Neural Computation of Arithmetic Functions

KAI-YEUNG SIU AND JEHOSHUA BRUCK

The basic processing unit of a neural network is a linear threshold element. It has been known that neural networks can be much more powerful than traditional logic circuits, assuming that each threshold element can be built at a cost comparable to that of AND, OR, NOT logic elements. Whereas any logic circuit of polynomial size (in n) that computes the product of two n-bit numbers requires unbounded delay, such computations can be done in a neural network with "constant" delay. We improve some known results by showing that the product of two n-bit numbers and sorting of n n-bit numbers can be computed by a polynomial-size neural network using only 4 and 5 unit delays, respectively. Moreover, the weights of each threshold element in our neural networks require O(log n)-bit (instead of n-bit) accuracy.

I. Introduction

Neural networks can be viewed as circuits of highly interconnected parallel processing units called "neurons." The most commonly used models of neurons are linear threshold gates or, when continuity or differentiability is required, elements with a sigmoid input-output function. Because of recent advances in VLSI technology, the neural network has also emerged as a new technology and has found wide application in many areas.

Much of the current research in neural networks is in the area of pattern classification and is concerned with developing efficient "learning" algorithms for adjusting interconnection weights adaptively to perform the desired classification. Heuristics such as the "back propagation algorithm" have obtained surprisingly good empirical results [1]. In this paper, we shall look at another area of application of neural networks. Our model of a neuron is the linear threshold gate, and the network architecture considered here is the layered feedforward network. We shall see how common arithmetic functions such as multiplication and sorting can be efficiently computed in a "shallow" neural network. Whereas the interconnection weights are modified adaptively for different inputs in pattern classification and the desired classification is usually only approximated, in our network the weights are fixed for all inputs and the desired function is computed exactly. We shall confine our attention to operations on numbers represented in binary and we assume the inputs are encoded in $\{+1, -1\}$ instead of $\{1, 0\}$. Little would change in our analysis if we adopted the conventional $\{1, 0\}$ encoding since the transformation $\{+1, -1\} \rightarrow \{1, 0\}$ can easily be done by $x \rightarrow (x + 1)/2$.

The remainder of this paper is divided into seven major sections. In Section II, we review the classical model of a neuron, indicate the limitation of its capability and address the issues of sensitivity and dynamic range of parameters from the practical point of view. In Section III, we introduce a more practical model of a neuron in which we restrict the weights to be integers and the growth rate of the magnitudes of the weights to be at most polynomial in the size of the inputs. In Section IV, we consider a feedforward network of such neurons and indicate its unrestricted capability to compute any Boolean function. In Section V, we present some known lower-bound results on the classical implementation of arithmetic functions such as multiplication of two n-bit integers to indicate that unbounded delay is required using AND, OR, NOT logic elements. In Section VI, we show that our model of a feedforward neural network is very fast in computing arithmetic functions. In particular, sorting, sum of n n-bit numbers, and multiplication of two n-bit numbers can all be computed by a shallow neural network. The fact that these two functions can be computed in a "constant-depth" neural network was shown in [2] (see also [3]); however, their construction is not depth-efficient and it is not explicitly stated how many constant layers are needed in each step of their construction. We shall see how the constant can be reduced by a more depth-efficient construction and by using the results in [4]. It has been known [5], [6] that more complicated arithmetic functions such as exponentation and division can be computed in a constant-depth neural network. We shall only review the technique of reducing division to exponentiation and refer interested readers to [5]. In the conclusion, we indicate some possible extension of these results and other directions of research.

Manuscript received Nov. 14, 1989; revised March 14, 1990. This work was done while K.-Y. Siu was a research student associate at IBM Almaden Research Center and was supported in part by the Joint Services Program at Stanford University (US Army, US Navy, US Air Force) under Contract DAAL03-88-C-0011, and the Department of the Navy (NAVELEX) under Contract N00039-84-C-0211, NASA Headquarters, Center for Aeronautics and Space Information Sciences under Grant NAGW-419-S6.

K.-Y. Siu is with the Information Systems Laboratory, Stanford, CA 94305, USA.

J. Bruck is with the IBM Research Division, Almaden Research Center, San Jose, CA 95120-6099.

IEEE Log Number 9039184.

