An Implementation of Adaptive Logic Networks

©Andrew Dwelly

November 11, 1990

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 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.

2 Writing applications with atree

Writing applications that perform a simple classification (yes or no) is relatively easy. 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`, `mult.c`, and `parity.c`.

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:-

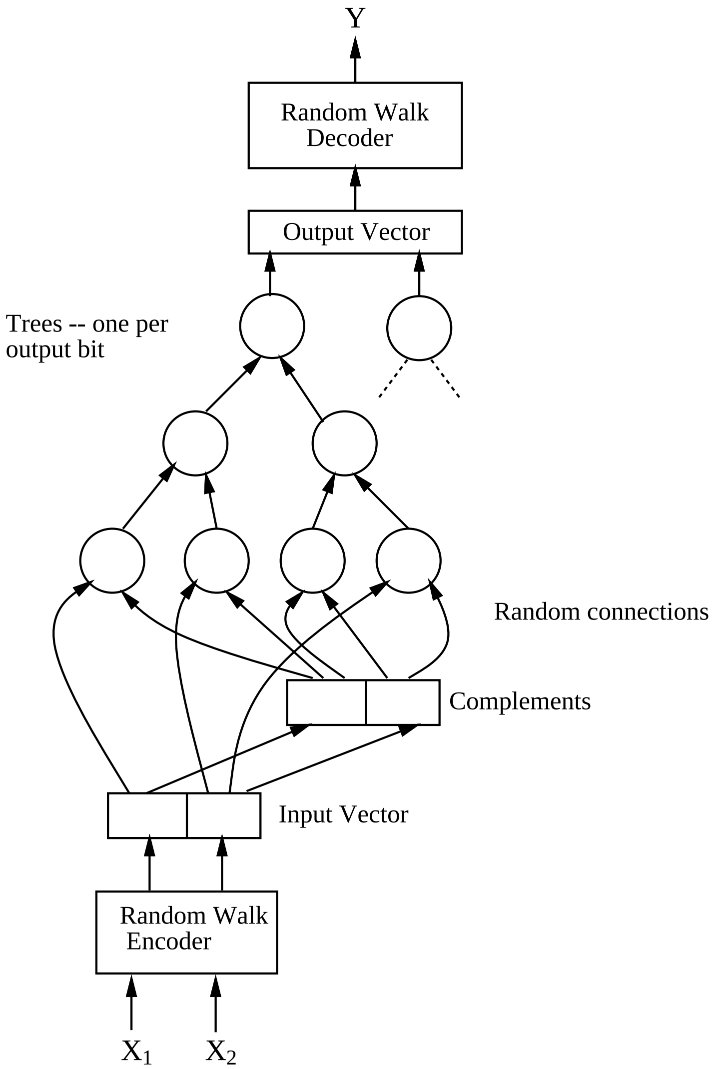

Figure 1: Using several trees to compute $Y = f(X_1, X_2)$

```
....
{
    int i;
    int j;
    bit_vec *train;
    bit_vec *result;
    atree **forest;

    /* Create the training set */

    train = domain();

    /* Create the result set */

    result = codomain();

    /*
     * Make enough room for the set of trees - one tree per bit in the
     * codomain
     */

    forest = (atree **) malloc((unsigned)sizeof(atree *) * 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 an interval of 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 quantizations 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, the 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;
    bit_vec *test;
    bit_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 = atree_bv_create(No_OF_TREES);

    /* Evaluate the trees */

    for (i = 0; i < NO_OF_TREES; i++)
    {
        /*
         * Insert into bit i of tree_result, the result of evaluating
         * the ith tree.
         */

        atree_bv_insert(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 = atree_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 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 file of routines `atree.c`. The routines permit the creation, training, evaluation and testing of adaptive logic networks, and there are a number of utility routines designed to make this task easier.

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 atree_bv_something() instead.

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

Macros are always in upper case.

3.2 The main data structures

The two main data structures are atree and bit_vec which represent atrees and bit vectors respectively. They are both fully specified in the file atree.h.

Examining the structure atree first, we find a recursive structure which represents a single node in an atree. Since it is binary, it consists of the data which the node holds, and two pointers to child nodes. If the subnode is a leaf of the tree it will contain a bit number rather than a pointer, so both left and right are unions of pointers and integers to represent these two possibilities.

The internal data of the node consists of a series of boolean flags (represented by chars here), an integer which describes the function of the node (AND, OR, LEFT, or RIGHT), two integers for signals from the child nodes, and two counters cnt_10 and cnt_01 used during training.

The left_leaf_flag (and/or the right_leaf_flag) is set to true if the left (or respectively, right) input of the node is a leaf node, which represents a “lead” to be connected to a bit of the input vector or its complement, rather than to an adaptive element. The flags left_cmp_flag and right_cmp_flag are only meaningful if the respective leaf_flag is true. They represent the fact that the value of the bit the leaf accesses is complemented before it is input into the tree.

3.3 Manifest Constants

All the constants defined are internal to the atree.c library and are declared at the top of this file.

Constants AND, OR, LEFT, RIGHT represent the current function of a node. TRUE and FALSE are used to represent boolean values both in program flags and also in the tree itself. We also include UNEVALUATED and ERROR in the boolean type. These complete the type lattice for booleans, but more importantly allow us to signal that a subtree has not been evaluated during training, or is in error.

