A new method for the re-implementation of threshold logic functions with cellular neural networks

Yohann Bénédic, Patrice Wira, Jean Merckle

International Journal of Neural Systems, Vol. 0, No. 0 (April, 2000) 00–00

A NEW METHOD FOR THE RE-IMPLEMENTATION OF THRESHOLD LOGIC FUNCTIONS WITH CELLULAR NEURAL NETWORKS

Y. BÉNÉDIC, P. WIRA*, J. MERCKLÉ
Laboratoire MIPS, Université de Haute Alsace, 4 rue des Frères Lumière
Mulhouse, 68093, France
{*yohann.benedic, patrice.wira; jean.merckle*}*@uha.fr*

Received (to be inserted
Revised by Publisher)

Abstract

A new strategy is presented for the implementation of threshold logic functions with binary-output Cellular Neural Networks (CNNs). The objective is to optimize the CNNs weights to develop a robust implementation. Hence, the concept of generative set is introduced as a convenient representation of any linearly separable Boolean function. Our analysis of threshold logic functions leads to a complete algorithm that automatically provides an optimized generative set. New weights are deduced and a more robust CNN template assuming the same function can thus be implemented. The strategy is illustrated by a detailed example.

1. Introduction

Cellular Neural Networks (CNNs) represent an important paradigm of Artificial Neural Networks (ANNs). Indeed, all cells of a CNN are identical and operate simultaneously in addition to every other friendly properties of ANNs. Through their particular architecture, CNNs are well suited for hardware implementation and their parallel computing properties can thus be fully exploited. As a consequence, the theoretical computational speed of CNNs can be very fast compared to other types of ANNs, specially in real-time applications.

Nevertheless, their correct operation is sensitive to noise. This happens mainly because of the electrical components that compose the cell and which are not ideal ones. Indeed, electrical components are affected by noise, imperfections in the fabrication process, or post-manufacturing disturbance like temperature. A CNN implementation is an analog processor, informations are therefore represented by electrical signals and operations are achieved by electrical circuits. Some quantification errors can therefore appear and be easily propagated resulting in erroneous behavior of cells for some tasks. This is referenced to as the CNN internal noise.

Binary-output CNNs, which share this property of being realizable in hardware, achieve final activations of ± 1 with certain restrictions over the original CNN model. Furthermore, with inputs and initial internal states restricted to Boolean values, their overall behavior is then the one of a Threshold Logic Function (TLF).

Hence, this article addresses the problem of implementing arbitrary threshold logic functions with noise-robust CNN templates. Therefore, we introduce a new method based on an original mathematical framework for linearly separable Boolean functions referred to as generative set. Synthesizing a CNN template with generative sets allows to optimize its weights and to increase its robustness to the internal noise. As an advantage, the pro-

*Corresponding author.

posed method uses any correct template as an *a priori* knowledge to decrease the time needed for its optimization.

The paper is structured as follows. In section 2, the CNN concept is presented and different ways to compute weights are reviewed. The paradigm of the generative set is then introduced in section 3 as an alternative representation of TLFs. ¿From an analysis of a CNN template, we deduce an algorithm able to compute its generative set. Section 4 proposes an algorithm that optimizes any known generative set. This section also shows how to use it in order to resynthesize the original template into a more robust one. Each step of this procedure is illustrated with an example. Finally, concluding remarks are offered in the last section.

2. Cellular Neural Networks

2.1. *The CNN concept*

CNNs where first introduced by Chua and Yang in [1]. They are large arrays of identical nonlinear dynamic systems called cells which satisfy two properties: interactions are local within a finite radius, and all state variables are continuous-valued signals. CNNs thus consist of cells connected only to the cells in their neighborhood. They are arranged in an internal layer and are connected to both the input layer and the output layer through neighboring cells, as shown by Fig. 1.

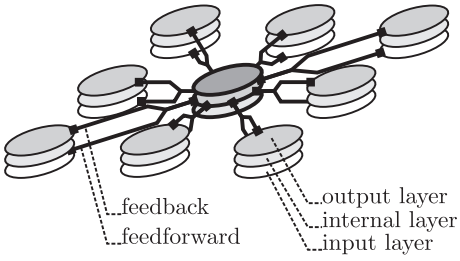

Fig. 1. Schematics of a two-dimensional CNN with a 3×3-wide neighborhood.

The main feature of CNNs, which are considered as a special class of recurrent neural networks, is therefore that information is directly exchanged just between neighboring cells. As a result, the connectivity of CNNs is greatly reduced and makes them particularly suitable for on-chip implementations, so allowing high speed parallel processing for real-time applications in many fields [2]: image processing, artificial vision, partial differential equation solution, nonlinear phenomena modeling, biological systems modeling, robot control, etc.

A mathematical formal description of the internal dynamics of a cell is given by the following equation with t the time variable:

$$\frac{dx(t)}{dt} = -x(t) + \mathbf{A} * \mathbf{N}(y(t)) + \mathbf{B} * \mathbf{N}(u(t)) + z, \quad (1)$$

where the scalars $x(t)$, $u(t)$ and $y(t)$ represent respectively the internal state, the input and the output of the cell. The matrix $\mathbf{A}$ holds the weights from the feedback connections (output) and likewise the matrix $\mathbf{B}$ holds the weights from the feedforward connections (input). z is a bias term. Since all the neurons of a CNN are identical, the values $\mathbf{A}$, $\mathbf{B}$, and z are the same for every cells and are therefore called a template of the CNN. $\mathbf{N}(.)$ is a neighborhood function and the symbol "$*$" stands for a linear convolution product.

