LUND
UNIVERSITY

Implementation of an 8-bit Dynamic Fixed-Point Convolutional Neural Network for Human Sign Language Recognition on a Xilinx FPGA Board

Department of Electrical and Information Technology
Lund University

Master of Science Thesis
— March 17, 2019 —

Author: Ricardo Núñez Prieto

Supervisor: Liang Liu
Examiner: Erik Larsson

Printed in Sweden
Tryckeriet i E-huset, Lund

Para Erik,
una pequeña red neuronal que está en camino...
Diciembre, 2018

Abstract

The goal of this thesis work is to implement a convolutional neural network on an FPGA device with the capability of recognising human sign language. The set of gestures that the neural network can identify has been taken from the Swedish sign language, and it consists of the signs used for representing the letters of the Swedish alphabet (a.k.a. *fingerspelling*).

The motivation driving this project lies in the tremendous interest aroused by neural networks in recent years for its ability for solving complex problems and its capacity to learn by example. More specifically, convolutional neural networks are being extensively used for image classification, and this project aims to design a hardware accelerator to compute the convolutional layers of such type of network topology and test its accuracy and performance when dealing with human sign language. Further applications for this hardware solution can be placed in the educational field, specially addressed to children with impaired hearing or as a translation system in specific situations.

The network topology of choice is Zynqnet, proposed by Gschwend in 2016, which is a topology that has already been implemented successfully on an FPGA platform and it has been trained with the large picture dataset provided by ImageNet, for its popular image recognition contest. In this regard, the aim of this work is not to propose a new neural network topology but to re-use an existent one by introducing some improvements like the utilisation of an 8-bit dynamic fixed-point scheme and challenge it with a different but related task, like human sign language recognition.

The methodology followed to carry out a successful hardware implementation has consisted, first, of the installation and setup of a reliable framework used for the training of the neural network. Different frameworks were tried out, like MATLAB or Caffe, but finally, DIGITS from NVIDIA was the more convenient due to its graphical environment and because it provides all the compatibility and drivers needed to run together with the GPU used in this project. Then, an image dataset of more than 13,000 pictures of hand gestures has been built up to grant enough input data

for the framework to fine-tune ZynqNet for the new task, i.e. to provide the neural network with the ability to classify the different hand-signs into its corresponding alphabet letter. In parallel, the Register-Transfer Level (RTL) abstraction of the hardware architecture has been generated using a High-Level Synthesis tool chain, in which the algorithmic descriptions are written in C/C++. Finally, the validation of the design has been done by means of co-simulation techniques where the golden data obtained with the C test bench is compared with the output data of the RTL implementation, and all of it within the simulation environment provided by the Vivado Design Suite.

As a result, the best-performing obtained solution achieved an accuracy of 80.1% in the inference test and a frame rate of 6.4 FPS with a clock frequency of 250 MHz.

Popular Science Summary

Neural networks are becoming more and more ubiquitous in our everyday lives, many times in ways that we do not even realise. Artificial Intelligence (AI) is extensively used nowadays to improve the user's experience with digital technologies, for instance, the on-the-fly translation service provided by Skype and Google. In other fields like robotics, improvements in object detection from the hand of machine learning, allow robots and autonomous cars to take better decisions. And in medicine, automatic detection of blood diseases like leukaemia and lymphoma, powered by neural networks algorithms, have accelerated and improved diagnosis. So the list of applications found in many diverse fields goes on and on.

Now, let's picture yourself in the hypothetical situation in which you have a friend or a relative who has been born with a hearing impairment. This person has been taught sign language from an early age on a specialised school, and you would like to learn sign language too so you both can have meaningful and pleasant communication. Furthermore, you want to be able to help this person in day-to-day situations where deaf people can be in clear disadvantage like a routine visit to the doctor or administrative processes.

You learn from one of your classmates at the university about a mobile app which employs a deep neural network to recognise human sign language just by using the phone camera. The application translates the captured sequence of gestures from video to written text in real-time and automatically reproduces the message in the phone speaker. It also includes a sign language tutorial which can help you to rapidly learn to communicate by using your hands. The software records your gestures and improves your learning abilities by telling how accurate are your movements.

Well, the situation just described is something that I believe it is not far to happen. The computational power found in embedded systems such as mobile phones is growing by the day. So far, at least, one can find mostly solutions that work one-way, that is, they convert speech into sign language, by mapping spoken language to signs and using a virtual animated human-like avatar. Fewer solutions can translate sign language into speech.

Some of them use special gloves with position sensors not so pleasant to wear by the *signer* person, and others use 3D cameras that can pick the speaker's body gestures and then compare the obtained frame sequence with a reference frame stored in a dictionary. These solutions though, rely on mapping methods and are limited in terms of the number of gestures they can interpret.

I really believe that neural networks are a game changer and they will bring powerful, elegant solutions to the kind of problems described above. This thesis work aims to provide proof-of-concept of a feasible implementation of a neural network trained for recognition of human sign language. The number of gestures to recognise is limited to the Swedish alphabet, and the network must show an acceptable level of prediction accuracy, throughput and area utilisation.

The content of this thesis is addressed to a variety of public: from people interested in neural networks in general, and the significant development experienced in the field in recent years, to people interested in learning the basic concepts and the methodology for training neural networks, or practitioners who look to implement a deep learning model on an AI accelerator.

Acknowledgements

First, I am grateful to professor Liang Liu for supervising this work and for providing fruitful ideas and support.

I want to express my gratitude to Lund University for giving me the opportunity to take this Master Program in Embedded Electronics Engineering and also for the four years I spent working for the MAX-IV project.

Thanks a lot to my teachers and classmates with whom I have shared these three years: Arun, Leo, Luis, Berta, Mayra, Shenba... thanks for the inspiring and funny little moments we have passed together and made of this a more pleasant journey. I wish you all the best of luck in both your professional and life projects.

Last but not least, I would like to thank especially my wife and my daughter for their love and support that carried me through every challenge I have faced, and also to my parents for their esteem and being always there.

Ricardo Núñez Prieto
Lund, February 2019

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial Neural Networks (ANNs) are computational models inspired by the human brain's neural circuits that aim to solve complex real-world problems in fields as diverse as statistical data analysis, biology or economics.

ANNs consist of anything from hundreds to thousands, or even millions of processing nodes (a.k.a. neurons) that are organised in a series of layers. A typical neural network has an input layer, which connects the system to the external inputs, an output layer, and the remaining layers in between called hidden layers which are fully-connected with the layers on either side, meaning that each neuron of a hidden layer is connected to every neuron in the previous layer.

A neural network mimics a human brain mainly in two ways: 1) it learns by example, through a process of training, hence the knowledge is not programmed *a priori*, and 2) the knowledge is stored in the inter-connections or synapses between the different neurons in the form of synaptic weights, which can be understood as the grade of influence of an individual neuron over the surrounding ones when fired.

During the 80s and the 90s, neural networks were developed as software applications and were simulated in computers that used general-purpose microprocessors because they offered a good trade-off among features like small size, low price, low power and high performance. These computer systems could perform complex tasks such as mammographic analysis [1] in the field of medical imaging, or prediction of the sea-surface water temperatures [2] in the field of palaeoceanography. The drawback of this approach is that microprocessors have sequential von-Neumann architectures and are not fully able to replicate the massive parallelism present on neural networks. For that reason, efforts were made to implement ANNs as custom hardware accelerators, since many applications, in fact, require high-speed

operations. The first successful ANN implemented in hardware was realised in 1991 using reconfigurable hardware, concretely on a logic cell array (LCA) from Xilinx [3]. It could compute 4.48 billion CPS (or inter-connections per second, meaning the rate of multiplication-accumulate operations during testing phase), making it suitable for applications in computing vision and, in particular, in industrial inspection tasks. Since then, technology advancements in VLSI have accelerated the transition of artificial neural networks from standard desktop computers and supercomputers to applications in embedded systems. The possibility of having available neural networks as a microelectronic component is very appealing as there is a need for its use in decentralised or mobile/embedded systems (like the upcoming IoT devices).

1.1 Background and Motivation

Convolutional Neural Networks (CNNs) or ConvNets are a class of specialised neural networks, known as deep neural networks, that are mostly used for image recognition tasks because they are very efficient in pattern recognition and feature extraction from the input images. In effect, they can be used in any task where the input data presents some hierarchical structure, with simple local patterns assembled in larger patterns which, in turn, are part of bigger complex arrangements.

CNNs belong to a new category of networks known as Deep Neural Networks (DNNs), which differ structurally from the early neural networks in the fact that they have more than one hidden layer. Aside from convolutional networks, there are other types of deep neural networks like autoencoders, deep belief networks (DBN) and recurrent neural networks (RNN). In particular, CNNs introduce a particular type of layer called convolutional layer which is simply a filter sliding, or convolving, around the input image computing element-wise multiplications.

Modern CNNs, as we know them today, derive from the works of LeCun et al. [4] who in 1998 presented the first successful one, named LeNet5, inspired by the ideas and previous works published by Fukushima [5]. The novelty here was to use an already known algorithm for the training of the network, named backpropagation. Essentially, backpropagation is what causes the network to learn. It is a feedback process where the output produced by the network is compared with the output it is meant to produce. The difference between them is used to modify the synaptic weights of the connections between the nodes in the network, working backwards, that is, from the output layer, through the hidden layers, back to the input layer.

After the appearance of LeNet5, research in CNNs continued progressing, although slowly. At the same time, more computing power began to be

available thanks to the development of powerful parallel Graphics Processing Units (GPUs) which also became general-purpose computing tools due to the arrival of CUDA, a C-like programming language for parallel computing, especially suited for GPUs and developed by NVIDIA. It was not until 2010 that it was published one of the first implementations of a ConvNet in a GPU [6]. Thanks to the parallel processing capabilities of GPUs, the time spent for the training of CNNs could be reduced in two orders of magnitude as compared with using a CPU [7]. The latter represented a significant advance, as until then it had been not possible to study deep CNNs with high-resolution input images due to the tremendous computational power required for their training. Since then, deep neural networks have been pushing the research in machine learning with renewed interest. In particular, deep learning methodology is behind the best performing systems in machine learning applications such as self-driving cars [8], grandmaster-level computer chess programs with self-learning capabilities [9], machine translator engines [10], speech-recognition [11] and so on.

It is indisputable that deep neural networks are one of today's hot topics in the research field of artificial intelligence. The potential for possible applications is clearly growing by the day and companies like Google, Facebook and Microsoft are investing a considerable amount of resources in the development of new products and exploring different hardware architectures that can take over GPUs. As an example, Intel recently announced the release of its Neural Network Processor (NNP) [12] that will compete with Google's Tensor Processing Units (TPUs) [13], both new custom ASIC chips made specifically for running deep neural networks. Given the above-stated background, the main motivation behind this thesis work is twofold: to gain insight into the mechanisms of artificial learning processes, and to bring together one of the more exciting fields in today's research with the aspiration to develop applications that can contribute to society.

1.2 Project Goals and Main Challenges

The aim of this Master's thesis is to design and implement a hardware-accelerated convolutional neural network for the recognition of human sign language. The system should be able to recognise the different hand positions corresponding to each of the letters of the sign language alphabet. The purpose is to endow the system with the ability to interpret fingerspelling which is a method of spelling words just by using hand gestures.

In a first version, the input images are previously preprocessed and fed to the neural network as a predetermined sequence of static images. In a more elaborated version, the user can provide the input by means of a

digital camera, and the video stream of images is captured and analysed by the neural network.

To simplify things, the sign language of choice is preferably the Swedish sign language (*Svenskt teckenspråk* or STS) because it is one-handed when used for fingerspelling (in opposition, for instance, to the British system that requires the two hands). It has a total of 29 positions, and most of them are static except for the distinctive vowels *å*, *ä* and *ö* that require additional hand motion.

The system's goal is to be able to recognise and translate STS into a written text message, thus the communication works only in one direction. A possibility to be explored in a future stage is to transform the system into a translation node allowing the communication in both directions between two users.

The main challenges identified *a priori* for the realisation of the project encompass different issues like the following:

- Selection of the proper CNN topology to perform the intended task.
- Selection of the proper hardware platform to implement the neural network.
- Efficient hardware implementation of the algorithm with particular attention to the on-chip memory limitations.
- Efficient number representation with minimal impact in the network's inference accuracy.
- Build a training dataset as large as possible from a broad group of volunteers in order to obtain a trained model with the highest prediction accuracy.

1.3 Approach and Methodology

Regarding the suitable hardware platforms, while GPU architecture is remarkably like that of a neural network, an FPGA implementation is ideal due to its compatibility with the high level of parallelism existent in artificial neural networks. The limiting factor of such parallelism is the possible lack of on-chip memory and the memory bandwidth causing bottlenecks when moving data across physical chip boundaries. Furthermore, the option of an FPGA seems more fitted than an ASIC for the implementation of a hardware-accelerated CNN for prototyping purposes.

For the training of the network, a compelling approach that is known as *transfer learning* is the preferred option. This method takes the computational model of a ConvNet that has been already pretrained on a

related task, preferably with the help of an extensive dataset. Then, the output layer or *classifier* is removed and replaced by another one that fits the number of categories required for the new task. Finally, the modified network is retrained with the new dataset to fine-tune the weights by means of *backpropagation*. The obtained new weights can be either hard-coded or read from external memory by the hardware version of the neural network implemented in the FPGA.

Finally, the following are additional aspects to be considered regarding the FPGA implementation of a neural network that one can find in the related literature:

1. **Data types and bit-width.** Floating-point operations are not natively supported in most of FPGAs and its implementation adds computational complexity. So, the alternative is to use fixed-point arithmetic, much more straightforward and less resource consuming. Regarding the number of bits needed to represent the network weights, neither a broad dynamic range nor very high precision is needed. Because the weights obtained during the training phase are represented in floating-point format, conversion to fixed-point format is required when transferring these weights to the FPGA. By reducing the bit-width, it is also reduced the size of the computation units, allowing more operators within the same area of logic. According to Guo *et al.* [14], 8-bit is the limit to ensure negligible accuracy loss although very high-performance designs can be even achieved by using binary weights (1 bit-width), as reported in [15] at the expense of accuracy loss.

2. **All-convolutional network approach.** Neural networks that consist only of convolutional layers are called *all-convolutional* networks. Making use of this design strategy results in a more simpler design and saves FPGA resources mainly in two ways: first, all-convolutional networks need less memory bandwidth since they do not contain any fully-connected layers (without the latter, there is no need to load a new weight for every single MAC operation). Second, pooling layers can also be eliminated from the network; pooling operations are periodically inserted between successive convolutional layers to perform down-sampling of the previous layer output, hence reducing the number of parameters and computation in the network. However, as demonstrated by Springenberg in a recent publication [16], adding pooling operations in a network does not always improve the CNN performance. Instead, it seems that if the network is large enough relative to the dataset that it is being trained on, it can learn all the necessary features and its invariants just with convolutional layers.

3. **Efficient convolution operations.** Convolutional layers make extensive use of concurrent multiply-accumulate units (MAC units). An efficient way to implement the convolution operation may be using parallel computing structures like *systolic arrays*, as proposed by Kung [17]. It is efficient because exploits locality: all operand data and partial results are stored within the processor array itself, and there is no need for any memory access during each operation. Each of the processing elements in the array performs its computation as soon as all the required data is available, and when finished, the resulting data is propagated on to its neighbours, with no need of synchronism with a global clock.

Chapter 2

Basic Concepts

First things first. This chapter covers the basic foundations and concepts that will help the reader to understand how a neural network works, its building blocks and different types of network structures or topologies. Furthermore, the learning algorithm used by neural networks to learn from a set of input examples is also explained. And finally, it introduces ZynqNet, the neural network topology that has been implemented in the hardware accelerator.

2.1 Foundations of Artificial Neural Networks

Neurons

Many of the terminology used for neural networks is borrowed from the field of neuroscience which is not a surprise at all as the idea behind neural networks is to try to solve complex problems by mimicking the human brain structure. Of course, even the simplest brain of the humble fruit-fly still presents an organisational complexity far from being replicated by any artificial neural network developed by humans nowadays. Living brain cells present complex behaviours which are not yet fully understood so the concept of *neuron* in the context of artificial neural networks is just an approximation of the functionality of a real biological neuron.

Hereby, in this context, a neuron is just a computational node with one or more numerical data inputs and a single output. This output is broadcasted, therefore allowing neurons to be interconnected with other neurons in the fashion of super-structures known as *layers*. Each neuron performs the linear combination of its weighted inputs, that is, it sums all of the inputs, each one previously multiplied by its corresponding constant,

also known as *weight*. An additional extra term, the *bias*, is added to the final sum of weighted inputs. The purpose of adding the bias is to work as an offset value, and this is better understood after reading the next paragraphs.

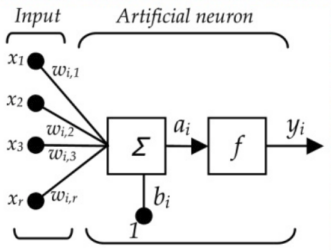

Figure 2.1: Representation of an artificial neuron with multiple inputs. The bias b_i is added to the linear combination of the neuron weighted inputs $x_1, x_2, ...x_r$ and the obtained sum a_i is then used as the input parameter of the activation function f to generate the neuron output y_i *(Source: derived from Tanikić and Despotović, 2012 [18]).*

By definition, neurons are non-linear units, and this is a valuable property a neuron must have in order to be able to interpret non-linear real-world data. The latter is achieved by applying a non-linear function, also known as *activation* function, to the linear combination of weighted input. By extension, neurons are also referred to as *activations*, and it is a term often used in the field. Figure 2.1 is a representation of an artificial neuron as described above.

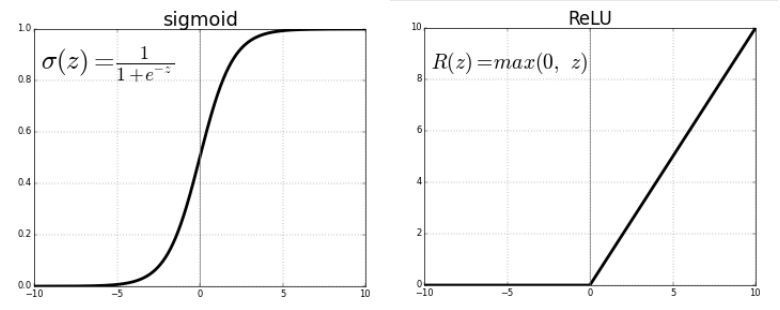

Figure 2.2: Graphs for the sigmoid and the ReLU functions.

Sigmoid functions have been used in the past as activation functions when modelling neural networks. They have a characteristic S-shape, and the output of these functions tends to saturate when the input reaches certain levels. Typical examples of sigmoid functions are the hyperbolic

tangent, $tanh(z)$ or the logistic function, $1/(1+e^{-z})$. More recently, sigmoid functions have been abandoned as they proved very difficult to train due to its computational complexity and have been replaced by a much more popular and still effective activation function known as ReLU defined as $f(z) = max(0, z)$. It states that the output of the activation function will be zero for negative values (inactive) or the identity function for positive values of the linear combination of the inputs.

The ReLU function is a much simpler, elegant solution, and very cheap to implement in hardware. Essentially, sigmoid functions as well as the ReLU function, aim to imitate the behaviour of the biological neurons by transiting from an inactive state to an active state when the input signals reach a certain threshold level.

Sigmoids belong to a larger family of functions known as *discriminant functions*, as it squishes or maps the inputs into one of two possible values (-1, +1) or (0, +1) depending on the sigmoid function of choice. Due to this property, the neural network can be modelled as a decision tree providing different outputs based on the value of its inputs.

In connection with the activation function, the purpose of the *bias* term is to shift the activation output in order get better fitting between the prediction and the input data. During the training phase of the neural network, both the bias and the weights are refined in each iteration, achieving better fitting and providing the neuron with the capability of learning. Very often weights and bias are also referred to as the *trainable parameters* of the network.

Layers

As mentioned in the previous section, neurons are grouped in bigger structures named *layers*. Typically, a neural network consists of an input layer, an output layer and one or more layers in between also known as *hidden* layers. The latter have no direct connection with the outside world (hence the name "hidden"). The hidden layers are, in fact, the source for the ability of the neural network to interpret non-linearities in the input data. Without hidden layers, the output is directly connected to the inputs, and its value corresponds to a linear combination of such inputs. Hence, this type of network would be limited to be used only in linear decision problems, such as linear classifiers.

By adding hidden layers, with its corresponding neurons performing non-linear activations, the capabilities for problem-solving of the neural network become greatly enhanced. With the proper training, neurons grouped in the same layer get specialised at looking to some unique features in the input data. For instance, in a neural network performing image recognition, each

layer is engaged in extracting specific features from the previous layer, such as edges, shades or colours. What precisely a convolutional neural network considers to be a significant feature is defined while learning.

