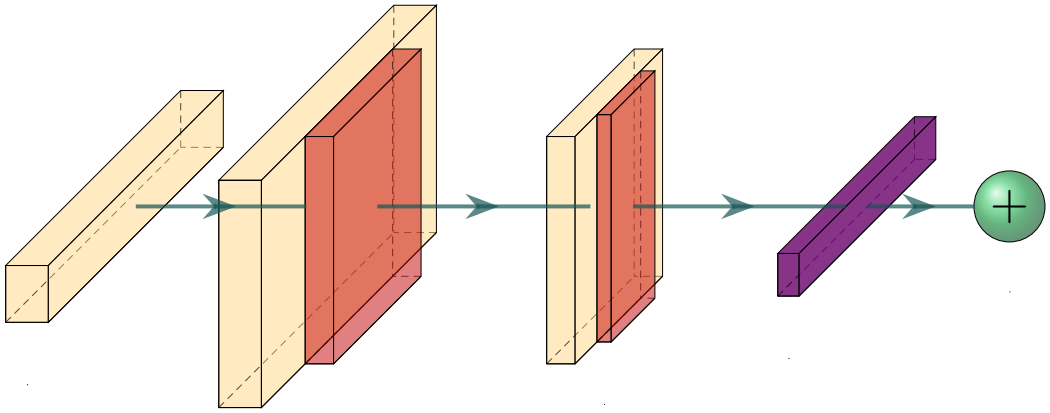
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Classification of Radar Targets Using Neural Networks on Systems-on-Chip

Master's thesis in Embedded Electronic System Design

GUSTAV HOLST

CHRISTOFFER KRULL

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Classification of Radar Targets Using Neural Networks on Systems-on-Chip

GUSTAV HOLST
CHRISTOFFER KRULL

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Classification of Radar Targets Using Neural Networks on Systems-on-Chip
GUSTAV HOLST
CHRISTOFFER KRULL

Supervisor: Lena Peterson, Department of Computer Science and Engineering
Advisor: Christian Takvam, Saab AB
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: The convolutional neural network model used in this project.

Typeset in LaTeX
Gothenburg, Sweden 2021

Classification of Radar Targets Using Neural Networks on Systems-on-Chip
Gustav Holst
Christoffer Krull
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The fast paced nature of modern combat situations has increased the need for quick and adaptable radar classifications to identify potential threats. One potential option to increase accuracy and performance is to introduce machine learning in radar classification tasks. Utilising convolutional neural networks to identify patterns within radar data provides an additional stream of information that could be used to classify targets. These neural nets require high-performance processing whilst still conforming to the low power and mobility requirements inherent in defence applications. These requirements make FPGAs a natural choice to be used as a hardware platform in radar classification tasks.

This project explores the capabilities of the new Xilinx Versal VCK190 ACAP which combines regular FPGA architecture with AI Cores, which can be used to accelerate neural network tasks. Our findings show that the amount of radar classifications per second can be increased by at least 20x compared to a neural net running on a consumer grade CPU. This increase was achieved by utilising the low latency interfaces and high performance acceleration of the AI cores which are unique to the new Versal platform. These aspects make the VCK190 an interesting platform to further develop upon but more research needs to be made to improve the accuracy of the model.

Keywords: Xilinx, Versal, VCK190, Convolutional neural networks, Radar classification, FPGA, Thesis.

Acknowledgements

Thank you to Saab AB for allowing us to pursue such an interesting project. A special thanks to Christian Takvam for his help as supervisor at the company and to Lena Peterson for her role as supervisor at Chalmers. Thank you to Xilinx for answering questions as well as providing introductory courses for the platform. We also want to thank Daniel Wallström for allowing us to work on this project and we want to thank Per Larsson-Edefors for his role as examiner during the project.

Gustav Holst & Christoffer Krull, Gothenburg, June 2021

Contents

List of Abbreviations

ACAP	Adaptive compute acceleration platform
AIE	Artificial intelligence engine
ANN	Artificial neural network
ATR	Automatic target recognition
AXI	Advanced extensible interface
CLE	Cross layer equalisation
CNN	Convolutional neural network
CPU	Central processing unit
DMA	Direct memory access
DPU	Deep learning processing unit
FCL	Fully connected layer
FFT	Fast fourier transform
FPGA	Field-programmable gate array
GPU	Graphics processing unit
IQ (data)	In-phase and quadrature components
LB	Logic block
NN	Neural networks
NoC	Network-on-Chip
PCIe	Peripheral component interconnect express
PESA	Passive electronically scanned array
PI	Programmable interconnect
PL	Programmable logic
PRF	Pulse repetition frequency
RAM	Random-access memory
ReLU	Rectified linear unit
RISC	Reduced instruction set computer
SIMD	Single instruction multiple data
SOC	System on a chip
VLIW	Very long instruction word

1
Introduction

Modern radar systems gather vast amounts of data to accurately identify incoming threats using advanced algorithms [1]. These algorithms could potentially benefit from a convolutional neural network (CNN) that is trained to recognize common radar targets. But a CNN that analyses large amounts of data requires high processing power, usually found in graphic processing units (GPUs) [2]. Therefore, the need for high-performance processing in a small, durable and cost effective form factor becomes apparent. This need could possibly be satisfied using modern field-programmable–gate-arrays (FPGAs), which are a more configurable alternative to regular computers. FPGAs are already commonplace in military and space applications due to being highly customisable, power-efficient and portable [3]. But most FPGAs on the market don't excel at the types of tasks carried out by neural networks (NNs), due to them requiring high levels of parallelism [4].

The new Versal adaptive compute acceleration platform (ACAP) [5] from Xilinx is a advancement on the FPGA architecture which introduces AI Engines [6]. AI Engines are designed to handle large vector arithmetics in parallel, which is done by interconnecting a grid of smaller processors. This allows vector operations to be handled with minimal latency within the grid and Xilinx claims that this will greatly improve performance for neural network tasks [7]. The ACAP also has other advancements compared to regular FPGA architectures; the most notable one being programmable network-on-chip (NoC) which allows for low latency data transfer throughout the device. This low latency could minimise delays caused by input and output from the system as well as reading and writing to the memory. This project aims to evaluate the capabilities of the Xilinx Versal AI Core Series VCK190 production board [8]. The focus will be upon the AI Core feature of the platform by implementing a CNN tasked with classifying radar targets. Evaluation of the platform will be done in comparison with the same CNN running on a consumer grade CPU and will focus on metrics such as throughput.

1.1 Versal VCK 190 platform

The Versal VCK 190 platform is the flagship product in the current Versal iteration produced by Xilinx. The platform combines the regular FPGA architecture with the new AI Engine hardware block, which contains 400 individual AI Cores. This platform is intended to target real-time digital signal processing tasks as well as machine learning applications. A block diagram of the architecture can be seen in Fig. 1.1, which shows the scalar and adaptable engines that have become more

common in Xilinx FPGA architecture in recent years. It also shows the intelligent engines and NoC which are more advanced features of the new ACAP architecture.

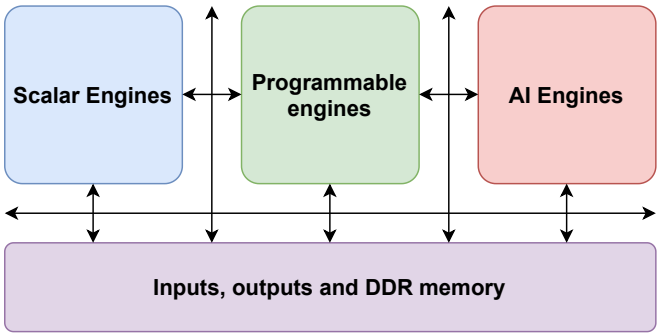


Figure 1.1: Functional diagram of Versal VCK 190.

The Versal VCK 190 offers the possibility to optimise the implementation for latency and deterministic performance, whilst still achieving good throughput even on a smaller batch size due to their flexible memory hierarchy [9]. The flexible memory hierarchy can be utilised in conjunction with AI Engines to increase compute efficiencies for neural network tasks of up to two times compared to leading GPUs [9]. The new 7nm Versal platform also allows for higher power efficiency in comparison to previous FPGA iterations [7]. This efficiency proves highly desirable in an active radar environment which, often has a limited power budget and strict latency requirements [10].

1.2 Neural networks

The term neural networks stems from the actual neural pathways present in a human brain and the way in which they interact. These complex brain functions can be simplified and turned into algorithms which can adapt and learn from the data they process, much like an actual neural network but on a smaller scale [11]. Artificial intelligence is nothing new in the world of automatic target recognition (ATR) and threat assessment. In fact the defence and aerospace industry have been among those driving this technology forwards [12],[13]. But this technology is also becoming more common in the automotive sector with autonomous vehicles [14]. The reasons are readily apparent as these systems are essential in providing information for different systems. The types of tasks that ATR systems carry out are perfectly suited for learning algorithms that can increase reliability and performance.

Neural network inference is a computationally intensive workload, often requiring billions of operations per second and 100s of MB of parameters [4]. One of the most commonly used accelerators for this type of workload are GPUs which offer a high degree of parallelism [2],[15]. One drawback of GPUs is that to achieve a high throughput, large data batch sizes are typically required which result in increased latency [16].

The use of neural networks and artificial intelligence (AI) for targeting algorithms is becoming more commonplace in active radar technology [17] and with the intro-

duction of AI Engines there is an opportunity to evaluate performance compared to current software based solutions running on a CPU.

1.3 Ground-based radar

Modern ground-based air-defence radar solutions are often used as early warning systems to protect against aerial attacks and ballistic targets [18]. Thus quick and accurate target recognition is essential in giving civilians and military personnel vital time to react to incoming threats. A typical ground-based radar system sweeps an entire 360° field at a rate of one to two seconds [18]. The range of ground-based radar arrays varies depending on the task at hand; short range radars covering around 50 km, medium range 200 km and long range around 500 km [18]. The amount of data captured by the radar during one revolution partially depends on the range and pulse repetition frequency (PRF) of the radar array [19]. PRF refers to the rate of pulses sent out by the radar during operation. The PRF of the radar array depends both on the accuracy required and the range of the radar. A higher PRF is preferable for short-range radars as there will be less time between sending the signal and receiving the echo response. The opposite is true for long-range radars, as the echo response takes a longer time to travel back to the array [19].

1.4 Problem description

AI in radar system applications have shown promise but the hardware required to achieve sufficient performance has hampered the implementation of these systems. Because ground and air based radar solutions are inherently portable they have limitations both in terms of size and power consumption. The new Versal VCK 190 platform opens up new possibilities as new AI features could be integrated into existing FPGA architecture on the same platform. This integration would improve performance both in reducing latency between different systems and by utilising the new AI Engine architecture.

1.4.1 Purpose

The primary mission of this project is to analyse the potential performance benefits of the Versal VCK 190 platform when implementing a CNN in comparison to the same network implemented in software. An existing TensorFlow [20] implementation of the CNN written in Python has been provided by Saab. The CNN has been trained to perform classification of common radar targets in the form of airplanes, jets, drones and helicopters. The implementation uses matrices of IQ radar data describing the strength of the radar signal after pulse compression as an input. The TensorFlow neural network consists of complex floating-point IQ values and depending on the precision required these can be either 16-bit or 32-bit values. The Versal VCK 190 board has the ability to use 8-, 16- or 32-bit values to represent floating-points, depending on the accuracy required.

The project will be split into two parts which explore different methods of creating the CNN within the Versal architecture. The first part utilises Xilinx proprietary software Vitis-AI [21] which is able to compile a TensorFlow model into code that can be run on the Versal platform. This conversion should create an implementation that can be directly compared to the software implementation but it does somewhat limit the scope of the project. The implementation utilises a deep learning processing unit (DPU) core to accelerate neural networks tasks. This DPU core is a Xilinx IP component and cannot be modified which could potentially limit performance. These problems made it apparent that a second part of the project needed to be implemented, which involved building a CNN from the ground up in C++. The implementation utilises Petalinux [22] which is a lightweight unix operating system designed for Xilinx devices. The operating system allows for full configurability of the platform and the implementation will be customised to fit the small size of the CNN model. The TensorFlow model provided by Saab will act as a reference during this part of the project, providing the specification for layers, nodes and weights. The custom implementation will not be trained but instead import the weights used by the TensorFlow model. This implementation gives the project a chance to explore wider possibilities with the Versal platform and avoids the limitations the come with the DPU implementation.

The two implementations will be considered functional if they provide the same level of accuracy as the quantized TensorFlow model while performing inference on a set of test data provided by Saab. The project focuses on the hardware architecture and the achieved performance, which will be compared to that of the software implementation developed by Saab, running on a CPU. The second goal is to evaluate the system, with the main performance metrics being the latency (time from input to output of the CNN) and throughput. Additional metrics of interest will be the resource allocations for the implementations. To summarise, the goals of the project are to:

- Implement a CNN and other system components on a Xilinx Versal VCK190
- Compare performance to TensorFlow results using the following metrics:
  - Latency
  - Throughput
  - Resource allocation

1.4.2 Scope and limitations

The focus of this thesis is the hardware implementation of a neural network using the Xilinx ACAP platform and evaluating the potential benefits and shortcomings of this implementation. This involves utilising an existing CNN graph and adapting it to work on a new platform. The performance of the hardware implementation was compared to that of a CPU implementation done using the same CNN. Because the TensorFlow model is a small CNN the training will be done on a regular CPU, requiring no more than 10 minutes. No training will therefore be done on the Versal platform, this would theoretically be possible but would require a larger amount of development time. The custom implementation will not be trained in any way but instead import the weights and biases generated via the TensorFlow model, the

intention being that these two systems should be comparable. The project does not intend to delve into topics regarding the signal processing side of the radar data. The input radar data used for this thesis was captured by Saab AB. Both implementations utilised pre-processed data which contained cutouts around targets and labels for each sample.

1.5 Thesis outline

The report structure starts with a chapter that will take a closer look at the technical background of the project. Starting out with a section on ground-based radar systems, this section gives insight into the type of data that will be utilised by the system. This section will also briefly talk about the equipment used to capture this data and how the system might fit into that infrastructure. The chapter continues with a section looking at convolutional neural networks, both the theory behind them, how they are implemented and application areas they might be used for. The next part of the chapter introduces the hardware platform utilised for this project and the features that make it an interesting platform to evaluate. The approach taken during the project will be described in chapter 3. It will describe how the neural network was implemented on the hardware platform as well as other components that are required for the system to function. It will also look at the evaluation methods used during this project and why the chosen parameters for performance are relevant to the project. In chapter 4 the results of the project will be shown, this includes both the output from the completed system as well as the results from the performance evaluation. These results will be discussed in chapter 5 and an evaluation of the hardware platform will be made. The conclusions from these discussions can be found in chapter 6 where several aspects of the project are weighed against each other.

2
Technical background

The following sections gives an insight into the technical background to give a better understanding of the implementations developed in this project. The chapter includes sections on radar theory, artificial neural networks and the hardware platform itself.

2.1 Radar and beamforming systems

This section aims to give the reader a basic understanding of the ground-based radar systems that were utilised during this project to gather testing data. The section will give a brief overview of general radar systems.

