An Implementation of ARTMAP

Peter Lane
University of Exeter,
April 1995

Contents

1 Introduction

ARTMAP is a design for a class of artificial neural network, and is the brainchild principally of Grossberg and Carpenter.[1] The aim of this mini-project being to create an implementation of ARTMAP and carry out some tests of its effectiveness for supervised learning. Before going on to a consideration of neural networks in general and ARTMAP in particular, the rest of this introduction is devoted to a brief description of the task of learning, and the ways it may be performed by a computer.

'Learning' is an activity all humans perform, albeit with varying degrees of efficiency. Performance on a task is reviewed and a repetition will see some variation, followed by a further review. This kind of activity is normally expected of an individual, even more so as boredom and inattention lead to unintentional variation, critical with certain tasks. The computer is not generally seen in the same light. A company may require a million bills to be calculated and sent to customers. No variation would be tolerated during the million repetitions.

Learning is seen as a useful skill for computers to possess, as then activities like bill production may be demonstrated to, as opposed to programmed into, the computer. A new bill layout may be introduced by a new demonstration, instead of costly reprogramming. The computer's strength of faultless iteration may then take over in producing the million bills.

Unfortunately, 'learning' is a nebulous concept, and we as yet have little idea of how humans themselves learn, let alone whether a computer is capable of emulating our ability. When should learning actually occur? What exactly is meant by saying learning has occurred/is occurring/will occur? What improvement are we expecting? And how should this be calibrated?

The process of learning may be viewed as an activity between two persons (minds?). The teacher, with a skill to impart, and the student, who will acquire the skill. Which activities each of the pair carries out, and the amount of dominance each has over the proceedings, leads to a differentiation in the learning process. In what follows, 'student' refers to a human or a computer.

Firstly, the teacher may directly brainwash, or program, the student with the necessary information. The student need do nothing, and this is obviously the familiar activity of programming a computer to perform a task. If we are ingenious enough to encode all the required skill into a set of instructions, the computer need only follow the instructions to possess the skill.

Instead, the teacher may present the salient features in some (natural) language, which the student must interpret and incorporate into any existing knowledge. This is standard education practice, and requires the student to do some work in understanding what is presented.

The teacher may alternatively present examples of what is to be done. For example, correct proofs of various mathematical theorems. Notice that the teacher's task is becoming easier, but the student must now put in a lot more work. The examples must be understood on their own terms, and the reasons for correctness, or not if negative examples are used, need to be comprehended. This form of learning is termed 'supervised learning', as the teacher guides the student to a model with examples and by correcting the student's guesses.

Finally, the teacher may be done away with altogether, in that the student must monitor his own performance and judge whether it is appropriate. This 'unsupervised learning' is akin to pure research, as concepts must be created and applied with no external guidance. Factors such as effectiveness, conciseness, ease of use etc become the self-applied criteria. The creative demands and necessity of self-monitoring require the greatest skills of the student.

These categories are not entirely independent, and Winston[2] views them as points on a continuum. A human student will learn in each of these ways in turn, dependent on inspiration and current levels of knowledge. The 'unsupervised' category may sometimes be a form of 'supervised' learning, when feedback from the environment can be relied upon to correct the student's hypotheses. In spite

[1]The principal sources for this implementation being, Carpenter, G. A., Grossberg, S., & Rosen, D. B., 'ART 2-A: An Adaptive Resonance Algorithm for Rapid Category Learning and Recognition' in *Neural Networks*, Vol. 4, pp. 493-504, 1991. Carpenter, G. A., Grossberg, S., & Reynolds, J. H., 'ARTMAP: Supervised Real-Time Learning and Classification of Nonstationary Data by a Self-Organising Neural Network' in *Neural Networks*, Vol. 4, pp. 565-588, 1991.

[2]See for instance, the article 'Learning Descriptions from Examples' in *Psychology of Computer Vision*, edited by Winston, Patrick Henry, MIT Press, Cambridge, 1975.

of this, we do have a framework in which to consider possible models of computer based learning and their relative capabilities.

Attempts at making a computer learn have had a long history. As pointed out above, traditional programming falls into the first category, but is a complex and error-prone process, through which knowledge and skills must be synthesised into a computer usable form. 'Learning by being told', the second category, although the best for humans, is of little use in computers. This is due to the linguistic demands, if we are to go beyond mere programming. Only limited systems in artificial domains, like database design[3] or robotic control,[4] can be considered in this manner.

Most attention has focussed on the 'supervised learning' category. Winston's seminal work, with his ARCH[5] sequence of positive and negative examples, demonstrated how a model of a concept may be built up through a carefully prepared training sequence. A model is formed of the initial arch presented. The first negative example differing only in that the supporting blocks are together, leads the program to the idea that the blocks must have a gap between them. Further examples enhance the description. A further consequence of this work is the reliance on a protocol between student and teacher. The student may suppose examples are relevant, and negative examples are given with some purpose. The learning process may break down severely if the student fails to understand the teacher's thinking behind any example.

Along the same lines, Michalski[6] provides his INDUCE system with a set of examples and some background knowledge. The language used is an extended form of predicate calculus, and various rules are applied to extract a description of the concepts represented. Again though, a lot of work must be done by a teacher to get the material into the correct form, and some knowledge cannot be represented in predicate calculus.

'Unsupervised learning' is a much harder proposition. The creative demands requiring a substantial background knowledge, which it is difficult to provide a computer with. It is impossible to draw the line between making explicit relations already present in the data set, and actual creativity, especially with a computer, dependent on the data given to it. Even so, Lenat's AM and EURISKO use various heuristics and 'interestingness' criteria to generate concepts in areas of mathematics and spaceship design.[7]

The ARCH and INDUCE systems each use the supplied examples to guide a search through the possible descriptions. The assumption being that a suitable pattern or arrangement of the internal symbols will perform the desired task. Lenat's systems use instead more abstract constraints on the arrangement of knowledge inside the system to formulate a model.

None of these attempts at learning has come close to the adaptability of the human. It is in an effort to directly model the ability of the human brain that the 'artificial neural network' was developed. Data is represented directly as a vector of numbers, as may come from a sensory device or go to a motor. Any ideas of forming a model of the learning activity and the knowledge being acquired are rejected. Instead, learning is assumed to be a property of the machine architecture itself, and so the problem becomes more one of guiding the learning in the appropriate direction. This architecture is modelled on the neural network which forms the major substance of our own brains.

A neuron is seen as a unit with a set of input links from other units, and a set of output links, to other units. Each unit may perform a computation, typically a summation of the input links, and often with some threshold applied to limit the value of the output. The links themselves have a weighting, so that they pass on a certain percentage of their input value to the next unit, in either an excitatory or inhibitory sense.

This can be compared with a real neuron, which may have of the order of ten thousand con-

[3]Databases may be programmed by laying out fields on a screen, with a point-and-click interface being used to set up data dependencies and output layouts.

[4]The sequence of movements the robot should perform may be 'walked through', by physically moving the end effector perhaps. The robot will then replicate these movements to perform the task.

[5]Described fully in 'Learning Descriptions from Examples'.

[6]Michalski, Ryszard S., 'Pattern Recognition as Rule-Guided Inductive Inference' *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 2, no. 4, 1980.

[7]For example, Lenat, Douglas B., 'AM:Discovery in Mathematics as Heuristic Search,' in *Knowledge-Based Systems in Artificial Intelligence*, edited by Randall Davis and Douglas B. Lenat, McGraw-Hill, New York, 1982.

nections to other neurons. Signals are propagated down the linking axons by a chemical process, dependent on concentrations of ions inside and outside the cell. Neurons link at synaptic junctions, where about fifty types of transmitter chemical have various tasks to play in conveying the signal. The chemical constitution of the material supporting the neurons is vital, demonstrated by the taking of drugs, which affect the efficacy of signal propagation, in the treatment of certain psychological maladies.

The actual learning task that is generally presented to an artificial neural network takes the form of a vector of numbers. The network performs a computation on this input, and results in an output value. In supervised learning this is compared to the desired output, and then the internal parameters of the network will be altered, so as to bring the computation closer to that required. Therefore, the teacher need have no idea of the desired function to model, or indeed whether there is a suitable function, but allow the network to find its own 'solution' to the problem of matching up input-output pairs.

This process is still one of search, whereby units and links alter their behaviour in response to local conditions. Globally, it is assumed that many such changes will finally result in a network performing the desired function.

2 History and Other Architectures

Neural network research has a long history, with many fundamental ideas dating from McCulloch and Pitts in 1943. These ideas will be introduced in the following examples,[8] along with a brief indication of historical placement. Mathematical definitions will be used for precision where necessary, although it should be realised that they are used to make the computations explicit and are not necessary for a complete understanding. Considerable variation is often possible in the set up of these networks, and should perhaps be encouraged in particular application areas.

2.1 Abstract Theory

The next few sections will see some mathematics to give a formal definition of the various functions. As regards implementing the networks, very little understanding is required, beyond being able to get the function computed. However, in a very real sense, these networks may be treated as classes of object amenable to mathematical analysis. The definition, operation, and resultant performance may all be formally described. It turns out, as will be described later in this section, that many types of network are examples of a more general class. This kind of work can be very useful in elucidating just what the different networks are capable of, and perhaps suggest new designs. Here, only a brief guide to the more formal side is possible, and to this end the mathematics is restricted.

An artificial neuron is a relatively simple object, being a *unit* possessing some *links*. These links carry signals between the units. It is possible to divide links into bidirectional types, which can pass a value in either direction, or unidirectional. Only the latter will be dealt with though, as a bidirectional link may be modelled by a pair of unidirectional links, working in opposite directions. Thus, the unit has a number of incoming links, I, and a number of outgoing links, J. Each unit is labelled with a unique number in the network, purely for descriptive purposes. Associated with each link is a real number, w_{ji}, where the j indexes the number of the unit the link is pointing to, and the i the unit the link is coming from. The set of w_{ji} is termed the set of *weights* of the network. If the originating unit i of a link is generating a value a, the destination unit j will receive a value $a \times w_{ji}$.

The units themselves can perform a range of computations, but generally they will begin by summing the values on the incoming links. This figure being termed the unit's *net input*. The second operation generally takes this net input and transforms it into the output value of the unit. This operation is termed the *activation function*.

[8]The architecture summaries are based upon lecture notes provided by W.B. Yates, 1994

One of the earliest activation functions,[9] being the *threshold function*, which outputs a one if the net input exceeds the *threshold parameter*, generally denoted θ, and a zero otherwise. This simplicity belies a latent ability, as McCulloch and Pitts also showed that networks constructed using such functions are capable of computing any Turing computable function.

In spite of this, a more common activation function is the sigmoid function. This being continuous, differentiable, and non-linear. Its definition being

$$sigmoid(x, \theta) = \frac{1}{1 + e^{-\theta x}}$$

where the parameter θ is called the *gain*. It should be noted that, as θ is increased, the sigmoid function tends towards the threshold function.

The units of the network are described in one of three distinct ways. The *input* units possess only output links. Their output value being the value of the input at that unit and time. *Output* modes possess only input links. Often they have an identity activation function, leaving the result of the net function as the output at that unit. These two types of unit form the visible part of the network, accessible to the user or world. Other units are termed *hidden*, and their links may run between input, output or other hidden units. A special kind of hidden unit is also often seen. This is the *bias* unit, which functions rather like an input unit, only its value is always 1.

Given the number of units present in the network, the architecture itself is determined by the numbers and connections of the links. Considering the units themselves, in the limiting case, every unit may have an input link from every other unit, including itself. Thus, for n units, a weight value w_{ji} exists for every i and j in the range $[1...n]$, a total of n^2 weights. These may conveniently be represented in a *weight matrix*, such that the weight w_{ji} appears in column i, row j.

A fully connected network like this is extremely hard to analyse, and for most purposes simpler networks are considered. These are formed by removing the links not needed, for instance the links between a unit and itself. This is done by placing a zero in the appropriate position of the matrix. Setting $w_{ii} = 0$ for all values of i would accomplish the previous suggestion of removing all links between any unit and itself. In this manner, the popular feed-forward networks described below are seen as a sub-class of fully connected architectures.

The actual weights on the links are formed by training the network on sets of examples which represent the task to be performed. In the unsupervised case, the examples must be grouped into categories revealing some underlying pattern of organisation. With the supervised case, the examples become input-output pairs, represented $(a_1, a_2, ..., a_I, b_1, ..., b_J)$ where I and J denote the numbers of input and output units respectively. If it is the case that $I = J$ and $a_i = b_i$ we have a special case of learning, termed *auto-associative*, generally used for associative memories. This is the subject of the first example below. The *hetro-associative* case is the more usual, where the input and output differ, and the network is trained to compute a function between the two. This is the subject of most of the rest of the examples.

2.2 Correlation Matrix Memories

A correlation matrix is a form of *associative memory*, which is capable of returning a piece of data given a portion of its key. The simplest (useful) kind of neural network models this type of memory. A set of input units is connected to a set of output units, where a link connects every input unit to every output unit. (It is possible to leave out some links, but the fully connected case is the most general.)

If we have I input units, and J output units, then input keys of up to size I may be placed into the input units. Every link has a weight value, w_{ji}, from input unit i to output unit j. The output unit computes the weighted sum of its inputs: if a_i is the value on input unit i, the output value on unit j will be:

[9] McCulloch W. S. and Pitts W., 'A Logical Calculus of the Ideas Immanent in Nervous Activity', in Anderson, J. A. and Rosenfeld, E., editors, *Neurocomputing: Foundations of Research* MIT Press, 1943, pp. 18-27.

$$\sum_{i=1}^{i=I} a_i w_{ji}$$

A model of supervised Hebbian learning (also termed Widrow-Hoff learning) is used to adjust the values of the weights during training. Initially weights are set to arbitrary values. A series of input-output pairs are presented to the network. The network computes an output for the given input, and this is then compared to the given output. If the input is represented as $(a_1, a_2, ...a_I)$ and the desired output as $(b_1, b_2, ...b_J)$, the weight update is given by:

$$\Delta w_{ji} = \eta a_i b_j$$

where η is a real valued *learning coefficient* and is set as a parameter of the network.

Over several presentations of these training patterns, the difference between the computed and desired output will be reduced. Learning is often stopped when this falls below some threshold value, 95% for instance, for all the training examples. For the correlation memory, the output is desired to be the same as the input. This enables its use as a signal 'purifier', where a corrupted instance of a trained example is returned to its original state.

This architecture is limited, in that it can only be trained on orthogonal patterns, that is, patterns which do not overlap. However, it does demonstrate some of the properties of neural networks. For instance, the ability to work on an imperfect example, $(0.1, 0.9)$ may be as good as $(0, 1)$. It is even possible to 'damage' the network in some way, by removing some links after training perhaps, and still have a system that works more or less as it did before, depending on the amount of damage, of course. These are both attractive properties, conspicuously absent from standard computer programs with their rigid input and propensity to crash at the slightest provocation, and is derived from the distributed representation of information shared by most neural network architectures.

2.3 Perceptrons

The perceptron was formally analysed by Minsky and Papert,[10], and is another early form of neural network. The architecture is a simple one, linking a series of input units into a single output unit. The input units differ from before in that they compute partial functions on the input vector. For example, given a square array of input values (as in a retina), the first unit may compute the logical 'and' of a vertical line, the second some function of a diagonal line. These units thus compute the existence of certain features in the input. In the case of a retina, these may be arranged at random across the entire image. The perceptron itself will then consider the utility of each of these features in the recognition of the target image. The output unit itself using a threshold value on the weighted sum of the values from the input units. Thus, the perceptron determines whether the input is part of the class it is trained to recognise, or not. The learning rule is as follows:

$$\Delta w_i = a_i \times (a_i - b)$$

where a_i is the input on unit i, and b is the desired output for this pattern.

