Hardware-Accelerated Neural Network

Luis Expósito Piñol

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya

A thesis submitted for the degree of

Master in Innovation and Research in Informatics (MIRI-HPC)

21st of April, 2015

Advisors:
Ramon Canal, Professor at UPC-DAC
Josep-Llorenç Cruz, Professor at UPC-DAC

To my family and friends.

Acknowledgements

I would like to express my gratitude to my supervisor Ramon Canal for the useful comments, remarks, and flexible accommodation while overseeing this work. Furthermore, thanks to Josep-Lloren Cruz and the Department of Computer Architecture of the Polytechnic University of Catalonia for lending me the material to complete this work and supporting me.

I would also like to thank the Department of Fundamental Physics in the University of Barcelona where I was working during this master's programme, and which allowed me the flexibility to successfully participate in it.

And finally, of course, I would like to thank my family and friends for the support.

Abstract

This work presents a design of a neural network on an FPGA, including an optimization technique which offers great benefits for hardware implementations. Between different classes of neural network, the feed forward network has been chosen, being trained previously in the CPU using backpropagation.

We have analyzed 2 multiplier implementations, 2 sigmoid function implementations and a parameterizable number of neuron on the FPGA. In all cases, we report delay, power and area.

The parametrization of the design offers the flexibility to adjust the design to the resources available and we tested different network configurations. The architecture has been designed to run the MNIST benchmark, which is a well-known test for neural networks to recognize handwritten digit characters. In early stages of the design, we used the simulator ModelSim, but also validated the proper behavior in an FPGA, the model used is a Cyclone IV EP4CE115F29C7N on a DE2-115 development board from Altera.

This work has been selected to participate in the 3rd edition of Altera's Innovate Europe Design Contest. Final winners will be announced in July 2016.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and objectives

Artificial Neural Networks (ANNs) provide an alternative method for solving a variety of problems in different fields of science and engineering, like pattern recognition, image processing and medical diagnostic. The biologically inspired ANNs are parallel and distributed information processing systems. The high speed operation in real time applications can be achieved only if the network is implemented using parallel hardware architectures.

The parallelization of this kind of structures is possible as shown in [16] for the ImageNet benchmark. This network is a convolutional neural network (CNN) Fig.1.1. The network inputs are 150,528-dimensional, and the number of neurons in the networks layers is 253,440 – 186,624 - 64,896 - 64,896 – 43,264 - 4096 - 4096 – 1000. This shows how large and complex may neural network be, so large that it cannot be implemented on an FPGA hence optimization is an important matter.

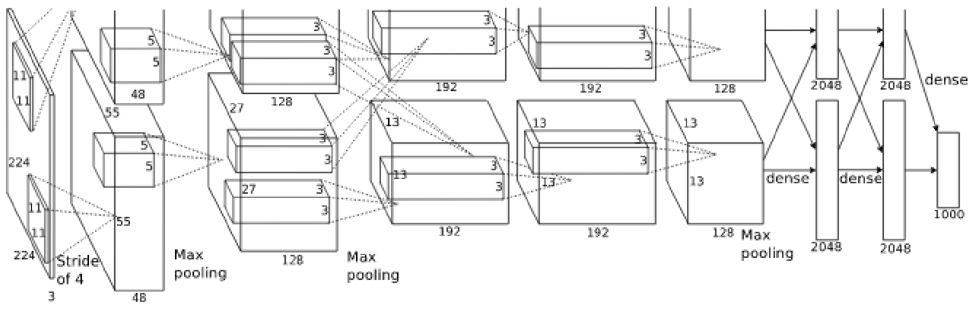


Figure 1.1: Convolutional Neural Network

But not all ANN have large dimensions, for instance, there are studies of the usage of neural for branch predictors, which speed and simplicity is the key. The multiple applications they have make them worth to research more deeply.

Implementation of ANNs falls into two categories: Software implementation and hardware implementation. Software versions are trained and simulated on general-purpose computers or gpus. They offer flexibility. However, hardware versions result really interesting due to the advantage of ANN's inherent parallelism. Hardware implementations provide high speed in real time applications, normally lacking the flexibility for structural modification and are prohibitively costly. Due to the inherit parallelism structure of the ANNs, one of the restrictions usually encountered are the resource limitation, in FPGA, the logical elements or memory. Nowadays, there are FPGA with more than 1 million of logic elements (Stratix V, Xilinx's Kintex-7). Even with these, big neural networks for image processing are not be able to show its maximum potential.

1.1 Objectives

The main objectives of this work are the following:

- Design of a flexible Feedforward Neural Network on an FPGA, a parametrized one capable of increase its performance and resource usage changing the configuration.

- Implementation of a technique for optimization: binary connect.
- Design space exploration. Two versions are compared: Precise version with floating point operations from Altera IP and 32bits floating point for weights; BinaryConnect version with fixed point operations from a VHDL library and using 1bit for weights. Both versions have 3 layers (one input layer, one hidden layer and one output layer) and the number of hardware neural units per layer tested are 5–1, 10–1, 20–2, 50-5 where the pair of numbers are the number of hardware neural units in the hidden layer and output layer, respectively. (Input layer does not count because each neuron in this layer is each pixel in the input image, it is not seen as a layer with neurons like the other two)

The main objective of this work is to implement a hardware accelerated neural network and evaluate features like scalability, performance, consumption and area, among different platforms where one can run a Neural Network (CPU and FPGA). For this purpose, it will be implemented as a parametrized feed forward neural network and tested in an FPGA. For validating bigger configurations that do not fit on the board, it will be simulated on ModelSim. This network will resolve a real problem, such as character recognition from a well-known benchmark like MNIST.

As the neural network might take a large number of logic elements, it will be tested in hardware considering the weight optimization from [6][19] in order to find out how much benefit one can make from it.

Frequently, one can find several works about neural networks solving benchmarks like MNIST, CIFAR,... where the more important metric is the accuracy, which is telling you if the neural network took a good decision. Other metrics like time or power seems to be more unnoticed, but imagine that you have to use the design in a mobile and that takes a lot of time to compute. Thus, in this work, we considered to use MNIST benchmark because the size and format of the image is well-suited for the dimensions and type of the neural network implemented.

Chapter 2

Artificial Neural Networks

2.1 Brief History

ANNs (Artificial Neural Network) have a longer history than one may think and they have gone through a lot of ups and downs, as shown in Fig. 2.1

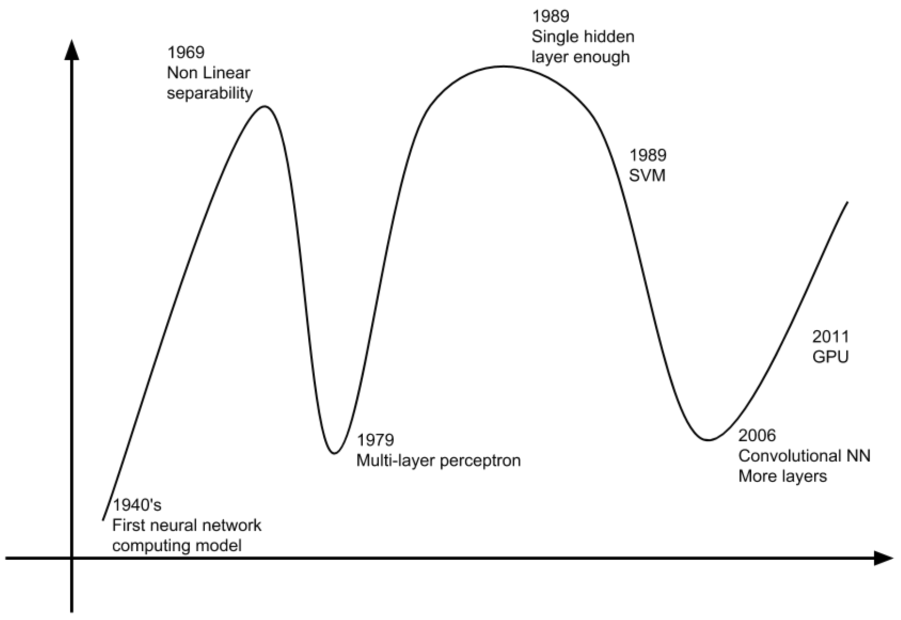


Figure 2.1: Evolution of Artificial Neural Networks

The earliest work in neural computing goes back to the 1940's when McCulloch and Pitts introduced the first neural network computing model[21]. Unfortunately, the technology available at that time did not allow them to do too much. These models made several assumptions about how neurons worked. These are ideas still fundamental in how ANNs operate nowadays, like the existence of a threshold (inside of the neuron) and once the threshold is reached the neuron fires, or also the idea of weights. The first learning rule was developed in 1949 by Hebb who wrote a book entitled "The Organization of Behavior"[12]. Hebb proposed a specific learning law, based on the mechanism of neural plasticity, incrementally modifies connection weights by examining whether two connected nodes are simultaneously on or off. Rosenblatts (1958) "Perceptron"[27] neural model and the associated learning rule are based on gradient descent, increasing or decreasing a weight depending on the satisfactoriness of a neuron's behavior. In 1960, Widrow and Hoff developed a different type of neural network processing element called ADALINE[29], which was equipped also with a learning rule based on gradient descent.

Then, it came a period of frustration. In 1969 Minsky and Papert wrote a book called "Perceptrons"[23]. They show that there are certain simple pattern recognition tasks that individual perceptrons cannot accomplish, they cannot represent non-linearly separable target functions. They left the impression that neural network research had been proven to be a dead end.

But when everybody thought it was a dead topic, a re-emergence came, during the late 1970s and early 1980s. One of the causes was the development of backpropagation by Werbos in 1974[28], however several years passed before this approach was popularized. Back-propagation nets are probably the most well known and widely applied of the neural networks today. Another was the creation of the predecessor of a famous network today(convolutional networks), the neocognitron, a stepwise trained multilayered neural network for interpretation of handwritten characters was introduced in a 1980 paper by Fukushima Kunihiko[10]. And finally, in 1989, Cybenko[8] has showed that given enough hidden neurons in an ANN with one hidden layer, the network can approximate any continuous function. Thanks to the hidden layer, the network can take non-linear decisions, and solve problems such as the canonical XOR.

And later, like a roller coaster, ANNs fell again. Vector machines and other much simpler methods such as linear classifiers gradually overtook neural networks in machine learning popularity.

But the advent of deep learning in the late 2000s sparked renewed interest in neural nets. Until this point, we had one input layer, one hidden layer and one output layer, in order to decide on more complex decisions. In 2006 several publications described more efficient ways to train neural networks with more layers and with the rise of efficient GPU computing, it has become possible to train larger networks. In 2011 they were refined and implemented on a GPU with impressive performance results.

Arriving to the present, in 2015 a team of researchers from UC Santa Barbara and Stony Brook University has now used memristors (an electronic component whose resistance changes depending on the current applied) to build a 12 x 12 memristive crossbar array, which implements a single layer perceptron.