2.1.1 Passive Electronically Scanned Array

A passive electronically scanned array utilises an array of antennas to send a coordinated wave of radio signals using a steered beam [23] which can be seen in Fig. 2.1. The emitted radio waves bounce off potential targets and return to the array which captures incoming signals via a sensor array and locates targets via beamforming. A passive electronically scanned array (PESA) is able to steer emitted waves into a plane wave by phase shifting the feed for each antenna which delays the array progressively, allowing the radio waves to overlap and form a wavefront. The beam can quickly change direction by altering the phase shift value for each antenna, this is usually done in a predetermined scan pattern at high frequency to gather sufficient data to build a three dimensional model of the target [23].

2.1.2 Beamforming

A beamformer refers to a set of sensor arrays connected to a processing unit with the purpose of amplifying signals from a specific direction or angle [24]. This arrangement allows a beamformer to perform spatial filtering to determine the location of an incoming signal, in this case radio waves that have reflected from a potential target. By utilising arrays of sensors the beamformer can map out incoming signals in three dimensions, giving a precise location to the source of the signal [25]. The signals received by the sensors are first delayed in proportion to the delay it takes for the signal to travel to the different sensors. The signals are then multiplied by complex weights and summated, giving a complex value which is added to a three dimensional output array. Fig. 2.2 gives a general overview of a radar beamforming system.

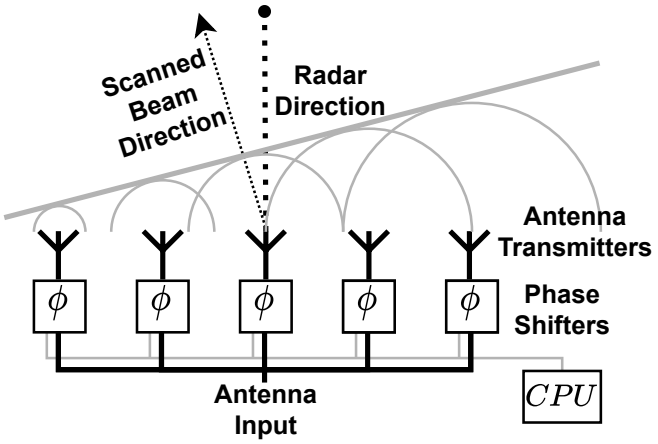


Figure 2.1: PESA array with beam steering via phase shifters.

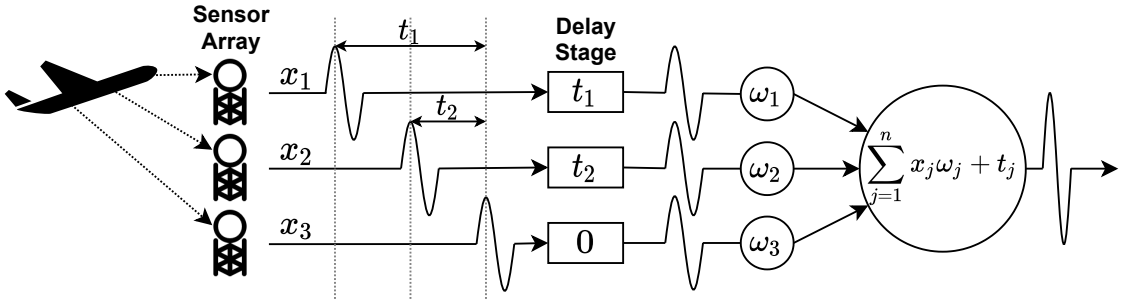


Figure 2.2: General overview of beamforming system. Adapted from: [26]

2.2 Artificial neural networks

The basis of this section is to provide a fundamental understanding of the neural network architecture used as a basis for this thesis. The section will give a brief background on what neural networks are and explain how they are implemented.

2.2.1 History of neural networks

The neural networks described in this chapter draw their inspiration from the neurological findings of researchers dating back to the 15th century. The most basic neural network designs were created in 1960 and only had a single layer between input and output, which limited training [27]. But the history of artificial neural networks is much more recent and the first articles started appearing in 1970. In 1986 a break through was discovered when backpropagation was introduced as an alternative during training [28]. The defining feature of this algorithm is that it propagates through the network backwards, training the neurons in the layer closest to the output first. This process means that each weight that is calculated is impacted by the weights of the layers in the previous layers, increasing accuracy. This method of training is still used today in CNNs due to the advantages it offers in image recognition. Deeper research on the topic started out in the 1990 and

continues to this day as the subject evolves in unison with increased performance of memory and processors. The field of ANNs is still in its early stages and advances in network architecture evolves every year. It is the task of the developer to find a suitable network architecture for the task at hand and whilst modern ANNs often excel at certain tasks there are many older models that could be re-explored using modern hardware.

2.2.2 Artificial neuron implementation

Artificial neural networks implement a highly simplified version of a biological neural model. In the simplified model each neuron is limited to two states, active and inactive. Each neuron has a given number of inputs (x_j), depending on the number of neurons present in the network. Each input has an adjustable weight (w_j) which allows the network to influence how much effect an input has on the output. These weight values are adjusted dynamically by the ANN during training to find the correct relation between input and output. The dynamic training is done by having a specific set of inputs used for training where each sample has a label identifying the correct output. Once the ANN is trained each input along with its weights is summated along with a bias (θ_i) as seen in (2.1).

$$y(i) = \sum_{j=1}^{n} x_j \omega_j + \theta_i \tag{2.1}$$

The bias is used as a threshold to adjust the impact of the neuron on the network. The activation function that follows the summation is used to set the neural output to active or inactive depending on the outcome of the summation along with the given bias [11]. Different activation functions are used depending on the task at hand, a common one used is the rectified linear unit (ReLU) activation function which can be seen in (2.2).

$$\varphi(i) = \begin{cases} y(i) & \text{if } \sum x\omega + \theta \geq 0 \\ 0 & \text{if } \sum x\omega + \theta < 0 \end{cases} \tag{2.2}$$

As shown in Fig. 2.3 the neuron produces a single output $\varphi(i)$ which is fed to all neurons in the next layer.

2.2.3 Convolutional neural network architecture

There are many different ways of modeling a neural network based on the types of problems they are implemented for [29]. In the field of signal processing the CNN stands out due to its ability to process high-dimensional input data using several different layers [30]. This allows a CNN to map complex problems such as LiDAR or radar data in a three-dimensional (3D) array allowing for more advanced forms of recognition [31].
The basic layout of a CNN consists of an input map which takes the form of an array of input values as seen in Fig. 2.4. In most cases CNN is utilised for image processing which means that these values take the form of pixels. In the case that

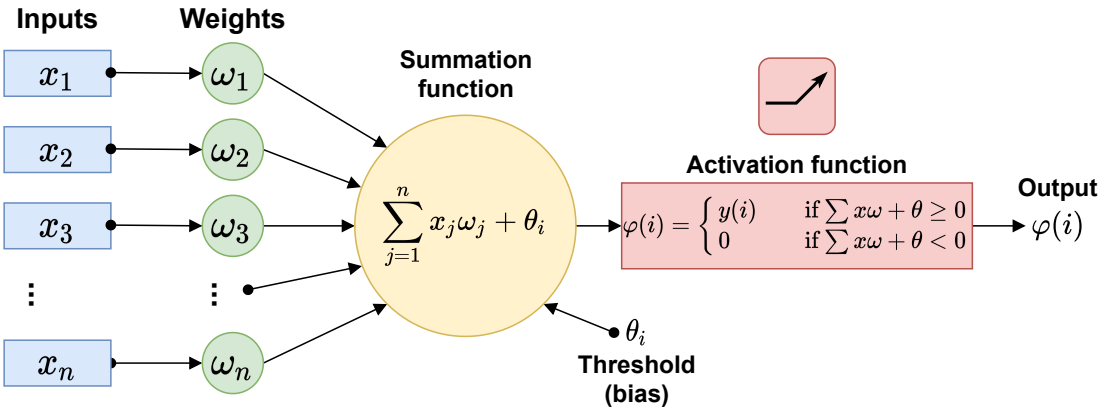


Figure 2.3: Schematic of artificial neuron. Each neuron has the index i and takes n inputs.

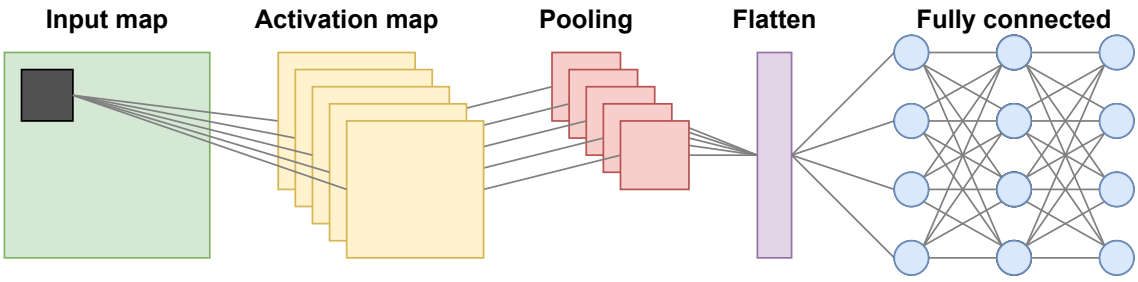


Figure 2.4: Basic layout of CNN including input mapping, activation map, pooling and finally fully connected layers.

the input data is three-dimensional the input map can come in several layers, each layer representing a certain depth in the 3D image. Activation maps are generated from the input maps using the convolutional layer, in some cases each input map can generate several activation maps, depending on how many parameters the CNN is looking at. Each activation map needs to be minimized using a pooling layer to reduce the size of the fully connected network. This pooling can be done using either the maximum or the average method but maximum is more common in image recognition to find outlying values [11].

2.2.4 Input layer

The input layer is the first layer within a CNN and it consists of a map containing the input data sets. From this map several different layers gather data depending on the features they are focused upon such as corners, edges and other areas of the map [11]. The input map can also come in several dimensions depending on the input data mapped. The process remains the same and subsequent layers gather relevant data from their respective areas.

2.2.5 Convolutional layer

The main feature of a CNN is the convolutional layer which utilises convolution between a convolution kernel and the input data. This process can be done in several layers utilising different input streams such as a three dimensional array of IQ data. Each input stream is iterated through using a stride value and the dot product is calculated between the input and the kernel. The result from the convolution is a two dimensional activation map for each layer [30] as seen in Fig. 2.5.

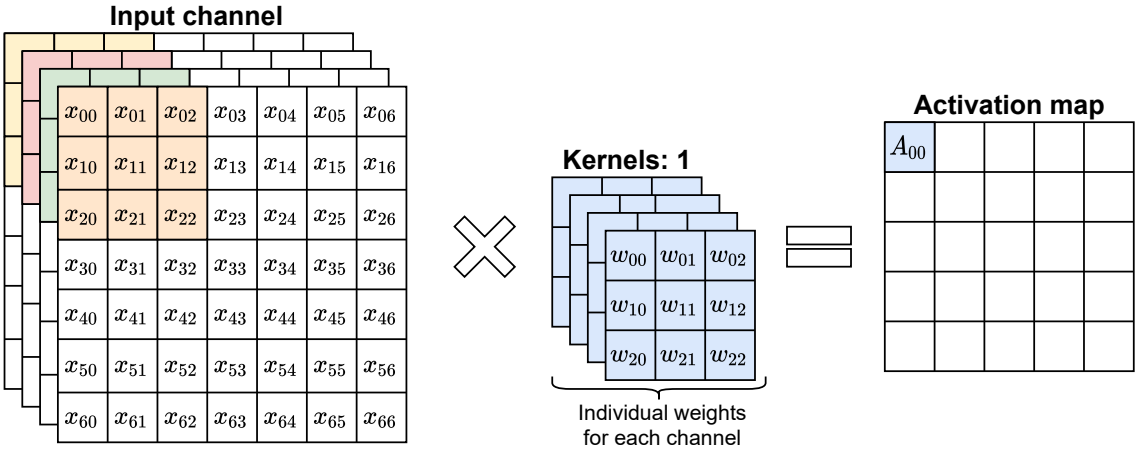


Figure 2.5: Convolution with 7x7x4 inputs and a 3x3 convolution kernel. The activation map is the dot product of x and w.

In a CNN the kernel weights are adjusted during training to find features in the input data that lead to correct conclusions. But convolution layers are a common occurrence in image processing and can be used to add blur and distortion effects to images. All of these use cases depend on the kernel weights, dimensions and stride length. The use of convolution layers does not necessarily mean that the input data is shrunk. By using zero padding and additional kernels the amount of data gathered increases exponentially as seen in Fig. 2.6.

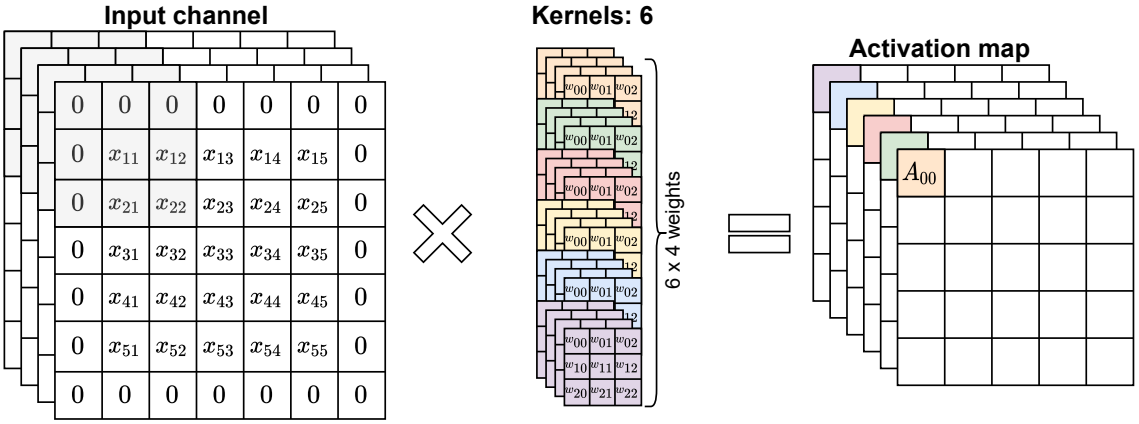


Figure 2.6: Each kernel has individual weights for each channel and the convolution between these are summated into one activation map per kernel.

Each additional kernel added has its own individual weights which makes the convolutional layer able to distinguish several different features in a single layer. To achieve true convolution the kernel should be reversed as it iterates upon the input, this can be seen in Fig. 2.7. If the kernel is not reversed then the layer simply calculates the cross correlation of the kernel and the input. But since the weights are estimated via training this does not matter, the flipping would be counteracted by changing the weights. This detail only becomes a factor when using weights from a CNN in another network, if flipping was done during training it must be done during all subsequent testing.

