affected by many physical effects. In addition to the direct signal, the receiver receives reflected, scattered and diffracted as the signal faces different surfaces and objects on its way. Due to this multipath propagation, the different components of the same signal can reach the receiver at different arrival times. This phenomenon is illustrated in Figure 2.1. In addition to multipath propagation, the signal is distorted by the atmospheric noise and signals from other transmitters. The signal power is attenuated as a function of the transmission distance. The radio channel is also only rarely static. Transmitters and other objects in the environment can move, and for example, weather can contribute significantly to the channel conditions.

The wireless system design must consider these distorting effects in the radio channel to enable effective and reliable wireless communication. In a system where the transmitter and receiver consist of one antenna each, the channel can be formulated

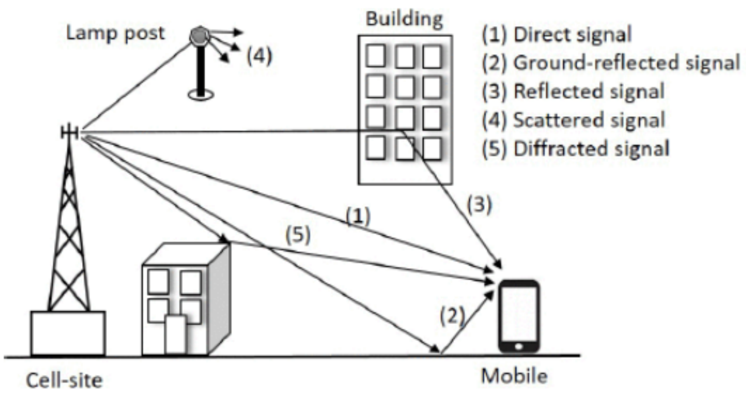


Figure 2.1: Example of physical effects affecting the channel conditions between transmitter (base station) and receiver (mobile phone) in urban environment [12].

as

$$y = Hx + n, \tag{2.1}$$

where $\boldsymbol{y} \in \mathbb{C}^n$ and $\boldsymbol{x} \in \mathbb{C}^n$ is the received and transmitted signal vectors, respectively, $\boldsymbol{H} \in \mathbb{C}^{n\times n}$ is the channel matrix and $\boldsymbol{n} \in \mathbb{C}^n$ is the noise vector. To be able to adapt the transmission to current channel conditions, many mathematical models have been developed to estimate the channel properties, i.e. to provide accurate estimates for the channel matrix $\boldsymbol{H}$. Channel estimation can apply statistical methods and measure impulse response when some known data is transmitted, which is known both at the transmitter, and the receiver [13]. Recently, deep learning-based solutions have been widely studied in channel condition estimation [7, 6, 14]. These approaches are discussed more in Section 2.2.

2.1.2 Transceiver

In most wireless systems, devices are typically a combination of both a transmitter and a receiver and are thus called transceivers commonly. Traditionally transceiver is described as a chain of functionality blocks, each of which performs a specific processing step for the transmitted or received signal. Figure 2.2 illustrates a simplified diagram of the functional blocks of a transceiver. Each block executes a solidly founded and isolated function, which are highly optimized based on long-term research in signal processing [7].

In the transmitter end (Tx), the data is first fed to source coding that uses a priori knowledge of the data properties to minimize redundancy. The process compresses the data and thus reduces the amount of data to be transmitted. If the application requires, the source coding can be combined with encryption to prevent access from unauthorized parties. After source coding, the data is channel encoded to add redundancy and make the signal more robust against errors during transmission over the air. There are many different approaches for the encoding, and the approach can be adjusted based on the channel conditions to optimize the error resistivity [13].

After channel encoding, the obtained codeword is mapped from a sequence of

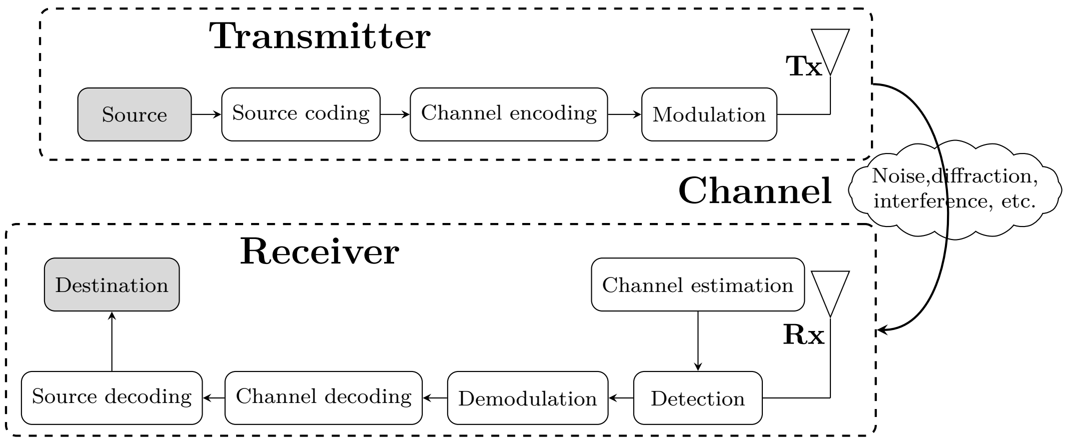


Figure 2.2: Simplified diagram of physical layer functionalities in a transceiver [3].

bits into a vector $\boldsymbol{x}$ of complex-valued symbols through a process called modulation. Modulation combines the carrier wave with an input signal that contains the information to enable data transmission with low energy. The most common modulation schemes include binary phase-shift keying (BPSK), quadrature phase-shift keying (QPSK) or quadrature amplitude modulation (QAM). The complex-valued symbols obtained with modulation are then sent over a radio channel, which attenuates, distorts and adds noise to the signal [13].

At the Receiver end (Rx), the signal is received with one or more antennas, and basically, the same operations are performed in the opposite direction. In the first phase, the received symbols are detected from the signal, and the channel between transmitter and receiver is estimated using known reference symbols. The channel estimate is applied to adapt transmissions to different channel conditions and improve transmission quality. After detection, the vector of received symbols are fed to the demodulator to obtain soft bits, a vector of real values between 0 and 1. The soft bits are passed to the channel decoder, which estimates the source coded bits. The algorithm used in channel decoding depends on the chosen coding and modulation schemes. Finally, the estimated bits are fed to the source decoding, which tries to recover the original data [13].

2.2 Deep learning in physical layer processing

In recent years Deep Learning (DL) has been proposed as a potential approach to improve the physical layer performance of wireless transceivers. It has already been successfully applied elsewhere in wireless communications, despite that the current networks are well-known and effective conventional algorithms exist for many tasks [15]. Because the physical layer strongly assesses the foundation of the overall network-level performance, the DL-based L1 processing is a promising technology. There are some reasons why DL could provide gains over the current state of the art in L1 processing [7]:

- Most L1 signal processing algorithms have long-term statistics and information theory foundations. They are often justified optimal with tractable mathematical models, generally linear, stationary, and normally distributed. However, in practice, systems have many imperfections and non-linearities (e.g. timing offset, finite resolution) that such models can only approximately collect. Thus, a DL-based physical layer solution or processing block that do not require a mathematically tractable model might be a justified choice. It can be optimized for a certain hardware configuration and channel conditions enabling better optimization for such imperfections.

- The leading design principle in the physical layer is to split the signal processing into a chain of multiple independent functionality blocks. Each block performs an isolated functionality such as channel encoding, symbol detection, modulation or channel estimation. This approach produces efficient, accomplished, and well controllable processing chains, but it is known that individually tuned processing blocks provide only sub-optimal end-to-end performance [16].

- It has been shown that neural networks are universal function approximators [17] and have excellent algorithmic learning capabilities even in complex channel conditions [7]. Neural networks are possible to execute the highly parallelized on different architectures [18] and combined with limited-precision data types [19]. Thus, there is evidence that DL-based solutions could save energy and computational costs compared to their manual counterparts originating from the information-theoretic approach. They might reduce the hardware design costs by decreasing the need for specialized signal processing hardware.

- Effective handling of large data sets is essential for all DL. It requires powerful hardware and software libraries. Massively parallel processing architectures with distributed memory architectures, such as graphic processing units (GPUs), are available for model development and increasingly specialized chips and compilers for NN inferences [20, 21]. Only since very recently have these tools been cheaply and widely available.

Recent studies on wireless physical layers have proposed alternative approaches to augment or replace certain parts of the conventional processing chain. In [7] neural network was applied in modulation recognition, and the obtained solution outperformed the conventional methods based on expert knowledge. Deep learning in channel encoding and decoding has been studied, for example, in [22] and [23]. DL-based end-to-end solutions, in which both the transmitter and receiver are learned simultaneously from the data without any prespecified modulation schemes or waveforms, are the utmost case and have also been studied in the literature, for example, in [7] and [8].

2.2.1 Use cases in 5G physical layer

The fifth-generation New Radio (5G NR) is the latest mobile network data transferring standard, first launched on a large scale in April 2019. Compared to the legacy 4G/LTE standards, the 5G standard offers advanced capabilities that enable higher data rates, reduced latencies, better energy efficiency and large connection density. This new capacity is implemented with a wide range of spectrum bands, Massive Multiple Input Multiple Output (MIMO) antennas combined with beamforming and flexible network configuration and slicing capabilities. Because 5G networks are rolled out while LTE networks are still fully-operational, tight interwork of these two networks using dual connectivity is one of the essential requirements for the new standard [1].

The three primary 5G development scenarios have guided the definition of network requirements: enhanced Mobile Broadband (eMBB), Ultra-Reliable and Low Latency Communications (URLLC) and massive Machine Type Communications (mMTC). eMMB is the most beneficial for a regular mobile network user. It does not point to a specific use case but rather a seamless user experience and superior data transfer rates. URLLC relates use cases where low latency service is essential or life-critical such as autonomous vehicles or remote surgery. mMTC scenario targets networks that connect many devices with low data rates and non-delay sensitive transmissions. Typically these devices are low cost and have long battery life, such as different IoT devices and sensors [1].

The physical layer of 5G NR is a tremendous topic, and covering its entirety even in a superficial manner is out of the scope of this work. Due to this complexity, only a high-level overview of two different functionalities in the 5G physical layer are provided. Recent works [14] and [5] have introduced deep learning solutions to use cases related to these functionalities. These convolutional neural network (CNN) based models are used to evaluate the inference framework implementation's correctness and performance in this work.

Autoencoder in pilot-based channel estimation

Channel estimation is a method for measuring the properties of the medium between the transmitter and the receiver. This knowledge can be obtained in wireless networks by sounding the channel, i.e. transmitting known reference signals and measuring the impulse response. In 5G NR mobile networks, the Demodulation reference signal (DMRS) is one of the standard pilot signals applied in channel estimation.

In [14], a CNN-based Autoencoder was developed to estimate wireless channel based on the DMRS pilot signal. The work aimed to optimize the channel estimation block in the 5G NR Physical Uplink Shared Channel (PUSCH). The autoencoder was trained under multiple channel conditions and a wide range of signal to noise ratio (SNR) values. The high-quality data for the model training was generated in a 5G-compliant link-level simulator applying a specific 5G Data Generation Tool. The training procedure trained the model to reach a level of abstraction where the model estimates the channel accurately in various channel conditions with different SNR values.

The solution was compared with the traditional Least Squares (LS) channel estimation method. The developed autoencoder surpassed the conventional method, proving to be a promising solution for channel estimation in the 5G physical layer. The work stated that the autoencoder achieved an average of up to 90 % less estimation error than the conventional method under certain conditions over all the channel conditions and SNR values. The encoder part of the autoencoder in [14] is used as one of the models to evaluate the inference framework developed in this work. The model is discussed in Chapter 7.

Preamble Detection and time of arrival estimation with deep learning

Accurate Time of Arrival (TOA) estimation and reliable signal detection is crucial in 5G NR. In the initial access process, signal detection is used for physical random access channel (PRACH) preamble detection, and TOA Estimation is used for signal synchronization. Conventionally TOA estimation is done using correlation-based methods, where the correlation between all possible preamble sequences and the received signal is computed. Due to multipath propagation and noise, this method for TOA estimation may not be accurate. The other is to use the so-called template matching method based on convex programming. It provides better accuracy than the correlation method, but the high computational complexity prohibits its application in real-time.

In [5], a CNN-based framework was proposed for preamble detection and TOA estimation without the need of knowing the exact transmitted waveform. The solution was evaluated with extensive simulations on synthetic and actual measured data. The results showed that the proposed method improves prediction accuracy roughly three times higher while keeping the same computational complexity as the correlation method. The method also provides 1000x computational reduction compared to the template matching method without loss of accuracy. The CNN in [5] is used as one of the models to evaluate the inference framework developed in this work. The model is discussed more in Chapter 7.

Chapter 3

Neural networks

Artificial intelligence (AI) is a thriving field of research that is applied successfully in many practical applications, such as automating routine tasks, speech recognition, and classifying images. AI systems are characterized by their ability to collect knowledge from data without being explicitly programmed. This capability is commonly recognized as machine learning (ML). It enables computers to deal with problems that require knowledge from the real world and make decisions. However, conventional machine learning algorithms heavily depend on input data representation. Recognizing features from representations requires specific techniques that are typically time-consuming and need much manual work. An approach called representation learning can overcome this manual feature engineering. In this approach, machine learning is applied to extract the mapping from representation to output and the representation itself. It allows the system to adapt to new tasks with minimal human intervention and often output more valuable results than hand-designed representations [2].

Deep learning (DL), part of the family of machine learning methods, covers a selection of methods that utilize representation learning capabilities. The basic idea is to combine simple machine learning models into more complex ones. Most DL methods are based on neural networks (NNs), which have layered structures that progressively extract higher-level features from the input. The idea is to store long-term experimental knowledge and utilize it later. Mathematically, they can be described as functions that map a set of input values to output values. The first concepts were developed already in the 1940s, but it was relatively unpopular for decades [24]. Recently, it has seen a massive increase in research activity and application proposals due to two key reasons. Firstly, the increasing digitization of society has genuinely increased the amount of accessible training data. Secondly, distributed computing and new powerful hardware have expanded available computational resources significantly. It has enabled the training of more complex NNs, which have been applied to a much broader set of applications with increasingly accurate results [25, 26].

This Chapter covers the basics of neural networks. The following four sections cover the relevant building blocks and design principles. Because the target network architecture in this work is a convolutional neural network, they are covered in more

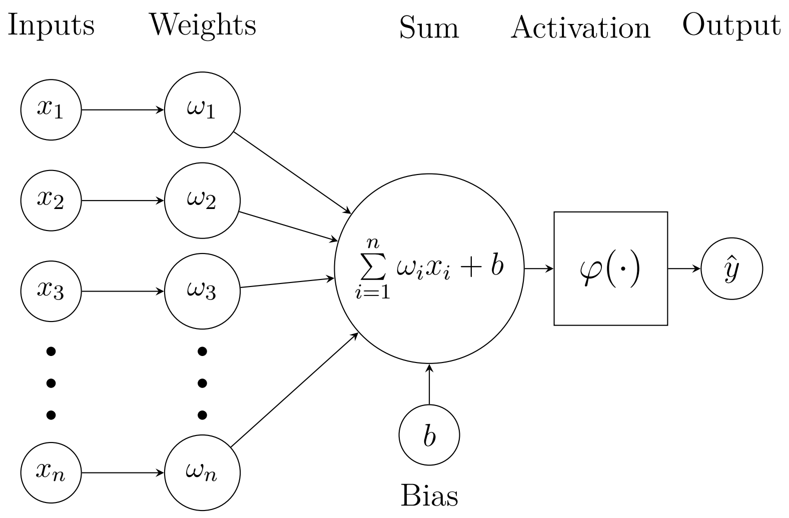


Figure 3.1: Logical view of a perceptron.

detail in Chapter 4.

3.1 Neuron

A neuron is the fundamental processing element of any NN. The first mathematical model, McCulloch-Pitts neuron (MPN), was introduced back in 1943 [24]. It is a highly simplified model of a real biological neuron, and with binary output and fixed-parameter values, it involves no learning capabilities. In the 1950s, Rosenblatt introduced a new, more flexible neuron model based on the idea of MPN. This so-called Rosenblatt's perceptron is the core building block of all modern NNs [27]. It acts essentially as a binary classifier operating in a supervised manner. A single perceptron is the simplest possible neural network model.

The basic operating logic of a perceptron is visualized in Figure 3.1. Perceptron's input consists of a vector $\boldsymbol{x} \in \mathbb{R}^n$, and the local field $y \in \mathbb{R}$ is computed as a weighted sum of the input

$$y = \boldsymbol{\omega} \cdot \boldsymbol{x} + b. \tag{3.1}$$

Vector of synaptic weights $\boldsymbol{\omega} \in \mathbb{R}^n$ and scalar bias term $b \in \mathbb{R}$ are the internal parameter of perceptron. Bias enables perceptron to model functions that do not pass through the origin, increasing the flexibility of the perceptron model. To obtain the final output $\hat{y}$, an activation function $\varphi : \mathbb{R} \to \mathbb{R}$ is applied to the local field of the neuron.

$$\hat{y} = \varphi(y) \tag{3.2}$$

Activation functions add non-linearity to the network and, in some cases, limit the magnitude of the neuron output. The activation function can be any non-linear and differentiable function. Multilayer NNs applying only linear activation functions can be reduced to a single layer of perceptrons. Therefore non-linear activation is required to be able to learn non-linear behaviour. The most commonly applied activation functions that are also used in this work are:

1. **Rectified Linear Unit (ReLU)** is very simple and nowdays the most frequently applied activation function. It is continuous, piecewise linear function defined as
$$\varphi_{\mathrm{ReLU}}(y) = \max(y, 0). \tag{3.3}$$
It is linear with positive input values, and with negative input values, it outputs zero. ReLU is differentiable everywhere except origin. The constant derivative of 1 on the positive side results in relatively large gradients during training, which enables a good training speed compared to other non-linear activation functions [26, 2].

   Leaky ReLU is a special version of standard ReLU, where a small positive slope is applied to the negative values instead of constant zero output:
$$\varphi_{\mathrm{Leaky}}(y) = \begin{cases} y & \text{if } y > 0 \\ \alpha y & \text{otherwise.} \end{cases} \tag{3.4}$$
Typically, the slope coefficient α is a minimal and not learnable parameter determined before the network training phase. Leaky ReLU can be applied in cases where the network suffers from a sparse activation problem, i.e. standard ReLU would output only a few non-zero activations.

2. **Sigmoid function** produces output activations compressed to the interval $]0, 1[$ and is defined as
$$\varphi_{\mathrm{sigmoid}}(y) = \frac{1}{1 + e^{-y}} \tag{3.5}$$
It is mainly applied to the output layer neurons as the output can be interpreted as a probability distribution. Sigmoid can also be applied to the hidden layers' neurons, limiting the magnitude of output values. Sigmoid is continuously differentiable and has a non-zero derivative at each point, ensuring that gradient-based learning algorithms never get completely stuck. The gradient is close to zero for most of the domain, and the function is sensitive only when the input value is close to zero. Thus, Sigmoid can suffer from gradient saturation when layer input is very large or small, making network training slow.

3. **Softmax function** $\varphi_{\mathrm{softmax}} : \mathbb{R}^n \to \mathbb{R}^n$ is generalization of a standard sigmoid function defined as
$$\varphi_{\mathrm{softmax}}(\boldsymbol{y})_i = \frac{e^{y_i}}{\sum\limits_{j=1}^{n} e^{y_j}}, \tag{3.6}$$
where the denominator performs normalization such that the function output elements sums to one. The output can be interpreted as a probability distribution of a discrete variable. Thus, softmax is typically applied on the output layer neurons of a classifier representing the probability distribution over n different classes.

3.2 Connection architecture

A NN consisting of a single perceptron can perform only simple tasks such as binary classification, but more versatile NNs are required for more complex applications. For example, when the output of a network is a vector, multiple neurons are required. Multiple neurons are organized to form larger computation units called layers. Multilayer perceptron (MLP) is a basic neural network model with multiple layers. It contains one input layer, one output layer, and at least one hidden layer. The basic principle of MLP is illustrated in Figure 3.2.

In MLP, a layer of neurons performs a mapping $f : \mathbb{R}^n \to \mathbb{R}^m$ to obtain a vector containing local fields of the neurons

$$\boldsymbol{a} = f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}. \tag{3.7}$$

Rows of weight matrix $\boldsymbol{W} \in \mathbb{R}^{m\times n}$ contain weight vectors of individual neurons, and bias vector $\boldsymbol{b} \in \mathbb{R}^m$ contains bias terms of individual neurons. All neurons except the ones in the input layer apply non-linear activation functions as described in Section 3.1. The activation function $\varphi : \mathbb{R}^m \to \mathbb{R}^m$ is applied element-wise to the vector containing local fields of the neurons on the layer

$$\boldsymbol{h} = \varphi(\boldsymbol{a}) = \varphi(f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b})), \tag{3.8}$$

where $\boldsymbol{h} \in \mathbb{R}^m$ is the layer output also called as activation. NN can consist of multiple sequential layers of neurons that apply different activation functions.

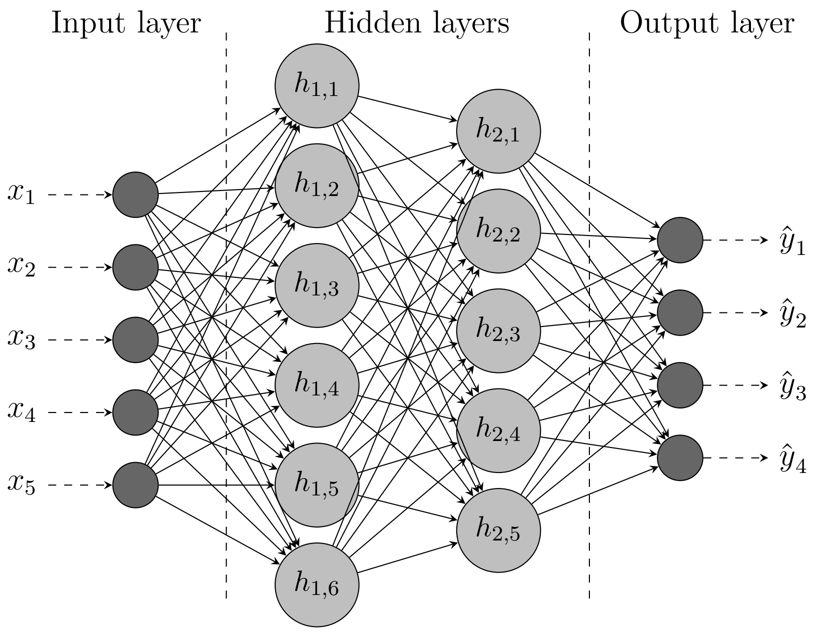


Figure 3.2: An example of multilayer perceptron (MLP) network. The network receives $\boldsymbol{x} \in \mathbb{R}^5$ as an input, and maps it to an output $\hat{\boldsymbol{y}} \in \mathbb{R}^4$. The network has a total of 11 hidden neurons divided into two layers.

MLP containing at least one hidden layer of sufficiently many neurons can operate as a universal function approximator as proved by Hornik et al. [17]. It means that such NN can approximate any functions up to arbitrary accuracy when certain mathematical assumptions are met. For example, any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is a general enough assumption for most practical applications [2]. The universal approximation theorem does not offer information about how many neurons are required that NN can provide adequate function approximations. In addition, the theorem states that a large enough MLP can represent a function but does not guarantee that the training algorithm can learn that function. The learning algorithm may not be able to find the correct set of parameters, or it can choose the wrong model as a result of overfitting [2].

Connection architecture defines the connections between neurons in different layers. MLP is an example of a fully connected network, i.e. each neuron in one layer is connected to every neuron in the next layer with a specific weight that describes the strength of the connection. In partially connected layers, neurons are connected to only a limited number of neurons in the subsequent layer. A neural network can contain simultaneously layers with both types of neurons, partially connected and fully connected. A convolutional neural network (CNN) is a typical example of a network that contains partially connected layers. It is the target network architecture in this work, and it is discussed in detail in Chapter 4.

The direction of information flow between the layers can also distinguish the network type. If there are no feedback connections from the neuron output back to the input, the network is called a feedforward network (FNN). MLP and CNN are examples of FNN that do not involve any feedback connections. If some layers are connected backwards to earlier layers, the network can hold its previous states in memory. Networks including these feedback connections are called recurrent neural networks (RNNs) and are out of the scope of this work.

3.3 Training

The popularity and usefulness of NNs rely strongly on their ability to learn from the data and store the information in the internal parameters of the network, i.e. weights and biases. The procedure of adjusting these parameters to improve the network performance is called training. Several different learning algorithms can be applied in training. These algorithms are traditionally divided into three broad categories [28]:

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning

In supervised learning, the algorithm applies labelled training data. It contains input-output pairs $\{\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\}$, where $\boldsymbol{y}^{(i)}$ is the desired output with the given input $\boldsymbol{x}^{(i)}$, and index $i \in \mathbb{Z}$ denotes the sample index. The learning algorithm tries to adjust the network parameter such that the predicted output by the network is as close as possible to the desired output $\boldsymbol{y}^{(i)}$. In this Section, it is assumed that $\boldsymbol{x}^{(i)}$, $\boldsymbol{y}^{(i)}$ and any intermediate input and output of the network can be tensors of arbitrary shape. To simplify the notation, each tensor can be imagined as flattened into a vector with an equal number of elements as the original tensor.

With unsupervised learning, only input samples $\boldsymbol{x}^{(i)}$ are provided to the learning algorithm. There is no desired output, and the network is trained to recognize patterns and structures from the data. Unsupervised learning is typically utilized for three main applications; clustering, association, and dimensionality reduction. Reinforcement learning is a learning procedure where the input samples $\boldsymbol{x}^{(i)}$ are fed to the network, and the network interacts with the environment and learns to maximize the reward function or other user-provided reinforcement signal. It can be treated as a game-like situation, where the network gets either rewards or penalties for the actions it performs.

Most neural network training algorithms are based on different variants of gradient descend optimization algorithms. The optimization is executed using model-specific loss functions, which are minimized by adjusting the network parameters. The computing of the gradients in multilayer networks is done using backpropagation [2]. The basic ideas behind these methods are introduced in this Section.

3.3.1 Forward propagation

Forward propagation is the procedure where an input is mapped to a an output using individual weight and bias parameters for each neuron in a neural network. When forward propagation is conserned, the entire NN of k layers can be viewed as an function f that consist simpler function describing individual layers $(f_1, f_2, \ldots, f_k)$ and activations $(\varphi_1, \varphi_2, \ldots, \varphi_k)$. Each layer has internal set of weights $\boldsymbol{W_i}$ and bias parameters $\boldsymbol{b}_i$ that are usually denoted with $\boldsymbol{\theta_i} = \{\boldsymbol{W_i}, \boldsymbol{b}_i\}$ that includes all free parameters of the ith layer. Forward propagation of a network with k layers mapping input $\boldsymbol{x}$ to an output can be written

$$\begin{aligned}
\boldsymbol{a_1} &= f_1(\boldsymbol{x}; \boldsymbol{\theta_1}) \\
\boldsymbol{h_1} &= \varphi_1(\boldsymbol{a_1}) \\
\boldsymbol{a_2} &= f_2(\boldsymbol{h_1}; \boldsymbol{\theta_2}) \\
\boldsymbol{h_2} &= \varphi_2(\boldsymbol{a_2}) \\
&\vdots \\
\boldsymbol{a_k} &= f_k(\boldsymbol{h_{k-1}}; \boldsymbol{\theta_k}) \\
\widehat{\boldsymbol{y}} &= \varphi_k(\boldsymbol{a_k}),
\end{aligned} \tag{3.9}$$

where $\widehat{\boldsymbol{y}}$ is the final output, and $\boldsymbol{a_i}$ and $\boldsymbol{h_i}$ are outputs of the ith layer before and after applying the elementwise activation function φ_i, respectively. As mensioned, the dimensions of $\boldsymbol{a_i}$, $\boldsymbol{h_i}$ and $\widehat{\boldsymbol{y}}$ can vary from single scalar value to any arbitrary shaped tensor depending on the NN and the task it is designed to perform. When forward propagation is performed using model with trained parameters, it is called inference. Inferences are discussed more detailed in Section 3.4.

3.3.2 Loss function

The loss function is a function that measures how an ML model is performing and gives the distance between the predicted output and the model's expected output. The loss function's return value is typically a single scalar value called loss or error. The model's internal parameters are adjusted to minimize the loss function in NN training. Most training algorithms operate using batches consisting of multiple data samples, and the loss function computes an average loss over the entire batch. A loss function must fulfil two assumptions for it to be used in backpropagation, an essential part of any NN training procedure. Firstly, a loss function must be an average over individual loss functions for n individual data samples $x^{(i)}$, i.e. $\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}^{(i)}$. Secondly, it must be written as a function of the network's output.

There are multiple feasible alternatives for the loss function. Choosing the best loss function depends on the NN architecture and type of the problem [2]. Two commonly applied loss functions are mean squared error (MSE) and mean absolute error (MAE). Mean squared error (MSE) for n input samples is given by

$$\mathcal{L}_{MSE}(\widehat{\boldsymbol{Y}}, \boldsymbol{Y}) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{m}\sum_{j=1}^{m}\left(\widehat{\boldsymbol{y}}_j^{(i)} - \boldsymbol{y}_j^{(i)}\right)^2, \tag{3.10}$$

where $\widehat{\boldsymbol{Y}} = (\widehat{\boldsymbol{y}}^{(1)}, \dots, \widehat{\boldsymbol{y}}^{(n)})$ are predicted outputs of the network and $\boldsymbol{Y} = (\boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(n)})$ are expected outputs of the network corresponding to the input samples $\boldsymbol{X} = (\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(n)})$. Arguments $\widehat{\boldsymbol{y}}^{(i)}$ and $\boldsymbol{y}^{(i)}$ can be any arbitrary shaped tensors of m elements in total, and are treated as a vectors. Normalized mean squared error (NMSE) is essentially a standard MSE function normalized with the average of squared expected outputs

$$\mathcal{L}_{NMSE}(\widehat{\boldsymbol{Y}}, \boldsymbol{Y}) = \frac{\frac{1}{n}\sum_{i=1}^{n}\frac{1}{m}\sum_{j=1}^{m}\left(\widehat{\boldsymbol{y}}_j^{(i)} - \boldsymbol{y}_j^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n}\frac{1}{m}\sum_{j=1}^{m}\left(\boldsymbol{y}_j^{(i)}\right)^2}, \tag{3.11}$$

Mean absolute error (MAE) [2] is common linear loss function option given as

$$\mathcal{L}_{MAE}(\widehat{\boldsymbol{Y}}, \boldsymbol{Y}) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{n}\sum_{j=1}^{n}\left|\widehat{\boldsymbol{y}}_j^{(i)} - \boldsymbol{y}_j^{(i)}\right|. \tag{3.12}$$

3.3.3 Backpropagation and optimization

Backward propagation of errors is an algorithm extensively applied for computing the gradient of a loss function with respect to the network parameters. It was first introduced as a general method for automatic differentiation by S. Linnainmaa in 1970 without NN context [29]. In NN training context, the algorithm was introduced in the 1980s using the term *backpropagation (BP)* [30, 31].

