POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Thesis

In-Memory Binary Neural Networks

Supervisors:
Prof. Maurizio Zamboni
Prof. Mariagrazia Graziano
Prof. Marco Vacca

Candidate:
Andrea Coluccio

April 10, 2019

Acknowledgments

I would like to thank all the people who made possible this course of studies.

A special and most important recognition goes to my parents, who gave me the opportunity to face these academic years with peace of mind, giving me all the support I needed.

I also thank my girlfriend Martina for having been always close, even in the most difficult moments and complicated choices. Thank you for supporting me everytime.

A special thank to my aunt Maria for her support.

I would like to express my gratitude to my high school Professor Anna Civarelli, who has always motivated me to do my best from the beginning of my scholastic carrier until now. She played an important role in my education, allowing me to develop a strong interest in Electronics.

I am grateful to the Politecnico di Torino who provided me with the means to fulfill myself in my field of interest; in particular, I thank my mentors Prof. Maurizio Zamboni, Prof. Mariagrazia Graziano and Prof. Marco Vacca who have encouraged me to give the best of me to achieve this important thesis.

Lastly, I address a thought of thanks to all those who have supported me over the years.

Sincerely,
Andrea Coluccio

Turin, April 10 2019.

Glossary

1T1R One transistor, one resistor: a memory cells' implementation used in RRAM to isolate the current of the selected cell from the others. 11, 51, 54, 58, 93, 94

ACCA Accumulation array. 107

AlexNet AlexNet is a convolutional neural network, which competed in the ImageNet Large Scale Visual Recognition Challenge in 2012. The network achieved a top-5 error of 15.3% [1].. 9, 11, 12, 23, 24, 28, 32, 48–50, 101, 109–116, 122, 124, 125, 250

BCNN Binary convolutional neural network. 48, 49, 102, 103

BNN Binary neural network. 11, 13, 61, 70, 71

CIFAR-10 The CIFAR-10 dataset (Canadian Institute For Advanced Research) is a collection of images. The CIFAR-10 dataset contains 60000 32x32x3 images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. [2]. 12, 25, 32, 34, 38, 39, 58, 60, 62, 103, 112

CIM Computation in memory. 45, 49, 126

CNN Convolutional neural network. 7, 9–11, 14, 23, 27–29, 35, 37, 39, 45, 49, 52, 53, 58, 62, 74, 75, 82, 95, 99, 101, 102, 107, 108, 113, 119, 121, 123, 128, 137

DPU Digital processing unit: a separated unit (external from memory) used to perform computations which are not executable in-memory. 49

DW Domain wall (magnetic). 50, 51, 53

FMEM Filter memory. 107

IFMAP Input feature map. 7, 8, 10, 62, 91, 98, 99, 101, 105, 108, 133, 134, 136, 201, 203

ImageNet The ImageNet project is a large visual database designed for use in visual object recognition software research. It contains about 14 million images [3]. 6, 10–12, 23, 27, 28, 32, 39, 49, 110–112

IMEM Image memory. 107

IPNE Input parallel neural engine, inputs are in parallel, while outputs are delivered in serial. Output of this configuration is compatible with OPNE's input. 12, 72–74, 115, 118–120, 123, 126–130

ISU Input feature map summation unit. 107

LeNet LeNet is a type of convolutional neural network. 86, 90, 91, 116

MLC Multi level cell, more then one bit can be hold into a single cell. 11, 12, 41, 42, 55–57, 94, 115, 118–120, 122, 126, 130

MLCS Memory logic conjugated system. 75–78

MLP Multilayer perceptron is a class of artificial neural network. Each node is a neuron that uses a nonlinear activation function, except for the inputs. MLP uses backpropagation for training. [4]. 8, 9, 12, 14, 23, 58, 60, 61, 66, 67, 86, 113, 115, 121–123, 134, 137, 146, 235, 236, 246

MLSA Multi level sense amplifier. 60–62, 123

MNIST The MNIST database (Modified National Institute of Standards and Technology database) is a dataset of handwritten digits with 60000 images in B/W. [5]. 13, 19, 34, 45, 58, 60–62, 66–68, 75, 81, 86, 91, 119, 131, 133, 138, 139, 201, 235

MRAM Magnetoresistive random-access memory (MRAM) is a non-volatile random-access memory technology. Data in MRAM is not stored as electric charge or current flows, but by magnetic storage elements. 6, 10, 41, 42, 46, 47

MSC Modified sensing circuit, designed for logic and full-add operations. 10, 42, 43, 45

MTJ Magnetic Tunnel Junction is a component composed by two ferromagnets separated by an insulator. Electrons can tunnel from one ferromagnet into the other.[6]. 6, 10, 41–43, 45, 47, 48, 50–53, 93, 117, 119, 120, 126, 129

NDP Near Data Processing. 95

NPU Neuron processing unit. 106, 107

NVM Non-volatile memory. 46, 92, 93

OFMAP Output feature map. 7, 8, 10, 37, 46, 62, 75, 91, 105, 132, 133, 136

OOM Out of memory implementation.. 7, 8, 16–20, 87, 117, 121, 127, 148, 156, 171, 174, 188, 192, 205–208, 210, 211, 214, 218, 219, 221–223, 241–250, 252–254

OPNE Output parallel neural engine, inputs are in serial, while outputs are delivered in parallel. Output of this configuration is compatible with IPNE's input. 12, 72–74, 115, 118–120, 123, 126–130

PIM Processing in memory module: it is formed by the combination of an OPNE and an IPNE. 74, 115, 123, 128

PU Processing unit. 105

ReLU Rectified linear unit, a type of neuron's activation function which consists into $ReLU(x) = max(0,x)$. In terms of training time, it is the best choice.. 24, 25, 34, 102–104, 106

RRAM Resistive switching random access memory. 6, 7, 10–12, 54–58, 61, 67, 69, 92–94, 115, 117, 123, 124, 126–130

SC Stochastic computing. 87, 88, 91

SCT Synapse configuration table. 70–72

SGD Stochastic gradient descent method. 37, 66, 67, 133

SOT Spin-orbit torque: a type of magnetic RAM. 6, 10, 12, 46, 47, 50, 115, 118, 120, 122, 126, 128–130

stride stride, in the context of CNNs, is the distance between the receptive field centers of neighboring neurons in a kernel map. 8–10, 24–26, 35, 99

STT Spin-transfer torque is an effect in which the orientation of a magnetic layer in a MTJ can be modified using a spin-polarized current [7]. 6, 12, 41, 42, 50, 92, 115, 118, 119, 122, 126, 130

SVHN SVHN (Street View House Numbers) is a dataset. It consists in a training set of 604K and a test set of 26K 32x 32 color images representing digits ranging from 0 to 9.. 34

top-1 top-1 error is measured by checking if the top class (the one having the highest probability) is the same as the target label.. 10, 11, 25, 26, 39, 109, 110, 112

top-5 top-5 error is measured by checking if the target label is one of your top 5 predictions (the 5 ones with the highest probabilities).. 10, 12, 25, 26, 39, 111

Summary

In this thesis, an In-Memory architecture of a binary neural network is presented. The concept of ”In-Memory” is related to the possibility to place near-memory very simple computational units, such as logic gates or full-adders, to implement a distributed circuit instead of Von Neumann’s classical one. This choice brings to relevant benefits such as lower energy consumption/delay, since the computation is performed very close to memory, the wasted energy and the corresponding latency caused by the data fetching are heavily reduced, allowing an higher parallelization.

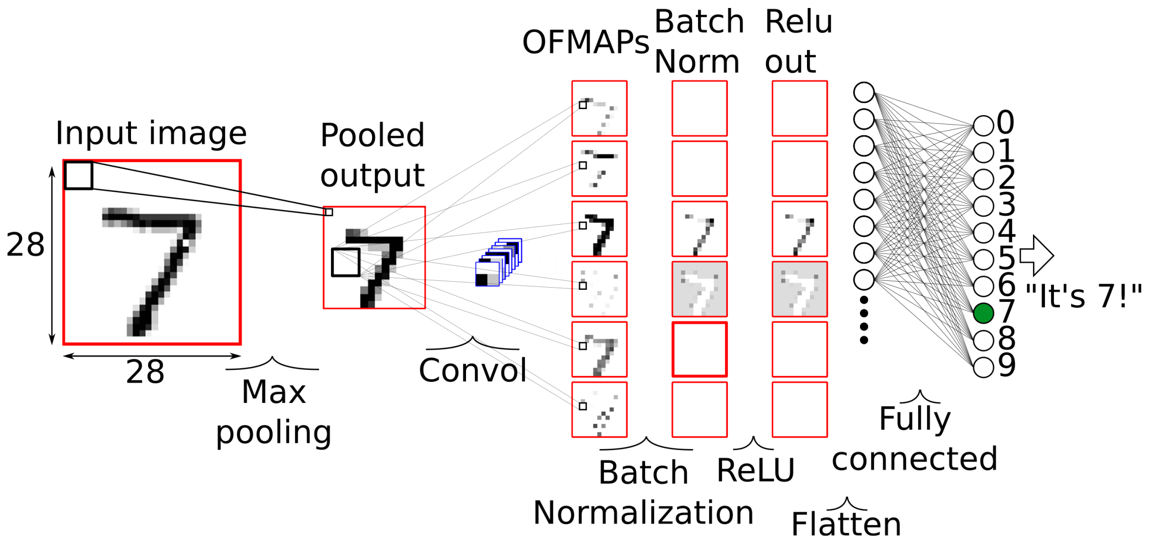


Figure 1: Convolutional neural network used as starting model. MNIST database is used, which is composed by handwritten digits in range $0 \div 9$.

As computational models, Convolutional Neural Networks (Figure 1) have been chosen. They are a class of neural networks that are able to recognize/classify raw data, such as images, sounds, natural language etc. The key parameters of a neural network are the number of layers and their dimensions, that influence the accuracy achievable and the usable dataset’s complexity. A ”binary” approximation called **XNOR Net** is considered, in which weights ($\mathbf{W}$) and inputs ($\mathbf{I}$) are binarized between $\{-1, +1\}$ by taking the sign, reducing the multiply-accumulate operations used in convolution into XNORs-popcounting sequences. The term ”pop-counting” refers to the following operation: number of 1s - number of 0s. The value computed is then multiplied by two scaling factors ($\mathbf{K}$ and α), obtaining the approximated convolution. This choice reduces memory required and computational cost, but degrades the achievable accuracy (from $\sim 97\%$ to $\sim 84\%$ for the model in Figure 1).

Architectures

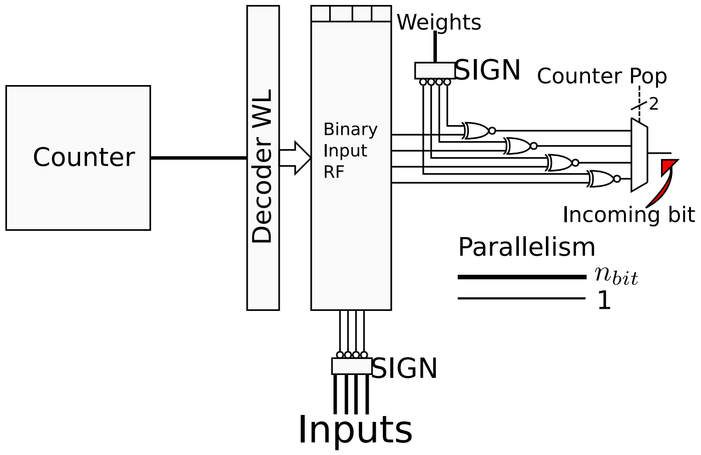


Figure 2: Classical implementation. In **Binary input RF**, the binary signs are precharged and then fetched one row per clock cycle to compute the XNORs. The incoming bit selected goes to pop-counting unit.

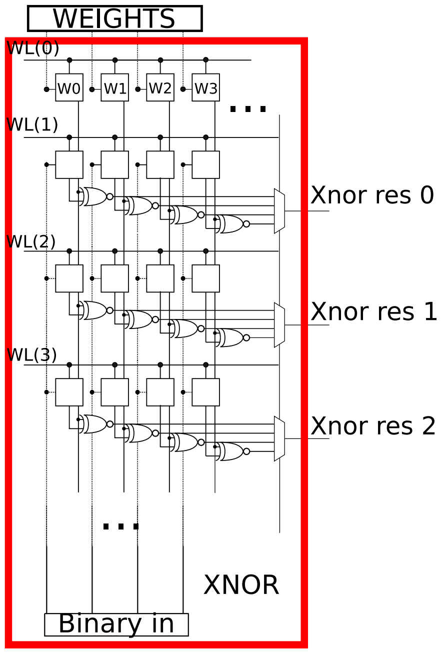


Figure 3: In-Memory implementation. Inputs are precharged into the memory cells and the XNOR gates perform the xnor operation between the binary weights (W_0,W_1,...) and inputs. Xnor results are then fetched from pop-counting parts.

Two architectures based on 45nm CMOS technology (In-Memory and classical implementations respectively), have been developed. The classical implementation has been used as reference architecture to compare the performance achieved in the In-Memory case. The computational model is well-suited for an In-Memory implementation, since XNOR gates and pop-counting circuits are very simple units that can be integrated into a memory array. In the classical implementation in Figure 2, a traditional memory has been used, in which data are simply stored and the computation is done out-of-memory (OOM). In the In-Memory alternative (Figure 3), the traditional structure has been replaced with a CAM-like array and the computation is perfomed inside the mesh by computing the xnors between binarized weights-inputs. One of the main advantage in the In-Memory alternative is the parallelization of the XNOR/pop-counting computations, which reduces the time required by the algorithm and the energy consumed.

Validation flow

Since the neural networks are often realized in software (for example Python with TensorFlow and Keras), a MATLAB model that computes both in floating point and fixed point representation has been carried out to convalidate the correctness of the VHDL implementation: when the floating point results are validated, the fixed point model is then verified, obtaining the validation flow depicted in Figure 4.

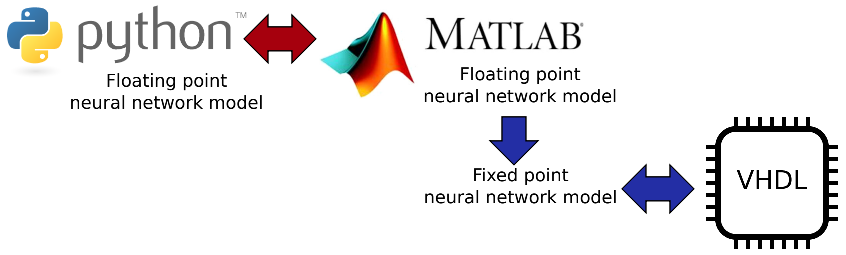

python
Floating point neural network model
MATLAB
Floating point neural network model
Fixed point neural network model
VHDL

Figure 4: Validation flow of the neural network model.

Performance

The results show that the classical implementation needs $\sim 2.5\times$ more computational time than the In-Memory architecture with an higher energy consumed ($\sim 1.7\times$), for the model in Figure 1. The architectures implemented have the possibility to realize any kind of neural network, with more complex models or datasets. For the model depicted in Figure 1, the framerate achieved in the In-Memory case is 16337 fps with 0.79 μJ consumed, while for the OOM case is 6652 fps with 1.33μJ and a clock frequency of 4.22ns for both cases. By evaluating the architectures' perfomance for different neural network models, the In-Memory alternative is able to consume $\sim 3.7\times$ less energy and to save up to $\sim 5.7\times$ computational delay than the classical counterpart. Roughly comparisons have been performed with the state-of-the-art based on innovative technologies (such as RRAMs, MTJs Memristors,etc), showing very good computational delay with relatively low energy consumption results for the In-Memory architecture: the perfomance estimations for this case are pessimistic, since the memory array has been synthesized by Synopsys Design Compiler as a register file and each cell as a flip-flop, that is more complex than a custom memory cell. However, the resulting normalized energy and delay for the In-Memory case are $\sim 650pJ/neuron$ and $\sim 17ns/neuron$ respectively, that are comparable to an analog MTJ-based single-level-cell solution with an energy value of $\sim 450pJ/neuron$ and a normalized delay of $\sim 16ns/neuron$. Choosing beyond-CMOS technologies, enables the realization of very efficient solutions.

Thesis structure

This work is composed by the following chapters:

1. **State-of-the-art**, in which actual neural network implementations and technologies are reported (both In-Memory and OOM solutions);

2. **Comparisons**: the implementations discussed in the state-of-the-art are compared in terms of perfomance;

3. **Software implementation**: an explanation of the starting neural network model is provided, in which Python code is analyzed and discussed;

4. **Hardware implementations**: a detailed explanation on how the neural network has been realized in VHDL is given, for both the computational model (OOM and In-Memory respectively). In this part, the neural network model depicted in Figure 1 is used, because it is easier to understand. Next, it is demonstrated how the circuit can be used to implement different neural network models, with any kind of structure and dimension;

5. **Verification**: the results are compared and the correspondence between Python-Matlab-VHDL is tested, as already described in Figure 4. Here, three different neural network models are tested to demonstrate the capability of the circuits to implement any kind of neural network model and dataset: the original one (Figure 1), an MLP network and a fashion-MNIST based CNN;

6. **Synthesis - Place&Route**: performance results are provided for the models analyzed in the verification part. Moreover, by performing several synthesis, a parametric sweep is performed on the key parameters of the neural network (such as IFMAP sizes, OFMAP sizes, contemporary input channels and so on), to evaluate the trend of Power, Area, Timing and energy for both In-Memory and OOM architectures respectively. Roughly comparisons with the state-of-the-art are performed in this part;

7. **Conclusions and future work**: conclusions and improvements are proposed.

Table of contents

List of figures

Chapter 1

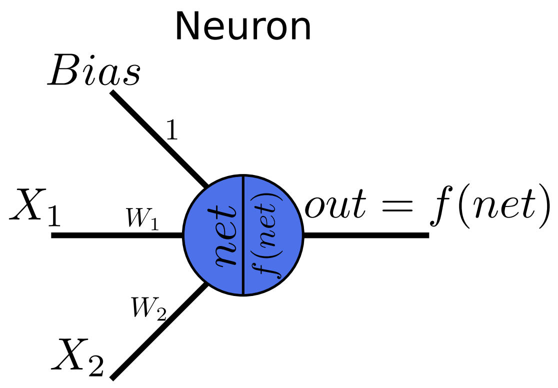
State of the art

1.1 Introduction

1.1.1 Artificial neural network [8]

Neuron

An artificial neural network is used to process very complex informations and in particular to give a classification. Its structure is based on the biological brain way-of-computation.[8] It is composed by "neurons", which are the basic blocks:



$$net = \sum_i^N X_i \times W_i + Bias$$

Figure 1.1: Neuron's structure

Neurons are organised in an interconnected network that is able to take decisions and to learn when these decisions are wrong [41]. Considering its equivalent structure from an ”electronic” point of view (Figure 1.1), the following terms are used:

- **Bias**: additive term;
- X_1,X_2: inputs of the neuron;
- W_1,W_2: weights. For each synapse there is a different weight. They can assume any value so they can be:
  1. Floating point weights: the values are represented in floating point, so the network can work at fully precision;
  2. Binarized weights: the weights can only assume ± 1 values;
  3. Ternary weights: the weights can assume $\{1,0,-1\}$. When a weight assumes the value 0 means that a particular neuron is not connected to another one.
- f: activation function of the neuron. There are different kind of activation functions, in particular the most used ones are:
  1. **Sigmoid function**: represented by the following equation

$$f(x) = \frac{1}{1+e^{-x}} \tag{1.1}$$

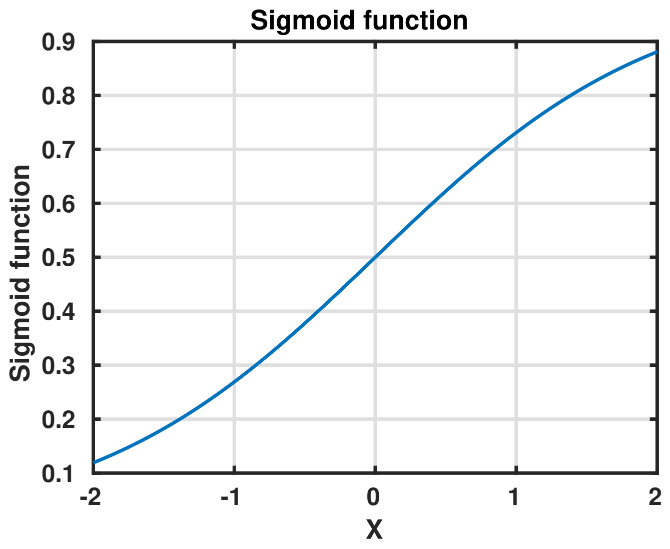


Figure 1.2: Sigmoid activation function

2. **Hyperbolic tangent**: given by

$$f(x) = tanh(x) \tag{1.2}$$

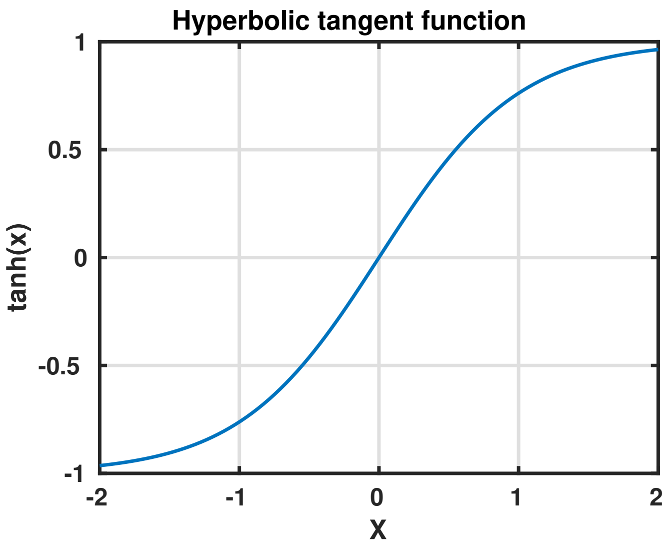


Figure 1.3: Hyperbolic tangent activation function

3. **ReLU function**: the term means "rectified linear unit" and it is given by
$$ReLU(x) = max(0,x) \tag{1.3}$$
This kind of activation is often used because it represents a good trade-off between accuracy and simplicity since, as it is possible to see in the plot in Figure 1.4, it is quite similar to the sigmoid or hyperbolic tangent functions.

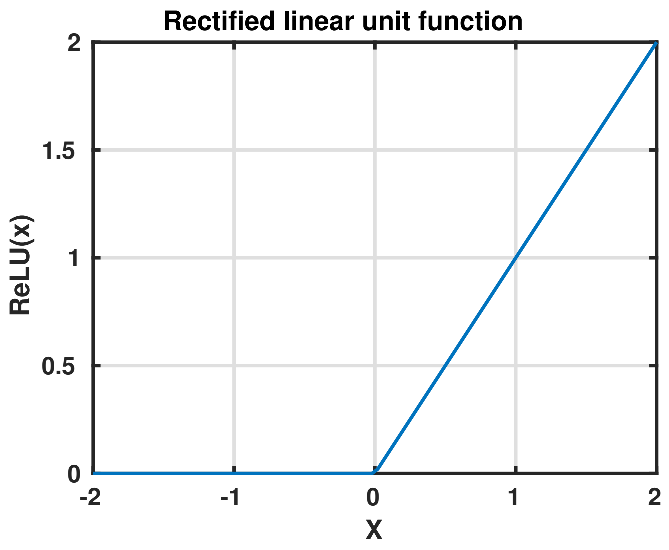


Figure 1.4: ReLU activation function

4. **Sign function**: this is used in binary/ternary neural networks and, considering the type of network, two different kinds of sign functions can be used:
$$sign^{(b)}(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases} \tag{1.4}$$
$$sign^{(t)}(x) = \begin{cases} 1, & \text{if } x \geq \rho \\ 0, & \text{if } -\rho \leq x < \rho \\ -1, & \text{if } x < -\rho \end{cases} \tag{1.5}$$

Neural network

In order to realize a neural network, it is possible to use multiple neurons into a neat structure composed by many layers. An example is reported in the following figure:

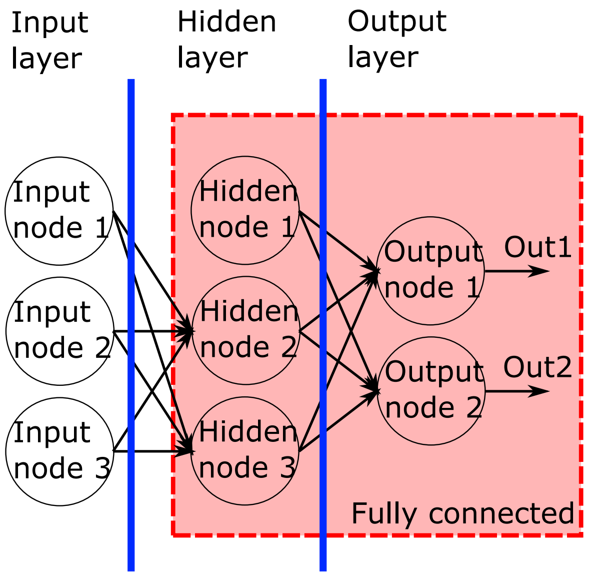


Figure 1.5: Neural network example from [32]

As it is possible to see in Figure 1.5, the network has **3 layers**:

1. **Input layer**: it simply reports the inputs to the following layer, by applying the neurons' activation fuction;

2. **Hidden layer**: the most important layer in the network, because it is used to do the computations explained before. Each neuron propagate in output the following quantity:

$$output(i) = f\left(\sum_{j=0}^{\#\text{inputs}} (input(j) \cdot w(j)) + bias\right) \tag{1.6}$$

3. **Output layer**: executes the same computations of the hidden layer, but the outputs coming from these neurons represent the **classification**, and so the result coming from the computations.

It is possible to have two different situations: the first one, when all the (N-1)-th neurons are connected to the following N-th neurons the network is called **fully connected**; otherwise, if this condition is not satisfied, the network is not fully connected. It is important to notice that a non fully-connected network is easily implementable by a ternary network, because in the ternary approach, weights can assume $\{1,0,-1\}$.

Size of the NN [42] The size of a neural network has to be chosen considering the application in which it will be used. For example, the usage of multiple hidden layers implies the capability to perform very difficult computations. Also the number of neurons in the hidden layers has to be chosen properly, in fact choosing many of them can imply **overfitting**, in which the number of the elements is very high and the dataset is not sufficient to update them all, reaching very low accuracies. On the contrary, if the number of neurons are minimal, the consequence is that the network is not capable anymore to perform complex computations and to process complex datasets (**underfitting**). A simple rule from [42] to choose the number of neurons is the following:

$$\begin{cases} \#\text{input\_neurons} < \#\text{hidden\_neurons} < \#\text{output\_neurons} \\ \#\text{hidden\_neurons} = \frac{2}{3} \cdot \#\text{input\_neurons} + \#\text{output\_neurons} \\ \#\text{hidden\_neurons} < 2 \cdot \#\text{input\_neurons} \end{cases} \tag{1.7}$$

Forward pass The first step in a NN is to "forward pass" the inputs towards the outputs. Inputs are propagated inside the neural network, which elaborates the partial results as explained before until the output is not reached. Once the outputs are computed, two different classifications are carried out from the NN, which represent the actual computation result with its original configuration (initial weights). The outputs are compared with the expected result and, if they are not the same, the network needs to be **trained** with a backward pass.

Backward pass Starting from output layer, the weights of the network needs to be updated, in order to reach the target output. To do this, it can be used the backpropagation algorithm method that will be explained later in subsection 1.1.4.

1.1.2 Convolutional neural networks

A convolutional neural network is a particular type of neural network which is able to process a very large data (such as an image) and to give a proper classification in output [34]. A CNN is composed by input, output and multiple hidden layers such as convolution, pooling, normalization and fully connected layers as depicted in Figure 1.6:

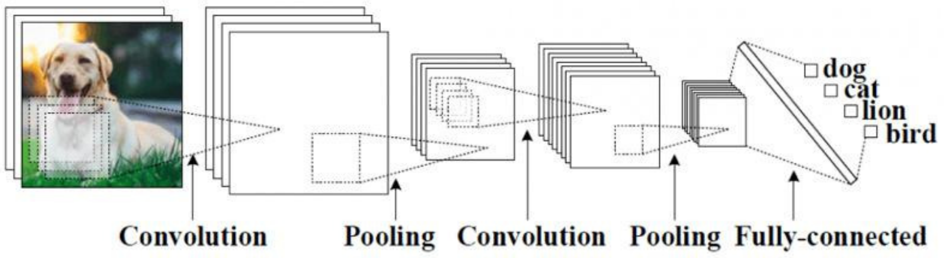


Figure 1.6: CNN example from [33]. There are several layers such as convolution, pooling, fully connected (already described) and normalization that can be trained in order to classify the input. In this case, an image is used but the CNNs can be used for different applications, such as natural language and speech recognition [34].

- **Convolutional layers**: a convolutional layer takes in input an image (considering the first layer, the IFMAP is represented by 3 matrices of pixels representing the RGB values) and gives in output a convolved group of matrices with a particular set of weights (called **kernels**). An example of kernel is depicted in the following figure:

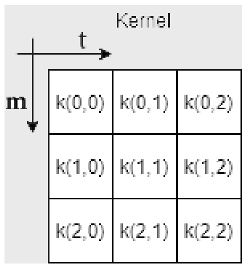


Figure 1.7: Example of a kernel in a CNN with 3x3 size.

The input image is also called **input feature map** (IFMAP) and the corresponding processed output is called **output feature map** (OFMAP). The general equation that defines the OFMAP can be formulated considering [31] and [16]:

$$y_o^{(l)}(j,i) = b_o^{(l)} + \sum_{c=0}^{\#\text{channels}-1} \sum_{m=0}^{\#\text{rows(kernel)}-1} \sum_{t=0}^{\#\text{cols(kernel)}-1} k_{o,c}^{(l)}(m,t) x_c^{(l)}(j + m + j(stride - 1), i + t + i(stride - 1))$$

Where:

- $^{(l)}$ is the layer. In this example, the first layer is considered;
- b_0 is the bias term;
- c is the input channel. As said previously, there can be more than one input channel in a convolutional neural network (RGB case);
- $k_{o,c}^{(l)}$ is the kernel weight of the channel and layer considered;
- $x_c^{(l)}$ is the corresponding input;
- stride is the corresponding step size used in the convolution.

This equation considers the case of a batch size equals to 1, where the batch size is the number of images in input [43]. Considering a simpler case, with a kernel size of 3x3, 5x5 IFMAP, only one channel and evaluated in the first layer, the equation that defines the OFMAP becomes:

$$\begin{aligned} y_0(j,i) = b_0 &+ k(0,0)x(j,i) + k(0,1)x(j,i+1) + k(0,2)x(j,i+2) + \\ &+ k(1,0)x(j+1,i) + k(1,1)x(j+1,i+1) + k(1,2)x(j+1,i+2) + \\ &+ k(2,0)x(j+2,i) + k(2,1)x(j+2,i+1) + k(2,2)x(j+2,i+2) \end{aligned}$$

By looking at this equation, it is possible to observe that it is quite similar to the neuron's equation, in fact:

$$\text{Conv} = b + \sum_{t=0}^{\#cols(kernel)-1} \sum_{m=0}^{\#rows(kernel)-1} k(m,t) \cdot x(j+m, i+t) \tag{1.8}$$

$$\text{Neuron} = \sum_{t=0}^{\#inputs} k(t)x(t) + b \tag{1.9}$$

It is possible to realize a convolutional layer by employing a fully connected neural network, in particular considering that a single convolved output can be realized as:

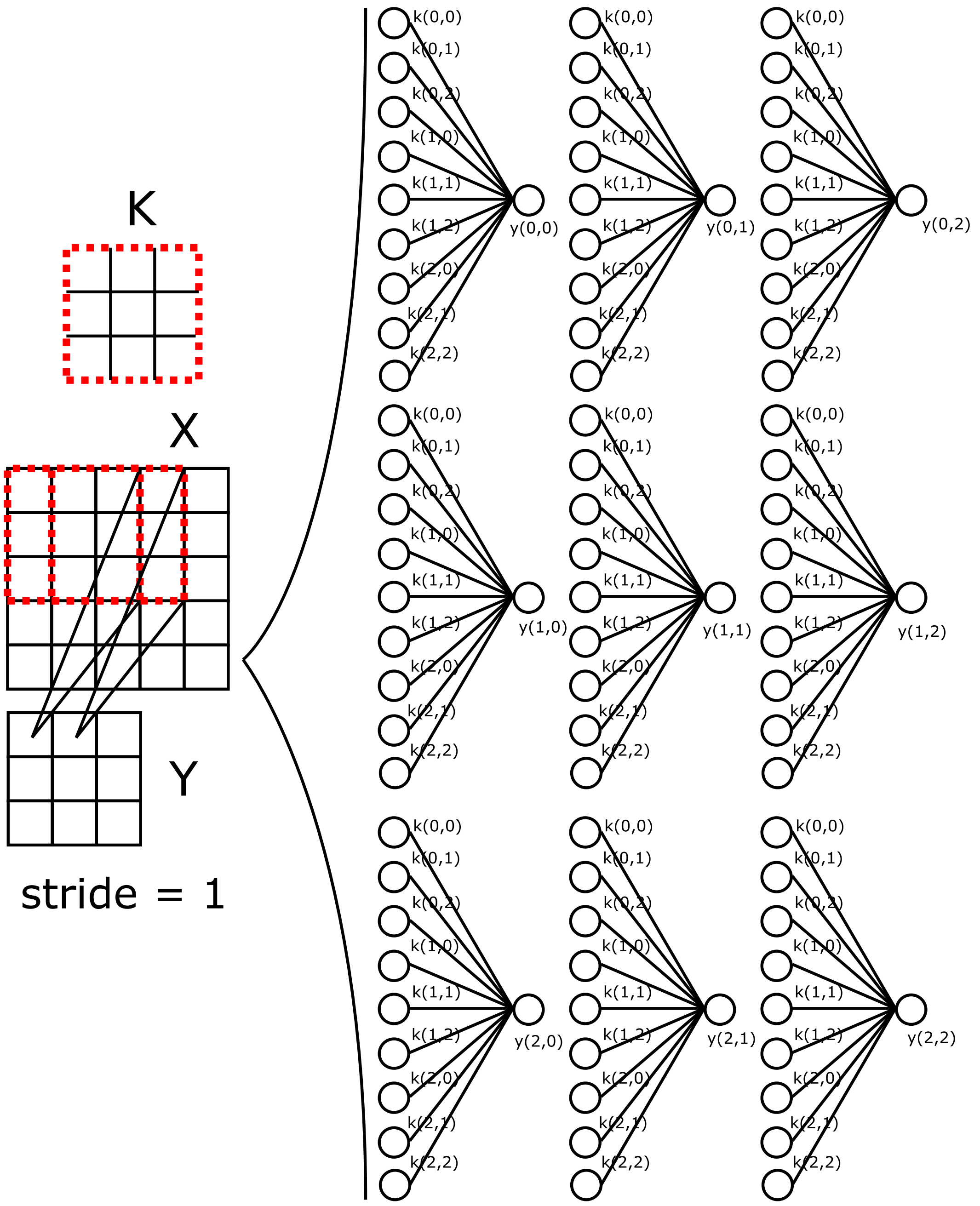


Figure 1.8: Convolution example with FC network. The weights used in the fully connected part are the same of the kernel.

Each element in the OFMAP is defined as the sum of products between the weights and the IFMAP.

- **Pooling layers**: Pooling is an important feature of CNNs, because it reduces the dimensions of the feature map, but maintaining the most important informations [35], allowing to reduce the size of the network and the parameters used, preventing overfitting. Considering the max pooling, it can be defined a window size (2x2 for example) and slide it into the OFMAP elaborated by the convolutional layer and take the largest element inside that window [35]. An intuitive example of max pooling is reported in the following figure:

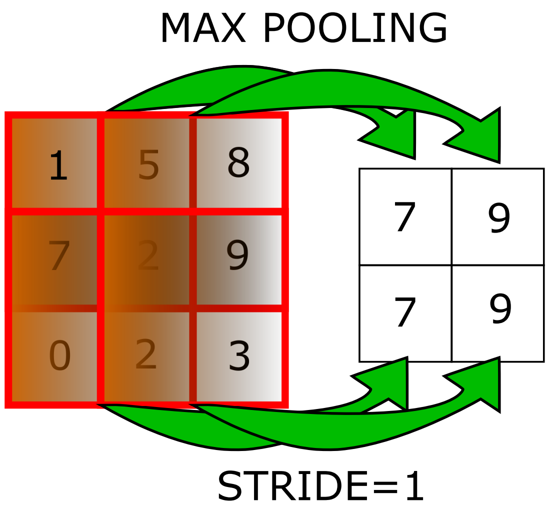


Figure 1.9: Example of max-pooling 2x2 and stride of 1 [35]

- **Batch normalization** [44]: it is a technique which is not reported in Figure 1.6 and it allows to reduce the problems coming from the training (such as slow convergence), in particular in very deep networks such as CNNs. The technique is based on the normalization of the inputs of each layer, in such a way that they will have mean output activation of 0 and standard deviation of 1 [44][45]. Main benefits are faster training, easier weights' initialization and the possibility to design deeper networks without losing precision[44].

The last part that composes a CNN is the fully connected layer, which has been explained previously.

1.1.3 Binary neural network [9]

As already mentioned, BNN has binary weights/activations. However, as analyzed in subsection 1.1.4, the fully precision weights are needed to compute the gradients. The main steps that characterize a BNN are the following ones:

1. Weights/inputs binarization [9]: the incoming activations and the weights are binarized as illustrated below:

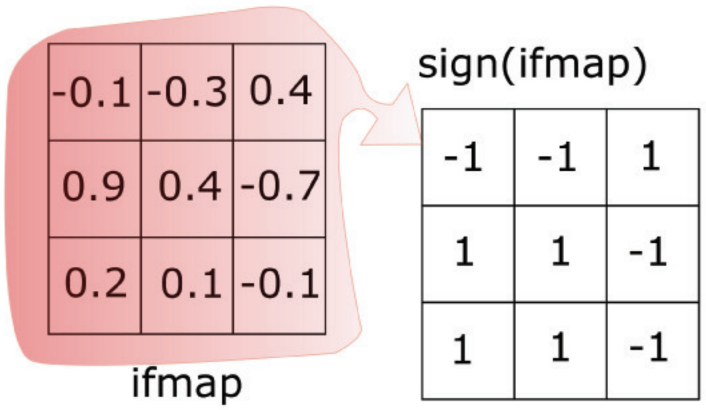


Figure 1.10: Binarization process based on the sign function of the input weights/activations.

2. Pass inputs in the neural network [9]: the binary matrices pass in the neural network, producing a result. Taking for example the following computation:

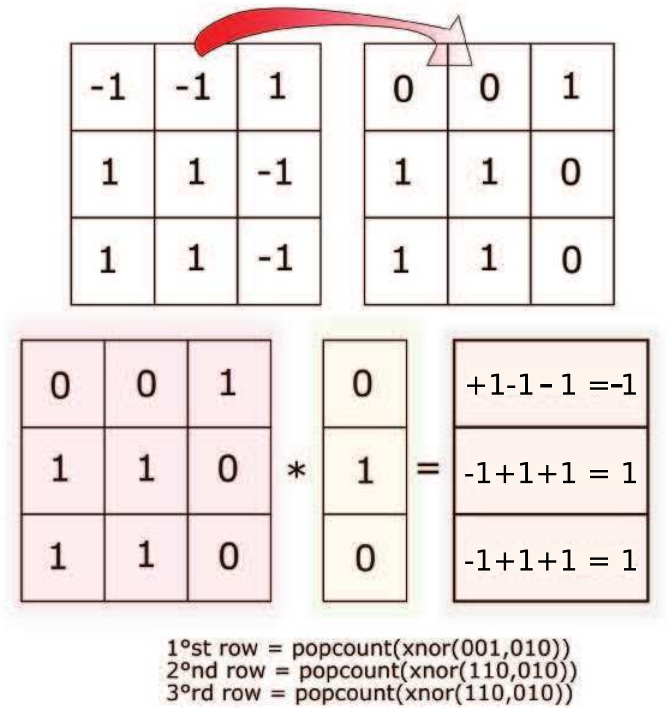

-1 -1 1
1 1 -1
1 1 -1
0 0 1
1 1 0
1 1 0
0 0 1
1 1 0
1 1 0
*
0
1
0
=
+1-1-1 =-1
-1+1+1 = 1
-1+1+1 = 1
1°st row = popcount(xnor(001,010))
2°nd row = popcount(xnor(110,010))
3°rd row = popcount(xnor(110,010))

Figure 1.11: Binary XNOR-Popcount based computation.

The result can be obtained by considering a series of XNOR operations and a final pop-count (number of ones - number of zeros). Considering the XNOR truth table:

Table 1.1: XNOR Truth table

A	B	OUT
0(-1)	0(-1)	1(+1)
0(-1)	1(+1)	0(-1)
1(+1)	0(-1)	0(-1)
1(+1)	1(+1)	1(+1)

As it is possible to see, "0" is considered as -1 and "1" as +1, so the bit-wise multiplication corresponds to XNOR's output. The final result of the computation in Figure 1.11 is given by:

$$y(0) = popcount(xnor(001,010)) = popcount(1,0,0) = -1 \tag{1.10}$$

Similarly, this procedure is applied to all the remaining rows.

3. Realization: BNN is then implemented as shown in the following figure:

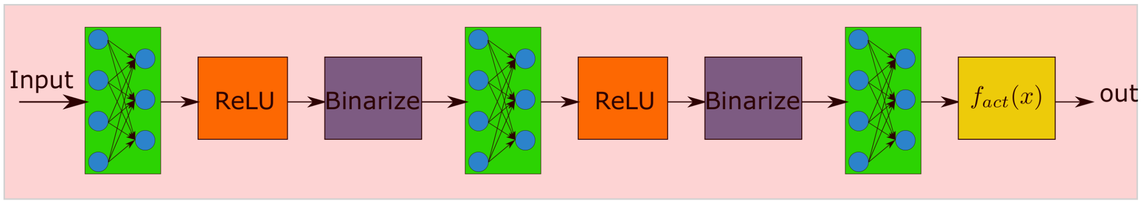


Figure 1.12: BNN implementation. The green boxes are fully connected layers.

The first layer of the BNN reported in Figure 1.12 is not binarized, because the correlation between unbinarized-binarized weights is weaker than the other layers [9].

It is possible to demonstrate that the accuracy of a BNN trained with MNIST dataset, compared to a fully precision neural network, slowly converges to the FP's one and this is a very important fact: BNN allows to reduce the resources and the computation complexity and are well suited for in-memory implementation [9].

1.1.4 Backpropagation algorithm [10]

In this part, the backpropagation algorithm is explained from [10]. If Ternary/Binary neural networks are considered, since the activation function is the $sign^{(t)}$, the derivative has to be approximated in some way: the approximation from [36] can be used.

Example of a 2x2x2 network [10]

The back-propagation algorithm is used to train a neural network, by computing the gradient that is needed in the calculation of the weights. [46] Backpropagation requires the derivative of the **loss function** (also known as error function) w.r.t. the network's output to be known. Consider the following neural network:

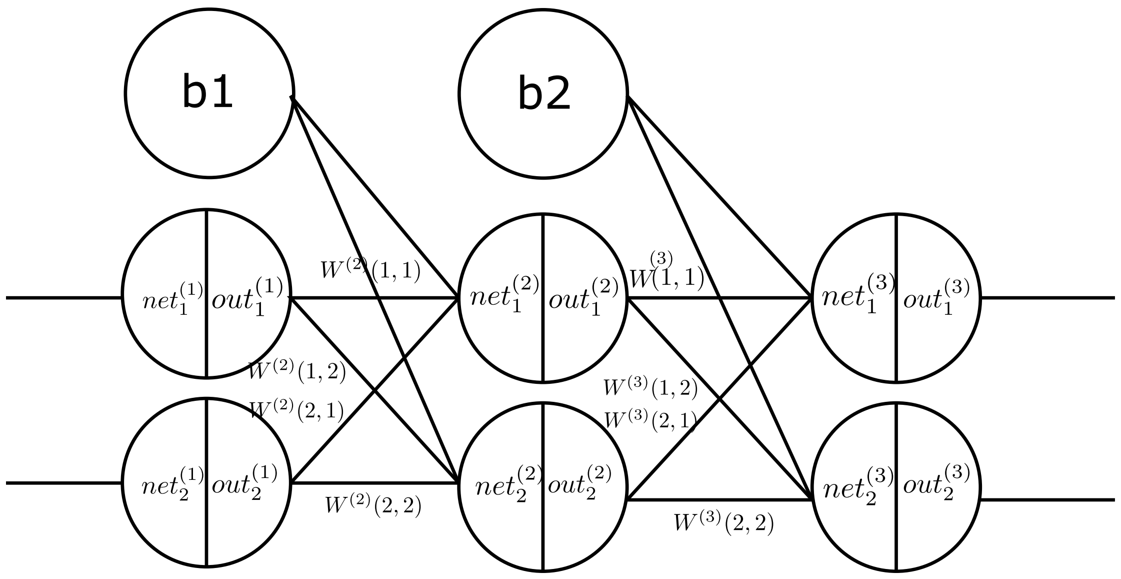

b1
b2
$net_1^{(1)}$ $out_1^{(1)}$
$net_2^{(1)}$ $out_2^{(1)}$
$net_1^{(2)}$ $out_1^{(2)}$
$net_2^{(2)}$ $out_2^{(2)}$
$net_1^{(3)}$ $out_1^{(3)}$
$net_2^{(3)}$ $out_2^{(3)}$
$W^{(2)}(1,1)$
$W^{(2)}(1,2)$
$W^{(2)}(2,1)$
$W^{(2)}(2,2)$
$W^{(3)}(1,1)$
$W^{(3)}(1,2)$
$W^{(3)}(2,1)$
$W^{(3)}(2,2)$

Figure 1.13: Neural network example. It is analyzed an MLP, in order to simplify the explanations. The same approach can be used in CNNs.

It has three layers (input layer, hidden layer and output layer) that are indicated as $^{(1)}$ for input, $^{(2)}$ for hidden and $^{(3)}$ for output.

Forward pass The first step of the backpropagation is to forward pass the inputs through the neural network and to see what is the result. This will be compared to the expected one (target), by considering the total error [10]:

$$E_{total} = \sum_{o=1}^{\#outputs} \frac{1}{2}(target(o) - out_o^{(3)})^2 = \tag{1.11}$$

$$= \frac{1}{2}(target(1) - out_1^{(3)})^2 + \frac{1}{2}(target(2) - out_2^{(3)})^2 + ... = \tag{1.12}$$

$$= E_{o1} + E_{o2} + ... \tag{1.13}$$

Backwards pass The errors obtained in output are backward-passed toward the input. The first layer encountered is the output layer:

- **Output layer**: considering for example the first output neuron depicted in Figure 1.14:

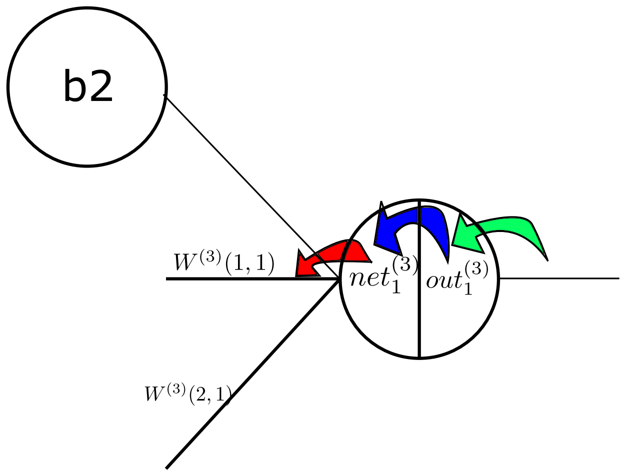


Figure 1.14: First output neuron

The neuron itself is divided into the input part (called **net**) and the output part (called **out**) [10]. In order to realize an algorithm which can be implemented in high-level language, the weights of each layer are stored into matrices in the following way:

$$\mathcal{W}^{(2)} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} \quad \mathcal{W}^{(3)} = \begin{bmatrix} w_5 & w_7 \\ w_6 & w_8 \end{bmatrix} \tag{1.14}$$

The input **net** can be defined as the weighted sum of all neuron's inputs (which correspond to the outputs of the previous layer), with their corresponding weights:

$$net_o^{(3)} = \sum_{i=1}^{\#col\,\mathcal{W}^{(3)}} x^{(2)}(i) \cdot \mathcal{W}^{(3)}(o,i) \tag{1.15}$$

And **out** as:

$$out_o^{(3)} = f_{act}(net_o^{(3)}) \tag{1.16}$$

The activation function into a binary/ternary neural network is the **sign**. In those particular case, out_o is defined as:

$$out_o^{(3)} = sign(net_o^{(3)}) \tag{1.17}$$

To determine the new value of w_5, the backpropagation algorithm computes the quantity $\frac{\partial E_{total}}{\partial w_5}$ and applies the chain rule from [10] expressed in Figure 1.14:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_1^{(3)}} \cdot \frac{\partial out_1^{(3)}}{\partial net_1^{(3)}} \cdot \frac{\partial net_1^{(3)}}{\partial w_5} \tag{1.18}$$

By expanding all the elements:

$$\begin{aligned} \frac{\partial E_{total}}{\partial out_1^{(3)}} &= \frac{\partial}{\partial out_1^{(3)}} \left(\sum_{o=1}^{\#outputs} \frac{1}{2}(target(o) - out_o^{(3)})^2 \right) \\ &= -(target(1) - out_1^{(3)}) \\ &= out_1^{(3)} - target(1) \end{aligned} \tag{1.19}$$

$$\begin{aligned} \frac{\partial net_1^{(3)}}{\partial w_5} &= \frac{\partial}{\partial w_5} \left(\sum_{i=1}^{\#col\,\mathcal{W}^{(3)}} x^{(2)}(i) \cdot \mathcal{W}^{(3)}(i,1) \right) \\ &= \frac{\partial}{\partial w_5} \left(x^{(2)}(1) \cdot w_5 + x^{(2)}(2) \cdot w_6 + bias \right) \\ &= x^{(2)}(1) \end{aligned} \tag{1.20}$$

The last term $\frac{\partial out_1^{(3)}}{\partial net_1^{(3)}}$ considers the derivative of the activation function $f_{act}(x)$. In the Binary/Ternary case, it can be approximated as indicated in Figure 1.15 from [36]. In formulas:

$$\frac{\partial Sign(x)}{\partial x}_{(c)} = \begin{cases} 1/2a, & \text{if } r - a \leq |x| \leq r + a \\ 0, & \text{others} \end{cases} \tag{1.21}$$

$$\frac{\partial Sign(x)}{\partial x}_{(d)} = \begin{cases} \frac{-1}{a^2}(|x| - (r + a)), & \text{if } r \leq |x| \leq r + a \\ \frac{1}{a^2}(|x| - (r - a)), & \text{if } r - a \leq |x| < r \\ 0, & \text{others} \end{cases} \tag{1.22}$$

To simplify the equations, $\frac{\partial f_{act}(x)}{\partial x}_{(d)}$ is expressed as:

$$f'_{act}(x) = \frac{\partial f_{act}(x)}{\partial x}_{(d)} \tag{1.23}$$

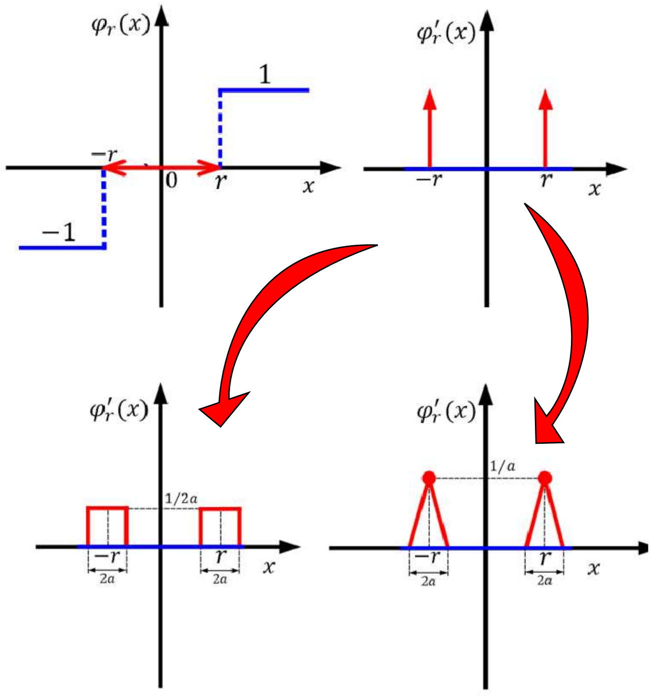


Figure 1.15: Approximation of the derivative of the sign function from [36]

Finally, the original equation of $\dfrac{\partial E_{total}}{\partial w_5}$ can be rewritten as:

$$\frac{\partial E_{total}}{\partial w_5} = (out_1^{(3)} - target(1)) \cdot x^{(2)}(1) \cdot f'_{act}\left(\sum_{i=1}^{\#col\,\mathcal{W}^{(3)}} x^{(2)}(i) \cdot \mathcal{W}^{(3)}(1,i)\right) \quad (1.24)$$

In order to simplify the expression, the following equality is imposed:

$$\delta_o = (out_o^{(3)} - target(o)) \cdot f'_{act}(net_o^{(3)}) \quad (1.25)$$

The final expression for w_5 is given by:

$$\frac{\partial E_{tot}}{\partial w_5} = \delta_1 \cdot x^{(2)}(1) \quad (1.26)$$

The update rule for the weight w_5 is the following:

$$w_5^+ = w_5 - \eta \cdot \frac{\partial E_{total}}{\partial w_5} \tag{1.27}$$

Where η is the learning rate, which is an important parameter that indicates how much the weights are adjusted with respect the loss function. By using a very small value of learning rate, it means that the algorithm moves very slowly and takes very long time to converge: typically the accuracy achievable is higher in the case of small learning rates. Trying now to provide a general expression for a weight w connected to a specific neuron (in particular the first output one), the following equation can be considered:

$$\frac{\partial E_{tot}}{\partial w} = \delta_1 \cdot x^{(2)}(k) \tag{1.28}$$

To simplify the equations, $\frac{\partial E_{tot}}{\partial w} = \psi$. The computation for all the weights becomes:

```
for k=1:$\#rows(\mathcal{W}^{(3)})$
    $\psi = \delta_1 \cdot x^{(2)}(k)$
    $\mathcal{W}^{(3)+}(k,1) = \mathcal{W}^{(3)}(k,1) - \eta \cdot \psi$
end
```

Extending this concept to all the output neurons in the last layer, the final steps for the output layer becomes:

```
for o=1:$\#cols(\mathcal{W}^{(3)})$
    for k=1:$\#rows(\mathcal{W}^{(3)})$
        $\psi = \delta_o \cdot x^{(2)}(k)$
        $\mathcal{W}^{(3)+}(k,o) = \mathcal{W}^{(3)}(k,o) - \eta \cdot \psi$
    end
end
```

- **Hidden layer**: for the hidden layer, the concept is more complicated, since it has to use all the computations done in the previous layer. So, considering for example the computation of w_1:

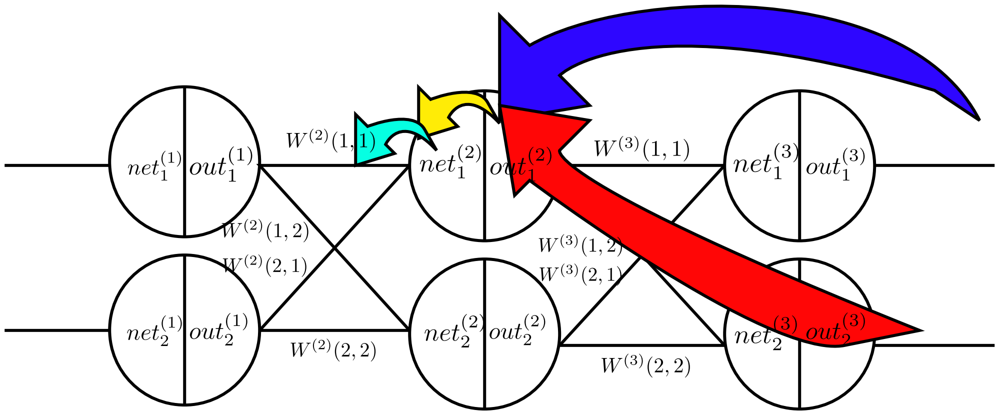


Figure 1.16: Example of w_1 ($W^{(2)}$(1,1)) computation from [10]. Biases are not reported because they are not used in the computation.

The term needed to be figured out is [10]:

$$\frac{\partial E_{tot}}{\partial w_1} = \frac{\partial E_{tot}}{\partial out_1^{(2)}} \frac{\partial out_1^{(2)}}{\partial net_1^{(2)}} \frac{\partial net_1^{(2)}}{\partial w_1} \tag{1.29}$$

The contribution of E_{tot} is given by the sum of the errors in output, so:

$$E_{tot} = \sum_{o=1}^{\#cols(\mathcal{W}^{(3)})} E_0(o) \tag{1.30}$$

$$\frac{\partial E_{tot}}{\partial out_1^{(2)}} = \frac{\partial}{\partial out_1^{(2)}} \left(\sum_{o=1}^{\#cols(\mathcal{W}^{(3)})} E_0(o) \right) \tag{1.31}$$

By exploiting the j-th element of the sum, the derivative can be rewritten as:

$$\frac{\partial E_0(j)}{\partial out_1^{(2)}} = \frac{\partial E_0(j)}{\partial net_j^{(3)}} \cdot \frac{\partial net_j^{(3)}}{\partial out_1^{(2)}} \tag{1.32}$$

$$\frac{\partial net_j^{(3)}}{\partial out_1^{(2)}} = \mathcal{W}^{(3)}(1,j) \tag{1.33}$$

Now $\dfrac{\partial E_0(j)}{\partial net_j^{(3)}}$ can be computed as:

$$\frac{\partial E_0(j)}{\partial net_j^{(3)}} = \frac{\partial E_0(j)}{\partial out_j^{(3)}} \cdot \frac{\partial out_j^{(3)}}{\partial net_j^{(3)}} \tag{1.34}$$

$$\frac{\partial E_0(j)}{\partial out_j^{(3)}} = out_j^{(3)} - target(j) \tag{1.35}$$

$$\frac{\partial out_j^{(3)}}{\partial net_j^{(3)}} = f'_{act}(net_j^{(3)}) \tag{1.36}$$

Putting all together:

$$\frac{\partial E_{tot}}{\partial out_1^{(2)}} = \left(\sum_{o=1}^{\#cols(\mathcal{W}^{(3)})} (out_o^{(3)} - target(o)) \cdot f'_{act}(net_o^{(3)}) \cdot \mathcal{W}^{(3)}(1,o) \right) \tag{1.37}$$

By remembering:

$$\delta_o = (out_o^{(3)} - target(o)) \cdot f'_{act}(net_o^{(3)}) \tag{1.38}$$

The term $\dfrac{\partial E_{tot}}{\partial out_1^{(2)}}$ becomes:

$$\frac{\partial E_{tot}}{\partial out_1^{(2)}} = \sum_{o=1}^{\#cols(\mathcal{W}^{(3)})} \delta_o \cdot \mathcal{W}^{(3)}(1,o) \tag{1.39}$$

The final expression is:

$$\frac{\partial E_{tot}}{\partial w_1} = \sum_{o=1}^{\#cols(\mathcal{W}^{(3)})} \delta_o \cdot \mathcal{W}^{(3)}(1,o) \cdot f'_{act} \left(\sum_{i=1}^{\#rows(\mathcal{W}^{(2)})} x^{(1)}(i) \cdot \mathcal{W}^{(2)}(i,1) \right) \cdot x^{(1)}(1) \tag{1.40}$$

The final expression can be further reduced for a single neuron's weights considering:

$$\delta^{(L)}(1,o) = \mathcal{W}^{(L)}(1,o) \cdot f'_{act}\left(\sum_{i=1}^{\#rows(\mathcal{W}^{(L-1)})} x^{(L-2)}(i) \cdot \mathcal{W}^{(L-1)}(i,1)\right) \tag{1.41}$$

For a generic weight w of the same neuron (the first one of the hidden layer):

$$\frac{\partial E_{tot}}{\partial w} = \psi = \sum_{o=1}^{\#cols(\mathcal{W}^{(3)})} \delta_o \cdot \delta^{(3)}(1,o) \cdot x^{(1)}(k) \tag{1.42}$$

From an algorithm point of view, each weight of the hidden layer's neurons can be obtained as follows:

```
for p=1:$\#cols(\mathcal{W}^{(2)})$
    for k=1:$\#rows(\mathcal{W}^{(2)})$
        $\psi = \sum_{o=1}^{\#cols(\mathcal{W}^{(3)})} \delta_o \cdot \delta^{(3)}(p,o) \cdot x^{(1)}(k)$
        $\mathcal{W}^{(2)+}(k,p) = \mathcal{W}^{(2)}(k,p) - \eta \cdot \psi$
    end
end
```

General case: N-layers network

The procedure is the same as described above, so in order to define the equation in the general case, a generic **U-th** neuron is considered. The $\psi = \frac{\partial E_{tot}}{\partial w}$ for a single weight w is defined as following in the different cases:

- **N-th layer**: $\psi = \delta_U \cdot x^{(N-1)}(k)$
- **(N-1)th layer**: $\psi = \sum_{o=1}^{\#cols(\mathcal{W}^{(N)})} \delta_o \cdot \delta^{(N)}(U,o) \cdot x^{(N-2)}(k)$

- **(N-2)th layer**: $\psi = \sum_{o=1}^{\#cols(\mathcal{W}^{(N)})} \sum_{p=1}^{\#cols(\mathcal{W}^{(N-1)})} \delta_o \cdot \delta^{(N)}(p,o)\delta^{(N-1)}(U,p) \cdot x^{(N-3)}(k)$

As it is possible to see, everytime the equation goes down by 1 layer, the number of sums increases and the equation considers the term δ of the previous layers. So it is not possible to define a-priori an equation that determines all the weights, because it depends on the number of layers and on the weights' values updated in the previous cycle. It is important to consider that this algorithm works well if the weights' values are in **floating point** representation: before binarizing, the neural network have to be trained.

1.2 Software based neural networks

1.2.1 ImageNet Classification with Deep Convolutional Neural Networks [11]

Introduction

This approach describes AlexNet, which is a convolutional neural network that is able to process a very large dataset such as ImageNet[11]. With a standard feedforward approach, the recognition task made on thousands of images in input requires a very big neural network with a very large number of parameters (weights). Considering in fact:

$$\#\text{parameters}_{FF} = \sum_{i=1}^{\#\text{layers}-1} \#\text{neurons}(i) \cdot \#\text{neurons}(i+1) \tag{1.43}$$

Where i indicates the current layer analyzed. CNNs instead are well suited for very large inputs, because the number of parameters are less than the MLP solution, but the precision is a little bit degraded. AlexNet-CNN is implemented with 2 Nvidia GTX 580 3GB GPUs with an algorithm optimized to train faster the network itself, to reduce overfitting and to achieve very good results on these datasets [11]. The images from ImageNet in input are prescaled to 224x224.

Architecture

The architecture consists into 8 layers (5 convolutional and 3 fully connected)[11] as reported in Figure 1.17:

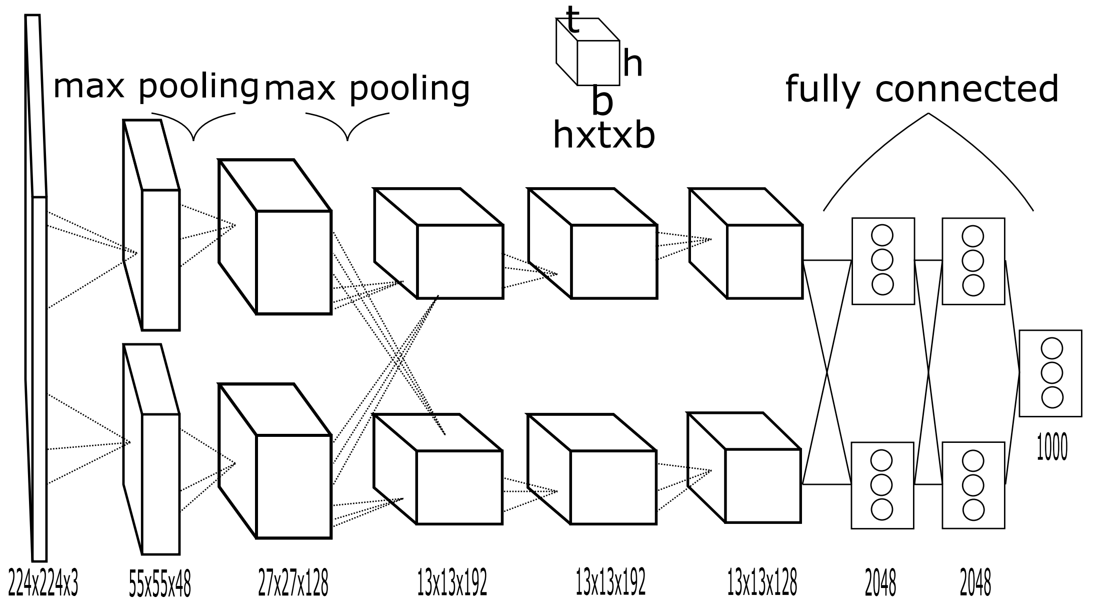


Figure 1.17: AlexNet architecture from [11]. The architecture is divided into two parts handled by the two GPUs respectively, with some layers in which they communicate with a DMA (direct memory access) approach. The two parts are identical and so the dimensions reported are the same.

In Figure 1.17, the output of the last fully-connected layers is connected to a final layer with **softmax** activation (a derivable function which simply takes the maximum of its inputs), which gives 1000 classification labels. ReLU is applied where specified and also after the fully connected part: this activation function is preferred to the others like $tanh(x)$ or $sigmoid(x)$, because the training time is improved. The layers are organized as follows:

- The first convolutional layer has 224x224x3 input image, processed by 96 kernels of size 11x11x3 with a stride of 4 pixels. The output dimensions can be determined considering:

$$t_{out} = \frac{t_{in} - t_{filter}}{stride} + 1 = \frac{224 - 11}{4} + 1 \simeq 55 \tag{1.44}$$

$$h_{out} = \frac{h_{in} - h_{filter}}{stride} + 1 = \frac{224 - 11}{4} + 1 \simeq 55 \tag{1.45}$$

- The second layer takes as input the output of the first layer, which has been pooled and normalized, and convolves it with 256 filters of size 5x5x48;

- The third, fourth and fifth layers have 384,384 and 256 kernels of size 3x3x256, 3x3x192 and 3x3x192 respectively;
- The fully-connected layers have 4096 neurons.

Using two GPUs in this configuration reduces top-1 and top-5 error rates by 1.7% and 1.2% w.r.t other solutions with only one GPU[11]. A local response normalization in [11] is used in order to reduce the top-1 and top-5 error rates by 1.4% and 1.2%, respectively, and it is used after applying ReLU in some layers. This has been tested also on CIFAR-10, producing an error of 11% w.r.t 13% without it.

Overlapping pooling Pooling procedure has already been explained in the introduction, but here it is used the overlapping pooling. Considering for example a pooling region of ZxZ: if the stride is larger than or equal to Z, the pooling windows does not overlap. Here it is presented a simple example:

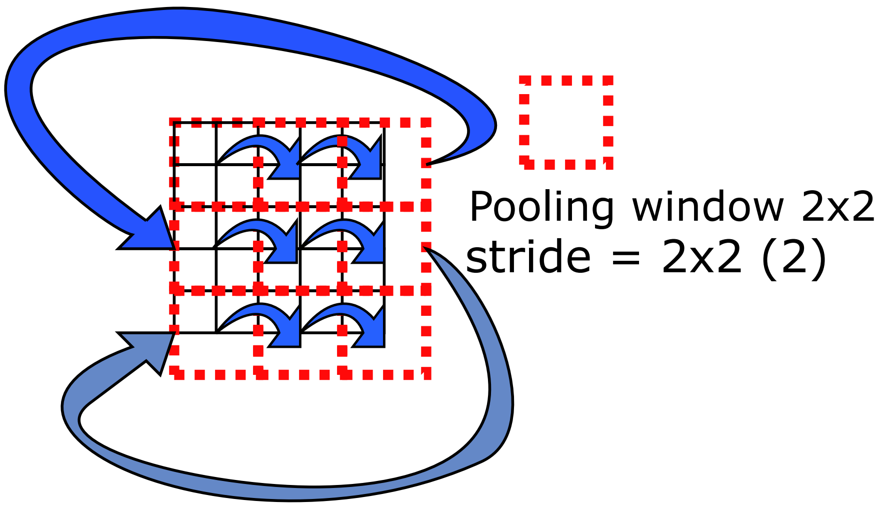


Figure 1.18: Example of a non-overlapped pooling and pooling procedural steps

If $s < z$, we obtain overlapping pooling. In particular [11] uses a solution in which stride = 2 and z = 3, reducing top-1 and top-5 error rates by 0.4% and 0.3%, respectively w.r.t non-overlapping scheme.

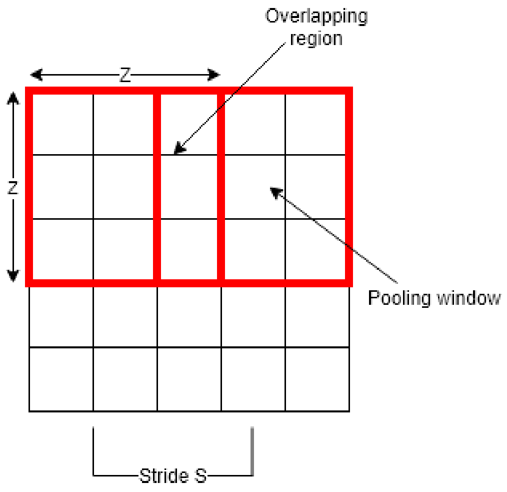


Figure 1.19: An example of a 3x3 window with an overlapping pooling with stride s = 2 and z = 3

Reducing overfitting

Since in this network there are 60 millions parameters [11] and it has to classify among 1000 different classes, overfitting problem could introduce a significant overhead in terms of performance.

Data augmentation One of the possible ways to reduce the overfitting is to expand the dataset using some transformations [11]:

1. Generation of image translations and horizontal reflections. 224x224 patches and their translations/reflextions from the 256x256 dataset images are used and training is performed, which size is improved by a factor of 2048;

2. Alternate the intensities of the RGB channels in images used in the training set, by means of an important property of natural images that consists on objects' identity-invariance to changes in the intensity and color of the illumination.

Dropout Dropout is a useful technique which consists to selectively turn off some neurons in the hidden layer with p probability, in order to speed up the training [11]. In fact, the inputs are sampled by a different network at each iteration step, allowing the backpropagation algorithm to converge faster. However, even if the architecture is different at each step, the weights are shared in the network.

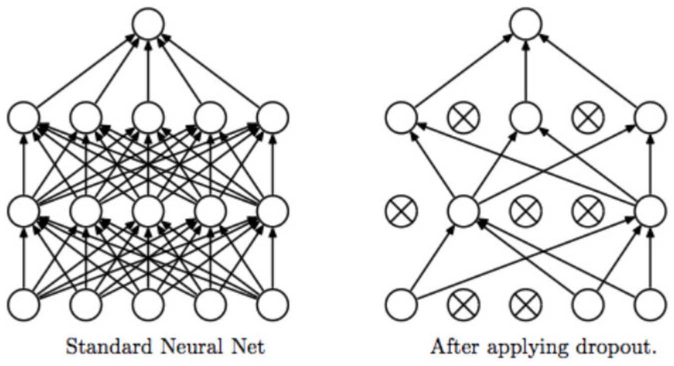


Figure 1.20: Example of Dropout technique from [37]

Results

Several results are reported in [11]. They are summarized in the following table:

Table 1.2: Classification results from [11]

Competition	top-1[%]	top-5[%]	Dataset/year	Network structure	Details
ILSVRC-2010	37.5	17	ImageNet	5 layers CNN	-
ILSVRC-2012	40.7	18.2	ImageNet	5 layers CNN	-
-	67.4	40.9	ImageNet 2009 (10184 categories, 8.9 million images)	5 layers CNN	Half images for training, half for classification

1.2.2 XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks [12]

Introduction

The CNNs are very good in activities like speech recognition, image classification and so on. The main CNNs' drawback is the high amount of computational power and memory required to perform all the computations, which are mainly based on millions of parameters as explained in the AlexNet. The goals are to enable mobile devices and low-power embedded systems to handle a neural network (such as a convolutional one), to recognize with an high accuracy and to save power due to the limited capacity of the batteries. One of the possible ways to reach these objectives is to binarize the neural network, in such a way that the accuracy will be comparable to the original implementation. Two alternatives can be analyzed:

1. Binary-Weight-Networks: only the weights are approximated to a binary value (± 1). The MAC operations are simply reduced to additions/subtractions. This kind of CNN can be easily integrated into an embedded system;

2. XNOR-Networks: both the weights and the inputs are approximated to the binary values. As already mentioned in the introduction, if weights and inputs are binarized, the MAC operation become simply a XNOR + popolation counting.

Differences between different types of networks are reported in the following table:

Table 1.3: Comparison between network types from [12]

Network type	Operations used	Memory saving	Computation saving	Accuracy on ImageNet % (AlexNet)
Standard (FP)	+,-,x	1x	1x	56.7
Binary weight	+,-	32x	2x	56.8
XNOR-Net	XNOR,bitcount	32x	58x	44.2

Binary convolutional neural network [12]

- **I**: represents the input tensor for each layer. $\mathbf{I} \in \Re^{c \times w_{in} \times h_{in}}$, where (c, w_{in}, h_{in}) represents channel, width and height respectively;

- **W**: represents the weight tensor of each layer. $\mathbf{W} \in \Re^{c \times w \times h}$, where $w \leq w_{in}$, $h \leq h_{in}$.

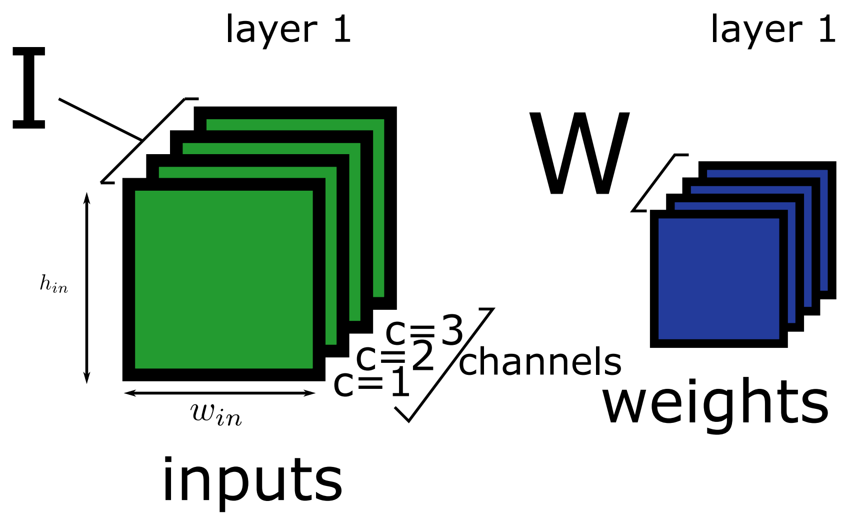


Figure 1.21: Weights and inputs represented as tensors.

Binary weight network In order to constraint a CNN to have binary weights, the following approximation is imposed from [12]:

$$\mathbf{W} \approx \alpha \mathbf{B} \tag{1.46}$$

where **B** is the binary filter and α is a scaling factor. So a convolution operation can be transformed from [12]:

$$\mathbf{I} * \mathbf{W} \approx (\mathbf{I} \circledast \mathbf{B})\alpha \tag{1.47}$$

Estimating binary weights [12] The α value can be estimated considering the loss function, which is defined as:

$$\text{Loss function} = \|\mathbf{W} - \alpha\mathbf{B}\|^2 \tag{1.48}$$

$$\alpha,\mathbf{B} = argmin(\text{Loss function}) \tag{1.49}$$

It is possible to demonstrate that this equation brings to an optimization based on the following assumptions [12]:

$$\begin{cases} \mathbf{B}_i = +1, \text{ if } \mathbf{W}_i \geq 0 \\ \mathbf{B}_i = -1, \text{ if } \mathbf{W}_i < 0 \end{cases} \tag{1.50}$$

So $\mathbf{B} = sign(\mathbf{W})$. While for the scaling factor α, the derivative of the loss function w.r.t. α is considered and set it to 0 from [12]. The result obtained is:

$$\alpha = \frac{\mathbf{W}^T\mathbf{B}}{n} = \frac{\mathbf{W}^T sign(\mathbf{W})}{n} = \frac{\sum|\mathbf{W}_i|}{n} = \frac{1}{n}\|\mathbf{W}\|_{l1} \tag{1.51}$$

Training The algorithm proposed by [12] to train the binary networks is the following:

1. Binarization of the weight filters at each layer by computing $\mathcal{B}$,$\mathcal{A}$;
2. Forward propagation with binary weights and their corresponding scaling factors;
3. Backward propagation, where the gradients are computed w.r.t. the estimated weight filters $\widetilde{W} = \alpha \times Sign(\mathbf{W})$;
4. Parameters and the learning rate gets updated.

XNOR-Networks

A convolution operation, which consists in dot products and shifts, can be performed by binarizing both inputs and weights. By doing this, convolution becomes a simple XNOR-Bitcounting sequence, which can be implemented with low cost. To approximate the dot product $(\mathbf{X},\mathbf{W})$ in a binary form in which $X \approx \beta H^T$ and $W \approx \alpha B$, the following equation can be considered from [12]:

$$\alpha^*,\mathbf{B}^*,\beta^*,\mathbf{H}^* = argmin\|\mathbf{X}\cdot\mathbf{W} - \beta\alpha\mathbf{H}\cdot\mathbf{B}\| \tag{1.52}$$

It is possible to demonstrate that the best solution is achieved when :

$$H = sign(X) \tag{1.53}$$

$$B = sign(W) \tag{1.54}$$

$$\beta = \left(\frac{1}{n}\|\mathbf{X}\|_{l1}\right) \tag{1.55}$$

$$\alpha = \left(\frac{1}{n}\|\mathbf{W}\|_{l1}\right) \tag{1.56}$$

For the binarizing input procedure, a more efficient procedure than computing β for all the combinations can be used, and it is based on K and α values. Once binarizing is completed, the convolution can be approximated as [12]:

$$\mathbf{I} * \mathbf{W} \approx (sign(\mathbf{I}) \circledast sign(\mathbf{W}))\cdot\mathbf{K}\alpha \tag{1.57}$$

where $\circledast$ represents XNOR-Bitcount operations and K and α are defined as:

$$\begin{cases} \mathbf{K} = \frac{\sum_{\text{channels}}|\text{inputs}|}{\#channels} * \text{2D Matrix}\left(\frac{1}{w^2_{filter}}\right) \\ \alpha = \frac{\sum\|\text{weights}\|}{\#weights} \end{cases} \tag{1.58}$$

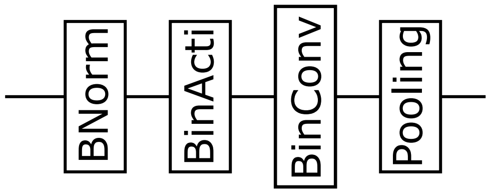


Figure 1.22: Structure of the XNOR-Net from [12]

By looking at Figure 1.22, max-pooling is placed after the convolution because the pooling itself reduces the accuracy in a binary solution (almost often returns +1).

Normalization of the inputs before binarization is done to improve the accuracy. The binary activation layer (BinActiv) computes K and sign(I), and in BinConv, given K and sign(I), binary convolution is performed.

Results

Here are reported some graphs which represents the various results obtained by measuring the efficiency,speedup,memory required,accuracy

Table 1.4: Required memory for different architectures from [12]

Architecture	Double precision [MB]	Binary precision [MB]
VGG-19	1000	16
ResNet-18	100	1.5
AlexNet	475	7.4

Table 1.5: Classification accuracy from [12]

Architecture	Dataset	Implementation	Error rate [%]	TOP-1 [%]	TOP-5 [%]
-	CIFAR-10	BWN	9.88	-	-
-	CIFAR-10	XNOR-NET	10.17	-	-
AlexNet	ImageNet	BWN	-	56.8	79.4
AlexNet	ImageNet	BinaryConnect	-	35.4	61.0
AlexNet	ImageNet	XNOR-NET	-	44.2	69.2
AlexNet	ImageNet	Binary Weight Binary Input	-	27.9	50.42
AlexNet	ImageNet	Fully precision	-	56.6	80.2

1.2.3 BinaryConnect: Training Deep Neural Networks with binary weights during propagations [13]

Introduction

BinaryConnect is an approach that enables low-power computations in a neural network adapted to be "binary". The following parts explain how this network works.

BinaryConnect [13]

The key is to impose the values of the weights to ± 1, as already seen previously: as a result, all MAC operations are reduced to only additions-subtraction, bringing less power consumption.

Deterministic/stochastic binarization One of the possibility to binarize the weights is to choose a very simple approach, based on taking the sign of the real-value weight from [13]:

$$w_b = \begin{cases} +1, if\, w \geq 0 \\ -1, \text{ otherwise} \end{cases} \tag{1.59}$$

Another possibility is to use a statistical approach [13]:

$$w_b = \begin{cases} +1, \text{ with probability } p = \sigma(w) \\ -1, \text{ with probability } p = 1 - \sigma(w) \end{cases} \tag{1.60}$$

Where σ is the "hard sigmoid" function from [13]:

$$\sigma(x) = max\left(0, min\left(1, \frac{x+1}{2}\right)\right) \tag{1.61}$$

About training, the network works exactly as the previous analyzed cases, but here no α,**K** values are used as in XNOR-Net. Weights during backpropagation have to be at fully precision, because otherwise the algorithm does not work anymore. Batch normalization is used here to accelerate training and **ADAM** [13] as learning rule is employed, which is a different algorithm than stochastic gradient descent, in fact ADAM produces a smaller error rate (10.47%) w.r.t 11.45%. ADAM learning rule has been reported in section 1.2.4.

Results In this part, the accuracy of BinaryConnect has been reported from [13], which uses three different approaches and some datasets:

1. Use the resulting binary weights w_b;

2. Use the real-valued weights w (binarized weights helps only to reduce training

time);

3. Stochastic case: different networks can be obtained in this case and their accuracy can be computed by averaging the output of all of them.

Table 1.6: Resulting error rates and network structures used in [13]

Method used	MNIST [%]	CIFAR-10 [%]	SVHN [%]
No regularizer	1.30 ± 0.04	10.64	2.44
BinaryConnect(Det.)	1.29 ± 0.08	9.90	2.30
BinaryConnect(stoch)	1.18 ± 0.04	8.27	2.15
50% dropout	1.01 ± 0.04	-	-
Details	3 hidden layers 1024 neurons Exponential Decay rate	Network structure: (2x128 C3)-MP2-(2x256C3)-MP2 (2x512C3)-MP2 (2x1024FC)-10SVM	Network structure: (2x128 C3)-MP2-(2x256C3)-MP2 (2x512C3)-MP2 (2x1024FC)-10SVM

The meaning of the terms indicated in the Table 1.6 are the following from [13]:

- **C3**: ReLU convolutional layer with 3x3 size;
- **MP2**: Max pooling (2x2 size);
- **FC**: fully connected;
- **SVM**: Support vector machine, a supervised learning model.

1.2.4 A Ternary Weight Binary Input Convolutional Neural Network: Realization on the Embedded Processor [14]

Introduction

Ternary network has three values of weights ($\{0,1,-1\}$) respectively: the value of "0" means that the computation can be skipped, improving the network speed. As already mentioned, this solution enables very low power architecture to work into an embedded system. If CNN's network structure is considered, the equation that is able to determine the output for the convolutional layer is reported again, but also considering the stride in the case of only one input channel from [16]:

$$y_0(j,i) = \sum_{m=0}^{\#\text{rows(kernel)}-1} \sum_{t=0}^{\#\text{cols(kernel)}-1} k(m,t)x(j+m+j(\text{stride}-1),i+t+i\cdot(\text{stride}-1) \tag{1.62}$$

Training methods for the CNN [14]

Back-Propagation method As discussed in subsection 1.1.4, the backpropagation method is the most used one and it allows to compute the weights by considering the *Loss* function. By applying the chain rule already explained, the new values of the weights can be computed by using the **stochastic gradient descent** (SGD). [14] uses modified algorithms to support weights' update in ternary neural networks and compares them:

- **Adam**: learning rule based on the following equation taken from [14], in which the weights are updated as:

$$w_{t+1} = w_t - \alpha \frac{E[\frac{\partial}{\partial w} Loss]}{\sqrt{E[(\frac{\partial}{\partial w} Loss)^2] + \epsilon}} \tag{1.63}$$

  Where E is the mean and α the learning rate.

