Imperial College London

Imperial College London

Department of Computing

Neural Cellular Automata on a Focal Plane

Author:
Maciej Dudziak

Supervisor:
Prof. Paul Kelly

Second Marker:
Prof. Wayne Luk

Submitted in partial fulfillment of the requirements for the MSc degree in Computing Science / Visual Computing and Robotics of Imperial College London

September 2, 2021

Abstract

The cellular automata is a discrete mathematical model of computation that consists of a regular grid of cells, each being in one of the finite number of states, with its next iteration determined by a set of rules applied locally at each cell. Traditionally those rules are predefined and crafted for a specific, desired operation, though recently it has been shown that they can also be learned with convolutional neural networks and applied to perform image classification or segmentation. The locality of the computation and presence of independent and organised agents resembles the architecture of the Focal Plane Sensor Processors (FPSP), a family of vision chips where every pixel/photo-diode in the array is equipped with a processing unit allowing for computations directly on a focal plane.

In this thesis, we investigate the suitability of FPSP architectures for hosting the neural cellular automata (NCA) by implementing an example automaton for the self-classification of MNIST digits. In the process, we present a comprehensive analysis of the computational and memory requirements of the NCA and relates them to the capabilities of existing FPSP devices, providing guidelines for the development of automata that is less resource consuming. We develop a step-by-step quantisation procedure customised for the application of neural cellular automata on FPSP, which reduces the 32-bit floating-point model into the integer representation with 5-bit weights, 10-bit input and 9-bit activations while entirely maintaining its performance and precision. Our implementation of this quantised NCA achieves an operation at **74 FPS** with using **77.68 mJ** energy per frame on the simulated FPSP device.

This work could also be treated as an exploration of neural cellular automata as a potential general computation algorithm for the FPSP family architectures and also as an exploration for the possible development directions of the next generation of FPSPs. Hence, we conclude this thesis with a discussion on the discovered bottlenecks and limitations and describing the ideas on possible solutions, such as more sophisticated patch multiplexing/superpixels.

Acknowledgements

I would like to thank Professor Paul Kelly and Edward Stow for all their guidance and support throughout the project and the time devoted to our weekly meetings, which helped me to stay on track and maintain motivation.

I would like to thank Ciro Cursio for his invested time and advice considering the network quantisation. I would also like to thank Jamal Mulla for his support with configuring and using the FPSP simulator which played an essential role in created implementation.

I am also grateful to my entire family and friends, in particular to Patryk K., for their continuous support and encouragement.

Contents

List of Tables

List of Figures

List of Abbreviations

BN Batch Normalisation

CNN Convolutional Neural Network

FPSP Focal Plane Sensor Processor

INQ Incremental Network Quantisation

MNIST NCA Self-Classifying MNIST Digits Automata

NCA Neural Cellular Automata

TFLite TensorFlow Lite Framework

Chapter 1

Introduction

1.1 Motivations

The cellular automaton is a well-known mathematical discrete model of computation, that is a subject of numerous research in automata theory and has found its application in physics as well as theoretical biology. Its dynamics are determined by the set of pre-defined rules which computes the next iteration's state of each cell in the grid based on its small neighbourhood. The most common cellular automaton example, that perfectly illustrates this principle is Conway's Game of Life which operates in the 2D orthogonal grid and allows cells to be either 'ALIVE' or 'DEAD', determining their next iteration's value based on the number of alive cells in the neighbourhood. Even though driven by simple rules, it exhibits diverse behaviour with the presence of oscillatory, permanent as well as moving shapes.

Most recently it has been shown that when embedded with a neural network to compute the rules, a (neural) cellular automaton can be applied to grow stable and regenerative patterns [32], build self-classifying digits [36] or even to segment an image [40]. These examples provide a view on the potential of neural cellular automata and open a question of further possible applications in computer vision and beyond.

In this work, we turn our attention to its evident resemblance with the Focal Plane Sensor Processors (FPSP) architectures. The FPSPs could be summarised as Same-Instruction Multiple-Data parallel vision chips, where each pixel in the focal plane has its processing element that can access data of direct neighbours and perform calculations, including computation of convolutional kernels [6, 7, 47]. Consequently, it appears as a suitable platform to execute neural cellular automaton directly on the focal plane, eliminating the necessity of expensive transfer of captured raw images and so might result in fast and efficient processing. The cellular automata could also be a general model of computation for the FPSP devices, even transferred to vision unrelated applications which include computing on a regular grid, for example, a grid of sensing devices.

We investigate this topic by the end-to-end implementation of the Self-Classifying MNIST Digits cellular automaton [36] on the FPSP architecture, providing analysis of discovered bottlenecks and limitations, and applying a range of optimizations to mitigate their impact. We approach the problem of efficient and precise quantization of the automaton and look at the internal dynamics examining the property of continuous stability and methods to improve it.

This work could also be viewed as a search for the most beneficial development directions for the future FPSP devices, such as SCAMP5, so they could facilitate more complex applications which might take advantage of their distinctive architecture to outperform existing alternatives.

1.2 Contributions

The main contributions of our work can be organised as follows:

- We provide an analysis of the current capabilities of state-of-the-art FPSP devices in terms of the Neural Cellular Automata and point out the architectural limitations, such as analogue arithmetic and storage, insufficient per pixel memory, and computation bottlenecks, and suggesting possible solutions. We present outlines for the optimisation of Neural Cellular Automata and the development of models that are more suitable for FPSP devices and can moderate the severity of their limitations, such as restraining the number of hidden channels. We propose a new NCA architecture for self-classifying MNIST digits that is four times smaller than the original and optimised for more efficient FPSP implementation at the cost of 8% accuracy degradation compared to the original automaton.
- We report an analysis of the Neural Cellular Automata internal dynamics, concentrating on the problem of the long-term stability and accuracy of the automaton. We show the study on training routine modification and application of the network normalization, reporting excellent results achieved by the Batch-Normalisation on stabilizing the exemplary model's dynamics and improving its performance.
- We successfully apply two different quantisation algorithms to the NCA model, an end-to-end approach offered by the TensorFlow Lite framework [45] and a customised incremental quantisation method inspired by the Incremental Network Quantization by Zhou et al. [50]. We develop a step-by-step procedure for applying those schemes in the context of FPSP devices and compare both approaches in terms of resulting model size and final performance. Our best, quantised architecture reduces each weight to *5 bits* and activations to *9/10 bits* while maintaining the performance of a 32-bit floating-point counterpart.
- We implement the proposed, optimised NCA architecture for the self-classifying MNIST digits in the FPSP device simulator from the two above quantisation methods, maintaining their precision and properties from the experiments in Python. We report on the efficiency of the final device code and achieve execution of the neural cellular automaton at 74 FPS for the model quantised with the customised, incremental approach.

1.3 Outline

This report is organised as follows, with Chapter 2 providing a synthesis of the background topics that this project builds on. In Chapter 3 we present the related work and publications in three main areas, neural cellular automata, FPSP devices, and convolutional networks quantisation, which are a base for the extensions and contributions of this work. The detailed analysis of the Neural Cellular Automata in terms of FPSP devices are reported in Chapter 4, in particular, the presentation of hardware requirements as well as the problem of continuous stability of the automaton. We then show the quantisation of the Neural Cellular Automata with two different approaches and provide a comparison in Chapter 5. Chapter 6 details the process of the actual implementation of NCA on the FPSP device simulator and gives an evaluation of obtained results. We end with Chapter 7 summarising achieved results and contributions and introducing the directions of possible future work.

Chapter 2

Background

This chapter reviews the relevant background material, starting with the definition and general characteristics of the Focal Plane Sensor Processors. We then follow with the description of the cellular automata fundamentals, finishing with the summary about the relevant elements of the convolutional neural networks.

2.1 Focal Plane Sensor Processors

Focal Plane Sensor Processors, often also equivalent to Vision Chips, Pixel Processor Arrays (PPAs), or Cellular Processor Arrays (CPAs), are the family of architectures that targets computer vision applications, by extending the camera sensor with the capability of processing the acquired image directly on it. Depending on the actual architecture, the FPSP devices equip a group, or every, pixel in the sensor array with a programmable processing element able to access data of its neighbours, usually in a north-south-east-west pattern, and perform relatively simple computation. The execution model in many of those devices obeys the Same-Instruction Multiple-Data type of parallel processing as each processing unit executes the same instruction set on its data and the ones obtained direct neighbours.

The principle is to enable low-latency and power-efficient computation directly on the focal plane and if that's the part of the application, limit the amount of data that has to be sent further. For example, simple edge detection with Sobel filters can be performed with latency as low as few milliseconds, such as $4.7\mu s$ reported by Komuro and Ishikawa [24], and only the locations that contain the edge could be reported to further, even in the binary format, hence limiting data transfer significantly and allowing operation at even 100k Frames-Per-Second [7].

The examples of the FPSP architectures are the vision chip by Komuro et al. [23] with the description of 64x64 chip implementation by Komuro and Ishikawa [24] and also the SCAMP family with the most recent design of SCAMP-5 architecture by Carey et al. [7]. The SCAMP-5 is described in more detail in section 3.5 as this project assumes its general architecture as representative for the FPSP devices and bases final implementation on his instruction set.

2.2 Convolutional Neural Networks

The origins of the neural networks start with the idea of perceptron by Rosenblatt [39], a machine learning computation algorithm that combines the input values with a set of weights, producing one or more outputs. It can be expressed as a graph, as shown in Figure 2.1 where every connection is associated with its weight and intermediate nodes, called neurons, accumulate

incoming connections and apply simple thresholding, producing an output of 0 or 1. Naturally,

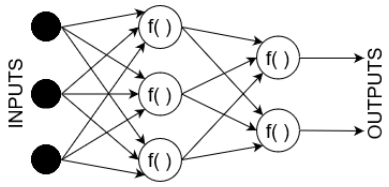


Figure 2.1: Diagram of a Simple Perceptron.

the perceptron has evolved, with more neurons and layers added, more sophisticated activation functions used, resulting in the general architecture of a fully connected neural network, in which every neuron at layer N has a connection, and hence weight parameter, to every neuron in layer $N + 1$.

However, as each connection relates to a single weight coefficient, the fully connected networks have an enormous number of parameters. That's why the convolutional neural network has arisenar, especially in the applications related to image processing, where the input is a large array with spatial relationships, making the fully connected networks extremely inefficient and non-optimal. Instead, in convolutional neural networks, the weights are organised in small spatial kernels, or filters, that are then slid over the entire image and the output of the feature is computed at every location. This process is visualised in Figure 2.2. This minimises the required

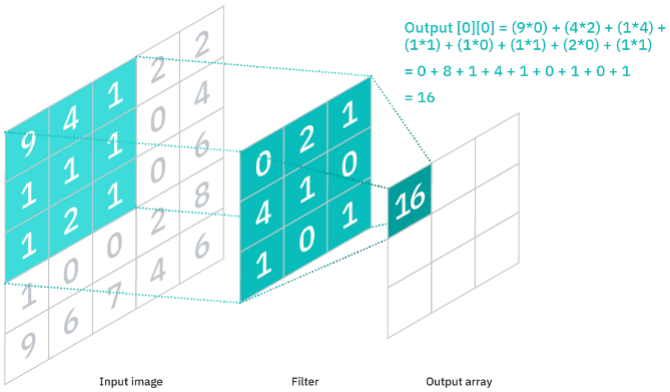


Figure 2.2: Computation diagram of convolutional filter. Source: `https://www.ibm.com/cloud/learn/convolutional-neural-networks`

weight parameters, as they are essentially shared for every input pixel and computation can utilise the spatial properties of the image. In addition, we can apply multiple kernels on the layer's input to compute a set of different features. The behaviour at the image boundaries is defined by the padding parameter, and the distance in pixels by which the kernel is moved over the image is set by the stride parameter. Naturally, the size of the kernels can vary and the values of its weights are adjusted in the process of network learning through backpropagation.

2.2.1 Fundamental Architectures

To achieve a network of a certain behaviour architectures combine multiple convolutional layers of configuration, usually interconnected by pooling layers, which works similarly to convolution layers, but their kernels instead apply a reduction function to the input, commonly MAX or MEAN. In classification tasks, the last layers are usually fully connected to integrate computed features and output the probability for prediction classes.

The LeNet 5 developed by Lecun et al. [27] and presented in Figure 2.3 is one of the earliest convolutional networks and was created to recognise simple handwritten digits images, being named as MNIST dataset [28], and contains 60k training- and 10k test-samples.

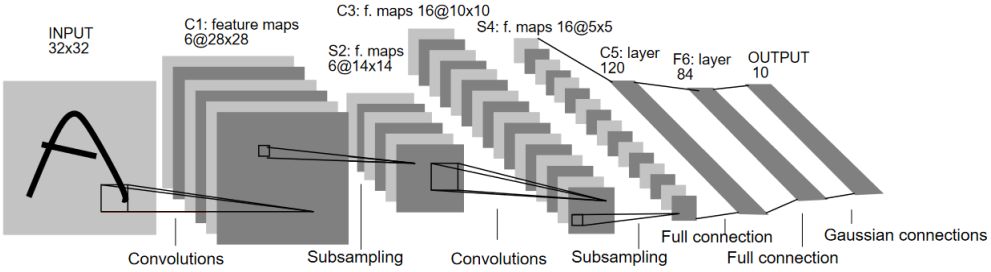


Figure 2.3: Architecture of LeNet 5. Source [27].

Over time more complex architectures has been developed to empower network for solving more complex tasks, such as classification of CIFAR-10 [25] dataset which contain small, colour images of ten different objects or the ImageNet1000 [11] that contains images of 1000 different classes and has become the most popular benchmark for convolutional neural networks in classification. The most notable architectures are AlexNet [26], VGG [41], ResNet [17] and GoogleNet [43].

2.2.2 Layer Normalisation

The normalisation of the network layers is a widely used component of the convolutional neural networks, that in the majority of situations substantially improves and stabilizes their behaviour both in training and evaluation. Essentially, normalisation standardizes the network's activations inputs by attempting to bring their mean of the to zero and the variance to one. To achieve that, in the common implementations such as in Keras, the normalisation layer shifts the activations by the mean and scales them by the squared root of variance. In addition, two coefficients γ and β are also usually learned by default during training, to empower the process even more, and hence the normalisation can be expressed as

$$\hat{x} = \gamma \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{2.1}$$

where ϵ is just a small constant added for numerical stability.

The way the mean and variance are computed and used varies with the normalisation scheme. The four most common normalisation schemes are, Batch- [19], Layer- [3], Instance- [46] and Group Normalisation [48]. How each of those computes the mean and variance is visualised in Figure 2.4.

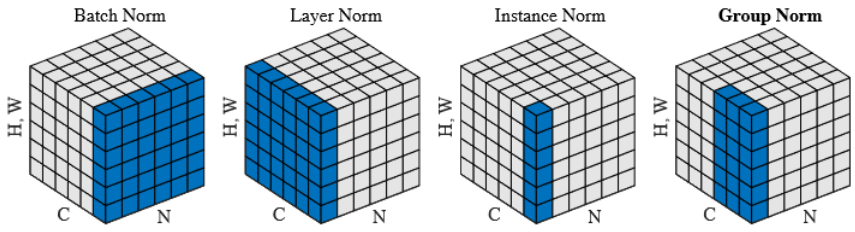


Figure 2.4: Comparison of normalisation methods. Blue boxes show how the values are grouped to calculate the mean and variance. H, W are the stacked spatial dimensions, C is the channel dimension, N is the number of images in the batch. Source [48].

As Batch-Normalisation computes the mean and variance over the entire batch, N dimension in Figure 2.4, it behaves differently in training and test. While in training, it calculates the moving mean and moving variance of every channel from the training batches and fixes them in test, requiring only to execute equation 2.1 computation on a per-channel basis. In other normalisations, the respective mean and variance depend only on the data of a current sample, resulting in more complex execution during the test runs.

2.3 Cellular Automata

The Cellular Automata is a discrete computational model that operates on a regular grid of cells, of which each can be at the time in one of the finite number of states. The dynamics, change of each cell's state at the next iteration, are determined by a set of rules defined over a neighbourhood, with the update being simultaneous for all cells. It has been proposed by Stanislaw Ulam and John von Neumann while working on self-replicating systems [35]. Over the years, it has been a field of intensive research, going from simple one-dimensional automata, also referred to as elementary, to even three-dimensional examples, as well as the specialised automata such as *BIO-LGCA* [12] that is used to model collective cell migration in computational biology.

The simplest and most known example of a two-dimensional cellular automaton is Conway's Game of Life developed by John Horton Conway [13]. The cells are allowed to be in either the "ALIVE" or "DEAD" state, and the next iteration is determined with only four rules, which are based on counting the number of "ALIVE" in a neighbourhood and thresholding it. Despite the rules being simple, the automaton exhibits complex behaviour, with some shapes being permanent, some oscillating, and some moving regularly through the grid.

The Game of Life uses the concept of a square neighbourhood, also referred to as Moore, while another possibility is a Von Neumann neighbourhood that takes only the cells in the horizontal and vertical direction. They both can be of various radius r which simply defines the number of cells included in each direction, in Figure 2.5 both has been presented in $r =, 1$ version. Naturally,

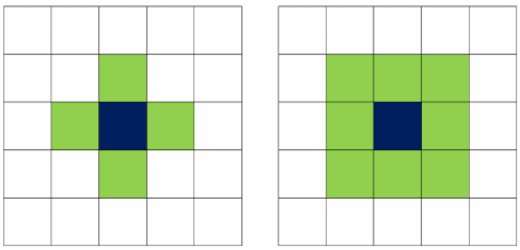


Figure 2.5: Cellular automata neighbourhoods, Von Neumann (left) and Moore (right), both of radius $r = 1$.

the neighbourhood can have a larger radius or even a random shape depending on the application.

A large part of the mathematical research on cellular automata concentrates on searching the rules space, which for binary automaton and the given neighbourhood is limited as there is a limited number of possible patterns, and analysing their behaviour and characteristics, such as stability, periodicity. A famous example is a so-called rule 110, which was proved to be universal, meaning it can simulate an arbitrary Turing Machine. In more complex examples, when the state has multiple elements the rule itself might be expressed as a mathematical function, like in *BIO-LGCA* [12] or possibly even calculated with neural networks, as they can be interpreted as function approximators. The work done on applying neural networks to the cellular automata is discussed in section 3.1.

Chapter 3

Related Work

This chapter presents and explains the work and publications related to the project topic and used throughout it. It first shows the beginning of the use of neural networks in cellular automata and then existing Neural Cellular Automata architectures, which are a base for the performed analysis and created extensions in the context of FPSP devices. It continues by presenting the state-of-the-art FPSP architecture, SCAMP5, introducing its capabilities and existing applications, providing a picture of the benefits, as well as the drawbacks of such devices. Further, the description of the development tools essential to the project is given, an FPSP architectures simulator and the convolutional kernels compiler, followed by the presentation of the relevant work in the area of neural networks quantisation, needed for the implementation of NCA on the focal plane.

3.1 Early Applications of Neural Networks with Cellular Automata

The ability of the Neural Networks to express the automaton rules has been shown by Wulff and Hertz [49] who applied them to find the rules of a one-dimensional automaton that has generated a specific history (pattern) and in another experiment where it was to simulate that pattern along.

The similarity of the cellular automata and convolutional neural networks has been investigated even further by Gilpin [14] who has trained a CNN to represent the dynamics of Conway's Game of Life and proposed a general architecture of a convolutional neural network for expressing the neural cellular automata, shown in Figure 3.1. Presented architecture is adjusted for the binary cellular automata, such as Conway's Game of Life, but it can be conveniently extended to allow multi-channel state vectors and approximate more complex dynamics, by adjusting the number of layers, hidden channels and output's size.

However, the important convention is that only the first layer of the network has kernel dimensions different than one, as it expresses the step of obtaining the neighbourhood information. Following 1x1 layers are then responsible for applying the rules and deducing the next state's value. This architecture, as well as the training process, has been expanded to a more complex and non-standard automaton by Mordvintsev et al. [32], as shown in the next section.

3.2 Growing Neural Cellular Automata

The general architecture proposed by Gilpin [14], has been applied by Mordvintsev et al. [32] in the design of "Neural" Cellular Automata, as embedded with a neural network, that can

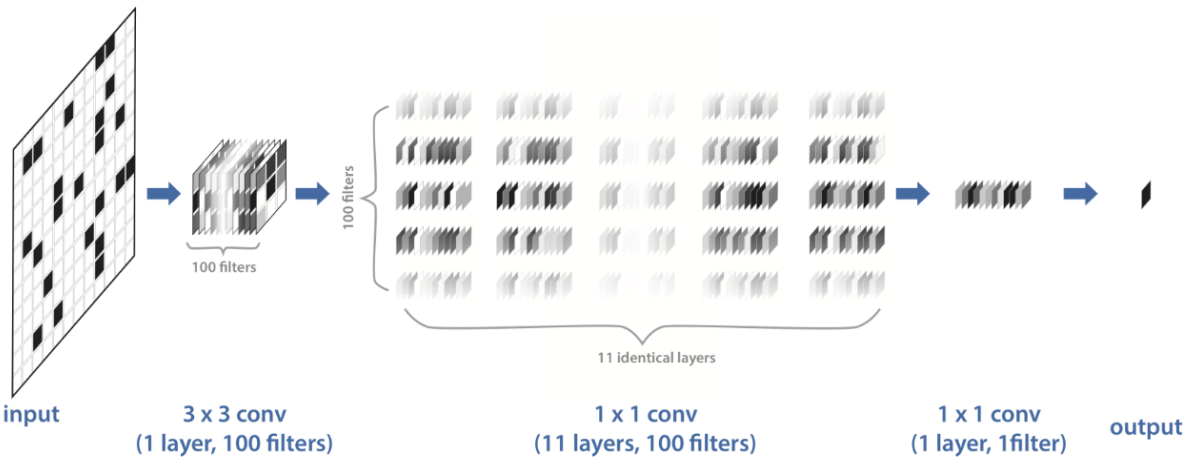


Figure 3.1: The general architecture of a convolutional neural network for learning cellular automata proposed by Gilpin [14]. Source Gilpin [14].

grow from a single seed a stable and regenerative pattern. The motivation for this work was the morphogenesis - process of an organism's shape development and self-organisation. Biological systems and organisms can grow and regenerate complex organs through the process of inter-cell communication and applying certain, hardly definable rules. The developed neural cellular automata aim to model that process in some sense and achieve that self-organisation property on the example of small patterns.

The automaton grid is a 64x64 image with each pixel being an individual cell in the automaton, and the state of each pixel is a 16 element vector. However, the convention of cells being "ALIVE" or "DEAD" is the same as in Conway's Game of Life and determined by the value of the so-called α channel, where a cell is "ALIVE" if its $\alpha > 0.1$ or it has a neighbour with $\alpha > 0.1$. This alteration is necessary to allow life to grow/spread out, as we start from a single seed, and the neighbour is determined according to the Moore neighbourhood with radius $r = 1$, effectively a 3x3 square block. Also, three of the channels are used to represent the RGB colour of the pixel and the task of the automaton is for each cell to match the colour of the corresponding pixel in the pattern that is to be formed.

The architecture of the designed network is formed by one 3x3 convolutional layer followed by two fully connected layers, or 1x1 convolutional layers. The 3x3 convolution is nicknamed as "perceive" step, which relates to collecting the data about the neighbourhood and uses three hardcoded kernels, vertical and horizontal Sobel kernel as well as the identity kernel that has all entries set to zero, except the middle. Choice of fixed Sobel kernels that effectively compute horizontal and vertical gradients is motivated by the biological analogy to chemical gradients. Then, fully connected layers compute the result of the update rule and output an update vector *ds*.

Those steps are effectively performed by every cell in the image grid and each one has its update value *ds*. Having computed the update vector for each cell, to obtain the next iteration it has to be added to the current state array. However, to model the lack of global synchronisation in living cells, the addition process is stochastic as the value of the update vector is completely zeroed-out before the addition for random 50% of all cells. Finally, all cells are now checked for being "ALIVE" or "DEAD" and as the "DEAD" cells are essentially meant as non-existent and should not carry any information, their state is explicitly set to zero before producing the state array for the next iteration.

This update process and the entire Growing NCA architecture are visualised in Figure 3.2.

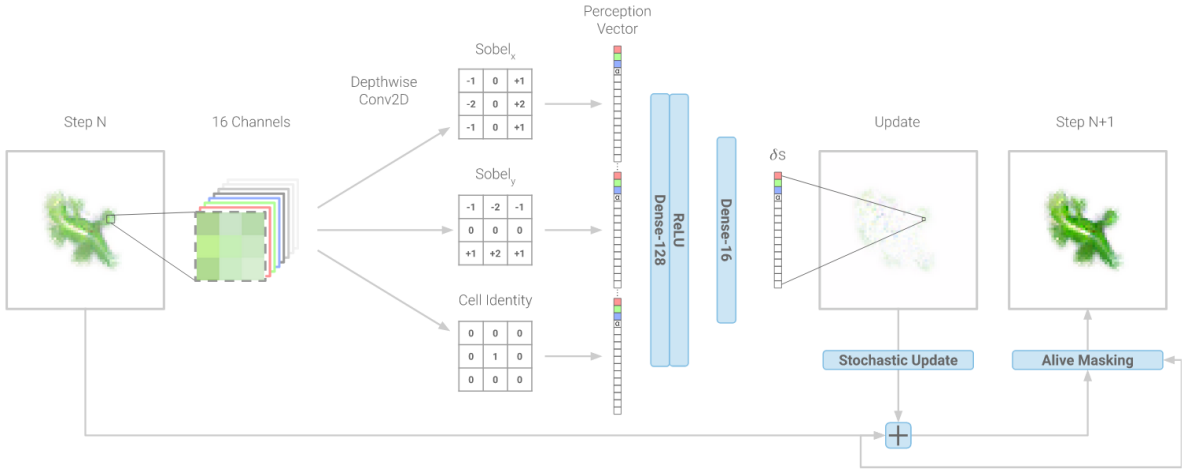


Figure 3.2: The architecture of the Growing NCA by Mordvintsev et al. [32] with the illustration of the step update procedure. Source [32].