0018-9219/90/1000-1669$01.00 © 1990 IEEE

II. Classical Model of a Neuron

The classical model of a neuron [7] is a linear threshold device, which computes a linear combination of the inputs, compares the value with a threshold, and outputs +1 (or −1) if the value is larger (or smaller) than the threshold. More formally, we have

Input:

$$\vec{x} = (x_1, \cdots, x_n) \in R^n$$

Parameters:

weights $\vec{w} = (w_1, \cdots, w_n) \in R^n$

threshold $\theta \in R$

Output:

$$f(X) = \operatorname{sgn}\left\{\sum_{i=1}^{n} x_i \cdot w_i - \theta\right\}$$

where

$$\operatorname{sgn}\{y\} = \begin{cases} +1 & \text{if } y \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

In this paper, we consider only Boolean inputs $\vec{x} \in \{+1, -1\}^n$. It is easy to see that logic elements such as AND, OR, NOT can be simulated by a neuron:

$$\operatorname{sgn}(x_1 + \cdots + x_n - n)$$

$$= \text{AND}(x_1, \cdots, x_n) = \begin{cases} +1 & \text{iff all } x_i = +1 \\ -1 & \text{otherwise} \end{cases}$$

$$\operatorname{sgn}(x_1 + \cdots + x_n + n - 1)$$

$$= \text{OR}(x_1, \cdots, x_n) = \begin{cases} +1 & \text{iff some } x_i = +1 \\ -1 & \text{otherwise} \end{cases}$$

$$\operatorname{sgn}(-x) = \text{NOT}(x) = \begin{cases} +1 & \text{if } x = -1 \\ -1 & \text{if } x = +1 \end{cases}$$

A Boolean function that can be realized by a neuron is called a *linear threshold function*. However, the class of linear threshold functions only constitutes a vanishingly small subclass of the totality of Boolean functions. In fact, there are 2^{2^n} Boolean functions in n variables, but only $2^{O(n^2)}$ are linear threshold functions. On the other hand, since any Boolean function can be implemented by a network of AND, OR, NOT elements, it follows that a network of neurons can implement any Boolean function. Note that we have not yet made any restriction on the size of the network, i.e., the number of elements in the network. In general, an arbitrary Boolean function must require the size of the network to grow exponentially large with the number of input variables. Later on, we shall see that some functions that can be computed by a network with a polynomial number of AND, OR, NOT elements require an unbounded number of delays, whereas only a constant delay is needed if computed by a neural network.

In the definition of the classical model of a neuron, the weights can take on real values. Since we would like to implement a neuron using an analog device, from a practical point of view it is important to see if the assumption of real valued weights is necessary. In other words, can all linear threshold functions be realized if the weights are of finite precision? Actually, it was known [8] that each of the weights in a linear threshold function of n variables can be assumed to be integers of $O(n \log n)$ bits. However, this still allows the weights to grow exponentially fast with the number of input variables. In fact, most linear threshold functions have weights that must grow exponentially fast. This fact can also be interpreted as the necessity of high accuracy and high sensitivity of parameters in the actual implementation of a neuron. Motivated by this consideration, in the next section we consider a more practical model of a neuron, in which the weights are restricted to grow only polynomially fast.

III. More Practical Model of a Neuron

In the following, we consider a restricted class of neurons, which is more practical as a computational model. Each function $f(X) = \operatorname{sgn}(\sum_{i=1}^{n} w_i \cdot x_i + w_0)$ computed in this subclass is characterized by the property that the weights w_i are integers and bounded by a polynomial in the number of input variables, that is, $|w_i| \leq n^c$ for some constant $c > 0$. For convenience, we refer to this restricted model of a neuron as an $\widehat{LT}_1$ element.

Since the weights in an $\widehat{LT}_1$ element are assumed to be polynomially large integers, this means that we only require $O(\log n)$-bit accuracy in each weight. Thus in actual analog implementation, the device is much less sensitive to small fluctuations of parameters than the classical model. Note that the logic elements AND, OR, NOT are also $\widehat{LT}_1$ elements (see Section II). A natural question to ask is how limited in capability are $\widehat{LT}_1$ elements in comparison with the classical model? In [4], it was shown that any classical neuron can be simulated by three layers of a polynomial number of $\widehat{LT}_1$ elements. In other words, we can trade off exponentially large weights with a polynomial increase in size and a constant increase in delay by a factor of three. Hence any function that can be computed by a network of a polynomial number of classical neurons with constant delay can also be computed by a network of $\widehat{LT}_1$ elements with constant delay and polynomial increase in size. This leads naturally to the consideration of the computational capability of a feedforward neural network of $\widehat{LT}_1$ elements.

IV. Feedforward Neural Network

A feedforward network is a network of interconnected functional elements $\in G$: $\{+1, -1\}^n \to \{+1, -1\}$ with no feedback. More formally, we define a feedforward network to be an acyclic labeled directed graph, with

- a list of n_{in} distinguished input nodes with indegree 0
- internal nodes with arbitrary indegree which compute functional gates $\in G$ of the outputs from precedent nodes
- a list of n_{out} distinguished output nodes.

