Location-Independent Neural Network Models

A Dissertation

Presented to the

Department of Computer Science

Brigham Young University

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

by

George Rudolph

July 26, 1995

This Dissertation by George Rudolph is accepted in its present form by the Department of Computer Science of Brigham Young University as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Tony Martinez, Committee Chairman

Dan Olsen, Committee Member

Scott Woodfield, Committee Member

Gordon Stokes, Committee Member

Parris Egbert, Committee Member

Date

Tony Martinez, Graduate Coordinator

TABLE OF CONTENTS

CHAPTER 1

Introduction

Artificial Neural Networks (ANNs) use a different computational paradigm than conventional von Neumann mechanisms. ANNs are composed of nodes and weighted connections between nodes, where each node computes its output based on a function of its weighted inputs. The overall function that a network computes is typically changed by altering the values of the weights between nodes, until the desired result is achieved. The main features of ANNs are learning ability, generalization, parallelism, self-organization and fault-tolerance. These features allow ANNs to solve various applications not handled well by current conventional computational mechanisms. Application areas include, but are not limited to, problems requiring learning, such as pattern recognition, control and decision systems, speech, and signal analysis [Rum86], [DAR88], [Kem92], [Mar89].

Hardware support for ANNs is important for handling large, complex problems in real time. Current typical learning times are too long for applications of any large size and complexity, when implemented with conventional computing schemes. Furthermore, hardware is becoming cheaper and easier to design. The *Location-Independent Transformation* (LIT), the hardware implementation strategy discussed in this dissertation, is a general implementation strategy for ANNs that overcomes several weaknesses of current implementation methods.

Most ANNs use static topologies—the topology is fixed initially, and remains the same throughout learning. ANNs with static topologies often suffer from the following short-comings:

- sensitivity to user-supplied parameters: learning rate(s), etc.,
- local error minima during learning,
- there is no mechanism for deciding on an effective initial topology (number of nodes, number of layers, etc.).

Current research is demonstrating the use of dynamic topologies in overcoming these problems [Dia94], [Fah90], [Kar92], [Mar91], [Odr93], [Rei88]. A *dynamic topology* is one which allows adding and deleting both nodes and weighted connections during learning.

Early ANN implementations are model-specific, and are intended to support only static topologies [Far85], [Gra90], [Mea89]. More recent *neurocomputer* systems have specialized neural hardware, and seek to support more general classes of ANNs [Dia94], [Gam93], [Ham91], [Lin91], [Ram93], [Sha92]. Although some neurocomputers could potentially support dynamic topologies more directly in hardware, rather than in software, they currently do not. Of course, general parallel machines, like the Connection Machine [Hil85] and the CRAY [Alm89], can simulate the desired dynamics in software, but these machines are not optimized for neural computation. LIT supports general classes of ANNs *and* dynamic topologies in an efficient parallel hardware implementation.

LIT redesigns the original network into a hierarchical, parallel network of Location-Independent nodes (LI-nodes). A *LI-node* contains enough information locally to compute its function independent of any other node in the network. The set of LI-nodes is then embedded in a uniform tree topology. Figure 1 gives a general view of LIT. Execution (running the network) and learning are based on efficient broadcast/gather operations in the tree. Because the nodes are location-independent, they can be placed anywhere in the tree, and dynamic topologies can be efficiently and easily supported: Adding nodes is as simple as allocating free nodes in the tree; deleting nodes is as simple as de-allocating the desired nodes.

LITs have been developed for static and dynamic versions of backpropagation [Rud95a,b], static and dynamic versions of counterpropagation and static and dynamic versions of competitive learning [Rud95c], and Adaptive Self-Organizing Concurrent

Systems' Adaptive Algorithm 2 (ASOCS AA2) [Rud95d]. (In addition to the models for which there are specific papers in the dissertation, LITs for a number of variations of these models and others were also investigated and partially developed. However, these additional results are not reported nor discussed in the dissertation.) LITs can potentially support a broad set of ANNs, thus allowing one efficient implementation strategy to support dynamic variations of many ANNs.

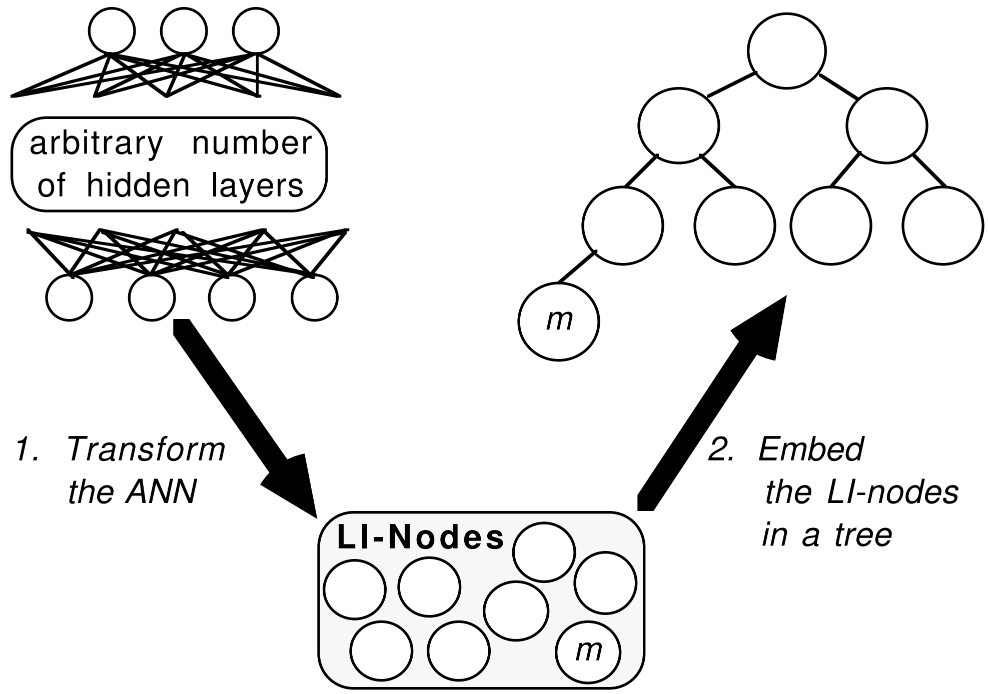


Figure 1. General LIT Transformation

LIT is a generalization of Rudolph's earlier work on a strategy for implementing in hardware a class of learning models called Adaptive Self Organizing Concurrent System (ASOCS) [Mar88], [Mar91a,b], in particular AA2 [Mar91b], which have inherently dynamic topologies. The initial results of that work were the Location-Independent ASOCS model (LIA) [Rud95d] and support for the Priority ASOCS model [Mar94a]. While the underlying mechanisms of ASOCS may differ significantly from those common among ANNs, the overall goals of ASOCS are similar to other such learning models, as is the need to support dynamic topologies.

The first contribution of this dissertation is, thus, the general LIT strategy for efficiently supporting hardware implementations of static and dynamic ANNs. The strategy could be used in software design as well, however the intent is directed at hardware mechanisms rather than software.

The second contribution of this dissertation relates more specifically to ASOCS. It is a PASOCS chip which was developed by Stout and Rudolph as a proof-of-concept of LIT ideas [Sto94b] and an investigation of the use of Multi-chip Modules in handling the dense interconnect required by ANNs [Sto94a]. Hardware implementations of static versions of Backpropagation and other popular models are ubiquitous in ANN literature. To date, however, only three hardware implementations of ASOCS models have been designed, fabricated and tested, each one using different approaches. The first of these was Jeff Chang's direct implementation of AA2 using logic arrays [Cha87], which only included partial on-chip learning. The other two, Bartczak's Hardware Adaptive Algorithm (HAA) [Bar94], and Stout, et. al.'s PASOCS (Priority ASOCS) chip were developed more or less concurrently, but independently. HAA uses a Content-Addressable Memory with an associative search to implement AA2. Bartczak and Rudolph have discussed the idea that

HAA could fit within the LIT strategy—HAA could be viewed as an implementation of LIA as well as AA2, in fact—and is, in some sense, independent proof-of-concept of the LIT approach [personal communication]. The PASOCS chip directly implements the logical structures described for PASOCS in [Mar94a], basically a binary tree of nodes. The main difference between LIA and PASOCS is that PASOCS uses priorities to resolve conflicts among learned data, while LIA uses discriminating variables (i.e. attributes) to resolve conflicts.

The third contribution of this dissertation also dovetails with ASOCS research, and is in the area of applications research. One goal of the LIT research is to develop hardware solutions for those applications which can take advantage of direct hardware implementations (as mentioned above). The paper on adaptive network routing [Mar94b] represents further research based on initial ideas originally proposed by McDonald and Martinez [McD91].

Finally, the appendix includes software simulations of three models as an illustration and verification of LIT concepts: the standard static Backpropagation and static Competitive Learning models [Rum86] and the inherently dynamic LIA model. The correctness of the LIT versions of Backpropagation and Competitive Learning is verified by comparison with the results of the well-known PDP software versions of these models [Rum88]. The correctness of LIA is verified by comparing its results to those for AA2 given in Martinez's PhD Dissertation [Mar86].

1. Thesis Statement

This dissertation introduces and describes Location-Independent Transformations (LITs), which are a general strategy for implementing *Artificial Neural Networks* (ANNs).

A *Location-Independent Transformation* transforms an ANN into a set of location-independent nodes (LI-nodes). A *LI-node* contains enough information locally to compute its function independent of any other node in the network. The set of LI-nodes is then embedded in a uniform tree topology. Execution (running the network) and learning are based on efficient broadcast/gather operations in the tree. Because the nodes are location-independent, they can be placed anywhere in the tree, and dynamic topologies can be efficiently and easily supported: Adding nodes is as simple as allocating free nodes in the tree; deleting nodes is as simple as de-allocating the desired nodes.

The dissertation describes, in detail, LITs for a number of important ANNs, including backpropagation, counterpropagation, competitive learning networks and ASOCS. A proof-of-concept hardware implementation of a network using the LIT approach is described. One application that could potentially use such hardware is also discussed.

2. Outline of the Dissertation

This dissertation is a collection of published or currently submitted original papers. Each paper is prefaced by introductory paragraphs which provides historical and logical links to the rest of the dissertation and various research areas. Chapters 2-8 are copies of the original papers reformatted into a consistent format for the dissertation. Hence, redundancy, particularly in the introductory sections of chapters 2-5, is to be expected. The material in the appendix has not previously been published in any form. Abstracts and acknowledgments have been omitted, and references have been compiled into a single bibliography. When necessary, permission to reprint previously published papers has been obtained from the rightful owners of copyrights, which were transferred upon publication.

Chapter 2 discusses localist neural networks. *Localist* means that a single node or prototype ultimately determines the output of the network. The two networks used as examples are a simple Competitive Learner [Rum86] and the Counterpropagation Network [Hec87]. The paper also includes a simple dynamic extension to the standard learning algorithm for illustration purposes. The transformations for localist models are the best place to start, when wishing to understand LIT, because the models are simple and the transformations are straightforward.

Chapter 3 discusses 2-layer ANNs with distributed representations. *Distributed* means that responsibility for determining the output of the network is distributed among many (perhaps all) nodes, not just one. The number of layers refers to the number of weight layers. The network used here, which is the most common and popular of all ANNs, is the standard Backpropagation network [Rum86]. A dynamic extension to the standard Backpropagation learning algorithm, based on the algorithm of [Odr93] is also presented as an illustration of a dynamic topology.

Chapter 4 extends the LIT for 2-layer networks of chapter 3 to handle an arbitrary number of layers. Chapters 3 and 4 are important results in that they indicate one way in which distributed ANNs can be represented in a localist fashion without loss of generality, functionality and efficiency. While some ANNs have only two layers (as in the network used in Chapter 3), most ANNs have more than two layers of weights. Hence, Chapter 4 is an important step toward making LIT general enough to support most ANNs, because an arbitrary number of layers can be supported.

Chapter 5 presents the LIA model which, like all ASOCS models, uses an inherently dynamic topology. This chapter presents essentially a journal version of Rudolph's MS Thesis [Rud91b]. However, the paper also goes beyond the MS Thesis in two ways: 1) The paper places LIA within the LIT framework, which was only beginning to be developed as a consequence of the Thesis, and 2) The paper provides formal descriptions of basic ASOCS instances, operators, instance sets, and execution and learning algorithms. These formalizations were found to be useful for hardware implementations by Stout, et. al. [Sto94b].

Chapter 6 contains a schematic-level description of the PASOCS prototype VLSI chip [Sto94b]. The chip was fabricated in 2μm digital CMOS. It was designed using a VLSI layout editor, and simulated using an event-driven, switch-level simulator. The actual circuit was designed using simple cells from standard cell libraries, and no special-purpose cells were required.

Chapter 7 presents a discussion on the use of Multi-Chip Modules (MCM) as an interconnect medium for ANNs. The paper focuses on MCM ideas, and a description of the MCM substrate—the PASOCS chip/network (of chapter 6) is used as an example of how an ANN can be embedded within the MCM framework.

Chapter 8 presents an exploratory paper on an application of ASOCS to adaptive network (as in LANs an WANs, not neural networks) routing.

Chapter 9 is discussion of results and verification of software simulations of two models covered in the dissertation: Competitive Learning and Backpropagation. The simulations for the Competitive Learner and Backpropagation (static versions) were written in C++ using the CFront compiler. The actual code for the simulations is in the Appendix.

Chapter 10 is the Conclusion. It summarizes the major results of the dissertation, and then discusses potential future research directions.

CHAPTER 2

A Transformation For Implementing Localist Neural Networks

Accepted by the Journal of Neural, Parallel and Scientific Computations © 1995

LIT is a generalization of Rudolph's earlier work [Rud91a] on a strategy for implementing in hardware a class of learning models called Adaptive Self Organizing Concurrent System (ASOCS) [Mar88], [Mar91a,b], in particular AA2 [Mar91b], which have inherently dynamic topologies. The initial results of that work were the Location-Independent ASOCS model (LIA) [Rud95d] and support for the Priority ASOCS model [Mar94a].

Both the LIA and PASOCS models are examples of "localist" models. *Localist* means that a single node or prototype ultimately determines the output of the network. Thus, investigation of hardware implementations for other localist models is a reasonable place to begin applying and developing the ideas for LIT. The two networks used as examples here are a simple Competitive Learner [Rum86] and the Counterpropagation Network [Hec87]. The paper also includes a simple dynamic extension to the standard learning algorithms of these two models for illustration purposes.

Shorter versions of this paper with slightly different emphases also appear in Intelligent Systems [Rud95c], and the International Journal of Artificial Intelligence Tools [Rud95e].

1. INTRODUCTION

Artificial Neural Networks (ANNs) use a different computational paradigm than conventional von Neumann mechanisms. ANNs are composed of nodes and weighted connections between nodes, where each node computes its output based on a function of its weighted inputs. The overall function that a network computes is typically changed by altering the values of the weights between nodes, until the desired result is achieved. The main features of ANNs are learning ability, generalization, parallelism, self-organization and fault-tolerance. These features allow ANNs to solve various applications not handled well by current conventional computational mechanisms. Application areas include, but are not limited to, problems requiring learning, such as pattern recognition, control and decision systems, speech, and signal analysis [DAR88].

Hardware support for ANNs is important for handling large, complex problems in real time. Learning times can exceed tolerable limits for complex applications with conventional computing schemes. Furthermore, hardware is becoming cheaper and easier to design. The *Location-Independent Transfromation* (LIT) is a general implementation strategy for ANNs that overcomes several weaknesses of current hardware implementation methods.

Most ANNs use static topologies—the topology is fixed initially, and remains the same throughout learning. ANNs with static topologies often suffer from the following short-comings:

- sensitivity to user-supplied parameter problems: learning rate(s), etc.,
- local error minima during learning,
- there is no mechanism for deciding on an effective initial topology (number of nodes, number of layers, etc.)

A *dynamic topology* is one which allows adding and deleting entire nodes and individual weighted connections during learning. Current research is demonstrating the use of dynamic topologies in overcoming these problems [Fahlm], [Mar91b], [Odr93], [Rei88] for a variety of applications. [Hay94], for example, shows that Radial-Basis Function Networks (RBF) with a dynamic topology performed significantly better on NETtalk than the same RBF network with fixed topology.

Early ANN hardware implementations are model-specific, and are intended to support only static topologies [Far85], [Gra90], [Mea89]. More recent *neurocomputer* systems have specialized neural hardware, and seek to support more general classes of ANNs

[Ham91], [Ram93], [Sha92]. Although some neurocomputers could potentially support dynamic topologies more directly in hardware, rather than in software, they currently do not. Of course, general parallel machines, like the Connection Machine [Hil85] and the CRAY [Alm89], can simulate the desired dynamics in software, but these machines are not optimized for neural computation. LIT supports general classes of ANNs *and* dynamic topologies in an efficient parallel hardware implementation.

LIT redesigns the original network into a hierarchical, parallel network of Location-Independent nodes. A *LI-node* contains enough information locally to compute its function independent of any other node in the network. The set of LI-nodes is then embedded in a uniform tree topology. Figure 1 gives a general view of LIT. Execution (running the network) and learning are based on efficient broadcast/gather operations in the tree. Because the nodes are location-independent, they can be placed anywhere in the tree, and dynamic topologies can be efficiently and easily supported: Adding nodes is as simple as allocating free nodes in the tree; deleting nodes is as simple as de-allocating the desired nodes.

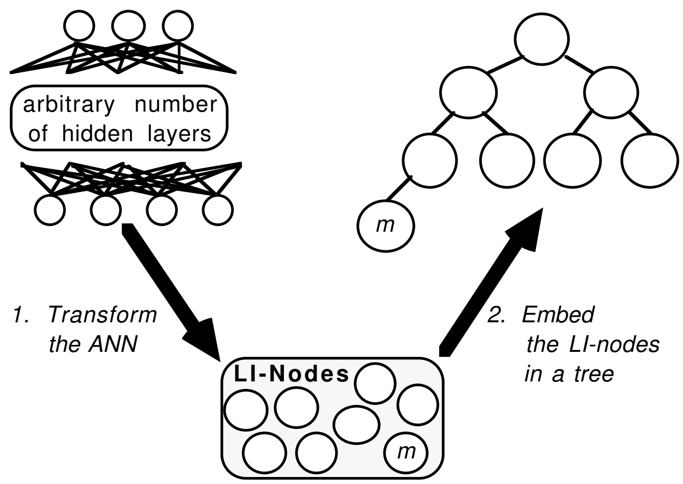


Figure 1. General LIT Transformation

LITs have been developed for backpropagation [Rum86], and Adaptive Self-Organizing Concurrent Systems' Adaptive Algorithm 2 [Mar91b], [Rud91b]. Transformations for other important ANNs are also being developed. LITs can potentially support a broad set of ANNs, thus allowing one efficient implementation strategy to support dynamic variations of many ANNs.

This paper presents the transformations for two networks, the competitive learning (CL) network with a single weight layer, and the counterpropagation network (CPN). CL and CPN are examples of *localist* networks, i.e. for each possible output, there is a single node that is ultimately responsible for that output. CL uses unsupervised learning, while CPN combines elements of competitive and supervised learning. Section 2 presents an overview of LIT. Section 3 describes the LIT for CL. Section 4 descibes the LIT for CPN. Section 5 presents a brief complexity analysis of the LIT algorithms for CL and CPN. Section 6 is the conclusion.

The standard CL and CPN models have static topologies. For the purpose of illustrating dynamic topologies in this paper, the standard learning algorithms are extended with the following rules:

- if the winning node's activation is less than some threshold θ for the current input, then add a new node which classifies the input at its center.
- if a node never wins during the current epoch, then delete that node at the end of the epoch.

LIT is not meant to include devising dynamic extensions to static models *per se*. The existence of dynamic extensions to static models, and the existence of inherently dynamic models such as RCE [Rei88], and dynamic RBF networks [Hay94] is assumed. Furthermore, the literature demonstrates the stability and applicability of various models and dynamic extensions. The CL and CPN networks, and the dynamic extension proposed here should be sufficient to indicate how LIT supports localist ANNs, whether these have static, extended dynamic, or inherently dynamic topologies. Note that LIT will support a particular dynamic topology regardless of the stability of the algorithm being used—LIT does not affect stability.

2. THE TRANSFORMED NETWORK

LIT redesigns the original network into a hierarchical, parallel network in which each node contains enough information locally to compute its part of the network output, independent of any other node. A network whose nodes have this property is said to be *location-independent.* The nodes also are location-independent—regardless of the physical location of any node in the network, the relative order in which they compute results, or the order in which those results are gathered, *the individual computations are the same, and the network output is the same*. Furthermore, because a node's information is local, adding or deleting nodes from the network can be done without affecting any other nodes. Thus, location-independence allows efficient support for dynamic topologies.

In this paper, the term *Control Unit* refers to a mechanism that broadcasts inputs to a network, and gathers results from it. The term *original* refers to a network before it is transformed, and the term *transformed* refers to a network after it has been transformed. These terms apply similarly to the nodes as well. The number of layers refers to the number of *weight layers*.

LIT is a two-step process:

1. Construct a set of LI-nodes based on the original model.
2. Embed the nodes in a communication topology (tree).

This process is outlined in figure 2. Based on an original ANN model (left), a set of LI-nodes is constructed (middle), and the nodes are embedded in a tree (right).

LIT is not simply a reorganization of the original network, but rather involves redesigning the basic structure. The heart of a transformation is the construction step: It defines the mapping between the original network and the set of LI-nodes. The original network in figure 2 is transformed to a network with five 4x3 (4 inputs, 3 outputs) LI-nodes. The mapping, in turn, affects how the behavior of the original is modeled in the transformed network. Thus, a construction typically also involves reformulating the network equations, in order to describe precisely the behavior of the transformed network. Although the original and transformed networks compute equivalent functions, correspondences between the two are not always direct, nor obvious. Constructions vary across different ANNs. LI-nodes for CL, which has only a single weight layer, do not have the same structure as LI-nodes for multilayer backpropagation.

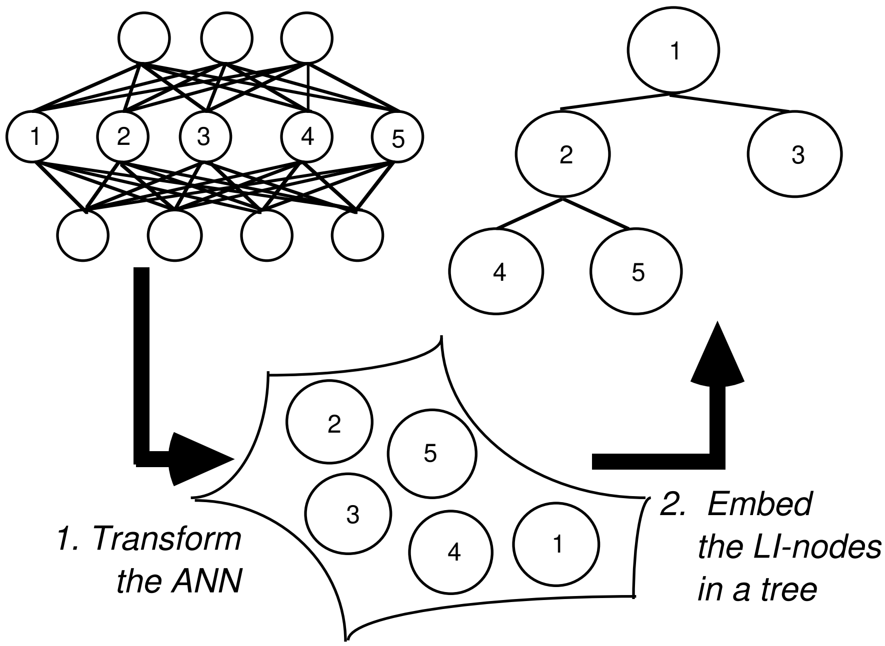


Figure 2. An Example Location-Independent Transformation

A communication topology can be chosen as a matter of speed, cost and reliability. The same topology can be used with many models. A binary tree topology is specified in the second step in figures 1 and 2. It is a simple, fast, regular, hierarchical topology that naturally supports broadcast and gather operations. It provides a basis for applying the transformation and explaining the associated execution and learning algorithms, without obscuring the details with a complex interconnect. However, a hypercube, or some other more fault-tolerant topology, could also be used. Such a hierarchical topology allows adding or deleting nodes with time logarithmic in the number of nodes in the network, while limiting connectivity among nodes. The connectivity does not explode because each node is connected to one parent and zero to two children. The algorithms for the transformed network show explicitly how the required communication can take place.

An example illustrates the support for dynamic topologies. When a node is added to (deleted from) an original network, both the node and its connections to other nodes are added (deleted). If a hidden node 6 were added to the original network in figure 2, four connections to the input nodes and three connections to the output nodes would also be added. This is accomplished in the transformed network by allocating a free node, such as the left child of node 3, and initializing it as a 4x3 node with the corresponding weights. The deletion of a hidden node in the original network, such as node 5, is accomplished in the transformed network by marking the corresponding node as “free”. No other nodes are affected.

A prototype VLSI chip set has been fabricated as proof-of-concept of the overall LIT strategy [Sto94b].

3. TRANSFORMATION OF A COMPETITIVE LEARNING NETWORK

The goal of the CL model is to spontaneously classify sets of similar inputs to the same class, and sets of different inputs into different classes, according to critical features discovered by the network [Rum86]. The original CL has one output node for each output class, and weights from each input node to every output node. The output nodes are classical sum-of-products nodes—the node with the highest activation for a particular input is chosen as the output class for that input. (The input nodes are place-holders for the input values.) Thus, the output of the network is the class of the input. During learning, only the winning node adjusts its input weights—the other nodes make no changes. The winning node causes its input weights to change so that the weights are more similar to the current input vector.

The original CL equations (see section 3.1) and structures reveal the localist nature of computation in the original CL model:

- The activation value for each class is computed locally at each output node.
- A single, global winner node determines the output of the network.
- Determination of a global winner can be accomplished using localized comparisons of subsets of the output nodes.

The LIT construction given here is based on these ideas.

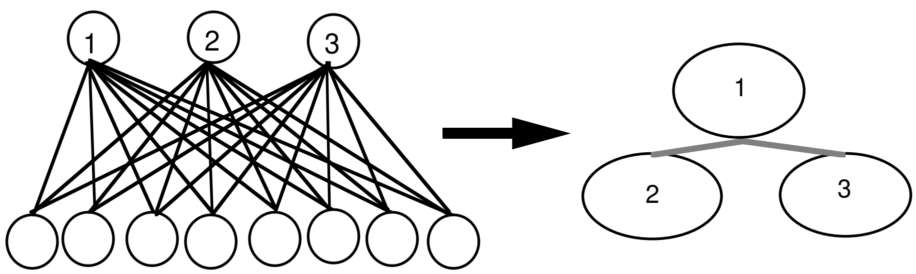


Figure 3. Transforming a CL Network

Figure 3 shows the transformation of a network into a LIT network with three 8x2 nodes. There is one node in the transformed network for each original output node. A transformed CL node stores two vectors. One vector contains weights on inputs $\boldsymbol{w_j}$, whose elements w_{ij} correspond to the weight connected to the respective original input node. The other vector contains the activation value net_j, and the class $class_j$ (or class number j) as a pair. The original node 2, in figure 3, has eight weights on inputs, and outputs its class number 2. The transformed node 2 has eight weights in $\boldsymbol{w_j}$ and outputs the tuple (net_j, $class_j$).

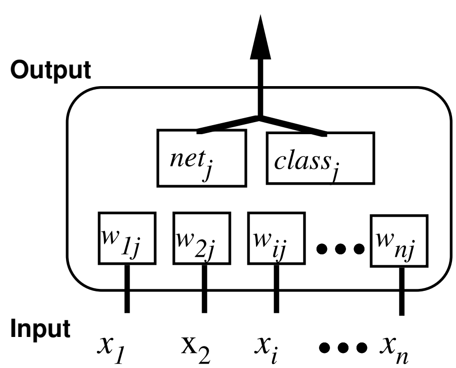


Figure 4. A CL Node

Figure 4 shows the structure of a new node in more detail. Each transformed node outputs its activation because determining a global winner requires non-local access to the activation values of each node. In the original network, the output nodes compete in some fashion, the node with the highest net activation being the winner, and all others being “quenched” somehow. The mechanisms for quenching are usually assumed, not explicitly given, but typically require that each node has access to the activation of every other node. The transformed network, on the other hand, uses a gather operation as an explicit "quenching" mechanism. The gather also localizes the competition, in that each node does not require the activations of every other node to decide a winner. Each node compares its own activation with the activations in the tuples received from its children. The tuple with the highest activation is sent to the parent, the others are discarded. In case of a tie, the tuple with the lowest class number is chosen. Thus, at any given transformed node, at most three tuples are compared, regardless of the size of the network. The output of the root node is the tuple which contains the class that has the highest activation value.

3.1 Execution Algorithm

Definitions

net_j	real-valued activation of node j
$class_j$	output class of the jth node, which is j itself (in this case)
w_{ij}	real-valued, initially random, weight between the original ith input node and the original jth output node (resides in the jth transformed node)
x_i	ith component in the Boolean input vector $\boldsymbol{X}$
m	number of output nodes in the original network, also the number of nodes in the transformed network
n	length or size of $\boldsymbol{X}$

Equations

$$(net, class) = (net_j, class_j) \text{ where } j = \min\{a \mid \forall(net_a, class_a): net_a = \max\{net_b \mid 1 \le b \le m\}\} \tag{1}$$

$$net_j = \sum_{i=1}^{n} w_{ij} x_i \tag{2}$$

The execution algorithm proceeds as follows:

1. Broadcast the input vector $\boldsymbol{X}$ to the network.
2. Each node j computes a sum-of-products activation, net_j, as described [equation (2)].
3. Each node compares its net_j with those of its children (if any). The highest activation (and corresponding class) are sent up, the others are discarded. If there is a tie in activation, the activation with the lowest class number is sent up to the parent [equation (1)].
4. The class of the node with the highest activation is the class of the input vector $\boldsymbol{X}$, and hence the output of the network (to the Control Unit).

3.2 Learning Algorithm

Additional Definitions

v	total number of 1's in the input vector (determined by counting)
g	real-valued learning constant
ε	real-valued, maximum acceptable error (usually predefined)

Equations

$$\Delta w_{ij} = g\left(\frac{x_i}{v} - w_{ij}\right) \tag{3}$$

The learning algorithm is as follows:

Until convergence (current epoch = desired number of epochs)

For each pattern $\boldsymbol{X}$ in the training set

1. Broadcast the input vector $\boldsymbol{X}$ to the network.
2. Each node j computes a sum-of-products activation, net_j, as described [equation (2)].
3. Each node compares its net_j with those of its children (if any). The highest activation and corresponding class are sent up to the parent, the others are discarded. If there is a tie in activation, the activation with the lowest class is sent up [equation (1)].
4. The class of the node with the highest activation is the class of the input vector $\boldsymbol{X}$, and hence the output of the network.
5. If the highest activation is less than threshold θ, goto step 9.
6. Broadcast the class of the winner to the network.

7. The selected (winning) node changes its weights appropriately, using the learning equation (3). All other nodes make no changes.
8. Goto to step 1 with next pattern ***X***.
9. Add a new node in the network—initialize the weights to be equal to ***X***.
10. If at the end of the current epoch, then any node that has not won at least once self-deletes from the network.

One note on the CL model: The learning equation (3) (see [Rum86]) gives a very non-robust type of learning. Both the zero vector and the vector with all ones will always be put in the class with the lowest number, because in each case, the activations of all nodes is the same. This helps to illustrate one important point of this paper—a different, more robust learning equation (algorithm) could be substituted, and the overall transformation would still be the same.

4. TRANSFORMATION OF A COUNTERPROPAGATION NETWORK

The goal of CPN is to learn approximations to input-output pairs presented to the network [Hec87]. The original network is presented as a four-layer "counterpropagation" flow between five sets of nodes (figure 5). However, CPN can be viewed as single-layer competitive learner with adjustable output weights (without loss of generality).

Throughout section 4, it is necessary to refer to node layers and weight layers both. To avoid confusion, *layers* refers to weight layers, and *sets* refers to node layers.

4.1 Transformation of CPN

The organization of the CPN can be simplified as follows:

1. Node sets 1 and 5 are combined into a single input set, divided into a ***(x,y)*** vector pair as desired, where n is the size of $\boldsymbol{x}$, and k is the size of $\boldsymbol{y}$.

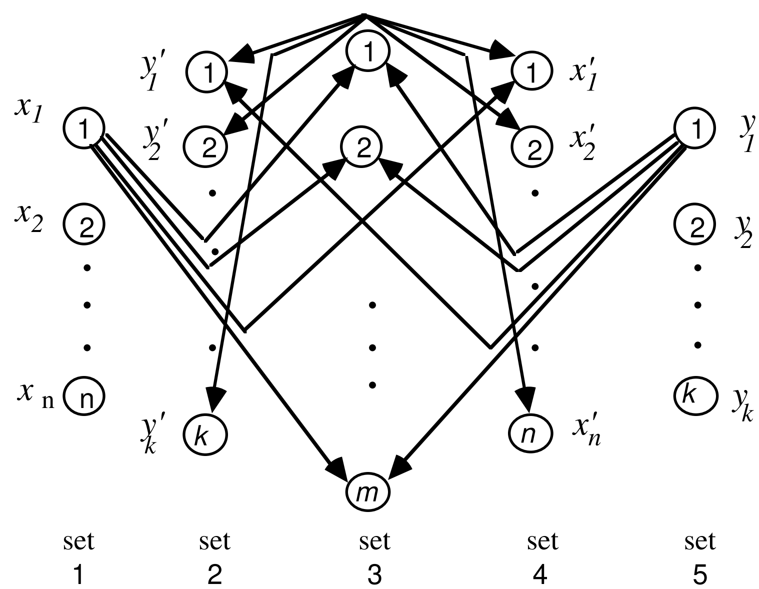


Figure 5. Original CPN Structure

2. Node sets 2 and 4 are combined into a single output set, divided into a ***(x', y')*** vector pair corresponding to ***(x,y)***.
3. Node set 3 remains as in the original description.
4. There is a component-wise weight layer from sets 1 and 5 to sets 2 and 4 respectively, used only during learning. This is a pass-through layer whose weight values (set to 1) never change. After the transformation, it will not be needed.

5. The typical competitive method for determining the winner among the nodes of set 3 involves connections from each node in the set to every other node in the set. The transformed CPN network performs the selection in the same fashion as the CL network (see section 3). Hence, these connections can be ignored in the present discussion.

At this point, the network can be seen as a 2-layer (three, counting the pass-through layer) feed forward network with one set of hidden nodes (see figure 6), without loss of functionality. However, one more reorganizing step will show that CPN can be considered a competitive learner with the addition of adjustable weights as output. Several additional notational simplifications, along with relabelling (noted in figure 6), done for the purposes of this paper, are given to clarify this:

- The input vector pair (***x*,*y***) is treated as a single vector ***X***, with components x_i, where $1 \le i \le k+n$.
- Similarly, the output vector pair (***x', y'***) is treated as a single vector ***Y***, with components y_i, where $1 \le i \le k+n$.

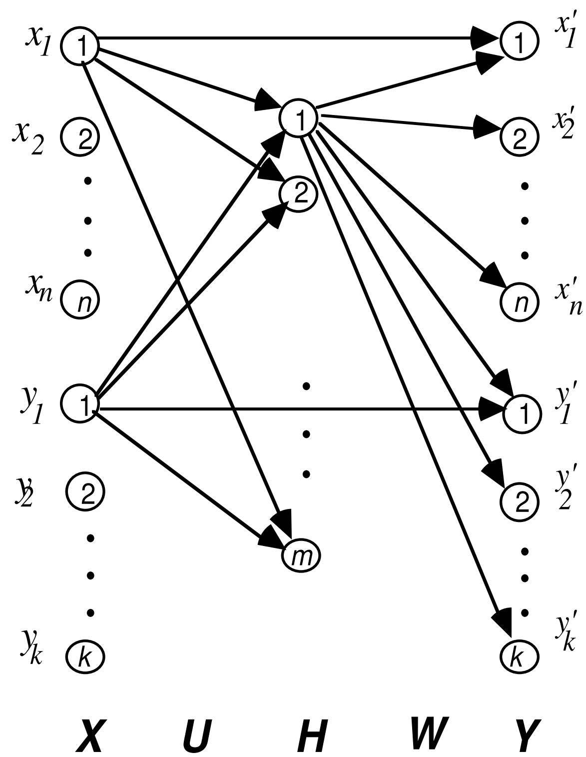


Figure 6. Rearranged CPN Network

- The input weight vectors ***u*** and ***v*** are treated as a single weight vector ***U***, with components u_{ij}, where $1 \le i \le k+n$, $1 \le j \le m$
- Node set 3 is labeled ***H***, where h_i is the *i*th node in that set.
- The output weight vector ***W*** (for ***x'*** and ***y'***) is treated as a single vector ***W***, where w_{ij} is the weight from h_i to y_j, and $1 \le i \le m$, $1 \le j \le k+n$.

The nodes are classical sum-of-products nodes. The nodes in set ***H*** compete in winner-take-all fashion, hence this hidden set acts as a selector for the output vector. Because ***H*** is a selector, there is no recombining of the weights in layer ***W*** to determine the output of the network, in either execution or learning mode. Only the winning node's output weights are (effectively) transmitted to the output layer ***Y***, because all other results are zero. This suggests that the real work of determining the output involves the weights in ***U***, the first weight layer, and the nodes in ***H***. Hence, CPN behaves as a spontaneous classifier like the original CL model of section 3: ***H*** corresponds to the output nodes, ***U*** corresponds to the single weight layer, and $\boldsymbol{X}$ corresponds to the input nodes. What CPN has, that CL does not, is $\boldsymbol{W}$, which are adjustable output values associated with each output class.

In transforming the CPN network, there are m nodes in the transformed network, one corresponding to each node in $\boldsymbol{H}$. The structure of a transformed CPN node is shown in figure 7. Each node has a fixed class number, which is also its priority and index, from 1 to m. In case of a tie, the node with the lowest index wins. A node stores $2(k+n)$ weights: $k+n$ weights on inputs $\boldsymbol{u_j}$, and $k+n$ weights on output $\boldsymbol{w_j}$. The weights that each transformed node stores are the input and output weights of the corresponding original node.

There are four ways in which the CPN network behaves differently from the CL network:

1. Each node outputs $\boldsymbol{w_j}$ in addition to net_j and $class_j$.
2. The winning node adjusts both its $\boldsymbol{w_j}$ and its $\boldsymbol{u_j}$
3. The input and output vectors can contain real values.
4. The learning equations (6) and (7) are different.

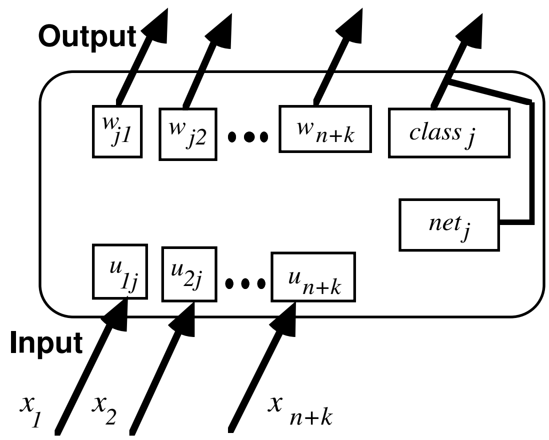


Figure 7. Structure of a Transformed CPN node

4.2 Execution Algorithm

The basic goal of execution is to retrieve patterns stored during learning. In the simplest case, this means retrieving the approximation $\boldsymbol{Y}$ given the input $\boldsymbol{X}$ or an input similar to $\boldsymbol{X}$.

<u>Definitions</u>

$\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{H}$, $\boldsymbol{U}$, $\boldsymbol{W}$ are defined as above. The domain of all of these is real. Note that the original component vectors $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ (as defined by Hecht-Nielsen) are all initially normalized, unit-length vectors.

- net_j real-valued activation of new node j
- $class_j$ output class of the jth node, which is j itself (in this case)
- u_{ij} real-valued, initially random normalized, weight between x_i and h_j (resides in the jth transformed node)
- w_{ij} real-valued, initially random normalized, weight between h_i and y_j (resides in the jth transformed node)
- $\boldsymbol{w_j}$ the set of output weights activated by node j (residing in the jth transfromed node)
- m number of output nodes in the original network, also the number of nodes in the transformed network
- n length or size of $\boldsymbol{X}$ and $\boldsymbol{Y}$

The execution equations for the transformed model are almost identical to those formulated for the CL model above. The only difference is that the output from each node is a triple, instead of a tuple. The additional information is the weight vector $\boldsymbol{w_j}$, sent up by each node along with the class number and activation.

Equations

$$(class,net,w) = (class_j,net_j,w_j) \text{ where } j = \min\{a \mid \forall (class_a,net_a,w_a): net_a = \max\{net_b \mid 1 \le b \le m\}\} \tag{4}$$

$$net_j = \sum_{i=1}^{n} u_{ij} x_i \tag{5}$$

The execution algorithm proceeds as follows:

1. Broadcast the input vector $\boldsymbol{X}$ to the network.
2. Each node j computes a *sum-of-products* activation, net_j, as described [equation (5)].
3. Each node compares its net_j with those of its children (if any). The highest activation (and corresponding class number and $\boldsymbol{w_j}$) are sent up to the parent, the others are discarded. If there is a tie in activation, the activation with the lowest output class is sent up [equation (4)].
4. The $\boldsymbol{w_j}$ associated with the highest activation is the $\boldsymbol{Y}$ that corresponds to the input vector $\boldsymbol{X}$.

