Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

Jonas Sundseth

Acceleration of deep learning algorithms for cardiac ultrasound processing by use of Xilinx FPGA

Masteroppgave i Elektronisk systemdesign og innovasjon
Veileder: Per Gunnar Kjeldsberg
Medveileder: Tormod Njølstad, Gabriel Kiss
Juni 2021

NTNU
Kunnskap for en bedre verden

Jonas Sundseth

Acceleration of deep learning algorithms for cardiac ultrasound processing by use of Xilinx FPGA

Masteroppgave i Elektronisk systemdesign og innovasjon
Veileder: Per Gunnar Kjeldsberg
Medveileder: Tormod Njølstad, Gabriel Kiss
Juni 2021

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

Kunnskap for en bedre verden

Assignment

Acceleration of deep learning algorithms for cardiac ultrasound processing by use of Xilinx FPGA.

Deep neural networks (DNNs) have recently achieved impressing results within medical ultrasound imaging. Usually, CPUs or GPUs are used for deployment of the DNN architectures. However, field-programmable logic (FPGA) can be considered as a soft DPU (DNN Processing Unit) suitable for deployment of a diversity of quantized DNN (QNN) architectures. However, for some applications (i.e. portable ultrasound devices) it is desirable to have a small footprint and perform the inference operations with a minimal power budget.

The aim of the proposed MSc thesis is to compare the performance of standard deep learning networks such as view classification and/or segmentation, having as input 2D cardiac ultrasound images. Furthermore, the project will investigate what accuracy, performance and latency are achievable in an embedded solution, under power budget and footprint limitations. Xilinx MPSoC FPGA Zynq Ultrascale+ is selected as target technology. Comparison of inference times between Xilinx FPGA and Nvidia GPUs should also be considered.

Abstract

In recent years the field of deep learning and deep neural networks (DNNs) has evolved and matured as a consequence of better technology in the form of better processing platforms, and more available and bigger datasets. As more research is put into deep learning, more complex models evolve, very often meaning deeper models with more weights and layers. This, in turn, results in more computationally intensive inference, often resulting in lower throughput, higher power consumption, and higher latency for each computed output. This report outlines the implementation of a U-net architecture on a Xilinx XCZU7EV field programmable gate array (FPGA) using quantized weights of 8 bit. The intended application is segmentation of cardiac ultrasound images. The model was trained on several image resolutions to obtain the best trade-off between accuracy and performance. The network was also implemented on central processing unit (CPU) and graphics processing unit (GPU) for comparison in terms of performance and accuracy. The FPGA implementation yielded a maximum speedup of 30x compared to an Intel Core i7 CPU and a maximum speedup of 2.6x compared to an Nvidia GeForce GTX 1060 GPU. The FPGA achieved a latency of 0.07x compared to the CPU and 0.68x compared to the GPU latency. The FPGA model utilizes quantized 8-bit integer weights, whereas the CPU and GPU uses 32-bit floating-point weights. The FPGA implementations resulted in a maximum accuracy reduction of 1% compared to the floating-point models, with all the models optimized for different resolutions achieving Sørensen-Dice coefficients higher than 89%.

Sammendrag

I de siste årene har dyp læring og dype nevrale nettverk utviklet seg som en konsekvens av bedre teknologi i form av bedre plattformer for prosessering og mer tilgjengelige, og større datasett. I takt med at det forskes mer på dyp læring, utvikles det stadig mer komplekse og bedre modeller, hvilket i mange tilfeller betyr dypere modeller med flere lag og vekter. En konsekvens av dette er at propageringen av data gjennom nettverket blir veldig beregningskrevende, og dette fører til lavere utførelseshastighet (eng:throughput), høyere energiforbruk og lengre ventetid (eng:latency). Denne rapporten sammenfatter en implementasjon av en U-net arkitektur på en Xilinx XCZU7EV FPGA ved bruk av 8-bits kvantiserte vekter. Det tiltenkte bruksområdet er segmentering av kardiologiske ultralydbilder. Modellen er trent for ulike bildeoppløsninger for å finne det beste kompromisset mellom nøyaktighet og ytelse. For å kunne sammenligne ytelse og nøyaktighet ble nettverket også implementert på en CPU og en GPU. FPGA-implementeringen gir en maksimal hastighetsøkning på 30x sammenlignet med en Intel Core i7 CPU og en maksimal hastighetsøkning på 2.6x sammenlignet med en Nvidia GeForce GTX 1060 GPU. FPGA-implementeringen oppnår en ventetid på 0.07x sammenlignet med CPUen og en ventetid på 0.64x sammenlignet med GPUen. FPGA-implementeringen bruker kvantiserte 8-bits heltallsvekter, i motsetning til CPUen og GPUen som bruker 32-bits flyttalsvekter. FPGA-implementeringen resulterte i en maksimal reduksjon av nøyaktighet på 1% sammenlignet med flyttalsmodellene, hvor alle modellene optimalisert for de ulike bildeoppløsninger oppnår en Sørensen-Dice koeffisient større enn 89%.

Preface

A common practice in the programming community is that when one encounters a bug, someone has likely encountered the same bug before and has requested help in a forum online. In most cases, they got a good answer. Similarly, if one wants to solve a great problem, there is a great probability that someone attempted to solve the same problem. During the different phases of my master thesis, I have been through both scenarios multiple times. What this means in practice is that parts of the codebase presented in this report are either inspired by, heavily inspired by, or borrowed from other developers. I recognize their tremendous effort in solving these problems and I have done my best to acknowledge and make reference to other developers' source code and contributions.

To begin with, I would like to thank my three supervisors who have led me through this thesis and offered great help all along the way. First, I would like to thank Tormod Njølstad for his unique eagerness to achieve great results and his dedication through weekly meetings with me. Furthermore, I would like to thank Per Gunnar Kjeldsberg for reviewing my report and giving great pointers and advice on both doing and writing a master thesis. Finally, I would like to thank Gabriel Kiss, who provided a handful of useful insights on the AI side of things, reviewed the theory section of my thesis, as well as ideas for the implementation, and introduced me to useful snippets of code.

Lastly, I would like to use this opportunity to mention a good friend and former classmate, Anders Austlid Taskén. He provided me with excellent knowledge and insights regarding implementation of DNNs in Python, as well as helped me by pointing out a good direction regarding training strategy.

Contents

Acronyms vii

List of Figures viii

List of Tables ix

1 Introduction 1
- 1.1 Objectives and limitations . . . 1
- 1.2 Main contributions . . . 2
- 1.3 Report structure . . . 2

2 Theory 3
- 2.1 Artificial intelligence . . . 3
- 2.2 Machine learning . . . 3
- 2.3 Neural Networks . . . 4
- 2.4 Convolutional neural networks . . . 6
- 2.5 Training . . . 10
  - 2.5.1 Data Augmentation . . . 12
  - 2.5.2 Preprocessing . . . 13
- 2.6 U-Net . . . 14
- 2.7 Cardiac ultrasound . . . 15
- 2.8 Evaluation . . . 17

3 Previous work and background 19
- 3.1 Accuracy . . . 19
- 3.2 Computing platforms . . . 19
- 3.3 Inference acceleration . . . 20
- 3.4 PyTorch . . . 21
- 3.5 Vitis AI . . . 22
  - 3.5.1 Quantization . . . 23
  - 3.5.2 DPU . . . 24

4 Implementation 27
- 4.1 A novel design approach . . . 27
- 4.2 Network implementation . . . 28
  - 4.2.1 Preprocessing . . . 29
  - 4.2.2 The network . . . 30
  - 4.2.3 Training . . . 31
  - 4.2.4 Post processing . . . 33

4.3 Porting PyTorch network to FPGA 34
4.4 Optimizating DPU 36

5 Results **37**
5.1 Accuracy 37
5.2 Performance: throughput and latency 38

6 Discussion **41**
6.1 Accuracy impact 41
6.2 Computing platform 41
6.3 Applications 42

7 Conclusions **45**

References **46**

Appendix **50**
A Workflow of generating FPGA models 50

Acronyms

AI artificial intelligence. 1, 3

ASIC application-specific integrated circuit. 19

CNN convolutional neural network. 1, 3, 6–8, 12, 14

CPU central processing unit. ii, ix, 1, 2, 19–22, 25, 27, 28, 33–42, 45

DNN deep neural network. ii, iv, 1–3, 6, 9–11, 16, 19–22, 24, 27, 28, 35, 41, 42, 45

DPU deep learning processing unit. 19, 22, 24, 25, 28, 36

FPGA field programmable gate array. ii, viii, 1–3, 17, 19–22, 24, 25, 27–29, 33–43, 45

fps frames per second. 18, 20, 27, 38, 39, 42, 43

GPU graphics processing unit. ii, 1, 2, 19–22, 27, 28, 31, 33, 35, 37–42, 45

HDL hardware description language. 21

HLS high-level synthesis. 21, 43

IP intellectual property. 29

ISA instruction set architecture. 22, 25

ML machine learning. 3, 11, 16

ReLU rectified linear unit. 5, 9, 14, 15, 23

List of Figures

2.1 Realtion between AI topics 3
2.2 Example of a simple neural network topology. Modified from [5]. 5
2.3 Visualization of how a neuron works. 6
2.4 Graphical simplification of convolution and pooling. Modified from [13]. 7
2.5 Fully connected compared to weight sharing. Modified from [6]. 8
2.6 Visualization of translation and pooling. Modified from [6] 9
2.7 Visualization of different strides. Modified form [14]. 9
2.8 Max pooling [6]. 10
2.9 The process of training. 11
2.10 Interpolation techniques. [20] 13
2.11 The original U-Net architecture. [18]. 14
2.12 $3x3$ convolution and its transposed. Modified from [14]. 15
2.13 A diagram of the different regions of the heart. Modified from [23]. 16
2.14 Sample from testing dataset. 17
3.1 DPU hardware architecture [4]. 25
4.1 Decision tree for chosing an architecture for FPGA inference. 27
4.2 Visualization of the preprocessing stages 29
4.3 Visualization of preprocessing example 30
4.4 The modified version of the U-Net architecture, here shown with an input size of 128x128. Modified from [18]. 31
4.5 Error estimates during training for 128x128 33
4.6 Merging segmentation masks using argmax 34
4.7 Visualization of dataflow 35
5.1 Computed segmentation mask from the sample in Figure 2.14 inferred on the FPGA 38

List of Tables

4.1 Augmentation algorithms used and their parameters 32
5.1 The hardware used to obtain benchmarks. 37
5.2 Model accuracy 38
5.3 Model throughput 39
5.4 Model latency 39
5.5 Performance comparison relative to the CPU implementation 40

This page was intentionally left blank.

1 Introduction

Ultrasound imaging is extensively used in medical diagnosis, disease monitoring, treatment planning, and prognosis. One of the most prominent reasons for this is the fact that ultrasound offers a non-invasive qualitative and quantitative assessment [1]. Segmentation of cardiac ultrasound images is a principal first step in several medical applications [1]. Segmentation can be explained as classifying subregions in an image by assigning each pixel a class, considering for example an application segmenting a car from the background, the network will assign each pixel one of two classes depending on whether the network believes the pixel belongs to the car or the background. Automating the task of segmenting cardiac ultrasound images with high accuracy could be beneficial and free up time and effort as these tasks are performed manually by specialists [2]. Introducing artificial intelligence (AI) into the cardiac ultrasound pipeline could also make less trained professionals able to perform such tasks. The segmentation masks can be utilized in the estimation of various cardiac indices such as the left ventricle volume [2].

This report will describe the implementation of a convolutional neural network (CNN) for a real-time application trained on segmenting cardiac ultrasound images. For an implementation to be suited for real-time applications, there are a number of characteristics to keep in mind during the design phase. For the design to be a viable option, the design should focus on minimizing latency, power consumption and size, while maximizing throughput [3][4]. In addition to this, we introduce downsampling and upsampling as part of the pre- and post-processing which will add additional speedup [5]. Pre- and post-processing refers to manipulating images before and after they are fed to the network. The importance of these metrics is relatively straightforward. The power consumption is vital due to both the environmental impact and maximizing battery life for battery powered systems, such as a handheld ultrasound device. A real-time system requires a satisfactory degree of responsiveness from the user, which is related to throughput and latency. If, for example, the CNN is part of the data flow in a real-time ultrasound image application, the resulting segmentation should be produced so that the user can respond to the provided output within a reasonable time. In order to fulfill the mentioned criterion, one could implement the CNN on an FPGA as the compute platform. This is due to a combination of the ability to generate specialized hardware, good power efficiency, and the possibility of reprogramming.

1.1 Objectives and limitations

The goal of this report is to speed up inference of a CNN using an FPGA as inference engine using quantization of weights. The aim is to implement a model with performance comparable to a GPU, with as low accuracy reduction as possible. We wish to obtain a network with high accuracy before quantization. We intend to obtain a qualitative measure of the different compute platforms and investigate whether FPGA is a viable option as compute platform for DNN for ultrasound segmentation applications.

The intended application is DNN inference, and we will therefore not include the pre- and post-processing in the benchmarks as these are run on CPU in all the implementations. Both stages could have easily be implemented in hardware on the FPGA.

1.2 Main contributions

The findings in this report are summarized in the list below:

- An approach towards finding a suitable DNN model for FPGA inference
- An implementation of the U-net architecture suited for cardiac ultrasound segmentation, with performance comparable with a GPU.
- Investigation of the impact on accuracy when using input downsampling and quantization as means of acceleration.

1.3 Report structure

In Chapter 2 we provide a thorough theoretical background of neural networks in general before narrowing towards more relevant theory regarding the implementation as well as DNN inference on alternative computing platforms. In Chapter 3 we present the background and previous work done in the field of cardiac ultrasound segmentation as well as FPGA inference of DNNs. The resulting implementation is presented in Chapter 4, and its performance and accuracy compared to CPU and GPU are presented in Chapter 5. The results and their relevance to the given application are then further discussed in Chapter 6 before we finally present the conclusions of the findings in Chapter 7.

2 Theory

This report outlines the implementation of a CNN for segmenting cardiac ultrasound images in a real-time system. Both CNNs and the segmentation application will be discussed in greater detail throughout this chapter. This chapter begins by offering basic theory of AI and machine learning (ML) before providing a more in depth description of DNNs, CNNs, cardiac ultrasound, FPGA inference of DNNs, and qualitative evaluation of DNN models on different computing platforms.

2.1 Artificial intelligence

During the past hundreds of years, inventors have dreamed about creating machines able to think and learn. With today's technology and knowledge, this is feasible through what is popularly called AI. In recent years the field of AI has grown substantially as a consequence of better technology, and larger and more available datasets [6]. AI has several practical applications. Intelligent software is used to automate routine labor, understanding complex data such as images and speech, natural language processing, medicine, support scientific research, finance, vision, image classification, and automated driving [6][7][8].

This report will cover several topics within AI, so in order to better understand how the major topics are connected, we refer the reader to Figure 2.1. The main topic of discussion in this report will be CNNs.

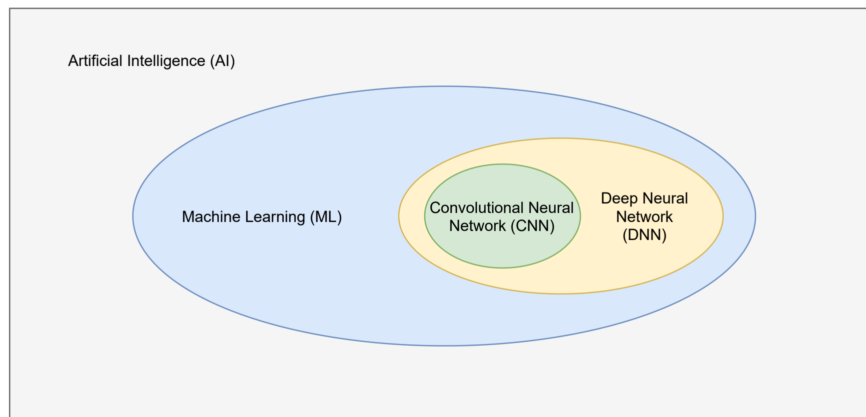


Figure 2.1: Realtion between AI topics

2.2 Machine learning

The field of ML is a part of computer science. ML can be described as algorithms which are optimized to perform a given task by providing them with large amounts of data which the given algorithm uses to train on [5][8], we will go into greater detail on training in Section 2.5.

Machine learning algorithms are made to detect patterns in data using statistical models, and use the knowledge learned during training to make decisions on unseen data [5]. There are several types of ML algorithms to chose from depending on the application. Some examples include: decision trees, K-means clustering, support vector machines, k-nearest neighbor, and neural networks [9].

2.3 Neural Networks

One type of machine learning algorithm which has become very common is neural networks. It is also sometimes referred to as multilayer perceptrons, or feedforward networks [5][6]. The applications for these types of machine learning algorithms are almost unlimited and vary greatly, proving how these models are good at generalization over many different applications and, in some cases, also exceeding human-level performance [6][8].

