In *Neural Parallel and Scientific Computations*, vol. **3**, no. 2, pp. 173-188, 1995.

A TRANSFORMATION FOR IMPLEMENTING LOCALIST NEURAL NETWORKS

George L. Rudolph and Tony R. Martinez
Brigham Young University, Computer Science Department
Provo, Utah 84602
e-mail: george@axon.cs.byu.edu

Abstract
Most Artificial Neural Networks (ANNs) have a fixed topology during learning, and typically suffer from a number of short-comings as a result. Variations of ANNs that use dynamic topologies have shown ability to overcome many of these problems. This paper introduces Location-Independent Transformations (LITs) as a general strategy for parallel implementation of feedforward networks that use dynamic topologies. A LIT creates a set of location-independent nodes, where each node computes its part of the network output independent of other nodes, using local information. This type of transformation allows efficient support for adding and deleting nodes dynamically during learning. This paper deals specifically with LITs for *localist* ANNs—localist in the sense that ultimately one node is responsible for each output. In particular, this paper presents LITs for two ANNs: a) the single-layer competitive learning network, and b) the counterpropagation network, which combines elements of supervised learning with competitive learning. The complexity of both learning and execution algorithms for both ANNs is linear in the number of inputs and logarithmic in the number of nodes in the original network.

Keywords - Neural Networks, Competitive Learning, Counterpropagation, Localist Networks, Dynamic Topologies, Reconfigurable Architectures, Implementation Design.

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 [2].

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.)

† This research is funded by grants from Novell Inc. and Word Perfect Corp.

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 [3], [10], [12], [14] for a variety of applications. [7], 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 [4-5], [11]. More recent *neurocomputer* systems have specialized neural hardware, and seek to support more general classes of ANNs [6], [13], [18]. 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 [9] and the CRAY [1], 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.

LITs have been developed for backpropagation [16], and Adaptive Self-Organizing Concurrent Systems' Adaptive Algorithm 2 [10], [15]. 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.

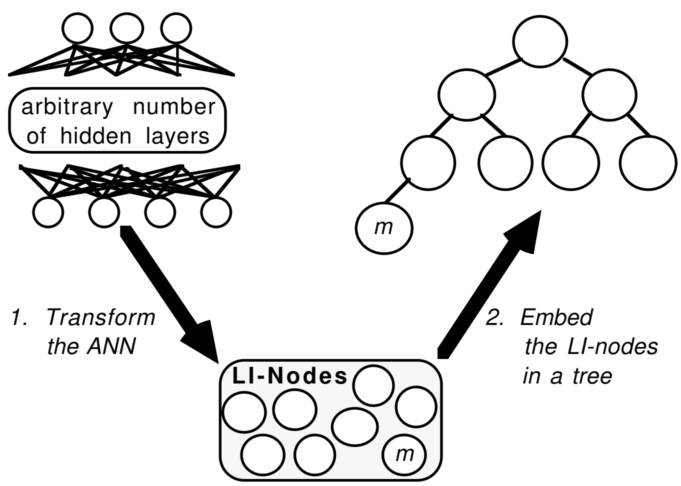

Figure 1. General LIT Transformation

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 [14], and dynamic RBF networks [7] 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.

