Some Results concerning Adaptive Logic Networks *

William W. Armstrong, Andrew Dwelly,
Jiandong Liang, Dekang Lin, Scott Reynolds
Department of Computing Science
The University of Alberta
Edmonton, Alberta
Canada T6G 2H1
Tel: (403) 492 2374
FAX: (403) 492 1071
Email: arms@cs.ualberta.ca

August 22, 1991

1 Introduction

This paper presents some results on the learned synthesis of boolean functions using adaptive tree networks (ALNs) whose elements, after training and elimination of redundant elements, perform logical operations AND and OR. This is logically equivalent to a tree of NAND gates. To compute an output, a tree is applied to a boolean input vector, and its vector of complements,

*This research has been partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under operating grant OGP-5714. Myrias computing technologies, Inc., of Edmonton and the Alberta Research Council provided access to SPS-2 supercomputers. Contributions of Ashvanee Bissonauth, Joe Culberson, Scott Fahlman and John Shillington are gratefully acknowledged.

according to a connection scheme which may itself be the result of learning, i. e. automatic tree construction.

In hardware form, a trained network can compute its output very rapidly in comparison to artificial neural systems (ANS) requiring digital arithmetic operations such as additions, multiplications and exponentiations (perhaps using table lookup). Complete evaluation takes time proportional to the depth of the tree, and is usually a matter of a few tens of nanoseconds in CMOS. Hardware can be synthesized using field programmable gate arrays provided a conversion program is written to translate from the format of the learning program to the desired hardware description language.

An ALN is a special case of the familiar multilayer perceptron (MLP) feedforward network. There are just a few changes, all of them simplifications. Briefly, the nodes (or elements) have (during training, at least) two input leads, the input signals x_1, x_2 are boolean (0 or 1), each connection weight is determined by a single bit of information, and the "squashing function" is a threshold. Specifically, if b_1 and b_2 are boolean, then the element outputs a boolean value which is 1 if and only if

$(b_1 + 1) * x_1 + (b_2 + 1) * x_2 \geq 2$.

The four combinations of b_1 and b_2 (00, 11, 10, 01) generate the four boolean functions of two variables: AND, OR, LEFT, and RIGHT, where LEFT$(x_1, x_2) = x_1$ and RIGHT$(x_1, x_2) = x_2$. For example $1 * x_1 + 1 * x_2 \geq 2$ if and only if both x_1 AND x_2 are 1. The four functions are precisely all of the non-constant, increasing boolean functions of two variables. A boolean function is increasing if changing any 0 to a 1 on any of its inputs never causes the output to change from 1 to 0. One can verify this for the above functions, but in contrast, the exclusive or operation XOR is not increasing since XOR(0,1)=1 but XOR(1,1)=0.

The atree release 2.0 software was not available at the time the present experiments were performed, so no direct comparisons between the results in this document and the results obtainable by release 2.0 can be made. Since the software used in the experiments was optimized for each experiment, the results here may be better than those obtainable by release 2.0.

For the kind of syntheses intended, but without loss of generality, the number of input variables n is often quite large, for example in the hundreds, and the boolean functions to which the method is applied have the property that input vectors in close proximity to each other in Hamming distance tend to have equal function values. Pattern-classification tasks such as recognition of the “A’s” among normalized handwritten characters scanned into a $w \times h$ grid could be approximated by functions having such a property. The functions are generally only partially defined, usually by means of a training set of samples of input vectors and the desired output bit. Early experiments [3] reported on pattern classification problems, in particular optical character recognition.

Generalization by tree networks from inputs presented during training to new inputs results from the insensitivity [7] of tree functions to changes of their input vectors, a property we discuss in section 2. Igor Aleksander [1] has done considerable work in the area of logical networks. The fan-in of Alexander’s networks is generally higher than two, and the elements are universal, i. e. they can realize any boolean function of the given fan-in. During training of our trees, which we discuss in section 3, the elements have fan-in two. Our reason for not using ternary or higher-fan-in trees of universal elements is that binary trees of universal elements have superior insensitivity properties [7]. Insensitivity is further enhanced by restricting the node functions to AND and OR, plus the functions LEFT and RIGHT, which serve to cut off one of the inputs which is not currently useful. Of course, once learning has taken place, groups of contiguous two-input ANDs (or ORs) can be coalesced into a higher fan-in gate without changing the overall function and with an increase in evaluation speed.

The heuristic responibility used in atree release 2.0 is superior to heuristics which were known in the 1970s [3,4]. Not only has it led to better learning, but it is implementable without using the flip-flops necessary for the most simple previous strategy, the latest error algorithm. In addition, a double adaptation to patterns for which a tree does not give the correct response helps to speed up adaptation when the tree has almost completely adapted.