The output $y(t)$ is computed from $x(t)$ thanks to the following piecewise-linear activation function:

$$y(t) = \tfrac{1}{2}(|x(t)+1| - |x(t)-1|). \quad (2)$$

Binary-output CNNs are derived from the latter model by giving the self-feedback weight a_0 in $\mathbf{A}$ a value greater than 1 [1]. Restricting the values of the inputs and the initial internal states to Boolean values, then leads to the conclusion that a binary-output CNN behaves like a threshold logic gate [3]:

$$y(t \to \infty) = 1 \Leftrightarrow \begin{cases} \mathbf{A} * \mathbf{N}(y(t)) + \mathbf{B} * \mathbf{N}(u(t)) - x(t) > -z \\ \forall\, t \geq 0 \end{cases} \quad (3)$$

The implementation of a TLF is achieved by setting $\mathbf{A}$, $\mathbf{B}$ and z.

2.2. *Weights computation*

Using conventional learning algorithms to find a template results in real-valued weights not necessarily optimal. The problem then remains that of finding appropriate weights. Appropriate weights are weights that yield robust CNNs, i.e., more tolerant against internal noise. Once determined,

these weights are kept constant during the application of the CNN template to a specific problem.

Since CNNs are able to implement any arbitrary linearly separable Boolean function [4], they can thus be analyzed with similar mathematical theories. This has led to many approaches, which have been developed over these last years.

Of these approaches, the most widely used is the look-up tables of Chow's parameters [5]. Indeed, a set of parameters is shown to fully characterize TLFs of up to eight variables. Thus, it is possible to recover the TLF implementation from a look-up table which would store the implementation of every possible TLF, along with their respective Chow's parameters. In a practical sense, such tables exist for TLF with less than six variables [6], but methods have been developed to break higher dimensional TLF into smaller ones [7]. Other techniques have been proposed, either inspired from Karnaugh map minimization procedures [8], or based on a formal analysis [4], on an analytical description [9] or on an algebraic representation [3].

Despite the strength of these methods, none of them is able to optimize the robustness of a given template without starting from scratch, thus leading to time consuming procedures. As opposed to them, the method developed thereafter is able to alter an existing CNN design in order to enhance its robustness to noise. It is based on specific properties of TLFs from which it derives a new arithmetic expression of a TLF: its *generative set* [10]. A simple algorithm is then used to tweak the template into a more robust one, according to the properties revealed by its generative set [11].

3. Generative Sets

3.1. *Introduction*

A generative set is an alternative representation of a threshold logic function, making good use of the latter's algebraic properties such as symmetry and monotony. It can be directly computed from a cloning template thanks to an analatical process detailed in [10], and which is briefly recalled in this section.

A threshold logic function $\mathcal{F}$ is formally defined by the following thresholded weighted sum:

$$\begin{aligned}\mathcal{F}(\mathbf{s}) &\equiv (\mathbf{w} * \mathbf{s} > Th) \\ &\equiv \left(\sum_{i=1}^{n} w_i s_i > Th \right), \end{aligned} \tag{4}$$

where Th is the so-called threshold and the weights w_i and inputs s_i are the elements of $\mathbf{w}$ and $\mathbf{s}$ respectively.

One can notice the similar look of Eq. (3) and Eq. (4). In the latter, the n elements of $\mathbf{A}$ and $\mathbf{B}$ were simply gathered into a single weighting vector $\mathbf{w}$, whereas the inputs and outputs from $\mathbf{N}(y(t))$ and $\mathbf{N}(u(t))$ were collected into $\mathbf{s}$. The expression of Th is known since [1]:

$$Th = (1 - a_0)x(0) - z, \tag{5}$$

where $a_0 > 1$ is the self-feedback weight value of $\mathbf{A}$. As a result, a binary-output cloning template ends up being fully characterized by the threshold logic function's attributes: $\mathbf{w}$, $\mathbf{s}$ and Th.

Without loss of generality, we will assume from now on that the weights w_i are positive, and that they are sorted such that $w_i \geq w_j$ if $i > j$. Before dealing with the generative set extraction algorithm, the following subsection introduces the definitions of symmetry and monotony in the Boolean algebra.

3.2. *Definitions*

The previous general expression of a TLF implies two important properties: symmetry and monotony. The smart usage of these properties makes the generative set representation to be more efficient than the truth table by being less redundant.

Definition 1 (The symmetry of a TLF)
A TLF is called symmetric if and only if it is invariant under any permutation of its inputs. A TLF is only partially symmetric in a set of inputs if and only if the permutation of these inputs leaves the function unchanged [12].

Definition 2 (The monotony of a TLF)
A TLF is [12]*:*

- *constant in one input s_i if and only if its output does not depend on s_i;*
- *increasing/decreasing in s_i if and only if its output increases/decreases with the increase of s_i.*

Thanks to the commutativity property of the addition in Eq. (4), the output of any arbitrary threshold logic function $\mathcal{F}$ is invariant under any permutation of equally weighted inputs and therfore symmetric in these inputs. As a result the value of $\mathcal{F}$ can only be affected by the number of true inputs among a set of symmetrical ones rather than by the way these true values are mapped.

Now let c be the number of sets of equally weighted variables and n_i the number of variables gathered in the set labeled i. Notice that a set might contain just one variable and that $\sum_{i=1}^{c} n_i = n$. Any input vector $\mathbf{s}$ can then be replaced by a vector $\mathbf{k}$, called reduced input vector and built as follows.