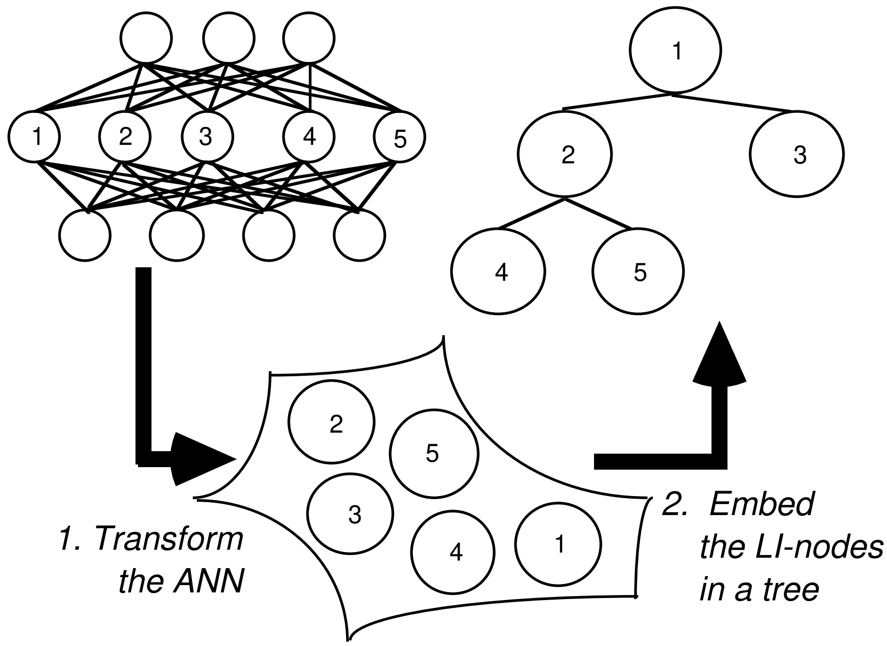

Figure 2. An Example Location-Independent Transformation

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 [19].

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 [17]. 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.

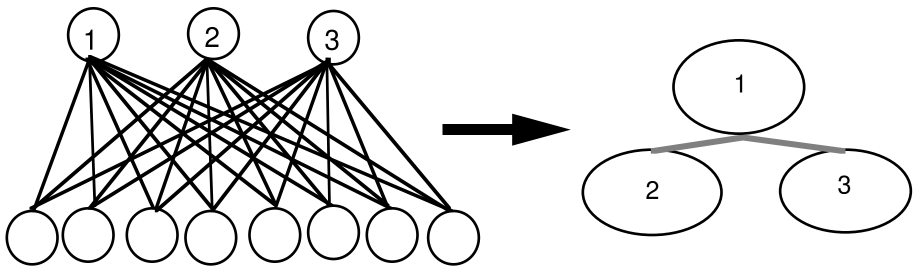

Figure 3. Transforming a CL Network

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, $class_j$).

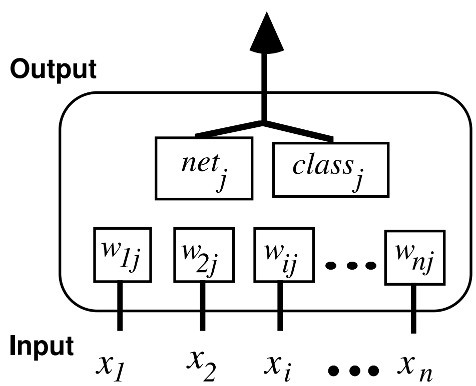

Figure 4. A CL Node

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 *j*th node, which is j itself (in this case)
w_{ij} real-valued, initially random, weight between the original *i*th input node and the original *j*th output node (resides in the *j*th transformed node)
x_i *i*th component in the Boolean input vector ***X***
m number of output nodes in the original network, also the number of nodes in the transformed network
n length or size of ***X***

<u>Equations</u>

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

$$net_j = \sum_{i=1}^{n} w_{ij} x_i \quad (2)$$

The execution algorithm proceeds as follows:

1. Broadcast the input vector ***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 ***X*****,** and hence the output of the network (to the Control Unit).

3.2 Learning Algorithm

<u>Additional Definitions</u>

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)

<u>Equations</u>

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

The learning algorithm is as follows:

Until convergence (current epoch = desired number of epochs)
For each pattern ***X*** in the training set

1. Broadcast the input vector ***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 ***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 [17]) 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 [8]. 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 $(\boldsymbol{x},\boldsymbol{y})$ vector pair as desired, where n is the size of $\boldsymbol{x}$, and k is the size of $\boldsymbol{y}$.