The use of logic networks has one extremely significant benefit: “parsimonious” (or “lazy”) evaluation, discussed in section 4. This property of logic networks makes it often *unnecessary* to evaluate more than a small fraction of the elements in a network to compute an output. Parsimony uses the fact that if one input of an AND (or OR) gate is 0 (or, respectively, 1), then the output is already known, and the subtrees producing the other inputs don’t have to be evaluated; nor indeed, do any of their inputs have to be determined. We note that for the learning elements used in ANS trained by the backpropagation technique with a sigmoidal non-linearity [9, 11], the computation of the output of an element generally requires the values of *all* of the inputs with non-zero weights. Some work has been done using non-sigmoidal functions acting on the weighted sum of inputs [10], and in this case, some computation is unnecessary and can be saved. Meisel has called that opportunity to save computation *parsimony*, and we have followed his usage. Hardware parallelism can be paired with parsimonious evaluation to obtain high speed of execution (and hence also adaptation) without incurring the cost of totally parallel hardware. Parsimony alone can make the training and execution of large logic networks feasible even on microcomputers.

Examples of learning of randomly defined boolean functions and multiplexers are given in section 5. The predetermined connection pattern from tree inputs to the bits of the input vector and its complements may make it impossible to find a function which learns the training data, and in section 6, we discuss some encouraging results on changing the tree structure during training. Multiplexer functions of 521 variables are approximated, based on very sparse training data, but using algorithms that chnge the structure of the tree as well as the functions in the nodes. The whole area of tree growth has begun to be studied in the last three years. Tree growth has not been implemented in atree release 2.0.

When functions of real or integer inputs and/or outputs are required, simple random-walk encodings of quantized real or integer values into boolean vectors are used. Figure 1 shows a tree network with two real valued inputs and a single real output. The inputs are separately encoded into bit vectors, which are concatenated and complemented. After processing through several logic trees, the resulting bits are decoded to get the nearest code for a quantization level of the continuous output value. Many useful real-valued functions have differentiability properties which translate into a low probability, on average, that a quantized output changes when the inputs are slightly perturbed, so they are amenable to treatment by the method of insensitive boolean functions. Questions of continuous function approximation are treated in section 7. There are two further examples of functions which use random walk encoding: learning the multiplication table in section 8 and the well-known "two-spirals" problem in section 9. Section 10 reports our conclusions.

The atree release 2.0 software package is available from the first author (for non-commercial use) that contains everything needed to create, train, evaluate, and print out logic networks. It has been tested on various systems: Sun 3/xx, Sparcstation, Myrias SPS-2, IBM-compatible PC, and Atari. It

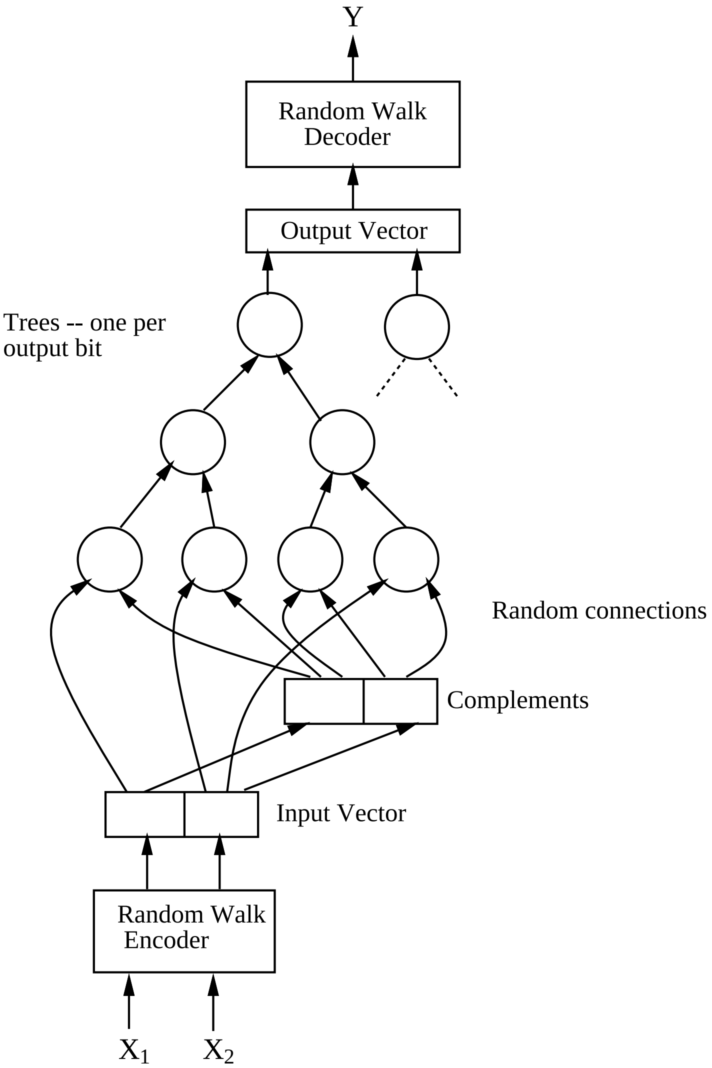

Figure 1: Using several trees to compute $Y = f(X_1, X_2)$

has been written for clarity and portability in the hope that it will stimulate research in the area of adaptive logic networks. It can be used perform some of the experiments reported in this paper and can be flexibly adapted to other problems by C programmers using the source code. A language "lf" allows non-programmers to conduct experiments on adaptive logic networks. Small demonstrations are included.

