An Implementation of Adaptive Logic Networks

November 11, 1990

initial implementation on Unix (TM AT&T)

bug-fixes and initial port to DOS
Rolf Manderscheid
April 15, 1991

ported to Microsoft Windows
Monroe M. Thomas
May 31, 1991

revised for Version 2.0
Monroe M. Thomas
July 17, 1991

1 Introduction

The C library **atree.c** contains an implementation of an unconventional kind of learning algorithm for adaptive logic networks[Arms], which can be used in place of the backpropagation algorithm for multilayer feedforward artificial neural networks [Hech], [Rume].

The ability of a logic network to learn or adapt to produce an arbitrary boolean function specified by some empirical "training" data is certainly important for the success of the method, but there is another property of logic networks which is also essential. It is the ability to generalize their responses to new inputs, presented after training is completed. The successful generalization properties of these logic networks are based on the observation, backed up by a theory [Boch], that trees of two-input logic gates of types AND, OR, LEFT, and RIGHT are very insensitive to changes of their inputs.

Some experiments on handwritten numeral recognition and satellite image classification have been successfully carried out. [Arms3, Arms4]. Recent experiments have shown this algorithm to learn quickly on some problems requiring learning of integer or continuous-valued functions where backpropagation has reportedly led to long training times; and it functions very well on boolean data [Arms5].

At the present time, only limited comparisons have been made with the conventional approach to neurocomputing, so the claims necessarily have to be muted. This situation should rapidly be overcome as users of this software (or improved variants of it yet to come) begin experimentation. However one property of these networks in comparison to others is an absolute, and will become apparent to computer scientists just by examining the basic architecture of the networks. Namely, when special hardware is available, this technique, because it is based on combinational logic circuits of limited depth (e. g. 10 to 20 propagation delays), can potentially offer far greater execution speeds than other techniques which depend on floating point multiplications, additions, and computation of sigmoidal functions.

A description of the class of learning algorithms and their hardware realizations can be found in [Arms, Arms2], but we will briefly introduce the concepts here. An **atree** (Adaptive TREE) is a binary tree with nodes of two types: (1) adaptive elements, and (2) leaves. Each element can operate as an **AND**, **OR**, **LEFT**, or **RIGHT** gate, depending on its state. The state is determined by two counters which change only during training. The leaf nodes of the tree serve only to collect the inputs to the subtree of elements. Each one takes its input bit from a boolean input vector or from the vector consisting of the complemented bits of the boolean input vector. The tree produces a single bit as its output.

Despite the apparent limitation to boolean data, simple table-lookups permit representing non-boolean input values (integers or reals for example) as bit vectors, and these representations are concatenated and complemented to form the inputs at the leaves of the tree. For computing non-boolean outputs, several trees are used in parallel to produce a vector of bits representing the output value.

This software contains everything needed for a programmer with knowledge of C and Windows

3.x to create, train, evaluate, and print out adaptive logic networks. It has been written for clarity rather than speed in the hope that it will aid the user in understanding the algorithms involved. The intention was to try make this version faster than variants of the backpropagation algorithm for learning, and to offer much faster evaluation of learned functions than the standard approach given the same general-purpose computing hardware. Users of the software are requested to provide some feedback on this point to the authors.

This software also includes a language "lf" that allows a non-programmer to conduct experiments using atrees, as well as a number of demonstrations.

A version of this software which is both faster and contains a more effective learning algorithm is planned for the near future.

Figure 1: Using several trees to compute Y = *f*(X1, X2)

2 Writing Applications With atree

Writing applications that perform a simple classification (yes or no) is relatively easy (within the constraints of Windows programming). The programmer creates a training set, then creates a tree using **atree_create()**. The tree is trained using **atree_train()** and then it can be used to evaluate new inputs using **atree_eval()**. Examples of this can be seen in the files **mosquito.c**, and **mult.c**, both of which hide most of Windows' dressings for clarity.

Writing applications where the tree has to learn real number valued functions is a little more complex, as the programmer has to come to grips with the encoding problem.

Because a single tree produces only one bit, the programmer must train several trees on the input data, each one responsible for one bit of the output data. This is made slightly simpler by the choice of parameters for **atree_train()** which takes an array of bit vectors as the training set, and an array of bit vectors for the result set. The programmer provides an integer which states which bit column of the result set the current tree is being trained on. Typical code might look as follows:-

```
....
{
   int i;
   int j;
   LPBIT_VEC train;
   LPBIT_VEC result;
   LPATREE *forest;

   /* Create the training set using your own domain function*/
   train = domain();

   /* Create the result set */
   result = codomain();

   /*
    * Make enough room for the set of trees - one tree per bit in the
    * codomain
    */
   forest = (LPATREE *) Malloc((unsigned)sizeof(LPATREE) * NO_OF_TREES);

   /* Now create and train each tree in turn */
   for (i = 0; i < NO_OF_TREES; i++)
   {
      forest[i] = atree_create(variables,width);
      atree_train(forest[i], train,  result, i, TRAIN_SET_SIZE,
               MIN_CORRECT, MAX_EPOCHS, VERBOSITY);
   }

   /*
```

```
    * Where TRAIN_SET_SIZE is the number of elements in train,
    * MIN_CORRECT is the minimum number of elements the tree should
    * get correct before stopping, MAX_EPOCHS is the absolute maximum
    * length of training and VERBOSITY controls the amount of
    * diagnostic information produced.
    */
.....
```

The standard encoding of integers into binary numbers does not work well with this algorithm since it tends to produce functions which are sensitive to the values of the least significant bit. So instead we use the routine **atree_rand_walk()** to produce an array of bit vectors where each vector is picked at random and is a specified Hamming distance away from the previous element. Picking the width of the encoding vector, and the size of the step in Hamming space is currently a matter of experimentation, although some theory is currently under development to guide this choice.

