TOP-DOWN VS. BOTTOM-UP NEUROCOMPUTER DESIGN

JOSEPH T. WUNDERLICH
Elizabethtown College Computer Engineering Program
Elizabethtown, Pennsylvania, USA

Abstract - Artificial neural networks are a form of connectionist architecture where many simple computational nodes are connected in a fashion *similar* to that of biological brains for the purpose of solving problems that require rapid adaptation or where underlying governing equations are not known or cannot be easily computed. This paper first discusses the use of various computer platforms for neurocomputer implementation. Two designs are then presented: (1) An artificial dendritic tree "bottom-up" VLSI chip; and (2) A vector-register microprocessor "top-down" design with on-chip learning and a fully-parallel, entirely-digital implementation.

INTRODUCTION

A discussion of machine intelligence *types* is a good place to begin a neurocomputer design process. Machine intelligence includes both "symbolic" AI and artificial neural networks. Symbolic AI programs use heuristics, inference, hypothesis testing, and forms of knowledge representation to solve problems. This includes "Expert Systems" and programming languages such as Prolog or LISP, with knowledge contained in logic, algorithms, and data structures. A neural network (NN) is a form of connectionist computer architecture (hardware or software) where many simple computational nodes are connected in an architecture *similar* to that of a biological brain. Neurocomputers implement NN's [1]-[3], [5]-[9]. A first step in designing a neurocomputer is choosing an architecture that is either structurally similar to, or merely produces results in a *similar fashion* to the human brain (i.e., "bottom-up" vs. "top-down" design).

Most NN's are <u>top-down</u> designs trained to react to external stimuli. They learn via iterative mathematics to change inter-neuron connection strengths (weights) until outputs converge to desired tolerances. The NN learns such that multiple input/desired-output pairs are satisfied simultaneously; the final set of weights represents compromises made to satisfy the constraints. Once trained, the NN can react to new stimuli (i.e., other than the training-set). An implementation problem to consider is that the matrix and vector calculations common to most NN's are often run on von Neumann uniprocessor machines with a "bottle-neck" forcing non-parallel computation. SMP (Symmetric Multi-Processing) architectures improve performance; however the best machines for these calculations are MPP (Massively Parallel Processing) or vector-register supercomputers; or embedded, application-specific, highly parallel systems – especially those providing learning in real-time. An all-digital vector-register NN processor (with on-chip learning) is presented below.

The <u>bottom-up</u> approach is to build a system which functions like a biological brain at the circuit-level. The artificial dendritic tree hybrid-analog/digital chip presented below is an example of this [6], [7].

Predictions of when computer performance will reach that of the human brain often employ Moore's Law to predict computing speed or number of transistors per chip:

$$Q_{NEW} = Q_{OLD}\left(2^{\left(n/1.5\right)}\right) \tag{1}$$

where Q_{old} is today's computing speed (or chip density), and Q_{new} is computing speed (or chip density) n years in the future (i.e., speed and chip density double every 18 months).

Although this law remains valid to-date, it must eventually break down; in less than 100 years, assuming a present day Q_{old} speed of 6Ghz and a chip density of 50 million transistor per chip, Moore's Law predicts a Q_{new} that would require electricity to travel through a transistor faster than the speed of light and more transistors on a chip than the number of atoms that could fit in the volume of a typical computer "case." This type of prediction can also be misleading if the degree of parallel processing (and pre-processing) that occurs in most biological brains is not considered. Multitasking manmade subsystems as efficiently and elegantly as the human brain is a major undertaking. The degree of parallelism (DOP) of the human brain is simply not found in PC's, workstations, or even mini-computers. Only in some supercomputers does parallelism come close to what might be required [1]. Embedded systems could eventually achieve these goals with many simple devices working independently; however embedded systems often lack the computation power (and precision) of even the simplest PC [5]. A comparison of computing platforms and their use for implementing machine intelligence is shown in Table 1.

Multitasking is a significant part of NN's where learning occurs between the many simple computational nodes. If an MPP machine could be built with billions of nodes (like the human brain), instead of just thousands (to-date), it could possibly implement an NN to rival the functionality of the human brain. Vector-register architectures are also well suited to the many parallel computations involved in the millions of "multiply-accumulates" often required for even the simplest of NN training.

Table 1. Levels of computing and machine intelligence use.

LEVEL	HARDWARE and DEVICES	OPERATING SYSTEM	MACHINE INTELLIGENCE USE
Em-bedded	**Microcontroller:** (Intel, Motorola, PIC) **Microprocessor:** (Intel/AMD, Motorola, PowerPC) **Application Specific IC's**	Typically none or custom	Not typical for symbolic AI. NN ASIC's excellent for high-speed real-time-learning NN applications.
PC	**Microprocessor** (Intel/AMD, PowerPC)	Windows, DOS, MAC OS, B, Linux	Acceptable for NN simulations and symbolic AI.
Work-station	Silicon Graphics , SUN, IBM RS6000 with multiple **Microproc's** (MIPS, SPARC, Intel/AMD, PowerPC)	Windows NT, UNIX, AIX	Good for NN simulations and symbolic AI.
Mini-Comp.	IBM AS400, Amdahl, HP, Hitachi (typically **SMP**)	UNIX, MVS, VMS, OS 390	Good for NN simulations and symbolic AI.
Super-Comp.	**SMP**: (IBM S/390) **MPP**: (IBM SP2, Cray) **Vector-register**: (Cray, IBM S/390 w/vector-register unit)	**SMP**: UNIX, MVS, OS 390 **MPP**: custom **Vector**:custom	Very good for symbolic AI, NN simulations, and real-time NN learning. **MPP** and **Vector-register** especially good for NN's.