Function gradient is typically straightforward to derive analytically, but numerical evaluation can be computationally expensive in many cases. Backpropagation provides a simple and inexpensive procedure to compute gradients for NN learning purposes. The algorithm is based on the chain rule of calculus. The gradient with respect to each parameter is computed layer by layer, iterating the network backwards starting from the last layer to avoid duplicate calculations of unnecessary intermediate values. Using the chain rule gives the gradient of a loss function w.r.t. to the parameters of the ith layer $\boldsymbol{\theta_i}$

$$\nabla_{\boldsymbol{\theta_i}} \mathcal{L}(\boldsymbol{\theta}) = \left(\frac{\partial \boldsymbol{a_i}}{\partial \boldsymbol{\theta_i}}\right)^\top \nabla_{\boldsymbol{a_i}} \mathcal{L}(\boldsymbol{\theta}). \tag{3.13}$$

The first factor on the right hand of (3.13) can be given

$$\frac{\partial \boldsymbol{a_i}}{\partial \boldsymbol{\theta_i}} = \frac{\partial}{\partial \boldsymbol{\theta_i}} f(\boldsymbol{h}_{i-1}, \boldsymbol{\theta_i}) = \boldsymbol{h_{i-1}}, \tag{3.14}$$

if it is assumed that every layer $f_i(\boldsymbol{h}_{i-1}; \boldsymbol{\theta_i})$ is a linear function. The second factor is usually called **error** and denoted

$$\nabla_{\boldsymbol{a_i}} \mathcal{L}(\boldsymbol{\theta}) = \boldsymbol{\delta_i}. \tag{3.15}$$

For the final kth layer can be written using chain rule

$$\boldsymbol{\delta}_k = \nabla_{\boldsymbol{a_k}} \mathcal{L}(\boldsymbol{\theta}) = \left(\frac{\partial \boldsymbol{h_k}}{\partial \boldsymbol{a_k}}\right)^\top \nabla_{\boldsymbol{h_k}} \mathcal{L}(\boldsymbol{\theta}) = \left(\frac{\partial \varphi_k(\boldsymbol{a_k})}{\partial \boldsymbol{a_k}}\right)^\top \nabla_{\widehat{\boldsymbol{Y}}} \mathcal{L}(\boldsymbol{\theta}) \tag{3.16}$$

With the hidden layers, i.e. $1 < i < k$, the recursive definition can be obtained using again the chain rule

$$\begin{aligned} \boldsymbol{\delta}_i = \nabla_{\boldsymbol{a_i}} \mathcal{L}(\boldsymbol{\theta}) &= \left(\frac{\partial \boldsymbol{a_{i+1}}}{\partial \boldsymbol{a_i}}\right)^\top \underbrace{\nabla_{\boldsymbol{a_{i+1}}} \mathcal{L}(\boldsymbol{\theta})}_{\boldsymbol{\delta}_{i+1}} = \left(\frac{\partial \boldsymbol{a_{i+1}}}{\partial \boldsymbol{h_i}} \frac{\partial \boldsymbol{h_i}}{\partial \boldsymbol{a_i}}\right)^\top \boldsymbol{\delta}_{i+1} \\ &= \left(\frac{\partial \boldsymbol{a_{i+1}}}{\partial \boldsymbol{h_i}} \frac{\partial \varphi_i(\boldsymbol{a_i})}{\partial \boldsymbol{a_i}}\right)^\top \boldsymbol{\delta}_{i+1} = \left(\frac{\partial f_{i+1}(\boldsymbol{h_i}; \boldsymbol{\theta_{i+1}})}{\partial \boldsymbol{h_i}} \frac{\partial \varphi_i(\boldsymbol{a_i})}{\partial \boldsymbol{a_i}}\right)^\top \boldsymbol{\delta}_{i+1} \end{aligned} \tag{3.17}$$

Using Equations (3.14), (3.16) and (3.17) the gradient for each layer w.r.t. to the layer parameters $\boldsymbol{\theta_i}$ in Equation (3.13) can be given as

$$\nabla_{\boldsymbol{\theta_i}} \mathcal{L}(\boldsymbol{\theta}) = \boldsymbol{h}_{i-1} \boldsymbol{\delta}_i, \text{ where} \tag{3.18}$$

$$\boldsymbol{\delta}_i = \begin{cases} \left(\frac{\partial \varphi_i(\boldsymbol{a_i})}{\partial \boldsymbol{a_i}}\right)^\top \nabla_{\widehat{\boldsymbol{Y}}} \mathcal{L}(\boldsymbol{\theta}), & \text{if } i = k \\ \left(\frac{\partial f_{i+1}(\boldsymbol{h_i}; \boldsymbol{\theta_{i+1}})}{\partial \boldsymbol{h_i}} \frac{\partial \varphi_i(\boldsymbol{a_i})}{\partial \boldsymbol{a_i}}\right)^\top \boldsymbol{\delta}_{i+1}, & \text{if } 1 \le i < k. \end{cases} \tag{3.19}$$

The computed gradients obtained with backpropagation can be applied in training the network parameters with any gradient-based optimization method.

With a properly determined loss function, the training procedure can be interpreted as a generic optimization problem, where the objective function is

$$J(\boldsymbol{\theta}) = \mathcal{L}(\widehat{\boldsymbol{Y}}, \boldsymbol{Y}) = \mathcal{L}(f(\boldsymbol{X}; \boldsymbol{\theta}), \boldsymbol{Y}). \tag{3.20}$$

Training inputs are used to adjust network parameters $\boldsymbol{\theta}$ by minimizing the objective function $J(\boldsymbol{\theta})$. In general, most training algorithms proceed in the following steps

1. **Initializing phase**. Initialize network parameters using some random initializing strategy.

2. **Forward propagation phase**. Perform forward propagation for each input-output pair $(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})$ in training batch $(\boldsymbol{X}, \boldsymbol{Y})$ proceeding from the input layer to layer k, output layer. Store the predicted outputs $\widehat{\boldsymbol{y}}^{(n)}$ of the network, activations of each layer $\boldsymbol{a_i}$ and output od each layer $\boldsymbol{a_i}$.

3. **Backpropagation phase**. Calculate the gradient of the objective function $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ layer by layer using Equation (3.18). Proceed backwards from the output layer kth to the input layer.

4. **Weight update phase**. Update the network weights using the gradient such that the objective function value decreases, i.e.

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{3.21}$$

5. **Repeat**. Repeat phases 2-4 until desired threshold is reached or the objective function value does not decrease with further iterations.

The parameter η in Equation (3.21) is some positive number called learning rate. The value of learning rate can vary during the training procedure depending on the applied optimization algorithm. Most popular algorithms are stochastic gradient descend (SGD) and its variants such as Adam optimizer [32]. More detailed description of different optimization methods can be found for example in [2].

3.3.4 Regularization

Regularization is a technique that is commonly applied to prevent model overfitting. Typically it is performed by adding the objective function an additional regularization term $\lambda R(\boldsymbol{\theta})$ resulting an objective function

$$\widetilde{J}(\boldsymbol{\theta}) = \mathcal{L}(f(\boldsymbol{X}; \boldsymbol{\theta}), \boldsymbol{Y}) + \lambda R(\boldsymbol{\omega}). \tag{3.22}$$

The regularizer R is a function of the network parameters $\boldsymbol{\theta}$, that penalizes specific weight values more than others. Bias parameters rarely cause overfitting, and thus regularization is typically limited to weight parameters $\boldsymbol{\omega}$. Multiplier λ is a hyperparameter determined separately from the training.

Regularizer, R, is typically chosen to prefer small weight values. L^1-regularization applies the absolute values of weights

$$R_{L^1}(\boldsymbol{\omega}) = \sum_i |\omega_i|. \tag{3.23}$$

It forces most weight values to be zero, increasing the network's sparsity. Thus, it is an effective method to decrease the network complexity and the possibility of overfitting.

L^2-regularisation uses the squared L^2-norm of the weights as a regularization element

$$R_{L^2}(\boldsymbol{\omega}) = \sum_i \omega_i^2. \tag{3.24}$$

With L^2-regularization, weights tend to have minimal but still non-zero values. It does not encourage zero weights, and it is not a robust regularization method if the data includes outlier samples.

In addition to L^1- and L^2-regularisation, the dropout regularization has become very popular recently [33]. *Dropout* is a regularization method that approximates training a large number of NNs with different architectures in parallel. The method prevents effectively overfitting by increasing the randomness of the network. It is a computationally inexpensive but robust procedure for regularizing different NNs. The idea behind dropout is rather simple: during training, a neuron is turned off with some random probability of $\boldsymbol{P} = [0, 1[$.

3.4 Inference

The inference is a process to deploy and use the neural network model with the trained parameters. Mathematically it is identical to the forward propagation in Equation (3.9). Modern NN architectures are designed to achieve the best possible accuracy in the given task. This principle has led to a situation where many current state-of-the-art NNs, especially CNNs, are very complex and computationally demanding [34]. Complexity is not a problem during training, although it requires a significant computation time. Dedicated hardware such as GPUs that process tensors in parallel are widely used in NNs during both training and inference [2]. In practice, NN model development and training are usually done using existing deep learning frameworks, which are shortly introduced later in this Section.

Performing inference typically unveils some restrictions and requirements for the model itself. When practical applications are considered, they can be deployed to limited computational capacity and memory devices, such as mobile devices, drones and many embedded devices. Moreover, in real-time software applications, the inference latency has a crucial role in the feasibility of the NN model. Therefore, active research has been conducted to optimize model sizes and inference times with minimal accuracy losses.

In general, there are roughly two different approaches for optimizing NN inferences. The first approach develops novel NN architectures to exploit inefficient computations and reduce memory requirements without a notable decrease in model accuracy. For example, MobileNet [35], and SqueezeNet [36] are compact CNNs developed for computer vision applications in portable devices with limited computational resources. The second approach tries to optimize existing models. Inferences can be converted to a more optimized format for the target device using, for example, specific deep learning compilers. In this Section, details of the most common deep learning compilers are discussed. Networks can also be optimized by converting data into formats that require less memory and are more efficient to process. Usually, this is done using quantization, supported by many deep learning frameworks and compilers. Because this method is also extensively used in this work, it is discussed in Section 3.4.3.

3.4.1 Deep learning frameworks

Standard workflow when developing NN models relies heavily on using some existing DL framework. They are platforms that provide support and tools for all steps needed to create a trained NN inference solution to be performed on a target device. Most popular DL frameworks are free, open-source software libraries offering a simple high-level approach to DL models construction. They support broadly different types of NNs and provide tools for practical model training and performing inferences. Typically, frameworks can utilize different hardware accelerators, such as GPUs, to optimize the training and inferences of large models. The two most popular DL frameworks include:

TensorFlow is DL framework developed by Google Brain released in 2015 [37]. TensorFlow has the most extensive support for different language interfaces, including Python, C++ and many others. TensorFlow can represent differentiable programs employing a dataflow graph of primitive operators extended with restricted control edges. TensorFlow Lite is an expansion toolset designed for mobile and embedded DL solutions [37], that offers tools for optimizing NNs into a more memory-efficient and mobile-optimized format. Keras is a popular, simplified frontend to ease the usage of the TensorFlow core [38].

PyTorch is a highly customizable DL framework based on the former Torch framework primarily developed by Facebook [39]. PyTorch defines a Tensor class to store and operate on multidimensional rectangular arrays. It uses primitive embedding to construct dynamic dataflow graphs in Python. Pytorch Mobile is a set of tools for deploying models to mobile and embedded devices. It provides tools for quantizing and optimizing models created in PyTorch that can then be saved into a serialized format for mobile deployment. Pytorch Mobile offers APIs for Android, iOS and Linux devices. Similar to Keras, FastAI is an advanced high-level frontend to simplify the usage of PyTorch [40].

ONNX, Open neural network exchange was introduced in 2017 to increase interoperability between different DL frameworks [41]. It provides an open standard for sharing DL models between different DL frameworks based on definitions for extendable computation graph models, built-in operators and standard data types. Each computation dataflow graph forms an acyclic graph consisting of a list of nodes. Each node is a call to an operator, including input and output values. ONNX is also utilized in this work.

In this work, PyTorch and ONNX are applied as a part of the developed inference framework, which is discussed in Chapter 6.

3.4.2 Deep learning compilers

DL frameworks enable reasonable network inferences in mobile devices and other consumer electronics. However, the scope of possible inference hardware is not limited to only them. Target hardware can include for example, field-programmable gate arrays (FPGA), specialized processors such as DSPs [42, 43] and application-specific integrated circuits (ASIC) [21, 18]. Recently internet giants have pointed increasing interest toward ASICs that are specifically designed to process network inferences, such as TPU developed by Google [44]. Different hardware is designed for certain computations, and hardware-specific optimization is required for efficient inferences. It is known that manual implementation for various hardware platforms is time-consuming.

As a solution, DL compilers, such as TVM [45] and Glow [46], are proposed. These compilers directly transform computational graphs from DL frameworks into an optimized machine code suitable for the target hardware. They use layered design, where the high-level graph is transformed into final machine code via one or more intermediate representations (IR), optimized separately at each step. When this approach is used, the target devices have to support only a limited number of generic operations. A limited number of needed operations enables inferences to be easily scaled for different types of hardware.

A typical starting point for DL compilers is a computational graph of a NN model offered by a DL framework such as TensorFlow or PyTorch or an exchange format such as ONNX [41, 45]. In the first stage, hardware-independent high-level dataflow rewriting is performed to form an optimized graph. In the second stage, the resulting high-level IR is further optimized at the operator level, e.g., removing unnecessary nodes and replacing operations with cheaper equivalents. This stage operates at a higher, deep learning specific abstraction level, which cannot be done efficiently with general-purpose compilers [45]. An optimization of this stage is operator fusion, which combines multiple operations into a single processing step, avoiding redundant memory accesses. After this, the high-level IR is transformed into a lower-level IR. The low-level IR is designed for hardware-specific optimization and code generation on multiple hardware targets [21]. As a result of this compiler chain, generated inference code optimized for the target hardware is produced.

DL compilers have been proved to provide powerful tools for accelerating NN inference on different hardware. However, their focus is on supporting general-purpose hardware (e.g. CPU, GPU) and designed explicitly for DL purposes, i.e. DL accelerators. This work focuses on implementing inference for DSP mainly designed for wireless communication, which is not supported by any available DL compiler. Thus, the required inference operations are implemented manually using the available general-purpose compiler. Because only a tiny subset of DL models are considered, and a similar approach has been successfully applied earlier [9, 10], this approach is justified.

3.4.3 Quantization

Quantization is a specific issue addressed by DL frameworks and compilers, related especially to memory usage but also computational efficiency [47]. In this context, quantization refers to converting the network parameters into a format requiring a smaller number of bits. Reduced bit-widths can be obtained e.g. by converting common 32-bit floating-point parameters into fewer bit floating-point representations [37], or integers [19, 34]. Integer quantization is particularly useful with embedded systems because network inferences can be performed using integer-only operations. In many embedded systems, there is no hardware support for floating-point operations, and integer quantization is necessary to perform any inferences in the first place. Though the system has floating-point support, integer operations are usually significantly faster than floating-point operations and require cheaper, less complex hardware [34]. Integer quantization can also be favoured in some cases due to memory-related reasons. Typical modern processors are much faster than the fastest memory available. With small bit-width integers, the limited memory bandwidth can be utilized more efficiently, providing significantly improved data-level parallelism [19].

There are several different approaches for performing integer quantization of NNs. The most common approach is to quantize parameters to 8-bit signed or unsigned integers, supported by all popular DL frameworks and compilers. 8-bit linear quantization can be done symmetrically such that the real-valued range endpoints map to the smallest and largest integers. The second option is to do it symmetrically so that the real-valued zero maps to the quantized integer zero [34]. Separate scale parameters are used to track the range of real values represented by the quantized values. The parameters must be chosen carefully to avoid losing accuracy when reducing the number of bits. In general, there are two approaches in which part of the development quantization is performed. In post-training quantization, network parameters are quantized after the model has been trained [47, 20]. In quantization aware training, the quantized inference is taken into account already during the training process [34, 19].

8-bit linear integer quantization is probably the most popular integer quantization scheme, but other useful schemes have been reported. Floating-point numbers can be converted to fixed-point numbers by applying appropriate bit-widths [47]. This approach is prevalent in many signal processing applications; thus, most signal processing hardware is usually designed for fixed-point computation [48]. The implementation of this work is designed for wireless physical layer purposes, and the target hardware is optimized for operating 16-bit fixed-point numbers. Related theory about fixed-point numbers and arithmetic will be discussed more detailed in Section 5.1.3, and the implemented quantization approach in Section 6.2.

Chapter 4

Convolutional neural networks

A convolutional neural network (CNN) is an essential subtype of feedforward neural networks. They employ a mathematical operation called convolution in at least one of their layers. In most applications, CNNs are designed to process multidimensional input data with known grid-like topology. They are instrumental in extracting information from input data where the relative position of elements is more important than the absolute position. Examples of such data include 1-D time series with regular intervals or 2-D images consisting of a grid of pixels.

CNNs have been studied extensively during the last decades and have shown excellent performance in many applications. LeNet-5, one of the earliest modern CNNs, was introduced in the 1990s by LeCun et al. [49]. It was initially developed to recognize handwritten digits and was later considered the basis of all modern CNNs. LeNet-5 introduced all the basic building blocks of CNNs, and during that time, it outperformed all other recognition methods. After that, different variations of LeNet-5 architecture have been widely used for many applications also in other fields outside object recognition.

The second breakthrough of CNNs came in the 2000s due to the development of fast NN implementations utilizing graphics processing units (GPUs). Krizhevsky et al. won the ImageNet ILSVRC challenge in 2012 with their GPU-based AlexNet architecture [26]. The network had a very similar architecture to LeNet-5, but it consisted of more layers with more learnable parameters. Since then, CNNs have won several machine learning and computer vision contests and raised emerging commercial interest [50]. So far, most CNN solutions are applied to 2-D data such as images using a 2-D convolution operation. However, 2-D convolution is not suitable for all applications, where the data cannot be converted to 2-D conveniently [51]. Thus, CNNs applying 1-D convolution have raised emerging interest during recent years, especially in applications where the data can be expressed easily in 1-D. Recently 1-D CNNs have been proposed and already achieved state-of-the-art performance levels in several applications such as electroencephalography (EEG) based early diagnosis [52], structural health monitoring [53], and damage detection in bearings [54, 55]. Because 1-D convolution is also computationally much less expensive, 1-D CNNs are particularly interesting when considering the real-time inference applications in embedded systems with limited computation capacity [51].

This chapter provides a brief theoretical background on CNNs. Because hardware implementations in this work cover only 1-D CNNs, the focus is mainly on 1-D convolution. The fundamental mathematical operations related to CNNs, convolution and cross-correlation are introduced in the first Section. It is followed by introducing some key advantages of CNNs, the general architectural principles in designing CNNs, and some of their basic building blocks.

4.1 Convolution

The convolution $(f * g) : \mathbb{R} \to \mathbb{R}$ of two functions $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$ produces a third function by

$$(f * g)(t) = \int_{\mathbb{R}} f(\tau)g(t-\tau)d\tau. \tag{4.1}$$

It is well-defined if $f(t)$ and $g(t)$ decay sufficiently rapidly at infinity such that the integral in equation (4.1) exists [2]. In the field of machine learning, as in this work, the data is not continuous and the domain of definition is always somehow discretized. If it is assumed that function arguments can take only integer values i.e. $f : \mathbb{Z} \to \mathbb{R}$ and $g : \mathbb{Z} \to \mathbb{R}$, the **discrete convolution** is defined as

$$(f * g)[i] = \sum_{n=-\infty}^{\infty} f[n]g[i-n]. \tag{4.2}$$

Especially in the field of machine learning, the first argument of the operation, in this case, $f[i]$, is referred to often as **input** [2]. The second argument, in this case, $g[i]$, is often referred to as **kernel** and the output $(f * g)[i]$ is called usually as a **feature map**. These three terms are used in this work when details related to convolution are discussed. An example of one-dimensional discrete convolution is visualized in Figure 4.1. The elements of f that do not contain actual values of f are visualized with zeros.

Commutativity is an important property of convolution that is convenient in many applications. It means that there holds

$$(f * g)[i] = (g * f)[i] = \sum_{n=-\infty}^{\infty} f[i-n]g[n]. \tag{4.3}$$

The related discrete operation without the kernel reversion is called **cross-correlation** and defined by

$$(f \star g)[i] = \sum_{n} f[n]g[n+i]. \tag{4.4}$$

Cross-correlation is not commutative. In the NN context, commutativity property is rarely necessary [2]. The learning algorithm will learn the correct values of the kernel in the appropriate location. An algorithm based on convolution applying kernel reversing will learn a reversed kernel, and an algorithm without the reversion will learn identical values, but just in different locations. Therefore, many widely used deep learning frameworks implement, in fact, cross-correlation but call it convolution [2]. In this work, this convention is followed, and both operations are

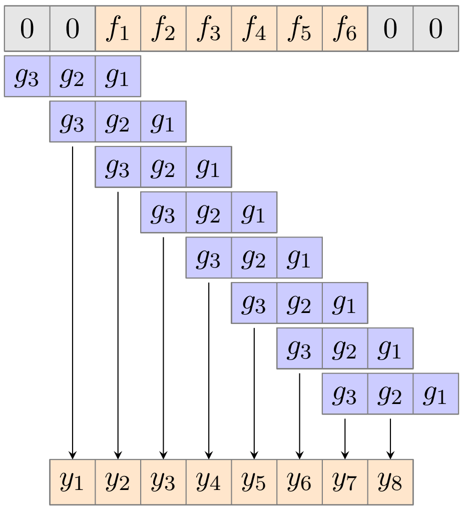


Figure 4.1: Discrete convolution with one dimensional input f and kernel g of width 3. The convolution $f * g$ is given by y.

called convolution. When discussing convolution in the context of deep learning and neural networks, it is not referred precisely to as the standard discrete convolution as it is usually understood in the mathematical literature and described in this Section. The differences and the unique properties of convolution in practical CNN applications are discussed in Section 4.3.1.

The convolution theorem is one of the most important properties applied in signal processing that connects the Fourier transform and convolution. For two discrete functions f and g, the theorem is written

$$(f * g)[i] = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g)), \tag{4.5}$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ denote discrete Fourier transform (DFT) and inverse discrete Fourier transform (IDFT) respectively, and $(\cdot)$ denotes the pointwise product between two functions [56]. Thus, discrete convolution can be replaced with DFT and pointwise product of the transformed sequences under suitable conditions. Many fast convolution algorithms apply this property to obtain more efficient computations than direct convolution. These approaches apply fast Fourier transform algorithms (FFT) to evaluate the DFT of the sequences, which can reduce the required amount of arithmetic operations significantly and speed up the computation [56].

4.2 Key concepts

Local connectivity, **parameter sharing** and **equivariance** are important concepts CNNs exploit to improve overall performance, compared to the traditional multilayer perceptrons (MLPs).

Local connectivity means that each neuron of a single layer is connected only to a small region of the input volume. In convolution layers, these sparse connections between local regions are achieved using smaller kernels than the input reducing the

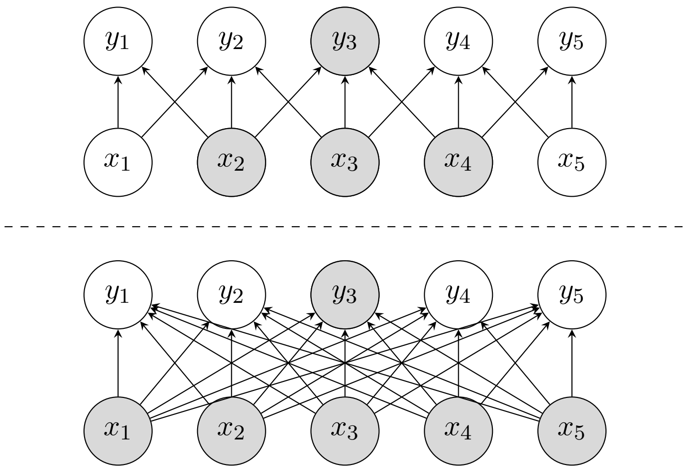


Figure 4.2: Local connectivity. The highlighted input elements in $\boldsymbol{x}$ are the receptive field of highlighted output element y_3. **Top**: When convolution with kernel of width 3 is applied, only three elements of $\boldsymbol{x}$ affect y_3. **Bottom**: With fully connected neurons, all inputs elements affect to y_3.

number of parameters. On the contrary, the number of parameters in fully connected layers might be vast, especially when multidimensional data is considered. Local connectivity is visualized in Figure 4.2. A receptive field hyperparameter describes how large input volume is connected to each output neuron.

Parameter sharing means applying the same parameter value in multiple locations inside a single model. On the contrary, in NNs with fully connected layers, every edge between neurons has a unique weight. Parameter sharing is visualized in

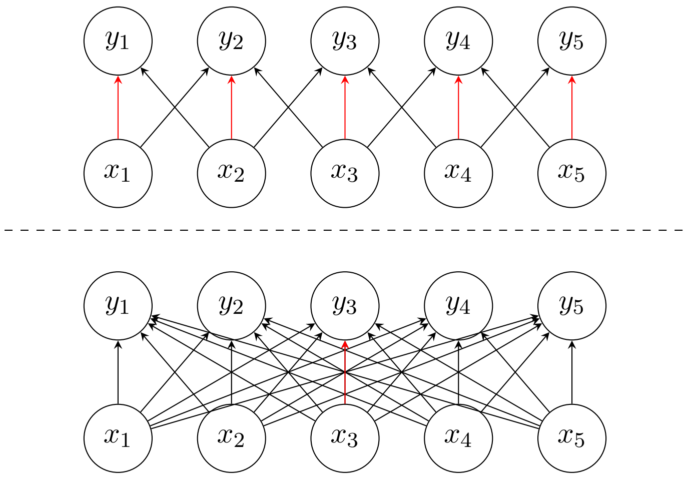


Figure 4.3: Parameter sharing. The connections applying a distinct parameter are indicated with red arrows. **Top**: The central element of a kernel is indicated with a red arrow in a simple convolution model. Since parameter sharing, the same parameter is used at all input locations. **Bottom**: Single red arrow shows the central element of a weight matrix in a fully-connected model. The model does not apply parameter sharing, and the parameter is used only once.

Figure 4.3. One essential advantage of applying convolution layers and parameter sharing is to reduce the memory requirements. Convolution can be dramatically more efficient in memory consumption and statistical efficiency.

Equivariance with respect to translation is a property of a convolution layer, that can be achieved by choosing the parameter sharing with a particular form. A function $f(x)$ is equivariant to a function $g(x)$ if

$$f(g(x)) = g(f(x)) \tag{4.6}$$

In practice, it means that if the layer's input is translated, its output translates in the same way. The resulting output is equivariant under translations of input features' because the same parameters are shared across the whole input volume.

4.3 Architecture

Most common layers in CNNs are **Convolution layer**, **Pooling layer** and **Fully-Connected layer**. An example of a classifying CNN architecture is illustrated in Figure 4.4. The example network and the network architectures applied in this work take multiple 1-D arrays as input. One array is called a channel, and input can consist of several channels. CNNs process input data through one or more convolution layers, the basic building block of all CNNs. A convolution layer produces a multichannel feature map as an output. The properties of feature maps can be adjusted with different hyperparameters, which are covered in more detail in Section 4.3.1. Linear

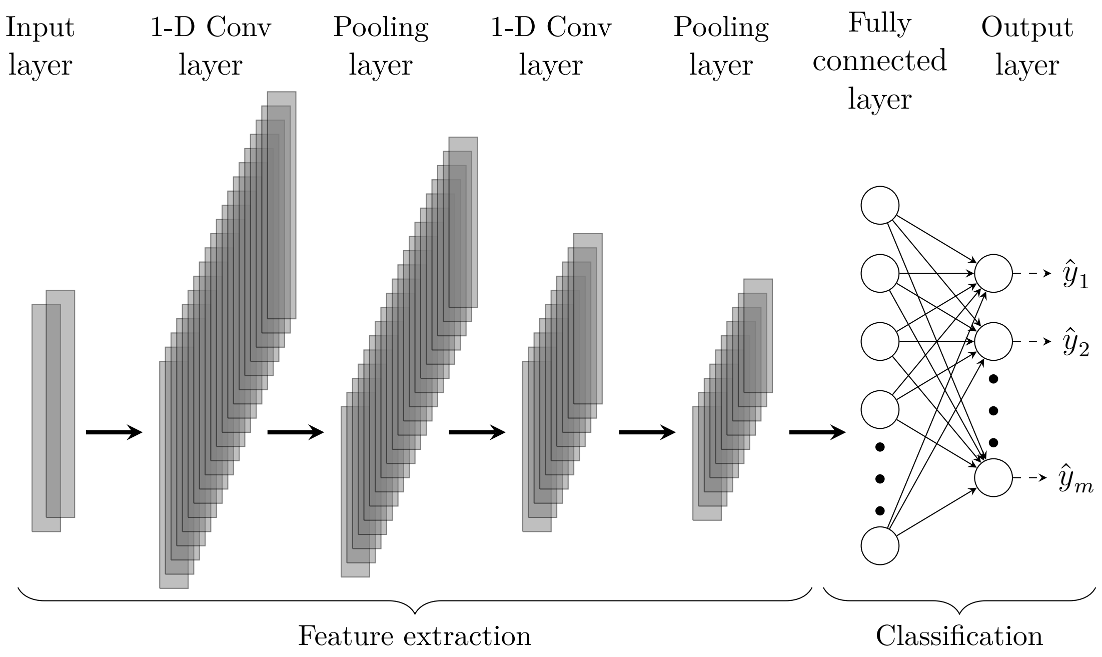


Figure 4.4: The high-level structure of an example convolutional neural network with two 1-D convolution layers, each followed by a pooling layer. The last layer of the network is a classifying fully-connected layer.

feature maps are then combined with non-linear activation functions to produce non-linear feature maps as an output of a layer. Different activation function choices are covered in Section 3.