Real numbers are encoded by dividing the real number line into a number of quantization levels, and placing each real number to be encoded into a particular quantization. Obviously, the more quantization levels there are, the more accurate the encoding will be. Essentially this procedure turns real numbers into integers for the purposes of training. The quantizations are then turned into bit vectors using the random walk technique again.

Once the trees are trained, we can evaluate them with new inputs. Despite their training, the trees may not be totally accurate, and we need some way of dealing with error. The normal approach taken is to produce a result from the set of trees, then search through the random walk for the closest bit vector. This is taken as the true result. Typical code might be as follows:-

```
....
    /* Continued from previous example */
    int closest_elem;
    int smallest_diff;
    int s;
    LPBIT_VEC test;
    LPBIT_VEC tree_result;

    /* Now create the (single in this example) test vector */

    test = test_element();

    /* Now create some room for the tree results */

    tree_result = bv_create(No_OF_TREES);

    /* Evaluate the trees */

    for (i = 0; i < NO_OF_TREES; i++)
    {
```

```
        /*
         * Set bit i of tree_result, the result of evaluating
         * the ith tree.
         */

        bv_set(i, tree_result, atree_eval(forest[i], test));
    }

    /*
     * tree_result probably has a few bits wrong, so we will look
     * for the closest element in the result array
     */

    closest_elem = 0;
    smallest_diff = MAX_INT;

    for (i = 0; i < TRAIN_SET_SIZE; i++)
    {
        if ((s = bv_diff(tree_result, result[i])) < smallest_diff)
        {
            smallest_diff = s;
            closest_elem = i;
        }
    }

    /*
     * At this point, result[closest_elem] is the correct bit vector,
     * and smallest_diff is the amount of error produced by the tree.
     */

   do_something_with(result[closest_elem]);

    /* Etc. */
}
....
```

3 The Windows atree Library

The atree library consists of a single include file **atree.h**, which must be included in all software making calls on the library, and a library of routines **atree.c**. The routines permit the creation, training, evaluation and testing of adaptive logic networks in a Windows environment, and there are a number of utility routines designed to make this task easier.

Important note: the module definition file for your application must include in its **EXPORT** section the name of the atree Status window procedure: **VerbosityWndProc**, along with any other window procedures your application may have - see **mosquito.def** for an example.

3.1 Naming Conventions

Throughout this software, the following conventions have been used :-

Publicly available functions are called **atree_something()**. If the routine is primarily concerned with bit vectors rather than atrees, it will be named **bv_something()** instead. The exceptions to this occur for functions that are directly responsible for maintaining performance of the atree software in the Windows environment.

Variables are always in lower case. The variables **i**, **j**, and **k** are reserved as iterators in "for" loops. The variable **verbosity** is reserved for controlling the amount of diagnostic information produced.

3.2 Public Macros

The following macros are defined in **atree.h** and are available to any application using the atree library.

The macro **MEMCHECK** allows us to check the validity of a pointer. For example, if the pointer **p** in **MEMCHECK(p)** is NULL, then a message box pops up with appropriate notification, and the application is terminated.

The macro **RANDOM** allows us to conveniently produce a random number between 0 and some user-specified x in the program. For example, in order to produce a random true or false value (0 or 1) we write **RANDOM(2)**.

The macro **Malloc** serves as a front end for the atree memory allocation routine **WinMem_Malloc()**. To allocate a chunk of 16 bytes to a pointer **p**, use **p = Malloc(16)**.

The macro **Free** serves as a front end for the atree memory routine **WinMem_Free()**. To free the memory pointed by a pointer **p** that was allocated with **WinMem_Malloc()** (or the macro **Malloc**), use **Free(p)**.

3.3 void atree_init(hInstance)

HANDLE hInstance;

This routine should be called by the user before making calls to any other atree library routine. The parameter **hInstance** should be the instance handle given to your application by Windows in your **WinMain()** procedure. Currently, **atree_init()** calls the **srand()** routine to initialize the random number generator and initializes the atree Status window.

3.4 void atree_quit()

This routine sets the internal **atree_quit_flag** variable to TRUE to notify all atree procedures that it is time to drop whatever it is they are doing and quit. This procedure should be called before your application exits so that any running atree procedures are not left in memory.

3.5 LPBIT_VEC atree_rand_walk(num,width,p)

int num;
int width;
int p;

The standard encoding of integers into binary is not suitable for adaptive logic networks, since the least significant bits vary quickly during incrementations of the integer compared to the more significant bits. The effect of binary number encoding is easy to see when we consider the result of a single bit error occurring in the output of a collection of trees (a forest): how important the error is depends on the position of the bit in the output vector. An error in the least significant bit of the vector makes a difference of one unit in the output integer; an error in the most significant bit causes a large difference in the output integer depending on the width of the vector.