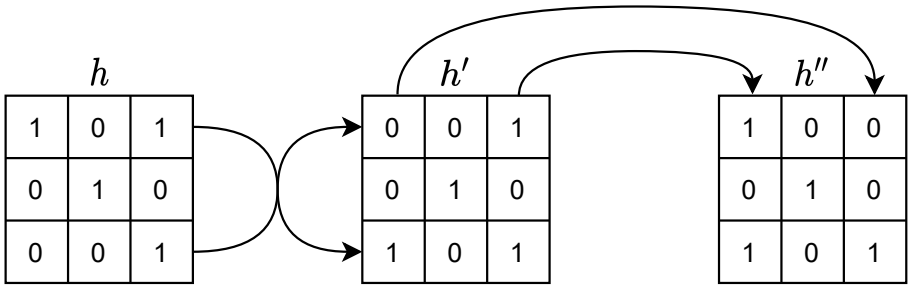


Figure 2.7: Flipping of kernel, sometimes done in convolutional layers.

2.2.6 Pooling layer

The pooling layer is commonly used in CNN to decrease the size of the activation map. This is beneficial to limit the amount of data that is being processed. Different pooling methods are used depending on the task at hand. An average pooling method will even out the results of the activation map, this means that outlying values might not be detected, which in some cases is preferred. A maximum pooling layer is more commonly used as this allows distinguishing features in the activation map to stand out. The act of shrinking the amount of data available might seem unintuitive if accuracy is the main focus of the CNN. But if used correctly the pooling layers allow the CNN to more clearly map the features that lead to a correct conclusion. Both methods of pooling can be seen in Fig. 2.8

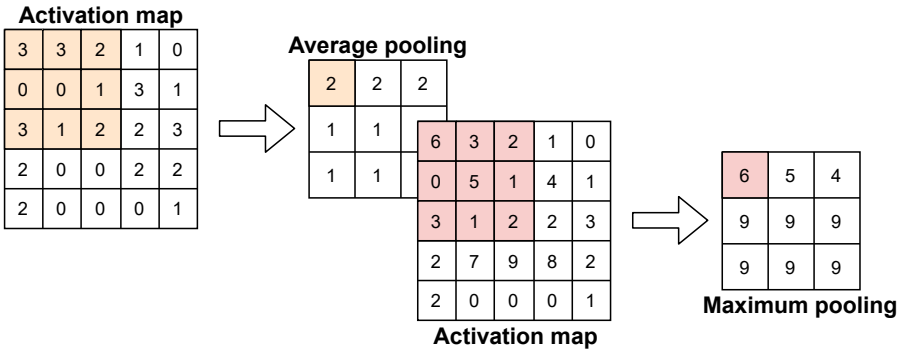


Figure 2.8: Average and maximum pooling. The 3x3 kernel takes either the average of the 9 values or the maximum.

2.2.7 Batch normalization layer

A common method for speeding up the training of a neural network is using batch normalization. This process involves using backpropagation to shift and normalise the data sets for the hidden layers [11]. Claims have been made that batch normalization reduces the impact of covariate shifts in hidden layers. Covariate shifts occur when a preceding layer in the network changes their output value due to shifting parameters, this change effects the upcoming layers, making the network more difficult to train. The claims that batch normalization reduces these effects have been disputed but the inherent quality of this layer allows for more normalised data sets, speeding up training and giving a similar result as that of eliminating covariate shifts [32].

2.2.8 Flatten layer

The flatten layer is used to reshape data that has been convoluted and pooled into different dimensions that can easily be modeled onto a fully connected layer. It is commonly used after pooling to reshape several layers of data into one graph. The reason why this is done at the very end of the CNN is due to the size of the network. As the data gets convoluted and pooled it becomes minimized and unnecessary information is discarded. Once this is done the data set is small enough to be flattened and fully connected.

2.2.9 Fully connected layer

The fully connected layer (FCL) also known as dense layer (DL) consists of the artificial neurons described in section 2.2.2. What separates a CNN from other NNs is the convolution and pooling that takes place before the FCL to minimize the neurons that are required to process the task. The FCL starts with an input layer which flattens and reshapes the pooled data that will be mapped to the neurons, this can be seen in Fig. 2.9. Each neuron takes inputs from previous neurons in the network and passes the produced output to the next set of neurons using hidden layers. This process allows the fully connected layer to pick out the values that are deemed important in identifying a specific target. These values are used by the final set of the fully connected, which is called the output layer, to determine if the inputs can be labeled as a specific target [29].

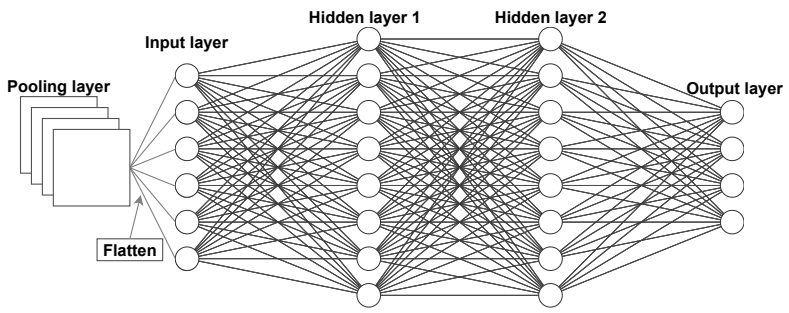


Figure 2.9: Fully connected layer with two hidden layers and four outputs.

2.2.10 Training

One of the standout features of an artificial neural network is the ability to learn from the given sample data and map a relationship between the inputs and the expected output. This learning process is done by adjusting the weights and bias of the network in small steps to create a generalized map of the system. These steps are often referred to as learning algorithms and together they create a training set. A complete run of all the sample data within a training set is often referred to as an epoch. Depending on the complexity of the relationship between inputs and outputs there might be a need for several training epochs to be run before a NN is fully trained [27]. This is an important aspect as running too few epochs might result in underfitting which means that the NN is unable to adapt efficiently to the sample data. Similarly if too many epochs are run the NN can suffer from overfitting which means that the system is too attuned to the samples used during training sets and cannot recognize other sample data outside the training set [29].

2.2.11 Supervised learning

During supervised learning the sample data for each training set has a given outcome which has been predetermined. The focus of this training is to coach the NN into finding the relations between input and output data. This requires a specific type of input data that has been formatted in such a way that it has a clear outcome which is obtainable as an output. The specific data used is referred to as a training set and is only used during training. Validation of the training is done using a separate validation set which reduces the risk of overfitting. During this process the weights and bias of the NN will be adjusted to find relevant connections in the sample data that leads to a correct output. The NN will be deemed sufficiently developed once it reaches a maximum accuracy for the validation set [27].

2.2.12 Unsupervised learning

The process during unsupervised learning is similar in many ways but it lacks the predetermined knowledge of the sample data output. This lack of output means that the NN needs to make certain assumptions about the sample data that it receives and adjust weights and biases in a way that seems plausible to end up with outputs within a given range. This method is useful in cases where the outcome of a problem is not readily apparent and further insight is required [11].

2.2.13 Evaluation of performance

Performance of the fully trained model is evaluated via a number of metrics. These metrics are based on the models ability to perceive the intended target and how often it misses. A fundamental part of this evaluation is the confusion matrix which allows insight into correctly identified targets and false-positives.

The most useful metrics can be seen in Fig. 2.10, recall and precision being the most important. Recall allows developers to see how many relevant items are selected by the model and precision specifies how many selected items are relevant. These two

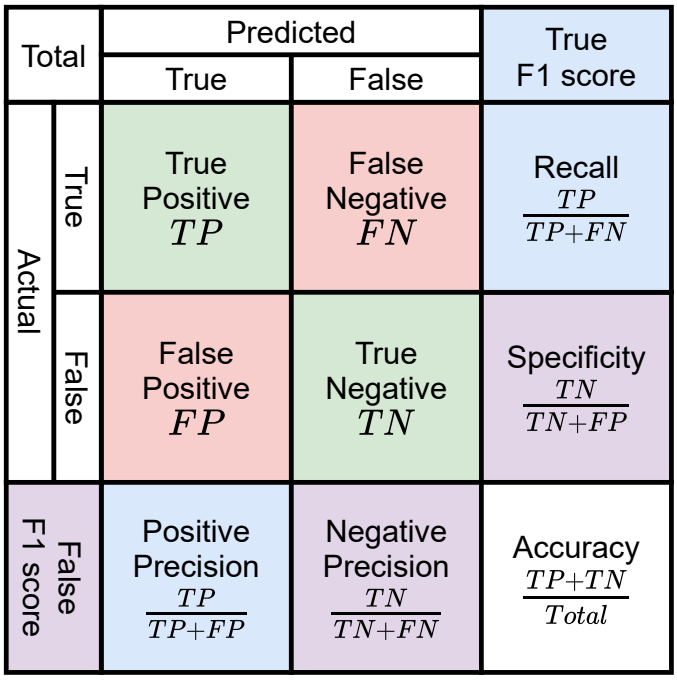


Figure 2.10: Confusion matrix showing useful metrics for CNN evaluation.

metrics can be used to calculate the F_1 score, which is a measurement of the accuracy of the model. The score is a harmonic mean of the precision and recall metrics and has a maximum value of 1 and a minimum of 0.

The F1 score can be calculated as

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

2.2.14 Application areas

The ability to model nonlinear problems make ANNs a useful tool in a number of different application areas. In prediction systems the ANNs can be used to estimate the next set of values given a number of samples. This can be used to estimate financial trends [33], predict the behaviour of different targets [34] and to recognise and classify a number of different signals [35]. Pattern recognition and classification is a particularly application for CNNs as these deep networks allow for mapping multilayered signals. These types of systems also have a given set of expected outcomes, making the network easier to train [27].

2.3 Hardware platform

This section will further describe the hardware platform that was evaluated and some of the features that made it a suitable target for this project. The ACAP device category was launched by Xilinx in unison with the new Versal series. This category was launched as an iteration upon the regular SoC and FPGA platforms produced by Xilinx and attempts to integrate the features of both into a single device along

with new standout features such as AI Engines and NoC. The platform has access to several high-speed interfaces such as eight lanes of PCI-express 4.0 which is a high-speed bus commonly found on most GPUs and motherboards. There is also an Ethernet interface which has a rate of 600 Gbps allowing for quick communication with external devices in the system [5]. These features make the Versal ACAP an especially interesting platform for projects involving real-time signal-processing and neural networks. It also allows projects implemented on older FPGA platforms to utilise the 7nm process which provides great benefits to performance, power efficiency and area usage.

2.3.1 Scalar processing system

The scalar processing system consists of a dual-core Arm Cortex-A72 and a dual-core Arm Cortex-R5F for safety-critical real-time applications. These processors use state of the art 7nm processes allowing for much lower power consumption and higher density than previous scalar processors. The platform is able to utilise the scalar processors for running real time applications such as lightweight operating-systems. This feature allows developers to create more advanced systems that can be tweaked and debugged via console commands running on the platform itself.

2.3.2 Programmable logic

The programmable logic (PL) utilises regular hardware-level programming as well as high-level synthesis to create hardware implementations of functions. The VCK190 platform consists of building blocks in the form of 1 968 400 system logic cells, 899 840 lookup tables and a distributed RAM of 27.5 Mbit. Together with programmable interconnect these building blocks can create logic circuits for real-time components. The PL uses regular FPGA architecture with logic blocks (LB) connected to programmable interconnects (PI) which allows several LBs to be connected together as is described in Fig. 2.11.

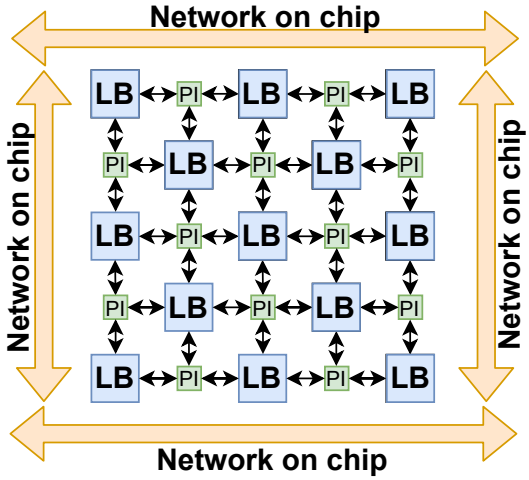


Figure 2.11: Programmable logic module consisting of LBs and PIs.

2.3.3 AI Engines

The VCK190 platform's main feature are the 400 interconnected AI Engines that promise to massively improve performance of neural network tasks compared to previous Xilinx platforms. Each AI Engine contains its own 16kB program memory as well as an 32b RISC scalar unit and 512b fixed and floating point vector unit. The AI Engines sit in an array where each nearby engine tile is interconnected with each other as seen in Fig. 2.12. These interconnections allow for multiple levels of parallelism; the first is the instruction level where each AI Engine can carry out two scalar operations, two loads, one vector multiplication and one store in a single clock cycle. The second data level parallelism is accomplished by vector-level operations which allows for multiple sets of data to be processed on a per clock-cycle basis. The AI Engine array sits within the architecture of the Versal platform and is connected to the programmable logic, scalar processors and IO via the NoC.

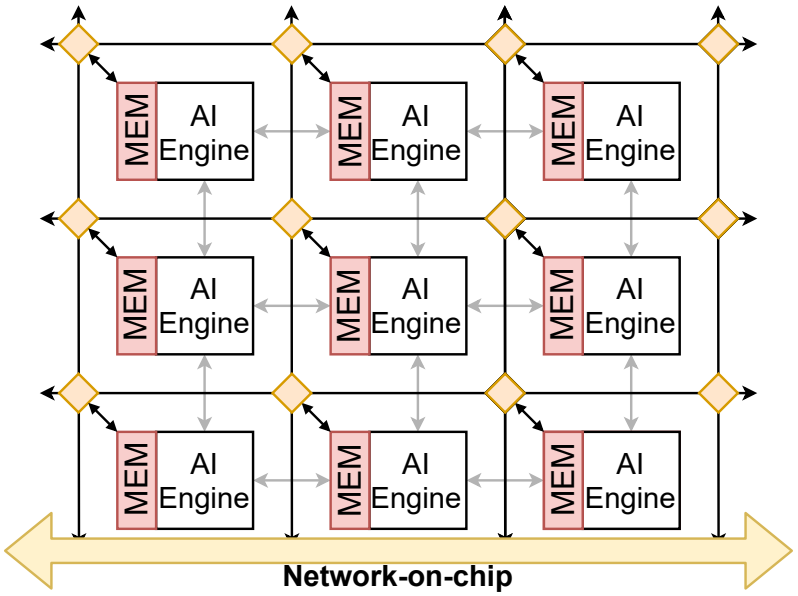


Figure 2.12: AI engine grid with data memory modules. Each AI Engine is connected to adjacent AI Engines via interconnect and to the rest of the grid via NoC.

Each AI Engine tile has its own memory module associated with it, allowing for low latency fetching of data as seen in Fig. 2.13. As the AI Engine tiles sit in a grid like array they have access to the memory modules of AI Engine tiles to the north, south, east and west assuming that the AI Engine tile is not placed at an edge or corner. Communication between tiles that are not neighbors is done via 32-bit stream interfaces which utilises the AXI network that surrounds each tile. The 384-bit cascade interface can quickly move data along a row in the array but cannot send data to other rows, thus it is useful to place AI Engines in either a cluster or a row depending on the size of the implementation.