The *depth* of a node v is defined to be the length of the longest path (each edge is a unit length) from the input nodes to v. The depth of the network is defined to be the maximum depth of all output nodes. If we group all gates with the same depth together, we can consider the network to be arranged in layers, where the depth of the network is equal

to the number of layers (excluding the input layer) in the network, and gates of the same layer are computed in parallel. Given an assignment of the input nodes from domain $\{+1, -1\}^{n_{in}}$, the value of the network at each output node is obtained by evaluation of the gates in increasing depth order. The network therefore defines a mapping from $\{+1, -1\}^{n_{in}}$ to $\{+1, -1\}^{n_{out}}$, and the depth of the network can be interpreted as the time for its parallel execution of the mapping.

We define a *neural network* to be a feedforward network of $\widehat{LT}_1$ elements. Similarly, a logic circuit is a feedforward network of AND, OR, NOT logic gates. Obviously, any Boolean function can be computed by a logic circuit (without any restriction on its size) and thus by a neural network, since AND, OR, NOT are also $\widehat{LT}_1$ elements.

Loosely speaking, a network is shallow if it has small depth. Before we show how a shallow neural network can compute arithmetic functions, we first review the classical implementation using AND, OR, NOT logic elements and the limitation of constant depth circuits of such elements. This is the subject of next section.

V. Classical Implementation of Arithmetic Functions

It is well known among experienced circuit designers that they cannot implement some common functions such as parity and multiplication with small programmable logic arrays (PLAs), a type of integrated circuit used inside microprocessors to compactly represent many functions. Thus it is of both practical and theoretical interest to see how large the size of logic circuits must be to compute such common functions. It turns out that PLAs are well modeled by bounded-depth circuits of AND, OR, NOT logic elements with arbitrary fan-in. In 1961, Lupanov [9] studied bounded-depth circuits and showed that parity circuits of depth 2 must have an exponential number of gates. A breakthrough in theoretical research occurred in 1981 [10]; Furst *et al.* showed that any bounded-depth logic circuits must use more than a polynomial number of gates with arbitrary fan-in. This lower-bound result was further improved by several researchers [11], [12], who showed that an exponential number of gates is necessary to implement the parity function in a bounded-depth logic circuit. All these results can be interpreted as proofs that any PLA implementing parity must have an exponential amount of chip area, and thus establishing a basis for the common belief among circuit designers.

Another way of interpreting these results is that any parity circuit which uses a polynomial amount of chip area must have unbounded delay. By introducing the notion of constant-depth reduction [2], similar results can be shown for other common functions such as multiplication and division. In fact, currently used multipliers require $O(\log n)$ delays for input number of n-bits. The lower-bound results also imply that the minimum possible delay for multipliers of polynomial size is $\Omega(\log n/\log \log n)$.

We can explain the preceding negative results by the fact that the basic processing logic elements AND, OR, NOT of the circuits are not powerful enough. In practical implementation, these logic elements are built using analog devices; perhaps we can build a more powerful gate out of analog devices to increase the computational power of the circuit? In Section III, because of some issues of implementation, we introduced a new model of a neuron, called an $\widehat{LT}_1$ element, as the basic building block in our neural network. In fact, the main theme of this paper is to see how a shallow neural network of polynomial size can compute common functions such as multiplication and sorting with small constant delay.

VI. Computing with Shallow Neural Networks

In this section we focus on the computational capability of the feedforward neural network model introduced in Section IV. We assume that each neuron takes a unit delay to compute and we consider only neural networks of polynomial size. We shall see how the product of two n-bit numbers and sorting of n n-bit numbers can be computed with only 4 and 5 unit delays, respectively. Since our construction of the "neural multiplier" generalizes a known technique of computing symmetric functions with a neural network, we first show how any symmetric function can be computed in two layers of neural networks [13], [14].

A. Computing a Symmetric Function

Definition: A Boolean function f is said to be *symmetric* if

$$f(x_1, \cdots, x_n) = f(x_{(1)}, \cdots, x_{(n)})$$

for any permutation $(x_{(1)}, \cdots, x_{(n)})$ of $(x_1, \cdots, x_n)$, or equivalently, there exists a set of numbers $\{k_1, \cdots, k_l\}$, $|k_i| \leq n$ such that

$$f(x_1, \cdots, x_n) = 1 \quad \text{iff } \sum_{i=1}^{n} x_i \in \{k_1, \cdots, k_l\}.$$

In other words, a symmetric function depends only on the sum of input values. Using the same notation, let

$$y_{k_j} = \operatorname{sgn}\left\{\sum_{i=1}^{n} x_i - k_j\right\};$$

$$\tilde{y}_{k_j} = \operatorname{sgn}\left\{k_j - \sum_{i=1}^{n} x_i\right\} \quad \text{for } j = 1, \cdots, l.$$