And finally, there are also important commercial efforts by companies like IBM, Microsoft, MIT, Google, University of Manchester, ... In [1] IBM built a new chip with brain-inspired non-von Neumann computer architecture has one million neurons and 256 million synapses. Microsoft also is doing efforts to run convolutional neural networks on FPGA [22]. MIT is working on energy-friendly chip which can perform powerful artificial-intelligence tasks and could enable mobile devices to implement ”neural networks” modeled on the human brain [24].

2.2 Related Work

In earlier stages, feed forward designs were heavy in terms of operations in different phases of the execution, multiple studies address this problem. Next, we describe the closest related work and the main advances in the field.

A common problem is the activation function (normally sigmoid function, which performs a division and an exponential, increase the resource consumption and time execution). In [13] and [9] the authors try to solve this problem. In the first, using a LUT (Look-Up-Table) to approximate activation function and in the second, using a nonlinear function. In both works, the authors have implemented a neural network using only one layer reused to simulate a complete network, a drawback is the impossibility of pipelined execution because of reusing layers. [11] and [14], both use the same approximation, but the second perform a parallelism study varying the number of neurons and precision and testing it on different Virtex FPGAs. It is noteworthy because they use the same benchmark that this work, MNIST, and trained previously in MATLAB.In paper [2], they implemented an activation function using a nonlinear function in order to approximate. A curious thing it is that instead of floating point elements it uses integers. Unfortunately, it lacks of any performance test. In [20], two different implementations are investigated: a high level solution to create a neural network on a soft processor design NIOS II and a low level solution. The interesting thing is the features they measured, logic elements and the embedded multiplier consumption.

In each neuron, there is a multiplier-accumulator(MAC) unit, which can be a bottleneck on the system. [18] presents a Multiple-input-single-output Neural Network using floating-point arithmetic on FPGA. The proposed algorithm focuses on optimizing pipeline delays by splitting the Multiply and accumulate algorithm into separate steps using partial products. The performance of the proposed architecture is presented using as target a Cyclone II FPGA Device.

Until here, all works tried to reduce the complexity of the designs decreasing the float precision, using fixed floating point or reusing layers. These recent papers [19] [6] address the problem of high consumption in terms of resources. An issue of neural networks is the quantity of floating point multiplications performed and the cost that this entails. Therefore, they design a feed forward neural network without multiplications. These are software solutions but, this opens great opportunities for hardware implementations, which will be used in this work.

[15] designs a hardware NN that uses ternary weights (1,0,-1) during forward propagations. They train a neural network with high-precision. After training, they ternarize the weights. With the 0 weight, they can ”disconnect” a neuron, but for that, extra logic is needed. In our design, only an XOR operation is needed instead of a multiplication.

In our work, we implement two versions of a neural network, one being precise using floating point operators from Altera IP and the other one with a smaller resource budget using fixed point operations from [4] and this technique [6], binary weights(1 bits) and using a xor instead of a multiplier operation. Layers are explicitly defined instead of multiplexing the layers ([13] and [9]), so we can pipeline the execution. As activation function we use an approximation function from [6] instead of a LUT ([13] and [9]), so we reduce the memory usage.

2.3 Sigmoid Neuron

A neuron is the basic processing unit inside a ANN, there are different neuron models, but we are going to focus on the one most used nowadays, sigmoid neuron. This neuron has n inputs (in[0],in[1],). These inputs can take on any values between 0 and 1. So, for instance, 0.43 is a valid input for a sigmoid neuron. It has weights for each input(w[0],w[1],), which determines the importance of each input and a single bias for all inputs, b.

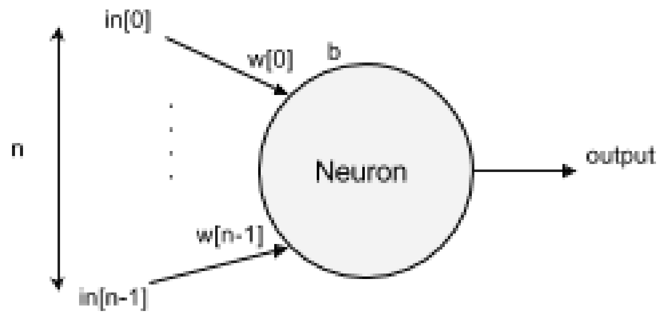


Figure 2.2: Sigmoid Neuron

Output is computed by:

$$\sigma(\sum_{j}^{n} w_j in_j + b) \tag{2.1}$$

Where σ is called the sigmoid function and is defined by:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.2}$$

To be more explicit, the output of a sigmoid neuron with inputs in[0],in[1]... weights w[0],w[1]... and bias b is

$$\sigma(z) = \frac{1}{1 + e^{-(\sum_{j}^{n} w_j in_j + b)}} \tag{2.3}$$

When z=w+b is a large positive number. Then $e - z \approx 0$ and so $(z) \approx 1$. So the output from the sigmoid neuron is approximately 1. When $z = w + b$ is very negative. Then $e - z \to \infty$, and $(z) \approx 0$. So the output is approximately 0. But what really matters is the shape of the function when plotted. Figure 2.3 shows the shape of the function:

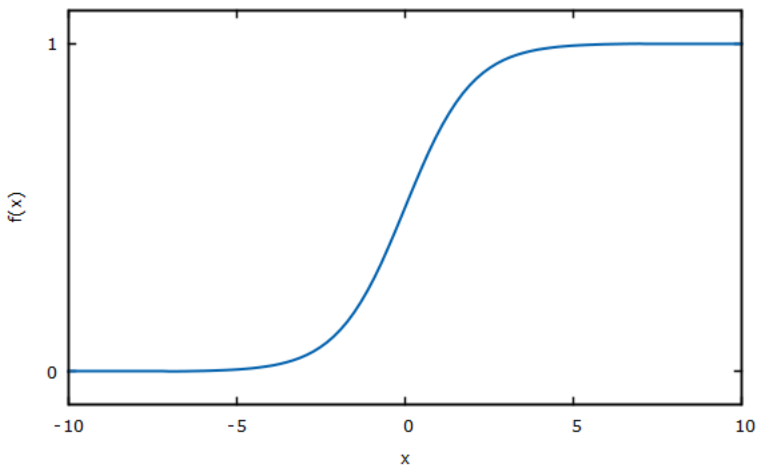


Figure 2.3: Sigmoid function plot

It's the smoothness of the σ function that is the crucial fact, not its detailed form. The smoothness of σ means that small changes in the weights Δwj and in the bias Δb will produce a small change in the output $\Delta output$ from the neuron.

With one neuron like this, we can only solve linear decision problems (datasets that could be linearly separable by drawing a line) Figure 2.4 The equation of the boundary line has the form:

$$b + in[0]w[0] + ... + in[n]w[n] \tag{2.4}$$

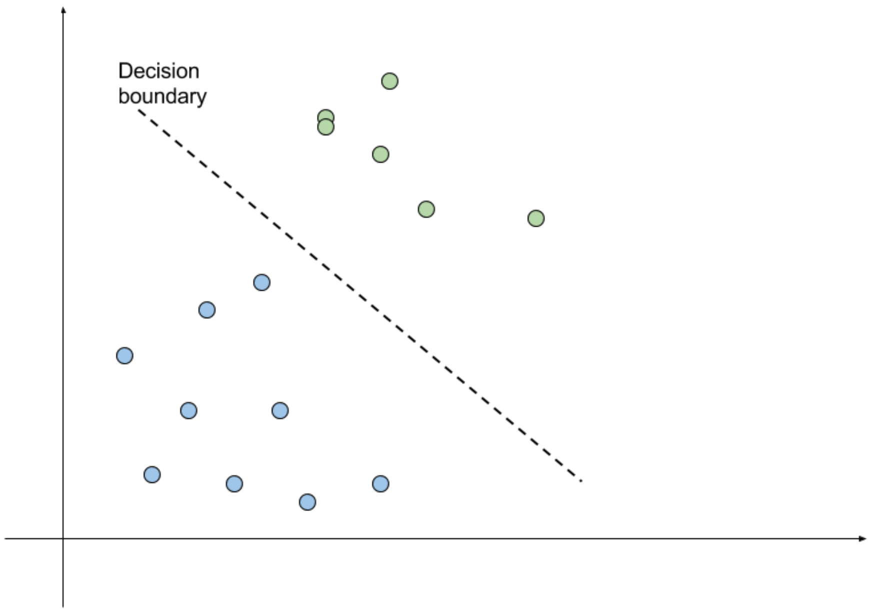


Figure 2.4: Linear decision

2.4 FFN

There are different types of ANN, we are going to focus on the Feed Forward Neural (FFN) or Multilayer Perceptrons (MLPs) network, where the output from one layer is used as input to the next layer. This means that there are no loops in the network, information is always fed forward, never fed back. If we had loops, we would end up with situations where the input to the function depended on the output. Neurons are usually arranged in layers: an input layer, an output layer and one or more intermediate layers called hidden layers. We chose to implement a FNN because it is the best documented ANN.

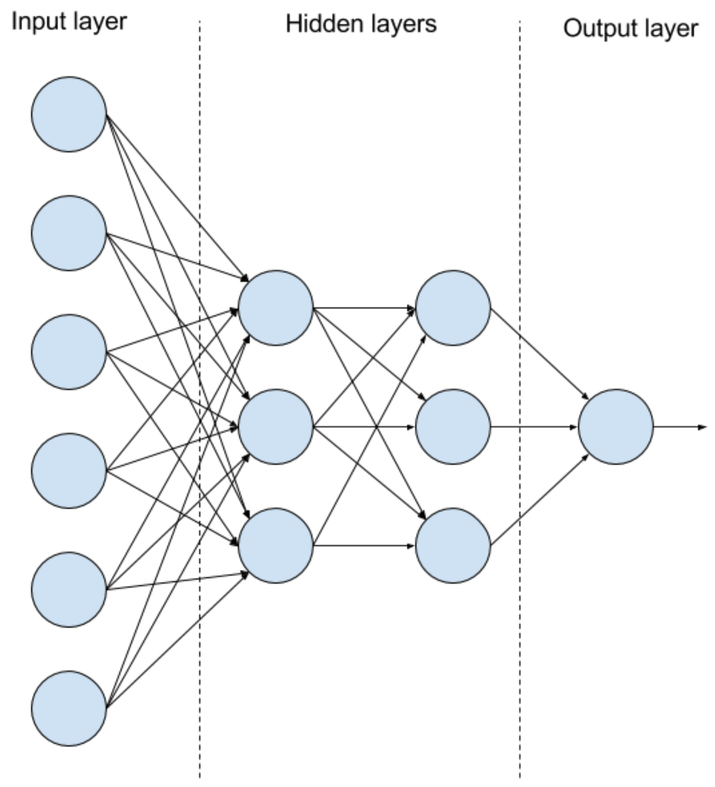


Figure 2.5: Artificial Neural Network

The design of the input and output layers are defined by the input and the output. For example, suppose we are trying to determine whether a handwritten image is a 0 or not. If the image is a 28 by 28 greyscale image, then we would have 28x28=784 input neurons that generate values scaled appropriately between 0 and 1. The output layer contains just a single neuron, with output values indicating the probability of the image being a 0.