2 Insensitivity of tree functions

Consider a binary tree whose nodes represent two-input logic gates. For the moment, let the tree be balanced and have $2^L - 1$ non-leaf nodes arranged in L layers. The 2^L unconnected inputs at the leaf nodes are now connected randomly to the variables $x_1, \ldots, x_n$ and their complements. Now to each of the nodes of the tree is assigned, again randomly, one of the four non-constant monotonic boolean functions of two variables: AND, OR, LEFT, RIGHT. The ensemble of boolean tree functions so formed, each weighted according to the number of times it is generated by choices of connections and node functions, has the property that, on average, a small change of an input vector (as measured in the Hamming metric) entails a small probability of change of the output.

A plausibility argument for this in the case of a disjoint tree (i. e. there are no repeated connections to input variables) is in order. Suppose one input at a leaf of the tree is changed, then the probability over the above ensemble that the output of the node to which it is connected changes is 0.5. Consequently, the probability that the next higher node's output changes is 0.25, and so on, until at the root, the probability of change of the tree's output signal is 2^{-L}. The general case of a balanced non-disjoint tree has been reported [6].

In [7], a modulus of sensitivity function is defined. The modulus of sensitivity of a class of boolean functions of n variables is, for a given argument d, the probability of obtaining different values of a given function for two input vectors at Hamming distance d, averaged over all pairs of vectors at that distance and all functions of the class. We can define the modulus of sensitivity for a singleton class, i. e. for a single function. In [7] it is shown that for monotonic increasing binary tree functions (realizable using AND, OR, LEFT, and RIGHT), the modulus of sensitivity function converges uniformly to 0 for all distances d as the number of layers L approaches infinity. (This does not hold if the allowed set of node functions is enlarged and includes the exclusive-OR. Using ternary or higher fan-in trees is also detrimental to insensitivity.) In short, the class of binary tree functions using the four node functions named above can be made arbitrarily insensitive just by choosing a deep enough tree.

Now consider the following function synthesis problem: let f be an insensitive boolean function of n variables such as might arise in a pattern-classification problem. Let $\hat{f}$ be a partially defined function obtained from f by restriction of its domain (to a "training set", although we are not speaking of adaptation yet). Let g be a boolean tree function defined by an L-layer tree whose restriction to the training set is equal to $\hat{f}$. (This requires a sufficiently large depth L.) Then since g is drawn from an insensitive ensemble by its very construction, it will tend to be a good approximation to f even for input vectors at some distance from points of the training set. This provides one solution to a fundamental pattern recognition problem. One remaining obstacle is the time required to find the connections and node functions defining g. If we use a standard circuit-synthesis algorithm, we cannot be sure of the insensitivity of g. There may be hundreds of vectors in the training set, so random search is completely out of the question.

An adaptation procedure [3] that appears to deliver insensitive functions is described in section 3. That approach involves random connections of the leaves of a binary tree to input variables and complements, a random initial assignment of functions AND, OR, LEFT and RIGHT to the nodes, and a random sequence of training input vectors. One advantage of randomness is that the process can be repeated to generate several different tree functions, all correct on the training data. If generalization using one tree is poor, a majority vote over several tree outputs can improve it significantly.

3 Adaptation in boolean tree networks

The adaptive procedure used consists of presenting a random sequence of boolean input vectors from the training set to the randomly initialized network. For each vector, the tree function is evaluated, the network is informed of the desired output, and the internal states of some nodes are changed. The correctness of the network's response is noted by a control program to enable stopping adaptation when the function synthesized by the tree satisfies requirements.

A concept of "true responsibility" of a given node for the computation of the network output corresponding to the current input vector has been defined [5]. Briefly, a node is truly responsible if changing its output signal would also change the network output, all other signals remaining the same except the outputs on the path from that node to the root. The concept can be defined recursively starting at the root by examining the node functions and the inputs to nodes. For example, if an input to a truly responsible AND-node is 0, then the input on the *opposite* side makes no difference to the node's output. Hence the child node on the opposite side is *not* truly responsible. If the input is a 1, then the opposite child *is* truly responsible.