The first layer of our network consists of neurons which compute the values y_{k_j} and $\tilde{y}_{k_j}$. In the second layer, the output neuron takes as inputs y_{k_j}, $\tilde{y}_{k_j}$ and outputs $\operatorname{sgn}\{\Sigma_{j=1}^{l}(y_{k_j} + \tilde{y}_{k_j}) - 1\}$. If $\Sigma_{i=1}^{n} x_i \notin \{k_1, \cdots, k_l\}$, then $y_{k_j} = -\tilde{y}_{k_j}$ for all $j = 1, \cdots, l$. Thus, $\Sigma_{j=1}^{l}(y_{k_j} + \tilde{y}_{k_j}) = 0$ and output $= -1$. On the other hand, if $\Sigma_{i=1}^{n} x_i = k_j$ for some $k_j \in \{k_1, \cdots, k_l\}$, then $y_{k_j} = \tilde{y}_{k_j} = 1$ and $y_{k_i} = \tilde{y}_{k_i}$ for $i \neq j$. Thus, output $= \operatorname{sgn}\{\Sigma_{j=1}^{l}(y_{k_j} + \tilde{y}_{k_j}) - 1\} = \operatorname{sgn}\{2 - 1\} = 1$. Hence our network correctly computes the desired symmetric function. Since the parity function is symmetric, it follows from the above results that parity can be computed in two layers of neural network, whereas it takes unbounded delay to compute parity in a logic circuit. Figure 1 illustrates a two-layer network for computing the parity function of three variables. Note that

$$\text{Parity } (x_1, x_2, x_3) = 1 \quad \text{iff number of } x_i\text{'s} = 1 \text{ is odd}$$

$$\text{iff } x_1 + x_2 + x_3 = -1 \text{ or } 3.$$

On closer observation, it is evident that the above construction also holds for any Boolean function $f(x_1, \cdots, x_n)$ whose value only depends on a weighted sum of the variables $\Sigma_{i=1}^{n} w_i \cdot x_i$, where the weights w_i are integers and poly-

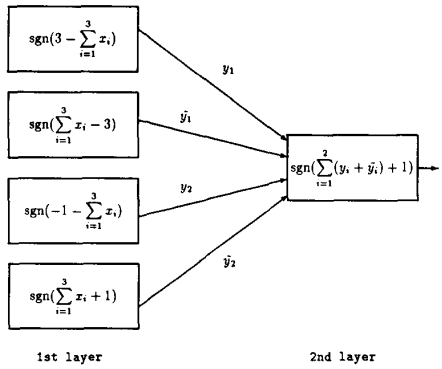


Fig. 1. Two-layer neural network for computing the Parity (x_1, x_2, x_3) function.

nomially bounded. Thus any such function can be computed in two layers of neural networks. Also notice that since we only use the output neuron to compute the linear combination of the outputs from the first layer, which only takes on value $+1$ or -1, it is redundant to compute the sgn $(\cdots)$ after computing the linear combination. We shall make use of these two observations in the next section.

B. Addition and Multiplication

Using the carry-look-ahead method, it was known that the sum of two n-bit numbers can be computed in a bounded-depth logic circuits of polynomial size with arbitrary fan-in. In fact, it was shown in [4] that a two-layer neural network suffices. This result is based on ideas from harmonic analysis of Boolean functions and we refer interested readers to [4], [15] for more details. Since the least significant bit of the sum is the EXCLUSIVE-OR function of the least significant bits of the two numbers, which is not a linear threshold function, it follows that the sum cannot be computed using only one layer. Thus a two-layer neural network is depth-optimal.

Whereas the sum of two n-bit numbers can be computed in bounded-depth logic circuits, the results in [10] imply that the sum of n n-bit numbers cannot be computed with bounded delay. However, such computations can be done with bounded delay in a neural network. In the following, we first show how to compute the sum of n log n-bit numbers in two layers by generalizing the techniques of computing symmetric functions. Based on this technique, we then show how to reduce the sum of n n-bit numbers to that of two $O(n)$-bit numbers using two layers. The results in [4] imply that two more layers suffice to compute the final sum. Afterwards we shall see how to combine the second and the third layers. Finally, we show how the product of two n-bit numbers can be reduced to the sum of n $2n$-bit numbers using one more layer, so that altogether only four layers are needed to compute the product.

