Hardware for Machine Learning
Two Single-Chip Neurocomputer Designs; One Bottom-Up, One Top-Down
DRAFT BOOK CHAPTER, *Joseph T Wunderlich PhD*
Last update: December 2017

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 chapter first discusses the use of various computer platforms for implementing neural networks, then focuses on two single-chip neurocomputer implementations: (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 facilitated by modifying the neuron transfer function using a polynomial approximation with clipping of neuron inputs outside of specified values. The validity of this methodology is supported by an analysis of the mathematics of gradient decent learning.

1. 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. 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) (Wunderlich 2004).

Most NN's are top-down 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 bottom-up 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 (Elias 1993, Wunderlich, et al. 1993)

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:

$$Q_{NEW} = Q_{OLD}\left(\left|2^{(Y/15)}\right|\right) \qquad (1)$$

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 (Wunderlich 2003). 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 (Wunderlich 1999). 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 I. Levels of Computing and Machine Intelligence Use

LEVEL	HARDWARE and DEVICES	OPERATING SYSTEMS	MACHINE INTELLIGENCE USE
Embedded	**Microcontroller:** (Intel, Motorola, PIC's) **Microprocessor:** (Intel/AMD, Motorola, PowerPC) **ASIC:** (Application Specific IC's)	None or custom	Not typically used for symbolic AI programs. Neural network ASIC's can be excellent for high-speed real-time-learning neural network applications.
PC	**Microprocessor** (Intel/AMD, PowerPC)	Windows, DOS, MAC OS, B, Linux	Acceptable for neural network simulations and symbolic AI programs.
Workstation	Silicon Graphics , SUN, IBM RS6000 with multiple **Microprocessors** (MIPS, SPARC, Intel/AMD, PowerPC)	Windows NT, UNIX, AIX	Good for neural network simulations and symbolic AI programs.
Mini-Computer	IBM AS400, Amdahl, HP, Hitachi (typically **SMP**)	UNIX, MVS, VMS, OS 390	Good for neural network simulations and symbolic AI programs.
Super-Computer	**SMP**: (e.g., IBM S/390) **MPP**: (e.g., IBM SP2, Cray) **Vector-register**: (e.g., Cray, IBM S/390 with vector-register unit)	**SMP**: UNIX, MVS, OS 390 **MPP**: custom distributed OS **Vector-register**: typically custom OS (e.g., for Cray)	Very good for neural network simulations and symbolic AI programs. **MPP** and **Vector-register** especially good for neural networks.

2. Design Methodology

The following steps can be used for any engineering design (Wunderlich 2001):

(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 (Campos & Wunderlich 2002). *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.

3. Neurocomputer problem definition

Two methods for designing neurocomputers are presented below. Both can be classified as embedded systems. The first is a bott om-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.

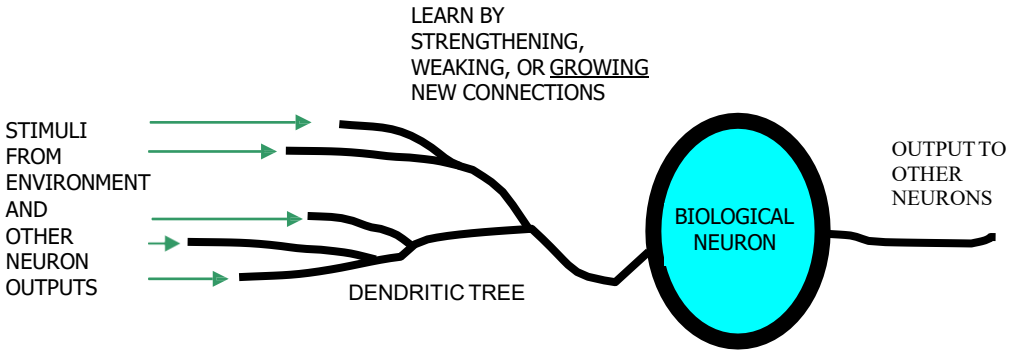


Figure 1. Biological neuron for bottom-up neurocomputer design.

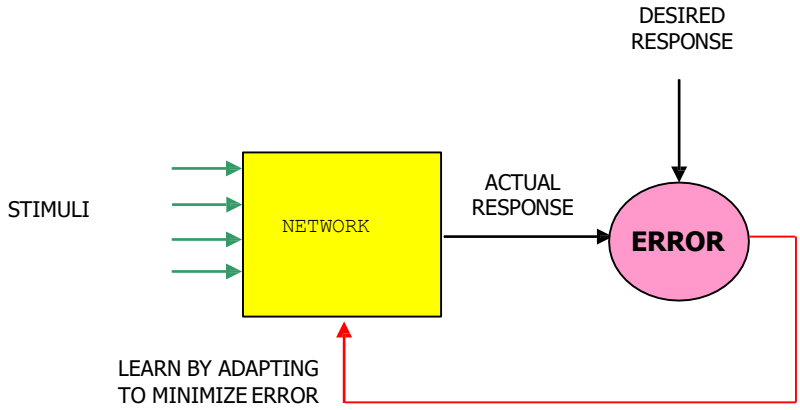


Figure 2. Behavioral model for top-down neurocomputer design.

4. 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 (Elias 1993) and the biology in Fig. 1. This is modeled as shown in Fig.3.

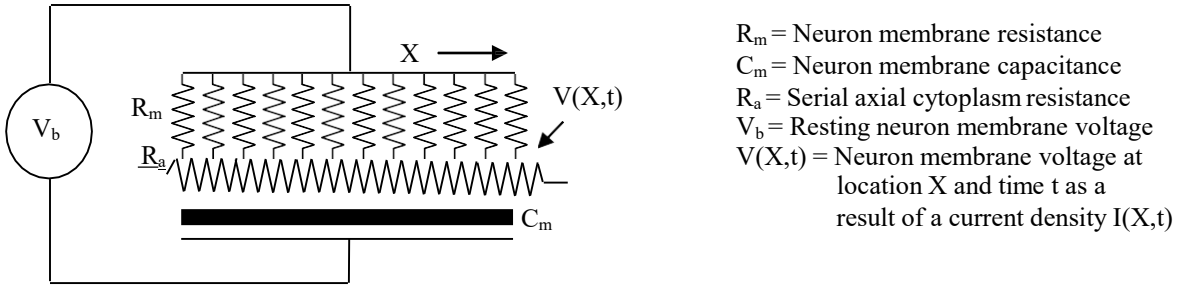


Figure 3. Analog circuit representation (Elias 1993) of biological model in Fig. 1.

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

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

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:

1. Back-propagation (Rumelhart & McClelland 1986, Rumelhart et al. 1986)
2. MADALINE III (Widrow et al. 1987, Andes 1988, Widrow and Lehr 1990)
3. Hopfield (Hopfield 1982, Hopfield 1984)
4. BOLTZMANN MACHINE (Ackley et al. 1985)
5. BAM [Bi-directional Associative Memory] (Kosko 1987)
6. NEOCOGNITRON (Fukushima 1983) 7.

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 (Widrow and Lehr 1990). 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 (Widrow and Lehr 1990). Backpropagation is therefore the model chosen for implementation here.

In the past there have been a number of successful integrated circuit implementations of neural networks which consolidated the many neural network multiply/accumulate operations onto a chip or a board, but performed the computationally expensive neuron transfer function execution by a host computer (i.e., off-chip or even off-board); this computation has also be done serially, one neuron at a time, through look-up tables or numerical methods. Examples are Hitachi neuro- chips (Masaki 1990), and Morton Intelligent Memory Chips (Morton 1988).

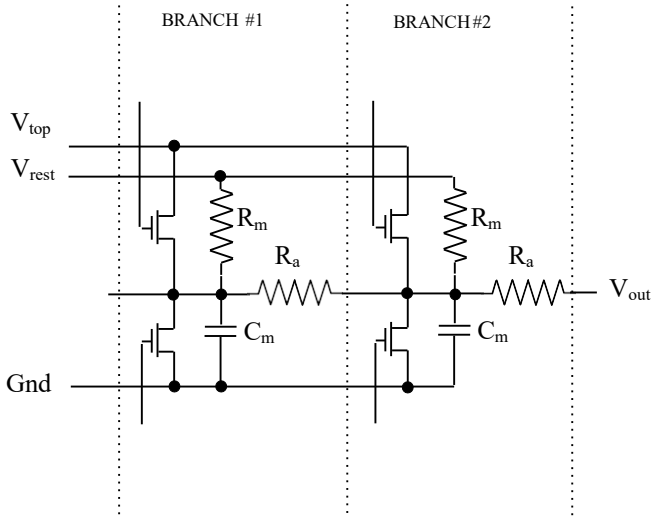
Until recently, creating a fully parallel neual network chip with parallel on-chip learning and a *large* neuron count was not fesible (i.e., too high of a transister count (wunderlich 1992)). A single-chip neurocomputer has the potential of much faster execution by eliminating the latency of off-chip tranfer function execution. Asingle-chip neurocomputer can be implimented in several ways:

1. Discrete analog components for all computations (Soucek 1989, Card 1995, Lin et al. 1999).
2. Digital circuits for all computations except transfer function implemented as serial or parallel analog circuits (Soucek 1989).
3. Digital circuits for all computations including transfer function implemented as serial or parallel look-up tables (Nihal et al. 2001).
4. Parallel vector-register digital circuits for all computations including a polynomial approximation of the transfer function.

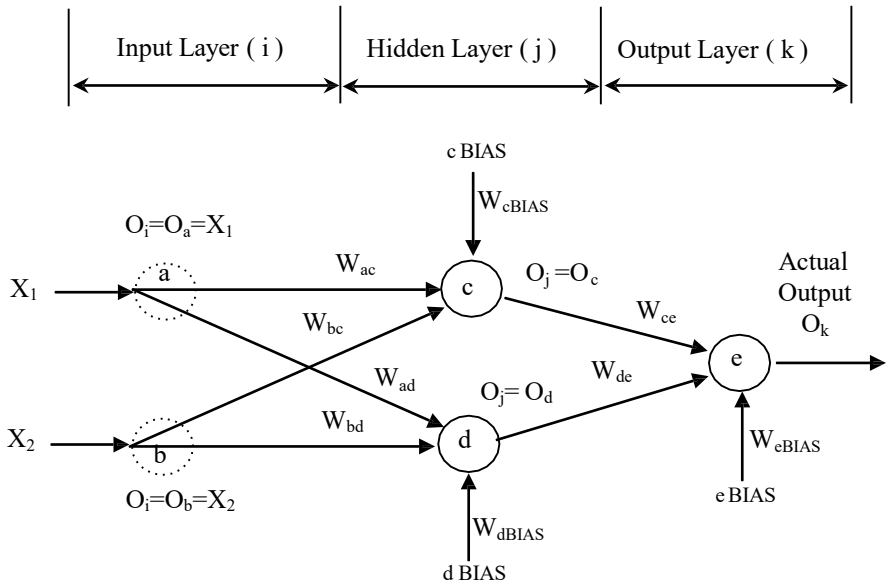
The first two approaches rely on analog circuits that can suffer from a number of limitations (e.g., drift, fabrication inconsistencies, conversion delays. etc.), (Card 1995, Lin et al. 1999); and although methods have been proposed to somewhat compensate for these problems (Card 1995), the approach 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 (Nihal et al. 2001) 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 (Wunderlich 2001). It's important to recognize that *greater* precision is needed for evaluation of each neuron transfer function during training (Beiu et al. 1998, Nihal et al. 2001).