In addition to the architecture design, Mordvintsev et al. created a comprehensive training routine that ensures the patterns are not only able to grow to the desired form, but also persist and even regenerate the damage done to grown patterns. Initially, during training, the automaton was allowed to execute forward for n steps before the loss was calculated, but that resulted in the patterns exploding when the NCA reached the number of iterations greater than n, as the network did not learn what to do with an already completed image. To overcome that, an idea of "sample pool" is introduced, where first a pool of 1024 seed samples is created and then for every training iteration 32 arrays are sampled to form a training batch. In a single training step, the automaton evolves for a certain amount of iterations producing an output batch, which is then placed back into the sample pool. Effectively throughout the process, NCA can learn how to proceed not only from the seed samples but also from the inputs that are partially or even completely ready. To ensure that NCA remembers how to start from seed, every training iteration one sample was replaced with a seed.

The neural cellular automata created by Mordvintsev et al. [32] with the development of appropriate training method forms a foundation for the development of next applications of NCA, including Self-Classifying MNIST digits NCA. Resulting NCA can be also classified as a recurrent model, as the value of state array at the next iteration depends directly on its current value, and even the authors, Mordvintsev et al., argue that it could be potentially named "Recurrent Residual Convolutional Network with per-pixel Dropout", where the "per-pixel Dropout" refers to the stochastic addition procedure.

3.3 Self-Classifying MNIST Digits NCA

The Self-Classifying MNIST Digits automata by Randazzo et al. [36] is an example of how the neural cellular automata could be used as a new approach to the classic classification problem. This work shows that the pixels that form an image of a digit are able, only through local communication with its neighbours, to agree on the digit they form.

The architecture of the neural cellular automaton extends the one used in Growing NCA [32] by using the learnable 3x3 convolutional layer instead of fixed Sobel kernels. The state vector is formed of 20 channels in total, where the first channel is immutable to the automaton and

represents the pixel's intensity. Out of the remaining mutable 19 channels, the last 10 are used to represent prediction labels for corresponding digits, with the rest not having assigned meaning and used to pass the information between the cells. The breakdown of the cell state vector is shown in Figure 3.3.

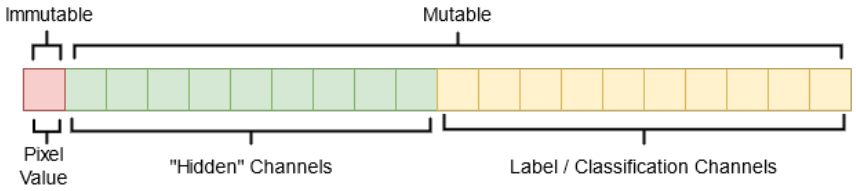


Figure 3.3: Breakdown of the cell state vector in the Self-Classifying MNIST Digits Automata by Randazzo et al. [36].

Similarly to Growing NCA [32], a cell can be either "ALIVE" or "DEAD", which is simply determined by thresholding on the pixel intensity so that only the cells that form a digit participate in the computation. Except that, the process of calculating the step update and producing the next iteration state array is analogous to the Growing NCA, including the stochastic update and explicit zeroing-out of "DEAD" cells.

The exact architecture of the neural network used to compute the step update is shown in Table 3.1. The ReLU function is not used after the last layer, as the network needs to be able to

Table 3.1: Structure and number of parameters of the neural network that computes the update step in Self-Classifying MNIST Digits NCA [36]. H and W denotes the size of the input image, the third parameter is the number of the layer's output channels.

Layer	Kernel	Input Size	Output Size	Number of parameters
2D Convolution + ReLU	3x3	H x W x 20	H x W x 80	14480
Dense + ReLU	1x1	H x W x 80	H x W x 80	6480
Dense	1x1	H x W x 80	H x W x 19	1539
				Total: 22499

produce both positive and negative updates. The total count of the network parameters is approx. 22.5k which is considered a very small model within the standards of deep learning.

The training process of the automaton uses the "sample pool" method from Growing NCA, to ensure that the trained model is regenerative and can cope with the situation when input digit changes and all the predictions are immediately entirely wrong and needs correction. In addition, usage of the L2 loss has resulted in the stabilisation of the magnitude of individual states, enabling long-term operation without the risk of saturation. Hence the model could be used in the real-time classification of the input video, as it is robust and able to reclassify smoothly as the input image changes. Figure 3.4 presents the performance of the NCA created by Randazzo et al., showing also a comparison of the results achieved with different loss functions. We will refer to this exact architecture in the latter part of the report as MNIST NCA. It can be seen from the figure that the model achieves an accuracy as high as 96% and the total agreement, which shows how often the cells reach a consensus about the prediction, of approx. 83%. The sudden spike at the 200th step represents the moment when the input digit changes and the total recovery of performance after that point proves the robustness of the automaton.

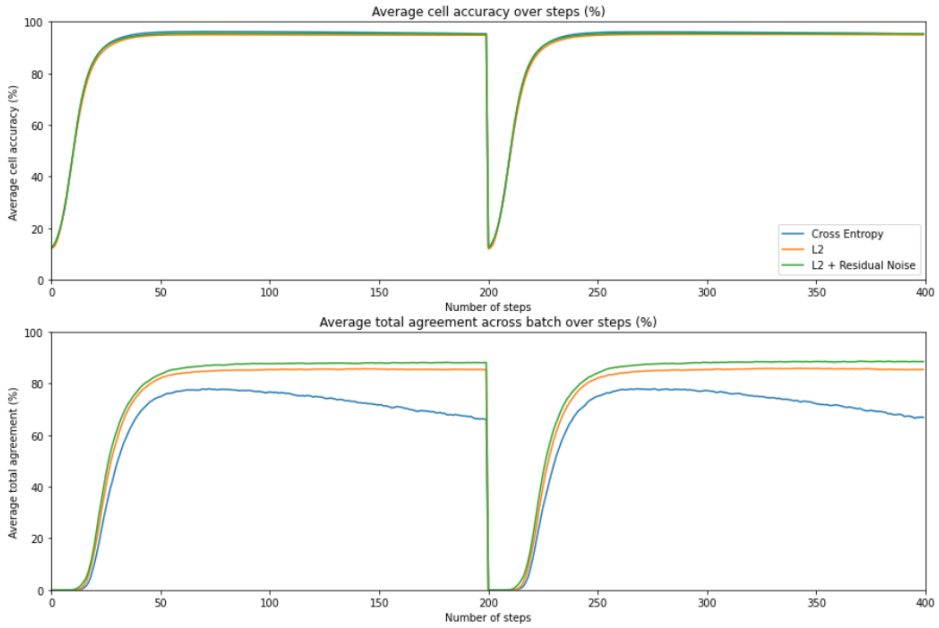


Figure 3.4: Accuracy and total agreement achieved by the Neural Cellura Automata model for MNIST digits self-recognition (MNIST NCA) with different loss functions. Green line present the metrics of the final model. Source [36].

3.4 Segmentation with NCA

The application of the NCA to the segmentation of the image has been shown by Sandler et al. [40], where it has performed stable and accurate segmentation of images into an object, boundary, and background. Authors emphasize the similarity of the NCA with the Recurrent (Residual) Neural Networks, as the production of the next iteration can be expressed as

$$S_{i+1} = S_i + UpdateCNN(S_i) \tag{3.1}$$

so next state iteration depends effectively on all previous ones. The term "residual" comes from direct use of the previous state and adding it to the network's output.

Consequently, they expand the architecture of the network used in Growing NCA [32] or MNIST NCA [36] with the so-called reset component. It is inspired by the GRU and LSTM cells, widely used in recurrent networks, and it gives each cell an additional mechanism to control its growth in magnitude. Proposed architecture is shown in Figure 3.5. The reset gate simply

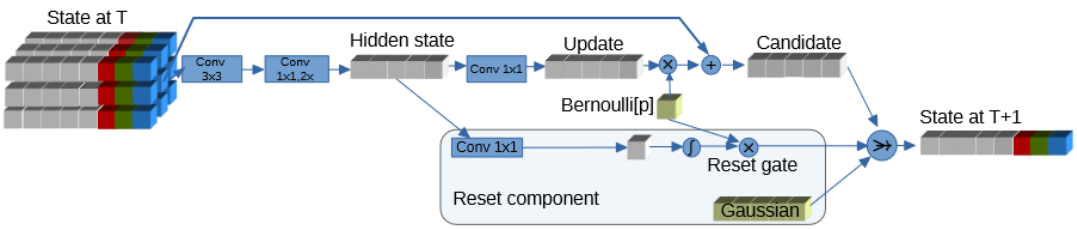


Figure 3.5: Architecture of the Neural Cellular Automata with reset component proposed by Sandler et al. [40]. Source [40].

controls the mixture of computed candidate state with the random Gaussian noise that is to form the next iteration state. The GRU and LSTM cells, that have the reset mechanism, form

the state-of-the-art recurrent architectures, and the design of neural cellular automata that incorporates it is likely to achieve similar success. Results obtained by Sandler et al. show that it indeed stabilises the state growth and enables the NCA to maintain accurate segmentation permanently. Authors have also experimented with a normalisation of the state array, using Batch-, Instance- and Layer-Normalisation and observed that each has stabilised the learning and inference process. The final model uses the state vector of 48 elements, with a possible reduction to 32 that results only in temperate degradation of performance.

In this work, we will investigate the impact of the normalisations on the modified architecture of MNIST NCA in section 4.2, analyzing how the internal dynamics change. However, we do not experiment with the addition of a reset gate.

3.5 SCAMP5

The SCAMP5 is the FPSP architecture developed and implemented by Carey et al. [7] as a vision sensor of 256x256 pixels, providing full SIMD parallelism, as every pixel is equipped with a simple processing element (PE). The structure of an individual processing element is shown in Figure 3.6.

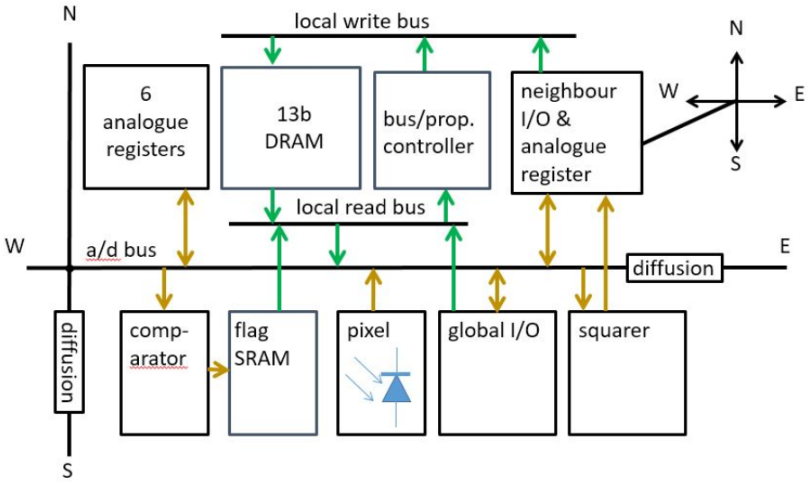


Figure 3.6: Single processing element in SCAMP5. Source: `https://personalpages.manchester.ac.uk/staff/p.dudek/scamp/default.htm#Architecture`.

Each PE can obtain directly the data from its North/South/West/East neighbours and is capable of performing analogue arithmetic and logical operations on the data in the respective registers. The comparator and a flag register allow for creating a device-wide mask enabling conditional execution of some instructions but also grouping a set of pixels into what can be called super-pixel, which has more memory capacity. The fact that the chip uses an analogue domain saves the need for an arithmetic unit, however, it causes computations to acquire noise as well as the values in analogue registers to degrade over time. The operations supported by the SCAMP5 are summation, inversion, division by a power of two, squaring and low-pass filtering, but no multiplication. Hence, the implementation of the convolutional kernels on the device requires converting multiplications into additions.

3.5.1 Existing Applications

There have been a large number of publications reporting the applications the exploit the advantages of SCAMP5 device, starting with an edge detection at 100k FPS [7], which thanks to

very low-latency also enables tracking of moving objects, by its edges. Further work by Chen et al. [8] has presented the implementation of features detection, while Bose et al. [4], as well as Murai et al. [34], have shown the realisation of the visual odometry, with the last one reporting the operation at 300FPS.

It has been proved that the SCAMP5 device can accommodate a CNN that classifies MNIST dataset digits with high accuracy. One such example is the AnalogNet by Wong et al. [47], which performs the computation on the analogue registers and achieves testing accuracy of **96.9%**. It uses a very simple architecture with three convolutional filters that are computed on the focal plane, in which activations are then thresholded and the events are then pooled using a custom-designed binning algorithm to feed the fully connected network accommodated on the micro-controller, as per Figure 3.7. The proposed training approach accounts for the

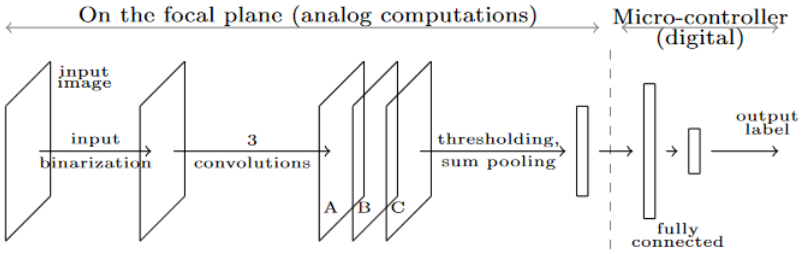


Figure 3.7: Architecture diagram of AnalogNet. The sum pooling process flattens the convolution's feature maps and creates a flat vector, largely reducing the amount of transferred data to the micro-controller. Source: [47].

noisy computations by first training the entire network with lossless, digital computations after which the convolutional layers are fixed and converted to the device code. Then noise-inclusive retraining is performed, and only the fully connected layer is learning so that it can compensate for the noisiness of real-life calculations.

A different approach has been exemplified in a work by Bose et al. [5], which focused on implementing the convolutions using the digital registers restricting the network's weights to ternary representation, with possible values of 1,0 or -1. To maintain accurate digital representation and storage of the image it groups the SCAMP5 pixels into 4x4 superpixels, where each store a single bit of a *16-bit* representation of image intensity. Naturally, it reduces the image resolution from 256x256 to 64x64 but on the contrary allows for accurate, lossless digital convolutions. To complete this representation efficient algorithms for addition and subtraction of such superpixels are provided, which can be carried out using XOR, AND and bit shifting operations only. The convolutional filters are then applied over this reduced in size image, with the results stored in analogue registers which are then immediately pooled and the resulting low-resolution image is transferred to the microcontroller which performs the fully-connected layer computation, producing the final prediction. The final accuracy of **95%** is similar to AnalogNet, but the maximum achieved frame-rate of 210 FPS, compares poorly to the 2260 FPS reported by AnalogNet. However, this work illustrates that the limited per-pixel resources might be overcome by grouping physical pixels, at the cost of increased latency.

Finally, a different publication by Bose et al. [6] presents the convolutional network that is entirely embedded on the focal plane, with no interaction from the microcontroller. The specificity of this approach is that the kernel weights are no longer incorporated into the device code, but instead stored in the digital registers. Because SCAMP5 is a SIMD device, the implementation of weights as a code enforces all pixels to calculate the same kernel. However, with the weights stored in registers, different pixels will be able to compute distinct kernels even while executing

the same commands. As shown in the visualisation of the approach in Figure 3.8, the input

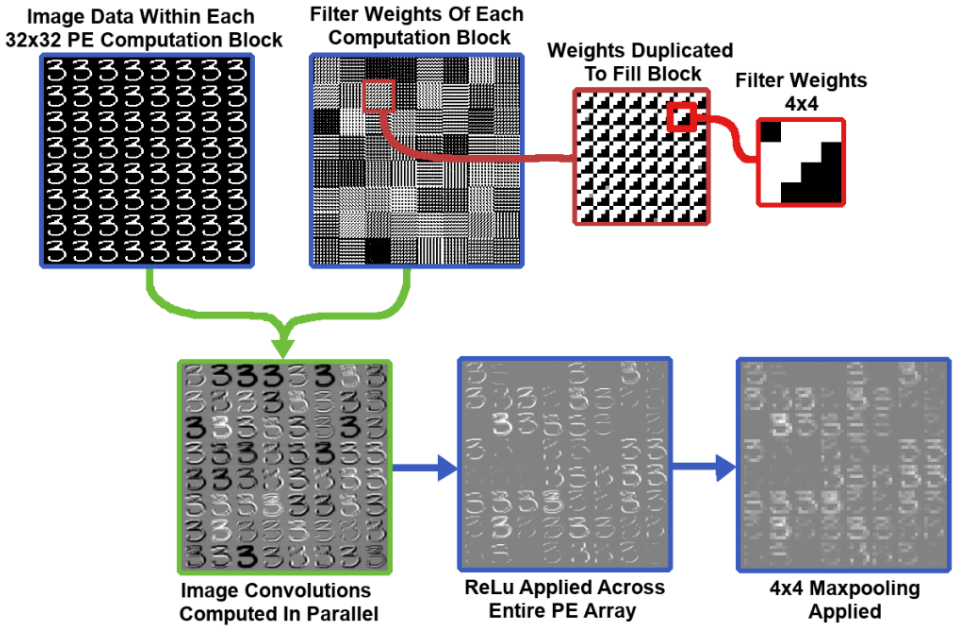


Figure 3.8: Illustration of the process of parallel computation of a convolutional layer with 64 kernels of size 4x4 on the 256x256 SCAMP5 device, with replicated 32x32 MNIST images. Source: [6].

image is converted to 32x32 representation and replicated over the entire pixel array, where the corresponding layer weights are already stored. With weights values limited to binary form, few convolutional layers can be stored on the chip at the same time, allowing for inference of deeper networks. A single-pixel stores only one weight of a specific 4x4 kernel and it needs to transfer the remaining ones from its neighbours, but that is ensured in the inference execution with each pixel executing the same code. The final fully connected layer is also executed directly on the focal plane, allowing for the processing speed of 3672 FPS while achieving the accuracy within **92-94%** in the case of a two-layer network. It is noticeably faster than 2260 FPS for the AnalogNet, at the cost of the precision drop of **3-5%**.

This work emphasises that the maximum usage of the SCAMP5's parallelism can lead to a large improvement in the execution speed of the CNN, however, limitations to low-precision weights and input image size might be too strict in more complex applications.

All of the presented implementations were targeting MNIST classification tasks from the classical perspective, with a network being asked to produce a probability vector for each classes base on the entire image. However, the NCA approach requires each pixel to decide what digit it forms only based on the information from its direct neighbours, which forms a more complex problem that needs more resources. The implications for the SCAMP5, and the FPSP devices in general, that arises from the NCA are discussed in chapter 4.1.

3.6 Simulator

The Focal Plane Sensor Processor Simulator developed by Mulla [33] plays an essential role in the validation and evaluation of produced implementations. Since the intention is to explore the most beneficial development directions, a parameterisable and adjustable simulator enables testing for far more complex applications than the existing ones. The simulator models all the

fundamental components such as buses, memory, registers and provides a convenient interface to implement the actual architecture from those building blocks. The current version provides mostly implemented SCAMP5 architecture and instructions, with the ability to measure parameters such as device execution time and processed frames per second when executing the submitted program. Consequently, in the implementation phase, the SCAMP5-like architecture is targeted. The alternative device configurations, for example, with additional registers, could be evaluated and compared using the expected power consumption and chip size metrics. Furthermore, a graphical user interface is supplied that allows convenient and real-time observation of selected registers' values across the entire device.

A more specific description of the utilised simulator features together with produced results and metrics is given alongside the implementation in Section 4.

3.7 CAIN

The effectiveness of the computation of convolutional kernels on the FPSP devices depends heavily on the quality of generated device code. As the most sophisticated FPSP architecture up to date, the SCAMP5 presented in Section 3.5 above, does not provide a multiplication instruction, kernels cannot be implemented right away. However, the SCAMP5 supply instead other powerful instructions, such as three-operand additions and multi-step shifts, so the multiplications can be effectively avoided. Consequently, a generation of the device code from the convolutional kernels requires a solution that can explore possible combinations, utilise common sub-expressions in given kernels and produce the most optimised device code. The Cain code generator by Stow et al. [42] is such an algorithm. It splits the desired kernels computation into a set of simplified operations called Atoms, effectively unrolling the multiplications into additions. This process recurses until only the identity kernels, with only one non-zero entry, are left in that set. Then the graph traversal algorithm is executed on the resulting computation graph to find the configuration that minimises the defined cost function. The Cain is highly configurable and provides a selection of both graph traversal algorithms as well as cost functions that can customise the search process, with the detailed description available in its codebase [42].

The implementation of the NCA requires transforming the entire neural network computation, with multiple convolutional kernels applied at every pixel, onto the focal plane. Consequently, Cain is essential for obtaining the best device code that maximises similarities between the kernels and the final execution plan produced by Cain is likely to be more efficient than if the multiplications were available.

The use of Cain in the NCA implementation is described in Section 4. Moreover, the project's codebase contains the SCAMP5 program of the implemented networks as well as the Python scripts that were used to invoke Cain and automate the code generation.

3.8 Neural Network Quantisation

3.8.1 TensorFlow Lite Framework

The TensorFlow Lite is a framework within the Tensorflow [2] that empowers the execution of deep learning models on devices with limited capabilities and resources, such as mobile or embedded devices. It supports multiple platforms and emphasizes optimising five key constraints amongst which the most relevant are latency, size, and power consumption [45]. One thing that enables that is a special file format represented with *.tflite* extension and known as FlatBuffers [1]. Consequently, the models have to be created with the TFLite API or by the conversion

from the standard TensorFlow Keras models, and running the inference requires invoking the interpreter.

The TFLite supports three optimisation mechanisms, namely pruning clustering, and quantisation, with only the latter being investigated in this work. From the available techniques, there are two that give a full integer network, the *"post-training"* or *"training-aware"* approach. The first one is simply applied to the already trained network, while the other introduces wrapping layers to the network during training so that it knows the effects of the quantisation and can adjust itself, which should reduce the involved loss of accuracy.

The conversion of all the parameters and operations to the sole integer format is possible when the representative dataset is provided. It should contain an exemplary set of the network inputs, so the min and max values of the input and layer's activations could be deduced. The network's input is further referred to also as activation for simplicity, as the quantisation scheme treats them equally.

This is necessary as the quantisation scheme, developed by Jacob et al. [20], uses an affine mapping of the form:

$$f = S(q - Z) \tag{3.2}$$

to relate integer values q with the floating-point ones f, where S and Z are constants/quantisation parameters.

The *"S"* is a floating-point number that represents the quantisation scale or step, which is the distance between two floating-point numbers represented by two consecutive integers and effectively denotes the loss of the quantisation. The scale *"S"* for the N-bit quantisation is simply determined by

$$S = \frac{max(x) - min(x)}{2^N - 1} \tag{3.3}$$

where x is a set containing all observed floating-point values in the array under quantisation. The *"Z"* is an integer value and stands for a zero-point, so it is simply the integer value that corresponds to the floating-point value of 0.

The quantisation scheme holds a separate set of those parameters for each activation array, while the layer weights are by default quantised channel-wise, resulting in separate quantisation parameters for every output channel of the layer.

The calculation of a single channel of the layer's output can be represented as a multiplication of two vectors, with the first one F_1 of size $1 \times N$ containing relevant input values and the second F_2 of size $N \times 1$ with the weights of that channel stacked respectively. Hence, the calculation of the floating-point output channel f_3 can be summarised as

$$f_3 = F_1 F_2 = \sum_{i=1}^{N} r_1^i \cdot r_2^i \tag{3.4}$$

However, after applying the quantisation scheme from the Equation 3.2 to both sides, this becomes

$$S_3(q_3 - Z_3) = \sum_{i=1}^{N} S_1(q_1^i - Z_1) \cdot S_2(q_2^i - Z_2) \tag{3.5}$$

and can be rewritten as

$$q_3 = Z_3 + M \sum_{i=1}^{N} (q_1^i - Z_1) \cdot (q_2^i - Z_2) \tag{3.6}$$

$$where\ M := \frac{S_1 \cdot S_2}{S_3} \tag{3.7}$$

giving a formula for the calculation of a single, integer-valued output channel of a layer. However, with the weights quantised for each output channel independently, the M coefficient for each output channel is different and that has to be respected in computation.

Consequently, when calculating the output of each layer of the quantised, fully integer model, care has to be taken about shifting corresponding elements by their zero points *"Z"* and finally rescaling the output by corresponding *"M"* as it is defined in the Equation 3.7. The *"M"* constant is the only non-integer element in the Equation 3.6, however, it can be pre-computed for every network's layer. Moreover, the Jacob et al. [20] argues that it always lies in the interval (0,1) and hence can be expressed in "normalised" form as

$$M = 2^{-n} \cdot M_0 \tag{3.8}$$

where M_0 is an integer. As a result, *"M"* can be expressed as integer multiplication by M_0 followed by the bit-shifting operation corresponds to the 2^{-n} part. It means that every computation in the model is converted to the efficient integer operation or a bit-shift, and so the network is fully integer quantised.

The described in detail operations are handled by the TFLite interpreter and not exposed to the end-user. However, running the TFLite quantised model on the FPSP devices, in this work emulated with simulator by [33], requires manual implementation of the above equations. The use of the described TFLite schemes in the NCA quantisation, together with the obtained results and required customisations, is provided later in Section 5.1.