There is a potentially more efficient way to gather the output vector $\boldsymbol{Y}$, which requires an extra step. In step 3, each node sends up only the activation and output class, (as in the CL model). In step 4, the Control Unit receives the activation and class only, not the output vector. In step 5 then, the winning class index is broadcast back to the network, which selects the winning node, and "quenches" all the others. The winning node can immediately send its output vector up the network to the Control Unit.

4.3 Learning Algorithm

The basic goal of learning is to generate approximations to the set of training vectors. That is, given input $\boldsymbol{X}$ learn some approximation $\boldsymbol{Y}$. There are two different learning equations, one for layer $\boldsymbol{W}$, and one for layer $\boldsymbol{U}$, equations (6) and (7), respectively.

Additional Definitions

a,b,c,α	learning constants
error	total sum squared $(\Delta w+\Delta u)$
ε	total maximum acceptable error

Equations

$$\Delta w_{ij} = (-aw_{ij} + bx_i), \text{ if } i \le n \tag{6}$$

$$\Delta w_{ij} = (-aw_{ij} + cx_i), \text{ if } n < i \le k+n$$

$$\Delta u_{ij} = \alpha(\boldsymbol{X}-u_{ij}) \tag{7}$$

The learning algorithm proceeds as follows:

Until convergence ($error \le \varepsilon$)

For each pattern $\boldsymbol{X}$ in the training set

1. Broadcast $\boldsymbol{X}$ to the network.
2. Each node computes its net_j, and sends the result and its output class to its parent.
3. Each parent compares its net_j with those received from its children. The highest net_j and corresponding class are sent up the tree.
4. The index with the highest net_j wins.
5. If the highest activation is less than threshold θ, then goto step 9.
6. Broadcast the output class of the winner to the network.

7. The chosen node changes its weights appropriately, using the corresponding learning equations (6) and (7) above. All other nodes make no changes.
8. Goto to step 1 with next pattern $\boldsymbol{X}$.
9. Add a new node in the network—initialize the weights so that the node outputs $\boldsymbol{Y}$ with activation 1.0 for input $\boldsymbol{X}$.
10. If at the end of the current epoch, then any node that has not won at least once self-deletes from the network.

The learning algorithm guarantees that only one node can win a competition. The learning equations show that a winning node will change its input weights to respond more strongly to the current input, and that the output weights will be adjusted to be more like the desired output [Hec87].

5. COMPLEXITY OF LIT CL AND LIT CPN ALGORITHMS

The complexity of the algorithms in sections 3 and 4 can be characterized by serial, pipelined broadcast and gather in a tree of nodes. The width of a broadcast block is assumed to be the size of a single variable. The tree has m nodes, one for each node in the (transformed) network, and the nodes operate in parallel. Assume that each node has n inputs. For CL, each node has two outputs, and for CPN, each node has $n+2$ outputs. Given these assumptions, the following analysis gives an overall complexity of $O(n+logm)$ for both execution and learning for both CL and CPN, for a single pattern.

The execution algorithm is outlined as follows. The numbers to the left are for convenience in referring to each step:

Broadcast

For the nodes in level 1 up to $\lfloor logm \rfloor$+1 do
1. receive an input variable from the parent node
2. pass it on to the two children nodes
3. compute the activation value of the node so far

end.

Strictly speaking, computing the activation is not part of broadcast, but it is convenient to view it as such, since the two overlap almost completely.

Gather

For the nodes in level $\lfloor logm \rfloor$+1 downto 1 do
4. receive (net_j, $class_j$) value pairs from the two children (if children exist)
5. determine winning pair
6. pass the result up to the parent
7. receive additional output values, if there are any, from the winning child
8. pass these onto the parent

end.

net_j and $class_j$ values can be sent up in successive steps, and used in pipelined computations at successive levels. For brevity, it is convenient to treat them as a single entity with two variables, however.

Broadcast complexity is the same for CL and CPN. Gather complexity is different, however, because CPN has n output values in addition to the (net_j, $class_j$) pair, while CL has no additional values. Even though the complexity of the gather is different, the overall complexity, taking broadcast and gather together, is the same for both models.

Steps 1, 2 and 3 are all $O(n)$ operations, for n variables. Because of pipelining, they overlap, and require $n+2$ steps (steps 2 and 3 are concurrent). The overall complexity of broadcast would thus be $O(nlogm)$, but the pipelining reduces this to $O(n+logm)$.

Step 4 is constant, since there are always four values to be received from the children. Step 5 is also constant, since at most four comparisons are needed to determine a local winner—two for the activation, and two for the class. Step 6 is constant, since only one activation and one class value are passed to each parent. Steps 7 and 8 do not apply to CL, but are both $O(n)$ operations, for n variables, for CPN. Thus, the overall complexity of gather is $O(logm)$ for CL, and $O(n+logm)$ for CPN (the n variables can be passed up in pipelined fashion).

The above shows that combined complexity of broadcast and gather for the execution algorithms for both CL and CPN is $O(n+logm)$.

The learning algorithms have essentially the same complexity as the execution algorithms—they require two additional actions:

1. a broadcast which selects the winning node, or selects a free node, and
2. a node operation that either updates the winner's weights or initializes the selected free node, respectively.

The first action is $O(logm)$; the second is $O(n)$ for the update,and $O(n+logm)$ for initialization. Thus, the combined complexity is $O(n+logm)$.

The analysis presented in this section shows that the complexity of both learning and execution algorithms for both ANNs is $O(n+logm)$, for a single pattern.

6. CONCLUSION

ANNs that use a static topology, i.e. a topology that remains fixed throughout learning, suffer from a number of short-comings. Current research is demonstrating the use of dynamic topologies in overcoming some of these problems. The Location-Independent Transformation (LIT) is a general strategy for implementing variations of ANNs that use dynamic topologies during learning.

This paper formally defined LITs with associated execution and learning algorithms for the competitive learning and counterpropagation networks. Analysis gives overall complexities for both execution and learning for both CL and CPN as $O(n+logm)$.

LIT is not typically concerned with extending static models to handle dynamic topologies—only with supporting static and dynamic models, assuming they exist in the literature. The applicability and stability of such models depends on the original models themselves, not on the LIT transfromation. In this paper, however, the authors give a simple dynamic extension to CL and CPN, without analysis or discussion. This extension is used to illustrate how LIT can support both static and dynamic versions of models, as proof-of-concept rather than proposing a new extension.

Current work includes the following:

- LITs for other networks
- VLSI design and fabrication of LIT models.

CHAPTER 3

An Efficient Transformation for Implementing Two-layer Feedforward Neural Networks

Accepted by the Journal of Artificial Neural Networks © 1995

Once ideas for localist ANNs had been developed, the next logical step was to consider ANNs with distributed internal representations. *Distributed* (as contrasted with localist) means that responsibility for determining the output of the network is distributed or shared among many nodes.

The example used here is the standard Backpropagation model, which is the most well-known and widely used ANN model. This paper gives the transformation for 2-layer Backpropagation networks, which supports a large number of ANN applications. Chapter 4 extends this transformation to a network with an arbitrary number of layers. The LIT given in this paper is of particular interest, because it indicates that distributed networks can be represented effectively and efficiently in localist fashion. This paper also gives one potential dynamic extension to Backpropagation based on that proposed in [Odr93].

A conference version of this paper also appears in the Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms, 1995 [Rud95f].

I. INTRODUCTION

Hardware support for ANNs is important for handling large, complex problems in real time. Learning times can exceed tolerable limits for applications of large size and complexity using conventional computing schemes. Furthermore, hardware is becoming cheaper and easier to design. The *Location-Independent Transformation* (LIT) is a general implementation strategy for ANNs that overcomes several weaknesses of current hardware implementation methods.

Most ANNs use static topologies—the topology is fixed initially, and remains the same throughout learning. ANNs with static topologies often suffer from the following short-comings:

- sensitivity to user-supplied parameters: learning rate(s), etc.,
- local error minima during learning,
- there is no mechanism for deciding on an effective initial topology (number of nodes, number of layers, etc.).

Current research is demonstrating the use of dynamic topologies in overcoming these problems [Baf92], [Den93], [Fah88], [Fahlm], [Kar90], [Lee93], [Mar91b], [Mea91], [10-13], [Rei88]. A *dynamic topology* is one which allows adding and deleting both nodes and weighted connections during learning.

Early ANN hardware implementations are model-specific, and are intended to support static topologies [Far85], [Gra90], [Mea91]. More recent *neurocomputer* systems have specialized neural hardware, and seek to support more general classes of ANNs [Ham91], [Ram93], [Sham92]. Although some neurocomputers could potentially support dynamic topologies more directly in hardware, rather than in software, they currently do not. Of course, general parallel machines, like the Connection Machine [Hil85] and the CRAY [Alm89], can simulate the desired dynamics in software, but these machines are not optimized for neural computation. LIT supports general classes of ANNs *and* dynamic topologies in an efficient parallel hardware implementation.

LIT redesigns the original network into a hierarchical, parallel network of Location-Independent nodes (LI-nodes). A *LI-node* contains enough information locally to compute its function independent of any other node in the network. The set of LI-nodes is then embedded in a uniform tree topology. Figure 1 gives a general view of LIT. Execution (running the network) and learning are based on efficient broadcast/gather operations in the tree. Because the nodes are location-independent, they can be placed anywhere in the tree,

and dynamic topologies can be efficiently and easily supported: Adding nodes is as simple as allocating free nodes in the tree; deleting nodes is as simple as de-allocating the desired nodes.

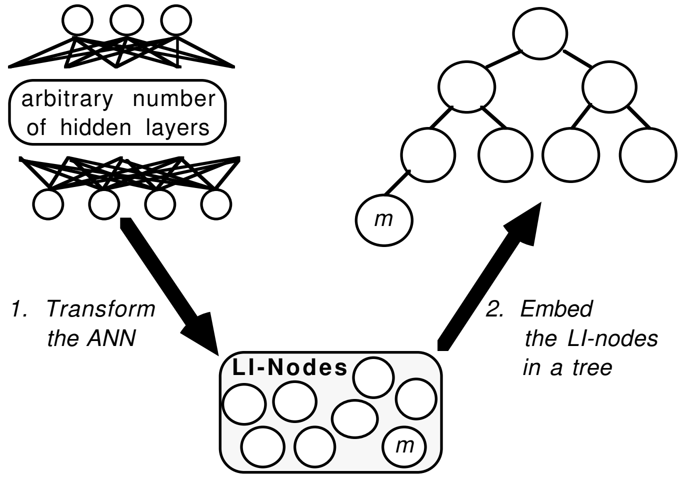


Figure 1. General LIT Transformation

LITs have been developed for counterpropagation and competitive learning [Rud95g], and Adaptive Self-Organizing Concurrent Systems' Adaptive Algorithm 2 [Mar91b], [Rud95d]. LITs can potentially support a broad set of ANNs, thus allowing one efficient implementation strategy to support dynamic variations of many ANNs.

This paper presents the transformation for backpropagation (BP) networks with a single hidden layer, though the same basic transformation applies to networks with an arbitrary number of layers [Rud95b]. Many dynamic extensions to BP have been proposed in the literature [Den93], [Fah88], [Fahlm], [Kar90], [Lee93], [Odr93], [Spe93]. The dynamic BP algorithm presented in this paper borrows ideas from [Odr93]. However, the reader should not assume that this is the only model supported by LIT—rather it is an illustrative example of the LIT strategy. Section II presents a general overview of LIT; section III demonstrates the transformation of 2-layer BP models, and gives the "transformed" versions of algorithms for the standard BP model [Rum86]; section IV presents an overview of the dynamic BP model of [Odr93], and extends the learning algorithm of section III accordingly; section V gives a complexity analysis of the algorithms in section III and IV; section VI is the conclusion.

II. THE TRANSFORMED NETWORK

A transformation redesigns the network so that each node contains enough information locally to compute its part of the network output, independent of any other node. A network whose nodes have this property is said to be *location-independent.* The nodes also are location-independent—regardless of the physical location of any node in the network, the relative order in which they compute results, or the order in which those results are gathered, *the individual computations are the same, and the network output is the same*. Furthermore, because a node's information is local, adding or deleting nodes from the network can be done without affecting any other nodes. Thus, location-independence allows efficient support for dynamic topologies. Intuitively, the transformation is applied to the intial topology of the original network, before any learning has taken place; however, it can be viewed as being applied to some "snapshot" of the network, at any time, and the process is still the same.

In this paper, the term *Control Unit* refers to a mechanism that broadcasts inputs to a network, and gathers results from it. The term *original* refers to a network before it is transformed, and the term *transformed* refers to a network after it has been transformed.

These terms apply similarly to the nodes as well. The number of layers refers to the number of *weight layers*.

LIT is a two-step process:

1. Construct a set of location-independent nodes based on the original model.
2. Embed the nodes in a tree.

This process is outlined in figure 2. Based on an original ANN model (left), a set of LI-nodes is constructed (middle), and the nodes are embedded in a tree (right).

LIT is not simply a reorganization of the original network, but rather involves redesigning the basic structure. The 4x5x3 original network in figure 2 is transformed to a network with five 4x3 nodes in a tree. A construction typically also involves reformulating the network equations, in order to more particularly describe the behavior of the transformed network. Although the original and transformed networks compute equivalent functions, correspondences between the two are not always direct, nor obvious. Details of a construction vary across different ANNs.

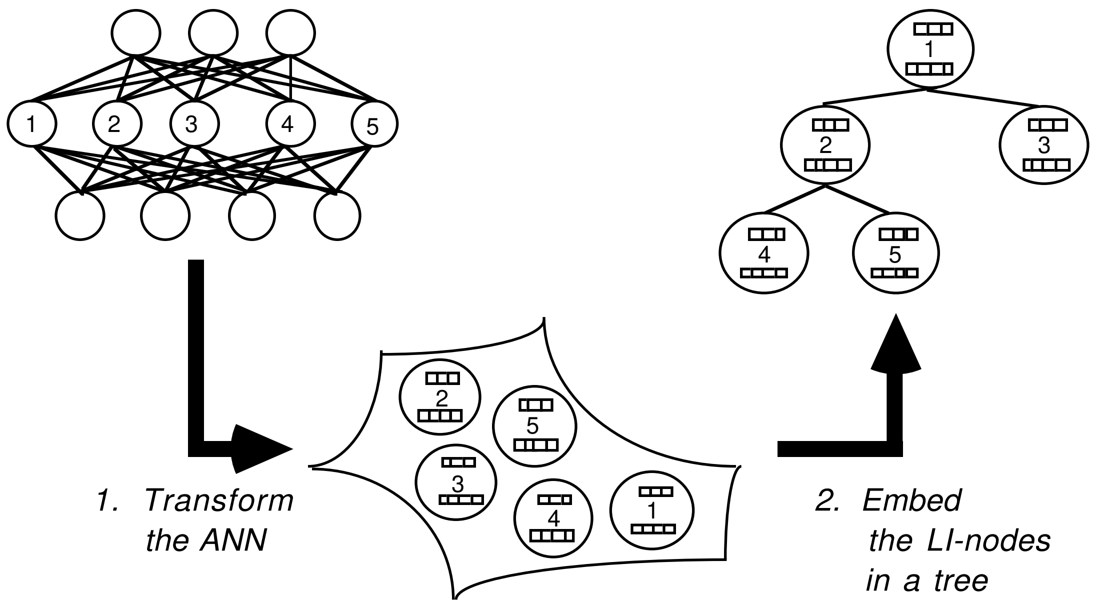


Figure 2. An Example Location-Independent Transformation

A communication topology can be chosen as a matter of speed, cost and reliability. The same topology can be used with many models. A binary tree topology is specified in the second step in figures 1 and 2. It is a simple, fast, regular, hierarchical topology that naturally supports broadcast and gather operations. It provides a basis for applying the transformation and explaining the associated execution and learning algorithms, without obscuring the details with a complex interconnect. However, a hypercube, or some other more fault-tolerant topology, could also be used. Hierarchical topologies allow adding or deleting nodes with time logarithmic in the number of nodes in the network, while limiting connectivity among nodes. The connectivity of a binary tree does not explode because each node is connected to one parent and zero to two children. The algorithms for the transformed network show explicitly how the required communication can take place.

An example illustrates the support for dynamic topologies. When a node is added to (deleted from) an original network, both the node and its connections to other nodes are added (deleted). If a hidden node 6 were added to the original network in figure 2, four connections to the input and three connections to the output nodes would also be added. This is accomplished in the transformed network by allocating a free node, such as the left child of node 3, and initializing it as a 4x3 node with the corresponding weights. The deletion of a hidden node in the original network, such as node 5, is accomplished in the transformed network by marking the corresponding node as "free". No other nodes are affected.

III. TRANSFORMATION OF A 2-LAYER BP NETWORK

It is assumed that the reader is familiar with the standard BP model [Rum86], however some necessary definitions are given here. Following that is a discussion of the structure and operation of a transformed node. In this description, variables are introduced, used and defined informally. Formal definitions of variables are given in sections III.C and III.D, which give the execution and learning algorithms, respectively.

A. BP Definitions

In a BP network, there are typically three groups of nodes—input nodes, output nodes, and hidden nodes (see figure 3). In the standard BP model, the input nodes act as place-holders only, and do not perform any function on the values they receive (some BP models may use the nodes for preprocessing such as normalization, but this is not part of the basic model). A network may have an arbitrary number of layers of weights, but 2-layer and 3-layer networks (counting by the number of weight layers) are the most common, and in this paper, only the LIT for 2-layer networks is presented. The LIT for networks with more than two layers is presented in [Rud95b]. Here, and throughout the rest of this paper, given a node j, all the nodes in the same node layer as j are referred to collectively as the *current layer*. The nodes in the node layer labeled with k are referred to as the *subsequent layer*. The nodes in the node layer labeled with i are referred to as the *previous layer*. This terminology is straightforward in execution mode or in the feedforward phase of learning. Note that the names remain the same during the backpropagation phase of learning. From the point of view of a hidden node j, the weights on its inputs are referred to collectively as $\boldsymbol{u_j}$, and the weights on its output as $\boldsymbol{w_j}$.

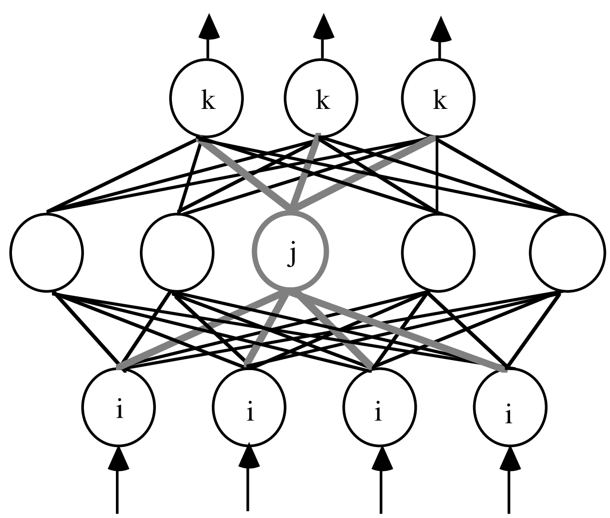


Figure 3. Use of the BP equations for an arbitrary hidden node *j*.

B. Transformation

Figure 4 shows the transformation of the 4x5x3 network into a 6-node location-independent network (from figure 2). There are five nodes in the new network, one node in the transformed network for each node in the original hidden layer, plus one bias node. Figure 5 shows a node in the transformed network (*transformed node*) in more detail—the left side shows a node operating during execution mode and the forward phase of learning, while the right side shows the backpropagation phase of learning.

The structure and operation of a transformed node (except the bias node) and transformed network is best introduced by informal comparison with an original hidden node, node 2 in figure 4, for example. The original node 2 has four weights from the input layer, and three weights to the output layer. A transformed BP node therefore stores two vectors (layers) of weights—one on inputs (hereafter $\boldsymbol{u_j}$) and one on outputs (hereafter $\boldsymbol{w_j}$). The transformed node 2 has four weights in $\boldsymbol{u_j}$ and three weights in $\boldsymbol{w_j}$. Each weight in the transformed node, u_{ij} or w_{jk} , has the value of the corresponding original weight. Each transformed node also stores a bias value θ_j, which corresponds to the bias value in the respective original hidden node. For purposes of execution and learning, θ_j can be treated as a component of $\boldsymbol{u_j}$.

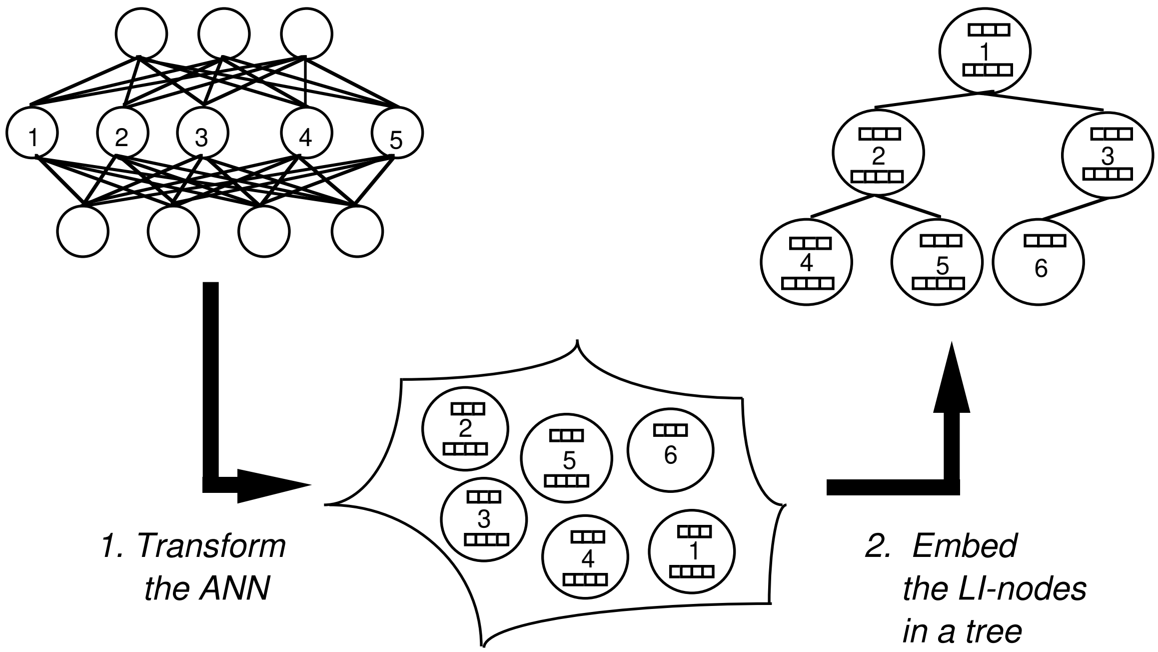


Figure 4. Transforming a 2-layer BP Network

The bias node is like the other transformed nodes, with some exceptions. It needs no $\boldsymbol{u_j}$, because its activation is always 1. The elements of its $\boldsymbol{w_j}$ each correspond to one of the biases from the nodes in the original output layer. It has no additional bias values of its own. In all other respects, the bias node behaves like the other transformed nodes.

B.1. Execution Mode in the Transformed Network

The network executes as follows: The Control Unit sends the inputs to the root node, which passes those values onto each of its children, and they to theirs, and so on. Each node (except the bias node) receives the broadcast, multiplies each input by the corresponding weight on the input, and sums the results together with each node's bias. The result of this summation is the same as the original net_j. $f(net_j)$, the sigmoid of each net_j, is then computed at each node. This corresponds to the scalar O_j for a hidden node j in the original network. In the bias node, $f(net_j)$ is not computed, but is always 1. The weights on output are each multiplied by $f(net_j)$, to produce an output *vector* which is the new $\boldsymbol{O_j}$. $\boldsymbol{O_2}$, the output vector for the transformed node 2, corresponds to the weights out of the original node 2 each multiplied by the scalar O_2. The vector summation of the transformed nodes' output vectors is accomplished by a gather operation. Node 2 receives $\boldsymbol{O_4}$ and $\boldsymbol{O_5}$ from its children, does the vector sum of these with $\boldsymbol{O_2}$, and sends the result to the root node, for example. The components of the vector $\boldsymbol{Y}$ (the vector sum of all $\boldsymbol{O_j}$

including the $\boldsymbol{O_j}$ of the bias node) at the root node correspond to net_j in each of the output nodes in the original network.

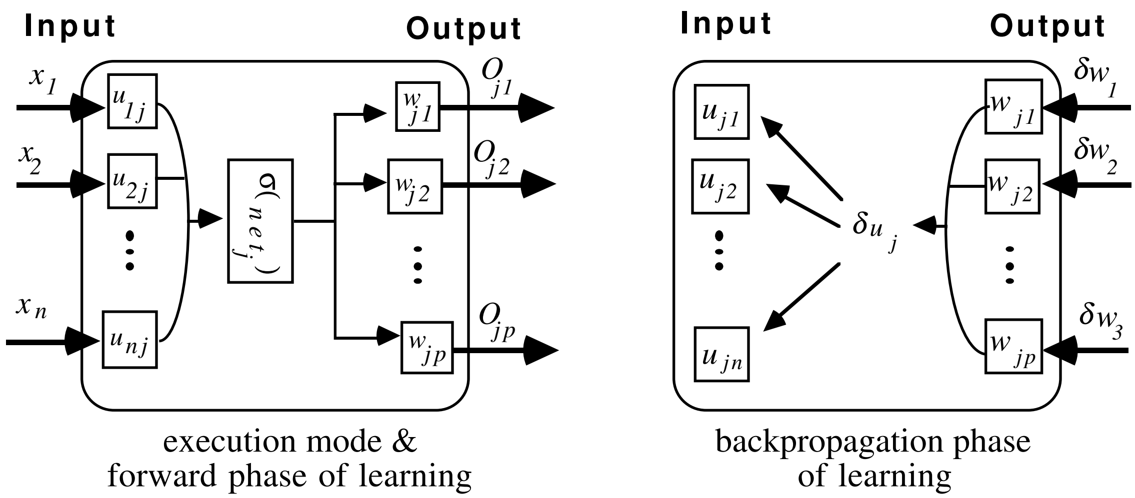


Figure 5. Two Views of a Node in the Transformed Network. The left shows a node operating in execution mode and/or the forward phase of learning. The right shows a node during the backpropagation phase of learning.

At this point $\boldsymbol{Y}$ resides at the root node. Now however, the sigmoid of each component of $\boldsymbol{Y}$ needs to be computed, in order to produce values corresponding to the output values of the original output nodes. This vector of output values is called $\boldsymbol{Z}$. $\boldsymbol{Z}$ can be computed in parallel fashion in two ways:

1. by a pipelined computation executed as the root node sends $\boldsymbol{Y}$ to the Control Unit, or
2. by distributing (through rebroadcasting) $\boldsymbol{Y}$ to the nodes in the network, where each node computes z_i from a component y_i, and then gathering $\boldsymbol{Z}$.

There are two main factors involved in deciding how to compute $\boldsymbol{Z}$ from $\boldsymbol{Y}$: computation time and communication time. If computation time is fast relative to communication time, then the first method is more efficient. If, on the other hand, communication is fast relative to computation, the second method is more efficient. For BP, computation of the sigmoid is very fast compared to the time needed to send values to the Control Unit, or to broadcast back to the network—thus, for the BP model, the method using the pipe is preferable.

Using the piped scheme then, $\boldsymbol{Z}$ is computed as the root node sends $\boldsymbol{Y}$ to the Control Unit, at which point the Control Unit has the network output. For execution, this pipe has only one computation—the sigmoid—but the pipe for learning (below) has more steps.

B.2. Learning Mode in the Transformed Network (Standard BP)

As in the original model, learning iterates over a series of patterns, until convergence is reached. As each pattern is presented for learning, the network proceeds through a forward phase and then a backpropagation phase, as follows: First, present the training pattern to the network as above for execution, up to the point at which $\boldsymbol{Y}$ resides at the root node. This is the forward phase. Second, the backpropagation phase begins.

The first step of the backpropagation phase is to compute an error value (δ) for each of the output values, and from that, to compute *psse*, the single value representing the pattern sum-squared error for the current pattern. This first step is accomplished by a pipe like the one at the end of execution, but with some additional computations. First, $\boldsymbol{Z}$ is computed from $\boldsymbol{Y}$ as above. Second, $\boldsymbol{Z}$ and $\boldsymbol{T}$ are used to compute the error vector $\delta\boldsymbol{w}$, which will be sent back to the nodes. (Target outputs $\boldsymbol{T}$ are at the Control Unit, and therefore are directly available to the pipe.) Third, δw_k (each component of $\delta\boldsymbol{w}$) is squared to produce δw_k^2 (for calculating sum-squared pattern error). Fourth, each δw_k^2 value is summed into *psse* at the "end" of the pipeline (*psse* is used only to check for convergence during learning.) Each δw_k in $\delta\boldsymbol{w}$ corresponds to the δ_k computed by the respective output node in the original network.

The second step of the backpropagation phase propagates the error backward (see the right half of figure 5). The vector $\delta\boldsymbol{w}$ is broadcast to the nodes. This value is then used to compute the required weight change on each weight (the other information already exists at the node). Also at each node, like node 2, each w_{jk} is multiplied by the corresponding component of $\delta\boldsymbol{w}$, and the results are summed. This new value is used to create a δ value, $\delta\boldsymbol{u_j}$, with which to calculate changes for each u_{ij} (except in the bias node, which has no $\boldsymbol{u_j}$). Once the appropriate computations are completed, all weights can be updated, and the backward phase ends. The network is ready to move on to the next pattern.

There are a variety of potential location-independent sets of nodes that could be constructed. The construction given here results in a node structure that is very similar to the original model. An analysis of the original equations and structures reveals a symmetry both in the information used for computation, and in the behavior during forward and backpropagation phases. The LIT structure reflects that symmetry, as well as achieving the other goals of LIT.

C. Execution Algorithm

The following equations describe execution mode for the 2-layer LIT BP. They are essentially the original equations, but altered to show the behavior of the LIT model.

<u>Definitions</u>

q	number of weight layers in the original network. $q = 2$.
n	number of nodes in the original input layer, size of $\boldsymbol{u_j}$.
m	number of nodes in the original hidden layer, number of nodes in the transformed network.
p	number of nodes in the original output layer, size of $\boldsymbol{w_j}$.
a	real-valued number, it will be either net_j or y_k.
$\boldsymbol{O_j}$	output vector of new node j.
O_j	intermediate activation value for new node j, it corresponds to O_j for an original node j.
net_j	real-valued sum of the inputs to node j [equation (4)].
f	sigmoid function.
$\boldsymbol{X}$	pattern clamped on inputs, has size n.
x_i	ith real-valued component of $\boldsymbol{X}$.
$\boldsymbol{u_j}$	vector of weights on inputs in a new node j, has size n.
u_{ij}	real-valued ith component of $\boldsymbol{u_j}$. It corresponds in the original network to the weight from input node i to hidden node j.
o_{jk}	real-valued kth component of $\boldsymbol{O_j}$. It corresponds in the original network to the weight from hidden node j to output node k, multiplied by the original O_j.
$\boldsymbol{w_j}$	vector of weights on output in a new node j, has size p.
w_{jk}	real-valued kth component of $\boldsymbol{w_j}$. It corresponds in the original network to the weight from hidden node j to output node k.
$\boldsymbol{Y}$	vector of intermediate activation values, the sum of all $\boldsymbol{O_j}$.

y_k real-valued kth component of $\boldsymbol{Y}$. It corresponds in the original network to the net_j of output node k.
Z vector of network output values.
z_k real-valued (always between 0 and 1) kth component of $\mathbf{Z}$. It corresponds in the original network to the output O_k of output node k.

Equations

$$f(a) = \frac{1}{1+e^{-a}} \tag{1}$$
$$O_j = f(net_j) \tag{2}$$
$$net_j = \sum_{i=1}^{n} x_i u_{ij} + \theta_j \tag{3}$$
$$o_{jk} = O_j w_{jk} \tag{4}$$
$$y_k = \sum_{j=1}^{m} o_{jk} \tag{5}$$
$$z_k = f(y_k) \tag{6}$$

The main steps of the execution algorithm (italics), numbered for convenient reference, describe the basic BP algorithm. The sub-steps (1.1, 2.1, etc.) specify the details of how the LIT network accomplishes these steps. The algorithm is as follows:

(1) Clamp an input pattern on the input nodes.
 1.1 Broadcast input vector **X** to the nodes.
(2) Each node takes the sigmoid of the sum-of-products of its weights and inputs.
 2.1 Each node computes its $\mathrm{net_j}$, according to equation (4), and then computes $\mathrm{O_j}$ according to equation (3). $\mathrm{O_j}$ for the bias node is 1.
 2.2 Each node's weight vector $\mathbf{w_j}$ is multiplied by its (scalar) $\mathrm{O_j}$ to produce its output vector $\mathbf{O_j}$ [equation (5)].
 2.3 Each node computes the vector sum of its $\mathbf{O_j}$ with the result from each child, and the sum (a vector) is sent to its parent. (The Control Unit is the parent of the root node.). This implements equation (6). The result is the vector **Y**.
(3) Wait until the activation values on the output nodes are valid.
 3.1 **Y** is sent (through the sigmoid pipe) to the Control Unit. The result **Z** is the output of the network.

D. Learning Algorithm (Standard BP)

Below are additional equations that describe the learning mode for the LIT BP. The forward phase of learning is identical to execution mode, and so follows the equations given above.

Additional Definitions

α weight change momentum constant.
$\boldsymbol{\delta w}$ vector of error values for all $\boldsymbol{w_j}$. It has size p.
δw_k real-valued jth component of $\boldsymbol{\delta w}$. It corresponds in the original network to δ_k for output node k.
δu_j real-valued error value for $\boldsymbol{u_j}$ in node j. It corresponds in the original network to δ_j for hidden node j.

T vector of target values for the current pattern being presented. It has size p.

t_k target value for z_k. It corresponds in the original network to the target value for O_k for output node k.

psse pattern sum-squared error for the current pattern being presented.

tsse total sum-squared error over all patterns in the current learning epoch.

ecrit maximum error tolerance for learning.

s number of patterns in the training set (epoch).

r simple index, used to avoid confusion with i, j, k, etc.

c current training cycle number.

η a real-valued learning constant.

Equations

$$f'(a) = \frac{\partial f(a)}{\partial a} = f(a)(1-f(a)) \tag{7}$$

$$\Delta w_{jk}(c) = \eta F_j \delta w_k + \alpha \Delta w_{jk}(c-1)$$

$$\Delta u_{ij}(c) = \eta x_i \delta u_j + \alpha \Delta u_{ij}(c-1) \tag{8}$$

$$\delta w_k = (t_k - z_k) f'(y_k) \tag{9}$$

$$\delta u_j = \left(\sum_{k=1}^{p} (\delta w_k w_{jk})\right) f'(O_j) \tag{10}$$

$$tsse = \sum_{r=1}^{s} psse_r \tag{11}$$

$$psse_r = \sum_{k=1}^{p} \delta w_k^2 \tag{12}$$

As with the execution algorithm above, the steps of the learning algorithm are numbered for convenient reference. The sub-steps give the details of how LIT accomplishes each step. The learning algorithm is as follows:

Until Convergence (tsse < ecrit at the end of an epoch)

(1) Present a training pattern to the network

1.1 Forward phase of learning—this is same as execution above, up to (and including) step 2.3.

(2) Calculate the error δ of the output units (T - O)

2.1 As each y_k is sent to the Control Unit, the following occurs:

2.1.1 z_k is computed [equation (6)], and received by the Control Unit.

2.1.2 z_k and t_k are used to compute δw_k [equation (9)], which is received by the Control Unit.

2.1.3 δw_k is used to compute δw_k^2.

2.1.4 Each δw_k^2 is summed with the value of *psse* so far [equation (12)].

(3) For each hidden layer Calculate δ_j using δ_k from the subsequent layer

3.1 The vector $\boldsymbol{\delta w}$ is broadcast to the nodes.

3.2 δu_j is calculated according to equation (10).

(4) Update the weights

4.1 Calculate Δw_{jk} and Δu_{ij} according to equation (8).

4.2 Change the weights.

(5) Update tsse according to equation (11)

end

Step 5 may actually be computed any time after the end of step 2—independent of steps 3 or 4. Most algorithms assume that *psse* and *tsse* can be maintained, but give no explicit mechanism for doing so—LIT provides a mechanism.

IV. THE DYNAMIC BP MODEL

This section gives an overview the dynamic BP (hereafter DBP) model proposed by Odri, et al [Odr93], then formally extends the learning algoirthm of section III to support these dynamics. The DBP model of [Odr93] is of interest because it allows the addition and deletion of both nodes and individual weights dynamically during learning.

Some dynamic BP models use pruning as a basic technique, while some both grow and prune structures during learning. Some use global state information, and change the topology only after learning is complete, while some use partial, localized information and attempt to change the topology incrementally as learning proceeds. The algorithm of [Odr93] can be outlined as follows. Details are discussed after the outline. The steps are numbered to correspond with the algorithm of section III, for convenience.

Until Convergence ($tsse < ecrit$)

1. Present a pattern to the network.

2-5. Perform standard BP learning, including adjusting the weights.

6. Perform probabilistic weight deletion.
   - 6.1 Each weight computes a probability of self-deletion in the range 0-1.
   - 6.2 Each weight generates a random value in the range 0-1.
   - 6.3 If the probability is greater than the random value, then the weight is deleted.
7. Perform node deletion on the hidden nodes (the number of input and output nodes remains fixed).
   - 7.1 If $\boldsymbol{u_j=0}$, or $\boldsymbol{w_j=0}$, then node j self-deletes.
   - 7.2 In the case that $\boldsymbol{u_j=0}$, and the bias $\theta_j \neq 0$, then adjust the biases of the remaining nodes to compensate.
8. Check for node addition (by probabilistic node division).
   - 8.1 Each node computes a probability of dividing, in the range 0-1.
   - 8.2 Each weight generates a random value in the range 0-1.
   - 8.3 If the probability is less than the random value, then the node does not divide—skip step 9.
9. Do node division (only if at least one node divides)
   - 9.1 Two descendant nodes are created from the node: The $\boldsymbol{u_j}$ for both remain the same as the parent, but $\boldsymbol{w_j}$ are mutated in order to guarantee that the nodes are unique.

end

Conceptually, step 9 could be done at the end of step 8. However, for algorithmic purposes, it is more convenient to check for the possibility of node division separate from actually performing node division.

Three ideas need to be discussed in connection with this algorithm:

- Node Addition and Deletion,
- Weight Addition and Deletion,
- Termination of Topological Dynamics.

The following definitions are used in the rest of section IV:

μ a real-valued weight mutation constant.

v_{ij} random real value generated by (for) each w_{ij} in checking for weight deletion (v_{jk} for w_{jk}). It has range 0-1.

ν_j random real value generated by each node j in checking for node division. It has range 0-1.

P^d_{ij} probability that w_{ij} will self-delete (P^d_{jk} for w_{jk}).

P^g_j probability that node j will divide.

k_g number of cycles since last node division (by any node).

k_d number of cycles since last weight was deleted (in any node).

$\beta_j(c)$ average cumulative change in cell output error at time c, ranging from 0-1.

A. Node Addition and Deletion

Nodes are added through a process of probabilistic node division. As learning proceeds, from some time 0 to time c, various δ values are computed by each node. At each node j, these values are used to track and compute $\beta_j(c)$, the *average cumulative change in node output error* at time c. $\beta_j(c)$ is directly proportional to the probability that a node will divide at time c during learning, according to the equation

$$P^g_j(c) = P^g_0\,\varphi(k_g)\beta_j(c). \tag{13}$$

P^g_0 is the base probability of division (user-specified). k_g is a counter (same value at every node) that counts the number of cycles (each pattern presentation is a cycle) since the last time any node divided. $\varphi(k_g)$ is a "recovery control" function, in the range 0-1. It is near zero immediately following a division, and near 1 at some user-specified number of cycles after the last division, increasing asymptotically.

When a node divides, two "descendant" nodes are created, and the "parent" deletes itself from the network. The $\boldsymbol{u_j}$ for both remain the same as the parent, but $\boldsymbol{w_j}$ are mutated according to the equations

$$w^{d1}_{jk} = (1+\mu)\,w^p_{jk}, \text{ and}$$

$$w^{d2}_{jk} = -\mu\,w^p_{jk}. \tag{14}$$

d1 and *d2* denote the new weight values in the children, and *p* denotes the original value in the parent. The bias values are the same as the parent's. These initial values guarantee that the division does not perturb the network, as the net effect of the descendants is the same as the original parent at the time of division. However, as learning proceeds, the two descendants will diverge toward different "hyperplanes" across which the parent was oscillating. This increases the overall stability of the network.

Nodes delete themselves in two cases:

1. When $\boldsymbol{u_i}$ is **0**, or
2. when $\boldsymbol{w_j}$ is **0**.

In case 1, the threshold or bias value may not be 0, even if all of the other weights are 0. [Odr93] gives a simple transformation which adjusts the biases of nodes in the subsequent layer to compensate for deleting a node with a non-zero bias

$$\theta'_k = \theta_k + w_{jk} f(\theta_j). \tag{15}$$

j indicates the node that is deleted, and *k* indicates the nodes in the subsequent layer.

