Master's thesis

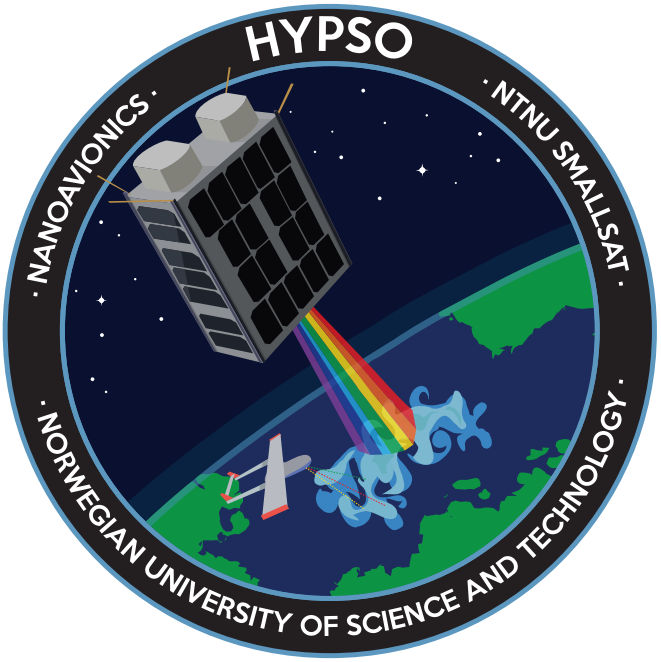
NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Simen Netteland

Exploration and Implementation of Large CNN Models for Image Segmentation in Hyperspectral CubeSat Missions

NTNU Small Satellite Lab

Master's thesis in Electronic Systems Design
Supervisor: Milica Orlandic
June 2022



NTNU
Norwegian University of
Science and Technology

Simen Netteland

Exploration and Implementation of Large CNN Models for Image Segmentation in Hyperspectral CubeSat Missions

NTNU Small Satellite Lab

Master's thesis in Electronic Systems Design
Supervisor: Milica Orlandic
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Abstract

Large Convolutional Neural Network (CNN) models such as the UNet have been shown to be especially efficient at performing semantic segmentation on hyperspectral images. Deploying such networks on Earth Observation (EO) satellites, such as CubeSats, provides the possibility for greatly reducing the required data for downlink. These networks are however computationally heavy and have large memory footprints which makes them difficult to implement on resource constrained embedded devices. This thesis tackles this issue through implementing a memory efficient UNet software framework capable of running on ARM devices with limited system memory. Additionally, a hardware/software codesign is proposed to accelerate the convolutional layers on an Field Programmable Gate Array (FPGA). The software framework achieves a pixel prediction time of 294μs on $512 \times 512 \times 102$ hyperspectral image cubes, with a system memory footprint of approximately 120MB at maximum. Finally, the thesis proposes a standardized format called Weights Interleaved by Filters (WIF) for efficient storage of trainable parameters from CNNs for embedded devices.

Sammendrag

Dype konvolusjonelle nevrale nettverksmodeller som UNet arkitekturen har blitt vist å være spesielt effektive til å segmentere hyperspektral bildedata. Det å implementere slike nettverk på kubesatellitter som driver med hyperspektral jordobservasjon kan potensielt drastisk redusere dataen nødvending for nedlastning. Disse nettverkene er derimot svært ressurskrevende og kan være utfordrende å implementere på mindre innvevde systemer med reduserte ressurser. Denne oppgaven håndterer dette problemet ved å implementere en ressurseffektiv UNet programvare som kan kjøre på ARM systemer. I tillegg legger oppgaven frem et forslag for akselerering av konvolusjonelle lag på en Field Programmable Gate Array (FPGA). Programvaren oppnår en piksel prediksjonstid på om lag 294μs på et $512 \times 512 \times 102$ hyperspektralt bilde. Avslutningsvis så foreslår oppgaven et standardisert vektformat for nevrale nettverk kalt Weights Interleaved by Filters (WIF), som gjør det mer effektivt å håndtere vektdata på innvevde systemer.

Preface

This thesis concludes my three plus two years of studies into electronics, automation and embedded systems here at NTNU. I have with this work written both my bachelor's and master's thesis within the SmallSat team at NTNU, and I have gotten to experience the workflow and team spirit up until and after the launch of the HYPSO-1 satellite. This is something I truly appreciate. Though my studies have given me much, being a part of the HYPSO team has taught me the most.

Working on this thesis has provided me with newfound insight into the concepts and use cases of convolutional neural networks. It has been exciting to explore this field of study and its challenges, and I'm now looking forward to seeing how it will evolve.

First of all, would I like to thank my supervisor Milica Orlandic, for guiding me through both my bachelor's and master's thesis here at the SmallSat team. You have pushed me in challenging directions, from which I've learnt so much.

I would also like to truly thank my family for supporting me through the years. You have inspired and encouraged me to do what I love for as long as I can remember, and is the reason for where I am today.

Finally, I'd like to thank the people working with me in the SmallSat lab for the last few months. Without our coffee break(in)s and joking around, I would probably have gone bonkers. This is also very much with the support of my girlfriend, thank you!

Contents

Acronyms

AI Artificial Intelligence. 13

AMBA Advanced Microcontroller Bus Architecture. 26

ANN Artificial Neural Network. 13, 14, 15, 17

API Application Programming Interface. 40, 44, 46, 58, 59, 97, 104

APU Application specific Processing Unit. 24

ASCII American Standard Code for Information Interchange. 65, 66, 91, 92, 93

ASIC Application-Specific Integrated Circuit. 25

AXI Advanced eXtensible Interface. 26, 27, 34, 80, 81, 84, 85, 107

BIL Band Interleaved by Line. 12

BIP Band Interleaved by Pixel. 12, 33, 45, 60, 63, 64, 65, 66, 78, 83, 95, 105, 109

BRAM Block Random Access Memory. 24, 27, 80, 84, 106

BSQ Band SeQuential. 12

CNN Convolutional Neural Network. iii, 4, 5, 6, 7, 19, 20, 21, 22, 25, 26, 36, 37, 38, 39, 40, 43, 44, 45, 46, 49, 51, 52, 53, 54, 57, 58, 60, 63, 64, 67, 74, 79, 80, 98, 104, 109, 110

CONOPS Concept of Operation. 7, 8

CPU Centeral Processing Unit. 24, 25, 26, 31, 32, 34, 46, 59, 60, 79, 80, 81, 84, 85, 94, 98, 100, 101, 105, 109, 110

DDR Double Data Rate. 94

DL Deep Learning. 4, 7, 13, 15, 19, 26, 29, 37, 54, 89

DMA Direct Memory Access. 34, 46, 80, 81, 84, 85, 107

EO Earth Observation. iii, 7, 8, 9, 37, 39

ESA European Space Agency. 4, 39

FPGA Field Programmable Gate Array. iii, v, ix, 5, 6, 7, 24, 25, 26, 27, 33, 34, 38, 45, 46, 49, 50, 57, 58, 59, 60, 78, 79, 80, 81, 82, 84, 85, 106, 107, 109, 110

FPU Floating Point Unit. 24

GCC GNU Compiler Collection. 32, 33, 99, 105

GPU Graphics Processing Unit. 24, 26, 37, 79, 94, 105

HAB Harmful Algal Bloom. 8

HLS High-Level Synthesis. 38, 46, 107, 110

HYPSO-2 HYPer-spectral Smallsat for ocean Observation. 5, 7, 8, 33, 34, 38, 39, 41, 63, 79, 106, 109, 110

HYPSO-1 HYPer-spectral Smallsat for ocean Observation. vii, 3, 4, 5, 6, 7, 8, 10, 33, 34, 37, 38, 39, 41, 47, 52, 63, 66, 94, 105, 106, 109, 110

IEEE Institute of Electrical and Electronics Engineers. 28, 29, 58

IoU Intersection over Union. 89

ISA Instruction Set Architecture. 24, 32

KSAT Kongsberg Satellite Services. 8

LUT LookUp Table. 25

MB MegaByte. 8

ML Machine Learning. 4, 7, 13, 16, 18, 60, 79

MMU Memory Management Unit. 31, 32

MPSoC Multi-Processing System on Chip. 24, 34

NTNU Norwegian University of Science and Technology. vii, 3, 4, 7, 8, 39

OS Operating System. 30, 39, 63, 64, 94, 105

PL Programmable Logic. 24

RAM Random Access Memory. 27, 31, 34, 50, 82, 84, 94, 103

ReLU Rectified Linear Unit. 15, 16, 67

RGB Red, Green and Blue. 4, 9, 10, 12, 19, 52, 53, 55, 88

RISC Reduced Instruction Set Computer. 32

RPU Real-time Processing Unit. 24

SIMD Single Instruction Multiple Data. 24, 100, 110

SmallSat Small Satellite Lab. vii, 3, 5, 7, 39, 44

SoC System on Chip. 24, 26, 33, 34, 79, 94, 95, 106

SOM System On Module. 25

WIF Weights Interleaved by Filters. iii, v, 5, 60, 61, 62, 63, 64, 65, 66, 78, 92, 104, 109

1 Introduction

> *Space is an inspirational concept that allows you to dream big.*
>
> — **Peter Diamandis**
> (Engineer)

Space offers a wast amount of possibilities for remote sensing of our globe, and as research and investment into Cube Satellites (CubeSats) is constantly growing, smaller companies and institutions like the Norwegian University of Science and Technology (NTNU) are now able to conduct their own space missions.

In January 2022, the Small Satellite Lab (SmallSat) at NTNU successfully launched its first CubeSat, the HYPer-spectral Smallsat for ocean Observation (HYPSO-1), through one of SpaceXs' rideshare missions. The HYPSO-1 satellite carries a hyper-spectral camera capable of capturing ocean colour across the world, supporting the growing field of hyperspectral remote sensing with its mission. Hyperspectral data is especially useful in remote sensing, as it captures information that can be used to determine the chemical compound of what is measured [1][p. 2]. In the case of HYPSO-1, this allows the satellite to support detecting harmful algal blooms in near-realtime [2].

Cubesats, such as HYPSO-1 do, however, come with their extra challenges, such as reduced power capacity, which propagates into less powerful hardware for processing and downlinking data. This can be an issue for hyperspectral imaging, as it produces notoriously large amounts of sampled data, resulting in a larger data size for downlink. There exist ways of reducing the size of the information, such as lossless compression with the CCSDS123 standard [3], or removing redundant hyperspectral bands. These are methods deployed by the HYPSO-1 satellite, however, data sizes still remain within the sizes of megabytes and cause issues where one desires fast downlink and detection.

A different approach to these reduction techniques is to instead perform on-board processing for extracting the essence of the information one seeks. In the case of HYPSO-1, where it is desired to detect ocean colour, one could instead determine

the type of algae and its location through on-board processing. This would reduce the data size needed for downlink and essentially make it insignificant as this type of information could be very compact and small. This would allow for far more processed captures within the scope of an orbit and support the desire for near-real-time detection.

Detecting specific compounds within hyperspectral data, which noticeably has been sampled from space, is not easy, as its characteristics must be determined while also correcting for atmospheric conditions. However, it has been shown that Machine Learning (ML) algorithms, such as neural networks, are especially good at tackling these types of issues [4].

Deep Learning (DL) algorithms used with remote sensing is a field of study that is starting to emerge with the rise of more energy-efficient and powerful embedded devices. DL networks, which are densely connected layers of artificial neurons, have an excellent ability to learn features within data [5]. Since hyperspectral images contain far more information for a given spatial pixel than that of an RGB image, DL algorithms can extract more precise features from the data. This has been shown through the use of Convolutional Neural Networks (CNNs), which performs better at semantic segmentation when trained on hyperspectral data [6, 7].

Semantic image segmentation is a processing method that could be utilized on a satellite to determine the chemical compounds of a hyperspectral capture. It could provide high accuracy if trained correctly and with a fitting CNN architecture. A famous CNN architecture for image segmentation is the UNet, which is capable of achieving highly accurate semantic segmentation with little training data [8]. Additionally, the UNet has also been shown to be very effective at image segmentation with hyperspectral data [6], making it a good option for satellite missions like HYPSO-1.

CNN networks like the UNet are computationally heavy, requiring several convolutional sliding window operations to segment images. They also utilize millions of trainable parameters that require large memory sizes and are therefore hard to implement on embedded devices. More compact versions of the UNet, such as the C-UNet and C-UNet++ exists, which attempt to address these issues [9]. However, these networks tend to be less accurate than the original UNet [10, 11].

The compact C-UNet and C-UNet++ were explored in [10, 11] as a part of the FAUBAI research project between NTNU and European Space Agency (ESA), and found that compact UNets could be implemented on embedded devices. The same project also looks to explore the use of the original UNet architecture on embedded

satellite systems. Due to this, the work presented in this thesis will explore the use of large CNN networks, specifically the UNet, on embedded devices.

Previous work has shown that CNNs can be computed effectively using accelerated designs on FPGAs [12, 10, 11]. The work in [10, 11] achieves fully accelerated designs of compact UNets, however is only achievable due to these networks utilizing far less trainable parameters than the original UNet. Accelerating the entire UNet on the FPGA has been proven difficult due to a lack of resources in programmable logic. Therefore this thesis will look into a software-hardware codesign for an embedded UNet. This thesis will explore methods that can be used to fit CNNs resembling the original UNet on the multiprocessing UltraScale+ architecture with limited resources. The goal will be to achieve a design that can perform semantic segmentation of hyperspectral image cubes, such as those captured on HYPSO-1. Additionally, the thesis will seek to create a UNet framework usable within the HYPSO-1 and HYPSO-2 on-board processing systems. Finally, it will explore a accelerated design to support a software framework, hopefully supporting the FAUBAI research project in pushing forward for fast CNN networks on embedded devices.

1.1 Contributions

- A software framework for large CNNs within Linux-based embedded systems with limited memory resources, compatible with both ARM and x86 architectures.
- A software implementation of the UNet architecture capable of performing image segmentation on hyperspectral image cubes of varying spatial dimensions with a pixel prediction time around $295us$.
- A standardized weight format called Weights Interleaved by Filters (WIF), providing a structured way of storing trainable parameters for CNNs, which is fast to load into system memory.
- A suggested architectural design of a convolutional accelerator for 2D convolutional layers with upwards to 1024 kernels, each with a maximum depth 1024, aimed at the UltraScale+ architecture.

The code produced throughout this thesis is stored on the SmallSat GitHub repository. If the code is desired, or if there are any questions, please contact the author through email at *simennett@gmail.com*. Alternatively, send an email to a different member of the SmallSat team.

1.2 Thesis Structure

The thesis is structured into seven chapters, including the introduction, where the first explains background information, theory and current state-of-the-art. The three in the middle take the reader through the design and implementations of a UNet architecture, while the last two include results, discussion, conclusions and further work. The chapters can be described more precisely as follows:

Chapter 2, Background and Theory, introduces the HYPSO-1 mission in more detail before providing relevant theory around remote sensing, hyperspectral imaging, machine learning, convolutional neural networks, FPGAs and embedded Linux. Additionally, state-of-the-art is presented along with previous relevant work.

Chapter 3, High-Level Model/Design, describes the design and training of a CNN architecture inspired by the UNet with the high level programming language Python. The chapter also presents a suggested standardized format for storing weight values for CNN designs.

Chapter 4, Software Implementation, provides a walk-through of the suggested UNet software framework created for an embedded Linux environment, in addition to the design choices made.

Chapter 5, Hardware Acceleration, describes a suggested accelerated design for the convolutional layer, and how it could be implemented in a Linux system to support a UNet software framework.

Chapter 6, Results and Discussion, presents the results from Chapter 3, 4 and 5, and discusses the results for each of the chapters.

Chapter 7, Conclusion, presents the final conclusions drawn from the work, and is finished off with a section on further work.

2 Background and Theory

This chapter is meant to provide some insight into the background of this thesis, which is helpful in understanding the need for a CNN design on a satellite system. In addition, will relevant theory be described, which will be helpful in understanding the design choices presented in the implementation and discussion chapters.

This chapter starts by introducing the HYPSO-1 and HYPSO-2 CubeSats and their mission concept. This information is helpful, as it provides a better understanding of the need for on-board image processing, like image segmentation, and highlights challenges with on-board processing for nanosatellites.

The chapter also describes theory such as remote sensing and hyperspectral imaging, which are important subjects to grasp, as they provide insight into why this type of data is used in semantic segmentation. Furthermore, theory surrounding ML, DL, CNNs, FPGA acceleration and embedded Linux is provided to give a clearer understanding of the computational and memory requirements of a UNet architecture implementation.

2.1 HYPer-spectral Smallsat for ocean Observation

HYPSO-1 is the introductory nanosatellite designed and operated by the SmallSat team at NTNU. Its mission is to perform Earth Observation (EO) using a hyperspectral imaging payload and to perform in-orbit image processing, such as target detection and dimensionality reduction [13]. The satellite was launched with the SpaceX Transporter-3 mission from Cape Canaveral Space Force Station in January of 2022 [14] and is currently orbiting Earth at a low-Earth orbit of around 500km [2]. An additional satellite, HYPSO-2, is currently in its design phase and will hopefully be carrying an additional software-defined radio payload and improved on-board processing hardware into space in future years.

Figure 2.1 shows the Concept of Operation (CONOPS) for the HYPSO-1 mission, and describes the different stages of its EO mission. The satellite utilizes two main ground stations for data transmission, the main one being an S-band antenna at the

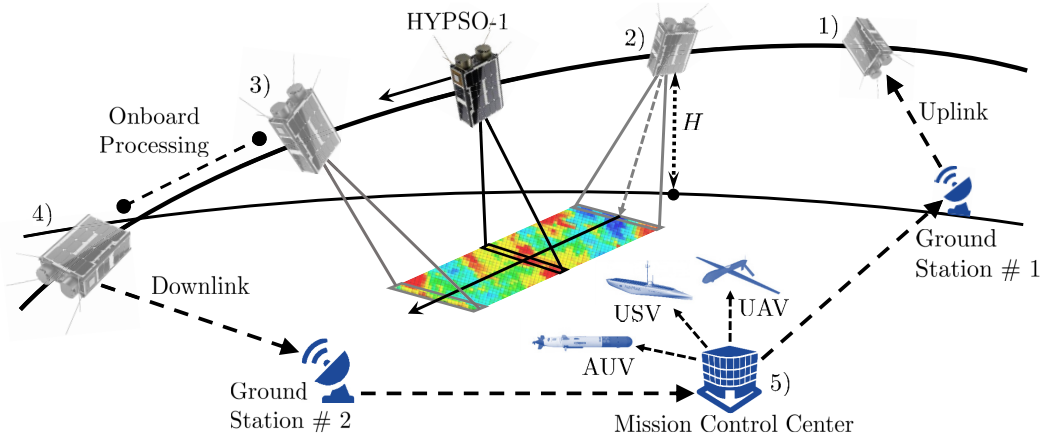


Fig. 2.1: HYPSO-1 CONOPS, where in 1) the satellite receives operational data; 2) HYPSO-1 starts a slew maneuver while performing a pushbroom scan; 3) capture completes and the slew maneuver ends; 4) in-orbit processing is performed and data is downlinked to a ground station [2].

roof of NTNU, whilst the other being the Kongsberg Satellite Services (KSAT) S-band base at Svalbard. Data cubes are compressed and binned to data sizes of around 40 ~70 MB, making it possible to downlink a cube within 2-3 passes, where an orbit takes around 80 minutes. As shown in the figure, HYPSO-1 will receive a target and autonomously orientate and capture the desired area using a slew manoeuvre while performing a pushbroom scanning technique. Section 2.2.2 describes this technique in more detail. Once complete, this data can be downlinked to one of two ground stations.

The HYPSO-1 mission looks to perform EO of the world's oceans, with the primary objective being to observe light-absorbing pigments, such as phytoplankton [2][p. 1]. These algal blooms, of which some are known to be harmful to humans and marine environments [15], may move considerable distances in a short time, impelling for EO missions like HYPSO-1 to allow for early detection. As Harmful Algal Bloom (HAB)s also will continue to arise in the future, with research indicating that they may even arise more frequently [16], it is vital to continue the research in the field, supported by satellites like HYPSO-1 and later HYPSO-2.

HABs, which are often visible on the surface of water bodies, have observable characteristics within wavelengths between 400 nm to 700 nm, commonly referred to as visible light [2]. Due to this, they can be observed using HYPSO-1s hyperspectral imaging payload, which has been designed to sample upwards to 1080 bands over this range with its image sensor.

2.2 Remote Sensing

Remote sensing is the concept of gathering information about an object, or scene, from afar, commonly through capturing electromagnetic waves that are either emitted or reflected by an object [1][p. 1]. In the book *Remote Sensing: Image Chain Approach* by J.R Schott, remote sensing is defined as

> *the field of study associated with extracting information about an object without coming into physical contact with it* [17].

Remote sensing is helpful as it allows the end-user, whether a human or a computer, to gather information from afar and, through analysing it, make decisions based on the sampled data [1]. Remote sensing is, however, not necessarily always meant for decision making and can be connected to more well-known cases, such as the capture of Red, Green and Blue (RGB) images, like the ones we see every day. For example, with the typical smartphone, a person can remotely capture a scene with its built-in camera. This produces what is called a multi-band image and is due to the camera sensor and optics being restricted to only capturing the bands of visible light perceivable by humans, such as red, green and blue (wavelengths of 650-, 550-, and 450-nm respectively.) [1][p. 4].

The wavelengths of red, green and blue in themselves do not provide any elaborate characteristic for the captured object except to provide a visual representation that humans can perceive. This is where hyperspectral imaging comes in and why it is valuable in EO scenarios, as it provides more bands, meaning more data about the captured object or scene.

2.2.1 Hyperspectral Remote Sensing

Hyperspectral remote sensing refers to the concept of performing remote sensing, though with the use of hyperspectral imaging sensors, such as the one illustrated in Figure 2.2. By utilizing various lenses and slits to diffract light to given locations on an imaging sensor, one can sample several bands of light for a given spatial area at a time. Depending on the imaging setup, a hyperspectral imager may be capable of capturing everything between several to thousands of bands.

Figure 2.2 shows how several bands correlate to a given spatial pixel in a captured scene. As one may see, as data accumulates, one gets a cube-like shape for the data. Due to this, hyperspectral data is often referred to as an image cube (spectral image cube in Figure 2.2) since it contains both spatial and spectral dimensions.

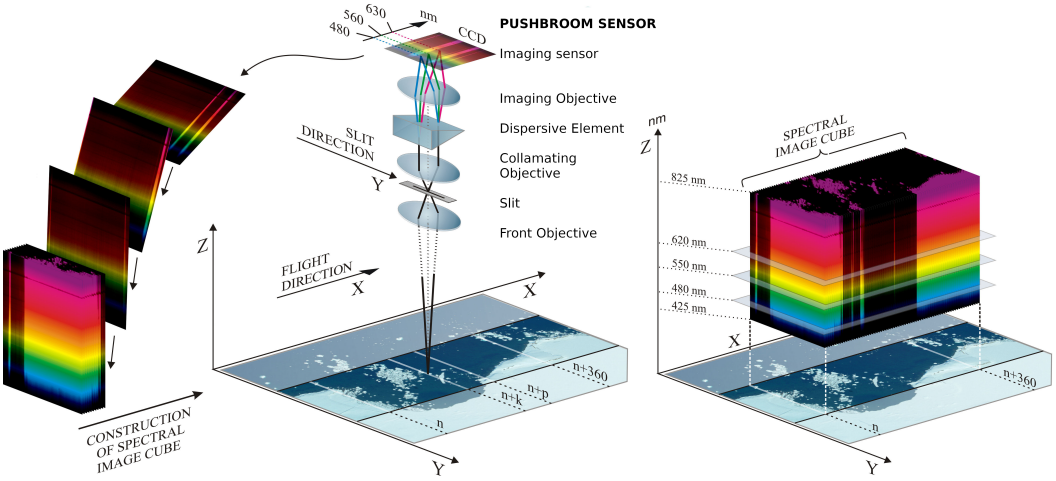


Fig. 2.2: Illustration of how a pushbroom scanning maps to a image sensor through various lenses and slits, and further into an spectral image cube. The figure also shows how different bands map to a given spatial locations on the right hand side. Adapted from illustration by Zsolt Volent with permission [18].

Through sampling several bands for a given pixel, it is possible to visualize and analyse the spectral characteristics of a spatial area $[X, Y]$ as shown in the right-hand part of Figure 2.2. The spectral dimension is denoted as Z. Comparing this with known characteristics allows for determining the materials or compounds, such as algae, in the given area [1][p. 8]. This is also the reason why hyperspectral imaging has shown to be advantageous for training neural networks [7, 6].

2.2.2 Pushbroom Scanning

Standard cameras, e.g., those used in smartphones, utilize sensors with separate photodiodes for each of the RGB wavelengths per pixel, allowing these sensors to sample the intensity of red, green and blue simultaneously. However, this is not the case with hyperspectral imaging sensors since having hundreds of specific diodes per pixel would not be viable. There are hyperspectral snapshot designs capable of capturing all bands for a spatial area within the same Δt; however, this technology often utilizes detector arrays that are more complex than what is used in, e.g., HYPSO-1 [19].

Two more commonly used methods for hyperspectral imaging are the pushbroom and whiskbroom scanning methods [1][p. 255-247], which involve “scanning” the surface that is to be captured.

Pushbroom scanning, as seen in Figure 2.3, is a technique that samples all bands for a spatial line at a time while moving over its target surface. As the satellite moves along its track, denoted as N_x in the figure, a frame is captured every Δt. The resulting dimensions of the image cube will therefore be defined by the number of pixels used for the cross-track, also called swath width, the number of frames captured, and the number of pixels used for the bands [1][p. 255-247]. Figure 2.4 shows how a spatial line maps to the image sensor for every Δt. Additionally, since pushbroom scanning utilizes optics for diffracting the light onto specific locations on the image sensor, it is common to utilize monochromatic sensors, which means that it is receptive to all wavelengths of light [1][p. 315].

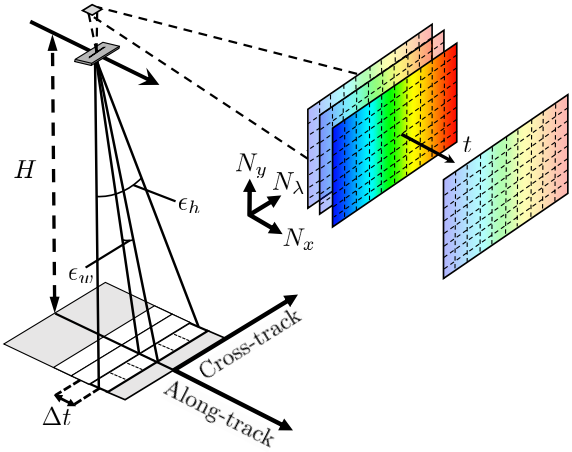


Fig. 2.3: Illustration of pushbroom scanning, where samples with their respective bands are captured per N_x. The Cross-track defines the number of samples N_y captured per frame, whilst N_λ defines the number of bands captured [2][p. 3].

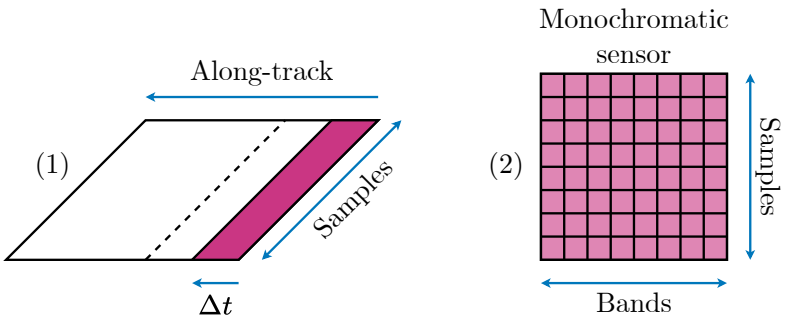


Fig. 2.4: Illustration of how a line of pixels (1) during pushbroom scanning maps to the monochromatic image sensor (2) for each sample of the sensor.

When working with hyperspectral image cubes, it is easy to confuse which spatial dimension refers to the frames and which refers to the swath width. Therefore, a more precise naming scheme will be used throughout the thesis and is described in Figure 2.5. In the figure, *Samples* refers to the spatial line captured for every Δt, while the *Lines* represent the number of frames. Additionally, the *Bands* logically represent the bands for each spatial location defined by the lines and samples. For reference, in Figure 2.2, the X denotes the lines, Y the samples and Z the bands.

2.2.3 Raster Formats for Hyperspectral Data

Raster files, or formats, are digital descriptions of image files and can be recognised through file formats such as JPEG, PNG and GIF. Compared to vector files, which utilise vectors to define pixel values, raster files have fixed pixel values. Due to this, if transformed with various processing techniques, they may become distorted and blurry, while vector data does not. Since hyperspectral data is sampled using an imaging sensor, this data is stored as fixed pixel values.

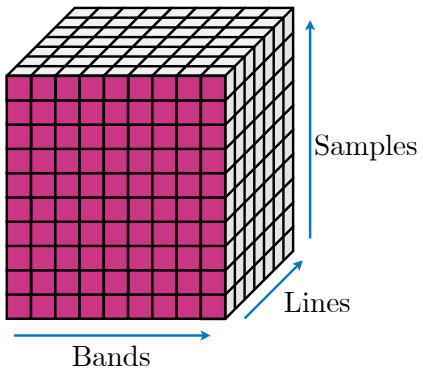


Fig. 2.5: Illustration of the naming schemes for hyperspectral image data.

Since hyperspectral images consist of several sampled bands for their spatial pixels, they cannot be fully visualized as images, as humans only see red, green and blue. However, hyperspectral data is often visualized through vivid images, where three bands are extracted to form an RGB representation.

Though they are not in themselves image formats, hyperspectral data is often encoded using three different encoding methods called Band SeQuential (BSQ), Band Interleaved by Pixel (BIP) and Band Interleaved by Line (BIL). These formats essentially structure the data in various ways, such as the BIP which structures all bands for a given pixel before moving on to the next as shown in Figure 2.6. When data is captured with BIP encoding during a pushbroom scan, the data will be stored in the order of bands, samples, lines.

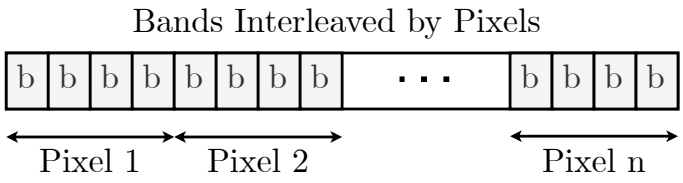


Fig. 2.6: Image data array encoded using the BIP format.

2.3 Deep Learning

Machine Learning (ML) is a branch within the field of Artificial Intelligence (AI) and computer science that focuses on mimicking the human learning behaviour and decision making through training statistical models on various input data. A ML algorithm is described in the book *What is Machine Learning* by El Naqa I and Murphy M as:

> *A machine learning algorithm is a computational process that uses input data to achieve a desired task without being literally programmed (i.e., "hard coded") to produce a particular outcome [20][p. 18].*

Programming the algorithms so that they may iterate on input data and "optimize" their variables accordingly allows these models to find patterns in data and perform predictions better than hard-coding them for each scenario, significantly reducing the programming needed.