DESIGN METHODOLOGY

The following steps can be used for any engineering design [2]:

(a) Define problem
(b) Simplify
(c) Find governing equations
(d) Build
(e) Test and rebuild as needed

Defining a problem includes creating or selecting the concepts to model, observe, and/or derive. An assessment is made of data needed and mathematical tools required. The "simplify" step involves making assumptions and considering different approaches. The selection of hardware platforms and programming languages can significantly effect the complexity, precision, and speed of both simulations and real-time systems. Finding governing equations involves identifying fundamental principles and may require deriving new equations. Different equation-solving techniques are considered; this may include selecting a solution for the fastest real-time response. For simulations, the selection and implementation of a solution may be more dependent on available programming constructs and functions, or on a choice of available numerical techniques. Care should be taken to ensure that the chosen approach does not cause discrepancies between simulations and real-time systems. The "build" step involves fabricating devices after simulating. Engineering of hardware and software may require real-time systems to interactively communicate with a concurrently running simulation [4]. Testing (and rebuilding as needed) involves verifying performance of hardware and software under various operating scenarios. This includes hand-checking computations and assessing resultant data for realistic results (e.g., *order-of-magnitude* checks). It can also involve gathering empirical data from observing real-time system performance, then modifying a simulation to create a more accurate model -- or possibly redesigning and rebuilding the real-time system. Assumptions made during the "simplify" step may need to be reconsidered.

NEUROCOMPUTER PROBLEM DEFINITION

Two methods for designing neurocomputers are presented below. Both can be classified as embedded systems. The first is a bottom-up design; an artificial dendritic tree where biological brain function is modeled as RC analog circuit elements that produce signals *similar* to those propagating through the dendritic tree inter-neuron connections of the human brain. This approach is modeled after the concept shown in Fig. 1. The second design is a top-down design that can process the vector and matrix operations of a typical NN mathematical model; and although it is designed as an embedded device, it has many of the design features of a vector-register supercomputer. The “behavioral” model shown in Fig. 2 inspired this approach.

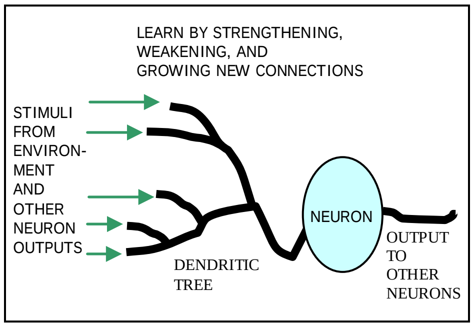

LEARN BY STRENGTHENING, WEAKENING, AND GROWING NEW CONNECTIONS
STIMULI FROM ENVIRON-MENT AND OTHER NEURON OUTPUTS
DENDRITIC TREE
NEURON
OUTPUT TO OTHER NEURONS

Figure 1. Biological neuron for bottom-up neurocomputer design.

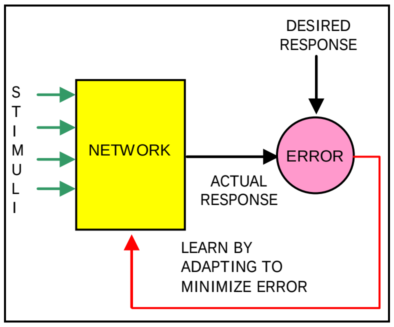
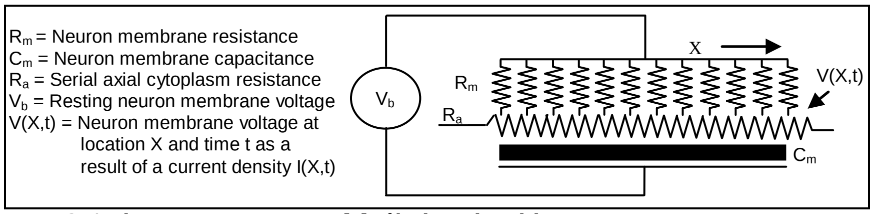


Figure 2. Behavioral model for top-down neurocomputer design.

SIMPLIFICATIONS AND ASSUMPTIONS

The bottom-up neurocomputer design is modeled and simplified by substituting analog circuit transient responses for the electrochemical signals and activations that occur in biological brain function. A design assumption is made that all neurons have fixed connections to all other neurons so that learning can take place by strengthening or weakening connections; the biological growing of new connections is mimicked by electrical connections that are simply inactive until a new connection is desired. The governing equation for the bottom-up neurocomputer presented here is based on the theory in [7] and the biology in Fig. 1. This is modeled as shown in Fig.3.