In order to explain how a neural network works, we will consider a relatively simple neural network topology shown in Figure 2.2. The nodes in the graph represent the neurons containing intermediate values in each layer; the first column represents the input, and the last representing the output. There are several hidden layers in between the input and output, here represented as one layer of nodes. The edges between the nodes illustrate weights and biases, where the biases are denoted with subscript 0 and are located at the bottom in the figure. If we were to use such a network on image classification, one would have one input node for each pixel, i.e., for an image of size $128x128$ one would need $D = 128^2 = 16,384$ input nodes [5][6]. The output will be a target vector with K entries containing probabilities of the given input to belong to a given class. As an example, we will consider a neural network classifying handwritten digits, for example, by using the famous MNIST dataset [5][10]. The dataset contains 60,000 images of handwritten digits of size 28x28 [10]. If the example network were to classify digits based on the MNIST dataset, we would have $28x28 = 784$ input nodes and ten output nodes, one for each class $k = 0, 1, ..., 9$. The ten resulting outputs will indicate the probability of the input image being of that specific digit, and one would intuitively classify the input image to the class of highest probability.

Using the neural network topology from Figure 2.2 as an example, the model is a small, fully connected neural network, meaning that all the nodes in two adjacent layers are connected [5]. The first layer creates M linear combinations of the inputs $x_1, x_2, ..., x_D$, where M is the number of nodes in the first hidden layer and is an arbitrary number chosen during the design of the neural network architecture, and D is the number of input nodes. The inputs are propagated to the next layer using Equation 2.1.

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{2.1}$$

We will treat the hidden layers as one, though this is seldom the case. Each node in the hidden layer gets its input from the activations from the previous layer. The activation is the obtained value after the linear combination from the preceding layer. The input activation to node j in the hidden layer is denoted as a_j in Equation 2.1. The term w_{ji} is the weight on the edge from node i to node j in the following layer, whereas w_{j0} is the bias term [5]. The superscript (1) denotes that the weights belong to the first layer. In order to better understand how a neuron works, we refer the reader to Figure 2.3. The inputs are multiplied by the weights, and the bias is added, as expressed in Equation 2.1, after that the activation is run through an activation function.

The activation a_j are not always directly propagated to the succeeding neurons in the next layer, in many cases an activation function is deployed [5][6]. When using an activation function the resulting value is computed using a_j defined Equation 2.1 as shown in Equation 2.2.

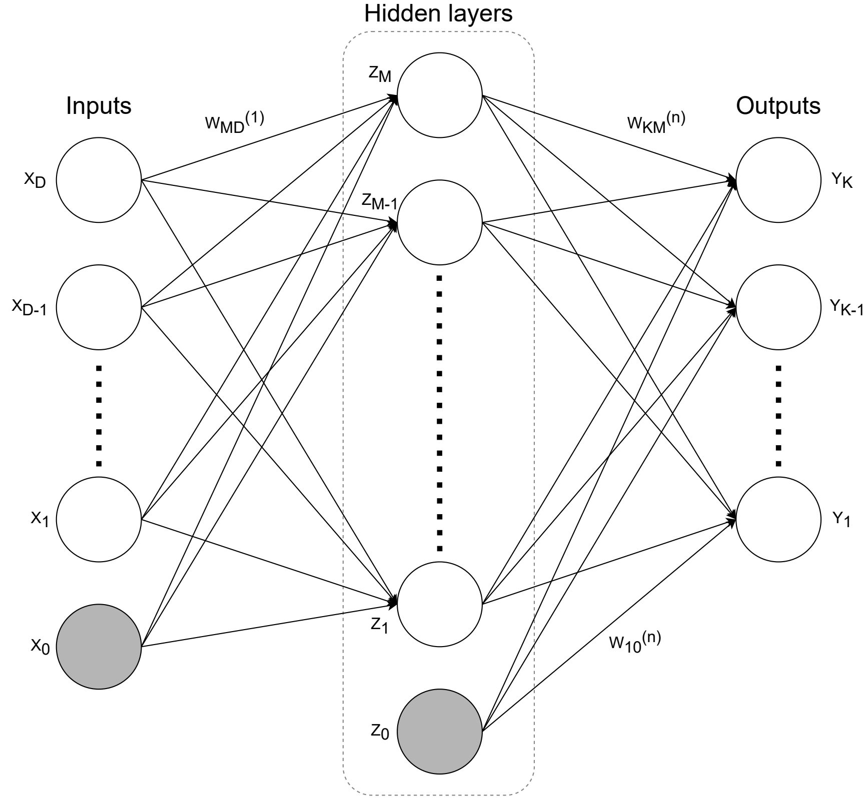


Figure 2.2: Example of a simple neural network topology. Modified from [5].

$$z_j = h(a_j) \tag{2.2}$$

Activation functions are used in order to transform the activation levels of a neuron to an output [11], it enables the model to make sense of non-linear mappings between the inputs and the corresponding outputs [12]. There are several options regarding which activation functions to use. One of the most common is the sigmoid function shown in Equation 2.3.

$$\sigma(a) = \frac{1}{1 + \exp(-a)} \tag{2.3}$$

The sigmoid is a rather complex function that is not always necessary and could, in some cases, require a longer training time. A more recent and simpler activation function is the rectified linear unit (ReLU) shown in Equation 2.4 [8].

$$R(a) = max(0, a) \tag{2.4}$$

The ReLU is becoming more common as it is both less computationally complex and is known to converge faster during training than the sigmoid [8]. There are several other options to which activation function to chose; we will only mention the sigmoid and the ReLU. The distinct layers can also have different activation functions.

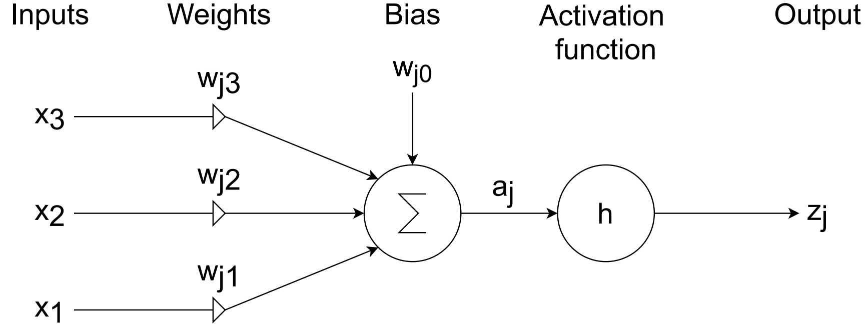


Figure 2.3: Visualization of how a neuron works.

In order to produce the result, the activations from the hidden layers are finally propagated to the output layer. The output is computed from the activations from the last hidden layer using Equation 2.5.

$$a_k = \sum_{j=0}^{M} w_{kj}^{(n)} z_j + w_{k0}^{(n)} \tag{2.5}$$

In this equation a_k denotes the activation of the input to be of class $k = 1, 2, ..., K$, where K is the total number of classes in the given application, which in turn, in most cases is processed by an activation function before outputted as a probability. Similarly to Equation 2.1 the variable w_{kj} denotes the weight used to process the value from node j to k and the weight w_{k0} represents the bias term for class k. M is the number of nodes in the last hidden layer, and z_j is the output from the activation function from the jth node in the previous layer. Here each output neuron will produce a probability for whether or not the input belongs to the given class k [5].

The term DNN refers to neural networks with many layers and weights. These have become more popular in recent years due to a general increase in the availability of processing power [6]. These networks have proven to produce good results and are therefore growing in popularity in the machine learning community. However, training large networks requires enormous amounts of data in order to gain sufficient invariance to generalize on data [5][6].

2.4 Convolutional neural networks

The term convolutional neural network suggests the use of convolutional operations. CNNs are neural networks that use convolution instead of the more general matrix multiplication at least in one of its layers [6]. The convolutional operation used can be expressed as in Equation 2.6.

$$a_{i,j} = \sum_{m=0}^{L-1} \sum_{n=0}^{L-1} k_{m,n} \cdot x_{i+m,j+n} \tag{2.6}$$

The term $a_{i,j}$ is the activation in a feature map at index i, j. $x_{i,j}$ is the input at index i, j and k is the convolutional kernel, and L is the kernel size, which will be described in greater detail further down in this section. CNNs are commonly used in image-based deep learning or similar applications. This is due to its

unique ability to quantify patterns using the convolutional layers, thus making it ideal for image applications [5].

CNNs exploit the fact that nearby pixels tend to be correlated. The CNN extracts local features by analyzing subregions of the image [5][6]. These local features are searched for through the whole image and thereafter merged in later stages of the inference to detect higher-order features in the image. As a consequence of this, the CNN becomes much more robust against translation, scaling, small rotations, and elastic transformations, which might not have been seen during the training phase [5]. As an example, we will consider the aforementioned MNIST dataset used to classify handwritten digits. Even though a digit might be shifted a bit, mildly rotated, or changes size, it should, in most cases, be classified as the same digit.

There are mainly three mechanisms that contribute to this robustness in CNNs; sparse interactions, weight sharing, and subsampling [5][6]. We will go into further detail into all three, starting with the sparse interaction and receptive fields.

Fully connected neural networks use a matrix with separate weights for each connection of nodes, CNNs however, use what is called sparse weights. This is done by making the kernel smaller than the input image. An input image might, for example, consist of thousands of pixels, but it is still possible to detect meaningful features such as edges by using a 3x3 kernel [6]. This significantly reduces the number of weights needed to be stored compared to a fully connected network [6]. In the case of a 3x3 kernel, one would store ten weights, including the bias term [5]. In addition to this, the network requires fewer operations to produce the output [6]. There are typically a number of such kernels extracting features into what are called feature maps. This is shown in Figure 2.4 where M kernels of size kxk, smaller than the input HxW extracts feature to the M feature maps. The depth D in Figure 2.4 would typically be the number of channels in an image application, for example, RGB in a color image.

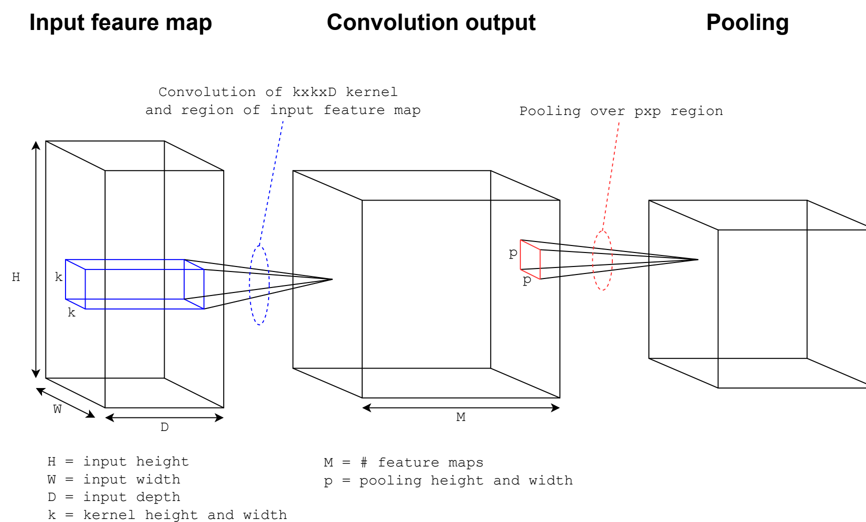


Figure 2.4: Graphical simplification of convolution and pooling. Modified from [13].

In a fully connected layer, each weight is used precisely once per input. It is multiplied by one element of the input and thereafter not used before a new input is computed [6]. On the other hand, in a convolutional

neural network, each entry in a kernel is used at every position of the input, except boundary pixels depending on the CNN architecture [6]. As a consequence of the weights being shared, the network does not have to learn unique weights for each input location, which reduces the storage requirements [6]. This is visualized in Figure 2.5, where the edges between the nodes represent weights. First and foremost, we see that the number of weights is much lower due to the sparse weights. In addition to this, the weights going out from each node in the bottom layer in the convolutional layer are equal for each node, whereas they are all unique in the fully connected layer [6]. Furthermore, for each input node, there are fewer output nodes affected in the convolutional layer compared to the fully connected layer, as visualized by the green nodes in Figure 2.5. This is also useful when regarding image applications, one would, for example, detect vertical edges in the first layers of the image, but the image will typically have multiple vertical edges throughout the whole image frame [6], therefore making it beneficial to share the weights detecting the edges on the whole input frame.

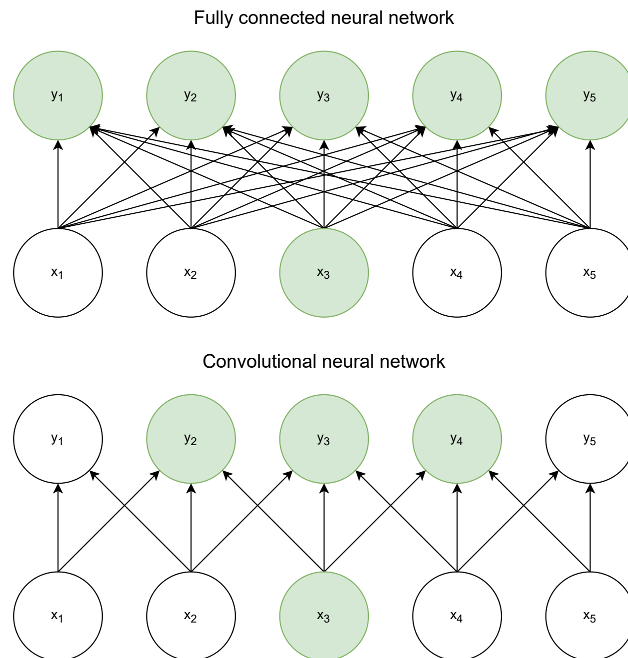
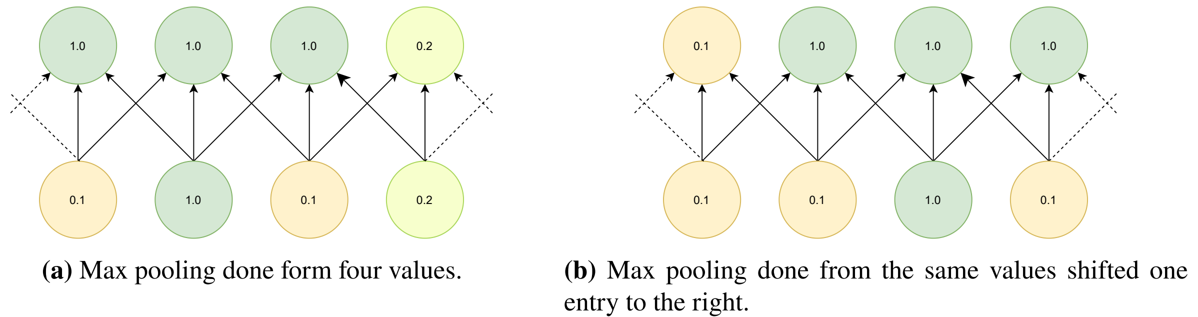


Figure 2.5: Fully connected compared to weight sharing. Modified from [6].

Finally, subsampling contributes to CNNs being robust against translations. This is because the convolutional operation is equivariant to translations, meaning that if a feature in the input is shifted, the resulting feature will be equally shifted in the resulting feature map. This equivariance and the subsampling make the network less prone to error when input images are slightly translated. This is visualized in Figure 2.6, where the outputs are subsampled using a max-pooling operation. Here the output of several nearby pixels is combined [6]. Pooling can be explained as a summary of statistics of nearby outputs [6]. There are several options as to how the pooling is performed. The most common methods include max-pooling, which reports the maximum output within a rectangular area, average pooling, and weighted average based on the distance from the central pixel within a rectangular area [5][6]. The pooling of outputs makes the network approximately invariant to small translations in the input, meaning that if the features in an image are slightly

moved, the change in the outputs can be considered negligible [6]. This is visualized in Figure 2.6a and 2.6b.

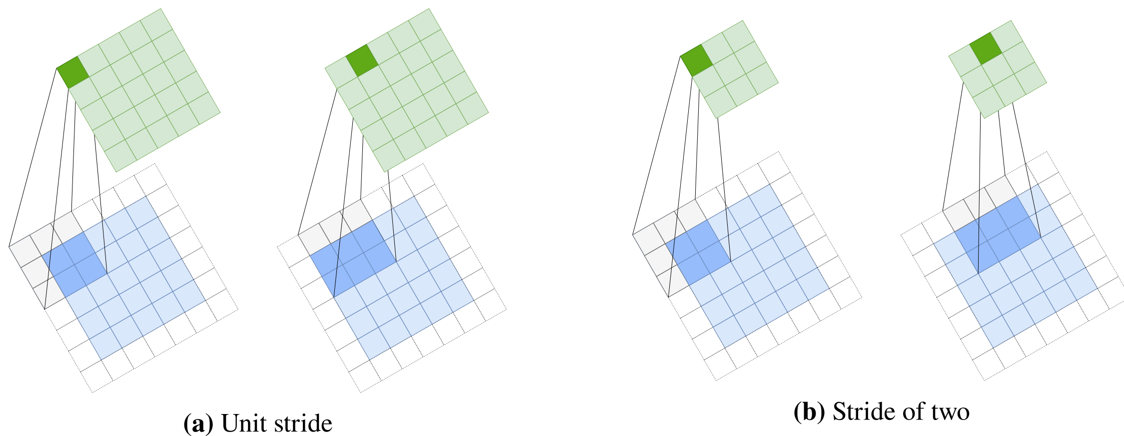


(a) Max pooling done form four values.

(b) Max pooling done from the same values shifted one entry to the right.