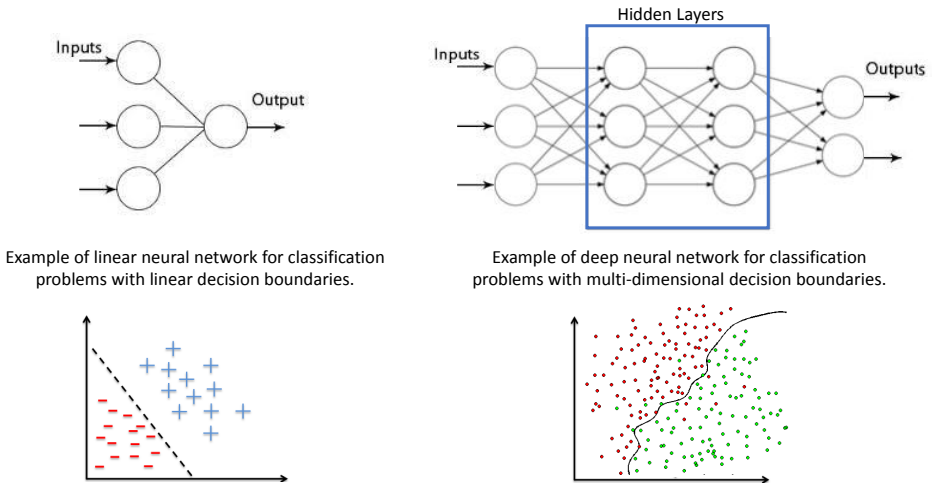

Figure 2.3: Example of a neural network without hidden layers used in a binary classification problem (left) versus a neural network with hidden layers performing as a non-linear classifier (right).

Fully-Connected and Convolutional Layers

Depending on the way neurons are connected to other neurons from the adjacent layers, the latter can be classified as *fully-connected* layers or *convolutional* layers, although convolutional layers are a subset of the fully-connected ones. On a *fully-connected layer*, each neuron is connected to all of the other neurons in the previous layer. An example of a very popular neural network with this type of architecture is the so-called *multilayer perceptron* (Figure 2.4) that has been used in the past for image and speech recognition tasks.

Because each connection has associated its own weight, it is easy to see that in the particular case of image recognition, where the input data consist of relatively large pictures with thousands of pixels, the number of weights required by a multilayer perceptron would grow enormously. A way to deal with that is by using *convolutional layers*, where each neuron is connected to a limited number of neurons in the previous layer. Not only the number of weights gets reduced by limiting the number of connections, but all the neurons in the same layer also share the weights. That is what is known as *weight sharing* and allows to reduce the number of trainable parameters drastically, which is particularly important if the neural network

is implemented on an FPGA with limited memory resources.

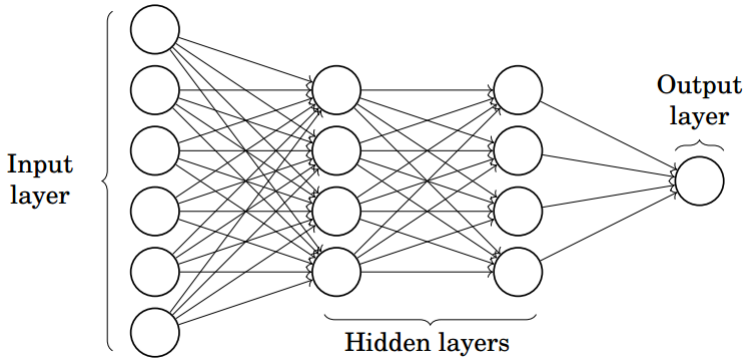

Figure 2.4: Diagram of a Multilayer Perceptron Network *(Source: Pavlovsky, 2017 [19])*.

In principle, a convolutional neural network should have the same learning capabilities than a fully-connected one. The difference is that fully-connected layers perform a *global* operation, as they can introduce any kind of dependence derived from the input data, and convolutional layers perform a *local* operation as each neuron is *looking* at a small portion of the data in the previous layer and that is why they perform so well in image analysis tasks. That small portion of data that is being analysed is also known as the *local receptive field* or *convolution window* and the set of weights used to calculate the weighted sum is known as *kernel* or *filter*. One can think about it as applying a filter to an image by sliding it all along the pixels. Each pixel of the output image is a linear combination of the values contained in its corresponding local receptive field, which is formed by the current input pixel and its neighbouring pixels, as depicted in Figure 2.5.

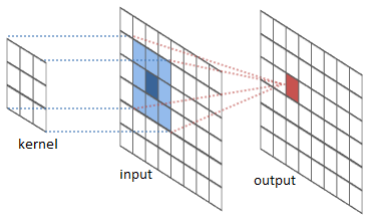

Figure 2.5: 2-D discrete convolution obtained by sliding the *kernel* along the input image *(Source: Intel Labs [20])*.

Typically, in convolutional neural networks, a fully-connected layer is placed at the final stage of the neural network as a *classifier*, to separate the

data into the various categories. Because each of its neurons has connections to all the elements in the previous layer, they can extract any kind of relevant dependencies from the input data. The fully-connected layer is the cause for the high-level reasoning in the neural network.

Activation Volumes and Feature Maps

At this point, it is necessary to introduce the idea of *volumes* since the input data is not organised just as a two-dimensional array but in a three-dimension matrix or *volume*. Let's consider an input image of size 8x8 and composed of 3 channels (Fig. 2.6), namely the standard RGB colour channels. The latter adds a new dimension to the input data, in this case, a depth of three channels, also referred as *slices*. Hence, the input data constitutes, in reality, a volume of data of size 8x8x3. It follows that the same applies to the kernel, which is not a 2-dimensional array of weights; for instance, a typical 3x3 kernel forms a volumetric filter of size 3x3x3 actually, as the kernel always has to have the same depth as the input volume.

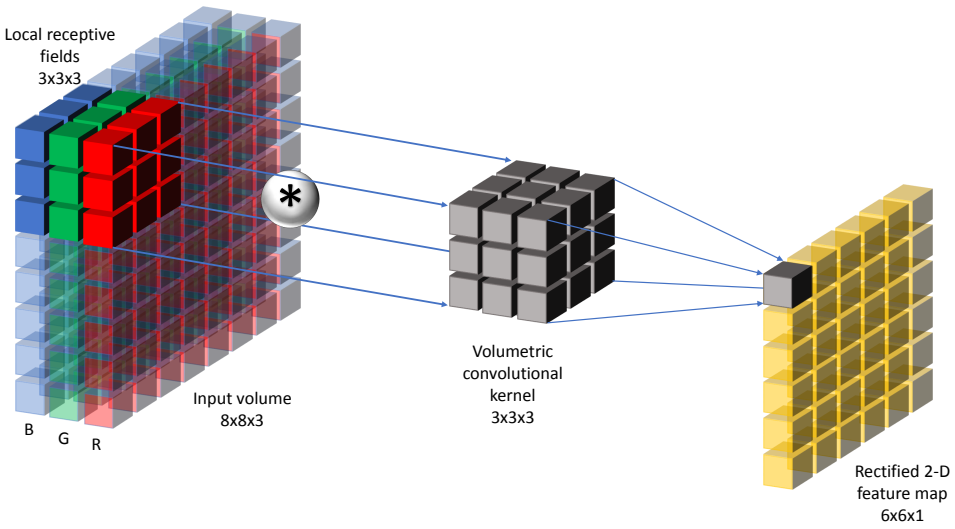

Figure 2.6: Representing a full-colour RGB input image as a volume and applying a volumetric convolutional filter.

Following with the example, after convolving each channel of the input volume with its corresponding kernel slice, the obtained output is a set of three *feature maps*. Then, these feature maps or matrices are added together with its corresponding bias, and the final sum is passed to the activation function or ReLU. The resultant 2-D matrix is an *activation output* or *rectified feature map* obtained from the input. As might be expected, in

order to extract more feature maps from the input, it just takes another different set of kernels and to convolve the input volume with them. One can generate as many features as needed, depending on different factors like the computational power, available memory or the type of input images one wants to analyse. Eventually, all these feature maps are piling up creating a 3-D volume of data again, often referred to as *activation* or *feature volume*.

Hyper-Parameters: Kernel Size, Stride and Padding

Complementary to the weights and the bias, which are network parameters that are refined or modified during the learning process, there is another set of parameters intrinsic to the neural network that is used to specify the structure of the layers. These parameters are known as *hyper-parameters* and are configurable values which are set before starting the training process. The main hyper-parameters of a CNN are the size of the receptive field, the kernel size, the padding, the stride length, and the dimensions of the activation volumes. Some of these hyper-parameters have been already mentioned in the previous sections, and the rest are detailed by following. There are other hyper-parameters, that are not listed here, that dictate the behaviour of the training algorithm and how it learns the parameters from the data. They are detailed in Chapter 3 which refers to the training of the CNN.

Kernel size

The size of the kernels that is used in the tapestry of convolutional neural network topologies available nowadays can be quite diverse. Typical sizes are 1x1, 3x3, 5x5 and 7x7. The heuristic rule for using a specific dimension relies upon the relative size of the feature that one needs to capture: the smaller the size of the feature to be extracted, the smaller the filter.

Stride

Another parameter to be considered is the *stride*, which refers to how the kernel slides along the input image. Typically, after performing the convolution operation on a local receptive field, the filter is shifted by one pixel, row-wise or column-wise, and placed over the next receptive field, and the process goes on until completing the output feature. The term *stride* is used to refer to the length of this displacement. It is also possible to use a shift bigger than one pixel, or *non-unity* strides, as a way to reduce the dimensionality of the activation volumes and the computational effort. For instance, a stride of 2 will produce an output with half the dimensions of the input (that is to say, the spatial dimensions of the input, height and width)

with a reduction factor of 4 in the total number of elements. An interesting discussion about the heuristic motivations for using different stride values is given by Kong and Lucey [21], who demonstrate that any non-unity stridden convolution can be replaced by a unity stridden convolution without loss of performance.

Padding

Padding is needed to keep the dimensions of the output volumes the same size of the input volumes. The standard method is to use *zero-padding* which consists in filling with zeros around the borders of the feature maps. If no padding were applied, the spatial dimensions of the activation volumes (height and width) would be reduced after each convolution progressively, washing quickly away all the information data contained in the borders. This effect can be seen more clearly by considering the following example, where a 5x5 matrix is convolved with a 3x3 kernel. Without padding, the result is a 3x3 matrix, and attempting to convolve that matrix with another 3x3 kernel will result in a 1x1 matrix. However, if the original 5x5 matrix is padded with zeros all around the borders, the result would be another 5x5 matrix (dimensions keep the same size), and by padding this matrix again, one can perform as many 3x3 convolutions as wished.

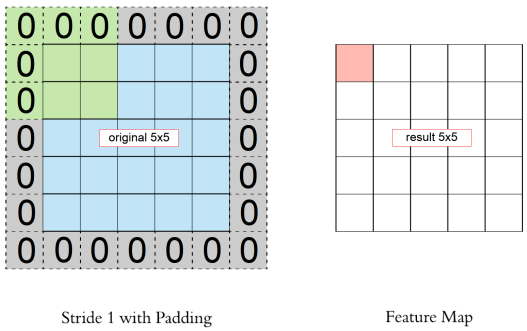

Figure 2.7: Example of 2-D matrix convolution with zero-padding *(Source: derived from Dertat, 2017 [22]).*

Pooling Layers

Another commonly used type of layer is the *pooling layer*. Its purpose is to reduce the spatial size (height and width, but not the depth) of the feature volumes, and this is achieved by inserting pooling layers along the structure of the network in a distributed way (see Figure 2.8).

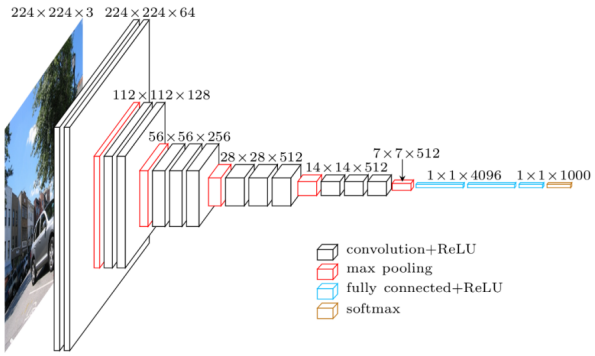

Figure 2.8: 3-D representation of a convolutional neural network with pooling layers *(Source: Cord, 2017 [23])*.

Mainly there are three types of pooling: *max*, *min* and *average* pooling. As a matter of example, Figure 2.9 shows a typical *max* pooling layer, with a filter of size 2x2 and a stride of 2. It decimates every slice of the feature volume, discarding exactly 75% of the activations. The *max* pooling operation would, in this case, take the maximum over four elements on the receptive field delimited by the filter.

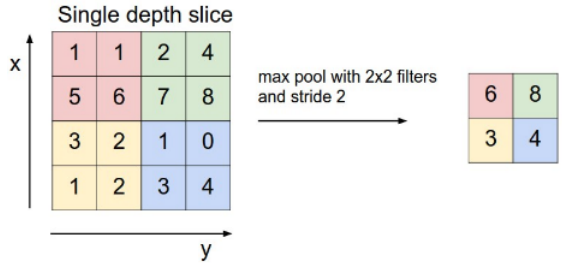

Figure 2.9: Example of a *max* pooling operation using 2x2 filters and stride 2 *(Source: Karpathy [24])*.

More recently, neural network topologies have shown up with a new approach, the *all-convolutional net*, consisting in to get rid of the pooling layers, based on a late paper from Springenberg *et al.* [16], who propose to discard the pooling layers in favour of an architecture that only consists of successive convolutional layers. To reduce the size of the activation volumes, they suggest the use of larger strides (typically a stride of 2) in the convolutional layers once in a while. As a matter of fact, that approach has also been implemented in Zynqnet [25], the network topology used in this project.

It is worth to mention that the ZynqNet architecture shown in Figure A.1

has only a pooling layer located at the final stage of the neural network, right after the convolutional layer *conv*10. This pooling layer performs a *global average pooling* operation that generates a flattened vector with dimension equal to the number of categories. That is an improvement also inherited from the authors of the SqueezeNet [26] architecture, who replaced the last fully-connected layer of the network with a convolutional layer and a global average pooling. The main advantage behind this modification is to reduce the memory requirements significantly, due to the vast amount of parameters associated with a fully-connected layer. That, together with the *all-convolutional net idea*, makes ZynqNet very suitable for FPGA implementations.

2.2 Backpropagation Algorithm

In the previous section, it was stated that during the training phase of the neural network, weights and bias are refined to get a better fitting between the obtained output and the desired output, iteration after iteration. This refinement mechanism is the one that makes possible that the neural network learns from the given examples and get the ability to generalise when later, during the test phase, is presented to new examples never seen before. This learning algorithm is known as *backpropagation* and consists in nothing more than the optimisation of a cost function, sometimes referred also as *loss* function. The input parameters of this function consist of all of the trainable weights and bias of the neural network, as many as hundreds of thousands, or millions. During the training phase, the examples are fed to the input layer of the neural network, and the data is propagated through the different layers all the way to the output, where the neurons on the fully-connected layer (which acts as a classifier) throw different values ranging on a specific interval. Ideally, only the output neuron which corresponds to the right category for the given input image should be active, and the rest of neurons should be inactive or zero-valued, but in reality, all of the output neurons display some value and the one with the most significant value will be the answer given by the CNN. The approach for optimising the cost function is to use the standard method of minimising the sum of the squared errors (SSE), where the errors are the difference between the actual value of each one of the output neurons and its target value. That needs to be done for all of the hundreds of thousands of training samples so, this technique takes the average of all of the SSE, and the result will be the total cost of the network that needs to be minimised.

To find out the amount of change to be applied to the weights and bias to minimise the cost function, the backpropagation algorithm calculates the

gradient of the function at a given starting point. Actually, it is the *negative* of the gradient that is the vector pointing towards the local minimum. Consequently, some weights will need to be increased whereas other weights will need to be reduced. The amount of change will depend on how significant is the activation value of the neuron in the previous layer associated to that weight. Hence, the change should be proportional; the bigger the activation, the bigger the change on the weights, and vice versa, therefore reinforcing the connections between neurons that have a desired effect on the output, and dimmering the ones that not. The net effect is to reduce the error and approach the desired output to its target value.

The algorithm is computing the changes needed in the weights and bias in order to find the local minimum of the cost function, by searching for the optimal path downhill along the *cost surface*, which ideally is in the direction of the negative gradient vector, and moving along that surface in small iterative steps or *deltas*. However, it should be noted that this calculation is done for each one of the training examples. For another training example, these changes will have different values, and hence, in the end, it is necessary to do an average of all of the desired changes in order to obtain an average value that will be kind of proportional to the gradient of the cost function. The name for this technique used for the training of neural networks is known as *gradient descent*.

Given the large size of the training datasets, it takes so much computational effort to calculate each one of the steps downhill the cost surface by trying to follow the optimal path, because the algorithm is taking into account all of the training examples at once. In practice, a very handy optimisation technique to speed-up the algorithm consists of randomly shuffling the whole set of the training examples and divide it into small lots or *batches* (also called *mini-batches*). Then, the backpropagation algorithm will calculate the deltas, and the obtained gradient will not be optimal, but just an approximation, as it will depend on the pictures contained in that mini-batch. The next step will be taken by applying the algorithm in the next mini-batch and so on.

Once the algorithm goes through all the mini-batches, which is called an *epoch*, the algorithm reshuffles the training examples and creates new mini-batches. This technique is extensively used nowadays and is known as Stochastic Gradient Descent (SGD). As shown in Figure 2.10, the path followed by the algorithm in order to find the local minimum is not a smooth optimal path but rather a random path, although oscillating around the optimal one.

At this point, a couple of question arises: How likely is it for the SGD

algorithm not to get trapped in a saddle point[1]? If not, is the local minimum found by the SGD algorithm the best one among all of the minima contained in the cost surface?

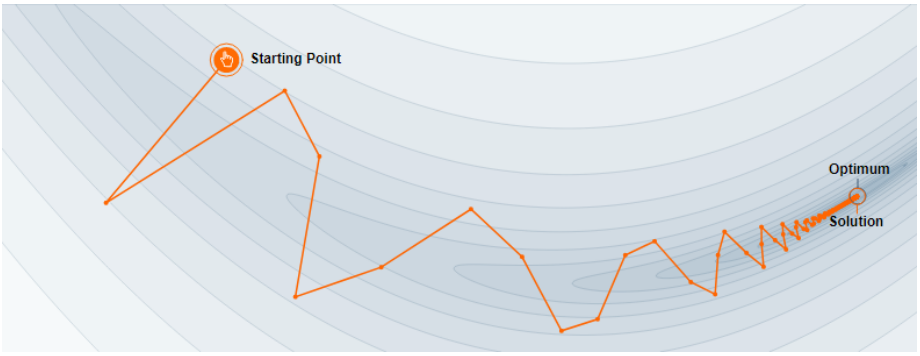

Figure 2.10: Stochastic Gradient Descent algorithm searching for the local minimum *(Source: Goh, 2017 [27])*.

Given the fact that the cost function is a non-convex function in a high-dimensional space, finding its absolute global minimum can be just a daunting computational task. One can visualise the loss function as a surface populated with thousands of local minima, maxima and even worse, saddle points were the SGD algorithm can get stuck. An analytic answer providing mathematical proof to this question is given by LeCun *et al.* [28] who demonstrate that while multilayer networks present numerous local minima, they are relatively easy to find, and they are all more or less equivalent in terms of quality when trying to minimise the cost function. Complementary, for the gradient descent algorithm may converge on a saddle point and spoil further optimisation of the cost function, a recent work from Lee *et al.* [29] demonstrates that the gradient descent can overcome this problem with random initialisation of parameters of the cost function (bias and weights), and the algorithm will almost surely converge to a local minimum.

Given the works mentioned above and prior experiences provided by numerous practitioners, it can be concluded that the backpropagation algorithm using the stochastic gradient descent is so far a mature and proven method for the training of convolutional neural networks.

[1] A saddle point is a singular point on the surface of a multivariable function where all the partial derivatives in the orthogonal directions are equal to zero, but the saddle point is neither a local maximum nor a local minimum of the function.

2.3 Neural Networks are Universal Approximators

Neural networks are the ultimate regression algorithm. That is an attractive and alternative approach for understanding what hides behind such fancy name of *neural networks*. One can look at them as universal function approximators, in the same way as the Taylor and Fourier series are function approximation techniques. The mathematical proof for this statement relies on the *universal approximation theorem* proven by Cybenko [30], who demonstrated that feed-forward[2] neural networks with only a single hidden layer containing a finite number of neurons can approximate any continuous function. Cybenko demonstrated the theorem for sigmoid activation functions, but the theorem can be extended to other nonlinear activation functions, provided they are continuous and bounded.

The following is an interesting and straightforward graphical demonstration provided by Nielsen [31] in his online inter-active book on how a neural network can compute any arbitrary function. Let's consider the most simple case, represented in Figure 2.11, namely a single input neuron, a single output neuron and a hidden layer with two neurons. The activation function in the hidden neurons is a sigmoid. The graph in the right accounts only for the effect of the top hidden neuron in the output. Initially, the parameters w (weight) and b (bias) associated with that neuron are arbitrary values, and the output still resembles a sigmoid curve. By changing those values, as shown in Figure 2.12, the output becomes almost a step-function.