R_m = Neuron membrane resistance
C_m = Neuron membrane capacitance
R_a = Serial axial cytoplasm resistance
V_b = Resting neuron membrane voltage
V(X,t) = Neuron membrane voltage at location X and time t as a result of a current density I(X,t)
V_b
R_m
R_a
X
V(X,t)
C_m
Figure 3. Analog circuit representation [7] of biological model in Fig. 1.

and where neuron membrane voltage V(X,t) is found by solving:

$$R_m * C_m * \frac{\partial V(X,t)}{\partial t} = \frac{R_m}{R_a} * \frac{\partial V^2(X,t)}{\partial X^2} - V + R_m * I(X,t) \qquad (2)$$

The selection of a NN model for a top-down neurocomputer implementation is made here by analyzing historical advances in NN's while keeping in mind the relative success of models to be implemented in hardware or software. The following models were considered [8]-[11]:

(a) Backpropagation, (b) MADALINE III, (c) Hopfield, (d) BOLTZMANN MACHINE, (e) BAM (Bi-directional Associative Memory), and (f) NEOCOGNITRON

The relatively limited applications of the BAM and the NEOCOGNITRON eliminated these two from consideration. The BOLTZMANN MACHINE was eliminated next since the generalized delta rule of backpropagation is a faster learning algorithm for multilayered NN's [9]. Although there have been a number of successful applications of the Hopfield model, the exhaustive connectivity between neurons is less desirable for a single-chip implementation. MADALINE III and Backpropagation function in a similar fashion, however backpropagation exhibits faster learning [10]. Backpropagation is therefore the model chosen for implementation here. A backpropagation neurocomputer chip can be implimented in several ways:

1. Discrete analog components for all computations [9], [13], [14].
2. Digital circuits for all computations except transfer function implemented as serial or parallel analog circuits [9].
3. Digital circuits for all computations including transfer function implemented as serial or parallel look-up tables [15].
4. Parallel vector-register digital circuits for all computations including a polynomial approximation of the transfer function.

The first two approaches rely on analog circuits that can suffer from a number of limitations (e.g., drift, fabrication inconsistencies, conversion delays. etc.) [13], [14]; and although methods have been proposed to somewhat compensate for these problems [14], the approach chosen here is all-digital. The third approach, although entirely digital, would require large on-chip memory to yield the precision required for parallel on-chip learning; look-up table approaches often restrict transfer function computations to serial (one neuron at a time) execution. They may also require learning to be done off-chip with weights down-loaded onto the chip after learning completed. A technique to improve this is proposed in [15] where a "Symmetric Table Addition Method" uses two or more table lookups per transfer function evaluation. However the fourth approach (using a polynomial approximation of the transfer function) is likely to *scale* better when the architecture is expanded to thousands of neurons. This method is therefore chosen here; and on-chip learning is accomplished by defining a new transfer function, the "clipped-sigmoid," which is non-linear over an input domain wide enough to allow the generalized-delta, gradient-decent learning of backpropagation to work. Conversely, this domain is narrow enough to allow the transfer function to be approximated with a relatively high degree of precision; and since the approximation is a simple polynomial, it is easily implemented in digital hardware. For any top-down neurocomputer design, assumptions must be made for the magnitudes and precision needed for weights. Hardware can be simplified by designing the NN to have the minimum required computational precision [2]. It's important to recognize that *greater* precision is needed for evaluation of each neuron transfer function during training [15], [16].

GOVERNING EQUATIONS

The governing equation for the bottom-up neurocomputer design presented here is based on the theory in [7]. This implementation is shown in Fig 4. and is represented by equation (2) above. The field effect transistors (FET's) in Fig. 4 act to inhibit or stimulate by pulling the effected node *down* to the Inhibitory voltage (i.e., V_I=GND=0volts) or *up* to the Excitation voltage (i.e., $V_E > V_b$).

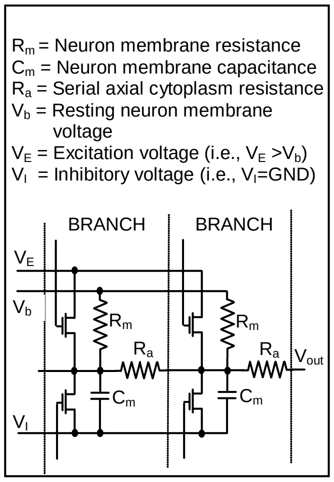
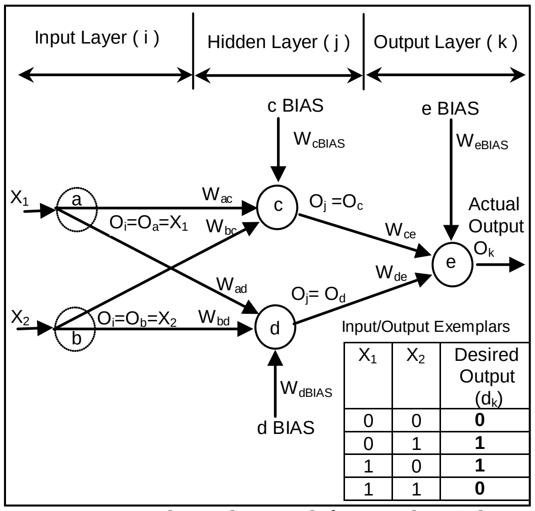

R_m = Neuron membrane resistance
C_m = Neuron membrane capacitance
R_a = Serial axial cytoplasm resistance
V_b = Resting neuron membrane voltage
V_E = Excitation voltage (i.e., $V_E > V_b$)
V_I = Inhibitory voltage (i.e., V_I=GND)
BRANCH
BRANCH
V_E
V_b
V_I
R_m
R_m
R_a
R_a
C_m
C_m
V_{out}

Figure 4. Analog circuit for "bottom-up" design [7].