- **AdaDelta** The update rule by the AdaDelta is given by [14]:

$$h_t = \beta h_{t-1} + (1-\beta)E\left[(\frac{\partial}{\partial w} Loss)^2\right] \tag{1.64}$$

$$v_t = w_t - \frac{\sqrt{s_t + \epsilon}}{\sqrt{h_t + \epsilon}} E\left[\frac{\partial}{\partial w} Loss\right] \tag{1.65}$$

$$s_{t+1} = \beta s_t + (1-\beta)v_t \tag{1.66}$$

$$w_{t+1} = w_t - v_t \tag{1.67}$$

Batch normalization Batch normalization speedups the training and has to be considered in the neural network's structure. In particular, taking into account the binary output of a neuron from [14]:

$$Y = y^{(b)} = f_{act}^{(b)}\left(\sum_{i=1}^{N} w_i^{(b)} x_i^{(b)}\right) \tag{1.68}$$

The activation function is the sign(x). Applying the batch normalization means to add an additional term into the activation function of the neuron itself as reported in [14]:

$$Y' = y'^{(b)} = f_{act}^{(b)}\left(\gamma\frac{Y-\mu_B}{\sqrt{\sigma_B^2+\epsilon}}+\beta\right) \tag{1.69}$$

Where γ, μ_B, σ_B^2, ϵ and β are parameters for BN and are mean, variance of the batch considered, correction term and offset terms respectively. By performing mathematical transformations used in [14], batch normalization's terms can be transformed into biases as follows:

$$Y' = f_{act}\left(\frac{\gamma}{\sqrt{\sigma^2+\epsilon}}\left(Y-\left(\mu_B-\frac{\sqrt{\sigma_B^2+\epsilon}}{\gamma}\beta\right)\right)\right) \tag{1.70}$$

As performed in [14], since the output takes the sign, $\frac{\gamma}{\sqrt{\sigma_B^2+\epsilon}}$ can be ignored if:

$$f'_{sgn}(Y) = \begin{cases} 1 \left(\text{if } Y < -\mu_B + \frac{\sqrt{\sigma_B^2+\epsilon}}{\gamma}\beta\right) \\ -1 \; (otherwise) \end{cases} \tag{1.71}$$

The final equation that defines a neuron's output (considering also that $x_0 = 1$) is the following from [14]:

$$Y_{final} = Y - \mu_B + \frac{\sqrt{\sigma_B^2 + \epsilon}}{\gamma}\beta \tag{1.72}$$

$$= \sum_{i=0}^{n} w_i x_i - \mu_B + \frac{\sqrt{\sigma_B^2 + \epsilon}}{\gamma}\beta \tag{1.73}$$

$$= \sum_{i=1}^{n} w_i x_i + \left(w_0 - \mu_B + \frac{\sqrt{\sigma_B^2 + \epsilon}}{\gamma}\beta\right) \tag{1.74}$$

$$= \sum_{i=1}^{n} w_i x_i + W' \tag{1.75}$$

Training the binary/ternary weights Since the SGD is not usable with binary weights, hidden weights w_{hid} are required, which correspond to the real floating point values of the corresponding binarized weights. During the training phase, only the hidden weights are updated, while binary weights are used at inference. This also happens in ternary networks, in which the only difference is the definition of the sign function as already explained. In order to train a ternary network, the **AdaDelta** algorithm is more suitable, because it generates an higher concentration of "0" weights, reducing the computational overhead and the power required in the CNN. However [14] proposes a comparison between AdaDelta and ADAM optimizers, in order to see the differences in terms of accuracy.

Realization Ternary Weight Binary Input CNN on Embedded Processor

Binary 2D convolutional operation Considering an architecture with multiple channels, the output of the convolutional layer is given by [16]:

$$y_o^{(l)}(j,i) = b_0^{(l)} + \sum_{c=0}^{\#\text{channels}-1} \sum_{m=0}^{\#\text{rows(kernel)}-1} \sum_{t=0}^{\#\text{cols(kernel)}-1} k_{o,c}^{(l)}(m,t) x_c^{(l)}(j + m + j(\text{stride} - 1), i + t + i(\text{stride} - 1)) \tag{1.76}$$

In Figure 1.23 it is shown the computation of $y_o^{(l)}(j,i)$ at coordinates (j,i) for the o-th OFMAP.

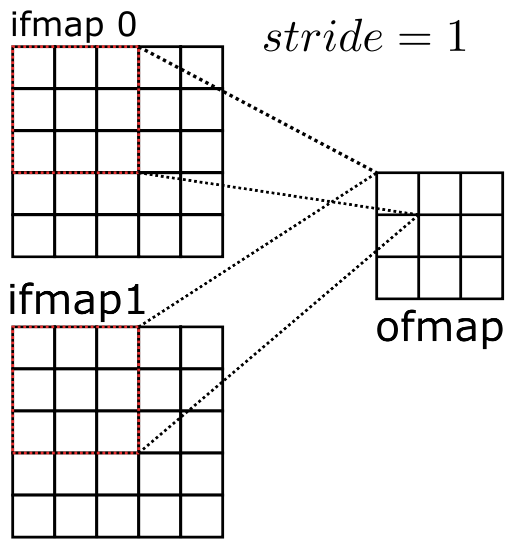


Figure 1.23: Example of the 2D convolutional operation for the ternary weight and binary input

Nx3x3 MAC operations are needed to compute a 2D convolution, so if the feature map becomes larger, more computation time will be required [14].

Results

The results of the network are determined by imposing an initial distribution of the weights (-1,0,1) of 2.5%:95%:2.5% respectively, in order to speedup the training, since most of the connections are set to 0 [14]. ρ is imposed to 0.2 in Equation 1.5 and the dataset used is CIFAR-10 (50,000 images as training set and 10,000 as test one). Two kinds of optimization algorithms are used as described before: Adam and AdaDelta. The architecture used to realize these networks is the **VGG16**, reported in the following figure:

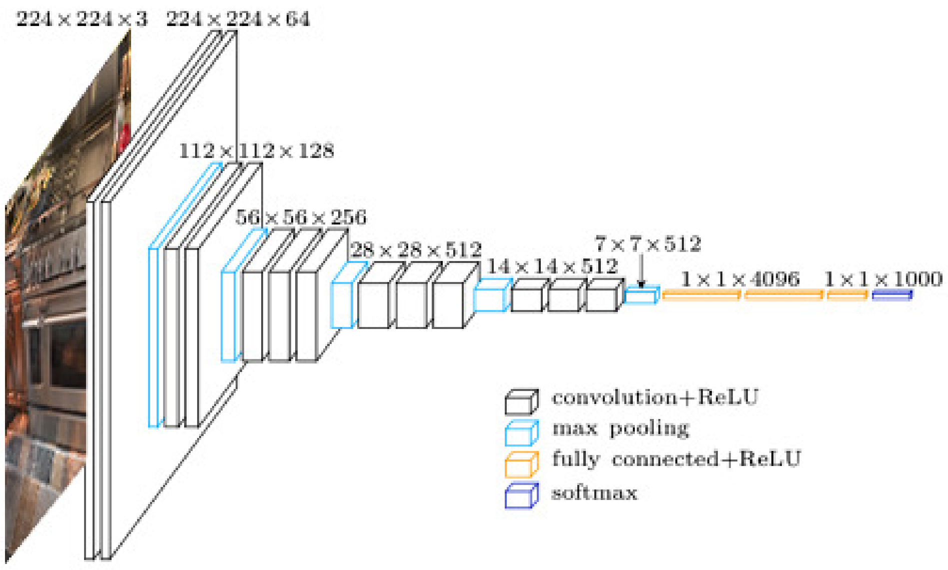


Figure 1.24: VGG16 architecture from [38]. It is composed by 16 layers and it is able to reach up to 70% on top-1 and 90% in top-5 recognition accuracies respectively on ImageNet.

Comparison Ternary Weight CNN with Binary One In order to do a comparison between binary/ternary networks, both of them have been trained by using Adam optimizer in [14]. The parameters evaluated are the error rate and the non-zero weight density, which gives an important indication of how many connections are present in the network. Ternary net has been trained also with AdaDelta, in order to see what are the main differences w.r.t the Adam. Here are reported the results:

Table 1.7: Comparisons between networks from [14] on CIFAR-10 and VGG16

Algorithm used	Network	Error rate [%]	Non-zero weight density[%]
Adam	Binary	19.6	100
Adam	Ternary	17.1	73.9
AdaDelta	Ternary	19	5.3

In Table 1.7 it is possible to see the differences between the optimizers, and in particular the results obtained in the ternary network in terms of accuracy are worse than binary ones in the case of Adam optimizer. But, if AdaDelta is considered, it is possible to see a similar error rate with only 5.3% of non-zero weights: that means that a ternary network trained with AdaDelta is able to achieve a similar result of a binary one but only with 5.3% of active connections [14]. This is a very important result, that indicates the possibility to implement very small, low power and embedded networks without losing accuracy.

Table 1.8: Comparison of required time on ARM Cortex-A53 1.2GHz and 1 GB DDR2 SDRAM from [14]

Convolutional neural network	Time required [s]
Binary weight	6.103
Ternary weight	0.750

1.3 MTJ-Based BNN

An MTJ (magnetic tunnel junction) is a device composed by two ferromagnets separated by a thin insulator [6], in which electrons can flow through by means of a tunnel injection.

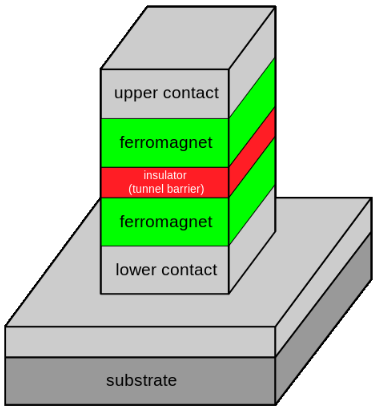


Figure 1.25: Magnetic tunnel junction (schematic) from [6]

The magnetizations of the two ferromagnets determines the intensity of the current flowing: if they are parallel, the current will be higher (R_p: low resistance state), while if they are anti-parallel the current will be lower (R_{AP}: high resistance state).

1.3.1 A Multilevel Cell STT-MRAM-Based Computing In-Memory Accelerator for Binary Convolutional Neural Network [15]

The classical Von-Neumann implementation in which the memory (SRAM for example) and the computations are performed separately, has many problems in terms of delay and power consumption. The key idea to use another type of memories based on a NV (non-volatile memories) approach, consisting on MRAM such as MLC-STT-MRAM, could be a substitute for SRAMs: these are able to perform memory and computing operations and could solve Von Neumann bottlenecks. MLC means that multiple bits can be stored into a single cell and some logical computations can be made inside the memory. [15]

MLC-STT-MRAM

The goal of this implementation is to integrate two bits into a single memory cell. An example from [15] has been reported, that corresponds to a 2x2 array with the corresponding cells' structure:

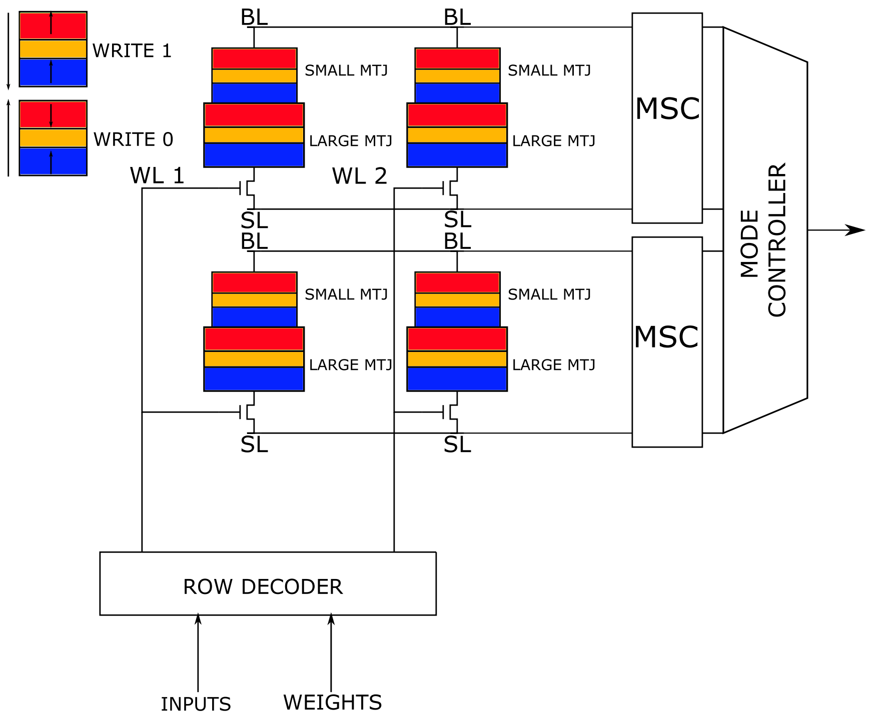


Figure 1.26: Cell structure and example of a 2x2 array from [15]. MSC stands for "modified sensing circuit" and it is able to do some computations based on the current of the source/bit lines. The mode controller is able to choose which operation to perform, while the row decoder handles the word lines. In order to write into the MTJ, a current has to flow through it, and the direction is expressed here. If "1" has to be stored, the current has to magnetize the layers in a parallel way, resulting less MTJ resistance (LRS), so a positive voltage is applied between BL and SL; otherwise, with "0", the magnetizations must have antiparallel direction.

The cells have 4 different configurations (R_{P-P},R_{AP-P},R_{P-AP},R_{AP-AP}) representing all the combinations given by two bits. The I_{SL} has four possible values, since the two MTJs are different from each other ($R_{AP-P} > R_{P-AP}$). The modified sensing circuit is simply composed by a set of comparators that compare the incoming current (I_{sl}) with three different currents I_{ref1},I_{ref2},I_{ref3} with the following relation from [15]:

$$I_{sl,11} > I_{ref1} > I_{sl,10} > I_{ref2} > I_{sl,01} > I_{ref1} > I_{sl,00} \tag{1.77}$$

With them, it is possible to realize some logic functions such as OR,NOR,XOR,NAND and AND.

Working mechanisms

1. Write Mode: This process is realized in two steps. The first one, with a large current, the state of the largest MTJ is changed and the second one if needed, a small current is used to modify the state of the smallest one;

2. Read Mode: to read, the source lines are connected to the comparators of the MSC circuit and the current of the SL is simply compared with the reference currents explained before;

3. Logic Mode: the sense amplifiers can realize some logical operations as already mentioned. Taking for example I_{ref1}, this is the largest current and if the I_{sl} is larger than I_{ref1}, it means that the cell is in the "11" configuration(parallel-parallel), that is translated in into the logical operation AND;

4. Full-Adder mode: it is possible to implement a full adder by considering that:

$$S_n = A_n \oplus B_n \oplus C_n \tag{1.78}$$
$$C_{out} = (A_n \& B_n)|((A_n \oplus B_n) \& C_n) \tag{1.79}$$

   Since the MSC is composed by comparators that realizes basical logic functions, its structure can be extended in order to implement a full-adder. This is done by adding three additional gates.

BCNN accelerator

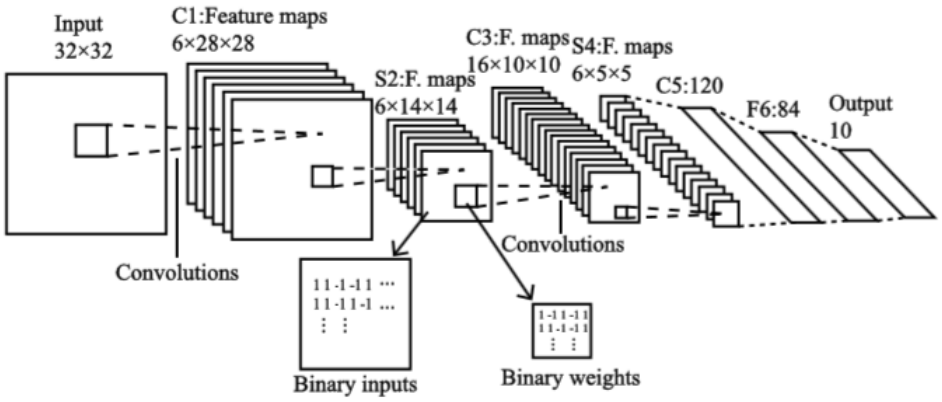


Figure 1.27: CNN architecture used in [15]

Considering a binary neural network, the computation of the convolution is performed by considering the PopCounting of the XNOR as follows:

$$I * W = PopCount(I(B)\&W(B)) \tag{1.80}$$

This can be fully implemented by the architecture described and depicted in the following figure:

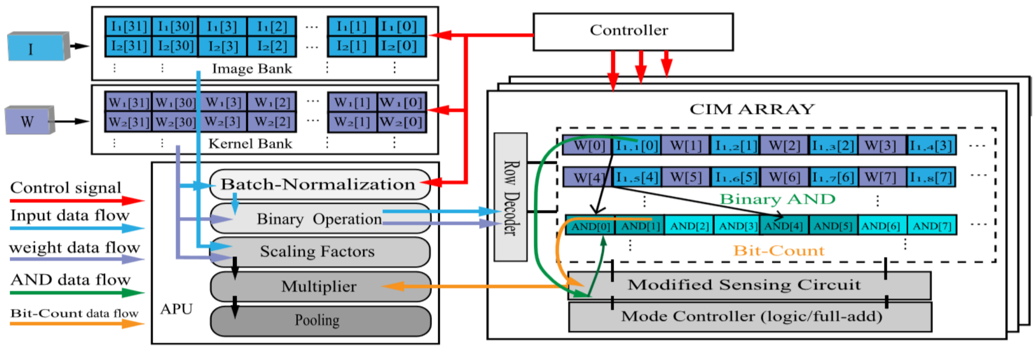


Figure 1.28: BCNN Accelerator from [15]. The logical computations are performed inside the memory array, while other intensive operations, such as batch normalization or scaling factors computations are performed outside the memory in a separate unit.

As it is possible to see, the part of Batch normalization, Binary operation, Scaling factors, Multipliers and Pooling are designed into a separate processing unit, that is not included into the CIM array [15]. The detailed calculation process is the following:

1. Batch normalization is perfermed on the inputs to reduce information loss;

2. Inputs and weights are binarized (sign);

3. Binarized inputs/weights are stored into the CIM to perform in-memory computations. Weights into a CNN are shared and so they are stored into the largest MTJ, while inputs in the smaller one; The colored lines indicated into the Figure 1.28 [15] have the following meanings:

   - Green line: represents AND data flow and its result coming from MSC is written directly into the CIM array;

   - Orange line: represent the popcounting data flow using MSC full-add operation;

4. Tensors I and W are sent to scaling factors to calculate α and K from subsection 1.2.2;

5. Convolutional results and scaling factors are delivered to Multiplier to complete the convolutional layer. At the end of the chain, the results goes into a Pooling layer, in which they will be reduced.

Experimental results

This architecture is tested on MNIST dataset and it is realized with parameters of CMOS 45nm. The resulting energy consumption of this design is only 0.38 μJ, while the cycle time (entire convolution performed) is 27.24 ns. Since the structure realized is a XNOR-NET, here there are reported some useful results:

Table 1.9: Results of the XNOR-NET implementation on the architecture from [15]. The energy reported refers to a convolutional layer with number of kernels that indicates number of OFMAPs coming from the convolution.

Layer	Number of kernels	# convolution operations	Energy Consumption	
			Layer(uJ)	Operation (nJ)
C1	6	4704	0.278	0.059
C3	16	1600	0.094	0.059
C5	120	120	0.007	0.059
F6	84	84	0.0003	0.003
Total	-	-	0.38	-

1.3.2 Energy Efficient In-Memory Binary Deep Neural Network Accelerator with Dual-Mode SOT - MRAM [16]

Introduction

An architecture based on a NVM system is employed in [16], in particular a solution SOT - MRAM that enables the computation in memory with zero standby leakage and very high integration density.

In-memory processing platform

SOT-MRAM This kind of MRAM is based on the scheme represented in Figure 1.29

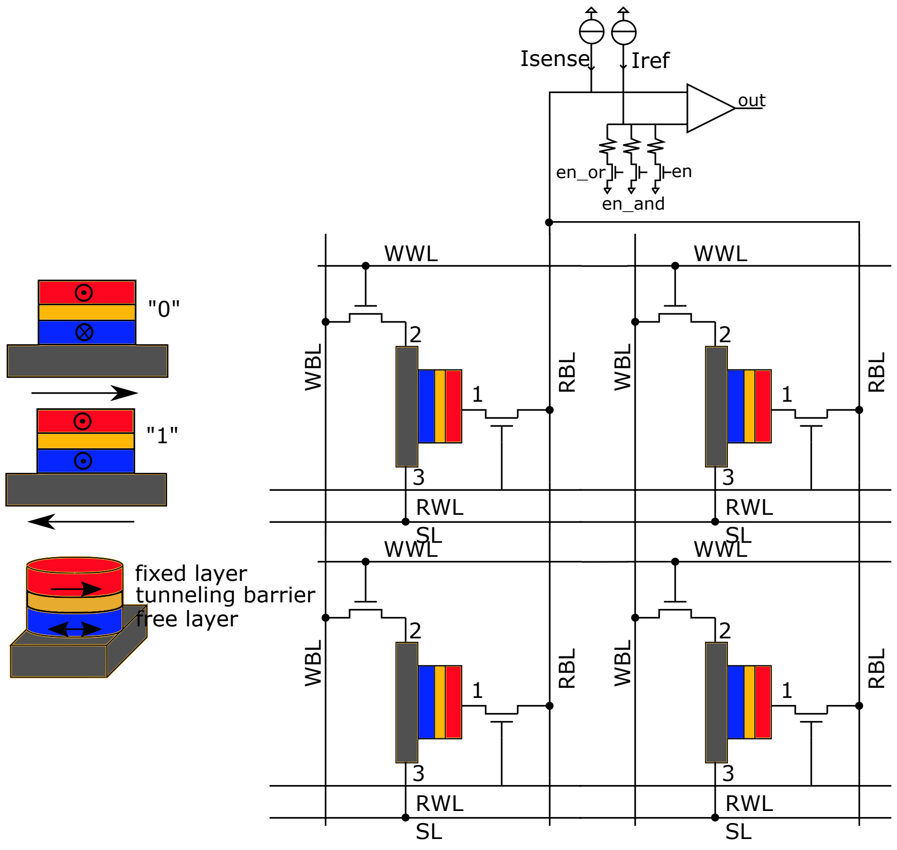


Figure 1.29: SOT-MRAM device structure and an example of 2x2 crossbar array from [16]

As shown in Figure 1.29 , depending on the direction of the current, the MTJ changes its magnetization in the free layer: as a consequence two states are possible: antiparallel and parallel. These two states, as already said, correspond to two resistances (HRS and LRS respectively). In Figure 1.29 it is shown the structure of the array cell, which has respectively RBL,WWL,WBL,SL that enable in-memory operations, like write/read and computation. Also here the data are fetched from the cells in form of current, and so the current on the RBL is fed to a current sense amplifier that gives the corresponding logical output.

Memory write [16] In order to write a data inside the cell, a write current has to be injected in the heavy metal substrate. The current has to flow from 2 to 3 terminals of the MTJ (or viceversa), in order to obtain the different magnetizations' states (Figure 1.29). So the operational steps are:

1. WWL is activated by row decoder;
2. SL is grounded;
3. Voltage driver on the WBL is set to positive (/negative) in order to obtain HRS (LRS).

Memory read [16] To perform a read operation, a read current has to flow from 1 to 3 terminals of the MTJ. The operational steps are the following:

1. Sense voltage generated in the SA used to read the values from the cells;
2. SA compares V_{sense} (which is determined by the current on the RBL multiplied the resistance of the MTJ) with V_{ref}, by selecting one of the possible enables;
3. SA's output is high when path resistance is higher than ref (R_{ref}) resistance.

Computing mode [16] The computing mode is performed by selecting two or multiple rows. If multiple rows are selected, the equivalent resistance on the RBL is given by the parallel of the individual resistances in the selected cells. This equivalent resistance is then compared with another specific reference, which has been selected by the enable signals (for instance EN_{OR},EN_{AND}) . In this case, the reference is chosen properly to obtain a SA output, which corresponds to the selected logic function. In particular:

- AND logic function: R_{ref} midpoint of $R_P//R_{AP}$ and $R_{AP}//R_{AP}$, and so only if both cell resistances are HRS (corresponding to "11"), the output will be high;
- OR logic function: R_{ref} midpoint of $R_P//R_P$ and $R_P//R_{AP}$.

BCNN accelerator [16]

To demonstrate how this architecture well-suites the computations inside a BCNN, [16] uses AlexNet architecture, which has 5 convolutional layers and 3 fully connected layers. In particular here it is adopted the variant AlexNet BCNN which is composed by 8 convolutional layers (no fully-connected layers used), with the first and the last that are not binarized. Each convolutional layer corresponds to Batch Normalization, Scaling factor computation, Multiplier, Pooling (handled by an external DPU) and Sign function, Binary-AND, Bitcount (handled by CIM).

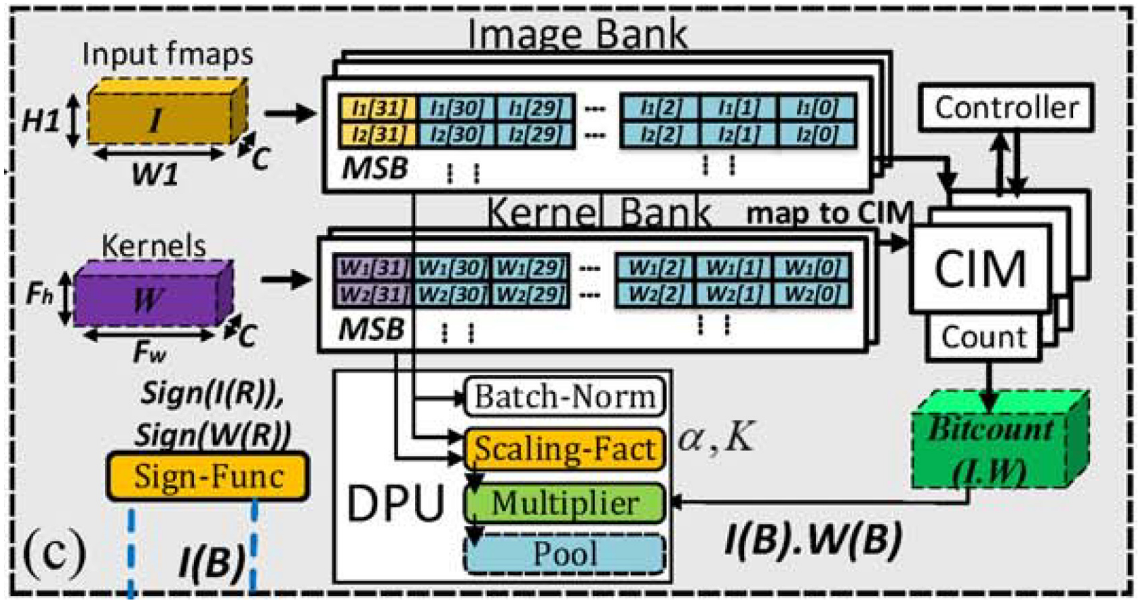


Figure 1.30: Inputs and weights are in ImageBanks and then it will be computed the binary convolution by performing an In-Memory AND logic operation followed by a Bitcounting. Source: [16]

Results [16]

Energy, area, delay and Memory usage estimation The computation energy, area, execution time and memory usage of different implementations of AlexNet BCNN/CNN on ImageNet dataset are tabulated:

Table 1.10: Memory usage of AlexNet DP (Double precision), SP and BCNN from [16]

Architecture	Required memory [MB]	Energy [uJ/img]	Area [mm^2]	Execution time/img (ms)
AlexNet DP	476.4	-	-	-
AlexNet SP	238.2	-	-	-
AlexNet BCNN (this)	39.7	310.42	5.28	10.7

About memory usage, there is a significant improvement w.r.t AlexNet DP and AlexNet SP (x12 and x6 respectively), because the binary architecture occupies less space in memory.

1.3.3 A Logic-in-Memory Design with 3-Terminal Magnetic Tunnel Junction Function Evaluators for Convolutional Neural Networks [17]

Introduction

[17] uses magnetic DW devices, which are able to perform logical and memory operations: the concept is based on a variable resistor that is read and written electrically in such a way that complex functions can be obtained, if the device is designed with particular conditions. The variable resistor is an MTJ and when there is a moving DW in the free layer, the MTJ resistance value R_{MTJ} can assume one of many values between R_P and R_{AP} depending on the position of the DW. The reference technology is the SOT, which has better motion properties than STT (DW requires less current to move). The positions of the DW are discrete and limited and they determine how many resistive values can be obtained from the MTJ function evaluator. If a sufficiently high resolution of resistive values is available, particular functions (activation functions like sigmoid or hyperbolic tangent) can be obtained by using MTJ as function evaluator, avoiding the usage of very complex digital circuits.

The MTJ function evaluator [17]

In [17] there is a mathematical explanation of how an MTJ can be designed to obtain a particular function in output, based on the DW motion. The resistance of an MTJ is defined as follows from [17]:

$$R_{MTJ} = R_P \left(\frac{x_0(I_{IN})}{L} \right) + R_{AP} \left(1 - \frac{x_0(I_{IN})}{L} \right) \qquad (1.81)$$

With $x_0(I_{IN})$ the final position of the DW w.r.t. the input current I_{IN} and L is the length of the MTJ. In [17], it is considered the domain wall velocity and derived the equation of the MTJ's width, assuming also that the current is applied in a finite pulse t_0. The resulting $w(x)$ equation is reported from [17]:

$$w(x_0) = \frac{\eta t_0}{d}\left(\frac{dI}{dx_0}\right) \tag{1.82}$$

$I(x_0)$ is the inverse function of $x_0(I_{IN})$ and d is the thickness of the MTJ. In [17] it is used a shifted sigmoid function, and so the goal is to obtain the DW position's equation that is proportional to the sigmoid. The only way to do this, is to find Wthe width equation, that defines the shape of the MTJ. Considering the shifted sigmoid function from [17]:

$$x_0(I) = x_A tanh\left(\frac{I - I_1}{I_2}\right) + x_B \tag{1.83}$$

Now by using the equation of the sigmoid (Equation 1.83), and applying it to the width equation (Equation 1.82) by doing the inverse derivative, an equation for $w(x)$ can be obtained. The MTJ designed will have a resistive behavior proportional to the shifted sigmoid.

Logic in memory system design [17]

Crosspoint array The memory array is organized as a crosspoint composed by 1T1R cells. The output coming from one column of the crosspoint array is the following:

$$I_j = \sum_i V_i G_{(i,j)} \tag{1.84}$$

The corresponding output voltage coming from the MTJ function evaluator is:

$$V_{OUT,j} = f(I_j) \tag{1.85}$$

The crosspoint array structure and architecture is defined in Figure 1.31:

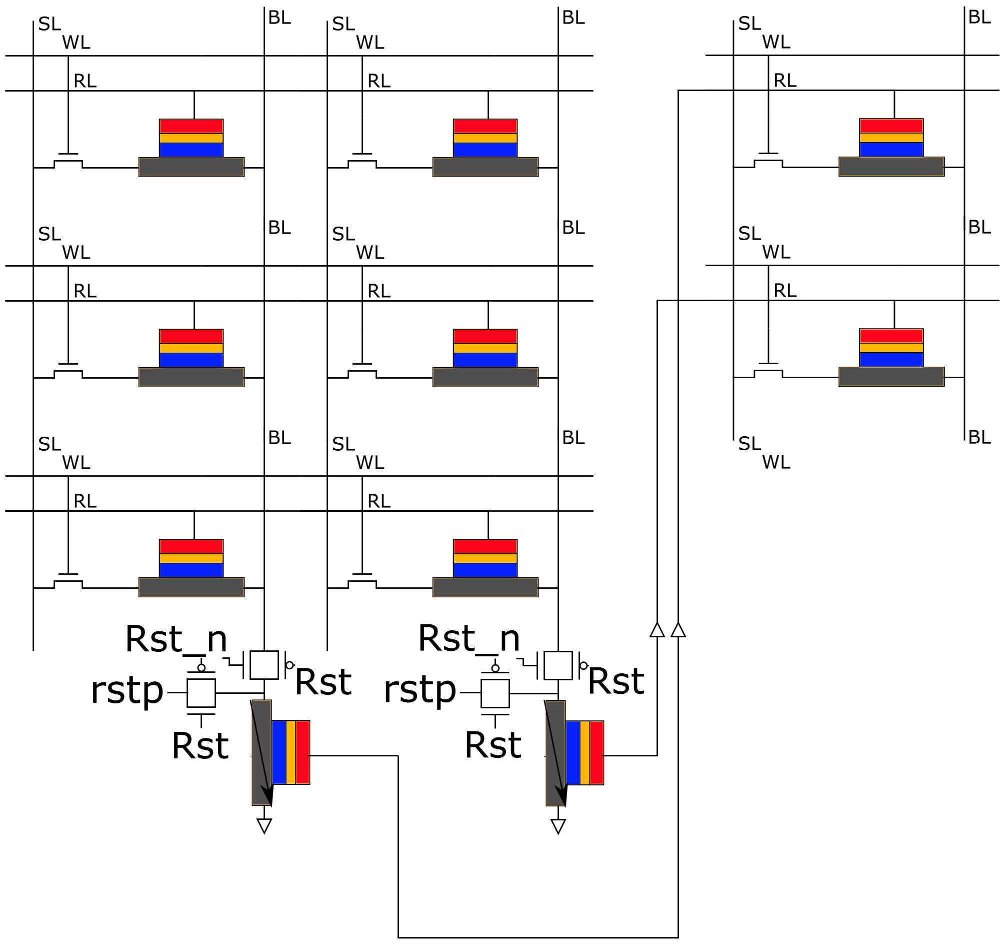


Figure 1.31: Crosspoint array architecture from [17]; two different types of MTJs are used in [17]: the synaptic MTJs are the classical ones, with two possible values of resistances(R_P and R_{AP}); while the thresholding MTJs are the ones discussed so far. The last MTJ (indicated by an arrow) acts as function evaluator and it implements the activation function of the neuron. This crossbar can be seen as an array of variable resistances.

The network can be larger, and this configuration allows the connection between multiple arrays simply by taking the output of the function evaluator MTJ of the previous array, without the need of using ADC/DACs, speeding up the system. Connections between CNN subarrays are programmed with multiplexers.

Perceptron mode [17] The steps to use the architecture in perceptron mode are:

1. The function evaluator MTJs are reset by imposing $RST_j = 1$ and a Domain

wall is injected with RSTP;

2. Both $RST_j = 0$ and $WL_i = 0$. On RL_i there is an input voltage that allows the MTJ to set its resistance;

3. $BL_j = 0$: the output is passed to the next layer or, if in the final cycle, sense the thresholding MTJ.

Memory mode [17] For reading operation:

1. One row is selected with $RL_i = 0$, others to 'Z';

2. $WL_i = 0$ set;

3. Sense the resistance MTJ on BL_j.

While for writing:

1. Write one row $WL_i = 1$, $RL_i = Z$, $SL_j = 0$;

2. Inject a DW current with BL_j.

Results

The architecture has been implemented in [17] with CMOS 45nm and magnetic tunnel junction process. This implementation is able to save energy up to 50x w.r.t a CPU-Based CNN. In this part they will be presented the results coming from the architecture:

Table 1.11: Results for 2 convolutional layers from [17]

Operations/Parameters		Feed forward operations	Memory write	Memory read
Power	Static [uW]	68.6	68.6	68.6
	Dynamic [nW]	15.4	10.7	129000
Latency per layer[ns]		4	4	2

1.4 RRAM Based

RRAM is a non-volatile random-access memory that changes its resistance based on the voltage applied across it [47]. A dielectric could conduct, if the voltage applied is sufficiently high to form a conduction path through it (**dielectric breakdown**) which in this case is temporary and reversible because of the materials used. Once the conduction path is formed, it may be reset (broken, resulting in high resistance) or set (re-formed, resulting in lower resistance) by another voltage.
These types of cells can be implemented into a 1T1R configuration as following:

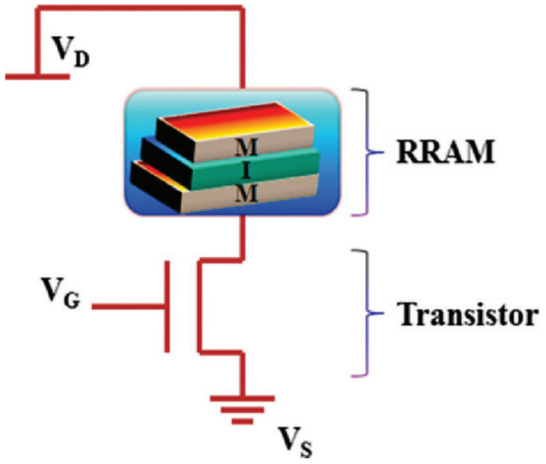


Figure 1.32: 1T1R configuration from [39]

Generally a V/2 scheme is adopted to avoid write disturbance, that is, V_{set} or V_{rst} is applied across the selected cells and $V_{set}/2$ or $V_{rst}/2$ is applied across all the unselected cells. For read operations, a voltage smaller than the threshold, V_{read} is applied on the selected cell: current coming from it, is compared with a reference to determine the output. It is possible to have an undesiderable current flow due to the non-isolation of cells, which is known as the sneak current: 1T1R is able to reduce this drawback.

1.4.1 The application of Non-volatile Look-up-table Operations based on Multilevel-cell of Resistance Switching Random Access Memory [18]

Introduction

In the approach presented by [18], the MLC is used, enabling multiple bits per cell. The resistance of the RRAM can be easily switched by varying pulse duration/amplitude of write voltage. In [18] it is discussed a ROM LUT-based implementation, where outputs are pre-stored and the input bits are used as the address, by means of a decoder to access to them. A novel approach to implement a multiplier is presented by [18], which is based on a LUT in-memory model.

Circuits design and implementation [18]

As it is possible to see by looking in Figure 1.33, the cells are organized in a crossbar configuration.

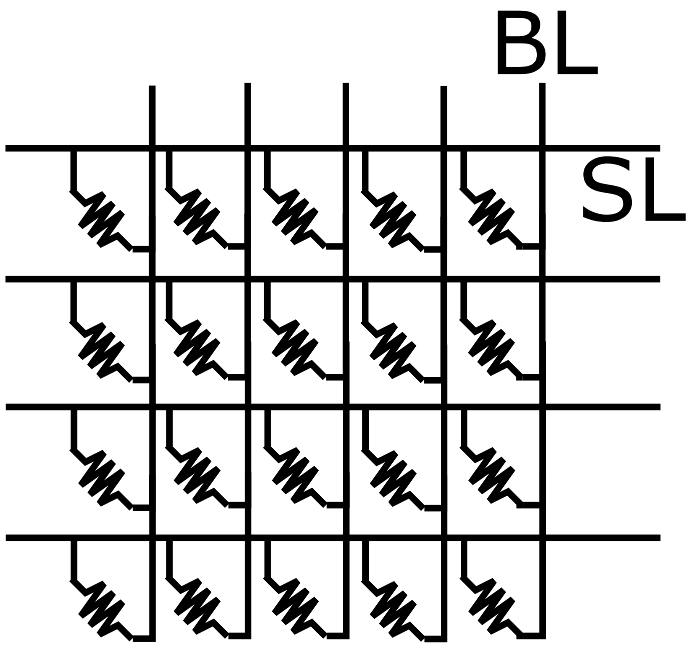


Figure 1.33: Crossbar array cell's organization from [18]. Each memory cell is a RRAM.

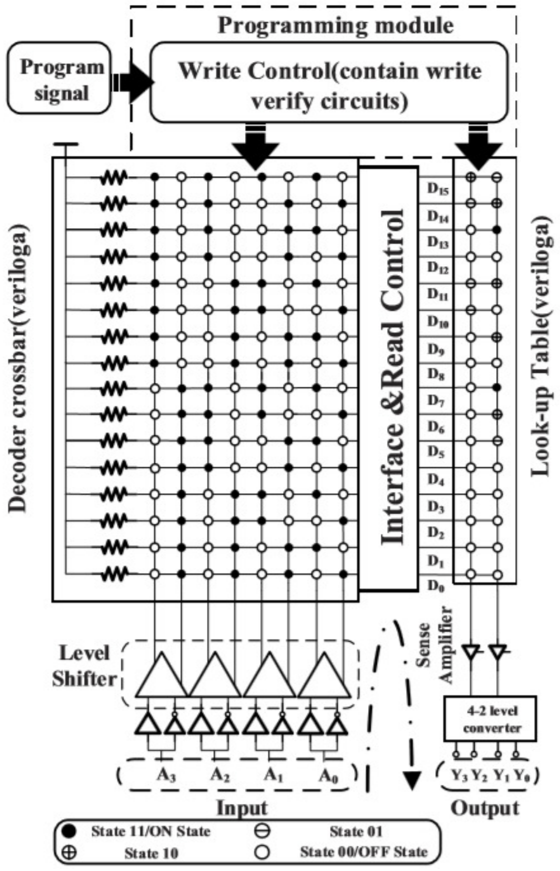


Figure 1.34: Architecture of the multiplier based on MLC RRAM from [18]

The circuits in Figure 1.34 is composed by:

- Row decoder: selects the specific row for read/write;
- LUT RAM on the right, stores the multiplication results (precharged);
- The MLC RRAM consists on a set of resistances which assume the following values:
  - → ”11” corresponds to 1kΩ;
  - → ”10” corresponds to 10kΩ;
  - → ”01” corresponds to 100kΩ;
  - → ”00” corresponds to 10MΩ.

- The programming block programs the two crossbars (writes values of the multiplier in the arrays for example);
- The write control circuit handles the multiplier in two steps:
  1. Initializes the two crossbars to "00" (high impedance) and verify the written values through write-verify circuit;
  2. Write data into the crossbars to accomplish a specific digital function, and verify the input data via write-verify circuit.

2x2 multiplier output bits will be stored into the LUT at the right hand of the Figure 1.34 and the 4-16 line decoder will address the LUT. The input interface (at the bottom of Figure 1.34) contains inverters (that are able to do the logic inversion if needed), buffers and level shifters, because the voltage level for a MLC RRAM is different from the CMOS's one. The row decoder is used to address the LUT: the columns of the crossbar array are used as inputs, while the rows as outputs that produce a read voltage. The interface circuit controls the read voltage on the selected address of the LUT, and the corresponding results pass through sense amplifiers and 4-2 level converters.

Simulation

[18] implements three types of multiplier (4x4,8x8 and 16x16 respectively) based on a CMOS 65nm process. Also a 1bit/Cell multiplier has been implemented to compare the results between the two different approaches. In the following table are reported the results:

Table 1.12: Results of the LUT-based multiplier from [18], with different configurations.

Multiplier	RRAM Type	Delay[ns]	Area[μm^2]
4x4	1bit/cell	1.01	166.87
	2bit/cell	1.21	137.86
8x8	1bit/cell	1.03	738.23
	2bit/cell	1.21	650.73
16x16	1bit/cell	1.06	2811.25
	2bit/cell	1.24	2460.75

1.4.2 XNOR-RRAM: A Scalable and Parallel Resistive Synaptic Architecture for Binary Neural Networks [19]

Introduction

[19] proposes RRAM based architecture, that is able to implement the XNOR-Bitcounting operations, enabling the realization of very deep binary neural networks. Both MLP and CNNs are implemented in [19], so the datasets used are MNIST and CIFAR-10 respectively. Also a novel architecture based on a parallel reading is employed in [19], which results more efficient than the sequential one. The structures of the network used in [19] are the following:

- MLP: 784-512-512-512-10 (MNIST dataset). The accuracy w.r.t floating point implementation passes from 99.0% to 98.77%;
- CNN: 6 convolutional layers and 3 fully connected layers (CIFAR-10 dataset). The accuracy is 88.47 % and in floating point implementation is 89.98%.

This architecture is implemented with 65nm node.

RRAM Based Synaptic Array [19]

In the following figure, it is shown the cell structure based on 1T1R implementation:

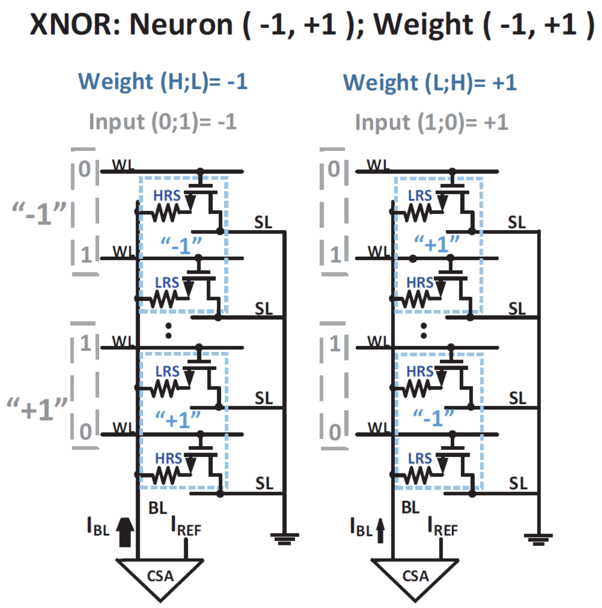


Figure 1.35: Bit cell structure from [19]

In Figure 1.35:

- -1: top/bottom cells are in HRS/LRS respectively;
- +1: reverse pattern.

For the WLs instead, the following representations are used:

- -1: input pattern (0,1);
- +1: input pattern (1,0).

The output current coming from the cell depends on the input pattern and the cell configuration For example:

- Input vector is -1 (0,1);
- Cells selected with weight -1;
- The activated row is LRS causing a large current, seen as "1" (XNOR);

If multiple WLs are selected in parallel, the LRS-cells will dominate the bitline current. So I_{BL} is proportional to the number of LRS-cells in the column, realizing the pop-counting. For example:

- Column's length = 64;
- $I_{ref} = 32LRS$ activated cells for the sense amplifier;
- If $I_{BL} < I_{ref}$ the output is -1 that represents the neuron's activation function.

Two kind of approaches can be made to realize the previous functions: in sequential approach, only one WL is activated per time and during the read, $V_{BL} = GND$, current sense amplifier injects a current on the bitline that will be compared with I_{ref}. This procedure is done for all the rows in the crossbar array, so MAC units and registers are needed. The final sum is sent to a comparator which generates the digital output. In the parallel architecture, a WL switch matrix enables multiple WLs usage simultaneously based on the input vector. The most important component in this architecture is the current sense amplifier, because it can be affected by unwanted offset that degrades the sensing pass rate[19]. This is worse when the bitline current is higher. The design of the current sense amplifier has to consider that the offset could change completely the output of a neuron and, consequently, producing a wrong result. Considering an array size of 512x512, the accuracy is only 15.04%: one of the possibilities to reduce the offset problem is to divide the initial array into subarrays in order to reduce the current I_{BL} and to perform a non-linear quantization on the partial sums. All of these considerations are discussed in detail in [19].

Array Partitioning After array partitioning, each array generates a partial sum that has to be added with the other ones. A partial sum has to be very precise, because it affects the whole accuracy in the final sum: ADC-like MLSA carries out partial sums in fixed point. Another important parameter is the number of bits of MLSA, which heavily influences the accuracy: bit-level of 2- bit imply >98% accuracy for MLP on MNIST, when sub-array dimensions are 32x32 or 64x64.

Benchmark results on MNIST and CIFAR-10 [19]

[19] reports also area, latency and energy efficiency per each subarray size:

Table 1.13: Parameters of the different architectures from [19]

Subarray size	MLSA bit-width	Area $[mm^2]$	Latency [ns]	TOPS/W
64x64	3	0.0832	12.7	81.79
128x128	3	0.047	13.69	141.18

MNIST In this part the results obtained in [19] from the benchmarks will be analyzed. In particular, considering the variations of MLSA offset and RRAM cell resistance (Gaussian distribution with a mean of $200k\Omega$ and a standard deviation of $3k\Omega$ from [19]), it is possible to demonstrate that the sensing pass rate is small when the bitcounting value is close to a sensing reference. When the bit-counting is far enough from a sensing reference, the pass rate can achieve 100%. There are reported the results in terms of accuracy on MNIST dataset from [19]:

Table 1.14: MNIST-based implementations results from [19]

Implementation	Sub-array size	MLSA Bit level	Network structure	Dataset	Accuracy[%]
XNOR-RRAM	64x64	2	MLP	MNIST	95.81
XNOR-RRAM	64x64	3	MLP	MNIST	98.56
XNOR-RRAM	128x128	3	MLP	MNIST	98.43
BNN Algorithm	-	-	MLP	MNIST	98.77
NN Algorithm (FP)	-	-	MLP	MNIST	99

The total number of subarrays used in both cases (64x64 and 128x128) are 136 and 36 respectively from [19]. With these data, it is possible to determine the total area as:

Table 1.15: Total latency of MLP based on MNIST from [19]

# employed	Subarray size	MLSA bit-width	Area $[mm^2]$
136	64x64	3	11.315
36	128x128	3	1.686

CIFAR-10 The results on CIFAR-10 dataset are reported in the following table:

Table 1.16: Results on CIFAR-10-Based implementations (CNN) from [19]

Implementation	Sub-array size	MLSA Bit level	Network structure	Dataset	Accuracy [%]
XNOR-RRAM	64x64	3	CNN	CIFAR-10	86.12
XNOR-RRAM	128x128	3	CNN	CIFAR-10	86.08
CNN Algorithm	-	-	CNN	CIFAR-10	88.47
CNN FP	-	-	CNN	CIFAR-10	89.98

The convolutional neural network has the following structure in details:

Table 1.17: CNN structure from [19]

Layer	Type	# IFMAP	# OFMAP	kernel size	# subarrays 64x64	# subarrays 128x128
1	Convol	3	128	3x3	-	-
2	Convol	128	128	3x3	36	9
3	Convol	256	256	3x3	72	18
4	Convol	256	256	3x3	144	36
5	Convol	256	512	3x3	288	72
6	Convol	512	512	3x3	576	144
7	F.Conn	8192	1024	-	2048	512
8	F.Conn	1024	1024	-	256	64
9	F.Conn	1024	10	-	16	8
Total	-	-	-	-	2436	863

The parameters related to this network are reported in the following table:

Table 1.18: Parameters on the CNN based on CIFAR-10 from [19]

# employed	Subarray size	MLSA bit-width	Area [mm^2]
2436	64x64	3	202.67
863	128x128	3	40.5

The best solution is the 128x128 array size with 3 bit-level MLSA in both cases (MNIST and CIFAR-10) and in particular for the last one, the energy efficiency is 141.18 TOPS/W from [19].

1.4.3 MAGIC-Memristor-Aided Logic [20]

Introduction

A memristor is a device which changes its resistance depending on the current flowing through it. It is possible to realize some logic functions considering that memristors have two different resistance states which can be used as "1" and "0": if the current is higher, then it is considered logic "1", otherwise logic "0".

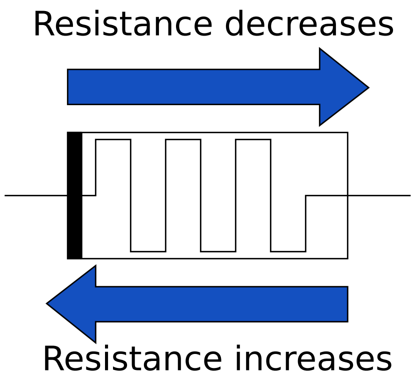


Figure 1.36: Memristor behavior from [20] depending on the current flow direction.

A NOR Gate can be implemented as shown in the following figure:

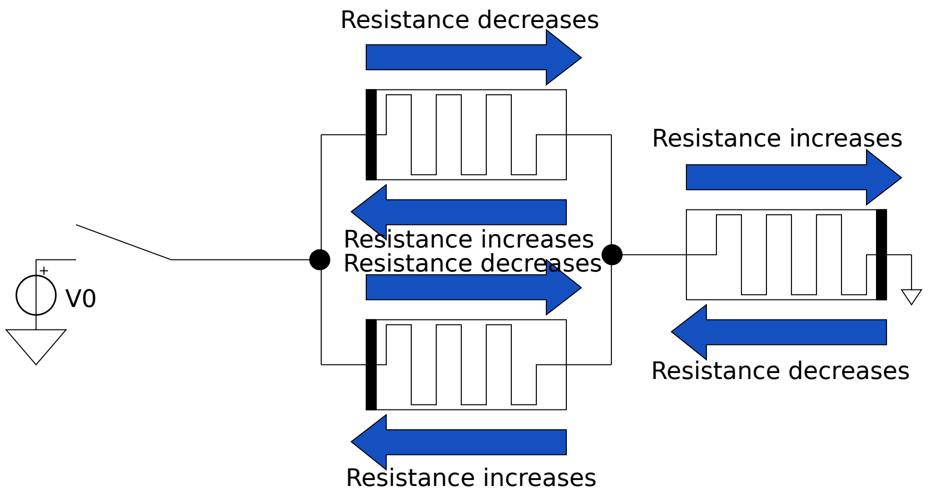


Figure 1.37: NOR Gate with memristors from [20]

Considering Figure 1.37, the two parallel memristors are considered as the inputs, while the last one (on the right) is the output. The operations to perform a NOR can be summarized as follows:

1. Initialization of the output memristor low resistance (logic 1);
2. A voltage V_0 is applied. If the input memristors are in high impedance, the current flowing through the output resistor is not sufficient to change the state, so it remains low (logic 1). If the input combination is different (10 or 01), the current will be higher than the threshold of the output memristor, and so it changes its state;

For the input combination "00", the voltage across the output memristor should be lower than $V_{T,OFF}$. In the other cases, it should be higher than this value. When an input memristor is "0", the voltage applied V_0 can change the input to logic "1" in the meanwhile. So, V_0 has to be less than the threshold voltage $V_{T,ON}$: V_0 has to be designed properly, as indicated in [20].

In-Memory structure

These memristors can be placed inside a crossbar array in order to be integrated in an in-memory solution.

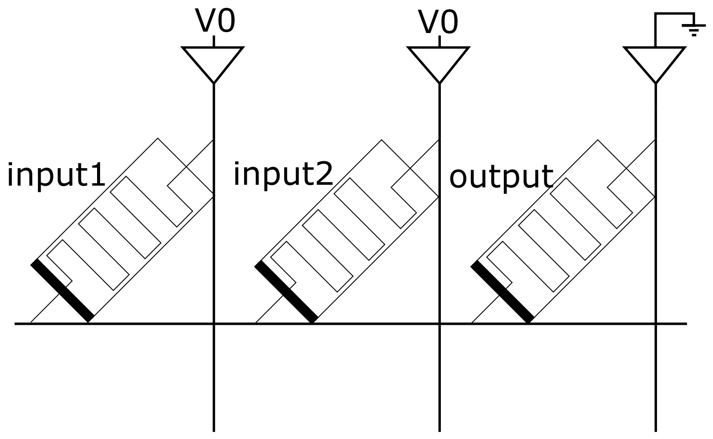


Figure 1.38: Memristor-based crossbar array: configuration for NOR logic gate from [20]

In Figure 1.38, the memristors are organized in a crossbar array. The NOR function is implemented by imposing on the inputs the voltage V_0 and on the output GND. The result of the NOR function is a current that flows through the row lines into a proper analog circuit, which translates the input current into a logical state. From [20], it is reported the delay of the NOR operation, which depends on V_0: if $V_0 = 1V$, it is equal to **1.3ns**, considering the slowest computation. This approach enables also the realization of other logic gates.

1.4.4 Mixed-precision architecture based on computational memory for training deep neural networks [21]

Introduction

[21] proposes a mixed approach based on a crossbar array of memristors and an high precision digital unit, which is able to perform both in memory and high precision computations. In particular, the second ones are useful in the training phase, in which an high grade of precision is required and so the weights are not binarized. The modifications of the weights in the crossbar array are obtained by changing the resistances by means of programming impulses.

The architecture

The architecture of the system is presented in the following figure from [21]:

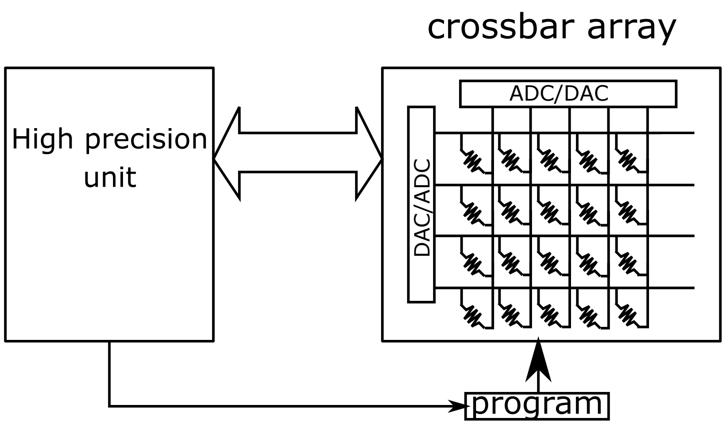


Figure 1.39: Principle scheme of the mixed precision architecture. Source: [21]

In Figure 1.39, the crossbar array on the right performs multiplications and stores the weights and on the left the high precision computational unit trains the neural network. The working principle is the following:

1. The inputs (neurons' activations) are fed to the crossbar array from the high precision unit and converted in analog voltages by means of DACs;

2. The crossbar array performs its evaluations and each column carries out a current, which is proportional to the multiplication between the weight stored in the crossbar (considered as a conductance) and the input voltage;

3. The currents are then converted into digital by means of ADCs. The digital vector is the result of the computation.

The same crossbar can be used to perform the backpropagation, in which the errors are converted in voltages. All the considerations done in subsection 1.1.4 are still valid, so the learning rule is applied also in [21]. Since the conductances are subjected to variations, the update is performed only when the accumulated weights updates reaches a multiple of the smallest and reliably achieved change of the conductance itself [21].

Neural network structure The MLP network in [21], is realized considering MNIST dataset. Its structure is 784-250-10 and the inputs are images of 28x28 size. The hidden layer and the output layer have **sigmoid** as activation function. The NN is trained for 10 epochs and the corresponding floating point implementation (64 bit) achieves an accuracy of **98%** with SGD.

Inaccuracies The inaccuracies arising from this architecture are a lot. Starting from the conductances, they do not have precise values because they depends on the physical properties of the material used, in particular they are subjected by granularity, stochasticity and asymmetric conductance responce. The weights update process is heavily influenced by the conductances and also the computations regarding the output classification/weights update depend on these variations, in fact, as mentioned before, the crossbar array is also used to compute multiplications regarding the backpropagation. The noise (in particular the read noise) is another important factor that has to be considered, because it degrades the accuracy [21]. Also the DAC/ADC inaccuracies influence the system behavior, and in particular, choosing a small resolution brings to a very low values of accuracy (about 50% from [21]). In [21] it is chosen 8-bit resolution for both DAC/ADCs, in order to avoid degradation. The architecture has been tested and trained, taking into account all these variations. After 10 training epochs, the architecture reaches **97.78%** of accuracy.

1.4.5 A hardware neural network for handwritten digits recognition using binary RRAM as synaptic weight element [22]

Introduction

[22] proposes a binary neural network based on RRAM devices, which implements a 784-10 MLP network for handwritten digit recognition (MNIST dataset). The network is realized as a resistive crossbar array, in which the columns are the outputs and the rows the inputs. The architecture achieves 81% accuracy with a custom training procedure, instead of the classical SGD based one.

Network structure

The network structure is the classical crossbar array, in which the input is a MNIST image which has been vectorized (from 28x28 to 784x1). The greyscale input values have been adapted to the voltage range (0;0.1V) and the outputs are currents which are compared with each other. The classification in output is given by the maximum current incomings from the columns. The following figure reports the structure from [22]

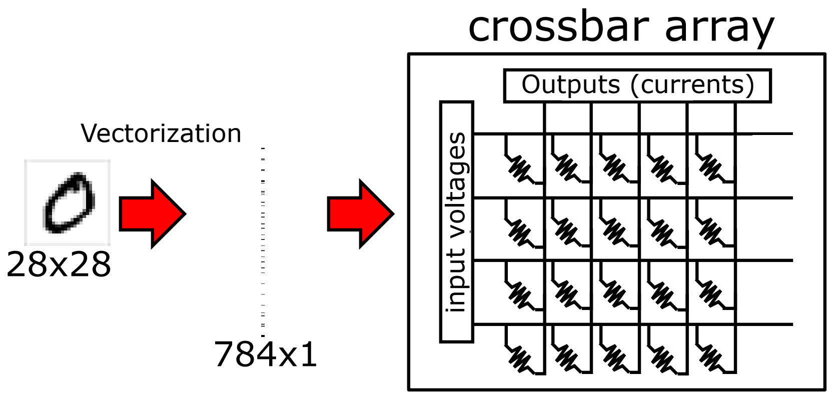


Figure 1.40: Network structure from [22]

Training strategy [22]

The memristors can modify their resistances according to the applied voltage: in particular, if the voltage is positive and larger than a certain threshold $V_{T,on}$, the resistance of the memristor becomes LRS and so the bit stored is a zero (Set). Otherwise, if the applied voltage becomes negative and less then $V_{T,off}$, the resistance will be reset to HRS. To train this network, the recognition result is considered: if it is not correct, the corresponding column that gives the result and the other one which is correct will be "stochastically reset" and "stochastically set" respectively, in order to decrease the current in "recognition result" node and increase the current in expected node. The "stochastic reset" process is performed by a sweeping increase voltage in the output node, in order to generate a negative voltage across the interested memristors. If any is reset, the applied voltage is removed. The threshold voltages of the memristors were chosen randomly from [22].

Results

Table 1.19: Results from [22]. When more than 1 arrays are used, the recognition result is improved. They are used in parallel and the output is evaluated in the same way explained before.

Total images	Training images	Dataset	Array size	# arrays	Accuracy[%]
60000	10000	MNIST	784x10	20	81
60000	10000	MNIST	784x10	1	62

Also the network robustness has been evaluated in [22] w.r.t Vset/Reset changes, and they demonstrated that it works well also when 50% of the RRAM is disabled.

1.4.6 Challenges of emerging memory and memristor based circuits: Nonvolatile logics, IoT security, deep learning and neuromorphic computing [23]

Introduction

[23] explores the NVM technology and its real applications and proposes a comparison between different realizations (such as RRAM, ReRAM, Memristors,PCM, STT etc). [23] has been reported in this section because it provides very interesting considerations on memory technologies.

Write voltages of emerging NVM

Figure 1.41 shows all recent emerging NVM technologies which are better in terms of performance than Flash:

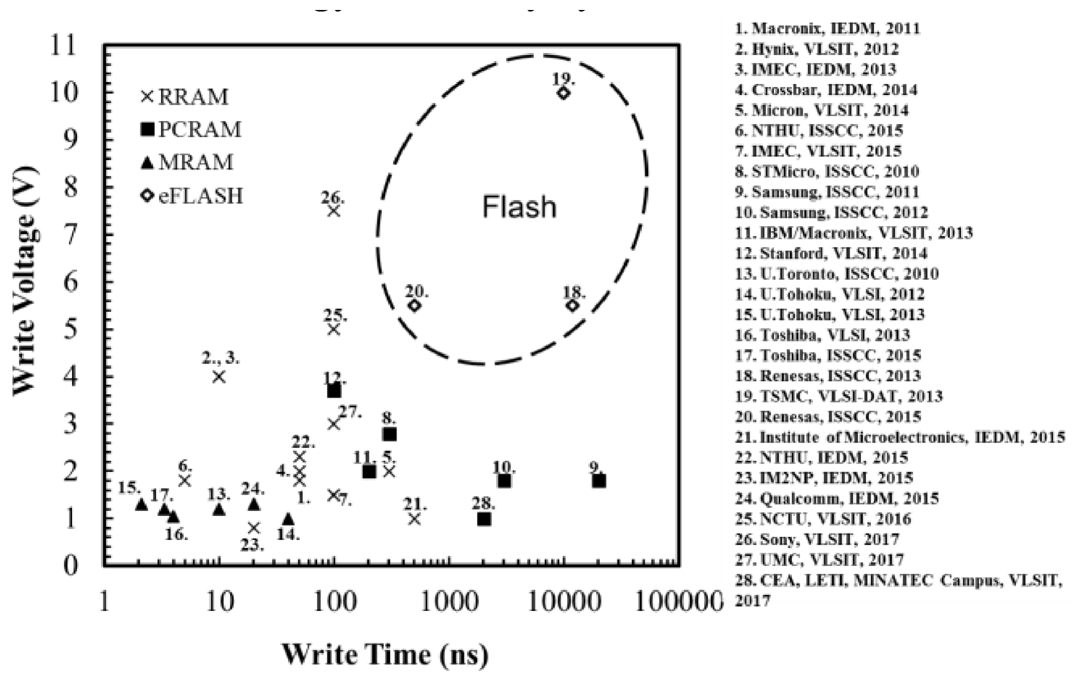


Figure 1.41: Write voltages of different technologies. Source: [23]

The analyzed parameters are the write voltage and the write time and, as it is possible to see, only three kind of NVMs are included in the flash area. So resistive-based memories are very good in terms of energy efficiency, because the required write voltage and write time are less than the flash memories.

NvLogics: non-volatile computational units

In a classical system, when the power is turned off, the logic circuits have to move their data to NVM, in order to keep them saved for the next operations. The systems discussed in [23] instead, use local NVM inside the computational part and since the new technologies based on RRAM, MTJs etc enables fast writing data at low power consumption, the operation of switching off-on a circuit is not so expensive. Another important parameter that has to be considered is the resistance ratio of the resistive memories: if it is small, there is not an evident difference between on-off state and so circuits able to sense small resistance difference has to be employed, considering also the presence of the sneak current between cells and other leakage currents. These last parasitic effects are reduced by using solutions such as 1T1R cells or similar. In the following figure, it is reported a simplified implementation of a 3-2 network with RRAMs in 1T1R configuration from [23]: the products are performed by the array itself and the weights are stored into the RRAMs, so by applying the binary inputs V0,V1,V2, the corresponding word line is enabled and the current flowing through the bitline is sensed by a sense amplifier which generates the binary output.

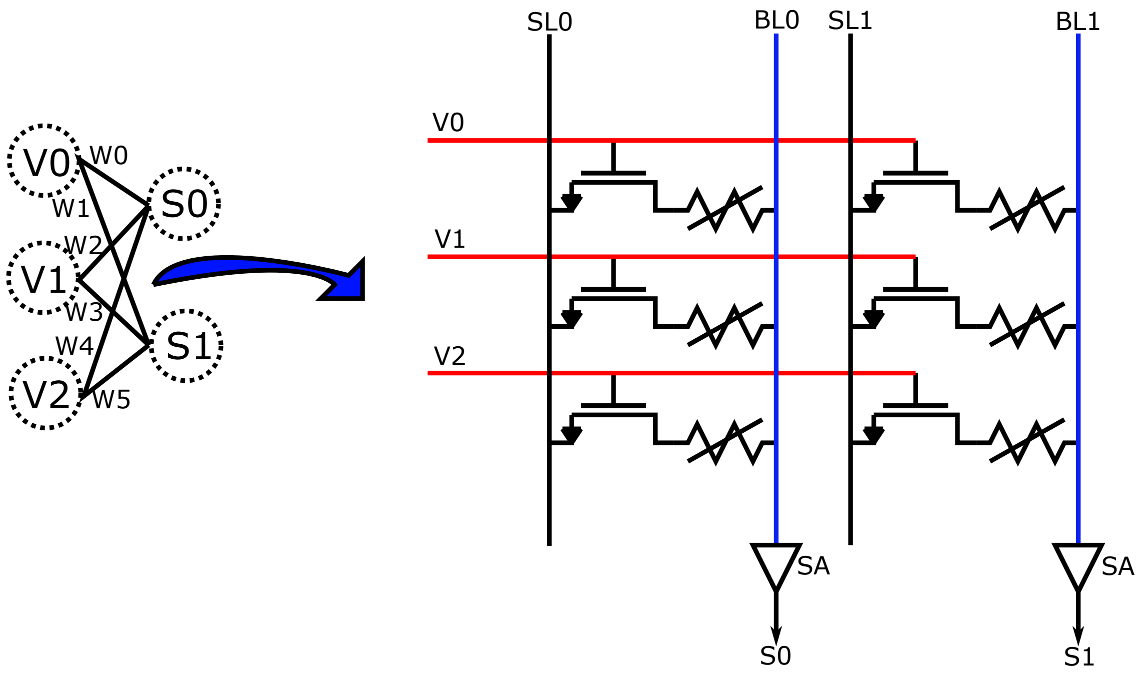


Figure 1.42: Simplified 3-2 neural network implemented with RRAMs 1T1R configuration. Source: [23]

But there are some problems that have to be solved [23]:

1. High performances and MLC cells with low power consumption are not reached yet;

2. Parasitic currents (like sneak current) are still present also with 1T1R configuration;

3. Since the computation in the array is based on an analog approach, a good interface between the array itself and CPU is needed.

1.5 SRAM based

In this section it will be discussed a solutions based on SRAM implementations.

1.5.1 In-Memory Area-Efficient Signal Streaming Processor Design for Binary Neural Networks [24]

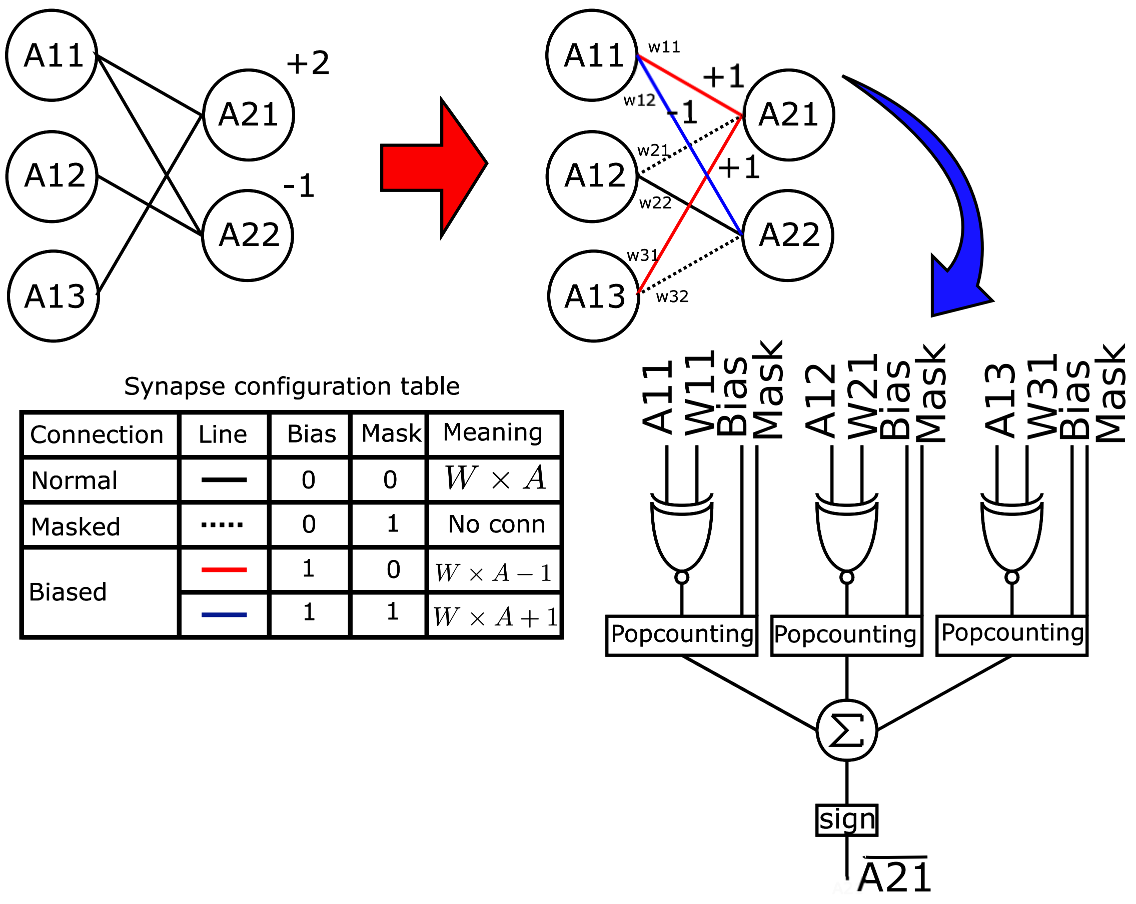
Introduction and architecture

[24] proposes an in-memory architecture which is based on BNN, so operations of XNOR and bitcounting are performed. In the implementation, it is used the concept of synapse configuration table SCT which is explained in the following parts. The NN depicted in Figure 1.43 has 3 input activations (named A_{11},A_{12},A_{13}) and 2 output activations (A_{21},A_{22}). As it is possible to see, there are also some numbers reported next to the output neurons (in this case +2 and -1): these represents the bias values that have to be added to the neuron's function to obtain the corresponding output activation. In Figure 1.43, the network is not fully connected: a general representation of these kinds of networks is needed. By considering the ternary networks, the NN in Figure 1.43 can be implemented by performing some transformations illustrated in the same image. The steps to compute a neuron's output are the following:

1. XNOR bitwise operation to compute the products between input-weight;
2. The sum in a BNN is computed by a pop-counting operation:

$$PopCount = number\_of\_1s - number\_of\_0s \tag{1.86}$$

3. Biases are added with the pop-count;
4. Activation function: the output of a neuron is the sign of the previous calculations.



Synapse configuration table

Connection	Line	Bias	Mask	Meaning
Normal	—	0	0	$W \times A$
Masked	·····	0	1	No conn
Biased	🟥	1	0	$W \times A - 1$
	🟦	1	1	$W \times A + 1$

Figure 1.43: An example of a 3-2 BNN from [24] and the transformation into a fully connected configuration. The Synapse configuration table is reported indicating the meaning of the connections. The fully connected network has been implemented considering also bias and mask signals. At the end, three popcounting results will be added together and it is taken the sign of the result, that defines the output.

An implementation of these steps has been represented in Figure 1.43. For each neuron, there is a set of (weight,bias,mask) bits that determines the meaning of the connection and the corresponding value of the weight to be multiplied with the input activation. In the example in Figure 1.43, the SCT is the following [24]:

Table 1.20: SCT of the NN depicted in Figure 1.43 from [24]

	A21			A22		
	Weight	Bias	Mask	Weight	Bias	Mask
A11	W11	1	1	W12	1	0
A12	X	0	1	W22	0	0
A13	W31	1	1	X	0	1

As it is possible to see in Table 1.20, the input activations are disposed on the rows, while the outputs on the columns. If this SCT is implemented into a SRAM, the rows correspond to the address, while the columns to the bitlines: if one row is accessed per time, it means that only one input per time can be processed. In fact this SCT configuration is called OPNE [24] (output parallel neural engine), in which the inputs are given serially, while the outputs are generated simultaneously when the scanning over all the inputs has finished. Moreover, if the network is extended into a 3-2-3 structure, the hidden layer takes the inputs (coming from the previous layer) in parallel and so a new SCT configuration has to be considered. In this case the synapse configuration table is called IPNE (input parallel neural engine) [24]:

Table 1.21: IPNE SCT from [24]

	A21			A22		
A31	Weight	Bias	Mask	Weight	Bias	Mask
A32	Weight	Bias	Mask	Weight	Bias	Mask
A33	Weight	Bias	Mask	Weight	Bias	Mask

The inputs now address multiple columns and only one output is provided per time. After an IPNE layer (which gives outputs in serial and takes inputs in parallel), an OPNE (which takes input in serial and provide outputs in parallel) can be connected without any interface circuitry [24].

General case OPNE and IPNE configurations can be also used in general with NNs different from the 3-2 example discussed before, in fact they can be extended to a general H-output/input case. In particular an OPNE takes 1bit serially and produce H outputs in parallel, while an IPNE takes H inputs in parallel and gives 1 output serially.

Additional details [40]

In this part, there are presented some additional details from [40], that implements the same architecture, but with more detailed explanations.

Batch normalization The batch normalization is an essential element in the binary/ternary neural networks in order to obtain an high accuracy with weights extremely approximated [40]. From section 1.2.4 in the introduction, the sign activation function to the formula of the batch normalization can be applied as follows:

$$\hat{Y} = sign\left(\gamma\left(\frac{Y-\mu}{\sigma}\right)+\beta\right) \text{ from [40]} \tag{1.87}$$

Where:

- Y is the weighted sum between W and activations (output of a neuron without activation function applied);
- μ,σ^2 are mean and variance of Y (over all input images);
- γ,β are scaling and offset factors

As done in section 1.2.4 in the ternary network explanation, the original Equation 1.87 can be transformed in:

$$\hat{Y} = sign\left(Y+\left(-\mu+\frac{\sigma}{\gamma}\beta\right)\right) = sign\,(Y+\text{bias}) \text{ from [40]} \tag{1.88}$$

The bias value is added after the pop-counting, requiring an additional space of memory for the bias term.

Computation cycle Considering a network with LxH size (where L is the number of rows in the SRAM and H is the number of outputs which an OPNE produces per time), from [40]:

- OPNE produces a result after L+1 cycles;

- IPNE can start computing and, in the meanwhile, OPNE can fetch another data;
- IPNE produces a result after only 1 cycle (when all the inputs are available from OPNE) and this output is used immediately from the OPNE of the next layer.

Results

In [24] is an OPNE-IPNE is considered as a PIM. Some parameters are reported:

Table 1.22: Results from [24]

#PIMS	6
H	144
L	484
Frequency [MHz]	400
Peak performance [GSOPS]	691
#neurons	3768
#synapses	836000
Power consumption [W]	0.6
Area [mm^2]	3.9
Energy efficiency [TSOPS/W]	1.2
Area Efficiency [TSOPS/mm^2]	0.177

In the table, the term SOPS indicates "synapse operation per second" which is simply a multiplication and an addition. H is the number of inputs in parallel into an IPNE, while L is the number of words into an SRAM array. Since there are 6 PIMs, the network structure is the following: 484-144-484-144-484-144-484-144-484-144-484(10). The critical path of this architecture is in the IPNE adder tree, since all the computations are performed in parallel. An additional implementation of a CNN has been analyzed by [40], in which the structure used is the following:

Table 1.23: Accuracy results of a CNN implementation from [40]

# Layers	Type	Kernel sizes	Stride	# output channels	OFMAP size	# of OFMAP
0	Input	-	-	-	22x22	1
1	Conv	5x5	4	4	6x6	4
2	Conv	5x5	4	12	6x6	12
3	F.Conn	432-144	-	-	-	-
4	F.Conn	144-10	-	-	-	-
Accuracy [%]						80
Dataset						MNIST

1.5.2 Deep learning consideration with novel approach - look-up-table based processing conjugated memory [25]

Introduction

The MLCS solution is able to implement some operations in memory, because it combines SRAM with look-up tables. A particular memory cell can be viewed as LUT logic or a simple memory element. If LUT functionality is considered, the result of the operation is simply obtained by using the inputs as addresses and so no computation are performed with this approach.

Typical structure

The typical structure used to implement an in-memory neural network is illustrated in Figure 1.44 from [25]:

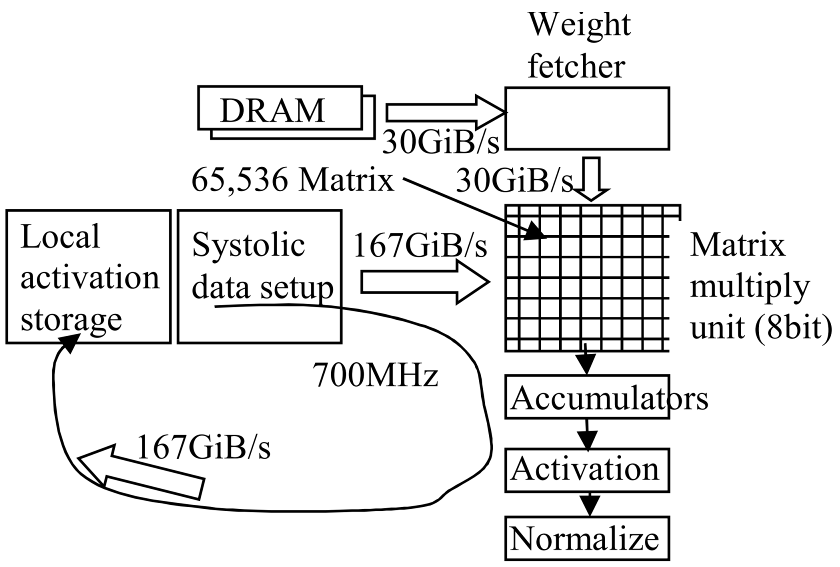


Figure 1.44: DL calculation structure at 700MHz from [25]

The image is fed to the array on the left, while the weights are precharged from the upper side, so the multiplication is then performed by the array. This operation is repeated for all the cells with an operating frequency of 700 MHz. An MLCS is a simple unit based on a SRAM (256words x 8 bits from [25]) which is used as a LUT and so it is addressed in order to give in output the result of a specific logic function, in particular it is able to perform a multiplication between two 4 bits numbers ($2^8 = 256 Words$).

Structure of MLCS for DL

Taking as example a 16x16 image in input (256 pixels), the inputs should be connected to the second layer and so there are required 256 multiply-accumulate operations. If the clock is 700MHz, since there are 256 units, the speed is reduced to 2.7MHz. By using 256 parallel architectures, the maximum frequency of 700MHz per layer can be achieved also with low power because LUT based multiplier is only addressed without any calculation.

Roughly Performance Estimation

These results have been taken from another paper ([48]) that is focused on the same approach. It is not guaranteed the correctness of these results, in fact they are reported only as reference.

Power Consumption [48] A comparison between pure logic vs FPGA vs LUT approach is reported in the following Table 1.24:

Table 1.24: Relative power results from [48]

Power	Pure logic	MLCS	FPGA
Relative ratio	1	0.05	0.1

MLCS's power is less than one twentieth of conventional pure logic's one, because the just one address access of the LUT memory is enough for the calculation.

Processing speed [48] The maximum SRAM frequency is fixed to 1GHz, due to SRAM wiring penalties. The pure logic approach with pipeline achieves 4GHz. Here is reported a comparison among different architectures:

Table 1.25: Speed comparison from [48]

Band frequency	Pure logic (8/64bits)	8bits		64bits	
		MLCS	FPGA	MLCS	FPGA
Speed	4GHz	1GHz	500MHz	1GHz	250MHz

Area Comparison [48] This comparison is done on a 8 bit multiplier. LUT based SRAM circuit needs 4096cells of SRAM ($0.5um^2$/memory cell [48]). In total $2.05kum^2$ (TSMC 65nm). SRAM memory cell often is 1/3 smaller than logic gate but 4× memories are needed for making LUT plus overheads coming from registers, I/Os etc [48]. As a consequence, the area of the SRAM-LUT is 7 times larger than pure logic.

Table 1.26: Area comparison from [48]

Area	Pure logic	MLCS	FPGA
Ratio	1	7	20

1.5.3 A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm [26]

Introduction

[26] proposes an implementation of a neurosynaptic core based on a SRAM crossbar array. The architecture is event-based that corresponds to the brain's way of computation. The real neuron's model is implemented in [26], in which synapses (connections between neurons), axons and neurons' core are integrated in-memory, in particular the chip has **256 digital neurons**, 1024 rows (axons) and so the array dimensions is 1024x256.