Definition 3 (Reduced input vector) *A reduced input vector is a vector* $\mathbf{k} = (k_c, \ldots, k_1)$ *made of reduced variables* k_i *which are the number of true inputs within the* i^{th} *set of symmetrical inputs of* $\mathbf{s}$. *Opposed to the values of the inputs* s_i *which are restricted to* ± 1, *the values of* k_i *are defined by* $k_i = 0, 1, \ldots n_i$.

As a consequence to the Def. 3, two input vectors $\mathbf{s}$ and $\mathbf{s}'$, derived from each others by one or more permutations of equally weighted inputs, can both be replaced by the same reduced input vector $\mathbf{k}$. This leads to a loss of redundancy in the description of a TLF made by the reduced input vectors compared to the one made with the usual input vectors. As a result, $\mathcal{F}(\mathbf{s})$ can be re-written as a function of $\mathbf{k}$: $\mathcal{F}(\mathbf{s}) \equiv \mathcal{F}(\mathbf{k})$ and the Eq. (4) becomes:

$$\mathcal{F}(\mathbf{k}) \equiv \left(\sum_{i=1}^{c} w_i(2k_i - n_i) > Th \right), \qquad (6)$$

where the equal weight values have been factorized and the respective set of symmetrical inputs have been replaced by their reduced input counterpart [10].

The symmetries are thus used to define a reduced input space in which the description of any TLF is much less redundant than in the usual input space. The following uses the monotony property to bring another improvement to the description of TLF.

Let $\mathbf{k}^+$ be a reduced vector such that $\mathcal{F}(\mathbf{k}^+) \equiv true$. Such a vector is called a minterm of $\mathcal{F}$. Since all the weights are considered positive, increasing any element k_i^+ of $\mathbf{k}^+$ clearly increases the value of the weighted sum of the Eq. (6). As a result, $\mathcal{F}$ is increasing in each one of its c reduced variables. Therefore, increasing either element of a given reduced vector $\mathbf{k}$ will only affect $\mathcal{F}(\mathbf{k})$ in one out of the following three ways:

- the weighted sum is not risen enough and $\mathcal{F}(\mathbf{k})$ remains $false$;
- the weighted sum reaches the threshold and $\mathcal{F}(\mathbf{k})$ switches from $false$ to $true$;
- the weighted sum remains over the threshold and $\mathcal{F}(\mathbf{k})$ remains $true$.

This shows that once a minterm $\mathbf{k}^+$ of $\mathcal{F}$ is known, many more can be derived from it, simply by increasing the value of at least one of its elements. These derived minterms are thus useless to the representation of a TLF. Hence, there must be a quite small set of minterms that can generate every single minterm of $\mathcal{F}$. This set is called the generative set of the function.

Definition 4 (The generative set of a TLF) *The set of minterms that generate every single minterm of* $\mathcal{F}$ *is referred to as the* generative set *of* $\mathcal{F}$ *and is written* $\{\mathcal{F}\}$. *It gathers generative vectors, i.e., any minterm* $\mathbf{k}$ *of* $\mathcal{F}$ *that cannot be derived from another minterm. As a convention, the generative vectors are indexed so as to be sorted in an anti-lexical ordering.*

According to the previous definition, the generative set $\{\mathcal{F}\}$ of a TLF obviously describes $\mathcal{F}$ completely. Furthermore, the number of generative vectors included in $\{\mathcal{F}\}$ can be considered as a way to measure the complexity of the function $\mathcal{F}$. The following subsection describes a way to extract the generative set of the TLF performed by a CNN template.

3.3. *Analyzing algorithm*

The analyzing algorithm discussed in this subsection aims at extracting the generative set of a TLF implemented by a template $\mathbf{w}$ and Th. It is assumed that $\mathbf{k} = (k_c, \ldots, k_1)$ and $\mathbf{w} = (w_c, \ldots, w_1)$, with $w_i > w_j$ if and only if $i > j$.

For a given value κ_c of k_c, Eq. (6) writes:

$$\sum_{i=1}^{c-1} w_i(2k_i - n_i) > Th - w_c(2\kappa_c - n_c), \qquad (7)$$

which is the definition of a TLF of $c-1$ reduced variables. As a result, the generative set of a TLF of c variables is the concatenation of admissible values of k_c with the generative sets obtained from their respective TLF of $c-1$ variables. This remark leads to the idea that the analyzing algorithm should be recursive. Such a recursion would stop when a TLF of one variable and a threshold value Th' are reached:

$$w_1(2k_1 - n_1) > Th'. \tag{8}$$

In this case, the generative set consists in a single mono-dimensional reduced vector, which component is the lowest integer value of k_1 that satisfies the latter inequality. It formally writes:

$$k_1 = 1 + \left\lfloor \frac{1}{2 \cdot w_1} \left(Th' + w_1 \cdot n_1 \right) \right\rfloor. \tag{9}$$

An adequate threshold value has to be forwarded from one recursive level to another in order to be evaluated and eventually stop the algorithm. This new threshold is defined at each recursive level by:

$$Th_c = Th + \sum_{i=1}^{c} w_i n_i. \tag{10}$$

Thanks to this, Eq. (7) becomes:

$$\mathcal{F}(\mathbf{k}) \equiv \left(\sum_{i=1}^{c-1} 2 \cdot w_i \cdot k_i > Th_c + 2 w_c \kappa_c \right). \tag{11}$$