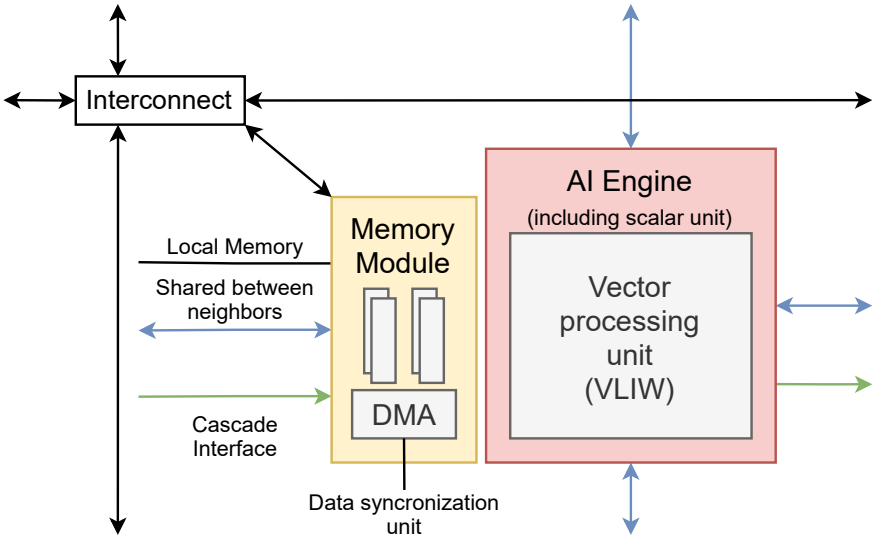


Figure 2.13: Detailed view of AI Engine architecture. Source: [36]

2.3.4 Network on Chip

The Versal ACAP platform contains a programmable NoC which consists of an AXI-4 interconnect within the ACAP architecture. AXI-4 is an interface protocol developed and freely distributed by ARM which allows for high-speed communication between on-chip components. The protocol allows for intercommunication between target and host via *ready signals* that communicate that the component is ready for transmission or reception. The interconnect allows for high-bandwidth data transfer across the platform which minimizes the resource requirements needed for interconnect between components. The Versal ACAP platform relies on NoC communication to provide low latency data streams both to and from IO and for intercommunication between modules. Since certain tasks perform better on certain hardware, operations can be split up without concern for routing.

2.3.5 Development

The Versal ACAP AI core platform allows for development in several different ways. Common real-time applications that are regularly used in signal-processing systems and developed for FPGAs can be created using an assortment of different languages. These languages include HDL (VHDL and Verilog) as well as HLS (C and C++). These languages can also be freely combined between different components, for example writing input and output components in VHDL and writing signal processing components in HLS. This freedom in development allows more complex designs to be hardware accelerated via FPGA architecture. These tasks include neural networks such as CNNs which benefit greatly from the increased parallelism introduced with the AI Engines. CNNs can be developed in a number of different ways on the platform but the simplest way is to import an existing design from a neural network framework such as CAFFE, PyTorch or TensorFlow. The models are converted to instructions for Xilinx's Deep Learning Processing Unit (DPU) via Xilinx pro-

prietary software Vitis-AI. Another way to develop CNNs on the platform is via C/C++ coupled with HLS which can be targeted to make use of specific resources including the scalar, adaptable or intelligent engines. A number of intrinsic functions are available to be used within the AI Engines. The intrinsic functions allow the developer to easily write large vector computations as well as load and store operations, all within one clock cycle.

3

Approach

This chapter explains the process of hardware implementation on the Versal ACAP platform as well as the process of evaluating the performance data of the completed system and comparing it to real world counterparts. The implementation was split into two parts as it became apparent that the DPU implementation would only allow for a limited amount of design decisions to be made. Therefore a custom implementation was developed in C++ that rebuilt the neural network from the ground up to take full advantage of the Versal platform, including the AI-Engine feature.

3.1 Radar input data

The input to the neural network consists of radar data captured at five separate occasions which amounts to approximately 100 000 detections. This data consists of complex in-phase and quadrature (IQ) values in a three-dimensional array and an example can be seen in Fig. 3.1. The pulses refer to the radar wave echo received by the radar array. The range bins are a measurement of how far away the target is, the actual measurements cannot be disclosed.

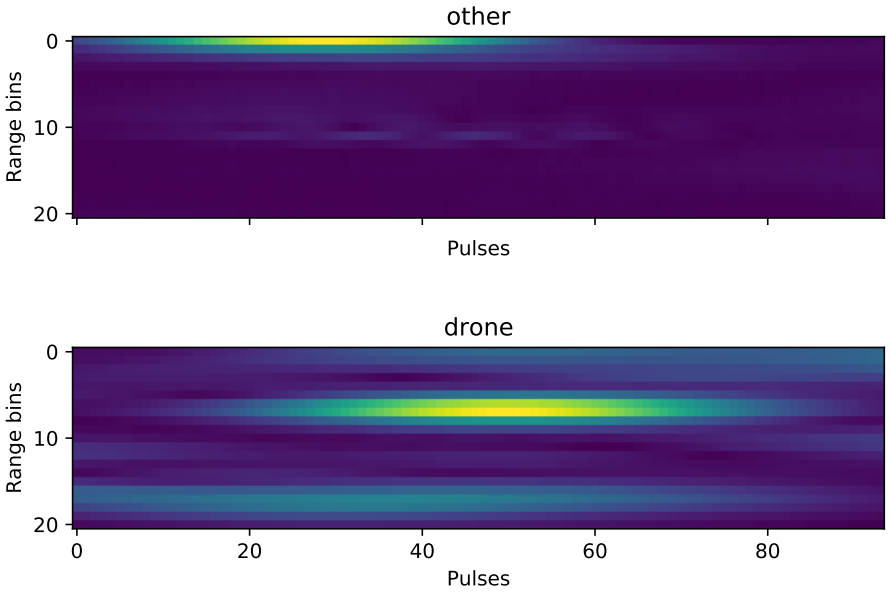


Figure 3.1: Raw radar data before cutout represented by absolute values.

To test the system the radar data has been annotated to show where targets can be found within the array and the type of target has also been specified. This means that there were an estimated 100 000 data points that contain useful information within the captured radar data. A basic overview of the radar data capture can be seen in Fig. 3.2

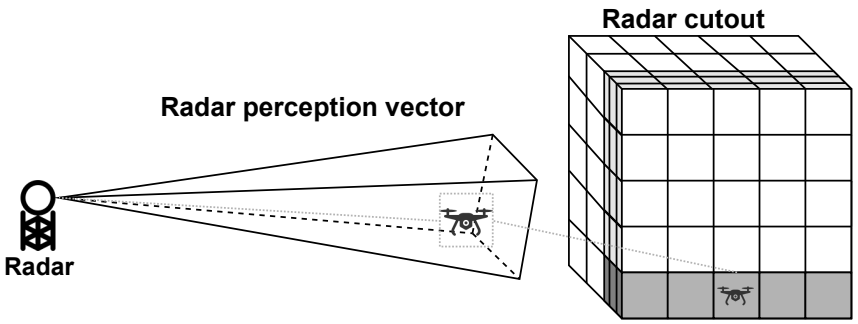


Figure 3.2: Radar cutout from raw radar data.

The sample data was preprocessed before it was sent to the CNN model for training. This pre-processing was done in two steps, first the targets within the sample data were labeled to provide a dataset for the supervised training. The target data was also cut out from the raw radar data sample by taking three depth layers, one height layer and 32 pulses as seen in Fig. 3.3.

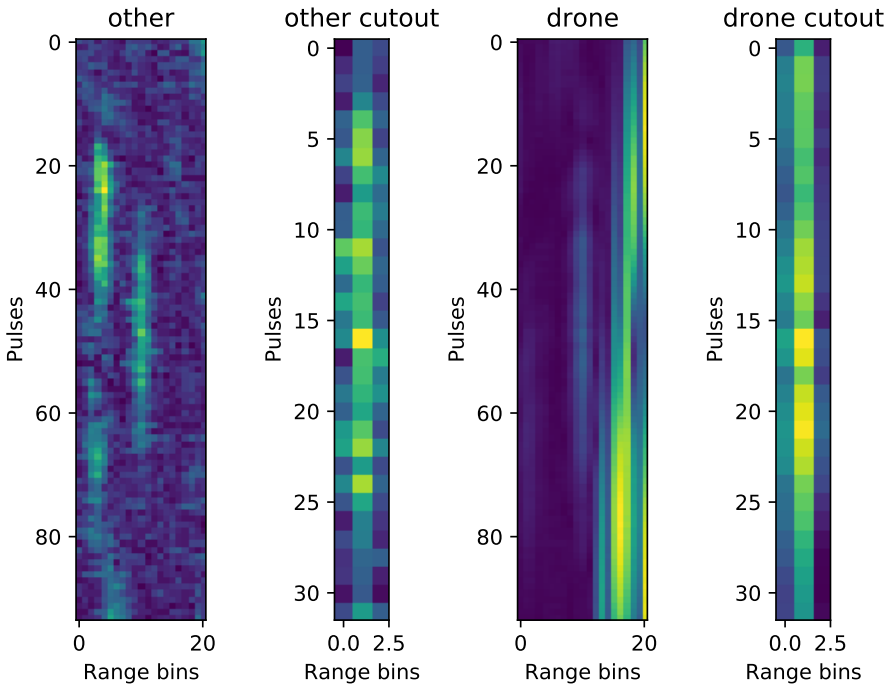


Figure 3.3: Comparison between raw radar sample and cut sample represented as absolute values.

The second step was to split the cut sample data into four separate channels by

taking the logarithmic absolute value, range-doppler map and the real- and imaginary part each as a separate channel as seen in Fig. 3.4. This was done to provide as much information as possible to the model with the limited amount of samples available.

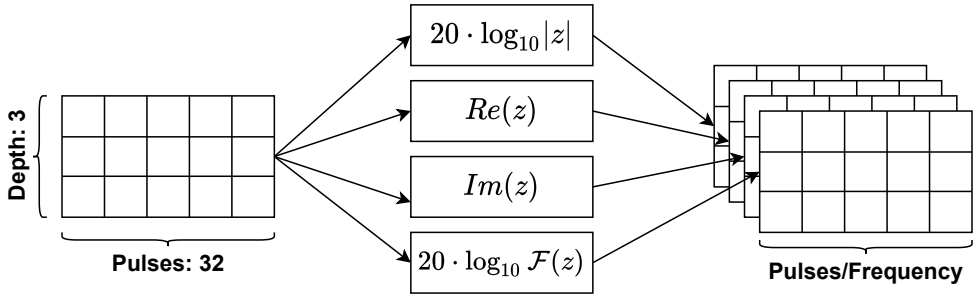


Figure 3.4: Splitting of sample data into four channels.

In Fig. 3.5 the channels for one sample can be seen. Once the sample data had been preprocessed into datasets they were split into four different sets: training, validation, testing and demonstration. The splitting of the datasets was done to provide unique sample data for each task carried out by the NN, for example the training and validation data must be independent from each other to ensure that the NN was not modeled only onto the sample data. This splitting can be done randomly so that each dataset was unique from any previous run or they can be split along predetermined tracks which was useful when comparing how the model performs on different hardware platforms.

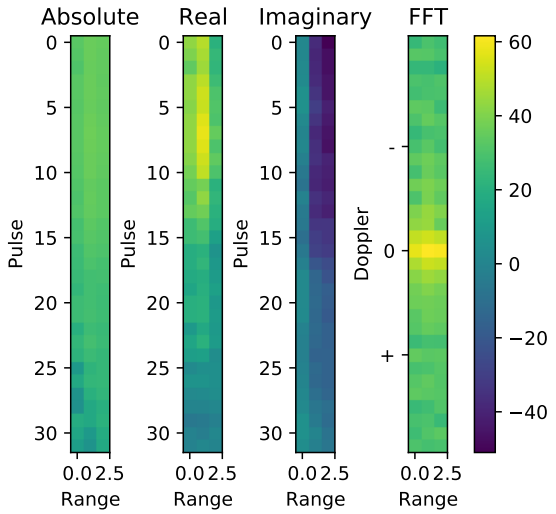


Figure 3.5: Representation of radar data from a drone sample. Plot shows all split channels.

3.2 DPU implementation

The neural network model was implemented using Xilinx proprietary software Vitis-AI which can compile several different neural network models to run on Xilinx DPUs. The model was first built and trained using a TensorFlow model from a previous project provided by Saab. After the model was built and trained it was frozen and quantized before it was compiled into DPU instructions and tested on the hardware platform.

3.2.1 Building and training the TensorFlow model

This project utilized code from a previous Saab project involving classification of radar targets using neural networks. As the focus of the project was on the hardware implementation of the neural network there were no attempts made to optimize the initial TensorFlow models provided by Saab. Certain adjustments to the model had to be made to get it to compile for the VCK190 DPU. The initial model used a kernel size during pooling of (2,1), which was not supported by the compiler. The pooling kernel was adjusted to (2,2) which alters the output shape of the layer from a depth of three in the initial model. The new depth from the output of the pooling layers was two for the first pooling, making the output shape (16,2,16) and one for the second pooling, making the shape (8,1,16). The CNN design consisted of a sequence of layers used to manipulate and interpret the dataset using a TensorFlow model.

In Fig. 3.6 the original CNN model provided by Saab is shown. This model had to be slightly adjusted as it utilised uneven pooling kernels, a feature that is unavailable in Vitis-AI. The adjusted model can be seen in Fig. 3.7. The CNN starts of with an input layer followed by two iterations of convolution, batch normalization and max pooling layers. These layers were designed to decrease the size of the network to make it manageable as a fully connected model. Once the data had been minimized a flattening layer was added which reshaped the layered data into one graph that could be modelled onto a fully connected layer. The fully connected layer consisted of two dense layers each with 16 nodes and rectified linear unit (ReLU) activation functions. The choice of using ReLU came from the previous implementation of the neural network by Saab but as this was the most common and reliable [37] activation method at the time there was no reason to change it.

3.2.2 Freezing model

Once the model was built and trained it was saved using the keras package in TensorFlow. The saved model contains variables for all nodes, weights and biases in the CNN as well as additional nodes utilised for training and loss functions used for optimisation. The saved model has to be frozen in order to be compatible with the Vitis-AI framework. This freezing can be done using the *freeze_graph* function in the Vitis-AI TensorFlow anaconda environment. Freezing the graph means that all variables in the model turn into constants. In addition, training nodes and

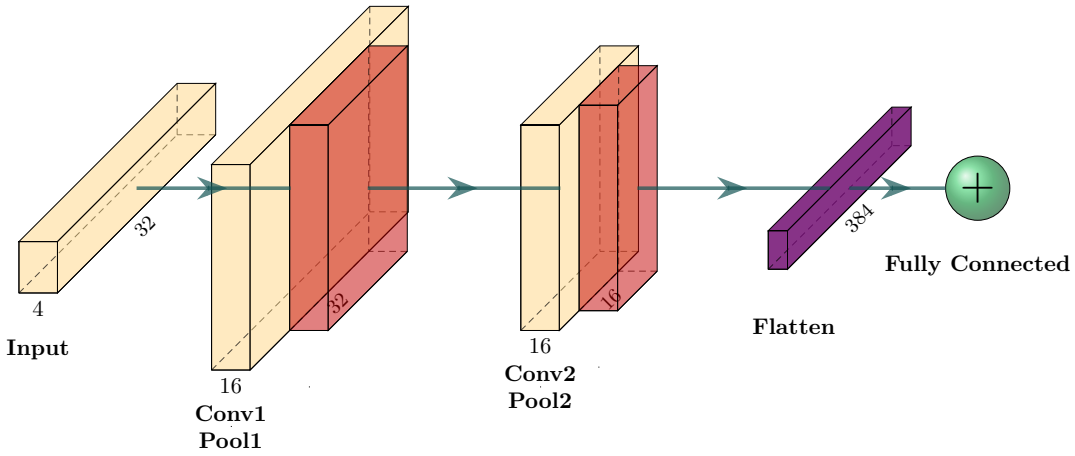


Figure 3.6: Original CNN model implemented in TensorFlow 1.15.