Table 1.27: Parameters from [26]

Network structure	Area [mm^2]	Technology	Energy per spike [pJ]
1024x256	4.2	45nm SOI	45

Architecture specification [26]

In Figure 1.45, the neurosynaptic core is composed by K axons (rows), KxM synapses and M neurons. The blue circles indicate the intersection between axons and columns, which represents the weight. At the end of a column there is a neuron indicated by the red box. In each time instant t, there is an activity bit $A_j(t)$ which indicates if a particular neuron has been fired or not in the previous time t-1. Connected to each each axon, there is a G_j value, which indicates what is the type of connection (0,1,2) (inhibitory,excitatory [26]). The synapse value of a neuron i is indicated as $S_i^{G_j}$ from [26], so a neuron's input is defined as in [26]:

$$A_j(t) \times W_{ji} \times S_i^{G_j} \tag{1.89}$$

The membrane potential of the neuron is considered from [26]:

- $V(t)$: membrane potential;
- L: leak;
- θ: threshold;

$$V_i(t+1) = V_i(t) + L_i + \sum_{j=1}^{\mathrm{K}} \left[A_j(t) \times W_{ji} \times S_i^{G_j} \right] \text{ from } [26] \tag{1.90}$$

When V(t) is higher than θ, the neuron produces a spike and its membrane potential is reset to 0.

Implementation [26]

In the following figure it is reported the architecture of the neurosynaptic core:

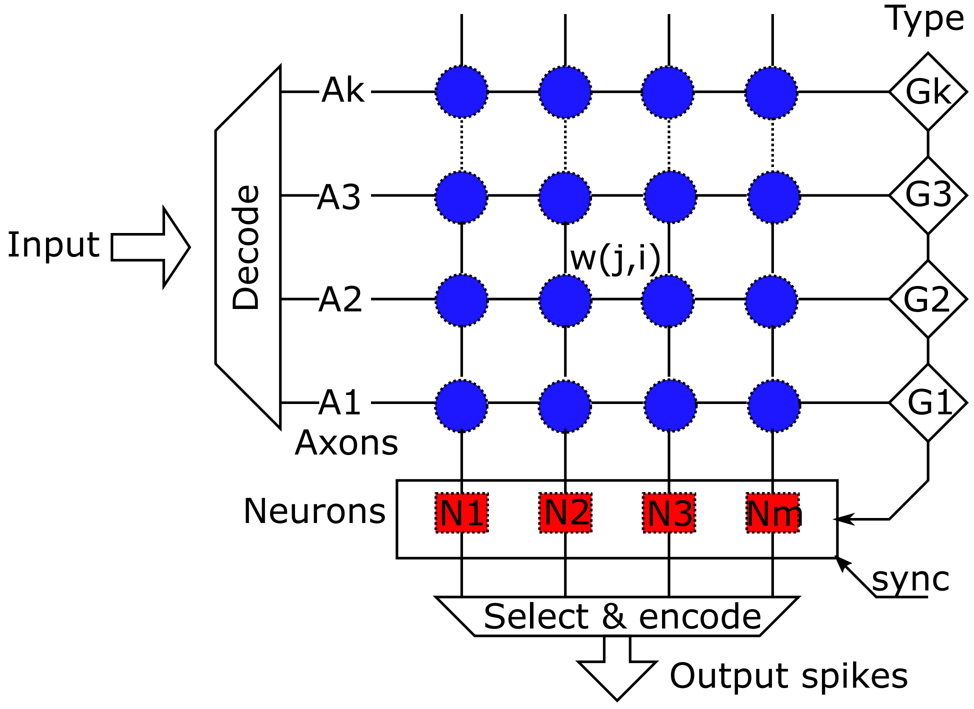


Figure 1.45: Structure of the neurosynaptic core from [26]

The communication between each block of the architecture is event-driven based, and so without any clock. In order to correctly synchronize all the operations, handshake signals have been implemented. All the neurons are implemented as stand-alone elements: no multiplexed structure has been used in [26] to realize all the computations in parallel. The steps that the architecture executes in the processing flow are the following:

1. The addresses are fed to the crossbar one at a time. The corresponding row is activated and the connections (weights) and the type of connection (G_j) are read;
2. All the connections of type 1, are sent to the neuron that perform the membrane update in Equation 1.90;
3. Once all the neurons are updated, the address read procedure has finished;
4. Everytime 1ms has passed (after the completion of the steps described so far), a Sync signal is sent to the neurons, which controls if the membrane potential is higher than θ or not. If so, the membrane potential is reset to 0 and a spike is produced (logic "1" coming from the corresponding neuron).

Results

The results and some useful parameters from [26] are reported:

Table 1.28: Network parameters for 1024x256 crossbar array dimensions from [26]

Parameters	
Network structure	1024x256
# Transistors	3.8 million
# Neurons	256
Neuron's area [μm^2]	3325
Bitcell area [μm^2]	1.3
Delay [ms/img]	1
Vdd [V]	0.85
Energy per spike [pJ/spike]	45
Worst case energy [pJ]	11520

The accuracy results from [26], considering a network structure of 484x256, are the following:

Table 1.29: Accuracy results from [26]

Accuracy test		Details
Dataset	<u>MNIST</u>	-
# neurosynaptic cores	2	(excitatory and inhibitory)
Network structure	484x256	-
# Training images	50000	-
# Test images	10000	-
Accuracy [%]	89	Neurosynaptic core
Accuracy [%]	94	Real value weights

The network is realized with 2 neurosynaptic cores of 484x256 which are configured with excitatory and inhibitory G_j bits.

1.6 DRAM Based

1.6.1 XNOR-POP: A processing-in-memory architecture for binary Convolutional Neural Networks in Wide-IO2 DRAMs [27]

Introduction

[27] proposes a novel architecture based on DRAM, which is able to implement a XNOR-NET: XNOR operations are performed inside the memory and are transferred to the logic layer by TSVs, in which the popolation-counting computing is performed. TSVs enable power saving, reduction of wires' length and consequently the delay.

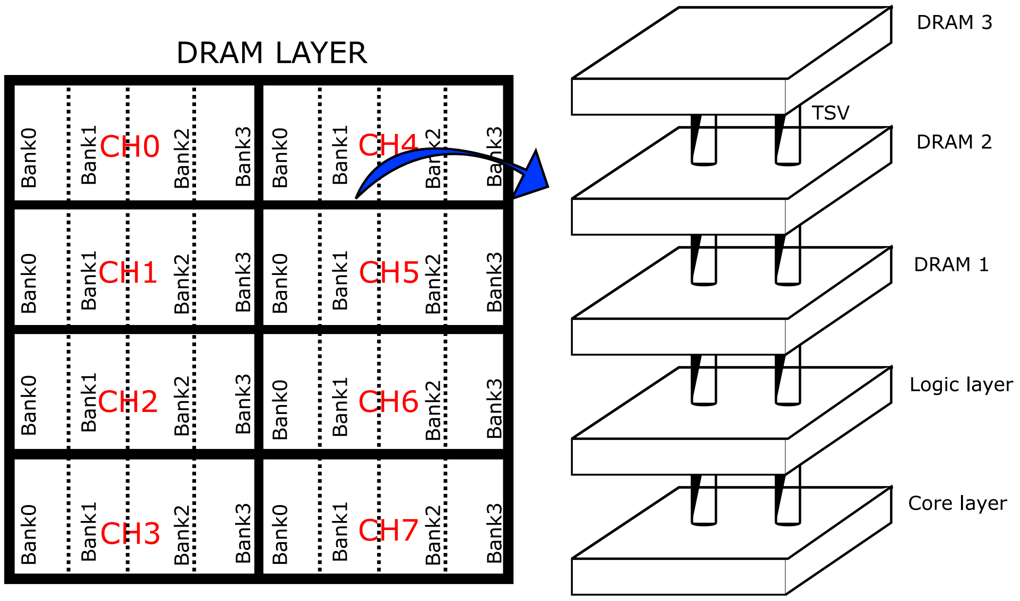


Figure 1.46: Architecture proposed by [27]. Source: [27]

The architecture is depicted in Figure 1.46: each DRAM layer (8Gb) has 8 channels with 64 bits and each channel has 4 banks[27].

XNOR-NET CNN The XNOR-NET CNN has the following building blocks:

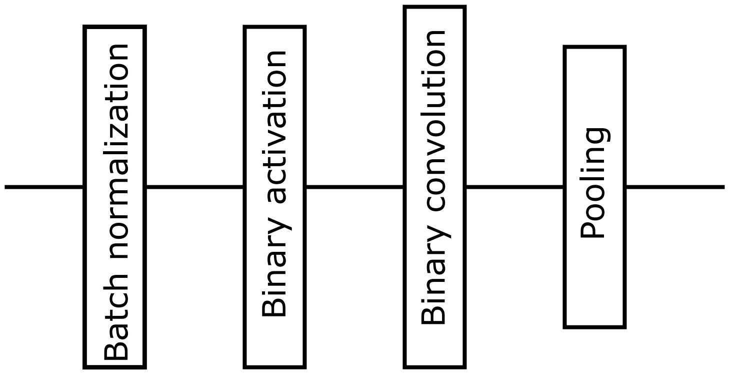


Figure 1.47: Building blocks of a XNOR-NET from [27]

The convolution in output is obtained as:

$$Y_{conv} = (I \circledast W) \cdot \mathbf{K}\alpha \tag{1.91}$$

As already mentioned, the batch normalization applied to a XNOR-NET can be reduced simply into the following equation:

$$y_{(\text{batch})} = \begin{cases} 1, & \text{if } x \geq \mu - \dfrac{\beta}{\gamma\sqrt{\sigma^2 + \varepsilon}} \\ 0, & \text{otherwise} \end{cases} \tag{1.92}$$

So a simple comparator can be used.

Binary Convolution: XNOR-Popcount

XNOR-Dram The structure of a bank is reported in the following figure from [27]:

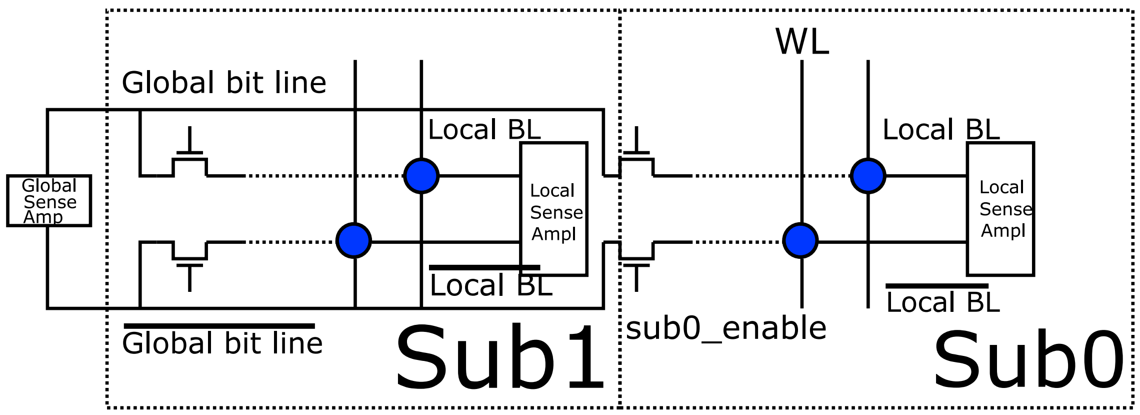


Figure 1.48: Bank structure. Source:[27]

The functional steps are the following from [27]:

1. At the beginning, all lines are precharged to 1/2 Vdd;
2. WL is activated: the local sense amplifier senses the difference between Local bit line, $\overline{\text{Local bit line}}$;
3. Cell content is restored by Local sense amplifier. The local bit lines are attached to global bit lines through switches.

A XNOR operation is performed considering an additional block inserted after the global sense amplifier. The operational steps to compute $\overline{A \oplus B}$ are the following from [27]:

1. A and $\overline{A}$ are fetched from the subarray 0 and memorized in Global sense amplifier;
2. Global sense amplifier/Sub0 connection is detached;
3. Local sense amplifier charges subarray 1;
4. B is read from subarray 1 and sent to the XNOR engine;
5. The connection between XNOR engine/global sense amplifier is attached and a result is produced and memorized;
6. XNOR/Subarray 1 are disconnected from global bit lines;

7. Local sense amplifier precharges subarray 1 again;
8. Global sense amplifier precharges the global bitlines.

The banks are organized in such a way that the input is stored in subarray0, while in subarray1 the corresponding weight. From [27] the total latency of this operation is 128ns that can be reduced to 78ns, when loop unrolling technique is used [27]. The results elaborated in the DRAM are sent to the logic die by means of TSVs to perform the popcount adopted in [49]: two of them are required to count 1s and 0s respectively. For the pooling technique, a 16 bit comparator is used and in the pooling phase also the matrix $\mathbf{K}$ and α are computed.

Results

At the beginning, the architecture has to fetch the weights and to dispose them in the banks in order to perform all the operation explained so far. If the network is very deep, weights could occupy a lot of memory. In the following table, there are reported the results from [27]. There are also presented comparisons between the floating point network accuracy and its corresponding XNOR-Net implementation:

Table 1.30: Accuracy and performance results of the architecture with different neural network models. Source:[27]

DRAM area [mm^2]		Logic die area [mm^2]		Power logic die [mW]		Power DRAM layers [W]
77.24		2.24		237		1.99
Network used	Dataset	Structure		Accuracy (FP) [%]	Accuracy XNOR [%]	Frame per second
LeNet-5	MNIST	Layer	Type	99.1	97.2	-
		1 to 3	Conv			
		4 to 5	Pooling			
		6	Fully connected			
MLP	MNIST	1 to 5	Fully connected	98.5	96.9	-
CNP	MNIST	1 to 3	Conv	97	96.1	-
		4 to 5	Pooling			
		6	Fully connected			
SCNN	MNIST	1 to 2	Conv	99	97.8	-
		3 to 4	Fully connected			
MCDNN	MNIST	1 to 3	Conv	96.8	95.7	-
		4 to 6	Pooling			
		7 to 9	Fully connected			
AlexNet	ImageNet	1 to 5	Conv	80.2	69.2	3390
		6 to 8	Pooling			
		9 to 10	Fully connected			
ResNet-18	ImageNet	1 to 18	Conv	89.2	73.2	1391
		19 to 20	Pooling			
		21	Fully connected			

1.7 OOM implementations

Particular mixed implementations or computation methods are presented in this section, that can be employed to realize a neural network. The NNs implemented in the following part are not realized in-memory but as hardware accelerators, in fact the term OOM means **out of memory**.

1.7.1 Energy-Efficient Hybrid Stochastic-Binary Neural Networks for Near-Sensor Computing [28]

Introduction

[28] proposes a solution in which raw data (such as data coming from sensors) have to be processed. One of the possible ways to operate on such data is the NN employment combined with near-data computing. [28] introduces a new way of computation based on a stochastic-binary approach (SC), where a bit sequence represents a probability. Its implementations is cheaper than the classical binary approach, but it requires longer computation time and consequently higher energy [28]. The precision in this case can be reduced in order to save energy/time. The stochastic approach is used only in the first layer of the neural network.

Architecture and considerations

The SC is based on the probabilities, and so a bitstream in SC has the following meaning:

$$X = 01101010 \rightarrow \text{Probability} = \frac{\#1s}{length} = \frac{4}{8} = 1/2$$

The probability in this case is, 0.5 because there are four 1s out of 8 possibilities. The arithmetic functions are easily implemented: multiplication is simply realized with an AND logic gate.

$$\begin{aligned}
p_1 &= 0.5 \rightarrow X_1 = 0110 \\
p_2 &= 0.25 \rightarrow X_2 = 0100 \\
p_1 \cdot p_2 &= 0.5 \cdot 0.25 = 0.125 \\
Y &= X_1 \text{AND} X_2 = 0100 \rightarrow p_Y = 0.125
\end{aligned}$$

In the following figure are reported the stochastic circuits used in [28]:

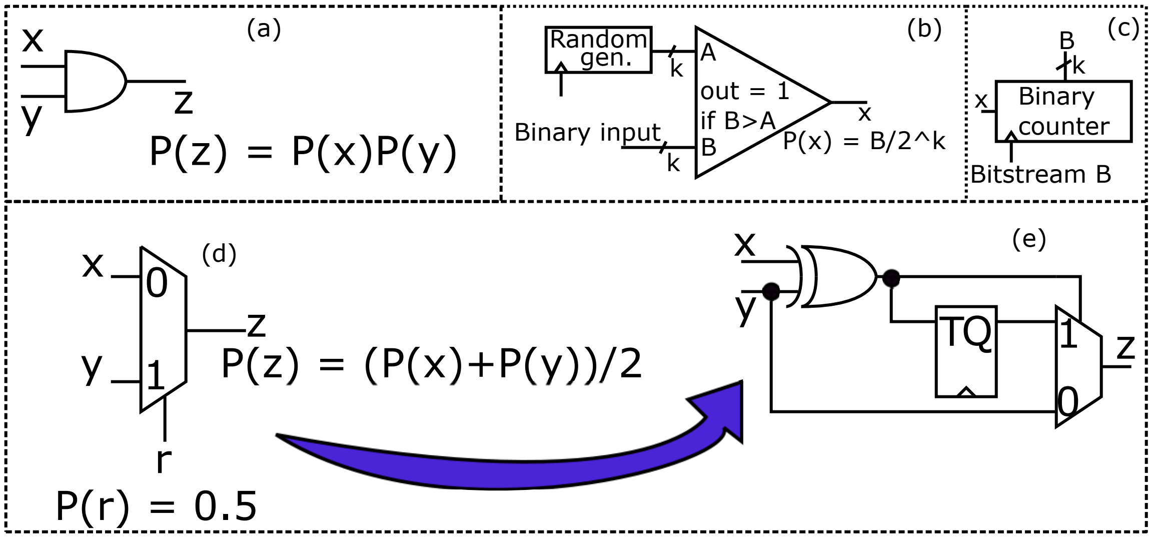


Figure 1.49: (a) Multiplier; (b) Binary - Stochastic converter; (c) Stochastic - Binary converter; (d) Multiplexer adder with random input r; (e) Improved version of the adder, without the random input. Source: [28]

This kind of computation presents some errors, depending on the positions of the incoming bits. One way to improve the precision is to enlarge the bit sequence, in fact the precision of the SC is given by:

$$\text{Precision} = log_2(\text{length}) \tag{1.93}$$

The probability is only in the range [0,1], but this problem can be easily solved by considering the value of X as $2p_X - 1$ [28]. The term length in the formula is the bitstream size. An adder is implemented from a stochastical point of view as a multiplexer, in which, as a selector it is used a random value with probability P(r)

= 0.5 (Figure 1.49 (d)). This implementation has been improved in [28], in such a way to eliminate the additional random input: considering the circuit in Figure 1.49 (e), at each clock cycle, if X and Y are the same, Y is propagated to the output[28]; otherwise, the state of the TFF is changed. In order to understand its functionality, consider the following example from [28]:

1. Initial TFF state = 0;
2. X = 0100 1010 (3/8);
3. Y = 0010 0010 (1/4).

By performing all the computations (showed in Figure 1.50), the output bitstream results to be equal to 00100010 (1/4). In fact:

$$Z_0 = 0.5 \cdot (3/8 + 1/4) = 5/16 \sim 1/4 \tag{1.94}$$

In case of initial TFF state equal to 1, the result will be 01001010. Considering the other circuits depicted in Figure 1.49, the binary to stochastic converter is designed as a comparator with its input connected to a random number generator and to the input binary: if this last one is higher than the number randomly generated, the output will be 1, otherwise 0 (Figure 1.49 (b))[28]. Similarly, the conversion from stochastic to binary can be performed by a binary counter which counts the total number of 1s into the bitsequence (Figure 1.49 (c))[28].

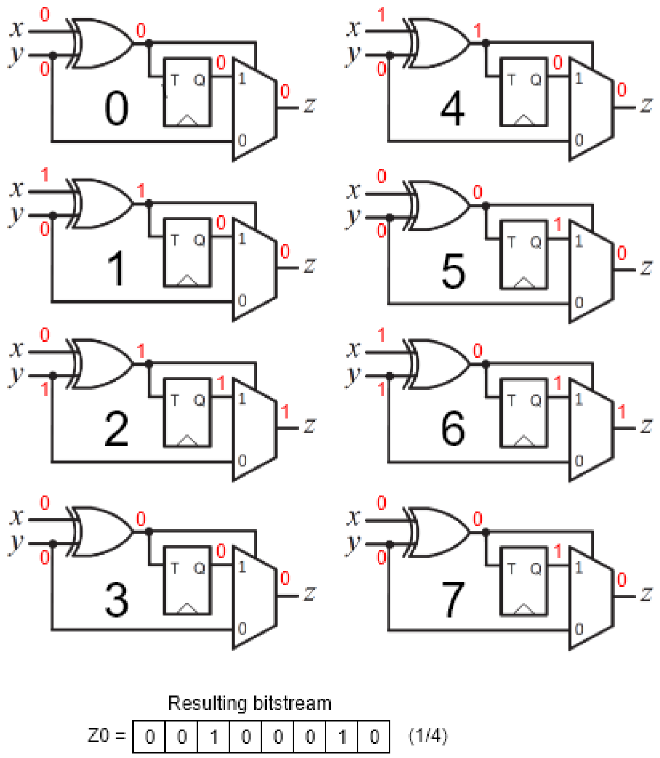


Figure 1.50: Example of computations of the new stochastic adder. Source: [28]

Stochastic binary neural network design

In order to implement the stochastic approach, [28] considers the LeNet-5 neural network which is composed by:

Table 1.31: LeNet-5 structure Source: [50]

Layer	Type	# Channels input	IFMAP size	Kernel size	OFMAP size	# Channels output	Details
0	Input	-	-	-	28x28	1	-
1	Conv	1	28x28	5x5	28x28	32	-
2	Max Pool	32	28x28	2x2	14x14	32	-
3	Conv	32	14x14	5x5	14x14	32	-
4	Max Pool	32	14x14	2x2	7x7	32	-
5	FC	-	-	128	-	-	50 % dropout
6	FC	-	-	10	-	-	25% dropout & softmax

The NN is made by bipolar operations, but the bipolar approach of SC is not used because the accuracy is degraded. [28] adopts unipolar operations by splitting the weights into positive/negative bit-streams w_{pos} and w_{neg} and so two different dot products are computed (negative/positive), converted in binary domain and the sign function is performed by a simple comparator.

Results

The results from [28] are now reported. The architecture has been tested on MNIST dataset as reported in the Table 1.32, with different bitstream lengths, in order to see what are the changes in the evaluated parameters. The power is normalized to the throughput, because depending on the application, the throughput can be chosen arbitrarily.

Table 1.32: Performance and accuracy results. Comparison with the classical binary approach and the discussed one. Source: [28]

	Hardware	Dataset	# Images		#Training images		#Test images	
	TSMC 65 nm	MNIST	70000		60000		10000	
	Bitlength	8	7	6	5	4	3	2
Stochastic	Accuracy [%]	99.06	99.01	98.96	98.88	98.96	97.8	56.18
	Power/Throughput [mW]	33.17	33.55	33.26	33.01	33.2	29.96	28.35
	Energy efficiency [nJ/frame]	543.42	274.82	136.22	67.6	34	15.34	7.26
	Area [mm^2]	1.321	1.282	1.24	1.2	1.166	1.11	1.057
Binary	Accuracy [%]	99.11	99.14	99.11	99.26	99.21	99.21	98.7
	Power/Throughput [mW]	40.95	72.8	121.52	204.96	325.36	501.76	683.2
	Energy efficiency [nJ/frame]	670.92	596.38	497.74	419.76	333.17	256.9	174.9
	Area [mm^2]	1.313	1.094	0.891	0.71	0.543	0.391	0.255

1.7.2 Towards Near Data Processing of Convolutional Neural Networks [29]

Introduction

[29] proposes an approach in which the memory wall problem is reduced by introducing the near-data processing (NDP). However, incorporating memory with logic is very expensive in terms of performances, but the solution of a 3D stacked structure connected via TSV is studied and in particular it is applied to CNN architecture. HMC (Hybrid memory cube) divided into vaults has been chosen by [29].

HMC Structure The technology of the HMC is made by DRAM layers (4 to 8), in which the image is split. They are stacked on top of each other and connected by TSV as already mentioned. At the bottom layer there is a computational unit which performs all the computations that a CNN requires[29]. Each DRAM layer is divided in 16 parts and a stack of these parts coming from different layers is called vault [29], which is divided into two parts called banks. The architecture of the system is reported in the Figure 1.51 from [29]. HMC has 4 layers of 4Gb each (total 2GB). In each vault controller there is a CLU (CNN logic unit) which computes the convolution operation for a specific vault. In particular [29] adopts the floating point double precision representation of the numbers (so it is not a binary network). The CLU contains a floating point multiplier, adder, some registers (that stores the bias value and partial results) and an SRAM (which contains the filter weights, that are the same for all the CLUs): all the needed elements to compute a convolution.

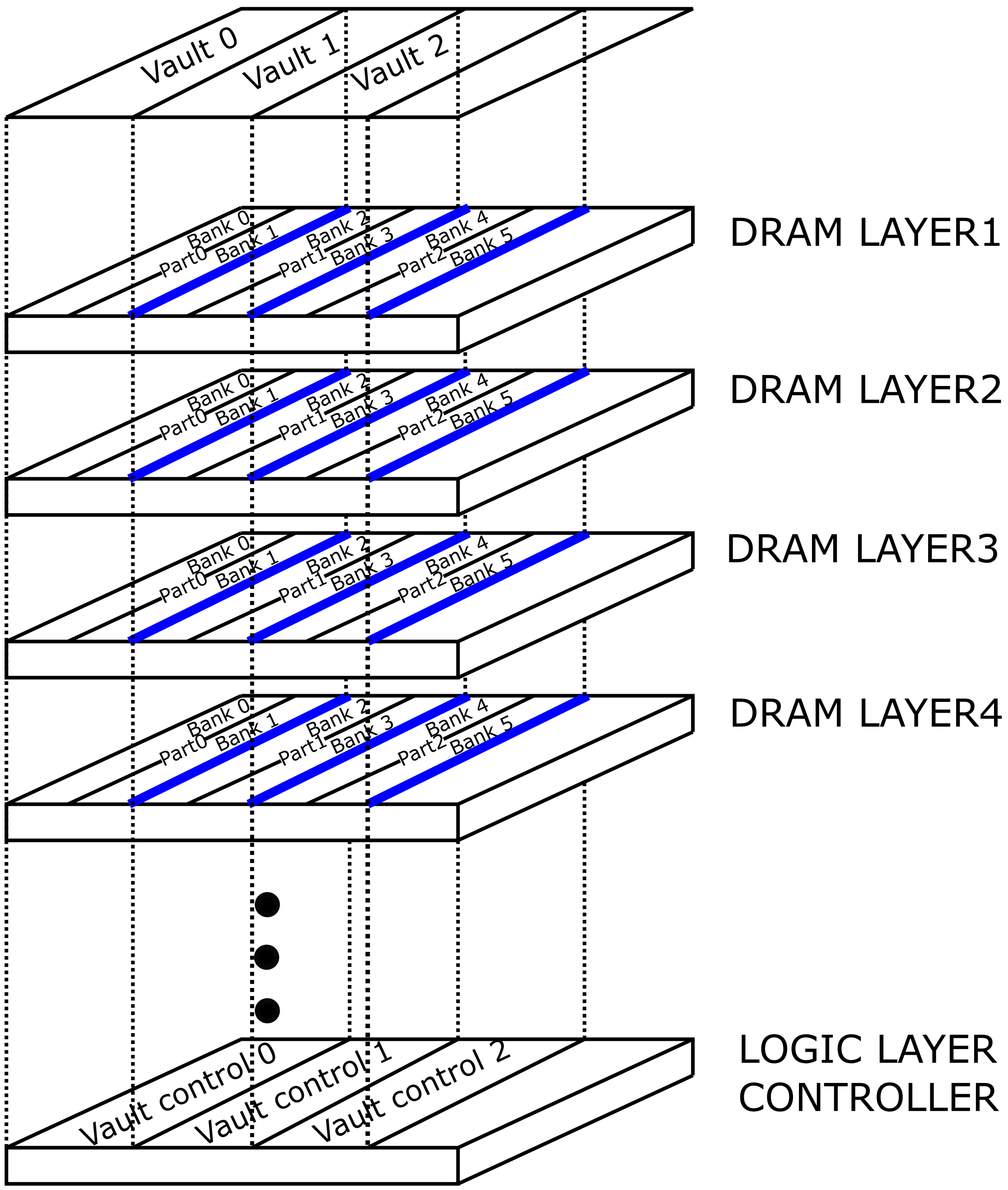


Figure 1.51: Architecture structure. Source: [29]

Computational steps When the host processor assert a start signal, the computation begins and it is performed in the following way:

1. The kernel's elements are loaded into the CLU SRAM and an image element

is loaded inside the CLU from DRAM banks;

2. The floating point multiplier in the CLU performs the multiplication between the weights stored in the SRAM and the incoming image element. Eventually a bias element is added by means of the floating point adder;

3. Results are sent back to the memory die. Some of them could be partial results, because the memory is split and so some elements of the image could be located into a different vault as shown in Figure 1.52.

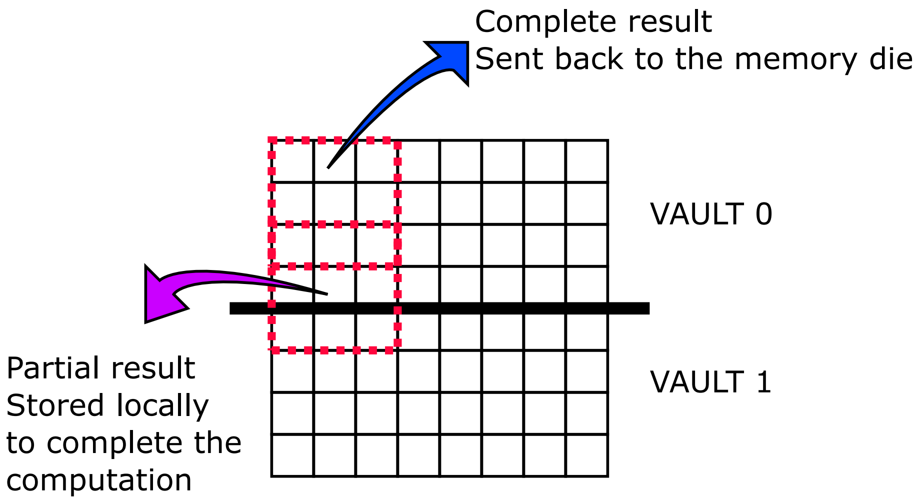


Figure 1.52: Complete and partial result computation. Source: [29]

The partial result is stored locally in order to be considered at the end of computation of the following vaults.

4. Partial results are then added together by means of inter-vault connections and then are written in the memory dies.

Results

Here there are reported the results of this implementation from [29]. Also the network structure employed by [29] is specified.

Table 1.33: Results and network structure (Source: [29]) of the floating point architecture

CLU								HMC	
CLU Frequency [MHz]	Technology			Delay [s]		Energy [J]		Area [mm^2]	Power [W]
500	UMC 90nm								
CNN STRUCTURE									
Layer	Type	IFMAP size	kernel size	CPU-Based	CLU	CPU-Based	CLU		
1	Conv	128x128x3	5x5x3	1.149	0.0138	11.0619	0.1848	729	11
2	Conv	124x124x3	5x5x3	1.0808	0.0133	10.4054	0.1779		
3	Conv	120x120x3	4x4x3	0.6643	0.0086	6.396	0.1148		
4	Conv	117x117x4	4x4x4	0.832	0.0165	8.0103	0.2057		
5	Conv	114x114x5	3x3x5	0.808	0.0091	7.7793	0.118		
6	Conv	112x112x3	5x5x3	0.873	0.0105	8.4049	0.1415		
7	Conv	108x108x3	5x5x3	0.809	0.0102	7.7888	0.1354		

1.7.3 Chain-NN: An energy-efficient 1D chain architecture for accelerating deep convolutional neural networks [30]

Introduction

[30] proposes an energy efficient and reconfigurable architecture which is based on a chain of processing elements interconnected. There can be different architectures that implement a CNN:

- **Memory-centric[30]**: there are not data reuses in the processor, and so the data are fetched from the memory. PEs in the CPU are simply stacked and are not interconnected to each other.

(Pro) Reconfigurability;

(Con) Low efficiency.

- **2D Spacial[30]**: Data are reused in the processor, since a connection between one PE and the following one exists. This solution reduces the data fetching from the memory because the PE maintains locally data frequently used, and passes them to the following PE if needed.

(Pro) Reduced data movements;

(Con) High power-area cost.

- **1D-Chain[30]**: PEs are arranged as a chain and piloted by an FSM. The sequential circuit is able to precharge the kernel parameters and, after that, the IFMAPs are streamed along the chain architecture in order to compute the CNN results.

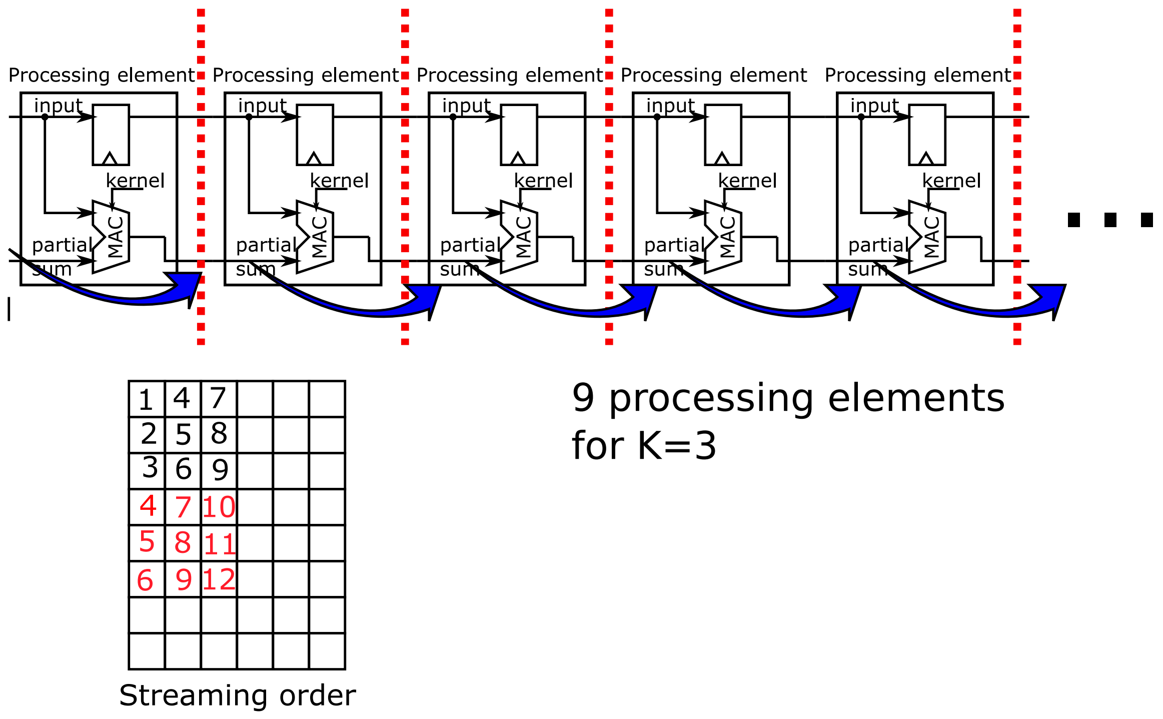
(Pro) Better energy efficiency;

(Pro) Data reusability;

(Pro) High reconfigurability and so high performance.

Chain-NN: 1D Chain Architecture

1D Chain architecture An example of chain NN is reported in the Figure 1.53, considering a kernel size of 3. Each chain is mapped to a convolution kernel window: the inputs are sent serially to the chain and each PE performs a MAC operation with kernel weight. This architecture works well, but in the case in which some pixels are not included in a convolutional window, there are required additional clock cycles to fetch the new pixels, resulting in a throughput decreasing (in particular with K = 3 and stride = 2, the maximum number of matching pixels are 6 in two different convolutional windows, so at least 3 pixels have to be fetched). For this motivation, dual channel architecture has been designed in [30].



$$k_1 \cdot (1) + k_2 \cdot (2) + k_3 \cdot (3) + k_4 \cdot (4) +$$
$$k_5 \cdot (5) + k_6 \cdot (6) + k_7 \cdot (7) + k_8 \cdot (8) + k_9 \cdot (9)$$

Figure 1.53: Chain NN architecture with k = 3, where k is the kernel size. 9 processing elements are needed because for each PE, a different weight is used. Inside a PE there are a MAC and a register and eventually the corresponding outputs can be pipelined, in order to improve performance (red dashed lines). Example of computation. Source: [30]

Dual channel [30] proposes a solution to this problem by increasing the total number of fetched data in a single PE. This implementation is called dual channel architecture, in which the column wise scanning is maintained, but this time at least 2K-1 row elements are fed to the PE. The PE fetches the even columns (evenIF) and, after K+1 clock cycles, the odd columns (oddIF) with the following order:

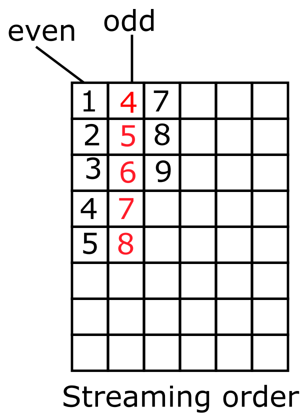


Figure 1.54: Streaming order in dual channel architecture. Source: [30]

In this way the Dual-channel architecture can continuously perform new convolutional operations without waiting times. Inside each PE, there is an internal storage (kMemory) that keeps the kernels (which are the same, since it is a CNN).

Results

From [30] are written the results. The implemented network is AlexNet (only with the convolutional layers and without the fully-connected part) and its structure is reported in the following table.

Table 1.34: Results and network structure. Source: [30]. The implementation is in fixed-point precision. For an OPS (operation per second) is a multiplication and an accumulation

Floating point precision									
Technology	# PE		Critical path delay [ns]		Max frequency [MHz]		Batch size		
TSMC 28nm	576		1.428		700		128		
CNN STRUCTURE (AlexNet)				Results					
Layer	Type	IFMAP size	kernel size	Time required [ms]	Memory required [MB]	Total power [mW]	Throughput [GOPS]	Power efficiency [GSOPS/W]
1	Conv	227x227x3	3x3	29.2	44.9	567.5	806.4	1421
2	Conv	55x55x96	3x3	43.83	175.3			
3	Conv	27x27x256	3x3	58.43	312.1			
4	Conv	13x13x384	3x3	102.53	234.3			
5	Conv	13x13x256	3x3	159.35	156.2			

1.7.4 An Energy-Efficient Architecture for Binary Weight Convolutional Neural Networks [31]

Introduction

[31] proposes a BCNN architecture, which can be used in an embedded system, since it is a low power implementation. Very deep CNN architecture can be realized with this architecture and it is compatible with BinaryConnect or BWN. [31] analyzes only the convolutional layers.

Background

Binary weight CNN [31] analyze BinaryConnect and BWN. The second one differs from the first one only by the scaling factor α, given by [31]:

$$\alpha_o^{(\ell)} = \frac{\|\mathbf{W}_{o,\mathrm{fp}}\|_{\ell 1}}{n} \tag{1.95}$$

So the correspondent output of the BWN is given by [31]:

$$y_{o,\mathrm{bwn}}^{(\ell)}(j,i) = \alpha_o^{(\ell)} \times y_{o,\mathrm{bc}}^{(\ell)}(j,i) \tag{1.96}$$

Where $_{(bc)}$ means BinaryConnect and $_{(fp)}$ floating point. As already mentioned, the α coefficient requires the fully precision weights in its computation. A basic stage of a BCNN is reported in the following table:

Table 1.35: Basic stages of a binary convolutional neural network. Source: [31]

Layer	Type	Operation
1	BCNN	Binary convolution
2	Scaling	y = αx
3	Batch normalization	Apply batch normalization formula
4	ReLU	Max(0,x)
5	Max Pooling	Downsampling

Algorithmic optimizations for BCNNs

The following optimizations are implemented in [31]:

1. [31] proposes the 1's complement to further reduce the complexity of the system, since the 2's complement requires an additional sum. This approximation introduce an error of 15% on CIFAR-10 with VGG-16 architecture. The error can be mathematically defined from [31] as:

$$x^* = x - n \tag{1.97}$$

   Since ± 1 are roughly equal, some considerations in advance can be made and this error can be compensated by knowing the number of -1s (n)

2. Since the max pooling layer selects only the maximum out of all the possible outputs, the others computed are useless. An earlier pooling can be made from [31]. This technique is based simply on the changing the order of the layers, in fact pooling layer is performed after convolution as shown in the following table:

Table 1.36: Basic stages with earlier pooling Source: [31]. Compared to Table 1.35, the pooling layer is placed as second in the order.

Layer	Type	Operation
1	BCNN	Binary convolution
2	Max Pooling	Downsampling
3	Scaling	y = αx
4	Batch normalization	Apply batch normalization formula
5	ReLU	Max(0,x)

3. Batch normalization transformation: since the batch normalization is defined as

$$y_{batch} = \frac{y_{conv} \times \alpha - \mu}{\sigma} \tag{1.98}$$

[31] considers the terms $m = \dfrac{\mu}{\alpha}$ and $n = \alpha/\sigma$ and transforms the equation in:

$$y_{batch} = (y_{conv} - m) \times n \tag{1.99}$$

So no divisions are performed.

4. Quantization of the activations: the ReLU layer has to be quantized somehow. [31] proposes a method based on a equal-discance nonuniform quantization which produces a maximum accuracy loss of 1.7%.

Architecture

Top-level architecture In Figure 1.55 it is reported the architecture proposed by [31]:

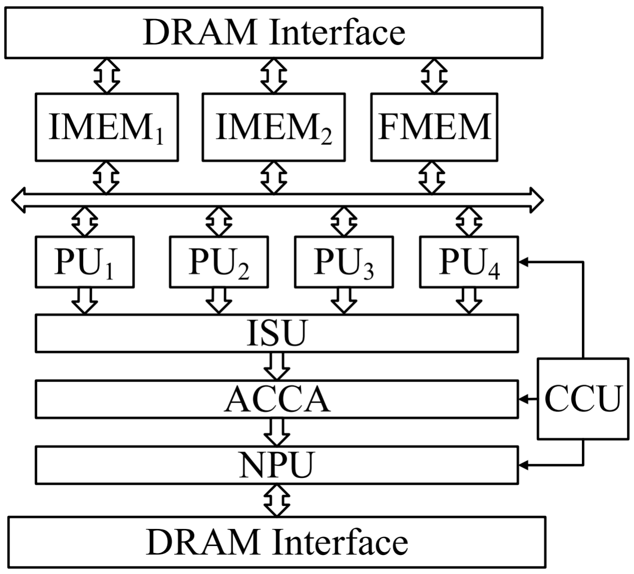


Figure 1.55: Architecture. Source: [31]

The architecture in Figure 1.55 [31] is composed by an image memory that stores two rows of the IFMAP (size $C_{\text{in}} \times 2 \times w_{\text{in}}$); a filter memory (FMEM) that contains the filter elements; some PUs that compute convolution operations and each of them process an OFMAP; an input feature map summation unit (ISU) that adds all the OFMAPs and produce the neuron's output; an accumulation array (ACCA) which accumulates the exceeding IFMAPs, if their number is higher than the PUs available; a Neuron PU (NPU) that computes scaling, batch normalization, ReLU, max-pooling and produces 256 output neurons per clock cycle[31]; a central control unit that schedules the architecture [31].

Processing unit The PU is able to compute the convolution since it is composed by multiple filters (MFIR) and their outputs are added together in multiple fast adder units (FAUs), which are made by optimized compressor tree structure based on 4:2 and 3:2 compressor circuits.

Adder tree All the multiplications have been removed from by the binary convolutional layers and so the critical path will be in the accumulation part. In one convolution, an output neuron is obtained by adding together $w_{kernel} \times h_{kernel} \times$ window_size = 36 data.

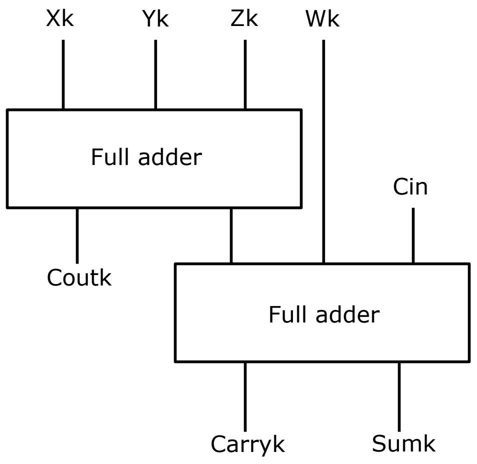


Figure 1.56: 4:2 compressors used in [31]

The compressor tree is made by 3:2 and 4:2 compressors, and in particular, the last ones do not have a carry chain, so the delay is not heavily influenced. The signals Coutk and Cink are not used, producing an approximated result.

Approximate Binary Multiplier Since the design uses 1's complement representation, an approximated version can be used, in which the adder which adds 1 to obtain the 2's complement is not implemented.

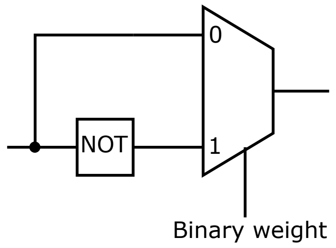


Figure 1.57: Approximate multiplier. Source: [31]

This implementation brings a 60% area reduction.

Approximate Adder [31] The adder tree inaccuracies have been alleviated by dividing the adder into two parts (N:k) part and (k-1:0) respectively. The input carry bit of the N:k part is taken from the k-th bit of input data in the (k-1:0) part

1. For the higher (N-k) -bits subadder, its input carry bit Cin is approximately speculated using the k th bit of one of the input data, reducing datapath delay and hardware complexity;
2. When the k is set to half of the word size, the hardware efficiency gain can reach the maximum, but the error rate increases with k. The error obtained with this approach is $\pm 2^k$, where k is the split position.

NPU, batch normalization unit [31] As already said, the NPU is able to compute the batch normalization, activation (ReLU) and max pooling, in particular 8 convolved neurons passes through it [31]. In order to save power, a "disable" signal is used to turn off the NPU when it is not needed. Since the ReLU function is 0 for all the elements below 0 (max(0,x)), a latch (piloted by the sign of the incoming bits) is introduced before the quantization unit in order to block the data propagation in the case of negative result coming from the adder.

Implementation results and comparison

Results from [31] are reported in Table 1.38, including the structure of the VGG-16 CNN model used. The technology used in the 130nm, while it is possible to demostrate that from 90nm the frequency reaches a maximum value of 650MHz. In Table 1.37 are reported the bit-lengths and memory sizes of the different building blocks in the architecture:

Table 1.37: Bit-lengths and memory used. Source: [31]

Bit-Lengths			
IMEM	ISU Adders(k=3)	ACCA Adders (k=4)	NPU
6	13	15	16
Memory sizes [KB]			
IMEM	FMEM	ACCA	NPU
21.6	295	53.76	2

Table 1.38: Results Source: [31] with corresponding CNN structure. An OPS is a MAC operation per second.

Technology							Dataset				
130nm SMIC							ImageNet				
CNN STRUCTURE							Total Power [mW]	Frequency [MHz]	Peak performance [TOPS]	Area [mm^2]	Core voltage [V]
Layer	Type	IFMAP size	Kernel size	# operations [MOPs]	Time required [ms]	Throughput [GOP-S/s]	768.7	190	3.5	44.957	1.08
1	Conv	224x224x3	3x3x64	183.04	0.34	549.67					
2	Conv + Pooling	224x224x64	3x3x128	3709.01	4.17	882.33					
3	Conv	112x112x128	3x3x128	1854.5	1.94	941.53					
4	Conv + Pooling	112x112x128	3x3x128	3704.19	3.86	949.03					
5	Conv	56x56x128	3x3x256	1852.1	1.94	937.7					
6	Conv	56x56x256	3x3x256	3701.78	3.89	941.4					
7	Conv + Pooling	56x56x256	3x3x256	3701.78	3.89	941.4					
8	Conv	28x28x256	3x3x512	1850.89	2.22	828.16					
9	Conv	28x28x512	3x3x512	3700.58	4.45	829.58					
10	Conv + Pooling	28x28x512	3x3x512	3700.58	4.45	829.58					
11	Conv	14x14x512	3x3x512	925.15	1.3	709.5					
12	Conv	14x14x512	3x3x512	925.15	1.3	709.5					
13	Conv + Pooling	14x14x512	3x3x512	925.15	1.3	709.5					
Total	-	-	-	30733.9	35.05						

Chapter 2

Comparisons

In this chapter, it is proposed a comparison in terms of performance among the different architectures analyzed so far in the state-of-the-art.

2.1 Algorithm accuracies

All the accuracies arising from the different algorithm implementations are compared. In order to compare correctly the results, the same reference architecture (such as AlexNet) and dataset are chosen.

TOP-1 and TOP-5 errors

The top-1 error comparison is reported in the following plot:

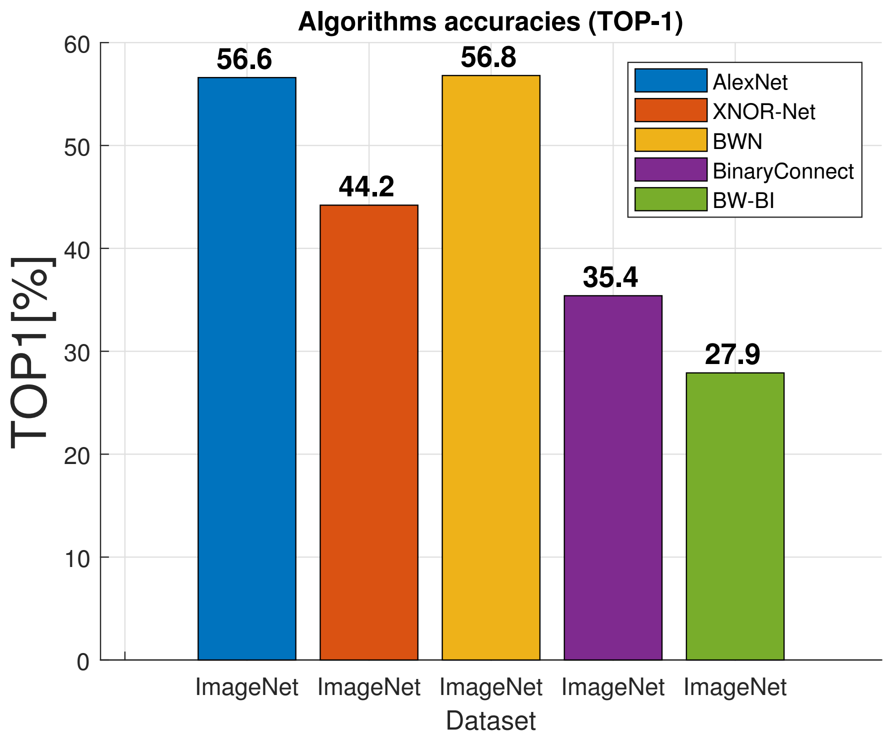


Figure 2.1: Comparison among different algorithms (ImageNet): top-1. AlexNet: [11], XNOR-Net: [12], BWN: [12], BinaryConnect: [13]. BW-BI: [13]

The architecture analyzed is the AlexNet for all the cases and the same dataset (ImageNet) has been chosen. In particular, starting from the left side, the first refers to the classical floating point implementation, in which the accuracy reaches 56.6% [11]. Passing to binarized alternatives, it is possible to observe that the accuracy obtained in the BWN [12] (binary weight network) case is very high: here the weights are binarized with an extra scaling factor, while the inputs are kept in floating point precision. Reducing the precision of the weights and keeping a scaling factor, allow to obtain similar accuracy to the ideal floating point case. If the weights are binarized, the convolution operations (MACs) are transformed into simple additions/subtractions, because they assume only ± 1 values. Similarly for the cases of XNOR-Net [12] (both weights-input binarized with extra scaling factors $\mathbf{K}$ and α), BinaryConnect [13] (only weights are binarized without any extra scaling factors) and BW-BI [13] (both inputs and weights are binarized without scaling factor), it is possible to observe that the first one reaches better results than the others, because some additional terms during the convolution operation are used and they guarantee to reach a good result:

$$Conv_{XNOR} \simeq (\text{sign}(\mathbf{I}) \circledast \text{sign}(\mathbf{W})) \cdot \alpha\mathbf{K} \tag{2.1}$$

BinaryConnect and BW-BI (BNN) can be used in small datasets, while XNOR-Net represents a good trade-off between complexity and accuracy.
In the following figure is reported the top-5 error values for the different networks analyzed:

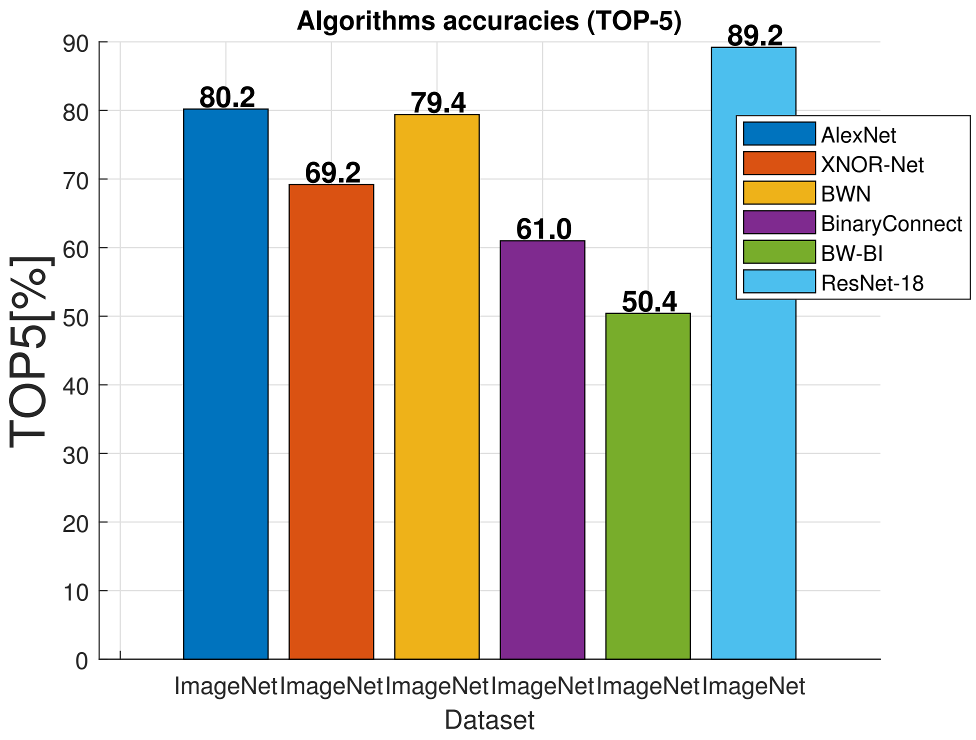


Figure 2.2: top-5 errors for the same dataset ImageNet. AlexNet: [11], XNOR-Net: [12], BWN: [12], BinaryConnect: [13]. BW-BI: [13]

Also in this case depicted in Figure 2.2, the architecture analyzed is the AlexNet for the first five cases: the XNOR-Net reaches a very good performance in term of accuracy. In the last bar, a deeper fully precision convolutional neural network is reported (ResNet-18), in which there are up to 18 convolutional layers: the precision is higher than the AlexNet because increasing the number of layers improves the accuracy. The last graph proposed is the top-1 error in the case of CIFAR-10 dataset, which is far less complicated than ImageNet and so the accuracy is expected to be higher than the previous plot for all the cases:

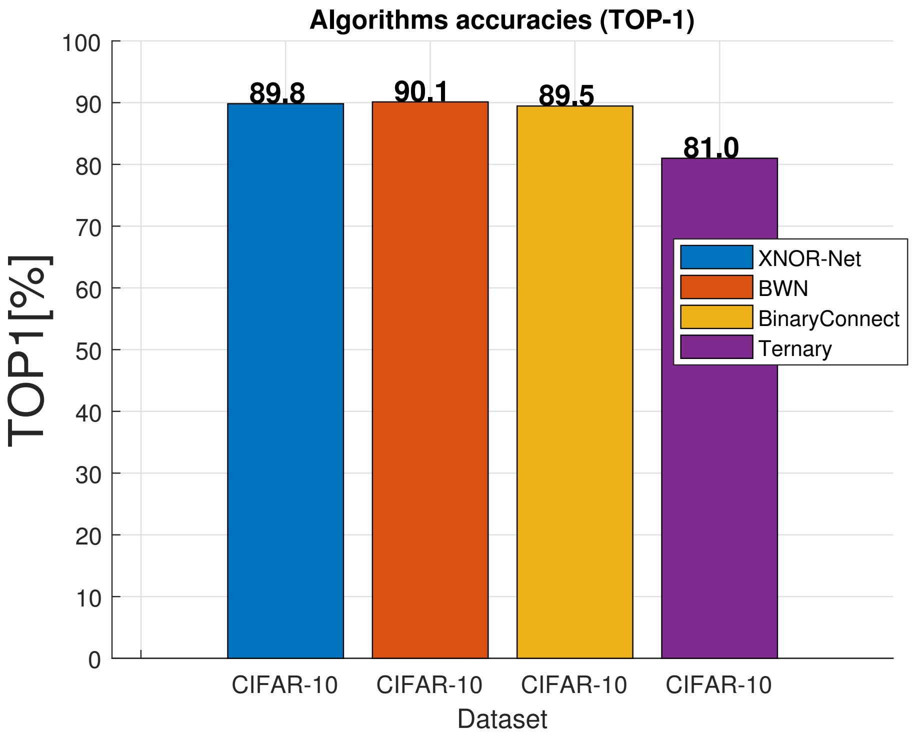


Figure 2.3: Accuracy comparison for CIFAR-10 dataset. XNOR-Net: [12], BWN: [12], BinaryConnect: [13], Ternary: [14]

The ternary network has been analyzed: setting to 0 some of the weights of the network produces an acceptable accuracy result for simple datasets with a lower power.

2.1.1 Performance comparisons

⚠ ATTENTION ⚠

The comparisons presented here are rough estimations! The data reported are reformulated considering linear dependencies. All the specific cases and distinctions are not considered (for example area of the array w.r.t area of the entire chip), since the data provided by the documents may not consider this difference. Some assumptions have been made (as described in the following part), but it is not guaranteed their correctness and accuracy. Unfortunately not all documents analyzed have been considered for the comparison, since some of them don't provide the parameters estimated, because focused on a different topic (for example accuracy). In order to correctly reproduce a more accurate comparison, the individual cases should be reproduced with the same benchmark model (such as AlexNet) and the same neural network type (CNN or MLP).

Number of neurons

The number of neurons has been computed to compare the architectures in terms of performance, in fact it is possible to normalize some network parameters (such as power, energy, area etc.) with different structures. The values obtained with this normalization are expressed as parameter per number of neurons. To do this estimation, consider the AlexNet network as an example:

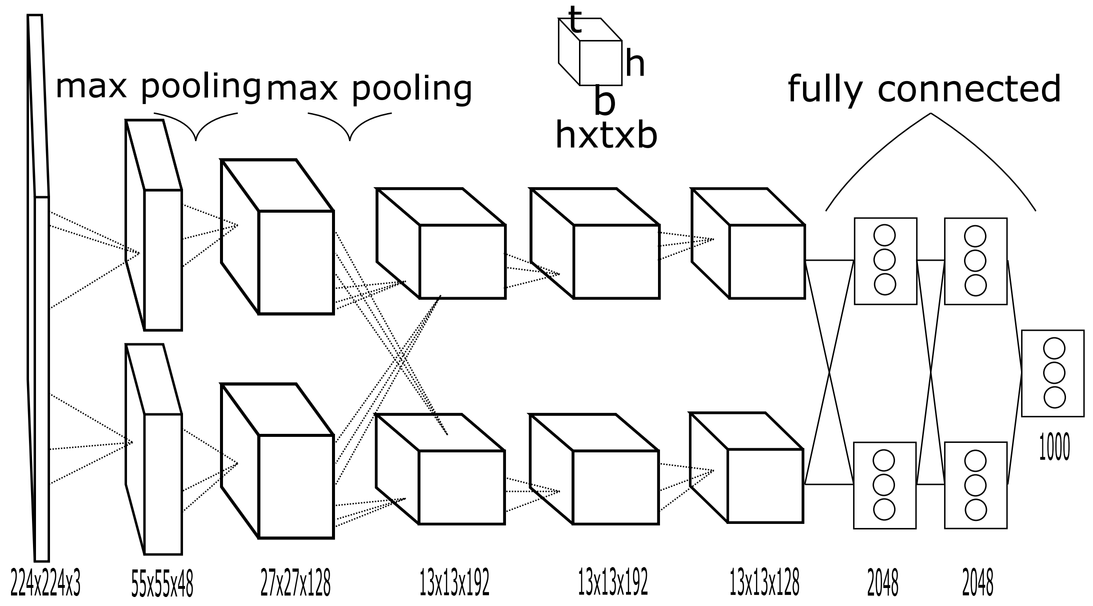


Figure 2.4: AlexNet architecture from [11]

Taking the first layer, the input has 96 feature maps of 55x55 pixels. Each pixel can be considered as a neuron, so the total number of neurons in the input layer is:

$$\#\text{neurons}_{1st} = 55 \times 55 \times 96 = 290400 \tag{2.2}$$

Adding all the number of neurons of individual stages, it is possible to obtain the total value, which results equal to:

$$\#\text{neurons}_{\text{AlexNet}} = 659272 \tag{2.3}$$

This parameter depends on which parameter has to be evaluated: for instance, if area/neuron is considered, the value to use is the effective number of neurons elaborated by the architecture. If instead the energy is considered, this parameter becomes the total number of neurons of a particular benchmark model (AlexNet for instance). In the following parts, each case is analyzed and the number of neurons (real or effective) is chosen.

Number of neurons

The other network's number of neurons has been computed as follows:

1. MLC-STT [15]: by looking at the scheme proposed in section 1.3.1, the convolutional neural network has a number neurons which results equal to:

   $$\#\text{neurons}_{\text{MLC}} = 6 \times 28 \times 28 + 16 \times 10 \times 10 + 84 = 6388 \tag{2.4}$$

   This is a binary neural network (XNOR-Net).

2. SOT [16]: the SOT architecture uses AlexNet BCNN as reference network as described section 1.3.2. The total number of neurons is 659272 as already said. This is a binary neural network (XNOR-Net).

3. OPNE-IPNE [26]: the network realized in [26] is a MLP with 6 PIMs of 484-144 neurons as described in Table 1.5.1. The total number of neurons is given by:

   $$\#neurons_{\text{OPNE-IPNE}} = 6 \times (144 + 484) = 3768 \tag{2.5}$$

   This is a ternary neural network (XNOR-Net).

4. Neurosynaptic core [26]: MLP with 1024x256 structure. The total number of neurons in this case is 256;

5. XNOR-RRAM [19]: only the MLP structure has been considered for the neuron's count. In particular, as reported in section 1.4.2:

   $$\#\text{neurons}_{\text{XNOR-RRAM}} = 3 \times 512 + 10 = 1546 \tag{2.6}$$

   This is a binary neural network (XNOR-Net).

6. Mixed-precision [21]: the network structure is 784-250-10 and so the total number of neurons is 260. Weights are binary while inputs are converted into a voltage range;

7. Synaptic weight [22]: two MLPs of 784x10 single layer and 20 parallel layers

of 784x10 are implemented. The total number of neurons is 10 and 200 respectively. The choice of using parallel arranged layers allows to improve the accuracy. Weights are binary, while inputs are converted into a voltage range;

8. Stochastic [28]: LeNet-5 network topology is considered. The stochastic approach is used only in the first layer of this CNN, so the total number of neurons is obtained as:

$$\#\text{neurons}_{\text{stochastic}} = 32 \times 28 \times 28 = 25088 \tag{2.7}$$

9. HMC [29]: convolutional neural network which is computed by fetching one layer per time into the 3D stacked memory. Since the performance provided in [29] refers to a single convolutional layer per time, a rough estimation can be made considering:

$$\begin{aligned}\#\text{neurons}_{\text{HMC}} =& mean(124 \times 124 \times 3, 120 \times 120 \times 3, 117 \times 117 \times 4,\\ & 114 \times 114 \times 5, 112 \times 112 \times 3, 108 \times 108 \times 3) = 46948\end{aligned} \tag{2.8}$$

This implementation is in floating-point representation.

10. Chain-NN [30]: the first five convolutional layers of the AlexNet fixed-point are used in this implementation, so:

$$\begin{aligned}\#\text{neurons}_{\text{Chain-nn}} =& 55 \times 55 \times 96 + 27 \times 27 \times 256 +\\ & + 13 \times 13 \times 384 + 13 \times 13 \times 256 = 585184\end{aligned} \tag{2.9}$$

11. Energy-efficient [31]: the benchmark model used is VGG-16 (section 1.7.4) with Binary Weight network (BWN) approximation, so α is computed. The

total number of neurons is given by:

$$\begin{aligned}\#\text{neurons}_{\text{ee}} &= 224 \times 224 \times 64 + 112 \times 112 \times 128 + 112 \times 112 \times 128 + \\ &+ 56 \times 56 \times 128 + 56 \times 56 \times 256 + 56 \times 56 \times 256 + \\ &+ 28 \times 28 \times 256 + 28 \times 28 \times 512 + 28 \times 28 \times 512 + \\ &+ 14 \times 14 \times 512 + 14 \times 14 \times 512 + 14 \times 14 \times 512 + \\ &+ 7 \times 7 \times 512 = 9759232\end{aligned} \tag{2.10}$$

It is presented a comparison in terms of number of neurons of the analyzed implementations from the state of the art.

Table 2.1: Number of neurons (real) of the analyzed architectures

Architecture		Number of neurons (real)	Network type	Technology
MLC-STT[15]		6388	XNOR-NET (binary)	MTJ
SOT [16]		659272	XNOR-NET (binary)	MTJ
OPNE-IPNE [24]		3768	XNOR-NET (ternary)	SRAM
Neurosynaptic core [26]		256	Binary weight	SRAM
XNOR-RRAM (MLP) [19]		1546	XNOR-NET (binary)	RRAM
Mixed-precision [21]		260	Binary weight	RRAM
Synaptic weight [22]	1 layer	10	Binary weight	RRAM
	20 layers	200	Binary weight	
Stochastic [28]		25088	Fixed point	OOM
HMC [29]		281688	Floating point	OOM
Chain-NN [30]		585184	Fixed point	OOM
Energy efficient [31]		9759232	BWN	OOM

Normalized energy

In Figure 2.5 it is reported a comparison between the normalized energy of different architectures:

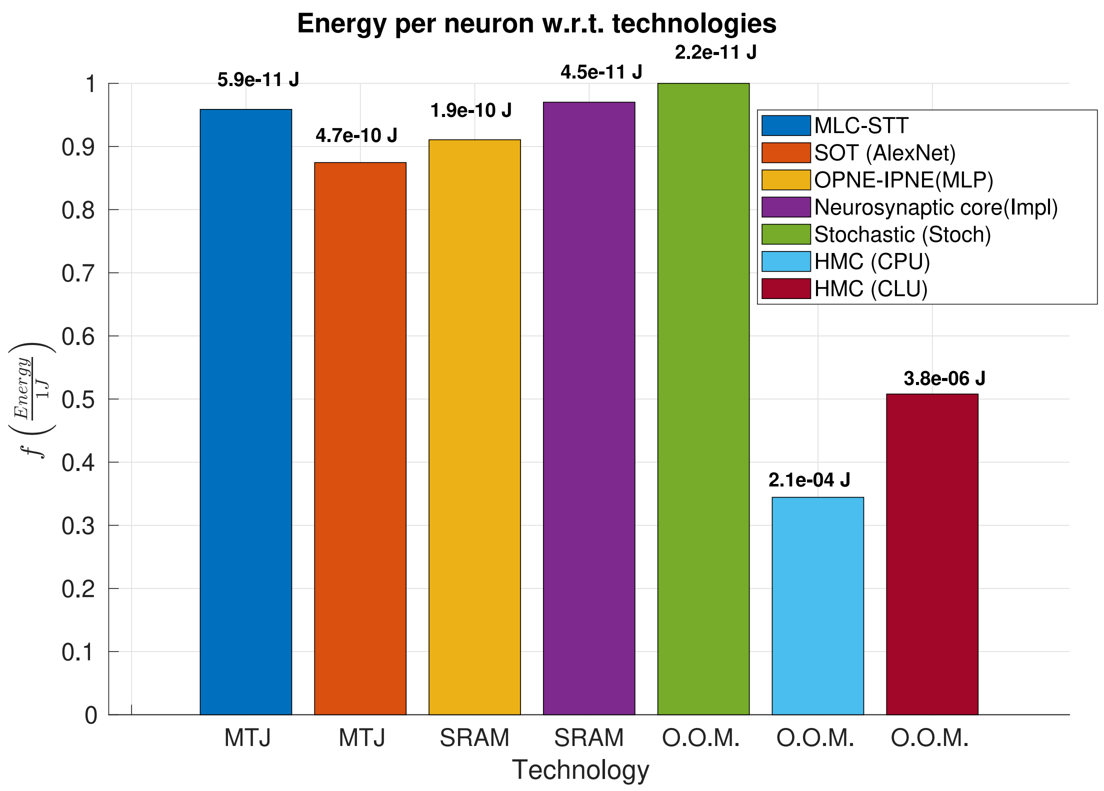


Figure 2.5: Energy comparison: the higher is better. MLC-STT: [15], SOT: [16], OPNE-IPNE: [40], Neurosynaptic core: [26], Stochastic: [28], CPU-CLU: [29]

In this case (and in the all the others analyzed after), it is used a normalization which, scales the different energies between 0 and 1. The value that reaches 1 has better performance in terms of energy. Since the energy values per number of neurons are very small, on Y axis it is used a logarithm function as follows:

$$\text{NormEnergy} = \frac{\text{Energy[J]}}{\#neurons} \tag{2.11}$$

$$f\left(\frac{\text{Energy}}{1J}\right) = \frac{log\left(\frac{\text{NormEnergy}}{1J}\right)}{min\left[log\left(\frac{\text{NormEnergy}}{1J}\right)\right]} \tag{2.12}$$

The minimum value of the logarithm at the denominator is a negative value and so the bar chart is projected from negative Y values to positive, resulting into the Figure 2.5. The absolute energy values of the different architectures are the following:

Table 2.2: Energy values picked from the documents. The HMC values are computed by taking the sum of the reported energies and the network considered is the entire CNN proposed by [29]. The OPNE-IPNE energy value has been computed considering [40]: in the result section, the total energy of 0.73J has been computed considering 1 million transaction of the MNIST dataset. This value has been obtained by dividing the original value of 0.73J by 1 million.

Architecture		Number of neurons	Energy [J]	Energy/neuron [J]
MLC-STT		6388	380.0n[15]	59.48p
SOT		659272	310.4μ[16]	470.9p
OPNE-IPNE		3768	730.0n [40]	193.7p
Neurosynaptic core		256	-	45.0p[26]
Stochastic		25088	542.4n [28]	21.6p
HMC	CPU	281688	59.8 [29]	0.212m
	CLU	281688	1.1 [29]	3.83μ

The results of the in-memory implementations are compatible with the expectations:

- MLC-STT: since this architecture is based on MTJs which are analog components, the energy consumption is not heavily influenced because the operations are performed with currents manipulations and resistance variations in the mesh array. A single neuron consists into a couple of resistances and a modified sensing circuit that performs the product computation required in the XNOR-net. Additional external units are employed to compute the batch normalization, pooling, K and α coefficients.

- SOT: similarly to the previous case, the neuron is composed by a single MTJ, so two active cells are needed to compute a logic function at the same time, requiring more energy. The computation is performed by a simple current comparator, which performs all the logical functions required.

- OPNE-IPNE (SRAM) [24]: the computation cycle in [40] consists in alternating OPNE and IPNE computation. Once OPNE part finishes, IPNE starts to produce a serial output. The serial output is then fetched from the following OPNE, which starts its computation concurrently:

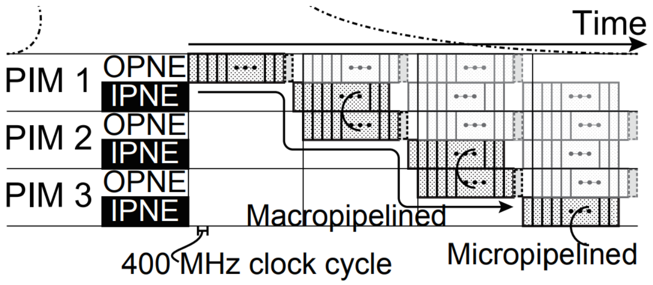


Figure 2.6: Macro-pipeline structure [40]. Once OPNE terminates, IPNE starts producing a serial output: this is elaborated by the following OPNE.

All the components are working in parallel until the data-stream is not finished, and considering that an OPNE and an IPNE are composed by XNOR gates and adders to compute the pop-counting operation, this implies higher energy consumption. This network is composed by 484 OPNEs, each of them with a XNOR, an adder and a register for accumulation. The 144 IPNEs instead are composed by 144 XNORs and an adder tree: by iterating this consideration to the overall dimension of the neural network, the energy will reach 730 nJ for a single dataset transaction. Considering the energy/neuron, this will be worse than the MLC solution (probably because of its compact structure and multiple bits into a single cell) but better than SOT.

- Neurosynaptic core: this is based on an analog computation, which updates the membrane potential of the neuron, producing a spike under a particular condition. SRAM is used as an analog component, reducing the energy required.

Considering now the OOM architectures:

- Stochastic [28]: produces the best result in term of energy required per neuron, because of its simple computation. The bit-stream length is 8 bit and adder/multiplier are replaced by a simple AND gate/multiplexer which are able to reduce the energy of the system. The architecture is in fixed point precision and only one layer of the chosen CNN is performed with a stochastic approach.

- HMC [29]: since the structure of this network is based on hybrid-memory cube, the data travels along distances that are reduced by means of TSV. The big drawback in terms of energy of the in-memory structures are the internal interconnections, which are based on very long bit-lines/word-lines. By looking at HMC structure, it is possible to see that the data are fetched from DRAM layers and computed in the floating point units and since the computation is divided in vaults, all the logic units perform the convolution simultaneously. The total number of vaults is 16, so considering the worst case all the computational units are active at the same time. As expected, these architectures have the worst resulting energies respect to the other cases: the CLU has better perfomances that the CPU one.

Normalized latency: rough estimation

Also an indication on latency is given in some papers. In order to evaluate properly the differences between the architectures, the following procedure has been used:

1. A neural network (CNN or MLP) has a certain number of layers, which is determined by simply counting them. In the case of CNNs, pooling and normalization layers are not considered;

2. The architectures analyzed in the documents are not the same! In certain cases, the dimensions of the array influence the delay (for example the XNOR-RRAM implementation in section 1.4.2). In these cases, a normalization has to be considered, but the dependency between delay-array size could not be linear. Each architecture is able to process a convolutional window with a certain number of neurons processed per time. This number influences the

array dimensions (in case of in-memory architecture). This problem is also present in MLPs, because they can be considered as convolutional networks too. The number of effective neurons processed per time could normalize the resulting delays by rescaling the networks to the same dimensions.

A rough estimation has been done considering all the cases:

- MLC-STT [15]: by looking at MLC architecture in section 1.3.1, it is possible to observe that there are 7 layers. Excluding pooling layers and the last output layer (simply gives a classification):

$$\#\text{layers}_{\text{MLC-STT}} = 4 \tag{2.13}$$

The number of neurons depends on which layer are considered during the actual computation, so a mean value can be computed:

$$\#\text{neurons}_{\text{MLC-STT(eff)}} = mean(6 \times 28 \times 28{,}16 \times 10 \times 10{,}120{,}84) = 1627 \tag{2.14}$$

The delay value given by [15] is a cycle time, in which a single convolution is performed. It is equal to 27.24ns and it can be considered as a time/(layer*neurons), since only one convolution provides the output of a single neuron, so layer-neurons normalizations are not required.

$$\text{Delay}_{\text{normalized(MLC)}} = 27.24ns \tag{2.15}$$

- SOT [16]: the SOT architecture uses AlexNet, so the total number of layers is 8. Also here a mean value is considered, that indicates how many neurons are processed per time:

$$\begin{aligned}\#\text{neurons}_{\text{SOT(eff)}} = mean(&55 \times 55 \times 48 \times 2{,}27 \times 27 \times 128 \times 2,\\ &13 \times 13 \times 192 \times 2{,}13 \times 13 \times 192 \times 2{,}13 \times 13 \times 128 \times 2,\\ &2048 \times 2{,}2048 \times 2{,}1000) = 82409\end{aligned} \tag{2.16}$$

The delay given by [16] is 10.7 ms, which is the total delay with batch normalization, scaling factors and convolution computation. This value can be divided by the number of layers and the neurons:

$$\text{Delay}_{\text{normalized(SOT)}} = \frac{10.7ms}{8 \cdot 82409} \simeq 16.3ns \tag{2.17}$$

- OPNE-IPNE [40]: since the network is an MLP with 6 PIMs of 484-144 neurons, the total number of layers is 13. The number of neurons processed per time is always 484 (by observing the macro-pipelined structure in Figure 2.6), so:

  $$\#neurons_{\text{OPNE-IPNE(eff)}} = 484 \tag{2.18}$$

  The clock frequency of 400MHz is given and for this estimation, 1 neuron computes its output after 484 clock cycles, in fact OPNE computation constrains the required time as indicated in Figure 2.6:

  $$\text{Delay}_{\text{normalized(OPNE-IPNE)}} = f_{ck} \times 484 = 2.5ns \times 484 \simeq 1.21\mu s \tag{2.19}$$

- Neurosynaptic core [26]: with an MLP structure, the total number of neurons is always 256 with only 1 layer. The delay of 1ms indicated by [26] is a cycle time, and so every millisecond the output is evaluated:

  $$\text{Delay}_{\text{normalized(Neurosynaptic)}} = \frac{1ms}{256} \simeq 3.91\mu s \tag{2.20}$$

- XNOR-RRAM [19]: here the case is a little bit different, because the CNN is implemented with subarrays having the same dimensions (128x128 as already said in Table 1.4.2) [19] arranged in parallel. Their outputs are fetched by the remaining logic, that computes the convolution. Also the MLSA bit level influences the delay value because of its complexity. For the sake of simplicity, the total number of layers in the MLP implementation is considered, which results equal to 4, while the total number of neurons is given by:

  $$\#\text{neurons}_{\text{XNOR-RRAM(eff)}} = mean(512{,}512{,}512{,}10) \simeq 386 \tag{2.21}$$

The value given by [19] is the delay of a single subarray of 128x128, without considering the cost of the decoding and other computations normally executed in a XNOR-Net, and it is equal to 16.69ns, which is already the delay required for the computation of a single neuron, so:

$$\text{Delay}_{\text{normalized(XNOR-}\underline{\text{RRAM}}\text{)}} = 16.69ns \tag{2.22}$$

- HMC [29]: since the architecture is fetching one layer per time into the 3D stacked memory, an average delay is estimated, given by [29] and divide it by 1. [29] provides two delay results: one referred to a CPU floating point implementation while the other to HMC(CLU) (already specified in section 1.7.2). The mean number of neurons processed per time is given by:

$$\begin{aligned}\#\text{neurons}_{\text{HMC(eff)}} =& mean(124 \times 124 \times 3{,}120 \times 120 \times 3{,}117 \times 117 \times 4,\\ & 114 \times 114 \times 5{,}112 \times 112 \times 3{,}108 \times 108 \times 3) = 46948\end{aligned} \tag{2.23}$$

So the corresponding delays are computed from the data provided in seconds:

$$\text{Delay}_{\text{normalized(HMC-CPU)}} = \frac{1.149 + 1.0808 + 0.6643 + 0.832 + 0.808 + 0.873 + 0.809}{7 \cdot 46948} \simeq 18.91\mu s \tag{2.24}$$

$$\text{Delay}_{\text{normalized(HMC-CLU)}} = \frac{0.0138 + 0.0133 + 0.0086 + 0.0165 + 0.0091 + 0.0105 + 0.0102}{7 \cdot 46948} \simeq 249.52ns \tag{2.25}$$

- Chain-NN [30]: only 5 convolutional layers of <u>AlexNet</u> are used in [30]. The delay given by [30] of 353.17ms is a total delay, so it has to be divided by the total number of layers and number of neurons. As specified in [30], the total number of neurons processed per time is equal to $C_{in} \times (2K - 1) \times w_{in}$ where

K is the kernel size of the AlexNet. Also here an average is considered:

$$\begin{aligned}\#\text{neurons}_{\text{chain-nn(eff)}} =& mean(224 \times (2 \times 11 - 1),55 \times 96 \times (2 \times 5 - 1),\\ & 27 \times 256 \times (2 \times 3 - 1),13 \times 384 \times (2 \times 3 - 1),\\ & 13 \times 384 \times (2 \times 3 - 1)) \simeq 27340\end{aligned} \tag{2.26}$$

The resulting delay is:

$$\text{Delay}_{\text{normalized(Chain-NN)}} = \frac{353.17ms}{5 \cdot 27340} \simeq 2.58\mu s \tag{2.27}$$

- Energy-efficient [31]: VGG-16 structure has 13 layers and the value given by [31] of 35.1ms is the total delay of the architecture that has to be divided by the total number of layers and by the number of neurons. Considering the overall benchmark model, the effective number of neurons can be computed as:

$$\begin{aligned}\#\text{neurons}_{\text{ee(eff)}} =& mean(224 \times 4 \times 64,112 \times 4 \times 128,112 \times 4 \times 128,56 \times 4 \times 128,\\ & 56 \times 4 \times 256,56 \times 4 \times 256,28 \times 4 \times 256,28 \times 4 \times 512,\\ & 28 \times 4 \times 512,14 \times 4 \times 512,14 \times 4 \times 512,14 \times 4 \times 512,\\ & 7 \times 4 \times 512) \simeq 43008\end{aligned} \tag{2.28}$$

So the delay is:

$$\text{Delay}_{\text{normalized(ee)}} = \frac{35.1ms}{13 \cdot 43008} \simeq 63ns \tag{2.29}$$

In order to compare the values obtained, a bar plot is provided, in which on the Y axis there are the normalized latency values rescaled from 0 to 1 as already done in the previous case:

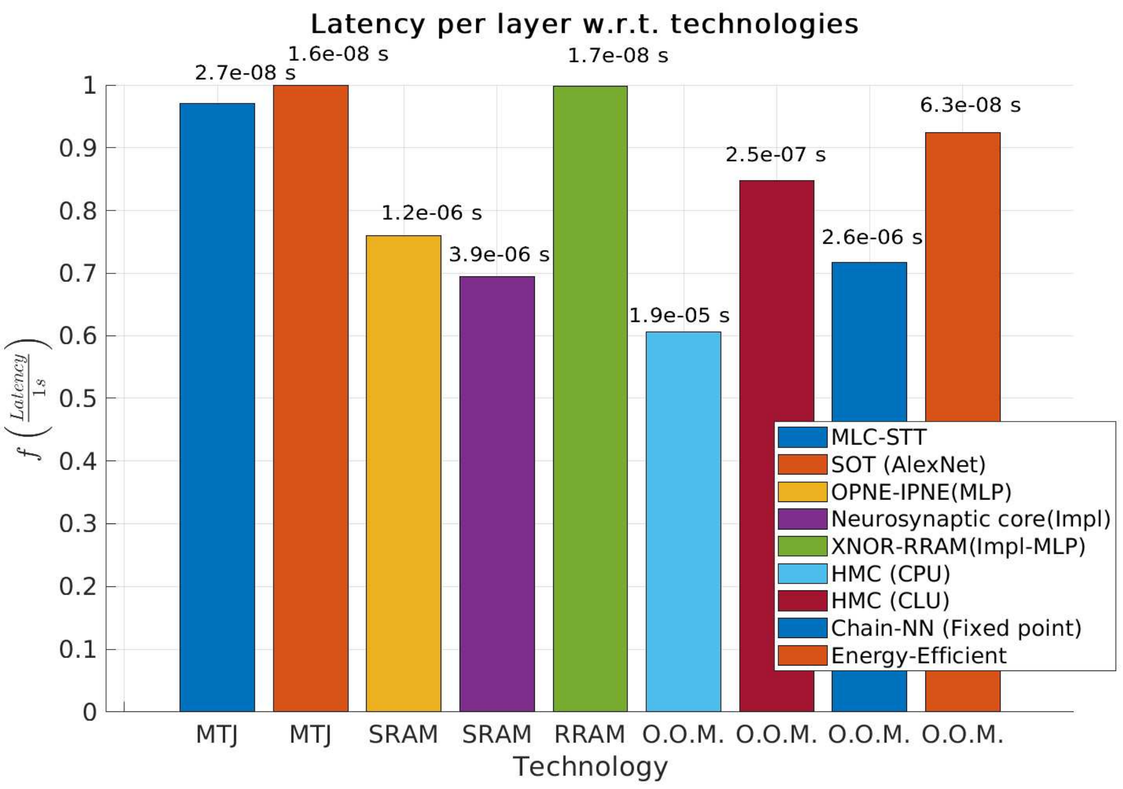


Figure 2.7: Delay comparison: the higher is better. MLC-STT [15], SOT [16], OPNE-IPNE [40], Neurosynaptic core [26], XNOR-RRAM [19], HMC [29], Chain-NN [30], Energy-efficient [31]

1. MLC-STT [15]: The delay obtained is very small, because of the internal structure: in fact multiple bits are stored in the same cell and this reduces the costs. The network used is smaller than the other ones, and this can affect the delay, since the bit-lines/word-lines lengths increases with network's complexity. All the computations are done in parallel, so a convolutional window is computed by multiple CIM arrays and this speeds up the execution time. Since it is an analog solution based on current levels comparison to compute the logic operations, this solution will be faster than a digital one: in general this concept is valid for all the analog solutions discovered;

2. SOT [16]: This case is very similar to the previous one, since it is an analog solution. The cells are composed by a single MTJ [16] and multiple cells are

activated simultaneously to perform a logic function. In general, the performance is expected comparable to the previous case based on MLC;

3. OPNE-IPNE [40]: the cost of the neuron's intrinsic structure (composed by three ram cells, a XNOR, adder and register as already said) is heavy in terms of delay. The architecture is constrained by the clock frequency, the corresponding critical path and the OPNE computation, which influences the execution time.

