COMPACT YET EFFICIENT HARDWARE ARCHITECTURE FOR MULTILAYER-PERCEPTRON NEURAL NETWORKS

Rodrigo Martins da Silva[*]
rdgengel@ig.com.br

Nadia Nedjah[*]
nadia@eng.uerj.br

Luiza de Macedo Mourelle[†]
ldmm@eng.uerj.com

[*]Department of Electronics Engineering and Telecommunications
State University of Rio de Janeiro
Rio de Janeiro, Brazil

[†]Department of Systems Engineering and Computation
State University of Rio de Janeiro
Rio de Janeiro, Brazil

RESUMO

Arquitetura de hardware compacta e eficiente para redes neurais artificiais do tipo múltiplas camadas
Em termos computacionais, uma rede neural artificial (RNA) pode ser implementada em *software* ou em *hardware*, ou ainda de maneira híbrida, combinando ambos os recursos. O presente trabalho propõe uma arquitetura de *hardware* para a computação de uma rede neural do tipo perceptron com múltiplas camadas (MLP). Soluções em *hardware* tendem a ser mais eficientes do que soluções em *software*. O projeto em questão, além de explorar fortemente o paralelismo das redes neurais, permite alterações do número de entradas, número de camadas e de neurônios por camada, de modo que diversas aplicações de RNAs possam ser executadas no *hardware* proposto. Visando a uma redução de tempo do processamento aritmético, um número real é aproximado por uma fração de inteiros. Dessa forma, as operações aritméticas limitam-se a operações inteiras, executadas por circuitos combinacionais. Uma simples máquina de estados é demandada para controlar somas e produtos de frações. A função de ativação usada neste projeto é a sigmóide. Essa função é aproximada mediante o uso de polinômios, cujas operações são regidas por somas e produtos. Um teorema é introduzido e provado, permitindo a fundamentação da estratégia de cálculo da função de ativação. Dessa forma, reaproveita-se o circuito aritmético da soma ponderada para também computar a sigmóide. Essa re-utilização dos recursos levou a uma redução drástica de área total de circuito. Após modelagem e simulação para validação do bom funcionamento, a arquitetura proposta foi sintetizada utilizando recursos reconfiguráveis, do tipo FPGA. Os resultados são promissores.

PALAVRAS-CHAVE: Redes neurais artificiais, hardware para redes neurais, sigmóide, paralelismo, FPGA.

ABSTRACT

There are several neural network implementations using either software, hardware-based or a hardware/software codesign. This work proposes a hardware architecture to implement an artificial neural network (ANN), whose topology is the multilayer perceptron (MLP). In this paper, we explore the parallelism of neural networks and allow *on-the-fly* changes of the number of inputs, number of layers and number of neurons per layer of the net. This reconfigurability characteristic permits that any application of ANNs may be implemented using the proposed hardware. In order to reduce the processing time that is spent in arithmetic computa-

Artigo submetido em 10/03/2011 (Id.: 01299)
Revisado em 06/05/2011, 19/07/2011
Aceito sob recomendação do Editor Associado Prof. Carlos Roberto Minussi

tion, a real number is represented using a fraction of integers. In this way, the arithmetics is limited to integer operations, performed by fast combinational circuits. A simple state machine is required to control sums and products of fractions. Sigmoid is used as the activation function in the proposed implementation. It is approximated by polynomials, whose underlying computation requires only sums and products. A theorem is introduced and proven so as to cover the arithmetic strategy of the computation of the activation function. Thus, the arithmetic circuitry used to implement the neuron weighted sum is reused for computing the sigmoid. this resource sharing decreased drastically the total area of the system. After modeling and simulation for functionality validation, the proposed architecture synthesized using reconfigurable hardware. The results are promising.

KEYWORDS: Artificial neural networks, hardware for neural networks, sigmoid, parallelism, FPGA.

1 INTRODUCTION

An artificial neural network (ANN) is an attractive tool for solving problems such as pattern recognition, generalization, prediction, function approximation, optimization and non–linear system behavior mapping. When dealing with an ANN implementation, systems based on hardware are usually faster than software alternatives (Zurada, 1992; Rojas, 2010; Zhu and Sutton, 2003; Dias et.al, 2004; Omondi and Rajapakse, 2008).

When a particular task does not require so much speed, a software-based neural network system can be sufficient and satisfactory in running the task, through a PC or a general-purpose processor. ANN systems based on software do not demand much design effort. On the other hand, ANNs provide an adequate research field for applying the parallel computation and, of course, this parallelism can be best explored in a hardware-based implementation (Chen, 2003; Omondi and Rajapakse, 2008). So, a hardware architecture can be devised in order to use the massive parallelism provided in the neuron–layer computation. Circuit components can be designed and adequately mapped to exploit details from both the arithmetic computation and control process.

The Hardware designed, in this work, can be used by any neural network applications. It supports ANNs with different number of layers, neurons per layer and inputs. This is one of the proposed hardware main features: flexibility through and *on-the-fly* reconfigurability. To perform a multilayer perceptron neural network (MLP), the hardware requires the following parameters:

1. The number of inputs of the network. Let i_{max} be this number;
2. The number of layers of the network. Let l_{max} be this number;
3. The number of neurons per layer. Let n_i be this number, where i represents the i^{th} layer: $i = 1, 2, \ldots, l_{max}$;
4. If at least one neuron of a certain layer is going to operate with a bias, parameter *bias* of such layer must be *on*.

A neural network usually has more than one layer. Nevertheless, the proposed hardware provides only one single physical layer, a *hardware layer*, which performs the entire computation due to all the layers of the ANN by reusing the neurons of this unique physical layer. In this context, the neural network layers are named *virtual layers*. Note that this is done without loss of performance as the computation due to the ANN layers are data-dependent, and thus need to be executed sequentially. This strategy reduces the designed circuit area. Note that there exists an overhead due to the required control to use a single physical layer. However, the time spent is minimal and so has a very little impact on the overall performance of ANN hardware. The time spent computing the weighted sum and activation function is far longer than that spent controlling the layers computation. Moreover, the neurons of the physical layer operate in parallel to perform the required computation. For instance, weighted sums are computed by all hardware neurons at the same time, and so is the case of the computation of the activation function. Hence, the overall processing time of the ANN is also reduced.

Whenever a digital hardware is designed with some similar circuit blocks, it reveals a feature which is very attractive to an implementation in *Field Programmable Gate Array* (Wolf, 2004). Since all hardware neurons are digital circuits that are literally equal, the synthesis of the hardware layer in a FPGA is easily and best achieved.

In this work, a real number is represented as a fraction of integers (Santi-Jones and Gu, 2008). Floating-point representation based on the IEEE-754 standard (Tanenbaum, 2007) is not used here. Mathematic operations, such as sums and multiplications, using floating-point numbers require specific routines and, in terms of hardware design, a circuit that needs very large silicon area to be implemented. Also, long computing time is also demanded (Nedjah et al., 2008).

This work aims at optimizing arithmetic computation for compact yet efficient implementation of ANNs. A sum or multiplication of two fractions of integers can be split into simple operations operating over integer numbers. Integer-based operations are achieved using combinational circuits (Uyemura, 2002). Thus, a less-complex yet more efficient

hardware is used and a simple finite state machine would rule those integer operations.

A neuron weighted sum is a set of sums and multiplications of fractions. The activation function used in this work is the logistic sigmoid, which is approximated by quadratic polynomials. These are derived from a curve-fitting method: *least mean squares*. This is done, in contrast with using a *lookup table*, which is known to compromise the neuron rendered result. The quadratic polynomial-based approximation yields a far more precise result.

Provided that the sigmoid is approximated by second-degree polynomials, only sums and multiplications are needed to get the final result. Thereby, the circuit, once designed to compute weighted sums, can be reused for computing the sigmoid function, without extra effort or cost. This strategy spared unnecessary circuit, which lead to area extension and motivated by increasing the precision of the neuron rendered results (Martins et al., 2009).

This paper is organized as follows: First, in Section 3, we describe the data representation used in this design as well most arithmetic operations. Then, in Section 4, we describe the sigmoid computation. Subsequently, in Section 5, the overall hardware architecture and controllers are presented. Thereafter, in Section 5.1, The hardware neuron layer is depicted and discussed. Next, in Section 5.1.1, we show and comment on the neuron circuit design. After that, in Section 6, we report some simulation and synthesis result and discuss them. Finally, in Section 7, we draw some conclusion about the reported work and point out some future directions and improvements.

2 RELATED WORK

Research in the area of neural networks have been ongoing for over two decades now and hence the are many related work published. There are many work surveys published (Moerland and Fiesler, 1997; Lindsey and Lindblad, 1994; Rojas, 2010). In (Zhang and Pal, 2002.), the authors report on an efficient systolic implementation of ANNs. In (Kung, 1988; Kung and Hwang, 1989), the authors describe a novel scheme for designing special purpose systolic ring architectures to simulate feed forward stage is artificial neural networks. In (Kung, 1988), the authors present interesting results on implementing the back propagation algorithm on CMU Warp. In (Ferrucci, 1994), the author describes a multiple chip implementation of ANNs, using basic building blocs, such as multipliers and adders. In (Nedjah et al., 2009), the authors also take advantages of MAC (multiply and Accumulate) hard cores implemented into the fabrics of the FPGA to implement efficiently sums and products that are necessary in the ANN underlying computations. In (Beuchat et. al., 1998), the authors developed an FPGA platform, called RENCO - a REconfigurable Network COmputer. In (Bade and Hutchings, 1994; Nedjah and Mourelle, 2007), the authors report an implementation of stochastic neural networks based on FPGAs. Both implementations result in a very compact circuit. In (Zhang et. al., 1990), the author presents an efficient implementation of the back propagation algorithm on the connection machine CM-2. In (Botros and Abdul-Aziz, 1994), the author introduce a system for feed forward recall phase and implement it on an FPGA. In (Linde et. al., 1992), the authors describe REMAP which is an implementation of whole neural computer using only FPGAs. In (Gadea et. al., 2000), the authors report on a pipelined implementation of an on-line back propagation network using FPGA. In (Canas, et al., 2008), the authors propose a hardware implementation of ANNs, where the activation function is discretized and stored in a lookup table.