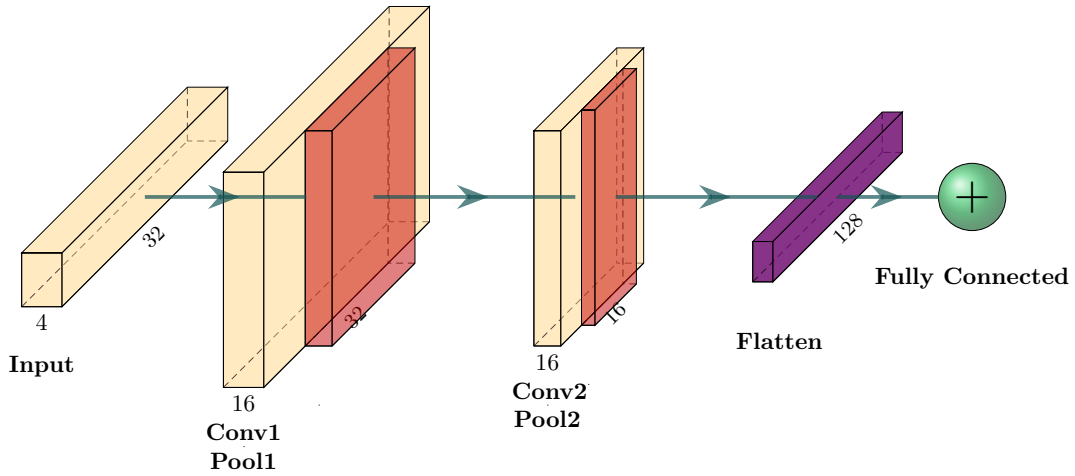


Figure 3.7: Adjusted CNN model utilising even pooling kernels that reduce the width of the pooled output.

optimisation functions are discarded as they were no longer necessary. The output from the freezing function is a single .pb file which can be used for quantization.

3.2.3 Quantising model

The hardware platform accelerator utilised 8-bit integer values for all neural network functions. The network built in TensorFlow 1.15 used 32-bit floating point values. A conversion to fixed-point values was necessary for the model to be compilable for the hardware platform. This conversion is done using Vitis-AI quantisation software. The software takes a frozen TensorFlow model along with a sample dataset and calibrates the weights using cross-layer equalisation (CLE) [38]. Quantisation via CLE is done by adapting the weights of the frozen model for each layer as the quantisation of one layer will impact the weights of a subsequent layer. This relationship between layers can be exploited to derive scaling factors for each layer that allows

for minimal accuracy loss. But this technique can only be used on weights, the error introduced in the bias during quantisation still needs to be taken into account. This error can be calculated during output by comparing the original bias value with the quantised one, the calculated error can then be subtracted from the biased output. To avoid unnecessarily large scaling due to one outlying bias value, high biases are absorbed into subsequent layers. This allows the bias values to be more normalised across the model, increasing the resolution of the 8-bit quantisation.

3.2.4 Compiling and deploying model on Versal platform

For the CNN model to function on the Xilinx hardware platform the model had to be compiled into a file which can be interpreted by the hardware. The Vitis-AI framework included a compiler which allows quantised models to be compiled for hardware deployment. Each DPU has a different architecture which meant that the compilation had to be made with a specific DPU in mind. Once the model was compiled for the VCK190 platform it was ready to be deployed to hardware. The simplest option when it came to transferring the necessary files was to use an SD-card. A bootable image was created on the SD-card which contained not only the compiled model and DPU but also library files and the Petalinux operating system files.

3.3 Custom implementation

The DPU implementation allowed for a simple hardware implementation of the CNN which could be tested and compared to the same neural network on a software implementation. While this implementation was a good outline for the project it did not allow an in-depth look at the hardware platform as the Vitis-AI software took care of all design decisions when recreating the TensorFlow model. A C/C++ model of a neural network was created which allowed for more customisation in the hardware implementation of the network.

3.3.1 System overview

The system was designed to utilise the primary features of the Versal ACAP AI-Core platform as well as to achieve a high level of performance. To allow for a fair comparison between the different implementations the CNN design was identical for the custom implementation. The dimensions and depth of the CNN were kept equal, the only differentiating factor being the type of hardware acceleration used. The system utilised the AI Engines for vector multiplications which were done in the convolution and fully connected layers. For pooling layers programmable logic (PL) was used since these tasks had low compute requirement [39]. A direct memory access (DMA) component was implemented in PL which was tasked with moving data to and from the memory as well as to and from the input and output of the system. Another step needed was the pre-processing of data before it was sent to the AI Engines, this process is described in Section 3.3.4. An overview of the implemented system can be seen in Fig. 3.8.

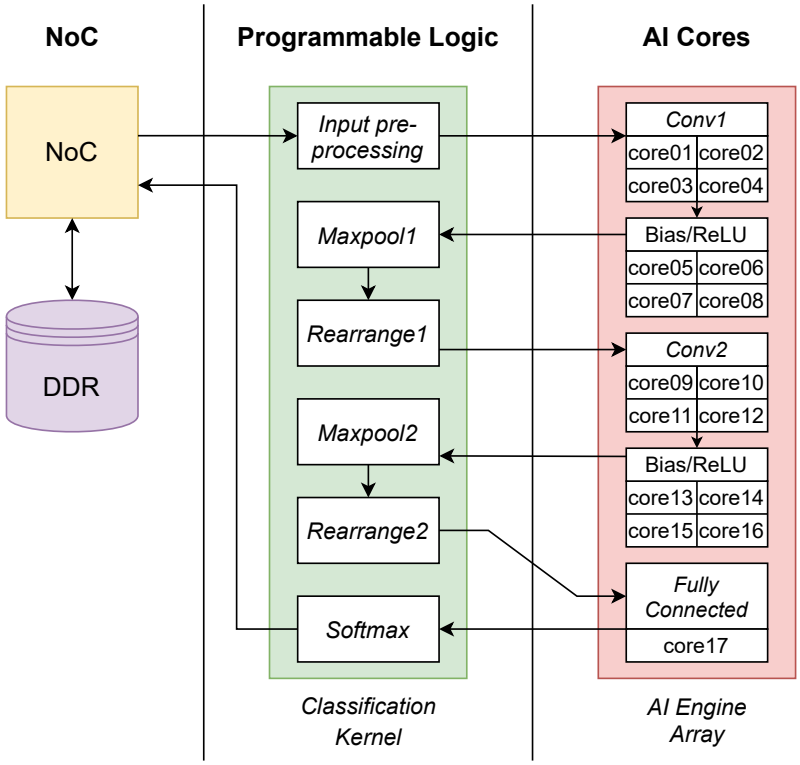


Figure 3.8: Overview of custom system.

3.3.2 Quantisation

To maximize performance as well as minimizing the memory requirements for the system the 32-bit floating point values used for inputs, weights and biases had to be quantised to 8-bit integers. This required the values to be normalised within a range of -128 to 127. Once normalized the values could be safely converted to 8-bit values as they no longer had the potential to overflow the variable. The weights and bias values were taken as quantised values from the TensorFlow model so no runtime adjustments had to be made. The input values were quantised during runtime to provide a fair comparison to the other implementations. The normalisation ratio that was used for the input data was taken from the quantisation calibration described in Section 3.2.3.

3.3.3 Data transfer

Data is transferred between the PL and AI Engines occur several times per sample, requiring low-latency, high-throughput communication. We decided to use AXI4 streams because they were compatible with the AI Engine array's interface. The streams were programmed to send 128-bits per cycle as this was the largest data size supported by the AI Engine libraries. AXI4 memory mapped interfaces were used for DMA transfers to and from the DDR memory which housed the input and output buffers. During runtime, these interfaces were given the address of the buffers as well as the length of the data to transfer. The NoC module was an IP block which could be integrated into the design, configured and then connected to

the relevant modules and hardware blocks.

3.3.4 Pre-processing layer

To take full advantage of the large vector multiplication functions performed by the AI Engine the quantised sample data had to be prepared into a vector format. The convolution layer used a 3x3 kernel with a stride of one to iterate through the quantised sample data. With the smallest 8-bit vector type of the AI Engine being 16 elements, the kernel data was vectorised and padded with seven trailing zeros to this length to allow for efficient memory loads during computation. This process can be seen in Fig. 3.9.

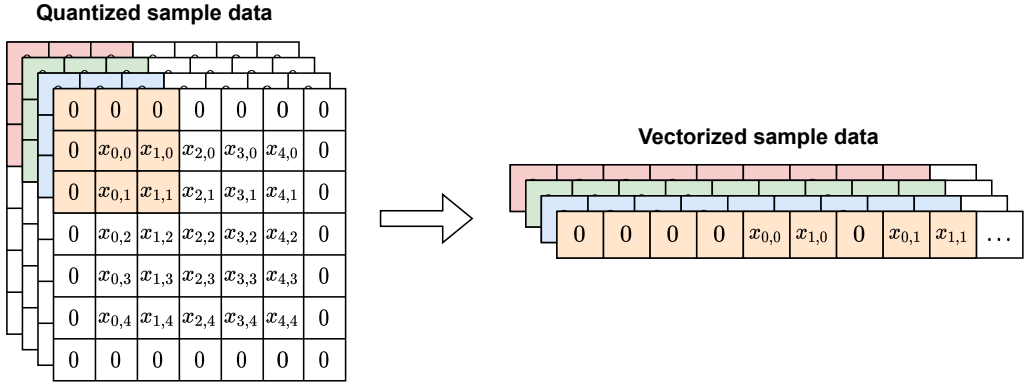


Figure 3.9: Illustration showing vectorisation of sample data. The nine kernel values are padded with seven trailing zeros.

In Fig. 3.10 a block diagram of the pre-processing layer can be seen. Double buffering was used so that one sample could be vectorised while the next sample was being quantised and stored. Depending on the performance and resource usage goals the convolution could be split onto several AI Engines. This affected the pre-processing as the samples could be split up into several sub-samples which each could be zero-padded and processed into vectors in parallel.

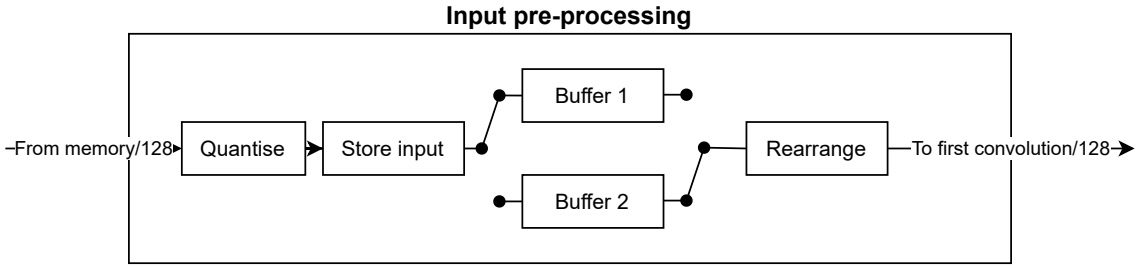


Figure 3.10: Block diagram of pre-processing layer

3.3.5 Pooling and rearrange layers

The pooling and rearrange layers of the CNN were implemented in PL as these layers would not gain any significant benefit from the intrinsic functions available in the AI

Engine. Data received from the convolutional layer was stored in a three dimensional vector which contains the pulses and depth for each kernel. The convolutional data received from layer one has the dimensions 32x3x16, resulting in 1536 8-bit values that need to be stored. The shape of the convolutional data must be retained as each value relates to the value next to it. A block diagram of the maxpool layer can be seen in Fig. 3.11.

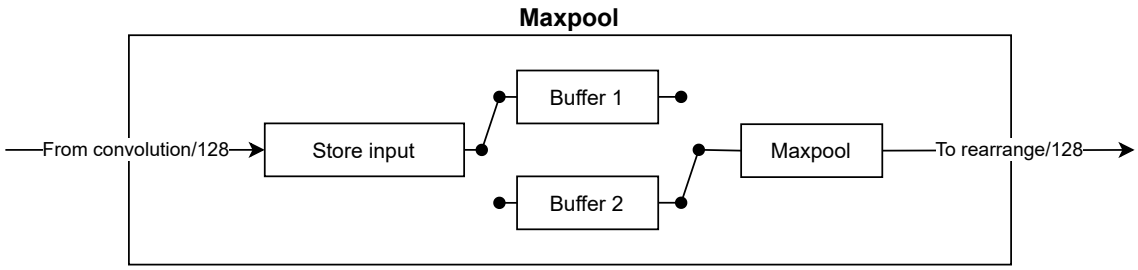


Figure 3.11: Block diagram of maxpool layer

The accumulated data was then iterated upon and pooling was done in a 2x2 matrix, taking the maximum value and outputting it to an activation map. Each kernel generates one activation map and the output dimensions of each map was 16x2. If a 2x1 matrix were used instead then the output shape would have been 16x3 as the matrix only iterates one step in the y-axis. The choice to use a 2x2 matrix was done in accordance with the initial TensorFlow model provided by Saab. The calculated activation maps was then sent to the rearrange layer where the data was prepared for the second convolution layer. A block diagram of the rearrange layer can be seen in Fig. 3.12, note the similarities to the pre-processing layer. The rearrange layer utilises a slightly modified version of the pre-preprocessing module to prepare samples for the second convolution.

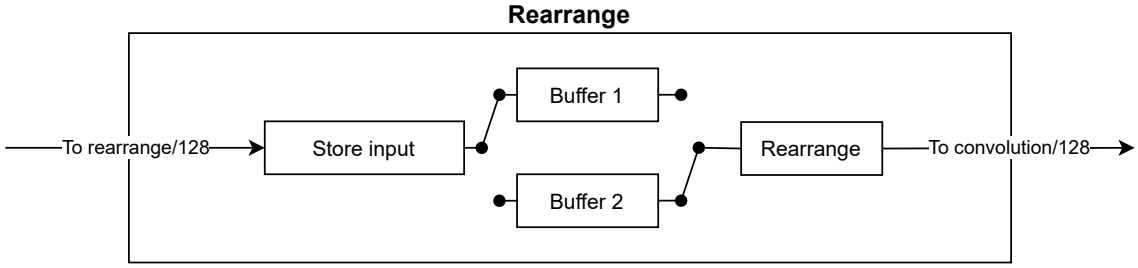


Figure 3.12: Block diagram of rearrange layer

3.3.6 Convolutional layers

The main function of the AI Engines was to implement the convolutional and fully connected layers as the AI Engines excel at large vector arithmetics. These vector operations were done using intrinsic functions created specifically for the AI Engine. By adapting the code to use intrinsic functions the performance could be greatly increased as the very long instruction word (VLIW) processors allow for large vector

arithmetics within one clock cycle. This gives a great performance boost compared to one scalar operation per clock cycle. The weights and biases used in the custom implementation were extracted from the quantised TensorFlow model as 8-bit integer values. The data memory present in each AI Engine was used to store the weights and biases as arrays.