Another important concept related to CNNs is pooling. Pooling layers are placed after convolution layers and perform non-linear down-sampling of the feature map. Most common pooling layer implementations are discussed in Section 4.3.2. When the CNN is used for classification, convolution and pooling layers can be followed by one or more fully-connected layers, as in the example in Figure 4.4. The multidimensional output of a convolution layer must be flattened into a single vector before applying fully-connected layers.

4.3.1 Convolution layer

CNN has at least one or more convolution layers that perform an operation consisting of multiple parallel applications of convolution or cross-correlation. The layer's purpose is to extract features from the input located in different locations across the volume. Because convolution operation with a single kernel can extract only one kind of feature albeit, in multiple locations, several parallel convolution operations are needed.

When working with one-dimensional data, such as time series, the actual software implementations of convolution layers in fact have 3-D tensors as an input. Let the layer input consist of observations $\boldsymbol{x} \in \mathbb{R}^{N \times C^{(in)} \times L^{(in)}}$, where N is the number of data samples, $C^{(in)}$ is the number of input channels, and $L^{(in)}$ is the spatial extend of each channel. The learnable parameters of the convolution layer consist of a set of kernels and a set of bias terms. Let $\boldsymbol{\omega} \in \mathbb{R}^{C^{(out)} \times C^{(in)} \times F}$ be the tensor of kernels, where $C^{(out)}$ is the number of output channels, $C^{(in)}$ is the number of input channels, and F is the receptive field of single kernel. The layer activation $\boldsymbol{a} \in \mathbb{R}^{N \times C^{(out)} \times L^{(out)}}$ is given

$$a_{i,j,k} = \sum_{l=1}^{C^{(in)}} \sum_{m=1}^{F} x_{i,l,k+m-1} \omega_{j,l,m} \tag{4.7}$$

, where the indexes correspond as follows: i is the data sample index, j is the output channel index, k is the output element index, l is the input channel index, and m is the offset between output and input elements.

The summation is illustrated in Figure 4.5. As can be seen, kernels have a small receptive field but they extend along all input channels enabling strong connections between spatially local input patterns. In the Figure, different colours illustrate different kernels that are applied to the input feature maps. Finally, bias term $\boldsymbol{b} \in \mathbb{R}^{C^{(out)}}$ is added to each output before applying non-linear activation function $\phi(\cdot)$ to obtain final output of the layer $\boldsymbol{h} \in \mathbb{R}^{N \times C^{(out)} \times L^{(out)}}$, where

$$h_{i,j,k} = \phi(a_{i,j,k} + b_j) \tag{4.8}$$

A typical choice is to have one bias term per one output channel and share the parameter over all locations within this channel.

In convolution layers, there are a few fundamental hyperparameters that are used to control the spatial arrangement and dimensions of the output feature maps:

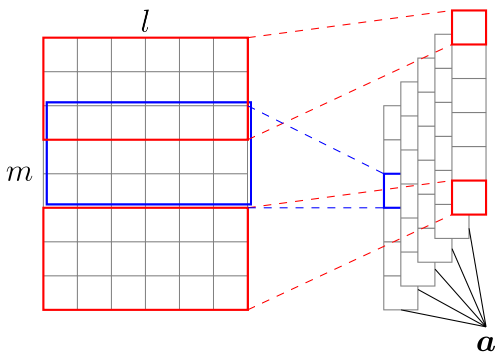


Figure 4.5: An illustration of 1-D convolution layer.

- **Depth**, $C^{(out)}$: Number of output channels, i.e. the number of kernels applied to the layer input. Adjusts the number of neurons connected to the same region in the input volume. Neurons learn to activate certain features from the input data, so this hyperparameter controls how many different features one convolution layer can activate.

- **Receptive field**, F: Neurons are connected only to a local region of the input volume as illustrated in Figure 4.5. The receptive field gives the spatial extent of this connectivity, i.e. the size of a kernel.

- **Stride**, S: Specifies the stride the kernel is slid along the input feature map. When stride is 1, the kernel is moved one element by one, leading to overlapping receptive fields and larger output feature maps. When stride is more than 1, there is less overlap between receptive fields, and output feature maps shrink.

- **Zero-padding**, P: The input can be zero-padded to control the size of the output feature map. Without padding, the width of the output shrinks by one element less than the receptive field size at each convolution layer. Zero-padding allows controlling the receptive field and the size of the output independently.

The effects of different hyperparameter on the spatial arrangement of the output are illustrated in Figure 4.6. The Figure illustrates the behavior of one neuron with different hyperparameter values. The kernel and bias values are omitted for simplicity reasons. Assuming a convolution layer accepting input with dimension $\boldsymbol{x} \in \mathbb{R}^{N \times C^{(in)} \times L^{(in)}}$. The shape of corresponding output $\boldsymbol{h} \in \mathbb{R}^{N \times C^{(out)} \times L^{(out)}}$ can be determined using the following rule

$$L^{(out)} = \frac{L^{(in)} - F + 2P}{S} + 1 \tag{4.9}$$

, where $L^{(out)}$ is the spatial extend of the input, that depends on receptive field F, zero-padding P and stride S.

The convolution layer has the most considerable computational load within a CNN, and it has been studied how efficiency can be improved. Most studies consider 2D CNNs, but the same principles can be applied to 1D CNNs when some

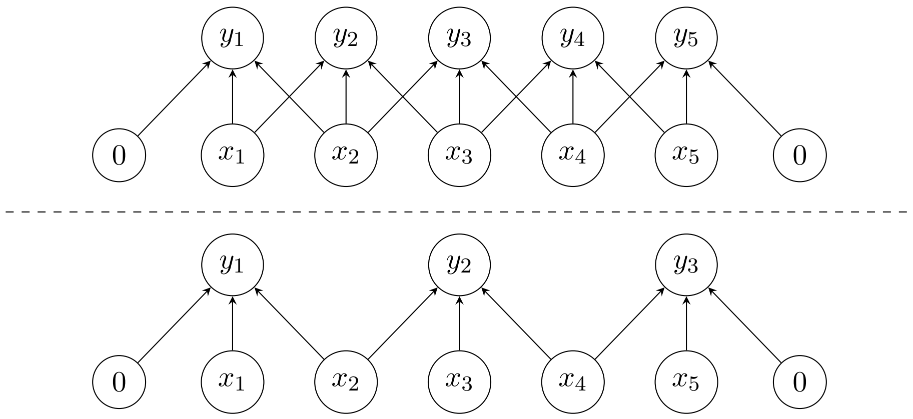


Figure 4.6: Hyperparameters. One 1-D neuron with receptive field $F = 3$, input size $L^{(in)} = 5$ and zero-padding $P = 1$. **Top**: The neuron strided across the input in stride of $S = 1$. The output is of size $L^{(out)} = (5 - 3 + 2)/1 + 1 = 5$. **Bottom**: The neuron uses larger stride of $S = 2$, giving output of size $L^{(out)} = (5 - 3 + 2)/2 + 1 = 3$.

simplifications are made. Roughly three approaches can be exploited to increase the efficiency of convolution layer processing. All approaches typically require hardware-aware implementations, but modern DL frameworks and compilers provide some support for efficient convolution layer computation with specialized hardware [57].

- Lower the convolution layer to a standard matrix-matrix multiplication and apply General Matrix Multiplication (GEMM). Kernels and input are transformed into two intermediate matrices so that the dot-product of these matrices matches the output obtained with direct convolution. In this method, existing robust GEMM implementations can be applied [57]. A major downside is that the intermediate matrices significantly increase memory consumption.

- Transform kernels and input to a domain, where elementwise multiplication can be applied to two transformed sequences. Many implementations are based on Fast Fourier transform (FFT) and convolution theorem (Eq. (4.5)) [58]. The transformation cost may be too high with a single convolution, but the cost can be amortized if the transformations can be reused effectively. FFT approach can bring major benefits when many kernels and long convolved vectors are used.

- Develop specialized hardware implementations based on the direct computation of convolution requiring detailed knowledge about the target hardware. Typically, the optimization leads to multiple specialized solutions optimized only for some subset of possible parameter choices for the convolution layer. This approach can enable efficient implementations for some models but does not allow the development of general implementation. This approach is applied in this work.

4.3.2 Pooling layer

The pooling layer is a very typical type of layer in CNNs that is placed after convolution layers. It performs non-linear downsampling of the input, which helps to make the model more invariant to small input translations. The pooling layer can be applied to sequentially reduce the feature maps' spatial size, decrease the number of parameters, and limit memory consumption and computation in the network. Thus, it is a suitable tool to control overfitting.

Few standard non-linear functions are commonly applied in pooling. The most popular choice is **max pooling**, the operation reports the maximum value in the receptive field region. The second standard option is **average pooling** that computes an average of the elements in the receptive field region. 1-D max pooling and average pooling operations are illustrated in Figure 4.7.

The pooling operation does not have any learnable parameter, so it does not introduce any new parameter to the network. **Stride**, S and **receptive field**, F are the hyperparameter that determine the size of the output. Pooling does not change the depth of the network, i.e. the number of channels the data contains. Assuming a pooling layer accepting input $\boldsymbol{x} \in \mathbb{R}^{N \times C^{(in)} \times L^{(in)}}$, where N is the number of batches, $C^{(in)}$ the number of input channels and $L^{(in)}$ is the spatial extend of the input. The shape of the output $\boldsymbol{h} \in \mathbb{R}^{N \times C^{(out)} \times L^{(out)}}$ can be determined using the following rule

$$L^{(out)} = \text{floor}\left(\frac{L^{(in)} - F}{S}\right) + 1 \tag{4.10}$$

, where $L^{(out)}$ is the spatial extend of the output and depending on the receptive field F and the stride S. The output depth $C^{(out)} = C^{(in)}$ because pooling does not affect the depth. In some cases, the rounding function floor$(\cdot)$ is replaced with function ceil$(\cdot)$.

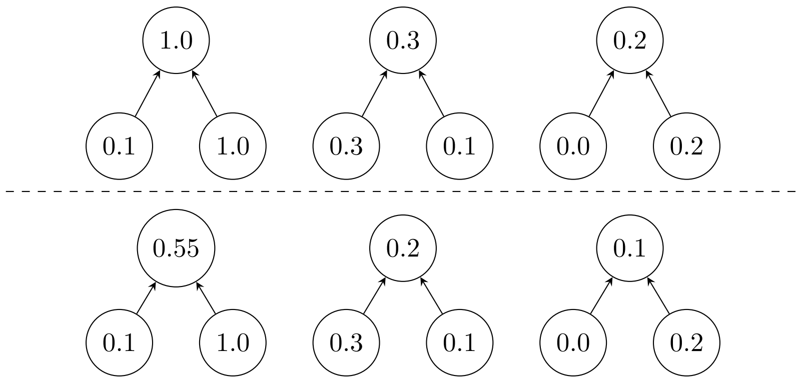


Figure 4.7: Pooling operation. One 1-D neuron performing pooling with receptive field $F = 2$, input size $L^{(in)} = 6$ and stride $S = 2$. The output is of size $(6 - 2)/2 + 1 = 5$. **Top**: Max pooling. **Bottom**: Average pooling.

Chapter 5

Digital signal processor

In this work, the target hardware for CNN inference is a digital signal processor (DSP), a part of an integrated circuit referred to as a System on a Chip (SoC). SoC typically comprises an entire computer system and can include general-purpose processors (GPP), internal memory units, I/O ports, application-specific hardware accelerators, and additional processors such as DSPs or graphics processing units (GPU). The purpose of SoCs is to integrate the functionalities of multiple chips into one, thus reducing the size, power consumption and manufacturing costs of mass-produced computer systems. The downside is that extending the features available in an existing SoC is not possible without designing and manufacturing a new chip. That is why SoCs are generally used in applications where the attainable benefits are valuable, and it is not crucial to replace a single component of the system.

To be more precise, the target hardware of this work is the DSP of a physical layer signal processing SoC designed to operate as a part of a 5G baseband solution. It is specifically designed to perform real-time signal processing and fulfil the strict time requirements introduced by the 5G technology. The SoC contains

- multiple CPU cores,
- a large number of application-specific hardware accelerator blocks for executing time-critical signal processing tasks, and
- several programmable DSP cores for accelerating computationally demanding algorithms.

In this Chapter, the target SoC is covered. First, some key concepts of DSPs are introduced, including parallelism, memory architecture and fixed-point number representation. The second Section of this Chapter gives a brief overview of the CEVA-XC4500 DSP architecture and the most important features affecting the inference implementation.

5.1 Key concepts

The target SoC is designed to perform physical layer signal processing in real-time, and thus it uses multiple methods to accelerate algorithms. The most critical software is handled by the application-specific hardware accelerators, while the other is processed by the general-purpose processors (GPPs) or digital signal processors (DSPs). The target SoC contains multiple identical DSP cores that process independent tasks parallel such that one task is processed entirely by the same core. The DSP job system is designed to not split tasks between the cores. The job system is why the parallelism mechanisms within the DSP cores are crucial when designing software with real-time processing capabilities.

5.1.1 Parallelism

Instruction-level parallelism

A single instruction on a processor can be divided into several stages, each taking at least one cycle to complete. A reduced instruction set computer (RISC) has a small and highly optimized set of instructions with a regular instruction pipeline. The instruction cycle in the classic RISC pipeline consists of five stages: instruction fetch, instruction decode, execute, memory access, and register write back [59]. Each stage is typically handled by a different functional unit of the processor. If each instruction is processed sequentially from the first stage to the last, only one functional unit of a processor is performing a task at a time, and processor resources are wasted.

Instruction set parallelism (ILP) can be applied to speed up computing. There are two typical approaches for ILP: dynamic and static. Dynamic ILP means the processor decides which instructions to execute in parallel at run time. Pipelining is the simplest form of dynamic ILP. If consecutive instructions are independent, the processor can speed up the execution by processing different stages of consecutive instructions simultaneously using different functional units. In the classic RISC case, up to five instructions can be processed simultaneously, and it is possible to achieve the processing speed of one instruction per clock cycle [59].

Many modern processors can perform more efficient dynamic ILP than pipelining. Superscalar processors contain redundant functional units, enabling parallel execution of multiple instructions. Superscalar processors perform runtime dynamic dependency analysis. If an instruction does not depend on the result of another instruction, it can be executed parallelly. It is possible to process more than one instruction per cycle with a superscalar processor combined with pipelining [60].

When dynamic ILP is applied, complex hardware logic is needed for the runtime dependency analysis, which increases hardware costs and limits the size of a possible instruction set. With static ILP, runtime dependency analysis is not necessary. Instead, the instructions scheduling is determined at compile time. Typically with higher-level languages such as C++, this task is handled by the compiler. Very long instruction word (VLIW) is a static ILP method where a specific compiler is used to compose a packet of instructions that can be executed simultaneously. The compiler is responsible for combining the instructions so that the packets contain only independent instructions. Typically VLIW processors are superscalar and support also pipelining. The target DSP in this work is a superscalar VLIW processor utilizing a high level of instruction-level parallelism [48].

Data-level parallelism

Data-level parallelism refers to the computing approach, where the data is distributed to multiple nodes that operate in parallel. One common way to implement data-level parallelism is Single instruction, multiple data (SIMD), where each unit performs the same instruction simultaneously on multiple data elements [59]. A common form of SIMD is vectorization, where the same operation is applied to a wide vector register instead of one element. The target DSP in this work utilizes vectorization as the primary mechanism for data-level parallelism [48]. Figure 5.1 illustrates scalar and SIMD versions of multiple-accumulate (MAC) operations. The operation is performed on single 32-bit floating-point numbers in the scalar version. In the SIMD version, eight 32-bit floating-point numbers are loaded to a 256-bits wide register and operated with single MAC instruction after that.

Especially for digital signal processing, SIMD can provide significant advantages and improvement in performance. Compared to general-purpose computing, SIMD also has many disadvantages that should be considered when designing implementations. All algorithms cannot be vectorized easily because data alignment may have some restrictions in SIMD algorithms. For example, algorithms that require collecting scattered data into SIMD registers cannot employ vectorization very well.

Robust SIMD implementations usually require using some architecture-specific SIMD instruction sets. Specific intrinsic functions and compilers must be used, requiring much manual work by the developer to develop new implementations. Typically different implementations must be provided to be able to support different hardware. So far, most compilers cannot generate SIMD instructions from a generic high-level program without human intervention [59].

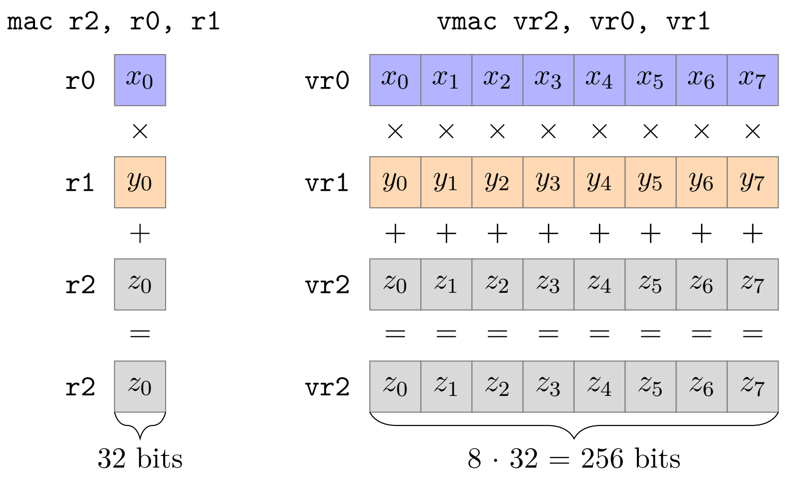

mac r2, r0, r1
r0
x_0
×
r1
y_0
+
r2
z_0
=
r2
z_0
32 bits
vmac vr2, vr0, vr1
vr0
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
vr1
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7
vr2
z_0 z_1 z_2 z_3 z_4 z_5 z_6 z_7
$8 \cdot 32 = 256$ bits

Figure 5.1: Visualization of multiply-accumulate (MAC) operation using 32-bit floating point numbers with scalar and SIMD instruction.

5.1.2 Memory

The DSP memory system is one of the critical factors affecting the overall performance of DSP. Most DSPs contain two essentially different types of memory; external and internal. Internal on-chip memory provides high-speed and low latency memory access to the DSP's processing units [61]. In addition, DSP has access to external and peripheral off-chip memory. For example, in typical SoC, external memory is shared with multiple other hardware subsystems and serves as the main system memory for the system. Internal memory should be preferred when designing DSP algorithms due to the superior latency compared to external memory.

Direct memory access (DMA) is used when data is transferred between internal and external memory. DMA is the hardware mechanism that allows peripheral components to transfer data directly without the need to involve the system processor [62]. It enables other hardware subsystems to access data in the main system memory without tying the processor resources for the duration of the data transfer. Typically, moving data to and from memory is much slower than the processing capability of the processor. Compared to the alternative methods where the processor is responsible for the data transfer, DMA can reduce the computational overhead significantly. DMA enables that computation and memory access can be done parallel as the processor continues normal processing, while DMA is responsible for handling the memory access simultaneously.

The internal memory architecture of the processors is one of the key differences between GPPs and DSPs. Most GPPs in modern computers are based on von Neumann architecture, which is visualized in Figure 5.2. It uses single, shared internal memory for program instructions and data combined with a single bus for memory access. The central processing unit (CPU) fetches an instruction from memory in a normal program execution procedure. A specific control unit decodes the instruction to determine what operations are executed, and then the instruction is executed in the arithmetic or logic unit. The instruction has typically two parts, the *opcode* and the *operand*. The *opcode* specifies the executed operation and the *operand* specifies what data should be operated.

In the classical von Neumann architecture, the processor operates serially, retrieving one instruction or piece of data at a time. When the instruction execution

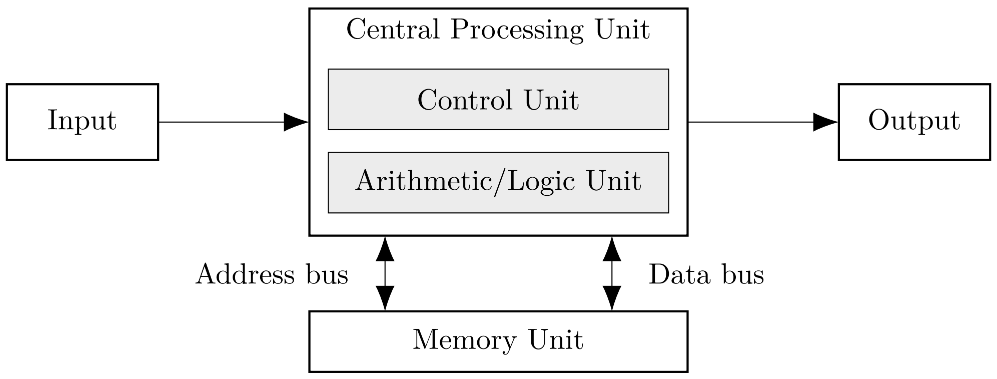


Figure 5.2: Von Neumann processor architecture.

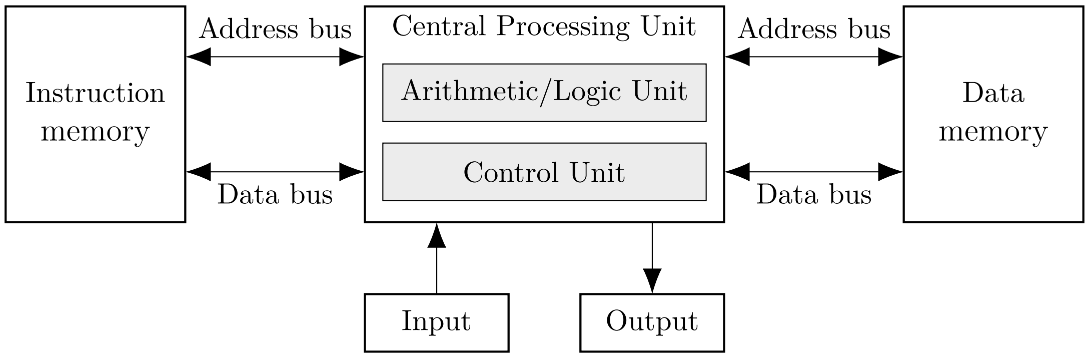


Figure 5.3: Harvard processor architecture.

requires memory access, the next instruction will be fetched after the previous instruction is processed completely. Thus, the shared memory for both instruction and data can be seen as a bottleneck in the von Neumann architecture [63].

DSPs are designed to quickly process a large amount of data requiring high-speed data fetching from memory. Therefore, most DSPs are based on the Harvard processor architecture introduced by IBM at Harvard University in 1944. It is visualized in Figure 5.3. The processor has two separate internal memory spaces combined with separated address buses and data buses. One memory is dedicated to the program instructions and one for the data. The processor can fetch program instructions and data parallelly using separate buses. This separation enables higher throughput and decreased latencies in signal processing tasks.

5.1.3 Fixed-point representation

Digital signal processing with numbers containing fractional parts is typically executed using two different arithmetics, fixed-point and floating-point arithmetics. Most GPPs perform arithmetic using floating-point representation and apply the IEEE-754 floating-point number standard in their arithmetic units [64]. The standard describes how the binary representation of numbers with fractional parts is formed and how the

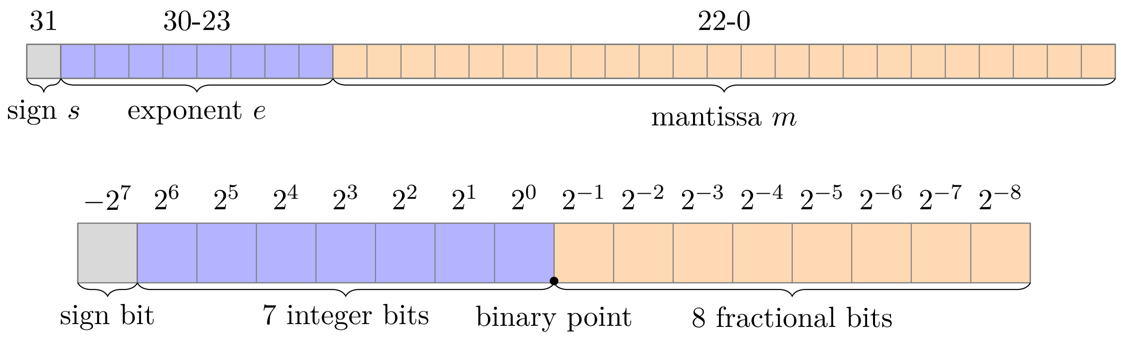


Figure 5.4: **Top**: Bit pattern of a 32-bit single precision floating point number in standard IEEE-754 format. **Bottom**: Bit pattern of a 16-bit Q8.8 fixed point number and the corresponding powers of two.

arithmetic operations are executed. A 32-bit single-precision floating-point number in IEEE-754 standard format consists one sign bit s, eight exponent bits e and 23 mantissa bits m. Bit positions are visualized in Figure 5.4. The real value r given by the binary sequence can be computed as

$$r = (-1)^s \cdot (1.m) \cdot 2^{e-127} \tag{5.1}$$

Because all bit positions and lengths are fixed, the IEEE-754 standard gives unambiguous real number representation for all possible 32-bit sequences. Floating-point numbers have a dynamic range, i.e. with a fixed number of digits, numbers of different orders of magnitude can be presented. The exponent part of the number indicates the binary point position of the numbers. The point can "float", which refers to the name of the representation.

Many DSPs and other embedded devices support only integer arithmetic and have limited or no support for floating-point numbers. Numbers with fractional parts can be processed with integer operations applying fixed-point representation. Fixed-point numbers are integers that are scaled to include the information of the fractional part of the number, and the arithmetic can be performed with the same integer arithmetic units as with the ordinary integers. However, the binary sequence of an integer representing a fixed-point number cannot be converted to a real number without additional information about the correct scaling factor. The developer is responsible for handling this scaling factor when using fixed-point arithmetic.

One widely applied notation for fixed-point numbers is Q-format. It is a short description of how two's complement integers can be interpreted as fractional numbers. The format $Qm.n$ describes a fixed-point number with $m-1$ integer bits and n fractional bits stored in a signed two's complement integer with a total of $N = m + n$ bits. Fixed-point numbers can take values from the subset given by

$$P = \left\{ p/2^n \mid -2^{N-1} \leq p \leq 2^{N-1} - 1, p \in \mathbb{Z} \right\}. \tag{5.2}$$

An example bit pattern of a $Q8.8$ fixed-point number is illustrated in Figure 5.4. The real value interpretation $r \in \mathbb{R}$ of a $Qm.n$ formatted integer $x \in \mathbb{Z}$ can be computed as

$$r = 2^{-n} \left(\sum_{i=0}^{N-2} 2^i x_i - 2^{N-1} x_{N-1} \right), \tag{5.3}$$

where x_i is the ith bit of x [65]. In the above defition, the most-significant bit in a signed two's complement number is the referred as the sign bit, and determines also the sign of the fixed-point representation. The conversion between from floating-point number x_{float} to fixed-point number $x_{Qm.n}$ in $Qm.n$ format using $N = m + n$ bits is given as

$$x_{Qm.n} = \text{int}\left(2^n \cdot x_{float}\right), \tag{5.4}$$

where operation $\text{int}(\cdot)$ rounds a floating-point number to nearest integer and casts the result to a signed N bits wide integer. Inverse conversion from fixed-point to floating-point is given as

$$x_{float} = 2^{-n} \cdot \text{float}\left(x_{Qm.n}\right), \tag{5.5}$$

where operation float$(\cdot)$ converts a signed integer to the closest floating-point representation of the same integer. As said earlier, the position of the binary point must be tracked manually by the developer and cannot be observed straight from the binary format of the fixed-point number. In addition to this there are some practicalities, that should be considered when performing fixed-point arithmetic

- **Addition**: When adding two fixed-point numbers $Qm.n$ and $Qp.q$, there must hold $m = p$ and $n = q$, i.e. the number of fractional bits must match between the summands. The summation result format is $Q(m + 1).n = Q(p + 1).q$.
- **Multiplication**: When multiplying two fixed-point numbers $Qm.n$ and $Qp.q$, the result format is $Q(m + p).(n + q)$.
- **Division**: When dividing fixed-point numbers $Qm.n$ with fixed-point number $Qp.q$, the result format is $Q(m + q).(n + p)$.

Information loss is possible because fixed-point arithmetic often produces more bits than the operands. The intermediate results of algorithms applying fixed-point numbers must be scaled, rounded, and truncated to avoid overflows. Limited bit-width might cause significant precision loss that should be taken into account when designing algorithms for devices using fixed-point numbers [65].

As seen earlier, the most significant advantage of floating-point arithmetic is much more simplified programming enabling the usage of higher-level languages. With floating-point values, a programmer has more flexibility as it is not needed to follow the position of the binary point and take care of the scaling, truncating, and rounding of the intermediate results. The floating-point model also has a broader dynamic range and precision, making it easier to avoid overflow and possible information loss. However, floating-point arithmetic is much more complex at the hardware level than much simpler integer arithmetic required by the fixed-point numbers. It requires more complicated devices that occupy more hardware space, are more expensive, and have greater power consumption. Floating-point units might also support limited parallelism, leading to some performance drawbacks. Therefore, a fixed-point implementation should be considered in algorithms that do not require dynamic range and greater precision of floating-point values.