Many analog hardware implementations of ANNs have also been reported in the literature. In general, the implementation are very fast, dense and low-power when compared to digital ones, but they come along with precision, data storage, robustness and learning problems, as shown in (Holt and Baker, 1991; Nedjah et al., 2011; Choi et. al., 1996). It is an expensive and not flexible solution, as any ASIC (Montalvo et. al., 1997).

3 NUMERIC REPRESENTATION

A neural network operates with real numbers. Fixed–point representation implies a great accuracy loss. Floating–point notation (IEEE-745) offers good precision, but requires a considerable silicon area and a considerable time for arithmetic computing.

Searching for speed and circuit area trade-off, the alternative chosen to represent a real number was the *Fractional Fixed Point*, where a Fraction of integers is used to represent a real number (Santi-Jones and Gu, 2008). This model is depicted in Figure 1, showing the binary structure of a general fraction. This piece of data has 33 bits. 17^{th} bit of the least significant bits is for the algebraic sign of the fraction.

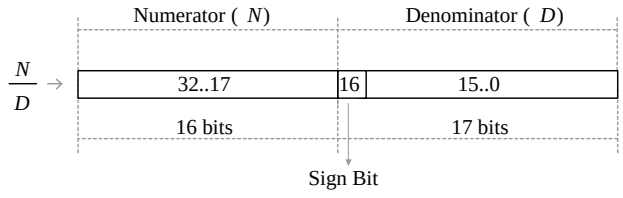

Figure 1: Binary representation of a fraction

A real float number converted to a fraction is shown in Eq. 1. Such fraction in its binary structure is also shown in Figure 2.

$$-0,003177124702144560 \mapsto \frac{12}{-3777} \quad (1)$$

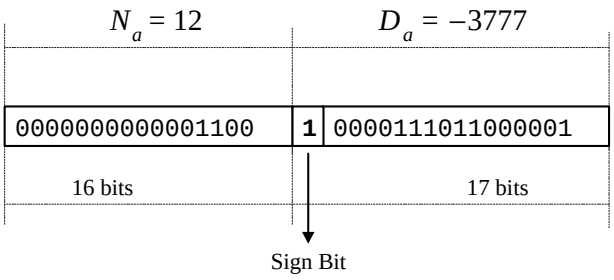

Figure 2: Example of number represented using a fraction

Regarding a and b real numbers, Equations 2, 3 and 4 display most of the arithmetic operations performed by the hardware. Neuron weighted sum and sigmoid computing are based on sums or subtractions and multiplications of fractions.

$$a + b \longmapsto \frac{N_a}{D_a} + \frac{N_b}{D_b} = \frac{N_a \times D_b + D_a \times N_b}{D_a \times D_b} \quad (2)$$

$$a - b \longmapsto \frac{N_a}{D_a} - \frac{N_b}{D_b} = \frac{N_a \times D_b - D_a \times N_b}{D_a \times D_b} \quad (3)$$

$$a \times b \longmapsto \frac{N_a}{D_a} \times \frac{N_b}{D_b} = \frac{N_a \times N_b}{D_a \times D_b} \quad (4)$$

There is an advantage of using fractions: a sum or a multiplication of two fractions is achieved through a mere sequence of integer operations, which require simple combinational circuits (Uyemura, 2002).

A sum of two fractions, for instance, demands 3 multiplications and 1 addition of integers: an unsophisticated finite–state machine is used to command the combinational computing sequence. Combinational Adder and multiplier provide attractive response timing and are easily allocated on FPGAs (Wolf, 2004).

3.1 Adaptive number framing technique

A fraction that results from a sum or product of two other fractions might require a bit range that exceeds the width of the binary structure of Figure 1. Repeated fraction operations would demand unlimited number of bits to keep up with the highest result precision. This is certainly impracticable. One possible and common solution is a *truncation*.

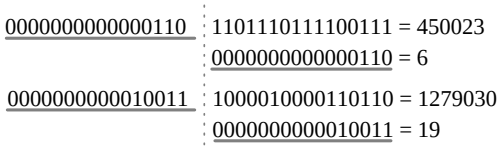

Figure 3: Direct framing

For instance, consider two fractions which fit into the binary structure of Figure 1, so that their multiplication results into the fraction $\frac{Num}{Den} = \frac{450023}{1279030}$. Neither the numerator nor the denominator would fit in the Figure 1 notation as $450023 > 65535$ and $1279030 > 65535$. Therefore, such fraction must be adjusted to fit into the binary limitation imposed in the hardware implementation.

The multiplication of two fractions (each one in the form of Figure 1) generates a fraction of 64 bits, in general, whose numerator and denominator are of 32 bits. The proposed hardware does not support this bit length and a framing technique must be thought of, aiming at minimizing the loss of accuracy.

An easy truncation or framing that could be done on $\frac{450023}{1279030}$ would be to take only the least–significant sixteen bits from numerator and from denominator – as Figure 3 displays – named *direct framing* (the simplest one).

The framing depicted in Figure 3, performed over fraction $\frac{450023}{1279030} = 0.3\mathbf{5}184\ldots$, yields $\frac{6}{19} = 0.3\mathbf{1}578\ldots$. This technique implies a high precision loss. In this work, an *adaptive framing technique* is used, where successive one-bit right-shifts of the binary representations of both the numerator and denominator, until the fraction under consideration is framed into the width of the representation of Figure 1, which is the fraction default binary length.

Algorithm 1 describes the steps of the proposed framing technique. It performs the adjustment of $\frac{Num}{Den}$ to produce $\frac{N}{D}$ that fits in the default structure used in this design. Note that Num is a natural of $\alpha > 16$ bits and $Den \neq 0$ is an integer of $\beta > 17$ bits. Recall that the MSB of Den is the fraction sign bit. Numerator N has 16 bits e denominator D has 17 bits; The bit MSB of D is the sign bit of the resulting fraction. Note that the largest number that can be represented is $\left\{-\frac{65535}{1}, \ldots, \frac{0}{1}, \cdots + \frac{65535}{1}\right\}$. Hence, when the framing operation reaches an all-zero denominator, the largest possible integer is used (see lines 10 and 11 of Algorithm 1.

Adaptive framing of the fraction $\frac{450023}{1279030}$ is illustrated in Figure 4. In this fraction, both the numerator and denominator go through five right shifts of one bit in order to frame the fraction within the binary structure of Figure 1, i.e. a numerator of 16 bits and denominator of 17 bits, including a sign

Algorithm 1 Framing technique

Require: α; β; $Num[\alpha-1..0]$; $Den[\beta-1..0]$;
Ensure: $N[15..0]$; $D[16..0]$;
1: $auxN[\alpha-1..0] \leftarrow Num[\alpha-1..0]$;
2: $auxD[\beta-2..0] \leftarrow Den[\beta-2..0]$;
3: **repeat**
4: **if** Or($auxN[(\alpha-1)..16]$) || Or($auxD[(\beta-2)..16]$) **then**
5: EndFraming $\leftarrow$ `false`;
6: RightShift $auxN[\alpha-1..0]$;
7: RightShift $auxD[\beta-2..0]$;
8: **if** Nor($auxD[\beta-2..0]$ **then**
9: EndFraming $\leftarrow$ `true`;
10: $auxN[15..0] \leftarrow 1111..11$; $//2^{16}-1$
11: $auxD[15..0] \leftarrow 0000..01$; $//1$
12: **end if**
13: **else**
14: EndFraming $\leftarrow$ `true`;
15: **end if**
16: **until** EndFraming;
17: $N[15..0] \leftarrow auxNum[15..0]$;
18: $D[15..0] \leftarrow auxDen[15..0]$;
19: $D[16] \leftarrow Den[\beta-1]$;
20: **return** $N[15..0], D[16..0]$

bit. This method is worthwhile because it minimize the loss of accuracy. Note that, fraction $\frac{450023}{1279030}$, which evaluates precisely to 0.3518471028, results in 0.3157894736 with a direct standard framing while when it yields $\frac{14063}{39969}$ using adaptive framing, which is equivalent to 0.3518476819. Note that an overall evaluation of precision loss throughout the computational process depends on the specific composition of the bits that are being shifted out from the numerator and denominator.

3.2 Precision of the adaptive framing

The loss of precision that is occasioned by a single iteration of the framing procedure described in Algorithm 1 depends on the the bit that is shifted out from the numerator and the corresponding in the denominator. Considering one framing iteration, there 4 possible cases as described below.

1. Both the numerator and denominator are even. i.e. $Num = 2\times N+0\times 2^0$ and $Den = 2\times D+0\times 2^0$. Thus, we have no loss of precision as explained in Equation 5, wherein E_{00} is the error introduced by right-shifting both the numerator and denominator:

$$E_{00} = \frac{2N}{2D} - \frac{N}{D} = 0 \tag{5}$$

450023	225011	112505	56252	28126	14063
1279030	639515	319757	159878	79939	39969

Figure 4: Adaptive framing

2. Both the numerator and denominator are odd. i.e. $Num = 2\times N+1\times 2^0$ and $Den = 2\times D+1\times 2^0$. Thus, we have a loss of precision as explained in Equation 6, wherein E_{11} is the error introduced by right-shifting both the numerator and denominator:

$$E_{11} = \frac{2N+1}{2D+1} - \frac{N}{D} = \frac{D-N}{D(2D+1)} \tag{6}$$

3. The numerator is even, i.e. $Num = 2\times N+0\times 2^0$ and denominator are odd, i.e. $Den = 2\times D+1\times 2^0$. Thus, we have a loss of precision as explained in Equation 7, wherein E_{01} is the error introduced by right-shifting both the numerator and denominator:

$$E_{01} = \frac{2N}{2D+1} - \frac{N}{D} = \frac{-N}{D(2D+1)} \tag{7}$$

4. The numerator is odd, i.e. $Num = 2\times N+1\times 2^0$ and denominator are even, i.e. $Den = 2\times D+0\times 2^0$. Thus, we have a loss of precision as explained in Equation 8, wherein E_{10} is the error introduced by right-shifting both the numerator and denominator:

$$E_{10} = \frac{2N+1}{2D} - \frac{N}{D} = \frac{1}{2D} \tag{8}$$

Therefore, assuming that it is 0s and 1s are evenly distributed in a binary representation, the average error introduced by a single shift can be evaluated as shown in Equation 9.

$$E_{avg} = \frac{4D-4N+1}{8D(2D+1)} \tag{9}$$

The proposed adaptive framing technique provides an adequate accuracy for the purpose of this work. The proposed hardware has to be equipped with a right-shifter to perform fraction adjustment. A one-bit right-shift is an integer-divide-by-2 operator. This property is also useful in the sigmoid computation as the strategy used for computing the sigmoid was forged in order to take advantage of the shifter already included in the hardware. This will be explained in the next section.