5. Governing equations

The governing equation for the bottom-up neurocomputer design presented here is based on the theory in (Elias 1993). 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., GND=0volts) or *up* to the Excitation voltage (i.e., $V_{top} > V_{rest}$).\



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.



Input / Output
"**Exemplars**"

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

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

5. Simulations

Here, 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 (Rumelhart & McClelland 1986, Rumelhart et al. 1986, Widrow and Lehr 1990). This is performed as follows:

1. Choose small random initial values for weights (W's), and choose BIAS' -- typically set to 1; however the bias can be treated as a variable to shift the neuron output values of the entire network or to selectively inhibit or excite certain neurons (e.g., perhaps to be influenced by another concurrently running network). This kind of mechanism was observed in human brain circuits by (Ross 1991) where the effect is to lower or raise the level at which the neurons fire:

   > *"This kind of circuit might prove essential to learning, cognition, and creativity, as it could help focus attention to certain incoming information, correlate neural activity over variable distances, and result in disregard of other simultaneous input that might otherwise be distracting. In the human brain, such circuits may provide the emotional context within which higher cognitive functioning occurs." (Ross 1991)*

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"):

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

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

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:

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

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

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

6. 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':

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

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

7. Repeat steps 2 to 5 for each exemplar.

8. Repeat steps 2 to 6 until desired outputs have been *approximately* obtained (i.e., within a specified tolerance).

Here is some Matlab code to implement the above methodogy:

```
%*************************************************************************************
%            A 2-2-1 or 3-3-1 back-propagation Neural Network
%                          by Joseph Wunderlich,Ph.D.
%*************************************************************************************
%******************** START TIMER AND INSTRUCTION COUNTER **********
 startTIME=cputime;
 %******************** PICK AN ARCHITECTURE of 2-2-1 or 3-3-1 ***************
ARCHITECTURE=1;                    %"1" means 2-2-1, "2" means 3-3-1 Network Architecture
%******************** 2-2-1 and 3-3-1 INPUT ****************************************
PLOTTING=1;                        %Turn plotting on "1" or off "0" for speed
RATE=1;                            %Learning Rate
EPOCHcountMAX=4000;                 %Stop if goal not reached after this many iterations
STOPtolerance=.1;                  %How close to get to asymptotes at 0 or 1
                                   %Training sets of exemplars for each architecture:
EXEMPLAR_221=[0 0 0;               %input1, input2, and desired output for exemplar #1
          0 1 1;                   %input1, input2, and desired output for exemplar #2
          1 0 1;                   %input1, input2, and desire doutput for exemplar #3
          1 1 0];                  %input1, input2, and desired output for exemplar #4
EXEMPLAR_331=[0 0 0 0;             %input1,2,3 and desired output for exemplar #1
          0 0 1 1;                 %input1,2,3 and desired output for exemplar #2
          0 1 0 1;                 %input1,2,3 and desired output for exemplar #3
          0 1 1 1;                 %input1,2,3 and desired output for exemplar #4
          1 0 0 1;                 %input1,2,3 and desired output for exemplar #5
          1 0 1 1;                 %input1,2,3 and desired output for exemplar #6
          1 1 0 1;                 %input1,2,3 and desired output for exemplar #7
          1 1 1 0];                %input1,2,3 and desired output for exemplar #8
if ARCHITECTURE==1 % START IMPLEMENTING 2-2-1 ARCHITECTURE

%********************************* 2-2-1 INITIALIZATION *********************************
 Wac=.5; Wad=.6;
 Wbc=.7; Wbd=.8;
 Wce=.9; Wde=1;
 WcBIAS=1;WdBIAS=1; WeBIAS=1;
 cBIAS=1; dBIAS=1;  eBIAS=1;
 Exemplar1_OutputLAST=[.5 .5 .5]; %just to get it started
 Exemplar2_OutputLAST=[.5 .5 .5];
 Exemplar3_OutputLAST=[.5 .5 .5];
 Exemplar4_OutputLAST=[.5 .5 .5];
 EPOCHcount=0;
 n=1;
%************************************ 2-2-1 MAIN LOOP ********************************
%*************************************************************************************
while ((EPOCHcount) < EPOCHcountMAX)& ...
      ((abs(Exemplar1_OutputLAST(3)-EXEMPLAR_221(1,3))> STOPtolerance)| ...
       (abs(Exemplar2_OutputLAST(3)-EXEMPLAR_221(2,3))>  STOPtolerance)|  ...
       (abs(Exemplar3_OutputLAST(3)-EXEMPLAR_221(3,3))>  STOPtolerance)|  ...
       (abs(Exemplar4_OutputLAST(3)-EXEMPLAR_221(4,3))> STOPtolerance))

  EPOCHcount=EPOCHcount+1;
  for i=1:4
   Oc=1/(1+exp((-(cBIAS*WcBIAS)- EXEMPLAR_221(i,1)*Wac - EXEMPLAR_221(i,2)*Wbc )));
   Od=1/(1+exp((-(dBIAS*WdBIAS)- EXEMPLAR_221(i,1)*Wad - EXEMPLAR_221(i,2)*Wbd )));
   Oe=1/(1+exp((-(eBIAS*WeBIAS)- Oc*Wce - Od*Wde )));
   if i==1
     Exemplar1_OutputLAST=[EXEMPLAR_221(i,1) EXEMPLAR_221(i,2) Oe];
     if PLOTTING==1
       figure(1);
       plot(EPOCHcount,Oe,'bo');
       hold on;
     end;
    elseif i==2
      Exemplar2_OutputLAST=[EXEMPLAR_221(i,1) EXEMPLAR_221(i,2) Oe];
      if PLOTTING==1
        figure(1);
        plot(EPOCHcount,Oe,'r.');
        hold on;
      end;
       elseif i==3
         Exemplar3_OutputLAST=[EXEMPLAR_221(i,1) EXEMPLAR_221(i,2) Oe];
         if PLOTTING==1
           figure(1);
           plot(EPOCHcount,Oe,'y.');
```


```
          hold on;
        end;
          else
            Exemplar4_OutputLAST=[EXEMPLAR_221(i,1) EXEMPLAR_221(i,2) Oe];
            if PLOTTING==1
              figure(1);
              plot(EPOCHcount,Oe,'go');
              hold on;
            end;
          end;

          Exemplars_OutputLAST=[EPOCHcount/10000 Exemplar1_OutputLAST; ...
                        EPOCHcount/10000  Exemplar2_OutputLAST;  ...
                        EPOCHcount/10000  Exemplar3_OutputLAST;  ...
                        EPOCHcount/10000 Exemplar4_OutputLAST]
  error=EXEMPLAR_221(i,3)-Oe;
  errorprop=error*Oe*(1-Oe);

  dWeBIAS=RATE*errorprop*eBIAS;
  dWce= RATE*errorprop*Oc;
  dWde=  RATE*errorprop*Od;

  dWcBIAS=RATE*Oc*(1-Oc)*(errorprop*Wce)*cBIAS;
  dWac=     RATE*Oc*(1-Oc)*(errorprop*Wce)*EXEMPLAR_221(i,1);
  dWbc=     RATE*Oc*(1-Oc)*(errorprop*Wce)*EXEMPLAR_221(i,2);
  dWdBIAS=RATE*Od*(1-Od)*(errorprop*Wde)*dBIAS;
  dWad= RATE*Od*(1-Od)*(errorprop*Wde)*EXEMPLAR_221(i,1);
  dWbd= RATE*Od*(1-Od)*(errorprop*Wde)*EXEMPLAR_221(i,2);

              Wac=Wac+dWac;
              Wad=Wad+dWad;
              Wbc=Wbc+dWbc;
              Wbd=Wbd+dWbd;
              Wce=Wce+dWce;
              Wde=Wde+dWde;
              WcBIAS=WcBIAS+dWcBIAS;
              WdBIAS=WdBIAS+dWdBIAS;
              WeBIAS=WeBIAS+dWeBIAS;
              Wdisplay=[Wac Wad Wbc Wbd Wce Wde WcBIAS WdBIAS WeBIAS];
    n=n+1;
  end;
end;

EPOCHcount
endTIME=cputime-startTIME
if PLOTTING==1
  figure(1);                          %open figure window #1
  % axis([-120 335 -50 300]);         %define x and y axis for figure window #1
  title(['LEARNING RATE =',num2str(RATE), ' Stopping tolerance = ',num2str(STOPtolerance),' ' ...
  num2str(endTIME), ' secs of CPU time ']);
  xlabel('LEARNING EPOCHS');
  ylabel('2-2-1 NEURAL NETWORK OUTPUT');
  h = legend('00 input','01 input','10 input','11 input',4);
  hold on;
end;
%****************************************** END 2-2-1  MAIN  LOOP***********************************
%****************************************** END 2-2-1 ARCHITECTURE  ***************************
%*************************************************************************************************

%****************************************** BEGIN 3-3-1 ARCHITECTURE ***************
elseif ARCHITECTURE==2 % START IMPLEMENTING 3-3-1 ARCHITECTURE
%****************************************** 3-3-1 INITIALIZATION **********************
  %Weight Values
  %A,B,C are input layer neurons
  %D,E,F are hidden layer neurons
  %G is output layer neuron
Wad= .4; Wae= .45; Waf= .5;
Wbd= .55; Wbe= .6;  Wbf= .65;
Wcd= .7; Wce= .75; Wcf= .8;
Wdg= .85; Weg= .9;  Wfg= .63;
%Bias values (MAY BE CHANGED BASED ON a concurrent SITUATION)
dBIAS= 1;  WdBIAS=1;
eBIAS= 1;   WeBIAS=1;
fBIAS= 1;  WfBIAS=1;
gBIAS= 1;  WgBIAS=1;
```


```
Exemplar1_OutputLAST=[.5 .5 .5 .5]; % just to get it started
Exemplar2_OutputLAST=[.5 .5 .5 .5];
Exemplar3_OutputLAST=[.5 .5 .5 .5];
Exemplar4_OutputLAST=[.5 .5 .5 .5];
Exemplar5_OutputLAST=[.5 .5 .5 .5];
Exemplar6_OutputLAST=[.5 .5 .5 .5];
Exemplar7_OutputLAST=[.5 .5 .5 .5];
Exemplar8_OutputLAST=[.5 .5 .5 .5];
EPOCHcount=0;
n=1;
%************************************ 3-3-1 MAIN LOOP  ********************************
%**************************************************************************************************
while ((EPOCHcount) < EPOCHcountMAX)& ...
      ((abs(Exemplar1_OutputLAST(4)-EXEMPLAR_331(1,4))> STOPtolerance)| ...
       (abs(Exemplar2_OutputLAST(4)-EXEMPLAR_331(2,4))> STOPtolerance)| ...
       (abs(Exemplar3_OutputLAST(4)-EXEMPLAR_331(3,4))> STOPtolerance)| ...
       (abs(Exemplar4_OutputLAST(4)-EXEMPLAR_331(4,4))> STOPtolerance)| ...
       (abs(Exemplar5_OutputLAST(4)-EXEMPLAR_331(5,4))> STOPtolerance)| ...
       (abs(Exemplar6_OutputLAST(4)-EXEMPLAR_331(6,4))> STOPtolerance)| ...
       (abs(Exemplar7_OutputLAST(4)-EXEMPLAR_331(7,4))> STOPtolerance)| ...
       (abs(Exemplar8_OutputLAST(4)-EXEMPLAR_331(8,4))> STOPtolerance))
 EPOCHcount=EPOCHcount+1;
 for i=1:8
Od=1/(1+exp((-(dBIAS*WdBIAS)- EXEMPLAR_331(i,1)*Wad - EXEMPLAR_331(i,2)*Wbd - EXEMPLAR_331(i,3)*Wcd )));
Oe=1/(1+exp((-(eBIAS*WeBIAS)- EXEMPLAR_331(i,1)*Wae - EXEMPLAR_331(i,2)*Wbe - EXEMPLAR_331(i,3)*Wce )));
Of=1/(1+exp((-(fBIAS*WfBIAS)- EXEMPLAR_331(i,1)*Waf - EXEMPLAR_331(i,2)*Wbf - EXEMPLAR_331(i,3)*Wcf )));
Og=1/(1+exp((-(gBIAS*WgBIAS)- Od*Wdg - Oe*Weg - Of*Wfg )));

 % 'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'gray', 'lightBlue', 'orange', 'darkGreen'
 if i==1
  Exemplar1_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
  if PLOTTING==1
    figure(1);
    plot(EPOCHcount,Og,'redo');
    hold on;
  end; elseif
  i==2
   Exemplar2_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
   if PLOTTING==1
     figure(1);
     plot(EPOCHcount,Og,'black.');
     hold on;
   end;
   elseif i==3
    Exemplar3_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
    if PLOTTING==1
      figure(1);
      plot(EPOCHcount,Og,'green.');
      hold on;
    end;
    elseif i==4
     Exemplar4_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
     if PLOTTING==1
       figure(1);
       plot(EPOCHcount,Og,'blue.');
       hold on;
     end;
     elseif i==5
      Exemplar5_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
      if PLOTTING==1
        figure(1);
        plot(EPOCHcount,Og,'cyan.');
        hold on;
      end;
      elseif i==6
       Exemplar6_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
       if PLOTTING==1
         figure(1);
         plot(EPOCHcount,Og,'magenta.');
         hold on;
       end;
       elseif i==7
        Exemplar7_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
        if PLOTTING==1
```


```
          figure(1);
          plot(EPOCHcount,Og,'yellow.');
          hold on;
        end;
        else
         Exemplar8_OutputLAST=[EXEMPLAR_331(i,1) EXEMPLAR_331(i,2) EXEMPLAR_331(i,3) Og];
         if PLOTTING==1
           figure(1);
           plot(EPOCHcount,Og,'blacko');
           hold on;
         end;
         end;

      Exemplars_OutputLAST=[EPOCHcount/10000 Exemplar1_OutputLAST; ...
                            EPOCHcount/10000  Exemplar2_OutputLAST;  ...
                            EPOCHcount/10000  Exemplar3_OutputLAST;  ...
                            EPOCHcount/10000  Exemplar4_OutputLAST;  ...
                            EPOCHcount/10000  Exemplar5_OutputLAST;  ...
                            EPOCHcount/10000  Exemplar6_OutputLAST;  ...
                            EPOCHcount/10000  Exemplar7_OutputLAST;  ...
                            EPOCHcount/10000 Exemplar8_OutputLAST]
    error=EXEMPLAR_331(i,4)-Og;
    errorprop=error*Og*(1-Og);

    dWgBIAS=RATE*errorprop*gBIAS;
    dWdg= RATE*errorprop*Od;
    dWeg= RATE*errorprop*Oe;
    dWfg=  RATE*errorprop*Of;

    dWdBIAS=RATE*Od*(1-Od)*(errorprop*Wdg)*dBIAS;
    dWad=     RATE*Od*(1-Od)*(errorprop*Wdg)*EXEMPLAR_331(i,1);
    dWbd=     RATE*Od*(1-Od)*(errorprop*Wdg)*EXEMPLAR_331(i,2);
    dWcd=  RATE*Od*(1-Od)*(errorprop*Wdg)*EXEMPLAR_331(i,3);

    dWeBIAS=RATE*Oe*(1-Oe)*(errorprop*Weg)*eBIAS;
    dWae=     RATE*Oe*(1-Oe)*(errorprop*Weg)*EXEMPLAR_331(i,1);
    dWbe=     RATE*Oe*(1-Oe)*(errorprop*Weg)*EXEMPLAR_331(i,2);
    dWce=  RATE*Oe*(1-Oe)*(errorprop*Weg)*EXEMPLAR_331(i,3);

    dWfBIAS=RATE*Of*(1-Of)*(errorprop*Wfg)*fBIAS;
    dWaf=     RATE*Of*(1-Of)*(errorprop*Wfg)*EXEMPLAR_331(i,1);
    dWbf=     RATE*Of*(1-Of)*(errorprop*Wfg)*EXEMPLAR_331(i,2);
    dWcf= RATE*Of*(1-Of)*(errorprop*Wfg)*EXEMPLAR_331(i,3);

                 Wad=Wad+dWad; Wbd=Wbd+dWbd; Wcd=Wcd+dWcd;
     Wae=Wae+dWae; Wbe=Wbe+dWbe; Wce=Wce+dWce;
     Waf=Waf+dWaf; Wbf=Wbf+dWbf; Wcf=Wcf+dWcf;

                 Wdg=Wdg+dWdg; Weg=Weg+dWeg; Wfg=Wfg+dWfg;

                 WdBIAS=WdBIAS+dWdBIAS;
                WeBIAS=WeBIAS+dWeBIAS;
                WfBIAS=WfBIAS+dWfBIAS;
                WgBIAS=WgBIAS+dWgBIAS;
     Wdisplay=[Wad Wbd Wcd Wae Wbe Wce Waf Wbf Wcf Wdg Weg Wfg WdBIAS WeBIAS WfBIAS WgBIAS];

    n=n+1;
  end;
  end;
  EPOCHcount
  endTIME=cputime-startTIME
   if PLOTTING==1
    figure(1);                          %open figure window #1
    % axis([-120 335 -50 300]);         %define x and y axis for figure window #1
    title(['LEARNING RATE =',num2str(RATE), ' Stopping tolerance = ',num2str(STOPtolerance),' ' ...
       num2str(endTIME), ' secs of CPU time ']);
    xlabel('LEARNING EPOCHS');
    ylabel('3-3-1 NEURAL NETWORK OUTPUT');
    h = legend('000 input','001 input','010 input','011 input','100 input','101 input','110 input','111 input',8);
    hold on;
   end;
  %************************************** END 3-3-1 MAIN LOOP  *********************************
  %************************************** END  3-3-1 ARCHITECTURE ***************************
  %*****************************************************************************************************
  end;
```


