CHAPTER – IV

TIME-MULTIPLEXING SCHEME FOR CELLULAR NEURAL NETWORKS BASED IMAGE PROCESSING

4.1 INTRODUCTION

Since its invention in 1988, Cellular Neural Networks have captured the attention of the scientific community partly due to their potential in image processing applications [13, 15, 16, 34]. Enormous advances have been made by many researchers in this field [21, 67]. Some among them are, the development of image processing work for the creation of templates capable of several image processing applications and the development of special purpose CNN algorithms such as halftoning and character recognition [55, 78]. Because of the threshold activation function at the output of the CNN structure, many applications have been traditionally based on black and white images. However, colour-based applications are emerging as well.

From a hardware point of view, the local interconnectivity of the array lends itself to practical VLSI implementations. Everything seems to indicate that CNN will assume the role of a new type of ‘analogic processor’ [20, 77]. In fact, CNN sets the platform for a new algorithmic style based on the spatiotemporal properties of the array. The key elementary instruction is a spatiotemporal transient generated by a two-dimensional nonlinear dynamic processor array. This basic instruction resembles the typical convolutional operator used in image processing applications.
While software prototypes prove the potential of CNN [50 - 51], a great deal of research has been advocated to hardware implementations, which can be used for on-line applications in real-time [8, 28 – 30, 35 - 36, 65, 75, 94 – 95, 99-100]. Unfortunately, even though these implementations are extremely efficient the problem of how to use them for processing large images has not been properly addressed. For practical image size applications, due to current state-of-the-art technological limitations, it is impossible to have a one-to-one mapping between the CNN hardware processors and all the pixels in the image involved. Thus it is a key issue for the proper use of these implementations in common-day situations. This chapter presents a practical solution by processing the input image block by block, with the number of pixels in a block being the same as the number of CNN processors in the hardware.

### 4.2 SOFTWARE BASED CELLULAR NEURAL NETWORK IMAGE PROCESSING

Consider an $M \times N$ cellular neural network, having $M \times N$ cells arranged in $M$ rows and $N$ columns, see Fig. 4.1(a). The basic unit of a CNN is called a cell [14, 15]. Any cell on the $i^{th}$ row and $j^{th}$ column, $C(i,j)$, is connected only to its neighbour cells, i.e. adjacent cells interact directly with each other. This *neighbourhood* is denoted as $N(i,j)$. For hardware implementations, due to the wiring complexity involved, typical neighbourhoods are of radius 1, although, for software simulations radius of 3 or more have been reported. It is to be noted that cells not in the immediate neighbourhood have an indirect effect because of the propagation effects of the dynamics of the network. Each cell has a state $x$, a constant external input $u$, and output $y$. The equivalent block diagram
of a continuous-time cell is shown in Fig. 4.1(b). The first order nonlinear differential equation defining the dynamics of a cellular neural network cell can be written as follows

\[
C \frac{dx_{ij}}{dt} = -\frac{1}{R} x_{ij}(t) + \sum_{C(k,l)\in N(i,j)} A(i,j;k,l) y_{kl}(t) + \sum_{C(k,l)\in N(i,j)} B(i,j;k,l) u_{kl} + I \tag{4.1}
\]

\[
y_{ij}(t) = \frac{1}{2} \left( |x_{ij}(t)| + 1 - |x_{ij}(t)| \right) \tag{4.2}
\]

\[\text{Figure 4.1 Cellular Neural Network architecture. (a) Array; (b) CNN cell}\]

where \(x_{ij}\) is the state of cell \(C(i,j)\), \(x_{ij}(0)\) is the initial condition of the cell, \(C\) is a linear capacitor, \(R\) is a linear resistor and \(I\) is an independent current source. In an actual circuit implementation, the integration process is not instantaneous and will depend on a time constant \(\tau = RC\). \(A(i,j;k,l)\) and \(B(i,j;k,l)\) are space invariant programming
templates for all cells $C(k,l)$ in the neighbourhood $N(i,j)$ of cell $C(i,j)$; $u_{ij}$ represents the external input and $y_i$ represents the output equation, i.e., the activation function for the cell. Notice from the summation operators that each cell is affected by its neighbour cells. $A(.)$ acts on the output of neighbouring cells and is referred to as the feedback operator. $B(.)$ in turn affects the input control and is referred to as the control operator. Specific entry values of matrices $A(.)$ and $B(.)$ are application dependent and space invariant. The matrices are also known as cloning templates [14-15, 21]. A constant bias, $I$, and the cloning templates determine the transient behaviour of the cellular neural network. If the system satisfies that the center element of template $A$ is greater than one, i.e., $A_{nn} > 1$, then the settled state values will converge to absolute values greater than one, i.e., $|x_0| \geq 1$.