X_1	X_2	Desired Output (d_k)
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

Figure 5. Layered neural network for "top-down" design.

The architecture for the top-down backpropagation neurocomputer design is shown in Fig. 5 including exemplars (i.e., desired outputs paired with corresponding inputs) for the simple XOR. Here, NN learning involves repeatedly feeding the network exemplars; each time changing weights as a function of a backpropagated error between the desired output and the actual output, until the approximate desired outputs are observed [10], [11]. This is performed as follows:

1. Choose small random initial values for weights (W's), and choose BIAS' (typically set to 1).
2. Feed the input layer an input vector (X_1, X_2) from an exemplar.
3. Propagate the signals forward via non-linear neuron transfer functions (i.e., "sigmoids"):

$$O_j = \frac{1}{1+e^{-(jBIAS*W_{jBIAS})+\sum_i(-O_i*W_{ij})}} \qquad (3)$$

$$O_k = \frac{1}{1+e^{-(kBIAS*W_{kBIAS})+\sum_j(-O_j*W_{jk})}} \qquad (4)$$

4. Create an error signal from the difference between actual and desired output for the exemplar, and use it to change the weights between the output layer (k) and the hidden layer (j), and also between the output neuron and bias:

$$\Delta W_{jk} = \eta * [(d_k - O_k) * O_k * (1 - O_k)] * O_j \qquad (5)$$

$$\Delta W_{kBIAS} = \eta * [(d_k - O_k) * O_k * (1 - O_k)] * kBIAS \qquad (6)$$

where η is the learning rate (typically set between 0.01 and 1).

5. Backpropagate a weighted error signal from the hidden layer (j) to the input layer (i) and use it to change the weights between the hidden layer (j) and the input layer (i), and also between the hidden layer neurons and bias':

$$\Delta W_{ij} = \eta * (O_j * (1 - O_j)) * \sum_k [(d_k - O_k) * O_k * (1 - O_k) * W_{jk}] * O_i \qquad (7)$$

$$\Delta W_{jBIAS} = \eta * (O_j * (1 - O_j)) * \sum_k [(d_k - O_k) * O_k * (1 - O_k) * W_{jk}] * jBIAS \qquad (8)$$

6. Repeat steps 2 to 5 for each exemplar.
7. Repeat steps 2 to 6 until desired outputs have been *approximately* obtained (i.e., within a specified tolerance).

The neuron transfer function presented here is a polynomial approximation that can be easily implemented using parallel vector-register digital circuits. In preliminary research, the polynomial chosen was a Taylor approximation expanded about a point $\boldsymbol{f}(x_0)=0$ [8]. The Taylor polynomial approximation of any function $\boldsymbol{f}(x)$ is given by:

$$P_{Taylor}(x) = f(x_0) + f'(x - x_0) + f''(x_0)\left[\frac{(x - x_0)^2}{2!}\right] + \ldots\ldots\ldots + f^n(x_0)\left[\frac{(x - x_0)^n}{n!}\right] \qquad (9)$$

where $\boldsymbol{P}(x_0) = \boldsymbol{f}(x_0)$, and where the error for all other x points is:

$$Error_{Taylor} = f(x) - P_{Taylor}(x) = (x - x_0)^{(n+1)}\left[\frac{f^{(n+1)}(\xi(x))}{(n+1)!}\right] \qquad (10)$$

for some number $\xi(x)$ between x and x_0 [17].

The error for a 15th degree Taylor polynomial approximation of the sigmoid neuron transfer function of equation (3) for (x_0=0) is shown in figures 6 and 7. Large approximation errors are encountered for (X < -2.5) and (X > 2.5). Through experiment, a better approximation was found to be a 10th degree Taylor polynomial approximation of the e^{-x} part of the sigmoid; this results in a good approximation of the sigmoid for (-3.5 > X < 3.5) as shown in Fig. 7. This approximation is termed the "Clipped Sigmoid" and yields sufficient accuracy to allow learning to occur; however certain network initializations need to be specified.

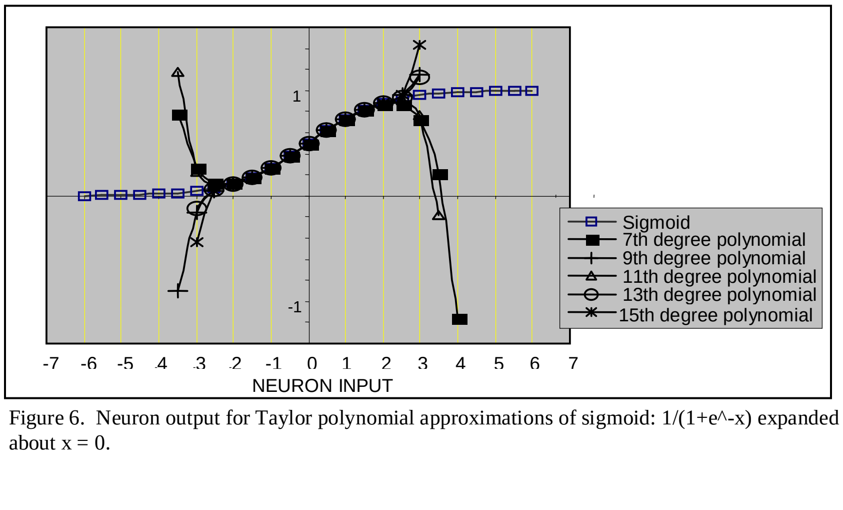


Figure 6. Neuron output for Taylor polynomial approximations of sigmoid: 1/(1+e^-x) expanded about x = 0.