3.8.2 Alternative quantisations

The main drawback of the TFLite quantisation is its lack of flexibility which comes as a cost of having an end-to-end solution with a simple API. However, the research area of deep network quantisation has many publications that attempt to implement various quantisation mechanisms to reduce the network's latency and memory requirements.

Binary and Trinary Neural Networks

The most fundamental concept is the Binarised Neural Networks, where the weights are restricted to 1 and -1. The BinaryConnect by Courbariaux et al. [9] uses two methods for the binarisation of networks weights. The first one simply takes the sign of the weight to determine the binarised value

$$w_b = \begin{cases} +1, & \text{if } w \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{3.9}$$

and the alternative one using a stochastic approach

$$w_b = \begin{cases} +1, & \text{with probability } p = \sigma(w) \\ -1, & \text{with probability } 1 - p \end{cases} \tag{3.10}$$

where σ is the hard sigmoid function. This approach is extended in BinaryNet [10], while the work by Rastegari et al. [37] shows by solving the optimisation problem, that the *sign-based* binarisation is the most optimal approximation. It also introduces additional scaling parameter α and determines its optimal value α^*

$$I * W \approx (I \oplus B)\alpha \tag{3.11}$$

$$\alpha^* = \frac{1}{n} \|W\|_{l1} \tag{3.12}$$

where I is the real-value input, W real-value weights with B being its binarised representation and $\oplus$ is a convolution without multiplication.

However, the most notable parts of the above work are on methods of training the binarised networks, implementing binary BatchNormalisation, and fine-tuned optimisation algorithms, such as a shift-based version of AdaMax[21]. The obtained results show the accuracy of 98.5% achieved by the fully connected model on the MNIST classification [28] and 89% by the convolutional model [10] on the CIFAR-10 classification [25]. Both Courbariaux et al. [10] and Rastegari et al. [37] introduces also an interesting concept of XNOR-Networks where also the activations are binarised.

Another very simple variant of the quantised network is the Ternary Weights Network, explored by Li et al. [29] and Lin et al. [30], which uses three values, 0, 1 and -1, to represent the weights. Although *2-bits* are required now for each weight, the existence of a third possible value largely extends the number of feasible kernel combinations, to $3^{N\times N}$ for the $N \times N$ kernel, compared with $2^{N\times N}$ in binary networks. The accuracy achieved on the MNIST recognition task [28] reaches 99.35% [29].

N-bit Quantised Neural Network - QNN

A natural extension of the Binary and Trinary Networks are the quantisations that allow an arbitrary number of bits to represent the original floating-point weights. The Quantised Neural Network by Hubara et al. [18] extends the methods used in Binarised Neural Network[10] to allow efficient training and evaluation while having the weights, gradients and activations quantised to N-bit representation.

It uses two quantisation schemes proposed by Miyashita et al. [31], linear quantisation

$$\mathbf{LinearQ}(x, N) = \text{clip}\Big(\text{round}\Big(\frac{|x|}{step}\Big) \cdot step, minVal, maxVal\Big) \tag{3.13}$$

$$step := \frac{maxVal - minVal}{2^N - 1} \tag{3.14}$$

and logarithmic quantisation

$$\mathbf{LogQ}(x, N) = \begin{cases} 0, & x = 0, \\ 2^Y & \text{otherwise}, \end{cases} \tag{3.15}$$

with

$$Y := AP2(x) = \text{clip}(\text{round}(\log_2(|x|)), maxExponent - 2^N, maxExponent) \tag{3.16}$$

$$\text{clip}(x, min, max) = \begin{cases} 0 & x \leq min, \\ max - 1 & x \geq max, \\ x & \text{otherwise}. \end{cases} \tag{3.17}$$

with the equations assuming x being non-negative. If x can be negative, the equations still apply, but the sign bit has to be stored requiring the use of *N+1* bits. The key feature of those schemes is they map a given floating-point value to a specific *4-bit* number. In the case of *LinearQ* the quantisation levels are spread by a constant distance, while in the *LogQ* the quantisation levels are certain powers of 2. Here, the function *AP2(x)* essentially finds the approximately closest exponent of two for a given x and the *maxExponent* is the maximum power of two that is allowed.

Depending on the distribution and magnitude of the values to be quantised one of the schemes might be better than the other, and the logarithmic scheme can be further tuned by changing

the logarithm's base. This is illustrated in Figure 3.9, taken from [31] which shows the example distribution of activations of a convolutional layer, here specifically in the case of VGG16 architecture [41]. In the log-based quantisation, the quantisation levels for small magnitudes

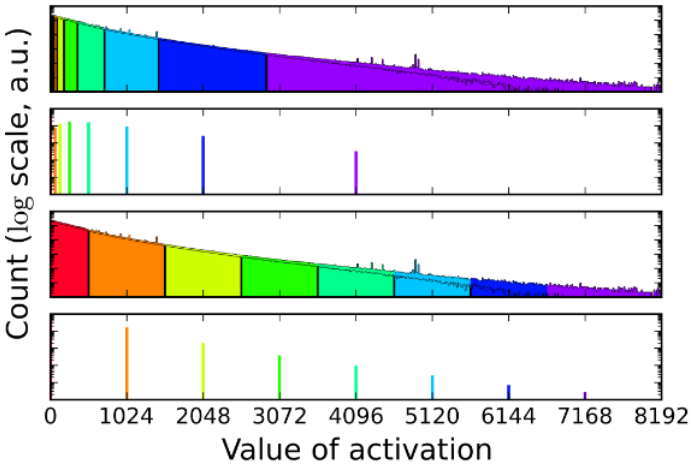


Figure 3.9: Illustration of linear and logarithmic quantisation on the exemplary activation layer. The top plot shows the histogram of activations, with the regions coloured accordingly to the bins in logarithmic quantisation on the plot below. The next pair shows the same relation, but for the linear quantisation scheme.

will be close to each other allowing for small errors at the cost of larger degradation for large magnitudes, which might benefit the activation distributions close to normal. Linear quantisation could be superior in the case of normal distribution. However, different weights might have different importance and the direct conclusion cannot be taken.

In addition to the above quantisation, Miyashita et al. [31] presents methods to compute the convolutions entirely in the log-domain and even an algorithm to perform training in that representation. Reported results show the AlexNet [26] and VGG16 architectures quantized with both methods achieving performance comparable with their full-resolution versions when using 5-bits per value[31].

Meanwhile, the QNN[18] targets to improve the results of the quantised network on the ImageNet 1000 classification task [11], which is a common classification benchmark. The *6-bit* QNN version of the GoogleNet architecture [43] achieves the Top-1 accuracy of 66.4% and Top-5 of 83.1%, which compared to the original (non-quantised) accuracies of 71.6% and 91.2% respectively, shows a great potential of this method.

N-bit Incremental Network Quantisation - INQ

The Incremental Network Quantisation (INQ) developed by Zhou et al. [50] concentrates on achieving the maximal accuracy of the quantised model while using only standard components of convolutional networks. It does not target a specific network architecture, but rather provides a method that can be applied to an arbitrary network. The innovation in this approach is that it takes the pre-trained model and quantises the network in steps, using the following Algorithm 1. Consequently, the quantisation process can consider the importance of different weights and also adjust to the performed quantisation by altering the non-quantised weights. Furthermore, a regulariser is added to the training routine, to push the values of network weights towards the quantisation levels $\mathbf{P}_l$, defined below, so that the training is "aware" of ongoing quantisation and preferred weight values.

Algorithm 1 Incremental Network Quantisation

Having full-resolution weights matrix W
Define two sets:
  A_q that contains the quantised weights
Initialise A_q as : $A_q \leftarrow \emptyset$
Define coefficient $a \in [0, 1]$
Define step value b, which is the increment to a at every step
Initialise a to initial partition : $a \leftarrow b$
while $a < 1$ **do**
  Quantise portion a of weights and add them to set A_q ▷ Note that weights quantised in the previous iteration remains in this set
  Retrain network with regularisation, keeping the weights $w \in A_q$ fixed
  Increment a in this step : $a \leftarrow a + b$
end while
At this point every weight is quantised and $a = 1$

The goal of the quantisation scheme is to convert the full-resolution weights W_l into the quantisation levels P_l, which is defined as follows

$$\mathbf{P}_l = \{\pm 2^{n_1}, \ldots, \pm 2^{n_2}, 0\}, \tag{3.18}$$

where n_1 and n_2 are two integers such that $n_2 \leq n_1$ and effectively bounds the non-zero weights to be in range $[-2^{n_1}, -2^{n_2}]$ or $[2^{n_2}, 2^{n_1}]$.

They are determined with the following formulas

$$n_1 = \text{floor}(\log_2(4s/3)), \tag{3.19}$$
$$s = \max(\text{abs}(\mathbf{W}_l)), \tag{3.20}$$
$$n_2 = n_1 + 1 - \frac{2^{(N-1)}}{2}, \tag{3.21}$$

where N is the number of bits used.

Among N *bits* used to represent a quantised number, one is always used to represent the value of zero. Then the remaining *N-1 bits* are used to represent other elements of $\mathbf{P}_l$ and there is no idea of the sign bit. Consequently, N-bit quantisation is able to represent $2^{N-1} + 1$ possible quantisation levels.

Finally, the actual mapping of full-resolution values from $\mathbf{W}_l$ is performed using a so-called ladder of powers algorithm [50], which here is omitted for clarity. The overall algorithm is performed layer-wisely as there might be significant differences in the magnitudes of weights and hence separate determination of $\mathbf{P}_l$ per layer takes that into account.

The second essential part of the INQ [50] is the method of weight partitioning. The authors have investigated two approaches, random masking and the "largest magnitude first" method. For the implementation with the random mask, it is important to ensure that the mask created in Nth step includes all entries that were in a mask at step $N - 1$, as the network quantisation is incremental. The inspiration for the "largest magnitude first" approach comes from the work on network pruning [16, 15], and considers weights with the larger absolute value being more important than the smaller ones. Hence, the point is to first quantise those forming a low-precision base and allowing the remaining ones to compensate for the approximation error.

The experimental comparison of the above weight partition strategies is then carried out with the ResNet-18 network, resulting in the magnitude-based approach achieving a performance better by around 1%, as shown in Figure 3.10, taken from the INQ [50].

Strategy	Bit-width	Top-1 error	Top-5 error
Random partition	5	32.11%	11.73%
Pruning-inspired partition	5	**31.02%**	**10.90%**

Figure 3.10: Comparison of the random and magnitude-based (pruning-inspired) strategies for weight partition in INQ on the ResNet-18 on the ImageNet 1000 task.

Finally, the visual representation of the is given in Figure 3.11. The INQ authors [50] bench-

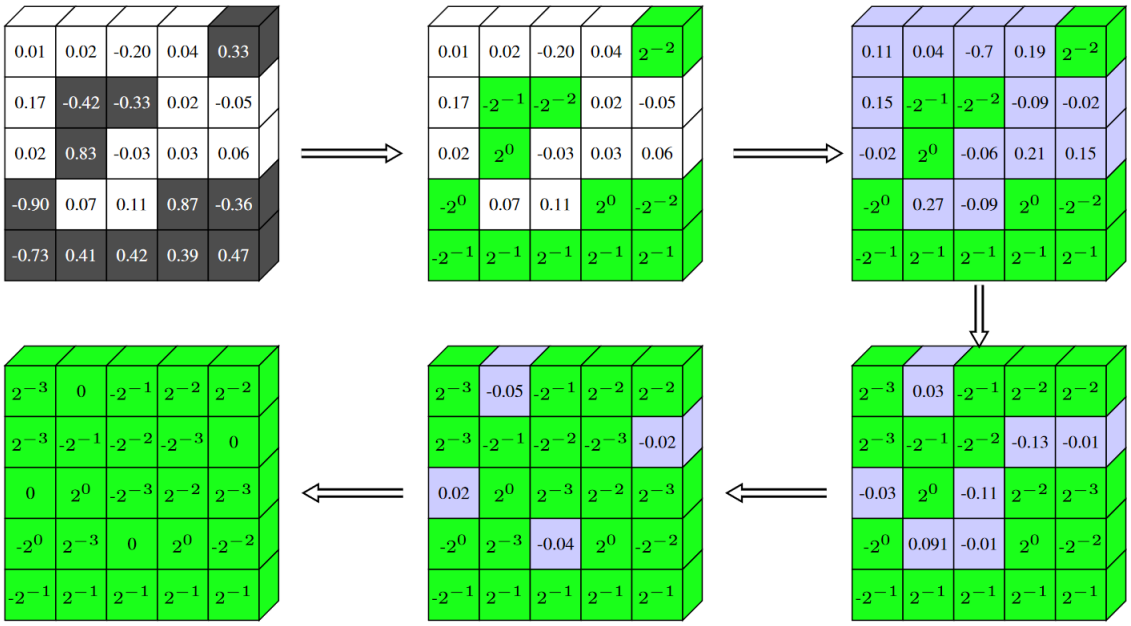


Figure 3.11: Illustration of the Incremental Network Quantisation Process. The first row shows the 1st iteration step by step. The top left array presents the weight partitioning with the magnitude-based strategy, with the grey block marking selected locations. The next box illustrates the quantisation of those chosen weights, and the top right finally, the retrained weights, marked with light grey. The second row simply illustrates results of further quantisation increments, with the whole presented process starting with 50%, to 75%, then 87.5% and finally 100%.

marked their method on the popular CNN architectures, such as AlexNet or ResNet-18, on the ImageNet 1000 task with the *5-bit* quantisation precision and compared obtained results with the performance of full-resolution, reference counterparts. The acquired comparison scores are shown in Figure 3.12 taken from [50].

As presented, the INQ quantisation has achieved a relatively small performance degradation with each of those architectures, using only 5-bits precision. Consequently, it has been prioritised for the alternative quantisation of the NCA and its use as well as adopted modifications are described in Section 5.2.

Network	Bit-width	Top-1 error	Top-5 error	Decrease in top-1/top-5 error
AlexNet ref	32	42.76%	19.77%	
AlexNet	5	**42.61%**	**19.54%**	0.15%/0.23%
VGG-16 ref	32	31.46%	11.35%	
VGG-16	5	**29.18%**	**9.70%**	2.28%/1.65%
GoogleNet ref	32	31.11%	10.97%	
GoogleNet	5	**30.98%**	**10.72%**	0.13%/0.25%
ResNet-18 ref	32	31.73%	11.31%	
ResNet-18	5	**31.02%**	**10.90%**	0.71%/0.41%
ResNet-50 ref	32	26.78%	8.76%	
ResNet-50	5	**25.19%**	**7.55%**	1.59%/1.21%

Figure 3.12: Performance comparison between the INQ quantised 5-bit models and their full-resolution counterparts on the ImageNet 1000 task.

Chapter 4

Neural Cellular Automata on FPSP Devices

The general architecture of the FPSP fits directly into the definition of the cellular automata as it enables individual pixels to access data of their immediate neighbours, hence emphasising the concept of local communication. Nevertheless, the implementation of the neural cellular automata on the FPSP devices faces several challenges related to the limited resources and simplicity of the individual processing units.

The first section of this chapter discusses the memory requirements of the existing NCA and compares them to the capabilities of the SCAMP5 architecture. It describes the efforts of modifying the MNIST NCA [36] to reduce necessary memory and shows discovered architectures. Section 4.2 provides an extensive analysis on the problem of continuous stability of NCA, diving into the example of the new architecture presented in section 4.1. A detailed comparison with the MNIST NCA [36] is given to explain the differences in internal dynamics between the two models, followed by the search for a viable solution, including training regime changes as well as normalisation.

4.1 Computational and storage requirements of neural cellular automata

4.1.1 Computational complexity of the NCA

The NCA is a highly distributed model of computation, as each cell is its entity and at every step needs to compute the update vector, and so execute by himself the forward pass of the CNN on its neighbourhood. Consequently, the bottleneck of the implementation is determined by the processing ability of a single pixel and the length and complexity of the code that it has to execute.

This points to the first limitation, the FPSP devices are highly parallel and efficient utilisation of that can lead to low-latency computation, as exemplified by Bose et al. [6] in Fully Embedded CNNs. The straightforward implementation of the NCA on FPSP use all pixels in the array, but the issue is that the costly computation of the update vector is effectively replicated and executes on every pixel/"thread".

Therefore, in the development of the NCA, the architectures with a fewer number of the total parameters are preferable as it directly relates to the number of necessary multiplications and summations, so smaller networks will give shorter device code. For example, in the MNIST NCA

architecture, Table 3.1, 23.5k parameters would transfer to the same amount of multiplications alone. However, reducing the number of parameters in the architecture might come at the cost of decreased performance.

In addition, on the device without a multiplication instruction, such as SCAMP5, each such operation is transferred to multiple addition code by the CAIN compiler, which differs in length depending on the kernel's complexity and magnitude. We present the length of the obtained device code for the implemented architectures later in the section 4.

One solution to this problem would be increasing the computational power of each processing element, although this will increase the overall power consumption and most likely be stopped by the space limitations. On the other hand, some degree of parallelism could be brought back by computation in the superpixel configuration, treating each superpixel as an automaton cell. This would reduce the cell array size, and effectively the resolution of the processed image, but that parallelism is vital for a low latency operation. Spreading a single automaton cell computation onto few pixels is difficult because of the SIMD nature of the SCAMP5, as each pixel executes the same instruction. Consequently, the weights would have to be stored in the registers, similarly to Fully Embedded CNNs by Bose et al., so that the same instruction set could produce results of different kernels. In this setup, the computation would also benefit from the existence of shared storage for a superpixel to minimize the number of data access and move instructions, for example, a per superpixel/patch RAM storage for computation results. The amount of networks parameters/weights is enormously large and storage of individual copies, even per superpixel is impossible, however, as they are the same for each cell could be stored in global, read-only memory pixels could read them as the calculation progresses.

4.1.2 Storage limitations of the existing FPSP architectures

The presented neural automata models of Growing NCA and MNIST NCA are very small architectures in the standards of deep learning, with the latter having only 22.5k parameters. However, in terms of the FPSP architectures, this number and related network size enormously exceeds the current capabilities of state-of-the-art devices and beats in complexity all already implemented architectures.

The first issue is the state vector associated with every cell in the automata, which in the case of Growing NCA has 16 elements and 20 in MNIST NCA, with each value being a floating-point number. In contrast, each pixel in the SCAMP5 device has 7 analogue and 13 binary registers available. In addition, the analogue registers are not suitable to store the values for longer periods due to the noise and leakage, and analogue arithmetics introduces computational errors. It has been observed empirically, that the accuracy of the state affects especially the stability of the prediction and the agreement between the cell. As shown before, NCA is essentially a recursive, residual neural network, and the next iteration is formed by incrementing the current state with the computed update. Hence, the accuracy of calculation and storage is also very important because an eventual error will propagate and increase with iterations. It means that for the accurate operation of the NCA, all the data has to be stored in the digital registers.

The problem of insufficient storage per pixel is even more severe if we consider the computation of the update by a neural network. By the nature of cellular automata, every pixel computes the update by itself, requiring enough storage to accommodate necessary intermediate results in layers computation. By looking at the architecture of the MNIST NCA network in Table 3.1, we can see that for a one-cell the first layer produces an intermediate vector of 80 elements. The entirety of this vector has to be stored, until the computation of the second layer is completed, as every output channel of the second layer uses each of 80 elements of the intermediate vector to compute its value. Now we also require 80 storage units to fit the computed output of the

second layer. Consequently, we end up requiring 160 storage units to compute just those two layers. Fortunately, in the computation of the third layer we can safely overwrite the registers used by the first layer's output, as with the second layer calculated, they are no longer needed.

However, as the update vector produced by the network is used to increment the current state, it does not have to be stored entirely, but its elements can be just added to corresponding channels of the state vector. Hence, the last layer of the network does not bring any additional space requirements. Finally, except the storage registers, some are required to facilitate the individual computations, especially when there is no multiplication operation available, such as in the case of SCAMP5. Through our work, the number of registers required by the CAIN code generator to implement convolutions was not greater than 8.

In summary, if the MNIST NCA were to execute on the entire SCAMP5 array device it would require at least **188** registers per pixel. Even though in this work we are not focusing on implementation on existing devices, but rather establishing directions for development, this number enormously exceeds existing capabilities and in the next section, we will look at the ways of reducing it.

4.1.3 Reduction of NCA memory footprint

In general, the number of storage units required to facilitate the computation of a neural network can be determined by looking at the number of output channels of two consecutive layers and finding the largest sum. In the above case of MNIST NCA, both the first and second layers have **80** output channels, which gives rise to **160** required registers. With that knowledge, we have searched for an alternative configuration, based on the MNIST NCA, that will decrease that number.

Reducing the size of the state vector have a minor effect on the required storage, however, it decreases the total number of computations that have to be done at the first layer. A convolutional layer with C_{in} input channels, kernel of size $H \times W$ and C_{out} output channels have:

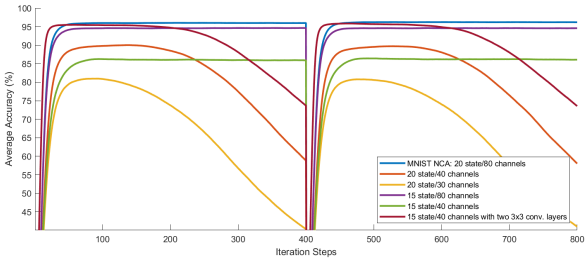
$$C_{in} \times H \times W \times C_{out} \tag{4.1}$$

parameters and perform the same number of multiplications. Having a smaller state vector, limits the complexity of the computation of the first and last layer, as the number of the channels in the update vector produced by the network must match the number of mutable channels in the state vector. In our research, we have tried the configurations with the **15** elements vector, which effectively reduces the number of mutable channels to **14**, and as the number of the classification channels is fixed at **10**, the resulting vector has only **4** "hidden" channels.

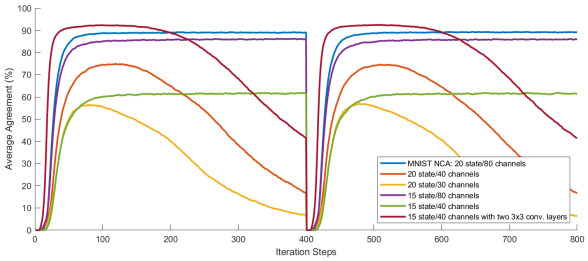
Nevertheless, the main issue lies in the number of hidden channels of the network. Hence, the variations of MNIST NCA architecture from Table 3.1 were tested, with the number of output channels of intermediate layers changed from base **80** to **40** or **30**, as it would reduce the number of indispensable storage units for network computation to **80** and **60** respectively. In Figure 4.1 are presented the performance results for the selected alternative combinations that were tested.

SMALL NCA

As presented, reducing the state vector size to **15** had only a minor effect on the accuracy, and slightly more notable on the agreement, however that does not solve the storage issue. From the configurations with less than 80 channels, the **15** state vector/**40** channel is acceptable, as the resulting accuracy is still approximately 86% although the agreement has dropped notably to the level of 60%. The agreement metric measures in how many cases all "ALIVE" cells have achieved consensus, however in a real setting the significance of that will be diminished as the



(a) Average Accuracy (%) - y-axis value starts at 40%.



(b) Average Agreement (%) - y-axis value starts at 0%.

Figure 4.1: Performance results for the selected configurations that were tested. First, five configurations in legend have the same architecture as MNIST NCA from Table 3.1, but the different sizes of the state vector, denoted by *X state* and output channels in intermediate layers, *Y channels*. Last configuration listed has one additional 3x3 convolutional layer and is presented in Table 4.2.

presence of noise and the fact that the image might be moving. In that situation, it is expected that some of the pixels will show incorrect prediction, but the achieved accuracy shows that the vast majority of labels will be correct. The exact architecture of this configuration is shown in Table 4.1 and we will refer to it further in the report as SMALL NCA, as both size of its state vector and number of intermediate channels is smaller than the base/MNIST NCA. Another

Table 4.1: Structure and number of parameters of the version of MNIST NCA with only 15 state elements and 40 intermediate channels, referred to as SMALL NCA. H and W denotes the input image dimensions, the third number in output size is the number of output channels.

Layer	Kernel	Input Size	Output Size	Number of parameters	Activation
2D Convolution	3x3	H x W x 15	H x W x 40	5400	ReLU
Dense	1x1	H x W x 40	H x W x 40	1600	ReLU
Dense	1x1	H x W x 40	H x W x 14	560	None
			Total	7560	

benefit of the SMALL NCA architecture is that it is three times smaller than the MNIST NCA, which means that it requires three times fewer multiplications alone to compute the output. This directly translates to the fewer instructions in the device code that are necessary to implement the network, the significance of this feature was discussed in section 4.1.1, which discusses the topic of computational complexity. Consequently, the degradation of performance is compensated by the simplicity of the model and potential for the significantly faster execution. Finally, the SMALL NCA architecture has preserved the long-term stability exhibited by the MNIST NCA, which means that it does not require any additional components for stabilisation.

For the reasons presented in this subsection, the SMALL NCA model has been a primary model in the work on quantisation and final implementation using the SCAMP5 simulator, as it can operate continuously without intervention and is most suitable in terms of used storage and computational complexity.

DEEP NCA

In Figure 4.1, we have also presented an alternative architecture, which mitigates the problem of required storage by adding one more layer. It contains one additional 3x3 Convolutional Layers,

as detailed in Table 4.2 and uses **40** channels for the intermediate layers together with **14** elements state. The goal of this architecture was to maintain the complexity of the MNIST NCA network,

Table 4.2: Structure and number of parameters of the deeper version of NCA architecture - DEEP NCA. H and W denotes the input image dimensions, the third number in output size is the number of output channels.

