A Novel Digital Neural Network Hardware Implementation Technique Targeting FPGAs

Andrei Dinu, *MEMBER IEEE*
Goodrich Corporation, Birmingham, UK

Marcian Cirstea, *SENIOR MEMBER IEEE*
Anglia Ruskin University, Cambridge, UK,

Abstract

A new algorithm for compact neural network hardware implementation is presented, which exploits the special properties of the Boolean functions describing the operation of perceptrones (artificial neurones with step activation function). The algorithm contains three main steps: the digitisation of the ANN mathematical model, the conversion of the digitised model into a logic gate structure, and finally the hardware optimisation by elimination of redundant logic gates. A set of C++ programs has been developed based on the presented algorithm. The programs generate an optimised VHDL model of the ANN implementation. This strategy bridges the gap between the ANN design and simulation software (such as specialised ANN MATLAB tool-box) and the software packages used in hardware design (Viewlogic, Xilinx). Although the method is directly applicable only to neural networks composed of neurones with step activation functions, it can also be extended to sigmoidal activation functions.

Index Terms - Neural Networks, Hardware Implementation, FPGAs.

1. INTRODUCTION

Hardware implemented Artificial Neural Networks (ANNs) have an important advantage over computer simulated ANNs because they fully exploit the parallel operation of the neurones, thereby achieving a very high speed of information processing. Some VLSI implementation algorithms achieve very efficient implementation by using a combination of AND gates and OR gates alongside with Threshold Gates (TG) [1]. The direct use of TGs to implement neurones leads to compact hardware structures but this method cannot be used for FPGA implementation because they are not available inside the Configurable Logic Blocks (CLBs) of FPGAs. Although any threshold gate can be directly replaced by an equivalent group of logic gates, this simple approach leads to inefficient implementations.

The algorithm presented in this paper is applicable to both ASIC and FPGA implementation of all ANNs composed of neurones with step activation function [2]. Each neurone is treated as a Boolean function and it is implemented separately. The complexity of the implementation is minimised based on the special mathematical properties of the Boolean functions describing the functionality of neurones. The most advantageous property of such a Boolean function is that if its truth table is constructed as a matrix with as many dimensions as neurone inputs, then the truth table

has only one large group of '1' and one large group of '0'. It is important to note that the solid group of '1' is not visible when the Gray codification is used and therefore classical Karnaugh maps or Quine-McClusky algorithms cannot efficiently make use of this mathematical property of the function. The algorithm presented here takes a different approach to the problem and eventually generates a multilayer pyramidal hardware structure where layers of AND gates alternate with layers of OR gates. The bottom of the pyramid consists of an incomplete layer of NOT gates. The obtained structure is optimised at a later stage by detecting and eliminating redundant groups of logic gates. Additional optimisation of the overall ANN is achieved by also eliminating redundant structures between different neurones.

2. THE ALGORITHM

This section presents the details of the implementation algorithm. Thus, each neurone of the ANN is first converted into a binary equivalent neurone whose inputs are only '1' and '0'. This is a two-step process described by sections 2.1.1 and 2.1.2. Subsequently, the binary neurone model is transformed into a logic gate structure by means of an iterative procedure explained in section 2.2.2 using a set concepts defined in 2.2.1.

2.1. The Digitisation of the Mathematical Model of one Neurone

The most appropriate binary codification to be used for neurone input quantities is the complementary code (also named "two's complement" C_2). It is generally used for integer number representations, but it can be readily adapted for real values in the interval [-1; +1). Thus, considering a n-bit representation "$b_{n-1}b_{n-2}b_{n-3}....b_1b_0$", the corresponding integer value (I_n) is given by:

$$I_n = -2^{n-1} \cdot b_{n-1} + \sum_{i=0}^{n-2} 2^i \cdot b_i \qquad (1)$$

The largest positive number, which can be represented on 'n' bits, is 2^{n-1}-1 while the smallest number is -2^{n-1}. Real values between -1.0 and +1.0 can be represented dividing the corresponding integer value I_n by 2^{n-1}. Therefore, equation (2) illustrates the complementary code for real numbers:

$$R_n = \frac{I_n}{2^{n-1}} = -b_{n-1} + \sum_{i=0}^{n-2} 2^{-n+1+i} \cdot b_i \qquad (2)$$

The transformation of the analogue neurone model into an appropriate digital model is a carried out in two stages. At each stage, the input weights and the threshold levels of the initial neural network are altered in a careful manner so that the functionality of the neurone is not affected. This can be achieved by keeping constant the sign of the argument of the activation function:

$$\operatorname{sign}\left(\sum_{i=1}^{m} w_i \cdot x_i - t\right) = \operatorname{sign}(net - t) = constant \tag{3}$$

However, for reasons of mathematical simplicity, a more restrictive condition is used instead, namely the argument "net-t" of the activation function is kept itself constant rather than only the sign of it:

$$\sum_{i=1}^{m} w_i \cdot x_i - t = net - t = constant \tag{4}$$

2.1.1. Conversion Stage One

The first step transforms the analogue inputs of the neurones into digital inputs expressed as groups of n_b bits. This process is associated with transforming each analogue neurone input into an equivalent group of n_b binary inputs. The task is achieved by splitting each input defined by its initial weight w_{ij} into n_b subinputs, whose weights w_{ijp} (p=0,1, ... n_b-1) are calculated as follows:

$$\begin{cases} w_{ijp}^{(1)} = \dfrac{2^{p+1}}{2^{n_b}} \cdot w_{ij} \;\forall\; p < n_b - 1 \\ w_{ij(n_b-1)}^{(1)} = -w_{ij} \\ t_i^{(1)} = t_i \end{cases} \tag{5}$$

The superscript '(1)' in equations (5) shows that the corresponding quantities have been calculated during the first conversion stage. Likewise, the superscript '(2)' identifies the quantities calculated during the second conversion stage.

