REFERENCES

[1] M. G. Larimore, J. R. Treichler, and C. R. Johnson Jr., "SHARF: An algorithm for adapting IIR digital filters," *IEEE Trans. Acoust., Speech Signal Processing*, vol. 28, pp. 428–440, Aug. 1980.

[2] H. Fan, "A structural view of asymptotic convergence speed of adaptive IIR filtering algorithms," *IEEE Trans. Signal Processing*, vol. 41, pp. 1493–1517, Apr. 1993.

[3] P. M. S. Burt and M. Gerken, "A polyphase IIR adaptive filter: Error surface analysis and application," in *Proc. ICASSP*, vol. 3, 1997, pp. 2258–2288.

[4] M. Bellanger, G. Bonnerot, and M. Coudreuse, "Digital filtering by polyphase network: Application to sample rate alteration and filter banks," *IEEE Trans. Acoust., Speech Signal Processing*, vol. 24, no. 2, pp. 109–114, Feb. 1976.

[5] P. A. Regalia, *Adaptive IIR Filtering in Signal Processing and Control*. New York: Marcel Dekker, 1995.

[6] P. M. S. Burt, "Uma investigação sobre filtros IIR adaptativos com aplicação a uma estrutura polifásica," Doctoral dissertation, Univ. São Paulo, 1997.

[7] W. Y. Chen, J. L. Dixon, and D. L. Waring, "High bit rate digital subscriber line echo cancellation," *IEEE J. Select. Areas Commun.*, vol. 9, pp. 848–860, Aug. 1991.

[8] T. Söderström, "On the uniqueness of maximum likelihood identification," *Automatica*, vol. 11, pp. 193–197, 1975.

[9] T. Söderström and P. Stoica, "Some properties of the output error method," *Automatica*, vol. 18, pp. 93–99, 1982.

Analog VLSI Neural Network With Digital Perturbative Learning

Vincent F. Koosh and Rodney M. Goodman

***Abstract*—Two feed-forward neural-network hardware implementations are presented. The first uses analog synapses and neurons with a digital serial weight bus. The chip is trained in loop with the computer performing control and weight updates. By training with the chip in the loop, it is possible to learn around circuit offsets. The second neural network also uses a computer for the global control operations, but all of the local operations are performed on chip. The weights are implemented digitally, and counters are used to adjust them. A parallel perturbative weight update algorithm is used. The chip uses multiple, locally generated, pseudorandom bit streams to perturb all of the weights in parallel. If the perturbation causes the error function to decrease, the weight change is kept; otherwise, it is discarded. Test results from a very large scale integration (VLSI) prototype are shown of both networks successfully learning digital functions such as AND and XOR.**

***Index Terms*—Analog very large scale integration (VLSI), chip-in-loop training algorithm, learning, neural chips, neural network, neuromorphic, perturbation techniques, VLSI feed-forward neural network.**

I. INTRODUCTION

Several circuits are presented for implementing neural-network architectures. Neural networks have proven useful in areas requiring man–machine interactions such as handwriting or speech recognition. Although these neural networks can be implemented with digital microprocessors, the large growth in portable devices with limited battery life increases the need of finding custom low-power solutions. Furthermore, the area of operation of the neural network circuits can be modified from low power to high speed to meet the needs of the specific application. The inherent parallelism of neural networks allows a compact high-speed solution in analog very large scale integration (VLSI).

First, a VLSI feed-forward neural network is presented that makes use of digital weights, analog synaptic multipliers and analog neurons. The network is trained in a chip-in-loop fashion with a host computer implementing the training algorithm. The chip uses a serial digital weight bus implemented by a long shift register to input the weights. The inputs and outputs of the network are provided directly at pins on the chip.

Next, a VLSI neural network that uses a parallel perturbative weight update technique is presented. The network uses the same synapses and neurons as the previous network, but all of the local, parallel, weight update computations are performed on chip. This includes the generation of random perturbations and counters for updating the digital words where the weights are stored.

The training algorithm used in both networks is a parallel weight perturbation method. For both implementations, training results are shown for a two-input, one-output network trained with an AND function, and for a two-input, two-hidden layer, one-output network trained with an XOR function.

II. VLSI NEURAL NETWORK WITH ANALOG MULTIPLIERS AND A SERIAL DIGITAL WEIGHT BUS

Training an analog neural network directly on a VLSI chip provides additional benefits over using a computer for the initial training and then downloading the weights. The analog hardware is prone to have offsets and device mismatches. By training with the chip in the loop, the neural network will also learn these offsets and adjust the weights appropriately to account for them. A VLSI neural network can be applied in many situations requiring fast, low-power operation such as handwriting recognition for portable devices or pattern detection for implantable medical devices [2].

There are several issues that must be addressed to implement an analog VLSI neural network chip. First, an appropriate algorithm suitable for VLSI implementation must be found. Traditional error backpropagation approaches for neural network training require too many bits of floating-point precision to implement efficiently in an analog VLSI chip. Techniques that are more suitable involve stochastic weight perturbation [1], [3]–[7], where a weight is perturbed in a random direction, its effect on the error is determined, and the perturbation is kept if the error was reduced; otherwise, the old weight is restored. In this approach, the network observes the gradient rather than actually computing it.

Serial weight perturbation [3] involves perturbing each weight sequentially. This requires a number of iterations that is directly proportional to the number of weights. A significant speed-up can be obtained if all weights are perturbed randomly in parallel and then measuring the effect on the error and keeping them all if the error reduces. Both the parallel and serial methods can potentially benefit from the use of annealing the perturbation. Initially, large perturbations are applied to move the weights quickly toward a minimum. Then, the perturbation sizes are gradually decreased to achieve finer selection of the weights and a smaller error. In general, however, optimized gradient-descent techniques converge more rapidly than the perturbative techniques.

Next, the issue of how to appropriately store the weights on chip in a nonvolatile manner must be addressed. If the weights are simply stored as charge on a capacitor, they will ultimately decay due to parasitic conductance paths. One option would be to use an analog memory cell [8], [9]. This would allow directly storing the analog voltage value. However, this technique requires introducing large voltages to obtain tunneling and/or injection through the gate oxide and is still being investigated. Another approach would be to use traditional digital storage

Manuscript received October 18, 2001; revised June 20, 2002. This paper was recommended by Associate Editor G. Cauwenberghs.

The authors are with the California Institute of Technology, Pasadena, CA 91125 USA (e-mail: darkd@ieee.org; e-mail: rogo@caltech.edu).

Publisher Item Identifier 10.1109/TCSII.2002.802282.

1057-7130/02$17.00 © 2002 IEEE

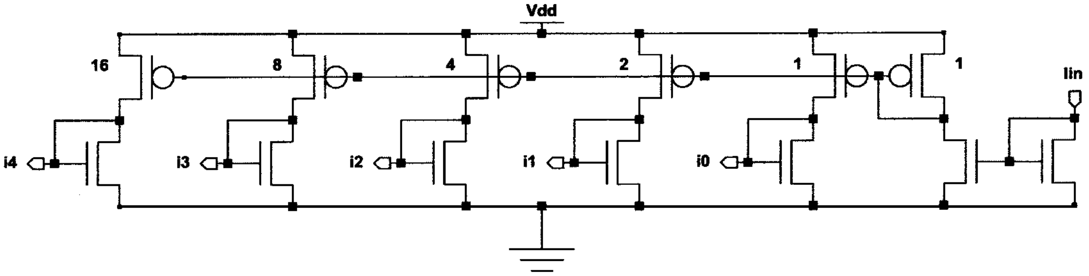

Fig. 1. Binary weighted current source circuit.

with EEPROMs. This would then require having one analog-to-digital (A/D) converter and one digital-to-analog (D/A) converter (A/D/A converters) for the weights. A single A/D/A converter would only allow a serial weight perturbation scheme that would be slow. A parallel scheme, which would perturb all weights at once, would require one A/D/A per weight. This would be faster, but would require more area. One alternative would remove the A/D requirement by replacing it with a digital counter to adjust the weight values. This would then require one digital counter and one D/A per weight.

