To appear in *Journal of Artificial Neural Networks*, 1995.

An Efficient Transformation for Implementing Two-layer Feedforward Neural Networks

George L. Rudolph
Tony R. Martinez
Computer Science Department
Brigham Young University
Provo, Utah 84602

Abstract— **Most Artificial Neural Networks (ANNs) have a fixed topology during learning, and often 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 implementing distributed feedforward networks that use dynamic topologies (dynamic ANNs) efficiently in parallel hardware. 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. In particular, this paper presents an LIT for dynamic Backpropagation networks with a single hidden layer. The complexity of both learning and execution algorithms is $O(n+p+logm)$ for a single pattern, 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.**
Keywords: Neural Networks, Backpropagation, Implementation Design, Dynamic Topologies, Reconfigurable Architectures.

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 [2-5], [10-13], [16]. 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 [6-7], [13]. More recent *neurocomputer* systems have specialized neural hardware, and seek to support more general classes of ANNs [8], [15], [21]. 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 (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

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

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 counterpropagation and competitive learning [18], and Adaptive Self-Organizing Concurrent Systems' Adaptive Algorithm 2 [12], [17]. 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 [19]. Many dynamic extensions to BP have been proposed in the literature [3-5], [10-11], [14], [22]. The dynamic BP algorithm presented in this paper borrows ideas from [14]. 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 [20]; section IV presents an overview of the dynamic BP model of [14], 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.

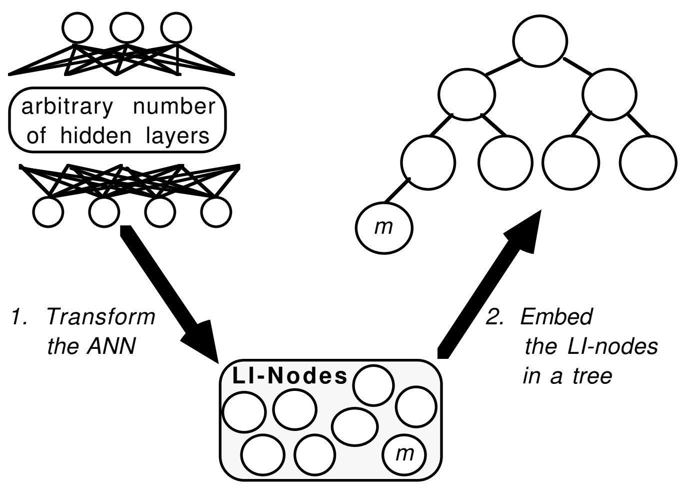

Figure 1. General LIT Transformation

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.