Even though the criteria for deleting a node is deterministic, the deletion of a node is a probabilistic event. It depends on the deletion of weights, which individually delete in probabilistic fashion (see below).

B. Weight Addition and Deletion

New weights are added to the network automatically when a new node is created. Just as the output error values and changes can be tracked over time, the contribution of a particular weight to a node's overall activity can be tracked over time. The authors call this value F_{ij} at some time c the *factor of action* between node i and node j. This factor is computed using the the previous factor value, the current weight value, and the values of outputs from the previous layer at the current time and at one time step earlier. This factor, and its normalized equivalent $\hat{F}_{ij}$ are then used to compute the probability that a weight will self-delete:

$$P_{ij}^{d}(c) = \begin{cases} P_0^d \varphi(k_d)(1 - \frac{1}{1-e^{-\gamma}}[1 - e^{-\nu} F_{ij}(c)]) & if\ \hat{F}_{ij}(c) \le F_0 \\ 0, & if\ \hat{F}_{ij}(c) > F_0 \end{cases} \quad . \qquad (16)$$

F_0 is the "activity threshold" for a weight. Any weight whose $\hat{F}_{ij}$ remains above this threshold will never self-delete. Over time, this means that only weights that contribute significantly to network activity remain, whereas all others will be forced to 0. (A 0 weight is the equivalent of being deleted.) P_0^d is the base probability that a weight will self-delete. k_d is the number of cycles since the last weight was deleted, analogous to k_g for the nodes. γ is a sensitivity parameter, and the authors do not explain the role of v, though clearly it is analogous to γ.

C. Termination of Topological Dynamics

The division of nodes is linked to the amount of error remaining in the network. As the error decreases over learning, the probability that a node (and thus any node) will divide will also decrease (and can reach zero). This effectively halts the addition of new nodes, and by implication, weights as well.

As mentioned above, weights are not continually deleted from the network. Sufficiently active weights remain in the network, while others are deleted. Once the addition of new nodes ceases, a point will be reached at which the only remaining weights in the network are all above the activity threshold, and will never be deleted. This also halts the deletion of nodes, for reasons explained in section IV.B.

D. Dynamic Extensions to the Learning algorithm of III.D.

(6) Check for self-deletion of weights

- 6.1 Every weight calculates the probability of self-deleting P_{ij}^d for u_{ij}, or P_{jk}^d for w_{jk}, respectively, according to equation (16).
- 6.2 Each weight generates a random value between 0 and 1, ν_{ij} or ν_{jk}, respectively.
- 6.3 Any weight u_{ij} for which $P_{ij}^d \ge \nu_{ij}$ self-deletes. Any weight w_{jk}, for which $P_{jk}^d \ge \nu_{jk}$ self-deletes.
- 6.4 If a node has at least one weight self-delete, it sets a deletion flag to 1; otherwise the flag is set to 0.
- 6.5 Each leaf node sends its deletion flag value to its parent; each parent ***OR***s its deletion flag with those received from its children, and sends the result to its parent.
- 6.6 If the Control Unit receives a 1, then k_d is reset to 0; otherwise k_d is incremented by 1.

(7) Check for self-deletion of nodes

7.1 Any node j for which $\boldsymbol{u_j=0}$ or $\boldsymbol{w_j=0}$ self-deletes.
7.2 If a node self-deletes because $\boldsymbol{u_j=0}$, it sets a deletion flag to 1; otherwise the flag is set to 0.
7.3 Adjustment values to compensate for deleted nodes with non-zero biases are computed according to equation (15). (Only nodes that are currently self-deleting compute non-zero results.)
7.4 Each leaf node sends its deletion flag and adjustment values to its parent. Each parent ***OR***s its deletion flag with those received from children; each parent sums its adjustment values with those received from its children; the resulting values are sent up to its parent.
7.5 If the Control Unit receives a deletion flag of 0, ignore adjustment values and go to step 8.
7.6 The Control Unit broadcasts the adjustment values to the nodes.
7.7 Each node adjusts its bias according to the equation (15).

(8) Check for node division

8.1 Each node j calculates the probability of division P_j^g, according to equation (13).
8.2 Each node generates a random value v_j between 0 and 1.
8.3 Any node j for which $P_j^g \geq v_j$, sets its division flag to 1; otherwise the flag is set to 0.
8.4 Each leaf node sends its division flag to its parent; the parent ***OR***s its division flag with those received from its children; the result is sent to its parent.
8.5 If the Control Unit receives a division flag of 0:
- increment division counter k_g by 1, and
- skip step 9.

(9) If the Control Unit receives a division flag of 1 (i.e., at least one node divides):

9.1 Reset the division counter k_g to 0.
9.2 Broadcast permission to divide to the nodes.
9.3 Each node which previously set its division flag generates two descendants, according to equation (14).
9.4 The parent node is initialized to match one of the children nodes (thus deleting the parent); the information for the other child is sent to the nearest free child in the subtree of the parent.

Step 9, node division, is *potentially* the most expensive of all steps in terms of complexity and performance. However, it is important to realize that node division is a *rare* event in the operation of the network; when it does occur, it is highly unlikely to tax the resources of the network—particularly large networks. First, only highly unstable nodes have high probabilities of dividing. Second, as training time increases and the network becomes more stable, it is less likely for any particular node to become unstable, and correspondingly more remote that even a few nodes will simultaneously divide.

V. COMPLEXITY AND PERFORMANCE

The complexity of the algorithms in section IV can be characterized by pipelined broadcast and gather in a tree of nodes. The width of a broadcast block is assumed to be the size of a single variable. The tree has $m+1$ nodes, one for each hidden node in the original network, plus a bias node, and the nodes operate in parallel. Assume that each node has n inputs and p outputs. Given these assumptions, analysis gives an overall complexity of $O(n+p+logm)$ for both execution and learning, for a single pattern.

The complexity of the steps of the execution algorithm is summarized, followed by elaboration where necessary. The numbers correspond to the steps from the algorithms in section III. The complexity of the learning algorithm is presented in the same format, including the dynamic extensions.

Execution Algorithm		
1.	Broadcast	$O(n+logm)$
2-2.2.	Node Computation	$O(n)$
2.3-3.	Gather	$O(p+logm)$
total		$\boldsymbol{O(n+p+logm)}$

Broadcast or gather for a single variable is clearly $O(logm)$. For n variables, one might expect a complexity of $O(nlogm)$, but the pipelined tree reduces this to $O(n+logm)$, since different levels of the tree operate on different variables simultaneously. The node computation is an $O(n)$ operation for n variables, but can be overlapped with broadcast. The sigmoid pipeline at the end of the gather is $O(p)$, one for each output variable, but this is overlapped with the gather of variables up the tree.

Learning Algorithm		
1.	Forward Phase	$O(n+p+logm)$
2.	Compute $\delta \boldsymbol{w}$	$O(p)$
3.1	Broadcast $\delta \boldsymbol{w}$	$O(p+logm)$
3.2	Calculate δu_j	$O(p)$
4.	Update weights	$O(n+p)$
5.	Partial sum *tsse*	constant
6.1-6.4	Weight deletion	$O(n+p)$
6.5-6.6	Gather deletion flags	$O(logm)$
7.1-7.2	Node deletion check	$O(n+p)$
7.3	Compute Adjustments	$O(p)$
7.4-7.5	Gather Adjustments	$O(p+logm)$
7.6	Broadcast Adjustments	$O(p+logm)$
7.7	Adjust biases	$O(p)$
8.1-8.3	Node division check	$O(n)$
8.4-8.5	Gather division flags	$O(logm)$
9.1-9.2	Broadcast divide flag	$O(logm)$
9.3-9.4	Allocate new node(s)	$O(n+p+logm)$
total		$\boldsymbol{O(n+p+logm)}$

Step 2 is $O(p)$ because there is one δ value for each output variable. Step 4 is $O(n+p)$ assuming that a node can update only one weight at a time. Steps 6.1-6.4 and 7.1-7.2 also assume that only one weight can be checked at a time. Step 7.3 is $O(p)$ because each node generates one value for each of its weights in $\boldsymbol{w_j}$. Steps 7.4-7.5 have the same character as the gather for execution, only the data is different. Step 7.7 is $O(p)$ because p summations are performed at each node. Steps 8.1-8.3 is $O(n)$ because the cumulative error values from each of the weights in $\boldsymbol{u_j}$ is used to compute $\beta_j(c)$, etc. Steps 9.3-9.4 are $O(n+p+logm)$ assuming that there is a free node in the subtree of the parent to hold each new node—n input variables and p output variables have to be sent to each new node. Probabilistically speaking, only one node will likely divide at any given time c. However, if more than one node does divide, and free nodes exist but are not guaranteed to be in the respective subtrees, then they could be added sequentially, one by one. In this case, Steps 9.3-9.4 are $O(d(n+p)logm)$, where d is the number of nodes that simultaneously divide.

The analysis in this section shows that the complexity of execution and learning for the algorithms of section V is $O(n+p+logm)$.

In concluding this section, it is interesting to speculate on the relationship between n, p, and m, and the depth of the network $logm$. Assume that any set of variables (input or output) can be represented as strings of binary digits. For present purposes, a single variable can be a binary digit. If the inputs to a node have length n, and the outputs have length p, then there are 2^n possible input "strings", and 2^p possible output strings. The maximum number of nodes needed to represent all of these is the cross product, namely $2^n 2^p$, or $2^{(n+p)}$. Thus, $m \leq 2^{(n+p)}$, and $logm \leq n+p$, where m is the actual number of nodes used. In practice, for an application, it is hoped that at least $m \leq 2^{(n+p-1)}$, and most often that $m \ll 2^{(n+p-1)}$, otherwise no significant generalization is taking place as the network learns. This does not mean that $logm \ll n+p-1$, only that $logm < n+p$, because the relationship between m and $logm$ is exponential.

The above might seem to suggest that the number of input/output variables dominates the performance. However, there are other factors to consider. Except when adding a new node to the network, the total size of any broadcast data is $O(n)$. (When adding a node, the broadcast size is $O(n+p)$.) The total size of any gather data is $O(p)$. Given that $n < n+p$, and $n < n+p$, is there a meaningful relationship between n, p, and $logm$ (or the depth of the network)? It is tempting to state that both n and $p < logm < n+p$, under the assumption that for applications of any size, $logm$ will be near $n+p$. It is easy to come up with examples, in terms of numbers, which make the left half of that relationship false, i.e. for which $n > logm$. Often, $p < n$, so that $p < logm < n < n+p$. In such cases, the number of input variables will dominate broadcast, while the depth of the network will dominate gather.

The previous two paragraphs indicate that there are clear upper bounds on complexity and performance, as presented here. Lower bounds, or bounds on actual performance, however, seem to be problem-specific.

VI. CONCLUSION

ANNs that use a static topology, i.e. a topology that remains fixed throughout learning, suffer from a number of short-comings. Current research is demonstrating the use of dynamic topologies in overcoming some of these problems. The Location-Independent Transformation (LIT) is a general strategy for implementing variations of ANNs that use dynamic topologies during learning.

This paper introduced ideas for LITs for static and dynamic feedforward, distributed neural networks. An LIT with associated execution and learning algorithms for backpropagation were formally defined, as an example. Analysis gives overall complexities for both execution and learning as $O(n+p+logm)$, where n is the number of inputs, p is the number of outputs, and m is the number of hidden nodes in the original network.

Current work includes the following:

- LITs for other ANNs, aside from those for which transformations have been developed.
- VLSI design and fabrication of LIT models.

CHAPTER 4

A Transformation for Implementing Distributed Multilayer Feedforward Neural Networks

submitted, 1995

This chapter extends the transformation for 2-layer distributed ANNs, given in the last chapter, to a network with an arbitrary number of layers. While some ANNs have only two layers (as in the network used in Chapter 3), most ANNs have more than two layers of weights. Hence, this is an important step toward making LIT general enough to support most ANNs, because an arbitrary number of layers can be supported.

The basic means of supporting arbitrary numbers of layers is the 2-layer transformation of Chapter 3, augmented with multiple broadcast and gather operations. A multilayer network is transformed into two or more sets of LIT nodes, as required by the depth of the original network. Broadcasts and gathers occur sequentially in waves *between the sets* of LIT nodes, where each node contributes to the network function only during its proper wave. Given this particular LIT transformation, the number of broadcast waves required is 1/2 the depth (longest path) of the original network. The transformation is reasonable because typical multilayer networks are not very deep.

1. Introduction

Hardware support for ANNs is important for handling large, complex problems in real time. Learning times can exceed tolerable limits for applications of large size and complexity using conventional computing schemes. Furthermore, hardware is becoming cheaper and easier to design. The *Location-Independent Transformation* (LIT) is a general implementation strategy for ANNs that overcomes several weaknesses of current hardware implementation methods.

Most ANNs use static topologies—the topology is fixed initially, and remains the same throughout learning. ANNs with static topologies often suffer from the following short-comings:

- sensitivity to user-supplied parameters: learning rate(s), etc.,
- local error minima during learning,
- there is no mechanism for deciding on an effective initial topology (number of nodes, number of layers, etc.).

Current research is demonstrating the use of dynamic topologies in overcoming these problems [Baf92], [Den93], [Fah88], [Fahlm], [Kar90], [Lee93], [Mar91b], [Mea91], [Rei88]. A *dynamic topology* is one which allows adding and deleting both nodes and weighted connections during learning.

Early ANN hardware implementations are model-specific, and are intended to support static topologies [Far85], [Gra90], [Mea91]. More recent *neurocomputer* systems have specialized neural hardware, and seek to support more general classes of ANNs [Ham91], [Ram93], [Sha92]. Although some neurocomputers could potentially support dynamic topologies more directly in hardware, rather than in software, they currently do not. Of course, general parallel machines, like the Connection Machine [Hil85] and the CRAY [Alm89], can simulate the desired dynamics in software, but these machines are not optimized for neural computation. LIT supports general classes of ANNs *and* dynamic topologies in an efficient parallel hardware implementation.

LIT redesigns the original network into a hierarchical, parallel network of Location-Independent nodes (LI-nodes). A *LI-node* contains enough information locally to compute its function independent of any other node in the network. The set of LI-nodes is then embedded in a uniform tree topology. Figure 1 gives a general view of LIT. Execution (running the network) and learning are based on efficient broadcast/gather operations in the tree. Because the nodes are location-independent, they can be placed anywhere in the tree,

and dynamic topologies can be efficiently and easily supported: Adding nodes is as simple as allocating free nodes in the tree; deleting nodes is as simple as de-allocating the desired nodes.

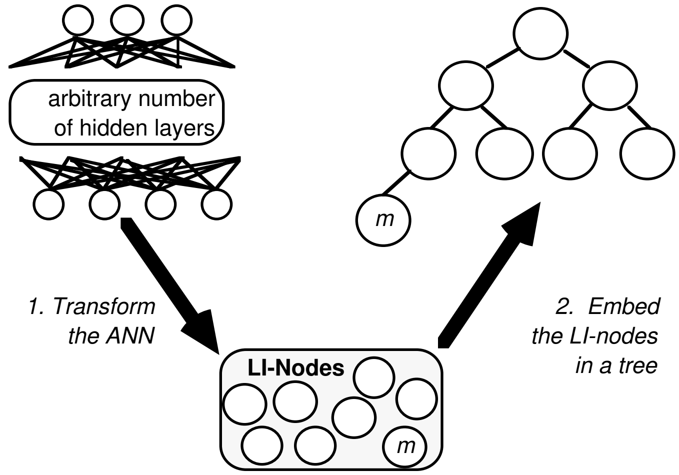


Figure 1. General LIT Transformation

LITs have been developed for the following models: counterpropagation and competitive learning [Rud95g]; Adaptive Self-Organizing Concurrent Systems' Adaptive Algorithm 2 [Mar91b], [Rud91b]; Backpropagation networks with two weight layers [Rud95a], [Rum86]. LITs can potentially support a broad set of ANNs, thus allowing one efficient implementation strategy to support dynamic variations of many ANNs.

This paper presents the transformation for multilayer backpropagation (BP) networks. This transformation extends the transformation for 2-layer networks given in [Rud95a] to handle an arbitrary number of layers. Section 2 presents a general overview of LIT; section 3 discusses the transformation of 2-layer BP models; section 4 discusses the transformation of BP networks with more than two weight layers; section 5 gives the algorithms for the transformed network; section 6 adds the steps to the learning algorithm that handle dynamic toplogies based on the model of [15]; section 7 gives a complexity analysis of the algorithms; section 8 is the conclusion.

Many dynamic extensions to BP have been proposed in the literature [Den93], [Fah88], [Fahlm], [Kar90], [Lee93], [Odr93], [Spe93]. The reader should not assume that the dynamic model of [Odr93] is the only one that is supported by LIT—rather, it is intended as an illustrative example of the LIT strategy. While dynamic topologies are important, the focus of this paper is the transformation and algorithms for multilayer networks—sections 4 and 5. More detail on the dynamic portion of the learning algorithm can be found in [Odr93], [Rud95a].

2. The Transformed Network

A transformation redesigns the network so that each node contains enough information locally to compute its part of the network output, independent of any other node. A network whose nodes have this property is said to be *location-independent.* The nodes also are location-independent—regardless of the physical location of any node in the network, the relative order in which they compute results, or the order in which those results are gathered, *the individual computations are the same, and the network output is the same*. Furthermore, because a node's information is local, adding or deleting nodes from the network can be done without affecting any other nodes. Thus, location-independence allows efficient support for dynamic topologies. Intuitively, the transformation is applied to the initial topology of the original network, before any learning has taken place; however, it can be viewed as being applied to some "snapshot" of the network, at any time, and the process is still the same.

In this paper, the term *Control Unit* refers to a mechanism that broadcasts inputs to a network, and gathers results from it. The term *original* refers to a network before it is transformed, and the term *transformed* refers to a network after it has been transformed. These terms apply similarly to the nodes as well. The number of layers refers to the number of *weight layers*.

LIT is a two-step process:

1. Construct a set of location-independent nodes based on the original model.
2. Embed the nodes in a tree.

This process is outlined in figure 2. Based on an original ANN model (left), a set of LI-nodes is constructed (middle), and the nodes are embedded in a tree (right).

LIT is not simply a reorganization of the original network, but rather involves redesigning the basic structure. The 4x5x3 original network in figure 2 is transformed to a network with five 4x3 nodes in a tree. A construction typically also involves reformulating the network equations, in order to more particularly describe the behavior of the transformed network. Although the original and transformed networks compute equivalent functions, correspondences between the two are not always direct, nor obvious. Details of a construction vary across different ANNs.

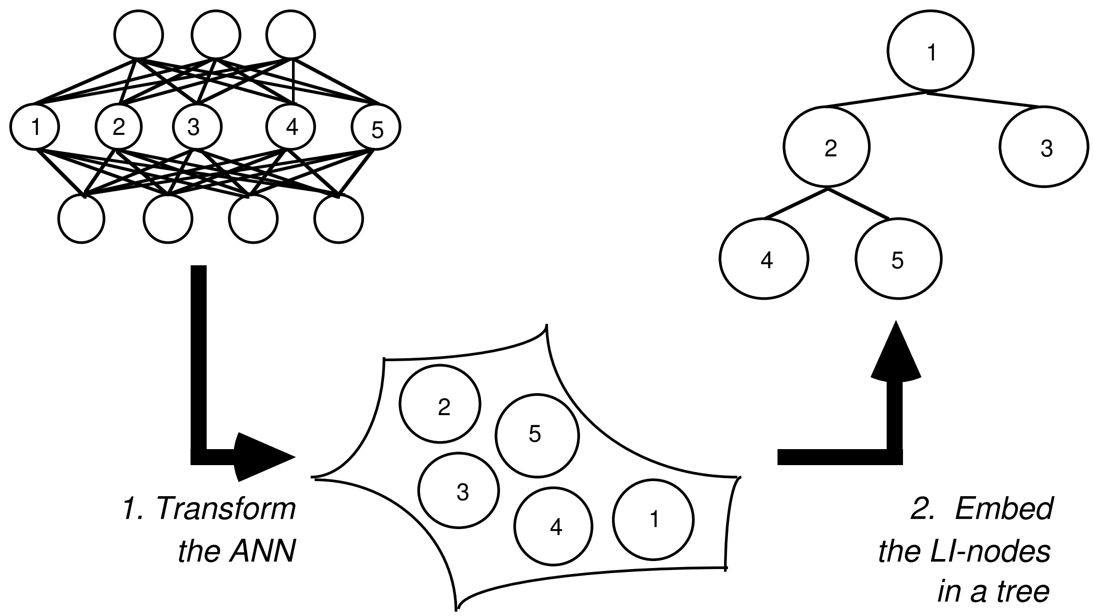


Figure 2. An Example Location-Independent Transformation

A communication topology can be chosen as a matter of speed, cost and reliability. The same topology can be used with many models. A binary tree topology is specified in the second step in figures 1 and 2. It is a simple, fast, regular, hierarchical topology that naturally supports broadcast and gather operations. It provides a basis for applying the transformation and explaining the associated execution and learning algorithms, without obscuring the details with a complex interconnect. However, a hypercube, or some other more fault-tolerant topology, could also be used. Hierarchical topologies allow adding or deleting nodes with time logarithmic in the number of nodes in the network, while limiting connectivity among nodes. The connectivity of a binary tree does not explode because each node is connected to one parent and zero to two children. The algorithms for the transformed network show explicitly how the required communication can take place.

An example illustrates the support for dynamic topologies. When a node is added to (deleted from) an original network, both the node and its connections to other nodes are added (deleted). If a hidden node 6 were added to the original network in figure 2, four connections to the input and three connections to the output nodes would also be added.

This is accomplished in the transformed network by allocating a free node, such as the left child of node 3, and initializing it as a 4x3 node with the corresponding weights. The deletion of a hidden node in the original network, such as node 5, is accomplished in the transformed network by marking the corresponding node as "free". No other nodes are affected.

3. Transformation Of 2-Layer BP Networks

It is assumed that the reader is familiar with the standard BP model [Rum86], however some necessary definitions are given here. These definitions hold throughout the rest of the paper. Following that is a discussion of the structure and operation of a transformed node in a 2-layer network. The transformation for 2-layer networks forms the basis of the transformation for multilayer networks. In this description, variables are introduced, used and defined informally. Formal definitions of variables are given in section 5 with the algorithms.

3.1. BP Definitions

In a BP network, there are typically three groups of nodes—input nodes, output nodes, and hidden nodes (see figure 3). In the standard BP model, the input nodes act as place-holders only, and do not perform any function on the values they receive (some BP models may use the nodes for preprocessing such as normalization, but this is not part of the basic model). A network may have an arbitrary number of layers of weights, but 2- and 3-layer networks (counting by the number of *weight* layers) are the most common. Here, and throughout the rest of this paper, given a node j, all the nodes in the same *node* layer as j are referred to collectively as the *current layer*. The nodes in the node layer labeled with k are referred to as the *subsequent layer*. The nodes in the node layer labeled with i are referred to as the *previous layer*. This terminology is straightforward in execution mode or in the feedforward phase of learning. Note that the names remain the same during the backpropagation phase of learning. From the point of view of a hidden node j, the weights on its inputs are referred to collectively as $\boldsymbol{u_j}$, and the weights on its output as $\boldsymbol{w_j}$.

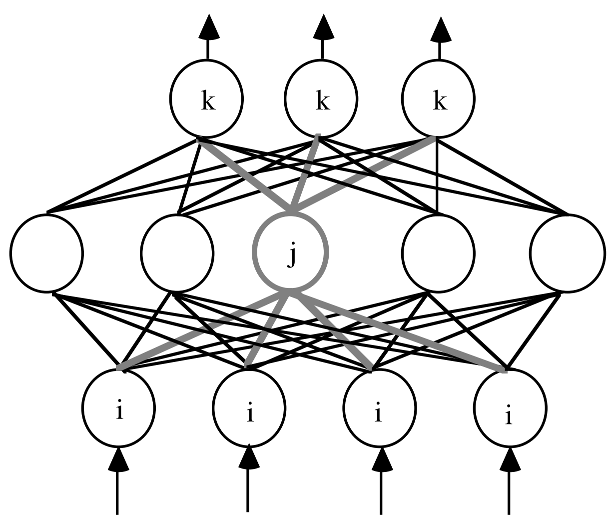


Figure 3. Hidden node *j* in a BP network.

3.2. Mapping 2-Layer BP Networks

Figure 4 shows the transformation of an original network into a 6-node location-independent network (from figure 2). There are six nodes in the new network, one node in the transformed network for each node in the original hidden layer, plus one bias node.

Figure 5 shows a node in the transformed network (*transformed node*) in more detail—the left side shows a node operating during execution mode and the forward phase of learning, while the right side shows the backpropagation phase of learning.

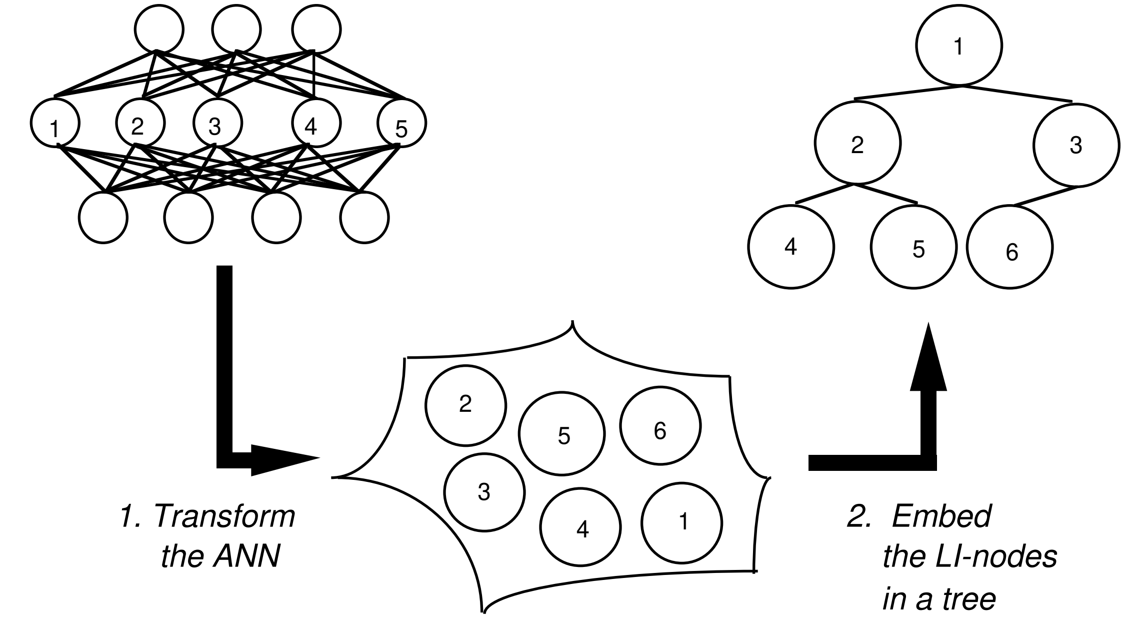


Figure 4. Transforming a 2-layer BP Network

The structure and operation of a transformed node (except the bias node) and transformed network is best introduced by informal comparison with an original hidden node, node 2 in figure 4, for example. The original node 2 has four weights from the input layer, and three weights to the output layer. A transformed BP node therefore stores two vectors (layers) of weights—one on inputs (hereafter $\boldsymbol{u_j}$) and one on outputs (hereafter $\boldsymbol{w_j}$). The transformed node 2 has four weights in $\boldsymbol{u_j}$ and three weights in $\boldsymbol{w_j}$. Each weight in the transformed node, u_{ij} or w_{jk}, has the value of the corresponding original weight. Each transformed node also stores a bias value θ_j, which corresponds to the bias value in the respective original hidden node. For purposes of execution and learning, θ_j can be treated as a component of $\boldsymbol{u_j}$.

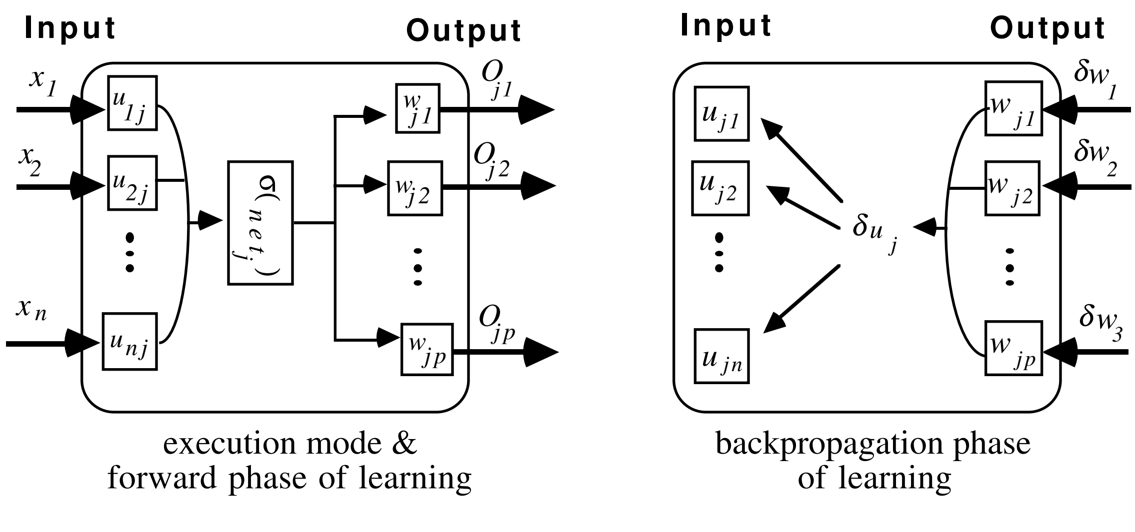


Figure 5. Two Views of a Node in the Transformed Network. (The left shows a node operating in execution mode and/or the forward phase of learning. The right shows a node during the backpropagation phase of learning.)

The bias node is like the other transformed nodes, with some exceptions. It needs no $\boldsymbol{u_j}$, because its activation is always 1. The elements of its $\boldsymbol{w_j}$ each correspond to one of the biases from the nodes in the original output layer. It has no additional bias values of its own. In all other respects, the bias node behaves like the other transformed nodes.

3.3. Execution Mode in the Transformed Network

The network executes as follows: The Control Unit sends the inputs to the root node, which passes those values onto each of its children, and they to theirs, and so on. Each node (except the bias node) receives the broadcast, multiplies each input by the corresponding weight on the input, and sums the results together with each node's bias. The result of this summation is the same as the original net_j. $f(net_j)$, the sigmoid of each net_j, is then computed at each node. This corresponds to the scalar O_j for a hidden node j in the original network. In the bias node, $f(net_j)$ is not computed, but is always 1. The weights on output are each multiplied by $f(net_j)$, to produce an output *vector* which is the new $\boldsymbol{O_j}$. $\boldsymbol{O_2}$, the output vector for the transformed node 2, corresponds to the weights out of the original node 2 each multiplied by the scalar O_2. The vector summation of the transformed nodes' output vectors is accomplished by a gather operation. Node 2 receives $\boldsymbol{O_4}$ and $\boldsymbol{O_5}$ from its children, does the vector sum of these with $\boldsymbol{O_2}$, and sends the result to the root node, for example. The components of the vector $\boldsymbol{Y}$ (the vector sum of all $\boldsymbol{O_j}$ including the $\boldsymbol{O_j}$ of the bias node) at the root node correspond to net_j in each of the output nodes in the original network.

At this point $\boldsymbol{Y}$ resides at the root node. Now however, the sigmoid of each component of $\boldsymbol{Y}$ needs to be computed, in order to produce values corresponding to the output values of the original output nodes. This vector of output values is called $\boldsymbol{Z}$. $\boldsymbol{Z}$ can be computed in parallel fashion in two ways:

1. by a pipelined computation executed as the root node sends $\boldsymbol{Y}$ to the Control Unit, or
2. by distributing (through rebroadcasting) $\boldsymbol{Y}$ to the nodes in the network, where each node computes z_i from a component y_i, and then gathering $\boldsymbol{Z}$.

There are two main factors involved in deciding how to compute $\boldsymbol{Z}$ from $\boldsymbol{Y}$: computation time and communication time. If computation time is fast relative to communication time, then the first method is more efficient. If, on the other hand, communication is fast relative to computation, the second method is more efficient. For BP, computation of the sigmoid is very fast compared to the time needed to send values to the Control Unit, or to broadcast back to the network—thus, for the BP model, the method using the pipe is preferable.

Using the piped scheme then, $\boldsymbol{Z}$ is computed as the root node sends $\boldsymbol{Y}$ to the Control Unit, at which point the Control Unit has the network output. For execution, this pipe has only one computation—the sigmoid—but the pipe for learning (below) has more steps.

3.4. Learning Mode in the Transformed Network (Non-dynamic BP)

As in the original model, learning iterates over a series of patterns, until convergence is reached. As each pattern is presented for learning, the network proceeds through a forward phase and then a backpropagation phase, as follows: First, present the training pattern to the network as above for execution, up to the point at which $\boldsymbol{Y}$ resides at the root node. This is the forward phase. Second, the backpropagation phase begins.

The first step of the backpropagation phase is to compute an error value (δ) for each of the output values, and from that, to compute *psse*, the single value representing the pattern sum-squared error for the current pattern. This first step is accomplished by a pipe like the one at the end of execution, but with some additional computations. First, $\boldsymbol{Z}$ is computed from $\boldsymbol{Y}$ as above. Second, $\boldsymbol{Z}$ and $\boldsymbol{T}$ are used to compute the error vector $\boldsymbol{\delta w}$, which will be sent back to the nodes. (Target outputs $\boldsymbol{T}$ are at the Control Unit, and therefore are directly available to the pipe.) Third, δw_k (each component of $\boldsymbol{\delta w}$) is squared to produce δw_k^2 (for calculating sum-squared pattern error). Fourth, each δw_k^2 value is summed into *psse* at the "end" of the pipeline (*psse* is used only to check for convergence during learning.) Each δw_k in $\boldsymbol{\delta w}$ corresponds to the δ_k computed by the respective output node in the original network.

The second step of the backpropagation phase propagates the error backward (see the right half of figure 5). The vector $\boldsymbol{\delta w}$ is broadcast to the nodes. This value is then used to compute the required weight change on each weight (the other information already exists at the node). Also at each node, like node 2, each w_{jk} is multiplied by the corresponding component of $\boldsymbol{\delta w}$, and the results are summed. This new value is used to create a δ value, $\boldsymbol{\delta u_j}$, with which to calculate changes for each u_{ij} (except in the bias node, which has no $\boldsymbol{u_j}$). Once the appropriate computations are completed, all weights can be updated, and the backward phase ends. The network is ready to move on to the next pattern.

There are a variety of potential location-independent sets of nodes that could be constructed. The construction given here results in a node structure that is very similar to the original model. An analysis of the original equations and structures reveals a symmetry both in the information used for computation, and in the behavior during forward and backpropagation phases. The LIT structure reflects that symmetry, as well as achieving the other goals of LIT.

4. Transforming Multilayer BP Networks

The transformation given in section 3 forms the basis of the transformation for multilayer networks. Similarly, the execution and learning algorithms discussed above form the basis of the algorithms that handle multiple layers.

Given an arbitrary number of weight layers q, a network is first partitioned into a sequence of disjoint 2-layer networks, each of which is called a *wave*. Second, each *wave* is mapped onto a set of LIT nodes like those shown previously in section 3, and the nodes are inserted into the tree as before. Each individual wave behaves like the 2-layer network of section 3, except that each transformed node now stores two additional pieces of information:

1. the number of the wave to which the node belongs, and
2. an "odd-layer" flag, whose value is 1 if and only if q is odd *and* the node is in the final wave, and 0 for all other nodes.

Each node uses the wave number to determine when to compute and contribute its information to the network. The purpose of the odd-layer flag is to support networks with an odd number of weight layers (using the same node structure as before). The key issues in the multilayer transformation are what and how information is communicated from one wave to the next (or previous) wave.

The number of waves in the new network is given by $ceiling(\frac{q}{2})$. If q is even, the original network divides evenly into disjoint 2-layer waves. Section 4.1 uses a 4-layer network to show how this mapping is accomplished. An informal discussion of execution and learning modes, like that of section 3, is also given. If q is odd, the network divides into 2-layer waves with the last layer "left over". There are a number of different ways to handle the "odd last layer" case. Section 4.2 describes one possible mechanism that is fairly straightforward. Section 4.3 gives a formal discussion of the execution algorithm, which is essentially an iteration over the algorithm of section 3.3. Section 4.4 describes how the 2-layer learning algorithm is extended to multiple layers.

The overall transformation can be summarized as follows:

1. Partition the original network into 2-layer waves, starting with the first weight layer.
   - Conceptually, each wave is composed of two consecutive layers of weights with the adjacent nodes.
   - If q is odd, the last layer of weights, with its adjacent nodes, goes into the final wave.
2. For each wave:
   1. Construct the set of LI-nodes for that wave,
   2. label each node with the wave number,
   3. set the odd-layer flag in each node to 1 if and only if q is odd *and* the node belongs to the final wave, otherwise set the odd-layer flag to 0.

4.1. Mapping an Even Number of Weight Layers

Figure 6 illustrates the transformation of a 4-layer network. The weight layers and nodes are labeled for reference purposes. The original network has 4 layers of weights. It is divided into 2 waves: The first wave handles first two weight layers, and the he second wave handles the third and fourth weight layers.

The first wave has four 5x4 transformed nodes (nodes 1-3 in figure 6, plus a bias node), which accounts for the first two layers of weights and the adjacent nodes. The second wave has three 4x2 transformed nodes (nodes 4 and 5 in figure 6, plus a bias node), which accounts for the second and third layer of weights and the adjacent nodes. Note that the location of a particular node is arbitrary. If the information in node 1 is switched with node 4 and node 3 is switched with node 5, the output of the network is unaffected by the change.

A "2-layer" node structure was chosen because it is convenient for gathering vectors of output values in the correct order (synchronization), and for the backpropagation of error during learning. It is a compromise between node complexity, synchronization and the number of broadcasts and gathers required.

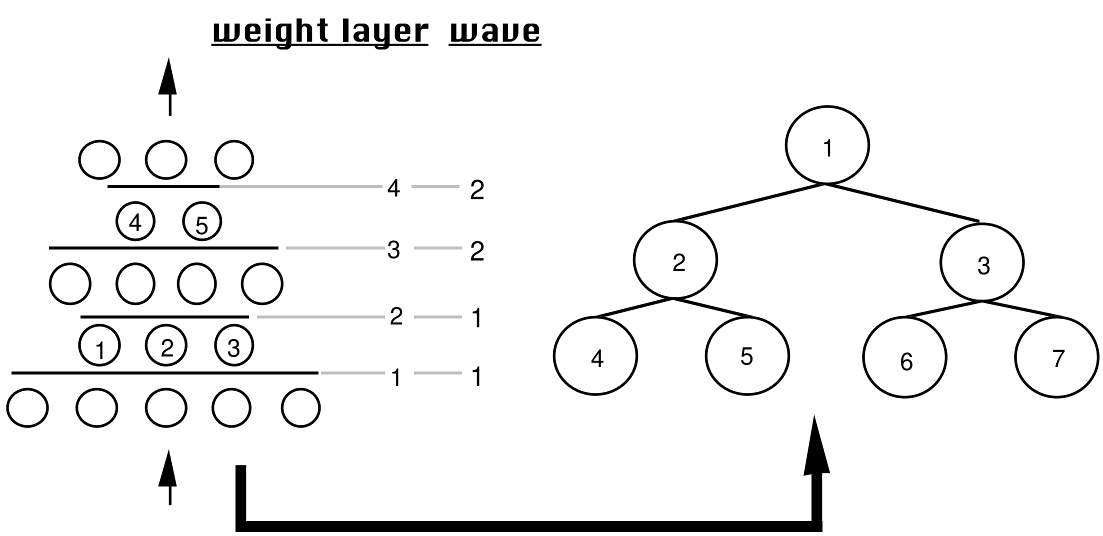


Figure 6. Example Transformation for a 4-layer Network (number of weight layers is even).

4.2. Mapping an Odd Number of Weight Layers

Figure 7 illustrates the transformation of a 3-layer network. As in the example above, the weight layers and nodes are labeled for reference purposes. The original network has three layers of weights. Basically, the solution presented here amounts to adding an extra layer of weights, making the network an even-layered network. However, this extra layer of weights is *never* trained, does not alter the function computed by the original network, and can be shown not to seriously affect the performance of the transformed network (see below and section 6).

The first wave has four 5x4 transformed nodes (nodes 1-3 in figure 7, plus a bias node 4), which accounts for the first two weight layers and adjacent nodes. Since q is odd (three), the second wave (final wave) only has to account for a single layer of weights with the adjacent nodes, referred to as the “odd layer”. Using the following method of handling the odd layer, the second wave will end up with two 4x2 nodes (nodes 5-6 in figure 7).

