Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Optimising convolutional neural networks for super fast inference on focal-plane sensor-processor arrays

Author:
Benoit Guillard

Supervisors:
Dr Sajad Saeedi
Prof. Paul Kelly

This report is submitted in partial fulfilment of the requirements for the MSc degree in Advanced Computing of Imperial College London

September 2019

Abstract

Convolutional Neural Networks (CNNs) have revolutionised the Computer Vision discipline in the last few years. CNNs now are state of the art methods to solve almost all classification, segmentation, or detection tasks. In parallel, domain specific architectures have been developed for Computer Vision applications, and amongst others, a new form of hardware has emerged: Focal Plane Sensor Processors (FPSPs). FPSPs consist in merging the light sensor and the the processing unit of a traditional vision system, by enabling each photo-diode with rudimentary analog computation capabilities.

In this work, we implement CNNs on an FPSP, a goal previously pursued only twice to the best of our knowledge [49] [5]. To benefit from the low latency and energy efficiency of existing FPSPs, the main challenge is the limited register availability and the inaccurate nature of their computations. An in-depth FPSP-specific optimisation of all components constituting a CNN allows us to beat the previous baseline by a margin of more than 4%. Our **AnalogNet2** architecture reaches a testing accuracy of **96.9% on the MNIST dataset**, at a speed of **2260 FPS**, all for a cost of **0.7 mJ per frame**. We also experiment two techniques to implement multi-layer CNNs on an FPSP - quantisation and pooling. The resulting accuracy is however gravely hindered by noise, for which we provide a quantitative study. Finally, we prove the impact of this work on a real application, with a proof-of-concept that extracts steering directions from a scene, for a wheeled robot platform.

Acknowledgements

I express my warmest thanks to the following people, without whom this thesis and work could not have been possible:

- Professor Paul Kelly at Imperial College, for putting this project in perspective thanks to his great height of view, and the many very interesting suggested readings.
- Doctor Sajad Saeedi at Imperial College, for keeping a saving eye on my work when I was in relative independence, for his weekly guidance and precious advice that lead to this work I found so refreshing.
- Dr Jianing Chen at the University of Manchester, for his patience and in-depth technical explanations about the SCAMP5 vision system, when I was in trouble going from theory to actual implementations.
- Matthew Wong at Imperial College, for his great work on which mine is based, and the valuable access to his source code and behind-the-scene explanations.
- Martin Fisch at Imperial College, for his time and giving me access to the Perceptbot platform.
- Doctor Laurie Bose at the University of Bristol, for explanations on his implementations of CNNs on the SCAMP5 device.
- My family and closest friends, for their unconditional support.

A note on terminology

This report is about implementing programs on a particular class of vision systems, that we refer to as Focal Plane Sensor Processors (FPSPs). Specialists also designate FPSPs as Cellular Processor Arrays (CPAs), or Pixel Processor Arrays (PPAs). Semantic subtleties exist between FPSPs, CPAs and PPAs, but for the purpose of this report, we choose to use the term FPSPs to designate this class of hardware.

Code availability

Code for all experiments and implementations described in this report is available at `https://gitlab.doc.ic.ac.uk/bag1418/cnns-on-fpsps/`.

Contents

Chapter 1

Introduction

Classical image capture devices rely on two distinct hardware components: a sensor, that transforms incoming light into an array of analog voltages approximately proportional to the incoming photon counts, and a digital processing unit, that acts on a digitised version of this array. Due to this digitisation and data movement, this traditional approach achieves relatively slow frame rate and is quite power hungry. New imaging devices have been designed to tackle these issues. Focal-Plane Sensor-Processors (FPSPs) merge the two above steps, by allowing computation to take place directly on analog photon counts, on the sensor. An FPSP can be considered as a light-sensitive processor, or, equivalently, as a computation-enabled image sensor. FPSPs empower each pixel with an independent, low power, low memory Processing Element (PE). At each clock cycle, one instruction is broadcast to the whole pixel-parallel array of PEs. FPSPs thus belong to the Single Instruction Multiple Data (SIMD) family of computer architectures. Various researchers have successfully demonstrated their very low power and high throughput capabilities for low level Computer Vision tasks.

In parallel, in recent years, the availability of powerful computation hardware and our finer understanding of their underlying mathematical optimisation have allowed Deep Neural Networks (DNN) to beat state of the art methods for Artificial Intelligence (AI) problems. In many contexts, Convolutional Neural Networks (CNN) are nowadays the standard method to solve many image-related problems (detection, segmentation, classification...). These methods being very computationally intensive, a focus has recently been on efficient ways to run CNNs, regarding energy consumption, but also processing power and memory requirements. The fields of applications for lightweight CNNs ranges from mobile phone, low powered autonomous systems, to Internet of Things (IoT) devices.

Despite being very constrained in terms of memory, and only allowing for noisy computations because of their analog nature, FPSPs appear as good contenders for embedding high speed CNNs in very tiny power budget environment. To the best of our knowledge, only two attempts to do so have been successful. Both implement rudimentary CNNs for digit classification, using the MNIST dataset as a test bed. Technical details and a thorough study are only available for one of the two

attempts, and we will mostly rely on it. It achieved unprecedented speed and energy efficiency, at the price of a decrease in accuracy. We intend to try and port more optimised and complex CNNs to FPSPs, aiming towards useful real-life implementations.

The main contributions we present are organised as follows:

- We provide an in-depth review of the parts and program components needed to build CNNs on an FPSP. We detail their efficient implementations on SCAMP5, given the particular limitations of the device - in terms of available registers, noisy computations or limited digital performance.
- We put these efficient building blocks into application to improve AnalogNet, a previously-published single layer CNN running on the SCAMP5 device. Among other things, the resulting network, that we call AnalogNet2, uses an unrolled loop for the fully connected layers' computation, and a new output events' binning layout. It yields both an increased throughput and a better accuracy, and forms the new baseline with which we compare to when testing other CNNs.
- We report on the technical feasibility of two layer CNNs on an FPSP, using quantisation and pooling techniques to overcome hardware limitations. We demonstrate two possible implementations of two layer CNNs on the SCAMP5 device, none of them beating the single layer baseline set by AnalogNet2. The practical results we get suggest that computations are seriously hindered by noise.
- We design new experiments to quantify noise accumulation on the focal plane, and quantitatively show that circuit depth increase leads to very large amounts of noise. These experiments are standardised, and can be used to explore the design space of FPSPs for a less noisy future version of a SCAMP device.

The above contributions are based on academic test beds and theoretical performance assessment. To provide a real world use case for these findings, in Chapter 7, we develop a proof-of-concept in which an FPSP is used for robot navigation. Steering directions are directly extracted from a scene using a CNN, with a very low latency and power consumption.

Chapter 2

Background

In this chapter, we provide an overview of the concepts, methods and ideas on which our work is based. After presenting how FPSPs work and the one we have at hand, we describe Convolutional Neural Networks, and the recent efforts of embedding them in portable devices.

2.1 Focal-plane sensor-processors

Standard digital cameras are most often designed and built with image quality as one of the first criteria, to produce images that match human visual perception. This can be expressed in terms of colour fidelity, amount of noise, optical deformations, or resolution of the output images. The requirements for Computer Vision tasks might be different, since the ultimate goal here is rarely to produce high fidelity images, but rather to extract meaningful information from a scene. Traditional RGB cameras and images are however often used as inputs for computer vision methods.

Computers and algorithms might not require images to be visually appealing to the human eye to perform efficiently. This fact suggests efforts should be made towards a joint sensor, algorithm and processor design. This idea has recently started to emerge as a guideline for future works [15].

FPSPs are a good materialisation of this principle: a single silicon chip that is both an image sensor and an analog processor [51]. This allows for vision tasks to happen closer to where the image was captured, even before it is converted to digital values. This improves speed and energy efficiency, at the expense of introducing noise in the computations' results. One of the latest instance of such a vision chip is the SCAMP5, designed by Carey et al. at the University of Manchester [7]. We have one available for our study, and will be conducting experiments on it.

While our work aspires to contribute to the broad space of FPSPs program design, we here focus on some of the specific details of the SCAMP5 device. Despite only being one particular instance of an FPSP, it presents some general issues that are valuable to address.

2.1.1 SCAMP5 hardware

The SCAMP5 device is made of a traditional digital micro-controller, connected to a light sensitive analog processor chip (the vision chip). As described in the official online documentation [10], the micro-controller has two cores, referred to as M0 and M4. In our case, we write and compile C++ vision programs for the M0 core, while the M4 core is in charge of other miscellaneous tasks (such as interfacing).

The vision chip is composed of a grid of 256*256 pixel-processors, or Processing Elements (PEs). Arithmetic and logic operations executed in parallel on the PEs are written in special sections among the M0 source code. The M0 core sends corresponding instructions to the whole array of PEs.

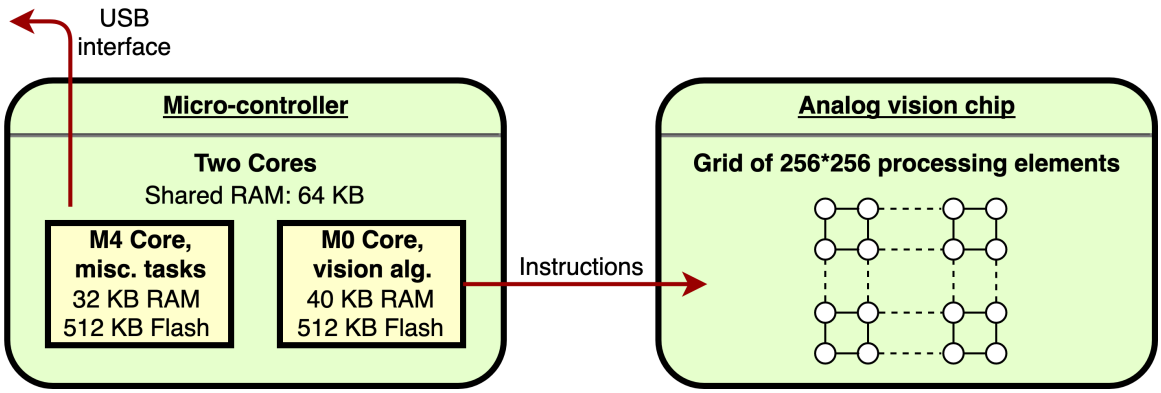


Figure 2.1: SCAMP5 vision system architecture

On the vision chip, each PE has a light sensitive diode, so the array functions as an image sensor. Each PE is enabled with an Arithmetic Logic Unit (ALU), that acts on its PE's memory registers, which are composed of:

- 13 digital registers, referred to as DREGs R0 to R12. Each of them stores a bit.
- 7 analog registers, referred to as AREGs A to F. Each of them stores a voltage, to represent values between -127 and +127.

On top of that, the PE array is also provided with a propagation network, allowing each PE to access values stored in its 4 neighbouring PEs' registers. These values transit through a specially reserved AREG, the NEWS register (for North East West South).

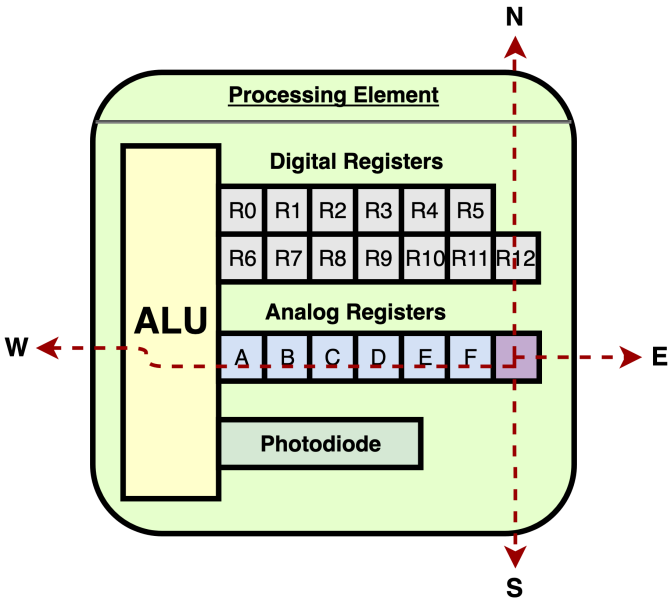


Figure 2.2: SCAMP5 Processing Elements registers.

2.1.2 SCAMP5 instruction set

The digital controller M0 can broadcast one instruction at a time to all PEs simultaneously. Each PE then independently executes it, on the data it holds locally. The computations are massively parallel, at the pixel level: as for every FPSP, the SCAMP5 device is a Single Instruction, Multiple Data (SIMD) computer.
The set of vision chip instructions is limited, and composed of three categories:

- *logical instructions*, such as *OR*, *AND*, or *NOT*, acting on DREGs. Their effect is pretty self-explanatory: for instance, the instruction *AND(C,B,A)* will perform a boolean *and* operation. Each of the 256*256 PEs will store the result of *and*-ing registers *B* and *A* into register *C*. This single instruction in fact leads to 256*256 *and* operations being carried out - one at each PE.

- *arithmetical instructions*, such as addition (*add*), subtraction (*sub*), or division (*div2*), acting on AREGs. Note that multiplications can only be performed by iterative additions, and divisions are only possible by a factor of 2. Moreover, contrarily to conventional processors, these arithmetical instructions are acting on analog values. For example, adding two values corresponds to physically adding the charges stored in two AREGs. These operations are both *imprecise* and *noisy*.

- *shifting instructions*, for PEs to access a value stored by of their neighbouring PE. For instance, *mov(A,B,west)* will store in a PE's AREG A the content of its west neighbour's AREG B ($A := B_{west}$). It is as if the 256*256 array of values stored in AREGs B was shifted **east** once, and the result stored in AREGs A.

Predication is allowed thanks to a special DREG, the FLAG register, that can be set

using the *WHERE* instruction. For every clock cycle, instructions are only performed by PEs whose FLAG register is 1. Section 4.2 provides an example for this.
In addition to the photo-diode, input values can also be provided by the micro-controller. There are instructions to either load a value in the -127,+127 range to an AREG, or draw simple shapes and patterns on DREGs. There also exist partial addressing instructions, to selectively load 1s in some PE's DREG, based on their coordinates on the focal plane (load 1s in DREG R5 of PEs whose x-coordinate is even, 0s everywhere else for example).
On top of that, readout operateven,ions can transfer digitised values of requested PEs' registers to the adjacent micro-controller. Whole arrays can thus also be transferred. DREGs can also be scanned for *events*, ie. coordinates of PE where the DREG is set to 1.

2.1.3 Performance and benefits over traditional vision systems

The idea behind an FPSP is to minimise data movement: computations happen on chip, in parallel, directly where each pixel was collected. The computation is said to happen *in the focal plane*. Power-hungry Analog to Digital (A/D) converters are only involved when a readout is required, and possibly on a subset of the PEs. Instead of slowly transferring each frame to a sequential processor, one can transfer the sparse result of a pixel-parallel computation on this frame.

A SCAMP5 device can, for instance, easily be turned into an event based camera, only outputting pixels that have changed between two frames. Classical event cameras such as the DAVIS240 [6] or the Gen3 ATIS [40] only record per-pixel intensity changes.
In [7], the authors demonstrate the capabilities of the SCAMP5 device by tracking a letter 'o' amongst distractors ('u') on a spinning wheel, at 100 000 frames per second (FPS), only outputting the coordinates of the letter for each frame. In [35], the authors compute the depth map of a scene using multiple frames taken through a lens changing focus at a very high speed. Only the depth map is transferred, but not every intermediate frame used to compute it. In a 2 Watts power budget, their system operates at up to 150 output FPS.

2.2 Convolutional neural networks

Artificial neural networks were originally proposed some time ago, but they have only recently been successful, becoming state-of-the-art methods for most Computer Vision tasks. We here intend to rapidly present their historical development and key ingredients.

2.2.1 Early neural networks principles

Inspired by a simplification of our brain's structure and the way neurons work, Artificial Neural Networks (ANN, or NN) have been around for a few decades. One

of the first implementations was the Perceptron, by Rosenblatt in 1957 [46], and it established general principles that are still in use nowadays.

Mathematical framework

As any machine learning technique, an artificial neural network can be seen as a function f_θ, that takes an input x and produces an output $\tilde{y}$ (most often vector data). θ is the set of the network's parameters, or *weights*. The network is composed of n layers f_1 to f_n, and each of them takes the previous layer's output as its input.

Let x_{i-1} be the vector output of the $(i-1)$-th layer ($x_0 = x$) of length k, K_i the $o * k$ weights' matrix of the i-th layer, B_i its *bias* vector of size o, and a_i the *activation function* of this layer. K_i and B_i both belong to the set of the network's parameters θ Then the i-th layer's output is a vector of size o, and its j-th component is:

$$\begin{aligned} x_{i,j} &= f_i(x_{i-1})_j \\ &= a_i\left(\sum_{l=1}^{k}(K_i)_{j,l} * x_{i-1,l} + (B_i)_j\right) \end{aligned}$$

Figure 2.3 shows a graphical representation of such a computation.

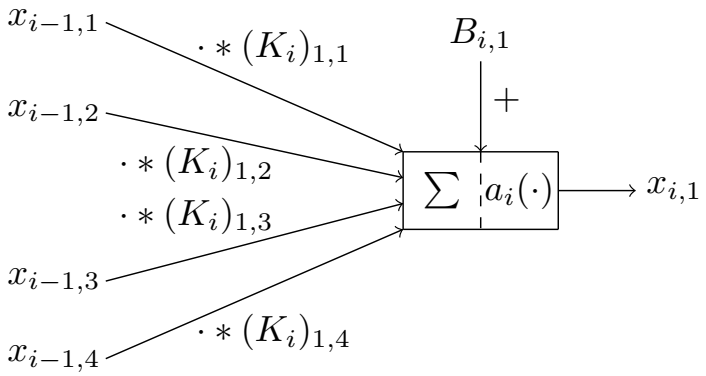


Figure 2.3: Computing the first activation of layer i, with layer $i-1$ having 4 components.

The computation of a fully connected layer can be represented as a matrix-vector multiplication. If we note A_i the point-wise application of a_i to a vector of length o, we have:

$$\begin{aligned} x_i &= f_i(x_{i-1}) \\ &= A_i\left(K_i * x_{i-1} + B_i\right) \end{aligned}$$

The output of the network can in fact be rewritten as $\tilde{y} = f_n\left(f_{n-1}\left(...f_1\left(x\right)\right)\right)$. Originally, typical activation functions were threshold functions. Each of the considered layers producing outputs whose components depend on every component of its input, through independent weights (only the bias is shared), they are called *fully connected layers* (see Figure 2.4).

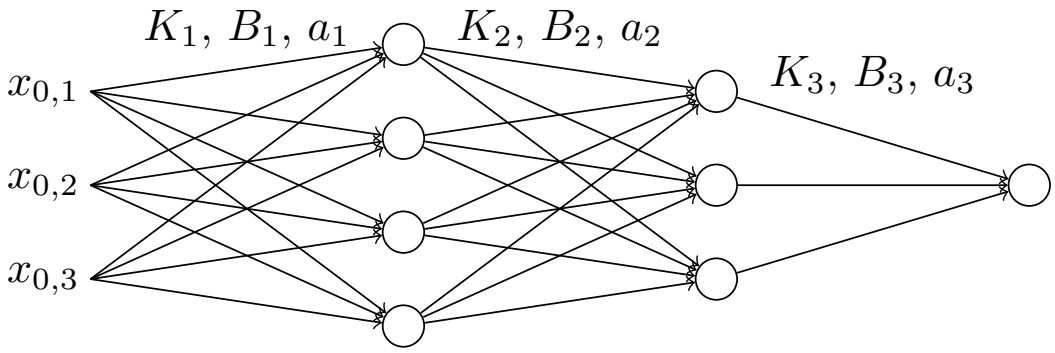


Figure 2.4: Schematic example of a fully connected network with 3 layers, respectively with 4, 3 and 1 node(s) or component(s).

The more layers l a network has, and the larger each layer is, the more complex is the ANN. It allows it to model more complex behaviour. These so called *hyper-parameters* are set by a human operator, expert in the machine learning field. Once the structure of the network is chosen, the parameter set $\theta = \{K_i, B_i | 1 \leq i \leq l\}$ must be chosen. This process is automated, for the final network to mimic the desired behaviour. During the training phase, the weights are optimised to satisfy a desired criterion.

Optimisation procedure

For a typical supervised learning problem, one has a dataset composed of a number of (input, label) pairs. A typical pair could be (written digit, digit value) [29]. During a *training phase*, the weights value are optimised for the network to output the correct label. As presented in [36], the general idea is to minimise a mathematical *loss function* that quantifies the current error of the network. By deriving the partial derivative of the loss value according to each weight, a so called *back-propagation algorithm* incrementally shifts each weight towards a value minimising the loss function.

In [19], the authors prove that in theory, with sigmoids as activation functions ($a(x) = \frac{1}{1+e^{-x}}$), a finite neural network can uniformly approximate any continuous function. This sets the theoretical guarantee that neural networks are suited for well defined supervised learning problems.

Worked minimal example of gradient descent

A whole ANN can be represented as a very large computational graph, composed of weights and inputs being used and combined in differentiable operations. As an example of how back-propagation works, Figure 2.5 shows a very minimalist computational graph.

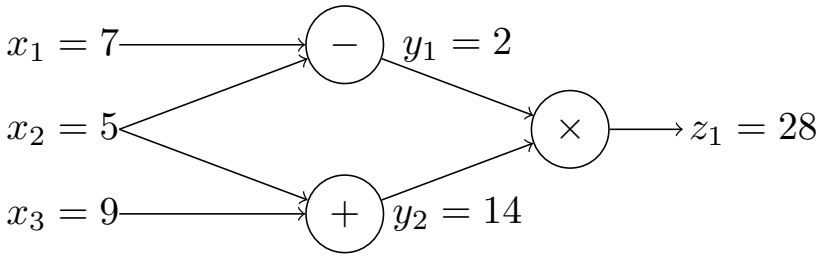


Figure 2.5: Minimalist computational graph, of which z_1 is the output.

During a typical training process, the computational graph gives a *loss* as an output, which measures how poorly the network performs in doing its intended task. In this example, if we define the output as z_1, the training process should minimise it. To this end, partial derivatives of the output according to all parameters are computed using the chain rule (see Figure 2.6). It is here crucial that the whole computational graph only uses differentiable operations.

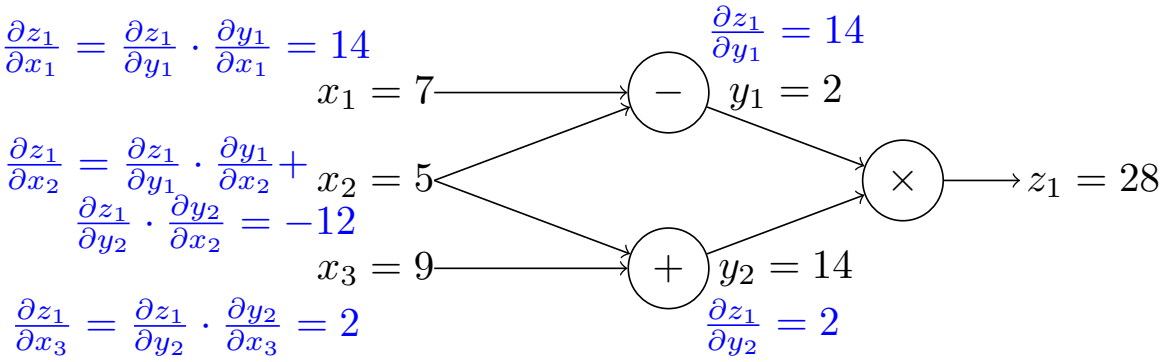


Figure 2.6: Computing the partial derivatives of the output according to all the parameters, in computational graph 2.5.

The use of the chain rule requires the value of the partial derivatives according to a nodes children to be available when the derivative according to this node is computed. For this reason, the graph is traversed backwards, from the final output value to the inputs - hence the name of *back-propagation*.
Once all these partial derivatives have been computed, weights can be updated to follow a gradient descent. The most straightforward and unsophisticated way of doing it is to subtract to each parameter the corresponding partial derivative, scaled by a constant step ϵ. In our case, with $\epsilon = 0.1$, the update would be:

$$
\begin{aligned}
x_1' &= 7 - 0.1 \cdot 14 = 5.6 \\
x_2' &= 5 + 0.1 \cdot 12 = 6.2 \\
x_3' &= 9 - 0.1 \cdot 2 = 8.8
\end{aligned}
$$

We can indeed verify that with these parameters as inputs, the output of the computational graph is now $z_1' = -9$, which is strictly inferior to $z_1 = 28$.

2.2.2 Convolutional neural networks

Due to the lack of computing power and effective algorithms for the training process, the early neural networks did not meet with tremendous practical success. Both these constraints have been enormously alleviated in the past few years. The democratisation of massively parallel and powerful processing units called General Purpose Graphics Processing Units (GPGPU, or GPU) and the development of frameworks such as Tensorflow [1] or Pytorch [39] that make use of these devices to efficiently train and run neural networks allowed almost anyone to tinker with simple forms of NNs. Secondly, new mathematical optimisation procedures such as Adam [27] or AdaGrad [17] allowed for a more stable and reliable convergence of the training process. This gave rise to the creation of Deep Neural Networks (DNNs) architectures, involving many more layers than previously.

The final ingredient that permitted DNNs to beat state-of-the-art methods in almost all computer vision related tasks was the introduction of convolutional layers. Inspired by traditional image processing, where convolutions are used for edge detection, sharpening and more, they consist in doing a convolution between a kernel and an image (see Figure 2.7).