Finally, the threshold value which must be forwarded to a recursive launch for a given value κ_c of k_c is:

$$Th_{c,\kappa_c} = Th_c - 2 \cdot \kappa_c \cdot w_c. \tag{12}$$

One can see that this algorithm requires exactly $\Pi_{i=1}^{c}(1+n_i)$ recursive launches, which is usually lower than 2^n, assuming that there exists i such that $n_i \neq 1$. This number can be significantly reduced. It is the case when the value of a reduced variable is high enough to ensure that the underlying TLF is always *true*, which means that the resulting generative set is the null vector and does not need to be computed. Speaking of k_c, this limit value is:

$$\kappa_{c,\max} = 1 + \left\lfloor \frac{Th_c}{2 \cdot w_c} \right\rfloor. \tag{13}$$

Likewise, a reduced input value might be too small to allow the underlying TLF to be *true*, thus leading to an empty generative set. The smallest value of k_c for which this does not happen is given by:

$$\kappa_{c,\min} = 1 + \left\lfloor \frac{1}{2 \cdot w_c} \left(Th_c - 2 \sum_{i=1}^{c-1} w_i \cdot n_i \right) \right\rfloor. \tag{14}$$

The way the expressions of both Eq. (13) and (14) are obtained is detailed in [10].

These two limits define a set of allowed integer values of k_c: $[\min(n_c, \kappa_{c,\max} - 1); \max(0, \kappa_{c,\min})]$ for which recursive launches must be made. The analyzing algorithm finally writes:

```
1  function extractGenerativeSet(c,wc...w1,Thc)

3  GenerativeSet = empty;
   if (c == 1)
5    compute(k1max); % cf. Eq. (13)
     if (k1max <= n1)
7      GenerativeSet.add(k1max);
     end;
9  else
     compute(kcmax); % cf. Eq. (13)
11   compute(kcmin); % cf. Eq. (14)

13   if (kcmax <= nc)
       GenerativeSet.add(kcmax,0,0,...,0); % c-1 zeros
15   end;

17   for kc = min(kcmax-1,nc):max(kcmin,0)
       % recursive launch, Thc updated with Eq. (12)
19     SubGSet = \
         extractGenerativeSet(c-1,wc-1...w1,Thc-2*kc*wc);
21
       % adds kc in front of every vector of SubGSet
23     GenerativeSet.add(kc,SubGSet);
     end;
25 end;
   return GenerativeSet;
27
   end function;
```

3.4. *Illustrative example*

This example considers an arbitrary CNN template to illustrate, through real values, how the analyzing method works. The weights can result from any of the synthesis methods previously discussed in section 2.2. For example:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & a_0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

$$\mathbf{B} = \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 2.5 & 5 \\ 5.5 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

and $z = 2.5$.

The inputs values are ± 1 and the initial state is $x(0) = 0$. Notice that the weights of $\mathbf{B}$ are sorted and are all positive. In this example, no weights are equal, thus $n_i = 1$ for every i and $c = 4$.

The algorithm is initialized with Th given by (5) and with Th_4 given by (10). In the present case $Th = -z = -2.5$ and $Th_4 = 11.5$.

For the sake of clarity, the operations involved by the analyzing algorithm are detailed line by line:

- **l. 1** since $c = 4$, the algorithm jumps to line 10.
- **l. 10** $\kappa_{4,\max} = 2$ according to (13).
- **l. 11** $\kappa_{4,\min} = 0$ from (14).
- **l. 13** since $\kappa_{4,\max} > n_4$, the vector $(2, 0, 0, 0)$ is not added to the generative set.
- **l. 17** the iteration will start at $k_4 = 1$ down to $k_4 = 0$.
- **l. 19** recursive launch for $k_4 = 1$:
  - **l. 1** since $c = 3$, the algorithm jumps to line 10.
  - **l. 10** $\kappa_{3,\max} = 1$ according to (13).
  - **l. 11** $\kappa_{3,\min} = 0$ from (14).
  - **l. 13** since $\kappa_{3,\max} \leq n_3$, the vector $(1, 0, 0)$ is added to the generative set.
  - **l. 17** the iteration will start at $k_3 = 0$ down to $k_3 = 0$.
  - **l. 19** recursive launch for $k_3 = 0$:
    - **l. 1** since $c = 2$, the algorithm jumps to line 10.
    - **l. 10** $\kappa_{2,\max} = 1$ according to (13).
    - **l. 11** $\kappa_{2,\min} = 0$ from (14).
    - **l. 13** since $\kappa_{2,\max} \leq n_2$, the vector $(1, 0)$ is added to the generative set.
    - **l. 17** the iteration will start at $k_2 = 0$ down to $k_2 = 0$.
    - **l. 19** recursive launch for $k_2 = 0$:
      - **l. 1** since $c = 1$, the algorithm jumps to line 5.
      - **l. 5** $\kappa_{1,\max} = 1$ according to (13).
      - **l. 6,7** since $\kappa_{1,\max} \leq n_1$, the vector (1) is added to the generative set.
      - **l. 25** the generative set is returned.
    - **l. 22** the vector $(0, 1)$ is added to the generative set.
    - **l. 23** no more iterations.
    - **l. 25** the generative set is returned.
  - **l. 22** the vectors $(0, 1, 0)$ and $(0, 0, 1)$ are added to the generative set.
  - **l. 23** no more iterations.
  - **l. 25** the generative set is returned.
- **l. 22** the vectors $(1, 1, 0, 0)$, $(1, 0, 1, 0)$ and $(1, 0, 0, 1)$ are added to the generative set.
- **l. 23** recursive launch for $k_4 = 0$:
  - **l. 1** since $c = 3$, the algorithm jumps to line 10.
  - **l. 10** $\kappa_{3,\max} = 2$ according to (13).
  - **l. 11** $\kappa_{3,\min} = 1$ from (14).
  - **l. 13** since $\kappa_{3,\max} > n_3$, the vector $(1, 0, 0)$ is not added to the generative set.
  - **l. 17** the iteration will start at $k_3 = 1$ down to $k_3 = 1$.
  - **l. 19** recursive launch for $k_3 = 1$:
    - **l. 1** since $c = 2$, the algorithm jumps to line 10.
    - **l. 10** $\kappa_{2,\max} = 1$ according to (13).
    - **l. 11** $\kappa_{2,\min} = 0$ from (14).
    - **l. 13** since $\kappa_{2,\max} \leq n_2$, the vector $(1, 0)$ is added to the generative set.
    - **l. 17** the iteration will start at $k_2 = 0$ down to $k_2 = 0$.
    - **l. 19** recursive launch for $k_2 = 0$:
      - **l. 1** since $c = 1$, the algorithm jumps to line 5.
      - **l. 5** $\kappa_{1,\max} = 1$ according to (13).
      - **l. 6,7** since $\kappa_{1,\max} \leq n_1$, the vector (1) is added to the generative set.
      - **l. 25** the generative set is returned.
    - **l. 22** the vector $(0, 1)$ is added to the generative set.
    - **l. 23** no more iterations.
    - **l. 25** the generative set is returned.
  - **l. 22** the vectors $(1, 1, 0)$ and $(1, 0, 1)$ are added to the generative set.
  - **l. 23** no more iterations.
  - **l. 25** the generative set is returned.
- **l. 22** the vectors $(0, 1, 1, 0)$ and $(0, 1, 0, 1)$ are added to the generative set.
- **l. 23** no more iterations.
- **l. 25** the generative set is returned.