When we consider equation (4.1), it is space invariant, which means that $A(i,j;k,l) = A(i-k,j-l)$ and $B(i,j;k,l) = B(i-k,j-l)$ for all $i, j, k, l$. Therefore, the image transformation process can be seen as a scanning procedure in which the pixels are mapped one-to-one to the CNN cells. The basic approach is to imagine a square subimage area centered at $(x,y)$, with the subimage being the same size of the templates involved in the simulation. The center of this subimage is then moved from pixel to pixel starting at the top left corner and applying the $A$ and $B$ templates at each location $(x,y)$ to solve the differential equation. This procedure is repeated for each time step, for all the pixels. We denote an instance of this image scanning-process as an ‘iteration’. The processing stops when it is found that the states of all CNN cells have converged to steady-state values, and when the outputs of its neighbour cells are saturated, e.g., they
have ± 1 value. This whole simulating approach is referred to as *raster simulation*. It has to be noted that when the templates are located on the border of the input image, the scanning process does not involve all of its elements. To deal with this border effect a center zero padded superposition model is used. That is to say, a virtual set of border cells, initialized to zero state values are created.

From the hardware implementation's point of view, this is a very exhaustive approach. For practical applications, in the order of 2,50,000 pixels, the hardware would require an enormous amount of processors, which would make its implementation unfeasible. An alternative to this scenario is to *multiplex* the image-processing operator.

Three of the most widely used Numerical Integration Algorithms are used in CNN time-multiplexing simulation which are the Euler's Algorithm, RK-Gill Algorithm (discussed by Oliveria [66]) and the RK-Butcher Algorithm (discussed by Badder [6, 7] and Murugesan et al [60 – 64]).

4.3 TIME-MULTIPLEXING SIMULATION

Under this approach one can define a block of pixels (subimage), which will be processed by an equal number of CNN cells. Once convergence is achieved and a new subimage adjacent to the one just processed, is scheduled for further processing. This procedure is repeated until the whole image has been scanned using a lexicographical order, say, from left to right and from top to bottom. It is obvious that with this approach the processing of large images becomes feasible in spite of the finite number of CNN cells. Even though
the approach seems simple and appealing, an important observation is necessary: *the processed border pixels in each subimage may have incorrect values since they are processed without neighbouring information.* As an example, let us consider the case of Fig. 4.2 in which a $4 \times 4$ CNN is processing one section of an image. Let us assume further that all the entries of the $B$ template are ones. Then under the conditions depicted, cell $C(3,4)$ will present a 50% computation error if the pixels surrounding the border are not considered. This can be concluded by observing that the total weighted input to the state of cell $C(3,4)$, without considering border pixels, is 6 units instead of 3. Fortunately, the latency of CNNs is such that only local interactions, depending upon the neighbourhood radius, are important. Hence, to cope with the previous problem, two sufficient conditions must be considered to ensure that each border cell properly interacts with its neighbours. These conditions are: (i) to have a *belt of pixels* from the original image around the subimage being processed; and (ii) to have *pixel overlaps* between adjacent subimages.

![Figure 4.2](image)

*Figure 4.2 Error quantification due to missing neighbour pixels using only the $B$ template*

\[ e_\phi^b = \sum_{i=1}^{i=3} b_{i,j+1} \text{sign}(u_{i,j}) = -3; \]
which induces a 50% error in the computations
4.4 NECESSARY AND SUFFICIENT CONDITIONS FOR TIME-MULTIPLEXING

It is possible to quantify the processing error of any border cell $C(i,j)$ within a given neighbourhood radius. Let us compute independently the error due to the feedforward operator and then due to interactions among cells for two horizontally adjacent processing blocks. Let us assume that the neighbourhood for a border cell is incomplete, e.g. we are missing the data coming from pixels out of the CNN array. Then, the absolute processing error of a border cell $C(p,q)$ due only to the effect of the $B$ template is obtained by subtracting the erroneous state value from the error-free states using equation (4.1). The erroneous state value is characterized by the absence of external input signals to the cell. This yields the following result

$$\varepsilon_y^g = \sum_{C(k,l) \in N(p,q)} B^*(p,q;k,l) \text{sign}(u_{k,l})$$

(4.3)