Deep Learning (DL) takes the concept of ML a step further by attempting to mimic the human brain's functionality. As Figure 2.7 suggests, DL is a subfield of ML, and is a field that focuses on the training of multilayered learning algorithms. The purpose of DL is to create multiple levels of abstractions from input data, which allows more features within the data to be extracted [5][p. 1]. DL has proven to be highly effective for use cases such as speech recognition, visual object recognition and object detection, as it manages to extract more intricate details from larger datasets [8][p. 1]. Due to the core component of DL being neural networks, it is common to use the name Deep Learning synonymously with Artificial Neural Network (ANN) and will also be done so throughout this thesis.

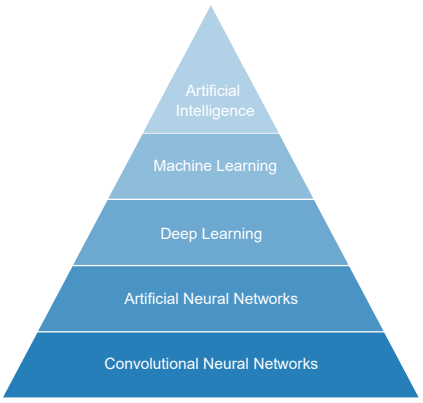


Fig. 2.7: A illustrating of how the the field of AI is structured for a specific branch within Deep Learning all the way down.

2.3.1 Artifical Neural Networks

ANNs are, as mentioned in the previous section, networks that attempt to resemble the way human brains work for processing data. The core component of the ANN is the neuron, which can be added together to form various layers as illustrated in Figure 2.8. The figure shows the inputs passing into various neurons, which are all densely connected to a network, before being passed out to the outputs. Each of these neurons performs a computation on its input data before passing it to the next layer. The neurons, which are not a part of the input and output, are commonly called hidden units, and together from the hidden layer [8][p. 3].

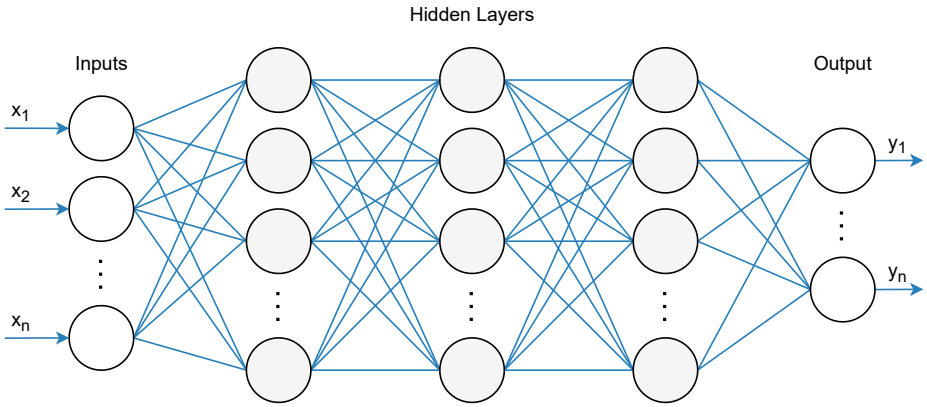


Fig. 2.8: Illustration of a dense ANN with n number of inputs, layers and outputs. The direction of the data is from left to right, making it a feed-forward network. Each circle corresponds to a neuron, and each blue line is its connections.

The neuron can be illustrated as shown in Figure 2.9. One can see that the neuron consists of a variable number of inputs, as previously shown in Figure 2.8, where each input x is multiplied by its respective weight w. The weight values are called trainable parameters in a ANN, as they, in addition to the bias value b, are tuned during the training process. Training is further described in Section 2.3.2.

The output of each weight multiplied by its input is summed together with the additional bias b value. The bias is another trainable parameter that can be tuned to either increase or decrease the chance of a neuron "firing". Once all values are summed together, the final sum is passed through an activation function $f(x)$ which checks whether or not the sum passes the threshold of the neuron being activated, as follows:

$$y = f(\sum_{n} x_n w_n + b) \tag{2.1}$$

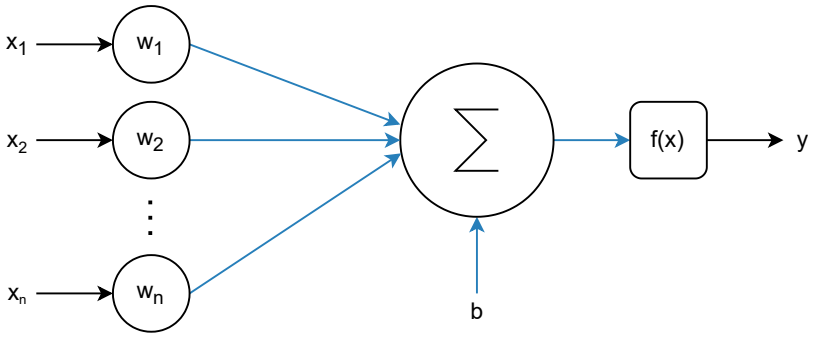


Fig. 2.9: Illustration of neuron used in ANNs, where w denotes the weights, b denotes the bias and $f(x)$ denotes the activation function.

Activation Functions

Activation functions are nonlinear functions that determine whether or not a neuron will activate. They are important for DL networks as they provide a collection of neurons with the ability to model nonlinear patterns in data [20][p. 212]. Many different activation functions are used within the field of DL; however, this thesis will focus on two, in particular, namely the Rectified Linear Unit (ReLU) and Sigmoid activation functions. The ReLU function can be seen in equation 2.2, and essentially removes negative input values, while passing positive ones. Figure 2.10 shows an illustration of the ReLU behaviour.

$$f(x) = \begin{cases} x & \text{if } a > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

The Sigmoid function is described in equation 2.3 and is an un-linear function that always produces a value between 0 and 1. The function is usually used in the last layers of a ANN network to predict the probability of various outputs. Figure 2.10 illustrates how the Sigmoid function $f(x)$ behaves according to different input data x:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

2.3.2 Network Training

ANNs like the ones described in Figure 2.8 can consist of anywhere between hundreds to millions of neurons depending on the complexity of the network.

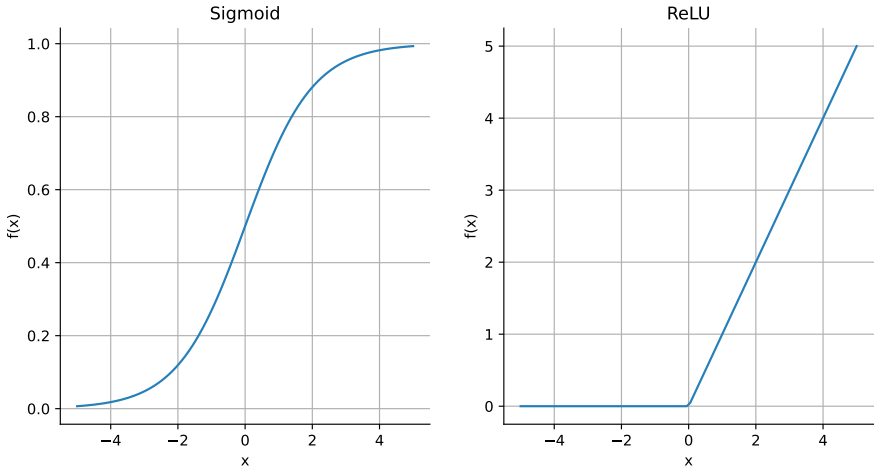


Fig. 2.10: Sigmoid (left) and ReLU (right) activation functions. The x-axis is the input value, while the $f(x)$-axis is the output.

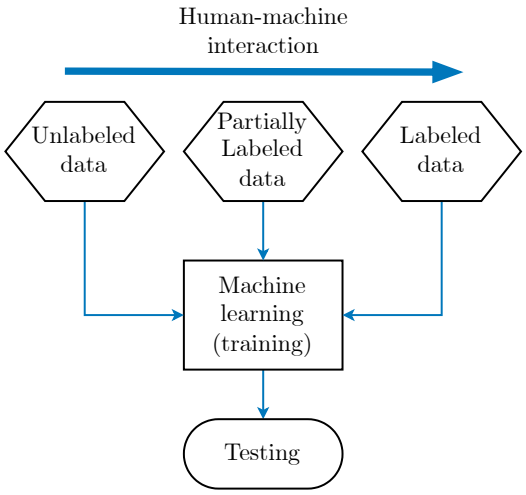


Fig. 2.11: Illustration of different methods for training ML algorithms. Adapted from [20][p. 7].

For these neurons to perform predictions on data, their parameters often referred to as hyperparameters, must be trained on similar data to what the network will be predicting. The process of tuning the hyperparameters is what is referred to as training the neural network. It can briefly be described as an iterative procedure where a computed output is compared to the desired values for a given input, and weights are changed accordingly. This process of training ML algorithms aims to emulate the way humans learn where one trains through repetitive encounters with specific tasks. The final goal of this process is to repeat the training to a degree such that the algorithm can classify unseen data [20][p. 4].

A common methodology for training neural networks is called *supervised learning*, in which an algorithm is passed an input and its expected classification (truth values). Figure 2.11 describes an additional two types of learning methods, namely *Semi-supervised learning* and *Unsupervised learning*. The first method trains on partly labelled data which may aid in classifying other parts, while the latter lets the

algorithm attempt to find the path for itself [20][p. 5]. For image segmentation, the *semi-supervised-* and *supervised learning* methods are the most common and, therefore, most relevant to this thesis.

2.3.3 Backpropagation

The procedure for tuning ANNs requires an algorithm capable of calculating changes to the weights backwards in the network. The procedure used for training a feed-forward ANN is called *Backpropagation*. It involves calculating an error function, also called loss or cost functions, and finding the gradient of the error whilst moving backwards (backpropagating) in the network. As mentioned previously in section 2.3.2 about *Network Training*, multi-layered networks like ANNs consist of large amounts of neurons, where each of these neurons contains a variable amount of weights. Weights are commonly real numbers represented as floating-point values with varying precision. Due to the vast number of weights requiring tuning, training neural networks may take a very long time depending on the network architecture, weight value precision and training hardware. However, Chapter 3 details further the caveats of arbitrary floating-point precision and network architectures. The tuning of weights can essentially be boiled down to changing the value of a floating-point number within the range of 0 and 1. [21, 5].

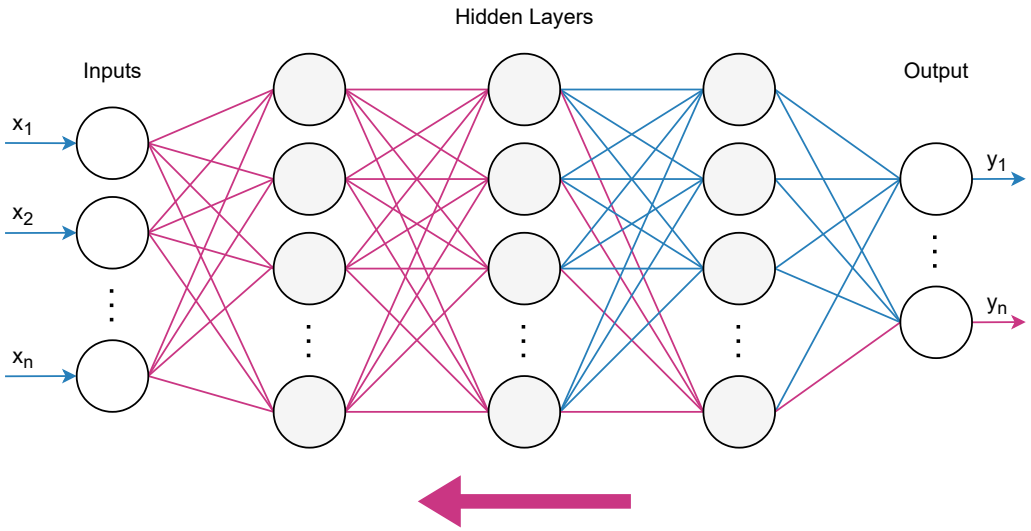


Fig. 2.12: Illustration of how backpropagation is used to look at the propagation of error for all weight values backwards in a feed-forward neural network. Pink lines represents a singular backpropagation while moving towards the left.

Figure 2.12 attempts to illustrate the connections between neurons and how error propagates backwards in a network. As the figure clearly shows, the number of

weights involved for a given output quickly becomes numerous as one moves past the last layers. Due to this, the denser the neural network is, the more computationally intensive the training procedure becomes, assuming a fully connected network.

Several loss functions exist for neural networks, which all have different pros and cons depending on the type of architecture. However, these will not all be mentioned in this thesis as they are irrelevant. One of the more common loss functions in ML is the cross-entropy loss function given by equation 2.4. The loss denoted as E can be calculated for a output $\widehat{y}_i$ and the desired output y_i (truth values) [22].

$$E = -\sum_i y_i log(\widehat{y}_i) \tag{2.4}$$

Cross-entropy entropy has become popular due to its increased training speeds and ability to generalize networks. Additionally, the loss function can be used for binary and multi-class classification, making it suitable for image segmentation and classification networks [22, 8].

Though loss functions like cross-entropy provide a sense of how far off a prediction was, it does not directly indicate how each weight value throughout the network should be adjusted. This is where the use of gradient descent comes into play.

Gradient descent is used with loss functions for calculating changes to weights, the "direction" and magnitude, which provides the highest reduction to the loss function E. The direction is derived from the mathematical operation of the loss functions gradient, which calculates the steepness of a slope for a given point. Gradient descent can be defined as in equation 2.5.

$$w^{\tau+1} = w^{\tau} - \eta \nabla E_n \tag{2.5}$$

In the equation, τ denotes the iteration number, η the learning rate, w the parameter vector and ∇E_n the gradient of the loss function. For each iteration, a new parameter vector is generated by subtracting the previous parameter vector with the gradient of the loss function multiplied by the learning rate. Learning rate η defines how much weight adjustment is performed each iteration, previously mentioned as the magnitude [23][p. 144].

As with loss functions, different ways of calculating gradient descent exist for training neural networks. One of the most popular ones is the Adaptive Moment Estimation optimizer, called Adam. The Adam optimizer utilizes dynamic learning rates for the parameters, proving to be competitive with other optimization methods [24].

2.4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of DL networks which are designed to process image arrays, such as a RGB or hyperspectral image cubes [5], which the goal of classifying or segmenting parts of the image. These types of networks are often structured as a set of convolutional and pooling layers in which the image data is passed through to extract features. Semantic segmentation, which is the goal in this thesis, is essentially the prediction task of "painting" a image with its classes. The previously mentioned UNet is a commonly used CNN network, and can be seen in Figure 2.29. As the figure shows, a CNN like the UNet utilizes 4 distinctive layers, namely *convolution*, *max pooling*, *transposed convolution* and *concatination*.

2.4.1 Convolution

The convolutional operation is the core of the CNN and is used to extract features from image matrices using kernels. The name implies that the CNN utilizes the mathematical operation of convolution. However, most CNN networks generally implement the related cross-correlation function, as seen in Equation 2.6. This function performs the same operation as convolution, though without flipping the kernels [25][p. 327]. For simplicity, the operation will be referred to as convolution throughout this thesis.

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, k+n) K(m,n) \tag{2.6}$$

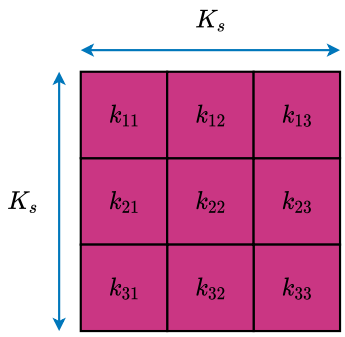


Fig. 2.13: Convolutional kernel of size 3×3.

The kernels used in convolutional layers are 2D matrices, as seen in Figure 2.13. The kernel size, denoted as K_s, defines the width and height of the kernel. In a convolutional layer, these kernels are passed over the image data multiplying element wise its weights with the image matrix, as described in Figure 2.14. This essentially means that the K_s defines the number of pixels that will be included in a convolution. The output of this operation is placed into a new matrix, as seen on the right hand side of Figure 2.14. Depending on the configuration of the kernel, e.g., the K_s and stride, the output will vary in size [25][p. 330].

For the example in Figure 2.14, $K_s = 3$ and a stride of 1 is used. This means the kernel moves one pixel horizontally and vertically over the image matrix while convolving. For each position of the kernel window, which in this case has four possible positions, the kernel values are multiplied with the pixels in the image as described in Equation 2.7. The f in the equation is one of the activation functions described in Equation 2.2 and 2.3. The element-wise multiplied pixels and weights are summed together before passing through the respective activation function, which defines the value within the outputted feature map.

$$o_{22} = f(k_{11}p_{22} + k_{12}p_{23} + k_{13}p_{24} + ... + k_{31}p_{44}) \tag{2.7}$$

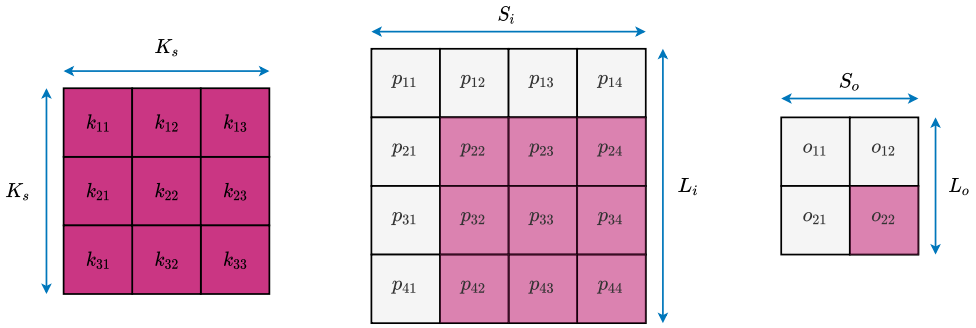


Fig. 2.14: Kernel sliding over image matrix performing cross-correlation with a stride of 1. The naming scheme of samples and lines is used to denote the dimensions of the image and output matrices. S denotes the samples, while L denotes the lines.

By changing the magnitude of the kernel weights, different feature maps can be computed and is the approach used by a neural network when training.

CNNs like the UNet utilize what's called 2D convolutional layers. This essentially specifies that a kernel only moves in a 2D space and always calculates a 2D output as seen in Figure 2.14. Therefore, in the case where a kernel is subjected to a 3D input, it will have an additional depth dimension K_d corresponding to the depth of its input. This is visualized in Figure 2.15. The same logic for convolving a 2 dimensional kernel applies to a 3 dimensional kernel, meaning weights are multiplied element-wise to the input cube, and summed together.

It's typical to have several kernels within a convolutional layer, which results in its output being a 3D feature map. Figure 2.15 attempts to further visualize this relationship between input dimensions, kernel number and layer output for a 2D convolutional layer. Kernel number is denoted as K_n.

Convolutional layers have a tendency to reduce the spatial dimensions of its inputs depending on the kernel size and stride. In the case where spatial dimensions should

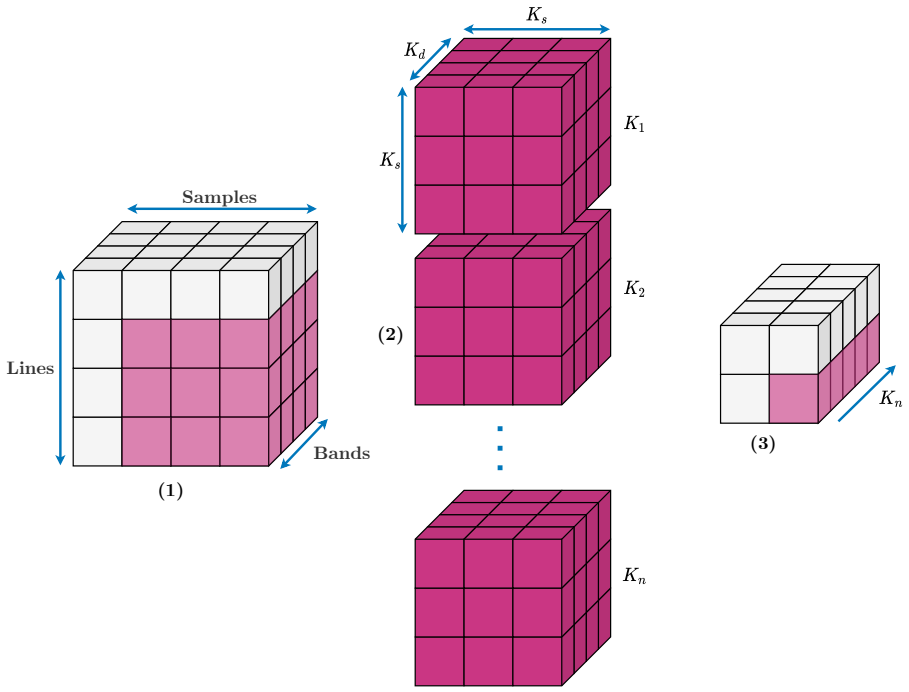


Fig. 2.15: Illustration of a convolutional layer in a CNN, consisting of K_n kernels. Each kernel (2) have the same depth as the input (1), while the output (3) will have the same depth as there are kernels k_n

be maintained through a convolutional layer, image data is padded before being subjected to the layer. A standard method is zero padding, which essentially places zeros around the image cube as seen in Figure 2.16.

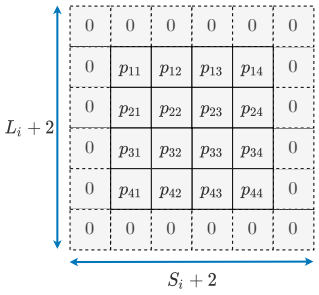


Fig. 2.16: Illustration of how padding is applied to a input to maintain spatial dimensions through a convolutional layer.

2.4.2 Max Pooling

Pooling is a CNN layer used to summarize and downscale features from previous convolutional layers [25][p. 335]. In the UNet described in Figure 2.29, the max-pooling layer is used, which finds the highest value within a window and places this into an output feature map as visualized in Figure 2.17.

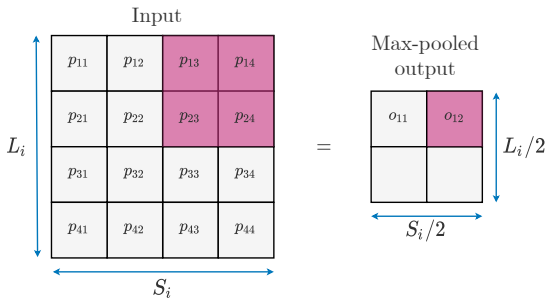


Fig. 2.17: Illustration of a max-pooling operation with a 2×2 window and a stride of 2.

As an example, the o_{12} output can be calculated as shown in Equation 2.8. A max-pooling layer with a window size of 2×2 and a stride of 2 will downscale both spatial dimensions of the input by half, though retaining its depth.

$$o_{12} = max(p_{13}, p_{14}, p_{23}, p_{24}) \tag{2.8}$$

2.4.3 Concatenation

The concatenation layer takes two feature maps of equal spatial dimensions and concatenates the bands into a new feature map, as seen in Figure 2.18. In Figure 2.29, concatenation, also referred to as skip connections are performed during the upscaling (decoding) path of the network. This brings features that have not been max-pooled forward into the network, merging with upscaled features.

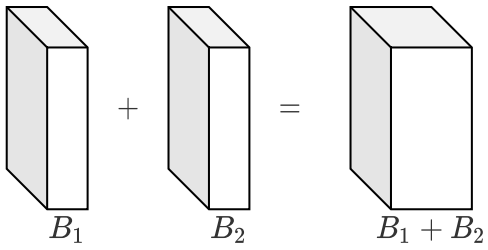


Fig. 2.18: Illustration of a concatenating layer combining two feature maps into one feature map of their combined depth.

2.4.4 Transposed Convolution

Transposed convolution is a method of upscaling input features with the goal of increasing the spatial dimensionality of the input [26]. Various methods of upscaling exist, such as nearest neighbour and max-unpooling. However, in contrast to these, the transposed convolutional operation utilizes trained weights to perform the upscaling of input features more accurately.

Much as with the convolutional layer, transposed convolution does not perform the actual mathematical operation of transposed convolution and is therefore often referred to as deconvolution [26, 27]. The transposed convolutional operation works much like the convolutional operation described in Subsection 2.4.1, and can be seen in Figure 2.19. The main difference is that the transposed convolution works the "opposite way" to the convolutional operation. The kernel is multiplied with each input value, creating a 2×2 representation shown on the right-hand side. The kernel slides over the output feature map, in this case with a stride of 2, meaning the output feature map will have twice the spatial dimensions as its input.

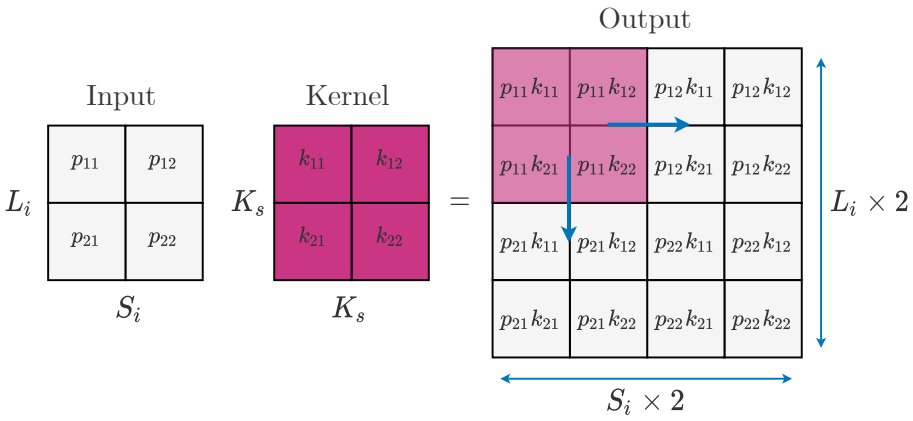


Fig. 2.19: Illustration of a transposed convolutional operation. The kernel has a size 2×2 and a stride of 2. The pink square in the *Output* represents one instance of the kernel sliding over the output.

If the stride is not equal to K_s, the kernel multiplications in the output overlap and are therefore summed together. Additionally, as shown in Subsection 2.4.1, the depth of the output feature depends on the number of kernels K_n in the layer.

2.5 UltraScale Architecture

The UltraScale Multi-Processing System on Chip (MPSoC) architecture is a SoC designed by Xilinx, enabling diverse multiprocessing capabilities for embedded systems. The architecture, as described in Figure 2.20, includes a Application specific Processing Unit (APU), Real-time Processing Unit (RPU), Graphics Processing Unit (GPU) and FPGA within the same chip [28].

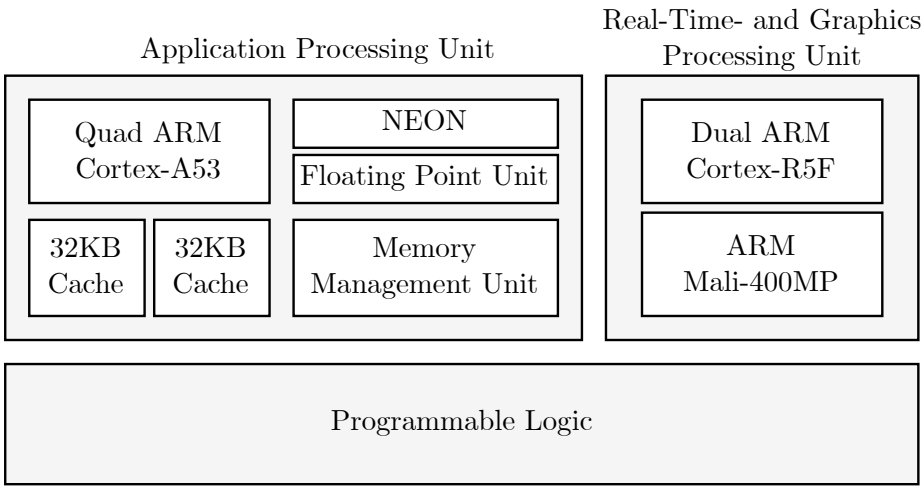


Fig. 2.20: Simplified UltraScale MPSoC architecture describing the processing units within the SoC.

The APU, typically referred to as a CPU throughout the thesis, includes a quad-core ARM Cortex-A53 processing system running upwards to 1.5GHz, and supporting both 32- and 64-bit operation. The Cortex-A53 implements the ARMv8-A ISA, which includes ARMs NEON technology, capable of Single Instruction Multiple Data (SIMD) instructions. Additionally, the Cortex-A53 includes a Floating Point Unit (FPU), providing more efficient floating-point arithmetic, and the possibility for floating-point SIMD operations [28, 29].

The Programmable Logic (PL) consists of a FPGA with 504K logic cells and 1727 DSP slices capable of running with a clock speed of upwards to 300MHz. Additionally, the FPGA has 38MB of BRAM, making it possible to store data within the FPGA for faster computing.

In conjunction with the APU and PL, the ultrascale architecture also includes a RPU for applications with real-time constraints, and a GPU meant for mobile graphics and user interfaces [28].

2.6 Field Programmable Gate Arrays

Embedded systems are progressing in processing power versus power consumption, which is especially useful in space applications. However, this progression is heavily supported by the rise of accelerators, such as FPGAs. Due to satellites, especially CubeSats, having less powerful SOMs, the use of FPGAs allows for heavy computations within limited time windows without utilizing huge amounts of power. This makes FPGAs very useful for CNNs, which are notorious for their high computational loads. With the use of different processing units such as the CPU and FPGA, the terminology *Hardware/Software Codesign* is introduced.

The Field Programmable Gate Array (FPGA) is a generic logic circuit that was designed to be programmed for specific applications [30][p. 21]. Its main strength is its ability for parallelism, in contrast to the sequential execution of processors, thus making it a popular choice for accelerating parts of sequential software. In many ways, the FPGA looks and functions like a Application-Specific Integrated Circuit (ASIC), however it consists of a tightly interconnected network of generic logic cells consisting of LookUp Table (LUT)s and latches, as seen in Figure 2.21 [30][p. 26].

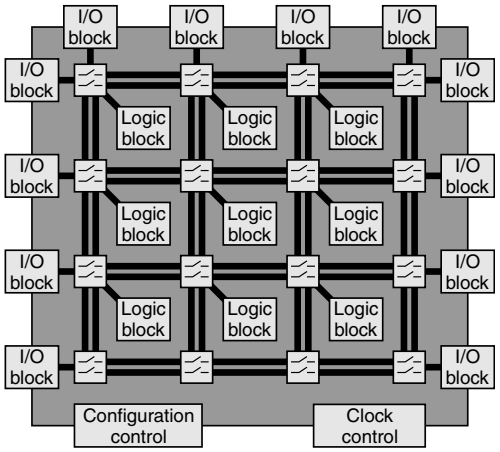


Fig. 2.21: The architecture of a FPGA illustrated in a simplified manner. Figure taken from [30][p. 26].

LUTs are essentially programmable logic blocks, making it possible to cascade them together to perform various processing tasks. Additionally, a FPGA runs on a set clock frequency, making it more deterministic than processor systems which often rely on multi-threading for simulating concurrency. This makes the FPGA very useful in settings where determinism is essential, such as real-time systems, and is often used in conjunction with processing systems.

The FPGA is often compared with the well-known Graphics Processing Unit (GPU), which essentially is a graphical processing accelerator, due to them both relying on the concept of parallelism. However, what the FPGA lacks in generalization compared to the GPU, it makes up in significantly decreased power consumption. Due to this, the FPGA has become highly popular in cases where a tradeoff between power consumption and processing power is needed, such as on satellites, phones, and drones [9].