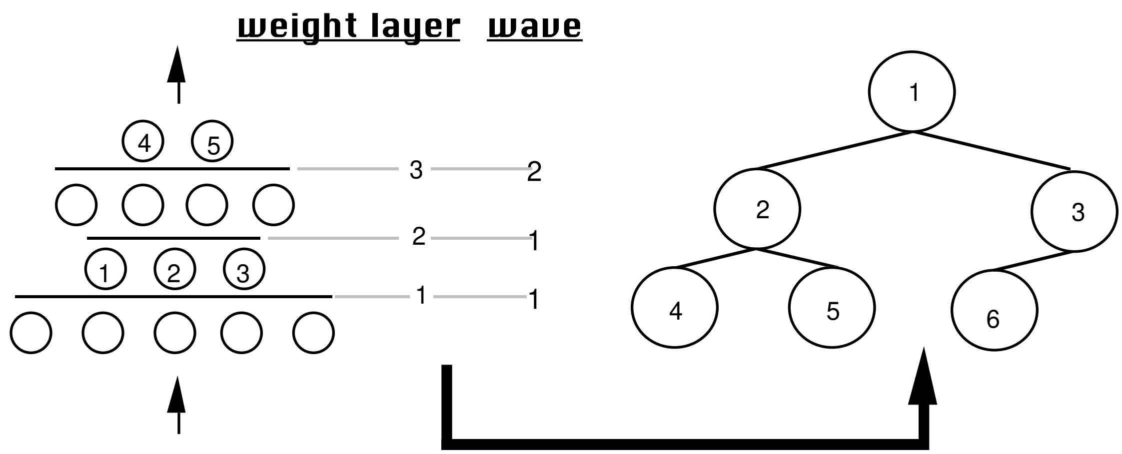


Figure 7. Example Transformation for a 3-layer Network (number of weight layers is odd).

If the odd layer is mapped into a 2-layer node structure in the same fashion as the previous weight layers, then each transformed node in the final wave would compute only a single output variable O_j, rather than the desired vector $\boldsymbol{O_j}$, because there would be no second layer of weights. The essential problem then, is to communicate each O_j from the node (in the final wave) to the CU without sacrificing a lot of parallelism.

The solution is to project each O_j (in the final wave only) onto a base vector of size p, where p is the number of outputs in the original network. This can be done using the second vector of weights $\boldsymbol{w_j}$ in the nodes of the final wave. Each $\boldsymbol{w_j}$ is set to 0, except exactly one weight is set to 1 in each node. The position of the 1 in each node must be unique, and corresponds to which (original) output variable the node computes. For example, suppose there are 3 output nodes in some odd-layer network. The $\boldsymbol{w_j}$'s of the nodes in the final wave would look like this: node 1 has 100, node 2 has 010 and node 3 has 001. These values are clamped for both learning and execution, and never get trained.

During learning, the odd-layer flag is used to clamp the $\boldsymbol{w_j}$'s of the transformed nodes in the final wave of an odd-layer network. However, this flag has another use during both learning and execution: it causes the node to make O_j equal to net_j instead of $f(net_j)$. In execution mode, for example (the feedforward portion of learning is the same), each node multiplies its single value O_j by its $\boldsymbol{w_j}$. The resulting vector $\boldsymbol{O_j}$ has 0 everywhere except where O_j should be. When the vectors are summed together, all the values end up in their proper places in $\boldsymbol{Y}$. In all the other waves, the components of $\boldsymbol{Y}$ are run through the sigmoid pipe at this point. But, the sigmoid of each value only needs to be taken once—so, in order to make the operation of the network uniform "outside" the node, each node projects it's net_j, rather than the sigmoid of net_j, so that the correct value enters the sigmoid pipe.

It should also be clear that the 0's mask out all error delta values (in learning) for a node except the single one pertaining to that node. Thus, each node always uses the correct delta values for its computations. Other than the changes specifically described here, learning and execution proceed as before.

4.3. q-layer Execution

Execution mode requires very little additional explanation beyond section 3 above. The basic idea is that the output vector $\boldsymbol{Z}$ for the current wave, becomes the input vector $\boldsymbol{X}$ for the subsequent wave. In the example of figure 6, five input values are broadcast to the network in wave 1. Nodes 1-3 and 6 respond, each producing a four element vector of partial values, summed as described in section 3. Nodes 4, 5 and 7 do not respond, or can be considered to produce $\mathbf{0}$ vectors of the proper size, because they have no children. $\boldsymbol{Z}$ for wave 1, then, is a four-element vector. Nodes 4 and 5, in wave 2, require four inputs—the $\boldsymbol{Z}$ from wave 1. During the second wave, nodes 1-3 and 6 do not respond with output vectors of their own, however, they do gather results from their children (if any exist) and pass these results on. The outputs of nodes 4 and 5 are summed in node 3, and the result is passed to node 1, etc. The three-element $\boldsymbol{Z}$ for wave 2 is the output of the network.

4.4. q-Layer Learning

In the original BP network, the error values at a given hidden layer depend on both the weights and the error values passed back from the subsequent layer. Since a transformed node stores two layers of weights, the error information in the transformed node becomes available through a single broadcast for each wave. In the original network, each node produces a single error value which is passed back to the next layer. In the transformed network, each node produces not a single error value, but instead a vector of partial error values $\boldsymbol{D_j}$ (with components d_{ij}) to be passed back. These vectors of partial values are gathered by vector summation, and the result at the root node is the vector of error values $\boldsymbol{V}$. Each component of $\boldsymbol{V}$, v_k, is then multiplied by the first derivative of the corresponding component of the output vector $\boldsymbol{Z}$, z'_k. The resulting products are the components of $\delta\boldsymbol{w}$ to be broadcast to the previous wave. It is assumed that each node outputs corresponding components of a vector in the proper order, therefore a gathered vector of results is always in the proper order.

The values for $\boldsymbol{Z}$ for each wave are needed to compute z'_k values during backpropagation. If the number of waves is small, each intermediate $\boldsymbol{Z'}$ (vector whose components are first derivatives of the components of $\boldsymbol{Z}$) can be computed and stored in the Control Unit during the forward phase of learning. Another option, that uses less memory and is more dynamic, is to recompute $\boldsymbol{Z}$ for each wave after $\boldsymbol{V}$ has been gathered, and use this to compute each $\boldsymbol{Z'}$ as it is needed.

5. Execution and Learning Algorithms

5.1. Execution Algorithm

The following equations describe execution mode for LIT BP with an arbitrary number of layers. They are essentially the original equations, but altered to show the behavior of the LIT model.

<u>Definitions</u>

- q number of weight layers in the original network.
- n number of nodes in the original input layer, size of $\boldsymbol{u_j}$.
- m number of nodes in the original hidden layer, number of nodes in the transformed network.

p	number of nodes in the original output layer, size of $\boldsymbol{w_j}$.
a	real-valued number, it will be either net_j or y_k.
$\boldsymbol{O_j}$	output vector of transformed node j.
O_j	intermediate activation value for transformed node j. In general, it corresponds to O_j for an original node j. If and only if q is odd, and transformed node j is in the final wave, then O_j is just net_j.
net_j	real-valued sum-of-products of the inputs to node j and $\boldsymbol{u_j}$.
f	sigmoid function.
$\boldsymbol{X}$	pattern clamped on inputs, has size n.
x_i	ith real-valued component of $\boldsymbol{X}$.
$\boldsymbol{u_j}$	vector of weights on inputs in a new node j, has size n.
u_{ij}	real-valued ith component of $\boldsymbol{u_j}$. It corresponds in the original network to the weight from input node i to hidden node j.
o_{jk}	real-valued kth component of $\boldsymbol{O_j}$. It corresponds in the original network to the weight from hidden node j to output node k, multiplied by the original O_j.
$\boldsymbol{w_j}$	vector of weights on output in a new node j, has size p.
w_{jk}	real-valued kth component of $\boldsymbol{w_j}$. It corresponds in the original network to the weight from hidden node j to output node k.
$\boldsymbol{Y}$	vector of intermediate activation values, the sum of all $\boldsymbol{O_j}$.
y_k	real-valued kth component of $\boldsymbol{Y}$. It corresponds in the original network to the net_j of output node k.
$\boldsymbol{Z}$	vector of network output values.
z_k	real-valued (always between 0 and 1)kth component of $\boldsymbol{Z}$. It corresponds in the original network to the output O_k of output node k.
$olbit_j$	Boolean flag in node j that is 1 if and only if q is odd and node j is in the final wave of the network, otherwise it is 0. In the text, it is called "odd-layer bit".

Equations

$$f(a) = \frac{1}{1+e^{-a}} \tag{1}$$

$$O_j = \begin{cases} f(net_j)\ ,\ if\ olbit_j\ \ is\ 0 \\ net_j\ ,\ if\ olbit_j\ is\ 1 \end{cases} \tag{2}$$

$$net_j = \sum_{i=1}^{n} x_i u_{ij} + \theta_j \tag{3}$$

$$o_{jk} = O_j w_{jk} \tag{4}$$

$$y_k = \sum_{j=1}^{m} o_{jk} \tag{5}$$

$$z_k = f(y_k) \tag{6}$$

The main steps of the execution algorithm (italics), numbered for convenient reference, describe the basic BP algorithm. The sub-steps (1.1, 2.1, etc.) specify the details of how the LIT network accomplishes these steps. The algorithm is as follows:

For wave 1 to *ceiling* $(\frac{q}{2})$

(1) Clamp an input pattern on the input nodes.

1.1 Broadcast input vector $\boldsymbol{X}$ to the nodes.

(2) Each node takes the sigmoid of the sum-of-products of its weights and inputs

2.1 Each node computes its net_j, according to equation (3), and then computes O_j according to equation (2). O_j for the bias node is 1.

2.2 Each node's weight vector $\boldsymbol{w_j}$ is multiplied by its (scalar)O_j to produce its output vector $\boldsymbol{O_j}$ [equation (5)].
2.3 Each node computes the vector sum of its $\boldsymbol{O_j}$ with the result from each child, and the sum (a vector) is sent to its parent. (The Control Unit is the parent of the root node.). This implements equation (6). The result is the vector $\boldsymbol{Y}$.

(3) Wait until the activation values on the output nodes are valid.

3.1 $\boldsymbol{Y}$ is sent (through the sigmoid pipe) to the Control Unit to compute $\boldsymbol{Z}$.
3.2 The result $\boldsymbol{Z}$ becomes $\boldsymbol{X}$ for the subsequent wave, unless the current wave is the last wave—then $\boldsymbol{Z}$ is the output of the network.

end

5.2. Learning Algorithm (Standard BP)

Below are additional equations that describe the learning mode for the LIT BP. The forward phase of learning is identical to execution mode, and so follows the equations given above.

Additional Definitions

α	weight change momentum constant.
$\delta\boldsymbol{w}$	vector of error values for all $\boldsymbol{w_j}$. It has size p.
δw_k	real-valued jth component of $\delta\boldsymbol{w}$. It corresponds in the original network to δ_k for output node k.
δu_j	real-valued error value for $\boldsymbol{u_j}$ in node j. It corresponds in the original network to δ_j for hidden node j.
$\boldsymbol{T}$	vector of target values for the current pattern being presented. It has size p.
t_k	target value for z_k. It corresponds in the original network to the target value for O_k for output node k.
psse	pattern sum-squared error for the current pattern being presented.
tsse	total sum-squared error over all patterns in the current learning epoch.
ecrit	maximum error tolerance for learning.
s	number of patterns in the training set (epoch).
r	simple index, used to avoid confusion with i, j, k, etc.
c	current training cycle number.
η	a real-valued learning constant.
$\boldsymbol{Z'}$	vector of size p, whose components are z'_k.
z'_k	real-valued kth component of $\boldsymbol{Z'}$, it has value $f'(y_k)$.
$\boldsymbol{V}$	vector of intermediate sums for computing $\delta\boldsymbol{w}$ for the previous hidden wave, it has size p.
v_k	real-valued kth component of $\boldsymbol{V}$.
$\boldsymbol{D_j}$	the part of $\boldsymbol{V}$ computed by node j.
d_{ij}	real-valued ith component of $\boldsymbol{D_j}$.

Equations

$$f'(a) = \frac{\partial f(a)}{\partial a} = f(a)(1-f(a)) \tag{7}$$

$$\Delta w_{jk}(c) = \eta O_j \delta w_k + \alpha \Delta w_{jk}(c-1)$$
$$\Delta u_{ij}(c) = \eta x_i \delta u_j + \alpha \Delta u_{ij}(c-1) \tag{8}$$

$$\delta w_k = \begin{cases} (t_k - z_k) f'(y_k), \text{if output wave,} \\ v_k z'_k, \text{ otherwise.} \end{cases} \tag{9}$$

$$\delta u_j = \left(\sum_{k=1}^{p} (\delta w_k w_{jk})\right) f'(O_j) \tag{10}$$

$$tsse = \sum_{r=1}^{s} psse_r \tag{11}$$

$$psse_r = \sum_{k=1}^{p} \delta w_k^2 \tag{12}$$

$$v_i = \sum_{j=1}^{m} d_{ij} \tag{13}$$

$$d_{ij} = \sum_{j=1}^{m} \delta u_j u_{ij} \tag{14}$$

As with the execution algorithm above, the steps of the learning algorithm are numbered for convenient reference. The sub-steps give the details of how LIT accomplishes each step. The learning algorithm is as follows:

Until Convergence ($tsse < ecrit$ at the end of an epoch)

(1) Present a training pattern to the network

- 1.1 Forward phase of learning—this is same as execution above for all but the last wave.
- 1.2 The last wave is the same up to, and including execution step 2.3.

(2) Calculate the error δ of the output units (T - O)

- 2.1 As each y_k is sent to the Control Unit, the following occurs:
  - 2.1.1 z_k is computed [equation (6)], and received by the Control Unit.
  - 2.1.2 z_k and t_k are used to compute δw_k [equation (9)], which is received by the Control Unit.
  - 2.1.3 δw_k is used to compute δw_k^2.
  - 2.1.4 Each δw_k^2 is summed with the value of $psse$ so far [equation (12)].

(3) For each hidden layer Calculate δ_j using δ_k from the subsequent layer

For wave $ceiling(\frac{q}{2})$ downto 1

- 3.1 The vector $\boldsymbol{\delta w}$ is broadcast to the nodes.
- 3.2 δu_j is calculated according to equation (10).
- 3.3 $\boldsymbol{D_j}$ is calculated according to equation (14).
- 3.4 $\boldsymbol{V}$ is calculated according to equation (13).
- 3.5 $\boldsymbol{\delta w}$ for the previous wave is calculated according to equation (9).

end

(4) Update the weights

- 4.1 Calculate Δw_{jk} and Δu_{ij} according to equation (8).
- 4.2 Change the weights, except for all $\boldsymbol{w_j}$ in nodes where the odd-layer bit is 1.

(5) Update tsse according to equation (11)

end

Step 5 may actually be computed any time after the end of step 2—independent of steps 3 or 4. Most algorithms assume that *psse* and *tsse* can be maintained, but give no explicit mechanism for doing so—LIT provides a mechanism. The weights may also be updated all at once at the end (as given here), or at the end of each backward wave.

While considering the algorithms presented in this section, it may have occurred to the reader that there may be many more nodes in the network than are active during any given time. A physical implementation of the network is typically concerned with node usage and efficiency. One way to increase these is to have a physical node store weights for more than one wave. That way, a particular node could be active a higher percentage of the time.

One might then consider how many waves each node should store, since a physical node will always have some limit. The most common BP networks are 2-layer networks—these require only one wave. 3-layer networks are next most common—and these require two waves (as in the example above). Networks with more than 3 layers of weights are less frequent, though they do exist. Lippmann provides a proof which states that 3 layers of weights are sufficient for any mapping [Lip87]. These ideas suggest that two waves is a reasonable number to store per physical node.

6. The Dynamic BP Model

This section outlines the dynamic BP model based on the model proposed by Odri, et al [Odr93]. This model is of interest because it allows the addition and deletion of both nodes and individual weights dynamically during learning. The algorithm is given here to indicate to the reader how dynamics can be supported by LIT. More detail can be found in [Rud95a]. The steps below are numbered to correspond to the algorithms of section 5.

Until Convergence ($tsse < ecrit$)

1. Present a pattern to the network.

2-5. Perform standard BP learning, including adjusting the weights.

6. Perform probabilistic weight deletion.
   - 6.1 Each weight computes a probability of self-deletion in the range 0-1.
   - 6.2 Each weight generates a random value in the range 0-1.
   - 6.3 If the probability is greater than the random value, then the weight is deleted.
7. Perform node deletion on the hidden nodes (the number of input and output nodes remains fixed).
   - 7.1 If $\boldsymbol{u_j}=\boldsymbol{0}$, or $\boldsymbol{w_j}=\boldsymbol{0}$, then node j self-deletes.
   - 7.2 In the case that $\boldsymbol{u_j}=\boldsymbol{0}$, and the bias $\theta_j \neq 0$, then adjust the biases of the remaining nodes to compensate.
8. Check for node addition (by probabilistic node division).
   - 8.1 Each node computes a probability of dividing, in the range 0-1.
   - 8.2 Each weight generates a random value in the range 0-1.
   - 8.3 If the probability is less than the random value, then the node does not divide—skip step 9.
9. Do node division (only if at least one node divides)
   - 9.1 Two descendant nodes are created from the node: The $\boldsymbol{u_j}$ for both remain the same as the parent, but $\boldsymbol{w_j}$ are mutated in order to guarantee that the nodes are unique.

end

7. Complexity And Performance

The complexity of the algorithms in section 5 (including the steps in section 6) can be characterized by pipelined broadcast and gather in a tree of nodes. The width of a broadcast block is assumed to be the size of a single variable. The tree has m+1 nodes, one for each hidden node in the original network, plus a bias node, and the nodes operate in parallel. Assume that n is the number of nodes in the widest layer of the *original* network, and that the original network has q weight layers. Given these assumptions, analysis gives an overall complexity of $O(q(n+logm))$ for both execution and learning, for a single pattern. Note that if q is constant, or very small, this reduces to $O(n+logm)$.

The complexity of the steps of the execution algorithm *for a single wave* is summarized, followed by elaboration where necessary. The numbers correspond to the steps from the algorithms in sections 5. The complexity of the learning algorithm is presented in the same format, including the dynamic extensions from section 6.

Execution Algorithm for a Single Wave

1.	Broadcast	$O(n+logm)$
2-2.2.	Node Computation	$O(n)$
2.3-3.	Gather	$O(n+logm)$
total		$\boldsymbol{O(n+logm)}$

Broadcast or gather for a single variable is clearly $O(logm)$. For n variables, one might expect a complexity of $O(nlogm)$, but the pipelined tree reduces this to $O(n+logm)$, since different levels of the tree operate on different variables simultaneously. The node computation is an $O(n)$ operation for n variables, but can be overlapped with broadcast. The sigmoid pipeline at the end of the gather is $O(n)$, one for each output variable, but this is overlapped with the gather of variables up the tree.

Learning Algorithm for a Single Wave

1.	Forward Phase	$O(n+logm)$
2.	Compute $\delta \boldsymbol{w}$	$O(n)$
3.1	Broadcast $\delta \boldsymbol{w}$	$O(n+logm)$
3.2	Calculate δu_j	$O(n)$
4.	Update weights	$O(n)$
5.	Partial sum *tsse*	constant
6.1-6.4	Weight deletion	$O(n)$
6.5-6.6	Gather deletion flags	$O(logm)$
7.1-7.2	Node deletion check	$O(n)$
7.3	Compute Adjustments	$O(n)$
7.4-7.5	Gather Adjustments	$O(n+logm)$
7.6	Broadcast Adjustments	$O(n+logm)$
7.7	Adjust biases	$O(n)$
8.1-8.3	Node division check	$O(n)$
8.4-8.5	Gather division flags	$O(logm)$
9.1-9.2	Broadcast divide flag	$O(logm)$
9.3-9.4	Allocate new node(s)	$O(n+logm)$
total		$\boldsymbol{O(n+logm)}$

Step 2 is $O(n)$ because the is one δ value for each output variable. Step 4 is $O(n)$ assuming that a node can update only one weight at a time. Steps 6.1-6.4 and 7.1-7.2 also assume that only one weight can be checked at a time. Step 7.3 is $O(n)$ because each node generates one value for each of its weights in $\boldsymbol{w_j}$. Steps 7.4-7.5 have the same character as the gather for execution, only the data is different. Step 7.7 is $O(n)$ because n summations are performed at each node. Steps 8.1-8.3 are $O(n)$ because the cumulative error values from each of the weights in $\boldsymbol{u_j}$ is used to compute $\beta_j(c)$, etc. Steps 9.3-9.4 are $O(n+logm)$ assuming that there is a free node in the subtree of the parent to hold each new node—at most$n$ input variables and at most n output variables have to be sent to each new node.

Probabilistically speaking, only one node will likely divide at any given time *c*. However, if more than one node does divide, and free nodes exist but are not guaranteed to be in the respective subtrees, then they could be added sequentially, one by one. In this case, Steps 9.3-9.4 are $O(dnlogm)$, where d is the number of nodes that simultaneously divide.

The analysis in this section shows that the complexity of execution and learning for the algorithms is $O(q(n+logm))$.

8. Conclusion

ANNs that use a static topology, i.e. a topology that remains fixed throughout learning, suffer from a number of short-comings. Current research is demonstrating the use of dynamic topologies in overcoming some of these problems. The Location-Independent Transformation (LIT) is a general strategy for implementing variations of ANNs that use dynamic topologies during learning.

This paper introduced ideas for LITs for static and dynamic feedforward, distributed neural networks with an arbitrary number of layers. An LIT with associated execution and learning algorithms for backpropagation were formally defined, as an example. Analysis gives overall complexities for both execution and learning as $O(q(n+logm))$, where q is the number of weight layers in the orignal network, n is the number of nodes in the widest node layer of the original network, and m is the number of hidden nodes in the original network.

Current work includes the following:

- LITs for other ANNs, aside from those for which transformations have been developed.
- VLSI design and fabrication of LIT models.

CHAPTER 5

LIA: A Location-Independent Transformation for ASOCS Adaptive Algorithm 2 (AA2)

submitted, 1995

The LIA and PASOCS models were the initial results of research into practical hardware implementations for ASOCS models, particularly AA2. The first complete description of LIA is Rudolph's MS Thesis [Rud91b]. It's main features are: 1) each physical node stores a complete instance, and 2) the computation at a single node is independent of all other nodes.

The paper included here is essentially a journal-length version of the Thesis, but also incorporates important ideas and perspectives gained from the additional research that has been done in the interim since the Thesis was published. This paper places LIA within the overall framework of LIT, and also includes formal descriptions of ASOCS structures and operators, which were found to be necessary in creating precise hardware implementations [Sto94b].

Early, introductory descriptions of LIA models can be found in two conference papers [Rud89], [Rud91a].

1. INTRODUCTION

Hardware support for ANNs is important for handling large, complex problems in real time. Learning times can exceed tolerable limits for applications of large size and complexity using conventional computing schemes. Furthermore, hardware is becoming cheaper and easier to design. The *Location-Independent Transformation* (LIT) is a general implementation strategy for ANNs that overcomes several weaknesses of current hardware implementation methods.

Most ANNs use static topologies—the topology is fixed initially, and remains the same throughout learning. ANNs with static topologies often suffer from the following short-comings:

- sensitivity to user-supplied parameters: learning rate(s), etc.,
- local error minima during learning,
- there is no mechanism for deciding on an effective initial topology (number of nodes, number of layers, etc.).

Current research is demonstrating the use of dynamic topologies in overcoming these problems [Baf92], [Den93], [Fah88], [Fahlm], [Kar90], [Lee93], [Mar91b], [Mea91], [Odr93], [Rei88]. A *dynamic topology* is one which allows adding and deleting both nodes and weighted connections during learning.

Early ANN hardware implementations are model-specific, and are intended to support static topologies [Far85], [Gra90], [Mea91]. More recent *neurocomputer* systems have specialized neural hardware, and seek to support more general classes of ANNs [Ham91], [Ram93], [Sha92]. Although some neurocomputers could potentially support dynamic topologies more directly in hardware, rather than in software, they currently do not. Of course, general parallel machines, like the Connection Machine [Hil85] and the CRAY [Alm89], can simulate the desired dynamics in software, but these machines are not optimized for neural computation. LIT supports general classes of ANNs *and* dynamic topologies in an efficient parallel hardware implementation.

LIT redesigns the original network into a hierarchical, parallel network of Location-Independent nodes (LI-nodes). A *LI-node* contains enough information locally to compute its function independent of any other node in the network. The set of LI-nodes is then embedded in a uniform tree topology. Figure 1 gives a general view of LIT. Execution (running the network) and learning are based on efficient broadcast/gather operations in the tree. Because the nodes are location-independent, they can be placed anywhere in the tree,

and dynamic topologies can be efficiently and easily supported: Adding nodes is as simple as allocating free nodes in the tree; deleting nodes is as simple as de-allocating the desired nodes.

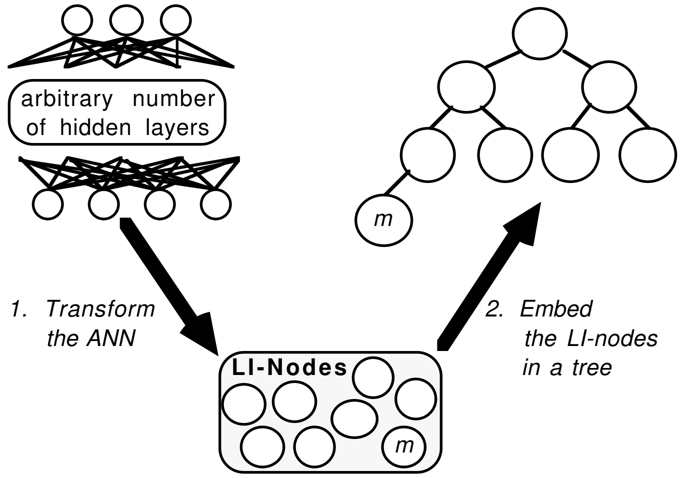


Figure 1. General LIT Transformation

LIT is a generalization of the authors' work on a strategy for implementing in hardware a class of learning models called Adaptive Self Organizing Concurrent System (ASOCS) [Mar88], [Mar91a,b] in particular AA2 [Mar91b], which have inherently dynamic topologies. The initial result of that work were the Location-Independent ASOCS model (LIA) [Rud91a] and the Priority ASOCS model [Mar94a], [Sto94b]. While the underlying mechanisms of ASOCS may differ significantly from those common among ANNs, the overall goals of ASOCS are similar to other such learning models, as is the need to support dynamic topologies. LITs have since been developed for counterpropagation and competitive learning [Rud95g], and backpropagation with an arbitrary number of layers [Rud95b]. LITs can potentially support a broad set of ANNs, thus allowing one efficient implementation strategy to support both inherently dynamic ANNs and variations of ANNs with dynamic extensions (such as backpropagation).

This paper presents the LIA model as a LIT for AA2. To understand the transformation, the reader needs to understand the AA2 model, relevant aspects of which are described at the beginning of section 4. However, the reader does not have to understand AA2 to understand LIA—knowledge of general ASOCS concepts and mechanisms, which are overviewed in section 3, is sufficient. This is because the approach taken for LIA is capable of supporting various ASOCS models, not just AA2. As such, the formal definitions given for LIA also provide a basis for formal definitions for ASOCS models in general.

Section 2 presents a general overview of LIT. Section 3 provides definitions of basic ASOCS terms and concepts. Section 4 gives the LIT transformation from AA2 to LIA. Section 5 presents formal definitions of the LIA execution and learning algorithms. Section 6 gives a complexity analysis of the algorithms in section 5. Section 7 is the conclusion.

2. THE TRANSFORMED NETWORK

A transformation redesigns the network so that each node contains enough information locally to compute its part of the network output, independent of any other node. A network whose nodes have this property is said to be *location-independent.* The nodes also are location-independent—regardless of the physical location of any node in the network, the relative order in which they compute results, or the order in which those results are gathered, *the individual computations are the same, and the network output is the same*. Furthermore, because a node's information is local, adding or deleting nodes from the network can be done without affecting any other nodes. Thus, location-independence allows efficient support for dynamic topologies. Intuitively, the transformation is applied to the initial topology of the original network, before any learning has taken place; however, it can be viewed as being applied to some "snapshot" of the network, at any time, and the process is still the same.

In this paper, the term *Control Unit* refers to a mechanism that broadcasts inputs to a network, and gathers results from it. The term *original* refers to a network before it is transformed, and the term *transformed* refers to a network after it has been transformed. These terms apply similarly to the nodes as well. The number of layers refers to the number of *weight layers*.

LIT is a two-step process:

1. Construct a set of location-independent nodes based on the original model.
2. Embed the nodes in a tree.

This process is outlined in figure 1. Based on an original ANN model (left), a set of LI-nodes is constructed (middle), and the nodes are embedded in a tree (right).

LIT is not simply a reorganization of the original network, but rather involves redesigning the basic structure. A construction typically also involves reformulating the network equations, in order to more particularly describe the behavior of the transformed network. Although the original and transformed networks compute equivalent functions, correspondences between the two are not always direct, nor obvious. Details of a construction vary across different ANNs. An example transformation of an AA2 network to an equivalent LIA network is given in the first part of section 4, which specifically describes AA2 and the transformation to LIA.

A communication topology can be chosen as a matter of speed, cost and reliability. The same topology can be used with many models. A binary tree topology is specified in the second step in figures 1 and 2. It is a simple, fast, regular, hierarchical topology that naturally supports broadcast and gather operations. It provides a basis for applying the transformation and explaining the associated execution and learning algorithms, without obscuring the details with a complex interconnect. However, a hypercube, or some other more fault-tolerant topology, could also be used. Hierarchical topologies allow adding or deleting nodes with time logarithmic in the number of nodes in the network, while limiting connectivity among nodes. The connectivity of a binary tree does not explode because each node is connected to one parent and zero to two children. The algorithms for the transformed network show explicitly how the required communication can take place.

A specific example of how dynamic topologies are supported is included in the discussion of the transformation at the beginning of section 4, after general ASOCS definitions, concepts and goals have been discussed.

3. ASOCS DEFINITIONS

At a high level, the basic operational features of an ASOCS system are as follows:

- A ASOCS network is a network of nodes which, as a whole, stores an internal representation of the knowledge learned up to that point by the system.
- The network learns new information incrementally, as it is presented to the network, over time.
- The network adapts in a self-organizing fashion in response to new information—in other words, the network modifies the internal representation of knowledge in accordance with new information as it is received.
- During processing, or what is referred to in this paper as "execution mode", the network takes the full input environment (i.e. no undefined variables are allowed) and computes the appropriate output for the current state of those inputs, based on the previously learned information.

New information is presented to the network as an "instance", or a series of instances. The finite, parsimonious, consistent representation of learned knowledge stored in the network is called an "instance set". Where ASOCS models, such as AA1, AA2, AA3, LIA and PASOCS, differ is in how each one represents an instance set internally (in a network), and consequently, in the details of how each network learns or adapts. However, all ASOCS models implement the concepts of instances and instance sets in some fashion, as outlined above.

While this paper deals with the LIA model, this section discusses variables, instances, instances sets, learning and related issues in terms of general ASOCS concepts, independent of any particular model or implementation.

Assume a logical universe ***U*** composed of two disjoint sets of Boolean variables: 1) input variables and 2) one or more output variables. (Feedback variables are discussed in the ASOCS literature, but not in this paper.) For notational convenience, each variable u_i can be represented informally by a unique capital letter (i.e. A, B, C, etc.) The value of any particular variable at any given moment is called its *polarity*. Polarity is defined to be *positive* or *negative*, which corresponds to high or low, true or false, respectively. Positive variables are denoted by the variable name, while negative variables are denoted by the name followed with a " ' " (prime). Thus "A" means that variable A is positive, and " A' " means that variable A is negative. The values of the variables at any given moment collectively are called the *state* of the universe. The polarities of the input variables are independent, but the polarity of any output variables depends on the input variables.

An *instance* is a partial function from a set of input variables onto one output variable. It specifies what value the output variable must become when a given set of input variables is *matched*. Any non-negated variable is matched when the state is positive for that variable. Any negated variable is matched when the state is negative for that variable. Some state U *matches* an instance I iff all of I's input variables are matched. For two instances X and Y, X *covers* Y iff every state that matches Y also matches X (this does not necessarily imply that Y covers X). An instance that is **not** matched says *nothing about what value its output variable should have*. Furthermore, an instance that is matched only affects the value of its output variable, and no others. Following are examples of instances:

AB $\Rightarrow$ Z

AB'C $\Rightarrow$ X

C'Y $\Rightarrow$ Z'

AB'C $\Rightarrow$ Z

The variables on the left side are the input variables. The variables Z and X on the right side are output variables. The instance

AB $\Rightarrow$ Z

asserts that when A and B are positive, Z must be set to positive. The instance

C'Y $\Rightarrow$ Z'

asserts that when C is negative and Y is positive, then Z must be set to negative. Neither of these instances specify a value for Z when they are not matched.

Don't care variables **do not** appear on the left side of an instance, and have no effect on whether that instance asserts its output or not. For example, C and Y are "don't care" variables in the instance

AB $\Rightarrow$ Z.

If A and B are positive,

AB $\Rightarrow$ Z

asserts Z is positive regardless of the values of C and Y.

Using "don't care" variables means that a single instance can match a large number of states. For example, assume the input variables are {A,B,C,Y}. Of the 16 possible states for the input variables, the instance

AB $\Rightarrow$ Z

is matched by four. In general, if there are m input variables, then there are 2^m possible states. An instance whose left side has length n is matched by $2^{(m-n)}$ states, because of the don't care variables. When necessary, the value of don't care variables will be denoted with "*", although don't care variables are usually omitted when writing an instance.

A *discriminant variable for two instances* is an input variable that occurs negated in the input variables of one instance and non-negated in the input variables of the other. Two instances that contain a discriminant variable in their left sides are *discriminated* with respect to each other—the two instances are never simultaneously matched by any state. The variable B discriminates the first two of the instances

AB $\Rightarrow$ Z

B'CE $\Rightarrow$ Z'

C'HIJ $\Rightarrow$ Z

while the variable C discriminates the second and third instances. The first and third instances above, on the other hand, have no discriminant variable.

Learning in an ASOCS system revolves around the relationship between pairs of instances—essentially, a new instance, and (each of) the instances already stored in the network. Thus, in order to understand ASOCS learning, it is necessary first to understand the various ways one instance can be related to another and various associated ideas.

The relationship between two instances is determined by comparing the output variables of each instance to each other, and the corresponding sets of input variables to each other. There are three mutually exclusive relationships possible for output variables:

1. concordant
2. discordant
3. notrelated.

Two instances are *concordant* iff they have the same output variable and the same polarity for that output variable. Two instances are *discordant* iff they have the same output variable and different polarities for that variable. Two instances are *notrelated* iff they have different output variables. Consider the following four instances

AB $\Rightarrow$ Z

ADE $\Rightarrow$ Z

AB'F $\Rightarrow$ Z'

AB $\Rightarrow$ Y.

The first instance is concordant with the second one, discordant with the third one, and notrelated to the fourth one. In terms of affecting the network output, an instance always asserts a "don't know" value (denoted by ?) for any output other than its single asserted variable.

There are five mutually exclusive relationships possible between the (sets) of input variables of two instances:

1. subset
2. equal
3. superset
4. overlap
5. discriminated.

The first three terms *subset*, *superset*, and *equal* have meanings similar to those associated with sets, but two conventions need to be clarified. First, *subset* always means *proper subset*, and *superset* always means *proper superset*. Second, when comparing sets of variables, the values of variables are as important as which variables are in the sets. The two sets of variables, S_1 = {A, B, C'} and S_2 = {A, B, C}, are **not** equal. Even though they reference the same variables A, B, and C, S_1 contains C' while S_2 contains C. The same idea applies to the subset, superset, and overlap relationships.

The *overlap* relationship refers to the case in which each instance contains a variable which the other does not, and yet the two instances are not discriminated. In other words, the relative complement of each left-hand side with respect to the other is non-empty and disjoint (as opposed to discriminated).

The name "overlap" derives from comparing the input states covered by the respective instances rather than directly comparing the variables. Two "overlapping" instances are matched by some, but not all of the same potential input states, and each one is respectively matched by at least one state that the other is not. Hence, the instances overlap, but not completely (unlike subset, superset, or equal). Note that the states *must* overlap, since the two instances cannot contain the same input variable but with opposite polarity in each, by definition.

The instances

$AB' \Rightarrow Z$

$B'E \Rightarrow Z$

are overlap: The first contains A, but the second does not, while the second contains E but the first does not.

Consider the two inconsistent instances

$AB \Rightarrow Z$,

$C'Y \Rightarrow Z'$.

If input variables A, B, and Y are positive and C is negative, then the first instance asserts Z, while the second asserts Z'. This is inconsistent. If two instances are discordant, and if there exists a state that matches both instances, then the two instances are *inconsistent*. The following lemma provides a test to check if two instances are inconsistent [Mar91b]:

Inconsistency Lemma. *Two instances are inconsistent iff they are discordant and have no discriminant variable.*

This lemma can also be used to test for consistency between instances. Two instances are *consistent* iff no state exists such that the instances can simultaneously assert opposite outputs; this is always true if they have a discriminant variable or they are concordant. Note that instances that are *notrelated* are by default consistent, since they can never assert conflicting outputs for the same state.

The union of a finite collection of consistent instances defines a partial function of input variables onto output variables. It is the function defined by the set of instances, called the *Instance Set*, denoted by the symbol Ω. Any collection of instances containing the pair

$AB \Rightarrow Z$

$C'Y \Rightarrow Z$

would **not** be an instance set, because the instances are inconsistent. Instances below, on the other hand, are an instance set because each pair of instances is consistent:

$AB \Rightarrow Z$

$AB'C \Rightarrow Z$

$B'C'Y \Rightarrow Z'$

Instance sets are always consistent, by definition. Two instance sets Ω_1 and Ω_2 are *equivalent* iff they assert the same output values for every possible state, that is, their partial functions are the same.

New instances can be added to an instance set *incrementally* over time, that is, one at a time, possibly changing the instance set and thus the partial function that the instance set defines. When a new instance is added to an instance set, it is compared against each of the instances already in the instance set to make sure each instance is consistent with the new instance. The new instance that is being compared against the others is called the *new instance*, denoted by α. The others are called old instances, or instances previously added to the set, each one being an *old instance*, denoted by β, where a subscript such as j indicates the jth old instance in Ω. When resolving inconsistencies between α and old instances in Ω, precedence is given to α. *The goal of changing the instance set is to add the new instance and modify only those old instances that are inconsistent (change only the information that is inconsistent).* Thus, it is necessary to distinguish between the new instance and the old instances so that we know how to modify the instance set. An algorithm for modifying instance sets is given in section 5.

In the process of adding new instances to an instance set, redundancies of various types may occur. For example, suppose we want to add the instance

$B \Rightarrow Z$

to the following instance set (for brevity only related instances are shown):

1. $AB \Rightarrow Z$
2. $AB'C \Rightarrow Z$
3. $B'C'Y \Rightarrow Z'$

The instance

$A \Rightarrow Z$

is consistent with all of the old instances, so it could be added to the instance set as is. But the new instance is a concordant subset in relation to instance 1, and discriminated from instances 2 and 3. The new instance matches all the same states as instance 1 and more, hence instance 1 becomes redundant. Instance 1 is deleted from the network and the new instance is added.

The process of detecting and removing redundancies in an instance set is called *minimization*. There are many types of minimization. Minimization does not affect consistency—it is desirable because it helps keep the size of an instance set as small as possible. It also permits learning generalization in that special-case instances are replaced by more general instances. Some types of minimization can be done very quickly, while others take a long time or require complex use of resources. Instance set modification can be kept fast by compromising on parsimony. Fast and easy types of minimizations can be done at the same time as modifications for consistency. Slower types of minimizations can be done periodically as "background" processes, during times when the machine is not actively executing or learning. This description of the LIA model includes algorithms for doing the fast minimizations as part of "normal" learning. Section 5.3 discusses some other minimizations.

To summarize and tie together the concepts discussed above:

- An ASOCS network (including LIA) learns new instances incrementally, as they are presented to the network, over time.
- The network adapts in a self-organizing fashion to represent a modified instance set, which is consistent with the new instance being added and parsimonious, whenever a new instance causes modifications to be necessary.
- During execution mode, the network takes the full input environment and computes the appropriate output for those inputs, based on the instances stored in the learned instance set.

4. TRANSFORMATION OF AA2 NETWORKS TO LIA NETWORKS

Figure 2 shows an example of an AA2 [Mar91b] network transformed to an equivalent LIA network. The numbers accompanying particular nodes are for reference only. The

AA2 network is composed of two types of nodes, 1) PNodes, and 2) DNodes. PNodes are the interior nodes in the network, the building blocks from which representations of instances are built in a hierarchical fashion. Each PNode represents the conjunction of its two inputs (represented by the letters inside each node in figure 2). The two inputs to a PNode may be either of two types: an input variable (or its negation), or the output of another PNode. The output of a PNode is Boolean, and the output of any given PNode may be connected (and thus shared) as input to one or more other PNodes or DNodes.