4. Neurosynaptic core [26]: the delay is determined by the cycle time of 1ms, so an entire result is produced with a frequency of 1kHz. The architecture is based on the neuron's membrane potential update, which is an analog approach based on comparators that slows the system;

5. XNOR-RRAM [19]: better results are obtained in this case thanks to parallel computation with multiple sub-arrays. A logical operation is performed by fetching data from two RRAM cells and the resulting current is compared with a reference. Also in this case, the considerations made for an analog solution are valid;

6. HMC [29]: this case is very interesting, because also the performance that will be obtained with a CPU are reported. Here the computations are in floating point and, as expected, the CPU case is the worst in terms of execution time. The usage of a 3D memory stack (CLU) allows to reduce the execution time due to the reduced wire lengths: the main characteristic of 3D memories with logic, allowing to obtain better performance than OPNE-IPNE case, because by reducing the wire length, the clock frequency can be higher;

7. Chain-NN [30]: good result is obtained also by the Chain-NN, which is a pipelined structure in fixed point representation. The pipeline allows to reduce the critical path delay and the fixed-point numbers reduce the computational cost;

8. Energy-efficient [31]: this architecture is a BWN OOM which is able to reduce the data fetching from the external memory by using efficient techniques. Computations are performed in an approximate form (CA1 instead of CA2) with

error compensation: this reduces the multiplier by ±1 into a simple structure as indicated in section 1.7.4. The usage of simpler logical structures allows to reduce the clock frequency required and to speed up the operations: the delay is comparable to an analog solution.

Normalized area: rough estimation

Also in this case, a similar approach has been applied. The area taken from the documents has been divided by an effective number of neurons that a specific architecture process per time, in order to obtain a normalized area per number of neurons which results comparable. The areas are listed below:

Table 2.3: Area values of different architectures.

Architecture	Number of neurons (eff)	Area [mm^2]	Area/# neurons [mm^2]
SOT [16]	82409	5.28	64.1×10^{-6}
OPNE-IPNE [40]	3768	3.90	1.0×10^{-3}
Neurosynaptic core [26]	256	4.20	16.4×10^{-3}
XNOR-RRAM [19] (MLP)	386	1.686	4.36×10^{-3}
Stochastic [28]	25088	1.32	52.7×10^{-6}
HMC [29]	46948	729.00	15.5×10^{-3}
Energy-efficient [31]	43008	44.96	1.0×10^{-3}

The number of neurons of the OPNE-IPNE [40] are the real number of neurons, since the area refers to the entire implementations with 6 PIMs. Same for Neurosynaptic core. The stochastic case instead considers only the first layer of the CNN, so also in this case, the effective number of neurons are the real one already calculated. The other cases' values are picked from the previous part. All the areas indicated in the table are obtained from the papers, exception for the XNOR-RRAM, that has been computed as:

$$\text{Area}_{\text{XNOR-RRAM}} = \#\text{sub-arrays} \cdot \text{Area}_{\text{subarray(128x128)}} = 36 \times 46824.1\mu m^2 \simeq 1.69mm^2 \tag{2.30}$$

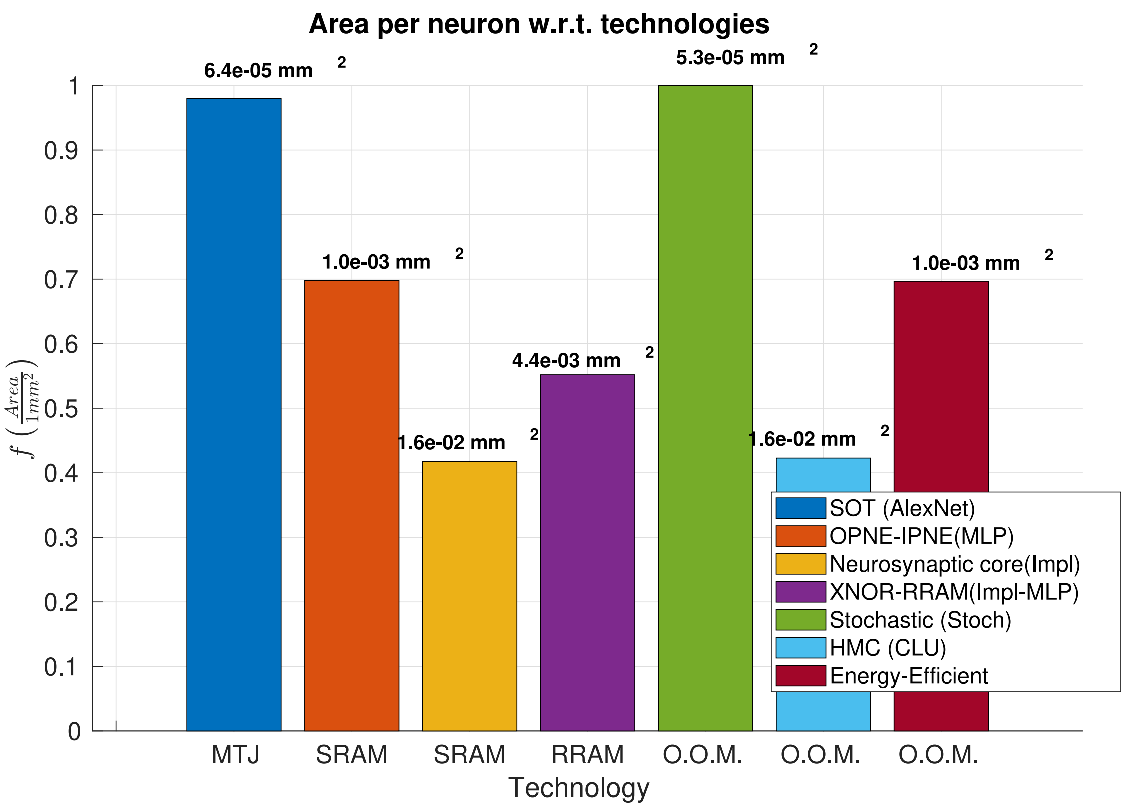


Figure 2.8: Area comparison: the higher is better. SOT [16], OPNE-IPNE [40], Neurosynaptic core [26], XNOR-RRAM [19] (MLP), Stochastic [28], HMC [29], Energy-efficient [31]

The best results are obtained in the stochastic and SOT [16] case, since the stochastic [28] is very efficient in terms of computational blocks used. A convolution operation (which is composed by multiply-and-accumulate sequences) is simply realized with ANDs (multiply) and multiplexers (accumulate). Probably this solution is also the slowest one, since the number si transformed into a bit sequence (for example the number 15 represented on 4 bits is transformed into a sequence of 15 bits). The SOT case allows to reach a good area performance, since the structure of the array is composed by a single MTJ. HMC [29] has bad area performance, due to its 3D structure and floating point representation, in fact the whole chip has an area of 729 mm^2, in fact floating point units are required and this increases the area occupation. OPNE-IPNE [40] value is very good, considering the neuron's structure and has similar performance to the XNOR-RRAM [19] case. The area provided by

[40] is the chip area, and so it considers also the control circuit. The neurosynaptic core [26] occupation is degraded by the analog circuits that perform the membrane potential update: this is a slow, big but energy efficient solution. Last but not least, the Energy-efficient solution [31] allows to reach good perfomances in terms of area (since the implemented neural network is a Binary weight network, so **K** computation is not performed).

2.1.2 Conclusions

1. MLC-STT: very good in terms of energy/latency. Since it is an analog solution, it is subjected to noise errors and unwanted parasitic effects. Small networks (or array partitioning) can be realized with this solution;

2. SOT: good in terms of energy/latency/area. Same considerations of MLC are valid;

3. OPNE-IPNE: good energy/latency/area. Since it is a synchronous architecture, latency is influenced by critical path but the calculation accuracy is higher than the previous cases since it is all implemented in digital;

4. Neurosynaptic core: worse latency/area resuts but good energy achieved. The motivations have been already explained previously;

5. XNOR-RRAM: very good in terms of latency and area, so a fast solution can be realized with RRAM technology. Partitioning is required to reduces the parasitic effects;

6. HMC (CLU): worse area/latency/energy performance w.r.t the others, but reaches the highest precision due to floating point computations. It is an interesting application of a 3D memory, that allows to reach very good performance w.r.t. the CPU based neural network implementation;

7. Energy-efficient: Good in terms of area/latency due to its optimizations. Computations are performed in fixed point and multipliers are replaced by approximated adders, in order to reduce power/latency and energy.

Chapter 3

Software implementation

The neural network type that has been chosen is the XNOR-Net with MNIST dataset, since it has good trade-offs in terms of accuracy, power and latency. The steps used to implement a XNOR-Net are the following:

1. Neural network implementation and training with Python using Tensorflow and Keras;

2. Parameters extracted from the Python implementation are fed to a MATLAB model for verification purposes.

The chosen dataset is the smallest one, because of the shorter simulation/synthesis time required by the VHDL model.

3.1 Network model

The neural network model that has been used in this analysis is the following one:

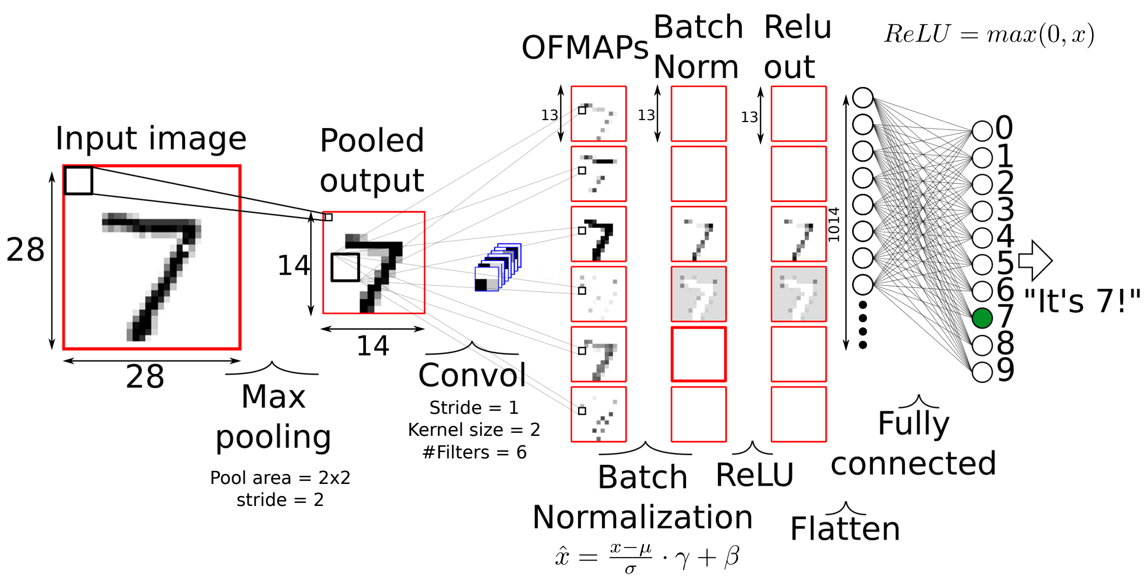


Figure 3.1: Neural network model

As it is possible to see, the network structure is composed by **input-image**, **max-pooling**, **convolution**, **batch normalization**, **ReLU**, **flatten** and **fully connected** layers. All the layers are zero-padding, it means that the dimensions of the feature maps, stride and filters have been chosen accordingly, avoiding input-resizing. In the following part is presented a description of the layers, but before some notations are introduced:

Symbol	Description
w_{in}	Input image dimension
w_{filter}	Kernel window dimension
w_{out}	OFMAP size, output dimension
c_{out}	Number of output channels
c_{in}	Number of input channels
h	Height of the memory
w	Width of the memory

Table 3.1: Notations used

- The **input layer** is a matrix of dimension 28x28x1 pixels which represents a digit from MNIST handwritten digit dataset;

- **Max-pooling** layer is the first computational layer encountered in the neural network. The parameter used in this layers are the following:

$$\begin{aligned}
&w_{in} = 28\\
&w_{filter} = 2\\
&stride = 2\\
&w_{out} = \frac{w_{in} - w_{filter}}{stride} + 1 = \frac{28-2}{2} + 1 = 14
\end{aligned}$$

  The pooling layer has been placed before the convolutional layer: this technique called earlier pooling already used in [31], decreases the power required and computational complexity in the following layers, since it reduces the dimensions of the input from 28x28 to 14x14 and so the convolutional layer has to process a smaller input image, without losing too much precision;

- **Convolutional layer** takes the pooled image and convolves it with 6 different kernels and so 6 OFMAPs are obtained, one for each kernel. The parameters used in this layer are the following:

$$\begin{aligned}
&w_{in} = 14\\
&w_{filter} = 2\\
&stride = 1\\
&w_{out} = \frac{w_{in} - w_{filter}}{stride} + 1 = \frac{14-2}{1} + 1 = 13
\end{aligned}$$

  Here no bias values have been applied in order to reduce the complexity;

- After the convolutional layer, **Batch normalization** (”batchnorm”) is realized. Each IFMAP is normalized w.r.t mean and variance that are computed over a batch. Batchnorm layer is useful during the training phase, because it speeds up the convergence of the training algorithm (SGD for example) and improves stability. Another two learnable factors called γ and β are considered in the batchnorm, producing the following output:

$$\hat{x} = \frac{x - \mu}{\sigma} \cdot \gamma + \beta \tag{3.1}$$

- **ReLU** is the activation function used in the network to allow better training performances. Compared to other activation functions, this is the simplest one, since it simply takes the maximum between 0 and its input. In hardware, this is simply realized by a multiplexer selecting between input and 0 based on the sign of the input itself.

$$ReLU = max(0,x) \tag{3.2}$$

- **Flatten** layer transforms the inputs (IFMAPs) into a vector, so if 6 IFMAPs of 13x13 pixels are considered, the output vector dimension is given by:

$$w_{out} = w_{in}^2 \cdot c_{out} = 13 \cdot 13 \cdot 6 = 1014 \tag{3.3}$$

  This vector is fed to the fully connected part (MLP);

- **Fully connected** takes the output of the flatten layer and by means of a MLP with size 1014-10 gives the classification in output. The highest result coming from the last 10 neurons corresponds to the output classification.

3.2 Network's computational model

As already said, the network's model is a XNOR-Net, it means that the convolution is approximated as

$$\mathbf{I} * \mathbf{W} \approx (sign(\mathbf{I}) \circledast sign(\mathbf{W})) \cdot \mathbf{K}\alpha \tag{3.4}$$

Where $\circledast$ represents XNOR-Bitcount operations and K and α are defined as:

$$\mathbf{K} = |\text{Input}| * \begin{bmatrix} \frac{1}{w_{filter}^2} & \frac{1}{w_{filter}^2} & \cdots \\ \frac{1}{w_{filter}^2} & \frac{1}{w_{filter}^2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \tag{3.5}$$

The size of the $\frac{1}{w_{filter}^2}$-matrix is the same of the kernel's one.

$$\alpha = \frac{\sum_{i=1}^{N} |W_i|}{N} \tag{3.6}$$

As it is possible to see, **K** is a simple matrix containing the same elements, while α is a scalar. Here it is reported an example:

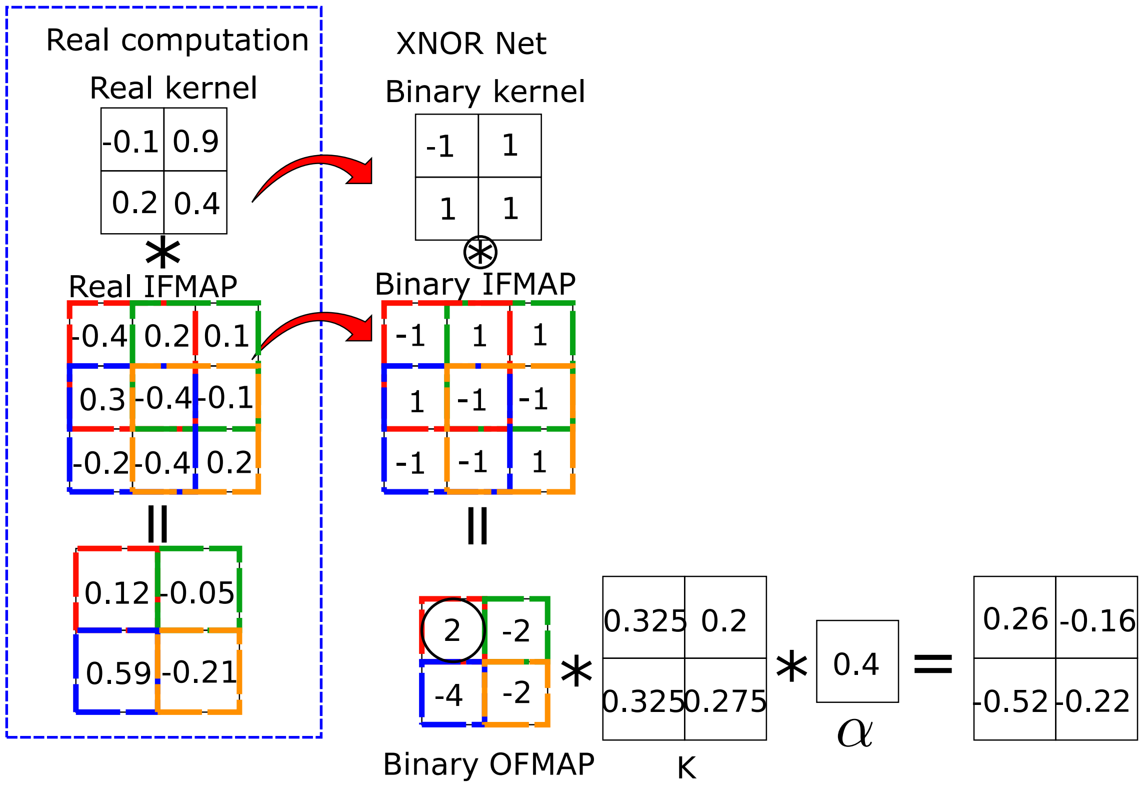


Figure 3.2: Xnor net computation example

Two computations are reported: the real case, which simply executes the sum of products of the kernel with the windowed part of the input and the XNOR net. The steps to compute the output in the second case are:

1. Computation of α as the mean of the absolute sum of the kernel elements;

2. Computation of **K** as the windowed part of input convolved with the matrix

defined before. In this case the computation is defined as:

$$K(1{,}1) = |-0.4| \cdot \frac{1}{2^2} + |0.2| \cdot \frac{1}{2^2} + |0.3| \cdot \frac{1}{2^2} + |0.4| \cdot \frac{1}{2^2} = 0.325 \qquad (3.7)$$

The other values are obtained with the same approach. **K** is a mean of the input sub-matrix considered by the convolutional window. If more than one input channels are evaluated, **K** is computed as the element-wise absolute sum over all the IFMAPs divided by the number of channels and convolved with the $\frac{1}{w_{filter}^2}$-matrix defined before:

$$\mathbf{K} = \frac{\sum_{c=1}^{c_{in}} |\text{Inputs}(:,:,\text{c})|}{c_{in}} * \begin{bmatrix} \frac{1}{w_{filter}^2} & \frac{1}{w_{filter}^2} & \cdots \\ \frac{1}{w_{filter}^2} & \frac{1}{w_{filter}^2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \qquad (3.8)$$

3. Binarization of inputs/weights by taking the sign. When the input/weight is 0, the sign function returns -1, so:

$$Binarize(x) = \begin{cases} -1, \text{ when } x \leq 0 \\ +1, \text{ when } x > 0 \end{cases} \qquad (3.9)$$

4. Binary convolution between the binary-input and binary kernel. Considering, for example, the first result, this step is perfomed as:

$$\text{BinConv}(1{,}1) = -1 \cdot (-1) + 1 \cdot 1 + 1 \cdot 1 + (-1) \cdot 1 = 2 \qquad (3.10)$$

5. Xnor-convolution: the output is then computed by the element-wise multiplication between the binary OFMAP and **K**. This new matrix is then multiplied by the scalar alpha. Considering the first element of the OFMAP:

$$\text{OFMAP}(1{,}1) = \text{BinConv}(1{,}1) \cdot \alpha \cdot K(1{,}1) = 2 \cdot 0.4 \cdot 0.325 \simeq 0.26 \qquad (3.11)$$

These assumptions are valid in the case of a convolutional layer. If the fully connected layer is considered, the **K** computation is different, since the sub-matrix considered has the size of the entire input.

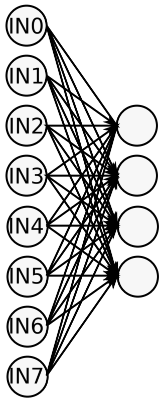


Figure 3.3: Fully connected layer - toy example

In order to compute **K**, the thing to consider is that the actual dimension of the kernel is equal to the number of input neurons. **K** becomes a scalar which is simply given by the mean of absolute input values:

$$\mathbf{K}_{fc} = \frac{\sum_{i=1}^{N} |I_i|}{N} \tag{3.12}$$

3.2.1 Python code

A software implementation of the neural network proposed has been realized in Python and the source code is based on [51], since python with Tensorflow and Keras allow a very easy and straight-forward realization and training of every kind of neural network (from CNNs to MLPs). In order to reduce the complexities of the synthesis-simulations of the VHDL implementation, the easiest CNN structure has been chosen, that contains all the most used components in a neural network: **max pooling**, **convolution**, **batch normalization**, **ReLU**, **flatten** and **fully connected**. It is reported an extract of the python code used from [51]:

1. # nn parameters: specification of the parameters used in the neural network. The *batch_size* is the total number of images that passes at the same time during forward/backward propagation in the training process; *epochs* parameter

specifies the number of times the entire training batch feed-forwards the neural network. The training size in this case is equal to 60000; *nb_channels*, *img_rows* and *img_cols* specify the input dimensions, which are equal to 28x28x1; *nb_classes* indicates the total number of classifications that can be obtained in output, so if MNIST is used, 10 classes can be recognized (from 0 to 9).

```
from binary_ops import binary_tanh as binary_tanh_op
from xnor_layers import XnorDense, XnorConv2D
H = 1.
# nn parameters
batch_size = 10
epochs = 5
nb_channel = 1
img_rows = 28
img_cols = 28
nb_classes = 10
use_bias = False
```

2. # Learning rate schedule specify the behavior of the η learning rate during the training phase;

```
# learning rate schedule
lr_start = 1e-3
lr_end = 1e-4
lr_decay = (lr_end / lr_start)**(1. / epochs)
```

3. # BatchNorm: the parameters specified for the batch normalization layer are *epsilon* and *momentum*. The *epsilon* is an additive term that allows to improve the stability during the training process and it is used as follows:

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma + \beta \tag{3.13}$$

In the VHDL implementation, this term has been neglected, since it is very small ($\sim 10^{-5}$). The paramenter *momentum* indicates how the BatchNorm computes the mean and variance in each iteration step. In particular a *momentum* of 0.9 is translated as:

$$\begin{aligned}\mu(t) &= (1 - momentum) \cdot \mu(t-1) + momentum \cdot \text{batch\_mean}(t) \\ &= 0.1 \cdot \mu(t-1) + 0.9 \cdot \text{batch\_mean}(t)\end{aligned}$$

momentum parameter allows to consider the previous mean value, improving stability.

```
# BatchNorm
epsilon = 1e-6
momentum = 0.9
```

4. # MNIST loading: loads the MNIST dataset, which is composed by images of 28x28 pixels in range 0 to 255. The total numbers of training images and testing images are set and then scaled between 0 and 1.

```
# MNIST loading
(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.reshape(60000, 1, 28, 28)
X_test = X_test.reshape(10000, 1, 28, 28)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train/ 255
X_test = X_test/ 255

# convert class vectors to binary class matrices
Y_train = np_utils.to_categorical(y_train, nb_classes) * 2 - 1 #
     -1 or 1 for hinge loss
Y_test = np_utils.to_categorical(y_test, nb_classes) * 2 - 1
```

5. # Neural network realization: each layer is added sequentially. A XNOR-Net is not a standard network, as a consequence ad-hoc layers are designed, such as the convolutional and fully connected ones. In the code, they are called XnorConv2D and XnorDense, which executes the operations already described

of computing α and $\mathbf{K}$. The Optimizer option is set to Adam already described in section 1.2.4.

```
# Neural network realization
model = Sequential()
model.add(MaxPooling2D(pool_size=(2, 2),strides=(2,
     2),name='pool1',input_shape=(nb_channel, img_rows,
     img_cols)))
model.add(XnorConv2D(8, kernel_size=(2, 2),strides=(1,1),H=H,
                     padding='valid',
                          use_bias=use_bias,name='conv1'))
model.add(BatchNormalization(epsilon=0, momentum=momentum,
     axis=1, name='bn1',mode=0, trainable=True))
model.add(Activation('ReLU', name='act1'))
model.add(Flatten())
# dense1
model.add(XnorDense(nb_classes, use_bias=use_bias, name='dense3'))

opt = Adam(lr=lr_start)
model.compile(loss='squared_hinge', optimizer=opt,
     metrics=['acc'])
model.summary()
```

6. Learning rate scheduler and model building: this part trains the network with the options described. At the end of the training, the accuracy is evaluated and trained network parameters can be saved.

```
lr_scheduler = LearningRateScheduler(lambda e: lr_start *
     lr_decay ** e)
history = model.fit(X_train, Y_train,
                    batch_size=batch_size, epochs=epochs,
                    verbose=1, validation_data=(X_test, Y_test),
                    callbacks=[lr_scheduler])
score = model.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

Training and binarization

During training, the propagating gradient is computed using derivatives, as already explained in subsection 1.1.4. The activation functions of the weights/inputs are sign functions: the derivative results equal to 0 almost everywhere. In binary neural networks, an alternative binarization process is used and the source code is reported here from [51]:

```python
def round_through(x):
    rounded = K.round(x)
    return x + K.stop_gradient(rounded - x)
def _hard_sigmoid(x):
    x = (0.5 * x) + 0.5
    return K.clip(x, 0, 1)
def binary_sigmoid(x):
    return round_through(_hard_sigmoid(x))
def binary_tanh(x):
    return 2 * round_through(_hard_sigmoid(x)) - 1
def binarize(W, H=1):
    Wb = H * binary_tanh(W / H)
    return Wb
```

The operations perfomed in this code are the following:

1. `round_through(x)`: takes as input x and rounds the value to the nearest integer. The return value has the function `K.stop_gradient` that indicates that the value is computed when the gradient propagation is stopped.

2. `hard_sigmoid(x)`: clipping obtained by the function $x = 0.5 \cdot x + 0.5$ and the values 0,1. The corresponding output is:

$$\text{hard\_sigmoid(x)} = \begin{cases} 0, \text{ when } x \leq -1 \\ 0.5 \cdot x + 0.5, \text{ when } -1 < x < +1 \\ 1, \text{ when } x \geq 1 \end{cases} \tag{3.14}$$

3. `binary_tanh(x)`: is obtained by applying the `round_through(x)` function to the `hard_sigmoid(x)` and rescaling the result between ± 1.

When an input/weight has to be binarized, the function `binarize(W,H=1)` is called and the final result is:

$$\begin{aligned}
&\text{binarize}(x) = \text{binary\_tanh}(x)\\
&\text{binary\_tanh}(x) = 2 \times \text{round\_through}(\text{hard\_sigmoid}(x)) - 1\\
&\text{round\_through}(\text{hard\_sigmoid}(x)) = \text{round}(\text{hard\_sigmoid}(x))
\end{aligned}$$

By flattening the equations:

$$\text{binarize}(x) = \begin{cases} 2 \times \text{round}(0) - 1, \text{ when } x \leq -1 \\ 2 \times \text{round}(0.5 \cdot x + 0.5) - 1, \text{ when } -1 < x < +1 \\ 2 \times \text{round}(1) - 1, \text{ when } x \geq 1 \end{cases} \tag{3.15}$$

Rounding in the (-1,1) interval can be split into two parts, since if the value of x is less-equal than 0.5 the result is rounded to 0, otherwise to 1. This procedure provides a piece-wise approximation of the sign function and of the estimators used in the training process. As it is possible to discover, the value of 0 is approximated to -1 instead of +1. In VHDL this binarization method is not used, because it is useful only during the training process: in fact it is sufficient to take the $\overline{\text{sign}}$ of the incoming input (exception for "0", which sign is considered).

Output of the program

At the beginning of the program, Python reports the neural network's structure summary:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
pool1 (MaxPooling2D)         (None, 1, 14, 14)         0
_________________________________________________________________
conv1 (XnorConv2D)           (None, 6, 13, 13)         24
_________________________________________________________________
bn1 (BatchNormalization)     (None, 6, 13, 13)         24
_________________________________________________________________
act1 (Activation)            (None, 6, 13, 13)         0
```

```
_________________________________________________________________
flatten_1 (Flatten)          (None, 1014)              0
_________________________________________________________________
dense3 (XnorDense)           (None, 10)                10140
=================================================================
Total params: 10,188
Trainable params: 10,176
Non-trainable params: 12
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
```

During training, the accuracy is continuously evaluated in order to see which is the trend of the network to achieve good recognition rate (train accuracy). At the end of an epoch, the network is tested and the corresponding result is called test accuracy. The accuracy's behavior of this simple neural network is the following:

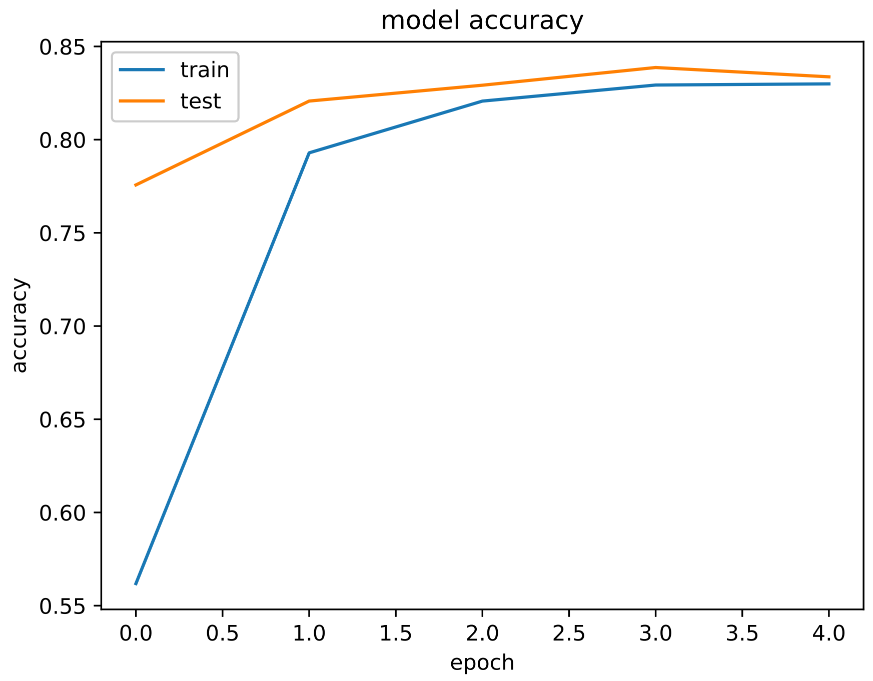


Figure 3.4: Accuracies' trend over 5 epochs and batch size of 10

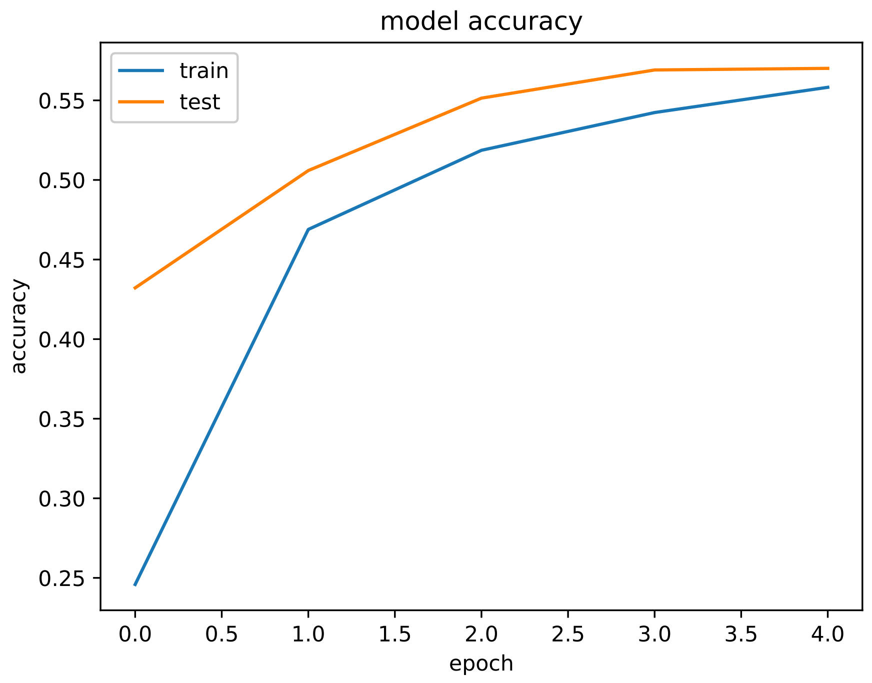


Figure 3.5: Accuracies' trend over 5 epochs and batch size of 100

In general, increasing the number of epochs allows to achieve better accuracy results, until the network reaches a saturation: this is determined by the complexity of the network itself and by the total number of learnable parameters available. Another important parameter is the batch size: the smaller it is, the higher is the accuracy achievable within an epoch as reported in Figure 3.5. In the following plot, it is shown how does the number of epochs influences the accuracy:

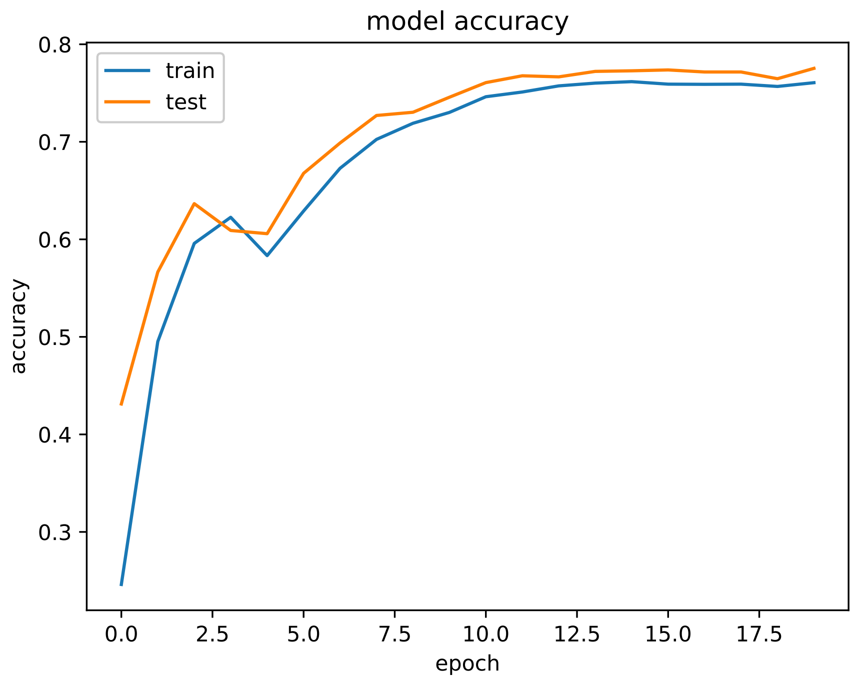
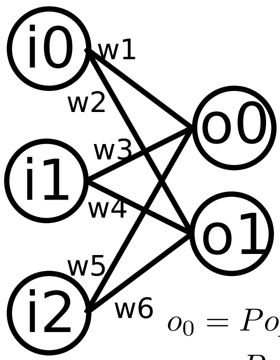


Figure 3.6: Accuracies' trend over 20 epochs and batch size of 100

As it is possible to see, the train accuracy behaves like a logarithmic function, so a saturation is always reached.

Approximations

Once the network is trained, some approximations have to be considered, in order to simplify the VHDL implementation. One of the most computational intensive part in the neural network is the **fully connected layer**, in fact the computing resources of α and $\mathbf{K}$ have to consider a very huge number of elements. $\mathbf{K}$ computation in the fully connected layer becomes the mean of the absolute values of the inputs, so considering the neural network model in Figure 3.1, it is a calculation over 1014 values, while α is the mean of the absolute value of each set of weights considered separately, as shown in the following figure:



$$\alpha_1 = \frac{|w_1|+|w_3|+|w_5|}{3}$$

$$\alpha_2 = \frac{|w_2|+|w_4|+|w_6|}{3}$$

$$K = \frac{\sum_{i=0}^{2} |i_i|}{3}$$

$$o_0 = Pop(\overline{i_0 \oplus w_1}, \overline{i_1 \oplus w_3}, \overline{i_2 \oplus w_5}) \times \alpha_1 \times K$$

$$o_1 = Pop(\overline{i_0 \oplus w_2}, \overline{i_1 \oplus w_4}, \overline{i_2 \oplus w_6}) \times \alpha_2 \times K$$

Figure 3.7: Example of **K** and α_i computation in the fully connected layer

To avoid this computational bottleneck, it is possible to neglect the computation of **K** and α for the fully connected layer and to approximate the output of a neuron as:

$$\mathbf{I} * \mathbf{W} \approx (sign(\mathbf{I}) \circledast sign(\mathbf{W})) \cdot \mathbf{K}\alpha \approx (sign(\mathbf{I}) \circledast sign(\mathbf{W})) \qquad (3.16)$$

In order to evaluate the impact of this approximation, the neural network has been trained with the original configuration and then in the computational model these two parameters have been removed:

```
class XnorDense(BinaryDense):
    def call(self, inputs, mask=None):
        inputs_a, inputs_b = xnorize(inputs, 1., axis=1,
             keepdims=True) # (nb_sample, 1)
        kernel_a, kernel_b = xnorize(self.kernel, self.H, axis=0,
             keepdims=True) # (1, units)
        print(K.get_value(kernel_a))
        output = K.dot(inputs_b, kernel_b) *
            kernel_a * inputs_a #<---- Original
        # output = K.dot(inputs_b, kernel_b) <--- Approximated
```

To properly evaluate the effect of this approximation, an MLP network has been built with a several number of fully connected layers:

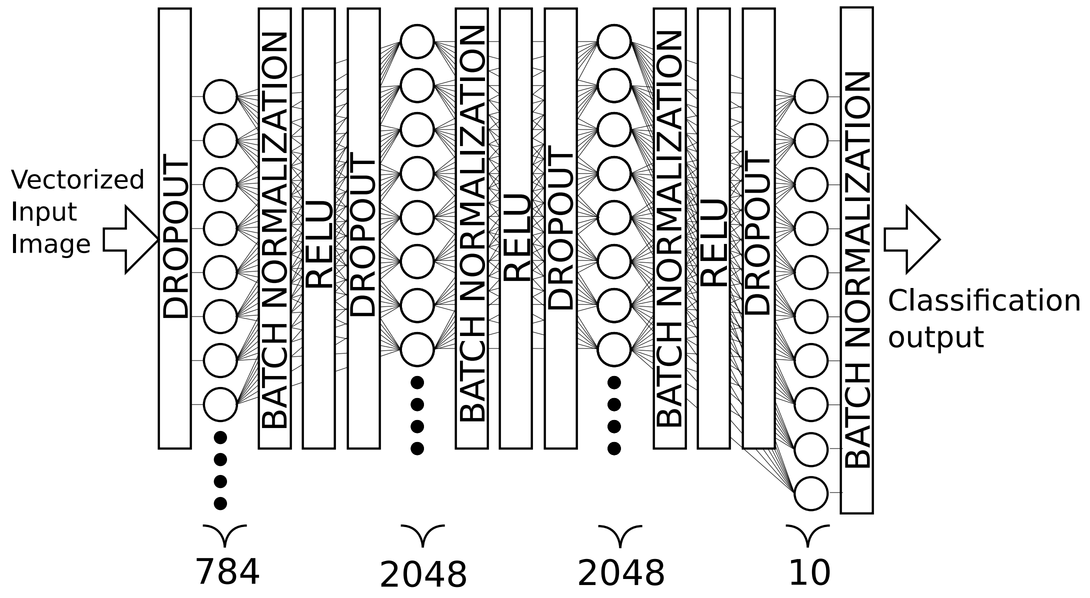


Figure 3.8: MLP network used to test the approximation drawback. This structure is able to achieve an accuracy of around 97% after 20 training epochs

Python's output reports the total number of trainable parameters and the network's accuracy:

```
Total params: 10,039,336
Test accuracy: 0.9181
```

After training with 1 epoch, the network achieves an accuracy of 0.9181. Trying to neglect $\mathbf{K}$ and α in the fully connected computation, the final accuracy is:

```
Test accuracy: 0.9155
```

This result is very good because by canceling two very computational-intensive parts in the neural network does not heavily influence the total accuracy: this is because the activations coming from the fully connected layer already contains the classification result without the usage of scaling factors, but batch normalization layers are required to maintain good accuracy. It is also possible to evaluate the impact of this approximation on the neural network used as reference (Figure 3.1):

```
Original accuracy = 0.8332
Approximated = 0.8338
```

Chapter 4

Hardware implementations

In this chapter, two different VHDL implementations of the neural network depicted in Figure 4.1 are reported, in particular an OOM and In-Memory structure.

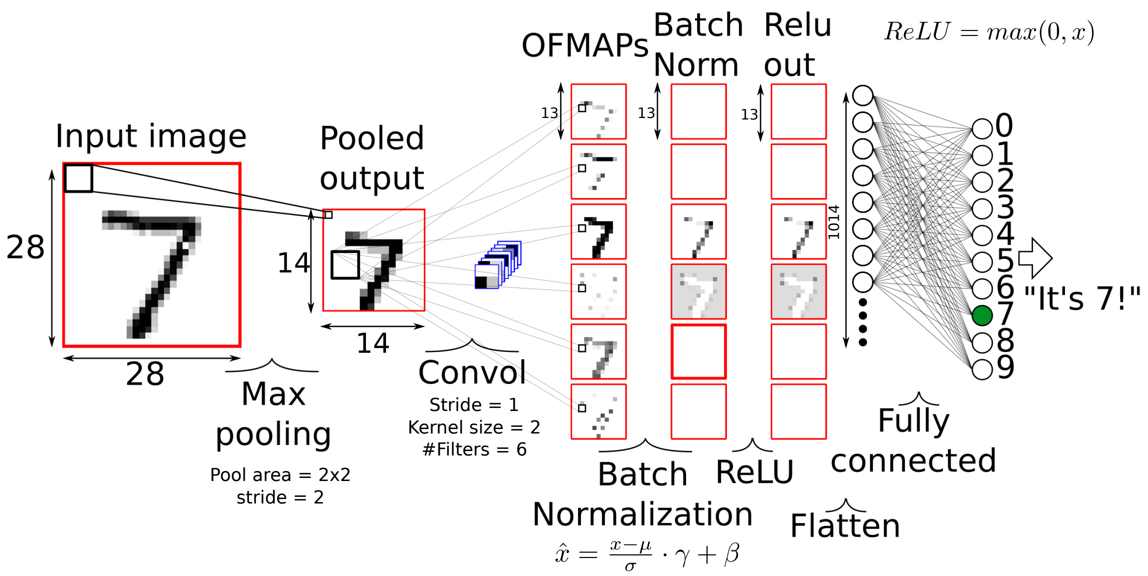


Figure 4.1: Neural network model used as starting point.

These implementations are discussed and compared in terms of performance. The steps used in hardware design flow are the following:

1. VHDL fixed-point implementation is realized and simulated with Modelsim;

2. Synthesis and network analysis in terms of area, timing and power are performed using Synopsys Design Compiler. Power estimation has been performed considering the worst case: all the switching activities are equal to 1;

3. Place and Route phase with Cadence Innovus;

4. Post Place & Route power estimation using `.vcd` file.

The possibility to implement any kind of neural network model is discussed after all the hardware design explanations. The model depicted in Figure 4.1 is used as a starting point, since it is a very simple and straight-forward example.

4.1 OOM implementation

As it is possible to see in Figure 4.1, the neural network is composed by several layers, that have different tasks. Each of them are now discussed.

4.1.1 Max pooling layer

Max pooling compute the maximum value of an input's subset and provide it to the output. The important parameters that max pooling layer uses are w_{in}, w_{filter} and stride that determines the size of the input, the overlapped window and the corresponding step size.

Input selection

Considering that the input's form is a matrix, it is possible to associate to each cell an index, representing the address of the considered input value:

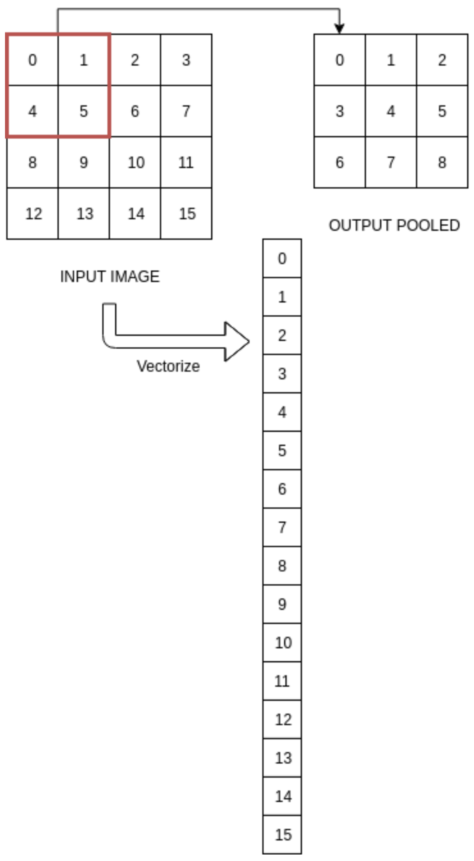


Figure 4.2: Max pooling: indexing example with $w_{in} = 4$, $w_{filter} = 2$ and stride $= 1$

In the example proposed in Figure 4.2, the first output cell (index 0) is obtained by computing the maximum value of the highlighted input cells with indexes 0,1,4,5. The second with the maximum value of 1,2,5,6 and so on. In VHDL it is possible to transform the input image matrix into a vector and to store it into a register file as shown in Figure 4.2. In general a w_{filter}^2 number of inputs are fetched and the addressing is performed considering the following pseudocode:

```
current_address[N] = initial_address;
counter = 0;
if (clk'event and clk= '1')
    counter ++;
    if(counter < w_out**2)
        for i=1:N
            current_address[i] = current_address[i] + stride;
        end
    else
        for i=1:N
            current_address[i] = current_address[i] + w_in*stride-(w_in-w_filter);
        end
        counter = 0;
    end
end
```

In the pseudocode, the current address is computed considering the stride, w_{in} and w_{filter} as follows:

1. At the beginning, `current_address[N]` is set to `initial_address`, which are the addresses of the first pooling window (in the example 0,1,4,5);

2. For each positive clock event, a `counter` is increased;

3. If the value of `counter` is less than w_{out}^2 (output pooling dimensions), the value added to each `current_address[i]` is the value of stride. In the example, if `counter < 3` then `current_address[i] = current_address[i] + 1`;

4. If `counter` has reached the value of w_{out}^2, it means that the pooling window has reached the end of the input columns and it has to be shifted also by rows: in the example, if `current_address = {2,3,6,7}` the following addresses should be `current_address = {4,5,8,9}`. To do this, the `current_address` has to be added by 2 instead of 1 and the algorithm becomes

   ```
   current_address[i] = current_address[i]+w_in*stride-(w_in-w_filter)
   ```

   This formula has been found experimentally, in fact by multiplying w_{in} and stride and adding to `current_address`, the address value obtained moves the pooling window by a number of rows equal to stride. Finally, the pooling region has to be shifted by columns by subtracting $(w_{in} - w_{filter})$.

5. `counter` is reset to 0.

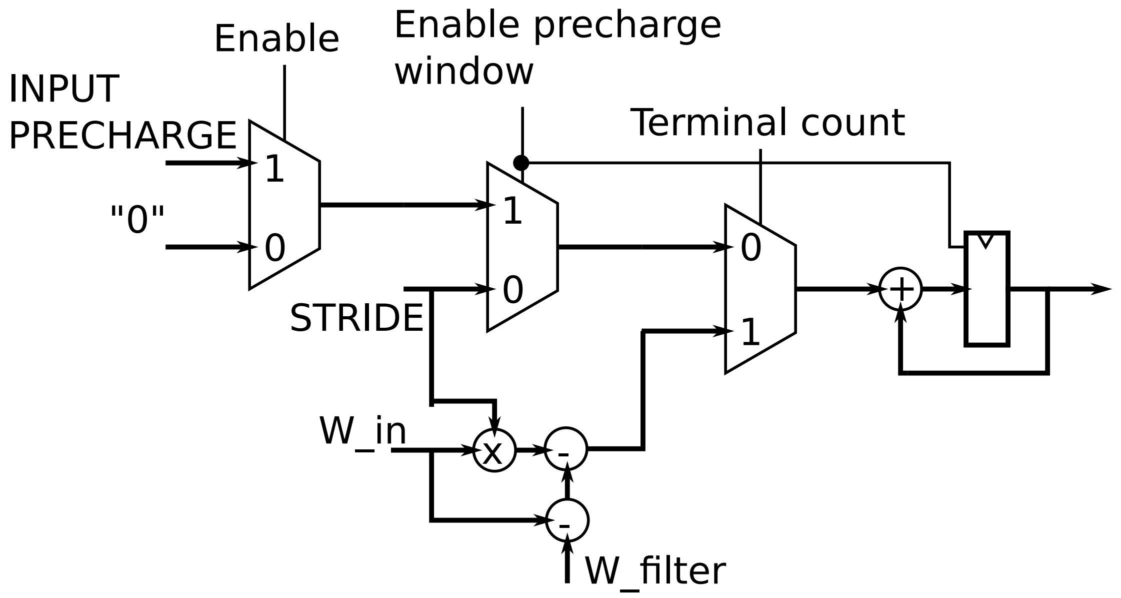


Figure 4.3: Input selection circuit

The circuit shown in Figure 4.3 works as follows: when enabled, the first multiplexer (left) propagates the precharging values toward the output, if enable precharge is '1' and terminal count '0'. Input preset value is stored into the final register and the adder adds every clock cycle the input value with the stored one. If terminal count is '1', it means that `current_address` has to be added by the new value instead of stride. This circuit is replicated for each element and so there are w^2_{filter} input selection circuits in parallel that are implemented in an external unit, which is not synthesized.

Max comparator

Once the w^2_{filter} number of inputs are selected, they are fed to the max comparator. It takes one input per clock cycle and compares it with a previously stored result: if it is higher than the stored one, the new value is saved and replaces the older one. The pseudocode is the following:

```
previous_value = -2^(n_bit); --minimum
counter = 0;
if(clk'event and clk='1')
```

```
    if(input(counter) > previous_value)
        previous_value = input(counter);
    end
    counter++;
end
```

1. At the beginning, the `previous_value` is set to the minimum value achievable, which is equal to `-2^n_bit`;
2. At each clock cycle, an input is selected and compared with the `previous_value`. If the input is higher, it will be stored.

Since the comparator works sequentially, the time required to compute a single comparation is equal to $w_{filter}^2 \times t_{ck}$.

Control unit

The FSM that controls the max pooling layer has the following structure:

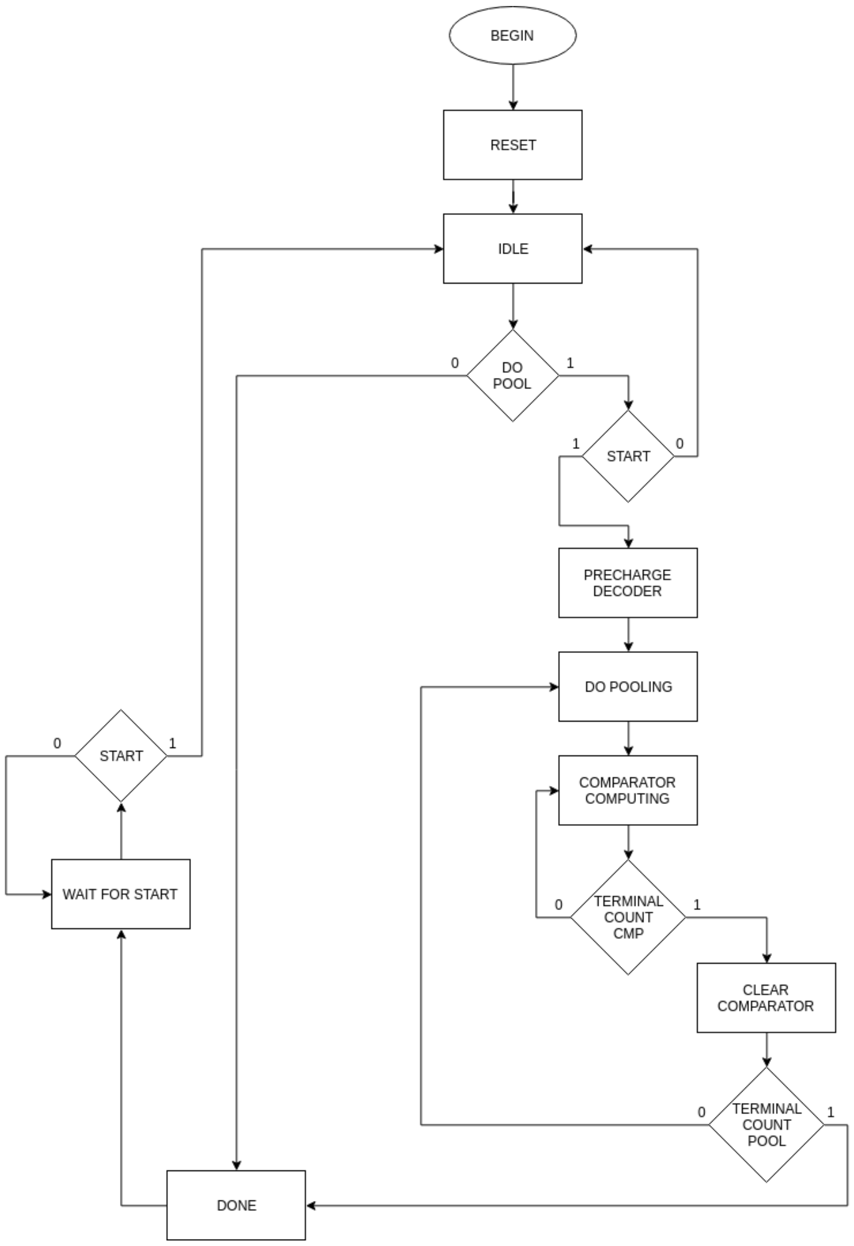


Figure 4.4: Max pooling layer FSM

As it is possible to see, at the beginning of the algorithm, it is checked the variable `do_pool`: if it is equal to 0, the pooling part is skipped by asserting the done signal otherwise the computations can start.

- Precharge decoder: the input selection precharges the `initial_address` registers (Figure 4.3), in order to select the inputs (enable and enable precharge window are set to '1');
- Do pooling: waiting state, that allows to store the values of `initial_address` into the corresponding registers and to start the pooling process;
- Comparator computing: the comparator starts to compute the maximum value and in the meanwhile a counter starts. The inputs are kept until the comparator has not finished, which is signaled by `terminal count cmp`;
- Clear comparator: once the comparator has finished, the result will be stored and the comparator's register will be cleared. At the same time, the input selection will be enabled and `initial_address` will be incremented, pointing to the new set of data required for the incoming computation;
- Done: once the pooling process has finished (signaled by a counter with `terminal count pool`), the FSM passes into a done state, where `done` signal is asserted and then into `wait for start`, in which the system is waiting for a new start. The counter that asserts the signal `terminal count pool` counts until w_{out}^2, in order to process all the outputs of the pooling layer.

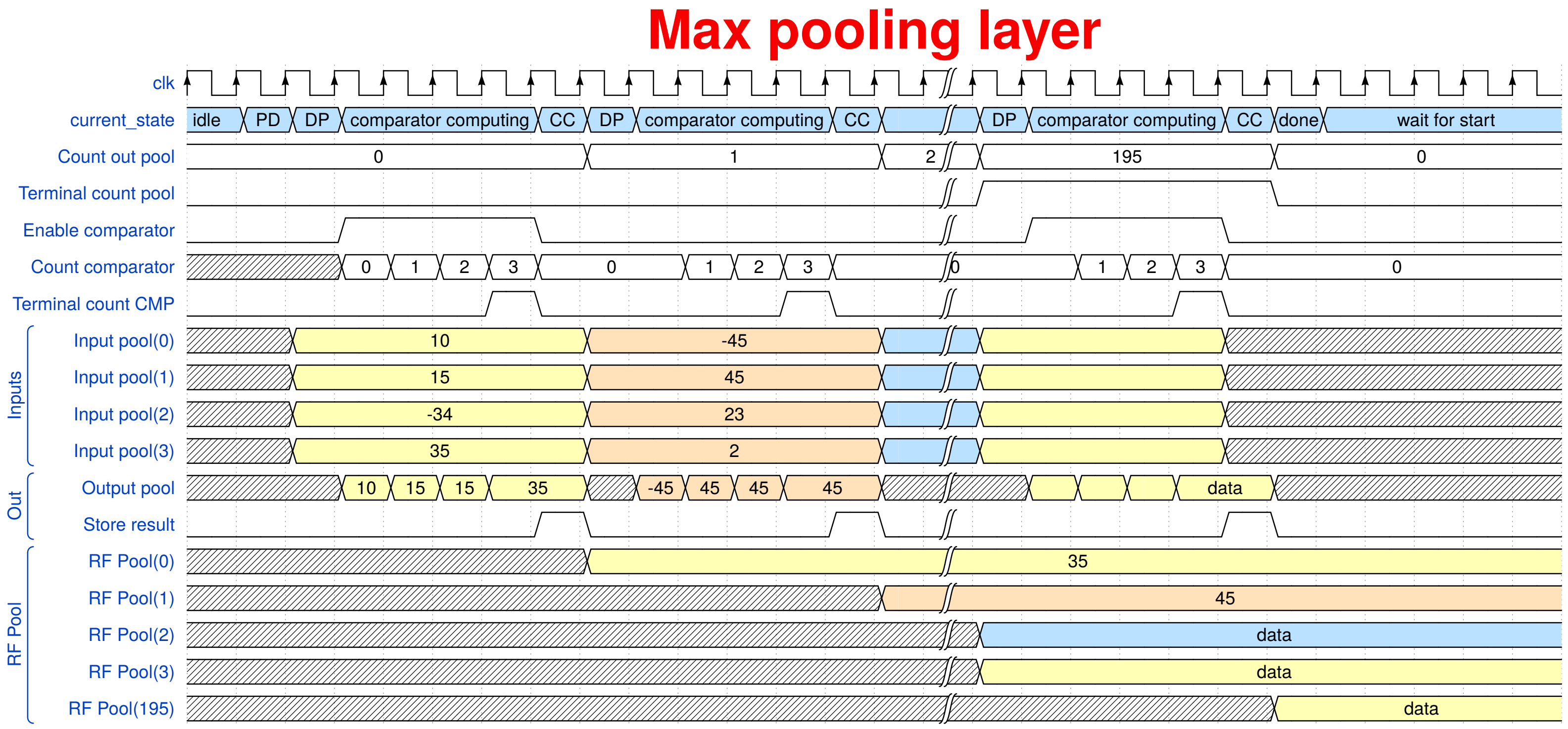


Figure 4.5: Timing diagram of the max pooling layer. Starting from idle, the FSM moves to **precharge decoder** (PD), in which the external decoder is precharged with its initial values. During **do pooling**, the inputs are provided to the max pooling layer and the computation starts with **comparator computing**, in which **Count comparator** is increased until it reaches $w^2_{filter(pool)} - 1$ value, that in the neural network model depicted in Figure 4.1 is equal to 3 (4-1). When the **terminal count CMP** is asserted, the FSM migrates to **clear comparator** (CC), in which the stored value inside the comparator is reset to the minimum. The result is stored inside the **RF Pool**, which is placed outside the chip (see Figure 4.27) and it is addressed by **count out pool**. The entire procedure is repeated until **count out pool** has not reached the **terminal count pool**, which is asserted when **count out pool** is equal to $w^2_{out(pool)}$, that in the neural network model in Figure 4.1 is 196. At this point, **done** and **wait for start** are reached, where FSM waits for a new start signal.

Scheduling

Since the max-pooling layers are the same for both OOM and In-Memory implementations, the scheduling is analyzed only once. By looking at the control unit depicted in Figure 4.4, the duration of the states are the following:

Table 4.1: Required clock cycles computation for max pooling layer.

State	Required clock cycles	Multiplicity
idle	1	1
weights_precharge	1	1
precharge_decoder	1	1
do_pooling	1	$w^2_{out(pool)}$
comparator_computing	w^2_{filter}	$w^2_{out(pool)}$
clear_comparator	1	$w^2_{out(pool)}$
done	1	1

Max pooling fetches data from the `RF INPUT Image` and perform the pooling operation. The total number of data fetched depends on stride, $w_{in(pooling)}$, $w_{filter(pooling)}$ which defines the size of the output data, which is equal to $w^2_{out(pooling)}$. For each set of data, there is the max comparation which requires $w^2_{filter(pooling)}$ to complete the computation. In the case of the neural network depicted in Figure 4.1, the total number of clock cycles required are:

$$\begin{aligned} Pool\_time &= (1+1+1+w^2_{out(pool)} \times (1+w^2_{filter}+1)+1) \times t_{ck} \\ &= (3+196 \times (1+4+1)+1) \times t_{ck} = 1180 \times t_{ck} \end{aligned} \tag{4.1}$$

4.1.2 Convolutional and fully-connected layers

The main parts composing a convolutional layer are alpha computation, **K** computation and XNOR Unit. Usually after a convolution are placed batch normalization and ReLU blocks, and so they are integrated inside the convolutional layer entity. Regarding the fully connected layer, it is quite similar to the convolution, since it requires the same procedures (except for α and **K**). Now they will be discussed the main parts composing the convolutional/fully-connected layer (from now called convolutional layer) and how the fully connected part can be integrated in the same architecture.

XNOR Unit

In order to realize the convolution operation, the signs of weights/inputs are XNORed and then XNOR results are pop-counted. The inputs that has to be XNORed with weights are the ones selected by the kernel window, as already discussed in the section 4.1.1. By selecting properly the inputs, their sign will be stored into a register file and the corresponding output is fed to the XNOR circuitry.

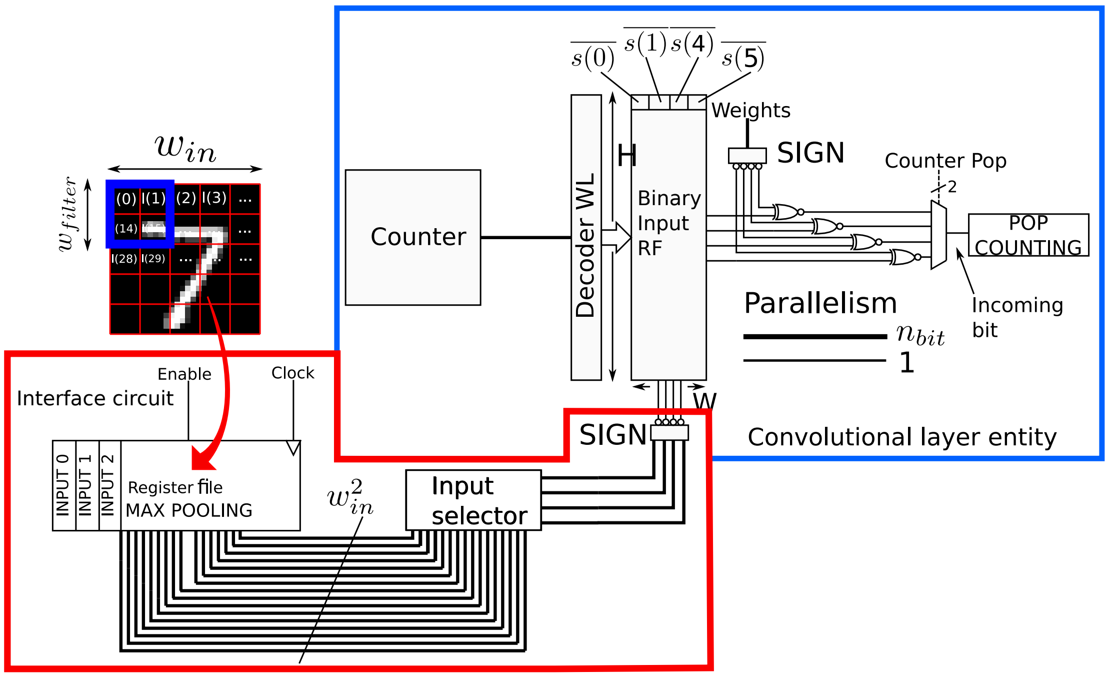


Figure 4.6: Example of a $w_{in} = 4$,$w_{filter} = 2$,$stride = 1$ input selection and saving circuits. The inputs are selected from the input selector and their sign is stored into the register file ($\overline{s}(0)$,$\overline{s}(1)$,$\overline{s}(4)$,$\overline{s}(5)$). Then, once the saving procedure is completed, the inputs are fetched from the **Binary Input RF** and XNORed with weights' signs. The XNOR results are selected from a multiplexer (Incoming bit)

The incoming bit (on the right of Figure 4.6) is fed to the pop-counting circuit, which is able to transform 1,0 into +1,-1 respectively. Once the multiplexer has completed the scanning, the final pop-result is obtained. Pop-counting has the following definition:

$$\text{Pop-Count} = \#1s - \#0s \tag{4.2}$$

A very simple circuit is used to perform the pop-counting:

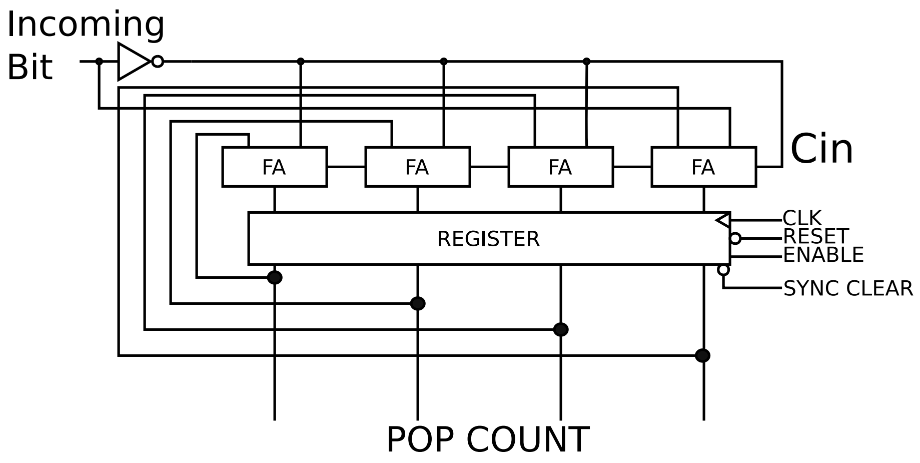


Figure 4.7: Pop-counting circuit: 4 bits example

If the incoming bit is equal to '1', Cin of the first FA is '0' while the FAs' inputs becomes "0001" which is added to the stored value into the register. Otherwise, if the incoming bit is '0', Cin is '1' and the inputs becomes "1110". Considering Cin the final number is "1111", which is added to the stored result. The times required by the entire processes of saving inputs-popcounting are given by:

$$t_{\text{save}} = w_{out}^2 \times t_{ck} \tag{4.3}$$

$$t_{\text{pop}} = w_{out}^2 \times w_{filter}^2 \times t_{ck} \tag{4.4}$$

Multiple input channels When a convolution has to be perfomed on multiple input channels, the entire architecture is parallelized in order to process all the matrices at the same time. The output equation becomes the following:

$$Conv = \sum_{i=1}^{c_{in}} \left[(sign(\mathbf{I}) \circledast sign(\mathbf{W}))_i \cdot \mathbf{K} \cdot \alpha\right] = \left[\sum_{i=1}^{c_{in}} (sign(\mathbf{I}) \circledast sign(\mathbf{W}))_i\right] \cdot \mathbf{K} \cdot \alpha \tag{4.5}$$

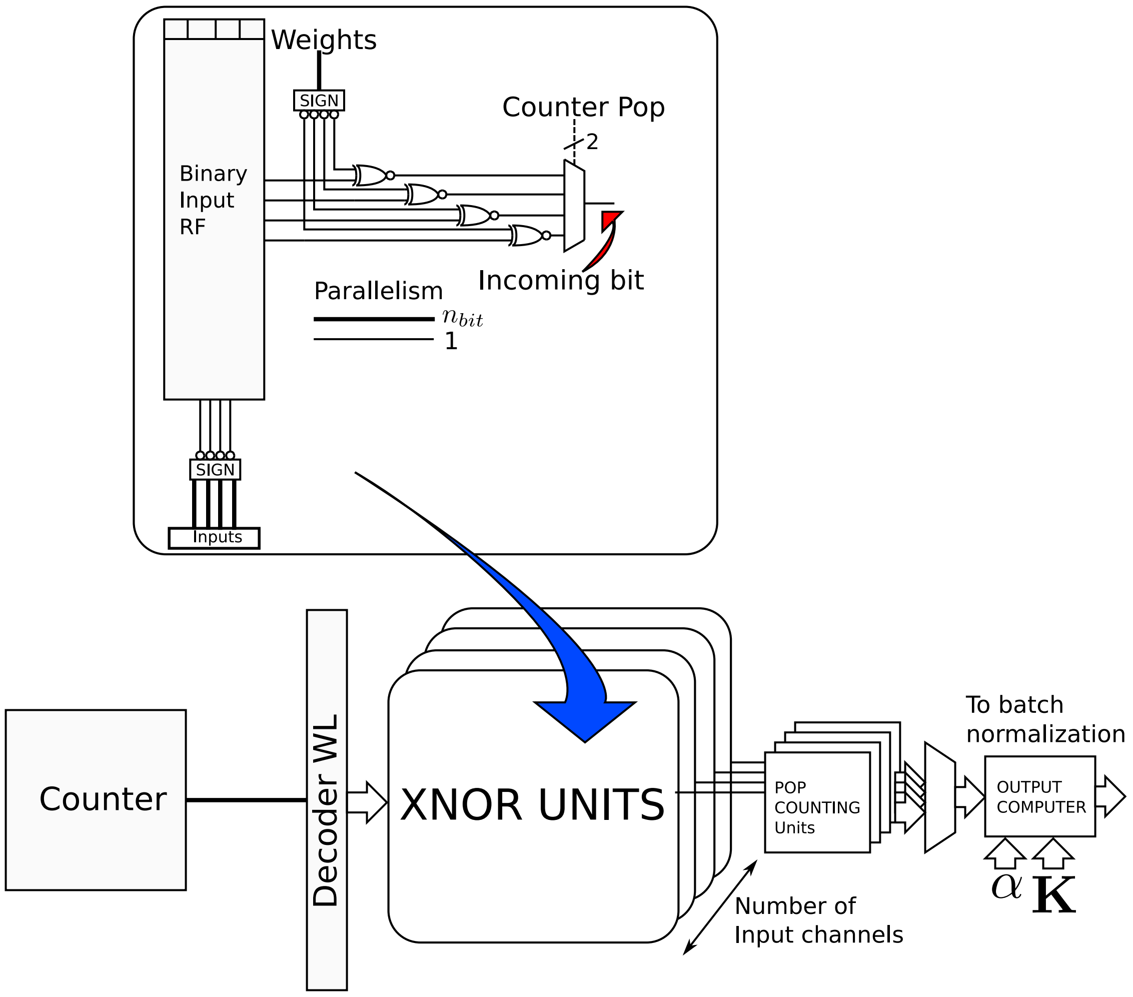


Figure 4.8: Multiple input channels architecture. The XNOR and pop-counting units are replicated for a number of input channels times, obtaining a parallel computation. Each channel contribution is added in the output computer unit.

For each channel there is a pop-counting unit. Each XNOR unit performs the computations in parallel and the final contribution is added serially in the output computer, that fetches one by one the OutPops.

Alpha computation

In order to compute the convolution, also α is required, which is the absolute mean of the considered kernel. Alpha computation starts during popcounting phase and in particular it requires w_{filter}^2 clock cycles. Alpha unit is very simple and it is composed by an adder, absolute block, register (which saves the partial results) and a divider.

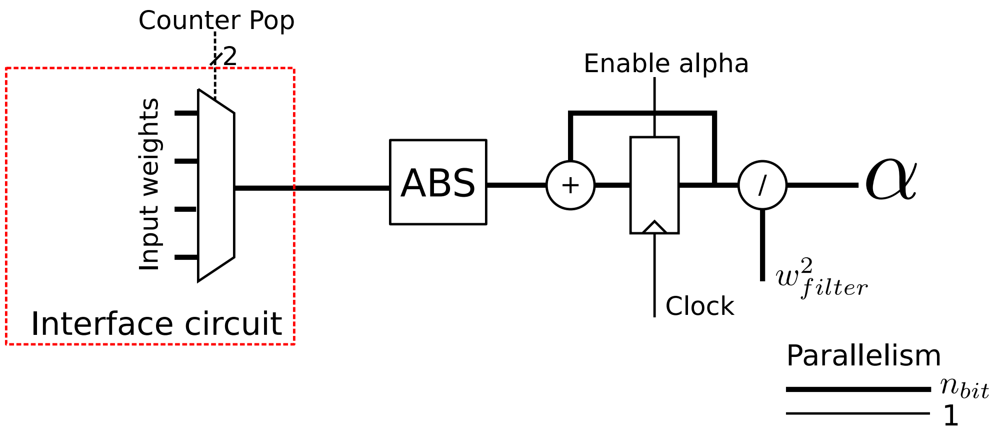


Figure 4.9: Alpha computational unit: example with $w_{filter} = 2$. The input multiplexer has been instatiated into an external unit, in order to reduce the total number of inputs of the chip.

As it is possible to see the multiplexer is piloted by the same counter used in Figure 4.6, since it has to choice one out of w_{filter}^2 possibilities. This multiplexer is instantiated outside the chip in order to reduce the total number of inputs of the chip, in fact considering the following example with $w_{filter} = 5$ and $n_{bit} = 18$, the total number of bits are equal to $w_{filter}^2 \times n_{bit} = 450$. By putting the multiplexer outside the chip guarantees only n_{bit} bits in input. These considerations are applied also in the following parts. When alpha computation is not required anymore, it is disabled by using `Enable alpha`.

Multiple input channels When multiple input channels are considered, before computing α, the architecture has to consider all the kernels and to perform the absolute sum of each kernel element as follows:

```
for i=1:c_in
    abs_weights(:,:) = abs_weights(:,:) + abs(kernel(:,:,i));
end
alpha = mean(abs_weights(:));
```

In formulas:

$$\alpha = \frac{\sum_{i=1}^{w_{filter}^2 \times c_{in}} |W_i|}{w_{filter}^2 \times c_{in}} \tag{4.6}$$

To compute it, the alpha circuit has been modified as follows:

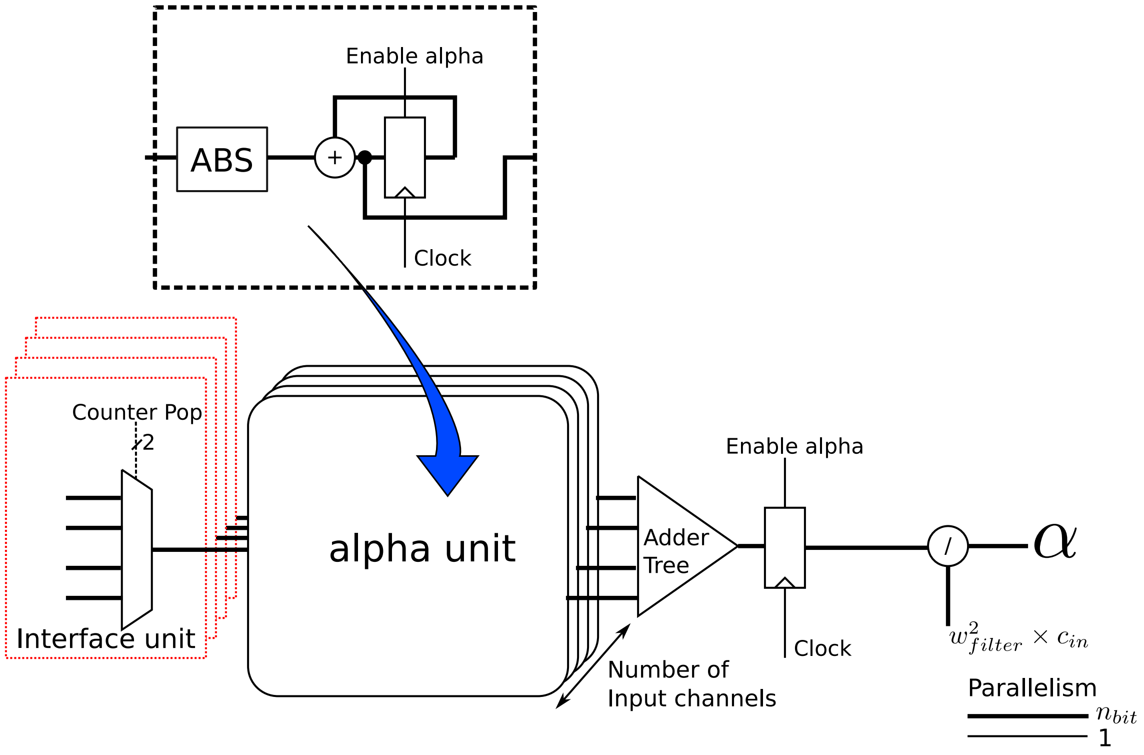


Figure 4.10: Alpha computation unit in case of multiple input channels. An adder tree adds all the multiplexed weights from the c_{in} inputs. The last division is perfomed also by the number of input channels. Re-timing technique has been used for the loop register, in order to reduce the critical path caused by an adder tree and a divider.

Multiple output channels When multiple output channels are considered (as in the Figure 4.1), a multiplexer is placed in alpha unit input for each input channel, which select the weight-set to consider, obtaining the final architecture which is depicted in Figure 4.11:

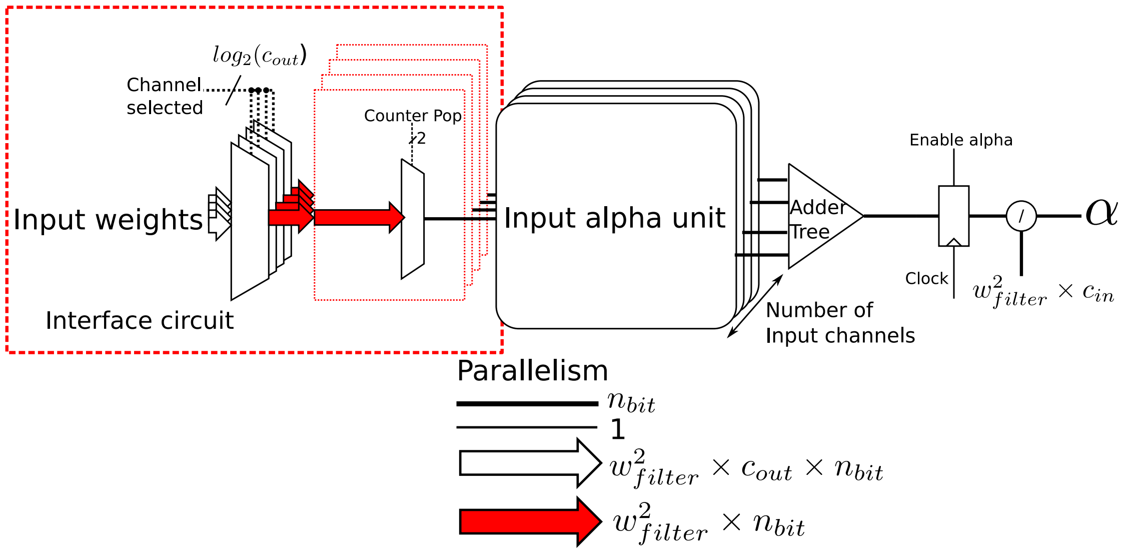


Figure 4.11: Alpha computation unit in case of multiple output/input channels

Depending on the output channel considered (addressed by `Channel selected`), the multiplexer selects a weight-set for each input channel and the computational scheduling of alpha unit is then executed. All the inputs contributions are then added in the adder tree and divided by $w_{filter}^2 \times c_{in}$. Placing the multiplexers outside the chip is fundamental for very large networks, since this approach reduces the maximum number of input bits from $w_{filter}^2 \times c_{out} \times c_{in} \times n_{bit}$ to $c_{in} \times n_{bit}$. Considering for example the first layer of AlexNet and imposing $n_{bit} = 18$, the total number of input bits can be computed as:

$$w_{filter(AlexNet)} = 11$$
$$c_{in} = 3$$
$$c_{out} = 96$$
$$n_{bit} = 18$$
$$\text{Number of bits(Mux inside)} = w_{filter}^2 \times c_{out} \times c_{in} \times n_{bit} = 34848$$
$$\text{Number of bits(Mux outside)} = c_{in} \times n_{bit} = 54$$

Division process About the division, some details have to be discussed. By defining `n_bit` and `n_bit_fractional` as two parameters indicating how many bits are used to represent an input/weight, the resulting fixed point value is split as follows:

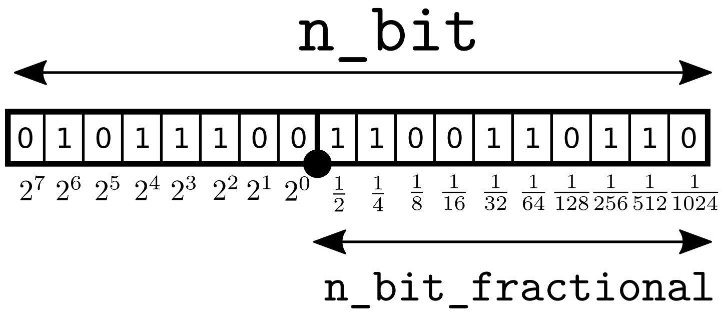

n_bit
0 1 0 1 1 1 0 0 1 1 0 0 1 1 0 1 1 0
2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 $\frac{1}{2}$ $\frac{1}{4}$ $\frac{1}{8}$ $\frac{1}{16}$ $\frac{1}{32}$ $\frac{1}{64}$ $\frac{1}{128}$ $\frac{1}{256}$ $\frac{1}{512}$ $\frac{1}{1024}$
n_bit_fractional

Figure 4.12: Fixed point representation: example with n_bit = 18 and n_bit_fractional = 10

To compute $\frac{1}{w_{filter}^2}$, the division process has to consider the following steps:

1. Division between $2^{n_{bit}-1}$ and the term to divide. The result is on `n_bit`. Considering the following example with $w_{filter}^2 = 4$:

$$Div = \frac{2^{n_{bit}-1}}{w_{filter}^2} = \frac{2^{17}}{4} = 32768 \tag{4.7}$$

Representing this result on 18 bits:

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

2. The fixed point division result is obtained by taking from the 16th bit to the 10th bit of the previous calculation and placing it in the 6th position toward 0. In general, the operation perfomed is the following:

```
to_divide(n_bit-1 downto n_bit_fractional) <= (others => '0') ;
to_divide(n_bit_fractional-1 downto 0) <= div(n_bit-2 downto
                                          n_bit-2-(n_bit_fractional-1));
```

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

K computation

The matrix **K** has to be computed during the `Binary input RF` precharging, since, once it is precharged, the fixed point input values are not considered anymore to avoid wasting of power caused by data migration. The design is based on the following considerations:

1. The precharging phase has a duration equal to w_{out}^2, since all the w_{filter}^2 inputs are precharged at the same time in `Binary input RF`;
2. During this period of time, **K** unit has to compute all the **K** values and to store them into a register file (called `k_array`);
3. **K** values must be ready before the convolution computation.