The result of the previous calculations is that the initial 'm' inputs are turned into 'm' input clusters, each cluster containing 'n_b' subinputs (Fig. 1). The symbol 'w_{ij}' stands for the weight number 'j' of the neurone 'i' in the network, while '$w_{ijp}^{(1)}$' represents the weight of subinput 'p' in cluster 'j' pertaining to neurone 'i'. The index p=0 corresponds to the least significant binary figure, while p=n_b-1 corresponds to the most significant one.

According to the previous considerations, only those neurone parameter changes that maintain the argument "net_i-t_i" of the activation function constant are allowed. The argument corresponding to the neurone after the first conversion stage is calculated as

$$net_i^{(1)} - t_i^{(1)} = \sum_{j=1}^{m}\sum_{p=0}^{n_b-1} w_{ijp}^{(1)} \cdot x_{jp}^{(1)} - t_i^{(1)} = \sum_{j=1}^{m}\left(-w_{ij} \cdot x_{jp}^{(1)} + \sum_{p=0}^{n_b-2} w_{ij} \cdot \frac{2^{p+1}}{2^{n_b}} \cdot x_{jp}^{(1)}\right) - t_i^{(1)} \tag{6}$$

where $x_{jp}^{(1)}$ (p=0,1,2,...n_b-1) are the bits of the complementary code received by each new neurone input. Equation (6) can be transformed into

$$net_i^{(1)} - t_i^{(1)} = \sum_{j=1}^{m} w_{ij} \cdot \left(- x_{j(n_b-1)}^{(1)} + \sum_{p=0}^{n_b-2} 2^{-n_b+p+1} \cdot x_{jp}^{(1)} \right) - t_i^{(1)} \quad (7)$$

The expression between parentheses corresponds to the extended complementary code definition given in equation (2). Therefore, (7) is further transformed into

$$net_i^{(1)} - t_i^{(1)} = \sum_{j=1}^{m} w_{ij} \cdot x_j - t_i^{(1)} = \sum_{j=1}^{m} w_{ij} \cdot x_j - t_i = net_i - t_i \quad (8)$$

where x_j is an analogue input value of the initial neurone. This proves that the condition expressed by (4) is fulfilled. Thus, during the first conversion stage the codification style based on the complementary code has been introduced and the required modifications of the neurone parameters have been performed so that the neurone behaviour has been maintained unchanged.

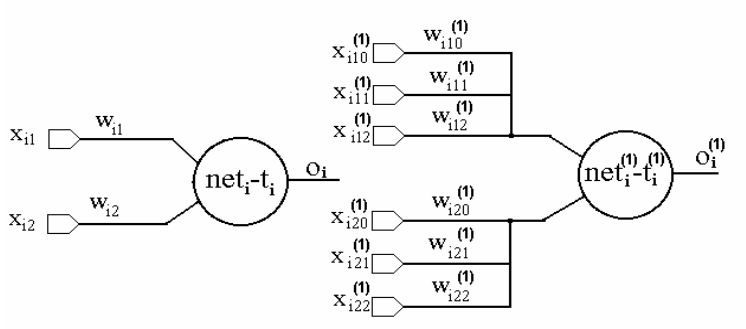

Fig. 1 The neurone model before and after stage one of the conversion

2.1.2. Conversion Stage Two

The neurones after the first conversion stage can have both positive and negative weights. The second conversion stage aims to replace these neurones with equivalent ones having only positive weights. The simplest way to eliminate negative input weights is to use only the module of their values. Consequently, the relationship between stage one neurone weights and their stage two counterparts is expressed by

$$w_{ijp}^{(2)} = \left| w_{ijp}^{(1)} \right| \quad (9)$$

Adopting this method means that supplementary parameter alterations are required in order to counteract the neurone behaviour alteration caused by changing the sign of some input weights. A simple solution is to reverse the value of the input bits corresponding to negative input weights at conversion stage one. The modification can be readily implemented into hardware with NOT logic gates. The relationship between the input bits supplied to stage-two neurones $x_{ijp}^{(2)}$ and the input bits supplied to stage-one neurones ($x_{ijp}^{(1)}$) is expressed by the following function:

$$x_{ijp}^{(2)} = \begin{cases} x_{ijp}^{(1)} \text{ if } w_{ijp}^{(1)} > 0 \\ 1 - x_{ijp}^{(1)} \text{ if } w_{ijp}^{(1)} < 0 \end{cases} \tag{10}$$

The two alternatives in (10) can be compressed into equation

$$x_{ijp}^{(2)} = \frac{1 - \text{sign}\left(w_{ijp}^{(1)}\right)}{2} + \text{sign}\left(w_{ijp}^{(1)}\right) \cdot x_{ijp}^{(1)} \tag{11}$$

where the 'sign' function is defined by

$$\text{sign}(x) = \begin{cases} +1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases} \tag{12}$$

Using (9) and (11), the argument of the transfer function for stage-two neurones can be calculated as

$$net_i^{(2)} - t_i^{(2)} = \sum_{j=1}^{m} \sum_{p=0}^{n_b-1} \left[\frac{\left|w_{ijp}^{(1)}\right| - \text{sign}\left(w_{ijp}^{(1)}\right) \cdot \left|w_{ijp}^{(1)}\right|}{2} + \text{sign}\left(w_{ijp}^{(1)}\right) \cdot \left|w_{ijp}^{(1)}\right| \cdot x_{ijp}^{(1)} \right] - t_i^{(2)} \tag{13}$$

This can be further expressed as

$$net_i^{(2)} - t_i^{(2)} = \sum_{j=1}^{m} \sum_{p=0}^{n_b-1} w_{ijp}^{(1)} \cdot x_{ijp}^{(1)} + \sum_{j=1}^{m} \sum_{p=0}^{n_b-1} \left(\frac{\left|w_{ijp}^{(1)}\right| - w_{ijp}^{(1)}}{2} \right) - t_i^{(2)} \tag{14}$$

The arguments of the activation function before and after the second conversion stage have to be equal. This condition is expressed by equation (15).