4 ACTIVATION FUNCTION

There are many functions, commonly used, which map neuron output. The most common are functions *ramp* as described in Equation 10, hyperbolic tangent as defined in

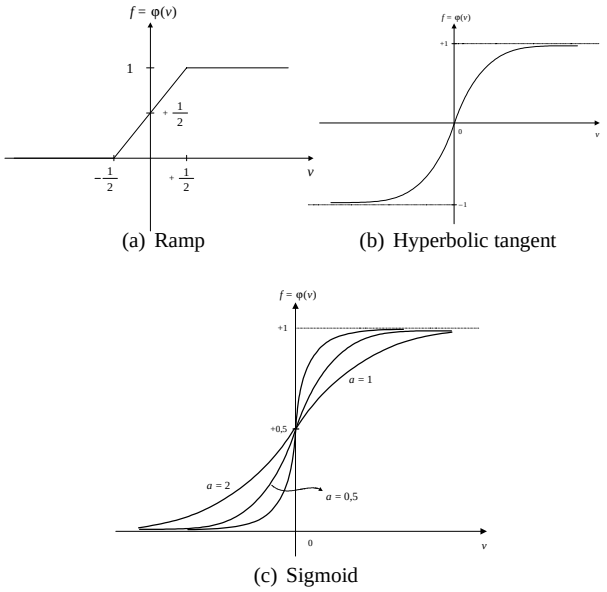

Figure 5: Common activation functions

Equation 11 and sigmoid as described in Equation 12. The curves of these three activation functions are shown in Figure 5.

$$y = \varphi(v) = \begin{cases} 1 & \text{if } v \geq b \\ \frac{1}{b-a}v - \frac{a}{b-a} & \text{if } a < v < b \\ 0 & \text{if } v \leq a \end{cases} \quad (10)$$

$$\varphi(v) = b \cdot \frac{e^{av} - e^{-av}}{e^{av} + e^{-av}}, \quad (11)$$

wherein $a \neq 0$ and $b \neq 0$.

$$\varphi(v) = \frac{1}{1 + e^{-av}} \quad (12)$$

The sigmoid is widely used in multilayer-perceptron neural networks (Haykin, 1999). The hardware, presented in this paper, computes sigmoid of Equation 13, where parameter a is set to 1 and v is the neuron weighted sum (including bias). Initially, through least mean squares, 3 quadratic polynomials are obtained, which fit into the curve e^{-v}. Each polynomial approximates e^{-v} in a certain range of the domain v, as Figure 6 and Equation 14.

$$\varphi(v) = \frac{1}{1 + e^{-v}} \quad (13)$$

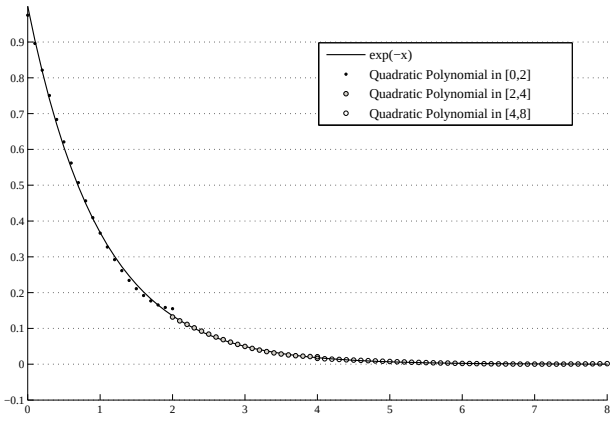

Figure 6: Curve–fitting: quadratic polynomials and e^{-v}, for $v \geq 0$

$$\exp(-v) \cong \begin{cases} P_{00}(v) & \text{if } v \in [0, 2[\\ P_{01}(v) & \text{if } v \in [2, 4[\\ P_{10}(v) & \text{if } v \in [4, 8[\\ P_{11}(v) = 0 & \text{if } v \in [8, +\infty[\end{cases} \quad (14)$$

In Figure 6, the approximation method generates the quadratic polynomial of (15) for $\exp(-v)$, wherein $F_{[x,y[}(v)$ is a fractional function in $v \in [x, y[$:

$$\begin{cases} f_{[0,2[}(v) = 0.1987234v^2 - 0.8072780v + 0.9748092 \\ f_{[2,4[}(v) = 0.0268943v^2 - 0.2168304v + 0.4580097 \\ f_{[4,8[}(v) = 0.0016564v^2 - 0.0235651v + 0.0840553 \\ f_{[8,+\infty[}(v) = 0 \end{cases} \quad (15)$$

We know that the ANN hardware operates with fractions, whose representation is depicted in Figure 1. So, it will be better to express the polynomial of (15) in the hardware rep-

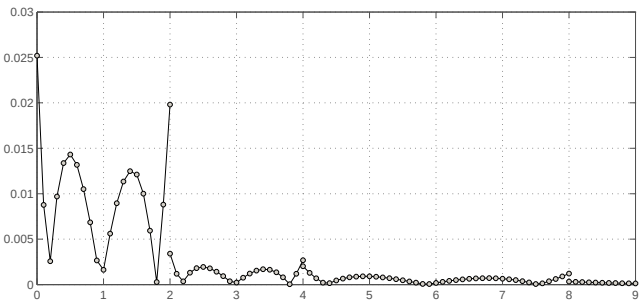

Figure 7: Error introduced by the activation function approximation

resentation. This is shown in (16):

$$\begin{cases} f_{[0,2[}(v) & \approx \frac{12858}{64703}\left(\frac{N_v}{D_v}\right)^2 + \frac{11691}{-14482}\frac{N_v}{D_v} + \frac{56072}{57521} \\ f_{[2,4[}(v) & \approx \frac{1046}{38893}\left(\frac{N_v}{D_v}\right)^2 + \frac{13883}{-64027}\left(\frac{N_v}{D_v}\right) + \frac{12560}{27423} \\ f_{[4,8[}(v) & \approx \frac{63}{38032}\left(\frac{N_v}{D_v}\right)^2 + \frac{581}{-24655}\left(\frac{N_v}{D_v}\right) + \frac{456}{5425} \\ f_{[8,+\infty[}(v) & \approx \frac{0}{1} \end{cases} \quad (16)$$

As an example, $P_{01}(v)$ refers to the polynomial which fits into e^{-v} for $v \in [2,4[$. This hardware deals only with binary structures which represent fractions. Thus, $P_{01}(v)$ is best expressed in the following form: $P_{01}(\frac{N_v}{D_v}) = \frac{N_v}{D_v}\left[\frac{1046}{38893}\left(\frac{N_v}{D_v}\right) + \frac{13883}{-64027}\right] + \frac{12560}{27423}$. Weighted sum parameter is in evidence to save 1 multiplication.

Using polynomials, e^{-v} is computed performing only multiplications and sums of real numbers – which are the basic operations of neuron weighted sum. So, weighted sum digital circuit is reused to compute e^{-v}.

The achieved precision is proportional to the highest degree of the exploited polynomial. Nevertheless, with higher degree polynomials, the required computation becomes more complex and thus the response time becomes longer. A second degree polynomial provides reasonable accuracy (on each range of Equation 14) and yet does not slow down the hardware. The error imposed by this approximation was plotted and the result is shown in Figure 7.

Domain ranges, in Equation 14, were chosen based on based on the fact that borderline values of each range are powers of 2. A right-shifter (discussed previously) is used to frame a fraction into the binary structure of Figure 1. During activation function computation, the same right-shift register is reused in the selection of the adequate polynomial to compute e^{-v}, taking into account a certain range of $v \geq 0$. In order to explain the polynomial selection procedure, non–negative weighted sums are considered initially: $v \geq 0$.

Let $\frac{N_v}{D_v}$ be the fraction notation of a neuron weighted sum. The hardware first checks $\frac{N_v}{D_v} < 2$. This comparison is equivalent to $\frac{N_v}{2} < D_v$, since $\frac{N_v}{D_v}$ is non–negative. The one-bit right-shifter is responsible for performing N_v div 2 and a combinational comparator provides the boolean result of N_v div $2 < D_v$.

If N_v is odd, then N_v div $2 \neq \frac{N_v}{2}$. Nonetheless, Theorem 2 ensures that when we have N_v div $2 < D_v$ then it immediately follows that $\frac{N_v}{2} < D_v$ and vice-verça.

If N_v div $2 < D_v$ is valid, then P_{00} is the selected polynomial, because the weighted sum $v \in [0,2[$. Otherwise, the hardware checks $\frac{N_v}{D_v} < 4$, which is has the same results of comparison $\frac{N_v}{4} < D_v$. Two right-shifts are required to get N_v div 4 and another comparison is done: N_v div $4 < D_v$. Theorem 2 still ensures that N_v div $4 < D_v \Leftrightarrow \frac{N_v}{4} < D_v$.