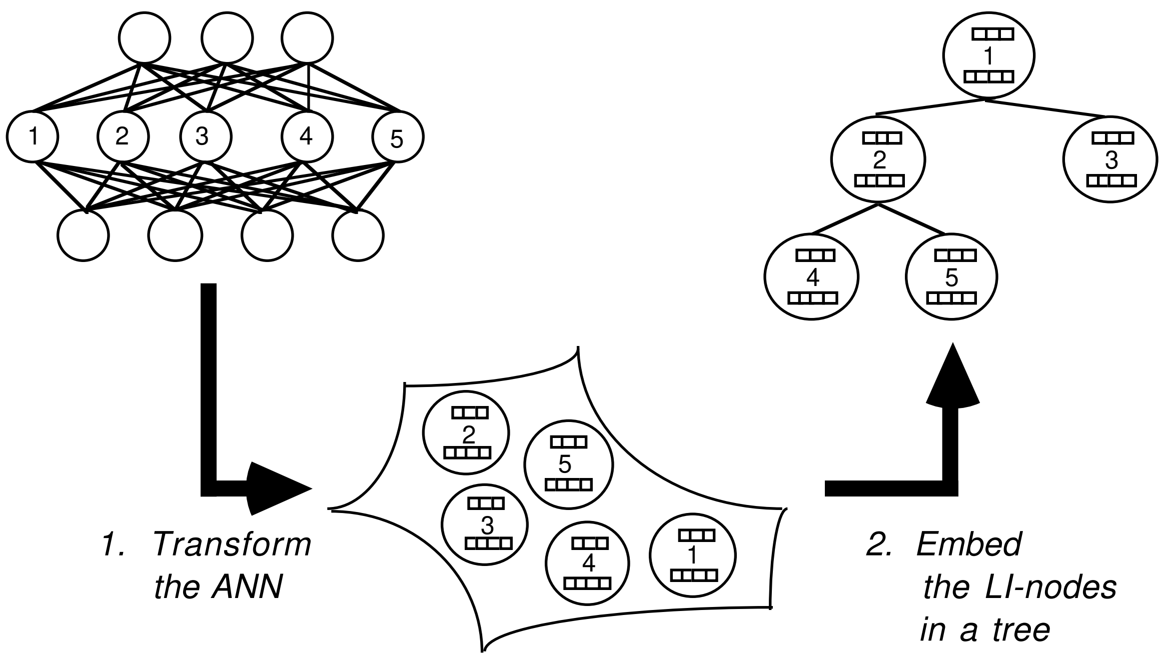

Figure 2. An Example Location-Independent Transformation

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 [20], 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 [19]. 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 $\boldsymbol{w_j}$.

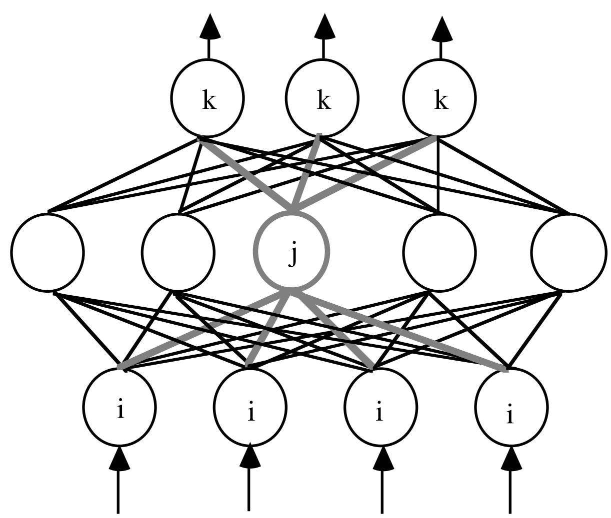

Figure 3. Use of the BP equations for an arbitrary hidden node j.

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 $\boldsymbol{u_j}$.

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.