A better encoding is one where each bit varies at about the same rate; and we can create such an encoding by taking a random walk through Hamming space [Smit]. A randomly produced vector is chosen to represent the first integer value in a sequence. For each subsequent integer, a specified number of bits, picked a random, are changed to create the next vector.

The routine **atree_rand_walk()** does this job, with the additional guarantee that each vector produced is unique. The parameter **num** gives the number of vectors, or "steps" in the walk, required, the parameter **width** gives the width in bits of each vector, and the parameter **p** is the distance of each step in the Hamming metric (the number of bits which change).

The uniqueness requirement makes the routine rather more complex than one might expect. Because we expect to be using large random walks, it was felt that a simple check against all the previously created vectors would not be efficient enough. Instead all vectors with the same weight (the weight of a bit vector is the number of 1s in it; e. g., the weight of 10110 is 3) are chained together, and only those vectors with a weight equal to the one currently being checked for uniqueness are examined. If the vector is not unique, the routine will go back to the previous unique vector and try randomly changing some other bits. In order to avoid an infinite loop, it will only try MAX_RETRY times to do this. If it cannot proceed, the routine aborts. A better version of the software would check to assure a minimum distance between points.

A bit of thought must go into the choice of **width**, the length of the bitstring used to encode a quantity, and to the stepsize **p**. Suppose we want **num** quantization levels for a variable x. Then the width used to code x must be at least as large as the logarithm base 2 of **num** to make the codes unique. A thermometer code would use **num** - 1 bits, where the quantization level i (starting at 0 and increasing to **num** - 1) is represented by i 1s at the left of the vector completed by 0s at the right. For example, if **num** = 100, then **width** must be at least 7, while the

thermometer code would use 99 bits. The width for an input variable is not as critical as for an output variable, since we need to train one tree for each bit in the output.

Suppose some training data contains vectors with two variables in the domain. Two domain vectors (x1, x2) could be (3.14, 9.33), corresponding to levels (11, 17), and (3.18, 9.48), corresponding to (13, 18), say. The function y=f(x1,x2) to be learned is supposed continuous, so the two function values could be 34.6 and 33.9, with neighboring quantization levels 67 and 66 respectively. If the training set has been learned perfectly, then we shall get the correct boolean codes for levels 67 and 66 from the trained forest of trees on input of these vectors. When we only have vectors close to the above two vectors in Euclidean distance, then problems arise.

Suppose we have an input (3.15, 9.34), corresponding also to levels (11, 17). Obviously, the trees give the same response as for (3.14, .933), namely level 67 of y. As long as this is an acceptable approximation to the desired function, there is no problem. If the quantized function value varied too much within the set of real vectors corresponding to (11,17), we would have to use a finer quantization on the inputs.

Next suppose that the training set contained no sample with quantization levels (12, 17). Then we would like the system to be able to take advantage of the similarity between the concatenated codes for (12, 17) and (11, 17) to be able to extrapolate its output. This can occur if the codes for levels 11 and 12 of x1 are close in Hamming distance. If, on the other hand, they were 1/2 of the width of the code for x1 apart, then the system could just as well extrapolate from a training point with levels (96, 17) as from (11, 71). So we would take **p** for the random walk for x1 to be less than, say, 1/4 **width** to make sure levels 11 and 12 of x1 are close. This is because random pairs of points in the Hamming space tend to be 1/2 width apart.

If neighboring training samples tended to have x1 values which are four levels apart, e. g. (11, 17) and (15, 17), then 4 * **p** would have to be less than 1/4 width. Now if we vary both x1 and x2, then neighboring input vectors might tend to be four levels apart in x1 and 7 levels apart in x2. Then the values px1, px2 chosen for **p** for the two walks should be such that 4 * px1 + 7 * px2 is less than 1/4 the sum of the widths wx1, wx2 for coding x1, x2.

There is a good reason for using a large value of the p for the output variable y. Namely, for a given input vector, some of the **width** trees may produce an output bit that is different from that of a code of the correct quantization level of y as one varies the input a bit. If fewer than **p**/2 output bits are changed, we are still close to the original code, and the same output quantization level would still be recovered by minimum distance decoding. Consider the case where there are only **num** = 2 levels of y, and they are encoded 00000 and 11111, with **width** = 5 and **p** = 5. As we move away from inputs resulting in a correct response, say 00000, to those having two bits different, like 01001, the decoded output will maintain its value.

So for the output variables, choosing larger values for **p** and **width** can provide error correction just as taking a majority vote does for a boolean output.

We are aware that this only touches the surface of the questions involved with choosing the Hamming codes for continuous variables. The general assumptions are that real intervals are mapped into random walks in a way that locally preserves "proximity", and it is the proximity of

elements in the domain and codomain that determines the quality of extrapolation. Instead of using random walks, some work has been done using algebraic codes, in particular Golay codes. This will be discussed in a thesis at U. of A. by Allen Supynuk, which is soon to be completed.

3.6 public LPATREE atree_create(numvars,leaves)

int numvars;
int leaves;

This is the routine used to create an atree of a given size. The parameter leaves gives the number of **leaves** or output leads to the tree, and hence controls its size, which is one less than this. A balanced tree is chosen if possible.

