POLITECNICO DI TORINO

Master degree in electronics engineering

Master Thesis

Design and implementation of neural networks on FPGA based on model compression analysis

Supervisors

Prof. Luciano LAVAGNO

Prof. Marisa LOPEZ-VALLEJO

Candidate

Gianmarco CANZONIERI

2020

Summary

The increasing amount of data available nowadays has made the use of automatic learning algorithms, also known as machine learning, spread. Machine learning is widely used for applications like speech recognition, natural language processing or robotics. The most popular technique used for these purposes is neural networks. These models require a great amount of computation capacity and until now, *GPUs* have mainly covered these computations. Recently, field programmable gate arrays (*FPGAs*) are becoming more common within these applications. The main difference between *GPUs* and *FPGAs* is that the latter offer the user the possibility of designing specific hardware instead of using a fixed architecture. Also, *FPGAs* offer a great parallel computation capacity as well as low power consumption compared with *GPUs*.

For this project, it has been decided to analyze the model compression for a neural network in order to understand how model compression can influence the accuracy and as a consequence the improvement in needed hardware and memory constrain. This is because computing edge neural network has fixed constrain that is fundamental to respect. After the extensive model compression operated in *MATLAB*, there is a design and implementation details part of an optimized neural network on *FPGA*. We will study the benefits the *FPGA* provides, paying special attention to the resources utilization, throughput and accuracy of the algorithm. Experimental results will be carried out on a Xilinx *FPGA*, in particular, we will use the Zynq-7000 SoC ZC706 Evaluation Kit from Xilinx. This board contains an XC7Z045 FPGA.

KEYWORDS

Neural network, model compression, computing edge, *FPGA*, Vivado, VHDL.

Acknowledgements

Prima di procedere con la trattazione, vorrei dedicare qualche riga a tutti coloro che mi sono stati vicini in questo percorso di crescita personale e professionale.

Un sentito grazie alla mia correlatrice la Professoressa Marisa Lopez Vallejo dell'Università Politecnica di Madrid che durante il periodo iniziale di tesi mi è stata accanto e supportato.
Un sentito grazie al mio relatore Luciano Lavagno per la sua infinita disponibilità e tempestività nelle risposte. Grazie per avermi fornito ogni materiale utile alla stesura dell'elaborato. Ringrazio il supporto dei miei genitori, non sarei mai potuto arrivare fin qui.

Ringrazio la mia ragazza che ha mi ha aiutato a superare gli ostacoli che sono sorti durante il percorso sopratutto negli ultimi mesi durante il lockdown forzato a Madrid.

Ringrazio i miei colleghi per essermi stati accanto in questo periodo intenso e per gioire, insieme a me, dei traguardi raggiunti.

Grazie a tutti, senza di voi non ce l'avrei mai fatta.

Gianmarco Canzonieri

Table of Contents

List of Tables

List of Figures

Acronyms

AI
Artificial Intelligence

ANN
Artificial Neural Network

BRAM
Block Random Access Memory

CNN
Convolutional Neural Network

DSP
Digital Signal Processing

FF
Flip Flop

FPGA
Field Programmable Gate Array

GPU
Graphics Processing Unit

LUT
Look-Up Table

MAC
Multiply Accumulate

MUX

Multiplexer

RAM

Random Access Memory

ROM

Read Only Memory

ReLU

Rectify Linear Unit

SOC

System On Chip

VHDL

Hardware Description Language

SGD

Stochastic Gradient Descend

PE

Processing Element

Chapter 1

Introduction

This chapter gives a formal definition of the machine learning approach and which is the relationship between this last one and the neural network. After that we will treat about the neural network concept inspiration based on human brain. In the last part there is a comparison of neural network approach and the advantages respect to normal machine learning solutions analyzing the areas in which is better using the one respect to the other.

1.1 Machine learning concept and algorithms

The Artificial Intelligence is still one of the most exciting challenges for the future which was described by John McCarthy as:

- *"The science and engineering of creating intelligent machines that have the ability to achieve goals like humans do".*

The improvement at the quantity of data available and improvement at hardware level give the possibility grow up one subset of Artificial Intelligence called **machine learning**.

There are different machine learning definitions but two of the most important and well noted are the following ones:

- *Field of study that gives computers the ability to learn without being explicitly programmed [Arthur Samuel, 1959].*
- *A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E [Tom Mitchell, 1998].*

It is possible to note that the first definition is a general concept of the machine learning approach, while the second one is more specific and it is applied to modern machine learning problems.

There is one very important consideration to do about the word "experience" in the definition of Tom Mitchell (nowadays called training), because it explains well how for the machine learning approach one machine thanks to "training" has the possibility to "learn" from the data and build a model that can be used with different data.

This way it is very different from the classical approach in which the model is built for a specific application, it is not defined by the processed data.

Figure 1.1 shows this approach in which in a classical approach there are the inputs that are data inputs and a pre-built model and the outcome of the model are the data outputs.

In case of machine learning there are the data inputs also called samples, the output, called label and we want that the machine creates a model that it will predict also the correct output-data when it will use data inputs different from the used as samples.

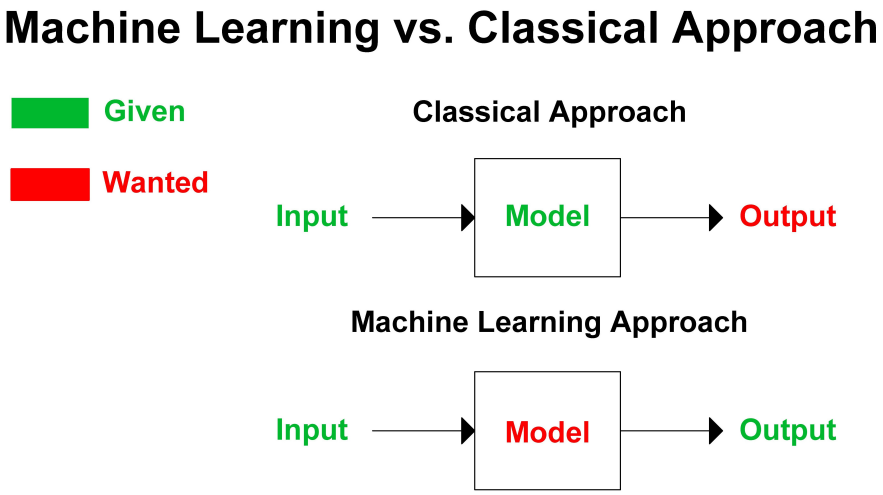

Figure 1.1: Classical model vs Machine learning model.

1.1.1 Machine learning categories

The machine learning algorithms can be divided into three big categories:

- **Supervised learning**: it is based on the fact that a set of known correct outputs are given so it is possible build a predictive model starting from the output and input. It is used in problems concerning classification and regression. The evaluation phase in this case is quantitative and the training process is mandatory.
- **Unsupervised learning**: it is used to group or interpret data based only on input data, the correct output data are not given. Its particular scope is identify the pattern/structure of the data. The evaluation phase in this case is just qualitative.
- **Reinforcement learning**: machines learn what to do based on a certain environment.

There are several types of machine learning algorithms for the three different categories previous explained, each of them is used for different specific situations. For all types of algorithms, some features will be defined and those features will be the inputs of the algorithm, to allow the computer to produce the output depending on the value of the different features.

Figure 1.2 summarizes the different possible machine learning algorithms.

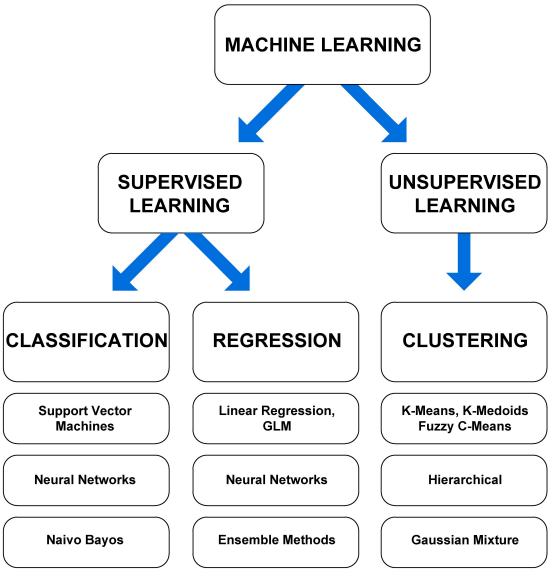

Figure 1.2: Different types of machine learning algorithms.

1.1.2 Neural networks concept and inspiration

There are two types of machine learning techniques which are inspired by the human-brain, one is called *spiking computing*, the second one is called *neural networks* as shown in figure 1.4.

An adult human brain is formed by between 85 and 86 billion neurons. A neuron is formed by a cell body, dendrites and an axon and it is an electrically excitable cell that receives, processes and transmits information through electrical and chemical signals. The information is received via the dendrites, the processing of these inputs is done in the cell body and the output of the processing goes through the axon.

The communication signals between neurons occur through specialized connections called synapses. Synapses are found between the axon of a neuron and the dendrites of the next neuron. The human brain is estimated to have about 10^{14} synapses. One important characteristic about synapses is that they increase or decrease activity in the target neuron, this means that the electrical value contained in the neuron is scaled in the synapse. This scaling factor can be called *weight*.

The way that the brain is believed to learn is by changes in those weights of the synapses and different weights result in different output values, as the inputs of the dendrites will change because of the weights. It is important to notice that the idea of how the brain learns is just through changes in the weights and not by changes in the organization of the brain.

When talking about neural networks, the organization of the brain can be mimic by the program, this means that in brain-inspired algorithms the program should not change to allow the network to learn; instead, some parameters, called weights, will be modified and it will result in the learning of the program [1].

The important consideration is the possibility to use *"one learning algorithm"* [2], in which the same system learns what it has to "learn" from the data.

Figure 1.3 shows in a schematic way which is the working flow for a brain system.

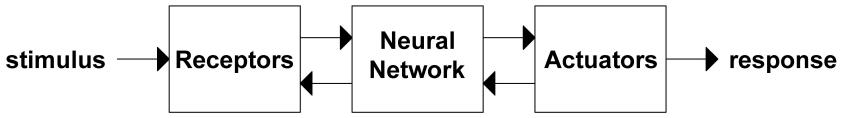

Figure 1.3: Example of working for brain system.

It is possible to distinguish three different types of blocks:

- **Receptors**: they convert the external stimulus to an electric impulse for the neural network.
- **Neural network**: it receives information from the receptors and it makes the decisions.
- **Actuators**: they convert the electric impulse generated by the neural networks into actions to the external environment.

In table 1.1 it is possible to see that the strong point of the human brain is that it presents a very low energy efficiency with respect to average electronic devices and it means the human brain spends less energy for doing the same number of operations per second.

Table 1.1: Comparison between the human brain and average electronic device according to efficiency and latency.

	$Efficiency[\frac{J}{s \cdot ops}]$	$Latency[ms]$
Human brain	10^{-16}	1
Average electronic device	10^{-6}	10^{-6}

Spiking computing takes inspiration from the fact that the communication on the dendrites and axons are spike-like pulses and that the information being conveyed is not just based on a spike's amplitude. Instead, it also depends on the time when the pulse arrives and that the computation that happens in the neuron is a function of not just a single value but the width of the pulse and the timing relationship between different pulses [3].

Neural networks take their inspiration from the notion that a neuron's computation involves a weighted sum of the input values. These weighted sums correspond to the value scaling performed by the synapses and the combination of those values in the neuron.

Figure 1.4 puts in evidence which are the relationships between the different AI fields previous discussed.

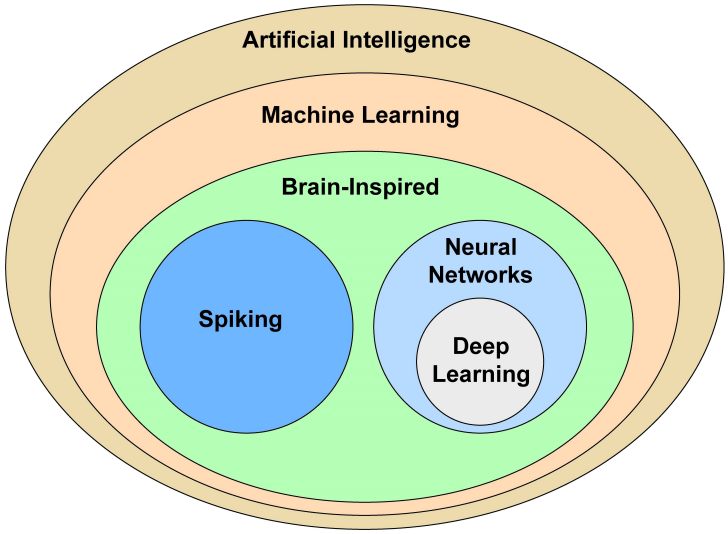

Figure 1.4: Summary of different fields of AI [1].

Our work is based on the neural network structure.

1.1.3 Neural network vs. Machine learning

It is important to explain in which cases it is better to use the neural network algorithm with respect to the normal machine learning algorithms. To do so it is fundamental to understand which are the advantages and possible limitations in using them.

In table 1.2 there is a summary of which are the advantages and drawbacks of neural networks design.

Table 1.2: Comparison between machine learning and neural network design.

Design	*Advantages*	*Drawbacks*
Neural network	• better accuracy • feature numbers • extracting features	• training time • execution time • not math model
Machine learning	• training time • execution time • math model	• worse accuracy • feature numbers • extracting features

In Table 1.2 it is possible to see that neural network design works well when the number of features is high, contrary to machine learning design where a lots of features cannot be used. Neural network have also a better accuracy and an easier possibility to extract the features from the sample [2].

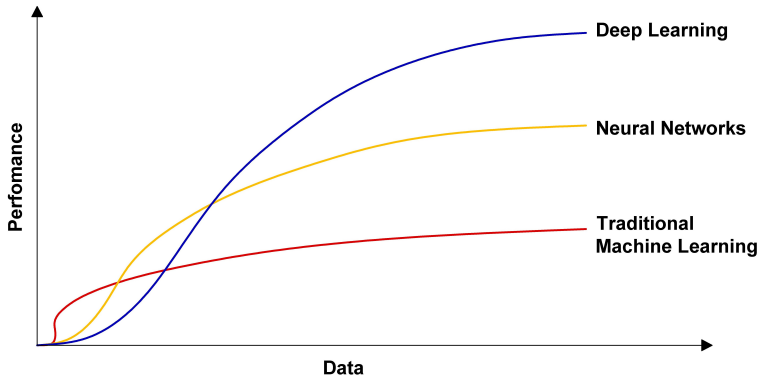

Figure 1.5: Performance deep learning comparison.

Chapter 2

Neural networks overview

This chapter is an overview on neural networks starting from the mathematical description of a single neuron in the first part, and focusing on the most commonly used activation functions in the second part. After that, there is an analysis of the two most important neural network architectures: *fully connected (FC)* and *convolutional neural network (CNN)*. The analysis of these types of architecture is done in order to understand which are the parameters that has more impact at hardware level like the number of weights and the needed interconnections by each one. At the end of the comparison the *FC* neural network shows different types of problems linked with the huge number of weights respect to *CNN*, as a consequence of this the number of performed operations is higher (the number of operations depends on number of weights in a *FC*). For this reason the next chapters are based on optimizing this problem by applying different techniques based on model compression.

2.1 Neuron description

The neuron is the fundamental part of neural networks and it is composed of three base elements:

1. Group of connections (weights) each of them has a particular value that can be positive or negative.
2. Adder function that has like inputs the weighted connections coming from different neurons and it produces a linear combination of the inputs.
3. Activation function used to limit the output amplitude in a particular range. The most common range are [0,1] and [-1,1].

So it is possible to formalize the function that each neuron does in the formula 2.1:

$$output_j = f_{activation}\left(\sum_{i=0}^{N}(input_i \cdot weight_{ij}) + bias\right) \tag{2.1}$$

where:

- $output_j$ is the output of the j neuron.
- $input_i$ is the input signal that comes from the i-th neuron for instance, these values can be pixels of an image, sampled amplitudes of an audio wave or the numerical representation of the state of some system or game.
- $weight_{ij}$ is the weight that determines the connection between i neuron linked to j neuron. They are the learnable parameter, it means that they are the values that are changeable during training.
- N is the total number of neurons linked to j neuron.
- *bias's* aim is fine tuning of the neuron operation.
- $f_{activation}$ is a non linear activation function that determines if the neuron output is higher than a specific threshold then it is called activated, in other case it is not activated.

Figure 2.1 shows in a schematic way the neuron behavior and its analogy with the brain neuron.

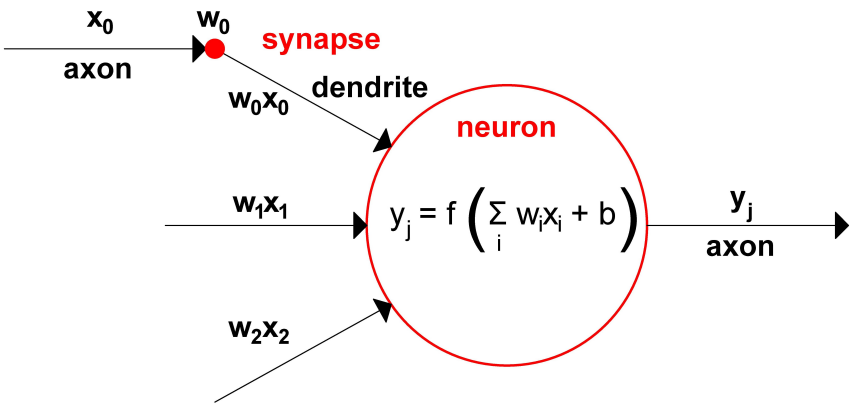

Figure 2.1: Neuron schematic description [4].

2.1.1 Activation functions

As we have seen, the neurons of a neural network perform a non-linear function to the weighted sum of inputs, which is called activation function.

The reason of non-linearity is because if it uses a linear-function the output of the function will not be confined between any range as a consequence data management is more difficult.

The most common functions with their respective equations and derivatives are shown in table 2.1.

Table 2.1: Examples of activation functions.

Function	Equation	Derivative
Sigmoid	$f(x) = \dfrac{1}{1+e^{-x}}$	$f'(x) = f(x) \cdot (1 - f(x))$
Tanh	$f(x) = \dfrac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$	$f'(x) = 1 - f(x)^2$
ReLU	$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases}$

Other important key metrics for activation functions are their derivatives (also defined as a slope) and the trend of the functions (monotonic or not function).

The motivations behind them are due to the fact that during training a neural network algorithm, the derivative of the activation function is needed to update the new weights in order to know in which direction and quantity the change must be done.

In figure 2.2 the graphs of activation functions are shown and in figure 2.3 the derivative graphs of activation functions are shown.

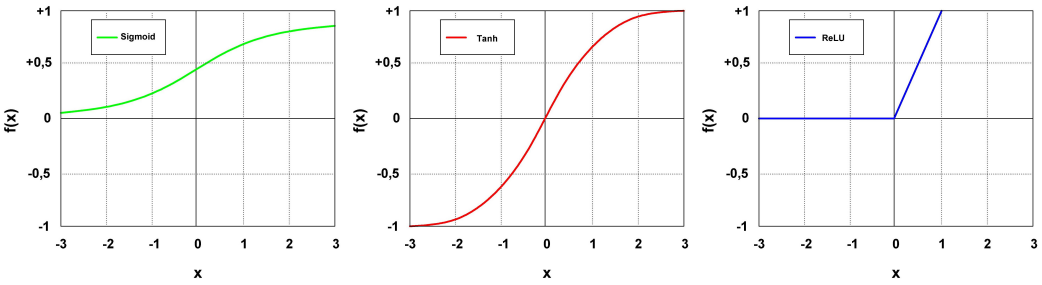

Figure 2.2: Common activation functions. *Sigmoid, Tanh and ReLU.*

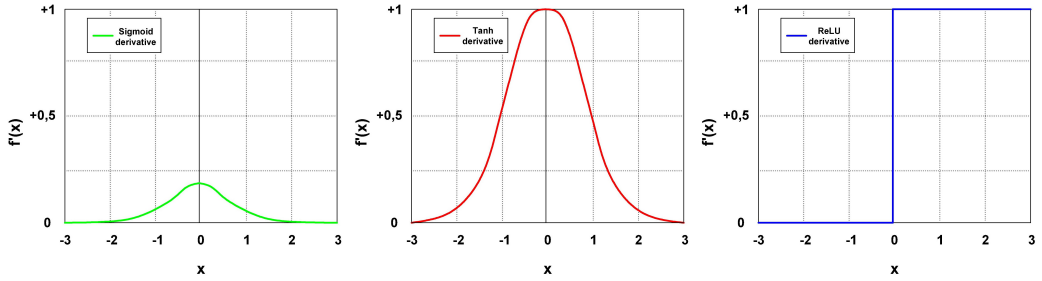

Figure 2.3: Derivatives of Activation functions.

In the following list there is a summary of the most important properties of each activation function shown in table 2.1 [5]:

1. *Sigmoid function*:

   - It limits the activation output in the range to [0,1].
   - It is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 to 1.
   - It is a differentiable and monotonic function but its derivative function is not monotonic.
   - It has difficult hardware implementation.
   - It can cause a neural network to get stuck at the training time due to the *vanishing gradient* phenomenon.

2. *Tanh function*:

   - It limits the activation output in the range to [-1,1].
   - Its negative inputs will be mapped to strongly negative values and the zero inputs will be mapped near zero in the *Tanh* graph.
   - It is a differentiable and monotonic function but its derivative function is not monotonic.
   - It has difficult hardware implementation.

3. *ReLU function*:

   - It limits the activation output in the range to [0,infinity).
   - Its function output is zero when the input is lower than zero and its output is equal to the input when the input is higher or equal to zero.
   - It is a differentiable and monotonic function and its derivative function is monotonic.
   - It is computationally less expensive with respect to the others and it has easy hardware implementation.
   - It has the problem that all inputs lower than zero are mapped to 0 since the first layers ot the network, it means that the ability of the model to properly fit or train from the data decreases.

2.2 Artificial neural networks

The neurons inside a neural network can be connected in various ways, each neuron with the others. Normally the neurons are divided into different layers, being the minimum number of layers 2, one for the inputs and one for the output results. When the neural network is composed by more than two layers (the input and the output layer), the other layers are called *hidden layers*.

In figure 2.4 there is an example of schematic representation of a neural network with one hidden layer composed by four neurons, an input layer composed by three inputs, called *features*, and an output layer composed by two neurons.

The connection of various neurons is thanks to specific connections, each of them has an associated *weight*. In order to unify the terminology, the standard general name given to the weights is W_{ij} where i represents the number of the neuron in the previous layer that it is linked to the j neuron in the current layer.

For instance in figure 2.4 the weight W_{11} connects the first neuron in layer 1 and the first neuron in layer 2.

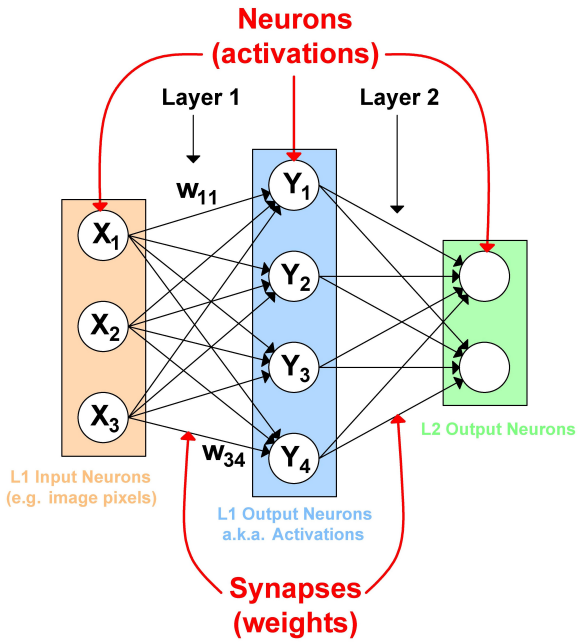

Figure 2.4: Example of artificial neural network [4].

In neural networks, each layer has an extra input besides the features or the data from the previous layer. This extra input is called the *bias*. As any other input, there is a weight associated with the bias.

As stated before, each neuron performs a non-linear function that causes the neuron to generate an output only if the weighted sum crosses some threshold. When incorporating the bias input and weight, that threshold can be changed, being shifted left or right depending on the value of the weight associated with the bias input.

The bias also provides a way to better fit the data as it is a constant that can be changed and this constant will modify the prediction to achieve a better performance. It allows, therefore, a fine tuning of the neuron operation [1].

A deep neural network (*DNN*) is a neural network with a number of the hidden layers greater than three. Today, the typical number of network layers used in deep learning ranges from five to more than a thousand [2].

2.2.1 Deep learning history and application areas

DNNs are capable of learning from high level features with more complexity and abstraction than shallower neural networks. A *DNN* learns about simple low level features in the first hidden layers, like particular lines patterns, while in the final hidden layers they group these features to form higher level features, for example a particular shape [1].

A deep neural network was used in 1980 for the first time for a hand-written digit recognition checker by a bank [6]. The real interest by a company to apply *DNN* comes from early 2000 for different reasons:

1. Existence of big data available for training the network.

2. Hardware improvement that allows to do the training and inferring phases in a reasonable time.

3. Appearance of frameworks developed by deep learning users in order to manage in an easy way the deep learning design. Nowadays the frameworks used for designing using deep learning are a wide range, the most common are *Caffe from Barkeley University, Tensorflow from Google, Pytorch from Facebook* [7]. There are also different frameworks in charge of implementing neural networks in a particular hardware like *FINN* [8].

Deep learning applications are the state of art in several fields because now large datasets from the growth of automation/web are available and usable by engineers and data scientist [1]. The main areas in which deep learning is in common use are the following:

- **Computer vision**, which is the new important scientific field, its aim regards the possibility of machines to capture useful information by videos and images (because they are composed from pixels). Computer vision has also different sub sections like object localization and detection, image segmentation, handwriting recognition, and action recognition.

- **Speech and language recognition** has many related tasks such as machine translation, natural language processing, and audio generation [9]. A vivid example is the *LipNet* system, which was developed using neural network technology by scientists at Oxford University. *LipNet* has become the world's first system that can recognize lip-speech, and not just individual words, but whole sentences [10].

- **Robotics** has always been an important application of machine learning and neural networks concretely. Especially when trying to get robots to mimic human's abilities, neural networks are an obvious choice, as they are brain-inspired algorithms, and are improving considerably the performance in these applications. When talking about quadricopters or drones, neural networks have also been used.
- **Medical space** has been very useful for medical applications for example to recognize patterns in data to study genetic diseases or used with medical images to detect some types of cancer.
- **Banking and insurance companies** want to compute the claims cost just according to the images thanks to the deep learning system pre-trained with a lot of cases of incidence. The same system can also help the bank in *ATM* control abnormal activity [8].

The market of *DNN* is increasing and the prospective in the future follows an exponential growth, like the one showw in figure 2.5 in all three sectors engaged: *Software*, *Services* and *Hardware*.

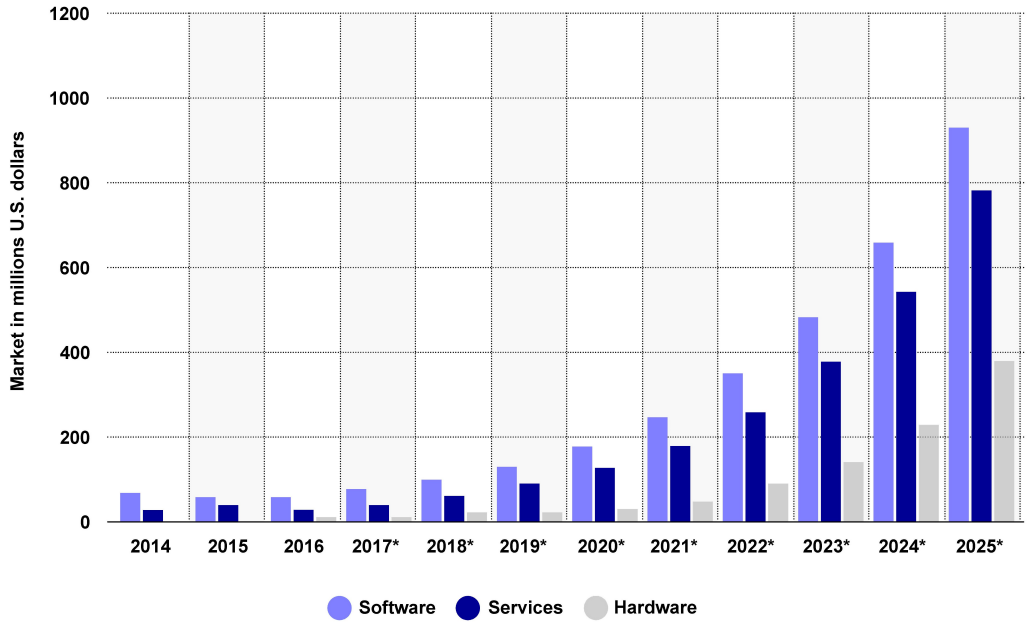

Figure 2.5: Trend in the deep learning market [8].

2.2.2 The mathematical reasons of *DNN*

It is clear, as was described in the subsection 2.2.1, that a *DNN* permits to capture more high level features than shallower neural networks but it is not the only motivation to pass from neural networks to *DNN*.

In [11] the author demonstrates according to mathematical proofs that for some functions very shallow networks require exponentially more circuit elements to compute than do deep circuits.

The author in particular proved that computing the parity of a set of bits requires exponentially many gates, if done with a shallow circuit. On the other hand, if you use deeper circuits it is easy to compute the parity using a small circuit: you just compute the parity of pairs of bits, then use those results to compute the parity of pairs of pairs of bits, and so on, quickly building up to the overall parity.

Deep circuits are more powerful than simpler circuit. So the mathematical reason for *DNNs* is that some computational function that has polynomial complexity with k levels but they have exponential complexity with $k-1$ levels.

At the neural network level this consideration can be translated into the fact that it is better to have a lot of layers with fewer neurons with respect to having fewer layers with a bigger number of neurons.

The problem at the hardware level is that *DNNs* have an important hardware requirement in terms of computational complexity and memory requirements so it is important to find the right trade-off.

2.3 Different neural networks arrangement

An important first division among neural network's arrangements can be done with the following categories:

1. **Feed forward**: this type of *DNN* has no memory because the output is a function just of the inputs, when the inputs change the outputs also change in order to follow the inputs, besides the inputs of a neuron always come from a previous layer.
2. **Recurrent**: this type of *DNN* has memory and in some cases the output depends on their stored values and not only on the input values from a previous layer.

2.3.1 2-D architecture

In previous subsection 2.2 it is explained the possibility to arrange neural network structures in different ways.

Nowadays there are several examples of different architectures:

- **Fully connected (FC)**: it means that every neuron in a specific layer is linked with all neurons in the previous layer. The behavior of a singular neuron is as described in equation 2.1.
- **Sparsely connected (SC)**: it consists in an arrangement close to *FC* but the connections between a particular layer and the previous layer are less than for the *FC* mode, because some of them are cut; usually it is important to profile how data are processed (sensitivity analysis) by the neural network in charge to cut the useless connections [12].
- **Recurrent (RNN)**: as mentioned before, it has memory and in some neurons of this architecture their inputs come from the same neuron, so they report the last memorized value [13].

The *FC* and *SC* belong to the feed forward category. These are used for image processing because they are static in time, so they just need the present value of pixels.

The *RNN*s are used to process videos in which the neural networks learn how to change the value of the pixels on the fly (the pixel values are not static). They are able to discriminate the timeline of the feature.