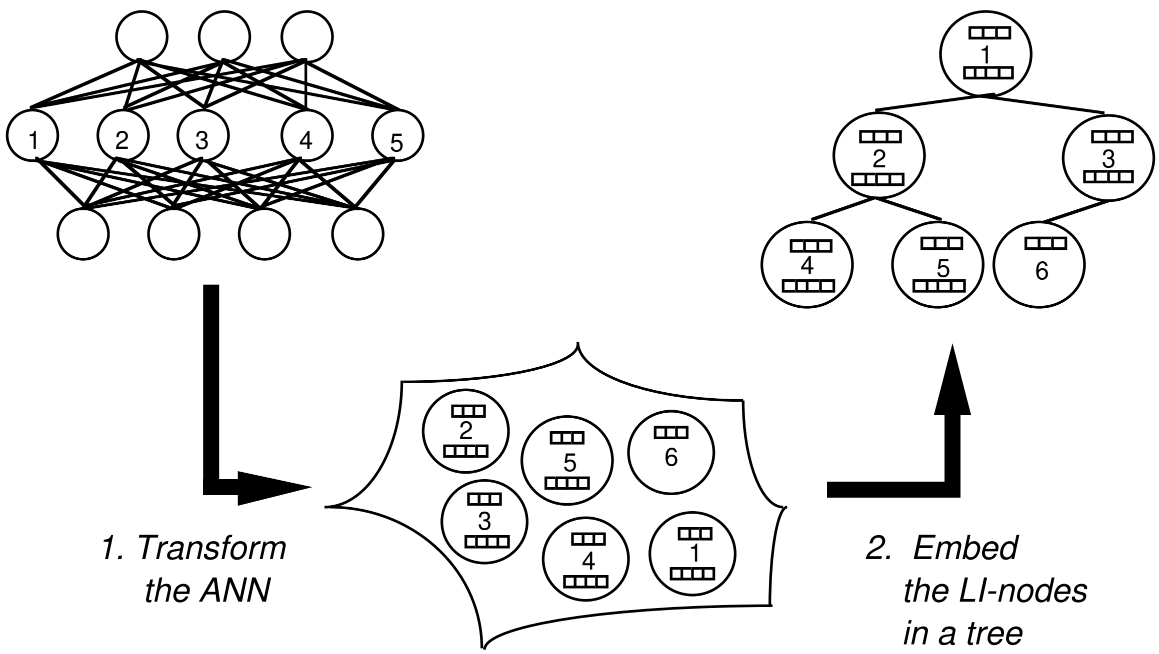

Figure 4. Transforming a 2-layer BP Network

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.

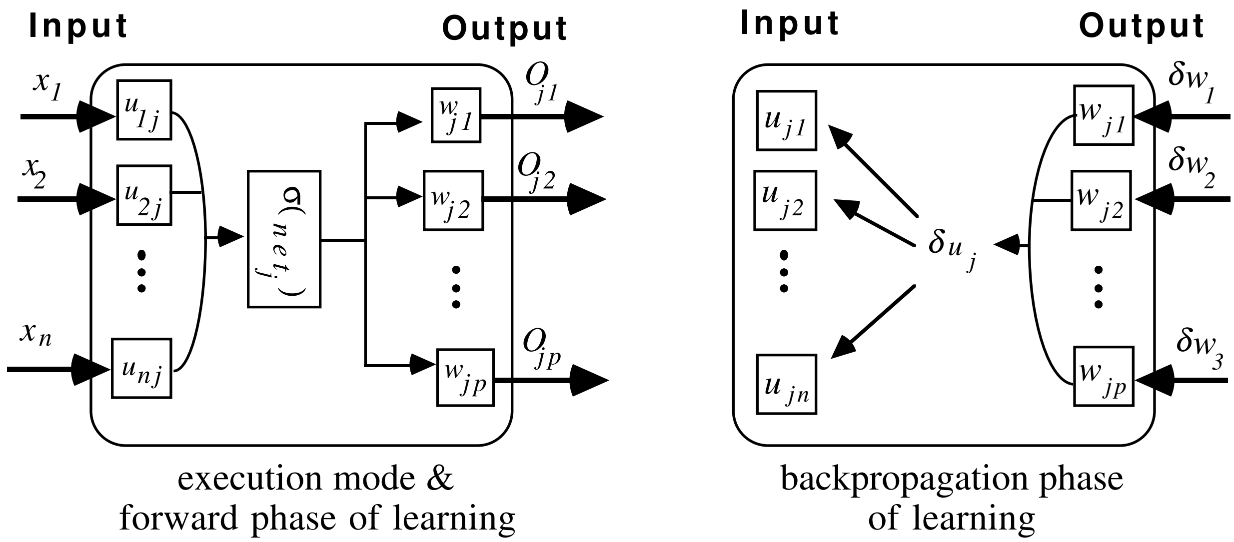

Figure 5. Two Views of a Node in the Transformed Network. The left shows a node operating in execution mode and/or the forward phase of learning. The right shows a node during the backpropagation phase of learning.

At this point ***Y*** resides at the root node. Now however, the sigmoid of each component of ***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 ***Z***. ***Z*** can be computed in parallel fashion in two ways:

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

There are two main factors involved in deciding how to compute ***Z*** from ***Y***: computation time and communication time. If computation time is fast relative to communication time, then method 1 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, ***Z*** is computed as the root node sends ***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 ***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, ***Z*** is computed from ***Y*** as above. Second, ***Z*** and ***T*** are used to compute the error vector ***δw***, which will be sent back to the nodes. (Target outputs ***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 $\delta w_k{}^2$ (for calculating sum-squared pattern error). Fourth, each $\delta w_k{}^2$ 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.