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 $\boldsymbol{f}(x_0)=0$ (Wunderlich 1992). The Taylor polynomial approximation of any function $\boldsymbol{f}(x)$ is given by:

$$P_{Taylor}(x) = f(x_0) + f'(x - x_0) + f''(x_0)\left[\frac{(x - x_0)^2}{2!}\right] + \ldots\ldots\ldots + f^{n}(x)_0\left[\frac{(x - x_0)^n}{n!}\right] \quad (9)$$

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

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

for some number $\xi(x)$ between x and x_0 (Burden et al. 2000).

The error for a 15th degree Taylor polynomial approximation of the sigmoid neuron transfer function of equation (3) for (x_0=0) can be seen 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 was termed the "Clipped Sigmoid" in (Wunderlich 1992) and yields sufficient accuracy to allow learning to occur; however certain network initializations need to be specified (i.e., initial weights and bias' values must be small to allow learning to begin).

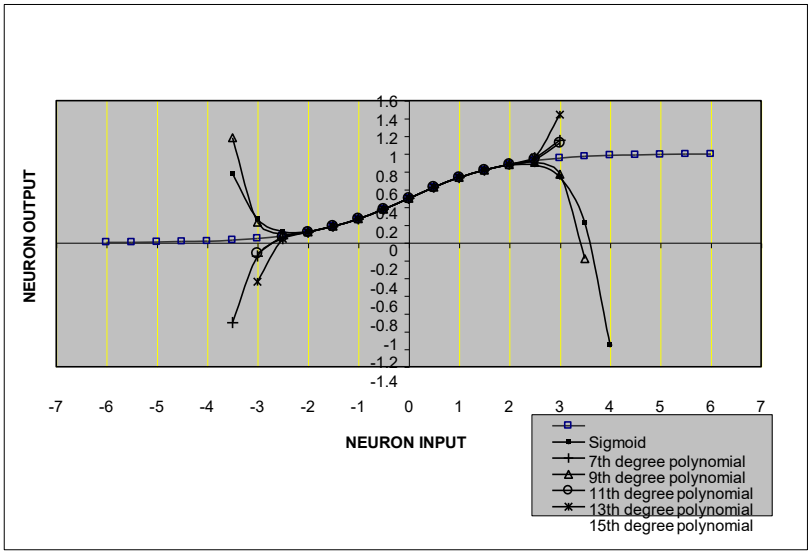


Figure 6. Taylor polynomial approximations of sigmoid: 1/(1+e^-x) expanded about x = 0

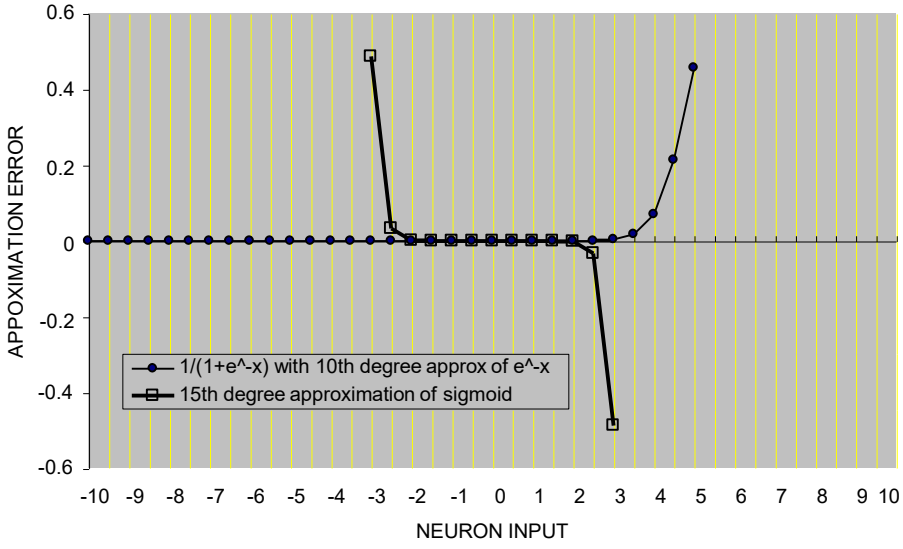


Figure 7. Error for Taylor series polynomial approximations of sigmoid: 1/(1+e^-x)

The initial research of (Wunderlich 1992) is extended here to include more precise polynomial approximations of the sigmoid. The approximation techniques considered are:

1. Cubic Spline Polynomial
2. Hermite Polynomial
3. Divided-Difference Polynomial

Cubic Spline was only given preliminary consideration since it is "piecewise" requiring different polynomials for different parts of the input domain. This would require each neuron computation to apply 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 $(2n + 1)$ degree Hermite polynomial approximation of a function $f(x)$ with points evaluated at $(x_0, x_1, \ldots, x_n)$ is given in by:

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

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

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

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

with error:

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

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 $(-3.5 > X < 3.5)$.