$$\sum_{j=1}^{m} \sum_{p=0}^{n_b-1} w_{ijp}^{(1)} \cdot x_{ijp}^{(1)} + \sum_{j=1}^{m} \sum_{p=0}^{n_b-1} \left(\frac{\left|w_{ijp}^{(1)}\right| - w_{ijp}^{(1)}}{2} \right) - t_i^{(2)} = \sum_{j=1}^{m} \sum_{p=0}^{n_b-1} w_{ijp}^{(1)} \cdot x_{ijp}^{(1)} - t_i^{(1)} \tag{15}$$

Therefore, the threshold level of the stage-two neurones is the following:

$$t_i^{(2)} = t_i^{(1)} + \sum_{j=1}^{m} \sum_{p=0}^{n_b-1} \frac{\left|w_{ijp}^{(1)}\right| - w_{ijp}^{(1)}}{2} \tag{16}$$

The parameters of the stage-one neurone in equation (16) depend on the initial parameters of the analogue neurone as described by equation (5). Consequently, by substituting (5) in (16), the expression of the neurone threshold can be successively transformed as:

$$t_i^{(2)} = t_i + \sum_{j=1}^{m} \frac{\left|w_{ij}\right| + w_{ij}}{2} + \sum_{j=1}^{m} \sum_{p=0}^{n_b-2} \frac{\frac{2^{p+1}}{2^{n_b}} \cdot \left|w_{ij}\right| - \frac{2^{p+1}}{2^{n_b}} \cdot w_{ij}}{2} \tag{17}$$

This expression can be successively transformed into increasingly simpler formats as demonstrated by equations (18) through (22).

$$t_i^{(2)} = t_i + \sum_{j=1}^{m} \frac{\left|w_{ij}\right| + w_{ij}}{2} + \sum_{j=1}^{m} \sum_{p=0}^{n_b-2} \frac{2^p}{2^{n_b}} \cdot \left(\left|w_{ij}\right| - w_{ij}\right) \tag{18}$$

$$t_i^{(2)} = t_i + \sum_{j=1}^{m} \frac{\left|w_{ij}\right| - w_{ij}}{2} + \sum_{j=1}^{m} \sum_{p=0}^{n_b-2} \frac{2^p}{2^{n_b}} \cdot \left(\left|w_{ij}\right| - w_{ij}\right) + \sum_{j=1}^{m} w_{ij} \tag{19}$$

$$t_i^{(2)} = t_i + \sum_{j=1}^{m} \sum_{p=0}^{n_b-1} \frac{2^p}{2^{n_b}} \cdot \left|w_{ij}\right| - \sum_{j=1}^{m} \sum_{p=0}^{n_b-1} \frac{2^p}{2^{n_b}} \cdot w_{ij} + \sum_{j=1}^{m} w_{ij} \tag{20}$$

$$t_i^{(2)} = t_i + \frac{2^{n_b}-1}{2^{n_b}} \cdot \sum_{j=1}^{m} \left|w_{ij}\right| - \frac{2^{n_b}-1}{2^{n_b}} \cdot \sum_{j=1}^{m} w_{ij} + \sum_{j=1}^{m} w_{ij} \tag{21}$$

$$t_i^{(2)} = t_i + \left(1 - 2^{-n_b}\right) \cdot \sum_{j=1}^{m} \left|w_{ij}\right| + 2^{-n_b} \cdot \sum_{j=1}^{m} w_{ij} \tag{22}$$

To sum up, the parameters of the binary neurones after the second conversion stage can be directly calculated as a function of the initial analogue neurone parameters as follows:

$$\begin{cases} w_{ijp}^{(2)} = \dfrac{2^{p+1}}{2^{n_b}} \cdot \left|w_{ij}\right| \quad p = 0,1,2,...n_b - 1 \\ t_i^{(2)} = t_i + \left(1 - 2^{-n_b}\right) \cdot \sum_{j=1}^{m} \left|w_{ij}\right| + 2^{-n_b} \cdot \sum_{j=1}^{m} w_{ij} \end{cases} \tag{23}$$

This final solution has been obtained by combining equation (22) with the result of substituting (5) in (9).

2.2. The Binary Neurone Implementation

As previously mentioned, the ANN implementation into a hardware structure is performed separately for each neurone. The implementation method requires at first that the input weights $w_{ijp}^{(2)}$ are sorted in descending order. The sorted array contains a number of A=m x n_b elements: $w_1^s, w_2^s, w_3^s \ldots w_A^s$, where 'm' is the number of initial analogue neurone inputs and 'n_b' the number of bits for each input binary code. These weights correspond to the input binary signals $x_1^s, x_2^s, \ldots x_A^s$. An iterative conversion procedure is used to analyse the input weights and to generate the corresponding netlist description of the logic gate implementation. At each conversion step, the iterative procedure decomposes the larger neurone into subneurones. Some of them can be implemented with only a few AND and OR logic gates while the rest are further decomposed into simpler subneurones until all of them have been implemented.

2.2.1. Preliminary Definitions

At this point, several important concepts need to be defined: **terminal weight group**, **group threshold level**, **cumulated weight**, **critical weight**, **non-critical weight, insignificant weight**.

A **terminal weight group** is a set of weights comprising the last N consecutive elements in the sorted array. Therefore any **terminal weight group** can be uniquely identified as $G_t(F)$ by specifying the index 'F' of its first element. There are a number of A overlapping **terminal weight groups** in the sorted array: $G_t(1)$, $G_t(2)$, $G_t(3)$, ...,$G_t(A)$. **Terminal weight group** $G_t(1)$ encompasses all the weights in the array.

The **group threshold level** is a quantity (denoted by T(F)) calculated by the conversion algorithm for any terminal group of weights which is to be converted into a subneurone. The **group threshold level** is equal to the corresponding subneurone threshold level. The group threshold level for $G_t(1)$ is $T(1)=t^{(2)}$.