Layer	Kernel	Input Size	Output Size	Number of parameters	Activation
2D Convolution	3x3	H x W x 20	H x W x 40	7200	ReLU
2D Convolution	3x3	H x W x 40	H x W x 40	14400	ReLU
Dense	1x1	H x W x 40	H x W x 40	1600	ReLU
Dense	1x1	H x W x 40	H x W x 19	760	None
			Total	23960	

keeping the number of parameters approximately the same, but reduce its requirements for the temporary storage units. The maximum sum of the number of output channels for two consecutive layers is just **80**, as the results of the third layer calculations can safely overwrite units used before by the first layer. For the reason of having one more layer than other tested configurations, we will refer to it further as the DEEP NCA to distinguish it from other configurations.

An interesting feature of this architecture is that it effectively uses a neighbourhood block of size 5x5, radius $r = 2$ Moore neighbourhood, as two stacked 3x3 convolutions are equivalent to a 5x5 convolution. The use of a larger neighbourhood results in DEEP NCA achieving its peak performance faster than other configurations, around the 20th step in contrary to the 40th step for the others.

However, the DEEP NCA is one of the configurations that were presented in Figure 4.1 and which has failed to maintain the continuous stability property. Every network has been trained with exactly the same routine, L2 loss with residual noise and the same amount of iterations, but did not learn how to stabilise prediction over time. Interestingly, the configurations such as **20** state/**40** channels that just change the parameters of the MNIST NCA architecture also experiences the same sudden fall of performance. We look into this problem in detail in the section 4.2, analysing the internal dynamics and looking for a viable solution to the problem. We have chosen to perform this research on the basis of the DEEP NCA model, which we also use later in the quantisation and for the implementation with the SCAMP5 simulator.

Summary

The analysis presented in this section shows that to accommodate the neural cellular automata, the FPSP devices would have to necessarily extend the amount of memory available to each cell in the grid. Compared to the existing architecture of SCAMP5, having only 6 general-purpose analogues and 13 binary registers, the required increase seems enormous and possibly restricted by the chip area. However, the solution could be achieved not only by scaling up but also by implementing more sophisticated multiplexing possibilities allowing for efficient computation in the superpixel configuration, where registers of every pixel contribute to the storage pool of a superpixel or maybe an addition of shared RAM for pixel patches.

At the same time, the network to be implemented could also be modified to reduce the required memory footprint. Developing the neural cellular automata with architectures that are deeper, but uses fewer intermediate channels will reduce the number of necessary storage units.

Consequently, two models that are used in the further work are SMALL NCA, as it exhibits simplicity while achieving acceptable performance and DEEP NCA, as it is an example of stacking additional layers to reduce required memory. Both of those are used in the quantisation and at the end implemented on the SCAMP5 simulator. We provide the summary of the storage requirements for those two models in Table 4.3, with the MNIST NCA presented for comparison.

Table 4.3: Summary of the number of storage units required per pixel/automaton cell by the MNIST, SMALL and DEEP NCA architectures.

NCA Configuration	State Vector	Storage of intermediate results	Required by CAIN compiler	Total
MNIST NCA	20	160	8	188
SMALL NCA	15	80	8	103
DEEP NCA	20	80	8	108

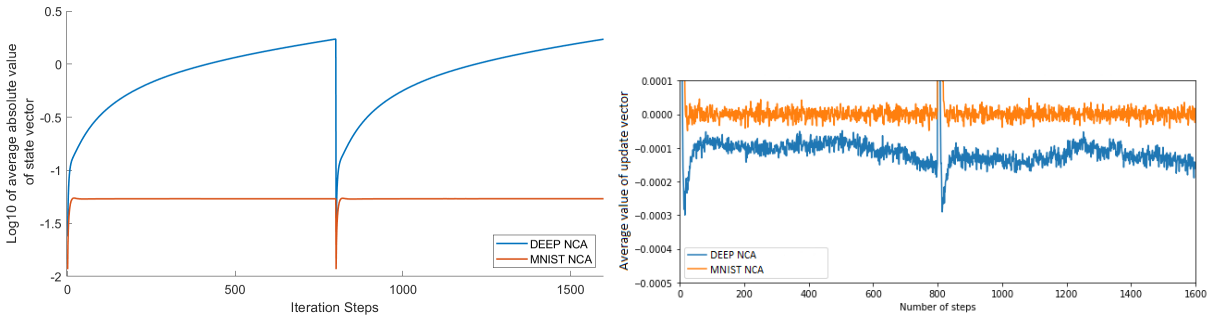
4.2 Long-term Stability

It has been shown in Figure 4.1 that few of the researched configurations exhibit a sudden drop in both the accuracy and agreement after exceeding a certain number of iterations. Although each of those models was trained with the same training configuration as the base MNIST NCA or the SMALL NCA, they have not maintained the continuous stability property.

Interestingly, just changing the number of channels in the intermediate layers in MNIST NCA from **80** to **40**, while keeping the state vector size at **20**, has resulted in a model without stability, but if the state vector is also reduced to **15**, the stability is back. Therefore, the question arises for the reason for such a change in the dynamics.

The authors of MNIST NCA Randazzo et al. [36], have achieved stability by using the L2 loss with noise. It has the effect of pushing the update vector to very small values when the correct prediction is made and input does not change. Consequently, the magnitude of the state is stabilized and the performance is not affected with time. However, for some reason, it does not help in all configurations, even very similar ones.

We examine this problem basing on the DEEP NCA architecture and start by looking at the average magnitude of the state vector and compare it with the MNIST NCA, presented in Figure 4.2a The behaviour of MNIST NCA agrees with results reported by Randazzo et al. and stabilizes



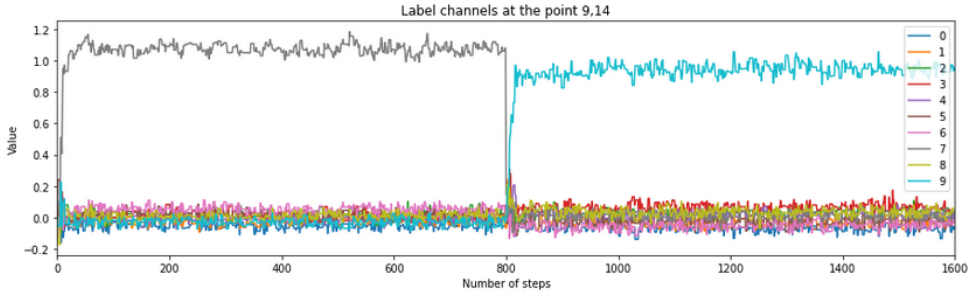
(a) The average absolute value of the state vector

(b) The average value of the update vector

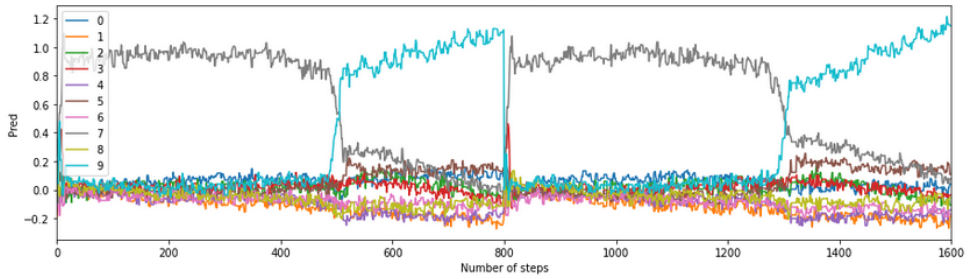
Figure 4.2: Comparison of the internal dynamics of the DEEP NCA (blue) and MNIST NCA (orange) for 800 steps.

the average absolute value of its state vector and preserves it. At the same time for the DEEP NCA model, we observe growth that continues over time. The maximum value it reaches on the 800th step is significantly larger than the limit hit by the MNIST NCA, note that the scale is logarithmic. This difference in the dynamics of the models suggests that the reason for a DEEP NCA, and other architectures, to start losing performance is the lack of the ability to limit the average absolute value of the state. It is caused by the average value of update vector in DEEP NCA not being zero, but rather permanently shifted by some constant, as presented in Figure 4.2b. It is the crucial difference between the dynamics of those models. The updates in the MNIST NCA averages to zero over time so that the magnitude of the state is kept at a constant level. On the other hand, in the DEEP NCA, this average is approximately constant around -0.0001, which explains approximately linear growth in the average state value. Consecutive updates accumulate more and more in the state vector, resulting in the performance downfall after a certain threshold is reached.

To observe the impact of the unconstrained growth of the state vector magnitude, we show in Figure 4.3 what happens in the label channels for both MNIST and DEEP NCA. This shows



(a) MNIST NCA - y-axis shows label prediction



(b) DEEP NCA - y-axis shows label prediction

Figure 4.3: Dynamics of the label channels predictions for an exemplary sample at a selected pixel. Each prediction is associated with separate colour according to the legend.

clearly that the MNIST NCA keeps all incorrect predictions at a very low value with a distinct maximum in the correct channel. The DEEP NCA though, is more unstable with some incorrect channels decreasing slightly and a presence of a large turnaround at the 500th step and once again at 1300th after the mutation. The correct label channel suddenly receives a large number of negative updates while one of the incorrect labels gets the confidence quickly, and the same repeats after the mutation. In the evaluations for longer periods of time, we have found out that the label to which the DEEP NCA steers is then usually constant, acting as an attractor and continuously growing in magnitude.

The above analysis shows that the cause for the unstable dynamics lies in the average value of state update being relatively far from zero, causing the elements of the state vector to grow continuously in magnitude and reaching an unspecified threshold after which the performance breaks. We investigate two methods for solving this issue, starting with the modifications to the

training regime presented in section 4.2.1 and following with the application of the normalization reported in section 4.2.2.

4.2.1 Training Regime Modification

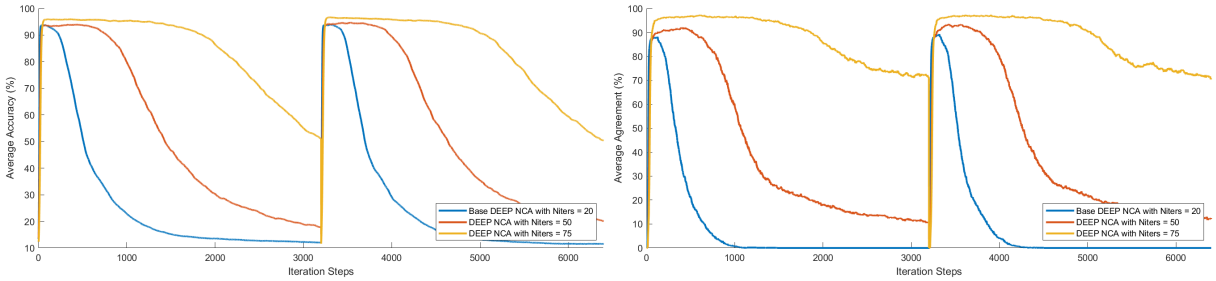
To bring the alternative model to the same behaviour as the original one, several experiments with the training regime were conducted. First, as the alternative model has one more layer and hence its optimal training parameters might be different, the length of the training, as well as the learning rate, has been varied. However, none of the tested configurations gave an improvement in the long-term performance. The parameter that modification has resulted in an improvement of DEEP NCA behaviour is the *NIters* variable, which determines for how many steps a mini-batch is processed before the loss is calculated and resulting images are placed back in the sample pool. The sample pool technique is the same one as in the Growing NCA from section 3.2 and detailed as an Algorithm 2.

Algorithm 2 Training Routine

Initialise a Sample Pool for training from 160 randomly selected samples from training set : Γ
for $step = 1, 2, \ldots, MaxSteps$ **do**
  Copy 16 randomly selected samples from the Sample Pool : $X \xleftarrow{16} \Gamma$
  Replace 1/4 of the selected samples with random, newly initialised samples (with all the state values set to zero) : $X \leftarrow X \vee X_{new}$
  Mutate another 1/4 of the selected samples (change only the underlying image, but not the state values) : $X \leftarrow X \wedge X_{mutated}$
  for $n = 1, 2, \ldots, NIters$ **do**
    Compute the forward pass of the neural cellular automata : $X \leftarrow NCA(X)$
  end for
  Calculate the prediction loss : $\sigma \leftarrow L(X, labels)$
  Compute and apply the gradients : $W \leftarrow \delta(W, \sigma)$
  Commit just computed output X back to the Sample Pool : $\Gamma \leftarrow X$
end for

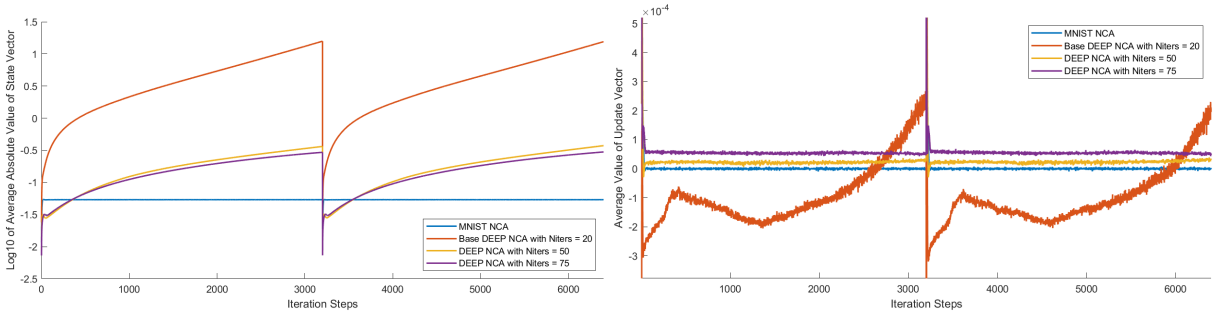
Consequently, the value of *NIters* affects the loss calculation and for how long the cellular automata on the samples in the sample pool has been iterated over. Learning in bigger "chunks", with larger *NIters*, should enable the NCA to adapt for long-term processing and how to cope with the already correct predictions and maintain them, hopefully enough to learn the continuous stability property. The value of *NIters* used by the original MNIST NCA is **20** and so we have tried the configurations with value increased to *50*, **75** and *90*. The last one has resulted in unsuccessful training as the period after which the loss is measured was too large, however, the performance results for the values of **50** and **75** are shown in Figure 4.4. Increasing *NIters* both to **50** and **75** have improved the stability margin substantially, but it only extended it to further point in time. The general behaviour of the DEEP NCA in both cases is still the same and the model has not learnt how to stabilise its state magnitude as visualised in Figure 4.5.

The average absolute value of the state vector is still increasing, although, the severity of this phenomenon is reduced. Models trained with larger *Niters* can reduce and stabilise the average value of the updates, but they are still not zero. They can control the increasing state magnitude for more iterations, but the issue of continuous instability is not solved. Consequently, it is not the training routine being not able to push the model towards stable behaviour, but it might be the model's architecture itself that blocks that from happening. We investigate the effect of expanding that architecture with normalisation layers in the next section.



(a) Average Accuracy (%) (b) Average Agreement (%)

Figure 4.4: Performance results achieved by the DEEP NCA with modified *Niters* training parameter, together with the results of DEEP NCA obtained with the original configuration of $Niters = 20$



(a) Log10 of the average absolute value of state vector (b) Average value of the update vector

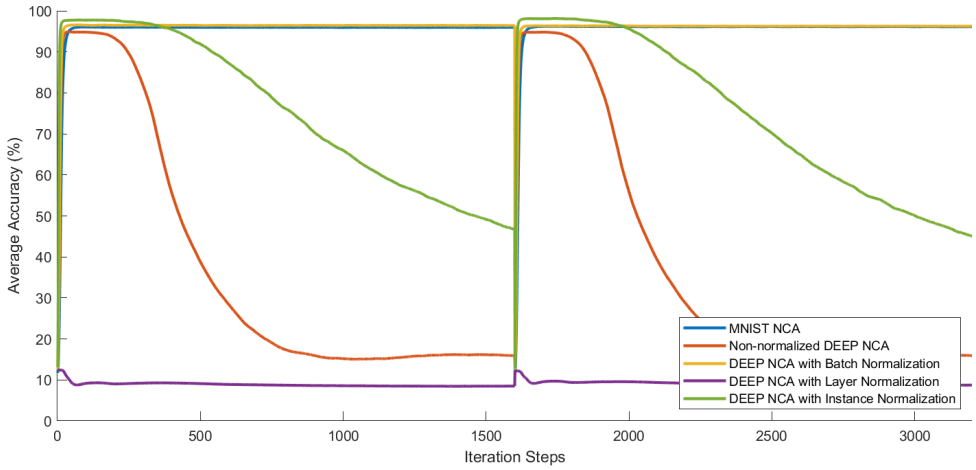
Figure 4.5: Comparison of internal dynamics of the DEEP NCA trained with $Niters = 50$ and $Niters = 75$, with the DEEP NCA trained with original $Niters = 20$ and MNIST NCA.

4.2.2 Normalisation Effects

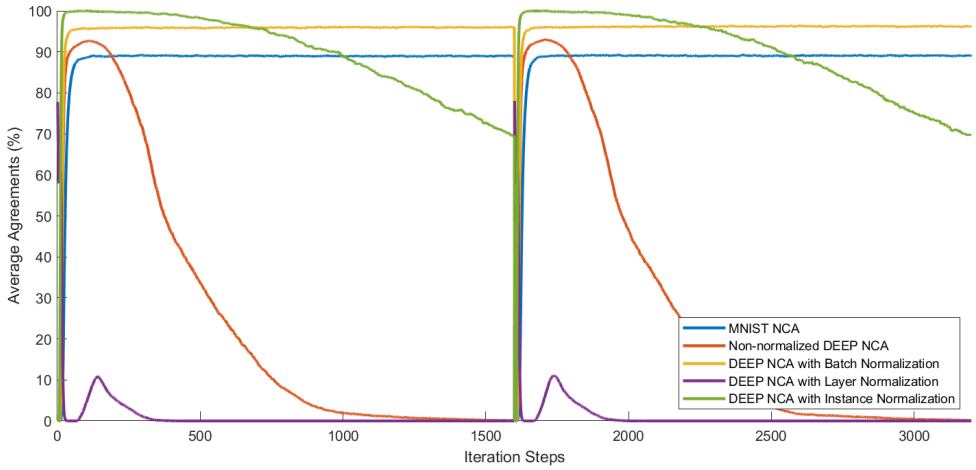
Another possible solution to the problem of unstable behaviour is to introduce normalisation layers to the network. They are widely used and are the go-to when searching for improvement on the stability of the network's training and test. We have experimented with the normalisation layers added to the DEEP NCA architecture after every layer, except the last one. The comparison of the obtained performance results is shown in Figure 4.6, with the non-normalised DEEP NCA provided for contrast.

The Layer Normalisation scheme does not match completely this application as the accuracy and agreement are reduced to 10% and 0% respectively. The Instance Normalisation enhances the initial dynamics of the automaton as it hits the largest accuracy and agreement and does that quicker than other configurations, but though the stability margin is improved the continuous stability is not achieved. The most impressive result is accomplished by the Batch Normalization, which brings back the stability property. The overall performance of the Batch Normalized model matches the accuracy of the MNIST NCA and provides even better agreement, as it utilizes the larger neighbourhood. Experiments with a range of up to 64k steps have proven that this behaviour is continuous.

To understand if the Batch Normalization has solved the problem of the continuously increasing magnitude of the state vector, we have investigated how it changes, with the results presented in Figure 4.7. As presented, the Batch-Normalized model can control the state magnitude initially, although on a larger level than the MNIST NCA, until it starts to rise after the 1000th iteration. This increase is much more dynamic than the growth in the DEEP NCA case and it looks



(a) Average Accuracy (%).



(b) Average Agreement (%).

Figure 4.6: Performance comparison of the DEEP NCA model with different normalization layers applied. The original MNIST NCA is shown in blue and non-normalized DEEP NCA in orange colour.

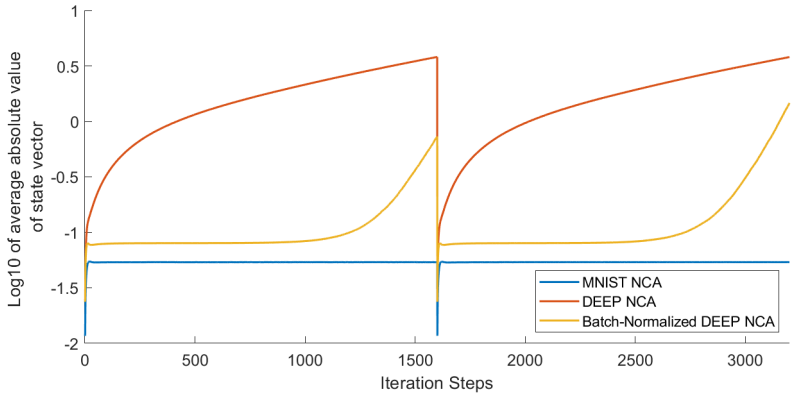
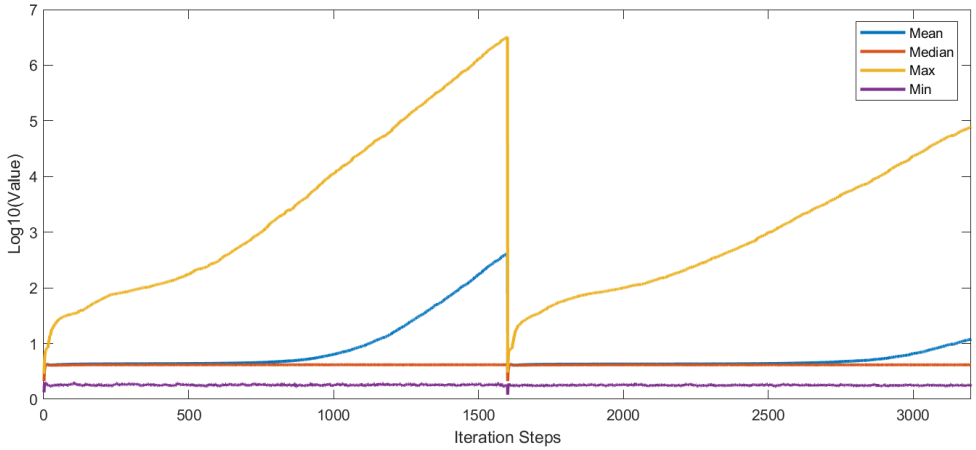


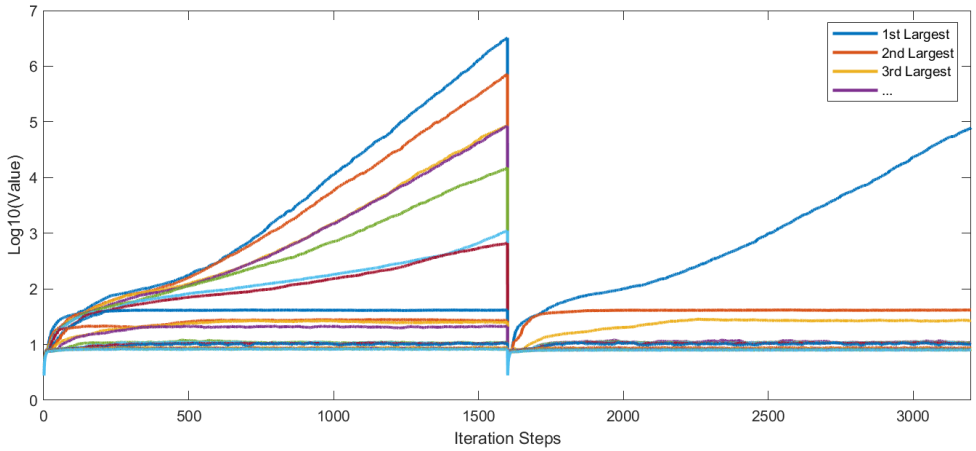
Figure 4.7: Comparison of the average magnitude of the state vector in the Batch-Normalized DEEP NCA (yellow), with the non-normalized DEEP NCA (orange) and MNIST NCA (blue).

contradictory to the excellent accuracy and agreements achieved by the Batch-Normalized DEEP NCA.

A possible explanation might be that few images exhibit extremely unstable behaviour, with the cell values achieving large magnitudes which could skew the mean. If the number of these unstable samples is relatively low compared to the entire test set of 10k samples, this would not appear in the average accuracy and agreements metrics though. To answer that we have collected data on the maximum absolute value in the state vectors array for every test image and presented the basic statistics: maximum, mean, median and minimum in Figure 4.8a. The discrepancy



(a) Mean, Median, Max, and Min statistics of the maximum absolute value in the state vector per sample

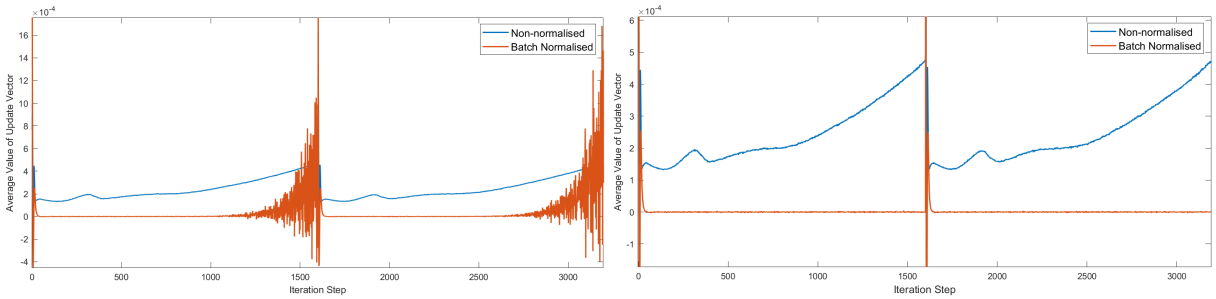


(b) First 20 of the maximum absolute values of the state vector per sample

Figure 4.8: Statistics of the maximum absolute value of the state vector for each test image for the Batch-Normalized DEEP NCA.

between the mean and median proves our theory, as it shows that the data is becoming more and more right-skewed. It is even more evident if we look at Figure 4.8b, which shows the 20 largest maximums of the state vector. We can see that it is just up to fifteen images that are causing a rapid increase in an average absolute value of the state vector shown in Figure 4.7, and have their maximum value in the state larger than 10, as the plot scale is logarithmic. For all the remaining images, the absolute value of a state vector element is bounded by 10 and hence stabilised.