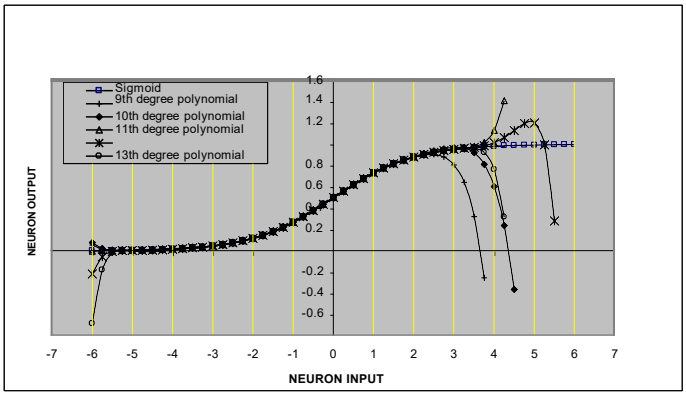


Figure 8. Hermite polynomial approximations of sigmoid: 1/(1+e^-x) with evaluation points at x = [-6 to +6 at 0.25 intervals]

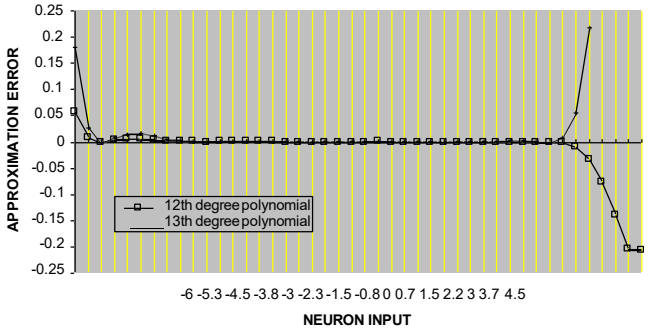


Figure 9. Error for 12th and 13th degree hermite polynomial approximations of sigmoid: 1/(1+e^-x)

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

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

where $a_0 = f[x_0] = f(x_0)$ (17)

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

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

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

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

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 set of input values). Two 12[th] 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.

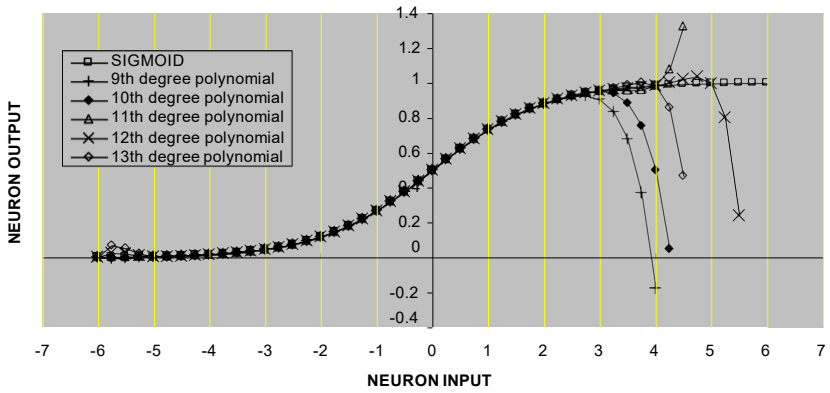


Figure 10. Divided-difference polynomial approximations of sigmoid: 1/(1+e^-x) with evaluation points at x = [- 6, - 5,- 4, - 3, - 2, - 1, - . 5, .5, 1, 2, 3, 4, 5, 6]

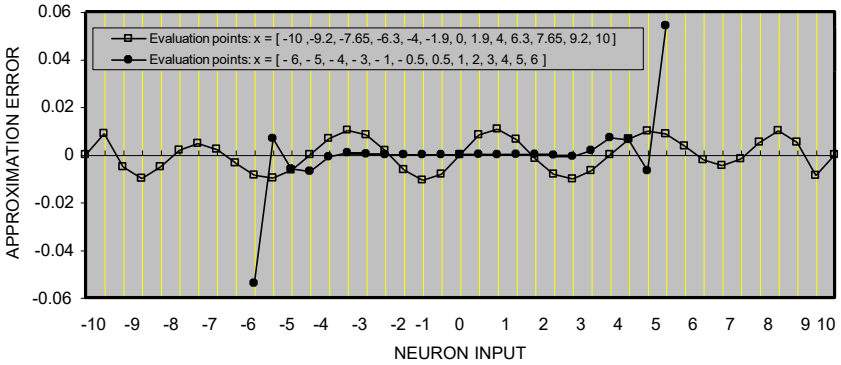
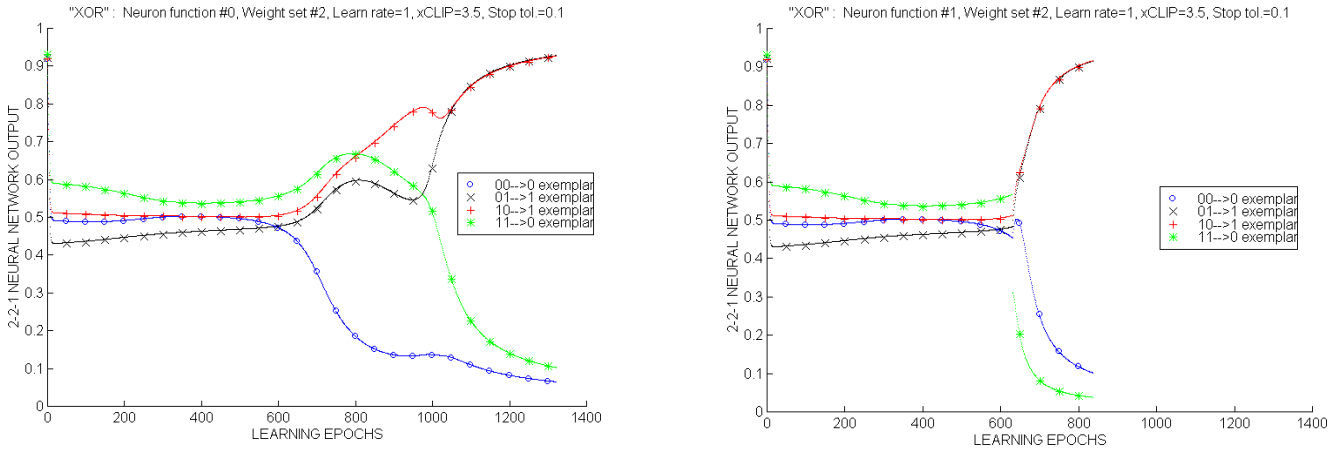


Figure 11. Error for 12th degree divided-difference polynomial approximations of sigmoid: 1/(1+e^-x)

To validate the choice of the Divided-Difference polynomial approximations of the sigmoid shown above, a neural network simulation was created and tested for all two-input logic gates including the non-linearly-seperable "XOR" used for the simulation runs shown below. The first test evaluates the effect of clipping the standard sigmoid (i.e., no polynomial approximation). Figure 12 shows that clipping does not only allow learning, but can actually improve learning time for some cases. In Fig. 13, the Divided Diffence polynomial approximation of the sigmoid is shown to allow successful learning. This simulation is clipped at -5.25 > X < 5.25 to show the robostness of the method (i.e., even though it works with a domain of –10 > X < 10).



(a) (b)

Figure 12. Comparison of backpropagation learning XOR using (a) Standard sigmoid and (b) A sigmoid "clipped" outside of –3.5 >X < 3.5.

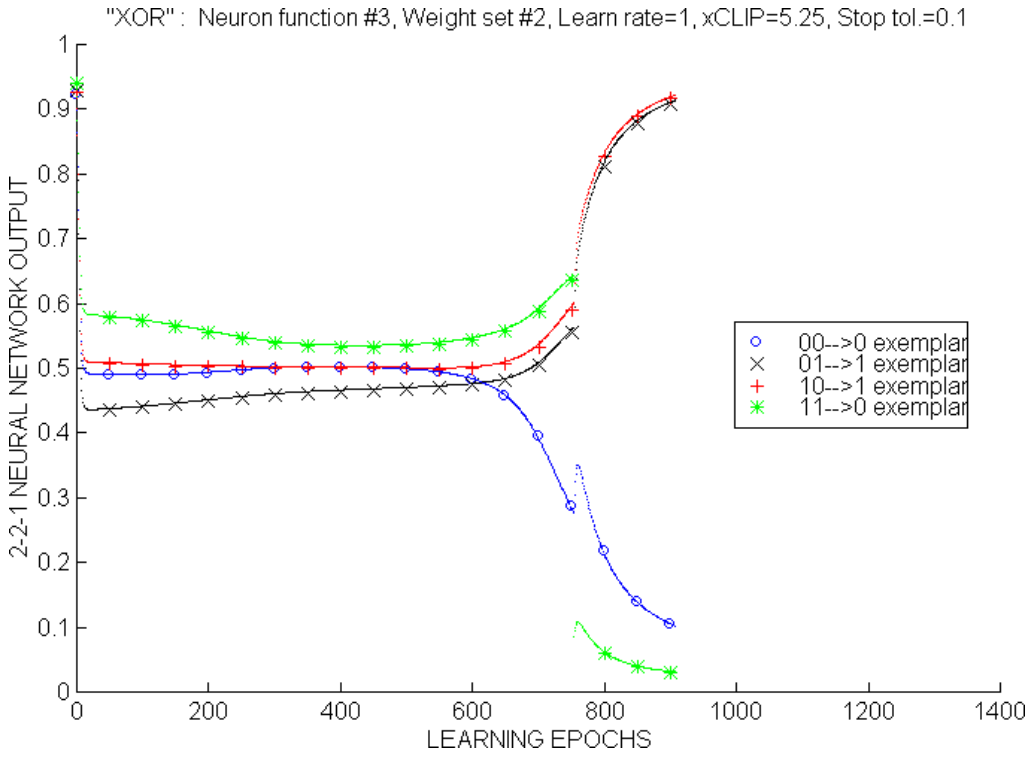


Figure 13. Backpropagation learning using 12th degree Divided-Difference polynomial approximation of sigmoid, and "clipping" outside of –5.25 >X <5.25.

