atree 2.7

Adaptive Logic Network testing and development package

Introduction
ALN Demo Programs
ALN Script Language : lf
ALN Software Development Library
ALN Technical Notes
ALN Bibliography

Introduction to ALNs

Human abilities to learn and to quickly identify patterns are exactly the abilities ALNs are designed to automate, and they do it very well. Factory automation, speech recognition, data analysis, and even the creative processes of abstract thought require these abilities.   Any tasks like these that require an ability to learn to respond to real or abstract patterns at very high speed are potential areas of application of ALN-based systems.

Our goal in what follows will be to try to familiarize the reader with ALN technology and outline what applications are good candidates for using ALN technology.

The Advantage of Extremely High Speed
What Kinds of Tasks Does an ALN Do Well?
How Does an ALN Overcome Noise?
Why Does an ALN Reduce the Effects of Distortion?
Training ALNs
Data Analysis Applications of ALNs
Safety of Artificial Neural Systems
Eliminating Redundant Inputs
The Future of Training versus Programming
An Invitation to Developers

ALN Demonstration Programs

There are three demonstration programs included with atree 2.7.   The demos all show how <u>ALN</u>s can distinguish patterns under seemingly insurmountable noise conditions based on only a fraction of the possible cases.   They illustrate the ability of <u>ALN</u>s to generalize from training samples to other inputs close to them.   This is not like the human ability to generalize from playing the flute to playing the tuba, which is not based on a measure of proximity, or at least not in a low-level sense.   <u>ALN</u>s, like other neural networks, have limitations that require combination of their talents with higher-level paradigms, for example expert systems, predicate logic, or other "symbolic" AI tools.

<u>The Multiplexor Problem</u>
<u>The Mosquito Problem</u>
<u>The OCR Problem</u>

ALN Script Language: lf

The "lf" language interpreter allows a non- programmer to experiment with <u>ALN</u>s. 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 errors or plots of the functions.

LFEDIT.EXE is an interactive file editor that lets you edit lf programs and then run them, but can only edit files of about 48K.   If your lf file is larger than this, LFEDIT.EXE will offer you the choice of executing the program anyway, without displaying it in the edit window.

IMPORTANT
There are 4 example files, *example1.lf*, *example2.lf*, *example3.lf*, and *example4.lf*.   Please work through them carefully, as they illustrate all the different language features available, as well as some techniques for training <u>ALN</u>s.

<u>Using lf: multiply.lf</u>
<u>lf Language Syntax</u>
<u>lf Example: abs_sine.lf</u>
<u>lf Examples: noisyand.lf & noisyxor.lf</u>

ALN Software Development Library

Writing atree applications is not difficult.   In the archive atre26.exe you can find source code for examples of atree applications, both simple and complex, both in C and C++.   All make use of the atree library, which deals with creating, training, and evaluating ALNs.   The BV library permits easy creation and use of the bit vectors that are used for input to the ALNs.   The Windows support library contains routines that facilitate the running of atree applications under Windows 3.x.

Writing Applications with atree 2.7
Data Structures
The Windows atree Library
The BV Library
The Windows Support Library

ALN Technical Notes

The software in the archive `atre26.exe` 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].

ALN Bibliography

Recommended Publications on Adaptive Logic Networks Based on Heuristic Responsibility

W. Armstrong, Adaptive Boolean Logic Element, U. S. Patent 3934231, Feb. 28, 1974 (multiple filings in various countries), assigned to Dendronic Decisions Limited.

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

W. Armstrong, Gilles Godbout: Use of Boolean Tree Functions to Perform High-Speed Pattern Classification and Related Tasks, Dept. d'IRO, Universite de Montreal, Doc. de Travail #53, 1974. (unpublished, except in summary form as follows:)

W. Armstrong and G. Godbout, "Properties of Binary Trees of Flexible Elements Useful in Pattern Recognition", IEEE 1975 International Conf. on Cybernetics and Society, San Francisco, 1975, IEEE Cat. No. 75 CHO 997-7 SMC, pp. 447-449.

W. Armstrong and J. Gecsei, "Architecture of a Tree-based Image Processor", 12th Asilomar Conf. on Circuits, Systems and Computers, Pacific Grove, Calif., 1978, pp. 345-349.

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

W. Armstrong, J.-D. Liang, D. Lin, S. Reynolds, Experiments Using Parsimonious Adaptive Logic, Tech. Rept. TR 90-30, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, T6G 2H1.   This is now available in a revised form via anonymous FTP from Menaik.cs.ualberta.ca [129.128.4.241] in pub/atree2.ps.Z (the title of the revised document is "Some Results concerning Adaptive Logic Networks").

W. Armstrong, A. Dwelly, J.-D. Liang, D. Lin, S. Reynolds, Learning and Generalization in Adaptive Logic Networks, in Artificial Neural Networks, Proceedings of the 1991 International Conference on Artificial Neural Networks (ICANN'91), Espoo, Finland, June 24-28, 1991, T. Kohonen, K.Makisara, O. Simula, J. Kangas eds. Elsevier Science Publishing Co. Inc. N. Y. 1991, vol. 2, pp. 1173-1176.

Aleksandar Kostov, Richard B. Stein, William W. Armstrong, Monroe Thomas, Evaluation of Adaptive Logic Networks for Control of Walking in Paralyzed Patients, 14th Ann. Int'l Conf. IEEE Engineering in Medicine and Biology Society, Paris, France, Oct. 29 - Nov. 1, 1992 (to appear).

Allen G. Supynuk, William W. Armstrong, Adaptive Logic Networks and Robot Control, Proc. Vision Interface Conference '92, also called AI/VI/GI '92, Vancouver B. C., May 11-15, 1992, pp. 181 - 186.

R. B. Stein, A. Kostov, M. Belanger, W. W. Armstrong and D. B. Popovic, Methods to Control Functional Electrical Stimulation in Walking, First International FES Symposium, Sendai, Japan, July 23 - 25, 1992 (Invited paper, to appear).

In case you have difficulty in obtaining the above documents or the atree release 2.0 software for Unix or release 2.7 for IBM-PC and compatibles under Windows, the software and all of the above documents prior to 1991 can be obtained from the University of Alberta for a media fee to cover the costs of copying and mailing of $150 (Canadian), made payable to the University of Alberta.   Two 3 1/2" diskettes are normally included but an attempt will be made to satisfy needs for other media, e.g. tapes.   Orders can be sent c/o Professor W. W.   Armstrong, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, T6G 2H1.

The Advantage of Extremely High Speed

OCR is nothing new, nor is it new that character shapes can be learned by a computer system.   One important new aspect of ALN technology is the potential to recognize things at extremely high speeds.   ALNs can be made orders of magnitude faster than any technology we know of today with competitive flexibility and noise immunity.