5.2 CEVA-XC4500

The target SoC of this work contains several CEVA-XC4500 DSP cores optimized for advanced wireless infrastructure equipment. Target DSP utilizes effectively very-long instruction word (VLIW) and single instruction, multiple data (SIMD) concepts. It is a scalable solution for many different signal processing tasks essential in wireless communication. Fully programmable architecture provides more flexibility than dedicated hardware accelerators, which are also extensively used in the target SoC to accelerate physical layer computational loads [48].

CEVA XC4500 offers a high parallelism level, low power consumption and high code density. The primary mechanism for instruction-level parallelism is VLIW, in which multiple instructions from different functional units can be executed simultaneously in one cycle. Data-level parallelism is based on SIMD features extensively used in many vectorized operations. The DSP instruction set contains many vector operations, including basic arithmetic, multiply and accumulate (MAC) and bit manipulation. The manufacturer provides a specialized C-level compiler to support VLIW and SIMD capabilities [48].

5.2.1 Architecture

The CEVA-XC4500 target DSP is based on the Harvard architecture with separate program instruction and data memories. It contains 128 kilobytes of internal program instruction memory and 256 kilobytes of internal data memory. However, the DSP support up to 4 GB of unified memory spaces using four physical interfaces, one for instruction memory and three for data memory [66]. Data transfer between internal and external memories is done using dedicated DMA engines of the DSP.

CEVA-XC4500 instruction set is based on a load-store computer architecture utilizing Reduced instruction set computer (RISC) operations and instructions only. The DSP has dedicated load/store units handling memory accesses. Units load data directly from the memory to the registers and store data from registers to the memory. These registers serve as sources and destinations for all other computation instructions. All loading instructions support only 4-byte (32 bits) aligned memory addresses. Because the fixed-point numbers in this work occupy mostly 2 bytes, this limitation causes some suboptimal usage of vectorization properties, which are discussed more in Chapter 6. When storing data in memory, also 2-byte aligned addresses are accessible.

The most important functional units of the DSP are [48]:

- The General Computation unit (GCU) handles all the general non-vectorized computations and bit-manipulation operations. It has four independent subunits, M0, M1, L, and S, that can execute instructions in parallel. GCU handles the Accumulate register file (ACF) consisting of 24 40-bit accumulators.

- The Data Address and Arithmetic Unit (DAAU) is responsible for all data memory accesses. It has two units, LS0 and LS1, capable of loading and storing from/to the data memory using various supported addressing modes. Addressing Register File (ARF) contains 25 32-bit registers for addressing purposes. A Scalar unit (SC) is also part of the DAAU and can perform arithmetic, logical, and shift operations on the ARF registers without affecting the accumulators. SC enables post-modification of pointers, i.e. the DAAU can increment and decrement pointers from which data is accessed, parallel with address generation.

- The Program Control Unit (PCU) aligns instructions from the program memory and dispatches them to different functional units. PCU also manages the program counter and correct program flow.

- Two Vector Computation Units (VCUs) perform all vector computations and vector bit-manipulation operations. Each VCU consists of three independent computational units and a vector register file (VRF). VCU is the core of the inference implementations in this work and is covered in more detail in Section 5.2.2.

5.2.2 Vector computation unit

Two VCUs support SIMD operations, the main parallelization mechanism in the CEVA-XC4500 DSP. Both VCUs contain three independent functional units:

- Vector arithmetic unit (VA) performs the arithmetic operations such as addition, subtraction, multiplication, and multiply-accumulation (MAC).
- Vector bit manipulation unit (VB) performs the bit-manipulation operations and generic 32-bit arithmetic operations such as bit insertion, bit extraction, bit pack, matrix transpose, and shift.
- Vector Move and Pack Unit (VM) performs vector manipulation operations such as moving data, packing, unpacking and scaling.

In addition to functional units, each VCU consists of a Vector Register File (VRF) that contains following register types:

- Vector Input (vi): Twelve 256-bit registers
- Vector Orthogonal (vo): Four 320-bit registers
- Vector Common (vc): Four 32-bit registers

The VA, VB, and VM units operate in different execution stages. As a result, various instructions take a different number of cycles to execute. The number of stages an instruction requires for execution determines the cycle penalty before another instruction can use the result. The instruction set for the target DSP contains mainly instructions for fixed-point numbers with variable bit widths from 8 bits up to 72 bits. However, most instructions support only 16-bit and 32-bit fixed-point numbers, called words and double-words, respectively. There is also limited support for floating-point numbers, including basic arithmetic operations such as addition and multiplication.

DSP implements so-called instruction replication to save program code and power consumption. By default, each vector instruction is executed identically in both VCUs, each applying the same instruction to a different set of data. The replication mechanism is flexible and can be controlled via dedicated mode-bits for each VCU, enabling execution only in a single VCU. There are also instructions for inter-vector operations that can combine results between VCU. Moving data across VCUs requires additional instructions, increasing processing overhead.

Conditional clauses in the DSP code lead to branching, accomplished with conditional jumps and calls. Conditions must be resolved before the processor can proceed further. Introducing a conditional branch instruction to the processing pipeline causes a four-cycle delay until the condition is read. The default behaviour is that the DSP guesses that the jump is not taken not to break the sequential pipeline operation. If this guess is incorrect, the pipeline must be flushed, leading to a 3 to 7 cycles penalty. If such branched code is executed often, for example, in a loop, the delays may add up to a very significant portion of the processing time [67].

With CEVA-XC4500, sequential code should be preferred whenever possible. Many signal processing algorithms require performing operations based on some condition. To prevent simple conditional operations from causing cycle penalties, the target DSP supports a predicate mechanism for almost all vector operations [48]. Each atomic operation of a VCU instruction can be conditioned on a 16-bit predicate register. A zero predicate bit in the predicate register does not prevent the execution of the atomic operation itself but only blocks writing its result to the output vector register. Thus, a zero predicate bit defends a word or double-word part of the output vector, and the part preserves its current value.

5.2.3 Compiler and intrinsic functions

The CNN inference implementation in this work is written in the C++ programming language using a specialized C/C++ compiler provided by the DSP manufacturer. The SIMD features of the target DSP are available as intrinsic functions, which are mainly handled as a single machine instruction by the compiler. The few exceptions are macro functions, which are handled as sequences of several machine instructions and perform more complex tasks such as converting floating-point and integer numbers. All SIMD intrinsics that are applied in this work, are described in Tables A.01 and A.02. In the target DSP context, a *word* refers to 16-bit signed or unsigned value, and *double-word* refers to 32-bit signed or unsigned value. In the intrinsic descriptions, HIGH and LOW refer to double-word parts. HIGH means the most significant 16 bits and LOW the least significant 16 bits.

The target DSP is a superscalar processor utilizing VLIW architecture. The instruction-level parallelism is achieved using static scheduling of machine instructions, which can be of 16-, 32-, 48-, 64- or 80-bit in width [68]. Performing the scheduling is mainly delegated to the compiler. When compiler optimization options are enabled, the compiler optimizes intrinsic functions into VLIW instruction packets. The target DSP supports eight-way 256-bit VLIW, i.e. up to eight instructions from different units can be executed in parallel for one processor cycle [48].

As the CNN inference modules are implemented in C++, some responsibility for code optimization is delegated naturally to the compiler. As such, performance improvements can be achieved by modifying the source code to allow the compiler to perform various optimizations.

Chapter 6

Implementation

This work aims to accelerate convolutional neural networks (CNNs) on the target hardware, CEVA-XC4500 DSP, with reasonable efficiency and accuracy. A comprehensive framework for accelerating different CNNs on the target hardware was developed to obtain this goal. In this context, the framework covers the required preprocessing to convert a CNN to a suitable format for the DSP and the performance-critical real-time inference.

The basic idea of the framework in this work is not novel, but CNNs have not yet been implemented for the target hardware to the author's best knowledge. Existing commercial DL frameworks, as well as the solutions introduced in [9] and [10] share at least some of the ideas. In [10] accelerating MLPs (fully-connected layers) in the target DSP was thoroughly studied. However, from the implementation point of view, CNNs are fundamentally different compared to ordinary MLPs and must be treated a bit differently.

The solution introduced here is tailored for one particular DSP, and identical implementations do not necessarily exist. The aim is to produce robust inference for this particular DSP, not to support multiple hardware. However, the general principles are not completely limited to this DSP, and some ideas can be leveraged to other hardware, even though the source code is not portable. Because the target hardware is not primarily designed to accelerate NN inferences, the framework lacks some flexibility and cannot be applied to any arbitrary CNN. The scope of this work was restricted to only 1D CNNs for simplicity reasons, as 1D CNNs already have existing applications in the 5G physical layer [5]. Extending the framework's support to 2D CNNs and beyond is left for future research.

The previous three chapters cover the relevant background related to neural networks and target hardware. This chapter presents the framework's main ideas applied to accelerate CNNs in the target DSP. The Chapter is distributed into three Sections. In the first Section, an overview of the solution is provided. The second Section describes how the network and its parameters are preprocessed before the performance-critical real-time inference is called. The parts of the real-time inference are described in the last Section.

6.1 Overview

The DSP job system controls the computational flow of the DSP. To make this framework compatible with the job system and the existing software architecture on the SoC, it must fulfil some requirements. The GPP of the SoC initializes DSP jobs by creating a job description. The description contains all necessary information for running the DSP job; in this context, the real-time NN inference. The job description consists of the job configuration section and addresses where input and output data are located in the SoC shared memory. The description is transferred to a hardware scheduler that enqueues the job to one of the SoC's DSP-cores and initializes input data transfer from SoC shared memory to DSP internal memory via the DMA engine. DSP then runs the job according to the job description's configuration section. The output is transferred back to SoC shared memory via DMA, or another job continues processing on the same DSP core using the previous job's output as an input.

Figure 6.1 illustrates the framework to run a CNN in the DSP. The large box on the left-hand side contains the **preprocessing** part executed outside SoC and has to be done only once for each model. In this part, the configuration section of the job description is formed, and all network parameters are quantized to a fixed-point format. All details of the CNN inference should be provided in the configuration section, which is described in Section 6.2. The large box on the right-hand side contains the performance-critical real-time inference due to optimization using the target DSP vectorization capabilities. All input for the real-time inference must be in one contiguous memory block addressed in the job description to be transferred to DSP internal memory via DMA. Input includes the input data and all network

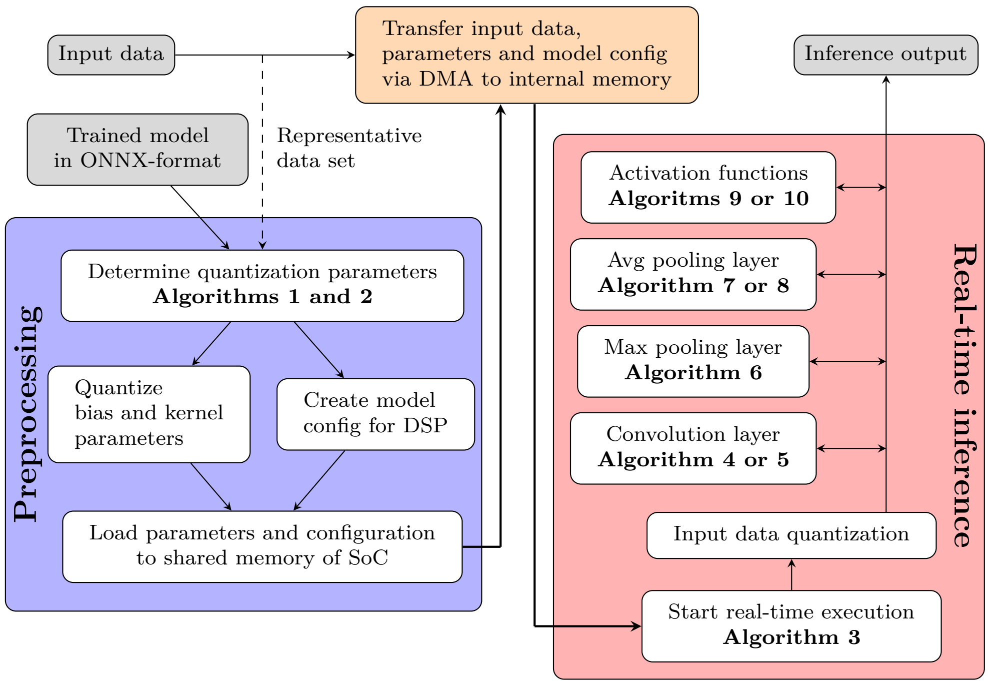

Input data
Transfer input data, parameters and model config via DMA to internal memory
Inference output
Trained model in ONNX-format
Representative data set
Preprocessing
Determine quantization parameters
Algorithms 1 and 2
Quantize bias and kernel parameters
Create model config for DSP
Load parameters and configuration to shared memory of SoC
Real-time inference
Activation functions
Algoritms 9 or 10
Avg pooling layer
Algorithm 7 or 8
Max pooling layer
Algorithm 6
Convolution layer
Algorithm 4 or 5
Input data quantization
Start real-time execution
Algorithm 3

Figure 6.1: An overview of the framework to run CNN inference in the target DSP.

parameters (i.e. kernels and biases). All performance-critical code is written in C++ and compiled with a compiler optimization level of -O3.

6.2 Preprocessing

Target DSP is designed for 16-bit and 32-bit fixed-point numbers, and the best performance is achieved when 16-bit numbers are used whenever possible [67]. Therefore, the trained ONNX floating-point network is converted into a fixed-point format. As a part of the preprocessing, all kernel tensors and bias vectors are extracted from the floating-point model, quantized and stored in a contiguous memory block, ready to be transferred to the SoC shared memory. Kernel vectors that are odd in length are padded with extra zeros at the end of the vector. Zero padding must be added because the target DSP supports only 32-bit aligned load instructions. The effect of this restricted memory access is explained in more detail in Section 4.3.1.

In addition to parameter quantization and kernel padding, the configuration describing the structure of the network inference is formed in this part. Information for the configuration section of the job description is retrieved from the floating-point model and the quantization procedure. All algorithms and procedures of this Section are implemented in Python. Because the complexity of the preprocessing is not critical for real-time inference, the performance of this part is not measured or evaluated in this work.

Parameter quantization

As discussed in Section 3.4.3, there exist many different methods to perform network quantization. Inference inputs and kernel weights are quantized to 16-bit fixed-point

Algorithm 1 Estimate the number of guard bits.

```
    Input: Kernel tensors $\boldsymbol{\omega}_1, \dots, \boldsymbol{\omega}_n$,
           Numbers of output channels $C_1^{(out)}, \dots, C_n^{(out)}$
 1: procedure EstimateGuardBits($\{\boldsymbol{\omega}_1, C_1^{(out)}\}, \dots, \{\boldsymbol{\omega}_n, C_n^{(out)}\}$)
 2:     $g_B \leftarrow 0$
 3:     for $i \leftarrow 1$ to $n$ do                                   ▷ Loop kernel tensors
 4:         $m \leftarrow 0$
 5:         while $j < C_i^{(out)}$ do                                   ▷ Loop output channels
 6:             $S \leftarrow \mathrm{sum}(\mathrm{abs}(\boldsymbol{\omega}_i[j, :, :])$         ▷ Absolute sum of node weights
 7:             $m \leftarrow \max(m, S)$
 8:             $j \leftarrow j + 1$
 9:         end while
10:         $g_i \leftarrow \lceil \log_2(\lfloor m \rfloor + 1) \rceil$                  ▷ Number of guard bits in layer $i$
11:         $g_B \leftarrow \max(g_B, g_i)$                          ▷ Update global number of guard bits
12:     end for
13:     return $g_B$                    ▷ Return estimate for the required number of guard bits
14: end procedure
```

numbers and the bias parameters to 32-bit fixed-point numbers. The quantization procedure applies a post-training approach with a small representative input data set. The optimal amount of fractional bits for kernel weights, biases, inputs and outputs are determined layerwise for each convolution layer. In this implementation, pooling layers and activation functions are designed not to change the fixed-point format. However, average pooling and activation function implementations need the current number of fractional bits of the input as a part of the layer configuration.

Determining the required amount of fractional bits as a part of the preprocessing enables efficient real-time computations using simple operations. Biases can be added directly to the results of multiplication in convolution layers. 40-bit activations of the multiply-accumulate operations in the layers can be normalized simply by bit-shifting to the resolution required by the subsequent layer and discarding the excess bits.

If the activations values of some layer happen to require more integer bits than the current Q-format for the layer output has, non-redundant integer bits are discarded,

Algorithm 2 Determine the number of fractional bits.

```
Input: Representative input data $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$,
       Kernel tensors $\boldsymbol{\omega}_1, \ldots, \boldsymbol{\omega}_k$,
       Bias vectors $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_k$,
       Number of guard bits $g_B$ from Algorithm 1
 1: procedure DetermineFracBits($\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, \{\boldsymbol{\omega}_1, \boldsymbol{b}_1\}, \ldots, \{\boldsymbol{\omega}_k, \boldsymbol{b}_k\}, g_B$)
 2:     $\{m_0, \ldots, m_k\} \leftarrow \{0, \ldots, 0\}$        ▷ Initialize max activations of each layer
 3:     for $i \leftarrow 1$ to $n$ do
 4:         $m_0 \leftarrow \max(m_0, \max(\text{abs}(\boldsymbol{x}_i)))$        ▷ Update max input absolute value
 5:         $\boldsymbol{a}_0 \leftarrow \boldsymbol{x}_i$
 6:         for $j \leftarrow 1$ to $k$ do
 7:             $\boldsymbol{a}_j \leftarrow f_j(\boldsymbol{a}_{j-1}; \boldsymbol{\omega}_j, \boldsymbol{b}_j)$        ▷ Process layer
 8:             $m_j \leftarrow \max(m_j, \max(\text{abs}(\boldsymbol{a}_j)))$        ▷ Update max activation for $j$:th layer
 9:             $\boldsymbol{a}_j \leftarrow \varphi_j(\boldsymbol{a}_j)$        ▷ Process activation function
10:         end for
11:     end for
12:     $f_0^{(a)} \leftarrow 15 - \lceil \log_2 m_0 \rceil - \lceil g_B/2 \rceil$        ▷ Input fractional bits
13:     for $j \leftarrow 1$ to $k$ do
14:         if layer type == Conv then
15:             $f_j^{(a)} \leftarrow 15 - \lceil \log_2 m_j \rceil - \lceil g_B/2 \rceil$        ▷ Activation fractional bits
16:             $f_j^{(w)} \leftarrow 15 - \lceil \log_2(\max(\text{abs}(\boldsymbol{\omega}_j))) \rceil - \lfloor g_B/2 \rfloor$        ▷ Kernel fractional bits
17:             $f_j^{(b)} \leftarrow f_{j-1}^{(a)} + f_j^{(w)}$        ▷ Bias fractional bits
18:         else
19:             $f_j^{(a)} \leftarrow f_{j-1}^{(a)}$        ▷ Activation fractional bits
20:         end if
21:     end for
22:     return $f_0^{(a)}, \ldots, f_k^{(a)}, f_1^{(w)}, \ldots, f_k^{(w)}, f_1^{(b)}, \ldots, f_k^{(b)}$
23: end procedure
```

which causes a significant error. Preventing overflow, i.e. discarding non-redundant during the activation normalization to 16-bits, is done using so-called **guard bits**. In this procedure, the resolution of the kernel weights and inputs is reduced to leave redundant bits in the normalized 16-bit outputs. If the resolution of kernel weight and layer input is reduced by one bit each, their fixed-point product occupies two bits less. The product has then room to overflow two bits more without compromising the accuracy of the 16-bit normalization due to discarding non-redundant bits. The selection of the number of guard bits balances between avoiding overflows and losing resolution. The guard bit estimation procedure applied in this work is presented in Algorithm 1. The estimation applies the maximum absolute sum of kernel weights that contribute to a single activation in the network. The number of guard bits is adjusted such that it has an equal amount of bits to the integer part of this maximum absolute sum. It estimates both the contribution of the total number of accumulations in a single activation and the scaling effect caused by usually small kernel weights [69]. The guard bit estimate g_B provided by Algorithm 1 is used for every layer throughout the network.

After an estimate for the number of guard bits is obtained, the number of fractional bits for the network input, network parameters and activations are obtained with Algorithm 2. The approach is somewhat similar to the MLP quantization in [10]. However, because this work operates with CNNs, there are some differences. The algorithm proceeds in the following manner:

1. The input of the first layer, i.e. the input data, is quantized using a representative data set. The largest absolute value of all representative inputs is recorded in m_0. The selected Q-format is a number with the highest possible resolution, i.e., the largest amount of fractional bits, which can be used to represent m_0. The number of fractional bits for the input data denoted $f_0^{(a)}$ is computed by subtracting the required number of integer bits and sign bit from the available 16 bits:

$$f_0^{(a)} = 16 - (\max(0, \lceil \log_2 m_0 \rceil) + 1) = 15 - (\max(0, \lceil \log_2 m_0 \rceil). \tag{6.1}$$

   In Q-format, which is explained in Section 5.1.3, this equals to

$$\mathrm{Q}\left(16 - f_0^{(a)}\right).\left(f_0^{(a)}\right). \tag{6.2}$$

   In the preprocessing part, only the Q-format for the input is determined. The actual input quantization is performed during the real-time inference.

2. For the inputs of the subsequent layers, i.e. the activations, the amounts of fractional bits are determined by running the floating-point network using a representative data set. In the Algorithm, each layers' maximum activation m_j is recorded. The number of fractional bits in the jth layer $f_j^{(a)}$ is then adjusted such that this maximum m_j can be represented in this format with the highest possible resolution as in Equation (6.1). If the layer is a pooling layers, the activation fractional bits are preserved, i.e. $f_j^{(a)} = f_{j-1}^{(a)}$. If the convolution layer includes the Leaky Relu activation function, the α-parameter is quantized to the same format as the layer input, i.e. the number of fractional bits is $f_{j-1}^{(a)}$.

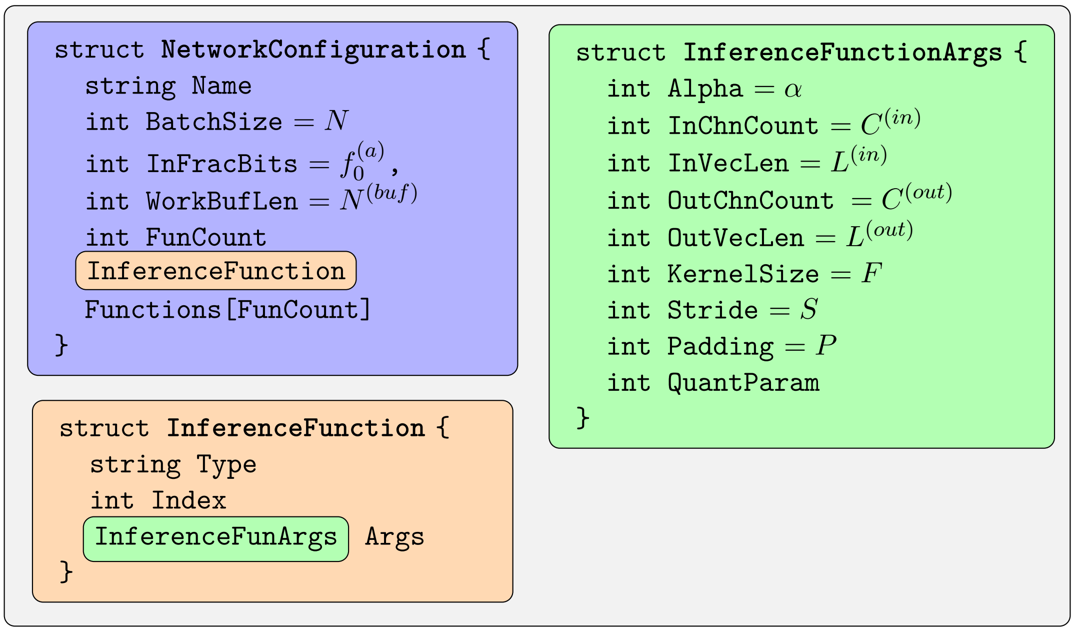
3. For the kernel weight quantization, each layer's largest absolute kernel weight value is recorded. The number of fractional bits $f_j^{(w)}$ is selected based on this value similarly as previously with activations.

4. For the 32-bit bias parameters in the jth layer, the number of fractional bits is obtained simply just summing the number of fractional bits of the layer inputs and the kernel weights
$$f_j^{(b)} = f_{j-1}^{(a)} + f_j^{(w)} \tag{6.3}$$
Because the jth layer's convolution product contains the same number of fractional bits, no normalization is required when the bias term is added.

Configuration creation

As a part of model preprocessing, the details of the network architecture and additional parameters for each layer are extracted from the floating-point model and converted into a suitable format for the target DSP. In addition, some parameters related to quantization are extracted from the Algorithm 2. The information is stored in the configuration section of the DSP job description. The content of the configuration section does not have a fixed format but is typically defined as a C-type struct. The structure of the DSP job configuration is illustrated in Figure 6.2.

Model configuration is described in a struct containing members for parameters necessary for the real-time inference execution, such as the number of fractional bits of input data, batch size, and the sizes of the fixed-size working buffers. In addition to parameter members, the top-level `NetworkConfiguration` struct also contains an array for the inference function members. These `InferenceFunction` structures contain information for running specific parts of the real-time inference.



```
struct NetworkConfiguration {
   string Name
   int BatchSize = $N$
   int InFracBits = $f_0^{(a)}$,
   int WorkBufLen = $N^{(buf)}$
   int FunCount
   InferenceFunction
   Functions[FunCount]
}

struct InferenceFunction {
   string Type
   int Index
   InferenceFunArgs Args
}

struct InferenceFunctionArgs {
   int Alpha = $\alpha$
   int InChnCount = $C^{(in)}$
   int InVecLen = $L^{(in)}$
   int OutChnCount  = $C^{(out)}$
   int OutVecLen = $L^{(out)}$
   int KernelSize = $F$
   int Stride = $S$
   int Padding = $P$
   int QuantParam
}
```

Figure 6.2: An illustration of the configuration section of the DSP job description.

All individual layers or activation functions are implemented as separate functions in this work. Functions are configured in `InferenceFunction` structs that have a tag and index members indicating which of the implemented functions should be executed. The struct also contains a lower-level `InferenceFunctionArgs` struct that includes the actual function arguments, which are passed as arguments to the implemented functions. The inference function arguments have a standardized format, so all functions receive the same type of struct as an argument. However, all argument are not needed to run the specific functions, for example `Padding`, `Stride` and `KernelSize` are needed only in convolution and pooling layers. `args.QuantParam` contains the post-shift coefficient for output normalization or the number of fractional bits in the layer input if the inference function requires the information.

6.3 Real-time inference

The real-time inference implementation is the performance-critical part of the NN inference, and therefore the program cycle counts are tracked only from this part of the inference. The performance-critical part starts after the model has been preprocessed. First, the program code, the job configuration, input data and network parameters are transferred to the internal memory via DMA. The inference implementation, described in Algorithm 3, calls sequentially multiple functions until the entire network inference is processed. In this framework, these functions forming the actual body of the real-time inference are called **inference functions**. All the functions calls are made according to the network configuration given in the DSP job description. The configuration also contains all the additional parameters needed for running the functions, such as parameters related to quantization and input/output dimensions. The workflow of the real-time inference is heavily influenced by the workflow developed in [10], but have some significant differences. Overall, the solution may also share many similarities with deep learning inference frameworks available commercially.

In this work, the input data format is assumed to be in the so-called "channels first" -format. It means that the network input and each layer activations have the following structure $\boldsymbol{x} \in \mathbb{R}^{N \times C \times L}$, where N is the batch size, C is the number of channels, and L is spatial extend of the channels. This data format does not require special flattening between convolution and fully-connected layers. The data is already in a continuous memory block stored in the correct order. Thus, in the future, it is possible to add fully-connected layers easily to models using principles presented in [10]. If another data format such as "channels last" is used, the data must be converted to the correct format. If this conversion is done as a part of the real-time inference, it might decrease performance significantly.

The floating-point input data is quantized before the first inference functions are called. This so-called real-time quantization is also vectorized. This vectorization procedure is described detailed in [10]. The same approach applies to the standard ReLU-activation function, which was studied as well in [10]. Even though the vectorization procedure of input quantization and ReLU activation is not repeated, the contribution to the overall performance of the networks is studied as a part of

Algorithm 3 The workflow of the real-time inference.

```
  Input: Network configuration config,
         Pointers to data arrays input_ptr, output_ptr
         Pointers to parameter arrays kernel_ptr, bias_ptr
 1: procedure NetworkInference(config, input_ptr, output_ptr,
        kernel_ptr, bias_ptr)
 2:     in_buf[config.workingBufferSize]            ▷ Fixed-size working buffers
 3:     out_buf[config.workingBufferSize]
 4:     in_buf ← QuantizeInput(config.InputFracBits, input_ptr)
 5:     while $i <$ config.FunctionCount do       ▷ Process all inference functions
 6:         fun ← config.Functions[$i$]
 7:         switch fun.Type do
 8:             case Convolution layer:
 9:                 ProcessConvLayer(fun.Args, kernel_ptr,bias_ptr,
                     in_buf,out_buf)
10:                 break
11:             case Max pooling layer:
12:                 ProcessMaxPoolLayer(fun.Args, in_buf, out_buf)
13:                 break
14:             case Average pooling layer:
15:                 ProcessAvgPoolLayer(fun.Args, in_buf, out_buf)
16:                 break
17:             case Sigmoid:
18:                 ProcessSigmoid(fun.Args, in_buf,out_buf)
19:                 break
20:             case Leaky ReLU:
21:                 ProcessLeakyRelu(fun.Args, in_buf, out_buf)
22:                 break
23:             case ReLU:
24:                 ProcessRelu(fun.Args, in_buf, out_buf)
25:                 break
26:         end switch
27:         swap in_buf and out_buf pointers
28:         $i \leftarrow i + 1$
29:     end while
30:     copy in_buf to output_ptr
31: end procedure
```

the performance evaluation in Chapter 8.