The design of hidden layers is not as deterministic as those explained before, there is no rule for that. Neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behavior they want out of their nets. The difference between having one or more hidden layers resides in the complexity of the decision. At the end of the previous section, we saw how one neuron can solve linear decision problems. If we seek to solve non-linear ones, the answer is increasing the number of hidden layers.

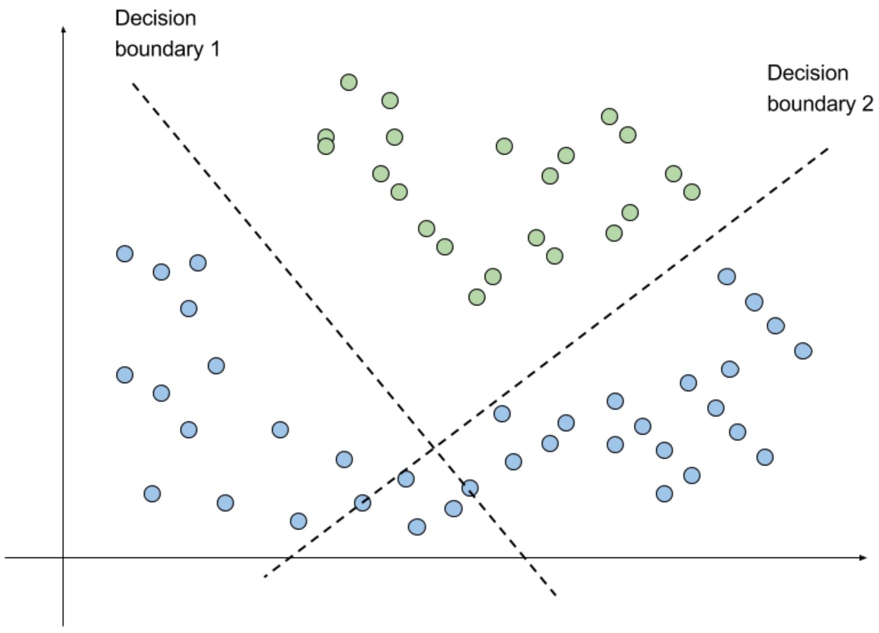


Figure 2.6: Non linear decision

2.5 Backpropagation

Due to its complexity, backpropagation won't be run on the FPGA in this work, but it will be executed on a CPU to train the network, therefore, it is necessary to be explained.

Backpropagation is a common method of training artificial neural networks used in conjunction with an optimization method such as gradient descent. The method calculates the gradient of a loss function with respect to all the weights in the network. The gradient is fed to the optimization method which in turn uses it to update the weights, in an attempt to minimize the loss function. It requires a known, desired output for each input value in order to calculate the loss function gradient. It is therefore considered a supervised learning method.

This algorithm will be explained step by step using the mathematical formulas. So, after an input is evaluated and produce an output, it can start the backpropagation. To calculate the error of the output layer we use the following formula:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{2.5}$$

Here, $\nabla_a C$ is defined to be a vector whose components are the partial derivatives $\frac{\partial C}{\partial a^l_j}$, which is multiplied by $\sigma'(z^L)$ using de Hadamard product($\odot$). You can think of it as the difference between the computed output and the desired output. This calculated error can be seen like the gradient of the bias of the neuron j.

$$\delta^l_j = \frac{\partial C}{\partial b^j_l} \tag{2.6}$$

In order to get the gradient of the weight associated of each neuron, it is necessary to multiple the error of j^{th} neuron by the output of the previous layer.

$$a^{l-1}_k \delta^j_l = \frac{\partial C}{\partial w^{jk}_l} \tag{2.7}$$

Once it has been computed the error of each neuron in the output layer, the previous layers error can be computed using the following formula:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{2.8}$$

Chapter 3

Methodology

The FPGA design was written in VHDL from a python code from the design published in [25] [26]. This was really useful in order to check the correct behavior of the network and also used this code executed in a CPU to compare with the design of this work. Our implementation is executed on an FPGA, but previously trained on a CPU. This training code creates the weights and bias, which will be used for evaluation.

Our input data is extracted from the well-known benchmark MNIST (Mixed National Institute of Standards and Technology), which is a large database of handwritten digits that is commonly used for training various image processing systems and also widely used for training and testing in the field of machine learning. The MNIST database contains 60,000 training images and 10,000 testing images, [7][3]. They are 28 by 28 pixel images of scanned handwritten digits, so the input layer contains 28x28=784 neurons. The input pixels are greyscale, with a value of 0.0 representing white, a value of 1.0 representing black, and in between values representing gradually darkening shades of grey.

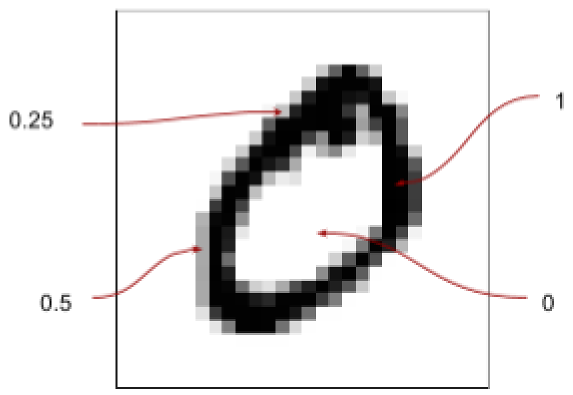


Figure 3.1: MNIST Image Encoding

Let's focus on the first output neuron, the one that's trying to decide whether or not the digit is a 0. It does this by weighing up evidence from the hidden layer of neurons. What are those hidden neurons doing? Well, just suppose for the sake of argument that the first neuron in the hidden layer detects whether or not an image like figure 3.2 is present. It can do this by heavily weighting input pixels which overlap with the image, and only lightly weighting the other inputs. In a similar way, let's suppose for the sake of argument that the second, third, and fourth neurons in the hidden layer detect whether or not the images in Figure.3.2 are present. These four images together make up the 0 image. So if all four of these hidden neurons are firing then we can conclude that the digit is a 0.

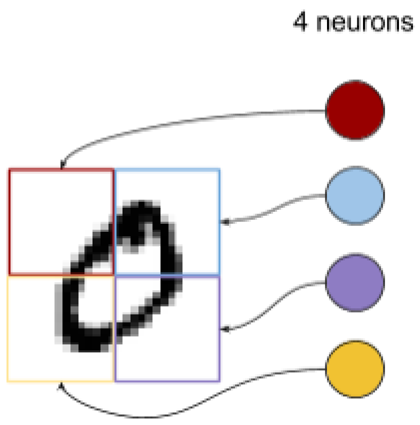


Figure 3.2: Image Neuron Detection

The output layer of the network contains 10 neurons. If the first neuron fires, i.e., has an output 1, then that will indicate that the network thinks the digit is a 0. If the second neuron fires then that will indicate that the network thinks the digit is a 1.

3.1 CPU implementation

CPU neural network implementation has been studied in [25] [26]. The performance of our design is compared also with this implementation.

The structure of the CPU program has been extracted with pycallgraph. Figure 3.3 shows the calls to extract the MNIST images, and the neural network. It has three block: mnist–loader to extract the image from the benchmark and store it in a matrix; random to generate random numbers and network which contains the neural network. Focusing on the module network, it has a function to initialize the network and a function to execute the benchmark, SGD. Inside SGD first is called a function to decide what is the image(feedforward) and then a function to learn from that evaluation(backprop). The benchmark is divided in mini–batches of images. This code spends more time in training than evaluating.

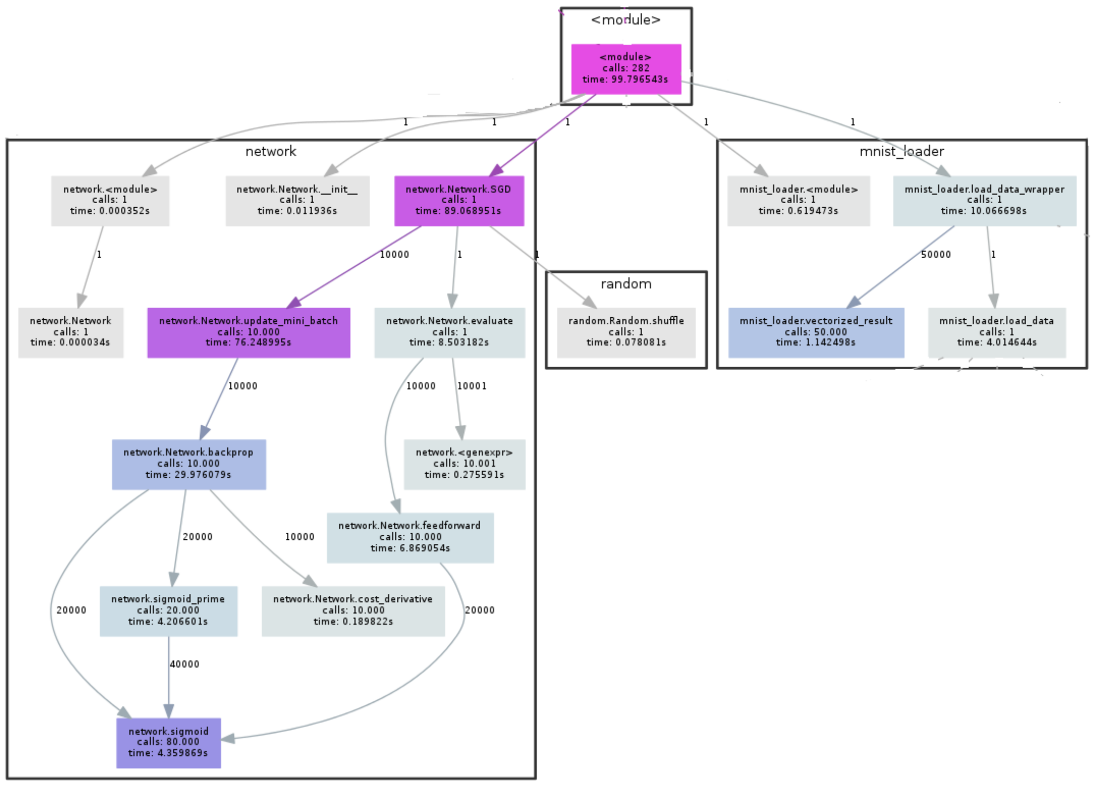

<module>
<module> calls: 282 time: 99.796543s
network
network.<module> calls: 1 time: 0.000352s
network.Network.__init__ calls: 1 time: 0.011936s
network.Network.SGD calls: 1 time: 89.068951s
network.Network calls: 1 time: 0.000034s
network.Network.update_mini_batch calls: 10.000 time: 76.248995s
network.Network.evaluate calls: 1 time: 8.503182s
network.Network.backprop calls: 10.000 time: 29.976079s
network.<genexpr> calls: 10.001 time: 0.275591s
network.Network.feedforward calls: 10.000 time: 6.869054s
network.sigmoid_prime calls: 20.000 time: 4.206601s
network.Network.cost_derivative calls: 10.000 time: 0.189822s
network.sigmoid calls: 80.000 time: 4.359869s
random
random.Random.shuffle calls: 1 time: 0.078081s
mnist_loader
mnist_loader.<module> calls: 1 time: 0.619473s
mnist_loader.load_data_wrapper calls: 1 time: 10.066698s
mnist_loader.vectorized_result calls: 50.000 time: 1.142498s
mnist_loader.load_data calls: 1 time: 4.014644s
1
1
1
1
1
1
10000
1
1
50000
1
10000
10000
10001
20000
10000
20000
20000
40000

Figure 3.3: Call graph of the CPU neural network

3.2 Configurations tested

Table 3.1: Configurations

Configurations	Precise	BinnaryConnect
Sigmoid function	Approx. function	Approx. function
MAC unit	Serial	No multipliers

Table 3.3 shows some different configurations.