It may be proved that assuming linear separability[11] of the input classes, the perceptron is guaranteed to converge on a set of weights which will correctly classify them.[12] Unfortunately, simple functions, such as the logical exclusive-or, are not linearly separable, and so the class of functions which may be trained in this manner is limited.

An interesting extension of this is formed by cascading perceptrons, so that the outputs of one bank, which perhaps have been trained to identify vertical or horizontal lines, feed into a further perceptron. This may be trained to recognise an image composed of those features, rectangles or

[10]Minsky M., and Papert S., *Perceptrons.* MIT Press 1969

[11]That is, plotting the points on a graph allows a straight 'line' to be drawn between the two classes. It should be noted that I-input units require a graph in I dimensions, and thus the 'line' is the appropriate hyper-plane of dimension $I - 1$.

[12]Alternatively formulated as 'the examples of the class to be recognised should be linearly dependent'.

triangles for instance. The classic biological example is the cat's retine, where work has shown the existence of such feature detectors.

2.4 Back Propagation

This is the most widely known, and most influential, model of learning for neural networks. In providing a mechanism by which a network may be trained to match non-linearly separable functions, Rumelhart and McClelland[13] brought neural networks into the mainstream of learning research for the '80s.

For definiteness, consider a network consisting of a layer of input units, a layer of output units, and one or more layers of hidden layers between them. Each layer is assumed to be fully connected to the next. A bias unit is assumed to be present, so that each hidden and output unit has a link from it.

Units compute the net activation of their input links, as before, and the sigmoid activation function is used, with a threshold of 1. One property of the sigmoid function,

$$G(x) = \frac{1}{1+e^{-x}}$$

not mentioned before and vital to the efficiency of learning, is that its gradient function, or differential, has a particularly concise expression:

$$G'(x) = G(x)(1 - G(x))$$

It remains only to specify the manner by which the weights of the links are updated during an input-output presentation for training.

Having presented an input, and computed the activation for all units, that of the output units is compared to the desired output. What follows is a method of 'tweaking' all the weights to guide the output towards what is required. Functional dependence of the units leads us to consider the output units first. Each unit has a certain 'error' value, dependent on how well its activation matches the desired output. Considering the weights and activation of all units input to it, a desired change can be computed. For instance, if our output is too high, all high inputs should be decreased in value. Therefore, those weights are decreased. This desired change is used in computing the error in output of the inputting unit. This propagation enables a similar process to be considered throughout the network, until the input units are reached. Hence the term 'back-propagation'.

Mathematically, the error in output unit j is,

$$\delta_j = (y_j - o_j)G'(net_j) = (y_j - o_j)o_j(1 - o_j) \qquad (prop.1)$$

where y_j is the desired output value, o_j the actual output, and $G'(net_j) = G(net_j)(1 - G(net_j)) = o_j(1 - o_j)$ is used to scale the difference.

The errors in the hidden units are given by:

$$\delta_j = G'(net_j)\sum_{k=1}^{J}\delta_k w_{kj} \qquad (prop.2)$$

The summation being a 'backwards' net function, summing the errors in the relevant output unit scaled with the weight to that unit.

The update to a weight w_{ji}:

$$\Delta w_{ji} = \eta o_i \delta_j \qquad (prop.3)$$

where η is a coefficient of learning, set as a system parameter.

Using these relatively simple equations, with a bit of care in a recursive structure, the basic back propagation algorithm may be used for training various functions. Typically, a set of input-output

[13] See for example, Hornik, A. K., Stinchcombe, A. M., and White, H., 'Multilayer feedforward networks are universal approximators'. *Neural Networks*, 2(5):369-366, 1989.

pairs is presented to the network in a continuous cycle until performance is within a certain tolerance. The network may then be used to classify some data previously unseen.

One small variation, which should be pointed out, is the distinction between on-line and batch processing. In on-line processing, the computed updates to the weights are applied immediately, in preparation for the next input. In batch processing, all the updates are retained in memory until the end of the training cycle, and applied 'en masse'.

The problem with back-propagation as a method of training a network to perform a function is in its learning method. In being so general, it has sacrificed a lot of speed. The value for the learning coefficient η should be made as small as possible, so that the network weights do not overstep their optimum values. However, this means that the network takes a long time to converge on a solution. The algorithm has in fact been subject to a considerable amount of variation and addition. Many of these work from the following useful analogy.

The network's weights may be considered as a long vector of real numbers, ie: $(w_{11}, w_{12}, w_{13}...)$. Assuming a static training set, a function may be considered which produces the number of errors on that training set, of the network with given weights. Imagining the weights as points on a plane, the error function produces a surface above that plane, with the height at a point being the number of errors of the associated network. (The error surface so formed will be a 'rough' one, as the error values are discrete, and may change unpredictably with small alterations in any of the weights.) Although this surface could never actually be computed, as the set of all possible weights is formed from several sets of real numbers, it is possible to recognise certain local properties and use these to guide the learning process. It should be recognised that the object of the learning is to minimise the number of errors, or reach the 'lowest' point on the surface, and hence is a form of search in an unknown domain. Potential problems, like an inverted form of 'hill-climbing' suggest themselves, as our algorithm may find a local minimum instead of a global one. We may also be lucky, in having several global minima to aim for.

Some of the slowness of back-propagation may be seen as the working about this surface of the network weights. If we could plot the steps taken, they would resemble zig-zag motions down a valley, as opposed to direct motion down the centre. Various plateaux with very slight slopes in the correct direction, will also require time to traverse, as the correct direction to travel is unclear.

For the first, and virtually essential, change, we add to the analogy the concept of a 'ball' representing the point we are at on the error surface due to the current configuration of weights. If the ball is heading in the right direction, downhill, it should tend to accelerate in that direction. Once the target, or lowest point, is passed, the ball will tend to slow down, as it climbs the opposite slope, and be pulled back to the minimum. Frictional damping ensures the ball will not continuously oscillate about any point. Using this analogy fairly directly, we add a term to the learning step of the algorithm. Equation (prop.3) now becomes:

$$\Delta w_{ji}(t+1) = \eta o_i \delta_j + \alpha \Delta w_{ji}(t) \qquad (prop.3a)$$

where, $(t+1)$ and (t) indicate the time step at which these refer. That is the new change in the weights refers to the old change in the weights. The new constant α is the momentum coefficient, and is set between 0 and 1.

It is possible to see how this relates to the analogy just given. A proportion of the old change is added into the change currently contemplated, so that continuing to move in the same direction will cause a larger jump. Similarly, trying to suddenly change direction would be damped, leading to a smaller jump. Making the momentum coefficient smaller than 1 provides for the frictional damping referred to. As the weight oscillates over the desired value, smaller and smaller changes will occur, until it finally converges.

Momentum leads to dramatic improvements in the algorithm's speed of convergence, and allows larger values of η to be used from the beginning.

Another improvement, related in intent, is to alter the learning coefficient η during training.[14] Every weight has an associated learning coefficient, so that if the weight is changing in the same

[14]Acredited to Almeida & Silva, 1989.

direction as previously, then η is increased, made 1.1 perhaps, whereas changes in direction might decrease this to 0.8.

Other methods include 'Conjugate gradient descent', which gives each weight its optimum value in turn by conducting a non-local search along its line of variation, and 'Quasi-Newtonian Methods'.[15] These last attempt to gather information on the error surface by preserving the gradients in memory. By assuming a particular type of surface, for instance a quadratic one, weights can be adjusted to end up directly at the minimum in one step. These tend to assume batch processing methods of learning, but can be an order of magnitude quicker.

2.5 Radial Basis Functions

The class of radial basis function presented here is described in Moody and Darken,[16] and is a specific example from a set of Generalised Single Layer Networks.

The network has the familiar three layers. The input layer feeds into a hidden layer of radial basis functions (RBFs). The number of these must exceed the number of inputs. These, along with a bias unit, feed into a solitary output unit.[17] The output unit performs the familiar weighted summation on the outputs from the hidden layer, and a scaled form of this sum is presented as the output.

$$output = \frac{\sum \phi_i w_i}{\sum \phi_i}$$

where ϕ_i denotes the output of hidden unit i. The divisor here ensures the output is in the range of $[0, 1]$.

ϕ_n, where n is the number of hidden units, is denoted the bias unit, and its value is always one. As indicated before, if r is the number of input values, $n > r$.

Each RBF is set up to perform a gaussian response to its r inputs. Each RBF possessing a centre, c, and a radius, d. Thus,

$$\phi(x) = exp(-\frac{||x - c||^2}{d^2})$$

where $||x - c||$ is the Euclidean norm, or distance between the points x and c, in r-dimensional space.

In training, the RBFs have their centres set by the k-clustering algorithm.[18] The points matching to each unit's centre having a range of distances from it, their standard deviation is taken. A plausible rule being to set the distance parameter of the RBF to twice this value.

The weights to the output unit may then be trained using any favoured method, such as Widrow-Hoff gradient descent.

The actual operation of this class of network is clearer cut than that of back-propagation. It is in some senses a brother of the perceptron. The perceptron produces a list of features, present or not, in the input, which are then fed via weighted links to the output node. The RBFs produce a somewhat similar computation, but the features now are proven clusterings of the training data. Each RBF calculating how well the current input matches one of the learnt features. The weighted links indicating the importance of each cluster in the recognition of the class. The accuracy may not always be up to that of back-propagation, but the learning can be more efficient.

Due to this statistical stability in feature selection, it is possible to predict performance values for the network before training begins. For example, given a finite set of data, we can indicate how many would be needed for training so as to ensure a certain probability of success on the full set (this assumes a set of weights can be found to compute the training data correctly enough). This is extremely important, as nothing at all like it exists for back propagation, and opens the way to treating the network as a reliable source of results.

[15] Scott Fahlman, 1988, and Quick Prop.

[16] Moody, J. & Darken, C., 'Learning with localised receptive fields' in Touretzky, T., Hinton, G. E., & Sejnowski, T. J., (eds.), *Proceedings of the 1988 Connectionist Models Summer School*, pp. 133-143, Kaufmann, 1988.

[17] As with Perceptrons, it is possible to add more output units, each with their own set of weights linking the hidden layer. In this case, everything works as described, simply repeated for each output unit.

[18] Also in Moody and Darken.

2.6 In Summary

This has been a brief look at certain architectures of neural network, and there are many others which may be used in a given task. There is no theory of which type of network is 'better', and impossible to determine the precise characteristics of, say, a back-propagation network for a particular job. Given this, it is necessary to experiment to provide the best results from a set of data. In fact, sometimes it is better to ackowledge the limitations of a particular method, providing it has some strengths, and use some cooperation to provide global performance.

If we view our trained network as providing a partially completed model of a function we want to be computed, it is possible to analyse the degree of partiality. For instance, a back-propagation network may provide a 95% match on the training set, and with a different set of starting weights, or hidden units, it may match to some other degree. The same may be true if we use a Radial Basis Function, or some other variant. If we look at our group of trained networks, we may find that where one fails to match the training set, another succeeds. In fact, by overlapping the performances of the networks we may be able to model the desired function to a very high percentage. It is possible to envisage a cluster of such networks being provided with some 'job manager' to control which of the various outputs are used, and the whole performing the required computation.

The turning of neural network design into a precisely engineered software tool is a subject of research, and developments are likely. As part of this, the limitations of the networks will have to be extracted and clarified. In particular, the actual conditions under which learning can occur. None of the networks presented so far performs what may be called 'real-time learning'. That is, each has to be given a training set to work upon before it is ready to tackle the job in hand. Once this learning stage is completed, no more learning can be carried out without the severe danger of losing all that has been gained.

A further problem with the back-propagation network is that under some conditions learning may not occur at all. Carpenter and Grossberg[19] have shown examples of a network in which only four training patterns, presented cyclically, will cause network weights to change continuously, never converging.

Given all these difficulties and uncertainties it is evident that some care must be exercised when using a neural network. We now turn to a deeper consideration of a network which answers some of these.

3 Adaptive Resonance Theory

The starting point of this theory derives from the dilemma of real-world learning. There is a conflict between being plastic enough to absorb new information, and yet stable enough to retain existing memories. This is not often a problem, as a network can be trained upon a static test set, and later use its generalisation capability in future tasks with no future learning anticipated. In modelling a network which could perform a useful task as well as learn in response to real-world data, which is always changing and never repeats, this dilemma takes on a central role.

An ART network maintains the necessary adaptivity in the light of new experience, whilst retaining knowledge already gathered. If current input accords with previous experience, the relevant category unit will 'resonate', and so dominate the activity of the network. Otherwise the network will 'adapt' an unused part of memory to record the new data.

We begin with models of unsupervised learning, based upon the adaptive resonance theory. They take as input a stream of patterns, and cluster these into categories. There is no suggestion during training of the kinds of categories which would be of most use, and so this form of learning is of the unsupervised type.

The definitions of these systems are largely based in mathematical models composed of differential equations, feedback loops and an assumption of a real-time environment. This is due to their formulation as models of neuronal activity in living brains, so that physical plausibility is paramount.

[19]Carpenter, G., and Grossberg, S. 'Neural dynamics of category learning and recognition: Attention, memory consolidation, and amnesia', in *Brain Structure, Learning and Memory* (AAAS Symposium Series), eds. J. Davis, R. Newburgh, and E. Wegman.

However, with a serial computer working in discrete time steps, convergence solutions simplify many groups of equations and the delicate control strategy emerging from the full implementation may be forcibly imposed on the system.

ART-1 deals with binary data only. The later developments include ART-2 to handle real-valued input data, with its faster cousin, ART-2a, and ART-3, which takes into account the physical chemistry of real neurons and their synapses. ARTMAP itself is derived from two of the unsupervised networks, and is a supervised learning system.

3.1 ART-1

ART-1 is composed of two distinct layers of units with sets of links progressing in each direction. The initial layer is the input layer, and takes the binary vector to be learnt. The top layer is the set of category units. Bottom up links from the input to the category units hold weightings, determining the degree of fit of the input to each category. Top down links from the category to the input units hold the definition of the category, in terms of its accepted input vectors. Control is modulated with a vigilance parameter, which defines the degree to which an input must match the definition of a category for it to be admitted as belonging to that category.

Initially, the category units are said to be *uncommitted*, and all their definitions, the top down links, hold the most general description, being all set to one. When an input vector is presented, the activation of each category unit is computed as the dot product of the bottom up weighted links with the input. The category node with the maximum activation dominates and passes its definition, via the top down links, for consideration against the input. A measure of similarity is computed, measuring the degree to which the input matches the category. If this is above the level set by the vigilance parameter, the category unit is valid, and is said to 'resonate'. If it is not valid, the category unit shuts off for this input presentation, and the unit with the next highest level of activation now dominates, and is checked as before.

A valid unit will now begin to learn, by modifying the bottom up and top down links in light of the current input. The definition of the category will be contracted, by changing to the logical and of the input and previous definition. Thus, only bits set in both will be retained. This principle means the definition will only retain data which is true for all examples. The bottom up weights are modified in a fashion based upon this. Once a category unit has done some learning, it is said to be *committed*, and it is the committed units which hold the knowledge of the network.

The structure of the learning cycle is such that many similar patterns will become grouped together, and the resultant category definition should be able to generalise over future examples. This will work along with providing a unique definition for any unexpected patterns, which demand their own category definition. Whether this was the first or last pattern, future learning will not corrupt its definition.

The full model of ART, based upon a consideration of neural activity, provides for rates of learning and committal. A neuron may modify its weights in a slow fashion, or a fast fashion. Typically, it is considered that short stimulations of a neuron will not provide it with enough time to learn new behaviour. Committal is said to be slow, and is relied upon to allow for deactivating categories and seeking for new ones without effecting those considered. The serial computer variant described below will perform its learning in one step, that is fast learning.