A. Synapse

A small synapse with one D/A per weight can be achieved by first making a binary weighted current source (Fig. 1) and then feeding the binary weighted currents into diode connected transistors to encode them as voltages. These voltages are then fed to transistors on the synapse to convert them back to currents. Thus, many D/A converters are achieved with only one binary weighted array of transistors. It is clear that the linearity of the D/A will be poor because of matching errors between the current source array and synapses which may be located on opposite sides of the chip. This is not a concern because the network will be able to learn around these offsets.

The synapse [2], [6] is shown in Fig. 2. The synapse performs the weighting of the inputs by multiplying the input voltages by a weight stored in a digital word denoted by b0–b5. The sign bit, b5, changes the direction of current to achieve the appropriate sign.

In the subthreshold region of operation, the transistor equation is given by [10] $I_d = I_{d0} \exp(\kappa V_{gs}/U_t)$ and the output of the synapse is given by [2], [10]

$$\Delta I_{\rm out} = I_{\rm out+} - I_{\rm out-} = W I_0 \tanh\left(\frac{\kappa(V_{\rm in+} - V_{\rm in-})}{2U_t}\right)$$

where W is the weight of the synapse encoded by the digital word and I_0 is the least significant bit (LSB) current. Thus, in the subthreshold linear region, the output is approximately given by

$$\Delta I_{\rm out} \approx g_m \Delta V_{\rm in} = \frac{\kappa I_0}{2U_t} W \Delta V_{\rm in}.$$

In the above threshold regime, the transistor equation in saturation is approximately given by $I_D \approx K(V_{gs} - V_t)^2$. The synapse output is no longer described by a simple tanh function, but is nevertheless still sigmoidal with a wider "linear" range. In the above threshold linear region, the output is approximately given by

$$\Delta I_{\rm out} \approx g_m \Delta V_{\rm in} = 2\sqrt{K I_0}\sqrt{W}\Delta V_{\rm in}.$$

It is clear that above threshold, the synapse is not doing a pure weighting of the input voltage. However, since the weights are learned on chip, they will be adjusted accordingly to the necessary value. Furthermore, it is possible that some synapses will operate below threshold while others above, depending on the choice of LSB current. Again, on-chip learning will be able to set the weights to account for these different modes of operation.

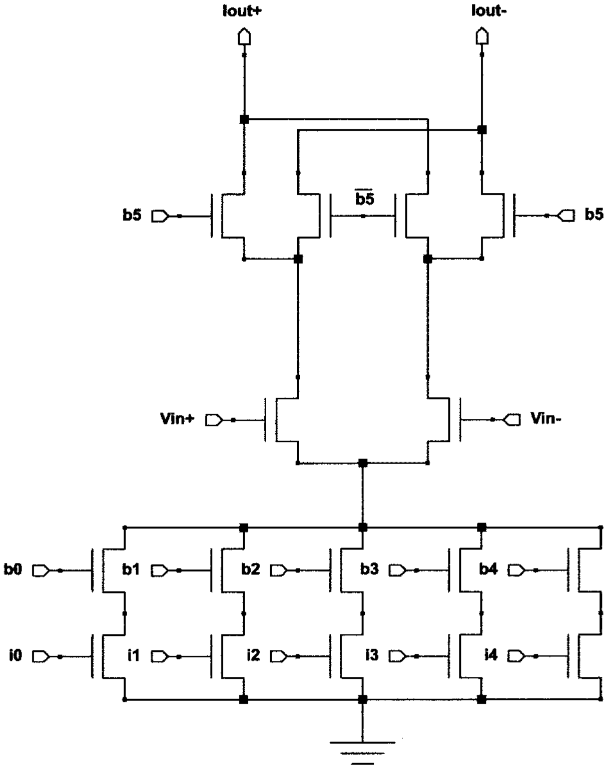

Fig. 2. Synapse circuit.

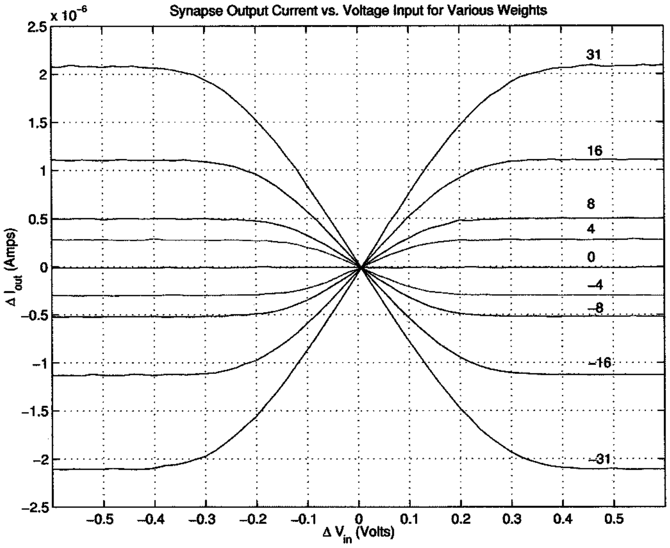

Fig. 3. Synapse differential output current as a function of differential input voltage for various digital weight settings.

Fig. 3 shows the differential output current of the synapse as a function of differential input voltage for various digital weight settings. The input current of the binary weighted current source was set to 100 nA.