3.3 Platforms and tools

During the development of the neural network, we used ModelSim 10.1e in order to simulate the behavior of the architecture. But, to make sure that the design is synthesizable and work properly on an FPGA, it has also been tested on an Altera DE-115 Fig.3.4 with the chip Cyclone IV EP4CE115F29C7 (Fig.3.5). [5] compiled using Quartus II 15.0.2. In order to use floating/fixed operators we used IP from Altera and the library [4].

In this work, we evaluate the different design options in the FPGA. But, it's also important to evaluate other platforms like a CPU. So, an Intel Core i7 CPU 870 2.93GHz is used.

Table 3.2: Platform features

	Cores	Processor Clock (MHz)
Intel Core i7	4	2800
Cyclone IV EP4CE115F29C7	1	200

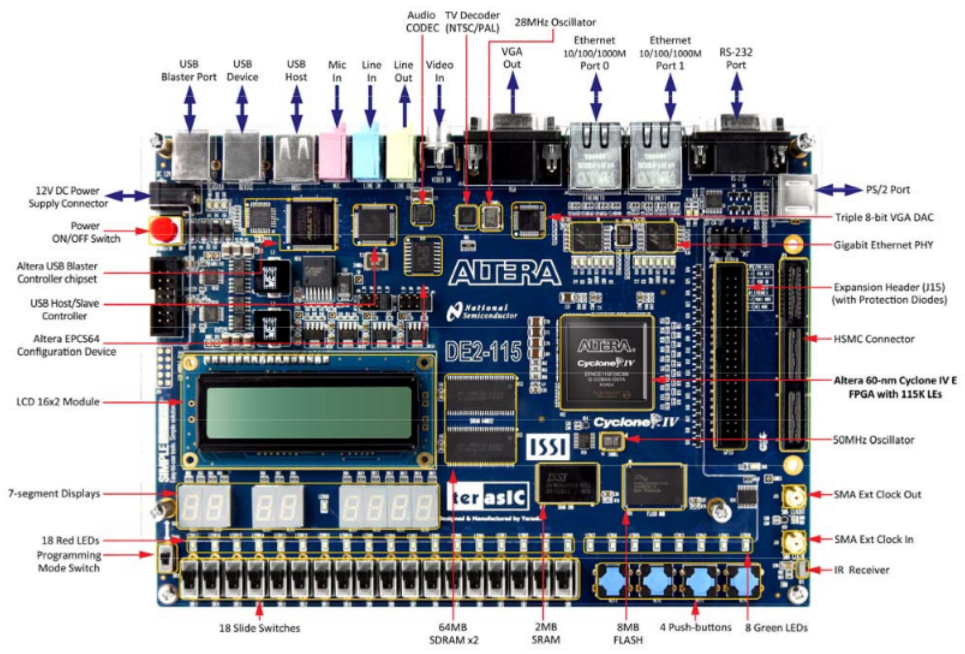


Figure 3.4: The DE2-115 board

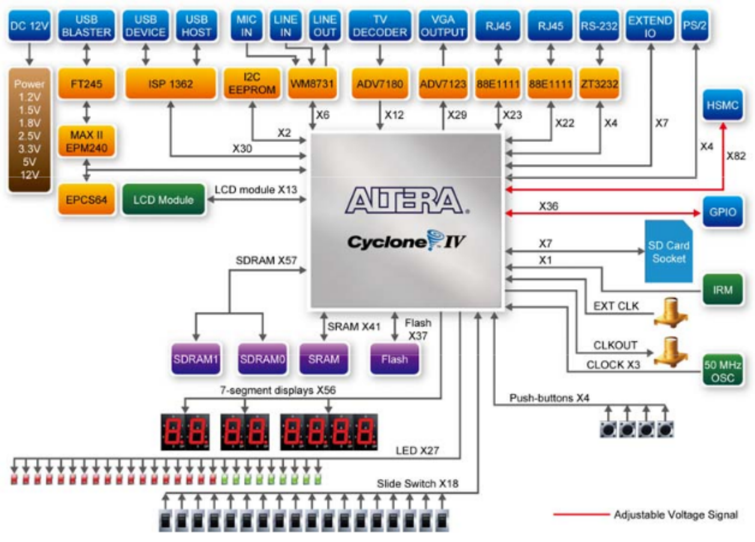


Figure 3.5: Block diagram of De2-115

Chapter 4

Implementation

4.1 System

All the knowledge (weights and bias) that needs the neural network in order to be trained are received by the SRAM interface. SRAM communication is a little slow, but this is only performed during the initialization phase. The Memory OnChip Interface stores the image which is going to be processed. The NeuralNetwork module contains a three layer Feed Forward Neural Network (784-100-10). The LCD Interface is connected to the LCD display and provides the final output values(source code from [17]). We have not used the DRAM of the board because of the complexity of its implementation.

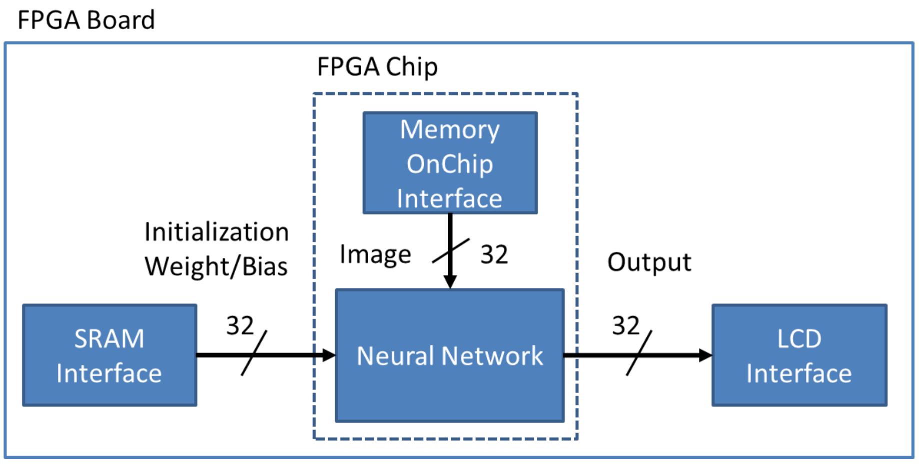


Figure 4.1: System architecture

4.2 Neural Network

Focusing on the Neural Network Design Fig.4.2, as previously stated, it has a three layer structure (784-100-10). The first layer (Input layer) would be the input of the image, the 784 pixels of the image that are connected to the next layer (Hidden layer). In the real model, Layer1 has 100 neurons, but in the FPGA design, we can have a different number of hardware neurons they map to depending on the resources availability (1 to 100). The same thing is applied to the last layer (Output Layer). (RTL diagram: A.1)

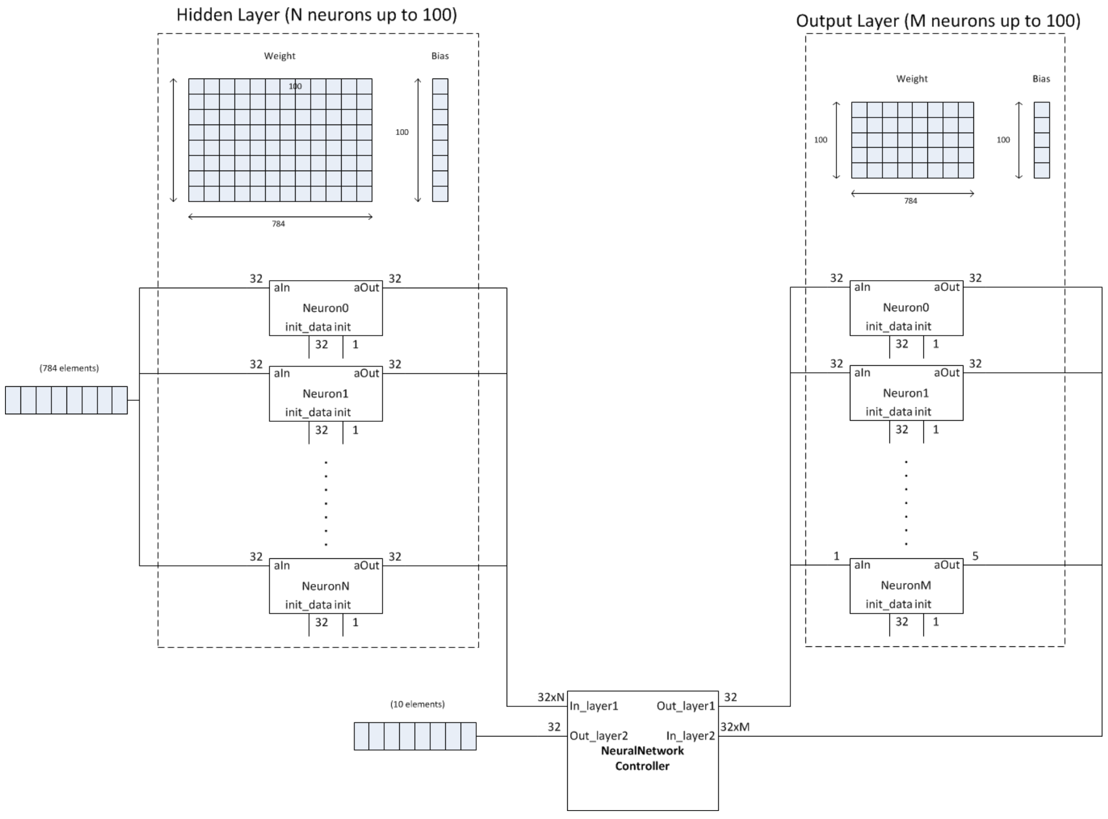


Figure 4.2: Neural Network architecture

The resources in Table 4.1 are from the minimum size neural network using one hardware neuron per layer.