where $B^*(p,q;k,l)$ are the missing entries from the $B$ template due to the absence of input signals $u_{k,l}$ and $\text{sign}(.)$ is the sign function. The latter function is used to represent the status of a pixel, e.g. \texttt{black} = 1 and \texttt{white} = -1. It is to be noted that the error is both image and template dependent. In other words, the steady state of a border cell may converge to an incorrect value due to the absence of its neighbours weighted input. One can easily conclude that the error is canceled if the missing external inputs are provided to the border cells as depicted in Fig. 4.3 (a). Since, the array is ‘embedded’ in the image during operation, this condition can easily be satisfied.
Figure 4.3 Conditions for Time Multiplexing operation. (a) Belt of inputs; (b) Overlapped pixels

Now, let us address the interactions among cells. For this effect, we can compute the absolute error in a similar form. Disregarding for the moment the $B$ template this error is

$$
\varepsilon_{ij}^A = \sum_{c(k,l) \in N(r,a)} A^*(p,q;k,l) y_{k,l}(t)
$$

(4.4)

where $A^*(p,q;k,l)$ are the missing entries from the $A$ template due to the absence of weighted output signals $y_{k,l}(t)$. In this situation, the problem is more complex because the output signals depend on the state of their corresponding cells. To minimize the error
an overlap of pixels between two adjacent blocks is proposed. In this form, the inner cells of the CNN array will always receive weighted processing information from the border cells.

The general time-multiplexing procedure consists of processing adjacent image blocks, one after another, in a lexicographical order. Each block with converged cells will have state output variables $y$ that are the values used for the final output image. Every time that a new sub image is being processed, the physical CNN array is initialized to the initial conditions of the original image, or black or white as required by the template in use. In the overlapping procedure the outer overlapped cells' and its converged values are discarded since they have been computed with incomplete neighbouring information. Only the inner cells' converged values are kept as valid values. This implies that for a neighbourhood radius of 1, an overlap of two pixel column/rows is needed to be able to ensure correct values for pixels assigned to the border cells. For instance, in Fig. 4.4(b) two pixel columns are overlapped. The converged values of the leftmost column are kept when processing $Block_i$ and the rightmost column values when processing $Block_{i+1}$. To illustrate the previous statements, consider the image depicted in Fig. 4.4(a) and assume an edge detection operation. For this image processing operation the middle pixel is turned white only when its two neighbours are black. If only one pixel overlap is assumed the center pixel will never be white because when it is processed in both blocks it still misses the neighbouring information, see Fig. 4.4(b). This situation is fixed by overlapping two pixels. In particular, the three black pixels will be present when processing $block_{i+1}$, see Fig. 4.4(c).
Figure 4.4 (a) Pixel overlap; shadowed pixels represent the ones that are overlapped in two consecutive block scans. In an edge detection operation a black pixel is turned white only when it has two adjacent black neighbour pixels. (b) 1 pixel overlap; the left neighbour is missing which results in a wrong edge detection result; (c) 2 pixel overlaps; both adjacent neighbours are present.

For the purpose of understanding the overall idea of this simulation approach better, the simplified algorithm is presented below:

Algorithm: (Time-Multiplexing CNN simulation)

\[ B = \{ C_{i,j} \mid i = 1,\ldots,\text{block}_x \land j = 1,\ldots,\text{block}_y \} \]

\[ P \subset B = \text{set of border cells (lower left corner)} \]

\[ \text{overlap} = \text{number of cell overlaps} \]

\[ \text{belt} = \text{width of input cells} \]

\[ M = \text{number of rows of the image} \]

\[ N = \text{number of columns of the image} \]

for \((i=0; i<M; i+=\text{block}_x - \text{overlap})\)

for\((j=0; j<N; j+=\text{block}_y - \text{overlap})\)

\{

/* load initial conditions for the cells in the block except for those in the borders */

for \((p=-\text{belt}, q<\text{block}_x+\text{belt}; p++)\)

for\((q=-\text{belt}, q<\text{block}_y+\text{belt}; q++)\) {

\[ x_{i+p,j+q}(t) = \begin{cases} u_{ij} & \forall C_{i+p,j+q} \in B \\ -1 & \text{otherwise} \end{cases} \]