The **cumulated weight** of a terminal group $G_t(F)$ is defined as the sum of its component weights.

$$W_t(F) = \sum_{i=F}^{A} w_i^s \tag{24}$$

There are neurones or subneurones whose outputs cannot be active unless certain input signals are active as well ('1'). These are critical inputs and correspond to **critical weights**. **Critical weights** are determined based on condition (25).

$$W(F) - w_{F+i}^s < T(F) \qquad \forall i = 0,1,2,\ldots Z-1 \tag{25}$$

Non-critical inputs can affect the neurone or subneurone output, but are not critical. The neurone output can be '1' even if a non-critical input is '0'. The corresponding weights are **non-critical weights**. In a terminal group there are **non-critical weights** if condition (26) is fulfilled:

$$\begin{cases} T(F) - W(F) < 0 \\ T(F) - \sum_{i=0}^{Z-1} w_{F+i}^s = T(F) - W(F) + W(F+Z) > 0 \end{cases} \tag{26}$$

Insignificant inputs do not influence the neurone output at all. Insignificant inputs correspond to **insignificant weights** that are very small and do not affect the relation between the **group cumulated weight** and the **group threshold level**.

2.2.2. The iterative implementation procedure

The hardware implementation process is conceived in terms of terminal groups. The starting point of the analysis is the terminal group $G_t(1)$ which contains all the binary inputs of the neurone. The iterative implementation procedure uses

three input parameters: the index defining the current terminal group (F), the current threshold level (T) and the logic gate type (LGT) which can have one of two possible values: ANY_GATE or AND_GATE. If LGT=AND_GATE then the next gate to be added to the netlist description is necessarily an AND gate, otherwise it can be either an OR gate or an AND gate.

At the first step F=1, $T=t^{(2)}$ and LGT=ANY_GATE, where $t^{(2)}$ is the neurone threshold level as calculated according to equations (23). The conversion procedure follows a sequence of 8 steps:

Step 1 If LGT=AND_GATE then go to Step 7, else go to Step 2.

Step 2 Calculate the number X of input weights which are larger then the T. These weights are: $x_F^s, x_{F+1}^s, \ldots x_{F+x-1}^s$. Calculate the cumulated weights W(F+X), W(F+X+1), ..., W(A). Determine number Y of the cumulated weights larger then T. If X>1 and Y=0 then go to Step 3. If X>1 and Y>0 then go to Step 4. If X=1 and Y=0 go to Step 5. If X=0 and Y=1 go to Step 7. If X=0 and Y=0 then go to Step 6.

Step 3 An X-input OR gate is added to the netlist. The gate is driven by the input signals $x_F^s, x_{F+1}^s, \ldots x_{F+X-1}^s$. Go to Step 8.

Step 4 An X+Y-input OR gate is added to the netlist. The gate is driven by the input signals $x_F^s, x_{F+1}^s, \ldots x_{F+X-1}^s$ and by the outputs of the subneurones corresponding to terminal groups Gt(F+X), Gt(F+X+1), ...,Gt(F+Y-1). For i=1,2,3,...,Y recall the analysis procedure Y times to generate these subneurones. The parameters for each call are: F=F+i and T=t, LGT=AND_GATE. Afterwards go to Step 8.

Step 5 A single input is capable of triggering the neurone output while the other inputs do not influence its operation. The neurone is implemented as a direct connection between this input and the output. Go to Step 8.

Step 6 No inputs can trigger the neurone. The neurone output can be implemented as a connection to the ground. Go to Step 8.

Step 7 An AND gate is required. The number Z of critical weights inside $G_t(F+1)$ is determined. If there are no non-critical weights then a Z+1-input AND gate is added to the netlist. The gate is driven by inputs $x_F^s, x_{F+1}^s, \ldots x_{F+Z}^s$. Go to Step 8. If there are non-critical inputs in the current terminal group then a Z+2-input AND gate is added. Its inputs are signals $x_F^s, x_{F+1}^s, \ldots x_{F+Z}^s$ and the output of a further subneurone based on terminal group Gt(F+Z+1). The procedure is recalled with parameters F=F+Z+1, T=T(F)-W(F)+W(F+Z), LGT=ANY_GATE.

Step 8 Return to the calling procedure. If there is no calling procedure left unfinished, then stop the process. After the iterative process is finished, inverter gates are added to those inputs corresponding to the initial negative weights at stage one of the neural model digitisation process.

2.3. Hardware Optimisation

The hardware implementation netlist obtained with the procedure described in the previous section is highly redundant. There are redundancies both inside each neurone and across different neurones. Most of the redundancies can be eliminated using a simple procedure: the file is repeatedly analysed and every time several logic gates are found which have the same input signals and are of the same type, all but one of these gates are removed from the netlist. The interconnections are updated so that the remaining logic gate supplies all the inputs that were initially driven by the eliminated gates. If at least one gate was eliminated, the process starts again. This optimisation ends when no more gates can be removed.

3. NEURONE IMPLEMENTATION EXAMPLE

For a better understanding of the implementation algorithm, an example is presented in Fig. 2. The neurone has A=12 input weights and a positive threshold level. Thus, the weights are sorted in descending order and the recursive implementation procedure is initiated with parameters F=1 and T_t=10. The first three weights in the ordered array are larger than the threshold level so that inputs 4, 7 and 1 will drive an OR gate alongside with the subneurones built on the basis of terminal groups $G_t(4)$, $G_t(5)$ and $G_t(6)$. Terminal groups $G_t(7)$ through $G_t(11)$ are not taken into account because their group weights do not surpass the threshold level. Terminal group $G_t(4+1)=G_t(5)$ has Z=0 critical weights and several non-critical weights so that a Z+2=2-input AND gate is used.