3.2 ART-2a

ART-2a is an extension of ART-1 to handle real-valued inputs. The internal structure and the learning pattern is similar, with the equations being altered to reflect the difference in data.

The input vector is initially normalised, by dividing each component by the Euclidean norm. This gives it a length of one. A further transformation removes each component which is below a certain threshold value. This vector is again normalised, and the result is passed to the network as input.

The bottom up weights now take on responsibility for holding the category definitions, and there are no top down links. Resonance will occur if the product of the weights and input is greater than the vigilance paramater. Learning occurs by altering the weights in some proportion of the previous values and the input ones. This proportion is a parameter of the network and determines how quickly the network will learn from a new input. If this proportion only allows a small change given a new input, the learning rate is slow, as the category will only gradually ‘drift’ over to a new definition. Faster learning is possible, but at the risk of losing previous results due to a spurious example. In both cases committal is fast, as the computations take immediate effect.

3.3 Supervised learning: ARTMAP

ARTMAP is a composite architecture, linking a pair of ART architectures so as to produce an adaptive supervised learning system. Essentially, one ART system is presented with the input data, and operates as described above, classifying these into categories. The other looks after the output data in an identical fashion. The mapping system will consider the categories generated by a given input-output pair, and assume these should be associated. If not already present, this input category will be irreversibly linked to the output category. In future, if this input category is selected in the absence of any output data, the system can predict the desired output category.

If, however, a given input-output pair violates a pre-existing link of the input category to another output category, the system must make some alterations to its knowledge base. To do this, the vigilance parameter controlling the degree of fit of an input example to a category is altered so that the given input is forced to match a *new* category. This event is termed *match tracking*, and provides ARTMAP with its flexibility in learning new categories. The mapping system may then again attempt an association between input-output categories.

ARTMAP inherits from the ART modules many features of stable learning, but with the match tracking it supplies an ability to focus on rare occurrences in the data set. If we present a series of similar input vectors with identical output, the ARTa module will group the input together in one category. If we then present an isolated exception to this which, although basically matching the main group, has some small difference and a different output, match tracking will force this input to be reclassified on its own. We may then see two classes, the main, default class, which catches most of the examples, and another, exceptional class. Most of the generalisation is formed from the main class, but many examples are caught by the exceptions, thus increasing the performance of the network.

4 Details of Implementation

4.1 ART-1

There follows a concrete definition of an ART-1 implementation, which may be found in the appendix. This is a synthesis of the two presented by Grossberg and Carpenter in their papers. It should be noted that there is nothing absolutely rigorous about ART-1s formulation, as it may be approached from several perspectives. The authors describe it as ‘something more than a philosophy, but much less concrete than a computer program’, and so implementations may vary in their detail.

The algorithm, implemented in ANSI-C, may be found in the appendix. The equation numbers appear within the listing for reference.

INITIALLY

All category units are set as uncommitted.

The bottom-up weights are initialised to small values, so that, for each category, all its input weights equal a value α_i,

$$0 < \alpha_i < 1/(\beta + |I|) \tag{1.1}$$

where, $\beta > 0$, $|I|$ is the maximum modulus of input.

The value for every top-down value is initialised to 1.

$$topdn[j][i] = 1 \tag{1.2}$$

ACTIVATION

When an input is presented, all category units become active for the current presentation. Every category unit's activation is computed as:

$$act[j] = \sum_{i=1}^{No.Inputs} Input[i] \times botup[i][j] \tag{1.3}$$

SEARCH AND RESONANCE

A category unit j is selected which is currently active, and possesses the highest activation of all active category units.

Validity of committed category units is checked by computing,

$$|I \cap topdn[j]| < \rho|I| \tag{1.4}$$

uncommitted category units are always valid.

The left term of the inequality is the number of bits set in both the input and the selected category's definition. If this is less than a given percentage (governed by ρ) of the number of bits in the input, the category selection is deemed invalid.

If the category selection was invalid, the unit will be flagged as inactive during this input presentation, and a new unit will be searched for.

If no valid unit exists, the categories must all be committed and none match the input sufficiently. In this case, the categories are deemed saturated, as there is no room to learn a new category, and the next input is considered.

Assuming a valid category was found, the learning stage is invoked.

LEARNING

In fast learning, the top-down and bottom-up weights need adjusting, and the category unit must be flagged committed, if not already so.

The top-down values, or category definitions, are altered so only bits set in both are retained.

$$topdn[j] = I \cap topdn[j] \tag{1.5}$$

The bottom-up weights assume the above has taken place,

$$botup[j] = topdn[j]/(\beta + |topdn[j]|) \tag{1.6}$$

4.2 ART-2a

ART-1 was extended to deal with real valued input in ART-2.[20] The algorithm was enhanced in the development of ART-2a,[21] principally by making explicit the results of certain groups of formulae.

A summary of the ART-2a algorithm, taken from the latter paper, follows. The structure is identical to that of ART-1, only additional parameters and new equations encode the real values. Similarly, the equation numbers are referred to within the program listing.

[20]Carpenter, G. A., & Grossberg, S. 'ART2: Self-organisation of stable category recognition codes for analog input patterns.' *Applied Optics*, **26**, pp.4919-4930.

[21]Carpenter, G. A., Grossberg, S., & Rosen, D. B., 'ART-2a: An Adaptive Resonance Algorithm for Rapid Category Learning and Recognition', *Neural Networks*, Vol. 4. pp.493-504, 1991.

INITIALISE

The bottom up vectors are initialised so that each category's inputs[22] are α, with,

$$0 < \alpha \leq 1/\sqrt{M} \tag{2.1}$$

INPUT

Given a non-uniform M-dimensional input vector I^0, transform it,

$$I = \eta F_0 \eta I^0 \tag{2.2}$$

where $\eta x = \frac{x}{||x||}$ and F_0 applies a threshold function to the bits of its input. The threshold, θ is in the range $0 < \theta \leq 1/\sqrt{M}$. Note that from this I will be non-zero and normalised.

ACTIVATION

As before, take the dot product of the input with the bottom up weights, to produce the activation level of each category.

$$act[j] = \sum_{i=1}^{No.Inputs} Input[i] \times botup[i][j] \tag{2.3}$$

SEARCH AND RESONANCE

Select from the active categories that with greatest activation. Validity is checked by comparing it directly to ρ.

$$act[j] < \rho \tag{2.4}$$

is a correct inequality for an invalid category. As with ART-1 the unit is then ignored for this input presentation, and a new category is searched for. Similarly, saturated categories lead to the input being ignored.

LEARNING

$$botup[j] = \begin{cases} I, & \text{if } j \text{ is an uncommitted node} \\ \eta(\beta\eta\Psi + (1-\beta)botup[j]), & \text{otherwise} \end{cases} \tag{2.5}$$

where β is a learning coefficient, between 0 and 1 and Ψ is defined:

$$\Psi_i = \begin{cases} I_i, & \text{if } botup[i][j] > \theta \\ 0, & \text{otherwise} \end{cases} \tag{2.6}$$

[22]The closer α is to the higher value, the more likely an uncommitted category will be selected for. Hence, the more categories will be formed. If α is close to 0, committed categories will be more readily selected. This result is true also of ART-1.

4.3 The MAPping section

Two ART networks are used, with a mapfield between them. The mapfield encodes a function, which, when resolved, identifies a single category from the first ART module, ARTa, with a single category from the second, ARTb. The ARTb architecture works without modification, except it must return the category that it selects, with a special code for saturation. ARTa works similarly, except provision must be made for match tracking. Firstly, the network is split into two. The first finding a suitable categorisation, the second performing the learning. This is a natural split, due to the composition of ART. Secondly, the similarity measure computed by ARTa must be accessible to ARTMAP, for it to force the match tracking.

It should be noted that as ARTMAP only requires the numbers of the categories selected by ARTa and ARTb, these may work with either binary or real-valued data, and may indeed be mixed. The present implementation only utilises ART-1 and binary vectors.

Initially, the mapfield is set up with values indicating nothing learnt. With a presentation of input, each ART module is called, with the returned category recorded. Behaviour now diverges depending on the value of the mapfield. If the category returned by ARTa has not yet been learnt, the mapfield now learns the given pairing.

$$mapfield[category_a][j] = \begin{cases} 1, & \text{if } j = category_b \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

If the category has been learnt, and agrees with that returned by ARTb,

$$mapfield[category_a][category_b] = 1 \tag{3.2}$$

then we have a correct prediction.

If this is not the case, then match tracking must ensue. We now set ρ_a to a value where it discriminates between the original definition and the input.

$$\rho_a > \frac{|I \cap topdn[category_a]|}{|I|} \tag{3.3}$$

The cycle above is now repeated with this new value of ρ_a altering the classification in ARTa. If this value exceeds one, then ARTa will be unable to return a new category, and will provide a 'saturated' message.

If saturation has not occurred, of either ARTa or ARTb, then learning of ARTa will occur, and the next input considered.

Provision in the implementation has been made to allow the program to save off its learnt weights, and reload them at the beginning of a cycle. This allows learning to proceed incrementally, as more of the database is gathered, and also is the means by which generalisation to a new data set is performed.

4.4 Comments on Design

Considering the checking of validity in ART-1, from equation (1.4), a measure of similarity between the learnt category and input vector is computed. This measure,

$$\frac{|I \cap topdn[j]|}{|I|}$$

is asymmetric between the new and learnt vector. For instance, a category definition of 1 1 0 0 0 confronted with an input of 1 1 1 0 0 generates a similarity of 2/3, whereas a category definition of 1 1 1 0 0 confronted with an input of 1 1 0 0 0 produces a similarity of 1.

As a consequence of this, the architecture is sensitive to the order of presentation of the vectors. A blank ART-1 network given its first input of 1 1 0 0 0 will create this as a category definition. The second input of 1 1 1 0 0 may match, if the vigilance parameter is less than 2/3. If not, then a second category will be created. If instead, the first input vector is 1 1 1 0 0, again a category definition is created on this vector. When the second input of 1 1 0 0 0 is presented, regardless of the vigilance parameter, it will be matched to the first category. Training will occur, and a single category definition of 1 1 0 0 0 will be left in memory.

The paper on ARTMAP describes this problem in relation to the match tracking that ARTMAP performs when the input vector predicts an output incompatible with the given output. In this case, if the similarity measure is 1, then no other category unit in ARTa can become active. Hence, the input is ignored, and no correlation is learnt from this pair. The solution is to use complement coding, which extends input vectors to twice their original length by appending the complemented forms. The two vectors above thus become 1 1 0 0 0 0 0 1 1 1 and 1 1 1 0 0 0 0 0 1 1. The similarity measure will always return a value of 16/25, whichever order these are presented in. It should be noted that this solution makes the number of set bits in the input a constant (half the length of the input). This property means that non-identical vectors can never form a similarity measure of 1, so the example of sub-sets above never occurs. This is also important in the ARTMAP architecture, as a similarity of 1 means that match-tracking cannot occur, so that a new category will not become active (nothing can match better than 1).

The problem with this solution is that, along with doubling the number of input bits, the storage and time requirements of the network are also doubled. A good alternative would be to alter the measure of similarity, and make it the square of the correlation between the input and category vectors, computed as:

$$\frac{|I \cap topdn[j]|^2}{|I||topdn[j]|} \tag{1.4a}$$

This is a simple extension of the former measure, and in the earlier example will generate a match of 2/3 with either order of the two vectors. Again, the measure will never become maximal unless there is a perfect match between the two vectors. Thus, the match tracking in ARTMAP is assured.

These issues are not important in examples akin to the mushroom database, to be examined below, where the number of bits in the input vectors is kept at a constant. In this case, the modification to the measure has the effect of altering the scaling, but not the order, of the inputs' similarity.

ART-2a has a more rational comparison method, derived from its entire internal representation being as vectors of unit length. The bottom up weights encode the vector defining the learnt category. In activation, the dot product is formed between each of these vectors and the input. As the lengths of all vectors are unitary, this equals the *cosine* of the angle between them. This ranges from 0 for orthogonal, totally dissimilar, vectors, to 1, for parallel, or identical, vectors. This function has the two desirable properties described above, being symmetric, and providing a perfect match only in the case of identical operands.

One property of ARTMAP not addressed in this implementation is the priming of the ARTb, or output, module. Input is presented to ARTa, which forms a category, and the MAPping field suggests an output category. The actual output is then compared with this category, and only if it does not match within the specified vigilance, will ARTb suggest a new category, and match tracking ensure. The current implementation assumes both the input and output arrive together, and the ART modules work simultaneously before the MAP field compares the results. It would be easy to model for the former case with small changes in the code, and this would be desirable when working with large output vectors. The present application of ARTMAP to the mushroom database only requires two orthogonal output classes, and there is nothing to be gained by the change.

A comparison of ART may be drawn with Radial Basis Functions, which use a set of features dependent upon the training data set. Training causing a weighting scheme to be calculated between these features for the desired function. The actual features used providing a degree of justification for any decisions made. ART also justifies its decisions, with the learnt definition of each category. It would be possible to alter the similarity measure in ways beyond that described above, to reflect the type of data under examination.

For example, training on vectors taken from 2D arrays of points, relies on breaking the 2D structure down into a single line of bits. The similarity measures previously described treat the setting of a bit as of equal importance no matter where that bit is located, and for a task like character recognition this is not always true. A new bit extending an existing line would be assumed to be part of that line, and thus not a real difference, whereas a bit placed in a clear area may be part of a character distinction.

As a further example, we may know the features for which the input is a coding. The first seven bits may represent the colour of an object, and one is set depending upon whether the object is red, orange, ..., violet. A red object and a blue one are obviously coloured differently, but are a red and orange one? The range of the spectrum would leave certain possibilities with an unclear classification. The similarity metric employed by ART-1 would make red and blue as distinct as red and orange. A metric using some knowledge of the data may instead assign a certain similarity to the latter case, making it more likely that the two objects would be matched.

The intention is not to go into this in any detail, but show a possible source of future development. Tests should reveal whether such methods produce better generalisation and/or fewer categories.

5 Testing of ART1

5.1 An example of learning

An illustration of ART-1's behaviour is now presented by following through the trace of a small example. Training occurs with input vectors of length three, and two available categories. A vigilance parameter of 0.6 provides 60% matching of the vectors. These are placed into a file **test.inf** in the following form:

```
3, 2, 3, 0.6 .5
```

The three patterns are placed into a file **test.dat**:

```
3
1,0,1
0,1,1
1,1,1
```

The simulation is called from a command line:

art test -t

The initial output consists of some header data, to confirm the network has the correct definitions.

```
****    ART-1 Simulation     ****
**** Peter Lane, April 1995 ****

Reading information file: test.inf.
Reading data file: test.dat.
num_inputs: 3, num_cats: 2, max_mod_input: 3
rho: 0.600000, beta: 0.500000
Number of patterns: 3.
```

The program now picks up the first pattern in the file, 1 0 1, and computes the activation of all its categories. The trace file displays the value of each array before learning occurs, and hence shows their initial values.

```
Pattern: 0
Input[0] = 1
Input[1] = 0
Input[2] = 1
Cats[0] = 0, Comm[0] = 0
Cats[1] = 0, Comm[1] = 0
Topdn[0][0] = 1
Topdn[1][0] = 1
```

```
Topdn[0][1] = 1
Topdn[1][1] = 1
Topdn[0][2] = 1
Topdn[1][2] = 1
Botup[0][0] = 1.116567e-14
Botup[0][1] = 2.815413e-04
Botup[1][0] = 1.116567e-14
Botup[1][1] = 2.815413e-04
Botup[2][0] = 1.116567e-14
Botup[2][1] = 2.815413e-04
Act[0] = 2.233134e-14
Act[1] = 5.630827e-04
Selected node = 1
VALID node!
```

As the activation of node 1 was highest, and it is uncommitted, it is selected. Learning ensues, and the next trace displays the effects, as the top down values for node 1 take on the pattern's values, and the bottom up rates are adjusted accordingly.