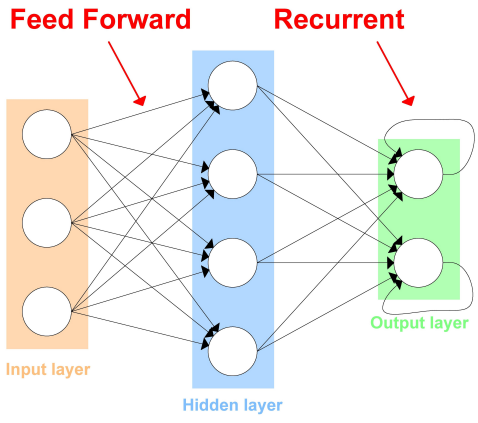

Figure 2.6: Feed foward vs Recurrent architectures [4].

2.3.2 3-D architecture

Another important type of *DNNs* are the convolutional neural networks (*CNN*). They belong to the group of feed foward networks.

The *CNN* respect to *FC* have a 3-D structure as shown in figure 2.7.

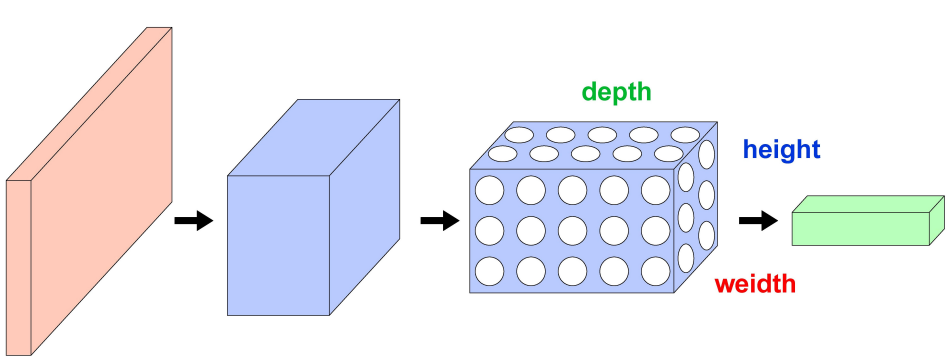

Figure 2.7: Example *CNN* 3-D structure [9].

The main difference is that *CNN* architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network. This is because in the *FC* architecture the weights are independent from each others and they are not shared.

One important implication is that the *FC* architecture does not scale well with the size of images, so it is possible to have a lot of weights and it implies difficult memory management and also some overfitting problems [9].

In table 2.2 there is a comparison between these two types of neural networks in terms of number for weights for a single layer using as example an image extracted from the dataset *CIFAR-10*. Every image in this dataset is composed by a 3-D matrix where every matrix is a part of an RGB color.

The number of the weights for the two different layer configurations can be computed in the following way:

- *FC*: the weights in the *FC* layer can be computed counting the pixels of the image and multiplying them by the number of neurons of the layer.
- *CNN*: the weights in the *CNN* depend on the chosen filter size. In this case not all pixels are connected to a single neuron but an operation of convolution between the filter and the image is done. In the example shown in table 2.2 the filter size is $5 \times 5 \times 3$.

Table 2.2: Comparison between the number of weights in a *FC* layer and a *CNN* layer for $filter_{5\times5\times3}$.

Image dimensions	*# of FC weights*	*# of CNN weights*
$32 \times 32 \times 3$	$3072 \times$ neurons	76
$200 \times 200 \times 3$	$120000 \times$ neurons	76

Another important parameter that it is necessary to take into account is the number of connections between the layers. Figure 2.8 shows an example of the same neural network with two different configurations (they have a different number of connections between the layers).

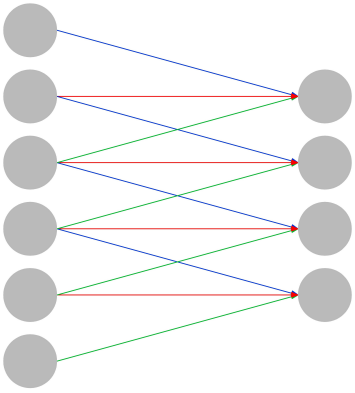

Figure 2.8: Example of different types of connections between *FC* and *CNN* [14].

In the equivalent *FC* structure all neurons in the previous layer are linked with all neurons in the following level. In a possible *CNN* structure each of the four neurons on the right is linked with just 3 neurons in the previous level. Shared weights are the same color.

Table 2.3 shows a summary of the different configurations in terms of number of weights and number of connections between two different layers.

Table 2.3: *FC* vs *CNN* comparison of the architecture showed in figure 2.8.

# of FC weights	*# of CNN weights*	*# of FC conn.*	*# of CNN conn.*
24	3	24	12

In order to avoid the huge quantity of weights and connections the *CNN* uses two different principles with respect to *FC*:

1. **Weights sharing**: according to the receptive field approach [4] that permits in image processing not to scan all pixels in an image, as required for the *FC* architecture, but just a small part of the images is scanned at each time and the size of the scanned part is determined by the receptive field size, also called filter size. The operation done between the filter and input image is a convolution where the filter scans a small portion of the image at each step. The effect of allowing weights sharing is a strong reduction of number of weights.

2. **Local connectivity**: neurons are linked just locally at the neurons of the previous layer. Neurons elaborate the features in a local way. The effect of local connectivity is a strong reduction of the number of connections.

Local connections and weights sharing permit that neurons process in the same way different parts of images, and it is useful because the different parts of the images scanned by the filter contain the same type of information (edges, corner, some part of object).

Figure 2.9 shows a schematic example of the two principles previous discussed.

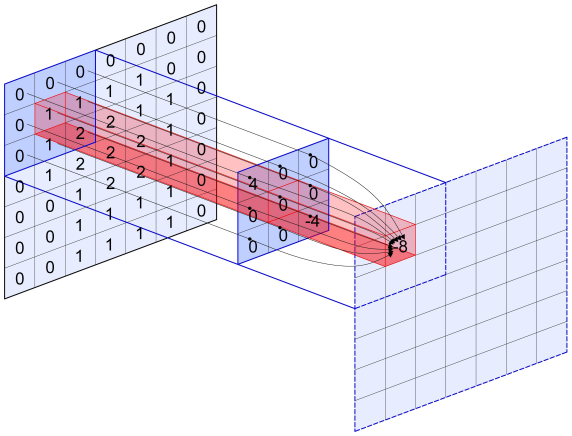

Figure 2.9: Weights sharing and local connectivity of *CNN* [14].

Complete CNN

As shown in figure 2.10 a complete *CNN* architecture is mainly composed of three elements: the convolutional layers, pooling layers and *FC* layers.

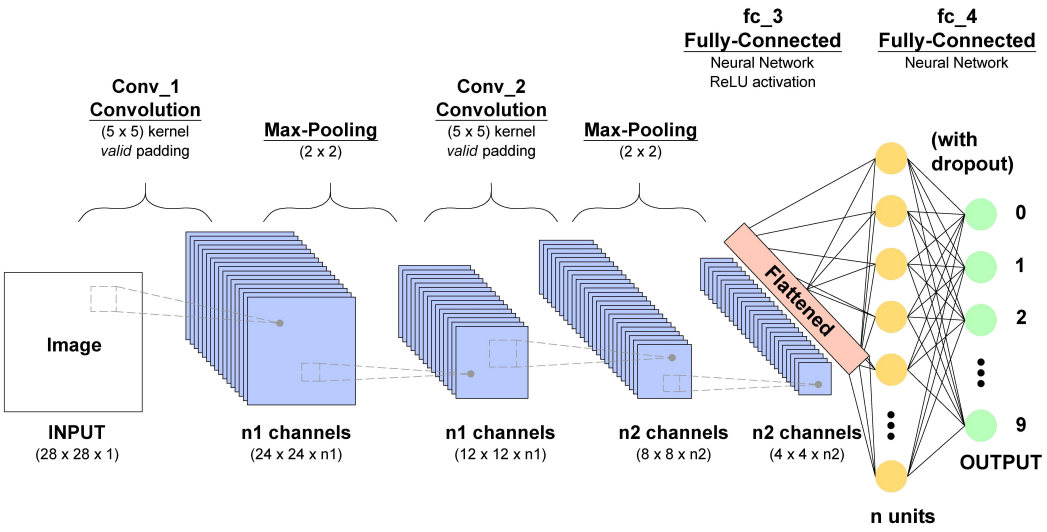

Figure 2.10: Example of convolutional neural network [15].

- Convolutional layers are the layers that are used in order to extract the features in an image. The filters used in the first convolutional layers learn about low level feature like line shape patterns. The filters used in the final convolutional layers learn about high level features like geometric shapes thanks to the input of low level features.
- Pooling layers are used to down-sample the data produced by a convolutional layer in order to extract just a useful part. The average and max functions are the most common for this scope.
- FC layers are necessary in order to do the classification of the features learned in the previous *CNNs* layers.

It is important underline that in a complete *CNN* structure there are also *FC* layers. These ones are problematic for the reason already discussed before about the number of the weights and connections.

Besides *CNN* are used mainly for images but in a general application they cannot be used. From this point of view the *FC* are more flexible but present more important memory constraints for the weights.

In table 2.4 it is possible to see how for some different popular *DNNs* the number of weights for the dense layers (*FC*) is always greater than the number of weights for the convolutional layers.

Besides, it is possible to see that in a dense layer the number of weights and the number of *MACs* is the same, while in a convolutional layer it is not true and it also depends on the input size features.

Table 2.4: Summary of popular DNNs key metrics [1].

Metrics	*LeNet*	*AlexNet*	*VGG*
Input size	28x28	227x227	224X224
Weights (conv)	2.6k	2.3M	14.7M
MACs (conv)	283k	666M	15.3G
Weights (dense)	58k	58.6M	124M
MACs (dense)	58k	58.6M	124M

For this reason in the next chapters we focus on studying possible optimization techniques for *FC* neural networks in terms of model design and model compression.

Chapter 3

Analysis and design of neural networks

This chapter is a focus on standard issues of generic neural network design. In the first part the most important hardware metrics for neural networks are explained and discussed together with possible improvements. A particular attention is given to how neural network parameters like the accuracy are linked with hardware parameters. Based on these key metrics there is a comparison in terms of performance between *FPGA, CPU* and *GPU*; particular focus is given on typical standard architectures used in these different devices. After that the advantages of *FPGA* design are discussed. Finally, some possible techniques are introduced in order to optimize the model design and the hardware implementation. We focus on the design and problems regarding the inference part of neural network architecture.

3.1 Metrics for *DNN* Hardware

Most computations in a neural network are composed by three different operations that are processed in the following way:

- Read data from memory.
- *MAC* operation.
- Write data to memory.

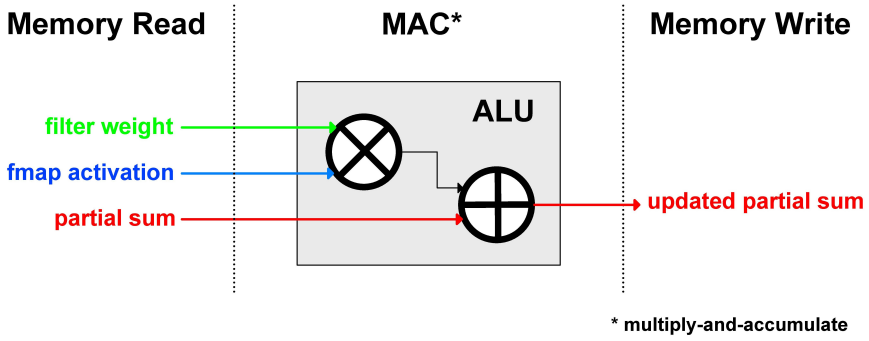

Figure 3.1: Scheduling of operations in a generic neural network [4].

The *MAC* operations are very expensive at hardware level in terms of area, energy and delay and they represent 90% of the computation.

Also data movement in and from memory represents the other bottleneck that it necessary to overcome to get high efficiency.

For instance in *AlexNet*, there are 724 million *MACs*, and nearly 3000 million *DRAM* accesses will be required [1].

It is important to see the parameters shown in figure 3.2 in order to characterize and analyze the design of a generic neural network. These parameters will be used in the computation of metrics to assess a neural network implementation.

Symbol	Description	Unit
W	Is the workload in term of operations needed to compute 1 inference.	GOP
η	Efficient factor that takes in account how many processing elements are active among all processing elements	–
f	Frequency circuit	GHz
P	Number of processing elements	–
C	Is the concurrent factor that consists in how many inference are processed simultaneously	–
E_{ops}	Energy spent for 1 operations of 1 interfence	J
$E_{access\text{-}memory}$	Energy spent to read 1 byte data from memory (SRAM or DRAM)	J/byte
$N_{access\text{-}memory}$	Number of accessed of memory	byte

Figure 3.2: Hardware design parameters of neural networks [16].

In the next section there is a description of the main key metrics in a hardware architecture in order to understand which are the main problems that limit the neural network design and which are possible solutions.

3.1.1 Throughput

Throughput, measured in $[s^{-1}]$, is defined as number of inferences processed per second. We can calculate it with formula 3.1 [16]:

$$Throughput = \frac{f \cdot P \cdot \eta}{W} \tag{3.1}$$

It is important to know that in order to have high throughput it is required:

- HIGH FREQUENCY: possible solutions can be, for example, to reduce the critical path of the singular *MAC* using reduced precision with low bit width or data quantization. Frequency can be bounded also by memory bandwidth. For example in a *DRAM* the read time is 10 times longer than in an *SRAM*, as shown in table 3.1.
- HIGH η: parallel execution is fundamental in order to efficiently use all available processing elements (PEs) and in this sense *FPGA* design has advantage with respect to *CPU* and *GPU*. The η factor can be limited also by memory management since some processing elements could be free but the memory bandwidth cannot permit to use them. To get better η factor different architecture strategies can be used, like for example *near data processing*.
- LOW W: it is important minimize the number of operations for each inference, what translates into a reduced number of *MACs* for each inference process. Possible solution is to use a technique called *pruning* of the network, it means that some weights are set to '0' (choosing a threshold based on sensitivity analysis [12]).

3.1.2 Energy efficiency

Energy efficiency, measured in $[GOP/J]$, is a way to quantify the energy needed for each inference [16]. It is a very important parameter for edge computing applications where the quantity of available energy is constrained.

$$efficiency = \frac{W}{E_{total}} \tag{3.2}$$

where E_{total} is defined in the following way:

$$E_{total} = E_{access-memory} \cdot N_{access-memory} + W \cdot E_{ops} + E_{static} \tag{3.3}$$

The energy efficiency is a parameter that takes into account the necessary operations for each inference divided by the total energy spent by the system in charge of computing that inference. The total energy is the sum of the energy spent in order to have data movement from memory and data processing.

In order to have good efficiency it is necessary to reduce the energy spent for each inference in this way:

- Low $E_{access-memory}$: the data movement is very expensive in terms of energy, it is especially important to note that data read from off-chip memory, like a *DRAM*, require three orders of magnitude more energy with respect to data read from on-chip memory, like *SRAM*, but the density of *DRAM* is bigger (1 transistor per bit vs 6 transistor per bit in SRAM), as shown in table 3.1.

Table 3.1: Comparison between *SRAM* and *DRAM* [17].

Parameters	*SRAM*	*DRAM*
Density F^2	140	6-12
Energy/bit (pJ)	0.0005	0.05
Read time (ns)	0.3-1	10
Write time (ns)	0.3-1	10

- Low $N_{access-memory}$: it is mainly related to architectural and operation scheduling problem. Possible solutions are strategies like *near data processing* that can minimize the distance between the *PE* and memory or maximize *data reusing*, in this way data from memory is read one time but it will be processed different times. For example in figure 3.3 there is an example of how different types of dataflows determine a different efficiency.

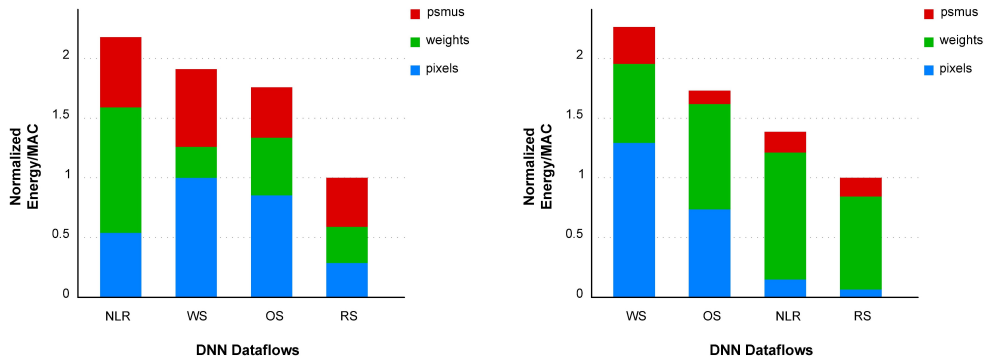

Figure 3.3: Comparison of energy efficiency between different dataflows in the *FC* (on the left) and *CONV* (on the right) layers of *AlexNet* [18].

3.1.3 Accuracy

Accuracy is a parameter that permits to evaluate how many predictions (output of the neural network during the evaluation phase) are equal to the correct data label.

The accuracy is linked with the number and the precision (# of bits) of parameters of the neural network, as shown in figure 3.4, and consequently it is strongly connected with the needed memory size. Besides, the number of weights is strongly connected with the number of operations done for each inference (especially for *FC* architectures, as was explained in subsection 2.3.2).

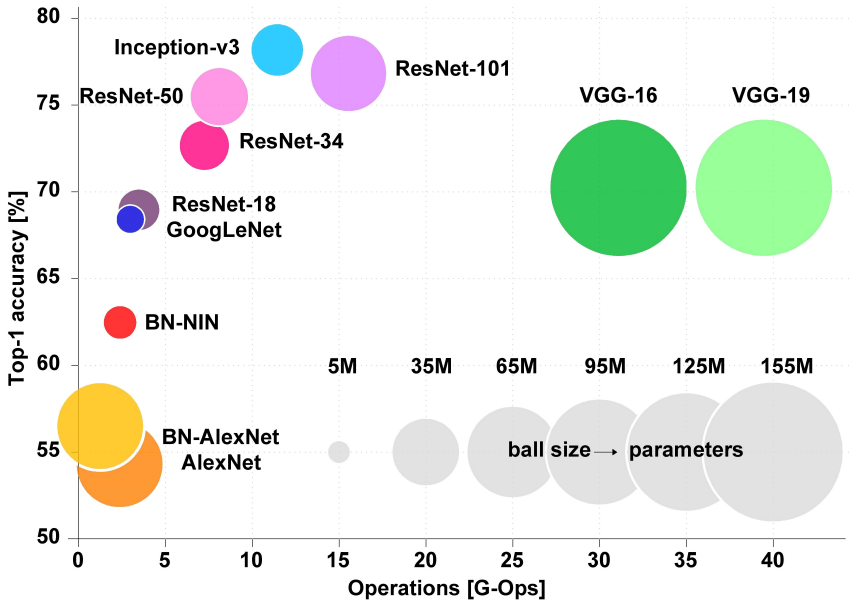

Figure 3.4: Accuracy vs Operations [19].

This is a very important aspect to take into account since when a neural network is designed on an *FPGA* there is a constraint on the memory on-chip in terms of area, and neural networks design needs a lot of data.

For example the largest memory available on an *FPGA* can be $<$ 50MB, but the number of parameters needed for some networks can be also from 100-1000MB. This gap can be covered by external memory, but this can limit the system performance [16].
For this reason is necessary that *FPGAs* read data from off-chip memory like *DRAM*, but it demands more power consumption and also requires more latency, so it can be a bottleneck of the design.

3.2 Comparison between *FPGA*, *GPU* and *CPU*

FPGA neural network design has to face different challenges in order to become the first choice regarding a neural networks accelerator [20]:

- Programmability: the most common frameworks are developed just for *CPU* and *GPU*. So the time to market is higher for *FPGAs* because their workflow is less optimized and requires highly skilled programmers.

- Area and performance Overhead: the working frequency of *FPGA* is lower with respect to *CPU* and *GPU*. Other two *FPGA* problems in terms of performance can be the flexibility, due to the fact that in some cases it is possible to have not used overhead area and also the connections that are not optimized and can be the first reason of power consumption.

In table 3.2 there is a comparison summary of the most important hardware parameters between these three different devices.

Table 3.2: Performance comparison among *CPU, GPU and FPGA*.

	CPU	*GPU*	*FPGA*
Market time	Low	Medium	High
Flexibility	Medium	Medium	High
Parallelism	Low	Medium	High
Power	High	High	Medium
Example	*Arm-cortex-a9*	*Nvidia-Titan-x*	*Virtex-ultra*
Efficiency[GOP/J]	1	10-100	>100
Speed[Oper./s]	0.01	10	0.02
Frequency[GHz]	1.9	1,5	0.3

3.2.1 Architectural comparison

In this part there is a comparison on what types of typical architectures are used in the different devices.

Figure 3.5 shows on the left the *temporal architecture* used by *CPU and GPU*, and on the right the *spatial architecture* used by *FPGAs*.

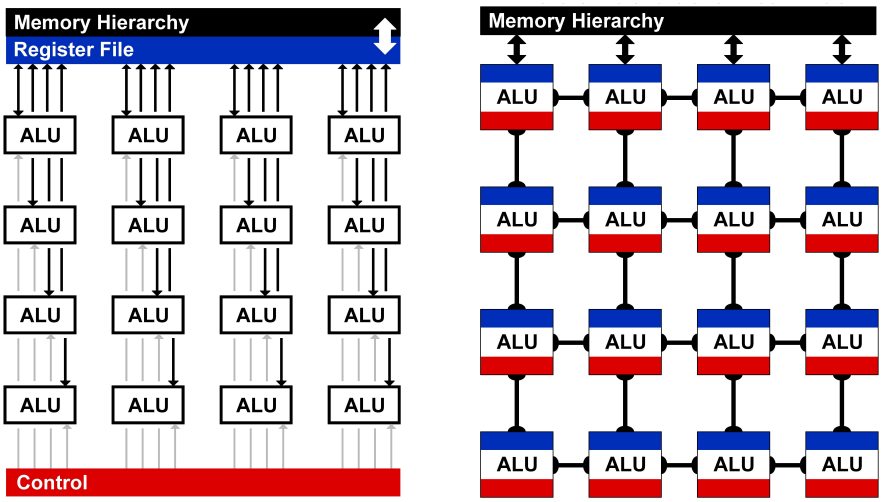

Figure 3.5: Temporal architecture vs. Spatial architecture [1].

1. Temporal architecture: temporal architecture uses a centralized control for a large number of *ALUs*. These *ALUs* can only fetch data from the memory hierarchy and cannot communicate directly with each other [1]. There are platform libraries that dynamically choose the appropriate algorithm to optimize the *MACs* operation for a given shape and size of neural networks.

2. Spatial architecture: spatial architecture consists of dataflow processing thanks to the fact that inside each *ALU* some logic and memory are integrated, thus in this way it is possible to avoid a lot of memory accesses from the main memory (avoiding power consumption) introducing some local memory like a register file *RF* or buffers. The block including *ALU*, memory and logic is called Processing Elemente, *PE*.

3.3 *FPGA* based accelerator

Nowadays neural network design on *FPGA* is an acive research area since *FPGA* design permits to obtain more flexibility and better performance compared to *CPU* and *GPU*.

For this reason the *FPGA* is the most considered when we speak about edge computing, where the energy budget is fixed and the energy efficiency becomes even more important than the speed.

One of the most important peculiarities of *FPGAs* is the possibility to explore the parallelism of the algorithm in a way that permits to increase the concurrency and as a consequence also the throughput.

It is important to consider that neural network structures can be implemented with specific architectures which could be tiled at different levels [21]:

- **Layer parallelism**: several layers can work in a parallel way.
- **Neurons parallelism**: a group of neurons can work in a parallel way.
- **Neuron parallelism**: a single neuron can process more than one input at each time.

Besides when the concurrency increases for a target throughput it is possible to decrease the frequency and as a consequence the power supply needed to bias the circuit. In this way *FPGA* based design can conquer high energy efficiency, maintaining the same throughput.

3.3.1 Hardware design bound

Computation and communication are two principal constraints in system throughput optimization. An implementation can be either computation-bounded or memory-bounded [22].

In order to check which is the limit of the architecture figure 3.6 can be useful in which the vertical axes labeled as hardware performance is the throughput of the system evaluated in $MAC/cycle$ and the horizontal axes represented by computation to communication ratio (CTC) is evaluated in $MAC/data$ and it represents how many *MACs* it is possible to do with a unit size memory access.

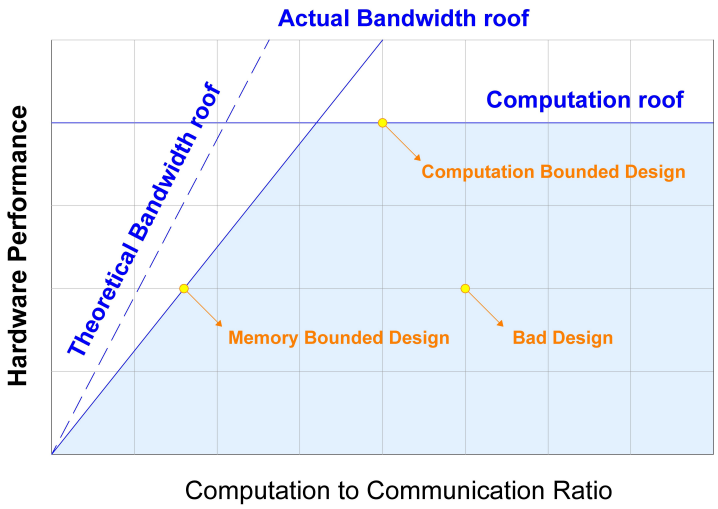

Figure 3.6: Roofline performance model [16].

In figure 3.6 every point of the graph is a possible hardware design of an architecture and it is possible to distinguish two different areas:

1. *Sloped area* means that design is memory bounded. It consists on the fact that the architecture is able to do more $MAC/cycle$ that it actually does since there are more processing elements that can be used, but memory constraints limit the possibility to use all available *PEs*. The slope of this line depends on which is memory management of the architecture.

2. *Flatten area* is bounded by the number of *PEs* that are available in the architecture. In the graph the intersection between the two lines means that according to memory bandwidth (BW) the architecture is able to process all available *PEs*. The constant line computation roof represents that $MAC/cycle$ can be also lower than the ideal one because in some cases not all *PEs* can be used concurrently (it depends on data dependencies).

The necessary BW for a fixed throughput can be defined with formula 3.4:

$$BW = \frac{throughput}{CTC} \tag{3.4}$$

where CTC is the computation to communication ratio.

The max performance in terms of throughput that it is possible to achieve is given with formula 3.5:

$$Max - performance = \min \begin{cases} Computation - Roof \\ CTC \cdot BW \end{cases} \tag{3.5}$$

3.4 Hardware design for efficient architectures

In figure 3.7 there is a summary of which are the possible techniques that can be used in order to optimize the hardware design.

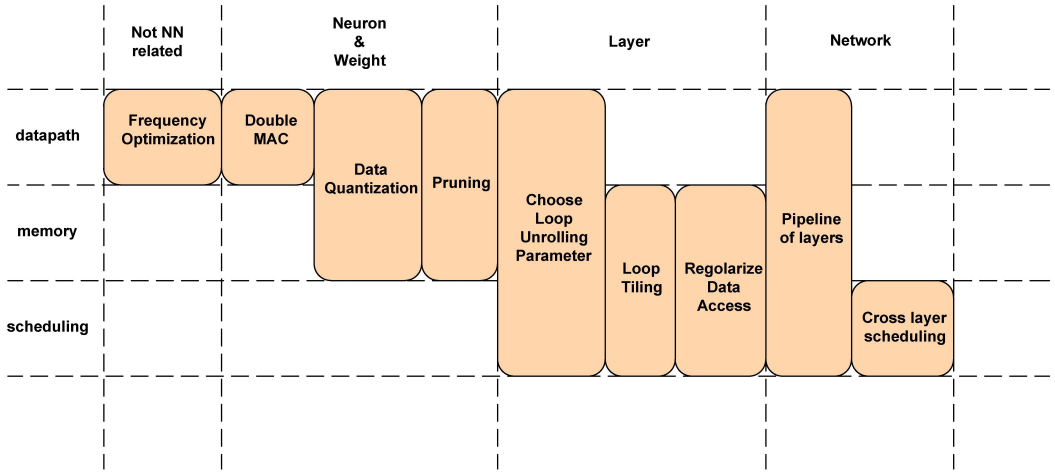

Figure 3.7: Overview of hardware optimization techniques [16].

The figure is divided into two views, the first part (horizontal view) identifies the different hierarchical levels of neural networks starting from neuron to a complete network, the second part (vertical view) identifies three different hardware characteristics (datapath, memory and scheduling of operations).

The various techniques are independent of each others, what means that it is possible to apply at the same neural network different optimization techniques.

It is important to note that different optimizations can have different types of impact on the hardware design. For example *data quantization* has impact on datapath and on memory, while *loop unrolling* has impact on all three characteristics.

In the next subsections there are some examples of system design and data compression optimizations for neural network. These are the two main aspects that will be taken into account in order to design an optimized neural network.

3.4.1 System level optimization

Memory hierarchy organization

The data movement from and to memory dominates the energy consumption so it is necessary to minimize it as much as possible.

The spatial architecture introduced in 3.2.1 allows to explore *near data processing* and maximize *data reusing*.

This aspect is possible since the *DNN* workflow is known so thanks to profiling algorithms there is the possibility to predict which data are more possible to reuse and in which way it is possible to minimize the access to main memory.

This optimization is very useful because access to main memory like *DRAM* is much expensive than *RF* access. The bets solution is to create a hierarchical organization of the memory and when a piece of data is read from main memory, it has to be used as many times as possible.

In figure 3.8 there is an example of this hierarchical organization of memory on *FPGAs* and also a histogram with normalized energy cost per access of different memories.

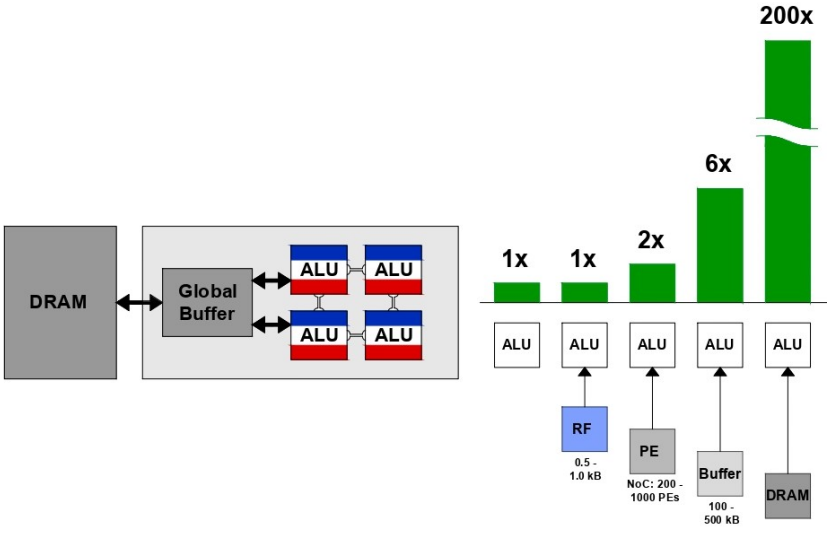

Figure 3.8: Example of hierarchical memory architecture on *FPGA* [18].

Regularize access memory and data reusing

Data reuse requires a small associated memory to the ALU unit. It is advantageous in terms of efficiency, pJ/b, but inefficient in terms of area, $\mu m^2/b$ [1].