} /*end for*/
/* if all the block is white or black don’t process it */
if (\(x_{i+p,j+q} = -1 \lor x_{i+p,j+q} = 1\ \forall c_{i+p,j+q} \in B\))
{
    Obtain the final states from memory
    continue;
}
do { /* normal raster simulation */
    for (p=0; p<block_x; p++) {
        for (q=0; q<block_y; q++)
        {
            /* calculation of the next state excluding the belt of inputs */
            \(x_{i+p,j+q}(t_n) = x_{i+p,j+q}(t_{n-1}) + \frac{1}{\tau} \int_{t_{n-1}}^{t_n} \langle (x_{i+p,j+q}(t))dt\)

            \(\forall c_{i+p,j+q} \in B\)

            /* convergence criteria */
            if \(\frac{dx_{i+p,j+q}(t_n)}{dt} = 0\) and \(y_{kl} = \pm 1\)
            \(\forall \langle k,l \rangle \in N_{i+p,j+q}\) {
                converged_cells++;
            } /* end for */
    } /* end for */
    /* update the state values */
    \(x_{i+p,j+q}(t_{n+1}) = x_{i+p,j+q}(t_n + 1)\ \forall c_{i+p,j+q} \in B\);
    } while (converged_cells < (block_x*block_y));

    /* store new state values excluding the ones corresponding to the border cells */
    A ← \(x_y \forall C_y \in B \setminus P\)
} /* end for */
4.5 SIMULATION RESULTS AND COMPARISONS

All the simulation reported here are performed using a SUN BLADE 1500 workstation, and the simulation time used for comparisons is the actual CPU time used. The input image format is the bitmap format (xbm), which is commonly available and easily convertible from popular image formats like GIF or JPEG. Using actual numbers, one can easily show how much improvement is achieved. The size of Fig. 4.5(a) is $355 \times 400$ (1,42,000 pixels), and an Averaging template is used for simulation comparisons. First, using the raster CNN simulator, the simulation took 200.42 seconds. Next, with the regular time-multiplexing simulator (with overlapping and input belt) the simulation took 342.28 seconds. Finally, the time-multiplexing with the time-saving scheme performed the same simulation in 240.20 seconds, almost a 33% improvement from the regular time-multiplexing. The size of two dimensional window of $10 \times 10$, with two column overlapping was used. It may be noted that this algorithm maintains all the edges of the original one.

![Figure 4.5](image)

*Figure 4.5* (a) Original image (b) After Averaging Template using Time-Multiplexing simulation
Since speed is one of the main concerns in the simulation, finding the maximum step size that still yields convergence for a template can be of help in speeding up the system. The speed-up can be achieved by selecting an appropriate step size $\Delta t$ for that particular template. Even though the maximum step size may slightly vary from one image to another, the values in Fig. 4.6 serve as a good reference for step size comparison. These results were obtained by trial and error over more than 100 simulations on a diamond figure. If the step size chosen is too small, the simulation might take much iteration; hence, it will take longer time to achieve convergence.

![Figure 4.6 Maximum step size for three different templates](image)

On the other hand, if the step size taken is too large, it might not converge at all or it would converge to erroneous steady state values (beyond step size 5); the latter remark can be observed for the Euler and RK-Gill algorithm which is plotted in Fig. 4.7. Hence, the speed of convergence of RK-Butcher Algorithm for large step size is much faster than Euler and RK-Gill algorithms.
The results of Fig. 4.8 were obtained by simulating a small image of size $16 \times 16$ (256 pixels) using Edge Detection template on a diamond figure.

**Figure 4.7** Simulation graph for time comparison using Edge Detection template

**Figure 4.8** CPU performance (SUN BLADE 1500 work station) for distinct image sizes in number of pixels
Simulation time computations are shown in Fig 4.8 using an Averaging template for images of sizes about 2,50,000 pixels and it is observed from Fig. 4.8, the simulation time will increase if the number of pixels chosen is too large in a log linear fashion.

4.6 CONCLUSION

Versatile algorithms have been developed using numerical integration algorithms for simulating CNN with Time-Multiplexing scheme. This Time Multiplexing algorithm is a very simple one and a powerful method for developing image processors using CNN. In fact, among all the three numerical integration algorithms, the one developed using RK-Butcher algorithm is performing very efficiently for solving this problem. The notable observation is Time Multiplexing CNN with Numerical Integration always converges for a larger step size, which in turn results in lowest simulation time for any given image size. For a given step size, the convergence time of this algorithm is log linear for all large size images and this is an additional attractive feature when we deal with high resolution / large images. This algorithm always preserves the edges as given in the original picture in addition and it enhances the picture quality.