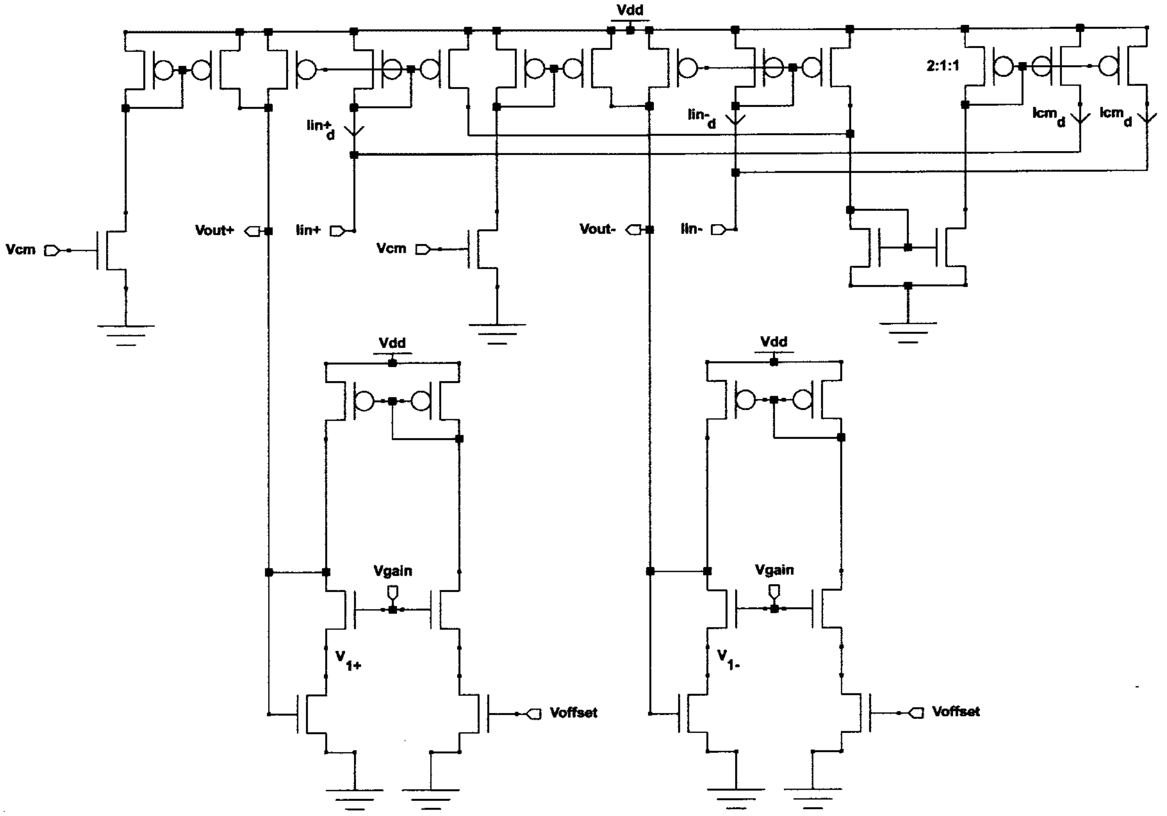

Fig. 4. Neuron circuit.

The output currents range from the subthreshold region for the smaller weights to the above threshold region for the large weights. All of the curves show their sigmoidal characteristics. Furthermore, it is clear that the width of the linear region increases as the current moves from subthreshold to above threshold. For the smaller weights, $W = 4$, the linear region spans only approximately 0.2 V–0.4 V. For the largest weights, $W = 31$, the linear range has expanded to roughly 0.6 V–0.8 V. As was discussed previously, when the current range moves above threshold, the synapse does not perform a pure linear weighting. The largest synapse output current is not 3.1 μA as would be expected from a linear weighting of 31 × 100 nA, but a smaller number. Notice that the zero crossing of ΔI_{out} occurs slightly positive of $\Delta V_{\text{in}} = 0$. This is a circuit offset that is primarily due to slight W/L differences of the differential input pair of the synapse, and it is caused by minor fabrication variations.

B. Neuron

The synapse circuit outputs a differential current that will be summed in the neuron circuit shown in Fig. 4. The neuron circuit performs the summation from all of the input synapses. The neuron circuit then converts the currents back into a differential voltage feeding into the next layer of synapses. Since the outputs of the synapse will all have a common mode component, it is important for the neuron to have common mode cancelation [2]. Since one side of the differential current inputs may have a larger share of the common mode current, it is important to distribute this common mode to keep both differential currents within a reasonable operating range. If $\Delta I = I_{\text{in}+} - I_{\text{in}-} = I_{\text{in}+_{\text{d}}} - I_{\text{in}-_{\text{d}}}$ and $I_{\text{cm}} = (I_{\text{in}+} + I_{\text{in}-})/2 = (I_{\text{in}+_{\text{d}}} + I_{\text{in}-_{\text{d}}} + 2I_{\text{cm}_{\text{d}}})/2 = 2I_{\text{cm}_{\text{d}}}$, then the neuron circuit ensures that $I_{\text{in}+_{\text{d}}} = I_{\text{in}+} - I_{\text{cm}}/2 = \Delta I/2 + I_{\text{cm}}/2$ and $I_{\text{in}-_{\text{d}}} = I_{\text{in}-} - I_{\text{cm}}/2 = -\Delta I/2 + I_{\text{cm}}/2$.

If the ΔI is of equal size or larger than I_{cm}, the transistor with $I_{\text{in}-_{\text{d}}}$ may begin to cutoff and the previous equations would not exactly hold; however, the current cutoff is graceful and should not normally affect performance. With the common mode signal properly equalized, the differential currents are then mirrored into the current-to-voltage transformation stage. This stage effectively takes the differential input currents and uses a transistor in the triode region to provide a differential output. This stage will usually be operating above threshold, because the V_{offset} and V_{cm} controls are used to ensure that the output voltages are approximately mid-rail. This is done by simply adding additional current to the diode connected transistor stack. Having the outputs mid-rail is important for proper biasing of the next stage of synapses. The above threshold transistor equation in the triode region is given by $I_d = 2K(V_{gs} - V_t - V_{ds}/2)V_{ds} \approx 2K(V_{gs} - V_t)V_{ds}$ for small enough V_{ds}, where $K = \mu C_{\text{ox}} W/L$. If K_1 denotes the prefactor of the cascode transistor and K_2 denotes the same for the transistor with gate V_{out}, the voltage output of the neuron will then be given by

$$V_{\text{out}} = \frac{I_{\text{in}}}{2K_2(V_{\text{gain}} - V_t) - 2\sqrt{\frac{K_2^2}{K_1} I_{\text{in}}}} + V_t$$

which for $K_1 = K_2$ converts to

$$V_{\text{out}} = \frac{I_{\text{in}}}{2K(V_{\text{gain}} - V_t) - 2\sqrt{KI_{\text{in}}}} + V_t.$$

For small input current I_{in}, the effective resistance is

$$R \approx \frac{1}{2K(V_{\text{gain}} - V_t)}.$$

Thus, it is clear that V_{gain} can be used to adjust the effective gain of the stage.

Fig. 5 shows how the neuron differential output voltage ΔV_{out} varies as a function of differential input current for several values of V_{gain}. The neuron shows fairly linear performance with a sharp bend on either

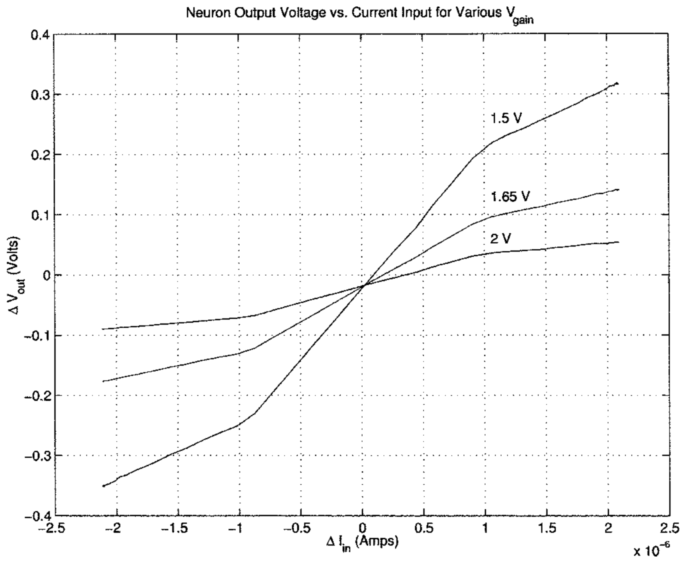

Fig. 5. Neuron differential output voltage, $\Delta V_{\rm out}$, as a function of $\Delta I_{\rm in}$, for various values of $V_{\rm gain}$.