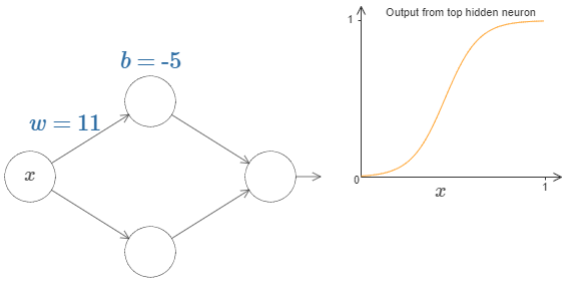

Figure 2.11: Single-input, single-output neural network with a 2-neuron hidden layer. The network output corresponds to the contribution of the top hidden neuron alone *(Source: Nielsen, 2015 [31])*.

The b parameter controls the amount of shifting in the horizontal axis, and the w parameter controls how steep the step-like function is.

[2]Feed-forward neural networks are a type of networks where there are no feedback connections. Convolutional neural networks belong to this group.

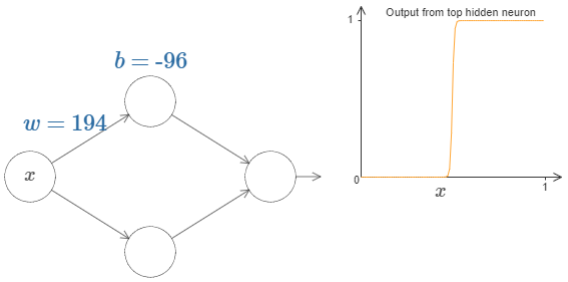

Figure 2.12: Network output re-shaped like a step-function by modifying parameters w and b *(Source: Nielsen, 2015 [31])*.

In Figure 2.13 the output reflects the combined effect of both hidden neurons. The amount of shifting for each sigmoid function has been replaced now by the bias parameters $s1$ and $s2$, and two extra weights, $w1$ and $w2$ have been added. The output is a linear combination of the activation functions of both hidden neurons, where $w1$ and $w2$ are the coefficients. These two parameters have the same magnitude but opposite sign, and this produces a rectangular function at the output. The height of this rectangular pulse is proportional to the magnitude of $w1$ and $w2$, and the difference between $s1$ and $s2$ gives the width.

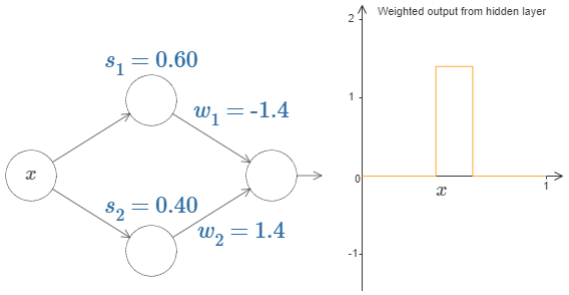

Figure 2.13: The network output corresponds to the linear combination of both hidden neuron activations *(Source: Nielsen, 2015 [31])*.

The next step is to add more pairs of neurons to the hidden layer in order to concatenate a series of rectangular pulses in the output, as shown in 2.14. Now, the values inside the circles representing each one of the hidden neurons correspond to the bias value, the before mentioned s parameter, and again they correspond to the amount of shifting given to the sigmoid function delimiting the rectangular pulses. Also, for each couple of hidden neurons, there is associated a h parameter which corresponds to the height

of the rectangular pulses.

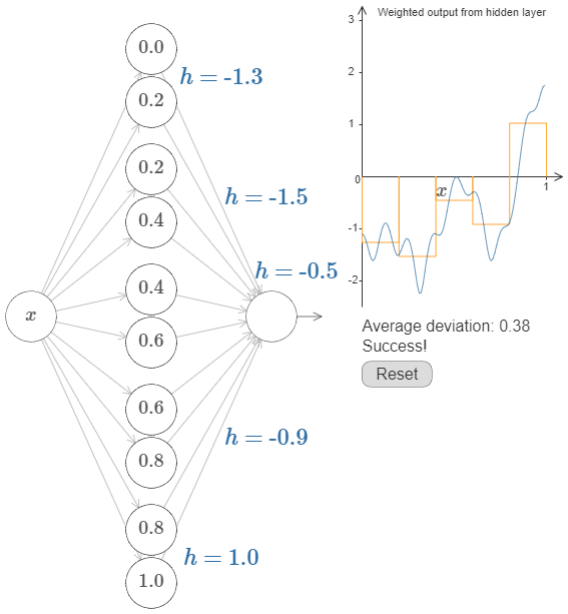

Figure 2.14: Approximation of an arbitrary function by adding additional neurons in the hidden layer *(Source: Nielsen, 2015 [31])*.

At this point, it should be clear how by adjusting these parameters it is possible to approximate any arbitrary function, like the blue curve in the output plot, to a reasonable deviation degree, and also explains why neural networks perform so well as non-linear classifiers, as mentioned in earlier sections.

2.4 Example of a Convolutional Neural Network: ZynqNet

ZynqNet [25] is a modified version of a very recent architecture known with the name of SqueezeNet [26]. The latter was released in 2016, and the most remarkable trait of this CNN is that it can obtain the same level of accuracy of other networks, but by using a factor x50 fewer parameters, which makes it very suitable for mobile applications. Although SqueezeNet is already a highly optimised architecture, ZynqNet pushes its design one step forward and provides a series of improvements that make it more convenient for FPGA implementations.

Table 2.1 is a description of the ZynqNet architecture, layer by layer,

together with their associated hyper-parameters. However, the architecture of both ZynqNet and SqueezeNet is best envisaged using Netscope [32], a web-based tool developed for visualising and analysing convolutional neural network architectures. A visual representation of ZynqNet generated by Netscope is illustrated in Figure A.1. In short, the core component of SqueezeNet/ZynqNet is the so-called *fire module*, which can be split into two layers, a *squeeze layer* followed by an *expansion layer*:

1. the *squeeze layer* consists of a 1x1 matrix convolution; although the receptive field is just 1x1, one should note that the main effect of this operation is on the third dimension, or depth, of the input volume, where the depth is equal to the number of input channels. Hence, the 1x1 convolution results in the combination of all the input channels into one, thus reducing the depth of the activation volume effectively.

2. the *expansion layer*, in reality, combines two parallel convolutional layers that use different kernel sizes (1x1 and 3x3) and concatenates both results into a single activation volume. The 1x1 convolutions are not fitted to detect spatial structures, but as mentioned above, they combine the channels in different ways. However, the 3x3 convolutions can, indeed, detect structures and patterns in the images. Thus, by combining different kernel sizes, one can obtain a much more expressive model and, at the same time, reduce the number of parameters needed.

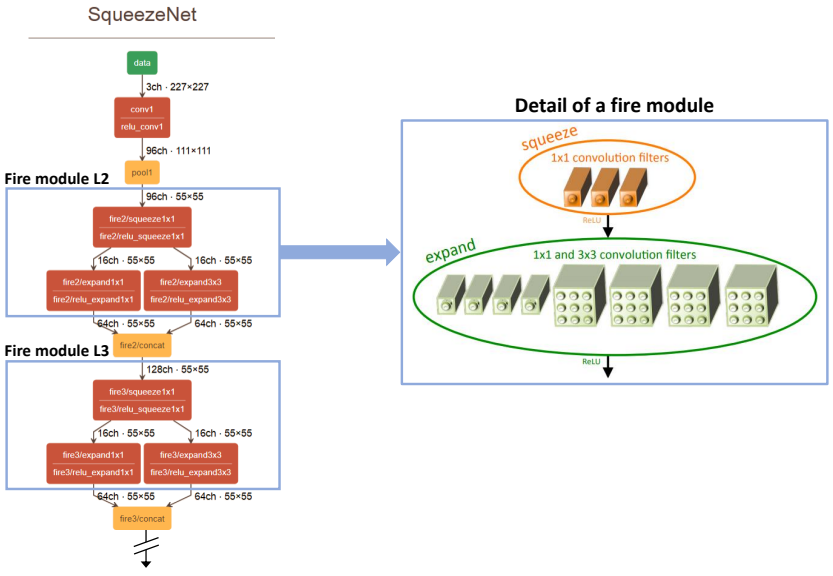

Figure 2.15: Detail of the fire module in SqueezeNet *(Source: derived from Netscope CNN Analyzer [32] and Iandola, 2016 [26])*.

SqueezeNet uses 8 of these fire modules (Figure 2.15) and a couple of convolutional layers as an input and output layer. It also completely avoids the use of fully-connected layers, which require large amounts of parameters. Instead, SqueezeNet uses four pooling layers (three *max* and one *average*) distributed along the network architecture, which does not require any weights.

By contrast, ZynqNet does some optimisation on the SqueezeNet architecture:

Layer name	Type	Kernel	Stride	CH in	WxH in	CH out	WxH out
conv1	Convolution	3x3	2	3	256x256	64	128x128
fire2/squeeze3x3	Convolution	3x3	2	64	128x128	16	64x64
fire2/expand1x1	Convolution	1x1	1	16	64x64	64	64x64
fire2/expand3x3	Convolution	3x3	1	16	64x64	64	64x64
fire3/squeeze1x1	Convolution	1x1	1	128	64x64	16	64x64
fire3/expand1x1	Convolution	1x1	1	16	64x64	64	64x64
fire3/expand3x3	Convolution	3x3	1	16	64x64	64	64x64
fire4/squeeze3x3	Convolution	3x3	2	128	64x64	32	32x32
fire4/expand1x1	Convolution	1x1	1	32	32x32	128	32x32
fire4/expand3x3	Convolution	3x3	1	32	32x32	128	32x32
fire5/squeeze1x1	Convolution	1x1	1	256	32x32	32	32x32
fire5/expand1x1	Convolution	1x1	1	32	32x32	128	32x32
fire5/expand3x3	Convolution	3x3	1	32	32x32	128	32x32
fire6/squeeze3x3	Convolution	3x3	2	256	32x32	64	16x16
fire6/expand1x1	Convolution	1x1	1	64	16x16	256	16x16
fire6/expand3x3	Convolution	3x3	1	64	16x16	256	16x16
fire7/squeeze1x1	Convolution	1x1	1	512	16x16	64	16x16
fire7/expand1x1	Convolution	1x1	1	64	16x16	192	16x16
fire7/expand3x3	Convolution	3x3	1	64	16x16	192	16x16
fire8/squeeze3x3	Convolution	3x3	2	384	16x16	112	8x8
fire8/expand1x1	Convolution	1x1	1	112	8x8	256	8x8
fire8/expand3x3	Convolution	3x3	1	112	8x8	256	8x8
fire9/squeeze1x1	Convolution	1x1	1	512	8x8	112	8x8
fire9/expand1x1	Convolution	1x1	1	112	8x8	368	8x8
fire9/expand3x3	Convolution	3x3	1	112	8x8	368	8x8
conv10/split1	Convolution	1x1	1	736	8x8	512	8x8
conv10/split2	Convolution	1x1	1	736	8x8	512	8x8
pool10	Avg. Pooling	8x8	-	1024	8x8	1024	1x1
loss	Softmax	-	-	1024	1x1	1024	1x1

Table 2.1: ZynqNet CNN architecture. Description of layers and hyper-parameters.

- SqueezeNet uses a 7x7 kernel in `conv1`, the first layer; this is typical

in many CNNs that use large kernels in the very first convolutional input layer. ZynqNet replaces the 7x7 kernel by a 3x3, reducing the number of MACC operations by a 5.4 factor while still maintaining almost the same final accuracy (it only drops by a 0.8%).

- ZynqNet sticks to the *all-convolutional-network* design principle, and does it so by replacing each one of the max pooling layers (together with its subsequent 1x1 squeeze layer) by a 3x3 convolution with a stride of 2. Although this modification increases the number of parameters and MACC operations, it results in a very consistent and unified architecture plus an increase of the final accuracy of 1.5 % [25]. Only the average pooling layer is left in the last stage of the network for classifying purposes.

- ZynqNet implements a design where all the spatial dimensions of the feature volumes are rounded to the nearest power of 2 (height, width and number of channels) which assures that multiplications or divisions, needed when accessing the feature elements from the FPGA block RAM, are done with inexpensive shift operations.

Chapter 3

Training of a Deep Neural Network

This chapter covers different aspects related to the training process of the neural network model. It gives an overview of the training framework and the hardware details of the workstation used for the training. It also explains the making of the training image dataset and the training-related basic concepts such as data augmentation, overfitting and transfer learning. Furthermore, a description of the most relevant training hyper-parameters is given, and a summary of the training results is presented.

Finally, it brings some insight about an advantageous post-training technique known as *network quantisation* that allows for further size reduction of the network and its implementation onto small embedded platforms by using dynamic fixed-point representation.

3.1 Training Framework

The NVIDIA Deep Learning GPU Training System (DIGITS) [33] is the software used for the training of the neural network model used in this thesis work. DIGITS is an open-source deep learning software for image classification; it is not a training framework as such but rather a *wrapper* for the most popular training frameworks used nowadays, namely Caffe [34], Torch [35] and TensorFlow [36]. All the latter are back-ends that integrate open-source computer vision GPU libraries (OpenCV).

DIGITS provides a graphical web-based interface instead of dealing with the command line as it is typically the case with the above-mentioned frameworks. By doing so, it facilitates to monitor and manage neural network training jobs, and analyse accuracy and loss in real time. The primary goal is to rapidly train the proposed model from large image datasets to make highly accurate image classifications. A very first step is obviously

to create a large enough dataset which is discussed in the following sections.

The computations performed by the training framework consist in the optimisation of the cost function described previously in Section 2.2 using the backpropagation algorithm together with the stochastic gradient descent (SGD). By this means, the neural network model acquires the feature hierarchies from the raw training data and gains the ability to infer real-world data. This capability is what is known as *machine learning*.

Overview of a GPU-based Training System

The training system (DIGITS) has been installed on a standard desktop computer running on a Linux distribution (CentOS). An NVIDIA GPU was added plus a new power supply, to power the GPU properly. The following is a description of the hardware that can be found in the computer used for the training of the neural network model:

Part	**Model Description**
Mother board	ASUS P8Z77-M PRO
CPU	Intel i7-2600K (3.4 GHz)
GPU	NVIDIA GeForce GTX 1080 Ti MSI Aero OC 11GB
RAM	Corsair XMS3 4x16 GB (1.3 GHz)
HDD	Western Digital WD10EZRX (1 TB, 6 GB/s)
Power Supply	Corsair Builder Series CX600 V2 (600Watt)

Table 3.1: List of the hardware installed in the computer used as training station.

As illustrated in Table 3.1, the training station makes use of one GPU only which is enough to run a training job in just a few minutes. Most likely, for a much more extensive training dataset than the one used for this project, more GPU computation power is required otherwise so that the training time can be kept at reasonable levels.

3.2 Making of a Training Dataset

As already stated in the preliminaries, the objective is that the neural network can recognise human sign language, specifically the Swedish hand alphabet used for *fingerspelling*, which is used in sign language to spell out names of people and places for which there is not a sign. The Swedish hand alphabet is illustrated in Figure 3.1. It is composed of 29 different hand poses, of which three of them correspond to the distinctive vowels *å*,

ä and *ö*. It should be noted that these vowels are accompanied by a whole hand movement in the direction indicated by the arrows in the picture. For that reason, they have been excluded from the training as they are not static hand positions. Hence, it is 26 signs that the CNN should be able to recognise.

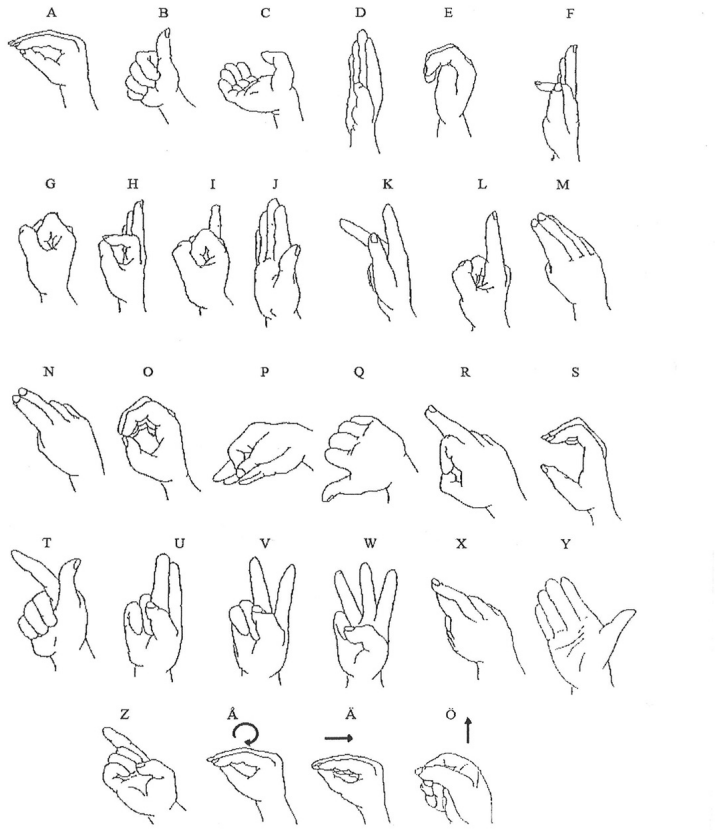

Figure 3.1: Swedish hand alphabet *(Source: Ene, 2015 [37]).*

In order to have a better algorithm that can take better decisions, it should learn from data that reflects the variety of the real world. For that reason, the larger the training dataset, the better. But not only quantity matters: a dataset with a million different cars would be useless if all of them were facing to the left in the picture. Therefore, the dataset should provide different points of view of the same object, and this is achieved by using *data augmentation* techniques.

The images used for training must be varied and also hard to learn. For this particular case, they must reflect the different traits one can find among

the hands in the general population like skin colour, bone structure, finger size and so on. It should be pointed out that the dataset has been created from scratch since a dataset with such specificity and characteristics did not exist previously. Thousands of pictures have been taken among more than 50 volunteers who did not have any previous knowledge on Swedish sign language.

The method used for that purpose was either to record or to take pictures from the hands of the volunteers while they were trying to mimic the 26 positions presented to them on a screen, where a video for learning fingerspelling was being reproduced. In one hand, it can be expected that the quality of the signs is not the best, in terms of precise positioning of the hand and fingers, as most of the gestures were performed for the very first time by the volunteers, but on the other hand, this fact adds enough variance and error tolerance to the training.

Figure 3.2 is a small example of some of the pictures taken in several sessions and locations, with different backgrounds and lighting conditions.

Figure 3.2: Representation of the Swedish sign alphabet by using sample images used for the training.

Overfitting and Underfitting

Two commonly frequent problems to be avoided as much as possible during the training of a neural network model are *overfitting* and *underfitting*.

Overfitting occurs when the model learns the dataset too well, either because there is not enough variety in the dataset or it is not large enough.

It can also be because the network is too complex or deep, and as a result of this, the model is not only capable of extracting the critical features to learn but also the *noise*, that is, the not relevant features. In other words, the model is capable of *memorising* the examples but it cannot gain the ability to infer a general rule, and it is not able to interpret anything else that is not close to the learnt data.

The opposite extreme is *underfitting*. The network model may be too simple to extract the right patterns from the training data and, as a consequence, some of the significant features are considered as noise and discarded as such.

Data Augmentation

Very often, it is not realistic or feasible to build up a large enough dataset because of a lack of resources, that is, enough number of volunteers and the time required to validate thousands of pictures before adding them in the dataset. In order to increase variance to improve the training accuracy, one can opt for *data augmentation* techniques to increase the size of the dataset. Some of the most common transformations that can be done to an already existent picture are horizontal or vertical flips, translations in any direction, image scaling using an arbitrary factor, arbitrary cropping and image rotations, both clockwise and counter-clockwise. All the ones mentioned here have been used in this project with the result of increasing the initial size of the dataset by a factor 5. Regarding rotations, the maximum applied angle is ± 20 degrees, to preserve the integrity of the symbol encoded by the hand sign. Other tricks worthy to consider with the purpose to make training really hard, but may require a more sophisticated image edition software, are: masking out random square regions of the images (cutout), the addition of Gaussian noise (with zero mean), modify image colours by arbitrarily adjusting the hue values (this one is recommended in order to mimic pictures taken with different illumination conditions) or applying filters to blur the images.

An example of possible image manipulations on the same picture is illustrated in Figure 3.3. Combining all of these techniques have the potential to largely increase an initial dataset with a modest size.

Additionally, it can be seen in the pictures that the background behind the hands is another factor to have into account. Some pictures were purposely taken by positioning the hand against a white wall while other pictures have been taken with the hand positioned in front of the volunteer's torso, adding to the background the shapes and colour patterns from the person's clothes. Moreover, in other pictures, the background contains many other peculiar objects like windows, wall signs, light switches and other

typical indoors elements. All these features make those pictures perfect candidates for the training dataset as they are adding *noise* that the model should learn to discard, while the ones with a blank background are preferred for the test dataset.

Figure 3.3: Data augmentation techniques applied on a same picture.

Training, Validation and Test Dataset

At this point, it is convenient to clarify a relevant point regarding the image dataset. Actually, not all the pictures taken from the volunteers are meant for the training. They have been organised in three different subsets, by using the following standard:

- First, there is the *training* set, which is the portion of images intended to adjust the weights on the neural network.
- Then, there is a smaller subset known as *validation* set, which is primarily used to minimise the overfitting. The weights are not being adjusted with this data set, but they are just used to verify that any increase in accuracy over the training dataset turns out into an increase in accuracy over a dataset that has not been shown to the network before, that is, the validation dataset itself. If, on the contrary, the accuracy over the training dataset increases, but the accuracy over the validation dataset stays the same or decreases, then the network is overfitting, and the training should be stopped and introduce the proper modifications.