Figure 2.6: Visualization of translation and pooling. Modified from [6]

The different types of layers are often defined by hyperparameters, such as the already mentioned kernel size, kernel type, padding, and activation function. An important example of a hyperparameter is the stride. Stride refers to the distance between two consecutive positions of a kernel [14], for example, during convolution or pooling. Strides can also act as a form of subsampling and can be viewed as how much of the input is retained [14]. This is shown in Figure 2.7 where a $3x3$ window is computed; this could, for example, be a convolution kernel. The input is zero-padded with one layer of zeros. We see that the input of $5x5$ retains its size in the case of the unit stride while using a stride of two results in an output of size $3x3$.



(a) Unit stride

(b) Stride of two

Figure 2.7: Visualization of different strides. Modified form [14].

A convolutional layer usually consists of three stages, namely convolution, detection, and pooling. The first stage entails the already mentioned convolution in order to produce a set of linear activations using several kernels [6]. Then, similar to the fully connected neural network the activations are run through an activation function, most often a ReLU. This stage is often referred to as the detection stage [6]. The detected features are thereafter pooled using, for example, max pooling. This flow is visualized in Figure 2.8 where we could, for example, have three kernels detecting different variations of the digit '5', and depending on the input image, the kernels would obtain different activations. However, after pooling, the activation for the digit '5' would end up in the same pooling unit and produce a high probability.

It is, of course, possible to use a fully connected DNN on an image application. However, this would

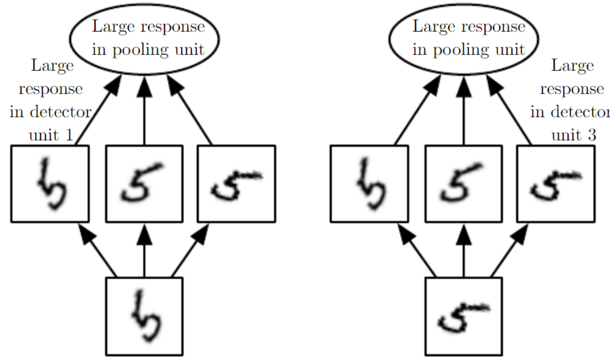


Figure 2.8: Max pooling [6].

require much more training to achieve similar levels of accuracy [5].

2.5 Training

Before going deeper into the learning algorithms, we will discuss the principles used during training and to qualitatively evaluate the obtained weights. There are several different approaches to learning, such as supervised-, unsupervised- and reinforcement learning. Supervised learning refers to learning where the training data consists of inputs and corresponding targets. On the other hand, Unsupervised consists of inputs without targets, where the goal is to discover groups of similar examples within the data. Finally, reinforcement learning is concerned with finding suitable actions to take in a given situation [5]. This report uses the approach of supervised learning.

We will continue using the example of digit classification. During training, the network is fed training data along with the labels for each sample. In the context of digit classification, this would be a number of images and a corresponding label telling which digit is in the given image so that the network is able to verify whether the output is correct or not while continuously trying to minimize the error [6]. The interesting metric to obtain here is not how good the model performs on the training data, but how good it performs on unseen data [6][10]. Therefore we use an additional dataset containing similar samples as in the training set, but we do not feed them to the model during training. This is called the testing dataset, and it is used afterward to quantify how good the accuracy is on unseen data samples [10]. Theory and experiments have shown that the accuracy gap between the data in the training and testing set is decreased with an increased number of samples in the training set [5][6][10]. Additionally, when training DNNs it is necessary to take out some samples from the training dataset to use for validation. During training, the weights obtained are evaluated using the validation dataset to find the best set of weights [6].

We will not go into great detail on how the training is done mathematically, as most of these algorithms can be easily implemented using different libraries. The method most widely used to train DNNs is called statistical learning and is done by using a method called backpropagation [5]. The main idea behind backpropagation is to obtain an error function $E(\boldsymbol{w})$ where $\boldsymbol{w}$ denotes the weights and biases. The goal of backpropagation is to minimize this error function [5]. The backpropagation algorithm consists of two stages. The first stage entails an evaluation of the derivatives with respect to $\boldsymbol{w}$, thereafter the obtained

derivatives are used to find a suitable adjustment for the weights in each layer, for example, by using gradient descent [5]. There are several different methods to train ML applications depending on what model is used; as we are mainly concerned with DNNs we will limit this section to describe theory relevant to this. In order to better understand the concepts of training, we refer to Figure 2.9 for a visualization of the training procedure.

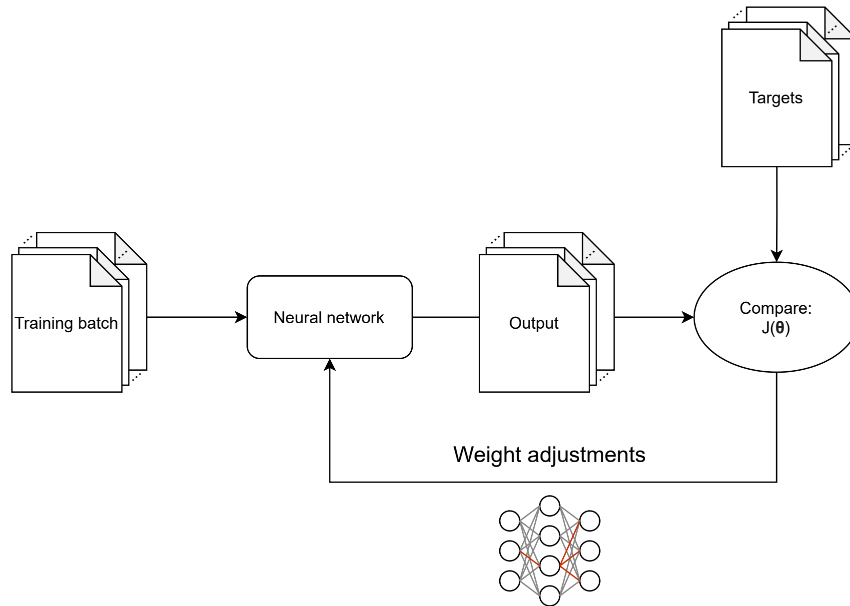


Figure 2.9: The process of training.

To optimize the accuracy of the model, we use a cost function $J(\boldsymbol{\theta})$ to quantify the error of the estimated outputs compared to the target output, also sometimes referred to as ground truth [6]. Thereafter the weights are slightly altered based on the gradient of the cost function, and the procedure is repeated [5][6]. The loss function will be chosen based on the application. Nearly all deep learning algorithms are optimized using what is called stochastic gradient descent [6]. The cost function is evaluated as a the sum of the per sample loss for the whole training set [6].

An alternative to computing the gradient descent using per sample loss is to use batches of training samples [5]. When optimizing using training data, one typically chose a number of input samples to forward when evaluating the gradients. This could either be one sample at a time, meaning a batch size of 1, which is often referred to as stochastic training [6]. It is also possible to use larger batch sizes to compute the gradient. The main limiting factor to the size is the memory of the system the training is performed on, which depends on what type of input is used. The batch size can be defined as the number of images used to train a single forward and backward pass of the network. Training on small images requires less memory, which means one can use larger batch sizes during training [6]. There are several reasons as to why one would use both larger and smaller batch sizes. Larger batch sizes provide a more accurate estimate of the gradient but with linear returns. It is also beneficial to use batches in multi-core architectures as the training is performed faster as they can often be computed in parallel [6]. In addition to this, a larger batch size results in higher

recognition accuracy and generalization [15], though this is also highly dependent on the optimizer used as well [16]. In some cases, smaller batch sizes allow for better finetuning and can also have a regularizing effect on the training [6][16].

2.5.1 Data Augmentation

In order to avoid overfitting CNNs rely on large datasets. However, this is not always available, especially in medical imaging, the available data is very limited [17]. Overfitting is explained as the case where a model learns a function with high variance so that it performs exceptionally on the training data, but not on the testing data [17]. In order to avoid overfitting and generalize better on the training data, one can deploy a technique called data augmentation on the training data. Data augmentation refers to several techniques that enhance the size and quality of the training dataset [17][18]. Simply put, data augmentation creates fake data based on existing samples and adds it to the training dataset [6]. Data augmentation is not applicable to any application but is highly relevant for image applications. This is because images are highly dimensional and have a vast range of variations, which in many cases is easy to simulate. For example, translating the image will increase generalization even though the model is designed to handle these variations using convolution and pooling [6]. We will look closer at a selection of data augmentation algorithms taken from Shorten and Khoshgoftaar [17] below.

A very simple data augmentation algorithm is flipping the input and might be the simplest augmentation algorithm. The most common way is to do horizontal flipping. Another simple data augmentation algorithm is random cropping; this refers to cropping the image randomly while preserving the most important contents of the image frame. This will provide an effect similar to translation. The translation augmentation shifts the image in some direction in order to reduce positional bias. The main difference between random cropping and translation is that the cropping algorithm does not preserve the spatial dimensions. Other popular techniques include random rotation, this simply entails rotating the image randomly some angle between $\pm\theta_{max}$, where typical values are $\theta_{max} \leq 20°$. Noise infusion is a data augmentation algorithm that consists of adding an array of the same size as the image containing noise drawn from a Gaussian distribution; this is shown in Equation 2.7.

$$\boldsymbol{X}_{noisy} = \boldsymbol{X} + \boldsymbol{w}, \tag{2.7}$$

where $\boldsymbol{w} \sim \mathcal{N}(0, \sigma^2)$ is zero-mean Gassian white noise with variance of σ^2. The last augmentation algorithm we will present is the gamma augmentation. Gamma augmentation adjusts the brightness of the image using Equation 2.8:

$$\boldsymbol{X}_{gamma} = c \cdot \boldsymbol{X}^{\gamma}, \tag{2.8}$$

where c and γ are coefficients adjusting the intensity in the image. These augmentation methods mentioned above are usually employed randomly in the training set, and one might also choose to use several of these on one sample. Furthermore, these augmentations will be deployed independently over the epochs of training, meaning that for a given sample, the augmentation algorithms used will be different from epoch to epoch.

An epoch refers to the number of passes of the training dataset during training.

One has to take precautions when utilizing such algorithms. The resulting augmented samples should make sense afterward [6]. Consider the digit classification application; if one were to use rotation as an augmentation algorithm, one should not use too great of an angle. For example, the digit '6' could become a '9' and vice versa if rotated close to 180°. Furthermore, it does not make sense to use augmentations mirroring the samples as these are no longer valid digits in most cases. When using data augmentation in segmentation algorithms, one must also consider that augmenting the input sample might render the ground truth segmentation incorrect. In that case, one has to perform the augmentation on both the sample and the ground truth; for example, if rotating an image, the ground truth segmentation must be rotated equally. In contrast to this, if for example, inducing noise in the sample, one should not add noise to the ground truth.

2.5.2 Preprocessing

After data augmentation and before data are fed into the network, the samples are typically preprocessed. A typical step in the preprocessing of input samples is to perform downsampling to have smaller images to achieve a faster inference, as the number of pixels to process per second decreases [5]. In addition to this, the dataset might consist of images of varying sizes, for example, such as the Camus dataset [2]. There are several methods for downsampling an image. We will not go into greater detail on this except for some standard interpolation techniques. There exist several libraries which do this elegantly, such as, for example, OpenCV [19].

There are a number of ways to downsample an image. If we consider the OpenCV library, the three most common interpolation methods include linear, cubic, and nearest neighbor and are used for both up- and downsampling images. The differences between the interpolation techniques are visualized in Figure 2.10. Which method is the ideal is highly dependent on the given application [19] and is beyond the scope of this report.

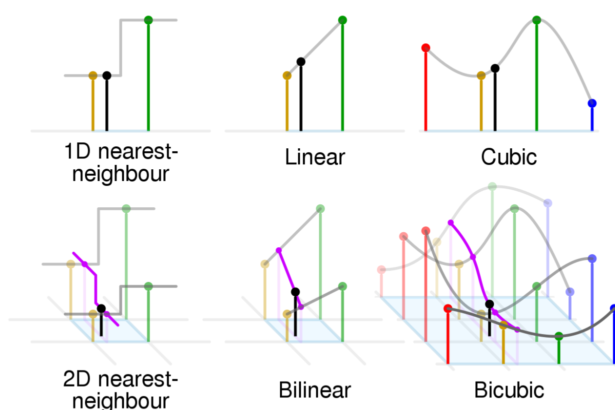


Figure 2.10: Interpolation techniques. [20]

In order to make the model more robust against variation in the data samples, it is common to either standardize or normalize the images [21]. Normalization refers to scaling the input to the range of $[0, 1]$. This is shown in Equation 2.9.

$$\boldsymbol{X}_{normalized} = \frac{\boldsymbol{X} - \boldsymbol{X}_{min}}{\boldsymbol{X}_{max} - \boldsymbol{X}_{min}} \tag{2.9}$$

$\boldsymbol{X}$ is the input image before normalization, and the max and min subscripts refer to the smallest and largest intensity value in the image. An alternative to normalization is standardization, shown in Equation 2.10.

$$\boldsymbol{X}_{standardized} = \frac{\boldsymbol{X} - \mu}{\sigma} \tag{2.10}$$

The $\boldsymbol{X}$ denotes the input, μ and σ are the mean and standard deviation of $\boldsymbol{X}$ respectively. The standardization also helps to reduce the impact of variations from the data acquisition and to improve the reproducibility [21].

2.6 U-Net

Ronneberger, P.Fischer, and Brox [18] implemented a CNN called U-net. The name comes from its signature shape as shown in Figure 2.11. U-net is a well known CNN architecture used to segment images, and has proven to be well suited for segmentation of ultrasound images [18].

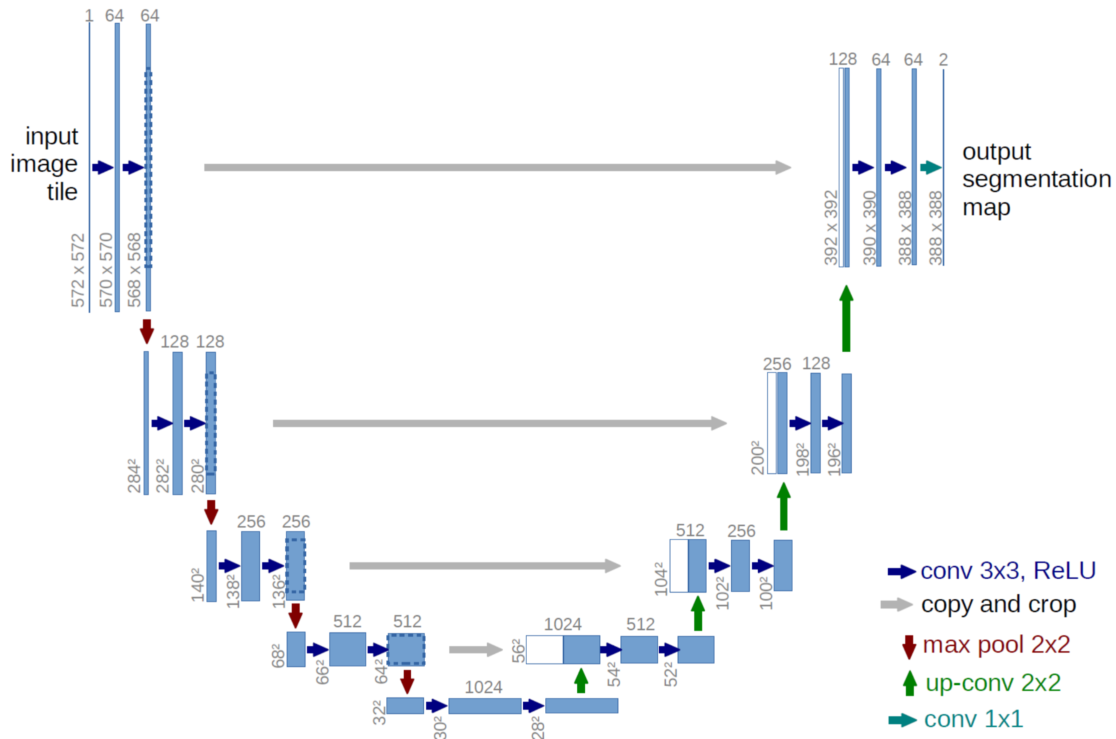
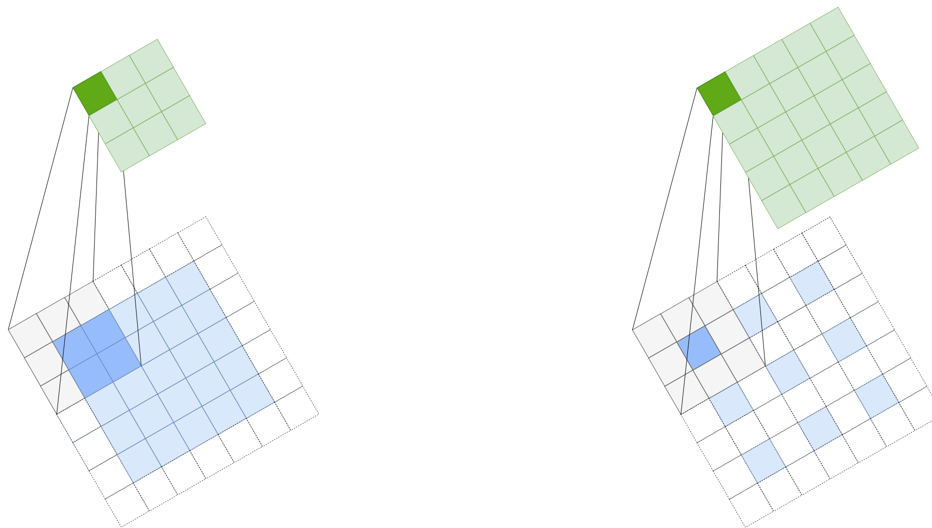


Figure 2.11: The original U-Net architecture. [18].

