Single-layer CNN simulator

Single–Layer CNN Simulator

Chi–Chien Lee and José Pineda de Gyvez

Department of Electrical Engineering,
Texas A&M University,College Station, Texas 77843–3128 USA

ABSTRACT

An efficient behavioral *simulator* for *Cellular Neural Networks* (CNN) is presented in this paper. The simulator is capable of performing *Single–Layer* CNN simulations for any size of input image, thus a powerful tool for researchers investigating potential applications of CNN. This paper reports an efficient *algorithm* exploiting the latency properties of Cellular Neural Networks along with numerical integration techniques; simulation results and comparisons are also presented.

BACKGROUND AND TECHNICAL RATIONALE

CNN is a hybrid of *Cellular Automata* and *Neural Networks* (hence the name Cellular Neural Networks), and it shares the best features of both worlds. Like Neural Networks, its continuous time feature allows real–time signal processing, and like Cellular Automata, its local interconnection feature makes VLSI realization feasible. Its grid–like structure is suitable for the solution of a high order system of first order non–linear differential equations on–line and in real–time . CNN is an analog nonlinear dynamic processor array, see Fig. 1a, characterized by the following features [2]:

1) Each analog processor is capable of processing continuous signals, in either continuos–time or discrete–time modes.
2) The processors are placed on a 3D geometric cellular grid (several 2D layers) and are basically identical.
3) Interaction among processors is local and mainly translation invariant.
4) The mode of operation may be transient, equilibrium, periodic, chaotic, or combined with logic (without A/D conversion).

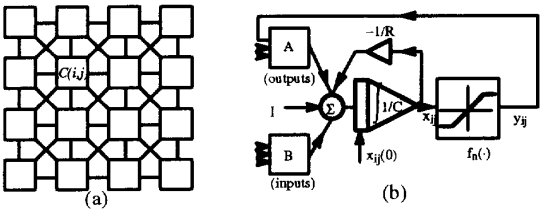

Fig. 1. CNN Structure and block diagram.

The basic circuit unit of CNN is called a *cell* [3]. It contains linear and nonlinear circuit elements. Any cell, *C(i,j)*, is connected only to its neighbor cells, i.e. adjacent cells interact directly with each other. This intuitive concept is called *neighborhood* and is denoted as *N(i,j)*. Cells not in the immediate neighborhood have indirect effect because of the propagation effects of the dynamics of the network. Each cell has a state x, input u, and output y. The state of each cell is bounded for all time $t > 0$ and, after the transient has settled down, a cellular neural network always approaches one of its stable equilibrium points. This last fact is relevant because it implies that the circuit will not oscillate. The dynamics of a CNN has both output feedback *(A)* and input control *(B)* mechanisms. The first order nonlinear differential equation defining the dynamics of a cellular neural network cell can be written as follows

$$C\frac{dx_{ij}(t)}{dt} = -\frac{1}{R}x_{ij}(t) + \sum_{C(k,l)\in N(i,j)} A(i,j;k,l)\, y_{kj}(t) + \sum_{C(k,l)\in N(i,j)} B(i,j;k,l)\, u_{kl} \qquad (1)$$

$$y_{ij}(t) = \frac{1}{2}(|x_{ij}(t) + 1| - |x_{ij}(t) - 1|)$$

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, I is an independent current source, $A(i,j;k,l)\ y_{kl}$ and $B(i,j;k,l)\ u_{kl}$ are voltage controlled current sources for all cells *C(k,l)* in the neighborhood *N(i,j)* of cell *C(i,j)*, and y_{ij} represents the output equation.

Notice from the summation operators that each cell is affected by its neighbor cells. $A(\cdot)$ acts on the output of neighboring cells and is referred to as the *feedback operator*. $B(\cdot)$ in turn affects the input control and is referred to as the *control operator*. Specific entry values of matrices $A(\cdot)$ and $B(\cdot)$ are application dependent, are space invariant and are called *cloning templates*. A current bias I and the cloning templates determine the *transient behavior* of the cellular nonlinear network. The equivalent block diagram of a continuos–time cell implementation is shown in Fig. 1b.