In figure 3.9 there are two different examples where different workflows are presented in order to increase data reuse and minimize data movement.

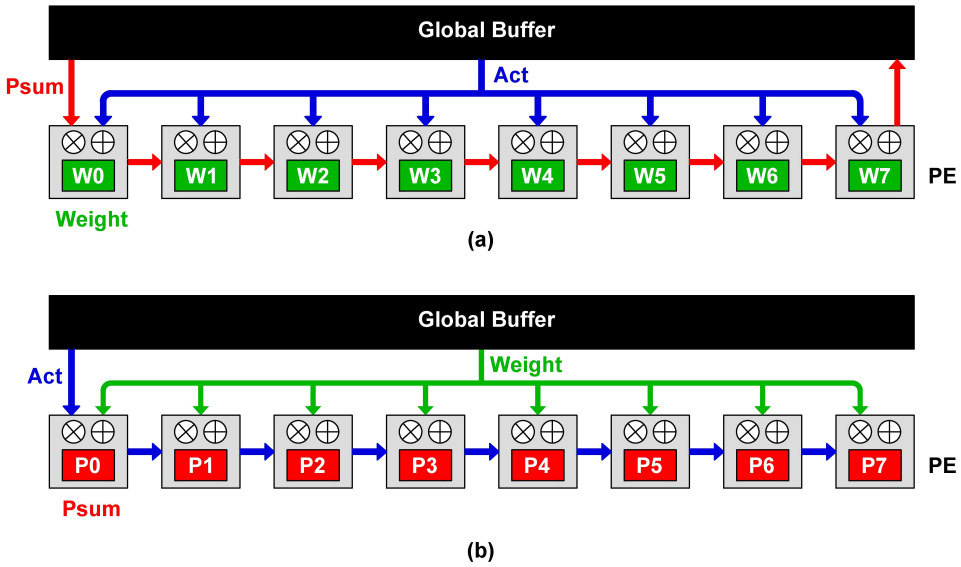

Figure 3.9: Data reuse exploration [18]. Weight stationary (a). Output stationary (b).

a) Weight Stationary: this approach stores in a RF near the $ALUs$ the weights after they are read from a global buffer. It permits to use more the weights that are in the RF and the partial sum is accumulated along the $ALUs$ chain and at the end the computation results are written in the global buffer.

b) Output Stationary: this approach stores the partial sums in an RF near the $ALUs$ after that they are read from the global buffer. This design allows, for example, to stream the activation along the $ALUs$ chain and broadcast the weights.

The best data reuse approach depends on the neural network design and on the different workflows. For example the number of FC layers, CNN layers, the number of weights for each layer, the number of overall layer and also other parameters.

3.4.2 Data compression

Data compression is a technique that impacts in the datapath in terms of area, speed and power consumption and also in memory size. One of the most important goals of data compression is to have the possibility to store all the weights into the *FPGA* and not to use external memory that for reasons previously discussed has a bad impact on the system performance. It is an open research field and object in different works [1], [12], [16], [23], [24] and [8].

It is important to evaluate how much accuracy will be lost when data compression is used. It is fundamental to find a trade-off between accuracy and resource reduction obtained by model compression.

It is useful to know in which part of the architecture it is possible to use data compression, for example on weights or on activations or both of them in a way that allows that the loss of accuracy matches our accuracy target.

There are different methods that can be exploited in order to make data compression [1]:

- *Uniform quantization*: [1], [8] and [17].
- *Not uniform quantization*: [12], [23] and [25].

Uniform quantization

Uniform quantization (uniform compression) permits to pass from floating-point operations to fixed point arithmetic that is more hardware friendly. Uniform quantization consists in choosing a number of allowed levels, N, that can be represented with $\log_2 N$ bits. In this case we talk about uniform quantization because the levels the we choose are equally spaced among them.

Not uniform quantization

Not uniform quantization (not uniform compression) provides quantization levels that are not uniformly spaced out because the distribution of weights and activations usually is not uniform, thus a possible approach is applying *weights sharing* in which a group of weights (or activations) can share a same unique value. The number of possible levels (unique values) N can be coded like before with $\log_2 N$ but in this case they do not represent the weights values but the index that point to LUT in order to read the real weights that must be used.

Chapter 4

Modeling of neural networks

The scope of this chapter is to develop a numerical model of *FC* neural network and characterize it in terms of accuracy loss when model compression is applied. Model compression has as a consequence a reduction of the model precision with the different techniques previously introduced: linear compression and not linear compression.
In the first part there is a description of how to model the training and inference phases in a neural network.
After that, there is an explanation about the model which is implemented on *MATLAB*. Finally the results coming from different reduced precision models are shown and are compared among each others. The found weights will be used to put them in the *FPGA* memory during the implementation part of the proposed inference hardware accelerator.

4.1 Neural network numerical model

Neural network functioning is divided into two main phases, the inference phase and the training phase.

The *training phase* is the time when neural networks has to search the neural networks weights that minimize the cost function.

The *inference phase* is when the weights have been trained and they are used to generate the output of the neural network. This phase is also called feed-forward phase because the propagation of computation comes from the first layer to a last layer where there is a predicted output.

The cost function (J) and regularization factor can be defined in the following way [2] for a neural network where the parameters and variables that appear are explained in table 4.1:

$$J(\theta) = \frac{1}{m} \cdot \sum_{i=1}^{m} \sum_{k=1}^{K} (y_k^{(i)} \cdot \log(h_\theta(x^i))_k + (1 - y_k^{(i)}) \cdot \log(1 - (h_\theta(x^i))_k) \tag{4.1}$$

$$Regularization - Factor = \frac{\lambda}{2 \cdot m} \cdot \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\theta_{ji}^{(l)})^2 \tag{4.2}$$

$$J(\theta)_{regularized} = J(\theta) + Regularization - Factor \tag{4.3}$$

It is important to note that the cost function is the sum of the error the neural network makes on its prediction and it depends just on θ value. If the cost function is high the accuracy will be low.

The regularization factor has to be added to the cost function in order to avoid overfitting of the training data during the training phase because it gives a little degree of freedom that permits to generalize the cost function for data different from the training data.

The cost function is a way to measure the quality of the network accuracy. The goal of training is to reduce it as much as possible changing the learnable parameters.

The problem of the neural network cost function is that it is not a convex function so there is the possibility that there are have several local minimum and it is more difficult to find the global minimum. There are different optimized functions that aim to find the minimum of cost function like *stochastic gradient descend* (*SGD*) or other optimized functions.

Table 4.1: Parameters explanation of the cost function and regularization factor.

Symbols	**Description**
K	Number of classes
m	Number of data sample
L	Number of total layers
s_l	Number of neurons in the layer l
$x^{(i)}$	Features-i_{th} of the data sample
$y_k^{(i)}$	Label-k_{th} of the sample i_{th}
$\theta_{ij}^{(l)}$	Learnable weights from i_{th} neuron in layer $l-1$ to j_{th} neurons in l layer
$h_\theta(x^{(i)})$	Activation function evaluated with θ weights and $x^{(i)}$ data sample
λ	Lambda factor

In figure 4.1 is possible to appreciate in a schematic way which are the different phases that are involved during neural network processing.

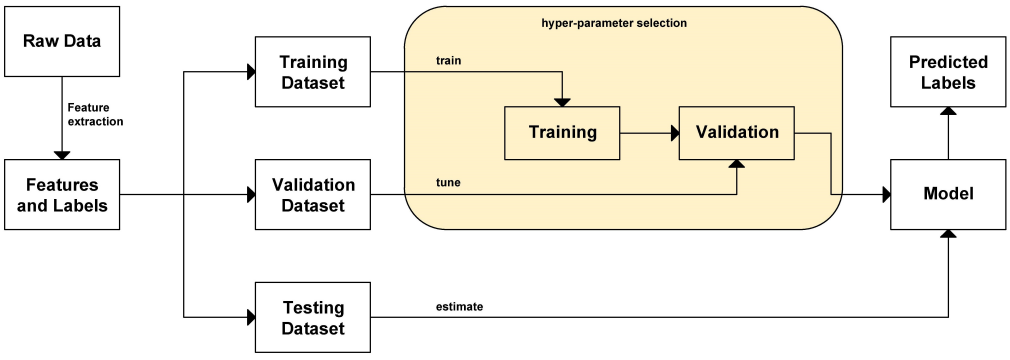

Figure 4.1: Schematic neural network working.

4.1.1 Training phase

The training phase has the objective of computing the parameters (weights) that minimize the cost function and so improve the accuracy of the system.

Its computational complexity is higher than the one required by the inference because it involves three different phases and one of them is the inference phase.

They are described in the following list and their formal steps are reported in algorithm 1 [23].

1. Feed forward: this phase consists in the propagation of the activation functions for each layer of each neuron according to equation 2.1 until the output layer.

2. Backward propagation: it consists in computing the gradient of each weight according to the derivative chain rule, starting from the output layer and going to the first layer.

3. Weights update: it consists in using the gradients previously computed for uploading the new weights along the negative direction of the cost function gradient with respect to the old weights.

This process is iterative and continues until the difference between the output activation and desired output meet the target.

The training process can be slow because it is an iterative process and so in a lot of cases the training is done off line because there are usually timing constraints on this phase that, for example for *FPGAs*, it is not possible to respect.

In some works the training is done on *FPGA* thanks to the application of the model compression also during training phase and not only on the inference. Some examples for these works are reported where partial binary or fully binary neural networks are applied [24], [23] and [26].

In algorithm 1 [23] all the steps to compute the new weights are reported.

In order to achieve the best possible accuracy these steps have to repeated several times dictated by the choice of number of iterations.

It is important to note that for each iteration the algorithm has to calculate all the activations for each neuron and all the gradients for each activation and for each weight.

Algorithm 1 *Training algorithm.* All operations are matrix-wise. L is the number of layers, σ is the activation function, LR is the learning rate and J is the cost function.

Require:
Initial random parameters W, input data features a_0 , the corresponding correct target y and LR.

procedure TRAINING(W, a_0, y, LR)
  ▷ 1.FEED FOWARD PROPAGATION
  for $k = 1$ `to` $L - 1$ **do**
    $\mathrm{a}_k = \sigma(a_{k-1} \cdot \mathrm{W}_k)$ ▷ Activation functions are computed
  end for
  ▷ 2.BACK FOWARD PROPAGATION
  $g_{aL} = \frac{\partial J}{\partial a_L}$ ▷ Initialize output layer's activations gradient
  for $k = L - 1$ `to` 2 **do**
    $\mathtt{g}_{a_{k-1}} = (\sigma'(a_k) \circ g_{a_k}) \cdot w_k$ ▷ Activation gradients are computed
    $\mathtt{g}_{W_k} = \mathtt{g}_{a_{k-1}} \cdot a_k$ ▷ Weights gradients are computed
  end for
  ▷ 3.WEIGHTS UPDATE
  for $k = 1$ `to` $L - 1$ **do**
    $\mathrm{W}_{k(updated)} = \mathrm{W}_k + LR \cdot g_{W_k}$ ▷ New weights are computed and upload
  end for
end procedure

4.1.2 Inference phase

The inference phase allows to compute the output of the neural network and after that it compares the output with the correct labels in order to characterize the accuracy of the system. The inference phase is less expensive with respect to training in terms of computational complexity because it requires less operations. Inference involves only in the feed forward phase.

During inference the pre-trained weights are used.

In the edge computing area it is very important that the inference phase is really quick in order to respond in a reasonable time. It is also desirable to be energy friendly and that the dedicated hardware occupies the smaller possible area.

In order to do it we trained the neural network on *MATLAB* and in the second phase we uploaded the weights on the *FPGA*. For optimizing the inference phase we adopted the strategies of data compression of the numerical model.

Data compression for the inference phase has an impact on *PEs* complexity and on weights memory constrain [1], [12] and [16].

In algorithm 2 [23] all the steps to compute all the operations involved during inference are reported.

Algorithm 2 *Inference algorithm.* All operations are matrix-wise. L is the number of layers. σ is the activation function.

Require:
A deep model with parameters W and $bias$, input data features a_0.

procedure INFERENCE(Model, a_0)
  ▷ 1.FEED FOWARD PROPAGATION
  for $k = 1$ `to` $L - 1$ **do**
    $a_k = \sigma(a_{k-1} \cdot W_k)$ ▷ Activation functions are computed
  end for
  return a_L ▷ Activation function in the final layer
end procedure

4.2 Neural networks modeling implementation

We choose to implement a *FC* neural network for hand digit recognition (*MNIST* data set [27]). Our network is composed by three different layers, the number of hidden units inside the hidden layer is a parameter that it is possible to choose.

Usually model compression is done with more than one hidden layer leaving the input layer and output layers with max precision. In this way the neural network is able to recover the loss of accuracy. In this case the worst case of data compression is considered, it means the obtained accuracy after model compression is the smallest possible, and adding other layers and re-training a possible accuracy improvement is possible to see.

The implemented model is composed of the following functions:

1. Load data set [28]: permits to upload the original data set and involves some pre-processing activities like choosing the number of features to extract from the original samples.
2. Rand weights: initializes the weights to random-values between [0,1] in order to break the symmetry and speed up the training. The initialized weights size depends on the input features, output possible labels and on the number of neurons for the hidden layer.
3. Data set splitter: the data set is split into three data subsets used respectively for training (60%), validation (20%) and testing (20%).
4. Trainer & validate: computation of the cost function of the training set with initialized weights and the weights' gradients according to the chain rule of back propagation. Thanks to the advanced optimizer using gradient weights, new weights that minimize the cost function are found and they are uploaded as new weights. This process is iterative and we can set the number of iterations until the cost function is as small as possible. The training process is repeated for different lambda values and the results are compared against the validation set in order to choose the best regularization factor λ. In this phase all the computation is done with max precision (*64-b floating point*) in order to improve the speed of convergence values.

5. TRANSLATE WEIGHTS: the full precision weights are split into different precision weights in order to analyze uniform quantization and not uniform quantization.

6. TESTING: a feed forward phase is done with the found weights and the resulting accuracy is computed.

7. FINE TUNING: the accuracy can be improved by re-training the neural network with quantized weights. In order to achieve improvements in accuracy it is important that during the feed forward and back ward propagation phases the weights are quantized but during the weights updating phase the weights should work with full precision, because the advanced function used in this case works well in this way.

Figure 4.2 summarizes which are the different function blocks and how they work in a schematic way.

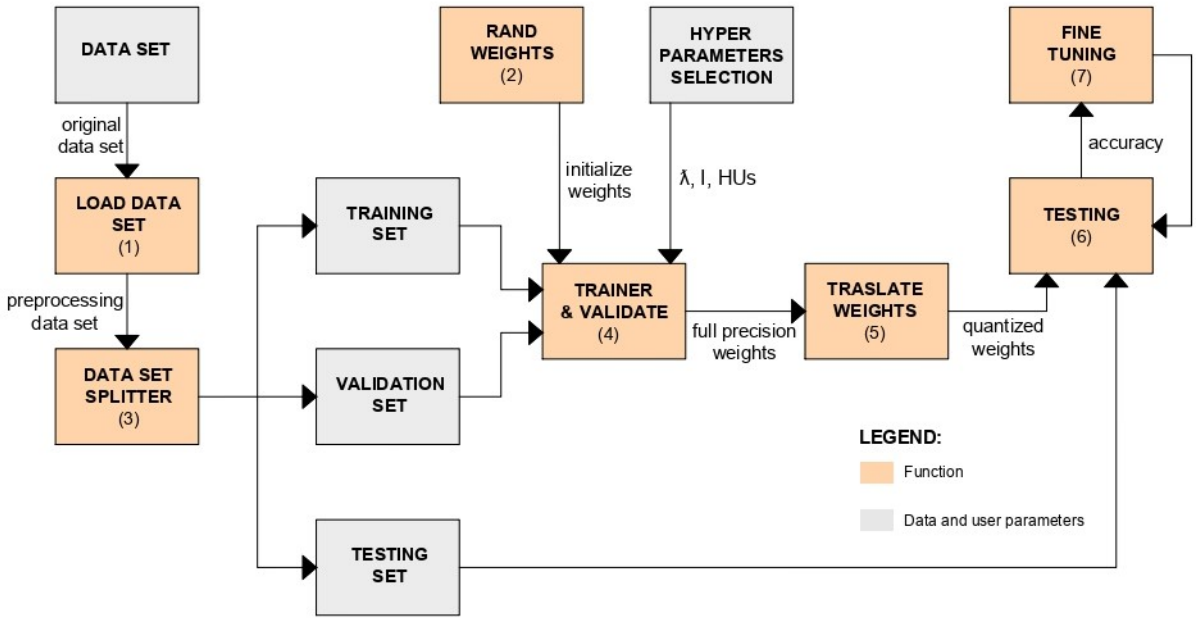

Figure 4.2: Schematic neural network modeling implementation.

Advanced optimizer: Sgdupdate vs Fmincg

The optimizer function during the *Weights update* phase computes which are the new weights that it is necessary to update in order to minimize the cost function.

There are various optimizer functions in order to work want different parameters, for example the number of iterations [N] (number of times it necessary to upload the new weights according to the computed weights' gradients) and the learning rate [α] that is a parameter that determines the speed of neural network learning.

It is important carefully set these parameters in order to avoid data overfitting/underfitting (it depends on the number of iterations), a too slow training or not training convergence (it depends on the value of the learning rate).

Two different types of optimizers will be used:

1. FMINCG: it is a built function provided by [2] and it will be used during normal training. It does not work well with quantized weights because during fine tuning of the cost function for quantized values it is not always decreasing, and this function when the value of current cost function is greater than the previous, stop the computation before to compute all scheduled iterations.
2. SGDUPDATE: it will be used for the fine tuning. It is based on stochastic gradient descent and it is provided by [28].

It is possible to note in table 4.2 that *Fmincg* optimizer is quicker than the other one because it needs less iterations to have the maximum of accuracy with training data set.

Table 4.2: Evaluation of accuracy with two different types of optimizer functions fixing the other parameters.

Parameters : m=5000, x=400, $\alpha = 0.1, HUs = 25$		
Iterations	**Accuracy**$_{Fmincg}$	**Accuracy**$_{Sgdupdate}$
N = 100	99.84	53.3
N = 1000	100	81
N = 1500	100	90

4.2.1 Data set

Data sets used for classification is *MNIST* [27]. This original data set consists of 28×28 grayscale pixels of each image of handwritten digits. There are ten classes (for ten digits) and 60k training images and 12k test images.

Pre-processing activity

It is possible reduce the sizes of the original data set decreasing the number of training images and the number of input features for each image thanks to previous introduced function called *load data set* (provided by [28]).

This because the number of used features (x) can determine under fitting or over fitting problems and also it impacts on the accuracy system.

But also the size of data set (m) used for training has relevant for the accuracy of the system.

This is not the only reason because a bigger number of features means computation more expensive and massive storage constrain.

Figure 4.3 shows the distribution of the input pixels in the two different cases with number of features = 784 and the number of features = 400 for each image.

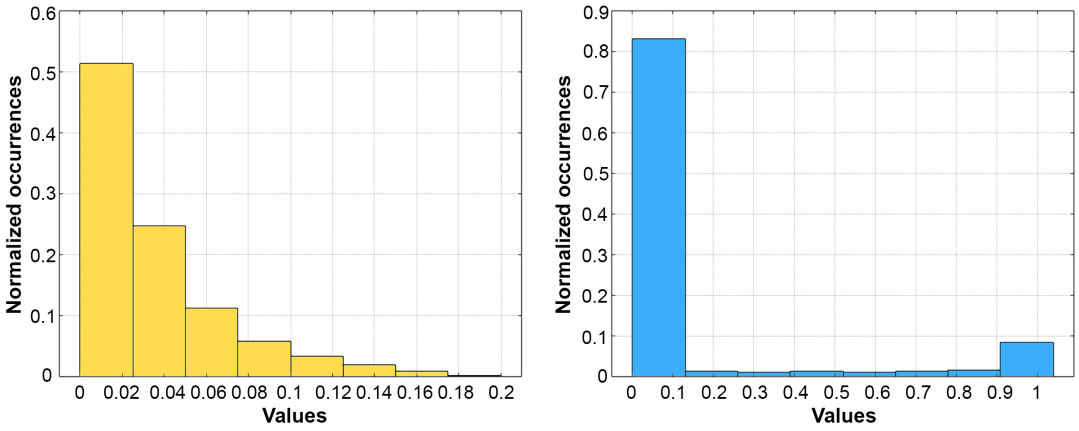

Figure 4.3: Normalized input features distribution. On the left: x = 784 & m = 60k. On the right: x = 400 & m = 5k.

The inputs are normalize in the range [0,1] in order to speed up the computation dividing each feature to the max possible range feature 255.

In table 4.3 it is possible to note that distribution in the two different cases study is very different.

Table 4.3: Properties of input features distributions.

Data set	60k x 784	5k x 400
Features	47040000	2000000
Zero-values(%)	80.8	0
Non-zero-values(%)	19.12	100
Bin-Width	0.13	0.025
Num-Bins	8	8
Ranges	**Occurences**(%)	**Occurences**(%)
1	83	51.2
2	1.1	24.7
3	1.2	11.2
4	1.3	0.05
5	1.2	0.03
6	1.2	0.01
7	1.1	0.0099
8	8.4	0.0034

In the case with x = 784 there are a lot of features that are zero. Contrary on the case x = 400 there aren't zero values.

The distribution in the first case is focus on the value [0,0.1] for the majority (80%) of them and in the value near to one (8%). In the second case the distribution is more uniform but with values that are smaller of one order of magnitude respect the first case.

The massive number of zero in the first case suggests that many information are redundant and can be reason of overfitting.

If we focus on non-zero-values they are more in the second case, so the actual needed memory storage can be higher. *MAC* operations with zero values could be skipped in the first case but in hardware implementation some hardware overhead has to be added in order to detect it so it is important evaluate trade off of this choice.

Thanks to *fixed point designer* instruments provided by *MATLAB* it is possible to understand which is the best way that represents the inputs data with a reduced number of bits in fixed point notation.

Table 4.4: Fixed point notation with x = 784 for each sample.

# of bits	*# of fract. bits*	*In range(%)*	*Below prec. (%)*	*Out range (%)*
16	15	100	0	0
8	7	99.95	0.048	0
4	3	97.96	2.04	0
2	1	94.11	5.89	0

Table 4.5: Fixed point notation with x = 400 for each sample.

# of bits	*# of fract. bits*	*In range(%)*	*Below prec. (%)*	*Out range (%)*
16	16	100	0.00037	0
8	8	90.3	9.7	0
4	4	17.3	82.7	0
2	2	0	100	0

In table 4.4 and 4.5 it is possible to see the more degradation in the second case where the total cover range is concentrated in a smaller values so in this case using fixed point approach doesn't help so much especially with low number of bits. In the first case the degradation is less (5.89% with 2 bits).

This impacts the storage requirement because higher number of features can be represented with a lower number of bits and cover range loss is lower. We study the results in terms of system accuracy using both the numbers of features.

To achieve the model compression we focus more on weights because they are more problematic regarding storage constrains and computation constrains. Besides when the training is done on the cloud the distribution of the weights can be studied and optimized instead the input features are usually random values.

4.2.2 Neural network max precision results

The two different data sets (the original one and a subset of this previous introduced) are training in different conditions for achieve the best in terms of accuracy for both configurations.

The parameters that it is important to take into account are the following ones:

- # OF ITERATIONS: if the number of iterations increases too much there is an improvement of the accuracy of the training set but there is a loss of accuracy in test set because there is over-fitting of the training set losing the general solution. During training the results are validated measuring also the cost function of validation set and when it begins to increase the training will stop at that number of iteration. This technique is called *early stop validation* [2].
- λ: different values of λ are evaluated and after that the value that report the best result on validation set is chosen.
- # OF HUS: they are the number of neurons in the hidden layer. An increasing of *HUs* determines an increasing of training time because the number of parameters is higher and also the number of iterations needed to achieve the best possible accuracy increases.

The best number of *HUs* depends on the data set size and features size like showed in table 4.6 where with few *HUs* neural network is able to have the possible max accuracy (95.7%) for the subset of dataset training with *HUs* = 25 and Iter. = 100. Contrary it is necessary more iterations and also more *HUs* for original bigger data set in charge of the best accuracy result (98.35%).

Table 4.6: Evaluation of accuracy with max precision operations with the original data set and a subset of this.

m	x	*Iter.*	*# of HUs*	λ	$Accuracy_{training}$	*# tests*	$Accuracy_{test}$
60k	784	1000	200	0.1	99.7	12k	98.35
5k	400	1000	200	0.1	99.7	1k	97.7
60k	784	100	25	0.01	94.4	12k	91.35
5k	400	100	25	0.01	99.4	1k	95.7

The results in the first line that reports the bigger network and the greater accuracy will be the reference for the model compression in terms of loss accuracy and model size reduction evaluation.

Training optimization: batch size approach

Possible approach to speed up the training (less iter.) is divided the training set in batches and evaluate the back propagation just for some different values at each iteration. The number of values is defined by batch size. With this approach it is possible to lose in term of accuracy because not the overall training set is used for upload the new weights but just small part each time.
In table 4.7 it is possible to see that increasing the *HUs* the accuracy loss is less with other fixed parameters reported in the table head.

Table 4.7: Accuracy evaluation with batch approach with different number of *HUs* of the original *MNIST* data set.

Batch-Size = 200, Epochs = 500			
Accuracy-(HU=50)	*Accuracy-(HU=200)*	*Accuracy-(HU=400)*	*Accuracy-(HU=600)*
93.1 %	94 %	94.6 %	95.2 %

4.3 Data compression

To perform model compression the training of the neural network is done with max precision, after that there is a phase of post-training model compression in which the weights are replaced by quantized weights analyzing the weights distribution. As possible to see in figure 4.4 the cost function descends with different speed based on which type of quantization is used.

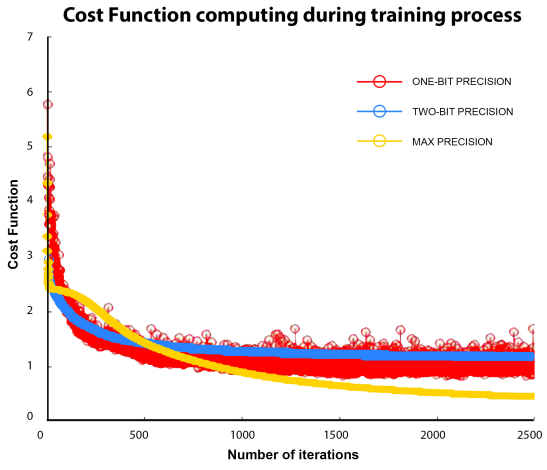

Figure 4.4: Example of cost function descending during fine tuning training with different applied precision on the weights.

4.3.1 Weights not uniform quantization analysis

In this section we want analyze and reduce the possible values of weights as low as possible for this reason we perform a not uniform quantization with these different cases:

- Two bits quantization
- Ternary bits quantization
- One bit quantization

In figure 4.5 there are the weights distributions for the neural network with topology 784 x 200 x 10 (the which one that reports the max achieved accuracy).

In figure it is possible discriminate the different distribution for the weights in the first layer (*Theta1*, blue one) and in the second layer (*Theta2*, red one) and also discriminate the normalized distribution with the original distribution.

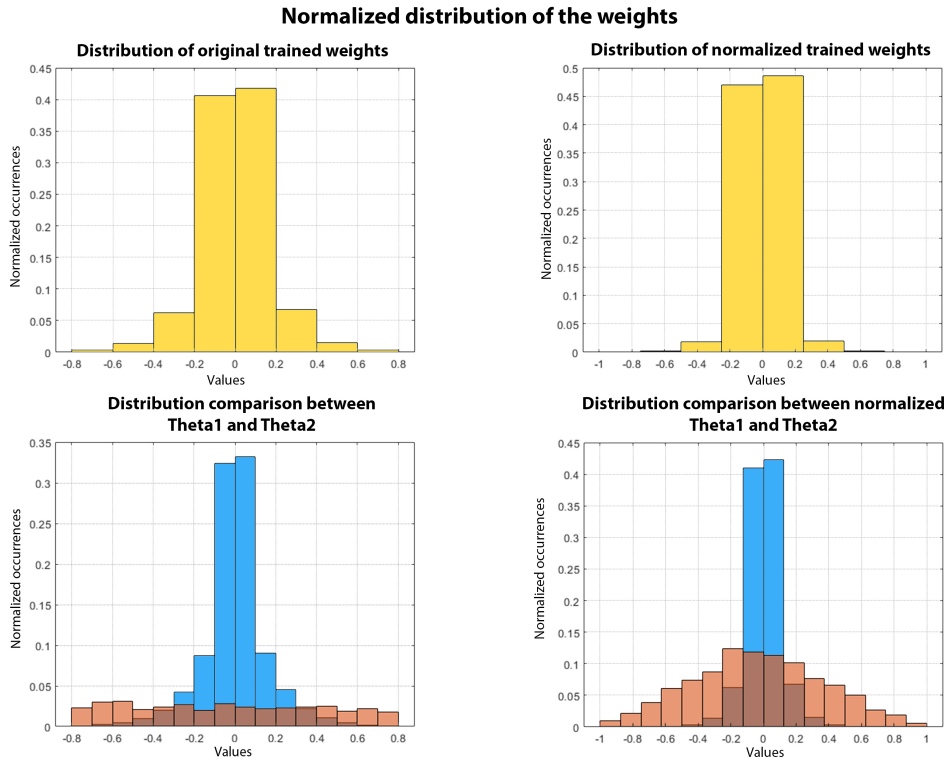

Figure 4.5: Trained weights distribution.

Two bits precision model

In this case it is necessary to have just four unique weights chosen by analysis of weights neural network distribution computed with max precision.

This technique is also called *weights sharing* [12].

In figure 4.5 it is possible to see the distribution of the trained weights (normalized or not) obtained of our neural network training.

It is possible to observe in the figure that the distribution of *Theta1* and *Theta2* is symmetrical respect to zero value and between them they are very different, the first is located near to zero value and it is shape is like a Gaussian, the second in more uniformly distributed in the same interval of the first.
It is possible to see that the values greater than 0.8 are not detected by histogram because they present a normalized distribution small (<0.025).

In order to simplify the model, the same not uniform quantization is applied for both the weights in the first and second layer.

For adopt the not uniform quantization with two bits it is necessary to choose four values for the weights that minimize the loss accuracy respect to the case with max-precision.

It is important to know that larger weights (in absolute value) play a more important role than smaller weights, but the density of them is inversely proportional for the weights that are more relevant respect to the weights less relevant [12] (because they permit to turn off or turn on the activation function).

The choice of the usable values with two bits is constrained by the following considerations:

- Two bits permit to discriminate four possible value for weights. So the constrain is to choose a threshold and in this way it will be possible to divide the weights between big value or small value. This division is done for negative values and positive values in the same way due to the symmetrical weights distribution respect to zero value.
- The parameters that are important to take into account are the normalized distribution of the weights (low density => high importance), the choice of threshold between the big values (more relevant) and small values (less relevant) and the choice of values inside each chosen range.

In table 4.8 there is a summary of the accuracy for different choices of not uniform two bits quantization applied to the weights.

The column *Threshold* is in absolute value and divides the values into two different ranges (four considering also the negative ones) discriminating the big value form the small value. The column *Values* is in absolute value and it identifies the numeric values that are associated to each range given by the *Threshold*. The *Values* inside each range is the average value of the range in order to minimize the introduced error. The value represented in the table are the same in absolute value for positive and negative range.

Table 4.8: Accuracy evaluation of two bits precision model.