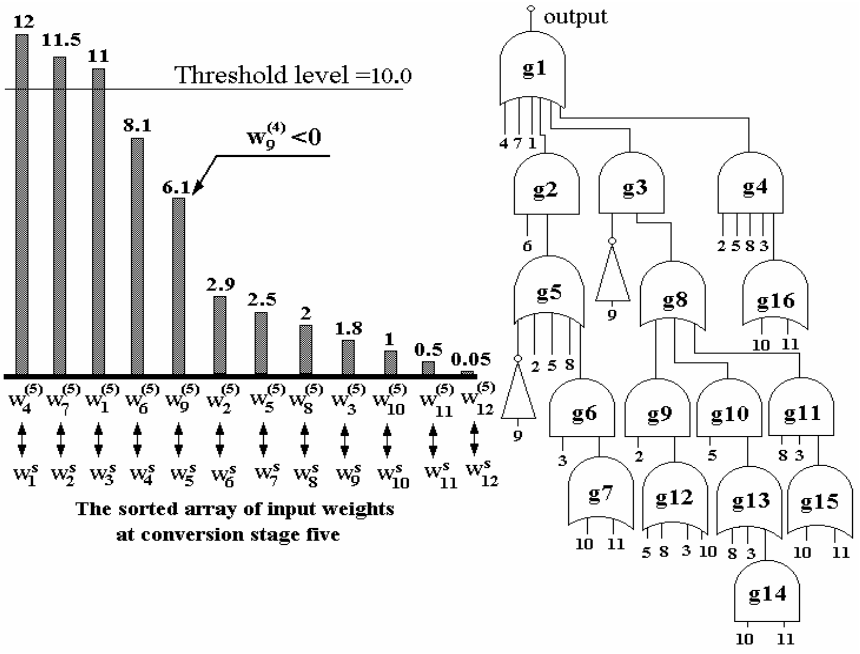

Fig. 2 Digital mathematical model to gate structure conversion example

The iterative procedure is recalled with parameters F=5 and T_t=1.9 to generate the implementation of the subneurone connected to gate g2. This subneurone has four dominant inputs, one critical input and two non-critical inputs. Therefore, gates g5 and g6 are inserted into the hardware structure. The remaining two inputs belong to a higher-order subneurone that requires the iterative procedure to be called for the third time, with the parameters F=10 and T_t=0.1. The corresponding subneurone contains two dominant inputs and it is implemented by the OR gate g7. At this stage, calls number 2 and 3 of the iterative procedure are finished. The control is handed over to call number 1 which initiates the call number 4 to generate the implementation of the subneurone connected to the AND gate g3. The parameters are F=5, T_t=1.0 which leads to the conclusion that the new subneurone has only non-critical weights. The subneurone is implemented by logic gates g8, g9, g10 and g11 and it is connected to three third-order subneurones. These subneurones are analysed during procedure calls number 5, 6 and 7 and their implementations contain the gates g12 to g15. The end of procedure call 7 brings procedure call 4 to an end as well. The control is passed to procedure call 1, which initiates the call number 8 that implements the subneurone connected to the AND gate g4. This second-order subneurone has two dominant inputs, so that it is implemented by the OR gate g16. The end of procedure call number 8 is followed by the end of procedure call number 1, which stops the recursive process. The third procedure is called and inverter gates are connected to the inputs related to the weight w_9. After this stage is finished, the neurone hardware implementation is complete. Due to its small value, the weight w_{12} is insignificant and the corresponding input is not necessary in any combination of non-critical inputs. The example demonstrates that ANNs containing several neurones require an amount of calculations, which cannot be efficiently performed without specialised software instruments.

4. THE AUTOMATED ALGORITHM IMPLEMENTATION ON A COMPLETE CASE STUDY

The implementation algorithm presented in this paper was automated by means of a set of three C++ programs that generate the netlist description of the implementation; optimise the netlist; and generate the VHDL model of the final circuit respectively. These are the programs CONV_NET.CPP, OPTIM.CPP and VHDL_TR.CPP illustrated in Fig. 3.

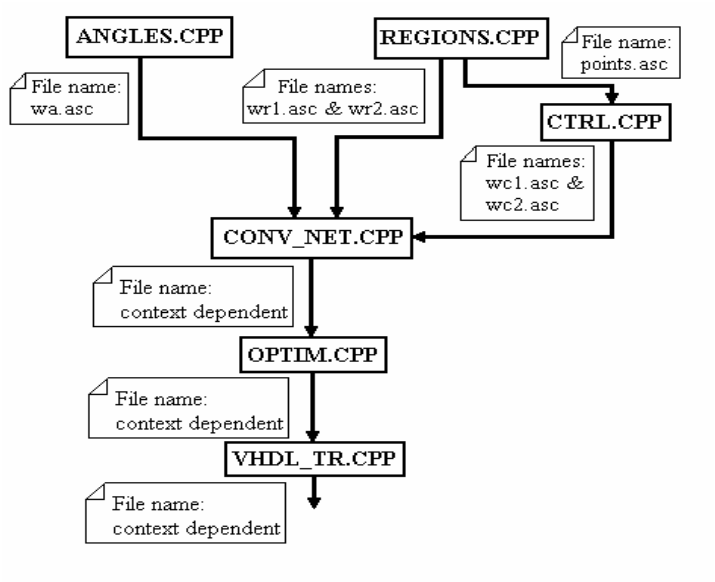

Fig. 3 The neural PWM generator design programs and their interconnections

The three ANNs described by ANGLES.CPP, REGIONS.CPP and CTRL.CPP are the constituent parts of a more complex feed-forward artificial neural network which generates the Pulse Width Modulation (PWM) switching pattern for a power inverter. The neural PWM generator operates such as to minimise the current ripple in the load of the power inverter when the load has an inductive character. This is achieved by analysing two two-dimensional vectors: the load current variation during one PWM pulse ($\Delta\underline{i}$) and the non-inductive load voltage ($\underline{V}_{nL}$) [3]. The vector $\underline{V}_{nL}$ is calculated by subtracting the voltage drop across the equivalent load inductance from the total load voltage.

$$\underline{V}_{nL} = \underline{V}_{load} - L\frac{d\underline{i}}{dt} \tag{27}$$

Alternatively, these vectors can be treated as complex numbers. As demonstrated in [3], [4] and [5], the optimal PWM sequence can be calculated as a function of three parameters: the argument of $\Delta\underline{i}$; the argument of $\underline{V}_{nL}$; and the absolute value of $\underline{V}_{nL}$. The tasks of the three neuronal subnetworks are divided as follows:

⇨ The first subnetwork (generated by ANGLES.CPP) analyses the argument of the vector $\Delta\underline{i}$.