Early experiments used this concept as follows: if a node is truly responsible for the input vector at a given step of training, then the state of the node is enabled to change during that step. Since the node functions used all have the property that a (0, 0) input to the node gives a 0 output from the node, and a (1, 1) input gives a 1 output, the only flexibility the node has is to determine its outputs for the (0, 1) and the (1, 0) input pairs. In order to determine the node function, we take two bounded bidirectional counters in each node associated respectively with these input pairs. The (0, 1) counter will count up if the node is responsible and the (0, 1) input pair is received and the desired network output, as specified by the training data is 1. If the counter is already at its maximum, no change of counter value occurs. If the desired network output is 0, the count is down, unless the counter is at its minimum already. The response to the (0, 1) input is then defined to be 1 if the (0, 1) counter value is in the upper half of its range, and 0 otherwise. The (1, 0) counter works similarly. The justification for using the desired *network* responses to determine the response of a *node* is the monotonic increasing property of the node functions. If both counters are above their mid-value, the node's function is OR. If both are below, it is AND. Otherwise, the node is a LEFT or RIGHT.

These early experiments led to failure; the desired functions were not often learned. An intuitive argument against using the original true responsibility concept is that a child node's responsibility depends on the node function currently in the parent node. The responsibility of the child node determines the function learned by the subtree rooted at the child node. Yet it is only when the functions of the two subtrees of the parent node are determined that the node function of the parent can be effectively determined. The failures led to attempts to define a "heuristic responsibility" concept for a node, a type of responsibility that doesn't entirely depend on the function currently in the node. Such a heuristic responsibility is, in effect, a solution to the "credit assignment problem": i. e. to decide which components of a complex system should be given credit for the correct behavior of the system or be punished for its incorrect behavior.

Several answers were found; a very simple one goes as follows: The root node is always heuristically responsible. If a node is heuristically responsible and one of its input signals is not equal to the desired network output, that input signal is called an "error". If the signal on the left is an error, then the right child node is made heuristically responsible ("error responsibility"). Two bits in each node keep track of the latest error signal values on the left and right. The right subtree is also made heuristically responsible if the left input signal is not an error, but is equal to the latest value of an error signal on the left. Heuristic responsibility of the left child is defined symmetrically. This is called the "latest error" strategy, and the details are given in [3, 4].

One improvement on latest error is to combine it with true responsibility and make a child node heuristically responsible if its parent is heuristically responsible and if the child satisfies the requirements for latest error or true responsibility. Other techniques for defining heuristic responsibility combine error responsibility with true responsibility. It has also appeared beneficial to make both subtrees of a LEFT or RIGHT node responsible whenever the node is. The determination of effective heuristics for adaptation is one of the main problems in training logic networks. We are indebted to Bernard Widrow for an insightful explanation [14] concerning the "minimal disturbance principle": "...during training it is advisable to inject new information into a network in a manner which disturbs stored information to the smallest extent possible." This principle has proved very helpful in devising highly specific credit assignment heuristics.

4 Parsimonious evaluation

We now investigate parsimony theoretically in the case of a balanced binary tree with AND, OR, LEFT and RIGHT nodes, as is the situation during training. Training requires evaluation not only of the whole tree's output, but also of some subtree outputs used only to determine heuristic responsibility, so parsimony is particularly important in training. We assume that the assignment of node functions is random. If $E(L)$ is the total work required to evaluate an L-layer balanced tree, then we get a recursion: $E(L+1) = E(L)+0.5E(L)+1$, where the 1 represents the work to evaluate a node, given the required input or inputs. We are making the plausible assumption here that if we always evaluate the left subtree of a node first, then we will need to evaluate the right subtree only half of the time. We get $E(L) = 2(1.5^L - 1)$. The speedup compared to a total evaluation, which involves work $2^L - 1$, is by a factor roughly $(4/3)^L/2$. If L = 10, then the value of this is about 8.9. For L=20, this theoretical speedup is by a factor of 157.

The value of parsimony may not seem evident in a situation where hardware is constructed to compute an L-layer tree's output in parallel. However, by a similar recursion to the above, the expected number of leaf inputs which must be known in an L-layer tree is 1.5^L. For $L = 10$ only 58 of the 1024 leaf inputs are required, and for $L = 20$ only 3325 of the more than one million. As percentages, these are 5.7% and 0.3% of all the leaf inputs respectively. Since several leaves of the tree are connected to the same bit (either coming directly from a boolean input vector, or coming out of the coders in Figure 1), the percentage of those bits needed is higher by a factor at most equal to the average replication factor of the connections.

Now let us suppose that our hardware tree exists in only one copy, and that we must load a function into the tree before it can work. Let us further suppose that we compose trees in such a way that the inputs to one tree function are the outputs of other tree functions. By not having to evaluate all of the inputs to the tree at the root, but only those which parsimony requires, we can get a considerable speedup. At some point, the cost of the hardware required to do any computation totally in parallel will become prohibitive for any type of neural network, but if provision is made for detecting those inputs whose value is not needed, parsimony can generally provide exponentially growing speedup to any depth of network.