ALNs, after training, become hardware circuits that are just collections of transistors, the basic switching component of all digital computers, connected together in the form of a tree.   In such a tree, the computation time is proportional to the number of layers of the tree. Complex tasks can be accomplished in nanoseconds. (For the uninitiated: a nanosecond is a thousandth of a thousandth of a thousandth of a second.)   Another way of putting this is in terms of the speed of light. Most people know how fast light is: it can circle the earth seven times in one second.   Hardware resulting from ALN training, can process an input pattern and compute its identity in the time it takes for light just to cross a small room.

In contrast to ALNs, other computer systems, including "massively parallel" artificial neural nets of other types, can barely get started on the same recognition task in that time.   They might be able to do one multiplication, or several simultaneously in different components, before the ALNs are finished, but they will still have many more to do before recognition of the pattern is complete.

ALNs are the system of choice in all areas requiring very high speed.

One example: researchers at an accelerator facility need a trigger which will operate in less than half a microsecond when particles of a certain type are detected.   They are looking seriously at ALN technology because nothing else appears to have the potential for such speed.   It appears now that ALNs have the potential to exceed the speed and accuracy requirements of that problem.

For those who don't need to identify the type of a particle traveling near the speed of light, there may be other very good reasons for requiring extreme speed.   One is that your system must do huge numbers of pattern recognition operations.   For example it might have to process scanned images of parts being assembled in a factory, where each image is made up of a million pixels.   Another possibility is that you want to analyze a large collection of images for a certain pattern.   Looking for a feature in an image in all possible positions and many sizes and orientations would be very time-consuming without ALNs.   A single ALN tree has the potential to carry out hundreds of millions of recognition operations per second.

Just as speed is a requirement for computer vision, it is also needed for sound signals, like voice, and for sensor signals in general.

In designing ALNs, we have consciously chosen the path for the highest possible speed. It can provably be made faster than any other digital system on a given problem.   This advantage of ALN hardware will never be overtaken as long as computing is done with digital circuitry (so we are probably safe for the next millennium anyway).   As digital logic becomes faster, ALNs will get faster at the same rate.

As problems get larger, the computation time of an ALN system does not grow proportionally to the size of the task, as it does with other artificial neural systems.   With ALNs, the computation time grows much more slowly: logarithmically in the size of the task. What other systems do you know of that can handle problems twice as large in less than one extra nanosecond?

Another important new aspect of ALN technology that we shall discuss is the inherent ability of ALNs to overcome distortions and noise in input patterns.   Other systems can do this too, but ALNs do it without a significant penalty in computation time.

What Kinds of Tasks Does an ALN Do Well?
Why Does an ALN Reduce the Effects of Distortion?
How Does an ALN Overcome Noise?

What Kinds of Tasks Does an ALN Do Well?

Extracting significant information from noisy, incomplete, even contradictory data, and thus making appropriate decisions is what ALN processing is all about.   Consider how important this ability is to you. When there are obstructions, or visibility is poor, it is not easy for you to recognize faces, house numbers, or highway signs, but your well-being and even your survival may depend on it.   During your waking hours, all of your senses are continuously generating large amounts of noisy data, while your brain and nervous system are at work identifying the origins of the stimuli and dealing with the significant observations.   ALNs allow similar tasks to be done automatically.   It doesn't matter whether the data is from an image scanner, a microphone or a relational database.   Successful applications of ALNs have used data from questionnaires, infrared spectra, pressure sensors, and elaborate detectors at particle accelerator facilities.

Why Does an ALN Reduce the Effects of Distortion?
How Does an ALN Overcome Noise?
Data Analysis Applications of ALNs
Eliminating Redundant Inputs
An Invitation to Developers

Why Does an ALN Reduce the Effects of Distortion?

ALN training results in a tree of AND and OR operators (logic gates).   This may later be turned into a tree of transistors in hardware, but let's just examine the tree of ANDs and ORs. The tree gets its inputs from a pattern of zeros and ones representing inputs from the problem domain.   Some of these signals are negated before being passed to the tree, since negations are required to make it possible to produce any desired relationship between the network inputs and the network outputs.   A gate in the tree takes several zero- or one-valued signals as inputs.   An OR gate produces a one output if ANY of its inputs is a one, and an AND gate produces a one if ALL its inputs are ones.

Obviously, the OR gates can allow the network to deal with alternative geometric forms of a character. The ANDs can combine parts of an image as required to form a whole image.   (Interchanging the roles of zero and one also interchanges the roles of AND and OR -- a principle called duality.)   So it is easy to understand how ALNs can cope with considerable distortions in what is considered to be an "A", say.

Some problems will not have this kind of variability in the patterns to be recognized, for example, suppose one wants to recognize whether there are exactly 200 dark pixels in the input grid of 1024 or not.   That is not a problem for ALNs, because there are so many alternative ways for making at least 200 pixels dark. (Though, surprisingly, ALNs can compute whether there are "roughly 200 dark pixels" quite easily.)

It is important to recognize when ALNs are not appropriate for a certain task.   Nothing could be more absurd than imagining they, or any other artificial neural system, will eliminate the need for programming and careful analysis of problems.

How Does an ALN Overcome Noise?
Safety of Artificial Neural Systems

How Does an ALN Overcome Noise?

Imagine that a tree of <u>AND</u>s and <u>OR</u>s responds correctly to one input pattern, say a handwritten "A". Imagine now perturbing that "A" by changing some light pixels to dark and vice-versa.   Assuming that the pattern still looks like an "A", why should the tree tend to still produce the same response as before?   To understand this we have to look at a gate, say a two-input <u>AND</u> gate, where the changed shape of the input character has led to an input to the <u>AND</u> gate being changed, say from a zero to a one.   What is the likelihood that the output of the <u>AND</u> gate will also be changed?   The answer is that if the other unperturbed input to the <u>AND</u> is a one, the output of the <u>AND</u> will change; but if the other input is a zero, the output won't change, but will still be zero. Hence the probability that the <u>AND</u> gate output changes is 0.5 overall.   In a ten layer tree, the probability that the output changes when one input to the tree changes is 1/1024.   This is very small.   In the case of the <u>OCR</u> demo, each pixel of the 32x32 input image can be sent to several gates of the tree (sometimes negated first).   The theory has been worked out for that case too, and it still shows   insensitivity to perturbations of the inputs.

Trees of <u>AND</u> and <u>OR</u> gates tend to be very insensitive to perturbations of the input images. Insensitivity is a property that is applicable to <u>ALN</u>s, but not to non-logical artificial neural systems in general.