⇨ The second subnetwork (generated by REGIONS.CPP) analyses the argument and the absolute value of the vector $\underline{V}_{nL}$.

⇨ The third subnetwork combines the outputs of the first two subnetworks and generates three binary outputs representing the instantaneous polarity of the three output voltages that need to be generated by the three-phase PWM inverter (the three outputs of a typical power inverter change their polarity without changing their absolute value).

The top three programs shown in Fig. 3 have been developed to generate automatically the matrix of weights describing each neurone layer in each of the three neural networks involved in the PWM application. All the programs in Fig. 3 communicate by means of ASCII files. These programs are called in the correct order by a master program that supervises the entire process of generating the optimised VHDL code.

Designing an ANN for a specific application involves the use of either training algorithms or constructive algorithms. In contrast with training algorithms, constructive ones are able to determine both the network architecture and the neurone weights and are guaranteed to converge in finite time. This makes them the preferable approach in many practical situations. Several constructive algorithms, reviewed in [7], have been developed in the last decade. They are divided into three categories: geometric ([8], [9]), network-based [10] and algebraic [11].

The numerical values of all the neurone weights and thresholds have been calculated using the geometric constructive solution known as Voronoi diagrams [8], [6]. This method proved particularly efficient in the case of designing the subnetwork which analyses the input quantity $\underline{V}_{nL}$. In this case, the complex plane has been divided up into triangular Voronoi cells which correspond with certain approximate modulus and argument values which require special decisions to be taken by the PWM generator. The conversion process is monitored by a master program that controls the user interface and calls all the six specialised programs in the correct order. This allows the user to control the main parameters of the neural networks to be generated [3]:

- Number of triangular Voronoi cells.
- Number of sectors used to divide the 360 degrees interval when analysing the argument of $\Delta\underline{i}$.
- Number of bits used to code the components of the two complex input quantities $\Delta\underline{i}$ and $\underline{V}_{nL}$.
- Maximum fan-in for the logic gate of the VHDL model.

Several combinations of the above parameters have been tried out and the solution generating the optimal performance-complexity ratio has been adopted. The same number of bits was used to code the components of the two complex input quantities $\Delta\underline{i}$ and $\underline{V}_{nL}$. This number was successively given the values 4, 5 and 6, to check the effect on the number of gates. The effect is shown below:

Nb = 4	**Nb = 5**	**Nb = 6**
Angle subnetwork - 252 gates	Angle subnetwork - 378 gates	Angle subnetwork - 408 gates
Position subnetwork - 132 gates	Position subnetwork - 242 gates	Position subnetwork - 343 gates
Control subnetwork - 709 gates	Control subnetwork - 709 gates	Control subnetwork - 709 gates

The number of gates in the Control Subnetwork is always the same, as it depends on the number of Voronoi cells in the previous two subnetworks, but not on the number of bits used to represent the voltage and the current.

The chosen implementation solution uses 5 input bits to code each analogue input, as a compromise between complexity (size of network in number of gates) and precision. The 360^o interval is divided into 36 sectors and the complex plane is divided into 54 triangular Voronoi cells. The initial netlist description of the first subnetwork contained 568 logic gates arranged on 11 gate layers. After the optimisation stage, the netlist contained 242 gates (representing 42.6% of the initial gate count). The initial and final numbers of gates for the second subnetwork are 660 and 378, which means a compression to 57.27%. This subnetwork has been implemented by a logic gate structure with 14 layers. On the other hand, the control signal generation subnetwork has been optimised from 1329 to 709 gates resulting in a compression to only 23.43%. The corresponding hardware implementation contains 6 layers of logic gates. Therefore, the total number of logic gates in the optimised neural implementation is 1329 logic gates. The longest path from inputs to outputs passes through 14+6=20 layers of logic gates. For comparison,

5. SIMULATION AND EXPERIMENTAL RESULTS

Two sets of tests were performed: a general ANN operation speed test (described in section 5.1) and an application related functional test of the PWM inverter controller (described in section 5.2). Both were performed in two stages: i) timing simulation, using Xilinx/Viewlogic software, and ii) oscilloscope practical test after implementation into a Xilinx XC4010XL FPGA, mounted on a XS40 test board.

5.1. General operation speed test

The carry out the operation speed of the neural network tests, a VHDL testbench, supplying the neural network with variable input signals, was developed. In the adopted configuration, the neural network requires a total of 20 input bits. A series of input patterns is generated by a 20-bit counter. To obtain a pseudo-random sequence of input patterns a supplementary block has been introduced in the testbench between the counter and the neural network. This block simply rearranges the 20 bits inside the neural network. The XS40 test board contains a 12MHz clock oscillator; a supplementary clock divider was added to the testbench structure to perform a frequency step down to 1.5MHz, as shown in Fig. 4.