Table 4.1: Resources Neural Network

LC Combinationals	LC Registers	Memory Bits	DSP Elements
8938	4514	2508800	14

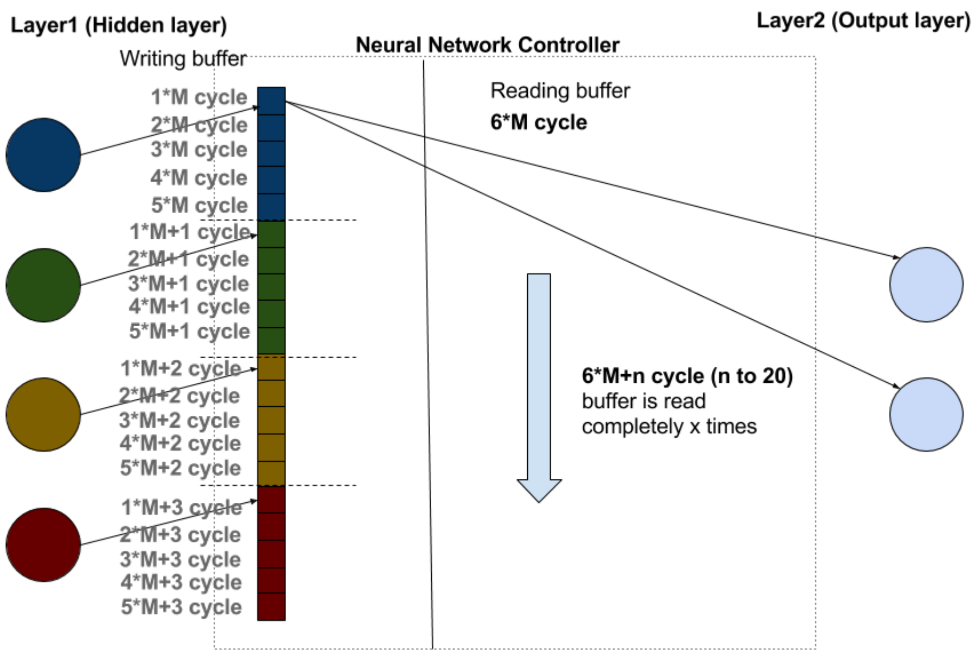
4.2.1 Neural Network Controller

When the neuron units of the hidden layer finish, the controller receives a signal to collect the outputs and copies them into a temporal buffer. This data is written following Figure 4.3, one neuron -and only one- writes in the buffer each cycle. So, in the example, the four neurons write its data one after the other.

When all of the writes are finished, they start reading the new set of inputs during the next M cycles. M is the size of the image, which is computed by the MAC and this is the reason for this latency. This process is iteratively done as long as there are neurons still to be processed.

Once all the outputs from the hidden layer are received, this data is split and sent to the neuron of the output layer. The temporal buffer is read from the beginning to the end, and because one hardware neuron can represent more than one real neuron, this buffer is read x times, where x is the number of real neurons that represent a hardware neuron. Every neuron of the output layer receives the same data.

The same process is repeated when the output layer finishes the computation. The final result is the output of each output neuron which indicates the probability of each possible solution.



4 hardware neurons in Layer1, each of them doing the work of 20 neurons
20 elements in the buffer = 20 real neurons
M = size of the image
2 hardware neurons in Layer2, each of them doing the work of 4 neurons
x = real neuron/hardware neuron of Layer2 , in this case 4/2 = 2

Figure 4.3: Neural Network Controller

The same process is repeated when the output layer finish the computation. The final result is the output of each output neuron which indicates the probability of each possible solution.

4.2.2 Neuron

In the first design, we can see that the vector multiplication that computes the data with its respective weight is inside of the neuron with the bias addition and the Sigmoid function. (RTL diagrams: A.2 and A.3)

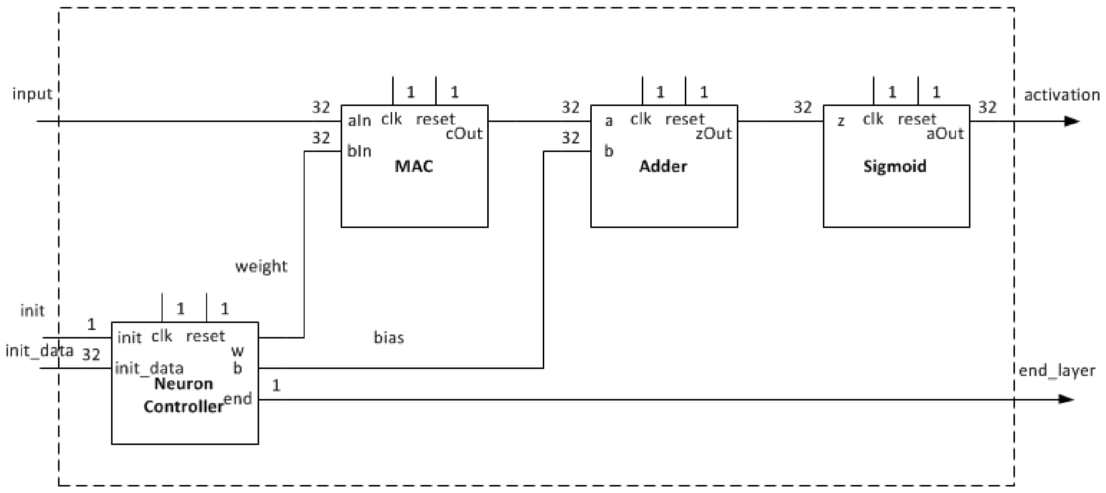


Figure 4.4: Neuron Architecture

Table 4.2: Resources Neuron

LC Combinationals	LC Registers	Memory Bits	DSP Elements
3203	411	2508800	7

4.2.2.1 Multiplier-accumulator

The computation of the inputs and the weights of the different connections. To apply the weight to an input is done through a multiplication. There are different options to implement this block, it can be a full parallel architecture, a serial one or in between these two, a semi-parallel one.

So, in Fig.4.5 we can see a design where multiplications are parallelized at cost of using more area in the FPGA to create one multiplier for each pair of values <input, weight> and then sum all together.

Assuming that multiplication costs 2 cycles and additions 1 cycle, the cost of the function would be 2+log(n) cycles, where n is the number of inputs.

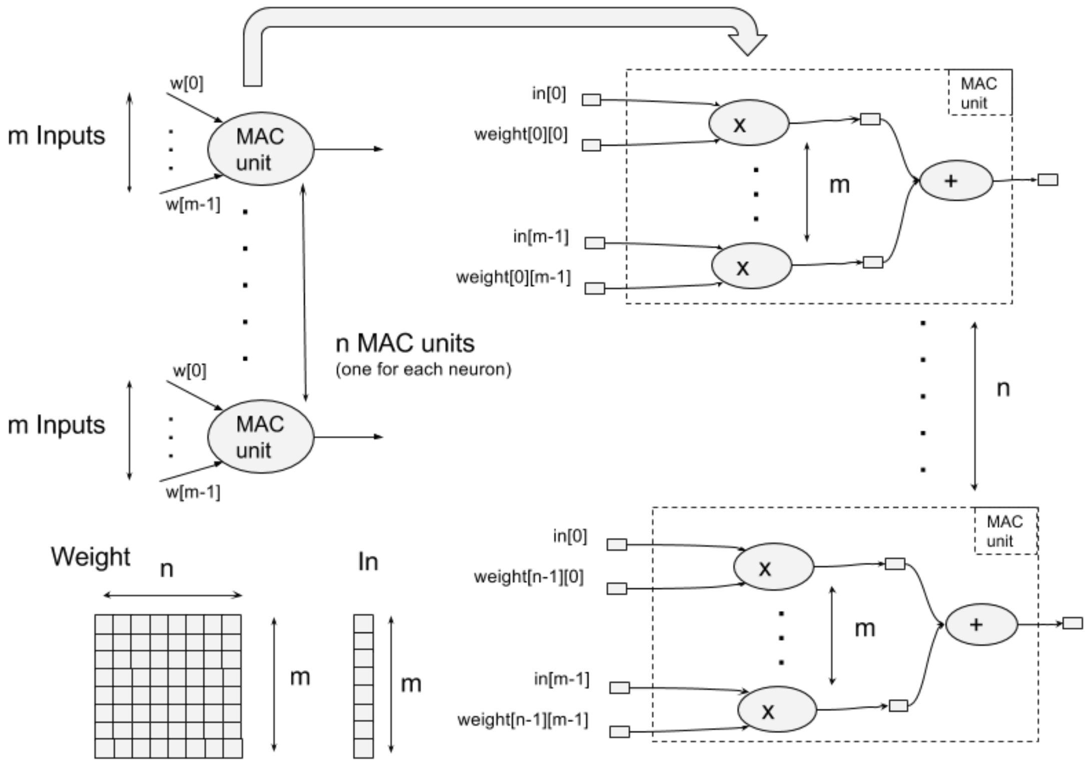


Figure 4.5: Multiplier-accumulator parallel architecture

The previous option implies a big usage of resources of the FPGA to implement all the multipliers and adders. To reduce the the usage of resources reducing the performance, an option can be seen in the Fig.4.6, where the multiplications are serialized. In this case the cost would be 2+n (Multipliers and adders are pipelined)

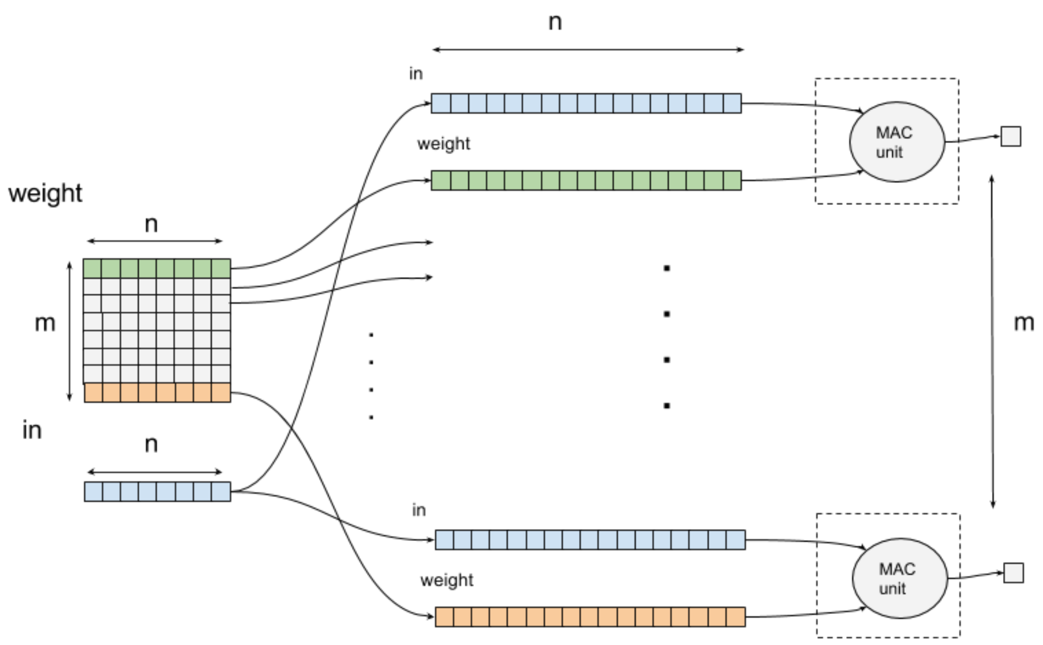


Figure 4.6: Multiplier-accumulator serial architecture

Table 4.3: Resources MAC Serial

LC Combinationals	LC Registers	Memory Bits	DSP Elements
994	183	0	7

But, sometimes we do not have enough resources to implement a full parallel architecture and a serial one is not giving the desired performance. The halfway solution would be a semi-parallel architecture where you can adjust the level of parallelism. The cost would depend on the level of parallelism.