<u>Training ALNs</u>
<u>Why Does an ALN Reduce the Effects of Distortion?</u>
<u>Safety of Artificial Neural Systems</u>

Training ALNs

Programming is tough to do.   Ask any programmer.   Even simple programs can cause problems when certain cases are overlooked.   For many problems, like handwritten character recognition, writing a program would be very difficult.   For example, you could start by analyzing which pixels in the character images are important for identifying the characters, then based on whether one such pixel is dark or light, you could branch to one of two possible continuations of the decision process where further branches are made, until finally a decision as to the character identity is made.   Because of noise, any particular pixel doesn't contain much information, so this decision-tree programming method approach would be long and tedious.

It would require you to have some idea of how to define the letters concerned.   How would you define the total variety of shapes of handwritten letters "A", "L", and "N"?   Providing such descriptions would be as hard as writing the program!

Hence, it makes sense to define the handwritten letter "A" by giving a large "training set" of samples, and to have a system which learns to respond to an input pattern on the basis of its similarity to one of the samples.   During training of an ideal system, contradictions in the identities of characters given in the training data would be sorted out, and a decision procedure would be developed that takes into account how close an input is to several close samples presented in training, and their identities.   Alas, then we would have the problem that the more precise we make the definition by giving more samples, the longer it would take to compare an input to all the samples.   More fundamentally, we still have to define "close", and make a procedure to resolve different identities that might be found among the close training samples.

When ALNs are trained, the information in the training set of samples is "compressed" into a digital logic circuit, so the problem of the time of comparison to training samples does not exist.   The decision time is always extremely fast.   On the other hand, there is no simple rule describing how ALNs operate, though there is a tendency to do something like the ideal procedure described above.   In fact, the raison-d'être of the OCR demo is to convince you that the method can work, even if it is not always easy or possible to explain exactly why. If you are working in a problem area where you have lots of empirical data but no simple models for making decisions, ALNs may be the method of choice.

Data Analysis Applications of ALNs
The Advantage of Extremely High Speed
Safety of Artificial Neural Systems

Data Analysis Applications of ALNs

We have successfully used ALNs to analyze environmental data, questionnaires, and data from attempts to design rules for stimulators that help spinal-cord damaged patients to walk.   Others are also using the data analysis capabilities of neural systems to try to predict fluctuations of prices on the stock market.

We are currently working on products that will allow you to take data from a relational database and apply ALNs to learn to predict the values in certain columns of a table from some other columns.   This could be of help in trying to analyze relationships among the columns of your data, for example.

ALNs operate by producing decisions.   However, it is often desirable to have an explanation of how the network arrives at the decisions.   If the network has developed a simple decision procedure, then that may give rise to new insights into the solution of your problem.   If the networks are fairly small, this job of finding a simple explanation for the decisions can be easily accomplished by printing out the logic tree resulting from training and studying it.

In large networks, unfortunately, the question of how the network makes its decisions is often too difficult to answer. This is because of a fundamental theorem of computing science that says that it is computationally intractable (more precisely, NP-complete) just to decide if a two-layer network with ORs feeding into one AND will ever produce a one output. If one can't determine with a reasonable expenditure of computer time whether a network will ever produce a one output, then how can one hope to deal with more complex questions?   This leads us to the important question of safety.

Safety of Artificial Neural Systems

Safety of Artificial Neural Systems

In some cases, the cost of a wrong decision is too high.   It is usually impossible to test an artificial neural network for all possible inputs and be sure that catastrophic decisions will not occur.   There are too many possible inputs to do that.   We have to have an overview of what the system does for all its possible inputs.   A simple explanation of what the network does would succeed, but as we have seen above, we usually can't get one from the trained network.   Hence, if one can't be sure that the network can never produce an output that leads to disaster, then artificial neural network technology will remain rather limited in its applicability. This limitation would exclude many important engineering problems, like controlling aircraft.

If an artificial neural network could erroneously apply full power just as an aircraft is moving slowly towards the terminal building, do you think the engineers would use it? The potential for disaster is a serious problem that must be addressed before artificial neural systems, including the special case of ALNs, can be deployed in many areas.

We don't believe the answer lies in understanding or explaining what a trained network does, in general. We do believe in creating ALN networks by a design methodology that leads to networks with well-understood properties. No one should be surprised about this, since it is exactly what is done in the case of software.   No feasible amount of testing can assure in general that either software or artificial neural systems are safe.   Only a high quality design process can assure that.

A safe design methodology for ALNs will be available in commercial versions of our software now being developed.   It will be based on careful control of where functions are increasing and decreasing, and will allow the designer to input a priori knowledge into the design process.   By knowing where the networks are increasing and decreasing, one can be assured of the safety of outputs between inputs which have been tested.   In this way one gains certainty of the safety of the system for all possible inputs.

Fortunately, we know how to make large ALN systems safe.   We feel other large neural systems cannot be made provably safe, in general. This criticism applies in particular to multilayer perceptrons, the most popular type today, of which ALNs are a special case.

We would be pleased to discuss solutions to problems involving both high speed and safety with anyone having those requirements.   We are sure we can convince you that ALNs can be made as safe as any computer system you are currently using.

An Invitation to Developers

Eliminating Redundant Inputs

Here is one way to use ALNs to analyze data :

ALNs allow one to try leaving out certain variables in the input data to see if reasonable decisions can still be obtained by training. In one problem, a set of 700 points in a spectrum could be reduced to 25 which gave almost the same quality of decision at a potentially great savings in time.   After studying the data in this way, you can apply other techniques that may be contractually specified to solve your problem.   The reduction in the number of input variables may even be required to make the other technique work.

The Future of Training versus Programming
The Advantage of Extremely High Speed
Data Analysis Applications of ALNs

The Future of Training versus Programming

In the future, there will be a lot of systems that are developed using training combined with programming. Training will be the principal means for developing robotic abilities to process information from sensors. Because of their high speed and immunity to noise and distortion, <u>ALN</u>s will be the tool of choice for those applications.

We expect <u>ALN</u>s to be used in many kinds of products that include sensors.   All homeowners would like to train a robot to cut their grass, for example.   As the population ages, there will be a greater need for robots to assist the physically and mentally impaired.   Right now, this looks like fantasy, but the point is that there have never been pattern recognizers with the capabilities of <u>ALN</u>s before, so it is only now that these developments can begin.

<u>An Invitation to Developers</u>

An Invitation to Developers

In order to appreciate what ALNs can do on one of your problems, you'll just have to try them out. You can get into the area as easily as writing a little program in the lf language, following one of the samples you call up when you click on the lfEdit icon and select the "Open" command.

When you have done your feasibility studies, we will be there to support you with superior commercial ALN products.

We invite all developers of systems for classifying inputs, analyzing data or requiring high speed pattern recognition capability to join us in experimenting with ALNs.   This non-commercial version of ALN software, which includes all source code, is for your use in trying out your ideas.   A commercial version of ALN software will be ready in late 1992 for inclusion in your future systems, and will focus on quality of training and decision making and on the critical aspect of safety.