In order to do this, the data precharging is stopped everytime a new input-set is ready, to allow the **K** computation unit to complete the computation. A valid output from **K** computation unit is achieved after w_{filter}^2 clock cycles. The corresponding scheduling obtained is depicted in Figure 4.13:

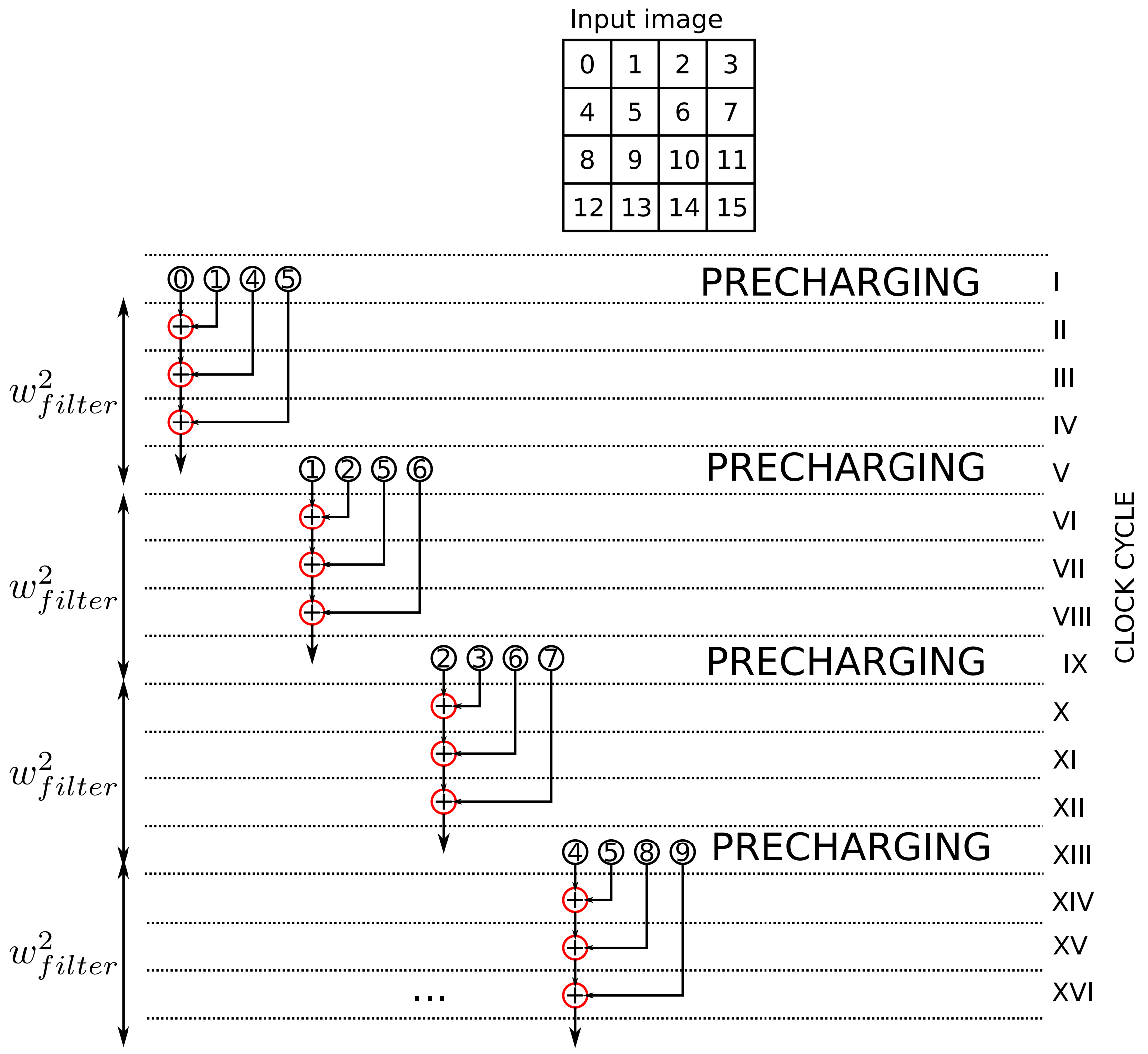


Figure 4.13: **K** scheduling. Example with $w_{in} = 4$ and $w_{filter} = 2$. Everytime a new data is precharged, **K** computation starts and lasts for w^2_{filter} clock cycles.

In general, the total number of inputs to be precharged inside the `Binary Input RF` is equal to w^2_{out}. As a consequence, the clock cycles required to achieve the end of computation is equal to:

$$\text{Max clock cycles} = w^2_{out} \times (w^2_{filter} + 1) \times t_{ck} \tag{4.8}$$

After the completion of the **K** computation, the data are ready to be processed, so pop-counting part can start.

Multiple input channels By looking at the Equation 3.8, the absolute sum of the inputs has to be considered. In order to reduce the complexity of the Equation 3.8, the following transformation is applied:

$$\begin{aligned}\mathbf{K} = \frac{\sum_{c=1}^{c_{in}} |\text{Inputs}(:,:,\text{c})|}{c_{in}} * \begin{bmatrix} \frac{1}{w_{filter}^2} & \frac{1}{w_{filter}^2} & \cdots \\ \frac{1}{w_{filter}^2} & \frac{1}{w_{filter}^2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} = \\ = \sum_{c=1}^{c_{in}} \left[\sum_{i=1}^{w_{filter}^2} \text{SelectedInputs}(c)(i) \right] \times \frac{1}{w_{filter}^2 \times c_{in}}\end{aligned} \tag{4.9}$$

Where SelectedInputs is the input-set selected in a clock cycle so, considering the example in Figure 4.13, they will be 0,1,4,5 $\rightarrow$ 1,2,5,6 etc. Each channel contribution is added by an adder tree and the corresponding output is divided by $w_{filter}^2 \times c_{in}$. Since each convolutional layer has a variable number of input channels, the output of the adder tree is chosen by a multiplexer piloted by the `conv_z` variable, which indicates how many input channels are used in that layer: if `conv_z` is equal to 1, it means that it is considered only 1 input layer so the output is simply given by the first `K scheduled unit`. When `conv_z` is equal to 2, two channels are selected, so also the contribution of the second parallel `K scheduled unit` has to be considered.

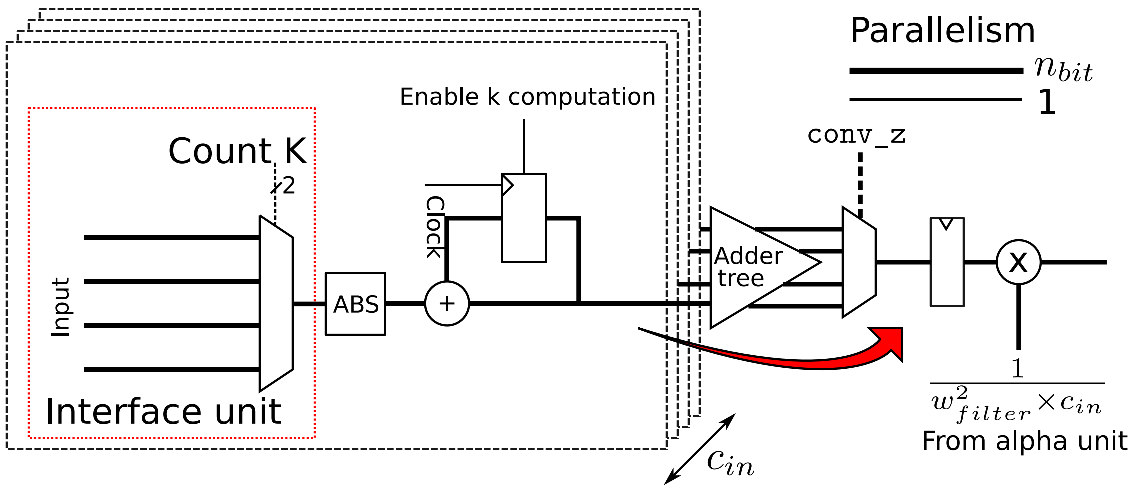


Figure 4.14: Example of **K** unit with $w_{filter} = 2$ with multiple input channels. The input multiplexer has been integrated into an external unit in order to reduce the number of contemporary inputs into the architecture. Since **conv_z** is fixed, the multiplexer selects only one input per time: the register indicated by the red arrow has been moved from its original location by applying re-timing: this technique avoids to have multiple adders connected to the final multiplier, reducing the critical path delay. The last term ($\frac{1}{w_{filter}^2 \times c_{in}}$) is taken directly from the alpha unit.

Fully connected integration

The fully connected layer has the same computational flow of the convolutional part, but α, **K** are not considered for the motivations explained in section 3.2.1. To compute the fully connected layer, the following example can be considered:

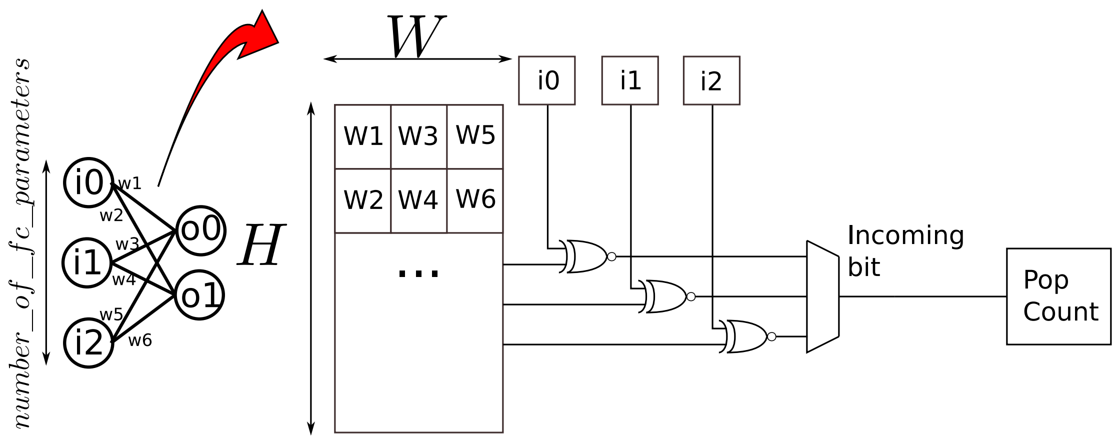


Figure 4.15: Example of fully connected layer integration. The data precharging pattern is inverted to compute the outputs values of the neurons o0 and o1. $number\_of\_fc\_parameters$ indicates the number of input neurons that in this example is equal to 3. In the real case depicted in Figure 4.1, $number\_of\_fc\_parameters = 1014$

In Figure 4.15, the fully connected layer has been implemented in the original convolutional structure. The data precharging pattern is inverted to compute the output values as follows:

$$\begin{aligned} o0 &= Pop(\overline{i_0 \oplus w_1},\overline{i_1 \oplus w_3},\overline{i_2 \oplus w_5}) \\ o1 &= Pop(\overline{i_0 \oplus w_2},\overline{i_1 \oplus w_4},\overline{i_2 \oplus w_6}) \end{aligned} \tag{4.10}$$

Since the fully connected layer in the model proposed in Figure 4.1 has 1014 binarized inputs and 10 outputs, the `Binary input RF` has to have at least 1014 columns and a number of rows which is equal to the maximum number of $w^2_{out}(i)$ of all the layers in the neural network that in Figure 4.1 is equal to:

$$\#Rows = max(w^2_{out}(i)) = 13 \times 13 = 169 \tag{4.11}$$

Since the wordlength is very huge, the chip has to have at least 1014 input bits for each `Binary input RF`. To eliminate this drawback, the fully connected process has been serialized to reduce the input data's bitlength, as depicted in Figure 4.16:

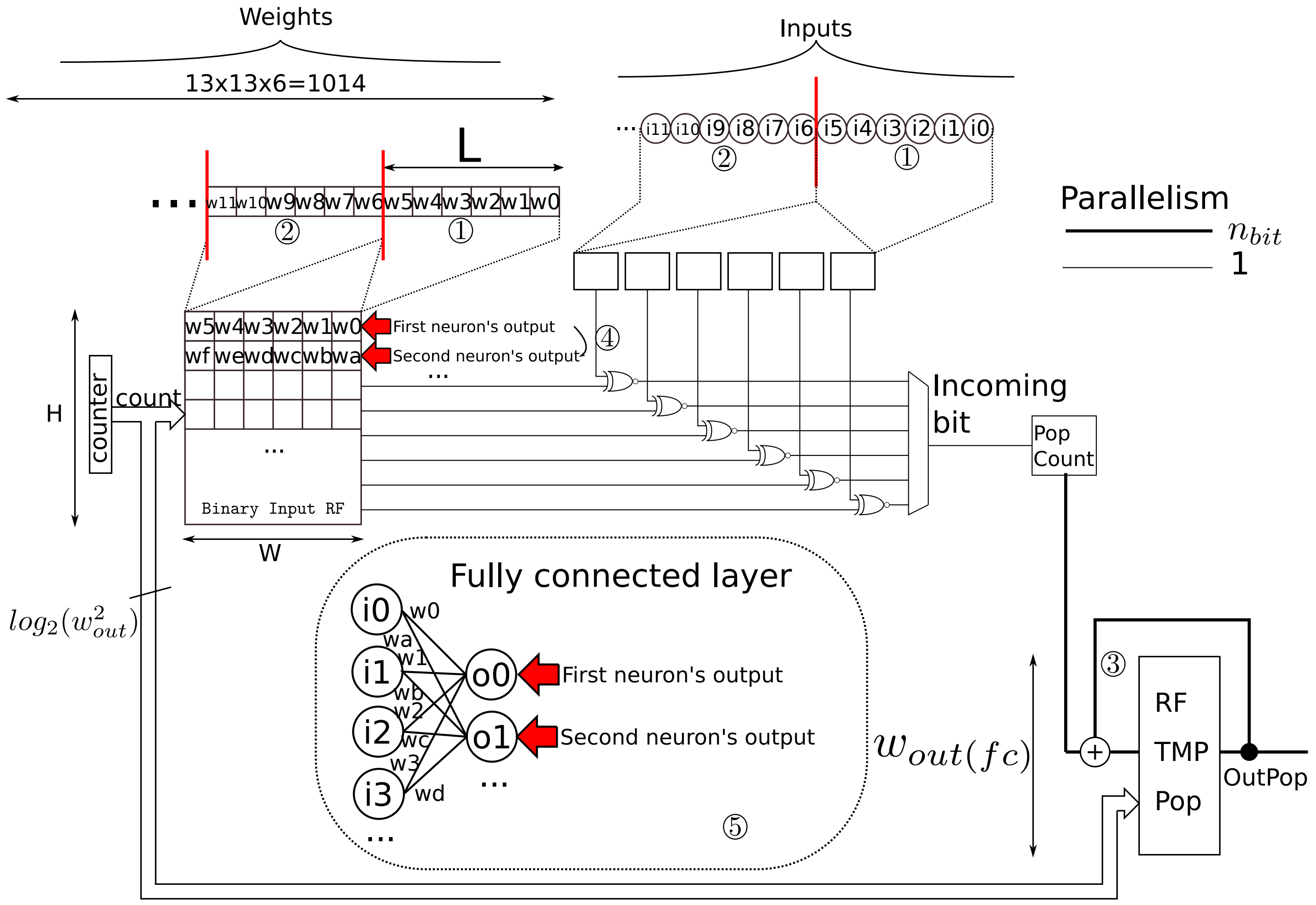


Figure 4.16: Fully connected layer scheduling. Inputs and weights are divided into subgroups of L elements and precharged inside the **Binary Input RF**. At each cycle, once the pop-counting has finished, a new set of inputs/weights is precharged in the **Binary Input RF** and the pop-counting part starts again. The register file **RF TMP Pop** holds the temporary values of pop-counting and it is addressed by the counter: the total number of registers used in **RF TMP Pop** is equal to the number of output neurons that, as in Figure 4.1, it is equal to 10.

The fully connected scheduling works as follows:

- ①: a sub-group of inputs/weights are precharged into the `Binary Input RF` and fed to xnors' inputs respectively. The width of the sub-groups (L) is defined according to the dimensions of the first fully connected layer ($number\_of\_fc\_parameters$), that in the model depicted Figure 4.1 is 1014: by choosing the minimum size that is also a divisor of $number\_of\_fc\_parameters$ brings advantages in terms of energy and power consumed. The constraint that has to be respected for the definition of W is:

$$\begin{cases} W \geq w_{filter}^2 \\ W \geq L \end{cases} \tag{4.12}$$

  For the neural network depicted in Figure 4.1, W=6. For each fully connected layer, the value of L is defined dinamically, according to the algorithm. As it is possible to discover in Figure 4.16, the fully connected layer model (⑤) is replicated in the `Binary Input RF`: the first row is dedicated to the first neuron's output, the second to the second neuron and so on. Each row is fetched and the XNORs between inputs/weights are computed: the XNORs' results are multiplexed and the Incoming bit is fed to the pop-counting unit;

- ③: the pop-count result is added with a previously stored value, which is addressed by the counter: if `count`=0, it means that the first line of `RF TMP POP` is addressed and the pop-counting result refers to the first output neuron. `RF TMP Pop` is useful in the OOM implementation, since once the first pop-count result has been computed, `count` increments selecting the second line and the temporary pop-count result is stored;

- ④: after the pop-counting procedure for the first neuron has finished, `count` increments selecting the second line. A new pop-count for the second neuron starts and finishes when all the inputs have been multiplexed. The temporary value is stored into the `RF TMP Pop`;

- The entire procedure is executed for each output neuron: it means that when `count`=9, the first fully connected layer cycle has been completed and the entire procedure starts again from the first output neuron. A new set of values (indicated by ②) are going to be precharged. Considering again the first output neuron, a new pop-counting procedure starts and the resulting value is added to the value stored in the `RF TMP Pop`.

This procedure ends when all the inputs/weights are considered for each output neuron, requiring a number of iterations that are equal to:

$$n_{iter} = \frac{number\_of\_fc\_parameters}{L} \tag{4.13}$$

Considering the neural network model depicted in Figure 4.1, since the number of input neurons are 1014 and L = 6, the Time duration can be computed as:

$$\begin{aligned}\text{Time duration} &= \frac{\text{number\_of\_fc\_parameters}}{L} \times (L \times w_{out(fc)}) \times t_{ck} = \\ &= \frac{1014}{6} \times (6 \times 10) \times t_{ck} = 10140 \times t_{ck}\end{aligned} \tag{4.14}$$

Output computation, Batch normalization and ReLU

Once α and $\mathbf{K}$ are computed (convolution case) and pop-counting routine has finished, the output is simply obtained by the product of these three values. Batch normalization takes the values of the convolution/fully connected layer and computes the batch-normalized output as already explained. ReLU layer is very simple, since it computes the maximum between 0 and its input. In hardware it is realized with a multiplexer that chooses between "0" and input by using the input's sign as select.

Multiple input channels In the convolutional case, in order to consider the contributions of the parallel architectures, an accumulator is used to add all the contributions. The formula for the output computation becomes:

$$Conv = \sum_{c=1}^{c_{in}} (sign(\mathbf{I}) \circledast sign(\mathbf{W}))_c \cdot \mathbf{K}\alpha \tag{4.15}$$

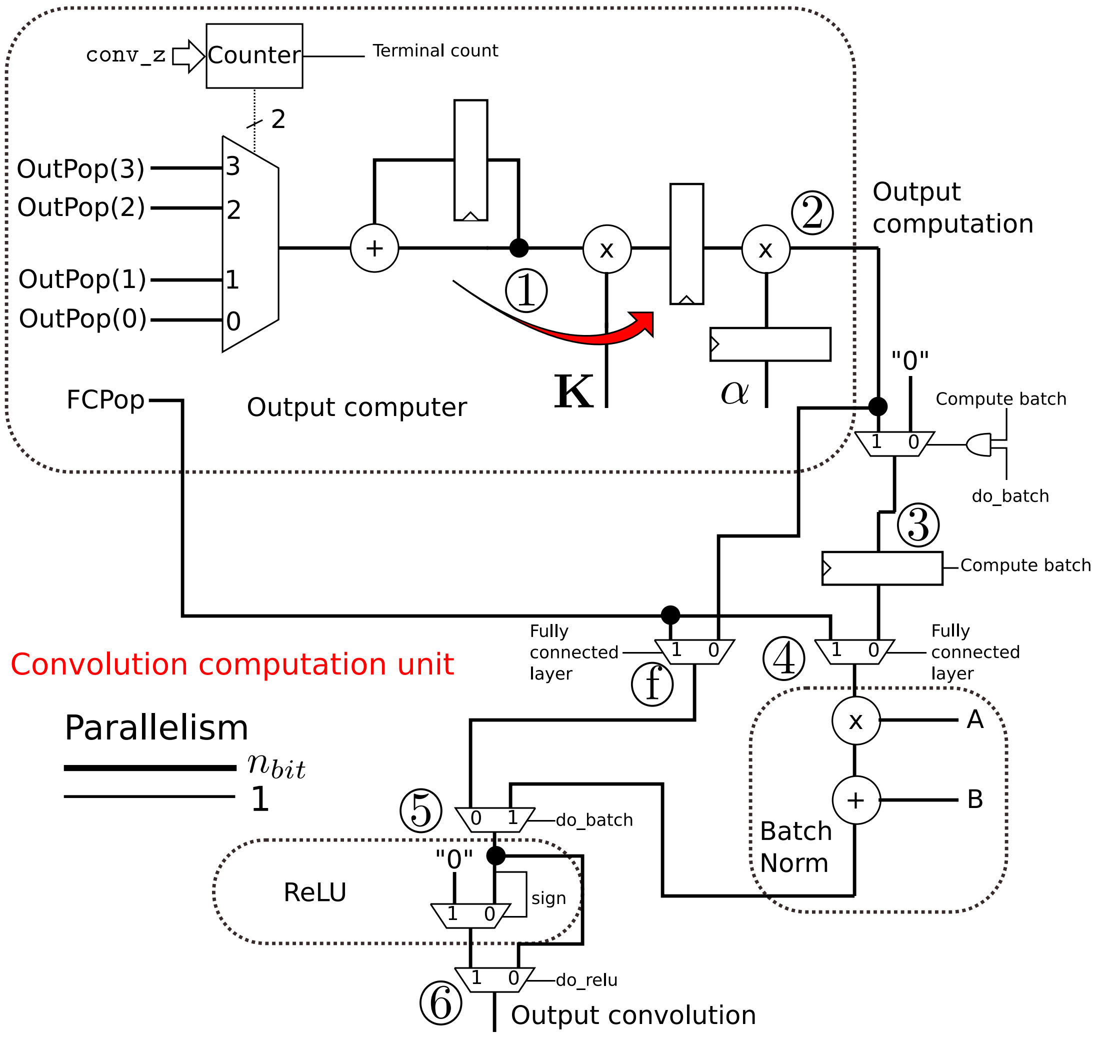


Figure 4.17: Convolution computation unit. Example of a 4 input channels output computer unit, with batch normalization and ReLU. α is delayed by a register in order to reduce the critical path. A, B are the batch normalization terms. The path indicated by the red arrow has been retimed to reduce the critical path delay.

The computational steps executed by this circuit are indicated by the circled numbers in Figure 4.17:

- ①: the first multiplexer is able to choose an `OutPop(i)` out of 4 possible inputs. Each `OutPop(i)` indicates a pop-counting result of an input channel and each of them is added in an accumulator unit;
- ②: the output computer computes the convolution result by multiplying the

pop-counting value by α and $\mathbf{K}$, as indicated in Equation 4.15;

- ③: the multiplexer selects between `output computation` and "0", based on `compute_batch AND do_batch`. The signal `compute_batch` is asserted by the control unit, when the output computer has finished its computation. The end of output computer's calculation is signaled by the counter's terminal count, which is reached when the counting value is equal to the number of input channels of the layer considered, indicated as `conv_z` (in the convolutional computation, it has a different meaning respect to the fully connected one, reported in section 4.1.2). The signal `do_batch` is handled by the user, who can decide if the Batch Normalization has to be used in that particular layer or not, so it is defined in the testbench (discussed in subsection 4.1.5);

- ④: a multiplexer selects between the output of the register and `FcPoP`, based on `Fully connected layer`, which is a signal that indicates if the layer considered is convolutional or fully connected. `Fully connected layer` is handled by the user that defines the neural network model, so it is declared in the testbench. When `Fully connected layer` is '1', `FcPop` is chosen, which is the output coming from the first input channel's pop-counting (that in the OOM implementation, corresponds to the output of the `RF TMP POP`, as depicted in Figure 4.19), because in the fully connected part the computations of α and $\mathbf{K}$ are neglected. Only the first channel's output is required in the fully connected, since one xnor pop-counting unit is sufficient to perform the computation because, by definition, the fully connected layer requires an input vector instead of matrices;

- ⑤: a multiplexer choose between the batch normalization output and the ⓕ's output: if the batch normalization is disabled, output computation or `FcPop` is chosen, depending on the layer's type. The terms A and B are given in inputs to compute the batch normalization; the computational cost of the

BatchNorm layer can be reduced considering:

$$\begin{aligned} BatchNorm &= \frac{x-\mu}{\sigma} \times \gamma + \beta \\ BatchNorm &= \frac{x}{\sigma} \times \gamma + \beta - \frac{\mu}{\sigma} \times \gamma \end{aligned} \tag{4.16}$$

$$A = \frac{\gamma}{\sigma} \tag{4.17}$$

$$B = \beta - \frac{\mu}{\sigma} \times \gamma \tag{4.18}$$

$$BatchNorm = x \times A + B \tag{4.19}$$

- ⑥: after ReLU has performed the maximum between 0 and the input, the user can choose if the ReLU's output has to be considered or not with `do_relu` signal.

Multiplication process To multiply two fixed point numbers it is sufficient to consider the following scheme:

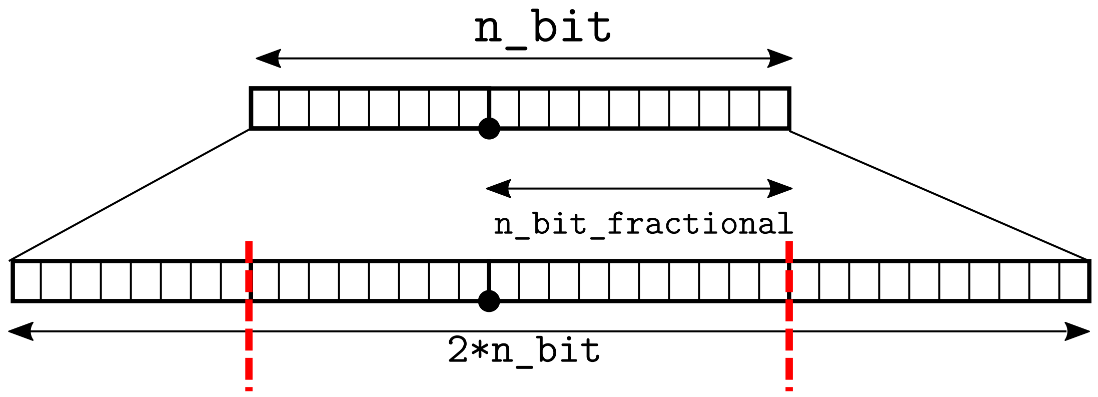


Figure 4.18: Multiplication scheme: example with n_bit=18 and n_bit_fractional = 10

The multiplication generate an output on `2*n_bit`, but since the architecture has a finite precision, the result is trucated on `n_bit` as shown in Figure 4.18.

Entire datapath

In the following figure, it is presented an example of a convolutional layer with $C_{out} = 2$ and $c_{in} = 4$: every component in the dashed red boxes is not included in the convolutional layer entity. Since the architecture has multiple input channels, the number of parallel `Max pooling` layers, `Input selectors`, `XNOR units` and `pop-counting` units are equal to c_{in}. Starting from the left side (①), the inputs are `Initial_input` provided by `Input source`, that can be the RF Input Image, max pooling output or convolutional layer output itself, as depicted in Figure 4.27. The inputs are fed to `Input selector`, which fetches the values to be convolved and stores their sign into the `Binary input RF` (②). There are c_{in} multiplexers reported in ② that are piloted by `count K` and they take one of w_{filter}^2 inputs: the blue arrow means that for each multiplexer, there are w_{filter}^2 inputs of n_{bit} each. These muxes have been already discussed in the K computation unit part (Figure 4.14), and they fed the architecture with one input pixel per time. Binary values are fetched from `Binary input RF` and convolution-α computation processes start. When the pop-count terminates (③), the output is computed by `Convolution computation unit` and batch normalization-ReLU are finally applied (④). At the end, the result is finally stored into the `Temporary RF CNV` and the new binary values are fetched from `Binary input RF` and the entire process is repeated. Once an entire OFMAP has been computed, the results are moved in parallel from `Temporary RF CNV` to `Output register files`. After that, a new output channel is processed: since the architecture depicted in Figure 4.19 has $C_{out} = 2$, multiple output kernels are used. For this purpose a multiplexer which selects the kernel values depending on the output channel considered (signaled by `Channel selected`) is employed and the entire procedure is repeated. The functionality of the convolutional layer entity is explained in the control unit part.

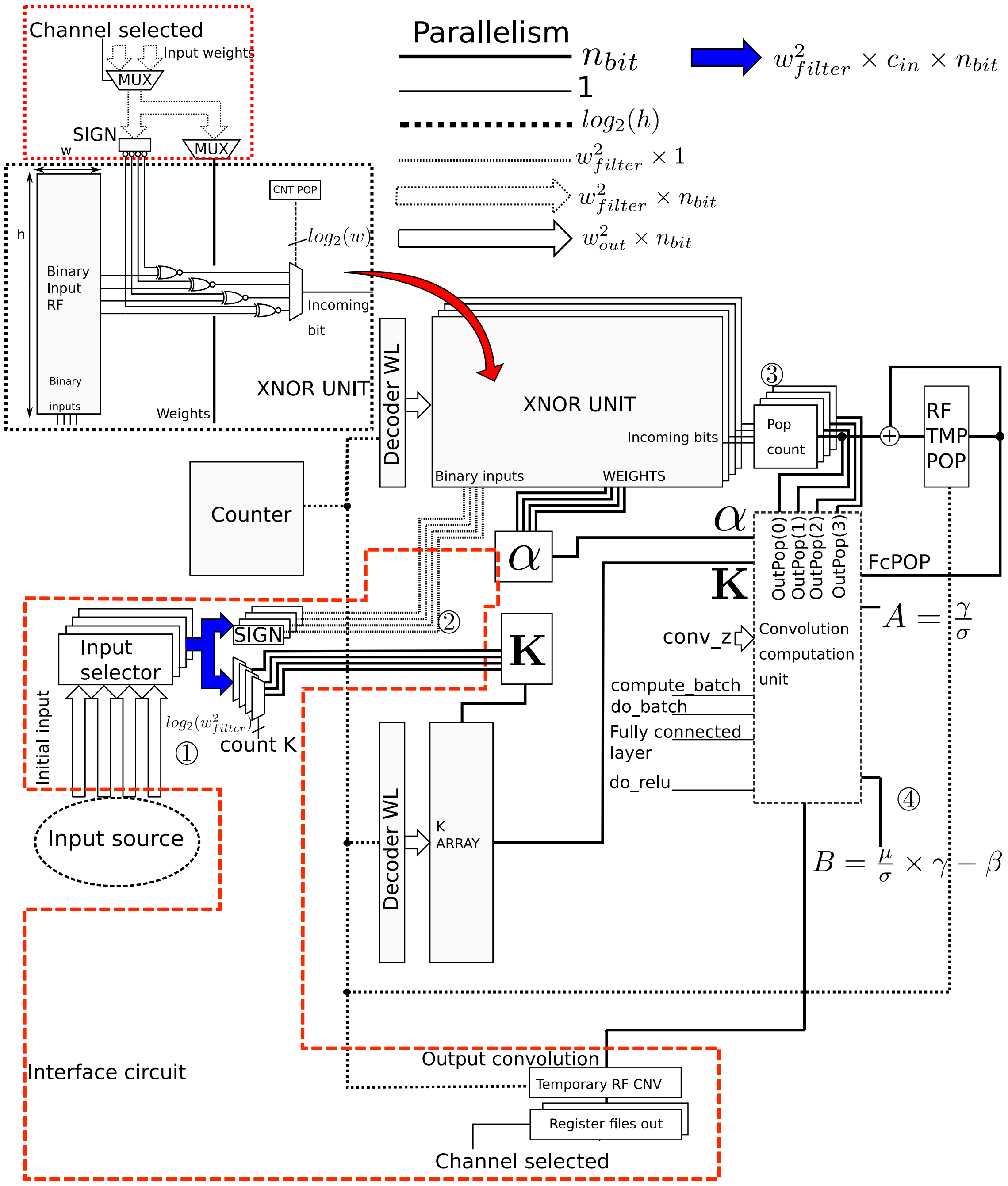


Figure 4.19: Entire convolutional layer datapath: example with $C_{out} = 2$, $C_{in} = 4$. The area highlighted by the red dashed line is implemented in an external unit.

Control unit

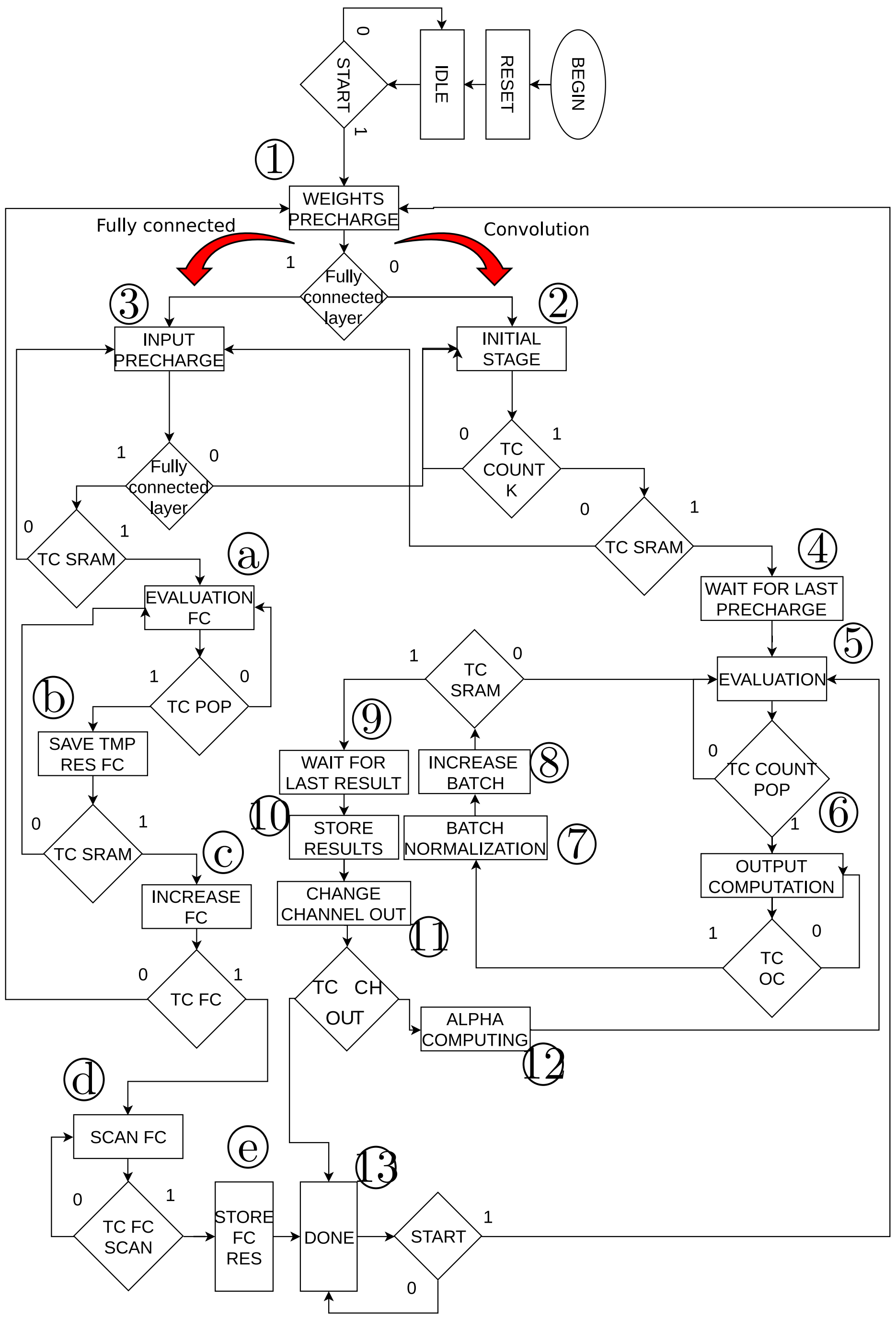


Figure 4.20: FSM of the convolutional/fully connected layers. The term "TC" indicates terminal count.

Convolution algorithm The convolutional part of the control unit is now explained. `Fully connected layer` is a signal that indicates if the layer considered is a fully connected or a convolutional, so in this part `Fully connected layer` is '0'.

- ①: dummy state that allows to precharge the first row of the `Binary input RF` and to met timing requirements. This state is particularly useful in the IN-MEMORY architecture, since also the weights are stored inside the `Binary input RF`;
- ②: during the `Initial stage`, **K** computational unit starts its computation. `Terminal count K` is reached when the first useful result of **K** is available after w^2_{filter} clock cycles (in the example proposed in Figure 4.13, after 4 clock cycles, since $w_{filter} = 2$). When it is reached, `Terminal count SRAM` is tested and, if it is '0', the FSM moves to `Input precharge`, where a new input set is provided, stored inside the `Binary Input RF` and the computed value of **K** is saved inside `k_array`. The FSM turns back to `Initial stage`, where a new value of K is computed with a new input-set. This routine ends when `Terminal count SRAM` is asserted, meaning that all the values from `Binary Input RF` have been precharged. In Figure 4.21 it is reported the timing diagram of the K computation unit;

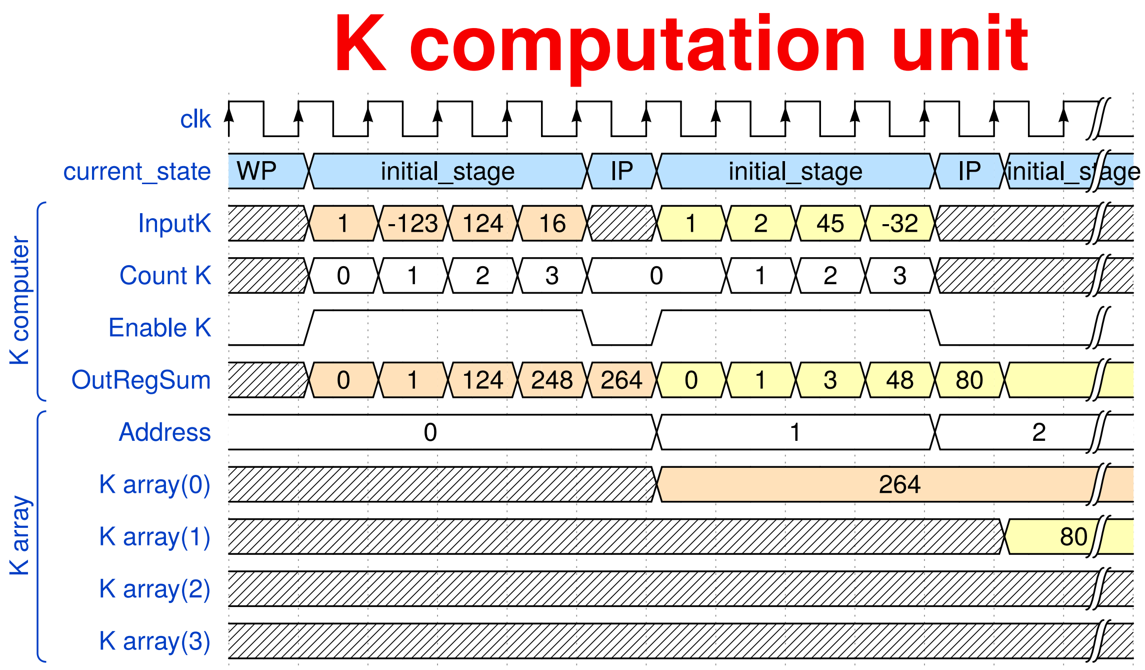


Figure 4.21: Timing diagram for the K computation considering only one input channel. When **Enable K** is asserted during **initial stage**, K computation starts to address one out of w^2_{filter} inputs with **Count K**, and the corresponding sum is obtained in **OutRegSum**. This phase lasts for w^2_{filter} clock cycles, that in this example it is equal to 4. After that, during **Input precharge** (IP), a new input set is provided and the just computed value of K is stored inside **K array**.

- ③: during `Input precharge`, the binary inputs are precharged in `Binary input RF` and `K array` starts to store the **K** results;
- ④: Dummy state that waits for the last precharge in the `Binary Input RF`;

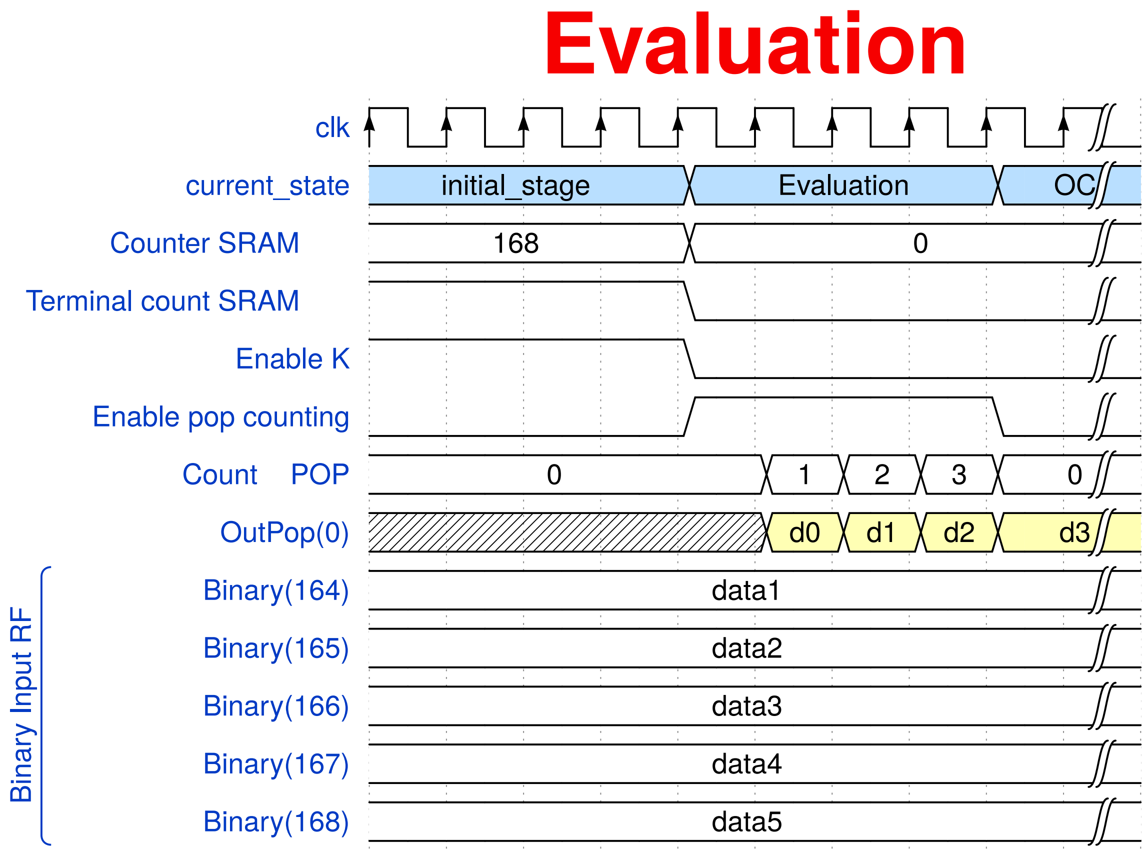


Figure 4.22: Example of timing diagram for **Evaluation** state, considering only one pop-counting unit. When **Counter SRAM** has terminated, **terminal count SRAM** is '1', allowing the FSM to move from **Initial stage** toward **Evaluation**. During this state, pop-counting is enabled and **Count pop** starts. OutPop(0) changes its value according to the xnor values: this procedure terminates when all the filter elements have been considered, so after w^2_{filter} clock cycles. In the meanwhile, alpha can start its computation.

- ⑤: during `evaluation`, the columns of the `Binary Input RF` are scanned and the pop counting procedure is performed. This state reaches the end when `Terminal count pop` is asserted: the value of `count pop` is equal to w^2_{filter};

- ⑥: output computer starts to perform its evaluation. The output computation phase has a duration given by the total number of input channels in the layer considered: `Counter` (Figure 4.17) asserts `terminal count OC` when all the inputs have been scanned, so $c_{in} \times t_{ck}$ cycles are required. Once the output computer has finished its computation (signaled by `Terminal count OC`), batch normalization and ReLU computations can start. During the state highlighted by ⑦, the result is stored in the temporary convolutional register (`Temporary RF CNV`) depicted in Figure 4.27 and the `Counter SRAM` is increased, allowing to address another row of the `Binary Input RF`: if the `Terminal count SRAM` is equal to 1, it means that the `Binary Input RF` has been completely scanned and the convolution is completed, otherwise the FSM returns on `Evaluation`, in which pop-counting operation is executed again;

- ⑨ and ⑩: once all the outputs have been computed, the FSM waits for a clock cycle and then stores the results by moving in parallel the content of `Temporary RF CNV` inside `Output register files` (Figure 4.27);

- ⑪: since it is possible to have more than one output channels, during the state `Change channel out` a counter is increased which is able to select the other set of weights by piloting the signal `Channel selected` (Figure 4.27). If the `Terminal count ch out` is not reached, meaning that there still remain output channels to consider, the FSM goes to `Alpha computing`, in which the new value of alpha is going to be computed and so the alpha unit is enabled to process new weights (Figure 4.9). The entire procedure for the new channel is repeated;

- ⑬: once all the output channels are processed, the convolutional layer has finished and asserts the `Done` signal. In this state, the FSM is waiting for a new start signal;

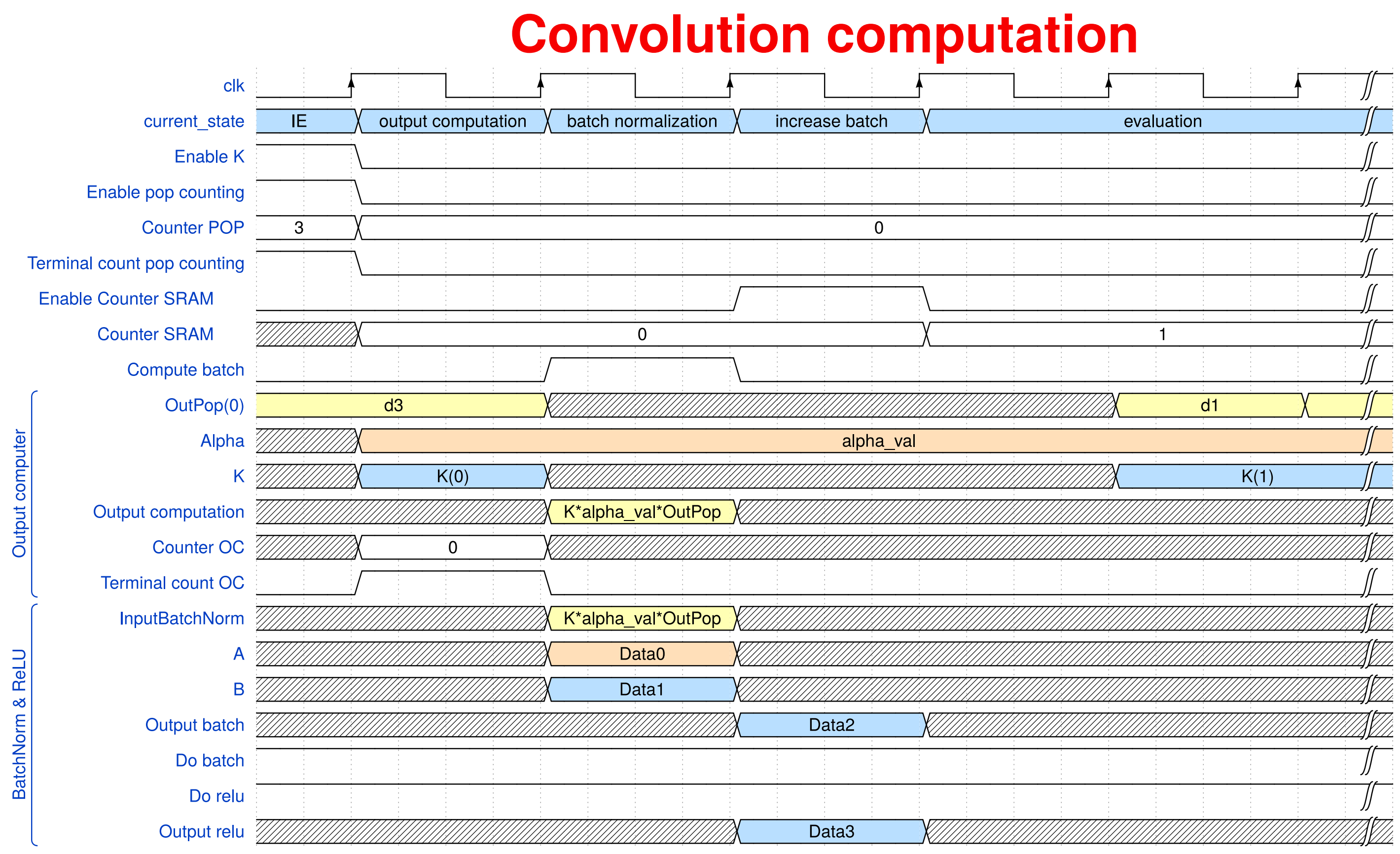


Figure 4.23: Timing diagram for the convolution computation. As it is possible to see, the FSM moves from **Evaluation** (IE) to **output computation** when **terminal count pop counting** is '1'. During **output computation**, the values of **K** (selected by the **counter SRAM**) and alpha are fed to the **output computer**, which performs the product between the **OutPop** result and these two values, obtaining **Output computation** (reference Figure 4.17). The FSM waits until **terminal count OC**, which is asserted when the **output computer** has scanned all the parallel input channels (Figure 4.17), so after c_{in} clock cycles. Since in the reference architecture depicted in Figure 4.1 there is only one parallel input channel, the FSM passes immediately to batch normalization state, which computes Batch Normalization/ReLU within a clock cycle. Moving to **increase batch** state, the **Counter SRAM** is enabled and the counting is increased, in order to consider another Binary input set from **Binary Input RF** and a new value of K, which is addressed by the counter itself. At the same time the convolution result is saved inside a temporary register file (**Temporary CNV RF** in (Figure 4.27)). The procedure restarts with **evaluation**.

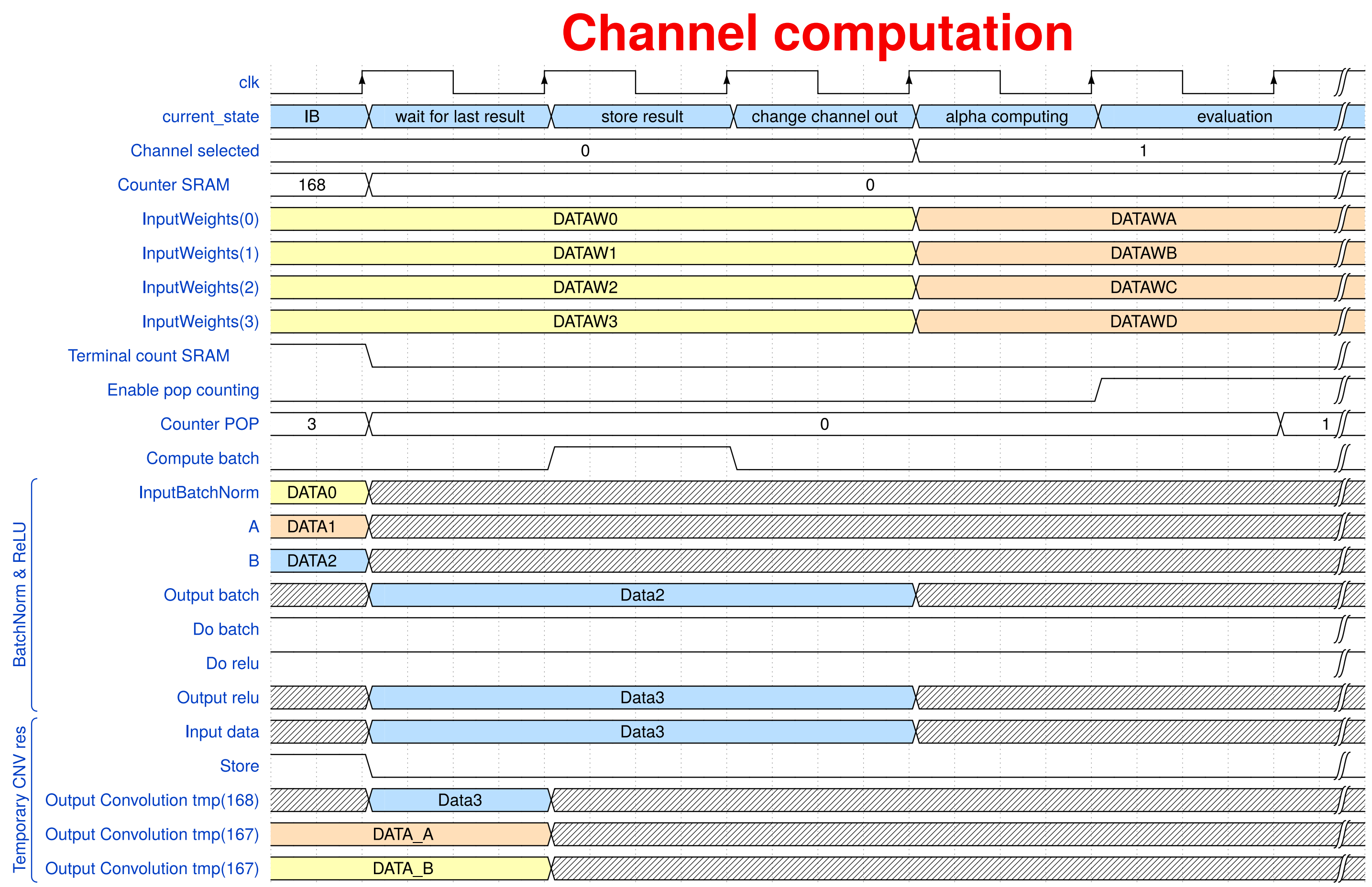


Figure 4.24: Timing diagram for multiple output channels handling. From **increase batch** (IB), the FSM moves toward **wait for last result**, since the **Counter SRAM** has reached the end of counting. The last valid data is saved inside the **Temporary CNV RF** (Figure 4.27) and, consequently, the entire content of the register file is stored in the **output register files** (Figure 4.27) during store results. At this point the channel is changed by increasing **channel selected**, which selects another weights set. Alpha is computed again and the entire process described in the previous parts is repeated.

Fully connected algorithm Now the functionality of the FSM when `Fully connected layer` is '1' is analyzed.

- ③: when fully connected layer is considered, the `Input precharge` phase is executed until the `terminal count SRAM` is not asserted. The Binary values are stored inside the `Binary Input RF`, without the execution of K computation;

- ⓐ: The evaluation of the fully connected part starts and terminates when `Terminal count pop` is asserted. This is '1' when all the L elements defined in the FC scheduling (section 4.1.2) are selected by the multiplexer, which is addressed by `Counter Pop` in Figure 4.6, so considering Figure 4.16, when `count pop` reaches 5 (0 to 5);

- ⓑ: once the pop-counting for FC has finished, the temporary result will be saved into the `RF TMP POP` (Figure 4.19);

- ⓒ: when the `Terminal count SRAM` is asserted, meaning that the precharging of `Binary Input RF` has finished, a counter that handles the FC input scheduling (as depicted in Figure 4.16) is increased. This counter allows to choose a new set of fc inputs/weights as follows:

```
InputRed = Inputs_fc(L*(count_fc+1)-1 downto L*(count_fc));
WeightsRed = Weights_fc(L*(count_fc+1)-1 downto L*(count_fc));
```

  `Inputs_fc/Weights_fc` refer to the entire word of fully connected inputs/weights. By analyzing the neural network model depicted in Figure 4.1, `Inputs_fc/Weights_fc` have a length of $number\_of\_fc\_parameter = 1014$ bits, while L (that is the width of the fc word fed into the `Binary Input RF`, as described in Figure 4.16), is equal to 6. `terminal count fc` is asserted when all the input word is scanned reaching 1014, so translating in formulas:

$$Terminal\_count\_value = \frac{1014}{6} = 169 \tag{4.20}$$

  The total number of iterations (n_{iter}) is equal to 169.

- $\textcircled{d}$: once all the inputs of the fully connected layer are considered, the output pop-counted values are scanned one by one in order to be stored in the `Temporary CNV RES`. In this state, `compute_batch` signal is equal to 1, in order to perform the batch normalization if requested;
- $\textcircled{e}$: during this state, data migrates from the `Temporary CNV RES` to the first `Output register files`. Moreover, a `.txt` file is generated containing the FC results.

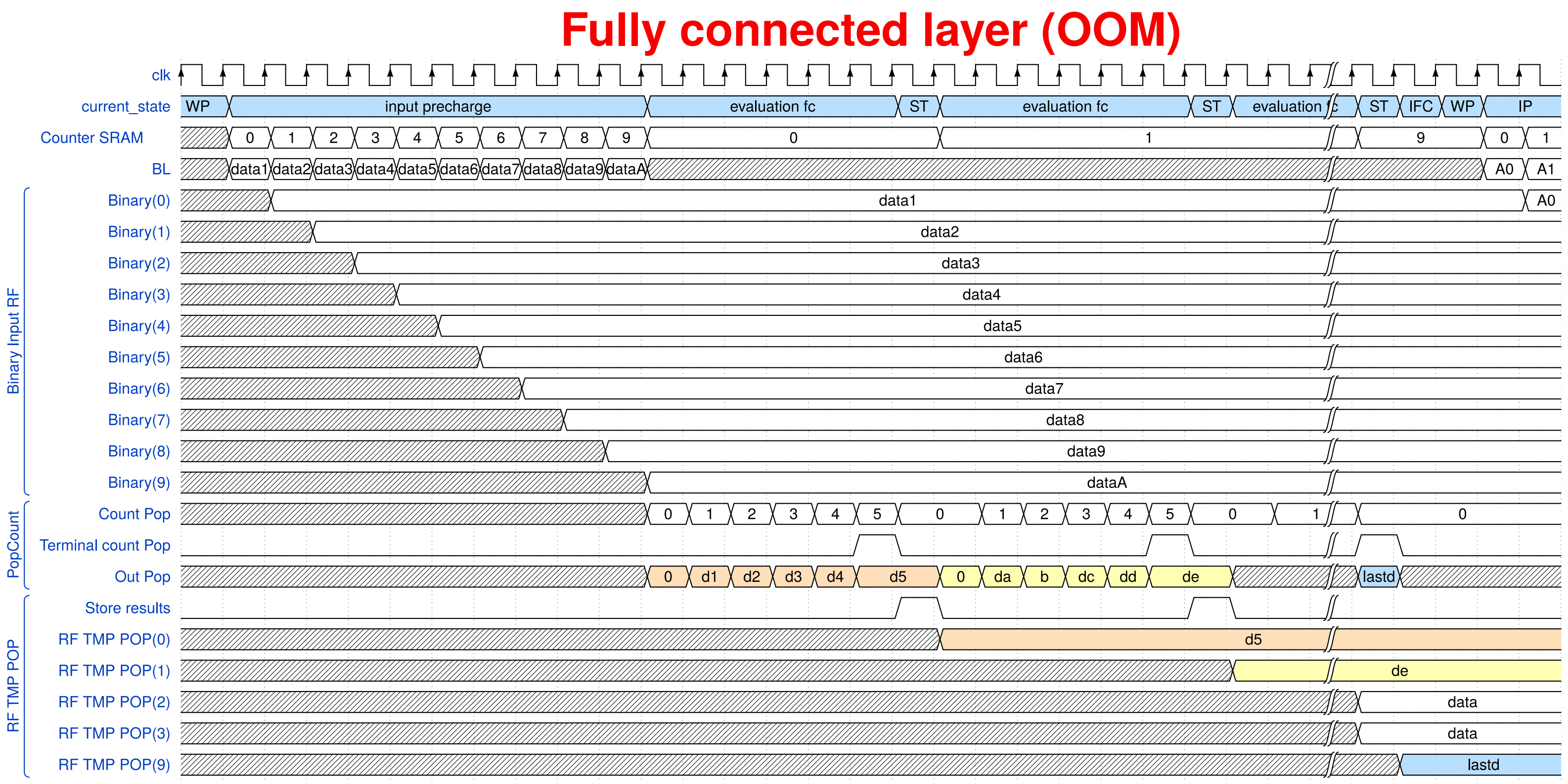


Figure 4.25: Timing diagram of the fully connected part. After **weight precharge** (WP), the FSM starts to save the binary values inside the **Binary Input RF** during **Input precharge**, as already discussed. After that, evaluation can start, in particular the first line addressed by **Counter SRAM** is pop-counted. The pop-counting procedure has a time duration equal to $L \times t_{ck}$, that in the neural network model depicted in Figure 4.1 is equal to $6 \times t_{ck}$. Once **Count Pop** has reached 5, the FSM moves to **save tmp results fc** (ST), in which the temporary result of the pop-counting procedure is saved inside the **RF TMP POP** (depicted in Figure 4.19) and the last register of the pop-counting unit is cleared (Figure 4.7). A new evaluation procedure starts, but now the second row of the **Binary Input RF** is considered, since **Counter SRAM** is increased. The entire procedure for the first part of the fc scheduling (discussed in section 4.1.2) ends when the value of **Counter SRAM** is equal to the number of output neurons, that in the neural network model depicted in Figure 4.1, it is 10. After that, the state **Increase fc** increases the value of **Count fc**, which allows to select another inputs/weights set, as reported in section 4.1.2. These computational steps are repeated for $n_{iter} = \frac{number\_of\_fc\_parameters}{L}$ number of times.

Scheduling

This layer has two different schedulings, since they are both performed convolutional and fully connected computations. By looking at the control unit depicted in Figure 4.20, it is possible to compute the clock cycles required by each state, as already done in the max-pooling part. For this purpose, the neural network model depicted in Figure 4.1 is considered.

- Convolution: it starts by storing the binary values inside `Binary Input RF` and, at the same time, K computation is perfomed requiring $w_{out}^2 \times (w_{filter}^2 + 1)$ clock cycles. The convolutional process takes, one by one, each row of the `Binary Input RF` and computes the pop-counting in w_{filter}^2 clock cycles. After that, the output computation, batch normalization/ReLU and storing results are performed taking c_{in}, 1 and 1 clock cycles respectively: the entire procedure is repeated for each output (w_{out}^2). These steps and `alpha_computation`, `store_res`, `change_channel_out` and `wait_for_last_result` have to be performed for each output channel (c_{out}), since everytime a different channel is considered, a new convolution starts.

Table 4.2: Clock cycles required by the convolutional algorithm for the OOM architecture.

State	Required clock cycles	Multiplicity
idle	1	1
weights_precharge	1	1
initial_stage	w_{filter}^2	w_{out}^2
input_precharge	1	w_{out}^2
wait_for_last_precharge	1	1
evaluation	w_{filter}^2	$c_{out} \times w_{out}^2$
output_computation	c_{in}	$c_{out} \times w_{out}^2$
batch_normalization	1	$c_{out} \times w_{out}^2$
increase_batch	1	$c_{out} \times w_{out}^2$
wait_for_last_result	1	c_{out}
store_results	1	c_{out}
change_channel_out	1	c_{out}
alpha_computing	1	c_{out}
done	1	1

Considering the neural network model depicted in Figure 4.1, the total number of clock cycles of the convolution algorithm is:

$$\begin{aligned} Convolution\_cycles &= 1 + 1 + w_{out}^2 \times (w_{filter}^2 + 1) + 1 + \\ &+ c_{out} \times w_{out}^2 \times (w_{filter}^2 + 1 + c_{in} + 1) \\ &+ c_{out} \times (1 + 1 + 1 + 1) + 1 = \\ &= 4 + 169 \times (4 + 1) + 6 \times 169 \times (4 + 1 + 1 + 1) \\ &+ 6 \times 4 = 7971 \end{aligned} \tag{4.21}$$

- Fully connected: the process starts with the weights and inputs precharging (`weights_precharge`, `input_precharge`) in the `Binary Input RF`, that requires at least $w_{out(fc)} + 1$ clock cycles to be performed. After the precharging phase has finished, the evaluation starts (`evaluation_fc`) and terminates only when it has scanned all the fully connected contributions, requiring L clock cycles (already explained in Figure 4.16): temporary results will be saved into the `RF TMP POP` (`save tmp res fc`). This procedure based on evaluation and saving results is repeated for all the output neurons, which are $w_{out(fc)}$. After all the temporary results are obtained, `increase_fc` state increases `count_fc`. Once the other inputs/weights have been selected, the entire procedure is repeated for a number of times equal to n_{iter}, which is defined as:

$$n_{iter} = \frac{number\_of\_fc\_parameters}{L} = \frac{1014}{6} = 169 \tag{4.22}$$

In fact, as already said in the fc scheduling (section 4.1.2), to fetch 1014 inputs with a L = 6 are required 169 clock cycles. At the end of the algorithm, the results are scanned in order to be saved outside the `neural_network` (`scan_fc`) and `store_fc_res` signals to the `datasave` to store the FC results.

Table 4.3: Clock cycles required by the fully connected layer algorithm.

State	Required clock cycles	Multiplicity
idle	1	1
weights_precharge	1	n_{iter}
input_precharge	$w_{out(fc)}$	n_{iter}
evaluation_fc	L	$n_{iter} \times w_{out(fc)}$
save_tmp_results_fc	1	$n_{iter} \times w_{out(fc)}$
increase_fc	1	n_{iter}
scan_fc	$w_{out(fc)}$	1
store_fc_res	1	1
done	1	1

Considering the neural network model depicted in Figure 4.1, the total number of cycles required is given by:

$$\begin{aligned} FC\_cycles &= 1 + n_{iter} \times (1 + w_{out(fc)} + \\ &+ w_{out(fc)} \times (L+1) + 1) + w_{out(fc)} + 1 + 1 = \\ &= 1 + 169 \times (1 + 10 + 10 \times (6+1) + 1) + \\ &+ 10 + 1 + 1 = 13871 \end{aligned} \tag{4.23}$$

4.1.3 Flatten layer

The flatten layer takes the convolutional results and vectorizes them. Considering Figure 4.19, `output register files` placed at the end, store the outputs coming from the convolutional process with the approach illustrated in Figure 4.2. The total number of `output register files` is equal to the maximum c_{out} of the considered neural network, and the procedure used to flatten the outputs is depicted in the following figure:

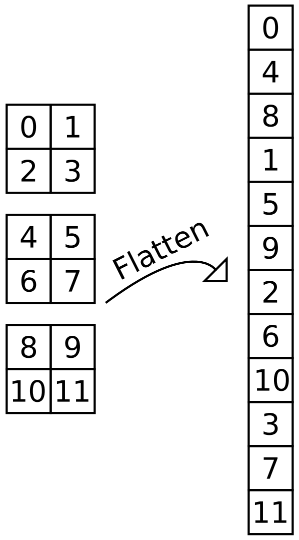


Figure 4.26: Example of flattening procedure. Each matrix represents a convolutional output channel.

And the corresponding algorithm:

```
for j=0:w_out**2-1
    for i=0:c_out-1
        if (output_convolution_stored(i)(j)==zeros)
            flat(i+j*(c_out)) = '0';
        else
            flat(i+j*(c_out)) =
            not(output_convolution_stored(i)(j)(n_bit-1));
        end
    end
end
```

This layer is simply implemented with two nested generate statements.

4.1.4 Neural network entity

The top entity of the project is called `neural_network` and contains the convolutional and max pooling layers. In Figure 4.27 it is reported an example of neural network with $c_{out} = 2$, $c_{in} = 4$.

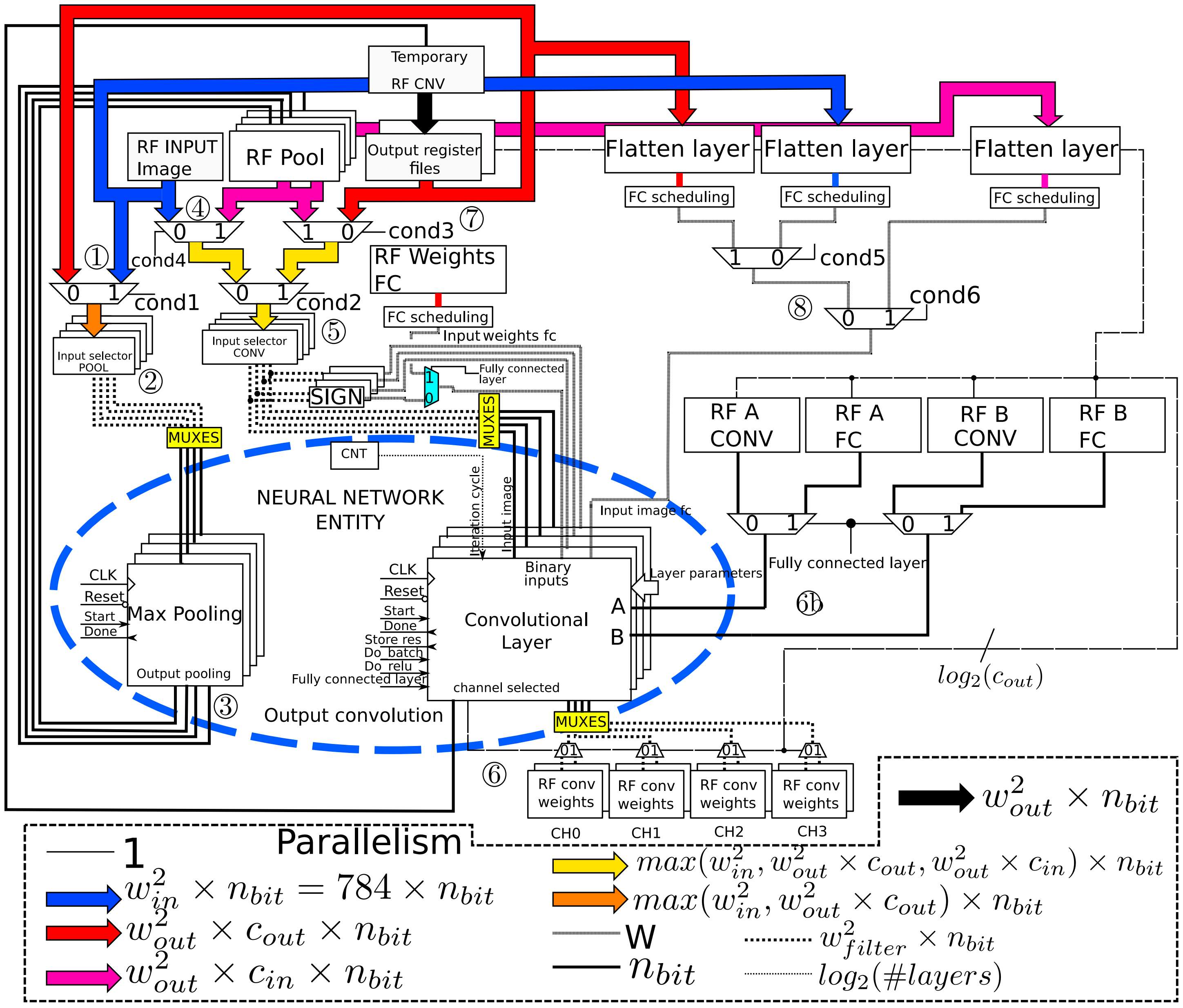


Figure 4.27: Example of a neural network top entity with $c_{out} = 2$, $c_{in} = 4$. The hardware in the dashed border-line are included in the Neural network top entity. This scheme is valid for both OOM and In-Memory architecture

Convolutional data flow

- In an external unit are placed several numbers of register files, which are used to store the values useful to the neural network to work properly. The external inputs such as input image, convolutional weights and so on are fed together to the `neural_network`. Starting from the `RF INPUT Image`, each output is fed to two multiplexers connected to `Input selector POOL` and `Input selector CONV` respectively (①), since the neural network circuit offers the possibility to perform both pooling/convolution on the input image, depending on the initial model. By these considerations, the `cond1` is defined as:

```
cond1 <= do_pool AND to_integer(unsigned(iteration_cycle))=0;
```

- The variable `iteration_cycle` indicates the layer considered in the neural network model: if it is equal to 0, the layer is the first one and so on. During an iteration cycle, it can be executed either pooling and convolutional/fully connected layers, meaning that `iteration_cycle` is increased only when both pooling/convolution have completed the algorithm asserting their `done` signals;

- If it is required a max pooling computation in the first layer, the variable `do_pool` will be equal to '1' allowing to feed the max-pooling layer with the input image. If pooling is perfomed on the input image (as done in the neural network model in Figure 4.1), the `Input image` goes into the `Input selector POOL` (②), which selects w_{filter}^2 inputs out of w_{in}^2 to fed the max pooling layers. Since the input image has 28x28x1 pixels, only the first pooling layer has to be considered.

- Once pooling has been computed, the values are stored inside the `RF Pool` and, since only one pooling layer has been used in the neural network model in Figure 4.1, the first register file is precharged. Moving toward point ④, the pooled outputs can be selected by two parallel multiplexers, both connected to the `Input preset selector CONV`. The first one (piloted by the signal `cond4`) selects between the `Input image` and the pooling result: this is useful at the beginning of the algorithm when the `Input image` has to be processed directly by the convolutional layer instead of max pooling. The `cond4` is defined as:

```
cond4 <= do_pool AND to_integer(unsigned(iteration_cycle))=0;
```

When this condition is verified (`cond4`=1), `RF Pool`'s outputs are selected and sent to the multiplexer highlighted by ⑤. The `cond2` is defined as:

```
cond2<= to_integer(unsigned(iteration_cycle))/=0;
```

When the `iteration_cycle` is higher than 0, it means that the `Input image` is not considered anymore since it has been already processed, so the multiplexer can only choose between the pooling results or the convolutional ones.

- Once the pooling results have been chosen, the `Input selector CONV` selects w_{filter}^2 out of w_{out}^2 input values and propagates them inside the convolutional layer. The yellow blocks denominated "MUXES" are placed outside the chip and they indicate that several muxes selects only one out of w_{filter}^2 inputs: this strategy allows to reduce the number of input bits of the architecture, as already discussed in K and α computation parts (section 4.1.2).

- The `convolutional layer` computes the convolution results by taking c_{in} inputs, depending on the neural network model considered (for example, Figure 4.1 uses only 1 input channel). In the example proposed in Figure 4.27, there are 2 output channels: it means that the convolution has to be executed with two different sets of input weights, that are fetched from `RF conv weights`. As it is possible to see in Figure 4.27, there are 4 couples of `RF conv weights`, since are required 2 output kernels for each input channel, so $c_{out} \times c_{in} = 2 \times 4 = 8$ register files.

- Once the convolution result has been computed, it is provided one by one to the `Temporary CNV RES`, which stores the temporary convolutional results and, when and entire output channel has been computed, data are transferred in parallel to one of the `output register files` (addressed by `channel selected`, which indicates the output channel considered). The number of `output register files` are always equal to the maximum number of output channels in the neural network model, so in this case there are 2 register files, while for the model depicted in Figure 4.1, they must be at least 6.

- The point 6b indicates the register files configuration for the A and B inputs required by the batch normalization: when `Fully connected layer`=0, each `RF A CONV`, `RF B CONV` outputs are selected by `channel selected`.
- Once the convolutional results are stored inside `output register files`, convolutional's `done` signal is asserted, `iteration_cycle` is increased, `cond2` becomes always true, since `iteration_cycle` is different from 0, and `cond3` is verified, since it is defined as:

  ```
  cond3 <= do_pool AND to_integer(unsigned(iteration_cycle))/=0;
  ```

Fully connected data flow

- When `Fully connected layer` is equal to '1', it means that the layer considered is the fully connected. In this case the convutional parameters are completely ignored by the convolutional layer, while the weights FC and the ones highlighted by ⑧ are considered. `cond5` selects between the output of the convolutional layer and the `Input image` and it is defined as:

  ```
  cond5<= fully_connected_layer AND
  to_integer(unsigned(iteration_cycle))/=0;
  ```

  It is also possible to have a max-pooling layer followed by a fully connected, so in this case the multiplexer with `cond6` signal as selector is able to choose also `RF Pool`'s outputs. `cond6` is defined as:

  ```
  cond6<=fully_connected_layer='1' and do_pool='1'
  ```

- The light blue multiplexer selects between the fc weights and the sign of the convolutional inputs, basing on `fully connected layer` signal, since the architecture has to choose the fully connected binary weights instead of the convolutional binary inputs to precharge them inside the `Binary Input RF`.

- When the `iteration_cycle` is equal to 0 and `fully connected layer` is equal to '1', the first layer considered is a fully connected and, consequently, the input image is considered as FC input. The flatten layers vectorize the matricial input and the corresponding output vector is fed to the FC scheduling, already discussed in section 4.1.2 (Figure 4.16).

Register files dimensions

The dimensions of each register file are now analyzed:

- The `RF INPUT Image` holds the input image values, which are 28x28x1 pixels of `n_bit` each;

- `RF Conv weights`: they hold the weights that are used in the convolutional process. By looking at the architecture in Figure 4.27, 2 kernels of w_{filter}^2 are needed with a bitlength of `n_bit`. The outputs are w_{filter}^2 weights of n_{bit} each that are selected by the MUXES, in order to feed only one of them per time to the convolutional layer;

- `RF A conv`, `RF B conv`: hold the values of A,B of the convolutional layer. The total number of registers with a bitlength of `n_bit` in each register file is equal to c_{out}, that in Figure 4.27 is equal to 2. The output is a single value of `n_bit` that depends on which output channel is considered;

- `RF Weights FC`: holds the values of the weights of the fully connected layer. The total number of weights required is equal to $number\_of\_fc\_parameters \times w_{out(fc)}$ (that in Figure 4.1 is equal to 1014 for each output, so $1014 \times 10 = 10140$);

- `Temporary RF CNV`: holds the temporary values of the convolution/fully connected layer. It is a register file with w_{out}^2 locations of n_{bit} each;

- `Output register files`: each register file has a number of registers equal to w_{out}^2 of `n_bit` each. The total number of register files used is equal to c_{out}, since each channel has to be stored. In the example proposed in Figure 4.27, $c_{out} = 2$;

- `RF A FC`, `RF B FC`: they store A and B parameters for the batch normalization in the fully connected layer. BatchNorm applied to a fully connected layer consists on normalize all the neurons' outputs, so the registers require at least $w_{out(fc)}$ number of registers of `n_bit` each. Considering the example in Figure 4.1, $w_{out(fc)}$ is equal to 10.

`Layer parameters` specifies to the convolutional layer what are the dimensions of the layer examinated, since each layer has different parameters (w_{filter}, w_{out}, w_{in}, ...). These define the terminal counts of the counters used in the entity, some constant values (such as $1/w_{filter}^2$ in the `Alpha computer` and `K unit`) and so on.

Control unit

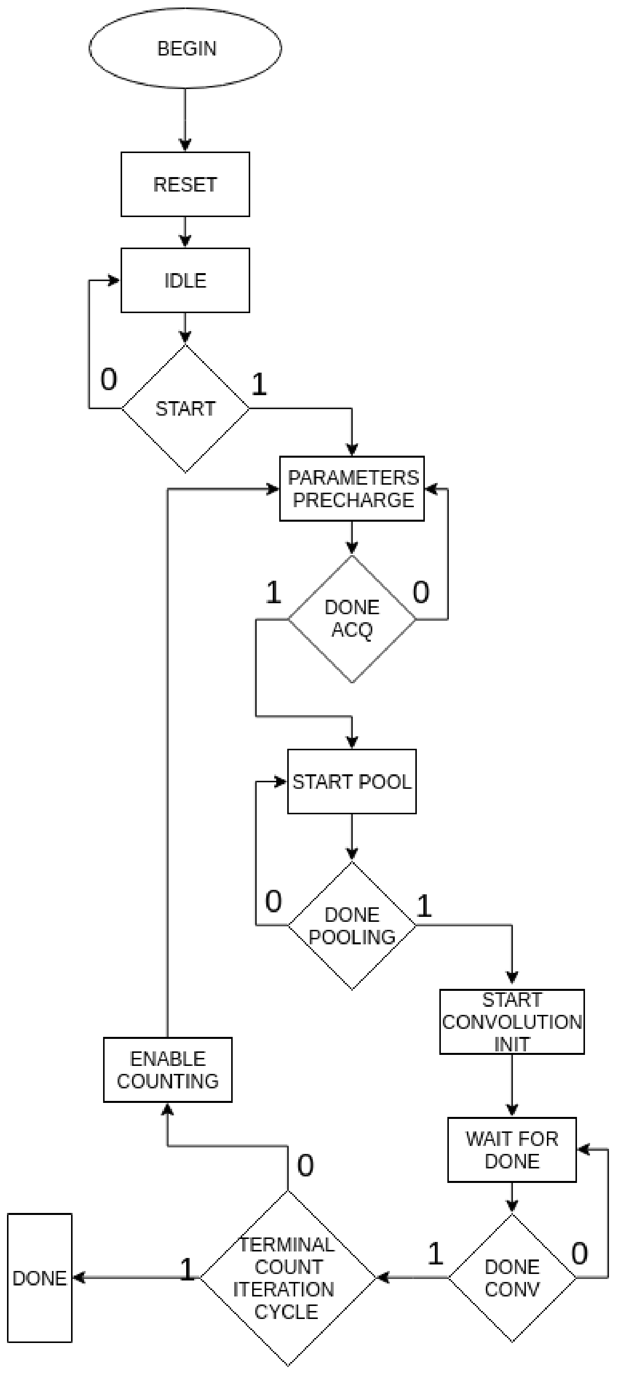


Figure 4.28: Neural network's FSM

- `Parameters precharge`: all the parameters are precharged in the register files. They are fetched one by one at the same time. This state terminates when the signal `done acq` is asserted: this is piloted from the external `data generator` and it is equal to '1' when all the inputs are stored. Considering the neural network in Figure 4.1, this state finishes when all the fully connected weights are read since they are 10140. This part will be explained in subsection 4.3.1;

- `Start pool`: pooling layer starts the computation and waits until the end. If the pooling layer is not performed (`do_pool`=0), the max-pooling's control unit asserts immediately `done pooling`;

- `Start convolution init`: once pooling asserts `done_pooling`, the convolution/fully connected can start the computation;

- `Wait for done`: FSM waits until the end of the convolution, signaled by `done conv`;

- When `done conv` is '1' it means that the `convolutional layer` has finished the computation (convolution or fully connected). At this point, the terminal count of the `iteration cycle` is tested and if it is '0', it means that the architecture has not processed all the layers defined in the neural network model and the `convolutional layer` has to be **reused** for another computations. At this point the counter that handles `iteration cycle` is enabled and the counting value increases. Also the `input parameters` of the `convolutional layer` changes, according to the layer to be examinated. The FSM moves to the `parameters precharge`, since the new layer need different values that are stored again in the register files. This is a very important concept, because it allows the **reusability** of the architecture;

- `Done`: the neural network has finished and the classification result is available.