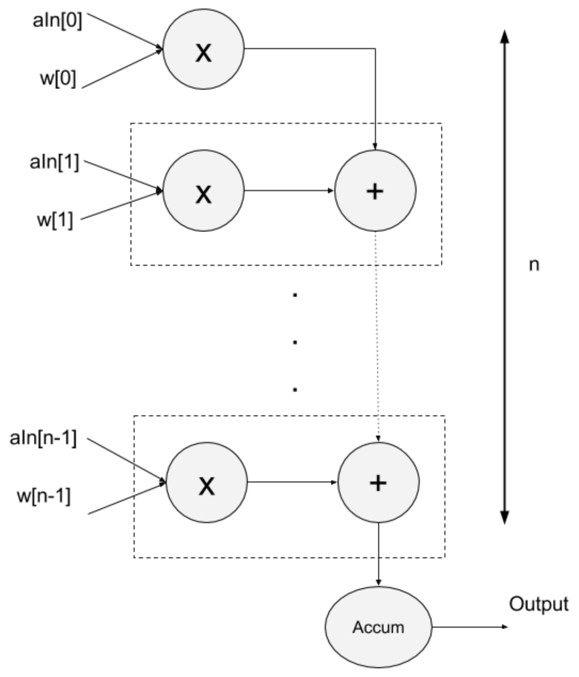


Figure 4.7: Multiplier-accumulator semiparallel architecture

A parallel MAC is not implemented, instead of that, it is be improved in a better way explained in the next subsection.

4.2.2.2 MAC improved

Until here, we have seen the quantity of floating point multiplications performed in a neural network. In an FPGA, improvement of the performance implies increasing the number of neuron units, and therefore, increasing the number of multipliers. So, in this section a MAC unit without multiplier is shown, following the idea covered in [19][6]. The papers, in broad terms, transform weights into 1 or -1.

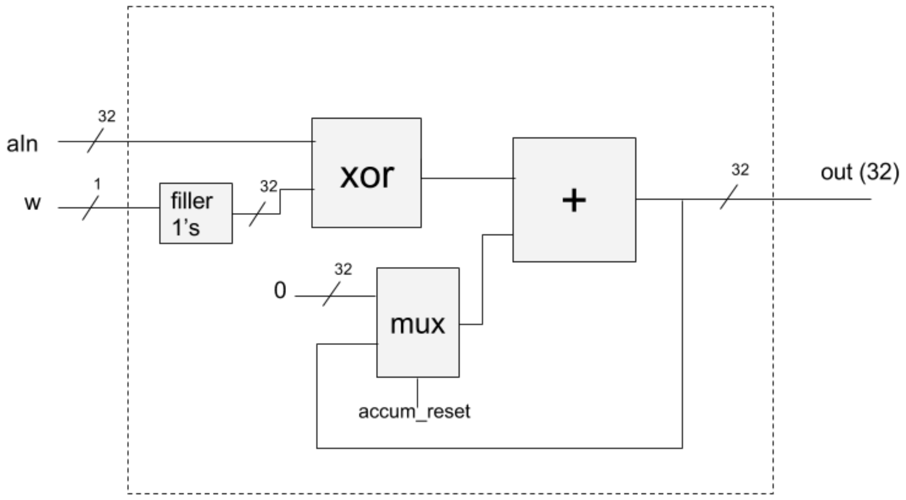


Figure 4.8: Accumulator without multiplier

As we have to multiply by 1 or -1, the multiplier is replaced by an XOR function in order to change the sign of the input aIn. The "filler 1's" box is for expanding the weight bit to 32bits, filling the other bits with 1(no effects on aIn). So, when weight is 1, it is positive and the XOR function doesn't change the sign bit on aIn, but when is 0 (is negative), it changes the sign of aIn. With this, especially power consumption is improved. In terms of logic elements, it is reduced to 19.91%. Also the number of embedded multipliers in the FPGA are reduced to 0.

Table 4.4: MAC Binary Connect

LC Combinationals	LC Registers	Memory Bits	DSP Elements
238	97	0	0

4.2.2.3 Sigmoid function

There are different methods to compute the sigmoid function. The more intuitive method would be to implement the function in the way it is, but this function computes an exponential and a division, and this can be costly in terms of resources and slow. Division costs 11 cycles, exponential 5 cycles, substraction 1 cycle, so the total cost of the function is 17.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{4.1}$$

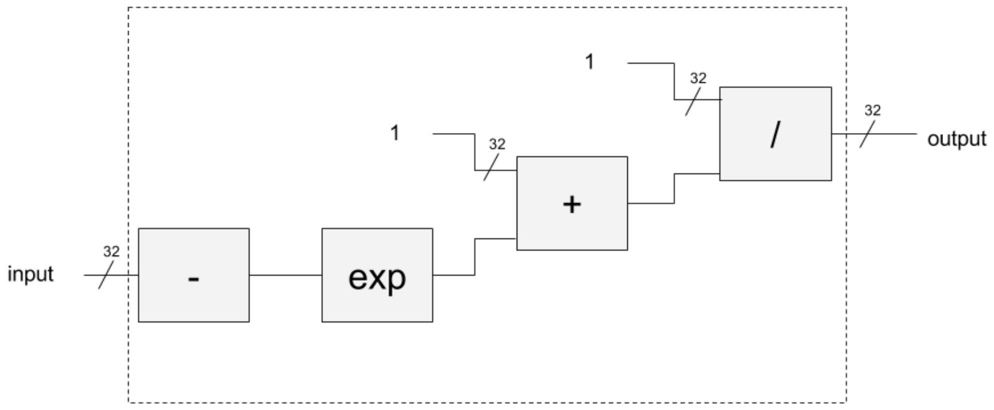


Figure 4.9: Complete sigmoid architecture

Table 4.5: Resources Sigmoid Original

LC Combinationals	LC Registers	Memory Bits	DSP Elements
2883	680	40448	15

Another method used in several experiments is an approximation function. This has been implemented with fixed point using the library [4] (not the Altera IP). In this work the following function is used:

$$\sigma(z) = min(max(\frac{1}{1 + z}), 0), 1) \tag{4.2}$$

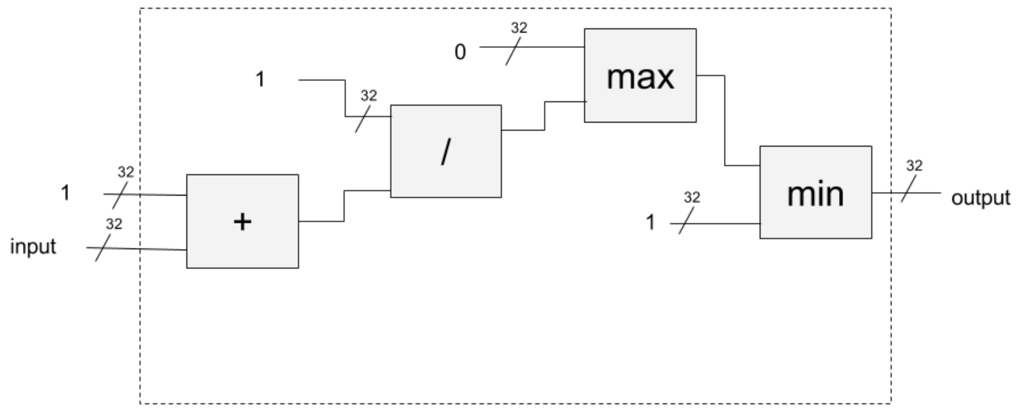


Figure 4.10: Approximation sigmoid architecture

About resources, due to be using a library an not the Altera IP, in terms of LC combinationals we use more than the previous version, but we improve on the other features.

Table 4.6: Resources Sigmoid aprox

LC Combinationals	LC Registers	Memory Bits	DSP Elements
3901	126	0	0

4.2.2.4 Neuron Controller

Neuron Controller is inside of each neural unit and stores the weights and bias of the real neurons which represents. So, before processing an image the weights and bias are initialized inside of each neuron controller. During the process of one input of the neural unit, the controller provides the weight and bias for MAC and the adder respectively.

Chapter 5

Evaluation

5.1 Evaluating different platforms

The software based neural network was very useful in order to validate the VHDL design. Moreover, a comparative between the executions was performed. In this case, the hardware implementation used is the one with the biggest precision.

Table 5.1: Comparative platforms

	CPU	FPGA
Time (ms) per decision	3.4632	0.794475
Power (mW)	31000	149.48

In Table 5.1, a comparison of the performance among CPU and FPGA (2800MHz and 200MHz respectively). With the fastest FPGA design, we achieve a much lower computation time than the CPU design, taking into account that the language used is Python and is not considered the fastest programming language. Implementing a LUT yields the best results in terms of computation time over precision.

5.1.1 Error analysis

Comparing the final results with the different versions, we encountered with a little variance. So, we extract the absolute error (Table 5.2). We analyzed the error between: the python version and the hardware version with activation using an approximation function (PyApprox), and with the complete sigmoid function (PySig) and finally between Sigmoid and approximation function activation (SigApprox).

The error PyApprox was expected because of the use of an approximation function for the Sigmoid function. Not so expected was the error found in the complete sigmoid function, this is due to the rounding in the floating operations (still it is 4 orders of magnitude smaller).

Table 5.2: Absolute error

Error PyApprox	Error PySig	Error SigApprox
5.64E-03	9.39E-07	5.64E-03

5.2 BinaryConnect version

This is the point which makes this design a little different, because this idea it has been subject of very recent works and usually implemented only in software solutions. But, it has several good points to be implemented on an FPGA.

Table 5.3: Precise and BinnaryConnect

	Time(us)	Area(logic elem.)	Mem bits	Power(mW)	Mult.
Precise(float)	794.140	8938	2547520	695.66	14
BinaryConn(fixed)	794.110	13473	135296	473.29	0
Improvement	-0.004%	33.65%	-94.68%	-31.97%	-100%

In terms of time, there is no difference. But considering the rest, this solution offers great benefits. It completely eliminates the usage of embedded multipliers, improve the power consumption notably and the most remarkable thing is the usage decrease in memory bits, with a -99.74%, because it uses 1 bit of weight instead of 32 bits. In terms of LC combinationals it uses more than the previous version, due to the new library and not Altera's IP.