The Advantage of Extremely High Speed
Data Analysis Applications of ALNs
The Future of Training versus Programming

The Multiplexor Problem

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

The trick to the whole problem is that an ALN doesn't know which inputs are the control leads, and which are the input leads... an unlabeled truth table for a multiplexor of 3 control leads and 8 input leads looks like a random jumble of ones and zeros. The job of the ALN is to correctly deduce which of the inputs are the control leads, and propagate the correct input lead value as the tree output.

This problem was solved to produce a circuit testing correctly on 99.4% of 1000 test vectors in 19 epochs. 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].

The code for the problem can be found in *mult.c*

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 anemia. We are inquiring into eighty boolean-valued factors of which "bitten by anopheles mosquito", "drinks Gin and Tonics", and "has sickle-cell anemia" 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.) 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 etiology of the disease.

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

The OCR Problem

This optical character recognition (OCR) demonstration shows the power of trained adaptive logic networks (ALNs) to identify images of printed or handwritten characters. Each input character image, before it is presented to the ALN recognition system, is translated, rotated and corrupted by "salt-and-pepper noise" (so-called, because some of the picture elements, or pixels, change from light to dark, or vice-versa, as though a grain of pepper or salt had fallen on them). The level of noise in this demonstration is much higher than could be tolerated by most current OCR systems, showing how immune ALNs are to undesirable distortions of the inputs.

When you test yourself against the trained networks in "reverse" mode, where each character is initially completely obscured by noise, you can compare ALNs to your own human visual abilities.   As the noise disappears, the ALNs may decide for a particular character, e.g. "A", "L", or "N", or whatever images you have drawn in yourself (children may prefer cartoon characters!).   A decision for a certain character is indicated by the light bulb corresponding to that character flashing once.   The networks may be wrong, as indicated by flashes that stop before the noise is gone.   We can define the moment of successful recognition by the ALNs as the earliest moment when the correct light flashes and continues to flash as the noise diminishes to zero.

How do you think your decisions compare to the adaptive logic network's decisions?

Using lf: 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
        vote = 5
        save tree to "multiply.tre"
```

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.   In some cases, where trees are learning the training set well, but are not generalizing effectively on test data, you can take votes on the result, which improves generalization.   Voting works by having an odd number of trees vote on the outcome of a particular bit in the output.   Having no vote statement is equivalent to having a vote of 1.

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
        save coding to "multiply.cod"
```

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 codomain dimension 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 sum of the <width> values for the codomain columns (which come after the domain columns) actually defines how many trees will be learning output bits of this function.   If each each bit of the output is determined by a vote of v trees, then the number of trees required is multiplied by v.

The quantization statement defines for each column the total number of coded bit vectors used to code values 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.

For a more complete discussion on random walks and encoding input and output dimensions, see atree_rand_walk().

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.   You can leave out a training set entirely by specifying "training set size = 0", and leaving out the "training set =" statement.   This is useful for loading and executing previously stored functions without further training.

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.   You can leave out a test set entirely by specifying "test set size = 0", and leaving out the "test set =" statement.

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     11.000000 10    33.000000 32     33.000000 32
3.000000 2     12.000000 11    36.000000 35     36.000000 35
4.000000 3      1.000000 0      4.000000 3       4.000000 3
4.000000 3      2.000000 1      8.000000 7       8.000000 7
4.000000 3      3.000000 2     12.000000 11     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).

ERROR HISTOGRAM

```
0 errors    144
1 errors    0
2 errors    0
3 errors    0
4 errors    0
5 errors    0
6 errors    0
7 errors    0
8 errors    0
9+ errors    0
```

Following the result table is the error histogram, which counts, for each of the codomains, the number of times the result quantization level differed from the actual quantization level by n.

This problem takes about an hour to run on a 386/33.   Please compare to the June/90 Article of AI Expert where it was estimated that it would take two months on a Cyber 205 for a backpropagation net to learn the twelve times multiplication table!

If Language Syntax

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" dim codim function_statements

dim :   "domain dimension =" integer

codim: /* empty */
        | "codomain dimension =" integer

function_statements :   function_statement
        | function_statements function_statement

function_statement :   quantization
        | coding
        | coding_io
        | 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

coding_io: "save coding to" string
        | "load coding from" string

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
                   | tree_io
                   | min_correct
                   | max_correct
                   | max_epochs
                   | vote_no

tree_size :   "size =" integer

tree_io :   "save tree to" string
            | "save folded tree to" string
            | "load tree from" string