If N_v div $4 < D_v$ is valid, then P_{01} is the selected polynomial as the weighted sum $v \in [2,4[$. Otherwise, other shifts are performed until the adequate polynomial is reached, taking into account the corresponding range of v.

For the sake of clarity, before proving Theorem 2, we first prove Theorem 1, which establish that for P odd, we have P div $2 = \frac{P}{2} - 0,5 < Q$ is equivalent to $\frac{P}{2} < Q$.

Theorem 1 $\forall P, Q \in \aleph^*$, *wherein* P *is odd, we have* P *div* $2 < Q \Leftrightarrow \frac{P}{2} < Q$.

Proof: P is odd, then it follows that P div $2 = \frac{P}{2} - \frac{1}{2}$. Therefore, P div $2 < Q$ is equivalent to $\frac{P}{2} - \frac{1}{2} < Q$ as P div $2 = \frac{P}{2} - 0.5$.

$$P \text{ div } 2 = \frac{P}{2} - \frac{1}{2} < Q \Leftrightarrow \frac{P}{2} < Q + \frac{1}{2} \Leftrightarrow P < 2Q + 1 \quad (17)$$

$$\frac{P}{2} < Q \Leftrightarrow P < 2Q \quad (18)$$

Considering Equation 17 and Equation 18, we need to prove that, for P odd, we always have $P < 2Q + 1 \Leftrightarrow P < 2Q$. Assuming $Q \neq 0$, we need to prove the following: $P < 2Q + 1 \rightarrow P < 2Q$ and $P < 2Q \rightarrow P < 2Q + 1$.

1. $P < 2Q + 1 \rightarrow P < 2Q$: $Q \neq 0$, then $2Q + 1$ is odd. For $Q = 1$, we have $P < 3 \rightarrow P < 2$. As P is odd, then $P < 3$ ensures that $P < 2$, as the largest odd integer smaller than 3 is $P = 1$. For $Q > 1$, the largest odd integer smaller than $2Q + 1$ is $P = 2Q + 1 - 2$.

Hence, for $P = 2Q + 1 - 2 = 2Q - 1$, we have $P = 2Q - 1 < 2Q$. Finally, for any P such that $P < 2Q - 1$, we have $P < 2Q$, as $P < 2Q - 1 < 2Q$. Therefore, $P < 2Q + 1 \rightarrow P < 2Q$ holds.

2. $P < 2Q \rightarrow P < 2Q + 1$: It is clear that, if $P < 2Q$, then $P < 2Q + 1$, as $P < 2Q < 2Q + 1$. This proves that $P < 2Q \rightarrow P < 2Q + 1$ holds.

□

Theorem 2 *$\forall\ P, Q \in \aleph^*$, we have P div $2^s < Q \Leftrightarrow \frac{P}{2^s} < Q$, wherein $s \in \aleph^*$.*

Proof: The result of P div 2^s can be formulated in terms of $\frac{P}{2^s}$ as shown in Equation 19, wherein P mod 2^s is the remainder of the integer division of P by 2^s.

$$P \text{ div } 2^s = \frac{P}{2^s} - \frac{P \bmod 2^s}{2^s} \tag{19}$$

1. For $s = 1$, Equation 19 reduces to $P \text{ div } 2 = \frac{P}{2} - \frac{P \bmod 2}{2}$. In this case, for P even, we have P div $2 = \frac{P}{2} - \frac{0}{2} = \frac{P}{2}$ and, thus, P div $2 = \frac{P}{2} < Q \Leftrightarrow \frac{P}{2} < Q$, $\forall P, Q \in \aleph^*$. For P odd, Theorem 1 ensures that P div $2 = \frac{P}{2} - \frac{1}{2} < Q \Leftrightarrow \frac{P}{2} < Q$; $\forall P, Q \in \aleph^*$.

2. For $s > 1$, we need to prove that, $\forall P, Q, s \in \aleph^*$, wherein $s > 1$, Equation 20 holds.

$$P \text{ div } 2^s = \frac{P}{2^s} - \frac{P \bmod 2^s}{2^s} < Q \Leftrightarrow \frac{P}{2^s} < Q \tag{20}$$

Using P div $2^s = \frac{P}{2^s} - \frac{P \bmod 2^s}{2^s} = \gamma$, where $\gamma \in \aleph$, we have (21).

$$P = 2^s\gamma + P \bmod 2^s \tag{21}$$

Comparing (21) to (20), The equivalence of Equation 20 can be expressed as in Equation 22.

$$\gamma < Q \Leftrightarrow \frac{2^s\gamma + P \bmod 2^s}{2^s} < Q \tag{22}$$

$\forall P, Q, s \in \aleph^*$, wherein $s > 1$ and $\gamma \in \aleph$. We know that $P \bmod 2^s \in \{0, 1, 2, \ldots, 2^s - 1\}$, i.e. $0 \leq P \bmod 2^s < 2^s$. Using $\delta = P \bmod 2^s$, we have $\delta \in \aleph$ and $0 \leq \delta < 2^s$. Replacing δ in Equation 22, we then have Equation 23.

$$\gamma < Q \Leftrightarrow \frac{2^s\gamma + \delta}{2^s} < Q \tag{23}$$

$\forall P, Q, s \in \aleph^*$, where $s > 1$, $\gamma \in \aleph$ and $\{\delta \in \aleph \mid 0 \leq \delta = P \bmod 2^s < 2^s\}$.

Observing the comparison operations in Equation 23, we get to Equation 24 and Equation 25.

$$\gamma < Q \Leftrightarrow Q - \gamma > 0 \tag{24}$$

$$\frac{2^s\gamma + \delta}{2^s} < Q \Leftrightarrow \delta < 2^s(Q - \gamma) \Leftrightarrow Q - \gamma > \frac{\delta}{2^s} \tag{25}$$

Based on Equation 24 and 25, the required proof (equivalence of Equation 23) would be completed if the propositions $Q - \gamma > 0 \rightarrow Q - \gamma > \frac{\delta}{2^s}$ and $Q - \gamma > \frac{\delta}{2^s} \rightarrow Q - \gamma > 0$ hold; That is, with the premisses $\forall P, Q, s \in \aleph^*$, where $s > 1$, $\gamma \in \aleph$ and $\{\delta \in \aleph \mid 0 \leq \delta = P \bmod 2^s < 2^s\}$.

(a) $Q - \gamma > 0 \rightarrow Q - \gamma > \frac{\delta}{2^s}$: Q is a non-zero natural and γ is natural. So, $Q - \gamma$ is an integer. As δ is a natural such that $0 \leq \delta < 2^s$, there follows $0 \leq \frac{\delta}{2^s} < 1$. If $Q - \gamma$ is positive, then $Q - \gamma \geq 1$ and, therefore $Q - \gamma > \frac{\delta}{2^s}$, since $0 \leq \frac{\delta}{2^s} < 1$. So, $Q - \gamma > 0$ ensures that $Q - \gamma > \frac{\delta}{2^s}$, i.e $Q - \gamma > 0 \rightarrow Q - \gamma > \frac{\delta}{2^s}$ holds.

(b) $Q - \gamma > \frac{\delta}{2^s} \rightarrow Q - \gamma > 0$: Assuming $0 \leq \frac{\delta}{2^s} < 1$, it is clear that $Q - \gamma > \frac{\delta}{2^s}$ ensures $Q - \gamma > 0$. Any integer $Q - \gamma$ larger than $\{0, 1, 2, \ldots, 2^s - 1\}$ is with no doubt positive ($Q - \gamma > 0$). There follows that $Q - \gamma > \frac{\delta}{2^s} \rightarrow Q - \gamma > 0$ holds.

□

Once the suitable polynomial is selected from Equation 14, the hardware computes it and returns the resulting fraction $\frac{N_{fe}}{D_{fe}}$, which represents the fraction value of e^{-v}, for $v \geq 0$. Thus, $e^{-v} \cong \frac{N_{fe}}{D_{fe}}$, where $v \geq 0$. Replacing $\frac{N_{fe}}{D_{fe}}$ in the sigmoid (from Equation 13), it easily comes to Equation 27.

$$\varphi(v) \cong \frac{1}{1 + \frac{N_{fe}}{D_{fe}}} \quad \text{if } v \geq 0 \tag{26}$$

$$\varphi(v) \cong \frac{D_{f_e}}{D_{f_e} + N_{f_e}} \quad \text{if } v \geq 0 \tag{27}$$

Whenever the weighted sum is negative, $v < 0$, a sigmoid property can be used: $\varphi(v) = 1 - \varphi(-v)$, $v \in \Re$. This way, through Equation 27, $\varphi(v)$ is finally solved for $v < 0$, as Equation 28 and Equation 29 show.

$$\varphi(v) \cong 1 - \frac{D_{f_e}}{D_{f_e} + N_{f_e}} \text{ if } v < 0 \tag{28}$$

$$\varphi(v) \cong \frac{N_{f_e}}{D_{f_e} + N_{f_e}} \text{ if } v < 0 \tag{29}$$

Note that using the same approximation, we can exploit any of the commonly used activation functions that are based

on the exponential function, such as hyperbolic tangent, described earlier. The nice properties of the exponential function would be taken advantage of so as to reduce the necessary overall computation. The ramp function can also be easily used. It does not require any approximation. The underlying computation requires a multiplication followed by an addition, in the general case. Two comparisons are needed to determine whether these this computation is necessary. Otherwise, either constants 0 or 1 are used instead. In order to accommodate a new activation function, the controlling sequence of the stage within the hardware must be slightly readjusted.

5 HARDWARE ARCHITECTURE

The proposed architecture consists of two subsystems: the load and control system (LCS) and the ANN computing hardware (ANNCH). Component LCS loads and stores the data needed for an neural network application. The ANNCH includes the digital circuit that implement the neuron's hardware, which include logic and arithmetic computations as well as the underlying control flow. The block diagram of the overall hardware is depicted in Figure 8.