The next pattern 0 1 1 is loaded in and considered:

```
Pattern: 1
Input[0] = 0
Input[1] = 1
Input[2] = 1
Cats[0] = 0, Comm[0] = 0
Cats[1] = 0, Comm[1] = 1
Topdn[0][0] = 1
Topdn[1][0] = 1
Topdn[0][1] = 1
Topdn[1][1] = 0
Topdn[0][2] = 1
Topdn[1][2] = 1
Botup[0][0] = 1.116567e-14
Botup[0][1] = 4.000000e-01
Botup[1][0] = 1.116567e-14
Botup[1][1] = 0.000000e+00
Botup[2][0] = 1.116567e-14
Botup[2][1] = 4.000000e-01
Act[0] = 2.233134e-14
Act[1] = 4.000000e-01
Selected node = 1
Selected node = 0
VALID node!
```

Here, we see that node 1 was first selected for, having the highest activation. However, the similarity of the learnt pattern 1 0 1 and the new pattern 0 1 1 is 0.5. This is less than the vigilance of 0.6 and so the network must seek another category node. Hence, node 0 is selected, and this being uncommitted is a valid one.

```
Pattern: 2
Input[0] = 1
Input[1] = 1
Input[2] = 1
Cats[0] = 0, Comm[0] = 1
Cats[1] = 0, Comm[1] = 1
```

```
Topdn[0][0] = 0
Topdn[1][0] = 1
Topdn[0][1] = 1
Topdn[1][1] = 0
Topdn[0][2] = 1
Topdn[1][2] = 1
Botup[0][0] = 0.000000e+00
Botup[0][1] = 4.000000e-01
Botup[1][0] = 4.000000e-01
Botup[1][1] = 0.000000e+00
Botup[2][0] = 4.000000e-01
Botup[2][1] = 4.000000e-01
Act[0] = 8.000000e-01
Act[1] = 8.000000e-01
Selected node = 0
VALID node!
```

The final pattern matches each of the categories equally, and one is selected at random.

```
Outputting committed categories to file: test.wgt
0 0 1 1
1 1 0 1
```

The program finishes by summarising what it has learnt, in terms of the category definitions. As we saw, node 0 took on the second pattern, and node 1 the initial pattern. No change was seen with the third pattern, as it matched to node 0 and included that definition as a subset.

5.2 Demonstration of sensitivity to training sequence

The following sequence demonstrates the sensitivity of the ART-1 implementation to input presentation order. First, ART-1 is run with the standard measure of similarity for a pair of input vectors, reversed in the second case. Note that only one category is learnt in the second case, whereas there are two distinct ones in the first.

```
test.inf:
5, 3, 4, 0.8, .1

text.dat:
2
1 1 0 0 0
1 1 1 0 0

****    ART-1 Simulation     ****
**** Peter Lane, April 1995 ****

Reading information file: test.inf.
Reading data file: test.dat.
num_inputs: 5, num_cats: 3, max_mod_input: 4
rho: 0.800000, beta: 0.100000
Number of patterns: 2.
Outputting committed categories to file: test.wgt
1 1 1 1 0 0
2 1 1 0 0 0

test.inf:
```

```
5, 3, 4, 0.8, .1

test.dat:
2
1 1 1 0 0
1 1 0 0 0

****    ART-1 Simulation     ****
**** Peter Lane, April 1995 ****

Reading information file: test.inf.
Reading data file: test.dat.
num_inputs: 5, num_cats: 3, max_mod_input: 4
rho: 0.800000, beta: 0.100000
Number of patterns: 2.
Outputting committed categories to file: test.wgt
2 1 1 0 0 0
```

The same pair of tests were performed using the modified measure, the square of correlation, as described in the 'Comments on Design'. Note that in each case two distinct categories are learnt, reflecting that the two input vectors are less similar than the vigilance parameter allows.

```
test.inf:
5, 3, 4, 0.8, .1

test.dat:
2
1 1 0 0 0
1 1 1 0 0

****    ART-1 Simulation     ****
**** Peter Lane, April 1995 ****

Reading information file: test.inf.
Reading data file: test.dat.
num_inputs: 5, num_cats: 3, max_mod_input: 4
rho: 0.800000, beta: 0.100000
Number of patterns: 2.
Outputting committed categories to file: test.wgt
1 1 1 1 0 0
2 1 1 0 0 0

test.inf:
5, 3, 4, 0.8, .1

test.dat:
2
1 1 1 0 0
1 1 0 0 0

****    ART-1 Simulation     ****
**** Peter Lane, April 1995 ****

Reading information file: test.inf.
Reading data file: test.dat.
```

```
num_inputs: 5, num_cats: 3, max_mod_input: 4
rho: 0.800000, beta: 0.100000
Number of patterns: 2.
Outputting committed categories to file: test.wgt
1 1 1 0 0 0
2 1 1 1 0 0
```

5.3 Example of Training, showing effect of vigilance

The ART-1 architecture was tested on an input domain of letters, with pixels labelled 0 or 1, from an 8×8 grid. These were input to the network as single vectors of 64 bits, but in what follows these have been relocated on the page to make the pattern obvious. The network was presented with each example, and after each one the current committed category definitions are shown. The cycle was repeated three times, with rho taking values: 0.9, 0.95, and 0.99.

First, these are the examples to be presented to the network:

```
Example 1: C       Example 2: B       Example 3: E

0 1 1 1 1 1 1 1    1 1 1 1 1 1 1 0    1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0    1 0 0 0 0 0 0 1    1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0    1 0 0 0 0 0 0 1    1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0    1 1 1 1 1 1 1 0    1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 0    1 0 0 0 0 0 0 1    1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0    1 0 0 0 0 0 0 1    1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0    1 0 0 0 0 0 0 1    1 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1    1 1 1 1 1 1 1 0    1 1 1 1 1 1 1 1

Example 4: E2      Example 5: E3

1 1 1 1 1 1 1 1    1 1 1 0 1 1 1 1
1 0 0 0 0 0 0 0    1 0 0 0 0 0 0 0
1 0 0 0 0 0 1 0    1 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0    1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 0    1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0    1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0    1 0 0 0 0 1 0 0
1 1 0 1 1 1 1 1    1 1 1 1 1 1 1 1
```

Note the distorted forms of 'E' which provide a test of the generalisation of the category units.

The information file provides for 64 inputs, 7 categories, and a maximum modulus of 32 in the input. Beta is set nominally at 0.1. Rho varies for the following three presentations. The number before the category definition is the category number, and allows the changes in each category to be followed.

```
num_inputs: 64, num_cats: 7, max_mod_input: 32
rho: 0.900000, beta: 0.100000

First Example

4
0 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1

Second Example

3                   4
1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1

Third Example

3                   4
1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1

Fourth Example

3                   4                   6
1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1     1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0     1 0 0 0 0 0 0 0     1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1     1 1 0 1 1 1 1 1
```

```
Fifth Example

3                    4                    6
1 1 1 1 1 1 1 0      0 1 1 1 1 1 1 1      1 1 1 0 1 1 1 1
1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0      1 0 0 0 0 0 0 0      1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0      0 1 1 1 1 1 1 1      1 1 0 1 1 1 1 1

rho: 0.950000

First Example

4
0 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1

Second Example

3                    4
1 1 1 1 1 1 1 0      0 1 1 1 1 1 1 1
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0      0 1 1 1 1 1 1 1

Third Example

3                    4                    6
1 1 1 1 1 1 1 0      0 1 1 1 1 1 1 1      1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0      1 0 0 0 0 0 0 0      1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0      0 1 1 1 1 1 1 1      1 1 1 1 1 1 1 1
```

```
Fourth Example

3                    4                    6
1 1 1 1 1 1 1 0      0 1 1 1 1 1 1 1      1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0      1 0 0 0 0 0 0 0      1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0      0 1 1 1 1 1 1 1      1 1 0 1 1 1 1 1

Fifth Example

3                    4                    5
1 1 1 1 1 1 1 0      0 1 1 1 1 1 1 1      1 1 1 0 1 1 1 1
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0      1 0 0 0 0 0 0 0      1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1      1 0 0 0 0 0 0 0      1 0 0 0 0 1 0 0
1 1 1 1 1 1 1 0      0 1 1 1 1 1 1 1      1 1 1 1 1 1 1 1

6
1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 1 0 1 1 1 1 1
```

```
rho: 0.990000

First Example

4
0 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1

Second Example

3                   4
1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1

Third Example

3                   4                   6
1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1     1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0     1 0 0 0 0 0 0 0     1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1     1 1 1 1 1 1 1 1

Fourth Example

3                   4                   5
1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1     1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0     1 0 0 0 0 0 1 0
1 1 1 1 1 1 1 0     1 0 0 0 0 0 0 0     1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1     1 1 0 1 1 1 1 1
```

```
6
1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1

Fifth Example

2                   3                   4
1 1 1 0 1 1 1 1     1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0     1 1 1 1 1 1 1 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 0 0 0 0 1 0 0     1 0 0 0 0 0 0 1     1 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1     1 1 1 1 1 1 1 0     0 1 1 1 1 1 1 1

5                   6
1 1 1 1 1 1 1 1     1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 1 0     1 0 0 0 0 0 0 0
1 1 1 1 1 0 0 0     1 1 1 1 1 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0     1 0 0 0 0 0 0 0
1 1 0 1 1 1 1 1     1 1 1 1 1 1 1 1
```

It may be seen that too low a value for the vigilance forms too few categories, lumping together patterns that would more naturally be separate, for instance, the B/E merge in the first example. Conversely setting the vigilance too high removes any generalisation at all, and a new category is formed for each input pattern.

6 Testing of ART2a

There follows a test run of ART-2a with a small sample set of three examples. Beta is varied between presentations, to illustrate the effect of altering the proportion by which the new and learnt vectors are combined.

```
test.inf:
4, 3, 0.8, .9, 0.2

test.dat:
3
1,0,1,1
0,1,1,1
1,0,1,0

rho: 0.800000, beta: 0.100000, theta: 0.200000
```

```
Categories:
0 0.600324 0.000000 0.600324 0.528415
2 0.000000 0.577350 0.577350 0.577350

rho: 0.800000, beta: 0.500000, theta: 0.200000

Categories:
0 0.673887 0.000000 0.673887 0.302905
2 0.000000 0.577350 0.577350 0.577350

rho: 0.800000, beta: 0.900000, theta: 0.200000

Categories:
0 0.705887 0.000000 0.705887 0.058713
2 0.000000 0.577350 0.577350 0.577350
```

The first and third examples both become matched to node 0. As beta is increased, the last bit is altered between being close to a one, as in the first example, down to being close to a zero. This controlled proportioning is in marked contrast to the ART-1 results:

```
1 0 1 1 1
2 1 0 1 0
```

where the equivalent category is the second one, and is shrunk irreversably to the smaller of the two vectors.

Something similar may be seen by running the letter data set through ART-2a.

```
let.inf:
64, 7, 0.95, 0.5, 0.1

Categories:
0
0.179605 0.179605 0.179605 0.179605 0.179605 0.179605 0.179605 0.000000
0.179605 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.179605
0.179605 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.179605
0.179605 0.179605 0.179605 0.179605 0.179605 0.179605 0.179605 0.000000
0.179605 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.179605
0.179605 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.179605
0.179605 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.179605
0.179605 0.179605 0.179605 0.179605 0.179605 0.179605 0.179605 0.000000

2
0.202846 0.202846 0.202846 0.100140 0.202846 0.202846 0.202846 0.202846
0.202846 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.202846 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.202846 0.202846 0.202846 0.202846 0.202846 0.000000 0.000000 0.000000
0.202846 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.202846 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.202846 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.202846 0.202846 0.049579 0.202846 0.202846 0.202846 0.202846 0.202846

4
0.000000 0.223607 0.223607 0.223607 0.223607 0.223607 0.223607 0.223607
0.223607 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
```

```
0.223607 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.223607 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.223607 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.223607 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.223607 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.000000 0.223607 0.223607 0.223607 0.223607 0.223607 0.223607 0.223607
```

Notice that categories 0 and 4 encode the B and C as before. Category 2 takes on a compound E shape, where the 0s in the patterns of E2 and E3 have produced a reduction in the weighting at that point, but the spurious 1s have been ignored.

7 Testing ARTMAP

ARTMAP needs testing in two ways. Firstly, it is necessary to check that the ART modules are correctly working independently. This may be done with ARTa by passing a data set which has been previously categorised with a standalone ART module. To accommodate the output though, we need to add a constant output vector to each pattern. This is constant to ensure match tracking never results to alter the behaviour of ARTa. ARTb may be similarly verified, only this time the data set needs some input vectors added, and these may be arbitrary, as ARTb's operation is not affected by ARTMAP. This was done with the letters data set used above, and the weight files compared to check the categories.

A small example now follows to illustrate ARTMAP's match tracking at work. Note the addition of an output line indicating performance so far. This provides information on the 'on-line' learning, and is discussed further in the next section.

```
test.inf:
10, 5, 5, 0.6, .1, 1
2, 2, 1, 0.8, .1

test.dat:
3
0 1 1 1 1 0 0 0 0 0, 1 0
0 0 0 0 0 0 0 1 1 1, 1 0
1 1 0 1 1 0 0 0 0 0, 1 0

****    ARTMAP Simulation   ****
**** Peter Lane, April 1995 ****

Trace Mode.
Reading information file: test.inf.
Reading data file: test.dat.
ARTa: num_inputs: 10, num_cats: 5, max_mod_input: 5
      rho: 0.600000, beta: 0.100000
ARTb: num_inputs: 2, num_cats: 2, max_mod_input: 1
      rho: 0.800000, beta: 0.100000
Number of patterns: 3, Step size: 1.

Pattern: 0.
Cat_a = 4, cat_b = 1.
Mapfield[0][0] = 2.
Mapfield[0][1] = 2.
Mapfield[1][0] = 2.
Mapfield[1][1] = 2.
Mapfield[2][0] = 2.
```

```
Mapfield[2][1] = 2.
Mapfield[3][0] = 2.
Mapfield[3][1] = 2.
Mapfield[4][0] = 0.
Mapfield[4][1] = 1.
Steps: 1, Corr: 0, Incor: 0, Unkn: 1, Cats: 1

Pattern: 1.
Cat_a = 3, cat_b = 1.
Mapfield[0][0] = 2.
Mapfield[0][1] = 2.
Mapfield[1][0] = 2.
Mapfield[1][1] = 2.
Mapfield[2][0] = 2.
Mapfield[2][1] = 2.
Mapfield[3][0] = 0.
Mapfield[3][1] = 1.
Mapfield[4][0] = 0.
Mapfield[4][1] = 1.
Steps: 2, Corr: 0, Incor: 0, Unkn: 1, Cats: 2
```

Neither of these input classes were known before hand, so new maps were created for them. The next pattern agrees with the first, as does the output class, so we have a correct prediction.

```
Pattern: 2.
Cat_a = 4, cat_b = 1.
Steps: 3, Corr: 1, Incor: 0, Unkn: 0, Cats: 2
```

The data file is now altered so that the third pattern requires a different output to the first. This provides a mis-match with the above classification, and so match tracking ensues.

```
test.dat:
0 1 1 1 1 0 0 0 0 0, 1 0
0 0 0 0 0 0 0 1 1 1, 1 0
1 1 0 1 1 0 0 0 0 0, 0 1

Pattern: 0.
Cat_a = 4, cat_b = 1.
Mapfield[0][0] = 2.
Mapfield[0][1] = 2.
Mapfield[1][0] = 2.
Mapfield[1][1] = 2.
Mapfield[2][0] = 2.
Mapfield[2][1] = 2.
Mapfield[3][0] = 2.
Mapfield[3][1] = 2.
Mapfield[4][0] = 0.
Mapfield[4][1] = 1.
Steps: 1, Corr: 0, Incor: 0, Unkn: 1, Cats: 1

Pattern: 1.
Cat_a = 3, cat_b = 1.
Mapfield[0][0] = 2.
Mapfield[0][1] = 2.
Mapfield[1][0] = 2.
```

```
Mapfield[1][1] = 2.
Mapfield[2][0] = 2.
Mapfield[2][1] = 2.
Mapfield[3][0] = 0.
Mapfield[3][1] = 1.
Mapfield[4][0] = 0.
Mapfield[4][1] = 1.
Steps: 2, Corr: 0, Incor: 0, Unkn: 1, Cats: 2

Pattern: 2.
Cat_a = 4, cat_b = 0.
Increase rho: 0.757500.
Cat_a = 2, cat_b = 0.
Mapfield[0][0] = 2.
Mapfield[0][1] = 2.
Mapfield[1][0] = 2.
Mapfield[1][1] = 2.
Mapfield[2][0] = 1.
Mapfield[2][1] = 0.
Mapfield[3][0] = 0.
Mapfield[3][1] = 1.
Mapfield[4][0] = 0.
Mapfield[4][1] = 1.
Steps: 3, Corr: 0, Incor: 1, Unkn: 0, Cats: 3
```

8 ARTMAP and the Mushroom Database

An example training ARTMAP on a large database of data is now presented, illustrating the degree of learning required for effective generalisation. The database is of mushrooms,[23] and the input vector encodes for various properties of a mushroom, and the output indicating a poisonous or edible mushroom. The database consists of 8,124 examples, splitting fairly evenly between poisonous and edible. The output vectors are simple (1,0) for poisonous, and (0,1) for edible examples.