max_correct :   "min correct =" integer

max_epochs :   "max epochs =" integer

vote_no :   "vote =" integer
```

If Example: abs_sine.lf

This is a straightforward problem involving taking the absolute value of sine around the circumference of the unit circle.   Because of the absolute value, backpropagation nets may have a great deal of difficulty fitting the curve, something that ALNs have little difficulty with.   The accuracy of fitting in a backprop net can be, of course, arbitrarily high, in theory, however, high values of the weights are required.   The need to allow high magnitude weights runs counter to the need for good generalization.   It can be dangerous to allow high weights, since spikes in network output of the backprop net can occur unexpectedly if they are missed during testing.

This problem takes about half an hour to run on a 386/33, achieving 2 decimal places of accuracy.

If Examples: noisyand.If & noisyxor.If

ALNs can often be given inconsistent or noisy data to train on, in which case they will try to learn the case that is presented most often.

noisyand.If

This is an If example that tries to learn the AND function of two boolean inputs.   The training is done with contradictory data.   For each pair of values of the input, for example A = 1, B = 0, there are three correct training samples and two incorrect ones in the set.   The optimal decision here is to learn the AND, but that may not always happen.

noisyxor.If

This is an If example that tries to learn the XOR function of two boolean inputs.   The training is done with contradictory data.   For each pair of values of the input, for example A = 1, B = 0, there are four correct training samples and one incorrect sample in the set.   The optimal decision here is to learn the XOR, but that may not always happen.

What is an ALN?

An <u>ALN</u> (adaptive logic network) is a kind of artificial neural network. It is a special case of the familiar multilayer perceptron (MLP) feedforward network and has been designed to perform basic pattern classification computations at extremely high speed -- literally at the upper limit of speed for any digital system.

Abilities and Properties of ALNs in atree 2.7

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.

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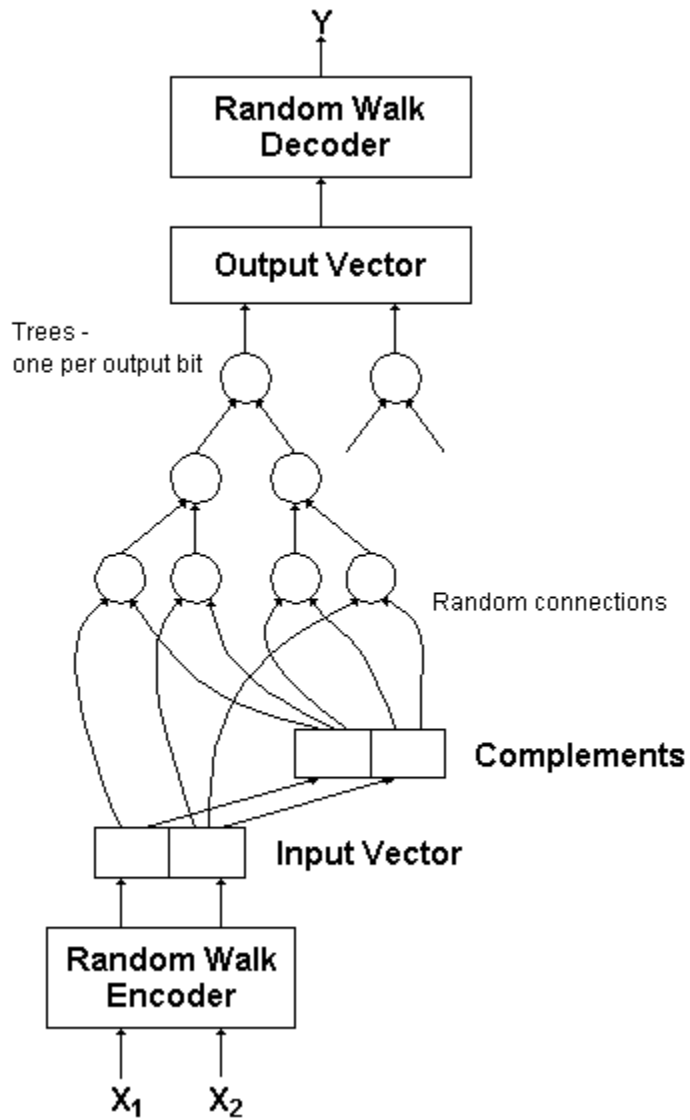
Despite the apparent limitation to boolean data, simple table-lookups permit representing non-boolean input values (integers or reals for example) as bit vectors, and these representations are concatenated and complemented to form the inputs at the leaves of the tree.   For computing non-boolean outputs,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

ALN Diagram: Using Several Trees to Compute Y = f(X1, X2)

Y

Random Walk
Decoder

Output Vector

Trees -
one per output bit

Random connections

Complements

Input Vector

Random Walk
Encoder

X_1 X_2

What Could One Use ALNs For?

They are currently being tried out in simulations using the atree software for various applications:

1. Optical Character Recognition (OCR) -- a demo of OCR is included in atree release 2.7; the great immunity of ALN trees to noise is made clear. (Thomas)

2. Discriminating between two kinds of particles produced by an accelerator, where a decision must be made in less than 1/2 microsecond. (Volkemer)

3. Designing control systems for prostheses to enable people with spinal cord damage to walk by means of functional electrical stimulation. Simplicity of fabrication, safety and lightness are important here. (Stein, et al.)

4. Analyzing spectra of tarsands to determine the composition; data analysis was used to cut down the number of spectral measurements required from 700 to 25. (Parsons)

5. Classifying the amount of fat in beef based on texture measures taken of ultrasound images of beef cattle. (McCauley et al.)

Inputs

The following assumes the reader has read something about typical feed-forward neural nets.   In the case of ALNs, the input data to the learning system must be boolean.   Non-boolean data is coded into boolean vectors before entering the learning part.   This may involve arithmetic and table lookup operations.   Several boolean outputs may be used to provide outputs which are more complex than boolean, say continuous values.   There are no a priori limitations on the kind of functions that can be treated using ALNs.

Architecture of the Net

The interconnections of nodes usually take the form of a binary tree, except that there are multiple connections of inputs to the leaf nodes of the tree, some involving inversions.   There is no limit to the number of hidden layers in practice.   Ten or fifteen is typical.   A small net might have 255 nodes, a large one 65,535 nodes.   One doesn't have to worry about the shape of the net as much as with other networks.

The Elements of the Neural Network

The nodes (or adaptive logic elements) have two input leads. The input signals x1, x2 are boolean (0 or 1).   A connection "weight" to be used for each input (to use the term for the usual kind of neural net) is determined by a single bit of information.   The nonlinearity, or "squashing function", used in ALNs is just a simple threshold (comparison to a constant).   Specifically, if b1 and b2 are boolean, then the element computes a boolean value which is 1 if and only if

   (b1 + 1) * x1 + (b2 + 1) * x2 >= 2.

The four combinations of b1 and b2 (00, 11, 10, 01) generate the four boolean functions of two variables: AND, OR, LEFT, and RIGHT, where LEFT(x1, x2) = x1 and RIGHT(x1, x2) = x2.   For example 1 * x1 + 1 * x2 >= 2 if and only if both x1 AND x2 are 1.

The LEFT and RIGHT functions, in effect, serve to provide some flexibility in the interconnections.   They disappear after training is complete.

Two counters in the node count up and down in response to 0s and 1s desired of the network when the input to the node is 01 or 10 respectively.   They try to determine the best function for the node even in the presence of noise and conflicting training data.   The counters also disappear after training is complete.

The Training of the Tree

A tree of such elements is randomly connected to some boolean input variables and their complements. Inputs enter the tree at the leaves and the output is at the root node. The input variables are components of a vector representing a pattern, such as a handwritten character. Training on a set of input vectors, furnished with the desired boolean outputs, consists of presenting input vectors in random sequence, along with the desired outputs. It results in an assignment of functions to nodes that allows the tree to approximate the desired outputs specified by the training data.

To orchestrate the changes to be made during training, a solution to the credit assignment problem is used. It is based on the idea that if one input to an AND (OR) is 0 (1), then it doesn't matter what the nodes of the subtree attached to the other input are doing. According to this simple rule, a node is responsible if a change in its output would change the network output. (This is like backprop, except that the quantity being propagated backwards is either 0 or 1.)

The fact that all node functions are increasing causes a positive correlation between the node output and the network output, which indicates the direction of changes that must be made. In addition, there is a small refinement which goes beyond this simple scheme. (You can look at the function starting

adapt(tree,vec,desired)

in atree.c to confirm that this is a very simple yet effective mechanism).

After training, the ALN then has built-in capacity to generalize, i.e. to give appropriate responses to other possible input vectors not presented during training. This comes about not because of any additional hardware or software, but just because of the architecture of the tree -- trees filter out perturbations of the inputs. Hence no speed is lost in order to generalize well -- the architecture does it.

The training set can contain contradictory samples, and the relative numbers of such conflicting samples for the same input will be taken into account in the training procedure to approximate maximum likelihood and Bayes decision rules.

The Learning Algorithm

For those who would like to understand the ALN learning algorithm used in the atree demo software, the best published introduction to ALN learning is the paper:

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

The only significant changes in atree 2.0 and 2.7 from that paper are that the heuristic responsibility is now a combination of true responsibility, error responsibility, and responsibility of both children of LEFT and RIGHT nodes.   The details are all in the "adapt" routine in atree.c shown in part below.