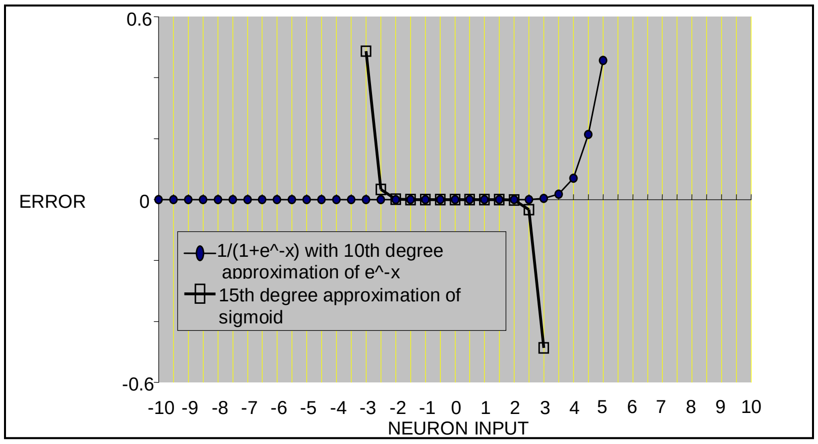


Figure 7. Error for Taylor series polynomial approximations of sigmoid: 1/(1+e^-x).

The initial research of [8] is extended here to include more precise polynomial approximations of the sigmoid (and an initial hardware implementation using FPGA's). The approximation techniques considered are: **(a) Cubic Spline Polynomial, (b) Hermite Polynomial, and (c) Divided-Difference Polynomial**

Cubic Spline is eliminated from further consideration since it is "piecewise" requiring different polynomials for different parts of the input domain. This would require each neuron computation to include a check for input values before applying one of several polynomial approximations. The remaining approaches are better suited for a fully parallel implementation; and will scale better to thousands of neurons.

A (2*n* + 1) degree Hermite polynomial approximation of a function ***f***(x) with points evaluated at ($x_0, x_1, \ldots, x_n$) is given in [17] by:

$$P_{Hermite}(x) = \sum_{j=0}^{n} f(x)H_{n,j}(x) + \sum_{j=o}^{n} f'(x)\hat{H}_{n,j}(x) \qquad (11)$$

where $$H_{n,j}(x) = \left[1 - 2\left(x - x_j\right)L'_{n,j}(x_j)\right]L^2_{n,j}(x) \qquad (12)$$

$$\hat{H}_{n,j}(x) = \left(x - x_j\right)L^2_{n,j}(x) \qquad (13)$$

$$L_{n,j}(x) = \prod_{\substack{i=0 \\ i \neq j}}^{n} \frac{\left(x - x_i\right)}{\left(x_j - x_i\right)} \qquad (14)$$

with error:

$$Error_{Hermite} = f(x) - P_{Hermite}(x) = \left[\frac{\prod_{i=0}^{n}\left(x - x_i\right)^2}{(2n+2)!}\right] f^{(2n+2)}(\xi) \qquad (15)$$

for some number between adjacent points in ($x_0, x_1, \ldots, x_n$). This Polynomial is shown approximating ***f***(x)=("sigmoid" of equation (3)) in figures 8 and 9. Here, a 12th degree Hermite polynomial yields a relatively good approximation for (–3.5 > X < 3.5).

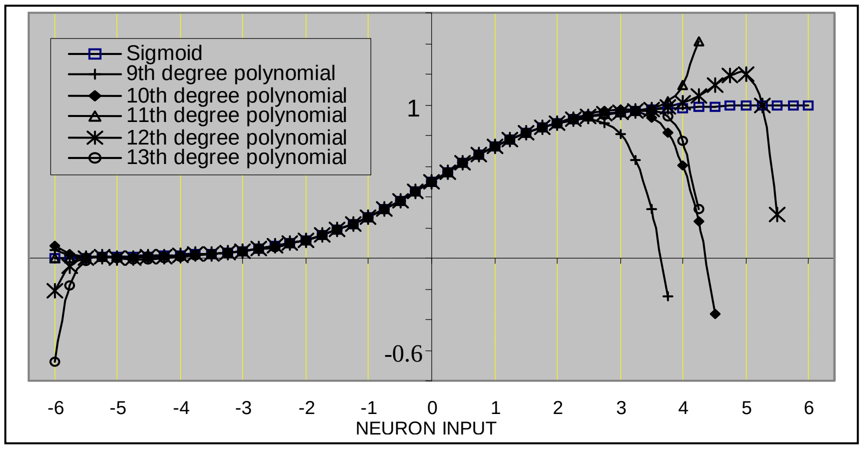


Figure 8. Neuron output for Hermite polynomial approximations of sigmoid: 1/(1+e^-x) with evaluation points at x = [-6 to +6] at 0.25 intervals.