Finally, returning to our case of boolean tree networks, we noted before that all LEFT and RIGHT functions can be eliminated from a trained tree, and maximal subtrees having the same node function assigned can be replaced by a single node in a tree with fan-in not restricted to two. This results in a tree having alternating layers of ANDs and ORs. The tree is not necessarily balanced, however that does not hinder its parsimonious evaluation, which is even faster than for the binary tree. Some work on this is due to Jiandong Liang, who has shown, for example, that such parsimonious evaluation of multiway trees in the case of fan-in 5 and depth 8 produces a speedup factor of 625.

5 Boolean functions

We look now at some experimental results. First, a task was done which tested the capacity of tree networks to learn random boolean training data. A training set of T vectors was chosen randomly from the space $\{0,1\}^n$; and a tree was randomly initialized and trained until all the required randomly-chosen responses to vectors in the training set were correctly learned, or until a termination criterion was met. An epoch here consisted of presenting T patterns in randomly chosen order. Here are the results: the task was completely learned in all cases.

n	T	Initial Tree Size	Patterns Presented	Epochs
20	200	8192	5000	25
20	400	8192	14000	35
20	800	65536	19000	24
40	800	65536	9000	12

We continue with some results by Scott Reynolds on learning multiplexer boolean functions. A multiplexer is a digital logic circuit which has k input leads called control leads, and 2^k other input leads. If the input signals on the control leads represent the number j in binary arithmetic, then the output of the circuit is defined to be equal to the value of the input signal on the j-th one of the other leads (in some fixed order). A multiplexer is thus a boolean function of $n = k + 2^k$ variables, and is referred to as an n-multiplexer. In this section, $k = 4, n = 20$.

The table shows the number T of vectors in the training set, the initial number of nodes in the tree before training, the final number after training and removal of branches cut off by LEFT and RIGHT functions, and the percentage of correct responses for a randomly chosen test set after training, %Test. All training runs reached 100% correct performance on the last 1000 training vectors. Only a part of the set of all training vectors was actually presented to the network for the last two runs; and testing was done on a sample of several thousand.

T	Initial Nodes	Final Nodes	%Test
200	2047	497	55
500	2047	680	63
1000	9999	2082	69
5000	29999	4311	98.6
1048576	29999	643	100
1048576	59999	2896	99.9

The results show that the multiplexer function is not only learned correctly for vectors in the training set, but that extrapolation is almost perfect if the initial trees contain enough elements. (We do not doubt that the results could have been made absolutely perfect, if training and testing had been more thorough). We interpret this to imply the following extremely important property of tree networks:*the trees tend to learn a "simple" explanation of the training data.* For each pattern of values on the control leads, the correct function of the values on the other leads is the value on a particular one of them. If a limited sample is chosen, then there may be many simple functions of two or three variables which do just as well. Indeed, in the case of a training set size of 200, there is at least one pattern of the control inputs represented by less than $\lfloor 200/16 \rfloor = 12$ vectors, and probably a few by less than 8. There is no reason, *a priori*, why the function learned should not be any function of the 16 other inputs which just happens to always equal the correct value for this small set of vectors. However the trees apparently pick a simple function. The correct extrapolation of the multiplexer requires being insensitive to all but the correct input bit, given a particular pattern on the control leads. We note that in the case of a very large initial tree of 59999 nodes, where there are many more choices of functions available for synthesizing the result, the resulting trees are larger and the extrapolation is worse than when a tree half as big was chosen initially.

6 Tree structure modification

This section gives some results on learning multiplexer functions with a large number of input variables by a technique which modifies the tree structure during learning. Clearly, if a binary tree is given, together with connections and inverted connections to bits of an input vector, then there may be no assignment of node functions which satisfies the given training data. One can then either start the adaptation process over with a larger tree, or try to add some nodes to the existing tree with additional connections to the input variables. To make the latter strategy feasible, one has to be able to identify leaves or internal nodes of the tree whose signals could be improved. Such a signal, instead of being passed directly to its parent could be routed through a LEFT function, whereby the other input of the LEFT node would be attached to a new randomly initialized subtree, randomly connected to the bits and complements of the input vector. The "error" heuristic responsibility component of our adaptation strategy assures that some adaptation of the newly attached part will occur.

The experiments done on multiplexers have shown that iterative addition of new tree structure can lead to very good fitting of the training data, probably only limited by the computer time one is willing to invest. The tree may grow in this manner, while adaptation of the entire tree can lead to the development of LEFT and RIGHT node functions which causes the tree to shrink when such nodes are eliminated.