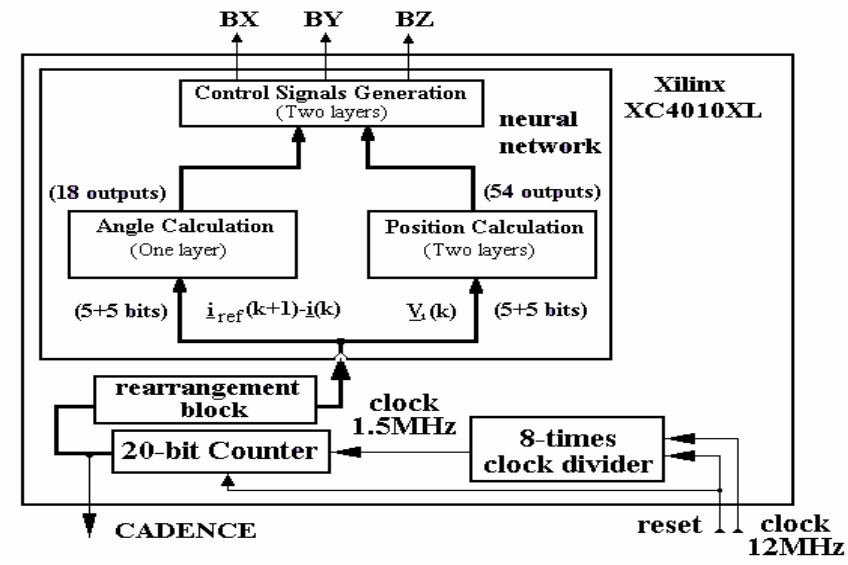

Fig. 4 Testbench for neural network operation speed testing

The least significant output bit of the 20-bit counter changes every time a new input pattern is generated. This signal (named 'CADENCE' in Fig. 4) indicates the rhythm of the pattern generation process and it can be used to measure the propagation delay through the neural network. Thus, the propagation delay is the time between the edge of signal CADENCE and the moment when signals BX, BY, BZ are stable on the output pins.

The VHDL testbench has been synthesised and downloaded into a Xilinx XC4010XL FPGA chip. The synthesis tool provided by Xilinx Foundation analyses the abstract VHDL model and generates an optimised implementation file (bitstream) in accordance with the application requirements and the structural details of the target chip. The software allows two types of optimisation: for speed and for implementation complexity (or chip area). Each of them can be carried out with high or with low computational effort. The FPGA chip used for this implementation contains 400 CLBs. Each CLB includes two D-type flip-flops and three logic function generators. Two of the function generators have 4 inputs each, while the third one has only 3 inputs. All the function generators are implemented as look-up tables and therefore the corresponding propagation delays are independent of the Boolean functions.

The synthesis of the complete VHDL testbench was performed using the options of chip area optimisation with high computational effort. The resulting implementation used 192 CLBs, meaning 48% of the hardware resources available in the chip. On the other hand, the separate implementation of the neural network (without the additional counter and frequency divider for testing) used only 179 CLBs, representing 45% of the chip resources.

After synthesis, timing simulation was performed to analyse the propagation delays in the FPGA chip. The maximal propagation delay found during the simulation is 110ns, the minimal one is 35ns, but the majority of the delay

times are about 50ns. In other words, the propagation time is less than 1.5 clock cycles, which proves the very high operation speed of the neural network. Fig. 5 presents a fragment of a timing simulation result. The three propagation delays vary from one transition of inputs to another. Hazard effects are also visible in the neural network output.

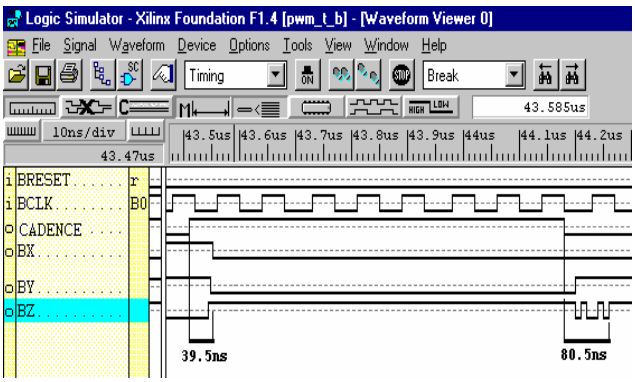

Fig. 5 Timing simulation using Xilinx software

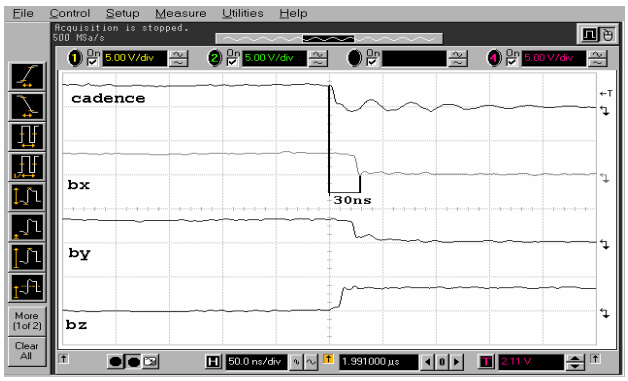

Fig. 6 FPGA implementation experimental result

Fig. 6 shows waveforms captured with a digital oscilloscope using a time scale of 50ns / division. The measurements performed led to the conclusion that the propagation delay is less than 80ns, which is equivalent to less than a single clock cycle. This demonstrates the fundamental advantage of hardware implemented NN: they achieve far superior operating speeds compared with any other digital circuit performing calculations of comparable complexity.

5.2. Application related test: PWM control signals

After the circuit was initially tested by simulation and verified experimentally using the method presented above, an application related test followed, when the circuit was used to generate the PWM pulses controlling a power inverter. The 6 channels (3 pairs of antiphase PWM signals) simulation waveform is illustrated in Fig. 7 and the experimental waveforms for the first 4 channels are shown in Fig. 8. The application test proved the correct operation ([2], [3]) of the hardware implemented neural network controller, developed using the new algorithm presented in this paper.