Finally, putting together the five generative vectors which were returned by the first recursive level of the algorithm gives the following generative set:

$$\{\mathcal{F}\} = \left\{ \begin{array}{l} (1,1,0,0) \\ (1,0,1,0) \\ (1,0,0,1) \\ (0,1,1,0) \\ (0,1,0,1) \end{array} \right\}. \tag{15}$$

The generative set (15) completely describes the operator studied in this example. The low redundancy of this representation is visible. Indeed, five vectors are only necessary to describe the operator compared to the sixteen rows usually involved in a truth table of four inputs. It is also noticable that the algorithm took care of the anti-lexical ordering mentioned in Def. 4.

Like this example, any binary-output cloning template can be analyzed into a generative set with the proposed algorithm.

4. Optimization of CNN templates

The behavior of a binary-output cloning template, i.e. of a TLF, is described as the linear separation of the 2^n vertices of a n-dimensional hypercube into the two classes labeled -1 and 1. As previously stated by Eq. (4), the separation is achieved by an hyperplane $\mathcal{H}$ defined by $\mathbf{w}$ and Th. The task implemented by a given CNN is considered non-optimal, or non-robust to internal noise, if the hyperplane is poorly positioned in the hypercube, i.e., being unnecessarily close to one of its vertices. This property is crucial since CNNs generally deal with analog signals, meaning that even binary values (namely, the inputs and the initial internal states) will slightly vary about their mean value.

We now propose an algorithm which partially solves this optimization problem. It is based on an analysis of the generative set of the TLF achieved by a CNN. By looking for non-implemented symmetries, this algorithm tweaks the original hyperplane to make it fit them. It results in an improved implementation of the TLF.

4.1. *Generative sets reduction*

The previously described algorithm is an efficie tool to know which TLF a given binary-outp CNN performs. As seen in section 3, the return Boolean expression is given in a reduced inp space of c dimensions, as opposed to the inp space which is the usual n-dimensional hypercube that was discussed so far. This reduction relies on the values $\mathbf{n} = (n_c, \ldots, n_1)$ introduced in the previous section, and which are representative of the way symmetries were implemented in the original hyperplane $\mathcal{H}$.

If it turned out from the analysis of the generative set implemented by $\mathcal{H}$ that a symmetry is not implemented, say between the reduced inputs k_i and k_{i+1}, then it would mean that $\mathbf{n}$ can be even more reduced into $\mathbf{n}' = (n_c, \ldots, n_i + n_{i+1}, \ldots, n_1)$. This extra reduction can then be applied to the generative vectors:

$$\forall \mathbf{k} \in \{\mathcal{F}\}, \mathbf{k} \rightarrow \mathbf{k}' = (\mathbf{k_c}, \ldots, \mathbf{k_i} + \mathbf{k_{i+1}}, \ldots, \mathbf{k_1}). \tag{16}$$

This extra reduction affects the weights in the following manner:

$$\mathbf{w} \rightarrow \mathbf{w}' = \left(w_c, \ldots, \frac{n_i w_i + n_{i+1} w_{i+1}}{n_i + n_{i+1}}, \ldots, w_1 \right), \tag{17}$$

whereas the threshold value is maintained: $Th' = Th$.

The resulting hyperplane $\mathcal{H}' = \{\mathbf{w}', \mathbf{Th}'\}$ was proven to implement the same TLF as $\mathcal{H}$ in [11]. The following subsection explains why $\mathcal{H}'$ is more robust to internal noise than $\mathcal{H}$.

4.2. *Toward an optimal hyperplane*

Given an implementation of a given TLF, the aim of any analytical optimization process is to find a better (if not the best) location Th and orientation $\mathbf{w}$ of the separating hyperplane $\mathcal{H}$. An adequate representation of the TLF which implementation is supposed to be optimized must therefore be chosen. The most straightforward one probably is the truth-table. The flaw behind this approach is that it is not threshold-logic-specific: every piece of informations about the equation of the sought hyperplane remain spread out, and yet, difficult to collect. On the contrary, the generative set representation is very close to the hyperplane's coordinates (see Fig. 2).