- And finally, another small portion is used as the *test* dataset, which is the data used only for testing the obtained model, to confirm the actual inference accuracy of the network. In particular, it is the dataset used to test the neural network after training and once it has been deployed on the FPGA.

Typically, the percentages used to split the entire dataset with these partitions are 60-20-20, in the same order as listed above, but these numbers are not strict and can vary substantially.

3.3 ImageNet and Transfer Learning

Another good reason to consider the use of the ZynqNet model for this project lies in the fact that it has been previously trained on the ImageNet dataset. The ImageNet project holds a popular image recognition contest since 2010 [38] where neural networks get tested to see which model obtains the better accuracy and has become *de facto* a standard benchmark for image classification algorithms.

The challenge training dataset is made up of 1.2 million images, each belonging to one of the 1,000 possible categories that cover a wide variety of objects, animals and scenes. In the first year of the competition, every team got at least 25% wrong. However, in 2017, about 75% of the competing teams got an error rate below 5% which is considered to be at human error level.

The notable advantage of having ZynqNet already trained with ImageNet is that the model is already able to detect general image features, such as edges or shapes. These fundamental skills can be transferred to a new recognition task, by modifying the existent layers and retraining the model on a new dataset. The latter is a very well known technique used in machine learning known as *transfer learning*. The benefits of this approach are far-reaching, as the new model is taking advantage of the expensive resources used in the pretrained model such as hardware, image datasets or training time, to acquire new recognition capabilities. For instance, it can take two or three weeks to train a deep neural network model on ImageNet even by using multiple GPUs.

In other words, transfer learning speeds up the training process by reusing an already trained model. Furthermore, it eliminates the need to invest large amounts of time and resources in the creation of a new and extensive image dataset for training purposes, but just the necessary to create a modest one, as it is the case for the dataset built for the present project. The ZynqNet pretrained model can be found online and publicly available at the author's Git repository [39]. The file with the pretrained

weights is used as the starting point for the backpropagation algorithm to fine-tune the model for the hand sign recognition task.

A common strategy used to implement transfer learning is, first, to replace the network classifier previously trained with ImageNet by another classifier that matches the number of classes in the new dataset and then, to retrain this new classifier with randomly initialised weights. Alternatively, it can be advisable to fine-tune the rest of weights from the convolutional layers or, keep the weight values from some of the earlier layers and fine-tune only the layers closer to the classifier which are responsible for extracting the high-level features from the dataset. This is based in the fact that low-level features extracted in earlier layers of the network contain more generic features, like edges, shapes or colours, that have the potential to be useful in many other recognition tasks. However, by moving forward through the layers, the features become progressively more and more specific to the details of the classes contained in the original dataset; thus those layers are more prone to be fine-tuned for the new task.

3.4 Training Hyper-Parameters

Several are the hyper-parameters one can adjust to explore the way of getting better training results before deploying the model. The basic approach is to run different training jobs where one parameter is changed each time. Slight changes in the parameter's value can have a significant impact on the model's training accuracy, so it is best to analyse the effect on each parameter, one at a time. This strategy derives in the run of several jobs where DIGITS provides a good tracking system of the obtained results and the history of the changes.

Maximum Number of Epochs

The maximum number of epochs to use before stopping the training. An epoch is the full pass of the backpropagation algorithm over the entire training dataset. Some frameworks give the option to automatically stop the training if the accuracy of the network reaches a plateau and it is not improving after a certain number of epochs.

Batch Size

Batch or mini-batch size is a subset of images from the training dataset that is used to evaluate the gradient of the loss function and update the weights (see Section 2.2). Optionally, one can choose to shuffle all the training images before each training epoch and shuffle all the validation

images before each network validation. In this way, the batches are never the same through the different epochs.

The batch size is proportional to how accurate is the calculation of the gradient at each iteration of the backpropagation algorithm. A batch size equal to the size of the training dataset gives the exact gradient or direction vector in the search for the function global minimum, but it takes a lot of computation time. In the other extreme, a batch size of one (with only one image), is very fast to compute but it returns a gradient value very inaccurate or noisy, not representative of the entire training set. Hence, it is a trade-off between gradient accuracy and computation time. On the other hand, noisy or inaccurate gradient values can also be desirable as it causes oscillations that can help the algorithm not to get stuck in local minima or saddle points (see Figure 2.10).

Learning Rate and Drop Factor

The learning rate affects the size of the steps taken in the direction of the estimated gradient in order to minimise the network's loss function. This parameter scales the magnitude of the weight updates in each iteration of training thus it determines how fast weights change. Typically, the learning rate is a small value ranging between 0.01 and 0.0001. Given the fact that the learning rate multiplies the computed gradient, it is a crucial parameter which, if not properly chosen, it can make the network either fail to train or take much longer to converge.

The learning can be kept constant, which is not recommendable, or it can be decreased accordingly after each epoch by a certain amount defined by the so-called *drop factor*. The decreasing rate or decay of the learning rate can be expressed either by a constant or by an exponential function.

When using the transfer learning methodology to train a network on a new task, it is common to use a smaller learning rate for the weights associated to the convolutional layers, which are considered as the feature extractor part of the network. In comparison, the learning rate used for the output layer that contains the classifier needs a larger learning rate. The reason for that is because the convolutional layers of the pretrained network have already learned features that can be similar to the ones of interest in the new dataset, so the weights are expected to be close to the optimal values and there is no need to distort them too much.

Momentum

Momentum is a parameter that helps to accelerate the backpropagation algorithm in the relevant direction. By providing momentum to the algorithm,

the current step is influenced by past actions, as some proportion of the last step is added to the current one, endowing the algorithm with a sort of memory capability. That helps the training algorithm of not being trapped in local minima or saddle points, improving the rate of convergence. Put in other words, adding momentum is a technique that creates an adaptive learning rate that varies for different weights.

In summary, the above described are some of the most relevant hyper-parameters one is free to change during a training session. The downside of having so many degrees of freedom is that there is no single method valid for optimising a neural network's accuracy, but one should rely more upon its own experience and heuristic techniques, and go through a lot of trial and error to find good values.

3.5 Post-Training Network Quantisation

There exist several compression techniques that aim to reduce the size and resource utilisation of neural networks. Research in this field continues nowadays by delivering new methods to widespread the use of deep neural networks, where the main obstacle for this technology is the vast amounts of memory required to store the weight parameters needed for classifying a single image (which can be in the order of hundreds of megabytes). Some of the most popular techniques are *network quantisation* [40], *weight pruning* [41], *regularisation* [42] and *knowledge distillation* [43]. These methods are typically applied at different design stages or can be applied more effectively combined, as suggested by Tung and Mori [44], who propose a deep network compression algorithm that performs weight pruning and quantisation jointly, and in parallel with network fine-tuning.

For the present work, it has been considered that one of the more hardware-effective techniques to apply after training the model would be the network quantisation. It refers to the reduction of the required number of bits to represent a quantity. This reduction is justified by the fact that the training process is performed in the DIGITS environment using a 32-bit floating-point format, and the learned weights and output activations are expressed accordingly.

Hence, the main benefit of applying quantisation is an important reduction in storage size and memory access. For instance, using 8-bit integers for weights and activations consumes four times less overall memory access compare to the use of 32-bit floating-point numbers. Furthermore, it is also much more efficient in terms of area and energy; according to Dally [45], significant savings in both figures can be obtained, as listed in Table 3.2:

INT8 Operation	Energy Saving vs. FP32	Area Saving vs. FP32
Addition	30x	116x
Multiplication	18.5x	27x

Table 3.2: Relative energy and area saving factors by comparing INT8 with FP32 operations.

3.5.1 CNN Quantisation with Ristretto

The tool of choice used for the quantisation of the network is *Ristretto*, developed by Gysel [46] which, as described by the author, is an automated CNN-approximation tool to condense 32-bit floating-point networks. Ristretto takes a trained model as input and automatically extracts a condensed network version. The output files obtained after the process contain a standard description file of the network (a file with the extension `.prototxt`) and the quantised network parameters.

Here is worth to note that quantisation, when used to directly approximate a model without retraining, this method is commonly referred to as *post-training quantisation*. Instead, if retraining is performed after quantisation, which is highly recommendable to make up for the loss of accuracy produced by the approximation process, this is referred to as *network fine-tuning*. Ristretto gives the option to automatically fine-tune the quantised network which can further improve the final accuracy indeed.

The quantisation flow proposed by Ristretto is shown in Figure 3.4.

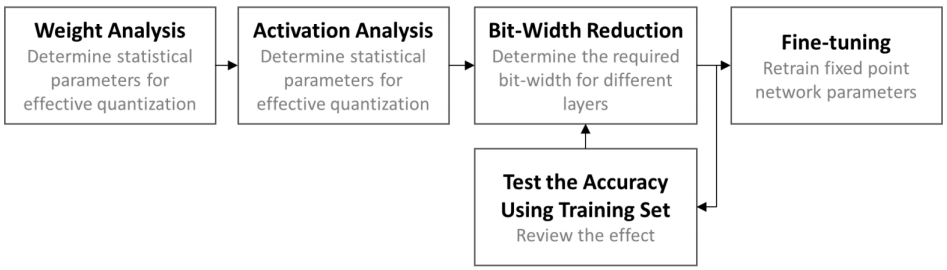

Figure 3.4: Ristretto's network approximation flow to compress a floating-point network into fixed-point *(Source: Gysel, 2016 [46])*.

As illustrated in the flow, during the first and second stages the dynamic range of both weights and activations is analysed using statistical methods

to find the proper fixed-point representation. The integer part of the fixed-point numbers used to represent the activations are set aside with enough bits to avoid saturation due to large values.

In the third stage, Ristretto performs a binary search to find the optimal number of bits for the weights and layer activations. In deep neural network models, the dynamic range of weights and activations are usually very inconsistent and can differ between layers in the model. For that reason, at this stage, firstly the weights are quantised, while the activations remain in floating-point, and then an accuracy test using the training dataset is performed in order to review the effect.

Then, another iteration follows but inverting the roles, that is, the activations get quantised while the weights remain in floating-point and then the effect is reviewed. Iteratively repeating the described process allows Ristretto to find the optimal bit-width for weights and activations. Once a good trade-off between fixed-point representation and classification accuracy is found, the resulting network can be retrained, or in other words, fine-tuned.

The fine-tuning is necessary to make up for the accuracy drop incurred by quantisation. During this retraining procedure, the network learns how to classify images with the new acquired fixed-point parameters. Ristretto does not quantise layer activations to fixed-point during the fine-tuning process, but uses floating-point activations instead, to enable Ristretto to analytically compute the error gradient with respect to each parameter.

3.6 Training Results

Several training experiments have been run within the DIGITS framework exploring the way up to an optimal accuracy. It is worth to mention that, typically, the graphs used to visualise the training progress show both the training and the validation accuracy plus the evolution of the loss function (see Figure 3.5). As mentioned in earlier sections, the validation set is not used to adjust the network weights but to verify that the model is not overfitting. Therefore, the validation accuracy is the figure of merit in the results being summarised here.

The size of the validation dataset has been kept constant to 163 images during the different training jobs. The criteria used for the selection of the images has been to have validation images that differ as much as possible from the ones used in the training dataset, thus making the validation hard to pass for the model.

The summary of the most significant results from the different experiments is summarised in Table 3.3. In short, each of the performed

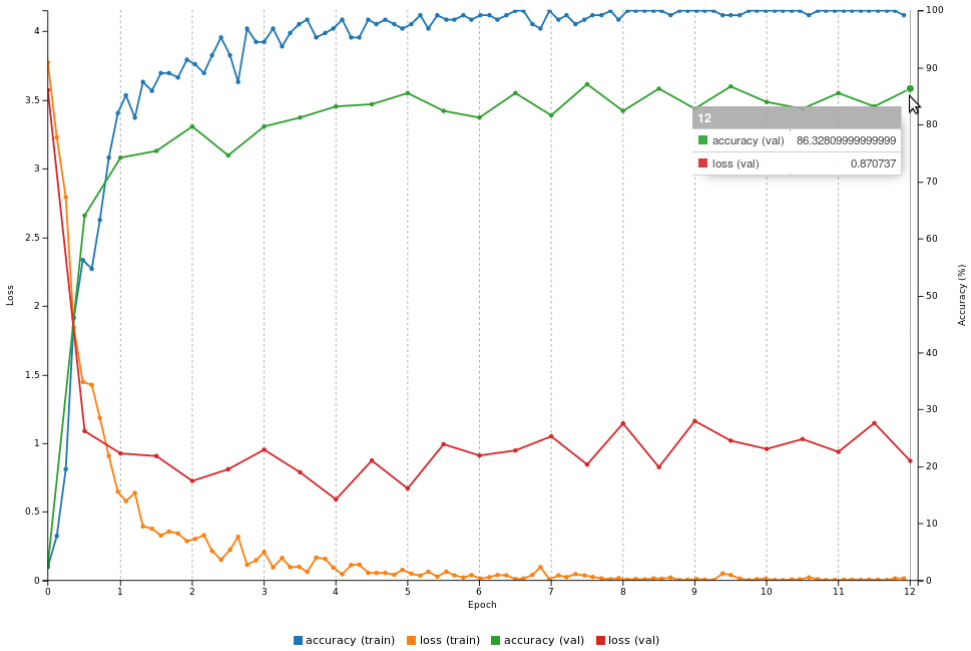

Figure 3.5: DIGITS. Graph showing the progression of the training along the different epochs. The quantities of interest are the training accuracy, the validation accuracy and the value of the loss function for both the training and the validation data.

experiments on the list has aimed to review the effect of one of the following actions:

- an increase in the size of the training dataset;
- training only the classifier layer;
- training the input layer plus the classifier;
- training all the model layers.

Effect of Dataset Size

The results clearly show how data augmentation and increasing the size of the training dataset have a direct impact on the improvement of the validation accuracy. By looking at the accuracy results obtained in jobs 1 and 2, where only the classifier is being trained, one can see that the accuracy can be slightly improved from 63.5% to 66.4% by increasing the dataset size a significant 56%, but a further increase of the dataset, as

described in job 4, does not translate into better accuracy, thus reaching a limit.

Effect of Transfer Learning Methodology

The results of the experiments, where only the classifier has been trained, show the beneficial impact of the transfer learning methodology. For instance, the validation accuracy obtained for job 1 is quite remarkable, with a value of 63.5% compared to the 63.0% top-1 validation accuracy obtained for ZynqNet in the 1000-class ImageNet challenge (Gschwend, 2016 [25]), considering the modest size of the training dataset (6,500 images) and that only the weights of the classifier have been retrained.

Job description	Training dataset size	After augmentation	Validation accuracy
1. Training classifier only	1,500	6,500	63.5%
2. Training classifier only	2,160	10,183	66.4%
3. Training conv1 + classifier	2,160	10,183	71.9%
4. Training classifier only	2,668	13,743	66.4%
5. Training conv1 + classifier	2,668	13,743	76.6%
6. Training all layers	2,668	13,743	86.3%

Table 3.3: Summary of the top-1 validation accuracy results obtained for different training jobs.

The explanation lies in the fact that the classifier performs its high-level reasoning from the non-linear combination of the features extracted from the image dataset. That is, it holds composite and aggregated information obtained from all the preceding convolutional layers, which gives the capacity of generalisation to the neural network.

Obviously, to further improve the accuracy, it is required to obtain better high-level features from the training dataset, that are more specific to the set of hand signs in question. Therefore, additional layers are added to the training, as described for jobs 3 and 5, where the first convolutional layer `conv1` is trained jointly with the classifier, resulting in a slight improvement of the accuracy.

Finally, the best result is achieved by fine-tuning all the convolutional layers together with the classifier, where a final validation accuracy of 86.3% is achieved as described in job 6.

Post-Quantisation Results

The quantisation analysis performed by Ristretto considers fixed-point representations with different bit-widths which results in a different accuracy drop respect to the floating-point model for each case. The proposed bit-widths for the quantised model are 16, 8 and 4 bits, being 8 bits the bit-width that offers the best trade-off.

	Fractional Length		
Layer name	**CH in**	**Weights&Bias**	**CH out**
conv1	0	7	-2
fire2/squeeze3x3	-2	6	-4
fire2/expand1x1	-4	6	-4
fire2/expand3x3	-4	7	-4
fire3/squeeze1x1	-4	6	-6
fire3/expand1x1	-6	7	-5
fire3/expand3x3	-6	7	-5
fire4/squeeze3x3	-5	8	-7
fire4/expand1x1	-7	7	-6
fire4/expand3x3	-7	8	-6
fire5/squeeze1x1	-6	7	-7
fire5/expand1x1	-7	7	-6
fire5/expand3x3	-7	7	-6
fire6/squeeze3x3	-6	8	-7
fire6/expand1x1	-7	7	-6
fire6/expand3x3	-7	8	-6
fire7/squeeze1x1	-6	7	-7
fire7/expand1x1	-7	8	-5
fire7/expand3x3	-7	8	-5
fire8/squeeze3x3	-5	8	-6
fire8/expand1x1	-6	8	-4
fire8/expand3x3	-6	8	-4
fire9/squeeze1x1	-4	8	-4
fire9/expand1x1	-4	8	-2
fire9/expand3x3	-4	9	-2
conv10	-2	9	-1

Table 3.4: List of the fractional lengths estimated by Ristretto for an 8-bit fixed-point quantisation of the model. Values are given layer by layer, for the input and output activations, weights and bias.

The accuracy drop estimated by Ristretto for this case is only 2.3% respect to the final accuracy achieved by DIGITS of 86.3%. It can be seen that, after fine-tuning the quantised model, this accuracy drop is not only recovered but overturned, increasing the final accuracy of the model up to 87.1%. Table 3.5 summarises the evolution of the model accuracy along the different training phases.

Training Phase	**Network Accuracy**
DIGITS (dataset with 13,743 images)	86.3%
Quantisation to 8-bits (Ristretto)	84.0%
After fine-tuning	87.1%

Table 3.5: Evolution of the model validation accuracy along the different training stages. Quantisation and fine-tuning are performed using an 8-bit fixed-point format.

The quantised network obtained with Ristretto is summarised in Table 3.4, where the fractional length for the input activations, weights and output activations are listed layer by layer.

By using dynamic fixed-point, different numerical representations can be used for the layer activations and weights. It can be observed that layer activations (denoted as *in/out channels* in the table) can be relatively big compared to the weights. The latter requires more bits in the fractional part in order to represent such tiny numbers.

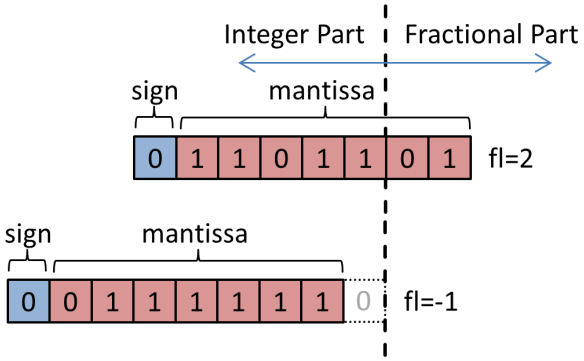

Figure 3.6: Example of 8-bit dynamic fixed-point numbers. A number with a negative fractional length of -1, means that its integer length is 9, although its bit-width is 8 *(Source: derived from Gysel, 2018 [47]).*

On the contrary, activations can have values so large that do not fit on an 8-bit representation and need to be compressed. That is why its fractional length is negative, which denotes that the mantissa does not have fractional part and the integer part needs to be extended, by padding the right side of the mantissa using as many zeroes as defined by the negative fractional length (see example illustrated in Figure 3.6). The hardware accelerator has to take care of this *zero-extension* operation.

Chapter 4

Design and Implementation of a CNN Hardware Accelerator

As already mentioned in the preliminaries, the present work is based on ZynqNet, an FPGA-accelerated embedded CNN, designed by Gschwend [25]. This network architecture is already highly optimised for its implementation on an FPGA device. It also has the advantage that it has been trained for image recognition tasks and one can use the transfer learning approach to retrain the neural network and use it for another image recognition task such as recognition of human sign language, as described in the previous chapter. Furthermore, the trained model is publicly available online in the author's Git repository [39].

This chapter is covering the different aspects related to the design of a CNN hardware accelerator and its RTL implementation on a Xilinx FPGA board. First of all, the project goals are presented, together with the design requirements and the design strategy proposed for a successful implementation. It continues with the description of the proposed C-based model and the behavioural description of the hardware, module by module, with a focus on the implemented optimisations.

Finally, the chapter concludes with a picture of the applied verification methodology, both pre-synthesis a post-synthesis plus a summary of some of the significant problems encountered during the design process.

4.1 Project Goals and System Requirements

The goal of the present thesis can be summarised in the following statement:

"Train Zynqnet for human sign language recognition, and fit the whole neural network onto an FPGA chip achieving both proper inference accuracy

and throughput, given by images or frames per second (FPS)."