1) Computing the Sum of n *log* n*-bit Numbers with Two Layers:* Given n log n-bit numbers, say in binary representation, $z_i = z_{i\log n} \cdots z_{i1}$ for $i = 1, \cdots n$, we would like to compute the binary representation of their sum

$$s = \sum_{i=1}^{n} z_i = \sum_{j=1}^{\log n} 2^{j-1}(z_{1_j} + z_{2_j} + \cdots + z_{n_j}).$$

Clearly, s is a polynomially bounded weighted sum of the variables z_{i_j} for $i = 1, \cdots, n$ and $j = 1, \cdots, \log n$. Thus, each bit of the binary representation of the sum s can be regarded as a Boolean function that depends only on a polynomially bounded weighted sum of $n \times \log n$ input variables. From the first remark given at the end of Section VI-A, any such function can be computed using 2 layers.

2) Reduction of the Sum of Two O(n)-bit Numbers: Suppose we are given n n-bit binary numbers: $x_i = x_{i_{n-1}} x_{i_{n-2}} \cdots x_{i_0}$, $i = 1, \cdots, n$ and we want to compute their sum. We shall see how to reduce this multiple sum to the sum of two numbers. Without loss of generality, we assume that $N = n/\log n$ and $\log n$ are integers, where log denotes logarithm to the base 2. Consider the following scheme: Partition each binary number x_i into N consecutive blocks $\tilde{x}_{i_0}, \tilde{x}_{i_1}, \cdots, \tilde{x}_{i_{N-1}}$ of $\log n$ bits each so that

$$x_i = \sum_{j=0}^{N-1} \tilde{x}_{i_j} \cdot 2^{\log n \cdot j}$$

where $0 \le \tilde{x}_{i_j} < 2^{\log n}$. Note that in binary representation, $\tilde{x}_{i_0} = x_{i_{\log n - 1}} x_{i_{\log n - 2}} \cdots x_{i_0}$ and $\tilde{x}_{i_{N-1}} = x_{i_{n-1}} x_{i_{n-2}} \cdots x_{i_{n-\log n}}$. We say a block $\tilde{x}_{i_j}$ is "odd" or "even" if j is odd or even, respectively.

Let s_{odd} denote the sum of the n numbers when the even blocks are set to zero and s_{even} denote the sum when the odd blocks are set to zero. The sum of the original n numbers will be the sum of s_{odd} and s_{even}. We now show how to compute s_{odd} and s_{even} in parallel using two layers.

Observe that for each $j = 0, \cdots, N-1$, the sum

$$\tilde{s}_j = \sum_{i=0}^{n-1} \tilde{x}_{i_j} < \sum_{i=0}^{n-1} 2^{\log n} = 2^{2\log n}$$

and thus $\tilde{s}_j$ can be represented in 2 log n bits. Observe that each $\tilde{s}_j$ is the sum of n log n-bit numbers. It follows from the previous section that the binary representation of each $\tilde{s}_j$ can be computed with two layers. Now

$$s_{\text{odd}} = \sum_{j\text{odd}} \tilde{s}_j \cdot 2^{2\log n \cdot j}; \quad s_{\text{even}} = \sum_{j\text{even}} \tilde{s}_j \cdot 2^{\log n \cdot j}$$

Since $\tilde{s}_j$ can be represented in 2 log n bits, there is no overlapping in the binary representation between

$$\tilde{s}_j \cdot 2^{\log n \cdot j} \quad \text{and} \quad \tilde{s}_{j+2} \cdot 2^{\log n \cdot (j+2)} = 2^{2\log n}(\tilde{s}_{j+2} \cdot 2^{\log n \cdot j})$$

Therefore, we can sum each odd block $\tilde{s}_j$ in parallel with two layers and concatenate the resulting bits of each sum together to obtain s_{odd}. We can obtain s_{even} in a similar fashion in parallel. To sum the two $O(n)$-bit numbers s_{odd} and s_{even}, another two layers suffice [4].

3) Combining the Second and Third Layers: Recall the second remark given at the end of Section VI-A. Since each output of the second layer is equal to a linear combination of the outputs from the first layer, which only takes on values $+1$ or -1, the sgn $(\cdots)$ in the second layer is not needed. Therefore we can directly feed the outputs from the first layer and take the linear combination as inputs to the third layer. As a result, the first three layers can be combined into two layers. So altogether only three layers are needed to compute the sum of n n-bit numbers.

A small numerical example will be helpful to illustrate the ideas. We take $n = 16$ and for simplicity, we only compute the sum of four 16-bit numbers. In Fig. 2, each of the four binary numbers x_1, x_2, x_3, x_4 are partitioned into four blocks $\tilde{x}_{i_0}, \tilde{x}_{i_1}, \tilde{x}_{i_2}, \tilde{x}_{i_3}$ of log $n = 4$ bits each, for $i = 1, \cdots, 4$. The

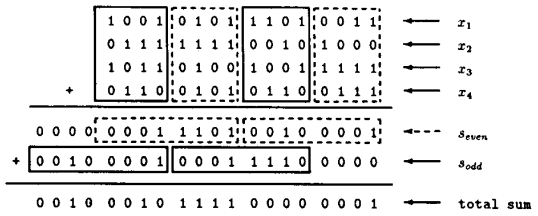


Fig. 2. Computing a multiple sum.

even blocks $\bar{x}_{i_0}, \bar{x}_{i_2}$ are denoted by a dotted rectangle and the odd blocks $\bar{x}_{i_1}, \bar{x}_{i_3}$ are denoted by a solid rectangle.