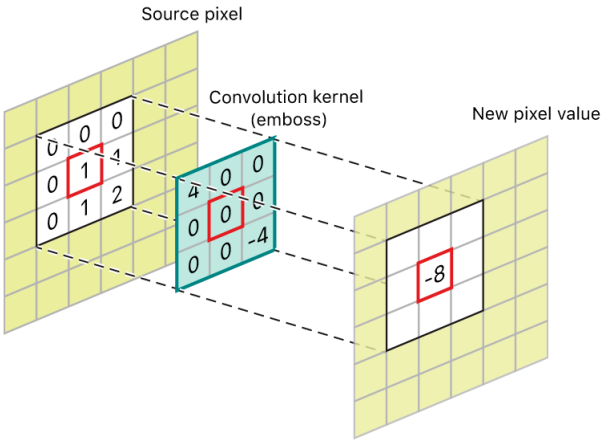


Figure 2.7: Creating an embossing effect on an image by convolving it with a kernel (from Apple Developer Documentation [4])

Used in DNNs, convolutional layers take advantage of the spatial coherency and structure of an image. What is being manipulated in a convolutional layer are no longer flat vectors, but rather arrays that explicitly represent images.
Using convolutional layers is also a great way to achieve weight sharing and thus reduce the total number of parameters of a network [26]. In such a Convolutional Neural Network (CNN), kernel values are the weights of the network, and a kernel is applied to all its input pixels with the same weights. Instead of multiplying each component (or 'pixel' in our case) of an input by an independent set of weights, the same kernel is applied to every input components: the transformation is convolutional. Moreover, since the spatial extent of a kernel is generally smaller than the 2D array it is convolved with, an output component typically depends of a small

number of inputs (and not all of them) that are spatially close: the transformation is local. Similarly to what is presented above, after this dot product operation, a bias is added before applying an activation function.

2.2.3 Typical architectures and ingredients of CNNs

Values transiting between convolutional layers are typically 4 dimensional arrays, and are called tensors. The dimensions of a tensor are usually annotated as `N`, `C`, `H` and `W` (or equivalently in the order `N`, `H`, `W` and `C`). The first dimension of a tensor, `N`, corresponds to the number of inputs being simultaneously computed. For each of these inputs, the 3 remaining dimensions form what is called a *feature space*, composed of `C` *feature maps*. `H` and `W` are the spatial extent (height and width) shared by all the feature maps in this feature space. A convolution layer transforms a feature space with $\texttt{C}_A$ feature maps to one with $\texttt{C}_B$ feature maps using $\texttt{C}_A\texttt{*C}_B$ convolution kernels (see Figure 2.8).

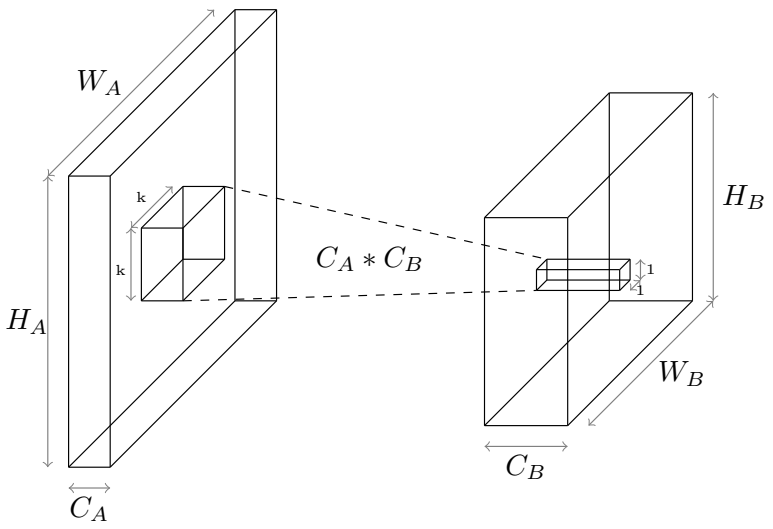


Figure 2.8: A `k*k` convolution layer, transforming a $\texttt{H}_A\texttt{*W}_A\texttt{*C}_A$ feature space into a $\texttt{H}_B\texttt{*W}_B\texttt{*C}_B$ one. Each block of size $\texttt{k*k*C}_A$ is vectorised, and multiplied by a matrix with $\texttt{C}_B\texttt{*(k*k*C}_A\texttt{)}$ weights, to generate one 'fiber' in the output layer: a vector of length $\texttt{C}_B$. The same matrix of weights is applied to every block of size $\texttt{k*k*C}_A$ in the input feature map.

What unequivocally marked the advent of CNNs as state of the art methods was the release of AlexNet [28] in 2012, that outperformed all other methods submitted so far for the ImageNet classification contest.

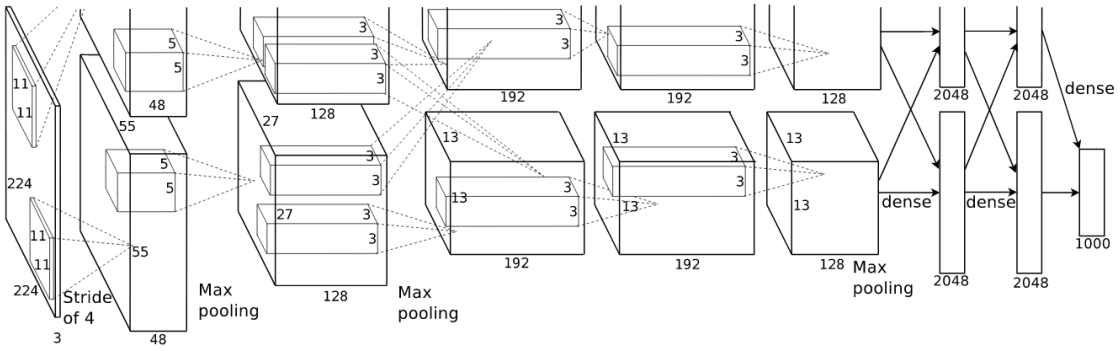


Figure 2.9: An overview of AlexNet's architecture, from [28]

As can be seen in Figure 2.9, AlexNet does not just use convolutional layers. This is generally true for CNNs, that most often also use the following ingredients:

- *pooling operations*, to reduce the spatial dimensions of feature spaces. Most of the time they are either averaging the values of each adjacent $k * k$ activations square to produce a single value (average pooling, or *AvgPool*), or taking the maximum of these $k * k$ activations (maximum pooling, or *MaxPool*).
- *fully connected layers*, as presented above, to map the result of successive convolution layers to the desired output space (a probability, a set of one-hot encoded labels, a single fixed-length vector...)

Some use cases such as image segmentation [31] require to output whole images. In these cases, fully connected layers are omitted, and the network is a so-called Fully Convolutional Network. Some architectures use upsampling layers called deconvolution layers, after having learned sparse representations of the input. This is useful for image segmentation [45], or auto-encoders that learn sparse feature representations of the inputs, and can be used as denoiser or as a part of a broader network. CNNs have also been used in the area of generative modelling [18] [41], but this might not be of interest for our present work.

2.3 Embedding CNNs

In view of the success of CNNs for numerous Computer Vision tasks, the need to implement them in power constrained environments without compromising for latency has naturally risen. Be it for smartphones, tablets, robots, or wearable devices, the number of possible use cases is very large. Two research areas are concurrently being pursued in that direction: devising CNNs that have lower computational cost by design, and devising specialised hardware.

2.3.1 Lighter CNNs design

In [22], the designers of MobileNet proposed a CNN using depthwise separable convolutions. Introduced in [47], depthwise separable convolutions consist of depthwise convolutions, followed by point-wise 1*1 convolutions (see 2.10). This way

of doing convolutions is not strictly equivalent to what has been described above, but greatly reduces the number of parameters and instructions. In addition to that, MobileNets make use of a lighter and thinner architecture than usual. By doing so, they get a model with fewer parameters (smaller model) and requiring fewer operations to operate (smaller complexity at inference time). The accuracy trade-off is very limited.

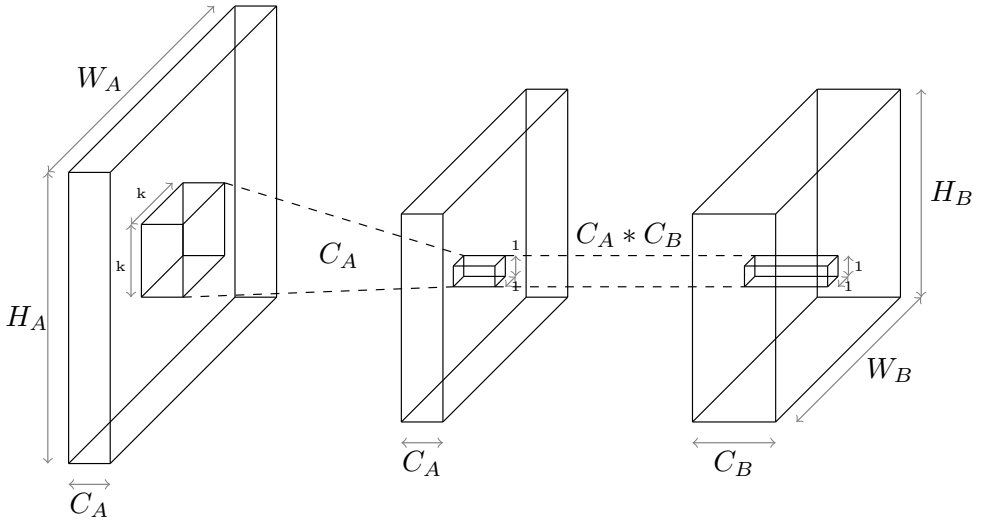


Figure 2.10: A `k*k` depthwise separable convolution layer, transforming a H_A*W_A*C_A feature space into a H_B*W_B*C_B one. Each feature map of a block of size `k*k*`C_A is convolved with an independent `k*k` kernel (`k*k*`C_A total weights), to generate one 'fiber' in the intermediate layer: a vector of length C_A. This fibre is then multiplied by a matrix with C_A*C_B weights to generate one fibre in the output layer: a vector of length C_B.

Another possible approach to reduce the memory footprint and optimise the inference latency of a CNN is to use lower precision (or quantised) values for its computation. Numerous attempts at it have shown an acceptable accuracy trade-off. Some approaches such as [20] or [52] quantise only the weights of the networks they are interested in, mainly for on-device storage and memory consumption reduction purposes.
Others, such as [14] or [24], develop whole mathematical optimisation frameworks that allow for both weights and activations to be stored in a reduced number of bits (only one in the case of [14]). In these two cases, they push their work further, and even propose an efficient computation model for their networks that also reduces inference latency. In the case of [14], this is in the form of a specialised GPU kernel, optimised to compute 1 bit convolutions. In the case of [24], this is in the form of an optimised integer-only matrix multiplication procedure.

2.3.2 Specialised vision hardware

There have recently been multiple attempts at developing specialised hardware for computer vision tasks, and especially for CNN inference. Globally, they all trade off generic computation capabilities for the following advantages in running CNNs:

- *energy efficiency*: a recent GPU used in a high end desktop PC draws tens of Watts. For example, an Nvidia GeForceGTX 1080Ti (the top of the line consumer grade GPU of the previous generation from Nvidia) draws around 10 Watts at idle, and more than 150 Watts under full load [3]. This cannot reasonably be powered in an embedded system. By giving up on generic computation capabilities that are of no interest in our case (3D shaders, raytracing...), and specialising it for computer vision and CNN inference purposes only, new devices use a far reduced amount of power for our use cases.

- *portability*: on the one hand, a desktop PC or a laptop cannot easily be fitted on an usual size robot or a drone, and even less in a wearable device. On the other hand, tinier micro-computers such as Raspberry Pi lack the necessary computing power to run CNNs at a decent frame rate. This opens the way for small sized computers embarking specialised computing power for CNNs.

- *reduced latency*: a practical solution to use the outputs of a CNN on a power and size constrained device is to rely on an auxiliary PC. A portable device can wirelessly stream the frames acquired by its on-device camera to a PC, then the PC can sequentially run a CNN on these frames, before sending the output of the network back to the device. The said PC can either be nearby, and form a local network with the device, or in the cloud. In that latter case, an internet connection is required, and privacy issues arise. Moreover, in addition to sacrificing the autonomous nature of such a device, latency is greatly increased by data transmissions, and is the main issue of this kind of setups.

We can identify two types of solutions to these problems. The first one consists in *accelerators chips*, that can be plugged into generic micro computers to enable them with advanced CNN computing capabilities. In that category fall Google's Coral Edge TPU [13], Intel's Neural Compute Stick 2 [23], the DaDianNao computer [11] or MIT's Eyeriss [12]. Referred to as Vision Processing Units (VPU), these hardware devices are designed with speed and power efficiency for CNN inference in mind. Each of them exploits different optimisation techniques such as limited data movements, data reuse, parallel computations, or specialised hardware operators for convolutions. As a result, when associated to low power micro-computers, the Eyeriss accelerator was demonstrated to run AlexNet at 35fps with a 278mW power consumption, and the Edge TPU to run MobileNet v2 at almost 400fps while being powered over USB only.

The second type of solutions consists in *integrated platforms*, providing both a general purpose micro computer and an accelerator chip (and sometimes even a camera) in a co-designed and bundled manner. In that category fall Nvidia's Jetson Nano [37], or the JeVois Smart Machine Vision Camera [25]. For example, the latter combines a mobile CPU, GPU and a camera in a power budget of 3.5W, and weighs less than 20 grams. Despite being quite lightly powered, it is fairly optimised, and can run a quantised version of MobileNet v1 at around 35 fps.

A typical application for these devices is to use them for robot navigation. The

authors of [43] use an Nvidia Jetson TK1 (previous generation integrated platform by Nvidia) to run a CNN that is directly controlling the direction of an autonomous wheeled robot. In [38], an ultra light quadcopter drone is controlled by a CNN outputting stirring commands based on the video stream acquired by its front facing camera. The CNN runs at around 10 fps, in a power budget of 7 mJ per frame, directly on an embedded accelerator that was specifically designed for this purpose.

Each one of these low power hardware implementations has its competitive advantage, be it low latency, minimal dimensions and weight, or small power budget. However, none of them is freed from A/D conversions nor from camera to processor data movements. Despite being far less generic because of their noisy computations, limited memory space and instruction sets, FPSPs do not suffer from A/D conversion's toll on latency and power efficiency. As a result, they still deserve a room in the space of efficient and specialised vision hardware.

As explained below, they have been demonstrated to successfully run primitive forms of CNNs, at a frame rate and energy per frame beating all of the above solutions. With some more improvement on the performances of the CNNs that can be implemented on current FPSPs, they could be very competitive. Some problems could greatly benefit from the advantages of analog computation, such as the ones requiring an extremely high frame rate (high speed robot navigation), or a very tightly restricted power budget (constrained on battery life, such as autonomous loiterer detection as in [8]).

We have here presented the new computing paradigm offered by FPSPs. We have also explained the present state of the art method of AI for imaging (CNNs), and the challenge to embed them in low-powered low-latency devices. In the next chapter, we show one way to tackle this challenge, and detail on how an FPSP can be used to run already trained CNNs.

Chapter 3

Literature review

In this chapter, we critically review and further explain then main pieces of work on which ours will be built. The first one is a kernel code generator for FPSP hardware. It will serve as a kind of compiler for our convolution kernels to run on the SCAMP5 device. Then we will thoroughly describe the structure and training process of AnalogNet, a CNN running on SCAMP5. Finally, we will quickly evoke a statistical noise model that has been built for the SCAMP5 device.

3.1 Kernel code generator

As explained in Subsection 2.1.2, the set of instructions available on the SCAMP5 device is very limited. In particular, divisions are only available by factors of two, and multiplications are only possible through iterative additions. On the other hand, as any FPSP, the SCAMP5 device presents the unusual ability to shift data in any of the four cardinal directions on a register, and by default uses the SIMD paradigm. These constraints being given, it seems quite arduous to convolve images with kernels in a traditional manner, despite convolutions being a typical operation in image processing and the key ingredient to CNNs.
In view of that, the authors of [16] present AUKE (AUtomatic KErnel code generation), a code generator for computing convolutions on FPSP hardware. For instance, for the $kernel$:

$$kernel = \begin{bmatrix} -0.25 & -0.25 & 0.25 \\ 0 & -0.25 & 0.25 \\ 0 & -0.25 & 0 \end{bmatrix}$$

the program in Listing 1 is convolving the content of AREG C, using registers D and E for intermediate computations.
Their ingenuous algorithm for code generation runs as follows:

1. Given a convolution kernel with generic floating point values as coefficients, it *approximates* it. Because of the above mentioned restrictions on multiplications and divisions, each coefficient is approximated as a sum of negative powers of 2, up to 2^{-D}, where D characterises the approximation depth (or precision) of

```
C = div2(C)
C = div2(C)
D = neg(C)
C = east(C)
C = add(C, D)
D = west(D)
C = add(C, D)
C = add(C, D)
D = south(D)
C = add(D, C)
D = east(D)
E = east(D)
D = add(D, E)
C = add(C, D)
D = north(D)
D = north(D)
C = sub(C, D)
```

Listing 1: Convolving the content of AREG C with $kernel$, using registers D and E for computations. We here use a standard notation for FPSP code, APRON. It is not the one in use in SCAMP5 code, but is trivially equivalent: *C=add(C,D)* translates to *add(C,C,D)* for example.

the process. For example, with $D = 3$,

$$k = \begin{bmatrix} 0.11 & 0.23 & 0.13 \\ 0.26 & 0.53 & 0.26 \\ 0.12 & 0.25 & 0.11 \end{bmatrix}$$

becomes

$$K = \begin{bmatrix} 0.125 & 0.25 & 0.125 \\ 0.25 & 0.5 & 0.25 \\ 0.125 & 0.25 & 0.125 \end{bmatrix} = \frac{1}{2^3} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

2. This approximate kernel is transformed into a sum of partial value representative (*SOPVR*), which is a multiset representation. It represents the set of relative coordinates of neighbours that have to be summed for each pixel to compute the convolution between the kernel and the image. For example, for the above

$$K = \frac{1}{8} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

we get the following representation (with matching colours):

$$\left\{ \begin{matrix} (-1,1), & (0,1), & (0,1), & (1,1), \\ (-1,0), & (-1,0), & (0,0), & (0,0), \\ (0,0), & (0,0), & (1,0), & (1,0), \\ (-1,-1), & (0,-1), & (0,-1), & (1,-1) \end{matrix} \right\}$$

The next steps operate on this representation, because each FPSP operation easily maps to a transformation on SOPVRs. For instance, the sum of two registers results in the union of their SOPVRs, etc.

3. A *reverse splitting* method generates a graph showing how to reach the initial state SOPVR ($\{(0,0)\}$) from the above final state SOPVR, by means of a succession of allowed operations, while reusing intermediate results. Heuristics guide this graph search, all while taking into account the limited number of registers on our device. The resulting optimal plan is then reversed for a way to construct the final state from the initial one.

4. A *relaxation* technique, inspired by circuit design, optimises the number of operations on the optimal plan above. Reducing the number of instructions allows for a faster execution, but most interestingly a less noisy one. Indeed, as each operation accumulates some noise, the less intermediate steps are involved, the more precise the result is.

5. A graph colouring algorithm is run on the obtained plan, as a mean of performing *register allocation*. It is constructed using the notion of liveliness, *ie.* the set of nodes required to compute a node (and itself).

The optimal nature of this method in terms of output instruction count is not proven. However, the authors demonstrate its validity by showing that the program length for a box filter of dimension d increases linearly with d, and not quadratically (as it would for a serial computation paradigm). Despite introducing noise because of the analog computations, the convolution of an image with a kernel greatly benefits from the pixel-parallel architecture. Compared to usual implementations on CPU or GPU, most experiments of convolving an image with standard filters ran by the authors show an improvement of between 180 and 2100 times less energy per frame, all while always being faster.
As a practical demonstration of the usefulness of their code generator, they implement a Viola-Jones face detector in an FPSP simulator. Because of the limited program size, they could not run all stages of the Haar cascade involved (they used 7 of the 25 stages). However, their implementation is still performing reasonably well, and achieves a classification rate, recall and precision comparable to the OpenCV implementation restricted to 7 stages. The estimated speed of this face detector on the SCAMP5 does not compete with a high end Intel processor (22.77ms estimated on the SCAMP5, 4.42ms measured on an E5-1630), but is highly competitive when it comes to energy per frame (28.0mJ estimated on the SCAMP5, 221.9mJ measured on an E5-1630). Without their generator, they would have had to manually code each one of the almost 50 convolution kernels they use - which is a highly non-trivial task to do for a human programmer.

3.2 AnalogNet

Based on the now available kernel code generation, the authors of [49] train and implement a rudimentary CNN, AnalogNet, that ultimately runs on the SCAMP5

device. They aim at solving the MNIST digits classification task.

3.2.1 Architecture

Figure 3.1 shows a schematic diagram of AnalogNet's architecture. Because of the limited number of registers available, their architecture is very light: three 3*3 convolution kernels acting on a single channel input image, followed by the addition of a bias for each of the 3 obtained feature maps, the application of a ReLU function, and a 9*9 sum pooling on the focal plane, in an analog manner. The result of this computation is then outputted to the adjacent digital micro-controller, that runs a 50 nodes fully connected layer, followed by a 10 nodes one, whose output is the network's output.

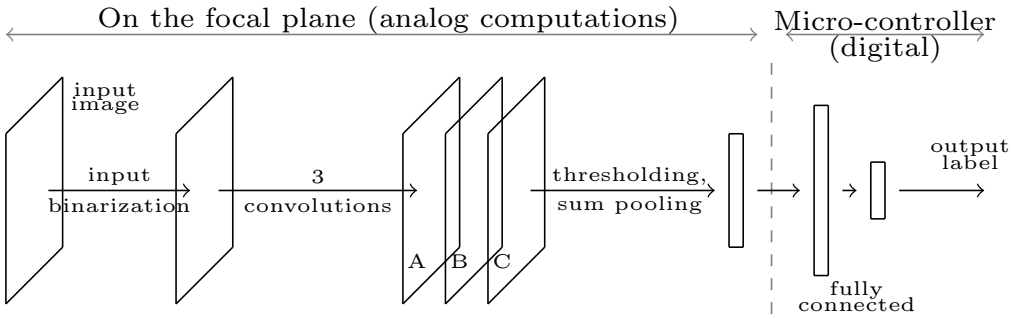


Figure 3.1: AnalogNet Architecture

Input binarisation

In order to remove the dependency on the environmental lighting conditions, the authors binarize the input image in the focal plane. The original exposure is read from the photo-diodes, and transferred to an AREG - A for instance. There, the image is stored using the full possible range of analog values, from -127 to +127. To remove any constraint on the exposure of the digit compared to its background, the image is binarized.
It is expected that the digits are displayed in black, on a white background. Using this fact, each PE's AREG A is compared to a fixed threshold: if the value stored in A is inferior to this threshold, the PE stores a 1 in its DREG R7, a 0 otherwise (see Listing 2).

```
scamp5_get_image(A);
scamp5_in(D, threshold_value);
scamp5_kernel_begin();
  add(A, A, D);
  CLR(R7);
  where(A);
    OR(R5, FLAG, R7);
  ALL();
  NOT(R7, R5);
scamp5_kernel_end();
```

Listing 2: Acquiring an analog image to AREG A, and negatively binarizing it to DREG R7.

As a result, DREG R7 contains a binarized negative of the original image: pixels belonging to the digit are marked positively (see Figure 3.2).

Figure 3.2: Input binarization. Left: original analog image (AREG A). Right: corresponding binarized image (DREG R7).

Running the convolutions

The binarised input image is then restored from R7 to AREGs A, B, and C, using a fixed mapping such as 0 being restored as 0, 1 being restored as 120. Each of the 3 convolutions is sequentially applied, each time using AREGs D and E to store intermediate results that are needed during the computation (see Listing 1 for an example). AREG F is reserved for some instructions that cannot be executed in-place. For instance, a division by 2 in fact splits the charges of a source register between two target registers, and hence needs a 'garbage' register in addition to its destination register. AREG F is called the *hardware workaround register*. Eventually, all AREGs are occupied, despite only 3 convolutions being computed (see Figure 3.3).

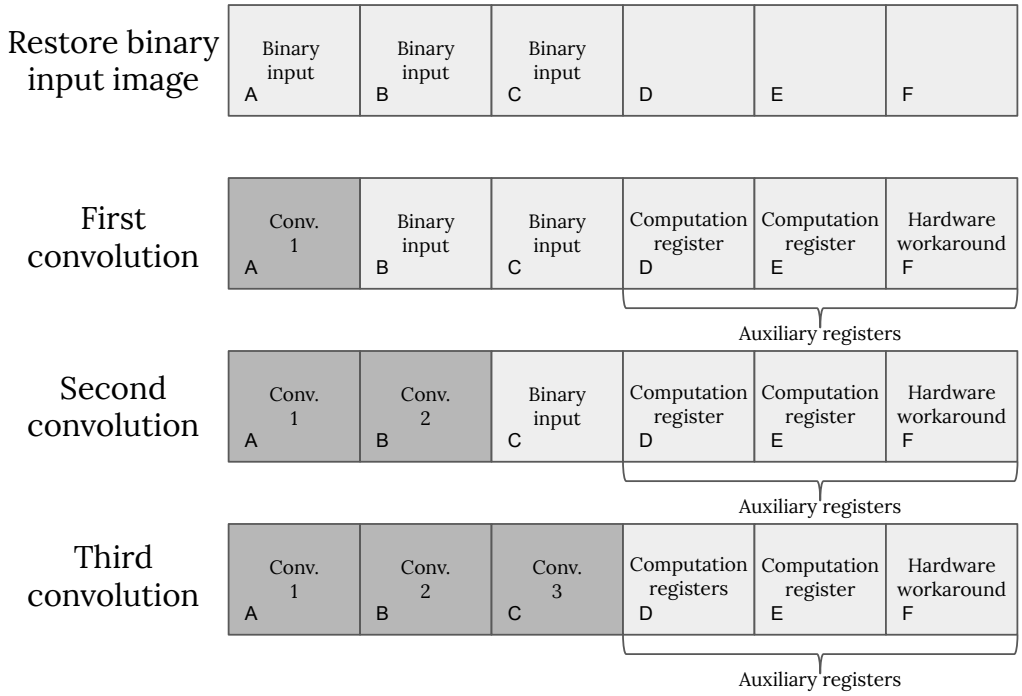


Figure 3.3: Sequentially computing the 3 convolutions in AREGs A, B and C, using AREGs D, E and F. Steps are represented in chronological order, from the top to the bottom.

Outputting to the micro-controller

At this stage, AREGs A, B and C each contain an analog feature map. The content of these 3 feature maps needs to be vectorized and transferred to the digital micro-controller M0 to compute the fully connected layers.

To this end, each feature map is first thresholded. Only strong activation values are kept, and the PEs where register A is superior to a fixed threshold store a 1 in their DREG R7 (see Figure 3.4). A similar process is run on AREGs B and C, whose binarized versions are respectively stored in DREGs R8 and R9.