The constant MAX_RETRY is used in the random walk routines to control the maximum amount of searching for a bit vector than has not previously been encountered.

The constants MAXSET,ABOVEMID,BELOWMID, and MINSET are used during the initialisation of a tree. The operation of a node in the tree is determined by the value of its counters, the integer atree.function in a node is determined by them. The above constants represent the full range of a counter and also mark its middle values.

3.4 Macros

The next section in `atree.c` is a series of macros used throughout the program. The macro `Printf` serves no purpose other than to avoid typing "`(void)`" in front of every `printf` statement.

The terms `public` and `private` are used to denote whether a routine is for use outside the package (public) or is strictly internal (private). They take advantage of the fact that in C, a routine that is declared as static may not be accessed outside its source file.

The macro `EVER` is a standard C trick used to describe an infinite loop; typically it will be used with a for as in `for EVER`.

In order to print out a boolean value (in our raised definition of booleans) we define the macro `PRINTBOOL`.

The macro RANDOM serves two purposes. Firstly, it overcomes the rather poor statistical behavior of the standard `rand()` routine by masking off the lower bits, which tend to cycle (see rand(3) in the Unix manual for further details). Secondly, it 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 `VERBOSE` is used to control the amount of diagnostic information produced by a program. The variable `verbosity` is set to a particular level, and if the statement s is associated with that or a lower level, it is executed. Typically we will see the following sort of usage `VERBOSITY(1,Printf("something"));` `something` will be produced if the verbosity level is greater or equal to 1.

The macro BYTE gives the size (in bits) of a byte on this machine.

The macros `EVAL`,`LEFTEVAL`,and `RIGHTEVAL` are used during tree evaluation. They will be explained in detail in the routines that call them.

3.5 `public void atree_init()`

This is the first of the routines in the code, and should be called by the user before making calls to any other routine. Currently, it merely calls the `srand()` routine to initialise the random number generator, but it may do more in future versions.

3.6 `public bit_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 [1] 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.

3.7 public atree *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 input 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:

$$\texttt{leaves} \geq 2 * \texttt{numvars}.$$

[1] The weight of a bit vector is the number of 1s in it; e. g., the weight of 10110 is 3.

The routine itself proceeds by deciding which bit of the input vector is to be connected to each leaf, and stores the information in two arrays `connection` which holds the bit numbers, and `complemented` which shows whether the connection is complemented or not. It then calls a private recursive tree-building routine `build_tree`. The latter routine depends on having enough space already allocated on the heap and `atree_create` is responsible for that.

3.8 public bool atree_eval(tree,vec)

```
atree *tree;
bit_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 [2] fashion [Meis][Arms5]. The Macro `LEFTEVAL` is responsible for evaluating the left subtree and `RIGHTEVAL` is responsible for the right subtree. They both use the $\hat{\mathrm{E}}$VAL macro which is a little complex since it has to check whether or not a node is a leaf and is connected to the input bit-vector, and if it is, whether the value is to be inverted or not.