Cases	**Threshold**	**Values**	**Accuracy(%)**
1	0.2	0.1 ; 0.3	54.62
2	0.6	0.3 ; 0.7	78.18
3	0.4	0.2 ; 2	82.34
4	0.5	0.25 ; 2	92.23
5	0.6	0.3 ; 2	92.16
Normalized weights			
6	0.3	0.15 ; 1	94.75

Fine tuning of case 6

Cases	**Iterations**	**Learning rate**	**Accuracy(%)**
7	1500	0.1	94.81

In *cases 1, 2* different thresholds are chosen in order to discriminate the big values from small values but in these first two cases the chosen values doesn't take into account the consideration done before about the importance of the big value and the inverse relationship between the density value and importance, in fact the values with a normalized occurrence less than 0.025 are not considered.

The result confirms the consideration so choosing values more likely (with normalized occurrences greater) inside the range dividing by four the total range in a not uniform way determines less accuracy (78.18%).

The *cases 3, 4, 5* show different thresholds with different values but in these cases the values that are chosen, they take into account all complete range of the weights also the values with very low normalized occurrence.

It is observable that the best value in term of accuracy is *case 4* in which there are unique values that are powers of two, it is also hardware friendly approach because it is able to replace multiply operations with just arithmetic shifts.

The *case 6* provides the normalization of weights in fact these ones are divided by the max possible value present in the range and so the weights are compressed in the range [-1;1]. The *case 6* registers an improvement of the accuracy respect to the *case 5*, it is clear that also the threshold and values used in this case are normalized.

Normalization increases the numbers of the values near to zero. This aspect permits also to have less dispersion of high value because now they are bounded and it provides the choice of unique big value easier.

Figure 4.6 shows how mapping of the weights sharing is performed in the *case 6*.

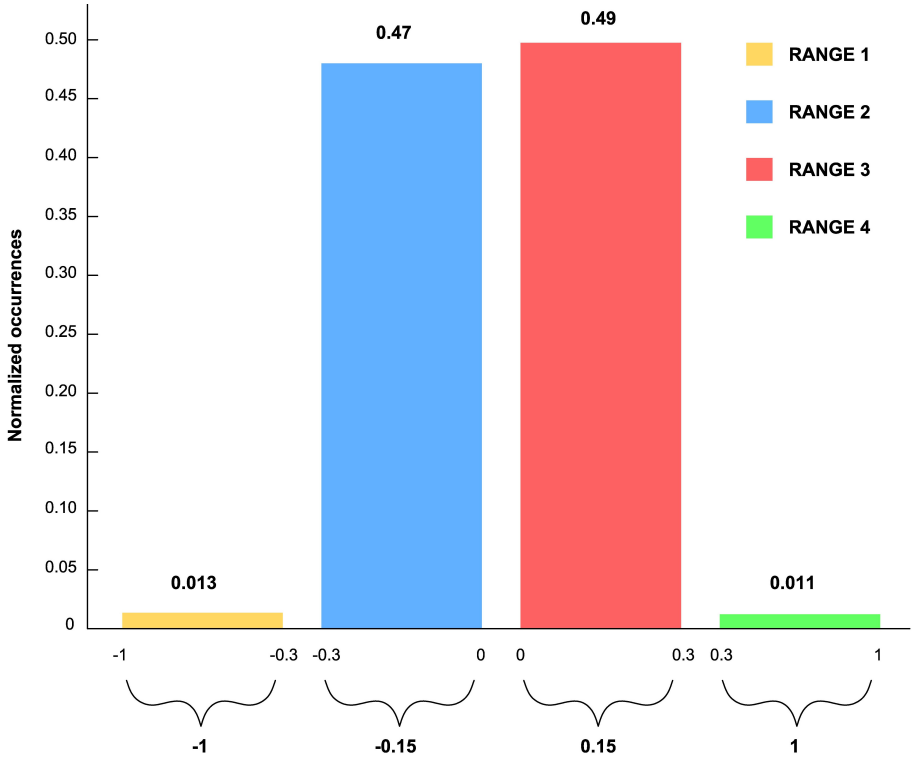

Figure 4.6: Weights sharing schematic view.

The *case 7* shows a small improvement of accuracy ($\simeq 0.06\%$) after a fine tuning phase with hyper-parameters showed in table 4.8.

Ternary precision

The scope of this not uniform quantization is split the original distribution of the weights into three values in which the central value is zero (-Value, 0, Value).

One consideration that it is possible to do is that the distribution of the weights is centered near to zero value where there are a lot of occurrences but some of them are redundant.

The idea of ternary precision is to use the *pruning* approach. It consists in choosing a threshold that permits to discriminate due to the magnitude the value that are relevant from the value that are not relevant. The values that are smaller than threshold are set to zero value and they can be cut in neural network connections [12].

The advantage of this techniques is that the number of *MAC* operations is reduced but it is necessary to introduce some overhead in order to detect zero weights and some *MACs* can be skipped.

This type of technique has also advantage in memory constrains because some strategies can be adopted in order to store the zero values in a smart way [1] [12].

Table 4.9 shows the results of ternary precision in terms of accuracy with different thresholds.

Table 4.9: Accuracy evaluation of ternary precision with different pruning level.

Cases	**Threshold**	**Value**	**Zero set values(%)**	**Accuracy(%)**
1	0.001	2	0.88	62.70
2	0.01	2	8.58	59.90
3	0.02	2	16.69	56.28
4	0.03	2	20.77	50.42
		Normalized weights		
5	0.001	1	1.40	61.56
6	0.01	1	13.56	57.21
7	0.02	1	21.0	49.57
8	0.03	1	24.40	37.53

One bit precision

In this case just one bit for the weights it will be used. The weights are well trained with max precision, after that they are quantized with two different values.

The distribution of the weights is symmetrical respect to zero value so that best approach is divided the overall distribution into two ranges respect to zero value. For each positive or negative range a value (positive or negative) is associated.

In table 4.10 various symmetric values couples are used for evaluate the accuracy.

Table 4.10: Accuracy evaluation with one bit precision weights with different values.

Cases	**Value**	**Accuracy**(%)	$Accuracy_{norm.}$(%)
1	0.2	53.53	52.65
2	0.4	61.37	59.81
3	0.6	62.61	61.08
4	0.8	62.83	61.40
5	1	62.89	62.39
6	2	62.86	62.34

Fine tuning of case 5

Cases	**Iterations**	**Learning rate**	**Accuracy**(%)
6	2500	0.01	67.84
7	2500	0.1	78.22

In this case the choice of small values is worse than bigger values (*case 1, 2*). The choice of extreme values (-1,1) of the distribution gives the better performance in terms of accuracy, but very small change respect *cases 3, 4, 5, 6*.

One of the most common approach with one bit precision regarding the usage of the values [-1,1] as unique values [23] [25] [24]. This approach is very hardware friendly because multiplier operations during inference in this case are just a possible CA2 of the number when weight equal to -1 or not operation when equal weights to 1.

The *cases 6, 7* show the results with the best accuracy after a fine tuning phase with the hyper-parameters showed in table.

In order to improve the accuracy a possible adoptable solution can be add extra *HUs* with random values to the original *HUs* and retrain the neural network, in this way we recover some accuracy loss like showed in figure 4.7.

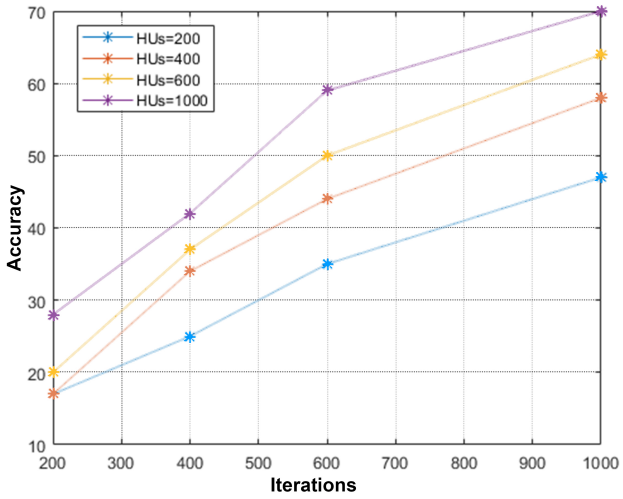

Figure 4.7: Evaluation of accuracy with one bit precision weight with different number of *HUs*.

We try also to train the network with quantized weights [-1,1] during the feed foward phase and during back ward phase reducing the precision but maintain full precision during update phase because *SGD* has to work with max precision. This type of approach can be useful when the training has to be in the *FPGA* because also the back ward propagation multiplier operations become because less hardware expensive [23] [25].

In table 4.11 the results of this type of tests suggest how the accuracy is linked with the number of *HUs* and also on type of training. This because the same information that it is possible to encode in a less number of *HUs* with max precision needes more *HUs* when the encoding contains just binary information. The results shows that it is better that training is done with max precision and after that a post training compression will be performed.

Table 4.11: Normal training vs binary precision training.

Training-(I=1000)	**Accuracy-(U=200)**	**Accuracy-(U=400)**	**Accuracy-(U=600)**	**Accuracy-(U=1000)**
Full precision	62.8	66.3	77.4	82.1
Binary precision	47.3	58.1	64.5	70.1

Weight memory compression

Not uniform quantization permits to store in the memory not the actual value of the weights but his codification and so it is possible to have memory constrain less restricted.

The number of bits for each weight for an efficient compression has less bits than actual weight value. Contrary there isn't any advantages in memory compression. We want evaluate the compression that we have for one bit and two bits weight sharing.

In order to evaluate it, formula 4.4 will be used [12]:

$$Compression - rate = \frac{nb}{n\log_2(k) + kb} \tag{4.4}$$

where n is the number of connection between the layers, b is the number of bits for the original weights and k is the number of bits chosen for the weight encoding.

In order to evaluate the *Compression rate* in table 4.12 we uses two different topology (the same used in table 4.6) in which the number of *HUs* is different.

Table 4.12: Compression rate for neural network topology of $784 \times 200 \times 10$ with $b = 64$ bits and for neural network topology of $784 \times 25 \times 10$ with $b = 64$.

Topology	*Cases*	*Compr. layer 1*	*Compr.layer 2*	*Compr. network*
$784 \times 200 \times 10$	Two bits	32x	30x	31x
$784 \times 200 \times 10$	One bit	64x	60x	63x
$784 \times 25 \times 10$	Two bits	31x	21x	31x
$784 \times 25 \times 10$	One bit	63x	42x	63x

Table 4.12 shows how the overall compression rate is the same for the two different topology. The compression rate of the layer 2 is different due to the fact that the amount of weights is less and in the computed overall compression rate doesn't have big influence.

Two topology are take into account due to the fact that not only the single weight precision impacts on the overall accuracy but also the number of *HUs*. This consideration has an impact on required memory constrains.

4.3.2 Uniform quantization analysis

Using floating point was the safest way to guarantee high accuracy and performance on the results. Floating point solutions in hardware are expensive computing and consume a lot of resources. They also have a big memory footprint.

Hence an alternative solution that comes is the use of fixed point, where they are more *FPGA* friendly computing. In *FPGA* designs, fixed-point formats are very efficient if we know beforehand the resolution and range of our data so that we can select the appropriate format.

Compare to floating point, fixed point requires less resources in *FPGAs* (*DSPs*) and they are less computational complex. In addition, we can reduce the memory footprint to a percentage that the network allows.

In table 4.13 it is possible appreciate which is the impact in terms of energy and area passing from floating point to fixed point and the impact on scaling the fixed point notation.

Table 4.13: Comparison between different types of values representation in terms of area and energy [29].

Parameters	*32-b floating point*	*32-b fixed point*	*8-b fixed point*
		Adder	
Area	116x	1x	0.26x
Energy	30x	1x	0.30x
		Multiplier	
Area	27.5x	1x	0.080x
Energy	18.5x	1x	0.064x

Another solution to be considered is the *dynamic fixed point*, the difference is that instead of using a global scaling factor, more can be used depending on the application's needs. This is useful for *DNNs*, since the dynamic range of the weights and activations can be quite different [1].

This idea was applied to the network so that each group of different layers can have different fixed point format using scaling factors.
We have noticed that the dense layer (*FC*) has the largest memory footprint compared to the others. So we tried to find the minimum number of representation bits having as a limit to the correct classification of the top classes.

Weights uniform quantization analysis

In this section the accuracy loss when the weights are uniform quantized is evaluated. In this type of tests it is used fixed point notation and the number of bits varies for the different tests.

In figure 4.8 there is a representation in fixed point notations of the weights (*Theta1, Theta2*) reducing the number of bits progressively (so also the required storage) but the other quantities (like the activations) remain unchanged. The different colors detect for the different number of bits how many values are *in range, below precision, out range).*

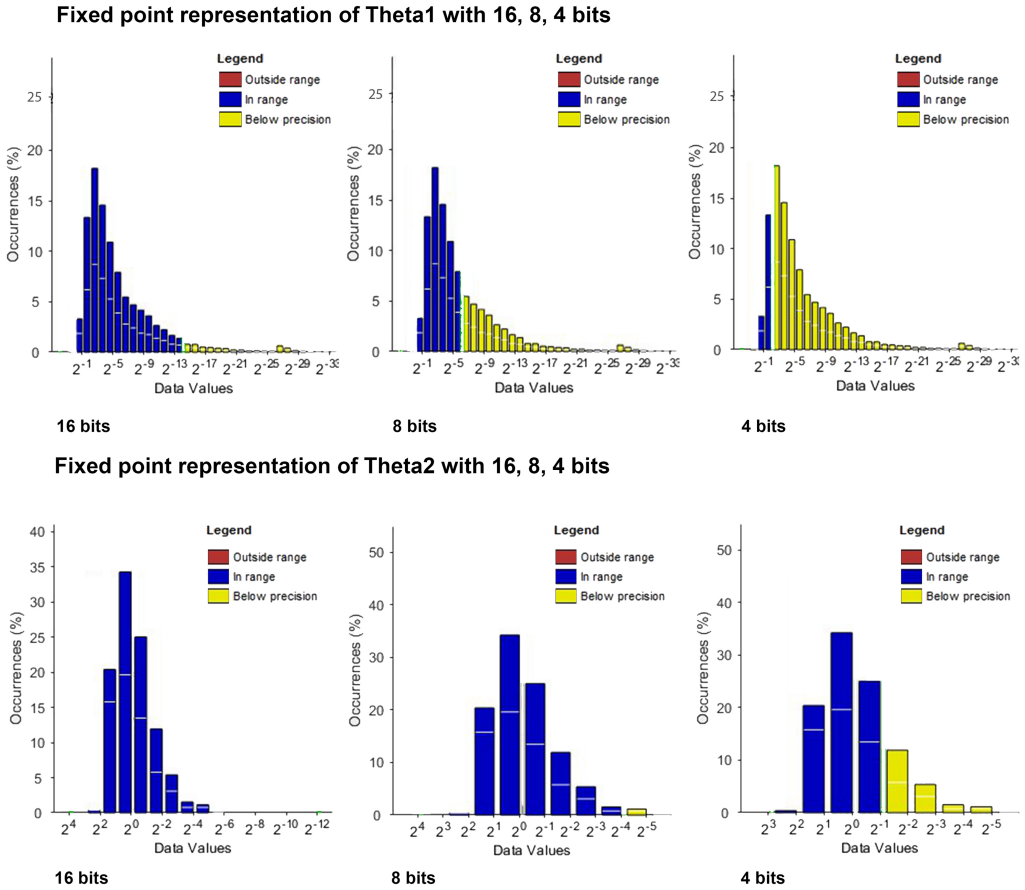

Figure 4.8: *Theta1* and *Theta2* fixed point notation representation.

Figure 4.8 is obtained by *fixed point designer* instruments provided by *MATLAB*. Thanks to this instrument it is possible to find the best dynamic fixed point representation when a data distribution is given.

In table 4.14 and 4.15 are showed the best fixed point representation obtained analyzing the weights distribution for *Theta1* and *Theta2* respectively (in this case they are not normalized).

It is important to see that the number of fractional bits for the two different layer weights is different due to the fact that also their distribution is different.

Table 4.14: Fixed point notation for *Theta1*.

# of bits	*# of fract. bits*	*In range(%)*	*Below prec. (%)*	*Out range (%)*
16	14	94.2	5.8	0
8	6	68.4	31.6	0
4	2	16.5	83.5	0

Table 4.15: Fixed point notation for *Theta2*.

# of bits	*# of fract. bits*	*In range(%)*	*Below prec. (%)*	*Out range (%)*
16	12	100	0	0
8	4	98.8	1.2	0
4	1	48	52	0

The table 4.16 shows the accuracy loss due to fixed point representation for a different number of weights bits. In this case a dynamic fixed point is used in order to improve the accuracy, so the number of fractional bits for the two different layer weights is different according the table 4.14 and 4.15.

Table 4.16: Evaluation of accuracy degradation with uniform weights quantization.

# of weights bits	**Memory compression**	**Accuracy**(%)
16	4x	97.52
8	8x	97.56
4	16x	95.88
2	32x	30.50

The accuracy results show a small degradation of the accuracy until 4 bits for the weights ($\simeq$ 2,47%). After that there is a bigger degradation ($\simeq$ 67.8%), contrary respect to the case with not uniform quantization in which with two bit and the right parameters it is possible to obtain an acceptable loss accuracy ($\simeq$ 3.6%).

Activation and weights uniform quantization analysis

In table 4.17 there is the accuracy evaluation when both the weights and activations are uniform quantized.

Table 4.17: Evaluation of accuracy degradation with uniform weights and activation quantization.

# weights bits	# Act. bits	# Act. fract. bits	Accuracy(%)
16	16	10	97.60
16	8	2	97.52
16	4	1	97.50
16	2	0	96.02
8	8	2	97.50
8	4	1	97.48
8	2	0	95.88
4	8	2	95.80
4	4	0	95.70
4	2	0	94.30

It is important to note that the quantization of activation is done before to perform activation function, it means that it is applied to the yellow graph in figure 4.9.

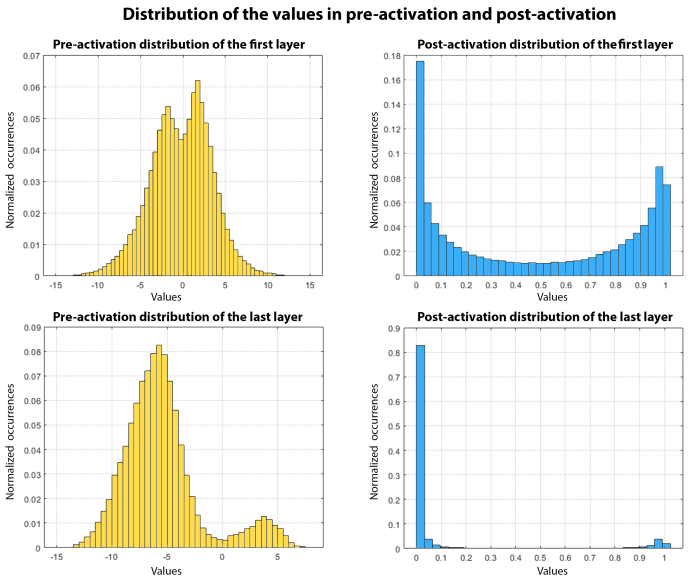

Figure 4.9: Example of the distribution of activation function.

4.3.3 Model compression analysis

In previous subsections 4.3.2 and 4.3.1 there are the complete explanation of the two different techniques that are used in order to perform model compression: *uniform compression* and *not uniform compression*.

At this point we want compare the two different used techniques in terms of accuracy and obtained model compression.

Figure 4.10 summaries the accuracy of some of the different types of used precision for the neural network. The different colors identify different type of model compression like described in figure.

In the horizontal axes the first number represents the number of the weights bits and the second number represents the number of activation bits. The vertical axes represents the accuracy.

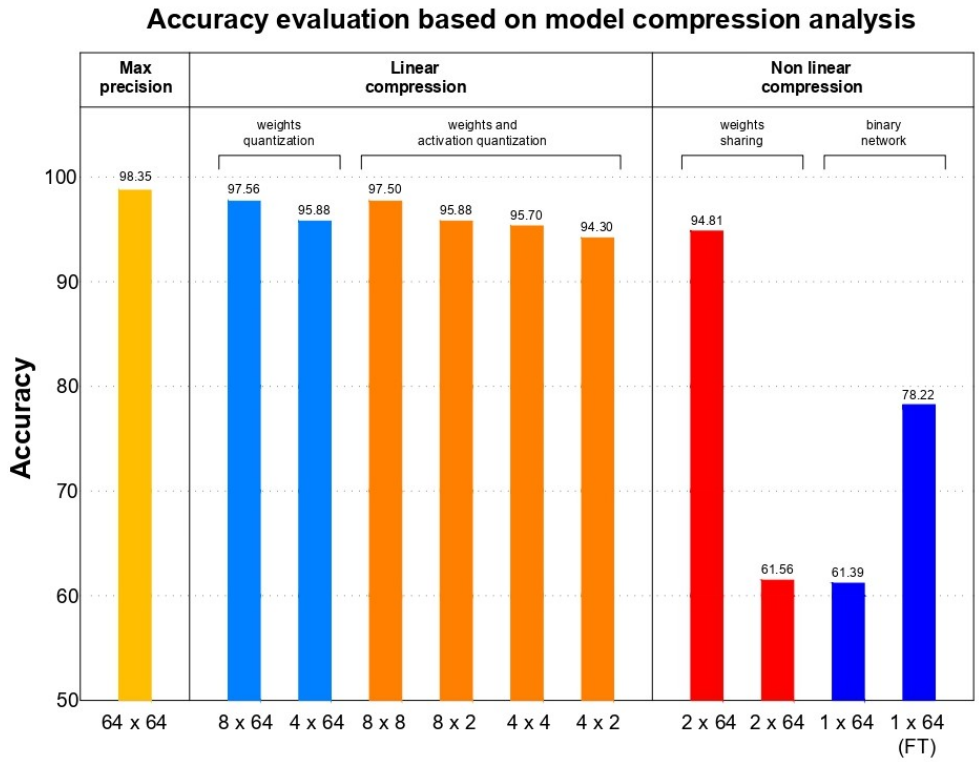

Figure 4.10: Accuracy evaluation of different types of model compression on neural network topology $784 \times 200 \times 10$.

The histogram shows how the limit case of uniform compression with 4 bits for the activation and 2 bits for activation has a small accuracy degradation ($\simeq 4.55\%$.) respect to the max precision case discussed in 4.2.2.

The not uniform compression with 2 bits shows a similar degradation of the previous one but with the possibility to use just an encoding of two bits for the weights ($\simeq 4.20\%$).

The partial binary neural network shows a bigger degradation with the same parameters of the others (the same number of *HUs*), an improvement is obtained after fine tuning like showed (FT). Another possible improvement can be increase the number of *HUs* and re-train the neural network like we described.

The advantage of the not uniform compression (weights sharing) is that it is possible not only improve the memory constrain but also using hardware that require less resources. For instance it is possible using *constant multiplier* (in a generic case), *shifter multiplier* (if the weights are powers of two) and a *binary multiplier* if the possible values of weights are -1 or 1. These possible choices are more hardware friendly respect to the case with normal multiplier with reduced number of bits given by linear compression. So not linear compression offers both memory constrain and hardware complexity bigger improvement respect linear compression but also a bigger accuracy degradation in a general case. It is normal because the neural network accuracy is strongly connected (more than precision activation) with the weight precision like discussed in 3.1.3.

A possible optimization of model compression can be mix the two different techniques, using for example weights sharing (not uniform quantization), with actual weights values with a reduced precision and quantized activation (uniform quantization).

Chapter 5

Design exploration of neural network

The purpose of this chapter is to design a neural network architecture paying attention to achieve high level of flexibility of the structure. The design analysis done in chapter 3 gives the idea of the hardware parameters for the structure for the different configurations. The idea is that a different neural network architecture can be generated depending on the constrained hardware resources and target throughput. The design takes into account the optimization techniques that are analyzed in the previous chapters in terms of *model compression* (results obtained by *MATLAB* analysis) and hardware techniques like *loop unrolling* and *pipelining*.

5.1 Design for tiling architecture

In order to achieve high hardware flexibility in processing the neural network is important to use tiling factors that permits to loop unroll the algorithm and partition the memory.

Loop unrolling is able to increase the concurrency of the architecture and as a consequence throughput of the neural network. Fixing the throughput, it is possible to increases the concurrency and so it is possible decrease working frequency and this choice can be impact on power consumption ($P_{din} \simeq f^2$). But increasing the concurrency reports more hardware resources and so more area ($P_{stat} \simeq A$), for this reason it's important to find a trade-off.

In listing 5.1 is possible to see that in order to process a single layer in a neural network it needs two nested cycle.

Listing 5.1: C code single layer neural network processing.

```
double input_vector [INPUT_DIMENSION];
double weights [INPUT_DIMENSION][OUTPUT_DIMENSION];
double output_vector [OUTPUT_DIMENSION];

for (int k = 0 ; k < OUTPUT_DIMENSION; k++)
{
   output_vector = bias;   //load the bias value;
for (int i = 0 ; k < INPUT_DIMENSION; i++)
{
   output_vector [k] += input_vector[i] * weights[i][k];
}
}
```

So the total number of iterations of a neural network with N_l neurons in the layer l and N_{l-1} neurons in the layer *l-1* are:

$$loop - iterations = N_{l-1} \times N_l \tag{5.1}$$

It's possible to apply loop unrolling of the two nested loops:

- *Outer loop*: in this way different neurons can be processed in parallel way this is possible because inside a layer there isn't problem of data dependency and the value of neurons weights are independent.
- *Inner loop*: in this way the single neurons can be able to process more inputs than one at each iteration.

The tiling factors that is possible to choose for the two different loops can be different. They are called respectively T_{in} for inner loop and T_{out} for outer loop.

It's possible to compute how many loop iterations are necessary after loop unrolling:

$$loop - iterations = \frac{N_{l-1}}{T_{in}} \times \frac{N_l}{T_{out}} \tag{5.2}$$

From equation 5.2 is possible to note that the number of loop iterations decreases like the product of the two tiling factors. It's fundamental to say that the tiling factors are directly linked to hardware resources of *FPGA*.

The two tiling factors increase the performance of architecture in terms of throughput because they improve the two different main parameters presented in 3.3.1 about possible causes of bound for hardware design:
MAC/data and *MAC/cycle*.

Listing 5.1 shows how to work a *FC* layer with $T_{in} = T_{out} = T = 2$.

Listing 5.2: C code single layer neural network processing after loop unrolling T=2.

```
#DEFINE T_IN 2
#DEFINE T_OUT 2

double input_vector [INPUT_DIMENSION/T_IN][T_IN];
double weights [INPUT_DIMENSION][OUTPUT_DIMENSION];
double output_vector [OUTPUT_DIMENSION/T_OUT][T_OUT];

// this loop is done in parallel with a tiling factor of T_OUT
for (int k = 0 ; k < OUTPUT_DIMENSION/T_OUT; k+T_OUT )
{  output_vector[k][1] = bias;

// this loop is done in parallel with a tiling factor of T_IN
for (int i = 0 ; k < INPUT_DIMENSION/T_IN; i+T_IN)
{
   output_vector_tmp[k] = input_vector[i][0]*weights[i][k]+
   input_vector[i][1]*weights[i+1][k]+;
   output_vector [k][1] += output_vector_tmp[k];
}
}
```

This means that architecture processes two neurons at each time and for each neuron two inputs at time are evaluated.

It's important to underline that these tiling factors doesn't not only influence the number and configuration of *PEs* but also in which way the data are stored in memory and memory configuration.

Indeed $T_{in} > 1$ has as consequence that more than one input at time has to be processed so it is necessary to partition the memory with the same tiling factors of T_{in}, it means that instead to have a big memory with all input features/activations the data are divided in T_{in} smaller *RAMs*.

The consequence of this choice doesn't impact on memory storage because required memory storage remain the same but permits to improve the concurrency of algorithm.

In equation 5.2 the first term indicates how many iterations are needed in order to compute the output of T_{out} neurons, the second term indicates how many times it is necessary to iterate the inner loop in order to process the complete layer.

For better understanding we can do an example.

In figure 5.1 there are two layers (*l-1, l*) of neural network where $N_{l-1} = 4$ and $N_l = 8$.

For instance setting $T_{in} = T_{out} = T = 2$, there are the following consequences of neural network processing:

- 2 neurons are processed concurrently at each algorithm step and each of them accepts 2 activation inputs and 4 weights at each step.
- $N_{l-1}/T_{in} = 2 =>$ it means that computing the output of 2 neurons requires 2 algorithm steps.
- $N_l/T_{out} = 4 =>$ it means that computing the output of all layer neurons requires that these 2 steps have to be repeated 4 times for a overall loop iterations of 8.

In figure 5.1 the example of data working flow is showed in a schematic way where is possible to see for each algorithm step which are the inputs and the outputs and which computation is done at each step.

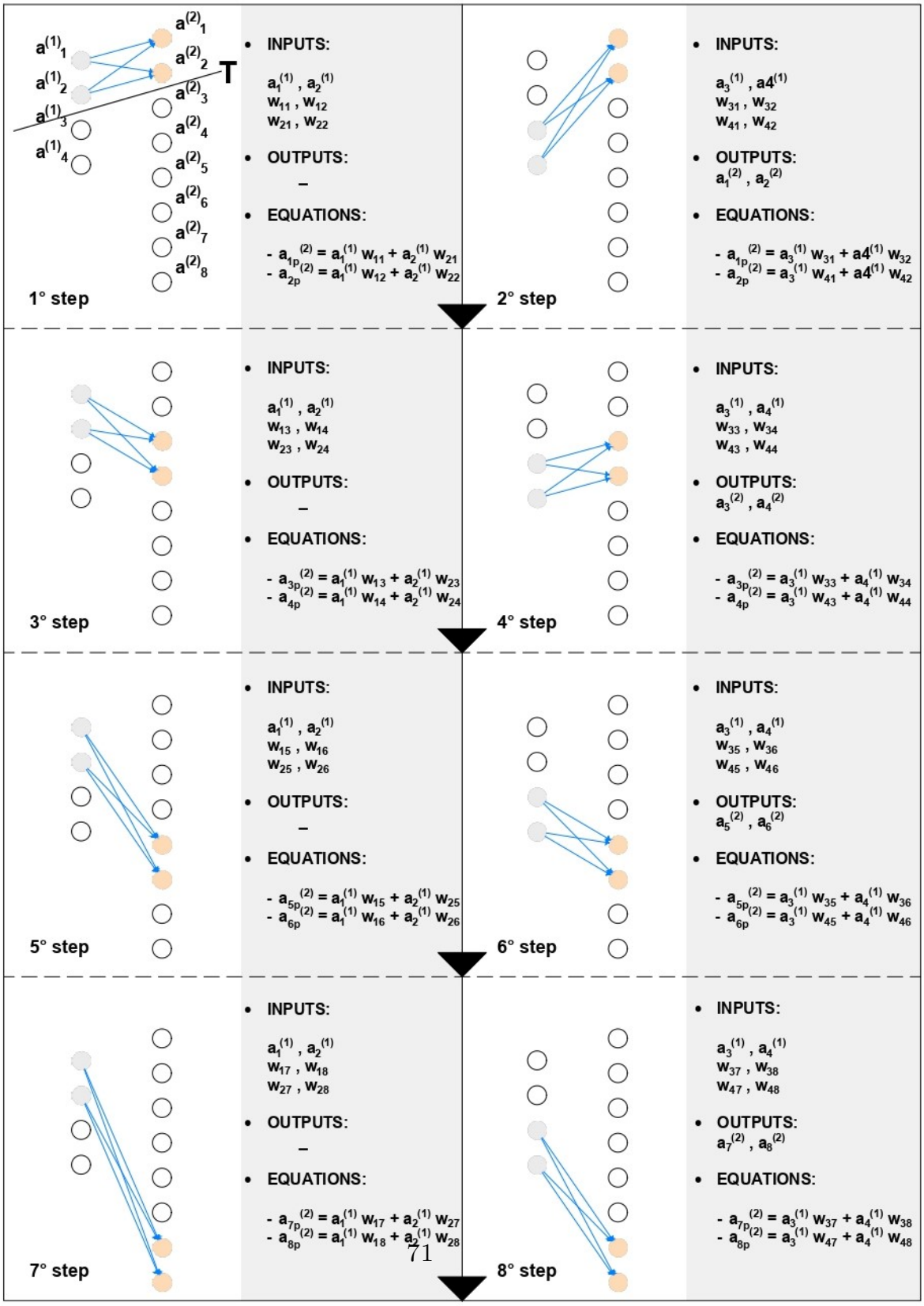

Figure 5.1: Example of data working flow for a unrolled neural network.