The left side of the architecture shown in Figure 2.11 is referred to as the contracting path. Each step in the contraction is built up by two $3x3$ convolutions, each followed by a ReLU [18]. After that, the output is downsampled to half its size using a max-pooling layer with a stride of two. For each step down, the number of feature maps is doubled, which is controlled by the number of kernels in the preceding layer [5]. On the

right-hand side of the network, we find the expansive part, which upsamples the feature maps. The feature maps are upsampled using transposed convolution. The transposed convolution can be thought of as the convolution required to go the opposite way [14], as an example, we refer to Figure 2.12 where Figure 2.12a shows a convolution of a $5x5$ input with a kernel of size $3x3$ and a layer of zero-padding around the input. The input is convoluted using a stride of two resulting in an output of $3x3$. The transposed convolution will then become what is shown in Figure 2.12b, a $3x3$ convolution of an input of $3x3$, which is padded with zeros around each value. The effective stride used is equal to one, resulting in an output of $5x5$. This is simply an emulation of how the transposed convolution works; adding rows and columns of zeros is not computationally efficient [14]. More precisely, the transposed convolution works by changing the forward and backward passes of a convolution [14].



(a) Convolution of $5x5$ input using $3x3$ kernel padded with a stride of two.

(b) The transposed convolution of 2.12a.

Figure 2.12: $3x3$ convolution and its transposed. Modified from [14].

After the transposed convolution, the resulting feature maps are concatenated with the cropped feature maps from the contracting path followed by two $3x3$ convolutions and ReLUs in a similar fashion as in the contracting path. The output is then $1x1$ convoluted to produce the desired amount of classes for the given application, which is 2 in the case of Figure 2.11.

2.7 Cardiac ultrasound

Analyzing cardiac structures from 2D echocardiographic images is a prevalent clinical task when establishing a diagnosis and measuring the cardiac morphology [2]. Using cardiac ultrasound images, one can estimate various clinical indices, often extracted from segmented images. A typical example could be to estimate the ejection fraction of the left ventricle. This requires an accurate delineation or segmentation of the left ventricular endocardium in both end-diastole and end-systole [2], which refers to the state where the volume is largest and smallest respectively [22]. The ejection fraction is used clinically to assess systolic cardiac

capability [22]. The areas of the heart relevant to this report are labeled in Figure 2.13, the white arrows in the figure denote the direction of blood flow.

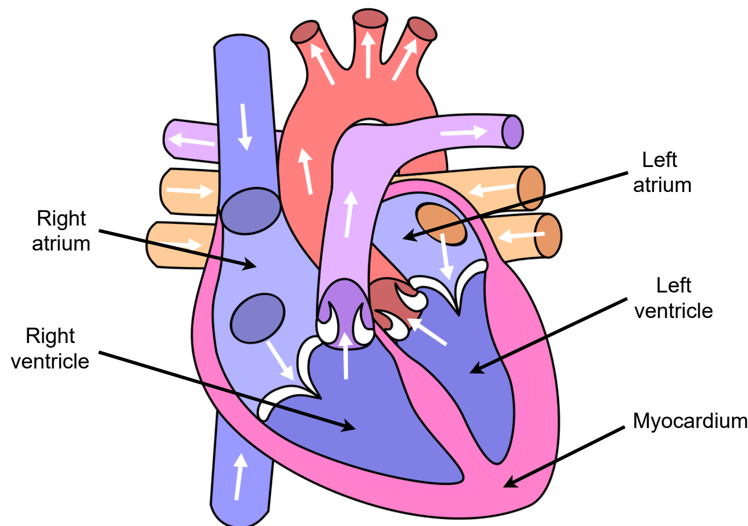


Figure 2.13: A diagram of the different regions of the heart. Modified from [23].

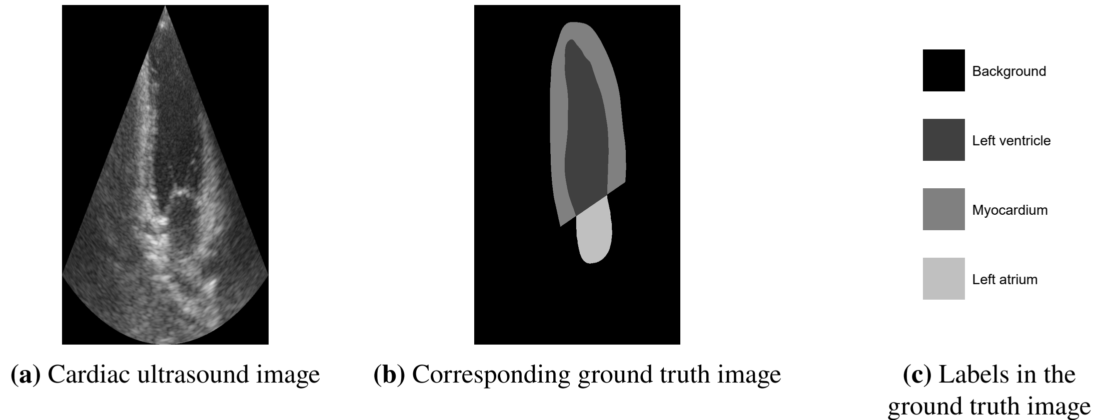
The process of segmenting cardiac ultrasound images can be challenging, especially obtaining high levels of accuracy. This is due to a variety of properties regarding both the anatomy of the heart and the ultrasound technique itself. We will not go into great detail but only mention some of the challenges. The images are prone to weak contrast between the myocardium and the blood pool, and there are several brightness inhomogeneities, variation in speckle pattern along the myocardium due to the orientation of the probe with respect to the tissue, presence of muscles with intensities similar to the myocardium, significant tissue echogenicity variability within the population. Finally, there is variation in shape, intensity, and motion of the heart structures across patients and pathologies [2].

Using DNNs instead of simpler ML based algorithms on ultrasound data is beneficial for many reasons. First and foremost, DNNs does not require feature engineering or prior knowledge to achieve satisfactory accuracy [1]. What this means in practice is that the simpler ML based algorithms need extraction and processing of features before the algorithm can process them, and in some cases, one also needs to know the prior distribution of the data [1].

The Camus (Cardiac Acquisitions for Multi-structure Ultrasound Segmentation) dataset is a publicly available dataset for cardiac ultrasound segmentation [2]. The dataset is part of a competition, where the idea behind the competition is to perform inference on the testing set and submit the results, and the organizers of the competition quantify the result to rate the participants. As a consequence of this, only the training set includes the ground truth targets for verification. It is not possible to qualitatively measure a model's performance without the ground truth in the testing dataset. An alternative is therefore to take out some of the samples in the training dataset and use these as the testing dataset.

The dataset contains a training set with 450 patients, where each patient sample includes two- and four-chamber views. Two- and four-chamber views refers to which areas are visible in the image. The

different areas are labeled in Figure 2.13. A two-chamber view contains the left ventricle and left atrium, and myocardium, while four-chamber views also contain the right ventricle and right atrium. We see an example of an ultrasound image of the two-chamber view in Figure 2.14a, while Figure 2.14b shows the corresponding ground truth segmentation. The ultrasound image is taken looking upwards, so the ultrasound image will therefore be depicted upside-down relative to Figure 2.13. The segmentation task intended using the Camus dataset is segmentation of four different areas in the cardiac ultrasound image. The four classes are background, left ventricle, myocardium, and left atrium. Figure 2.14c labels the different areas in Figure 2.14b. Three independent cardiologists have performed the segmentation in the ground truth images [2].



(a) Cardiac ultrasound image **(b)** Corresponding ground truth image **(c)** Labels in the ground truth image

Figure 2.14: Sample from testing dataset.

The dataset consists of clinical exams of 450 patients. The available samples are highly heterogeneous in terms of image quality and pathological cases [2]. The dataset contains images of varying quality, where about 35% are of good quality, 46% of medium quality, and 19% of poor quality [2].

2.8 Evaluation

There are several important benchmarks relevant to this report. We will focus on evaluating accuracy and performance in terms of speed. We are mainly concerned with the inference on FPGA, and will therefore only focus on the inference and not on pre- and post-processing.

During training we can for example use cross-entropy loss to optimize the model, as this is common practice [6]. To quantify the obtained results, we could use the Sørensen-Dice Coefficient to calculate the accuracy of the network on the testing dataset using another metric than the one used to train the network. The Sørensen-Dice coefficient is defined in Equation 2.11. The coefficient gives an indication of the degree of shared values in the output and the ground truth image [24] and is therefore well suited to evaluate a segmentation application qualitatively.

$$d = \frac{2|\boldsymbol{X} \cap \boldsymbol{Y}|}{|\boldsymbol{X}|+|\boldsymbol{Y}|} \tag{2.11}$$

In Equation 2.11 the d is the Sørensen-Dice coefficient, $\boldsymbol{X}$ is the estimated segmentation and $\boldsymbol{Y}$ is the ground truth segmentation. In contrast to an image classification application which outputs a target

vector, a network such as the mentioned U-net architecture outputs a segmentation mask. A consequence of downsampling the input image is that the corresponding output will be of a smaller size than the ground truth segmentation. In order to get a good estimate of the model's accuracy, we need to upsample the output segmentation mask. If one were to downsample the ground truth segmentation, one would lose much information, rendering the accuracy estimate less relevant as they would also be different for each input resolution, making it easier to obtain high accuracy for small image resolutions as they contain less information.

In order to compare the performance of the different computing platforms, we typically measure the latency and throughput of the system. These metrics can be obtained by using, for example, software counters in the code and thereafter be averaged over a number of inference runs. The latency estimate reports the time from an input is fed to the network before the results are produced, whereas the throughput tells us how many inputs are processed per second, typically in terms of frames per second (fps) in image applications.

3 Previous work and background

This section will present some of the previous work done on DNN inference on FPGA and other relevant subjects. The idea of using FPGA as an inference engine is not new, and there are options on how to do this using existing technologies and frameworks. This section will briefly discuss some of the ideas behind these and what important aspects to keep in mind during the design phase.

3.1 Accuracy

Leclerc et al. [2], which also provided the Camus dataset, has surveyed different DNN models on segmenting images from the dataset. The highest obtained accuracies in terms of the Sørensen-Dice coefficient is, in this case, $\sim 95\%$ and was done with a U-net model optimized for accuracy and is called U-net 2 [2]. The accuracy is given per class, whereas we use the average over all classes, so we therefore consider the average of the state of the art to estimate what accuracy is achievable. Though the U-net 2 achieved the highest accuracy Leclerc et al. [2] concluded, among other things, that the U-net architecture was the most effective considering the trade-off between the number of parameters and achieved performance. The U-net achieved a Sørensen-Dice score about $\sim 0.3\%$ lower than the U-net 2 model. The U-net 2 model contained in this case 18 million trainable parameters in contrast to U-net, which only contained 2 million [2], hence providing great relief in terms of memory footprint with a small reduction of accuracy.

3.2 Computing platforms

Inference of DNN is most commonly performed on either a CPU or a GPU if the amount of processing is very compute heavy. However neither the CPU or GPU is optimized for neural network inference. In other words, there is room for improvement by utilizing more specialized processing units. There are several options as to how this can be done. The most common examples include using an application-specific integrated circuit (ASIC), FPGA or a more general deep learning processing unit (DPU) [1].

CPU and GPU both have a higher theoretical peak performance than FPGA, however the hardware on the two computing platforms are not optimized to perform the arithmetic operations associated with DNN inference, in contrast to the reconfigurable FPGA where one can simply generate dedicated hardware [25]. FPGA is able to offer orders of magnitude higher performance, and with the ability to be reconfigured in contrast to ASIC [25]. ASIC is in general better than an FPGA when it comes to energy efficiency and performance, however the design is locked as the ASIC cannot be changed after production. The gap between the two is closing as FPGAs are becoming better [25].

GPUs achieve their performance because of their ability to process large image batches in parallel. However, in some applications such as video streams where the output latency should be minimized, the video must be processed frame by frame. In some cases, the power consumption can also limit the deployment of a GPU, such as in an embedded system [8].

Diminishing returns from technology scaling has resulted in the research community focusing on specialized accelerators. Utilizing ASICs yields the best results, but they are however not able to cope with the changing DNN architectures [7]. The development cycles and costs of ASIC implementation is also

significantly higher [7]. The design cycles of FPGA implementations are also longer than CPU, and GPU implementations as these have a wide variety of well-established frameworks to use. However, there are emerging new frameworks for FPGA inference which is promising.

Tu et al. [26] investigated the use of heterogeneous computing platforms for DNN inference. More precisely, the use of a combination of GPU and FPGA. In their initial benchmarks they used floating-point precision for the CPU and GPU, and 16 bit fixed-point precision for the FPGA. They compared a Xilinx Artix-7 FPGA with an Nvidia GTX1080 GPU and an Intel E5-2609 CPU on a VGG16 implementation, which contains both convolutional and fully connected layers. They found that the GPU offered a speedup of 31.2x over the CPU, while the FPGA achieved a speedup of 9.7x. However, considering the energy efficiency, they found the GPU to be 26x more energy efficient than the CPU, whereas the FPGA yielded a 65x improvement in energy efficiency. They suggested an architecture where parts of the inference are performed on a GPU and FPGA. The authors concluded that one could perform convolutional operations on the GPU and fully connected layers on a FPGA, as the GPU offered higher degrees of parallelism and FPGA can be better optimized for sequential processing.

3.3 Inference acceleration

One of the main challenges when using an FPGA as an accelerator is the limited preset on-chip memory and limited off-chip bandwidth [7][27][28][29]. DNNs tend to have a substantial memory footprint. One widespread approach to lighten the memory footprint is utilizing weights and activations of lower precision. This could, for example, be done by using 8-bit integers instead of 32-bit floating-point values [28]. In some cases, the weights can be quantized down to as much as 1-bit weights. However, this comes at the cost of accuracy [27][28]. The reduction of accuracy is comparably small, so the level of quantization works as a trade-off between performance in terms of speed and accuracy [28]. Some real-time requirements might not be possible to fulfill with floating-point precision [25]. In addition to this, the choice of architecture should also reflect the fact that the inference will be performed on an FPGA. This entails choosing an architecture with the possibility of a high level of parallelism and relatively simple operations which can be mapped onto FPGA fabric.

Miyama [30] implemented a U-net architecture on FPGA, using a quantization down to three bits. The inference engine was implemented as a dedicated hardware architecture for a slightly lightened version of the U-net network. The target application was segmenting the Cambridge-driving Labeled Video Database (CamVid), an on-vehicle image dataset [30]. The network was made lighter by reducing the number of contraction- and expansion steps in the U-net architecture from three to two [30]. This resulted in a framerate of 123 fps and was compared with a Vitis AI implementation of a similar network which achieved a resulting throughput of 32 fps. This was done using an image resolution of $256x256$ on a Xilinx Alveo U200 FPGA. The Vitis AI implementation was run using 8-bit weights [30].

Borkovkina et al. [31] implemented a U-net architecture for GPU, which utilized a less complex model of the U-net architecture as well as specialized GPU hardware and Nvidia's TensorRT. TensorRT optimizes networks for inference by for among other things quantizing the weights. The model was trained on an application segmenting retinal layers in optical coherence tomography. Using the techniques mentioned

above and quantizing the weights to 8-bit, they achieved a speed up during inference of 21x compared to a similar network with 32-bit floating-point weights on GPU.

There are several options when designing inference engines for DNNs on FPGA. One could, for example, utilize end-to-end compilers such as, for example, the Finn framework made by Xilinx. This framework converts high level specifications of DNNs into an FPGA bitstream. The models are typically quantized down to 1 to 2 bits, but it supports an arbitrary number of bits [27][28]. Using such high levels of quantization allows even simpler arithmetic operations, resulting in increased throughput. Similarly it is possible to implement a DNN using high-level synthesis (HLS) to generate hardware description language (HDL) code [1]. These approaches tend to increase the design cycles. All the mentioned alternatives require hardware synthesis, which is a very time-consuming procedure. In the best case, this is only performed once. However, if the network topology is slightly altered, this would, in most cases, require the hardware to be synthesized over new. Considering the end-to-end compilers, these are in most cases very academic and under development and are therefore more specialized, less stable, and are much harder to use.

3.4 PyTorch

Implementing DNNs can be easily done using for example Tensorflow, Caffe, Keras, or PyTorch. The frameworks differ slightly, but many of the concepts are similar. PyTorch is a framework focused on balancing both high usability and speed [32]. The framework enables simple implementation of complex models using its simple API to implement distinct layers and modules. PyTorch runs on an efficient C++ code, resulting in high performance and bypasses Pythons global interpreter lock, which prevents multiple threads running at once [32]. PyTorch also has support for GPU acceleration and needs minor alterations in the code in order to work with Nvidia's CUDA framework to run asynchronously on a GPU [32].

