# DESIGN OF A PIPELINED HARDWARE ARCHITECTURE FOR REAL-TIME NEURAL NETWORK COMPUTATIONS

*J. L. Ayala, A. G. Lomeña, M. López-Vallejo, A. Fernández*

Departamento de Ingeniería Electrónica
Universidad Politécnica de Madrid, Madrid (Spain)
{jayala,lomena,marisa,angelfh}@die.upm.es

**ABSTRACT**

*In this paper, we present a digital hardware implementation of a Neural Network server. The key characteristics of this solution are on-chip learning algorithm implementation, sophisticated activation function realization, high reconfiguration capability and operation under real time constraints. Experimental results have shown that our system exhibits better response in terms of recall speed, learning speed and reconfiguration capability than other implementations proposed in the literature. Additionally, an in depth analysis of data quantization effects on network convergence has been performed and a set of design rules has been extracted.*

## 1. INTRODUCTION

Neural Networks (*NNs*) have become one of the most investigated information processing algorithms in different fields. This interest responds to the amazing capacity of NNs for representing artificial knowledge.

Traditional software implementations of NNs lack the efficiency provided by the hardware implementations (which are suitable for applying on hard time constrained applications) but rely on their easiness and short-time development. The architecture we are introducing here is an ASIC capable of on-chip learning providing fast response with high reconfigurability. Besides, the design of the chip, presenting processing units with internal and independent memories, takes into account the possibility of a future interface which could transfer the weights from a PC to the net, allowing the possibility of using different learning algorithms.

The paper is organized as follows: section 2 presents the considered design alternatives, section 3 describes the architecture of the proposed system and synthesis results are shown in section 4. Finally, some conclusions are drawn.

## 2. DESIGN ALTERNATIVES

### 2.1. Software solution

To date, practical implementations of NNs have been usually performed on software platforms. Flexibility and short time of development in the learning algorithms are the key for this choice. However, practical applications of NNs on real time applications like artificial vision and adaptive control usually require a very high data throughput that software simulation cannot generally satisfy [1].

The software simulation of a NN on a SUN UltraSparc at 200 MHz runs at 1.5 KCUPS[1] rate. Taking into account that a medium size net (10 neurons at the input layer, 100 neurons at the hidden layer and 2 neurons at the output layer) has more than 1000 synaptic connections, and that a complete learning phase could require more than 100 learning cycles, it can be shown that the software needs up to 2 minutes for the system training. But the most limiting factor is the up to 2 seconds of response time during the recall phase which usually does not satisfy real time constraints.

### 2.2. Hardware solution

ASICs are the ideal choice when high performance is required because the architecture can be adapted to the problem. However, long development time is needed and the complexity of the system could become unavoidable. In this kind of designs [2], an area-flexibility trade-off is found. ASICs cannot easily provide the flexibility demanded by NNs (diverse learning algorithms, reconfigurability, etc.) without an expensive area cost. Consequently, hardware sharing policies have to be used to optimize the hardware resources.

Different solutions regarding the implementation of NNs on programmable digital devices (FPGAs and DSPs) can be found in the literature [3, 4] but in these cases it is needed to employ many of these devices what increases both price and power dissipation. However, important advances have been achieved on this matter[5].

### 2.3. Hardware-Software solution

The problem of a joint hw/sw solution is to get an accurate partition of the subsystems being implemented on hardware and those being implemented on software. The recall phase of the neural algorithm must be as fast as required to satisfy the real time constrains; therefore, the forwardpropagation of the data during the recall phase should be implemented on the hardware architecture.

The training of the net is a task performed once before the NN begins working, so it could be thought that this phase can be performed on a software platform: the resulting synaptic weights would be transfered to the internal memories, and the hardware would be used only for the recall phase. However, the architecture we propose here, with pipelined processing, allows us to implement an on-chip learning algorithm (*backpropagation*) with high throughput and low area cost.