4.1.5 VHDL implementation

From a VHDL point of view, a package has been defined, giving the possibility to implement every kind of neural network. Once the model has been designed, two different parameter-sets (called fixed and variable parameters) are chosen accordingly:

- Fixed parameters: define the worst case dimensions of the network, such as the `Binary Input RF` size, the w_{out}, w_{in} and w_{filter} values and so on;

- Variable parameters: define the actual layer's dimensions and type. They are addressed by `iteration_cycle` and allow to dinamically program the behavior of the architecture, based on the layer examinated.

Fixed parameters

In order to implement the neural network model depicted in Figure 4.1, the following fixed parameters have been used:

```
-----------------FIXED PARAMETERS---------------------
constant w_out :  integer := 14 ;
constant h_max :  integer := 169 ;
constant w_in :  integer := 28 ;
constant w_filter :  integer := 4 ;
constant w_filter_s :  integer := 2 ;
constant width_sram : integer := 6 ;
constant number_of_output_channels :  integer := 6 ;
constant number_of_input_channels :  integer := 1 ;
constant number_of_fc_parameters : integer := 1014 ;
constant number_of_sum_elements :  integer := 11 ;
constant number_of_neurons_output :  integer := 10 ;
constant input_image_size_x : integer := 28 ;
constant input_image_size_z : integer := 1 ;
constant flatten_x : integer := 169 ;
constant flatten_z : integer := 6 ;
---------- counters and other parameters ------------
constant n_bit_channel_sel :  integer := 8 ;
constant n_bit_cnw_pos_type :  integer := 9 ;
constant n_bit_counter_k :  integer := 8 ;
constant n_bit_counter_sram :  integer := 8 ;
constant number_of_bits_counter : integer := 10 ;
constant count_fc_terminal_count : integer := 169 ;
```

- `w_out`: the maximum output dimension is defined by the pooling, which takes in input a matrix of `w_in**2`=28x28 pixels and elaborates them with a `stride`= 2 and `w_filter` = 2, so:

$$w_{out} = \frac{w_{in} - w_{filter}}{stride} + 1 = \frac{28-2}{2} + 1 = 14 \tag{4.24}$$

  Since the convolutional layer after the max-pooling has as input a 14x14 matrix, the corresponding output dimension is 13x13, which is less than the required output size of max-pooling. For this motivation, `w_out` is fixed to 14;

- `h_max`: defines the number of rows (H) of the `Binary Input RF`, which has to be equal to the maximum w_{out}^2 dimension of the convolutional layer. Since in the neural network model depicted in Figure 4.1 there is only one convolutional layer, `h_max` is fixed to $13 \times 13 = 169$;

- `w_in`: defines the input size dimension, which is equal to 28, since the input image has 28x28 pixels;

- `w_filter_s` and `w_filter`: they represent w_{filter} and w_{filter}^2 respectively. They define the maximum number of contemporary inputs given in input to the convolutional/max pooling layers. Considering the neural network model depicted in Figure 4.1, the maximum number of inputs are equal to 2x2=4, since both max pooling and convolutional layers have the same kernel size;

- `width_sram`: represents the W dimension of the `Binary Input RF`. For the motivations explained in section 4.1.2, this is imposed equal to 6. This dimension must be greater-equal than L and the maximum kernel's dimensions (`w_filter`), that in the model depicted in Figure 4.1 is 4;

- `number_of_output_channels`: defines the maximum number of output channels in the model. Considering Figure 4.1, the maximum number of output channels is 6;

- `number_of_input_channels`: maximum number of contemporary input channels in the architecture. Considering Figure 4.1, only 1 channel is fed to the convolutional/pooling layers. After the convolutional layer, it is placed a fully connected layer which takes the vectorized input;

- `number_of_fc_parameter`: total number of inputs required by the fully connected layer. This number is defined by the vectorization process, which takes in input 13x13x6 IFMAPs and vectorizes them into a 1014 elements vector;
- `number_of_sum_elements`: defines the maximum number of bits required to perform the pop-counting operation. This value has been computed considering the worst case, which is the fully connected layer, since a sum among 1014 elements (`number_of_fc_parameters`) has to be computed. If they are all equal to -1, the number of bits has to be at least:

$$number\_of\_sum\_elements = log_2(|-1014|) + 1 = 11 \tag{4.25}$$

- `number_of_neurons_output`: maximum number of output neurons of the fully connected part. In the model proposed in Figure 4.1, it is equal to 10;
- `input_image_size_x` and `input_image_size_z`: define the maximum input dimensions. Since MNIST has been used, they are equal to 28 and 1;
- `flatten_x` and `flatten_z`: define how to transform the output matrix convolution into a vector. The vectorization procedure has been already presented in subsection 4.1.3.

The other parameters are the number of bits required by the counters in the architecture.

Variable parameters

The variable parameters play an important role in the architecture, since they allow to dinamically program the behavior of the neural network:

```
-----------------VARIABLE PARAMETERS---------------------
constant n_layers : integer := 2 ;
constant number_of_layers :  std_logic_vector ( n_layers-1 downto 0 ) := "01";
constant conv_layer_size_x :  int_vect ( n_layers-1 downto 0 ) := ( 1 ,14 );
constant conv_layer_size_x_pow : int_vect ( n_layers-1 downto 0 ) := ( 1 ,196 );
constant conv_layer_size_z :  int_vect ( n_layers-1 downto 0 ) := ( 169 ,1 );
constant kernel_size_xy_pow : int_vect ( n_layers-1 downto 0 ) := ( 6 ,4 );
constant kernel_size_xy :  int_vect ( n_layers-1 downto 0 ) := ( 1 ,2 );
constant kernel_size_z :  int_vect ( n_layers-1 downto 0 ) := ( 1 ,6 );
constant stride_sel_c :  int_vect ( n_layers-1 downto 0 ) := ( 1 ,1 );
constant output_size_conv :  int_vect ( n_layers-1 downto 0 ) := ( 1 ,13 );
constant output_size_conv_pow :  int_vect ( n_layers-1 downto 0 ) := ( 10 ,169 );
-------------
---------------POOLING-------------
constant pool_filter_size : int_vect ( n_layers-1 downto 0 ) := ( 1 ,4 );
constant pool_x_size : int_vect ( n_layers-1 downto 0 ) := ( 1 ,28 );
constant pool_out_size : int_vect ( n_layers-1 downto 0 ) := ( 1 ,196 );
constant pool_filter_size_s : int_vect ( n_layers-1 downto 0 ) := ( 1 ,2 );
constant pool_stride : int_vect ( n_layers-1 downto 0 ) := ( 1 ,2 );
constant pool_x_size_pow : int_vect ( n_layers-1 downto 0 ) := ( 1 ,784 );
constant pool_z_size : int_vect ( n_layers-1 downto 0 ) := ( 1 ,1 );
-------------
-------------Layer types-------------
constant do_batch_layer :  std_logic_vector ( n_layers-1 downto 0 ) := "01" ;
constant do_pool_layer : std_logic_vector ( n_layers-1 downto 0 ) := "01" ;
constant do_relu_layer :  std_logic_vector ( n_layers-1 downto 0 ) := "01" ;
constant fully_connected : std_logic_vector ( n_layers-1 downto 0 ) := "10" ;
```

As it is possible to see, they are all vectors. Each element is selected by the variable `iteration_cycle`, which takes trace on what layer is going to be computed. `Layer types` parameters defines the behavior of the network:

1. `do_batch_layer`: when '1', the variable `do_batch` that is used in the `convolution computation unit` (Figure 4.17), obtained by

   ```
   do_batch <= do_batch_layer(to_integer(unsigned(iteration_cycle)))
   ```

   it is equal to '1'. In the first layer, batch normalization is computed;

2. `do_pool_layer`: defines if the pooling layer has to be computed or not. From this vector, the variable `do_pool` is obtained, which is useful in the FSM of the pooling layer (Figure 4.4).

```
do_pool <= do_pool_layer(to_integer(unsigned(iteration_cycle)));
```

3. `do_relu_layer`: defines the variable `do_relu` as:

   ```
   do_relu <= do_relu_layer(to_integer(unsigned(iteration_cycle)))
   ```

   It is used in the `convolution computation unit` (Figure 4.17). In the first layer, ReLU is computed;

4. `fully_connected_layer`: when '1', fully connected computation is considered.

The other parameters have the following meanings:

- `n_layers`: defines the dimensions of the parameters' vectors;
- `number_of_layers`: maximum number of layers to be considered in the neural network. Considering Figure 4.1, there are only three layers (max-pool, convolution, fully connected). Since the variable `iteration_cycle` is incremented everytime a convolution/fully connected layer terminates the computation, `number_of_layers` is equal to 2 ("01"), in fact max pooling computation is integrated in the convolution computation. This is also the terminal count for the `iteration_cycle` variable, so only the first two elements of each vector can be selected;
- `convolutional_layer_size_x`: defines the dimensions of the IFMAP of the layer considered. The first layer is a convolutional with the dimensions defined in Figure 4.1: $w_{in} = 14$. The second layer is a fully connected and, since it is a different type of computation, these values are equal to 1 and they are not considered;
- `kernel_size_xy` and `kernel_size_xy_pow` are w_{filter} and w^2_{filter} respectively. Considering Figure 4.1, the first convolutional layer has a kernel size of 2x2, so `kernel_size_xy` = 2 and `kernel_size_xy_pow` = 4. The value `kernel_size_xy_pow` defines also the `terminal count pop` used in the convolutional layer's control unit (Figure 4.20) and indicates how many columns

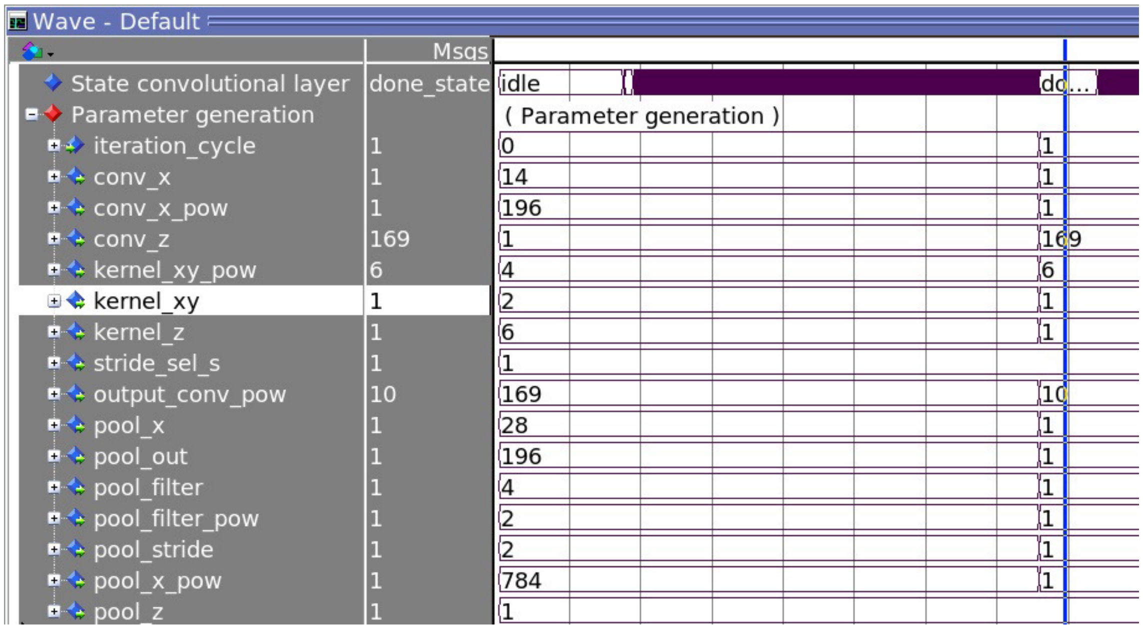
of the `Binary input RF` have to be considered for the pop-computation. In the case of the convolutional layer, this is equal to 4, while for the fully connected layer it is equal to 6 for the motivations explained in the fc scheduling in section 4.1.2: `kernel_size_xy_pow` for the fully connected layer, indicates the L value, reported in Figure 4.16;

- `convolutional_layer_size_z`: defines the number of contemporary input channels processed by the convolutional layer. The first convolutional layer has only 1 input channel. In the case of a fully connected layer, it has a different meaning: this value is equal to 169 and indicates the total number of times the FC scheduling divides the fc inputs (n_{iter}) (discussed in section 4.1.2). Considering `number_of_fc_parameters` =1014 and `kernel_size_xy_pow` = 6:
$$n_{iter} = convolutional\_layer\_size\_z = \frac{1014}{6} = 169 \tag{4.26}$$
6 out of 1014 FC inputs/weights have to be elaborated 169 times.

- `stride_sel_c`: stride values used in the convolutional layers. In the fully connected layer this value is not used;

- `output_size_conv` and `output_size_conv_pow`: refer to w_{out} and w_{out}^2 respectively. Considering Figure 4.1, the convolutional layer has $w_{out} = 13$ and $w_{out}^2 = 169$. The value of `output_size_conv_pow` is used as
`terminal count SRAM` in the convolutional layer's control unit (Figure 4.20) and defines how many rows of the `Binary input RF` have to be considered in the computation. In the fully connected layer, since there are only 10 output neurons (Figure 4.1), `output_size_conv_pow` is equal to 10.

The same considerations are valid for the pooling layer.



	Msgs		
State convolutional layer	done_state	idle	do...
Parameter generation		(Parameter generation)	
iteration_cycle	1	0	1
conv_x	1	14	1
conv_x_pow	1	196	1
conv_z	169	1	169
kernel_xy_pow	6	4	6
kernel_xy	1	2	1
kernel_z	1	6	1
stride_sel_s	1	1	
output_conv_pow	10	169	10
pool_x	1	28	1
pool_out	1	196	1
pool_filter	1	4	1
pool_filter_pow	1	2	1
pool_stride	1	2	1
pool_x_pow	1	784	1
pool_z	1	1	

Figure 4.29: Parameter generation entity. In base on the value of iteration_cycle, that changes everytime a done signal from the convutional layer is asserted, the parameters are chosen accordingly.

4.2 In-memory implementation

The original OOM circuit has been reviewed, in order to implement an In-Memory alternative. The XNOR Unit is integrated into a memory array, allowing the computation near-data and reducing the Von Neumann's bottleneck. Since the Popcounting circuitry is composed by very simple elements (memory element + full adder), it is possible to implement it in a memory-like structure, as already made for the XNOR Unit. All the other components (such as K,α and convolution computational units) remain the same.

4.2.1 Convolutional/fully connected layer

In Figure 4.30, it is reported the XNOR part integrated in memory: as it is possible to see, for each memory cell (represented by a rectangle and implemented as a flip-flop), there is a XNOR gate that execute $\overline{w_i \oplus in_j}$. Once the memory is precharged, the computation starts and all XNOR gates provide a result at the same time: for each wordline, there is a multiplexer piloted by `count pop`, that selects which xnor result to consider for the pop-counting part. At the end of pop circuits, there is a multiplexer that selects one of the pop result to be considered for the `output computer`. By having only one result per time, enables the reutilization of the convolution computation unit of the OOM implementation, depicted in Figure 4.17. The remaining parts are the same of the Figure 4.19.

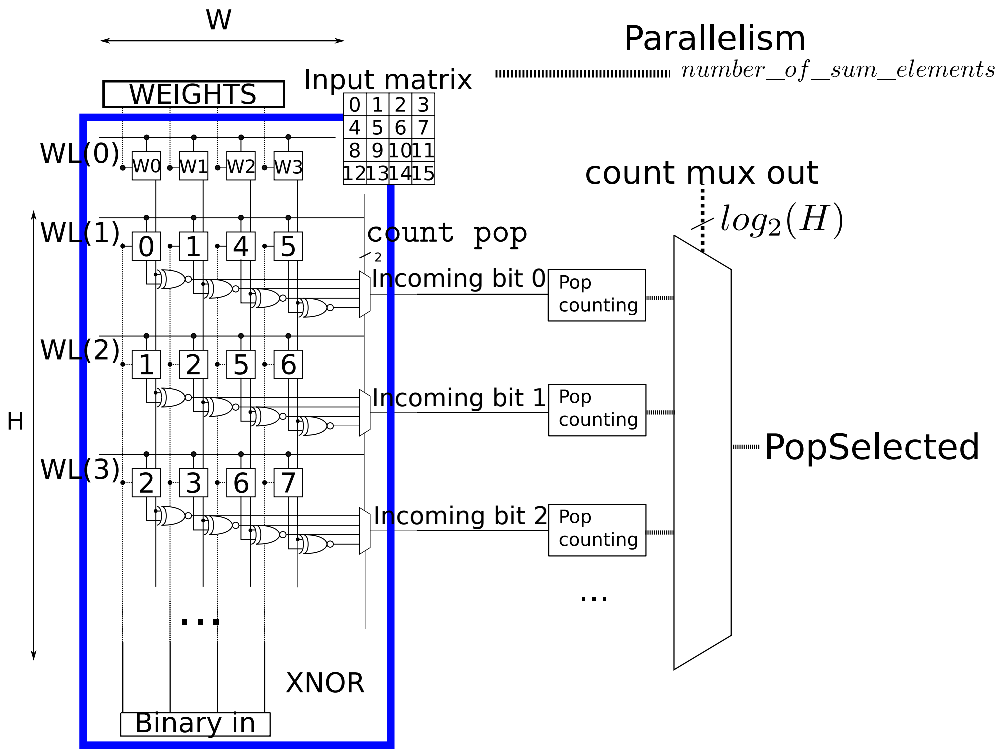


Figure 4.30: Example of XNOR in memory with $w_{in} = 4$, $w_{filter} = 2$ and W = 4. For each memory cell there is a XNOR gate that computes the xnor between the binary weights (first row) and the corresponding binary inputs. At the end of each row (excluding the first one reserved to the binary weights), there is a multiplexer which selects the Incoming bit as discussed in the OOM implementation. For each incoming bit there is a pop-counting unit and each pop-output is selected by a final multiplexer.

In the following figure it is reported the entire convolutional layer in-memory architecture:

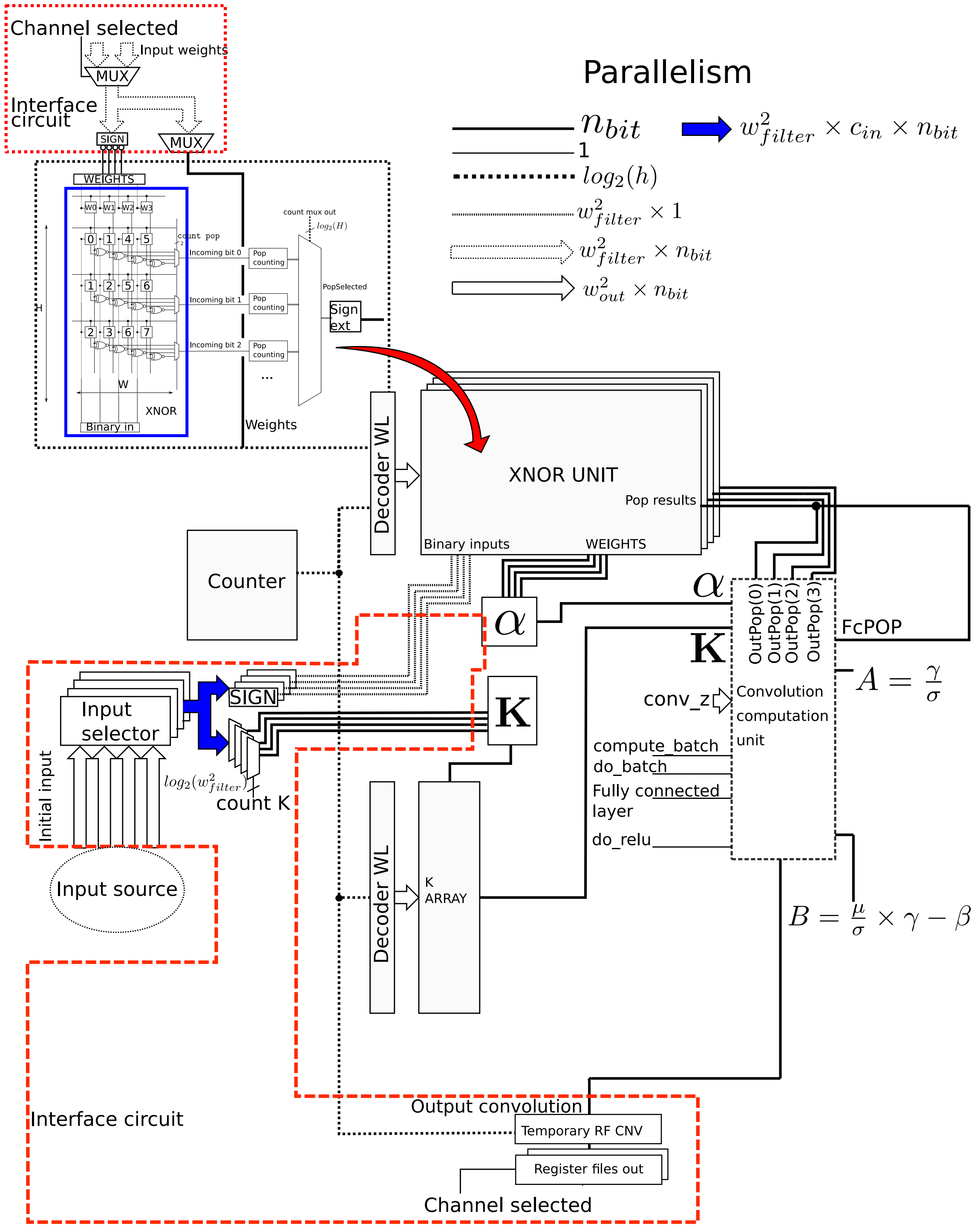


Figure 4.31: Example of an in-memory convolutional layer architecture with $c_{in} = 4$ and $c_{out} = 2$.

To implement the fully connected layer, the same approach of OOM architecture has been used, which has been already described in section 4.1.2. The main difference respect to OOM architecture is that the `RF TMP pop` is not used, because the temporary values are already stored inside of each pop-counting unit. It is sufficient to switch the output multiplexer, depicted in Figure 4.30, to have directly the correct pop-counting value.

Control unit

In the following figure, it is reported the FSM of the convolutional/fully connected layer for the In-Memory implementation. The numbers and letters depicted in Figure 4.32 indicate the differences between the OOM control unit (Figure 4.20) and the In-Memory one. The other states execute the same operations already described in the OOM implementation.

- ①: after batch normalization, `change cnv res` (change convolution result) state is executed. This state is useful, because it allows to change the result selected by the last multiplexer depicted in Figure 4.30, since the `count mux out` is increased;
- ②: the `terminal count change cnv res` is tested. It is equal to '1' when all the pop-counting outputs are scanned by the last multiplexer in Figure 4.30 that, in the neural network model in Figure 4.1, happens when `count mux out` is equal to 169. In this case, the output will be stored to `Temporary RF CNV`, otherwise a new `output computation` is performed;
- ⓐ: the most important difference between the In-Memory and the OOM architectures is located in the fully connected part. Since the in-memory architecture has multiple pop-counting units, the temporary result is already stored inside them. Once the `evaluation fc` phase has terminated, the FSM moves directly to `increase fc`, allowing to select another set of fc inputs/weights (as already described in section 4.1.2) and to speed-up FC computation.

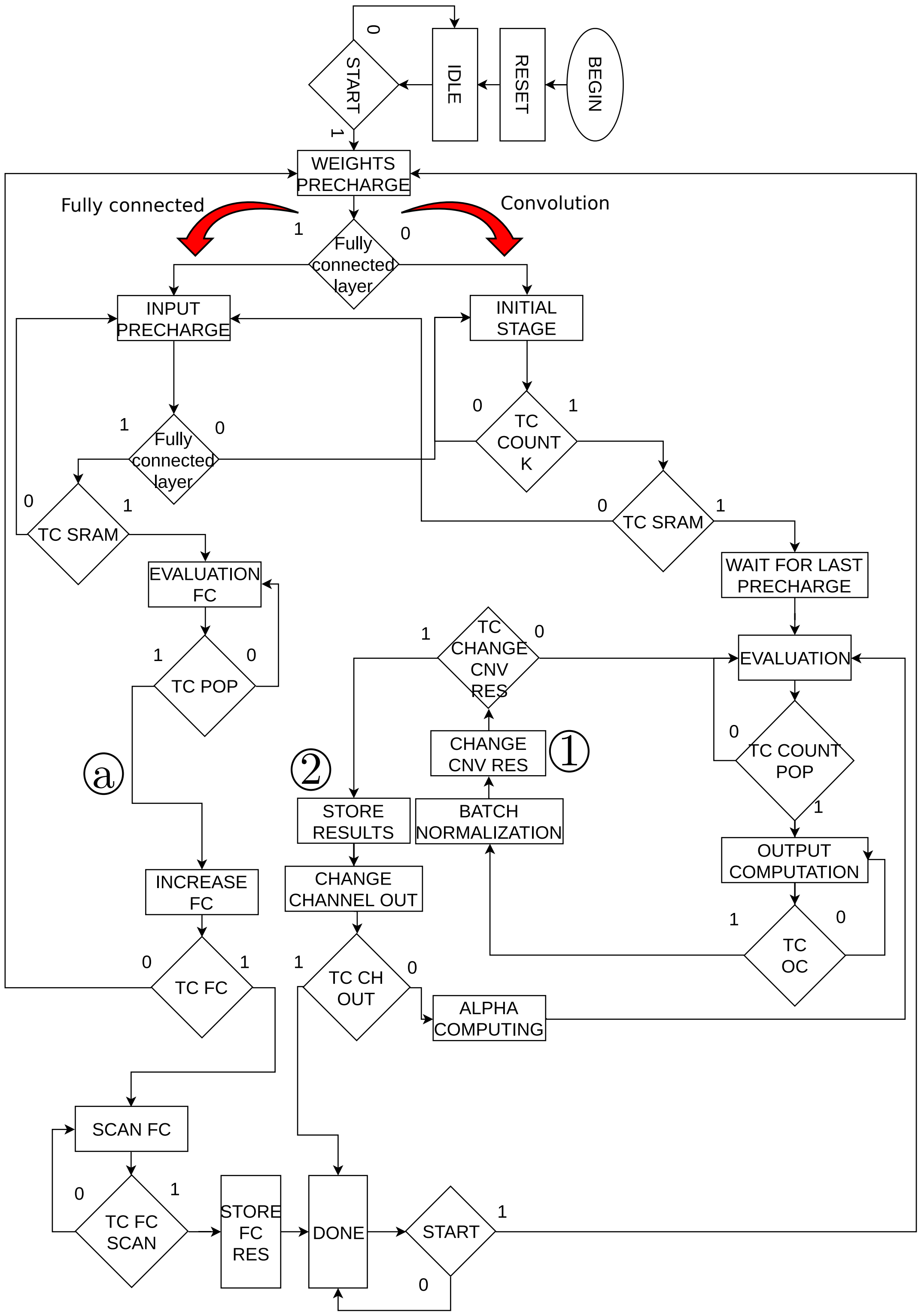


Figure 4.32: FSM of the convolutional/fully connected layer of the In-Memory implementation.

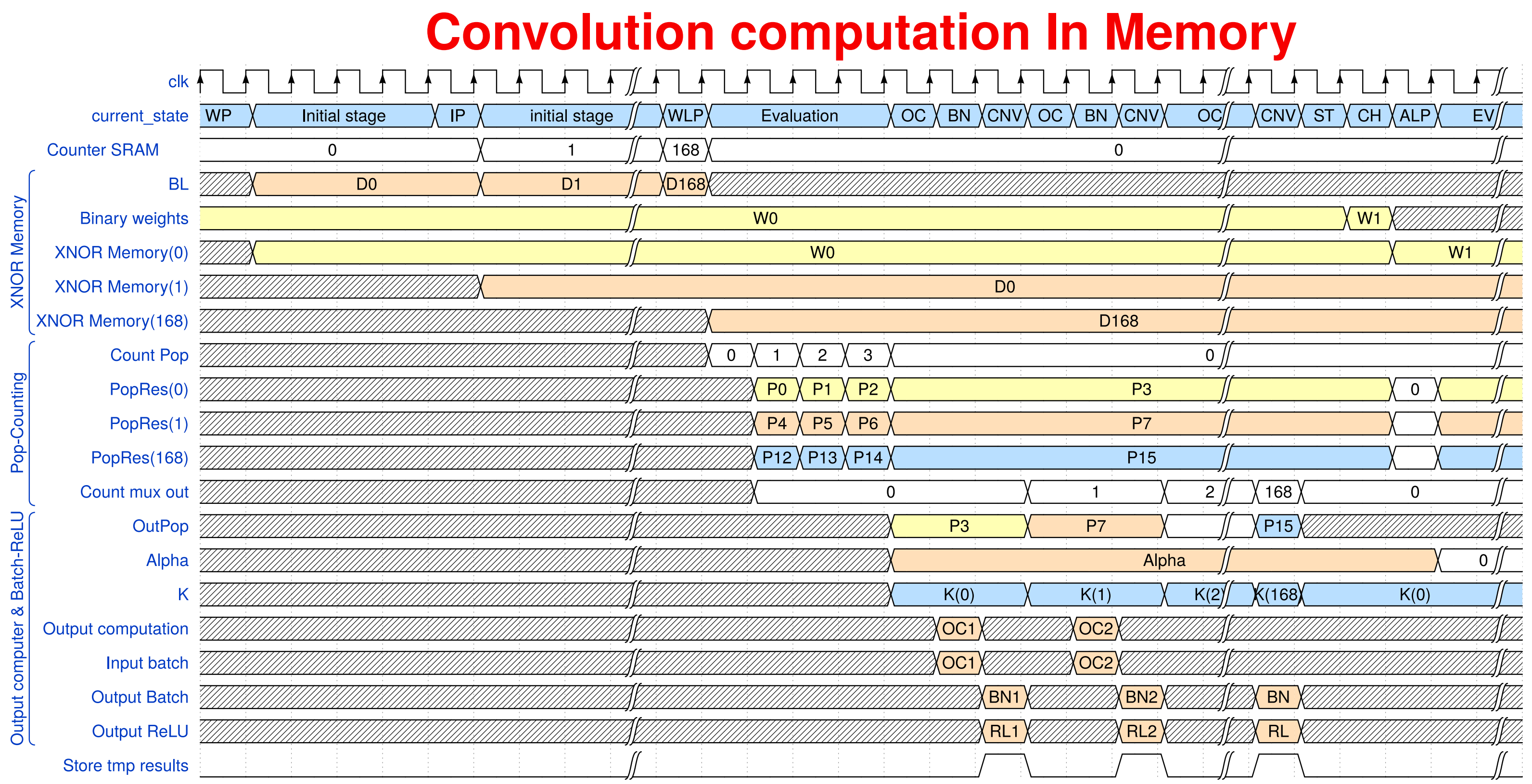


Figure 4.33: Timing diagram of convolution computation in the In-Memory architecture. Starting from **Weights precharge** (WP), the binary weights are precharged inside the first row of the **XNOR UNIT**. During **Initial stage**, K computation starts requiring w^2_{filter} clock cycles. Binary inputs are precharged inside the memory array during **Input precharge** (IP), in which also the **Counter SRAM** is increased. During **evaluation**, α starts and the pop-counting results will be computed in parallel, requiring w^2_{filter} clock cycles: this is the most important difference respect to OOM architecture, in which the evaluation process has to be repeated for each output (Figure 4.23). After pop-counting has finished, **output computation** (OC), **batch normalization** (BN) and **ReLU** computations are perfomed and repeated for each output. In **Change CNV Res** (CNV), the **count mux out** is increased and the final multiplexer in Figure 4.30, addresses another output. The procedure finishes when **count mux out** is 168 and, at this point, the second weight set is selected, α is computed again and the FSM restarts with **evaluation** (EV).

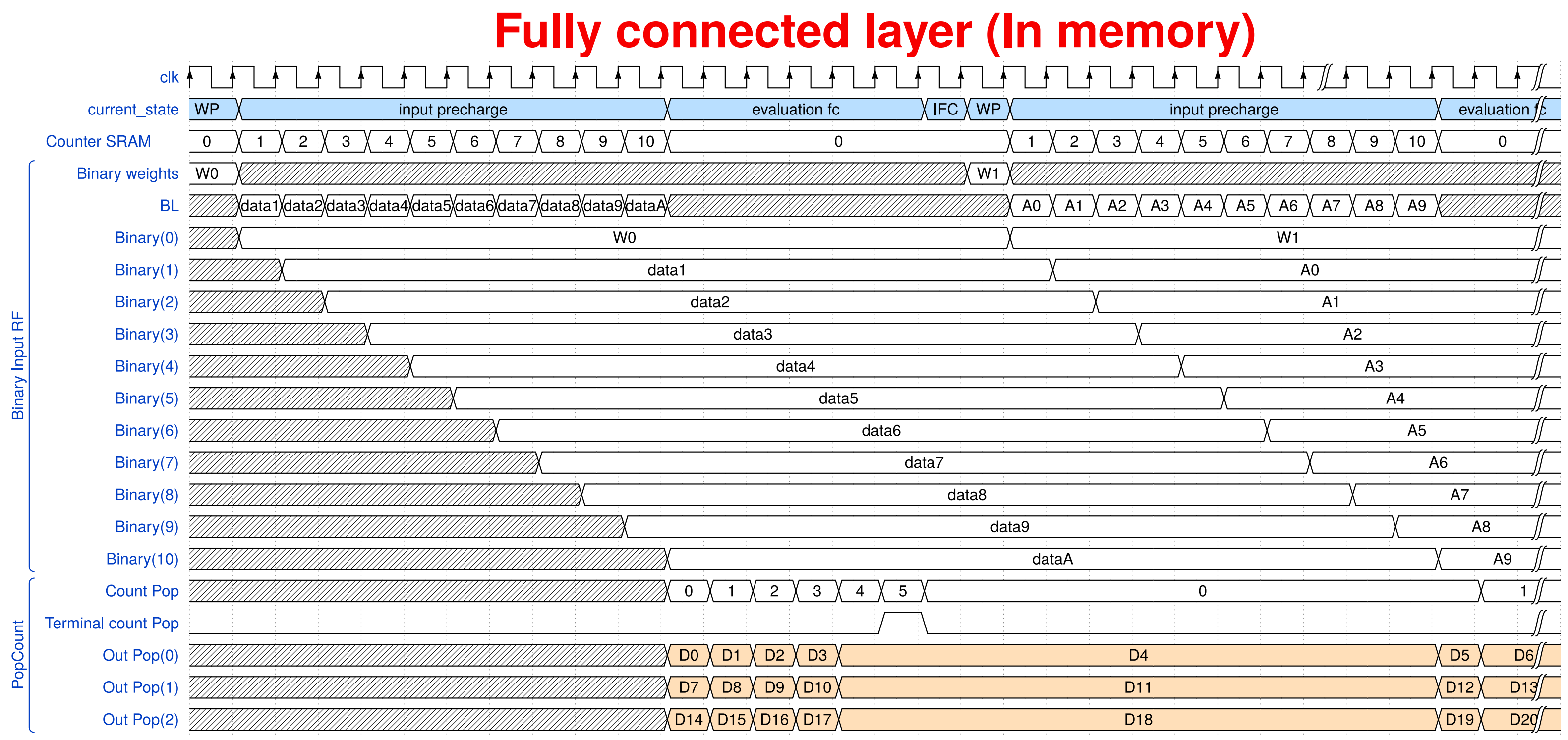


Figure 4.34: The algorithm starts with **Weights precharge** (WP) state, in which the binary fc inputs are precharged in the first row of the **XNOR Memory**, because of the inverted precharging order between weights-inputs (section 4.1.2). During **input precharge**, also the fully connected weights are stored inside the memory. **Evaluation fc** starts and ends within 6 clock cycles, since $L = 6$: in this phase, all the parallel pop-counting units are computing, obtaining at the same time the partial results of the $w_{out(fc)}$ neurons, which it is equal to 10, considering the neural network model depicted in Figure 4.1. After **evaluation fc**, the FSM increases **count fc** during **increase fc** (IFC), for the fc scheduling already explained in section 4.1.2. At this point the algorithm start again from **weights precharge**. Considering the timing diagram of the fully connected layer for the OOM case (Figure 4.25), it is possible to see the big difference between them: OOM needs to perform serially the pop-counting calculations by storing the partial results inside the **RF TMP POP**, while the In-Memory alternative can do the computation in parallel, without the need of storing the partial results, since they are maintained by the last register of the pop-counting units (Figure 4.7).

Scheduling

Also in the In-Memory case are provided the clock cycles required to compute both the convolutional and fully connected parts:

- Convolution: the process starts with the binary inputs/weights precharging (states `weights_precharge`, `initial_stage`, `input_precharge` and `wait_for_last_precharge`), that requires at least $3 + w_{out}^2 \times (w_{filter}^2 + 1)$ clock cycles, as already explained in the OOM part. After that, the data are ready to be processed: `Evaluation` begins and performs all the pop computations in parallel, obtaining the outputs ready within w_{filter}^2 clock cycles. These values are chosen by the last multiplexer, piloted by `count mux out` in Figure 4.30, and output computation is performed, requiring c_{in} clock cycles for each output. After output computation, batch normalization and ReLU are performed and need only 1 clock cycle to be executed. This procedure is repeated for all the w_{out}^2 outputs. When all the outputs have been computed, the results can be stored (`store_results`) and the output channel can be changed (`change_channel_out`). By changing the kernel, the entire procedure is repeated for c_{out} number of times, requiring also a new alpha computation. Detailed informations on time durations of each state are reported in Table 4.4.

Table 4.4: Clock cycles required by the convolutional algorithm for the In-Memory architecture.

State	Required clock cycles	Multiplicity
idle	1	1
weights_precharge	1	1
initial_stage	w_{filter}^2	w_{out}^2
input_precharge	1	w_{out}^2
wait_for_last_precharge	1	1
evaluation	w_{filter}^2	c_{out}
batch_normalization	1	$c_{out} \times w_{out}^2$
output_computation	c_{in}	$c_{out} \times w_{out}^2$
change_cnv_res	1	$c_{out} \times w_{out}^2$
store_results	1	c_{out}
change_channel_out	1	c_{out}
alpha_computing	1	c_{out}
done	1	1

For the neural network model depicted in Figure 4.1, the total convolution delay clock cycles are equal to:

$$\begin{aligned} Convolution\_cycles &= 1 + 1 + (w_{filter}^2 + 1) \times w_{out}^2 + 1 + \\ &+ c_{out} \times (w_{filter}^2 + w_{out}^2 \times (1 + c_{in} + 1)) + \\ &+ c_{out} \times (1 + 1 + 1) + 1 = \\ &= 1 + 1 + (4 + 1) \times 169 + 1 + 6 \times (4 + 169 \times (1 + 1 + 1)) + \\ &+ 6 \times 3 + 1 = 3933 \end{aligned} \tag{4.27}$$

- Fully connected: weights/inputs are precharged requiring $w_{out(fc)} + 1$ clock cycles. Evaluation starts (`evaluation_fc`) and terminates when all the columns of the custom memory have been scanned (L clock cycles). The results are already stored inside the pop-counting units, so the algorithm moves to `increase_fc`, in which `count_fc` is increased. The entire procedure is repeated n_{iter} times, in order to complete the entire fully connected layer. After that, the outputs are scanned to be saved inside the external memory, requiring $w_{out(fc)}$ clock cycles.

Table 4.5: Clock cycles required by the fully connected layer algorithm for the In-Memory architecture.

State	Required clock cycles	Multiplicity
idle	1	1
weights_precharge	1	n_{iter}
initial_stage	$w_{out(fc)}$	n_{iter}
evaluation_fc	L	n_{iter}
increase_fc	1	n_{iter}
scan_fc	$w_{out(fc)}$	1
store_fc_res	1	1
done	1	1

Where n_{iter}:

$$n_{iter} = \frac{number\_of\_fc\_parameters}{L} = \frac{1014}{6} = 169 \tag{4.28}$$

Considering the neural network model depicted in Figure 4.1, the total number of cycles required is given by:

$$\begin{aligned} FC\_cycles &= 1 + n_{iter} \times (1 + w_{out(fc)} + L + 1) + w_{out(fc)} + 1 + 1 = \\ &= 1 + 169 \times (1 + 10 + 6 + 1) + \\ &+ 10 + 1 + 1 = 3055 \end{aligned} \tag{4.29}$$

4.3 Memories' sizes

As reference, it is used the neural network model depicted in Figure 4.1. These considerations are valid for both the in-memory and OOM architectures, since they are very similar to each other.

4.3.1 Parameters precharging

During the parameters precharge phase, executed at the beginning of the algorithms, the values are stored inside the register files. Each parameter is fed to them in parallel and only one per clock cycle. Taking for example the convolutional weights precharging:

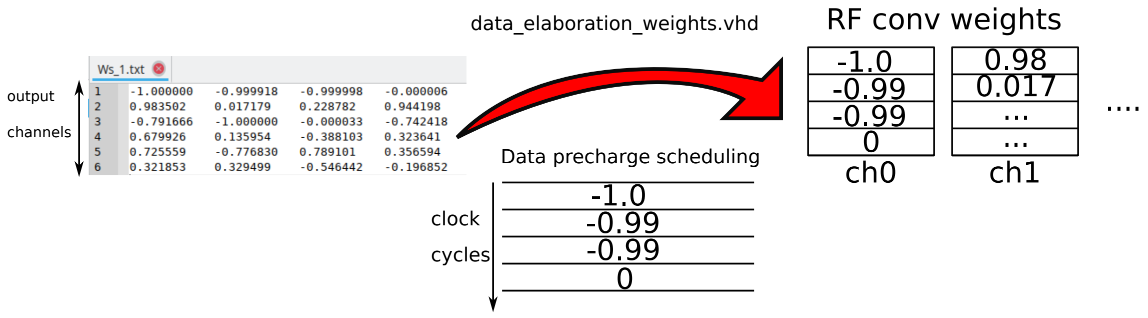

```
-1.000000   -0.999918   -0.999998   -0.000006
0.983502    0.017179    0.228782    0.944198
-0.791666   -1.000000   -0.000033   -0.742418
0.679926    0.135954    -0.388103   0.323641
0.725559    -0.776830   0.789101    0.356594
0.321853    0.329499    -0.546442   -0.196852
```

Figure 4.35: Data precharging scheduling. One data of n_{bit} per clock cycle is stored in the register files.

This scheduling is valid for all the parameters of the network, exception for the fully connected weights, in fact they are fed to the memory already binarized, in order to reduce the total number of input bits and the required memory. However, since a wordlength of 1014 is very long, because it requires at least 1014 input bits, the precharge scheduling of the fully connected part has been changed making the following considerations:

1. For the fully connected computation are required at least 1014 weights for the total number of neurons in output, which for the model considered in Figure 4.1 is 10, producing 10x1014 total bits, where the first number indicates the rows and the second one the columns of a matrix. The straight-forward way to save them is to feed them by columns, i.e. 1014 bits for each output neuron;

2. By inverting the precharging order and selecting the rows instead of columns, only 10 bits are given in input for 1014 times.

The total time required by data precharging is given by the maximum precharging time of all the required parameters, which are:

- Input image: $w_{in}^2 = 784$ clock cycles required;

- Convolutional weights: $w_{filter}^2 \times c_{out} = 4 \times 6 = 24$ clock cycles required;
- Fully connected weights: $number\_of\_fc\_parameters = 1014$ clock cycles required;
- A,B convolutional parameters: the batch normalization parameters are used for each output channel, so $c_{out} = 6$ clock cycles required;
- A,B fully connected parameters: not considered in the neural network model depicted in Figure 4.1, but in general, they are needed $w_{out(fc)}$ batch normalization parameters in the fully connected computation (one for each output).

The precharge time is equal to:

$$Precharge\_time = max(1014,784,24,10,6) \times t_{ck} = 1014 \times t_{ck} \tag{4.30}$$

4.3.2 Memory required

The total memory required to store all the parameters required by the architecture, can be computed considering the Table 4.6. The values used for the evaluations refers the neural network model depicted in Figure 4.1 and are the following:

$$w_{in} = 28$$
$$image\_size\_z = 1$$
$$w_{out} = 14$$
$$w_{filter} = 2$$
$$c_{out} = 6$$
$$c_{in} = 1$$
$$w_{out(fc)} = 10$$
$$number\_of\_fc\_parameters = 1014$$
$$n_{bit} = 18$$

Table 4.6: Memory required with $n_{bit} = 18$. All the parameters used in these computations are defined in the fixed parameters part in section 4.1.5 and the model used is depicted in Figure 4.1

Parameter	Size	Memory [kB]
Output convolution	$n_{bit} \times (c_{out} + 1) \times w_{out}^2$	3.0870
Output pooling	$n_{bit} \times w_{out}^2 \times c_{in}$	0.441
Convolution weights	$n_{bit} \times w_{filter}^2 \times c_{out} \times c_{in}$	0.054
A_{conv},B_{conv}	$n_{bit} \times c_{out}$	0.027
A_{FC},B_{FC}	$n_{bit} \times w_{out(fc)}$	0.045
Fully connected weights	$w_{out(fc)} \times number\_of\_fc\_parameters$	1.268
Input image	$w_{in}^2 \times image\_size\_z \times n_{bit}$	1.764
Total		6.686

The number of bits `n_bit` is fixed to 18, as discussed in section 4.5. In general, the trend of required memory in function of `n_bit` is reported in the following plot:

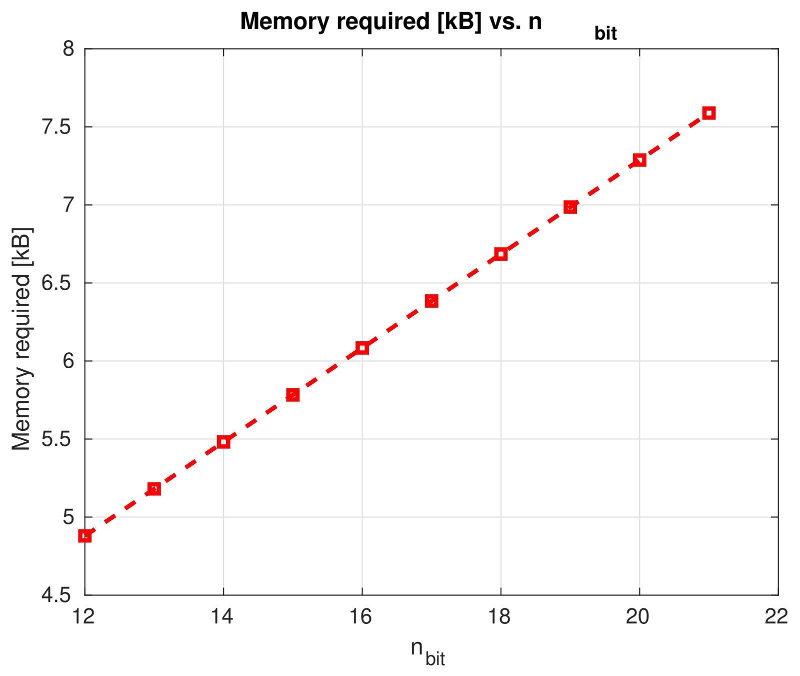


Figure 4.36: Memory required in function of n_bit for the neural network model depicted in Figure 4.1

4.4 Timing comparison

To perform a timing comparison between the OOM and the In-Memory architectures, the neural network model depicted in Figure 4.1 is used.

4.4.1 OOM implementation

The OOM architecture requires an amount of time that can be defined by considering all the stages of the `neural_network` and the time durations of each state of the FSMs, that have been already computed. The total time required by the algorithm is given by the sum of the following contributions:

1. Data acquisition: the time required is equal to the maximum number of parameters that has to be fetched from the `data generator`. In the case reported in Figure 4.1 is equal to $number\_of\_fc\_parameters = 1014$. This procedure is performed everytime a done signal from the convolutional layer is asserted. At the beginning, the image is precharged and, once it has been stored, it will be not precharged anymore. The precharging phase delay contribution is given by:

$$\begin{aligned}Delay_{data(acq)} = [&max(w_{filter}^2, number\_of\_fc\_parameters, w_{in}^2, A_{conv}, A_{FC}) + \\ +&(n_{layers} - 1) \times max(w_{filter}^2, number\_of\_fc\_parameters, A_{conv}, A_{FC})] \times t_{ck}\end{aligned} \tag{4.31}$$

   Where n_{layers} is the number of convolutional/fully connected layers in the network. To reduce the equation's length:

$$\begin{aligned}&Delay_{data(acq)} = (\phi + (n_{layers} - 1) \times \psi) \times t_{ck} \\ &\phi = (max(w_{filter}^2, number\_of\_fc\_parameters, w_{in}^2, A_{conv}, A_{FC}) \\ &\psi = max(w_{filter}^2, number\_of\_fc\_parameters, A_{conv}, A_{FC})\end{aligned} \tag{4.32}$$

2. Max pooling: the total Pool time, already reported by Equation 4.1, is given

by:

$$\begin{aligned} Pool\_time &= (1+1+1+w_{out(pool)}^2 \times (1+w_{filter}^2+1)+1) \times t_{ck} \\ &= (3+196 \times (1+4+1)+1) \times t_{ck} = 1180 \times t_{ck} \end{aligned} \tag{4.33}$$

3. Perform convolution: from Equation 4.21:

$$\begin{aligned} Convolution\_cycles &= 1+1+w_{out}^2 \times (w_{filter}^2+1)+1+ \\ &+ c_{out} \times w_{out}^2 \times (w_{filter}^2+1+c_{in}+1) \\ &+ c_{out} \times (1+1+1+1)+1 = \\ &= 4+169 \times (4+1)+6 \times 169 \times (4+1+1+1) \\ &+ 6 \times 4 = 7971 \times t_{ck} \end{aligned} \tag{4.34}$$

It is possible to distinguish between the precharging values time and perform convolution as follows:

- The term $3+w_{out}^2 \times (w_{filter}^2+1)$ is formed by the contributions of `idle`, `initial_stage`, `input_precharge` and `wait for last precharge` states. During these periods, the `Binary input RF` is precharged so:

$$Precharging\_Values = 3+w_{out}^2 \times (w_{filter}^2+1) \tag{4.35}$$

- The remaining terms in the Equation 4.34 come from the convolution computation delay:

$$Perform\_convolution = c_{out} \times w_{out}^2 \times (w_{filter}^2+c_{in}+2)+c_{out} \times 4+1 \tag{4.36}$$

4. Fully connected layer:

$$\begin{aligned} FC\_cycles &= 1+n_{iter} \times (1+w_{out(fc)}+ \\ &+ w_{out(fc)} \times (L+1)+1)+w_{out(fc)}+1+1 = \\ &= 1+169 \times (1+10+10 \times (6+1)+1)+ \\ &+ 10+1+1 = 13871 \times t_{ck} \end{aligned} \tag{4.37}$$

Table 4.7: Timing of the OOM architecture. The reference neural network model is in Figure 4.1

Operation	**Time required**	**Value**
Data acquisition	$(\phi + (n_{layers} - 1) \times \psi) \times t_{ck}$	$2 \times 1014 \times t_{ck}$
Max Pooling	$(3 + w^2_{out(pool)} \times (2 + w^2_{filter}) + 1) \times t_{ck}$	$1180 \times t_{ck}$
Convolutional layer		
Precharge binary values	$3 + w^2_{out} \times (w^2_{filter} + 1) \times t_{ck}$	$848 \times t_{ck}$
Perform convolution	$(c_{out} \times w^2_{out} \times (w^2_{filter} + c_{in} + 2) + c_{out} \times 4 + 1) \times t_{ck}$	$7123 \times t_{ck}$
Fully connected layer		
Perform computation	$[3 + n_{iter} \times (2 + w_{out(fc)} + w_{out(fc)} \times (L + 1)) + w_{out(fc)}] \times t_{ck}$	$13871 \times t_{ck}$
Total		$25050 \times t_{ck}$

4.4.2 In-memory implementation

Here the main differences are in the convolutional-fully connected layers:

1. Perform convolution: after the binary values are precharged, since there are w^2_{out} pop-counting units in parallel, the pop operation has a time duration equal to $w^2_{filter} \times t_{ck}$. Once pop-counting has been performed, the final multiplexer (Figure 4.30) needs w^2_{out} clock cycles to select all the inputs. This procedure is repeated for each output channel, but the inputs coming from the `XNOR UNIT in memory` are not fetched because the computation is already performed inside the memory: the `evaluation` phase can start reducing the total number of clock cycles required.

$$\begin{aligned} Convolution\_cycles = {} & [3 + (w^2_{filter} + 1) \times w^2_{out} + \\ & + c_{out} \times (w^2_{filter} + w^2_{out} \times (c_{in} + 2)) + \\ & + c_{out} \times 3 + 1] \times t_{ck} \end{aligned} \tag{4.38}$$

2. Fully connected layer: for the same motivations, the fully connected layer, once the inputs/weights are precharged, can be computed in parallel, without the need to fetch each row of the `XNOR UNIT in memory`.

$$FC\_cycles = [3 + n_{iter} \times (2 + w_{out(fc)} + L) + w_{out(fc)}] \times t_{ck} \tag{4.39}$$

Table 4.8: Timing of the In-Memory architecture. The reference neural network model is in Figure 4.1

Operation	Time required	Value
Data acquisition	$(\phi + (n_{layers} - 1) \times \psi) \times t_{ck}$	$2 \times 1014 \times t_{ck}$
Max Pooling	$(3 + w^2_{out(pool)} \times (2 + w^2_{filter}) + 1) \times t_{ck}$	$1180 \times t_{ck}$
Convolutional layer		
Precharge binary values	$(3 + (w^2_{filter} + 1) \times w^2_{out}) \times t_{ck}$	$848 \times t_{ck}$
Perform convolution	$(c_{out} \times (w^2_{filter} + w^2_{out} \times (c_{in} + 2)) + c_{out} \times 3 + 1) \times t_{ck}$	$3085 \times t_{ck}$
Fully connected layer		
Perform computation	$((1 + n_{iter} \times (w_{out(fc)} + L + 2) + w_{out(fc)} + 2) \times t_{ck}$	$3055 \times t_{ck}$
Total		$10196 \times t_{ck}$

The main differences in terms of timing are located in the convolutional/fully connected computation in fact, thanks to the parallel In-Memory computation, time can be saved up to $\sim 2.46\times$ (considering the same clock period for both the architectures). By looking at the convolutional delay expressions:

$$\text{Delay } \underline{\text{OOM}}_{convolution} = (c_{out} \times w^2_{out} \times (w^2_{filter} + c_{in} + 2) + c_{out} \times 4 + 1) \times t_{ck} \tag{4.40}$$

$$\text{Delay In-Memory}_{convolution} = (c_{out} \times (w^2_{filter} + w^2_{out} \times (c_{in} + 2)) + c_{out} \times 3 + 1) \times t_{ck} \tag{4.41}$$

Since the data are computed directly in the memory array and w^2_{out} parallel pop-counting units are used, there is no need to fetch them from the memory and to pop-counts them one by one as OOM does. This allows to the In-Memory architecture to reduce the computational time by transforming the a part of the delay equation from $c_{out} \times w^2_{out} \times (w^2_{filter} + c_{in} + 2)$ to $c_{out} \times (w^2_{filter} + w^2_{out} \times (c_{in} + 2) + 1)$. In the fully connected layer instead, the gain is much more evident:

$$\text{Delay } \underline{\text{OOM}}_{FC} = [3 + n_{iter} \times (2 + w_{out(fc)} + w_{out(fc)} \times (L + 1)) + w_{out(fc)}] \times t_{ck} \tag{4.42}$$

$$\text{Delay In-Memory}_{FC} = ((1 + n_{iter} \times (w_{out(fc)} + L + 2) + w_{out(fc)} + 2) \times t_{ck} \tag{4.43}$$

The In-Memory architecture has a big advantages in terms of delay w.r.t. the OOM, because of the usage of multiple pop-counting units and XNOR gates for each couple binary weight/input. As already said, once the pop-counting procedure has finished, there is no need to save them into an external register file (`RF TMP POP`). In the In-Memory architecture, at the end of the algorithm, they are simply multiplexed by the last multiplexer in Figure 4.30. These motivations brings to a delay ratio of $\sim 4.54\times$ for the FC layer. In the following figures are reported the total required time in both OOM and In-Memory case, in order to verify the correctness of the computations:

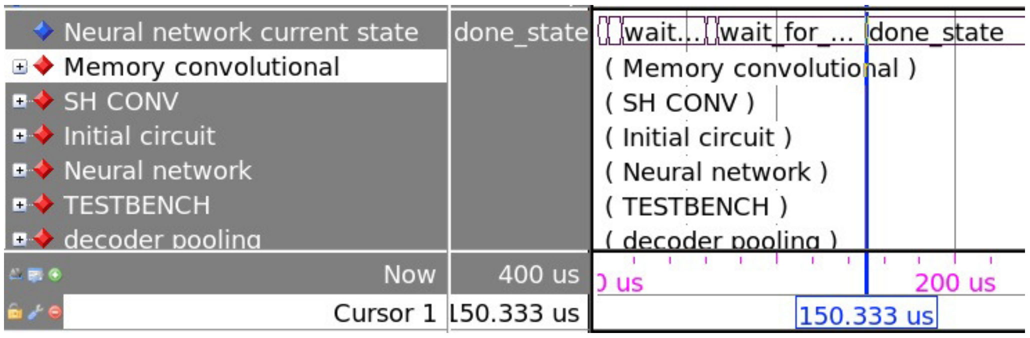


Figure 4.37: Computational delay of the OOM architecture with $t_{ck} = 5.5ns$

The delay ratio is given by:

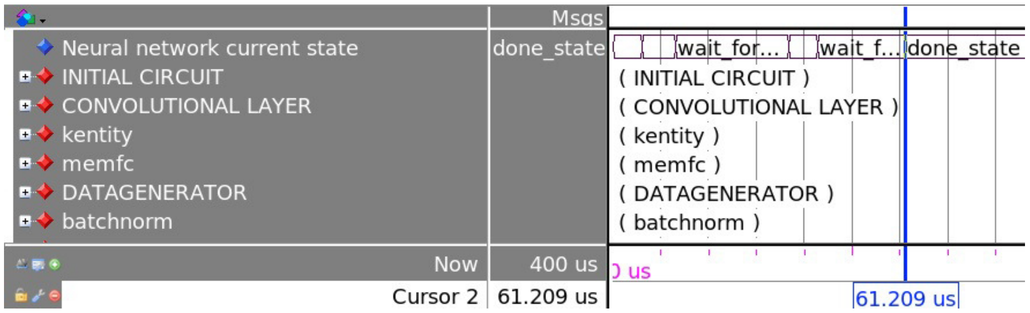


Figure 4.38: Computational delay of the In-Memory architecture with $t_{ck} = 5.5ns$

$$\text{Delay Ratio} = \frac{150.333\mu s}{61.209\mu s} \simeq 2.456 \tag{4.44}$$

There is a small difference w.r.t the computed value (2.46) and the simulated one (2.456): the reason is that some states of the `neural network`'s control unit are not considered in the computation (such as reset state, idles, etc).

4.4.3 General cases

General cases are analyzed by sweeping the parameters of the network, in order to evaluate the timing ratio between the two alternatives (OOM/In-memory), considering the same clock period and the reference architecture in Figure 4.1. The accuracy is evaluated in all the cases, with batch size of 100 and 5 epochs training, in order to see what is the impact of the different choices on the architecture's precision:

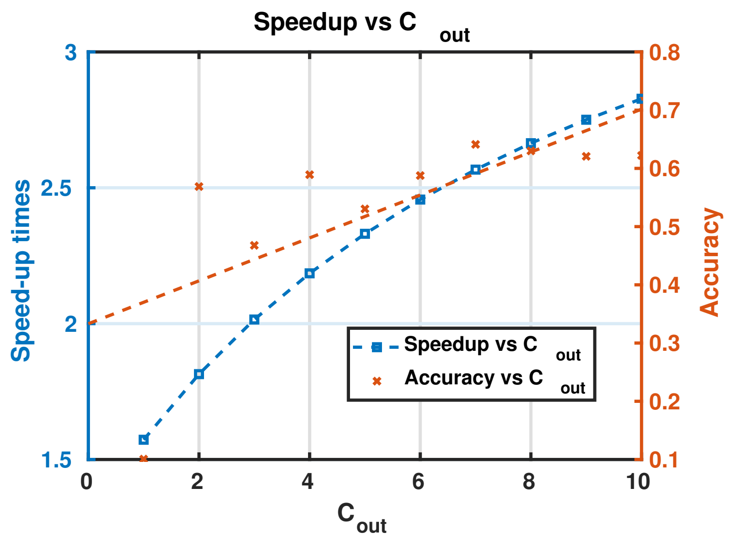


Figure 4.39: Speedup vs C_{out}: higher number of C_{out} increases the time ratio, but the complexity of the architecture is badly influenced (higher number of parameters required)

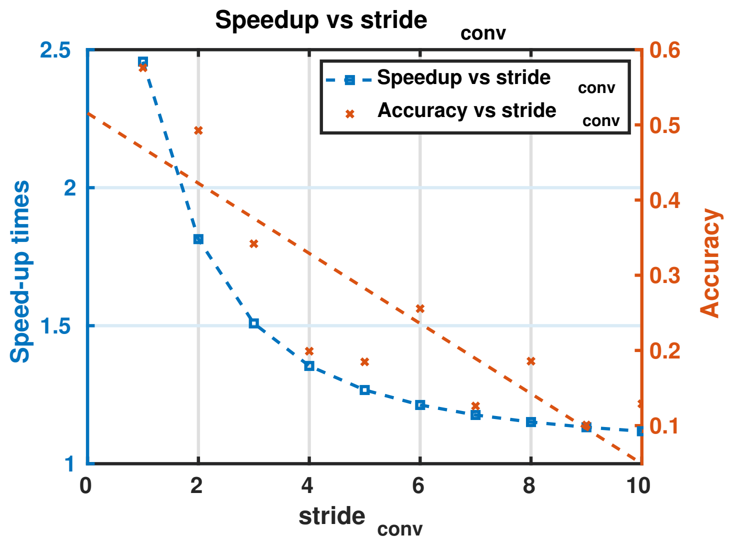


Figure 4.40: Speedup vs stride conv: the consequence of increasing the stride are worse accuracy and speedup, but the complexity of the network decreases.

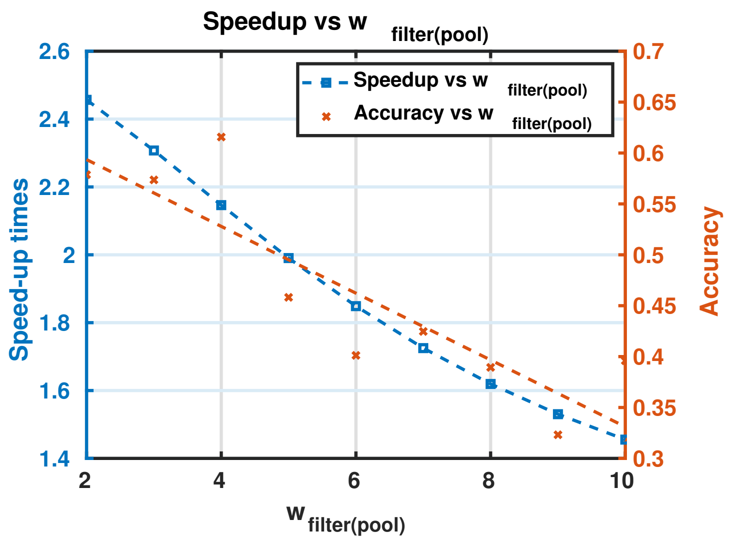


Figure 4.41: Speedup vs $w_{filter(pool)}$: speedup ratio decreases, but the accuracy is worse since the higher is the $w_{filter(pool)}$, the lower is the input quality image.

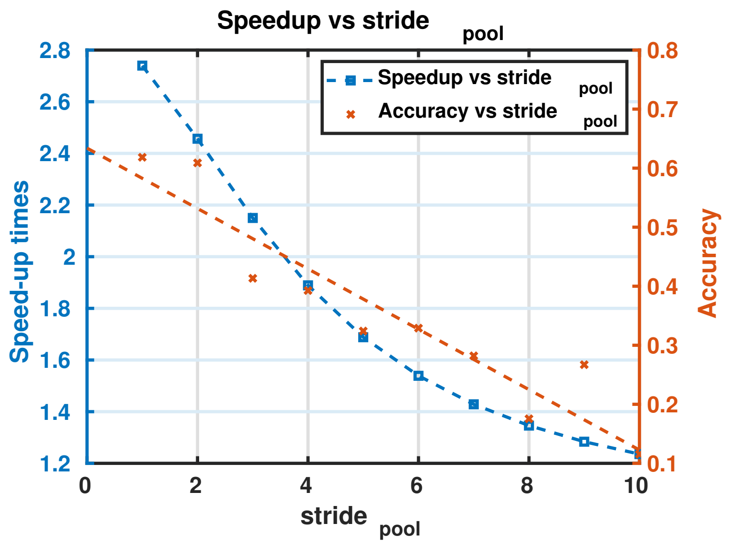


Figure 4.42: Speedup vs stride pool: speedup ratio decreases, but the accuracy is worse since an higher stride implies bad quality input image.

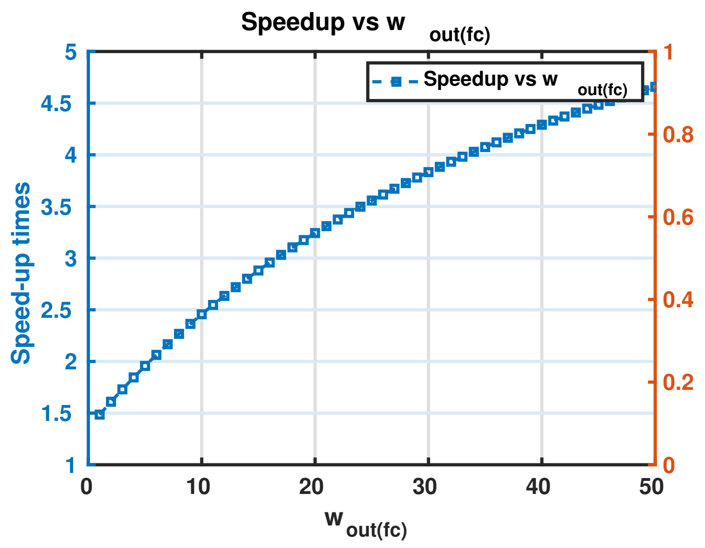


Figure 4.43: Speedup vs $w_{out(fc)}$: the higher is better, but in the case reported in Figure 4.1, no more than 10 outputs are used. If the neural network is structured with more than one fully connected layer, this brings some advantages.

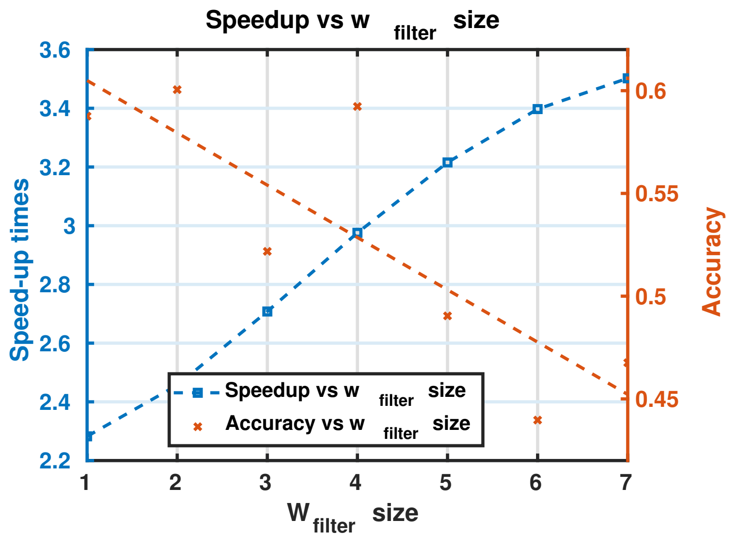


Figure 4.44: Speedup vs $w_{filter(conv)}$: increasing $w_{filter(conv)}$ also the speedup increases, but the accuracy is degraded.

Regarding the total delay of the architectures, it can be demonstrated that the delay of the OOM case is always higher than the In-Memory alternative, considering the following equations for the convolution algorithm from the timing computation:

$$\begin{aligned} Delay_{OOM(convolution)} = {} & [3 + w_{out}^2 \times (w_{filter}^2 + 1) + \\ & + (c_{out} \times w_{out}^2 \times (w_{filter}^2 + c_{in} + 2) + c_{out} \times 4 + 1)] \times t_{ck} \end{aligned} \tag{4.45}$$

$$\begin{aligned} Delay_{In-Memory(convolution)} = {} & [3 + w_{out}^2 \times (w_{filter}^2 + 1) + \\ & + (c_{out} \times (w_{filter}^2 + w_{out}^2 \times (c_{in} + 2)) + c_{out} \times 3 + 1)] \times t_{ck} \end{aligned} \tag{4.46}$$

By imposing the $Delay_{In-Memory(convolution)}$ equation less than $Delay_{OOM(convolution)}$:

$$\begin{aligned} Delay_{In-Memory(convolution)} &< Delay_{OOM(convolution)} \\ w_{filter}^2 + w_{out}^2 \times (c_{in} + 2) + 3 &< w_{out}^2 \times (w_{filter}^2 + c_{in} + 2) + 4 \\ w_{filter}^2 + 3 &< w_{out}^2 \times w_{filter}^2 + 4 \\ w_{filter}^2 + \cancel{-1} &< w_{out}^2 \times w_{filter}^2 \\ 1 &< w_{out}^2 \end{aligned} \tag{4.47}$$

By neglecting the -1 in the equation, it is demonstrated that $Delay_{In-Memory(convolution)}$ is always less than $Delay_{OOM(convolution)}$. By performing the same steps for the fully connected computational delay:

$$\begin{aligned} & Delay_{OOM(FC)} = [1 + n_{iter} \times (w_{out(fc)} + 3 + w_{out(fc)} \times (L + 1)) + \\ & w_{out(fc)} + 2] \times t_{ck} \end{aligned} \tag{4.48}$$

$$Delay_{In-Memory(FC)} = ((1 + n_{iter} \times (w_{out(fc)} + L + 2) + w_{out(fc)} + 2) \times t_{ck} \tag{4.49}$$

Imposing the inequality, it results always verified:

$$\begin{aligned} Delay_{In-Memory(FC)} &< Delay_{OOM(FC)} \\ \cancel{w_{out(fc)}} + L + 2 &< \cancel{w_{out(fc)}} + 3 + w_{out(fc)}(L+1) \\ L &< 1 + w_{out(fc)}(L+1) \\ \frac{L-1}{L+1} &< w_{out(fc)} \end{aligned} \tag{4.50}$$

$w_{out(fc)}$ is always greater-equal than 1 and, since the ratio $\frac{L-1}{L+1}$ is always less than 1, the equation is verified. The following plots represent both convolutional/fully connected delay ratios between the OOM and In-Memory architectures. Delay ratios have been evaluated by sweeping two parameters per time. Starting from the convolutional computation, the X-Y meshes used are c_{in} - w_{filter} (fixed parameters are $w_{in} = 28$, $stride = 1$ and $c_{out} = 1$); c_{in} - c_{out} ($w_{in} = 28$, $w_{filter} = 2$, $stride = 1$); w_{filter} - c_{out} ($w_{in} = 28$, $stride = 1$, $c_{in} = 1$); w_{in} - w_{filter} ($c_{in} = 1$, $c_{out} = 1$, $stride = 1$). Regarding the fully connected computation, the dependency of the delay ratio has been evaluated respect to $w_{out(fc)}$ and n_{iter}, considering $number\_of\_fc\_parameters = 1000$. n_{iter} is chosen with the `divisors` function in MATLAB, that generates the following vector:

$$n_{iter} = \begin{pmatrix} 1 & 2 & 4 & 5 & 8 & 10 & 20 & 25 & 40 & 50 & 100 & 125 & 200 & 250 & 500 & 1000 \end{pmatrix} \tag{4.51}$$

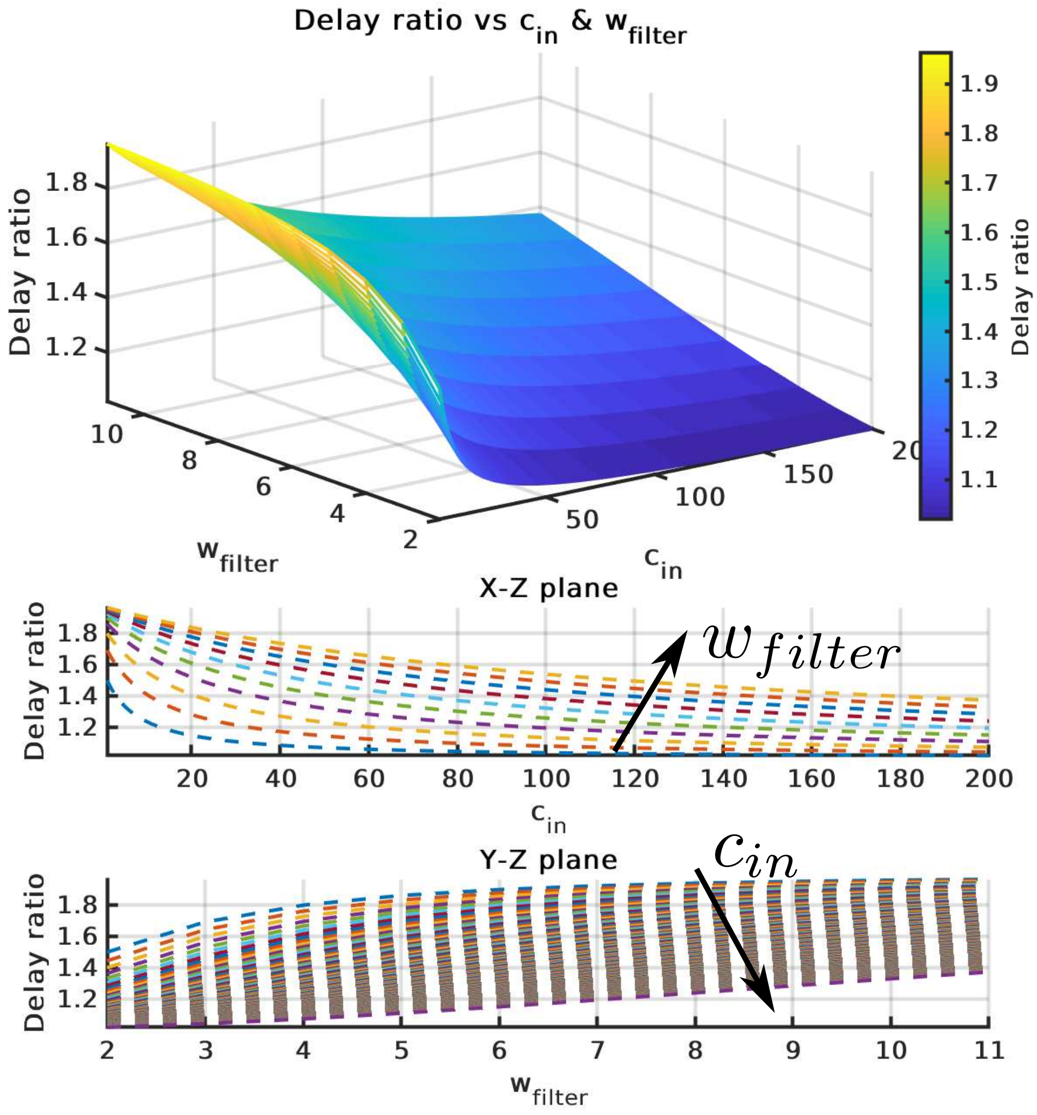


Figure 4.45: c_{in} - w_{filter} plot for a convolutional computation. By increasing the c_{in}, the delay ratio decreases, because by looking at Equation 4.45 and Equation 4.46, the ratio tends towards 1 for high values of c_{in}. Delay ratio increases with higher values of w_{filter}.

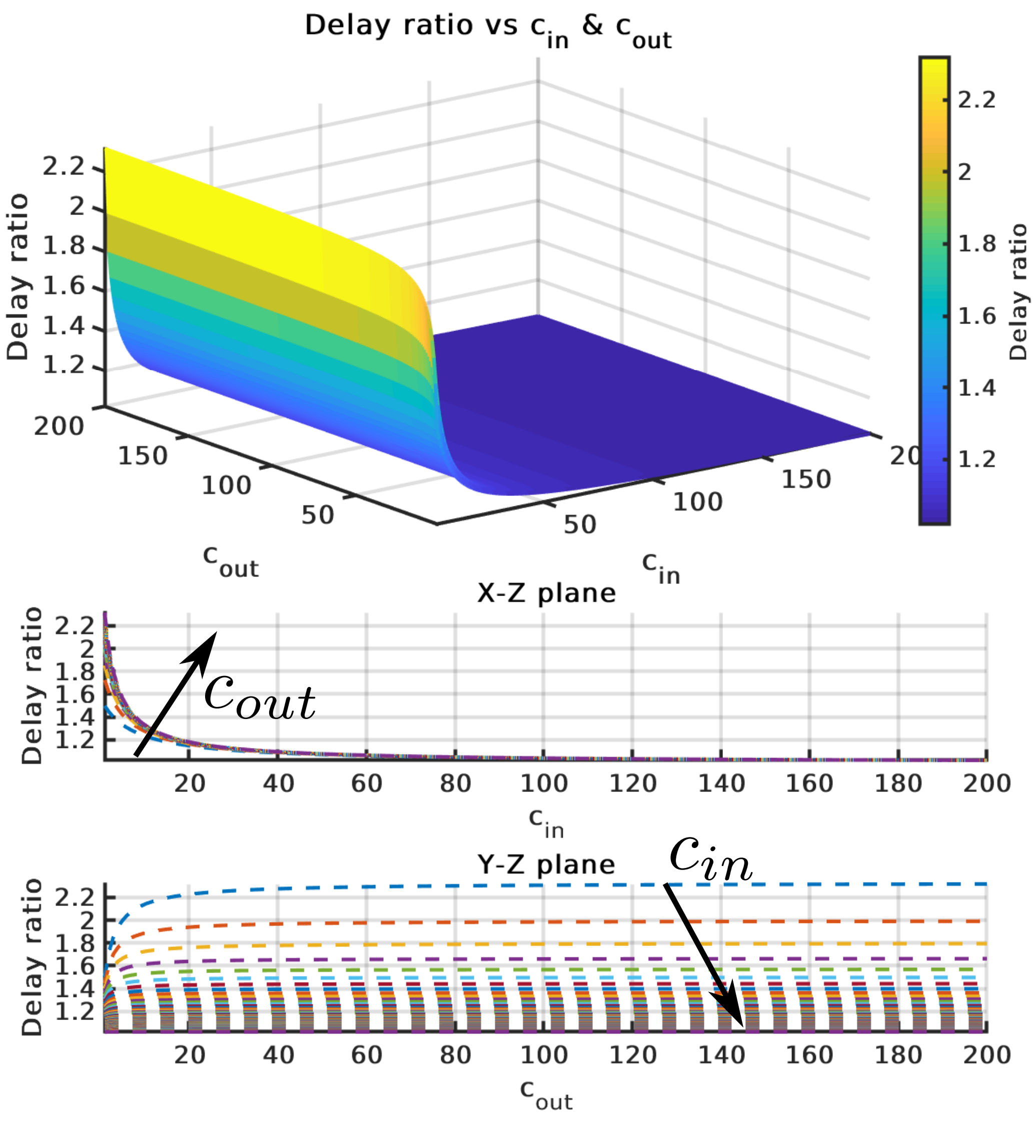


Figure 4.46: c_{in} - c_{out} plot for a convolutional computation. For c_{in}, the same considerations made in Figure 4.45 are valid. Regarding c_{out}, by increasing it the delay ratio slowly rises as a logarithm-like function until it reaches a saturation, since by performing the limit of the Delay ratio function for $c_{out} \to \infty$, the result is a constant.

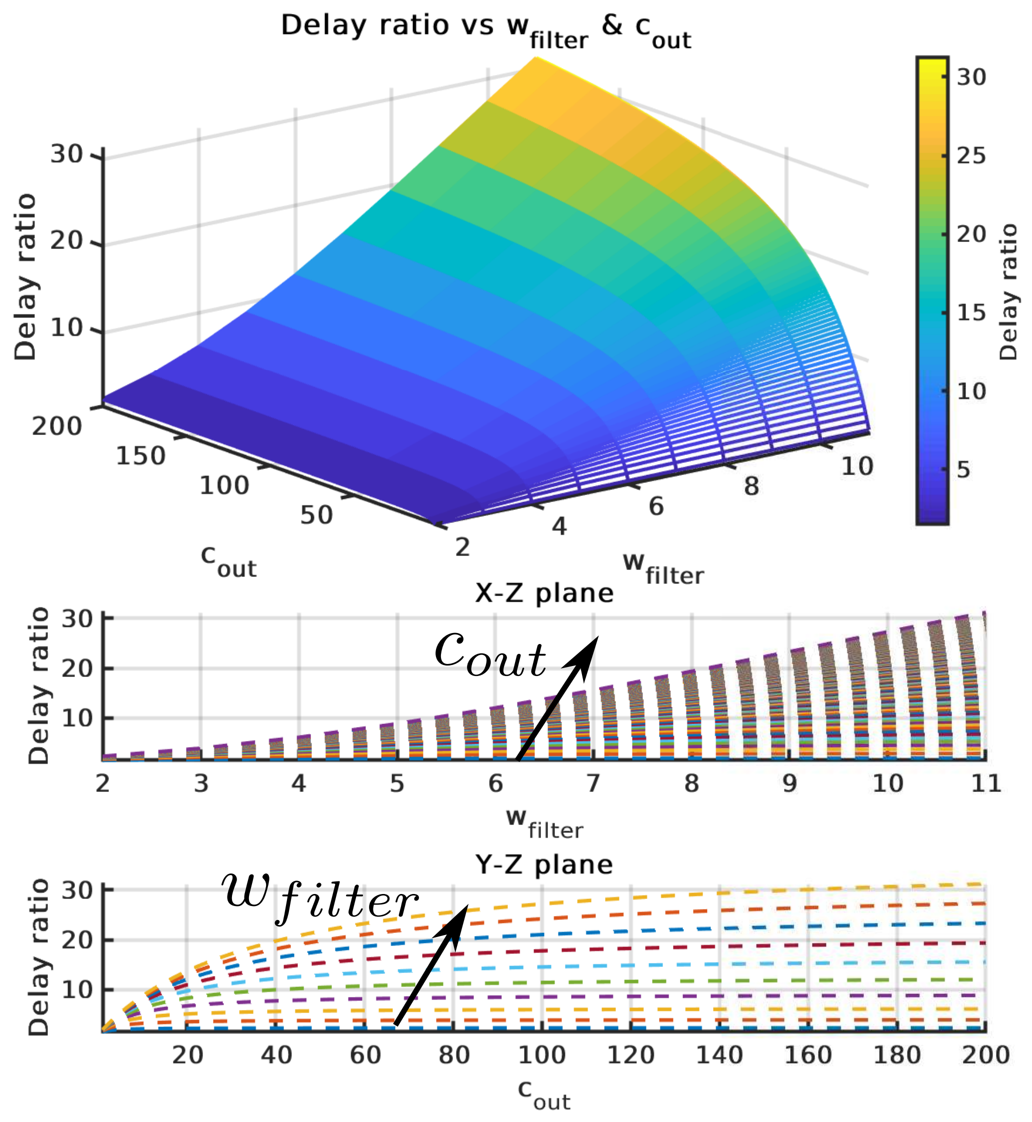


Figure 4.47: w_{filter} - c_{out} plot for a convolutional computation. The big advantage of the In-Memory architecture in terms of delay respect to OOM one, is obtained with high values of w_{filter} and c_{out}. Considering for example the first layer of AlexNet, the total number of OFMAPs are 96 with $w_{filter} = 11$ and the delay ratio will be $\sim 27\times$.