Figure 3.4: Output binarization. Left: result of the first convolution, analog feature map (AREG A). Right: corresponding binarized feature map (DREG R8).

A SCAMP5 primitive is then used to gather *events* from these DREGs: the coordinates of the PEs where one DREGs is set to 1 is sent to the micro-controller. The maximum number of events to be gathered is set to 100 per feature map. These coordinates are then binned as shown on Figure 3.5 to form 3 vectors of size 9 (one per feature map). These 3 vectors are concatenated and passed as a single vector of size 27

to the first fully connected layer. These operations are equivalent to binarizing the feature maps, and sum-pooling them in 9*9 regions.

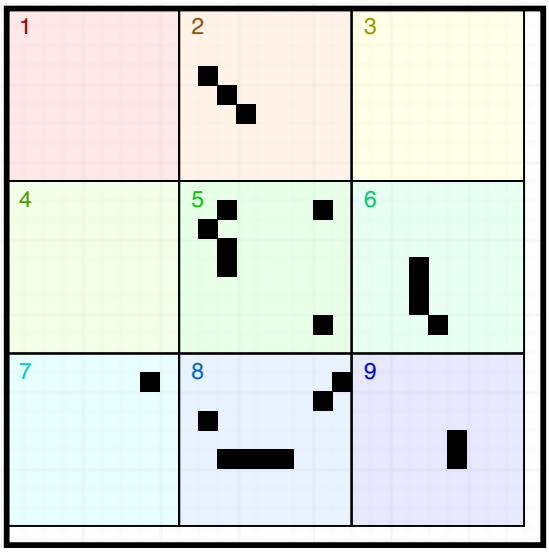


Figure 3.5: Event binning process. Events fall into nine 9*9 bins, and the last row and column are discarded. Black pixels correspond to a possible events configuration: in this case the output vector is $(0, 3, 0, 0, 6, 4, 1, 7, 2)$.

This process is successful in minimising data movement from the focal plane to the digital micro-controller. As a result, for a 28*28 input image (standard MNIST digit), only (up to) 3*100 2D integer coordinate values are transmitted from the focal plane to the micro-controller, where the fully connected layers computations is carried on, as standard matrix-vector multiplications. These coordinates correspond to locations where significant features are detected on the focal plane.
This *division of labour*, as the authors call it, allows for the fully connected layers to be computed noiselessly on the digital micro-controller, at the price of minimal data transfers. The fully connected layers could otherwise have been computed in the focal plane, by transforming them into 1*1 convolution layers (as in [31]). This would however have required complex data movements in the focal plane (the 27 values would have been considered as 27 1*1 feature maps), not benefiting from pixel-parallel computations (a different set of weights is applied to each value), and introducing additional noise.

3.2.2 Training

To account for all the specificity of this unusual architecture, an atypical training process had to be devised. It is sequentially rolled out in phase, using the framework Keras (with a Tensorflow backend):

1. A standard training process is run: using the Adam optimiser to minimise cross-entropy between the network's outputs and the ground truth labels. The only unusual bit here is the binarisation of input MNIST digits, that are normally 8

bits greyscale images (values ranging from 0 to 255). Here, to take account of the target hardware, the input images are thresholded, and only take 2 possible values (0 or 120). This first phase produces standard network weights for both the convolution kernels and the fully connected layers.

2. In view of the rounding operation that is applied by AUKE (see Section 3.1), not all convolution kernel weights are possible. In a second, separate training process, a new regulariser is added to force kernel weights to take values as close as possible to 'acceptable ones'. This means that a new loss is added to what the Adam optimiser is minimising: in addition to cross-entropy, a periodic differentiable function of the kernel weights now contributes to the total loss. This regulariser, based on a cosine function, is minimal when the kernel weights are closer to multiples of 2^{-D}, with D being the approximation depth ($D = 2$ in their implementation). As a result of this *custom regularisation* re-training, kernel weights converge close to multiples of 0.25. The two fully connected layers are not subject to this constraint, and are hence free to change values to take into account the convolutions modified results. After this re-training, kernel weights are rounded to their closer 0.25 multiple and frozen.

3. At this point, two specific constraints have not yet been taken into account, namely the noise introduced by the convolutions' computations, and the thresholding that takes place just before data is transmitted to the micro-controller. To account for these two effects, the kernels codes are generated by AUKE, and loaded on the SCAMP5 device. Placed in front of a PC monitor that sequentially displays the whole MNIST dataset, the SCAMP5 device performs all the computations that happen on the focal plane (from input binarisation to output thresholding, see Figure 3.1). For each input digit, the resulting 27 values are sent to a nearby computer, and aggregated to form a new dataset. This freshly acquired dataset corresponds to 'mid-computation' data of the network. It is then utilized to train only the two fully connected layers on a computer, since training is not technically possible directly on the micro-controller.

At this point, the full training process is done, and the weights can be loaded on the SCAMP5 device.

It is worth mentioning that the authors have also developed a way to account for analog noise during training without having to actually run the convolutions on the focal plane. Taking place right after step 2., their *Noise in the loop* technique consists in converting the convolution kernels to SCAMP5 instructions using AUKE, and adding simulated synthetic noise accordingly for each instruction, directly in the Keras framework. Despite being much simpler than having to capture a new data set as in 3., their technique relies on a synthetic noise model that is very crude, and does not give interesting results in reality. It is thus not put into application.

As a side note, step 2. is an unusually simple quantisation technique compared to what has been cited above (such as in [52]), but it works well (the testing accuracy

of the network reportedly remains at a value around 93% before and after). This might be a benefit of the very simple one-layer architecture that is used, and the relative independence of the weights being quantised.

3.2.3 Performance

To assess the performance of their implementation of AnalogNet on the SCAMP5 device, they compare it to the plain digital implementation of a similar network architecture in Keras. The digital version does not include any hardware specificity of the SCAMP5 device, except for the input binarization (no restriction on the kernels' values, no noise, no output thresholding). Except for speed and power consumption, the digital version runs similarly on a CPU (Intel Core i7-4930K 3.40GHz), a GPU (NVIDIA GTX 1080), or a VPU (Intel Movidius Myriad 2 Neural Compute Stick, in combination with a Raspberry Pi Zero).

In terms of accuracy, they report a 93.16% testing accuracy for their digital version (independently of the hardware it is run on), a 92.65% accuracy for the SCAMP5 version at 15 fps, and 90.2% at 3000 fps.
In terms of latency, they report an inference time of more than 2000 μs for the digital version running on a CPU, a GPU, or a VPU. Meanwhile, running at 3000 fps on the SCAMP5 device achieves an inference time of around 330 μs.
In terms of power consumption, they report a 18.9 mJ per frame energy consumption on the CPU, 88.6 mJ on the GPU, 3.69 mJ for the VPU, and only 0.567 mJ on the SCAMP5 device.

Despite not being very competitive in terms of its accuracy, an AnalogNet running on a SCAMP5 camera achieves state of the art inference speed, and in a very competitive restricted power budget.

3.3 Noise model

In the last part of their work, the authors of [49] conduct a statistical evaluation of the noise introduced by each analog instruction on the focal plane of the SCAMP5 device. Using a dataset containing the results of the running of 753,000 SCAMP5 instructions, they devise a noise model consisting in the sum of:

1. A Systematic Error Model, polynomially modelled, and fitted with regression. This term is responsible for taking into account consistent and systematic shifts in the results of analog instructions.

2. A Random Error Model, modelled as a sum of Gaussian kernels, fitted using Kernel Density Estimation. This term is responsible for taking into account the random variations in the results of analog instructions, with a stochastic model.

Using the above method, they create one parametrised noise model per analog instruction. Despite not including any spatial dependency, structure, or thermal contribution, it is constructed using real data and could be used when and if a noise estimation is needed.

There already exists a tool-chain to train, simulate and implement a rudimentary CNN on the SCAMP5 device. Our work builds on this foundations, to bring improvements in performance and functionality. In the next chapter, we add new CNN primitives to our collection, and describe how some already existing ones can be optimised.

Chapter 4

Adapting CNN components

As presented in Section 3.1, we can build on previous work to generate SCAMP5 kernels for the convolutional layers involved in CNNs. If we wish to enrich the architectures of the CNNs we run on the SCAMP5, we have to craft the corresponding new building blocks. In this chapter, we present the construction of such small parts that are later used. Practical aspects of the prototyping and training processes are also evoked. We believe these to be useful for future works.

4.1 Streamlining AUKE's output

AUKE [16] can process convolution kernels to produce generic FPSP code to execute them. Being generic and aimed at all FPSPs, it is not specific to the SCAMP5 vision system we have at hand. As a consequence, the code it generates is expressed in CSIM or APRON, which are FPSP simulators' languages, not accounting for any hardware specificity. There exist differences between both these languages and the standard SCAMP5 library.
Some of these differences are just syntactic sugar. For instance, shifting instructions in CSIM are of the form

$$west(B, A);$$

to shift the content of AREG A to AREG B, whereas the SCAMP5 library requires it to be written as

$$mov(B, A, west);$$

Other differences concern semantically incorrect instructions. For instance, many operations cannot be executed in-place on the SCAMP5 system, whereas CSIM does not have any such restriction. As an example,

$$neg(A, A);$$

is a perfectly valid CSIM code which negates the content of AREG A. On the SCAMP5 device, this must be rewritten using the hardware workaround register into

$$mov(F, A); neg(A, F);$$

Similar workarounds must be used for the *add(target, source1, source2)* and *sub(target, source1, source2)*, whose *target* and *source2* cannot be similar. Likewise, CSIM code includes an in-place division instruction,

$$div2(A, A);$$

whereas SCAMP5 code needs two 'trash' registers to split the content of an AREG:

$$diva(A, E, F);$$

Previously, the work of rewriting CSIM into SCAMP5 code was done manually. This was feasible for the three kernels involved in an AnalogNet. This however turns into a laborious obligatory step for each kernel of larger networks, or even iterated versions of small ones. We thus believed it valuable for our prototyping process to streamline this part, using automated scripts for efficiency purposes.
To this end, we wrote an overlay to AUKE, consisting in two successive parts:

1. a Python script, which ensures all instructions can validly be executed on a SCAMP5 device. When an invalid operation such as inplace division is encountered, it is replaced by an appropriate SCAMP5 shim.

2. a set of C++ macros, to handle minor syntactic discrepancies between AUKE and the SCAMP5 library. Listing 3 provides one example for this.

```
#define south(X,Y)  {mov(X,Y,south);}
#define neg_inplace(X,Y)({\
  mov(F,Y);\
  neg(X,F);\
})
```

Listing 3: Portion of the C++ macros to interface AUKE with SCAMP5.

This work is focused on practicability only, and allows us to directly copy and paste code from AUKE's output to M0 source code. It preserves the generic nature of AUKE, all while enabling us to quickly prototype.

4.2 ReLU and leaky ReLU

A very common activation function used in CNNs is the Rectified Linear Unit (ReLU) one:

$$a_{ReLU}(x) = x \cdot \mathbb{1}(x \geq 0)$$

Pretty simple to compute, it consists in clipping negative values to 0, and provides very satisfying results. ReLU can be considered as the standard activation function for CNNs, with many famous architectures relying on it, such as AlexNet [28], or ResNet [21]. As used in [49] for AnalogNet, there exist pretty straightforward implementations of the ReLU activation function for AREGs. The code in Listing 4 is

one of them. It simply performs a comparison on the AREG values, and replaces negative values by 0.

```
scamp5_in(F, 0);
neg(E, A);
where(E);
  mov(A, F);
all();
```

Listing 4: Implementation of a ReLU activation function on AREG A, using auxilliary AREGs E and F.

A ReLU activation function however has what can be considered as a flaw: its derivative for $x < 0$ is equal to zero:

$$\frac{\mathrm{d}a_{ReLU}(x)}{\mathrm{d}x} = \mathbb{1}(x \geq 0)$$

As a consequence, there is no effective back-propagation (see the worked example in Subsection 2.2.1) for neurons that are not activated, and all their previous ones in the computational graph. Corresponding weights are not updated for these 'dead neurons', and this can be seen as a defect of the learning process.
For this reason, a very similar activation function was crafted, but with a non-zero derivative for negative values. Called the leaky ReLU (or LReLU, [33]), it has a small slope for negative values. This slope is a fixed parameter α, with typical values between 0.01 and 0.2:

$$a_{LReLU}(x) = x \cdot \mathbb{1}(x \geq 0) + \alpha \cdot x \cdot \mathbb{1}(x < 0)$$

This time, the derivative is nowhere equal to zero:

$$\frac{\mathrm{d}a_{LReLU}(x)}{\mathrm{d}x} = \mathbb{1}(x \geq 0) + \alpha \cdot \mathbb{1}(x < 0)$$

This eliminates the issue of dead units during back-propagation. LReLU is commonly used in modern CNN architectures, such as in UNets to denoise images [9].
It can also be implemented easily on the focal plane of an FPSP. Using the SCAMP5 device, α is however restricted to negative powers of 2. The code in Listing 5 shows how it can be done, for $\alpha = 0.25$.

```
neg(D, A);
where(D);
  diva(A,E,F);
  diva(A,E,F);
all();
```

Listing 5: Implementation of a $.25$ leaky ReLU activation function on AREG A, using auxilliary AREGs D, E and F.

The community's opinion on ReLU versus LReLU is not definitive [50]. There is no absolute advantage of LReLU on ReLU, with the latter performing slightly better in some contexts. In our case, despite the division being reputed as imprecise, LReLU might yield some advantages in passing down information from a layer to subsequent ones. We propose to test it in our implementations of multi-layer CNNs on the SCAMP5.

4.3 Pooling

As presented in Subsection 2.2.3, it is common to use pooling operations in CNNs. A pooling operation is used to reduce the height and width dimensionality of the feature maps, by locally combining neighbouring activation values. Such clusters of values - typically 2*2 or 3*3 squares - are each summarized into a single activation that is fed to the next layer. The down-sampled feature map is more synthetic and robust to small changes in the input image.

As stated before, only single convolutional layer architectures have been implemented on the SCAMP5 vision system. As a result, no such pooling operation has been designed for it. We here present our implementation of the two most common pooling operations.

4.3.1 Average pooling

In an average pooling operation, clusters of neighbouring activation values are summarized into their average. We have implemented an average pooling operation that runs on the focal plane, and acts on clusters which are squares of size 2*2. It uses the standard SCAMP5 instructions of division by 2, shifting and summation.

```
// Divide the content of A by 4
diva(A,E,F);
diva(A,E,F);
// Sum along x-axis
mov(F, A, west);
add(A, A, F);
// Sum along y-axis
mov(F, A, south);
add(A, A, F);
```

Listing 6: Implementation of a 2*2 average pooling on AREG A, using auxiliary AREGs E and F. As a result, each PE in the new AREG A contains the average of the 4 previous values of the 2*2 square of which it is the top right corner.

As a result of algorithm in Listing 6, each value in the new AREG A is the average of the previous values of the 2*2 square of which it is the north east component.

Figure 4.1 depicts this process: all the content of the AREG is processed, and each PE's AREG A corresponds to a local average.

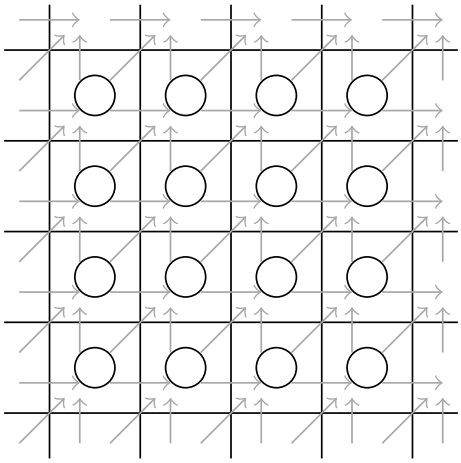


Figure 4.1: Average pooling process. Upon execution of algorithm in Listing 6, each PE's AREG contains the average of the old values of the three PE's pointing to itself and itself.

The AREG to which the pooling operation is applied with code in Listing 6 can of course be changed from A to any other AREG, as can the auxiliary AREGs E and F. More interestingly, the shifting directions can also be changed. The code in Listing 6 uses directions *west* and *south*, which computes the average in the top right corner, as shown in Figure 4.1. The other cardinal directions can be used, to consequently change the final location of the pooled value in each 2*2 square. This point is summarised in Table 4.1. The reader is reminded that shifting directions can have a counter-intuitive effect, as explained in Subsection 2.1.2.

x-axis shifting direction	y-axis shifting direction	final position of the pooled value
east	*north*	bottom-left
east	*south*	top-left
west	*north*	bottom-right
west	*south*	top-right

Table 4.1: Influence of the pooling directions on the location of the pooled value in each 2*2 square.

With this method, *every* 2*2 square is averaged, with the result stored in one of its corners. For our purpose, not all PE's AREGs are of interest: what matters is only the average of *disjoint and adjacent* 2*2 squares, forming an exhaustive cover of the whole feature map, with no repetition. To get the 2*2 average pooled version of the original feature map, one must keep only one quarter of the locations, and the rest can be discarded.
To achieve this, we use the *partial addressing* capabilities of the SCAMP5 device, mentioned in Subsection 2.1.2. With one M0 instruction, we load a sparse chequer-

board pattern in a DREG - loading 1s to PEs whose x and y coordinates are both even, 0s everywhere else, see Figure 4.2. We use this method to discard values in the above averaged feature map, at locations marked as 0s in the chequerboard like DREG. As a result, we get a 2*2 average pooled feature map, that is sparsely distributed on an AREG.

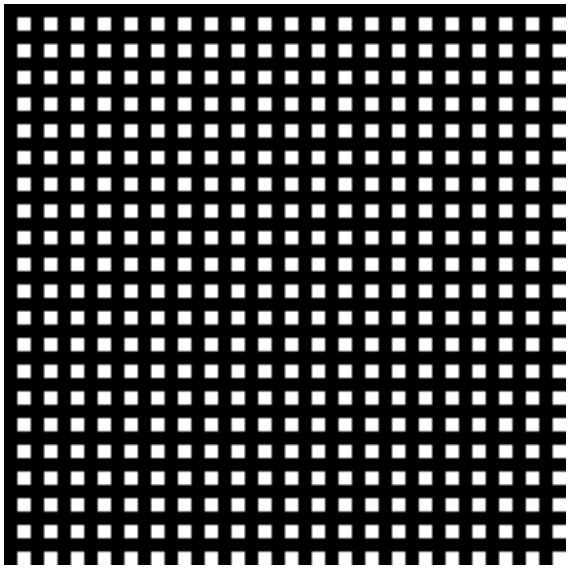


Figure 4.2: 42*42 crop of the content of one DREG, after executing one *scamp5_load_pattern* instruction. One quarter of the PEs are selected, based on the parity of their coordinates, to get this sparse chequerboard pattern.

On such an average pooled feature map, only one quarter of the PEs store a relevant information on the corresponding AREG. To take advantage of this sparsity, we can store up to 4 pooled feature maps on a single AREG. By shifting the sparse chequerboards and partially copying AREGs' content, we can interleave multiple feature maps on one AREG. Table 4.2 shows a schematic example of it. As an alternative to shifting pooled feature maps, results can also be directly computed to the right location by changing the pooling directions, as explained in Table 4.1.

.	.	.	.	.	.	.	.
.	A	B	A	B	A	B	.
.	C	D	C	D	C	D	.
.	A	B	A	B	A	B	.
.	C	D	C	D	C	D	.
.	A	B	A	B	A	B	.
.	C	D	C	D	C	D	.
.	.	.	.	.	.	.	.

Table 4.2: Four feature maps can be interleaved on one single AREG. A, B, C and D schematically represent four different feature maps.

Interleaving sparse feature maps requires very few shifting instructions compared to densely grouping each feature map in one region. This helps in reducing the number

of instructions used in our programs, and hence the resulting amount of noise. As explained in Section 4.4, computations can still easily be done on interleaved feature maps, again using partial addressing.

4.3.2 Maximum pooling

In addition to doing average pooling, a standard way of reducing the dimensionality of a feature map is through maximum pooling, also known as *max-pooling*. In this case, the greatest activation value of each input cluster is chosen as its representative in the pooled feature map. For instance, a 2*2 max-pooling only preserves the greatest of the 4 values in each adjacent 2*2 square.
Though a little bit more complex than average pooling, max-pooling can also easily be implemented on a SCAMP5 vision system. Using shifting instructions, local subtractions, comparisons to 0 and conditioned additions, we designed the code in Listing 7 as our implementation of max-pooling.

```
// Pooling along x-axis
sub(F, A, west, A);
where(F);
  add(A, A, F);
all();
// Pooling along y-axis
sub(F, A, south, A);
where(F);
  add(A, A, F);
all();
```

Listing 7: Implementation of a 2*2 maximum pooling on AREG A, using auxiliary AREG F. As a result, each PE in the new AREG A contains the maximum of the 4 previous values of the 2*2 square of which it is the top right corner.

Again, after executing the code in Listing 7, the whole content of the AREG corresponds to local maxima. As presented in Subsection 4.3.1, we use partial addressing to select relevant locations only, and interleave up to 4 sparse feature maps on one single AREG.

4.4 Convolutions on pooled data

As a result of the pooling operation presented in Section 4.3, feature maps are sparsely stored on AREGs. Values that are neighbours in the pooled feature map are in fact separated on the focal plane, by one row and one column of data belonging to another feature map (see Table 4.2). The traditional implementation of a convolution on the focal plane is therefore no longer relevant and must be adjusted.

Conceptually, applying a 3*3 convolution kernel on a sparse chequerboard feature requires to transform the kernel into a sparse 5*5 one. For example, the kernel

$$k_3 = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

is transformed into

$$K_5 = \begin{bmatrix} a & 0 & b & 0 & c \\ 0 & 0 & 0 & 0 & 0 \\ d & 0 & e & 0 & f \\ 0 & 0 & 0 & 0 & 0 \\ g & 0 & h & 0 & i \end{bmatrix}$$

This transformation can easily be done as a preprocessing step, before feeding the kernel to AUKE. The kernel explicitly imitates the spacing of the feature map to account for it. The corresponding output of AUKE can directly be applied to an AREG that contains a sparse feature map, whose content will correctly be convoluted with the kernel.

We developed another approach to tackle the issue of convolution on sparse feature maps. Inspired by [34] super-PE technique, we consider each 2*2 square of the pooled feature map as a group forming a virtual super Processing Element. Each super-PE is equipped with with four times more registers than a normal PE. Figure 4.3 shows an example of this grouping scheme.

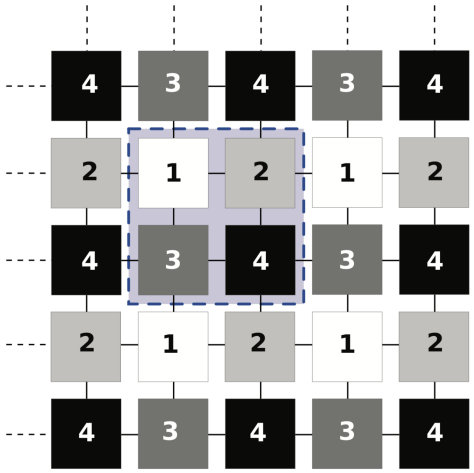


Figure 4.3: Grouping 4 PEs to form a super-PE. Figure from [34].

This data layout is similar to the one we create by interleaving feature maps (as in Table 4.2), and it presents the same trouble: neighbouring values are instead separated by one PE. Computations cannot be carried as usual. The authors of [34] developed a double-shifting procedure to replace traditional shifting instructions for super-PEs. We implemented them on the sCAMP5 vision system, using an auxiliary AREG. As presented in Listing 8, the idea is to shift the content of the original AREG

twice, but only copy the content of one selected position (1, 2, 3 or 4). This set of instructions are encapsulated in macros, for a greater ease of use.

```
// Assuming DREG R5 contains a sparse chequerboard-
// like pattern, with ones at positions 1 only.
// This can be achieved with:
scamp5_load_pattern(R5, 0, 0, 254, 254);

scamp5_kernel_begin();
  // Shift the whole content of A twice, to AREG F.
  mov(F,A,west);
  mov(F,F,west);
  // Selectively copy shifted values at position 1 only
  WHERE(R5);
    mov(A,F);
  ALL();
scamp5_kernel_end();
```

Listing 8: Double shifting the content of A in position 1 only. This is equivalent to shifting once for super-PEs, and is normally called through a *double_west(A);* macro.

Replacing traditional shifting instructions by these double shifting instructions allows to almost transparently use super-PEs as if they were normal PEs. The new, sparse data layout is abstracted. Some simple macros and the overlay to AUKE presented in Section 4.1 take care of transforming the SCAMP5 code for a 3*3 kernel to one that uses double shifts. It only remains for the developer to specify which position in the super-PE (1, 2, 3 or 4) the kernel is applied to, i.e. to which feature map. The code in Listing 9 shows the structure of such a kernel.

```
// Assuming DREG R5 contains a sparse chequerboard-
// like pattern, with ones at positions 1 only.
WHERE(R5);
  /* AUKE's output for the desired 3*3 kernel, in which shifting
   * instructions are replaced by double shifting instructions.
   * For example: */
  double_north(A, A);
  div2_inplace(A, A);
  div2_inplace(A, A);
  double_west(B, A);
  add(A, A, B);
  double_south(B, B);
  double_south(B, B);
  add(A, A, B);
ALL();
```

Listing 9: Example of a kernel being applied to a sparse feature map, stored in position 1 on AREG A. Thanks to the double shifting instructions and partial addressing, other values (in position 2, 3 and 4) in AREG A are preserved.

The main, and most important improvement of this super-PE technique over the transformation of a 3*3 kernel into a 5*5 sparse one is that it is not destructive for other values. With these double shifts, data is preserved in PE that are on the same AREG plane, but not affected by the kernel that is currently computed. To make it clearer, in Figure 4.3, a 3*3 convolution kernel can be applied to values in position 1, with values in position 2, 3 and 4 kept unchanged. This allows for a more efficient use of AREGs, since we can compute results in-place, without destroying neighbouring PEs' content.

Most of the time, AUKE gives results on sparse 5*5 that are similar to 3*3 in which we replace shifting instructions by double shifting instructions. However, we found it faster and more reliable explicitly use the double shifting trick. The search tree of AUKE is indeed smaller in this case, and less subject to undesirable pruning. Moreover, with explicit double shifting instructions, data conservation is guaranteed for locations where no kernel is applied, despite being on the same AREG.