On the pursuit of this objective, the proposed design aims to provide proof of concept of the transfer learning principle and brings the opportunity to improve the ZynqNet architecture. The improvements are brought on two major aspects:

1. to avoid external memory access, taking out of the equation the impact in the performance due to external memory bandwidth limitations. All the weights, bias, input image and intermediate feature maps should fit in the on-chip memory of the FPGA.

2. the quantisation of the CNN network; in contrast with the 32-bit floating-point format used in the original ZynqNet, the proposed design will use 8-bit dynamic fixed-point for representing all the weights and bias, input images and partial feature maps obtained after each convolution operation.

As might be expected, the algorithmic description of the Zynqnet network must be written from scratch and has to be adapted to the target goals, ensuring the originality of the design presented in this work. The High-Level Synthesis design process is the method of choice for that purpose, given the complexity and the size of the project.

4.1.1 Memory Requirements

This section provides an estimation of the memory requirements to take into consideration when choosing the proper FPGA to fit in the neural network. It must have enough block RAM to allocate the necessary data at any time during the convolution operations. These data consist of:

1. all the weights and bias generated during the training of the neural network and that are loaded into the FPGA. The number of weights per layer is calculated as follows:

   $$\#of weights = input\_channels \cdot output\_channels \cdot kernel\_size$$

   where the kernel size can be either 3x3 or 1x1. Summing the weights for all the layers gives a total of 1,794,294 weight elements. The number of bias required per layer is just equal to the number of output channels for that layer. That is a total of 3,456 bias elements.

2. enough cache memory to store the partial results after each convolution operation. The size for that cache will be given by the largest feature volume produced after a convolution, and that corresponds to the

convolution `conv1` (see Table 4.2), which produces an output volume of 64x128x128 elements. The design will implement two of these memory caches, one for the input volumes and another one to store the output volumes produced by each convolution operation.

Item	Number of bits		Mb
Weights	$1,794,294 \cdot 8$	=	13.7
Bias	$3,456 \cdot 8$	=	0.03
Feature Map Cache 1	$64 \cdot 128 \cdot 128 \cdot 8$	=	8
Feature Map Cache 2	$64 \cdot 128 \cdot 128 \cdot 8$	=	8
	TOTAL		29.73

Table 4.1: Estimated memory requirements expressed in megabits.

Table 4.1 summarises the estimated memory requirements for the project, which turns out to be a little bit less than 30 Mb.

FPGA: Choosing the Right Device for the Project

When considering which FPGA would be the right choice for the present project, mainly two reasons were taken into account:

1. a device with enough amount of resources adequate to contain the whole CNN network, including all the necessary weights and bias and, most importantly, enough on-chip memory to allocate the partial feature maps produced after each convolution operation. The main idea here is to avoid the access to external memory altogether, and to use only the available memory on the FPGA chip, thus exploring the possibilities for future small mobile applications. An additional advantage to this approach is also a reduced system latency, in compliance with the project goals stated above.

2. device readiness; reuse one of the many FPGA boards already available at the university department, provided that it fits the given specification.

Both requirements meet in the Xilinx Board ADM-PCIE-KU3 [48] which incorporates a Xilinx FPGA from the Kintex UltraScale family, device model XCKU060 [49] and package version FFVA1156. This board uses Xilinx PCI Express technology (PCIe), so it can be installed via PCI connector into servers or workstations for its use in communications or data-centre

applications, where the host CPU can be used for additional computations as well as the host system memory, that is accessed through featured DMA IPs. The following are some of the highlights available in this FPGA:

- Maximum frequency of a global clock tree (BUFG): 725 MHz
- Maximum Distributed RAM (LUTs): 9.1 Mb
- Block RAM: 2160 blocks of 18 Kb each (38 Mb)
- DSP48E (multipliers): 2760 units

At first glance, it can be seen that the number of block RAM available on this board is sufficient to cope with the memory requirements estimated in the previous section. Moreover, there are available large amounts of multipliers and distributed memory (registers) which facilitate the parallelisation opportunities in the design. It also provides enough room for additional blocks or more complex control units such as a MicroBlaze microprocessor that could be added afterwards for further system development purposes (see final conclusions in Chapter 5).

4.2 Adapting the ZynqNet Topology

Table 4.2 is a description of a slightly modified version of the ZynqNet topology, adapted to the project specifications. The modifications, compared to the original ZynqNet architecture described in Table 2.1, consist of the following:

1. Undo the split of the `conv10` layer and change the number of output channels to 32. Since `conv10` is the last convolutional layer, right before the average pooling operation, the number of outputs should be equal or greater to the number of categories that the neural network can recognise, in this case, 26 different hand gestures used for Swedish fingerspelling, hence 26 outputs that are extended to 32, as they are rounded to the nearest power of 2.

   Because ZynqNet was trained with the ImageNet dataset, the last convolutional layer required 1,000 output channels, which corresponds to the number of possible categories in which the images in the ImageNet dataset are classified. The amount of memory required to perform this operation exceeded the on-chip memory capabilities of the original ZynqNet design; therefore it was split into two consecutive convolutions with 512 output channels and the results concatenated afterwards. In the present modified ZynqNet version this is not

necessary due to the smaller amount of output channels and a smaller bit-width used for number representations (8-bit dynamic fixed-point).

Layer name	Type	Kernel	Stride	CH in	WxH in	CH out	WxH out
conv1	Convolution	3x3	2	3	256x256	64	128x128
fire2/squeeze3x3	Convolution	3x3	2	64	128x128	16	64x64
fire2/expand1x1	Convolution	1x1	1	16	64x64	64	64x64
fire2/expand3x3	Convolution	3x3	1	16	64x64	64	64x64
fire3/squeeze1x1	Convolution	1x1	1	128	64x64	16	64x64
fire3/expand1x1	Convolution	1x1	1	16	64x64	64	64x64
fire3/expand3x3	Convolution	3x3	1	16	64x64	64	64x64
fire4/squeeze3x3	Convolution	3x3	2	128	64x64	32	32x32
fire4/expand1x1	Convolution	1x1	1	32	32x32	128	32x32
fire4/expand3x3	Convolution	3x3	1	32	32x32	128	32x32
fire5/squeeze1x1	Convolution	1x1	1	256	32x32	32	32x32
fire5/expand1x1	Convolution	1x1	1	32	32x32	128	32x32
fire5/expand3x3	Convolution	3x3	1	32	32x32	128	32x32
fire6/squeeze3x3	Convolution	3x3	2	256	32x32	64	16x16
fire6/expand1x1	Convolution	1x1	1	64	16x16	256	16x16
fire6/expand3x3	Convolution	3x3	1	64	16x16	256	16x16
fire7/squeeze1x1	Convolution	1x1	1	512	16x16	64	16x16
fire7/expand1x1	Convolution	1x1	1	64	16x16	192	16x16
fire7/expand3x3	Convolution	3x3	1	64	16x16	192	16x16
fire8/squeeze3x3	Convolution	3x3	2	384	16x16	112	8x8
fire8/expand1x1	Convolution	1x1	1	112	8x8	256	8x8
fire8/expand3x3	Convolution	3x3	1	112	8x8	256	8x8
fire9/squeeze1x1	Convolution	1x1	1	512	8x8	112	8x8
fire9/expand1x1	Convolution	1x1	1	112	8x8	368	8x8
fire9/expand3x3	Convolution	3x3	1	112	8x8	368	8x8
conv10	Convolution	1x1	1	736	8x8	32	8x8
pool10	Avg. Pooling	8x8	-	32	8x8	32	1x1

Table 4.2: Modified architecture of ZynqNet CNN for classification of 32 categories (each category is a letter of the alphabet).

2. Change the number of input/output channels to 32 in the `pool10` layer for the same reason stated above. The pooling layer calculates the average value for each one of the 8x8 input channels and the output channel with the highest average value corresponds to the category inferred by the neural network.

3. Remove the `softmax` layer. The `softmax` function (or normalised exponential function) is often used in the final layer of a neural network and gives the distribution of probabilities over each one of

the possible categories. This probability distribution turns to be very useful in the ImageNet challenge, where one of the essential metrics to understand how good is the performance of the neural network is the *top-5 error*, which is the fraction of test images for which the correct category is not among the five categories considered most probable by the model.

4.3 Design Strategy

In a first approach, a plain, straightforward design, aiming at functionality, not throughput, was developed. Being also the first time contact with the HLS toolchain, the primary purpose was to keep things as simple as possible, while the learning curve was getting steeper. In that sense, the design lacked any of the primary HLS compiler directives that allow for the optimisation of performance, like the partitioning of large arrays to speed up the access to its elements, the unrolling of `for` loops to increase parallelisation or the pipelining of functions. However, a successful implementation was obtained with minimum utilisation of the FPGA resources and with an inference accuracy slightly over the 80%, proving the feasibility of the project. The performance of the circuit was its weak side, as expected, with a latency of about 6 seconds to process one image at a clock frequency of 100 MHz.

Then, a second design aiming to improve the performance was carried out, but this time incorporating strategies to accelerate the algorithm performance, such as image *block processing* and *array partitioning* of the cache memories listed in Table 4.1, which boosted the throughput of the neural network to 6.3 FPS at a clock frequency of 250 MHz. Both approaches are described in the following subsections.

4.3.1 Block Processing

Rather than processing the whole input image at once, the algorithm splits the image channels into tiles or 2-D *blocks* of pixels. Then, those input blocks are processed one by one producing the corresponding output blocks after a certain amount of clock cycles. The data blocks are read and stored from and into the memory caches at the right location under the action of a memory controller that handles the memory offsets that need to be applied at every moment.

The size of these blocks has been decided upon the channel dimensions provided by Table 4.2, and corresponds to a block size of 8x8. As it can be seen in the table, all the layer activations in Zynqnet are formed by channels whose dimensions are multiples of this basic 8x8 block.

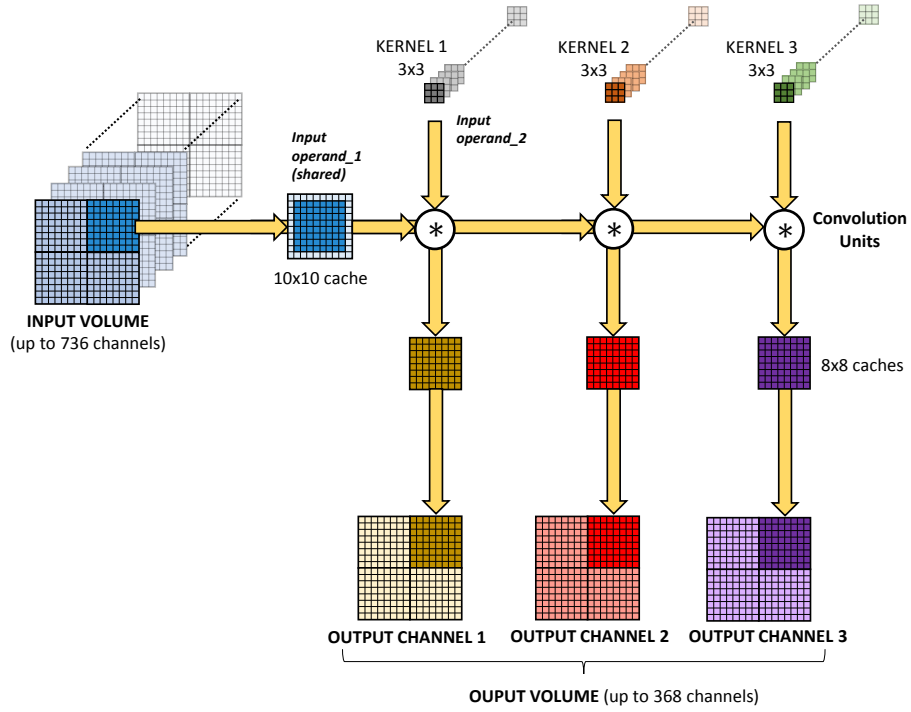

Figure 4.1: Dataflow representation of the block processing approach.

Figure 4.1 illustrates the concept of *block processing*. The input volume is formed by an arbitrary number of channels, where a channel is a two-dimensional square array of real numbers, and each array is split in an integer number of blocks[1]. The algorithm goes throughout the whole input volume, channel by channel, one block at a time. Each 8x8 input block is loaded in a small 10x10 cache[2] and then used by the convolution unit as one of the operands, being the kernel corresponding to that input channel the other operand. The result of the convolution is stored in another 8x8 cache before being pushed by the memory controller to its proper location in the larger feature map cache (see also Figure 4.2 for more detail).

[1]N.B. that the above-mentioned quantities, that is to say, the number of input and output channels and the number of blocks per channel, are powers of 2. The latter is a good design practice to apply to FPGA implementations whenever possible.

[2]This cache is oversized purposely in order to apply the padding before computing the convolution and keep the original dimensions all along the different operations.

4.3.2 Array Partitioning

The way the data is accessed through the memory caches is undoubtedly going to affect the performance. The caches that are susceptible of partitioning are: 1) the cache that contains all the weights used by the neural network, and 2) the caches that store the partial results of the convolutions or activations.

The goal is to read the operands required by the convolution units as fast as possible, namely the 8x8 data blocks from the input volume and the nine elements that form the 3x3 kernels from the weight cache (or one element for the 1x1 kernels). A simple way to achieve this goal is to define these caches as two-dimensional arrays and then to split completely one of the dimensions into distinct memory blocks that can be accessed concurrently. In the case of the weights, it follows that the dimension to be partitioned must be equal to the number of elements that form the 3x3 kernel, allowing the reading of all the nine elements in the same clock cycle. Table 4.3 lists the size of the dimensions of the arrays as they have been declared in the HLS source code, where the second dimension of each array has been completely partitioned.

Cache	**Number of Elements**	**Dim 1**	**Dim 2**
Weights	$1,794,294$	199,366	9
Feature Maps 1	$64 \cdot 128 \cdot 128 = 1,048,576$	131,072	8
Feature Maps 2	$64 \cdot 128 \cdot 128 = 1,048,576$	131,072	8

Table 4.3: Cache sizes declared as two dimensional arrays.

Additionally, the memory caches have been implemented using dual-port block RAM, allowing for the access of two elements in the same cycle. Hence, for instance, reading all the 64 elements from one block would take only 4 cycles.

4.4 Model Operation and Hardware Description

The neural network model is best described with the pseudo-code listed in Algorithm 1, but a cautionary note should be made here since the pseudo-code is just an approximated description of the circuit operation and it ignores the cases where the kernel size is 1x1 or the convolution stride is equal to 2.

The input parameters of the top function (labelled as *CNN*) are the trained weights and bias, and the hand gesture image to be classified, while

the output parameter is the letter of the alphabet inferred by the CNN from the given image. As illustrated in Algorithm 1, in the first place, these arrays passed as parameters to the function are loaded in their correspondent on-chip memory caches, one for the weights (W), one for the bias(B) and one for the input image (IFM). It is worth to mention that the loading of the weights and bias is done only once, at the power-up of the system.

Then, the main loop of the CNN is processed, layer by layer, executing the following operations:

1. load the hyper-parameters of the current layer, that is to say, the dimensions of the input volume (height, width and number of input channels), the dimensions of the activation or output volume generated after the convolution, the stride, kernel size, and the fractional lengths necessary for the dynamic fixed-point representation of the data: input, output and weights/bias (see Table 3.4).

2. calculate the number of 8x8 blocks per channel that are going to be processed, given the height and width of the channel.

3. iterate through all the input channels.

4. iterate through all the blocks of the current input channel.

5. load the data of the current 8x8 input block from the input feature map cache (IFM) into the 10x10 cache and perform the padding. The 10x10 array is the matrix to be convolved by the convolution unit later on.

6. iterate through all the output channels; from the current 8x8 input block loaded in the previous loop, generate all of the 8x8 output blocks, one for each output channel.

7. load the set of weights corresponding to the current combination of input-output channels. A total of $input\_channels \cdot output\_channels$ is the number of kernels required by the current layer to perform the full convolution of the input volume.

8. perform the matrix convolution operation between the 10x10 block and the 3x3 kernel (stride 1) and store the result into an 8x8 array cache (*result*8x8).

9. each output channel is the result of the cumulative addition of the convolution of each input channel with its corresponding kernel (see Figure 4.1); hence, the obtained result is added to the previously stored one in *stored*8x8(if a previous one exists already).

Algorithm 1 : Pseudo-code for the neural network algorithm

```
1: function CNN(weights, bias, input_image, &inferred_class)
2:     W = LoadAllWeights(weights);            ▷ Only once, at power-up
3:     B = LoadAllBias(bias);                  ▷ Only once at, power-up
4:     ▷ Init the Input and Output 'Feature Map' Caches, IFM and OFM
5:     IFM = LoadInputImage(input_image);
6:     OFM = 0;
7:     for L = 0 to layers − 1 do      ▷ Main loop, iterate through layers
8:         (CH_in,CH_out,X_in,Y_in, ...) = LoadLayerDef; ▷ Load hyperparams
9:         blocks = X_in ∗ Y_in/64;            ▷ Calculate blocks per channel
10:        for ci = 0 to CH_in − 1 do
11:            for bl = 0 to blocks − 1 do
12:                block10x10 = LoadBlock(IFM,bl, ci);          ▷ Load Op1
13:                for co = 0 to CH_out − 1 do
14:                    kernel3x3 = LoadKernel(W, ci, co);       ▷ Load Op2
15:                    result8x8 = 2DConvOp(block10x10, kernel3x3);
16:                    if (ci ≠ 0) then
17:                        stored8x8 = LoadFromCache(OFM, bl, co);
18:                        result8x8 = result8x8 + stored8x8;
19:                    end if
20:                    if (ci == CH_in − 1) then
21:                        bias = LoadBias(B, co);
22:                        result8x8 = ReLU (result8x8 + bias);
23:                    end if
24:                    Store2Cache(OFM, bl, co, result8x8);    ▷ Write-back
25:                end for
26:            end for
27:        end for
28:        IFM = OFM;                  ▷ Init input cache for the next Conv
29:    end for
30:    output32x1 = averagePooling(OFM);                    ▷ Average pooling
31:    [max_value, index] = max(output32x1);
32:    inferred_class = index;                    ▷ Return inferred category
33: end function
```

10. if the above-mentioned cumulative sum is complete (meaning that all the input channels have been convolved), add the bias and apply the activation function, which is a simple ReLU. When computing the addition operations, the different fractional lengths are adjusted, and the *rounding-to-the-nearest* method is applied. Furthermore, keeping track of the possible overflows is done by applying saturation arithmetic (for 8-bits, all the results are kept in the range [+127, -128]).

11. the generated output volume that got stored in the output feature map cache *OFM* now becomes the input volume of the next convolutional layer.

12. when all the convolutions are completed, calculate the average pooling of the last activation output; the result is a one-dimensional vector with 32 elements. The index of the element with the maximum value corresponds to the category inferred by the CNN.

The functional behaviour described by Algorithm 1 maps onto the block diagram depicted in Figure 4.2. It consists of the necessary logic and control blocks that transfer the data between the different memory caches and the convolution unit. In the following subsections, a succinct description of the different modules that constitute the circuit hierarchy is given in a top-down order.

4.4.1 Main Process Unit

The Main Process unit reads the trained weights and bias from external memory and writes them onto their corresponding internal caches, implemented as dual-port block RAM. It reads the state of a bit flag to ensure that this action is done only once.

Then, it reads the image containing the hand gesture to be classified from external memory, in blocks of 8x8 pixels, and writes them back in the *IFM* input cache. The rule applied here is to read the blocks that compose the image from left to right, as illustrated in Figure 4.3. The left-to-right convention is kept throughout the design whenever required.