The PyTorch framework is typically used to implement DNN for CPU and GPU. The framework provides an API to construct the different layers, their operations, and connections [32]. The model can either be implemented by hand or imported as a pre-trained model. An example of how a fully connected network can be implemented is shown in Listing 1. The code describes a simple, fully connected neural network similar to the one in Figure 2.2, with 6 input nodes, 12 nodes in the second layer, and 10 output nodes. The `forward` function defines how the data is passed through the network.

```
1  import torch.nn as nn
2  import torch.nn.functional as F
3
4  class Example_Net(nn.Module):
5      def __init__(self):
6          super(Example_Net, self).__init__()
7          self.fc1 = nn.Linear(6, 12)
8          self.fc2 = nn.Linear(12, 10)
9
10     def forward(self, x):
11         x = self.fc1(x)
12         x = F.relu(x)
13         x = self.fc2(x)
```

```
14        output = F.log_softmax(x, dim=1)
15        return output
```

Listing 1: Example implementation of small fully connected neural network

The PyTorch framework also provides many functions to ease the implementation of high-level constructs, such as data loaders, more complex layers, training algorithms, and more [32].

3.5 Vitis AI

Vitis AI is a framework made by Xilinx for converting high-level descriptions of DNNs to a model compatible with the corresponding Vitis AI DPUs to speed up inference. A DPU is a pre-synthesised compute platform for FPGA. The framework converts a high level descriptions of a DNN to the instruction set architecture (ISA) for a target DPU. Vitis AI supports TensorFlow 1.x and 2.x, Pytorch, and Caffe [4]. The model is implemented and trained on either a CPU or GPU, and there is therefore no need to implement complex training algorithms on the FPGA.

In order to meet the requirements of high throughput and low latency, the application requires high memory bandwidth. An elegant way to optimize for this is to use quantization of weights and activations. The Vitis AI framework quantizes the model's weights and activations using its built-in quantizer, which converts the models weights from 32-bit floating-point to fixed-point precision, for example, 8-bit integers. This is explained in further detail throughout Section 3.5.1. This increases the performance and reduces the power consumption of inference [4]. The training is done using the 32-bit floating-point implementation to obtain high levels of accuracy and is thereafter quantized to the target bit width, which reduces the complexity with little loss of accuracy.

The Vitis AI quantizer takes in a floating-point model and performs a set of preprocessing algorithms to optimize the model before quantization. After the weights and activations are quantized, the quantizer runs several iterations of inference to calibrate the activations using a calibration image dataset. The calibration does not perform backpropagation, meaning it does not evaluate the outputs, so the calibration dataset does not need to be labeled [4]. The quantizer returns a DPU deployable model, which the compiler can compile to run on a target FPGA.

The compiler translates the model into an intermediate representation in the form of a control- and dataflow graph optimized by fusion of computation nodes and efficient instruction scheduling by exploiting parallelism and data reusage to maximize on-chip memory usage.

The Vitis AI framework is still under development and therefore still lacks the support of some arithmetic operations. More precisely, the DPU does not support arbitrary zero padding, meaning that zero-padding as part of operations such as convolution and pooling is supported, whereas zero-padding, in order to meet a target resolution, is not. Consequently, the image sizes have to be a multiple of 16 to be able to concatenate correctly between the contracting and expanding layers in a U-net architecture.

3.5.1 Quantization

The Vitis AI framework uses the quantization method described in [33], we will not go into great detail on how the quantization is deduced mathematically, refer to [33] a more detailed description. Quantizing weights, biases, and activations introduces quantization noise, which can lead to reduced model performance in terms of accuracy [33]. The model is quantized per channel as the range of weights can be of varying size, consider for example two channels where one channel has weights in the range of $[-128, 128]$ while the other channel has $\langle -0.5, 0.5 \rangle$, the latter will be quantized to zero. The quantization is done in four steps; cross-layer equalization, bias absorption, quantization, and bias correction.

Cross-layer equalization can simply be explained as factoring the weights and adjusting the bias so that the weight ranges become more equal between layers in order to prevent small weights from being quantized to zero similarly as the case explained in the paragraph above. When using for example a ReLU, which is described in Equation 2.4, it is simple to see that the scaling equivariance $h(sx) = sh(x)$ holds. It can also be shown that this holds for any piece-wise linear activation [33]. Using a fully connected neural network with two layers as an example, the outputs from the first layer can be expressed as: $\boldsymbol{z} = h(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{W}_0^{(1)})$, and $\boldsymbol{y} = h(\boldsymbol{W}^{(2)}\boldsymbol{z} + \boldsymbol{W}_0^{(2)})$ for the output layer. Using the scaling equivariance we can write this as:

$$\boldsymbol{y} = h(\boldsymbol{W}^{(2)} h(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{W}_0^{(1)}) + \boldsymbol{W}_0^{(2)}) \tag{3.1}$$

$$= h(\boldsymbol{W}^{(2)}\boldsymbol{S} h(\boldsymbol{S}^{-1}\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{S}^{-1}\boldsymbol{W}_0^{(1)}) + \boldsymbol{W}_0^{(2)}) \tag{3.2}$$

$$= h(\widehat{\boldsymbol{W}}^{(2)} h(\widehat{\boldsymbol{W}}^{(1)}\boldsymbol{x} + \widehat{\boldsymbol{W}}_0^{(1)}) + \boldsymbol{W}_0^{(2)}) \tag{3.3}$$

where $\boldsymbol{W}$ is the weights from the layer denoted with the superscript, $\boldsymbol{W}_0$ is the bias, h is the activation function, $\boldsymbol{x}$ is the input and $\boldsymbol{y}$ is the output. The factor $\boldsymbol{S} = diag(s)$ is a diagonal matrix where the value $\boldsymbol{S}_{ii}$ is the scaling factor s_i for the ith neuron. If for example the neurons in the first layer contains much larger values than in the second layer, this would then allow us to scale the weights and biases by using $\widehat{\boldsymbol{W}}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{S}$, $\widehat{\boldsymbol{W}}^{(1)} = \boldsymbol{S}^{-1}\boldsymbol{W}^{(1)}$ and $\widehat{\boldsymbol{W}}_0^{(1)} = \boldsymbol{S}^{-1}\boldsymbol{W}_0^{(1)}$, thus making the weights more similar so that the obtained weight induces less quantization noise.

Bias absorption entails moving large biases to subsequent layers. If a layer has a great bias the range of the activation quantization will increase, resulting in a lower accuracy [33]. Using the same example as above, we can write:

$$\boldsymbol{y} = h(\boldsymbol{W}^{(2)}\boldsymbol{z} + \boldsymbol{W}_0^{(2)}) \tag{3.4}$$

$$= h(\boldsymbol{W}^{(2)} h(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{W}_0^{(1)} + \boldsymbol{c} - \boldsymbol{c}) + \boldsymbol{W}_0^{(2)}) \tag{3.5}$$

$$= h(\boldsymbol{W}^{(2)} h(\boldsymbol{W}^{(1)}\boldsymbol{x} + \widehat{\boldsymbol{W}}_0^{(1)} + \boldsymbol{c}) + \boldsymbol{W}_0^{(2)}) \tag{3.6}$$

$$= h(\boldsymbol{W}^{(2)}\hat{\boldsymbol{z}} + \widehat{\boldsymbol{W}}_0^{(2)}) \tag{3.7}$$

where $\widehat{\boldsymbol{W}}_0^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{c} + \boldsymbol{W}_0^{(2)}$, $\hat{\boldsymbol{z}} = \boldsymbol{z} - \boldsymbol{c}$ and $\widehat{\boldsymbol{W}}_0^{(2)} = \boldsymbol{W}^{(2)} - \boldsymbol{c}$, and $\boldsymbol{c}$ is found using the distribution

of the pre-bias activation. This relation can be used if for example the bias terms in the first layer are much greater than in the second layer. One can equalize this by moving $\boldsymbol{c}$ to the bias in the second layer, thus reducing the bias and making the activation ranges in the two layers more equal.

After quantization the bias is corrected, denoting the floating point precision weights and biases as $\boldsymbol{W}$ and the corresponding quantized weights and biases as $\widetilde{\boldsymbol{W}}$, we can write the outputs as $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x}$ and $\widetilde{\boldsymbol{y}} = \widetilde{\boldsymbol{W}}\boldsymbol{x}$ respectively. From this we can write:

$$\widetilde{\boldsymbol{y}} = \boldsymbol{y} + \boldsymbol{\epsilon}\boldsymbol{x}, \tag{3.8}$$

where ϵ is the quantization induced error [33]. The quantizer adjusts for this error by using the relation:

$$\begin{aligned} \mathbb{E}[\boldsymbol{y}] &= \mathbb{E}[\boldsymbol{y}] + \mathbb{E}[\boldsymbol{\epsilon}\boldsymbol{x}] - \mathbb{E}[\boldsymbol{\epsilon}\boldsymbol{x}] && (3.9) \\ &= \mathbb{E}[\widetilde{\boldsymbol{y}}] - \mathbb{E}[\boldsymbol{\epsilon}\boldsymbol{x}] && (3.10) \end{aligned}$$

Thus, by subtracting the expected error, the mean is preserved. The expected error is estimated by comparing the expected activation before and after quantization [33].

3.5.2 DPU

The DPU is a programmable computation engine optimized for DNN inference, which runs the instructions generated by the Vitis AI compiler. It consists of a group of parameterizable cores pre-implemented to a given FPGA architecture, meaning it does not require the time-consuming place-and-route algorithm. It is designed to accelerate typical workloads in DNN inference such as image and video classification, semantic segmentation and object detection, and object tracking [4]. This is done through parallelization of the computation and custom processing engines. Additionally, the DPU supports multi-threading for parallelization of inference of multiple inputs [4]. The multi-threading is easily deployed using standard multi-threading libraries in Python. A visualization of one of the architecture is shown in Figure 3.1.

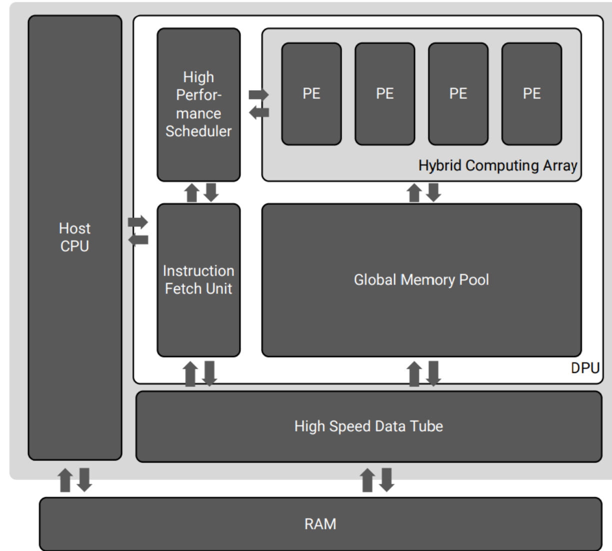


Figure 3.1: DPU hardware architecture [4].

The architecture shown in 3.1 is one of many different available pre-made architectures in the Vitis AI framework, this particular DPU architecture is optimized for the Xilinx XCZU7EV FPGA. The XCZU7EV is an edge FPGAs and is therefore relatively small, larger FPGAs has larger area of FPGA fabric, effectively meaning that they have room for larger and more complicated DPUs [4].

A consequence of the DPUs being pre-synthized is that not all arithmetic operations are supported by default as these might not be implemented. The DPUs supports the most common operations, but this might not be sufficient in all cases. If one were to run inference of a network containing one or more unsupported arithmetic operations, there are some alternatives to bypass this. By default, the arithmetic operations that are not supported by the DPU are reassigned to the onboard CPU. Alternatively, the DPU architectures can be adjusted to support operations missing by implementing these and adding them to the ISA [4].

4 Implementation

This chapter will describe the work done as part of the research in this report. The whole codebase can be found in [34]. The goal of the implementation is to investigate whether FPGA is a viable option as compute platform in DNN inference of cardiac ultrasound images. The models resulting accuracy is of course important, though we recognize that achieving the state of the art accuracy is beyond the scope of this project. The main goal regarding the models accuracy is to implement a DNN on an FPGA with as small loss of accuracy compared to CPU and GPU as possible. The loss of accuracy is mainly induced by the fact that FPGA inference is optimized using quantization. The weights are quantized from 32-bit floating-point to 8-bit integers in order to minimize the memory footprint. Further, we evaluate inference with different image resolutions to investigate the trade-off between accuracy and throughput.

4.1 A novel design approach

This design approach was developed as part of the implementation done during this report. It is a suggestion towards choosing a DNN suitable for FPGA inference. We will consider a real-time, embedded system used for segmentation of cardiac ultrasound images. First and foremost, the model itself should be able to perform adequately with regards to the application of segmenting the ultrasound images. Furthermore, the system should fulfill real-time requirements. The system should be able to support a typical framerate of a standard video stream of approximately 50 fps with a latency lower than 100 ms in order to be responsive.

Using the approach described in this report, we need to take several things into account when choosing the DNN architecture. Basing the design choices on using the Vitis AI library, there are some important aspects to consider when choosing the network model for inference. These decisions are summarized in Figure 4.1.

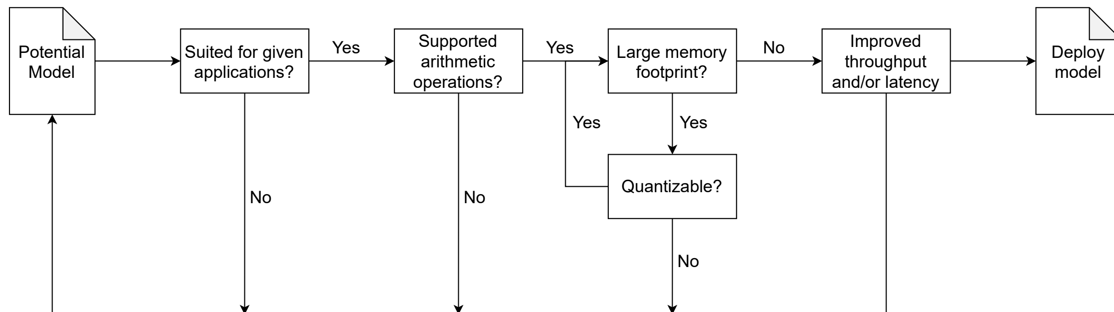


Figure 4.1: Decision tree for chosing an architecture for FPGA inference.

Starting on the left side of Figure 4.1 with a potential model. The most important choice entails choosing a model that is well suited for the given application in order to obtain satisfactory accuracy. Quantizing the model will not lead to higher accuracy, so the initial accuracy must be acceptable. The remaining conditions are more specific to FPGA implementation. The first involves verifying that the selected network model contains supported arithmetic operations. Vitis AI and similar frameworks do support most common arithmetic operations used in DNNs inference. However, some network models could contain very specialized

instructions which might not be natively supported or implemented in the DPU overlays. Further, using Vitis AI, some built-in functions in the supported frameworks, i.e. PyTorch, TensorFlow, and Caffe, are not supported and should be avoided though the underlaying arithmetic operations are supported. Vitis AI bypasses the unsupported arithmetic operations by moving these from the accelerator back to the onboard CPU. This is of course in some cases a viable option, but it will act as a bottleneck as the CPU will not be able to achieve the same levels of parallelism as the FPGA, and the transfer of data could add some overhead.

The main bottleneck for FPGA inference is the memory bandwidth. Ideally, all the weights and biases should fit in on-chip memory. This might be problematic using DNN models as these tend to contain a large number of weights and biases. It is possible to cope with this using quantization. However, in some cases with very large networks, it could require very high levels of quantization, in some cases to as much as one bit [28]. This might not always be feasible since most applications aim to maximize accuracy. Off-chip memory access should be minimized in order to relieve the memory bandwidth. Networks with large memory footprint should therefore be avoided. Quantizing the weights for FPGA inference could, in most cases, be performed regardless of the memory footprint in order to speed up inference.

Intuitively, there is no reason to deploy the model on an FPGA if there is no improvement in either throughput or latency, making the FPGA inference redundant. In some cases, depending on the initial compute platform, the FPGA could be more energy efficient in its inference of DNNs, this is especially the case if compared with a GPU, which was discussed in Section 3.2. Considering an embedded real-time system employing a GPU might not feasible because of high power consumption.

4.2 Network implementation

As discussed in Section 4.1, the first step towards finding a well-suited network for FPGA inference is to find a model with high initial levels of accuracy. As described in Section 2.6, the U-net network is well suited as it achieves high accuracy on ultrasound and medical imaging applications. Furthermore, the network consists mainly of very standard arithmetic operations well suited for FPGA inference. The most notable abnormality being the concatenation of feature maps, which in turn is a straightforward arithmetic operation that can easily be done in hardware and is supported by the Vitis AI DPU. Though the network consists of 23 layers containing trainable parameters, they are all convolutional layers, meaning they have sparse weights as discussed in Section 2.4, thus making a smaller memory footprint compared to a fully connected network of similar size and depth. The U-net model therefore fulfills the suggestions from Section 4.1, which likely makes this a good choice for FPGA inference.

This section will describe the implementation of the U-net architecture as well as the pre- and post-processing stages. The model used is based on [18] with some alterations and is described in further detail in Section 4.2.2. The model will be evaluated in terms of throughput, latency, and accuracy. The network will mainly be measured during inference, using samples from the Camus dataset as input data. As there is no ground truth in the original testing dataset, we create our own testing dataset by replacing the samples in the testing dataset with samples from the training dataset. We will go into more detail on this in Section 4.2.3.