The restricted memory footprint of computing a kernel with these super-PEs makes some room for new optimisation. For instance, if we have 4 interleaved feature maps stored in AREG A, we can compute one feature map of the next convolutional layer by:

1. Applying kernel k_1 to position 1 in A, using auxiliary registers D, E and F for computations. Values at positions 2, 3 and 4 in A are preserved.

2. Applying kernel k_2 to position 2 in A, using auxiliary registers D, E and F for computations. Values at positions 1, 3 and 4 in A are preserved.

3. Applying kernel k_3 to position 3 in A, using auxiliary registers D, E and F for computations. Values at positions 1, 2 and 4 in A are preserved.
4. Applying kernel k_4 to position 4 in A, using auxiliary registers D, E and F for computations. Values at positions 1, 2, and 3 in A are preserved.
5. Summing the four activation values in each super-PE.

In this process, AREGs B and C are not used, and can each store other persistent feature maps. There is no spare space on AREG A during these computations, and we take advantage of their in-place nature to efficiently work out a new convolutional feature map.

4.5 Quantisation

As seen on Figure 3.3, after computing three feature maps of a single layer CNN, there is no more available AREG for subsequent layers computations on the SCAMP5. Register availability is a very strong constraint in designing CNNs for such an FPSP.

To free some AREGs for computations, we propose to store intermediate results in DREGs. Values typically stored in an AREG, in the $[-127, 127]$ range can be digitised, and stored on multiple bit values in DREGs. For a full resolution quantisation, 8 bits are required to store these 255 possible values. However, given that computations are imprecise and noisy in AREGs, we suggest an approximated quantisation could be enough. Storing a small number Most Significant Bits (MSB) only decreases the number of required DREGs, with a toll on precision that might not even be perceptible.

The binary representation of an AREG can easily be computed on the focal plane, by iterative subtractions and comparisons to fixed thresholds. This process can be stopped at the desired precision level. Listing 10 presents such an example. To restore to an AREG a digitised content stored across multiple DREGs, only one addition per DREG is required.

Figure 4.4 shows an example of the lossy quantisation of an AREG to 4 DREGs: one for the sign, 3 for the 3 MSB (corresponding to values 64, 32 and 16).

If we assume all the AREG values are positive - which is typically the case after applying a ReLU on the result of a convolution - it is possible to cut down the number of used DREGs by one: there is no purpose in storing a constant sign. Figure 4.5 displays such an example of the unsigned lossy quantisation of a positive AREG to three DREGs. The corresponding code is shown in Listing 10.

Figure 4.4: Capture of a quantisation on 4 bits processed on the focal plane of the SCAMP5 vision system. Top-left: acquired input image. Top right: DREG storing the sign. Bottom right: DREG storing the first Most Significant Bit. DREGs storing the second and third MSBs are not shown here. Bottom left: reconstructed image.

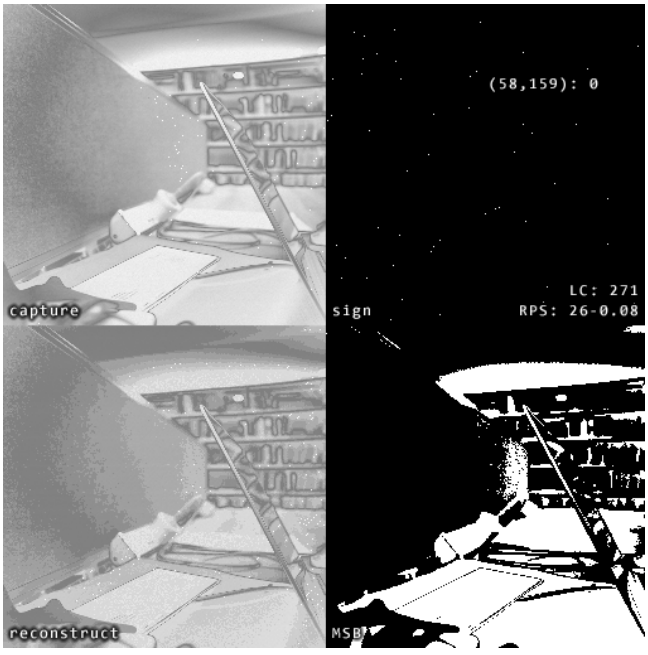


Figure 4.5: Capture of an unsigned quantisation on 3 bits processed on the focal plane of the SCAMP5 vision system. Top-left: absolute value of an acquired input image. Top right: DREG storing the sign: not in use here. Bottom right: DREG storing the first Most Significant Bit. DREGs storing the second and third MSBs are not shown here. Bottom left: reconstructed image.

```
/* Second MSB in target1 */
scamp5_in(E, 64);
scamp5_kernel_begin();
  CLR(target1);
  abs(D,source);
  sub(F, D, E);
  where(F);
    MOV(target1, FLAG);
    mov(D, F);
  all();
scamp5_kernel_end();
/* Second MSB in target2 */
scamp5_in(E, 32);
scamp5_kernel_begin();
CLR(target2);
  sub(F, D, E);
  where(F);
    MOV(target2, FLAG);
    mov(D, F);
  all();
scamp5_kernel_end();
/* Third MSB in target3 */
scamp5_in(E, 16);
scamp5_kernel_begin();
  CLR(target3);
  sub(F, D, E);
  where(F);
    MOV(target3, FLAG);
  all();
scamp5_kernel_end();
```

Listing 10: Unsigned and approximated quantisation of AREG *source* to 3 DREGs: *target1*, *target2* and *target3*. *target1* stores bits corresponding to the first MSB (value of 64), *target2* to the second MSB (value of 32) and *target3* to the third MSB (value of 16).

We now have code to quantise the content of an AREG to multiple DREGs. This quantisation can be signed or unsigned, and can be done at a chosen precision level. We intend to use it to store intermediate results of 2-layer CNNs, using unoccupied DREGs to our profit.

4.6 Output thresholding

As explained in Subsection 3.2.1, output convolutional feature maps of AnalogNet are thresholded, to create binary activation maps. These are in turn sent to the digital micro-controller in the form of an events' list, for the computations of the

fully connected layers. This output binarisation is effective in reducing data transfers from the focal plane to the digital micro-controller. This however establishes a difference with traditional CNNs: with AnalogNet, output feature maps consist in binary activations. This characteristic will also be shared by our other CNN architectures, since this is the best (to our knowledge) way not to transmit full AREG copies to the digital micro-controller.

In this section, we detail how it is possible to account for this specificity during the training process, and why a manual adjustment is still needed after training.

4.6.1 Simulating output thresholding during training

In the legacy AnalogNet training process, the output thresholding (or binarisation) is only taken into account once the convolution kernels have been frozen and loaded onto the SCAMP5 device, preventing them to account for it. We here propose to include the effects of output binarisation during the training process, and before the network weights are frozen. We let the Adam optimiser find the most suitable way for convolution kernels to account for it.
All operations applied to the input of a CNN to get the corresponding output must be differentiable, to allow for back-propagation and training (see Subsection 2.2.1). Thresholding is not differentiable as such, and thus we must approximate it. To this end, we use a sigmoid function as the activation function of our output convolutional layer, instead of the ReLU function. The sigmoid was standardly used as an activation function of DNNs, before being replaced by simpler and more efficient ones. In our case, it however provides a differentiable manner to simulate binarisation, in a standard way:

$$sigmoid_\alpha(x) = \frac{1}{1 + e^{-\alpha * x}}$$

The steepness of the sigmoid function can be parametrised by α, and as seen in Figure 4.6, a high value for α is nearly indistinguishable from a standard binarisation. The thresholds values are represented by the biases added before applying the convolution.

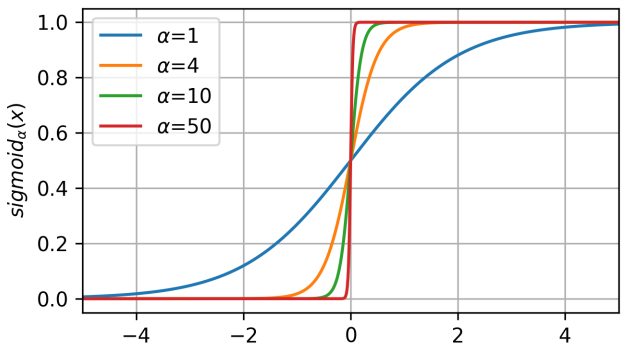


Figure 4.6: Activation function $sigmoid_\alpha$, for different α values.

In our new training process, the parameter α is set to 1 for the first training epochs. After a few standard training epochs, α is increased, to a high value such as 50. Eventually, the network can be tested without simulating the binarisation by a steep sigmoid, but with a genuine thresholding.

During our tests, we found out that the final testing accuracy does not seem to depend on the order custom regularisation (presented in Subsection 3.2.2) and output binarisation are applied during training. Results also seem quite insensible to the way the sigmoid's steepness is increased, whether it is brutally or progressively changed.

Using this technique, we are able to train CNNs whose outputs are binarised, in an end-to-end manner in Tensorflow. The training does not involve any simulator nor the actual SCAMP5 camera. Used along with the custom regularisation technique of [49], the produced CNNs take every hardware specificity of our FPSP into account, except the noisy and imprecise nature of its computations.

The resulting testing accuracy of the AnalogNet architecture in Tensorflow was 91.1%. As such, it cannot really be compared to any baseline, since this is the first digital implementation of AnalogNet (in the sense that it runs on an actual computer) that accounts for output binarisation.

4.6.2 Adjusting thresholds to noise

With the above method, the Adam optimiser finds optimal threshold values, i.e. one value per output feature map above which a pixel is considered as activated. These proposed thresholds however do not take noise into account, and for this reason, they need to be manually adjusted once the kernels are implemented on the camera.

We found out that the most efficient way to do it is to plug the camera to a host computer, and adjust through sliders in the host application. A good rule of thumb is to choose a value that removes noisy spiking activation, and preserves feature localisation around the desired object (a digit with the MNIST test-bed). See Figure 4.7 for an example.

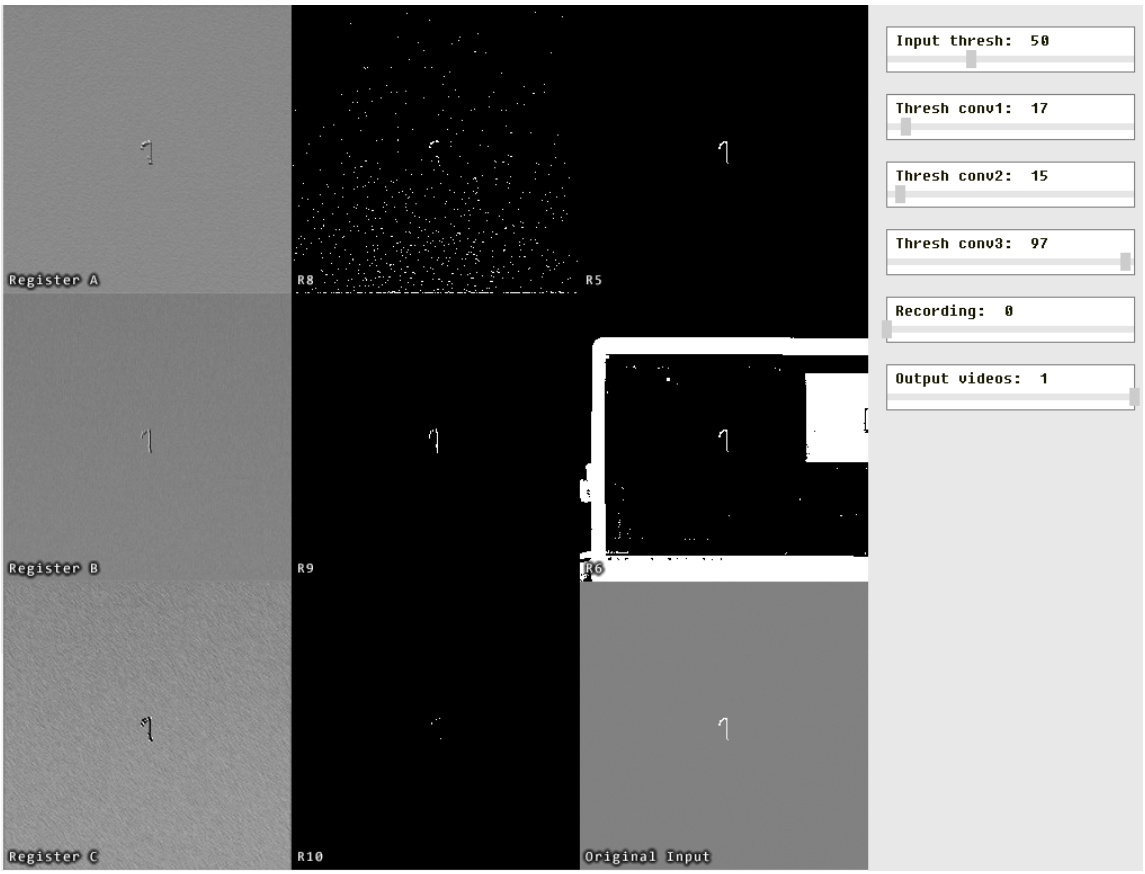


Figure 4.7: Capture of the SCAMP5 host program. The first column shows the content of AREGs A, B and C, storing three different feature maps. The second column shows the thresholded feature maps, according to the threshold values on the sliders. The third column shows the input image at different pre-processing stages. In this example, the first threshold value is too low (we notice noisy activated pixels in the binarised feature map in DREG R8), the second seems correct (in R9), while the third seems too high (there is almost no activated pixel in the binarised feature map in R10).

This step, despite being manual, is not very laborious - there is only one slider to adjust per output feature map. Pre-computing useful values for the thresholds would require having a precise noise model, which is not available at the moment.

4.7 Noisy data acquisition

The CNN weights given by a digital training process cannot account for the noise induced by analog computations. For this reason, and as explained in Subsection 3.2.2, once the network is trained with Tensorflow, we load the kernel weights onto the SCAMP5 vision system and gather noisy retraining data. This consists in showing each digit class (0 to 9), and set (testing and training) to the vision system, and recording the outputs of the convolutional part. With these flat vectors, we retrain the fully connected layers, so that they can account for noise.
The authors of [49] place the camera in front of a computer's screen and display

all the instances of one digit and one set at a time. Using the host program, they save one file per class/set pair, i.e. 20 files - one of them for instance corresponds to the testing '5's, another to the training '7's. This process requires the action from a human operator after each digit has finished recording, to save the file and launch the display and recording of the next digit. This was needed to correctly label data and know which digit was displayed when which output was acquired. The whole process takes approximately one day of intermittent action, and the fully connected layers can easily be retrained on this labelled data.

Judging this process too heavy and hindering the ease of prototyping, we propose to automate it and to carry out the full MNIST dataset capture in one run, without requiring an human operator.
Since there is no way to simply synchronise the display program and the SCAMP5 camera, the main issue is to label acquired data. To solve this, the classes and subsets are displayed in a pre-determined order, and separated by an empty screen lasting 30 seconds. The display program for example shows the digits in the following order:

1. training '0's;
2. empty screen of 30 seconds;
3. testing '0's;
4. empty screen of 30 seconds;
5. training '1's;
6. empty screen of 30 seconds;
7. ...

38. empty screen of 30 seconds;
39. testing '9's.

In addition to recording the vectors corresponding to the binned events (see Subsection 3.2.1), for each output we also record a single value corresponding to the number of positive pixels in the input image. A low number (between 0 and 2) means the camera acquired an empty screen; an amount greater than 10 implies that a digit was shown. Figure 4.8 displays the evolution of this count during an acquisition process. Using this indication combined to the predefined order in which the digits are shown, it becomes straightforward to label training data.

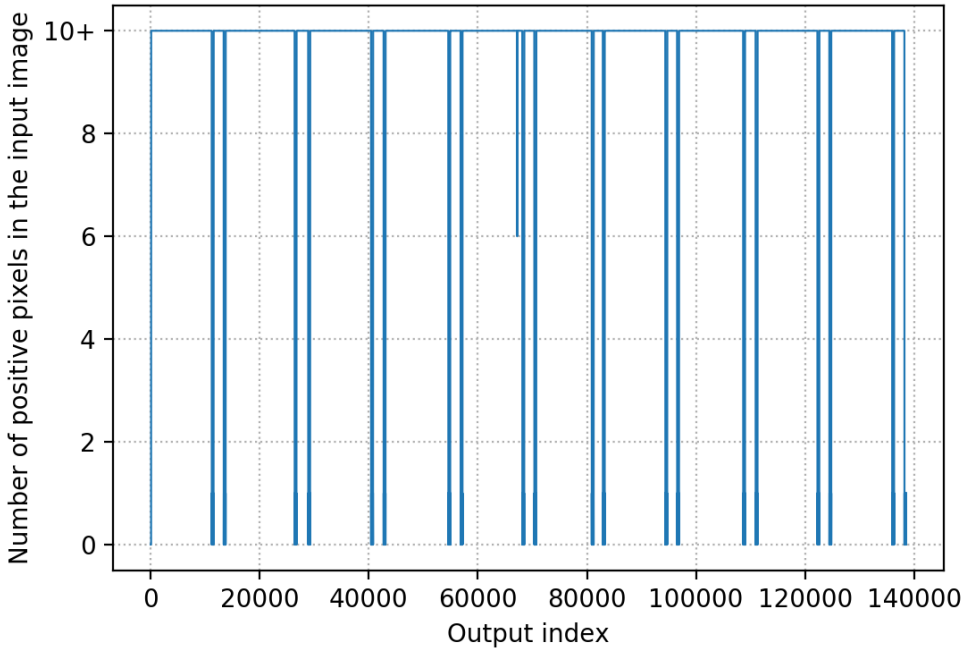


Figure 4.8: Number of positive input pixels for each saved output, during noisy data acquisition. A number inferior to 3 means the camera did not capture an input digit, but an empty screen instead. The alternating pattern shows 20 clearly defined periods, each of them corresponding to a different class and subset: the first one consists in training '0's for instance, etc.

The display and acquisition process are loosely synchronised to each other, with the SCAMP5 capturing 12 frames per second, and the display program showing 6 images per second. This relatively low speed ensures a consistent illumination and a well controlled exposure time for the camera. The whole acquisition can be done in under 4 hours, to get data ready to use for the fully connected layers training.

This method can of course easily be generalised to any other visual supervised learning task, and is not restricted to the MNIST dataset.

4.8 Fully connected layers

The fully connected layers of our CNNs are computed on the digital micro-controller, once the binary feature locations have been collected on the focal plane, and binned into flat a vector. SCAMP5's micro-controller offers precise computations that are not subject to noise - just as any digital micro-controller, but as opposed to the analog vision chip - yet has some speed and capabilities limitations that we will discuss in this section.

In this work, only one fixed architecture of the fully connected layers is used, com-

posed of two layers, with 50 hidden units and 10 output units, using a ReLU activation function. The following remarks however remain valid with other architectures.

4.8.1 Rounding weights

SCAMP5's digital micro-controller offers no floating point computations' capabilities, whereas any machine learning framework's weights for a fully connected layer are stored in single-precision floating-point format with 32 or 16 bits. To avoid any difficulty with a custom training that would yield integer weights, we instead decide to scale and round them before using them on the micro-controller.

Suppose the first layer has a weight matrix K_1 and a bias vector B_1, and the second layer a weight matrix K_2 and a bias vector B_2. Then, for an input vector x, the output of the fully connected layer y is:

$$y = K_2 \cdot ReLU[K_1 \cdot x + B_1] + B_2$$

If all weights and biases are scaled by a factor s, except for B_2 being scaled by s^2, then

$$\begin{aligned} s^2 \cdot y &= s^2 \cdot (K_2 \cdot ReLU[K_1 \cdot x + B_1] + B_2) \\ &= s \cdot K_2 \cdot ReLU[s \cdot K_1 \cdot x + s \cdot B_1] + s^2 \cdot B_2) \end{aligned}$$

Using the integer part of the up-scaled weights and biases allows for an approximation of the up-scaled output:

$$s^2 \cdot y \simeq \lfloor s \cdot K_2 \rfloor \cdot ReLU\left[\lfloor s \cdot K_1 \rfloor \cdot x + \lfloor s \cdot B_1 \rfloor\right] + \left\lfloor s^2 \cdot B_2 \right\rfloor$$

This shows that the fully connected layers' computation can be approximated by integer computations (x corresponds to counts in bins, and is already composed of integers). This allows for the fully connected layers to be implemented on the micro-controller. To this end, it suffices to up-scale K_1, B_1 and K_2 by s, and B_2 by s^2, before rounding them. In this approximation, the error is a function of the input values x, but also of s: the larger s is, the smaller the error.
The loss of precision is controlled by s, and can be acceptable for large enough values. As an example, with an original bias vector

$$(K_2)^T = (-1.4000176, 4.7173615, ..., -2.3324227, -2.3793328)$$

and a scale factor $s = 10$, we get

$$(\lfloor s^2 \cdot K_2 \rfloor)^T = (-141, 471, ..., -234, -238)$$

With $s = 100$, the same bias vector gets approximated by

$$(\lfloor s^2 \cdot K_2 \rfloor)^T = (-14001, 47173, ..., -23325, -23794)$$

and the error is reduced.

Getting an up-scaled version of the output y is not an issue here. During inference time, what indeed matters is the index of the maximum in the output vector, which is the predicted label.

4.8.2 Profiling

Table 4.3 reports the breakdown of the elapsed time during one forward pass of AnalogNet on the SCAMP5 device, after an image is captured.

Step	Time taken (micro-seconds)	Total elapsed time (micro-seconds)
Input binarisation	13	13
3 Convolutions and output binarisation	68	81
Events reading and binning	273	354
Fully connected (2 layers)	389	743

Table 4.3: Profiling AnalogNet.

As we can see, a considerable amount of time is spent gathering events (i.e. transferring data from the vision chip to the micro-controller), and computing the fully connected layers result. The former cannot really be improved, as we use the already optimised official SCAMP5 library to interface with the analog vision chip.
However, there is some room for improvement on the latter point. The legacy AnalogNet implements very standard double loops for matrix-vector multiplication. The current and standard implementation is in the form of Listing 11.

```
int weights[IN_SIZE][OUT_SIZE] = {{...},{...},...};
int biases[OUT_SIZE] = {...};


...
  for (int j=0; j<OUT_SIZE; j++){
    partialSum = 0;
    for (int i=0; i<IN_SIZE; i++){
      partialSum += in[i]*weights[i][j];
    }
    out[j] = partialSum + biases[j];
  }
 ...
```

Listing 11: Standard matrix-vector multiplication, as implemented in the legacy AnalogNet.

We scripted the generation of C++ code for defining int arrays, based on Python objects. This script takes a Python 2 dimensional array as input that corresponds to the weights matrix of one of the fully connected layers, and simply re-parses it into a valid C++ array. It can then be directly copied and pasted into our SCAMP5 source code.

4.8.3 Improving speed

In our case, we can take advantage of two things:

- The fact we know in advance (at compilation time), the number of iterations needed in each loop. This is because our vectors are all of fixed size.

- The fact that we do not use all of the available 512KB flash memory for storing the program and constant data.

Based on these two points, we propose to unroll the loop before compilation, and to hard-code the weights in the unrolled loop. Loop unrolling eliminates delay from controlling the loop, and is a long known idea of program optimisation [2]. Moreover, instead of declaring a global constant array for the weights and accessing them, we replace them by their actual value. This removes the overhead of accessing memory, and even frees some RAM. The resulting implementation of matrix-vector multiplication is in the form of Listing 12.

```
int input[IN_SIZE], output[OUT_SIZE];
out[0] =
  in[0]*(322) +
  ... +
  in[IN_SIZE-1]*(77) +
  (-47);
...
out[OUT_SIZE-1] =
  in[0]*(18) +
  ... +
  in[IN_SIZE-1]*(27) +
  (20);
```

Listing 12: Unrolled loop for matrix-vector multiplication, with hardcoded weights.

Using a Python script, we generate C++ code that corresponds to the linear program equivalent to the loop of code in Listing 11. The length of our program is considerably increased, but still largely within the limit of 512 KB. The resulting speed-up is substantial, as presented in Table 4.4 and Table 4.5. Loop unrolling in itself does not bring a significant improvement, but is a requirement for using hardcoded weights.

Method	Time for fully-connected computation (micro-seconds)	% of baseline
Standard loop (baseline)	389	100
Unrolled loop	374	96.1
Unrolled loop + hardcoded weights	109	28.0

Table 4.4: Comparison of the time taken for the computation of a 2 layers fully connected network (27 input units, 50 hidden, 10 output), with three different methods used for matrix-vector multiplication.

Method	Time for AnalogNet forward pass (micro-seconds)	% of baseline
Standard loop (baseline)	743	100
Unrolled loop	728	98.0
Unrolled loop + hardcoded weights	463	62.3

Table 4.5: Comparison of the time taken for a forward pass on AnalogNet on the SCAMP5 vision system, with three different methods used for the matrix-vector multiplication involved in the fully connected layers.

This technique is equivalent to putting some manual optimisation upstream of the compiler. It yields longer programs, but exploits the freely available ROM space to drastically decrease the inference time on the SCAMP5 device. The frame rate is increased by 60% for AnalogNet. Further optimisation would maybe require parallel computations, which are not easily possible here on the M0 core.
We designed Python scripts to generate unrolled loops from Python arrays storing the fully connected layers weights and biases. The output can directly be copied and pasted into M0 C++ source code. The optimisation we presented here is very generic and applicable for other architectures of fully connected networks than the one we used.

This chapter provided useful and optimised implementations of CNN building blocks. Prototyping CNNs is now easier thanks to the work in Section 4.1 (Streamlining AUKE's output) and Section 4.7 (Noisy data acquisition). Digital simulations are more precise thanks to what is presented in Section 4.6 (Output thresholding). Moreover, we are now equipped with new building blocks to include in our CNNs' architectures, presented in Section 4.2 (ReLU and leaky ReLU), Section 4.3 (Pooling), Section 4.4 (Convolutions on pooled data) and Section 4.5 (Quantisation). Finally, Section 4.8 (Fully connected layers) provides a major improvement in inference speed on the SCAMP5 device. In the next chapter, we use this to substantially improve the baseline of CNNs on the SCAMP5 device, and further explore the architecture space for 2 convolutional layer networks.

Chapter 5

MNIST experiments

Equipped with refined and new CNN primitives for the SCAMP5 device, we test these on the classical MNIST dataset. The goal we pursue is double: increasing raw performance (accuracy, inference time) and demonstrating the feasibility of multi-layer CNNs.