5.3 Scalability and restrictions

This work aims to provide enough flexibility (through the usage of parameters) in order to be able to change the resources used, just changing a configuration file. In the following subsections, we evaluate different design points in terms of execution time, power, energy efficiency and area.

5.3.1 Performance

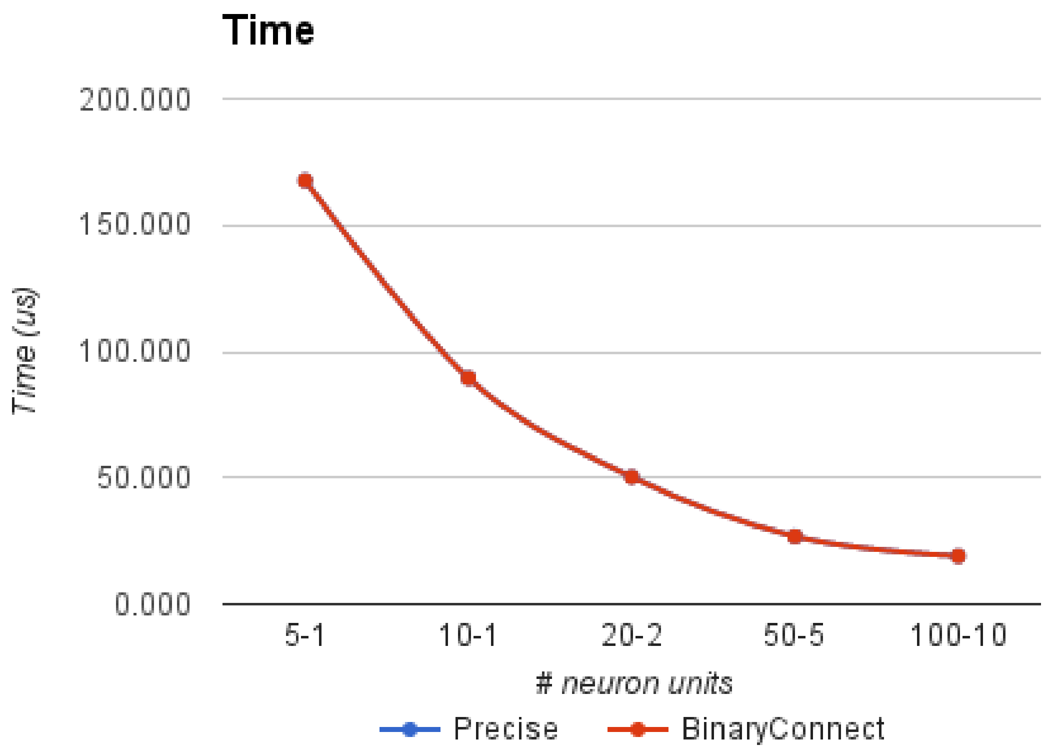


Figure 5.1: Time plot

The performance behaves as expected. When you increase the number of hardware neurons the speed up is almost linear. (Remember, the real number of neurons never changes, but you can parallelize or serialize the execution on the hardware). The horizontal axis represents the number of hardware neuron units of each layers (hidden layer-output layer). 5-1 means that there are 5 neurons in the hidden layer and 1 in the output layer.

5.3.2 Area

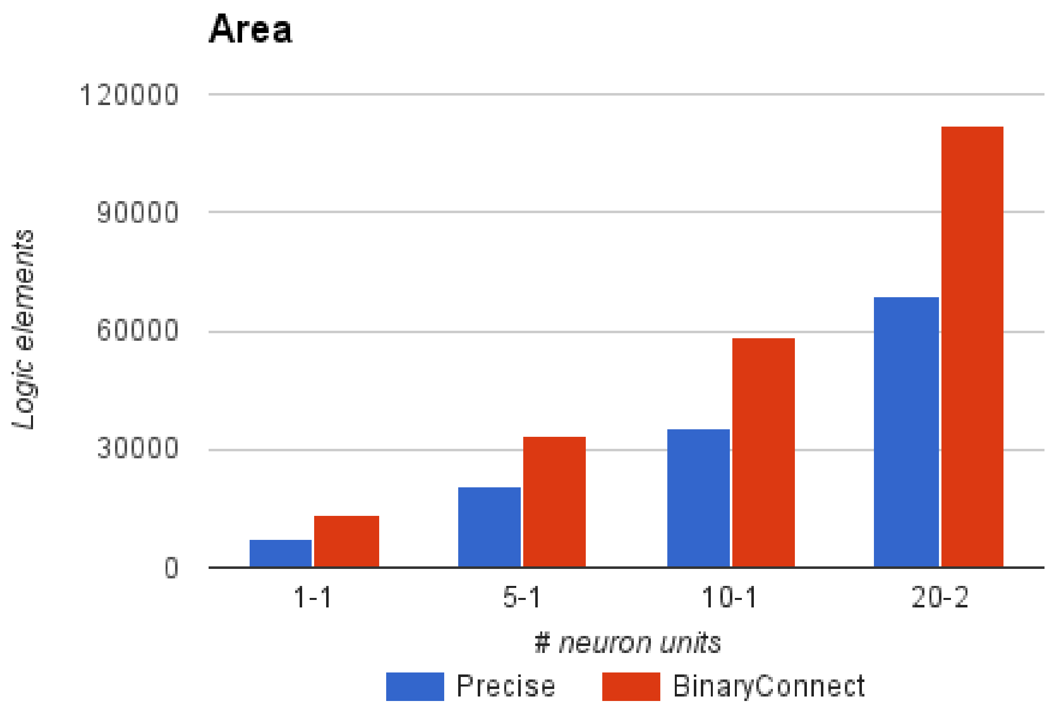


Figure 5.2: Area plot

An FPGA has a limited number of resources, the basic component are the logic elements. So we study how the area varies with the different changes in the design. In terms of resources, the version using binary connect uses more logic elements because is using a VHDL library and fixed floating point, but it reduces memory and allows to implement a lighter MAC.

5.3.3 Power

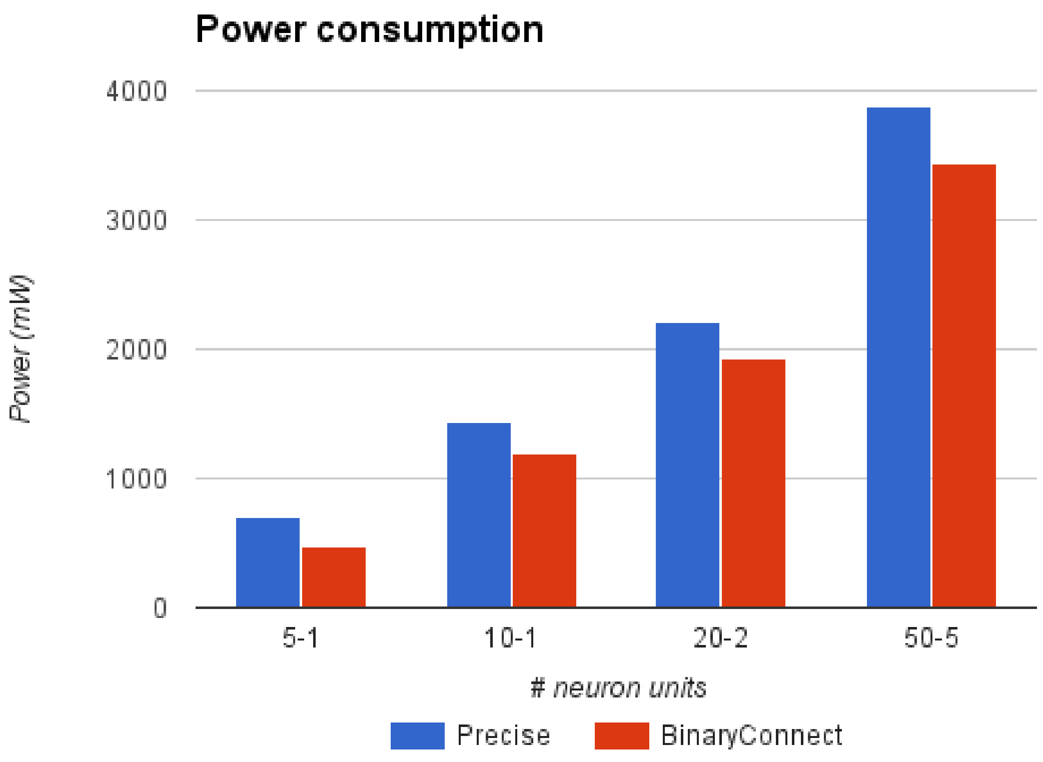


Figure 5.3: Power plot

Nowadays, the power consumption is a key aspect in every workload execution, because power is money. So, in this section we will analyze how changes the power consumption with the different implementations. Obviously, Increasing the number of hardware neurons increase the power consumption, but we have to take into account that it also increases performance, so the time to process one image is lower.

5.3.4 EnergyDelay^2

With $EnergyDelay^2$, we measure the energy efficiency of performing an execution in a given design, combining performance and energy metric. We see in 5.4 that using 5–1 neurons, the BinaryConnect version is more efficient, but when we increase the number of neurons the efficiency of both versions approach each other.

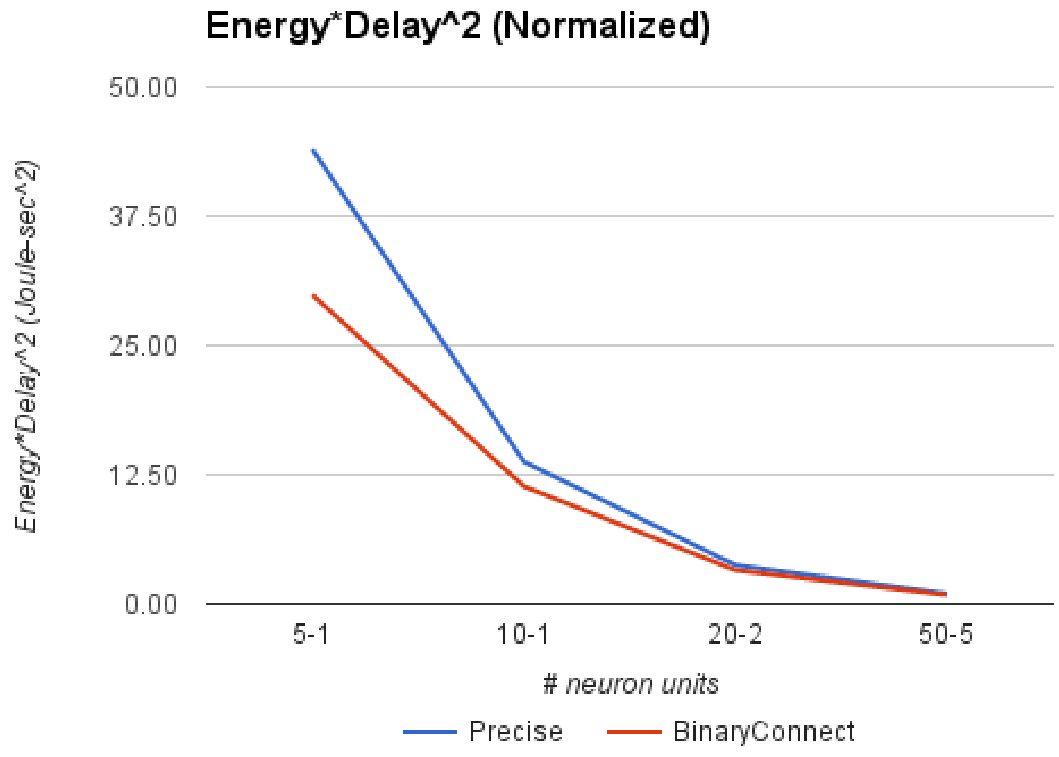


Figure 5.4: EnergyDelay$^2 plot$

Chapter 6

Conclusions and future work

6.1 Conclusions

Our work provides a design space evaluation of a neural network, the implementation on an FPGA offers almost an unlimited number of possibilities to improve the performance only limited to the available resources.