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.
$\boldsymbol{Z}$	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

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

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 $\boldsymbol{X}$ 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 (4), and then computes O_j according to equation (3). 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. The result $\boldsymbol{Z}$ 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.
- $\boldsymbol{T}$ 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.

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 [14], then formally extends the learning algoirthm of section III to support these dynamics. The DBP model of [14] 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 [14] 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.

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

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

P_j^g 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

$$P_j^g(c) = P_0^g \varphi(k_g)\beta_j(c). \tag{13}$$

P_0^g 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

$$w_{jk}^{d1} = (1+\mu)w_{jk}^{p}, \text{ and}$$

$$w_{jk}^{d2} = -\mu w_{jk}^{p}. \tag{14}$$

$d1$ 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 $\mathbf{0}$, or
2. when $\boldsymbol{w_j}$ is $\mathbf{0}$.

In case 1, the threshold or bias value may not be 0, even if all of the other weights are 0. [14] 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 $\widehat{F}_{ij}$ are then used to compute the probability that a weight will self-delete:

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

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 ν, 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 $P_{ij}^{d} \geq \nu_{ij}$ self-deletes. Any weight w_{jk}, for which $P_{jk}^{d} \geq \nu_{jk}$ 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 between 0 and 1, v_j.

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 $O(n+p+\log m)$ 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 III. The complexity of the learning algorithm is presented in the same format, including the dynamic extensions.

<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		$\boldsymbol{O(n+p+\log m)}$

Broadcast or gather for a single variable is clearly $O(\log m)$. For n variables, one might expect a complexity of $O(n\log m)$, 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 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.

<u>Learning Algorithm</u>

1.	Forward Phase	$O(n+p+\log m)$

2.	Compute $\delta \mathbf{w}$	$O(p)$
3.1	Broadcast $\delta \mathbf{w}$	$O(p+logm)$
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 the 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 $\mathbf{w_j}$. 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 $\mathbf{u_j}$ 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.

REFERENCES

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

[2] P. T. Baffes, J. M. 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.

[3] T. Denoeux, R. Lengelle, “Initializing Back Propagation Networks With Prototypes” *Neural Networks*, Vol. 6, #3, pp 351-363, Pergamon Press Ltd, New York, 1993.

[4] S. Fahlmann, “Faster-Learning Variations on Back-Propagation: An Empirical Study”, *Proceedings of the 1988 Connectionist Models Summer School*, pp 38-51, 1988.

[5] S. Fahlmann, C. Lebiere, “The Cascade-Correlation Learning Architecture”, *Advances in Neural Information Processing 2*, pp 524-532, Morgan Kaufmann Publishers: Los Altos, CA.

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

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

[8] Hammerstrom, D., 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.

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

[10] E. Karnin, “A Simple Procedure for Pruning Back-Propagation Trained Neural Networks”, *IEEE Transactions On Neural Networks*, Vol. 1, #2, pp 239-242, June, 1990.

[11] C. Lee, “A Simple Procedure for Pruning Back-Propagation Trained Neural Networks”, *Neural Networks*, Vol. 6, #3, pp 385-392.

[12] T. Martinez, D. Campbell, "A Self-Adjusting Dynamic Logic Module", *Journal of Parallel and Distributed Computing*, Vol. 11, #4, pp 303-313, 1991.

[13] C. Mead, *Analog VLSI and Neural Systems*, Addison-Wesley Publishing Company, Inc, 1991.

[14] 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.

[15] Ramacher, U., 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.

[16] D. Reilly, L. Cooper, C. Erlbaum, "A Neural Model Category Learning", *Biological Cybernetics*, Vol. 45, pp 35-41, 1982.

[17] G. Rudolph, T. Martinez, "An Efficient Static Topology for Modeling ASOCS", *Artificial Neural Networks*, Editors, Kohonen et al, pp 279-734, Elsevier Publishers, North Holland, 1991.

[18] G. Rudolph, T. Martinez, "A Transformation for Implementing Localist Neural Networks", Submitted, 1994.

[19] Rudolph G., Martinez, T. R. "A Transformation for Implementing Multiple-Hidden-Layer FeedForward Networks", Submitted, 1994.

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

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

[22] 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.