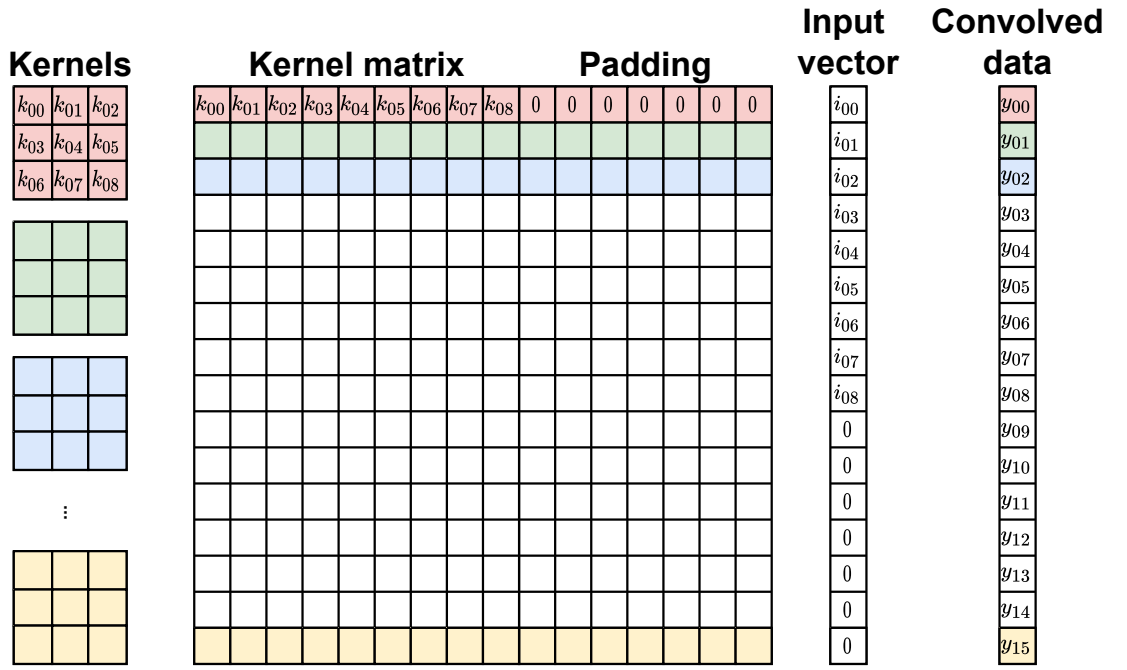


Figure 3.13: Convolution between kernels and input in AI Engines.

As seen in Fig. 3.8 there were 16 AI Engines used for the two convolution layers. The implementation of the two layers were very similar as both utilised 16 kernels but they differ in that they operated on different input dimensions. The first convolution layer used the input dimensions 32x3x4 which was convolved using 16 kernels with individual weights for each channel. These convolutions were performed using intrinsic vector instructions, which made it possible to perform a $[16 \times 8] \times [8 \times 1]$ matrix multiplication in a single clock cycle. An example of how these multiplications were done can be seen in Fig. 3.13.

```
INLINE_DECL void core01(
    input_window_int8 *Sample_in,
    output_stream_acc48 *C_out)
```

Figure 3.14: Convolutional layer function declaration.

The function declaration in Fig. 3.14 shows two arguments used by the convolutional layer. The input to the function is a window, which is a buffer that is filled with input values. The output is a cascade stream of accumulator values containing the 32x3x16 activation map of the convolution. This is sent to the next AI Engine, which performs bias addition and the ReLU activation function.

```
v16int8* restrict pRow  = (v16int8*) Sample_in->ptr;
v16int8* restrict pRow2 = pRow + N_CHANNELS;
v16int8* restrict pRow3 = pRow + N_CHANNELS*2;
v16int8* restrict pRow4 = pRow + N_CHANNELS*3;
v16int8* restrict pWeights = (v16int8*) Weights;
```

Figure 3.15: Pointers used in convolutional layer

The input sample window and weights were stored in the data memory of the AI Engine and loaded using pointer dereferencing as shown in Fig. 3.15. To minimize the number of times the weights had to be loaded, four sub-samples were calculated at the same time. The `restrict` keyword allows for more aggressive optimisations by the compiler by stating that the data the pointers are pointing to are independent of each other.

```
acc = mul16(Xbuff,0,0x33323130,32,0x3120,
            Zbuff,0,0x00000000,2,0x1010);
acc = mac16(acc,Xbuff,0,0x33323130,32,0x3120,
                Zbuff,0,0x00000000,2,0x1010);
```

Figure 3.16: Intrinsic vector multiplication functions.

AI Engines provide high-performance vector multiplications using intrinsic functions, examples of which are shown in Fig. 3.16. The specific functions used in this implementation were `mul16` and `mac16`. These functions can perform 128 multiply-accumulate (MAC) operations using 8-bit input vectors. The configuration of these operations depend on the length of the input vectors. If `Xbuff` in Fig. 3.16 is a 128-element vector and `Zbuff` is a 32-element vector, the operations are arranged in 16 rows and 8 columns and the sum of each row is the output. Both `mul16` and `mac16` perform the same operation except that `mac16` accumulates the result with the values in the given accumulator register. The first five arguments of the functions, excluding the accumulation argument for the `mac16` function, refer to the first vector. They specify which vector to use, the starting index in the vector, the offsets for the rows, index step size between columns and finally the square value used for fine-grain data permutation. Using the square value, we could specify in which order the elements in the vector were placed as can be seen in Fig. 3.17. Subsequent five arguments refer to the second vector used in the multiplication. The `mul16` function was applied to the first input of each sub-sample and subsequent inputs utilised the `mac16` function as all channels were added into the activation maps.

The matrix multiplication calculates one value for each of the 16 kernels. However, with the intrinsic function limited to 8 columns only 8 of the 16 input values for each channel could be part of the computation, requiring two function calls per channel. When all channels had been processed, the 16 output values in the accumulator were sent to the next AI Engine using a cascade stream. This AI Engine added the bias to the accumulated values and the ReLU activation function was applied, which sets negative values to zero. The ReLU function was applied using the intrinsic function

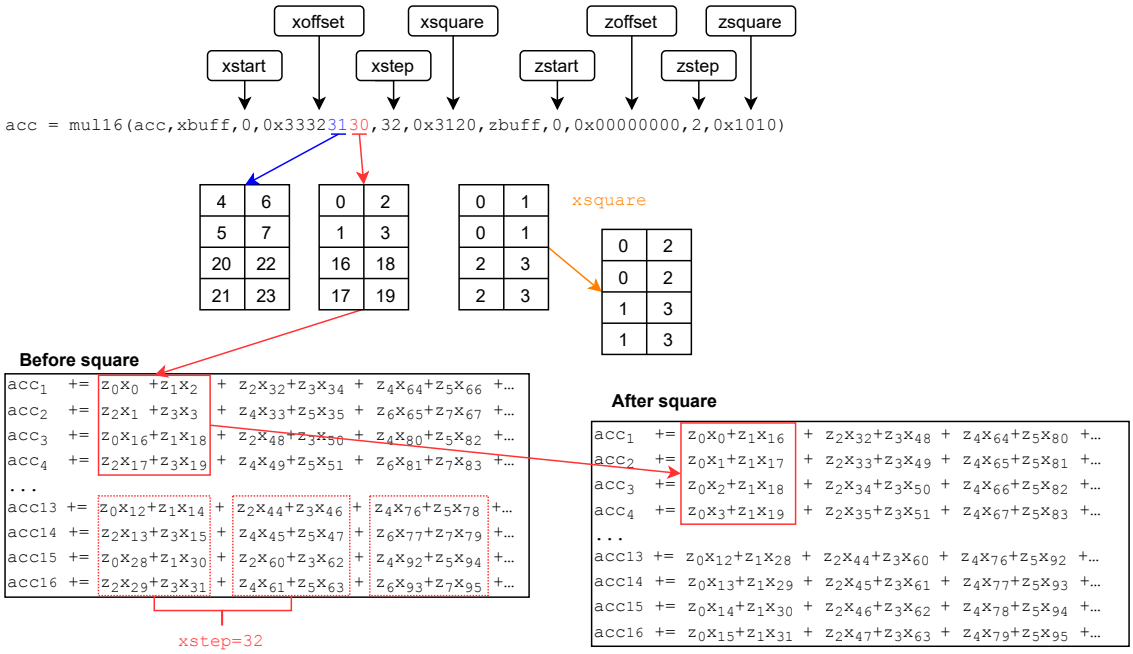


Figure 3.17: Selection of vector data via the square argument. Adapted from [40].

`maxdiff32`, which takes the maximum of 0 and the difference between the two input vectors, where the `null_v32int16()` intrinsic returns a 32-element vector of zeros. Two output vectors were calculated in parallel, resulting in the 32-element vector seen in Fig. 3.18. The output of the ReLu function was then stored in the output window, transferring the convolved data to the maxpool layer.

```
v32int16 vPreReLU = add32(val,bias);
v32int16 result = maxdiff32(vPreReLU,null_v32int16());
```

Figure 3.18: Addition of bias and application of ReLU.

3.3.7 Fully connected layers

The fully connected layers, also known as dense layers, were implemented using a single AI Engine as described in Fig. 3.8. Similarly to the convolutional layer described in Section 3.3.6 the dense layer utilised intrinsic vector functions for acceleration of vector multiplications. The input to the first dense layer was a 128-element 8-bit vector which contained the output of the maxpool2 layer. These values have already passed trough ReLU and as such the vector contained only positive values which meant that the first dense layer required no activation function. In Fig. 3.19 we can see the structure of the dense layers including the shape of the input and output.

The dense layer comprised artificial neurons, the number of neurons was specified by the output shape and each one was connected to all inputs. Each input to the neuron was weighted, which meant that for dense layer 1 there were: $128 \cdot 16 = 2048$ individual weights. Along with each neuron a bias value was added which was

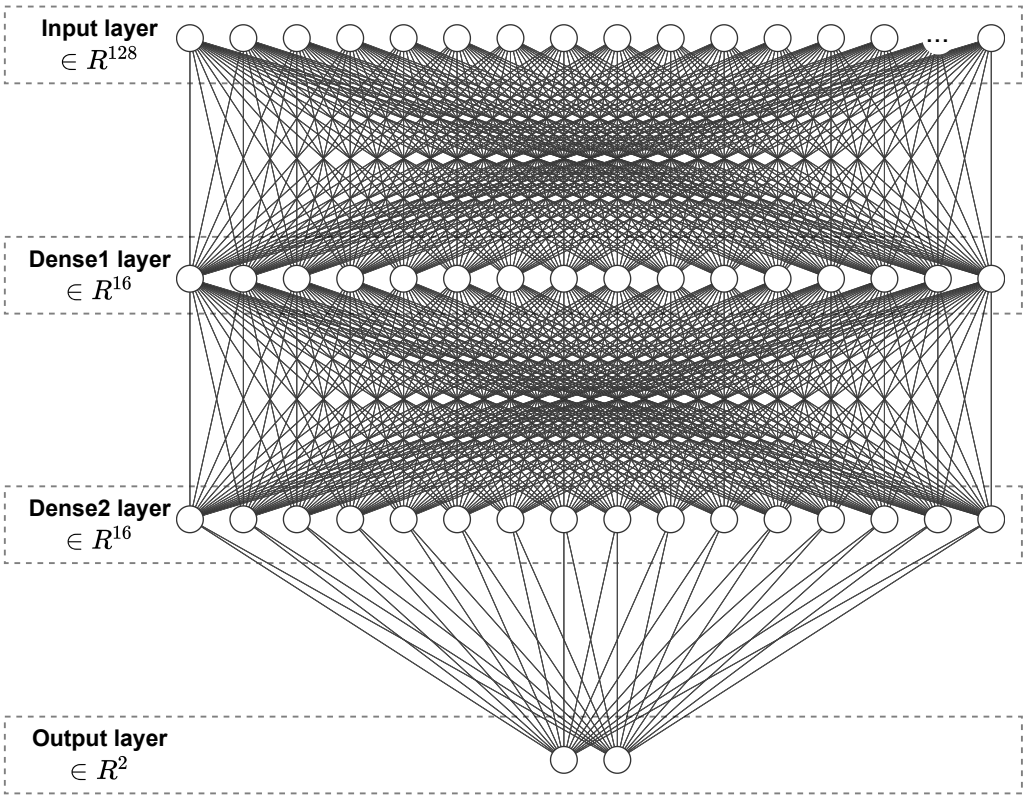


Figure 3.19: Fully connected layer consisting of two dense layers, an input layer and an output layer.

derived from supervised training. The formulas used to calculate the neuron output can be seen in (2.1) and (2.2). The multiplication and summation of inputs used the same *mul16* and *mac16* functions as the convolutional layer.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{3.1}$$

The activation functions for the dense2 and output layers differ. The dense2 layer applied the regular ReLU function, which was used in the convolutional part of the network. For the output layer a softmax activation function was used, the function can be seen in (3.1). Softmax was used to calculate the probability between the two output types, drone or other. The output with the highest statistical likelihood was chosen as the prediction for the sample.

3.4 Evaluation of performance

The final evaluation of the system was performed with the full system implementation running on the evaluation board. The performance metrics that were compared to the CPU baseline implementation consisted of: latency, throughput and resource allocation. The reason for choosing a CPU as baseline instead of running the TensorFlow model on a GPU was due to the size of the neural network. The time taken to transfer data to the GPU, process it, and sending it back creates more overhead

than time gained by the higher performance. This made a CPU baseline more favorable for the TensorFlow model which is in line with the findings that Saab had during their previous project.

The development of the system followed the design flow recommended by Xilinx [41]. The components were developed and tested individually before being integrated into the system and tested together. Testing of the system was done via simulator and test-benches in the Vitis platform. This was done to speed up development but final testing was done on the hardware itself. The system was implemented using C/C++, making use of high-level synthesis (HLS) which was helpful in avoiding some of the time sinks present in VHDL coding such as pipelines.

4
Results

This chapter intends to present the results of the project. These include the pre-processing of radar data, modifications of model to fit the hardware requirements and a summary of the achieved performance.

4.1 Developing and training the model

The provided TensorFlow implementation seen in Fig. 3.6 was used as a baseline for performance comparisons. The model had to be slightly adapted to be compiled and deployed to the VCK190 board as can be seen in Fig. 4.1. This meant that the previous results provided by Saab were not representative of the final model and new evaluations had to be made. These evaluations are an integral part of training a new CNN model as they determine if each training epoch has had a positive or negative effect on the overall accuracy of the system.