Once this is done, the image starts to be processed by the neural network, applying the convolutional layers in an orderly manner. The Main Process unit keeps track of the successive convolutions performed all the way through the neural network layers. It reads the hard-coded configuration for each layer from an internal ROM that contains all the layer definitions, namely the layer hyper-parameters such as channel size, number of channels, kernel size, stride, and the fractional length to be used for the appropriate

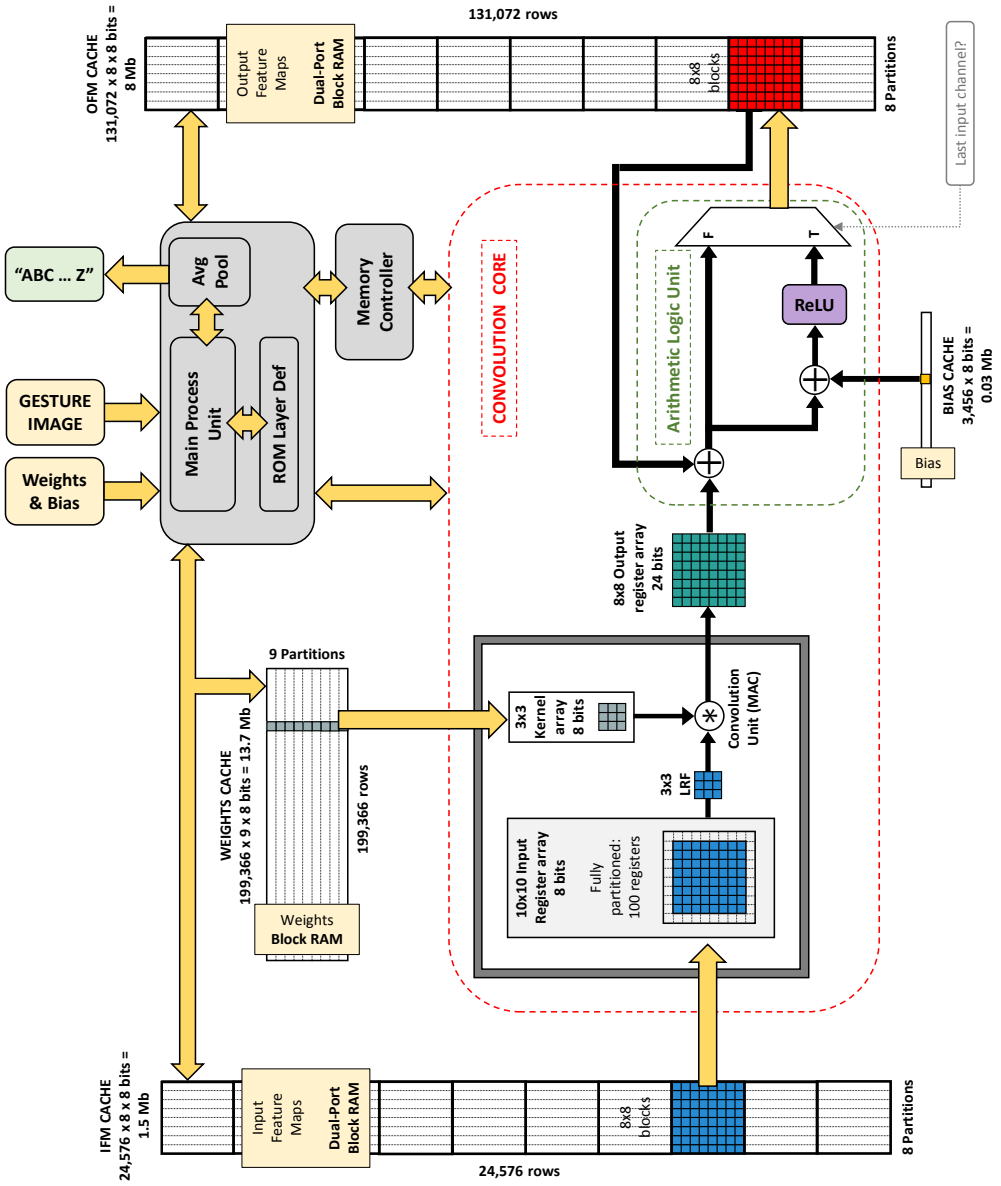

Figure 4.2: Block diagram of the CNN hardware accelerator.

representation of the data in 8-bit dynamic fixed-point at each particular convolution operation. All this parameters are listed in Table 4.2 and Table 3.4 respectively.

Finally, when all the convolutions have been completed, it computes the average pooling of the last activation volume stored in memory, which produces a vector of 32 elements or categories, one for each of the letters of the alphabet, and looks at the category with the largest value. The vector index of that category is the value returned through the output port and corresponds to the letter of the alphabet inferred by the neural network accelerator.

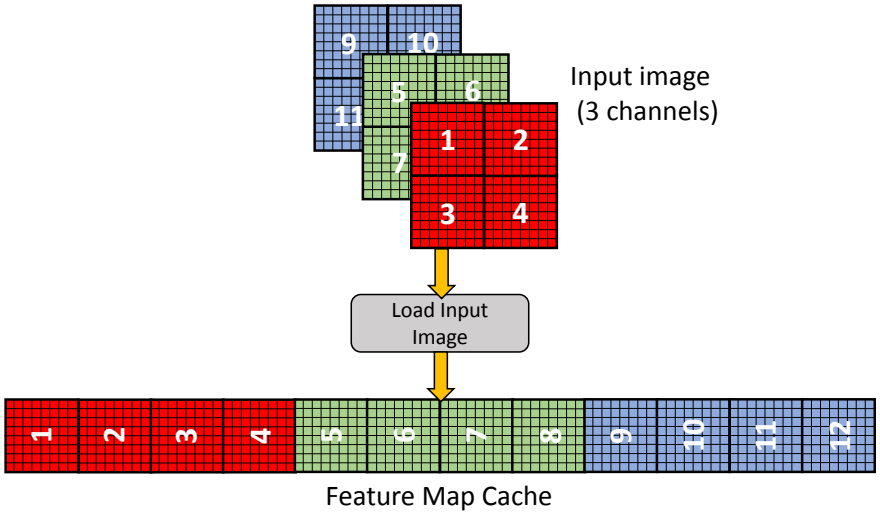

Figure 4.3: Representation of the loading process of the input image into the internal cache.

4.4.2 Memory Controller

The memory controller keeps control of the offsets required to properly access the data from the main caches (weights, bias and input/output activation volumes), and it works closely together with the Main Process unit and the Convolution Core, as they are continuously addressing data from/into the different internal memory caches.

4.4.3 Convolution Core

The Convolution Core is the module that performs the matrix convolutions iteratively, aiming to keep latency as low as possible. For that purpose, the data blocks stored in the block RAM are transferred to smaller caches made up by internal registers with much faster access. The convolution operation is done by processing the input feature maps block by block, one at a time, hence the output feature maps are also produced one block at a time. The

amount of latency needed to finish the whole convolution of an input volume will depend on the number of channels and its size; therefore it will vary from one layer to another, for they have different hyper-parameters.

The convolution processing unit requires two matrices as input operands: the first operand is an 8x8 pixel block loaded from the input feature map cache, and the second operand is a 3x3 kernel loaded from the weight cache. As mentioned above, the memory controller is keeping track of the memory offsets ensuring that the operands are accessed from the right memory locations.

The weights are written into a 3x3 register array, and the 8x8 pixel block is written onto a 10x10 register array, both implemented in the FPGA as distributed RAM memory (based on LUTs[3]) which allows that all the individual registers can be accessed simultaneously in the same clock cycle. When a 3x3 kernel is used in the convolution, it is necessary to perform the padding of the edges around the 8x8 block, hence the reason to use a 10x10 register array.

Padding

The rules applied to the padding operation are the following:

- if the edge to be padded corresponds to an edge placed at the border limit of the feature map, then pad it with zeros (example block #4 in Figure 4.4)
- otherwise, pad it with the neighbouring pixels from the adjacent blocks (example block #11 in Figure 4.4)

The blocks are orderly processed according to the above-mentioned left-to-right convention, conforming to its original position in the feature map. There would be no difference in terms of latency though if blocks were accessed in top-to-down precedence order. It is worth to remember here that the feature map caches are partitioned by columns, consisting of 8 separate dual-port block RAM that can be accessed concurrently in the same clock cycle. The padding operation also greatly benefits from this partitioning; for instance, the padding of the edges, corresponding to the *upper* and *lower* rows on the 10x10 cache (see Figure 4.4), would require only one clock cycle, as all the elements are located in different memory blocks. Whereas the padding of the left and right columns is more expensive, requiring five clock cycles to access those 20 elements.

[3]LUTs or Look-Up Tables are built out of SRAM bits and are typically used by FPGAs to implement combinational logic or lookup tables associated to complex logic functions.

There is the possibility to re-use one of the current block columns for the padding of the next block. By re-assigning the current values of the registers in the column number 9 (see example block #11 in Figure 4.4) to the column number 1, labelled as *left* in the figure, it eliminates the need to access the block RAM to retrieve the data for that column. However, this advantage gets overshadowed by the fact that the data for the padding of the *right* column still needs to be retrieved from the block RAM, which takes five clock cycles, so there is no reduction in latency by doing so.

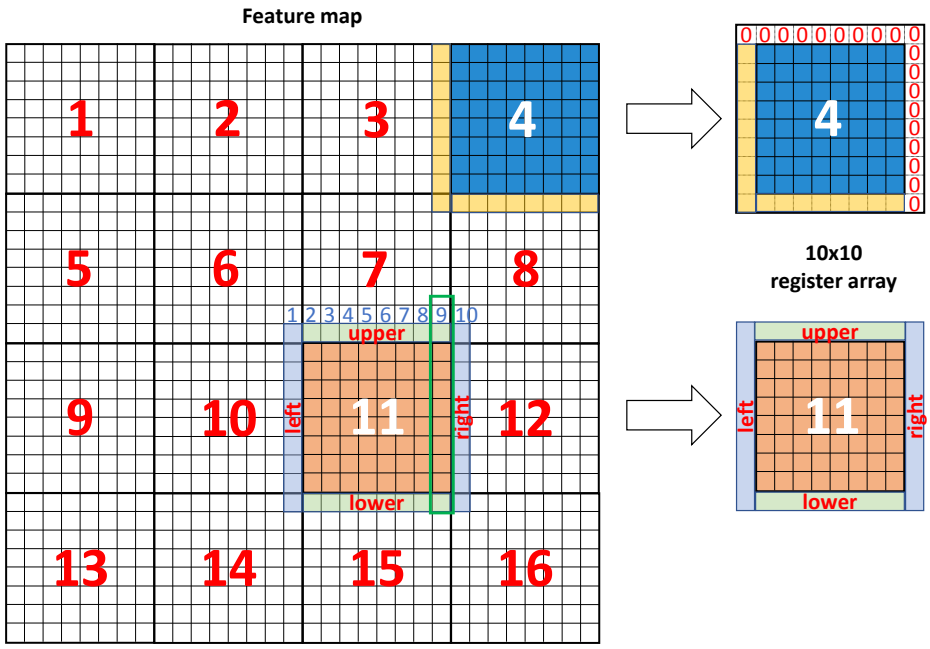

Figure 4.4: Examples of the padding of the blocks forming the feature maps. The numbering in the blocks indicates the processing order, row-wise from left to right.

Convolution Unit

The 2-D convolution operation is performed by *sliding* the 3x3 kernel throughout all the pixels contained in the current 8x8 block, that has been previously padded, one pixel at a time. The kernel is centred over the pixel that is going to be processed, which, together with its neighbouring pixels surrounding it, form a pixel area termed as *local receptive field*[4] (LRF), also known as *convolution window*. The data content of this window, as it might

[4]See Section 2.1

be expected, is changing as long as the kernel is changing its position. For that purpose, a small 3x3 register array has been implemented (labelled as LRF in the block diagram) that stores the temporary content of the convolution window.

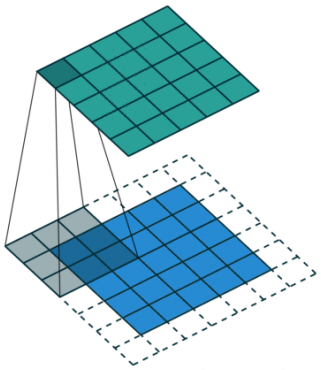

Figure 4.5: The kernel (grey 3x3 matrix) slides over the padded block (blue matrix with white borders) producing one pixel at a time in the output matrix (in green). The grey area covered by the kernel corresponds to the *local receptive field (Source: Dumoulin and Visin, 2016 [50])*.

Once both the 3x3 array registers are loaded (the convolution window and the kernel), the convolution unit can perform a partial convolution computation, which is iteratively done pixel by pixel. By definition, a 2D matrix convolution requires the flipping of the elements of the kernel matrix in both dimensions, from left to right and from top to bottom, as shown in the following example:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \Rightarrow \begin{bmatrix} i & h & g \\ f & e & d \\ c & b & a \end{bmatrix}$$

This additional manipulation of the kernel by the hardware can be just avoided, by previously flipping the weight kernels before loading them in the FPGA cache. This action is done *offline* actually by the training framework (NVIDIA DIGITS [33]) and it is advantageous because the convolution unit can be simplified to a multiply-accumulate operation (MAC) and thus, be implemented in hardware simply as a MAC unit as illustrated in Figure 4.6.

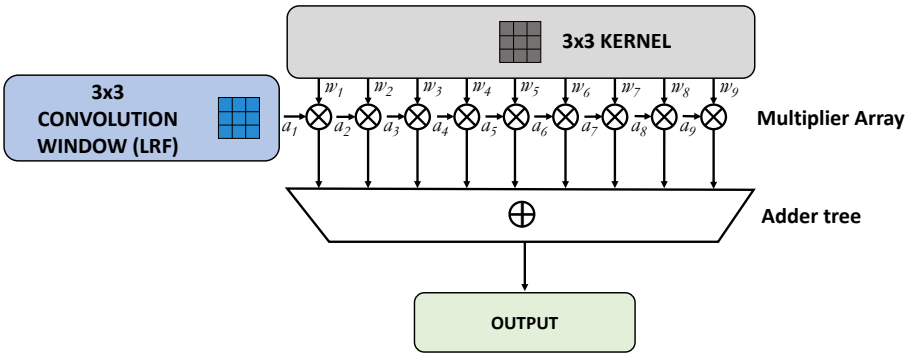

Figure 4.6: Multiply-accumulate unit.

The performance of the MAC unit is improved by implementing an array of 9 multipliers in parallel, which delivers the element-wise multiplication of the two matrices in the same clock cycle, and then followed by an adder tree that computes the total sum of the 9 elements.

The algorithmic description in C of the convolution operation makes use of two nested `for` loops, one for each block dimension (see Listing 4.1), which can be further optimised in terms of latency by unrolling the inner loop. In that way, the convolution unit hardware can be parallelised, and several block pixels can be processed at the same time. For instance, if the inner loop is completely unrolled, as shown in the listed code, the compiler creates 8 different convolution windows (each made of a 3x3 register array) and the same number of kernel register arrays. In the same way, the compiler implements 8 different MAC units with 9 multipliers and one adder tree each. As a result, eight output pixels can be processed in the same clock cycle.

4.4.4 Arithmetic Logic Unit

As previously described in Section 2.1 and also by the pseudo-code in Algorithm 1, each output channel is composed by the total sum of the convolutions of all the input channels by its corresponding kernels. The accumulation of the input channel convolutions is stored in the output cache (OFM). Once the convolution operation is completed for one block, the arithmetic logic unit accumulates the partial outcome to its corresponding output block, stored in the OFM cache, where the sum is performed as a simple addition of 8x8 matrices.

```
void conv2d_3x3(
                int8_t  in_cache[10][10],
                int8_t  kernel[3][3],
                int16_t out_cache[8][8]
                )
{
int8_t window_3x3[3][3];
#pragma HLS ARRAY_PARTITION variable=window_3x3 complete dim=0

CONV2D_3x3_ROWS:
  for (int8_t ifm_row = 1; ifm_row < 9; ifm_row++){
CONV2D_3x3_COLS:
    for (int8_t ifm_col = 1; ifm_col < 9; ifm_col++){
    #pragma HLS UNROLL
      slide_window(in_cache, ifm_row, ifm_col, window_3x3);
      out_cache[ifm_row-1][ifm_col-1] = macc_3x3(window_3x3, kernel);
    }
  }
}
```

Listing 4.1: C code for the 2-D convolution operation.

Dynamic Fixed-Point Arithmetic, Rounding and Overflow Handling

Since the implemented design uses 8-bit dynamic fixed-point for data representation, the decimal point can vary its position according to the fractional lengths given in Table 3.4. In order to be able to perform dynamic fixed-point arithmetic, the unit is endowed with the following functionalities:

- track the position of the decimal point after multiplication of two quantities.
- align the decimal point between quantities before addition.
- align the decimal point of the addition/multiplication result to the proper fractional length using bit-shifting operations.
- when applying bit shifting, quantities are not truncated but rounded; the rounding method applied is the *rounding-to-nearest* method, with rounds the quantity to the closest representable number in the direction of positive infinity.
- handle the overflows using a saturating overflow rule (magnitudes are kept in the range [+127, -128] which corresponds to the range of 8-bit signed integers).

Bias Addition and ReLU Activation Function

The arithmetic logic unit also adds the corresponding bias to the output when all the input channels have been convolved, performing an element-wise addition between the bias (a scalar) and an 8x8 matrix. After adding the bias, the ReLU activation function is applied, which consists in zeroing all the negative values in the matrix. The final result is stored back in the output cache.

4.5 Behavioural Model Validation and RTL Verification

This section describes the methodology used to validate both the C-based algorithm description that models the behaviour of the neural network, and the verification of the RTL description obtained after the successful synthesis of the model.

4.5.1 Validation of the C-based Model Description

The Vivado HLS Design Suite from Xilinx [51], supports complete bit-accurate validation of the C model, at the pre-synthesis phase, and also provides a productive C-RTL co-simulation verification solution after the synthesis of the proposed solution.

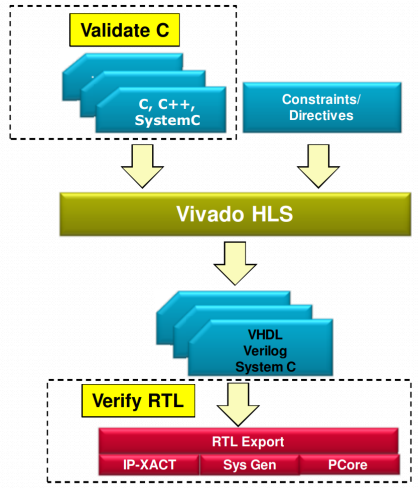

Figure 4.7: Vivado HLS Flow: C Validation and RTL Verification *(Source: Xilinx Inc., 2013 [52]).*

The validation of the C-based algorithm behaviour has been tested at different levels. At an initial step, a script has been used to generate

random input matrix volumes with user-defined number of input channels and sizes. Random weight kernels and bias are also generated by the script, accordingly to the number of output channels and the kernel size. Additionally, it computes the 2-D convolution of the given matrices, adds the bias and applies the ReLU activation function to the obtained result. Afterwards, the script was modified in order to generate and compute results using a dynamic fixed-point representation of the quantities. The randomly generated input and the weights are stored in different text files, as well as the result of the convolution, which later are used as *golden data* files by the C model validation testbench.

The testbench is defined on top of the function to be synthesised, namely the function that contains the neural network algorithm. Hence, the testbench contains the `main()` function definition from where the call is made.

On a second step, once the convolution functionality has been validated, a layer-by-layer analysis approach has been taken, using the intermediate feature maps generated by DIGITS as a golden data and compared to the intermediate feature maps generated by the C algorithm. In that way, the validation of the neural network is done layer after layer, by checking that the obtained data matches the golden results. Accordingly, one can check that the different implemented functionalities work as expected, like the use of different kernel sizes (1x1 or 3x3) combined with the use of different stride lengths, the block processing approach used for the convolution, the fixed-point arithmetic and so on.

Ultimately, in the last step, once the whole neural network proofs to infer the correct prediction for a given image, another testbench is used to measure the inference accuracy of the network. For that purpose, a test dataset of 136 pictures never *seen* by the neural network during the training phase, is used (see results in Chapter 5). Thus, one can compare the inference accuracy rate obtained by the C algorithm with the one obtained during the training.

4.5.2 Post-Synthesis RTL Verification

Finally, the RTL design obtained after the synthesis needs to be verified. Typically, one can program the target FPGA device with the generated RTL netlist and with the help of a digital logic analyser check the target signals or ports. Alternatively, the Vivado HLS design toolchain offers a verification solution named C/RTL co-simulation which reuses the pre-synthesis C testbench (using same input files for image, weights and bias) but the output is actually generated by the RTL implementation, not the C algorithm, and compared with the testbench golden output (see results in Chapter 5).

4.6 HLS Limitations and Issues

This section summarises the most significant problems encountered during the design phase and the workarounds or solutions proposed to tackle them.

Once the functionality of the C-based algorithm was validated, to boost the throughput of the convolutional neural network proved to be the more challenging part of the project. Although the HLS flow dramatically enhances the productivity, it takes quite a lot of time to dominate the methodology and to write optimised algorithms from a hardware designer perspective.

HLS Limitations

- **Unrolling loops with variable bounds.** This type of loops cannot be unrolled, and as a consequence, it prevents pipelining from being applied. That is the case for the neural network, where each layer is defined by hyper-parameters with different values. As depicted in Algorithm 1, one can see that several nested loops listed in the pseudo-code are bounded by variable quantities, like the number of layers, or the number of input/output channels. The very inner loop of this algorithm, the one that iterates through the output channels, has the potential to be unrolled and produce several output blocks in parallel. One way to work around the unrolling limitation is to describe this behaviour explicitly in the code, meaning to say, by making in the same loop iteration as many function calls as output blocks in parallel are to be produced. That is done in conjunction with the memory controller which has to adjust the memory offsets before calling the due functions so they can properly access the data to be processed from the large feature map caches.

- **Pipelining.** The pipeline compiler directive can be applied to both loops and functions, and greatly improve the throughput. Pipelining a function though, automatically completely unrolls all the loops in the body of that function (except those with variable bounds as mentioned in the previous point) and that also includes the loops contained in subordinate functions. In that sense, the designer does not have much control over the final area obtained after synthesis, so performance is traded-off by an area that is not finely optimised.

- **C simulation of large arrays.** Another problem encountered when using very large arrays is that the C simulation may run out memory. That is the case for the cache that stores the weights. The cause for this behaviour is that, by default, the array is placed in the smaller stack memory, instead of the heap memory, which uses local disc space

and can grow dynamically as much as needed. The best solution is to declare such arrays as `static` with a negligible impact on the code, in opposition to the typical use of the `malloc()` function, which obliges two have two different versions of the code, one for the simulation and another one for the synthesis, because `malloc()` cannot be synthesised by the HLS tool.