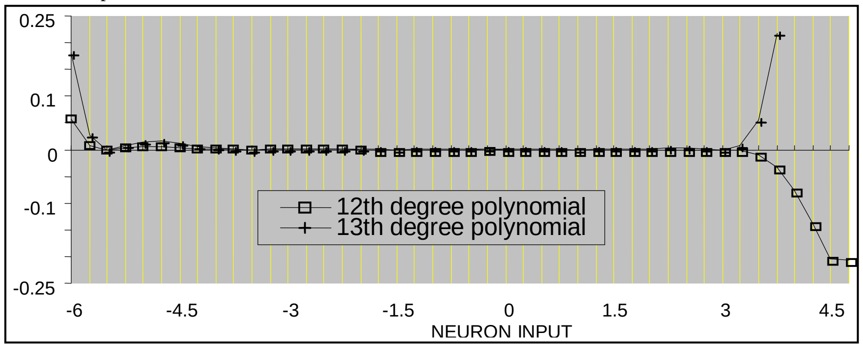


Figure 9. Error for 12th and 13th degree Hermite polynomial approximation of sigmoid: 1/(1+e^-x).

An nth degree Divided Difference polynomial approximation of any function ***f***(x) with points evaluated at ($x_0, x_1, \ldots, x_n$) is given in [17] by:

$$P_{DivDiff}(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \ldots + a_n(x - x_0)(x - x_1)\cdots(x - x_{n-1}) \tag{16}$$

where

$$a_0 = f[x_0] = f(x_0) \tag{17}$$

$$a_1 = f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{(x_1 - x_0)} \tag{18}$$

$$a_2 = f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{(x_2 - x_0)} = \frac{1}{x_2 - x_0}\left[\frac{f(x_2) - f(x_0)}{x_2 - x_1} - \frac{f(x_2)}{x_1 - x_0}\right] \tag{19}$$

$$= \frac{f(x_0)}{(x_0 - x_1)(x_0 - x_2)} + \frac{f(x_1)}{(x_1 - x_0)(x_1 - x_2)} + \frac{f(x_2)}{(x_2 - x_0)(x_2 - x_1)}$$

$$a_n = f[x_0, x_1, \ldots x_n] = \frac{f(x_0)}{(x_0 - x_1)\cdots(x_0 - x_n)} + \frac{f(x_1)}{(x_1 - x_0)\cdots(x_1 - x_n)} + \ldots + \frac{f(x_n)}{(x_n - x_0)(x_n - x_{n-1})} \quad (20)$$

Divided-Difference polynomial approximations of the sigmoid are shown in figures 10 and 11, and provide the *best* approximations here (i.e., best precision over the widest domain of input values). Two 12th degree Divided Difference polynomial approximations are shown in Fig. 11; one with evaluation points over an X domain from –6 to 6; the other from –10 to 10. The second one yields an approximating error of (0.01) over a relatively wide domain and is the polynomial chosen for implinentation.

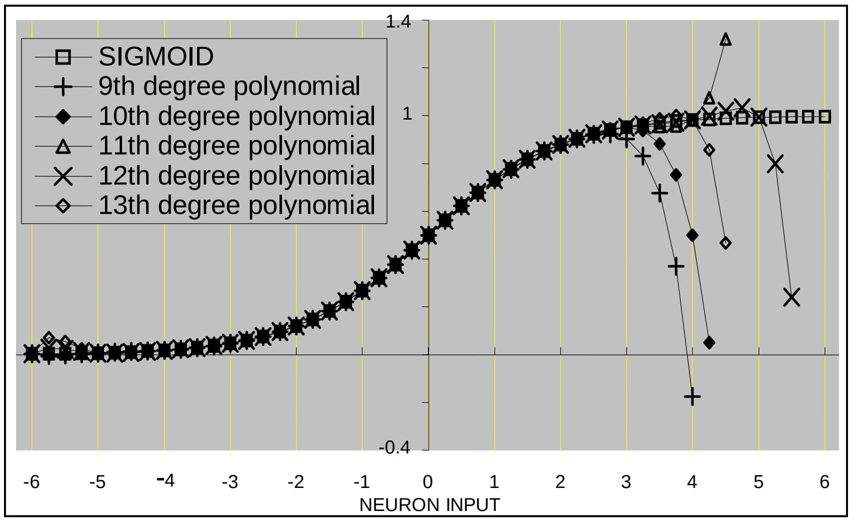


Figure 10. Neuron output with Divided-Difference polynomial approximations of sigmoid: 1/(1+e^-x) with evaluation points at x = [- 6, - 5, - 4, - 3, - 2, - 1, -0. 5, 0.5, 1, 2, 3, 4, 5, 6]