The parameter **numvars** is the number of boolean variables in the bit vector input to the tree. It is used during initialization of the (random) connections between leaf nodes of the tree and the input bit vector. Usually the bits of the input vector, and their complements will be required as inputs to the tree since there are no NOT nodes in the tree itself. It is therefore recommended that there be at least twice as many inputs to the tree as there are bits in the input vector for a given problem:

leaves >= 2 * numvars

The atree library maintains two free lists, one each for leaves and nodes. **atree_create()** always uses memory from these lists. In the event that a free list is empty, a large block is allocated and added to the list. The size of the block can be adjusted by editing the compile-time constant **NEWMALLOC** defined in **atree.c**.

3.7 void atree_free(tree)

LPATREE tree;

This routine returns memory used by nodes and leaves of **tree** to the appropriate free list. Note that memory is not freed from the free lists.

3.8 BOOL atree_eval(tree,vec)

LPATREE tree;
LPBIT_VEC vec;

This routine is responsible for calculating the output of a tree from a given bit vector. It takes advantage of the standard C definition of **&&** and || to do this in the required parsimonious[1]

1I really don't like this word - it makes me think of Scrooge (A.D.). However, if you really had to pay for massive parallelism rather than parsimonious parallelism, I suppose you could be persuaded to like the term (W.A.). No I couldn't (A.D.).

fashion [Meis][Arms5].

This routine also marks subtrees that are unevaluated, and sets the internal **atree.n_sig_left** and **atree.n_sig_right** values for a node. This information is used when **atree_eval()** is used from within **atree_train()**.

3.9 BOOL atree_train(tree,tset,...)

LPATREE tree
LPBIT_VEC tset;
LPBIT_VEC correct_result;
int bit_col;
int tset_size;
int no_correct;
int epochs;
int verbosity;

This is the routine that adapts a tree to learn a particular function. It is a little more complex than you might expect as it has been arranged to make it convenient to train multiple trees on the same training set.

The parameter **tree** is the tree to be trained, and the parameter **tset** is the array of bit vectors which the tree is to be trained on (the training set). An atree only produces a single bit, so in principle all that is needed for
the **correct_result** parameter is an array of bits, with one bit corresponding to each bit vector in the training set. In training multiple trees (when learning a quantized real-valued function, for example), it is more convenient to keep the correct results in an array of bit vectors, and specify which column of the array a tree is supposed to be learning. This is the purpose of the array **correct_result** and the integer **bit_col**.

The next parameter **tset_size** gives the number of elements in **tset** and **correct_result** (which have to be the same --- there must be a result for every input to the function).

The next two parameters control the amount of training that is to be done. We train on the vectors of the training set in pseudo-random order. The term epoch here is used to mean a number of input vector presentations equal to the size of the training set. The parameter **epochs** states how many epochs may be completed before training halts. The parameter **no_correct** states how many elements in the training set the tree must get correct before training halts. The routine will therefore stop at whichever of these two conditions is true first. For example given that we have a training set with 10 elements and we wish to train for 15 epochs or until 90% of the elements in the training set have been responded to correctly. We can achieve this by setting **no_correct** to 9 and **epochs** to 15.

The **verbosity** parameter controls how much diagnostic information the routine will produce. At the moment only 0 (silent) or 1 (progress information) is implemented. The progress information

consists of an atree Status window that shows the progress of training.

The routine decides which vector is the next to be presented to the tree and extracts the result bit from the **correct_result** array. It also keeps track of the number of epochs, and the number of correct responses from the tree.

3.10 void atree_print(tree,verbosity)

LPATREE tree;
int verbosity;

This routine allows the programmer to output an atree to disk before, during, or after training, in a form suitable for printing. The parameter **tree** is the tree to be printed, and verbosity is the amount of information
produced. The disk file is currently hard coded as "atree.out" (future versions of the software will allow user selected output streams).

3.11 int atree_store(tree, filename)

LPATREE tree;
LPSTR filename; (LPSTR is Windows for "char far *")

This routine stores **tree** to **filename**. This routine is used to store a single tree, if you want to store a forest use **atree_write()**. Returns 0 for success, non-zero on failure.

3.12 LPATREE atree_load(filename)

LPSTR filename;

This routine reads **filename** and returns the tree stored therein. **atree_load()** reads exactly one tree from **filename**, if you want to read multiple trees use **atree_read()**. A NULL pointer is returned if any error or EOF is encountered.

3.13 LPATREE atree_read(stream)

FILE *stream;

This routine reads a single tree from the file referenced by **stream** and returns a pointer to it. Subsequent calls to **atree_read()** will read further trees from **stream**. A NULL pointer is returned if any error or EOF is encountered.

3.14 int atree_write(stream, tree)

FILE *stream;

LPATREE tree;

This routine writes **tree** onto the file referenced by **stream**. Trees are stored in postfix notation, with the characters `&', `--', `L', `R' representing the node functions **AND**, **OR**, **LEFT**, **RIGHT** respectively. Leaves are stored as a number, representing the bit index, optionally preceded by a `!' for negation. The end of the tree is marked by a semicolon. Returns 0 for success, 1 on failure.

3.15 LPATREE atree_fold(tree)

LPATREE tree;

This routine removes all **LEFT** and **RIGHT** nodes from **tree** and returns the result. This does not change the function represented by the tree, but the resulting tree may be considerably smaller and hence faster to execute. Nodes and leaves that are discarded are added to the free lists.