This work aims to modify program code to apply the target hardware features to speed up the computations in different inference parts. The main method for the improvements is vectorization, for which the CEVA DSP offers extensive support. In this Section, the vectorization of individual inference functions is described. The pseudocodes in this Section try to catch the main ideas behind the vectorization of each function. All vectorized intrinsics that are used in the pseudocodes are explained in Tables A.01 and A.02. Pseudocodes cannot be directly translated to code that can be executed in CEVA-XC4500 DSP, and some of the details have been abstracted away for clarity reasons. For example, the effect of zero-padding on the program code is omitted from the pseudocodes to keep them even somewhat compact. When zero padding is applied, it has to be handled every time vector heads and tails are processed. In practice, it is implemented using a vector predicate mechanism. When batch sizes containing multiple input data samples are processed, the Algorithms illustrated in this Section are iterated until all input samples have been processed. At the Algorithm level, it means an additional for-loop that iterates through all input samples.

6.3.1 Convolution layer

For the convolution layer DSP implementations, two different vectorization approaches are applied. The first approach only supports convolution layers that use stride parameter $S = 1$. The second approach is applied with convolution layers that utilize stride parameter $S > 1$. Both approaches are based on vectorization of direct computing of convolution (or cross-correlation). There are also other approaches to speed up the convolution layer computation that are discussed in Section 4.3.1. The modification of the direct computation was chosen in this work as it is the best approach for the target hardware. The effects of convolution layer vectorization on the inference performance is discussed in Section 8.2.3.

First vectorization approach: Kernels as coefficients

The first convolution layer optimization approach uses effectively parallelized multiply-accumulate (MAC) operations that the target DSP supports. The implementation is illustrated in Algorithm 4. The basic idea of the approach is to maximize the number of output elements stored back in the SoC shared memory in one iteration. The Algorithm iterates overall output channels producing 28 elements of the output channel with each iteration. All input channels are iterated within each 28 element chunk as all input channels contribute to each output element. Adding the bias term is done initially, where every element of the output vector is initialized with bias values (line 6).

The two innermost loops of the Algorithm are illustrated in Figure 6.3. Only eight elements are shown in the Figure for both VCUs instead of the 16 elements to save space. As seen, kernel elements are loaded as a coefficient. The target DSP supports only 32-bit aligned memory access when loading registers. In addition, coefficient registers are 32-bit in width, and hence two 16-bit kernel elements are loaded simultaneously. Also, the input vector is loaded in the intervals of two elements, as seen in lines 15 and 16. Different parts of the 32-bit doublewords in kernels and input vectors are accessed using LOW and HIGH. As the keywords operate separately in both VCUs, loading/storing input and output elements from memory must be done separately for both VCUs.

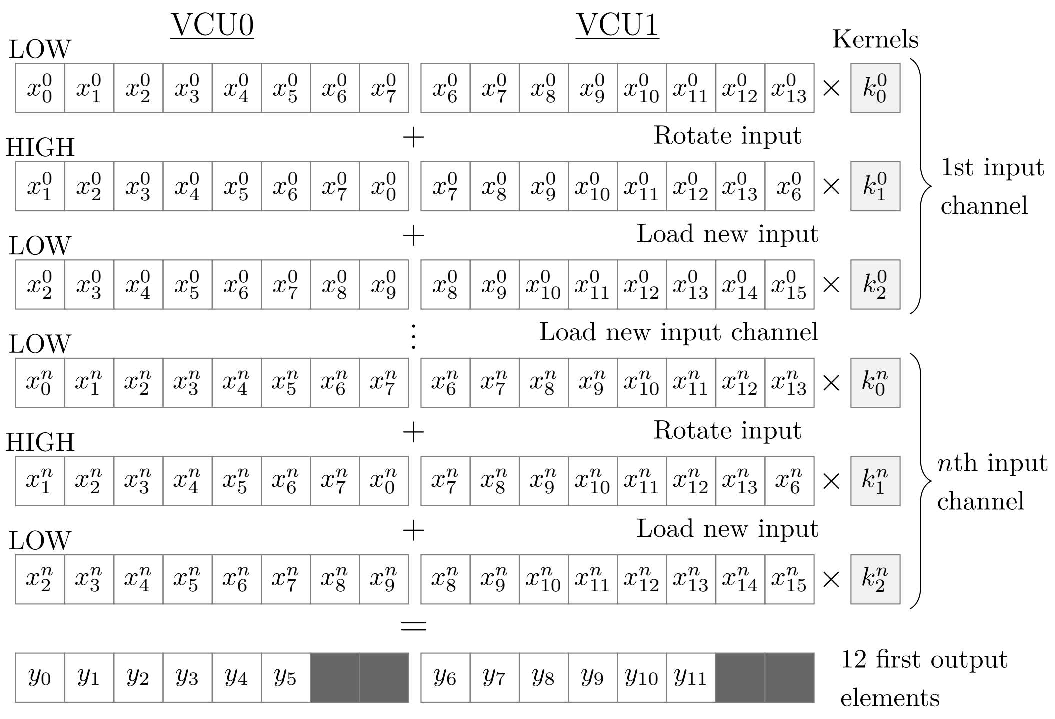


Figure 6.3: An illustration of the first vectorization approach that handles kernel elements as coefficients. The Figure shows processing the first 12 elements of one output channel when the input has n input channels with a kernel size of three, no padding and stride equal to one, i.e. $(F, S, P) = (3, 1, 0)$. The target hardware VCUs can process 16 elements simultaneously, but the Figure visualizes only eight elements to save space.

Due to the 32-bit aligned memory access in vector register loading, input registers are updated from memory only every second iteration. Every second iteration, they are rotated using LOW/HIGH keywords, resulting in that all 32 output elements are not usable. Two dummy elements must be left in the output registers in both VCUs, which totals 28 usable output elements. If the kernel length is odd, one extra iteration must be performed, which is seen in line 21. In the pseudocode, this is written with if-clause. Because conditional code should be avoided with the target DSP, this conditional is executed in practice such that completely separate functions are made for odd and even size kernels. In the case of odd kernels, to keep the memory pointer correctly aligned, an extra zero is added to the end of each kernel as a part of the preprocessing discussed in Section 6.2. The output elements are accumulated into 40-bit wide integers in the output vector register $\boldsymbol{o}$. The values are normalized back to a 16-bit fixed-point number using the normalizing bit-shift determined as a part of the preprocessing and stored in `args.QuantParam` (lines 28-29).

Handling zero-padding and vector tails are abstracted away from the pseudocode. Zero-padding is handled with the vector predicate mechanism, and vector tails are handled using the so-called *overwriting* technique [67]. It means that only the whole

Algorithm 4 Vectorized convolution layer handling kernels as individual coefficients.

```
1: function ProcessConvLayer(args, kernel_ptr, bias_ptr, input_ptr, out_ptr)
2:     for all output channels do
3:         input_mov_ptr $\leftarrow$ input_ptr                                    $\triangleright$ Reset pointer
4:         $b \leftarrow$ c_load(bias_ptr); bias_ptr $+= 1$
5:         for all 28 element chunks of the output vector do
6:             $\boldsymbol{o} \leftarrow$ v_fill($b$)                              $\triangleright$ Fill with bias parameter
7:             input_chn_ptr $\leftarrow$ input_mov_ptr                             $\triangleright$ Reset pointer
8:             input_mov_ptr $+= 28$
9:             k_ptr $\leftarrow$ kernel_ptr                                        $\triangleright$ Reset kernel pointer
10:            for all input channels do
11:                x_ptr0 $\leftarrow$ input_chn_ptr                                $\triangleright$ Reset pointer
12:                x_ptr1 $\leftarrow$ x_ptr0 $+ 14$                                $\triangleright$ Second pointer for VCU1
13:                input_chn_ptr $+=$ args.InputVectorLength
14:                for all 2 element chunks of kernel vector do
15:                    $\boldsymbol{x} \leftarrow$ v_load_vuX(x_ptr0, $0$); x_ptr0 $+= 2$
16:                    $\boldsymbol{x} \leftarrow$ v_load_vuX(x_ptr1, $1$); x_ptr1 $+= 2$
17:                    $k \leftarrow$ c_load(k_ptr); k_ptr $+= 2$
18:                    $\boldsymbol{o} \leftarrow$ vc_mac($\boldsymbol{x}, 0, \text{LOW}, k, \text{LOW}, \boldsymbol{o}$)
19:                    $\boldsymbol{o} \leftarrow$ vc_mac($\boldsymbol{x}, 0, \text{HIGH}, k, \text{HIGH}, \boldsymbol{o}$)
20:                end for
21:                if kernel size is odd then
22:                    $\boldsymbol{x} \leftarrow$ v_load_vuX(x_ptr0, $0$)
23:                    $\boldsymbol{x} \leftarrow$ v_load_vuX(x_ptr1, $1$)
24:                    $k \leftarrow$ c_load(k_ptr); k_ptr $+= 2$
25:                    $\boldsymbol{o} \leftarrow$ vc_mac($\boldsymbol{x}, 0, \text{LOW}, k, \text{LOW}, \boldsymbol{o}$)
26:                end if
27:            end for
28:            $\boldsymbol{o} \leftarrow$ v_shift($\boldsymbol{o}$, $-$args.QuantParameter)     $\triangleright$ Post shift
29:            $\boldsymbol{o} \leftarrow$ v_pack($\boldsymbol{o}$)                      $\triangleright$ Cast to 16-bit elements
30:            out_ptr $\leftarrow$ v_store_vuX($14, \boldsymbol{o}$, out_ptr, $0$); out_ptr $+= 14$
31:            out_ptr $\leftarrow$ v_store_vuX($14, \boldsymbol{o}$, out_ptr, $1$); out_ptr $+= 14$
32:        end for
33:        kernel_ptr $+=$ args.InputChannelCount $\cdot$ args.KernelSize
34:    end for
35: end function
```

28 element chunks are processed. If the length of the vector is not multiple of 28, some dummy values are processed, enabling some flexibility compared to solutions with the exact number of elements produced. Key observations of this implementation:

- Algorithm can only be applied when stride $S = 1$, i.e. the kernel is moved one element at one iteration. When the stride is larger, this approach fails.

- Does not require complicated preprocessing of the network parameters. Only odd size kernel vector must be zero-padded to even length.
- Enables high utilization of vector registers with any kernel size. Due to the 32-bit aligned loading instructions, some dummy values are present in the output registers.
- Bias term addition can be vectorized effectively as the accumulator vector is initialized with bias parameters in line 6.

Second vectorization approach: Kernels as vectors

The second approach for the convolution layer uses the most straightforward way to vectorize dot products between kernels and input vectors. This approach is reported in Algorithm 5. Contrary to the first approach, this method also supports models where stride parameter S larger than one is used. The algorithm produces two output elements with one iteration. In the version presented in Algorithm 5, only kernel sizes up to 16 is supported because the kernel is cloned to both VCUs, that operate parallel the same procedure. However, the algorithm can easily be converted to larger kernels if this VCU parallelization is not applied.

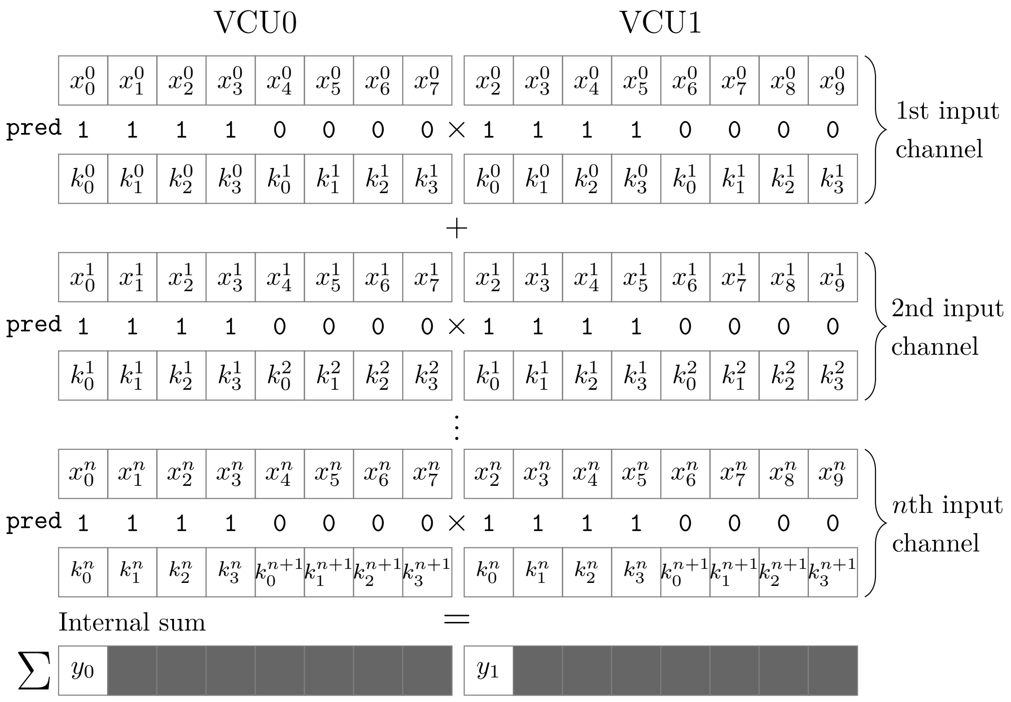


Figure 6.4: Illustration of the second vectorization approach handling kernels as vectors. The Figure shows processing the first two elements of one output channel when the input has n input channels with a kernel size of four, no padding and stride equal to two, i.e. $(F, S, P) = (4, 2, 0)$. VCUs can process 16 elements simultaneously in the target hardware, but only eight are visualized to save space.

Algorithm 5 Vectorized convolution layer handling kernels as vectors. This version is applicable for models with $F \leq 16$.

```
1: function ProcessConvLayer(args, kernel_ptr, bias_ptr, input_ptr, output_ptr)
2:     ptr_shift ← args.InputVectorLength − args.Stride
3:     pred ← vpr_move(0xFFFF ≫ (16 − args.KernelSize))
4:     for all output channels do
5:         input_mov_ptr ← input_ptr                         ▷ Reset moving input pointer
6:         $b$ ← c_load(bias_ptr); bias_ptr += 1              ▷ Bias parameter
7:         for all 2 element chunks of the output vector do
8:             x_ptr ← input_mov_ptr                         ▷ Reset input pointers
9:             input_mov_ptr += (args.Stride ≪ 1)
10:            k_ptr ← kernel_ptr                            ▷ Reset kernel pointer
11:            $\boldsymbol{o}$ ← v_fill(0)                  ▷ Reset accumulator
12:            for all input channels do
13:                $\boldsymbol{x}$ ← v_load_vuX(x_ptr, 0)   ▷ Load input to VCU0
14:                x_ptr += args.Stride
15:                $\boldsymbol{x}$ ← v_load_vuX(x_ptr, 1)   ▷ Load input to VCU1
16:                x_ptr += ptr_shift
17:                $\boldsymbol{k}$ ← v_load(k_ptr)          ▷ Load kernel
18:                k_ptr += args.KernelSize
19:                $\boldsymbol{o}$ ← vv_mac($\boldsymbol{x}$, 0, LOW, $\boldsymbol{k}$, 0, LOW, $\boldsymbol{o}$, pred)
20:            end for
21:            $\boldsymbol{o}$ ← v_add_int($\boldsymbol{o}$)  ▷ Intra-vector sum
22:            $\boldsymbol{o}$ ← v_add($\boldsymbol{o}$, $b$)  ▷ Add bias
23:            $\boldsymbol{o}$ ← v_shift($\boldsymbol{o}$, −args.QuantParameter)
24:            v_store_vuX(1, $\boldsymbol{o}$, output_ptr, 0); output_ptr += 1
25:            v_store_vuX(1, $\boldsymbol{o}$, output_ptr, 1); output_ptr += 1
26:        end for
27:        kernel_ptr += args.InputChannelCount · args.KernelSize
28:    end for
29: end function
```

The procedure taking place inside the two innermost loops of the Algorithm is illustrated in Figure 6.4. Only eight elements per VCU are visualized to save space instead of the 16 elements in the actual hardware. As can be seen, the vector predicate mechanism is applied to disable excess MAC operations if the kernel length is less than 16. After the contribution from all input channels has been summed up, a final intra-vector summation is performed (line 21) to obtain the final result of the dot product sums. Bias term is added with a separate operation in the end. As can be observed from the pseudocode, the second vectorization approach has some suboptimal properties:

- Intra-vector summation is required to combine the partial sums of the dot products resulting in that only two output elements are obtained from one iteration. It is not the ideal use of the target hardware [67].

- The vectorization properties can be utilized properly only if the kernel length equals 16. Otherwise, some of the operations must be switched off with vector predicates.
- Bias parameter has to be added to the result with an additional intrinsic and cannot be vectorized properly.

6.3.2 Pooling layer

Pooling layers are an essential part of the model in most CNN architectures and are discussed detailed in Section 4.3.2. This work implements two different pooling methods: max pooling and average pooling. The implementations provided in this Section are not general examples applicable to any model. Thus, some Algorithms are limited to support only some set of network parameters. The effects of pooling layer vectorization on the inference cycle counts are discussed in Section 8.2.4.

Algorithm 6 Vectorized average pooling layer for unit stride.

```
1:  function ProcessAvgPoolLayer(args, input_ptr, out_ptr)
2:      $k \leftarrow (1 \ll$ args.QuantParameter$)/$args.KernelSize
3:      for all input channels do
4:          for all 28 element chunks of the input vector do
5:              x_ptr $\leftarrow$ in_ptr                              ▷ Reset input pointer
6:              in_ptr $+=$ 28                               ▷ Increase input pointer
7:              $\boldsymbol{o} \leftarrow$ v_fill(0)                          ▷ Reset accumulator vector
8:              for all 2 element chunks of the kernel do
9:                  $\boldsymbol{x} \leftarrow$ v_load(x_ptr); x_ptr $+=$ 2
10:                 $\boldsymbol{o} \leftarrow$ vc_mac$(\boldsymbol{x}, 0, \mathrm{LOW}, k, \mathrm{LOW}, \boldsymbol{o})$
11:                 $\boldsymbol{o} \leftarrow$ vc_mac$(\boldsymbol{x}, 0, \mathrm{HIGH}, k, \mathrm{LOW}, \boldsymbol{o})$
12:             end for
13:             if kernel size is odd then
14:                 $\boldsymbol{x} \leftarrow$ v_load(x_ptr)
15:                 $\boldsymbol{o} \leftarrow$ vc_mac$(\boldsymbol{x}, 0, \mathrm{LOW}, k, \mathrm{LOW}, \boldsymbol{o})$
16:             end if
17:             $\boldsymbol{o} \leftarrow$ v_shift$(\boldsymbol{o}, -$args.QuantParameter$)$ ▷ Post shift to input format
18:             $\boldsymbol{o} \leftarrow$ v_pack$(\boldsymbol{o})$                       ▷ Cast to 16-bit elements
19:             out_ptr $\leftarrow$ v_store$(28, \boldsymbol{o},$ out_ptr$)$; out_ptr $+=$ 28
20:         end for
21:     end for
22: end function
```

Average pooling

The average pooling layer implementation was made for two slightly different purposes. In the first approach, presented in Algorithm 6, the implementation supports only models where unit stride ($S = 1$) is applied. When unit stride is used, the operation is very close to the convolution layer in Algorithm 4. It does not perform downsampling, which is the typical function of a pooling layer in many CNNs. The layer performs averaging of the input vector within the receptive field, which is the same as convolution with identical and constant weight equal to $k = 1/F$, where F is the kernel size. Due to the 32-bit aligned memory access when loading elements to the registers, the input is processed in 28 element chunks.

The implementation of the average pooling with a unit stride in Algorithm 6 is very similar to convolution layer implementation in Algorithm 4 with two key differences:

- The accumulator vector $\boldsymbol{o}$ is initialized with zeros instead of bias parameter (line 7).
- Constant kernel parameter k (line 2) is used for all iterations instead of loading kernel parameters from memory. The fixed-point format for k is the same as the layer input, and the number of fractional bits is stored in `args.QuantParam`. If the layer input has large values, this might cause problems, and some other format for this parameter should be used.

The second implementation for average pooling in Algorithm 7 is for more conventional feature vector downsampling, where stride is equal to kernel size. In this case, the implementation supports kernel size and stride two, i.e. $F = S = 2$,

Algorithm 7 Vectorized average pooling layer, when $F = S = 2$.

```
 1: function ProcessAvgPoolLayer(args, input_ptr, output_ptr)
 2:     $k \leftarrow (1 \ll \texttt{args.QuantParameter})/\texttt{args.KernelSize}$
 3:     $\boldsymbol{cfg} \leftarrow$ v_move(0, 0x02468ACE)          ▷ Vector for permutation
 4:     $\boldsymbol{cfg} \leftarrow$ v_move(1, 0x13579BDF)          ▷ In 32-bit hexadecimal
 5:     for all input channels do
 6:         for all 32 element chunks of the input vector do
 7:             $\boldsymbol{o} \leftarrow$ v_fill(0)                  ▷ Reset accumulator vector
 8:             $\boldsymbol{x} \leftarrow$ v_load(input_ptr); input_ptr += 32
 9:             $\boldsymbol{o} \leftarrow$ vc_mac($\boldsymbol{x}$, 0, LOW, $k$, LOW, $\boldsymbol{o}$)
10:             $\boldsymbol{o} \leftarrow$ vc_mac($\boldsymbol{x}$, 0, HIGH, $k$, LOW, $\boldsymbol{o}$)
11:             $\boldsymbol{o} \leftarrow$ v_shift($\boldsymbol{o}$, −args.QuantParameter) ▷ Post shift to input format
12:             $\boldsymbol{o} \leftarrow$ v_pack($\boldsymbol{o}$)                ▷ Cast to 16-bit elements
13:             $\boldsymbol{o} \leftarrow$ v_permute($\boldsymbol{o}$, $\boldsymbol{cfg}$)    ▷ Move elements with even index to front
14:             v_store(16, $\boldsymbol{o}$, output_ptr); output_ptr += 16
15:         end for
16:     end for
17: end function
```

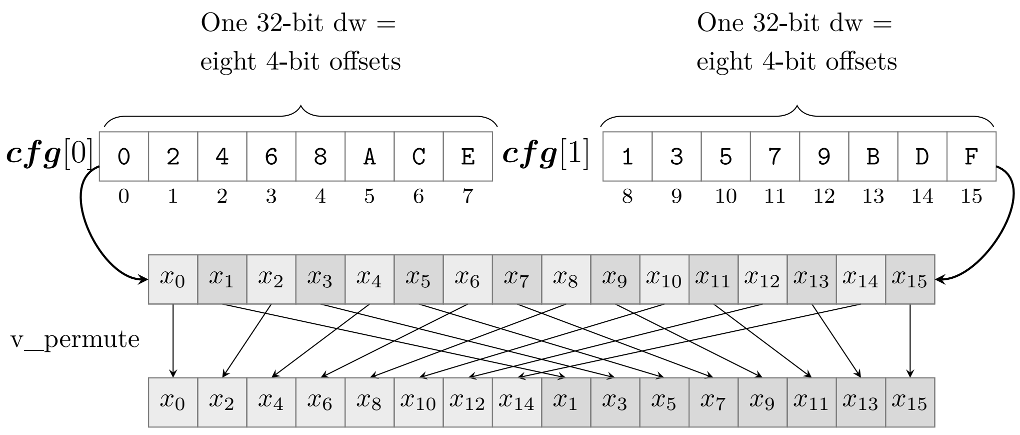

One 32-bit dw = eight 4-bit offsets
One 32-bit dw = eight 4-bit offsets
$\boldsymbol{cfg}[0]$
0 2 4 6 8 A C E
$\boldsymbol{cfg}[1]$
1 3 5 7 9 B D F
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
v_permute
x_0 x_2 x_4 x_6 x_8 x_{10} x_{12} x_{14} x_1 x_3 x_5 x_7 x_9 x_{11} x_{13} x_{15}

Figure 6.5: Illustration of the vector shuffling applied in Algorithms 7-8. Configuration vector $\boldsymbol{cfg}$ contains 16 4-bit indexes which define positions of words to be extracted from the input vector $\boldsymbol{x}$. The shuffling operation is executed identically in both VCUs, and thus only 16 elements are visualized.

which means that the output feature vector size is half the input feature vector size. It is the most common choice of parameters when average pooling is applied for downsampling purposes. However, the same procedure can be utilized with larger kernels with some modifications.

The algorithm operates in 32 input element chunks producing 16 output elements. Two sequential input elements are averaged by performing MAC-operation with constant k similarly as in Algorithm 6. Then every second element of the output is discarded using vector shuffling. The vector shuffling using v_permute-intrinsic is illustrated in Figure 6.5. The operation is controlled using configuration vector $\boldsymbol{cfg}$ from which only two first doublewords are needed. $\boldsymbol{cfg}$ contains 16 4-bit indexes, which define the positions of words to be extracted from the input vector. Initializing configuration vector is visible in lines 3-4 in Algorithm 7. The 32-bit values are given in hexadecimal because they seemingly illustrate the 4-bit indexes that are initialized.

Max pooling

Max pooling is the most common type of pooling layer applied in CNNs. The operation reports the maximum value in the receptive field region. In this work, vectorized max-pooling layer performing downsampling is implemented for equal kernel sizes and strides of 2 and 4. Currently, the pseudocode presented in Algorithm 8 supports only equal strides and kernel sizes. The algorithm operates with entire 32 element chunks and produces either 16 or 8 output elements depending on the kernel size (line 2). Because v_max-intrinsic does not support offsetting with LOW/HIGH-keyword, the second input vector must be obtained by vector shuffling. Otherwise, the algorithm is very similar to the corresponding average pooling layer in Algorithm 7.

Algorithm 8 Vectorized max pooling layer for $F = S = \{2, 4\}$.

```
1: function ProcessMaxPoolLayer(args, input_ptr, output_ptr)
2:     $k \leftarrow 32/$args.KernelSize                  ▷ Number of output elements
3:     $\boldsymbol{cfg1} \leftarrow$ v_move(0, 0x12345678)          ▷ Vector for permutation
4:     $\boldsymbol{cfg1} \leftarrow$ v_move(1, 0x9ABCDEF0)           ▷ In 32-bit hexadecimal
5:     $\boldsymbol{cfg2} \leftarrow$ v_move(0, 0x02468ACE)          ▷ Vector for permutation
6:     $\boldsymbol{cfg2} \leftarrow$ v_move(1, 0x13579BDF)
7:     for all input channels do
8:         for all 32 element chunks of the input vector do
9:             $\boldsymbol{x} \leftarrow$ v_load(input_ptr); input_ptr += 32
10:            for all 2 element chunks of the kernel do            ▷ 1-2 iterations
11:                $\boldsymbol{y} \leftarrow$ v_permute($\boldsymbol{x}, \boldsymbol{cfg1}$)            ▷ Rotate vector by one
12:                $\boldsymbol{z} \leftarrow$ v_max($\boldsymbol{x}, \boldsymbol{y}$)
13:                $\boldsymbol{x} \leftarrow$ v_permute($\boldsymbol{z}, \boldsymbol{cfg2}$) ▷ Move elements with even idx to front
14:            end for
15:            output_ptr $\leftarrow$ v_store($k, \boldsymbol{x}$, output_ptr); output_ptr += $k$
16:        end for
17:    end for
18: end function
```

6.3.3 Activation functions

Activation functions are integral parts of many CNNs that allow networks to learn phenomena with non-linear properties. As a part of this work, two different vectorized activation functions, Leaky Rectified Linear Unit (Leaky ReLU) and Sigmoid were implemented. Standard ReLU activation function is already implemented in [10] and the same procedure is applied here. The impact of vectorization of activation functions on the inference performance is discussed in Section 8.2.5.

Both activation functions are elementwise and operate in 32 element chunks to ensure optimal vectorization. Handling the input tail when the input vector length is not a multiple of 32 is omitted from this Subsection's pseudocodes. The tails are handled with the so-called overwriting technique, similar to the previous Algorithms. The working buffers for storing intermediate results of the inference are allocated to a size of a multiple of 32 words. Even if the input size is not multiple of 32, it is possible to read and write the complete 32 elements chunks without any overflows. Processing garbage values in the vector tail do not affect the result with elementwise functions. This approach avoids using potentially ineffective conditional jumps for tail processing.

Leaky ReLU

The vectorized implementation of the Leaky ReLU activation function is illustrated in Algorithm 9. The implementation relies on the vector predicate mechanism discussed in Section 5.2.2. The v_max-intrinsic with a vector of zeroes on line 6 outputs a vector predicate `pred`, where the bits are set corresponding to the positions of

Algorithm 9 Vectorized Leaky ReLU activation function.

```
1: function ProcessLeakyRelu(args, input_ptr, output_ptr)
2:     $c \leftarrow 1 \ll$ args.QuantParameter
3:     $\alpha \leftarrow$ args.Alpha
4:     for all 32 element chunks of the input do
5:         $\boldsymbol{x} \leftarrow$ v_load(input_ptr); input_ptr += 32
6:         pred $\leftarrow$ v_min($\boldsymbol{x}, \mathbf{0}$)        ▷ Minimum with vector of zeros
7:         $\boldsymbol{o} \leftarrow$ vc_mltply($\boldsymbol{x}$, 0, LOW, $c$, LOW)
8:         $\boldsymbol{o} \leftarrow$ vc_mltply($\boldsymbol{x}$, 0, LOW, $\alpha$, LOW, pred)
9:         $\boldsymbol{o} \leftarrow$ v_shift($\boldsymbol{o}$, −args.QuantParameter)
10:        $\boldsymbol{o} \leftarrow$ v_pack($\boldsymbol{o}$)
11:        v_store(32, $\boldsymbol{o}$, out_ptr); output_ptr += 32
12:    end for
13: end function
```

positive values. On line 8, this predicate vector is applied in vector multiplication, resulting in only positive elements being multiplied with α.

The α-coefficient, `args.Alpha` is the negative slope coefficient of the function. The parameter value is defined as a part of the model constructing procedure, and it is quantized to a 16-bit fixed-point number as a part of the preprocessing discussed in Section 6.2—the number of fractional bits stored in `args.QuantParam` is the same as the input activations have. If activations are large and α is small, this approach might lose the α-coefficient accuracy in the quantization, and it is rounded to zero. In that case, the function outputs the same as the standard ReLU. When activations are small enough, and the number of fractional bits is enough, no problems should be observed. With large activations, the problem can be solved using different fixed-point formats for α-coeffients. Output has the same format as the input, which requires that the resulting vector is normalized with bit-shift and elements are packed back to 16-bit values. Because this normalization is done for all elements in the output vector $\boldsymbol{o}$, the negative input values are also bit-shifted by multiplying them on line 7 with constant c.