Finally, a hybrid approximation could employ the hardware during the forwardpropagation while the backpropagation of data would be performed on the software platform to take advantage of its flexibility (avoiding the on-chip implementation of the learning algorithm). However, table 1 shows

This work is funded by CICYT project TIC2000-0583-C02-02

[1]Kilo-Connections Updated Per Second

as time consumed in data communication (typically, a serial interface to communicate synaptic weights from the PC to the chip is needed) is higher than time consumed in data processing, forcing us to reject this approximation.

| Topology | Interface (s) | Forwardprop. (ms) |
|---|---|---|
| 10:10:10 | 0.25 | 0.066 |
| 40:40:40 | 4 | 1.066 |
| 100:100:100 | 25 | 6.66 |
| 200:200:200 | 100 | 26.66 |

Table 1: Time consumed by the serial interface and the net (port speed = 19200 bps, net speed = 3 MCPS)

## 3. DESCRIPTION OF THE ARCHITECTURE

The architecture we present emulates the operation of parallel processor architectures. The Processing Units (PUs) are arranged on a single ring, and each PU has its own internal memories. With this arrangement, we can achieve high reconfigurability[2] with only a little more area with respect to the shared memory configuration [6] (figure 1).

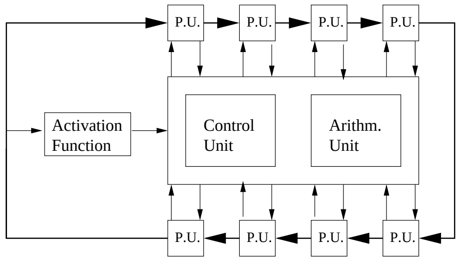

Figure 1: System architecture scheme

The ring is composed of 8 PUs and a combinational module that calculates the activation function. During the forwardpropagation, the ring carries the partial results computed by the PUs (weighted sums) and during the weight correction, the PUs communicate with the Control Process Unit (composed of a Control FSM (*CFSM*) that imposes the reconfigurations of the PUs, and an Arithmetic Unit that computes the weight correction terms). Therefore, after a few ring cycles (the ratio between the number of neurons in the network layer and the number of PUs), the outputs of the layer are obtained. These outputs are internally stored in the set up memories and they are used as inputs during the next computation stage. Finally, the resulting values of the output layer are communicated to the Control Process Unit for computing the weight correction terms. The following section explores the experimental results and methodology on the analysis of data quantization on network convergence and training error.

### 3.1. Rationale on data quantization

One of the special characteristics of the architecture we propose here is the use of floating-point representation. The oscillation of the synaptic weights of the net is extremely high and unforseeable during the training. Besides, a high precision in data codification is required due to the reduced weight correction terms managed during the training. Floating point accurately manages this situation, expanding the representation range and improving the precision.

For our current architecture, a 24 bits arrangement (18 for the mantissa and 6 bits for the exponent) has been selected. With this configuration, we are able to represent values near zero (the most probable situation because synaptic weights exhibit a Gaussian distribution) with up to six decimal digits. Extremely high (positive and negative) values can be reached with lower accuracy.

As it was said before, the system we are showing works with floating point representation in the data buses and arithmetic circuits. The reason behind this innovation with respect to previous hardware implementations of NNs is the on-chip learning algorithm implementation, to prevent the saturation of the synaptic weights during the learning phase that could cause the miss-convergence of the network.

To evaluate this effect, we have performed several analysis with a software tool from the University of Stuttgart (SNNS [7]), modified with the code required to imitate the behavior of our hardware architecture. The quantization effect on the network convergence has been devised by simulating several experiments. A set of heuristics in the hardware implementation of this type of NN can be extracted. These results have guided the design of the architecture. Figure 2 shows the learning error in a common and simple example like the XOR problem when it is simulated with the software tool. This could be considered the most accurated execution because SNNS works with simple-accuracy floating point notation (32 bits), so it will be considered as a reference in our conclusions. The graph shows the evolution of the MSE[3] (*Y* axis) versus the number of epochs (*X* axis).