3.16 LPFAST_TREE atree_compress(tree)

LPATREE tree;

This routine returns the **fast_tree** derived from **tree**. A **fast_tree** is essentially a list of leaves; each leaf includes two pointers to subsequent leaves to evaluate, one for each possible result of evaluating the current leaf. It is the function of **atree_compress()** to calculate these two "next" pointers for each leaf. Experiments show that evaluating a fast_tree is almost twice as fast as evaluating the corresponding folded atree. This is due to the fact that recursion is eliminated. Fast_trees are also slightly more compact than the equivalent atree. Note that there is no "decompression" routine, and there are no fast_tree I/O routines.

3.17 int atree_fast_eval(tree, vec)

LPFAST_TREE tree;
LPBIT_VEC vec;

This routine is the equivalent of **atree_eval**, but for fast_trees.

3.18 void atree_fast_print(tree)

LPFAST_TREE tree;

This routine writes a representation of **tree** to the file "fasttree.out". Each line of output corresponds to a leaf and includes the leaf index, bit numbers (possible preceded by a `!' to indicate negation), and the two "next" pointers (shown as indices). NULL pointers are

represented by -1.

3.19 int atree_set_code(code, high, low, ...)

LPCODE_T code;
double low;
double high,
int vector_count;
int width;
int dist;

This is the constructor function for the type **code_t**. If width is greater than one, **atree_set_code()** calls **atree_rand_walk()** to get a random walk containing **vector_count** vectors, with adjacent vectors having a Hamming distance of **dist** between them. This random walk will represent numbers in the closed interval [**low**, **high**]. The functions **atree_encode()** and **atree_decode()** translate floating point quantities and bit vectors, respectively, into an index into the random walk. **atree_set_code()** also calculates the (real) distance between adjacent vectors and stores this in the **step** field of **code**.

If **width** is equal to one, the code represents a boolean variable, and no random walk is produced. In this case **low** , **high** , and **vector_count** are taken to be 0, 1, and 2 respectively. The **vector** field will be set to point to bit vectors of length one having the appropriate values.

3.20 int atree_encode (x, code)

double x;
LPCODE_T code;

This routine returns the quantization level of **x** as represented by **code**. To obtain the corresponding bit vector, use the expression:

my_bit_vec = code -> vector + atree_encode(x, code)

If the code is boolean (code -> width == 1), then **atree_encode()** returns 0 if **x** is 0, otherwise it returns 1. For non-boolean codes, **atree_encode()** issues a warning if **x** is out of range, and the output is clipped so that it lies within the range 0 .. code -> vector - 1.

3.21 int atree_decode(vec, code)

LPBIT_VEC vec;
LPCODE_T code;

This routine returns the quantization level of **vec** as represented by **code**. To obtain the corresponding floating point value, use the expression:

my_value = code -> low + (code -> step * atree_decode(vec, code)

The quantization level corresponds to the first bit vector stored in the random walk having the smallest Hamming distance from **vec**. If the code is boolean, the quantization level is simply the value of **vec** (whose length must be 1)**.**

3.22 LPCODE_T atree_read_code(stream, code)

FILE *stream;
LPCODE_T code;

This routine reads a coding from **stream** and fills the entries of the code structure. A NULL pointer is returned if any error or EOF is encountered.

3.23 int atree_write_code(stream, code)

FILE *stream;
LPCODE_T code;

This routine writes the contents of **code** onto **stream**. If the code is boolean, the **vector** field is not written. Returns 0 for success, 1 for failure.

4 The bv Library

4.1 LPBIT_VEC bv_create(length)

int length;

Creates a vector of **length** bits, where each bit is initialized to 0, and returns a long pointer to the bit vector.

4.2 LPBIT_VEC bv_pack(unpacked,length)

LPSTR unpacked;
int length;

This routine has been provided to make it easy for the programmer to produce bit vectors. The routine is handed an array of characters containing the value 0 or 1 (**unpacked**) and an integer **length** giving the number of bits. The routine returns a long pointer to a bit_vec.

4.3 int bv_diff(v1,v2)

LPBIT_VEC v1;
LPBIT_VEC v2;

This routine calculates the Hamming distance between **v1** and **v2**, i.e.

weight (v1 XOR v2)

where weight is the number of 1 bits in a vector and XOR is the bitwise exclusive-or operation. This routine is used to find the closest vector in a random walk array to some arbitrary vector. Just search through the random walk for the vector with the smallest difference from the vector of tree output bits. (Inefficient, but easier to understand than decoding an algebraic code!).

4.4 LPBIT_VEC bv_concat(n,vectors)

int n;
LPBIT_VEC far *vectors;

This routine is used by the programmer to join several bit vectors end-to-end to give the string concatenation of the vectors. This routine is most frequently used during the construction of training sets when elements of several random walks have to be joined together to obtain an input vector to a tree.

The parameter **vectors** is an array of LPBIT_VEC pointers, and the parameter **n** states how many of them there are. Vector pointers are used to make this routine a little faster since there is less copying involved. A long pointer to the concatenated bit_vec is returned.

4.5 void bv_print(stream, vector)

FILE *stream;
LPBIT_VEC vector;

This is a diagnostic routine used to print out a bit_vec.