Other Issues

- **Demo setup**. The original idea has been to provide proof of concept of a neural network being able to recognise human sign language, by means of a demo setup which requires the use of a digital camera, a PC-based system for processing the images and the hardware accelerator proposed in this thesis, deployed on a Xilinx FPGA for the image classification. To make this possible, apart from the hardware accelerator, it also needs the design of additional modules that bring up the required functionality, such as streaming of the images from the camera to the host CPU, pre-processing of the captured images and implement the communication interface between the FPGA and the CPU. As mentioned in an earlier section, the FPGA of choice is the Xilinx Board ADM-PCIE-KU3, from the Kintex UltraScale family, which mostly benefits from the SDAccel development environment designed by Xilinx [53]. This FPGA device targets standard 64-bit x86-based workstations, and is installed via the PCIe connector. Unfortunately, the SDAccel environment, although it brings powerful tools also comes at the expense of a more complex design flow that requires additional training. Given the project's time constraints together with the learning curve required by SDAccel, it has not been possible to complete a satisfactory working demo setup.

- **Test the RTL design in real hardware**. Initially, the verification of the RTL design was meant to be done with the standard HDL verification tools provided by MATLAB, specifically, the *FPGA-in-the-loop* simulation flow [54] that uses the Simulink software environment to directly stimulate the implemented design on a real FPGA with input image files an then analyse its response. Moreover, any internal signals can be monitored by using an integrated logic analyser included in the Simulink tool menu. Unfortunately, at the moment there is no available FPGA board support package that includes the definition files for this specific board, which allows for the verification flow proposed above. Additionally, power consumption measurements have not been performed either for the same reasons exposed here.

Chapter 5

Results and Conclusions

This chapter includes the results obtained from the different accuracy tests and experiments conducted on the hardware accelerator. Accuracy is assessed with a test dataset, and the obtained data is used to build a confusion matrix and to calculate the recall and the precision of the model. Finally, the area and performance results from the different solutions obtained during the exploration of the design space are presented and reviewed.

5.1 Accuracy Results

As described in a previous chapter, a *training* dataset and a *validation* dataset were used to train the model (see Section 3.2). Now, once the RTL design has been obtained and verified, a *test* dataset is going to be used to measure the inference accuracy of the implemented solution. This small set of 136 images contains hand gestures taken against a white wall background (see Figure B.1 in Appendices) which should make it easier for the hardware-embedded network to recognise them.

Confusion Matrix

The obtained results have been taken to construct a *confusion matrix*, like the one shown in Figure 5.1, which is a table used to describe the performance of a neural network model on a set of test data for which the classes or categories are known.

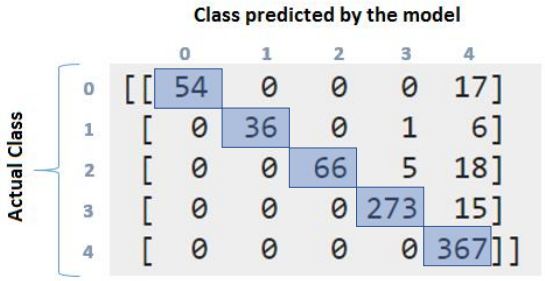

Figure 5.1: Example of a confusion matrix for a test dataset with five different classes labelled as 0, 1, 2, 3 and 4 *(Source: derived from Aditya, 2015 [55]).*

The terms in the matrix diagonal correspond to *true positives*, meaning the number of times an actual class has been correctly inferred by the model, while the rest of the elements out of the diagonal are the *false positives* or the number of times an actual class has been confused by another one.

Table 5.1 is a simplified view of the obtained confusion matrix which contains two important metrics: the *true positive rate* (TPR, also termed as *recall* or *sensitivity*) and the model *precision*. The precision metric is a different way to look at the results; for instance, in Figure 5.1 it can be seen that the input class 4 has been correctly predicted all the times (367 times), but it can also be seen that not every time the model predicted the class 4, the prediction was correct. For example, regarding class 3, it has been confused by class 4 in 15 occasions, and seemingly class 2 has been confused by class 4 in 18 occasions. Hence, precision measures how *accurate* is the classifier when predicting positive instances.

Looking at the actual data presented in Table 5.1, it can be seen that, for example, the worst TPR corresponds to class F with a 25% rate, while the precision for the same class is 100%. In another case, class N presents very low values for both indexes, of 40% and 50% respectively. In both cases, the data reveal possible weak points of the neural network performance.

Indeed, the data obtained by the confusion matrix can largely help when trying to fix these issues. In order to improve the TPR and the precision, different actions may be required before retraining the model again, such as increasing the number of images for a specific class or, on the contrary, drop-out some images that could be problematic for the training. For the sake of the project, the refinement of the training dataset has not been taken further, and the focus has been kept in the optimisation of the hardware.

Test Accuracy

In overall, the classification test accuracy of the proposed accelerator is 80.1%, which is a little bit below compared with the final 87.1% validation accuracy obtained after fine-tuning the model with Ristretto (see Table 3.5). The difference must be accounted to the fact that the validation dataset and the test dataset are not equal, but nonetheless, a test accuracy about 80% should be considered rather satisfactory.

Class	#positives	TPR (recall)	Precision
A	6	100.0%	54.5%
B	6	85.7%	100.0%
C	2	50.0%	100.0%
D	5	100.0%	71.4%
E	3	60.0%	50.0%
F	1	25.0%	100.0%
G	2	40.0%	100.0%
H	4	80.0%	100.0%
I	4	80.0%	80.0%
J	5	100.0%	100.0%
K	5	100.0%	83.3%
L	6	85.7%	85.7%
M	4	80.0%	100.0%
N	2	40.0%	50.0%
O	5	83.3%	83.3%
P	5	83.3%	71.4%
Q	3	75.0%	75.0%
R	3	75.0%	100.0%
S	6	100.0%	85.7%
T	5	100.0%	100.0%
U	6	100.0%	75.0%
V	4	80.0%	100.0%
W	5	100.0%	83.3%
X	3	60.0%	50.0%
Y	4	60.0%	100.0%
Z	5	83.3%	83.3%
	Test dataset size: 136 images		
	Total number of positives: 109		
	Overall test accuracy: 80.1%		

Table 5.1: Rate of true positives (recall) and precision of the model for each one of the classes, obtained from the simulation of the RTL description model.

It is worth to mention that the obtained test accuracy is not a hardware-

dependant quantity, considering that the algorithm has been described properly. It depends on the training methodology and how good the attained weights are. Once the weights are hard-coded in the FPGA hardware, the accuracy of the accelerator must be the same whatever the chosen design to implement the algorithm is.

Accuracy Comparison with Previous Works

Table 5.2 shows a comparison of previous works that used a very similar training methodology like the one described in this report. The list of works showed in the table are not hardware implementations of a CNN though, but actually, what it is worth to compare is the results obtained from the training phase. The transfer learning method has been adopted in all the cases except in one, where the neural network was trained from scratch with randomly initialised weights, and the number of categories to classify by the models is similar in all cases.

Proposed method	Hand alphabet (#categories)	ConvNet	Transfer Learning	Validation Accuracy
Garcia-Biesca [56]	American (24)[1]	GoogleNet	Yes	72.0%
Bheda-Radpour [57]	American (24 + 10)[2]	*custom*	No	82.5%
Kang *et al.* [58]	American (24 + 7)[3]	CaffeNet	Yes	85.5%
This work	Swedish (26)	ZynqNet	Yes	87.1%

Table 5.2: Accuracy comparison of previous works with the model proposed in this thesis work.

5.2 Resource Utilisation and Performance

Design space exploration has been performed in various ways to find an optimal hardware with the desired functionality. The different solutions obtained have been split into two groups, labelled below as *experiment 1* and *experiment 2*.

The reason for having these two separate sets of results is because a different approach has been taken for each one when describing the algorithm in C. The coding approach in *experiment 1* has been to avoid conditional

[1] American sign language includes two non-static gestures that have been removed from the dataset, which are *J* and *Z*. Therefore, the size of the set gets reduced to 24.

[2] Signs for representing the digits from 0 to 9 have been included in the dataset.

[3] Signs for representing the digits 1, 3, 4, 5, 7, 8, and 9 have been included in the dataset.

branching as much as possible, thus separating different functionalities into different functions or methods. For instance, there is a function used for the 3x3 convolutions and another one for the 1x1 convolutions. Both functions have large portions of code in common, so more area is required to implement them.

On the other hand, the coding approach in *experiment 2* completely embraces conditional branching, unifying similar methods in one function body and using `if-else` statements when required, thus obtaining a more compact code. The purpose for having these two approaches is to test which one results in a better task schedule, and therefore, a better performance of the RTL design synthesised by the HLS compiler.

The other two variables used during the design exploration are the clock period and the number of convolution cores in parallel. As shown in Tables 5.3 and 5.4, for different combinations of these two variables one can obtain hardware solutions with different performance and area requirements. The resource utilisation is expressed in the tables as a percentage of the total available, and the performance is given in terms of latency and throughput, expressed in milliseconds and frames per seconds respectively, where the latter refers to the number of images that can be classified per second by the hardware accelerator.

Regarding the convolution core (see the module in Figure 4.2), the potential for parallelisation of this module relies on the fact that the output channels can be computed concurrently, block by block. The maximum number of output channels that can be processed in parallel can be determined with the help of Table 4.2, which describes the ZynqNet architecture, layer by layer. The minimum number of output channels is 16, as defined for layers `fire2\squeeze3x3` and `fire4\squeeze3x3`. The number of output channels for the rest of layers is a multiple of 16. Hence, it turns out that 16 is the maximum number of output channels that can be processed in parallel, using one convolution core per channel.

Experiment 1

In experiment 1, two separate functions for 3x3 and 1x1 convolutions have been implemented, and the maximum number of convolution cores that could be synthesised is 4 per function (8 in total). When trying to implement 8 cores, the proposed solution run out of resources.

Table 5.3 shows the data obtained from the different solutions implemented during the experiment. The amount on block RAM remains immutable because it is used for the storage of the weights, bias, input images and intermediate activations, which have precisely defined sizes. The number of convolutions cores has a clear impact in the utilisation of the

other FPGA resources, specially DSP48 multipliers and LUTs. The more cores, the more consumed area, but also the system throughput improves. Seemingly, by increasing the system clock frequency, the performance also increases significantly, although at the cost of a slight increase in area.

#cores (x2)	T_{CK} [ns]	BRAM %	DSP48 %	FF %	LUT %	Latency [ms]	Throughput [FPS]
1	10	69	23	6	47	302.8	3.3
1	7.5	69	23	8	48	271.8	3.7
1	4	69	23	15	48	188.4	5.3
2	10	69	47	8	59	266.0	3.8
2	7.5	69	47	11	59	240.9	4.2
2	4	69	48	20	59	169.7	5.9
4	10	69	71	10	71	247.8	4.0
4	7.5	69	71	14	71	247.8	4.4
4	4	69	72	28	72	159.0	6.3

Table 5.3: Experiment 1. Post-synthesis resource utilisation and latencies as a function of the clock frequency and the number of convolution cores in parallel (FPGA device: Xilinx XCKU060).

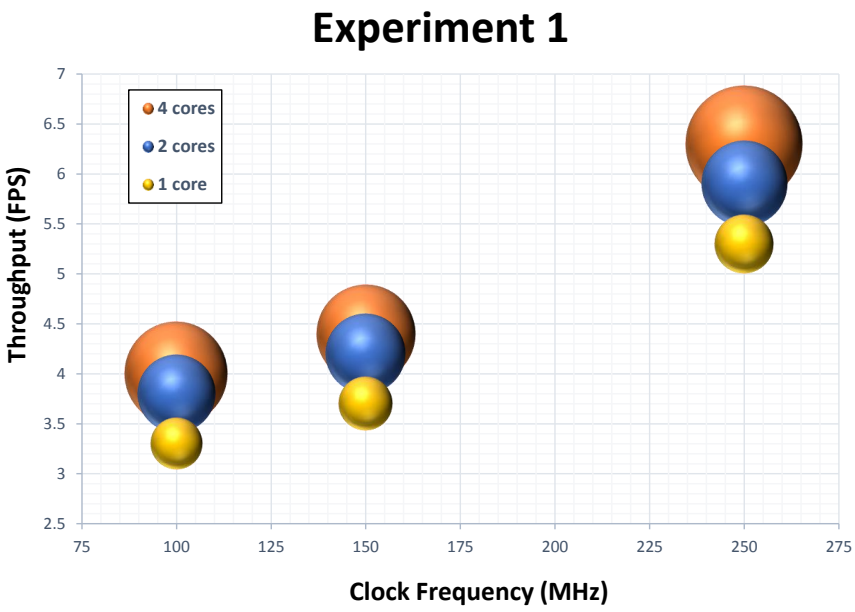

Figure 5.2: Representation of throughput versus number of cores and FPGA clock frequency. The width of the spheres is proportional to the total resource utilisation of the FPGA.

The task scheduler elaborated by the HLS compiler clearly shows the convolution operations running in parallel. However, the bottleneck in the performance is mostly due to the fact that the output blocks processed by the convolution units cannot be stored concurrently, as they share the same block RAM partitions, so this step is done sequentially.

Nonetheless, a rather acceptable classification rate of 6.3 images is achieved at 250 MHz and with 8 convolution cores. On the other end of the spectrum, there is a much smaller and slower solution, but with a factor 6 less power consumption, with 3.3 FPS at 100 MHz and with only 2 cores.

Figure 5.2 illustrates in a more visual view the data displayed in Table 5.3. The width of the spheres is proportional to the area consumed by the proposed solution, which depends on the number of cores and the system clock frequency.

Experiment 2

The results obtained for experiment 2 are illustrated in Table 5.4 and Figure 5.3 respectively, in a similar way as it was done for experiment 1. In this experiment, the methods for 3x3 and 1x1 convolutions have been unified in a single function, and conditional branching is used instead.

#cores -	T_{CK} [ns]	BRAM %	DSP48 %	FF %	LUT %	Latency [ms]	Throughput [FPS]
1	10	69	23	4	30	287.7	3.5
1	7.5	69	23	6	28	272.5	3.7
1	4	69	23	9	26	217.0	4.6
2	10	69	24	5	32	250.0	4.0
2	7.5	69	24	6	30	235.1	4.3
2	4	69	24	10	28	182.0	5.5
4	10	69	24	5	32	231.5	4.3
4	7.5	69	24	6	30	216.2	4.6
4	4	69	24	10	28	164.4	6.1
8	10	69	25	5	34	222.0	4.5
8	7.5	69	25	6	32	206.7	4.8
8	4	69	25	11	30	155.5	6.4

Table 5.4: Experiment 2. Post-synthesis resource utilisation and latencies as a function of the clock frequency and the number of convolution cores in parallel (FPGA device: Xilinx XCKU060).

For experiment 2, the number of convolution cores in parallel could be increased up to 8. Further increasing up to reach the limit of 16 have not

been possible because the C/RTL co-simulation failed every time (the reason for that behaviour is not clear yet). It can be seen that the area is barely increased by a rise in the number of cores. A more thorough inspection of the scheduled tasks showed that only one convolution process runs at a time, no more function instances are found running concurrently. The scheduler does not start the processing of another output channel until the current computed output block is stored in the BRAM memory. Hence, the HLS compiler only generates one single instance for the convolution process. Surprisingly, the obtained performance is almost similar to the one achieved in experiment 1 for the solution with the fastest throughput, but the resource utilisation is decreased by almost a factor 3. This is a clear indication again that performance could be further improved if additional partitions were done to the BRAM blocks that store the layer output activations because this would allow the concurrent storing of the processed blocks.

It is also observed that all the solutions obtained with different combinations of number of cores and clock frequency utilise almost the same area; the increase of either number of convolution cores or clock frequency always results in a significant increase of performance with minimal effect in the resource consumption.

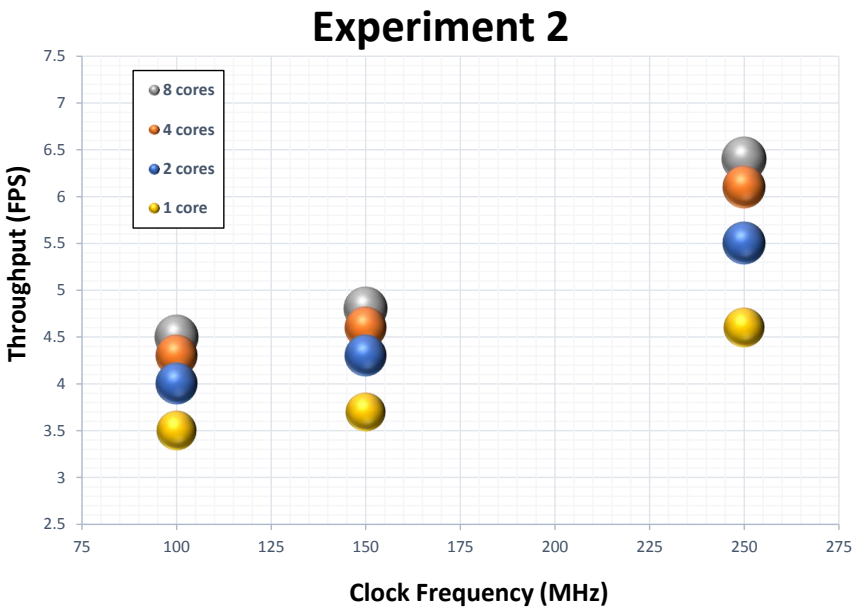

Figure 5.3: Representation of throughput versus number of cores and FPGA clock frequency. The width of the spheres is proportional to the total resource utilisation of the FPGA.

Comparison of Resource Utilisation and Performance

A comparison of resource utilisation and performance is made against the original ZynqNet from Gschwend [25], derived from SqueezeNet, and SqueezeJet, proposed by Mousouliotis and Petrou [59], which is a small FPGA convolutional layer accelerator for SqueezeNet with the aim to enable the development of mobile computer vision applications as described by the authors.

The comparison is just illustrative to evaluate the effect of adopting techniques such as transfer learning and network quantisation used in this work. It is also worth not to forget that the performance results obtained by ZynqNet and SqueezeJet are obtained from the 1000-class ImageNet challenge, so the classifier layer has a larger latency compared to the 32-class classifier used for the sign recognition task.

	Gschwend, 2016 [25]	Mousouliotis & Petrou, 2018 [59]		This work
Topology	ZynqNet	SqueezeJet	SqueezeJet	ZynqNet
Derived from	SqueezeNet[4]	SqueezeNet	SqueezeNet	SqueezeNet[4]
Dataset	ImageNet	ImageNet	ImageNet	Hand alphabet
Platform	xc7z045 [60]	i3-7100[5] [61]	xc7z020 [60]	xcku060 [49]
Company	Xilinx	Intel	Xilinx	Xilinx
Technology [nm]	28	14	28	20
Representation	Floating-point	Floating-point	Fixed-point	Fixed-point
Bit-width	32	32	8-16[6]	8
Frequency [MHz]	100	2,400	100	100
Use off-chip RAM	Yes	Yes	Yes	No
BRAM [18Kb]	996	-	270	1492
DSP48	739	-	186	700
FF	137K	-	31K	35K
LUT	154K	-	21K	113K
Latency [ms]	1955	169.2	333.1	222.0
FPS	0.51	5.9	3.0	4.5

Table 5.5: Resource comparison between the original floating-point ZynqNet and the quantised version used in this thesis for sign language recognition.

As shown in Table 5.5, the amount of BRAM blocks used by the accelerator proposed in this thesis work is still a 50% larger compared to

[4]The used topology is derived from SqueezeNet.

[5]The authors of SqueezeJet have also implemented the network in a Intel CPU for comparison purposes.

[6]8-bits is used to represent weights and bias, and 16-bits is used for the activations.

the original ZynqNet. However, as stated in the project goals, this design completely avoids to access external memory and keeps everything in the on-chip memory of the FPGA, that is weights, bias, the input image and the intermediate layer output activations (the input image gets overwritten later by the subsequently computed layer activations). ZynqNet suffers from the fact of adopting the floating-point representation; otherwise, the BRAM utilisation would be similar to that of SqueezeJet.

The number of DSP48 multipliers used in this design is similar to ZynqNet, whereas SqueezeJet uses about one-fourth of them. Again, ZynqNet would likely reduce the number of DSPs drastically if quantised. In comparison to SqueezeJet, the design proposed in this work makes more extensive use of the multipliers due to the block processing approach. However, this allows a 10x10 image tile to be convolved in only two clock cycles, reducing the inference latency and boosting the number of frames per second (up to 6.4 FPS at 250 MHz).

5.3 Conclusions

In light of the results exposed in the previous section, the following conclusions can be drawn. Firstly, it is demonstrated that the transfer learning methodology is an effective technique to be used as the starting point for the training of a neural network model on a new task, whenever it is not possible to build a more extensive training dataset due to lack of resources. Additionally, data augmentation is always a good asset to be used for the sake of improving the size and quality of the dataset. In the same direction, the results also showed that accuracy was further improved as more images obtained from more volunteers were included to the training dataset.