The input vectors encode for 22 different properties. These range through Cap-Shape, Cap-Surface, Cap-Colour, Habitat, Population, Stalk-Colour-Below-Ring and so on. Each property possesses a range of values, so that Cap-Shape may be Bell, Conical, Convex, Flat, Knobbed or Sunken. This is represented by six bits in the input vector, so that a Bell mushroom would have the first bit set etc. From the 22 properties, the input vector is 126 bits in size, each having 22 set bits and 104 not.

The network was now run through a series of tests, following the descriptions given by Carpenter, Grossberg, and Reynolds. Total simulation time, including reading in the data, training, and reading out the results, varied, taking up to two minutes for the complete data set. This is comparable to the times quoted.

8.1 On-line learning

On-line learning means that for each input vector, it is used to predict a possible output. This prediction is compared against the actual output supplied, and then learning proceeds, before the next input presentation. It has already been remarked that provision is made in the code for reporting the results of such learning. This is in the form of a report after each scheduled 'step size' of the number of correct,incorrect and unknown predictions made in the previous step.

The database may be sampled in one of two ways to obtain data for training on. The first involves selecting items at random from the data set, with replacement. The second does not

[23]Schlimmer, J.S. (1987a), *Mushroom database* UCI Repository of Machine Learning Databases. (aha@ics.uci.edu)

replace samples back into the pool after selecting them. With the former the training set may contain the same sample a number of times, and hence reinforcement occurs. The latter produces a greater spread in samples across the total data set.

Experiments were performed with the vigilance parameter ρ_a set at one of two values, 0 and 0.7. The former is termed 'forced-choice', as no alteration can occur in the ARTa module to its initial selection. The latter is termed 'conservative', as the input category is guaranteed to match the input to within the tolerance given.

Initial experiments were repeated five times, results averaged, and the following table produced:

	$\rho_a = 0$	$\rho_a = 0$	$\rho_a = 0.7$	$\rho_a = 0.7$
Trial	No Replace	Replace	No Replace	Replace
100	71.2	70.8	67.2	66.6
200	96.0	97.8	89.2	88.4
300	98.2	98.0	92.8	94.2
400	96.8	99.2	94.6	95.8
500	98.4	99.8	96.8	96.6
600	99.4	98.8	97.8	98.2
700	100	99.8	98.4	98.6
800	99.6	99.8	99.0	99.0
900	99.8	100	99.8	99.4
1000	99.6	99.6	98.6	99.4
1100	99.8	99.6	99.6	99.6
1200	100	99.8	99.4	100
1300	100	99.8	99.6	99.4
1400	99.8	100	100	99.8
1500	100	99.6	99.4	99.8
1600	100	100	100	99.4
1700	100	99.6	100	99.8
1800	100	100	99.8	100
1900	100	100	100	99.8
2000	100	100	99.8	100

Having performed the initial experiments of both on-line and off-line learning it was seen that the 'forced-choice' of $\rho_a = 0$ was not so forced after all, with many 'unknown' results occurring. The reason for this is in the initial setting up of the bottom up weights. It is not sufficient for these to be in the given ranges, as described earlier, α close to the maximum allows uncommitted categories to compete easier with committed ones. When α is set uniformly to $1/100$ for all the bottom up weights at initialisation, then it becomes a very rare event for an uncommitted node to be selected instead of an already committed one. This scenario then leads to the 'forced-choice' mentioned above, and leads to a different set of results. These are more dramatic with off-line learning, but for comparison with the above are results of a similar experiment, but only performed with one sample.

	$\rho_a = 0$	$\rho_a = 0$	$\rho_a = 0.7$	$\rho_a = 0.7$
Trial	No Replace	Replace	No Replace	Replace
100	84	83	66	69
200	97	87	87	83
300	92	92	95	97
400	96	93	97	98
500	92	96	96	95
600	100	98	100	98
700	99	100	98	98
800	99	100	99	100
900	100	100	100	99
1000	100	100	100	98
1100	100	100	100	98
1200	100	100	100	100
1300	99	100	99	100
1400	100	100	100	100
1500	99	100	99	100
1600	98	100	100	100
1700	99	100	99	100
1800	99	100	100	99
1900	100	100	100	99
2000	99	100	99	100

As can be seen, the latter results are higher scoring with $\rho_a = 0$, due to the selection of a category already committed. This makes a prediction, whereas previously an uncommitted category would have been selected, leading to no prediction being made.

It should be noted that these results are in approximate agreement to those in the ARTMAP paper.

8.2 Off-line learning

Off-line learning refers to the training upon a data set, usually until 100% proficiency is reached, and then using what has been learnt in generalising to the full data set. The experiments performed here were done on samples ranging from 3 to 4000. It was rare for 100% performance not to be reached upon a single presentation of the training set, so the differences are not recorded. Again we distinguish the conservative and forced-choice cases.

Initially experiments were performed, five times, with the random selection of bottom up weights, with the following results:

Off-line Conservative $\rho_a = 0.7$

Set Size	% Correct	% Incorrect	% Unknown	No. ARTa categories
3	0.46	0	99.54	2-3
5	0.76	0	99.24	4-5
15	28.76	0	71.24	14-15
30	73.10	0.22	26.68	21-22
60	83.85	6.63	9.52	26-29
125	86.11	0.78	13.11	34-39
250	93.37	0.20	6.43	44-50
500	98.07	0.22	1.71	50-63
1000	99.36	0.08	0.54	54-68
2000	99.77	0	0.23	57-69
4000	99.98	0	0.02	59-69

Off-line Forced Choice $\rho_a = 0.0$

Set Size	% Correct	% Incorrect	% Unknown	No. ARTa categories
3	0.46	0	99.54	2-3
5	0.76	0	99.24	4-5
15	28.76	0	71.24	14-15
30	77.28	1.20	21.52	21-22
60	91.15	4.50	4.35	25-28
125	91.15	3.31	0.62	26-30
250	98.57	1.05	0.38	28-32
500	99.14	0.75	0.11	29-32
1000	99.77	0.17	0.06	29-37
2000	99.91	0.08	0.01	30-39
4000	100	0	0	30-40

Experiments were repeated singly with the low value of α.

Off-line Conservative $\rho_a = 0.7$

Set Size	% Correct	% Incorrect	% Unknown	No. ARTa categories
3	43.5	0.4	56.1	3
5	45.2	0.4	54.3	4
15	62.2	0	37.8	9
30	67.8	0.3	31.9	15
60	76.6	0.5	19.4	25
125	80.1	0.5	19.4	37
250	95.8	0.3	3.9	49
500	99.4	0.2	0.4	59
1000	99.4	0.2	0.4	62
2000	99.3	0	0.7	64
4000	100	0	0	69

Off-line Forced Choice $\rho_a = 0.0$

Set Size	% Correct	% Incorrect	% Unknown	No. ARTa categories
3	73.9	26.1	0	2
5	72.6	27.4	0	2
15	83.7	16.3	0	3
30	92.0	8.0	0	3
60	93.7	6.3	0	6
125	93.9	6.1	0	6
250	94.6	5.4	0	7
500	98.4	1.6	0	11
1000	99.4	0.4	0	12
2000	99.9	0.1	0	13
4000	100	0	0	17

The last table shows the effects of the change in initialisation, with true forced choice behaviour. In each case though, after a number of samples were presented, generalisation became very good. This demonstrates that the mushroom database clusters well. There is a range of categories finally learnt, with the smallest values being with the forced choice case. The conservative case branches out with more categories, however, it very rarely makes an incorrect decision, this figure being below 1% in the second case, and only a peculiar result with a sample size of 60 marring the same figure in the first.

It is evident that this implementation is performing at an equivalent level to that in the ARTMAP paper. Further, the extra set of experiments has demonstrated the necessity to be careful with the initial parameters of a neural network so as to get the best results from it.

9 Conclusion

An implementation of ART-1, ART-2a and ARTMAP has been constructed and demonstrated. Examples of unsupervised and supervised learning tasks have been illustrated, with extended testing of the latter.

The unsupervised learning example, as in ART-1, provides an interesting example of self-organisation by a neural network. The difficulty being the dependence on the setting of the vigilance parameter. The differing results of which are illustrated in the sequences of letter presentations shown earlier. There is no real theory to provide a guide to this, and it is really a case for experimentation and seeing which of the resulting categorisations is of most benefit. This accords with the comments in the introduction about unsupervised learning in general. Especially in this case where the computer is working 'blind', there is no real benchmark to gauge the effectiveness of particular decisions.

ARTMAP itself provides a sound test bed on which to perform supervised learning. The operation is rapid and clear cut, with immediate learning of new data. It is fully incremental, up to the capacity of the network, and with the match tracking ability is sensitive to the exceptions present in data sets. This sensitivity extends to alleviating the impact of incorrect examples, which will find themselves relegated to a class of their own, and probably not referred to again. However, outright contradictions where two identical input vectors provide differing outputs, are not accommodated, with the later of the two being ignored.

Further extensions of the architecture were suggested in the earlier comments. It would be interesting to combine the self-organising properties of ARTMAP with a domain sensitive control structure.

Guide to Using ART-1

ART-1 is an adaptive resonance theory neural network, which performs unsupervised categorisation of binary input patterns. The network is characterised by a set of system parameters:

num_inputs - the number of bits making up the input vector. Must be at least 1 and not exceed the available maximum (formed at compilation of the code).

num_cats - the maximum number of categories the system may create. Must be at least 1 and not exceed the available maximum.

max_mod_input - the maximum number of bits which may be set in the input vector.

rho - the vigilance parameter of the network.

beta - a (small) coefficient affecting the initial set up of weights

These parameters are passed to the network in an *information file*, called **somename.inf** in the form:

num_inputs, num_cats, max_mod_input, rho, beta

Spaces and/or commas are permitted separators. An example with three inputs, up to five categories, all the bits may be set, a vigilance of 0.6 (60%), and a weighting coefficient of .5, being:

```
3, 2, 3, 0.6 .5
```

The data is passed to the network in a *data file*, called **somename.dat** in the form:

number of patterns on the first line, as a positive integer.

each subsequent line contains a pattern, items separated by a space and/or comma. All items must be a 0 or a 1, and there must be at least as many items as given in the num_inputs parameter or an error will occur. An error will also result if there are not as many patterns as given in the first line of the file.

Example of a three pattern input data set:

```
3
1,0,1
0,1,1
1,1,1
```

The network is used by passing the program the name of the above files: **art somename**

The extensions, .inf and .dat, will be considered automatically. Output will be in the form of some visual display, and an output file, **somename.wgt** containing the definitions of the categories which became committed. Each line of this value has the category number first, followed by the value of each bit of the category definition, separated by spaces. Thus:

```
0 0 1 1
1 1 0 1
```

Would signify that categories 0 and 1 were committed. The classes being 0 1 1 and 1 0 1 respectively.

A trace is obtained by appending -t to the call:

art somename -t

Output to the screen will comprise the value of each array on each presentation of input.

Guide to Using ART-2a

ART-2a is an adaptive resonance theory neural network, which performs unsupervised categorisation of real-valued input patterns. The network is characterised by a set of system parameters:

num_inputs - the number of bits making up the input vector. Must be at least 1 and not exceed the available maximum (formed at compilation of the code).

num_cats - the maximum number of categories the system may create. Must be at least 1 and not exceed the available maximum.

rho - the vigilance parameter of the network, set between 0 and 1.

beta - a learning coefficient: set to 0 and only the initial category example is retained, set to 1 for the category definition to be the latest example, intermediate values give varying ratios.

theta - a threshold value, any components whose normalised value is below this are ignored. Set between 0 and $1/\sqrt{num\_inputs}$.

These parameters are passed to the network in an *information file*, called **somename.inf** in the form:

num_inputs, num_cats, rho, beta, theta

Spaces and/or commas are permitted separators. An example with three inputs, up to five categories, a vigilance of 0.9 (90%), a weighting coefficient of .5, and a threshold of 0.2, being:

```
3, 5, 0.9 .5, 0.2
```

The data is passed to the network in a *data file*, called **somename.dat** in the form:

number of patterns on the first line, as a positive integer.

each subsequent line contains a pattern, items separated by a space and/or comma. All items are real numbers, and there must be at least as many items as given in the num_inputs parameter or an error will occur. An error will also result if there are not as many patterns as given in the first line of the file.

Example of a three pattern input data set:

```
3
1,0.556,0.005
0,1.6e-2,1
1,1,1
```

The network is used by passing the program the name of the above files: **art somename**

The extensions, .inf and .dat, will be considered automatically. Output will be in the form of some visual display, and an output file, **somename.wgt** containing the definitions of the categories which became committed. Each line of this value has the category number first, followed by the value of each bit of the category definition, separated by spaces. Thus:

```
2 0.321796 0.321798 0.000000
4 0.466187 0.000000 0.045195
```

Would signify that categories 2 and 4 were committed. The classes as shown.

A trace is obtained by appending -t to the call:

art somename -t

Output to the screen will comprise the value of each array on each presentation of input.

Guide to Using ARTMAP

ARTMAP is used in a similar fashion to ART-1 and ART-2a. There is an information file, formed from two of the information files above, with one extra parameter, thus:

num_inputs_a, num_cats_a, max_mod_input_a, rho_a, beta_a, stepsize

num_inputs_b, num_cats_b, max_mod_input_b, rho_b, beta_b

The stepsize is the number of steps to perform before reporting back with the online learning details. The other parameters are the familiar ones for ART-1 from above.

The data file is similarly extended, simply by making each pattern line a joint input output line.

The program may be run in three modes.

Learning mode, called with **artmap -l file** [**datafile**] will do the learning.

Trace mode, called with **artmap -t file** [**datafile**] will perform a similar job with reporting of the ARTMAP stage.

Generalisation mode, called with **artmap -g file** [**datafile**] will not do any learning, but instead report back on how well the learnt weights classified the data set.

The third parameter is optional. If the data set to be used is not called **file.dat** then it may be declared on the command line. This overrides the default data file.

After learning, the weights are saved off in a file **file.wgt**. If this file exists when ARTMAP is called, it will be loaded into memory. This allows for incremental learning to occur, as well as being the means by which generalisation can be calculated.