The points above have also made the FPGA very popular in the field of Deep Learning, as neural networks consist of large amounts of multiplications and additions, which are generally parallelizable. The FPGA have even been proven to be more efficient than the GPU in some instances, such as sliding window operations which are widely used in image processing and CNNs [31].

2.6.1 Interface

FPGAs are often used together with other peripherals and processing units, which all need to interface with the FPGA. These interfaces are used to either deliver data for processing or are simply control signals for the logic implemented on the FPGA. It is possible to find standalone FPGAs; however, today, it has become common to include them in multiprocessing System on Chip (SoC)s where specialized interfaces communicate between, e.g., CPUs and FPGAs.

One such interface is the Advanced eXtensible Interface (AXI) protocol designed by ARM, mainly designed for on-chip communication. The protocol, which is part of the Advanced Microcontroller Bus Architecture (AMBA), makes it relatively simple to connect several devices through an interconnection device. Additionally, the AXI4 protocol includes various side-channel communication for handshake synchronization, making it capable of high data throughput and control signalling [32].

Figure 2.22 shows how a master device can connect to a slave device through the AXI interconnect. The figure shows that the interconnect behaves like a slave and master in itself, being able to “command” slave devices whilst also being “commanded” by the master device. In a multiprocessing system, the master and slave could be, e.g., a CPU and a FPGA.

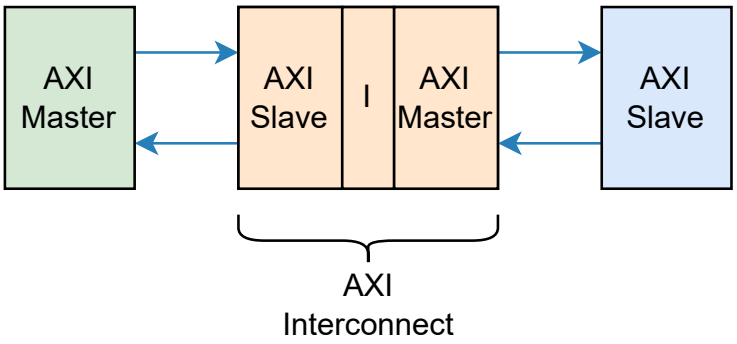


Fig. 2.22: Illustration of the AXI protocol, showing how a master can communicate with a slave device through a AXI interconnection [32].

2.6.2 On-Chip Memory

One of the main bottlenecks in high throughput processing on FPGAs is the memory bandwidth between the device and system memory. As FPGAs lack large storage capacity on-chip, it is often required to stream data to the device from, e.g., Random Access Memory (RAM). Due to bandwidth limitations in streaming data between devices, a FPGA design might not be parallelized as much as wanted. However, smaller portions of data can be stored on-chip using Block Random Access Memory (BRAM) [30][p. 400].

2.6.3 Fixed Point Precision

Fixed point precision is a term used when talking about fractional numbers (non-integers) in computers and touches upon the concept of balancing precision versus range for floating-point numbers. Fixed point precision refers to decimal numbers with a fixed exponent and mantissa, meaning there is a fixed number of digits after the radix point. Commonly, software running on modern systems utilizing 32- and 64-bit architectures rely on single and double precision data types for working with decimal values; however, these are often floating-point data. Compared to fixed point, a floating-point may have a varying number of digits after the radix point, providing more flexibility as a tradeoff for performance. In most programming languages, single and double precision refers to the data types *float* and *double*, which are 32- and 64-bit, respectively.

Figure 2.23 shows how a single-precision floating-point number is divided into an exponent and mantissa in a register. This structure is due to floating-point values relying on scientific notation, e.g., $1.234 * 10^3$, to represent different fractional values;

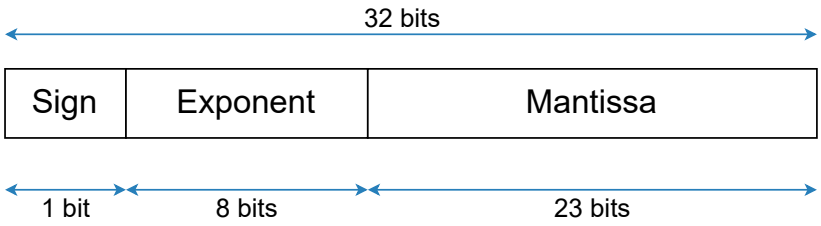


Fig. 2.23: Illustration of how a *float* number is stored in a register using the IEEE standard for floating point arithmetic.

the same goes for fixed point. This makes it possible to represent an extensive range of numbers without needing hundreds of bits. Additionally, the sign bit specifies whether the value is positive or negative.

The exponent is often regarded as the integer part in a floating-point value, whilst the mantissa is the value to the right of the decimal point. However, it is more sensible to consider the exponent as the part defining the values range, whilst the mantissa defines the precision. Due to this, if one were to increase the number of bits for the exponent while decreasing the bit count for the mantissa, one would sacrifice some fractional precision for more dynamic range [33].

Fixed point precision is commonly used when computing data where one knows the acceptable balance between precision and range. As fixed-point numbers can be any total bit length, they can use less memory and be less computationally heavy, thus increasing performance. They can also be computed faster on architectures which do not have floating-point arithmetic logic.

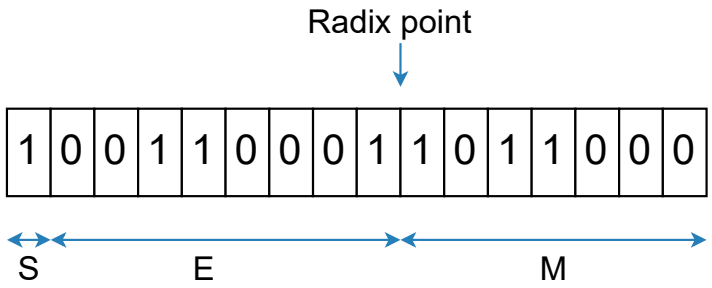


Fig. 2.24: Illustration of the BFloat(16) fixed point number where the sign bit is denoted as S, the exponent as E and the mantissa as M.

Fixed-point precision is often used in hardware accelerators as it decreased the resources needed for arithmetic operations, and can as a result provide higher speedup.

Brain Floating Point

Brain Floating Point (BFloat16) is a half-precision (16-bit) format created for Deep Learning by Google Brain [34]. It's structure can be seen in Figure 2.24 where $E = 8$ and $M = 7$. This format aims to balance range and precision with DL training in mind. The BFloat16 format, which has the same dynamic range as the IEEE standard for single precision, has been shown to provide a good balance between range and precision for neural networks [35].

2.7 Linux

Linux is a family of OSs that is very popular in the world of embedded devices due to being open-source and resource-efficient. Though many different distributions exist, they all rely on the Linux kernel. The Linux kernel acts as an abstraction level between the space where applications run, often called *User Space*, and the system hardware. These abstraction levels are illustrated in Figure 2.25. If the user-space requires access to hardware components, such as memory or peripherals, they can do this through system calls to the kernel. The kernel is also responsible for scheduling and other OS operations and provides better security in the system [36][p. 8-9]. E.g., a user-space application cannot as simply cause system crashes.

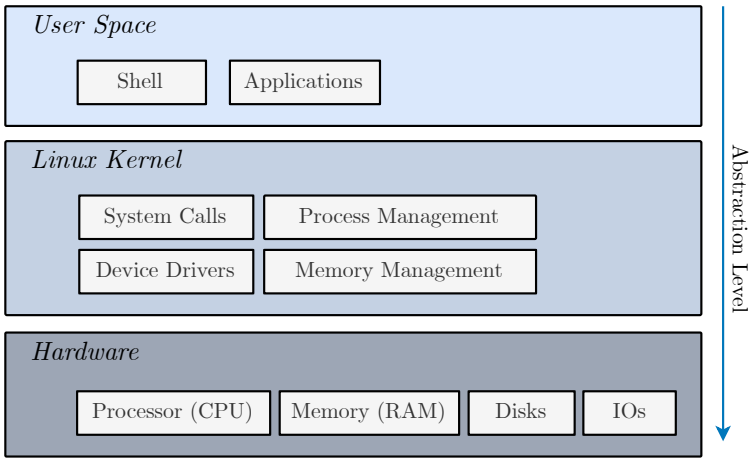


Fig. 2.25: Illustration of the different abstraction levels in a device running a Linux OS. The *User Space* is the highest abstraction level, whilst the *Hardware* is the lowest. The figure is adapted from [36][p. 9]

2.7.1 System Calls and C Library Functions

When applications within the *User Space* want to perform operations such as read and write on files within the Linux system, they need to perform system calls. These operations can, however, be expensive, as they make the Linux system perform context switches to kernel threads. The C library implements more efficient ways to perform system calls in commands that bundle operations together. Some of the more relevant are:

- *fopen()* and *fclose()* opens and closes files.
- *fscanf()* reads formatted input stream.

- *fread()* and *fwrite()* reads and writes from/to files.
- *malloc()* allocates memory.
- *free()* frees allocated memory.

These C library functions allow low-level control over system resources and files through bundled system calls while still providing a higher abstraction level from the *Kernel Space*.

2.7.2 Devices and Address Space

The way hardware is accessible to the kernel space, or *Linux Kernel* in Figure 2.25, is through a memory map describing the physical addresses for various IOs and memory devices [36][p. 11].

Depending on the system, the address space can have different ranges and consist of more than one address space. E.g., the UltraScale architecture utilizes a main address space of 32-bit, though it has two additional address supersets of 36- and 40-bits [37]. Figure 2.26 illustrates what a 32-bit address space could look like, where 2GB of RAM is defined first, and various devices are mapped throughout the address space.

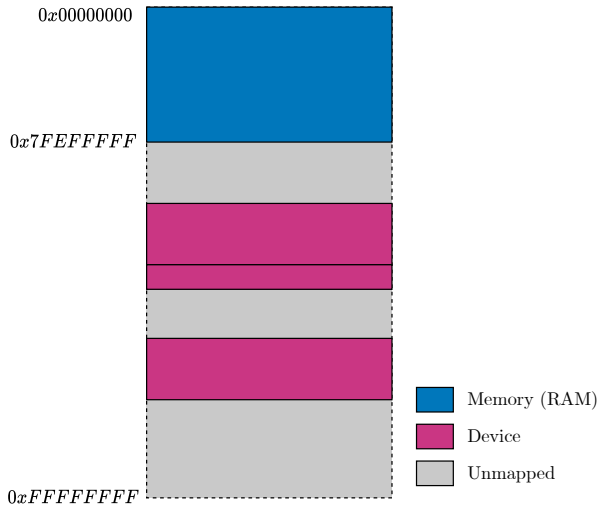


Fig. 2.26: Illustration of a 32-bit address map.

The *Linux Kernel* can control hardware devices by modifying the bits within their respective address offsets. However, modern CPUs, such as the ARM Cortex-53, include Memory Management Unit (MMU)s, which allows for a memory access method called *virtual memory*. *Virtual memory* allows processes within the system to act as if they have access to the entire system, though when they, in reality, have to go through the MMU to access a physical address. This

outsources a lot of the memory and caching operations for the CPU over to the MMU [36][p. 11].

2.7.3 Compiler Optimizations

Most software written for Linux is coded in either C or C++, as these languages provide the low-level control desired when working with embedded systems [38][p. 5]. When working with general-purpose code like C and C++, compilers are necessary tools for converting the source code into executables/binaries for a given architecture. Intel and ARM processors utilize different types of Instruction Set Architectures (ISAs), whereas Intel uses the x86 architecture and ARM the less resource-intensive Reduced Instruction Set Computer (RISC) architecture. An ISA can shortly be explained as an abstract model defining how a processing system, e.g., a CPU, handles data and computations. Some of these processing system vendors, e.g., Intel, provide specific compilers for their CPUs. However, a more generalized compiler like the GCC is often easier to use while also providing cross-compilation compatibility with other architectures like RISC [38][p. 3-4].

Compilers like GCC provide powerful optimizations for generated files, with the aim of either increasing speed or reducing file size [38][p. 45]. Some types of optimizations will increase the speed of the application, though with the cost of size, and vice versa, and is therefore referred to as the *speed-space tradeoff* [38][p. 47]. Depending on the application, binary sizes could be of concern, such as during uplinking data to a satellite.

Some standard optimizations are subexpression elimination and function inlining, which modifies code to reuse variables and function calls in more optimized ways. Additionally, more advanced optimizations such as loop-unrolling, scheduling and pipelining can provide even more significant speedup, though at the cost of size [38][p. 45-49].

These optimizations can be accessed by utilizing certain flags during the compilation procedure. However, many optimizations have been collected into so-called optimization levels, as seen in Table 2.1. Using various optimization flags will call a bundle of optimizations fitting the level [38][p. 50]. Although, some optimizations such as *-funroll-loops* are not included within any of the *-O* flags. Depending on the target architecture, various optimizations depend on the available hardware within the said architecture. E.g., for ARM processors such as the ARM Cortex-A53, the *-ftree-vectorize* flag will be enabled when using the *-O3* flag, which will

Optimization	Description
-O0	No optimization and compiles as straightforward as possible. Good for debugging.
-O1	Most common optimizations that do not require *speed-space tradeoffs*.
-O2	More optimizations than -O1, such as instruction scheduling. Still, no optimizations affect file size.
-O3	Maximum optimization at the cost of file size, includes everything from -O2 and -O1, plus more.
-funroll-loops	Attempt to unroll loops.

Tab. 2.1: GCC optimization levels and optimization flags [38][p. 49-50].

attempt to vectorize arithmetic operations [29, 39]. Additionally, specifying flags such as *-mcpu=cortex-a53* will optimise the compiler more specifically for the given architecture.

2.8 HYPSO Pipeline

This section is meant to make it easier to understand how embedded processing devices can be utilized on satellite systems while providing better insight into the HYPSO-1 processing pipeline. Additionally, Subsection 2.8.1 is meant to show how a embedded Linux system can be used to control FPGA accelerated designs. This description is however kept short, as understanding the entire HYPSO-1 processing system is not necessary.

The HYPSO-1 and HYPSO-2 missions aim to perform in-orbit processing of image acquisitions, such as super-resolution, atmospheric corrections, target detection and compression, to name a few [40]. The current HYPSO-1 CubeSat currently deploys a minimal processing pipeline of image acquisition and compression, which minimizes the data size for downlink. Due to the architectural design of the satellite software, new system images may be uplinked and deployed during orbit, allowing for updating the processing pipeline [40].

Figure 2.27 describes the current minimal processing pipeline deployed on the HYPSO-1 satellite. As the satellite passes over a target, the satellite performs a pushbroom scan as described in Section 2.2.2. During a capture, image data is streamed to the on-board processing unit, a Zynq-7030 SoC, which performs software binning on the BIP formatted image cube to reduce its size in memory. Upon completion of the acquisition, a FPGA accelerated CCSDS123 compression

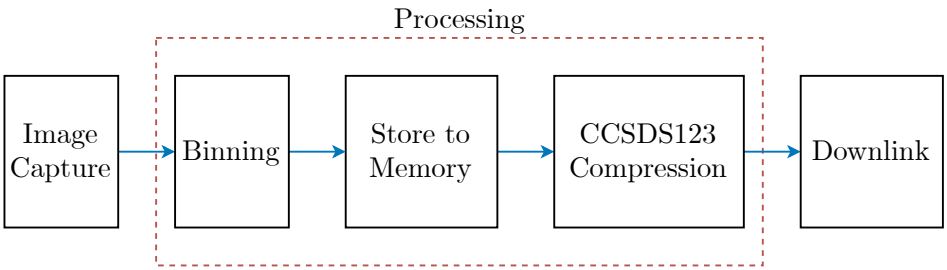


Fig. 2.27: HYPSO-1 and HYPSO-2 minimal processing pipeline, involving capturing, compressing and downlinking hyperspectral image cubes.

algorithm [3] is performed on the image data, outputting a significantly reduced image cube ready for downlink [40].

2.8.1 Linux control of FPGA

The HYPSO-1 processing system consists of a Zynq-7030 SoC from Xilinx, utilizing a simpler, yet similar architecture to that of the UltraScale MPSoC. The SoC has a dual-core ARM Cortex-A9 CPU in conjunction with a FPGA, allowing for a software/hardware codesign.

The CPU runs an embedded Linux distribution called Yocto, where the base of the HYPSO-1 software runs. The CCSDS123 accelerator is implemented on the on-chip FPGA, and has a AXI Lite interface for control. Internally on the FPGA, the CCSDS123 device connects to a custom DMA module through a AXI stream interface. This device further connects to the on-board memory, allowing it to stream data from the RAM with minimal interaction from the CPU [3].

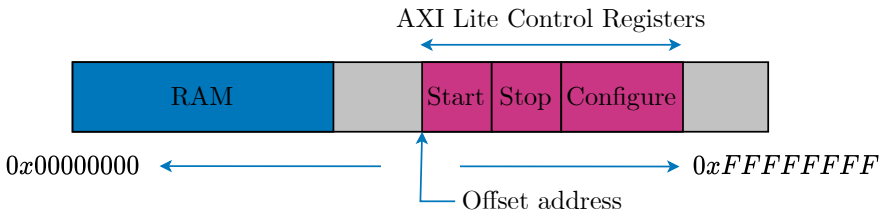


Fig. 2.28: Illustration of how AXI Lite control registers could be mapped within a arbitrary Linux system.

This design allows an application running on the CPU to map the physical memory address of the CubeDMA AXI Lite interface, visualized in Figure 2.28, to virtual memory. This allows the application to both control and configure the programmable

logic. This methodology of controlling accelerated designs is a common approach for embedded Linux-based systems.

2.9 State of the Art

In the paper *U-Net: Convolutional Networks for Biomedical Image Segmentation* [8] the UNet architecture was introduced, as seen in Figure 2.29, which is a fully CNN. The UNet was, in its original work, used to perform semantic segmentation on biomedical imaging, and it showed that it could achieve highly accurate segmentation with little training data. Additionally, it outperformed prior networks shown through the ISBI challenge; a semantic segmentation challenge [8].

The UNets increased ability to perform semantic segmentation comes from its architectural design, which is the origin of its name. The network utilizes two layered paths, a contraction (encoding) path that extracts features from the input data and a symmetric expansive (decoding) path which enables precise localization [8].

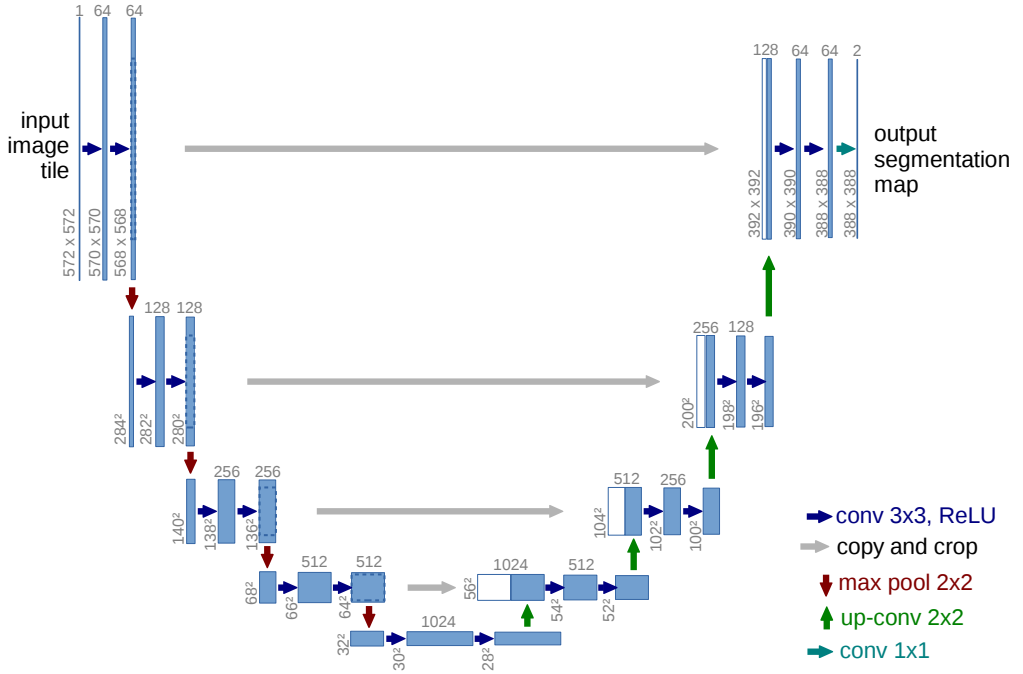


Fig. 2.29: The architecture of the original UNet. The left hand side is called the contraction path, while the right hand side is the expansive path. The network utilizes 5 levels, where the deepest uses 1024 kernels per convolutional layer. The "copy and crop" arrows in the legend represents skip connections. Figure is taken from [8].

Firstly, the contraction path consists of a series of convolutional and max-pooling layers, which extract features form their inputs, and downscales the spatial dimensions of the feature maps as described in Subsection 2.4.1 and 2.4.2 [8].

Following the contraction path is the expansive path that utilizes a series of transposed convolutional and convolutional layers to increase the spatial dimensions of the feature maps. Additionally, the expansive path utilizes skip connections on each level to bring richer feature maps from the contraction path and concatenate them with upscaled features [8].

Furthermore, the original UNet achieved segmentation on 512×512 images within less of a second on a standard GPU, using the Caffe DL framework, thus making it faster than many other CNN methods [8]. Since the network utilizes max-pooling layers in the contraction path, the input image constantly decreases in spatial dimensions, reducing the computational load of deeper convolutional layers placed further down the path.

The performance of the UNet architecture has made it a popular CNN within the field of DL and also within the field of hyperspectral imaging [7, 41, 6]. In the field of hyperspectral EO where labelled data is sparse, the architecture has shown to be exceptionally efficient due to its ability to train on smaller datasets [6, 10]. Also, the increased amount of bands within hyperspectral imaging has shown to make the network more efficient at performing segmentation [6, 10, 7].

On-board processing, such as segmentation and classification, for nanosatellites, is becoming increasingly desirable and possible [40] with the emergence of more efficient embedded computing [9]. However, CubeSats, such as HYPSO-1, having considerable weight and power constraints, cannot utilize power-hungry accelerators such as high-end GPUs. Due to this, the UNet architecture has inspired more compact CNNs like the C-UNet, and C-UNet++ [9] to reduce the resource demand for in-orbit segmentation processing.

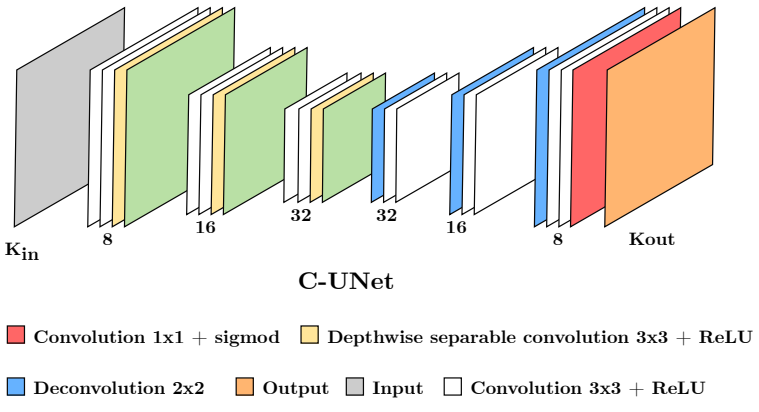


Fig. 2.30: Illustration of the layers within the C-UNet architecture, inspired by the original UNet. Figure is adapted from [9].

The C-UNet and C-UNet++, visualized in Figures 2.30 and 2.31 respectively, has shown to achieve close to the segmentation accuracy of the UNet while being up to *100000* times less complex than the state-of-the-art UNet [9]. These CNN networks reduce the memory demand of the networks by removing the skip connections and reduce complexity by using lightweight layers, such as depth-wise separable convolution [9]. The workings of these lightweight layers can be explored further in work such as [9, 10].

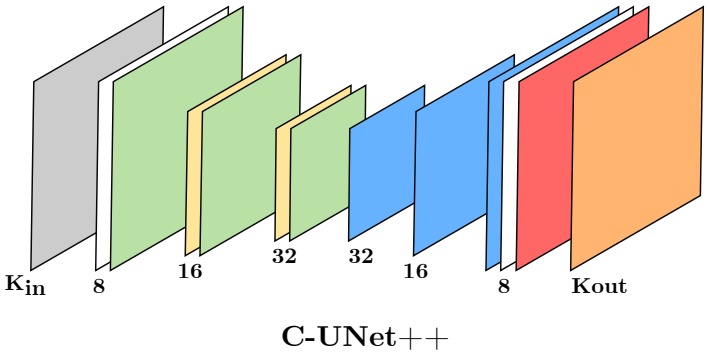


Fig. 2.31: Illustration of the layers within the C-UNet++ architecture, inspired by the original UNet. Figure is adapted from [9].

Due to the decreased complexity of the C-UNet and C-UNet++, they become more viable options for embedded devices, such as the previously mentioned Zynq-7030 and Ultrascale architectures. However, with the cost of reduced segmentation accuracy [10, 11]. Additionally, the reduced size of the CNN networks leads to far less trainable weights and lower memory footprints. With this, work such as [10, 11] has shown that it is possible to achieve fully accelerated FPGA designs of the C-Unet++ and C-UNet architectures.

A fully accelerated design of the UNet currently does not exist, mainly due to the extensive resource constraint of such a network. However, it has been shown that layer-specific accelerators for convolution and deconvolution are possible and could significantly increase the segmentation speed of a software/hardware design approach [12, 26]. The memory footprint of the UNet would still need to be addressed; however, if solved, a UNet implementation could be applicable to satellite systems such as HYPSO-1 and HYPSO-2, even with longer runtimes.

It is, however, clear that accelerated designs, such as the C-UNet and C-UNet++ explored in [10, 11], and the CCSDS123 accelerator introduced in [3] can take long to implement on an existing satellite system, especially for a research-driven CubeSat mission like HYPSO-1 and HYPSO-2 [40]. They are often designed and tested within specific environments, such as the Xilinx tools like Vitis High-Level Synthesis (HLS)

and Vitis Software Platform. Though this decreases development time, which can be crucial for research projects, it does increase the time until the research can be explored further in a realistic scenario, such as on an orbiting and re-configurable CubeSat like HYPSO-1. This thesis will therefore argue that accelerated designs and software implementations should be explored on a as-close-to the existing system as possible, which in the case of HYPSO-1 and HYPSO-2 would be on an embedded Linux OS running on their respective on-board processing hardware.

2.10 FAUBAI Project

The research conducted in this thesis is, in addition to the HYPSO-1 and HYPSO-2 missions, also motivated by the FAUBAI project, a research agreement between NTNU, the University of Oslo and Silvisense, guided by European Space Agency (ESA). Silvisense is a company focusing on autonomous forest monitoring using hyperspectral EO, which is very similar to the research conducted by the SmallSat team at NTNU. The FAUBAI project aims to create a satellite system for performing semantic segmentation of hyperspectral imaging to identify both forest regions and respective tree types. Since downlink is one of the bottlenecks of EO satellites, performing segmentation in-orbit will seriously reduce the data size needed for downlink.

Currently, the Silvisense team have a trained CNN highly inspired by the UNet architecture capable of performing semantic segmentation of forest areas. However, due to the computational load of the UNet, it is desired to look into alternative designs and convolutional accelerators to viably fit UNet architectures on embedded systems within satellites. Previous work within the SmallSat team, such as [10, 11] has looked into the acceleration of compact UNets. As previously mentioned, however, further research into none compact UNets has yet to be explored.

2.11 Python Libraries

Table 2.2 describes the main libraries used for the high-level model/design and storage of weight data. The most important libraries are the Tensorflow, QKeras and Numpy, which make up the core design of the network. Numpy provides ease of use when working with multidimensional arrays, such as datasets which can consist of 4-dimensional data structures, e.g., $N \times H \times W \times D$ where N is the number of images,

H image height, W image width, and D the image bands. Tensorflow provides a set of layers that can be cascaded together to form the desired network architecture and have easy-to-use functions for training the networks with desired parameters. QKeras is a community-designed library for fixed-precision neural network training and functions as an extension to the Tensorflow API.

Library	Description
Tensorflow	Used to create machine learning models, training and predicting on data. The library also provides Keras.
Numpy	Adds support for large multi-dimensional arrays and useful when working with image data.
Scikit-image	Library used for image processing, e.g., format scaling between various data formats, data normalization and exposure scaling.
Scikit-learn	Library used for splitting dataset into train, test and validation sets for network training.
Pillow	Library used for image processing, e.g., saving dataset into patches.
Matplotlib	Used for plotting data and images for visualization purposes.
QKeras	Fixed-precision extension library to the Keras/Tensorflow API. Used to train the network on fixed-precsision weight data.

Tab. 2.2: Main Python libraries used for image pre- and postprocessing, data handling, network training and quantization, and visualization and UNet implementation design and training.

2.12 Datasets

Datasets are sets of data that are used to train neural networks to be able to predict similar data. In the case of CNNs and semantic segmentation, datasets usually consist of a set of images and respective ground truth data. Ground truths usually come in the shape of a 2D image, describing the class for a spatial location in the image data.

Though the number of labelled hyperspectral datasets is few, some publicly available ones are taken from airborne sensors within aeroplanes or satellites. These are relevant for this thesis, as they provide remote sensing scenes resembling that which will be captured from HYPSO-1 or HYPSO-2. These datasets listed below can be found at [42].

- Pavia Centre scene is originally a $1096 \times 1096 \times 102$ hyperspectral dataset captured with the ROSIS sensor. The scene has, however, been reduced in size. The scene has relatively few labelled pixels, with a total of 7456. Good for testing segmentation on urban areas.
- Pavia University scene is a $1096 \times 1096 \times 102$ hyperspectral dataset. The scene is captured with the same sensor used for the Pavia Centre, and also, here is some data removed. However, the dataset has far more labelled pixels, totalling 42776. Good for testing segmentation on urban areas.
- Salinas scene is a $512 \times 217 \times 224$ hyperspectral image captured with the AVIRIS sensor. The scene captured a series of fields of different crops and could be useful to test segmentation on different foliage. The scene has a total of 54129 labelled pixels.
- Indian Pines is a $145 \times 145 \times 224$ hyperspectral image captured with the same sensor as the Salinas scene. Also, this scene consists of fields, much like the Salinas scene. The scene has a total of 16249 labelled pixels.

3 High-Level Model/Design

This chapter presents the high-level model/design of a UNet architecture, providing insight into the tools chosen, an overview of why a high-level model is needed, and a walkthrough of the steps taken from a high-level design to a similar implementation running on an embedded system. Additionally, this chapter provides further insight into the challenges with large CNN models and attempts to find ways to fit such networks on a resource constraint processing system.

3.1 An Overview

To go from a conceptually described architecture, specifically the UNet presented in [8], to an implementation running on an embedded system, there are a series of steps and tools needed to arrive at a functioning but also viable implementation. Figure 3.1 attempts to visualize these steps in a very superficial way, going from a *High-level model* where network design and training are performed, to a *Embedded model* implementation, where the prior exported network weights are used to segment input images. The figure also includes a final and third step, in which an embedded design is improved through a hardware/software codesign.