Figure 5.2 shows the difference phases that involves during layer processing showed in figure 5.1.

They consists in read the activations, read the weights and write the final output. The different colors in the image underline that it needs to read T activations and T^2 weights at each step. After T steps it is possible to write the final output for the first T neurons and after that it is possible to restart the computation for the next T neurons.

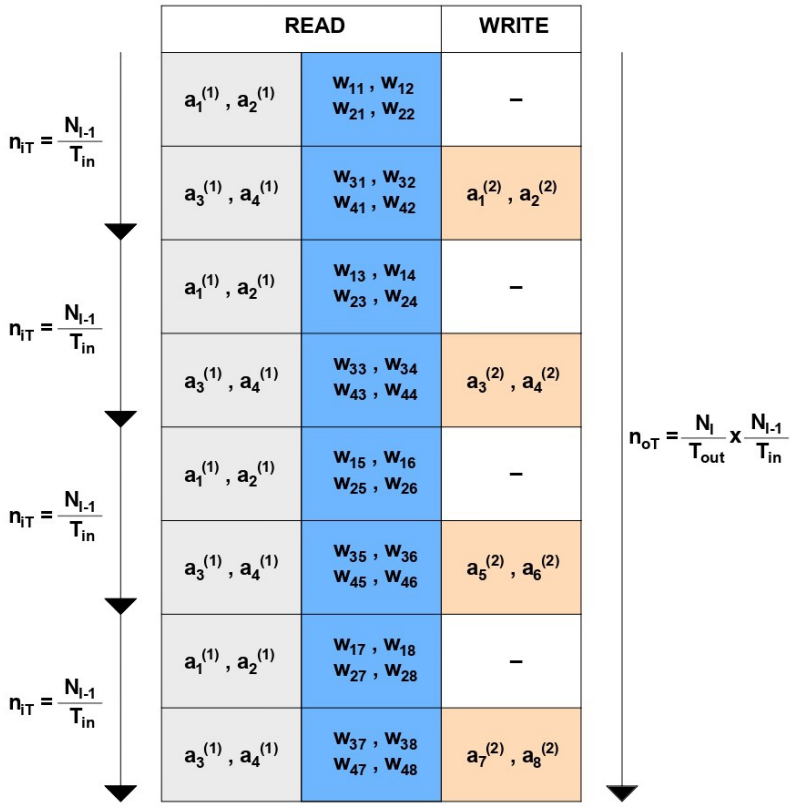

Figure 5.2: Example of timing working flow for a unrolled neural network.

In the next sections the various components of architecture will be described according the different tiling factors.

The same notation about N_l and N_{l-1} will be maintained where:

- N_l: identify the number of neurons for a generic l layer.
- N_{l-1}: identify the number of neurons for a generic *l-1* layer.

5.2 Neural network layer architecture

An example of complete neural network architecture is showed in figure 5.3. In the showed example $T_{in} = 3$ and $T_{out} = 3$.

It consists of datapath and control unit:

1. **Datapath unit** is composed by main three different elements:

   - *RAMs*: they are necessary in order to store the input features for the first layer or the activations for the other layers. The number of memories that it needs to use is directly linked with the T_{in} factor. If the architecture has $T_{in} = 1$ means that computational block uses just one input at each time and so the architecture will have a single memory with all input features/activations. If the architecture has a $T_{in} > 1$ then it has the same data stored in T_{in} memories. In this way it is possible that each *PE* (*MAC-memories*) can elaborate T_{in} inputs at time.

   - *MAC-memories*: *MAC-memories* are the *PEs* of the layer. Their aim is to compute the output of all neurons in the current layer. The number of different *MAC-memory* instances is T_{out}. It means that T_{out} different neurons are processed in parallel way. Each *MAC-memory* has own *ROM* that contains the weights. It is fundamental take care about in which way the weights are storage in different *ROMs* since they have to match with the right activation. An important block inside each *MAC-memory* is the *MAC-parallel*. It reads weights from *ROM* and after that multiply these last ones with T_{in} activations and accumulates the result until it finishes to compute the output neuron. The final result is passed through the activation function and moved to buffer.

   - *Buffer*: buffer permits to right delay the outputs that are computed in the current layer before they will be stored in the *RAMs* of the next layer. It's important the using of buffer because when perform a multiple inference gives the time needed to read the data from *RAMs* in the next layer. Contrary, the outputs data of the current layer can overwrite the *RAMs* data before the computation is finished.

2. **Control unit** is able to give the right control signals based on status signals from datapath. In order to have better control on datapath the control unit is the factoring of two *FSM*. The two *FSM* are in charge to do the two different purpose of the architecture: read data from previous layer and when data are ready process them.

   (a) *FSM load-data* : the main aim of this *FSM* is upload the data in the *RAMs* layer from the buffer of the previous layer or from *BRAM* if it is the first layer.

   (b) *FSM elaborate*: the main aim of this *FSM* is to control the elaboration of *MAC-memories* thanks different counters that take into account about iterations loop presented in the equation 5.2. Besides this *FSM* controls also the signals of the buffer in charge to permit that data moving on the buffer is correct.

In the next sections possible design exploration for hardware neurons implementation will be studied in deep since it represents the actual *PE*.

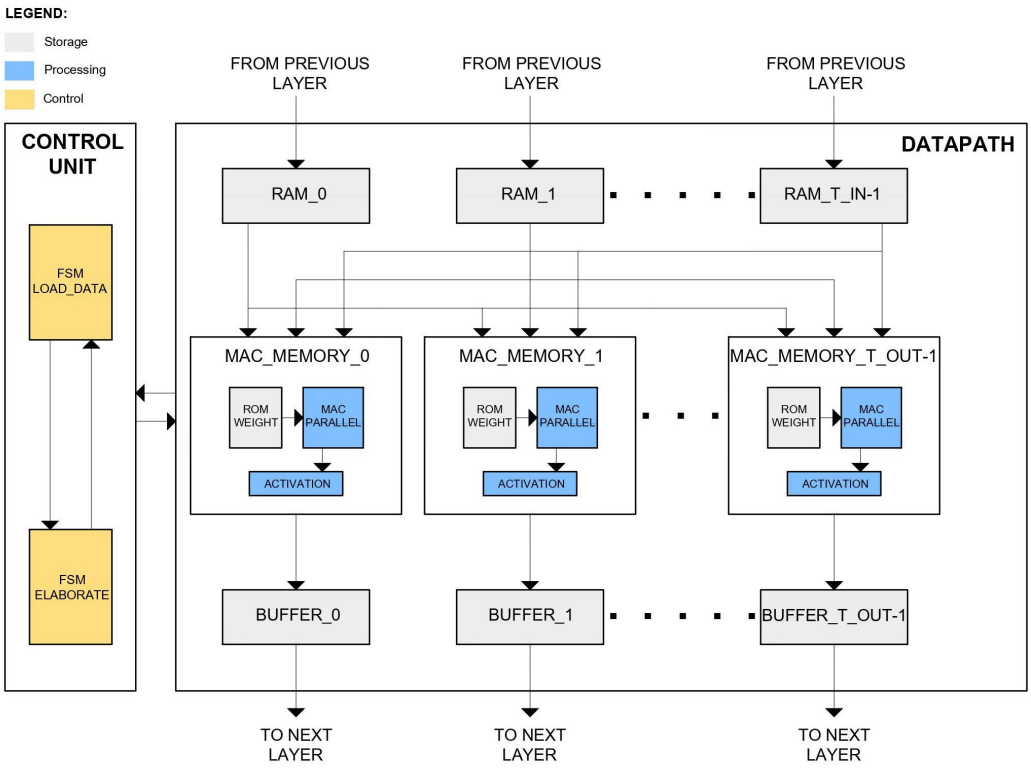

Figure 5.3: Example of layer neural network complete architecture.

5.3 Neurons design for tiling architecture

Some of the most important parts in the layer architecture are the different T_{out} *MAC-memories* instances. They simulate a number of T_{out} PEs that works in parallel way.

It is important to underline that if $N_l > T_{out}$ it means that more than one neuron will be processed by singular the *PE*.

Each of these ones is divided in three different components:

- *MAC-parallel*: it is able to compute the multiplication between a parametric number of T_{in} inputs and their corresponding weights. It simulates the behaviour of single neuron since it has just one output and it is able to process the output for just one neuron at each time.
- *ROM-weights*: it can contain the weights values or encoded weights. It is clear that when memory contains encoded weight is necessary to add a coder in order to decode the weights to their actual values. The possibility to use an actual memory weights or encoded weights is a choice that depends of neural network model. Also the type of encoding for the weights (1 bit, 2 bits or more) is a parameter that it is possible to change. It is important to say that when weights encoding changes also the coder changes.
- *Activation-function*: it is in charge to perform the activation function that it is used in network. The implemented activation function is the $ReLU$ described in 2.1.1.

The parameters that are defined T_{in} and T_{out} are user parameters that it is possible to change before the synthesis taking into account the target resource utilization, throughout and the high level parameters of neural network like the total number of the neurons of the layer or the total number of activation that have to be read by the layer.

The parameters T_{in} and T_{out} are defined at layer level on the hierarchy, so in the same neural network is possible to have different tiling factors for different layers.

The only one important constrain that it is need to maintain in the choice of these parameters is that T_{out} in the layer l has to be equal to T_{in} of $l+1$. This is fundamental for matching the output of one layer to the input of the next layer.

5.3.1 Neurons designs exploration

Different tiling factors involves in different possible architectures that can simulate the behaviour of a layer of neurons.

In figure 5.4 there is an example of these.

We investigate the four limit cases, starting from the standard one with no degree of concurrency case **a)** to the full concurrency architecture case **d)**. There are also two intermediate cases **b)** and **c)** in which architecture uses just one degree of concurrency, inputs concurrency for the case **b)** and output concurrency for the case **c)**.

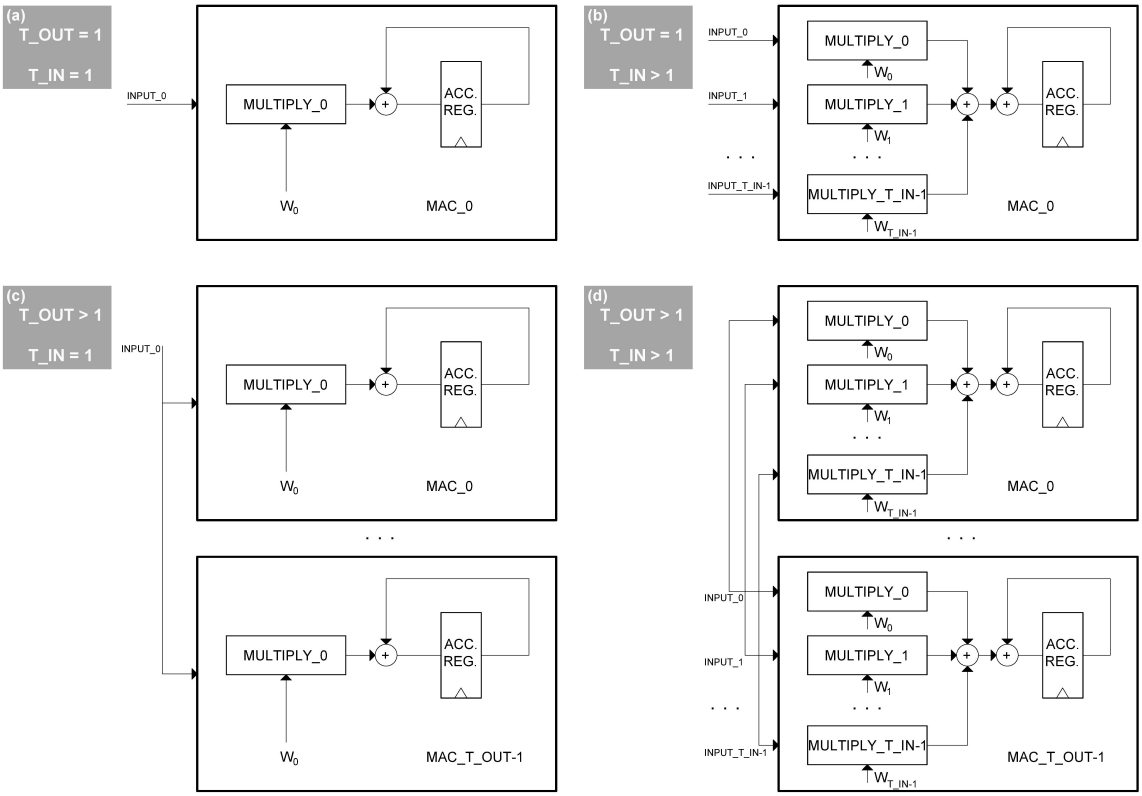

Figure 5.4: Design explorations for different hardware architecture neurons based.

We want analyze the different cases study in order to understand which are the advantages and drawbacks for the different configurations.

- Critical path: the case **a)** and **b)** have the same critical path, it is the path between the multiply block and the adder block. In order to break it, it will be possible to add a pipeline level between the two blocks. The case **b)** and **d)** have the same critical path and it is the path between the singular multiplier, the tree adder block and the adder. Also in this case it will be possible to add a pipeline level between multiplier and tree adder block but the resulting critical path is greater than the first case. Especially when T_{in} is high then the tree adder block deep increases and this can limit the max possible frequency. But it's important to underline that it can to decrease the max possible frequency but it increases the concurrency so the resulting throughout could be the same.

- Resources & loop unrolling compression: the case **a)** uses the minimum possible resources and it has unroll factor equal to 1. The other cases visible in table 5.1 show that unroll factors are directly connected to the number of used resources. The number of registers can be increased if a pipeline levels are applied to architecture. The multiplier that it is considered in a general case is normal multiplier, so it is a big and slow component especially when the number of bits is high. But performed model compression suggests that it is possible in some cases to use another type of multiplier like shifter multiplier, constant multiplier or partial binary multiplier that permits to save resources respect a normal multiplier.

- Memory requirement: the loop unrolling has an impact also on the memory requirement since it changes the number of inputs needed from *RAMs* and the number of needed weights from *ROMs* in order to perform the *MAC* operation in efficient way (using all available *PEs*). As a consequence become necessary to partition the memory in order to have more than one input at each cycle.

Table 5.1 and 5.2 summary the result in terms of critical path, resources utilization and memory requirement respectively.

Table 5.1: Design results evaluation for different architecture.

Design	*Multipliers*	*Adders*	*Registers*	*Critical path*	*Unroll*
a)	1	1	1	$t_{adder} + t_{multiply}$	1
b)	T_{in}	T_{in}	1	$T_{in} \cdot t_{adder} + t_{multiply}$	T_{in}
c)	T_{out}	T_{out}	T_{out}	$t_{adder} + t_{multiply}$	T_{out}
d)	$T_{out} \cdot T_{in}$	$T_{out} \cdot T_{in}$	T_{out}	$T_{in} \cdot t_{adder} + t_{multiply}$	$T_{out} \cdot T_{in}$

Table 5.2: Memory requirements for different architecture.

Design	*# of RAMs*	*# of ROMs*	*# Req. inputs*	*# Req. weights*
a)	1	1	1	1
b)	T_{in}	1	T_{in}	T_{in}
c)	1	T_{out}	1	T_{out}
d)	T_{in}	T_{out}	T_{in}	$T_{in} \cdot T_{out}$

We choose to implement the design **d)** for *PEs* for our architecture in which the singular *MAC* simulates the behaviour of singular neuron with a parametric number of inputs like in case **b)**.

Besides different instances of *MACs* simulate the behaviour of the different neurons that works in a parallel way.

The singular *MAC* is called *MAC-parallel* and it will be study in deep in the next section.

5.3.2 *MAC* parallel design overview

In last subsection possible designs for hardware neurons implementation are introduced.

Now we want focus on the single neuron implementation done with the component called *MAC-parallel*. The presented architecture has the same elements described in the case **b)** and further two control signals:

1. *MAC-enable*: this signal permits to process the inputs data just when they are available and valid. This is useful because when data are not valid they are not processing and the switching activity in following blocks doesn't change so it is applied a sort of data gating that permits also to save power.
2. *Set-accumulator*: this signal load the set value on the accumulator register. The set value is the bias of the neural network and it can be choice like a input parameter.

In figure 5.5 is possible appreciate an example of *MAC-parallel* complete architecture.

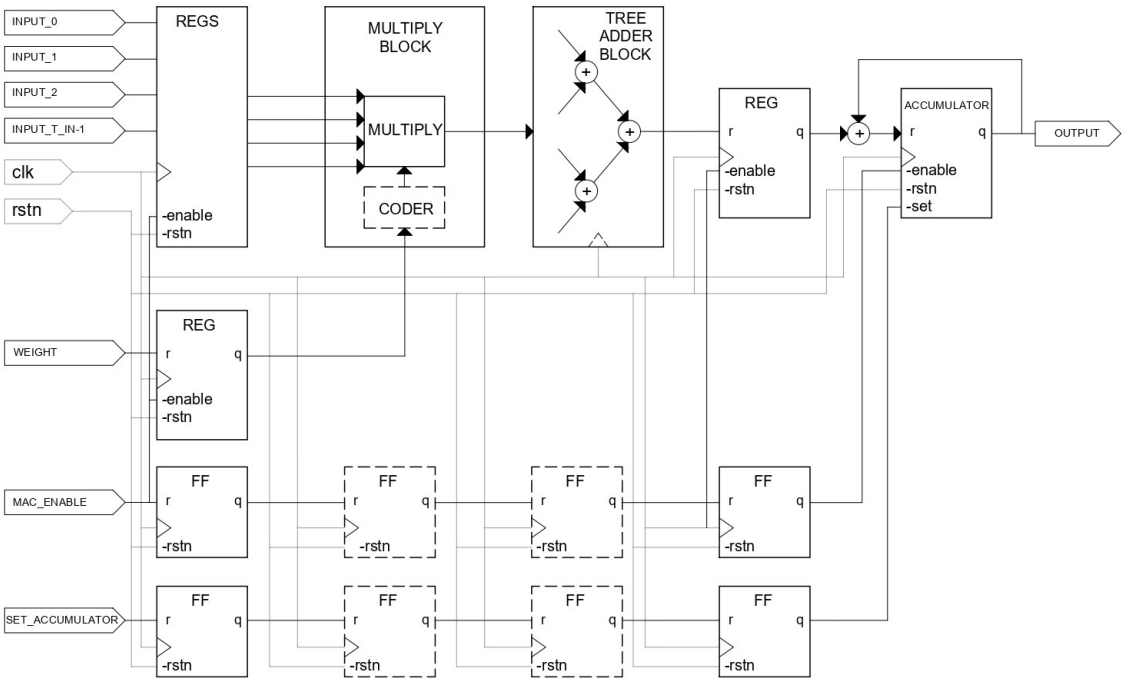

Figure 5.5: Example of *MAC parallel* architecture.

Pipeline levels

The *MAC-parallel* component presents two levels of pipeline.

The first level of pipe is also important because permits to interface the component with the inputs coming from external components.

The second level of pipeline is present after the tree adder block in order to break the possible critical path and so speed up the frequency of component.

The three adder block is purely combinatorial block but it is possible to transform it in a pipelined adder adding registers after each steps like showed in 5.6. The possibility to choose a pipelined or combinatorial version depends on a generic parameter present in a *MAC parallel* component.

The reason is that when T_{in} is high then the critical path between the multiplier and tree adder can limit the frequency. When pipeline levels increase it needs also to increase the pipeline levels of the same amount like showed in dashed FF in figure 5.5.

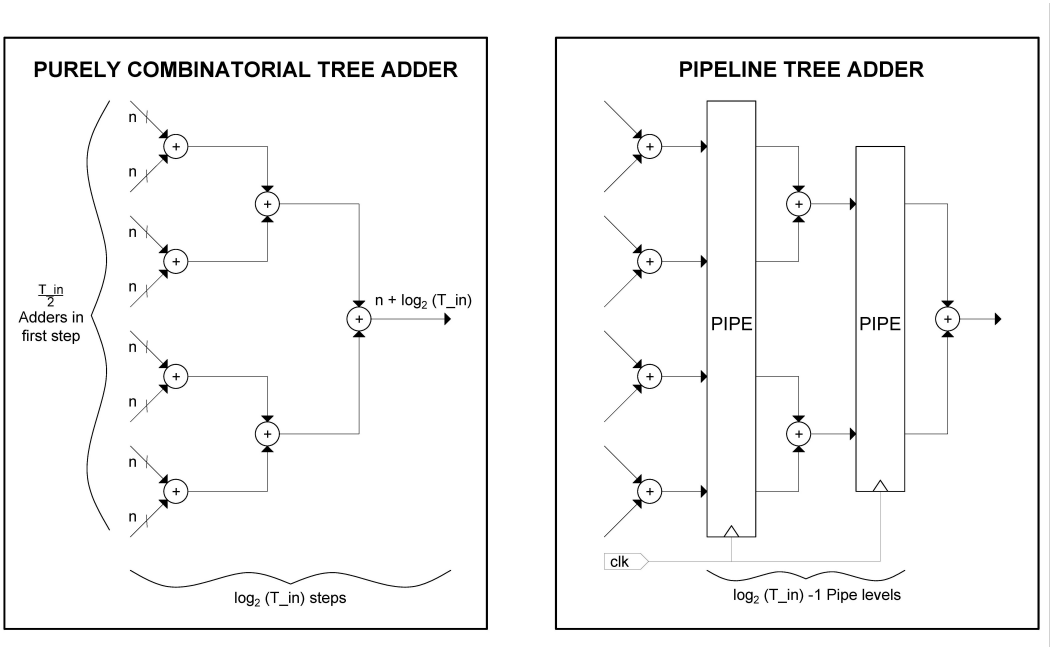

Figure 5.6: On the left: tree adder parameters. On the right: pipelined tree adder.

Coder

The dashed coder block can be added when encoded weights are used. Coder scope is translate the encoded weights to their actual associated values, the values of weights are a parameter that it is possible to choose. The coder is implemented with a multiplexer which output is selected through the select signals of the multiplexer connected to codified weight.

The coder is a component that it is possible change if the type of encoding (number of bits associated to one weight) is different.

We focus on use coder for two bits not uniform quantization of the weights. This because it presents different advantages in terms of memory compression and it can be hardware friendly (different types of optimized multipliers can be adopted).

The usage of the coder is necessary when encoded weights are all possible generic values but for power of two weights and binary weights it is possible not use the coder and this hardware overhead can be avoid and as a consequence the architecture will use less resources.

The complexity (# hardware resources) of the coder increases with the number of bits used for encode the weights like showed the relationship in table 5.3.

In order to compare how the hardware needed for the coder increases in table the comparison of area and critical path is done in term of 2 way multiplexer (for reproduce the same unit component).

In any case the coverage behavioral function implemented during decoding is a $2^n - to - 1$ multiplexer where n is number of bits used for weight encoding.

Table 5.3: Single coder design evaluation for different number of bits.

# bits encoded weight	*# values weight*	*# muxes 2-to-1*	*Critical Path*
n	2^n	$2^n - 1$	$t_{mux} \cdot n$

The number of multiplexer 2-to-1 in table 5.3 is true for $T_{in} = 1$ in other cases it will be T_{in} times than single coder. This because it needs a coder (multiplexer) for each different encoded weight.

In figure 5.7 there is an example of coder implementation with showed parameters.

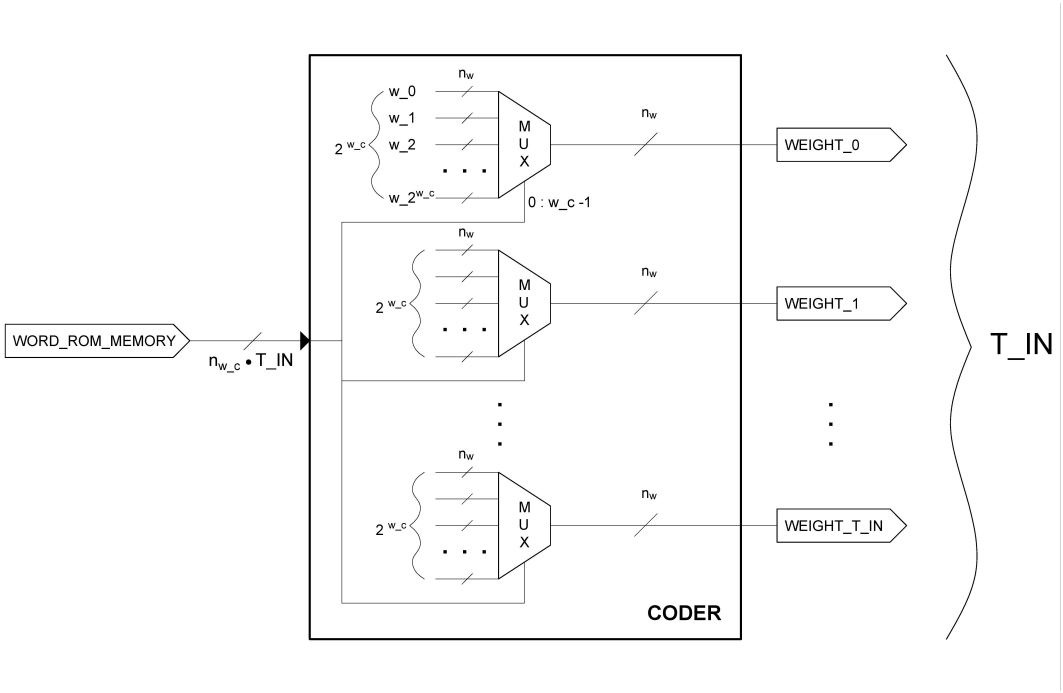

Figure 5.7: Example of coder architecture.

Data parallelism

During the elaboration of *MAC-parallel* is important to maintain the good precision of computation. It depends on type of hardware (multiplier or adder) but also it is connected with N_{l-1}. This because N_{l-1} impacts on how many adder iterations have to do in order the complete the output neuron. Data parallelism is important in charge to not lose precision but not using to much useless bits.

It is important that the different components have different number of bits like showed in table 5.4. Correct alignment of the data and correct sign extension is done when necessary in order to match the component size.

Table 5.4: Parallelism evaluation for *MAC-parallel* component.

Activation	*Weight*	*Multiplier*	*Tree adder*	*Accumulator*
N	M	N+M	(N+M)+$\log_2(T_{in})$	(N+M)+$\log_2(N_{l-1})$

After accumulation, the precision of the final output activation is typically reduced to N bits, the reduced output precision does not have a significant impact on accuracy if the distribution of the weights and activations is centered near zero such that the accumulation would not move only in one direction [1].

Latency and throughput

MAC-parallel computes the output of one single neuron at each time. Since in a *FC* neural network all the activations have to multiplied with their proper weights the partial result is accumulated until the neuron output can be moved to higher hierarchic component.

The number of cycles that a *MAC-parallel* will need to compute all the operations that have to be done for output neuron can be calculated with following formula:

$$cycles = \frac{N_{l-1}}{T_{in}} + 3 + [\log_2(T_{in}) - 1] \tag{5.3}$$

where the last term is added just for pipelined tree adder, in other case is not considered. The three extra cycles correspond to the neuron's latency to produce its first result, are added to the total time and are due to the need to fill the pipeline of the workflow of the neuron. In order to not lose cycles the bias values set the accumulator register in the first latency cycle.

5.3.3 Multiplier blocks

The fixed point multiplier block has an area $\simeq n^2$ and delay $\simeq n^2$ where n is the number of input bits.

Different types of multipliers are take into account since the model compression results operated by *MATLAB* showed an acceptable loss of accuracy in some cases.
The cases study are the following ones:

- *Generic multiplier*: it accepts any values of operands.
- *Constant multiplier*: it accepts any value from one operand but the other is constant.
- *Shift multiplier*: it accepts any value from one operand and apply a properly arithmetic shift. It works just for power of two weights.
- *Partial binary multiplier*: it accepts any value from one operand and the second operand can be -1 or 1.

In the *MAC* operation applied to neural network the value that it can be change is the input features or the activation, instead the value that it is constrained is the weight.

The analysis is concentrated about not uniform compression weights with two bits (except for binary multiplier) so the implementation of the constant multiplier and shift multiplier consists of four constant multipliers following by a multiplexer and four shift multipliers following by a multiplexer respectively like showed in 5.8.

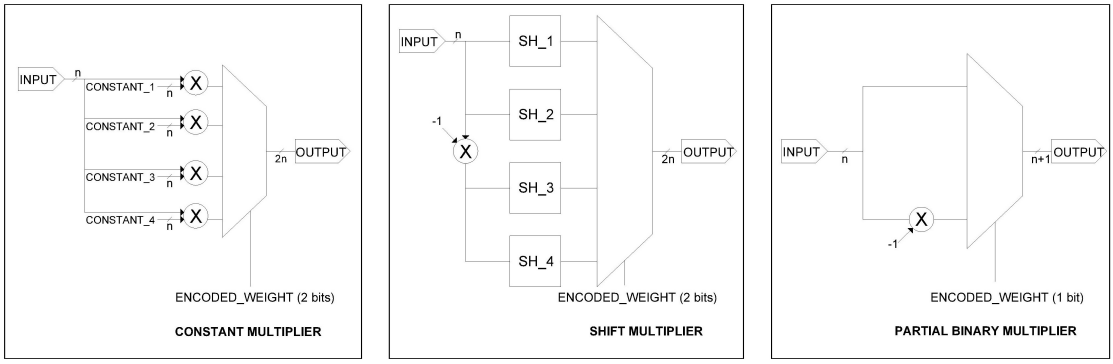

Figure 5.8: Different multiplier blocks.

5.4 *MAC*-memory

The *MAC-memory* component contains the *MAC-parallel* and a *ROM* like showed in figure 5.9.

The neurons that are associated with a single *MAC-memory* can be vary and depend on neural network model (N_l) but also on hardware parameter (T_{out}). This because the first is the number of neurons for the l layer, the second determines how many of them are processed in a parallel way. The number of neurons associated to one *MAC-memory* determines *ROM* weight size.

The control signal *write-output* is added in charge to permit that when single neuron computation is finished the output can be passed to another register and so the accumulator register can be resetted and start again a new neurons elaboration.

The *write-output* control signal is delayed of two cycles because the *MAC-parallel* controls signals are also delayed of the same cycles. The dashed *FF* takes into account the possibility that the *MAC-parallel* increases its pipeline levels (due to pipelined tree adder) and as a consequence also this control signals will be delayed.