This agrees with the earlier hypothesis, that only relatively few samples exhibit exceptionally unstable behaviour, while the vast majority act well which enables a batch normalized model to achieve the performance even better than the original model. We show that the batch-normalization indeed brings the average update value to zero and so solves the issue of continuous instability on Figure 4.9. The average value of the update vector computed with all test images,



(a) Average state update value using all test samples (b) Average state update without 20 largest outliers

Figure 4.9: Comparison of the average value of update vector between the non-normalised DEEP NCA and Batch- Normalised DEEP NCA. The left plot uses all test samples, while the right has removed 20 largest outliers beforehand, ten positive and ten negative ones.

presented in Figure 4.9a, indeed shows that sometime after the 1000th, the dynamics of the Batch-Normalized model are dominated by the extreme samples and the measured average jumps up and down repeatedly. Removal of the 20 largest outliers, for 10 on the positive and negative side, finally confirms our theory and reports that for a vast majority of images, 99.8% of the test set, the Batch-Normalization solves the stability issue, brings the average updates to zero and provides a DEEP NCA model that permanently achieves high accuracy and agreement.

We have observed over multiple executions that the images that cause the unstable behaviour are always the same, and for those images, the instability spreads quickly from cell to cell. As presented in Figure 4.10, the average number of outlier cells over the entire test set is up to 1.4 at the 3200th step, while the maximum number of such cells in a single image grows up to 3k.

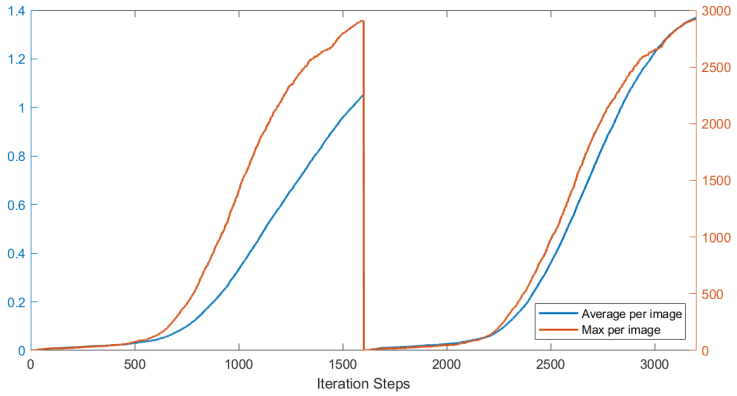


Figure 4.10: Average and Maximum number of the outlier cells in the test image for the Batch-Normalised DEEP NCA. The Colour of the axis determines the curve to which it relates.

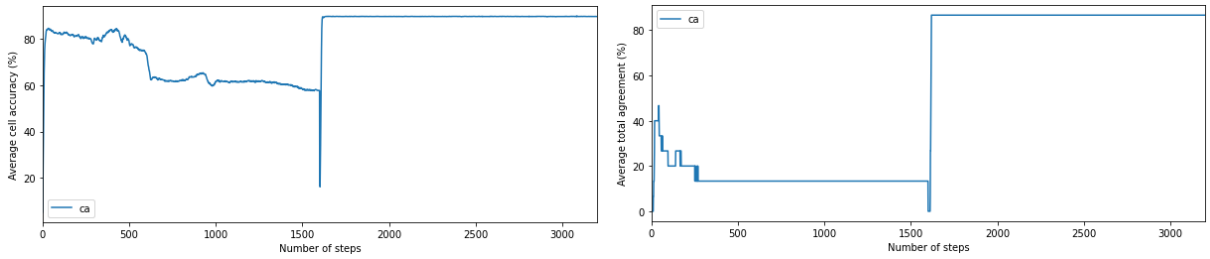
We present a group of those unstable images in Figure 4.11. It could be argued for presented

Figure 4.11: Fifteen images from the MNIST test set that causes the Batch-Normalized DEEP NCA model to behave unstably and have its maximum absolute value of the state vector exceeding 10.

examples, that they have some smaller or larger defects which might be one of the causes. For

example, all of the eights as well as the four and the first five, have one line a bit dragged out and both sevens have a horizontal line that ends with the left edge of the image. However, the MNIST test set [28] contains wide coverage of handwritten digits, with many more imperfect examples which do not exhibit such unstable behaviour. The batch normalisation statistical parameters, running mean and variance, as well as trainable scaling parameters gamma and beta, have been obtained using the entire available MNIST training set [28]. Hence, the fact that the normalisation operation was effective for a vast majority of test samples suggests that the remaining, unstable ones might lie outside some abstract distribution and characteristic of the correct samples, but determining that beforehand might be impossible because of its complexity. This might be an effect of the phenomenon known as bifurcation, where a small change to the parameter of a dynamic system results in the abrupt change of its behaviour, and responsible for that could be some minor specific properties of those images.

Nevertheless, the training process of the automaton ensures that it can recover when the input image changes. A neural network is taught how to deal with old, residual values in cells and quickly propagate changes. Using this mechanism it is also able to heal itself even after the extremely unstable samples, such as those in Figure 4.11, as presented in Figure 4.12. One of the



(a) Average Accuracy (%)

(b) Average Agreement (%)

Figure 4.12: The performance recovery in the Batch-Normalised DEEP NCA model, when the unstable image mutates to the stable one.

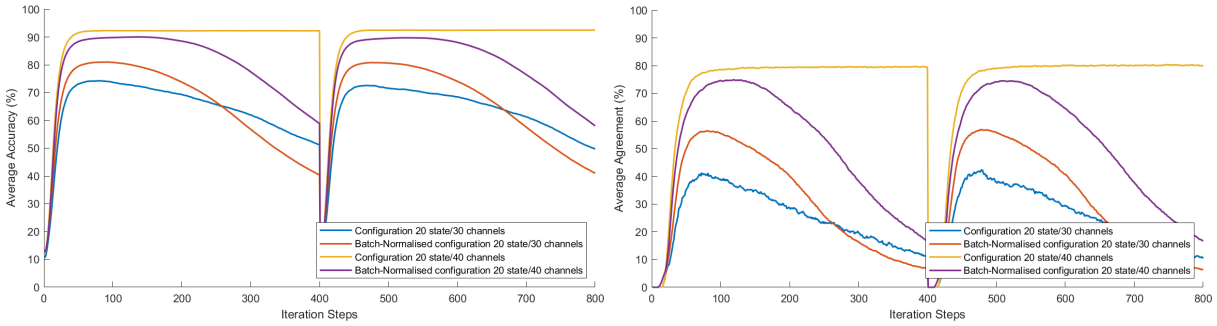
reasons is that the state values of dead automaton cells are explicitly set to zero, consequently clearing out a portion of extreme magnitudes. The cells, where the previous and current digit overlap produce the opposed updates so that the new state values does not exceed the stable range and give a correct prediction.

Consequently, the unstable samples problem could be safely ignored, because of their relatively low occurrence and the ability of the model to recover back to the correct behaviour and prediction. The Batch-Normalisation solves the long-term stability issue for the presented example of the DEEP NCA architecture and even enables it to outperform the MNIST NCA model. However, this solution is not generalisable and its effectiveness depends on the architecture. We have applied the batch normalisation to two other unstable configurations of MNIST NCA shown in Figure 4.1, 20 state / 30 channels and 20 state / 40 channels, and obtained a stabilisation only for the latter one, as presented in Figure 4.13.

Summary

We have presented a detailed analysis of the internal dynamics of the DEEP NCA architecture and shown that the addition of batch normalisation layers has solved the stability issue. However, we have not generalised our search and out of two additional architectures which we tested with batch normalisation, one has experienced a degradation of the performance.

It is clear that determining whether a developed model will be stable or not is difficult, as even small changes to the architecture configuration can result in different behaviour, as shown



(a) Average Accuracy (%)

(b) Average Agreement (%)

Figure 4.13: Result of applying batch normalisation to two other unstable configurations of MNIST NCA, the 20 state / 30 channels and 20 state 40 channels.

in Figures 4.1 and 4.13. The neural cellular automata is a recurrent neural network with the addition of residual connection, per Equation 3.1, which also can be interpreted as a feedback controller with the transfer function defined by the network. Consequently, a general approach to this problem should relate to the theory of stability and research on stability analysis of recurrent neural networks, such as a PhD thesis by Knight [22]. An alternative approach could be to extend every cell by a reset component, as shown by Sandler et al. [40], which gives each cell a mechanism to control its magnitude and stabilise the updates.

Chapter 5

Automaton Quantisation

The floating-point resolution models require a substantial amount of memory as well as many relatively expensive and demanding operations to calculate the neural network's output. At the same time, the FPSP devices rely on every pixel in the focal plane performing a simple, fast and power-efficient computation, which in the case of SCAMP5 architecture is achieved with analogue arithmetics that tend to be lossy and erroneous. However, accurate implementation of NCA would require discrete storage and arithmetic, as argued in Section 4.1. Consequently, quantisation is a necessary step to reduce the complexity and latency of the network and convert it to a representation that might be feasible by future iterations of FPSP devices.

This chapter describes the quantisation process of the example of SMALL NCA architecture with two alternative methods. The first sections discuss the usage and results of the post-training quantisation procedure from the TensorFlow Lite framework. In the next section, the development of custom quantisation procedure, inspired by work of Zhou et al. [50], is presented providing the step by step procedure as well as the evaluation of the results.

5.1 TensorFlow Lite Quantisation

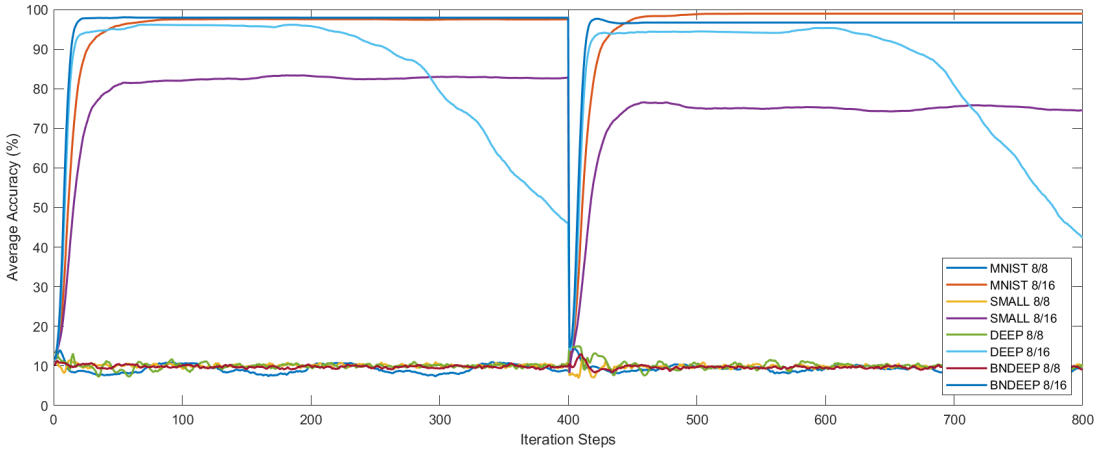
The TFLite framework provides a selection of few optimisation configurations described in Section 3.8.1, including full integer quantisation in a post-training and training-aware approach. Both methods were attempted in the project, however, due to the significantly different training procedure than in exemplary models and resulting issues with configuration, the training-aware strategy was discarded. In the post-training approach, it is possible to quantise the weights of the network to 8-bit representation, while the activations, as well as input and output, could be quantised to either 8-bit or 16-bit form, with each being possible in signed and unsigned versions. In the presented NCAs, both the weights and inputs/outputs are allowed to take negative values, so only the signed arrangements are suitable. Consequently, two quantisation configurations were tested, 8-bit weights paired with 8-bit activations and 8-bit weights with 16-bit activations, which will be further referred to as *8/8* and *8/16* respectively.

5.1.1 Model Quantisation

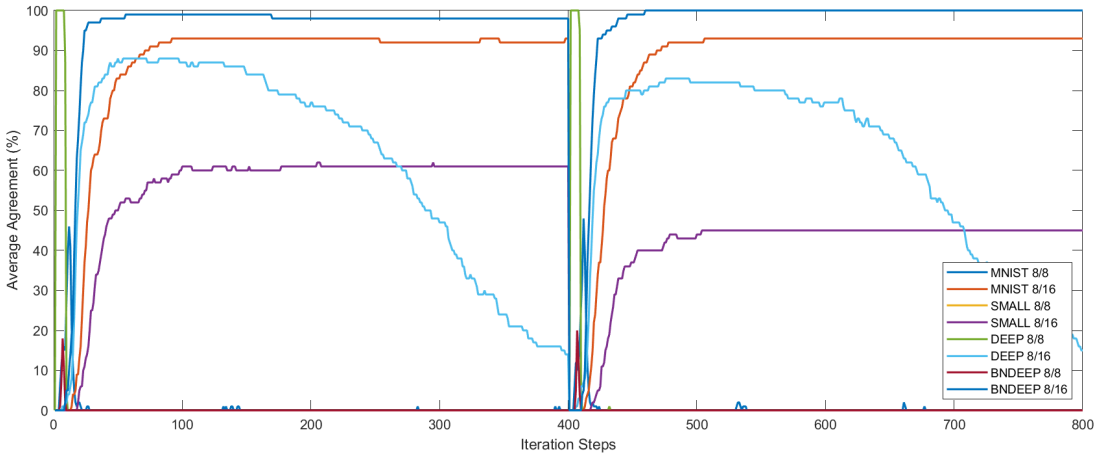
The presented NCAs operates originally on the floating-point input, representing the grayscale images by values in the range from 0 to 1. The remaining, mutable channels of the cell's state are initialised with zero, but their value change over the iterations, as the system operates similarly to the feedback loop. Hence, the efficient and accurate quantisation of the network to the sole integer form requires a representative dataset, a set of network inputs that represent the range of

possible values. Such a dataset is created first by sampling 200 random images from the MNIST training set [28] and processing them by NCA for 200 iterations, saving the cell state matrix at every step. Then the result, which contains the possible NCA cell state at every step from the 1st to the 200th, is randomly sampled to create a representative dataset of 1000 elements. Such downsampling is necessary due to memory limitations. The TFLite framework uses it then to determine the maximum and minimum values of the NCA's input and the activation of each layer.

Having the representative dataset and trained TensorFlow Keras network, the conversion to quantised *.tflite* is performed by the framework. The comparison of the performance of quantised NCAs is shown in Figure 5.1. The evaluation of the batch normalised DEEP NCA ensures that the outlier samples from Figure 4.11 are excluded to prevent biasing the min/max values.



(a) Average Accuracy (%).



(b) Average Agreement (%).

Figure 5.1: Performance comparison of TFLite 8/8 and 8/16 quantisations of following NCAs: MNIST, SMALL, DEEP and Batch Normalised DEEP (BNDEEP) over a small batch of 100 test images.

Due to the long execution time, as the TFLite interpreter can invoke a single sample at a time, not a batch, the analysis was performed on a small batch of 100 samples. This explains the visible performance drop for the *8/16* quantisation of the SMALL NCA when compared to its full-resolution results. Nevertheless, obtained data shows that the *8/8* quantisation is not sufficient as it performs poorly in all cases. The fact that the *8/16* configurations shares the same weights quantisation suggests that the issue lies in not sufficient representation of the input,

which because of the recursive nature grows in magnitude with time. To explain that, Table 5.1 shows the max and min value observed in the representative dataset for each configuration together with the quantisation step, the difference between two consecutive quantised values. The

Table 5.1: Comparison of MIN and MAX values observed in a representative dataset and the quantisation step for the TFLite quantised models.

Configuration	MNIST 8/8	MNIST 8/16	SMALL 8/8	SMALL 8/16	DEEP 8/8	DEEP 8/16	BN DEEP 8/8	BN DEEP 8/16
Min Value	-3.299	-3.304	-3.557	-4.495	-7.135	-6.638	-6.445	-6.422
Max Value	3.378	3.374	4.470	3.532	8.416	10.879	7.971	7.993
Quantisation Step	$2.62 \cdot 10^{-2}$	$1.03 \cdot 10^{-4}$	$3.15 \cdot 10^{-2}$	$1.37 \cdot 10^{-4}$	$6.1 \cdot 10^{-2}$	$3.32 \cdot 10^{-4}$	$5.65 \cdot 10^{-2}$	$2.44 \cdot 10^{-4}$

difference in the quantisation step for every configuration is of the order of two and is causing the *8/8* configurations to lose too much accuracy in the cell state representation, resulting in degradation of the label prediction.

5.1.2 Manual Integer Inference with Quantised Model

As described in Section 3.8.1, the *.tflite* models are treated by the TFLite Interpreter that exposes straightforward API and hides all complex operations. To achieve that on the custom architectures, such as FPSP devices, the necessary operations that arise from equations from Section 3.8.1, has to be implemented and the quantisation parameters pre-processed. This section describes the inference algorithm that enables to manually interpret the *.tflite* models and hence is also a base for the actual procedure for the FPSP devices. While the next section presents the final pre-processing of parameters and conversion of the remaining floating-point coefficients into the integer format.

The Algorithm 3 presents the scheme of the evaluation execution for the TFLite quantised NCA model. The logic of incrementing the current state matrix of cells by the computed update to produce the next iteration's state as well as the accuracy and agreement determination is omitted here for clarity, as they do not change. Also, only a convolutional layer is mentioned here, but it could be any layer that is a standard TensorFlow Keras layer, however, the exact execution details might differ. The detailed and convenient breakdown of the *.tflite* model and its actual layers can be conveniently seen with a Netron tool by Roeder [38] and an example of the alternative layer - quantised Batch Normalisation, is explained later in Section 5.1.3.

Essentially, the modification to the original evaluation approach [36] includes the addition of scaling operation after every layer's computation, per Equation 3.6. This is handled internally by the TFLite Interpreter, however, with the implementation being intended on FPSP devices these operations has to be done manually. Furthermore, the algorithm shows the calculation with the M constants already decomposed to the normalised representation, as integer M_0 and 2^{-n}, Equation 3.8, with the derivation of that decomposition is presented in the next section.

Finally, this algorithm demonstrates the subsequent operations that have to be transferred to the FPSP device code. Moreover, when implemented in Python code, with every operation executed in integer resolution, provide a convenient simulation of the execution performance on the FPSP. Naturally, assuming that it uses digital computation and storage, not the analogue.

5.1.3 Extracting Quantisation Parameters For Manual Implementation

The TFLite provides the integer form of the layer weights, so only three steps are remaining to obtain all the parameters in the integer format. First, converting the floating-point representation of the MNIST images into the integer form, respecting the correct scale. Second, implementing the scaling procedure at every layer according to Eq. 3.6, ensuring that the respective scales are

Algorithm 3 Manual Fully Integer Inference of the .tflite model

Convert all images to the integer format beforehand
Initialise immutable channel of X with a sampled image and set all the mutable channels to zero

for $iter = 1, 2, \ldots, IterationSteps$ **do** ▷ Number of automaton execution steps
  $X_{in} \leftarrow X$ ▷ Take starting state array for automaton

  for $L = 1, 2, \ldots, NetworkLayers$ **do** ▷ Rule computation by CNN
    if $L > 1$ **then** ▷ If not the First Layer
      $X_{in} \leftarrow X_{out}$ ▷ Take Output of Previous Layer
    end if
    $X_L \leftarrow Conv2D(X_{in}, W_L)$ ▷ Convolve with Layer Weights
    $X'_L \leftarrow Multiply(X_L, M_{0,L})$ ▷ Multiply channel-wise by Layer M_0 vector
    $X''_L \leftarrow ShiftRight(X'_L, n_L)$ ▷ Shift bits of X''_L right by n_L positions
    $X_{out} \leftarrow ReLU(X''_L)$ ▷ Apply the activation function
  end for
  $X'_{out} \leftarrow X_{out} \cdot M_{0,output}$ ▷ Multiply by the Output-to-Input scale $S_{0,output}$
  $X''_{out} \leftarrow ShiftRight(X'_{out}, n_{out})$ ▷ Shift bits of X'_{out} right by n_{out} positions
  $X \leftarrow AddToMutableChannelsOnly(X, X''_{out})$
  ▷ Stochastically add the computed step update to the mutable channels of the current cell state matrix to produce the state at the next iteration

  GetAccuracyAndAgreement(X, labels) ▷ Evaluate the classification metrics
end for

used for corresponding channels, and also decomposing them into the normalised representation per Eq. 3.8, which uses only integer operations and bit-shifts. Finally, it has to be ensured that the produced step update (network's output) that is to be added to the current cell state, are both represented with the same scale. The above steps are described below in this section in the same order.

Input Image Quantisation

The conversion of the input is achieved by using the Equation 3.2 in the form

$$q_{inp} = round(\frac{f_{inp}}{S_{inp}} + Z_{inp}) \tag{5.1}$$

where q_i denotes the quantised input image, f_i floating-point image in range (0,1), S_i the scale and Z_i zero point. The TFLite quantisation gives a single S and Z parameter for the entire input array, so the operation is straightforward. This process is necessary, as the value of 1.0 in the input image will correspond to different integers depending on the range of the observed values in a . The particular values of S_{inp} and Z_{inp} for the SMALL NCA input quantisation are shown in Table 5.2. In that instance, the integer value that equates to 1.0 in the floating-point image is 7290 and the *alive_threshold* of 0.1 becomes 729. However, it has been observed empirically for all 8/16 quantised models that they perform very well with standard *uint8* image representation, using range 0-255, and it even prolongs the stability margin in the case of DEEP NCA, as multiplication of two large factors is avoided, resulting in a smaller increase of state magnitude. It is also beneficial in terms of SCAMP5 implementation, as it reads the image in *uint8* format and so no additional scaling is needed.

Calculation of the M constants

Similarly, every layer activation and the final output has its pair of S and Z coefficients that concern the entire array. However, as the layer weights are quantised separately for every output channel, the layer with N output channels has N scale and zero point coefficients, that can be grouped as a vector

$$\bar{S_w} = \left[S_1, S_2, \ldots, S_N\right] \tag{5.2}$$

$$\bar{Z_w} = \left[Z_1, Z_2, \ldots, Z_N\right] \tag{5.3}$$

Consequently, at the layer *"L"* with *"N"* output channels, the computation of the M parameter results in a vector

$$\bar{M_L} = \left[M_1, M_2, \ldots, M_N\right] \tag{5.4}$$

that in the computation has to be applied to the respective channels. The quantisation parameters obtained for the SMALL NCA are shown in Table 5.2.

Table 5.2: Quantisation parameters of the 8/16 TFLite quantisation of the SMALL NCA model.

	Input	First Layer Activation	Second Layer Activation	Third Layer Activation/Output
Scale S	$13.7 \cdot 10^{-5}$	$6.24 \cdot 10^{-5}$	$3.43 \cdot 10^{-5}$	$3.16 \cdot 10^{-5}$
Zero point Z	0	0	0	0
	-	First Layer Weights	Second Layer Weights	Third Layer Weights
Average Scale S	-	$3.34 \cdot 10^{-3}$	$3.16 \cdot 10^{-3}$	$3.13 \cdot 10^{-3}$
Zero point Z	-	0	0	0
	-	First Layer Computation	Second Layer Computation	Third Layer Computation/Output
Average M	-	$7.35 \cdot 10^{-3}$	$5.75 \cdot 10^{-3}$	$3.39 \cdot 10^{-3}$

The scales $\bar{S}_w$ for every layer output channel are only presented averaged to provide a view of their magnitude but also avoid displaying a large quantity of data, as the first two layers of the SMALL NCA model have 40 output channels. Finally, the zero point parameter Z for every quantised array is 0 which simplifies the equations and avoids shifting the values before multiplications.

Therefore, as every M coefficient is calculated it remains to convert them to the "normalised" representation, shown in Equation 3.8 so that the multiplication by floating-point number M, Equation 3.6, could be implemented as an integer multiplication followed by bit-shift operation. It is necessary considering the goal is to minimise the latency and create the implementation on the FPSP device.

Decomposition and Quantisation of the M constant

The TFLite interpreter uses the C++ *std::frexp* function to handle the decomposition into normalised fraction M_0 and an integral power of two. It returns the M_0 as a double type, which is then scaled up to the *32-bit* integer. Consequently, the initial floating-point coefficient M is represented with *32-bit* integer and a specific number of right shifts. This method guarantees relatively high accuracy of the representation as it maps the floating-point numbers in the range (0,1) to the 2^{31} possible levels.

However, allowing the M_0 to take such a large value will raise a problem on the platform without the multiplication instruction, such as SCAMP5, as it would require numerous add instructions to model it. Therefore, we attempt to achieve more compact decomposition while still maintaining the performance.

The process is performed separately for each layer $\bar{M}_L$ vector because of significant differences in their average value as shown in Table 5.2 - for the first layer, it is an order of magnitude larger than for the others. Handling each separately guarantees minimal representation and precision.

The approach is to first find the negative exponent of two n, that results in 2^{-n} being small enough so that when used as a quantisation step will maintain a sufficient amount of accuracy and prevent all the values in the vector from being mapped to the same integer. Then, having the n selected for a layer L, the vector $\bar{M}_L$ can be quantised with element-wise operation

$$\bar{M}_{0,L} = round\Big(\frac{\bar{M}_L}{2^{-n}}\Big) \tag{5.5}$$

where $\bar{M}_{0,L}$ is a integer-type vector. However, choosing the right n is intuitive, and although some idea for starting value might be taken from Table 5.1 by looking at the magnitude order of quantisation step in the successful 8/16 quantizations, the ultimate selection needs to be searched experimentally.