4.6 void bv_set(n,vec,bit)

int n;
LPBIT_VEC vec;
BOOL bit;

This routine allows the programmer to explicitly set (or reset) the **n**th bit (0 to bit_vec.len - 1) bit in the vector **vec** to have the value in the parameter **bit**.

4.7 BOOL bv_extract(n,vec)

int n;
LPBIT_VEC vec;

This routine returns the value of the **n**th bit (0 to bit_vec.len - 1) in the bit vector **vec**.

4.8 BOOL bv_equal(v1,v2)

LPBIT_VEC v1;
LPBIT_VEC v2;

This routine tests two bit vectors for equality.

4.9 LPBIT_VEC bv_copy(vec)

LPBIT_VEC vec;

This routine returns a copy of **vec**.

4.10 void bv_free(vector)

LPBIT_VEC vector;

This routine frees the memory used by a bit_vec; accessing a bit_vec after it has been freed is usually disastrous.

5 Windows Support Library

5.1 void Windows_Interrupt(cElapsed)

DWORD cElapsed; (DWORD is Windows for "unsigned long")

When called, this procedure allows Windows to multitask an atree application with other Windows applications. This is accomplished with a **PeekMessage()** call (see the Windows Programmer's Reference for more details). The programmer may want to use this procedure during long tree evaluation and training set generation loops, or during other processing where control may not be passed back to the application's window procedure for lengthy periods of time (the price you pay for non-preemptive multitasking!). Since
PeekMessage() calls can be quite time consuming, this procedure will only call **PeekMessage()** after **cElapsed** milliseconds have passed since the last call to **PeekMessage()**. Experimentation has shown a value for **cElapsed** of about 1500 to work fairly well.

5.2 LPSTR WinMem_Malloc(wFlags, wBytes)

WORD wFlags; (WORD is Windows for "unsigned int(16-bit)")
WORD wBytes;

Since the segmented memory architecture of DOS based PC's can cause great grief when allocating large amounts of memory, the atree package includes its own memory manager. Requests for memory are obtained from dynamically allocated segments from the global heap in which local heaps have been initialized. The memory is actually allocated by Windows' local heap manager, and the resultant near (16 bit) pointer is combined with the global segment descriptor of the corresponding global heap segment to form a long (32 bit) pointer suitable for use in atree applications. wFlags indicates the kind of memory to allocate, usually LMEM_MOVEABLE, and wBytes indicate the number of bytes to allocate. See the Windows Programmer's Reference **LocalAlloc()** routine for more information on the different values wFlags may take. For ease of use, the programmer may simply wish to use the **Malloc(wBytes)** macro, which expands to

WinMem_Malloc (LMEM_MOVEABLE | LMEM_ZEROINIT, wBytes).

5.3 LPSTR WinMem_Free(lpfree)

LPSTR lpfree;

This function frees the block of memory pointed to by **lpfree**, which is decomposed into a segment selector, which is used to identify the global segment from which the near pointer was allocated from, and a near pointer, which is used by Windows' **LocalFree()** to free memory from the local heap in the dynamically allocated segment. If there remains no more allocated memory in the local heap the global segment is deallocated. For ease of use, the **Free(lp)** macro expands to **WinMem_Free((LPSTR) lp)**.

The function returns NULL if successful, otherwise it returns **lpfree**.

6 The Language lf

The second major product included in the current release is the "lf" language interpreter that allows a non-programmer to experiment with tree adaptation. The user specifies a training set, and a test set, and selects
the encoding and quantization levels for a particular experiment. The interpreter checks the statements for errors then executes the desired experiment, finally outputting a table comparing the desired function with
the function actually learned. Various post-processors can use the information to produce histograms of error or plots of the functions.

It is recommended that the user read and understand [Arms5] before using this language.

There are two versions of lf: LF.EXE and LFEDIT.EXE. LF.EXE inputs a file "lf.in" and outputs to a file "lf.out". LFEDIT.EXE is an interactive editor, but can only handle files of about 48K. Use LF.EXE to test SPHERE.LF (after copying it to "lf.in") or other lf files larger than 48K.

6.1 multiply.lf

The language is best learned by examining an example. The file multiply.lf contains a simple experiment where we are trying to teach the system the multiplication table. The program is divided into a "tree" section which describes the tree and the length of training, and a "function" section which describes the data to be learned. Comments are started with a `#' mark and continue to the end of the line.

```
# A comment.
tree
        size = 4000
        min correct  = 144
        max epochs  = 20
```

The tree and function sections can be in any order, in this particular example the tree is described first. Apart from comments, tabs and newlines are not significant; the arrangement chosen above is only for readability. The first line after tree tells the system how large the atree is going to be. In this case we are choosing a tree with 4000 leaves (3999 nodes). We are going to train it until it gets 144 correct from the training set, or for 20 complete presentations of the training set, whichever comes first.

Trees may also be read from a file with the "load tree from" statement. If this statement is specified, the tree size will be ignored and lf will output a warning message. Trees can be written to files using either the "save tree to" or "save folded tree to" statements.

The statements in the tree section may be in any order.

```
function
        domain dimension = 2
        coding = 32:12 32:12 32:7
        quantization = 12 12 144
        training set size = 144
        training set =


1       1       1
1       2       2
1       3       3
1       4       4
....
```

test set size = 144
test set =

1 1 1
1 2 2
1 3 3
1 4 4
....