Program Listings

```
/* ART-1 Simulator, Peter Lane, April 1995 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

/* define constants determining memory for arrays */
#define MAX_INPUTS 70
#define MAX_CATS   70

/* amount of space for a line of text from a file */
#define BUFFERSIZE 500

/* length set aside for a file name */
#define FILENAMESIZE 50

/* array to hold input vector */
int input[MAX_INPUTS];
/* array for the category selected vector */
int cats[MAX_CATS];
/* array of flags, element set if category node committed */
int comm[MAX_CATS];
/* array of flags, element set if category node ignored during input */
int active[MAX_CATS];
/* array of category definitions */
int topdn[MAX_CATS][MAX_INPUTS];
/* array of weights from input into category nodes */
double botup[MAX_INPUTS][MAX_CATS];
/* array to hold activation values of category nodes during input */
double act[MAX_CATS];

/* define some global variables, to hold system parameters et al */
double rho, beta;
int mod_i, num_inputs, num_cats, num_pats, count_p, max_mod_input, trace;

void initialise();
void display();
void present_input(FILE *);
void dot_product();
void clear_active();
void clear_cats();
int get_choice();
int do_search(int );

main(int argc, char *argv[])
{
int selected, valid, sum, i, j, nlearn, left;
FILE *fp;
char *str;
char buffer[BUFFERSIZE], filename[FILENAMESIZE];
```

```
/* look after the command line, at least one parameter expected */
if ((argc != 2) && (argc!=3))
  {
  printf("****    ART-1 Simulation     ****\n");
  printf("**** Peter Lane, April 1995 ****\n\n");
  printf("  art file [-t]\n");
  exit(0);
  }

/* if exists, the second parameter may ask for a trace of activity */
trace=0;
if (argc==3)
  if (strcmp(argv[2], "-t") == 0)
    trace=1;

printf("****    ART-1 Simulation     ****\n");
printf("**** Peter Lane, April 1995 ****\n\n");

/* loads in system parameters from the .inf file */
strcpy(filename,argv[1]);
strcat(filename,".inf");

if ((fp=fopen(filename, "r")) == NULL)
  {
  printf("Cannot open information file: %s for reading.\n", filename);
  exit(0);
  }
else
  printf("Reading information file: %s.\n", filename);

if (fgets(buffer, BUFFERSIZE, fp) == NULL)
  {
  printf("Error: Could not read a line from information file.\n");
  exit(0);
  }

fclose(fp);

/* any errors in the information file cause the program to abort */
str=strtok(buffer, ", ");
if (str==NULL)
  {
  printf("Error: Number of inputs not read.\n");
  exit(0);
  }
num_inputs=atoi(str);
if (!((num_inputs>0) && (num_inputs<MAX_INPUTS+1)))
  {
  printf("Error: invalid Number of inputs.\n");
  exit(0);
  }
str=strtok(NULL, ", ");
if (str==NULL)
  {
```

```
  printf("Error: Number of categories not read.\n");
  exit(0);
  }
num_cats=atoi(str);
if (!((num_cats>0) && (num_cats<MAX_CATS+1)))
  {
  printf("Error: invalid Number of Categories.\n");
  exit(0);
  }
str=strtok(NULL, ", ");
if (str==NULL)
  {
  printf("Error: Maximum Input Modulus not read.\n");
  exit(0);
  }
max_mod_input=atoi(str);
if (!((max_mod_input>0) && (max_mod_input<MAX_INPUTS+1)))
  {
  printf("Error: invalid Maximum Input Modulus.\n");
  exit(0);
  }
str=strtok(NULL, ", ");
if (str==NULL)
  {
  printf("Error: ‘rho’ not read.\n");
  exit(0);
  }
rho=atof(str);
str=strtok(NULL, ", ");
if (str==NULL)
  {
  printf("Error: ‘beta’ not read.\n");
  exit(0);
  }
beta=atof(str);

/* prepare for reading patterns from .dat file */
strcpy(filename,argv[1]);
strcat(filename,".dat");
if ((fp=fopen(filename, "r")) == NULL)
  {
  printf("Cannot open data file: %s for reading.\n", filename);
  exit(0);
  }
else
  printf("Reading data file: %s.\n", filename);

if (fgets(buffer, BUFFERSIZE, fp) == NULL)
  {
  printf("Error: Could not read number of patterns.\n");
  exit(0);
  }
num_pats=atoi(buffer);
if (!(num_pats>0))
```

```
  {
  printf("Error: Number of patterns not a positive integer.\n");
  exit(0);
  }

/* display the system parameters for confirmation */
printf("num_inputs: %d, num_cats: %d, max_mod_input: %d\n",
  num_inputs, num_cats, max_mod_input);
printf("rho: %f, beta: %f\n", rho, beta);
printf("Number of patterns: %d.\n", num_pats);

initialise();

/* input presentation loop */

for (count_p=0; count_p<num_pats;count_p++)
  {
  if (trace)
    printf("\n\nPattern: %d\n", count_p);

  /* pick up next pattern in the file */
  present_input(fp);
  clear_active();
  clear_cats();

  /* compute the activation of each category */
  dot_product();
  if (trace)
    display();

  valid=1;    /* flag for validity of category */
  nlearn=0;   /* flag to prevent learning from proceeding */
  left=1;     /* a count of the available categories */

  while ((valid != 0) && (left != 0))
    {
    selected=get_choice();    /* find max. unused category */
    if (trace)
      printf("Selected node = %d\n", selected);
    valid=do_search(selected);  /* check validity */

    if (valid)      /* IF INVALID */
      {
      active[selected]=1; /* flag out of current cycle */
      left=0;
      for (i=0;i<num_cats;i++)
        if (active[i]==0)
          left++;
      if (left==0)    /* any categories left? */
        {       /* if none, suppress learning */
        nlearn=1;
        if (trace)
          printf("\n***** Saturated categories! *****\n\n");
        }
      }
    else          /* VALID */
      {
      cats[selected]=1; /* so mark as selected */
      if (trace)
        printf("VALID node!\n\n");
      }
    }


  if (nlearn==0)        /* learning step */
    {
    sum=0;
    for (i=0;i<num_inputs;i++)
      {
      topdn[selected][i]&=input[i]; /* see equation (1.5) */
      sum+=topdn[selected][i];
      }
    for (i=0;i<num_inputs;i++)      /* see equation (1.6) */
      botup[i][selected]=topdn[selected][i]/(beta+sum);
    comm[selected]=1;         /* flag committed */
    }
  }

fclose(fp);

/* save off and display resultant categories */
strcpy(filename, argv[1]);
strcat(filename, ".wgt");
printf("Outputting committed categories to file: %s\n", filename);

if ((fp=fopen(filename,"w")) == NULL)
  {
  printf("Error: Cannot open file for output.\n");
  exit(0);
  }

for (j=0;j<num_cats;j++)
  if (comm[j] == 1)
    {
    fprintf(fp, "%d", j);
    printf("%d", j);
    for (i=0;i<num_inputs;i++)
      {
      fprintf(fp, " %d", topdn[j][i]);
      printf(" %d", topdn[j][i]);
      }
    fprintf(fp, "\n");
    printf("\n");
    }

return 0;
}
```

```
/* reset the category vector */
void clear_cats()
{
int i;

for (i=0;i<num_cats;i++)
  cats[i]=0;

}

/* reset the flags of available categories */
/* active == 1 implies the category is ignored */
/* for current input presentation */
void clear_active()
{
int i;

for (i=0;i<num_cats;i++)
  active[i]=0;

}

/* is selected node matching the input sufficiently? */
int do_search(int selected)
{
int i, sum, valid, sum2;

if (comm[selected]==0)  /* if category is not committed, always ok */
  valid=0;
else
  {
  sum=0;
  sum2=0;
  valid=0;

  for (i=0;i<num_inputs;i++)
    {
    sum+=(input[i] & topdn[selected][i]);
    sum2+=topdn[selected][i];     /* if we are using (1.4a) */
    }

/* alter the comments below to select the validity equation desired */
/*  if (sum*sum < mod_i*sum2*rho)  see equation (1.4a) */
  if (sum < mod_i*rho)      /* see equation (1.4) */
    valid=1;
  }

return valid;         /* return 0 if matches sufficiently */
}

/* pick from the available categories the one with maximum activation */
int get_choice()
{
int i, selected;
```

```
selected=0;
while (active[selected])  /* NB: routine not called if none available */
  selected++;

for (i=selected;i<num_cats;i++)
  if ( (act[selected]<act[i]) && (active[i] == 0) )
    selected=i;

return selected;
}

/* compute the category activation values */
void dot_product()
{
int i,j;

for (j=0;j<num_cats;j++)
  {
  act[j]=0.0;
  for (i=0;i<num_inputs;i++)
    act[j]+=input[i]*botup[i][j]; /* see equation (1.3) */
  }
}

/* pick up next pattern from file, exit if any errors found */
void present_input(FILE *filep)
{
char buffer[BUFFERSIZE];
int i;
char *str;

if (fgets(buffer, BUFFERSIZE, filep) == NULL)
  {
  printf("Error: Not enough input patterns.\n");
  exit(0);
  }

str=strtok(buffer, ", ");
if (str==NULL)
  {
  printf("Error: Insufficient items in pattern %d.\n", count_p);
  exit(0);
  }
input[0]=atoi(str);
if ((input[0] != 0) && (input[0] != 1))
  {
  printf("Error: Item 0 of pattern %d not a 0 or 1.\n", count_p);
  exit(0);
  }
for (i=1;i<num_inputs;i++)
  {
  str=strtok(NULL, ", ");
  if (str==NULL)
```

```
    {
    printf("Error: Insufficient items in pattern %d.\n", count_p);
    exit(0);
    }
  input[i]=atoi(str);
  if ((input[i] != 0) && (input[i] != 1))
    {
    printf("Error: Item %d of pattern %d not a 0 or 1.\n", i, count_p);
    exit(0);
    }
  }

mod_i=0;      /* modulus of input */
for (i=0;i<num_inputs;i++)
  mod_i+=input[i];
if (mod_i == 0)
  {
  printf("Error: Modulus of pattern is zero.\n");
  exit(0);
  }
}

/* set up all arrays to start up values */
void initialise()
{
int i,j;
double rnd;

for (i=0;i<num_inputs;i++)
  input[i]=0;

for (i=0;i<num_cats;i++)
  {
  cats[i]=0;
  comm[i]=0;
  }

for (i=0;i<num_inputs;i++)
  for (j=0;j<num_cats;j++)
    topdn[j][i]=1;        /* see equation (1.2) */

for (j=0;j<num_cats;j++)
  {
  while ( !((rnd=drand48()/(beta+max_mod_input)) > 0.0) ) ;
  for (i=0;i<num_inputs;i++)
    botup[i][j]=rnd;      /* see equation (1.1) */
  }
}

/* if trace is set, this routine is called to give details of all arrays */
void display()
{
int i,j;
```

```
for (i=0;i<num_inputs;i++)
  printf("Input[%d] = %d\n", i, input[i]);

for (i=0;i<num_cats;i++)
  printf("Cats[%d] = %d, Comm[%d] = %d\n", i, cats[i], i, comm[i]);

for (i=0;i<num_inputs;i++)
  for (j=0;j<num_cats;j++)
    printf("Topdn[%d][%d] = %d\n", j, i, topdn[j][i]);

for (i=0;i<num_inputs;i++)
  for (j=0;j<num_cats;j++)
    printf("Botup[%d][%d] = %e\n", i, j, botup[i][j]);

for (i=0;i<num_cats;i++)
  printf("Act[%d] = %e\n", i, act[i]);

}
```

```
/* ART-2a Simulator, Peter Lane, April 1995 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

/* define constants to determine amount of memory for storage */
#define MAX_INPUTS  70
#define MAX_CATS  70

/* amount of space for a line of text from a file */
#define BUFFERSIZE  500

/* length set aside for a file name */
#define FILENAMESIZE  50

/* array to hold input vector */
double input[MAX_INPUTS];
/* array for use in learning computation */
double psi[MAX_INPUTS];
/* array for the category selected vector */
int cats[MAX_CATS];
/* array of flags, element set if category node committed */
int comm[MAX_CATS];
/* array of flags, element set if category node to be ignored during input */
int active[MAX_CATS];
/* array of weights from input into category nodes */
double botup[MAX_INPUTS][MAX_CATS];
/* array to hold activation values of category nodes during input */
double act[MAX_CATS];

/* define some global variables, to hold system parameters et al */
double theta, beta, rho;
int num_inputs, num_cats, num_pats, count_p, trace;

void initialise();
void display();
void present_input(FILE *);
double mod_input();
void dot_product();
void clear_active();
void clear_cats();
int get_choice();
int do_search(int );

main(int argc, char *argv[])
{
int i, j, valid, nlearn, left, selected;
double sum;
FILE *fp;
char *str;
char buffer[BUFFERSIZE], filename[FILENAMESIZE];
```

```
/* look after the command line, at least one parameter expected */
if ((argc != 2) && (argc != 3))
  {
  printf("****   ART-2a Simulation    ****\n");
  printf("**** Peter Lane, April 1995 ****\n\n");
  printf("  art2 file [-t]\n");
  exit(0);
  }

/* if exists, the second parameter may ask for a trace of activity */
trace = 0;
if (argc == 3)
  if (strcmp(argv[2], "-t") == 0)
    trace = 1;

printf("****    ART-2a Simulation   ****\n");
printf("**** Peter Lane, April 1995 ****\n\n");

/* loads in system parameters from the .inf file */
strcpy(filename, argv[1]);
strcat(filename, ".inf");

if ((fp=fopen(filename, "r")) == NULL)
  {
  printf("Cannot open information file: %s for reading.\n", filename);
  exit(0);
  }
else
  printf("Reading information file: %s.\n", filename);

if (fgets(buffer, BUFFERSIZE, fp) == NULL)
  {
  printf("Error: Could not read a line from information file.\n");
  exit(0);
  }

fclose(fp);

/* any errors in the information file cause the program to abort */
str=strtok(buffer, ", ");
if (str==NULL)
  {
  printf("Error: Number of inputs not read.\n");
  exit(0);
  }
num_inputs=atoi(str);
if (!((num_inputs>0) && (num_inputs<MAX_INPUTS+1)))
  {
  printf("Error: invalid Number of inputs.\n");
  exit(0);
  }
str=strtok(NULL, ", ");
if (str==NULL)
  {
```

```
  printf("Error: Number of categories not read.\n");
  exit(0);
  }
num_cats=atoi(str);
if (!((num_cats>0) && (num_cats<MAX_CATS+1)))
  {
  printf("Error: invalid Number of Categories.\n");