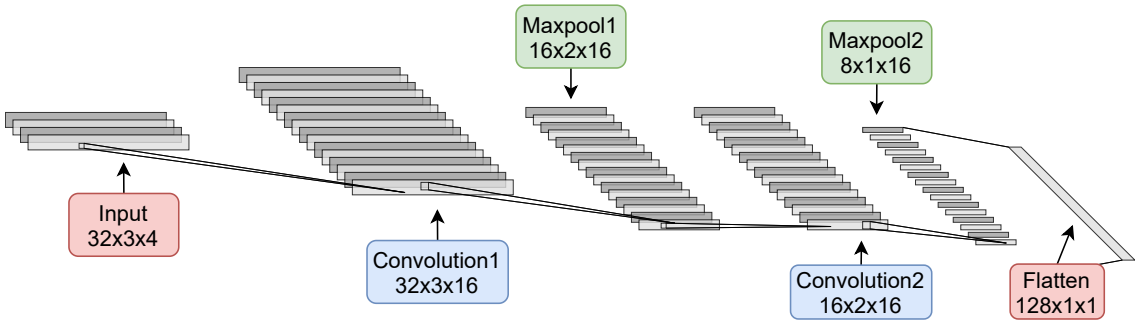


Figure 4.1: Alternative view of adapted CNN model shown in Figure 3.7.

The adapted model was trained for 50 epochs using a specified training set that was balanced in that it contained an equal number of positive and negative outcomes. As can be seen in Fig. 4.2 the model only improves slightly after 30 epochs, this makes 50 epochs a safe assumption that the model is fully trained. Once a training epoch was completed a validation set was run that contained separate data to that of the training set. The weights and biases of the model were updated if the accuracy in the validation set improved compared to previous epochs.

The prediction score for the model can be seen in Fig. 4.3. These results were captured using a CPU and show the effects of quantization on the overall accuracy of the system. Small differences in accuracy can be seen between the platforms due to platform architecture and rounding errors but the quantized model is a

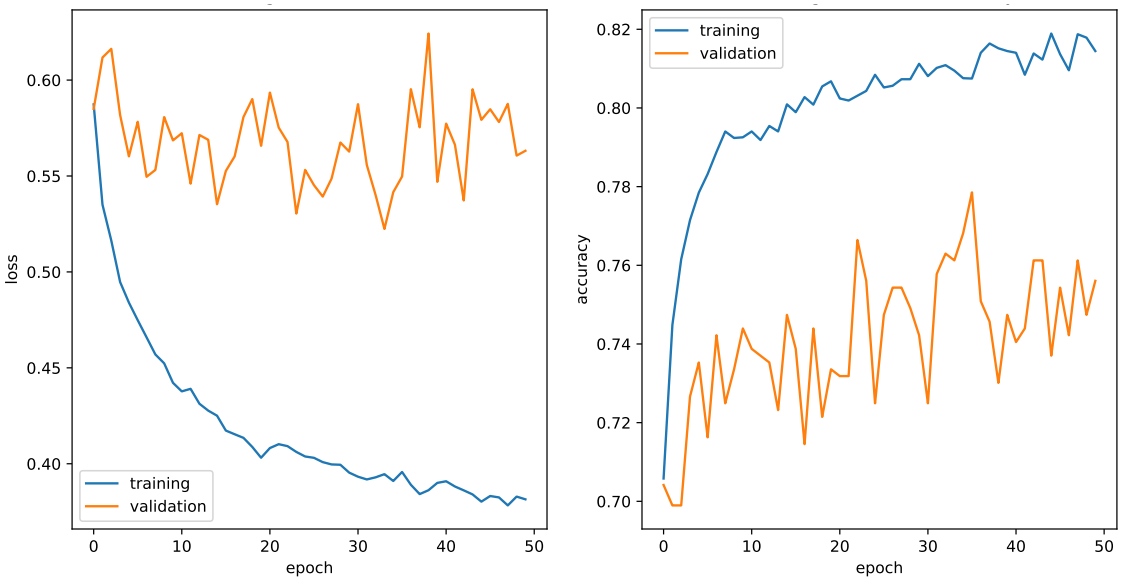


Figure 4.2: Left plot shows loss values for the model during training, right plots shows accuracy.

representation of the accuracy achieved on the VCK190. The confusion matrix allows us to see the correct and incorrect predictions as described in Section 2.2.13. The dataset used for testing, which contained 10 092 samples, was separate from the ones used during training and validation. By splitting up the datasets we could minimise the likelihood of overfitting. The size of the training dataset was around 100 000 samples and the validation set contained around 30 000 samples. The same datasets was used for all implementations to get a fair comparison between different approaches.

Frozen model

Total 10092		Predicted: True	Predicted: False	F1 score 73%
Actual	True	True Positive 1365	False Negative 220	Recall 86%
Actual	False	False Positive 808	True Negative 7699	Specificity 91%
F1 score 94%		Positive Precision 63%	Negative Precision 97%	Accuracy 90%

Quantized model

Total 10092		Predicted: True	Predicted: False	F1 score 51%
Actual	True	True Positive 610	False Negative 975	Recall 38%
Actual	False	False Positive 194	True Negative 8313	Specificity 98%
F1 score 93%		Positive Precision 76%	Negative Precision 90%	Accuracy 88%

Figure 4.3: Prediction score of frozen and quantised model.

4.2 Performance comparison

Once developed the model was trained and tested using the TensorFlow framework running on an Intel Core i7-9850H CPU. The clock frequency used for the PL was 300 MHz while the AI Engine array was clocked at 1.2 GHz. The first implementation, which was developed using Vitis-AI, utilised the created TensorFlow model file to compile the model for the DPU. The second implementation used extracted weights from the TensorFlow model in the convolution and fully connected layers. The results from each layer were compared to the corresponding outputs from the TensorFlow model to assure that the two models had the same behaviour.

Table 4.1: Description of performance metrics used.

Metric	Measurement	Description
Throughput	Samples/s	Number of samples classified per second
Latency	Milliseconds	Time from input to output for one sample
Execution time	Milliseconds	Time to execute entire dataset

The metrics described in Table 4.1 were used to rate the performance of the implementations in comparison with the TensorFlow model. All results were gathered by taking the average of 1000 runs on the unbalanced test dataset containing 10 092 samples.

Table 4.2: Results from all implementations, measurements found in Table 4.1.

Implementation	Tensorflow	DPU 1 thread	DPU 2 threads	Custom
Troughput (S/s)	124 838	15 102	20 400	2 446 520
Latency (ms)	14.5	0.468	N/A	0.042
Execution time (ms)	80.8	640	588	4.125

Performance was measured by logging the time taken to complete one run of the unbalanced test dataset. Timing was only measured during the computational part of the execution, removing the overhead caused by loading data from memory. Results measured in Table 4.2 shows throughput measured on the entries test dataset. The latency for the implementations was measured by sending one single sample through the model. Latency could not be measured for the 2-threaded DPU as a single sample could not be split over two threads. The resource usage for the two implementations running on the VCK190 can be found in Table 4.3.

Table 4.3: Resource usage of hardware implementations.

Implementation	DPU (Both)	Usage (%)	Custom	Usage (%)	Available
LUTs	225 253	25.03	22 036	2.45	899 840
FFs	283 103	15.73	32 896	1.86	1 766 784
BRAMs	33	3.41	84	8.68	967
DSPs	200	10.16	29	1.47	1968
AI Engines	96	24	18	4.5	400

5
Discussion

This chapter discusses some of the different aspects of the project as well as the results presented in chapter 4.

5.1 Evaluation of the VCK190 platform

The primary focus of this project was to evaluate the potential benefits and drawbacks of the VCK190 platform, both in terms of performance and functionality. The intention of using two separate implementations to evaluate the platform was to highlight different aspects of the development process.

5.1.1 DPU implementation

The DPU implementation shows a decrease in performance compared to the TensorFlow implementation which might be attributed to the small size of the model. As the DPU implementation utilises a larger amount of resources it might waste performance by trying to fit such a small model. The amount of time required to create a functional DPU implementation is quite low, due to the fact that the tools can compile an implementation from a TensorFlow model. There are drawbacks to this method as not all TensorFlow layers are supported by the Vitis-AI tools, which could require small modifications to the model.

5.1.2 Custom implementation

The custom system provides an immense performance boost compared to the other approaches as seen in Fig. 5.1. This performance is achieved by efficiently pipelining data throughout the implementation. This approach heavily relies on Vitis HLS to synthesize C++ code but most parts of the implementation could be written in VHDL if more precise management of clock cycles was necessary. The AI Engines however can only be programmed using C++ and vector instructions are given using intrinsic functions. These intrinsic functions use fixed data widths, which means that if the provided data does not fit within these dimensions it either needs to be split up or padded. Limiting the amount of padding done by efficiently handling the data can provide big performance benefits. Similarly the amount of AI Engines used can be increased by splitting up sample data, due to the small sample sizes used during this project it was deemed unnecessary to use more than four cores per convolution layer.

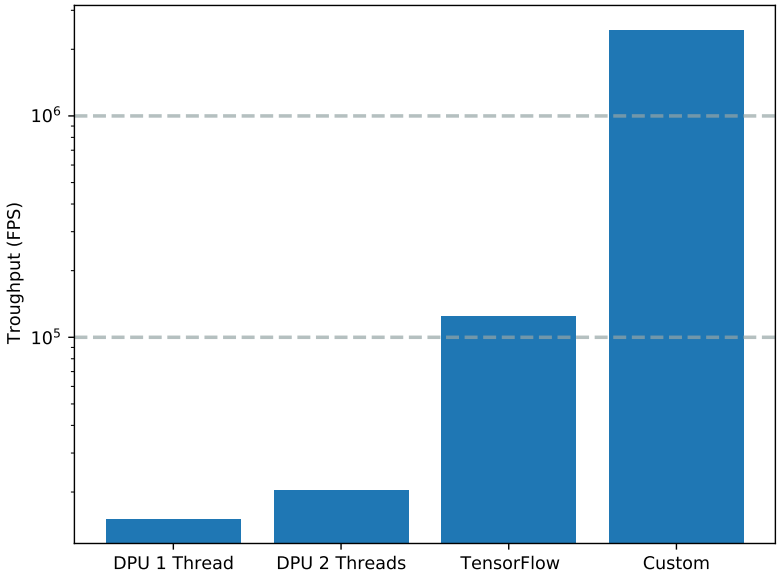


Figure 5.1: Bargraph of throughput for all implementations.

5.2 Loss of accuracy due to quantisation

While the performance of the custom implementation is many times better than that of the baseline TensorFlow implementation it also has reduced accuracy of the model due to quantisation of the input data and weights. As the quantisation is an important step to extract maximum performance from the platform it would be beneficial to explore possible ways to improve the accuracy without removing the quantisation. Since the difference between the maximum and minimum value in the sample set is used to normalise the sample before quantisation it might be beneficial to decrease this difference for each sample, increasing the resolution of the 8-bit value. Another way to increase resolution of the 8-bit quantised value is by looking at the channelised data. Is there one channel that is reducing performance? Would it be beneficial to remove a channel to reduce the spread of values? It might also be useful to look at different methods for processing the complex IQ data. Maybe there is a signal transformation that would be more suitable for neural networks?

5.3 Hardware acceleration of CNN tasks

Convolutional neural networks are inherently compute heavy tasks with multiply accumulate (MAC) operations accounting for more than 99% of all operations run [42]. These types of operations benefit from parallelisation to minimise downtime associated with load and store operations. By applying hardware acceleration to run MAC tasks in parallel it is possible to increase the performance of the CNN. This application is becoming more commonplace with manufacturers developing hardware intended specifically for neural network tasks, such as the Google Tensor processing units [43]. These high-performance platforms are mostly aimed at cloud computing applications in data centers. Platforms developed for edge applications have to com-

promise performance to allow for lower power consumption and minimal hardware requirements. The VCK190 platform tries to bridge this gap by providing high-performance computing for neural network tasks as an edge device. The platform provides opportunities for tasks such as real-time classification of radar targets as these systems are highly mobile and have strict power consumption requirements. These perks accrue a higher purchasing price compared to other AI edge devices due to the complexity of the hardware. Another limiting factor is the development time and hardware knowledge required to develop a high-performance architecture for the system.

5.4 Comparing CNN model between platforms

Comparing a CNN model between different hardware platforms or implementations presents certain problems. Some kernel sizes are not supported by other frameworks and certain layers might not behave the same way in different platforms. These types of problems requires research into how a specified framework builds the CNN model. When creating the custom implementation these issues were taken into consideration as the created implementation needed to mimic the provided model as closely as possible. However, the accuracy and precision of the different implementations will be slightly skewed compared to the TensorFlow model. This skew is due to several factors which includes rounding errors and how the compiler interprets the code that is compiled into hardware instructions.

5.5 Possibility of real-world use

The developed implementations as well as the original TensorFlow model suffer from low accuracy. This is partly due to the limited amount of radar data that was available to train the network. The accuracy could also be affected by the chosen CNN model and other models might provide better results. A reliable CNN should preferably be able to accurately classify targets within a certain precision range. This range depends on the type of application area that the network is deployed in [44]. One option to evaluate a CNN is by meeting the baseline requirements of MLPerf [45], an open-source project aimed at evaluating the performance of cloud-based neural networks. This baseline requirement states that an accuracy of 76.46% must be measured using the Resnet50 [46] model on the ImageNet [47] dataset. While these circumstances are not applicable to the project at hand it is an interesting baseline to consider when evaluating the accuracy of the model. A CNN that identifies hundreds of thousands of targets per second demands a higher accuracy to avoid a large number of incorrect conclusions.

The outcome of the classification also plays a part in the amount of accuracy needed [48]. A CNN could be deployed to help sort unripe produce from a conveyor belt. This type of task have no immediate risk involved if an incorrect classification is made. Applying a CNN in a radar application carries a higher risk as an incorrect classification can lead to severe consequences. Finding an acceptable accuracy value for radar applications is a subject of its own but it is likely that a value above 95% is

achievable with enough data and a more optimized network. In its current state it is unlikely that any implementation could see real-world use due to the low reliability associated with the accuracy readings.

5.6 VCK 190 development environment

The development process for the Versal series of ACAPs has different environments depending on the type of implementation that needs to be done. The two environments both utilise Vitis to compile code into instructions for the platform but differ in the programming language used. This difference also extends to the underlying knowledge needed to work with the platform.

5.6.1 DPU implementation

The DPU implementation allows developers to transfer a pre-existing neural network written in one of the most popular frameworks to the Xilinx hardware. This transfer is highly useful in quickly gauging the performance of the platform and the potential benefits of it without prerequisite knowledge of the hardware. It is however important to remember that the created neural network will be limited by the DPU it is compiled for. A basic platform layout can be sourced from Xilinx which allows developers without insight into embedded systems to still develop on the platform. But a basic platform might not always provide the right solution and adaptations might have to be made for the platform to function according to specifications.