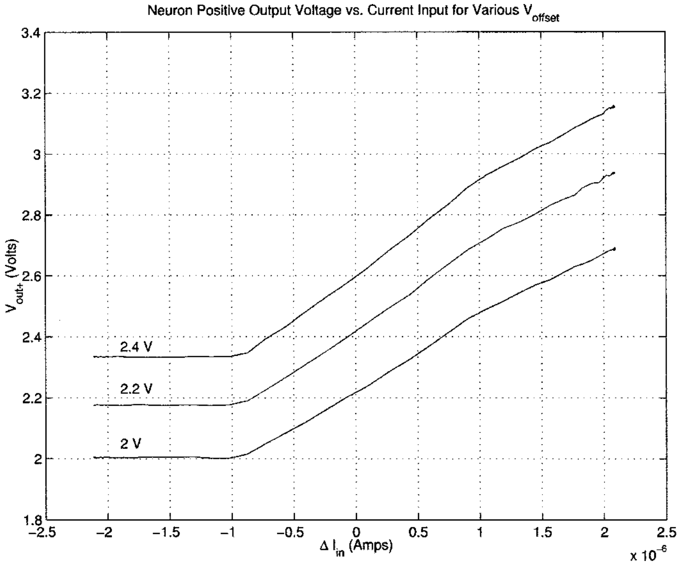

Fig. 6. Neuron positive output, $V_{\rm out+}$, as a function of $\Delta I_{\rm in}$, for various values of $V_{\rm offset}$.

side of the linear region. This sharp bend occurs when one of the two linearized, diode connected transistors with gate attached to $V_{\rm out}$ turns OFF.

Fig. 6 displays only the positive output $V_{\rm out+}$ of the neuron. The diode connected transistor, with gate attached to $V_{\rm out+}$ turns OFF where the output goes flat on the left side of the curves. This corresponds to the left bend point in Fig. 5. The baseline output voltage corresponds roughly to $V_{\rm offset}$, however, as $V_{\rm offset}$ increases in value, its ability to increase the baseline voltage is reduced because of the cascode transistor on its drain. At some point, especially for small values of $V_{\rm gain}$, the $V_{\rm cm}$ transistor becomes necessary to provide additional offset. Overall, the neuron shows very good linear current-to-voltage conversion with separate gain and offset controls.

C. Feed-forward Network

Using the synapse and neuron circuit building blocks, it is possible to construct a multilayer feed-forward neural network. A serial weight bus is used to apply weights to the synapses. The weight bus merely consists of a long shift register to cover all of the possible weight and threshold bits. The input to the serial weight bus comes from a host computer which implements the learning algorithm.

```
Initialize Weights;
Get Error;
while(Error > Error Goal);
   Perturb Weights;
   Get New Error;
   if (New Error < Error),
      Weights = New Weights;
      Error = New Error;
   else
      Restore Old Weights;
   end
end
```

Fig. 7. Parallel perturbative algorithm.

Note that the nonlinear squashing function is actually performed in the next layer of synapse circuits rather than in the neuron as in a traditional neural network. This is equivalent as long as the inputs to the first layer are kept within the linear range of the synapses. However, equivalence is unnecessary as long as the chip is trained in the loop. Also, for digital functions, the inputs need not be constrained as the synapses will pass roughly the same current regardless of whether the digital inputs are at the flat part of the synapse curve near the linear region or all the way at the end of the flat part of the curve. Furthermore, for nondifferential digital signals, it is possible to simply tie the negative input to mid-rail and apply the standard digital signal to the positive synapse input. The biases, or thresholds, for each neuron are simply implemented as synapses tied to fixed bias voltages. The biases are learned in the same way as the weights.

Also, depending on the type of network outputs desired, additional circuitry may be needed for the final squashing function. For example, if a roughly linear output is desired, the differential output can be taken directly from the neuron outputs. In the current implementation, a differential to single ended converter is placed on the output neuron. The gain of this converter determines the size of the linear region for the final output. Normally, during training, a somewhat linear output with low gain is desired to have a reasonable slope to learn the function on. However, after training, it is possible to take the output after a dual inverter digital buffer to get a strong standard digital signal to send off-chip or to other sections of the chip.

D. Training Algorithm

The neural network is trained by using a parallel perturbative weight update rule [1]. The perturbative technique requires generating random weight increments to adjust the weights during each iteration. These random perturbations are then applied to all of the weights in parallel. In batch mode, all input training patterns are applied and the error is accumulated. This error is then checked to see if it was higher or lower than the unperturbed iteration. If the error is lower, the perturbations are kept, otherwise they are discarded. This process repeats until a sufficiently low error is achieved. An outline of the algorithm is given in Fig. 7. Since the weight updates are calculated offline, other suitable algorithms may also be used. For example, it is possible to apply an annealing schedule wherein large perturbations are initially applied and gradually reduced as the network settles.

E. Test Results

A chip implementing the above circuits was fabricated in a 1.2-μm CMOS process [11]. All synapse and neuron transistors were 3.6 μm/3.6 μm to keep the layout small. The unit size current source transistors were also 3.6 μm/3.6 μm. An LSB current of 100 nA was

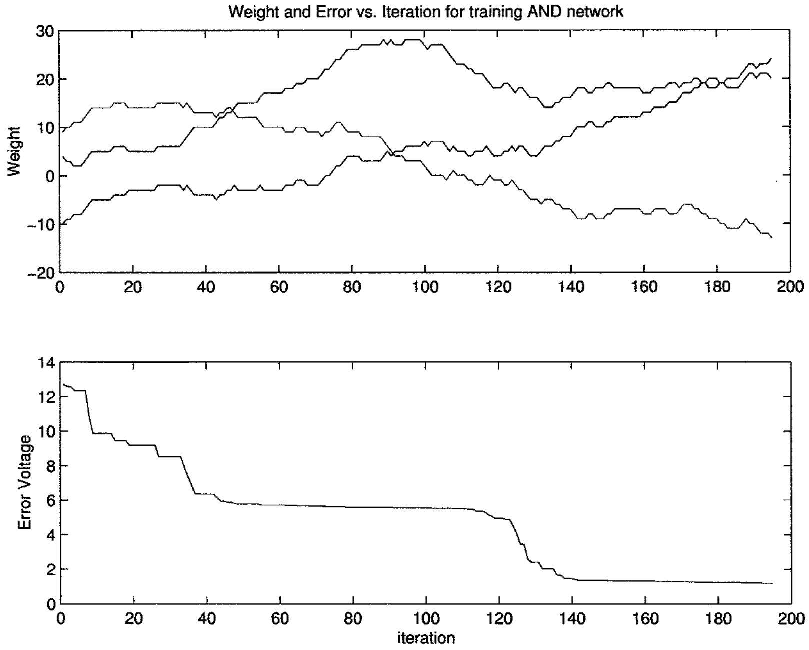

Fig. 8. Training of a 2 : 1 network with AND function.