  exit(0);
  }
str=strtok(NULL, ", ");
if (str==NULL)
  {
  printf("Error: ‘rho’ not read.\n");
  exit(0);
  }
rho=atof(str);
str=strtok(NULL, ", ");
if (str==NULL)
  {
  printf("Error: ‘beta’ not read.\n");
  exit(0);
  }
beta=atof(str);
str=strtok(NULL, ", ");
if (str==NULL)
  {
  printf("Error: ‘theta’ not read.\n");
  exit(0);
  }
theta=atof(str);
if (!((theta>0) && (theta<(1/sqrt(num_inputs)))))
  {
  printf("Error: ‘theta’ not in [0,%f].\n", (1/sqrt(num_inputs)));
  exit(0);
  }

/* prepare for reading patterns from .dat file */
strcpy(filename,argv[1]);
strcat(filename,".dat");
if ((fp=fopen(filename, "r")) == NULL)
  {
  printf("Cannot open data file: %s for reading.\n", filename);
  exit(0);
  }
else
  printf("Reading data file: %s.\n", filename);

if (fgets(buffer, BUFFERSIZE, fp) == NULL)
  {
  printf("Error: Could not read number of patterns.\n");
  exit(0);
  }
num_pats=atoi(buffer);
if (!(num_pats>0))
```

```
  {
  printf("Error: Number of patterns not a positive integer.\n");
  exit(0);
  }

/* display the system parameters for confirmation */
printf("num_inputs: %d, num_cats: %d\n", num_inputs, num_cats);
printf("rho: %f, beta: %f, theta: %f\n", rho, beta, theta);
printf("Number of patterns: %d.\n", num_pats);

initialise();

/* input presentation loop */

for (count_p=0; count_p<num_pats; count_p++)
  {
  if (trace)
    printf("\n\nPattern: %d\n", count_p);

  /* pick up next pattern in the file */
  present_input(fp);
  clear_active();
  clear_cats();

  /* compute the activation of each category */
  dot_product();
  if (trace)
    display();

  valid=1;      /* flag for validity of category */
  nlearn=0;     /* flag to prevent learning from proceeding */
  left=1;       /* a count of the available categories */

  while ((valid != 0) && (left != 0))
    {
    selected=get_choice();    /* find max. unused category */
    if (trace)
      printf("Selected node = %d.\n", selected);

    /* check for validity of a node, if it is already committed */
    valid=0;
    if ((act[selected]<rho) && (comm[selected] == 1))
      valid=1;        /* see equation (2.4) */

    if (valid)          /* IF INVALID */
      {
      active[selected]=1;   /* flag out of current cycle */
      left=0;
      for (i=0;i<num_cats;i++)
        if (active[i]==0)
          left++;
      if (left==0)      /* any categories left? */
        {         /* if none, suppress learning */
        nlearn=1;
        if (trace)
          printf("\n***** Saturated Categories! *****\n");
        }
      }
    else            /* VALID */
      {
      cats[selected]=1;   /* so mark as selected */
      if (trace)
        printf("VALID node!\n");
      }
    }

  if (nlearn==0)            /* learning step */
    {
    if (comm[selected]==0)      /* see equation (2.5) */
      for (i=0;i<num_inputs;i++)
        botup[i][selected]=input[i];
    else
      {
      sum=0.0;
      for (i=0;i<num_inputs;i++)  /* see equation (2.6) */
        {
        if (botup[i][selected]>theta)
          psi[i]=input[i];
        else
          psi[i]=0.0;
        sum+=psi[i]*psi[i];
        }
      for (i=0;i<num_inputs;i++)
        {
        if (sum != 0.0)
          psi[i]=psi[i]/sqrt(sum);  /* normalise psi */
        if (trace)
          printf("Psi[%d] = %f\n", i, psi[i]);
        }
      sum=0.0;
      for (i=0;i<num_inputs;i++)
        {
        botup[i][selected]=beta*psi[i]+(1-beta)*botup[i][selected];
        sum+=botup[i][selected]*botup[i][selected];
        }
      for (i=0;i<num_inputs;i++)
        botup[i][selected]=botup[i][selected]/sqrt(sum);
      }
    comm[selected]=1;     /* flag committed */
    }
  }

fclose(fp);

/* save off and display resultant categories */
strcpy(filename, argv[1]);
strcat(filename, ".wgt");
printf("Outputting committed categories to file: %s\n", filename);
```

```
if ((fp=fopen(filename, "w")) == NULL)
  {
  printf("Error: Cannot open file for output.\n");
  exit(0);
  }
for (j=0;j<num_cats;j++)
  if (comm[j] == 1)
    {
    fprintf(fp, "%d", j);
    printf("%d", j);
    for (i=0;i<num_inputs;i++)
      {
      fprintf(fp, " %f", botup[i][j]);
      printf(" %f", botup[i][j]);
      }
    fprintf(fp, "\n");
    printf("\n");
    }

return 0;
}

/* set up all arrays to start up values */
void initialise()
{
int i,j;
double rnd;

for (i=0;i<1000;i++)
  rnd=drand48();        /* do some rnds */

for (i=0;i<num_inputs;i++)
  input[i]=0.0;

for (i=0;i<num_inputs;i++)
  {
  cats[i]=0;
  comm[i]=0;
  }

for (j=0;j<num_cats;j++)
  {
  while (!((rnd=drand48()/sqrt(num_inputs)) > 0.0));
  for (i=0;i<num_inputs;i++)
    botup[i][j]=rnd;      /* see equation (2.1) */
  }

}

/* reset the category vector */
void clear_cats()
{
int i;
```

```
for (i=0;i<num_cats;i++)
  cats[i]=0;

}

/* reset the flags of available categories */
/* active == 1 implies the category is ignored */
/* for current input presentation */
void clear_active()
{
int i;

for (i=0;i<num_cats;i++)
  active[i]=0;

}

/* pick up next pattern from file, exit if any errors found */
void present_input(FILE *filep)
{
char buffer[BUFFERSIZE];
int i;
char *str;
double tmp;

if (fgets(buffer, BUFFERSIZE, filep) == NULL)
  {
  printf("Error: Not enough input patterns.\n");
  exit(0);
  }

str=strtok(buffer, ", ");
if (str == NULL)
  {
  printf("Error: Insufficient items in pattern %d.\n", count_p);
  exit(0);
  }
input[0]=atof(str);
for (i=1;i<num_inputs;i++)
  {
  str=strtok(NULL, ", ");
  if (str == NULL)
    {
    printf("Error: Insufficient items in pattern %d.\n", count_p);
    exit(0);
    }
  input[i]=atof(str);
  }

if ((tmp=mod_input()) == 0.0)
  {
  printf("Error: Modulus of pattern: %d is zero.\n", count_p);
  exit(0);
  }

/* transform input, see equation (2.2) */

for (i=0;i<num_inputs;i++)
  {
  input[i]=input[i]/tmp;
  if (!(input[i]>theta))    /* first normalisation */
    input[i]=0.0;     /* remove those below threshold of theta */
  }

tmp=mod_input();
for (i=0;i<num_inputs;i++)
  input[i]=input[i]/tmp;    /* normalise second time */

}

/* return the modulus of the input array */
double mod_input()
{
int i;
double sum;

sum=0.0;
for (i=0;i<num_inputs;i++)
  sum+=input[i]*input[i];

return sqrt(sum);
}

/* if trace is set, this routine is called to give details of all arrays */
void display()
{
int i,j;

for (i=0;i<num_inputs;i++)
  printf("Input[%d] = %f\n", i, input[i]);

for (i=0;i<num_cats;i++)
  printf("Cats[%d] = %d, Comm[%d] = %d\n", i, cats[i], i, comm[i]);

for (i=0;i<num_inputs;i++)
  for (j=0;j<num_cats;j++)
    printf("Botup[%d][%d] = %f\n", i, j, botup[i][j]);

for (i=0;i<num_cats;i++)
  printf("Act[%d] = %f\n", i, act[i]);

}

/* compute the category activation values */
void dot_product()
{
int i,j;
```

```
for (j=0;j<num_cats;j++)
  {
  act[j]=0.0;
  for (i=0;i<num_inputs;i++)
    act[j]+=input[i]*botup[i][j];   /* see equation (2.3) */
  }
}

/* pick from the available categories the one with maximum activation */
int get_choice()
{
int i, selected;

selected = 0;
while (active[selected])  /* NB: routine not called if none available */
  selected++;

for (i=selected;i<num_cats;i++)
  if ((act[selected]<act[i]) && (active[i] == 0))
    selected = i;

return selected;
}
```

```
/* ARTMAP Simulator, Peter Lane, April 1995 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

#define MAX_INPUTS_A 200
#define MAX_CATS_A   200
#define MAX_INPUTS_B 200
#define MAX_CATS_B   200
#define BUFFERSIZE 5000
#define FILENAMESIZE 50

/* define modes of operation of network */
#define LEARN 0
#define TRACE 1
#define GENER 2

int input_a[MAX_INPUTS_A];
int cats_a[MAX_CATS_A];
int comm_a[MAX_CATS_A];
int active_a[MAX_CATS_A];
int topdn_a[MAX_CATS_A][MAX_INPUTS_A];
double botup_a[MAX_INPUTS_A][MAX_CATS_A];
double act_a[MAX_CATS_A];

int input_b[MAX_INPUTS_B];
int cats_b[MAX_CATS_B];
int comm_b[MAX_CATS_B];
int active_b[MAX_CATS_B];
int topdn_b[MAX_CATS_B][MAX_INPUTS_B];
double botup_b[MAX_INPUTS_B][MAX_CATS_B];
double act_b[MAX_CATS_B];

int mapfield[MAX_CATS_A][MAX_CATS_B];

double set_rho_a, rho_a, beta_a;
int mod_i_a, num_inputs_a, num_cats_a, selected_a, nlearn_a, num_pats, count_p;
int max_mod_input_a, max_mod_input_b;
double rho_b, beta_b;
int mod_i_b, num_inputs_b, num_cats_b;
int mode, stepsize;
double similarity_a;

int art_a_cat();
void art_a_learn();
void initialise_a();
void dot_product_a();
void clear_active_a();
void clear_cats_a();
int get_choice_a();
int do_search_a(int );
int art_b();
```

```
void initialise_b();
void learn_mode(FILE *, char *);
void check_mode(FILE *);
void dot_product_b();
void clear_active_b();
void clear_cats_b();
int get_choice_b();
int do_search_b(int );
void initialise();
void display_map();
void present_input(FILE *);
void read_weights(FILE *);
int num_used_cats();

main(int argc, char *argv[])
{
FILE *fp;
char *str;
char buffer[BUFFERSIZE], filename[FILENAMESIZE];

/* look after the command line */
if ((argc != 3) && (argc != 4))
  {
  printf("****    ARTMAP Simulation   ****\n");
  printf("**** Peter Lane, April 1995 ****\n\n");
  printf("  artmap -l file [datafile]   to learn\n");
  printf("  artmap -t file [datafile]   to trace\n");
  printf("  artmap -g file [datafile]   to generalise\n");
  exit(0);
  }

printf("****    ARTMAP Simulation   ****\n");
printf("**** Peter Lane, April 1995 ****\n\n");

if (strcmp(argv[1], "-l") == 0)
  {
  mode=LEARN;
  printf("Learn Mode.\n");
  }
else if (strcmp(argv[1], "-t") == 0)
  {
  mode=TRACE;
  printf("Trace Mode.\n");
  }
else if (strcmp(argv[1], "-g") == 0)
  {
  mode=GENER;
  printf("Generalisation Mode.\n");
  }
else
  {
  printf("Error: Illegal mode of operation.\n");
  exit(0);
  }
```

```
/* load in system parameters from the .inf file */
strcpy(filename,argv[2]);
strcat(filename,".inf");

if ((fp=fopen(filename, "r")) == NULL)
  {
  printf("Cannot open information file: %s for reading.\n", filename);
  exit(0);
  }
else
  printf("Reading information file: %s.\n", filename);

if (fgets(buffer, BUFFERSIZE, fp) == NULL)
  {
  printf("Error: Could not read a line from information file.\n");
  exit(0);
  }

/* information held in two lines of file, get that for ARTa first */
str=strtok(buffer, ", ");
if (str==NULL)
  {
  printf("Error: Number of inputs for A not read.\n");
  exit(0);
  }
num_inputs_a=atoi(str);
if (!((num_inputs_a>0) && (num_inputs_a<MAX_INPUTS_A+1)))
  {
  printf("Error: invalid number of inputs for A.\n");
  exit(0);
  }
str=strtok(NULL, ", ");
if (str==NULL)
  {
  printf("Error: Number of categories for A not read.\n");
  exit(0);
  }
num_cats_a=atoi(str);
if (!((num_cats_a>0) && (num_cats_a<MAX_CATS_A+1)))
  {
  printf("Error: invalid number of categories for A.\n");
  exit(0);
  }
str=strtok(NULL, ", ");
if (str==NULL)
  {
  printf("Error: Maximum Input Modulus for A not read.\n");
  exit(0);
  }
max_mod_input_a=atoi(str);
if (!((max_mod_input_a>0) && (max_mod_input_a<MAX_INPUTS_A+1)))
  {
  printf("Error: invalid maximum input modulus for A.\n");
```

```
    exit(0);
    }
  str=strtok(NULL, ", ");
  if (str==NULL)
    {
    printf("Error: ‘rho’ for A not read.\n");
    exit(0);
    }
  set_rho_a=atof(str);
  str=strtok(NULL, ", ");
  if (str==NULL)
    {
    printf("Error: ‘beta’ for A not read.\n");
    exit(0);
    }
  beta_a=atof(str);
  str=strtok(NULL, ", ");
  if (str==NULL)
    {
    printf("Error: stepsize not read.\n");
    exit(0);
    }
  stepsize=atoi(str);
  if (!(stepsize>0))
    {
    printf("Error: invalid stepsize.\n");
    exit(0);
    }

  /* now get second line for ARTb */
  if (fgets(buffer, BUFFERSIZE, fp) == NULL)
    {
    printf("Error: Could not read a line from information file.\n");
    exit(0);
    }

  str=strtok(buffer, ", ");
  if (str==NULL)
    {
    printf("Error: Number of inputs for B not read.\n");
    exit(0);
    }
  num_inputs_b=atoi(str);
  if (!((num_inputs_b>0) && (num_inputs_b<MAX_INPUTS_B+1)))
    {
    printf("Error: invalid number of inputs for B.\n");
    exit(0);
    }
  str=strtok(NULL, ", ");
  if (str==NULL)
    {
    printf("Error: Number of categories for B not read.\n");
    exit(0);
    }
```

```
num_cats_b=atoi(str);
if (!((num_cats_b>0) && (num_cats_b<MAX_CATS_B+1)))
  {
  printf("Error: invalid number of categories for B.\n");
  exit(0);
  }
str=strtok(NULL, ", ");
if (str==NULL)
  {
  printf("Error: Maximum Input Modulus for B not read.\n");
  exit(0);
  }
max_mod_input_b=atoi(str);
if (!((max_mod_input_b>0) && (max_mod_input_b<MAX_INPUTS_B+1)))
  {
  printf("Error: invalid maximum input modulus for B.\n");
  exit(0);
  }
str=strtok(NULL, ", ");
if (str==NULL)
  {
  printf("Error: ‘rho’ for B not read.\n");
  exit(0);
  }
rho_b=atof(str);
str=strtok(NULL, ", ");
if (str==NULL)
  {
  printf("Error: ‘beta’ for B not read.\n");
  exit(0);
  }
beta_b=atof(str);

/* if there is a .wgt file, then pick up the weights from it */
strcpy(filename, argv[2]);
strcat(filename, ".wgt");