The training set *must* start with a dimension statement which gives the number of columns in the function table. The domain dimension refers to the number of *input* columns. Lf supports training of multiple functions using the same inputs. This is done using the codomain dimension statement. If the codomain dimension statement is not specified, the number of codomains is assumed to be 1 (as in the above example). The total number of columns in the training and test sets must equal the sum of the domain and codomain dimensions (this doesn't mean a restriction on the format, just on what the number of elements in the table must be). In the above example, we are defining a problem with three columns: two input and one output.

The other statements may come in any order; note however that the definition of the training set size must be defined before the training set. This also applies to the test set definition.

The coding statement defines is a series of **<width>:<step>** definitions, one for each column. The **<width>** is the number of bits in the bit vector for that column, the **<step>** is the step size of the walk in Hamming space that defines the encoding of this column. Because a tree only produces a single bit in response to an input vector, the **<width>** of the codomain columns (which come after the domain columns) actually defines how many trees will be learning output bits of this function.

The quantization statement defines for each column the total number of coded bit vectors for that column. Entries in the test and training sets are encoded into the nearest step, so this statement defines the accuracy
possible.

Codings may also be read from a file using the "loading code from" statement. If this is specified, coding and quantization statements are ignored, and lf will warn the user. Note that codings *must* be specified by a "read coding from" statement, or combinations of coding and quantization statements.

Codings can be saved to a file with the "save coding to" statement, which may be placed anywhere in the function section.

The training set statement defines the actual function to be learned by the system. An entry in a table can be either a real number or an integer. If the width of the a column (as specified by the coding) is 1, then that column is boolean. For boolean columns, zero values are FALSE, and any non-zero value is considered TRUE.

The test set statement defines the test that is run on the trees at the end of training to see how well

the learned function performs. Like the training set, reals or integers are acceptable.

After lf has executed, it produces a table of output showing how each element in the test set was quantized, and the value the trained tree returned. Consider the following results that **multiply.lf** produced. Note that the quantization level is one less than the number represented. This is because the range of numbers is from 1 to 144, and 0 corresponds to the first quantization level.

```
1
.....
3.000000 2
33.000000 32
3.000000 2
36.000000 35
4.000000 3
4.000000 3
4.000000 3
8.000000 7
4.000000 3
12.000000 11
.....
```

Each column consists of two numbers, the entry specified by the user, and an integer describing the quantization level it was coded into.

The fourth column is the result produced by the trained tree. It shows the quantization level produced (the second figure) and how this may be interpreted in the space of the codomain (the first figure).

6.2 sphere.lf

This lf example uses a spherical harmonic function Y2 defined by:

$$\begin{aligned} Y2(m, f) = \; & A0(3m2 - 1/2) \\ & + 3m(1 - m2)1/2 + [A1 \cos f + B1 \sin f] \\ & + 3(1 - m2)\,[A2 \cos 2f + B2 \sin 2f] \end{aligned}$$

where $A0 = 1.0$, $A1 = 0.4$, $B1 = 0.9$, $A2 = 2.4$, $B2 = 7.9$. The values of m were in the interval [0.0, 1.0], and the values of f were in [0.0, p]. The values of $Y2$ range between -26.0 and 26.0.

The m and f intervals were quantized into 100 levels each; the random walks had 64 bits and a stepsize of 3. The $Y2$ values were quantized into 100 levels, the random walk having 64 bits with a stepsize of 3. Training 64 networks of 8191 elements on 1000 samples resulted in a function which, during test on 1000 new samples, was decoded to the correct quantization level, plus or minus three, 88.6% of the time. The error in the quantized result was no more than nine quantization levels for all of the test samples. (A slightly better learning algorithm got within three levels 95.8% of the time, and was always within eight levels.)

The function section introduces the optional "largest" and "smallest" statements. These may be used if the user needs to explicitly define the largest and smallest values in the test and training sets. If they are missing, lf will just use the largest and smallest values for each column in both the test and training sets.

This problem takes about 80 minutes of CPU time on a Sun Sparcstation 1. We have included a sample set of results in the file sphere.out.

5.3 The Syntax of lf

The syntax has been defined using YACC. Tokens have been written in quotes to distinguish them. Note that the following tokens are synonyms :-

dimension, dimensions
max, maximum
min, minimum

The syntax is defined as follows :-

program :

function_spec :

dim :

codim:

function_statements :

| function_statements function_statement

function_statement :

| coding

| coding_io

| train_table_size

| train_table

| test_table_size

| test_table

| largest

| smallest

quantization :

quant_list :

coding :

code_list :

coding_io:

train_table_size :

train_table :

test_table_size :

test_table :

table :

num :

largest :

largest_list :

smallest :

smallest_list :

tree_spec :

tree_statements :

tree_statement :

tree_size :

tree_io:

max_correct :

max_epochs :

7 Other Demonstrations

In this section we briefly present some boolean function problems which atrees have solved.

7.1 The Multiplexor Problem

A multiplexor is a digital logic circuit which behaves as follows: there are k input leads called control leads, and 2k leads called the "other" input leads. If the input signals on the k 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 jth one of the other leads (in some fixed order). A multiplexor is thus a boolean function of $n = k + 2k$
variables and is often referred to as an n-multiplexor.