Here is the Matab code to test the different polynomial numerical methods of implementing the sigmoid transfer function, and to test the effects of clipping the transfer function:

```
%*************************************************************************************************
%                          A 2-2-1 back-propagation Neural Network
%                                by Joseph T Wunderlich PhD
%                          Including "clipped sigmoid" option
%                        and Taylor Series (T) or Divided-Difference (DD)
%                          Polynomial Approximations of the Sigmoid
%
%FILE: NN6.m   Last updated:~4/12/04
%*************************************************************************************************
% Number of Training Epochs to Convergence:
% learning RATE=1 or 5?

%                                 XOR  XORC     XORC     XORC     XORCT      XORCDD    XORCDD
%         INITIAL WEIGHTS             clip@     clip@    clip@    clip@       clip@     clip@
%    (Wca Wda Wcb Wdb Wec Wed)         +-3.5     +-5.25   +-10     +-3.5      +-5.25    +-10
%Wt set1 (.4 .5 .6 .7  .8    .9)   791  791(Y)   791(N)   791(N)   4141(Y)    7859(N)   747(N)
%Wt set2 (.5 .6 .7 .8  .9  1.0)   1287 3336(Y)  1291(Y)  1287(Y)  4200(Y)    1120(Y)   991(Y)
%Wt set3 (.6 .7 .8 .9  1.0 1.1)   481  3263(Y)  480Y(Y)  481(N)   4112(Y)    1107(Y)   1082(Y)
%Wt set4 (.7 .8 .9 1.0 1.1 1.2)   577  3236(Y)  559Y(Y)  568(Y)   4193(Y)    1118(Y)   1123(Y)
%
% Y = clipping observed in at least one neuron during training
% N = no clipping observed during training
% input neurons: a,b   hidden neurons: c,d     output neuron: e
%
%=================================================================================
%********* INPUT *********
WEIGHTset=2;
 if WEIGHTset==0     Wac=.04; Wad=.05; Wbc=.06; Wbd=.07; Wce=.08; Wde=.09; %Wt set0
 elseif WEIGHTset==1 Wac=.4; Wad=.5;    Wbc=.6;      Wbd=.7; Wce=.8; Wde=.9; %Wt set1
 elseif WEIGHTset==2 Wac=.5;  Wad=.6;   Wbc=.7;     Wbd=.8; Wce=.9; Wde=1.0; %Wt set2
 elseif WEIGHTset==3 Wac=.6;  Wad=.7;   Wbc=.8;    Wbd=.9; Wce=.10; Wde=1.1; %Wt set3
 elseif WEIGHTset==4 Wac=.7; Wad=.8;    Wbc=.9;   Wbd=1.0; Wce=1.1; Wde=1.2; %Wt set4
 elseif WEIGHTset==5 Wac=1;    Wad=2;   Wbc=3;    Wbd=4;   Wce=5;   Wde=6;   %Wt set5
 elseif WEIGHTset==6 Wac=10;  Wad=20;   Wbc=30;      Wbd=40; Wce=50; Wde=60; %Wt set6
 end;
 WcBIAS=1;  WdBIAS=1;   WeBIAS=1;

ARCHITECTURE=1;               % (1) means 2-2-1 this is for future expansion
RATE=1;                       % Learning Rate
GATE=2;                       % Pick a logic GATE to learn (0)AND (1)OR (2)XOR (3)other
      if GATE==0% AND GATE                          %f0:434 f1:522 f2:499
                EXEMPLAR=[0 0 0;0 1 0;1 0 0;1 1 1]; % input1 input2 desiredoutput; etc.
  elseif GATE==1% OR GATE                           %f0:316 f1:318 f2:260
                EXEMPLAR=[0 0 0;0 1 1;1 0 1;1 1 1]; %
  elseif GATE==2% XOR GATE
                EXEMPLAR=[0 0 0;0 1 1;1 0 1;1 1 0]; %
  elseif GATE==3% Other GATE
                EXEMPLAR=[0 0 1;0 1 1;1 0 1;1 1 1]; % SPECIFY "OTHER" GATE HERE
               end;

xCLIP=3.5;                      % Clip transfer function output at +-this x value
TRANSFERfunc=3;                 % Pick transfer function:
                                % (0)sigmoid w/ matlab EXP (no clipping)
                                % (1)sigmoid w/clipped matlab EXP
                                % (2)sigmoid w/clipped nth order Taylor polynmial aprx of EXP
                                % (3)sigmoid w/clipped nth order divided-diff polynmial aprx of SIGMOID!
order=10;                       % Order of TAYLOR polynomial if approximating
DivDiffPOLYpointSET=2;          % Anchor points for divided diffence polynomial:
                                %  set(1) 10, 9,    8,    7,    6, 3,    0
                                %  set(2) 10, 9.2, 7.65, 6.3, 4, 1.9, 0
EPOCHcountMAX=100000;           % Stop if goal not reached after this many iterations
STOPtolerance=.1;               % Although driving toward 0 or 1, stop at 0-tol and 1-tol
ROUNDtoNdigits=15;              % Round set(2) divdiff poly coeficients to N digits left of
                                %  decimal, and LOOSE A DEGREE if N < 10 (input 9,10,or 15)

NoGraphics=0;                  % Turn off graphics to reduce execution time
Exemplar1_color='b.';           % Color and style of exemplar data point
Exemplar2_color='k.';           %
Exemplar3_color='r.';           %
Exemplar4_color='g.';           %
Exemplar1_colorINTERMIT='bo';   % Color and style of exemplar data intermittant point
Exemplar2_colorINTERMIT='kx';   %
Exemplar3_colorINTERMIT='r+';   %
Exemplar4_colorINTERMIT='g*';   %
```


```
INTERMITindex=50;               % Plot extra symbol intermittently
xAXISmax=1400;                  % Define max of scale on x axis
%*********** INITIALIZATION ***********************************************************
 WcBIAS=1; WdBIAS=1;    WeBIAS=1;
 cBIAS=1;   dBIAS=1;     eBIAS=1;
 Exemplar1_OutputLAST=[.1 .1 .1];      %just to get it started
 Exemplar2_OutputLAST=[.1 .1 .1];
 Exemplar3_OutputLAST=[.1 .1 .1];
 Exemplar4_OutputLAST=[.1 .1 .1];
 Exemplar1_Oc_LAST=0;  Exemplar1_Od_LAST=0;  Exemplar1_Oe_LAST=0;
 Exemplar2_Oc_LAST=0;  Exemplar2_Od_LAST=0;  Exemplar2_Oe_LAST=0;
 Exemplar3_Oc_LAST=0;  Exemplar3_Od_LAST=0;  Exemplar3_Oe_LAST=0;
 Exemplar4_Oc_LAST=0; Exemplar4_Od_LAST=0; Exemplar4_Oe_LAST=0;
 CLIPcountHIGH=[0 0 0];                %counting clips near 1 at neuron c,d,e
 CLIPcountLOW=[0 0 0];                        %counting clips near 0 at neuron c,d,e
 EPOCHcount=0;
 intermittantPLOTflag=0; % Reset flag
 intermittantPLOTcount=0; % Reset count
 n=1;
 %Polynomial Coefficients:
 C_Mclaurin=[1 1/2 1/6 1/24 1/120 1/720 1/5040 1/40320 1/362880 1/3628800 0 0 0       0 0 0  0];

%*****************************************************************************
%************************  MAIN LOOP  ****************************************
%*****************************************************************************
while ((EPOCHcount) < EPOCHcountMAX)& ...
       ((abs(Exemplar1_OutputLAST(3)-EXEMPLAR(1,3))> STOPtolerance)| ...
        (abs(Exemplar2_OutputLAST(3)-EXEMPLAR(2,3))> STOPtolerance)| ...
        (abs(Exemplar3_OutputLAST(3)-EXEMPLAR(3,3))> STOPtolerance)| ...
        (abs(Exemplar4_OutputLAST(3)-EXEMPLAR(4,3))> STOPtolerance))

  EPOCHcount=EPOCHcount+1;
  intermittantPLOTcount=intermittantPLOTcount+1;
  if intermittantPLOTcount==INTERMITindex
     intermittantPLOTflag=1;
  end;
%************************************ INITIAL GRAPHICS ************************
if (NoGraphics==0) & (EPOCHcount==2)
 figure(1);                      %open figure window #1
 % axis([-120 335 -50 300]);    %define x and y axis for figure window #1
 axis([0 xAXISmax 0 1]);    %define x and y axis for figure window #1
 xlabel('LEARNING EPOCHS');
 ylabel('2-2-1 NEURAL NETWORK OUTPUT');
 if GATE==0       %AND GATE
  h = legend('00-->0 exemplar','01-->0 exemplar','10-->0 exemplar','11-->1 exemplar',4);
  title(['"AND" :  ',                              ...
         'Neuron func.#',num2str(TRANSFERfunc),  ...
         'Wt.set#',num2str(WEIGHTset),           ...
         'Learn rate=',num2str(RATE)             ...
         'xCLIP=',num2str(xCLIP),                ...
         'Stop tol.=',num2str(STOPtolerance)]);

 elseif GATE==1 %OR GATE
  h = legend('00-->0 exemplar','01-->1 exemplar','10-->1 exemplar','11-->1 exemplar',4);
  title(['"OR" :  ',                               ...
         'Neuron func.#',num2str(TRANSFERfunc),  ...
         'Wt.set#',num2str(WEIGHTset),           ...
         'Learn rate=',num2str(RATE)             ...
         'xCLIP=',num2str(xCLIP),                ...
         'Stop tol.=',num2str(STOPtolerance)]);

 elseif GATE==2 %XOR GATE
  h = legend('00-->0 exemplar','01-->1 exemplar','10-->1 exemplar','11-->0 exemplar',4);
  title(['"XOR" :  ',                               ...
         'Neuron function #',num2str(TRANSFERfunc), ...
         ', Weight set #',num2str(WEIGHTset),           ...
         ', Learn rate=',num2str(RATE)              ...
         ', xCLIP=',num2str(xCLIP),                 ...
         ', Stop tol.=',num2str(STOPtolerance)]);

 elseif GATE==3 %OTHER GATE
  h = legend('00-->? exemplar','01-->? exemplar','10-->? exemplar','11-->? exemplar',4);
  title(['"OTHER" :  ',                              ...
         'Neuron func.#',num2str(TRANSFERfunc), ...
         'Wt.set#',num2str(WEIGHTset),           ...
         'Learn rate=',num2str(RATE)             ...
```


```
         'xCLIP=',num2str(xCLIP),                 ...
         'Stop tol.=',num2str(STOPtolerance)]);
 end;

end;
hold on;
%************************************ END INITIAL GRAPHICS ***************

  for i=1:4
     Xc=(cBIAS*WcBIAS)+ EXEMPLAR(i,1)*Wac + EXEMPLAR(i,2)*Wbc;
     Xd=(dBIAS*WdBIAS)+ EXEMPLAR(i,1)*Wad + EXEMPLAR(i,2)*Wbd;
%*********************
   if TRANSFERfunc==0       %sigmoid with matlab EXP

      Oc=1/(1+exp(-Xc));
      Od=1/(1+exp(-Xd));
     Xe=(eBIAS*WeBIAS)+ Oc*Wce + Od*Wde;
     Oe=1/(1+exp(-Xe));
%*************************