5.1 AnalogNet2

Taking a close look at AnalogNet's implementation and architecture as we did in Section 3.2 let us see some room for improvement. Before experimenting with radically new CNN architectures that have 2 layers in further chapters, we here review some refinements we brought to AnalogNet. We created a very similar single layer CNN for the SCAMP5 device that we call AnalogNet2.

5.1.1 Output event binning process

As presented in Figure 3.5, the legacy AnalogNet implementation bins events into 9 different squares. With this binning procedure, features located in the central area all fall within the same bin, and cannot be differentiated by the fully connected layers. We assess this as detrimental for the accuracy of the network, and propose a new binning procedure. Using the prior knowledge that MNIST digits are located in the central area of the 28*28 square, we suggest to discard events located in the four corners, and to use different bins for the central events - to preserve more information about their location.
As seen on Figure 5.1, the new binning procedure uses 12 overlapping bins instead of 9 adjacent ones. Each convolutional feature map thus yields one third more values for the fully connected layers, and first of the two fully connected layers takes 33% more time to compute. However, the data rate from the analog vision system to the micro-controller still remains exactly the same - up to on hundred 2D coordinates of events per output feature map. As shown on Table 5.1, having more input values (36 isntead of 27) indeed increases the time required to compute the fully connected layers' output. It is however still faster than the legacy method by a large margin,

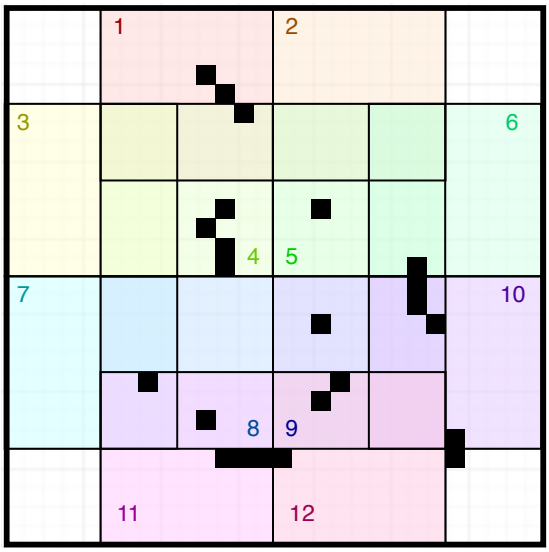


Figure 5.1: New binning process. Events fall into twelve 9*9 overlapping bins. Black pixels correspond to the same possible events configuration as in Figure 3.5: in this case the output vector is $(3, 0, 0, 5, 2, 1, 1, 2, 6, 4, 5, 3)$.

and the increase in time is negligible compared to what loop unrolling brings (see Subsection 4.8.3).

Method	# of input values	Time for fully-connected computation (micro-seconds)
Standard loop	27	389
Unrolled loop + hardcoded weights	27	109
Unrolled loop + hardcoded weights	36	136

Table 5.1: Comparison of the time taken by the fully connected layers' computations on the SCAMP5's micro-controller, for different matrix-vector multiplication methods and number of input values.

Replacing the legacy binning process with this one brings a very substantial improvement of more than 3% in testing accuracy, once implemented on the SCAMP5 device. As explained in the next sections, we were indeed able to train networks that reach between 96% and 97% testing accuracy, compared to 92%-93% for the simpler pooling operation (as reported by [49]).
Our Tensorflow fully digital simulation of a single layer network with this pooling technique reaches 97.3% testing accuracy, which is significantly superior to the 91.1% we reported in Subsection 4.6.1, with the standard pooling operation.

This new output pooling is specifically designed for the MNIST digit classification task, in which the corners of the image are far less relevant than the center. The general idea of trying specically designed binning layouts is however valuable for

other use cases.

5.1.2 Analog register management

In AnalogNet's legacy implementation, the input image is first transferred to AREGs A, B and C, where the three convolution kernels are independently applied using AREGs D, E and F for computations. After this computation, there is not enough remaining AREGs to carry on computations on the focal plane, and feature maps are therefore thresholded and sent to the micro-controller. This is shown on Figure 3.3 in Subsection 3.2.1.

We propose a new AREG management, to reduce congestion on the focal plane. The idea is to sequentially copy the input image to AREG A, then run the computations for a convolution kernel using AREGs B to F, before outputting the thresholded feature map to the micro-controller. This sequence of operation, shown on Figure 5.2, can be run as many times as needed, for as many kernels as we want.

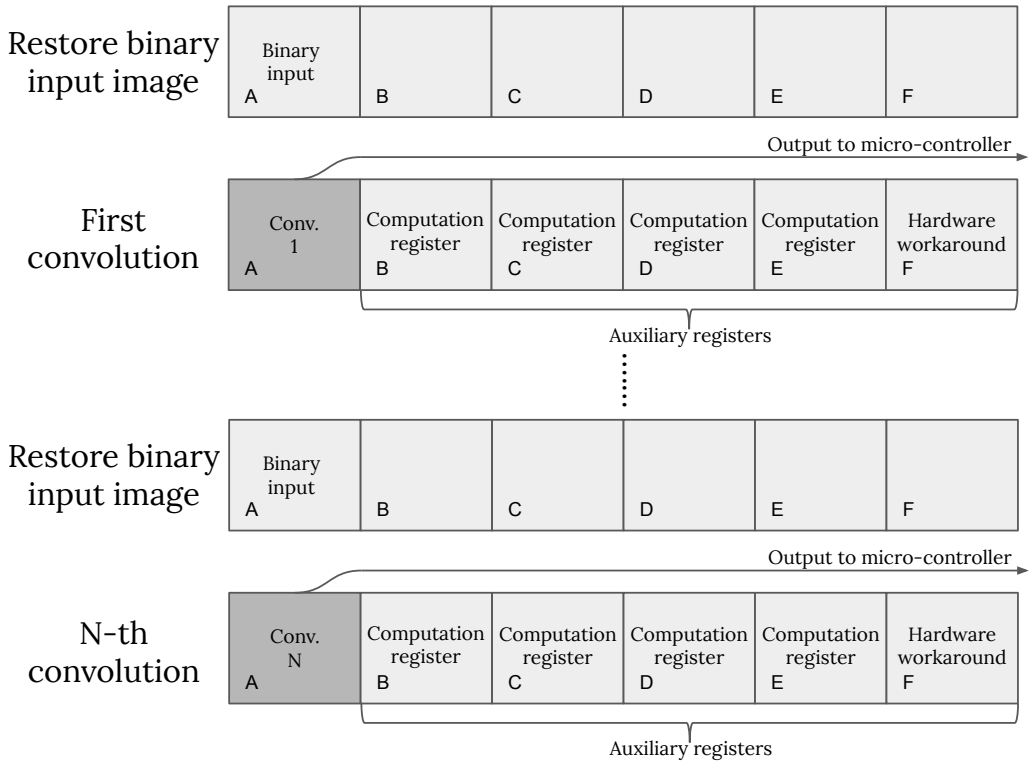


Figure 5.2: New AREG management: each convolution is computed using AREGs. The input image is each time loaded from a persistent DREG.

The first consequence of this new organisation is that it increases the number of AREGs that are available as auxiliary registers for the convolutions' computations. This should enable AUKE to possibly find shorter programs, and hence enhance signal quality. This however does not bring such an improvement in reality. We were

indeed not able to witness any real decrease in instruction count in codes produced by AUKE, and hence no noise reduction.
Besides, the main positive consequence is that the number of convolutions in the first and only convolutional layer is no longer limited to 3. We can indeed iterate the procedure as many times as required, once for each convolution. The only cost incurred is the computation time of the convolutional part, which scales linearly with the number of feature maps it has.

5.1.3 Increasing the number of kernels

Freed from the constraint of having at most 3 convolutions in the first layer thanks to the new AREG management, we here explore the architecture space of 1-layer CNNs. We study the accuracy of single layer CNNs, using increasingly many convolutions. All other parameters - such as the size of the fully connected layers - are fixed, and similar to the ones used by the legacy AnalogNet. To sum it up, the differences between AnalogNet and these networks are the use of:

- a varying number of kernels in the convolutional layer (between 1 and 7), allowed by a new AREG management (see Subsection 5.1.2).
- a new output binning layout, with 12 bins instead of 9 (see Subsection 5.1.1).
- unrolled loops for the fully connected layers, for faster execution (see Section 4.8).

Figure 5.3 reports the resulting testing accuracy, both for a digital implementation in Tensorflow, and for a final implementation on the SCAMP5 device. For each kernel number between 1 and 7, the full training process is run - including custom regularisation, output binarisation and fully connected layers retraining with noisy data. As one could expect, a network with more kernels reaches a higher accuracy, but the returns of adding new kernels is diminishing.

We notice that such high testing accuracy seen on Figure 5.3 were previously unreported. Compared to the 92.65% testing accuracy of AnalogNet reported in [49], even a CNN using 2 kernels only achieves higher performance. With 3 or more kernels, the reached accuracy is above 96%, which is an unprecedented level of precision. A 7 kernels CNN even beats the 98% mark. These very substantial improvements are the result of both the new pooling layout (as confirmed by the testing accuracy of the 3 kernels CNN) and the feasible increase in kernels number.

The drawbacks of using more kernels are an increased latency and power consumption. Using Tables 4.3 and Table 4.4, that report the time spent at each stage of a standard AnalogNet, we devise a very crude approximation of the inference time for these new networks using a variable number of kernels k.
The time spent in binarising the input is constant in k, the total duration of the convolutions scales linearly in k (including kernel execution, output thresholding, events reading and binning), and in varies in an linear manner with k for the fully

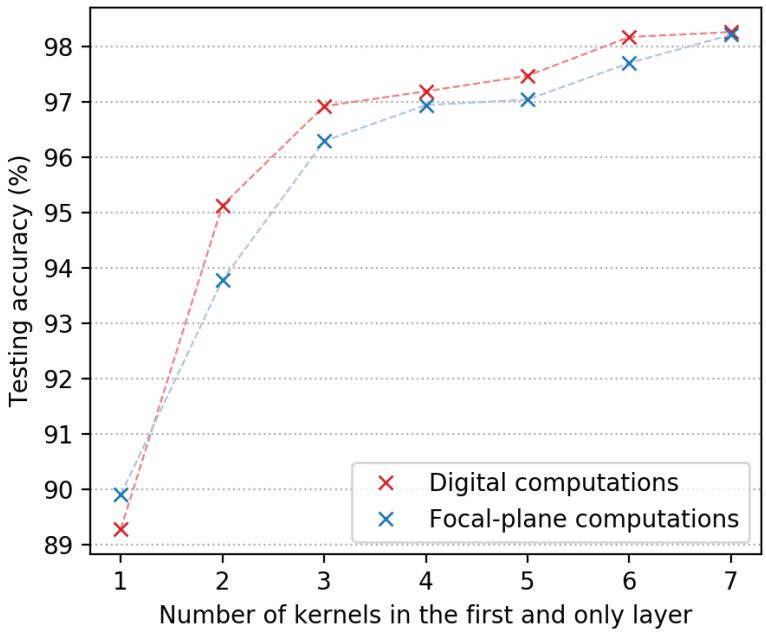


Figure 5.3: Testing accuracy, with increasingly many kernels in a single layer CNN

connected layers (only the first matrix multiplication is impacted, the second layer remains constant in time). We can therefore estimate the required time for one forward pass of the network on the SCAMP5 device as a function of k, in microseconds:

$$time(k) = 13 + k \cdot \frac{68}{3} + k \cdot \frac{273}{3} + 109 \cdot \frac{12 \cdot k \cdot 50 + 50 \cdot 10}{27 \cdot 50 + 50 \cdot 10} \tag{5.1}$$

Figure 5.4 presents estimated inference duration and frames per second derived from Equation 5.1. The trade-off between inference speed and accuracy is presented on the Pareto diagram in Figure 5.5.

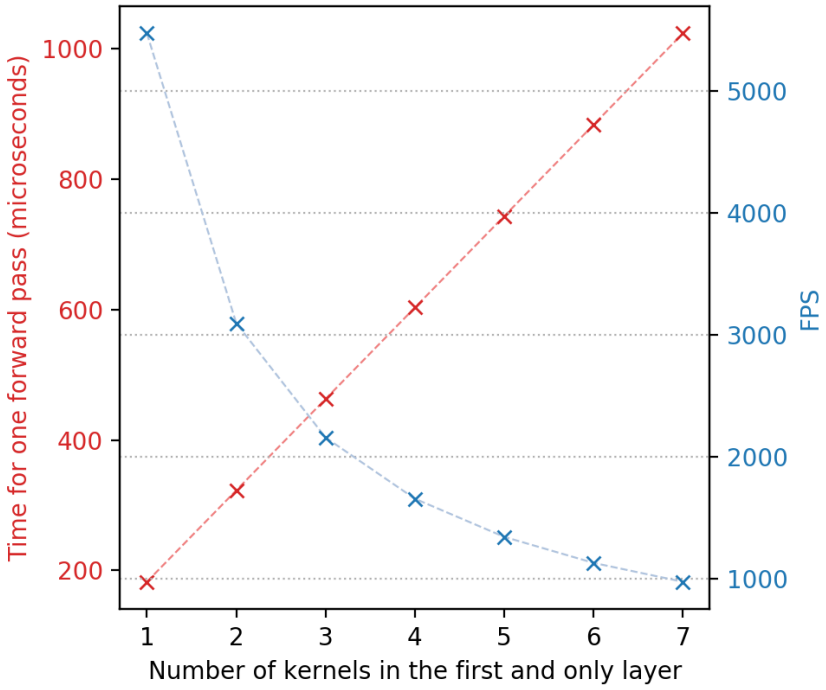


Figure 5.4: Estimation of the time required for one forward pass of a single layer network with increasingly many kernels, and the resulting FPS

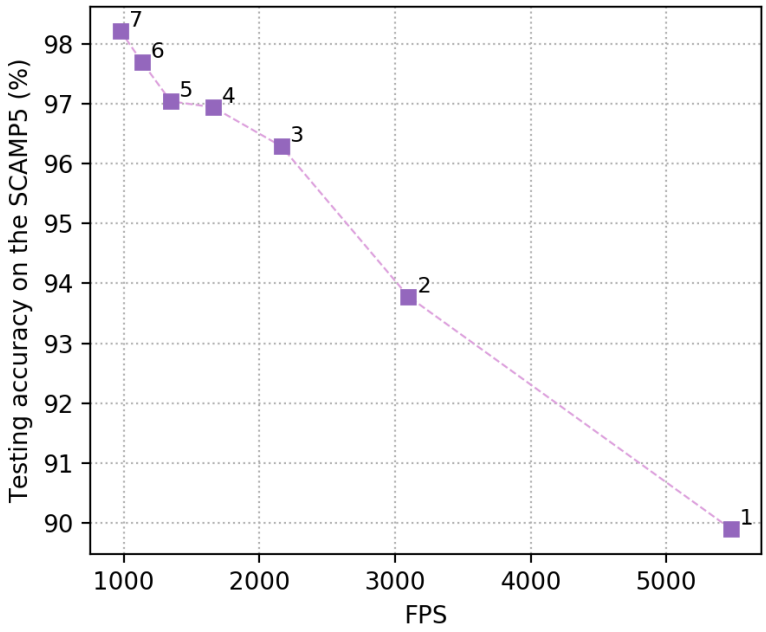


Figure 5.5: Visualisation of the trade off at stake: an increase in accuracy yields a decrease in FPS. A higher accuracy is better (vertical axis), a higher FPS is better (horizontal axis).

This systematic search of the architecture space of single layer CNNs clearly exhibits the trade-off at stake. We consider the 3 kernels, single layer CNN to be at the sweet spot of this design space. With more kernels, the networks fall below the landmark of 2000 FPS. Being at the inflexion point on Figure 5.3, adding more than 3 kernels gives diminishing returns in terms of testing accuracy.
Moreover, there is still some room for improvement in the intrinsic learning process variability. The figures reported above are indeed the result of one training process per network. Launching several training processes for networks with 3 kernels, and keeping only the best of them might lead to a testing accuracy slightly above the 96.3% reported on Figure 5.3.

5.1.4 Final result and architecture

Based on the above findings, we propose an updated version of AnalogNet, that we call AnalogNet2. Using a single convolutional layer with 3 kernels, an updated binning layout and optimised fully connected layers computations, it reaches **96.9% accuracy** on the MNIST test set. With a measured inference time of 442 micro-seconds, it can run at **up to 2260 FPS** on the SCAMP5 device, in a **power budget of 0.7 mJ per frame**.
AnalogNet2 uses the same number of convolutions as AnalogNet (3 kernels). In the case of AnalogNet, this figure was a consequence of hardware limitations - namely a limited number of AREGs. This hardware limitation now being overcome, having 3 convolutions in AnalogNet2 is a choice of design. Depending on use cases, it can easily be adjusted upwards for an increased accuracy, or downwards for an increased throughput.

5.2 Implementing two layer CNNs

AnalogNet2 achieves unprecedented accuracy and efficiency on the MNIST test bed, with a single convolutional layer. This architecture is extremely simple compared to modern CNNs, which include tens of convolutional layers, each with tens of feature maps. Having this many convolutions helps in extracting complex hierarchical features, and is required to perform high level tasks such as object detection [44] or pose estimation [48].
For this reason, we here explore the feasibility of implementing a two layer CNN on the SCAMP5 device. It is indeed the first step towards implementing more than one convolutional layer. For practical reasons and for a fair comparison with AnalogNet2, we decide to also use the MNIST dataset to assess the performance of our models.

The main design issue that arises is caused by limited register availability. Each feature map in the second convolutional layer is indeed the sum of the results of convolutions applied to each feature map in the first layer, as presented in Subsection 2.2.2. This creates the need to store many partial results, since all feature maps of the first layer are required until the very last feature map of the second layer is

computed. We found two possible ways of overcoming this issue, and hence propose two two-layer CNN architectures for the SCAMP5 device in this section.

5.2.1 Two layers with quantisation

Our first attempt at making a two layer CNN consists in duplicating the convolutional part of AnalogNet2. This results in the very basic architecture of a two layer network, both layers having three feature maps. In the second layer, each feature map is the sum of all first layer feature maps, each one convoluted with a different kernel. Figure 5.6 shows an example of such computations.

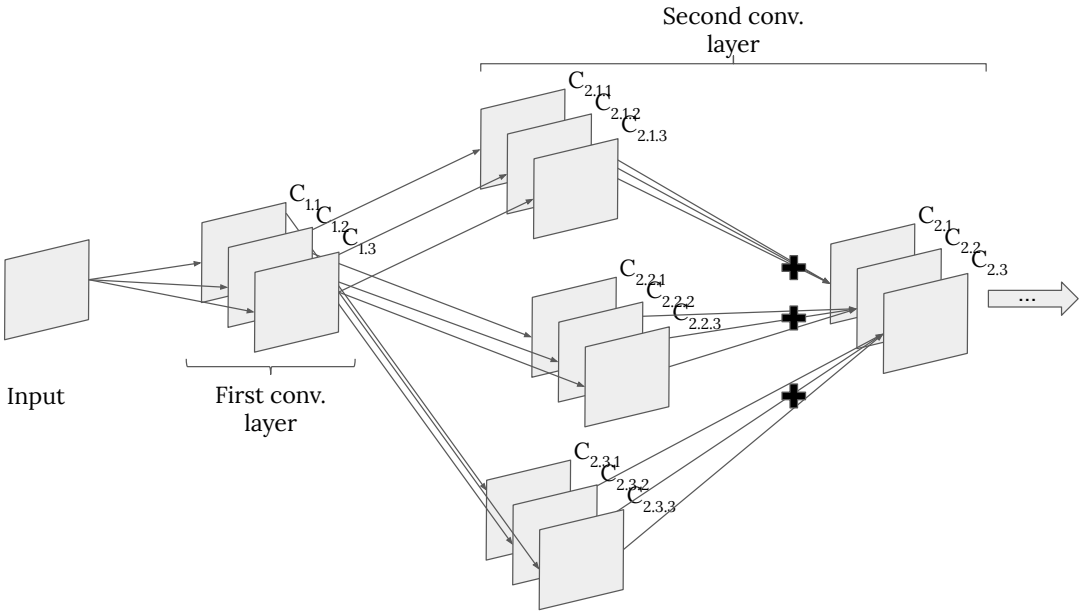


Figure 5.6: Computations involved in the convolutional part of a two layer CNN, each layer having three feature maps. It requires the application of 3+3*3=12 convolution kernels. The fully connected layers are omitted here.

To illustrate the challenge of implementing even this very basic architecture, suppose that convolutions are computed in lexicographical order on their numbering in Figure 5.6. Assume $C_{1.1}$, $C_{1.2}$, $C_{1.3}$ and $C_{2.1.1}$ have already been computed on the focal plane, and $C_{2.1.2}$ should now be calculated. On SCAMP5's focal plane, one AREG is reserved for the hardware workaround, and one AREG must be reserved for storing each of the already computed convolutions - since all of these results will be required at a later point. Out of 6 AREGs, 5 are occupied, and only one remains available for computing $C_{2.1.2}$, which restricts it to be an extremely simple kernel.
To answer our newly created need of storing many partial result, we propose to store some AREG's content in DREGs for later use, when not presently needed for any computation. Using the quantisation procedure presented in Section 4.5, one AREG's content can easily be digitised and moved to multiple DREGs. The AREG can then be freed and used for current computations, while its content is saved and

available for later use. The precision level of the quantisation procedure can easily be adjusted, to choose between very frugal DREG use and high precision quantisation.

Architecture

We use the quantisation procedure to binarise the first layer's feature maps when required. Since these feature maps are applied a ReLU activation function, they only contain strictly positive values, and hence unsigned quantisation can be used. This decreases the count of DREGs needed to store one AREG content by one (the sign is no longer required).
Quantisation is an idea that has already been explored to reduce the memory footprint of CNNs, as presented in Subsection 2.3.1. Since then, Tensorflow already includes quantisation instructions, and these exotic CNNs can be trained without much hassle. This helped us simulating them, and determining the best amount of DREGs (or bits) needed to store one AREG. Given the limited number of DREGs, we can use up to 4 DREGs per feature map. Table 5.2 sums up our experiments for digital networks, in Tensorflow. The best trade-off is reached with using convolution kernels trained with the 4 bits quantisation architecture on the 3 bits quantisation one, and retraining only the fully connected layers. It provides a network that can be used with 3 DREGs per feature map at inference time, with only a minimal decrease in theoretical (i.e. digital) testing accuracy - 0.1%. We thus decide to use 3 DREGs per feature map, since having one more DREG does not seem to bring any improvement.

CNN implementation	MNIST testing accuracy
unquantised	97.6%
quantised on 4 bits	97.0%
quantised on 3 bits, using convolution kernels of the above 4 bits network	96.9%

Table 5.2: Testing accuracy comparison of a 2 layer CNN, in Tensorflow. Each layer has 3 feature maps. The reported figures include all side-effects of the SCAMP5 device (input/output binarisation, limited convolutions, ...), except noise. The last line corresponds to re-using the weights obtained with the 4 bits architecture with 3 bits quantisation, and retraining only the fully connected layers.

The program flow of the convolutional part of the network is presented in Chapter A of the appendix. It is composed of two convolutional layers, each one having 3 feature maps. The convolution kernels used are of size 3*3, and the fully connected layers' structure remain unchanged compared to AnalogNet and AnalogNet2 (2 layers, 50 hidden units and 10 output units).
One of the first layer's feature does not need to be quantised, and its original version can be used in full precision analog form. Besides, when an output feature map has been computed, it is immediately binarised and sent to the micro-controller. This

principle is very similar to what was presented in Subsection 5.1.2, and frees AREGs for remaining computations.

Results

The twelve convolution kernels are loaded onto the SCAMP5 device, and a full acquisition of the MNIST data set is run. As explained in Section 4.7, this is done to retrain the fully connected layers.
When implementing the two layers, a bias should normally be applied to each feature map between the first and the second layer. However, bias values resulting from the training process in Tensorflow are small (between -5 and +5) compared to the inaccuracy and noise of the device. For this reason, bias addition is omitted.

The final implementation on the SCAMP5 device reaches **95.8% accuracy** on the MNIST test set. With an inference time of 803 micro-seconds, it can run at **up to 1245 FPS** on the SCAMP5 device, in a **power budget of 1.5 mJ per frame**.
The 80% increase in inference time compared to AnalogNet2 is mainly due to a change in collected events per feature map: from 100 to 150. The output feature maps of this two-layer CNN are indeed denser, and more events are required to capture them.
In-situ results on the SCAMP5 device are deceivingly lower than with the fully digital implementation (TODO% decrease), and than AnalogNet2 (1.1% decrease). Our intuitive explanation is that it is caused by the accumulation of noise on the focal plane, due to the increased depth of our computational graph.

5.2.2 Two layers with pooling

As explained in Section 4.3 and Section 4.4, we are now enabled with an efficient implementation of feature map pooling - from the pooling itself, the interleaving of four pooled feature maps on a single AREG, and the execution of convolutions on interleaved feature maps.
In this section we propose a two-layer CNN that makes use of this, to reduce the dimensionality of feature maps and be able to fit on our FPSP, the SCAMP5 vision system. Storing multiple feature maps on a single AREG makes the implementation of a two layer CNN possible on our heavily memory-constrained device, without requiring to quantise any intermediate result. This is a different approach as the one explored in Subsection 5.2.1, aimed at the same objective. It presents the advantage of using a much more common ingredient than quantisation - pooling, which one can easily argue to play a useful role in selecting the most important features of a feature space.

Architecture

We choose to implement a 2 layer CNN composed of 4 convolutions in the first layer, followed by a 2*2 pooling operation, and 8 convolutions in the second layer. The

convolutions are all composed of 3*3 kernels, and Section 4.4 explains how convolutions are run for the second layer.