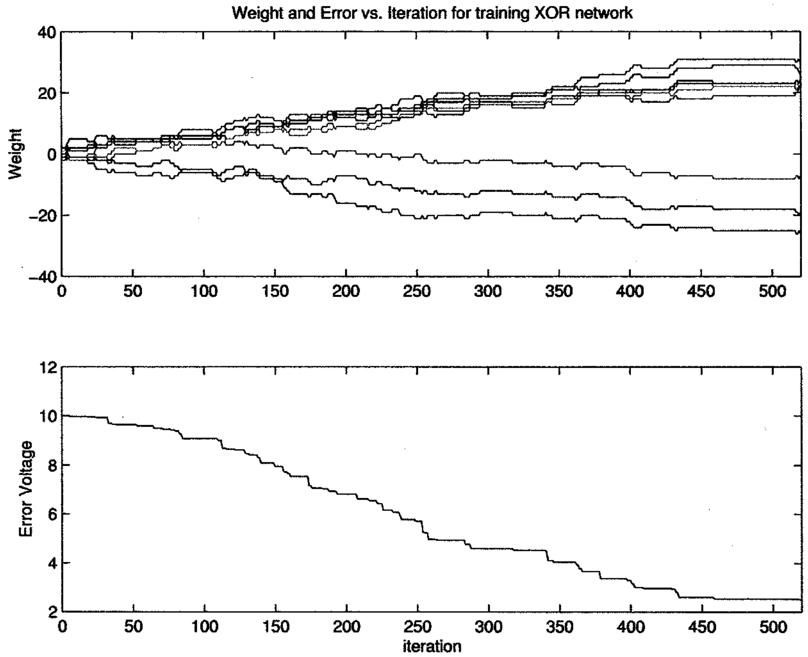

Fig. 9. Training of a 2 : 2 : 1 network with XOR function starting with small random weights.

chosen for the current source. The above neural network circuits were trained with some simple digital functions such as two-input AND and two-input XOR. The results of some training runs are shown in Figs. 8 and 9. As can be seen from the figures, the network weights slowly converge to a correct solution. Since the training was done on digital functions, a differential to single-ended converter was placed on the output of the final neuron. This was simply a five-transistor transconductance amplifier. The error voltages were calculated as a total sum voltage error over all input patterns at the output of the transconductance amplifier. Since V_{dd} was 5 V, the output would only easily move to within about 0.5 V from V_{dd} because the transconductance amplifier had low gain. Thus, when the error gets to around 2 V, it means that all of the outputs are within about 0.5 V from their respective rail and functionally correct. A double inverter buffer can be placed at the final output to obtain good digital signals. At the beginning of each of the training runs, the error voltage starts around or over 10 V indicating that at least two of the input patterns give an incorrect output.

Fig. 8 shows the results from a two-input, one-output network learning an AND function. This network has only two synapses and one bias for a total of three weights. The weight values can go from -31 to $+31$ because of the 6 b D/A converters used on the synapses.

Fig. 9 shows the results of training a two-input, two-hidden unit, one-output network with the XOR function. The weights are initialized as small random numbers. The weights slowly diverge and the error monotonically decreases until the function is learned. As with

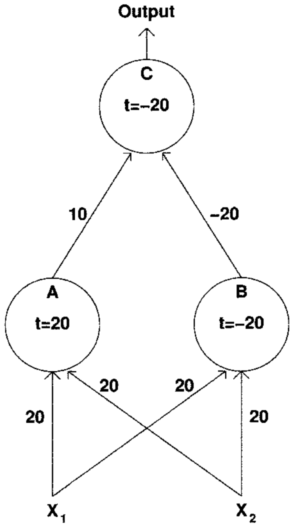

Fig. 10. Network with "ideal" weights chosen for implementing XOR.

gradient techniques, occasional training runs resulted in the network getting stuck in a local minimum.

An ideal neural network with weights appropriately chosen to implement the XOR function is shown in Fig. 10. The inputs for the network and neuron outputs are chosen to be (-1, $+1$), as opposed to (0,1). This choice is made because the actual synapses which implement the squashing function give maximum outputs of $+/-I_{\mathrm{out}}$. The neurons are assumed to be hard thresholds. The function computed by each of the neurons is given by $Out = \mathrm{sgn}((\sum_i W_i X_i) + t)$. The weights were chosen to be within the range of possible weight values, -31 to $+31$, of the actual network. This ideal network would perform perfectly well with smaller weights. For example, all of the input weights could be set to 1 as opposed to 20, and the A and B thresholds would then be set to 1 and -1, respectively, without any change in the network function. However, weights of large absolute value within the possible range were chosen (20 as opposed to 1), because small weights would be within the linear region of the synapses. Also, small weights such as $+/-1$ might tend to get flipped due to circuit offsets. A SPICE simulation was done on the actual circuit using these ideal weights and the outputs were seen to be the correct XOR outputs.

Fig. 11 shows the $2:2:1$ network trained with XOR, but with the initial weights chosen as the mathematically correct weights for the ideal synapses and neurons. Although the ideal weights should, both in theory and based on simulations, start off with correct outputs, the offsets and mismatches of the circuit cause the outputs to be incorrect. However, since the weights start near where they should be, the error goes down rapidly to the correct solution. This is an example of how a more complicated network could be trained on computer first to obtain good initial weights and then the training could be completed with the chip in the loop. Also, for more complicated networks, using a more sophisticated model of the synapses and neurons that more closely approximates the actual circuit implementation would be advantageous for computer pretraining.

III. Parallel Perturbative VLSI Neural Network

A fully parallel perturbative algorithm cannot truly be realized with a serial weight bus, because the act of changing the weights is performed by a serial operation. Thus, it is desirable to add circuitry to allow for parallel weight updates.

First, a method for applying random perturbation is necessary. The randomness is necessary because it defines the direction of search for finding the gradient. Since the gradient is not actually calculated, but observed, it is necessary to search for the downward gradient. It is possible to use a technique which does a nonrandom search. However, since no prior information about the error surface is known, in the worst case a nonrandom technique would spend much more time investigating upward gradients which the network would not follow.

A conceptually simple technique for generating random perturbations would be to amplify the thermal noise of a diode or resistor. Unfortunately, the extremely large value of gain required for the amplifier makes the amplifier susceptible to crosstalk. Any noise generated from neighboring circuits would also get amplified. Since some of this noise may come from clocked digital sections, the noise would become very regular, and would likely lead to oscillations rather than the uncorrelated noise sources that are desired.

Such a scheme was attempted with separate thermal noise generators for each neuron [14]. The gain required for the amplifier was nearly one million and highly correlated oscillations of a few megahertz were observed among all the noise generators. Therefore, another technique is required.

Instead, the random weight increments can be generated digitally with linear feedback shift registers that produce a long pseudorandom sequence. These random bits are used as inputs to a counter that stores and updates the weights. The counter outputs go directly to the D/A converter inputs of the synapses. If the weight updates lead to a reduction in error, the update is kept. Otherwise, an inverter block is activated which inverts the counter inputs coming from the linear feedback shift registers. This has the effect of restoring the original weights. A block diagram of the full neural network circuit function is provided in Fig. 12.

A. Multiple Pseudorandom Bit Stream Circuit

Linear feedback shift registers are a useful technique for generating pseudorandom noise [12], [13]. However, a parallel perturbative neural network requires as many uncorrelated noise sources as there are weights. Unfortunately, an LFSR only provides one such noise source. It is not possible to use the different taps of a single LFSR as separate noise sources because these taps are merely the same noise pattern offset in time and thus highly correlated. One solution would be to use one LFSR with different feedback taps and/or initial states for every noise source required. For large networks with long training times, this approach becomes prohibitively expensive in terms of area and possibly power required to implement the scheme. Another approach [15] builds from a standard LFSR with the addition of an XOR network with inputs coming from the taps of an LFSR and with outputs providing the multiple noise sources. Other approaches involving cellular automata can also be found [16], [17].