   elseif TRANSFERfunc==1       %sigmoid with clipped matlab EXP

     if Xc > xCLIP
      Oc=1/(1+exp(-xCLIP));    CLIPcountLOW(1)=CLIPcountLOW(1)+1;
     elseif Xc < -xCLIP
      Oc=1/(1+exp(-xCLIP));    CLIPcountHIGH(1)=CLIPcountHIGH(1)+1;
     else
      Oc=1/(1+exp(-Xc));
     end;

     if Xd > xCLIP
      Od=1/(1+exp(-xCLIP));    CLIPcountLOW(2)=CLIPcountLOW(2)+1;
     elseif Xd <- xCLIP
      Od=1/(1+exp(-xCLIP));    CLIPcountHIGH(2)=CLIPcountHIGH(2)+1;
     else
      Od=1/(1+exp(-Xd));
     end;

     Xe=(eBIAS*WeBIAS)+ Oc*Wce + Od*Wde;

     if Xe > xCLIP
      Oe=1/(1+exp(-xCLIP));    CLIPcountLOW(3)=CLIPcountLOW(3)+1;
     elseif Xe < -xCLIP
      Oe=1/(1+exp(-xCLIP));    CLIPcountHIGH(3)=CLIPcountHIGH(3)+1;
     else
      Oe=1/(1+exp(-Xe));
     end;
%*************************
   elseif TRANSFERfunc==2        %using a polynomial approximation of EXP part of sigmoid
     if TRANSFERfunc==2          %sigmoid with 10th order Mclaurin polynomial approx of EXP
      C=C_Mclaurin;
     end;

     if Xc > xCLIP
       Oc=1/(1+exp(-xCLIP));   CLIPcountLOW(1)=CLIPcountLOW(1)+1;
     elseif Xc < -xCLIP
       Oc=1/(1+exp(-xCLIP));   CLIPcountHIGH(1)=CLIPcountHIGH(1)+1;
     else
       approxEXP=1;
       for j=1:order
         approxEXP=approxEXP+C(j)*(-Xc)^j;
       end;
       Oc=1/(1+approxEXP);
     end;

     if Xd > xCLIP
       Od=1/(1+exp(-xCLIP)); CLIPcountLOW(2)=CLIPcountLOW(2)+1;
     elseif Xd < -xCLIP
       Od=1/(1+exp(-xCLIP)); CLIPcountHIGH(2)=CLIPcountHIGH(2)+1;
      else
      approxEXP=1;
       for j=1:order
         approxEXP=approxEXP+C(j)*(-Xd)^j;
       end;
       Od=1/(1+approxEXP);
     end;
```


```
     Xe=(eBIAS*WeBIAS)+ Oc*Wce + Od*Wde;

     if Xe > xCLIP
       Oe=1/(1+exp(-xCLIP)); CLIPcountLOW(3)=CLIPcountLOW(3)+1;
     elseif Xe < -xCLIP
       Oe=1/(1+exp(-xCLIP)); CLIPcountHIGH(3)=CLIPcountHIGH(3)+1;
     else
       approxEXP=1;
       for j=1:order
         approxEXP=approxEXP+C(j)*(-Xe)^j;
       end;
       Oe=1/(1+approxEXP);
     end;
%*******************************
   elseif TRANSFERfunc==3 % using a divided-difference
                          %   polynomial approximation of entire sigmoid

if DivDiffPOLYpointSET==1

xo=       -10.0000;
xx=        -9.0000;
xxx=       -8.0000;
xxxx=      -7.0000;
xxxxx=     -6.0000;
xxxxxx=    -3.0000;
xxxxxxx=    0.0000;
xxxxxxxx=   3.0000;
xxxxxxxxx=  6.0000;
xxxxxxxxxx= 7.0000;
ax=         8.0000;
bx=         9.0000;
cx=        10.0000;
%dx=?;
y= 0.000045397868702;
z= 0.000077996707284;
a= 0.000066979423598;
b= 0.000038297777043;
c= 0.000016347449410;
d= 0.000008828164269;
e=  0.000001281054535;
f=-0.000000703499833;
g=  0.000000110453775;
h=-0.000000012391490;
ii=0.000000001015827;
j=-0.000000000053467;
k= 0.000000000000000;

elseif DivDiffPOLYpointSET==2

xo=        -10.0000;
xx=        -9.2000;
xxx=       -7.6500;
xxxx=      -6.3000;
xxxxx=     -4.0000;
xxxxxx=    -1.9000;
xxxxxxx=    0.0000;
xxxxxxxx=   1.9000;
xxxxxxxxx=  4.0000;
xxxxxxxxxx= 6.3000;
ax=         7.6500;
bx=         9.2000;
cx=        10.0000;
%dx=?;
if ROUNDtoNdigits==9
y= 0.000045398;
z= 0.000069539;
a= 0.000073302;
b= 0.000051342;
c= 0.000035849;
d=  0.000017214;
e=-0.000000234;
f=-0.000001299;
g=  0.000000306;
h=-0.000000037;
ii=0.000000003;
j=-0.000000000000000; % DEGREE LOST !!
```


```
k= 0.000000000000000;
elseif ROUNDtoNdigits==10
y= 0.0000453979;
z= 0.0000695392;
a= 0.0000733020;
b= 0.0000513420;
c= 0.0000358490;
d=  0.0000172141;
e=-0.0000002338;
f=-0.0000012995;
g=  0.0000003063;
h=-0.0000000374;
ii=0.0000000031;
j=-0.0000000002;
k= 0.000000000000000;
else;
y= 0.000045397868702;
z= 0.000069539156507;
a= 0.000073302054179;
b= 0.000051342042275;
c= 0.000035849029832;
d=  0.000017214138083;
e=-0.000000233764314;
f=-0.000001299511922;
g=  0.000000306352087;
h=-0.000000037368965;
ii=0.000000003135243;
j=-0.000000000163294;
k= 0.000000000000000;
end;
end;
     if Xc > xCLIP
       Oc=1/(1+exp(-xCLIP));   CLIPcountLOW(1)=CLIPcountLOW(1)+1;
     elseif Xc < -xCLIP
       Oc=1/(1+exp(-xCLIP));   CLIPcountHIGH(1)=CLIPcountHIGH(1)+1;
     else
       X=Xc;
       Oc= y+(X-xo)         *(z+(X-xx)          *(a+(X-xxx)      *(b+(X-xxxx)      ...
         *(c+(X-xxxxx)      *(d+(X-xxxxxx)      *(e+(X-xxxxxxx) *(f+(X-xxxxxxxx) ...
         *(g+(X-xxxxxxxxx) *(h+(X-xxxxxxxxxx) *(ii+(X-ax)       *(j+(X-bx)        ...
         *k))))))))))));
     end;

     if Xd > xCLIP
       Od=1/(1+exp(-xCLIP)); CLIPcountLOW(2)=CLIPcountLOW(2)+1;
     elseif Xd < -xCLIP
       Od=1/(1+exp(-xCLIP)); CLIPcountHIGH(2)=CLIPcountHIGH(2)+1;
     else
       X=Xd;
       Od= y+(X-xo)         *(z+(X-xx)          *(a+(X-xxx)      *(b+(X-xxxx)      ...
         *(c+(X-xxxxx)      *(d+(X-xxxxxx)      *(e+(X-xxxxxxx) *(f+(X-xxxxxxxx) ...
         *(g+(X-xxxxxxxxx) *(h+(X-xxxxxxxxxx) *(ii+(X-ax)       *(j+(X-bx)        ...
         *k))))))))))));
     end;

     Xe=(eBIAS*WeBIAS)+ Oc*Wce + Od*Wde;

     if Xe > xCLIP
       Oe=1/(1+exp(-xCLIP)); CLIPcountLOW(3)=CLIPcountLOW(3)+1;
     elseif Xe < -xCLIP
       Oe=1/(1+exp(-xCLIP)); CLIPcountHIGH(3)=CLIPcountHIGH(3)+1;
     else
       X=Xe;
       Oe= y+(X-xo)         *(z+(X-xx)          *(a+(X-xxx)      *(b+(X-xxxx)      ...
         *(c+(X-xxxxx)      *(d+(X-xxxxxx)      *(e+(X-xxxxxxx) *(f+(X-xxxxxxxx) ...
         *(g+(X-xxxxxxxxx) *(h+(X-xxxxxxxxxx) *(ii+(X-ax)       *(j+(X-bx)        ...
         *k))))))))))));

     end;

  end; %end "if TRANSFERfunc==" loop
%*********************************
   if i==1
      Exemplar1_Oc_LAST=Oc;
      Exemplar1_Od_LAST=Od;
      Exemplar1_Oe_LAST=Oe;
```


```
   Exemplar1_OutputLAST=[EXEMPLAR(i,1) EXEMPLAR(i,2) Oe];
   if NoGraphics==0
     figure(1);
     if  (intermittantPLOTflag==1)  |  (EPOCHcount==1)
       plot(EPOCHcount,Oe,Exemplar1_colorINTERMIT);
       else
          plot(EPOCHcount,Oe,Exemplar1_color);
        end;
   end;
   hold on;
 elseif i==2
    Exemplar2_Oc_LAST=Oc;
    Exemplar2_Od_LAST=Od;
    Exemplar2_Oe_LAST=Oe;
    Exemplar2_OutputLAST=[EXEMPLAR(i,1) EXEMPLAR(i,2) Oe];
    if NoGraphics==0
     figure(1);
     if (intermittantPLOTflag==1) | (EPOCHcount==1)
         plot(EPOCHcount,Oe,Exemplar2_colorINTERMIT);
       else
         plot(EPOCHcount,Oe,Exemplar2_color);
       end;
    end;
    hold on;
     elseif i==3
        Exemplar3_Oc_LAST=Oc;
        Exemplar3_Od_LAST=Od;
        Exemplar3_Oe_LAST=Oe;
        Exemplar3_OutputLAST=[EXEMPLAR(i,1) EXEMPLAR(i,2) Oe];
        if NoGraphics==0
         figure(1);
         if (intermittantPLOTflag==1) | (EPOCHcount==1)
           plot(EPOCHcount,Oe,Exemplar3_colorINTERMIT);
         else
           plot(EPOCHcount,Oe,Exemplar3_color);
         end;
        end;
        hold on;
            else
               Exemplar4_Oc_LAST=Oc;
               Exemplar4_Od_LAST=Od;
               Exemplar4_Oe_LAST=Oe;
               Exemplar4_OutputLAST=[EXEMPLAR(i,1) EXEMPLAR(i,2) Oe];
               if NoGraphics==0
                figure(1);
                if (intermittantPLOTflag==1) | (EPOCHcount==1)
                   plot(EPOCHcount,Oe,Exemplar4_colorINTERMIT);
                 else
                   plot(EPOCHcount,Oe,Exemplar4_color);
                end;
               end;
               hold on;
            if intermittantPLOTflag==1
               intermittantPLOTflag=0; % Reset flag
               intermittantPLOTcount=0; % Reset count
            end;
            end;

 Exemplars_OutputLAST=[EPOCHcount/10000  Exemplar1_Oc_LAST  Exemplar1_Od_LAST  Exemplar1_Oe_LAST;  ...
                       EPOCHcount/10000  Exemplar2_Oc_LAST  Exemplar2_Od_LAST  Exemplar2_Oe_LAST;  ...
                       EPOCHcount/10000  Exemplar3_Oc_LAST  Exemplar3_Od_LAST  Exemplar3_Oe_LAST;  ...
                       EPOCHcount/10000 Exemplar4_Oc_LAST Exemplar4_Od_LAST Exemplar4_Oe_LAST];

error=EXEMPLAR(i,3)-Oe;
errorprop=error*Oe*(1-Oe);

dWeBIAS=RATE*errorprop*eBIAS;
dWce=   RATE*errorprop*Oc;
dWde=   RATE*errorprop*Od;

dWcBIAS=RATE*Oc*(1-Oc)*(errorprop*Wce)*cBIAS;
dWac=   RATE*Oc*(1-Oc)*(errorprop*Wce)*EXEMPLAR(i,1);
dWbc=   RATE*Oc*(1-Oc)*(errorprop*Wce)*EXEMPLAR(i,2);
dWdBIAS=RATE*Od*(1-Od)*(errorprop*Wde)*dBIAS;
dWad=   RATE*Od*(1-Od)*(errorprop*Wde)*EXEMPLAR(i,1);
dWbd=   RATE*Od*(1-Od)*(errorprop*Wde)*EXEMPLAR(i,2);
```


```
        Wac=Wac+dWac;
        Wad=Wad+dWad;
        Wbc=Wbc+dWbc;
        Wbd=Wbd+dWbd;
        Wce=Wce+dWce;
        Wde=Wde+dWde;
        WcBIAS=WcBIAS+dWcBIAS;
        WdBIAS=WdBIAS+dWdBIAS;
        WeBIAS=WeBIAS+dWeBIAS;
        Wdisplay=[Wac Wad Wbc Wbd Wce Wde WcBIAS WdBIAS WeBIAS];    %display weights

        if NoGraphics==1
          Exemplars_OutputLAST
        else
          Exemplars_OutputLAST
        end;

   n=n+1;

 end; %end "for i=1:4" loop
%*******************************
end;   %end while loop
%*****************************************************************************
%***************** END MAIN LOOP ***********************************************
%*****************************************************************************
EPOCHcount
CLIPcountLOW
CLIPcountHIGH
%endTIME=cputime-startTIME;
%instructionCOUNT=flops;
```