CNNs have as input a set of analog values and its programmability is done via cloning templates. Thus, *programmability* is one of the most attractive properties of CNNs, but how to choose the optimal network and how to program it to perform a given task are still topics under investigation. This is the reason why there is a need for a

behavioral CNN simulator capable of helping investigators design and manipulate cloning templates ("programming"). Existent tools are not meant to deal with a significant number of pixels typical in common image processing applications[6]. The simulator presented here not only satisfies this need, but it also can be used for *testing* CNN hardware implementations.

BEHAVIORAL SIMULATION

Recall that equation (1) 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 solution of the system of difference equations can be seen as a convolution process between the image and the CNN processors. 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, say, at the top left corner and applying the A and B templates at each location (x,y) to solve the differential equation, see Fig. 2. This procedure is repeated for each time step, for all the pixels. An instance of this image scanning–processing is referred to as an "iteration". The processing stops when it is found that the states of all CNN processors have converged to steady–state values [3], and the outputs of its neighbor cells are saturated, e.g. they have a ± 1 value.

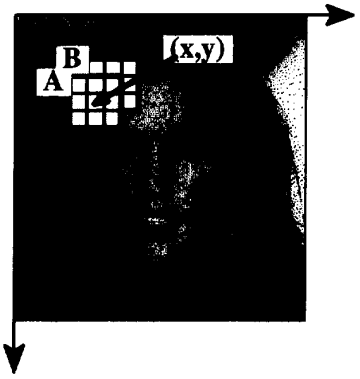

Fig. 2 Raster simulation approach

This whole simulating approach is referred to as *raster simulation*. A simplified algorithm is presented below for this approach. The part where the integration is involved (i.e. calculation of the next state) is explained in the Numerical Integration Methods section.

Algorithm: (Single–Layer or Raster CNN simulation)

Obtain the input image, initial conditions and templates from user;
/* M,N = # of rows/columns of the image */
while (converged_cells < total # of cells) {
for (i=1; i<=M; i++)
  for (j=1; j<=N; j++) {
    if (convergence_flag[i][j])
      continue; /* current cell already converged */
  /* calculation of the next state*/

$$x_{ij}(t_{n+1}) = x_{ij}(t_n) + \int_{t_n}^{t_{n+1}} f'(x(t_n))\, dt$$

    /* convergence criteria */
  if ($\frac{dx_{ij}(t_n)}{dt} = 0$ and $y_{kl} = \pm 1,\ \forall\ C(k,l) \in N_r(i,j)$) {
      convergence_flag[i][j] = 1;
      converged_cells++ ;
    }
  } /* end **for** */
  /* update the state values of the whole image*/
  for (i=1; i<=M; i++)
    for (j=1; j<=N; j++) {
      if (convergence_flag[i][j]) continue;
      $x_{ij}(t_n) = x_{ij}(t_{n+1})$;
    }
  #_of_iteration++;
} /* end **while** */

The raster approach implies that each pixel is mapped onto a CNN processor. That is, we have an image processing function in the spatial domain that can be expressed as:

$$g(x,y) = T(f(x,y)) \qquad (2)$$

where $f(\cdot)$ is the input image, $g(\cdot)$ the processed image, and T is an operator on $f(\cdot)$ defined over the neighborhood of (x,y). From hardware implementation's point of view, this is a very exhaustive approach. For practical applications, in the order of 250,000 pixels, the hardware would require an enormous amount of processors which would make its implementation unfeasible. An alternative is to multiplex the image processing operator. A *time–multiplexed* CNN simulator is presented in a companion paper [1].

NUMERICAL INTEGRATION METHODS

The CNN is described by a system of nonlinear differential equations. Therefore, it is necessary to discretize the differential equation for performing behavioral simulation. For computational purposes, a normalized time differential equation describing CNN is used [4]:

$$f'(x(n\tau)) = \frac{dx_{ij}(n\tau)}{d(n\tau)} = -x_{ij}(n\tau) + \sum_{C(k,l)\in N_r(i,j)} A(i,j;k,l)\ y_{kl}(n\tau) + \sum_{C(k,l)\in N_r(i,j)} B(i,j;k,l)\ u_{kl} + I$$

$$y_{ij}(n\tau) = \frac{1}{2}(|x_{ij}(n\tau) + 1| - |x_{ij}(n\tau) - 1|) \qquad (3)$$

where τ is the normalized time. For the purpose of solving the initial–value problem, well established Single–Step methods of numerical integration techniques are used [5]. These methods can be derived using the definition of the definite integral

$$x_{ij}((n+1)\tau) - x_{ij}(n\tau) = \int_{\tau_n}^{\tau_{n+1}} f'(x(n\tau))\ d(n\tau) \qquad (4)$$

Three of the most widely used single–step algorithms are used in the CNN behavioral simulator dscribed here. They are the Euler's algorithm, the Improved Euler Predictor–Corrector algorithm and the Fourth–Order (quartic) Runge–Kutta algorithm. These methods differ in the way they evaluate the integral presented in (4).