Another simplified approach utilizes two counterpropagating LFSRs with an XOR network to combine outputs from different taps to obtain uncorrelated noise [18]. Since the two LFSRs are counterpropagating, the length of the output sequences obtained from the XOR network is equal to the product of the lengths of the two individual LFSR sequences. It is possible to obtain more channels and larger sequence lengths with the use of larger LFSRs. This was the scheme that was ultimately implemented in hardware.

B. Up/Down Counter Weight Storage Elements

The weights in the network are represented directly as the bits of an up/down digital counter. The output bits of the counter feed directly into the digital input word weight bits of the synapse circuit. Updating the weights becomes a simple matter of incrementing or decrementing the counter to the desired value.

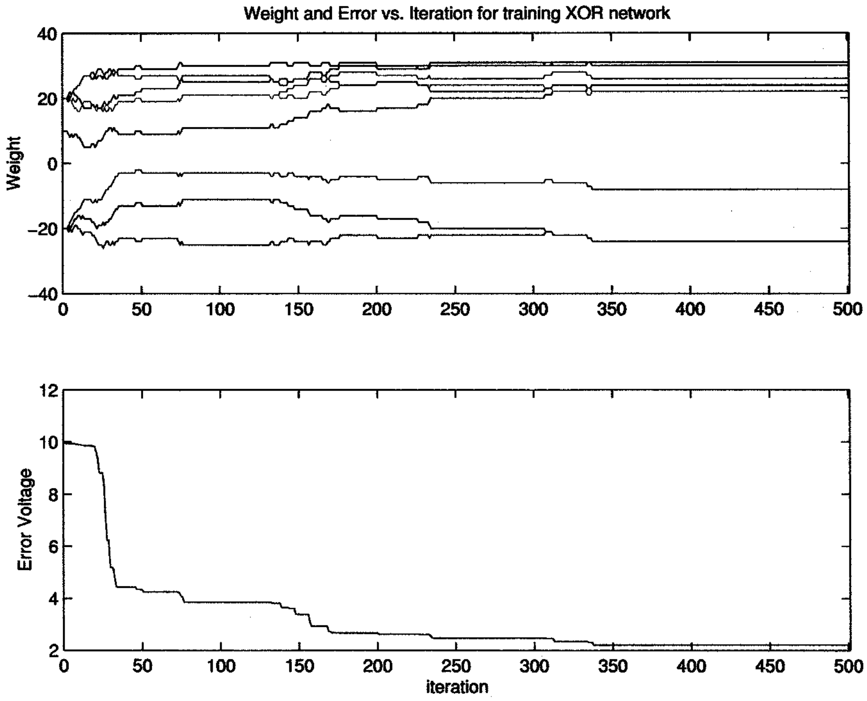

Fig. 11. Training of a 2 : 2 : 1 network with XOR function starting with “ideal” weights.

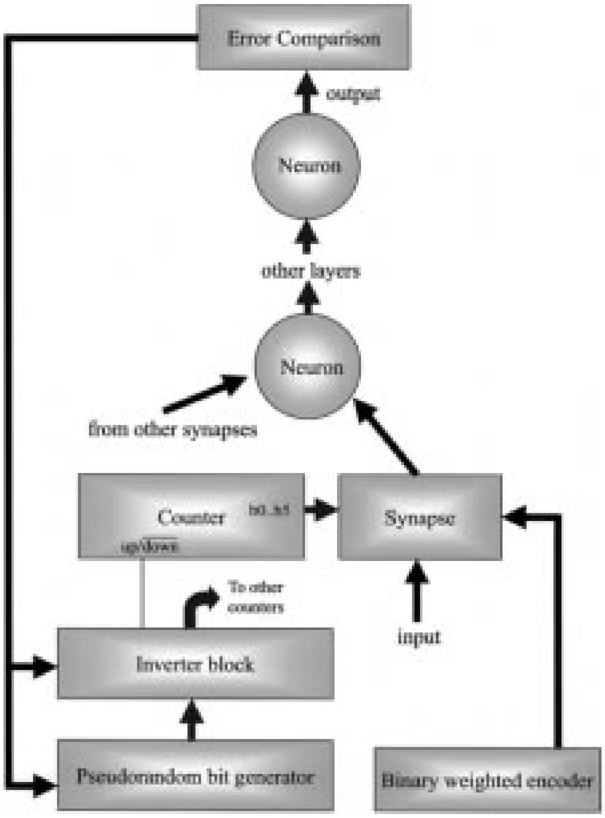

Fig. 12. Block diagram of parallel perturbative neural network circuit.

C. Parallel Perturbative Feed-forward Network

Fig. 12 shows a block diagram of the parallel perturbative neural network circuit operation. The synapse, binary weighted encoder and neuron circuits are the same as those used for the serial weight bus neural network. However, instead of the synapses interfacing with a serial weight bus, counters with their respective registers are used to store and update the weights.

```
Initialize Weights;
Get Error;
while(Error > Error Goal);
   Perturb Weights;
   Get New Error;
   if (New Error < Error),
      Error = New Error;
   else
      Unperturb Weights;
   end
end

function Perturb Weights;
   Set Invert Bit Low;
   Clock Pseudorandom bit generator;
   Clock counters;

function Unperturb Weights;
   Set Invert Bit High;
   Clock counters;
```

Fig. 13. Pseudocode for parallel perturbative learning network.

The counter up/down inputs originate in the pseudorandom bit generator and pass through an inverter block. The inverter block is essentially composed of pass gates and inverters. If the invert signal is low, the bit passes unchanged. If the invert bit is high, then the inversion of the pseudorandom bit gets passed. Control of the inverter block is what allows weight updates to either be kept or discarded.

D. Training Algorithm

The algorithm implemented by the network is a parallel perturbative method [1], [5]. The basic idea of the algorithm is to perform a modified gradient descent search of the error surface without calculating derivatives or using explicit knowledge of the functional form of the