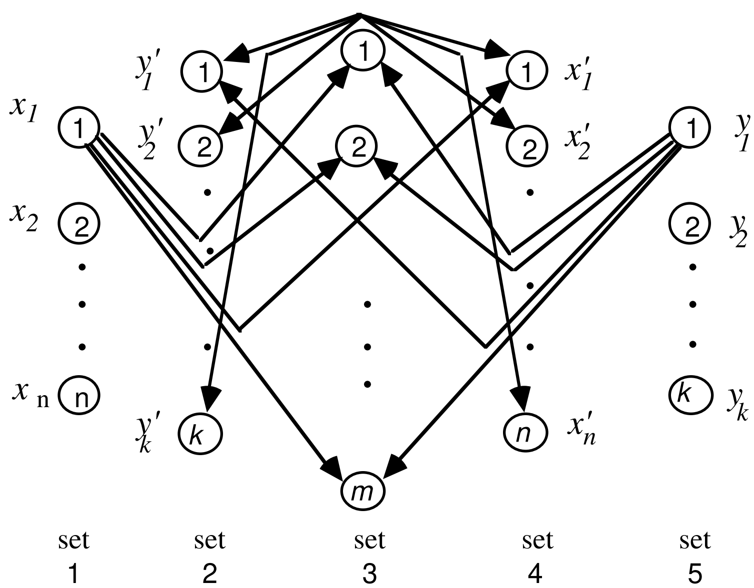

Figure 5. Original CPN Structure

2. Node sets 2 and 4 are combined into a single output set, divided into a $(\boldsymbol{x}', \boldsymbol{y}')$ vector pair corresponding to $(\boldsymbol{x},\boldsymbol{y})$.
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 ($\boldsymbol{x},\boldsymbol{y}$) is treated as a single vector $\boldsymbol{X}$, with components x_i, where $1 \le i \le k+n$.
- Similarly, the output vector pair ($\boldsymbol{x'}, \boldsymbol{y'}$) is treated as a single vector $\boldsymbol{Y}$, with components y_i, where $1 \le i \le k+n$.

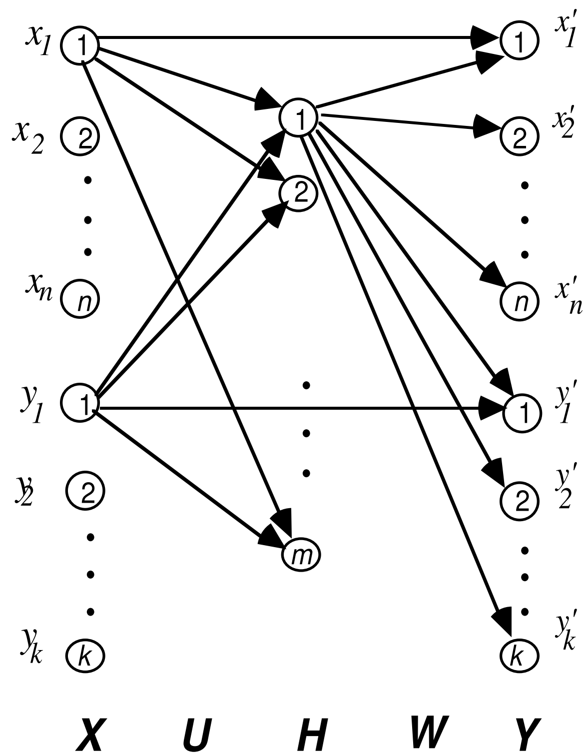

Figure 6. Rearranged CPN Network

- The input weight vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ are treated as a single weight vector $\boldsymbol{U}$, with components u_{ij}, where $1 \le i \le k+n$, $1 \le j \le m$
- Node set 3 is labeled $\boldsymbol{H}$, where h_i is the ith node in that set.
- The output weight vector $\boldsymbol{W}$ (for $\boldsymbol{x'}$ and $\boldsymbol{y'}$) is treated as a single vector $\boldsymbol{W}$, 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 $\boldsymbol{H}$ compete in winner-take-all fashion, hence this hidden set acts as a selector for the output vector. Because $\boldsymbol{H}$ is a selector, there is no recombining of the weights in layer $\boldsymbol{W}$ 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 $\boldsymbol{Y}$, because all other results are zero. This suggests that the real work of determining the output involves the weights in $\boldsymbol{U}$, the first weight layer, and the nodes in $\boldsymbol{H}$. Hence, CPN behaves as a spontaneous classifier like the original CL model of section 3: $\boldsymbol{H}$ corresponds to the output nodes, $\boldsymbol{U}$ 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.

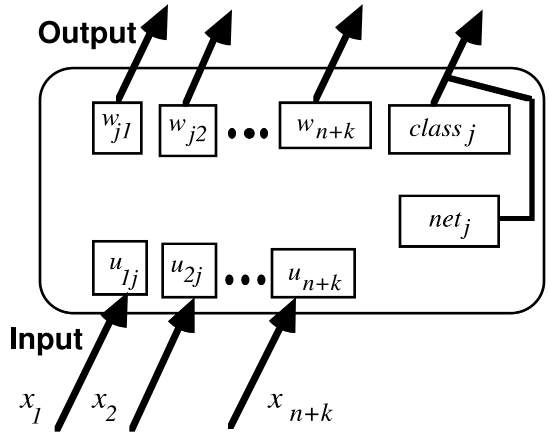

Figure 7. Structure of a Transformed CPN node

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.

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 $\boldsymbol{X}$.