```
private void
adapt(tree,vec,desired)

atree* tree;
bit_vec* vec;
BOOL desired;

{
    register int lres;
    register int rres;
    register int funct;

/*
 * INCR and DECR implement bounded counters.
 */
#define INCR(t) ((t) += ((t) < MAXSET))
#define DECR(t) ((t) -= ((t) > MINSET))

    funct = tree -> c_tag;
```

The *c_tag* is 0 for an AND node and 1 for an OR node.   LEFT and RIGHT nodes have tags 2 and 3. Which is which doesn't matter here.

The node we are adapting is the root of a recursively defined "tree" datatype.   First the node has to have its inputs evaluated if that hasn't already been done.   (Of course, we don't adapt leaves.)

```
    if (funct != LEAF)
    {
        /* If the left child is unevaluated, evaluate it */

        if ((lres = tree -> n_sig_left) == UNEVALUATED)
        {
            lres = atree_eval(tree -> n_child[LEFT_CHILD], vec);
            tree -> n_sig_left = lres;
        }
```

Now *n_sig_left* stores the result of evaluation on the left.

```
        /* If the right child is unevaluated, evaluate it */
```

...   Same thing on the other side.

```
        /* Update the counters if needed */
```

There are two counters, one associated with the (left,right) = (0,1) input pair and one with the (1,0) input pair.   The appropriate counter is counted up or down depending on the desired network output.   (Yes, the desired *network* output -- because all nodes are monotonic increasing functions.   There is thus a positive correlation between the net output and any element's output.)

```
        if (lres != rres)
```

If there is a (0,0) or (1,1) input pair to the node, no adaptation occurs, according to the above condition.

```
    {
        if (lres)
        {
            (void) (desired ? INCR(tree -> n_cnt_left)
                            : DECR(tree -> n_cnt_left));
        }
        else
        {
            (void) (desired ? INCR(tree -> n_cnt_right)
                            : DECR(tree -> n_cnt_right));
        }

        /* has the node function changed? */
```

The values of the two counters determine the function of the node; e.g. if they are both above (below) their midpoints, it is an OR (AND) node.   If they are on different sides of the midpoints, the node is a LEFT or RIGHT.

```
        tree -> c_tag = node_function(tree);
        funct = tree -> c_tag;
    }

    /* Assign responsibility to the left child */
```

It's symmetric to the responsibility of the right child discussed below, so we don't need to discuss it.