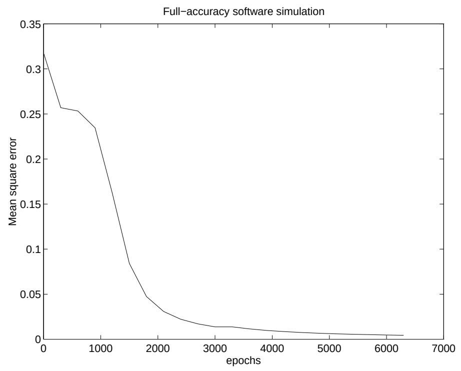

Figure 2: Simple-accuracy software simulation

One of the alternatives that can be thought is the quantization with fixed-point notation and a reduced number of bits[4] of the activation function and its derivative form. Figure 3 shows the effect of such approximation, where a longer training period can be observed. If a fixed-point quantization of the activation function and its derivative form is selected,

[2]We define *reconfigurability* as the ratio between *Enhanced throughput* and *Throughput* [6].

[3]Mean Square Error

[4]6 bits for the integer part and 4 bits for the fractional one

the word length must be significantly increased. A simulation with a 16-8 fixed-point configuration is also presented in figure 3 (*increased quantization*). A similar approach can be studied concerning the data quantization in the data buses and arithmetic circuits of the system, where a 32 bits fixed-point architecture is needed for achieving quite low error (figure 4).

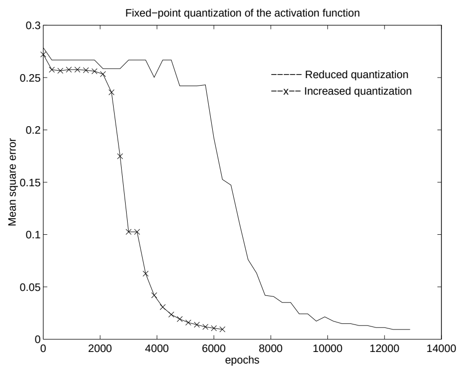

Figure 3: Quantization of the activation function

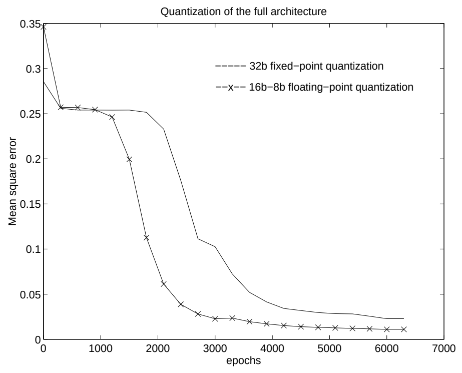

Figure 4: Quantization of the full architecture

Finally, the average word length can be reduced and the precision and representation range can be increased by employing a floating-point representation. Figure 4 also shows the results of the simulation in this case with 16 bits for the mantissa and 8 bits for the exponent, where it can be compared with the effect of the fixed-point quantization.

## 3.2. Processing Units

The PUs are composed of floating-point multiplier and adder, a register, and 32 word RAM and dual-port RAM. Each PU can operate as an individual neuron, but this is not an efficient functioning because the number of neurons per layer would be limited to the number of PUs implemented in the chip. Therefore, these modules are reconfigured at run-time to operate in conjunction with any other PU.

The size of the memories limits the maximum topology of the simulated NNs to 200 neurons per layer. Internal memories store both synaptic weights and partial results computed by the PUs. In this way, PUs can calculate the weighted sums, the delta terms involved in the weight correction calculation and the weight correction itself [8].

## 3.3. Control Process Unit

The kernel of the system is composed of a CFSM and an Arithmetic Unit (*AU*).