4) Reducing the Product to a Multiple Sum: Computing the product of 2 n-bit binary numbers $x = x_{n-1}x_{n-2} \cdots x_0$, $y = y_{n-1}y_{n-2} \cdots y_0$ is equivalent to computing the sum of n $2n$-bit binary numbers:

$$z_i = z_{i_{2n-1}}x_{i_{2n-2}} \cdots z_{i_0}, \qquad i = 0, \cdots, n-1,$$

where

$$z_k = \begin{cases} 0 & \text{if } (i+n \le k \le 2n-1) \text{ or } (0 \le k < i) \\ x_{k-i} \wedge y_i & \text{if } i \le k < i+n \end{cases}$$

where $\wedge$ denotes the logic AND function. In other words,

$$z_i = \underbrace{0 \cdots 0}_{n-i}\,(x_{n-1} \wedge y_i)\,(x_{n-2} \wedge y_i) \cdots (x_0 \wedge y_i)\,\underbrace{0 \cdots 0}_{i}$$

Given the two n-bit input binary numbers $x = x_{n-1}x_{n-2} \cdots x_0$, $y = y_{n-1}y_{n-2} \cdots y_0$, the first layer of our multiplier network outputs the n $2n$-bit binary numbers $z_i = z_{i_{2n-2}}x_{i_{2n-2}} \cdots z_{i_0}$ and then computes the sum in three more layers. Thus the product of two n-bit numbers can be computed in four layers.

C. Sorting

Here we shall see how sorting of n n-bit numbers can be computed in a neural network with depth 5. The techniques are mainly based on the results in [2]. We assume that the input is a list of the n n-bit binary numbers and the output will be the same list sorted in nondecreasing order. A number which appears m times in the input list will be duplicated m times in the output list.

In sorting, the basic operation is the comparison of two numbers, i.e., given two n-bit binary numbers $x = x_n x_{n-1} \cdots x_1$, $y = y_n y_{n-1} \cdots y_1$, we want to compute whether $x \ge y$. It is tempting to conclude that comparison can be computed in a single layer since

$$x \ge y \quad \text{iff} \quad \operatorname{sgn}\left\{\sum_{i=1}^{n} 2^i \cdot (x_i - y_i)\right\} = +1$$

However, notice that the weights chosen above are exponential in n and thus do not satisfy the conditions in our definition of a $\widehat{LT}_1$ element. In fact, it was shown in [4] that the comparison function cannot be computed using a single $\widehat{LT}_1$ element, but it can be computed in two layers of $\widehat{LT}_1$ elements.

Let $z_i = z_{i_n}z_{i_{n-1}} \cdots z_{i_1}$, for $i = 1, \cdots, n$, denote the input binary numbers. Define

$$c_{ij} = \begin{cases} +1 & \text{if } z_i > z_j \text{ or } (z_i = z_j \text{ and } i \ge j) \\ -1 & \text{otherwise} \end{cases}$$

Note that for each i, $p_i = \Sigma_{j=1}^{n}(1 + c_{ij})/2$ is the position of z_i in the sorted list. If we let

$$EQ_m(p_i) = (\operatorname{sgn}\{p_i - m\} + \operatorname{sgn}\{m - p_i\}) - 1$$

$$= \begin{cases} +1 & \text{if } p_i = m \\ -1 & \text{otherwise} \end{cases}$$

then the kth bit of the mth number in the sorted list is

$$\bigvee_{1 \le i \le n} (EQ_m(p_i) \wedge z_{ik})$$

where $\vee$ and $\wedge$ respectively denote the OR and AND functions.

In our neural network, the comparison functions c_{ij}s are computed in the first two layers. The next two layers are used to compute

$$(EQ_m(p_i) \wedge z_{ik}) = \operatorname{sgn}\{z_{ik} + (\operatorname{sgn}\{p_i - m\} + \operatorname{sgn}\{m - p_i\}) - 3\}$$

and the last layer is used to compute the OR of the outputs $(EQ_m(p_i) \wedge z_{ik})$, $i = 1, \cdots, n$.