This routine also marks subtrees that are unevaluated, and sets the internal `atree.sig_left` and `atree.sig_right` values for a node. This information is used when `atree_eval` is used from within `atree_train`.

3.9 public bool atree_train(tree,tset,...)

```
atree *tree
bit_vec *tset;
bit_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

[2] I 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.).

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 during an epoch 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 presented during an epoch 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 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. The process of training is done by the private `train()` routine.

3.10 `public void atree_print(tree,verbosity)`

```
atree *tree;
int verbosity;
```

This routine allows the programmer to print out an `atree`, before, during, or after training. The parameter `tree` is the tree to be printed, and `verbosity` is the amount of information produced.

The routine makes an immediate call to the private `print_tree` routine.

3.11 `public bit_vec *atree_bv_create(length)`

```
int length;
```

Creates a vector of `length` bits, where each bit is initialised to 0.

3.12 public bit_vec *atree_bv_pack(unpacked,length)

```
char *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 pointer to a `bit_vec`.

3.13 public int atree_bv_diff(v1,v2)

```
bit_vec *v1;
bit_vec *v2;
```

This routine calculates the Hamming distance between `v1` and `v2`, i. e. `weight(v1` $\bigoplus$ `v2)` where weight is the number of one bits in a vector and $\bigoplus$ 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!).

3.14 public bit_vec *atree_bv_concat(n,vectors)

```
int n;
bit_vec **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 `bit_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.

3.15 public void atree_bv_print(vector)

```
bit_vec *vector;
```

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

3.16 public void atree_bv_insert(n,vec,bit)

```
int n;
bit_vec *vec;
bool bit;
```

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

3.17 public int atree_bv_extract(n,vec)

```
int n;
bit_vec *vec;
```

This routine returns the value of the `n`th bit (0 to `bit_vec.len - 1`) in the bit vector `vec`. The rather unpleasant expression works as follows :-

The parameter `n` is divided by eight to get the number of the byte where the bit is held. This number is added to the first byte to get the actual address of the byte concerned.

The remainder of the division `n % BYTE` is used to find where in the byte, the bit is. A mask is shifted left this number of times and logically and-ed with the byte. If the result is 0 the bit was 0. If the result is greater than one, the bit was 1. The test for equality to zero forces the result to be just 1 or 0.

3.18 public int atree_bv_equal(v1,v2)

```
bit_vec *v1;
bit_vec *v2;
```

This routine tests two bit vectors for equality.

3.19 public void atree_bv_free(vector)

```
bit_vec *vector;
```

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

3.20 private atree *build_tree(connection,....)

```
int *connection;
bool *complemented;
int start;
int end;
atree *tree;
```

This is the recursive routine that does most of the work when creating an atree. It has an array of leaf-to-bit-vector connections (`connection`) and another array telling it which of the connections are to be inverted (`complemented`). It has a `start` and an `end` parameter which mark the part of the `connection` array this subtree is connected to.

The routine allocates a single node, and then makes a decision based on the size of the remaining subtree (which it knows from the `start` and `end` parameters. The simplest case is when there is a difference of four or more between `start` and `end`. Under these circumstances the routine recursively calls itself to allocate the left and right subtrees. The left subtree is allocated immediately after the current node, and the right subtree is allocated after the left subtree. It is able to do this since `build_tree()` always returns the next available space for node allocation.

If the difference between `start` and `end` is less than four, the routine allocates some leaves using the information in the `connection` and `complemented` arrays to specify how the input vector is to be accessed.

3.21 private void print_tree(tree,indent,verbosity)

```
atree *tree;
int indent;
int verbosity;
```

This routine does most of the work of printing out trees. It recursively calls itself with a larger `indent` to print out the rightmost subtree, then it prints out information about the current node, then recursively calls itself again to print out the left subtree. This particular order was chosen so that if the printout is tipped onto its side, it will resemble the usual diagram of a tree.

The `verbosity` can currently be set to 0 (tree structure) or 1 (signal values).

3.22 private bool train(tree,vec,result)