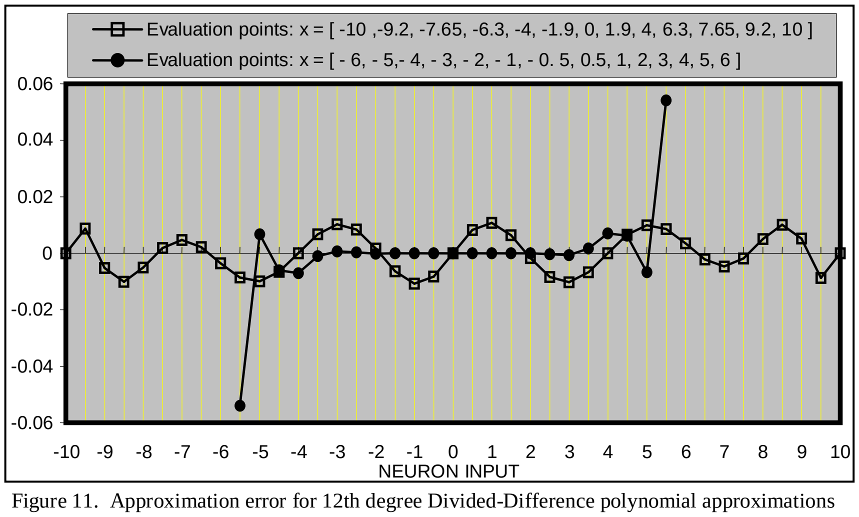


Figure 11. Approximation error for 12th degree Divided-Difference polynomial approximations of sigmoid: 1/(1+e^-x)

To validate this choice, all two-input logic gates were simulated. Tests included evaluating the effect of clipping the standard sigmoid (i.e., no polynomial approximation) which even improved learning for some cases. In Fig. 12, the Divided-Diffence polynomial sigmoid approximation is shown to allow successful learning. This XOR simulation is clipped at -5.25 > X < 5.25 to show the robostness of the method (i.e., it also works with a domain of –10 > X < 10).

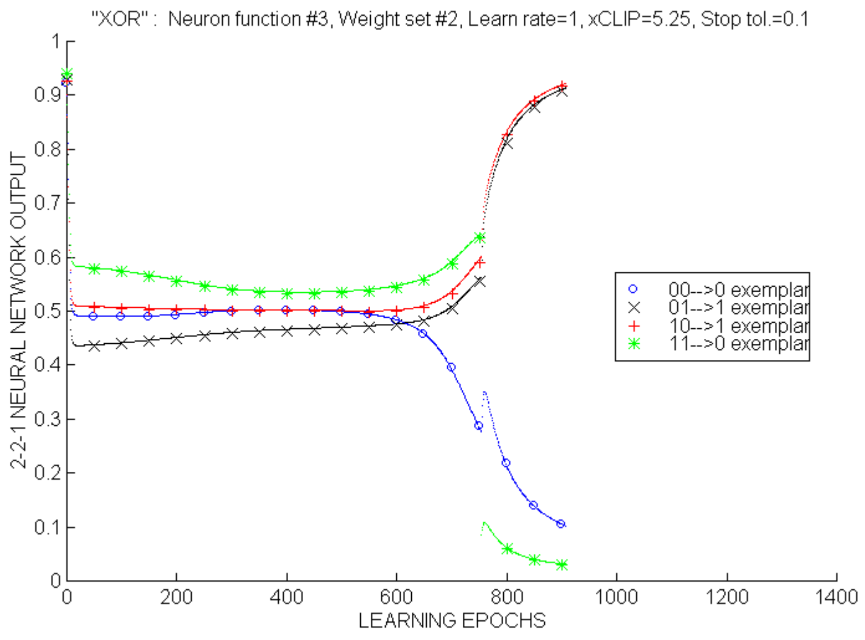


Figure 12. Backpropagation learning using 12th degree Divided-Difference polynomial approximation of sigmoid, and "clipping" outside of domain –5.25 >X <5.25.

BUILDING AND TESTING

The bottom-up design in Fig. 13 is an artificial dendritic tree VLSI chip. It has 64 neurons and combines the analog circuits of Fig. 4 with digital circuits to latch in 4-bit values allowing each node to be excited or inhibited for 16 different "pulse-lengths." The chip has the equivalent of 10,000 transistors on a 2mm x 2mm die [6].

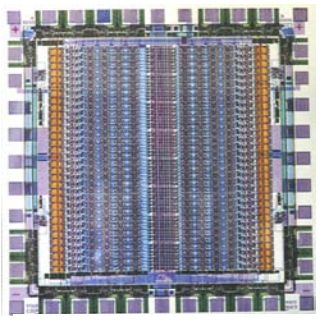


Figure 13. Bottom-up neurocomputer (artificial dendritic tree VLSI chip).

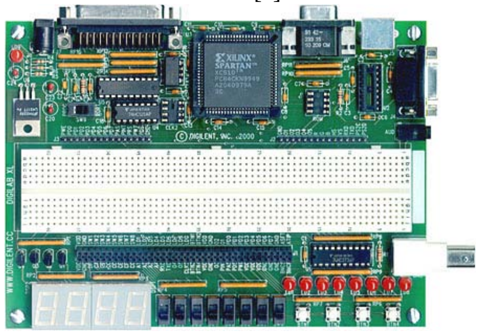


Figure 14. Field programmable gate array (FPGA) for top-down neurocomputer development.

This chip was fabricated and bench-tested by latching in various pulse-length values, then measuring outputs for desired transient output voltage responses.

The top-down neurocomputer design is presently under development using the Field Programmable Gate Array (FPGA) shown in Fig. 14. Testing any digital circuit implementation (FPGA, discrete IC chips, VLSI, etc.), or even an analog circuit implementation of a backpropagation model, involves feeding the network each exemplar after training is completed to verify desired outputs are obtained. This can be followed by testing *never-seen* inputs. Testing complex NN hardware can also involve developing verification programs for both simulated and physical prototype architectures [18].