```
    /* Assign responsibility to the right child */

    if (lres != desired || funct != (lres ? OR : AND))
    {
        adapt(tree -> n_child[RIGHT_CHILD], vec, desired);
    }
```

This finishes the algorithm, except for the one line that must be understood to grasp the entire atree credit-assignment algorithm, and that is the condition in the last if-statement:

lres != desired || funct != (lres ? OR : AND)

The current node is heuristically responsible when we get to this if statement, and the question is which children should be heuristically responsible?

If the left input is not equal to the desired network output (For a heuristically responsible node lres != desired is called an "error" on the left), the right child is made responsible.   The rationale for this is that right child is made responsible if there is an error on the left, since maybe the left child can't correct the error (maybe it's a leaf).

If the node function is not OR or AND (i.e. it is LEFT or RIGHT with *funct* == 2 or 3, which are never values of the expression (lres ? OR : AND)), then the right child is made heuristically responsible.   This says: try to put the subtrees that are cut off by a LEFT or RIGHT to some useful work. If such a cut-off subtree gets good enough, the node function might change to AND or OR to give that subtree an effective role in the net.

Finally, if the left input is 0 for an OR node or 1 for an AND node (i.e. if the right signal *rres* is enabled to pass through), then the right child is heuristically responsible.

This last part of the credit assignment by heuristic responsibility, if implemented by itself, is true responsibility for the AND and OR nodes: a node is truly responsible if a change in its output would change the network's output.   This component of heuristic responsibility is just hill-climbing or gradient

descent.   The two earlier components of heuristic responsibility go beyond hill climbing to give the learning algorithm much more power than hill climbing alone.

Note that the left input controls the right child's responsibility, and vice-versa.   This implies that every node in the tree can interact with every other node at every pattern presentation.

There is one other trick: if the network output is still wrong after the first adaptation, adapt again.   This compensates for a shortage of errors to correct as the end of adaptation approaches.

(The code above was coded by Rolf Manderscheid, to whom should go the glory or blame for putting a set of nested conditions into one line.)

Building Hardware From the Trained System

Training an ALN results in a combinational logic circuit.   After adaptation, one eliminates LEFT and RIGHT functions, which are now redundant and groups the connected subtrees of two-input ANDs together to get ANDs of fanin >= 2.   Similarly for ORs.   The (alternating) layers of ANDs and ORs are logically equivalent to layers consisting entirely of NANDs.   (Use De Morgan's Law.)

The result is directly translatable into easily built hardware -- essentially a tree of transistors.   This is far, far simpler than any other neural system.   There are no built in delays, flip-flops, arithmetic units or table look-ups to slow the system down.

Hardware Speed

The execution time of such a circuit depends on the depth, not the size.   If we were using a twenty layer tree of transistors in submicron CMOS, the entire function would be executed in a few nanoseconds.

In the usual terms of numbers of one-bit connections per second, we would get many trillions of connections per second from an ALN, thousands of times more than existing special hardware systems for implementing backprop.   The fastest we know of on the market for backprop only allows a few billions of connections per second at peak rate.

Cost of Hardware

For small problems, even one of the off-the-shelf programmable chips that exist today can do the job, though at a slower rate than a custom chip.   Such fast, inexpensive hardware is being investigated at an accelerator facility for making a fast trigger to recognize elementary particles in under 1/2 microsecond. A few hundred dollars should cover the materials.

This is in sharp contrast to other kinds of neural networks where you need to have processors costing thousands of dollars just to get into the game seriously, and even then the result of all their expensive, learning will be a system that executes far more slowly than an <u>ALN</u> tree of transistors.   With <u>ALN</u>s, you are seriously in the game from day one with the atree simulators.

Speed of the atree Simulator

Even in the atree software simulator, evaluation is fast, thanks to the fact that boolean logic can be lazily evaluated.   Once one input to an <u>AND</u> gate is computed to be 0, we don't need the other input, so we can suppress a whole subtree of the computation.   Laziness can provide speedups by factors of 10 or 100 for typical problems.   On the other hand, backprop systems do not enjoy such speedups.   Non-logical MLPs compute *everything*.   It's like doing programming without any "if"-statements.   (It's no wonder the usual neural networks need massive parallelism -- they suffer from this serious omission.)

Training Speed

The adaptation algorithms are also designed to be fast: what corresponds to backprop, a credit assignment computation, is also combinational logic in hardware, so it will be possible to build systems which will adapt a tree at a rate of, say, 25 million patterns per second.

The difference in speed of training and execution of backprop systems on the one hand, and ALN systems on the other, results from the drastic simplification of the usual MLP paradigm to base it on boolean logic.   The difference is immense.   One shouldn't think of it as getting a racing car from a station wagon (twice as fast, say), it is more like getting a rocket from a hot air balloon (thousands of times faster).   As the problems get larger, the advantages of the ALN system over the usual MLP just keep on growing.

Generalization and Noise Immunity

Good generalization without loss of speed is the strong point of ALNs.   Pattern classification problems generally require an output which is the same for all patterns in a class.   Members of a class are in some way similar.   Similarity could be based on all sorts of criteria, but generally it can be defined by saying two individuals have many features in common.   If a feature is represented by a boolean variable, then the above says, similarity can be measured by Hamming distance between boolean vectors.   In order to use ALNs, we generally have to code other representations of individuals into boolean vectors in such a way that similarity for pattern classification purposes is reflected in Hamming distance.

Binary trees with node functions AND, OR, LEFT, and RIGHT have been shown to have the property that the functions they realize tend to have the same output value when the inputs are perturbed.   This depends on averaging over all functions on the tree, weighted by the number of ways each can be produced by setting the two bits at each node which determine the node's function.   It is also necessary to average over all vectors and all perturbations of a given number of bits.

The plausible reasoning behind this is that when an input signal to a gate changes, it may or may not change the output.   A change to an input to an AND will not change its output if the other input to the AND is 0.   Over all, the probability of a change is 0.5, if one input changes.   So for a 10-layer tree, the probability of a single bit change at a leaf causing a change of the tree output is 1/1024.

ALNs match the solution to the problem to provide fast pattern recognition.   Simple properties of boolean logic provide speed, immunity to noise, and offer the possibility of lazy evaluation.

The Future

In the Fall of 1992, we expect to have a much enhanced version, embodying some recent developments, that have resulted in part thanks to interaction with people on Internet via email and news.   It will have

- superior adaptive algorithms for both continuous and boolean data

- an ability to deal with continuous values, without significant loss in speed, that will make other MLPs obsolete, including those using   variants of backprop

- a sound statistical basis for fitting continuous functions

- a design methodology that will be easy to understand and will result in provably *safe* systems; there will be no more problem of wild values being produced by a "black box", so ALNs can be applied in safety-critical areas

- an interface to common database management systems on micros and mainframes

- an interface to a popular spreadsheet program under Windows.

- facilities to encourage developers to apply ALNs to pattern recognition problems: OCR, voice, control of virtual realities etc.

- a price, since we can't carry on otherwise.

An Invitation

We hope the above arguments will be enough to convince the reader to try the atree release 2.0 software for UNIX and/or the new release 2.7 for Windows on the IBM PC and compatibles.   At the very least, you should take a look at the <u>OCR</u> demo in release 2.7 to see the great noise immunity which results from such simple means.

The lf language should make it easy to get started, and there are lots of examples in release 2.7. (You could look at them even if you only want the UNIX version.)

If you like what you see, work with us.   Lots of researchers are doing so, some via the net: physicists, engineers, neuroscientists, speech pathologists, ...

Join the alnl mailing list (email to alnl-request@cs.ualberta.ca providing a correct address that will last for a while please).

The interest in backprop-type neural networks which has blossomed in the last few years is obviously fading fast and funding is less than expected.   We heard that it is because "neural networks" were just slightly better than standard techniques.   The problems that need high speed pattern recognition are still there, though, and when speed is an issue, <u>ALN</u>s are not just slightly better, but thousands of times better than anything else.

As for the quality of <u>ALN</u> classifiers, you'll just have to try them out in your area of application. For beef grading, at last report, <u>ALN</u>s were the best technique they had.   A particle physicist, as far as we know, is still trying to explain what information an <u>ALN</u> was using that managed to learn even when he deprived it of momentum information that was thought to be essential.   In prosthetics, the <u>ALN</u> didn't make a mistake a physiotherapist did, even though the mistake was in the training set.   A lot of people are pretty happy about their results with <u>ALN</u>s.

Good luck with your results using them!

Writing Applications with atree 2.7

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 file **easymosq.c**, which hides most of Windows' dressings for clarity by using the Borland C++ 3.x EasyWin library.   A C++ application can be found in the OCR demo directory, and more traditional Windows C applications can be found in the mosquito, multiplexor, and lf directories.

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 is a long (far) pointer to a bit_vec */
    LPBIT_VEC result;
    LPATREE *forest;      /* LPATREE is a long (far) pointer to an atree */

    /* 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.   In future commercial versions of the software under development, there is no need for encoding input into random walks.   It is, however, necessary for atree 2.7.

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 level.   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.

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. */
}
....
```

To see how to incorporate majority voting into your programs, which can increase the accuracy of your results, consukt the example program file *mult.c*.

Data Structures

The following constitutes the data structures available for use in the atree and BV libraries.   Please the files *atree.h* and *bv.h* for further details.

LPATREE   :   a long pointer to an atree.   Declare your ALN variables to be this type.   You can get pointers to created trees by using atree_create(), atree_load(), and atree_read().

bit_vec   :   a structure containing an LPSTR field *bv*, which points to the chunk of memory containing the bit vector, and an integer field *len*, which contains the length in bits of the bit vector.

LPBIT_VEC   :   a long pointer to a bit_vec.   You should declare most of your bit vector variables to be this type.   There are a number of bit vector functions which return pointers to bit vectors.   See the BV Library for more details.

code_t   :   a structure containing several fields for building encoded representations of input data.

LPCODE_T   :   a long pointer to a code_t.   Use the function atree_set_code() to fill in an LPCODE_T variable.

LPFAST_TREE   : a long pointer to a fast tree, which is a compressed representation of an atree.   Use the function atree_compress() to generate a fast tree.   A fast tree is executed much like a decision tree.

The Windows atree Library

This section is intended for advanced programmers.   Users can skip it!

The atree library consists of a the include files *atree.h* and *bv.h*, and the library of routines in *atree.c* and *bv.c*. *Atree.h* must be included in all software making calls on the library (it includes *bv.h* automatically); please remember to include **\atree_27\include** in your include path. 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: **StatusDlgProc**, along with any other window procedures your application may have - see *mosquito.def* for an example.   Also, your application must include *atree.rc* in its resource file in order to see the Status window.