To better illustrate the approach, the first elements of the $\bar{M}_L$ for the first layer are given below

$$\begin{aligned}\bar{M}_{FirstLayer} = [&0.01441852 \quad 0.00600085 \quad 0.00788765 \quad 0.00668339 \quad 0.01357831 \\ &0.00621187 \quad 0.00670437 \quad 0.00762232 \quad 0.01313682 \quad 0.00508501 \quad \dots\,]\end{aligned} \tag{5.6}$$

and the remaining elements of the vector are relatively similar to those presented. The first negative exponent n that results in 2^{-n} smaller than every element in $\bar{M}_{0,L}$ is 7, so it could be the first candidate. However, as $2^{-8} = 0.003906$ it would map the average M in the first layer, from Table 5.2, through the Eq. 5.5 only to 2. Similarly, all the elements presented in Eq. 5.6 will be mapped to very low integers such as 2, 3, or 4, which will result in visible precision loss. The distance between the consecutive quantisation levels, the quantisation step, is too big when compared with the differences in the values to be quantised. Consequently, the numbers that should be considered dissimilar are mapped by Eq. 5.5 to the same bag/integer value, as only few of the integer values are in use. The evaluation trial confirms the above issues and the performance drop with $n = 8$ is significant.

Applying this procedure repeatedly for every layer and trying out different combinations of n, has led us to the parameters shown in Table 5.3, for each of the SMALL NCA model's layers. The corresponding integer vector $\bar{M}_{0,L}$ can be determined by applying the equation 5.5 to the respective $\bar{M}_L$ vector.

Table 5.3: Value of the exponential n in the manual implementation of the TFLite internal logic for the FPSP devices in the example of quantised SMALL NCA.

	First Layer Computation	Second Layer Computation	Third Layer Computation / Output	Output-to-Input
Exponential n in 2^{-n}	14	14	14	2

The resulting integer representation at every layer uses the values in the range (0,118) which is significantly lower than the *32-bit* conversion offered by the TFLite interpreter. In the case of the quantization of DEEP NCA architecture with the same method, this range was even smaller, (0,18).

Output to Input Scaling

To complete the automaton iteration, the produced update needs to add to the previous cell state to yield the next iteration's state. However, as presented in Table 5.2, the network's input, which is the cell state, has a different scale S than the network's output. Consequently, before the output is added to the cell state it needs to be rescaled.

Incrementing a floating-point number f_1 by another floating-point number f_2, where both are represented in a quantised form according to the Eq. 3.2, can be expressed as

$$f_1' = f_1 + f_2 \tag{5.7}$$

$$S_1(q_1' - Z_1) = S_1(q_1 - Z_1) + S_2(q_2 - Z_2) \tag{5.8}$$

$$q_1' = q_1 + \frac{S_2}{S_1}(q_2 - Z_2) \tag{5.9}$$

with the subscript 1 denoting cell state and 2 the computed update.

In the presented example of SMALL NCA 8/16 quantisation, the zero point Z of the output is 0, as per Table 5.2. Hence, before we add the computed update to the cell state it only needs to be scaled by

$$M_{output} := \frac{S_2}{S_1} = 0.2307 \tag{5.10}$$

according to the values read from Table 5.2.

Now it only remains to apply the normalisation procedure, same as for the M constants, to represent this with integer operation. The first n that gives $2^{-n} < M_{output}$ is $n = 2$ and applying the method from Eq. 5.5 results

$$M_{0,output} = round\Big(\frac{0.2307}{2^{-2}}\Big) = 1 \tag{5.11}$$

for the quantised representation of M_{output}. The evaluation runs has shown that this value of $M_{0,output}$ and $n = 2$, together with parameters obtained above for M_0 works perfectly and guarantee minimal accuracy loss. Naturally, selecting larger n is also possible, but with little benefits in precision.

Summary of performed Parameters Processing

Ultimately though, the process described above relies on the trial and error method and requires repeated evaluation executions to find out the good configuration. To even start with an experimental evaluation shown as Algorithm 3, the initial decomposition has to be done for M in all layers as well in the *Output-To-Input* scaling. The above sections have provided some insight on how to find suitable decomposition based on the values of actual TFLite parameters.

The complexity of the total search rises with the number of networks layers. In the case of the SMALL NCA, an example starting configuration for each layer was chosen so that $2^{-n_{start}}$ is a few times smaller than the minimum value of M at the layer. The obtained setup was then being tuned over few executions to adjust the n at every layer to minimise the performance loss. This process though could be automated in a way similar to a hyper-parameter search for the network training and several combinations can be executed in parallel and then compared. Possibly this conversion could be also applied layer by layer, starting with the first one, tuning it while using the floating-point parameters for the other ones. Then going to the next layer, finally arriving at output-to-input scaling, though that might not necessarily be entirely correct for deeper and complex networks.

Alternative Layer example - Batch Normalisation

Apart from the convolutional layer presented above, The TFLite quantisation supports all the standard layers that are part of the TensorFlow Keras library. Depending on the function of those layers, their *.tflite* quantised version might take on a different form and even be decomposed into two or more sublayers.

For example, the Batch Normalisation layer is decomposed by the TFLite quantisation into two layers, namely "multiply" and "add", as shown in Figure 5.2.

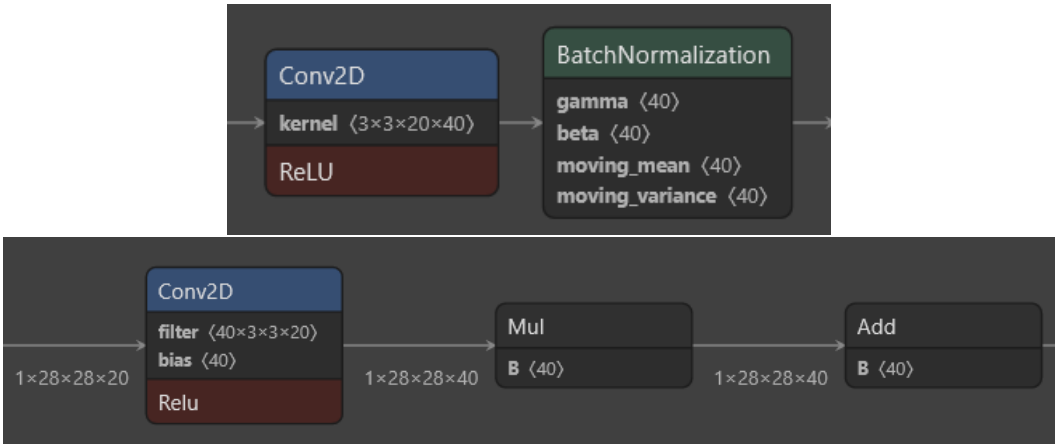


Figure 5.2: Visualisation of the Batch Normalisation decomposition in *.tflite* model. The figure shows a section of the Batch Normalised DEEP NCA. The top picture being a model before TFLite quantisation and the bottom shows the same part, but in the *.tflite* quantised format. Note the notation difference in the kernel representation, with the top image using (Kernel H; Kernel W; Input Ch; Output Ch), while the bottom shows the number of output channels first. Obtained using Netron [38].

The reason for that decomposition comes from the fact, that the parameters in the Batch Normalisation are fixed during the evaluation and hence, the normalisation Equation 2.1 can be rewritten as

$$\hat{x} = a \cdot x + b \tag{5.12}$$

$$a := \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \tag{5.13}$$

$$b := \beta - \frac{\gamma \cdot \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{5.14}$$

where both a and b can be pre-computed, forming vectors of size equal to the number of channels in the input. Those parameters are treated the same as the activations and in the 8/16 quantisation they are quantised to *16-bit* representation, with a single scale S.

The calculation process rewritten for quantised values, using mapping from Equation 3.2 and $Z = 0$, can be represented as

$$x_1 = a \cdot x_{in} \longmapsto q_1 = M_{BN} \cdot q_a q_{in} \tag{5.15}$$

$$M_{BN} := \frac{S_a S_{in}}{S_1} \tag{5.16}$$

and

$$\hat{x} = x_{out} = x_1 + b \longmapsto q_{out} = K_1 q_1 + K_b q_b \tag{5.17}$$

$$K_1 := \frac{S_1}{S_{out}} \text{ and } K_b := \frac{S_b}{S_{out}} \tag{5.18}$$

and all three parameters M_{BN}, K_1, K_b are constants and can pre-computed. Naturally, both multiplication and addition are performed channel-wisely, using respective values of q_a and q_b for corresponding channels in the input array.

Table 5.4 shows the above parameters for the first Batch Normalisation layer in the Batch Normalised DEEP NCA model. The obtained constants M_{BN}, K_1 and K_2 would then need to

Table 5.4: TFLite quantisation parameters for the first Batch Normalisation layer in the Batch Normalised DEEP NCA model.

Input Scale S_{in}	Intermediate Scale S_1	Output Scale S_{out}	a scale S_a	b scale S_b
$1.0013 \cdot 10^{-4}$	$5.7020 \cdot 10^{-4}$	$5.6435 \cdot 10^{-4}$	$6.1204 \cdot 10^{-4}$	$0.3956 \cdot 10^{-4}$
M_{BN}	K_1	K_2	Average $\lvert q_a \rvert$	Average $\lvert q_b \rvert$
$1.0748 \cdot 10^{-4}$	$1.0103 \cdot 10^{-4}$	$0.0701 \cdot 10^{-4}$	13265.35	10500.15

be decomposed into the normalised representation, following the method applied above for the parameters of convolutional layers.

Finally, the significantly large average $|q_a|$ is worrying in terms of the implementation on the FPSP devices deprived of the multiply instruction, such as SCAMP5, as the equivalent addition code would be long and most likely not efficient, and the remaining batch normalisation layers in the Batch Normalised DEEP NCA also have a similar characteristic. As imitating a multiplication by large quotient using add operations is not efficient, to improve the eventual performance of the FPSP implementation of batch normalisation layers a quantisation scheme with fewer bits could be used or the multiplication instruction could be provided.

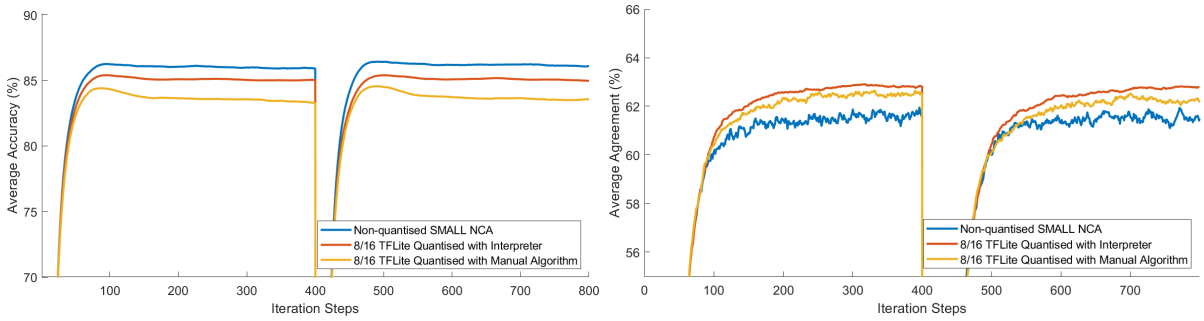
5.1.4 Performance of Manual Integer Inference

Here we show the comparison of the final results of TFLite quantisation on the SMALL NCA model. In Figure 5.3 are presented the base, non-quantised SMALL NCA together with the 8/16 quantisation of SMALL NCA that used the TFLite Interpreter to calculate the output and its alternative implemented manually following the Algorithm 3 and approach presented above. As shown, the quantisation process has introduced some degradation of the accuracy, which could be reduced further by enlarging the n parameters from Table 5.3. Interestingly, the quantization process improves the agreement slightly, most likely through the slight regularization by injection of quantization noise. The manual inference of the 8/16 TFLite quantisation was also performed on the DEEP NCA model and has achieved even better results as shown in Figure 5.4.

We compare the achieved performance, as well as model reduction size and required storage to the results of INQ quantization in the section 5.3, providing discussion on the benefits of each approach.

5.2 Implementing Modified INQ Quantisation

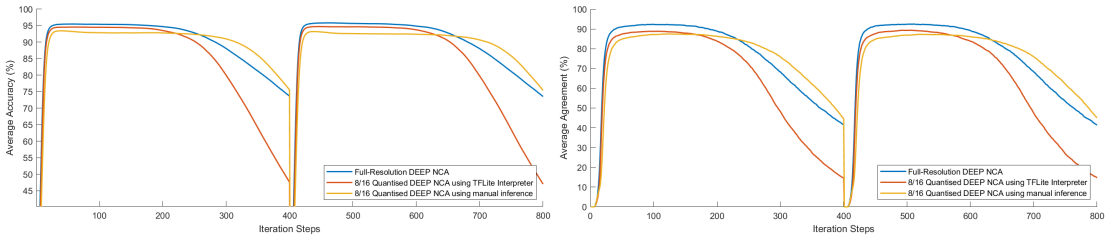
The main drawback of the TFLite post-training quantisation is a lack of flexibility that prevents from maximising the size reduction and implementing advanced optimisations. The disparity



(a) Average Accuracy (%) - y-axis value starts at 70%.

(b) Average Agreement (%) - y-axis value starts at 55%.

Figure 5.3: Performance of the 8/16 quantised SMALL NCA model on the full test set, showing results obtained using supplied TFLite interpreter (orange) as well as with presented manual approach (yellow). Full-resolution SMALL NCA results (blue) are provided for comparison.



(a) Average Accuracy (%) - y-axis value starts at 40%.

(b) Average Agreement (%) - y-axis value starts at 0%.

Figure 5.4: Performance of the 8/16 quantised DEEP NCA model on the full test set, showing results obtained using supplied TFLite interpreter (orange) as well as with presented manual approach (yellow). Full-resolution DEEP NCA results (blue) are provided for comparison.

between the 8/8 and 8/16 configurations, in terms of ability to express activations is also immense, 2^8 versus 2^{16}. Consequently, a search for an alternative quantisation was performed to try to reduce both weights and activations even further and hence improving the efficiency of the FPSP implementation.

The most desired conversion would be to transfer the NCA model to binary or ternary representation, allowing extremely fast and efficient operation. However, though binary and ternary networks proved to be efficient in the standard classification of MNIST digits, the complexity of the NCA approach is much larger as the network operates recursively and does use only small kernels, without any striding or pooling. Therefore, a realistic possibility is a scheme that quantises networks weights to a *M-bit* representation and the activations to a *N-bit*, with $M < N$. Finally, an INQ method by Zhou et al. [50] was selected as it provides a relatively straightforward algorithm that can be applied to any architecture, with a flexible configuration and most importantly the incremental approach that allows the network to adapt to approximation loss and maintain its long-term dynamics and accuracy.

This section is organised as follows, first, the preliminary optimisations are applied to the model so that the model itself is better suited for quantisation. Then the modified quantisation algorithm is described in detail with motivations for the changes applied to INQ, and followed by its application in the incremental quantisation of weights of the SMALL NCA model as well as the activations. Finally, it presents results obtained on the SMALL NCA model and confronts them with the floating-point version. Whereas, the comparison of this approach with the TFLite

quantisation, both in terms of the quantised model size and precision is given later in Section 5.3.

5.2.1 Pre-Quantisation Optimisation - Weights Clipping

The quantisation process is first applied to the weights and then follows to the activations, as the approximated weights might change the model's dynamics and produce slightly different output values. To start with the weights quantisation, an analysis of the distribution of weights values was performed to understand their min-max range and spread.

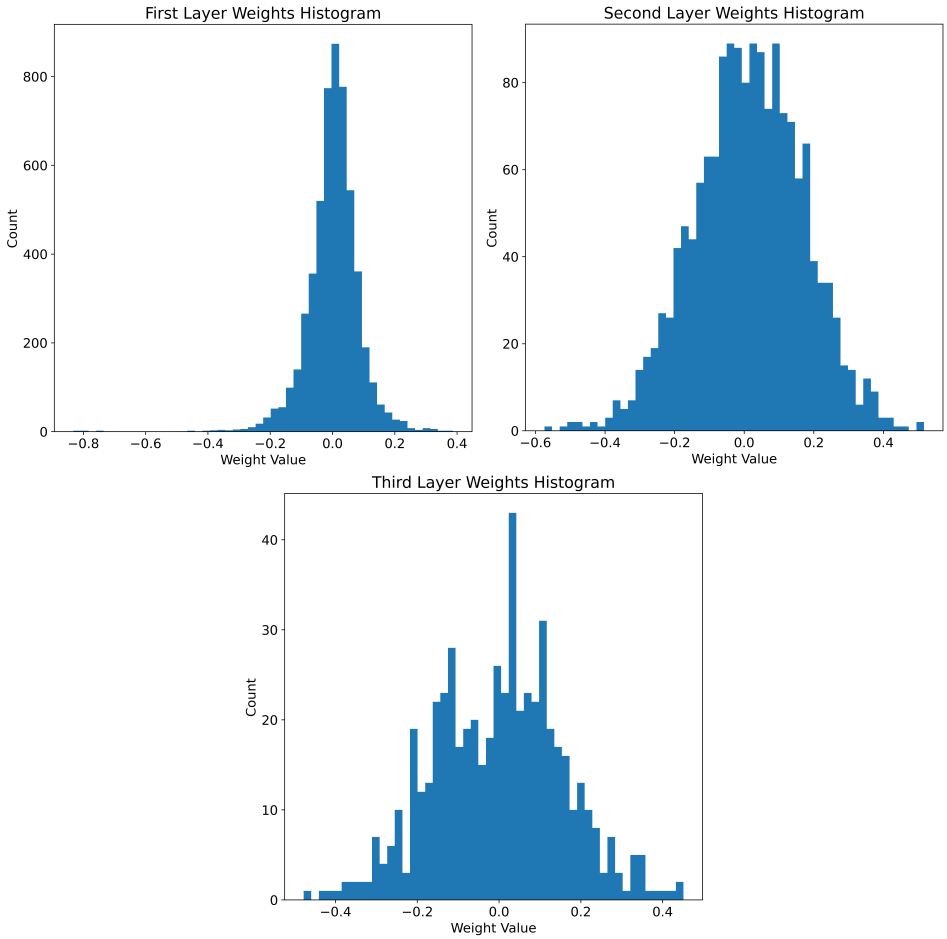


Figure 5.5: Layers Weights histograms for the SMALL NCA model.

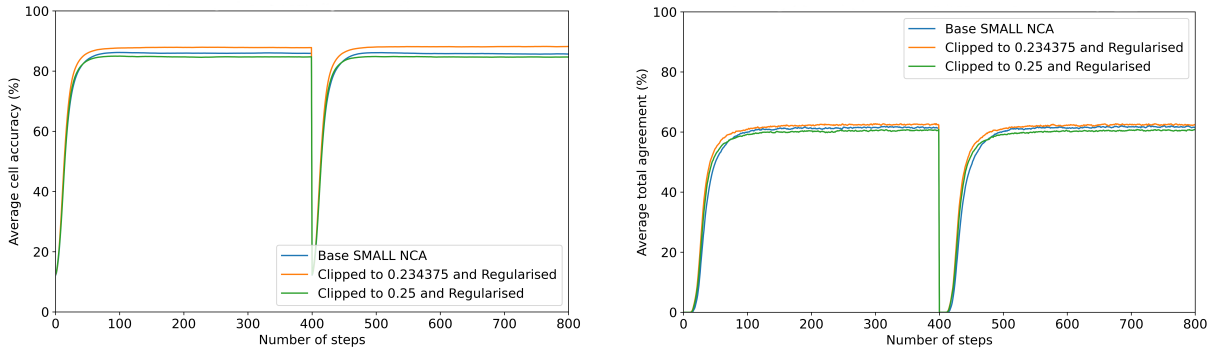
The histograms of the weights for all three layers of the SMALL NCA model are shown in Figure 5.5. The first observation that arises is that the weights at all layers have a distribution relatively close to the normal distribution, with approximate symmetry of the values. However, the first layer has a few comparably significant outliers that might affect the quantisation accuracy, namely a few occurrences of values around -0.8.

The precision of the quantisation algorithms depends directly on the mix-max range it has to cover, both linear approach from QNN [18] or TFLite [44] as well as power-of-two from [50] benefits from it being as small as possible. In addition, an approximate symmetry of the data is also desired as it eliminates the need to shift values by their zero points, simplifying the usage.

From the histograms, in Figure 5.5 we see that the majority of weights fit in range (-0.2,0.2) at all layers. It raises the question if the model will be able to achieve comparable performance if all the weights will be forced during training to limits themselves to that value range. The results of

that experiment, with two different clip values, are shown in Figure 5.6, although the exact value of 0.234375 is used instead of 0.2 for the reasons that will be explained in Section 5.2.4.

Finally, Figure 5.6 shows the performance comparison between the base SMALL NCA model and its clipped version.



(a) Average Accuracy (%)

(b) Average Agreement (%)

Figure 5.6: Performance comparison of the base SMALL NCA (blue) with its clipped and regularised version (orange and green).

5.2.2 Quantisation Algorithm

The advantage of the INQ scheme [50], is that it allows a flexible selection of the actual quantisation algorithm for the weights. Originally the "power-of-two" is used by the authors, expressing a specific quantisation level 2^q by a corresponding bit pattern in N-bit number. This approach has the advantage of the low precision levels, as $q \to -\infty$, being closer to each other, allowing for precise quantisation of small weights. However, in the inference process bit pattern has to be interpreted correctly to perform the multiplication with the activation value.

Hence, for the actual quantisation a linear algorithm was chosen as it enables to reuse of the Algorithm 3, created for the TFLite quantisation, and also can be used to conveniently quantised the activations. As the distribution of the weights is narrow and symmetric and the outliers are removed by the clipping operations, the linear quantisation can be also very efficient.

The linear quantisation algorithm is defined in Equation 3.13 or by a mapping in Equation 3.2 from TFLite. With the distributions of the weights being symmetrical, the zero-point constants Z from TFLite are eliminated, it remains only to define the min-max range of values and compute the quantisation scale, or step, for a selected *N-bit* approximation. We could simply follow the approach in the TFLite, Equation 3.3, taking max and min of data. However, this causes a complex process of decomposition of constant M into integer M_0 multiplier and 2^{-n}, which adds complexity to final code.

Instead, we could enforce symmetrical range and ensure that the scales S are already powers of two, which would drastically simplify the process and eliminate the entire decomposition procedure. To achieve that for a *N-bit* quantisation, a maximum allowable absolute value has to be chosen so that it can be expressed as

$$MaxAbsVal = \frac{2^{N-1} - 1}{2^F} \tag{5.19}$$

where $N - 1$ comes from the one that the most significant bit is required for encoding a sign. The integer F reflects how many bits in the binary format of the number are used to represent its fractional part.

In the case of SMALL NCA presented in the previous section, it has been shown that most of the weights are in range (-0.2,0.2) and with clipping the weights into this range in training, we could simply use this as a min and max value, but this would require decomposition procedure. Hence, the value close to 0.2 but representable as in Equation 5.19 should be selected for maximum optimisation. Estimating such a value could be done by first converting Equation 5.19 to form:

$$MaxAbsVal = 2^{N-1-F} - 2^{-F} \tag{5.20}$$

and selecting integers N and F such that 2^{N-1-F} is approximately equal to a desired range. In the case of 0.2, the closest power of two is 2^{-2}, so we have to ensure that:

$$-2 = N - 1 - F \tag{5.21}$$

which naturally depends on the selected bit-width N. The value of F then affects the quantisation scale, as

$$S = \frac{MaxAbsVal - 0}{2^{N-1} - 1} = \frac{1}{2^F} = 2^{-F} \tag{5.22}$$

so as F gets larger, the scale S gets smaller resulting in more and more accurate quantisation, but at the cost of increased bit width. Thanks to the symmetry, the scale could be calculated simply on the positive "side", that's why 0 is used as the minimum value in Equation 5.22. So now, the scale S is only represented as a power-of-two and thereby can be efficiently implemented on any hardware only with bit-shift operations.

Consequently, the vector of quantisation levels $\mathbf{P}$ will be the same for every layer, as each layer of SMALL NCA is clipped in the same way, and can be expressed as

$$\mathbf{P} = \{0, \pm S, \pm 2 \cdot S, \ldots, \pm(2^{N-1} - 1) \cdot S\} \tag{5.23}$$

Naturally, an advanced clipping strategy could be used, with different values per layer, end even different negative and positive clips, resulting simply in different scales for every layer.

The presented process of determining the *MaxAbsVal* can be applied also to the activations, knowing their representative dataset, and hence the entire evaluation of the network, according to the Algorithm 3, is deprived of the multiplication by M_0 constant.

5.2.3 Regularisation of Model Training

To finalise the implementation of INQ [50], a regularisation function was added to the training procedure, that penalizes the weights that are far from quantisation levels $\mathbf{P}$ and hence encourages the model to adjust to the quantisation regime.

If necessary, the regularisation itself could be tuned by adjusting its λ parameter if the effect is not sufficient. In the case of SMALL NCA, the obtained results report very good performance with used regularisation with $\lambda = 0.1$, shown in Figures 5.6 and 5.7.

5.2.4 Quantisation of SMALL NCA Model

In the previous section, the modified quantisation algorithm has been described in detail, allowing for its re-use with other models. Here the application to the SMALL NCA model will be described, with the model-specific results and parameters will be presented. The quantisation is divided into two parts, first, the model's weights are quantised utilising the incremental scheme, and then the quantisation is applied to the activations.

Weights Quantisation

With all the equations defined, it is left to decide on the bit-width of the weights quantisation and calculate the resulting parameters. The $N = 5$ was selected as both QNN [18] and INQ [50] have reported very accurate quantisations of models more complex than SMALL NCA, such as ResNet-18 [17], with the *5-bit* approximation.