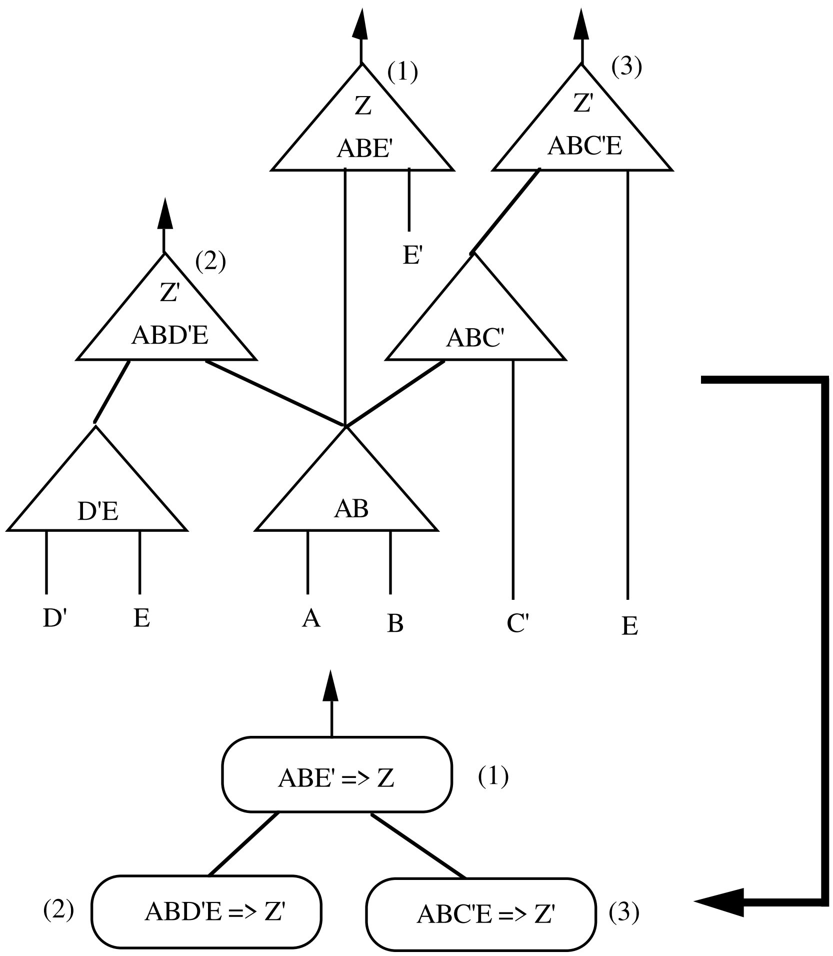


Figure 2. An Example AA2 network transformed to LIA

DNodes are the leafnodes in an AA2 network (i.e. nodes 1, 2 and 3). Like a PNode, each DNode represents the conjunction of its inputs, and the two inputs of a DNode may be either a variable (or its negation) or the output of a PNode. In addition, however, a DNode also represents one of the instances in the instance set currently stored in the network. In otherwords, there is one DNode for each instance stored in the network, and each Dnode determines the output of the network when the instance it represents is matched. The output value of the network for a DNode is indicated by the Z or Z', respectively, at the top of each Dnode in figure 2. The output of a DNode is Boolean—the actual output of the network is determined by whether the output of the DNode is connected to the positive collector (think of each collector as an OR-plane) or negative collector of the output variable. Z therefore represents the positive collector for output variable Z, while Z' represents the negative collector for variable Z.

This brief overview of AA2 concludes with two further observations: First, a network outputs a ? ("don't know" value) in the case where none of the instnces is matched, and therefore every DNode outputs a 0, and therefore both the positive and negative collectors ouptut a 0. Second, the instance set stored in a network is changed by adding and deleting PNodes and Dnodes, and by altering connections between PNodes, Dnodes and the connections to output collectors.

There is one node in the LIA network for each Dnode in the AA2 network. Each transformed node stores the instance represented by the corresponding AA2 Dnode (as shown in figure 2). Another way of looking at the transformation is this: A transformed node can store a complete instance; each node in the transformed network stores one instance in the instance set represented by the AA2 network. So, LIA can be seen either as a LIT transformation (implementation) of AA2, or as a different way of representing an instance set, whichever is convenient.

Suppose that a new instance

AB'F $\Rightarrow$ Z'

is to be added to the AA2 network of figure 2. None of the possible conjunctions AB', AF or B'F already exist as PNodes, so a new PNode representing one of them, AB' for example, would be added to the network. In that case, a new DNode with inputs from the new PNode and the variable F would also be added to the network. The output of the DNode would be connected to the negative collector for Z.

The same new instance is added to the LIA network of figure 2 by allocating a free node in the network, the left child of node 2 for example, and storing the new instance in that node.

Now suppose the instance

ABD'E $\Rightarrow$ Z'

is to be deleted from the AA2 network of figure 2. DNode 2 is deleted, along with its connections to the negative collector and any PNodes or variables to which it is directly connected—in this case PNodes representing D'E and AB. In addition, since the PNode representing D'E is not shared by any remaining instances, it is deleted from the network.

The same instance is deleted from the LIA network by marking node 2 as "free". No other nodes in the LIA network are affected.

A 2-input AA2 node (either Pnode or Dnode) may or may not actually store its variable list(s) at each node, depending on the specific implementation. An LIA node, on the other hand, stores two lists, or vectors of variables: One is the left-hand side of an instance $\boldsymbol{V_j}$. Each element v_{ij} has the value *, or the value of the corresponding variable of the instance that the node stores. The other vector is the right-hand side of an instance $\boldsymbol{R_j}$. It has ? values in every element r_{kj} except the one corresponding to the output variable for that instance, in which case the value is 1 or 0, depending on the polarity of that variable. Even though each instance only has one right-hand side variable, it is easier to visualize (and perhaps handle) how multiple output variables are gathered in parallel if the output of each node is a vector rather than a scalar. Figure 3 shows a conceptual view of an LIA node.

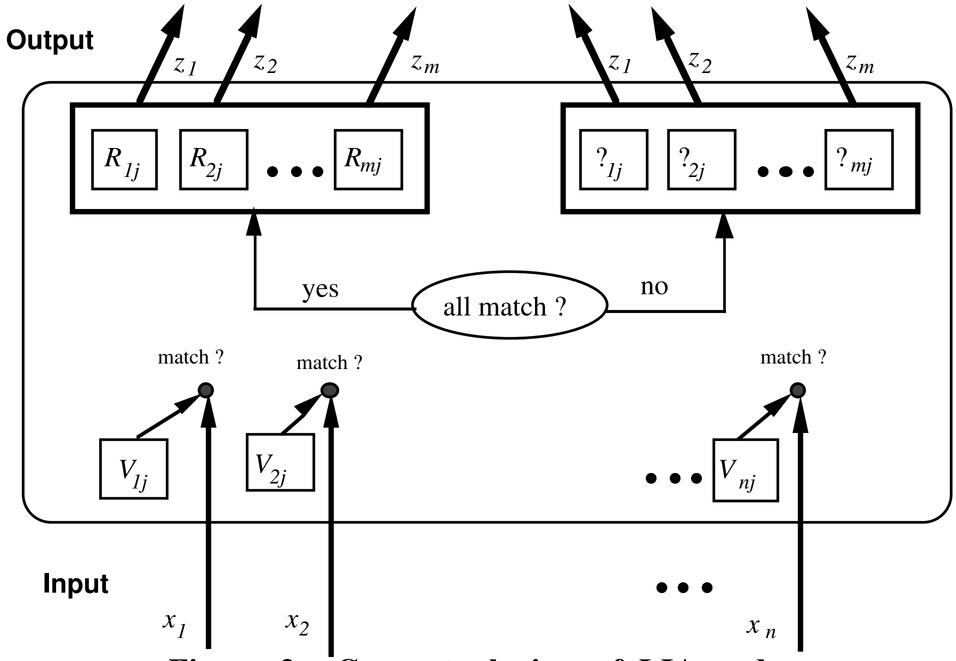


Figure 3. Conceptual view of LIA node.

4.1. LIA Execution Mode

The goal of execution mode is to determine what is the output of the network for a given set of inputs. More precisely, it is to determine which nodes in Ω match a given state or set of input values $\boldsymbol{X}$, and thus determine the output of the network. This is done in two steps, broadcast and gather. During the broadcast step, $\boldsymbol{X}$ is broadcast down the tree to the nodes and each node j compares each element x_i of $\boldsymbol{X}$ with the corresponding element v_{ij} of $\boldsymbol{V_j}$ (the left-hand-side of β_j the instance stored at node j) to see if the two vectors "match". In this matching function, a 1 matches a 1, a 0 matches a 0, and * ("don't care") matches either 1 or 0. The domain of x_i is Boolean, while the domain of v_{ij} is Boolean and *. A node that is matched will assert the right-hand side $\boldsymbol{R_j}$ of its β_j as output $\boldsymbol{O_j}$, whereas a node that is not matched (there exists at least one mismatch) will assert a vector of ? ("don't know") values as output $\boldsymbol{O_j}$. The latter is equivalent to the node itself not responding to some set of input values.

During the gather step, each node combines the elements of its asserted output $\boldsymbol{O_j}$ with the corresponding values received from its children, and passes the result $\boldsymbol{Y_j}$ up to its parent. The parent of the root node is the CU, and thus $\boldsymbol{Y_j}$ from the root node becomes the output of the network $\boldsymbol{Z}$.

The underlying purpose of the gather function is one of transmission—transmitting output values from the nodes to the CU. Thus, the general sense of the gather function at each node is that of an *OR*, namely *OR*ing together the respective output elements from all the nodes to determine a single value for each output variable, and the $\vee$ symbol is used to denote this operation. However, the semantics of the gather function are somewhat different from *OR* as it is generally understood, even beyond the fact that the gather function here is extended to handle ? values. These semantics can be explained in terms of a dyadic operation using one output variable o_{kj} and the corresponding output variable from one child y_{kjh}, however the operation extends to any number of varibles (or children). The gather operation is defined as follows:

okj	*ykj*	result
?	?	?
?	0	0
?	1	1
0	?	0
0	0	0
0	1	impossible case
1	?	1
1	0	impossible case
1	1	1

Table 1. Definition of gather operation

The idea is that the indefinite value ? is replaced by a definite 1 or 0 (as the case may be) when such is asserted as the output of the network for a variable. Note that although more than one node may match a given ***X***, the nodes will never have to combine the cases $0 \vee 1$ or $1 \vee 0$ for any variable, because the network is consistent—if any one node asserts a definite value for some output variable, no other node will contradict that value. (In a physical implementation, such a case might occur only if one or more nodes has been corrupted, which results in a system error in any case.)

Three cases can occur for execution mode for any single variable, given some input vector ***X*** and some network that stores instance set Ω: 1) no instances match, 2) exactly one instance matches, 3) more than one instance matches. In the first case, the network will output a vector of ? values. In the second case, the output of the network is clearly the output of that matched instance for that variable, since there is only one instance. The third case arises from the fact that two or more concordant instances may match the same ***X***, even though Ω is minimized, because such instances may overlap in their coverage of states.

In the AA2 literature, an AA2 node is represented as a 2-input AND-gate with a node CU controlling the inputs and output of the node. In execution mode, the network can be viewed essentially as a hired-wired hierarchy of 2-input AND-gates whose outputs flow into an OR-plane (the collectors). Likewise, a LIA node can be viewed as an n-input AND-gate with a node CU controlling the inputs and output(s) of the node. These views are conceptually correct, intending primarily to show how the nodes implement conjunctions of variables and the required mapping of inputs to outputs. However, Stout *et. al.* found these high-level views to be too simplistic and somewhat misleading, when attempting to design actual implementations [Sto94b]. The critical issue is that such a view ignores the control circuitry (switches, memory, etc.) needed to "set up" or "preprocess" and maintain the inputs to the AND-gates. Considering the functionality required by the overall AA2 system (in which the network is embedded) and the local CU at each node to compute with instances in an AA2 network, the true overall function is more akin to the equivalence (XNOR) function. The "matching" function used in LIA and PASOCS models is essentially an equivalence function extended to handle * values in the instances. This view was more convenient for the VLSI implementation of PASOCS in [Sto94b].

4.2. LIA Learning Mode

Let Ω_0 be the null, or empty instance set, which corresponds to a network that stores no instances. Let α_t be the *t*-th new instance added to a network which stores instance set

Ω_{t-1} before α_t is presented for learning, and stores Ω_t after learning α_t. For puposes of the discussing modifications that occur during learning, consider the initial value of Ω_t to be Ω_{t-1}—all potential modifications are discussed with reference to Ω_t, rather than Ω_{t-1}.

The goal of learning is to add α_t in such a way that Ω_t includes the mapping specified by α_t, is consistent with α_t, and parsimonious. The behavior of the network in learning mode is motivated by the search for β_j that are inconsistent with α_t the search for β_j that can be involved in minimizing Ω_t (assuming that α_t is added to Ω_t) and necessary adaptive actions of the network based on the results of these searches.

These ideas can be summarized by the following semi-formal definition of instance set learning:

$$\Omega_0 = \varnothing$$

$$\Omega_t = \begin{cases} \Omega_t, \textit{ iff } \exists \supseteq \beta_j \textit{ that covers } \alpha_t \\ (\Omega_{t-1} \cup \alpha_t \cup A) - B - C, \textit{ otherwise} \end{cases}$$

where

$A = \{$*all inconsistent* β_j *made consistent*$\}$,

$B = \{$*all redundant* $\beta_j\}$,

$C = \{$*all inconsistent* $\beta_j\}$.

The searches and resulting actions are based around the fact that α_t, partitions Ω_t into disjoint subsets based on the relationship of each β_j to α_t. The right-hand side of any β_j must be Concordant, Discordant, or Notrelated with respect to the right-hand side of α_t. By definition, each of these subsets of Ω_t is mutually exclusive. The left-hand side of any β_j must be Subset, Equal, Superset, Overlap, or Discriminated with respect to the left-hand side of α_t. By definition, each of these subsets of Ω_t is also mutually exclusive. Thus, the combination of left-hand side and right-hand side relationships results in a partition.

There are 15 subsets in the partition of Ω_t, but only seven of these cause adaptive actions and are therefore of interest during learning. The eight which do not are those which contain any β_j that is Notrelated, is Discriminated, or is Concordant Overlap. With the exception of Notrelated, which should be obvious, the others are discussed in more detail in [Mar91b] and other ASOCS literature. Of the remaining seven subsets: four would contain any possible inconsistent β_j, those being Discordant Subset, Equal, Superset or Overlap; three would contain any instance that could possibly be used to minimize Ω_t, those being Concordant Subset, Equal or Superset. Note that minimization as it is used here refers to simple, fast, pairwise minimizations only. Other minimizations are briefly discussed in section 5.3.

The adaptive actions for each of these seven subsets are as follows:

Concordant Subset or Equal: If either of these partitions are non-empty, then there exists at least one β_j that covers all possible input states that α_t covers, and possibly more, and has the same output as α_t. Consequently, adding α_t to Ω_t would be redundant. Furthermore, if this is true, all other partitions are empty, otherwise Ω_t is not consistent and parsimonious to begin with. Hence α_t is not added, and learning terminates for this instance.

Concordant Superset: If this partition is non-empty, then there exists at least one β_j which is redundant with α_t because α_t covers all the states that β_j covers and more, and they have the same output. Any such β_j can self-delete, and α_t is added to Ω_t.

Discordant Equal or Superset: If either of these partitions are not empty, then there exists at least one β_j which completely contradicts α_t because α_t covers all the states that β_j covers and more, and they have opposite output. The contradiction can be properly resolved by removing β_j. Thus, any such β_j can self-delete, and α_t is added to Ω_t.

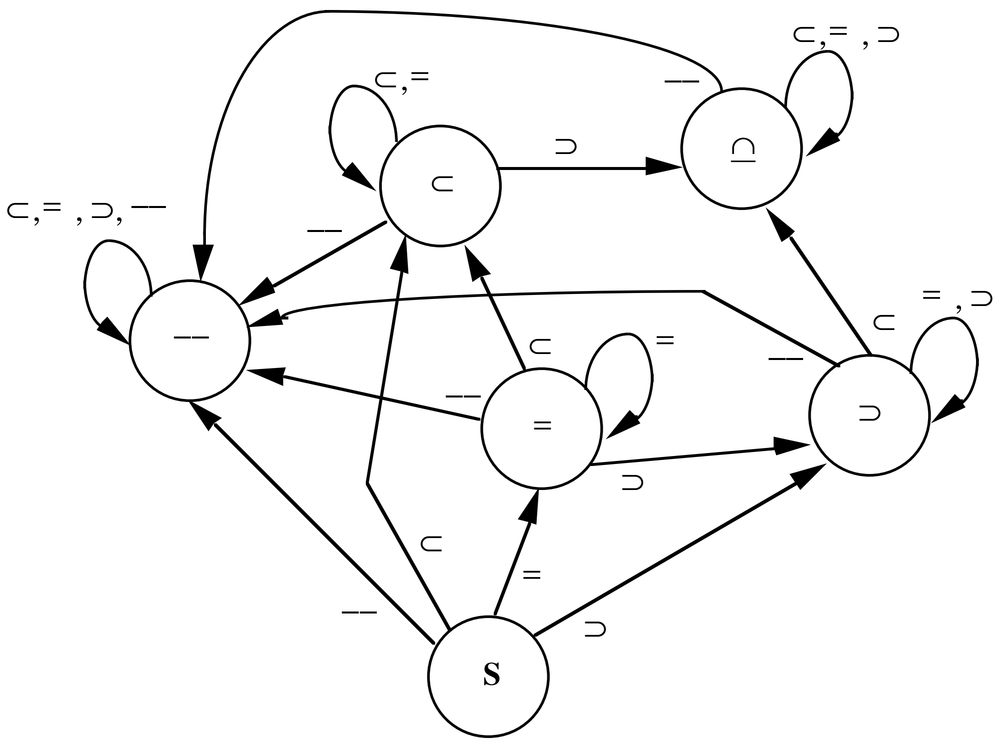
Discordant Subset or Overlap: If either of these partitions are not empty, then there exists at least one β_j which is partially contradicted by α_t: Either β_j covers all the states that α_t covers and more, or β_j covers some states that α_t covers and some states that α_t does not, and they have opposite output. The contradiction can be properly resolved by specializing each contradictory β_j so that it no longer covers contradictory states, but still covers all non-contradicted states as before. This specialization process is called *Discriminant Variable Addition* (DVA).

By definition, inconsistencies exist between α_t and β_j because their left-hand sides contain no discriminant variable(s). The most logical way to specialize β_j with discriminant variables is to find all the variables in $\boldsymbol{V}$ (the left-hand side of α_t) that are not in $\boldsymbol{V_j}$, invert their values (so they discriminate from α_t), add each one to the left-hand side of a copy of the original β_j, delete the original β_j, and add these modified instances to Ω_t. α_t is also added to Ω_t.

The following list summarizes the relationships that cause adaptive actions and their associated actions:

Relationship	Action
Concordant Subset	exit (already learned)
Concordant Equal	exit (already learned)
Concordant Superset	node self-deletes
Discordant Superset	node self-deletes
Discordant Equal	node self-deletes
Discordant Subset	DVA
Discordant Overlap	DVA

Learning assumes that the instance set Ω_t is consistent and parsimonious before α_t is presented for learning. As α_t is broadcast to the nodes, each node receives the broadcast, passes it on to its children (if any) and computes its relationship to α_t. Figure 4 shows a sequential state-flow diagram that indicates how the relationship between the left-hand sides of two instances can be determined. The diagram uses only single-variable comparisons, one comparison per transition. The comparison can also be done in parallel blocks depending on the width of the data path, as in [Sto94b].



S is the starting state; all other symbols represent states and /or single-variable comparisons as follows:

⊂ is subset
= is equal
⊃ is superset
$\underline{\cap}$ is overlap
–– is discriminated

Figure 4. A sequential view of how the relationship between two instances can be determined using only single variable comparisons.

When all nodes have determined the relationships, the CU begins checking for the adaptive actions as outlined above. The CU broadcasts a "covered" flag, for which each node computes a 1 if it covers α_t (the node's β_j is subset or equal) and is concordant, and a 0 otherwise. Since the result is Boolean, the results can be gathered at each node using the Boolean OR, and the result passed up to each node's parent. If the CU receives a 1, then there exists at least one node (there may be relatively minimal overlapping nodes that respond) which properly covers α_t. In other words, α_t has already been learned. Since Ω_t is already consistent and parsimonious, no further modification is needed to generate Ω_t, therefore the network can proceed to the next instance. If the algorithm does not exit at this point, then α_t will be added to Ω_t once any and all other adaptive actions have been completed.

If α_t has not been learned, the next step is to check for nodes that can self-delete. The CU broadcasts the self-deletion flag. Any node that can self-delete (i.e. any superset instance and any equal discordant instance) does so, and returns a 1, while all others return a 0. As before, each node combines (Boolean OR) its result with those of its children, and passes the result up to its parent. The result of this particular gather is not necessarily needed, except to indicate whether any node(s) in fact self-deleted. The reason for doing self-deletion at this point in the algorithm is that any nodes which self-delete may potentially be re-allocated during DVA, if necessary. This makes more efficient use of the nodes. This differs from the AA2 algorithm of [Mar91b]. There, self-deletion takes place *after* DVA, in order to take advantage of any pre-existing Pnode hierarchies for node combination in adding properly discriminated instances to the network. Note that self-deletion effectively deletes redundant instances and totally contradicted instances.

The next step is to check for partially contradicted instances (i.e. subset or overlap discordant instances), and execute DVA if needed, as outlined above. This is accomplished as follows: The CU broadcasts the DVA flag. Each partially contradicted node returns a 1, while all others return a 0. If the CU receives a 1 from the gather of these results, it then broadcasts a flag which permits the required nodes to execute DVA. Each such node j now computes the list of discriminant variables S from α_t, one of which it adds to its left-hand side $\boldsymbol{V_j}$ (this is equivalent to deleting the original β_j and replacing it with one of the specialized instances), deleting it from S. If S is not empty after this, the node splits S in half and sends a copy of the original β_j with half of S to the right, and a copy of the original β_j with the other half of S to the left (if S is odd, the "extra" variable can be included in either the left or right half of S). If a child node is *not* empty when it receives this DVA list, it will split the S it receives and simply pass these on, with copies of the original β_j, as did the original parent. If however, a child is free, it will allocate itself, store the original β_j, and add one variable from the S it receives to its left-hand side, deleting that variable from S. The node then proceeds with the original β_j and its remaining S as did the original parent. DVA terminates at the node where there are no more variables to send. DVA terminates for the network when all instances created by DVA have been stored in a node.

DVA can be executed in parallel [with complexity $O(\log m)$, where m is the number of nodes in the network] in the tree because, for any two nodes at the same level of the tree, DVA lists are sent down mutually disjoint subtrees. This remains true as long as it is assumed that there are always enough nodes in the branches of the subtree of the original contradictory node to store instances required by DVA. If there are not enough nodes in a subtree, then either DVA cannot be completed until more nodes are added where required, or the DVA lists must be passed back up the tree and rerouted down subtrees where free nodes exist.

There are several methods for handling the case where there are insufficient nodes in a particular subtree (but sufficient nodes in the network). Sequentially selecting and adding each remaining instance has complexity $O(k\log m)$, where m is the number of nodes in the network and k is the number of instances remaining to be added as a result of DVA. Other, potentially more parallel and more localized rerouting algorithms could be used instead of the sequential method. Such parallel methods would seem to have faster average times (than the sequential method), but also risk a worst case search that is linear in the number of nodes in the tree.

Once contradictions have been resolved (if there were any), a free node is selected from any remaining free nodes in the network, and α_t is stored in it. The learning of α_t is complete, and the network may proceed to the next instance.

5. FORMAL EXECUTION AND LEARNING ALGORITHMS

5.1 Execution Algorithm

Definitions

= used either as assignment operator or as the equivalence operator, as appropriate [see equation (4)].

* "don't care" value. Only occurs in the left-hand-side V_j of an instance.

? "don't know" value. Only occurs in the right-hand side of an instance $\boldsymbol{R_j}$, and thus possibly in $\boldsymbol{O_j}$, $\boldsymbol{Y_j}$ and $\boldsymbol{Z}$.

n number of currently defined inputs to the network, also the size of $\boldsymbol{X}$. If "don't care" variables are counted in each node, it is also the size of $\boldsymbol{V}$.

p number of currently defined outputs of the network, also the size of $\boldsymbol{R_j}$, $\boldsymbol{O_j}$, $\boldsymbol{Y}$ and $\boldsymbol{Z}$, if "don't know" variables are counted. For most ASOCS descriptions, p=1. In this paper, p≥1.

w_j the number of children of node j.

$\boldsymbol{X}$ input pattern presented to the network.

x_i ith Boolean component of $\boldsymbol{X}$.

$\boldsymbol{V_j}$ left-hand-side of the instance stored in node j, has size n if don't care values are counted.

v_{ij} ith component of $\boldsymbol{V_j}$. It has domain {0,1,*}.

$\boldsymbol{R_j}$ right-hand-side of the instance stored in node j, has size p if don't know values are counted.

r_{kj} kth component of $\boldsymbol{R_j}$. It is also the output of node j with respect to output variable z_k. It has domain {0,1,?}.

$\boldsymbol{O_j}$ individual output vector of node j—it is either $\boldsymbol{R_j}$ or "don't know", depending on the value of $match_j$.

o_{kj} kth component of $\boldsymbol{O_j}$. It has domain {0,1,?}.

m number of D-nodes/instances in the original AA2 network and the number of nodes in the LIA network.

$match_j$ Boolean value for node j that is 1 if $\boldsymbol{V_j}$ covers $\boldsymbol{X}$, and 0 otherwise.

$\boldsymbol{Y_j}$ vector of intermediate network output values computed by node j.

y_{kj} kth component of $\boldsymbol{Y}$. It has domain {0,1,?}.

$\boldsymbol{Z}$ vector of network output values.

z_k real-valued (always between 0 and 1) kth component of $\boldsymbol{Z}$. It corresponds in the original network to the output O_k of output node k.

Equations

$$y_{kj} = o_{kj} \vee \left(\bigvee_{h=1}^{w_j} y_{kjh} \right) \qquad (1)$$

$$o_{kj} = ? \vee (match_j \wedge r_{ij}) \qquad (2)$$

$$match_j = \bigwedge_{i=1}^{n} [(v_{ij} = *) \vee (v_{ij} = x_i)] \qquad (3)$$

The steps of the execution algorithm are as follows:

1. Broadcast $\boldsymbol{X}$ to the nodes.
2. Each node does the following:
   2.1 Compute $\boldsymbol{O_j}$.
      2.1.1 Determine if $\boldsymbol{V_j}$ matches $\boldsymbol{X}$.
      2.1.2 $match_j$ is 1 if all variables match, 0 otherwise.
      2.1.3 The node contributes $\boldsymbol{R_j}$ to the output of the network if and only if $match_j$ is 1.
   2.2 Compute $\boldsymbol{Y_j}$.
   2.3 Send $\boldsymbol{Y_j}$ to the parent node.
3. $\boldsymbol{Y_j}$ from the root node is the output $\boldsymbol{Z}$ of the network.

5.2 Learning Algorithm

Additional Definitions

α_t the t-th new instance presented for learning.

$\boldsymbol{V}$ the left-hand side of α_t.

$\boldsymbol{R}$ the right-hand side of α_t.

r_k	the kth component of $\boldsymbol{R}$, corresponding to z_k.
r_{kj}	the kth component of $\boldsymbol{R_j}$.
β_j	the instance stored at a node j, or "old instance". It has left-hand side $\boldsymbol{V_j}$ and right-hand side $\boldsymbol{R_j}$.
$\subset$	proper subset
$=$	equal sets
$\supset$	proper superset
$\underline{\cap}$	overlap
$--$	discriminated
$covered_{kj}$	flag computed at node j relative to α_t with right hand side z_k, which indicates whether node j covers α_t; it is 1 if so, 0 otherwise.
$delete_{kj}$	flag computed at node j relative to α_t with right hand side z_k, which indicates whether node j should self-delete; it is 1 if so, 0 otherwise.
dva_{kj}	flag computed at node j relative to α_t with right hand side z_k, which indicates whether node j should execute DVA; it is 1 if so, 0 otherwise.
S	the set of variables in $\boldsymbol{V}$ which are not a node's $\boldsymbol{V_j}$; the variables in S are computed and used only for DVA.
$rconc_{kj}$	an intermediate Boolean variable used to make the equations below more readable; it is 1 iff β_j is related to α_t for output variable z_k and they are concordant—otherwise, it is 0. (See equation 4.)
$rdisc_{kj}$	an intermediate Boolean variable used to make the equations below more readable; it is 1 iff β_j is related to α_t for output variable z_k and they are discordant—otherwise, it is 0. (See equation 5.)

Equations

$$rconc_{kj} = (r_{kj} \neq ?) \wedge (r_{kj} = r_k) \tag{4}$$

$$rdisc_{kj} = (r_{kj} \neq ?) \wedge (r_{kj} \neq r_k) \tag{5}$$

$$\text{covered}_{kj} = \begin{cases} 1, \text{ if } \text{rconc}_{kj} \wedge (\mathbf{V_j} \subset \mathbf{V}) \\ 0, \text{ otherwise} \end{cases} \tag{6}$$

$$delete_{kj} = \begin{cases} 1, \text{ if } (r_{kj} \neq ?) \wedge (\boldsymbol{V_j} \supset \boldsymbol{V}) \\ 1, \text{ if } rdisc_{kj} \wedge (\boldsymbol{V_j} = \boldsymbol{V}) \\ 0, \text{ otherwise} \end{cases} \tag{7}$$

$$dva_{kj} = \begin{cases} 1, \text{ if } rdisc_{kj} \wedge [(\boldsymbol{V_j} \subset \boldsymbol{V}) \vee (\boldsymbol{V_j} \underline{\cap} \boldsymbol{V})] \\ 0, \text{ otherwise} \end{cases} \tag{8}$$

The steps of the learning algorithm are given below.

1. α_t is broadcast to the network.
2. Each node computes its relationship to α_t.
3. Determine if α_t is already covered by Ω_t.

3.1 The CU broadcasts the covered flag.
3.2 Each node j returns a 1 if β_j is *Concordant Equal*, or *Concordant Subset*, a 0 otherwise.
3.3 If the CU receives a 1 from the network, exit.

4. Delete redundant and wholly contradicted information.
4.1 The CU broadcasts the self-delete flag.
4.2 Any node whose β_j is *Discordant Equal*, *Concordant Superset* or *Discordant Superset* self-deletes.

5. Resolve partial contradictions, if any.
5.1 The CU broadcasts the DVA flag.
5.2 Each node j returns a 1 if β_j is *Discordant Subset* or *Discordant Overlap*, and returns 0 otherwise.
5.3 If the CU receives a 0, skip to step 6.
5.4 Each node j that returns a 1 does the following:
5.4.1 Compute the set difference $S = \boldsymbol{V} - \boldsymbol{V_j}$
5.4.2 Choose a v in S, and delete it from S.
5.4.3 Add $\sim v$ to the original $\boldsymbol{V_j}$.
5.4.4 For each remaining v in S do the following:
a. Choose a v and delete it from S.
b. Allocate a free node in the network.
c. Make the new node's variable list the original $\boldsymbol{V_j}$ concatenated with $\sim v$.
d. The new node's R is the same as $\boldsymbol{R_j}$.
end

6. Add α_t to the network.
6.1 Allocate a free node.
6.2 Store α_t in the new node.

end (learning)

5.3 Optional Network Minimizations

As in [Mar91b], the learning algorithm above omits optional minimization procedures that may lead to a smaller network. These procedures are optional because they do not affect the correctness, or consistency, of the network, but rather the size in terms of the number of defined input variables and the number of instances in the network. Furthermore, minimization requires rebroadcasting modified instances to the network, which adds to the complexity of learning and potentially causes further minimizations, affecting the speed of learning. This section provides details similar to those of sections 4.2 and 5.2 for readers who may be interested—further details can be found in the ASOCS literature. This section begins with definitions for one type of pairwise minimizations based on certain one-difference concordant instances, which were defined for the original ASOCS models. The definition of the term "one-difference" differs here and in [Sto94b] from previous ASOCS usage in slight, but fairly significant, ways. Following this, the idea of "background" minimization is briefly discussed as an alternative to minimization during active learning.

β_j is *one-difference* with respect to α_t iff $\boldsymbol{V_j}$ and $\boldsymbol{V}$ share exactly one discriminant variable δ_{kj}. For example, ABC' $\Rightarrow$ Z is one-difference with respect to ABC $\Rightarrow$ Z. These instances can be replaced with the single, shorter instance AB $\Rightarrow$ Z. Likewise, AB'C $\Rightarrow$ Z is one-difference with respect to AB $\Rightarrow$ Z. These two can be replaced by the pair AC $\Rightarrow$ Z and AB $\Rightarrow$ Z.

Relating this back to the discussion of section 4.2, the one-difference relationship is a subset of the discriminated left-hand side relationship/partition. To be precise, the previous discriminated relationship can be divided into two mutually exclusive relationships, one-difference (denoted by 1d-) and two-difference (denoted by 2d-). As before, none of the 2d- relationships cause adaptive actions (and hence minimization), and so are not of interest here. The 1d- relationship, however, may be divided into four mutually exclusive subsets: 1d-subset, 1d-equal, 1d-superset and 1d-overlap. While this adds 12 subsets to Ω_t (because they can be concordant, discordant, or notrelated on the right-hand side), only three of these subsets cause minimization: concordant 1d-subset, concordant 1d-equal and concordant 1d-superset. Of the other nine, four are notrelated, four are discordant discriminated and one, the concordant 1d-overlap, also cannot be pairwise minimized with any other instances.

Note that both 1d-overlap concordant and discordant relationships never induce adaptive actions, just like the 2d- instances. Therefore, there is no real point to distinguishing the 1d-overlap relationship from the 2d- ones. The state-flow diagram of figure 4, which can be modified to include 1d- relationships, helps illustrate this idea. From the overlap state, the "machine" can move directly to the discriminated state whenever a discriminated input comparison is encountered, as is currently shown—hence, an extra state for 1d-overlap relationships is not needed.

Concordant 1d-equal: If this partition is non-empty, then there exists at least one β_j such that the set of states covered by ($\beta_j \wedge \alpha_t$) is also covered by α_t with $\sim\delta_{ikj}$ dropped from $\boldsymbol{V}$. α_t with $\sim\delta_{ikj}$ dropped from $\boldsymbol{V}$ is rebroadcast to the network as if it were the original instance—this causes the now redundant β_j to self-delete, and allows any other potential minimizations to take place.

Concordant 1d-subset: If this partition is non-empty, then there exists at least one β_j such that the following is true: Choosing exactly one such β_j, there exists a subset P of the states covered by β_j for which the output of the network is the same as α_t and for which there is a state covered by α_t that differs only in the value of δ_{ikj}. For the states in P and their counterparts, the discriminant variable is therefore not needed. α_t with $\sim\delta_{ikj}$ dropped from $\boldsymbol{V}$ covers the same states as α_t and additionally the states in P. α_t with $\sim\delta_{ikj}$ dropped is rebroadcast to the network as if it were the original instance—β_j is now concordant overlap with respect to α_t, but any further potential minimizations can take place.

Concordant 1d-superset: This case is analogous to the 1d-subset case, but with α_t and β_j reversed. If this partition is non-empty, then there exists at least one β_j such that the following is true: For each β_j , there exists δ_{ikj} and a respective subset P of the states covered by β_j for which the output of the network is the same as α_t and for which there is a state covered by α_t that differs only in the value of δ_{ikj}. For the states in each P and their counterparts, the discriminant variable is therefore not needed. β_j with δ_{ikj} dropped from $\boldsymbol{V_j}$ covers the same states as β_j and additionally the states in P. Each modified β_j with its δ_{ikj} dropped is removed from Ω_t and rebroadcast to the network, potentially causing further minimizations. (α_t is not modified when it is added to the network as the new instance in this case.)

The following summarizes the 1d- relationships and their associated adaptive actions:

Relationship	Action

Concordant 1d-subset, or

Concordant 1d-equal — if any such nodes,
pick exactly one such β_j (*not* one from both partitions!)
delete $\sim\delta_{ikj}$ from $\boldsymbol{V}$
rebroadcast the modified α_t to the network

Concordant 1d-superset — for all such nodes,
delete $\sim\delta_{ikj}$ from $\boldsymbol{V_j}$
remove β_j from Ω_t
rebroadcast the modified β_j to the network

<u>minimization definitions</u>

$-\subset$	one-difference (proper) subset.
-=	one-difference equal.
$-\supset$	one-difference (proper) superset.
δ_{ikj}	one-difference variable v_{ij} in β_j relative to α_t with right-hand side z_k.
$min\_ni\_var_{ikj}$	flag computed at node j relative to α_t with right hand side z_k; it is the one-difference variable δ_{ikj} if α_t can be minimized by dropping$\sim\delta_{ikj}$, ? otherwise.
min_is_{ikj}	flag computed at node j relative to α_t with right hand side z_k; it is 1 if β_j can be minimized by dropping δ_{ikj}, 0 otherwise.

<u>minimization equations</u>

$$min\_ni\_var_{kj} = \begin{cases} \delta_{ikj}\,, \; if \; rconc \wedge ((\boldsymbol{V_j}\text{-}\subset \boldsymbol{V}) \vee (\boldsymbol{V_j}\text{-=}\boldsymbol{V})) \\ ?, \; otherwise \end{cases} \tag{9}$$

$$min\_is_{kj} = \begin{cases} 1, \; if \; rconc \wedge (\boldsymbol{V_j} \text{-}\supset \boldsymbol{V}) \\ 0, \; otherwise \end{cases} \tag{10}$$

Background minimization is the idea of delaying minimizations requiring rebroadcast of modified instances to times when the network is not actively engaged in either execution or learning. This is possible, again, because minimization can take place independently of checks for consistency and at any time. This allows for rapid learning, while also taking advantage of time when the network might be otherwise idle. One might consider that minimization during active learning can significantly reduce the size of the network, thus increasing the speed of learning. However, considering that with a hierarchical topology such as a tree, the growth (size) of the tree is logarithmic, and the processing time is relatively small even for very large networks.

Pairwise minimizations are fast approximations to finding globally minimal representations. The use of background minimizations suggests the possibility of implementing *m-way* minimizations (use of *m* instances or higher-order combinations of up to *m* variables) to achieve more minimal instance set representations. This idea is the subject of future research, perhaps in connection with on-going research in the area of generalization algorithms.

6. COMPLEXITY AND PERFORMANCE

The complexity of the algorithms in section 5 can be characterized by pipelined broadcast and gather in a tree of nodes. The width of a broadcast block is assumed to be the size of a single variable. The tree has *m* nodes, one for each instance in Ω_t, and the nodes operate in parallel. Assume that each node has *n* inputs and *p* outputs. (Each node only has one output variable, but with multiple output variables the network as a whole has *p* output variables. Each node outputs a ? value for every variable except the one specified by the instance stored at the node). Given these assumptions, analysis gives an overall complexity of *O(n+p+log m)* for both execution and learning, for a single pattern.

The complexity of the steps of the execution algorithm is summarized, followed by elaboration where necessary. The numbers correspond to the steps from the algorithms in section 5. The complexity of the learning algorithm is presented in the same format.

<u>Execution Algorithm</u>

1.	Broadcast	*O(n+log m)*
2-2.2.	Node Computation	*O(n)*
2.3-3.	Gather	*O(p+log m)*
total		***O(n+p+log m)***

Broadcast or gather for a single variable is clearly *O(log m)*. For *n* variables, one might expect a complexity of *O(nlog m)*, but the pipelined tree reduces this to *O(n+log m)*, since different levels of the tree operate on different variables simultaneously. The node computation is an *O(n)* operation for *n* variables, but can be overlapped with broadcast. The gather is *O(p)*, one for each output variable, but this is overlapped with the gather of variables up the tree.

<u>Learning Algorithm</u>

1.	Broadcast α_t	*O(n+p+log m)*
2.	Compute Relationships	*O(n+p)*
3.	Check for already learned	*O(log m)*
4.	Check for self-delete	*O(log m)*
5.	Check for DVA	*O(n+p+log m)*
6.	Add α_t to network	*O(n+p+log m)*
total		***O(n+p+log m)***

Steps 1 and 2 can actually overlap. The broadcast is *O(n+p)* under the assumption that values for all *n* input variables and all *p* output variables are sent, even though many of the inputs may be * values and all but one of the output variables are ?. Combining this with the pipelined broadcast down the tree [*O(log m)* for one variable] yields *O(n+p+log m)*. Computation of left- and right-hand side relationships can proceed as each value is received, and hence can overlap with broadcast. Steps 3 and 4 are both *O(log m)* because a single flag is broadcast, and a single value is gathered. Step 6 is *O(log m)* because a free node needs to be located and chosen (this can be done with a single broadcast and gather). It is *O(n+p)* because the left- and right-hand sides need to be sent to the node for storage once the free node is located. The combined complexity is thus *O(n+p+log m)* for step 6.

The complexity of step 5 given above assumes that DVA can take place in parallel, and that there are always enough free nodes in the subtree of any node to accommodate DVA growth. The worst-case scenario is that the deepest leaf node in the tree must add *n*-1 instances to the instance set. In that case, the depth of the tree will be increased by *O(log n)*, for *n* >1, but on the average *n+p+(n-1)/2* variables must be passed to the children at each level. It takes *O(log m)* time for the DVA flag to reach the deepest leaf node. The

O(log n) growth factor will always be overshadowed by the *O(n+p)* time required to pass variables (instances) between nodes. This leads to an overall DVA complexity of *O(n+p+log m)* if DVA takes place in parallel.