A key issue is then: where should a new subtree be connected into the existing tree? The strategy we have used is to consider, for every node input or output signal in the tree, the subset of the training vectors for which that signal is heuristically responsible in some sense. The number of those vectors for which the signal is not equal to the desired network output is a criterion which can be used to pick out the "worst" signal. That signal then is the one which will have a new subtree grafted to it by inserting a LEFT node carrying the signal. Some modification of the above criterion has been made to avoid putting too much emphasis on the root node. This method has been developed by Dekang Lin.

One interesting question in regard to our results is whether this incremental function synthesis technique can preserve good extrapolation. Since the training sets for larger numbers of control bits form a really sparse set in the set of all boolean n-vectors, there may be many "explanations" of the training data. For example if the number of control bits is $k = 9$, then there are, on average, only 8000/512, or about 16 samples for each choice of control bit values. There is a 0.0078 probability that some other bit position than the one used in defining the multiplexer also has the correct values.

But there could be many simple functions of two or more bits among the 512 which could also "explain" the training data as well as the particular bit position indicated by the choice of control bits. The fact that extrapolation does occur is indicative that the adaptive tree method, when augmented by the above procedure for adding new tree structure, *does* find the correct explanation quite often.

We use the following symbols below (times are for a SPARC station):

k =	number of control bits
n =	number of bits in an input vector $(k + 2^k)$
inodes =	number of nodes in the initial tree
T =	number of training vectors
train =	correct fraction on training set
test =	correct fraction on testing set
fnodes =	number of nodes in the final tree
time =	CPU time used (seconds) for a NON-parsimonious algorithm

k/n	inodes	T	train	test	fnodes	time
6/70	149	4000				
tree 1.			0.991250	0.940500	1558	8365.0
tree 2.			0.987750	0.936250	1411	7034.2
tree 3.			0.964750	0.940500	1489	9481.0
vote				0.988000		
7/135	999	6000				
tree 1.			0.881333	0.840500	2509	9479.0
tree 2.			0.870167	0.824500	2262	7889.7
tree 3.			0.900667	0.863167	2516	8136.4
vote				0.928333		
8/264	999	6000				
tree 1.			0.992000	0.970500	256	1754.1
tree 2.			0.986000	0.966667	241	1894.6
tree 3.			0.984500	0.958667	295	2038.0
vote				0.998167		
9/521	999	8000				
tree 1.			0.874000	0.814500	2149	13261.1
tree 2.			0.860750	0.813875	2271	12843.2
tree 3.			0.896500	0.836875	2307	13392.7
vote				0.911000		

The result for the 264-multiplexer is strange, as though the adaptation had three times found a simple way of synthesizing the function. The tree seems to be smaller than a straightforward synthesis would dictate. More work will have to be done to find out why the results were so good in that particular case.

Since the ability to form simple explanations which support generalization is a fundamental component of intelligent behavior, we feel that it is important to explore further the capabilities of adaptive tree networks in this regard.

7 Results on continuous real-valued functions.

There are many ways of transforming data from real-valued functions of several real variables into boolean data in order to use adaptive logic tree networks. The obvious way: using the bits of the real numbers represented radix two, is not generally appropriate, since the less significant bits of any number are essentially meaningless when taken in isolation, or when any of the more significant bits is missing. Hence an encoding of reals as bit strings is needed which puts about equal weight on each bit of the representation. One technique has been described by D. Smith and P. Stanford [12]. Given a sequence of k quantization levels of a real variable, k points in a Hamming space $\{0,1\}^n$ are generated as follows. A random point is picked to represent the first quantization level. Then k-1 steps are taken, each one to a point Hamming distance p from the one previously reached, and the second and following quantization levels are coded into the boolean vectors of the resulting random walk.

Insert figure 2

Figure 2: Approximation of $\mid \sin(X) \mid$.

Figure 2 shows a one-variable function used to test this approach: two complete periods of a rectified sine wave. The X vectors, quantized to 1000 levels, were encoded into 256 bit vectors with stepsize 1, and the Y vector, quantized to 10 levels, into 20 bits with stepsize three. Only 200 input-output pairs were used for training out of the 1000 possible. Twenty tree networks were each trained to compute one of those bits as a function of the boolean-encoded X-vector. For X-vectors not in the training set, the bits emerging from the trees may not represent a quantization level of Y. A quantized real result was obtained by finding the boolean code vector of a quantized Y value closest to the trees' output vector. (This is an inefficient technique and amounts to decoding a random code! Systematic codes are also being used.) It appears that the few mistakes made were by one or two quantization levels of Y.