D. Extensions to Other Functions

It is natural to continue our study of neural networks on computation of more complicated arithmetic functions such as exponentation, division, and extraction of square roots. In fact, it can be shown that multiplication of n n-bit numbers and division of 2 n-bit numbers can also be computed by a constant-depth neural network. In [5], it was shown how multiplication of n n-bit numbers can be computed in logic circuits of $O(\log n)$ depth, using the Chinese Remainder theorem. The basic idea of the construction is to hardwire in a polynomial size table of discrete logarithms for some prime powers and then reduce the problem to one of iterated addition. On closer observation, it is not hard to see how this construction of $O(\log n)$ depth logic circuits can be adapted to a construction of a constant-depth neural network. A presentation of such results would take us too far from the scope of this paper because of the necessary number-theoretic background. Moreover, the constant obtained by direct application of the algorithm in [5] will be too large for the resulting neural network to be considered shallow, and therefore the difference between log n and the constant in the delay is not significant unless the input numbers are astronomically large. These results are of theoretical importance, however, because they describe the fundamental difference in computation between neural networks and logic circuits. At this time, we are not able to reduce the constant to obtain a shallow neural network for division and multiple product. Below, we shall only indicate how division can be computed in constant-depth neural network, provided that exponentiation can be computed in constant delay. (See [5] for more details.)

Suppose we are given two n-bit binary numbers x, y, and we wish to compute the n-bit representation of $\lfloor x/y \rfloor$, i.e., the greatest integer $\le x/y$. We shall assume $2 \le y < x$. Since x/y is equal to the product of x and y^{-1}, it is enough to get a finite underapproximation $\tilde{y}^{-1}$ of y^{-1} with error $< 2^{-n}$. Then in a constant-depth neural network, we can compute

$q = x \cdot \tilde{y}^{-1}$ with error <1 and determine which one of the $\lfloor q \rfloor$ or $\lfloor q \rfloor + 1$ is $\lfloor x/y \rfloor$.

Let $j \geq 2$ be an integer such that $2^{j-1} \leq y < 2^j$. Note that $|1 - y2^{-j}| \leq \frac{1}{2}$ and we can express y^{-1} as a series expansion

$$y^{-1} = 2^{-j} \cdot (1 - (1 - y2^{-j}))^{-1}$$

$$= 2^{-j} \sum_{i=0}^{\infty} (1 - y2^{-j})^i$$

If we put

$$\tilde{y}^{-1} = 2^{-j} \sum_{i=0}^{n-1} (1 - y2^{-j})^i$$

then the difference between $\tilde{y}^{-1}$ and y^{-1} is less than 2^{-n}. Since the exponentiation $(1 - y2^{-j})^i$ is a special case of multiple products, we can compute them in parallel with a constant-depth network from the previous remarks and compute the multiple sum as shown in Section V-B.

Since evaluation of the numerator and denominator of a rational function involves computing a sum of multiple products in constant depth, applying the numerator and denominator to the division network gives the value of the rational function. As a result, any rational function can be computed in constant delay by neural networks. In general, we can conclude that an analytic function which is well approximated by a truncated power series can be evaluated by a constant-depth neural network.

VII. Conclusion

We have introduced a restricted model of a neuron, which is more practical as a model of computation than the classical model. We define our model as a feedforward network of such neurons. We have shown how common arithmetic functions such as multiple addition, multiplication, and sorting can be computed by a polynomial-size shallow neural network with 3, 4, and 5 unit delays, respectively, whereas it was known that these functions cannot be computed in constant-depth logic circuits. Applying the results in [5], we also indicated how these results can be extended to more complicated functions such as multiple products, division, rational functions, and approximation of analytic functions.

A natural continuation of our study is to consider even more complicated functions such as indicator functions for graph connectivity, bipartite matching, and network flow, all of which have well-known polynomial time algorithms. Another direction of research is to obtain lower-bound results in order to obtain a depth-optimal neural network. In fact, it was shown in [13] that computing the product of two n-bit numbers requires more than two layers, whereas our depth-4 multiplier network provides an upper bound of four layers. However, the well-known lower-bound techniques for unbounded fan-in logic circuits appear to break down completely in the case of neural networks, where the linear threshold elements are the basic processing units. Even though there are many candidate functions which appear not to be computable by a polynomial-size depth-3 neural network, at present no such function is explicitly proved to exist. It is of theoretical interest to note here that any function computable in a polynomial size constant-depth logic circuit with unbounded fan-in is also computable in a depth-3 neural network of superpolynomial—that is, $n^{O(\log n)}$ (instead of exponential)—size [16]. Therefore, proving that our multiplication and sorting networks are depth-optimal will be a difficult task and new lower-bound techniques in circuit complexity for networks of linear threshold elements have to be developed.