```
atree *tree;
bit_vec *vec;
bool result;
```

This routine trains a particular `tree` to return the given `result` when it sees the given bit vector `vec`. It does this by working out the current response of the tree to the vector using `atree_eval()` then adapting the tree using the private `adapt` routine.

3.23 private void adapt(tree,vec,result)

```
atree *tree;
bit_vec *vec;
```

`bool result;`

This routine contains the heuristic responsibility algorithm. We define two macros which are to be used locally, `INCR` and `DECR`, they are used to increment and decrement the `atree.cnt_10` and `atree.cnt_01` counters which are bounded by the constants `MAXSET` and `MINSET`.

The adaptation algorithm works its way recursively down the tree changing the counters on nodes that are determined to be "heuristically responsible". If a node is determined to be heuristically responsible, the algorithm requires the evaluation of unevaluated subtrees to proceed — this may not have been completed by the evaluation stage of `train()` because of parsimony [3]. So the left and right subtrees are evaluated at this stage if necessary.

The next stage is to update one of the counters; if the left subtree has the value 1 and the right subtree has value 0, then the `cnt_10` counter is the one changed. If the desired result of the tree is 1, it is incremented, otherwise it is decremented. The other counter `cnt_01` is left unchanged, since it only changes when the left subtree is 0 and the right subtree is 1. The function value of the node may now have changed, so this is recomputed.

Finally, a fairly complex condition is used to decide whether either the left or right subtree, or both, are heuristically responsible and thus should be adapted in turn. The source code is the most concise definition of this and it is recommended that you examine the code directly. The heuristics used are intended to solve the "credit assignment problem", i. e. they determine which nodes are responsible for the correct or incorrect result of the tree. The success of these heuristics depends a lot on fact that the allowed node functions are monotonic. Finding good heuristics for assigning responsibility is the most difficult question in connection with using adaptive logic networks.

3.24 private int node_function(tree)

`atree *tree;`

This routine determines the action of a node `AND`, `OR`, `LEFT`, `RIGHT` from the value of its counters.

3.25 error()

This is a last ditch routine called by `atree_rand_walk()` if it can not proceed further.

[3] laziness!

4 The language lf

The second major product included in the current release is the "lf" language, that allows a non-programmer to experiment with this 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.

A typical session might be :-

```
% lf lf_multiply | histogram
```

Where "lf_multiply" is an lf program specifying some experiment. If no file is specified, lf assumes it is reading from the standard input, so we could also do the following :-

```
% cat lf_multiply | lf | histogram
```

4.1 `lf_multiply`

The language is best learned by examining an example. The file **`lf_multiply`** 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  = 143
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 tree 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 143 correct from the training set, or for 20 complete presentations of the training set, whichever comes first.

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. Currently the codomain size is restricted to one so we are defining a problem with 3 columns — 2 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 column (the last one) 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.

If a particular column is has a coding entry `1:1`. It is treated as a special case, a boolean column. Only values of `0` representing false, and `1` representing true make any sense in this column (although this is not currently checked).

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 at the moment. Boolean values are special cases of integers.

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 test 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 **lf_multiply** produced.

```
.....
3.000000 2 11.000000 10 33.000000 32 32.777779 32
3.000000 2 12.000000 11 36.000000 35 35.756945 35
4.000000 3 1.000000 0 4.000000 3 3.979167 3
4.000000 3 2.000000 1 8.000000 7 7.951389 7
4.000000 3 3.000000 2 12.000000 11 11.923611 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).