4.2.1 Preprocessing

As mentioned in Section 3.5, the Vitis AI lacks support for zero-padding. As a result, the images used during training and inference should be of a resolution which is a multiple of 16, in order to support the concatenation operations in the U-net model.

The samples in the Camus dataset are of varying sizes; we therefore choose to implement a pipeline where the images are preprocessed before each run so that the system itself supports different input sizes and codecs without extensively altering the codebase. Alternatively, we could have made a preprocessed training dataset, making the need for preprocessing redundant. Keeping the data in its original form and preprocessing the data for each run, we create a more scalable solution that can easily be altered to fit a wider selection of applications and easily benchmark different resolutions by parsing the dimensions as arguments to the model. As the primary goal of this report is to speed up the inference, the preprocessing will not be part of the benchmarks, assuming that in an application-specific implementation, we would infer on the given input or alternatively implement the preprocessing in another intellectual property (IP) such as a video block on the FPGA.

The different stages of the preprocessing are visualized in Figure 4.2. The samples in the training set are of varying size and dimensions, so we convert them to a quadratic form before feeding them to the network. The first stage entails rescaling the image towards the target dimension TxT. In the rescaling block, we scale the sample so that the largest dimension becomes the target size T while preserving the aspect ratio as shown in Equation 4.1. The $size$ in the equation refers to a tuple $[W_{new}, H_{new}]$, namely the new width and height.

$$size = \begin{cases} [\frac{T \cdot W}{H}, T], & W < H \\ [T, \frac{T \cdot H}{W}], & W \geq H \end{cases} \tag{4.1}$$

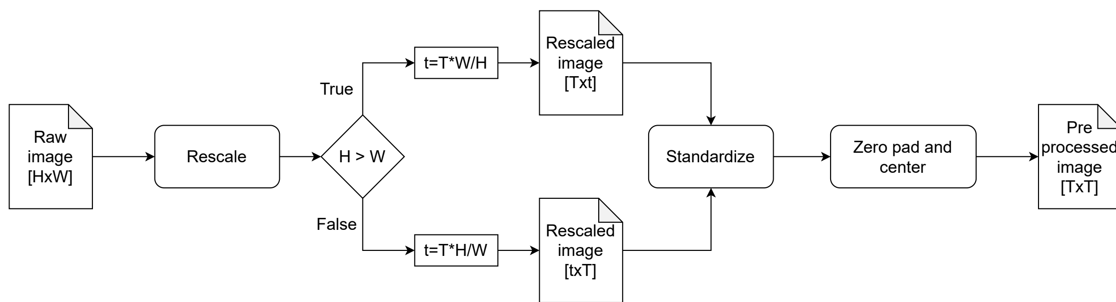


Figure 4.2: Visualization of the preprocessing stages

Thereafter the image is intensity standardized to reduce the impact of variations from the data acquisition using Equation 2.10 as discussed in Section 2.5.1. Finally, to obtain a square image, the image is padded with zeros in the shortest dimension so that the image ends up with a resolution of TxT. We also trained the network using normalization as defined in Equation 2.9. However, this yielded lower accuracy than standardization.

As an example of the preprocessing pipeline, we consider an image of the resolution $[W, H] = [200, 400]$ and a target dimension $T = 200$. This is also shown in Figure 4.3. When rescaling, the height H will be rescaled to 200, and the width to $W = 200 \cdot 200/400 = 100$. Thereafter the image is standardized using Equation 2.10 and finally padded with 50 zeros on both sides in the horizontal direction to obtain an input size of $[200, 200]$.

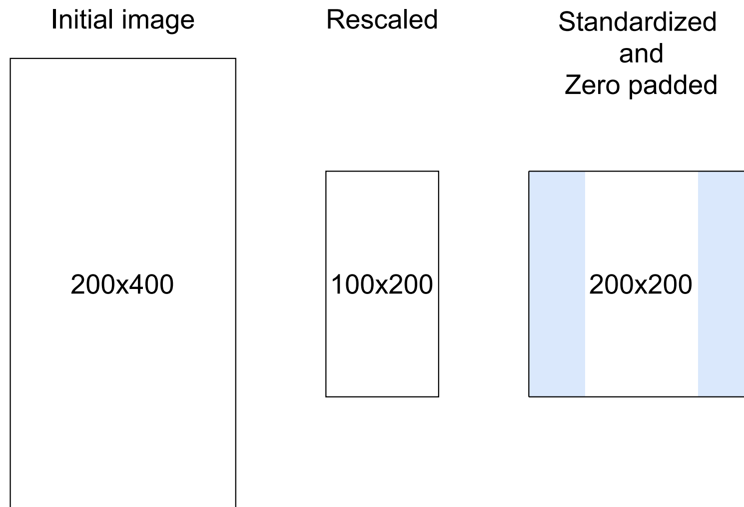


Figure 4.3: Visualization of preprocessing example

4.2.2 The network

The network was implemented using PyTorch 1.4.0, mainly because PyTorch is easy to use and supports all the layers in the U-net model and is compatible with Vitis AI. To begin with, the codebase was built up using TensorFlow, but this was quickly changed to PyTorch. The two frameworks are pretty similar; however, PyTorch is much simpler to use.

The implemented network is a modified version of the U-net network described in Section 2.6 and [18], and is shown in Figure 4.4. The main difference is that the original U-net uses cropping before concatenation and does not employ zero-padding in the convolutional layers, whereas we do, thus making an architecture producing the output of equal size as the input image.

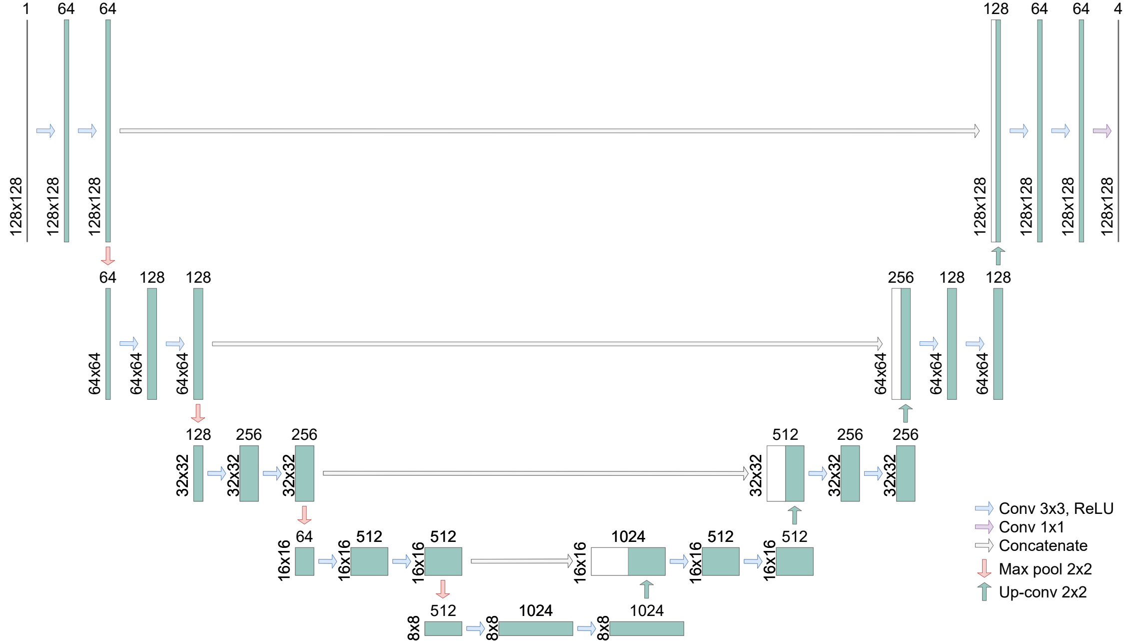


Figure 4.4: The modified version of the U-Net architecture, here shown with an input size of 128x128. Modified from [18].

In order to emulate an application processing a video stream, we preprocess all the inputs before inference. Thereafter, the network is fed one image at a time, in contrast to inferring a batch of images. The reason for this is that in an application processing a video stream, one will in most cases need to process frame by frame in order to keep the latency to an acceptable level. If one for example were to process the frames in batches of five, the system would have to wait for all frames to be ready before starting the inference, therefore inducing latency. Batching is commonly used to for example speed up GPU inference because it allows even higher utilization.

4.2.3 Training

In order to train the implemented network we use the Camus dataset described in Section 2.7 and [2]. As mentioned in Section 2.7 only the training dataset includes ground truth segmentations. We therefore construct our testing dataset using samples from the training dataset. Since the training dataset only consists of 450 samples, we want to separate only a small subset of these to the testing set so that there is as little impact on accuracy as possible while still having enough testing data to obtain a good estimate of the model's accuracy as discussed in Section 2.5. We chose to extract 50 samples from the training dataset to the testing set as a good compromise.

We only train on two-chamber views since the primary objective regards improving the speed and obtaining the resulting accuracy reduction, in contrast to obtaining the highest possible accuracy. Training on only two-chamber views makes it far simpler to obtain a good initial accuracy as the network needs to learn fewer variations. We train the application to segment out four different classes in the image; background, left ventricle, myocardium, and left atrium, as discussed in Section 2.7. We train the network using 50 epochs

and a batch size of five for all the image resolutions except $512x512$. In the case of $512x512$, the system used for training did not have sufficient RAM to train with a batch size larger than one.

The implementation described in this report is not intended to be part of the competition. This is because the resulting network will not be able to compete with the state of the art networks due to the quantization, and the main goal is not to obtain the state of the art accuracy but to speed up the inference time. However, the Camus dataset is readily available and relatively large and is therefore an excellent place to start.

4.2.3.1 Data augmentation

The number of samples in the Camus dataset is limited with a total of 450 samples of varying quality. In order to make the model more robust and invariant to slight differences of input, we use data augmentation to synthetically make the training set larger as described in Section 2.5.1. This means taking samples from the training dataset and perform transformations on the data to create similar but different data. As discussed in Section 2.5.1 data augmentation has proven to increase the accuracy on small datasets.

The augmentation algorithms are triggered randomly during training with a probability of 33% for a given algorithm. The dataset is reloaded to the model in each epoch, and the images are preprocessed over new. For each preprocessed sample, there is a 33% chance that a given augmentation algorithm will be performed, considering for example the first sample in the training dataset, the first epoch one might see a version free of augmentation, whereas in the next epoch it might be slightly rotated and infused with noise. The augmentations are deployed independently for each sample and are performed directly on a sample within an epoch, which means that if a sample is augmented, the network is only presented with the augmented sample in that given epoch. The network will therefore see both augmented and original samples as these will vary throughout the epochs, but never an augmented and its original within one epoch. The augmentation algorithms used in our implementation are summarized in Table 4.1.

Table 4.1: Augmentation algorithms used and their parameters

Augmentation	Arguments
Random cropping	Crop ratio = 0.1
Random rotation	θ_{max} = 15°
Random blackout	Max blackout = 100
Noise infusion	σ^2 = 0.05
Gamma augmentation	c = 1, γ = 0.35

The arguments used during the augmentation are found empirically, and the obtained accuracy will be different between runs of the training algorithm, which is not only because of the statistical method of training but also that these algorithms are deployed randomly. The idea behind each of these algorithms is explained in Section 2.5.1. We will only explain the parameters used in this section.

The cropping ratio limits the random cropping. The ratio dictates the maximum portion of the image frame that can be cropped out. The actual cropped area is a random value between 0 and the cropping ratio. In the random rotation augmentation, the θ_{max} refers to the maximal rotation angle that can be used. The max blackout argument refers to the largest area that can be blacked out. Since these augmentation

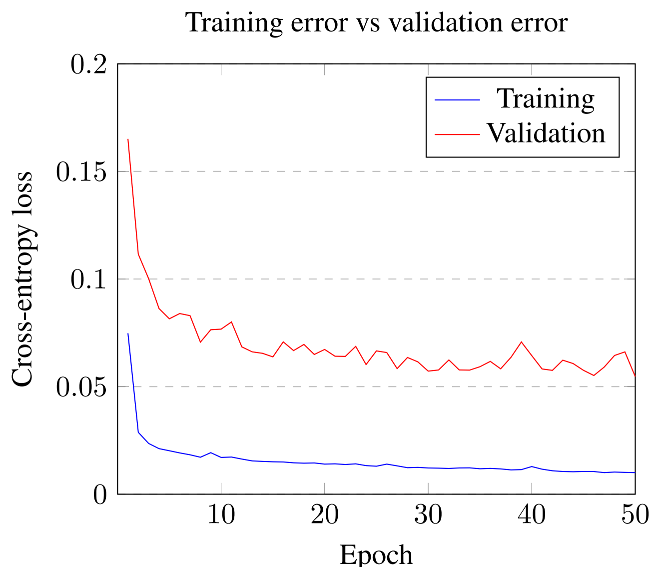


Figure 4.5: Error estimates during training for 128x128

algorithms are performed before preprocessing, this will not be affected by the resulting image size, whereas if they were to be applied afterward, one should employ this as a ratio where the area is dependent on the image size. The arguments in the two last augmentation algorithms are explained in Section 2.5.1.

4.2.3.2 Loss functions

In order to qualitatively measure the accuracy of a set of weights during training, we use a loss function as described in Section 2.5. There are several options as to which loss functions can be used. In this implementation, we use cross-entropy loss during training to optimize the model. Figure 4.5 shows the error as a function of time in terms of epochs of training. The graph shows the error from the training of the $128x128$ model. The training error refers to the error compared to the ground truth for the training dataset, whereas the validation error is the error on the validation set. Both the training and validation errors are declining as the number of epochs increases, indicating that the network is learning and generalizing on the features in the dataset. If, for example, the training error were to decrease while the validation error was increasing, it would be a sign of overfitting. The training algorithm will evaluate the obtained weights and save them if they have a lower validation error than the previous best set of weights.

When comparing the trained model during inference on CPU, GPU and FPGA we quantify accuracy reduction of quantization. This is done by calculating the Sørensen-Dice coefficient described in Section 2.8 on the obtained bitmask from the inference of the testing dataset.

4.2.4 Post processing

The network outputs four bitmasks of the same size as the input image, TxT, one for each of the four classes. Each bitmask encodes a probability for whether a pixel belongs to the given class or not. In order to measure the model's Sørensen-Dice coefficient, we merge the four bitmasks into one so that the resulting mask is of

the same format as in Figure 2.14b. Each pixel is labeled with a class $C = 0, 1, 2, 3$ based on the on which masks yield the highest probability, as shown in Figure 4.6. This results in one mask with equal size as the input image, TxT.

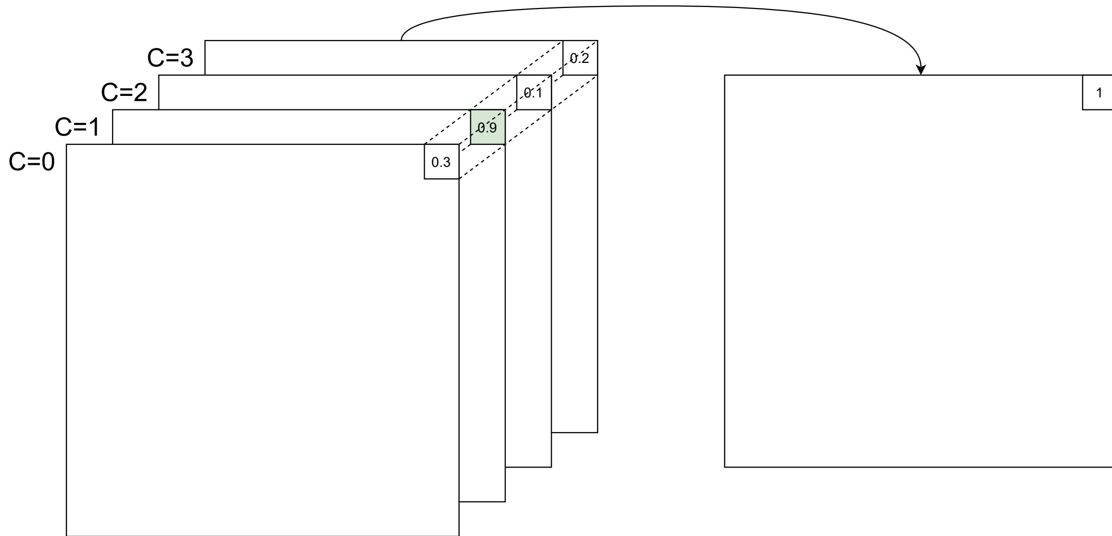


Figure 4.6: Merging segmentation masks using argmax

As discussed in Section 2.8 the segmentation mask will be of a different resolution than the ground truth segmentation and should therefore be upsampled. In order to quantify the outputted segmentation mask, it needs to be converted back to the initial resolution. In practice, this means reversing the preprocessing stages. The segmentation mask is outputted in a quadratic form due to the zero-padding in the preprocessing stage. Before upsampling the image, we remove zero padding. This procedure is similar to that of the zero-padding since we know the target dimensions from both the original input and the ground truth.

We used OpenCV's resize function [19] to upsample the image to the target resolution once the zero-padding is removed. As briefly discussed in Section 2.5.2 the OpenCV library contains different interpolation techniques. For upsampling, we use the nearest-neighbor interpolation. This was found using trial and error. The pixel value in the segmentation mask should be integers from 0 through 3, so using linear interpolation and thereafter quantizing the result back to integers would effectively be equal to the nearest neighbor interpolation.