5.6.2 Custom implementation

The custom implementation allows for more freedom in development as components can be written in both C++ and VHDL. However this freedom means that the developer needs to have more in-depth knowledge about the hardware that they are developing for. Connections between components need to be specified and need to adhere to the specifications of the component. These specifications could include read and write speeds for interface ports, the datawidth possible during transfer and the clocking of specific components on the platform. Documentation for some of these specifications are hard to come by and often intricate in nature. This makes development using the custom framework more time-consuming but ultimately results in higher performance.

6
Conclusion

The main benefits of the VCK190 platform when performing neural network tasks is increased performance and smaller batch sizes compared to a CPU implementation. The performance is increased in a number of different ways. Throughput is increased by utilising VLIW processing units for vector multiplications and by pipelining and buffering data efficiently. The latency is reduced by utilising the interconnects between memory, programmable logic and AI Engines. In addition the 7nm architecture provides lower power consumption thus reducing the heat emission from the device. This lower heat emission proves highly beneficial in a military aspect as the device will have a smaller thermal signature, which means that it will be harder to spot the radar array with infrared sensors. The small scale of this project means that only a fraction of the resources available on the platform are utilised. It would be advisable to increase the number of tasks running on the device, for example by introducing radar pre-processing or beamforming. A smaller version of the device could also be acquired which should lower the power consumption of the platform.

6.1 Future research

Due to the limitations set early on in the project some aspects could not be explored but warrant further research. Here are some of the suggestions for future work that would greatly benefit the project.

6.1.1 Implementing training and backpropagation

As the created custom implementation attempted to mimic the original design as closely as possible it was natural that it used the same weights. This is however a limitation as training the model and exporting the weights to the custom platform is both time consuming and inefficient as training via CPU is sub-optimal. Supervised training of the model could be accomplished by introducing backpropagation [49],[50] and dynamically updating the weights during training. This training becomes more necessary if the model is adjusted to fit higher resolution data as mentioned in section 6.1.2.

6.1.2 Larger datasets

The dataset provided by Saab comprises approximately 100 000 samples. In comparison, the popular dataset ImageNet [47], which is used as a standard for verifying

functionality in CNNs, contains 1 281 167 training samples, 50 000 validation samples and 100 000 test samples. Increasing the amount of training data available should theoretically increase the accuracy of the model, making it more viable in a radar classification scenario. The quality of the data provided is also a factor that should be further investigated. Could accuracy and training performance be improved with higher-resolution data? Modifying the CNN to allow for processing of raw radar data instead of pre-processed radar cutouts is a natural step for the project to take in the future. The project utilises radar cutouts for samples that measure 32 pulses and 3 range bins. The raw radar data available is much larger than this, each sample containing up to 21 range bins and up to 134 pulses. The cutting of data is always made from the center of the sample, meaning that sometimes vital data used to identify a target might be cut out. This cutting was done to be able to run the system on a CPU which is no longer a concern as the VCK190 can handle larger sample sizes.

6.1.3 New network model

One alternative to increase accuracy is to use a new neural network model that takes full advantage of the considerable resources available on the VCK190 platform. The current custom implementation utilises 18 AI Engines out of the 400 available on the platform. This utilisation means that the neural network could easily be expanded without any substantial degradation in throughput. A different CNN model that could be investigated is VGGNet [51], which has a very deep neural network structure. Whether the input data is detailed enough to warrant this type of deep processing is currently unknown. Another alternative CNN is ResNet [46], which utilises deep residual learning in an 18 or 34 layer architecture. A common theme with these CNN models is that they are significantly deeper than the implementations used in this project. This difference in size is due to the dimensions of the input data, which in the case for this project are much smaller than other CNNs. Thus research into CNNs that employ similar dimensions might be the best course of action to increase accuracy and performance of the model.

6.2 Ethical considerations

This project is done in cooperation with Saab which means it is in nature defense related. As stated in chapter 7 article 51 of the United Nations charter [52] it is every nations right and duty to protect its citizens from armed attack. The implemented system is intended for classification of radar targets which is an essential tool in protecting both civilians and military personnel. As such any failure of the system could have steep consequences including damage to vital infrastructure and potential loss of life. These consequences are something that have to be taken into account when testing and evaluating the hardware implementation.

Bibliography

[1] J. Bosse, O. Krasnov, and A. Yarovoy, "Direct target localization with an active radar network," *Signal Processing*, vol. 125, pp. 21–35, 2016. DOI: 10.1016/j.sigpro.2016.01.001.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007. DOI: 10.1111/j.1467-8659.2007.01012.x.

[3] "2020 Military and Aerospace Technology Innovators Awards announced for aerospace and defense achievement," *Military and Aerospace Electronics*, vol. 31, no. 9, pp. 4–14, 2020.

[4] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A Survey of FPGA-Based Neural Network Inference Accelerator," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 9, no. 4, 2017. DOI: 10.1145/3289185.

[5] S. Ahmad, S. Subramanian, V. Boppana, S. Lakka, F.-H. Ho, T. Knopp, J. Noguera, G. Singh, and R. Wittig, "Xilinx First 7nm Device: Versal AI Core (VC1902)," no. 2019 IEEE Hot Chips 31 Symposium, HCS 2019, 2019. DOI: 10.1109/HOTCHIPS.2019.8875639.

[6] Xilinx, *AM009 - Versal ACAP AI Engine*, https://www.xilinx.com/support/documentation/architecture-manuals/am009-versal-ai-engine.pdf, 2020. (visited on 06/21/2021).

[7] S. Ahmad, S. Subramanian, V. Boppana, S. Lakka, F. Ho, T. Knopp, J. Noguera, G. Singh, and R. Wittig, "Xilinx First 7nm Device: Versal AI Core (VC1902)," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1–28. DOI: 10.1109/HOTCHIPS.2019.8875639.

[8] Xilinx, *UG1366 - VCK190 Evaluation Board - User Guide*, https://www.xilinx.com/support/documentation/boards_and_kits/vck190/ug1366-vck190-eval-bd.pdf, 2021. (visited on 06/25/2021).

[9] ——, *WP505 - Versal: The First Adaptive Compute Acceleration Platform (ACAP)*, https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf, 2020. (visited on 06/21/2021).

[10] M. Iriarte, "The path to smarter, autonomous radar and ew platforms," *Military Embedded Systems*, vol. 14, no. 1, pp. 20–23, 2018.

[11] B. Mehlig, *Artificial Neural Networks*, In arXiv: 1901.05639v2 [cs.LG], 2019.

[12] N. M. Nasrabadi, "DeepTarget: An Automatic Target Recognition Using Deep Convolutional Neural Networks," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 55, no. 6, pp. 2687–2697, 2019. DOI: 10.1109/TAES.2019.2894050.

[13] H. Du, T. Jin, Y. He, Y. Song, and Y. Dai, "Segmented convolutional gated recurrent neural networks for human activity recognition in ultra-wideband radar," *Neurocomputing*, vol. 396, pp. 451–464, 2020. DOI: 10.1016/j.neucom.2018.11.109.

[14] X. Gao, G. Xing, S. Roy, and H. Liu, "Ramp-cnn: A novel neural network for enhanced automotive radar object recognition," 2020. DOI: 10.1109/JSEN.2020.3036047.

[15] M. Bader, A. Bode, H.-J. Bungartz, M. Gerndt, G. R. Joubert, and F. Peters, *Parallel Computing : Accelerating Computational Science and Engineering (CSE)*, ser. Advances in Parallel Computing: v.25. IOS Press, 2014.

[16] A. Gupta, "Architecture Apocalypse Dream Architecture for Deep Learning Inference and Compute," in *Embedded World*, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/EW2020-Apocalypse-Dream-Arch-DeepLearning-Inference-AICore.pdf.

[17] S. Z. Gurbuz, H. D. Griffiths, A. Charlish, M. Rangaswamy, M. S. Greco, and K. Bell, "An Overview of Cognitive Radar: Past, Present, and Future," *IEEE Aerospace and Electronic Systems Magazine*, vol. 34, no. 12, pp. 6–18, 2019. DOI: 10.1109/MAES.2019.2953762.

[18] L. Melvin William and A. Scheer James, "8. Ground-Based Early Warning Radar (GBEWR): Technology and Signal Processing Algorithms," in *Principles of Modern Radar, Volume 3 - Radar Applications*. Institution of Engineering and Technology, 2015. DOI: 10.1049/SBRA503E_ch8.

[19] H. You, X. Jianjuan, and G. Xin, *Radar Data Processing with Applications*, ser. Wiley - IEEE Ser. John Wiley & Sons, Incorporated, 2016. DOI: 10.1002/9781118956878.

[20] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, https://www.tensorflow.org/, Software available from tensorflow.org, 2015. (visited on 06/21/2021).

[21] Xilinx, *Vitis-AI: Adaptable and Real-Time AI Inference Acceleration*, https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html, 2020. (visited on 06/25/2021).

[22] ——, *PetaLinux Tools*, https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html, 2020. (visited on 06/25/2021).

[23] Z. Bahman, *Radar Energy Warfare and the Challenges of Stealth Technology*. Springer, 2020, vol. 1st ed. 2020. DOI: 10.1007/978-3-030-40619-6.

[24] B. Van Veen and K. Buckley, "Beamforming: A versatile approach to spatial filtering," *IEEE ASSP Magazine*, vol. 5, no. 2, pp. 4–24, 1988. DOI: 10.1109/53.665.

[25] L. Wei and W. Stephan, *Wideband Beamforming : Concepts and Techniques*, ser. Wiley Series on Wireless Communications and Mobile Computing. Wiley, 2010, pp. 1–18. DOI: 10.1002/9780470661178.

[26] D. Antonsson and M. Li, "Ultrasonic Source Localization with a Beamformer Implemented on an FPGA Using a High-density Microphone Array," URL: https://hdl.handle.net/20.500.12380/255327, M.S. thesis, Chalmers University of Technology, 2018.

[27] I. N. da Silva, D. Hernane Spatti, R. Andrade Flauzino, L. H. B. Liboni, and S. F. dos Reis Alves, *Artificial Neural Networks: A Practical Course*. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-43162-8.

[28] D. Rumelhart, R. Williams, and G. Hinton, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. DOI: 10.1038/323533a0.

[29] V. K. Ojha, A. Abraham, and V. Snášel, "Metaheuristic design of feedforward neural networks: A review of two decades of research," *Engineering Applications of Artificial Intelligence*, vol. 60, pp. 97–116, 2017. DOI: https://doi.org/10.1016/j.engappai.2017.01.013.

[30] J. Schmidhuber, "Deep learning in neural networks: An overview," 2014. DOI: 10.1016/j.neunet.2014.09.003.

[31] D. Maturana and S. Scherer, "VoxNet: A 3D Convolutional Neural Network for real-time object recognition," in *IEEE International Conference on Intelligent Robots and Systems*, vol. 2015-December, Robotics Institute, Carnegie Mellon University, 2015, pp. 922–928. DOI: 10.1109/IROS.2015.7353481.

[32] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach and D. Blei, Eds., ser. Proceedings of Machine Learning Research, vol. 37, Lille, France: PMLR, Jul. 2015, pp. 448–456. DOI: 10.5555/3045118.3045167.

[33] J. French, "The time traveller's CAPM," *Investment Analysts Journal*, vol. 46, no. 2, pp. 81–96, 2017. DOI: 10.1080/10293523.2016.1255469.

[34] D. Zissis, E. K. Xidias, and D. Lekkas, "A cloud based architecture capable of perceiving and predicting multiple vessel behaviour," *Applied Soft Computing*, vol. 35, pp. 652–661, 2015. DOI: https://doi.org/10.1016/j.asoc.2015.07.002.

[35] N. Sengupta, M. Sahidullah, and G. Saha, "Lung sound classification using cepstral-based statistical features," *Computers in Biology and Medicine*, vol. 75, pp. 118–129, 2016. DOI: 10.1016/j.compbiomed.2016.05.013.

[36] Xilinx, *WP506 - Xilinx AI Engines and Their Applications*, https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf, 2020. (visited on 06/21/2021).

[37] A. Nader and D. Azar, "Searching for activation functions using a self-adaptive evolutionary algorithm," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '20, Cancún, Mexico: Association for Computing Machinery, 2020, pp. 145–146. DOI: 10.1145/3377929.3389942.

[38] M. Nagel, M. V. Baalen, T. Blankevoort, and M. Welling, "Data-Free Quantization Through Weight Equalization and Bias Correction," *2019 IEEE/CVF International Conference on Computer Vision (ICCV), Computer Vision (ICCV), 2019 IEEE/CVF International Conference on*, pp. 1325–1334, 2019. DOI: 10.1109/ICCV.2019.00141.

[39] M. Blott, "Overview of Deep Learning and Computer Architectures for Accelerating DNNs," in *Hot Chips 2018 Tutorial*. [Online]. Available: https://old.hotchips.org/hc30/0tutorials/T2_Part_1_Overview_finalv3.pdf.

[40] Xilinx, *UG1079 - AI Engine Kernel Coding - Best Practices Guide*, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1079-ai-engine-kernel-coding.pdf, 2021. (visited on 06/21/2021).

[41] ——, *UG1273 - Versal ACAP Design Guide*, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1273-versal-acap-design.pdf, 2020. (visited on 06/21/2021).

[42] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on, CVPR*, pp. 6071–6079, 2017. DOI: 10.1109/CVPR.2017.643.

[43] K. Brooker, "Google on the brain," *Fast Company*, no. 235, pp. 76–92, 2019. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=bsu&AN=138329165&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[44] F. Libano, B. Wilson, M. Wirthlin, P. Rech, and J. Brunhaver, "Understanding the Impact of Quantization, Accuracy, and Radiation on the Reliability of Convolutional Neural Networks on FPGAs," *IEEE Transactions on Nuclear Science*, vol. 67, no. 7, pp. 1478–1484, 2020. DOI: 10.1109/TNS.2020.2983662.

[45] MLCommons, *MLPerf*, https://mlcommons.org/en/. (visited on 06/21/2021).

[46] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Computer Vision and Pattern Recognition (CVPR), 2016 IEEE Conference on*, pp. 770–778, 2016. DOI: 10.1109/CVPR.2016.90.

[47] Stanford Vision Lab and Stanford University and Princeton University, *ImageNet image database*, https://www.image-net.org/. (visited on 06/21/2021).

[48] B. Du, S. Azimi, C. de Sio, L. Bozzoli, and L. Sterpone, "On the Reliability of Convolutional Neural Network Implementation on SRAM-based FPGA," *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, pp. 1–6, DOI: 10.1109/DFT.2019.8875362.

[49] I. Goodfellow, Y. Bengio, and A. Courville, "Back-propagation and other differentiation algorithms," in, http://www.deeplearningbook.org, MIT Press, 2016, p. 200.

[50] T. Liu, S. Fang, Y. Zhao, P. Wang, and J. Zhang, *Implementation of training convolutional neural networks*, 2015. arXiv: 1506.01195 [cs.CV].

[51] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds. arXiv: 1409.1556 [cs.CV].

[52] United Nations, *United Nations Charter, Chapter VII: Action with Respect to Threats to the Peace, Breaches of the Peace, and Acts of Aggression*, https://www.un.org/en/about-us/un-charter/chapter-7. (visited on 06/21/2021).