if ((fp=fopen(filename, "r")) == NULL)
  initialise();
else
  {
  read_weights(fp);
  fclose(fp);
  }

/* prepare for reading patterns from .dat file */
if (argc == 3)
  {
  strcpy(filename,argv[2]);
  strcat(filename,".dat");
  }
else
  strcpy(filename,argv[3]);
```

```
if ((fp=fopen(filename, "r")) == NULL)
  {
  printf("Cannot open data file: %s for reading.\n", filename);
  exit(0);
  }
else
  printf("Reading data file: %s.\n", filename);

if (fgets(buffer, BUFFERSIZE, fp) == NULL)
  {
  printf("Error: Could not read number of patterns.\n");
  exit(0);
  }
num_pats=atoi(buffer);
if (!(num_pats>0))
  {
  printf("Error: Number of patterns not a positive integer.\n");
  exit(0);
  }

/* display the system parameters for confirmation */
printf("ARTa: num_inputs: %d, num_cats: %d, max_mod_input: %d\n",
              num_inputs_a, num_cats_a, max_mod_input_a);
printf("      rho: %f, beta: %f\n", set_rho_a, beta_a);
printf("ARTb: num_inputs: %d, num_cats: %d, max_mod_input: %d\n",
              num_inputs_b, num_cats_b, max_mod_input_b);
printf("      rho: %f, beta: %f\n", rho_b, beta_b);
printf("Number of patterns: %d, Step size: %d.\n", num_pats, stepsize);

if ((mode == LEARN) || (mode == TRACE))
  learn_mode(fp, argv[2]);
else if (mode == GENER)
  check_mode(fp);

return 0;
}

void check_mode(FILE *filep)
{
int category_a, category_b, correct, unknown, incorrect;

correct=0;
unknown=0;
incorrect=0;
for (count_p=0; count_p<num_pats; count_p++)
  {
  present_input(filep);
  rho_a=set_rho_a;
  category_a=art_a_cat();
  category_b=art_b();
  if (category_a==MAX_CATS_A)
    unknown++;
  else if (category_b==MAX_CATS_B)
    unknown++;
  else if (mapfield[category_a][category_b]==2)
    unknown++;
  else if (mapfield[category_a][category_b]==1)
    correct++;
  else if (mapfield[category_a][category_b]==0)
    incorrect++;
  }
printf("Corr: %d, Incor: %d, Unkn: %d, Num_cats: %d, \n",
        correct, incorrect, unknown, num_used_cats());

fclose(filep);

}

void learn_mode(FILE *filep, char *file)
{
int category_a, category_b, sat, i, j, track;
char filename[FILENAMESIZE];
int correct, incorrect, unknown, steps;

correct=0;
incorrect=0;
unknown=0;
steps=0;
for (count_p=0; count_p<num_pats;count_p++)
  {
  if (mode==TRACE)
    printf("\nPattern: %d.\n", count_p);
  present_input(filep);

  rho_a=set_rho_a;
  track=1;         /* track is reset when we can stop looking */

  while ((rho_a<1.0) && (track!=0))
    {
    sat=0;
    category_a=art_a_cat();
    category_b=art_b();
    if (category_a == MAX_CATS_A)
      {
      sat=1;
      if (track!=2)   /* needed to keep counts correct when tracking */
        unknown++;
      track=0;
      if (mode==TRACE)
        printf("Saturated categories on ARTa.\n");
      }
    else if (category_b == MAX_CATS_B)
      {
      sat=1;
      if (track!=2)
        unknown++;
      track=0;
      if (mode==TRACE)
```

```
        printf("Saturated categories on ARTb.\n");
      }
    else
      {
      if (mode==TRACE)
        printf("Cat_a = %d, cat_b = %d.\n", category_a, category_b);
      /* check for a predicted output, given input */
      if (mapfield[category_a][0]==2)
        {   /* learn category */
        if (track!=2)
          unknown++;
        for (j=0;j<num_cats_b;j++)                /* see equation (3.1) */
          mapfield[category_a][j]=0;
        mapfield[category_a][category_b]=1;
        track=0;
        if (mode==TRACE)
          display_map();
        }
      else
        {   /* check it works */
        if (mapfield[category_a][category_b]!=1)  /* see equation (3.2) */
          {
          if (track!=2)
            incorrect++;

          /* Match Tracking ensues */
          rho_a=1.01*similarity_a;                /* see equation (3.3) */
          if (mode==TRACE)
            printf("Increase rho: %f.\n", rho_a);
          track=2;
          }
        else
          {
            if (track!=2)
              correct++;
            track=0;
          }
        }
      }
    }

  if ((sat == 0) && (rho_a<1.0))
    art_a_learn();
            /* can do learn phase of a if not saturated and rho_a valid */
        steps++;
  if (steps==stepsize)
                {
                printf("Steps: %d, Corr: %d, Incor: %d, Unkn: %d, Cats: %d\n",
                       count_p+1, correct, incorrect, unknown, num_used_cats());
                correct=0;
                incorrect=0;
                unknown=0;
                steps=0;
        }
  }

fclose(filep);

/* save off learnt network */
strcpy(filename, file);
strcat(filename, ".wgt");
printf("Outputting network data to file: %s\n", filename);

if ((filep=fopen(filename, "w")) == NULL)
  {
  printf("Error: Cannot open file for output.\n");
  exit(0);
  }

for (i=0; i<num_cats_a;i++)
  fprintf(filep, "%d ", comm_a[i]);
fprintf(filep, "\n");

for (i=0;i<num_inputs_a;i++)
  {
  for (j=0;j<num_cats_a;j++)
    fprintf(filep, "%d ", topdn_a[j][i]);
  fprintf(filep, "\n");
  }

for (i=0;i<num_inputs_a;i++)
  {
  for (j=0;j<num_cats_a;j++)
    fprintf(filep, "%e ", botup_a[i][j]);
  fprintf(filep, "\n");
  }

for (i=0; i<num_cats_b;i++)
  fprintf(filep, "%d ", comm_b[i]);
fprintf(filep, "\n");

for (i=0;i<num_inputs_b;i++)
  {
  for (j=0;j<num_cats_b;j++)
    fprintf(filep, "%d ", topdn_b[j][i]);
  fprintf(filep, "\n");
  }

for (i=0;i<num_inputs_b;i++)
  {
  for (j=0;j<num_cats_b;j++)
    fprintf(filep, "%e ", botup_b[i][j]);
  fprintf(filep, "\n");
  }

for (i=0;i<num_cats_a;i++)
  {
  for (j=0;j<num_cats_b;j++)
```

```
    fprintf(filep, "%d ", mapfield[i][j]);
  fprintf(filep, "\n");
  }
fclose(filep);
}

void initialise()
{
int i,j;

initialise_a();
initialise_b();

/* mapfield == 2 for undefined, vectors set to 0s & 1s when trained */
for (i=0;i<num_cats_a;i++)
  for (j=0;j<num_cats_b;j++)
    mapfield[i][j]=2;

}

int art_a_cat()
{
int valid, i, left;

clear_active_a();
clear_cats_a();
dot_product_a();
valid=1;
nlearn_a=0;
left=1;
while ((valid != 0) && (left != 0))
  {
  selected_a=get_choice_a();
  valid=do_search_a(selected_a);
  if (valid)
    {
    active_a[selected_a]=1;
    left=0;
    for (i=0;i<num_cats_a;i++)
      if (active_a[i]==0)
        left++;
    if (left==0)
      nlearn_a=1;
    }
  else
    cats_a[selected_a]=1;
  }
if (nlearn_a==1)
  selected_a=MAX_CATS_A;
return selected_a;
}

void art_a_learn()
{
```

```
int i, sum;
if (nlearn_a==0)
  {
  sum=0;
  for (i=0;i<num_inputs_a;i++)
    {
    topdn_a[selected_a][i]&=input_a[i];
    sum+=topdn_a[selected_a][i];
    }
  for (i=0;i<num_inputs_a;i++)
    botup_a[i][selected_a]=topdn_a[selected_a][i]/(beta_a+sum);
  comm_a[selected_a]=1;
  }
}

void clear_cats_a()
{
int i;

for (i=0;i<num_cats_a;i++)
  cats_a[i]=0;
}

void clear_active_a()
{
int i;
for (i=0;i<num_cats_a;i++)
  active_a[i]=0;
}

int do_search_a(int selected)
{
int i, valid;
double sum;

if (comm_a[selected]==0)
  valid=0;
else
  {
  sum=0.0;
  valid=0;
  for (i=0;i<num_inputs_a;i++)
    sum+=(input_a[i] & topdn_a[selected][i]);
  if (sum < mod_i_a*rho_a)
    valid=1;

  /* place value of similarity in global variable */
  similarity_a=sum/mod_i_a;
  }

return valid;
}

int get_choice_a()
```

```
{
int i, selected;
selected=0;
while (active_a[selected])
  selected++;
for (i=selected;i<num_cats_a;i++)
  if ( (act_a[selected]<act_a[i]) && (active_a[i] == 0) )
    selected=i;
return selected;
}

void dot_product_a()
{
int i,j;
for (j=0;j<num_cats_a;j++)
  {
  act_a[j]=0.0;
  for (i=0;i<num_inputs_a;i++)
    act_a[j]+=input_a[i]*botup_a[i][j];
  }
}

void initialise_a()
{
int i,j;
double rnd;
for (i=0;i<num_inputs_a;i++)
  input_a[i]=0;
for (i=0;i<num_cats_a;i++)
  {
  cats_a[i]=0;
  comm_a[i]=0;
  }
for (i=0;i<num_inputs_a;i++)
  for (j=0;j<num_cats_a;j++)
    topdn_a[j][i]=1;
for (j=0;j<num_cats_a;j++)
  {
  while ( !((rnd=drand48()/(beta_a+max_mod_input_a)) > 0.0) ) ;
  for (i=0;i<num_inputs_a;i++)
    botup_a[i][j]=rnd;
  }
}

int art_b()
{
int selected, valid, sum, i, nlearn, left;
clear_active_b();
clear_cats_b();
dot_product_b();
valid=1;
nlearn=0;
left=1;
while ((valid != 0) && (left != 0))
```

```
  {
  selected=get_choice_b();
  valid=do_search_b(selected);
  if (valid)
    {
    active_b[selected]=1;
    left=0;
    for (i=0;i<num_cats_b;i++)
      if (active_b[i]==0)
        left++;
    if (left==0)
      nlearn=1;
    }
  else
    cats_b[selected]=1;
  }
if (nlearn==0)
  {
  sum=0;
  for (i=0;i<num_inputs_b;i++)
    {
    topdn_b[selected][i]&=input_b[i];
    sum+=topdn_b[selected][i];
    }
  for (i=0;i<num_inputs_b;i++)
    botup_b[i][selected]=topdn_b[selected][i]/(beta_b+sum);
  comm_b[selected]=1;
  }
if (nlearn==1)
  selected=MAX_CATS_B;
return selected;
}

void clear_cats_b()
{
int i;
for (i=0;i<num_cats_b;i++)
  cats_b[i]=0;
}

void clear_active_b()
{
int i;
for (i=0;i<num_cats_b;i++)
  active_b[i]=0;
}

int do_search_b(int selected)
{
int i, sum, valid;

if (comm_b[selected]==0)
  valid=0;
else
```

```
  {
  sum=0;
  valid=0;
  for (i=0;i<num_inputs_b;i++)
    sum+=(input_b[i] & topdn_b[selected][i]);
  if (sum < mod_i_b*rho_b)
    valid=1;
  }
return valid;
}

int get_choice_b()
{
int i, selected;
selected=0;
while (active_b[selected])
  selected++;
for (i=selected;i<num_cats_b;i++)
  if ( (act_b[selected]<act_b[i]) && (active_b[i] == 0) )
    selected=i;
return selected;
}

void dot_product_b()
{
int i,j;
for (j=0;j<num_cats_b;j++)
  {
  act_b[j]=0.0;
  for (i=0;i<num_inputs_b;i++)
    act_b[j]+=input_b[i]*botup_b[i][j];
  }
}

void initialise_b()
{
int i,j;
double rnd;
for (i=0;i<num_inputs_b;i++)
  input_b[i]=0;
for (i=0;i<num_cats_b;i++)
  {
  cats_b[i]=0;
  comm_b[i]=0;
  }
for (i=0;i<num_inputs_b;i++)
  for (j=0;j<num_cats_b;j++)
    topdn_b[j][i]=1;
for (j=0;j<num_cats_b;j++)
  {
  while ( !((rnd=drand48()/(beta_b+max_mod_input_b)) > 0.0) ) ;
  for (i=0;i<num_inputs_b;i++)
    botup_b[i][j]=rnd;
  }
```

```
}

void display_map()
{
int i,j;
for (i=0;i<num_cats_a;i++)
  for (j=0;j<num_cats_b;j++)
    printf("Mapfield[%d][%d] = %d.\n", i, j, mapfield[i][j]);
}

void present_input(FILE *filep)
{
int i;
char buffer[BUFFERSIZE];
char *str;
if (fgets(buffer, BUFFERSIZE, filep) == NULL)
  {
  printf("Error: Not enough input patterns.\n");
  exit(0);
  }
str=strtok(buffer, ", ");
for (i=0;i<num_inputs_a;i++)
  {
  if (str==NULL)
    {
    printf("Error: Insufficient items in pattern: %d\n", count_p);
    exit(0);
    }
  input_a[i]=atoi(str);
  if ((input_a[i] != 0) && (input_a[i] != 1))
    {
    printf("Error: Item %d of pattern %d not a 0 or a 1.\n", i, count_p);
    exit(0);
    }
  str=strtok(NULL, ", ");
  }
mod_i_a=0;
for (i=0;i<num_inputs_a;i++)
  mod_i_a+=input_a[i];
if (mod_i_a == 0)
  {
  printf("Error: Modulus of pattern %d on ARTa is zero.\n", count_p);
  exit(0);
  }

if ((mode == LEARN) || (mode == TRACE) || (mode == GENER))
  {
  for (i=0;i<num_inputs_b;i++)
    {
    if (str==NULL)
      {
      printf("Error: Insufficient items in pattern: %d\n", count_p);
      exit(0);
      }
    input_b[i]=atoi(str);
    if ((input_b[i] != 0) && (input_b[i] != 1))
      {
      printf("Error: Item %d of pattern %d not a 0 or a 1.\n", i, count_p);
      exit(0);
      }
    str=strtok(NULL, ", ");
    }
  }
mod_i_b=0;
for (i=0;i<num_inputs_b;i++)
  mod_i_b+=input_b[i];
if (mod_i_b == 0)
  {
  printf("Error: Modulus of pattern %d on ARTb is zero.\n", count_p);
  exit(0);
  }
}

/* read weights from .wgt file */
/* NB: not so much error checking, as file is generated by program */
void read_weights(FILE *filep)
{
int i, j;
char buffer[BUFFERSIZE];
fgets(buffer, BUFFERSIZE, filep);
comm_a[0]=atoi(strtok(buffer, " "));
for (i=1; i<num_cats_a;i++)
  comm_a[i]=atoi(strtok(NULL, " "));
for (i=0;i<num_inputs_a;i++)
  {
  fgets(buffer, BUFFERSIZE, filep);
  topdn_a[0][i]=atoi(strtok(buffer, " "));
  for (j=1;j<num_cats_a;j++)
    topdn_a[j][i]=atoi(strtok(NULL, " "));
  }
for (i=0;i<num_inputs_a;i++)
  {
  fgets(buffer, BUFFERSIZE, filep);
  botup_a[i][0]=atof(strtok(buffer, " "));
  for (j=1;j<num_cats_a;j++)
    botup_a[i][j]=atof(strtok(NULL, " "));
  }
fgets(buffer, BUFFERSIZE, filep);
comm_b[0]=atoi(strtok(buffer, " "));
for (i=1; i<num_cats_b;i++)
  comm_b[i]=atoi(strtok(NULL, " "));
for (i=0;i<num_inputs_b;i++)
  {
  fgets(buffer, BUFFERSIZE, filep);
  topdn_b[0][i]=atoi(strtok(buffer, " "));
  for (j=1;j<num_cats_b;j++)
    topdn_b[j][i]=atoi(strtok(NULL, " "));
  }
```

```
for (i=0;i<num_inputs_b;i++)
  {
  fgets(buffer, BUFFERSIZE, filep);
  botup_b[i][0]=atof(strtok(buffer, " "));
  for (j=1;j<num_cats_b;j++)
    botup_b[i][j]=atof(strtok(NULL, " "));
  }
for (i=0;i<num_cats_a;i++)
  {
  fgets(buffer, BUFFERSIZE, filep);
  mapfield[i][0]=atoi(strtok(buffer, " "));
  for (j=1;j<num_cats_b;j++)
    mapfield[i][j]=atoi(strtok(NULL, " "));
  }
}

int num_used_cats()
{
int i, sum;
sum=0;
for (i=0;i<num_cats_a;i++)
        sum+=comm_a[i];
return sum;
}
```