The most usual pooling operator used in CNNs is average pooling. However, our simulations in Tensorflow yield better results with maximum pooling, which we therefore use in our proposed architecture. One possible explanation for this resides in the fact that our CNN uses a limited set of convolution kernels' weights, for which it might be easier to deal with a restricted set of input values. Taking the maximum value in each 2*2 activation cluster from the first layer indeed creates fewer distinct values than combining them in an average.
The pooled feature space of the first layer can fit on a single AREG, with interlaced feature maps. The 8 feature maps of the second layer must be separated in two groups of 4, each one fitting on an AREG. Half of the output feature space can thus be transferred to the digital micro-controller with a single event read. Based on the parity of its coordinates, an event can easily be attributed to its feature map. Since the output feature maps now have size 14*14, the binning layout must be changed accordingly. Guided by our observation in Subsection 5.1.1 that differentiating central pixels is helpful for the subsequent fully connected layers, we divide a feature map in four quarters, one in each corner. The corresponding layout is shown in Figure 5.7. Each of the 8 feature map is hence transformed in a vector of size 4, with the input to the fully connected layers hence being of size 32.

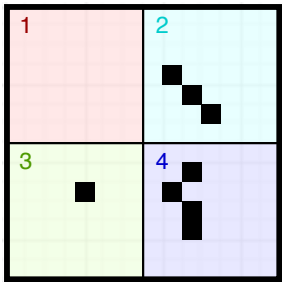


Figure 5.7: Binning process. Events fall into four 7*7 disjoint bins. Black pixels correspond to a possible events configuration: in this case the output vector is $(0, 3, 1, 4)$.

SCAMP5 excessive program length

Implementing the above architecture on the SCAMP5 device, we faced an unexpected issue: we hit the limit of vision chip instructions count. As explained in Subsection 2.1.1, the M0 core is responsible for controlling the analog vision chip. Corresponding instructions - or *kernels* - are written in special sections among the M0 source code, and compiled into SCAMP5 machine code on first encounter.
A SCAMP5 kernel is defined using either a lambda function of type `scamp5_kernel`, or `scamp5_kernel_begin()` / `scamp5_kernel_end()` delimiters. Upon the first execution of a `scamp5_kernel`, it is first compiled, and then sent to the vision chip. The compiled version is stored, so that further executions are faster: the code can directly be sent to the vision chip. The compiled SCAMP5 Kernels are stored in a

specific part of the memory, that can only store 1024 instruction words. Once that limit is reached, the whole program stops functioning. For relatively long programs such as ours, this limit is quickly reached. We indeed have to use 4 (first layer) + 4*8 (second layer) = 36 convolutions, whereas our previous models only had 3 or 12.

The official SCAMP5 library [10] describes one possible solution to this issue. It consists in using the `move_to_heap` keyword when creating a SCAMP5 kernel, which allows kernels to be stored in a new memory location that is larger than the default one. It however still is too small for our needs.
The new idea we developed to circumvent that issue is to dynamically allocate SCAMP5 Kernels. It offers a virtually unlimited amount of instructions to be executed on the vision chip, with a technique known as *overlay* [30], allowing to execute programs that do not fit in the live memory. The general idea is to cut a long program in blocks, and sequentially transfer them to a computer's memory for execution. In our case, a `scamp5_kernel` is in turn:

1. Created, in the heap (using the C++ `new()` instruction);
2. Compiled;
3. Sent to the vision chip for execution (as many times as required);
4. Deleted.

This frees us from the above limitation: one can cut his program in sub-parts, virtually as many as one wants or needs. The new limitation is the 512 KB flash memory. The only drawback of that method is speed. Indeed, vision algorithm are often infinite loops, in which the same kernels are run at each iteration. With our solution, the kernels are recompiled each time, which slows down the main loop of the program. This however yields frame rates that are still in hundreds of frames per seconds, and has the great advantage of allowing the creation of prototypes and proof of concept programs.
Note: when using such dynamically allocated SCAMP5 Kernels, one should not use 'traditionally' defined Kernels. Mixing the two leads to highly erratic behaviours.

Results

The accuracy on the MNIST test-set of this architecture is of 96.4% for the digital implementation in Tensorflow. Once implemented on the SCAMP5 device, and after retraining the fully connected layers with noisy data, the final **in-situ testing accuracy is of 92.9%**. Because of the slow overlay technique, the inference time of 2762 microseconds only allows for **360 FPS**. The power budget also suffers from cutting the program in blocks at **6.6 mJ per frame**.

Even if the digital implementation of the network does not perform better than AnalogNet2 in-situ (96.9% accuracy on the SCAMP5 device), we considered it valuable to test it on the SCAMP5 device. AnalogNet2 indeed uses a specific binning

trick designed for MNIST, whereas this CNN is generic and does not use this prior information. In some cases, devising an optimised handcrafted pooling operation might be difficult, and for this reason, generic architectures also must be explored.

Once again, we explain the important accuracy degradation between the digital implementation and its in-situ counterpart with noise. To the best of our knowledge, this is indeed the only missing part of our digital simulations. The difference between the simulation and the actual implementation would surely be even worse were we shifting AREGs' content instead of interlacing them, since it requires more instructions and longer computation circuits.
The 3 CNN implementation we have on the SCAMP5 device (AnalogNet2, 2 layers using quantisation, 2 layer using pooling) tend to exhibit a more important degradation when the instruction chain between the input and the output on the focal plane is longer.

The results of our experiments on multi-layer CNNs call for a more thorough analysis of focal plane's noise, which is presented in the next chapter.

Chapter 6

Noise analysis

The two attempts to introduce a second convolutional layer (using quantisation or pooling) yield deceptive and counter-intuitive results on the SCAMP5 vision system. Despite being implemented using two very different approaches, both in situ testing accuracy figures suffer a significant decrease compared to digital computations, on a standard computer. However, as described above, our fully digital computations account for all specific aspects of our FPSP except noise. We hence conclude that both experiments call for the same diagnostic: analog noise hinders computations in such proportions that prevent the reliable implementation of traditional two layer CNNs.

In this chapter, we explore further the impact of noise on analog computations happening in the focal plane, and discuss how it could possibly be restrained.

6.1 Noise accumulation

As long as we restrict the architecture space of CNNs to single layer network such as AnalogNet, noise remains within acceptable amounts. Retraining the fully connected layers with real data produced by the convolutional layers implemented on the focal plane was enough to tackle this issue. In this case, the in situ testing accuracy is extremely close to fully digital CNNs. Once a second layer is added, this retraining process is no longer enough to maintain an acceptable accuracy. This suggests that noise has accumulated in such a way that it can no longer be accounted for by the fully connected layers only. The second layer acts on already noisy data, and noise is added on top of noise. The computation chain between the input image and the output feature maps is longer, and the information is supposed to be refined, but is in reality diluted in increasing amounts of noise. What really is detrimental here is the *accumulation* of noise.

In this section, we gather experimental results and provide quantitative indicators to support the claim that our two layer CNNs fail to beat or even compete with AnalogNet because of noise accumulation.

6.1.1 Depth separable layers

One can argue that our implementations of 2 layer CNNs both rely on newly introduced techniques (quantisation and feature map interlacing) which might cause the large decrease in testing accuracy between digital version of the networks and real implementation on the SCAMP5 device. To fully isolate their potential side effects, we here assess the performance of multi-layer CNNs that are not relying on such primitives.
Since it is not possible to execute multiple standard convolutional layers on the SCAMP5 device because of hardware limitations, we here use *depth separable convolutions*. With such layers, feature maps are not combined with summation as with traditional convolutional layers (shown on Figure 5.6). Instead, series of convolutions and non-linearities are applied to the input images, each one forming an independent thread in the computational graph. This greatly decreases the requirement to store partial results on the focal plane.
Since feature maps are not combined, feature extraction is not as powerful and efficient as with standard convolutional layers. However, the accumulation of non-linearities and convolutions still helps in improving testing accuracy.

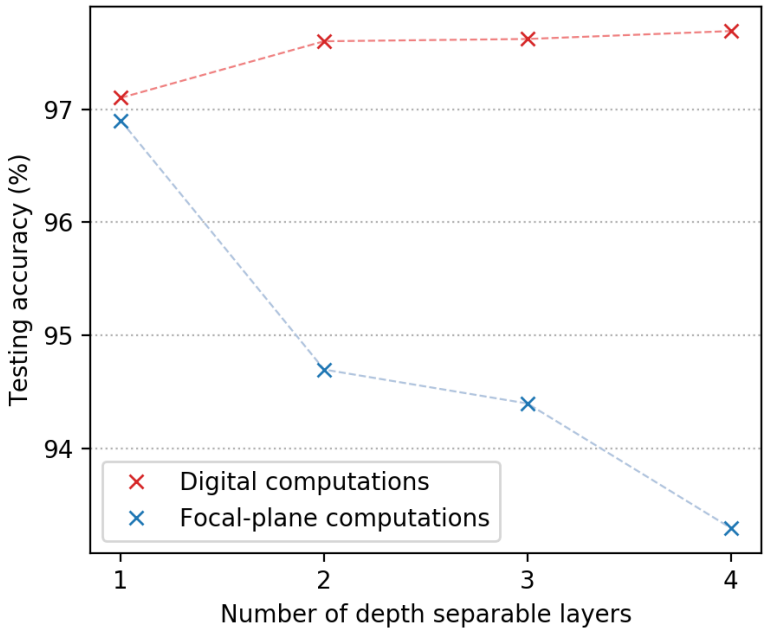


Figure 6.1: MNIST testing accuracy of CNNs using increasingly many layers of depth separable convolutions. Each layer is made of 3 3*3 convolutions, and a ReLU activation function. The case with 1 layer corresponds to AnalogNet 2.

Figure 6.1 shows the MNIST testing accuracy of CNNs using increasingly many layers of depth separable convolutions, both in our noiseless digital simulations and on the SCAMP5 device. In simulations, having more depth separable layers yields

better results, with diminishing returns. On the SCAMP5 device, the exact inverse phenomenon happens, with results getting worse each time a new layer is added.

This results confirm our intuition that despite being more powerful in theory - i.e. with noiseless and precise digital implementations - networks involving longer chains of computation perform poorly on the SCAMP5 device.

6.1.2 Composing noise-inducing operations

The accumulation of noise on the focal plane is not specific to the computation of convolutional feature maps. Any analog computation is subject to noise. In this experiment, we repeatedly run a very simple operation on an uniform AREG content, and witness the results deterioration. The results we present here are very generic, and remain valid in broader contexts not related to CNN implementations, nor specific to the SCAMP5 device.

Data collection

We define the kernel presented in Listing 13, that should in theory be equivalent to the identity operation.

```
scamp5_kernel id_kernel([]{
  using namespace scamp5_kernel_api;
  diva(A,E,F);
  mov(F,A);
  add(A,A,F);
});
```

Listing 13: Supposedly identity kernel

If analog computations were precise and not subject to noise, this kernel should not have any side-effect on AREG A. It is indeed:

1. dividing the content of analog register A by 2;
2. duplicating the content of A to F;
3. adding the content of A and F into AREG A.

It will serve as our basic operation that is repeatedly applied to uniform AREGs. While being very short and generic, it uses three basic operations very commonly used in convolution kernels produced by AUKE (division by 2, copy and addition). This kernel is applied up to 32 times, on original input values of 0, 5, 10, 60, 100 and 120. For statistical relevance, this operation is repeated 100 times. Pseudo-code in Algorithm 1 presents the complete process, and an example is shown in Figure 6.2.

```
for INPUT_VALUE in (0, 5, 10, 60, 100, 120):
  for ITERATION in (1...32):
    for K in (1...100):
      AREG_A = uniform(INPUT_VALUE)
      save AREG_A as input

      apply id_kernel ITERATIONS times on AREG_A
      save AREG_A as result
```

Algorithm 1: Noisy data collection

Figure 6.2: Left: crop of 10*10 pixels on a supposedly uniform input value of 100. Right: result after applying our identity kernel 4 times on the input. We can notice a global shift (right image is darker), and an increased amount of noise (right image has more internal variability).

The resulting images can be saved within the host application. We use this to collect 100 input/result pairs for each input value, for a total of 2*100*6*32=38400 images. We believed it important to not only save results, but inputs too, as they also are subject to random noise.
Given one input value (in $(0, 5, 10, 60, 100, 120)$) and one iteration value (in $[0, 32]$), we have at hand 100 input/result pair. Each pair is composed of 2 256*256 arrays, saved as 8 bits grey scale .BMP files. As in [49], we assume a noise model that is composed of a constant bias and a centered random component, as follows:

$$result = id\_kernel^{iteration}(input) = input + bias + random\_noise$$

with $\mathbb{E}[random\_noise] = 0$. We seek to quantitatively characterise both the bias and the random noise.

Systematic bias computation

Using the centered nature of the random noise, we simply compute the average of the mean difference between the inputs and the results to get an unbiased estimate of the bias, for each original input value i and iteration count j:

$$bias_{i,j} = \frac{1}{100} \cdot \left[\sum_{k=1}^{100} \frac{1}{256 \cdot 256} \cdot \sum_{1 \leq x,y \leq 256} ((input_{i,j,k})_{x,y} - (result_{i,j,k})_{x,y}) \right]$$

Signal to Noise Ratio computation

A common measure of image degradation between a clean target image and its noisy version is the Peak Signal to Noise Ratio (PSNR). It represents the similarity between two images. Expressed in dB, the higher its value, the less noise is introduced by computations.

For a given input/result pair, the PSNR is calculated as:

$$\begin{aligned} PSNR &= 20 \cdot log_{10}\left(\frac{max\_value}{mean\_square\_error}\right) \\ &= 20 \cdot log_{10}\left(\frac{255}{\sqrt{\frac{1}{256 \cdot 256} \cdot \sum_{1 \leq x,y \leq 256}(input_{x,y} - result_{x,y} - bias)^2}}\right) \end{aligned}$$

Notice that the mean square error is in fact calculated on images on which the systematic bias previously computed is subtracted, to compensate for it.

To get a more synthetic and meaningful result, we only report the average of the 100 values computed for one input value i and iteration count j.

Standard deviation computation

To measure the random variability introduced by stochastic noise, we compute its standard deviation (STD) within one image.

$$std = \sqrt{\frac{\sum_{1 \leq i,j \leq 256}\left(input_{i,j} - result_{i,j} - bias - mean\_dif\right)^2}{256 \cdot 256 - 1}}$$

where

$$mean\_dif = \frac{1}{256 \cdot 256} \cdot \sum_{1 \leq x,y \leq 256}(input_{x,y} - result_{x,y} - bias)$$

As is done with the PSNR metric, the 100 values of standard deviations we compute for one input value i and iteration count j are averaged.

Results

The results of the above computations are respectively reported in Figures 6.3, Figure 6.4 and Figure 6.5.

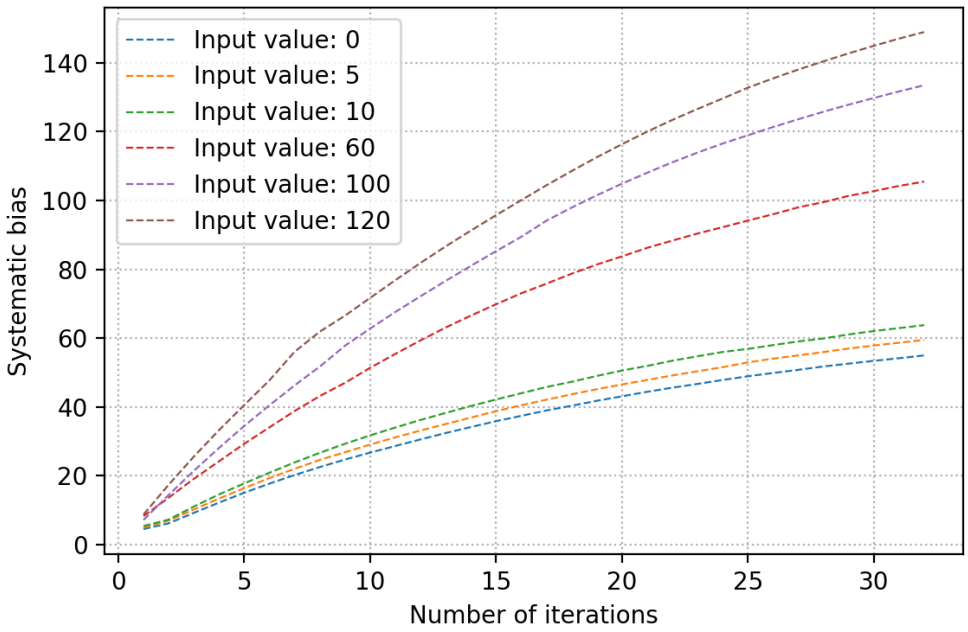


Figure 6.3: Evolution of the systematic bias when iterating a supposedly identity kernel up to 32 times, on 6 different input values.

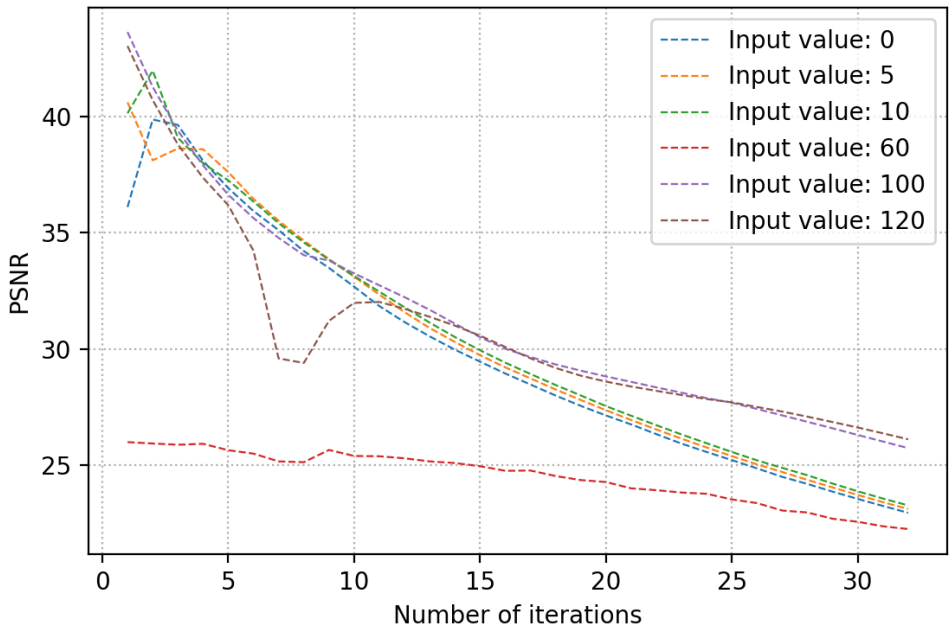


Figure 6.4: Evolution of the PSNR when iterating a supposedly identity kernel up to 32 times, on 6 different input values.

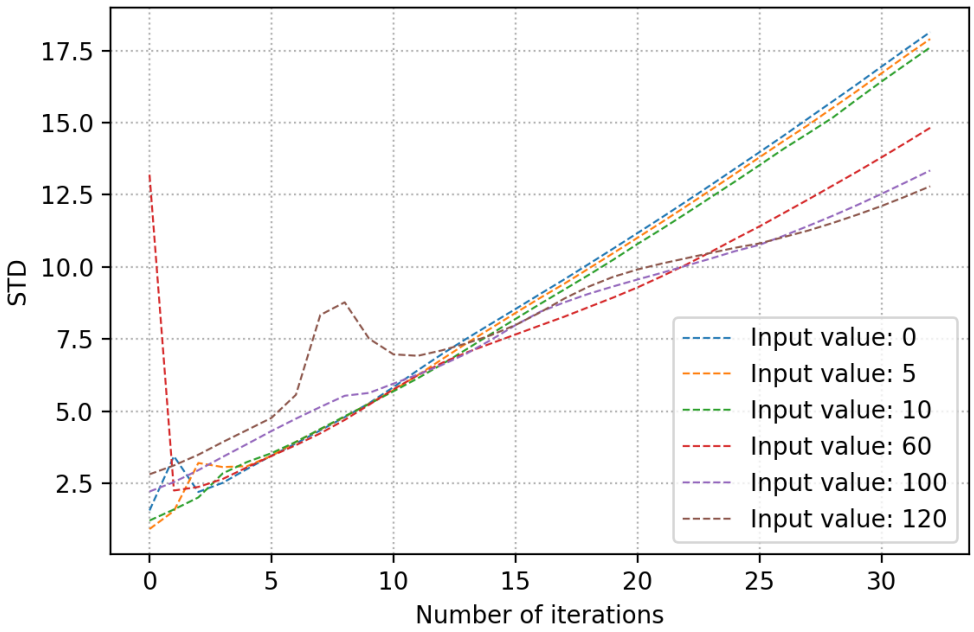


Figure 6.5: Evolution of the standard deviation when iterating a supposedly identity kernel up to 32 times, on 6 different input values.

We can notice that the more we iterate our supposedly identity kernel, the more noise is introduced, since the PSNR decreases and the STD increases. Moreover, the systematic bias is dependent on the input value in a seemingly increasing manner: the higher the original input value, the higher the systematic bias.

In addition to that, we can discern two phenomena that are slightly out of the general trend:

- With an input value of 120, there is a stronger than usual signal degradation when running the kernel 6 to 9 times. This is shown by the drop in PSNR value and the surge in STD value for these iteration numbers.

- With an input value of 60, the PSNR values seem considerably lower than with all other input values. This would suggest a stronger than usual signal degradation. However, as seen on Figure 6.5, the input image we create contains a lot of internal variability (ie. noise): the STD value for 0 iteration is abnormally high. A further investigation shows this noise appears in the form of multiple spiking pixels, as seen on Figure 6.6. Those pixels are averaged out by applying our almost-identity kernel, which explains why the STD values for iteration counts strictly superior to zero rejoin the general trend. However, the corresponding PSNR values are heavily distorted, since result images are compared to base images that are already noisy.

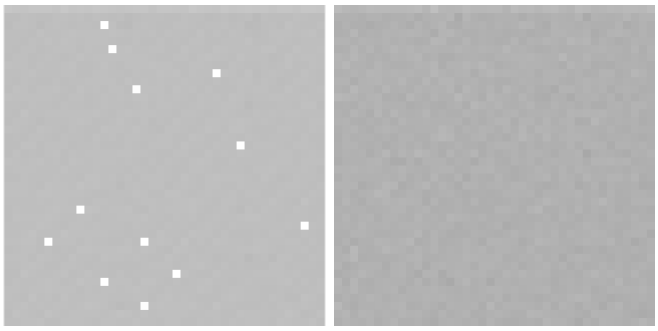


Figure 6.6: Left: crop of 40*40 pixels on a supposedly uniform input value of 60. Right: result after applying our identity kernel 3 times on the input. We can notice abnormal spiking pixels on the input, that are averaged out in the result.

The causes of these two observations remain unexplained, and multiple runs of the experiment all exhibit the same singularities. This seems to suggest that the camera's behaviour cannot be approximated by a linear model. Complex side effects come into play, and specific input values, combined with specific instructions result in out of trend results.

Comparison with the existing noise model

As explained in Section 3.3, the authors of [49] created a noise model for the SCAMP5 camera, which quantifies the systematic bias and the stochastic contribution for many low-level instructions. We here focus on the former, and compare their theoretical predictions with our empirical results.
To quantify the noise in advance, they fitted first order polynomials and created a Systematic Error Model, which notably describes the division and addition operations:

$$div2(x) = 0.482x + 3.39 \tag{6.1}$$

$$add(a, b) = min(0.958a + 0.930b + 6.86, 127) \tag{6.2}$$

Combining Equation 6.1 and Equation 6.2 for an input value of 100 gives the following derivation:

$$\begin{aligned} id\_kernel(100) &= add(div2(100), div2(100)) \\ &= add(0.482 \cdot 100 + 3.39, 0.482 \cdot 100 + 3.39) \\ &= add(51.59, 51.59) \\ &= min(0.958 \cdot 51.59 + 0.930 \cdot 51.59 + 6.86, 127) \\ &= min(104.26, 127) \\ &= 104.26 \end{aligned}$$

This gives a theoretical value for the systematic bias of 4.26 for one iteration of *id_kernel* on an input value of 100. The experimental data we collected exhibits an empirical value of 7.16, which is almost 70% more.
This major difference could partly be explained by the absence of such a polynomial noise model for the *MOV* operation, and for the creation of an uniform AREG content

- which both surely introduce noise and imprecision. Added to this, first order polynomials are incompatible with our observation that noise cannot be approximated as a linear phenomenon.
This calls the existing noise model into question. It might have been devised using another SCAMP5 device than ours, with some minor hardware differences, or in other environmental conditions - ambient temperature is for instance known to influence noise level of physical measurements. As a result, these polynomials are of no help in reliably approximating noise level in our case.

6.2 Tackling the issue of noise

In this section, we evoke possible future research tracks to attempt to tackle this issue of noise, given the current hardware capabilities. We did not have time to implement any of the following, but identified some possibly interesting ideas.

6.2.1 On the difficulty of accounting for noise during training

On possible idea would be to directly train convolution kernels that are robust to noise, by simulating an accurate synthetic noise in our digital computations - i.e. one that realistically simulates the inaccuracy of a SCAMP5 device.
The pitfall here would be *differentiability*. As presented in Subsection 2.2.1, the computational graph of a CNN needs to involve differentiable operations only, to allow for back-propagation and training. At the moment, the noise models we consider, in addition to being inaccurate as explained in Subsection 6.1.2, are all at the instruction level - i.e. they simulate noise for each FPSP instruction, such as addition, shifting or division. Using them for convolution kernels therefore requires the use of AUKE, to compile a kernel into FPSP code. As it is currently formulated, AUKE uses traditional branching instructions (`if`, `then`, `else`...) which are not differentiable. As seen from an optimiser such as Adam [27], AUKE is a black box breaking the computational graph, through which back-propagation and weights update is impossible.

A potentially very interesting future work could therefore be to devise a differentiable noise model, in the sense that it is usable during training. A way of doing so would be to **learn** it. For instance, a neural network could be trained to predict the standard deviation and bias when given a convolution kernel as input. Alternatively, a neural network could even be trained to be a full FPSP simulator, taking a convolution kernel and the content of an AREG as inputs, and predicting the content of the AREG as if it was convolved on the focal plane. Both cases would require gathering a lot of training data from a real FPSP, but present the great benefit of being fully differentiable by nature once trained.

6.2.2 Averaging methods for noise reduction

The most common practice to reduce random noise due to physically noisy measurement is to average results. Averaging n independent measurements results in a

scaling of the standard deviation in $1/\sqrt{n}$.

Temporal averaging

A possible solution to achieve this averaging would be to run the network several times on one input image, and average all the results. The stage at which averaging would take place still remains an open question, both in terms of technical feasibility and experimental optimality: one could average the final outputs, or each layer, or even collected events through a voting mechanism.