The emphasis on the *Testing* within Figure 3.1 is also critical, as the high-level model functions as a reference model to later designs. The segmentation achieved on a high-level model can be used to verify the correct behaviour of implemented layers on an embedded device.

3.1.1 High-level Model

Python was chosen as the programming language for designing, training and testing a UNet architecture, and for data preprocessing and export. Languages such as Python are advantageous in such cases as they offer a high abstraction level for operations such as CNNs, image pre-/postprocessing, and data visualization. Additionally, having high-level models for what is later to be low-level designs acts as good test benches for faster debugging.

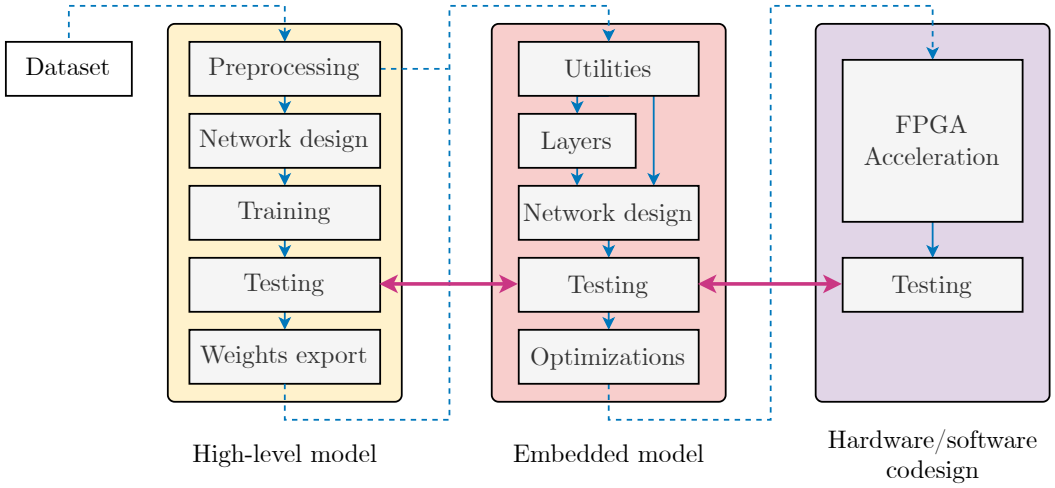


Fig. 3.1: Illustration of the steps required to go from a trained high-level model on a dataset to have a running implementation capable of performing segmentation on the dataset in an embedded device. Additionally, the illustration shows a final step where the embedded implementation is improved with hardware acceleration. The pink arrow between the *Testing* boxes highlights how testing from the start is used until the end.

The *High-level model* can be broken down into 5 separate tasks: *Preprocessing*, *Network design*, *Training*, *Testing* and *Weights export*, as seen in the left hand side of Figure 3.1. These tasks are crucial for a later embedded design, as they provide both the trained weights and testing data for segmentation. *Preprocessing* involves taking in a dataset consisting of image data and truth values and making the data compatible with a high-level neural network. This is an important step, as datasets are often stored differently and are most likely not compatible out of the box. For this step, the *numpy* library was used, as it is compatible with most larger Python libraries.

Network design goes into the use of designing a high-level CNN model, in this case resembling the UNet architecture as seen in Figure 2.29. For this task, the Tensorflow API was chosen, as it is widely used and a lot of existing information exists on the tool. There exists similar APIs for designing and training neural networks, such as PyTorch, Caffe and Neurolab; however, prior use of the Tensorflow tool within the SmallSat team was the deciding factor.

Contradictory to the way it's visualized in Figure 3.1, the tasks of *Training* and *Testing* are more of an iterative process. Within these steps, the neural network is trained and tested repeatedly until the desired performance is achieved. Also here was the Tensorflow tool used.

Once a CNN network is trained to the desired performance, the final task of *Weights export* is performed. This step exports the weights from the high abstraction level Tensorflow model to a compatible format for an embedded Linux application. Additionally, as Figure 3.1 shows, the *Preprocessing* step is also used to export image data in a compatible format for the embedded implementation, which in this case is with the BIP format.

3.1.2 Embedded Model

The *Embedded model* in Figure 3.1 represents the tasks needed for creating a low-level CNN framework capable of performing similar or equal segmentation to that of the high-level model. The tasks are divided into: *Utilities*, *Layers*, *Network design*, *Testing* and *Optimizations*, however the first three are what essentially defines the framework.

The C programming language was chosen for the *Embedded model*, as this is a language that provides great control over system resources through standard C-library functions. The *Utilities* represents the functions created to perform the essential tasks of bringing weights and image data in and out of memory from the Linux filesystem. Additionally, the *Utilities* includes similar preprocessing steps as the high-level model. The *Layers* represents the C-functions written to perform CNN layer operations, such as convolution, max-pooling and transposed convolution. These functions also rely on the use of utility functions, and is visualized in Figure 3.1 as the *Utilities* passing to both the *Layers* and *Network deisgn*. Finally, the *Network design* represents the code used to bring the *Layers* and *Utilities* together to perform image segmentation. The UNet framework is further described in Chapter 4.

Once an embedded UNet design is functioning, the task of *Testing* is performed to ensure that the implementation performs correctly by comparing it with tests performed in the high-level model. Much like the iterative training and testing process in the *High-level model*, the same is performed with compiler optimizations and code changes within the *Embedded model* to get the best segmentation speed. This procedure is further presented and discussed in Section 6.2.

3.1.3 Hardware/Software Codesign

The right-most illustration in Figure 3.1 shows the final step of a design, in which the CNN framework in the *Embedded model* can be improved through FPGA acceleration

of its layers. This step was explored using the Xilinx HLS tool. HLS was chosen as it greatly reduces the complexity of a FPGA design by using object-orientated programming languages such as C++. Additionally, the Vivado Design Suite was used to explore the use of DMA modules to interface FPGAs with system memory and CPUs.

From the figure, it is clear that creating and testing FPGA implementations is hard to do without a foundation, which in this case is the software framework and a high-level model. For this thesis, the *Hardware/Software codesign* represents a suggested convolutional design described further in Chapter 5.

3.2 UNet Analysis and Design

The full UNet architecture, as presented in [8], was chosen as the architecture for this thesis research, motivated by the reasons described in Section 2.9 and 2.10. The terminology "full" in this setting refers to the network utilizing skip connections, which are excluded in more compact versions of the network, e.g., C-UNet and C-UNet++ [9] for being memory demanding on embedded devices. Figure 3.2 shows the network designed for testing in this thesis and is made using the Tensorflow API in Python.

Due to the complexity and computational load of training a neural network, as seen in Section 2.12 regarding backpropagation, the designed network was trained using the Tensorflow API on a high-performance system. This makes it so that the embedded system will only need to perform the prediction with predefined weights from the high-level model.

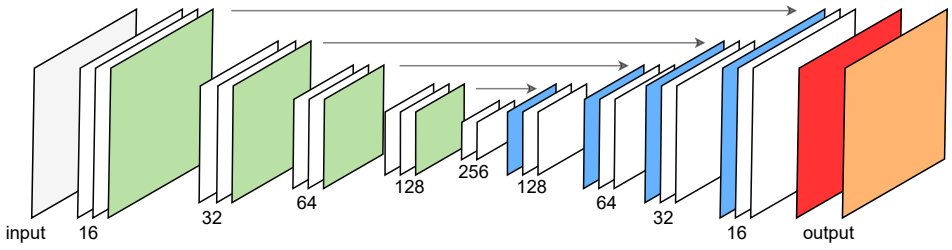


Fig. 3.2: Illustration of a feed-forward CNN network using the UNet architecture. Data flows from left to right, where the layers colored green represent max-pooling layers, the white representing convolutional layers and the blue, transposed convolutionoal layers. Additionally, the red and orange layers represents a final 1×1 convolutional layer.

Comparing 2.29 and 3.2 shows that the latter is slight reduced in the sense that it utilizes less filters per convolutional and transposed convolutional layers. Though this affects the network's accuracy, it does not affect the design choices presented in this thesis, as the UNet architecture remains. A reduced network was chosen mainly for having a reduced number of weights during training and testing, as this circumvented long training times for changes in the code. Additionally, reduced versions of a neural network downscale the complexity, making it easier to grasp core concepts in its functionality. The thought process is to create a fundamental base network that can be expanded or minimized by desire. Hopefully, the core functionality and ideas provided in this thesis will make the full UNet architecture a viable option for an embedded system such as HYPSO-1.

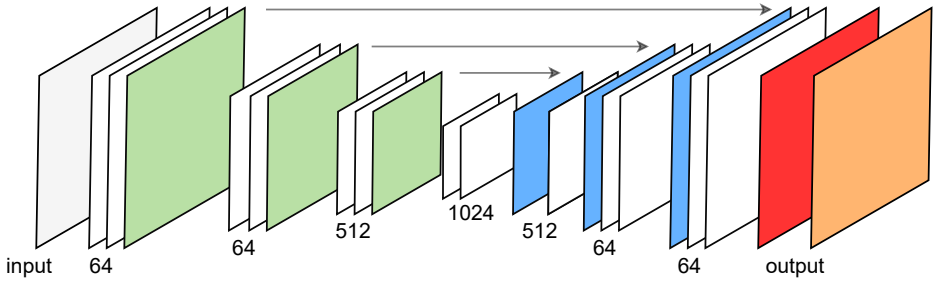


Fig. 3.3: Illustration of an example network provided by the FAUBAI project, usable for e.g., cloud segmentation of satellite images.

An example of a more extensive network is the cloud segmentation network suggested through the FAUBAI project, as seen in Figure 3.3. The essential difference between this network and the one in Figure 3.2 is the number of filters per convolutional and transposed layers, in addition to the number of max-pooling operations defining the depth of the network. Though the UNet can be modified into various versions with different depths and filter sizes, as shown in Figure 3.3 and 2.29, they still remain conceptually the same. Their main difference is the number of trainable parameters and filter output sizes which are essentially only constrained by a device's memory size. Increasing a network's depth and filter sizes will however add to the overall runtime, as more arithmetic operations are required. However, the key components such as the convolutional, max-pooling, transposed convolution and skip connection layers are the same and can be generalized for use in many different versions of the original UNet.

It was important to determine which of the networks presented in Figure 3.2, 2.29 and 3.3 could fit an embedded design. Since it was assumed that a software framework would need to have both the weights and skip connections stored in system memory for fast execution, it was necessary to estimate their memory

requirements. Additionally, it was interesting to determine how their differences affected their system requirement. For this analysis, computational load is neglected, as this essentially only affects the runtime of a network.

One can, e.g., look at the skip connections of the networks presented. As skip connections are required in the decoding (upscaling) part of the network, they must be stored up until the point of need. In other words, the deeper the network, assuming all levels before the deepest layer utilize skip connections, the more memory is required.

Equation $D_{skip} = \frac{H_i \cdot W_i \cdot K_n \cdot F_{byte}}{(1024)^2}$ shows how to calculate the size of the feature maps outputted by a convolutional layer, e.g., the data size required for a given skip connection layer. In the equation, D_{skip} denotes the data size in MB, while H_i and W_i the input width and height, K_n the number of kernels, while F_{byte} is the file format in bytes.

With two-dimensional convolutional layers, as described in section 2.4, each filter corresponds to a feature map in the outputted data in a given layer. E.g., if a layer has 64 kernels, the output will, regardless of input size and depth, be 64 bands or feature maps. With the equation for D_{skip}, one can calculate the size of each skip connection and sum them to find the largest memory space required to run the network effectively. Figure 3.4 attempts to illustrate how the total memory needed for skip connections adds up as one moves down the encoding path of the previously presented UNets.

Figure 3.4 also visualizes how increased filter sizes and depth affect the skip connection memory requirements; however, it shows that they remain within reasonable sizes for embedded systems (< 1GB), even for the more extensive networks. E.g., the FAUBAI UNet skip connections reach 448MB in size (as the bottom level 4 convolutional layer is not used as a skip connection) with 32-bit floating-point values. However, as the last layer needs to be stored until the transposed layers, the total memory footprint in this case essentially adds up to 512MB.

Level	Reduced UNet [MB]	Original UNet [MB]	FAUBAI UNet [MB]
1	64	256	256
2	32	128	64
3	16	64	128
4	8	32	64
5	8	16	-

Tab. 3.1: Skip connection size for the different levels in the reduced UNet, original UNet and FAUBAI UNet. The same data is used in Figure 3.4.

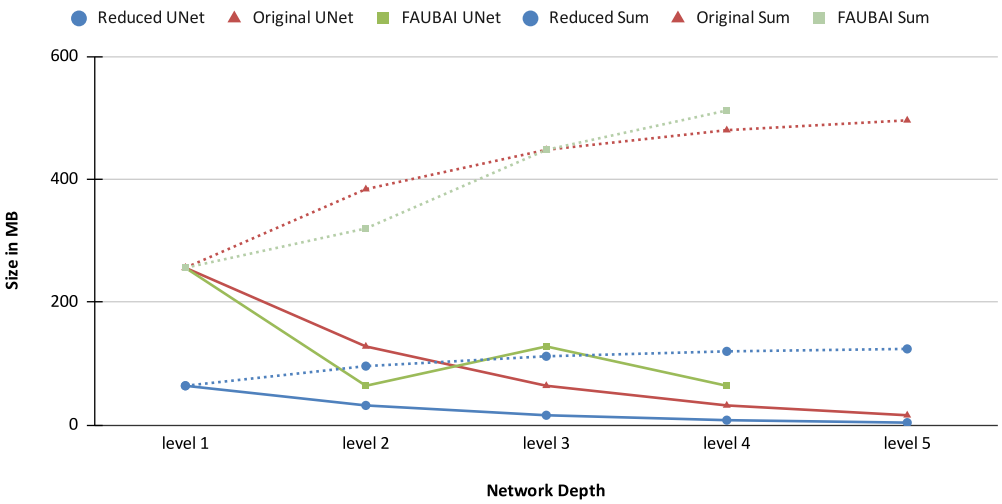


Fig. 3.4: Plot showing the skip connection size for each level in the orignial UNet, reduced UNet and FAUBAI UNet. Additionally, each networks total skip connection size sum is shown as dashed lines with similar colors. The plot is given for a $1024px \times 1024px$ sized input on level 1, which is max-pooled between each level. Also, $F_{byte} = 4$, as the plot assumes the 32-bit float data format. See Table 3.1 for more detailed values.

Trainable hyperparameters are an additional element to consider when calculating the memory footprint of a neural network, such as the UNet, as these tend to become in the number of millions. Compact networks such as the C-UNet and C-UNet++ attempt to reduce these parameters through lightweight encoders and decoders. Additionally, this is needed to reduce the size of the weights to a point where they can be synthesized on FPGAs [9, 10].

Compared to the skip connections, the hyperparameters can be shown to add less to the memory requirement for a network. The number of parameters N_p for a given filter can be calculated with the equation $N_p = K_s^2 \cdot B_i \cdot K_n + K_n$. The depth of each kernel directly correlates with the number of bands for the given input, which is visualized in Figure 2.15. In the equation, the input band size is denoted as B_i, the number of kernels in the filter as K_n, and the kernel size as K_s. Additionally, one may see that an K_n is added to the multiplication, representing the bias value described in Chapter 2 on neural networks. The number of bias values is the same as the the number of kernels in the layer, because each kernel, which corresponds to a neuron in a CNN, has its own specific bias value.

Equation $D_{weight} = \frac{N_p \cdot F_{byte}}{1024}$ describes how the data size of the hyperparameters for a given layer can be calculated using N_p, though noticeably in KB. As with the equation for D_{skip}, the F_{byte} denotes the data format in bytes. Using the equation

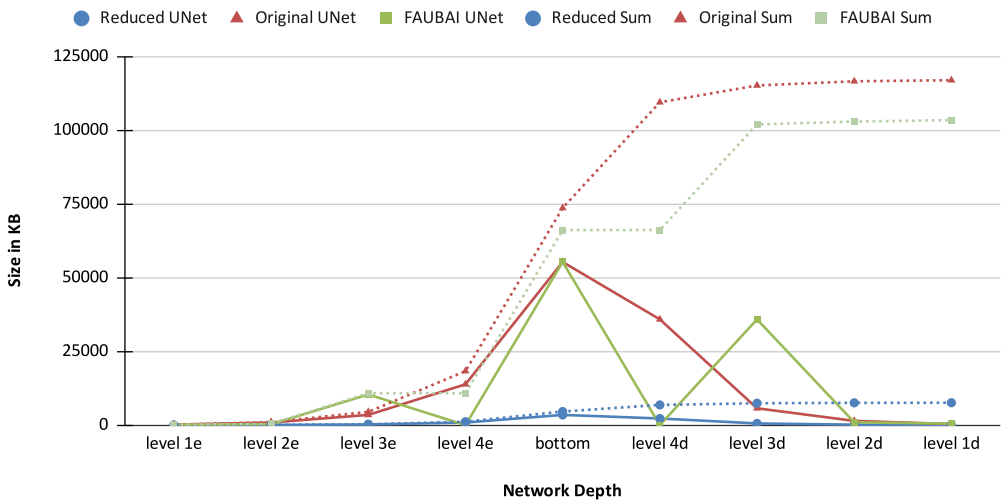


Fig. 3.5: Plot showing the size of hyperparameters at each level for the reduced, original and FAUBAI UNet neural networks. The data size for each layer is given in KB and calculated with a three-band deep input on level 1e. The letters *e* and *d* represents the encoding and decoding levels in the networks. Dashed lines represent the total sum of the given layer, while continuous lines represent layer-specific sizes. The sizes were calculated assuming F_{byte} was 32-bit. See Table 3.2 for more detailed values.

for D_{weight}, the hyperparameter sizes for the reduced, original and FAUBAI networks could be calculated, as seen in Figure 3.5.

The figure shows that though the original and FAUBAI UNets utilize more kernels per layer, the total hyperparameter data size remains within a reasonable memory requirement of approximately 114MB and 100MB, respectively. The figure also makes it clear that weight sizes increase as one moves down the encoding path. This is due to more depth per feature map as one moves deeper into the network. E.g., at the lowest level and at the last convolutional layer, the number of hyperparameters amounts to $N_p = (3^2 \cdot 1024 \cdot 1024) + 1024 \approx 9.438M$ hyperparameters. Additionally, the figure shows that weights for the UNets make up less of a memory requirement than the skip connections. Though the weight number and size do not take up large amounts of memory in, e.g., RAM, they still are large in the sense of FPGA storage. However, this issue is further explored in later chapters.

With the findings presented up until the current point of this chapter, it was relatively safe to assume that larger UNet architectures, such as the original and FAUBAI UNets, should be possible to implement with a hardware/software codesign approach on embedded systems. However, it is clear that as a network increases in depth, the weights' sizes increase quicker, but the memory requirement for skip connections

Level	Reduced UNet [KB]	Original UNet [KB]	FAUBAI UNet [KB]
1e	9.5	151.3	151.3
2e	54.2	865	288.5
3e	216.5	3458	10372
4e	865	13828	-
bottom	3458	55304	55304
4d	2241.5	35846	-
3d	560.7	5698.3	35846
2d	140.4	1425.1	944.8
1d	35.9	359.1	499.3

Tab. 3.2: Hyperparameter size for the different levels in the reduced Unet, original Unet and FAUBAI Unet. The same data is used in Figure 3.5.

decreases. This means having large, fully accelerated CNNs, such as the C-UNet and C-UNet++ implementations suggested in [10, 11], would prove to be not possible with the available architectures. However, accelerating parts of the design and opting for a hardware/software codesign appears to be the realistic approach for an embedded full-sized UNet.

An additional observation was that it could be possible to push the number of kernels per filter to greater than 1024 for a pure software approach, though this would push the data sizes required for bottom layers quite a lot. E.g., if one assumes an additional level to the original UNet with a convolutional layer utilizing 2048 kernels and an input with 2048 bands, this would amount to $D_{weight} \approx 144MB$ for one layer. This is, however, not explored further in this thesis.

With the observations described, it seemed reasonably safe to assume that a UNet architecture could fit the UltraScale architecture. Due to this, the reduced UNet presented in Figure 3.2 was used for further testing, as it was smaller and faster to test.

3.3 Training

The reduced UNet architecture, from now on referred to as the UNet for simplicity, was trained using the Python Tensorflow API. It was decided to train the UNet on the Pavia Centre dataset due to its large spatial dimensions and a similar number of bands to the HYPSO-1 satellite (120 bands). In addition, the dataset had few labelled pixels, compared to other scenes presented in Section 2.12, and was therefore presumed to make training faster.

3.3.1 Pavia Centre Scene

The Pavia Centre scene is a hyperspectral image captured from a plane over northern Italy and resembles the capture of a satellite. The image had originally a spatial dimension of $1096px \times 1096px$, with and 102 bands. However, due to issues with parts of the image, it was cropped to $715 \times 1096 \times 102$, which is visible in the image as a vertical cut. The capture was sampled with a ROSIS sensor, where each pixel was sampled with 16-bit data values. Figure 3.6 shows the full-sized Pavia Centre scene, however where three bands are extracted to create an RGB composite for visualization purposes [42].

The datasets' ground truth contains nine classes; water, trees, asphalt, self-blocking bricks, bitumen, tiles, shadows, meadows and bare soil, as seen in Figure 3.7. Including the unlabeled pixels makes the dataset's classes amount to 10. This value is relevant as it has a slight architectural effect on the UNet design. A CNN creates a final prediction on the pixel class through a 1×1 2D convolutional layer at the end of the network. This layer is shown as an orange plane in Figure 3.2 and 3.3, and needs to contain equal number of kernels as there are classes.

3.3.2 Architecture Parameters

The UNet was constructed using the Keras functions (built into Tensorflow) for the convolutional, transposed convolutional, max-pooling and concatenation layers. Listing 3.1 shows how the various parameters per layer were specified.

```
1 Conv2D(kernel_number, kernel_size, activation_func, padding, input,
      ↪ something_else)
2 MaxPooling2D(kernel_size, input)
3 Conv2DTranspose(kernel_number, kernel_size, strides, padding, input
      ↪ )
```

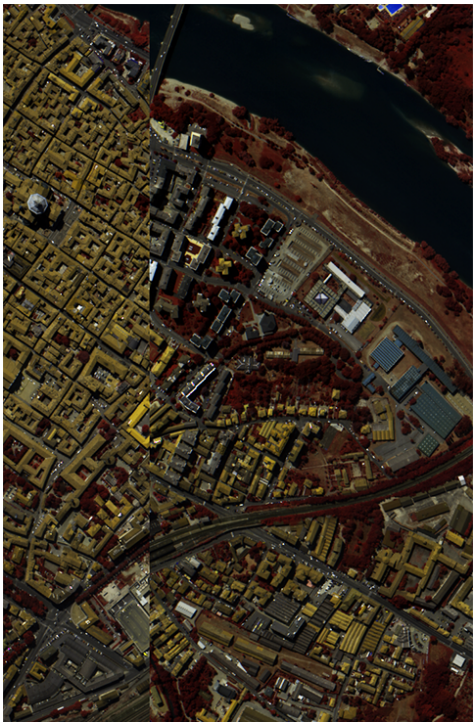


Fig. 3.6: RGB composite of the Pavia Centre scene, where the bands 70, 51 and 19 have been used respectievly.

```
4 concatenate(input1, input2)
```

Listing 3.1: Python code for defining CNN layers for the UNet

The Conv2D layers used the *kernel_number* equal to the ones described in Figure 3.2, and a *kernel_size* of 3×3. Additionally, the stride was not defined, as it was desired to use a stride of 1, which is the default with these layers. With these parameters for the convolutional layers, pixels will overlap as the sliding window of size 3×3 moves one pixel at a time, in both horizontal and vertical directions. These parameters were chosen to mimic the ones used in the original UNet paper [8]. The same goes for the parameters for the other layers.

The Maxpooling2D layers utilized a *kernel_size* of 2×2, as it is desired to split the spatial height and width by two before moving to a new level in the network. This is a crucial feature of the architecture, as this decreases the number of convolutional operations needed the deeper one goes and decreases the size of skip connections.

The Conv2DTranspose layers also used a *kernel_size* of 2×2, as in these layers, one wants to upscale the spatial dimensions by a factor of two. Additionally, for this to be possible, the *stride* is also set to 2, making the 2×2 sliding window move two pixels at a time in both horizontal and vertical spatial directions. Upscaling the

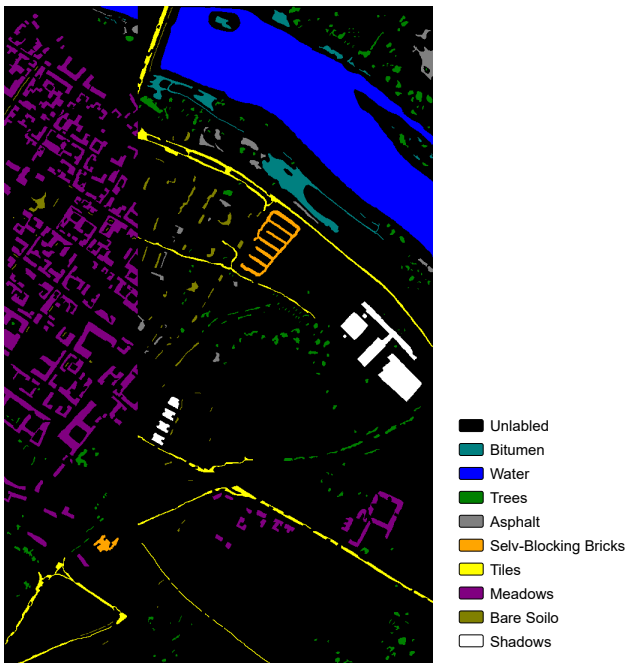


Fig. 3.7: Image showing the labeled pixels for the Pavia Centre scene presented in Figure 3.6. Each class has its own color as described in the right hand class list.

feature maps is necessary to match the spatial dimensions of the skip connections from the encoding path on the same level. Additionally, this upscaling results in the output on the last layer matching the dimensions of the input in the beginning.

Lastly, each convolutional and transposed layer utilized padding to ensure their respective feature map did not lose spatial dimensionality. Zero padding was used to pad the inputs around their spatial dimensions, essentially making the input size $(H_i + 2) \times (W_i + 2)$.

3.3.3 Preprocessing

Preprocessing is an essential step in DL, as datasets often come in various shapes and sizes. Therefore, more often than not, preprocessing is required before being able to pass the data to a neural network as training or prediction data. Hyperspectral datasets are often different in how many bands are captured for a given spatial area and potentially using different sampling sizes, such as 8- or 16-bit values.

A common issue with image segmentation for hyperspectral data is the lack of existing labelled data. Neural networks like CNNs often require larger datasets to generalize, meaning fewer labelled hyperspectral datasets cause issues with training. However, there are methods for extracting more training data from a single dataset,

such as dividing it into patches and augmenting them to appear like different data. Such augmentations can be rotating or mirroring the image data [10]. Using such techniques with the UNet architecture makes it possible to train on relatively small datasets.

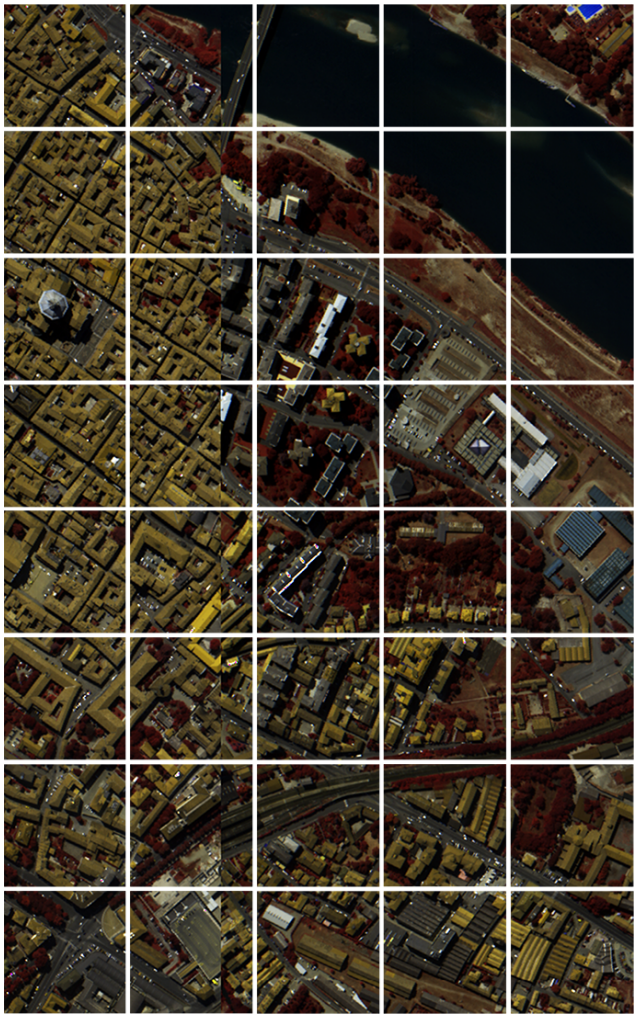


Fig. 3.8: Image containing the patchified version of the Pavia Centre shown in Figure 3.6. The RGB image represents the bands 70, 51 and 19 of the original dataset. In the figure each patch is $128px \times 128px$, and they add up to 40 in total.

Figure 3.8 shows how the Pavia Centre was patched using Python libraries into 40 patches. As the original dataset was 715×1096, it was possible to create 5×8 patches where each had the dimensions $128 \times 128 \times 102$. Note that figure only shows an RGB representation of these. A patch size of 128 was a good balance between amount and size. Various tests were performed with a patch size of 64 and 256, which caused inadequate training results.

The patch sizes were chosen to be equal for both the horizontal and vertical spatial dimensions, simplifying the convolutional operations. It is possible to have different sizes; however, for this thesis, this is not explored.

During training, it is desirable to have the input data in the form of values between 0 and 1, as this helps to produce accuracy and loss calculations which are more understandable during training, in addition to having the probability values outputted by a layer to be within 0 and 1. A way to achieve this is to divide the sampled values by the highest value for a 16-bit number, which is $2^{16} - 1 = 65535$. However, this method works poorly in cases where the dynamic range of the 16-bit value is poorly sampled. E.g. for the Pavia Centre, the highest sampled value is 8000. Therefore, before scaling the input data, the intensity of the data needs to be rescaled to utilize the entire dynamic range of 16-bit values. The way this is achieved is to first find the highest value, in this case, 8000, and calculate a scaling factor $S = \frac{2^{16}-1}{8000} = 8.191875$. All values can be rescaled equally by multiplying each value in the input data with S. Once all values have been rescaled, they can be divided by 65535 to be transformed into floating-point values between 0 and 1.

3.3.4 Training

When training neural networks, it is customary to split the dataset used for training into what is commonly called training, testing and validation sets. In the case of image segmentation, will each set consist of a set number of images and their respective ground truth. These sets have different functionality during training, as illustrated in Figure 3.9. The network uses the training set to perform a prediction, calculate the cost function, and perform backpropagation to determine changes to the parameters. The validation set is used to check whether or not these changes pushed the network in the right direction. This process is repeated for the number of epochs the network is set to train [20][p. 137].

When training neural networks, it is typical to set a maximum amount of epochs before the training halts. Additionally, it is customary to add checks that stop the training once the validation set does not experience any progress. For the training of the UNet, a patience of 100 epochs was used. Additionally, 20% of the Pavia Centre dataset was set as validation data, while the remaining 80% was used for training. This was deemed a good balance between training and validation [20][p. 137].