The following functions are available to the programmer that relate to ALN management:

void atree_init(hInstance, hWindow)
void atree_quit()
LPBIT_VEC atree_rand_walk(num,width,p)
LPATREE atree_create(numvars,leaves)
void atree_free(tree)
BOOL atree_eval(tree,vec)
BOOL atree_train(tree,tset,...)
void atree_print(tree,stream,verbosity)
int atree_store(tree, filename)
LPATREE atree_load(filename)
LPATREE atree_read(stream)
int atree_write(stream, tree)
LPATREE atree_fold(tree)
LPFAST_TREE atree_compress(tree)
int atree_fast_eval(ftree, vec)
void atree_fast_print(ftree, stream)
int atree_set_code(code, high, low, ...)
int atree_encode (x, code)
int atree_decode(vec, code)
LPCODE_T atree_read_code(stream, code)
int atree_write_code(stream, code)

The BV Library

The following functions provide methods to manipulate and create a bit vector structure. Bit vectors are used in many of the functions in the atree library.

LPBIT_VEC bv_create(length)
LPBIT_VEC bv_pack(unpacked,length)
int bv_diff(v1,v2)
LPBIT_VEC bv_concat(n,vectors)
void bv_print(stream, vector)
void bv_set(n,vec,bit)
BOOL bv_extract(n,vec)
BOOL bv_equal(v1,v2)
LPBIT_VEC bv_copy(vec)
void bv_free(vector)

The Windows Support Library

This library includes functions to allow Microsoft Windows specific support during various atree routines and functions. The Windows Support Library contains the following function:

void Windows_Interrupt(cElapsed)

void atree_init(hInstance, hWindow)

HANDLE hInstance;
HWND hWindow;

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.   the parameter *hWindow* should be the handle to the window that will be the parent of the atree Status Dialog box, which pops up during training.   If you don't know the window handle, or don't care, set *hWindow* = NULL.   Currently, **atree_init()** calls the **srand()** routine to initialize the random number generator and initializes the atree Status window.

If using the EasyWin library with Borland C++ (to generate C style programs without the Windows "stuff"), you can get the application instance handle with:

    extern unsigned _hInstance;

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.

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 increments 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 bit string 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 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.

The idea of encoding the input and output will no longer be necessary with new software currently under development.   The trouble with encoding is that a bit changes back and forth between 0 and 1 during the walk, which destroys any monotonicity we might want to have in the learned function,   Monotonicity of pieces of the functions to be learned will be a powerful tool in making ALNs acceptable in safety-critical applications.

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 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:

$$leaves >= 2 * numvars$$

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*.

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.

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 fashion, which uses lazy evaluation: if the first input to an AND (OR) is 0 (1), don't bother computing the other input [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().

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.

void atree_print(tree,stream,verbosity)

LPATREE tree;
*FILE *stream;*
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.

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.

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 <u>atree_read()</u>.   A NULL pointer is returned if any error or EOF is encountered.

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.

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 <u>AND</u>, <u>OR</u>, <u>LEFT</u>, <u>RIGHT</u> 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.

LPATREE atree_fold(tree)

LPATREE tree;

This routine removes all <u>LEFT</u> and <u>RIGHT</u> 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.

Trees can be folded, stored, and reloaded.   However, once nodes are eliminated, further training will not recover them.   Future software will allow "tree growth".

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, at each of which a binary decision based on the leaf input is made.   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.

A fast tree is like a decision tree, but the fast tree may reuse the same tree input bit if it sent to several leaves.   (A faster procedure without repetitions could be written, but we think that would use a lot of storage.)

int atree_fast_eval(ftree, vec)

LPFAST_TREE ftree;
LPBIT_VEC vec;

This routine is the equivalent of <u>atree_eval()</u>, but for fast_trees.

void atree_fast_print(ftree, stream)

LPFAST_TREE ftree;
*FILE *stream*

This routine writes a representation of *ftree* to the file pointed to by *stream*. 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.

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.   The coding for a boolean variable is thus 1:1 and quantization is 2.

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.

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).

A faster decoding procedure might use algebraic codes.   Also, <u>ALN</u> forests are natural decoders.   Such a forest computing an index to a list of possibilities is a powerful, general tool for many applications.

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.

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.

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.

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.

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!).

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.

void bv_print(stream, vector)

*FILE *stream;*
LPBIT_VEC vector;

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

void bv_set(n,vec,bit)

int n;
LPBIT_VEC vec;
BOOL bit;

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

BOOL bv_extract(n,vec)

int n;
LPBIT_VEC vec;

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

BOOL bv_equal(v1,v2)

LPBIT_VEC v1;
LPBIT_VEC v2;

This routine tests two bit vectors for equality.

LPBIT_VEC bv_copy(vec)

LPBIT_VEC vec;

This routine returns a copy of *vec*.

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.

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 500 to work fairly well.

ALN is short for Adaptive Logic Network.   ALNs are a type of artificial neural system (ANS), but are unique in that they employ a logic gate architecture that makes them extremely fast.

An AND gate takes two inputs, each of which may have the value 1 or 0.   If both inputs are 1, then the output of the AND gate is 1, otherwise it is 0.

A NAND gate takes two inputs, each of which may have the value 1 or 0.   If both inputs are 0, then the output of the NAND gate is 1, otherwise it is 1.

An OR gate takes two inputs, each of which may have the value 1 or 0.   If both inputs are 0, then the output of the OR gate is 0, otherwise it is 1.

An XOR gate takes two inputs, each of which may have the value 1 or 0.   If both inputs are the same, then the output of the XOR gate is 0, otherwise it is 1.

A LEFT gate takes two inputs, each of which may have the value 1 or 0.   If the left input is 1, then the output of the LEFT gate is 1, otherwise it is 0.

A RIGHT gate takes two inputs, each of which may have the value 1 or 0.   If the right input is 1, then the output of the RIGHT gate is 1, otherwise it is 0.

OCR stands for Optical Character Recognition.   Computers are able to scan in images of paper and store them in digital form, where they can attempt to recognize the characters that existed on the paper.

[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, A. Dwelly, J.-D. Liang, D. Lin, S. Reynolds, Learning and Generalization in Adaptive Logic Networks, in Artificial Neural Networks, Proceedings of the 1991 International Conference on Artificial Neural Networks (ICANN'91), Espoo, Finland, June 24-28, 1991, T. Kohonen, K.Makisara, O. Simula, J. Kangas eds. Elsevier Science Publishing Co. Inc. N. Y. 1991, vol. 2, pp. 1173-1176.

[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: <u>Parallel Distributed Processing</u>, 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.