4.3 Porting PyTorch network to FPGA

Converting a high-level Python implementation to a model compatible with FPGA inference requires multiple steps. The Vitis AI framework implements an OpenCL-like dataflow where the FPGA acts as an accelerator to a code running on a CPU, this is visualized in Figure 4.7. The data flow of the program consists of a preprocessing which is performed on a CPU, the preprocessed images are then inferred on an FPGA before the resulting output from the FPGA is post-processed on the CPU. This means that the porting mainly entails converting the inference network to the FPGA.

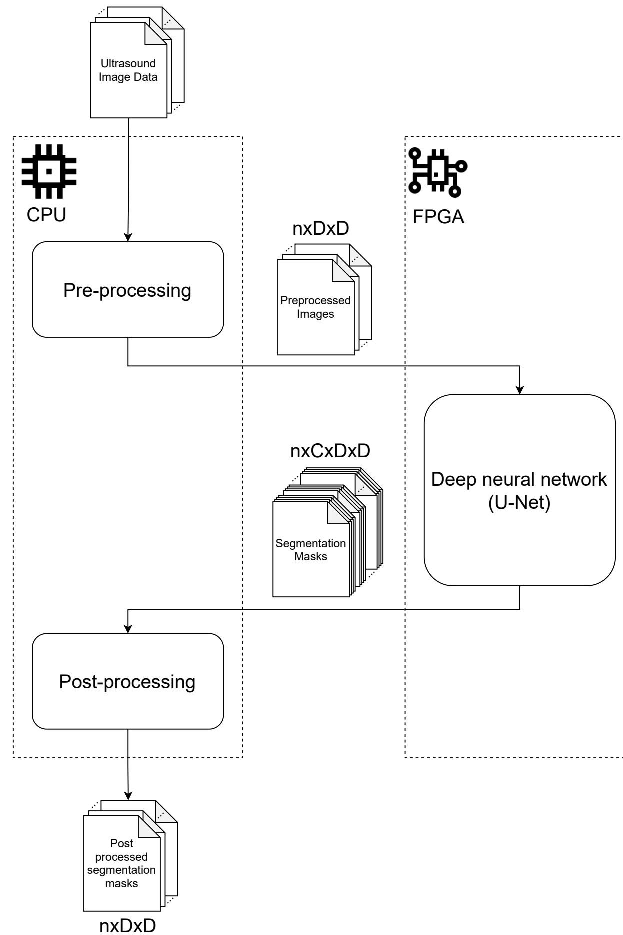


Figure 4.7: Visualization of dataflow

One major bottleneck in the inference of DNNs is the memory bandwidth. As an effort to relieve the memory footprint and thereby the memory bandwidth, the weights obtained during training are quantized from 32-bit floating-point precision down to 8-bit integer precision. This is described in Section 3.5.1 and [33]. The quantized model is saved into an intermediate representation containing information about both the weights and the model's structure. The final step of the porting process is to compile this intermediate representation to a target FPGA architecture. The compiled model can be imported into a Python script running on the FPGA, and inputs can be asynchronously be assigned to the accelerator, using Xilinx runtime API. The Vitis AI runtime API does not support high-level objects such as data loaders and tensors, so the preprocessing algorithms have to be implemented using arrays.

The FPGA runs a Linux distro provided by Xilinx called Petalinux, where all the dependencies are pre-installed. For an easier setup we chose to use Petalinux as provided by Xilinx, the distro runs on the FPGA's onboard CPU, but for simplicity we will hereafter refer to the FPGA-CPU as the FPGA. One should pay great attention to which libraries are used when implementing the model for the CPU and GPU, and the implementation for FPGA, as these should ideally be the same. To begin with, we used two different libraries for downsampling, namely, scikit on the CPU and GPU for training and OpenCV on the FPGA. This

alone resulted in much lower accuracy as the two behaved differently. The limiting factor for the deployment of libraries is the FPGA as it simply does not have as wide support in its operating system like Ubuntu or Windows. The CPU implementation should therefore cater to the FPGA implementation in terms of the libraries used in for example pre- and post-processing.

4.4 Optimizating DPU

Using the U-net implementation described in Section 4.2 we do not need to alter the DPU in order to meet the requirements in terms of arithmetic operations. There is a possibility to adjust the DPU architecture. However, we chose not to do this in this implementation because of the limited time available during the master thesis, as the implementation and synthesis of hardware are very time-consuming. It is possible to adjust the parameters in the DPU to for example optimize for parallelization of a given application. In addition to this, one could give more area to the arithmetic operations the model employs in order to utilize the FPGA even more.

5 Results

This section will present the obtained results in this report. We will start this section by looking at the accuracy before and after quantization for different image resolutions. We will thereafter look at the performance of the implemented model on CPU, GPU and FPGA for different image resolution. The hardware used in the following benchmarks is summarized in Table 5.1.

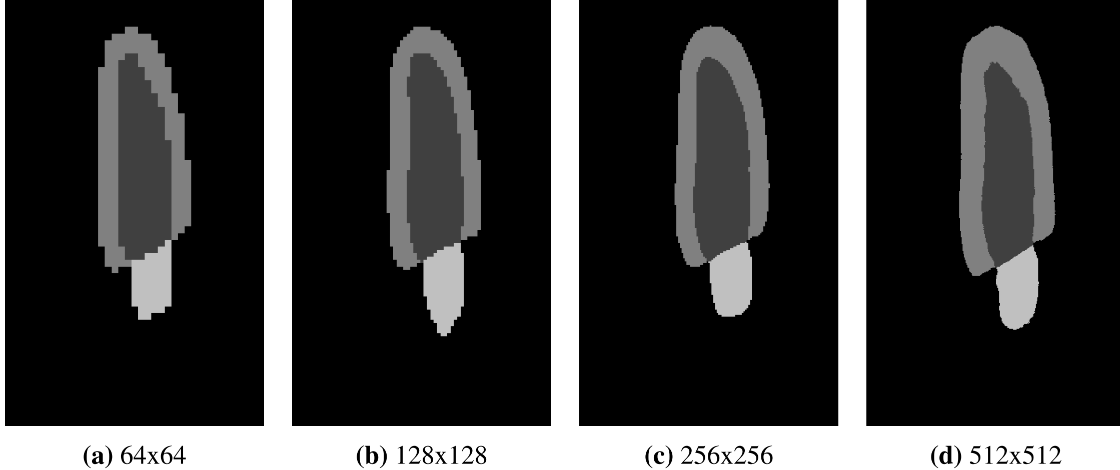
Table 5.1: The hardware used to obtain benchmarks.

Platform	Device
CPU	Intel Core i7-8650U
GPU	Nvidia GeForce GTX 1060
FPGA	Xilinx XCZU7EV

In order to evaluate the different models and compute platforms, we run inference on the constructed training set. The accuracy is estimated using the Sørensen-Dice coefficient, and the throughput and accuracy are computed using software timers. In order to get a good estimate, we run through the whole testing dataset twice so that the estimate is based on 100 runs. To measure the latency, we measure the time spent from an input is fed into the network until a result is ready, whereas the throughput is measured by timing the network from the dataset is ready to all the frames has been inferred, then dividing by the number of samples. The results include an evaluation of models using input sizes of $64x64, 128x128, 256x256$ and $512x512$, running on CPU, GPU and FPGA.

5.1 Accuracy

The obtained accuracies are shown in Table 5.2. We see that the obtained accuracy for the CPU and GPU is highest for the $128x128$ model, with the $256x256$ having a very similar level of accuracy. The $64x64$ and the $512x512$ obtain similar accuracies but lower than the two former. The accuracy for the FPGA are similar to that of CPU and GPU, with a small reduction in accuracy. There are several things to note from the accuracy of the implemented models. Since both the CPU and GPU implementations both run the same PyTorch model, their accuracy will be equal. The first and most important is that the accuracy reduction as a consequence of quantizing the weights is relatively small for all the models. The biggest difference in accuracy can be found in the model trained on images of resolution $512x512$, where the Sørensen-Dice coefficient is reduced by 1.12% when quantizing. Furthermore, observing the initial accuracies of the different models, we see that the model trained for images of resolution $64x64$ yields the lowest accuracy, but more noticeably, the $512x512$ yields the second-lowest accuracy. The fact that the model trained with the most information achieves the second-lowest accuracy can be due to several different effects. During training, the backpropagation searches for the minimum error, but in some cases, the algorithm might end up in a local minimum without being able to escape, therefore yielding a suboptimal result. Furthermore, the $512x512$ model was trained using a batch size equal to one, whereas the other models were trained with a batch size of five. This is due to the fact that the models were trained on a system with 10 GB of RAM, so in the case of $512x512$, the amount of memory was not sufficient for any batch size larger than one. This



(a) 64x64 **(b)** 128x128 **(c)** 256x256 **(d)** 512x512

Figure 5.1: Computed segmentation mask from the sample in Figure 2.14 inferred on the FPGA

might have an impact on accuracy as described in Section 2.5.

Table 5.2: Model accuracy

Resolution	CPU & GPU[%]	FPGA[%]	Δ
$64x64$	90.14	89.93	-0.21
$128x128$	92.61	92.61	0.00
$256x256$	92.33	92.18	-0.05
$512x512$	90.23	89.11	-1.12

It is also worth noting that there is very little difference in accuracy between an image resolution of $128x128$ and $256x256$, meaning that the obtained accuracy is comparable though the input data is reduced to 25% of its size. This in turn yields a pretty significant speedup as we will see in Section 5.2. Looking at the results for $64x64$ we see some loss in accuracy, but it should be noted that there is much information lost during the downsampling compared to the inherent resolution of the input data. However, we can view this as an interesting trade-off for applications where the throughput is more important than high accuracy.

In Figure 5.1 we see the impact of the different input resolutions. At $128x128$, one can see the impact of the low resolution in the form of a pixelated image, which in turn is even more apparent in the $64x64$ image. These masks are the estimated masks of the cardiac ultrasound image in Figure 2.14a and with the ground truth shown in Figure 2.14b, which is used to compute the Sørensen-Dice coefficient. The estimated masks in Figure 5.1 are obtained from inference on the FPGA.

5.2 Performance: throughput and latency

To benchmark the performance, we measure the throughput of the system in terms of fps as well as the system's latency. The results in terms of throughput and latency are summarized in Table 5.3 and 5.4 respectively. Due to limited amounts of RAM the benchmarks were not possible to perform using four threads when using an input resolution of $512x512$.

Table 5.3: Model throughput

Resolution	CPU [fps]	GPU [fps]	FPGA [fps]			
			1 thread	2 threads	3 threads	4 threads
$64x64$	9.02	76.25	123.78	186.76	185.32	201.61
$128x128$	2.95	56.78	42.74	77.31	66.39	83.77
$256x256$	0.78	27.54	11.16	21.57	17.54	23.34
$512x512$	0.18	9.08	2.80	5.47	4.35	-

From Table 5.3 we see that the obtained throughput is highest for the FPGA using a resolution of $64x64$ using four threads, the FPGA yield the highest framerates up to a resolution of $128x128$, thereafter the GPU yields the highest framerates. Considering the resulting framerates in Table 5.3 we see that the models generally perform better using four threads, whereas using three in all cases yielded worse results than using two. This could indicate the FPGA not being able to run more than two threads concurrently, resulting in a lowered framerate when sharing resources between three threads instead of four. Looking at the obtained framerates, this would make sense assuming there is some overhead. Comparing the resulting framerate of the FPGA with the CPU we can observe an increase for all configurations.

In Table 5.4 we see the latency of the different implementations. We see that the lowest latency is obtained using an FPGA with one thread and a resolution of $64x64$, thereafter the GPU yields the lowest latency for the remaining resolutions. Similar to the throughput, the latency is also lowest for the lower amount of threads. Comparing with the CPU we see that the FPGA has a lower latency in all cases, while in comparison with the GPU the latency is higher for all cases except small image sizes. This could be a consequence of the higher memory bandwidth in the GPU, in addition to more processing power. There is a remarkable increase in latency when going from two to three threads, which might be a consequence of the aforementioned degree of utilization, where the FPGA is likely limited to two concurrent threads, therefore the ltency is increased as a consequence of resource sharing.

Table 5.4: Model latency

Resolution	CPU [ms]	GPU [ms]	FPGA [ms]			
			1 thread	2 threads	3 threads	4 threads
$64x64$	115	13	8	9	14	18
$128x128$	320	16	23	24	37	45
$256x256$	1,103	34	88	89	141	166
$512x512$	4,107	105	349	354	555	-

In Table 5.5 we see a comparison of the obtained results. We are comparing the highest achieved results, meaning that for throughput, we base the speedup on the implementation using four threads, whereas the latency is compared using the implementation running one thread. Comparing the benchmarks with the CPU we see that the obtained speedup is larger than 22x and 30x with the heaviest workload when using an FPGA, and 50x when using an GPU. The latency on the FPGA is about 0.07x of the CPU latency. Comparing with the GPU yields less favorable results. For small image sizes, the FPGA obtains a higher throughput, with a 2.64x speed up compared to the GPU in the case of $64x64$, which is also the only implementation where the

FPGA yields a lower latency than the GPU. This is due to a combination of the mentioned gap in computing power as well as memory bandwidth compared to the GPU. In addition, the developed framework used in GPU inference is also a lot more mature than Vitis AI, which could also explain the gap in both throughput and latency.

Table 5.5: Performance comparison relative to the CPU implementation

Resolution	Throughput			Latency		
	CPU	GPU	FPGA	CPU	GPU	FPGA
$64x64$	1x	8.45x	22.35x	1x	0.11x	0.07x
$128x128$	1x	19.25x	28.40x	1x	0.05x	0.07x
$256x256$	1x	35.31x	29.92x	1x	0.03x	0.08x
$512x512$	1x	50.44x	30.39x	1x	0.03x	0.08x

Looking at the obtained performance metrics, we see that there is a possible trade-off between latency and throughput in the FPGA implementations for a given resolution. An increase in throughput through more threads results in an increase in latency and vice versa. Take for instance the implementation with an input resolution of $128x128$. We see that the lowest latency is obtained with one thread, which also yields the lowest throughput for the given image resolution. Increasing the number of threads to four yields an increase in both throughput and latency. This is most likely a consequence of the limited memory bandwidth, meaning that a system with high throughput will process larger amounts of data, requiring more off-chip memory access, thus resulting in higher latency.

6 Discussion

The FPGA obtained a level of throughput superior to that of a CPU and with comparable performance to a GPU without any notable deprecation of accuracy for either quantization or reduction in image resolution. In this chapter we discuss the findings in this report more generally, starting with the impact of quantization in terms of accuracy, throughput and latency. Finally discuss possible applications and the relevance of the findings.

6.1 Accuracy impact

Considering the obtained results presented in Chapter 5 we see that the quantization had little impact on the accuracy of the model. This is true for all the different resolutions evaluated in this report. However, the accuracy reduction between the quantized 8-bit integer implementation and the 32-bit floating-point implementation might be artificially small as a consequence of the suboptimal initial accuracy, meaning that if the 32-bit floating-point model would have had a higher initial accuracy, one could perhaps see a larger gap in accuracy as more information might be lost during quantization. Further experiments are needed to determine this however. Though the accuracy is lower than what can be obtained when comparing to the state of the art models discussed in Section 3.1, the obtained accuracy is most likely high enough to be utilized in some applications.

We see that the obtained results suggest that there is little reduction of accuracy when using smaller input sizes, while there is much speed to gain from it. It is considered common practice to downsample the inputs, but in this report, the degree of downsampling is in some cases extensive. In addition, we do not provide any means of optimization in terms of accuracy regarding the downsampling but use this as a technique to speed up inference.

The means of accelerations provided in this report could to some extent be application-specific. Some of the techniques used to obtain the highest levels of performance have a larger impact on accuracy, meaning that they are in general best suited for applications in which the need for high throughput is more important than accuracy. The reduction of accuracy is still relatively low, so it might still act as a viable option given high initial accuracy.

6.2 Computing platform

In Table 5.5 the FPGA implementation is compared to the CPU and GPU implementations. When comparing the FPGA with the CPU, the FPGA is better both in terms of throughput and latency for all the implemented models. This is however not surprising, seeing that a CPU is not suited to perform DNN inference as there is little room for parallelization compared to both the FPGA and GPU. When comparing the FPGA implementation to the GPU implementation, there is some difference in terms of performance. We see that the GPU tend to perform better both in terms of throughput and latency, except for the model using $64x64$ inputs. As mentioned in Section 5.2 this is due to the much higher computational power as well as parallelization and higher memory bandwidth. When using small image sizes, there is less pressure on the bandwidth, and the FPGA is able to benefit from the parallelization and that the arithmetics on fixed-point

are simpler compared to floating-point operations. In addition to this, there is some overhead associated with the GPU acceleration as the data has to be transferred to the GPU memory before inference. As briefly discussed in Section 3.2, the authors of [26] found that in their benchmarks that the FPGA they utilized was about 2.5x more energy efficient during inference than the GPU they used. This might indicate that the FPGA could be a viable option for embedded real-time systems with limited power.