Sigmoid

Algorithm 10 shows the pseudocode of vectorized version of an approximation of Sigmoid activation function. Real-time implementation of exponential functions such as Sigmoid with fixed-point numbers can be tedious, and can easily result inefficient hardware implementations. Approximations enable applying only integer arithmetic, can provide significant preformance improvements. This might decrease the accuracy of the function, which is not crucial in most cases when NN actication functions are conserned. In this work, Sigmoid function (Eq. (3.5)) is approximated using

Algorithm 10 Vectorized Sigmoid activation function.

```
function PROCESSSIGMOID(args, input_ptr, output_ptr)
    $\boldsymbol{a} \leftarrow$ v_fill($-2 \ll$ args.QuantParameter)
    $\boldsymbol{b} \leftarrow$ v_fill($2 \ll$ args.QuantParameter)
    $c \leftarrow 1 \ll$ args.QuantParameter
    for all 32 element chunks of the input do
        $\boldsymbol{x} \leftarrow$ v_load(input_ptr); input_ptr += 32          ▷ Load 32 values
        $\boldsymbol{o} \leftarrow$ v_shift($\boldsymbol{x}, -1$)                ▷ Divide by 2
        $\boldsymbol{o} \leftarrow$ vv_add($\boldsymbol{x}, c$)                  ▷ Add c
        $\boldsymbol{o} \leftarrow$ v_shift($\boldsymbol{x}, -1$)                ▷ Divide by 2 again
        pred $\leftarrow$ v_min($\boldsymbol{x}, \boldsymbol{a}$)
        $\boldsymbol{o} \leftarrow$ v_fill($0$, pred)      ▷ Replace values below threshold with 0
        pred $\leftarrow$ v_max($\boldsymbol{x}, \boldsymbol{b}$)
        $\boldsymbol{o} \leftarrow$ v_fill($c$, pred)      ▷ Replace values above threshold with c
        v_store($32, \boldsymbol{o}$, out_ptr); output_ptr += 32      ▷ Store 32 values
    end for
end function
```

piecewise function

$$\varphi_{\text{sigmoid}}(y) \approx \begin{cases} 0 \quad \text{if } y < -2 \\ \frac{y}{4} + \frac{1}{2} \quad \text{if } -2 \leq y \leq 2 \\ 1 \quad \text{if } y > 2. \end{cases} \tag{6.4}$$

, which is in fact first-order Taylor expansion of Sigmoid-function in the interval $[-2, 2]$. With this approximation, all multiplication can be implemented efficiently with bit-shifts. Same approximation is applied also in the scalar implementation of the function [70]. Replacing values below and above the thresholds is done using vector predicate mechanism discussed in Section 5.2.2. Initializing these vector predicates is visible on lines 10 and 12 in Algorithm 10.

Chapter 7

Experimental setup

In this Chapter, the experimental setup used in this work is explained. Implementing CNNs to target DSP and evaluating the obtained solution requires multiple tools and software. The first section describes these tools and the workflow. Because this work aimed to provide a scalable implementation that could effectively accelerate different CNNs, five models with different characteristics were applied in the testing stage. These models are described in Section 7.2. Because the context of this work is real-time physical layer processing using hardware with limited resources, the models' computational complexity and memory consumption are important aspects that should be considered as part of the implementation evaluation process. Thus, some metrics are derived for estimating the computational complexity and memory consumption of different NNs.

In this work, the performance of the real-time inferences is measured in processor cycles recorded with a performance-accurate simulator. The detail of the performance-related metrics and measurements are explained in Section 7.4. In addition, effective utilization of the target DSP properties requires quantization of the parameters and the input data. Some information loss is unavoidable because the quantized fixed-point numbers use representations with a smaller number of bits. Thus, as part of the performance evaluation, quantized inference accelerated in the target hardware is compared with the reference floating-point inference to evaluate the quantized model's correctness. The metrics and the procedure to evaluate the correctness of the inferences is described in Section 7.4.

7.1 Workflow

The DSP code is run in a software simulator instead of an actual SoC. Measuring cycle-accurate results in the SoC would require capturing program traces with specialized equipment such as Lauterbach in-circuit program trace tool [9]. Because DSP manufacturer provides performance accurate software simulator, that has been verified in previous work [9], it is an obvious choice to perform experiments in this simulated environment. The workflow is designed to ensure that the simulated results are accurate and that the procedure can be easily ported to actual hardware. The following workflow is applied to evaluate the performance of each NN real-time inference:

1. **Creating test model**
   Create a floating-point model using PyTorch DL framework [39], and export the model to ONNX-format [41]. The model would be trained and tested in the standard deep learning workflow applying some DL framework such as PyTorch before exporting it to ONNX-format. However, model training is typically one of the most time-consuming parts of model development and requires a large amount of training data. Because model training is not necessary for the performance evaluations executed in this work, models were not trained, and no real-life data was used. Instead, all network parameters and the input data was generated randomly using techniques described in Section 7.3. Thus, the model outputs in the experiment part are not related to any real-life phenomena.

2. **Preprocessing**
   Import the ONNX-model and preprocess it to a suitable format for DSP. The number of guard bits was determined by running Algorithm 1 and the number of fractional bits by running Algorithm 2. For each model, 1000 representative input data samples were generated for Algorithm 2. Because the data was generated from the normal distribution in these experiments, much fewer samples would have been enough. However, when real-life data is used, the size of the representative data set should be large enough. Then network parameters are quantized based on previously determined parameters. In the final step of preprocessing, the DSP suitable network configuration is created as described in Section 6.2. The configuration and the quantized parameters are stored in separate files, mimicking storing them in the shared memory of the actual SoC.

3. **Real-time inference**
   Performance critical run-time implementation is executed using a software simulator, a part of the CEVA Toolbox integrated development environment (IDE). It offers a cycle-accurate simulation environment for the target CEVA-XC4500 DSP, and thus the experiments can be executed without accessing the actual hardware. The simulator cannot simulate the DSP's external memory; thus, it provides cycle-accurate results only when the code size is less than the internal instruction memory size, and all data fits internal data memory. Each experiment starts by loading model configuration, model parameters and the floating-point input data from files to the internal memory of the DSP. Loading from separate files mimics data transfer via DMA on the actual SoC. After that, the inference function described in Algorithm 3 is called, and the cycle counts are recorded. When all steps of the inference function have been processed, the model's output is stored in a file for further analysis.

4. **Evaluation**
   The output of DSP run-time inference is compared with the floating-point reference model executed using ONNX run-time framework [71]. The correctness of the DSP inference is evaluated based on the procedure and the metrics described in Section 7.4.

7.2 Models

A total of five different 1D CNN models with a varying number of parameters and layers were used to evaluate the hardware implementations. Two of the models are CNNs designed for physical layer processing in 5G networks [14, 5]. These models are examples of models that the inference framework for the target DSP could be applied in practice. Three other models are not related to the physical layer domain at all [52, 72]. They are literature examples of possible CNN architectures for performance-critical applications. These models also give more variation for the properties of the test models to provide more understanding of the performance of the run-time implementations developed in this work. Because 1D CNNs are still somewhat novel ideas in deep learning, the literature did not yet provide many well-documented examples that could have been applied in the implementation testing in this work. Thus, also some models not related to physical layer processing were used.

All test models contain convolution layers and pooling layers that perform feature extraction from the data. Some models also contain fully-connected layers that perform classification or prediction based on the extracted features. Because fully-connected layer implementation is already extensively studied [10], only feature extraction parts are modelled, including convolution layers, pooling layers and needed activation functions.

The summary of the models is presented in Table 7.2. Models are named from Model A to Model E, and these names are used in the following Sections when test models are referred to. Because the purpose of this work is to study hardware-specific CNN implementations, it is essential to estimate the computational complexity and the memory consumption of the models. The computational complexity of the models is estimated using the number of multiply-accumulate (MAC) operations performed to obtain the layers' outputs. It is reasonable because convolution layers are computationally the most intensive layers in CNNs and apply mostly MAC operations. Thus, the number of MAC operations is computed only for these layers without activation functions. The number of operations is computed based on portable-reference computation without any DSP specific optimizations that might reduce the number of operations. Estimating the complexity of pooling layers and activation functions is not that straightforward as they might not utilize MAC operations and are thus excluded. Based on the Equation (4.7), the number of MAC operations performed in one convolution layer for one input sample is

$$N_i^{(MAC)} = C_i^{(in)} C_i^{(out)} L_i^{(out)} F_i, \tag{7.1}$$

where i denotes the layer index.

The memory requirements of the models are also estimated. The instruction memory consumption depends heavily on the chosen compiler optimization level and cannot be explicitly estimated. Thus, only data memory consumption can be reasonably computed. The DSP data memory is applied during the real-time inference to store the network parameters, intermediate results of the layers, and the model's input. The minimum memory requirement for network inference can be estimated by computing how much memory is needed to store all this information.

Table 7.1: Number of learnable parameters and input elements of a CNN.

Parameter	Type	Formula	Condition
Input data	`32-bit float`, `4 bytes`	$N^{(in)} = NC_1^{(in)} L_1^{(in)}$	only first layer.
Kernel weights	`16-bit int`, `2 bytes`	$N_i^{(F)} = C_i^{(in)} C_i^{(out)} F_i$	**if** convolution layer.
		$N_i^{(F)} = 0$	**else**.
Bias parameters	`32-bit int`, `4 bytes`	$N_i^{(B)} = C_i^{(out)}$	**if** convolution layer.
		$N_i^{(B)} = 0$	**else**.

This requirement depends on the number of elements to be stored and the type of the elements. In this work, the intermediate results of the layers are stored in two fixed-size buffers with type `16-bit int` occupying `2 bytes`. The needed buffer sizes for each layer are determined in the preprocessing stage using the following rule

$$N_i^{(buf)} = \max\left(32\left\lceil\frac{NC_i^{(out)}L_i^{(out)}}{32}\right\rceil, 32\left\lceil\frac{NC_i^{(in)}L_i^{(in)}}{32}\right\rceil\right), \tag{7.2}$$

where i is the layer index, and N is the batch size. Buffers are allocated to a size of a multiple of 32 words to avoid buffer overflow in algorithms where 32 words are processed simultaneously without specific tail handling. Working buffers are reused between layers by swapping memory addresses pointing to input and output buffers; thus, only two are needed to process the entire network. Computing the number of input elements and the number of learnable parameters in each layer of a CNN is summarized in Table 7.1. The network parameters are always the same for all input samples within and between batches; thus, the parameters are independent of the batch size N.

By applying the above ruling, the total number of learnable parameters in the network is

$$N^{(P)} = \sum_i \left(N_i^{(F)} + N_i^{(B)}\right) \tag{7.3}$$

and the corresponding total memory requirement $M^{(tot)}$ for processing batch size of N input samples expressed in `bytes` is

$$M^{(tot)} = 4\ \texttt{bytes} \times N^{(in)} + 2\ \texttt{bytes} \times \max_i\left(2N_i^{(buf)}\right) \tag{7.4}$$

$$+\sum_i \left(2\ \texttt{bytes} \times N_i^{(F)} + 4\ \texttt{bytes} \times N_i^{(B)}\right) \tag{7.5}$$

The total memory consumptions $M^{(tot)}$ in kilobytes (KB), total numbers of network parameters $N^{(P)}$ and total computational complexities $N^{(MAC)}$ of the test models are reported in Table 7.2. The total computational complexity is computed by summing up all MAC operations in convolution layers (Eq. (7.1)).

Table 7.2: Summary of the CNNs used in the DSP implementation evaluation.

Name	Application	Layers n	Params. $N^{(P)}$	Memory, $M^{(tot)}$ (KB)	Operations, $N^{(MAC)}$
Model A	Near-infrared transmittance spectroscopy signal regression [72]	6	106	2,45	5 125
Model B	Near-infrared reflectance spectroscopy signal regression [72]	4	146	16,55	62 300
Model C	ECG signal classification for arrhythmia detection [52]	8	1 234	13,18	186 274
Model D	Preamble detection and TOA estimation in 5G networks [5]	6	722	65,43	289 792
Model E	Autoencoder for channel estimation in 5G networks [14]	9	10 302	44,53	1 915 200

Model A

This model is developed for predicting the fat content in meat samples by analyzing the measured near-infrared transmittance spectra. The architecture details are presented in Table 7.3. Each input data sample contains 100 spectral observations, and the model consists of three small convolution layers combined with average pooling layers for the feature extraction. After the last pooling layer, the channels are concatenated to form a feature vector considered as a new representation of the input sample. The final layer, which is not modelled in this work, is a fully-connected layer acting as a standard linear regressor during model training. In [72], two additional regression methods are proposed as well. Model A is an example of a compact CNN with a relatively small computational load and memory consumption.

Table 7.3: **Model A**. Architecture of a CNN for predicting the fat content in meat samples by near-infrared transmittance spectra regression [72]. The shape of the model input is $(N, C_1^{(in)}, L_1^{(in)})$=(N,1,100), where N refer to the batch size of one inference iteration.

Idx, i	Layer type	Kernel (F_i, S_i, P_i)	Output shape $(N, C_i^{(out)}, L_i^{(out)})$	Activation function	Params. $N_i^{(P)}$	Memory, $M_i^{(tot)}$ (KB)	Oper. $N_i^{(MAC)}$
1	Conv.	(7,1,0)	(N,5,94)	Sigmoid	40	1,92	3 290
2	Ave pool.	(2,2,0)	(N,5,47)	-	0	1,84	-
3	Conv.	(7,1,0)	(N,1,41)	Sigmoid	36	0,99	1 435
4	Ave pool.	(2,2,0)	(N,1,20)	-	0	0,16	-
5	Conv.	(5,1,0)	(N,5,16)	Sigmoid	30	0,38	400
6	Ave pool.	(2,2,0)	(N,5,8)	-	0	0,31	-

Model B

Model B is very similar to Model A and is designed for predicting saccharose content in orange juice samples measured with near-infrared reflectance spectroscopy. Each measured spectra contains 700 observations. Model B applies two convolution layers

Table 7.4: **Model B**. Architecture of a CNN for determining the concentration of saccharose in orange juice samples by near-infrared reflectance spectra regression [72]. The shape of the model input is $(N, C_1^{(in)}, L_1^{(in)})$=(N,1,700), where N refer to the batch size of one inference iteration.

Idx, i	Layer type	Kernel (F_i, S_i, P_i)	Output shape $(N, C_i^{(out)}, L_i^{(out)})$	Activation function	Params. $N_i^{(P)}$	Memory, $M_i^{(tot)}$ (KB)	Oper. $N_i^{(MAC)}$
1	Conv.	(9,1,0)	(N,5,692)	Sigmoid	50	13,62	31 140
2	Ave pool.	(2,2,0)	(N,5,346)	-	0	13,51	-
3	Conv.	(19,1,0)	(N,1,328)	Sigmoid	96	6,95	31 160
4	Ave pool.	(2,2,0)	(N,1,164)	-	0	1,28	-

combined with average pooling layers to extract features. Model B is slightly more complex and requires more memory than Model A due to the much larger input spectra size [72].

Model C

This CNN is designed for automated detection of arrhythmias, and the architecture is presented in detail in Table 7.5. The model inputs an electrocardiogram (ECG) signal of 500 elements and classifies the signal into four classes representing different types of arrhythmia. Model A represents a typical classification model with four convolution layers combined with max-pooling layers for feature extraction, followed by three fully-connected classifying layers. Fully-connected layers performing classification are not modelled in this work. Model C has achieved an accuracy of 92,50 % in the experiments reported in [52].

Table 7.5: **Model C**. Architecture of CNN for automated detection of arrhytmias based on ECG Signals [52]. The shape of the model input is $(N, C_1^{(in)}, L_1^{(in)})$=(N,1,500), where N refer to the batch size of one inference iteration.

Idx, i	Layer type	Kernel (F_i, S_i, P_i)	Output shape $(N, C_i^{(out)}, L_i^{(out)})$	Activation function	Params. $N_i^{(P)}$	Memory, $M_i^{(tot)}$ (KB)	Oper. $N_i^{(MAC)}$
1	Conv.	(27,1,0)	(N, 3,474)	Leaky ReLU	84	5,72	38 394
2	Max pool.	(2,2,0)	(N, 3,237)	-	0	5,55	-
3	Conv.	(14,1,0)	(N,10,224)	Leaky ReLU	430	9.61	94 080
4	Max pool.	(2,2,0)	(N,10,112)	-	0	8,75	-
5	Conv.	(3,1,0)	(N,10,110)	Leaky ReLU	310	5	33 000
6	Max pool.	(2,2,0)	(N,10,55)	-	0	4,30	-
7	Conv.	(4,1,0)	(N,10,52)	Leaky ReLU	410	2,97	20 800
8	Max pool.	(2,2,0)	(N,10,26)	-	0	2,03	-

Model D

Model D is a CNN designed for random access channel (RACH) preamble detection and time of arrival (TOA) estimation in 5G networks (Table 7.6). TOA estimation is typically applied for synchronization purposes in initial access and localization of the transmitter. Traditional TOA estimation has been executed using correlation-based methods. The proposed NN solution improves estimation results, especially in conditions where nonlinearities occur in the RF chain due to multipath propagation and noise. The model input consists of the received signal, in this case, a complex-valued vector with 4095 values. The network architecture uses three convolution layers combined with Max pooling layers for feature extraction and dimension reduction. After that, in [5] feature vector is fed to two fully-connected layers, which are not included in this work. Model D provided significant improvements in prediction accuracy and computational complexity compared to the existing methods [5].

Table 7.6: **Model D**. Architecture of Convolution Neural Network for preamble detection and Time of Arrival (TOA) estimation in initial access and localization of 5G network [5]. The shape of the model input is $(N, C_1^{(in)}, L_1^{(in)})$=(N,2,4095), where N refer to the batch size of one inference iteration.

Idx, i	Layer type	Kernel (F_i, S_i, P_i)	Output shape $(N, C_i^{(out)}, L_i^{(out)})$	Activation function	Params. $N_i^{(P)}$	Memory, $M_i^{(tot)}$ (KB)	Oper. $N_i^{(MAC)}$
1	Conv.	(16,2,8)	(N,4,2048)	ReLU	132	32,27	262 144
2	Max pool.	(4,4,0)	(N,4,512)	-	0	32,00	-
3	Conv.	(8,4,2)	(N,6,128)	ReLU	198	8,40	24 576
4	Max pool.	(4,4,0)	(N,6,32)	-	0	3,00	-
5	Conv.	(8,4,2)	(N,8,8)	ReLU	392	1,53	3 072
6	Max pool.	(4,4,0)	(N,8,2)	-	0	0,25	-

Model E

Model E is an autoencoder designed to perform pilot-based channel estimation in a 5G cellular network. In this work, only the encoder part is modelled. The model architecture consists of five convolution layers and four average pooling layers. The best results were obtained in the proposed solution when 192 complex Demodulation Reference Signals (DMRS) were used as an input sample. The solution was compared to the current industry standard, Least Squares (LS) channel estimation and significant improvements were achieved.

Table 7.7: **Model E**. Architecture of CNN Autoencoder in pilot-based Channel Estimation for 5G Physical Uplink Shared Channel [14]. The shape of the model input is $(N, C_1^{(in)}, L_1^{(in)})$=(N,2,192), where N refer to the batch size of one inference iteration.

Idx, i	Layer type	Kernel (F_i, S_i, P_i)	Output shape $(N, C_i^{(out)}, L_i^{(out)})$	Activation function	Params. $N_i^{(P)}$	Memory, $M_i^{(tot)}$ (KB)	Oper. $N_i^{(MAC)}$
1	Conv.	(7,1,3)	(N,192,10)	ReLU	150	7,81	26 880
2	Ave pool.	(3,1,0)	(N,190,10)	-	0	7,50	-
3	Conv.	(7,1,3)	(N,190,20)	ReLU	1420	17,66	266 000
4	Ave pool.	(3,1,0)	(N,188,20)	-	0	14,84	-
5	Conv.	(7,1,3)	(N,188,30)	ReLU	4230	30,35	789 600
6	Ave pool.	(3,1,0)	(N,186,30)	-	0	22,03	-
7	Conv.	(7,1,3)	(N,186,20)	ReLU	4220	30,08	781 200
8	Ave pool.	(3,1,0)	(N,184,20)	-	0	14,53	-
9	Conv.	(7,1,3)	(N,184,2)	ReLU	282	14,93	51 520

7.3 Data

Network training is typically one of the most time-consuming phases in constructing a neural network for practical application. It requires specialized hardware such as GPUs and plenty of relevant training data. In the context of this work, processor cycle counts of fixed-point NN inferences were considered attractive. It was also monitored that the accuracy of the fixed-point inference accelerated in DSP was not substantially worse than the floating-point reference. Measuring this information does not necessarily require accelerating trained networks. Thus, models are not trained, and real-life data is not used because it is not necessary for the scope of this thesis. Random values generated from the standard normal distribution were used as input data for the models, i.e.

$$\boldsymbol{x} \sim \mathcal{N}(\mu, \sigma^2), \quad \text{where } \mu = 0 \text{ and } \sigma = 1. \tag{7.6}$$

A typical workflow with NNs includes input data scaling as a part of the preprocessing step. It ensures that training can be executed more effectively and gradient-based methods converge faster, even in cases where the range of values in the input data is wide [2]. This scaling can be carried out using many approaches, such as min-max normalization or standardization. Values generated from standard normal distribution have comparable statistical properties as any data set converted to a standard score by standardization.

Because models in this work are not trained, also network parameters must be generated for experiments. To ensure that the network activation values remain finite, well known neural network weight initializing strategies were employed. In layers applying ReLU activation function, Kaiming initialization stategy [73] is used, i.e. the kernel weight values are generated from uniform distribution:

$$\boldsymbol{\omega} \sim U(-a, a), \quad \text{where } a = \sqrt{\frac{2}{(1+c^2)N_i^{(in)}}}, \tag{7.7}$$

c is the negative slope of the ReLU activation, in stardard ReLU $c = 0$. Layers applying other activation than ReLU, Xavier initialization strategy [74] is employed

$$\boldsymbol{\omega} \sim U(-a, a), \quad \text{where } a = \sqrt{\frac{6}{N_i^{(in)} + N_i^{(out)}}}. \tag{7.8}$$

$N_i^{(in)}$ and N_i^{out} are number of incoming and outgoing network connections of the ith layer, respectively. For convolution layers they equal

$$N_i^{(in)} = F_i C_i^{(in)} \quad \text{and} \quad N_i^{(out)} = F_i C_i^{(out)}. \tag{7.9}$$

Bias parameters are initialized using identical distributions as the weights.

7.4 Metrics

The performance of the real-time inferences is measured in DSP processor cycles. Cycle counts are recorded in a performance-accurate simulator. Cycle-count recording is started when all required data and the program code are available in the internal memory of DSP, and the inference function described in Algorithm 3 is called. Because the cycle counts in this work are recorded in the simulator, they do not necessarily take into account all effects in the hardware that affect the total cycle counts. However, the simulated cycle counts provide reliable lower-bound estimates for the inference latency times in the SoC. They can be applied in evaluating the feasibility of the solutions in practical applications.

For each individual cycle-count result, the inference was executed 100 times in the simulator using new data set for every iteration. Before executing simulations, it was manually verified that each version of the DSP code gives an identical output when the same data set is inserted. The reported cycle counts are arithmetic means from those 100 iterations. Because absolute cycle counts and latency times are confidential information, they are not reported in Section 8.2 that discusses the performance of the implementation. Instead differences in percentages compared to the starting point of the optimization, i.e. the fully-scalar portable reference, are reported.

Because all models are converted to fixed-point format before running the inferences in DSP, some information loss inevitably occurs. The obtained outputs are compared with the floating-point reference outputs to verify that the quantized inference output is still valid. Three key statistics are recorded based on the feature vectors extracted by the models for each iteration, which are reported in Section 8.1. Before determining these statistics, the fixed-point output of the DSP inference is converted back to floating-point format. **Maximum Absolute Error (MaxAE)** and **Mean Squared Error (MSE)** are used to distinguish some possibly incorrect elements in the DSP inference output:

$$\text{MaxAE} = \max_i |e_i| = \max_i |y_i - \hat{y}_i| \tag{7.10}$$

$$\text{MSE} = \frac{1}{n}\sum_i^n e_i^2 = \frac{1}{n}\sum_i^n (y_i - \hat{y}_i)^2 \tag{7.11}$$

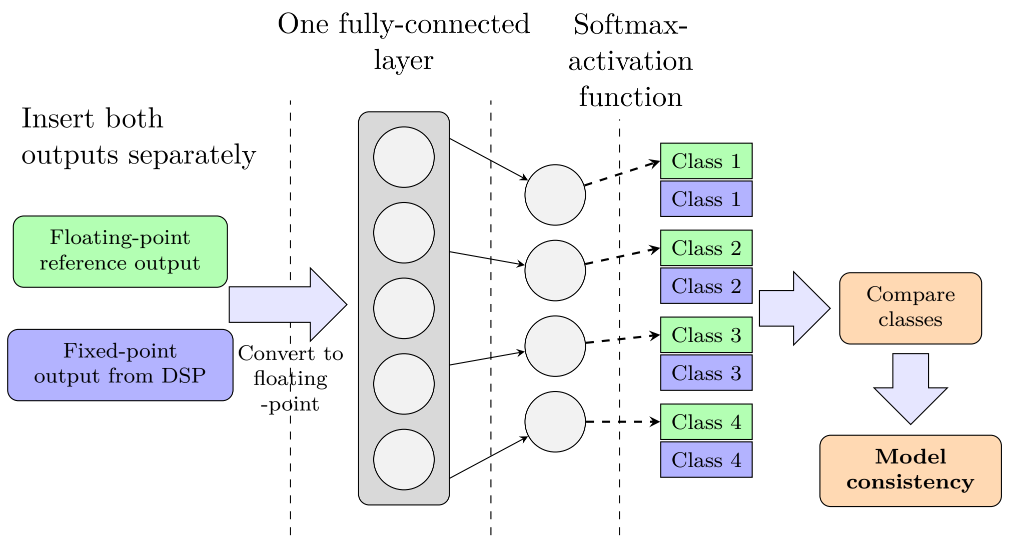

One fully-connected layer
Softmax-activation function
Insert both outputs separately
Floating-point reference output
Fixed-point output from DSP
Convert to floating -point
Class 1
Class 1
Class 2
Class 2
Class 3
Class 3
Class 4
Class 4
Compare classes
Model consistency

Figure 7.1: An overview of the procedure to compare the fixed-point real-time inference output and floating-point reference output using a simple NN classifier.

,where i covers all indices of the output and $\boldsymbol{y}$ is the output of the floating-point reference model, and $\hat{\boldsymbol{y}}$ is the output of the real-time DSP inference. It is essential to eliminate possible programming faults in the DSP implementations and detect the integer overflow caused by suboptimal quantization parameter choices. The purpose was not to avoid any overflow happening because, in many cases, they might not have that severe effect on the overall correctness of the model. However, it was necessary to distinguish if the model quantization caused some apparent deviation from the floating-point reference that could be avoided by adjusting the quantization parameters.

The third statistic, **model consistency**, measures the usefulness of the feature vectors the models provide as an output. The feature vectors are fed to a fully-connected layer with the Softmax activation function, which performs a simple classification to four classes. The procedure is illustrated in Figure 7.1. The layer weights are random weights generated using distributions described in Section 7.3. The classification is done with floating-point numbers using the PyTorch DL toolbox. The reported model consistencies are the percentages of iterations, where the classification result performed using the DSP inference feature vector, and floating-point reference feature vector agree.

Chapter 8

Evaluation

The previous two chapters thoroughly explain the procedure to build and evaluate the DSP implementation. The metrics for measuring the correctness of the results as well as the execution performance were introduced in Section 7.4. The performance evaluations for the five test models are reported in this Chapter using these metrics. The content of this Chapter can be divided into three different parts. In the first part, the correctness of the DSP inference results is evaluated and discussed. In the second part, the computational performance is evaluated based on the cycle counts measured in the software simulator. Lastly, a summary of the evaluation is provided.

8.1 Quantization and model correctness

As a part of the performance evaluation process, the correctness and usefulness of the DSP inference output are evaluated. Before real-time inference implementations were executed in the target DSP, they were quantized to fixed-point format using the procedure explained in Section 6.2. The obtained amounts of guard bits, as well as Q-formats for layer inputs/outputs and network parameters for all test models, are reported in Table 8.1. The Table shows parameters for only convolution layers, as the framework is designed so that the quantization format can only change in convolution layers. Representative data set was generated from the distribution discussed in Section 7.3, and the size of the data set was 1000 samples for each model. When measuring computational performance, the reported Q-formats were used in all DSP inference executions.

The Q-formats obtained with the Algorithm 2 are pretty consistent. Input data has the same format Q6.10 in all models because identically generated input data was used. Kernel parameters apply the same number of fractional bits in almost all models and layers. It suggests that layerwise determination of quantization parameters is unnecessary in this case. The same format for inputs, activations and parameters could have been used throughout the network. In this case Q6.10 for input data and activations, Q2.14 for kernel parameters and Q8.24 for bias parameters. However, layerwise quantization formats might be a good idea when real-life data and trained parameters are used. Especially when fewer bit numbers are applied, layerwise quantization is probably necessary. The models should be

Table 8.1: Quantization parameters for test models A-E obtained with Algorithms 1 and 2 with representation data set of size 1000 samples. Only convolution layers are included, other layers does not affect quantization format.