Here is a program to define a multiplexor with three control leads, v[2], v[1] and v[0], the fact that they are these particular variables being irrelevant due to randomization in the programs:

/* Windows window procedure and initialization omitted for clarity */

/* An eleven input multiplexor function test */

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "atree.h"

#define TRAINSETSIZE 2000
#define WIDTH 11
#define TESTSETSIZE 1000
```

```
#define TREESIZE 2047

char multiplexor(v)

    char *v;

{
     return(v[ v[2]*4 + v[1]*2 + v[0] + 3]);
}

main(hInstance)

HANDLE hInstance;

{
   int i;
   int j;
   LPBIT_VEC training_set;
   LPBIT_VEC icres;
   LPBIT_VEC test;
   char vec[WIDTH];
   char ui[1];
   int correct = 0;
   LPATREE tree;
   char szBuffer[80];

   /* Initialise */

   training_set = (LPBIT_VEC) Malloc (TRAINSETSIZE * sizeof(bit_vec));
   MEMCHECK(training_set);

   icres = (LPBIT_VEC) Malloc (TRAINSETSIZE * sizeof(bit_vec));
   MEMCHECK(icres);

   atree_init(hInstance);

   /* Create the test data */

   MessageBox(NULL, "Creating training data", "Multiplexor", MB_OK);

   for (i = 0; i < TRAINSETSIZE; i++)
   {
      for (j = 0; j < WIDTH; j++)
      {
         vec[j] = RANDOM(2);
      }
      training_set[i] = *(bv_pack(vec,WIDTH));
```

```
        ui[0] = multiplexor(vec);
        icres[i] = *(bv_pack(ui,1));
    }

    /* Create a tree and train it */

    MessageBox(NULL,"Training tree", "Multiplexor", MB_OK);

    tree = atree_create(WIDTH,TREESIZE);
    (void) atree_train(tree,training_set,icres,0,TRAINSETSIZE,
                    TRAINSETSIZE-1,100,1);

    /* Test the trained tree */

    MessageBox(NULL,"Testing the tree", "Multiplexor", MB_OK);

    for (i = 0; i < TESTSETSIZE; i++)
    {
        for (j = 0; j < WIDTH; j++)
        {
            vec[j] = RANDOM(2);
        }
        test = bv_pack(vec,WIDTH);
        if (atree_eval(tree,test) == multiplexor(vec))
        {
            correct++;
        }
        bv_free(test);
    }

    wsprintf(szBuff,"%d correct out of %d in final test",correct,TESTSETSIZE);

    /* discard training set */
    for (i = 0; i < TESTSETSIZE; i++)
        {
        Free(training_set[i].bv);
        Free(icres[i].bv);
        }

    Free(training_set);
    Free(icres);

    /* Discard tree */
    atree_free(tree);

    return;
}
```

This problem was solved to produce a circuit testing correctly on 99.4% of 1000 test vectors in 19 epochs, or about 530 seconds on a Sun 3/50. The time may vary considerably depending on the random numbers used. It is possible to learn multiplexors with twenty inputs (four control leads) with a straightforward but improved adaptation procedure, and multiplexors with up to 521 leads (nine of them control leads) using much more elaborate procedures which change the tree structure during learning [Arms5].

7.2 The Mosquito Problem

Suppose we are conducting medical research on malaria, and we don't know yet that malaria is caused by the bite of an anopheles mosquito, unless the person is taking quinine (in Gin and Tonics, say) or has sickle-cell anaemia. We are inquiring into eighty boolean-valued factors of which "bitten by anopheles mosquito", "drinks Gin and Tonics", and "has sickle-cell anaemia" are just three. For each of 500 persons in the sample, we also determine whether or not the person has malaria, represented by another boolean value, and we train a network on that data. We then test the learned function to see if it can predict, for a separately-chosen test set, whether person whose data were not used in training has malaria.

Suppose on the test set, the result is 100% correct. (Training and test can be done in about five seconds on a Sun Sparcstation 1.) Then it would be reasonable to analyze the function produced by the tree, and note all the variables among the eighty that are not involved in producing the result. A complete data analysis system would have means of eliminating subtrees "cut off" by LEFT or RIGHT functions (such as **atree_compress()**), to produce a simple function which would help the researcher understand some factors important for the presence of the disease. If there were extraneous variables still left in the function in one trial, perhaps they would not show up in a second trial, so that one could see what variables are consistently important in drawing conclusions about malaria.

We apologize for the simplistic example, however we feel the technique of data analysis using these trees may be successful in cases where there are complex interactions among features which tend to mask the true aetiology of the disease.

The code for the problem can be found in mosquito.c.

8 References

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

[Arms2] W. W. Armstrong, Adaptive Boolean Logic Element, U. S. Patent 3,934,231, Jan. 20, 1976, assigned to Dendronic Decisions Limited.

[Arms3] 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.

[Arms4] 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.

[Arms5] W. W. Armstrong, Jiandong Liang, Dekang Lin, Scott Reynolds, Experiments using Parsimonious Adaptive Logic, Technical Report TR 90-30, Department of Computing Science, University of Alberta, September 1990.

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

[Hech] Robert Hecht-Nielsen, Neurocomputing, Addison-Wesley, 1990.

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

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

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