The CFSM controls the rest of the hardware modules existing on the system, regulating the data transfer along the ring, the data capture and the reconfiguration of the PUs. It also communicates with the AU, transferring the results to the PUs. The CFSM is composed of four concurrent and independent FSMs. These FSMs capture the input and intermediate output data (*machine 1*), communicate with the AU and reconfigure the PUs during the weight correction (*machine 2*), control the forward data transfer (*machine 3*) and capture and store the network outputs (*machine 4*).

The AU consists of floating point multiplier and dder, and the logic needed to perform the arithmetic calculations of the weight correction terms (*delta* terms). This module operates in an asynchronous way, being the CFSM responsible of capturing the result data at the precise time.

## 3.4. Activation Function

The activation function performs an accurate partition of the sample space in the classification and clustering problems (it simulates the non-linear response of the biologic neuron and allows the net to classify the input vectors into non-linear boundary sets). This must be a non-linear function, because most interesting problems show a non-linear partition of the sample space. Nowadays, the most frequent activation function is the sigmoid one[5], because it produces fast convergence, it is easy to derive and it is capable to solve the most of the application problems [9].

Traditional implementations of the activation function are based on LUTs, where the analytic function is sampled, codified and stored. This solution presents the problems of long time accesses and large area used in external memory devices (which imply high power consumption) [10].

There are many references showing that diverse approximations to the sigmoid function can be used to train the network [11]. The characteristics to be preserved in these functions are non-linearity, continuity and smooth grow between -1 and +1 (or between -0.5 and +0.5). Therefore, we can think about an approximation that emulates the behavior of the sigmoid function while being easily implemented.

We propose a combinational approximation to the sigmoid activation function based on a novel hardware implementation. This implementation approximates the sigmoid shape with linear segments becoming smaller when closer to the origin. The number and size of segments can be experimentally chosen to simulate the sigmoid function as close as needed. In our system, a seven segment configuration has been chosen, being each one half size from the former, and achieving a standard deviation of 0.0254 with respect to the analytic form. Figure 5 shows the basic algorithm implemented by the module. The input pattern is classified in a segment corresponding to its absolute value. The output

[5] $f(x) = \frac{1}{1+e^{(-D \cdot x)}}$

data is formed with a label associated to the input segment, a number of bits coming from the original data that we use to allocate the data in the segment, and a constant that permits to normalize the output range. All the operations required can be synthesized with combinational logic.

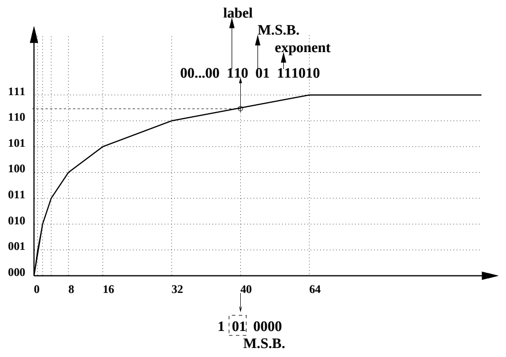

Figure 5: Activation function implementation

## 3.5. Pipeline

The architecture described in the previous paragraphs allows data process parallelization. However, such functionality could not be possible without an exact arrangement of data into the internal memories of the PUs and a precise control mechanism for reading and writing those RAMs. This control mechanism arranges data in the internal memories in a FIFO-like way, writing zeros when no-operation cycles have to be inserted. Reading begins from the bottom of the FIFO and continues upwards.

## 4. SYNTHESIS RESULTS

The system has been synthesized with a technological library of 0.35 $\mu m$ provided by *Mietec* (MTC45000). The results of the synthesis with the tools from Synopsys are: active area 46420 gates (2.5 $mm^2$), memory area 8.64 $mm^2$ and clock frequency 111 MHz.