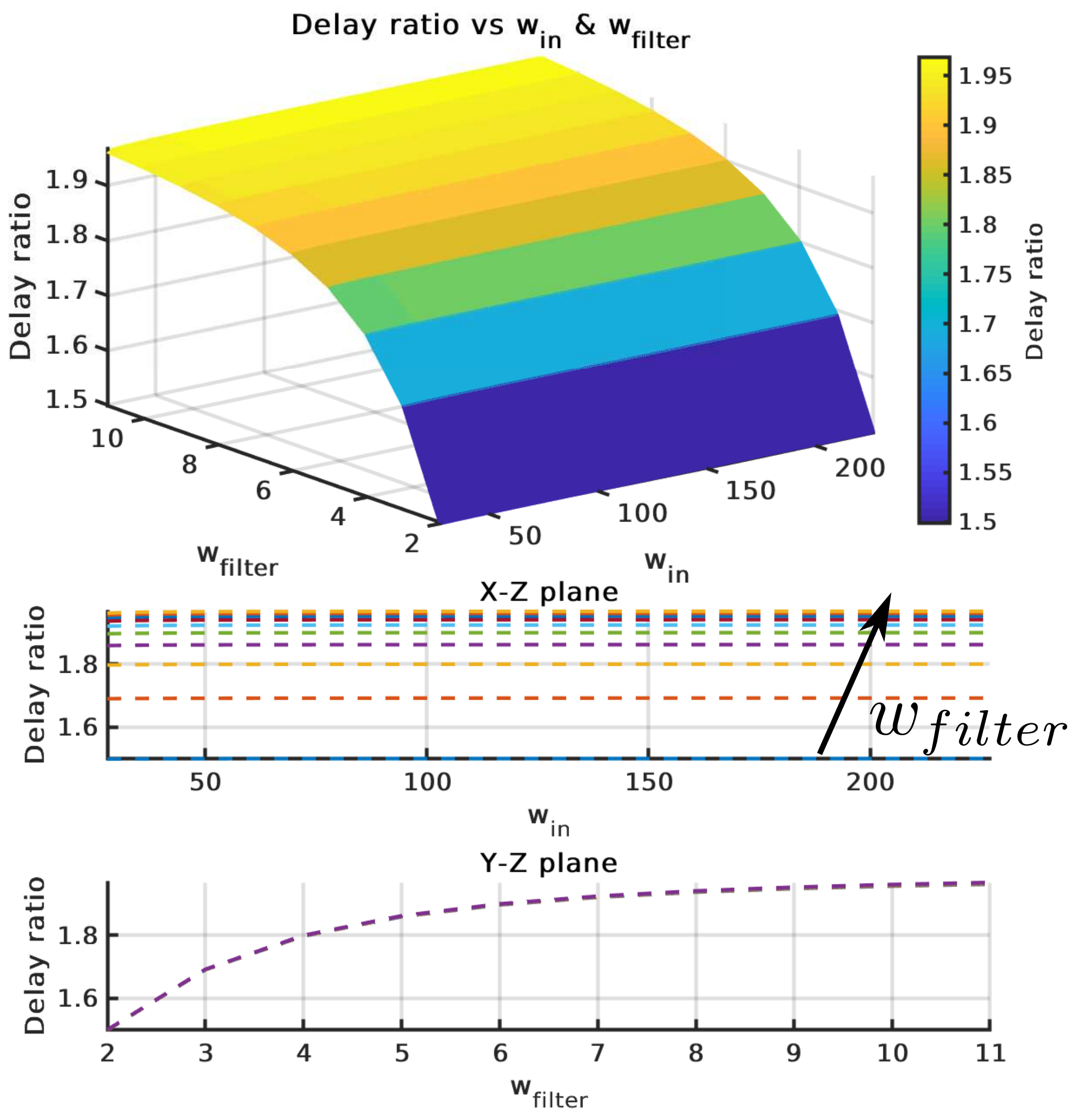


Figure 4.48: w_{in} - w_{filter} plot for a convolutional computation. By increasing w_{in}, the delay ratio remains approximately the same, while w_{filter} dependency is the same described in Figure 4.47

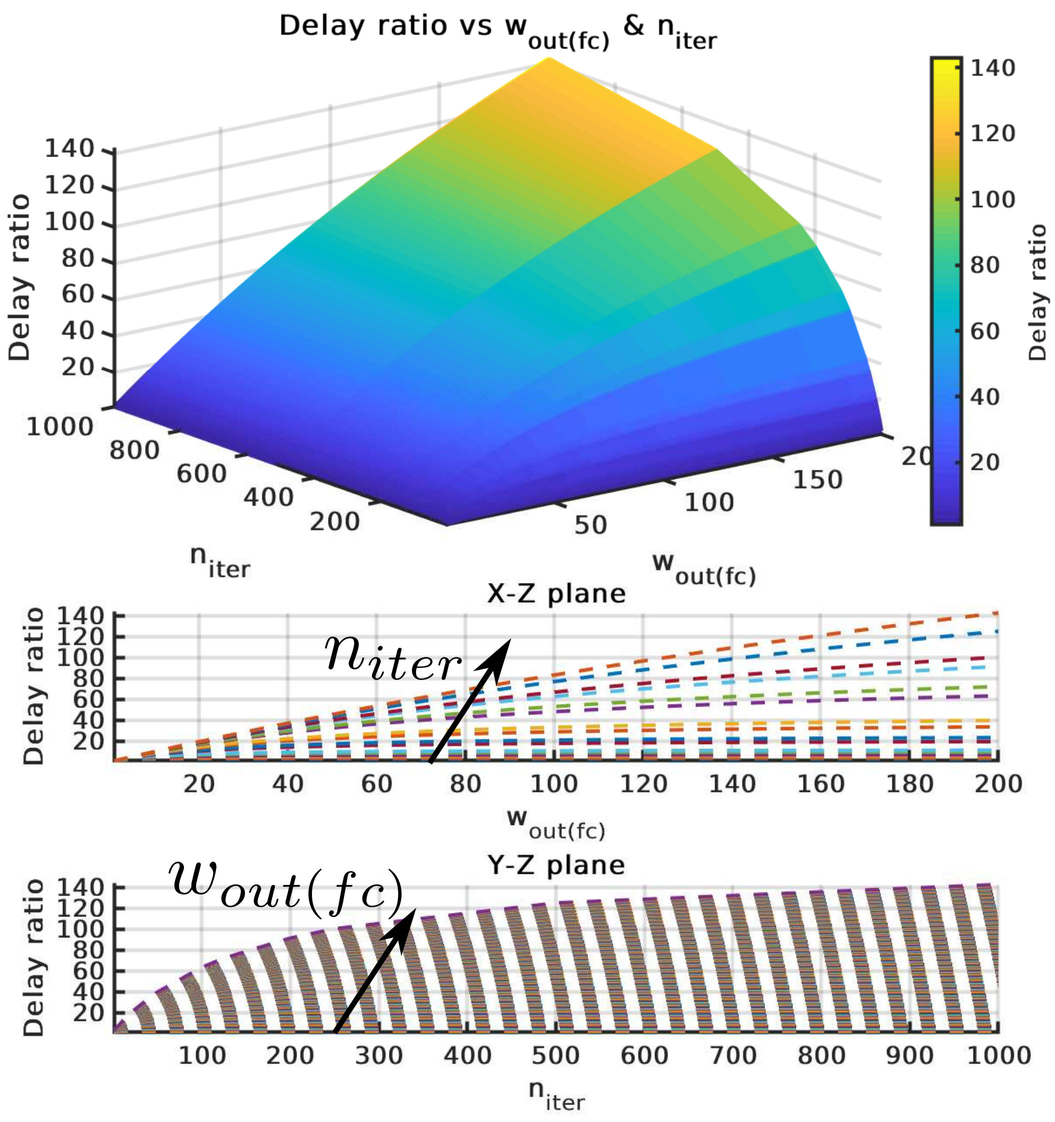


Figure 4.49: $w_{out(fc)}$-n_{iter} plot, considering a fully connected layer. By increasing both the quantities brings relevant benefits in terms of Delay ratio. In particular it is demonstrated that with high values of n_{iter}, the In-Memory architecture takes advantages of a more scheduled FC computation (Figure 4.16): this is a very important result, since high n_{iter} implies a smaller array, since $W \geq \frac{number\_of\_fc\_parameters}{n_{iter}} = L$, allowing to further reduce power consumption/area/energy consumption of the In-Memory architecture.

Considerations

From the previous plots, the following considerations can be made for convolutional computation:

1. By increasing w_{filter}, the delay ratio increases, taking advantages of a more complex network (such as AlexNet, in which the first layer has a kernel size of 11x11). In combination of an higher number of output channels (c_{out}), this behavior is much more evident (Figure 4.47);

2. An higher value of w_{in} (which is translated in $w_{out} = \frac{w_{in} - w_{filter}}{stride} + 1$), do not brings relevant advantages in terms of delay ratio, since it has a constant behavior;

3. Higher values of c_{in} reduce the delay ratio, because of the similar delay expressions.

Regarding the FC layer, an higher value of n_{iter} allows to increase the delay ratio and also the power consumption. The In-Memory architecture takes relevant advantages when fully connected computation is performed.

4.5 Choosing the number of bits (n_bit)

An important parameter is the number of bits for the fixed point implementation. In order to properly choose this value, a fixed-point neural network model has been implemented in MATLAB (discussed in chapter 5). Following Figure 4.1, the MATLAB code takes the pre-trained parameters, input images and labels from Python and computes the accuracy for each combination [n_bit,n_bit_fractional] simply considering:

$$\text{Accuracy} = \frac{\text{Score}}{\text{\#Mnist images}} \tag{4.52}$$

The total number of bits dedicated to the integer part are fixed and contrained to the fractional ones, and to have a full precision computation, these have to be at least equal to $floor(log_2(number\_of\_fc\_parameters) + 1)$, since the pop-counting part counts all the inputs:

$$\texttt{n\_bit\_integer} = \texttt{n\_bit} - \texttt{n\_bit\_fractional}$$

$$\texttt{n\_bit\_integer} = floor(log_2(number\_of\_fc\_parameters) + 1) = 11$$

The analysis is focused on the accuracy which derives from a different number of fractional bits. The result is reported in Figure 4.50:

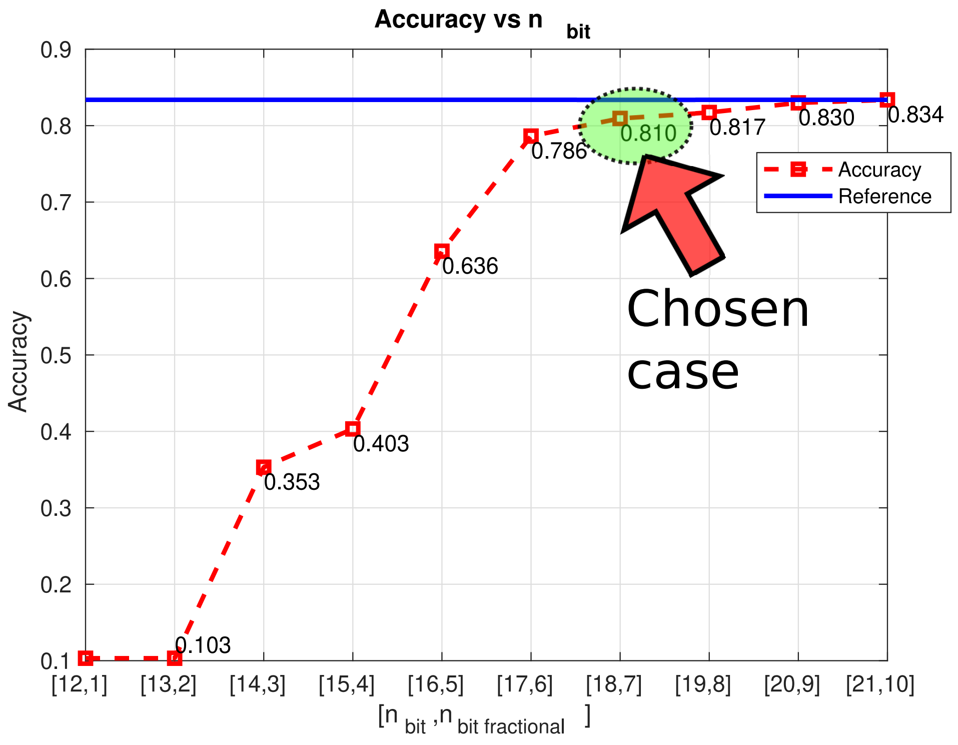


Figure 4.50: Accuracy vs number of bits. The total number of images tested are 10000. The reference accuracy is set to 0.8338 from section 3.2.1

If a new neural network model is considered, the n_{bit} analysis can be performed again in order to find the best trade-off between accuracy-complexity.

Chapter 5

Verification

In order to check the results given by the VHDL fixed point model, the verification steps include:

1. The realization of the Python program, from which the results of the single layers have been extracted;
2. Design of MATLAB floating point model and validity verification by comparing the Python results and the MATLAB ones;
3. Derivation of a fixed-point MATLAB model;
4. Comparison between the VHDL results and MATLAB ones.

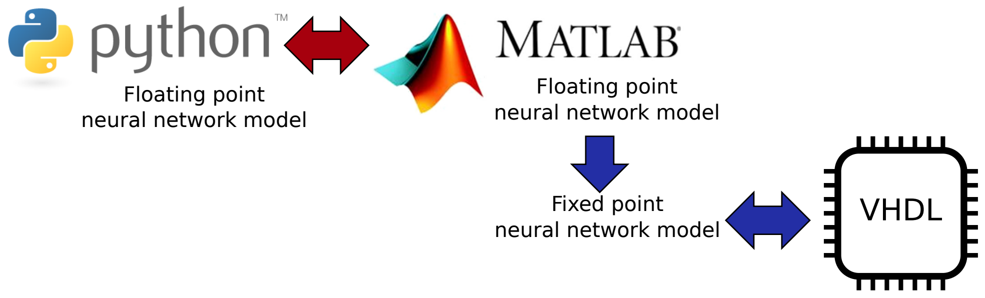


Figure 5.1: Verification flow

In order to convert a floating point value to a fixed point, the following formula has been used:

$$quantization(x) = fix\left(\frac{x}{2^{-n\_bit\_fractional}}\right) \tag{5.1}$$

Where $fix(y)$ rounds toward 0 both positive and negative results. For a multiplication result, the steps to compute its fixed-point equivalent can be derived by the following example with A = 0.25, B = 0.125 and `n_bit_fractional` = 4:

- Full precision multiplication gives 0.03125:

```
A = 00.0100 x
B = 00.0010 =
-------------
 00|00.0000|1000
```

- The result is truncated after the 4th fractional bit, giving 0 as final result. From a software point-of-view, the corresponding operation is $floor(x)$, so:

```
tmp = X.*2^(n_bit_fractional);
tmp = floor(tmp);
quantized = tmp.*2^(-n_bit_fractional);
```

Now the entire MATLAB program is reported and explained:

1. Loading of the the Python trained parameters by using `readNPY` function, that allows MATLAB to read Numpy vectors:

```
Image = readNPY('./Image.npy');              % Input images: 10000
     testing images
Ws_Conv = readNPY('./Parameters/Ws_1.npy'); % Convolutional
     layer's weights
Ws_FC = readNPY('./Parameters/Ws_2.npy')';  % Fully connected
     layer's weights
mu = readNPY('./Parameters/mu.npy');         % Mean for batchnorm
sigma = readNPY('./Parameters/sigma.npy');  % Std deviation for
     batchnorm
scale = readNPY('./Parameters/scale.npy');  % Scale for batchnorm
offset = readNPY('./Parameters/offset.npy');% Offset for batchnorm
labels = readNPY('./Parameters/labels.npy');% Labels for the
     accuracy
```

2. Saving the extracted parameters for the VHDL simulation:

```
path = '../VHDL_MODEL/INPUT_PARAMETERS_VHDL/';
s = strcat(path,'Ws_Conv.txt');
s1 = strcat(path,'*.txt');
delete(s1)
sz = size(Weights_NN);
for i=1:sz(4)
    vect = Ws_Conv(:,:,i)';
    mat = vect(:)';
    dlmwrite(s,mat,'delimiter','\t','precision','%.6f','-append');
end
A = scale.*(sqrt(sigma).^(-1));
B = -mu.*(sqrt(sigma).^(-1)).*scale + offset;
dlmwrite(strcat(path,'Ws_FC.txt'),Ws_FC,...
'delimiter','\t','precision','%.6f','-append');
dlmwrite(strcat(path,'Aone.txt'),A,...
'delimiter','\t','precision','%.6f','-append');
dlmwrite(strcat(path,'Bone.txt'),B,...
'delimiter','\t','precision','%.6f','-append');
dlmwrite(strcat(path,'Image.txt'),...
Image(:,:,1),'delimiter','\t','precision','%.6f','-append');
```

3. Reading of Python results:

```
conv_out = readNPY('./Parameters/conv.npy');          %
     Convolutional layer output
fully = readNPY('./Parameters/fully.npy');            % Fully
     connected layer output
batch_norm_output = readNPY('./Parameters/batch.npy');% Batch
     normalization output
ReLU_out = readNPY('./Parameters/ReLU.npy');          % ReLU
     output
```

4. Setting the fixed-point number of bits by using two global variables (`n_bit` and `n_bit_fractional` respectively) with `SetGlobals(18,7)`. By choosing `SetGlobals(0,0)`, the computation is performed in floating point representation;

```
SetGlobals(18,7); % 18 = n_bit; 7 = n_bit_fractional
```

5. Parameters' quantization. `Ws_FC` are not quantized since only the sign is taken:

```
if to_quantize == 1
    QntImage = quantization(Image);
    QntWS_Conv = quantization(Ws_Conv);
    QntA = quantization(A);
    QntB = quantization(B);
end
```

6. Neural network's realization:

```
XNOR_NET = 1; % Sets the computational model to the XNOR_NET one.
%% Max pooling layer %%
pool = Max_pooling_layer(QntImage,2,2); % The first parameter is
     the w_filter size, while the second the stride.
%% Convolutional layer %%
[K_conv,alpha_conv,conv_xnor] =
     Convolutional_layer(pool,QntWS_Conv,1,1,XNOR_NET,0); %
     (Input argument, Weights, Number of input channels, stride,
     XNOR_NET computational model, disable k computation).

%% Batch normalization layer %%
[conv_xnor,vectors] = Batchnorm(conv_xnor,A,B); % (Input
     argument, A,B constants)

%% ReLU %%
conv_xnor = max(0,conv_xnor);
%% Flatten layer %%
input_fc = flatten_layer(conv_xnor);

%% Fully connected %%
disable_k = 1;
[K,alpha,output_full] =
     Fully_connected_layer(input_fc,XNOR_NET,Ws_FC,disable_k);
```

7. To perfom the validation of the results, the absolute difference is taken between the Python/Floating-Point MATLAB and Fixed-Point MATLAB/VHDL. If

any of the difference values is higher than a certain threshold, the computational model is considered wrong.

5.1 VHDL's output

To ease the verification procedure, the results have been printed on a file in matricial form by using a `data save`. It is reported a toy example of convolutional/fully connected outputs with 5x5 input image, 2x2 kernel sizes, stride=1, 3 output channels and 4 output neurons. The last results are produced by the fully connected layer and the maximum value represents the final classification result.

```
convolution.txt
1.250000e-01 1.250000e-01 1.250000e-01 1.250000e-01
1.250000e-01 1.250000e-01 1.250000e-01 1.250000e-01
1.250000e-01 1.250000e-01 1.250000e-01 1.250000e-01
1.250000e-01 1.250000e-01 1.250000e-01 1.250000e-01

0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00

3.281250e-01 3.281250e-01 3.281250e-01 3.281250e-01
3.281250e-01 3.281250e-01 3.281250e-01 3.281250e-01
3.281250e-01 3.281250e-01 3.281250e-01 3.281250e-01
3.281250e-01 3.281250e-01 3.281250e-01 3.281250e-01

-5.200000e+01
-1.200000e+02
-3.200000e+01
-6.200000e+01
```

MATLAB just read these values with the directive `dlmread` and compares with its approximated fixed-point computation.

5.2 MATLAB's output

First of all the, floating point model is verified and the output of all layers is compared with the Python's ones. For demonstration purposes, it has been reported only the convolutional layer's first result, after performing Batch normalization and ReLU:

```
if(validation_python==1)
    diff = abs(conv_xnor-conv_out); % conv_xnor is the output given by
         the MATLAB model, while conv_out the python's one
    fprintf('Maximum Convolution difference between python-MATLAB
         is:');
    max(diff(:))
end
```

```
Maximum Convolution difference between python-MATLAB is:
ans =

  single

  3.3930e-07
```

As it is possible to see, a very small difference between the two models exists due to the saving procedure precision, but it can be neglected.

MATLAB FP model's first ReLU output

$$\begin{pmatrix}
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.385 & 0.127 & 0.127 & 0.211 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0 & 0 & 0 & 0.127 & 0.127 & 0.127 & 0.127 & 0.481 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.481 & 0.127 & 0 & 0 & 0 & 0 & 0 & 0 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.151 & 0.127 & 0.127 & 0 & 0 & 0 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0 & 0 & 0.243 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.303 & 0 & 0 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.154 & 0 & 0 & 0 & 0.208 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0 & 0 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.313 & 0 & 0 & 0 & 0.176 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0 & 0 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0 & 0 & 0.2 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127
\end{pmatrix} \tag{5.2}$$

Pythons model's first ReLU output

$$\begin{pmatrix}
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.385 & 0.127 & 0.127 & 0.211 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0 & 0 & 0 & 0.127 & 0.127 & 0.127 & 0.127 & 0.481 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.481 & 0.127 & 0 & 0 & 0 & 0 & 0 & 0 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.151 & 0.127 & 0.127 & 0 & 0 & 0 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0 & 0 & 0.243 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.303 & 0 & 0 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.154 & 0 & 0 & 0 & 0.208 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0 & 0 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.313 & 0 & 0 & 0 & 0.176 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0 & 0 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127 \\
0.127 & 0.127 & 0.127 & 0.127 & 0.127 & 0 & 0 & 0.2 & 0.127 & 0.127 & 0.127 & 0.127 & 0.127
\end{pmatrix} \tag{5.3}$$

Considering the VHDL results, the MATLAB model is switched to fixed-point computation and the outputs are compared:

MATLAB Fixed point model's first ReLU output

$$\begin{pmatrix}
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.375 & 0.125 & 0.125 & 0.195 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0 & 0 & 0 & 0.125 & 0.125 & 0.125 & 0.125 & 0.461 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.461 & 0.125 & 0 & 0 & 0 & 0 & 0 & 0 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.133 & 0.125 & 0.125 & 0 & 0 & 0 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0 & 0 & 0.227 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.281 & 0 & 0 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.133 & 0 & 0 & 0 & 0.195 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0 & 0 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.289 & 0 & 0 & 0 & 0.156 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0 & 0 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0 & 0 & 0.18 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125
\end{pmatrix} \tag{5.4}$$

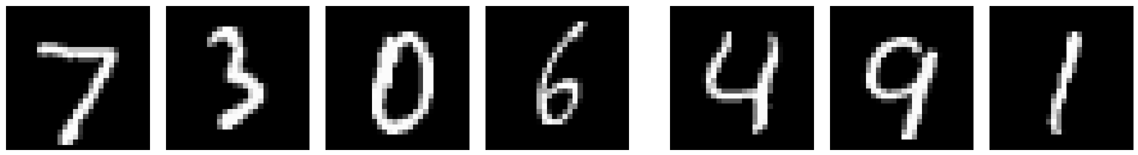
VHDL model's first ReLU output

$$\begin{pmatrix}
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.375 & 0.125 & 0.125 & 0.195 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0 & 0 & 0 & 0.125 & 0.125 & 0.125 & 0.125 & 0.461 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.461 & 0.125 & 0 & 0 & 0 & 0 & 0 & 0 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.133 & 0.125 & 0.125 & 0 & 0 & 0 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0 & 0 & 0.227 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.281 & 0 & 0 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.133 & 0 & 0 & 0 & 0.195 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0 & 0 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.289 & 0 & 0 & 0 & 0.156 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0 & 0 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125 \\
0.125 & 0.125 & 0.125 & 0.125 & 0.125 & 0 & 0 & 0.18 & 0.125 & 0.125 & 0.125 & 0.125 & 0.125
\end{pmatrix} \tag{5.5}$$

MATLAB checks the values given by the VHDL and, if the model is correct, should report the following messages:

```
First convolution (1) is correct!
First convolution (2) is correct!
First convolution (3) is correct!
First convolution (4) is correct!
First convolution (5) is correct!
First convolution (6) is correct!
Output FC results are correct!!
```

By comparing the results given by the fixed point model and the floating point one, the difference is not so evident, since the number of layers is reduced and the approximation imposed by the fixed point representation does not influences so much the calculations. Trying now to recognize some inputs, the resulting fully connected VHDL outputs are the following. The maximum value out of 10 is the classification result: the first one refers to ”0” and the last one to ”9”.



```
-6.400000e+01 -7.800000e+01  2.000000e+01 -5.600000e+01    -4.600000e+01 -4.000000e+01 -6.200000e+01
-6.400000e+01 -2.600000e+01 -1.040000e+02 -2.800000e+01    -5.400000e+01 -9.600000e+01  1.800000e+01
-4.400000e+01 -2.200000e+01 -6.000000e+01 -2.800000e+01    -3.800000e+01 -3.200000e+01 -4.200000e+01
-3.800000e+01  4.000000e+00 -2.600000e+01 -3.400000e+01    -8.000000e+01 -8.200000e+01 -2.800000e+01
-3.400000e+01 -3.200000e+01 -7.000000e+01 -3.000000e+01     1.200000e+01 -2.200000e+01 -4.400000e+01
-3.400000e+01 -2.000000e+01 -6.000000e+00 -2.600000e+01    -4.800000e+01 -7.000000e+01 -4.800000e+01
-8.000000e+01 -1.400000e+01 -3.600000e+01  1.600000e+01    -4.600000e+01 -4.400000e+01 -4.200000e+01
4.600000e+01  -3.200000e+01 -7.000000e+01 -4.600000e+01    -2.400000e+01 -1.400000e+01 -2.800000e+01
-7.400000e+01 -5.200000e+01 -3.000000e+01 -4.200000e+01    -5.600000e+01 -5.000000e+01 -5.200000e+01
-4.000000e+01 -4.600000e+01 -4.400000e+01 -5.200000e+01    -3.400000e+01  0.000000e+00 -5.000000e+01
It’s 7!!      It’s 3!!      It’s 0!!      It’s 6!!         It’s 4!!      It’s 9!!      It’s 1!!
```

5.3 Other neural network models

To demonstrate the capability of the VHDL architecture to implement every kind of neural network, in the following part are proposed other neural network models.

5.3.1 MLP Implementation

The network structure is the following:

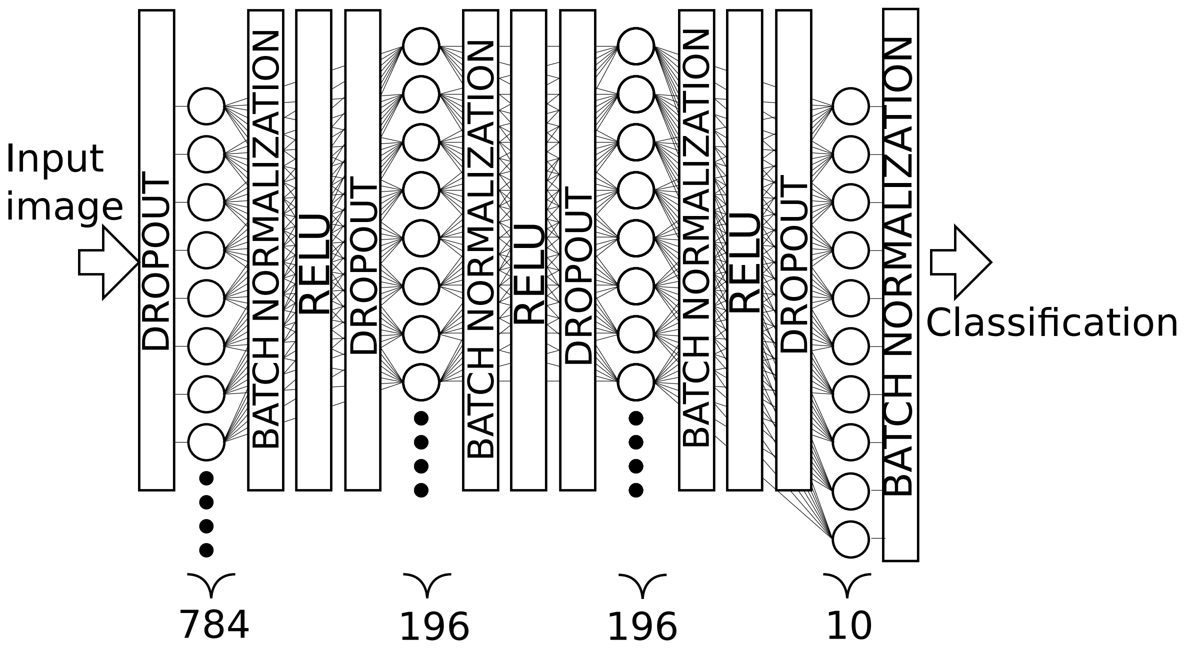


Figure 5.2: MLP model. The network has 15 layers and it is able to achieve $\sim 90\%$ of accuracy on MNIST dataset.

To implement an MLP, the variable and fixed parameters discussed in subsection 4.1.5, must be changed according to the new structure. For comparison purposes, these values are reported for both MLP and original neural network models.

```
-----------------FIXED PARAMETERS---------------------
  -----------------MLP NETWORK-----------------
constant w_out: integer:=14;
constant h_max: integer:=196;
constant w_in: integer:=28;
constant w_filter: integer:=4;
constant w_filter_s: integer:=2;
constant width_sram:integer:=14;
constant number_of_output_channels: integer:=1;
constant number_of_input_channels: integer:=1;
constant number_of_fc_parameters:integer:=798;
constant number_of_neurons_output: integer:=196;
constant input_image_size_x:integer:=28;
constant input_image_size_z:integer:=1;
constant flatten_x:integer:=196;
constant flatten_z:integer:=1;

-----------------FIXED PARAMETERS---------------------
 -----------------ORIGINAL NETWORK-----------------
constant w_out: integer:=14;
constant h_max: integer:=169;
constant w_in: integer:=28;
constant w_filter: integer:=4;
constant w_filter_s: integer:=2;
constant width_sram:integer:=6;
constant number_of_output_channels: integer:=6;
constant number_of_input_channels: integer:=1;
constant number_of_fc_parameters:integer:=1014;
constant number_of_neurons_output: integer:=10;
constant input_image_size_x:integer:=28;
constant input_image_size_z:integer:=1;
constant flatten_x:integer:=169;
constant flatten_z:integer:=6;
```

1. `h_max` has changed into 196 respect to 169 of the initial neural network model (Figure 4.1), since in this case there are 196 maximum output neurons. For this motivation, 196 rows of memory are required;

2. The `width_sram` chosen is 14, since it is a divider of both 784 (input layer size) and 196 (hidden layers size). The iterations required for input and hidden layers are given by:

$$n_{iter(MLP-Input)} = \frac{784}{14} = 56 \tag{5.6}$$

$$n_{iter(MLP-hidden)} = \frac{196}{14} = 14 \tag{5.7}$$

3. `number_of_output_channels`: in a MLP network, no output channels are required, since it is an operation performed on vectors instead of matrices;

4. `number_of_fc_parameters`: in the MLP case, the maximum input size of the fully connected layer is 784. For motivations related to the VHDL implementation, it is incremented to 798 to avoid addressing problems;

5. `flatten_x` and `flatten_z`: since there is only one `output register file`, the flattening procedure has to be performed only on 1 register file of size 196.

Considering now the variable parameters:

```
-----------------VARIABLE PARAMETERS---------------------
    -----------------MLP NETWORK-----------------
constant n_layers:integer:=4;
constant number_of_layers: std_logic_vector(n_layers-1 downto 0):= "0011";
constant conv_layer_size_x: int_vect(n_layers-1 downto 0):=(1,1,1,1);
constant conv_layer_size_x_pow:int_vect(n_layers-1 downto 0):=(1,1,1,1);
constant conv_layer_size_y: int_vect(n_layers-1 downto 0):=conv_layer_size_x;
constant conv_layer_size_z: int_vect(n_layers-1 downto 0):=(14,14,14,56);
constant kernel_size_xy_pow:int_vect(n_layers-1 downto 0):=(14,14,14,14);
constant kernel_size_xy: int_vect(n_layers-1 downto 0):=(1,1,1,1);
constant kernel_size_z: int_vect(n_layers-1 downto 0):=(1,1,1,1);
constant stride_sel_c: int_vect(n_layers-1 downto 0):=(1,1,1,1);
constant output_size_conv: int_vect(n_layers-1 downto 0):=(1,1,1,1);
constant output_size_conv_pow: int_vect(n_layers-1 downto 0):=(10,196,196,196);
-------------------------------------------------------------------------------
---------------POOLING-------------
constant pool_filter_size:int_vect(n_layers-1 downto 0):=(1,1,1,1);
constant pool_x_size:int_vect(n_layers-1 downto 0):=(1,1,1,1);
constant pool_out_size:int_vect(n_layers-1 downto 0):=(1,1,1,1);
constant pool_filter_size_s:int_vect(n_layers-1 downto 0):=(1,1,1,1);
constant pool_stride:int_vect(n_layers-1 downto 0):=(1,1,1,1);
constant pool_x_size_pow:int_vect(n_layers-1 downto 0):=(1,1,1,1);
constant pool_z_size:int_vect(n_layers-1 downto 0):=(1,1,1,1);
-------------------------------------------------------------------------------
-------------Layer types-------------
constant do_batch_layer: std_logic_vector(n_layers-1 downto 0):="1111";
constant do_pool_layer:std_logic_vector(n_layers-1 downto 0):="0000";
constant do_relu_layer: std_logic_vector(n_layers-1 downto 0):="0111";
constant fully_connected:std_logic_vector(n_layers-1 downto 0):="1111";

-----------------VARIABLE PARAMETERS---------------------
 -----------------ORIGINAL NETWORK-----------------
constant n_layers : integer := 2 ;
constant number_of_layers :  std_logic_vector ( n_layers-1 downto 0 ) := "01";
constant conv_layer_size_x :  int_vect ( n_layers-1 downto 0 ) := ( 1 ,14 );
constant conv_layer_size_x_pow : int_vect ( n_layers-1 downto 0 ) := ( 1 ,196 );
constant conv_layer_size_z :  int_vect ( n_layers-1 downto 0 ) := ( 169 ,1 );
constant kernel_size_xy_pow : int_vect ( n_layers-1 downto 0 ) := ( 6 ,4 );
constant kernel_size_xy :  int_vect ( n_layers-1 downto 0 ) := ( 1 ,2 );
constant kernel_size_z :  int_vect ( n_layers-1 downto 0 ) := ( 1 ,6 );
constant stride_sel_c :  int_vect ( n_layers-1 downto 0 ) := ( 1 ,1 );
constant output_size_conv :  int_vect ( n_layers-1 downto 0 ) := ( 1 ,13 );
constant output_size_conv_pow :  int_vect ( n_layers-1 downto 0 ) := ( 10 ,169 );
-------------
---------------POOLING-------------
constant pool_filter_size : int_vect ( n_layers-1 downto 0 ) := ( 1 ,4 );
constant pool_x_size : int_vect ( n_layers-1 downto 0 ) := ( 1 ,28 );
constant pool_out_size : int_vect ( n_layers-1 downto 0 ) := ( 1 ,196 );
constant pool_filter_size_s : int_vect ( n_layers-1 downto 0 ) := ( 1 ,2 );
constant pool_stride : int_vect ( n_layers-1 downto 0 ) := ( 1 ,2 );
constant pool_x_size_pow : int_vect ( n_layers-1 downto 0 ) := ( 1 ,784 );
constant pool_z_size : int_vect ( n_layers-1 downto 0 ) := ( 1 ,1 );
-------------
-------------Layer types-------------
constant do_batch_layer :  std_logic_vector ( n_layers-1 downto 0 ) := "01" ;
constant do_pool_layer : std_logic_vector ( n_layers-1 downto 0 ) := "01" ;
constant do_ReLU_layer :  std_logic_vector ( n_layers-1 downto 0 ) := "01" ;
constant fully_connected : std_logic_vector ( n_layers-1 downto 0 ) := "10" ;
```

1. The number of layers `n_layers` are four, since there are 4 fully connected computations: batch normalization and ReLU are considered inside the fully connected layer. Dropouts layers are not useful for classification routine, since are used during training to prevent overfitting;

2. The useful parameters of the convolutional part are `conv_layer_size_z`, `kernel_size_xy_pow` and `output_size_conv_pow`. `conv_layer_size_z`, as already said, indicates the number of iterations (n_{iter}) required by the considered layer to compute the fully connected output. `kernel_size_xy_pow` indicates the size of the FC input (L) (discussed in section 4.1.2): in this network, L is equal to 14 for all the cases. `output_size_conv_pow` indicates the number of output neurons of the considered layer;

3. Since pooling layer is not performed in the neural network model depicted in

Figure 5.2, the values specified in the vectors are not considered;

4. Batch normalization and Fully connected layer are always performed, consequently `do_batch` and `fully connected layer` are always enabled. Same considerations can be made for ReLU and max pooling.

Results verification

With the same approach described in Figure 5.1, the results are verified. Since the fully connected layers have vectors that flows into the architecture, the saving format is a vector, that is compared with the result provided by MATLAB. The vectors have a size of 196, 196, 196, 10 for the four layers respectively, which are separed by `---------------`. In the following part it is reported an output example with the number "7": as it is possible to see, the classification in output, which is given by the maximum number of the last fully connected layer, is 7 in both cases.

MATLAB Results

$$FC_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0.414 \\ 0.484 \\ 0.32 \\ 0 \\ \ldots \end{pmatrix} FC_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0.719 \\ 0 \\ 0.227 \\ 0.195 \\ \ldots \end{pmatrix} FC_3 = \begin{pmatrix} 0 \\ 0.703 \\ 0 \\ 1.04 \\ 0 \\ 0 \\ 0 \\ \ldots \end{pmatrix} FC_4 = \begin{pmatrix} -1.6 \\ -1.31 \\ -0.906 \\ -1.05 \\ -1.44 \\ -1.04 \\ -1.21 \\ 0.875 \\ -1.12 \\ -1.51 \end{pmatrix}$$

VHDL Results

```
0.000000e+00      0.000000e+00      0.000000e+00      -1.601562e+00
0.000000e+00      0.000000e+00      7.031250e-01      -1.312500e+00
0.000000e+00      0.000000e+00      0.000000e+00      -9.062500e-01
4.140625e-01      7.187500e-01      1.039062e+00      -1.046875e+00
4.843750e-01      0.000000e+00      0.000000e+00      -1.437500e+00
3.203125e-01      2.265625e-01      0.000000e+00      -1.039062e+00
0.000000e+00      1.953125e-01      0.000000e+00      -1.210938e+00
.....             .....             ....               8.750000e-01
------------      ------------      ------------      -1.125000e+00
                                                      -1.507812e+00
```

The output of the MATLAB program is the following:

```
(1) fully connected layer is correct!
(2) fully connected layer is correct!
(3) fully connected layer is correct!
Classification is correct!
```

5.3.2 Fashion-MNIST neural network model

To validate the VHDL implementation with another CNN network and Dataset, the following neural network model depicted in Figure 5.4 has been implemented with Fashion-MNIST. This is a dataset containing 60000 greyscale images of 28x28 representing 10 classification classes, that are identified by the following one-hot positions:

1. T-shirt/top;
2. Trouser;
3. Pullover;
4. Dress;
5. Coat;
6. Sandal;
7. Shirt;

8. Sneaker;

9. Bag;

10. Ankle boot.

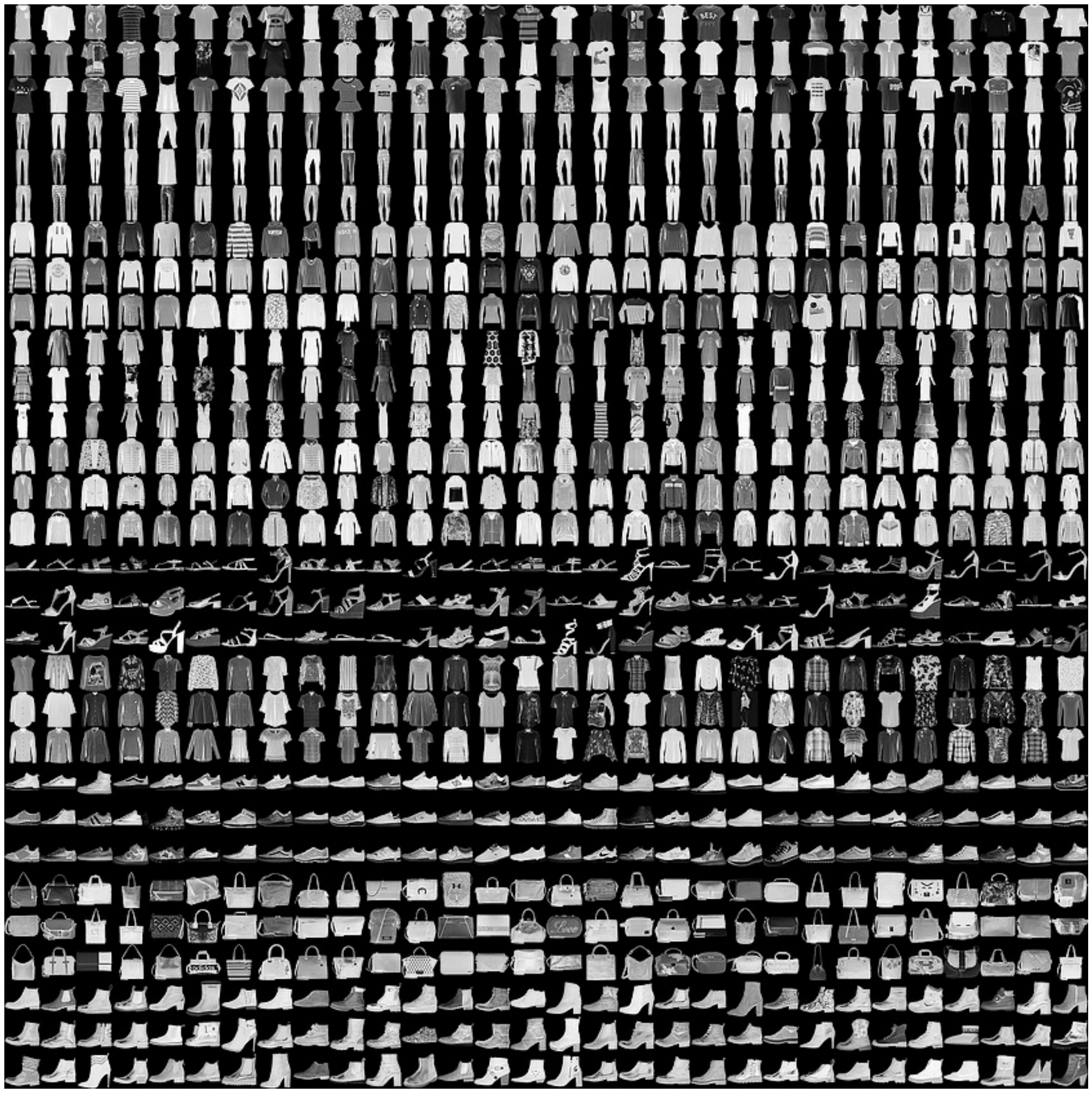


Figure 5.3: Fashion MNIST dataset

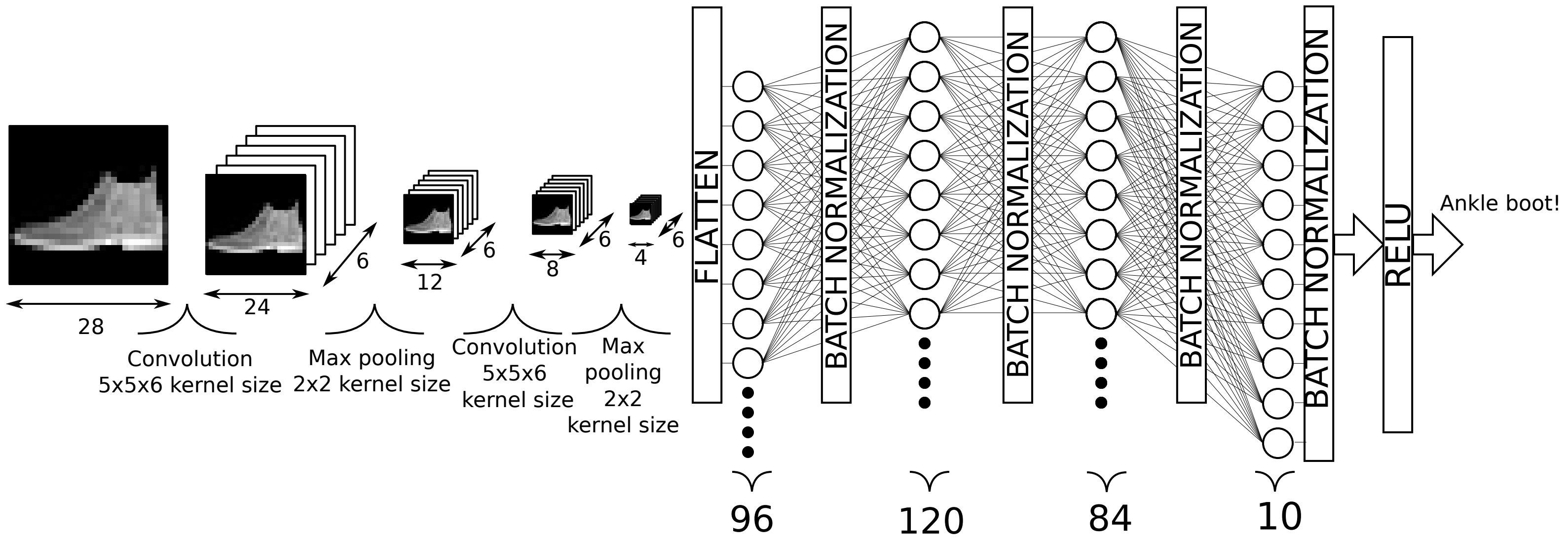


Figure 5.4: CNN model used for fashion-MNIST dataset. All convolutional layers have a kernel size of 5x5x6 with stride 1. Max pooling layers have a kernel size of 2x2 with stride 1. After each fully connected layer there is a batch normalization computation, in order to reduce the inaccuracies caused by the approximated computation introduced in section 4.1.2. This model is able to achieve up to 70% of accuracy.

This neural network has been designed to verify the possibility to implement any kind of CNN configuration, since max-pooling layers have been placed after convolutional layers respect to the original neural network model depicted in Figure 4.1. In Figure 5.4 there are multiple input channels used, since the first convolution has 6 OFMAPs that are fed to the following stages. The package that defines the variable/fixed parameters is:

```
-------------FIXED PARAMETERS---------------
------------FASHION MNIST CNN---------------
constant w_out: integer:=24;
constant h_max: integer:=576;
constant w_in: integer:=28;
constant w_filter: integer:=25;
constant w_filter_s: integer:=5;
constant width_sram:integer:=32;
constant number_of_output_channels: integer:=6;
constant number_of_input_channels: integer:=6;
constant number_of_fc_parameters:integer:=152;
constant number_of_neurons_output: integer:=120;
constant input_image_size_x:integer:=28;
constant input_image_size_z:integer:=1;
constant flatten_x:integer:=16;
constant flatten_z:integer:=6;
constant n_layers:integer:=5;
-----------------VARIABLE PARAMETERS---------
constant number_of_layers: std_logic_vector(n_layers-1 downto 0):= "00100";
---------------------------------------------------------------------------
constant conv_layer_size_x: int_vect(n_layers-1 downto 0):=(1,1,1,12,28);
constant conv_layer_size_x_pow:int_vect(n_layers-1 downto 0):=(1,1,1,144,784);
constant conv_layer_size_y: int_vect(n_layers-1 downto 0):=conv_layer_size_x;
constant conv_layer_size_z: int_vect(n_layers-1 downto 0):=(6,6,3,6,1);
constant kernel_size_xy_pow:int_vect(n_layers-1 downto 0):=(14,20,32,25,25);
constant kernel_size_xy: int_vect(n_layers-1 downto 0):=(1,1,1,5,5);
constant kernel_size_z: int_vect(n_layers-1 downto 0):=(1,1,1,6,6);
constant stride_sel_c: int_vect(n_layers-1 downto 0):=(1,1,1,1,1);
constant output_size_conv: int_vect(n_layers-1 downto 0):=(1,1,1,8,24);
constant output_size_conv_pow: int_vect(n_layers-1 downto 0):=(11,84,120,64,576);
---------------------------------------------------------------------------
---------------POOLING-------------
constant pool_filter_size:int_vect(n_layers-1 downto 0):=(1,1,4,4,1);
constant pool_x_size:int_vect(n_layers-1 downto 0):=(1,1,8,24,1);
constant pool_out_size:int_vect(n_layers-1 downto 0):=(1,1,16,144,1);
constant pool_filter_size_s:int_vect(n_layers-1 downto 0):=(1,1,2,2,1);
constant pool_stride:int_vect(n_layers-1 downto 0):=(1,1,2,2,1);
constant pool_x_size_pow:int_vect(n_layers-1 downto 0):=(1,1,64,576,1);
constant pool_z_size:int_vect(n_layers-1 downto 0):=(1,1,6,6,1);
---------------------------------------------------------------------------
constant do_batch_layer: std_logic_vector(n_layers-1 downto 0):= "11100";
constant do_pool_layer:std_logic_vector(n_layers-1 downto 0):=   "00110";
constant do_relu_layer: std_logic_vector(n_layers-1 downto 0):=  "10000";
constant fully_connected:std_logic_vector(n_layers-1 downto 0):= "11100";
```

Starting from fixed parameters:

- `w_out`: the maximum output dimension is given by the first convolutional layer, which convolves 28x28 IFMAP with 6 kernels of 5x5 and stride 1.

$$w_{out(max)} = \frac{w_{in} - w_{filter}}{stride} + 1 = 24 \tag{5.8}$$

- `h_max`: maximum number of rows of the custom memories, which is equal to $w^2_{out(max)}$, so 576;

- `w_filter`: maximum kernel size used in the architecture, which is 25;

- `width_sram`: fixed to 32. Considering the W contraints:

$$\begin{cases} W \geq w^2_{filter} = 25 \\ W \geq L \end{cases} \tag{5.9}$$

  In this architecture there are 3 fully connected layers, and the values of L are different for each of them. The first one takes 96 inputs, the second 120 and the last one 84: for the first case, L is fixed to 32, since it is the first divisor of 96 after $w^2_{filter} = 25$. In the second case, L is fixed to 20 and in the third one L = 14. Consequently, W=32.

- `number_of_output_channels`: the architecture produces 6 output channels in both convolutions;

- `number_of_input_channels`: the second convolution takes 6 channels in input and produces 6 OFMAPs. For this motivation, both architectures should be replicated 6 times, imposing $c_{in} = 6$;

- `number_of_fc_parameters`: the maximum size of the fully connected layers is 120, but in the VHDL implementation it is imposed equal to 152 to avoid indexing errors in the fc scheduling. This value has been obtained as 120 + 32, where 32 is W;

- `flatten_x` and `flatten_z`: since the output sizes before the first fully connected layer are 4x4x6, the flatten layer has to consider 16 elements of each `Output register files`.

Regarding variable parameters, only the relevant changes are discussed, since for the others the same considerations made before are valid:

- `conv_layer_size_x`: the first layer takes in input the entire image with 28x28 pixels. The second one, takes the convolved and pooled image with size 12x12. The remaining numbers refer to the fully connected layers, in which this parameter is not used;

- `kernel_size_xy_pow`: in the convolution computation, it indicates the kernel size which is 25, while in the FC part the L sizes;

- `kernel_size_z`: 6 kernels are used in the convolutions, since 6 OFMAPs are produced in both the cases;

- `conv_layer_size_z`: the first layer takes in input only one channel, since the image has 28x28x1 pixels. The second one takes 6 input channels, as reported in Figure 5.4. The number in position 3 is equal to 3 and indicates n_{iter} in the fully connected computation: since the first FC layer takes 96 inputs, with L fixed to 32, the total number of iterations required to perform the fc scheduling (section 4.1.2) is equal to 3 (32*3=96). Same considerations have been made for the other cases.

The MATLAB's output is the following:

```
First convolution (1) is correct!
First convolution (2) is correct!
First convolution (3) is correct!
First convolution (4) is correct!
First convolution (5) is correct!
First convolution (6) is correct!
Second convolution (1) is correct!
Second convolution (2) is correct!
Second convolution (3) is correct!
Second convolution (4) is correct!
Second convolution (5) is correct!
Second convolution (6) is correct!
(1) FC is correct!
(2) FC is correct!
(3) FC is correct!
```

For demonstration purposes, it is reported only the first OFMAP of the second max pooling layer, since the other matrices are very big:

VHDL output

```
 7.812500e-03 9.375000e-02 5.234375e-01 1.039062e+00
 3.320312e-01 4.648438e-01 6.718750e-01 5.546875e-01
 7.382812e-01 6.718750e-01 -6.523438e-01 -9.804688e-01
-8.945312e-01 -1.554688e+00 -1.492188e+00 -1.625000e+00
```

MATLAB's output

$$Pool_2(:\ ,:\ ,1) = \begin{pmatrix} 0.00781 & 0.0938 & 0.523 & 1.04 \\ 0.332 & 0.465 & 0.672 & 0.555 \\ 0.738 & 0.672 & -0.652 & -0.98 \\ -0.895 & -1.55 & -1.49 & -1.62 \end{pmatrix}$$

Chapter 6

Synthesis - Place & Route

6.1 Original architecture

In this section, are reported the synthesis and place&route results obtained for the neural network model depicted in Figure 4.1. The dimensions of the architecture are summarized in the following table:

Table 6.1: Dimensions of the top entity in terms of # input bits, XNOR Gates, Pop units etc. The reference neural network is depicted in Figure 4.1.

Layer	Type	Parameter	Formula	Value
CONVOLUTIONAL LAYER	Memory sizes	W	$\frac{number\_of\_fc\_parameters}{n\_iter}$	6
		H	h_{max}	169
		Z	c_{in}	1
	Number of XNOR GATES	IN MEMORY	$W \times H \times c_{in}$	1014
		OOM	$W \times c_{in}$	6
	Number of POP Units	IN MEMORY	$H \times c_{in}$	169
		OOM	c_{in}	1
	Input sizes [# bits]	Input filters	$c_{in} \times n_{bit}$	18
		Input image	$c_{in} \times n_{bit}$	18
		Input weights FC	W	6
		Input image FC	W	6
		A,B	n_{bit}	18
		Binary inputs/Binary weights	$W \times c_{in}$	6
	Output sizes [# bits]	Output convolution	$n_{bit} \times c_{in}$	18
POOL	Input sizes [# bits]	Input pool	$c_{in} \times n_{bit}$	18
	Output sizes [# bits]	Output pool	$c_{in} \times n_{bit}$	18

6.1.1 Synthesis

The two circuits are synthesized with Synopsys Design Compiler with CMOS 45nm, and, in particular, are analyzed the differences in terms of area, timing and power. Power estimation is performed in the worst case: switching activity is equal to '1' in each node of the network. The options used for the synthesis are the following:

1. Chosen clock period of 5.5ns:
   `create_clock -name MY_CLK -period 5.5 clk`;

2. Clock uncertainty (jitter) is applied:
   `set_clock_uncertainty 0.07 [get_clocks MY_CLK]`;

3. Delay of inputs/outputs:
   ```
   set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs] clk]
   set_output_delay 0.5 -max -clock MY_CLK [all_outputs]
   ```

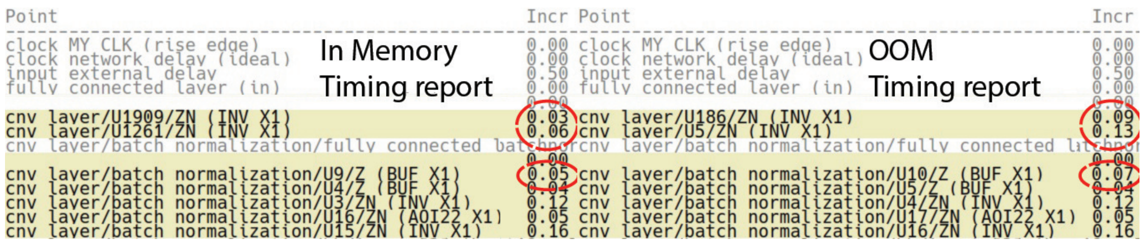
4. For sake of simplicity, the input capacitance of `BUF_X4` (which is equal to 3.40 fF) is chosen as load for the outputs:
   ```
   set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
   set_load $OLOAD [all_outputs]
   ```

Results

Table 6.2: Results in terms of area, critical path delay, power, total energy and time required by the two architectures for the neural network model depicted in Figure 4.1

Area[mm^2]	Critical path delay[ns]	Power [mW]	Time required [μs]	Total energy [nJ]
In Memory architecture				
0.0923	4.22	12.9	61.209	789.6
OOM architecture				
0.0564	4.38	8.85	150.333	1330.4

Critical path delay The critical path is formed by a multiplier and an adder of the `batch normalization` unit, as depicted in Figure 4.17. The values of the critical path delays are different because of the different synthesis choices that Synopsys has made. In the following figure it is reported a part of the timing report of both architectures:



```
Point                                              Incr
---------------------------------------------------------
clock MY CLK (rise edge)                           0.00
clock network delay (ideal)                        0.00
input external delay                               0.50
fully connected layer (in)                         0.00
                                                   0.00
cnv layer/U1909/ZN (INV X1)                        0.03
cnv layer/U1261/ZN (INV X1)                        0.06
cnv layer/batch normalization/fully connected batchnor
                                                   0.00
cnv layer/batch normalization/U9/Z (BUF X1)        0.05
cnv layer/batch normalization/U4/Z (BUF X1)        0.04
cnv layer/batch normalization/U3/ZN (INV X1)       0.12
cnv layer/batch normalization/U16/ZN (AOI22 X1)    0.05
cnv layer/batch normalization/U15/ZN (INV X1)      0.16
```

```
Point                                              Incr
---------------------------------------------------------
clock MY CLK (rise edge)                           0.00
clock network delay (ideal)                        0.00
input external delay                               0.50
fully connected layer (in)                         0.00
                                                   0.00
cnv layer/U186/ZN (INV X1)                         0.09
cnv layer/U5/ZN (INV X1)                           0.13
cnv layer/batch normalization/fully connected litchnor
                                                   0.00
cnv layer/batch normalization/U10/Z (BUF X1)       0.07
cnv layer/batch normalization/U5/Z (BUF X1)        0.04
cnv layer/batch normalization/U4/ZN (INV X1)       0.12
cnv layer/batch normalization/U17/ZN (AOI22 X1)    0.05
cnv layer/batch normalization/U16/ZN (INV X1)      0.16
```

Figure 6.1: Part of timing reports for both architectures. The main differences are highlighted by the red dashed circles. The same logic gate has been implemented into two different ways in the architectures.

The worst case is analyzed: critical path delays of both architectures are equal. The other results are now discussed.

Area An analysis of the main contributions that defines the area of the designs is performed, taking into account only the main differences between the two architectures:

1. Number of XNOR gates: since the architectures are different from each other, they have a different number of XNOR gates, that can be defined considering the dimensions of the `XNOR UNIT` and `XNOR UNIT in memory` for OOM and In-memory structures respectively:

$$\#\text{XNOR Gates(in memory)} = w \times h = 6 \times 169 = 1014$$
$$\#\text{XNOR Gates(OOM)} = w \times 1 = 6$$
$$XNOR\_Gate\_ratio = 169$$

2. Pop-counting units: in the OOM architecture there is only one pop-count unit, since the structure is serialized, while for the in-memory architecture there are 169 of them:

$$Pop\_ratio = 169 \tag{6.1}$$

These two contributions produce the following area ratio:

$$AR = \frac{Area_{\underline{OOM}}}{Area_{In-Memory}} = \frac{0.0564}{0.0923} \simeq 0.611 \tag{6.2}$$

Power As expected, the resulting power is worst in the case of in-memory implementation, since Synopsys is not able to perform in-memory designs for XNOR-UNIT and pop-counting parts. The estimations performed in the In-Memory case are pessimistic, since the memory has been implemented as a register file and a flip-flop is more complicated than a custom memory cell, composed by a memory element and a XNOR gate. The in-memory architecture consumes $\approx 1.45\times$ more power than the OOM counterpart.

Total energy Taking the power and total delay values, the total energy of both architectures has been evaluated. The energy ratio is given by:

$$ER = \frac{1330.4nJ}{789.6nJ} = 1.7 \tag{6.3}$$

The In-Memory architecture consume $\sim 1.7\times$ less energy than the OOM counterpart. This is a very good result, because the main goal of an In-Memory architecture is to change the design approach in order to find a solution with lower energy consumed and computational delay, since the computational elements (in this case XNOR gates and full-adders) are placed near-memory element. Consequently, the Von Neumann's bottleneck is reduced.

6.1.2 Place & Route

In this section are reported the Place & Route results for both the architectures:

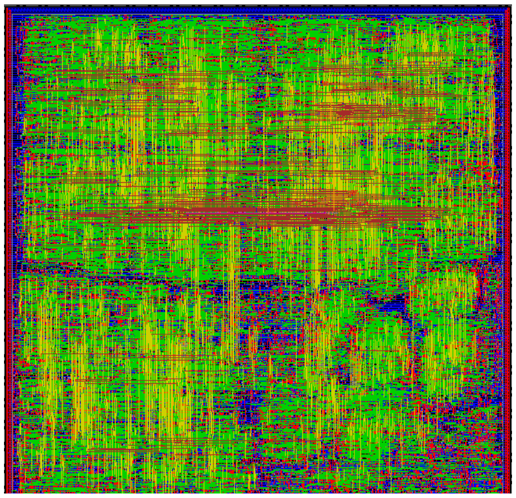


Figure 6.2: Physical chip of OOM architecture

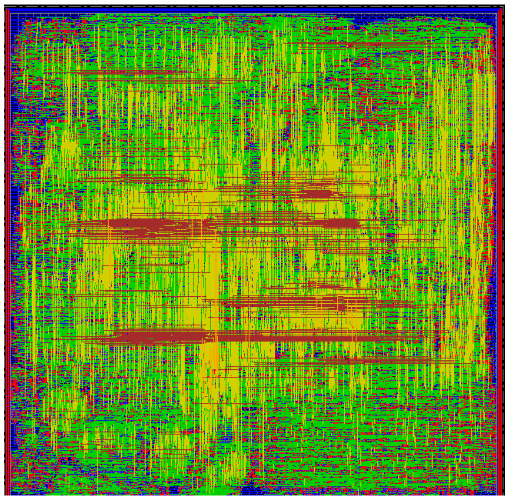


Figure 6.3: Physical chip of In-Memory architecture

By looking at Figure 6.2, it is possible to see that the structure is less complex than the In-Memory alternative: this confirms the expectations also on the power consumption, as discussed in the synthesis part. Now there are reported the power reports for a clock period of 5.5ns of the two architectures. They are performed with `.vcd` files, in order to take into account also the switching activities of the circuits.

OOM implementation

```
* Power Units = 1mW
-----------------------------------------------------------------------------
Cell                              Internal   Switching       Total     Leakage
                                     Power       Power       Power       Power
-----------------------------------------------------------------------------
Total (   26504 of   26504   )       6.747      0.2858       8.08        1.047
Total Capacitance                  1.444e-10 F
```

In-memory implementation

```
* Power Units = 1mW
-----------------------------------------------------------------------------
Cell                              Internal   Switching       Total     Leakage
                                     Power       Power       Power       Power
-----------------------------------------------------------------------------
```

```
Total (  46104 of  46104  )        10.98       1.861      14.54       1.705
Total Capacitance                2.554e-10 F
```

In the OOM case, the capacitance load used by Synopsys influences the power consumption, producing an higher result than the real one, computed by Innovus. In the second one, the interconnections have a big impact in terms of power consumption, increasing the synthesis estimated one by 1.64 mW. The energy ratio can be evaluated considering the interconnections contributions:

$$ER_{\text{Place\&Route}} = \frac{8.08mW \times 150.333\mu s}{14.54mW \times 61.209\mu s} =\simeq 1.37 \tag{6.4}$$

Since the In-Memory architecture has a more complex structure than the OOM counterpart, the energy is much more degraded: as a consequence the energy ratio increases by ~ 0.33. Regarding the critical path delay, after the Place&Route phase, the worst slack values have been analyzed for both architectures with $t_{ck} = 5.5ns$. Their values are reported from `neural_network_postRoute_hold.slk`:

```
Worst slack (In-Memory) = 0.005 ns
Worst slack (OOM) = 0.005 ns
```

The interconnections increase the clock period from 4.22ns to $\sim 5.5ns$.

6.2 MLP architecture

The results for the MLP architecture depicted in Figure 5.2 are reported. The dimensions of both the architectures (OOM and In-Memory) are the following:

Table 6.3: Dimensions of the top entity. The reference neural network is depicted in Figure 5.2. The (-)s indicate don't care, since it is a MLP architecture, they are fixed to the minimum size.

Layer	Type	Parameter	Formula	Value
CONVOLUTIONAL LAYER	Memory sizes	W	$\frac{number\_of\_fc\_parameters}{n\_iter}$	14
		H	h_{max}	196
		Z	c_{in}	1
	Number of XNOR GATES	IN MEMORY	$W \times H \times c_{in}$	2744
		OOM	$W \times c_{in}$	14
	Number of POP Units	IN MEMORY	$H \times c_{in}$	196
		OOM	c_{in}	1
	Input sizes [# bits]	Input filters	$c_{in} \times n_{bit}$	-
		Input image	$c_{in} \times n_{bit}$	-
		Input weights FC	W	14
		Input image FC	W	14
		A,B	n_{bit}	18
		Binary inputs/Binary weights	$W \times c_{in}$	-
	Output sizes [# bits]	Output convolution	$n_{bit} \times c_{in}$	18
POOL	Input sizes [# bits]	Input pool	$c_{in} \times n_{bit}$	-
	Output sizes [# bits]	Output pool	$c_{in} \times n_{bit}$	-

6.2.1 Synthesis & Place-Route chips

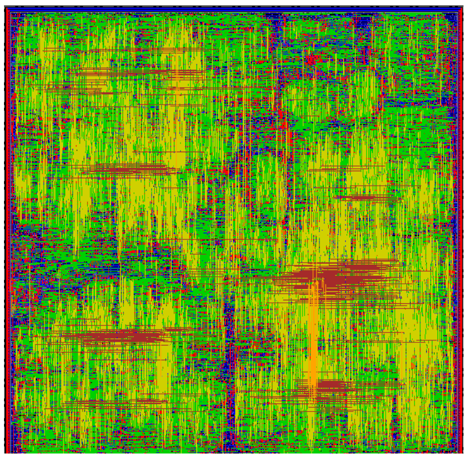


Figure 6.4: OOM chip implementing the neural network model depicted in Figure 5.2

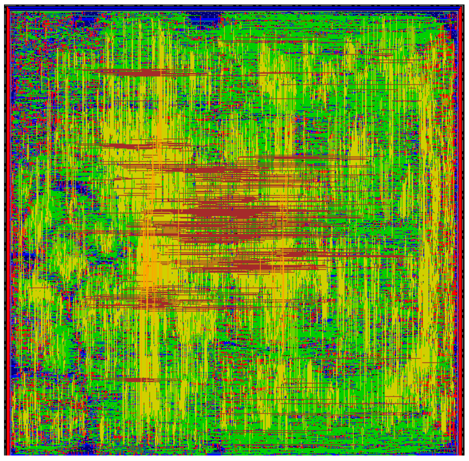


Figure 6.5: In-Memory chip implementing the neural network model depicted in Figure 5.2

By performing the synthesis, the results obtained in terms of area, critical path delay and power are reported in Table 6.4. The time required by the two algorithms can be computed considering that are 4 fully connected layers. The steps are the following:

1. At the beginning of the algorithm, the data are precharged inside the memories. The total number of clock cycles required can be computed also considering that this procedure is performed everytime `iteration_cycle` increases:

$$Delay_{Data_{acq}} = (\phi + (n_{layers} - 1) \times \psi) \times t_{ck} = 4 \times 798 \times t_{ck} \tag{6.5}$$

2. The fully connected layers have the following delays for the In-Memory and OOM alternatives:

$$Delay_{FC(In-Memory)} = ((1 + n_{iter} \times (w_{out(fc)} + L + 2) + w_{out(fc)} + 2) \times t_{ck} \tag{6.6}$$

$$\begin{aligned} Delay_{FC(OOM)} &= [3 + n_{iter} \times (2 + w_{out(fc)} + w_{out(fc)} \times (L + 1)) + \\ &+ w_{out(fc)}] \times t_{ck} \end{aligned} \tag{6.7}$$

The computations of the layers are the following:

(a) First layer $n_{iter} = 56$, $w_{out(fc)} = 196$, $L = 14$:

$$\begin{aligned} Delay_{FC(In-Memory)} &= ((1 + 56 \times (196 + 14 + 2) + 196 + 2) \times t_{ck} \\ &= 12071 \times t_{ck} \\ Delay_{FC(OOM)} &= [3 + 56 \times (196 + 2 + 196 \times (14 + 1)) + 196] \times t_{ck} \\ &= 175927 \times t_{ck} \end{aligned}$$

(b) Second layer $n_{iter} = 14$, $w_{out(fc)} = 196$, $L = 14$:

$$\begin{aligned} Delay_{FC(In-Memory)} &= ((1 + 14 \times (196 + 14 + 2) + 196 + 2) \times t_{ck} \\ &= 3167 \times t_{ck} \\ Delay_{FC(OOM)} &= [3 + 14 \times (196 + 2 + 196 \times (14 + 1)) + 196] \times t_{ck} \\ &= 44131 \times t_{ck} \end{aligned}$$

(c) Third layer $n_{iter} = 14$, $w_{out(fc)} = 196$, $L = 14$:

$$Delay_{FC(In-Memory)} = ((1 + 14 \times (196 + 14 + 2) + 196 + 2) \times t_{ck}$$
$$= 3167 \times t_{ck}$$
$$Delay_{FC(\underline{OOM})} = [3 + 14 \times (196 + 2 + 196 \times (14 + 1)) + 196] \times t_{ck}$$
$$= 44131 \times t_{ck}$$

(d) Fourth layer $n_{iter} = 14$, $w_{out(fc)} = 10$, $L = 14$:

$$Delay_{FC(In-Memory)} = ((1 + 14 \times (10 + 14 + 2) + 10 + 2) \times t_{ck}$$
$$= 377 \times t_{ck}$$
$$Delay_{FC(\underline{OOM})} = [3 + 14 \times (10 + 2 + 10 \times (14 + 1)) + 10] \times t_{ck}$$
$$= 2281 \times t_{ck}$$

The final delays of the FC layers are:

$$Delay_{FC(In-Memory)} = 18782 \times t_{ck}$$
$$Delay_{FC(\underline{OOM})} = 266470 \times t_{ck}$$

Considering all the contributions:

$$Delay_{\underline{OOM}} = (21974 + overheads) \times t_{ck}$$
$$Delay_{In-Memory} = (269606 + overheads) \times t_{ck}$$
$$DR = \frac{269606}{21974} \simeq 12.27$$

From Modelsim, the real times required by the architectures to perform the algorithm with a clock period of 6ns are given by:

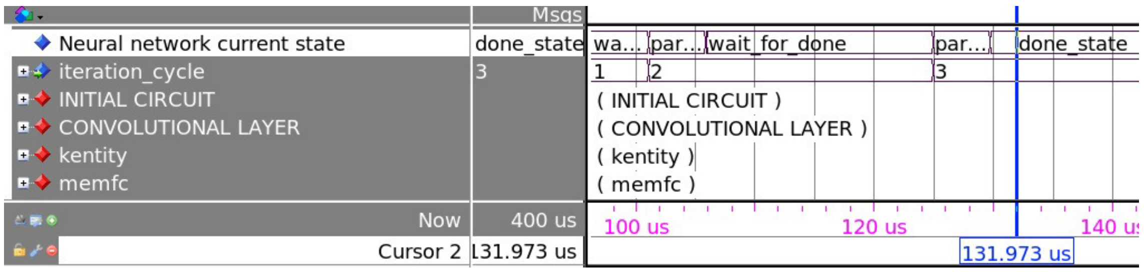


Figure 6.6: Computational delay of the In-Memory architecture, implementing the neural network model depicted in Figure 5.2.

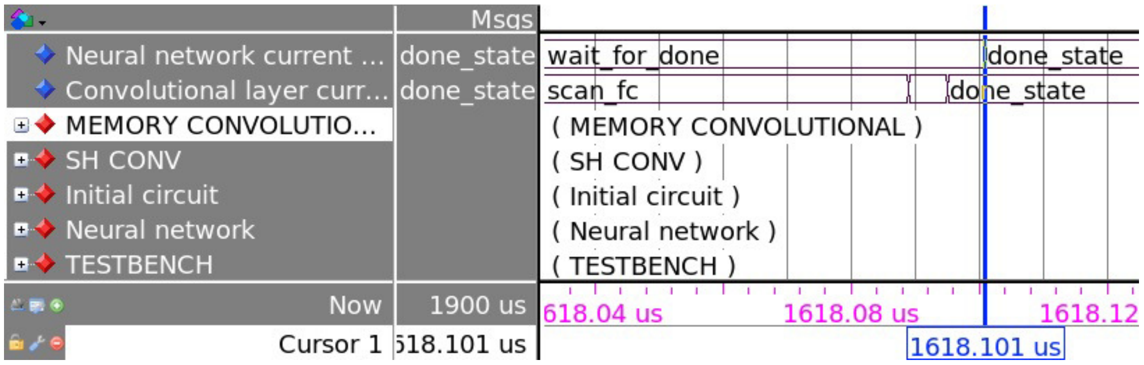


Figure 6.7: Computational delay of the OOM architecture, implementing the neural network model depicted in Figure 5.2.

Giving a ratio of $\sim$ 12.26. From a delay point of view, the In-Memory architecture is very efficient to perform the fully connected computations w.r.t the OOM, because of the parallelization technique and the possibility to perform the XNORs/pops directly inside the memory array.

Table 6.4: Results in terms of area, critical path delay, power, total energy and time required by the two architectures for the neural network model depicted in Figure 5.2

Area[mm^2]	Critical path delay[ns]	Power [mW]	Time required [ms]	Total energy [μJ]
In Memory architecture				
0.1055	4.220	15.1	0.132	1.99
OOM architecture				
0.0876	4.32	14.32	1.62	23.2

The powers in Table 6.4 are comparable, which is a very good result considering the dimensions and gate count of the In-Memory architecture respect to OOM. The energy ratio is given by:

$$ER = \frac{23.2\mu J}{1.99\mu J} \simeq 11.7\times \tag{6.8}$$

Also from the energy consumption point of view, the In-Memory is far more efficient than the OOM one.

6.3 Fashion-MNIST CNN

The case of fashion-MNIST CNN depicted in Figure 5.4 is now discussed. The parameters chosen for this model assume the following values:

1. The number of bits used in this architecture is 16, with 8 fractional bits and 8 integer bits;
2. The number of input channels is equal to 6, since there are 2 convolutional layers and the second one takes in input 6 channels;
3. w_{filter}^2 is fixed to 25 for the convolutional part, while max-pooling has only 2x2 kernel size;
4. W is fixed to 32, for the motivations already explained in subsection 5.3.2;
5. H is 576 from the first convolutional layer's output size, which is given by 24x24 OFMAPs.

Table 6.5: Dimensions of the top entity. The reference neural network is depicted in Figure 5.4.

Layer	Type	Parameter	Formula	Value
CONVOLUTIONAL LAYER	Memory sizes	W	$\frac{number\_of\_fc\_parameters}{n\_iter}$	32
		H	h_{max}	576
		Z	c_{in}	6
	Number of XNOR GATES	IN MEMORY	$W \times H \times c_{in}$	110592
		OOM	$W \times c_{in}$	192
	Number of POP Units	IN MEMORY	$H \times c_{in}$	3456
		OOM	c_{in}	6
	Input sizes [# bits]	Input filters	$c_{in} \times n_{bit}$	96
		Input image	$c_{in} \times n_{bit}$	96
		Input weights FC	W	32
		Input image FC	W	32
		A,B	n_{bit}	16
		Binary inputs/Binary weights	$W \times c_{in}$	192
	Output sizes [# bits]	Output convolution	$n_{bit} \times c_{in}$	16
POOL	Input sizes [# bits]	Input pool	$c_{in} \times n_{bit}$	96
	Output sizes [# bits]	Output pool	$c_{in} \times n_{bit}$	96

This network is composed by 2 convolutional layers, 2 pooling layers and 3 fully connected layers. The total delay can be computed as:

1. Precharge

$$Delay_{Data_{acq}} = (\phi + (n_{layers} - 1) \times \psi) \times t_{ck} = (784 + 4 \times 152) \times t_{ck} \tag{6.9}$$

2. First convolutional layer: $w^2_{filter} = 25$, $w^2_{out} = 576$, $c_{out} = 6$, $c_{in} = 1$

$$Delay_{In-Memory(convolution)} = [3 + w^2_{out} \times (w^2_{filter} + 1) + (c_{out} \times (w^2_{filter} +$$
$$+ w^2_{out} \times (c_{in} + 2)) + c_{out} \times 3 + 1)] \times t_{ck}$$
$$= 25516 \times t_{ck}$$
$$Delay_{OOM(convolution)} = [3 + w^2_{out} \times (w^2_{filter} + 1) +$$
$$+ (c_{out} \times w^2_{out} \times (w^2_{filter} + c_{in} + 2) + c_{out} \times 4 + 1)] \times t_{ck}$$
$$= 111772 \times t_{ck}$$

3. Max pooling: $w^2_{filter} = 4$, $w^2_{out(pool)} = 144$

$$\begin{aligned} &Delay_{Pooling} = (4 + w^2_{out(pool)} \times (2 + w^2_{filter})) \times t_{ck} \\ &= (4 + 144 \times (2 + 4)) \times t_{ck} = 868 \times t_{ck} \end{aligned} \tag{6.10}$$

4. Second convolutional layer: $w^2_{filter} = 25$, $w^2_{out} = 64$, $c_{out} = 6$, $c_{in} = 6$

$$\begin{aligned} &Delay_{In-Memory(convolution)} = [3 + w^2_{out} \times (w^2_{filter} + 1) + (c_{out} \times (w^2_{filter} + \\ &+ w^2_{out} \times (c_{in} + 2)) + c_{out} \times 3 + 1)] \times t_{ck} \\ &= 4908 \times t_{ck} \\ &Delay_{OOM(convolution)} = [3 + w^2_{out} \times (w^2_{filter} + 1) + \\ &+ (c_{out} \times w^2_{out} \times (w^2_{filter} + c_{in} + 2) + c_{out} \times 4 + 1)] \times t_{ck} \\ &= 14364 \times t_{ck} \end{aligned}$$

5. Max pooling: $w^2_{filter} = 4$, $w^2_{out(pool)} = 16$

$$\begin{aligned} &Delay_{Pooling} = (4 + w^2_{out(pool)} \times (2 + w^2_{filter})) \times t_{ck} \\ &= (4 + 16 \times (2 + 4)) \times t_{ck} = 100 \times t_{ck} \end{aligned} \tag{6.11}$$

6. First fully connected layer: $W_{out(fc)} = 120$, $L = 32$, $n_{iter} = \frac{96}{32} = 3$

$$\begin{aligned} &Delay_{FC(In-Memory)} = ((1 + n_{iter} \times (w_{out(fc)} + L + 2) + w_{out(fc)} + 2) \times t_{ck} \\ &= 585 \times t_{ck} \\ &Delay_{FC(\underline{OOM})} = [3 + n_{iter} \times (2 + w_{out(fc)} + w_{out(fc)} \times (L + 1)) + \\ &+ w_{out(fc)}] \times t_{ck} \\ &= 12369 \times t_{ck} \end{aligned}$$

7. Second fully connected layer: $W_{out(fc)} = 84$, $L = 20$, $n_{iter} = \dfrac{120}{20} = 6$

$$\begin{aligned}
&Delay_{FC(In-Memory)} = ((1 + n_{iter} \times (w_{out(fc)} + L + 2) + w_{out(fc)} + 2) \times t_{ck} \\
&= 723 \times t_{ck} \\
&Delay_{FC(\underline{OOM})} = [3 + n_{iter} \times (2 + w_{out(fc)} + w_{out(fc)} \times (L + 1)) + \\
&+ w_{out(fc)}] \times t_{ck} \\
&= 11187 \times t_{ck}
\end{aligned}$$

8. Third fully connected layer: $W_{out(fc)} = 11$, $L = 14$, $n_{iter} = \dfrac{84}{14} = 6$

$$\begin{aligned}
&Delay_{FC(In-Memory)} = ((1 + n_{iter} \times (w_{out(fc)} + L + 2) + w_{out(fc)} + 2) \times t_{ck} \\
&= 176 \times t_{ck} \\
&Delay_{FC(\underline{OOM})} = [3 + n_{iter} \times (2 + w_{out(fc)} + w_{out(fc)} \times (L + 1)) + \\
&+ w_{out(fc)}] \times t_{ck} \\
&= 1082 \times t_{ck}
\end{aligned}$$

Considering all the contributions:

$$\begin{aligned}
&Delay_{\underline{OOM}} = (34268 + overheads) \times t_{ck} \\
&Delay_{In-Memory} = (153134 + overheads) \times t_{ck} \\
&DR = \frac{153134}{34268} \simeq 4.47
\end{aligned}$$

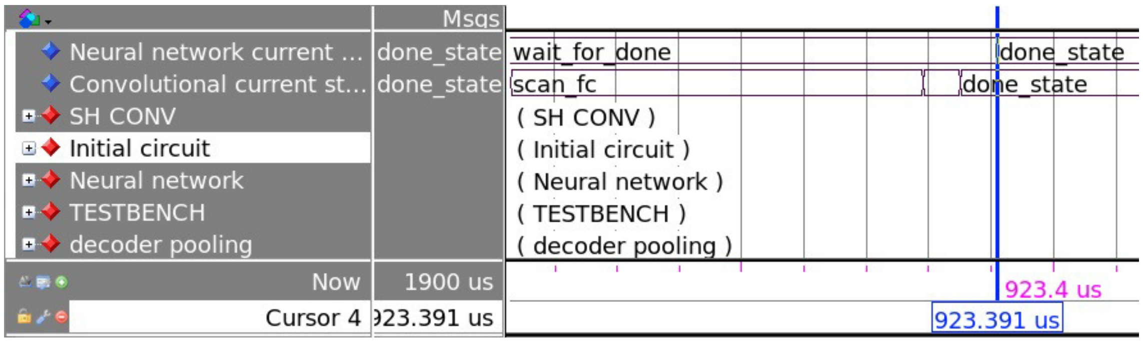


Figure 6.8: Computational delay of the OOM architecture, implementing the neural network model depicted in Figure 5.4.

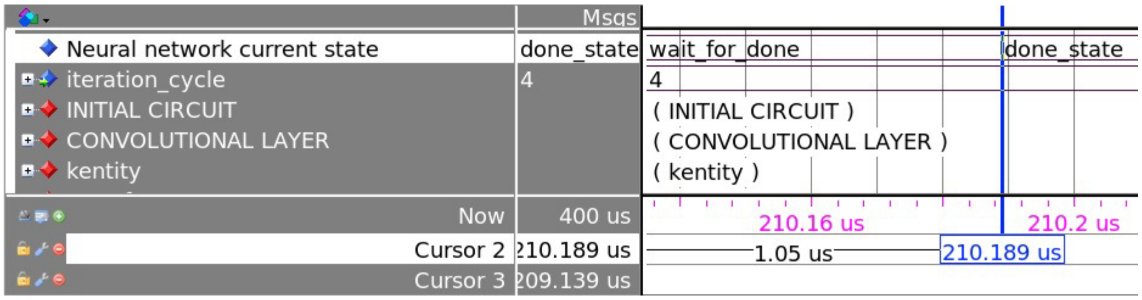


Figure 6.9: Computational delay of the In-Memory architecture, implementing the neural network model depicted in Figure 5.4.

The real delay ratio is $\sim 4.4\times$. The synthesis results are the following:

Table 6.6: Results in terms of area, critical path delay, power, total energy and time required by the two architectures for the neural network model depicted in Figure 5.4

Area$[mm^2]$	Critical path delay[ns]	Power [mW]	Time required $[ms]$	Total energy $[\mu J]$
In Memory architecture				
1.68	4.11	254.5	0.210	53.445
OOM architecture				
1.10	4.14	193.30	0.923	178.41

The energy ratio is $\sim 3.34\times$, which is a very good result considering the dimensions of the network.