The load and control system LCS, in Figure 8, includes three memories: one for ANN inputs, another for weights and biases and a third memory, for the polynomial coefficients that allow to compute the neuron activation function. Since the hardware is able to adapt it-self to different MLP topologies, the number of inputs, weights and biases may be altered *on–the–fly*, enabling the hardware to perform different ANN applications.

The hardware synthesis in FPGA requires the exact sizing of the LCS memories. For instance, weight (and bias) memory is sized as Equation 30.

$$(i_{max}+1)n + n(n+1)(l_{max}-1) \qquad (30)$$

where i_{max} is the maximum number of inputs, n is the maximum number of neurons per layer that the hardware is able to support. This is actually the number of neurons in the physical layer. Parameter l_{max} is the maximum number of layers the actual ANN configuration can include so as to be implemented in the proposed hardware. Activation function memory stores nine coefficients: three for each polynomial. Note that, because of this parameter modeling, the proposed ANN hardware can accommodate any ANN topology that exploits at most c_max neurons in any of its layers. The sole modification that is required for different topologies consists of the size of the three data memories managed by component LCS.

LCS also controls the ANN application operation in ANNCH. A 33–bit data bus enables the necessary data flow to the ANNCH unit. A control bus is also available and establishes the communication between LCS and ANNCH. The LCS is the master controller the data bus. It sends the required data upon ANNCH's requests.

In Figure 8, ANNALU is the ANNCH arithmetic and logic unit and includes the digital neurons, whereby the weighted sums and sigmoid are computed. Neurons within the same layer of such an ANN application are performed by the hardware in parallel.

The control unit, ANNCU in Figure 8, commands the digital neuron arithmetics only. This is executed by ANNALU and includes the weighted sum and activation function computation. A clock generator synchronizes the communication between ANNCU and ANNALU. The LCS as master triggers ANNCU, and the latter leads the whole computation corresponding to the current layer until neuron outputs are ready. Afterwards, LCS restarts ANNCU to compute another ANN application layer – this process goes on until the neural network outputs are available on y*i* data buses. In the following, i.e. Section 5.1 and Section 5.2, we describe the architecture and operation of ANNALU and ANNCU respectively.

5.1 Hardware layer: ANNALU

As mentioned previously, the hardware architecture is designed with only one physical layer. This is a set of digital hardware neurons that work in parallel and define the ANNALU, as shown in Figure 9. Whenever an ANN is performed, the physical layer is reused for computing all the layers of the application neural network. If the hardware layer has n_{max} neurons, so the number of neurons in all the ANN layers must not exceed n_{max}.

For instance, assuming that the k^{th} layer of such an ANN application has 3 neurons, the LCS activates only hardware neurons 1, 2 and 3 of Figure 9, to compute the k^{th} layer. Next, when the $(k+1)^{th}$ is going to be computed, outputs y1, y2 and y3 from neurons 1, 2 and 3, respectively, are fed back through registers Regy*i*, buffers and multiplexers. For instance, assuming that layer $(k+1)^{th}$ has two neurons, then only hardware neurons 1 and 2 would be switched on. Still in Figure 9, neural network inputs flow via the same data bus and are sent to digital neurons through x1, x2, ..., x*n*. Registers Regw*i* stores weights and biases, also provided by LCS via data bus.

When computing a certain ANN application layer, the input is shared with all hardware neurons, but each neuron has its own weight (for the same input), as Figure 10 presents – for a three-neuron-layer example. First product of Figure 10 is computed, at the same time, for all hardware neurons, and

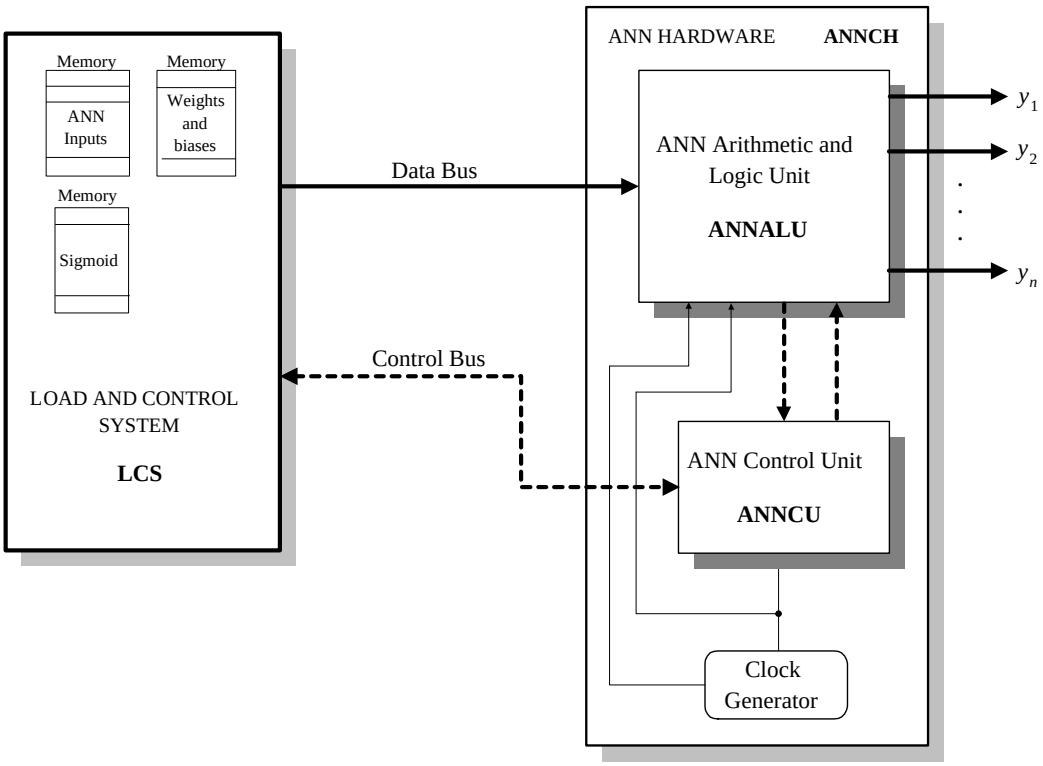

Figure 8: Overall hardware architecture

this happens to the whole weighted sum and also to the activation function computing.

5.1.1 Hardware neuron model

The neuron architecture is illustrated in Figure 11. As seen previously, the weighted sum and sigmoid computing require only sums and multiplications of fractions. These operations, in turn, consist of other simpler operations: sums and multiplications of integers, which are performed by the combinational circuits MULTIPLIER and ADDER, respectively, as Figure 11 shows.

The two shift registers (ShiftReg1 and ShiftReg2) are included so as to adjust the fraction, that results from a multiplication of two other fractions. This adjustment refers to the *adaptive framing* explained in Section 3.1. ShiftReg1 shifts the numerator while ShiftReg2 shifts the denominator of such a fraction.

In Figure 11, ShiftReg3 is another shift-register that shifts the fraction numerator which results from an addition of two other fractions. ShiftReg2 is also used for shifting the denominator of such a fraction. The neuron weighted sum is accumulated in ShiftReg3 used for the numerator and Reg4 for the denominator.

The combinational circuits TwosComple1 and Twoscomple2, when needed, perform the two's complement (Tanenbaum, 2007) on signed integers stored in ShiftReg1 and ShiftReg2, respectively. During an addition of two fractions, there is an sum or subtraction of two signed integers: one stored in ShiftReg1 and the other in ShiftReg2.

A combinational Comparator is necessary to select the adequate polynomial for computing sigmoid function, as explained in Section 4. The arithmetic signals of X_i and W_i are processed by component ASPU (Arithmetic Signal Processing Unit) of Figure 12) in order predict the signal of the fraction obtained from multiplying or adding two given fractions. The ASPU also decides if an Adder operand will go through the two's complement (TwoCompl1 and/or TwoCompl2 of Figure 11). The Adder component is responsible for the additions of the neuron's weighted sum. RShiftReg3 and Reg4 are assigned to accumulate the weighted sum as a fraction: numerator in RShiftReg3 and denominator in Reg4.

In Figure 12, component NTC (Negative transition Controller) releases the negative transition of Clk1 to the Flip-Flop D, whenever signal E is set. ASPU works in parallel with multiplications and additions of fractions during weighted sum and activation function computation. All Neurons of Figure 9 work in parallel. When the computation of

$$\text{Neuron (1):}\quad x_1w_1 + x_2w_2 + 1\cdot 0 = v_1 \xrightarrow{f_A(\cdot)} f_A(v_1) = y_1$$

$$\text{Neuron (2):}\quad x_1w_3 + x_2w_4 + 1\cdot w_0 = v_2 \xrightarrow{f_A(\cdot)} f_A(v_2) = y_2$$

$$\text{Neuron (3):}\quad x_1w_5 + x_2w_6 + 1\cdot 0 = v_3 \xrightarrow{f_A(\cdot)} f_A(v_3) = y_3$$

1º Prod. 2º Prod. 3º Prod.