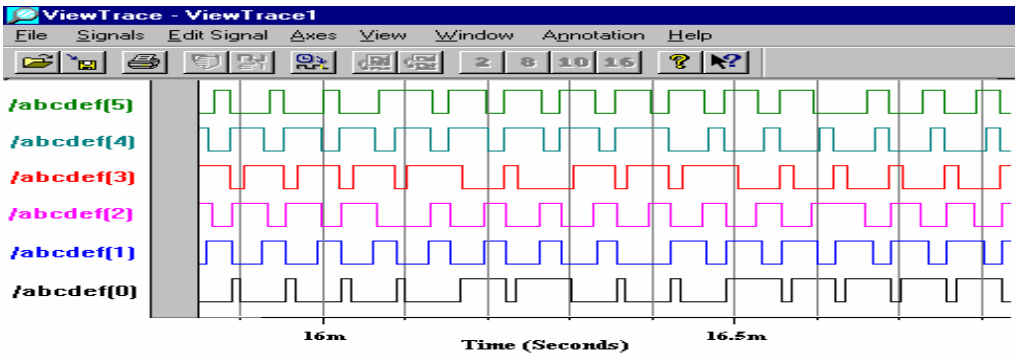

Fig. 7 Timing simulation of the 6 PWM inverter control channels

Fig. 8 Experimental testing: the oscilloscope waveforms of the first 4 PWM channels

The PWM controller was then used to control the speed of an 11.1 kW three-phase induction motor, with the parameters given in Table 1. The controller inputs are set to: Is=10A, ω=314rad/s. Experimental tests were carried out [3] **with the neural controller implemented in the Xilinx XC4010XL FPGA and the motor. Fig. 9 illustrates the controlled torque characteristic versus the natural one, plotted using experimental measurements. An improvement in maintaining constant speed for variable load torque is achieved, proving high performance of the neural controlled drive for low-dynamic applications. This validates the neural hardware FPGA implementation technique.**

TABLE 1

ELECTRICAL PARAMETERS OF THE INDUCTION MOTOR

Quantity		Value
Stator Resistance	R_s	0.371 Ω
Rotor Resistance	R_r	0.415 Ω
Stator Leakage Induct.	$L_{s\sigma}$	2.72 mH
Rotor Leakage Induct.	$L_{r\sigma}$	3.3 mH
Mutual Inductance	L_m	84.33 mH
Number of pole pairs	p	1
Rated Power	P	11.1 kW
Moment of inertia	J	0.015 kg·m^2

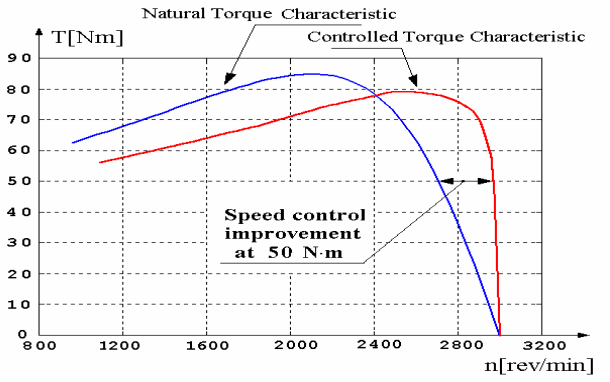

Fig. 9. Controlled versus natural torque characteristic

6. CONCLUSIONS AND DISCUSSION

A new digital hardware implementation strategy for feed-forward ANNs with step activation functions is reported. The novel algorithm developed treats each neurone as a special case of Boolean function with special properties which can be exploited to achieve a very compact implementation.

The hardware implementation is accomplished by means of reusable VHDL code that can be easily translated into an FPGA implementation using suitable EDA software [2]. The VHDL programs bridge the gap between the facilities offered by the simulation software (such as the specialised ANN MATLAB tool-box) and software packages specialised in hardware design (Viewlogic, Xilinx). The approach outlined is applicable to a large category of modern digital controllers for power electronics, based on neural networks.

The present algorithm can be extended to encompass neurones with sigmoidal activation function if the activation function is approximated by a set of small successive steps. The complexity of the resulting circuit will be much higher in this case because each neurone will be approximated by a series of Boolean functions. At the same time, the redundancies in the implementation netlist will be much higher as well, due to the mathematical similarity of the consecutive steps, approximating the sigmoid of each neurone. Efficient optimisation algorithms are therefore essential for such applications.

7. REFERENCES

[1] H. E. Ayestaran and R. W. Prager, 'The Logical Gates Growing Network", *Technical Report CUED/F-INFENG/TR 137,* Engineering Department, Cambridge University, 1993.

[2] M. N. Cirstea, A. Dinu, J. Khor and M. McCormick: "Neural and Fuzzy Logic Control of Drives and Power Systems", Elsevier Science Ltd., Oxford, UK, 2002, ISBN 0750655585.

[3] A. Dinu, "FPGA Neural Controller for Three Phase Sensorless Induction Motor Drive Systems", PhD Thesis, De Montfort University, 2000.

[4] A. Dinu, M. Cirstea, M. McCormick, A. Ometto and N. Rotondale, "Neural ASIC Controller for PWM Power Systems", *IEEE ASIC Conference*, pp.29-33, 1998.

[5] A. Dinu, M. Cirstea and M. McCormick, "A Novel Neural PWM Controller", *IEE SIMULATION'98 Conference*, pp. 375-379, 1998.

[6] R. A. Dwyer, "High-dimensional Voronoi Diagrams in Linear Expected Time", *Discrete Computational Geometry,* vol. 6, pp. 343-367, 1991.

[7] V. Beiu, "VLSI Complexity of Discrete Neural Networks", Gordon and Breach & Harwood Academics Publishing, 1998.

[8] B. K. Bose and A. K. Garga, "Neural Network Design Using Voronoi Diagrams", *IEEE Transactions on Neural Networks,* vol. 4, no. 5, pp. 778-787, 1993.

[9] U. Ramacher and M. Wesseling, "A Geometrical Approach to Neural Network Design", in *IEEE IJCNN'89 Conf.*, vol 2, pp. 147-153, January 1989.

[10] F. J. Smieja, "Neural Network Constructive Algorithm: Trading Generalisation for Learning Efficiency?", *Circuits, Systems, Signal Processing,* vol. 12, no. 2, pp. 331 - 374, 1993.

[11] J. E Hopcroft and R. L. Mattson, "Synthesis of Minimal Threshold Logic Networks", *IEEE Transactions on Electronic Components*, EC-6, pp. 552-560, 1965.