The testing set is used once the network is finished training and allows the network to perform an unbiased prediction on a piece of similar data [20][p. 137]. In the

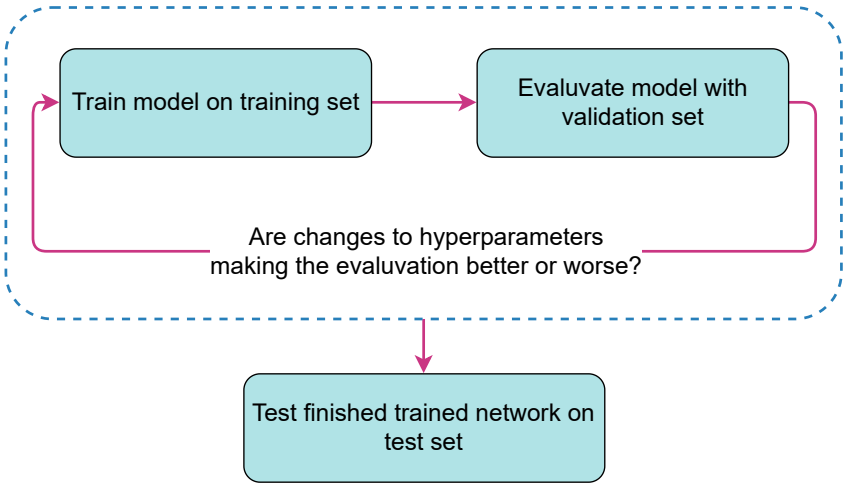


Fig. 3.9: Flowchart describing the neural network training flow, using the training, validation and testing sets.

case of the UNet training, no testing data was explicitly defined, and predictions were performed on the validation set. Though this contradicts Figure 3.9, it was deemed not too big of an issue since training was merely a process for acquiring a high-level prediction model and model weights.

The final results of the trained network can be seen in Chapter 6.1. The parameters described in Figure 6.1 were found through testing and tweaking. The results are not ideal, as the prediction is not perfect; however, for the sake of the leading research topic of this thesis, the segmentation accuracy seen in Chapter 6.1 was deemed good enough.

3.4 Quantized Model

CNNs like the UNet quickly become large with more kernels per convolutional layer. For example, the original UNet shown in Figure 2.29 utilizes around 31 million trainable parameters, where the largest convolutional layer utilized 9.438M parameters alone. The size of such a layer with 32-bit weights would amount to around $9.438 \cdot 10^6 \cdot 4/1024^2 \approx 36MB$. Comparing this with the resource constraints of the UltraScale+ FPGA described in Section 2.5, it was clear that even a single layer would have issues fitting on the FPGA.

Due to the FPGA memory capacity, it was desirable to explore fixed-point precision to reduce the weights' size for individual convolutional layers. Therefore, an additional modified UNet architecture utilizing the QKeras Python library was explored.

The concept of fixed-point precision in accelerated designs is not new. It is often used to reduce the resources needed for arithmetic operations on FPGAs [9][p. 2]. Additionally, fixed-point precision has also been shown to increase the performance for CNN processing on ARM devices [43]. However, the downside of weight quantization is that precision is decreased, which may affect the accuracy of neural network prediction. E.g., using post-quantization of neural network weights is prone to induce inaccuracy, as the network loses parts of its trained weight precision.

However, a more effective approach compared to post-quantization is to perform training quantization, where the network itself is limited to a certain fixed-point precision during training, allowing it to adapt to this reduced precision. This method has shown that prediction accuracy can be maintained relatively well with quantization of weights [43, 10, 11]. Previous work such as [10] also utilized a tool for performing layer-specific quantization, meaning the weights of different layers use different precision based on a specific optimizer for the neural network. Though this can potentially decrease the size of the weights even further, it does introduce another level of complexity to an accelerated design.

For the sake of simplicity, it was decided to use 16-bit fixed-point for each layer in the UNet, as this halved the size of the largest UNet convolutional layer to around 18MB, which would fit the FPGA memory constraint. Previous work has shown that the BFloat16 format presented in Section 2.6.3, could perform better than the half-precision, 16-bit float IEEE standard, due to its increased precision [35]. Due to this, the BFlaot16 was interesting to explore and was therefore used for training the quantized model.

The QKeras, a community addition to the Tensorflow/Keras Python API for network quantization, was used to limit the network to the BFloat16 format. This was used for both the bias and weight values for the convolutional and transposed convolutional layers. The results of training and performing segmentation can be seen in Chapter 6.1.

3.5 Weights File Format

Weights and biases previously referred to as parameters and hyperparameters, make up the core of the UNet. To be able to perform predictions on data with an ARM CPU as presented in Chapter 4, or accelerated on an FPGA as suggested in Chapter 5, the weights need to be in a format that's available, fast to access and possible to interpret.

Previous work such as [10, 11, 12] have had the weights stored on the FPGA itself for fast parallel computations. However, this approach is not possible with more comprehensive networks like the UNet, as explored in previous sections. Moreover, this introduces the issue that every time the network is retrained or changed, the FPGA implementation needs to be resynthesized. Therefore, an alternative approach for storing layer-specific weights in a standardized format is explored. This approach avoids storing every layer's weights within some header file, and allows for a more generalized design that is easily adaptable.

3.5.1 Tensorflow Weights

When training the UNet model with the Tensorflow/Keras API, the API produces a Python model object which contains all trainable and non-trainable parameters for each of the layers of the network. The layers can be accessed by calling *model.layers* with the index value of the desired layer. This approach can be been seen in Listing 3.2, where *i* represents an integer value describing layer number. Line 1 illustrates how to access a specific layer configuration, whilst line 2 shows how to access the given layer's weight values.

```
1 layer_config = model.layers[i].get_config()
2 layer_weights = model.layers[i].get_weights()
```

Listing 3.2: Python code for fetching layer information from network model.

These function calls are practical as they allow for easily fetching information about each layer and their respective weights. The Tensorflow API does provide functionality for exporting the weights and biases; however, this stores the data in a format that is meant to be compatible with Tensorflow. Therefore it was clear that an alternative way of storing the data was needed.

3.5.2 Weights Interleaved by Filter

An observation is that even though CNNs are a well-researched topic, there does not seem to be a standardized method for storing and loading weights and biases. A series of high-level python libraries exist for ML, though most seem to utilize different ways to store their trained data. Additionally, for existing implementations, weight data is often stored into header files which are synthesized or compiled into a FPGA design or software application. This makes it challenging to modify the weights within an embedded device, as new application binaries and bitstream flash files must be provided.

It was therefore desirable to look into a standardized format for storing and reading weight files which were more simplified than the high-level construct provided through Tensorflow in Python. One could optionally store the weight data using the built-in functionality in e.g., *numpy* library's *array.tofile()* function, however this stores the data with an undesired structure. These files quickly become large due to the number of trainable parameters in neural networks, which are easily confusing and hard to manage if not organized well. Additionally, the format or, more specifically, the orientation of the data stored is advantageous to be in a format which is quickly accessible from memory or disk while also being stored in a manner that makes use of data locally in the sense of when the data is needed. This could prove useful, as CPUs often utilize prefetchers relying on data locality.

A new weight and bias storage format is introduced in this thesis and is named Weights Interleaved by Filters (WIF). In this setting, the word *filter* is utilized synonymously with the word *kernel*. The format is noticeably inspired by the BIP format, which is commonly used for hyperspectral images. However, the similarity is not random, as the inspiration comes from how the BIP format stores all bands per pixel, allowing for quick access to bands. The same logic is used for the WIF format, where all weight values for a given kernel are stored before the next kernel comes. To give a better idea of the concept, Figure 3.10 shows a 3x3 convolutional layer kernel on the left. The cube represents the dimensions of a single kernel in a two-dimensional convolutional filter, where x and y denote the *kernel size* K_s, while z denotes the *kernel depth*. z is the same as the number of input bands B_i for the given filter.

Storing the weight data kernel-wise is helpful as, during a convolutional operation, one must calculate the sum of an entire kernel for the corresponding position in the image data. This makes indexing faster and easier when data lie close together. The

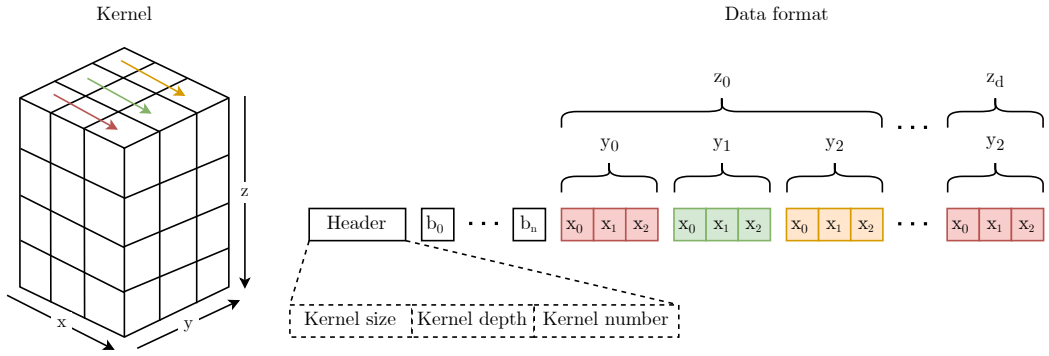


Fig. 3.10: Illustration of WIF file storage format, which stores the weight, bias and configuration values from convolutional layers in a weights per filter (kernel) fashion, from where the name Weights Interleaved by Filters comes from.

same logic was used in the structure of a single kernel, where each level z is stored after each other.

The WIF format was thought out to be a layer-specific format. What is meant by this is that only the weights for all kernels within a single layer are stored together or as a single file. This way, it is easier to manage the weights of specific layers. The format, which is essentially a description of how data is placed in a file, can be seen on the right-hand side of Figure 3.10. The structure starts with a header describing the configuration of the specific layer, such as kernel-specific dimensions and the number of kernels in the layer. The header has its data structured in the order *kernel width*, *kernel depth* and *kernel number*. The reason for the header is to provide quick access to the structure of the data, which will help later on during indexing and looping through the data.

Following the header comes all the bias values for each of the kernels, and are in the figure denoted as b_n, where n is the *kernel number*. It was experimented with having the bias values following each specific kernel; however, this proved to add more complexity than needed during indexing, as skipping bias values would have had to be implemented. Additionally, having the header and bias part stored first in the file format should allow for easier streaming of data later on, as one could more easily stream from specific addresses.

Following the bias values comes the first 3×3 pixels for the first level of z, as shown on the right-hand side of Figure 3.10. The pattern illustrated within the z_0 bracket continues until z_d, where d denotes the *kernel depth*.

Though the figure does not show it, the next kernel in the layer follows straight after the last weights of the previous kernel, continuing with the pattern shown in Figure

3.10 until reaching the last *kernel number*. With this, the WIF file format contains all kernel weights, biases and configuration data and can then be stored in a file with the correct naming of the layer and the proper extension, e.g., *weights_conv2d_c2_2.wif*. *c2_2* in this example would be the second convolutional layer on level 2 in the network.

The WIF format provides the weight and bias values for all filters in a given layer in a very compact manner, which has a simple-to-follow structure and can quickly be loaded into memory through low-level function calls in Linux. The design choice of separate files for each layer is deliberate, as it provides more flexibility in creating different-sized networks and pushes for a more generalized embedded network design. The weight data, which may take upwards to $\sim 100MB$ of data storage, are also divided into smaller pieces, allowing for more flexibility in loading them into memory.

An additional advantage with the WIF format is its structure regarding when the weight values are needed. Figure 3.11 attempts to illustrate how a kernel is passing over the image data while performing convolution. The convolution operation, which takes in a 3x3 pixel area, can get its weight values for the first layer of the kernel by simply looping through the data from the index value of the first weight value. This is due to the storage order described in Figure 3.10. The index value of the first value can be calculated with Equation 3.1, where $i(x, y, z, n)$ is the index function. The variables x, y, z and n denotes the position and *kernel number*, as shown in Figure 3.10. Note that this assumes a 1D array storing the data. The constant H_n denotes the length of the header data, which, unless changed, would be the value 3. K_n, B_i, and K_s denotes the number of kernels, number of input bands, and kernel size, respectively. These constants would be located in the header.

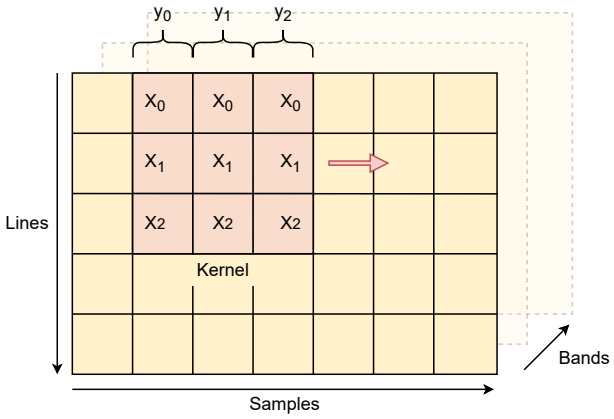


Fig. 3.11: Showing how the weight values in a given filter looks when overlayed on a image grid whilst sliding over the image data.

$$i(x, y, z, n) = H_n + K_n + (n \cdot K_s^2 \cdot B_i) + (z \cdot K_s^2) + (y \cdot K_s) + x \tag{3.1}$$

4 Software Implementation

This chapter will introduce this thesis' approach to a software design of the UNet architecture described in Chapter 3. The design is explored with a 64-bit Linux OS in mind, such as the one running on the ARM Cortex-A53 testbench presented in Chapter 6.2. Furthermore, the programming language C is utilized due to its fine control over system memory whilst providing direct control through system calls to the Linux kernel.

C is a general-purpose programming language, meaning it can be compiled and run on various processing systems and embedded devices. As projected through the WIF format, this thesis attempts to create a generalized design that can be reused and modified. The C language adds to this goal in hopefully making the design choices presented throughout this thesis applicable to different systems. Such systems could be current and future satellite systems, such as the HYPSO-1, HYPSO-2 and FAUBAI projects.

This chapter aims to introduce a foundational design written in C that can perform semantic segmentation on hyperspectral image data with the UNet on an embedded system. The main issue with the network's size is addressed by attempting to minimize the memory footprint of the network during prediction. Additionally, the BIP format will be the core format throughout the design, and every feature map and skip connection will be stored in this format. This choice was mainly motivated by the HYPSO-1 hyperspectral imager storing its data using BIP.

4.1 Application Framework

To perform the processing required by a CNN, a handful of base functionality is needed, such as fetching and preparing the data, making it available in memory, and preprocessing, such as normalization and scaling. The desired functionality can be summarized as follows:

- Read WIF and BIP formats from file and place into memory.

- Layers such as convolution, transposed convolution, max-pooling and skip connections.
- Preprocessing such as normalization, value scaling and padding.
- Array indexation for WIF and BIP formats.
- Write processed data to file.

It was clear that a data handling framework was needed before being able to implement the UNet itself on a Linux OS. This framework will also be beneficial for later testing of eventual accelerating designs in programmable logic.

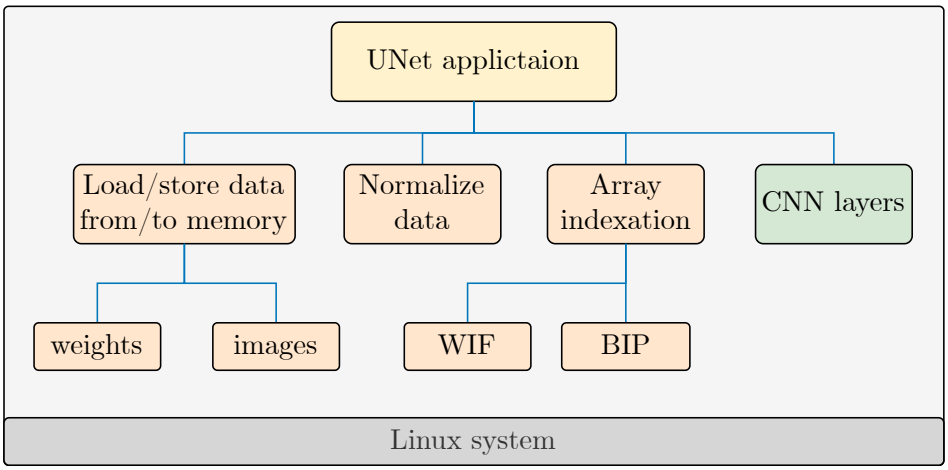


Fig. 4.1: Illustration showing the proposed data handling framework for the UNet application. The data handling functionality is shown in orange, and CNN layers shown in green.

Figure 4.1 shows a proposed application design, where the application is divided into three main parts; the top module implementing the network architecture (yellow), the utility module providing data handling (orange), and a layers module providing CNN layers (green). The proposed design is module-based, whereas each different functionality is made into specific functions. This follows how the UNet is built upon layers executed step by step. This modularity approach helps in mimicking processing flow while also providing better readability of code and ease of use when stitching the functions together for the network architecture.

4.1.1 Utility Module

The utility module described in Figure 4.2 encapsulates the functionality needed for preparing the image and weight data for processing. The module is designed with

the notion that it should work with raw captured BIP and WIF data stored within the Linux system.

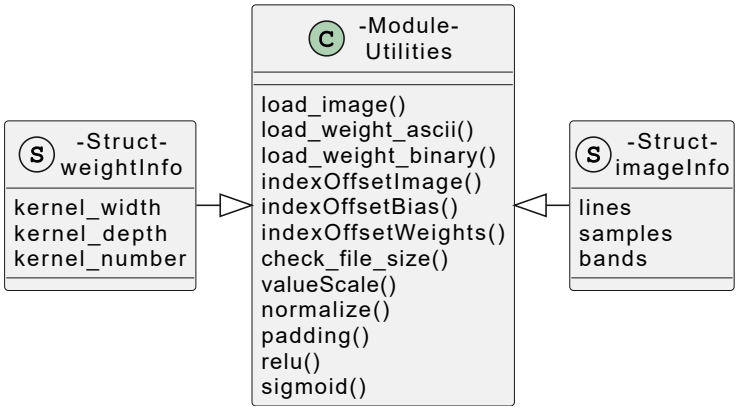


Fig. 4.2: The *utility* module used for various utility operations such as data handling and preprocessing of image data. The *weightInfo* and *imageInfo* are extensions to the utilities module.

Load Functions

The load functions fetch data from the filesystem and move the data to preallocated memory addresses using the C library *fopen* and *fread* functions.

- *load_image()* loads image data formatted with binary into preallocated memory as 16-bit unsiged integer type.
- *load_weight_ascii()* loads weight data formatted with ASCII into preallocated memory as 32-bit floating point type.
- *load_weight_binary()* loads weight data formatted with binary into preallocated memory as 32-bit floating point type.
- *check_file_size()* checks the size of a file in bytes.

Each function that requires a allocated memory space is passed this as an pointer. The reason for this design choice was that it desired to have proper control over dynamically allocated memory. Having functions allocate and free memory at will could potentially cause memory leaks and debugging issues. Therefore, all memory allocations are performed in the top function, where they are freed once used.

The *load_weight_ascii()* was first used to fetch data from *ascii* formatted files; however as seen in Section 6.1, this was not as efficient as loading binary files. This

was because the data had to be parsed when loaded into memory from ASCII files. Therefore, an improved function was created called *load_weight_binary()*.

Index Functions

The C language allows for allocating multidimensional arrays, which could be practical when working with BIP and WIF data formats. However, as multidimensional arrays are essentially an array of pointers to other pointers, it was less complex to load data into one array instead and calculate the data index with specified functions.

- *indexOffsetImage()* calculates the index of a specified value in the 3 dimensional BIP data format.
- *indexOffsetBias()* calculates the index of specified bias values in the WIF data format.
- *indexOffsetWeights()* calculates the index of a specified weight value for a given kernel in the WIF data format.

These functions make it so that one can think about the arrays as multidimensional arrays, making it easier to grasp how data should be processed.

Preprocessing

The neural network described in Chapter 3 is trained on input data that has been scaled to floating-point values between 0 and 1. Due to this, before passing raw BIP to the UNet, the data has to be scaled. The hyperspectral imager of HYPSO-1 utilizes an image sensor that samples the captured image data as 16-bit values, much like the sensor used for the Pavia Centre. As the sensor does not utilize the entire dynamic range of 16-bit, the data also has to be intensity scaled.

- *valeScale()* scales 16-bit image data to 32-bit floating point values in between 0 and 1.
- *normalize()* performs rescaling of image intensity to utilize entire dynamic range of 16-bit unsigned integers.

CNN Specific Utilities

The output spatial dimensions of each convolutional layer will be $H_i - 2 \times W_i - 2$ due to the kernel size and stride. Therefore padding is needed before each convolutional operation.

- *padding()* performs zero padding on input data.
- *relu()* performs the ReLU activation function check.
- *sigmoid()* performs the Sigmoid activation function check.

Structs

Chapter 2 describes how hyperspectral data is often stored as cubes shown in Figure 2.5. During processing with the UNet, it was important to keep track of the dimensions of feature maps and weights. This was solved using custom structs.

Structs provide a structured way of combining user-defined parameters and types within a single pointer. By defining a struct template such as *imageInfo* and *weightInfo* shown in Figure 4.2, they could be used to create collections with information about specific feature maps and weight data throughout the network. Each layer relying on input dimensions could be passed such a struct.

4.1.2 Layer Module

Figure 4.3 shows the layers module containing the CNN layers which will be used to construct the UNet architecture.

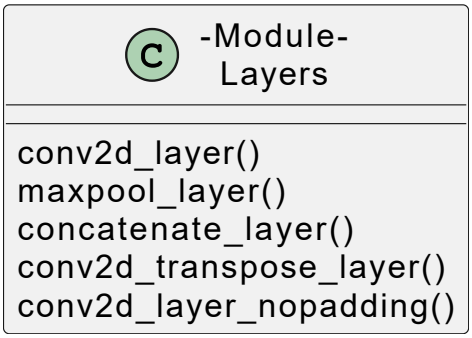


Fig. 4.3: The *layers* module including the CNN convolutional, max-pooling, concatenating and transposed convolutional operations.

Convolutional Layers

The *conv2d_layer()* and *conv2d_layer_nopadding()* functions take in pointers to image and weight data, and perform a sliding window operation over the image data as shown in Section 2.4. The *conv2d_layer()* expect the image to have padding, while the *conv2d_layer_nopadding()* assumes no padding, as it's meant to be used on the last layer of the network to form the final class predictions. The *conv2d_layer_nopadding()* also utilize a 1×1 kernel with a stride of 1.

```
// slide over input
for line=1 to (lines-1) do
    for sample=1 to (samples-1) do

        // calculate convolution for all kernels
        for filter=0 to kernel_number do
            for band=0 to bands do
                for kernel_y=0 to kernel_size do
                    for kernel_x=0 to kernel_size do
                        sum += image_data[index_i] * weight_data[
                            ↪ index_w]
```

Listing 4.1: Convolutional algorithm pseudocode.

Listing 4.1 shows a pseudocode describing how the convolutional operation was performed using a series of nested loops. For every location in the image, given by the line and the sample, one calculates the convolutional sum before calculating the next filter.

```
index_i = indexFunc(line+kernel_x-1, sample+kernel_y-1, band)
index_w = indexFunc(kernel_x, kernel_y, band, filter)
```

Listing 4.2: Index algorithm for image and weight data.

The index for the image and weight data named *index_i* and *index_w* in Listing 4.1 can be calculated with the loop values as shown in Listing 4.2.

Max-pooling and Concatenate Layers

The *maxpool_layer()* functionality was implemented similarly to the convolutional function in that it slides over the input data with a stride of 2, determining the max value in every 2×2 area and placing this value into a new array with half the spatial dimensions. See Section 2.4.2 for more information on the layer.

The *concatenate_layer* loops over the spatial dimensions of the two inputted feature maps (which should be the same) and places their respective data into a new array with the input feature maps' combined band size. Though this approach works fine, it could most likely be solved more efficiently, such as using the C library *mmcpy()* functions, which copies a specified number of data values from one memory location to another.

Transposed Convolutional Layer

The *conv2d_transpose_layer()* is designed to perform the transformed convolution on its input data, outputting the resulting data into a new array with double the spatial dimensions.

The standard way of performing transposed convolution is to store zeros in between between the image values before sliding over the image data with the trained kernels.

However, the transposed convolution used for the UNet utilizes a kernel size 2×2, and a stride of 2. With these parameters, there is no overlap between kernels, as shown in Figure 4.4. Due to this, the transposed convolution operation can be simplified as described in Section 2.19.

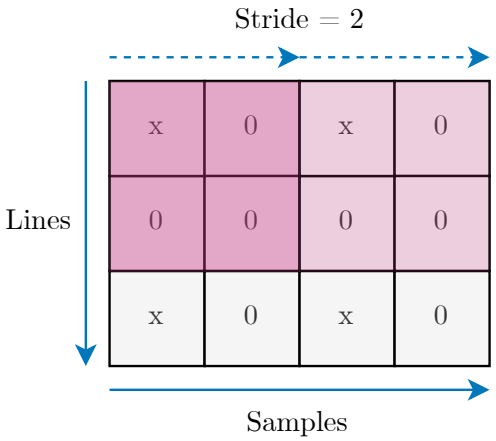


Fig. 4.4: Illustration of a 2×2 kernel moving with a stride of 2 over an image.

Instead of creating a new array of data expanded with inserted zeros, the new pixels in the transposed output can be calculated directly, as shown in Figure 4.5. Here a given pixel in the input data can be multiplied with four respective weights and placed into a new feature map with spatial dimensions $samples \cdot 2 \times lines \cdot 2$. This makes the code simpler and the processing quicker.

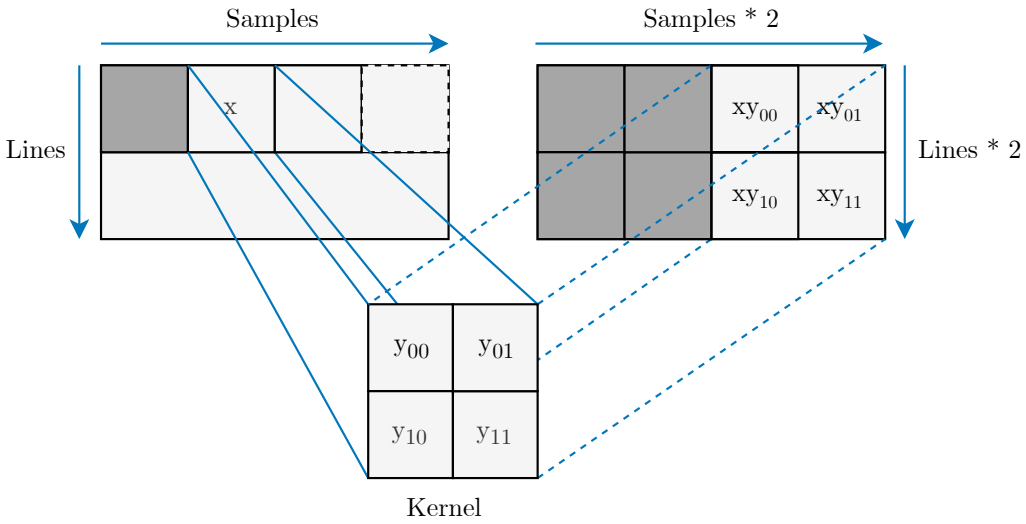


Fig. 4.5: Illustration of how a single pixel is transformed into four pixels using kernel weights.

4.2 UNet Architecture

Figure 4.6 shows the combined UNet application. The top function utilizes the functionality provided through the layers and utility modules to build up the UNet architecture. The structure of the UNet code resembles that of the UNet network in that each operation, e.g., loading input data, preprocessing, or convolution is thought of as layers. Due to this, operations such as normalization and value scaling, which include a series of steps, are referred to as layers.

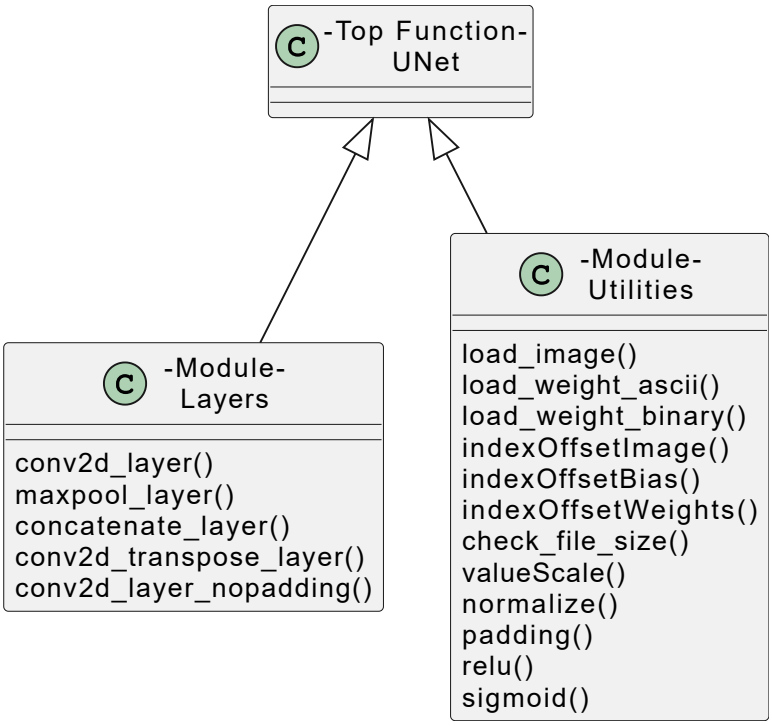


Fig. 4.6: The UNet application consisting of the top function with the network architecture and the layers and utilities modules.

4.2.1 Load Image Data

Image data is loaded into memory as shown in Figure 4.7. One can see that memory is allocated within the UNet top function, and the pointer to the memory address is passed onto the *load_image()* function. The *load_image()* call is also passed a *image_info* struct pointer, which upon success from the Utilities module, will contain dimensions of the input. These values are important as they are used throughout the network to determine spatial dimensions for other layers. The idea is to make a network somewhat generalized so that different image sizes can be tested. The only

limit is that the spatial dimensions need to be divisible by 2^d, where d denotes the depth of the network. In the case of the UNet, $d = 5$, meaning the input must be divisible by $2^5 = 32$. This has to do with the max-pooling and convolutional layers having to pass over the spatial dimension correctly.

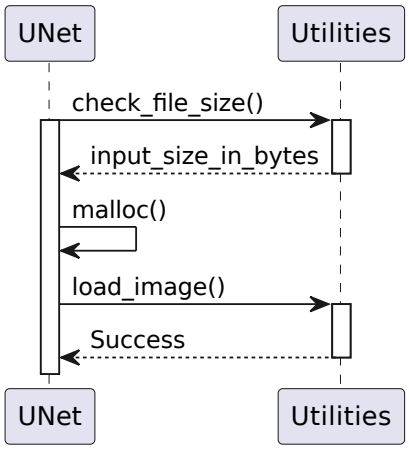


Fig. 4.7: Sequence diagram showing the interaction between the UNet and Utilities to load image data into memory.

4.2.2 Preprocessing

To prepare the input data for the network, it's passed through the preprocessing steps of intensity rescaling and float transformation with the Utilities functions *normalize()* and *valueScale()*. Figure 4.8 describes the flow of these two operations, where they calculate and allocate the required memory size using prior layer information and perform processing on prior layer data.