The moral of our study is that circuits based on threshold elements could be extremely powerful. Of course, these hopes are based on the assumption that a linear threshold ($\widehat{LT}_1$) element can be implemented using analog devices whose unit cost is small. This would justify research in device technology to investigate the feasibility of building such elements with small cost.

Acknowledgment

The first author would like to thank Prof. Thomas Kailath for his guidance, constant encouragement, and financial support.

References

[1] J. L. McClelland, D. E. Rumelhardt, and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1.* MIT Press, 1986.

[2] A. K. Chandra, L. Stockmeyer, and U. Vishkin, "Constant depth reducibility," *Siam J. Comput.*, vol. 13, pp. 423–439, 1984.

[3] N. Pippenger, "The complexity of computations by networks," *IBM J. Res. Develop.*, vol. 31, no. 2, pp. 235–243, Mar. 1987.

[4] K. Y. Siu and J. Bruck, *On the Dynamic Range of Linear Threshold Elements*, Tech. Rep. RJ 7237, IBM Research, Jan. 1990, to be submitted to *SIAM J. Discrete Math.*

[5] P. W. Beame, S. A. Cook, and H. J. Hoover, "Log depth circuits for division and related problems," *Siam J. Comput.*, vol. 15, 994–1003, 1986.

[6] J. Reif, "On threshold circuits and polynomial computation," *Proc. 2nd Ann. Structure in Complexity Theory Symp.*, pp. 118–123, 1987.

[7] M. Minsky and S. Papert, *Perceptrons*. MIT Press, expanded edition, 1988.

[8] P. Raghavan, *Learning in Threshold Networks: A Computation Model and Applications*, Tech. Rep. RC 13859, IBM Research, July 1988.

[9] O. Lupanov, "Implementing the algebra of logic functions in terms of constant-depth formulas in the basis $+, *, -$," *Sov. Phys. Dokl.*, vol. 6, no. 2, 1961.

[10] M. Furst, J. B. Saxe, and M. Sipser, "Parity, circuits and the polynomial-time hierarchy," *Proc. IEEE Symp. Found. Comp. Sci.*, vol. 22, pp. 260–270, 1981.

[11] J. Hastad, "Almost optimal lower bounds for small depth circuits," *Proc. ACM Symp. Theor. Computing*, vol. 18, pp. 6–20, 1986.

[12] R. Smolensky, "Algebraic methods in the theory of lower bounds for Boolean circuit complexity," *Proc. ACM Symp. Theor. Computing*, vol. 19, pp. 77–82, 1987.

[13] A. Hajnal, W. Maass, P. Pudlak, M. Szegedy, and G. Turan, "Threshold circuits of bounded depth," *IEEE Symp. Found. Comp. Sci.*, vol. 28, pp. 99–110, 1987.

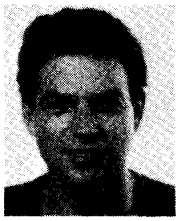
[14] J. Bruck, "Harmonic analysis of polynomial threshold function," *SIAM J. Discrete Math.*, vol. 3, no. 2, pp. 168–177, May 1990.

[15] J. Bruck and R. Smolensky, *Polynomial Threshold Functions, AC^0 Functions and Spectral Norms*, Tech. Rep. RJ 7140, IBM Research, Nov. 1989; to appear *IEEE Symp. Found. Comp. Sci.*, 1990.

[16] E. Allender, "A note on the power of threshold circuits," to in *IEEE Symp. Found. Comp. Sci.*, vol. 30, 1989.

Kai-Yeung Siu was born in Hong Kong on October 9, 1966. He received the B.Sc. degree in mathematics and computer science from New York University, NY, and the B.Eng. degree in electrical engineering from The Cooper Union, NY, both in 1987. In June 1988, he received the M.Sc. degree in electrical engineering from Stanford University, CA.

He is currently associated with the Information Systems Laboratory at Stanford University and pursuing his Ph.D. degree under the guidance of Prof. Thomas Kailath. He is also a research student associate with the Computer Science Department at IBM Almaden Research Center, San Jose, CA. His research interests include computational complexity theory, neural networks and parallel computation.



Jehoshua Bruck was born in Haifa, Israel, on April 19, 1956. He received the B.Sc. and M.Sc. degrees in electrical engineering from the Technion, Israel Institute of Technology, in 1982 and 1985, respectively, and the Ph.D. degree in electrical engineering from Stanford University in 1989.

From 1982 to 1985 he was with the IBM Haifa Scientific Center, Israel. In March, 1989, he joined the IBM Research Division at the Almaden Research Center, San Jose, CA, where he is presently a Research Staff Member.

Dr. Bruck's research interests include error-correcting codes, fault-tolerant computing, parallel computing, and neural networks.