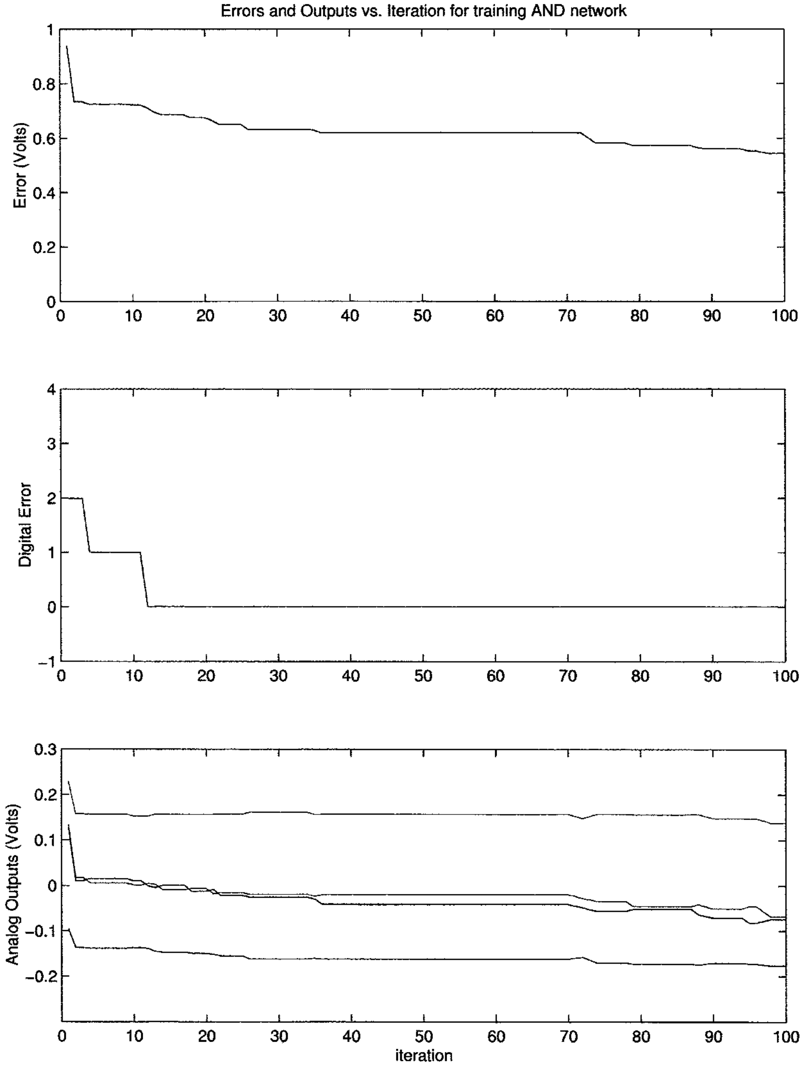

Fig. 14. Training of a 2 : 1 network with the AND function.

neurons. This is done by applying a set of small perturbations on the weights and measuring the effect on the network error. First, the network error, $E(\vec{w})$, is measured with the current weight vector, $\vec{w}$. Next, a perturbation vector, $\overrightarrow{pert}$, of fixed magnitude, but random sign is applied to the weight vector yielding a new weight vector, $\vec{w}+\overrightarrow{pert}$. Afterwards, the effect on the error, $\Delta E = E(\vec{w}+\overrightarrow{pert}) - E(\vec{w})$, is measured. If the error decreases then the perturbations are kept and the next iteration is performed. If the error increases, the original weight vector is restored. Thus, the weight update rule is of the following form:

$$\vec{w}_{t+1} = \begin{cases} \vec{w}_t + \overrightarrow{pert_t}, & \text{if } E\left(\vec{w}_t + \overrightarrow{pert_t}\right) < E\left(\vec{w}_t\right) \\ \vec{w}_t, & \text{if } E\left(\vec{w}_t + \overrightarrow{pert_t}\right) > E\left(\vec{w}_t\right) \end{cases}.$$

The use of this rule may require more iterations compared to some other perturbative weight update rules since it does not perform an actual weight change every iteration and since the weight updates are not scaled with the resulting changes in error. Nevertheless, it significantly simplifies the weight update circuitry. Some means must still be available to apply the weight perturbations; however, this rule does not require additional circuitry to change the weight values proportionately with the error difference, and, instead, relies on the same circuitry for the weight update as for applying the random perturbations. Some extra circuitry is required to remove the perturbations when the error does not decrease, but this merely involves inverting the signs of all of the perturbations and reapplying in order to cancel out the initial perturbation. A pseudocode version of the algorithm is presented in Fig. 13.

E. Error Comparison

The error comparison section is responsible for calculating the error of the current iteration and interfacing with a control section to implement the algorithm. Both sections could be performed off-chip by using a computer for chip-in-loop training, as was chosen for the current implementation. This allows flexibility in performing the global functions necessary for implementing the training algorithm, while the local functions are performed on chip. However, the control section could be implemented on chip as a standard digital section such as a finite state machine. Also, there are several alternatives for implementing the error comparison on chip. First, the error comparison could simply consist of A/D converters which would then pass the digital information to the control block. Another approach would be to have the error comparison section compare the analog errors directly and then output digital control signals.

F. Test Results

A chip implementing the parallel perturbative neural network was fabricated in the same technology and with similar transistor sizing as

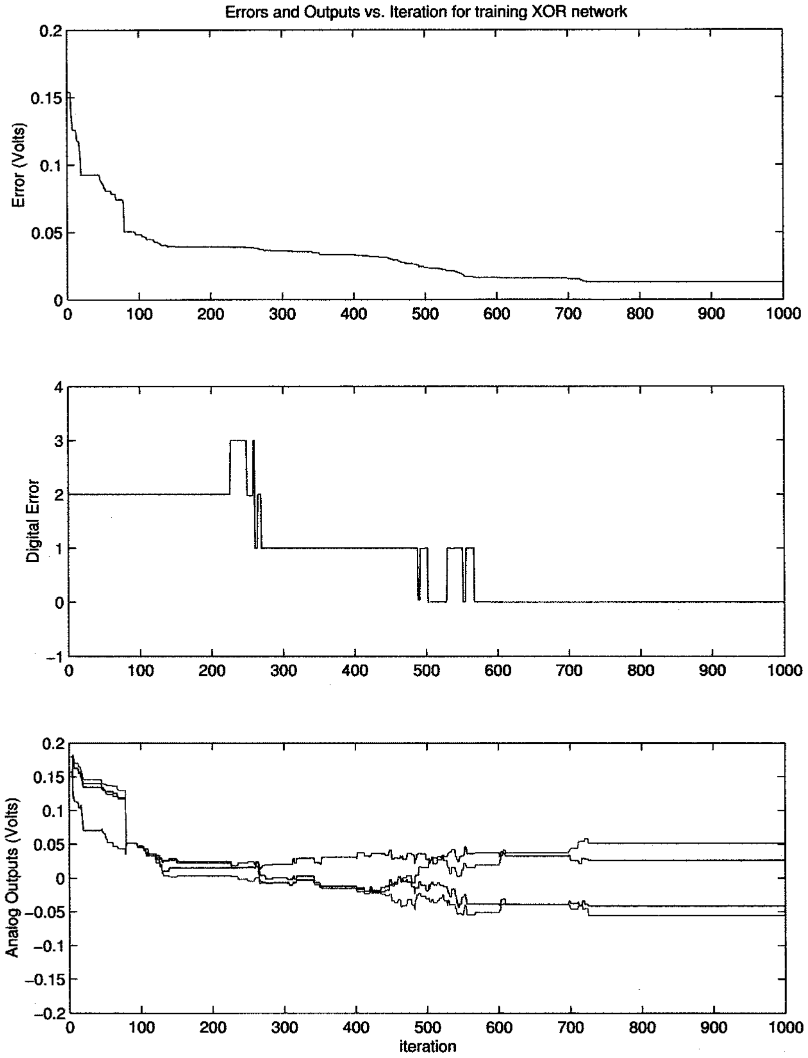

Fig. 15. Training of a 2 : 2 : 1 network with the XOR function.

for the previous test chip [19]. The results of some training runs are shown in Figs. 14 and 15. As can be seen from the figures, the network weights slowly converge to a correct solution. The error voltages were taken directly from the voltage output of the output neuron. The error voltages were calculated as a total sum voltage error over all input patterns. The actual output voltage error is arbitrary and depends on the circuit parameters. What is important is that the error goes down. Also shown is a digital error signal that shows the number of input patterns where the network gives the incorrect answer for the output. The network analog output voltage for each pattern is also displayed.