Having $N = 5$ fixed, as the closest power-of-two to a proposed clip value of 0.2 is 2^{-2}, it remains to apply Equation 5.21 and obtain all the quantisation parameters as follows:

$$F_5 = 6 \tag{5.24}$$

$$S_5 = 2^{-F} = 0.015625 \tag{5.25}$$

$$MaxAbsVal_5 = 2^{5-1-6} - 2^{-F} = 0.234375 \tag{5.26}$$

$$\mathbf{P}_5 = \{0, \pm 0.015625, \pm 0.03125, \ldots, \pm 0.234375\} \tag{5.27}$$

with the $N - 1$ term being a consequence that one bit is needed to represent the sign, subscript 5 denotes the $N = 5$ quantisation.

It is now possible to train the final version of the SMALL NCA network, which is prepared for quantisation with a weight clip to ± 0.234375 and a regulariser towards the elements of P_5, and finally apply the modified incremental quantisation process. For the weights partitioning, the "largest magnitude first" was used, as INQ [50] reports its better performance, and two increment sequences experimented:

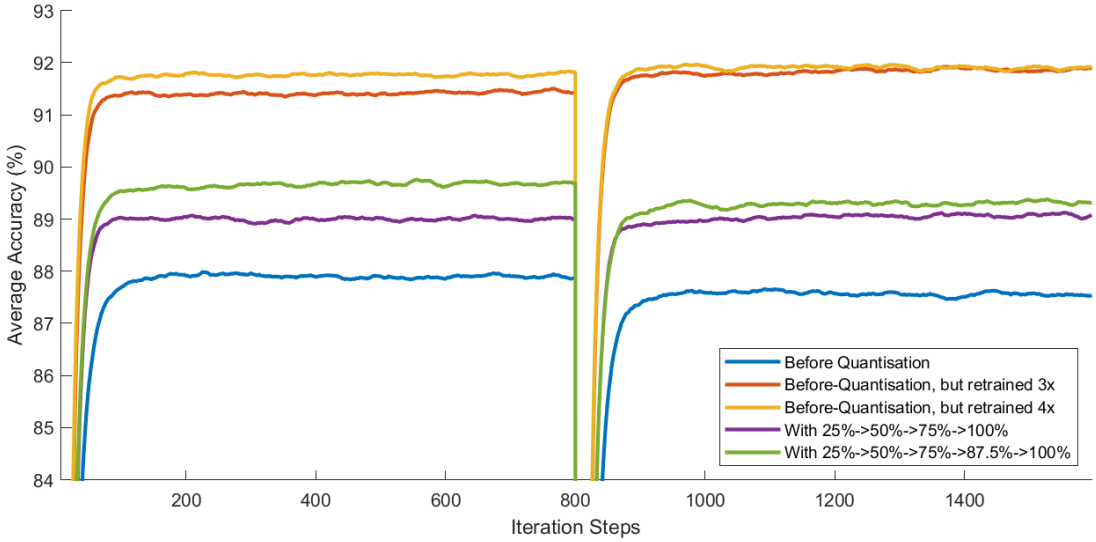
- $25\% \rightarrow 50\% \rightarrow 75\% \rightarrow 100\%$
- $25\% \rightarrow 50\% \rightarrow 75\% \rightarrow 87.5\% \rightarrow 100\%$

with the percentages denoting the number of quantised weights in each layer. Further, for simplicity, we will also refer to them as shorter and longer respectively.

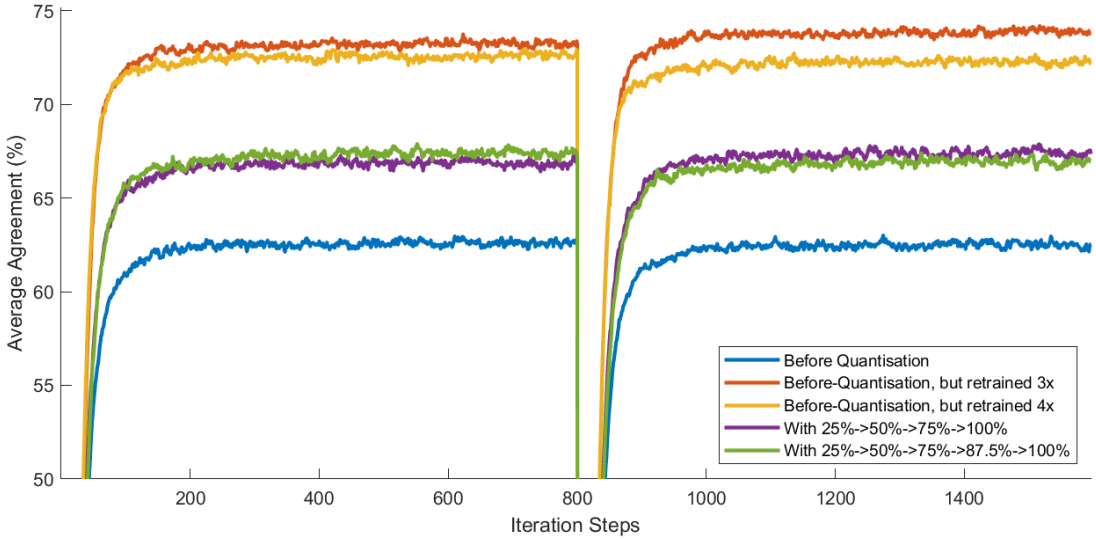
The results of the quantisation are shown in Figure 5.7, comparing the performance of the fully quantised SMALL NCA model, for each of the above partitioning sequences, with the full-resolution counterparts. During the quantisation, the model is re-trained three and four times respectively for the shorter and the longer sequence, and the re-training routine simply uses the procedure used to train networks, only it starts with the initial weights set. Consequently, to express the effect of repeated re-training and provide a relevant comparison for the quantisation results, we have also plotted the full-resolution SMALL NCA model re-trained three and four times.

As presented the *5-bit* quantisation of the weights achieve very good results, obtaining the performance better than the base SMALL NCA, however the loss due to the quantisation is visible when compared to full-resolution re-trained models and the difference oscillates around 2.5%. Amongst the two partitionings, the longer sequence can achieve slightly higher accuracy, as only 12.5% of model weights are not quantise after the last retraining, as opposed to 25% in a shorter one. Naturally, more steps can be added that allow tuning of the quantisation process to find the most optimal sequence for a specific model.

In addition, the behaviour of the model under the incremental quantisation process is shown in Figure 5.8, which presents the model's accuracy and agreement after every step in the shorter partitioning sequence. The metrics were always measured directly after a portion of weights were quantised, denoted as "After X%" and then following the model retraining, "X% Retrained". It shows that every retraining indeed allows the model to adapt to the applied quantisation and recover lost accuracy and agreement for every "Quantise-Retrain" pair. Repeated retraining even improves the performance of the model compared to the base SMALL NCA, however,



(a) Average Accuracy (%) - y-axis value starts at 84%.



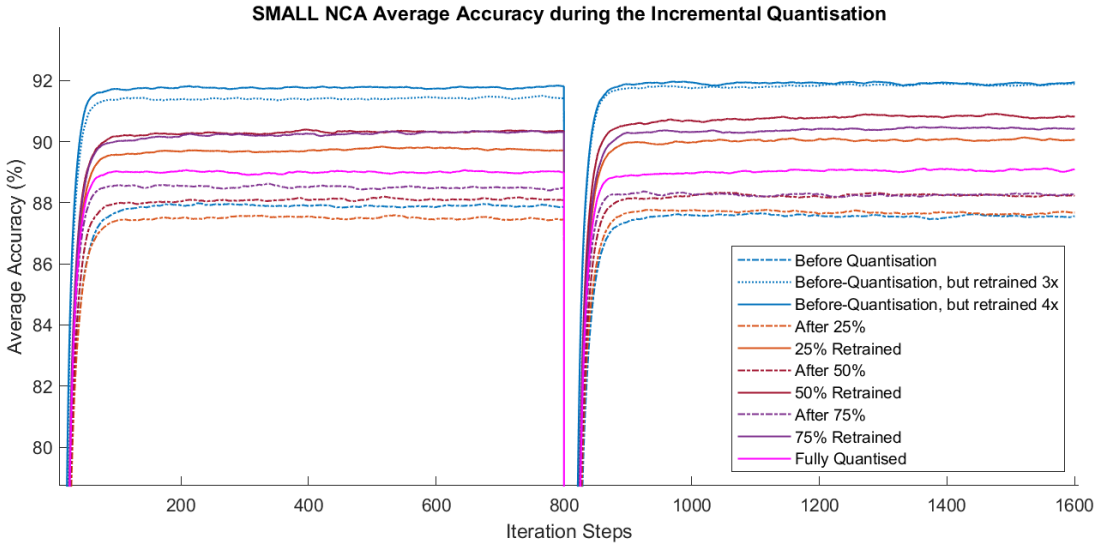
(b) Average Agreement (%) - y-axis value starts at 50%.

Figure 5.7: Performance comparison of the SMALL NCA with fully quantised weights, using incremental approach and two different partitioning sequences, shorter (orange) and longer (green), against the non-quantised version (blue).

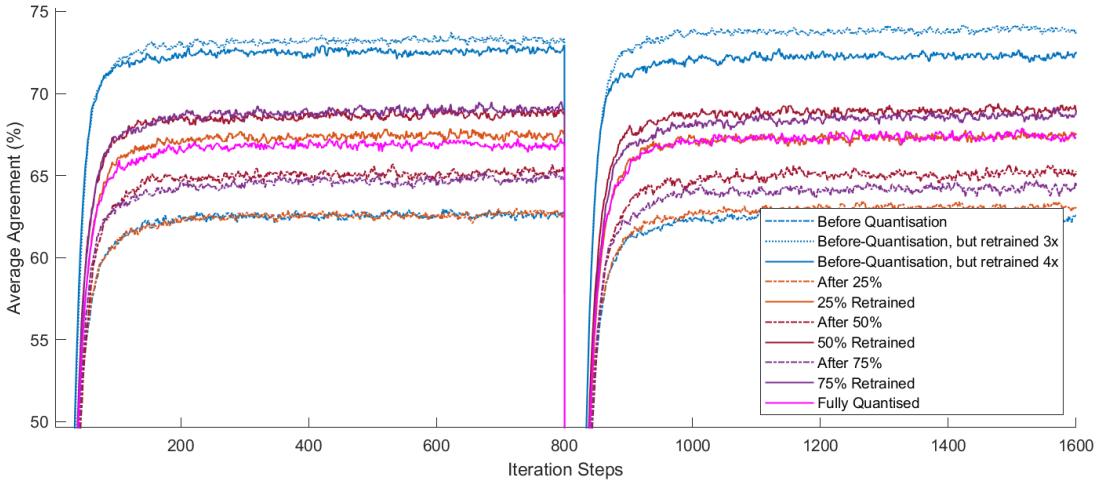
when compared to three times re-trained version, we see that some degradation is introduced. Naturally, it is to be expected with the approximation of quantisation, and the final overall performance is very good.

Activations Quantisation

The quantisation of the activations, output and input follows the approach utilised in the TFLite quantisation. As before, to establish the scale S it is necessary to determine the min-max range of the values. The cells state matrix, referred to as input, accumulates the updates over the iterations, hence its values are relatively larger than the ones of the activations or network's output, and might even require more bits to maintain precision. Whereas, the activations are obtained with a ReLU function and hence can take only non-negative values, which enables the removal of a sign bit.



(a) Average Accuracy (%) - y-axis value starts at 78%.



(b) Average Agreement (%) - y-axis value starts at 49.5%.

Figure 5.8: Evolution of performance in the SMALL NCA model during the steps of incremental quantisation. The blue line denotes the starting non-quantised model, magenta the final model with quantised weights. Intermediate steps are at 25%, 50% and 75% weights quantised, with a dashed-dotted line being measured just after quantisation of respective portion of weights, while continuous line represents model after retraining at that step.

Therefore, unlike the weights, they are considered separately allowing each to have a different bit-width and scale. To start with the quantisation, the idea of the representative dataset has to be applied to the quantised SMALL NCA model. It has been obtained on the entire training set, to keep it separately from the evaluation set, which was executed for 3200 steps with a digit mutation in the middle, and maximum absolute values in input, output, and each activation were recorded at every iteration. Furthermore, five separate executions were performed to provide statistical significance and indeed obtain a representation of possible maximum absolute value at every layer. The recorded maximums over all iteration steps and executions are shown in Table 5.5, together with a summary of each network layer. Having acquired those limits, we can now determine the required number of integer bits in quantised representation for each layer. As shown before in Section 5.2.2, we should start by determining the relationship between the N

Table 5.5: Maximum absolute value as well as the value range in consecutive layers in SMALL NCA.

	State/Input	First Layer Activation	Second Layer Activation	Update/Output
MaxAbsVal	∼3.4	∼2.85	∼2.5	∼1.8
Values Range	(-MaxAbsVal; MaxAbsVal)	(0; MaxAbsVal)	(0; MaxAbsVal)	(-MaxAbsVal; MaxAbsVal)

and F for each of them, being

$$2 = N_{in} - 1 - F_{in} \tag{5.28}$$

for the input,

$$2 = N_{act} - F_{act} \tag{5.29}$$

for activations as they contains only non-negative values and finally

$$1 = N_{out} - 1 - F_{out} \tag{5.30}$$

for the output array.

Now we can experimentally search for the most optimal bit-width N for each of the layers through the set and evaluate process. It has been started with each $N_{in} = N_{act} = N_{out} = 9$, as the *8-bit* quantisation of input, output and activations has failed in the TFLite quantisation case. The summary of attempted configurations, with the definition of their identifiers used later in the discussion, is shown in Table 5.6.

The results obtained for selected configurations are presented and discussed in the next section, together with a comparison against the full-resolution model as well as the one with only weights quantised.

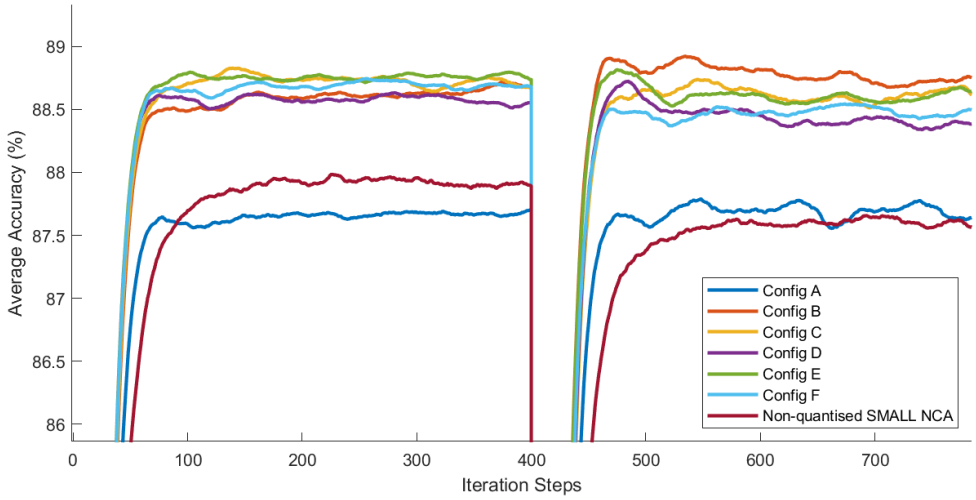
5.2.5 Final performance of modified incremental quantisation on SMALL NCA

We started the search for the optimal quantisation of input, output, and activations with the value of $N = 9$, as *8 bits* were not enough for every model in the TFLite scheme. We present selected configurations in Table 5.6, detailing a bit-width N and scale exponent F. Obtained performance for each configuration is displayed in Figure 5.9, with legend pointing to the corresponding configuration from the table.

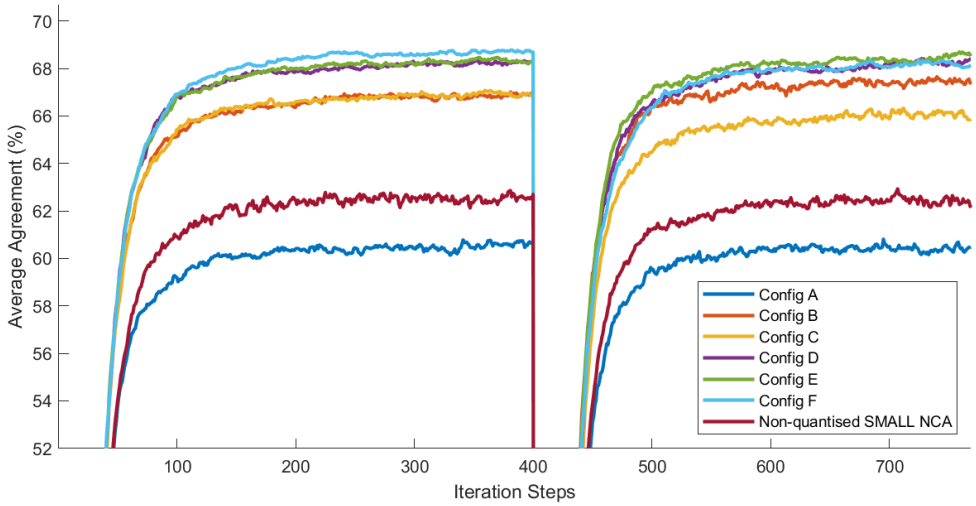
Table 5.6: Details of the selected input, output and activations quantisation configurations for the full, incremental quantisation of SMALL NCA. The obtained performances are presented in Figure 5.9.

Configuration	Input N	Input F	Activations N	Activations F	Output N	Output F
A	9	6	9	7	9	7
B	10	7	9	7	9	7
C	10	7	10	8	10	8
D	11	8	10	8	10	8
E	11	8	11	9	11	9
F	12	9	12	10	12	10

Similar to the TFLite quantisation, as the input scale varies with the value of F: $S = 2^{-F}$, the floating-point images theoretically should be scaled to the appropriate integer value, according to the Equation 5.1. However, just like in the case of TFLite quantisation, that the operation works very well with the standard *8 bit* image representation in range(0,255). Every configuration,



(a) Average Accuracy (%) - y-axis value starts at 82%.



(b) Average Agreement (%) - y-axis value starts at 50%.

Figure 5.9: The final performance of fully quantised SMALL NCA for different bit-width configurations, and non-quantised model (red) for comparison. Details of each configuration are shown in Table 5.6.

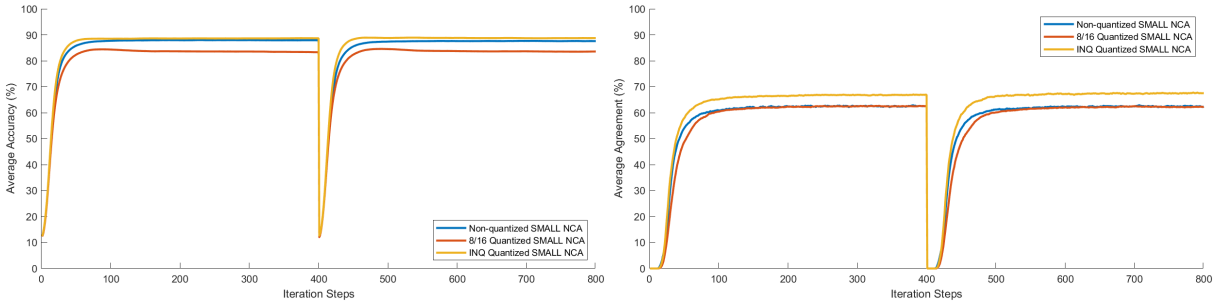
except "A", has achieved quite similar performance, especially in terms of accuracy. Amongst the presented, the configuration "B" uses the least memory, in terms of bits, and hence was chosen as the final one for the implementation. However, if the resulting behaviour on the device is not satisfying, change to another configuration requires only changing the scaling parameters between the layers.

5.3 Comparison of the Quantisation Approaches

5.3.1 SMALL NCA quantisation results

In Figure 5.10 we introduce a final comparison of the quantisations performance, presenting the 8/16 TFLite quantisation and the incremental quantisation according to the "Config B", and comparing them to the base SMALL NCA model.

The incremental quantisation performs better than the TFLite counterpart thanks to the incorporation of re-training, which boosts the model's performance and also allows it to adapt



(a) Average Accuracy (%) - y-axis value starts at 0%

(b) Average Agreement (%) - y-axis value starts at 0%

Figure 5.10: Final performance comparison of the TFLite 8/16 quantization and selected configuration (Config B) of the INQ quantization, with the base SMALL NCA model.

to the applied quantisation approximations. However, if the TFLite quantisation were to be implemented on the model re-trained three or four times it could match the presented performance of the incremental approach. The demonstrated quantised models are selected for the implementation in the SCAMP5 simulator.

5.3.2 Reduction of the model size and computation complexity

Model Weights

Initially, every weight/parameter of the model is expressed as a floating-point number with *32 bits*. In the 8/16 TFLite quantisation the weights were quantised to the *8 bit* representation, while with the incremental approach we were able to reduce that number to *5 bits*, which means that model size was reduced by a factor of 4 and 6.4 respectively.

The significance of the bit-width in the final weights representation is relevant to the devices with and without multiplication operation, both when they are coded with instructions or stored. When the weights are hardcoded for the device without the multiply instruction, such as SCAMP5, the length of the resulting kernel code generated by a compiler like CAIN, will directly depend on the complexity of the kernel and magnitude of its weights, as the multiplication is to be converted to repeated additions. Hence, with the smaller bit-width, the kernel compiler will generally produce shorter code and so decrease the latency. In a scenario with the weights stored in memory, the bit-width simply translates to the used space and the less we use the better.

The bit-width of weights quantisation between TFLite and INQ approach differs only by 3 bits, however, it translates to a significant difference in possible weight magnitude. In the 8/16 TFLite quantisation the weights are represented as *int16* and hence can be in range (-128,127). On a contrary, the *5 bits* used for weights in the modified INQ approach, gives possible integer values in the range (-15,15), they were obtained using sign-bit representation, not the two's complement. With such a large difference, the INQ quantised model will result in a much faster and efficient device code. We provide the details of the execution time and code length for both TFLite and INQ quantisation in Section 6, illustrating the significance of this property.

Input, Output and Activations

The quantisation of the input/state vector, as well as intermediate results/activations, determines the size of the storage required to facilitate the network's computation. We have presented in Table 4.3 the lower bound for the number of registers required by the SMALL NCA and the representation of input and activations will set their minimal size.

Computation of the neural network is performed by first multiplying the input values with the corresponding weights and then summing the resulting products respectively for each output channel. If we denote the bit-width of input value as A and bit-width of weights as B, the maximum possible absolute value of their multiplication is:

$$2^A \cdot 2^B = 2^{A+B} \tag{5.31}$$

Then we need to sum $C_{in} \times H_{kernel} \times W_{kernel}$ such values for a single output channel, where C_{in} is the number of channels in input, as convolutions extend to the entire depth. So the maximum value in the channel sum, before it is scaled down, can be written as

$$2^{A+B} \cdot (C_{in} \cdot H_{kernel} \cdot W_{kernel}) \rightarrow 2^{A+B+1} \tag{5.32}$$

because the value of product $C_{in} \times H_{kernel} \times W_{kernel}$, also if we include the multiplication by M_0, is usually smaller than 2^{A+B}.

However, the channel sum is then always scaled-down back to its standard representation. Consequently, we only need two large registers, the first one to store the immediate multiplication result and the second one to accumulate them. Once we have added all the relevant products, we scale down the resulting value to its standard bit-width and can safely move it to its output register. In the INQ quantised model, this scaling process requires only right-shifting the channel sum, while in the TFLite it has to be first multiplied by derived M_0, and then shifted. Additional multiplication necessary for TFLite adds to the code length and complexity, increasing latency.

Total size of required memory

We provide the final summary of the required memory by quantised SMALL NCA models in Table 5.7. We report the total size in bits only, as the bit widths used by the INQ approach do not correspond to the standard number formats and this is where the difference between the TFLite 8/16 approach exhibits.

Table 5.7: Summary of the necessary memory size for the quantised SMALL NCA model for two implemented quantization, 8/16 TFLite Quantisation and Incremental Quantisation with Configuration D. Here we refer only to the total number of bits necessary.

Quantisation	Bit-width of weights	Bit-width of state vector/input	Bit-width of activations	Bit-width of update vector/output	Bit-width of multiplication result	Bit-width of channel sum
8/16 TFLite	8	16	16	16	24	25
INQ Config D	5	11	10	10	16	17
Number of required registers	None	15	80	None	8	1
Total size of required memory in bits		8/16 TFLite Quantisation 1737		Incremental Quantisation - Config D 1110		

If we were to use the standard bit widths only, each register for the INQ quantization would have to be *16 bit*, as it could be argued that in the vast majority of cases the *16 bit* will also be enough to express channel sum. On the contrary, for the 8/16 TFLite quantization, nine of the registers, those for multiplications results and channel sum, would need to be *32 bit* wide.

5.3.3 Summary

We have presented step by step application of two different quantisation approaches that could be applied to the Neural Cellular Automata networks.

The TFLite quantisation has benefits of relative simplicity if it comes just to quantisation of the model, but the process of preparing the resulting quantised network for the implementation on

the FPSP device is very involving and requires an empirical search for the best configuration of parameters. The post-training approach that we have used here allows for the usage of two quantisation configurations only, from whom the 8/8 one proved to be insufficient for every attempted model. Consequently, the only viable, 8/16 quantisation introduces large over-allocation and uses much more memory than necessary. It also results in the convolutional kernel weights as well as the scale constants M_0 being relatively large, which results in longer device code on the devices that do not have multiplication instruction.