The analysis in this section shows that the complexity of execution and learning for the algorithms of section 5 is *O(n+p+log m)*.

7. CONCLUSION

ANNs that use a static topology, i.e. a topology that remains fixed throughout learning, suffer from a number of short-comings. Current research is demonstrating the use of dynamic topologies in overcoming some of these problems. The Location-Independent Transformation (LIT) is a general strategy for implementing variations of ANNs that use dynamic topologies during learning.

This paper presented the LIA model, which is an LIT for ASOCS AA2 model. AA2, like other ASOCS models, has an inherently dynamic topology. In addition to high-level definitions for execution and learning algorithms, more formal definitions are presented for those interested in hardware (and software) implementations. Analysis gives overall complexities for both execution and learning as *O(n+p+log m)*, where *n* is the number of inputs, *p* is the number of outputs, and *m* is the number of instances or Dnodes in original network.

Current on-going work includes the following:

- LITs for other ANNs, aside from those for which transformations have been developed.
- VLSI design and fabrication of LIT models.
- Investigation of appropriate applications, generalization algorithms and global minimization algorithms for ASOCS/LIA class models.

CHAPTER 6

A VLSI Implementation of a Parallel, Self-Organizing Learning Model

in Proceedings of the 12th International Conference on Pattern Recognition © 1994, pp. 373-376

This paper presents a proof-of-concept hardware implementation of the PASOCS model [Mar94a]. This implementation was done in collaboration with Matt Stout of BYU's Electrical Engineering Department, who was researching digital VLSI wafer process technology and investigating the use of Multi-Chip Modules (MCMs) in handling dense interconnect topologies typically required by ANNs. The PASOCS model was suitable for the purposes of this research effort because it can be mapped into digital VLSI technology in straightforward fashion.

Every aspect of the technical design was affected by time constraints, limitations on die size (i.e. chip area), the tools available for design, testing and verification, available fabrication technology and our own evolving experience with VLSI design. We made a number of insightful and creative decisions in this particular design, however there are also some that we could improve upon, given hindsight and somewhat different technical constraints. Overall, the results were very encouraging. Testing showed that the PASOCS chip worked to design specifications.

1. Introduction

Over the past few years, many companies and researchers have announced hardware implementations of neural networks [Int91, [Ame92], [Wri91], [Ver89]. The integrated circuits (ICs) cited are representative of much current neural network implementation research. They are direct implementations of classical neural network research. They are direct implementations of classical neural network structures—a large number of simple computing nodes connected by a dense number of weighted links [Ver89], [Ham90], [Ram90]. The node function is typically a variation on sum-of-products [Rum86]. Early learning models and implementations support only static topologies in the sense that learning does not involve changes in the network topology. More recent models support dynamic topologies in learning, and therefore implementations should also support this. The style of implementations presented here accomplishes this goal.

The implementation described is based on the connectionist architecture Adaptive Self-Organizing Concurrent System (ASOCS) [Rum86], [Mar90], [Mar91b], [Mar94a]. The primary goal of an ASOCS is similar to the goals of many decision -making connectionist systems-the system attempts to learn an arbitrary set of input-to-output vector mappings. However, an ASOCS differs from many other connectionist systems in the manner that this task is accomplished. ASOCS is essentially a rule-based system. That is, an ASOCS learns by the introduction, storage, and comparison of rules. The system is able to learn by keeping itself consistent with the rules and by dynamically changing its topology as new rules are introduced. In this way, an ASOCS can change its structure to suit a particular problem.

This paper presents an implementation of a version of ASOCS called PASOCS (Priority ASOCS) [Mar90], [Mar 94a] that is built on a multi-chip module (MCM) interconnections substrate. Although this implementation is modeled specifically after the PASOCS mode, the general design of this MCM-based system is versatile and can be modified to reflect other connectionist models [Mar90], [Mar94a], [Rud91a]. This model has potential application in areas such as pattern recognition, robotics, logical inference, and dynamic control.

Due to space considerations, background information dealing with the PASOCS learning model is not presented here but can be found in [Mar94a], [Sto94c]. The hardware implementation is described in Section 2, and the PASOCS MCM system, its test results, and ideas for related research are presented in Section 3.

2. Implementation

The IC described in this section was fabricated in 2μm digital CMOS. It was designed using a VLSI layout editor and simulated using an event-driven switch-level simulator. The circuit was constructed using simple cells (NAND gates, NOR gates inverters, etc.) from standard cell libraries and no special purpose cells were required.

Each IC contains the functional hardware for one node of a PASOCS. The nodes of the PASOCS are connected together in a binary tree structure (the logic network), and the root node is connected to a general purpose logic analyzer that acts as a control unit. The node can be described as a combinational logic block with internal latches for information storage. Information presented to the PASOCS is broadcast from the control unit (CU) to the logic network (LN), and information from the LN that must be sent to the CU flows up from the bottom of the tree (the information is gathered.)

The general operation of the PASOCS is synchronous, that is, a global clock is *not* used to drive the system into sequential states. Rather, the clock is replaced by a global mode (sub-mode) signal that places the entire network into a particular mode of operation. Thus, the mode signal is an asynchronous signal that drives the system. Once the CU generates this mode signal, it broadcasts information to the LN, the nodes operate on the new information, and the results of these operations are gathered and eventually reach the CU. After the CU has waited a sufficient amount of time for information to be gathered, the CU generates the next mode signal.

The operation of the system is divided into three general modes: initialization, execution, and learning. In addition, the learning mode is divided into six sub-modes. The functions of the modes and sub-modes are summarized in this section.

2.1 Execution mode

The execution mode (*ex*) is the default mode of the system. If the system is not involved in learning, then it constantly updates its output in response to changes in the environment state. The output of the system consists of the right-hand side (RHS) of the winning node's stored instance (SI) and its corresponding priority. (The winning node is the node with the highest priority whose SI matches the environment state.) If the environment does not match the SI of any node, then a priority of zero is received by the CU.

Each variable of the SI (and the new instance in the learning sub-modes) consists of two bits: a "data" bit and a "care/don't care" bit. The data bit is the actual value of the variable. However, if the value of the care/don't care bit is zero, then the variable is a "don't care". The match output of every node during *ex* for a single variable is summarized in Table 1.

d	*SId*	*SIc*	Output	*d*	*SId*	*SIc*	Output
0	0	0	match	1	0	0	match
0	0	1	match	1	0	1	no match
0	1	0	match	1	1	0	match
0	1	1	no match	1	1	1	match

Table 1. The match function for one variable. The abbreviations are as follows: *d*, the value of the environment variable; *SId* the data bit of the SI; *SIc*, the care/don't care bit of the SI.

This prototype has an instance size of four input variables and one output variable. Therefore, the four-variable match function is accomplished by using four single-variable match functions as shown at the top of Figure 1. In this figure, *d0-3* represent the four environment variables, *SId0-3* the four SI data bit values, and *SIc0-3* the four SI care/don't care bit values.

The four single-variable match functions and the AND gate at the top of Figure 1 determine whether the left-hand side (LHS) of the SI matches the environment state. The priority is passed to the Priority Comparison Function only if a match occurs and the node is not free-that is, if the node contains a SI. Otherwise, the priority is masked to the value of zero. The priority Comparison Function compares the priority of the node with the priorities of its children, and the winning (or highest) priority and corresponding RHS is sent to its parent. This process is continued for each node until the CU receives the RHS and priority of the LN's winning node.

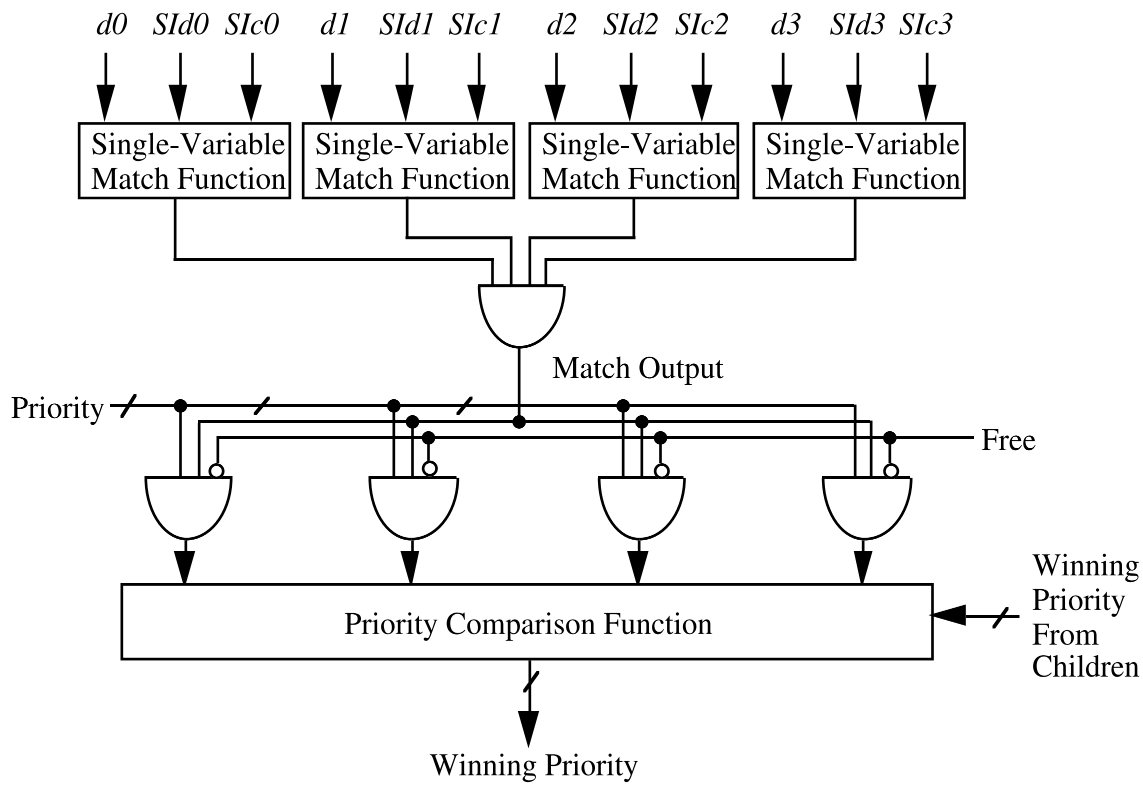


Figure 1. Block diagram of execution mode.

2.2 Learning sub-mode 1

In learning sub-mode 1 (*l1*), the relationship of the SI to the new instance (the instance to be learned) must be determined. The relationship is determined with a three-level logic structure (shown at the top of Figure 2) that operates only on single variables of the new instance (NI) and SI. Figure 3 is a sequential state-flow diagram that clarifies the process of determining the relationship between the LHSs of two instances using single-variable comparisons. From the "Start" state, the four possible single-variable comparisons determine the path to the next state. For example, if the first-order relationship is "SB" (subset), then the path for the "Start" to "SB" is followed. All successive comparisons determine the path to the next state. For example, if the next comparison results in "D" (discriminated), then the path to "D" is followed. Every successive comparison will remain at "D".

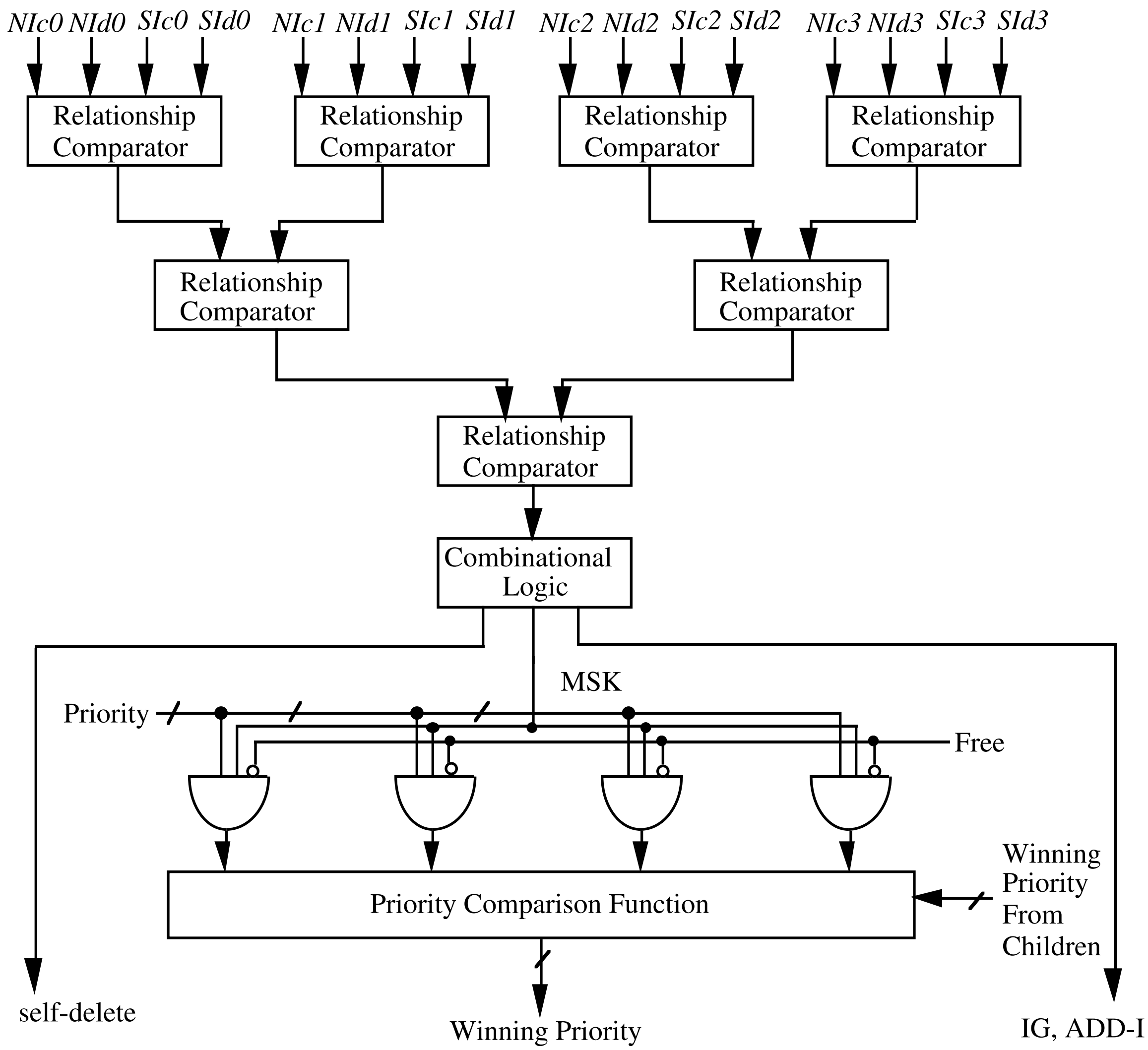


Figure 2. Block diagram of learning sub-mode 1.

The relationship between the SI and NI determines whether the action signals MSK, IG, ADD-I, and self delete (shown in Figure 2) for each node are asserted, as shown in Table 2. Once the values of these signals are determined, each node compares its action and priority with that of its children, resulting in a winning action and corresponding priority for the entire system.

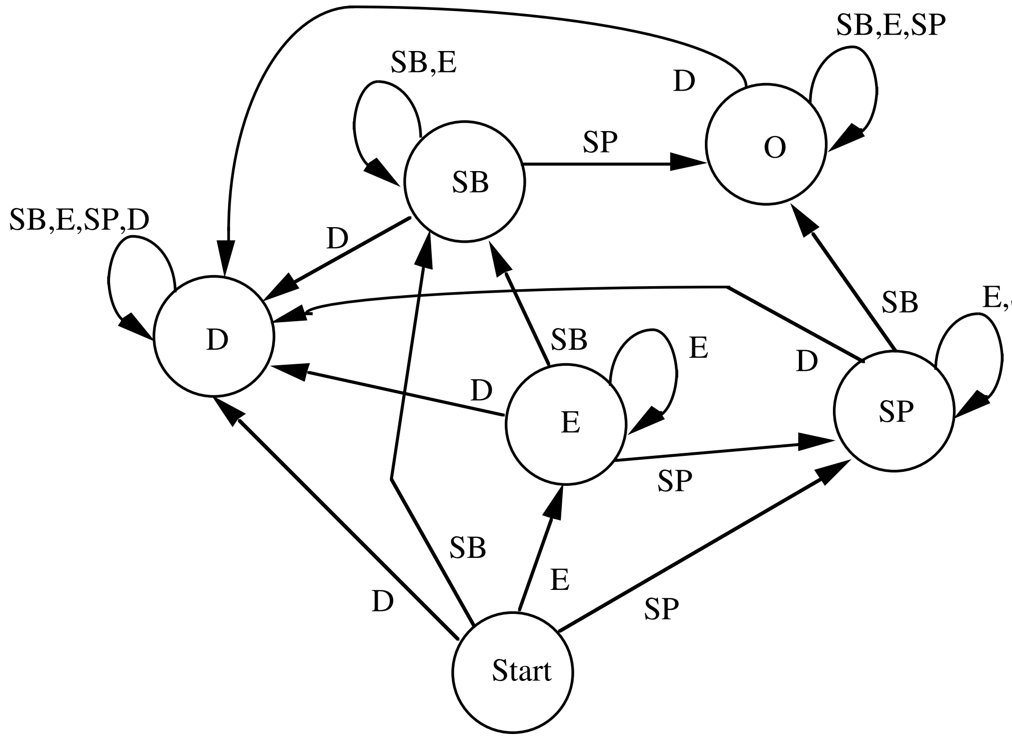


Figure 3. A sequential view of the parallel relationship comparison circuit (from *l1*) that shows how the relationship between two instances can be determined using only single-variable comparisons. The abbreviations are as follows: SB, subset; E, Equal; SP, Superset; D, Discriminated; O, overlap.

	Subset	Equal	Superset	Discr.	Overlap
Con.	ND/IG	D/MSK	D/MSK	ND/MSK	ND/MSK
Dis.	ND/ADD-I	D/MSK	D/MSK	ND/MSK	ND/ADD-I

Table 2. The actions generated by the node. The abbreviations are as follows: Discr., Discriminated; Con., Concordant; Dis., Discordant; D, Delete node; ND, do not delete node; MSK, Mask priority to zero; IG Ignore NI; ADD-I, Add the NI with a priority one higher than the highest priority of any inconsistent SI.

The next mode or sub-mode of operation depends on the winning action reported to the CU in l1. That is, if the action IG is reported, then none of the following learning sub-modes are performed. If the reported action is ADD-I or if a priority of zero is received by the CU (MSK), then the remaining learning sub-modes are performed.

2.3 Learning sub-modes 2-6

The function of the remaining sub-modes is to keep the system minimal and consistent. This is done by allocating a new node that contains the NI and corresponding priority and by deleting redundant or contradicting information from the system. After this is done, the CU proceeds to the default mode (*ex*) or to *l1* if additional learning is desired.

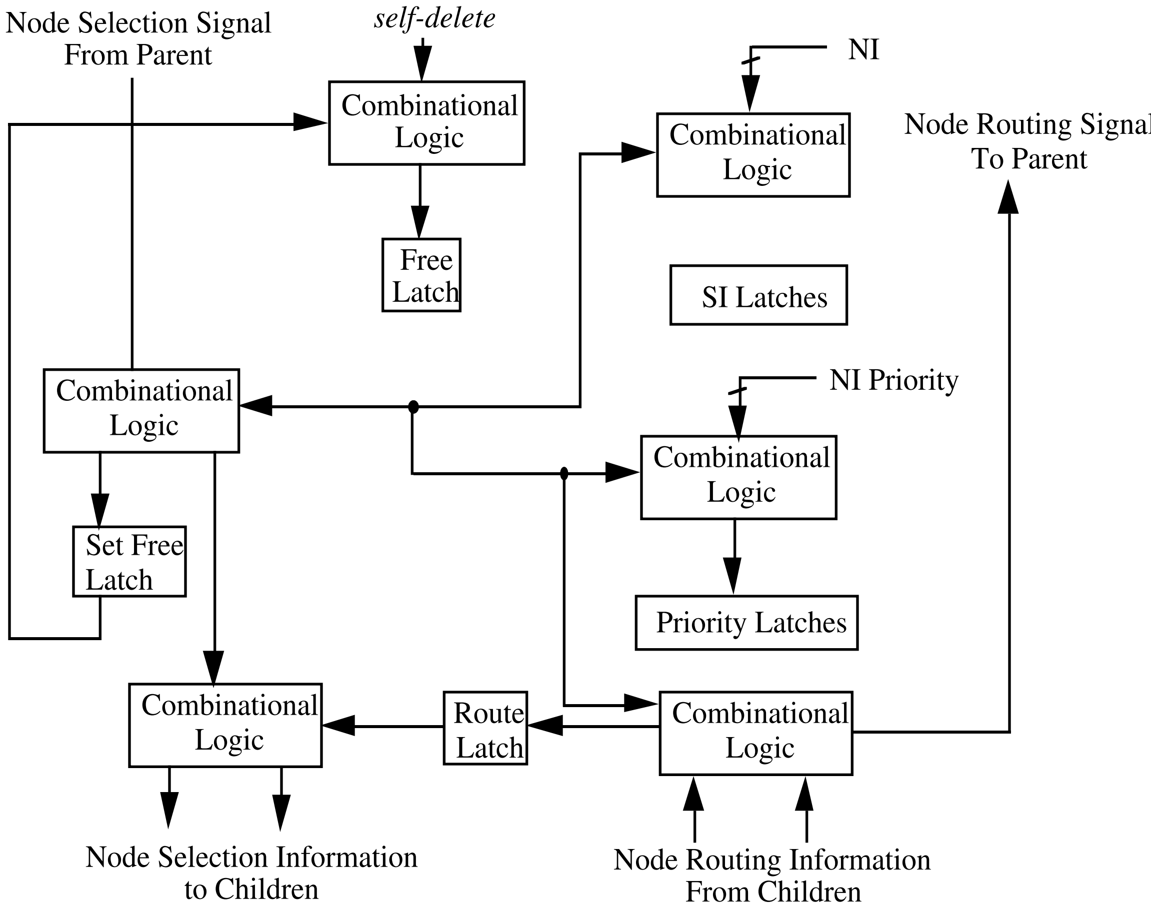


Figure 4. Block diagram of learning sub-modes 2-6.

The block diagram of learning sub-modes 2-6 is shown in Figure 4. In learning sub-mode 2 (*l2*), the free latch is toggled to a high value if the self-delete signal was asserted in *l1*, effectively deleting the node from the system. In learning sub-mode 3 (*l3*), the NI is stored in the latches of every free node in the system. In *l4*, each node sends its routing information to its parent. At the end of this sub-mode, every route latch in the system contains information that determines a unique path from the root node down through the tree. This unique path is guaranteed to contain at least one free node that now has the NI and its priority stored in its latches. From the information contained in the newly acquired routing information and the present state of the free latch, the next state of the free latch of each node is determined in learning sub-mode 5 (*l5*). This state is temporarily stored in the set free latch until its value is needed in learning sub-mode 6 (*l6*). Finally, in *l6*, the set free latch's value is used to update the free latch's value. In other words, a single free latch's value is set to a low logic level, effectively adding it to the tree.

3. System design, testing, and ongoing research

Figure 5. The PASOCS MCM prototype. The dimensions of the MCM are 2.5cm×2.3cm.

Three of the ICs described in Section 2 have been interconnected on a multi-chip module interconnection substrate (see Figure 5) in a three-node feasibility study, and this small PASOCS has been tested. To test the system, various combinations of instances were presented, and the system's outputs were observed.

The execution mode of the PASOCS operates as predicted by the software simulation, but two functions of the learning sub-modes (associated with overall network minimization and rule relationships) do not operate to store invalid information and therefore return incorrect information in the execution mode. It should be noted that these are problems with the specific implementation of the ICs and not with the functionality of the PASOCS model or with the original conceptual design of the IC's, described in more detail in [Sto94c]. Despite these problems, most of the functions of the three-node PASOCS are functioning according to original design specifications. A detailed report of the test results can be found in [Sto94c].

In addition to the research reported here, ASOCS models other than PASOCS are also being investigated. These and other more classical neural network models can benefit from the results presented in this paper. Current research seeks to extend the general ideas presented here to other models.

Other related research seeks high-density interconnect technologies that can be exploited to create larger neural network systems [Sto94a,c]. MCM interconnection and packaging techniques offer a promising solution to the high interconnect and processing element densities required for such hardware implementations. The MCM for this prototype was fabricated in the Integrated Microelectronics Laboratory at Brigham Young University. Other models that can benefit from MCM characteristics are also being investigated.

4. Conclusion

This paper described a VLSI implementation of a connectionist system that was built using a MCM interconnection substrate. The IC differs significantly from many other commercially available ICs and research projects dealing with connectionist architectures. Many of these are modeled after the classical "weighted connections" neural systems. however, this IC is modeled after PASOCS (Priority Adaptive Self-Organizing Concurrent System) and represents a different approach to implementation.

CHAPTER 7

A Multi-Chip Module Implementation of a Neural Network

in Proceedings of the IEEE Multi-Chip Conference © 1994, pp. 20-25

The ANN discussed in this paper is the PASOCS model discussed in Chapter 6, however this paper focuses on the Multi-Chip Module (MCM) substrate that is used as the interconnect medium. The network nodes are physically located on a set of chips, in our prototype one node per chip. The chips are then embedded in the MCM substrate, which connects the nodes together and also links the network with the external host, or Control Unit.

This paper discusses the use of MCMs for ANNs, fabrication process technology, and testing and verification issues. These results are primarily the work of Matt Stout, but working jointly with George Rudolph.

1. Introduction

Artificial neural networks offer an exciting area of research because of their ability to solve difficult problems, typically those dealing with pattern recognition. This ability is due, in part, to their densely interconnected parallel architecture. However, often neural networks are simulated on sequential computers and lose much of their potential speed capability, since this inherent parallelism is lost [Ram91]. Hardware implementations of neural networks offer a solution to this drawback but also present several challenges.

One particular challenge to the development of neural network hardware implementations is the typically high interconnect density that these implementations require. The required density can be appreciated by considering the equations that govern the dynamics of neural network models. Ramacher and Schürmann [Ram 91] have shown that the operation of most neural network models can be summarized by three general equations, the first of which is presented here without derivation. The output characteristics of a neuron (processing element) are described by:

$$y_{i,p} = f_i(x), \; 1 \leq i \leq N, \; p \in P \tag{1}$$

where the argument of f_i is

$$x = \sum_{j=-1}^{N} W_{i,j} \, y_{j,p}. \tag{2}$$

In equations 1 and 2, N is the number of neurons, i and j are arbitrary neurons, P is the index set for the patterns, p is an element of this set of patterns, f_i is the activation function of neuron i, $W_{i,j}$ is the connection strength or weight from neuron i to neuron j, and $y_{i,p}$ is the input to neuron j corresponding to pattern p. Specifically, y-1,p is the neuron's individual input, $W_{i,0}$ is the neuron's individual threshold, and

$$W_{i,-1} = y_{j,p} = 1 .$$

The other general equations reported in [Ram91] are not presented here. Briefly, they represent the changes in the values of the weights between neurons and the updated values of these weights.

These general equations can be used to estimate the general hardware requirements of neural networks. For instance, Equations 1 and 2 show that for the most general neural network consisting of N completely interconnected neurons (each neuron is connected to every other neuron), the outputs of the neurons $y_{i,p}$ are determined by the sum of products of the interconnecting weights $W_{i,j}$ and corresponding node inputs $y_{i,p}$. For this case, the number of connections is $O(N^2)$, which shows that present neural models could potentially require huge amounts of interconnections.

This observation has additional significance when practical implementations of neural networks are considered. A DARPA study [DAR88] has reported characteristics and requirements of various neural network applications. These proposals include radar pulse identification, robot arm movement, isolated word recognition, low-level vision, and risk evaluation. The number of neurons ranges from 312 to 64,000, and the number of synapses ranges from 3,600 to approximately 4,000,000. In this sense, a synapse refers to a weighted connection, or in other words, the product $W_{i,j}y_{j,p}$ from Equation 2. Since the required interconnect density for an implementation is directly related to the number of synapses, it can be seen that practical hardware implementations of neural networks could potentially require many interconnections.

This requirement for dense interconnect in artificial neural network systems has led researchers to seek high-density interconnect technologies [Cam93], [Dis90], [Yas91]. This paper reports an implementation of a connectionist model using a multi-chip module (MCM) as the interconnect medium. In this sense, a connectionist model refers to a system composed of fairly simple processing elements which are densely interconnected; artificial neural networks constitute a subset of connectionist systems. In general, the dynamics of all connectionist systems are not accurately described by the general equations presented earlier. However, like neural networks, typically the processing elements of general connectionist models require many interconnections. The authors believe that MCMs can be used as an effective interconnect medium for artificial neural networks as well as other general connectionist systems.

The integrated circuits (ICs) used in this MCM system are modeled after PASOCS (Priority Adaptive Self-Organizing Concurrent System), which is based on the connectionist architecture ASOCS (Adaptive Self-Organizing Concurrent System) [Mar91a,b]. Although this connectionist model differs significantly in its mechanisms from the neural network models whose characteristics are described by the general equations mentioned earlier, the goal of both types of connectionist models is the same—both attempt to learn an arbitrary set of vector mappings. As mentioned previously, the models' similarities extend even further, since a practical system based on the ASOCS architecture would also typically require many interconnections.

The paper begins by briefly describing the main differences between MCM and other and other high-density packaging technologies, and then introduces the MCM process used for this implementation. Next, general overviews of the learning model and its hardware implementation are presented. Finally, several important issues pertaining to this general implementation approach are briefly explained.

2. High-density Interconnection

In this section, the characteristics of multi-chip module packaging technology are briefly reviewed. Then, two other technologies that offer high-density printed circuit boards. Finally, the specific multi-chip module process used for this implementation is described.

2.1 Overview of multi-chip modules

The typical process used in the production of electronic systems consists of fabricating many identical IC's on a single wafer, separating and packaging the ICs, and then externally connecting the packaged ICs on some type of interconnect medium, such as a printed circuit board (PCB). In an MCM, unpackaged ICs are mounted on a substrate which contains multiple layers of interconnect. The ICs are mounted to the substrate and electrically connected (by wire-bonding or other methods) to the substrate [Doa93], [Her 90].

Multi-chip module packaging technology is an important technique which results in high chip density, small interchip propagation delay, low power consumption, and high interconnect density. A four-layer thin film MCM process may have a maximum interconnection density as high as 300-800 linear centimeters of interconnections per square centimeters of die area (cm/sq. cm) [Doa93, page 97]. In addition, as will be explained in section 3.3, and MCM can accommodate many different types of ICs, such as logic, memory, analog, etc. For these reasons, an MCM approach was taken for the system implementation described in this paper. As will be discussed later, other neural network models may also benefit from the characteristics of MCM packaging technology described above.

2.2 Overview of other technologies

Wafer-scale integrations (WSI) and printed circuit boards (PCBs) are two other packaging approaches that can have relatively high interconnect densities. However, these methods suffer from other disadvantages compared to MCMs.

In WSI, ICs and their interconnect are fabricated on a wafer concurrently. The result is a very dense collection of ICs and interconnect which can be packaged as a single unit [Her90]. As with MCMs, this technique results in high chip density, small interchip propagation delay, low power consumption, and high interconnect density, typically somewhat higher than the interconnect density that can be obtained with MCMs. However, a WSI fabrication approach has several limitations: it suffers from relatively low yield, it demands a high entry-level cost to evaluate, and the types of ICs fabricated must be very similar. Due to this last constraint, WSI has proven to be primarily useful in applications where a large number of identical or similar ICs are required. many connectionist systems, such as neural networks, fall into this category [Cam93], [Dis90], [Her90], [Yas91]. However, as will be explained later, some connectionist models may benefit from the ability to use different types of ICs in a hardware implementations.

A printed circuit board is another approach that can be used to implement neural networks. Even though this approach offers high yield and is relatively simple and inexpensive compared to MCMs, it suffers from lower wiring densities and therefore may not be capable of providing the high interconnect density required for some connectionist system applications. One estimate of the interconnect density of a 20-layer PCB is about 140-260 cm/sq. cm [Doa93, page 97] which is substantially lower than the density that can be obtained using MCMs. In addition, if packaged components are mounted on PCBs, much more space is required than a comparable system using MCMs and unpackaged die.

2.3 Specific MCM approach

A hardware implementation of the PASOCS model is being developed at Brigham Young University. The implementation consists of three identical die mounted on an MCM substrate; future implementations will use different types of ICs as will be explained later. The die are 2μm digital CMOS ICs fabricated through the MOSIS service and are 2.5mm times 2.7mm in dimension.

The entire MCM structure used in this implementation was fabricated in the class-10 clean room facility of the Integrated Microelectronics Laboratory (IML) at Brigham Young University. The MCM substrate was fabricated using two levels of the standard four level metal IML process as described below. After substrate fabrication, the custom ICs are mounted the MCM substrate and tested using a digital tester. The process used for the PASOCS implementation was optimized for ease of fabrication and maximum interconnect density. As a result, comparatively thin metal layers were used. Only two levels of metal were used in this three-die feasibility study, but future applications will require greater interconnect density and will use the additional metal layers available in the process. The entire four-level process with 40μm lines and pitch results in an interconnect density and will use the additional metal layers available in the process. The entire four-level process with 40μm lines and pitch results in an interconnect density of about 500cm/sq. cm.

A cross section of the IML MCM substrate process is shown in Figure 1. The thickness of the various layers are to scale, although they are not at the same scale as the lateral dimensions. The slope of the vias and metal layers is also approximately correct.

Deviation from vertical is caused by the characteristics of the metal and via definition etches.

The first step of the process is the growth of a 1250A layer of $Si0_2$ on the MCM substrate to isolate the interconnect structure from the conductive silicon substrate. Following $Si0_2$ growth, the first level metal layer of 2µm of aluminum is sputter deposited and patterned using a wet etch process.

Figure 1. Cross-section of the IML MCM substrate. Although the four-level process is shown, only two levels of metal were required for this implementation.

The first level metal layer is followed by spin coating of the substrate with an 8µm interlevel dielectric of Dupont 2611 polyimide. Vias are plasma etched in the polyimide to provided connection between the first and second level metals. Following via definition, a second level of 2µm TiAl is deposited and wet etched. In the full four-level metal process, the polyimide/metal steps are repeated two additional times, although in this application only two levels of metal were required. Following definition of the final metal layer, a top coat of polyimide is spin deposited for protection of the interconnect structure, and pad vias are etched to provide access for wire bonding.

3. The learning model

In this section a brief overview of the PASOCS model is given. Then, a description of the hardware implementation, including the individual VLSI IC's and the final MCM system, is presented. Finally, important characteristics of this implementation approach are examined which are applicable to other connectionist models.

3.1 Overview

The system implemented as an MCM is based on the connectionist architecture ASOCS. The primary goal of an ASOCS is similar to that of many decision-making connectionist systems—the system attempts to learn an arbitrary set of vector mappings. However, an ASOCS differs from many other connectionist systems in that it learns by the introduction of rules rather than a training set of input/output vectors. The system is able to learn by keeping itself consistent with the rules and by dynamically changing its topology as new rules are introduced. In this way, an ASOCS can change its structure to suit a particular problem.

The particular model implemented in this study is PASOCS, which is one of a class of ASOCS connectionist models. See [Mar 91a,b], [Mar 94a] for background information on the ASOCS and PASOCS models. Briefly, a PASOCS is a network of self-organizing digital processing elements (or nodes) which accomplishes the following [Mar 94a]:

- Processes inputs in the form of Boolean variables and outputs Boolean results
- accepts rules made up of a conjunction of Boolean inputs which imply a Boolean output
- learns new rules over time;
- automatically resolves rule conflicts;
- combines specific rules into more where appropriate; and
- maintains an associated priority with each rule.

In order to accomplish these goals, the nodes in the network self-modify using local information. The rules (inputs) to a PASOCS are stored in the nodes. Typically, a practical implementation of a PASOCS would require hundreds or thousands of nodes which are densely interconnected. This is one of the requirements that makes a multi-chip module implementations advantages.

3.2 Hardware implementation and testing procedures

Each IC described in Section 3.1 contains the functional hardware for one node of a PASOCS. Details of these ICs can be found in [Sto94c]. For this study, four packaged ICs and four unpackaged ICs were fabricated. First, the individually packaged ICs were tested and compared to the simulation results obtained during the initial design. Then, three of the packaged ICs were connected on a PCB as a three-node PASOCS and tested. Next, as a prerequisite to mounting the ICs on the MCM substrate, the unpackaged ICs were tested separately. The three ICs are mounted on the MCM substrate using a die epoxy and tested by using a pad ring on the MCM which allows access to all of the I/O of the PASOCS. This pad ring can be seen at the top of Figure 2. The 40-pin die are mounted and wire-bonded to the three pad rings at the bottom of the figure.

Figure 2. The mutli-chip module substrate.

The results of the tests have generally been positive. Most of the functions of this three-node PASOCS are performing according to original design specifications except for two functions associated with overall network minimization and rule relationships. It should be noted that these problems with the specific implementation of the ICs and not with the functionality of the PASOCS model or with the original conceptual design of the ICs as shown in [Sto94c].

3.3 Important characteristics of an MCM approach

MCM technology has important characteristics that make it an attractive option for implementing connectionist systems. some of these, which have already been mentioned in section 2.1, include high chip density, small interchip propagation delay, low power consumption , and high interconnect density. Another important characteristic is the potentially high yield that can be obtained from this approach as opposed to WSI. Additionally, improvements in bare die testing techniques will help make higher MCM yield possible [Kee92, Par92], since this will assure that only KGD (known good die) are mounted on the MCM. The die in this project were not individually tested using an approach which is not appropriate for general manufacturing requirements.

Another important advantage of this approach is the ability to use different types of ICs in the design. the model described in this paper, for instance, includes both logic and memory circuits on the individual ICs. The memory of the ICs, however, is bulky and inefficient. The memory requirements would be met much more elegantly if smaller and denser memory cells (DRAMs, for instance) were used. The logic could be designed on CMOS ICs and DRAMs could be mounted next to these logic ICs on the substrate. In addition, other connectionist systems, including artificial neural network models whose general equations are discussed in Section 1, may also benefit from this ability. In some cases, it may be most efficient to fabricate the neurons and associated logic circuits in digital CMOS, while the adjustable weight ICs may be more easily of efficiently implemented as DRAMs or some type of analog device. The neural network could then be built with these different types of ICs using an MCM as the interconnect scheme.

Present research at Brigham Young University includes testing and simulation of the ideas presented as well as initial research into other connectionist systems that may benefit from an MCM implementation approach. Current work has shown that the general ideas presented are versatile and can be modified to reflect other models [Rud91a].

4. Conclusion

This paper described a multi-chip module implementation of a connectionist system. This system differs significantly from many other commercially available neural network systems and research projects, since MCMs are used as the interconnect medium.

The paper described the specific MCM process currently in use by the Integrated Microelectronics Laboratory at Brigham Young University. the specific connectionist system implemented as an MCM is also described. The general ideas pertaining to MCM implementation of connectionist systems also apply to other models, including artificial neural networks.

CHAPTER 8

A Learning Model for Adaptive Network Routing

in Proceedings of the International Workshop on Artificial Neural Networks and Telecommunications © 1994, pp. 183-187

The increasing size, complexity, and dynamics of the internet, WANs and LANs requires adaptive routing mechanisms. This paper proposes initial concepts toward a learning and generalization mechanism to support adaptive real-time routing, presenting additional research based on ideas originally proposed by McDonald and Martinez [McD91]. An ASOCS learning model is employed as the basic adaptive router. Generalization of routing is based not only on source/destination address, but also on such factors as packet size, priority, privacy, network congestion, etc. Mechanisms involving continual adaptation based on feedback are presented. Extensions to conventional addressing which can support learning and generalization are proposed.

1. Introduction

As networks increase in size and routing complexity, new types of mechanisms are required to support routing decisions. In particular, network routers must accommodate and adapt to rapid changes in network topology and usage.

This paper proposes a learning and generalization scheme to support adaptive real-time routing in computer networks. Traditional routing algorithms are based on tabular storage of destination/gateway tuples which decide where a packet should be forwarded upon arrival at a gateway. This method suffices when this tabular information is available and up to date. When not available,mechanisms such as default gateways, random forwarding, or message discarding are employed. We propose that generalization schemes may be used to forward packets in a more efficient and accurate manner when confronted with insufficient and inaccurate routing information.

2. Approach

Most current algorithms base routing decisions primarily on the destination address. Though destination address is obviously crucial, other information may be important in making a proper routing decision. For a packet, this information includes packet size, priority, preferences (latency, throughput, reliability, etc.), privacy classification, etc. For the network such information as congestion, current failures, load, etc., may be pertinent. We call these *packet features* and *network features* respectively. We refer to address, packet features and network features collectively as *routing features* in this paper. Ad hoc algorithms require unique handling of these different features. A more general learning scheme treats all of these uniformly, with weighting of importance taking place during learning, thus both simplifying and expanding the approach to routing.