6. Analysis of modifying backpropagation

To fully understand the implications of clipping the inputs to neuron transfer functions outside of specified values, one must analyze the mathematics of backpropagation learning and the derivation of equations (3 to 8) above.

For the network to learn we minimize the error (E) between actual output (O) and desired output (d):

$$E = O - d \qquad (21)$$

And learning is completed when $E = 0$ at the minimum of the error surface (such as that of the simple example shown in Figure 14) when the W's are found that satisfy the desired outputs for all input combinations. This is accomplished by taking small simulation steps (n) along a calculated gradient via partial derivatives towards the minimum. This is gradient decent learning.

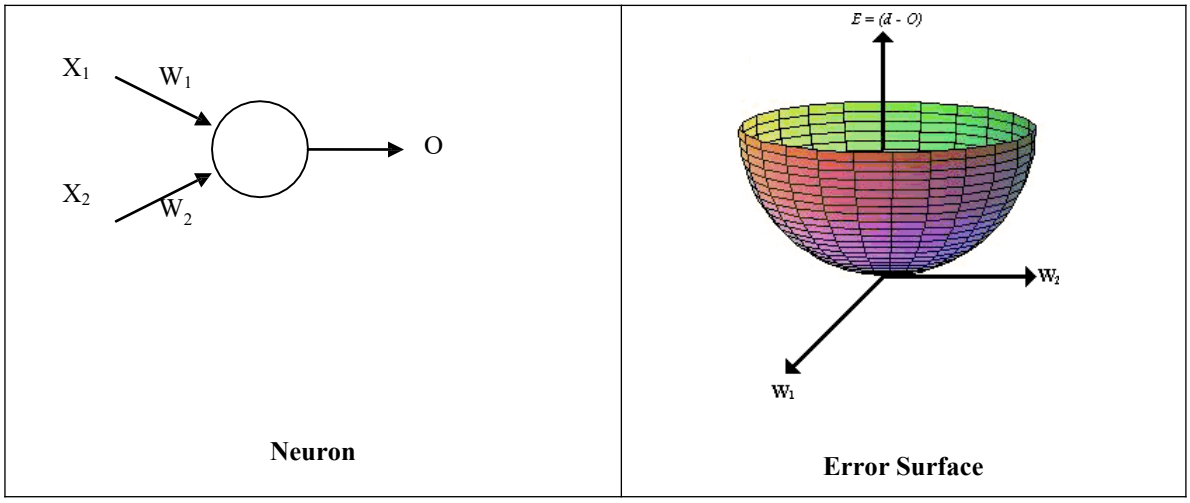


Figure 14. Error Surface for Backpropagation Learning

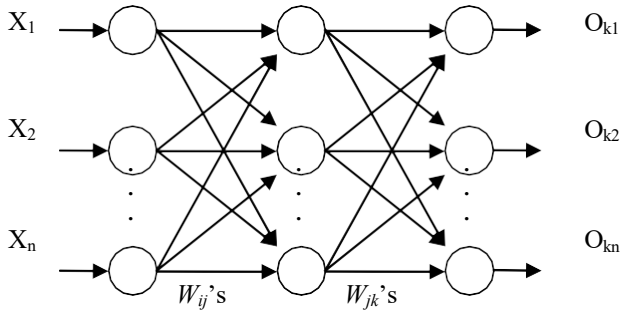


Figure 15. General three-layered (i.e., an input layer, one hidden layer, and an output layer) backpropagation network configuration.

And for the general network configuration shown in Figure 15, we derive equations (3 to 8) by minimizing the sum squared error

$$E = \frac{1}{2}\sum_{k}(d_k - O_k)^2 \qquad (22)$$

between desired outputs d_k coming out of output neuron in output layer k, and the actual outputs O_k ,by changing weights in small learning steps *n,* and in the opposite direction as the uphill sloping gradient; therefore:

$$\Delta(W_{jk}) \alpha - \left(n \frac{\partial E}{\partial W_{jk}} \right) \qquad (23)$$

and then we backpropagate the error to the next layer such that:

$$(\Delta W_{ij}) \alpha \left(- n \frac{dE}{dW_{ij}} \& \Delta W'_{jk} s \right) \qquad (24)$$

To get O_{jk} into calculations, we use the chainrule:

$$\Delta W_{jk} = -n \frac{dE}{dO_k} \frac{dO_k}{dsum_k} \frac{dsum_k}{dW_{jk}} \qquad (25)$$

where $sum_k = \sum_j O_j W_{jk}$ and $O_k = f(sum_k) = \frac{1}{1+e^{-sum_k}}$ as shown in Figure 16.

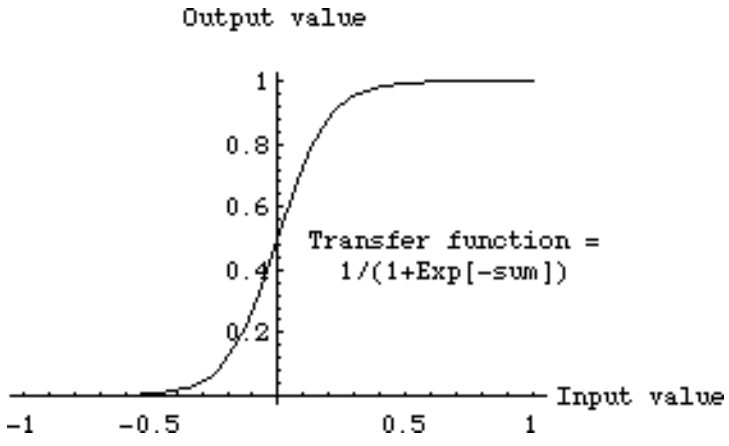


Figure 16: Nonlinear continuously differentiable neuron transfer function.

Solving each piece of (25) yields:

$\Delta W_{jk} = -n$	$* \dfrac{dE}{dO_k}$	$* \dfrac{dO_k}{dsum_k}$	$* \dfrac{dsum_k}{dW_{jk}}$
	$\dfrac{\partial\left[\frac{1}{2}\sum_k (d_k - O_k)^2\right]}{\partial O_k}$ $= \frac{1}{2} * 2(d_k - O_k)(-1)$ $= -(d_k - O_k)$	$= f'(sum_k)$	$\dfrac{\partial\left[\sum_j O_j W_{jk}\right]}{\partial W_{jk}}$ $= \sum_j O_j$ $= O_j$ for a given ΔW_{jk}

$$\Delta W_{jk} = -n(-(d_k - O_k))(f'sum_k))(O_j) \quad (26)$$

and using the Quotient Rule to evaluate $(f'\ sum_k)$ and letting sum_k be ***x***:

$$f'(x) = \frac{v(x)u'(x) - u(x)v'(x)}{(v(x))^2} \quad (27)$$

with $u(x) = 1$ and $v(x) = 1 + e^{-x}$ for our $f(sum_k) = \dfrac{1}{1+e^{-sum_k}} = f(x) = \dfrac{1}{1+e^{-x}}$:

$$\therefore f'(x) = \frac{(1+e^{-x})(0) - (1)(0 + e^{-x}(-1))}{(1+e^{-x})^2}$$

$$= \frac{e^{-x}}{(1+e^{-x})^2}$$

And then we manipulate it to get it in the form of all $O_k = f(x) = \dfrac{1}{1+e^{-x}}$ terms:

$$= \frac{e^{-x} + (1-1)}{(1+e^{-x})^2} = \frac{(e^{-x}+1) - 1}{(1+e^{-x})^2} = \frac{(1+e^{-x})}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} = \frac{1}{(1+e^{-x})} - \frac{1}{(1+e^{-x})^2}$$

$$= f(x) - f(x)^2$$

$$= f(x)(1 - f(x))$$

$$f'\ sum_k = f'(x) = O_k(1 - O_k) \quad (28)$$

To yield our equation 5 when we substitute (28) in (26): $\Delta W_{jk} = \eta * [(d_k - O_k) * O_k * (1 - O_k)] * O_j$

This has the effect of magnifying the weight changes when the net input to a given neuron is near zero as shown in Figure 17; and results in "*pushing"* neuron outputs towards the asymptotes at 0 and 1.

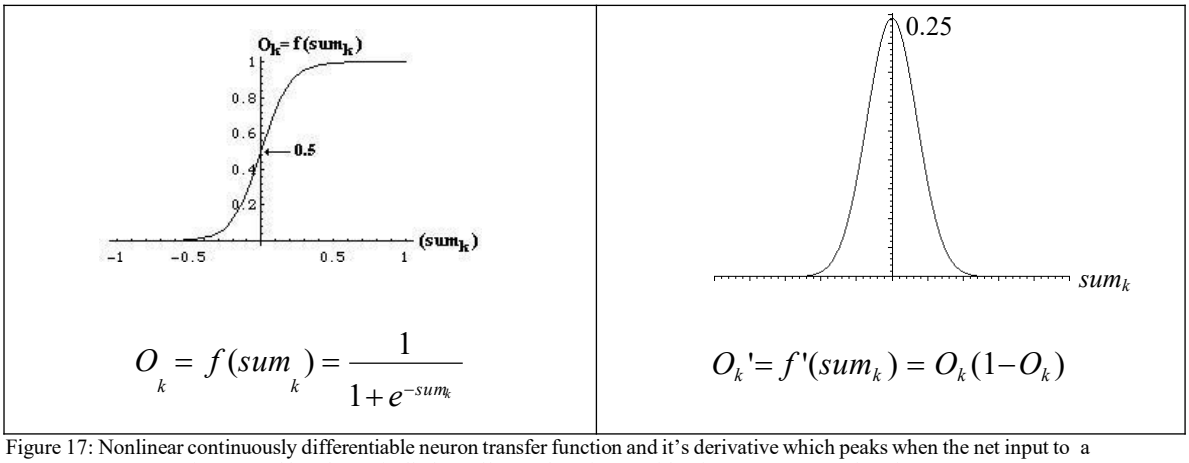


Figure 17: Nonlinear continuously differentiable neuron transfer function and it's derivative which peaks when the net input to a neuron is zero and has dramatically less effect on learning outside of a narrow range of net input.