Figure 10: Arithmetics of a three-neuron layer

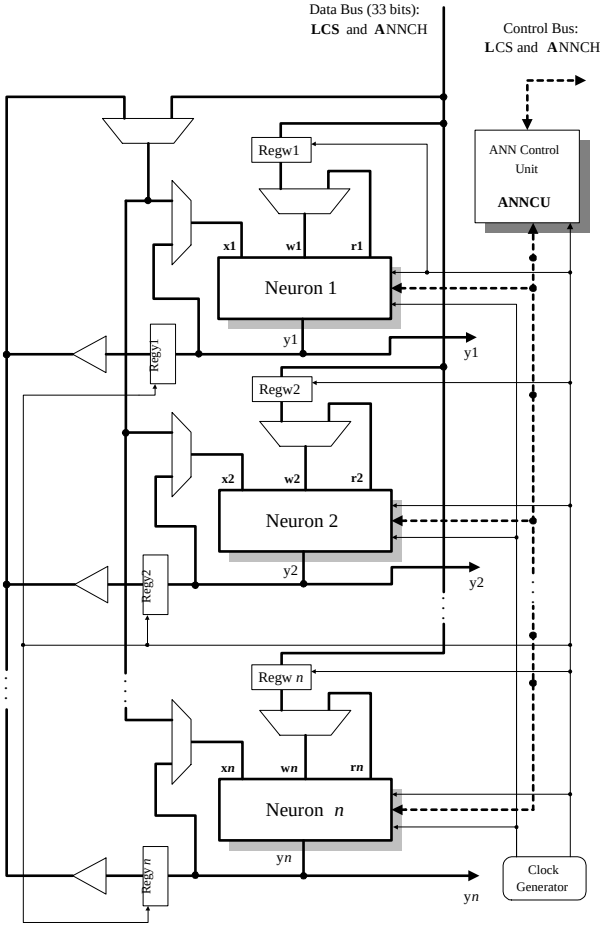

Figure 9: Hardware layer: ANNALU

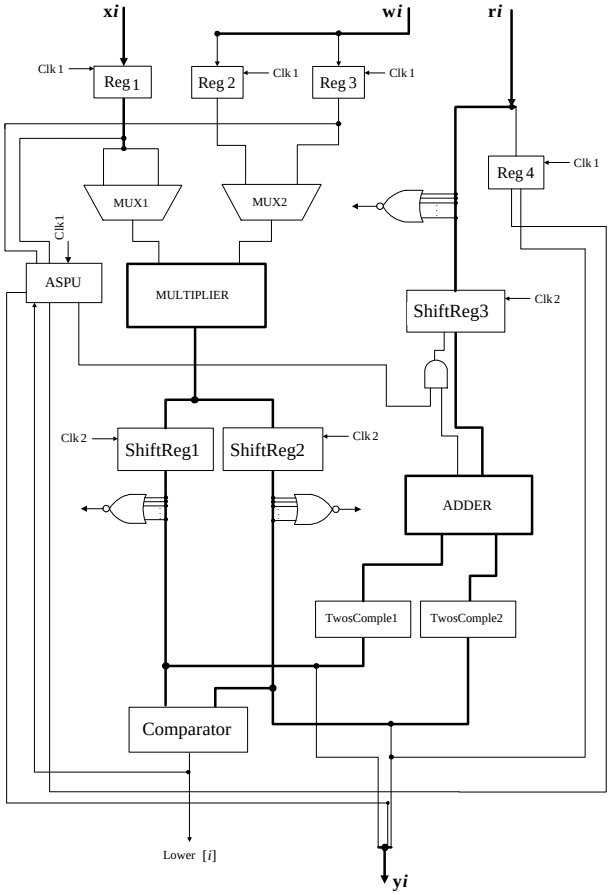

Figure 11: Neuron architecture

a given layer is being performed, the one related to the next layer is being initialized. As soon as the computation of all

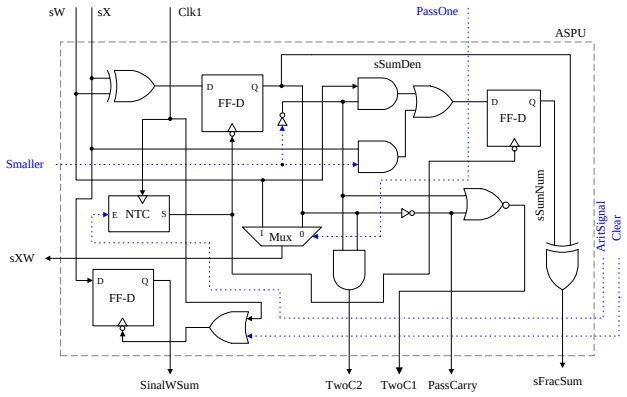

Figure 12: Arithmetic signal processing unit to predict the fraction signal obtained from a sum or product of two other fractions

layers has been completed, ANNALU sends a signal to the ANNCU, informing that the output of the Neural Network is available. So, the latter informs the software sub-system that the whole computation is done.

During weighted sum, for instance, register Reg1 is used for storing a neuron input and registers Reg1 and Reg2 store, respectively, numerator and denominator of the synaptic weight related to that input. Figure 11 displays fraction sign bit from Reg1 (input data) and fraction sign bit from Reg3 (denominator of a weight) injected to the ASPU unit.

5.2 Control Unit: ANNCU

A general view of the control unit ANNCU is depicted in Figure 13. Processing an application neural network starts by obtaining the required data via the control block DI. The finite state machine (FSM) within this block is responsible for requesting to component LCS the inputs x_i of the net. These data are then stored into specific registers in ANNALU. The state machine within block WSC controls the computation of the weighted sum through fraction sums and products using the provided components in the neuron hardware described earlier.

Once the weighted sum yielded, the state machine within block AFC initiates the control of the same components available in the hardware layer so as to compute the activation functions of the neurons. Once this is done, the computation due to the current neuron layer has been achieved. If the current layer is not the last in the net, then the FSM implemented in block AFC sends the control to that implemented by block DI to iterate the process once more. Otherwise, the output results are ready and therefore, the control unit enters state `End` and waits for a reset trigger.

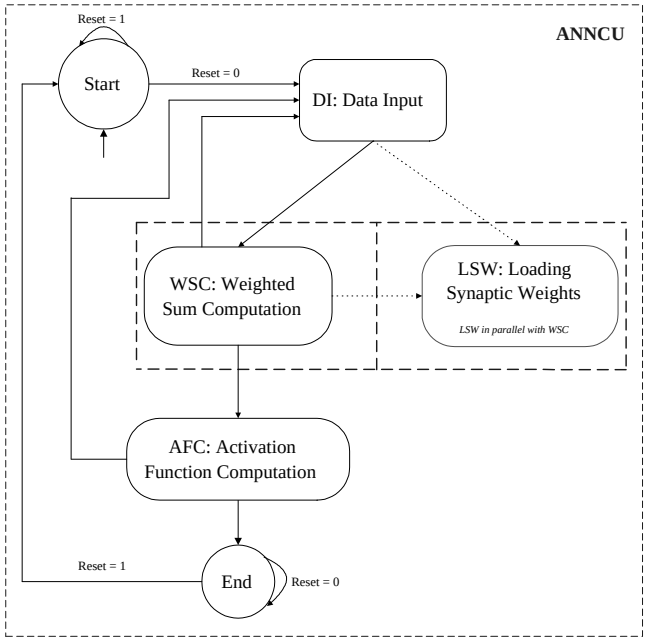

Figure 13: Control flow within ANNCU

As shown in Figure 13, the control unit ANNCU includes two FSMs: primary and secondary. The primary FSM consist of blocks DI, WSC, AFC as well as states `start` and `End`. The secondary FSM is defined solely by the control within block LSW. which is responsible for preparing the synaptic weights for the next layer, if any, while and in parallel with the computation of the weighted sums due to the current layer. The current layer synaptic weights are stored in registers within the neuron hardware, allowing for the kind of parallelism. (Details of the FSMs description can be found in (Martins, 1996). These were not included here as this kind of detailed description is not necessary for understanding the overall behavior of the control unit.)

5.3 Clock generator

The macro architecture of Figure 8 include a clock generator, which yields two signal Clk1 and Clk2. Both these signals implement the synchronization of the general activities of the ANNCH, which also interferes in both included units: ANNALU and ANNCU. The time diagram of the clock signals is shown in Figure 14.

Component ANNCU consists mainly of a state machine the controls the operations performed by ANNALU. The state transitions in this machine occur during the positive transitions of clock signal Clk1. The time period t_{H1} is the required time for the stabilization of the output signals. The right-shift register used in the neuron micro-architecture op-

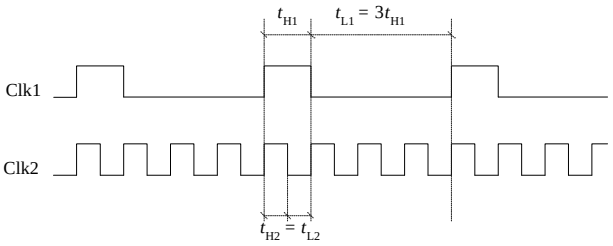

Figure 14: Clock signals used by the hardware sub-system ANNCH

erates in synchrony with the negative transition fo clock signal Clk2. Note that during one clock cycle with respect to Clk1, four shifting operation may take place. This accelerates the the time spent in shifting operations. Shifting operations are required so as to frame the results of additions and multiplications within the fractional data representation. The use of faster clock Clk2 is only possible thanks to the reduced time of a shifting operation with respect to that required by the more complex operations performed by ANNALU.

6 PERFORMANCE RESULTS

The architecture is entirely described in VHDL. It was simulated in ModelSim XE 6.3c. The detailed VHDL code can be found in (Martins, 1996). Simulation snapshots accompanied by detailed explanations of the computations done throughout a layer of the ANN hardware are given in (Martins, 1996).

The VHDL specification of the ANN proposed hardware was synthesized for the Xilinx Virtex-5 XC5VFX70T FPGA, through Xilinx XST Synthesis tool (ISE Design Suite 11.1). This synthesis tools allows for integration of software/hardware co-designs. The ANN hardware sub-system was implemented through automatic synthesis and the software sub-system was implemented in C language and executed by a MicroBlaze processor. The MicroBlaze (Xilinx, 2008) is a RISC microprocessor IP from Xilinx ^{TM}, which can be synthesized in reconfigurable devices. Although we had access to an embedded PowerPC core, only the MicroBlaze offers a low latency point-to-point communication link, known as *Fast Simplex Link* (FSL) (Xilinx, 2009), which allows the connection of a component, identified as co-processor, to the microprocessor. Therefore, the MicroBlaze is connected to the ANN hardware co-processor through an FSL channel and provides the necessary input data, as shown in Figure 15. The C program, executed by the microBlaze, provides the parameter and coefficient settings for the LCS three memories via the *from-microBlaze* FSL FIFO, then after a while receives the results, which are sent by the co-processor via the *to-microBlaze* FSL FIFO. The C program is also responsible of printing the received results on the terminal via the available UART (*Universal Asynchronous Receiver/Transmitter*). The timer is used to measure the number of elapsed cycle during the ANN hardware operation.