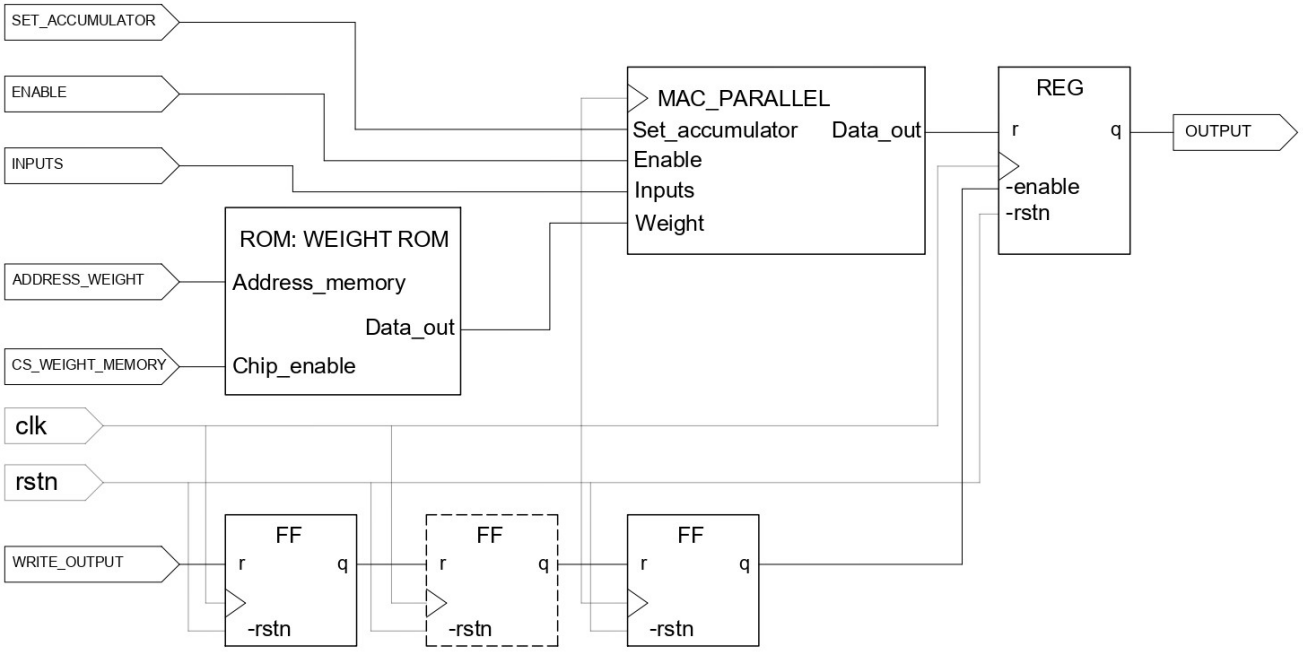

Figure 5.9: *MAC-memory* architecture.

5.4.1 Weights *ROM*

It is important to notice that the weights are stored in *ROM*, this means that we can only read them but not change them. This has been implemented this way because we are training the network off-line, the weights that are loaded into the *ROMs* must come from a process of training done not at the hardware level.

It is important to say that the *ROM* is asynchronous. This means that when the address that we want to read from the *ROM* changes the data out will be available shortly after that and the weight will be accessible. The reason of having implemented the *ROM* in an asynchronous way is because otherwise we would have to wait for a clock cycle to obtain the data we want and it will either make the neuron slower or make the control of the addresses more difficult. This choice was made knowing that the combinatorial path before the following register is not long so this will not interfere with the timing constraints.

The control signals that are the inputs of weights *ROMs* are:

- *Address-weight*: it has to take in consideration which is the correct neuron for the current elaboration (due to the fact that more than one neuron can be associated with a single *MAC-memory* and its right weight matching with the activation. This signal comes from the datapath of the layer thanks to two counters, one of them takes into account iterations for a single neuron (N_{l-1}/T_{in}), the other how many neurons are just processed (N_l/T_{out}).
- *CS-memory-weight*: it permits to turn off the memory outputs when they are not needed, in this way it is possible save power because no new data on data-out of memory. It comes from the control unit and depends in which state of elaboration the machine is working.

It is important to underline that the memory size in terms of precision of singular weight impact on accuracy of the neural network but also in a hardware key metrics like the power consumption and utilization resources due to the fact that the number of bits of the weights influences the parallelism of the structure. For this reason we focus on weight compression with an acceptable loss of accuracy on our model in order to reduce the storage constrain and as a consequence the complexity hardware.

The precision of the singular weight is not the only aspect that it is important to consider, it's fundamental also the deep of the memory ($\simeq N_l$) because it involves in different execution time (that increases with other fixed parameters) so in different energy and throughput.

Weight memory organization

In figure 5.10 it is possible to appreciate which is memory organization when $T_{in} = 1$. The total number of *ROMs* in the architecture depends on which tiling factor T_{out} is chosen.

The total storage constrain for the weights remain the same respect to the case with no memory partitioning but changes the way in which the weights are divided into different *ROMs*.

The control signals that are given to different *ROMs* are the same ones so in this way a regularization of data access is obtained.

In the example showed in figure 5.10 it is possible to see that with $T_{out} = 2$ in the first memory we stored the even weights and in the second memory the odd ones. This because in this case two neurons at each time are computed concurrently.

In general case with $T_{out} > 1$, the organization follows the same principle to fill the memory.

The size of word memory depends in this case with $T_{in} = 1$ only on the precision choose by neural network model for the weights.

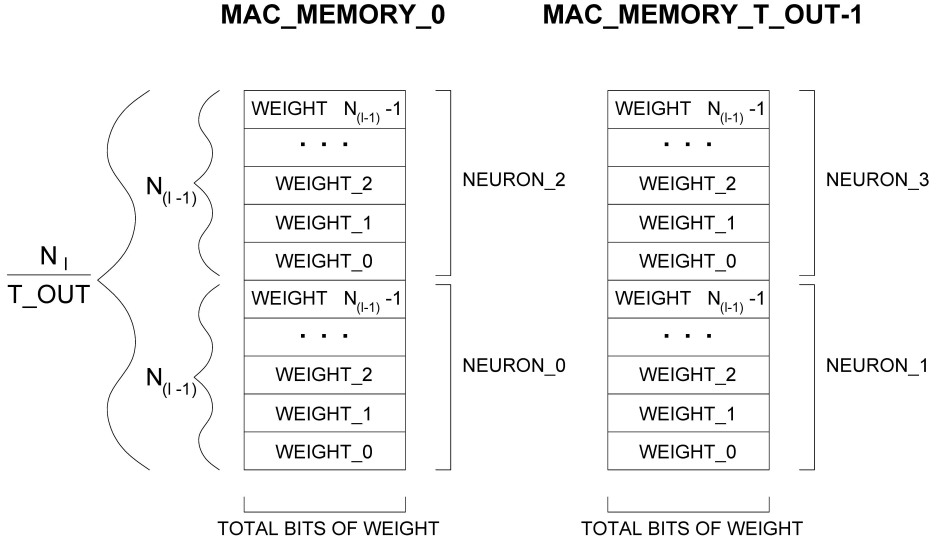

Figure 5.10: Memory weight organization with $T_{in} = 1$.

In the general case when $T_{in} > 1$, the required weight for not lose efficiency (using all available *PEs*) become $T_{in} \cdot T_{out}$.

For manage this situation there are two possible available options:

- Using T_{in} different memories for each different T_{out} *MAC-memory*.
- Using T_{out} different memories for different T_{out} *MAC-memory* but using a different size for memory word obtained by the product between singular weight bits number and T_{in}. Increasing the word size the consequence is that the memory deep decreases like showed in figure 5.11.

The option that we have chosen is the second one. With this option, the management of the weights is easier and it is more flexible respect to the first one. Besides the first one has no particular advantages respect to the second one.

One possible drawback of the second option can be when the T_{in} factor is high or the number of weight bits is high because the word memory size increases and the memory access can be slower.

Table 5.5 summaries the size of memory for different loop unrolling factors.

Table 5.5: Weights *ROMs* parameters for different tiling factors.

Cases	*# ROMs*	*ROMs size*	*Word ROMs size*
$\mathrm{T}_{in} = 1, T_{out} = 1$	1	$\mathrm{N}_l \cdot N_{l-1}$	bits_{weight}
$\mathrm{T}_{in} > 1, T_{out} = 1$	1	$\mathrm{N}_{l-1}/T_{in} \cdot N_l$	$\mathrm{bits}_{weight} \cdot T_{in}$
$\mathrm{T}_{in} = 1, T_{out} > 1$	T_{out}	$\mathrm{N}_l/T_{out} \cdot N_{l-1}$	bits_{weight}
$\mathrm{T}_{in} > 1, T_{out} > 1$	T_{out}	$\mathrm{N}_l/T_{out} \cdot N_{l-1}/T_{in}$	$\mathrm{bits}_{weight} \cdot T_{in}$

In general case the overall deep memory and word memory size are the following ones:

$$ROM - size = \frac{N_{l-1}}{T_{in}} \times \frac{N_l}{T_{out}} \tag{5.4}$$

$$Word - memory = T_{in} \times bits_{weight} \tag{5.5}$$

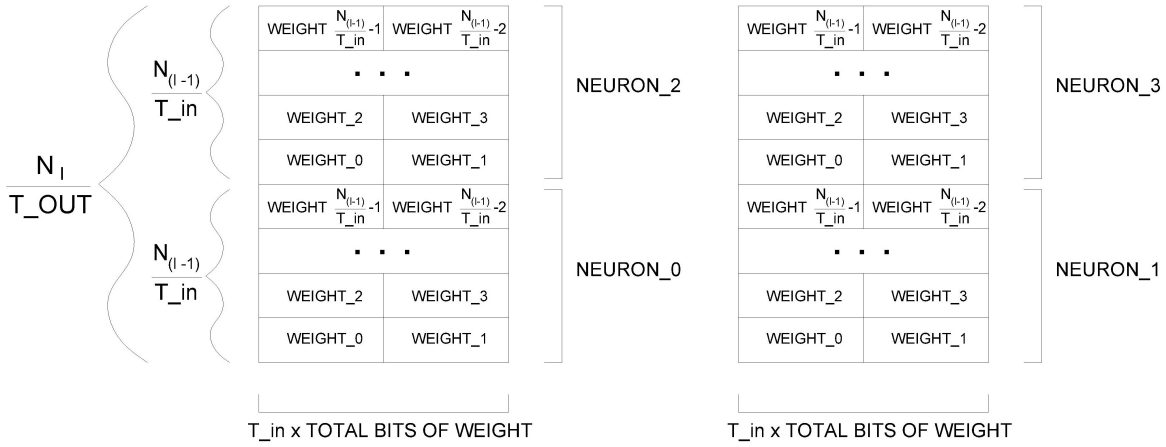

Figure 5.11: Memory weight organization with $\mathrm{T}_{in} = 2$.

In equation 5.4 the first term indicates how many memory locations are necessary for the single neuron elaboration and the second term indicates how many neurons are contained in a single *MAC-memory* instance.

The loop unrolling permits to efficient partition the memory using more smaller memories respect to use a bigger one in this way theoretically the memory access can be quicker and consumes less power (less capacitance in the path to drive).

Besides it is possible to increases the concurrency. It's important to underline that overall memory accesses to complete a neural network layer computation when $T_{in} > 1$ decrease according a factor T_{in} like showed in equation 5.6.

$$\#Memory - accesses = N_l \times \frac{N_{l-1}}{T_{in}} \tag{5.6}$$

It is an important point since in a *FC* neural network the amount of weights is important and the memory accesses and data movement are one of the most important sources of energy consumption.

5.5 Datapath layer

The layer datapath is composed of the different components that are introduced in section 5.3. Besides, the layer datapath contains different counters that are used by controller to manage the different elaborations that it is necessary do to.

It is possible to identify the following needed counters:

- *Counter load-data*: it is used by the *FSM load-inputs* in charge to write the activation input in the *RAMs* of the layer. When its TC signal is asserted the *FSM elaborate* can start to work. The max-value of this counter is N_{l-1}/T_{in}.
- *Counter neuron-iteration*: it is used by the *FSM elaborate* in charge to takes into account the current iteration for single neuron output. When its TC signal is asserted the *FSM* gives the signal to restart the computation for the following neuron. The max-value of this counter is N_{l-1}/T_{in}.
- *Counter layer-iteration*: it is used by the *FSM elaborate* in charge to takes into account how many neurons output are ready. When its TC signal is asserted the *FSM load-inputs* of the next layer is ready to move the data from the buffer to *RAMs* of the next layer. The max-value of this counter is N_l/T_{out}.

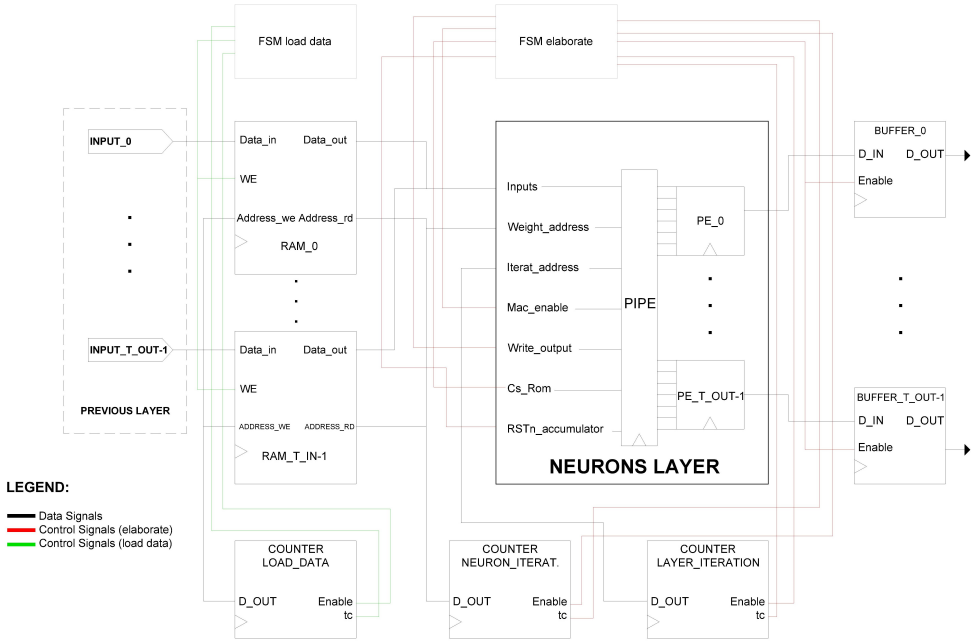

Figure 5.12: Datapath layer architecture.

5.5.1 Input *RAMs* and buffer

The input *RAMs* and buffer are the components that interface the *PEs* represented by different *MAC-memory* instances from the inputs (coming from another layer or the *BRAM* if the first layer) to the output that is is represented by another layer or the overall output.

The signals (except for the inputs data) that arrive in the different memories and buffers are the same in this way the architecture presents a regularize data access that permit an easier control.

Input *RAMs*

The *RAMs* are synchronous and dual ports (one for the input of the previous layer and the other for the output).

The number of different *RAMs* instances is T_{in} and the size of each *RAM* is defined in equation 5.7:

$$RAM - size = \frac{N_{l-1}}{T_{in}} \tag{5.7}$$

It is important to underline that the number of inputs *RAMs* are defined at layer level and not for complete architecture according to layer constrain.

It is important take into account the possibility that N_{l-1} is not divisible with T_{in} in this case it is necessary to add N_{l-1} mod T_{in} memory locations that they will be filled with zero bits and in this way not consequence in terms of precision loss.

The choice of using $T_{in} > 1$ permits to explore concurrency of the inputs because the different inputs (coming from different memories) are given to the same *PE* at the same time and in this way the iterations number of the algorithm showed in 5.1 for the inner loop decreases.

Another consequence of using $T_{out} > 1$ is that has an impact of number the *PEs* elements that are working in parallel. This aspect has the consequence that permits data reuse of the *RAM* data that are read one time and processed by T_{out} neurons, so the total number of accesses to *RAMs* memory decreases according a factor T_{out}:

$$\#Memory - accesses = N_{l-1} \cdot \frac{N_l}{T_{out}} \tag{5.8}$$

The *RAMs* input organization is important due to the fact that in a *FC* architecture all the inputs have to be multiplied with their proper weights.

In figure 5.13 there is an example of how data are divided from a singular memory to two different memory. In order to not change the data algorithm workflow the data are organized respecting the same order of the singular memory.

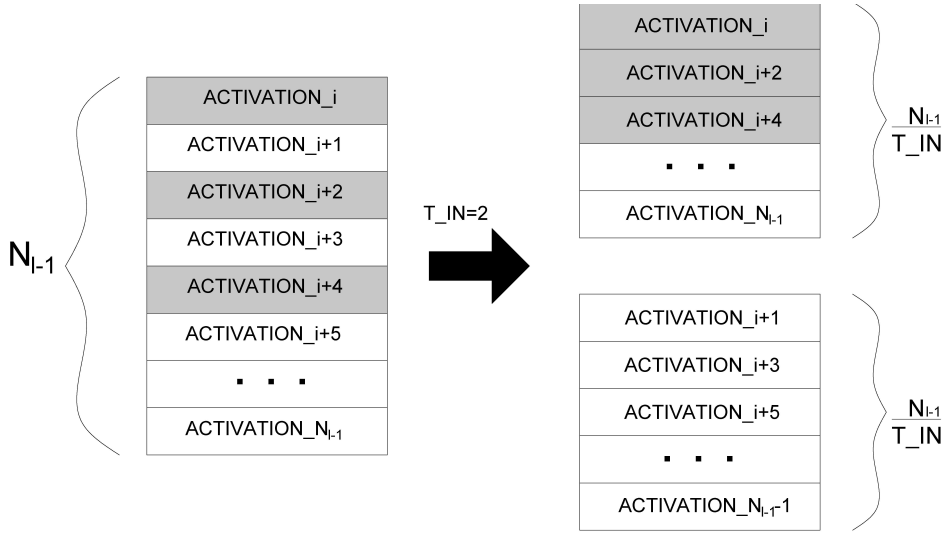

Figure 5.13: Example of inputs *RAMs* organization with $T_{in} = 2$.

Buffer design

The buffer is necessary to perform multiple data inferences due to the fact that in a general case the number of neurons of each layer can be different.

As a consequence the loop iterations of each layer can be different and the architecture has to be able to do the possibility do read the *RAMs* for the correct number of needed iterations.

Contrary, if the buffer is not inserted the new data overwrite the *RAMs* data before the complete elaboration is done.

The behaviour of the buffer is similar to a *FIFO* memory because the data go out from the buffer with the same order in which they are get in. The buffer size can be chosen like a parameter. It's important to underline that this parameter is constrained by neural network model (N_{l-1}, N_{l+1}) and target throughput performance.

5.5.2 Neurons layer

This component is composed by different T_{out} *MAC-memory* instances (called PEs in figure 5.12). The all signals that are the inputs of *MAC-memory* instances are delayed of one clock cycle due to inserted pipe level between the *MAC-memory* instances and input RAMs.

Before the data moves to buffer the activation function is performed is needed.

All *MAC-memory* instances receive the same control and data signals in this way the architecture present a symmetrical structure. The only difference is the content of *ROM* that determines different neurons. As a consequence each *ROM* has to present the same structure described in 5.4.1.

The number of neurons that it associated to each *MAC-memory* (*PE*) is:

$$\frac{\#neurons}{PEs} = \frac{N_l}{T_{out}} \tag{5.9}$$

In order to compute the right *ROM* address in the *neurons layer* there are two input signals coming from the two counters present in the datapath.

It is necessary use two different counters for two reasons:

1. The *counter neuron-iteration* is also the read address data also for input *RAMs* and so it has to scan N_{l-1}/T_{in} values in order to complete one complete read cycle of each *RAMs* and so it determines the output for T_{out} neuron. So in this way it is easier matching the activation with the right weight since they receive the same address.

2. The *counter layer-iteration* is important because it takes into account how many neuron outputs are just computed and it permits to know which is the current neuron elaboration.

The two output counters signal are mixed dynamically in the following way to determine which is the effective *ROM* address:

$$ROM - address = neuron - iteration + (iteration - layer * \frac{N_l}{T_{out}}) \tag{5.10}$$

Data gating

Equation 5.9 can return a value that is not an integer. This means that it is not possible divide the various neurons with different *PEs* in symmetrical way.

In order to manage this situation there are two possible options:

- It is possible lose some flexibility and set the constrain that the result of 5.9 is mandatory an integer.
- It is possible to insert some hardware overhead like which one showed in 5.14 that permits to gating the right amount of *MAC-memory* instances when they are not needed. In this way the switching activity in the gated block doesn't change and as a consequence it is possible to save power. It's possible to do it thanks a masking operation of *MAC-enable* signal during the last iteration where needed.

In order to guarantee the maximum flexibility the option number two is chosen.

The number of *MAC-memory* instances that are gated during last iteration is given by formula 5.11:

$$\#Gated - MACs = T_{out} - (N_l \mod T_{out}) \tag{5.11}$$

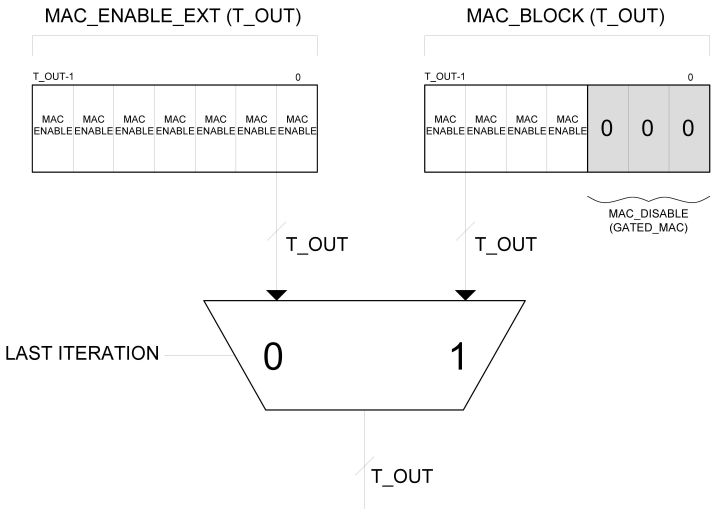

Figure 5.14: Masking *MACs* hardware.

5.6 Controller

Controller aim's is to read the status signals that coming from datapath and send the correct control signals.

The controller is divided into two *FSM* for each layer. The two *FSMs* communicate according a three flag signals for each layer.

1. *Input-load*: this signal is a sort of start of the layer elaboration because it permits to upload the data into input *RAMs*. When the signal is asserted the datapath begins to load the data into memory. This flag is disabled when *input-ready* is asserted.
2. *Input-ready*: this signal is asserted when all the inputs are stored in input *RAMs* and so they are ready to be elaborated. This signal is passed to the *FSM* elaborate in order to start the elaboration. This flag is disabled when *output-ready* is asserted.
3. *Output-ready*: this signal is asserted when all the outputs are in the buffer and so it is possible that the next layer upload the new available data and the current layer can also upload the new data for elaboration.

Figure 5.15 shows as the controllers for two different layer are connected between them and how the two *FSMs* of single layer are connected between them.

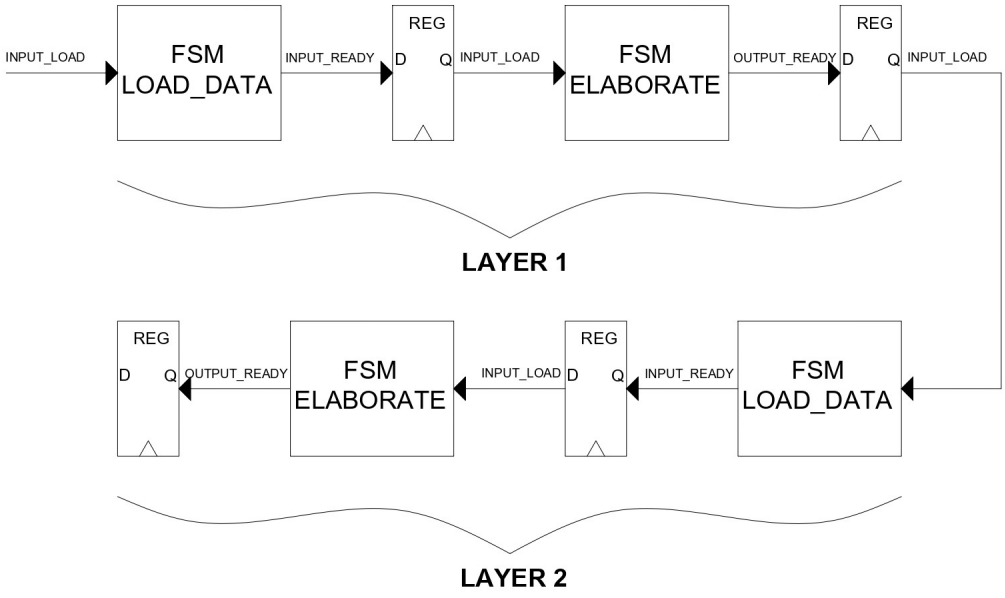

Figure 5.15: Example of complete controller connections.

Figure 5.16 shows the main asserted control signals that from control unit are sent to datapath and the various state of machine.

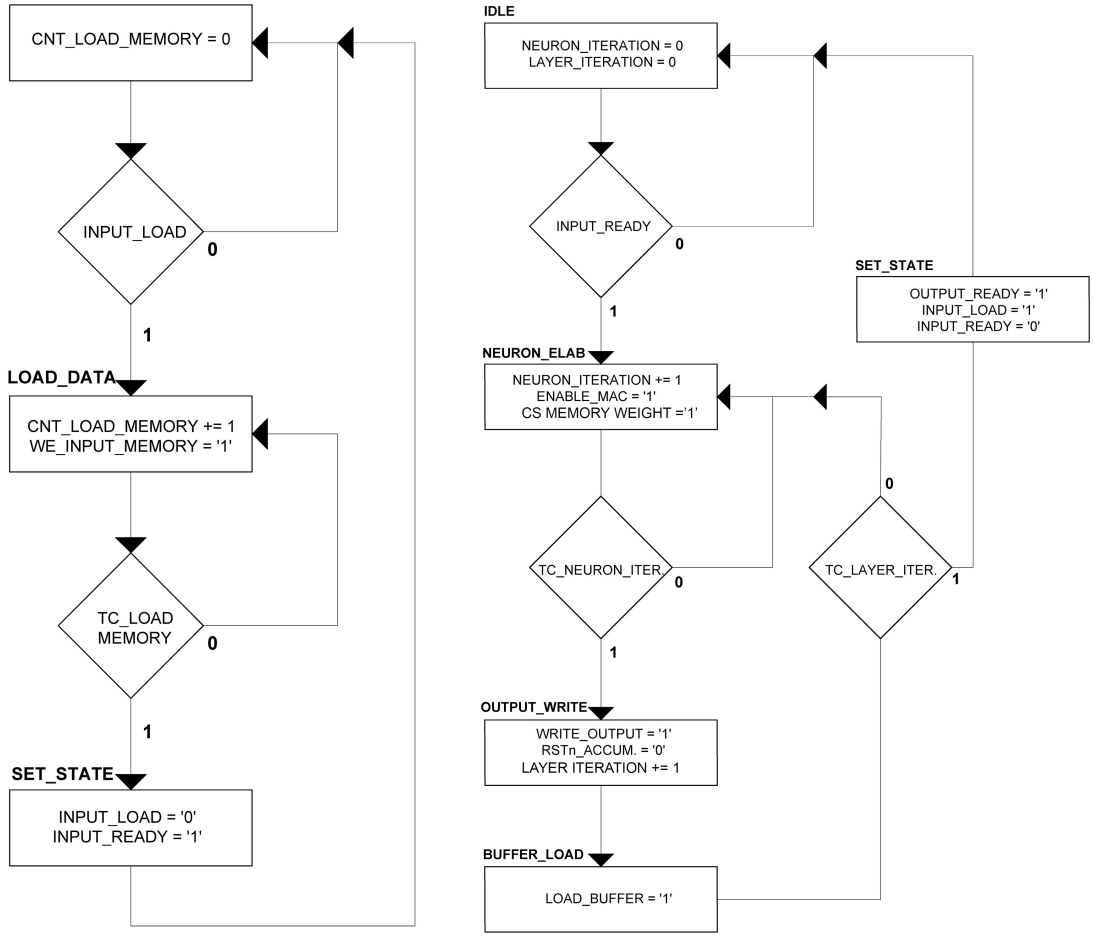

Figure 5.16: On the left: chart of FSM-load-dataASM chart of *FSM load-data*. On the right: *ASM* chart of *FSM elaborate*.

5.7 Optimization techniques

In this part there is a summary of how the optimization techniques presented in figure 3.7 are applied to the proposed design:

- PIPELINING: the design presents four pipeline levels in order to speed up the elaboration (increasing the working frequency) and so increasing throughput.
  1. Two levels are present inside *MAC-parallel* because it is composed by a multiplier followed by an adders and this can limit the max possible frequency.
  2. Two levels are present inside the *Neurons-layer* block. At the input because read data from memory can represent a bottleneck for the max frequency and at the output to take into account the delay introduced by activation function.

- LOOP UNROLLING: the design permits to choose different loop unrolling factors with the constrains discussed in previous sections.
  1. T_{in} has an impact on input *RAMs* organization, on *MAC-parallel* resources and on weight memory organization.
  2. T_{out} has an impact on *MAC-memory* organization, on weight memory organization and on buffer structure.

- MODEL COMPRESSION: the design is flexible in term of model compression since it permits to choose the precision of weights.
  1. Uniform quantization: there is the possibility to choose the number of bits for the weights.
  2. Not uniform quantization: there is the possibility to choose the quantization for the weights and using an appropriate coder in order to decode the weights. Based on different applied model compression it is possible to choose a different multiplier block to optimize the computation.

- REGULARIZE DATA ACCESS: all the memories in the architecture receive the same signals from the datapath and from controller, this permits an easier control of the structure that presents a symmetrical design.

Chapter 6

Detailed implementation of the neural network

This chapter results of the implemented design will be discussed. The first part will be focused on some *VHDL* details of design implementation. This because developed code is able to give the possibility to the user to change different parameters that during the design description in chapter 5 are analyzed and discussed. In the second part there are the results of implementation on XC7Z045 *FPGA* in terms of utilization resources and throughput of the neural networks taken in consideration and some components of the neural network. The test will be done on the same neural networks analyzed and discussed in chapter 4. One of the most important parameter that it is used in the various tests is type of multiplier, this because as discussed in chapter 3, multipliers based operation are the majority of neural network elaboration. In the final part there are the final considerations about personal evaluation, possible future works and conclusion.

6.1 Implementation strategy

To implement the proposed architecture we have used *VHDL* as hardware description language and the commercial tool Vivado [30], as we want to design the specific hardware of a neural network targeting *Xilinx FPGAs*.

The proposed *VHDL* code shows a hierarchic structure that permit to define the component starting from lower level (*MAC-parallel*) to arrive a complete neural network that it is composed by different layer instances. The structure of layer is well described in 5.5. The component at higher level defines the generic parameters of the component at lower level. A package in the highest level permits to define a complete neural network starting from a description of the parameters of the single layer.

The most important parameters that are possible to choose are the tiling factors that have the effects explained in 5.7 on the architecture. Other important parameters are the types of used multiplier (depends on data compression applied on the model) and the precision of the data input. The flexibility of the *VHDL* developed code give the possibility to adapt at each *FC* neural network, what it can be different is just the content and size of input *RAMs* and *ROMs* and the available *PEs*.

VHDL code has the flexibility according to the following aspects:

- Hierarchic organization: it permits to explore the "*divide et impera*" approach in which the design and the test of each component is more feasible. Besides, this approach permits also in the future some possible modifications easily for example changing the type of coder from two bits to three bits because it will be necessary change just the coder component and not all architecture.
- Generate construct: gives the possibility to iterate the hardware instances for parametric value of times for example when different tiling factors are used or give the possibility to instance or not an hardware in a particular situation, for example the case explained in 5.5.2.
- Generic construct: permits that every component has the parameters that are adaptable to every situations. For example the number of bits of the input data, the type of used *PEs* or at layer level the number of neurons that composes the layer.