This report outlines several results on throughput and latency comparing an FPGA with a GPU, however in our benchmarks only the FPGA utilizes quantized weights. Considering the findings discussed in Section 3.2, Borkovkina et al. [31] quantized the weights for the GPU implementation which yielded a speedup of 21x compared to a 32-bit floating-point model on the same GPU. The method of quantization could also have been deployed on the CPU and GPU models in our benchmarks, which likely would have resulted in far higher throughput in both cases. This would make the gap in performance between the FPGA and GPU even larger, and most likely result in the GPU having the highest throughput for all image resolutions.

The obtained results in this report are comparable to the findings made by Miyama [30] with their Vitis AI implementation. The model is approximately $2/3$ the size of the model implemented in this report, and its throughput is 32 fps on $256x256$ in comparison to our model, which achieved a throughput of 23 fps.

6.3 Applications

In the context of embedded real-time systems there are several important measures to keep in mind during the design. First and foremost, it is desired to keep the power consumption to a minimum, especially if the system is battery-driven. Inference of DNNs entails billions of arithmetic operations which in turn means a comparably high power consumption if the means of processing is not optimized for the application. Furthermore, it is highly beneficial to strive for as low latency as possible. This is especially the case for real-time systems where the user is dependent on the output of the system in order to interact. If the outputs are produced with a latency in the order of seconds, there might not be possible for a user to react intuitively to the output.

It is also important to obtain a level of throughput high enough so that the system does not induce any bottlenecks, meaning for example that the rate of input is higher than the rate of output. In this case, one would need large buffers in order to process all the inputs and rely on the fact that the feed at some point will stop so that the remaining contents of the buffers can be processed. If this is not the case, one would overfill the buffer and lose potentially important information and induce high levels of latency. One option in this case would be to drop frames systematically to equalize the input rate to that of the throughput. However, if the throughput is high enough to process the data at a rate equal to or higher than the input rate, such a bottleneck will not occur.

The implementations made in this report show great potential for various applications. Considering a real-time system as we described in Section 4.1, certain specifications should be upheld. For instance, to provide an acceptable degree of responsiveness, the latency of the system should not be greater than 100 ms. Furthermore, the obtained throughput should be high enough to support the frame rate of the intended application in order to be able to segment all the frames in the video stream. This limits the deployment of the implemented models to the models with image resolutions lower than 256x256. Considering our

specification, a model with performance between the models using a resolution of $128x128$ and $256x256$, for example, with a resolution of $192x192$ would be best suited. This configuration also adheres to the resolution constraint. As already discussed, using a higher resolution could improve accuracy through more available information, which could mean that though we in this report find similar accuracies, there could be applications where a resolution of for example $192x192$ would offer a higher accuracy than $128x128$. Furthermore, interpolating the throughput and latency metrics, we see that the performance is sufficient to what we assume would be required in a real-time ultrasound application, namely providing a throughput close to 50 fps and a latency lower than 100 ms.

However, though the given resolution is a multiple of 16, it does not work with the implemented network, which most likely is a consequence of the compiler setting the number of inputs to a power of two, which will result in a mismatch in the number of inputs and input data. This could either be a design choice made by Xilinx or simply a consequence of the Vitis AI framework still being under development. A possible way to bypass this would be to upsample the image to $256x256$. However, this would be counter-intuitive taking the objectives of this report into account. However, extrapolating the accuracies obtained in this report, we see that in our case, there is nothing to gain from choosing the $192x192$ model over the $128x128$ model.

We base the rest of the discussion on the implementation trained on $128x128$ input images as this implementation has good levels of both throughput and latency. The number of threads used in an implementation is dependent on the data rate at the input. Using an ultrasound transducer as an example, the implementation at hand yields a minimum of 42 fps which is sufficient to process a video stream of standard quality. If this should not be sufficient, the number of threads could be increased at the price of latency. Using the maximum amount of threads, which will introduce the highest latency, the resulting system will have a framerate of 83 fps and a latency of 45 ms, which is more than sufficient for the discussed application.

If the obtained results in terms of performance are unsatisfactory, an option could be to deploy an even smaller image resolution at the price of accuracy or alternatively implement a different model with fewer layers so that the inference will be less computationally intensive. An example could be to use a U-net architecture with two steps in the contracting and expansive path as done in [31]. The FPGA used in this implementation, namely the Xilinx XCZU7EV, is an edge computing platform, and one can therefore increase the performance by utilizing a more powerful FPGA at the cost of power consumption. Another option could be to generate an application specific hardware configurations for the given model. This can either be done by hand, using a framework converting a high-level description, such as the Finn framework, or using HLS as mentioned in Section 3.3.

7 Conclusions

The main goal of this report was to speed up the inference of a DNN using an FPGA. We successfully implemented a U-net architecture for CPU and GPU using PyTorch, and for FPGA using Vitis AI. We trained the U-net architecture on segmenting cardiac ultrasound two-chamber views with four different image resolutions; $64x64$, $128x128$, $256x256$, and $512x512$, obtaining a minimum Sørensen-Dice coefficient of 89%. The maximal reduction in accuracy as a result of quantizing a floating-point model was 1% for the $512x512$ model. Using the Xilinx XCZU7EV FPGA and optimizing the U-net model by quantizing the 32-bit floating-point weights to 8-bit integers, the maximal obtained speedup was 30x compared to an Intel Core i7 CPU and 2.6x compared to an Nvidia GeForce GTX 1060 GPU. The FPGA achieved a latency 0.07x the CPU latency, and a maximal latency reduction of 0.68x the GPU latency. Utilizing an FPGA for DNN inference yield superior results compared to an CPU, effectively achieving higher throughput and lower latency for all implemented models. The FPGA achieved comparable results to a GPU when limiting the size of the input data. The FPGA obtained higher levels of throughput than the GPU for image resolutions of $64x64$ and $128x128$, and lower latency for $64x64$. The implementations were trained and evaluated using four different input sizes to investigate input size as an additional method of performance optimization. This showed that low input resolutions are able to obtain comparable accuracies while speeding up inference. In conclusion, the resulting network performed with high accuracy on small input sizes while achieving great speedup, hence proving that FPGA can be considered a viable option for DNN inference of cardiac ultrasound images in real-time systems.

References

[1] Chen Chen et al. "Deep Learning for Cardiac Image Segmentation: A Review". In: *Frontiers in Cardiovascular Medicine* 7 (2020), p. 25. ISSN: 2297-055X. DOI: 10.3389/fcvm.2020.00025. URL: https://www.frontiersin.org/article/10.3389/fcvm.2020.00025.

[2] Sarah Leclerc et al. "Deep Learning for Segmentation Using an Open Large-Scale Dataset in 2D Echocardiography". In: *IEEE Transactions on Medical Imaging* PP (Feb. 2019), pp. 1–1. DOI: 10.1109/TMI.2019.2900516. URL: https://arxiv.org/pdf/1908.06948.pdf.

[3] Xilinx. "Convolutional Neural Network with INT4 Optimization on Xilinx Devices White Paper". In: 2014. URL: https://www.xilinx.com/support/documentation/white_papers/wp521-4bit-optimization.pdf.

[4] Xilinx. "Vitis AI User Guide". In: 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/vitis_ai/1_3/ug1414-vitis-ai.pdf.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[7] H. Sharma et al. "From high-level deep neural models to FPGAs". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–12. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7783720.

[8] A. Shawahna, S. M. Sait, and A. El-Maleh. "FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review". In: *IEEE Access* 7 (2019), pp. 7823–7859. DOI: 10.1109/ACCESS.2018.2890150. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8594633.

[9] Batta Mahesh. *Machine Learning Algorithms -A Review*. Jan. 2019. DOI: 10.21275/ART20203995. URL: https://www.researchgate.net/publication/344717762_Machine_Learning_Algorithms_-A_Review.

[10] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=726791.

[11] Sibi Ittiyavirah, S. Jones, and P. Siddarth. "Analysis of different activation functions using Backpropagation Neural Networks". In: *Journal of Theoretical and Applied Information Technology* 47 (Jan. 2013), pp. 1344–1348. URL: https://www.researchgate.net/publication/290710343_Analysis_of_different_activation_functions_using_Backpropagation_Neural_Networks.

[12] Siddharth Sharma, Simone Sharma, and Anidhya Athaiya. "ACTIVATION FUNCTIONS IN NEURAL NETWORKS". In: *International Journal of Engineering Applied Sciences and Technology* 04 (May 2020), pp. 310–316. DOI: 10.33564/IJEAST.2020.v04i12.054. URL: https://www.researchgate.net/publication/342195376_ACTIVATION_FUNCTIONS_IN_NEURAL_NETWORKS.

[13] Kalin Ovtcharov et al. *Accelerating Deep Convolutional Neural Networks Using Specialized Hardware*. Feb. 2015. URL: https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/.

[14] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: (2018). arXiv: 1603.07285 [stat.ML]. URL: https://arxiv.org/pdf/1603.07285.pdf.

[15] Pavlo Radiuk. "Impact of Training Set Batch Size on the Performance of Convolutional Neural Networks for Diverse Datasets". In: *Information Technology and Management Science* 20 (Dec. 2017), pp. 20–24. DOI: 10.1515/itms-2017-0003. URL: https://www.researchgate.net/publication/322408789_Impact_of_Training_Set_Batch_Size_on_the_Performance_of_Convolutional_Neural_Networks_for_Diverse_Datasets.

[16] Ibrahem Kandel and Mauro Castelli. "The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset". In: *ICT Express* 6.4 (2020), pp. 312–315. ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2020.04.010. URL: https://www.sciencedirect.com/science/article/pii/S2405959519303455.

[17] C. Shorten and T.M. Khoshgoftaar. "A survey on Image Data Augmentation for Deep Learning". In: (2019). DOI: https://doi.org/10.1186/s40537-019-0197-0.

[18] O. Ronneberger, P.Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Vol. 9351. LNCS. (available on arXiv:1505.04597 [cs.CV]). Springer, 2015, pp. 234–241. URL: http://lmb.informatik.uni-freiburg.de/Publications/2015/RFB15a.

[19] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[20] Wikipedia contributors. *Bicubic interpolation — Wikipedia, The Free Encyclopedia*. [Online; accessed 13-May-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Bicubic_interpolation&oldid=1005441439.

[21] Xiao Tian Li and Raymond Y Huang. "Standardization of imaging methods for machine learning in neuro-oncology". In: *Neuro-Oncology Advances* 2.Supplement_4 (Jan. 2021), pp. iv49–iv55. ISSN: 2632-2498. DOI: 10.1093/noajnl/vdaa054. eprint: https://academic.oup.com/noa/article-pdf/2/Supplement_4/iv49/36091652/vdaa054.pdf. URL: https://doi.org/10.1093/noajnl/vdaa054.

[22] J. E. Hall. *Guyton and hall textbook of medical physiology*. 14th ed. Philadelphia, USA: Elsevier, 2021. ISBN: 978-0-323-67280-1.

[23] *File:Diagram of the human heart.svg*. URL: https://en.wikipedia.org/wiki/File:Diagram_of_the_human_heart.svg#filelinks.

[24] Reuben R Shamir et al. "Continuous Dice Coefficient: a Method for Evaluating Probabilistic Segmentations". In: *bioRxiv* (2018). DOI: 10.1101/306977. eprint: https://www.biorxiv.org/content/early/2018/04/25/306977.full.pdf. URL: https://www.biorxiv.org/content/early/2018/04/25/306977.

[25] E. Nurvitadhi et al. "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC". In: *2016 International Conference on Field-Programmable Technology (FPT)*. 2016, pp. 77–84. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7929192.

[26] Yuexuan Tu et al. "A Power Efficient Neural Network Implementation on Heterogeneous FPGA and GPU Devices". In: *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*. 2019, pp. 193–199. DOI: 10.1109/IRI.2019.00040. URL: https://ieeexplore.ieee.org/abstract/document/8843495.

[27] Yaman Umuroglu et al. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. ACM, 2017, pp. 65–74. URL: https://arxiv.org/pdf/1612.07119.pdf.

[28] Michaela Blott et al. "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3 (2018), pp. 1–23. URL: https://arxiv.org/pdf/1809.04570.pdf.

[29] Jiantao Qiu et al. "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. Monterey, California, USA: Association for Computing Machinery, 2016, pp. 26–35. ISBN: 9781450338561. DOI: 10.1145/2847263.2847265. URL: https://doi.org/10.1145/2847263.2847265.

[30] Masayuki Miyama. "FPGA Implementation of 3-bit Quantized CNN for Semantic Segmentation". In: *Journal of Physics: Conference Series* 1729 (Jan. 2021), p. 012004. DOI: 10.1088/1742-6596/1729/1/012004. URL: https://doi.org/10.1088/1742-6596/1729/1/012004.

[31] Svetlana Borkovkina et al. "Real-time retinal layer segmentation of OCT volumes with GPU accelerated inferencing using a compressed, low-latency neural network". In: *Biomed. Opt. Express* 11.7 (July 2020), pp. 3968–3984. DOI: 10.1364/BOE.395279. URL: http://www.osapublishing.org/boe/abstract.cfm?URI=boe-11-7-3968.

[32] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[33] Markus Nagel et al. "Data-Free Quantization Through Weight Equalization and Bias Correction". In: (2019). arXiv: 1906.04721 [cs.LG]. URL: https://arxiv.org/pdf/1906.04721.pdf.

[34] Jonas Sundseth. *Acceleration of U-net using Vitis AI: Code base*. 2021. URL: https://github.com/jonassundseth/U-net_Vitis-AI.

[35] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: (2015). arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/pdf/1512.03385.pdf.

[36] J. Gordon Betts et al. *Anatomy and Physiology*. OpenStax, Houston, Texas, 2013. URL: https://openstax.org/books/anatomy-and-physiology/pages/19-1-heart-anatomy.

[37] Kirk T. Spencer et al. "Focused Cardiac Ultrasound: Recommendations from the American Society of Echocardiography". In: *Journal of the American Society of Echocardiography* 26.6 (2013). ASE 24th Annual Scientific Sessions, pp. 567–581. ISSN: 0894-7317. DOI: https://doi.org/10.1016/j.echo.2013.04.001. URL: https://www.sciencedirect.com/science/article/pii/S0894731713002599.

[38] Hayit Greenspan, Bram van Ginneken, and Ronald M. Summers. "Guest Editorial Deep Learning in Medical Imaging: Overview and Future Promise of an Exciting New Technique". In: *IEEE Transactions on Medical Imaging* 35.5 (2016), pp. 1153–1159. DOI: 10.1109/TMI.2016.2553401. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7463094.

[39] Xilinx. "ZCU104 Evaluation Board User Guide". In: 2018. URL: https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf.

[40] Bingzhe Wu et al. "Reducing Overfitting in Deep Convolutional Neural Networks Using Redundancy Regularizer". In: *Artificial Neural Networks and Machine Learning – ICANN 2017*. Ed. by Alessandra Lintas et al. Cham: Springer International Publishing, 2017, pp. 49–55. ISBN: 978-3-319-68612-7. URL: https://link.springer.com/chapter/10.1007/978-3-319-68612-7_6.

[41] Nurshazlyn Mohd Aszemi and Dhanapal Durai Dominic Panneer Selvam. "Hyperparameter Optimization in Convolutional Neural Network using Genetic Algorithms". In: *International Journal of Advanced Computer Science and Applications* 10 (June 2019), pp. 269–278. DOI: 10.14569/IJACSA.2019.0100638. URL: https://www.researchgate.net/publication/334151021_Hyperparameter_Optimization_in_Convolutional_Neural_Network_using_Genetic_Algorithms.

Appendix

A Workflow of generating FPGA models

The code provided along with this project can simply be run using the tutorial presented here.

Training the model

There are several arguments that can be parsed, use `--h` to display the options. To train the model with a dimension of 128x128 simply run:

```
$ python3 unet.py --train --dim 128 --input PATH_TO_DATASET/
```

Quantizing the model

Running the following steps of the pipeline requires use of the docker prvided in Vitis-AI. From inside the Vitis AI folder simply run the command:

```
$ ./docker_run.sh xilinx/vitis-ai:latest
```

Installing the dependencies is done by running:

```
$ pip3 install -r requirements.txt
```

As we have implemented the U-net class with the training algorithm as part of the class we simply use a script containing an identical model and omitting the training function; models.py. When the accuracy of the model is satisfactory the model can be quantized to int8 by running:

```
$ python3 models.py --input PATH_TO_DATASET/ --model TRAINED_MODEL.pth --dim 128 --
  quant_mode calib
```

To test the impact on accuracy of the quantized model simply run:

```
$ python3 models.py --input PATH_TO_DATASET/ --model TRAINED_MODEL.pth --dim 128
--quant_mode test
```

To deploy simply add the deploy flag:

```
$ python3 models.py --input PATH_TO_DATASET/ --model TRAINED_MODEL.pth --dim 128
--quant_mode test --deploy
```

Compiling the model:

In the Vitis-AI docker compile the model from the previous step by running:

```
$ vai_c_xir -x quantize_result/unet_int.xmodel -a /opt/vitis_ai/compiler/arch/
DPUCZDX8G/ZCU104/arch.json -o quantize_result/ -n unet_deploy_128
```

which in this case is compiled for the ZCU104.

Running the deployed model on FPGA:

Use the serial connection to set the FPGAs ip-adress, thereafter transfer the generated files and datasets to the FPGA using `scp` commands. Simply run the quantized model on the FPGA using:

```
$ ./run.sh
```

make sure that the arguments within the shell script is correct.

NTNU

Kunnskap for en bedre verden