6.4 General cases

To evaluate the performance of the network with different parameters, several synthesis have been performed. In each of them are evaluated power, energy, area and timing and the results are compared between the In-Memory and OOM architecture's ones. The same architecture implemented for the neural network model in Figure 4.1 is used and it is sweeped only two values per time. The parameters chosen are the following:

1. n_{bit}: considering the plot in Figure 4.50, the evaluation can start from 12 bits (11 integer and 1 fractional) to 21 bits (11 integer and 10 fractional);
2. The w_{filter} parameter has been sweeped from its initial value ($w_{filter} = 2$) to $w_{filter} = 11$, emulating the kernel size in deeper neural networks, such as AlexNet;
3. c_{in} number of input channels are sweeped from 2 to 7, in order to evaluate the cost in terms of performance of having parallel architectures working at the same time;
4. H size to evaluate the impact of having a bigger OFMAP.

Regarding the energy estimations, it has been considered only a convolution computation with $c_{out} = 1$ (exception for the H case, in which also the fully connected algorithm case is considered), because it represents the worst case in terms of delay ratio respect to fully connected, which is depicted in Figure 4.49. Taking the delay ratio trend, energy ratio is given by Power ratio multiplied by Delay ratio. This procedure produces the worst case energy results, since a neural network (CNN or MLP) is always composed by fully connected layers, in which there is the effective gain in terms of delay. To better understand this consideration, the original network depicted in Figure 4.1 is taken as example.

1. The power values for OOM and In-Memory architectures are 8.85 mW and 12.9 mW respectively, producing an energy ratio of ~ 1.7, as already discussed in section 6.1.1;

2. Considering only one convolution computation with $c_{out} = 6$ and the same dimensions of the model (Figure 4.1), the resulting delay ratio is:

$$\begin{aligned} DR &= \frac{3 + w_{out}^2 \times (w_{filter}^2 + 1) + (c_{out} \times w_{out}^2 \times (w_{filter}^2 + c_{in} + 2) + c_{out} \times 4 + 1)}{3 + w_{out}^2 \times (w_{filter}^2 + 1) + (c_{out} \times (w_{filter}^2 + w_{out}^2 \times (c_{in} + 2)) + c_{out} \times 3 + 1)} \\ &= \frac{3 + 169 \times (4 + 1) + (6 \times 169 \times 7 + 6 \times 4 + 1)}{3 + 169 \times 5 + (6 \times (4 + 169 \times 3) + 6 \times 3 + 1)} = \frac{7971}{3933} = 2.03 \end{aligned} \tag{6.12}$$

Delay ratio is degraded respect to its real value (~ 2.46), with also fully connected part, reducing the energy ratio to $ER = 1.39$ (from ~ 1.7). In general, by having less c_{out}, delay ratio becomes worse, in fact by performing the same computation with $c_{out} = 1$, $DR = 1.49$.

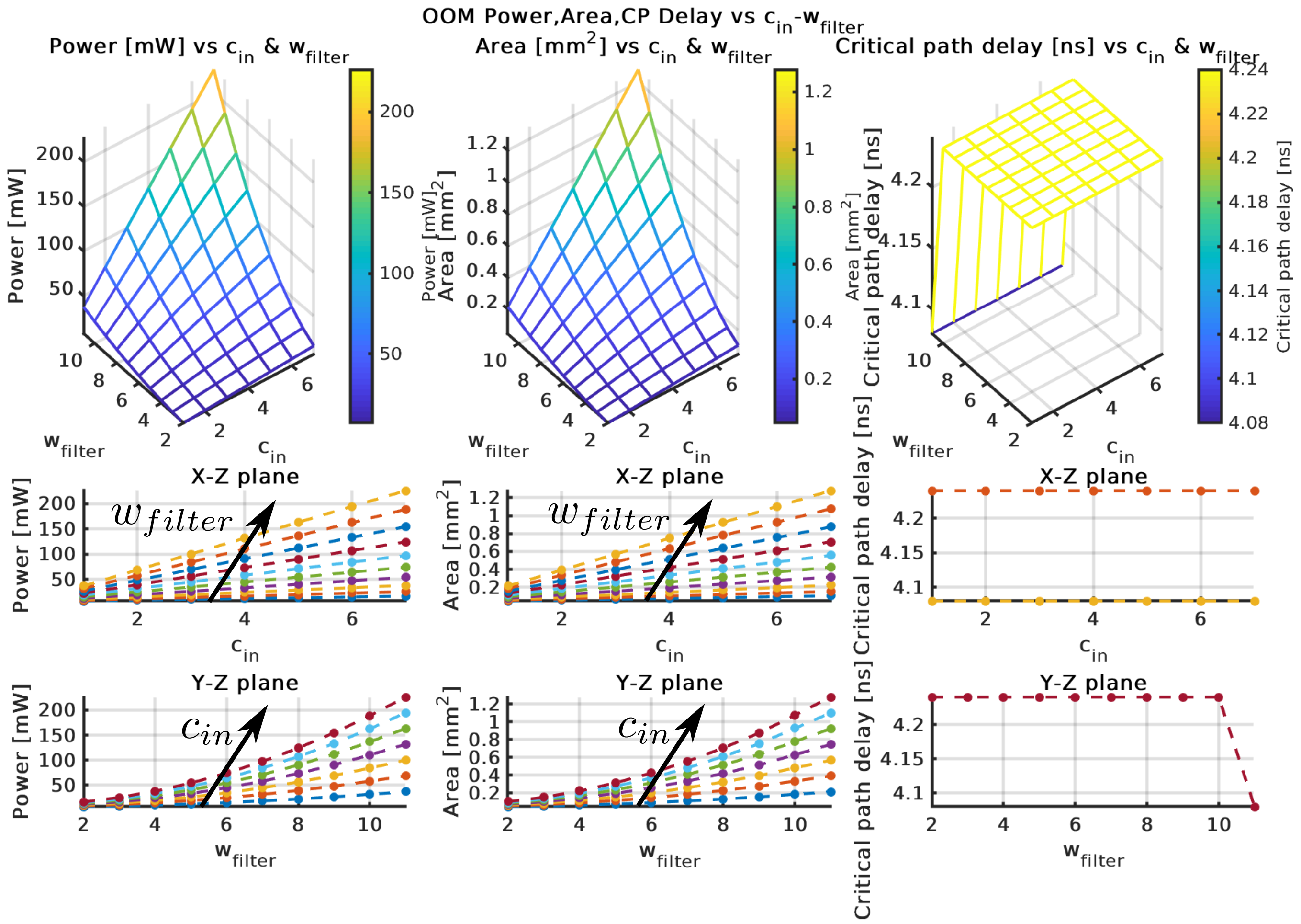


Figure 6.10: Area, CP delay, Power vs c_{in} - w_{filter} for the OOM architecture ($H = 169$, $c_{out} = 1$, $W = w_{filter}^2$). Power vs c_{in} - w_{filter}: power increases almost linearly with c_{in}, because more parallel architectures are working at the same time. With higher w_{filter}, the power rises almost exponentially, because it is required a larger memory array and more XNOR gates are used. Area vs c_{in} - w_{filter} behaves in the same way. CP delay vs c_{in}-w_{filter}: remains almost constant, since it is caused by a multiplier-adder sequence. For an higher amount of c_{in}, more adders are used in the adder trees in **K**-α computations (Figure 4.11 and Figure 4.13), but the critical path remains the same.

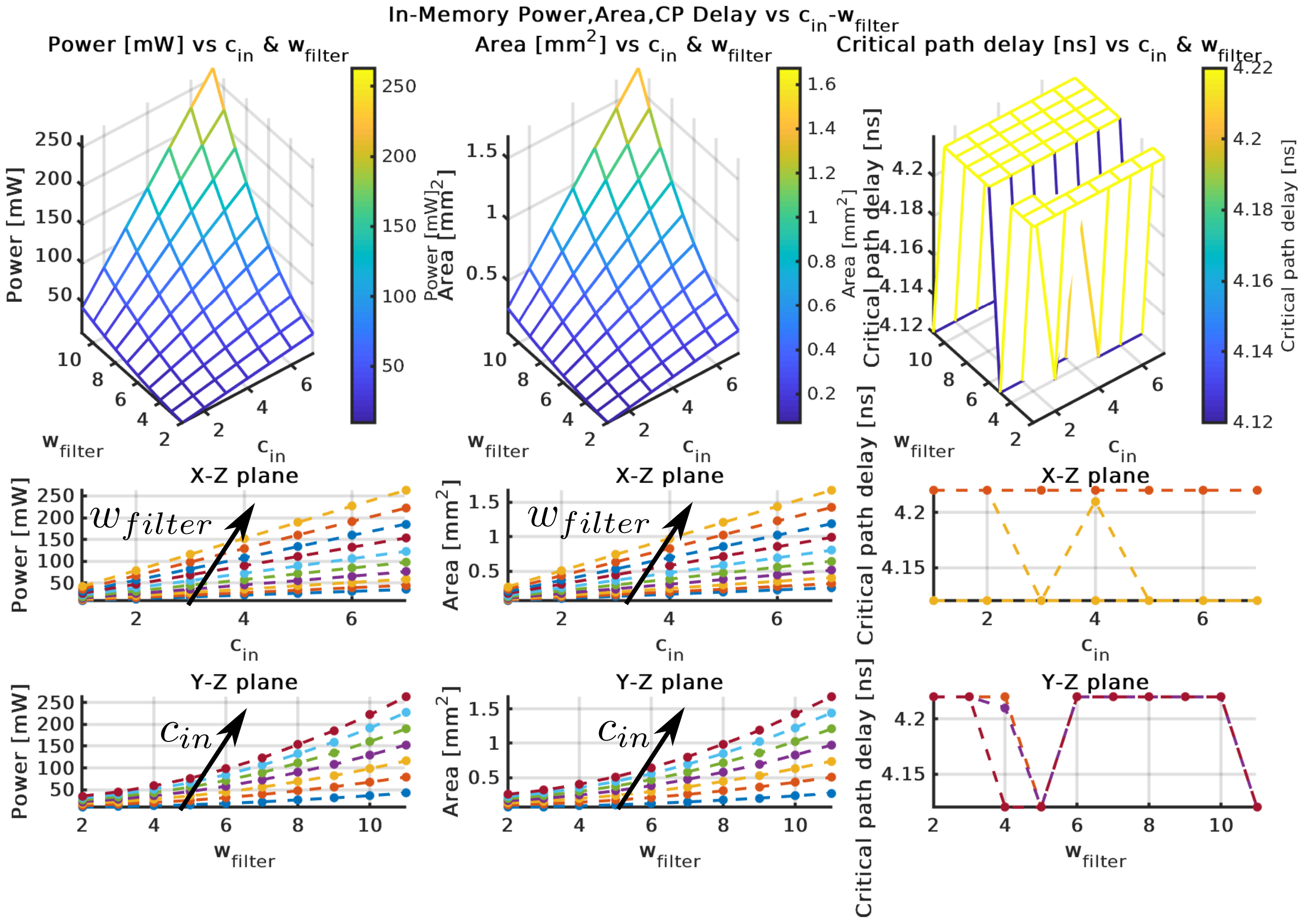


Figure 6.11: Area, Critical path delay, Power vs c_{in} - w_{filter} for the In-Memory architecture ($H = 169$, $c_{out} = 1$, $W = w_{filter}^2$). Same considerations made in Figure 6.10 are valid here. The maximum power achieved in this case is $\sim 260mW$ respect to $\sim 230mW$ of the previous case. Considering the higher number of logic gates required in the In-Memory architecture, it is a very good result that allows also to reduce also the computational time normally required by the OOM architecture.

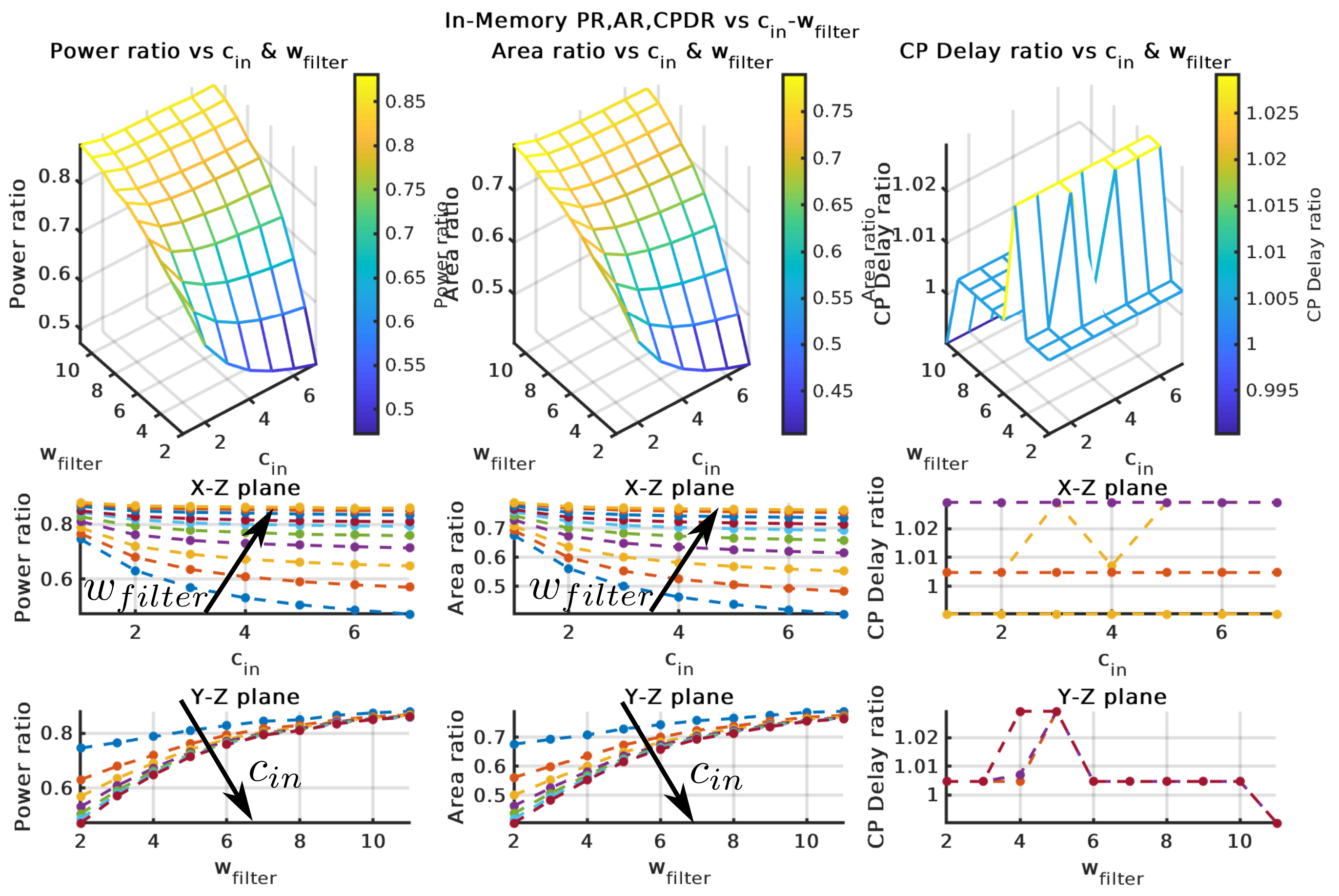


Figure 6.12: Area ratio, Critical path delay ratio, Power ratio vs c_{in} - w_{filter} obtained as OOM/In-Memory ($H = 169$, $c_{out} = 1$, $W = w_{filter}^2$). Increasing c_{in} brings to power/area ratios reductions, since In-Memory architecture requires more building blocks than OOM case. w_{filter}'s rise brings power benefits in the In-Memory architecture, since the registers start to have a predominant contribution respect to the sequential/combinational powers: since the architectures have approximately the same number of registers, the power ratio tends towards 1 for $w_{filter} \to \infty$. From a power consumption point of view, it is convenient to implement an In-Memory architecture with high c_{in} and w_{filter}.

The plot reported in Figure 6.12 reports an important consideration: the higher is w_{filter} and c_{in} the lower are power ratio and area ratio. It means that for the In-Memory architecture it is convenient to realize an architecture with high w_{filter} and c_{in}, as already said. To understand the behavior of the PR, it is sufficient to analyze the following power reports:

OOM architecture's power report with $c_{in} = 7$, $w_{filter} = 11$

```
                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )
------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000              0.0000             0.0000  (   0.00%)
register       1.9380e+05          22.3686          1.1818e+07         2.0563e+05  (  90.96%)
sequential     5.5253e-02       7.8112e-03          2.7743e+03             2.8373  (   0.00%)
combinational  1.7291e+03       7.9070e+03          1.0788e+07         2.0425e+04  (   9.04%)
------------------------------------------------------------------------------------------
Total          1.9552e+05 uW    7.9294e+03 uW       2.2609e+07 nW      2.2606e+05 uW
```

In-Memory architecture's power report with $c_{in} = 7$, $w_{filter} = 11$

```
                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
-------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000              0.0000             0.0000  (   0.00%)
register       2.0721e+05          64.1904          1.2895e+07         2.2016e+05  (  83.71%)
sequential     8.9353e-03       2.7972e-03             38.9396         5.0672e-02  (   0.00%)
combinational  8.3320e+03       1.6143e+04          1.8368e+07         4.2838e+04  (  16.29%)
-------------------------------------------------------------------------------------------------
Total          2.1554e+05 uW    1.6207e+04 uW       3.1263e+07 nW      2.6300e+05 uW
```

The highest power contribution is related to the registers, so the combinational power overhead for the In-Memory case decreases with higher c_{in},w_{filter}.

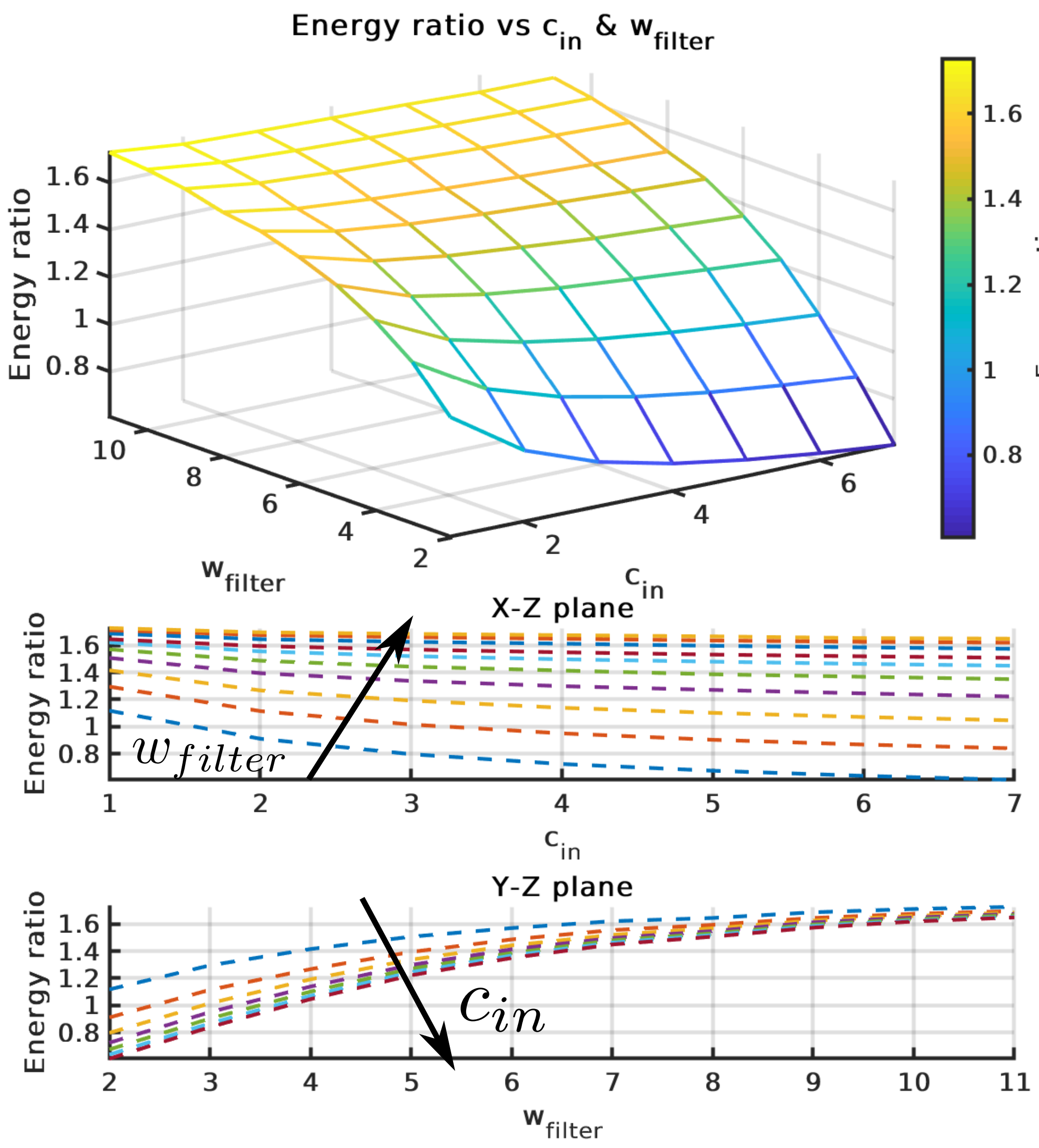


Figure 6.13: Energy ratio vs c_{in}-w_{filter} ($H = 169$, $c_{out} = 1$, $W = w_{filter}^2$). Taking the delay ratio respect to c_{in}-w_{filter} depicted in Figure 4.45, it has been multiplied by the obtained power ratio. The result shows that the In-Memory architecture becomes more efficient in terms of energy for higher values of w_{filter}. Consequently, the effect of c_{in}'s rise is reduced. This is a very good result, since for very deep networks such as AlexNet, the In-Memory architecture reaches better energy results.

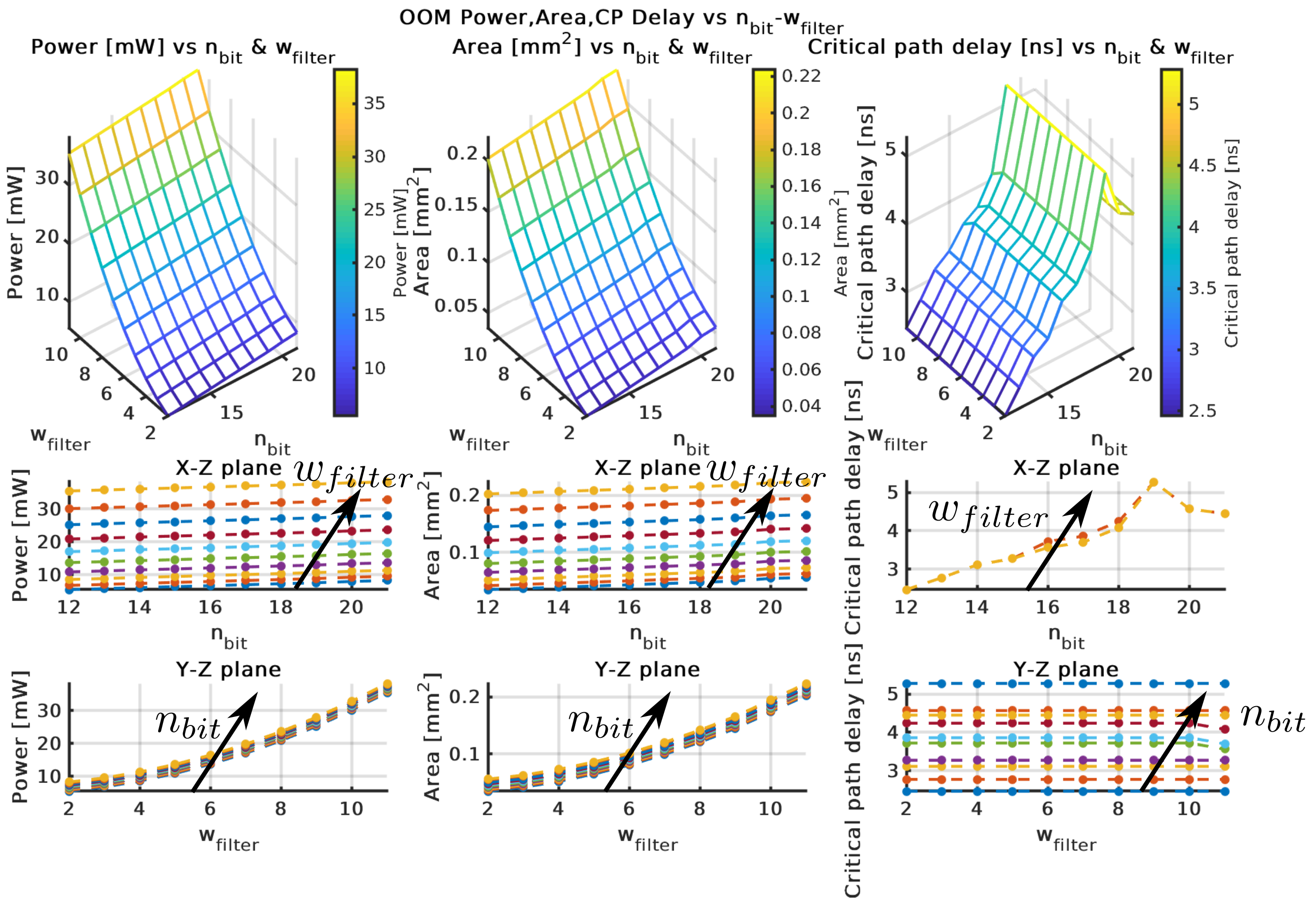


Figure 6.14: Area, Critical path delay, Power vs n_{bit} - w_{filter} for the OOM architecture ($H = 169$, $c_{out} = 1$, $W = w_{filter}^2$, $c_{in} = 1$). Increasing n_{bit}, also power and area rises, since an higher number of bits implies more complicated operators (adders, multipliers etc). In the critical path delay case, it is possible to see a peak located at 19 bits: from the timing report, the critical path is located in the divider of the α unit. As already seen in Figure 6.10, with high values of w_{filter}, both area and power rise exponentially.

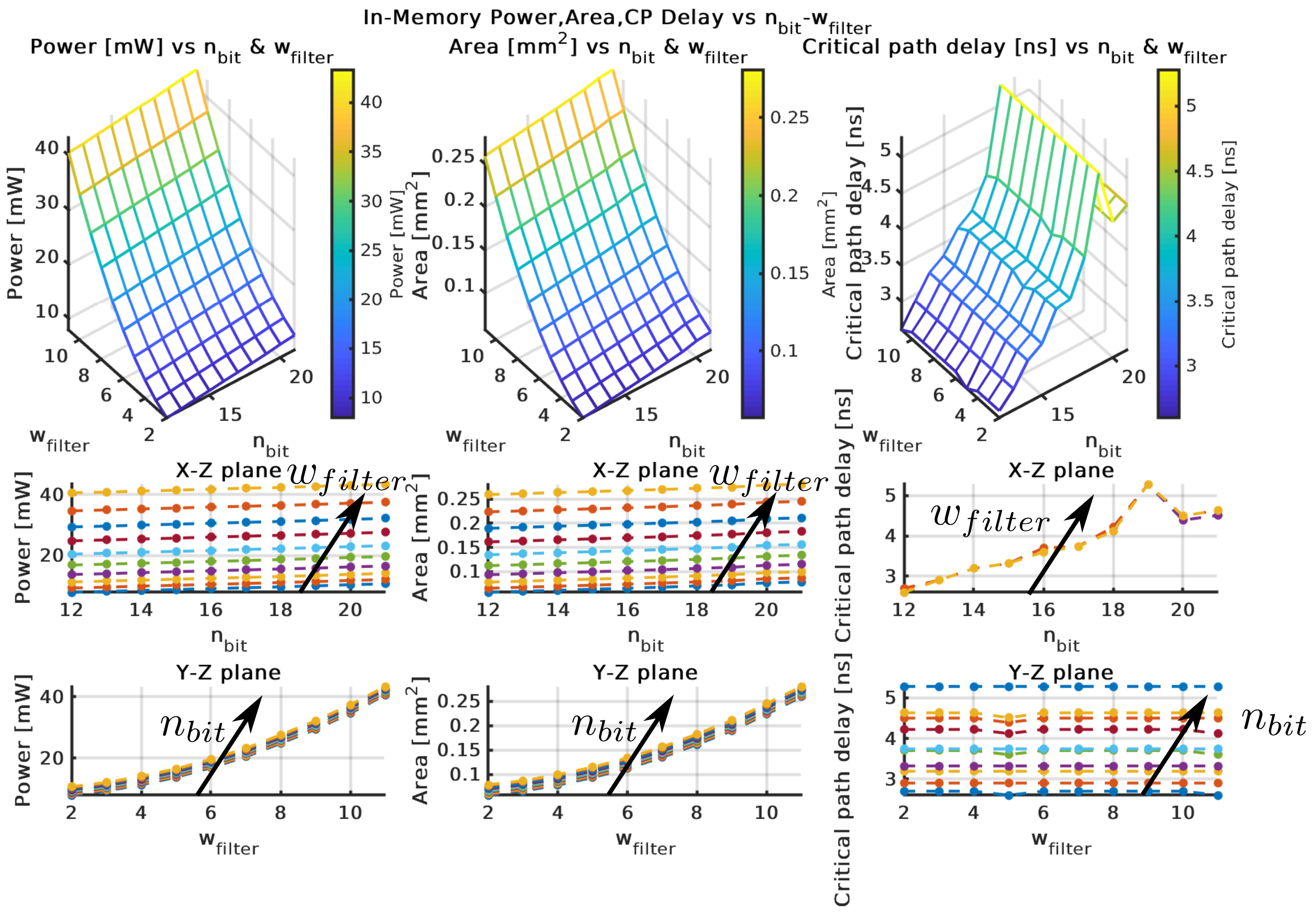


Figure 6.15: Area, Critical path delay, Power vs n_{bit} - w_{filter} for the In-Memory architecture($H = 169$, $c_{out} = 1$, $W = w_{filter}^2$, $c_{in} = 1$). Same considerations of Figure 6.14 are valid here.

In Figure 6.15 and Figure 6.16 there is a peak located in $n_{bit} = 19$. By looking at the timing reports for $n_{bit} = 19$ and $n_{bit} = 20$ of OOM architecture, it is possible to see that the critical path delay in the first case is located in the divider of the α computation unit, while in the second one in the batch normalization unit (adder-multiplier):

OOM architecture's timing report with $n_{bit} = 19$, $w_{filter} = 11$

```
cnv_layer/alpha_computation/div_103/U27/ZN (OR2_X1)     0.06       0.15 f
cnv_layer/alpha_computation/div_103/U26/ZN (NOR3_X1)
                                                        0.06       0.21 r
cnv_layer/alpha_computation/div_103/U25/ZN (NAND2_X1)
                                                        0.03       0.24 f
cnv_layer/alpha_computation/div_103/U24/ZN (NOR3_X1)
                                                        0.06       0.30 r
...
clock MY_CLK (rise edge)                                5.50       5.50
clock network delay (ideal)                             0.00       5.50
clock uncertainty                                      -0.07       5.43
cnv_layer/alpha_computation/r2/dout_reg[0]/CK (DFFR_X1)
                                                        0.00       5.43 r
library setup time                                     -0.04       5.39
data required time                                                 5.39
--------------------------------------------------------------------------
data required time                                                 5.39
data arrival time                                                 -5.28
--------------------------------------------------------------------------
slack (MET)                                                        0.11
```

In-Memory architecture's power report with $n_{bit} = 20$, $w_{filter} = 11$

```
cnv_layer/batch_normalization/U10/Z (BUF_X1)            0.06       0.73 f
cnv_layer/batch_normalization/U6/Z (BUF_X1)             0.04       0.77 f
cnv_layer/batch_normalization/U3/ZN (INV_X1)            0.13       0.90 r
cnv_layer/batch_normalization/U16/ZN (AOI22_X1)         0.06       0.96 f
cnv_layer/batch_normalization/U51/ZN (INV_X2)           0.11       1.07 r
cnv_layer/batch_normalization/bb/inputs[1] (bnorm_n_bit20_multiplication_sx_extreme28)
                                                        0.00       1.07 r
...
clock MY_CLK (rise edge)                                5.50       5.50
clock network delay (ideal)                             0.00       5.50
clock uncertainty                                      -0.07       5.43
output external delay                                  -0.50       4.93
data required time                                                 4.93
--------------------------------------------------------------------------
data required time                                                 4.93
data arrival time                                                 -4.57
--------------------------------------------------------------------------
slack (MET)                                                        0.36
```

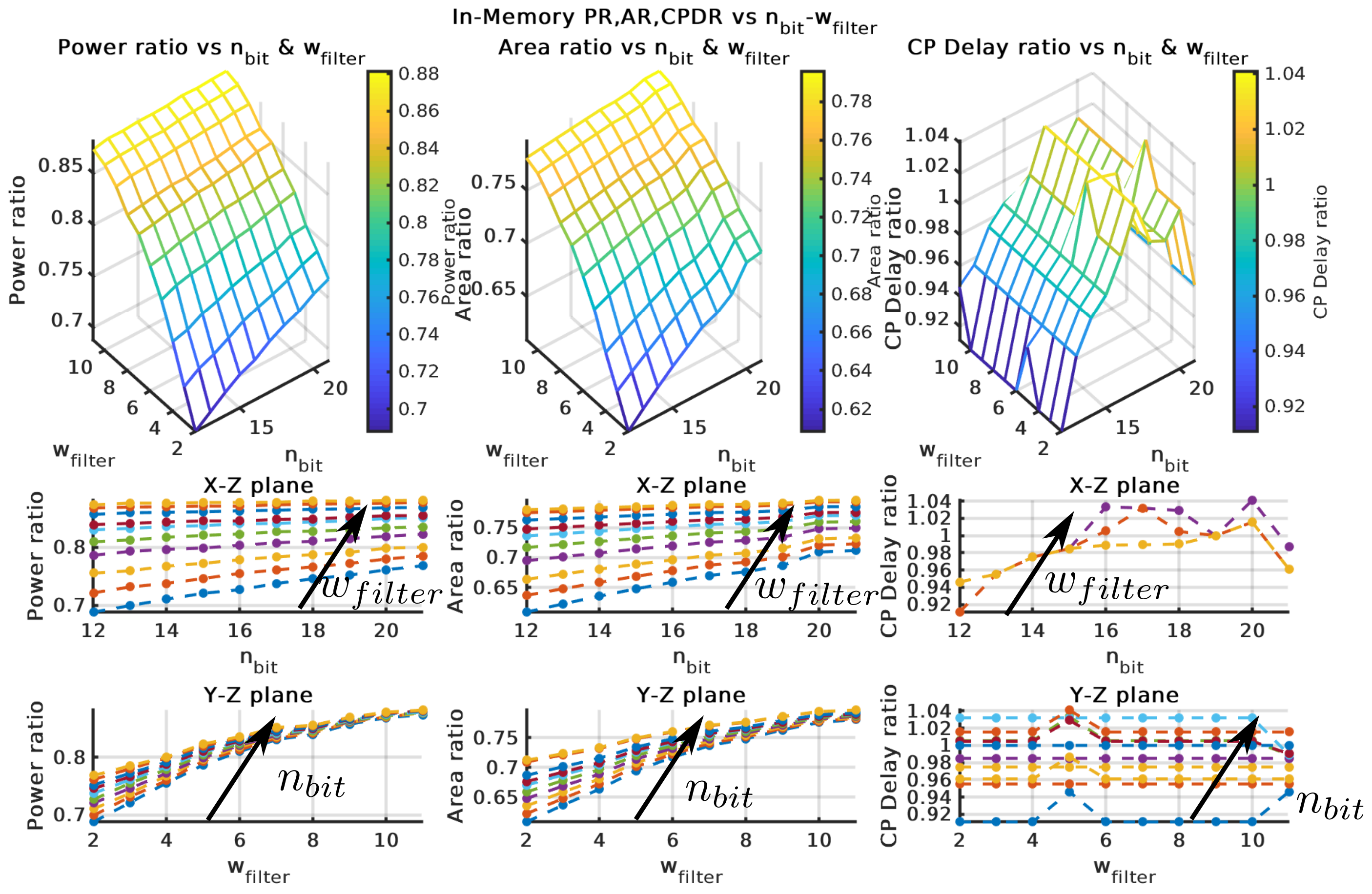


Figure 6.16: Area ratio, Critical path delay ratio, Power ratio vs n_{bit} - w_{filter} obtained as OOM/In-Memory ($H = 169$, $c_{out} = 1, W = w^2_{filter}, c_{in} = 1$). For an high value of n_{bit}, area-power ratios increases. This implies that the In-Memory architecture takes performance advantages, if a more precise representation is used.

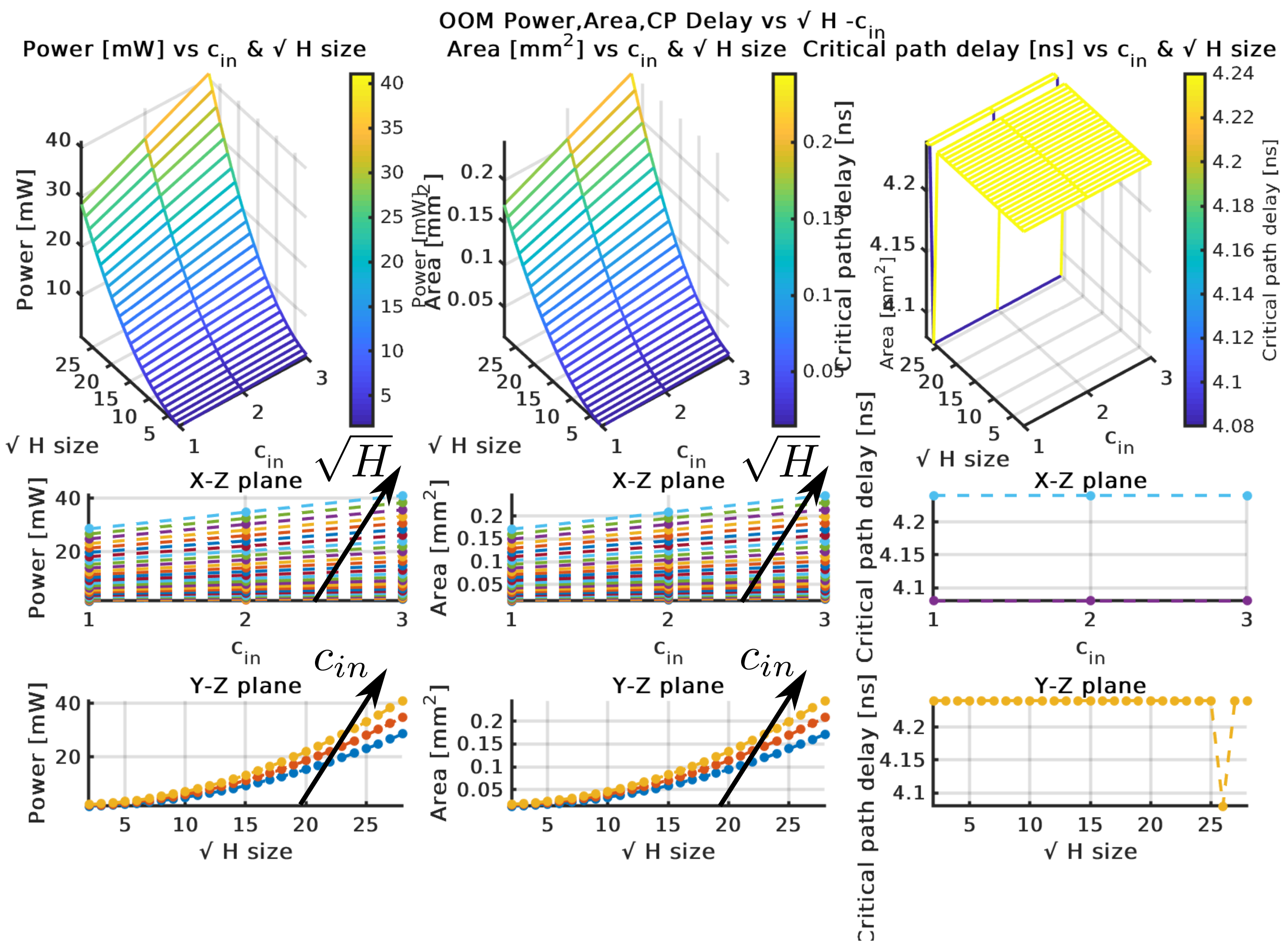


Figure 6.17: Area, Critical path delay, Power vs $\sqrt{H}$ - c_{in} for OOM architecture ($c_{out} = 1$, $W = w_{filter}^2 = 4$). The higher is the $\sqrt{H}$ size, the higher are power consumption and area, since registers have very big sizes (exponential trend). Regarding c_{in}, as already said, power/area increase almost linearly. Critical path delay remains almost the same for each value of $\sqrt{H} - c_{in}$.

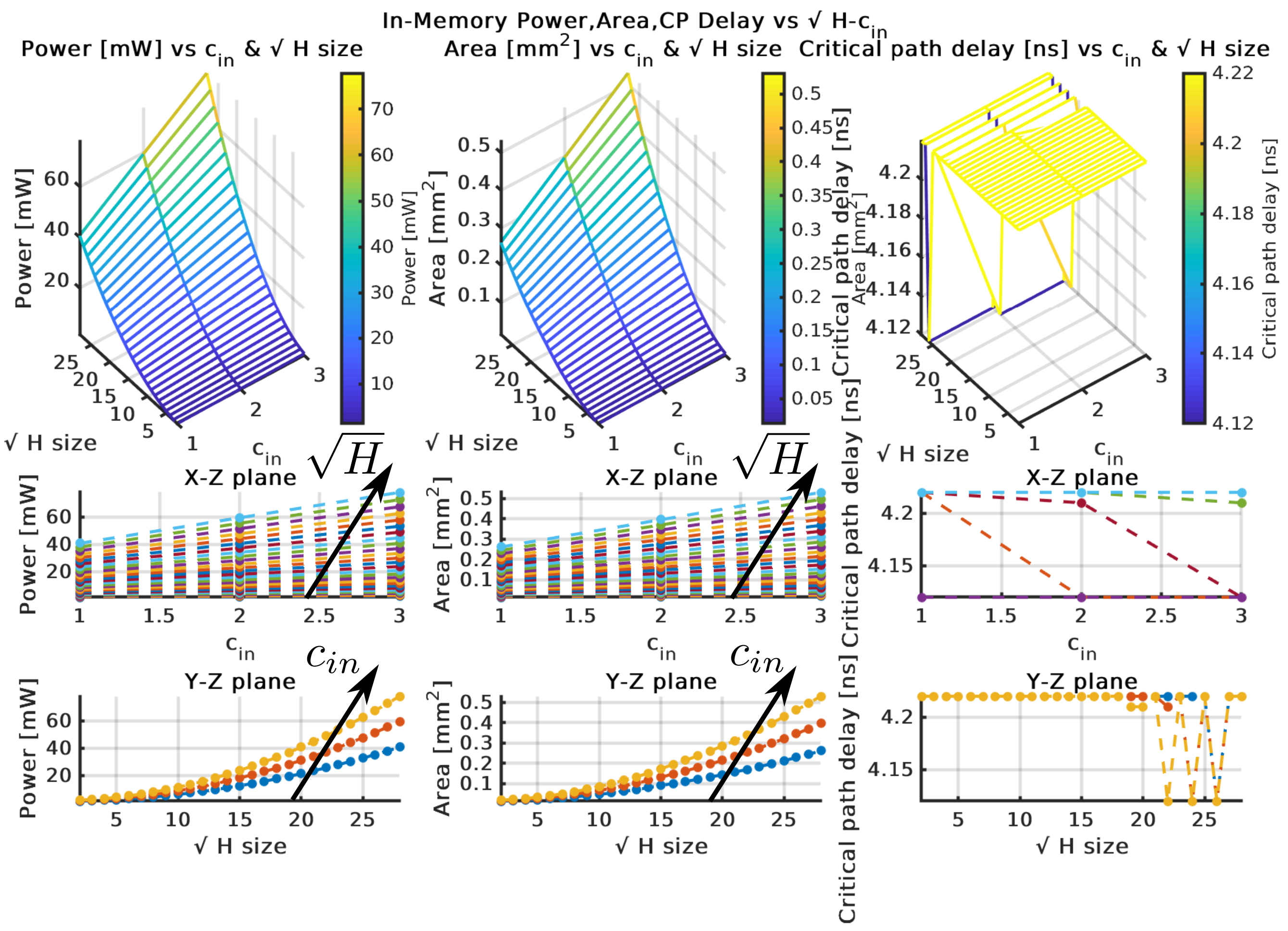


Figure 6.18: Area, Critical path delay, Power vs $\sqrt{H}$ - c_{in} for In-Memory architecture ($c_{out} = 1$, $W = w_{filter}^2 = 4$). Same considerations made for Figure 6.17 are valid in this case. The power/area values reached are higher than the previous case, because of the higher number of registers/logic gates.

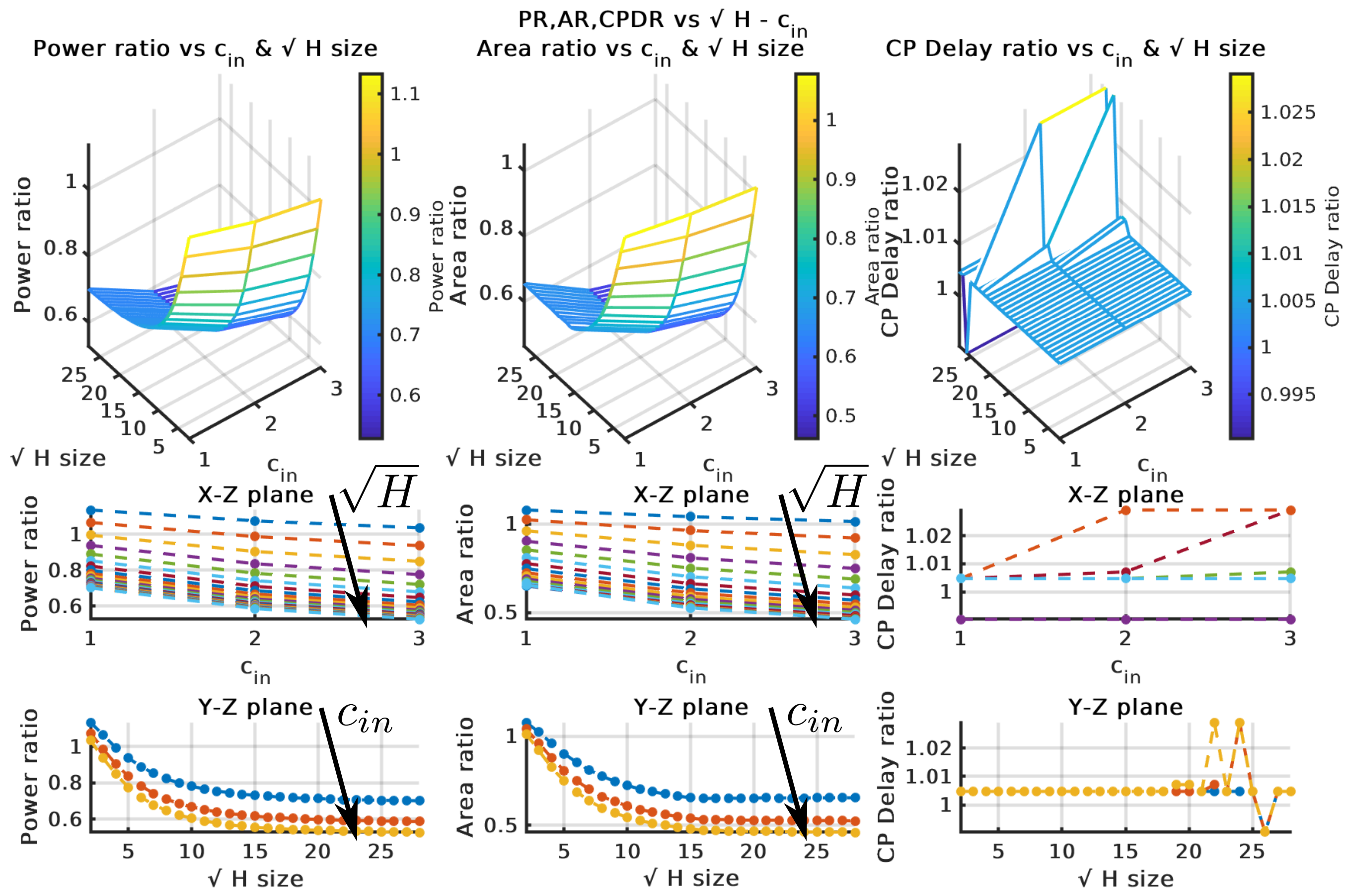


Figure 6.19: Area ratio, Critical path delay ratio, Power ratio vs $\sqrt{H}$ - c_{in}, obtained as OOM/In-Memory ($c_{out} = 1$, $W = w_{filter}^2 = 4$). By increasing both c_{in} and $\sqrt{H}$, power/area ratios decrease, because of the higher amount of logic gates inside the In-Memory architecture.

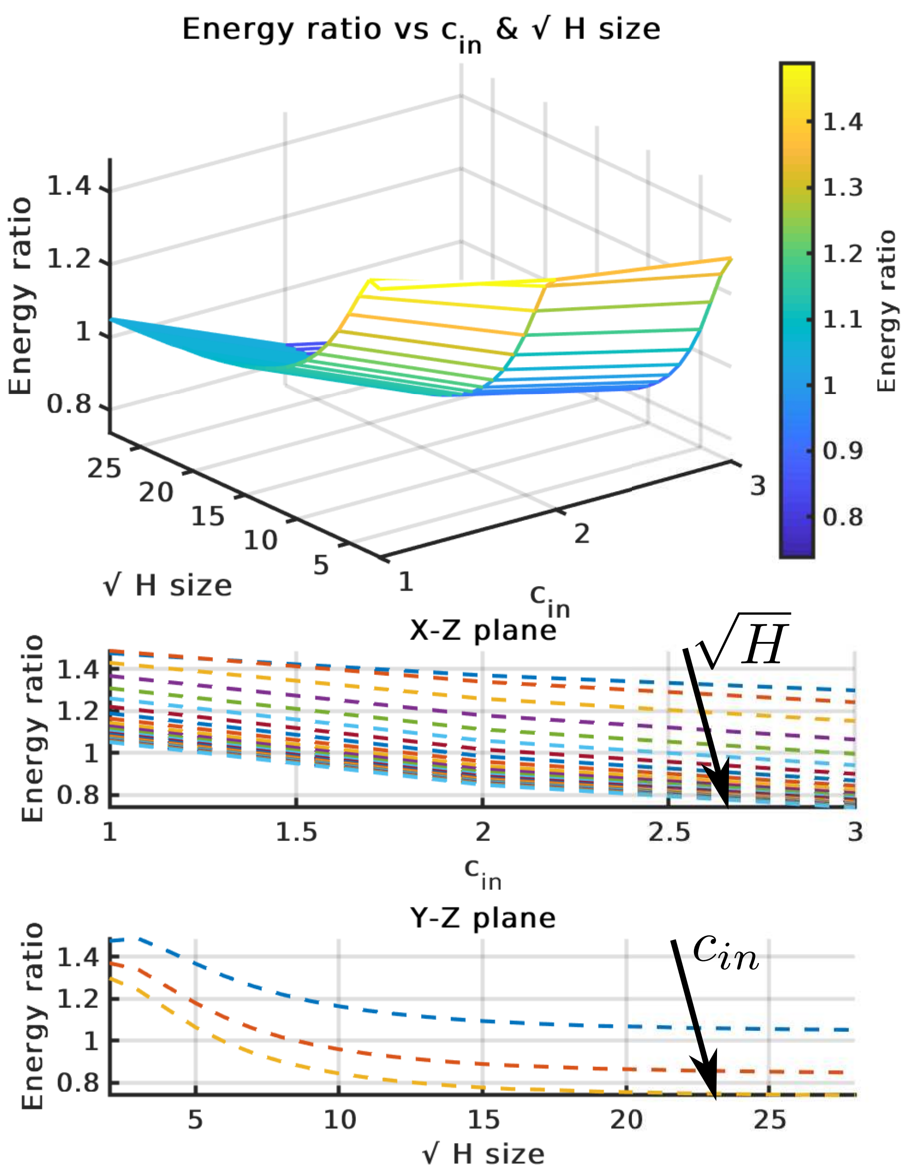


Figure 6.20: Energy ratio vs $\sqrt{H}$ - c_{in}, obtained as <u>OOM</u>/In-Memory ($c_{out} = 1$, $W = w_{filter}^2 = 4$). This is the worst case, because by increasing both c_{in} and $\sqrt{H}$ the energy ratio decreases, because of the higher amount of logic gates inside the In-Memory architecture. With higher values of both w_{filter} and c_{out}, the energy ratio will decrease for the motivations explained before.

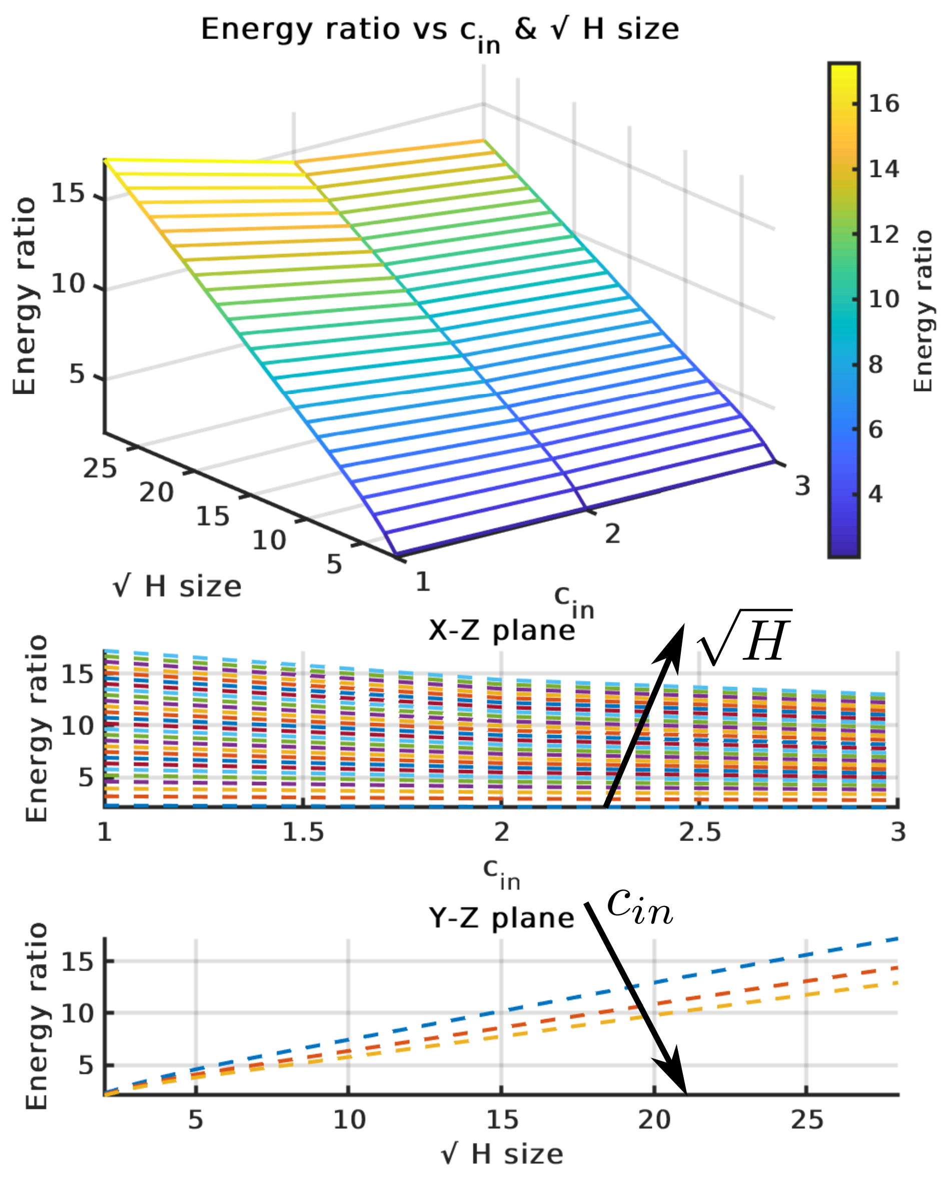


Figure 6.21: Energy ratio vs $\sqrt{H}$ - c_{in} for the fully connected algorithm, obtained as OOM/In-Memory ($c_{out} = 1$, $W = w_{filter}^2 = 4$, $number\_of\_fc\_parameters = 1000$, $n_{iter} = 250$). In this case, the energy ratio increases a lot, since the fully connected algorithm is far more efficient in the in-memory case respect to OOM one. Depending on the algorithm type, the performance can be better or worse: an higher number of fully connected layers with an high value of n_{iter}, implies a more efficient In-Memory architecture than OOM counterpart.

It is possible to delineate the behavior of the performance in both cases, by considering a mean of the obtained values of energy, power, area, delay and timing:

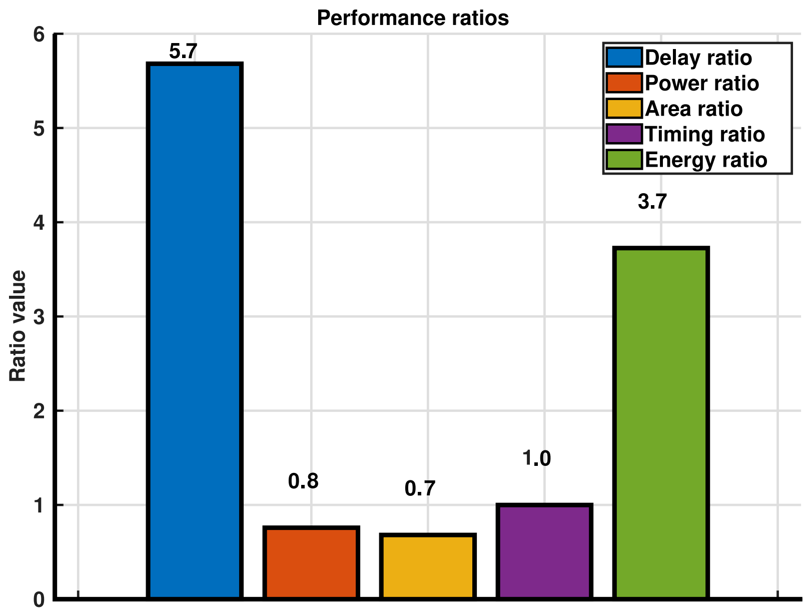


Figure 6.22: Mean Delay, Power, Area, Timing and Energy ratios, obtained as $\frac{OOM}{In-Memory}$. If the ratio value is higher than 1, it means that the In-Memory architecture obtained a better result. As expected, In-Memory alternative is more efficient in terms of Energy/Delay than OOM counterpart.

Figure 6.22 gives an important confirmation of the advantages coming from an In-Memory design respect to a classical Von-Neumann's based one: by placing near-memory very simple elements (such as XNOR gates and full-adders), allows to reduce fetching latency, energy consumption and computational delay.

6.5 State-of-the-art comparisons

⚠ ATTENTION ⚠

The following performance comparisons are based on the assumptions made in chapter 2, in which a linear dependency between the evaluated parameter and the network's complexity is used. The correctness of the obtained values is not guaranteed.

6.5.1 Number of neurons

The examinated architecture is the original one, which implements the model depicted in Figure 4.1. The total number of layers is 3 while the # of neurons can be computed as:

$$\#Neurons_{O.W.} = 14 \times 14 + 13 \times 13 \times 6 + 10 = 1220 \tag{6.13}$$

Where the acronym O.W. stands for **Our Work**.

6.5.2 Results

In the following part are reported the results in terms of energy consumption, latency and area rescaled by the number of neurons, as already did in chapter 2:

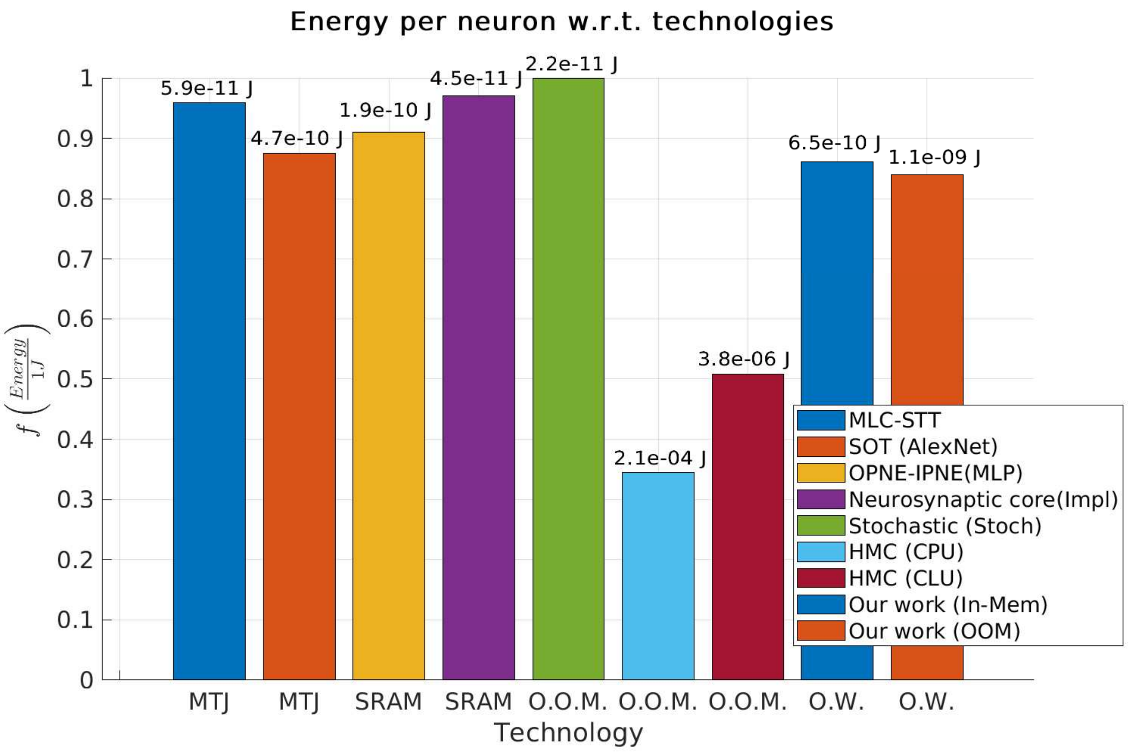


Figure 6.23: Energy comparison: the higher is better. MLC-STT: [15], SOT: [16], OPNE-IPNE: [40], Neurosynaptic core: [26], Stochastic: [28], CPU-CLU: [29].

The energy values obtained for the two architectures are given by:

$$\text{NormEnergy}_{\text{In-Memory}} = \frac{0.79\mu J}{1220} \simeq 647.5pJ$$
$$\text{NormEnergy}_{\underline{\text{OOM}}} = \frac{1.33\mu J}{1220} \simeq 1.09nJ$$

The resulting perfomance is very good, especially for the In-Memory case. The possibility to binarize the network and to transform MAC operations into Xnor-Pop counting sequence, decreases the energy required. By designing a custom memory cell, the energy consumed can be further reduced, because the performance inefficiency coming from the usage of a flip-flop in the model, will be cancelled.

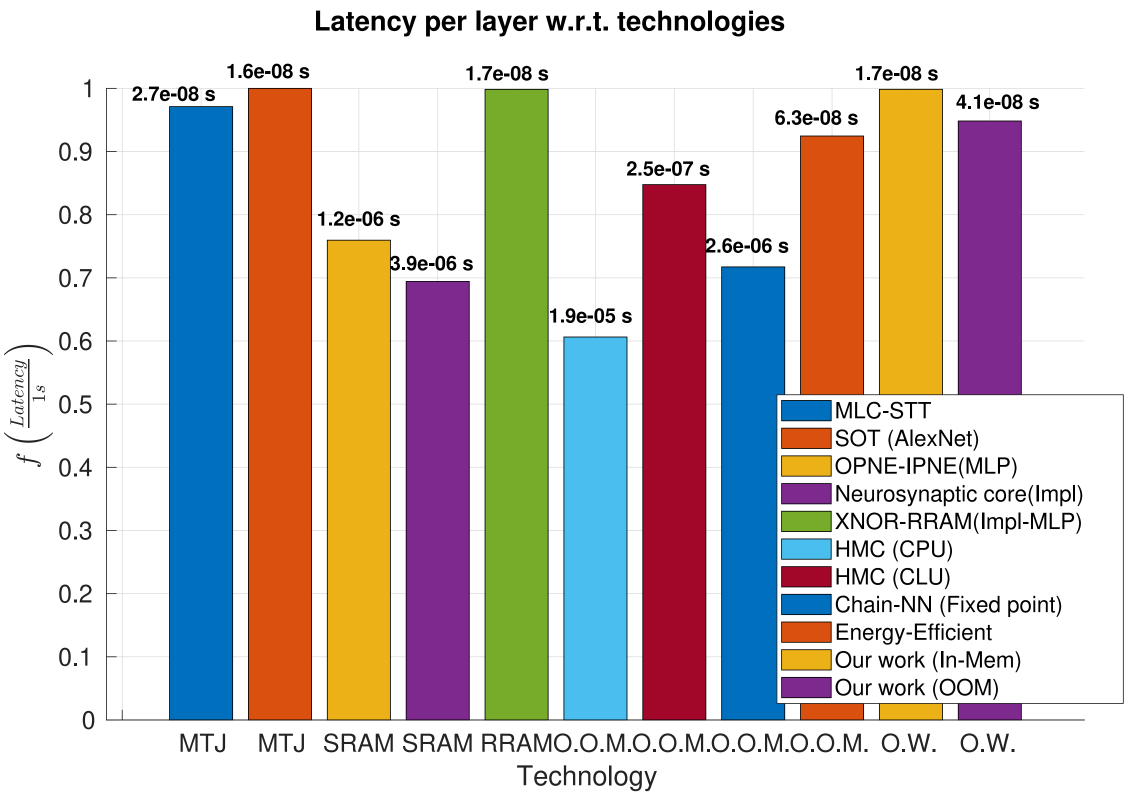


Figure 6.24: Delay comparison: the higher is better. MLC-STT [15], SOT [16], OPNE-IPNE [40], Neurosynaptic core [26], XNOR-RRAM [19], HMC [29], Chain-NN [30], Energy-efficient [31].

Also the delay values obtained are very good compared to the other implementations. They are obtained as:

$$\text{Delay}_{\text{normalized(O.W.(In-Memory))}} = \frac{0.061ms}{3 \times 1220} \simeq 16.7ns$$
$$\text{Delay}_{\text{normalized(O.W.(\underline{OOM}))}} = \frac{0.15ms}{3 \times 1220} \simeq 40.98ns$$

The parallelization of the Xnor-Pop procedure allows to reach very high efficiency in terms of latency, obtaining results that are comparable with RRAM implementations which, by their nature, are very fast.

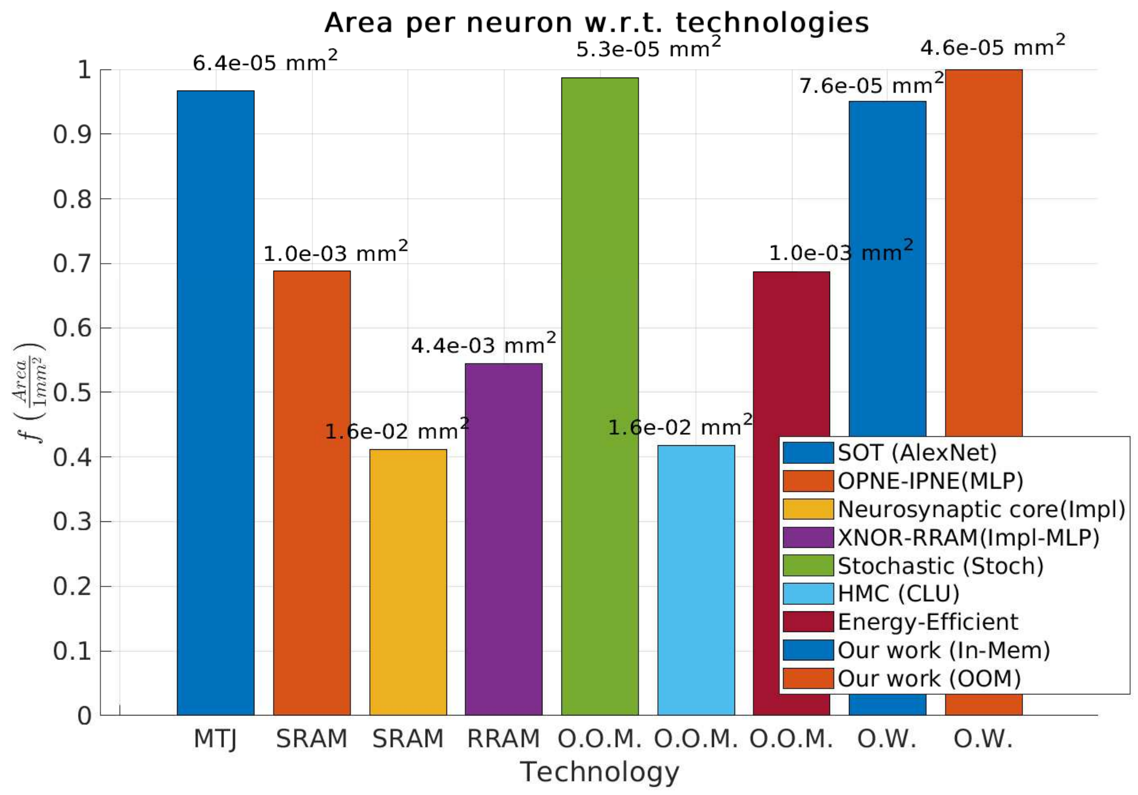


Figure 6.25: Area comparison: the higher is better. SOT [16], OPNE-IPNE [40], Neurosynaptic core [26], XNOR-RRAM [19] (MLP), Stochastic [28], HMC [29], Energy-efficient [31]

$$\text{Area}_{\text{Normalized(O.W.(In-Memory))}} = \frac{0.0923mm^2}{1220} \simeq 75.6 \times 10^{-6}mm^2$$
$$\text{Area}_{\text{Normalized(O.W.(\underline{OOM}))}} = \frac{0.0564mm^2}{1220} \simeq 46.2 \times 10^{-6}mm^2$$

In this last case, area performance in the O.W. In-Memory case reaches a value which is comparable with SOT In-Memory architecture [16] and Stochastic [28] cases (in fact, this last one has a similar computation complexity, since multiplication is performed by an AND gate and the sum by a multiplexer). The O.W. OOM case reaches the best resulting area, because of its simplicity and serialization.

Chapter 7

Conclusions and future work

As already discussed, the In-Memory architecture allows to reduce the Von Neumann's bottlenecks. In general, by increasing the sizes of the neural network, by choosing a deeper model, the dimensions of the circuit increase and, consequently, power consumption/area of both solutions. The In-Memory architecture has a big advantage respect to OOM counterpart: in Synthesis & Place&Route chapter, the estimations represent the worst case values, since the XNOR-Unit part & Pop-Counting can be realized inside a memory array, without employing discrete gates and flip flops, which are far more complicated than a custom memory cell. The power reports show that for an higher dimensionality, the most important power contribution is given by the registers: by designing a custom memory cell, it is possible to reduce this drawback.

7.1 Future work

New pop-counting design Regarding the pop-counting unit, it is possible to optimize the design for the In-Memory part considering the following equations:

$$\begin{aligned} Pop-Counting &= \#1s - \#0s \\ Pop-Counting &= \#1s - (length(Word) - \#1s) \\ Pop-Counting &= 2\#1s - length(Word) \end{aligned} \tag{7.1}$$

It is sufficient to count the number of ones inside the word, that can be performed by a chain of half-adders, instead of full-adders. A circuit that can perform this operation is the following:

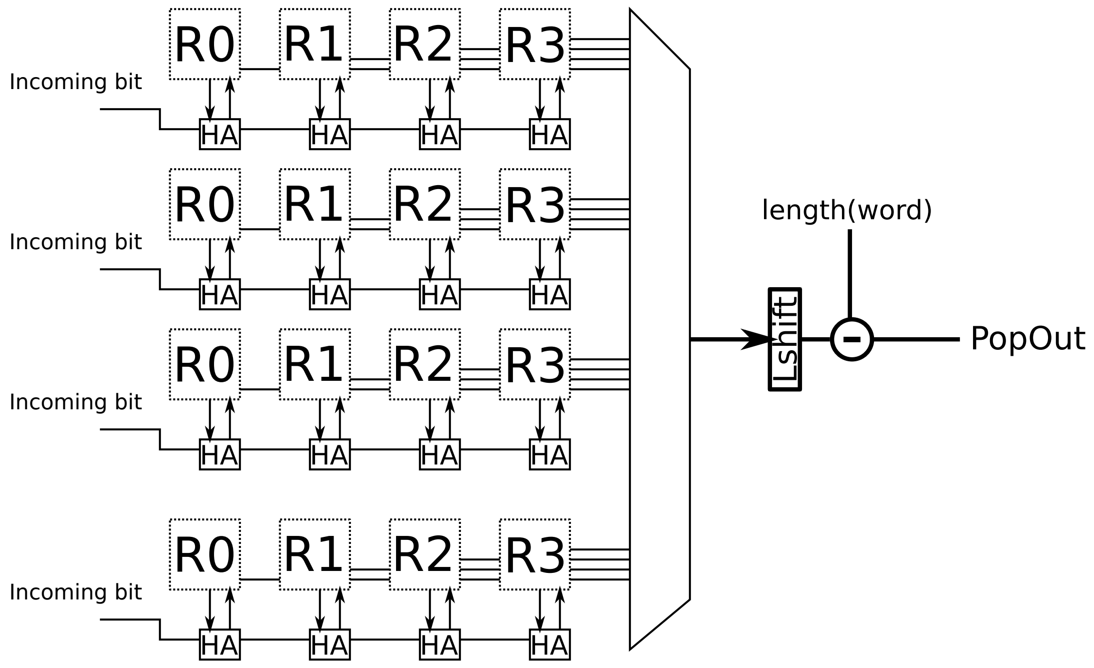


Figure 7.1: Modified pop-counting circuit for the In-Memory architecture.

The reduction in terms of logic gates used is equal to 5/2, since FA contains 5 logic gates and HA only two.

Beyond-CMOS technology By employing Beyond-CMOS technologies, it is possible to further improve the performance: resistive-based technologies, such as MTJ, RRAM etc can be used to realize the XNOR-Unit and pop-counting parts.

Bibliography

[1] Wikipedia contributors. Alexnet — Wikipedia, the free encyclopedia, 2019. [Online; accessed 25-February-2019].

[2] Wikipedia contributors. Cifar-10 — Wikipedia, the free encyclopedia, 2019. [Online; accessed 25-February-2019].

[3] Wikipedia contributors. Imagenet — Wikipedia, the free encyclopedia, 2019. [Online; accessed 25-February-2019].

[4] Wikipedia contributors. Mlp — Wikipedia, the free encyclopedia, 2018. [Online; accessed 25-February-2019].

[5] Wikipedia contributors. Mnist database — Wikipedia, the free encyclopedia, 2019. [Online; accessed 25-February-2019].

[6] Wikipedia contributors. Tunnel magnetoresistance — Wikipedia, the free encyclopedia, 2018. [Online; accessed 6-November-2018].

[7] Wikipedia contributors. Stt — Wikipedia, the free encyclopedia, 2018. [Online; accessed 25-February-2019].

[8] Wikipedia contributors. Artificial neural network — Wikipedia, the free encyclopedia, 2018. [Online; accessed 11-October-2018].

[9] Yash Akhauri. Binary neural networks. https://software.intel.com/en-us/articles/binary-neural-networks, Aug 2018.

[10] Mazur. A step by step backpropagation example, Nov 2017.

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

[12] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[13] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[14] Haruyoshi Yonekawa, Shimpei Sato, and Hiroki Nakahara. A ternary weight binary input convolutional neural network: Realization on the embedded processor. In *2018 IEEE 48th International Symposium on Multiple-Valued Logic (ISMVL)*, pages 174–179. IEEE, 2018.

[15] Yu Pan, Peng Ouyang, Yinglin Zhao, Wang Kang, Shouyi Yin, Youguang Zhang, Weisheng Zhao, and Shaojun Wei. A multilevel cell stt-mram-based computing in-memory accelerator for binary convolutional neural network. *IEEE Transactions on Magnetics*, (99):1–5, 2018.

[16] Deliang Fan and Shaahin Angizi. Energy efficient in-memory binary deep neural network accelerator with dual-mode sot-mram. In *Computer Design (ICCD), 2017 IEEE International Conference on*, pages 609–612. IEEE, 2017.

[17] Sumit Dutta, Saima A Siddiqui, Felix Buttner, Luqiao Liu, Caroline A Ross, and Marc A Baldo. A logic-in-memory design with 3-terminal magnetic tunnel junction function evaluators for convolutional neural networks. In *2017 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pages 83–88. IEEE, 2017.

[18] Feng Zhang, Dong-Yu Fan, Qi-Peng Lin, Qiang Huo, Yun Li, Lan Dai, Cheng-Ying Chen, and Hai-Hua Shen. The application of non-volatile look-up-table operations based on multilevel-cell of resistance switching random access memory. In *VLSI Design, Automation and Test (VLSI-DAT), 2018 International Symposium on*, pages 1–4. IEEE, 2018.

[19] Xiaoyu Sun, Shihui Yin, Xiaochen Peng, Rui Liu, Jae-sun Seo, and Shimeng Yu. Xnor-rram: A scalable and parallel resistive synaptic architecture for binary neural networks. *algorithms*, 2:3, 2018.

[20] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. Magic-memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(11):895–899, 2014.

[21] SR Nandakumar, Manuel Le Gallo, Irem Boybat, Bipin Rajendran, Abu Sebastian, and Evangelos Eleftheriou. Mixed-precision architecture based on computational memory for training deep neural networks. In *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*, pages 1–5. IEEE, 2018.

[22] Wei Wang, Yang Li, Ming Wang, Lingfei Wang, Qi Liu, Writam Banerjee, Ling Li, and Ming Liu. A hardware neural network for handwritten digits recognition using binary rram as synaptic weight element. In *Silicon Nanoelectronics Workshop (SNW), 2016 IEEE*, pages 50–51. IEEE, 2016.

[23] Chunmeng Dou, Wei-Hao Chen, Yi-Ju Chen, Huan-Ting Lin, Wei-Yu Lin, Mon-Shu Ho, and Meng-Fan Chang. Challenges of emerging memory and memristor based circuits: Nonvolatile logics, iot security, deep learning and neuromorphic computing. In *ASIC (ASICON), 2017 IEEE 12th International Conference on*, pages 140–143. IEEE, 2017.

[24] Kota Ando, Kodai Ueyoshi, Kazutoshi Hirose, Kentaro Orimo, Shinya Takamaeda-Yamazaki, Masayuki Ikebe, Tetsuya Asai, Masato Motomura, Haruyoshi Yonekawa, Shimpei Sato, et al. In-memory area-efficient signal streaming processor design for binary neural networks. *Sign*, 21:A22, 2017.

[25] Kanji Otsuka and Yoichi Sato. Deep learning consideration with novel approach-look-up-table based processing conjugated memory. In *Electronics Packaging and iMAPS All Asia Conference (ICEP-IAAC), 2018 International Conference on*, pages 152–156. IEEE, 2018.

[26] Paul Merolla, John Arthur, Filipp Akopyan, Nabil Imam, Rajit Manohar, and Dharmendra S Modha. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pages 1–4. IEEE, 2011.

[27] Lei Jiang, Minje Kim, Wujie Wen, and Danghui Wang. Xnor-pop: A processing-in-memory architecture for binary convolutional neural networks in wide-io2 drams. In *Low Power Electronics and Design (ISLPED, 2017 IEEE/ACM International Symposium on*, pages 1–6. IEEE, 2017.

[28] Vincent T Lee, Armin Alaghi, John P Hayes, Visvesh Sathe, and Luis Ceze. Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 13–18. European Design and Automation Association, 2017.

[29] Palash Das, Shivam Lakhotia, Prabodh Shetty, and Hemangee K Kapoor. Towards near data processing of convolutional neural networks. In *VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), 2018 31st International Conference on*, pages 380–385. IEEE, 2018.

[30] Shihao Wang, Dajiang Zhou, Xushen Han, and Takeshi Yoshimura. Chain-nn: An energy-efficient 1d chain architecture for accelerating deep convolutional neural networks. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1032–1037. IEEE, 2017.

[31] Yizhi Wang, Jun Lin, and Zhongfeng Wang. An energy-efficient architecture for binary weight convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(2):280–293, 2018.

[32] Ujjwalkarn. A quick introduction to neural networks. https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/, Aug 2016.

[33] Kdnuggets - using topological data analysis to understand the behavior of convolutional neural networks. https://www.kdnuggets.com/2018/06/topological-data-analysis-convolutional-neural-networks.html.

[34] Wikipedia contributors. Convolutional neural network — Wikipedia, the free encyclopedia, 2018. [Online; accessed 24-September-2018].

[35] Ujjwal Karn. An intuitive explanation of convolutional neural networks. *The Data Science Blog*, 2016.

[36] Lei Deng, Peng Jiao, Jing Pei, Zhenzhi Wu, and Guoqi Li. Gxnor-net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework. *Neural Networks*, 100:49–58, 2018.

[37] Amar Budhiraja. Learning less to learn better dropout in (deep) machine learning. https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5, Dec 2016.

[38] Davi Frossard. Vgg in tensorflow. https://www.cs.toronto.edu/ frossard/-post/vgg16/, Jun 2016.

[39] T. Bertaud, D. Walczyk, M. Sowinska, D. Wolansky, B. Tillack, G. Schoof, V. Stikanov, C. Wenger, S. Thiess, T. Schroeder, and et al. (invited) hfo2-based rram for embedded nonvolatile memory: From materials science to integrated

1t1r rram arrays. *ECS Transactions*, 50(4):21–26, 2013.

[40] Kota Ando, Kodai Ueyoshi, Kentaro Orimo, Haruyoshi Yonekawa, Shimpei Sato, Hiroki Nakahara, Shinya Takamaeda-Yamazaki, Masayuki Ikebe, Tetsuya Asai, Tadahiro Kuroda, et al. Brein memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 tops at 0.6 w. *IEEE Journal of Solid-State Circuits*, 53(4):983–994, 2018.

[41] S Agatonovic-Kustrin and R Beresford. Basic concepts of artificial neural network (ann) modeling and its application in pharmaceutical research. *Journal of pharmaceutical and biomedical analysis*, 22(5):717–727, 2000.

[42] The number of hidden layers. https://www.heatonresearch.com/2017/06/01/hidden-layers.html.

[43] Frank Gaillard. Batch size (machine learning) — radiology reference article.

[44] Jaron Collis. Glossary of deep learning: Batch normalisation-deeper learning-medium, Jun 2017.

[45] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[46] Wikipedia contributors. Backpropagation — Wikipedia, the free encyclopedia, 2019. [Online; accessed 25-March-2019].

[47] Wikipedia contributors. Resistive random-access memory — Wikipedia, the free encyclopedia, 2018. [Online; accessed 6-November-2018].

[48] Knaji Otsuka and Yoichi Sato. High speed, flexible, robust and low power processing approach. In *Microsystems, Packaging, Assembly and Circuits Technology Conference (IMPACT), 2015 10th International*, pages 38–41. IEEE, 2015.

[49] Rajaraman Ramanarayanan, Sanu Mathew, Vasantha Erraguntla, Ram Krishnamurthy, and Shay Gueron. A 2.1 ghz 6.5 mw 64-bit unified popcount/bitscan datapath unit for 65nm high-performance microprocessor execution cores. In *VLSI Design, 2008. VLSID 2008. 21st International Conference on*, pages 273–278. IEEE, 2008.

[50] Basis of lenet 5 hyper parameters selection. https://stats.stackexchange.com/questions/269874/basis-of-lenet-5-hyper-parameters-selection.

[51] DingKe. Neural network playground. https://github.com/DingKe/nn_playground, 2018.