Model	Layer, i	Input	Kernel	Bias	Output	Post shift
Model A Guard bits: 4	1	Q6.10	Q1.15	Q7.25	Q5.11	14
	3	Q5.11	Q1.15	Q6.26	Q4.12	14
	5	Q3.13	Q1.15	Q5.27	Q3.13	14
Model B Guard bits: 3	1	Q6.10	Q2.14	Q8.24	Q5.11	13
	3	Q5.11	Q1.15	Q6.26	Q3.13	13
Model C Guard bits: 3	1	Q6.10	Q1.15	Q7.25	Q6.10	15
	3	Q6.10	Q1.15	Q7.25	Q6.10	15
	5	Q6.10	Q1.15	Q7.25	Q5.11	14
	7	Q5.11	Q1.15	Q6.26	Q4.12	14
Model D Guard bits: 3	1	Q6.10	Q1.15	Q7.25	Q6.10	15
	3	Q6.10	Q1.15	Q7.25	Q6.10	15
	5	Q6.10	Q1.15	Q7.25	Q5.11	14
Model E Guard bits: 4	1	Q6.10	Q2.14	Q8.24	Q6.10	14
	3	Q6.10	Q1.15	Q7.25	Q5.11	14
	5	Q5.11	Q1.15	Q6.26	Q4.12	14
	7	Q4.12	Q1.15	Q5.27	Q4.12	15
	9	Q4.12	Q1.15	Q5.27	Q3.13	14

tested with trained parameters and actual input data to increase the reliability of the quantization.

The number of guard bits is determined with Algorithm 1, and it turns out that a pretty moderate number of guard bits, 3-4 bits, are used with all models. When 16-bit fixed-point data and parameters are randomly generated from known distributions, it seems that more guard bits are not needed. The quantization procedure is probably robust enough for many deep learning applications. It should be noted that when the quantization strategy of this implementation is applied, data outliers have the same effect as the guard bits. They increase the number of integer bits in the input and activation formats and thus decrease the fixed-point model resolution. The robustness against outliers can be increased by modifying Algorithm 2 to use some percentile containing most of the values instead of the largest absolute values. Thus the most extreme outliers could be left outside when determining the amounts of integer and fractional bits.

Table 8.2 reports the Model consistencies, Maximum Absolute Error (MaxAE) and Mean Squared Error (MSE) for test models A-E. The measured values are obtained from all iterations executed to measure the computational performance of different implementation versions. The Table includes minimum, maximum and average values for MaxAE and MSE. From MaxAEs and MSEs can be seen that

Table 8.2: Model consistencies, Maximum Absolute Error (**MaxAE**) and Mean Squared Error (**MSE**).

Model	Number of iterations	Model consistency		MaxAE	MSE
Model A	7 500	99.48 %	Min.	$2.93 \cdot 10^{-3}$	$1.26 \cdot 10^{-6}$
			Avg.	$1.28 \cdot 10^{-2}$	$3.01 \cdot 10^{-5}$
			Max.	$3.58 \cdot 10^{-2}$	$2.18 \cdot 10^{-4}$
Model B	7 500	99.69 %	Min.	$6.40 \cdot 10^{-3}$	$5.06 \cdot 10^{-6}$
			Avg.	$1.89 \cdot 10^{-2}$	$1.31 \cdot 10^{-4}$
			Max.	$6.61 \cdot 10^{-2}$	$1.22 \cdot 10^{-3}$
Model C	7 500	99.96 %	Min.	$8.88 \cdot 10^{-4}$	$8.75 \cdot 10^{-8}$
			Avg.	$9.70 \cdot 10^{-3}$	$4.16 \cdot 10^{-6}$
			Max.	$1.55 \cdot 10^{-1}$	$2.01 \cdot 10^{-4}$
Model D	4 300	100 %	Min.	$2.77 \cdot 10^{-4}$	$9.00 \cdot 10^{-9}$
			Avg.	$1.34 \cdot 10^{-3}$	$4.98 \cdot 10^{-7}$
			Max.	$3.57 \cdot 10^{-3}$	$2.17 \cdot 10^{-6}$
Model E	2 700	99.81 %	Min.	$1.12 \cdot 10^{-3}$	$2.08 \cdot 10^{-7}$
			Avg.	$1.32 \cdot 10^{-2}$	$3.67 \cdot 10^{-6}$
			Max.	$7.39 \cdot 10^{-2}$	$3.30 \cdot 10^{-5}$

the accuracy of the fixed-point model is quite good. MSEs indicate that even the absolute values of the outputs are accurate, which is not even that necessary when CNNs are used as feature extractors. On average highest error rates in terms of MSE are reported with models A and B. Those models employ only the Sigmoid activation function, which is heavily approximated in the DSP implementations as discussed in Section 6.3.3. Approximation causes a further decrease in accuracy in addition to quantization. However, the observed values indicate that using approximation is still a suitable option if Sigmoid activation is necessary to apply.

The model correctness was applied to evaluate the quantized models' usefulness as feature extractors. Overall, the results reveal that the DSP implementation can be used without significantly decreasing classification consistency with all five models. As seen from model consistencies in Table 8.2, the implementation can reproduce the predictions of the floating-point model generally very well. Model A has the lowest consistency of 99.48 %, still indicating excellent agreement with the reference. With Model D the DSP inference even agrees with floating-point inference perfectly.

With good network design combined with parameter regularization and data normalization, even much fewer bits than 16-bits would be enough. For example, an 8-bit fixed point option could be an option. It would have an advantageous effect on computational performance and significantly decrease memory consumption and required memory bandwidths. However, the target hardware in this work does not support 8-bit numbers that well. In addition, the experiments simulate only the performance of the DSP inference from the point where all data is available in the DSP internal memory, so it is not suitable to study, for example, limited memory bandwidth related issues.

8.2 Performance optimization

This Section characterizes the procedure of how the processor cycle counts of the DSP inference is incrementally optimized. The starting of the process was a portable, fully-scalar version of the inference algorithm, where no DSP intrinsics were used. Different inference algorithm versions are illustrated in Table 8.3. A part of the algorithm is vectorized and optimized to decrease cycle counts in each version. The effects of vectorization with each version are evaluated by measuring processor cycles consumed by the NN inference with all five models.

This section reports differences in cycle count between versions and the percentages of cycles consumed in different parts of the inference algorithm.

Table 8.3: Summary of different DSP implementation versions.

Version (`Label`)	Description
Version 1 (`v1`)	Fully-scalar, portable reference implementation. Pure C++ version applying only scalar operations. No DSP features used, thus in principle can be ported to any device.
Version 2 (`v2`)	Real-time input quantization from floating-point to fixed-point representation is vectorized applying implementation described in [10]. All other inference parts still utilize only scalar operations.
Version 3 (`v3`)	Convolution layers are vectorized using Algorithm 5 for Model D and Algorithm 4 for rest of models. Pooling layers and activation functions are still evaluated using only hardware-independent scalar operations.
Version 4 (`v4`)	Vectorized pooling layers are added by applying Algorithm 6 or 7 for models with average pooling layers, and 8 for models with max pooling layers. Activation functions are evaluated using only scalar operations.
Version 5 (`v5`)	Fully vectorized and optimized version. All suitable parts of the inferences are vectorized. Leaky ReLU and Sigmoid activation functions apply SIMD by utilizing Algorithms 9-10, and normal ReLU follows approach described in [10].

8.2.1 Scalar reference

The portable, fully-scalar implementation (Version 1) was profiled to determine the number of cycles spent on different inference parts as a starting point for cycle count optimization. Figure 8.1 illustrates the percentages of cycles spent processing different parts of the inference in Version 1. The control overhead cycles are computed

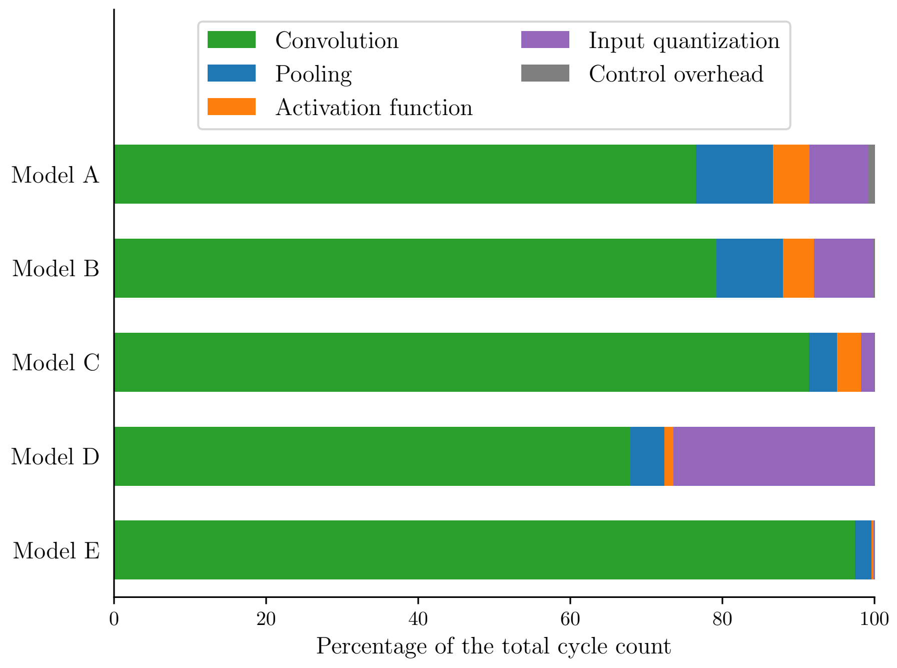


Figure 8.1: **Version 1**. Cycle count shares of different inference parts in portable, fully-scalar implementation.

by subtracting the cycles of all inference functions from the total cycle count of the DSP inference. In Figure 8.1, it can be seen very clearly that most cycles are spent processing the convolution layers as expected. However, in small models such as Models A and B, many cycles are also consumed in processing other network parts. When the model is computationally heavy such as Models C and E, almost all cycles are consumed in convolution layers. Model D is slightly different from other models as roughly 25 per cent of cycles are consumed in input quantization. Model D has the largest input data size, which explains the difference. Control overhead cycles consume a tiny fraction of total cycle counts in all models. It indicates that applying SIMD capabilities in the inference functions should effectively reduce cycle counts and processing time.

8.2.2 Input quantization

The first optimization to the fully-scalar implementation is the vectorization of the run-time input quantization. In Version 2, the scalar quantization function is replaced by a vectorized one, which implementation is described detailed in [10]. All other parts of the inference remain the same as in the portable, fully-scalar Version 1. Input quantization was chosen as the first part to be vectorized because it has been previously implemented, and reliable performance improvement was observed. As seen from Figure 8.1, some models also consume a significant amount of clock cycles in run-time quantization and thus should be vectorized in the early stage of this process.

Table 8.4: **Version 2**. Cycle count reduction in input quantization function itself, reduction in total inference cycles compared to fully-scalar reference (Version 1) and total cycles left in Version 2 compared to Version 1.

	Model A	Model B	Model C	Model D	Model E
Reduction in input quantization cycles	-95.9 %	-98.0 %	-98,0 %	-98.3 %	-97.9 %
Reduction compared to total cycles of `v1`	-7.4 %	-7.7 %	-1.7 %	-25.9 %	-0.1 %
Total cycles left compared to `v1`	92.6 %	92.3 %	98.3 %	74.1 %	99.9 %

Table 8.4 shows the performance results obtained in experiments with Version 2. The Table consists of three different statistics for all models. The first row contains the clock cycle reduction in per cent in vectorized input quantization compared to the scalar implementation. As can be seen, the vectorized implementation is very effective compared to the scalar one; in most models, roughly 98 % of cycles are saved. The second row contains the reduction in total clock cycles of the inference compared to the fully-scalar Version. From those numbers can be observed that in Model D, total cycles are reduced significantly, over 25 %, which is somewhat expected. In Models E and C, the input quantization does not consume many cycles,

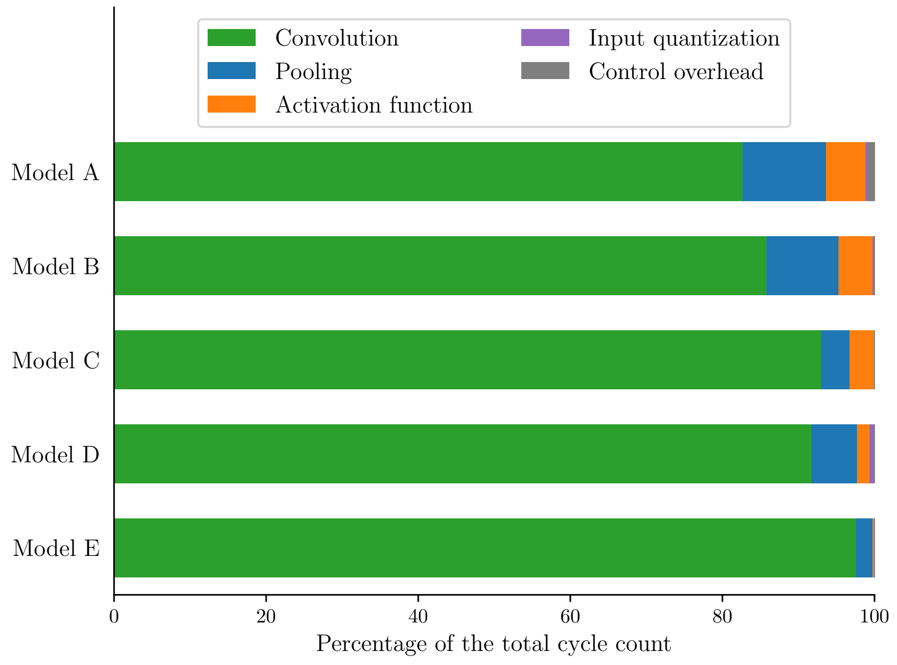


Figure 8.2: **Version 2**. Cycle count shares of different inference parts in implementation with vectorized input quantization.

even in Version 1, and thus the performance is only very minor. The third row gives the clock cycles left compared to the full-scalar reference after the optimizations performed in this stage are applied.

Figure 8.2 illustrates the percentages of cycles spent processing different parts of the inference in Version 2. Vectorization seems to be adequate. Cycle counts consumed to input quantization are minimized, even with Model D. It is even more apparent at this stage that the cycle counts are primarily consumed in convolution layers in all models tested in this work.

8.2.3 Convolution layers

The first vectorized part in the actual network inference is the convolution layer. Other parts of the inference algorithm, i.e., pooling layers and activation functions, still use the scalar versions of the functions. Two different implementations were developed for the convolution layer as discussed in Section 4.3.1. The second approach treating kernels as vectors (Algorithm 5) is applied with Model D, and the first approach treating kernels as a separate coefficient (Algorithm 4) is applied with the rest of the models. The obtained performance improvements with Version 3 are shown in Table 8.5. As expected, the vectorization of convolution layers offers excellent performance improvement because they are computationally the most intensive part of the inferences. In models employing Algorithm 4 in vectorization, the reduction in convolution layer clock cycles is well over 90 % in all models. With the smallest model, Model A, the reduction is not as high as in Models B, C and E. Relatively low number of arithmetic operations and small layer input dimensions does not enable full utilization of SIMD features. In the heaviest model, Model E, Version 3 consumes only 6.5 % of cycles compared to Version 1, meaning that over 93 % reduction in total cycles is achieved with only convolution layer vectorization. Most performance improvement potential is thus achieved already at this stage. With Models A-C, about 10 to 20 % clock cycles are still left in Version 3, which means that vectorization of the other network parts probably significantly impacts performance.

Table 8.5 shows that in Model D, convolution layer optimization does not offer

Table 8.5: **Version 3**. Cycle count reduction in convolution layer functions, reduction in total inference cycles compared to fully-scalar reference (Version 1) and total cycles left in Version 3 compared to Version 1.

	Model A	Model B	Model C	Model D	Model E
Reduction in conv. layer cycles	-94.1%	-95.8 %	-96.0 %	-77.9 %	-95.8 %
Reduction compared to total cycles of `v1`	-72.0 %	-75.9 %	-87.7 %	-52.9 %	-93.4 %
Total cycles left compared to `v1`	20.5 %	16.4 %	10.6 %	21.1 %	6.5 %

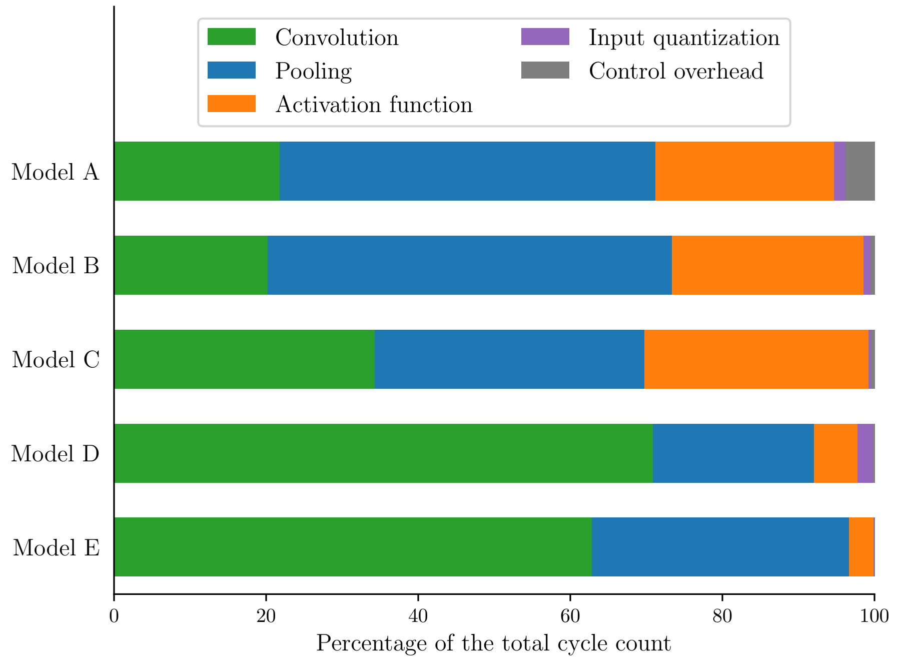


Figure 8.3: **Version 3**. Cycle count shares of different inference parts in implementation with vectorized convolution layers.

a significant performance gain. Model D employs convolution layers with stride parameters of more than one, which constraints to applying Algorithm 5. It treats kernels as vectors and computes internal sums of vectors to obtain the inner products of vectors. Algorithm 5 does not provide an optimal solution as the vectorized version results in a cycle count reduction of only about 78 %. From Table 7.6 can be observed that in Model D the first convolution layer is computationally far heavier than subsequent layers. It employs kernels of size 16, which means that with Algorithm 5, the vectorization can be performed without any dummy values occupying vector registers during computation. Despite this, the obtained speedups are worse than other test models, and the solution should be reconsidered.

Figure 8.3 shows the percentage shares of cycle counts that different parts of the inference consume. As can be observed, in Models A-C, most of the cycles are consumed in other parts of the inference after the convolution layers are vectorized. It suggests that there is still potential to improve performance by optimizing these parts, i.e. pooling layers and activation functions. In Models D and E, convolution layers consume most cycles even after SIMD is enabled. With these models, further vectorization will not probably provide that good gain. Instead, convolution layers should be optimized more.

8.2.4 Pooling layers

Three different approaches were developed to vectorize the pooling layers of the models as a part of this work. These are discussed more in Section 4.3.2. Two of the

Table 8.6: **Version 4**. Cycle count reduction in pooling layer functions, reduction in total inference cycles compared to fully-scalar reference (Version 1) and total cycles left in Version 4 compared to Version 1.

	Model A	Model B	Model C	Model D	Model E
Reduction in pool. layer cycles	-77.6 %	-95.4 %	-94.3 %	-96.0 %	-91.7 %
Reduction compared to total cycles of `v1`	-7.9 %	-8.3 %	-3.5 %	-4.3 %	-2.0 %
Total cycles left compared to `v1`	12.7 %	8.1 %	7.0 %	16.8 %	4.5 %

approaches, Algorithms 7 and 6 supports different stride parameters and execute average pooling layers that are applied in Models A-B and E. Algorithm 8 applies max pooling, which was used in Models C and D. As can be seen from the Figure 8.3 shown in the previous subsection, pooling layers consume second-highest number of cycles in all models. They should be properly optimized in order to obtain powerful DSP inference.

The observed changes in cycle count with Version 4 compared to previous versions are shown in Table 8.6. Vectorization of the pooling layer brings significant improvement in all models regardless of the used approach. Model A's cycle count

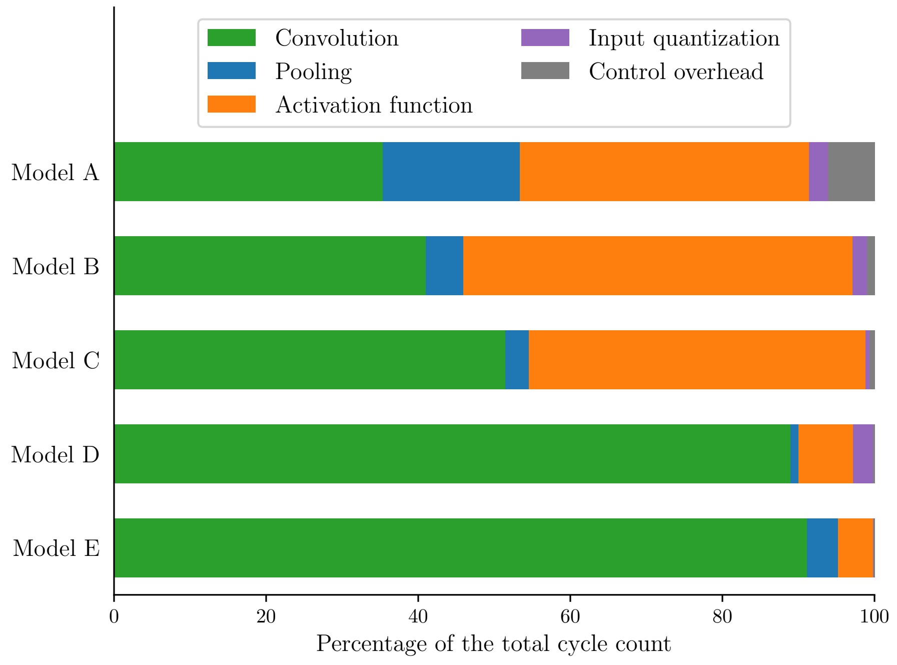


Figure 8.4: **Version 4**. Cycle count shares of different inference parts in implementation with vectorized pooling layers.

reduction in pooling layers is significantly lower than in other models, only about 78 %. Small models with short vector lengths do not benefit from vectorization that well. With other models, vectorization reduces cycle count by over 90 % and Model E offers the least performance improvement. Model E applies Algorithm 6, which performs average pooling with stride one. It seems that Algorithms 7 and 8 that apply pooling with stride $S = 2$ or $S = 4$ can be vectorized more effectively. These results make it hard to distinguish any differences between max and average pooling.

Figure 8.4 visualizes the cycle count shares of Version 4 inference implementation. It can be seen that in Models B-E, the share of pooling layers is very small after vectorization. Thus, it is hard to improve performance with pooling layers without optimizing other network parts. In the smallest model, Model A, the fraction of pooling is higher mainly due to the ungrateful input dimensions for SIMD in the pooling layers.

When only activations functions are the last part applying fully-scalar operations, the model speedups are in the same order as the model computational loads if Model D is excluded. Model D lacks speed up due to different convolution layer implementation and has almost 17 % cycles left compared to the reference. The most excellent speedup is obtained with Model E, with less than five per cent of the initial cycles left. As seen from Figure 8.4 there is still optimization to be done in all test models applied in this work.

8.2.5 Activation functions

Activation functions are the last part of the actual network inference, optimized as a part of this work. Vectorized implementation for two different activation functions is developed. Test models employ three different activation functions: Rectified Linear Unit (ReLU), Leaky ReLU and Sigmoid. Leaky ReLU and Sigmoid are implemented in this work, which is discussed in Section 6.3.3. Standard ReLU has already been developed in [10] and the exact implementation is applied here.

In Figure 8.4 shown earlier can be seen easily that different activation functions have different performances when scalar implementation is applied. Sigmoid in

Table 8.7: **Version 5**. Cycle count reduction in activation functions, reduction in total inference cycles compared to fully-scalar reference (Version 1) and total cycles left in Version 5 compared to Version 1.

	Model A	Model B	Model C	Model D	Model E
Reduction in act. function cycles	-81.9 %	-93.8 %	-96.0 %	-94.0 %	-93.9 %
Reduction compared to total cycles of `v1`	-3.9 %	-3.9 %	-3.0 %	-1.1 %	-0.2 %
Total cycles left compared to `v1`	8.7 %	4.2 %	4.0 %	15.7 %	4.3 %

Models A-B and Leaky ReLU in Model C consume more cycles than normal ReLU in models D-E. It is logical as the ReLU is a relatively simple activation function to implement on hardware compared to the two other functions. Thus vectorization of activation functions has good performance improvement potential in Models A-C.

Table 8.7 shows the reduction in cycle counts of the fully-vectorized implementation (Version 5) compared to the fully-scalar reference Version 1. The Table also shows the percentage share of cycles left after all parts of the inference is vectorized compared to the reference. As can be observed, all activation functions can be optimized very effectively using SIMD features. The cycle count reduction in activation function cycle counts is from roughly 94 % up to 96 % in models B-E. Leaky ReLU seems to utilize SIMD features the best, reaching the highest reduction in clock cycles. However, marginals between different activation functions are relatively small. Because Model A is very compact, it cannot utilize SIMD that well, and the reduction is only a bit over 80 %.

As expected, the overall gain in total cycle count reduction is relatively low because activation functions are not computationally the heaviest part of the inference. In models that apply Leaky ReLU and Sigmoid, the contribution is roughly 3-4 % of the total cycle counts of the scalar reference. The cycle reduction is significantly lower in Models D and E applying ReLU. Compared to the starting point, the total cycles left vary from Model A's 4.0 % to Model D's 15.7 %.

Figure 8.5 shows the cycle count shares of different inference parts in Version 5. Compared to the shares in scalar reference in Figure 8.1, the optimized implementation of Models B-E is well balanced, indicating that the SIMD features benefit all parts

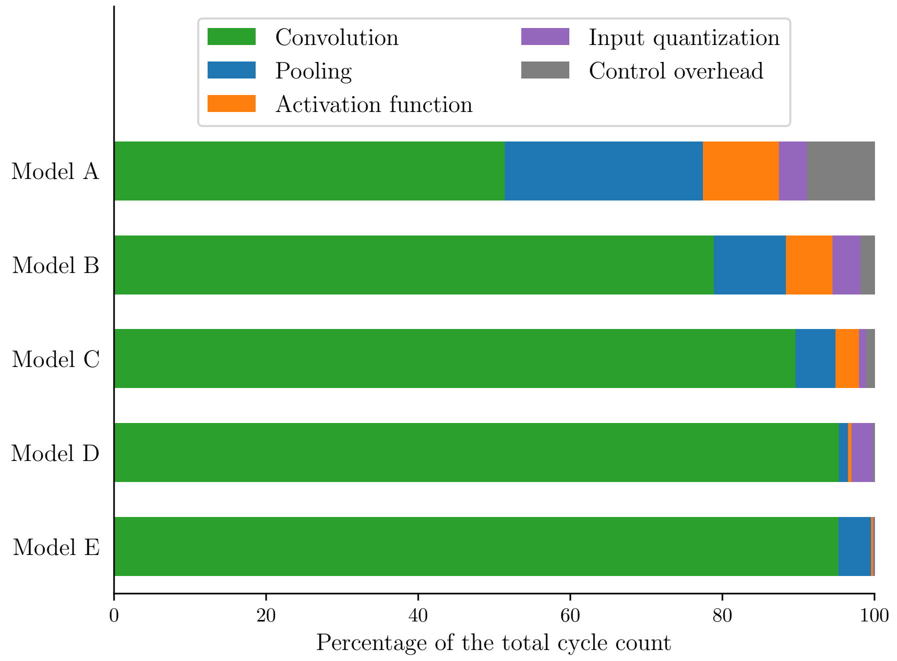


Figure 8.5: **Version 5**. Cycle count shares of different inference parts in fully-vectorized implementation.

of the inference relatively evenly. In Models D-E, convolution layers consume over 95 % of cycles, so the only potential to optimize further is to improve convolution layer implementation. In Model B and especially in Model A, the shares of other parts other than the convolution layer have increased significantly during vectorization. It indicates that other parts are more ineffective than convolution layers and that the control overhead has increased. With compact models such as model A, processing multiple input samples, i.e. batching, could improve performance. Activation functions are elementwise functions, and thus all input samples can be processed as one long vector when multiple input samples are processed in one batch.

8.2.6 Batching

The previous four subsections discussed the optimization of the DSP inference by minimizing cycle counts by applying different SIMD methods introduced in Section 6.3. The cycle counts of computationally intensive functions such as convolution layer are reduced dramatically. The robust implementation of inference functions results in some increased control overhead in smaller models that can be observed easily from Figure 8.5. Increasing the number of input samples processed per DSP job could better utilize the computation resources. At this point, each implementation has processed inference for one input sample at a time. In this subsection, the fully-vectorized DSP inference is accelerated using different batch sizes up to 32 input samples, and the obtained results are evaluated. When multiple input samples

Table 8.8: Total cycles left when processing a different number of input samples in one batch using the fully-vectorized version compared to fully-scalar reference (Version 1) processing batch size of one input sample. In brackets, the cycle count reduction is reported compared to the fully-vectorized inference (Version 5) processing batch size of one input sample.

Batch size	Model A	Model B	Model C	Model D	Model E
1	8.7 % (0.0 %)	4.2 % (0.0 %)	4.0 % (0.0 %)	15.7 % (0.0 %)	4.3 % (0.0 %)
2	7.1 % (-18.6 %)	4.0 % (-4.0 %)	4.0 % (-1.7 %)	15.7 % (-0.3 %)	4.2 % (-1.5 %)
4	6.3 % (-28.0 %)	4.0 % (-5.9 %)	4.0 % (-2.5 %)	15.4 % (-1.7 %)	4.1 % (-3.5 %)
8	5.9 % (-32.6 %)	3.8 % (-9.8 %)	3.9 % (-2.9 %)	- (-)	4.1 % (-4.6 %)
16	5.6 % (-35.1 %)	3.8 % (-10.3 %)	3.7 % (-7.1 %)	- (-)	- (-)
32	5.5 % (-36.3 %)	- (-)	3.7 % (-7.6 %)	- (-)	- (-)

are processed applying *batching*, the total number of inference function calls and conditional jumps in Algorithm 3 are reduced. Input quantization and all activation functions applied in this work are elementwise functions. Batching can improve DSP utilization with small models because it enables the processing of entire vector registers.