CONCLUSIONS

The top-down design presented here can process the mathematics of the well-known backpropagation NN model; and although designed as an embedded device, it has an architecture similar to a vector-register supercomputer. This design is entirely digital, fully parallel, and implements a polynomial approximation of the sigmoid transfer function to allow parallel on-chip learning.

Even though the semiconductor industry continues to increase the number of transistors per unit area, the chip-area required to include the neurons needed for a bottom-up neurocomputer to produce useful *higher-reasoning* would need to be much larger than a typical chip. Biological brains have the advantage of being three-dimensional whereas integrated circuits are mostly two-dimensional (despite multiple levels of *layerization*). Another problem is connecting all of these neurons since wire routing would be in mostly two dimensions. Even with several layers of *metallization* (for wires), it would be very difficult to connect all neurons (with each potentially connected to all others). Perhaps the most difficult problem to overcome in mimicking biological learning is that inter-neuron connections are not only strengthened or weakened, but are grown. Wires on chips are fixed, and considering the required extensive connectivity between neurons, useful bottom-up designs can be difficult to realize.

Future research will investigate merging bottom-up techniques for pre-processing sensory data (e.g., visual, auditory, olfactory) with top-down techniques for higher reasoning; including combining neural networks with symbolic AI programming.

References

[1] J. T. Wunderlich, " Defining the limits of machine intelligence," in *Proc. of IEEE SECon 2003 Nat'l Conf.*, Ocho Rios, Jamaica, 2003.

[2] J. T. Wunderlich, "Simulation vs. real-time control; with applications to robotics and neural networks," in *Proc. of ASEE Nat'l Conf.*, Albuquerque, NM, 2001.

[3] Seiffert, U., "Artificial neural networks on massively parallel computer hardware," in *Proc. ESANN 2002 European Symposium on Artificial Neural Networks*, Bruges, Belgium, 2002.

[4] Campos, D. and Wunderlich, J. T., "Development of an interactive simulation with real-time robots for search and rescue," in *Proc. of IEEE/ASME Int'l Conf. on Flexible Manufacturing*, Hiroshima, Japan, 2002.

[5] Wunderlich, J. T., "Focusing on the blurry distinction between microprocessors and microcontrollers," in *Proc. of ASEE Nat'l Conf.*, 1999, Charlotte, NC.

[6] J. T. Wunderlich, et al., "Design of an artificial dendritic tree VLSI microprocessor," University of Delaware research report, 1993.

[7] Elias, J. G., "Artificial dendritic trees," *Neural Computation, vol. 5*, pp. 648-663, 1993.

[8] Wunderlich, J. T., "Design of a Neurocomputer Vector Microprocessor (with on-chip learning)," Masters thesis, Pennsylvania State University, Jan. 1992.

[9] Soucek, B., "*Neural and Concurrent Real-Time Systems, The Sixth Generation*." New York: John Wiley & Sons, 1989.

[10] Widrow, B., "30 years of adaptive neural networks: perceptron, madaline, and backpropagation," in *Proc. 2nd IEEE Intl. Conf. on Neural Networks*, 1990.

[11] Rumelhart, D.E., and McClelland, J. L., "*Parallel Distributed Processing*." Cambridge, MA: M.I.T. Press, 1986.
[12] Mirhassani, M., Ahmadi, M., Miller, M. C., "A mixed-signal VLSI neural network with on-chip learning," in *Proc. of the 2003 IEEE Canadian Conference on Electrical and Computer Engineering*, 2003.
[13] Liu, J., and Brooke, M., "Fully parallel on-chip learning hardware neural network for real-time control," in *Proc. IEEE International Symposium on in Circuits and Systems*, 1999.
[14] Card, H. C., McNeill, D. K., and Schneider, R. S., "How forgiving is on-chip learning of circuit variations?," in *Proc. of the 5th Irish Neural Networks Conference*, Maynooth, Ireland, 1995.
[15] Nihal, K. S., Schlessman, J. A., and Schulte, M. J., "Symmetric table methods for neural network approximations," in *Proc. SPIE: Advanced Signal Processing Algorithms, Architectures, and Implementations XI*, San Diego, CA., 2001.
[16] Beiu, V., "How to build VLSI-efficient neural chips," *in Proc. ICSC Symp. On Engineering Intelligent Systems, Tenerife*, Spain, 1998.
[17] Burden, R. L., and Faires, J. D., *"Numerical Analysis,"* Brooks Cole publishing, 7 ed., 2000.
[18] Wunderlich, J. T., "Random number generator macros for the system assurance kernel product assurance macro interface," *Systems Programmers User Manual for IBM S/390 Systems Architect ure Verification*, IBM S/390 Hardware Development Lab, Poughkeepsie, NY, 1997.

JOSEPH T. WUNDERLICH
Dr. Wunderlich is the primary Computer Engineering Program Coordinator for Elizabethtown College. Previously, he worked for Purdue University as an Assistant Professor and for IBM as a researcher and hardware development engineer. He received his Ph.D. in Electrical and Computer Engineering from the University of Delaware, his Masters in Engineering Science/Computer Design from the Pennsylvania State University and his BS in Engineering from the University of Texas at Austin.