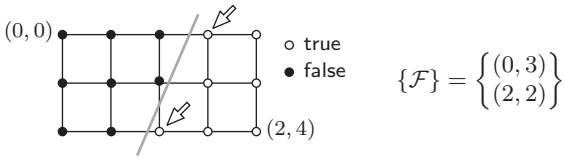

Fig. 2. A TLF represented in its two-dimensional reduced space. The vertices from its generative set are pointed by the arrows.

Amongst the many hyperplanes which implement a given TLF, the optimal ones are defined as those which distances to their respective closest vertices of the hypercube, are the furthest. More formally:

Definition 5 (Optimal hyperplanes) *Consider a hyperplane $\mathcal{H}$ defined by $\{\mathbf{w}, Th\}$. Let $\mathbf{s}^{(1)}$ be the regular input vector assigned to the closest vertice of the n-dimensional hypercube, $\mathbf{s}^{(2)}$ the second closest one, and so on. $\mathcal{H}$ is said optimal, for the TLF it implements, if and only if* [11]*:*

1. *the distance $\mathcal{D}^{(1)} = \left| Th - \sum_{i=1}^{n} s_i^{(1)} w_i \right|$, between $\mathcal{H}$ and $\mathbf{s}^{(1)}$, is greater than or equal to the distance between any other hyperplane implementing the same TLF, and its respective closest vertice;*
2. *amongst the hyperplanes satisfying item 1, $\mathcal{H}$ is the hyperplane which distance $\mathcal{D}^{(2)}$ to its second closest vertice $\mathbf{s}^{(2)}$ is again greater than or equal to the other corresponding distances;*
3. *and so on with every remaining vertice.*

The following theorem establishes the fact that an optimal hyperplane necessarily fits the symmetries of the TLF it implements. This means that it gives the same value to the respective weights.

Theorem 1 *Let $\mathcal{H}^{(o)}$ be an optimal hyperplane defined by $\{\mathbf{w}^{(o)}, Th^{(o)}\}$ for a given TLF. If this TLF is partially symmetrical in two distinct inputs s_i and s_j (with $i \neq j$), then $w_i^{(o)} = w_j^{(o)}$.*

The proof of this theorem is given in [11]. It relies on the fact that if an optimal hyperplane, which does not satisfy Th. 1, is found, then it can systematically be turned into a more robust hyperplane under the meaning of Def. 5 thanks to the weight alteration of the Eq. (17).

The consequence of Th. 1 thus is that the closest an hyperplane $\mathcal{H}$ is from being optimal, the closest its descriptor $\mathbf{n}$ is to the real symmetries of the implemented TLF, and reciprocally. The optimization process hence consists in the two following stages:

1. identify any non-implemented symmetry between two or more reduced inputs;
2. reduce them using Eq. (17).

This procedure is called: *generative set reduction*.

4.3. *Optimization algorithm*

The first stage of the generative set reduction is to check the symmetry of every pair of reduced inputs k_i, k_j, $i \neq j$, using the generative set of the TLF which implementation is to be optimized. Testing every combination leads to $c(c-1)/2$ tests. The real number of tests is actually much lower than this. Indeed, having the weights sorted implies that if k_i and k_{i+j} were symmetrical, then so would k_i, k_{i+1}, k_{i+2}, ... and k_{i+j} be. As a result, one only needs to check k_c with k_{c-1}, k_{c-1} with k_{c-2} and so on until k_2 is finally checked with k_1. In this manner, the number of symmetry tests is reduced to $c-1$ which is linear in the number of reduced inputs.

The reduction of a generative set is achieved through the Th. 2, using the following notations and terms:

$$\{\mathcal{F}\} = \left\{ \begin{array}{c} (k_{1,c}, \ldots, k_{1,2}, k_{1,1}) \\ (k_{2,c}, \ldots, k_{2,2}, k_{2,1}) \\ \vdots \\ (k_{q,c}, \ldots, k_{q,2}, k_{q,1}) \end{array} \right\} = \left\{ \begin{array}{c} \mathbf{k}_1 \\ \mathbf{k}_2 \\ \vdots \\ \mathbf{k}_q \end{array} \right\}, \tag{18}$$

where q is number of generative vectors included in the set, and c is the dimension of the reduced input space.

In Eq. (18), we call the j^{th} *i-order root* of $\{\mathcal{F}\}$ the head $(k_{j,c} \ldots k_{j,c-i+1})$ obtained from the generative vectors $\mathbf{k}_j$ with $1 \leq i \leq c$ and $1 \leq j \leq q$.

To each i-order root value is associated a *i-order sub-generative set* extracted from $\{\mathcal{F}\}$. It is made of the generative vectors which have a same i-order root value, and with their root part removed.

For example, the 2-order roots and subgenerative sets of the generative set (15) are highlighted by Fig. 3.

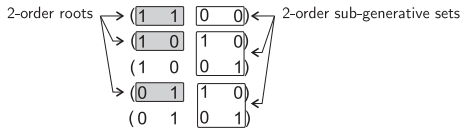

Fig. 3. 2-order roots and sub-generative sets of the TLF of Eq. (15).