More details about the developed code of various more important components described in the chapter 5 and the explained features are showed in appendix A.

6.1.1 Data files

As was explained before, every piece of data that enters a neuron is different in each neuron, and we have to access the right weight that corresponds to that neuron and that specific input. This means that in the same address in different *ROMs*, the weight corresponds to different inputs.

The weights that *MATLAB* provided us were a real number so they have to be changed from real to two's complement thanks to already developed [31]. In this function we have to provide both the number of total bits and the number of fractional bits due to the fact that we are using a fixed point signed representation of the data. Once the weights are transformed into two's complement, they have to be divided depending on the neuron that those weights belong to. When the weights are divided, they have to be ordered depending on the neuron that it corresponds to. First of all we need to know the size of the *ROM* in which the weights are going to be stored, which is described in equation 5.10. Knowing this parameter, we need to fill the exceeding positions with zeros.

The weights and the input features are upload in their respectively memories in the right order thanks to a *MATLAB* processing of the data respecting the same working flow also when loop unrolling is applied. The way in which these types of data are upload in the *VHDL* code are described in appendix A in the component called *ROMs*.

6.2 Result implementation

The results that are take into account after a post synthesis simulation are the following ones:

1. Resources utilization: *Xiling FPGAs* has $LUTs, FFs, DSPs, BRAMs$ in order to mapping the architecture. Different components of our architecture are tested with different parameters and at the end a complete neural network architecture is mapped.
2. Power analysis: some components are tested in the worst case condition ($E_{sw} = 1$), in order to know in which way power consumption varies with different configurations.
3. Throughput: it is described in 3.1.1.
4. Memory footprint: the constrains for memory weights are computed for different cases and they are associated with accuracy degradation when different model compression is applied. The accuracy is strongly connected with the number and the precision of the weights like explained in 3.1.3.

6.2.1 Memory footprint

In table 6.1 and in table 6.2 it is possible appreciate the memory constrain for the weights for different weight precision and for two different topology (changing the # of *HUs*). The different topology that are taken into account are the same that are discussed in the chapter 4 with *MNIST* data set.

The purpose of these tables is to see how different memory compression has an impact on accuracy degradation. This one is computed with the *error-rate*.

The error-rate and memory compression are computed in relationship with the case **a)** that report the maximum accuracy and the highest memory footprint.

The formula for the error-rate that it will be used is 6.1:

$$Error - rate = Accuracy_{case-a} - Accuracy_{other-cases} \tag{6.1}$$

The memory compression is computed dividing the needed memory footprint for the case **a)** for the needed footprint memory for the other cases showed in the tables.

Table 6.1: Weights *ROM* footprint with $784 \times 200 \times 10$ neural network topology.

Cases	*# W*	*# Bits*	*Footprint*	*Mem. Compr.*	*Error-Rate(%)*	*Ratio(M/E)*
a)	1568000	64	100.1Mb	-	-	-
b)	1568000	32	50.1 Mb	2x	0.12	17
c)	1568000	8	10.2Mb	8.4x	0.83	10
d)	1568000	2	3Mb	50.5x	3.6	14
e)	1568000	1	1.5Mb	67x	20.13	3

Table 6.2: Weights *ROM* footprint with $784 \times 25 \times 10$ neural network topology.

Cases	*# W*	*# Bits*	*Footprint*	*Mem. Compr.*	*Error-Rate(%)*	*Ratio(M/E)*
f)	196000	64	12.1Mb	8.27x	7	1.18
g)	196000	32	6.2 Mb	16.145x	7.34	2.19
h)	196000	8	1.5Mb	66.73x	8.08	8.25
i)	196000	2	0.4Mb	250x	11.05	23
l)	196000	1	0.19Mb	526.8x	22.45	23.46

The case **a)** represents the first original set of weights with 64 bits precision that report the highest accuracy ($\simeq 98.35\%$) but the maximum of requested memory.

The case **b)** and **c)** report the footprint for uniform quantization of the weights that our *MATLAB* model shows an acceptable loss of accuracy ($\simeq 0.8\%$ respect to the best one).

The case **d)** and **e)** report the footprint for not uniform quantization of the weights since the uniform quantization under 8 bits precision shows an accuracy degradation not acceptable. In this case the weights are chosen for the two different not uniform quantization according the methodologies explained in 4.3.1.

The other cases in table 6.2 represents the same types of quantization but using different topology. In this case model compression is applied reducing the number of overall weights.

Figure 6.1 summaries the results of the two tables reporting the memory compression for all different cases and the respective error rate.

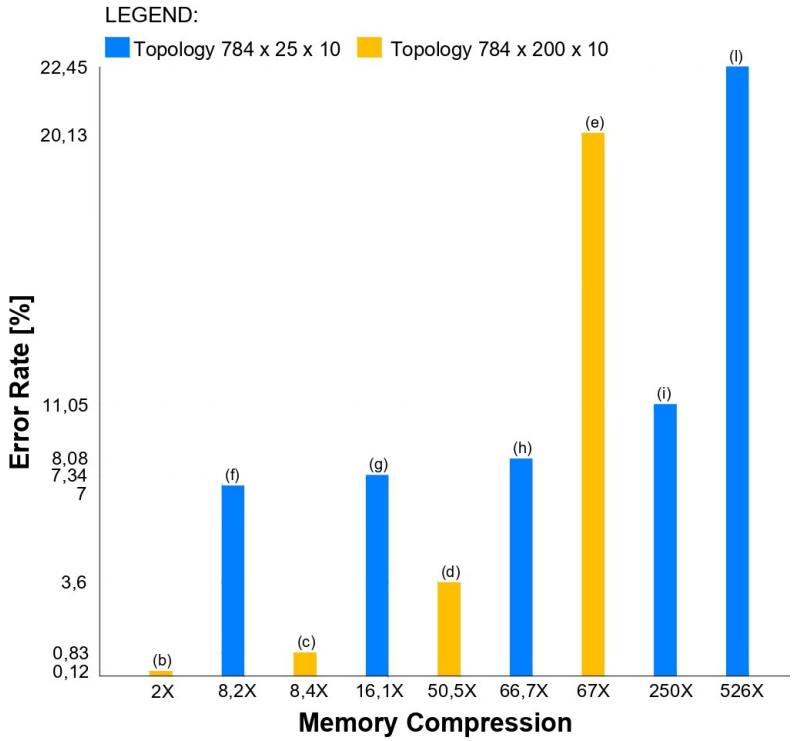

Figure 6.1: Error-rate vs Memory compression.

6.2.2 Resources utilization

To characterize and test the architecture we have designed, we have run a set of test to know the area and timing constraints our design has. We have started first by analyzing in depth the implementation of the different types of multiplier block presented in 5.3.3. After that we have characterized the implementation of *MAC-parallel* and *Neurons-layer* with different parameters, because these architectural elements are the one's that play a key role in the proposed architecture.

Multiplier block resources

As discussed in chapter 4 model compression permits not only to reduce the memory constrains but also the hardware complexity in some cases (with specific weights).

Figure 6.2 shows in a schematic way the number of used resources of each multiplier and the highest working frequency for each multiplier. The tested optimized multipliers are the same showed in figure 5.8, it means that they are implemented to satisfy the not uniform quantization with two bits for the constant multiplier and shift multiplier and not uniform quantization with one bit for the partial binary multiplier.

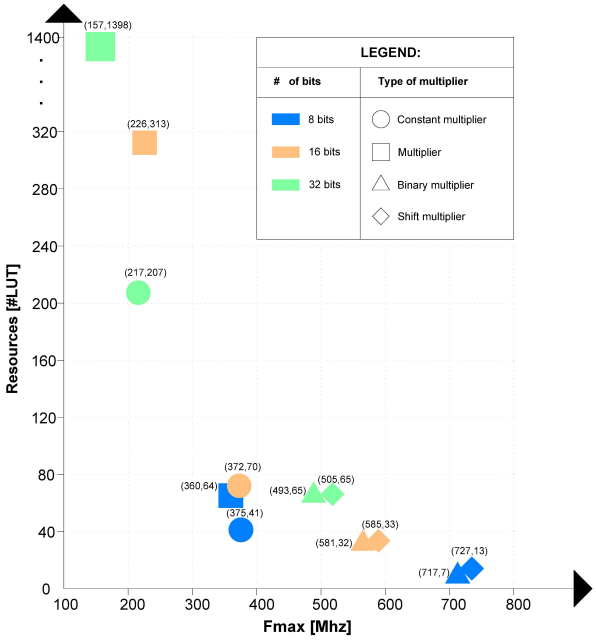

Figure 6.2: Multipliers comparison in terms of needed resources and maximum frequency.

In order to perform a correct comparison during synthesis of these components, the possibility to choose the *DSP* block from synthesizer is disabled. This because in this case we want do a comparison with the same elements.

The figure takes into account not only the different types of multipliers but also the number of input bits. An important aspect that it is important underline is that normal multiplier is the biggest one and also it is the slowest. Binary multiplier is the smallest and also the fastest but it's important to say that using one bit has an important accuracy degradation. Constant multiplier and shifter multiplier save important resources utilization respect normal multiplier and they can work with higher frequency, but with constrained weights there is a loss of accuracy like showed in 6.2.1.

MAC-parallel resources

The architecture and the parameters of this component is well described in 5.3.2.

Table 6.3 shows the results of utilization resources for the two different type of multipliers that can be used when not uniform quantization with two bits is performed. Contrary to tests done in last subsection, in this case the synthesis permits to synthesizer to choose also *DSP* if needed.

The results in the table show which are the resources needed in our neuron, if the number of bits is low, these operations can be implemented with just *LUTs* and *FFs*, but when the number of bits is higher a *DSP* will become necessary. *DSPs* are a scarce resource and it also involves a bigger area than *LUTs* and *FFs*. For these reasons the synthesis tools will try to use them only if they are strictly necessary.

Table 6.3: Results of *MAC-parallel* implem. with $T_{in} = 1$. Area vs input bits.

# BITS	*LUT*	*FF*	*DSP*	*Max-frequency* [MHz]
Normal multiplier				
32	70	87	2	191
16	6	38	1	375
8	67	42	0	370
Shift multiplier				
32	130	134	0	490
16	67	70	0	547
8	33	38	0	729

Table 6.4 shows how the utilization resources increase when the inputs concurrency processing increases with input bits of 8 bits. The table results are for the worst case (normal multiplier) and for best case (shift multiplier), constant multiplier based architecture is expected have middle properties.

In this case also with the increases of T_{in} the number of *DSP* used remain zero. It means that their usage depends only on the input bits. The architecture with normal multiplier has to use also the coder in order to decode the weights, instead for the shift multiplier coder is not needed, this aspect has an impact on resources. About max working frequency it is possible to say that increasing T_{in} factor there is a degradation of the max frequency and this is more marked with the normal multiplier respect to shift multiplier, this result can be explained in 5.3.1.

The concurrency permits to increase the throughput but also there is an increase of resources, we want compare how resources increase with the two different used multipliers:

- $Ratio_{mult.} = \frac{LUTs_{T_{in}=16}}{LUTs_{T_{in}=1}} \simeq 13.$ • $Ratio_{shift} = \frac{LUTs_{T_{in}=16}}{LUTs_{T_{in}=1}} \simeq 10.$

The results of the computed ratio show that not only the shifter multiplier has less resources respect to normal multiplier, but also that the applied input concurrency has an increment of resources that it is bigger for the normal multiplier respect to the shifter.

Table 6.4: Results of *MAC-parallel* implem. with a variable number of T_{in} and input bits = 8. Area vs T_{in}.

Multiplier block	*LUT*	*FF*	*DSP*	*Max-frequency* [MHz]
		$\mathbf{T}_{in} = 2$		
Normal multiplier	131	52	0	290
Shift multiplier	62	56	0	420
		$\mathbf{T}_{in} = 4$		
Normal multiplier	255	72	0	243
Shift multiplier	108	76	0	345
		$\mathbf{T}_{in} = 8$		
Normal multiplier	480	112	0	200
Shift multiplier	221	116	0	251
		$\mathbf{T}_{in} = 16$		
Normal multiplier	899	192	0	172
Shift multiplier	422	196	0	201

Neurons layer resources

The architecture and the parameters of this component are described in 5.5.2. At this point, we want compare the effect of output concurrency in terms of resources, this aspect it's important to understand which is the impact of output concurrency in terms of resources and do a comparison with input concurrency.

Table 6.5: Results of *Neurons layer* implem. with a variable number of T_{out} and input bits = 8. Area vs T_{out}, T_{in}. The evaluated layer is the 784×200.

Multiplier block	*LUT*	*FF*	*DSP*	*Max-frequency* [MHz]
$\mathbf{T}_{out} = 1, T_{in} = 1$				
Normal multiplier	83	103	0	397
Shift multiplier	48	101	0	410
$\mathbf{T}_{out} = 16, T_{in} = 1$				
Normal multiplier	781	940	0	397
Shift multiplier	282	822	0	422
$\mathbf{T}_{out} = 2, T_{in} = 2$				
Normal multiplier	264	180	0	287
Shift multiplier	116	94	0	397
$\mathbf{T}_{out} = 4, T_{in} = 4$				
Normal multiplier	781	328	0	243
Shift multiplier	310	356	0	325

- $Ratio_{mult.} = \dfrac{LUT_{T_{out}=16}}{LUT_{T_{out}=1}} \simeq 9.$ • $Ratio_{shift} = \dfrac{LUT_{T_{out}=16}}{LUT_{T_{out}=1}} \simeq 6.$

  In this case when T_{out} increases the number of LUT in the two different cases increases in a different way, so increase the output concurrency can be very efficient with shift multiplier respect to normal multiplier. The number of FF instead grows up in the same way about. The max frequency remains more or less constant because T_{out} has no big impact on this aspect.

- $Ratio_{mult.} = \dfrac{LUT_{T_{in}=4,T_{in}=4}}{LUT_{T_{out}=16,T_{in}=1}} \simeq 1.$ • $Ratio_{shift} = \dfrac{LUT_{T_{in}=4,T_{in}=4}}{LUT_{T_{out}=16,T_{in}=1}} \simeq 1.099.$

  In this test the product of the two tiling factors are the same, it means the loop iterations are the same (expected similar throughput) in the two different configurations. The number of the $LUTs$ is similar ($ratio \simeq 1$), but the number of FF is bigger in the first case with $T_{out} = 16, T_{in} = 1$, this suggest that the second case with $T_{out} = 4, T_{in} = 4$ is better because same loop iterations and less resources.

Neural network resources

Table 6.6 and table 6.7 shows the results of resources utilization for two different neural network topology, the same used in chapter 4. As showed in 6.2.1, the model size determines the accuracy and also the needed hardware resources.

In this type of test the possibility to use *DSP* is not allowed, instead *BRAM* will be the synthesizer that choose if is necessary to use or not.

Table 6.6 shows the results of resources utilization for the different topology of neural network with fixed tiling factors and a normal multiplier. In this first test no model compression to the weights is applied.

Table 6.6: Results of *Neural network* implem. with a variable topology, input bits = 32 and not weights compression applied.

Multiplier block	*LUT*	*FF*	*BRAM*	*Max-frequency* [MHz]
TOPOLOGY 784 X 200 X 10, $T_{out} = 2, T_{in} = 2$				
Normal multiplier	18730	10300	2	173
TOPOLOGY 784 X 25 X 10, $T_{out} = 2, T_{in} = 2$				
Normal multiplier	15432	9540	1	173

Table 6.7 shows the results when applied model compression is done in terms of number of bits from 32 to 8 and not uniform compression of the weights into two bits is performed. In this case we applied an optimization on multiplier and see the results. It is clear that the used resources are less than table 6.6 but also it is possible to see that frequency increases passing from bigger model to smaller model. An increase of frequency is also possible to see when passing from normal multiplier to shift multiplier.

Table 6.7: Results of *Neural network* implem. with a variable topology, input bits = 8 and two bits not uniform compression applied to weights.

Multiplier block	*LUT*	*FF*	*BRAM*	*Max-frequency* [MHz]
TOPOLOGY 784 X 200 X 10, $T_{out} = 2, T_{in} = 2$				
Normal multiplier	8730	4300	2	250
Shift multiplier	7840	4230	2	350
TOPOLOGY 784 X 25 X 10, $T_{out} = 2, T_{in} = 2$				
Normal multiplier	8420	4194	1	287
Shift multiplier	7694	3980	1	405

6.2.3 Power analysis

Multiplier block power analysis

The tested multipliers are the same ones showed in figure 5.8 and discussed in previous section 6.2.2. Table 6.3 shows different groups characterized by the number of input bits, we select the lowest frequency (reported by normal multiplier) for each group in order to have comparable measurement and compute the power. The power analysis showed in figure 6.3 is computed in the worst case condition with $E_{sw} = 1$ and it includes dynamic power and static power. Infact the reported power analysis is biased about $P_{static} \simeq 200mW$.

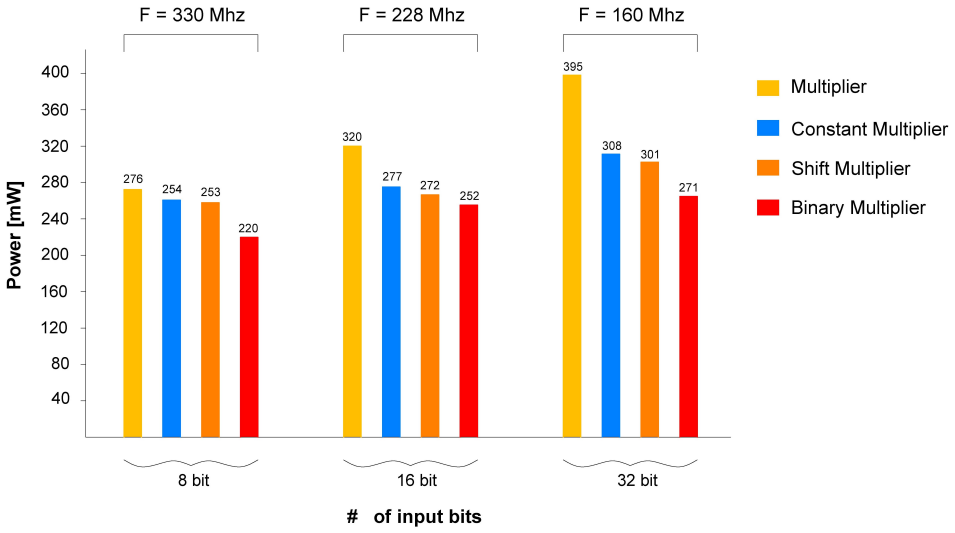

Figure 6.3: Multiplier blocks power analysis.

Neurons layer power analysis

The following study cases are take into account in table 6.8:

- **a)** $T_{out} = 1, T_{in} = 1$.
- **b)** $T_{out} = 1, T_{in} = 16$.
- **c)** $T_{out} = 16, T_{in} = 1$.
- **d)** $T_{out} = 2, T_{in} = 2$.
- **e)** $T_{out} = 4, T_{in} = 4$.

These are the same study cases in 6.2.2, so it's possible compare power consumption and resources utilization in the different configurations. The different examples of the possible configurations are explained in 5.3.1.

The case **a)** is the standard case. The case **b)** and **c)** have the same loop iterations computed with formula 5.2 (=16) but different hardware arrangement, the case **b)** shows a power consumption less than the case **c)**. Also the case **e)** has the same loop iterations but with another arrangement shows less power consumption.

This consideration is important due to the fact the same loop iterations suggests a similar throughput with a fixed frequency and different hardware configuration can have different power consumption.

Table 6.8: Power report of *Neurons layer* with different tiling factors configurations.

Power[mW]. Frequency = 250 MHz. $\mathbf{E}_{sw} = 1.$					
8 bits	*Case a)*	*Case b)*	*Case c)*	*Case d)*	*Case e)*
Normal multiplier	263	550	908	309	425
Shift multiplier	248	480	776	287	423

Neural network power

In table 6.9 it is possible to observe the power report of a complete neural network for a different number of input bits and for a fixed frequency.

The results show that power depends on the topology, in this case on the number of the *HUs*. Smaller topology has less power consumption respect the bigger ones ($\simeq 3\%$).

Lowering the number of input bits, it is possible increase the working frequency from 170 to 250 MHz. Expected dynamic power increases ($\simeq f^2$) but the static power decreases ($\simeq area$), so the final mixed result is that there is saving of power consumption passing from 32 bits to 8 bits of $\simeq 4.65\%$ and of $\simeq 18\%$ respectively for normal multiplier and for shift multiplier based architectures.

Table 6.9: Power report of different *Neural network* topology.

Power[mW], $\mathbf{T}_{in} = 2, T_{out} = 2$		
32 bits, f = 170 MHz	784 x 200 x 10	784 x 25 x 10
Normal multiplier	710	690
8 bits, f = 250 MHz	784 x 200 x 10	784x25x10
Normal multiplier	677	627
Shift multiplier	583	577

6.2.4 Throughput

The definition of the throughput for a neural network and possible optimizations of it are discussed in 3.1.1. In order to compute the throughput, the number of clock cycles needed to process the all network are computed, knowing the loop iterations for the different topology and the *FSMs* working of controller.

After that fixing the working frequency, it is possible to calculate the period. According to period and overall clock cycles of processing we can compute how much time is spent for one image processing (one inference) in seconds. The inverse of this is the throughput.

Table shows that throughput is affected by model compression because determines an increment of working frequency passing from 32 to 8 bits. Besides, throughput is influenced by model size (less or more loop iterations), finally of course also the used concurrency determines the final throughput like showed in 3.1.

Table 6.10: Throughput report of different *Neural network* topology.

Throughput [imgs/sec]		
32 bits, f = 170 MHz	784 x 200 x 10	784 x 25 x 10
$T_{in} = 1, T_{out} = 1$	357	2817
$T_{in} = 2, T_{out} = 2$	1422	10706
$T_{in} = 4, T_{out} = 4$	5652	-
8 bits, f = 250 MHz	784 x 200 x 10	784 x 25 x 10
$T_{in} = 1, T_{out} = 1$	523	4167
$T_{in} = 2, T_{out} = 2$	2127	15873
$T_{in} = 4, T_{out} = 4$	8333	-

6.2.5 Comparison between different neural network architecture

Table 6.11 takes in consideration some neural network with the same scope of ours, classify the data set *MNIST* with a *FC* neural network. The used topology is bigger, so the resources utilization and the throughput are bigger than ours, but the flexibility of our architecture permits to manage the resources as we want and as a consequence the throughput. Bigger number of *HUs* suggests that accuracy is better than our model compression proposed model (94% with a memory compression of 50.5x).

Table 6.11: Report of different neural networks architecture.

Cases	*Topology*	*LUTs*	*FFs*	*BRAM*	*DSPs*	*Throughput [imgs/sec]*
[**32**]	784x1024x10	213593	136677	750	900	70000
[**8**]	784x1024x10	91131	-	4.5	0	12361000
Proposed	784x200x10	7694	3980	1	0	2127

6.2.6 Conclusion e future works

Doing this project, I have been able to learn about different machine learning algorithms and I have focused in neural networks. I have learnt the methodology used in neural networks, which is a topic that is becoming more important every day. During this period i have to face up different problems, firstly the modeling of neural network on *MATLAB* and understand how to quantize the weights in a efficient way. In order to do it, it was very important and long the phase of the documentation. After that, i have been able to design the architecture of a neural network, thinking about the specific hardware that should be implemented. Afterwards, I was able to implement that architecture in hardware with *VHDL* and map it into a specific *FPGA*. The proposed architecture is configurable at the highest level, we have put emphasis on designing fully configurable layers that can be used to create complex neural networks in an easy and efficient way.

Possible future works are the possibility to implement the modeling of more complex neural network (more number of layers) or the possibility to apply different methodologies in order to perform a model compression in a efficient way and have less accuracy degradation. Other possibilities are apply model compression to another data set and study in deep methodologies to perform in automatic way the not linear compression. In order to make it, it will be possible using other environments or frameworks that they can be able to simplify the working flow like *Tensorflow* or *Caffe*. Another improvement at time level can be develop the same architecture using the methodologies of *HLS* that permit to describe hardware with high level language.

Appendix A

Details about *VHDL* implementation

A.1 Code of the main components

The code of the main components is reported, the adopted hierarchic is the same explained in 5.2.

A.1.1 *MAC-parallel component*

Listing A.1: MAC-parallel component

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.layer_parameters_pkg.all;
use work.bit_evaluation_pkg.all;
use IEEE.math_real.all ;


entity mac_parallel is
generic
(
 a_n_tot : integer := d_n_tot; -- total number of bits of
 a_n_frac : integer := d_n_tot/2; -- number of fractional bits of
    input A
 b_n_tot : integer := d_n_tot; -- total number bits of weight value
 b_n_frac : integer := d_n_tot/2; -- number of fractional bits of
    input B
 multiplier_type : integer := 1;   --0 = normal multiplier, --1 =
    shift_multiplier , 2 = binary_multiplier
```

```
17  tiling_factor : integer := T_in;   --parallelism of structure macs
      T_in
18  weight_codification : integer := 2 ;
19  pipelined_adder : integer := 0
20  );
21  port
22  (
23  clk : in std_logic ;
24  rstn : in std_logic ;
25  a: in MAC_array_type_gen;
26  b : in signed((tiling_factor*weight_codification)-1 downto 0);
27  enable : in std_logic ;
28  q : out signed( a_n_tot + b_n_tot -1 downto 0);
29  rstn_accumulator : in std_logic
30  );
31 end mac_parallel;
32
33 architecture Behavioral of mac_parallel is
34
35
36 constant tot_n_tot : integer := a_n_tot+b_n_tot;
37 constant increment_bits_last_stage : integer := integer(ceil(log2(
      real(tiling_factor))));
38
39 type MAC_array_type is array (0 to tiling_factor-1) of signed(
      tot_n_tot-1 downto 0);
40
41 signal output_multiplier_d1 : signed (tot_n_tot-1 downto 0);
42 signal output_multiplies : MAC_array_type;
43 signal  rstn_accumulator_d0 : std_logic;
44 signal  rstn_accumulator_d1 : std_logic;
45 signal enable_d0 : std_logic;
46 signal enable_d1 : std_logic;
47 signal a_d0 : MAC_array_type_gen;
48 signal b_d0 : signed((weight_codification*tiling_factor)-1 downto 0);
49 signal weights_values : MAC_array_type_gen;
50 signal multiply_result_concatenated : signed ((tiling_factor*
      tot_n_tot)-1 downto 0);
51 signal adder_tree_sum : signed ((tot_n_tot +
      increment_bits_last_stage) -1 downto 0);
52 signal adder_tree_sum_d1 :  signed ((tot_n_tot +
      increment_bits_last_stage) -1 downto 0);
53 signal acc_out : signed((tot_n_tot + increment_bits_last_stage) -1
      downto 0);
54 signal acc_in : signed((tot_n_tot + increment_bits_last_stage) -1
      downto 0);
55 signal sum_accumulator : signed((tot_n_tot +
      increment_bits_last_stage)-1 downto 0);
56
```

```
57
58 begin
59
60 --pipe0 FF (Control signal)
61
62 enable_d_0:  entity work.ff port map (clk => clk, rstn => rstn, D =>
       enable, Q1=> enable_d0 );
63 rstn_accumulator_d_0:  entity work.ff port map (clk => clk, rstn =>
       rstn, D => rstn_accumulator, Q1=>rstn_accumulator_d0 );
64
65 --pipe0 Register (data and word weights)
66
67 data_input_a :
68 for i in 0 to tiling_factor-1 generate
69 Register_input_data : entity work.RegisterN
70 generic map (num_bits => a_n_tot)
71 port map (enable => enable , clk => clk , rstn => rstn, D => a(i), Q1
       => a_d0(i));
72 end generate;
73
74 -- weights_input_b are the concatenation of T*2 bits represents T
       different weights
75
76 weights_input_b : entity work.RegisterN generic map ( num_bits => (
       weight_codification*tiling_factor))
77 port map (enable => enable , clk => clk , rstn => rstn, D => b, Q1 =>
       b_d0);
78
79 --Trasform the codification in real weight value
80
81 decodificator_gen : if (multiplier_type = 0) generate
82 decodificator_instance : entity work.weight_decodificator generic map
       (tiling_factor => tiling_factor,  b_n_tot => b_n_tot, b_n_frac =>
       b_n_frac , weight_codification => weight_codification)
83 port map (b => b_d0, q => weights_values);
84 end generate;
85
86 -- Align the values
87
88 --palign_values : process (weights_values, a_d0)
89 --begin
90 --for i in 0 to (tiling_factor-1) loop
91 --if(max_n_frac < max_n_tot)then
92 ---- extend sign bit
93 --a_d0_ext(i)(max_n_tot -1 downto max_n_frac) <= (others => a_d0(i)(
       a_n_tot -1));
94 --if (multiplier_type = 0) then
95 --weights_values_ext(i)(max_n_tot -1 downto max_n_frac) <= (others =>
       weights_values(i)(b_n_tot -1));
```

```
96 --end if;
97 --end if;
98 --if(max_n_frac > 0)then
99 --a_d0_ext(i)(max_n_frac -1 downto 0) <= (others => '0');
100 --if (multiplier_type = 0) then
101 --weights_values_ext(i)(max_n_frac -1 downto 0) <= (others => '0');
102 --end if;
103 --end if;
104 --a_d0_ext(i)(a_n_int+max_n_frac -1 downto max_n_frac-a_n_frac) <=
    a_d0(i);
105 --if (multiplier_type = 0) then
106 --weights_values_ext(i)(b_n_int+max_n_frac -1 downto max_n_frac-
    b_n_frac) <= weights_values(i);
107 --end if;
108 --end loop;
109 --end process;
110
111 multiplier_generator :
112 for i in 0 to tiling_factor-1 generate
113 normal_multiplier_generator :
114 if (multiplier_type = 0) generate
115 multiplier_instance : entity work.multiplier generic map (a_n_tot,
    tot_n_tot)
116 port map (a=> weights_values(i) , b => a_d0(i), output=>
    output_multiplies(i));
117 end generate;
118 shift_multiplier_generator :
119  if (multiplier_type = 1) generate
120 shift_multiplier_instance : entity work.shift_multiplier generic map
    (tot_n_tot,a_n_tot, weight_codification)
121 port map (a => a_d0(i), weight_in => b_d0((i*weight_codification)+1
    downto weight_codification*i), b => output_multiplies(i));
122 end generate;
123 binary_multiplier_generator :
124 if (multiplier_type = 2) generate
125 binary_multiplier_instance : entity work.binary_multiplier generic
    map (tot_n_tot, a_n_tot, 1)
126 port map (a => a_d0(i), weight_in => b_d0(i), b => output_multiplies(
    i));
127 end generate;
128 end generate;
129
130 process (output_multiplies)
131 variable index :  integer := tot_n_tot;
132 begin
133 for i in 0 to (tiling_factor-1) loop
134 if (tiling_factor > 1) then
135 multiply_result_concatenated((i*index)+tot_n_tot-1 downto index*i) <=
     output_multiplies(i);
```