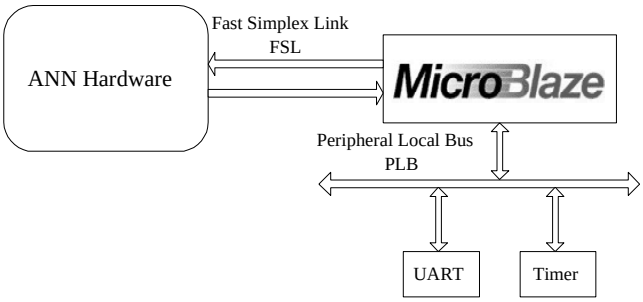

Figure 15: The MicroBlaze processor connected to the ANN hardware co-processor

In Table 1, we report the hardware area required by a single neuron for different numbers of inputs, as well as that needed to implement to whole network, considering the maximum number of inputs allowed (i_{max}), the maximum number of neuron per layer (n_{max}) and the maximum number of layers (l_{max}). The required area is given in terms of slices. Virtex-5 FPGA slices are organized differently from previous generations. Each Virtex-5 FPGA slice contains four 6-input lookup tables and four flip-flops. We compare these figures to those imposed by the binary-radix straightforward design and the stochastic computing-based design (Nedjah and Mourelle, 2007) and a previous MAC-based design, reported in (Nedjah et al., 2009).

The used FPGA has 11,200 slices. Therefore, a rough approximations, given the data presented in Table 1, we can expect possible implementations of ANNs of a hundreds of neurons per layer for virtually infinite number of layers. However, only the actual mapping of the hardware with such number of neurons would provide exact figures, as the required area depends on the how the synthesis tool would optimize the hardware resources via sharing *vs.* duplication.

In Table 2, we show the net delay imposed in comparison with those imposed by the implementations reported in (Nedjah and Mourelle, 2007) and (Nedjah et al., 2009).

From the performance results given in Table 1 and Table 2, it can be observed that in the proposed architecture both the area and computation time are reduced and so the performance factor, which is defined as $\frac{1}{area \times time}$, was improved, as depicted in the chart of Figure 16. The comparison indicates that the trade-off between area and time is improved for the proposed implementation as the net size increases.

Table 1: Area requirements of one neuron and network for different number of inputs and net size

i_{max}	n_{max}	l_{max}	Neuron area (#Slices)				Net area (#Slices)			
			BIN[1]	STO[2]	MAC[3]	FFP[4]	BIN[1]	STO[2]	MAC[3]	FFP[4]
2	6	3	116	8	4	6	98	21	8	11
4	9	5	212	12	8	9	574	75	25	41
8	13	7	436	20	11	15	780	421	57	129

[1]Binary-radix based (Nedjah and Mourelle, 2007) [2]Stochastic (Nedjah and Mourelle, 2007)
[3]MAC-based fbating-point (Nedjah et al., 2009) [4]Fraction-based proposed

Table 2: Time delay of a network and performance factor for different number of inputs and net size

i_{max}	n_{max}	l_{max}	Net delay (ns)				Performance factor ($\times 10^{-3}$)			
			BIN[1]	STO[2]	MAC[3]	FFP[4]	BIN[1]	STO[2]	MAC[3]	FFP[4]
2	6	3	3.45	5.85	3.67	3.33	2.498751	21.3675	68.1198	53.2481
4	9	5	4.92	7.09	5.11	4.71	0.958736	11.7536	24.461	28.1293
8	13	7	11.32	19.87	11.79	9.05	0.202613	2.5163	8.4817	13.3779

[1]Binary-radix based (Nedjah and Mourelle, 2007) [2]Stochastic (Nedjah and Mourelle, 2007)
[3]MAC-based fbating-point (Nedjah et al., 2009) [4]Fraction-based proposed

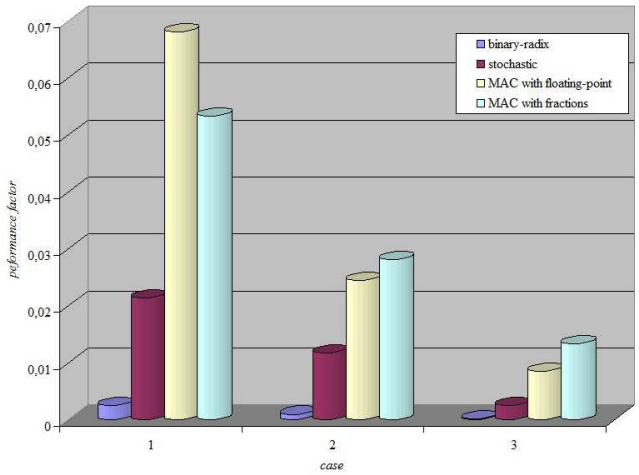

Figure 16: Comparison of the performance factor yield by the neural network hardware proposed here and those reported in the literature

For a testbed application, we use the MLP that implements word recognition of a given speech. The neural network requires 220 data as input nodes and returns 10 results as output nodes. The network input consists of 10 vectors of 22 components obtained after preprocessing the speech signal. The output nodes correspond to 10 recognizable words extracted from a multi-speaker database (Waibel et. al., 1989). After testing different architectures (Canas, et al., 2003), the best classification results, which achieved a 96.83% of correct classification rate in a speaker-independent scheme, have been yielded using 24 nodes in a single hidden layer, with full feed forward connectivity. Thus the MLP has two layers of 24 and 10 neurons respectively, as shown in Figure 17, which was taken from (Canas, et al., 2008).

The FPGA implementation of the MLP of Figure 17 has been first used in (Canas, et al., 2008), wherein the activation function is implemented using lookup table after discretization and the handled data are all of 8 bits. Three alternative architectures has been investigated based on how the required memories that store the input data, the weights and biases as well as the lookup tables used to implement the activation functions are implemented. The implementation alternatives are: distributed memory blocks (DRAM) or embedded memory blocks (BRAM). In the former, the flip-flops available within the configurable logic blocks (CLBs) are used while in the latter specific blocks of RAM are used. Note that, in general, the delay due to the memory access of DRAM blocks is much shorter than that due to BRAM.

Table 3 shows the number of slices required to implement the described MLP, the number of clock cycles that are necessary to accomplish a whole net computation through all layers, together with the corresponding minimal clock period. The product of the number of clock cycles times the duration of one cycle defines the evaluation time, given in the last column of Table 3. These figures are reported for three different implementation alternatives: in alternative (a), only distributed RAM for the whole designs is used; in alternative (b), the weights associated with synaptic signals together with the biases are stored in BRAM, while

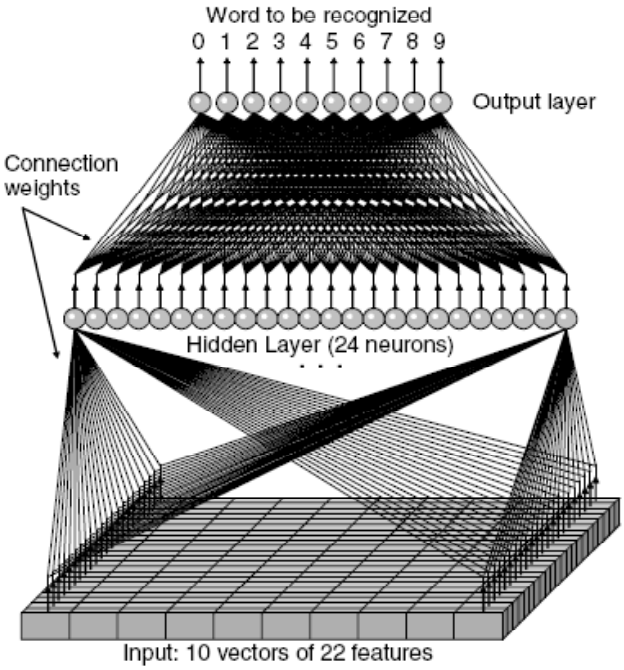

Figure 17: Testbed application MLP

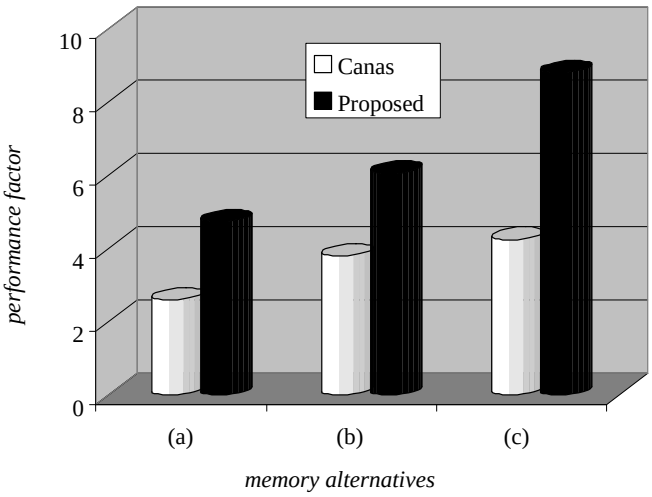

Figure 18: Comparison of the performance factor for the testbed application

the remaining data are stored in DRAM; in alternative (c), BRAMs are used to implement all required memories. In this testbed MLP, input data memory requires a total of 220 entries, the weight and biases memory needs to accommodate $220 \times 24 + 24 \times 10 = 5520$ words and the activation function table, which in our case requires only 9 fractions. However, in the implementation reported in (Canas, et al., 2008), this memory requires much more that this. The authors did not specify how many entries they used to digitize the sigmoid activation function. It is worthwhile noting that a memory entry in our design is of 32 bits while in (Canas, et al., 2008), it is of 8 bits only. Also, the design of (Canas, et al., 2008) was mapped using Virtex-E 2000 FPGA device is used. This family of FPGAs is based on 4-input LUTs.