Table 8.8 shows the cycle counts left per input sample using different batch sizes compared to Version 1 processing one input sample. The results are obtained by dividing the entire batch's cycle counts by the number of input samples. In this work, the maximum batch size is 32. With Models B, D and E, only limited batch sizes can be applied because the entire input batch, all model parameters and intermediate results must fit in the internal memory of the DSP at the same time. As can be observed, the significant speedup from batching can be obtained with compact models. With the smallest model, Model A, almost 40 % cycle count reduction is observed when 32 input samples are processed with batching instead of one input sample. The reduction is much smaller with larger models, but all models are faster when batching is applied. As can be observed, the effect of batching saturates quite fast, and there is no point to increase batch sizes limitless. Even with small models, the most benefit from batching is achieved already with a batch size of 16.

Figure 8.6 shows the cycle count shares in different models when the batch size is the maximum possible amount of input samples. Increasing batch size has the desired effect of minimizing the share of cycles consumed by the control code and maximizing the time spent on useful computation. Cycles spent to control code almost vanish when batching is applied. It also improves the DSP utilization in

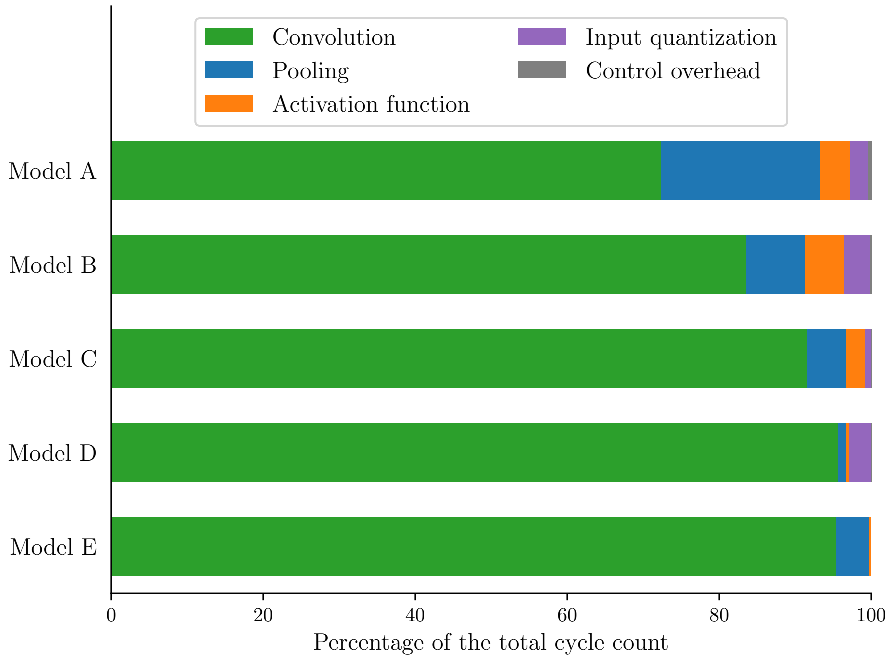


Figure 8.6: Cycle count shares of different inference parts in fully-vectorized implementation when maximum number of input samples is processed.

the activation functions and input quantization decreasing their cycle count shares, especially in smaller models. The results indicate that batching can be a powerful tool to minimize control overhead and improve DSP utilization in models with small layer input/output, which otherwise would be difficult.

8.3 Evaluation summary

This work aims to implement a framework that enables the efficient accelerating of 1-D CNNs in the CEVA-XC4500 digital signal processor. The framework includes an offline network parameter quantization procedure and real-time input quantization from floating-point values to fixed-point values. The applied quantization procedure behaves well and can be used without a notable accuracy decrease. The procedure was thoroughly tested with all five models using randomized input data and parameters. The obtained results are robust and accurate and show almost perfect consistency with the floating-point reference model. However, the quantization should be adequately tested with real-life data and trained parameters, which was not in the scope of this work. An approximation of the Sigmoid activation function was applied in two test models. It might be a potential source of inaccuracy in addition to quantization. However, it was observed that activation function approximation did not cause a significant accuracy decrease. Based on the experiences of this work, the quantization procedure should behave very well with any carefully built CNN model.

Five different network inference implementation versions with increasing order of vectorization were tested using five test models. All versions were designed such that

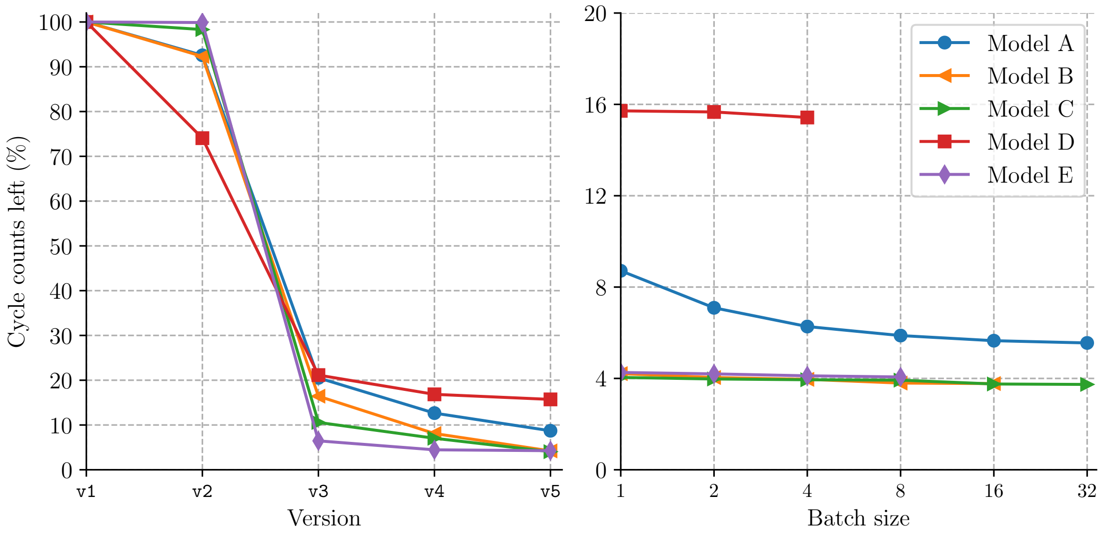


Figure 8.7: **Left**: Share of total cycles left in different implementation versions after inference parts are increasingly vectorized compared to the starting point, fully-scalar reference (`v1`). **Right**: Total cycles left for fully-vectorized version (`v5`) with different batch sizes compared to the starting point, fully-scalar reference (`v1`).

they provide comparable, bit-exact outputs. The performance evaluations presented in this Section show that every part of the network inference can be optimized effectively using the SIMD features the target DSP supports. Figure 8.7 illustrates the relative cycle counts of different implementation versions compared to the scalar reference implementation (`v1`), i.e. the cycle counts of `v1` equals to 100 %. The Figure also shows the relative cycle counts for the fully-vectorized version (`v5`) with different batch sizes. As can be seen, vectorization of each inference part separately significantly improves models' end-to-end performance. Because all test models consume most of the clock cycles in convolution layers, vectorization of those layers has the greatest impact in all cases.

Apparent differences in performance between models can be observed. Models B, C and E perform similarly and obtain almost identical relative performance after optimizations. With Model A, the SIMD approach cannot utilize a similar reduction in cycle counts. It is the most petite model with a lower computational load than other models and revealed worse performance in all inference functions, i.e. network inference parts. It seems that vectorization can be done effectively only when the layer input dimensions are large enough and the models have enough computational load. Then, most clock cycles are spent on the sequential parts controlling the inference workflow instead of the inference parts that execute the actual inference arithmetic, which can be parallelized with vector operations.

Model A is the only model that benefits from batching, i.e. processing multiple input samples combined. It indicates that batching can improve DSP utilization in small models with a low computational load. Model D is the most challenging model for optimization. It uses Algorithm 5 to process convolution layers that do not provide optimal performance. Algorithm 5 was also tested with other models than D, but due to the superior performance of Algorithm 4, the results are not even included here. It needs further investigation how a model that applies a stride parameter greater than one can be accelerated faster in the target DSP.

Table 8.9 shows the obtained speedups compared to the fully-scalar reference implementation with both batch size one and maximum possible batch size for the given model. The developed DSP implementation achieves the best speedups with Model C, resulting in 24.7 and 26.8 times fewer cycle counts than the scalar version without batching and with batching, respectively. The worst speedups are achieved with Model D, resulting in 6.4 and 6.5. As can be seen from the obtained speedups of Models B, C and E; with a large enough model that supports Algorithm 4 in convolution layers, the performance is very predictable.

As already discussed in Section 3.4 accelerating CNN inferences in a real-time

Table 8.9: Obtained speedups.

	Model A	Model B	Model C	Model D	Model E
Without batching	11.5	23.8	24.7	6.4	23.5
With batching	18.0	26.5	26.8	6.5	24.6

context, and especially 1-D inferences, is quite a novel and immature field. It makes comparing with literature examples challenging as not many comparable studies are available. However, some closely related examples can be found. In [10] and [9], MLP inferences were implemented and optimized for the same target DSP. The observed speedup of 11.5 with Model A in this work is roughly in line with the highest speedups achieved in those two previous works. Model A is the smallest of the five models, and the computational load is the most comparable with the models in those previous works. These previous works did not implement any larger models, showing how the performance converges if the computational load of the inference is increased. It should also be noted that Model A is a CNN, not MLP and the tested models are rather different, making straightforward comparison challenging.

In [75] vectorized Frequency Offset Compensation (FOC) algorithm based on finite impulse response (FIR) filter was developed for the same target DSP. As the FIR filter is based on linear convolution, and the hardware implementation of the FOC algorithm shares the key ideas with Algorithm 4 despite that the use cases for the implementations are somewhat different. Speedups up to 26 were observed, which is relatively close to speedups obtained with models B-C and E. Both implementations are likely close to the limit, how high cycle count reduction compared to portable scalar implementations can be achieved with vectorization based optimization when this particular DSP is used.

Chapter 9

Conclusions

This work studies how effectively a state-of-the-art digital signal processor designed for 5G physical layer processing can be utilized to accelerate convolutional neural network inference. The author of this work designed, implemented and optimized a one-dimensional convolutional neural network inference framework specifically for CEVA-XC4500 DSP. The framework consists of an offline preprocessing part implemented in Python, and a real-time inference part implemented in C++. The preprocessing part includes quantizing the floating-point network parameters into fixed-point representations and constructing a specific network configuration to execute the real-time inference on the DSP.

The real-time part of the framework consists of the inference code, which applies multiple DSP specific features to accelerate the neural network inference. The optimized DSP code is designed to utilize the SIMD capabilities of the target hardware to reduce the processor cycles required to process the inference significantly. Compared to scalar reference implementation that does not utilize SIMD, up to 27 times faster inferences were achieved with optimized implementations. The results indicate that the DSP can significantly accelerate inference computation with a minimum accuracy decrease. However, highly tailored hardware implementation restricts the NN model architecture. As properties of the model such as layer parameters notably affect how well the SIMD features can be utilized, it might be possible to get significant gains in inference performance with only modest changes to the CNN architectures.

The developed framework shares ideas with the general-purpose inference frameworks and the most specialized DSP algorithms investigated in previous works. The framework supports only a single DSP and a single class of NN models instead of a broader set of different hardware and a more comprehensive range of NN model types. It makes the framework far more specialized than the general-purpose DL frameworks, which typically support a more comprehensive range of DL model types and different hardware such as mobile phones and embedded devices. At the same time, the framework tries to offer some flexibility by supporting a wide range of different network configurations. For example, the number of layers and their dimensions are almost freely configurable. It is more flexible than hard-coding the implementation details to fit specific model architectures. The purpose is to enable easy reuse of the same framework with different model sizes and layer configurations. The obtained performance characteristics suggest that the increased flexibility does not automatically lead to significant performance penalties. The key is to find the appropriate balance between the flexibility of general-purpose frameworks and the performance of highly hardware optimized and unscalable solutions.

From the beginning of this work, the DSP implementation is designed to be compatible with the current software architecture, particularly the existing DSP job system on the target SoC. The developed inference implementation is optimized and evaluated entirely from the DSP point of view. The evaluation ignores any potential performance bottlenecks in the end-to-end performance when the framework is evaluated in the actual SoC context. An obvious continuation of this work would be to consolidate the framework with an adequately trained model, evaluate the combination as part of a complete signal processing chain, and study the framework's feasibility on the target SoC in authentic conditions.

Bibliography

[1] M. Enescu, *5G New Radio: A Beam-based Air Interface*. John Wiley & Sons, 2020.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. Available: http://www.deeplearningbook.org. Accessed: 28 Jan 2022.

[3] T. Wang, C.-K. Wen, H. Wang, F. Gao, T. Jiang, and S. Jin, "Deep learning for wireless physical layer: Opportunities and challenges," *China Communications*, vol. 14, no. 11, pp. 92–111, 2017. Available: https://www.doi.org/10.1109/CC.2017.8233654.

[4] M. Honkala, D. Korpi, and J. M. Huttunen, "DeepRx: Fully convolutional deep learning receiver," *IEEE Transactions on Wireless Communications*, vol. 20, no. 6, pp. 3925–3940, 2021. Available: http://doi.org/10.1109/TWC.2021.3054520.

[5] H. Sun, A. O. Kaya, M. Macdonald, H. Viswanathan, and M. Hong, "Deep Learning Based Preamble Detection and TOA Estimation," in *2019 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, 2019. Available: https://doi.org/10.1109/GLOBECOM38437.2019.9013265.

[6] H. Huang, J. Yang, H. Huang, Y. Song, and G. Gui, "Deep Learning for Super-Resolution Channel Estimation and DOA Estimation Based Massive MIMO System," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 9, pp. 8549–8560, 2018. Available: https://doi.org/10.1109/TVT.2018.2851783.

[7] 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. Available: https://doi.org/10.1109/TCCN.2017.2758370.

[8] S. Dörner, S. Cammerer, J. Hoydis, and S. t. Brink, "Deep Learning Based Communication Over the Air," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 132–143, 2018. Available: https://www.doi.org/10.1109/JSTSP.2017.2784180.

[9] T. Alonen, "Inference with a neural network in digital signal processing under hard real-time constraints," Master's thesis, Aalto University, School of Electrical

Engineering, Jan 2020. Available: http://urn.fi/URN:NBN:fi:aalto-202001261880.

[10] J. Korvuo, "Multilayer Perceptron Inference Solution for Digital Signal Processing on the Physical Layer," Master's thesis, Aalto University, School of Science, Nov 2020. Available: http://urn.fi/URN:NBN:fi:aalto-2020122056371.

[11] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, 1980. Available: https://www.doi.org/10.1109/TCOM.1980.1094702.

[12] M. M. Ahamed and S. Faruque, "Propagation factors affecting the performance of 5G millimeter wave radio channel," in *2016 IEEE International Conference on Electro Information Technology (EIT)*, pp. 0728–0733, 2016. Available: https://www.doi.org/10.1109/EIT.2016.7535329.

[13] A. F. Molisch, *Wireless communications*. John Wiley & Sons, 2 ed., 2011.

[14] J. Ly Ponce, "Application of Autoencoders in pilot-based Channel Estimation for 5G Physical Uplink Shared Channel," Master's thesis, Aalto University. School of Electrical Engineering, Jan 2021. Available: http://urn.fi/URN:NBN:fi:aalto-202101311760.

[15] M. E. Morocho-Cayamcela, H. Lee, and W. Lim, "Machine learning for 5G/B5G mobile and wireless communications: Potential, limitations, and future directions," *IEEE Access*, vol. 7, pp. 137184–137206, 2019. Available: https://doi.org/10.1109/ACCESS.2019.2942390.

[16] E. Zehavi, "8-PSK trellis codes for a Rayleigh channel," *IEEE Transactions on Communications*, vol. 40, no. 5, pp. 873–884, 1992. Available: https://www.doi.org/10.1109/26.141453.

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

[18] D. Moolchandani, A. Kumar, and S. R. Sarangi, "Accelerating CNN Inference on ASICs: A Survey," *Journal of Systems Architecture*, vol. 113, p. 101887, 2021. Available: https://doi.org/10.1016/j.sysarc.2020.101887.

[19] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," in *Proceedings of the 32nd International Conference on Machine Learning*, vol. 37 of *Proceedings of Machine Learning Research*, pp. 1737–1746, PMLR, 2015. Available: http://proceedings.mlr.press/v37/gupta15.html.

[20] CEVA, “CEVA Deep Neural Network (CDNN),” 2015. Software available: https://www.ceva-dsp.com/product/ceva-deep-neural-network-cdnn/. Accessed: 28 Jan 2022.

[21] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, “The Deep Learning Compiler: A Comprehensive Survey,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, p. 708–727, 2021. Available: http://doi.org/10.1109/tpds.2020.3030548.

[22] T. Gruber, S. Cammerer, J. Hoydis, and S. t. Brink, “On deep learning-based channel decoding,” in *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–6, 2017. Available: https://www.doi.org/10.1109/CISS.2017.7926071.

[23] E. Nachmani, Y. Be’ery, and D. Burshtein, “Learning to decode linear codes using deep learning,” in *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 341–346, 2016. Available: https://www.doi.org/10.1109/ALLERTON.2016.7852251.

[24] W. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943. Available: https://doi-org/10.1007/BF02478259.

[25] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, “Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012. Available: http://doi.org/10.1109/MSP.2012.2205597.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” in *Advances in Neural Information Processing Systems*, vol. 25, 2012. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[27] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain,” *Psychological review*, vol. 65, no. 6, pp. 386–408, 1958. Available: https://doi.org/10.1037/h0042519.

[28] S. Shanmuganathan, “Artificial neural network modelling: An introduction,” in *Artificial Neural Network Modelling*, vol. 628 of *Studies in Computational Intelligence*, pp. 1–14, Springer, 2016. Available: https://doi.org/10.1007/978-3-319-28495-8_1.

[29] S. Linnainmaa, “The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors,” Master’s thesis (in Finnish), University of Helsinki, Department of Computer Science, 1970.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. Available: https://doi.org/10.1038/323533a0.

[31] J. L. McClelland, D. E. Rumelhart, and PDP Research Group, *Parallel distributed processing: explorations in the microstructure of cognition*, vol. 2. MIT Press, 1986.

[32] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *3rd International Conference for Learning Representations*, 2015. Available: https://arxiv.org/abs/1412.6980.

[33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. Available: https://jmlr.org/papers/v15/srivastava14a.html.

[34] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. Available: https://doi.org/10.1109/CVPR.2018.00286.

[35] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint*, 2017. Available: https://arxiv.org/abs/1704.04861.

[36] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," *arXiv preprint*, 2016. Available: https://arxiv.org/abs/1602.07360.

[37] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015. Software available: https://tensorflow.org. Accessed: 28 Jan 2022.

[38] A. Gulli and S. Pal, *Deep learning with Keras.* Packt Publishing Ltd, 2017.

[39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library,"

in *Advances in Neural Information Processing Systems*, vol. 32, pp. 8024–8035, 2019. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[40] J. Howard and S. Gugger, "FastAI: A Layered API for Deep Learning," *Information*, vol. 11, no. 108, 2020. Available: https://doi.org/10.3390/info11020108.

[41] "ONNX: Open Neural Network Exchange." Software available: https://onnx.ai. Version 1.9.0. Accessed: 27 Jan 2022.

[42] S. Jagannathan, M. Mody, and M. Mathew, "Optimizing convolutional neural network on DSP," in *2016 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 371–372, 2016. Available: https://doi.org/10.1109/ICCE.2016.7430652.

[43] G. Zeng, X. Hu, and Y. Chen, "Optimizing Convolution Neural Network on the TI C6678 multicore DSP," *MATEC Web Conf.*, vol. 246, 2018. Available: https://doi.org/10.1051/matecconf/201824603044.

[44] N. P. Jouppi, C. Young, N. Patil, D. Patterson, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017. Available: https://doi.org/10.1145/3079856.3080246.

[45] T. Chen, T. Moreau, Z. Jiang, L. Zheng, *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, USENIX Association, 2018. Available: https://www.usenix.org/conference/osdi18/presentation/chen.

[46] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, *et al.*, "Glow: Graph Lowering Compiler Techniques for Neural Networks," *arXiv preprint*, 2018. Available: http://arxiv.org/abs/1805.00907.

[47] D. Lin, S. Talathi, and S. Annapureddy, "Fixed Point Quantization of Deep Convolutional Networks," in *Proceedings of The 33rd International Conference on Machine Learning*, vol. 48 of *Proceedings of Machine Learning Research*, pp. 2849–2858, 2016. Available: https://proceedings.mlr.press/v48/linb16.html.

[48] CEVA, "CEVA-XC4500 Architecture Specification Volume I," Dec 2018.

[49] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. Available: https://doi.org/10.1109/5.726791.

[50] "CS231n Convolutional Neural Networks for Visual Recognition." Course material, Stanford University, 2021. Available: https://cs231n.github.io. Accessed: 28 Jan 2022.

[51] S. Kiranyaz, O. Avci, O. Abdeljaber, T. Ince, M. Gabbouj, and D. J. Inman, "1D convolutional neural networks and applications: A survey," *Mechanical systems and signal processing*, vol. 151, p. 107398, 2021. Available: https://doi.org/10.1016/j.ymssp.2020.107398.

[52] R. U. Acharya, H. Fujita, O. S. Lih, Y. Hagiwara, J. H. Tan, and M. Adam, "Automated detection of arrhythmias using different intervals of tachycardia ECG segments with convolutional neural network," *Information Sciences*, vol. 405, pp. 81–90, 2017. Available: https://doi.org/10.1016/j.ins.2017.04.012.

[53] O. Abdeljaber, O. Avci, M. S. Kiranyaz, B. Boashash, H. Sodano, and D. J. Inman, "1-D CNNs for structural damage detection: Verification on a structural health monitoring benchmark data," *Neurocomputing*, vol. 275, pp. 1308–1317, 2018. Available: https://doi.org/10.1016/j.neucom.2017.09.0699.

[54] L. Eren, T. Ince, and S. Kiranyaz, "A Generic Intelligent Bearing Fault Diagnosis System Using Compact Adaptive 1D CNN Classifier," *Journal of Signal Processing Systems*, vol. 91, no. 2, pp. 179–189, 2019. Available: https://doi.org/10.1007/s11265-018-1378-3.

[55] W. Zhang, C. Li, G. Peng, Y. Chen, and Z. Zhang, "A deep convolutional neural network with new training methods for bearing fault diagnosis under noisy environment and different working load," *Mechanical Systems and Signal Processing*, vol. 100, pp. 439–453, 2018. Available: https://doi.org/10.1016/j.ymssp.2017.06.022.

[56] C. S. Burrus, *Fast Fourier Transforms.* OpenStax CNX, 2012. Available: http://cnx.org/content/col10550/1.22.

[57] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *arXiv preprint*, 2014. Available: https://arxiv.org/abs/1410.0759.

[58] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," in *2nd International Conference on Learning Representations, ICLR 2014*, 2014. Available: https://arxiv.org/abs/1312.5851.

[59] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A quantitative approach.* Morgan Kaufmann, 5 ed., 2011.

[60] J. P. Shen and M. H. Lipasti, *Modern processor design: fundamentals of superscalar processors.* Waveland Press, Inc., 1 ed., 2013.

[61] R. Oshana, *DSP for Embedded and Real-Time Systems.* Newnes, 2012.

[62] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers.* O'Reilly Media, Inc., 3 ed., 2005.

[63] L. Tan and J. Jiang, *Digital Signal Processing: Fundamentals and Applications*. Academic Press, 3 ed., 2018.

[64] IEEE, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. Available: https://doi.org/10.1109/IEEESTD.2019.8766229.

[65] R. Yates, "Fixed-Point Arithmetic: An Introduction." Digital Sound Labs, 2001. Available: https://www.cpplab.net/web/dsp/fp.pdf.

[66] CEVA, "CEVA-XC4500 Architecture Specification Volume III (MSS)," Dec 2018.

[67] CEVA, "CEVA-XC4500 Methodology for Optimal Vectorization Application Note," Oct 2014.

[68] CEVA, "CEVA-XC4500 Architecture Specification Volume II," Dec 2018.

[69] V. Rajagopal, C. K. Ramasamy, A. Vishnoi, R. N. Gadde, N. R. Miniskar, and S. K. Pasupuleti, "Accurate and Efficient Fixed Point Inference for Deep Neural Networks," in *2018 25th IEEE International Conference on Image Processing (ICIP)*, pp. 1847–1851, 2018. Available: https://doi.org/10.1109/ICIP.2018.8451268.

[70] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "Fast Approximations of Activation Functions in Deep Neural Networks when using Posit Arithmetic," *Sensors*, vol. 20, no. 5, 2020. Available: https://www.mdpi.com/1424-8220/20/5/1515.

[71] "ONNX Runtime," 2021. Software available: https://onnxruntime.ai/. Version 1.7.0. Accessed: 27 Jan 2022.

[72] S. Malek, F. Melgani, and Y. Bazi, "One-dimensional convolutional neural networks for spectroscopic signal regression," *Journal of Chemometrics*, vol. 32, no. 5, p. e2977, 2018. Available: https://doi.org/10.1002/cem.2977.

[73] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, 2015. Available: https://doi.org/10.1109/ICCV.2015.123.

[74] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, vol. 9 of *Proceedings of Machine Learning Research*, pp. 249–256, PMLR, 2010. Available: https://proceedings.mlr.press/v9/glorot10a.html.

[75] T. Ebeling, "Frequency Offset Compensation in a 5G Base Station Receiver," Master's thesis, Aalto University, School of Electrical Engineering, Nov 2019. Available: http://urn.fi/URN:NBN:fi:aalto-201912226629.

Appendix A

CEVA intrinsics

Table A.01: CEVA intrinsics with description applied in the pseudocodes of Chapter 6. Part 1.

CEVA intrinsic	Description
v_load(`ptr`)	Loads 16 double-words from the memory address pointed by `ptr`, and writes them to a vector register file addresses by the return value.
v_load_clone(`ptr`)	Loads 8 double-words from the memory address pointed by `ptr`, and clones them identically into both VCUs.
v_load_vuX(`ptr`,i)	Loads 8 double-words to destination VCU indicated by i from the memory address pointed by `ptr`.
c_load(`ptr`)	Loads one double-word from the memory address pointed by `ptr`, and writes the value as a coefficient to common register file addresses by the return value.
v_store(k,$\boldsymbol{a}$,`ptr`)	Stores k first 16-bit words in vector $\boldsymbol{a}$ to a internal memory address pointed by `ptr`.
v_store_vuX(k,$\boldsymbol{a}$,`ptr`,i)	Stores k first 16-bit words in source VCU of vector $\boldsymbol{a}$ to a internal memory address pointed by `ptr`.
v_shift($\boldsymbol{a}$,k)	Shifts the elements in vector pointed by $\boldsymbol{a}$ with k bits.
v_fill(c,`pred`)	Fills the vector pointed by the return value with constant c. Atomic operations can be controlled with vector predicate `pred`.
v_add($\boldsymbol{a}$,$\boldsymbol{b}$,`pred`)	Performs inter-vector addition for vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. Second argument can be also constant c, which is added to all elements in vector $\boldsymbol{a}$. Atomic operations can be controlled with vector predicate `pred`.
v_permute($\boldsymbol{a}$,$\boldsymbol{cfg}$)	Performs shuffling for vector $\boldsymbol{a}$ based on indices in configuration vector $\boldsymbol{cfg}$. Vector $\boldsymbol{cfg}$ contains 16 4-bit indexes which define positions $(0 \ldots 15)$ of words to be extracted from vector $\boldsymbol{a}$.

Table A.02: CEVA intrinsics with description applied in the pseudocodes of Chapter 6. Part 2.

CEVA intrinsic	Description
v_min($\boldsymbol{a}$,$\boldsymbol{b}$)	Compares if elements in vector $\boldsymbol{a}$ are lower than elements in vector $\boldsymbol{b}$. Returns a vector containing lower elements, or a vector predicate in which 1 indicates that the element is lower in vector $\boldsymbol{a}$ and 0 otherwise.
v_max($\boldsymbol{a}$,$\boldsymbol{b}$)	Compares if elements in vector $\boldsymbol{a}$ are greater than elements in vector $\boldsymbol{b}$. Returns a vector containing greater elements, or a vector predicate in which 1 indicates that the element is greater in vector $\boldsymbol{a}$ and 0 otherwise.
vv_mltply($\boldsymbol{a}$,g,K,$\boldsymbol{b}, h, L,$ `pred`)	Performs dot product for vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. K and L indicate word part (LOW,HIGH) that is used from vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, respectively. Similarly, constants g and h determine the offset in double-words for the dot product and `pred` controls atomic operations.
vc_mltply($\boldsymbol{a}, g, K, c, L,$ `pred`)	Performs dot product for vector $\boldsymbol{a}$ and constant c. K and L indicate word part (LOW,HIGH) that is used from operands, respectively. Constant g determines the offset in double-words for the vector $\boldsymbol{a}$ and `pred` controls atomic operations.
vv_mac($\boldsymbol{a}, g, K, \boldsymbol{b}, h, L, \boldsymbol{o},$ `pred`)	Performs dot product for vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, and accumulates the result with vector $\boldsymbol{o}$. K and L indicate word part (LOW, HIGH) that is used from operands, respectively. Constants g ja h determines the offset in double-words for the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. Vector predicate `pred` controls atomic operations.
vc_mac($\boldsymbol{a}, g, K, c, L, \boldsymbol{o},$ `pred`)	Performs dot product for vector $\boldsymbol{a}$ and constant c, and accumulates the result with vector $\boldsymbol{o}$. K and L indicate word part (LOW, HIGH) that is used from operands, respectively. Constant g determines the offset in double-words for the vector $\boldsymbol{a}$. Vector predicate `pred` controls atomic operations.
v_pack($\boldsymbol{a}$)	Performs packing of vector $\boldsymbol{a}$ containing 32 32-bit elements into a vector of 32 16-bit element vector addressed by the return value. By default LOW part of the elements is used.
v_move(g, c)	Performs data move from value c to o a vector register file addresses by the return value. Constant g determines the offset in double-words for the operation.