Theorem 2 (Generative set reduction) *Let $\{\mathcal{F}\}$ denote a generative set with the notations of the Eq. (18). The two reduced inputs k_i and k_{i-1} are symmetrical if and only if for all j, the $(c-i)$-order sub-generative sets $\{\mathcal{F}_j\}$ each satisfy:*

test 1 *every possible integer value of the sum $(k_{j,i} + k_{j,i-1})$, from the smallest which appears in $\{\mathcal{F}_j\}$ to the greatest one, indeed appear in $\{\mathcal{F}_j\}$;*

test 2 *for each one of these sum values, every possible combination of values $k_{j,i}$ and $k_{j,i-1}$, giving the right sum appear in $\{\mathcal{F}_j\}$;*

test 3 *the $(c-i+2)$-order sub-generative sets of $\{\mathcal{F}_j\}$ attached to a same value of $(k_{j,i} + k_{j,i-1})$ are identical.*

A formal proof of this theorem is provided in [11]. It is the translation into the reduced input space of Def. 1.

Once the symmetrical reduced inputs have been detected, the optimized weights can be computed using Eq. (17). The reduction algorithm thus is:

```
function reduction(w(c...1),n(c...1),GSet)

% symmetry between k_i and k_j
for i=c:2
  j = i−1; % index of the next reduced input
  pass = true;

  % extract every sub−generative sets
  % and check them with Th. 2
  for subGSet = each(subGenerativeSet(c−i,GSet))
    pass = pass && test1(subGSet);
    pass = pass && test2(subGSet);
    pass = pass && test3(subGSet);
  end;

  % reduction of the two weights if needed (Eq (17))
  if (pass)
    (w(i),w(j)) −> (n(i)∗w(i)+n(j)∗w(j))/(n(i)+n(j));
    (n(i),n(j)) −> n(i)+n(j);
  end;
end;
return(w);

end function;
```

4.4. *Illustrative example*

In order to illustrate the optimization process, we propose to synthesize a more robust cloning template from the template used in the previous example (see section 3.4). Its analysis resulted in the generative set given in (15) and recalled hereafter:

$$\{\mathcal{F}\} = \left\{ \begin{array}{c} (1,1,0,0) \\ (1,0,1,0) \\ (1,0,0,1) \\ (0,1,1,0) \\ (0,1,0,1) \end{array} \right\}. \qquad \text{(copy of Eq. (15))}$$

with $c = 4$ reduced variables, $c - 1 = 3$ symmetry tests are thus needed. Applying the optimization algorithm to this generative set is detailed thereafter.

Symmetry k_4 with k_3

Sub-generative sets extraction There is only one 0-order sub-generative set to deal with: the generative set itself.

Only sub-generative set Merging k_4 with k_3 according to Eq. (16) yields the potential reduced input $k_{4,3} = k_4 + k_3$. Its value respectively is 2, 1, 1, 1 and 1 for each one of the five generative vectors of Eq. (15):

test 1 There is no missing integer between the two values $\max(k_{4,3}) = 2$ and $\min(k_{4,3}) = 1$.

test 2 The only way to obtain 2 is $k_4 = 1/k_3 = 1$ which achieved by the first generative vector of Eq. (15); whereas the two ways to obtain 1 are $k_4 = 1/k_3 = 0$ and $k_4 = 0/k_3 = 1$ which are achieved by at least one generative vector each.

test 3 There is only one 2-order sub-generative set associated with $k_4 = 1/k_3 = 1$ and it is therefore equal to itself; the 2-order sub-generative set associated with $k_4 = 1/k_3 = 0$ and $k_4 = 0/k_3 = 1$ are the same.

Reduction of k_4 and k_3 The two reduced inputs are symmetrical and their weights are reduced to the value $w_{3,4} = \frac{1}{2}(w_4 + w_3) = 5.25$. The generative set obtained after the reduction and with the duplicate generative vectors removed is:

$$\{\mathcal{F}\}' = \left\{ \begin{array}{c} (2,0,0) \\ (1,1,0) \\ (1,0,1) \end{array} \right\}. \qquad (19)$$

Symmetry k_3 with k_2

Sub-generative sets extraction There is only one 0-order sub-generative set to deal with: the generative set itself.

Only sub-generative set Merging k_3 with k_2 according to Eq. (16) yields the potential reduced input $k_{3,2} = k_3 + k_2$. Its value respectively is 2, 2 and 1 for each one of the three generative vectors of the generative set (19).

test 1 There is no missing integer between the two values $\max(k_{3,2}) = 2$ and $\min(k_{3,2}) = 1$.

test 2 The two ways to obtain 2 are $k_3 = 2/k_2 = 0$ and $k_3 = 1/k_2 = 1$ and they are achieved by the first two generative vectors; whereas the two ways to obtain 1 are $k_3 = 1/k_2 = 0$ and $k_3 = 0/k_2 = 1$ but the second possibility is missing. The symmetry test thus fails.

test 3 Not necessary.

Reduction of k_3 and k_2 The two reduced inputs are not symmetrical and their weights cannot be reduced.

Symmetry k_2 with k_1

Sub-generative sets extraction The two 1-order sub-generative sets are:

$$\{\mathcal{F}_1\}' = \{ \ (0,0) \ \} \qquad (20)$$

$$\text{and } \{\mathcal{F}_2\}' = \left\{ \begin{array}{c} (1,0) \\ (0,1) \end{array} \right\}. \qquad (21)$$

First sub-generative set Merging k_2 with k_1 according to Eq. (16) yields the potential reduced input $k_{2,1} = k_2 + k_1$. Its only value is 0.

test 1 There is no missing integer between the two values $\max(k_{2,1}) = 0$ and $\min(k_{2,1}) = 0$.

test 2 The only way to obtain 0 is $k_2 = 0/k_1 = 0$ and it is achieved by the only generative vector.

test 3 There is no 2-order sub-generative sets, they are thus equal.