McDonald [McD91] proposes an Adaptive Self-Organizing Concurrent System (ASOCS) model as an adaptive network router. ASOCS [Mar88], [Mar91a,b] are a class of connectionist learning models which seek computational efficiency as highly parallel and adaptive connectionist logic systems that map discrete input values to output values. An ASOCS can infer routing decisions from routing rules that have been expressed as discrete input variables. This paper seeks to extend this approach through more robust generalization techniques. Below, the adaptive mechanism used for learning routing information and doing actual forwarding is simply referred to as the *router*.

The approach proposed in [McD91] assumes a version of the learning bridge algorithm. A router observes the source address and routing features of each incoming packet and remembers on which interface it arrived. When the address and other features are later observed as a destination, then the router forwards the packet on the interface on which the address was earlier observed as a source address.

As new nodes are added to a network (or even with a network initialized with no routing information), these nodes can send out a *beacon* packet which is forwarded along all routing lines except that on which the beacon was received. This process "primes" the network with initial routing information which is learned as the network evolves. By adding distance information (number of hops, time lapse, etc.) to the beacon packet, an accurate assessment of efficient routing is captured. However, accuracy of information may still be faulty due to the following.

- The beacon process above must be limited because of excessive bandwidth requirements.
- Conditions at the time of the initial beacon were not representative of average case.
- A constantly changing network causes initial information to be inaccurate.
- The initial beacon is based only on source address and not on other routing features.

These and other factors cause the necessity of ongoing learning and adaptation to the network environment. The model above provides a reasonable initial state from which continual adaptation and refinement can occur. This adaptation requires some source of feedback to the router to allow update of its current state. Some potential approaches include:

- Feedback Protocol: This protocol would cause (optionally) the router which received the packet to send information back to the source address regarding the "cost" of receiving the packet. This cost is then used to update and improve the initial sending router. Note that cost in this context may involve not only time to arrival, but metrics relating to errors, privacy, reliability, etc.
- Piggy-Back Metrics: A feedback protocol will add to network traffic. Another option is to maintain the current cost metric with the packet. Upon arrival at a router, if this metric is excessive are not in line with a *predicted* cost stored at the receiving router, only then need this information be sent back to the source routers.
- Inquiry Packets: Discovery of improved routing can be initiated by a router through the sending of *inquiry* packets which are a more selective version of beacon packets. In particular, a router with poor predictive cost may seek improved routes to commonly occurring destinations. These can be initiated at times of lower network load so as to seek constant improvement without an adverse effect on network traffic.

3. Learning and Generalization

There are a number of learning and generalization models which could be used to support generalization of network routing. In the case where a gateway has seen an incoming source address and later needs to send to that address as a destination, the mechanisms of backward learning overviewed above could suffice. However, there are times when a source needs to send to an address from which it has not yet received a message and thus no information regarding that address will be found in the gateway's current routing tables. This lack could be due to newness of the destination site, error at the routing table, etc.

In this case the address given by the initial sender could not just be the standard symbolic name (i.e. jdoe@cs.univ.edu), since the name server is not yet aware of this address. The numeric name could be given and although the gateway has no record of this yet, generalization could be done to make a best guess at routing, rather than the standard return of host unknown. However, since numeric addresses are typically given randomly to new hosts, no interesting generalization can be made from the address number itself. For example, a learning model could note that addresses 123, 129 and 140 are best delivered to output line 1 and 124, 160, and 162 to output line 2. If a new address, 144, is the destination and it has not been previously seen, the learning model could only make an arbitrary guess as to the best output line. This is because the input feature is a random "address" which has no regularity or critical sub-features which can be learned to discriminate or join itself with/from other addresses. As mentioned above, other routing features can be used to aid in generalization, but without at least some candidate output lines based on address the other information is moot.

For any learning system to actually be able to generalize on inputs not seen before, the input must include features which have within them some kind of regularity which the learning model can extract. We propose that to support generalization on unknown addresses the address itself must contain some extractable regularities. For example, a uniquely identifying address (which could be a random set of bits) could be augmented to also contain geographical information, such as the approximate latitude and longitude of the address site. (Similar extensions having different motivation and advantages are proposed elsewhere [For93], [Kat93]). In this case, there is extractable regularity to be learned from addresses both incoming and already available in the routing table. When a new address is the destination, one possible learning bias is that the packet should be sent on an output line which has been successful for addresses with a similar geographic location as the new one. At this point, deciding between candidates can be enhanced using other routing features which are a part of the packet itself.

Other regular features could be part of the basic address. One which is already part of current internet addresses is the organization type (i.e. edu, com, gov). This type of information can be used to aid in learning what types of lines (security, speed, size, etc.) are typically best suited to communication involving these classes of users.

While too much addition of regularity would lead to cumbersome addresses, a judicious extension to the basic address mechanism could allow generalization on routing to unknown locations with reasonable cost and accuracy.

4. Minimization

While not supporting full generalization as in the previous section, the learning bridge algorithm can lead to more concise routing information for addresses actually received. Hierarchical IP addresses (i.e. address prefixes, geographical information, or an equivalent labeling system) could provide hierarchical abstraction through the process of minimization. A pair of prefixes of length N can be minimized to a single prefix of length N-1 if the prefixes differ in only one bit (e.g. the prefixes 1010 and 1011 can be summarized to the prefix 101*, where * is a don't care variable). This can be repeatedly applied so that a set of longer prefixes can be combined into a single, much shorter prefix. (Note that this same philosophy can be used with the geographical extension of section 2 to allow the representation of a large number of routing tuples into a single rule based on geography or other feature.) This includes prefixes for both route "table" entries (internal minimization), and route update information sent across the network (external minimization). Minimization thus helps reduce growth and size of routing tables, as well as route information traffic in the network. If eight sites are attached to the same network regional router, and their prefixes can be minimized, then the regional router can store, and advertise, its reachability to all eight sites with a single prefix. As a result, the number of routing entries and routing information traffic will scale less than linearly with respect to the number of networks.

5. Conclusion and Current Work

This paper presents initial approaches which can support routing generalization for novel network addresses. While these ideas are still in an early stage, current work seeks to demonstrate potential through network routing simulation and generalization studies.

CHAPTER 9

Software Simulations of LIT Models: Results and Verification

not previously published

This chapter discusses software simulations for two LIT models as an illustration and verification of LIT concepts: the static Competitive Learning (CL) model [Rum 86] and the standard static Backpropagation model (BP) [Rum86]. The major purposes of the simulations are 1) to demonstrate that the alternate data structures produced by LIT are functionally equivalent to the original models, and 2) to provide concrete examples of the structure and operations of LIT models. The correctness of the LIT versions of Backpropagation and Competitive Learning is verified by comparison with the results of the well-known PDP software versions of these models [Rum88].

The LIT CL and BP models are verified by showing that given the same initial weights as the respective original model, the LIT model computes the same final set of weights as the original after training. The specific procedure was as follows:

1. Run the PDP software model with a specific network structure, allowing the program to generate a random set of initial weights.
2. Train the network on the desired set of input patterns until learning is complete.
3. Initialize an LIT network with the corresponding transformed structure using the same .net file and the initial weights generated in step 1.
4. Train the LIT network with the same set of input patterns as used in step 2 until learning is complete.
5. Verify by comparison that the final set of weights generated in step 4 is the same as the final set of weights generated in step 2.

This procedure was used for a variety of different networks for both the CL and BP networks. One example for a BP network is given here. The file "bp.net", given below, specifies for the PDP software a BP network with 3 input nodes, four hidden nodes and 2 output nodes:

```
nnodes    4
ninputs   3
noutputs  2
lrate    .5
momentum .9
nepochs   70
errthresh .04
```

One set of initial weights generated by the PDP program for this network are as follows. There are 29 in all—12 from the three input nodes to the four hidden nodes, eight from the hidden nodes to the two output nodes, and nine bias values (one per node, but the bias values on the input nodes are always fixed at 0):

```
-0.251358
-0.305154
0.066575
-0.479103
0.256752
-0.105454
-0.190942
-0.091310
-0.199735
0.320085
0.145440
```

-0.286628
0.291473
-0.129479
0.003832
-0.056903
0.489464
-0.469373
0.237448
-0.116433
0.000000
0.000000
0.000000
-0.480474
-0.451008
0.350069
0.236098
-0.265642
-0.128016

The set of input patterns used in this example are the following. Each pattern is preceded by a name for convenience in displaying the current pattern, if desired.

```
p1 0 0 0  0 1
p2 0 0 1  1 0
p3 0 1 0  1 0
p4 0 1 1  1 0
p5 1 0 0  1 1
p6 1 0 1  1 1
p7 1 1 1  1 0
```

The basic structure of a network in the PDP simulations is a collection of one- and two-dimensional arrays of weights and values. The weights are stored in a two-dimensional array, where the weight between two nodes *i* and *j* is the value stored at array location [*i*][*j*]. Various other arrays, with correlated indexes, are used to implement the full functionality of the CL and BP models [Rum88].

The LIT simulations represent the same information and functionality, but in a fashion that is more convenient for dynamic topologies (adding and deleting nodes/information in an efficient manner). Below are the declarations for the LIT CL and LIT BP nodes:

```
class Clnode: public Listnode {
 private:
float activation;
   Reallist weights;
   int classNum;
   char *Name;
 public:
   Clnode();
   Clnode(Reallist*, int, char *);
   ~Clnode();
   float Compute_activation(Intlist *);
   void set_activation(float);
   float get_activation();
   int get_class();
   char *getname();
   Reallist *getweights();
```

```
  void setweights(Reallist *);
  void Changeweights(Intlist *, float);
  virtual void Print();
};

class Bpnode: public Listnode {
 protected:
  float activation;
  float netin;
  RWnode inbias;
  RWlist inwts;
  RWlist outwts;
  Reallist outvals;
  int nodenum;
  char *Name;

 public:
  Bpnode();
  Bpnode(RWlist*,RWlist*,int,RWnode *,char *);
  ~Bpnode();
  virtual Reallist* Compute_output(Reallist *);
  Reallist *getoutput();
  int getnodenum();
  char *getname();
  RWlist *getiwts();
  RWlist *getowts();
  void setiwts(RWlist *);
  void setowts(RWlist *);
  float getbias();
  void setbias(float);
  virtual void Changeweights(Reallist *,Reallist *);
  virtual void setlrate(float);
  virtual void setmu(float);
  virtual void Print();
};
```

The set of weights resulting from training are the following. (Again, the corresponding weights from the LIT model are the same.)

2.658971
-2.051182
-0.922298
-2.487324
4.209701
4.020051
0.097965
-1.872894
-1.690708
3.996062
1.663158
1.676500
1.057750
3.396119
-2.038943
5.184994

3.684223
-6.131297
2.574614
0.610079
0.000000
0.000000
0.000000
-0.591907
-2.601937
0.988773
-1.700835
-1.824855
0.236653

CHAPTER 10

Conclusion

This chapter summarizes the major results of the dissertation, and then discusses potential future research directions.

1. Summary

The major aim of this dissertation was to develop a general strategy for supporting hardware implementations of static and dynamic versions of common ANN models. More direct hardware support for neural mechanisms can potentially pay off in terms of speed, cost efficiency, and the development of existing and as-yet-undeveloped applications that can take advantage of such hardware support.

The *Location-Independent Transformation* (LIT) strategy is a general hardware implementation strategy for static and dynamic ANNs. LIT research emphasizes support for ANNs with dynamic topologies for three main reasons: 1) Hardware implementations of static models are ubiquitous in ANN literature, 2) LIT is a generalization of ideas developed for hardware implementations of ASOCS models, which use inherently dynamic topologies, and 3) current research indicates that dynamic topologies have the ability to overcome many learning limitations of static topologies.

The dissertation presents LITs for Competitive Learning, Counterpropagation, Backpropagation, and ASOCS AA2 models. Complexity analyses of each of the LIT versions of these models indicates reasonable performance characteristics. This collection of models is not intended to be exhaustive, but rather representative of general classes of similar models. The transformations given in the dissertation, likewise, are intended as guides to implementation strategy as much as anything else.

The prototype PASOCS chip, which is an artifact of this research, provides a proof-of concept implementation of LIT ideas. The design, implementation and testing results are very encouraging, and suggest a number of areas for future research. Two of the important ideas that resulted from this particular effort were: 1) the circuit design was very simple and no specialized circuitry was required, and 2) the PASOCS effort was successfully combined with an investigation into the development and use of Multi-Chip Modules as a general interconnect medium for ANNs.

Adaptive Real-Time Network Routing is one example of a potential application which requires, speed, adaptivity (learning or changing over time) and generalization capabilities of ANNs. It appears to be particularly suited to the dynamic topologies and incremental learning features of ASOCS models. However, our preliminary research indicates that in order for *any* adaptive scheme to be successful in adaptive routing, an internet protocol that makes use of consistent, hierarchical naming/addressing schemes must be in place. Otherwise, useful learning of adaptive routes is simply not possible.

2. Future Research

It is certainly clear in ANN literature that if connectivity constraints and certain other technological limitations could be eased or overcome, many issues and problems that arise in the hardware implementation of ANNs would disappear. Thus, it remains an interesting challenge to design and construct neurocomputing machines that can be used in handling applications both using current technology, and anticipating and driving toward, emerging and potential future technologies.

Both the areas of 1) developing LIT versions of other models and, 2) designing hardware implementations of LIT models, offer specific areas for continuing LIT research. Applications research, the third area discussed in this dissertation, is a much broader area, which may have important implications for neural networks in general, and is not specifically limited to LIT research *per se*. Furthermore, applications for LIT models follow by implication from those models for which LIT versions have been defined (i.e. the LIT Backpropagation model handles the same applications as the original Backpropagation model). Hence, applications research is not discussed in terms of future research specific to LIT.

With regard to the LIT models, the following offer areas for further research:

- **LIT(s) for Constraint Satisfaction Models**

  Given typical applications for Constraint Satisfaction models, such as Hopfield networks, the "problem size", or size of the network, is known beforehand. Thus, these networks do not appear to require the same types of dynamics as other networks, and no advantage in applying the LIT strategy is immediately obvious. If however, someone were to develop suitable dynamic versions of these models, further research would be warranted.

- **LIT for Recurrent Feedforward Networks (viz. Backpropagation)**

  Recurrent feedforward networks are often used for applications which involve time or context dependencies. Recurrence can be implemented in the LIT scheme using the fact that any broadcast to nodes in wave *i* (for example) is also available to any other nodes as well—in particular, node in waves previous to *i*. However, the required structures and operations remain to be precisely and formally specified.

- **Application of LIT to (non-neural) Machine Learning Models**

  Examples of models of interest in this area are Clark's CN2 [Cla89], Michalski's AQ [Mic86], and Quinlan's ID3 [Qui86].

- **Investigation of Hierarchical Topologies Other Than Binary Trees for LIT Communication Schemes**

  The binary tree topology is simple and fast. However, it is not fault-tolerant. Thus, investigation of more fault-tolerant topologies, such as the hypercube, would seem to be useful.

With regard to hardware implementations of LIT models, the following offer areas for continued research:

- **Improvements on the Prototype PASOCS Chip Design**

  As mentioned in the introduction to chapter 6, there are several aspects of the prototype design that could be improved, and one minor bug to be eliminated. It would be interesting to investigate the same conceptual design using other tools than those specified in the paper, and to investigate other non-conventional design schemes as well.

- **Design and Implementation of an LIA Chip**

  Another direction would be to implement an LIA chip design as well as PASOCS.

Appendix

not previously published

This Appendix contains code for the LIT versions of the static Competitive Learning (CL) model and the static standard Backpropagation (BP) model. This appendix has three sections. Section 1 contains code for the makefile, list-handling and data object primitives upon which the CL and BP simulations are built, using CFront C++. Section 2 contains code for LIT CL. Section 3 contains code for LIT BP.

1. List-Handling and Data Objects Primitives

1.1. Makefile

```
OBJS = list.o clnet.o valnode.o bpmain.o bp.o
LIBS = -L/usr/lib/X11R5 -lX11 -lm
#CFLAGS = -Aa-g
CFLAGS = -g
CC = CC
.SUFFIXES: .C

# without the .SUFFIXES line, the .C.o line will not work

.C.o:
	CC -c $(CFLAGS) $(INCLUDES) $<

bpmain: $(OBJS)
	$(CC) $(CFLAGS) $(OBJS) $(LIBS) -o bpmain

list.o: list.h
clnet.o: list.h valnode.h clnet.h
valnode.o: list.h valnode.h
bpmain.o: list.h valnode.h clnet.h bp.h
bp.o: list.h valnode.h bp.h
```

This particular makefile is set up to generate a BP network. To generate a CL network, "bpmain" should be changed to "clmain".

1.1.1. Bpmain

```
#include "list.h"
#include "valnode.h"
#include "bp.h"

Randval randx;
extern FILE *iop;

void main() {

  Bpnet    *Net;
  BptrainSet *data;
  BptestSet *rundata;


  Net = new Bpnet("bp.net");
  (*Net).read_weights("bp342.wts");
```

```
  (*Net).Print();
  (*Net).save_weights("lit342.wts");
  data = new BptrainSet("lit342.pat");
  rundata = new BptestSet("lit342t.pat");
  (*Net).train(data);
  (*Net).save_weights("litfin.wts");
  (*Net).Print();
  (*Net).test(rundata);
}
```

1.1.2. Clmain

```
#include "List.h"

Randval randx;
extern FILE *iop;

void main() {

  Clnet    *Cnet;
  ClPatternSet *ps;

  Cnet = new Clnet("cl.net");
  (*Cnet).read_weights("cl.wts");
  fprintf(iop, "Original network\n");
  (*Cnet).Print();
  ps = new ClPatternSet("cl.pat");
  (*ps).Print();
  (*Cnet).train(ps);
  fprintf(iop, "Trained network\n");
  (*Cnet).Print();
  (*Cnet).test(ps);
  delete ps;
  delete Cnet;

}
```

1.2 List.h, .C and Valnode.h, .C files

1.2.1 list.h

```
#define volatile
#define signed
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <assert.h>

#define rnd() ((float)rand()*0.4656612875e-9)
```

```
char *String(char *);
void itoa(int, char [], int);
float sigmoid(float);
float sigmoidprime(float);

class Listnode {
   friend class List;
 protected:
   Listnode *next;
   Listnode *prev;
 public:
   Listnode();
   ~Listnode();
   virtual void Print();
};

class List {
 private:
   Listnode *first;
   Listnode *last;
   Listnode *current;
   int position;
   int size;

 public:
   List();
    ~List();
     int listlen();
    void FindFirst();
    void FindLast();
    void FindNext();
    void FindPrev();
    void Findith(int);
   Listnode* get_current();
    void InsertBefore(Listnode *node);
    void InsertAfter(Listnode *node);
   Listnode *Remove();
    void RemX();
   virtual void Print();
   virtual void Print(char *Filename);

};

class Randval {
 private:
   int random_seed;

 public:
  Randval();
  ~Randval();
   void randomize();
   void reseed();
   float genval();
  int genint(int);
```

```
 int genint(int, int);
};
```

1.2.2 List.C

```
#include "list.h"

FILE *iop = stdout;

float sigmoid(float x) {
   return(1.0 / (1.0 + (float) exp( (double) ((-1.0) * x))));
}

float sigmoidprime(float x) {
   float sx;
   sx = sigmoid(x);
   return ((sx)*(1-sx));
}

char *String(char *s) {
   int len;
   char *str;

      len = strlen(s);
      str = new char[len+1];
      assert (str != 0);
      strcpy(str, s);
      return (str);
}

void itoa(int n, char s[], int radix) {
  int i, j, sign, c;

  if ((sign = n) < 0) n = -n;
  i = 0;
  do {
     s[i++] = (n % radix) + '0';
  } while ((n/= radix) > 0);
  if (sign < 0) s[i++] = '-';
  s[i++] = '\0';

  for (i=0, j=strlen(s)-1; i<j; i++,j--) {
     c =s[i];
     s[i] = s[j];
     s[j] = c;
  }
}


List::List() {
   first = 0;
   last = 0;
   current = 0;
```

```
  position = 0;
  size = 0;
}

List::~List() {

  while (first != 0) {
    current = first;
    first = first->next;
    delete current;
  }
  first = last = current = 0;
  position = size = 0;
}

int List::listlen() {
  return size;
}

void List::FindFirst() {
  current = first;
  position = 1;
}

void List::FindLast() {
  current = last;
  position = size;
}

void List::FindNext() {

  if (current == 0) fprintf(iop, " {NIL} ");
  else current = current->next;

}

void List::FindPrev() {

  if (current == 0) fprintf(iop, " {NIL} ");
  else current = current->prev;

}

void List::Findith(int pos) {
  Listnode *temp;
  int i;

  if ((pos < 1) || (pos > size)) printf("position %d is out of range.\n");
  else {
    for (i=1, temp = first; i < pos; i++) temp = temp->next;
    position = pos;
    current = temp;
  }
```

```
}

Listnode* List::get_current() {
  Listnode* temp;

  temp = current;
  return temp;
}

void List::InsertBefore(Listnode *node) {
  Listnode *before;

  if (first == 0) {
    first = node;
    last = node;
    current = node;
    position++;
    size++;
    return;
  }
  if (current == first) {
//node->prev is 0
     node->next = current;
     current->prev = node;
     current = node;
     first = node;
    size++;
    return;
  }
    before = current->prev;
    node->prev = before;
    node->next = current;
    before->next = node;
    current->prev = node;
    current = node;
    size++;
}

void List::InsertAfter(Listnode *node) {
  Listnode *after;

  if (first == 0) {
    first = node;
    last = node;
    current = node;
    position++;
    size++;
    return;
  }
  if (current == last) {
//node->next = 0
    node->prev  = current;
    current->next = node;
    last = node;
```

```
    current = node;
    size++;
    position++;
    return;
  }
  after = current->next;
  node->prev = current;
  node->next = after;
  current->next = node;
  after->prev = node;
  current = node;
  size++;
  position++;
}

Listnode *List::Remove() {
  Listnode *temp, *before, *after;

  if (first == 0) return 0;

  if (current = first) {
    first = current->next;
    first->prev = 0;
    current->next = 0;
    temp = current;
    current = first;
    size--;
    return (temp);
   }

  if (current = last) {
    last = current->prev;
    last->next = 0;
    current->prev = 0;
    temp = current;
    current = last;
    size--;
    position--;
    return (temp);
   }

   before = current->prev;
   after = current->next;
   temp = current;
   before->next = after;
   after->prev = before;
   current->prev = 0;
   current->next = 0;
   current = before;
   return(temp);
}

void List::RemX() {
  Listnode *X;
```

```
   X = (*this).Remove();
   delete X;
}

void List::Print() {
  Listnode *temp;

  if (first != 0) {
     fprintf(iop, "( ");
     for (temp = first; temp != 0; temp = temp->next)
        temp->Print();

     fprintf(iop, ")\n");
     }
  else
     fprintf(iop, "{empty}\n");
}

void List::Print(char *FileName) {
 FILE *opf;

 if ((opf = fopen(FileName, "w")) == NULL) {
   printf("\nALERT: Unable to open file \"%s\" for output.\n", FileName);

   return;
  }
 iop = opf;
 (*this).Print();
 fclose (iop);
 iop = stdout;
}

Listnode::Listnode() {
   next = 0;
   prev = 0;
}

Listnode::~Listnode() {
   next = 0;
   prev = 0;
}

Randval::Randval() {
   random_seed = rand();
   srand(random_seed);
}

Randval::genint(int low, int high) {
    float   answer;
    float   randf;
    int     range;

    randf = rnd();
```

```
    range = high - low + 1;
    answer = randf * range + low;
    return((int)answer);
}

Randval::genint(int high) {
  return (Randval::genint(0,high));
}

Randval::~Randval() {
}

void Randval::randomize() {
 srand(random_seed);
}

void Randval::reseed() {
  random_seed = rand();
  srand(random_seed);
}

float Randval::genval() {

    return (rnd());
}

//the following are virtual hooks for inheritance purposes
void Listnode::Print() {

}
```

1.2.3 Valnode.h

```
class Intnode: public Listnode {
 private:
   int value;
   char *display_parms;

 public:
   Intnode();
   Intnode(int);
   ~Intnode();
   void Setval(int);
   int getval();
   void Print();
   void set_display_parms(char *);
};

class Realnode: public Listnode {
 private:
   float value;
   char *display_parms;
```

```
 public:
   Realnode();
   Realnode(float);
   ~Realnode();
    void Setval(float);
    float getval();
   Realnode *operator+=(Realnode *);
   Realnode *operator-(Realnode *);

    void Print();
    void set_display_parms(char *);
};


class Intlist: public List {
 public:
    Intlist();
     Intnode *get_current();
     Intnode *Remove();
};


class Reallist: public List {
 public:
   Reallist();
   Realnode *get_current();
   Realnode *Remove();
   Reallist *operator+=(Reallist *);
   Reallist *operator-(Reallist *);
   float ss();
  };

class RWnode: public Realnode {
private:
   float dweight;
   float lrate;
   float momentum;
public:
   RWnode();
   RWnode(float, float,float,float);
   float getdweight();
   float getlrate();
   float getmu();
   void setdweight(float);
   void setlrate(float);
   void setmu(float);
};

class RWlist : public List {
public:
  RWlist();
  RWnode *get_current();
  RWnode *Remove();
};
```

1.2.4 Valnode.C

```
#include "list.h"
#include "valnode.h"

extern FILE *iop;

//////////// Realnode stuff ///////////////////////////

Realnode::Realnode(): Listnode(){
   display_parms = String("%3.3f ");
   value = 0.0;
}

Realnode::Realnode(float val): Listnode(){
   display_parms = String("%3.3f ");
   value = val;
}

void Realnode::Setval(float val) {
   value = val;
}

void Realnode::set_display_parms(char *s) {
  delete []display_parms;
  display_parms = String(s);
}

float Realnode::getval() {
   return value;
}

Realnode *Realnode::operator+=(Realnode *dval) {

   value += (*dval).getval();
   return this;
}

Realnode *Realnode::operator-(Realnode *extra) {
   float result;
   Realnode *node;

   result = value - (*extra).getval();
   node = new Realnode(result);
   return node;
}

void Realnode::Print() {
   fprintf(iop, display_parms, value);
}
```

```
//////////////// Reallist stuff //////////////////////
Reallist::Reallist(): List() {

}

Realnode *Reallist::get_current() {
  return ((Realnode *) List::get_current());
 }
Realnode *Reallist::Remove() {
    return ((Realnode *)List::Remove());
}

Reallist *Reallist::operator+=(Reallist *partial) {

int len = (*this).listlen();
int i;
Realnode *left;
Realnode *right;

   for (i=1, FindFirst(), (*partial).FindFirst();
         i <= len; i++, FindNext(), (*partial).FindNext()) {
      left = get_current();
      right = (*partial).get_current();
      (*left) += right;
     }
   return this;
}

Reallist *Reallist::operator-(Reallist *extra) {

int len = listlen();
int i;
Realnode *left, *right, *diff;
Reallist *difflist = new Reallist();

   for (i=1, FindFirst(), (*extra).FindFirst();
         i <= len; i++, FindNext(), (*extra).FindNext()) {
      left = get_current();
      right = (*extra).get_current();
      diff = (*left) - right;
      (*difflist).InsertAfter(diff);
     }
   return (difflist);
}
float Reallist::ss() {
   int i, len;
   Realnode *place;
   float dval, sum = 0;

   len = listlen();
   for(i=1,FindFirst();i<=len;i++,FindNext()) {
      place = get_current();
      dval = (*place).getval();
      sum += (dval*dval);
```

```
   }
  return sum;
}
////////////////////// more Realnode stuff //////////////////////

Realnode::~Realnode() {
 delete []display_parms;
}

////////////////////////// Intnode stuff /////////////////

Intnode::Intnode(): Listnode(){
  display_parms = String("%d ");
  value = 0;
}

Intnode::Intnode(int val): Listnode(){
  display_parms = String("%d ");
  value = val;
}

Intnode::~Intnode() {
  delete []display_parms;
}
int Intnode::getval() {
  return value;
}

void Intnode::Setval(int val) {
  value = val;
}

void Intnode::set_display_parms(char *s) {
 delete []display_parms;
 display_parms = String(s);
}

void Intnode::Print() {
 fprintf(iop, display_parms, value);
}

////////////////////  Intlist stuff ////////////////

Intlist::Intlist(): List() {

}

////////////////// more Intnode stuff //////////////////
Intnode *Intlist::get_current() {
 return ((Intnode *)List::get_current());
 }

Intnode *Intlist::Remove() {
   return ((Intnode *) List::Remove());
```

```
}

////////////////  RW node stuff: RWnode is for real weights, a subclass
///////////////   of Realnode class //////////////////////////////////

RWnode::RWnode() : Realnode() {
   dweight = 0;
   lrate = 0;
   momentum = 0;
}

RWnode::RWnode(float val, float dw, float lr,
             float mu) : Realnode(val) {
   dweight = dw;
   lrate = lr;
   momentum = mu;
}

float RWnode::getdweight() {
  return dweight;
}

float RWnode::getlrate() {
  return lrate;
}

float RWnode::getmu() {
  return momentum;
}

void RWnode::setdweight(float dval) {
  dweight = dval;
}

void RWnode::setlrate(float rate) {
  lrate = rate;
}

void RWnode::setmu(float muval) {
  momentum = muval;
}

//////////////////// RWlist stuff: RWlists hold RWnodes ///////////////////
/* currently, RWlist is a direct descendent of List, not a descendent of
   Reallist.  Maybe the inheritance should be changed, because RWnodes
   inherit from Realnodes, not directly from Listnodes.
*/

RWlist::RWlist() : List() {
}

RWnode *RWlist::get_current() {
  return ((RWnode *) List::get_current());
}
```

```
RWnode *RWlist::Remove() {
  return ((RWnode *) List::Remove());
}
```

2. The CL model

2.1. Clnet.h

```
class ClPattern : public Listnode {
 private:
   char *name;
   Intlist *P;
 public:
   ClPattern();
   ClPattern(char *, Intlist*);
   ~ClPattern();
   char *get_name();
   void set_name(char *);
   Intlist *get_pattern();
   void set_pattern(Intlist *);
   virtual void Print();
};

class ClPatternSet : public List {
 private:  //no additional fields
 public:
   ClPatternSet();
   ClPatternSet(char *);
   ClPattern *get_current();
   ClPattern *Remove();
};

class Clnode: public Listnode {
 private:
float activation;
   Reallist weights;
   int classNum;
   char *Name;
 public:
   Clnode();
   Clnode(Reallist*, int, char *);
   ~Clnode();
   float Compute_activation(Intlist *);
   void set_activation(float);
   float get_activation();
   int get_class();
   char *getname();
   Reallist *getweights();
   void setweights(Reallist *);
   void Changeweights(Intlist *, float);
   virtual void Print();
```

```
};

class Clnet: public List {
 private:
   float lrate;
   Clnode *winner;
   int ninputs;
   int nepochs;
 public:
   Clnet();
   Clnet(char *fname);
   Clnode *get_current();
   Clnode *Remove();
   void Compute_class(Intlist *);
   float get_lrate();
   void set_lrate(float val);
   void update_winner(Intlist *);
   void read_weights(char *);
   void save_weights(char *);
   int getnepochs();
   void setnepochs();
   void train(ClPatternSet *);
   void test(ClPatternSet *);
};
```

2.2 Clnet.C

```
#include "list.h"
#include "valnode.h"
#include "clnet.h"

extern FILE *iop;
extern Randval randx;  //defined in mymain.C

Clnode::Clnode():Listnode() {
   activation = 0.0;
   classNum = 0;
   Name = 0;
}

Clnode::Clnode(Reallist *w, int clindex, char *s) :Listnode() {

   if (w != 0) weights = (*w);
   classNum = clindex;
   if (s != 0) Name = String(s);
}

Clnode::~Clnode() {
  delete [] Name;
  Name = 0;
}
```

```
float Clnode::Compute_activation(Intlist *input) {
   int len;
   int i;
   int i_val;
   float w_val;
   float sum = 0.0;

   len = weights.listlen();
   weights.FindFirst();
   (*input).FindFirst();
   for (i=1; i <= len; i++) {
        i_val=((*input).get_current())->getval();
        if (i_val != 0) {
           w_val = (weights.get_current())->getval();
           sum += i_val*w_val;
         }
   weights.FindNext();
   (*input).FindNext();
   }
   activation = sum;
return(activation);
}

float Clnode::get_activation() {
  return activation;
}

void Clnode::set_activation(float val) {
   activation = val;
}

Clnode::get_class() {
   return(classNum);
}

void Clnode::Changeweights(Intlist *input, float lrate) {

   int i;
   float nactive = 0;
   int len = (*input).listlen();
   float w_val;
   int i_val;

    for (i = 1, (*input).FindFirst(); i <= len; i++, (*input).FindNext())
           if (((*input).get_current())->getval()) nactive += 1;

    if(nactive == 0) return;

    for (i = 1, weights.FindFirst(),(*input).FindFirst();
         i <= len; i++, weights.FindNext(), (*input).FindNext()) {
       w_val = (weights.get_current())->getval();
       i_val = ((*input).get_current())->getval();
       w_val += lrate * ((i_val/nactive) - w_val);
       (weights.get_current())->Setval(w_val);
```

```
    }
}

Reallist *Clnode::getweights() {
  return &weights;
}

void Clnode::setweights(Reallist *newlist) {
   delete (&weights);
   weights = (*newlist);
}

char *Clnode::getname() {
   return Name;
}
void Clnode::Print() {
  if (Name != 0) fprintf(iop, "%s  %3.3f ", Name, activation);
  else fprintf(iop, "C%d ", classNum);
  weights.Print();
}

//  this next section implements a cl network as a list of clnodes

Clnet::Clnet() : List() {
   ninputs = 0;
   lrate = 0.5;
   winner = 0;
   nepochs = 0;
}

Clnet::Clnet(char *fname): List() {

   FILE *fp;

   char label[80];
   char cname[80];
   char s[74];
   int nnodes;
   int i, j;
   float *newweight;
   float vectlen = 0;

   Clnode *newnode;
   Realnode *rnode;
   Reallist *wts;

   winner = 0;

   if ((fp = fopen(fname, "r")) == NULL)
      printf("\nALERT: Unable to open file %s for input.\n");
   else {
       fscanf(fp, "%s %d", label, &nnodes);
       fscanf(fp, "%s %d", label, &ninputs);
       fscanf(fp, "%s %f", label, &lrate);
```

```
    fscanf(fp, "%s %d", label, &nepochs);

    newweight = new float[ninputs];
    randx.randomize();

    for (i=1; i<=nnodes; i++) {
      wts = new Reallist();
      for (j=1; j<= ninputs; j++) {
        newweight[j] = randx.genval();
        vectlen += newweight[j];
         }

      for (j=1; j<= ninputs; j++) {
       newweight[j] *= 1.0/vectlen;
       rnode = new Realnode(newweight[j]);
       (*wts).InsertAfter(rnode);
      }
      vectlen = 0;
      strcpy(cname, "Class ");
      itoa(i,s,10);
      strcat(cname, s);
      newnode = new Clnode(wts, i, cname);
      (*this).InsertAfter(newnode);
    }
  }
fclose(fp);
delete []newweight;
}

Clnode *Clnet::get_current() {
  return ((Clnode *)List::get_current());
}

Clnode *Clnet::Remove() {
  return ((Clnode *)List::Remove());
}

void Clnet::Compute_class(Intlist *input) {

  int len = (*this).listlen();
  float curract, maxact = 0;
  int classofmax = 0;
  int i;
  Clnode *place;

 (*this).FindFirst();

  for (i = 1; i <= len; i++) {
       curract = ((*this).get_current())->Compute_activation(input);
       if (curract > maxact) {
         maxact = curract;
         classofmax = ((*this).get_current())->get_class();
        }
  (*this).FindNext();
```

```
    }

    (*this).FindFirst();
    for (i =1; i <= len; i++) {
        place = (*this).get_current();
            if ((*place).get_class() == classofmax) {
                (*place).set_activation(1);
                winner = place;
            }
            else (*place).set_activation(0);
            (*this).FindNext();
    }
}

float Clnet::get_lrate() {
  return lrate;
}

void Clnet::set_lrate(float val) {
   lrate = val;
}

void Clnet::update_winner(Intlist *input) {
   (*winner).Changeweights(input, lrate);
}

void Clnet::read_weights(char *fname) {

FILE *fp;
float w_val;
int i,j;
Reallist *wlist;

   if ((fp = fopen(fname, "r")) == NULL)
      printf("\nALERT: Unable to open file %s for input.\n");
   else {
       for (i= 1,(*this).FindFirst(); i<= (*this).listlen();
       i++, (*this).FindNext()) {
           wlist = ((*this).get_current())->getweights();
         for (j=1,(*wlist).FindFirst(); j<= (*wlist).listlen();
         j++, (*wlist).FindNext()) {
            fscanf(fp, "%f", &w_val);
            ((*wlist).get_current())->Setval(w_val);
             }
       }
   }
fclose(fp);
}

void Clnet::save_weights(char *fname) {
   FILE *fp;
   float w_val;
   Reallist *wlist;
   int i,j;
```

```
  if ((fp = fopen(fname, "w")) == NULL)
    printf("\nALERT: Unable to open file %s for output.\n");
  else {
     for (i=1,(*this).FindFirst();i<=(*this).listlen();
     i++, (*this).FindNext()) {
       wlist = ((*this).get_current())->getweights();
       for (j=1,(*wlist).FindFirst(); j<= (*wlist).listlen();
       j++, (*wlist).FindNext()) {
         w_val = ((*wlist).get_current())->getval();
         fprintf(fp, "%f ", w_val);
          }
     }
  }
 fclose(fp);
}

void Clnet::train(ClPatternSet *ts) {

  int epochno, i;
  int npatts = (*ts).listlen();
  ClPattern *cp;
  Intlist *input;

  for (epochno = 1; epochno <= nepochs; epochno++) {
    for (i=1,(*ts).FindFirst(); i<= npatts; i++, (*ts).FindNext()) {
      cp = (*ts).get_current();
      input = (*cp).get_pattern();
      (*this).Compute_class(input);
      (*this).update_winner(input);
    }
  }
}

void Clnet::test(ClPatternSet *ts) {

  int i;
  int npatts = (*ts).listlen();
  ClPattern *cp;
  Intlist *input;
  char *pname, *cname;

  for (i=1,(*ts).FindFirst(); i<= npatts; i++, (*ts).FindNext()) {
    cp = (*ts).get_current();
    pname = (*cp).get_name();
    input = (*cp).get_pattern();
    (*this).Compute_class(input);
    cname = (*winner).getname();
    fprintf(iop, "%s %s\n", pname, cname);
  }
}

//Pattern is the class def for both training patterns and test patterns
ClPattern::ClPattern() : Listnode() {
```

```
  name = 0;
  P = 0;
}
ClPattern::ClPattern(char *s, Intlist *L) : Listnode() {
  name = String(s);
  P = L;
}

ClPattern::~ClPattern() {
  delete []name;
  delete P;
}

char *ClPattern::get_name() {
  char *temp;
   temp = String(name);
  return (temp);
}
void ClPattern::set_name(char *s) {
  delete []name;
  name = String(s);
}

Intlist *ClPattern::get_pattern() {
   return (P);
}

void ClPattern::set_pattern(Intlist *newP) {
  delete (P) ;
  P = newP;
}

void ClPattern::Print() {
   fprintf(iop, "%s ", name);
   (*P).Print();
}


ClPatternSet::ClPatternSet() : List() {
}

ClPatternSet::ClPatternSet(char *fname) : List() {

   FILE *fp;
   int ninputs;
   char label[80];

   int i;
   char name[80];
   int newval;
   Intnode *inode;
   Intlist *ilist;
   ClPattern *pnode;
```

```
  if ((fp = fopen(fname, "r")) == NULL)
    printf("\nALERT: Unable to open file %s for input.\n");
  else {
    fscanf(fp, "%s %d", label, &ninputs);

    while (fscanf(fp, "%s ", name) != EOF) {
      ilist = new Intlist;
      for (i=1; i <=ninputs; i++) {
       fscanf(fp, "%d ",  &newval);
       inode = new Intnode(newval);
       (*ilist).InsertAfter(inode);
       }
      pnode = new ClPattern(name, ilist);
      (*this).InsertAfter(pnode);
    }

  } //end else
}
ClPattern *ClPatternSet::get_current() {
  return ((ClPattern *)List::get_current());
}

ClPattern *ClPatternSet::Remove() {
  return ((ClPattern *)List::Remove());
}
```