Euler's method is the simplest of all algorithms for solving ODEs. It is an explicit formula which uses the Taylor–series expansion to calculate the approximation

$$x_{ij}(n+1)\tau) = x_{ij}(n\tau) + \tau\, f'(x(n\tau)) \qquad (5)$$

The Improved Euler Predictor–Corrector method uses both explicit (predictor) and implicit (corrector) formulae. The integral is calculated by multiplying the stepsize τ with the averaged sum of both the derivative of $x(n\tau)$ and the derivative of the predicted $x_p((n+1)\tau)$ at the next time step:

$$x_{ij}((n+1)\tau) = x_{ij}(n\tau) + \frac{\tau}{2}\,[f'(x(n\tau)) + f'(x_p((n+1)\tau)) \qquad (6)$$

The Fourth–Order Runge–Kutta method is the most costly among the three methods in terms of computation time, as it requires four derivative evaluations per time step. However, its high cost is compensated by its accuracy in transient behavior analysis.

$$x_{ij}((n+1)\tau) = x_{ij}(n\tau) + \frac{k_1^{ij} + 2k_2^{ij} + 2k_3^{ij} + k_4^{ij}}{6} \qquad (7)$$

where

$$k_1^{ij} = \tau\, f'(x_{ij}(n\tau)) \qquad \forall_{C(l,m)\in N_r(i,j)}$$

$$k_2^{ij} = \tau\, f'(x_{ij}(n\tau) + \frac{1}{2}k_1^{ij}) \qquad \forall_{C(l,m)\in N_r(i,j)}$$

$$k_3^{ij} = \tau\, f'(x_{ij}(n\tau) + \frac{1}{2}k_2^{ij}) \qquad \forall_{C(l,m)\in N_r(i,j)}$$

$$k_4^{ij} = \tau\, f'(x_{ij}(n\tau) + k_3^{ij})$$

where $f(\cdot)$ is computed according to (1). There are many single–step methods available to us for this purpose. But, one option worth considering is the combination of two methods in solving for the solution. Since the fourth–order Runge–Kutta is among the most widely used single–step method for starting the solution of the initial–value problem in ODEs, the Predictor–Corrector method for continuing the solution can be combined with the Runge–Kutta starter to make a very efficient computer simulation method for solving the problem.

SIMULATION RESULTS AND COMPARISONS

All the simulations reported here are performed using a Sun SPARC2 workstation, and the simulation time used for comparisons is the actual CPU time used. The input image format for this simulator is the X windows bitmap format (xbm), which is commonly available and easily convertible from popular image formats like GIF or JPEG.

(a) (b)

Fig. 3. Image processing. (a) After Averaging Template (b) After Averaging and Edge Detection Templates

Fig. 3 shows results of the raster simulator obtained from a complex image of 125,535 pixels. For this example an *Averaging* template followed by an *Edge Detection* template were applied to the original image to yield the images displayed in Figs. 3a and 3b, respectively.

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 helpful in speeding up the system. The speed–up can be achieved by selecting an appropriate Δt for that particular template. Even though the maximum step size may slightly vary from one image to another, the values in Fig.4 still serve as good references. These results were obtained by trial and error over more than 100 simulations on a diamond figure.

The importance of selecting an appropriate Δt can be easily visualized in Fig. 5. If the step size chosen is too small, it might take many iterations, hence longer time, to achieve convergence. 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; the latter remark can be observed for the Euler integration method The results of Fig. 5 were obtained by simulating a small image of size 16x16 (256 pixels) using an Edge detection template on a diamond figure. In Fig. 6, simulation time computations using an Averaging template for images of sizes to about 250,000 pixels are shown.