To minimize the memory footprint of the network, data used for a layer is freed once used, as shown twice in Figure 4.8. The same procedure is also used for all other layers throughout the network. Since a layer in a sequential model depends on the output of the layer prior to itself, it cannot free the memory before finishing its operation. Figure 4.9 attempts to visualise better how layers in the UNet code, in this case, the normalization and value scaling layers, manage allocated memory. This method constantly allocates and frees memory as one moves through the network, minimizing the memory footprint.

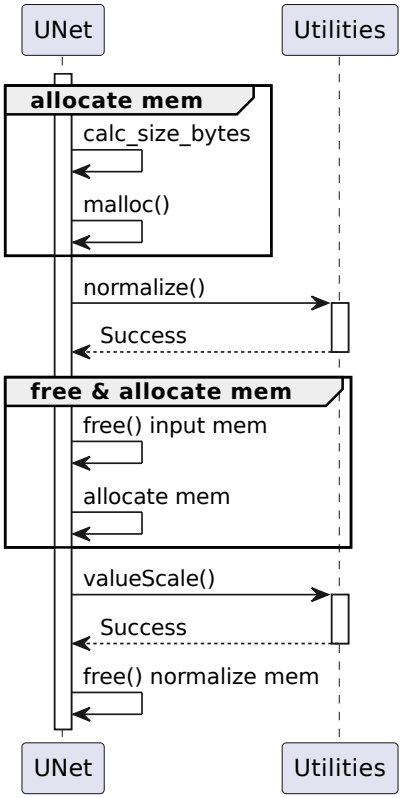


Fig. 4.8: Sequence diagram showing the interaction between the UNet and Utilities module for performing preprocessing on input data.

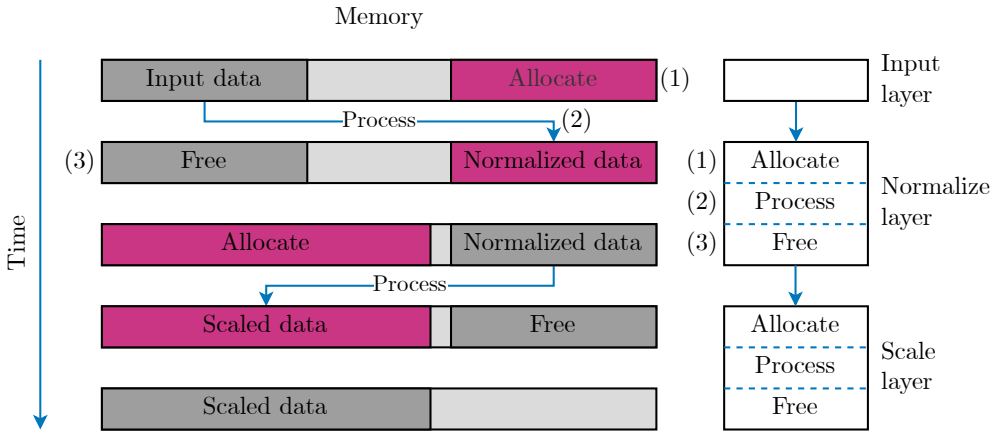


Fig. 4.9: Illustration of how layers in the network perform memory allocation, processing and freeing of memory. The right-hand figure describes the layers in parallel with the left-hand figure describing the layout of the memory. Light grey areas are unallocated memory, while darker grey represents allocated memory. Pink represents newly allocated data.

4.2.3 Network Layers

To keep track of data throughout the network, layers were given specific naming such as *c11* meaning convolutional layer 1 on level 1, *p2* for max-pooling on level 2, and *u6* for transposed layer on level 6. Previously in Chapter 3, the naming of the encoding and decoding paths were separated by the letters *e* and *d*. However, as Figure 4.10 describes, the level denotes the position while moving from left to right. This was a more sensible naming scheme when implementing the network in software.

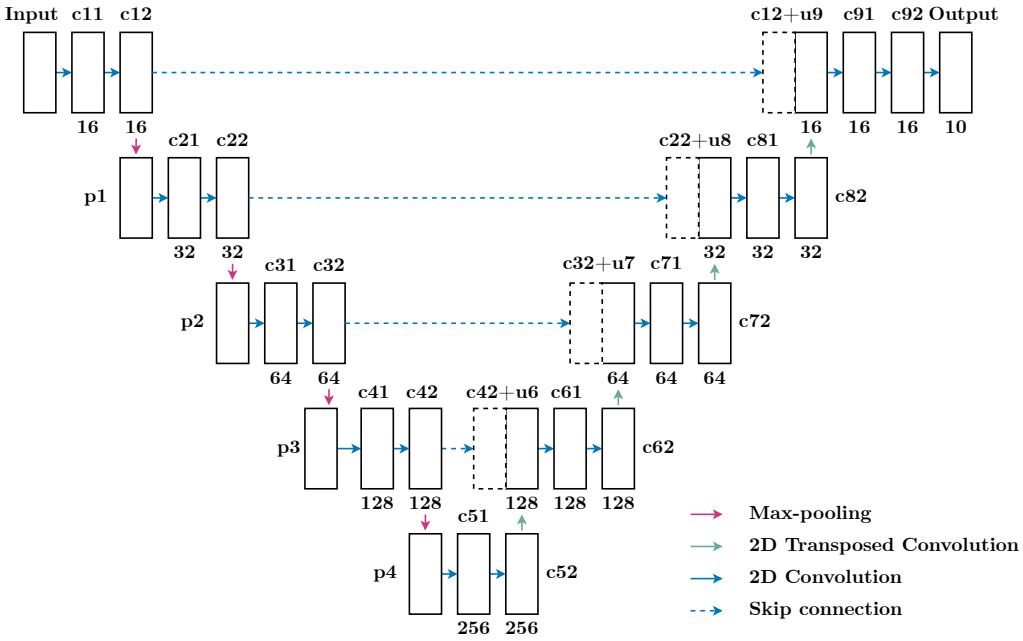


Fig. 4.10: A descriptive model of the UNet where each layer has been given a specific name and arrows represent CNN operations such as convolution, max-pooling, transposed convolution and skip connections.

The convolutional, max-pooling, transposed and skip connection layers were made similarly as the layers described in Figure 4.8 and 4.9 for the preprocessing, in that prior layers information structs were used in calculating new memory sizes which were allocated on need. As a layer finishes, it frees the memory of its prior layer. The only situation this does not occur is on the *p1*, *p2*, *p3* and *p4* layers, as the *c12*, *c22*, *c32* and *c42* outputs are saved for the skip connection layers later on.

Convolutional Layers

Before the convolutional layer can perform its processing, it requires a padded input and weight data. Since weight data is only utilized during a convolutional layer, this data can be discarded once the layer is complete. Figure 4.11 shows the procedure used for a convolutional layer in Figure 4.10, e.g., layer *c21*. The procedure is repeated for all the convolutional layers throughout the network, except for the last layer named *Output* in Figure 4.10, where the *conv2d_nopadding_layer()* function is used instead of *conv2d_layer()*.

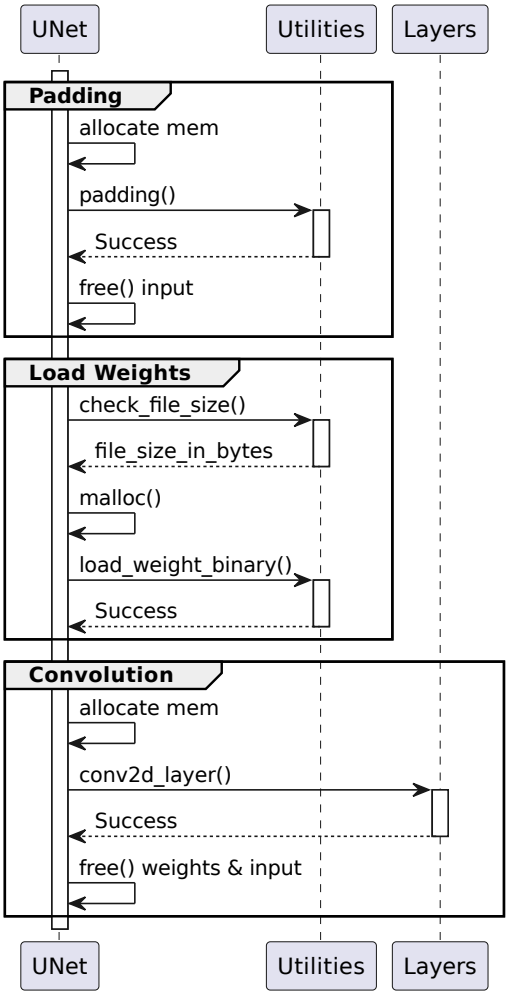


Fig. 4.11: Sequence diagram showing the steps needed to perform convolution on an input. The steps are divided into separate groups representing the various steps required before actually processing the convolution. Note that the *allocate mem* is the group defined in Figure 4.8 and uses previous layer information.

Figure 4.11 describes the processing flow for a convolutional layer, where the operations of padding and fetching weight data for the specific layer are performed first.

The weight data is loaded from the filesystem and discarded once the convolutional operation is complete, along with the input (padded image). In the case where the next layer is a max-pooling layer, the convolved output of the convolutional layer is not freed after the max-pooling, as it later in the network will be used as a skip connection.

Max-pooling Layers

The max-pooling operation does not utilize any trainable parameters or require any padding before use and is therefore relatively simple, as seen in Figure 4.12. It follows the same convention as previously described of calculating the new size from the previous layer; however, noticably, it does not free the input data from memory, as this data will be used as a skip connection.

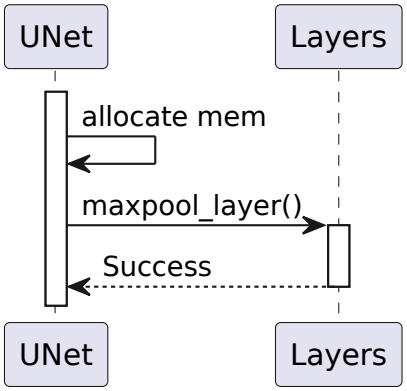


Fig. 4.12: Sequence diagram showing the procedure for performing max-pooling on input data.

Transposed Convolution and Skip Connections

The transposed convolution does not require padding and is, therefore, more simplified in its procedure than the convolutional operation. However, the layer is also grouped with the concatenating layer, as they together upscale and combine feature maps. The combined transposed convolutional layer and skip connections are represented as *c42+u6*, *c32+u7*, *c22+u8* and *c12+u9* in Figure 4.10. Figure 4.13 shows the procedure for loading layer-specific weights, performing transposed convolution and concatenating the transposed output with its respective skip connection.

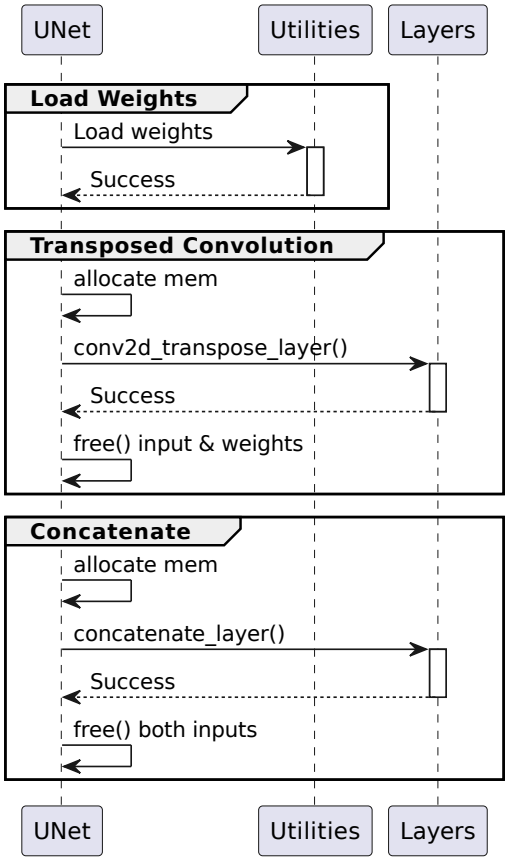


Fig. 4.13: Sequence diagram showing the procedure for performing transposed convolution and concatenation. The *Load Weights* have been simplified as it follows the same process as shown in Figure 4.11.

The procedure for generating the transposed convolution and skip connection shown in Figure 4.13 is repeated for all respective layers in Figure 4.10.

4.3 Linux Filesystem

With the design proposed in Section 4.2, one can perform image segmentation on raw 16-bit image data stored with the BIP format. Additionally, the UNet application reads in 32-bit weight data with the WIF format stored within the Linux file system. This makes it simple to change the weight files for the network. Figure 4.14 proposes a file system layout for the weights and images that allows for easy network testing.

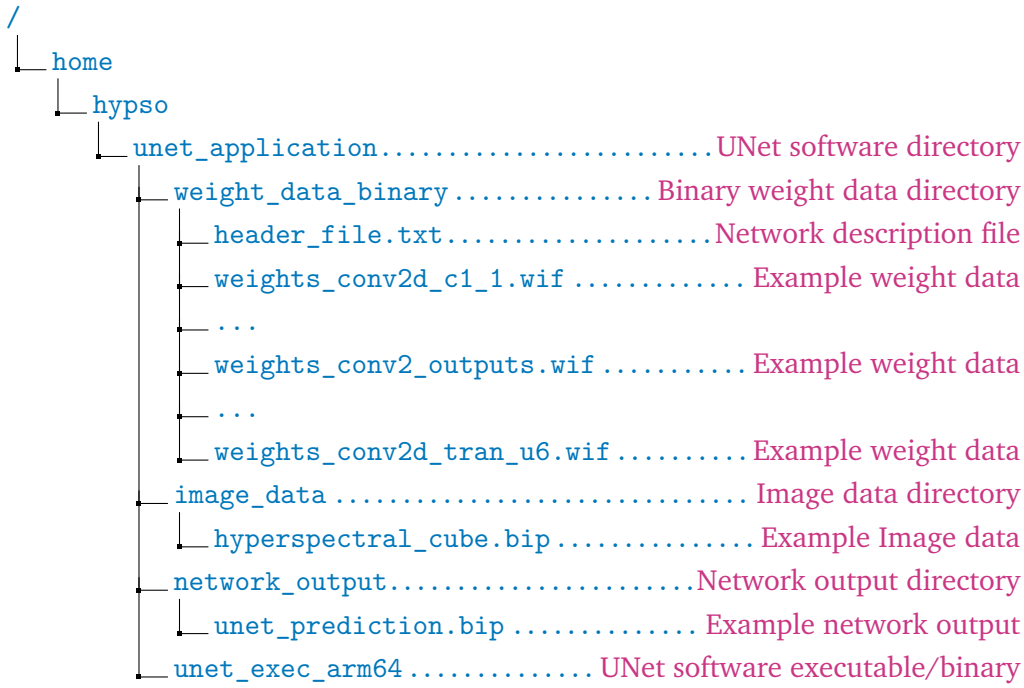


Fig. 4.14: Proposed directory for weights and image data, in addition to the application executable.

This framework makes it relatively fast to test UNet configurations and explore the acceleration of specific layers. The framework provides the tools needed for placing image and weight data into memory, where it can, in a later design, be streamed to an FPGA. However, there are still more aspects to the framework to be explored and discussed further in Section 6.2.

5 Hardware Acceleration

This chapter introduces a suggested convolutional accelerated design thought to be running on an embedded FPGA to support the software implementation of the UNet. In the case of this thesis, the accelerated design is aimed at the UltraScale+ architecture, which utilizes both an ARM Cortex-A53 CPU and a FPGA within the same SoC. This architecture has been chosen as it is used for the UNet software implementation. However, it is more specifically due to being a proposed architecture for on-board processing of the HYPSO-2 satellite.

The reason behind accelerating the convolutional operation is because it is the most computationally heavy operation in CNNs like the UNet architecture. As seen in Figure 4.10, the UNet explored for this thesis utilizes 19 convolutional layers of varying filter number and kernel depth, meaning accelerating these layers may provide great speedup for image segmentation, compared to a CPU only implementation. The benefits are not only speedup, however, as described in Section 2.6. FPGAs are especially good at performing parallel operations at a lower power cost compared to e.g., a GPU [31]. The benefits of accelerating the convolutional layers will therefore be useful in providing faster segmentation of image cubes and reducing the network's power usage, which can be very valuable in a cube satellite.

However, accelerating sliding window operations like convolution is not a new concept, as these operations are used in many image processing applications. In research such as [31], they show that FPGAs outperformed GPUs in energy consumption in most cases of window operations while still maintaining competitive speeds. Additionally, previous work such as [12, 26] has explored the acceleration of the convolutional operation for ML purposes, showing great speedups compared to CPU only implementations.

Previous work has explored the acceleration of the entire C-UNet and C-UNet++ networks [10, 11], as already described previously in the thesis. However, these designs are limited in their network depth and filter sizes, as the FPGAs does not have the resources available to store deep networks consisting of millions of hyperparameters, even with quantization. Approaches within previous work have therefore been to reduce the number of hyperparameters and weight bit-size to make the networks fit the resource constraints. Since the task is to explore the original

UNet, reducing the number of parameters is not an option. However, the size of the UNet, with its 31 million parameters, cannot fit within the resource constraint of an FPGA. The only option is then to explore quantization. In Section 3.4 it was explored to use 16-bit quantization, which makes it possible to fit the weights of the largest convolutional layer (1024 kernels) within the resource constraint of the UltraScale FPGA.

5.1 Constraints

The main issue with accelerated designs such as convolution is that both image and kernel data need to be on the FPGA to perform multiplications and additions of the data. A common approach is, therefore, to stream data from an external device to the FPGA, perform operations, and pass the processed data back to the external device [30][p. 113]. This way, the data does not have to be stored on the FPGA itself.

However, in the case of the convolutional operation, as it is used within CNNs, each kernel has its own set of weights. This means that the weights must be either streamed alongside the image data to the device, or be stored in BRAM logic within the fabric of the FPGA. The approach used in previous work, such as [10, 11] was to synthesise entire FPGA implementations with the weights for each layer. This also avoided the complexity of streaming weights to the device along with the image data. As it is not viable to store the entire UNet weights within the fabric, an alternative approach of storing layer-specific weights is proposed.

5.2 Proposed Design

Figure 5.1 shows a superficial visualization of a proposed convolutional accelerated design. Through using DMA modules, image and weight data can be passed from system memory to the FPGA using AXI stream interfaces, and controlled by the CPU thorugh AXI Lite.

5.2.1 Weight Data

Instead of storing the weights of the entire UNet on the FPGA, it was instead explored to store layer-specific weights using FPGA BRAM logic. In Chapter 3, it was shown

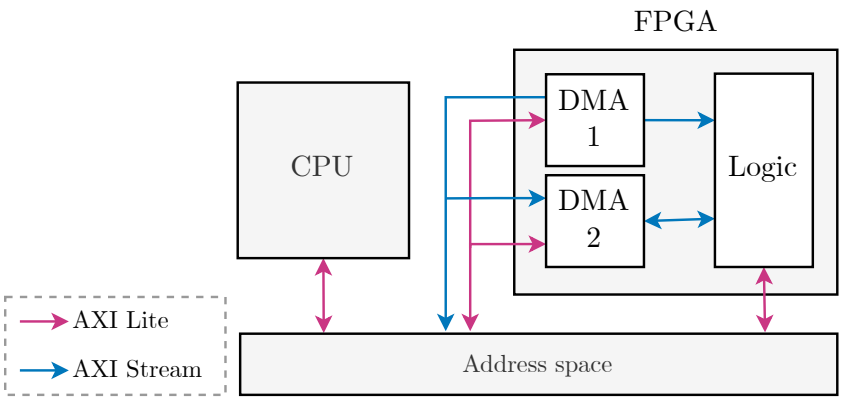


Fig. 5.1: Proposed design for convolutional accelerator controled by the through AXI Lite interface.

that the largest convolutional layer in the original UNet reached a size of around 36MB when the network was trained for 32-bit weights. This did, however, surpass the memory constraint of the on-chip memory of the FPGA. It was shown in Section 3.4, that 16-bit weights should fit the resource constraint of the FPGA, as with the halved bit usage, the highest weight size amounted to around 18MB.

The case of varying weights per layer is, however, an issue as simply storing the weights of one convolutional layer would only reduce the runtime of the software implementation slightly. It was therefore also clear that the weights would have to be streamed to the FPGA as seen in Figure 5.1. Therefore, the suggested design also had to consider varying kernel sizes and input widths. A proposed way of solving this is through having a $1024 \times 1024 \times 9$ buffer for weights and a 1024 buffer for bias values. Since this is the largest convolutional layer possible, it could allow for streaming the weights of smaller layers.

To avoid the complexity of streaming both weight and image data at the same time, it is proposed to stream the weights beforehand, allowing the convolutional acceleration to be as efficient as possible. If one assumes a 32-bit wide DMA bus, and 100MHz clock frequency, which is not unrealistic for the proposed design, the transfer of 18MB could take around $18/((100 \cdot 10^6 \cdot 4)/1024^2) \approx 0.047s \approx 47ms$. Since the DMA can perform the transmission with limited interaction from the CPU, it could be initiated during the padding layer leading up to the convolutional layer to reduce the runtime even further.

5.2.2 Image Data

The suggested design showcased in Figure 5.1 also suggests streaming the image data to the FPGA, in addition to an output stream with the convolved data back into RAM. This has to be done, as image cubes can potentially be large and would most likely not fit the memory constraint of the FPGA. As described in Section 2.4.1, the convolutional operation is essentially a sliding window operation of size $3 \times 3 \times K_d$ moving over the image data while performing multiplications and additions. This is further visualized in Figure 5.2 for a 2D image, which shows how the kernel slides over the image data.

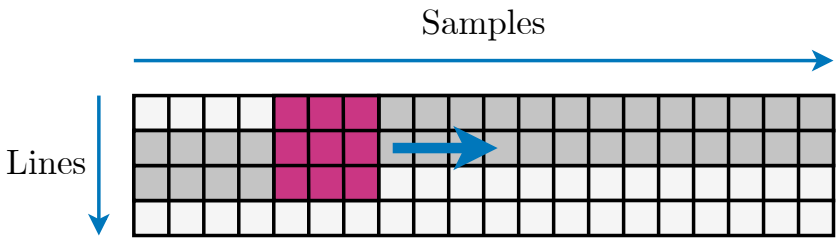


Fig. 5.2: 2D illustration of row buffering, where a kernel of size 3×3 slides over image data with a stride of 1. Adapted from [30][p. 117].

Instead of streaming the same image pixels over and over, which would also slow the acceleration, a common method is to buffer them into the so-called row and window buffers. A row buffer is essentially a form of a cache for image data and stores, in the case of a 3×3 kernel, two rows of pixels [30][p. 117] in an image. In the case of Figure 5.2, this would be the number of *Samples* times 2, which are displayed as grey areas in the figure. When performing a sliding window operation, data arriving can immediately be convolved and stored for the next window "pass" in the row buffers.

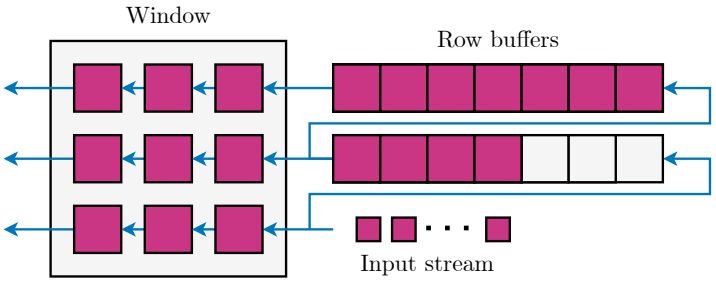


Fig. 5.3: 2D illustration of a 3×3 window buffer with two respective row buffers. Pink cubes represent pixels of data. Adapted from [30][p. 117].

The window buffer is essentially a buffer holding the 3×3 pixels that will be convolved as seen in Figure 5.3. As data arrives, it is placed into the bottom row buffer and bottom window. As indicated in the figure, the data is also shifted through the row buffers, which act like FIFOs. Once a window is full, it can convolve the content with the respective weight values and add to the kernel sum. It is worth noting that these illustrations showcase two-dimensional row and window buffers. However, in reality, they would be three-dimensional, as they would also take in the bands of an image. This is further visualized in Figure 5.4 showing a row- and window buffer for hyperspectral data. Since data, in this case, arrives in BIP order, all bands for a given pixel come first. Depending on the computational architecture for the convolutional part, one could either convolve as bands arrive or wait until all bands for a given window position are available.

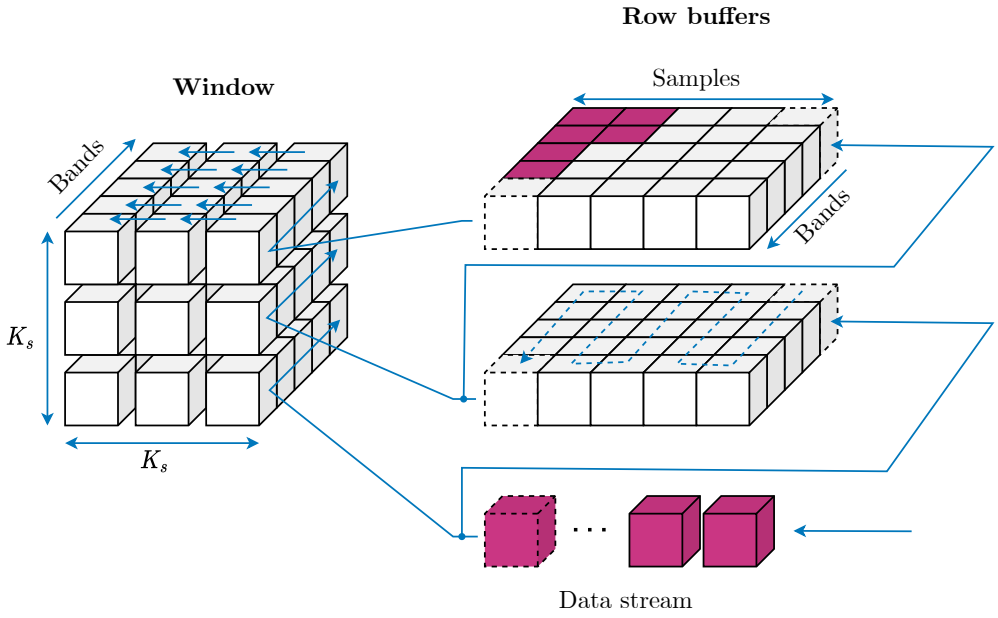


Fig. 5.4: 3D Illustration of how row and window buffers work on image data with several bands. The row buffers act as FIFOs, and are connected together.

5.2.3 Interfaces

Figure 5.5 shows the suggested AXI Lite communication between the CPU and FPGA modules in the accelerated design. As shown in the figure, the CPU could control the modules through their respective control registers in within the address space.

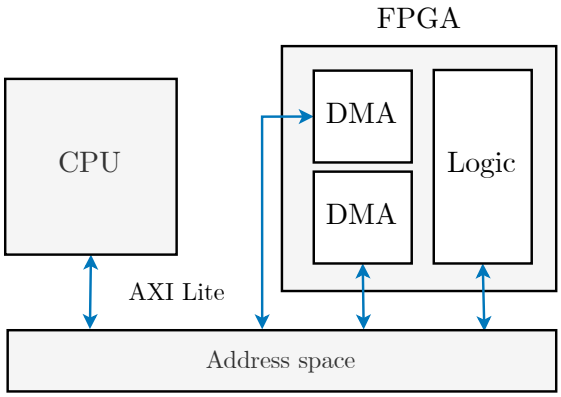


Fig. 5.5: Illustration of the AXI Lite commiunication between the FPGA modules and the CPU.

5.2.4 Software/Hardware Cooperation

The proposed convolutional design is meant to be a supportive accelerator to the UNet framework presented in Chapter 4. Figure 5.7 provides a visualization of how it would fit into the processing flow of the software implementation. The figure shows how the transfer of weights could be issued before starting the padding operation on the image data. Since the DMA can operate on the RAM without significant support from the CPU, the weights can be transferred during the padding operation. Afterwards, the status register of the DMA may be checked to confirm the data is completely transmitted before preparing and sending the actual padded image data.

Figure 5.6 shows a more detailed layout of the suggested convolutional accelerator in Figure 5.1. Buffers are shown as 2D buffers for simplicity; however, in reality, they would be 3D, as seen previously. The weights could be stored within BRAM with multiple ports, allowing for parallel access for convolutional operations. Noticeably,

the internal design would have some control signals, which would be controlled through the AXI Lite interfaces.

The suggested design could be designed to be able to accelerate the largest layer in the original UNet, which is a convolutional layer with 1024 kernels with an input depth of 1024. If calculated assuming 16-bit fixed-point precision, this would require a weight and bias buffer of around 18MB. Additionally, since the convolutional layer should be usable for the first layers in the network, it should have a row buffer width of at least 512. Assuming 32-bit image data and a depth of 102 bands, the row buffer would require less than 1MB in size. In theory, should a design utilize less than 20MB of fabric memory and fit well into the UltraScale FPGA.

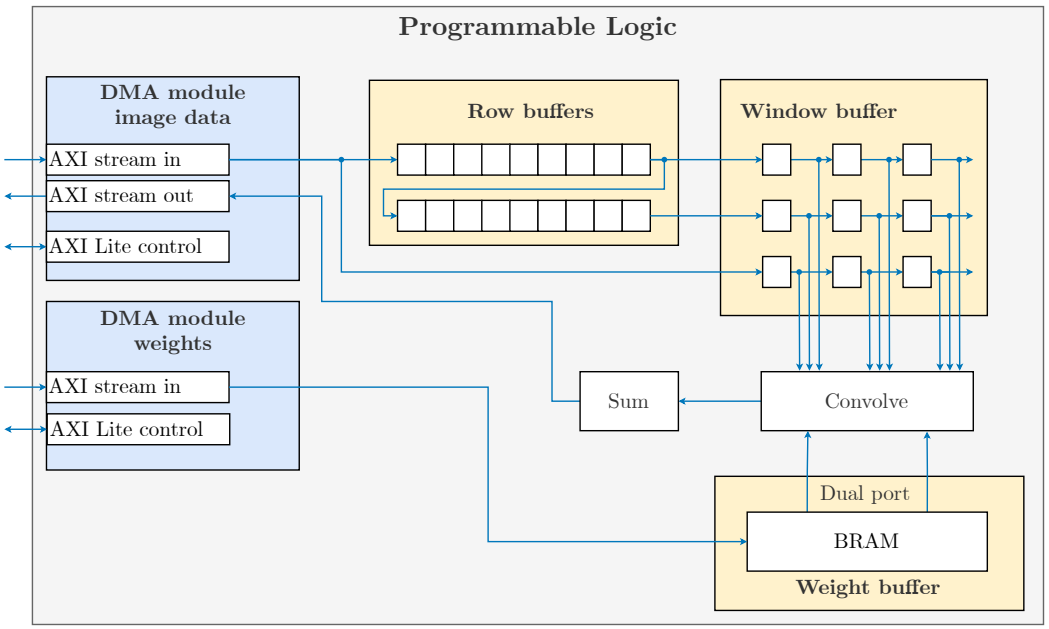


Fig. 5.6: Illustration showing the suggested data flow within the FPGA. Two DMA modules communicate with the CPU through system memory.