Next, we backpropagate the error to derive ΔW_{ij} :

$$\text{Where } \Delta W_{ij} \text{ is proportional to both } \left(-n \frac{\partial E}{\partial W_{ij}} \right) \text{ and } \left(\Delta W_{jk} \right) \tag{29}$$

And to get O_k and O_j into calculations, we use the multivariable use chain rule:

$$\Delta W_{ij} = -n \frac{\partial E}{\partial W_{ij}} = n \frac{\partial E}{\partial O_k} \frac{\partial O_k}{\partial sum_k} \frac{\partial sum_k}{\partial O_j} \frac{\partial O_j}{\partial sum_j} \frac{\partial sum_j}{\partial W_{ij}} \tag{30}$$

Solving each piece of (30) yields:

$-n$	$*\frac{\partial E}{\partial O_k}$	$*\frac{\partial O_k}{\partial sum_k}$	$*\frac{\partial sum_k}{\partial O_j}$	$*\frac{\partial O_j}{\partial sum_j}$	$*\frac{\partial sum_j}{\partial W_{ij}}$
	$= -(d_k - O_k)$	$= O_k(1-O_k)$	$\frac{\partial[\sum_k O_j W_{jk}]}{\partial O_j} = \sum_k W_{jk}$	$f'(sum_j) = O_j(1-O_j)$	$\frac{\partial[\sum_i O_i W_{ij}]}{\partial W_{ij}} = \sum_i O_i = O_i$ for a given ΔW_{ij}

$$\Delta W_{ij} = \eta * (d_k - O_k) * (O_k(1-O_k)) * \sum_k W_{jk} * (O_j(1-O_j)) * O_i \tag{31}$$

Then reordering terms:

$$\Delta W_{ij} = \eta * (O_j * (1-O_j)) * (d_k - O_k) * O_k * (1-O_k) * \sum_k [W_{jk}] * O_i \tag{32}$$

And to include the backpropagating error from all output neurons from the k layer (i.e., include all k terms in the series):

$$\Delta W_{ij} = \eta * (O_j * (1-O_j)) * \sum_k [(d_k - O_k) * O_k * (1-O_k) * W_{jk}] * O_i \tag{33}$$

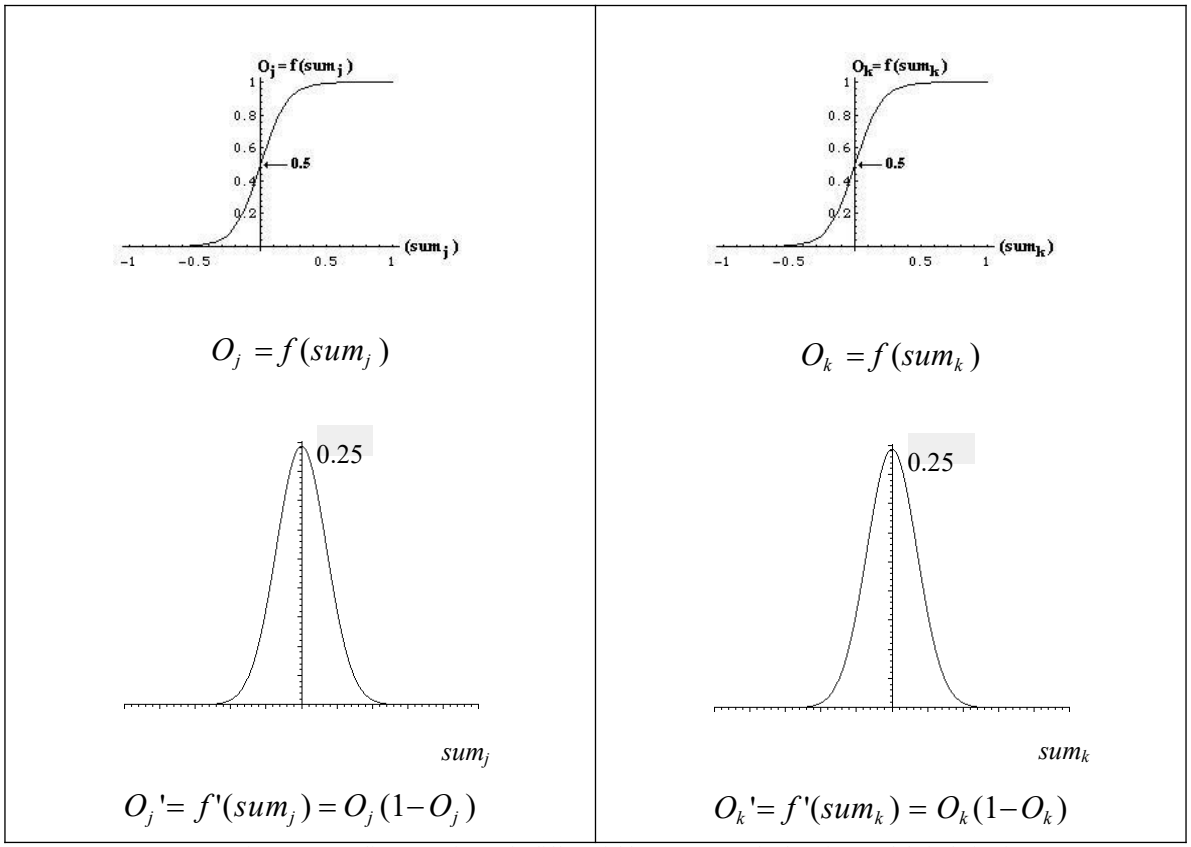


Figure 18: Nonlinear continuously differentiable neuron transfer functions and their derivatives.

The analysis above shows that the derivative of the transfer functions has the effect of magnifying the weight changes when the net input to a given neuron is near zero; and results in *pushing* neuron outputs towards asymptotes at 0 and 1. Additionally, as shown in figures 17 and 18, the derivative of the transfer function also significantly decreases outside of a narrow set of neuron input values; therefore minimizing weight changes outside of these values. This allows backpropagation learning to continue even when the neuron transfer function is clipped.

8. Building and testing

The bottom-up design in Fig. 19 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 approximatily 10,000 transistors on a 2mm x 2mm die (Wunderlich et al. 1993).

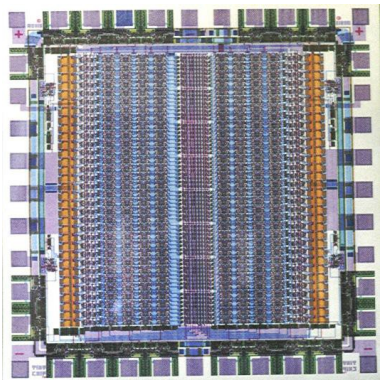


Figure 19. Bottom-up neurocomputer chip.

This chip was fabricated and bench-tested by latching in various pulse-length values, then measuring outputs for desired transient output voltage responses.
Part of the top-down neurocomputer design was filed as a Patent Disclosure document in the U.S. Patent office, and will be implemented as future research.

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 (Wunderlich 1997). Testing more complex neural network hardware (especially those with complex instruction-sets) *should* involve all of the following (Wunderlich 1997):

1. Architectural verification programs run in a simulated prototype-machine environment.
2. Digital and analog VLSI circuit simulation testing.
3. Machine architectural verification programs run on top of a VLSI circuit simulation.
4. Machine architectural verification programs run on prototype hardware.
5. Various instruction-mix and performance benchmark testing.

9. Conclusions

The top-down design proposed 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. The validity of this methodology is supported by an analysis of the mathematics of gradient decent learning under the effects of clipping the transfer function outside of the range where the overwhelming majority of learning occurs.

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 could include 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

Anderson and Rosenfeld, Neurocomputing: foundations of research. Cambridge, MA: M.I.T. Press, 1988, pp.1-3, 43, 57, 82, 89-91, 115, 123-125, 157-160, 209, 229, 243, 347, 457-459, 509, 523-525, 551-553, 635, 673.

V. Beiu,, "How to build VLSI-efficient neural chips," *in Proc. ICSC Symp. On Engineering Intelligent Systems, Tenerife*, Spain, 1998, pp. 66-75.

R. L. Burden and J. D. Faires, *"Numerical Analysis,"* Brooks Cole publishing, 7 ed., 2000.Seiffert, U., "Artificial neural networks on massively parallel computer hardware," in *Proc. ESANN 2002 European Symposium on Artificial Neural Networks*, Bruges, Belgium, 2002.

D. Campos, and J. T. Wunderlich, "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, [CD-ROM].

H. C. Card, D. K. McNeill,, and R. S. Schneider, "How forgiving is on-chip learning of circuit variations?," in *Proc. of the 5th Irish Neural Networks Conference,* Maynooth, Ireland, 1995, p. 105.

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

E. Kreyszig, Advanced Engineering Mathematics. New York: John Wiley & Sons, 1988, p.827

J. Liu and M. Brooke, “Fully parallel on-chip learning hardware neural network for real-time control,” in *Proc. of IEEE International Joint Conference on Neural Networks,* vol. 4, Jul. 1999, pp. 2323-2328.

Masaki, Hirai, and Yamada, "Neural Networks in CMOS: A Case Study", IEEE Circuits and Devices mag., July, 1990, pp.13-17.

M. Mirhassani, M. Ahmadi, and M. C. Miller, “A mixed-signal VLSI neural network with on-chip learning,” in *Proc. of the 2003 IEEE Canadian Conference on Electrical and Computer Engineering,* 2003. pp. 591-595.

S. G. Morton, “Intelligent memory chips provide a leapfrog in performnce for pattern recognition, digital signal processing, 3-D graphics and floating-point supercomputing, *Proc. Int. Neural Network Society Conf.*, Boston, 1988.

K. S. Nihal, J. A. Schlessman, and M. J. Schulte,, “Symmetric table methods for neural network approximations*,”* in *Proc. SPIE: Advanced Signal Processing Algorithms, Architectures, and Implementations XI*, San Diego, CA., 2001, Paper #: 4474-16.

M. D. Ross, "Biological Neural Networks: Models for Future Thinking Machines", NASA Tech Briefs, vol.15, number 6, June, 1991, p.130.

D. E. Rumelhart and J. L. McClelland, “*Parallel Distributed Processing*.” Cambridge, MA: M.I.T. Press, 1986.

B. Soucek, “*Neural and Concurrent Real-Time Systems, The Sixth Generation*.” New York: John Wiley & Sons, 1989.

B. Soucek and M. Soucek, Neural and Massively Parallel Computers, The Sixth Generation. New York: John Wiley & Sons, 1988.

B. Widrow and M. A. Lehr, “30 years of adaptive neural networks: perceptron, madaline, and backpropagation,” in *Proc. 2nd IEEE Intl. Conf. on Neural Networks*, 1990, pp. 1415-1442.

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

J. T. Wunderlich, et al., “Design of an artificial dendritic tree VLSI microprocessor,” University of Delaware research report, 1993.

J. T. Wunderlich, “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.

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

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, [CD-ROM].

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

J. T. Wunderlich, Top-down vs. bottom-up neurocomputer design. In *Intelligent Engineering Systems through Artificial Neural Networks, Proceedings of ANNIE 2004 International Conference, St. Louis, MO*. H. Dagli (Ed.): Vol. 14., New York, NY ASME Press, 2004, pp. 855-866.