4.2 lf_sphere

This lf example uses a spherical harmonic function Y_2 defined by:

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

where $A_0 = 1.0$, $A_1 = 0.4$, $B_1 = 0.9$, $A_2 = 2.4$, $B_2 = 7.9$. The values of μ were in the interval [0.0, 1.0], and the values of ϕ were in [0.0, π]. The values of Y_2 range 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 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_res`

4.3 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 tree_spec
        | tree_spec function_spec

function_spec : "function" dimension function_statements

dimension : "domain dimension =" integer

function_statements : function_statement
    | function_statements function_statement

function_statement : quantization
   | coding
   | train_table_size
   | train_table
   | test_table_size
   | test_table
   | largest
   | smallest

quantization : "quantization =" quant_list

quant_list : integer
   | quant_list  integer

coding : "coding =" code_list

code_list : integer ":" integer
  | code_list integer ":" integer
```

```
train_table_size : "training set size =" integer

train_table : "training set =" table

test_table_size : "test set size =" integer

test_table  : "test set =" table

table : num
      | table num

num : real
    | integer

largest : "largest =" largest_list

largest_list : num
             | largest_list num

smallest : "smallest =" smallest_list

smallest_list : num
      | smallest_list num

tree_spec : "tree" tree_statements

tree_statements : tree_statement
| tree_statements tree_statement

tree_statement : tree_size
       | max_correct
       | max_epochs

tree_size : "size =" integer

max_correct : "min correct =" integer

max_epochs : "max epochs =" integer
```

5 Other demonstrations

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

5.1 The multiplexer problem

A multiplexer is a digital logic circuit which behaves as follows: there are k input leads called control leads, and 2^k 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 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 often referred to as an n-multiplexer.

Here is a program to define a multiplexer 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:

```
/* An eleven input multiplexer function test */

#include <stdio.h>
#include "atree.h"

#define TRAINSETSIZE 2000
#define WIDTH 11
#define TESTSETSIZE 1000
#define RANDOM(b) ((rand() >> 15) % (b))
#define TREESIZE 2047

char multiplexer(v)

     char *v;

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

main()
{
    int i;
    int j;
    bit_vec training_set[TRAINSETSIZE];
    bit_vec icres[TRAINSETSIZE];
    bit_vec *test;
    char vec[WIDTH];
    char ui[1];
    int correct = 0;
    atree *tree;
```

```
/* Initialise */

atree_init();

/* Create the test data */

printf("Creating training data\n");

for (i = 0; i < TRAINSETSIZE; i++)
{
    for (j = 0; j < WIDTH; j++)
    {
        vec[j] = RANDOM(2);
    }
    training_set[i] = *(atree_bv_pack(vec,WIDTH));
    ui[0] = multiplexer(vec);
    icres[i] = *(atree_bv_pack(ui,1));
}

/* Create a tree and train it */

printf("Training tree\n");

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

/* Test the trained tree */

printf("Testing the tree\n");

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

```
    printf("%d correct out of %d in final test\n",correct,TESTSETSIZE);

    exit(0);
}
```

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 multiplexers with twenty inputs (four control leads) with a straightforward but improved adaptation procedure, and multiplexers with up to 521 leads (nine of them control leads) using much more elaborate procedures which change the tree structure during learning [Arms5].

5.2 The parity problem

The parity function on n boolean variables $X_1, \ldots, X_n$ has value ONE if the number of those variables having value ONE is even. Otherwise it has value ZERO. It is just the sum modulo 2 of the variables. This function is very sensitive to change of its inputs, and so extrapolation of values of the function to neighboring input vectors is not an appropriate requirement. However, we can test the ability of the trees to learn the function. The following code learns the parity function of six variables using the library functions.

```
/* A seven input parity function test */

#include <stdio.h>
#include "atree.h"

#define TRAINSETSIZE 2000
#define WIDTH 7
#define TESTSETSIZE 1000
#define RANDOM(b) ((rand() >> 15) % (b))
#define TREESIZE 8191

char parity_function(v)

char *v;
{
int j, temp;

  temp = 0;
  for(j = 0; j <WIDTH; j++){
```

```
    temp += (int) v[j];
  }
    return(temp%2);
}

main()
{
        ...as above, with the parity function...
}
```

This is compiled using cc -o parity -O parity.c atree.c, and we must have the include file atree.h available.

We have used the research version of the atree code to learn the parity function of seven variables in 97 epochs.

5.3 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 analyse 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, 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.

6 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.

7 Acknowledgements

I'd like to thank Bill Armstrong for going through this document and correcting it. His mark can be seen on some of my footnotes. Alas, there are bound to be various errors and omissions still present, and for these, I apologize in advance.