Spatial averaging

Another possible solution to implement this noise reduction technique would be with spatial averaging. For example, a CNN could be run on up-sampled inputs - say a 56*56 image instead of a 28*28 one, with each pixel optically duplicated 4 times. The convolutions could be applied using the technique developed in Section 4.4, and a voting process performed in each 2*2 square when collecting events.
This solution is basically equivalent to running the same network four times, but in only one pass. It takes the benefit of an advantage of our hardware (pixel parallel computations) to compensate for its major defect (noisy computations).

We have here discussed the issue of noise, and proposed new experiments to quantify it. We explained why it is the most serious obstacle to implementing multi-layer CNNs on the SCAMP5 device, and what could potentially by done to mitigate it. In the next chapter, we present a concrete real-world use case for the SCAMP5 device.

Chapter 7

Demonstration: steering direction prediction

In this chapter, we provide a new demonstration and use case for the SCAMP5 device, by using the vision sensor for an embedded robotic application. We use its capabilities to run simple CNNs at a very low power consumption and very high frame rates. As opposed to a traditional camera, an FPSP should provide meaningful and refined information extracted from a scene, and not a raw image that has yet to be interpreted by some central unit. We here apply knowledge and techniques developed earlier in these report, to prove their generic nature.

The high frame rate and low power consumption of the device make it a good fit for robotic applications. Typical network architectures used for robot navigation such as in [38], [43] or [32] involve tens of layers, each of them with tens of feature maps. This is far beyond what can currently be implemented on a SCAMP5 device. For this reason, we focus on a very simple example: robot line following. It is long known how to achieve it with very basic sensors and micro-controllers, but these solutions require domain specific knowledge. Here, we intend to demonstrate a completely novel approach by learning to solve this task in an end-to-end manner. The only requirement is that the SCAMP5 vision system should see the floor on which the line is drawn. To put it in a nutshell, we provide a proof-of-concept that uses a new paradigm applied to an otherwise long solved task.

7.1 Implementation

We use an existing robot platform, running ROS, an open-source Robot Operating System [42]. The wheeled platform is controlled by a Raspberry Pi running Raspbian. We aim at interfacing the SCAMP5 vision system via USB, and to have it issuing steering commands based on what the camera sees. Figure 7.1 shows our wheeled platform equipped with the SCAMP5 device.

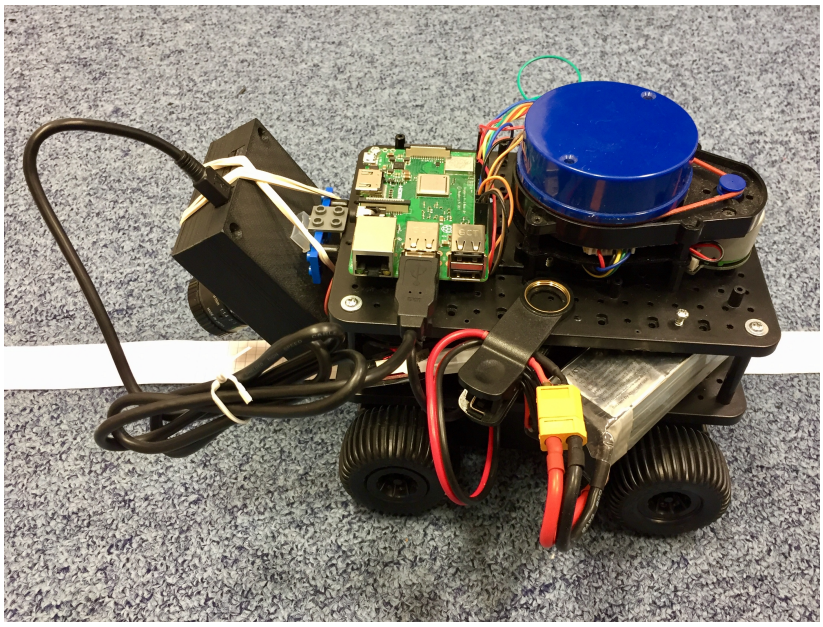


Figure 7.1: The wheeled robot, with the SCAMP5 vision system attached to it, interfaced via USB. We can also see the white line on the floor, made of paper strips.

7.1.1 CNN architecture

We use our standard AnalogNet2 architecture: a single layer CNN, composed of three 3*3 convolutions followed by a two layer fully connected network - with 50 hidden units and 3 output units. The three output units each correspond to a steering direction, that can either be to turn left, turn right, or continue straight. As such, this CNN is a classifier that recognises the direction to follow.
The most notable difference with AnalogNet2 used for MNIST is that we here need to preserve the whole visual field of view of the camera, and not the central 28*28 pixels. Preserving input images of size 256*256 pixels would imply reading too meany events on whole 256*256 DREGs, which would considerably slow down our inference time. To avoid this, we propose a slight optimisation, and only use the central 192*192 input pixels, that we downsample by a factor of 2 along each direction. To do this, we simply:

1. mask out 32 pixel borders on the input images,

2. mask out 1/4 of the pixels of the remaining central area, with a pattern similar to what is used for pooling (see Section 4.3),

3. use double shift instructions for running the 3*3 kernels, with values that are no longer neighbouring on the focal plane (see Section 4.4).

Moreover, we no longer use the new binning layout presented in , as each part of the image is judged equally important here. Each output feature map is hence divided into nine equally distributed bins, and the input of the fully connected layers is a vector of size 27.

7.1.2 Getting training data

To get training data, we remotely control the robot so that it follows a line we draw on the floor, with the SCAMP5 mounted on it. We then manually annotate the acquired frames, depending if it corresponds to a right or left run, or a straight line. The images are then down-sampled by a factor of 2 along each direction.

On these labelled images, a severe data augmentation process is run, in the form of random tilts of angles between -5 and +5, and random crops of size 96*96 pixels. The final dataset includes 1600 labelled sampled, of which 30% is reserved for testing, and the rest for training purposes. In our digital simulations, our very simple architecture reaches 75% testing accuracy.

Once the convolution kernels have been implemented on the SCAMP5 vision system, we once again collect training data, but this time in the form of vectors of size 27, which are the outputs of the convolutional part of the network. This noisy, in-situ data is used to re-train the fully connected layers.

7.2 Results

The full network running on the SCAMP5 device correctly interprets the scene to extract steering directions. Figure 7.2 shows a capture of the host program recording live data from the vision system, and a video is available at `https://youtu.be/xQ4vnRv1O0Y`. With a latency at 8.8 ms, the system can run at **110 FPS**, with a power budget of **15.2 mJ per frame**, or 1.7 Watts at full speed (including USB communication).

As seen on Figure 7.2, input images are extremely dense, with a lot of activated pixels. As a result, so are output feature maps, and an unrealistically high number of events has to be used to collect them - 5000 events per feature map, or 15000 in total. This is drastically higher than the previous architectures we explored, which required between 300 and 450 events to be collected to feed the fully connected layers. This explains why computations are much slower and voracious in energy in this case compared to AnalogNet2 for MNIST digits recognition, all other things being equal. As a comparison, the author of [49] report the Intel Movidius Myriad 2 Neural Compute Stick to be able to run AnalogNet in a power budget of 3.7 mJ per frame, which is strictly inferior to the 15.2 mJ we report here.

The very first improvement that one could bring to this architecture could be to extract and only preserve the borders of the input images before running the CNN. This can basically be done for free on the focal plane, and would make the input images much sparser. We could expect the output feature maps to also be sparse, and to possibly be collected in fewer events.

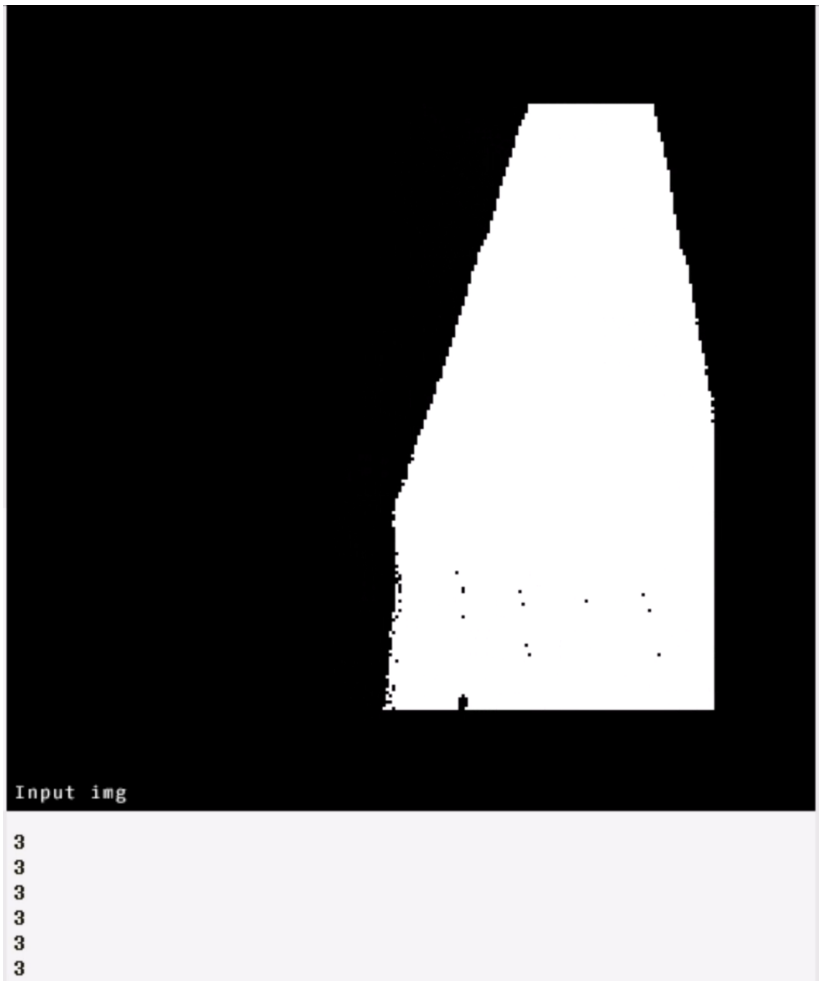


Figure 7.2: Capture of the SCAMP5 host program. Top: input image, showing the white line making a right turn, as seen from the vision system. Bottom: USB text message, corresponding to a right turn (3). Other possibilities not shown here are going forward (2) and turning left (1).

7.3 Robot control

Unfortunately, we were unable to make the SCAMP5 vision system control the robot. A lack of time and engineering issues related to powering the camera through the Raspberry Pi constituted obstacles we could not overcome in time. Our experiments also proved that setting a correct illumination and keeping it constant was also a topic on its own, as for temporally averaging and smoothing the SCAMP5's steering commands.
Sadly, we could not fully carry out this demonstration with the robot following a line on its own, based on instructions issued by the SCAMP5 vision system. However, we are confident enough to say that the CNN is able to accurately extract steering directions, which in addition to demonstrating serious computing capabilities in terms of network inference, provides an improvement in functionality - whereas AnalogNet2 mainly brought an improvement in performance.

Chapter 8

Conclusion and future work

We have presented the unusual computation paradigm of FPSPs, be it for its benefits and its restrictions. Despite being hard to apprehend and noisy, their pixel-wise parallel analog computations allow for the most competitive latency and energy consumption for some computer vision tasks. We have a clearer view of the broader landscape this work contributes to, at the intersection of deep-learning and re-configurable computing.

8.1 Contributions summary

Diving into technical implementation details, we were able to push the boundaries of CNN classification ever achieved on the SCAMP5 device - to the best of our knowledge. AnalogNet2 defines the new baseline on MNIST, with an in-situ testing accuracy of 96.9%. We also provided valuable advice for future CNNs on FPSP, be it the SCAMP5 device or another one. It takes the form of improvements in training, simulation, implementation easiness, inference speed and final accuracy.

Moreover, we explained the limitations we face in trying to port regular CNNs to our instance of an FPSP, the SCAMP5 device. We have clearly identified the issues that have to be overcome to implement a second convolutional layer, and the most promising way of doing it is using pooling operations or quantisation.

Finally, we devised reproductible experiments to quantify noise accumulation on the focal plane.

8.2 Future work

Software-wise, the most promising future work we see is to create a differentiable noise model. It could be used during the training phase, to create convolution kernels robust to noise, that could potentially be chained to create multi-layer CNNs performing better in-situ. It would help narrowing the gap between digital simulations and focal-plane computations, and ease the training process - the re-acquisition

and re-training process for the fully connected layers would no longer be needed. We see a potential solution in using deep learning to simulate noise. As explained in Section 6.2, another possibility would be to re-think network architectures and executions to mitigate noise, with averaging techniques for example.

Hardware-wise, the priority would be to develop a less noisy FPSP. This can easily be quantified with the experiments we provide, which could potentially be used to navigate the design space to propose a more accurate future version of SCAMP5.
We can accommodate the limited number of instructions on the focal plane (such as the absence of multiplication or division) and on the M0 core (such as integer only multiplications). We would however greatly benefit from an increased capacity in AREGs and DREGs, from the ability to write longer programs for the vision chip, and from a faster M0 core. Finally, having a dynamic range on AREGs centered around zero seems unoptimised for CNN applications, since half of it (the negative part) is wasted after applying a ReLU activation function.

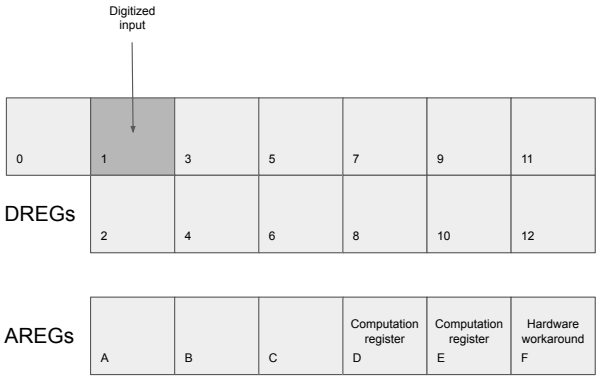
Finally, for new interesting demonstrations, we could imagine pairing the SCAMP5 device with a latest generation VPU such as Googles Edge TPU [13] to run an encoder-decoder architecture. The encoder part would run on the focal plane, only the code would be transmitted to the VPU that would execute the decoder. This association could allow an unbeatably low latency and power consumption for such diabolo networks.

Appendices

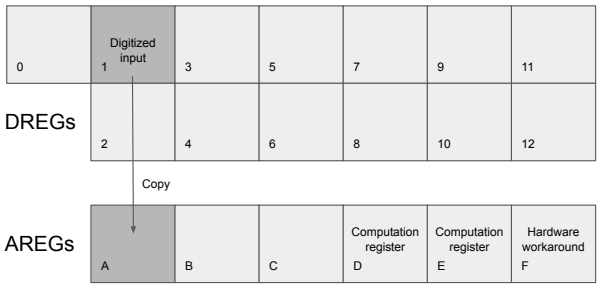
Appendix A

Two layer network using quantisation

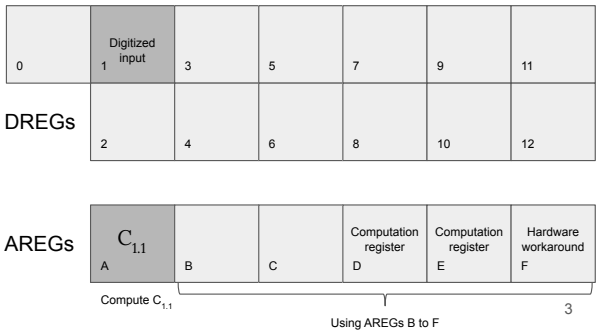
The following pages present the complete program flow of the CNN presented in Subsection 5.2.1. Each one of the 32 steps is numbered in the lower right corner, and shows the current operation and registers' content.


Digitized input
0
1
3
5
7
9
11
DREGs
2
4
6
8
10
12
AREGs
A
B
C
Computation register
D
Computation register
E
Hardware workaround
F

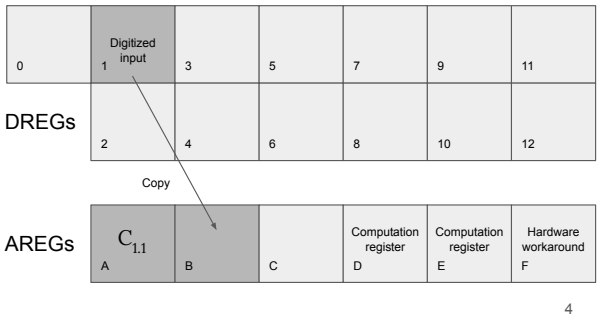
1


Digitized input
Copy
DREGs
AREGs
Computation register
Hardware workaround

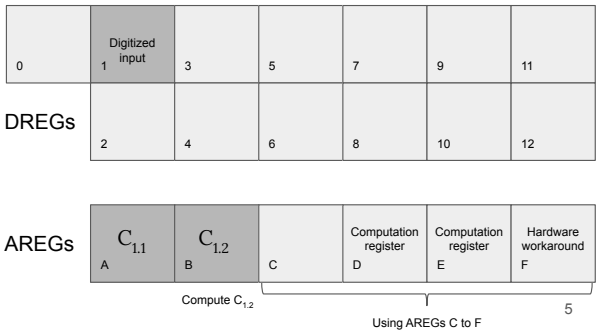
2


Digitized input
DREGs
AREGs
$C_{1.1}$
Computation register
Hardware workaround
Compute $C_{1.1}$
Using AREGs B to F

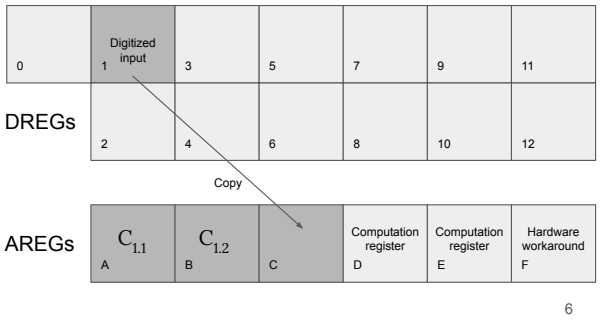
3


Digitized input
Copy
DREGs
AREGs
$C_{1.1}$
Computation register
Hardware workaround

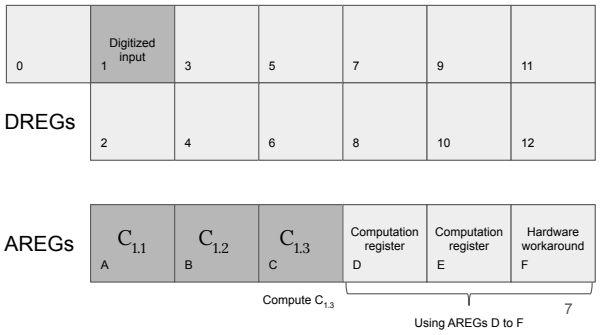
4


Digitized input
DREGs
AREGs
$C_{1.1}$
$C_{1.2}$
Computation register
Hardware workaround
Compute $C_{1.2}$
Using AREGs C to F

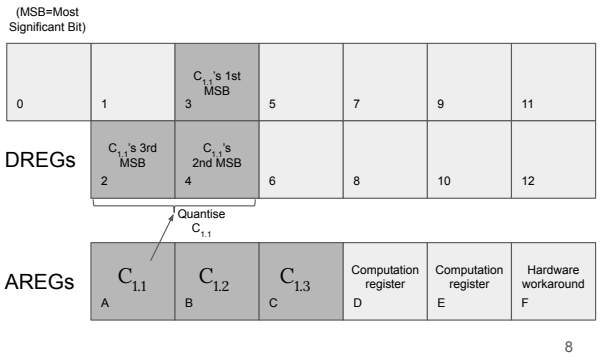
5


Digitized input
Copy
DREGs
AREGs
$C_{1.1}$
$C_{1.2}$
Computation register
Hardware workaround

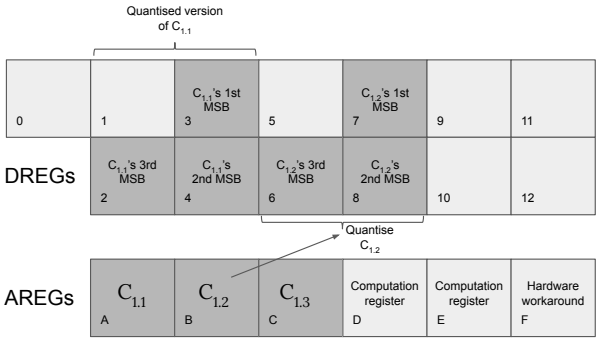
6


Digitized input
DREGs
AREGs
$C_{1.1}$
$C_{1.2}$
$C_{1.3}$
Computation register
Hardware workaround
Compute $C_{1.3}$
Using AREGs D to F

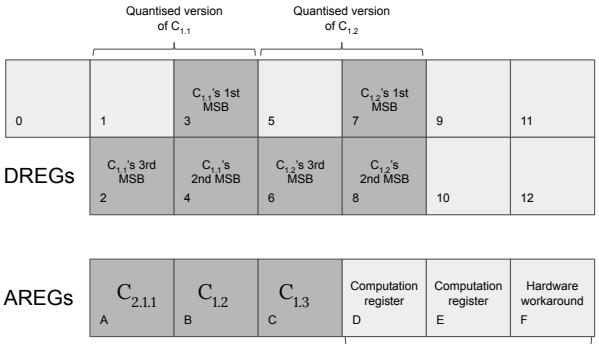
7


(MSB=Most Significant Bit)
$C_{1.1}$'s 1st MSB
$C_{1.1}$'s 3rd MSB
$C_{1.1}$'s 2nd MSB
Quantise $C_{1.1}$
DREGs
AREGs
$C_{1.1}$
$C_{1.2}$
$C_{1.3}$
Computation register
Hardware workaround