3. The BP model

3.1. Bp.h

```
class BptestPatt: public Listnode {
 private:
  char *name;
  Reallist *ip;
 public:
  BptestPatt();
  BptestPatt(char *, Reallist*);
  ~BptestPatt();
  char *getname();
  void setname(char *);
  Reallist *getpattern();
  void setpattern(Reallist *);
  virtual void Print();
};

class BptrainPatt: public Listnode {
 private:
  char *name;
  Reallist *ip;
  Reallist *tp;
 public:
  BptrainPatt();
```

```
  BptrainPatt(char *, Reallist*, Reallist *);
  ~BptrainPatt();
  char *getname();
  void setname(char *);
  Reallist *getipattern();
  void setipattern(Reallist *);
  Reallist *gettpattern();
  void settpattern(Reallist *);
  virtual void Print();
};

class BptestSet : public List {
 private:  //no additional fields
 public:
  BptestSet();
  BptestSet(char *);
  BptestPatt *get_current();
  BptestPatt *Remove();
  Reallist *getpatt();
};

class BptrainSet : public List {
 private:  //no additional fields
 public:
  BptrainSet();
  BptrainSet(char *);
  BptrainPatt *get_current();
  BptrainPatt *Remove();
  Reallist *getipatt();
  Reallist *gettpatt();
};

class Bpnode: public Listnode {
 protected:
  float activation;
  float netin;
  RWnode inbias;
  RWlist inwts;
  RWlist outwts;
  Reallist outvals;
  int nodenum;
  char *Name;

 public:
  Bpnode();
  Bpnode(RWlist*,RWlist*,int,RWnode *,char *);
  ~Bpnode();
  virtual Reallist* Compute_output(Reallist *);
  Reallist *getoutput();
  int getnodenum();
  char *getname();
  RWlist *getiwts();
  RWlist *getowts();
  void setiwts(RWlist *);
```

```
  void setowts(RWlist *);
  float getbias();
  void setbias(float);
  virtual void Changeweights(Reallist *,Reallist *);
  virtual void setlrate(float);
  virtual void setmu(float);
  virtual void Print();
};

class Bpbiasnode: public Bpnode {

 public:
  Bpbiasnode();
  Bpbiasnode(RWlist*, int, char *s);
  virtual Reallist* Compute_output(Reallist *);
  void setiwts(RWlist *);
  float getbias();
  void setbias(float);
  virtual void setlrate(float);
  virtual void setmu(float);
  virtual void Changeweights(Reallist*, Reallist *);
  virtual void Print();
};

class Bpnet: public List {
 private:
  float lrate;
  float momentum;
  int ninputs;
  int noutputs;
  int nepochs;
  float errthresh;
 public:
  Bpnet();
  Bpnet(char *fname);
  Bpnode *get_current();
  Bpnode *Remove();
  Reallist *Compute_output(Reallist *);
  float geterrthresh();
  void seterrthresh(float);
  float get_lrate();
  void set_lrate(float);
  float getmu();
  void set_mu(float);
  void read_weights(char *);
  void save_weights(char *);
  int getnepochs();
  void setnepochs(int);
  int getnnodes();
  void train(BptrainSet *);
  void test(BptestSet *);
};
```

3.2 Bp.C

```
#include "list.h"
#include "valnode.h"
#include "bp.h"

extern FILE *iop;
extern Randval randx;

///////////////////////// individual Bp nodes stuff ////////////////////
Bpnode::Bpnode() : Listnode() {
  nodenum = 0;
  activation = 0;
  netin = 0;
  Name = 0;
}

Bpnode::Bpnode(RWlist *iw,RWlist *ow,int newnum,
            RWnode *newbias, char *s) : Listnode() {

inbias = (*newbias);
nodenum = newnum;
activation = (*newbias).getval();
netin = 0;
if (s != 0) Name = String(s);
if (iw != 0) inwts = (*iw);
if  (ow != 0) outwts = (*ow);
// list outvals is empty except after valid computation

}

Bpnode::~Bpnode() {
  delete []Name;
}

Reallist *Bpnode::Compute_output(Reallist *input) {
  int len;
  int i;
  float w_val, i_val, o_val;
  Reallist *temp = new Reallist();
  Realnode *outnode;

  netin = inbias.getval();
  len = (*input).listlen();
  for (i=1, (*input).FindFirst(), inwts.FindFirst(); i <= len;
   (*input).FindNext(), inwts.FindNext(), i++) {
     i_val=((*input).get_current())->getval();
     if (i_val != 0) {
       w_val = (inwts.get_current())->getval();
       netin += i_val*w_val;
     }
  }
```

```
   activation = sigmoid(netin);
   len = outwts.listlen();
   for (i=1, outwts.FindFirst(); i <= len;
    outwts.FindNext(), i++) {
      w_val = (outwts.get_current())->getval();
      o_val = activation*w_val;
      outnode = new Realnode(o_val);
      (*temp).InsertAfter(outnode);
   }
   outvals = (*temp);
return(temp);
}

Reallist *Bpnode::getoutput() {
   return (&outvals);
}

int Bpnode::getnodenum() {
   return nodenum;
}

char *Bpnode::getname() {
  return Name;
}

RWlist *Bpnode::getiwts() {
   return (&inwts);
}

RWlist *Bpnode::getowts() {
   return (&outwts);
}

void Bpnode::setiwts(RWlist *wts) {
   inwts = (*wts);
}

void Bpnode::setowts(RWlist *wts) {
   outwts = (*wts);
}

float Bpnode::getbias() {
   return inbias.getval();
}

void Bpnode::setbias(float newbias) {
   inbias.Setval(newbias);
   inbias.setdweight(0);
// node lrate and momentum remain unchanged
}
void Bpnode::setlrate(float newval) {
   int i, len;
   RWnode *place;
```

```
  len = outwts.listlen();
  for (i=1, outwts.FindFirst(); i <= len; i++, outwts.FindNext()) {
     place = outwts.get_current();
     (*place).setlrate(newval);
  }

  len = inwts.listlen();
  for (i=1, inwts.FindFirst(); i <= len; i++, inwts.FindNext()) {
     place = inwts.get_current();
     (*place).setlrate(newval);
  }
}
void Bpnode::setmu(float newval) {
  int i, len;
  RWnode *place;

  len = outwts.listlen();
  for (i=1, outwts.FindFirst(); i <= len; i++, outwts.FindNext()) {
     place = outwts.get_current();
     (*place).setmu(newval);
  }

  len = inwts.listlen();
  for (i=1, inwts.FindFirst(); i <= len; i++, inwts.FindNext()) {
     place = inwts.get_current();
     (*place).setmu(newval);
  }
}
void Bpnode::Changeweights(Reallist *input, Reallist *oerrdelta) {

int i, len;
float wed, dw;
float wval, oedval, ival;
float ierrdelta, tempidelta=0;
Realnode *ocp, *icp;
RWnode *wcp;

  len  = outwts.listlen();

  for (i=1,outwts.FindFirst(),(*oerrdelta).FindFirst(); i <=len;
       i++, outwts.FindNext(), (*oerrdelta).FindNext()) {
    ocp = (*oerrdelta).get_current();
    wcp = outwts.get_current();
    oedval = (*ocp).getval();
    wval = (*wcp).getval();
    wed = activation*oedval;
    dw = (*wcp).getlrate()*wed+(*wcp).getmu()*(*wcp).getdweight();
    (*wcp).Setval(wval+dw);
    (*wcp).setdweight(dw);
    tempidelta += (oedval*wval);
  }
  ierrdelta = tempidelta*sigmoidprime(netin);

  len = inwts.listlen();
```

```
  for (i=1, inwts.FindFirst(), (*input).FindFirst(); i <=len;
      i++, inwts.FindNext(), (*input).FindNext()) {
    wcp = inwts.get_current();
    icp = (*input).get_current();
    wval = (*wcp).getval();
    ival = (*icp).getval();
    wed = ival*ierrdelta;
    dw = (*wcp).getlrate()*wed+(*wcp).getmu()*(*wcp).getdweight();
    (*wcp).Setval(wval+dw);
    (*wcp).setdweight(dw);
  }
  wval = inbias.getval();
  wed = ierrdelta;
  dw = inbias.getlrate()*wed+inbias.getmu()*inbias.getdweight();
  inbias.Setval(wval+dw);
  inbias.setdweight(dw);

}

void Bpnode::Print() {
 inwts.Print();
 inbias.Print();
 outwts.Print();
 outvals.Print();
}

/////////////// stuff for biasnode in lit bp ///////////////////////

Bpbiasnode::Bpbiasnode() : Bpnode() {
     activation = 1.0;
}

Bpbiasnode::Bpbiasnode(RWlist *ow, int newnum, char *s) : Bpnode() {
  nodenum = newnum;
  activation = 1.0;
  if  (ow != 0) outwts = (*ow);
  if (s != 0) Name = String(s);

// list outvals is empty except after valid computation

}

Reallist *Bpbiasnode::Compute_output(Reallist* inputs) {
 int len;
  int i;
  float o_val;
  Reallist *temp = new Reallist();
  Realnode *outnode;

  activation = 1.0;

  len = outwts.listlen();
  for (i=1, outwts.FindFirst(); i <= len;
   outwts.FindNext(), i++) {
```

```
    o_val = (outwts.get_current())->getval();
    outnode = new Realnode(o_val);
    (*temp).InsertAfter(outnode);
  }
  outvals = (*temp);
return(temp);
}

void Bpbiasnode::setiwts(RWlist *wts) {
//override Bpnode memeber function
//do nothing
}

float Bpbiasnode::getbias() {
  return 0.0;
}

void Bpbiasnode::setbias(float newbias) {
//override Bpnode memeber function
//do nothing
}

void Bpbiasnode::Changeweights(Reallist *input, Reallist *oerrdelta) {

int len, i;
float wed, dw;
float wval, oedval;
Realnode *ocp;
RWnode *wcp;

  len  = outwts.listlen();

  for (i=1,outwts.FindFirst(),(*oerrdelta).FindFirst(); i <=len;
       i++, outwts.FindNext(), (*oerrdelta).FindNext()) {
    ocp = (*oerrdelta).get_current();
    wcp = outwts.get_current();
    oedval = (*ocp).getval();
    wval = (*wcp).getval();
    wed = oedval;
    dw = (*wcp).getlrate()*wed+(*wcp).getmu()*(*wcp).getdweight();
    (*wcp).Setval(wval+dw);
    (*wcp).setdweight(dw);
  }
}

void Bpbiasnode::setlrate(float newval) {
  int i, len;
  RWnode *place;

  len = outwts.listlen();
  for (i=1, outwts.FindFirst(); i <= len; i++, outwts.FindNext()) {
     place = outwts.get_current();
     (*place).setlrate(newval);
  }
```

```
}

void Bpbiasnode::setmu(float newval) {
  int i, len;
  RWnode *place;

  len = outwts.listlen();
  for (i=1, outwts.FindFirst(); i <= len; i++, outwts.FindNext()) {
     place = outwts.get_current();
     (*place).setmu(newval);
  }
}

void Bpbiasnode::Print() {
 outwts.Print();
}

/////////////////////// Bpnet stuff ///////////////////////
Bpnet::Bpnet(): List() {
  lrate = 0;
  momentum = 0;
  ninputs = 0;
  noutputs =0;
  nepochs = 0;
  errthresh = 0;
}

Bpnet::Bpnet(char *fname): List() {
  FILE *fp;

  char label[80];
  int nnodes;
  int i, j;
  float *newinputs, newoutputs;

  Bpnode *newnode;
  RWlist *iwts, *owts, *outbiases;
  RWnode *biaswt;
  RWnode *wtnode;

  if ((fp = fopen(fname, "r")) == NULL) {
    printf("\nALERT: Unable to open file %s for input.\n", fname);
    return;
  }
     fscanf(fp, "%s %d", label, &nnodes);
     fscanf(fp, "%s %d", label, &ninputs);
     fscanf(fp, "%s %d", label, &noutputs);
     fscanf(fp, "%s %f", label, &lrate);
     fscanf(fp, "%s %f", label, &momentum);
     fscanf(fp, "%s %d", label, &nepochs);
     fscanf(fp, "%s %f", label, &errthresh);

     randx.randomize();
```

```
      for (i=1; i<=nnodes; i++) {
        iwts = new RWlist();
        owts = new RWlist();
        biaswt = new RWnode(randx.genval(), 0, lrate, momentum);
        for (j=1; j<=ninputs; j++) {
          wtnode = new RWnode(randx.genval(), 0, lrate, momentum);
         (*iwts).InsertAfter(wtnode);
        }
        for (j=1; j<=noutputs; j++) {
          wtnode = new RWnode(randx.genval(), 0, lrate, momentum);
         (*owts).InsertAfter(wtnode);
        }

         newnode = new Bpnode(iwts, owts, i, biaswt, 0);
        (*this).InsertAfter(newnode);
      }
    outbiases = new RWlist();
    for (j=1; j<=noutputs; j++) {
      wtnode = new RWnode(randx.genval(), 0, lrate, momentum);
       (*outbiases).InsertAfter(wtnode);
     }
     newnode = new Bpbiasnode(outbiases, nnodes+1, 0);
     (*this).InsertAfter(newnode);
     nnodes++;

  fclose(fp);
}


Bpnode *Bpnet::get_current() {
 return ((Bpnode *)List::get_current());
}

Bpnode *Bpnet::Remove() {
 return ((Bpnode *)List::Remove());
}

Reallist *Bpnet::Compute_output(Reallist *inputs) {

int len;
int i;

Reallist *netouts = new Reallist();
Reallist *nodeouts;
Realnode *temp;
Bpnode *place;
float val;

//creates a list of real nodes with value 0
  for (i=1; i<= noutputs; i++) {
    temp = new Realnode(0);
    (*netouts).InsertAfter(temp);
  }
```

```
//sums together all the partial outputs generated by each node
  len = listlen();
  for (i=1, FindFirst(); i <=len; i++,FindNext()) {
   place = get_current();
   nodeouts = (*place).Compute_output(inputs);
   (*netouts) += nodeouts;
   }

//replaces the net value for each output value with the sigmoid activation
  for (i=1,(*netouts).FindFirst(); i <= noutputs; i++,(*netouts).FindNext()) {
    temp = (*netouts).get_current();
    val = (*temp).getval();
    val = sigmoid(val);
    (*temp).Setval(val);
   }
  return(netouts);
}

float Bpnet::get_lrate() {
 return lrate;
}

void Bpnet::set_lrate(float newval) {
  int i, len= listlen();
  Bpnode *place;

  lrate = newval;
  for (i=1, FindFirst(); i <= len; i++, FindNext()) {
    place = get_current();
    (*place).setlrate(newval);
   }
}

float Bpnet::getmu() {
 return momentum;
}

void Bpnet::set_mu(float newval) {
  int i, len =listlen();
  Bpnode *place;

  momentum  = newval;
  for (i=1, FindFirst(); i <= len; i++, FindNext()) {
    place = get_current();
    (*place).setmu(newval);
   }
}

/* readweights assumes the weights file has same format
** as pdp bp weights file.
** readweights also assumes that in the lit bpnet, the
** extra bias node for outputs is the last node in the list--
** it does not have to be searched for
*/
```

```
void Bpnet::read_weights(char *fname) {
  int len, i, j;
  float newval;
  Bpnode *place;
  RWnode *wt;
  int nnodes = listlen();
  FILE *fp;

  if ((fp = fopen(fname, "r")) == NULL) {
      printf("\nALERT: Unable to open file %s for input.\n", fname);
      return;
   }
// reads weights on inputs from file
   for (j=1,FindFirst(); j<nnodes;j++,FindNext()) {
     place = get_current();
    for (i=1,(*place).getiwts()->FindFirst();
     i<=ninputs;i++,(*place).getiwts()->FindNext()) {
      fscanf(fp, "%f", &newval);
      wt = (*place).getiwts()->get_current();
      (*wt).Setval(newval);
     }
   }
//reads weights on outputs from file
   for (i=1,FindFirst();i<nnodes;i++,FindNext()) (*get_current()).getowts()->FindFirst();
   for (i=1; i<=noutputs;i++) {
     for (j=1,FindFirst(); j<nnodes;j++,FindNext()) {
        place = get_current();
        fscanf(fp, "%f", &newval);
        wt = (*place).getowts()->get_current();
        (*wt).Setval(newval);
        (*place).getowts()->FindNext();
       }
    }

//biases on input nodes in pdp net are all 0 typically
  for (i=1; i<=ninputs;i++) {
    fscanf(fp, "%f", &newval);
 }

//reads biases on original hidden nodes--
// corresponds to input biases in lit bp
    for (i=1,FindFirst(); i<nnodes;i++,FindNext()) {
      fscanf(fp, "%f", &newval);
      place = get_current();
      (*place).setbias(newval);
     }
//reads biases on original output nodes--
// corresponds to extra bias node in lit bp
    FindLast();
    place = get_current();
    len = (*place).getowts()->listlen();
    for (i=1,(*place).getowts()->FindFirst(); i<=len;i++,(*place).getowts()->FindNext()) {
      fscanf(fp, "%f", &newval);
```

```
    wt  = (*place).getowts()->get_current();
    (*wt).Setval(newval);
   }
  fclose(fp);
}

/* saveweights writes out a weight file in the same format
** as pdp bp weights file, so that weight files are
** interchangeable between the two programs.
** like readweights, save weights also assumes that in the lit bpnet, the
** extra bias node for outputs is the last node in the list--
** it does not have to be searched for
*/

void Bpnet::save_weights(char *fname) {
  int len, i, j;
  float wtval;
  Bpnode *place;
  RWnode *wt;
  int nnodes = listlen();
  FILE *fp;

  if ((fp = fopen(fname, "w")) == NULL) {
    printf("\nALERT: Unable to open file %s for input.\n", fname);
    return;
  }
 for (i=1, FindFirst();i<=nnodes;i++,FindNext()) {
   place = get_current();
   (*place).getowts()->FindFirst();
  }
// saves weights on inputs to file
 for (j=1,FindFirst(); j<nnodes;j++,FindNext()) {
    place = get_current();
    for (i=1,(*place).getiwts()->FindFirst();
     i<=ninputs;i++,(*place).getiwts()->FindNext()) {
     wt = (*place).getiwts()->get_current();
     wtval= (*wt).getval();
     fprintf(fp, "%f\n", wtval);
    }
  }

//saves weights on outputs to file
 for (i=1; i<=noutputs;i++) {
   for (j=1,FindFirst(); j<nnodes;j++,FindNext()) {
     place = get_current();
     wt = (*place).getowts()->get_current();
     wtval = (*wt).getval();
     fprintf(fp, "%f\n", wtval);
     (*place).getowts()->FindNext();
    }
  }
//biases on pdp bp input nodes in pdp net are all 0 typically
// this is included here to preserve the format of the
// pdp weights file
```

```
  for (i=1; i<=ninputs;i++) {
      fprintf(fp, "0.000000\n", wtval);
 }

// save input biases in lit bp nodes
//correspond to biases on pdp bp hidden nodes--

    for (i=1,FindFirst(); i<nnodes;i++,FindNext()) {
     place = get_current();
     wtval = (*place).getbias();
     fprintf(fp, "%f\n", wtval);
    }
// saves biases in extra bias node in lit bp
//corresponds to  biases on pdp bp  output nodes--

    FindLast();
    place = get_current();
    len = (*place).getowts()->listlen();
    for (i=1,(*place).getowts()->FindFirst(); i<=len;i++,(*place).getowts()->FindNext()) {
     wt  = (*place).getowts()->get_current();
     wtval = (*wt).getval();
     fprintf(fp, "%f\n", wtval);
    }
   fclose(fp);
}

int Bpnet::getnepochs() {
   return nepochs;
}

void Bpnet::setnepochs(int newepochs) {
   nepochs = newepochs;
}

int Bpnet::getnnodes() {
  return listlen();
}

void Bpnet::train(BptrainSet *tset) {

   int i,j,k, len, outlen;
   float tsse, psse;
   float errval, activ, newval;
   Reallist *input;
   Reallist *target;
   Reallist *result;
   Reallist *errors;
   Reallist *errdelta;
   Realnode *newnode;



   int nnodes = getnnodes();
```

```
   len = (*tset).listlen();

   for (i=1; i <=nepochs; i++) {
printf("epoch = %d   ", i);
      tsse = 0;
      for (j=1, (*tset).FindFirst(); j<=len;j++, (*tset).FindNext()) {
        input = (*tset).getipatt();
        target =(*tset).gettpatt();
        result = Compute_output(input);
        errors = (*target) - result;
        outlen = (*target).listlen();
        errdelta = new Reallist();
         for (k =1, (*errors).FindFirst(),(*result).FindFirst();
          k <=outlen;k++,(*errors).FindNext(),(*result).FindNext()) {
           errval= ((*errors).get_current())->getval();
           activ = ((*result).get_current())->getval();
           newval = errval*activ*(1-activ);
           newnode = new Realnode(newval);
          (*errdelta).InsertAfter(newnode);
         }
         psse = (*errors).ss();
        tsse += psse;
         for (k=1,FindFirst();k<=nnodes;k++,FindNext()) {
           (get_current())->Changeweights(input,errdelta);
         }
           }
printf("tsse = %f  by %f\n", tsse, errthresh);
                 if (tsse < errthresh) { printf("break!\n"); break; }
    }
}

void Bpnet::test(BptestSet *tset) {
  int i, len;
  Reallist *inputs;
  Reallist *results;

  len =(*tset).listlen();
  for (i=1,(*tset).FindFirst(); i<=len;i++,(*tset).FindNext()) {
    inputs = (*tset).getpatt();
    results = Compute_output(inputs);
    (*inputs).Print();
    (*results).Print();
  }
}

float Bpnet::geterrthresh(){
  return errthresh;
}

void Bpnet::seterrthresh(float newval){
  errthresh = newval;
}
```

```
/* i now have an idea how to write out the weights, biases, etc,
to show that the networks initially, and after learning generate
the same weight values:  we compare the weight files at
the same epoch for the respective networks.

as far as in testing mode, the inputs and actual output patterns
for the lit network can be dumped to a file, and those
values compared with the corresponding ones for the original.
this will be sufficient to show that the networks perform
equivalently.
*/

/////////////////// individual test pattern stuff /////////////////
BptestPatt::BptestPatt(): Listnode() {
  name = 0;
  ip = 0;
}

BptestPatt::BptestPatt(char *s, Reallist *pattern): Listnode() {
  if (s!= 0) name = String(s);
  ip = pattern;
}

BptestPatt::~BptestPatt() {
  delete []name;
}

char *BptestPatt::getname() {
  return name;
}

void BptestPatt::setname(char *s) {
  delete []name;
  if (s==0) name = 0;
  else name = String(s);
}

Reallist *BptestPatt::getpattern() {
  return ip;
}

void BptestPatt::setpattern(Reallist *newpattern) {
  ip = newpattern;
}

void BptestPatt::Print() {
  int i, len;
  float ival;
  fprintf(iop, "%s ", name);
  len = (*ip).listlen();
  for (i=1,(*ip).FindFirst();i<=len;i++,(*ip).FindNext()) {
    ival = (*ip).get_current()->getval();
    fprintf(iop,"%d", (int)ival);
```

```
  }
  fprintf(iop,"\n");
}

///////////////// individual training pattern stuff //////////////

BptrainPatt::BptrainPatt(): Listnode() {
  name = 0;
  ip = 0;
  tp = 0;
}

BptrainPatt::BptrainPatt(char *s, Reallist *ipatt, Reallist *tpatt): Listnode() {
  if (s!= 0) name = String(s);
  ip = ipatt;
  tp = tpatt;
}

BptrainPatt::~BptrainPatt() {
  delete []name;
}

char *BptrainPatt::getname() {
  return name;
}

void BptrainPatt::setname(char *s) {
  delete []name;
  if (s==0) name = 0;
  else name = String(s);
}

Reallist *BptrainPatt::getipattern() {
  return ip;
}

Reallist *BptrainPatt::gettpattern() {
  return tp;
}

void BptrainPatt::setipattern(Reallist *newpattern) {
  ip = newpattern;
}

void BptrainPatt::settpattern(Reallist *newpattern) {
  tp = newpattern;
}

void BptrainPatt::Print() {
  int i, len;
  float ival;
  fprintf(iop, "%s ", name);
  len = (*ip).listlen();
  for (i=1,(*ip).FindFirst();i<=len;i++,(*ip).FindNext()) {
```

```
    ival = (*ip).get_current()->getval();
    fprintf(iop,"%d", (int)ival);
  }
  fprintf(iop,"   ");
  len = (*tp).listlen();
  for (i=1,(*tp).FindFirst();i<=len;i++,(*tp).FindNext()) {
    ival = (*tp).get_current()->getval();
    fprintf(iop,"%d", (int)ival);
  }
  fprintf(iop,"\n");
}

//////////  test set stuff  ////////////////////////

BptestSet::BptestSet(): List() {
}

BptestSet::BptestSet(char *fname): List() {

   FILE *fp;
   char label[80], instring[80];
   int ninputs, npatterns;
   int i, j;
   float ival;
   Realnode *newnode;
   Reallist *inputs;
   BptestPatt *newpattern;

   if ((fp = fopen(fname, "r")) == NULL) {
     printf("\nALERT: Unable to open file %s for input.\n", fname);
     return;
    }
   fscanf(fp, "%s %d", label, &ninputs);
   fscanf(fp, "%s %d", label, &npatterns);
   for (i=1; i<=npatterns; i++) {
      inputs = new Reallist();
      fscanf(fp, "%s", &label);
      for (j=1;j<=ninputs; j++) {
         fscanf(fp,"%f", &ival);
         newnode = new Realnode(ival);
         (*inputs).InsertAfter(newnode);
       }
printf("in testpattmaker\n");
(*inputs).Print();
     newpattern = new BptestPatt(label,inputs);
     InsertAfter(newpattern);
    }
  fclose(fp);
}

BptestPatt *BptestSet::get_current() {
  return ((BptestPatt *)List::get_current());
}
```

```
BptestPatt *BptestSet::Remove() {
  return ((BptestPatt *)List::Remove());
}

Reallist *BptestSet::getpatt() {
  BptestPatt *temp;

  temp = get_current();
  return (*temp).getpattern();
}

//////////////  training set stuff ///////////////////////

BptrainSet::BptrainSet(): List() {
}

BptrainSet::BptrainSet(char *fname): List() {

   FILE *fp;
   char label[80];
   int ninputs, noutputs, npatterns;
   int i, j;
   float ival;
   Realnode *newnode;
   Reallist *inputs, *outputs;
   BptrainPatt *newpattern;

   if ((fp = fopen(fname, "r")) == NULL) {
     printf("\nALERT: Unable to open file %s for input.\n", fname);
     return;
   }
   fscanf(fp, "%s %d", label, &ninputs);
   fscanf(fp, "%s %d", label, &noutputs);
   fscanf(fp, "%s %d", label, &npatterns);
   for (i=1; i<=npatterns; i++) {
      inputs = new Reallist();
      outputs = new Reallist();
      fscanf(fp, "%s", &label);
      for (j=1;j<=ninputs; j++) {
         fscanf(fp, "%f", &ival);
         newnode = new Realnode(ival);
         (*inputs).InsertAfter(newnode);
      }
      for (j=1;j<=noutputs; j++) {
         fscanf(fp, "%f", &ival);
         newnode = new Realnode(ival);
         (*outputs).InsertAfter(newnode);
      }
     newpattern = new BptrainPatt(label,inputs,outputs);
     InsertAfter(newpattern);
   }
  fclose(fp);
}
```

```
BptrainPatt *BptrainSet::get_current() {
  return ((BptrainPatt *)List::get_current());
}

BptrainPatt *BptrainSet::Remove() {
  return ((BptrainPatt *)List::Remove());
}

Reallist *BptrainSet::getipatt() {
  BptrainPatt *temp;

  temp = get_current();
  return (*temp).getipattern();
}

Reallist *BptrainSet::gettpatt() {
  BptrainPatt *temp;

  temp = get_current();
  return (*temp).gettpattern();
}
```

Bibliography

[Alm89] G. Almasi, A. Gottlieb. *Highly Parallel Computing*. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc. 1989.

[Ame92] American NeuraLogix, Inc. *NLX420 Neural Processor Slice*. February, 1992.

[Baf92] P. Baffes, J. Zelle. Growing Layers of Perceptrons: Introducing the Extentron Algorithm. *Proceedings of 1992 IEEE/INNS International Joint Conference on Neural Networks—Baltimore*, Vol. 2, pp. 4979-4984, 1992.

[Bar94] A. Bartczak. Hardware Implementation of Adaptive Self-Organizing Concurrent Systems. Ph.D. Dissertation, University of Rhode Island, 1994.

[Cam93] M. Campbell, et. al. 3-D Wafer Stack Neurocomputing. *Proceedings of the IEEE International Conference on Wafer Scale Integration*, pp. 67-74, 1993.

[Cha87] J. Chang, J. Vidal. Inferencing in Hardware. *Proceedings of the MCC-University Research Symposium, Austin Texas*, 1987.

[Clar89] P. Clark, T. Niblett. CN2 Induction Algorithm. *Machine Learning 3*, pp. 261-283, 1989.

[DAR88] DARPA. *Neural Network Study*. AFCEA International Press, 1988.

[Den93] T. Denoeux, R. Lengelle. Initializing Back Propagation Networks With Prototypes. *Neural Networks*, Vol. 6, #3, pp. 351-363, Pergamon Press Ltd., New York, 1993.

[Dia94] J. Diamond. Intel, Corp. Ni1000. (personal communication). 1994.

[Dis90] F. Distante, et. al. A General Configurable Architecture for WSI Implementation for Neural Nets. *Proceedings of the IEEE International Conference on Wafer Scale Integration*, pp. 116-123, 1990.

[Doa93] D. Doane, P. Franzon, eds. *Multi-chip Module Technologies and Alternatives: The Basics*. Van Nostrand Reinhold, 1993.

[Fah88] S. Fahlmann. Faster-Learning Variations on Back-Propagation: An Empirical Study. *Proceedings of the 1988 Connectionist Models Summer School*, pp. 38-51, 1988.

[Fah90] S. Fahlmann, C. Lebiere. The Cascade-Correlation Learning Architecture. in *Advances in Neural Information Processing 2*, pp. 524-532. Morgan Kaufmann Publishers: Los Altos, CA, 1990.

[Far85] N. Farhat, D. Psaltis, A. Prata, E. Paek. Optical Implementation of the Hopfield Model. *Applied Optics,* Vol. 24, #10, pp. 1469-1475, 1985.

[For93] P. Ford, Y. Rekhter, and H. Braun. Improving the Routing and Addressing of IP, *IEEE Network*, Vol. 7, #3, pp. 10-15, May 1993.

[Gam93] G. Gamrat, P. Peretto, A. Mougin, O. Ulrich. MIND-1024: A 1024 Neurons Fully Connected Real-Time Neurocomputer. *Proceedings, World Congress on Neural Networks 1993*, Vol. 4, pp. 783-786. INNS Press, 1993.

[Gra90] H. Graf, L. Jackel, W. Hubbard. VLSI Implementation of a Neural Network Model. In *Artificial Neural Networks: Electronic Implementations*, Nelson Morgan, Ed., pp. 34-42, 1990.

[Ham90] D. Hammerstrom, T. Leen, E. Means. Dynamics and VLSI Implementation of Self-Organizing Networks. In *Advanced Neural Computers*. pp. 185-192. North-Holland, 1992.

[Ham91] D. Hammerstrom, W. Henry, M. Kuhn. Neurocomputer System for Neural-Network Applications. In *Parallel Digital Implementations of Neural Networks*. K. Przytula, V. Prasanna, Eds. Prentice-Hall, Inc., 1991.

[Hay94] S. Haykin. *Neural Networks: A Comprehensive Foundation.* New York: Macmillan College Publishing Company, Inc., pp. 266-268, 1994.

[Hec87] R. Hecht-Nielsen. Counterpropagation Networks. *Applied Optics,* Vol. 26, #23, pp. 4979-4984, 1987.

[Her90] D. Herrell, H. Hashemi. *Frontiers in Computing Systems Research*. Plenum Press, 1990.

[Hil85] W. Hillis. *The Connection Machine.* Cambridge, Mass.: MIT Press, 1985.

[Int91] Intel Corporation. *80170NX Electrically Trainable Analog Neural Network*. June, 1991.

[Kar92] E. Karnin. A Simple Procedure for Pruning Back-Propagation Trained Neural Networks. *IEEE Transactions On Neural Networks,* Vol. 1, #2, pp. 239-242, January, 1992.

[Kat93] D. Katz, P. Ford. TUBA: Replacing IP with CLNP, *IEEE Network*, Vol. 7, #3, pp. 38-47, May 1993.

[Kee92] D. Keezer. Bare Die Testing and MCM Probing Techniques. *Proceeding of the IEEE Multi-Chip Module Conference*, pp. 20-23, 1992.

[Kem92] D. Kemsley, T. Martinez, D. Campbell. A Survey of Neural Network Research and Fielded Applications. *International Journal of Neural Networks,* Vol. 2, #2/3/4, pp. 123-133, 1992.

[Lee93] C. Lee, "A Simple Procedure for Pruning Back-Propagation Trained Neural Networks", *Neural Networks*, Vol. 6, #3, pp. 385-392, 1993.

[Lin91] W. Lin, V. Prasanna, K. Wojtek. Algorithmic Mapping of Neural Network Models onto Parallel SIMD Machines. *IEEE Transactions On Computers,* Vol. 40, #12, pp. 1390-1401. December, 1991.

[Lip87] R. Lippman. An Introduction to Computing with Neural Networks, *IEEE ASSP Magazine,* Vol. 3, pp. 4-22, 1987.

[Mar86] T. Martinez. Adaptive Self-Organizing Logic Networks. Ph.D. Dissertation. *Tech. Report CSD 860093*, University of California at Los Angeles. May, 1986.

[Mar88] T. Martinez, J.J. Vidal. Adaptive Parallel Logic Networks. *Journal of Parallel and Distributed Computing,* Vol. 5, #1, pp. 26-58, 1988.

[Mar89] T. Martinez. Neural Network Applicability: Classifying the Problem Space. *Proceedings, IASTED International Symposium on Expert Systems and Neural Networks*, pp. 41-44, 1989.

[Mar90] T. Martinez. Consistency and Generalization of Incrementally Trained Connectionist Models. In *International Symposium on Circuits and Systems*, pp. 706-709, 1990.

[Mar91a] T. Martinez, D. Campbell. A Self-Organizing Binary Decision Tree for Incrementally Defined Rule Based Systems. *IEEE Trans. on Systems, Man and Cybernetics,* Vol. 21, #5, pp. 1231-1237, 1991.

[Mar91b] T. Martinez, D. Campbell. A Self-Adjusting Dynamic Logic Module. *Journal of Parallel and Distributed Computing,* Vol. 11, #4, pp. 303-313, 1991.

[Mar94a] T. Martinez, D. Campbell, B. Hughes. Priority ASOCS. *Journal of Artificial Neural Networks*, Vol. 1, #3, pp. 403-429, 1994.

[Mar94b] T. Martinez, G. Rudolph. A Learning Model for Adaptive Routing. *Proceedings of the International Workshop on Artificial Neural Networks and Telecommunications*. 1994.

[McD91] K. Mcdonald, T. Martinez, and D. M. Campbell, A Connectionist Method for Adaptive Real-Time Network Routing, *Proceedings of the 4th International Symposium on Artificial Intelligence*, pp. 371-377, 1991.

[Mea91] C. Mead. *Analog VLSI and Neural Systems.* Addison-Wesley Publishing Company, Inc., 1991.

[Mic86] R. Michalski, et. al. The Multi-purpose Incremental Learning System AQ15 and its Application to 3 Medical Domains. *Proceedings of the 5th National Conference on Artificial Intelligence*, pp. 1041-1045. San Mateo, CA: Morgan Kaufman Publishers, 1986.

[Odr93] S. Odri, D. Petrovacki, G. Krstonosic. Evolutional Development of a Multilevel Neural Network. *Neural Networks*, Vol. 6, #4, pp. 583-595. Pergamon Press Ltd.: New York. 1993.

[Par92] R. Parker. Bare Die Test. *Proceeding of the IEEE Multi-Chip Module Conference*. pp. 24-27, 1992.

[Qui86] J. Quinlan. Induction of Decision Trees. *Machine Learning 1*, pp. 81-106, 1986.

[Ram90] U. Ramacher. Hardware Concepts for Neural Networks. In *Advanced Neural Computers*, pp. 209-218. North Holland: 1990.

[Ram91] U. Ramacher, B. Schürmann. Unified Description of Neural Algorithms for Time-Independent Pattern Recognition. In *VLSI Design of Neural Networks*, U. Ramacher and U. Rückert, Eds., pp. 255-270. Kluwer Academic Publishers: 1991.

[Ram93] U. Ramacher, W. Raab, J. Anlauf, U. Hachmann, J. Beichter, N. Brüls, M. Weißling, E. Schneider, R. Männer, J. Gläß. Multiprocessor and Memory Architecture of the Neurocomputer SYNAPSE-1. *Proceedings, World Congress on Neural Networks 1993*, Vol. 4. pp. 775-778. INNS Press, 1993.

[Rei88] D. Reilly, L. Cooper, C. Elbaum. Learning Systems Based on Multiple Neural Networks. (Internal paper). Nestor, Inc., 1988.

[Rud89] G. Rudolph, T. Martinez. DNA: Toward an Implementation of ASOCS. Proceedings of the IASTED *International Symposium on Expert Systems and Neural Networks*, pp. 12-15, 1989.

[Rud91a] G. Rudolph, T. Martinez. An Efficient Static Topology for Modeling ASOCS. *International Conference on Artificial Neural Networks*. Helsinki, Finland. In *Artificial Neural Networks,* Kohonen et al, pp. 279-734. North Holland: Elsevier Publishers, 1991.

[Rud91b] G. Rudolph. A Location-Independent ASOCS Model. MS Thesis, Brigham Young University, 1991.

[Rud95a] G. Rudolph, T. Martinez. An Efficient Transformation for Implementing Two-Layer Feedforward Neural Networks. *To appear in Journal of Artificial Neural Networks, 1995.*

[Rud95b] G. Rudolph, T. Martinez. A Transformation for Implementing Multilayer Feedforward Neural Networks. Submitted, 1995.

[Rud95c] G. Rudolph, T. Martinez. A Transformation for Implementing Neural Networks with Localist Properties. *Intelligent Systems*, E. A. Yfantis, ed., pp. 637-645. Kluwer Academic Publishers, 1995.

[Rud95d] G. Rudolph, T. Martinez. LIA: A Location-Independent Transformation for ASOCS Adaptive Algorithm 2 (AA2). Submitted, 1995.

[Rud95e] G. Rudolph, T. Martinez. Location-Independent Transformations: A General Strategy for Implementing Neural Networks. *To appear in the International Journal of Artificial Intelligence Tools*, 1995.

[Rud95f] G. Rudolph, T. Martinez. A Transformation for Implementing Efficient Dynamic Backpropagation Networks. *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*, pp. 41-44, 1995.

[Rud95g] G. Rudolph, T. Martinez, A Transformation for Implementing Localist Neural Networks. *To appear in the Journal of Neural, Parallel and Scientific Computations*, 1995.

[Rum86] D. Rumelhart, J. McClelland, et. al., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1, 2*. MIT Press, 1986.

[Rum88] D. Rumelhart, J. McClelland, et. al.*Explorations in Parallel Distributed Processing*, MIT Press, 1988.

[Sha92] S. Shams, Dream Machine—A Platform for Efficient Implementation of Neural Networks with Arbitrarily Complex Interconnect Structures. Ph.D. Dissertation, *Technical Report CENG 92-23*, USC, 1992.

[Spe93] A. Sperduti, A. Starita. “Speed Up Learning and Network Optimization With Extended Back Propagation”, *Neural Networks*, Vol. 6, #3, pp. 365-383, Pergamon Press Ltd., New York, 1993.

[Sto94a] M. Stout, G. Rudolph, T. Martinez, L. Salmon. A Multi-Chip Module Implementation of a Neural Network. Proceedings of the *IEEE Multi-Chip Conference*, pp. 20-25, 1994.

[Sto94b] M. Stout, G. Rudolph, T. Martinez, L. Salmon. A VLSI Implementation of a Parallel, Self-Organizing Learning Model.Proceedings of the *International Conference on Pattern Recognition*, pp. 373-376, 1994.

[Sto94c] M. Stout. An Approach to Connectionist System Hardware Implementation Using a Multi-Chip Module Interconnection and Packaging Scheme. MS Thesis, Brigham Young University, 1994.

[Ver89] M. Verleysen, P. Jespers. An Analog VLSI implementation of Hopfield's Neural Network. *IEEE Micro.*, pp. 47-55. December, 1989.

[Wri91] M. Wright. Neural-network IC Architectures Define Suitable Applications. *EDN*, pp. 84-90. July, 1991.

[Yas91] M. Yasunaga. A Self-Learning Neural Network Composed of 1152 Digital Neurons in Wafer-Scale LSIs. *IEEE Joint Conference on Neural Networks*, Vol. 3, pp. 1844-1849, 1991.

Location-Independent Neural Network Models
George Rudolph
Computer Science Department
Ph.D. Degree, August 1995

ABSTRACT

Neural networks that use a static topology, i.e. a topology that remains fixed throughout learning, suffer from a number of short-comings. Current research is demonstrating the use of dynamic topologies in overcoming some of these problems. The Location-Independent Transformation (LIT) is a general strategy for implementing in hardware neural networks with static and dynamic topologies. LIT maps an arbitrary neural network onto a uniform tree topology which uses global broadcast and gather operations for communication.

This dissertation formally defined LITs with associated execution and learning algorithms for static and dynamic versions of competitive learning, counterpropagation, 2-layer backpropagation, and multilayer feedforward networks. An LIT for ASOCS AA2 model (which is only dynamic) is also described. This is a representative set of neural network models, not an exhaustive list, providing potential guidelines for implementing other models of interest.

COMMITTE APPROVAL:

Tony Martinez, Committee Chairman

Dan Olsen, Committee Member

Scott Woodfield, Committee Member

Gordon Stokes, Committee Member

Parris Egbert, Committee Member

Date

Tony Martinez, Graduate Coordinator