After obtaining a satisfactory trained model, further optimisation has been achieved by applying quantisation. This network compression technique has allowed obtaining a compressed version of the model that uses 8-bit dynamic fixed-point representation, which granted a posterior implementation of a hardware accelerator that does not require any access to external memory to fetch data, thus improving the accelerator performance.

Regarding the hardware generation methods used for this thesis work, the results confirm the benefits of using the HLS methodology for the design of deep neural networks in general. However, it requires a long learning curve, and the obtained RTL may not be as perfectly optimised as desired, but the benefit in using HLS comes from the notable gain in productivity, which in turn, translates into a more efficient exploration of different design alternatives.

Of course, the coding style has a significant impact on the obtained hard-

ware solutions, as it has been demonstrated in the performed experiments. For instance, conditional branching has proven to be the right coding style to embrace, as it allows to obtain designs with the same performance but using much fewer resources.

Finally, regarding the accelerator performance, the results in Table 5.3 and Table 5.4 show that the block processing strategy is efficient and furthermore, by processing the output channels concurrently with implementing various convolution cores in parallel, the throughput can be further improved. However, as expected, the bottleneck is produced when trying to store those parallel-processed 8x8 blocks for the different output channels as it cannot be done concurrently. Therefore, additional partitioning of the output feature cache is needed. By applying this action, it is expected that the performance gets improved even more.

5.4 Future Work

As mentioned above, the output cache that stores the layer activations should be further partitioned for greater performance to be achieved. For instance, for eight convolution cores to run in parallel, the output cache should be split accordingly into eight new partitions. Table 4.3 describes the activation caches, which are defined as 2-dimensional matrices of 131,072 x 8 elements and, as already described in Section 4.3.2, the second dimension of these caches has been completely partitioned into 8 banks of 131,072 x 1 elements each to improve performance. Now, the extra boost in performance requires also the partition of the first dimension, which would allow storing multiple 8x8 blocks in parallel, as many as new partitions. Accordingly, the memory controller needs to be carefully modified to manage the new cache organisation.

Besides, further network compression to spare FPGA resource utilisation can be still done, from the hand of weight pruning. Pruning removes the weights which do not have a significant impact in any of the network output classes, and this is done by approximating the small weight values with zeros, with virtually no accuracy degradation. Different frameworks exist nowadays that can analyse the model and perform the pruning automatically using different methods [62], [63].

Finally, to set a demonstrator that can show the performance of a neural network in real-time can probably be a good starting point for another thesis work, where the recognition task can be similar to the one described in this report. Alternatively, the task of recognising static hand signs can be extended to the dynamic domain with the use of 3D-CNNs [64] which can detect actions or movements from a video stream, though they might

require more computational power. A 3D-CNN would solve the problem for detecting, for instance, the non-static signs used for fingerspelling the characteristic vowels *ä*, *å* and *ö* in the Swedish hand-alphabet, or could even be used for more complex tasks as large-scale video classification. There exist several off-the-shelf CMOS camera modules for streaming video to the FPGA [65], [66]. It would also require the implementation of a soft microprocessor core such as MicroBlaze, a 32-bit RISC microprocessor designed for Xilinx FPGAs, which would perform the preprocessing of the video images (like object location) before feeding the images to the neural network for inference. As shown in the results, with the resource utilisation required for the implementation of ZynqNet in a Kintex UltraScale XCKU060, there is room enough for the implementation of a MicroBlaze and additional logic modules. Regarding the FPGA device used for this project and other similar versions which provide connectivity through the PCI Express bus standard (PCIe), it would be highly recommendable to adopt the SDAccel development environment provided by Xilinx [53].

References

[1] A.P. Dhawan, Y. Chitre, and C. Kaiser-Bonasso. "Analysis of mammographic micro-calcifications using Gray-level image structure features". In: *IEEE Transactions on Medical Imaging*. June 1996.

[2] Björn Malmgren and Ulf Nordlund. "Application of artificial neural networks to palaeoceanographic data". In: *Palaeogeography, Palaeoclimatology, Palaeoecology* 136.Issues 1-4 (Dec. 1997), pp. 359–373.

[3] Charles E. Cox and Ekkehard Blanz. "GANGLION – A fast hardware implementation of a connectionist classifier". In: *Proceedings of the IEEE on Custom Integrated Circuits Conference*. May 1991.

[4] Yann LeCun, Yoshua Bengio, and Leon Bottou. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. Vol. 86. Issue 11. Nov. 1998.

[5] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36 (1980), pp. 193–202.

[6] Dan Claudiu Ciresan et al. "Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition". In: *Computing Research Repository* abs/1003.0358 (2010).

[7] Dominik Scherer, Hannes Schulz, and Sven Behnke. "Accelerating Large-scale Convolutional Neural Networks with Parallel Graphics Multiprocessors". In: *Proceedings of the 20th International Conference on Artificial Neural Networks: Part III*. ICANN'10. Springer-Verlag, 2010, pp. 82–91.

[8] Yuchi Tian et al. "DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. 2018, pp. 303–314.

[9] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *Computing Research Repository* abs/1712.01815 (2017).

[10] Yonghui Wu, Mike Schuster, and Zhifeng Chen et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *Computing Research Repository* abs/1609.08144 (2016).

[11] Veera Ala-Keturi. "Speech Recognition Based on Artificial Neural Networks". In: *Helsinki University of Technology* (2004).

[12] Naveen Rao. *Intel Nervana Neural Network Processors (NNP) Redefine AI Silicon*. Oct. 2017. URL: https://ai.intel.com/intel-nervana-neural-network-processors-nnp-redefine-ai-silicon (visited on 06/01/2019).

[13] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. Toronto, 2017, pp. 1–12.

[14] Kaiyuan Guo et al. "A Survey of FPGA Based Neural Network Accelerator". In: *Computing Research Repository* abs/1712.08934 (2017).

[15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "BinaryConnect: Training Deep Neural Networks with binary weights during propagations". In: *Computing Research Repository* abs/1511.00363 (2015).

[16] Jost Tobias Springenberg et al. "Striving for Simplicity: The All Convolutional Net". In: *Computing Research Repository* abs/1412.6806 (2014).

[17] H. T. Kung, Bradley McDanel, and Sai Qian Zhang. "Adaptive Tiling: Applying Fixed-size Systolic Arrays To Sparse Convolutional Neural Networks". In: *ICPR*. IEEE Computer Society, 2018, pp. 1006–1011.

[18] Dejan Tanikić and Vladimir Despotović. "Metallurgy – Advances in Materials and Processes". In: 2012. Chap. 7 - Artificial Intelligence Techniques for Modelling of Temperature in the Metal Cutting Process. URL: http://dx.doi.org/10.5772/47850.

[19] Vojtech Pavlovsky. *Introduction To Artificial Neural Networks*. 2017. URL: https://medium.com/@temi.ayo.babs/multi-layer-perceptron-for-beginners-6aee246c6a03 (visited on 06/01/2019).

[20] Intel Lab's River Trail Project. *Bringing Parallelism to the Web with River Trail*. URL: http://intellabs.github.io/RiverTrail/tutorial/ (visited on 06/01/2019).

[21] Chen Kong and Simon Lucey. "Take it in your stride: Do we need striding in CNNs?" In: *Computing Research Repository* abs/1712.02502 (2017).

[22] Arden Dertat. *Applied Deep Learning - Part 4: Convolutional Neural Networks*. 2017. URL: https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2 (visited on 06/01/2019).

[23] Matthieu Cord. *Global Average Pooling in Deep ConvNets*. 2017. URL: http://webia.lip6.fr/~cord/pdfs/news/2017CordPoolingDeepNets.pdf (visited on 06/01/2019).

[24] Andrej Karpathy. *Stanford University Course CS231n: Convolutional Neural Networks for Visual Recognition*. URL: http://cs231n.github.io/convolutional-networks/#pool (visited on 06/01/2019).

[25] David Gschwend. "ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network". MA thesis. ETH Zürich, 2016.

[26] Forrest N Iandola et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size". In: *arXiv preprint arXiv:1602.07360* (2016).

[27] Gabriel Goh. *Why Momentum Really Works*. 2017. URL: http://distill.pub/2017/momentum (visited on 06/01/2019).

[28] Yann LeCun et al. "The Loss Surface of Multilayer Networks". In: *Computing Research Repository* abs/1412.0233 (2014).

[29] Jason D. Lee et al. "Gradient Descent Converges to Minimizers". In: *29th Annual Conference on Learning Theory*. Vol. 49. Proceedings of Machine Learning Research. PMLR, June 2016, pp. 1246–1257.

[30] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314.

[31] Michael A. Nielsen. "Neural Networks and Deep Learning". In: Determination Press, 2015. Chap. 4 - A visual proof that neural nets can compute any function. URL: http://neuralnetworksanddeeplearning.com/chap4.html (visited on 06/01/2019).

[32] David Gschwend. *Netscope CNN Analyzer*. 2016. URL: https://dgschwend.github.io/netscope/quickstart.html (visited on 06/01/2019).

[33] NVIDIA Corporation. *DIGITS: Interactive Deep Learning GPU Training System*. URL: https://docs.nvidia.com/deeplearning/digits/digits-user-guide/index.html (visited on 06/01/2019).

[34] Yangqing Jia. *CAFFE - Convolutional Architecture for Fast Feature Embedding*. URL: https://github.com/BVLC/caffe (visited on 06/01/2019).

[35] Ronan Collobert, Koray Kavukcuoglu, and Clement Farabet. *Torch - A Scientific Computing Framework for LuaJIT*. URL: http://torch.ch (visited on 06/01/2019).

[36] Google Brain Team. *TensorFlow - An open source machine learning framework for everyone*. URL: https://www.tensorflow.org (visited on 06/01/2019).

[37] Johanna Ene. *Johanna Ene's Blog - Det Hemliga Språket (The Secret Language)*. 2015. URL: http://www.tandskoterskan.net/det-hemliga-spraket/ (visited on 06/01/2019).

[38] image-net.org. *ImageNET Challenge*. URL: http://image-net.org/challenges/LSVRC/ (visited on 06/01/2019).

[39] David Gschwend. *ZynqNet Trained Model*. 2016. URL: https://github.com/dgschwend/zynqnet/tree/master/_TRAINED_MODEL (visited on 06/01/2019).

[40] Raghuraman Krishnamoorthi. "Quantizing deep convolutional networks for efficient inference: A whitepaper". In: *Computing Research Repository* abs/1806.08342 (2018).

[41] Pavlo Molchanov et al. "Pruning Convolutional Neural Networks for Resource Efficient Transfer Learning". In: *Computing Research Repository* abs/1611.06440 (2016).

[42] Guoliang Kang, Jun Li, and Dacheng Tao. "Shakeout: A New Regularized Deep Neural Network Training Scheme". In: *AAAI - Association for the Advancement of Artificial Intelligence*. 2016.

[43] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. "Distilling the Knowledge in a Neural Network". In: *NIPS Deep Learning and Representation Learning Workshop*. 2015. URL: http://arxiv.org/abs/1503.02531.

[44] Frederick Tung and Greg Mori. "CLIP-Q: Deep Network Compression Learning by In-Parallel Pruning-Quantization". In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. 2018, pp. 7873–7882.

[45] William Dally. *High-Performance Hardware for Machine Learning [Tutorial, Neural Information Processing Systems (NIPS)]*. 2015. URL: https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf (visited on 06/01/2019).

[46] Philipp Gysel. "Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks". In: *Computing Research Repository* abs/1605.06402 (2016). URL: http://arxiv.org/abs/1605.06402.

[47] Philipp Gysel. *Ristretto: Approximation Schemes*. 2018. URL: http://lepsucd.com/?page_id=639 (visited on 06/01/2019).

[48] Alpha Data Parallel Systems Ltd. *ADM-PCIE-KU3 User Manual*. June 2017. URL: https://www.alpha-data.com/pdfs/adm-pcie-ku3%20user%20manual.pdf (visited on 06/01/2019).

[49] Xilinx Inc. *UltraScale Architecture and Product Data Sheet: Overview*. 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf (visited on 06/01/2019).

[50] Vincent Dumoulin and Francesco Visin. *A guide to convolution arithmetic for deep learning*. 2016. URL: https://github.com/vdumoulin/conv_arithmetic/blob/master/gif/same_padding_no_strides.gif (visited on 06/01/2019).

[51] Xilinx Inc. *Vivado Design Suite - HLx Editions*. URL: https://www.xilinx.com/products/design-tools/vivado.html (visited on 06/01/2019).

[52] Xilinx Inc. *Introduction to High-Level Synthesis with Vivado HLS (2013.3 Version)*. 2013. URL: http://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/vivado_hls/VivadoHLS_Overview.pdf (visited on 06/01/2019).

[53] Xilinx Inc. *SDAccel Environment User Guide*. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1023-sdaccel-user-guide.pdf (visited on 06/01/2019).

[54] MathWorks Inc. *FPGA-in-the-Loop Simulation*. URL: https://se.mathworks.com/help/supportpkg/xilinxfpgaboards/fpga-in-the-loop-simulation.html (visited on 06/01/2019).

[55] Aditya. *How to interpret scikit's learn confusion matrix and classification report?* 2015. URL: https://stackoverflow.com/questions/30746460/how-to-interpret-scikits-learn-confusion-matrix-and-classification-report/30748053#30748053 (visited on 06/01/2019).

[56] Brandon Garcia and Sigberto Alarcon Viesca. "Real-time american sign language recognition with convolutional neural networks". In: *Proceedings of Machine Learning Research*. 2016, pp. 225–232.

[57] Vivek Bheda and Dianna Radpour. "Using Deep Convolutional Networks for Gesture Recognition in American Sign Language". In: *Computing Research Repository* abs/1710.06836 (2017).

[58] Byeongkeun Kang, Subarna Tripathi, and Truong Q. Nguyen. "Real-time sign language fingerspelling recognition using convolutional neural networks from depth map". In: *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)* (2015), pp. 136–140.

[59] Panagiotis G. Mousouliotis and Loukas P. Petrou. "SqueezeJet: High-level Synthesis Accelerator Design for Deep Convolutional Neural Networks". In: *Computing Research Repository* abs/1805.08695 (2018).

[60] Xilinx Inc. *Zynq-7000 SoC Data Sheet: Overview*. 2018. URL: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (visited on 06/01/2019).

[61] Intel Corporation. *Product Brief: Intel NUC Kits NUC7i3BNK and NUC7i3BNH*. 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/nuc-kit-nuc7i3bnh-nuc7i3bnk-brief.pdf (visited on 06/01/2019).

[62] Carl Lemaire, Andrew Achkar, and Pierre-Marc Jodoin. "Structured Pruning of Neural Networks with Budget-Aware Regularization". In: *Computing Research Repository* abs/1811.09332 (2018).

[63] Tianyun Zhang et al. "A Systematic DNN Weight Pruning Framework using Alternating Direction Method of Multipliers". In: *Computing Research Repository* abs/1804.03294 (2018).

[64] Shuiwang Ji et al. "3D Convolutional Neural Networks for Human Action Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35 (2010), pp. 221–231.

[65] ArduCAM. *OV7670 Camera Module*. URL: http://www.arducam.com/products/camera-breakout-board/0-3mp-ov7670/ (visited on 06/01/2019).

[66] ON Semiconductor. *MT9M114 Evaluation Board - User's manual*. URL: https://www.onsemi.com/pub/Collateral/EVBUM2524-D.PDF (visited on 06/01/2019).

List of Acronyms

AI Artificial Intelligence. v

ANN Artificial Neural Network. 1

CNN Convolutional Neural Network. 2, 13

FPS frames per second. 41

GPU Graphics Processing Unit. 2, 3

HLS High-Level Synthesis. iii

NNP Neural Network Processor. 3

ReLU Rectified Linear Unit. 8, 9, 12

RTL Register-Transfer Level. iii, iv, 41

SGD Stochastic Gradient Descent. 16, 17

TPU Tensor Processing Unit. 3

Appendix A

ZynqNet

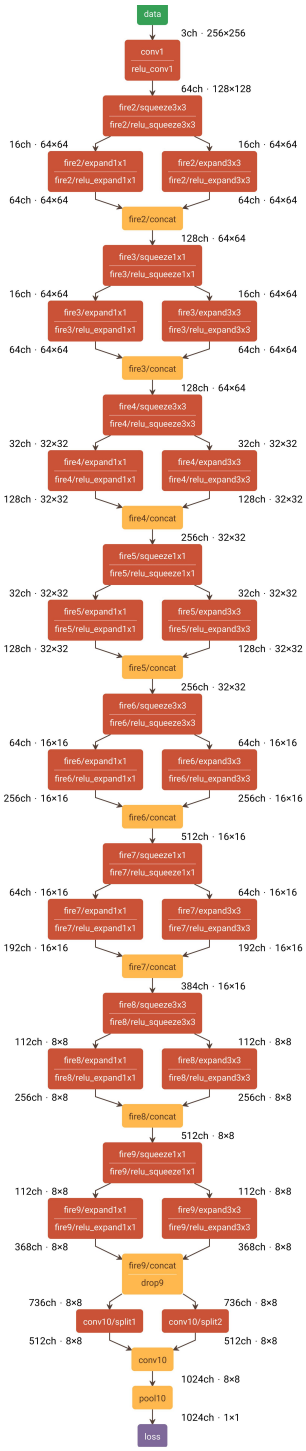

Figure A.1: ZynqNet architecture *(Source: Netscope CNN Analyzer [32])*.

Appendix B

Test Dataset

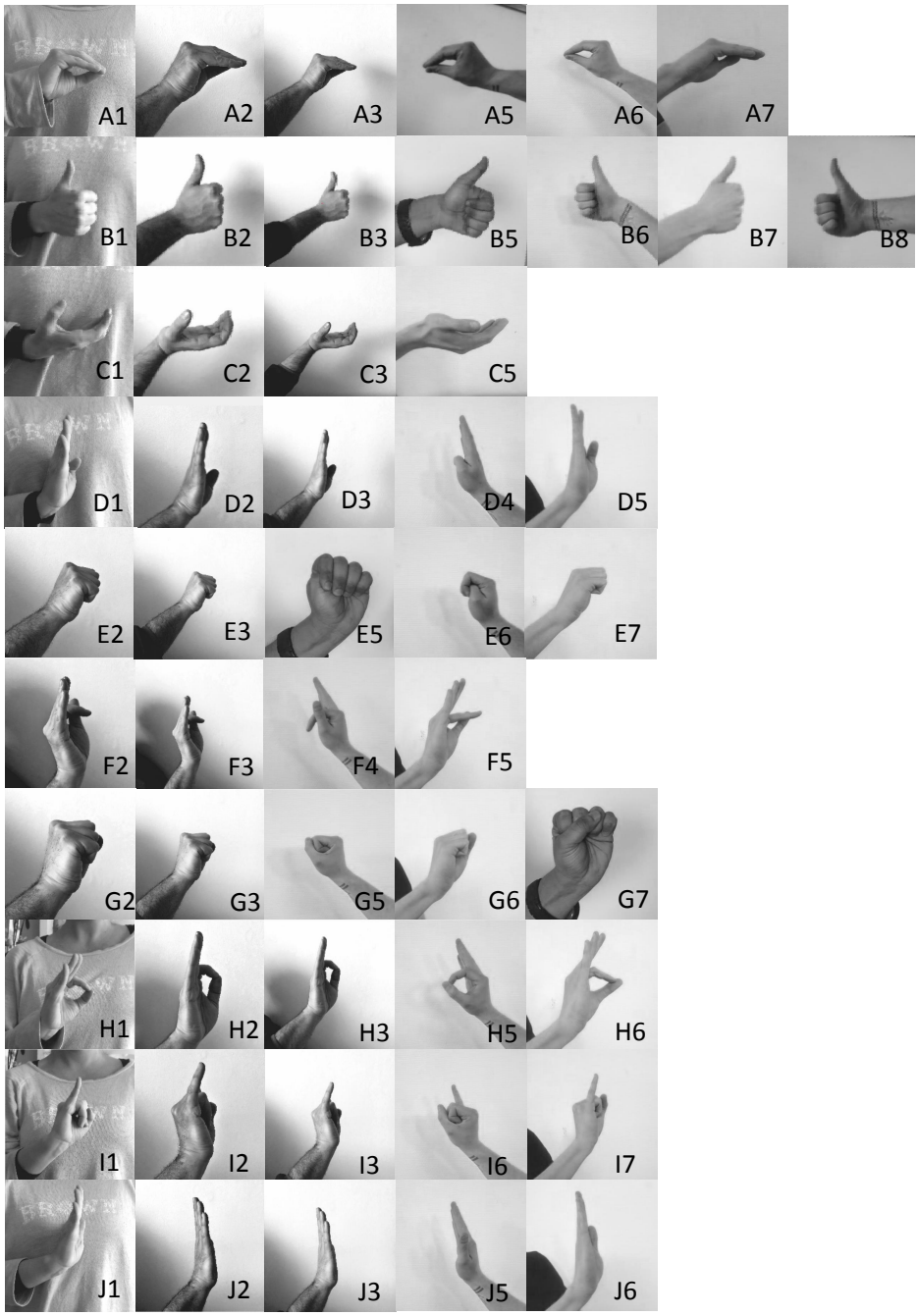

Figure B.1: Images from the test dataset (categories from A to J).