The actual weight values were not made accessible and, thus, are not shown; however, another implementation might also add a serial weight bus to read the weights and to set initial weight values. This would also be useful when pretraining with a computer to initialize the weights in a good location.

Fig. 14 shows the results from a two-input, one-output network learning an AND function. This network has only two synapses and one bias for a total of three weights. The network starts with two incorrect outputs which is to be expected with initial random weights. Since the AND function is a very simple function to learn, after relatively few iterations, all of the outputs are digitally correct. However, the network continues to train and moves the weight vector in order to better match the training outputs.

Fig. 15 shows the results of training a two-input, two-hidden unit, one-output network with the XOR function. Although the error voltage is monotonically decreasing, the digital error occasionally increases. This is because the network weights occasionally transition through a region of reduced analog output error that is used for training, but which actually increases the digital error. This seems to be occasionally necessary for the network to ultimately reach a reduced digital error. The function is essentially learned after only several hundred iterations.

Some of the analog output values occasionally show a large jump from one iteration to another. This occurs when a weight value which is at maximum magnitude overflows and resets to zero. The weights are merely stored in counters, and no special circuitry was added to deal with these overflow conditions. It would require a simple logic block to ensure that if a weight is at maximum magnitude and was incremented, that it would not overflow and reset to zero. However, this circuitry would need to be added to every weight counter and would be an unnecessary increase in size. These overflow conditions should normally not be a problem. Since the algorithm only accepts weight changes that decrease the error, if an overflow and reset on a weight is undesirable, the weight reset will simply be discarded. In fact, the weight reset may occasionally be useful for breaking out of local minima, where a weight value has been pushed up against an edge which leads to a local minima, but a sign flip or significant weight reduction is necessary to reach the global minimum. In this type of situation, the network will be unwilling to increase the error as necessary to obtain the global minimum.

IV. Conclusion

Several VLSI implementations of a neural network have been demonstrated. Digital weights are used to provide stable weight storage. Analog multipliers are used because full digital multipliers would occupy considerable space for large networks. Although the functions learned were digital, the networks are able to accept analog inputs and provide analog outputs for learning other functions. A parallel perturbation technique was used to train the networks successfully on the 2-input AND and XOR functions.

The size of the neuron cell in dimensionless units was $300\lambda \times 96\lambda$, the synapse was $80\lambda \times 150\lambda$, and the weight counter/register was $340\lambda \times 380\lambda$. In the 1.2-$\mu$m process used to make the test chips, λ was equal to 0.6 μm. In a modern process, such as a 0.35-μm process, it would be possible to make a network with over 100 neurons and over 10 000 weights in a 1 cm × 1 cm chip.

Thus, the ability to learn of a neural network which uses analog components for implementation of the synapses and neurons and with 6 b digital weights has been successfully demonstrated. The choice of 6 b for the digital weights was made in order to demonstrate that learning was possible with limited bit precision. The circuits can easily be extended to use 8 b weights. Using more than 8 b may not be desirable since the analog circuitry itself may not have significant precision to take advantage of the extra bits per weight. Significant strides can be taken to improve the matching characteristics of the analog circuits, but then the inherent benefits of using a compact, parallel, analog implementation may be lost.

References

[1] J. Alspector, R. Meir, B. Yuhas, A. Jayakumar, and D. Lippe, “A parallel gradient descent method for learning in analog VLSI neural networks,” in *Advances in Neural Information Processing Systems*. San Mateo, CA: Morgan Kaufman, 1993, vol. 5, pp. 836–844.

[2] R. Coggins, M. Jabri, B. Flower, and S. Pickard, “A hybrid analog and digital VLSI neural network for intracardiac morphology classification,” *IEEE J. Solid-State Circuits*, vol. 30, pp. 542–550, May 1995.

[3] M. Jabri and B. Flower, “Weight perturbation: An optimal architecture and learning technique for analog VLSI feedforward and recurrent multilayer networks,” *IEEE Trans. Neural Networks*, vol. 3, pp. 154–157, Feb. 1992.

[4] B. Flower and M. Jabri, “Summed weight neuron perturbation: An O(N) improvement over weight perturbation,” in *Advances in Neural Information Processing Systems*. San Mateo, CA: Morgan Kaufman, 1993, vol. 5, pp. 212–219.

[5] G. Cauwenberghs, “A fast stochastic error-descent algorithm for supervised learning and optimization,” in *Advances in Neural Information Processing Systems*. San Mateo, CA: Morgan Kaufman, 1993, vol. 5, pp. 244–251.

[6] P. W. Hollis and J. J. Paulos, “A neural network learning algorithm tailored for VLSI implementation,” *IEEE Trans. Neural Networks*, vol. 5, pp. 784–791, Oct. 1994.

[7] G. Cauwenberghs, “Analog VLSI stochastic perturbative learning architectures,” *Analog Integr. Circuits and Signal Process.*, vol. 13, pp. 195–209, 1997.

[8] C. Diorio, P. Hasler, B. A. Minch, and C. A. Mead, “A single-transistor silicon synapse,” *IEEE Trans. Electron Devices*, vol. 43, pp. 1972–1980, Nov. 1996.

[9] ——, “A complementary pair of four-terminal silicon synapses,” *Analog Integrated Circuits Signal Processing*, vol. 13, no. 1–2, pp. 153–166, May-June 1997.

[10] C. Mead, *Analog VLSI and Neural Systems*. Reading, PA: Addison-Wesley, 1989.

[11] V. F. Koosh and R. M. Goodman, “VLSI neural network with digital weights and analog multipliers,” in *Proc. IEEE Int. Symp. Circuits and Systems*, vol. III, May 2001, pp. 233–236.

[12] S. W. Golomb, *Shift Register Sequences*. Laguna Hills, CA: Aegean Park, 1982.

[13] P. Horowitz and W. Hill, *The Art of Electronics*, 2nd ed. Cambridge, U.K.: Cambridge Univ. Press, 1989, pp. 655–664.

[14] J. Alspector, B. Gupta, and R. B. Allen, “Performance of a stochastic learning microchip,” in *Advances in Neural Information Processing Systems 1*, D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufman, 1989, pp. 748–760.

[15] J. Alspector, J. W. Gannett, S. Haber, M. B. Parker, and R. Chu, “A VLSI-efficient technique for generating multiple uncorrelated noise source and its application to stochastic neural networks,” *IEEE Trans. Circuits Syst. II*, vol. 38, pp. 109–123, Jan. 1991.

[16] S. Wolfram, *Theory and Applications of Cellular Automata*. Singapore: World Scientific, 1986.

[17] A. Dupret, E. Belhaire, and P. Garda, “Scalable array of Gaussian white noise sources for analogue VLSI implementation,” *Electron. Lett.*, vol. 31, no. 17, pp. 1457–1458, Aug. 1995.

[18] G. Cauwenberghs, “An analog VLSI recurrent neural network learning a continuous-time trajectory,” *IEEE Trans. Neural Networks*, vol. 7, pp. 346–361, Apr. 1996.

[19] V. F. Koosh, “Analog computation and learning in VLSI,” Ph.D. dissertation, California Inst. Technol., Pasadena, 2001.