A two-variable function that was tested was a spherical harmonic Y_2 defined by:

$$\begin{aligned} Y_2(\mu, \phi) &= A_0(3\mu^2/2 - 1/2) \\ &\quad +3\mu(1-\mu^2)^{1/2}[A_1 \cos\phi + B_1 \sin\phi] \\ &\quad +3(1-\mu^2)[A_2 \cos 2\phi + B_2 \sin 2\phi] \end{aligned}$$

We set $A_0 = 1.0$, $A_1 = 0.4$, $B_1 = 0.9$, $A_2 = 2.4$, $B_2 = 7.9$. The values of μ were in the interval [-1.0, 1.0], and the values of ϕ were in [0.0, 2π], while the values of Y_2 ranged between -26.0 and 26.0. The μ and ϕ intervals were quantized into 100 levels each; the random walks had 64 bits and a stepsize of 3. The Y_2 values were quantized into 10 levels, the random walk having 8 bits with a stepsize of 3. Training 8 networks of 2047 nodes on 1000 samples resulted in a function which, during test on 1000 new samples, was decoded to the correct quantization level 75.3% of the time. The error in the quantized result was no more than 2 quantization levels for 98.5% of the test samples.

In another test, the Y_2 value was quantized into 100 levels and coded into 64 bits with a stepsize of 3. Upon testing on 1000 random points, the decoded value from the trees was within 13 quantization levels of the correct value for all test points, within 10 levels for 99% of the points, and within 7 levels for 96.2% of the points.

These results on extrapolation of real-valued functions of real variables show promise for high-speed computation of quite general functions when coarse quantization of the output is adequate. We have in mind problems in controlling robot motion in a closed loop, where errors can be corrected.

In case high precision is required, we suspect that it would be better to use a collection of k logic networks to divide the space of continuous inputs into 2^k regions. In each region, some simple linear or polynomial approximant(s) would be used. If special tree hardware were available, this would be very fast.

Despite our attempts to interpolate by random walk encoding, which locally preserves a linear topology, trees appear to interpolate by reproducing the value from some neighboring point; undoubtedly the result of the insensitivity property of trees. We see this again in the next section.

8 Learning the multiplication table

An experiment was done with the ordinary multiplication table, which is a mapping of type $\{0, \ldots, 10\} \times \{0, \ldots, 10\} \to \{0, \ldots, 100\}$. In a paper by James A. Anderson [2], it is stated "Due to computing limitations (we estimated that a full simulation of the times table would require several hours of CPU time on a Cyber 205) we were forced to use a simplified times table ...." The suggested difficulty provided the stimulus to try the problem.

The hardware used for these experiments was the Myrias SPS-2. Contrary to our expectation, the problem proved to be quite easy by the method of adaptive tree networks using random walks in Hamming space for the encoding. Each quantity was encoded into 32 bits, with stepsizes of 3 and 1 on the inputs and output respectively. The complete table was learned in less than fifteen minutes. The difference between this result and the dire predictions of Anderson is due to parsimonious evaluation and the computation of the output bits of all the trees in parallel using an MIMD machine.

In another experiment, only sixty-one input pairs were presented in training (selected in the pattern of a checkerboard). All of the sixty-one were learned perfectly. Eleven others were extrapolated perfectly, namely where one of the factors was a zero so that a neighboring value could be used. The total learning time taken on the SPS-2 for this was only three and one-half minutes.

A fact that must be remembered in working with ALNs is that they do not arithmetically interpolate values. Although it might seem reasonable to expect that if "2 x 2 = 4" and "2 x 4 = 8" have been learned in training, then "2 x 3 = 6" will be the result of generalization, this is not the way ALN extrapolation works. An ALN would probably generalize to either "2 x 3 = 4" or "2 x 3 = 8". On the other hand, ALNs might not overshoot like a polynomial, which might generalize to "2 x 3 = 10".

We are working on ways to perform interpolations correctly. The idea is to use ALNs to do what they are good at: classification. If a forest of ALNs can classify an input point into a class i, and if all points in class i have a simple (e. g. cubic spline) approximant given by a set of coefficients that can be accessed from RAM, then we obtain a fast computation that may outperform either a global numerical method or a standard neural network one.

9 The "two spirals" problem

The two spirals problem (from Alexis Wieland of MITRE Corp.) is reported in [13] and is part of the collection of neural network problems maintained by Scott Fahlman at Carnegie Mellon University [8]. The task consists of 194 points, i. e. pairs of real numbers, arranged in the form of two interlocked spirals on a plane. One of the spirals consists of black points, the other of white. The white points were to be output as 1's and the blacks as 0's.

We required the adaptive algorithm under test to learn the correct output for each of the points, and we wanted to determine if the required 0 and 1 values would extrapolate smoothly for the points between the spirals, which it had not seen during training.

We quantized the real numbers from -7.0 to 7.0 into 57 levels, and represented each by a 32-bit code. Each code differed from its neighbours by one bit. For every point in the training set, the X and Y coordinate values were encoded using this scheme, and the two 32-bit words were concatenated to form a 64-bit boolean input vector. The connections to those bits and their complements from the leaves of the tree were randomly picked prior to training.