```
136 end if;
137 end loop;
138 end process;
139
140 --Adder tree
141
142 adder_tree_generate :
143 if (tiling_factor > 1) generate
144 adder_trees_inst: entity work.adder_tree generic map (NINPUTS =>
        tiling_factor ,IWIDTH => tot_n_tot , OWIDTH => tot_n_tot +
        increment_bits_last_stage, pipelined => pipelined_adder )
145 port map (rstn  => rstn , clk => clk , d =>
        multiply_result_concatenated, q => adder_tree_sum);
146 end generate;
147
148  -- Pipe1 flip flops
     ------------------------------------------------------------------
149 enable_d_1:  entity work.ff port map (clk => clk, rstn => rstn, D =>
        enable_d0, Q1=>enable_d1 );
150 rstn_accumulator_d_1:  entity work.ff port map (clk => clk, rstn =>
        rstn, D => rstn_accumulator_d0, Q1=>rstn_accumulator_d1 );
151
152 --Pipe1 register
     ------------------------------------------------------------------
153
154 Output_multiplier_register_generation :
155 if (tiling_factor > 1) generate
156 result_adder_tree_d1 : entity work.RegisterN generic map ( num_bits
        => tot_n_tot + increment_bits_last_stage)
157 port map (enable => enable_d0 , clk => clk , rstn => rstn, D =>
        adder_tree_sum, Q1 => adder_tree_sum_d1);
158 end generate;
159
160 Output_multiplier_singular :
161 if (tiling_factor = 1) generate
162 result_adder_tree_d1 : entity work.RegisterN generic map ( num_bits
        => tot_n_tot + increment_bits_last_stage)
163 port map ( enable => enable_d0 , clk => clk , rstn => rstn, D =>
        output_multiplies(0), Q1 =>adder_tree_sum_d1 );
164 end generate;
165
166 -- Sum ==========================================
167 sum_accumulator <= adder_tree_sum_d1 + acc_out ;
168 acc_in <= sum_accumulator ((tot_n_tot + increment_bits_last_stage)-1
        downto 0);
169 --acculumator-register
170
171 accumulator_register : entity work.accumulator_registerN generic map
        ( num_bits => tot_n_tot + increment_bits_last_stage  ,  bias => 1)
```

```
172 port map (enable => enable_d1 , clk => clk , rstn =>
        rstn_accumulator_d1 , D => acc_in, Q1 => acc_out);
173
174 q <= acc_out((a_n_tot + b_n_tot)-1 downto 0) ;
175
176 end Behavioral;
177
178 }
```

A.1.2 *MAC-memory component*

Listing A.2: MAC-memory component

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  use ieee.math_real.all;
5  use work.bit_evaluation_pkg.all;
6  use work.layer_parameters_pkg.all;
7
8
9  entity mac_memory is
10 generic
11  (
12  b_n_tot : integer := d_n_tot; -- total number of bits of input B,
       this is the total parallelism of memory ,depends on T, T = 8 means
        4 weights, for mac in parallel
13  b_n_frac : integer := d_n_tot/2; -- number of fractional bits of
       input B
14  tiling_factor : integer := T_in;
15  a_n_tot : integer := d_n_tot; --data total number of bits
        --a
16  a_n_frac : integer := d_n_tot/2; --data number of fractionary bits
17  weight_codification : integer := 2;           --weight codification
18  multiplier_type : integer:= 1;
19  Wj : integer := 16; -- Nl-1\T_in , how many iteration i used in
       order to compute the output of 1 neuron, every iteration has an
       access in memory with the word to T*2 bits
20  neurons_num : integer := 4 ;--neurons assigned to this MAC = Nl
21  pipelined_adder : integer := 0;
22  memory_file_weight : string := "C:/Users/Gianmarco/Desktop/project_1
       /weight_1.txt"
23  );
24  port
25 (clk : in std_logic;
26  rstn : in std_logic;
27  -- I/O
28  input : in MAC_array_type_gen;
```

```
29  output : out signed(a_n_tot+ b_n_tot -1 downto 0);
30  weight_in_codify_word : in signed((weight_codification*tiling_factor
      ) -1 downto 0);
31  -- Addresses
32  weight_read_addr  : in unsigned(bitEvalSafe((Wj)*neurons_num)-1
      downto 0);
33  -- Control signals
34  MAC_enable : in std_logic;
35  write_output: in std_logic;
36  cs_control_memory_weight : in std_logic;
37  rstn_accumulator  : in std_logic
38  );
39 end mac_group;
40
41 architecture Behavioral of mac_group is
42
43
44  constant mac_n_tot : integer := a_n_tot+b_n_tot;
45  constant mac_n_frac : integer := a_n_frac+b_n_frac;
46
47  signal output_buffer : signed(a_n_tot+b_n_tot -1 downto 0);
48  signal MAC_output : signed(mac_n_tot -1 downto 0);
49  signal write_output_d0 : std_logic;
50  signal write_output_d1 : std_logic;
51
52  signal weight_address : unsigned(bitEvalSafe((Wj)*neurons_num)-1
      downto 0);
53  signal weight_memory_word : std_logic_vector ((weight_codification*
      tiling_factor)-1 downto 0 );
54  signal weight_memory_word_read : std_logic_vector ((
      weight_codification*tiling_factor)-1 downto 0 );
55
56 begin
57
58 iMAC: entity work.mac_parallel
59  generic map(
60  a_n_tot => a_n_tot ,
61  a_n_frac => a_n_frac ,
62  b_n_tot => b_n_tot ,
63  b_n_frac => b_n_frac,
64  multiplier_type => multiplier_type ,
65  tiling_factor => tiling_factor,
66  weight_codification => weight_codification,
67  pipelined_adder => pipelined_adder
68  )
69  port map
70  (
71  clk => clk,
72  rstn => rstn,
```

```
73  a => input,
74  b => signed(weight_memory_word_read),
75  enable => MAC_enable,
76  q => MAC_output,
77  rstn_accumulator => rstn_accumulator
78  );
79
80  -- Delay registers
      ================================================================
81  iwrite_output_d0:entity work.ff
82  port map(clk=>clk, rstn => rstn,D => write_output,Q1 =>
      write_output_d0);
83  iwrite_output_d1:entity work.ff
84  port map(clk=>clk, rstn => rstn,D => write_output_d0 , Q1 =>
      write_output_d1);
85
86  -- Register used to save the MAC result
      ===================================
87 ioutput_reg:entity work.RegisterN generic map(a_n_tot + b_n_tot)
88  port map(enable => write_output_d1, clk => clk, rstn=> rstn, D =>
      MAC_output , Q1 => output_buffer);
89
90  output <= output_buffer;
91
92  -- Weights Memory
      ================================================================
93
94 irom_mem : entity work.rom
95 generic map(DATA_WIDTH => tiling_factor*weight_codification,
      ADDR_WIDTH => bitEvalSafe((Wj)*neurons_num),memory_file_weight=>
      memory_file_weight )
96 port map (clk => clk, address => weight_address, data_out =>
      weight_memory_word_read, cs =>  cs_control_memory_weight );
97
98 end Behavioral;
99 }
```

A.1.3 *Neurons-layer component*

Listing A.3: Neurons-layer component

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 use ieee.math_real.all;
5  use work.bit_evaluation_pkg.all;
6  use work.layer_parameters_pkg.all;
7  use work.generic_array_pkg.all;
```

```
8
9  entity neurons_layer is
10 generic
11 (
12  a_n_tot : integer := d_n_tot; --data total number  of bits
13  a_n_frac : integer := d_n_tot/2; --data number of fractionary bits
14  tiling_factor : integer := T_in;
15  Wj : integer := 784; --layer weights per neuron T number  Nl-1\T_in
16  Sj : integer := 200; --layer neurons number Nl
17  Mj : integer := T_out; --layer MACs number (T) T_Out
18  weight_codification : integer := 2;
19  a_func : typestr := "Relu";         --activation function
20  multiplier_type : integer:= 0
21 memory_file_weight : weight_memory_string := ("C:/Users/Gianmarco/
       Desktop/project_1/weight_1.txt", "C:/Users/Gianmarco/Desktop/
       project_1/weight_2.txt","C:/Users/Gianmarco/Desktop/project_1/
       weight_3.txt","C:/Users/Gianmarco/Desktop/project_1/weight_4.txt")
22  );
23  port
24  (
25  clk : in std_logic;
26  rstn : in std_logic;
27  -- I/O
28  input : in MAC_array_type_gen;
29  output_array : out MAC_array_type_out;
30  weight_addr : in unsigned(bitEvalSafe(Wj)-1 downto 0);
31  iterat_addr : in unsigned(bitEvalSafe(ceil_ratio(Sj,Mj))-1 downto 0)
       ;
32  MAC_addr : in unsigned(bitEvalSafe(Mj)-1 downto 0);
33  -- Control signals
34  MAC_enable : in std_logic;
35  write_output : in std_logic;
36  last_mac : in std_logic;
37  cs_control_memory_weight : in std_logic;
38  rstn_accumulator : in std_logic
39  );
40
41 end neurons_group;
42
43 architecture Behavioral of neurons_layer is
44
45 constant SMrat : integer := ceil_ratio(Sj, Mj);    --Nl/T_out
46 constant Wtot : integer := (Wj)*SMrat;    --Nl-1\T_in * Nl\T_out
47 constant Mac_disable : integer := Mj-(Sj mod Mj); --M j - (S j mod M
       j )
48 --SIGNALS
49
50 signal write_output_d0 : std_logic;
51 signal rst_n_accumulator_d0 : std_logic;
```

```
52 signal MAC_enable_d0 : std_logic;
53 signal MAC_output_array : MAC_array_type_out;
54 signal MAC_output_array_d0 : MAC_array_type_out;
55 signal input_d0 : MAC_array_type_gen;
56 signal weight_in_d0 : signed((tiling_factor*weight_codification)-1
       downto 0);
57 signal iterat_addr_mul : unsigned(bitEvalSafe(Wtot)-1 downto 0);
58 signal weight_addr_d0 : unsigned(bitEvalSafe(Wj)-1 downto 0);
59 signal iterat_addr_mul_d0 : unsigned(bitEvalSafe(Wtot)-1 downto 0);
60 signal weight_tot_addr : unsigned(bitEvalSafe(Wtot)-1 downto 0);
61 signal weight_addr_d0_ext:unsigned(bitEvalSafe(Wtot)-1 downto 0);
62 signal write_weight_vec : std_logic_vector(Mj-1 downto 0);
63 signal write_weight_vec_d0 : signed(Mj-1 downto 0);
64 signal output_single : MAC_array_type_gen;
65
66 signal weight_ext : unsigned (bitEvalSafe(Wtot)-1 downto bitEvalSafe(
       Wj));
67
68 signal rstn_accumulator_d0 : std_logic;
69 signal last_mac_d0 : std_logic;
70 signal mac_block : std_logic_vector(Mj-1 downto 0);
71 signal mac_enable_ext : std_logic_vector (Mj-1 downto 0);
72 signal mac_gating : std_logic_vector (Mj-1 downto 0);
73 signal mac_enable_general: std_logic_vector (Mj-1 downto 0);
74
75 begin
76
77 mac_enable_ext <= (others=> MAC_enable_d0);
78
79 mac_gating_signals_generation : if (Sj mod Mj /= 0) generate
80 mac_block(Mj-1 downto Mac_disable) <= (others=>'1'); --mac actived
81 mac_block(Mac_disable-1 downto 0) <= (others=>'0'); --mac_disattived
82 mac_gating <= mac_enable_ext and mac_block;
83 with last_mac_d0 select mac_enable_general <= mac_enable_ext when
       '0',
84                                                 mac_gating when others;
85 end generate;
86
87 mac_enable_generation : if (Sj mod Mj = 0) generate
88 mac_enable_general <= mac_enable_ext;
89 end generate;
90
91
92 input_register_gen :
93 for i in 0 to tiling_factor-1 generate
94 iinput_d0:entity work.RegisterN generic map(num_bits => a_n_tot)
95 port map(enable => MAC_enable, clk=>clk, rstn => rstn , D => input(i)
       , Q1 => input_d0(i));
96 end generate;
```

```
97
98  iweight_in_d0:entity work.RegisterN generic map(num_bits => (
        tiling_factor*weight_codification))
99  port map(enable => write_weight ,clk=>clk, rstn => rstn,D =>
        weight_in,Q1 => weight_in_d0);
100
101 iMAC_last_mac_d0:entity work.ff
102 port map(clk=>clk ,rstn => rstn , D => last_mac ,Q1=>last_mac_d0);
103
104 iMAC_rstn_accum_d0:entity work.ff
105 port map(clk=>clk ,rstn => rstn , D => rstn_accumulator ,Q1=>
        rstn_accumulator_d0);
106
107 iMAC_enable_d0:entity work.ff
108 port map(clk=>clk ,rstn => rstn , D => MAC_enable,Q1=>MAC_enable_d0);
109
110 iMAC_write_output_d0: entity work.ff
111 port map(clk => clk , rstn => rstn ,D => write_output ,Q1=>
        write_output_d0 );
112
113 -- COMPUTE the address of weight_memory
114
115 iterat_addr_mul <= to_unsigned((Wj)*to_integer(iterat_addr),
        bitEvalSafe((Wj)*SMrat));
116
117 iweight_addr_d0:entity work.RegisterN_unsigned generic map(
        bitEvalSafe(Wj))
118  port map(enable => '1', clk=>clk, rstn => rstn, D => weight_addr, Q1
       => weight_addr_d0);
119
120 iiterat_addr_mul_d0: entity work.RegisterN_unsigned generic map(
        bitEvalSafe(Wtot))
121 port map(enable => '1', clk=>clk, rstn => rstn, D => iterat_addr_mul,
       Q1 => iterat_addr_mul_d0);
122
123 weight_ext <= (others => '0');
124 weight_addr_d0_ext <= weight_ext & weight_addr_d0;
125
126 weight_tot_addr <= weight_addr_d0_ext + iterat_addr_mul_d0;
127
128 gmac_groups:
129  for i in 0 to Mj-1 generate
130
131  imac_group: entity work.mac_group
132  generic map(
133  tiling_factor => tiling_factor,
134  a_n_tot => a_n_tot,
135  a_n_frac => a_n_frac ,
136  weight_codification =>  weight_codification,
```

```
137  Wj => Wj,    -- Nl-1\tiling_factor, how many iterationS are used in
       order to compute the output of 1 neuron, every iteration has an
       access in memory with the word to T*2 bits
138  neurons_num => SMrat,     --neurons assigned to this MAC = Nl/
139  memory_file_weight => memory_file_weight(i),
140  multiplier_type => multiplier_type
141  )
142  port map(
143  clk => clk,
144  rstn => rstn,
145  -- I/O
146  input => input_d0,
147  output =>  MAC_output_array(i),
148  weight_in_codify_word => weight_in_d0  ,
149  -- Addresses
150  weight_read_addr  =>  unsigned(weight_tot_addr),
151  -- Control signals
152  MAC_enable => mac_enable_ext(i),
153  write_output => write_output_d0,
154  cs_control_memory_weight =>  mac_enable_general(i),
155  rstn_accumulator => rstn_accumulator_d0
156  );
157  end generate;
158
159  gmac_group_output:
160  for i in 0 to Mj-1 generate
161  imac_group_output_register: entity work.RegisterN
162  generic map(num_bits => 2*a_n_tot)
163  port map (enable => '1', clk=>clk, rstn => rstn, D =>
       MAC_output_array(i), Q1 => MAC_output_array_d0(i));
164  end generate;
165
166  -- Activation function
       ================================================
167  activation_output:
168  for i in 0 to Mj-1 generate
169  iactivation:entity work.activation
170  generic map(2*a_n_tot ,a_n_tot, a_func)
171  port map( x=>MAC_output_array_d0(i) , y=>output_array(i));
172 end generate;
173
174
175 end Behavioral;
```

A.1.4 *buffer component*

Listing A.4: buffer component

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.layer_parameters_pkg.all;
use IEEE.NUMERIC_STD.ALL;
use work.bit_evaluation_pkg.all;

entity buffer_adaption is
generic
(buffer_size : integer := 5;
a_n_tot : integer := d_n_tot*2
);
port(
clk : in std_logic;
rstn : in std_logic;
data_in : in MAC_array_type_out;
data_out : out MAC_array_type_out;
enable_buffer_load : in std_logic
);
end buffer_adaption;

architecture Behavioral of buffer_adaption is

type buffer_array_type_gen is array (0 to buffer_size) of signed(
    a_n_tot -1 downto 0);
type buffer_array_type_complete is array (0 to T_out-1) of
    buffer_array_type_gen;
signal data_out_tmp : buffer_array_type_complete;
signal data_in_tmp : buffer_array_type_complete;
begin

buffer_adaption_gen :
for i in 0 to T_out-1 generate
data_in_tmp(i)(0) <= data_in(i);
buffer_size_gen_T :
for j in 0 to buffer_size-1 generate
horizontal_register_parallel_0 : entity work.RegisterN generic map (
    num_bits => a_n_tot)
port map (enable => enable_buffer_load, clk => clk , rstn => rstn , D
     => data_in_tmp(i)(j) , Q1 => data_out_tmp(i)(j+1) );
data_in_tmp(i)(j+1) <= data_out_tmp(i)(j+1);
end generate;
data_out(i) <= data_out_tmp(i)(buffer_size);
end generate;

end Behavioral;
```

A.1.5 *ROMs*

Listing A.5: ROM component

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.NUMERIC_STD.ALL;
5  use IEEE.std_logic_textio.ALL;
6  use work.layer_record_pkg.all;
7  use std.textio.all;
8
9  entity rom is
10 generic (
11         DATA_WIDTH :integer := 8;
12         ADDR_WIDTH :integer := 8;
13         memory_file_weight : string := "C:/Users/Gianmarco/Desktop/
       project_1/weight_1.txt"
14     );
15     port (
16         clk        : in std_logic;
17         address : in unsigned (ADDR_WIDTH-1 downto 0);
18         data_out : out std_logic_vector (DATA_WIDTH-1 downto 0);
19         cs      : in std_logic
20     );
21 end rom;
22
23 architecture Behavioral of rom is
24
25 constant ROM_DEPTH :integer := 2**ADDR_WIDTH;
26
27 type rom_type is array (0 to ROM_DEPTH-1) of std_logic_vector (
       DATA_WIDTH-1 downto 0);
28 impure function weight_rom(memory_file_weight : in string) return
       rom_type is
29     file rf : text open read_mode is memory_file_weight;
30     variable v_l : line;
31     variable v_i : std_logic_vector(DATA_WIDTH-1 downto 0);
32     variable v_rom  : rom_type;
33 begin
34     for i in 0 to ROM_DEPTH-1 loop
35         readline (rf, v_l);
36         read (v_l, v_i);
37         v_rom(i) :=  std_logic_vector(v_i);
38     end loop;
39     return v_rom;
40 end function;
41
42 constant data_rom : rom_type := weight_rom(memory_file_weight);
```

```
43
44 begin
45
46 process (clk)
47 begin
48
49             if (cs = '1') then
50               data_out <= data_rom(to_integer(address));
51               else
52               data_out <= (others=> '0');
53               end if;
54 end process;
55
56 end Behavioral;
```

A.1.6 *layer component*

Listing A.6: layer component

```
1
2 library IEEE;
3 use IEEE.STD_LOGIC_1164.ALL;
4 use IEEE.NUMERIC_STD.ALL;
5 use work.bit_evaluation_pkg.all;
6 use work.layer_parameters_pkg.all;
7
8 entity layer is
9 generic
10  (
11  a_n_tot : integer := d_n_tot; --data total number of bits
12  a_n_frac : integer := d_n_tot/2; --data number of fractionary bits
13  L_info : layer_record := (layer => "fulc", neurons_previous_layer =>
        784, neurons => 200, neurons_next_layer => 36 , a_func => "posl",
        index => 1);
14  weight_codification : integer := 2
15  );
16  port(
17  clk : in std_logic;
18  rstn : in std_logic;
19  --I/O
20  input : in MAC_array_type_gen;
21  output : out MAC_array_type_out;
22  weight_in : in signed((T_in*weight_codification)-1 downto 0);
23  --ADDRESS
24  write_addr_next_layer: in unsigned(bitEvalSafe((L_info.
      neurons_previous_layer)/T_in)-1 downto 0);
25
26  --Control IN
```

```
27    enable_read_memory : in std_logic;
28    we_write_memory :in     std_logic;
29    enable_iteration_address : in std_logic;
30    enable_mac_address :       in std_logic;
31    MAC_enable : in std_logic;
32    write_output : in std_logic;
33    enable_buffer_load : in std_logic;
34    last_mac : in std_logic;
35    cs_control_memory_weight : in std_logic;
36    rstn_accumulator : in std_logic;
37
38    --Control out
39    tc_read_memory : out std_logic;
40    tc_iteration : out std_logic;
41    tc_mac_address : out std_logic
42   );
43  end layer;
44
45  architecture Behavioral of layer is
46
47  constant tiling_factor : integer := T_in;
48  constant Mj : integer := T_out; --T_out, memory_number_layer
49  --size of small ram
50
51  constant Wj : integer := (L_info.neurons_previous_layer)/
        tiling_factor;  --Nl-1/T_in
52  constant Sj : integer := L_info.neurons;     -- Nl
53
54  --data to write into memory coming from the previous layer
55
56  signal data_to_memory : MAC_array_type_gen;
57  signal read_address_memory_input : unsigned(bitEvalSafe(Wj)-1 downto
        0);
58  signal iteration_address : unsigned (bitEvalSafe(ceil_ratio(Sj,Mj))-1
        downto 0);
59  signal data_from_memory : MAC_array_type_gen;
60  signal output_neurons_group : MAC_array_type_out;
61  signal mac_address : unsigned(bitEvalSafe(Mj)-1 downto 0);
62
63  begin
64
65  --counter (Nl-1/T) used to read the input memories and the weight
        address
66
67  counter_input_memory_weight_address : entity work.counter generic map
        (num_max => Wj, incr => 1)
68  port map (clk => clk , rstn => rstn, enbl => enable_read_memory , tc
        => tc_read_memory, q => read_address_memory_input);
69
```

```
70 --counter Nl/T
71
72 counter_iteration_address : entity work.counter generic map (num_max
       => ceil_ratio(Sj,Mj), incr => 1)
73 port map (clk => clk , rstn => rstn, enbl => enable_iteration_address
       , tc => tc_iteration, q => iteration_address);
74
75 data_to_memory <= input;
76
77 input_memories_gen :
78 for i in 0 to tiling_factor-1 generate
79 input_memory : entity work.ram_sincr_dp
80 generic map  (DATA_WIDTH => a_n_tot, ADDR_WIDTH => bitEvalSafe(Wj))
81 port map (clk=>clk, address_read => std_logic_vector(
       read_address_memory_input), data_read=>data_from_memory(i),
82 cs=>cs, address_write => std_logic_vector(write_addr_next_layer) ,
       data_write =>  data_to_memory(i) ,
83   we_write => we_write_memory );
84   end generate;
85
86 neurons_instance : entity work.neurons_layer
87 generic map (
88  a_n_tot => a_n_tot,
89  a_n_frac => a_n_frac,
90  Wj => Wj,
91  Sj => Sj,
92  Mj => Mj,
93  a_func => L_info.a_func
94  )
95  port map
96  (clk => clk,
97  rstn => rstn,
98  -- I/O
99  input => data_from_memory,
100  output_array => output_neurons_group,
101  -- Addresses
102  weight_addr => read_address_memory_input,
103  iterat_addr => iteration_address,
104  -- Control signals
105  MAC_enable => MAC_enable,
106  write_output => write_output,
107  last_mac => last_mac,
108  cs_control_memory_weight => cs_control_memory_weight,
109  rstn_accumulator => rstn_accumulator
110  );
111
112 buffer_instance : entity work.buffer_adaption
113 generic map (buffer_size => L_info.neurons/T_out, a_n_tot =>2*a_n_tot
       )
```

```
114 port map (clk => clk,
115 rstn => rstn,
116 data_in => output_neurons_group,
117 data_out => output,
118 enable_buffer_load => enable_buffer_load
119 );
120 end Behavioral;
```

A.2 Package

A.2.1 Layer-parameters-pkg

Listing A.7: layer component

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.generic_array_pkg.all;

package layer_parameters_pkg is

 type typestr is array (1 to 4) of character ;

 constant T : integer := 1;
 constant T_in : integer := T;
 constant T_out : integer := T;
 constant d_n_tot : integer := 8;

 type layer_record is record
 layer : typestr ; -- fulc, maxV
 neurons_previous_layer : integer;    --Nl-1--
 neurons : integer ;                  --Nl--
 neurons_next_layer : integer;        --Nl+1--
 a_func : typestr ;                   --RElu
 index : integer;
 end record layer_record ;

 type layer_array is array ( natural range <>) of layer_record ;
 type MAC_array_type_gen is array (0 to T_in-1) of signed(d_n_tot -1
    downto 0);
 type MAC_array_type_out is array (0 to T_out-1) of signed((2*d_n_tot
    ) -1 downto 0);
 type weight_memory_string is array (0 to T_in-1)of string(49 downto
    1);
 type real_array is array (0 to 3) of real;

constant L_info : layer_record := (layer => "fulc",
    neurons_previous_layer =>784,neurons => 200 , neurons_next_layer
    => 30 , a_func => "Relu", index => 1);

 end package layer_parameters_pkg ;
```

Bibliography

[1] V. Sze, Y. Chen, T. Yang, and J. S. Emer. «Efficient Processing of Deep Neural Networks: A Tutorial and Survey». In: *Proceedings of the IEEE* 105.12 (Dec. 2017), pp. 2295–2329. ISSN: 1558-2256. DOI: 10.1109/JPROC.2017.2761740 (cit. on pp. 5, 7, 15, 16, 24, 26, 32, 37, 38, 44, 57, 61, 83).

[2] Andrew Ng. «Machine Learning, Stanford University. Course available in Coursera platform». In: 2016 (cit. on pp. 5, 8, 15, 40, 47, 51).

[3] B. Glackin, J. Harkin, T. M. McGinnity, L. P. Maguire, and Q. Wu. «Emulating Spiking Neural Networks for edge detection on FPGA hardware». In: *2009 International Conference on Field Programmable Logic and Applications*. 2009, pp. 670–673 (cit. on p. 7).

[4] A. Karpathy F.-F. Li and CS Class CS231n: Convolutional Neural Networks for Visual Recognition. J. Johnson. «http://cs231n.stanford.edu/». In: 2017 (cit. on pp. 11, 14, 19, 22, 26).

[5] https://towardsdatascience.com. «activation-functions-neural-networks». In: (cit. on p. 13).

[6] Y. Le Cun, I. Guyon, L. D. Jackel, D. Henderson, B. Boser, R. E. Howard, J. S. Denker, W. Hubbard, and H. P. Graf. «Handwritten Digit Recognition: Applications of Neural Network Chips and Automatic Learning». English (US). In: *IEEE Communications Magazine* 27.11 (Nov. 1989), pp. 41–46. ISSN: 0163-6804. DOI: 10.1109/35.41400 (cit. on p. 16).

[7] https://docs.google.com/presentation/d/1fbTKtp9xOlCL4JtpiLF39x7Vxq2Z-jgn77CTqcBrwEE/editslide=id.g4965ef18a1$_{01}$5. «Open-Source Frameworks for Deep Learning: an Overview». In: 2017 (cit. on p. 16).

[8] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Heng Wai Leong, Magnus Jahre, and Kees A. Vissers. «FINN: A Framework for Fast, Scalable Binarized Neural Network Inference». In: *CoRR* abs/1612.07119 (2016). arXiv: 1612.07119. URL: http://arxiv.org/abs/1612.07119 (cit. on pp. 16, 17, 38, 112).

[9] http://cs231n.github.io/convolutional-networks/. «CS231n: Convolutional Neural Networks for Visual Recognition». In: (cit. on pp. 16, 20).

[10] https://stfalcon.com/en/blog/post/5-fascinating-applications-of-deep-learning. «5 Fascinating Applications of Deep Learning». In: 2017 (cit. on p. 16).

[11] Johan Hastad. «On the correlation of parity and small-depth circuits». In: Mar. 2014 (cit. on p. 18).

[12] Song Han, Huizi Mao, and William Dally. «Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding». In: Oct. 2016 (cit. on pp. 19, 28, 38, 44, 54, 57, 60).

[13] Sepp Hochreiter Jurgen Schmidhuber. «LONG SHORT-TERM MEMORY, Technische Universitat München 80290 Munchen, Germany». In: 1997 (cit. on p. 19).

[14] Università di bologna Machine learning. «Deep learning». In: (cit. on pp. 21, 23).

[15] https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. «comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53». In: 2018 (cit. on p. 23).

[16] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. «[DL] A Survey of FPGA-Based Neural Network Inference Accelerators». In: *ACM Trans. Reconfigurable Technol. Syst.* 12.1 (Mar. 2019). ISSN: 1936-7406. DOI: 10.1145/3289185. URL: https://doi.org/10.1145/3289185 (cit. on pp. 27, 28, 30, 34, 35, 38, 44).

[17] Xiaowei Xu, Yukun Ding, Sharon Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. «Scaling for edge inference of deep neural networks». In: *Nature Electronics* 1 (Apr. 2018). DOI: 10.1038/s41928-018-0059-3 (cit. on pp. 29, 38).

[18] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. «Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks». In: *SIGARCH Comput. Archit. News* 44.3 (June 2016), pp. 367–379. ISSN: 0163-5964. DOI: 10.1145/3007787.3001177. URL: https://doi.org/10.1145/3007787.3001177 (cit. on pp. 29, 36, 37).

[19] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. «An Analysis of Deep Neural Network Models for Practical Applications». In: (May 2016) (cit. on p. 30).

[20] http://www.cellstrat.com/2017/12/20/deep-learning-application-of-ai/. «deep-learning-application». In: 2017 (cit. on p. 31).

[21] Amos R. Omondi and Jagath C. Rajapakse. *FPGA Implementations of Neural Networks*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387284850 (cit. on p. 33).

[22] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. «Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks». In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 161–170. ISBN: 9781450333153. DOI: 10.1145/2684746.2689060. URL: https://doi.org/10.1145/2684746.2689060 (cit. on p. 33).

[23] T. Guan, P. Liu, X. Zeng, M. Kim, and M. Seok. «Recursive Binary Neural Network Training Model for Efficient Usage of On-Chip Memory». In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.7 (July 2019), pp. 2593–2605. ISSN: 1558-0806. DOI: 10.1109/TCSI.2019.2895216 (cit. on pp. 38, 42–44, 58, 59).

[24] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. «XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks». In: vol. 9908. Oct. 2016, pp. 525–542. ISBN: 978-3-319-46492-3. DOI: 10.1007/978-3-319-46493-0_32 (cit. on pp. 38, 42, 58).

[25] Matthieu Courbariaux and Yoshua Bengio. «BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1». In: *CoRR* abs/1602.02830 (2016). arXiv: 1602.02830. URL: http://arxiv.org/abs/1602.02830 (cit. on pp. 38, 58, 59).

[26] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. «BinaryConnect: Training Deep Neural Networks with binary weights during propagations». In: *CoRR* abs/1511.00363 (2015). arXiv: 1511.00363. URL: http://arxiv.org/abs/1511.00363 (cit. on p. 42).

[27] yann lecun. «http://yann.lecun.com/exdb/mnist/». In: (cit. on pp. 45, 48).

[28] Maths-works. «https://it.mathworks.com/products/deep-learning.html». In: (cit. on pp. 45, 47, 48).

[29] Mark Horowitz. «1.1 Computing's energy problem (and what we can do about it)». In: vol. 57. Feb. 2014, pp. 10–14. ISBN: 978-1-4799-0920-9. DOI: 10.1109/ISSCC.2014.6757323 (cit. on p. 61).

[30] Vivado Design Suite. Xilinx. «https://www.xilinx.com/products/design-tools/vivado.html». In: (cit. on p. 100).

[31] Drew Compston. «Two-Complement-Binary-Strings». In: 2020 (cit. on p. 101).

[32] Jinhwan Park and Wonyong Sung. «FPGA based implementation of deep neural networks using on-chip memory only». In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2016), pp. 1011–1015 (cit. on p. 112).