First,taking advantage of the inherent parallelism, a flexible design has been implemented in order to help in this work. This allows to check how important is the resources optimization, because increasing the number of neurons reduces the computation time at cost of increasing the resources, which in an FPGA, are limited.

Second, focusing in one of the more interesting parts, the multiplier-accumulator of each neuron. An improved version of MAC has been implemented in order to reduce considerably the quantity of memory, logic elements and embedded multipliers.

Third, the activation function, in this case, sigmoid function. Two possible configurations are permitted in this work. The complete implementation has the maximum precision at cost of the speed and the area. The approximation function, which consumes more logic elements in our implementation (due to be using a VHDL library and not Altera IP) uses less memory and multipliers.

The results of the comparison with the Python neural network showed as expected, although necessary in order to check the correct functionality of the neural network. Checking the results, little errors have been found, although both versions use floats of 32 bits, the methods for rounding cause a little variance. But, we checked that these small differences do not affect the system. This could be something to take into account when a more complex decision has to be made by a neural network.

6.2 Future work

In this work, we focused in resource optimization, but there is still more work to do in this direction. The forward computation part of the neural network has been implemented reducing the resources consumption and eliminating the usage of embedded multipliers. Continuing the development of this design would consist on including backpropagation and apply the same optimizations, also other recent techniques as [19] and [6] could be used and analyzed how they we can be implemented in a hardware platform.

References

[1] New ibm synapse chip could open era of vast neural networks, 2014. 6

[2] Haitham Kareem Ali and Esraa Zeki Mohammed. Design artificial neural network using fpga. *International journal of computer science and network security*, 10(8):88–92, 2010. 7

[3] Rodrigo Benenson. Mnist ranking, 2014. 14

[4] D. Bishop. fixed/float vhdl library, 2000. 8, 18, 29

[5] Altera Corporation. De2-115: User manual, 2010. 18

[6] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3105–3113, 2015. 3, 7, 8, 27, 38

[7] Yann Le Cun. Mnist webpage, 2015. 14

[8] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989. 5

[9] Daniel Ferrer, Ramiro González, Roberto Fleitas, Julio Péerez Acle, and Rafael Canetti. Neurofpga-implementing artificial neural networks on programmable logic devices. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 3, pages 218–223. IEEE, 2004. 7, 8

[10] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980. 5

[11] S Hariprasath and TN Prabakar. Fpga implementation of multilayer feed forward neural network architecture using vhdl. In *Computing, Communication and Applications (ICCCA), 2012 International Conference on*, pages 1–6. IEEE, 2012. 7

[12] Donald Olding Hebb. *The organization of behavior: A neuropsychological approach.* John Wiley & Sons, 1949. 5

[13] S Himavathi, D Anitha, and A Muthuramalingam. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *Neural Networks, IEEE Transactions on*, 18(3):880–888, 2007. 7, 8

[14] Thang Viet Huynh. Design space exploration for a single-fpga handwritten digit recognition system. In *Communications and Electronics (ICCE), 2014 IEEE Fifth International Conference on*, pages 291–296. IEEE, 2014. 7

[15] Jonghong Kim, Kyuyeon Hwang, and Wonyong Sung. X1000 real-time phoneme recognition vlsi using feed-forward deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 7510–7514. IEEE, 2014. 8

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 1

[17] Scott Larson. Lcd module controller (vhdl), 2013. 20

[18] Antonino Laudani, Gabriele Maria Lozito, Francesco Riganti Fulginei, and Alessandro Salvini. An efficient architecture for floating point based miso neural neworks on fpga. In *Computer Modelling and Simulation (UKSim), 2014 UKSim-AMSS 16th International Conference on*, pages 12–17. IEEE, 2014. 7

[19] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009*, 2015. 3, 7, 27, 38

[20] Gabriele-Maria Lozito, Antonino Laudani, Francesco Riganti Fulginei, and Alessandro Salvini. Fpga implementations of feed forward neural network by using floating point hardware accelerators. *Advances in Electrical and Electronic Engineering*, 12(1):30–39, 2014. 7

[21] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. 5

[22] Microsoft. Microsoft is building fast low power neural networks with fpgas, 2015. 6

[23] Marvin Minsky and Seymour Papert. Perceptrons. 1969. 5

[24] MIT. Neural chip artificial intelligence mobile devices, 2016. 6

[25] Michael A. Nielsen. "neural networks and deep learning" determination press, 2015. 14, 16

[26] Michael A. Nielsen. "neural networks and deep learning" determination press (codes), 2015. 14, 16

[27] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. 5

[28] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. 1974. 5

[29] Bernard Widrow and Marcian E Hoff. Adaptive switching circuits. in 1960 wescon convention record part iv, pages 96–104. reprinted in ja anderson and e. rosenfeld. *Neurocomputing: Foundations of Research*, 1960. 5

Appendix A

Source code and RTL Diagrams

The source code will be included as a separate file to this work.

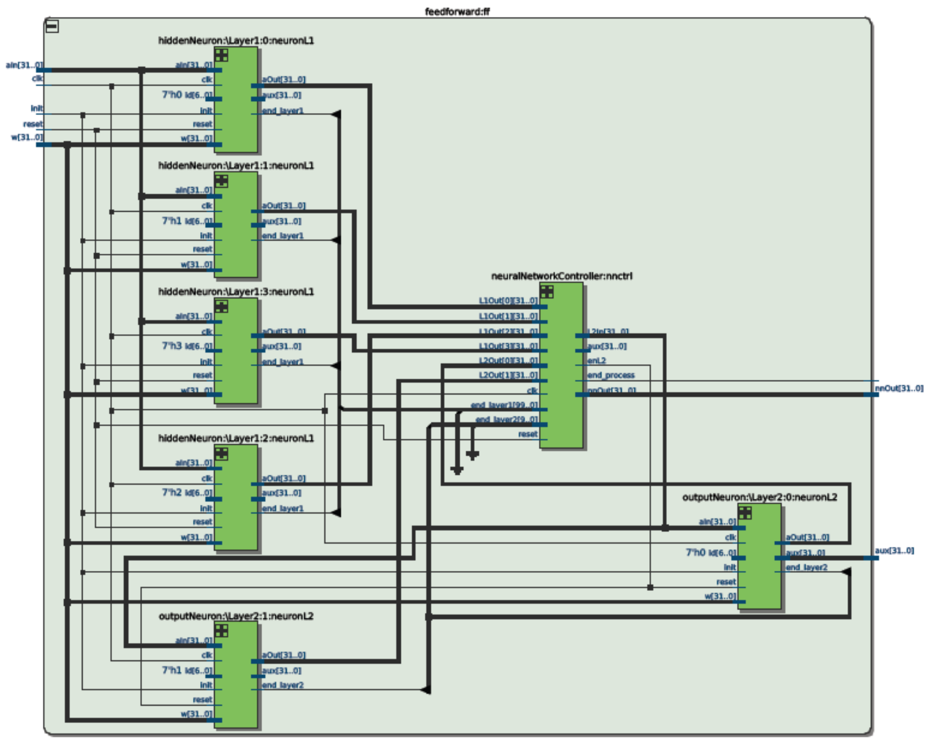


Figure A.1: RTL Neural Network with 4 hidden neuron and 2 output neuron

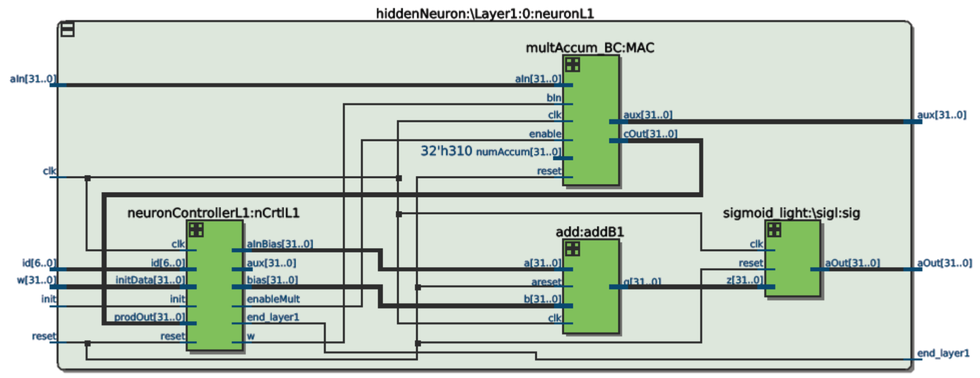


Figure A.2: RTL Hidden Neuron

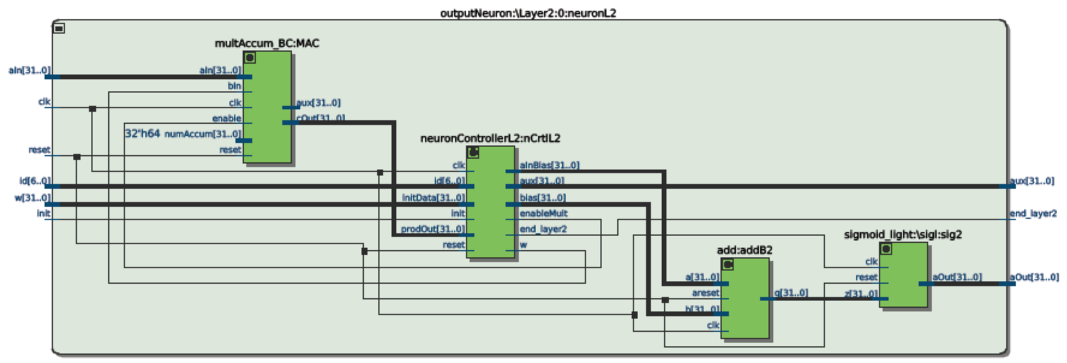


Figure A.3: RTL Output Neuron