The presented modified version of the INQ method allows for great flexibility in the choice of quantisation bit widths and hence allows to optimise the model size reduction to the maximum. However, to take the advantage of the non-standard bit widths the actual implementation would need to be more complex and treat each value accordingly to its type, especially if the used bit widths vary between the layers. Reducing the number of bits that are used has also the benefit of reducing the largest possible magnitude of values, which helps to create efficient and fast code when no multiplication operation is available. On the other hand, this quantisation process is even more engaging and time-consuming as it involves repeated re-training which is then followed by the empirical search for the most efficient quantisation of the input and activations.

Finally, to obtain the best compression and reduction of the NCA model it must be stable, having a limited range of possible state magnitudes. The smallest the range of values that needs to be represented is the fewer bits would have to be used to provide precise enough scale.

Chapter 6

Automaton Implementation and Evaluation

6.1 Implementation on the Simulator

Once we have all the weights in the integer format, as well as the scaling parameters, the NCA could be implemented in the simulator of the SCAMP5 architecture. In our implementation, we follow the instruction set of the SCAMP5 device, with a single exception as we have added generating a random global mask from the uniform distribution, as it is necessary for the stochastic state updates. We also add an experimental multiplication instruction, although we do not specify its cycles count, leaving it as a parameter.

The platform of the implementation is the FPSP device simulator [33], already described in section 3.6, which we configure the create a device with enough registers to accommodate the SMALL NCA computation, according to Table 4.3. Every register is created in the simulator as an analogue one for code compliance, although we are relying on the discrete computation and hence are not simulating the noise related to analogue arithmetic. Based on the device size as well as the number of registers specified in the configuration file, the simulator provides physical characteristics for a hypothetical device, such as transistor count, chip and singe processing element size, and area, which are independent of the running program. We provide the comparison of those metrics, collected for a base configuration that imitates the actual SCAMP5 register suite and the running NCA configuration, in Table 6.1. We can see that for the NCA registers configuration the transistor count is 7.35 times larger than in the actual SCAMP5 device of respective array size. At the same time, the increase in size along both axes is around 3. However, it is important to note that those metrics are measured assuming the SCAMP5-like architecture,

Table 6.1: Physical characteristic of the hypothetical SCAMP5-like architecture with registers configuration for the NCA, compared with the actual SCAMP5 device. Obtained using FPSP Simulator [33].

Registers Configuration	Pixel Array Size	Transistor Count	Single Processing Element Dimensions	Single Processing Element Area	Total Chip Dimensions
Existing SCAMP5	28 x 28 256 x 256	92512 7733248	33µm × 30µm	$9.9 \cdot 10^{-4}$ mm	1.016mm × 0.924mm 9.292mm × 8.448mm
NCA	28 x 28 256 x 256	679728 56819712	99µm × 99µm	$9.8 \cdot 10^{-3}$ mm	3.049mm × 3.049mm 27.88mm × 27.88mm

which means for example that the added computation registers are considered to be analogue, but as we argued before the efficient computation of NCA requires fully digital computation. Consequently, the presented metrics should be treated rather as an indication of possible growth of the chip size of we would expand register suite of each pixel to **103** as per Table 4.3, but not as a definite measurement.

For the generation of the device code with have used CAIN, including the 1x1 convolutional layers. The general process for implementing the NCA in the SCAMP5 code is presented in Algorithm 4.

Algorithm 4 SCAMP5 NCA implementation loop

Define N as the state vector size and M as the required number of intermediate registers
Define state registers $S_1, S_2, \ldots, S_N$
Define first set of intermediate output registers $T_1, T_2, \ldots, T_M$
Define register for single kernel output A_o and for channel summation A_s
Define digital register to store binary mask D
while True **do**
    Read new image frame
    Sample from uniform distribution and create a new mask in D
    Reset the state registers S where pixel intensity is below alive threshold
    for $L = 1, 2, \ldots, NetworkLayers$ **do**
        for $i = 1, 2, \ldots, LayerOutputChannels$ **do**
            Reset A_s
            for $j = 1, 2, \ldots, LayerInputChannels$ **do**
                Reset A_o
                Compute kernel result $\rightarrow A_o$ ▷ Use respective input register ▷ Either state or intermediate one
                Add result : $A_s \leftarrow A_s + A_o$
            end for
            Multiply A_s by respective M_0 : $A_s \leftarrow A_s * M_0$
            Right shift A_s by the respective number of bits : $A_s \leftarrow RightShift(A_s, n)$
            if Not A Last layer **then**
                Apply ReLU function on the value in A_s
                Store channel result in a free output register : $T_k \leftarrow A_s$
            else
                Reset A_s where R is True
                Reset A_s where pixel intensity is below alive threshold
                Add the value in A_s to the corresponding state register S : $S_l \leftarrow S_l + A_s$
            end if
        end for
    end for
    Determine prediction label for each pixel
end while

It translates the original inference process to the device limitations and specifics. We have opted to calculate every kernel separately to minimize the number of larger registers A_o and A_s, and in this approach, we just need one of each. The approach is simple, calculate the kernel's result into A_o, add it to A_s and repeat until we have iterated through each input channel. Then, scale accumulated output channel result and move it to its storage register, applying ReLU activation beforehand. If it is the last layer, we instead reset the computed update where the uniform mask is true and also where the pixels are "DEAD" and can finally increment the respective element

of the state vector. For the classification code, we have opted to use ten available digital registers per pixel and assigned them to respective labels. By going over every prediction produced by a pixel, we set the register that corresponds to the largest observed value.

We have implemented three NCA models discussed above, the SMALL NCA versions obtained with 8/16 TFLite quantization and with INQ quantization as well as the 8/16 TFLite quantized DEEP NCA.

6.2 Performance evaluation

Each of the implemented automata performs very well in the simulator, achieving a behaviour expected from the experiments performed in Python. We present the execution statistics for each of the implementations in Table 6.2, showing the implementation without multiplication instruction and the one where scaling and 1x1 convolutions were done using multiplications. In each of those, the 3x3 convolutions are implemented with CAIN as it optimises generated code well and handles necessary neighbour access. As said before, we have left the cycle count of the multiplication as a parameter and indicate it as "C".

Table 6.2: Execution statistics for the SMALL NCA and DEEP NCA implementations with the FPSP simulator. The presented device execution times are obtained by the simulator by counting the execution clock cycles and dividing a sum by the clock rate, 10^7Hz. The parameter "C" in the last column is the number of required clock cycles to perform a single multiplication.

NCA	Resolution	Iteration head instructions	Classification instructions	Neural network instructions without multiplications	Device Execution Time (ms) for 1 frame	Total Architecture Power (mW)	Neural network instructions with 1x1 convolutions using multiplications	Number of multiplication instructions	Device Execution Time (ms) for 1 frame
Incrementally Quantised SMALL NCA - Config B	28 x 28	39	116	27823	13.44	5.78	21579	2160	$8.93 + \frac{2160 \times C}{10^4}$
8/16 TFLite Quantised SMALL NCA	28 x 28	39	116	40874	20.19	5.54	25853	2254	$11.15 + \frac{2254 \times C}{10^4}$
8/16 TFLite Quantised DEEP NCA	28 x 28	39	116	84334	43.13	6.03	68123	2499	$34.34 + \frac{2499 \times C}{10^4}$

As presented, the incrementally quantised SMALL NCA provided the shortest device code, resulting in the shortest execution time. It demonstrates that thanks to the small magnitude of weights as well as scaling with bit-shift instructions only, the generated device code is indeed much more efficient than the TFLite quantised counterpart. The difference is especially significant when the scaling operations in 8/16 TFLite quantisation are implemented as addition code, but still exists when the multiplications are available. It comes mostly from the fact that the average length of the compiled 3x3 kernel in the first layer is **26.77**, compared to **21.26** in the case of the incrementally quantised model. With six hundreds of 3x3 kernels in the first layer, it gives a difference of **3900** instructions.

The DEEP NCA architecture contains one additional 3x3 convolutional layer and consequently, it is expected that its device code is significantly longer than the others.

From the presented data, we can approximate the final latency of each implemented automaton for the implementations that do not rely on the multiplication instruction. The best of the presented configurations, the Incrementally Quantised SMALL NCA, using configuration B - Table 5.6, has achieved an execution of a single frame in **13.44**ms giving a theoretical frames-per-second rate of **74 FPS**, according to the data obtained with the FPSP simulator [33]. It is significantly smaller than the MNIST classification networks presented in section 3.5, such as 2260 FPS achieved by AnalogNet. With the device power of **5.78 mW**, the cost of execution of one frame is **77.68 mJ**, which compared to the 0.7 mJ in AnalogNet, are a hundred times larger. It shows that just adding additional registers will not solve the issue of complexity and amount of computations, resulting in costly and relatively long execution.

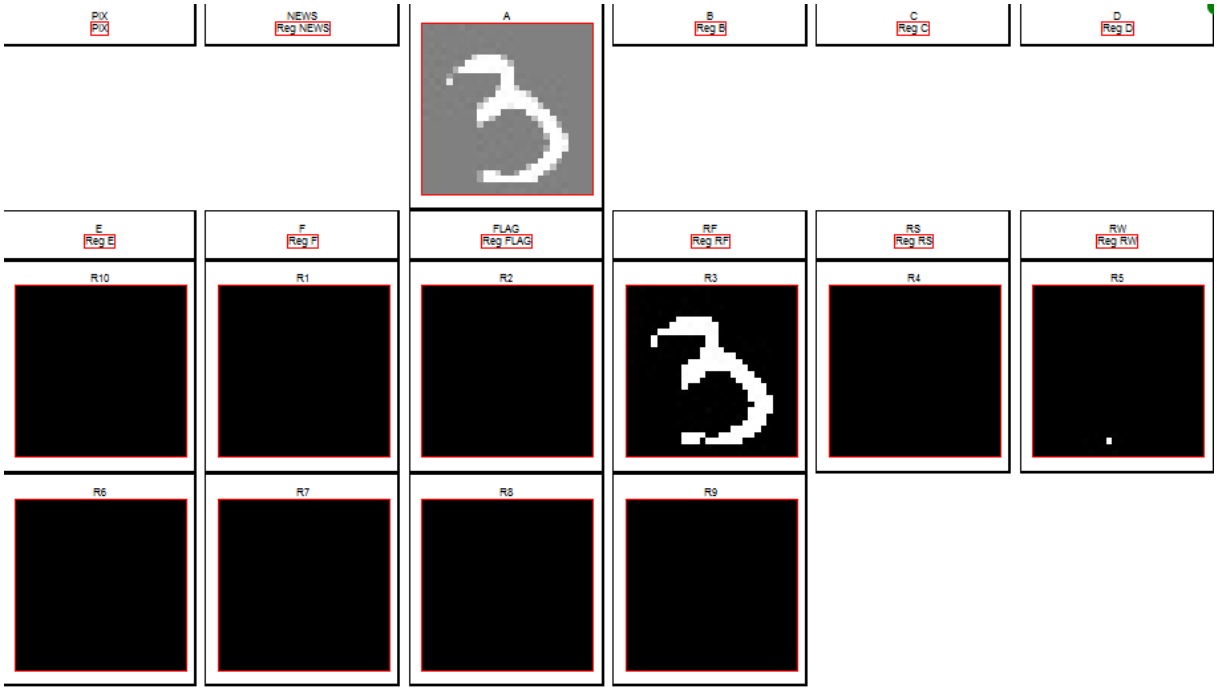


Figure 6.1: An example of the NCA output from the FPSP simulator. Register A shows the input image, while the registers R show the pixels that predict a label corresponding to the register's subscript, with exception of R10 being assigned to label 0. The presented example shows ready prediction, although one pixel is not in an agreement.

Example output from the simulator is shown in Figure 6.1 and the recorded demonstration video for the operation of each of the presented implementations is given with the submitted software archive, and also available here `https://youtube.com/playlist?list=PLJSdfwhgNpuEmsvTOPFFO-fAHVRfKVXOU`. The archive contains also a device code for presented configurations, with and without multiplication instruction, expressed as a *.txt* files and the generator code, Python notebook and library file, which was used to automate the process.

We have also experimented with an input video that contains moving digits to observe the automaton behaviour in a more realistic setting. The DEEP NCA has performed better in this situation as it gains confidence much quicker than the SMALL NCA, as presented in Figure 4.1. The SMALL NCA models require more iterations to propagate the messages between the individual cells. However, the automaton still requires an input image to be stable for a certain amount of iterations, and too frequent movement of the input digit results in the cells changing frequently from "ALIVE" to "DEAD" and hence resetting their values, preventing the progress of computation. In Algorithm 4 we read the new image frame every iteration, however, to solve the above issue, the new image frame could be read every few or more iterations so that the automaton can evolve. The reduction of the number of processed frames could be ignorable if the overall operation latency is low enough.

6.3 Possible further improvements

The achieved latency is not final and can be further improved, for example by the addition of the efficient multiplier per pixel which could perform the integer multiplication in a small number of cycles, to reduce the execution time even further. With such a multiplier present, there might also be a benefit in the implementation of the 3x3 convolutional layers, however, it is

not guaranteed, especially for kernels with small weights, as the CAIN compiler highly optimises common subexpressions. Speaking of CAIN, the compilation of the convolutional kernels could also be improved, as in the presented result we have compiled each 3x3 kernel separately, not utilising its full potential. By compiling a few kernels at the time, CAIN could exploit their similarities and shorten the resulting code, at the cost of using more intermediate output Ao and channel summation As registers. A pair of such registers is required for every channel that is currently in computation.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this work, we have presented an end-to-end implementation of the modified Neural Cellular Automata for the self-classifying MNIST digits by Randazzo et al. [36] on the SCAMP5 device simulator. We have covered all the necessary steps, starting with an analysis of the SCAMP5 limitations and development of the optimised architecture that moderates their impact, which we then quantise using two alternative methods resulting in the full-integer NCA models that are successfully implemented in the SCAMP5 simulator.

In particular, we have discussed the issue of the computational complexity of the Neural Cellular Automaton, which arises from the necessity of neural network calculation at each cell that effectively replicates costly operation over the entire FPSP device array. We have proposed that a viable solution is the introduction of efficient path-multiplexing, with access to local and global shared memory. This would also answer the problem of limited per-pixel memory, enabling the device to hold more complex automata. We have proposed an optimised self-classifying MNIST digits NCA that limits the number of parameters three times and reduces required storage twice compared to the original MNIST NCA.

The analysis of the long-term stability problem on the exemplary network has resulted in obtaining a successful solution using the Batch Normalization for our exemplary DEEP NCA network, although it does not generalise to all network configurations.

We have presented a step-by-step application of two alternative quantisation approaches, forming a procedure that can be followed to convert any future NCA architectures for the FPSP implementation, with our customized incremental quantisation we have achieved a reduction of a 32-bit floating-point model into 5-bit weights and 9/10-bit activations, while maintaining the original performance. Our SCAMP5 simulator implementation of that model has achieved a processing speed of **74 frames per second**.

7.2 Future Work

Our analysis of the Neural Cellular Automata has revolved around the self-classifying MNIST digits application and evolutions to the existing model in terms of the FPSP implementation. However, the MNIST recognition task is the most fundamental benchmark achieved by relatively simple architectures, and one of the prospective development directions is to apply it in new situations. Research in this area is not necessarily tied up to the vision applications on the FPSP

devices and focal plane and can be extended as the analysis of cellular automata as a general computational algorithm for the organised grids.

In addition, future work could also target the optimisation towards the processing of moving input images, so that pixel shifts are causing instantaneous forgetting of data in the cells that turned from "ALIVE" to "DEAD" and experiment with a mechanism of graduated forgetting. Also, the automaton was executed on a very small array of 28x28 cells/pixels and future work could investigate how it performs on larger resolution and how it affects the convergence speed and overall accuracy.

The long-term stability of the neural cellular automata is a crucial property that is necessary for efficient and accurate use in any application. Our analysis was situational and treated a specific architecture, but this work could be further expanded by relating to the research on the stability of recurrent neural networks and formulating a general theory and development guidelines. Finally, we have not experimented with the resettable automaton cells, proposed and successfully applied by Sandler et al. [40] in the image segmentation NCA, which could be a general solution to this issue.

Presented step-by-step procedures of the quantisation were relying on the manual conversion of scaling parameters and experimental search for the most suitable configurations, which could be possibly automated and integrated with an end-to-end FPSP device code generation. It would require developing a configurable search that can deal with the arbitrary convolutional networks, not only the neural cellular automata, and select the best quantisation configuration according to the specified heuristic. The device code generator could use a compiler such as CAIN to compile convolutional kernels and multiplication instructions (if available) to search for the most efficient program.

7.3 Thoughts on the further development of the FPSP architectures

The implementation of the self-classifying MNIST automaton in the SCAMP5 simulator has highlighted a set of limitations the currently prevent the use of any existing architecture and point out the necessary improvements, that have been described in this thesis.

The main issue with the cellular automata on the FPSP devices is the complexity and volume of computation that needs to be performed by every single cell in the grid. Each of them has to individually execute forward computation of the neural network on its neighbourhood which is relatively complex, when compared to the processing element capabilities, even for such a simple application as recognition of MNIST digits. Consequently, if we want to execute the NCA using each cell of the FPSP device, we end up replicating costly operations on the simple processing elements that have to execute ten thousand instructions even for a very small architecture. Effectively, one of the main advantages of those devices, large scale parallelism is suppressed and each pixel is overwhelmed both in terms of computation as well as a required storage size and becomes a bottleneck.

Naturally, the simplest approach could be to improve the capabilities of individual processing elements, providing them with more resources. However, the necessary increase in the registers is enormous if we compare seven analogue registers available in SCAMP5 with over the hundred need to carry out SMALL NCA computation. Needed to mention that each of them should be digital and larger than 8 bit, which effectively may conflict with the chip space limitations.

Another possible direction is to restore some parallelism in such a situation could be introducing the support for no-overhead computation in the superpixel setup or patch-multiplexing. Existing

implementations use this configuration to combine binary registers into n-bit digital registers but at the cost of a substantial number of additional instructions that arise from more complex procedures for neighbour data access, need to access superpixel data stored on different individual pixels and no support for digital arithmetics. To solve the local memory issue, we propose that the individual pixels in the focal plane could be arranged physically into groups/superpixels, that has a shared memory cell allowing each member of a superpixel to access the superpixels data in constant time. This memory cell hopefully could be large enough to fit the storage requirements of the single automaton cell, enabling the execution on the focal plane although in reduced resolution. It could be configurable so that in standard execution each pixel has assigned an equal part of this "local" memory and directly connected with other such cells effectively creating a grid of "local" memory cells, which would enable constant direct access to the neighbours' data in the superpixel configuration.

Now, we need to enable each pixel in the superpixel to compute different features in parallel, which because the device is SIMD (Same-Instruction Multiple-Data), means that the network's weights need to be stored in memory instead of being hardcoded. This is perfectly possible for a few kernels at the same time, but not for the entire network, as even the simple SMALL NCA contains 7.5k of parameters, which translates to a very large storage requirement even with the weights in 5-bit representation which we have achieved. Consequently, because the weights are the same for every pixel/superpixel in the focal plane they could be stored in the separate, "global" memory that is read-only and accessible efficiently by every pixel/superpixel so that they can be loaded in parts as the computation progresses.

Finally, as every storage needs to be digital to eliminate computational losses and maintain accuracy, each pixel would benefit from having an efficient integer multiplier. The above discussion shows a possible direction for the further development of the devices such as SCAMP5 that would answer the computational need of neural cellular automata, however, implementation of the proposed elements will increase the architecture complexity and might result in new problems of its own.

7.4 Legal, Social, Ethical and Professional Requirements

The goal of this work is to produce openly available research on the implementation of the neural cellular automata on the family of FPSP devices, presenting the limitations of the existing architectures and providing guidelines for the development of the next-generation devices. For the benefit of future work obtained results need to be presented truthfully and provided procedures and algorithms, as well as submitted code and data, should allow for the reproduction of achieved results.

This project does not target any potential military applications, however, as it aims to support the development of the next iterations of FPSP visions chips it can be potentially used in military vehicles and systems that rely on the vision sensors, such as autonomous drones or vehicles. Similar analysis refers to civilian applications and the produced work might be used in the surveillance systems that violate privacy and are used in a malicious way such as facial recognition or data theft.

Bibliography

[1] FlatBuffers: Memory efficient serialization library. URL `https://google.github.io/flatbuffers/`. Google Open Source.

[2] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[4] Laurie Bose, Jianing Chen, Stephen Carey, Piotr Dudek, and Walterio Mayol-Cuevas. Visual odometry for pixel processor arrays. pages 4614–4622, 10 2017. doi: 10.1109/ICCV.2017.493.

[5] Laurie Bose, Jianing Chen, Stephen J. Carey, Piotr Dudek, and Walterio W. Mayol-Cuevas. A camera that cnns: Towards embedded neural networks on pixel processor arrays. *CoRR*, abs/1909.05647, 2019. URL `http://arxiv.org/abs/1909.05647`.

[6] Laurie Bose, Jianing Chen, Stephen J. Carey, Piotr Dudek, and Walterio W. Mayol-Cuevas. Fully embedding fast convolutional networks on pixel processor arrays. *CoRR*, abs/2004.12525, 2020. URL `https://arxiv.org/abs/2004.12525`.

[7] S.J. Carey, A. Lopich, David Barr, Bin Wang, and Piotr Dudek. A 100,000 fps vision sensor with embedded 535gops/w 256×256 simd processor array. pages C182–C183, 01 2013. ISBN 978-1-4673-5531-5.

[8] Jianing Chen, S. Carey, and P. Dudek. Feature extraction using a portable vision system. 2017.

[9] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations, 2016.

[10] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

[12] A. Deutsch, J. M. Nava-Sedeño, S. Syga, and H. Hatzikirou. Bio-lgca: A cellular automaton modelling class for analysing collective cell migration. *PLoS computational biology*, 2021. doi: https://doi.org/10.1371/journal.pcbi.1009066.

[13] Martin Gardner. The fantastic combinations of John Conway's new solitaire game life. *Mathematical Games. Scientific American.*, 223(4):120–123, October 1970. doi: 10.1038/scientificamerican1070-120.

[14] William Gilpin. Cellular automata as convolutional neural networks. *Physical Review*

E, 100(3), Sep 2019. ISSN 2470-0053. doi: 10.1103/physreve.100.032402. URL http://dx.doi.org/10.1103/PhysRevE.100.032402.

[15] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns, 2016.

[16] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks, 2015.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[18] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations, 2016.

[19] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

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

[21] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[22] James N. Knight. *Stability analysis of recurrent neural networks with applications*. PhD thesis, Colorado State University, 2008.

[23] T. Komuro, I. Ishii, and M. Ishikawa. Vision chip architecture using general-purpose processing elements for 1 ms vision system. In *Proceedings Fourth IEEE International Workshop on Computer Architecture for Machine Perception. CAMP'97*, pages 276–279, 1997. doi: 10.1109/CAMP.1997.632052.

[24] Takashi Komuro and Masatoshi Ishikawa. 64×64 pixels general purpose digital vision chip. pages 15–26, 01 2001. ISBN 978-1-4757-6530-4. doi: 10.1007/978-0-387-35597-9_2.

[25] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). URL http://www.cs.toronto.edu/~kriz/cifar.html.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[27] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[28] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

[29] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks, 2016.

[30] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications, 2016.

[31] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation, 2016.

[32] Alexander Mordvintsev, Ettore Randazzo, Eyvind Niklasson, and Michael Levin. Growing neural cellular automata. *Distill*, 2020. doi: 10.23915/distill.00023. URL `https://distill.pub/2020/growing-ca`.

[33] Jamal Mulla. Focal plane sensor processor simulator, June 2021. URL `https://github.com/JamalMulla/JSS`.

[34] Riku Murai, Sajad Saeedi, and Paul H. J. Kelly. BIT-VO: visual odometry at 300 FPS using binary features from the focal plane. *CoRR*, abs/2004.11186, 2020. URL `https://arxiv.org/abs/2004.11186`.

[35] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[36] Ettore Randazzo, Alexander Mordvintsev, Eyvind Niklasson, Michael Levin, and Sam Greydanus. Self-classifying mnist digits. *Distill*, 2020. doi: 10.23915/distill.00027.002. URL `https://distill.pub/2020/selforg/mnist`.

[37] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks, 2016.

[38] Lutz Roeder. Netron. A viewer for neural network, deep learning and machine learning models., August 2021. URL `https://github.com/lutzroeder/netron`. Release 5.1.0.

[39] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. ISSN 0033-295X. doi: 10.1037/h0042519. URL `http://dx.doi.org/10.1037/h0042519`.

[40] Mark Sandler, Andrey Zhmoginov, Liangcheng Luo, Alexander Mordvintsev, Ettore Randazzo, and Blaise Agúera y Arcas. Image segmentation via cellular automata, 2020.

[41] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[42] Edward Stow, Riku Murai, Sajad Saeedi, and Paul H. J. Kelly. Cain: Automatic code generation for simultaneous convolutional kernels on focal-plane sensor-processors, 2021.

[43] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[44] TensorFlow. TensorFlow Lite. Set of tools that enables on-device machine learning on embedded devices. URL `https://www.tensorflow.org/lite/guide`.

[45] *TensorFlow Lite Guide*. TensorFlow. URL `https://www.tensorflow.org/lite/guide`. Framework's Guide available from tensorflow.org.

[46] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization, 2017.

[47] Matthew Z. Wong, Benoit Guillard, Riku Murai, Sajad Saeedi, and Paul H. J. Kelly. Analognet: Convolutional neural network inference on analog focal plane sensor processors, 2020.

[48] Yuxin Wu and Kaiming He. Group normalization, 2018.

[49] N. H. Wulff and J. A. Hertz. Learning cellular automaton dynamics with neural networks. In *Proceedings of the 5th International Conference on Neural Information Processing Systems*,

NIPS'92, page 631–638, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 1558602747.

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