The training set formed in this way was presented to a number of trees in parallel on the Myrias SPS-2. Each tree consisted of 1599 nodes and typically learned the training set perfectly after 7 epochs. This observation compares extremely favorably with 1700 epochs required by the Cascade-Correlation algorithm, and 20,000 epochs required by backpropagation [8]. Some of the problems used spirals with 770 points instead of the 194 we used, but that should not completely invalidate the comparison.

After the training period was complete, each tree was applied to all $57^2 = 3429$ discrete points, and its classification, black or white, was recorded. An individual point was then classified by the majority vote over all the trees run during the experiment.

Insert figure 3

Figure 3: Solution of the two-spirals problem.

We obtained the result shown in Figure 3, using a majority vote of 31 trees which required 1062 seconds to execute on the SPS-2. Keeping in mind that the algorithm has seen less than 6% of the points, we feel that this demonstrates excellent extrapolation; indeed, the spirals are quite easily made out in the figure.

10 Conclusions

A learning procedure has been described for binary tree functions. The experiments on boolean functions showed that randomly specified responses to 800 randomly-chosen 40-bit input vectors could be learned by trees of under 60,000 nodes. Experiments on learning sample data derived from multiplexer functions showed it is necessary to take trees and a training set large enough, in which case a 20-input multiplexer could be perfectly learned on a training set of 5000 vectors and extrapolated to a test set with 98.8% accuracy. For training on the whole set of 20-bit vectors, the result of learning and testing was perfect in one case, up to the limits of our measurement. Experiments on multiplexers defined on high dimensional boolean spaces (up to 521 variables) have shown that good extrapolation from a training set to a test set is preserved by a procedure for adding new nodes to a tree structure during adaptation. The use of several tree functions in a majority vote improves improve extrapolation.

The technique of encoding continuous or integer values into the steps of a random walk in Hamming space worked very well in supporting learning of such functions as a rectified sine wave, a spherical harmonic, and the multiplication table. It appears that the boolean tree networks can provide approximate synthesis of functions with such values as long as the number of quantization levels of the output is small. Extrapolation appears to act by repeating a neighboring learned output value. Other problems attacked, e.g. the two spirals problem, show that logic networks can learn to perform benchmark tasks done by ANS using backpropagation or Cascade-Correlation.

Speedups obtained through parsimony were estimated theoretically, and shown to amount to one or two orders of magnitude in small problems. Parsimony is not only valuable in the case of networks executed by simulation on standard computers, it provides a speedup when a parallel hardware unit is used iteratively; and the speedup compared to a complete evaluation grows exponentially with the depth of the tree. That advantage of logic trees, and the inherent speed of programmable combinational hardware, make adaptive logic networks an attractive prospect for ANS applications requiring extremely high speed.

11 References

1 I. Aleksander, H. Morton, An Introduction to Neural Computing, Chapman & Hall, London, 1990.

2 James A. Anderson, Data Representation in Neural Nets, AI Expert, June 1990, 30 - 37.

3 W. W. Armstrong, J. Gecsei: Adaptation Algorithms for Binary Tree Networks, IEEE Trans. on Systems, Man and Cybernetics, SMC-9 (1979), pp. 276 - 285.

4 W. W. Armstrong, Adaptive Boolean Logic Element, U. S. Patent 3,934,231, Jan. 20, 1976, assigned to Dendronic Decisions Limited.

5 W. W. Armstrong, J. Gecsei, Architecture of a Tree-Based Image Processor, 12th Asilomar Conf. on Circuits, Systems and Computers, IEEE Cat. No. 78CHI369-8 C/CAS/CS Nov. 1978, 345-349.

6 W. W. Armstrong, G. Godbout, Properties of binary trees of flexible elements useful in pattern recognition, Proc. IEEE Int'l. Conf. on Cybernetics and Society, San Francisco (1975) 447 - 450.

7 G. v. Bochmann, W. W. Armstrong: Properties of Boolean Functions with a Tree Decomposition, BIT 14 (1974), pp. 1 - 13.

8 Scott Fahlman, "Two Spirals" problem data, personal communication, August 1990.

9 Robert Hecht-Nielsen, Neurocomputing, Addison-Wesley, 1990.

10 William S. Meisel, Parsimony in Neural Networks, Proc. IJCNN-90-WASH-DC, vol. I, pp. 443 - 446.

11 D. E. Rumelhart and J. L. McClelland: Parallel Distributed Processing, vols. 1&2, MIT Press, Cambridge, Mass. (1986).

12 Derek Smith, Paul Stanford: A Random Walk in Hamming Space, Proc. Int. Joint Conf. on Neural Networks, San Diego, vol. 2 (1990) 465 - 470.

13 David S. Touretzky, Dean A. Pomerleau, What's Hidden in the Hidden Layers?, BYTE, August (1989), 227 - 233.

14 Bernard Widrow, Lectures on Neural Networks (tutorial notes), Int'l Joint Conf. on Neural Networks, San Diego, 1990.