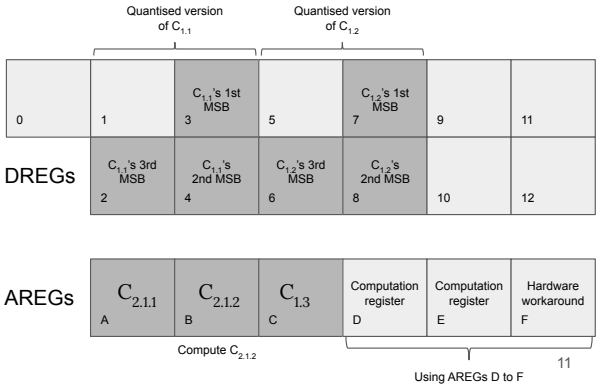
8



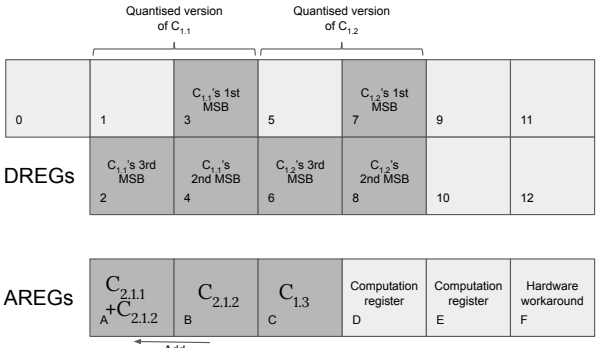
9



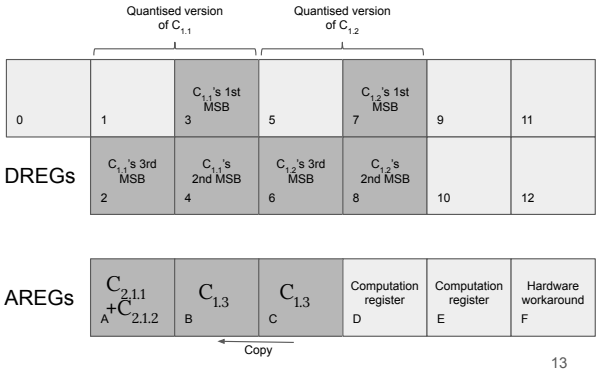
10



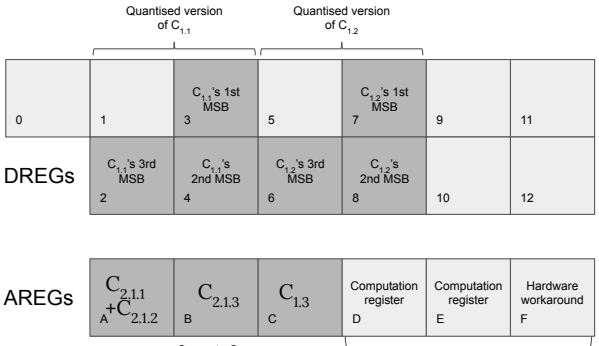
11



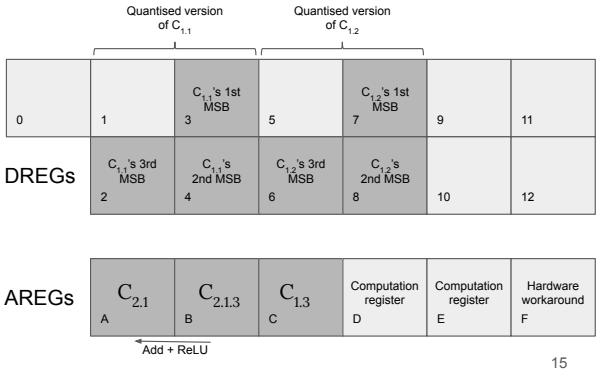
12



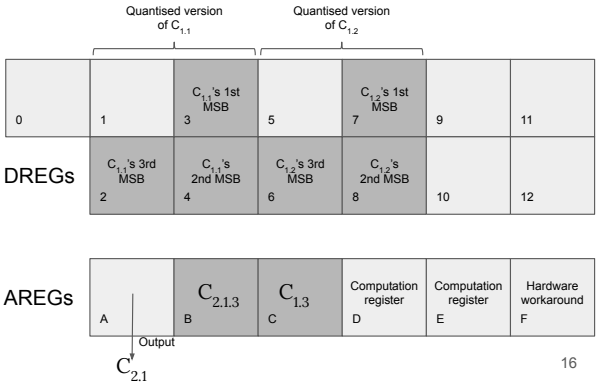
13



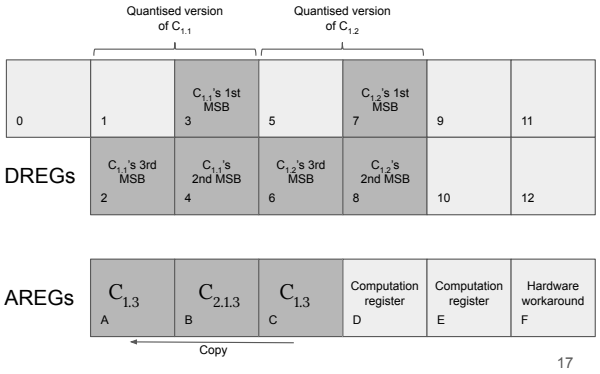
14



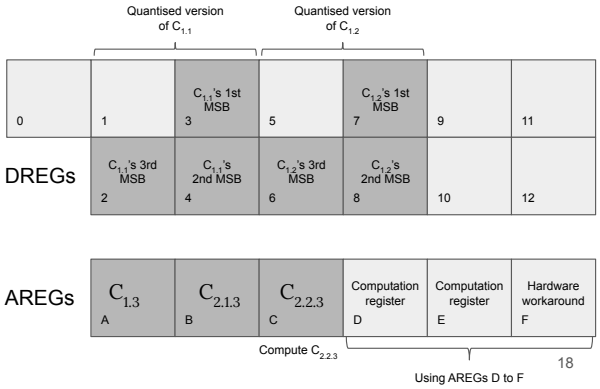
15



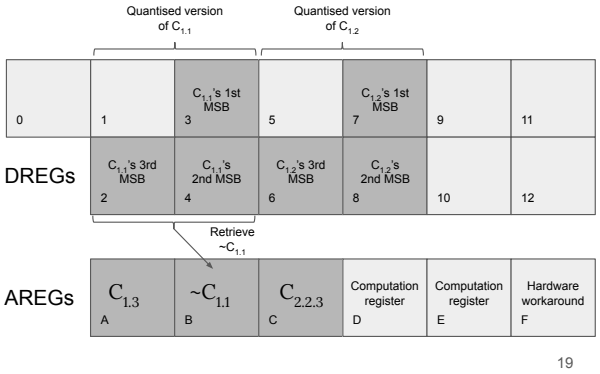
16



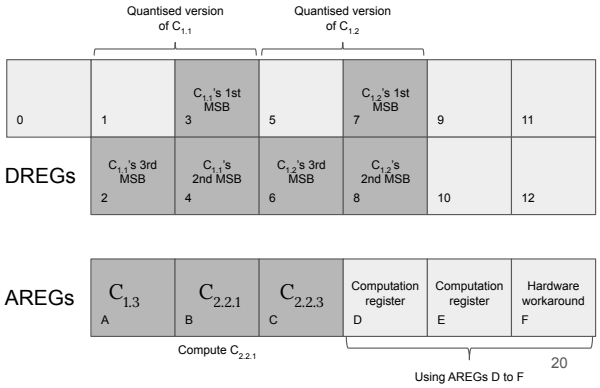
17



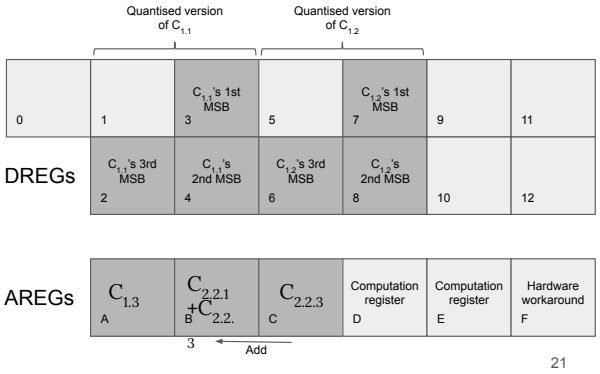
18



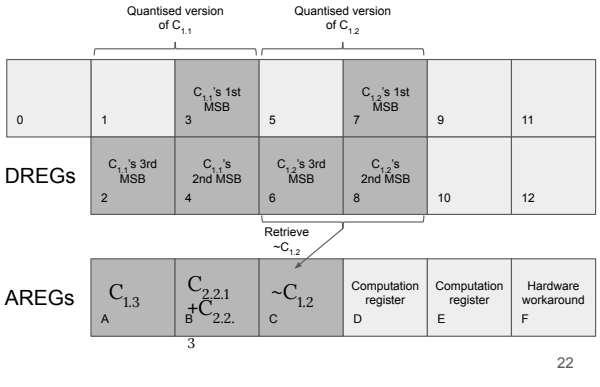
19



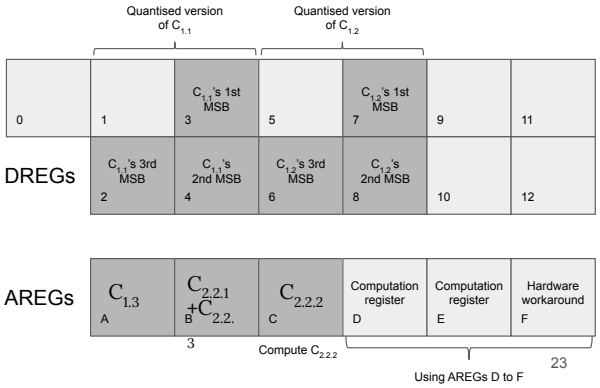
20



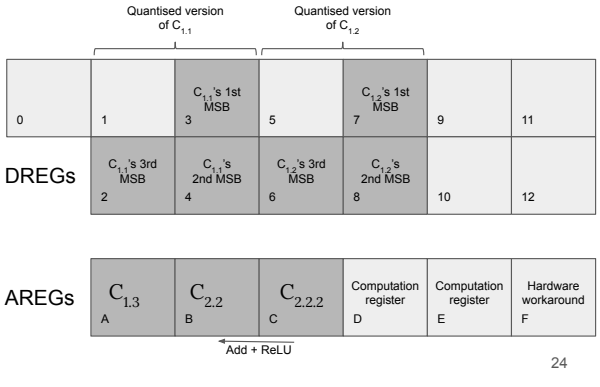
21



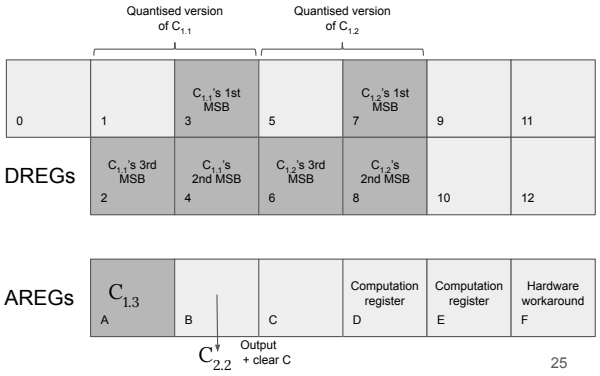
22

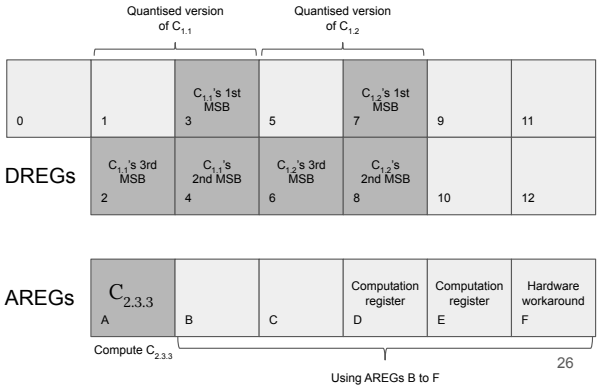


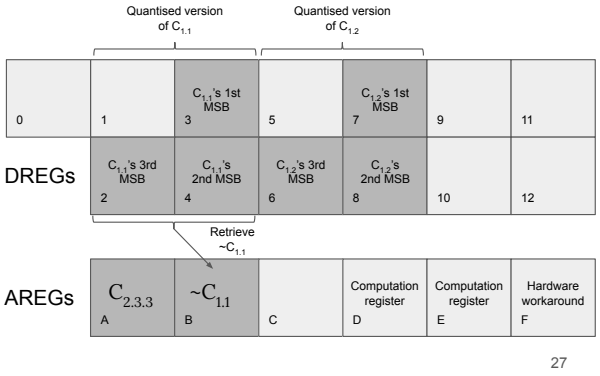
23

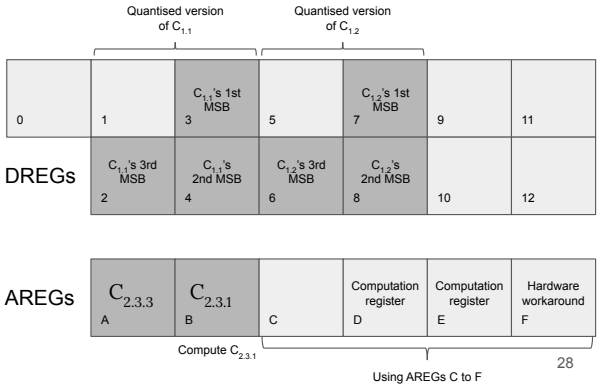


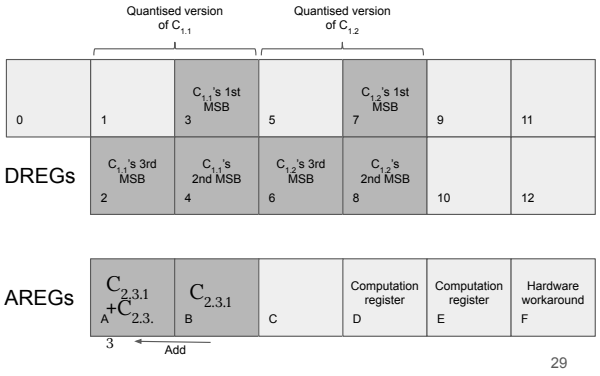
24

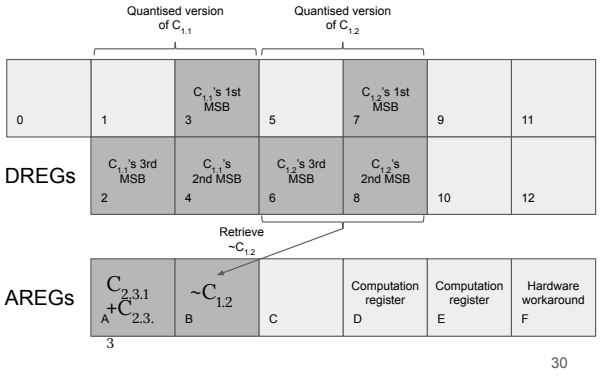

Quantised version of $C_{1.1}$
Quantised version of $C_{1.2}$
0
1
3
$C_{1.1}$'s 1st MSB
5
7
$C_{1.2}$'s 1st MSB
9
11
DREGs
2
$C_{1.1}$'s 3rd MSB
4
$C_{1.1}$'s 2nd MSB
6
$C_{1.2}$'s 3rd MSB
8
$C_{1.2}$'s 2nd MSB
10
12
AREGs
A
$C_{1.3}$
B
C
D
Computation register
E
Computation register
F
Hardware workaround
$C_{2.2}$
Output + clear C
25

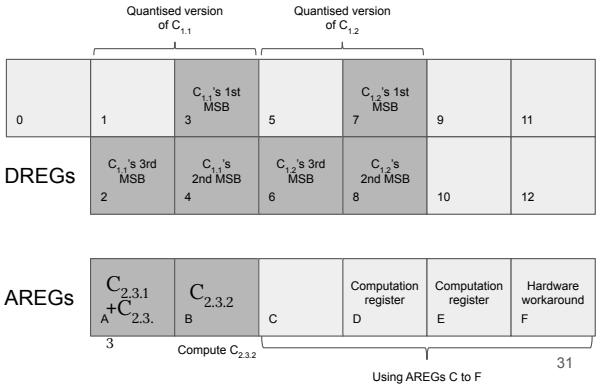

Quantised version of $C_{1.1}$
Quantised version of $C_{1.2}$
0
1
3
$C_{1.1}$'s 1st MSB
5
7
$C_{1.2}$'s 1st MSB
9
11
DREGs
2
$C_{1.1}$'s 3rd MSB
4
$C_{1.1}$'s 2nd MSB
6
$C_{1.2}$'s 3rd MSB
8
$C_{1.2}$'s 2nd MSB
10
12
AREGs
A
$C_{2.3.3}$
B
C
D
Computation register
E
Computation register
F
Hardware workaround
Compute $C_{2.3.3}$
Using AREGs B to F
26

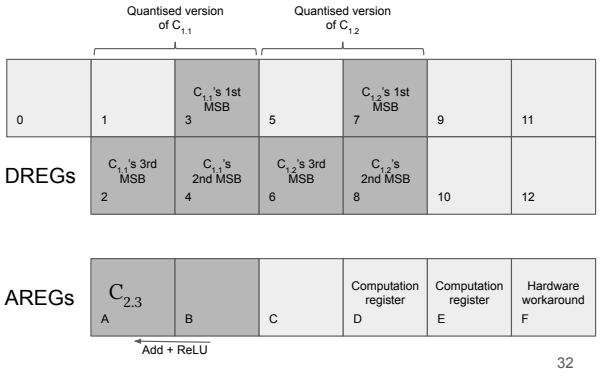

Quantised version of $C_{1.1}$
Quantised version of $C_{1.2}$
0
1
3
$C_{1.1}$'s 1st MSB
5
7
$C_{1.2}$'s 1st MSB
9
11
DREGs
2
$C_{1.1}$'s 3rd MSB
4
$C_{1.1}$'s 2nd MSB
6
$C_{1.2}$'s 3rd MSB
8
$C_{1.2}$'s 2nd MSB
10
12
Retrieve $\sim C_{1.1}$
AREGs
A
$C_{2.3.3}$
B
$\sim C_{1.1}$
C
D
Computation register
E
Computation register
F
Hardware workaround
27


Quantised version of $C_{1.1}$
Quantised version of $C_{1.2}$
0
1
3
$C_{1.1}$'s 1st MSB
5
7
$C_{1.2}$'s 1st MSB
9
11
DREGs
2
$C_{1.1}$'s 3rd MSB
4
$C_{1.1}$'s 2nd MSB
6
$C_{1.2}$'s 3rd MSB
8
$C_{1.2}$'s 2nd MSB
10
12
AREGs
A
$C_{2.3.3}$
B
$C_{2.3.1}$
C
D
Computation register
E
Computation register
F
Hardware workaround
Compute $C_{2.3.1}$
Using AREGs C to F
28


Quantised version of $C_{1.1}$
Quantised version of $C_{1.2}$
0
1
3
$C_{1.1}$'s 1st MSB
5
7
$C_{1.2}$'s 1st MSB
9
11
DREGs
2
$C_{1.1}$'s 3rd MSB
4
$C_{1.1}$'s 2nd MSB
6
$C_{1.2}$'s 3rd MSB
8
$C_{1.2}$'s 2nd MSB
10
12
AREGs
A
$C_{2.3.1}$ +$C_{2.3.}$
B
$C_{2.3.1}$
C
D
Computation register
E
Computation register
F
Hardware workaround
3
Add
29


Quantised version of $C_{1.1}$
Quantised version of $C_{1.2}$
0
1
3
$C_{1.1}$'s 1st MSB
5
7
$C_{1.2}$'s 1st MSB
9
11
DREGs
2
$C_{1.1}$'s 3rd MSB
4
$C_{1.1}$'s 2nd MSB
6
$C_{1.2}$'s 3rd MSB
8
$C_{1.2}$'s 2nd MSB
10
12
Retrieve $\sim C_{1.2}$
AREGs
A
$C_{2.3.1}$ +$C_{2.3.}$
B
$\sim C_{1.2}$
C
D
Computation register
E
Computation register
F
Hardware workaround
3
30


Quantised version of $C_{1.1}$
Quantised version of $C_{1.2}$
0
1
3
$C_{1.1}$'s 1st MSB
5
7
$C_{1.2}$'s 1st MSB
9
11
DREGs
2
$C_{1.1}$'s 3rd MSB
4
$C_{1.1}$'s 2nd MSB
6
$C_{1.2}$'s 3rd MSB
8
$C_{1.2}$'s 2nd MSB
10
12
AREGs
A
$C_{2.3.1}$ +$C_{2.3.}$
B
$C_{2.3.2}$
C
D
Computation register
E
Computation register
F
Hardware workaround
3
Compute $C_{2.3.2}$
Using AREGs C to F
31


Quantised version of $C_{1.1}$
Quantised version of $C_{1.2}$
0
1
3
$C_{1.1}$'s 1st MSB
5
7
$C_{1.2}$'s 1st MSB
9
11
DREGs
2
$C_{1.1}$'s 3rd MSB
4
$C_{1.1}$'s 2nd MSB
6
$C_{1.2}$'s 3rd MSB
8
$C_{1.2}$'s 2nd MSB
10
12
AREGs
A
$C_{2.3}$
B
C
D
Computation register
E
Computation register
F
Hardware workaround
Add + ReLU
32

Appendix B

Ethical and professional considerations

Our exploratory work can be considered as providing new primitives for very high throughput and low energy embedded CNN inference. It hence opens the way to potentially new and unforeseen applications. A small, embedded vision system, with advanced feature extraction capabilities and extremely long battery life can be used for numerous applications. Some, such as wildlife protection or crop surveillance, could be a boon for tackling modern days issues. Others, such as hidden autonomous snooping, could arguably raise important privacy concerns, and be used for criminal purposes. The range of potential applications even extends to the military domain, be it for area of operations surveillance or in the form of new wearable vision systems.
We however consider this considerations to be relevant for any 'foundational' work, especially when related to image processing and computer vision. It should therefore not refrain us in pursuing our work.
The following two pages include the ethics checklist provided by the College.

	Yes	No
Section 1: HUMAN EMBRYOS/FOETUSES		
Does your project involve Human Embryonic Stem Cells?		✓
Does your project involve the use of human embryos?		✓
Does your project involve the use of human foetal tissues / cells?		✓
Section 2: HUMANS		
Does your project involve human participants?		✓
Section 3: HUMAN CELLS / TISSUES		
Does your project involve human cells or tissues? (Other than from "Human Embryos/Foetuses" i.e. Section 1)?		✓
Section 4: PROTECTION OF PERSONAL DATA		
Does your project involve personal data collection and/or processing?		✓
Does it involve the collection and/or processing of sensitive personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction)?		✓
Does it involve processing of genetic information?		✓
Does it involve tracking or observation of participants? It should be noted that this issue is not limited to surveillance or localization data. It also applies to Wan data such as IP address, MACs, cookies etc.		✓
Does your project involve further processing of previously collected personal data (secondary use)? For example Does your project involve merging existing data sets?		✓
Section 5: ANIMALS		
Does your project involve animals?		✓
Section 6: DEVELOPING COUNTRIES		
Does your project involve developing countries?		✓
If your project involves low and/or lower-middle income countries, are any benefit-sharing actions planned?		✓
Could the situation in the country put the individuals taking part in the project at risk?		✓
Section 7: ENVIRONMENTAL PROTECTION AND SAFETY		
Does your project involve the use of elements that may cause harm to the environment, animals or plants?		✓
Does your project deal with endangered fauna and/or flora /protected areas?		✓
Does your project involve the use of elements that may cause harm to humans, including project staff?		✓
Does your project involve other harmful materials or equipment, e.g. high-powered laser systems?		✓
Section 8: DUAL USE		
Does your project have the potential for military applications?	✓	
Does your project have an exclusive civilian application focus?		✓
Will your project use or produce goods or information that will require export licenses in accordance with legislation on dual use items?		✓
Does your project affect current standards in military ethics – e.g., global ban on weapons of mass destruction, issues of proportionality, discrimination of combatants and accountability in drone and autonomous robotics developments, incendiary or laser weapons?		✓
Section 9: MISUSE		

Does your project have the potential for malevolent/criminal/terrorist abuse?	✓	
Does your project involve information on/or the use of biological-, chemical-, nuclear/radiological-security sensitive materials and explosives, and means of their delivery?		✓
Does your project involve the development of technologies or the creation of information that could have severe negative impacts on human rights standards (e.g. privacy, stigmatization, discrimination), if misapplied?	✓	
Does your project have the potential for terrorist or criminal abuse e.g. infrastructural vulnerability studies, cybersecurity related project?		✓
SECTION 10: LEGAL ISSUES		
Will your project use or produce software for which there are copyright licensing implications?		✓
Will your project use or produce goods or information for which there are data protection, or other legal implications?		✓
SECTION 11: OTHER ETHICS ISSUES		
Are there any other ethics issues that should be taken into consideration?		✓

✓

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] FE Allen and J Cocke. *A catalogue of optimizing transformations*. IBM Thomas J. Watson Research Center, 1971.

[3] Chris Angelini. Nvidia geforce gtx 1080 ti 11gb review. `https://www.tomshardware.com/reviews/nvidia-geforce-gtx-1080-ti,4972-6.html`, 2017. [Online; accessed 5-June-2019].

[4] Apple. Apple developper documentation: Blurring an image. `https://developer.apple.com/documentation/accelerate/vimage/blurring_an_image`. [Online; accessed 4-June-2019].

[5] Laurie Bose, Jianing Chen, Stephen J. Carey, Piotr Dudek, and Walterio Mayol-Cuevas. Live demonstration: Digit recognition on pixel processor arrays. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019.

[6] Christian Brandli, Raphael Berner, Minhao Yang, Shih-Chii Liu, and Tobi Delbrück. A 240 * 180 130 db 3 μs latency global shutter spatiotemporal vision sensor. *J. Solid-State Circuits*, 49(10):2333–2341, 2014.

[7] S. J. Carey, A. Lopich, D. R. W. Barr, B. Wang, and P. Dudek. A 100,000 fps vision sensor with embedded 535gops/w 256*256 simd processor array. In *2013 Symposium on VLSI Circuits*, pages C182–C183, June 2013.

[8] Stephen J. Carey, David Robert Wallace Barr, and Piotr Dudek. Low power high-performance smart camera system based on SCAMP vision sensor. *Journal of Systems Architecture - Embedded Systems Design*, 59(10-A):889–899, 2013.

[9] Chen Chen, Qifeng Chen, Jia Xu, and Vladlen Koltun. Learning to see in the dark. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3291–3300, 2018.

[10] Jianing Chen. Scamp5d vision system. `https://personalpages.manchester.ac.uk/staff/jianing.chen/`, 2019. [Online; accessed 9-August-2019].

[11] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, Dec 2014.

[12] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pages 262–263, 2016.

[13] Coral. Coral tpu usb accelerator datasheet. `https://coral.withgoogle.com/docs/accelerator/datasheet/`. [Online; accessed 16-August-2019].

[14] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[15] Andrew J. Davison. Futuremapping: The computational structure of spatial AI systems. *CoRR*, abs/1803.11288, 2018.

[16] Thomas Debrunner, Sajad Saeedi, and Paul H. J. Kelly. AUKE: automatic kernel code generation for an analogue SIMD focal-plane sensor-processor array. *TACO*, 15(4):59:1–59:26, 2019.

[17] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[18] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[19] G Gybenko. Approximation by superposition of sigmoidal functions. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[20] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[22] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[23] Intel. Intel® neural compute stick 2. `https://software.intel.com/en-us/neural-compute-stick`. [Online; accessed 16-August-2019].

[24] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877, 2017.

[25] JeVoisInc. Jevois smart machine vision. `https://www.jevoisinc.com/`. [Online; accessed 16-August-2019].

[26] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition. `http://cs231n.github.io/convolutional-networks/`, 2018. [Online; accessed 4-June-2019].

[27] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[29] E Kussul, T Baidyk, L Kasatkina, and V Lukovich. Rosenblatt perceptrons for handwritten digit recognition. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222)*, volume 2, pages 1516–1520. IEEE, 2001.

[30] John R Levine. *Linkers & loaders*. Morgan Kaufmann Publishers., 2000.

[31] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[32] Antonio Loquercio, Ana I. Maqueda, Carlos R. del-Blanco, and Davide Scaramuzza. Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3(2):1088–1095, 2018.

[33] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[34] Julien N. P. Martel, Miguel Chau, Matthew Cook, and Piotr Dudek. Pixel interlacing to trade off the resolution of a cellular processor array against more registers. In *European Conference on Circuit Theory and Design, ECCTD 2015, Trondheim, Norway, August 24-26, 2015*, pages 1–4. IEEE, 2015.

[35] Julien NP Martel, Lorenz K Müller, Stephen J Carey, and Piotr Dudek. High-speed depth from focus on a programmable vision chip using a focus tunable lens. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, 2017.

[36] Tom M. Mitchell. *Machine learning, International Edition*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[37] Nvidia. Jetson nano developer kit. `https://developer.nvidia.com/embedded/jetson-nano-developer-kit`. [Online; accessed 16-August-2019].

[38] Daniele Palossi, Antonio Loquercio, Francesco Conti, Eric Flamand, Davide Scaramuzza, and Luca Benini. Ultra low power deep-learning-powered autonomous nano drones. *CoRR*, abs/1805.01831, 2018.

[39] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration. *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration*, 6, 2017.

[40] Prophesee. Prophesee event based cameras. `https://www.prophesee.ai/event-based-evk/`. [Online; accessed 12-August-2019].

[41] Yunchen Pu, Zhe Gan, Ricardo Henao, Xin Yuan, Chunyuan Li, Andrew Stevens, and Lawrence Carin. Variational autoencoder for deep learning of images, labels and captions. In *Advances in neural information processing systems*, pages 2352–2360, 2016.

[42] Morgan Quigley, Brian P. Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.

[43] Lingyan Ran, Yanning Zhang, Qilin Zhang, and Tao Yang. Convolutional neural network-based robot navigation using uncalibrated spherical images. *Sensors*, 17(6):1341, 2017.

[44] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[45] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[46] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[47] Laurent Sifre and Stéphane Mallat. Rigid-motion scattering for texture classification. *CoRR*, abs/1403.1687, 2014.

[48] Alexander Toshev and Christian Szegedy. Deeppose: Human pose estimation via deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1653–1660, 2014.

[49] Matthew Wong, Sajad Saeedi, and Paul H. J. Kelly. Analog vision - neural network inference acceleration using analog SIMD computation in the focal plane. Master's thesis, Imperial College London - Department of Computing, 2018.

[50] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[51] Ákos Zarándy. *Focal-plane sensor-processor chips*. Springer Science & Business Media, 2011.

[52] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *CoRR*, abs/1702.03044, 2017.