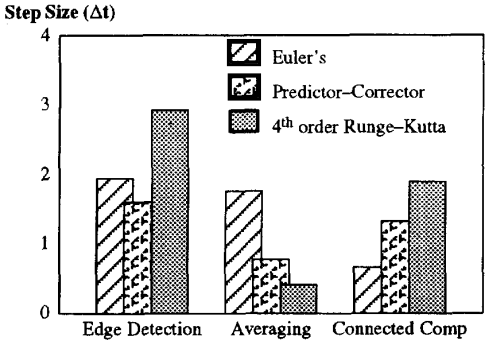

Fig. 4. Maximum step size that still yields convergence for 3 different templates

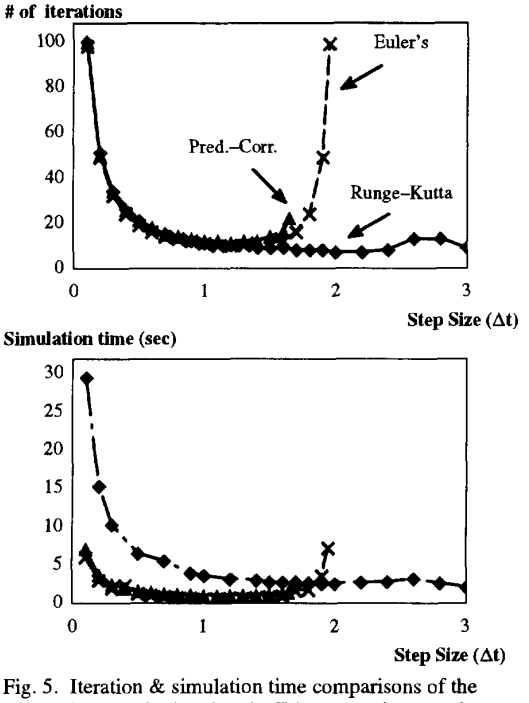

Fig. 5. Iteration & simulation time comparisons of the three methods using the Edge Detection template

CONCLUSION

As researchers are coming up with more and more CNN applications, an efficient and powerful simulator is needed. The simulator hereby presented meets the need in three ways: 1) Depending on the accuracy required for the simulation, the user can choose from three popular methods to perform the numerical integration, 2) The input image format is the X Windows bitmap (xbm), which is commonly available and 3) The input image can be of any size, allowing simulation of images available in common practices.

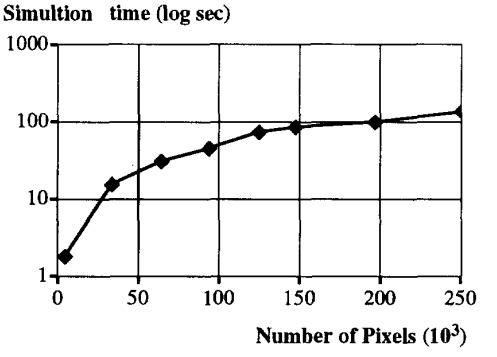

Fig. 6. Simulation time for images of sizes ranging from 64x64 (4096 pixels) to 415x603 (250245 pixels) using Averaging template

REFERENCES

[1] "Time–Multiplexing CNN Simulator", *ISCAS'94*

[2] L.O. Chua and T. Roska, "The CNN Universal Machine Part 1: The Architecture", in *Int. Workshop on Cellular Neural Networks and their Applications (CNNA)*, pp. 1–10, 1992.

[3] L. O. Chua and L. Yang, "Cellular Neural Networks: Theory & Applications," *IEEE Trans. Circuits and Systems*, Vol. CAS–35, pp. 1257–1290, 1988.

[4] J. A. Nossek, G. Seiler, T. Roska and L. O. Chua, "Cellular Neural Networks: Theory and Circuit Design," *International Journal of Circuit Theory and Applications*, Vol. 20, pp. 533–553, 1992.

[5] W. H. Press, B. P. Flannery, S.A. Teukolsky, and W.T. Vetterling, "Numerical Recipes. The Art of Scientific Computing", Cambridge University Press, New York, 1986

[6] J. Varrientos and E. Sanchez–Sinencio, "CELLSIM: A cellular neural network simulator for the personal computer," in Proc. 35th Midwest Symp. Circuits Systs, pp. 1384–1387, 1992.