Definitions

$\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{H}$, $\boldsymbol{U}$, $\boldsymbol{W}$ 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

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

$$net_j = \sum_{i=1}^{n} u_{ij} x_i \quad (5)$$

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

$$\Delta wij = (-awij + bxi),\ if\ i \leq n \tag{6}$$
$$\Delta w_{ij} = (-aw_{ij} + cx_i),\ if\ \ n < i \leq k+n$$
$$\Delta u_{ij} = \alpha(\boldsymbol{X}-u_{ij}) \tag{7}$$

The learning algorithm proceeds as follows:

Until convergence ($error \leq \varepsilon$)

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 [8].

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:

<u>Broadcast</u>

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.

<u>Gather</u>

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 $O(n+\log m)$.

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.

ACKNOWLEDGEMENTS

This research is funded in part by grants from Novell Inc. The authors wish to thank the referees for their review and suggestions.

REFERENCES

[1] Almasi, G. S., A. Gottlieb. (1989). *Highly Parallel Computing*. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc.

[2] DARPA. (1988). *Neural Network Study*. AFCEA International Press.

[3] Fahlmann, Scott, C. Lebiere. The Cascade-Correlation Learning Architechture. in *Advances in Neural Information Processing 2*. pp. 524-532. Morgan Kaufmann Publishers: Los Altos, CA.

[4] Farhat, N., D. Psaltis, A. Prata, and E. Paek. (1985). Optical Implementation of the Hopfield Model. *Applied Optics,* v.24, #10. pp.1469-1475.

[5] Graf, H., L. Jackel, W. Hubbard. (1990).VLSI Implementation of a Neural Network Model. In *Artificial Neural Networks: Electronic Implementations*. Nelson Morgan, Ed. pp. 34-42.

[6] Hammerstrom, D., W. Henry, M. Kuhn. (1991). Neurocomputer System for Neural-Network Applications. In *Parallel Digital Implementations of Neural Networks*. K. Przytula, V. Prasanna, Eds. Prentice-Hall, Inc.

[7] Haykin, Simon. (1994). *Neural Networks: A Comprehensive Foundation*. New York: Macmillan College Publishing Company, Inc. pp. 266-268.

[8] Hecht-Nielsen, R. Counterpropagation Networks. (1987). *Applied Optics,* v.26, #23. pp. 4979-4984.

[9] Hillis, W. Daniel. (1985).*The Connection Machine.* Cambridge, Mass.: MIT Press.

[10] Martinez, T.R., D.M. Campbell. (1991). A Self-Adjusting Dynamic Logic Module. *Journal of Parallel and Distributed Computing,* v.11, #4. pp. 303-313.

[11] Mead, Carver. (1989). *Analog VLSI and Neural Systems.* Addison-Wesley Publishing Company, Inc.

[12] Odri, S.V., D.P. Petrovacki, G.A. Krstonosic. (1993). Evolutional Development of a Multilevel Neural Network. *Neural Networks,* v.6, #4. pp. 583-595. Pergamon Press Ltd.: New York.

[13] Ramacher, U., W. Raab, J. Anlauf, U. Hachmann, J. Beichter, N. Brüls, M. Weißling, E. Schneider, R. Männer, J. Gläß. (1993). Multiprocessor and Memory Architecture of the Neurocomputer SYNAPSE-1. *Proceedings, World Congress on Neural Networks 1993,* v. 4. pp. 775-778. INNS Press.

[14] Reilly, D.L., L.N. Cooper, C. Elbaum. (1988). Learning Systems Based on Multiple Neural Networks. (Internal paper). Nestor, Inc.

[15] Rudolph G., and T.R. Martinez. (1991). 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.

[16] Rudolph G., Martinez, T. R. (1995). A Transformation for Implementing Efficient Dynamic Backprpagation Networks. To appear in the Proceedings of the *International Cconference on Artificial Neural Networks and Genetic Algorithms*.

[17] Rumelhart, D., J. McClelland, et. al. (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* v.1. MIT Press.

[18] Shams, S. Dream Machine—A Platform for Efficient Implementation of Neural Networks with Arbitrarily Complex Interconnect Structures. (1992).*Technical Report CENG 92-23*. PhD Dissertation, USC.

[19] Stout, M., G. Rudolph, T.R. Martinez, L. Salmon. (1994). A VLSI Implementation of a Parallel, Self-Organizing Learning Model. To appear in *Proceedings of the 12th IEEE International Conference on Pattern Recognition*.