Second sub-generative set Merging k_2 with k_1 according to Eq. (16) yields the potential reduced input $k_{2,1} = k_2 + k_1$. Its only value is 1.

test 1 There is no missing integer between the two values $\max(k_{2,1}) = 1$ and $\min(k_{2,1}) = 1$.

test 2 The only two ways to obtain 1 are $k_2 = 1/k_1 = 0$ and $k_2 = 0/k_1 = 1$; they are both achieved by at least one generative vector.

test 3 There is no 2-order sub-generative sets, they are thus equal.

Reduction of k_2 and k_1 The two reduced inputs are symmetrical in both $\{\mathcal{F}_1\}'$ and $\{\mathcal{F}_2\}'$ and therefore, so are they in $\{\mathcal{F}\}'$. Their weights are thus reduced to the value $w_{2,1} = \frac{1}{2}(w_2 + w_1) = 1.75$ and the obtained generative set is:

$$\{\mathcal{F}\}'' = \left\{ \begin{array}{c} (2,0) \\ (1,1) \end{array} \right\}. \qquad (22)$$

Finally, the cloning template from the subsection 3.4 finally gives the following optimized template:

$$\mathbf{B}^{\text{robust}} = \left[\begin{array}{ccc} 1.75 & 1.75 & 5.25 \\ 5.25 & 3 & 0 \\ 0 & 0 & 0 \end{array} \right], \qquad (23)$$

$\mathbf{A}$ and z remaining unchanged.

4.5. *Discussion*

All these operations have been implemented in a C++ application which can be used to evaluate the computational costs of the processing chain analysis/reduction described in this paper. For example, templates with four inputs (like the one used in the previous example) approximately take 13 ms to be optimized whereas templates with 16 inputs need 693 ms. These durations are mean values over a great number of templates and were executed on a Pentium Centrino Duo 1.8 GHz. The exact

processing time depends on the complexity of the threshold logic function, i.e., on its number of generative vectors as well as on the number of symmetries.

We also applied the proposed technique to the templates listed in the CNN bank presented in [6]. A wide range of templates were improved. Indeed, over the 20,097 templates which were published, 17,799 were successfully optimized in an overall time of 2 minutes and 22 seconds.

5. Conclusion

This paper introduces a method for the re-implementation of threshold logic functions applied to CNNs. Based on a set of mathematical theories, the paradigm of generative sets is developed as a new representation of linearly separable Boolean functions. Thanks to it, the symmetries of any TLF can be detected more efficiently than with a truth table approach and their CNN implementation can thus be optimized against internal noise through symmetry reduction. To this effect, this paper provides the two core algorithms involved in the overall optimization process: the first one extracts the generative set from a correctly operating CNN template while the second detects symmetries in the generative set structure and reduces them to lead to a more robust CNN template.

Computer simulations have shown that most of the existing CNN templates can be efficiently rewritten with the proposed method. As a result, any template designed for a specific task can be optimized with our approach. The result is a template more robustness to internal noise.

References

1. L.O. Chua and L. Yang, "Cellular neural networks: Theory," *IEEE Trans. on Circuits and Systems*, **35**(10), 1257–1272 (1988).
2. L.O. Chua and L. Yang, "Cellular neural networks: Applications," *IEEE Trans. on Circuits and Systems*, **35**(10), 1273–1290 (1988).
3. D. Monnin, A. Köneke, and J. Hérault, "Boolean design of binary initialized and coupled CNN image processing operators," *Proc. IEEE Intl. Work. on Cellular Neural Networks and their Appl.*, Frankfurt, Germany, 124–131 (2002).
4. F. Chen, G. He, and G. Chen, "Realization of Boolean functions via CNN: Mathematical theory, LSBF and template design," *IEEE Trans. on Circuits and Systems I*, **53**(10), 2203–2213 (2006).
5. C. Chow, "On the characterization of threshold functions," *Proc. Symposium on Switching Circuit Theory and Logical Design*, New York, USA, 34–38 (1961).
6. F. Chen and G. Chen, "Realization and bifurcation of Boolean functions via cellular neural networks," *Intl. J. of Bifurcation and Chaos*, **15**(7), 2109–2129 (2005).
7. P.D. Picton, "Deriving weights for single threshold logic gates using decomposition," *Proc. IEEE Intl. Conf. on Neural Information Processing*, Shanghai, China, (2007).
8. P. Celinski, G.D. Sherman, J.F. Lopez, and D. Abbott, "A mapping technique for the synthesis of linear threshold gate networks used to implement Boolean functions," *Proc. WSES Intl. Conf., Neural Networks and applications*, Puerto de la Cruz, Tenerife, Spain, 4251–4255 (2001).
9. M. Hänggi and G.S.Moschytz, *Cellular neural networks: Analysis, design and optimization* (Kluwer Academic Publishers, Norwell, MA, USA, 2000).
10. Y. Bénédic, D. Monnin and J. Mercklé, "Fast analysis method for both coupled and uncoupled binary-output cloning templates," *Proc. IEEE Intl. Biannual Workshop on Cellular Neural Networks and their Applications*, Budapest, Hungary, 184–189 (2004).
11. Y. Bénédic and J. Mercklé, "Optimisation of binary-output CNNs: First step of an analytical design process," *Proc. IEEE Intl. Joint Conf. on Neural Networks*, Vancouver, Canada, 5107–5113 (2006).
12. Z. Kohavi, *Switching and finite automata theory* (Mc Graw Hill, New Delhi, India, 2nd edition, 1978).