The chart of Figure 18 illustrate the comparison of the performance factor achieved by the proposed design and that obtained for the design using lookup table for the activation function, as reported in (Canas, et al., 2008). The chart shows that our design always wins, i.e. with respect to the investigated alternatives for the implementation of the data memories. Our design always requires less hardware area due to reuse of circuitry by both weighted sum and activation function computation. Furthermore, despite the fact that the proposed design requires mores clock cycles to complete the needed computation, it does so at a higher operation frequency.

7 CONCLUSIONS

In this paper, we presented a novel hardware architecture for processing an artificial neural network, whose topology configuration can be changed *on-the-fly* without any extra effort. An extra effort was undertaken to implement efficiently arithmetic and computing models. Furthermore, the model minimizes the required the silicon area as it uses a single physical layer and re-uses by feedback it to perform all the computation executed by all the layers of the net. This is done so without deteriorating the neural network inference time. The IEEE Standard for Floating-Point Arithmetic (IEEE-754) was not used. Instead, the search for simple arithmetic circuits and which require less silicon area motivated the use of fractions to represent real numbers.

The model was specified in VHDL, simulated to validate its functionality. We also synthesized the system to evaluate time and area requirements. The comparison of the performance result of the proposed design was then compared to three similar implementations: the binary-radix straightforward design, the stochastic computing based design and the MAC-based implementation with floating-point operations. Furthermore, the design performance was compared to a similar one that uses a look-up table to implement the activation function. The proposed design has been proven superior in many aspects.

The next stage of this work is to implement all the adjustments to accommodate some learning techniques, so that the hardware could be able to infer the weights depending on the data training set presented by the user.

Table 3: Performance comparison of the proposed design with a design that uses a lookup table as activation function

MLP Design	Alternative	#Slices	#BRAM	Clock (ns)	#Cycles	Time (ms)	Performance factor
(Canas, et al., 2008)	(a)	6321	0	58.162	282	16.402	2.59
	(b)	4411	24	59.774		16.856	4.73
	(c)	4270	36	64.838		18.284	3.80
Proposed	(a)	3712	0	49.341	356	17.565	6.05
	(b)	2988	13	51.003		18.157	4.25
	(c)	2192	20	54.677		19.465	8.80

THANKS

We are grateful to FAPERJ (Fundação de Amparo á Pesquisa do Estado do Rio de Janeiro, `http://www.faperj.br`) and CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico, `http://www.cnpq.br`) for their continuous financial support. We are also thankful to the reviewers whose critics and suggestions improved the paper greatly.

REFERENCES

Bade, S. L. and Hutchings, B. L. (1994). *PGA-Based Stochastic Neural Networks*, in IEEE Workshop on FPGAs for Custom Computing Machines, pp. 189–198, IEEE, Los Alamitos.

Beuchat, J.-L. and Haenni, J.-O. and Sanchez, E. (1998). *Hardware Reconfigurable Neural Networks*, 5th. Reconfigurable Architectures Workshop, Orlando, Florida.

Botros, N. M. and Abdul-Aziz, M. (1994). *Hardware Implementation of an Artificial Neural Network using Field Programmable Arrays*, IEEE Transactions on Industrial Electronics, vol. 41, pp. 665–667.

Canas, A., Ortigosa, E. M., Diaz, A. F. and Ortega J. (2003). *XMLP: a Feed-Forward Neural Network with Two-Dimensional Layers and Partial Connectivity*, Lecture Notes in Computer Science, LNCS, vol. 2687, pp. 89–96.

Canas, A., Ortigosa, E. M., Ros, E. and Ortigosa, P. M. (2008). *FPGA implementation of a fully and partially connected MLP — Application to automatic speech recognition*, In: FPGA Implementations of Neural Networks, A. R. Omondi, J. C. Rajapakse (Eds.), Springer.

Chen, C. (2003). *Fuzzy logic and neural network handbook*, McGraw-Hill, New York.

Choi, Y. K. Ahn, K.H. and Lee, S.-Y. (1996). *Effects of multiplier output offsets on on-chip learning for analog neuro-chips*, Neural Processing Letters, vol. 4, pp. 1–8.

Dias, F. M., Antunes, A. and Mota, A. M. (2004). *Artificial neural networks: a review of commercial hardware*, Engineering Applications of Artificial Intelligence, Vol. 17, No. 8, pp. 945–952.

Ferrucci, A. T. (1994). *A Field Programmable Gate Array Implementation of self adapting and Scalable Connectionist Network*, Ph.D. thesis, University of California, Santa Cruz, California.

Gadea, R. Ballester, F. Mocholí, A. and Cerdá, J. (2000). *Artificial Neural Network Implementation on a Single FPGA of a Pipelined On-Line Backpropagation*, in Proceedings of the 13th International Symposium on System Synthesis, IEEE, Los Alamitos.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*, Second Edition, Prentice Hall International, New Jersey.

Holt, J. L. and Baker, T. E. (1991). *Backpropagation simulations using limited precision calculations*, In: Proceedings, International Joint Conference on Neural Networks, vol. 2, pp. 121–126.

Kung, H. T. (1988). *How We got 17 million connections per second*, in International Conference on Neural Networks, vol. 2, pp. 143–150.

Kung, S. Y. (1988). *Parallel architectures for artificial neural networks*, In International Conference on Systolic Arrays, pp. 163–174, 1988.

Kung, S. Y. and Hwang, J. N. (1989). *A Unified Systolic Architecture for Artificial Neural Networks*, Journal of Parallel Distributed Computing, vol. 6, pp. 358–387.

Linde, A., Nordstrom, T. and Taveniku, M. (1992). *Using FPGAs to implement a Reconfigurable Highly Parallel Computer*, In Selected papers from: Second International Workshop on Field Programmable Logic and Applications, pp. 199–210, Springer-Verlag, Berlin.

Lindsey, C. S. and Lindblad, T. (1994). *Review of hardware neural networks: A user's perspective*, 3rd. Workshop

on Neural Networks: From Biology to High Energy Physics.

Martins, R. S. (2010). *Implementação em hardware de redes neurais artificiais com topologia configurável*, M.Sc. Dissertation, Post-graduate Program of Electronics Engineering, State University of Rio de Janeiro – UERJ, UERJ/REDE SIRIUS/CTCB/S586, 2010.

Martins, R. S., Nedjah, N. and Mourelle, L. M. (2009). Reconfigurable MAC-based architecture for parallel hardware implementation on fpgas of artificial neural networks using fractional fixed point representation, *Procs. of ICANN09*, LNCS 5164, pp. 475–484, Springer, Berlin.

Moerland, P. D. and Fiesler, E. (1997). Neural Network Adaptations to Hardware Implementations, in Handbook of Neural Computation, Oxford University Publishing, New York.

Montalvo, A. Gyurcsik, R. and Paulos, J. (1997). *Towards a general-purpose analog VLSI neural network with on-chip learning*, IEEE Trans. on Neural Networks, vol. 8, no. 2, pp. 413–423.

Nedjah, N. and Mourelle, L. M. (2007). Reconfigurable hardware for neural networks: binary versus stochastic, Journal of Neural Computing and Applications, Vol. 72, No. 12, pp. 249–155. Springer, London.

Nedjah, N., Martins, R. S., Mourelle, L. M. and Carvalho, M. V. S.(2008). Reconfigurable MAC-based architecture for parallel hardware implementation on FPGAs of artificial neural networks, *Procs. of ICANN08*, LNCS 5768, pp. 169–178, Springer, Berlin.

Nedjah, N., Martins, R. S., Mourelle, L. M. and Carvalho, M. V. S. (2009). Dynamic MAC-based architecture of artificial neural networks suitable for hardware implementation on FPGAs, Neurocomputing, Vol. 72, No. 10–12, pp. 2171–2179, Elsevier, Amsterdam.

Nedjah, N., Martins, R. S., and Mourelle, L. M. (2011). *Analog Hardware Implementations of Artificial Neural Networks*, Journal of Circuits, Systems, and Computers, vol. 20, no. 3, pp. 349–373.

Santi-Jones, P. and Gu, D. (2008). Fractional fixed point neural networks: an introduction, Department of Computer Science, University of Essex, UK.

Tanenbaum, A. S. (2007).*Structured computer organization*, 5th. Edition, Prentice Hall PTR, New Jersey.

Uyemura, J. P. (2002). *Introduction to VLSI circuits and systems*, 10th. Edition, Wiley, New York.

Rojas, R. (1996). *Neural networks*, Springer-Verlag, Berlin.

Omondi, R. , Rajapakse, J. C. (2008). *FPGA implementation neural networks*, Springer, Berlin.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K. and Lang K. (1989). *Phoneme Recognition Using Time-Delay Neural Networks*, IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 37, no. 3, pp. 328–339.

Wolf, W. (2004). *FPGA-based system design*, Prentice Hall PTR, New Jersey.

Xilinx, (2008). *Microblaze processor reference guide*, `http://www.xilinx.com/support/documentation/ sw\_manuals/mb\_ref\_guide.pdf`, last acess: April, 2011.

Xilinx, (2009). *Fast simplex link v2.11b*, `http://www.xilinx.com/support/documentation/ip\_documentation/ fsl\_v20.pdf`, last access: April, 2011.

Zhang, X. and et al.,(1990). *An Efficient Implementation of the Back propagation Algorithm on the Connection Machine*, Advances in Neural Information Processing Systems, vol. 2, pp. 801–809.

Zhang, D. and Pal, S. K. (1992). *Neural Networks and Systolic Array Design*, World Scientific Company, Singapore.

Zhu, J. and Sutton, P. (2003). *FPGA Implementations of neural networks – A survey of a decade of progress*, In: Field Programmable Logic and Application, Lecture Notes in Computer Science, Vol. 2778, pp. 1062–1066, Springer, Berlin.

Zurada, J. M. (1992). *Introduction to artificial neural systems*, 1st. Edition, West Group, USA.