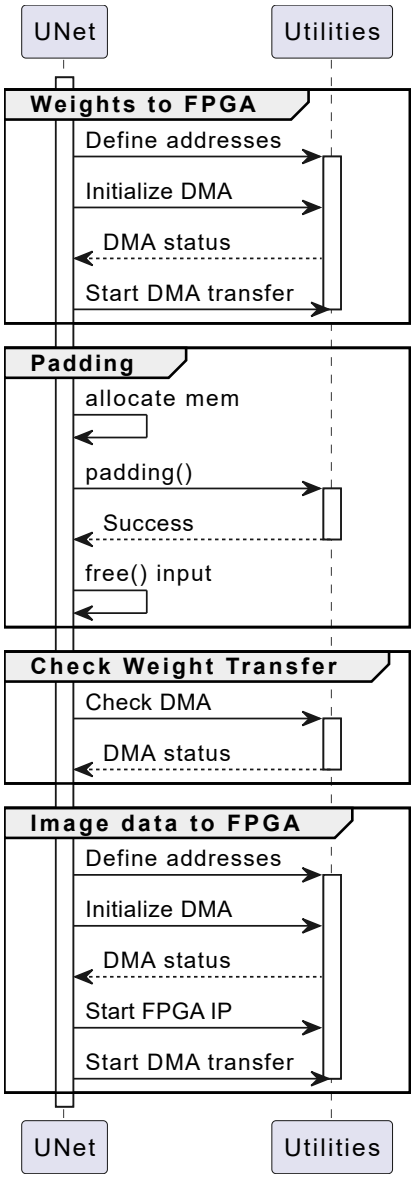


Fig. 5.7: Sequence diagram showing the proposed usage of the convolutional accelerator within the UNet software implementation.

6 Results and Discussion

This chapter will present the results produced throughout the research conducted in this thesis. The chapter is divided into three sections representing the Chapters 3, 4 and 5, which presents the results for each of the chapters. This way of presenting the results was chosen as it provides a more natural walkthrough of the results as they arose in the thesis work. The chapter will also include discussions on each of the chapter results, making it easier to follow for the reader.

To summarize, this chapter presents the results for the training of the UNet and generation of weights, runtimes and memory utilization for the software implementation and discussions around a proposed hardware implementation.

6.1 Training

The UNet described in Chapter 3 consist of two versions, one utilizing 32-bit weights and biases, and one where quantization was utilized to reduce the weight and biases bit size to 16-bit, using the BFloat16 format. Throughout this chapter, the 32-bit UNet will be referred to as the *UNet*, while the quantized one is the *quantized UNet*. The UNet was mainly utilized during testing of the software implementation described in Chapter 4, while the quantized UNet was tested to substantiate the arguments in Chapter 5 for the accelerated design.

Both the UNet and quantized UNet models were trained on the Pavia Centre dataset using 128×128 sized patches, as seen in Figure 3.8. The spatial dimensions of 128×128 per patch were found the be a good middle ground, as it increased the training data from one large image into 32 smaller ones (subtracting the data used for validation) while still maintaining relatively good amounts of information per patch. The patch size had also previously been seen to work well with the Pavia Centre scene in [10][p. 35].

The UNet was trained with two sets of patches, one where the patch size was $128 \times 128 \times 3$, and another of sizes $128 \times 128 \times 102$, which was the original number of bands in the Pavia Centre dataset. The reason was to explore how the difference in

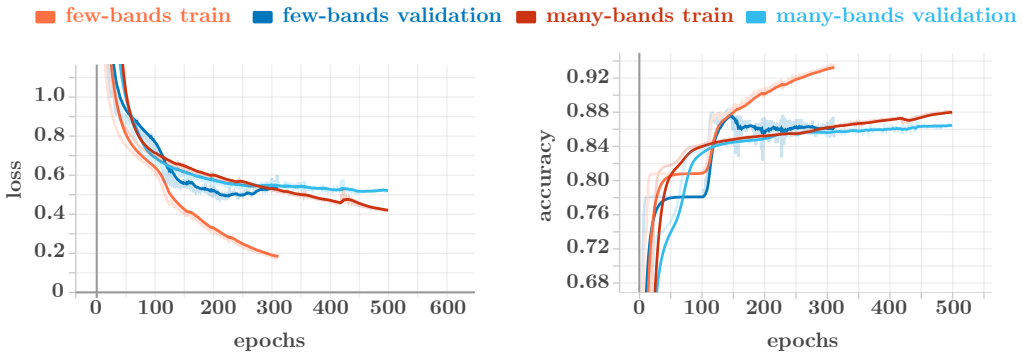


Fig. 6.1: Plots showing the calculated accuracy and loss per epoch for both the train and validation sets on the Pavia Centre dataset for th UNet model. Max epochs were set to 500, with a batch size of 128 and a patience of 100.

network input depth affected the number of weights in the network and the accuracy of the prediction. In this case, the $128 \times 128 \times 3$ patches can be thought of as RGB images, while the $128 \times 128 \times 102$ are hyperspectral data.

The metrics of accuracy and loss was used to measure how the training evolved. Loss can be thought of as the distance between the truth values and the predicted values. The higher the loss, the larger the error. Loss is therefore also dependant on the weight values, which is in this case between 0 and 1. Accuracy, on the other hand, can be seen as a merit of the networks performance. This means that a high accuracy means the model performs better, however not necessarily that the model is good.

The model trained on the $128 \times 128 \times 3$ patches can be seen in Figure 6.1, and is named *few-bands* in the figure. For the training, a patience of 100 was used with validation loss as its metric. This means that if the model cannot improve the validation loss in 100 epochs, it will return and save the model 100 epochs back. This happened for the *few-bands* model, which was aborted by the patience callback, as seen in the figure, and caused the model to not move past 300 epochs. The model reached a validation loss of around 0.5, and an accuracy of around 0.86.

Comparably, the *many-bands* model ran for the entire 500 epochs, however the model ended with similar metric values as the *few-bands* model. This most like has to do with the *many-bands* model having more data to work with, making it slower at improving than the *few-bands* model. This also makes sense, as the *few-bands* model seems to lose generalization around 150 epochs, where one can see that the metrics for the training sets get better faster while the metrics for the validation start to drop.

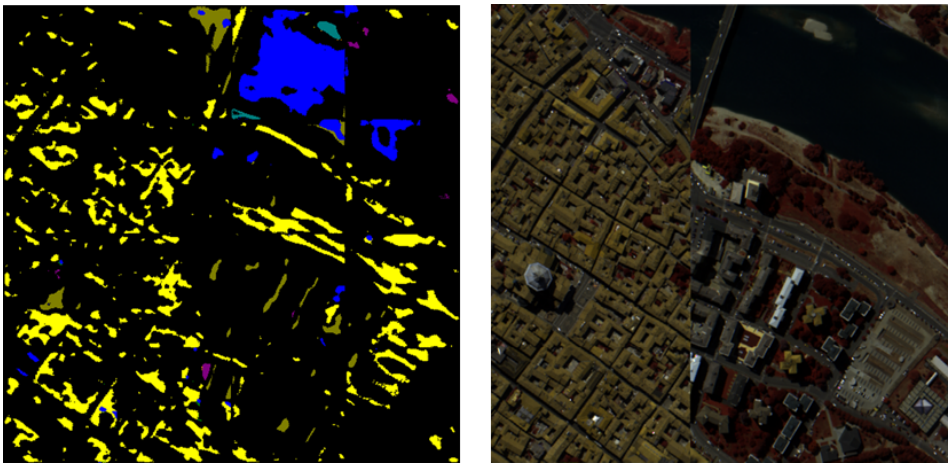


Fig. 6.2: Prediction on $16 \times 128 \times 128 \times 3$ patches in the Pavia Centre, with bands combined together to form a 512×512 image. The bands used was 70, 51, 19.

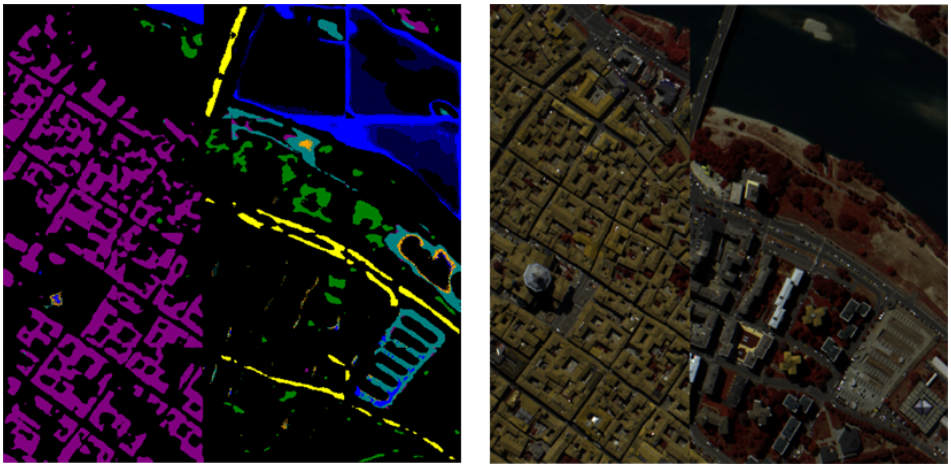


Fig. 6.3: Prediction on $16 \times 128 \times 128 \times 102$ patches in the Pavia Centre, with bands combined together to form a 512×512 image. Note that the image storage cause some color fading, which is not a part of the prediction.

Figure 6.2 and 6.3 shows the prediction with the *few-bands* and *many-bands* models respectively. One can see that training with 102 bands provides much more accurate segmentation than with 3 bands, as assumed, and shows why training networks with more bands produce better predictions. This has to do with the neural network being better at extracting features when more data is available, as explained in Chapter 2. There are ways of measuring how well a network performs, such as the metric Intersection over Union (IoU), which is more accurate than visual inspection; however, for this thesis, visual inspection was deemed adequate. Nevertheless, it's clear that hyperspectral data provides DL networks more data to work with, and therefore more accurate at semantic segmentation.

An interesting observation is that difference in the number of weights for a UNet trained for 102 bands, compared to 3 bands, is only among 13k parameters. This is a result of the UNet utilizing 2-dimensional convolutional layers as described in Section 2.4, which create feature map representations of the input with the same depth, regardless of input. Additionally, the number of sliding window operations is only increased for the first convolutional layer, which should have little overall effect on total runtime.

The segmentation in Figure 6.2 and 6.3 are however not very good, especially when compared to more compact versions like C-UNet++ in work such as [10] which achieved better segmentation accuracy. It should have been possible to acheive similar or better results if data augmentation was used. Techniques such as overlapping, mirroring or rotating patches could make the dataset much larger, giving the network more data to work with. However, this was not explored in this thesis, as the main goal of the training process was to gain weight data and a prediction reference for the software implementation.

6.1.1 Quantized Model

The quantized UNet model was trained with the same parameters as the UNet, 500 epochs, with early stopping on validation loss with 100 in patience. However, the network was only trained with $128 \times 128 \times 102$ sized patches. Figure 6.4 shows the changes in loss and accuracy for every epoch. One can see that the model evolves much like the models shown in Figure 6.1, even when constrained to 16-bit weight values with less precision, as opposed to the 32-bit model.

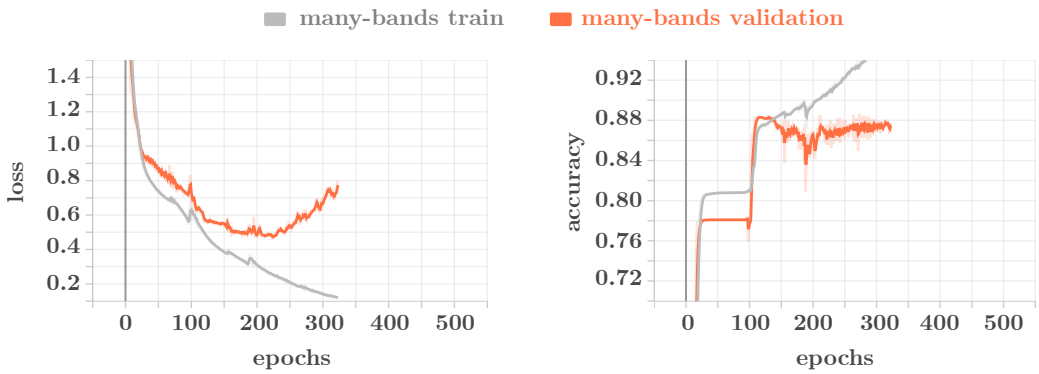


Fig. 6.4: Plot showing the calculated accuracy and loss per epoch for the train and validation sets on the Pavia Centre dataset for the quantized UNet.

Figure 6.5 shows a prediction of a collection of patches from the Pavia Centre dataset, the same as passed through the UNet model shown in Figure 6.3. One can see that the network still manages to extract good features even with weight quantization, as suggested in [43]. The model also seems to predict better in some areas than the prediction in Figure 6.3. However, looking at the plot in Figure 6.4, there are signs of the network memorizing the data as validation loss goes back up. This could explain why some areas are segmented better, as the data used for testing is the same used for training and validation. These results does however show that a quantized UNet, utilizing the BFloat16 format, is comparable to a network trained with 32-bit weights.

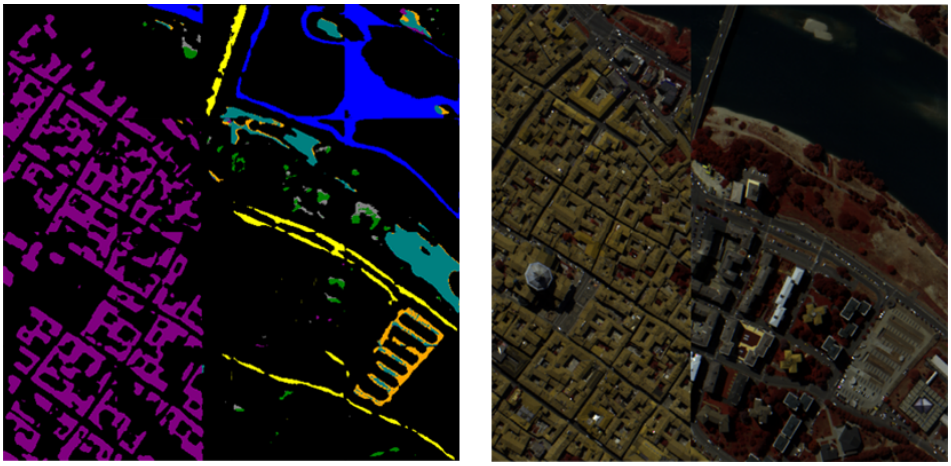


Fig. 6.5: Prediction on $16 \times 128 \times 128$ patches fro the Pavia Centre scene. Their prediction is combined into a 512×512 image.

6.1.2 Weights Testing

When working with file storage in a high abstraction level language such as Python, data is usually stored to a disk using text encoding formats such as American Standard Code for Information Interchange (ASCII). When the weight values are stored as text files, the floating values are cast to ASCII representations and saved to disk. The ASCII format is essentially a character encoding which gives each letter an integer value representation. Though these files are easy to work with and provide better readability and ease of use, it affects how the data needs to be parsed once handled on an embedded system. In case of Linux, the data would need to be parsed using C-functions such as *fopen()* and *fscanf()*, which add extra latency to the read operations.

Due to this, it was explored to store the weight data as binary data instead, which should take up less space on disk and be faster to read into memory. A test was performed where weight data for the UNet, encoded with the WIF format, was read into memory on the ARM test setup described in Section 6.2. Figure 6.6 shows the difference between loading different sized weight files stored with either ASCII or binary (32-bit float) encoding.

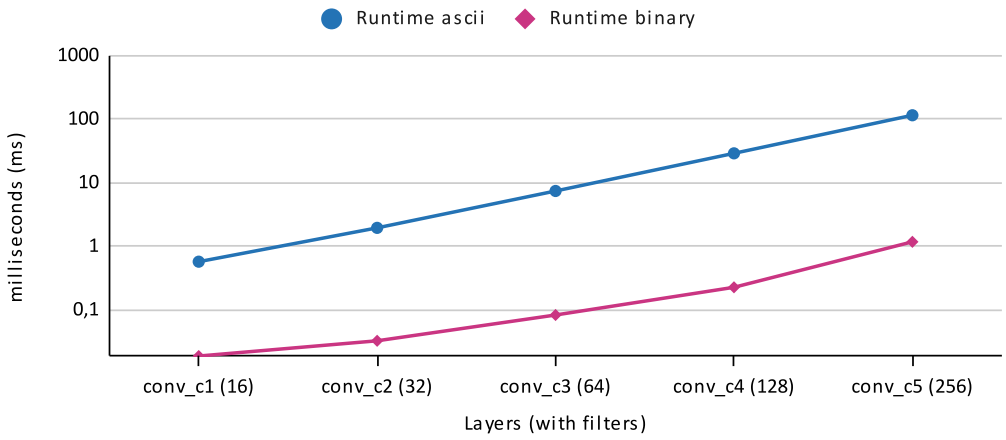


Fig. 6.6: Logarithmic plot of the average runtime for loading weight data into memory from ASCII and binary files with C-functions. Test was done for different sized UNet convolutional filters, with number of tests $N = 10$. Test was run on a Intel i7-9750H running a Ubuntu Linux distrubution.

As Figure 6.6 shows, there is a reasonable difference between reading and parsing ASCII files, compared to binary. Even though the difference was expected, it was higher than anticipated and shows that avoiding the text-to-value parsing with ASCII files provides good speedup, especially if weight data is to be loaded in and out of memory often. For example, as the figure shows, the weight file for a 256-filter convolutional layer, in Figure 6.6 referred to as *conv_c5*, with a kernel size of 3×3, and an input of 256 bands, has approximately $\sim 500K$ parameters. For the ASCII file, the runtime was an average of 116ms, whilst the binary file was 1.2ms. With such low read times, the weight data can be moved in and out of memory during execution without taking up too much runtime.

Additionally, as expected, storing weights as pure binary data proved to be much more space-efficient since binary data is much more compact. For reference, the files for the UNet amounted to around $7.4MB$ when binary, while around $21.7MB$ as ASCII files. Since a future design should allow for uploading new weights to a satellite, these differences make a big difference due to the limited uplink speed of cube satellites.

Having weights stored as binary files was also explored for quantized weights. Logically this reduced the total size of the weights further, reducing the weight size of the UNet to $3.7MB$ when stored as 16-bit data. However, a caveat of binary data compared to ASCII is that the way a computer interprets the data may change, depending on the system's endianness. In the case of the i7 and ARM architectures utilized in this thesis, this was not an issue because they both utilised Little-edian.

6.2 Software Implementation

This section will present the results and discussion around the software implementation described in Chapter 4. The section starts by describing the test setup used for embedded software testing and further the design results with and without compiler optimizations applied. Testing with no optimizations was performed, as it provides a good reference for how optimizations can improve runtime.

6.3 Test Setup

The implementation described in Chapter 4 was tested on different systems utilizing components with significant difference in performance. This gave some good insight into the difference between an edge case device and a more powerful system.

The high performance system utilized an Intel i7-9750H CPU, paired with a Nvidia GTX 1650 GPU. The system utilized the Windows 10 OS, mainly used for running Python code, such as neural network training. However, the system had a Linux co-OS, specifically the Ubuntu 20.04 distribution running as Windows Subsystem for Linux. This was used for the software implementation tests, as it proved to be a good reference for the tests on the embedded system. Additionally, it proved to be a good debugging tool during development.

The low performance system was a ZCU104 development kit from Xilinx, which utilized the Zynq UltraScale+ SoC. The board has 2GB of Double Data Rate (DDR)4 RAM which is accessible to both the SoCs' CPU and programmable logic.

Figure 6.7 shows the development kit used for low performance testing. The board was booted with a Yocto Linux image, built using Xilinxs' Petalinux tool, which runs on the ARM Cortex-A53 CPU. The Linux OS was chosen as it provides a similar environment to the HYPSO-1 processing system, though with a slightly different architecture. For control of the board, UART and Ethernet peripherals were used. The UART peripheral provides access to the board's printouts during boot and is useful to check that it boots correctly and control it via a connected workstation. Additionally, the Ethernet peripheral provided fast remote access to the Linux filesystem through secure shell access (SSH). This was used to transfer binaries and weight files to the board for testing and observe printouts during execution.

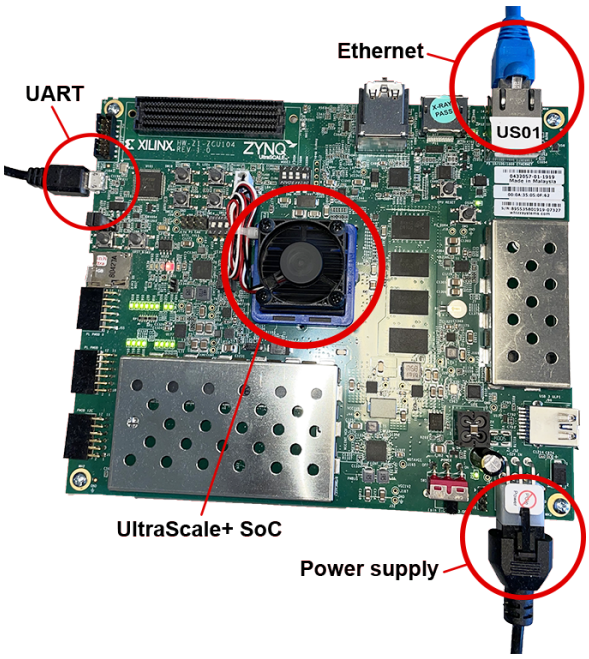


Fig. 6.7: Image showing the ZCU104 development kit test setup, running the UltraScale+ SoC. The image also shows how the board is connected. Ethernet and UART peripherals are used for communication with the board.

6.3.1 Verification Tests

To initially verify that the UNet implementation described in Chapter 4 managed to perform predictions correctly, the code was fed $512 \times 512 \times 3$ and $512 \times 512 \times 102$ image data from the Pavia Centre scene, noticeably the same as used for the network testing in Section 6.1, as this provides a good reference for what to expect as predictions. The hope was that the segmentation from the software implementation would look similar to that of the high-level model.

Figure 6.8 shows the result from feeding the software implementation a $512 \times 512 \times 3$ BIP formatted image from the Pavia scene, utilizing 32-bit weights from the *few-band* model. Comparing the prediction to the one in Figure 6.2 through visual inspection, the software segmentation looks very similar and could even be categorized as performing better in some areas. One can, e.g., see that the model manages to segment the water area more continuously. However, this was somewhat unexpected, as the segmentation should perform the same convolutional operations with the same weight and bias values in theory.

Figure 6.9 shows the result from feeding the software implementation a $512 \times 512 \times 102$ image cube, utilizing 32-bit weights from the *many-band* model. The resulting segmentation performs much better than the *few-band* model, as expected from the

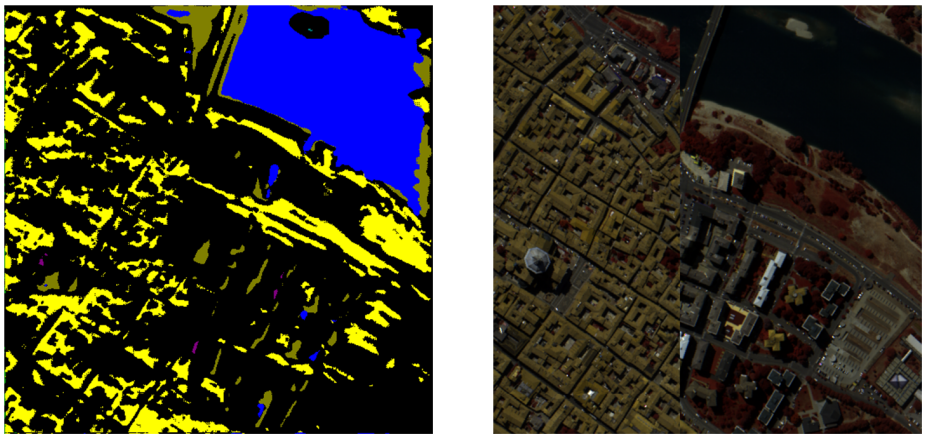


Fig. 6.8: Software implementation prediction on $512 \times 512 \times 3$ image taken from the Pavia Centre dataset, run on the testbench using the UNet 32-bit weights.

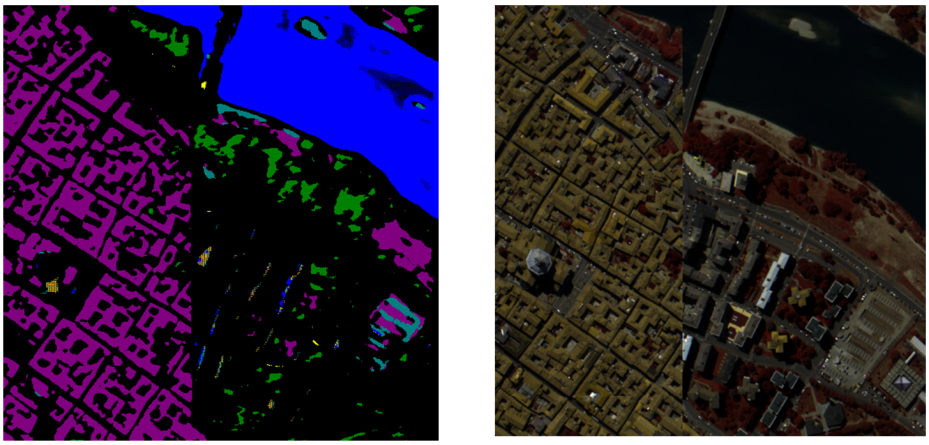


Fig. 6.9: Software implementation prediction on $512 \times 512 \times 102$ image data taken from the Pavia Centre dataset, run on the testbench using the UNet 32-bit weights.

high-level model testing results. However, similar to that seen in the test for the *few-bands* model, the segmentation shows increased accuracy in the water areas. Despite this, the segmentation seems less accurate for certain labels when compared to the high-level segmentation seen in Figure 6.3. This gives cause to believe there are differences in how the UNet implementation and the Tensorflow models perform their predictions.

A hypothesis for where this difference originated was with how the networks were passed image data. Since the high-level model only accepts 128×128 input sizes, the model has to separably segment 16 patches which afterwards may be stitched together to form the 512×512 labelled image. In contrast, the software implementation can perform prediction on 512×512 image cubes in one go and was therefore

not subjected to patched image cubes. Since the UNet utilizes 3×3 kernels with a stride of 1, convolutions overlap, as shown in Section 2.4, making the convolutional operation good where the window is filled with actual image data. However, in the case of the window being on the image borders, padding is applied to maintain spatial dimensionality, which may cause convolutions to perform poorly since it has less actual image data to utilize. Therefore, patches in the high-level model that reside within the image, away from the border, will be subjected to padding when in the software implementation, they are not.

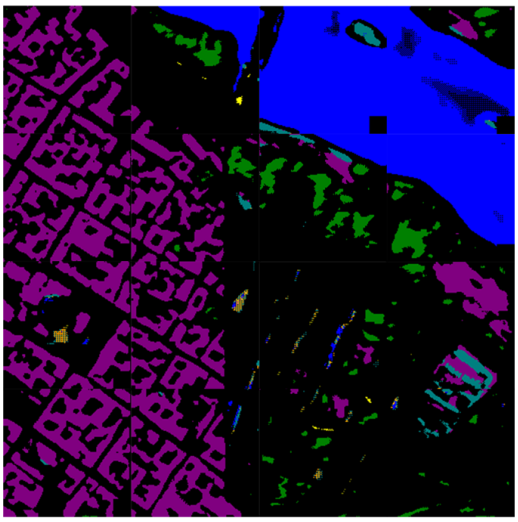


Fig. 6.10: Software implementation prediction on $128 \times 128 \times 102$ patches, which are stitched together to form a 512×512 image representing the segmentation. The 32-bit UNet weights were used for prediction.

To make sure the UNet implementation and its respective high-level model were subjected to the same image data, the $512 \times 512 \times 102$ image cube previously tested was patched into 16 $128 \times 128 \times 102$ patches. These were passed through the UNet implementation and afterwards stitched together. Figure 6.10 shows the resulting labeled image. As one can see, the segmentation performs similarly to that shown in Figure 6.9. This contradicted the hypothesis and meant the reason had to lie elsewhere. Since it is hard to identify exactly how the Tensorflow API performs a prediction, it is not easy to narrow down the cause.

Though the test did not provide further insight into the reason for the difference in segmentation, it did highlight a bug within the UNet implementation. This can be seen in Figure 6.10 as dark cubes in the bottom right corners of the image patches. This most likely is an indexation issue within the implementation, though it does not cause any meaningful issues to the segmentation.

6.3.2 Runtime Tests without Optimizations

To test the performance of the implementation, the start and end of the software implementation were timestamped using the standard C library *time.h*. Additionally, each CNN layer was timestamped to provide a more exact runtime for the different processing parts, which also makes it possible to infer the time used by utility functionality like loading weights and images. The first tests were run for the x86 and ARM test benches to produce comparable results. Additionally, it is important to specify that the ARM testbench only utilizes one core of its CPU. In contrast, the intel i7 testbench is locked to 20% of the CPU resources, though it does utilize multithreading.

Firstly the code was tested without any compiler optimizations or code modifications to get an idea of the implementation performed without modifications. It was expected that this would cause the code to be very slow, as the implementation utilizes large nested loops. Table 6.1 shows the resulting runtimes for both systems, where two image cube dimensions were tested to understand the difference between few and many bands as inputs.

Input Data Cube	Intel i7 Runtime [s]	ARM Cortex-A53 Runtime [s]
$512 \times 512 \times 3$	135.3	1451.2
$512 \times 512 \times 102$	175.3	1905.1

Tab. 6.1: UNet software implementation runtime on ARM and Intel systems without optimizations in seconds.

Table 6.1 shows that the ARM system spent upwards of 20 minutes on one run. This is very slow, however, also expected as there are no optimizations, and the code runs on a single core. Comparably, the Intel i7 used upwards of three minutes to compute the UNet network for a $512 \times 512 \times 102$ input, which is still slow for a system that runs with considerably higher clock speeds and utilizes more cores. It was clear that optimizations would be critical to get usable runtimes.

Table 6.2 shows the runtime for specific layers, and are values taken from the $512 \times 512 \times 102$ no optimization test shown in Table 6.1. As the table shows, most of the runtime goes into the convolutional operation as expected. It is slightly surprising that the transposed operation amounts to only around 2.2% of the runtime, especially since it also utilizes relatively large nested loops and sliding window operation as the convolutional operation. However, at the same time, as described in Chapter 4, the UNet utilizes a 2×2 kernel with a stride of 2, meaning there is no overlap for the transposed convolution. This highly reduces the layer's complexity and the

Layer Type	Intel i7 Runtime [s]	ARM Cortex-A53 Runtime [s]
Convolution	170.52	1897.16
Max-pooling	0.03	0.20
Transposed Convolution	3.64	42.66
Utilities	1.10	7.95

Tab. 6.2: Combined layer runtime on ARM and Intel systems for the UNet with an image cube of $512 \times 512 \times 102$ as input.

number of multiplications and additions per kernel. This has a larger impact on the runtime of transposed layer than anticipated.