The resulting system (*Neurapse*) has been compared with other hardware, software and mixed proposed solutions (table 2 [12]), exhibiting greater learning and recall speed, reconfiguration and learning on-chip capabilities. The system showing higher recall speed (*Neuroclassifier*) lacks on-chip learning capability and it presents a low reconfigurable architecture specifically designed for High Energy Physics and Image Processing applications.

| Type | Name | Learn | Neurons | Speed |
|---|---|---|---|---|
| Analog | ETANN | No | 64 | - |
| Digital | HNC 100-NAP | prog. | 100 PU | 64 MCUPS |
| Digital | WSI | BP | 144 | 300 MCUPS |
| Digital | N64000 | prog. | 64 PU | 220 MCUPS |
| Digital | [2] | prog. | 12 PU | 39 MCUPS |
| Digital | MANTRA I | prog. | 40 x 40 | 133 MCUPS |
| Digital | Neurapse | BP | 8 PU | 11000 MCPS 6000 MCUPS |
| Hybrid | Neuroclassifier | no | 6 | 21000 MCPS |
| Hybrid | RN-200 | BP | 16 | 3000 MCPS 40 MCUPS |
| DSP | GRD | prog. | 15 | 7 MCUPS |
| PC-Accel. | NeuralBoard | prog. | 64000 | 1500 GCUPS |
| PC-Accel. | SNAP | prog. | 100 | 128 MCUPS |
| Soft. | SUN Ultra2 200 MHz | prog. | - | 0.0015 KCUPS |

Table 2: Neural Network implementations

## 5. CONCLUSIONS

We have presented a hardware architecture cable of simulating a great diversity of NN topologies with real time constraints. This chip can perform the training phase itself, thanks to the on-chip implementation of a learning algorithm (*backpropagation*). To avoid the delay times, noise errors and power consumption of the traditional hardware implementations of the activation function (through the use of LUTs), a combinational approximation based on a compression algorithm has been proposed. The effect of data quantization on network convergence and learning error has been novelty analysed and a set of design rules has been extracted from the experimental results.

The resulting system exhibits high speed response on learning and recall phases, greater than software and previous hardware implementations. Moreover, an excellent reconfiguration capability has been achieved.

## 6. REFERENCES

[1] G. Erten, “Real time realization of early visual perception”, IC Tech, Inc., 1999.

[2] Y. Kondo, “A 1.2 GFLOPS neural network chip for high-speed neural networks servers”, *IEEE Journal of Solid State Circuits*, , n. 6, June 1996.

[3] J. Zhu, “Towards an FPGA based reconfigurable computing environment for neural network implementations”, in *ANN Conference*, pp. 661–666, September 1999.

[4] M. Murakawa, “The GRD chip: reconfiguration of DSPs for neural network processing”, *IEEE Transactions on Computers*, vol. 48, pp. 628–639, June 1999.

[5] R. Gadea, “Artificial neural network implementation on a single FPGA of a pipelined on-line backpropagation”, in *ISSS*, September 2000.

[6] P. Ienne, “Digital hardware architectures for neural networks”, *SPEEDUP Journal*, vol. 1, n. 9, June 1995.

[7] University of Stuttgart, *Stuttgart Neural Network Simulator*, www-ra.informatik.uni-tuebingen.de/SNNS/.

[8] A. R. Omondi, “Arithmetic-unit and processor design for neural networks”, Centre for High Performance Embedded Systems, Nanyang Tech. University, 2000.

[9] B. Mendil, “Simple activation functions for neural and fuzzy neural networks”, Electrical Engineering Department, University of Bejaia, 1999.

[10] L. Benini, “System-Level Power Optimization: Techniques and Tools”, *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, n. 2, pp. 115–119, July 2000.

[11] M. Skrbek, “Fast neural network implementation”, in *ICS*, pp. 375–391, September 1999.

[12] C. S. Lindsey, “Review of hardware neural networks: a user’s perspective”, Technical report, Royal Institute of Technology, Stockholm, 1998.