6.3.3 Runtime Tests wtih Optimizations

To improve the runtime of the UNet running on the ARM system, general and architecture-specific compiler flags were tested to improve the runtime. Compilers like the GCC have a series of compiler flags that can be used to automate optimization for arithmetic operations and loops, further described in Subsection 2.7.3. File size may be a concern in the case of uplinking new binaries to a satellite system; however, as they tend to remain relatively small regardless of optimization flag due to the application size, this testing will opt for speed over small binary size.

Compiler Optimizations

The same image cube of size $512 \times 512 \times 102$ used for none-optimization testing as seen in Table 6.2 was used for optimized testing, as it provided a good reference for the added benefit of optimizations.

Optimization	Runtime [s]
-O1 -mcpu=cortex-a53	469.33
-O2 -mcpu=cortex-a53	130.84
-O2 -ftree-vectorize	130.28
-O3 -mcpu=cortex-a53	130.73

Tab. 6.3: Testing of optimizations flags for running the UNet implementation faster on the the ARM Cortex-A53. The image cube with dimensions $512 \times 512 \times 102$ was used.

Table 6.3 shows the runtime with various optimization flags used for compiling the UNet implementation for the ARM 64-bit architecture. The best runtime without

manual code optimization achieved was 130.73 seconds with the *-O2 -ftree-vectorize* flags. This is around 14.5 times the speed compared to the same test without optimization flags seen in Table 6.2.

Various tests were also conducted using the *-mcpu=cortex-a53* flag, which is an architecture-specific flag attempting to optimize the code for the given CPU. This did, however, not show any significant improvement, most likely due to the *-O2* and *-O3* enabling most of the optimizations already.

Code Optimization

Additionally, a few tests were conducted where changes to the convolutional layer were done in an attempt to improve runtime further. It was found that the compiler automatically attempts to optimize the code using Single Instruction Multiple Data (SIMD) instructions when using *-O2/3* and *-mcpu* flags [39]. It was therefore attempted to modify the code to accommodate this type of optimization, such as loop-unrolling.

```
1 for i to kernel_y do
2     for j to kernel_x do
3         sum += image_data[index] * weight[index];
```

Listing 6.1: Original code for calculating kernel sum in convolutional layer.

```
1 sum += image_data[index] * weight[index];
2 sum += image_data[index+1] * weight[index+1];
3 ...
4 sum += image_data[index+n] * weight[index+n];
```

Listing 6.2: Adapted code for calculating kernel sum in convolutional layer.

Listing 6.1 shows the original loop for calculating the kernel sum within the convolutional layer. By unrolling this section of the code, as shown in Listing 6.2, the compiler optimized the code more effectively. Since the UNet always utilizes 3×3 kernels for convolution, this approach was viable. However, in the case of different convolutional kernel sizes, an alternative approach would be needed.

The improvement can be seen in Table 6.4, which shows an decrease in runtime from 130.8 to 75.6 seconds. It was also found that the *-mcpu=cortex-a53* seemed to add slightly better runtime after the code changes. It was therefore concluded that the *-O3 -mcpu=cortex-a53* flags with loop-untrolling provided the best runtime, and was therefore used for further testing. Additionally, they had no significant binary size difference compared to other optimization flags.

Optimization	Runtime [s]
-O2 (with CM)	76.24
-O3 (with CM)	75.74
-O3 -mcpu=cortex-a53 (with CM)	75.58

Tab. 6.4: Testing of loop-unrolling, also with optimization flags, while running the UNet implementation on the the ARM Cortex-A53. The image cube with dimensions $512 \times 512 \times 102$ was used. *CM* is an abbreviation of the word *Code Modifications*, and refers to code optimizations.

6.3.4 Final Testing

The *-O3 -mcpu=cortex-a53* with loop-unrolling were used for further testing of the UNet implementation on different input sizes to see how it performed. Two tests were performed for each input size due to the nondeterminism of CPUs, which provides more realistic results. One 3-band test was also performed to showcase the difference in runtime with different input depths.

Table 6.5 shows the average runtime for different input cube sizes for the ARM system. The total runtime expectantly decreases linearly with lower spatial dimensions, following a factor of 4 between tests. The major contributor to this is the convolutional layers, which are the most time demanding. This can further be seen in Figure 6.11. Convolutional layer runtime changes between input sizes since the number of sliding window operations are directly correlated to the spatial dimensions.

	Runtime [s]			
	$512 \times 512 \times 102$	$512 \times 512 \times 3$	$256 \times 256 \times 102$	$128 \times 128 \times 102$
Conv	73.36	55.68	18.33	4.58
Pool	0.022	0.023	0.01	~ 0.00
TConv	1.33	1.34	0.33	0.083
Utilities	2.56	3.97	1.10	0.27
Total	77.27	61.01	19.77	4.93

Tab. 6.5: Testing of different input cube sizes on the ARM Cortex-A53 system for UNet implementation. Compiled using optimizations. Each input was tested for $N = 2$, meaning the values in the table are averages.

It can be seen in Figure 6.12 that convolutional layers also depend on the input depth. The figure shows that the overall runtime increases after passing the *C5* bottom convolution. This is due to the added skip connections increasing the weight depths at the start of a new level in the decoding path, adding more computations than the following convolutional layer.

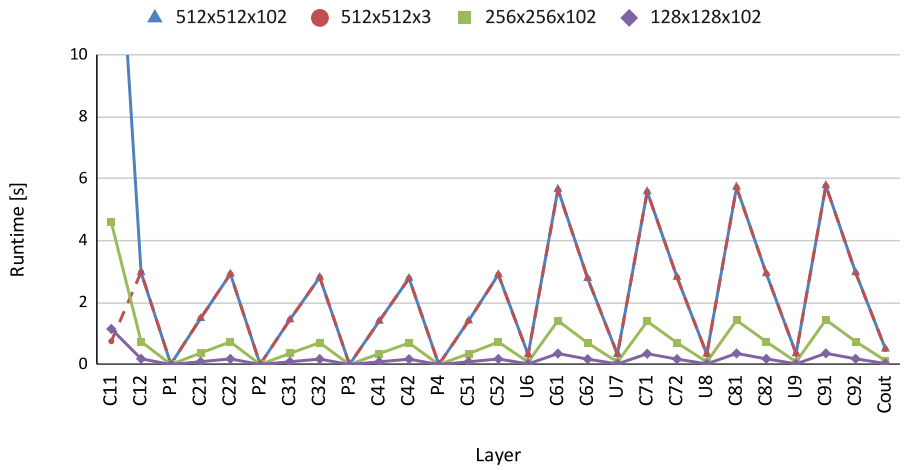


Fig. 6.11: Plot showing the runtime of each layer in the UNet network run on the ARM system. The runtimes are from the tests shown in Table 6.5. For reference, the naming scheme of the layers follows that showcased in Figure 4.10.

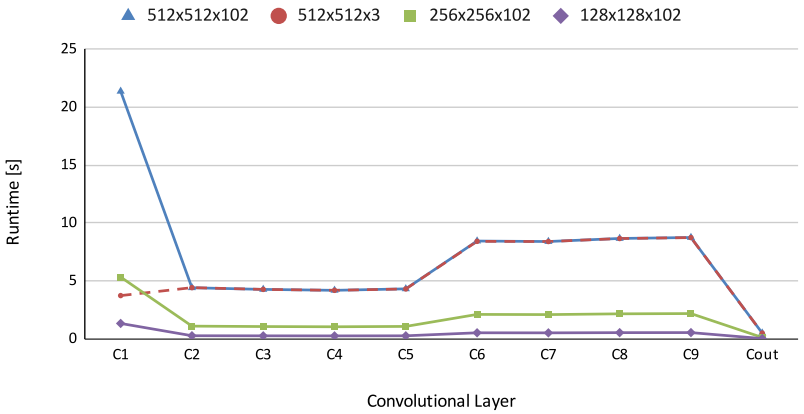


Fig. 6.12: Plot showing the runtime for the different convolutional layers in the UNet implementation from the tests shown in Table 6.5. Each level contain two convolutional layers which are combined together in the figure with respective level names.

From the prior observations, it's clear that the runtime of the utility and max-pooling operations are essentially neglectable when compared to the runtimes of the convolutional and transposed layers. This can be observed for all input sizes tested.

6.3.5 Memory Utilization

To check the memory utilization of the software implementation during segmentation, the system memory was logged. It was interesting to check how the memory behaved while comparing it to the predicted skip connection sizes in Figure 3.4.

To look at the memory usage, the */proc/meminfo* file within the Linux system was logged due to the file containing information about the status of the system memory. Figure 6.13 shows the memory usage observed while segmenting two different image cubes. Memory usage is shown as a percentage of the total memory capacity. Additionally, a dashed line shows the memory usage during idle state, lying around 32.8%, or about 650MB as a reference.

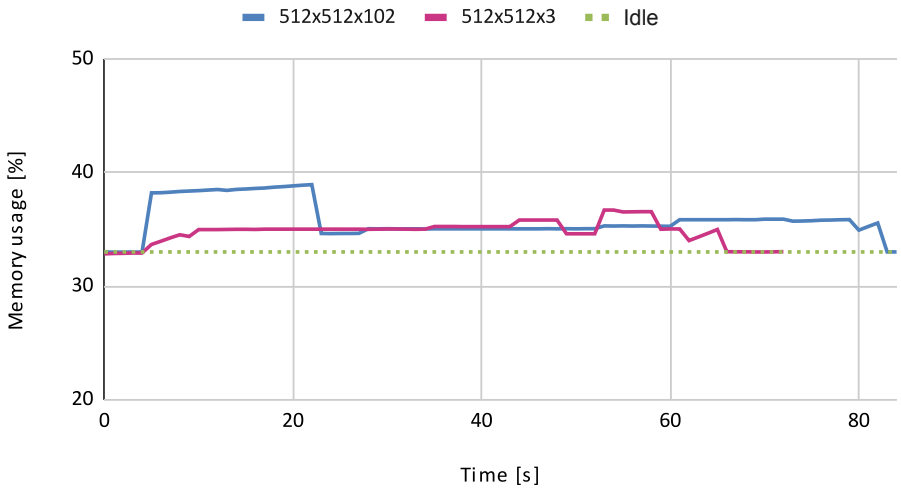


Fig. 6.13: Plot of the memory usage as a percentage of total memory for two input cubes to the UNet implementation. The plots show the usage before, during and after the segmentation of input cubes. Tests were performed on the ARM system, which has a total of 2GB RAM available. The green dashed line named *Idle* represents the system's idle memory usage.

For a $512 \times 512 \times 3$ image cube, the median memory usage of 35% was observed, while for a $512 \times 512 \times 102$ image cube, the median was slightly higher at 35.3%. However, this variation is expected, as the input with more bands requires more space in memory when loaded into memory. This becomes more clear in Figure 6.13, as one can see that the maximum memory usage reached around 38.9%, equalling approximately 120MB above idle state for the input with 102 bands. Since features are extracted in the first layer, and the feature depth of the first layer corresponds to the number of kernels, which are equal for both cases, one can observe that the memory usage aligns after the first layer around 20 seconds.

Compared to the calculated size of a $512 \times 512 \times 102$ input image and the weight size of a 16 kernel filter, the total size would amount to around 102MB when all data is 32-bit. Therefore, the memory usage observed versus the calculated size seems to match quite closely, meaning the previously calculated skip connection sizes in Figure 3.4 are relatively close estimations to the memory requirement for a network. This can be useful for future determination of the resource demand for various CNN architectures.

6.3.6 Summary and Final Notes

The tests proved that segmentation worked with the software implementation of the reduced UNet model, though with certain variances in the accuracy of the segmentation compared to the high-level model. Additionally, certain bugs such as dark cubes are apparent; however, they do not seem to have a significant impact on the segmentation itself.

The variances experienced in the segmentation are hard to explain, as the weights used for both the software and high-level networks should be the same. One hypothesis is that precision is lost during the conversion from the high-level model into a binary file, causing the software application's activation functions to fire slightly differently.

A different explanation could be the orientation of the weight data within the kernel window. Since the Tensorflow API calls that perform prediction are of a high abstraction level, it is difficult to determine which order kernel weights are positioned. Since the kernels multiply specific pixels within a window with a specific weight value, a change to this should cause a difference in the total sum of the kernel. The kernel will, however, still produce a collective sum that might not be too different from the "correct" sum. This hypothesis would also make sense, as certain areas or labels gain segmentation accuracy while others decrease.

The UNet framework designed proved to be very efficient to work with, as changing the layer weights can be done quickly by simply storing new WIF files into the Linux filesystem. This approach, in contrast to previous work [10, 11], allows the network to be updated without recompiling the application binaries, assuming the architecture itself is not to be modified. With a deployed satellite system in mind, this is a useful feature, as updating the on-board processing image is a risky procedure. Instead, weight files can be uplinked onto the satellite processing system, having little to no risk of changing the network.

An additional feature is that the software implementation, with minor changes, can accept many different input sizes, though they have to be equal in size for both samples and lines. This makes it easy to pass smaller or larger image cubes through the network. Additionally, since the implementation is built around using the BIP format, the network can easily accept image data from the HYPSO-1 hyperspectral imaging payload with the only preprocessing step possibly being dimensionality reduction.

The runtime of the UNet, noticeably a reduced version and using 32-bit floating weights, took quite a while to perform segmentation, with runtimes upwards to 77.27 seconds for an input size of $512 \times 512 \times 102$. This results in a pixel prediction time of $77.27/(512 \cdot 512) \approx 294.76us$. Comparably, the accelerated C-UNet++ implementation in [10] achieved a pixel prediction time of approximately 12.46 us with floating-point image data. Additionally, the original UNet segmented $512 \times 512 \times 1$ in less than a second on a high-end GPU. If one considers that the C-UNet++ utilized 12-bands, far fewer kernels per layer and significantly fewer parameters; the prediction time for the software implementation is not too bad. This shows that reasonable runtimes can actually be achieved even with CPU only implementations.

The implementation manages to keep its memory footprint very small, with the reduced UNet implementation keeping a maximum of 120MB above idle memory usage. Though this starts to be less of an issue with more advanced embedded systems, which have access to larger memories, it can be useful for even more edge case devices with less memory. An example is the HYPSO-1 satellite currently in orbit, which has a maximum memory capacity of 1GB, where approximately 500MB of this is available. Due to the small memory footprint, the application can be uplinked to the satellite with some modifications and be used for testing with a deployed system.

To further add to this, the implementation is very adaptable because it is written in C and for an embedded Linux system. Since many embedded devices run on the Linux OS, the application can be recompiled to the respective architecture using the GCC.

The C++ programming language could have been an alternative option to programming in C, as it adds slightly more functionality such as templates while still providing the same low abstraction level control as C. This would have made parts of the code more general, such as the design of the UNet network, and functions with different data types.

6.4 Hardware Acceleration

The convolutional acceleration design proposed in Chapter 5 aims to decrease the runtime of the software implementation by accelerating convolutional layers. The proposed design is different from previous work [10, 11, 12] because it aims to accelerate large convolutional layers using up to 1024 kernels, each 1024 deep. Additionally, the proposed design aims to be generalized, instead of layer-specific, by allowing weights to be streamed to the accelerator.

First, achieving the task of accelerating large convolutional layers requires a design to have the weights stored within the fabric of the programmable logic. This could, however, be avoided by streaming the weights along with the input data. However, this would introduce the same issue experienced with the input data, which needs to be streamed repeatedly. Therefore, the suggested design explored having the parameters for the largest possible convolutional layer (in the original UNet) stored in the fabric as BRAM. It was, as Chapter 5 suggests, found that a 1024 kernel large layer, with an input feature map with a depth of 1024, would require approximately 18MB with 16-bit quantization. This was within the resource constraint of the UltraScale architecture's FPGA. Additionally, it was seen that 16-bit quantization did not affect the segmentation accuracy in any noticeable way in Section 6.1.

Secondly, the concept of a generalized accelerator will affect the performance, as more resources are needed for generalization. However, with a generalized design, it would be easier to use the accelerator for each convolutional layer throughout the network. This does, however, mean it will have to accept both the largest and smallest possible convolutional layers. This would require a row buffer of $512 \times 512 \times 1024$, as the first layer's largest spatial dimensions would be 512×512. While the feature output at its lowest spatial dimension, e.g., 16×16, would have 1024 bands. This makes for large buffers in the fabric. However, as suggested in Chapter 5 should be viable.

The proposed design would likely reduce the runtime of the software UNet implementation significantly and provide a more energy-efficient design. This assumption is also supported by previous work [12] where a gain in runtime through convolutional accelerators is reported. Though a generalized design would be less effective, it would likely be a better approach than layer-specific accelerators. The proposed hardware design would most likely be beneficial on CubeSats like HYPSO-1 and HYPSO-2, where energy capacity is limited and desired to decrease the runtime. The Zynq-7030 SoC used in the HYPSO-1 does, however, have lower resource constraints

in the form of on-chip memory than the UltraScale. Further research into network quantization would therefore be needed.

Streaming layer-specific weights into the FPGA also circumvents the issues with storing all UNet weights on the FPGA. Additionally, through a hardware/software codesign where one moves the weights to the accelerator at clever points in the processing flow, one can circumvent some drawbacks of moving new weights into the device for every convolutional layer.

The convolutional accelerator is not fully implemented and, therefore, not tested within this thesis. The main issue experienced was with incorrect usage of HLS optimizations within Vitis HLS tool, which caused a long synthesis time. However, a smaller implementation of the DMA control was implemented and tested as it would be useful for the future UNet framework. The Vivado Design Suite from Xilinx was used for the test to set up a DMA running on the UltraScale+ FPGA. Figure 6.14 shows the design where the DMA is connected to the processing system through a AXI Lite interface.

Fig. 6.14: Setup within the Vivado Design Suite for testing DMA transfer between the processing system and FPGA.

For the sake of the test, the DMA was connected to itself with the AXI stream interface. This means that data transferred with the module will be looped back to its destination memory address. Through using the Linux *mmap()* function, one can map the physical address of the AXI Lite interface for the DMA to virtual memory. With this, the DMA could be controlled, and it was managed to transfer data from one memory region to another. This would be beneficial in a future design, as it is an effective way for interfacing with the FPGA within Linux.

7 Conclusion

The work presented in this thesis has shown that it is possible to fit a UNet architecture on an embedded system like the UltraScale+ through a software/hardware approach. Through this, it has shown that large scale CNNs, like the original UNet presented in [8] is possible to run on a cube satellite processing systems, like the one used for HYPSO-1 and in the future HYPSO-2.

Furthermore, a UNet software framework is introduced that is capable of performing semantic segmentation on BIP formatted hyperspectral image cubes. This is proven through semantic segmentation of $512 \times 512 \times 102$ image cubes from the Pavia Centre scene. The UNet implementation, which utilizes convolutional layers with up till 256 kernels, manages to segment image data with a pixel prediction time of around $294.75us$ running on a single core of an ARM Cortex-A53 CPU. It is also shown that the framework can be extended to utilize convolutional layers with up to 1024 kernels.

Additionally, in an attempt to standardize the format of CNN hyperparameters for embedded devices, a new weight format called Weights Interleaved by Filters (WIF) is introduced, which stores the weight values for kernels in a structured manner. It is shown that WIF binary files can be read quickly from disk into memory, allowing for small memory footprints, as weights can be allocated and freed upon desire. The WIF format also splits the weights for specific layers into separate files, making for a more generalized network, as the network can quickly be updated.

Finally, a proposed design for a convolutional accelerator has been provided, showing how the UNet implementation can be further improved through streaming data back and forth from the UltraScale FPGA. The design proposes to stream weight data to the FPGA, along with the image data, allowing for a generalized accelerator that can support any layer up till 1024 kernels.

7.1 Future Work

The UNet framework provided in this thesis will hopefully provide a base CNN network, from where further research and improvements can be conducted. Hope-

fully in a future state, it can be utilized for the HYPSO-1 and HYPSO-2 missions. Additionally, the software framework provided should be relatively easily portable into the HYPSO-2 software.

Due to lack of time, the proposed convolutional accelerator didn't reach a state where it was functional. The main cause of this was the lack of experience with HLS, and the Vitis HLS tool. Someone with more experience, or more time, could most likely learn to implement the suggested design in Chapter 5. Since the UNet software framework is now available, it should provide a good test environment for an accelerated design. Such a design could potentially significantly increase the speed of image segmentation. Code adaptions from work such as [10, 11, 12] could provide useful for creating such an accelerated design.

As shown in [43] quantization of neural networks can speed up processing and decrease weight size. As shown in Section 6.1, 16-bit quantization using the BFloat16 format still provided good segmentation accuracy compared to the models trained on 32-bit. Efficient use of 16-bit fixed-point arithmetic could be explored for thr ARM CPU. Since the architecture supports NEON SIMD instructions [29], this could significantly speed up the CPU implementation.

Though explored in this thesis, compiler and code optimizations could most likely be explored further, as there are many ways to efficient code. An example could be to explore the use of multithreading, as all tests within this thesis looked into single-core execution. Systems like the ARM Cortex-A53 would most likely perform significantly better, as it has four cores. This would make the CPU implementation in itself faster, though if combined with quantization and FPGA acceleration, it could make the system very fast and energy-efficient. Concurrency is however a slippery slope, and could potentially provide issues.

The design proposed in Chapter 5 would most likely use most of the FPGA resources, making acceleration of other layers difficult. However, since flashing FPGAs takes little time, it could be interesting to explore partial reconfiguration or fully re-flashing of the FPGA with separate bitstreams for different CNN layers. E.g., one for convolution and another for transposed convolution.

Testing with different number of bands showed that more bands provided a more accurate segmentation. However, more bands does provide a slight decrease in network speed due to the first convolutional layer. If runtime is of a concern, it might be interesting to explore the reduction of bands as a way to increase runtime. The number of bands does however have little affect on the total number of network parameters.

Bibliography

[1] Michael T. Eismann. *Hyperspectral Remote Sensing*. 2012 (cit. on pp. 3, 9–11).

[2] Mariusz E. Grøtte, Roger Birkeland, Evelyn Honoré-Livermore, et al. "Ocean Color Hyperspectral Remote Sensing with High Resolution and Low Latency-The HYPSO-1 CubeSat Mission". In: *IEEE Transactions on Geoscience and Remote Sensing* 60 (2022) (cit. on pp. 3, 7, 8, 11).

[3] Johan Fjeldtvedt, Milica Orlandic, and Tor Arne Johansen. "An efficient real-time FPGA implementation of the CCSDS-123 compression standard for hyperspectral images". In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 11 (10 Oct. 2018), pp. 3841–3852 (cit. on pp. 3, 34, 38).

[4] Zachary Fasnacht, Joanna Joiner, David Haffner, et al. "Using Machine Learning for Timely Estimates of Ocean Color Information From Hyperspectral Satellite Measurements in the Presence of Clouds, Aerosols, and Sunglint". In: *Frontiers in Remote Sensing* 3 (May 2022) (cit. on p. 4).

[5] "Deep learning". In: *Nature 2015 521:7553* 521 (7553 May 2015), pp. 436–444 (cit. on pp. 4, 13, 17, 19).

[6] Zhuokun Pan, Jiashu Xu, Yubin Guo, Yueming Hu, and Guangxing Wang. "Deep Learning Segmentation and Classification for Urban Village Using a Worldview Satellite Image Based on U-Net". In: *Remote Sensing 2020, Vol. 12, Page 1574* 12 (10 May 2020), p. 1574 (cit. on pp. 4, 10, 37).

[7] Shih Yu Chen, Yu Chih Cheng, Wen Long Yang, and Mei Yun Wang. "Surface Defect Detection of Wet-Blue Leather Using Hyperspectral Imaging". In: *IEEE Access* 9 (2021), pp. 127685–127702 (cit. on pp. 4, 10, 37).

[8] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9351 (2015). Trying to find information on the original UNET algortihm, on it's layer buildup, and potential resource usage in hardware.
, pp. 234–241 (cit. on pp. 4, 13, 14, 18, 36, 37, 43, 46, 53, 109).

[9] Gaetan Bahl, Lionel Daniel, Matthieu Moretti, and Florent Lafarge. "Low-power neural networks for semantic segmentation of satellite images". In: *Proceedings - 2019 International Conference on Computer Vision Workshop, ICCVW 2019* (Oct. 2019), pp. 2469–2476 (cit. on pp. 4, 26, 37, 38, 46, 49, 58).

[10] Sondre Tagestad. "Hardware acceleration of a compact CNN model for semantic segmentation of hyperspectral satellite images". In: *NTNU Open* (2021) (cit. on pp. 4, 5, 37–39, 49, 51, 55, 58, 59, 79, 80, 87, 90, 104–106, 110).

[11] Kristina Andersland Thue. “Accelerating compact CNN models for image segmentation using Vivado HLS”. In: *NTNU Open* (2020) (cit. on pp. 4, 5, 38, 39, 51, 58, 59, 79, 80, 104, 106, 110).

[12] Bing-tong Ren-hao Cai, Li-quan Song, Peng Li, and Bing-tong Zhang. “Acceleration of convolution layer in FPGA of infrared target detection algorithm”. In: *https://doi.org/10.1117/12.2580137* 11563 (5 Nov. 2020), pp. 146–151 (cit. on pp. 5, 38, 59, 79, 106, 110).

[13] “The effect of dimensionality reduction on signature-based target detection for hyperspectral remote sensing”. In: *https://doi.org/10.1117/12.2529141* 11131 (Aug. 2019), pp. 164–182 (cit. on p. 7).

[14] *Transporter-3 Mission - SpaceX - Updates*. `https://www.spacex.com/launches/transporter-3-mission/`. (Accessed on 03/20/2022) (cit. on p. 7).

[15] Kevin G. Sellner, Gregory J. Doucette, and Gary J. Kirkpatrick. “Harmful algal blooms: causes, impacts and detection”. In: *Journal of Industrial Microbiology and Biotechnology* 30 (7 July 2003), pp. 383–406 (cit. on p. 8).

[16] “Global change and the future of harmful algal blooms in the ocean”. In: *Marine Ecology Progress Series* 470 (Dec. 2012), pp. 207–233 (cit. on p. 8).

[17] John R. (John Robert) Schott. “Remote Sensing: The Image Chain Approach”. In: (2007), p. 666 (cit. on p. 9).

[18] Geir Johnsen. “Kelp forest mapping by use of airborne hyperspectral imager”. In: *Journal of Applied Remote Sensing* 1 (1 Dec. 2007), p. 011503 (cit. on p. 10).

[19] Jibo Yue, Haikuan Feng, Xiuliang Jin, et al. “A Comparison of Crop Parameters Estimation Using Images from UAV-Mounted Snapshot Hyperspectral Sensor and High-Definition Digital Camera”. In: *Remote Sensing 2018, Vol. 10, Page 1138* 10 (7 July 2018), p. 1138 (cit. on p. 10).

[20] I El Naqa, M J Murphy, Issam El Naqa, and Martin J Murphy. “What Is Machine Learning?” In: *Machine Learning in Radiation Oncology* (2015), pp. 3–11 (cit. on pp. 13, 15–17, 56).

[21] *Backpropagation | Brilliant Math & Science Wiki*. `https://brilliant.org/wiki/backpropagation/#:~:text=Backpropagation`. (Accessed on 05/20/2022) (cit. on p. 17).

[22] *A Gentle Introduction to Cross-Entropy for Machine Learning*. `https://machinelearningmastery.com/cross-entropy-for-machine-learning/`. (Accessed on 05/21/2022) (cit. on p. 18).

[23] Christopher M. Bishop. “Pattern Recognition and Machine Learning”. In: *Pattern Recognition and Machine Learning* (Dec. 2006) (cit. on p. 18).

[24] *An overview of gradient descent optimization algorithms*. `https://ruder.io/optimizing-gradient-descent/`. (Accessed on 05/21/2022) (cit. on p. 18).

[25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016 (cit. on pp. 19, 22).

[26]"Optimizing CNN-based Segmentation with Deeply Customized Convolutional and Deconvolutional Architectures on FPGA". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11 (3 Dec. 2018) (cit. on pp. 23, 38, 79).

[27]Vincent Dumoulin, Francesco Visin, and George E P Box. "A guide to convolution arithmetic for deep learning". In: (Mar. 2016) (cit. on p. 23).

[28]Xilinx. *ds891 - Zynq UltraScale Plus Overview*. `https://docs.xilinx.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview`. (Accessed on 06/23/2022) (cit. on p. 24).

[29]*Neon – Arm®*. `https://www.arm.com/technologies/neon`. (Accessed on 06/23/2022) (cit. on pp. 24, 33, 110).

[30]Donald G. (Donald Graeme) Bailey. "Design for embedded image processing on FPGAs". In: (Aug. 2011), p. 482 (cit. on pp. 25, 27, 80, 82).

[31]"A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications". In: *ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA* (2012), pp. 47–56 (cit. on pp. 26, 79).

[32]*An introduction to AMBA AXI*. `https://developer.arm.com/documentation/102202/0200/AXI-protocol-overview`. (Accessed on 05/30/2022) (cit. on pp. 26, 27).

[33]*Difference Between Fixed Point and Floating Point - Pediaa.Com*. `https://pediaa.com/difference-between-fixed-point-and-floating-point/`. (Accessed on 05/27/2022) (cit. on p. 28).

[34]*BFloat16: The secret to high performance on Cloud TPUs | Google Cloud Blog*. `https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus`. (Accessed on 06/23/2022) (cit. on p. 29).

[35]Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, et al. "A Study of BFLOAT16 for Deep Learning Training". In: (May 2019) (cit. on pp. 29, 58).

[36]Brian Ward. *How Linux works : what every superuser should know*. 3rd. 2014, p. 437 (cit. on pp. 30–32).

[37]Xilinx and Inc. "Zynq UltraScale+ MPSoC Product Tables and Product Selection Guide". In: (2016) (cit. on p. 31).

[38]*An introduction to GCC for the GNU Compilers gcc and g++*. Vol. 10. Network Theory Limited, 2004, p. 144 (cit. on pp. 32, 33).

[39]*AArch64 Options (Using the GNU Compiler Collection (GCC))*. `https://gcc.gnu.org/onlinedocs/gcc/AArch64-Options.html`. (Accessed on 06/16/2022) (cit. on pp. 33, 100).

[40]"Hyperspectral Image Processing Pipelines on Multiple Platforms for Coordinated Oceanographic Observation". In: *Workshop on Hyperspectral Image and Signal Processing, Evolution in Remote Sensing* 2021-March (Mar. 2021) (cit. on pp. 33, 34, 37, 38).

[41] Gan Zhan, Yutaro Uwamoto, and Yen-Wei Chen. “HyperUNet for Medical Hyperspectral Image Segmentation on a Choledochal Database”. In: (Mar. 2022), pp. 1–5 (cit. on p. 37).

[42] *Hyperspectral Remote Sensing Scenes - Grupo de Inteligencia Computacional (GIC)*. `https://www.ehu.eus/ccwintco/index.php/Hyperspectral_Remote_Sensing_Scenes`. (Accessed on 05/31/2022) (cit. on pp. 41, 52).

[43] Peng Peng, You Mingyu, and Xu Weisheng. “Running 8-bit dynamic fixed-point convolutional neural network on low-cost ARM platforms”. In: *Proceedings - 2017 Chinese Automation Congress, CAC 2017* 2017-January (Dec. 2017), pp. 4564–4568 (cit. on pp. 58, 91, 110).

Simen Netteland

Exploration and Implementation of Large CNN Models for Image Segmentation in Hyperspectral CubeSat Missions

NTNU
Norwegian University of
Science and Technology