In *Future Generation Computer Systems*, vol. **12**, no. 6, pp. 547-564, 1997.

A Transformation Strategy for Implementing Distributed Multilayer Feedforward Neural Networks: Backpropagation Transformation

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

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 feed forward 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 a LIT that supports both the standard (static) multilayer backpropagation network, and backpropagation with dynamic extensions. The complexity of both learning and execution algorithms is $O(q(N+logM))$ for a single pattern, where q is the number of weight layers in the original network, N is the number of nodes in the widest node layer in the original network, and M is the number of nodes in the transformed network (which is linear in the number hidden nodes in the original network). This paper extends previous work with 2-weight-layer backpropagation networks.

Keywords. Neural Networks, Backpropagation, Implementation Design, Dynamic Topologies, Reconfigurable Architectures.

1. Introduction

Hardware support for ANNs is important for enabling real-time handling of large, complex problems, in particular for enabling real-time learning. Learning times can exceed tolerable limits for applications of large size and complexity using conventional computing schemes. As

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

hardware is becoming cheaper and easier to design, viable hardware mechanisms that directly support neural computation, especially ANNs with dynamic topologies, can be explored and implemented. (A *dynamic topology* is one in which the learning algorithm allows adding and deleting both nodes and weighted connections during learning, whereas a *static* topology remains fixed throughout learning.)

The *Location-Independent Transformation* (LIT) is a general implementation strategy for ANNs. The strategy overcomes several weaknesses of current hardware implementation methods, in that it provides support for ANNs with dynamic, as well as static, topologies. LIT maps an ANN onto a parallel, hierarchical network, providing bounded logarithmic broadcast and gather times. The purpose of LIT is not to invent new or improved learning algorithms, but rather to support existing algorithms in an efficient way for hardware implementation.

This paper presents a LIT for the Backpropagation learning model (BP) [23], with an extended learning algorithm that supports a dynamic topology [17]. A complete, formal description of a LIT transformation includes the following:

- giving the mapping from the original network onto the LIT architecture,
- providing formal execution and learning algorithms (for those interested in potential implementation and for formal verification of the mapping), and
- giving a complexity analysis of the formal algorithms.

Backpropagation is of interest as an example LIT model because it is well-known and used, and is a multilayer network with a distributed internal knowledge representation.

Most ANNs use static topologies, which often suffer from the following short-comings: sensitivity to user-supplied parameters—learning rate(s), etc.; local error minima during learning; and there is no mechanism for choosing 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], [11-12], [14], [16], [19].

Early ANN hardware implementations are model-specific, and are intended to support static topologies [6-7], [16]. More recent *neurocomputer* systems have specialized neural hardware, and seek to support more general classes of ANNs [9], [18], [24]. 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 [10] 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 could be used as a model for software implementation on conventional machines, on general-purpose parallel machines, or on neurocomputers—the authors simply want to target more direct hardware support.)

Figure 1 gives a general view of LIT. First, LIT maps 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.) Second, the set of LI-nodes is then embedded in a uniform tree topology. Execution (running the network) and learning are based on efficient broadcast and gather operations in the tree. An algorithm can place the location-independent nodes anywhere in the tree, thus efficiently and easily supporting dynamic topologies: Adding a node is as simple as allocating a free node in the tree; deleting a node is as simple as de-allocating the desired node. This paper assumes that each LI-node maps to its own physical node—however, in an actual implementation that a single physical node could contain many LI-nodes.

Figure 1. General LIT Transformation

The LIT strategy is a generalization of the authors' work on a strategy for implementing in hardware the ASOCS class of models [14], [20], in particular AA2 [14], which have inherently dynamic topologies. The initial results of that work were the Location-Independent ASOCS

model (LIA) [20] and the Priority ASOCS model [15]. The authors have applied and expanded the initial strategy to more common ANNs. This led to the development of LITs for counterpropagation, competitive learning [22], 2-layer backpropagation [21], [23] and backpropagation with an arbitrary number of layers (the subject of this paper). 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).

The transformation presented in this paper extends the transformation for 2-layer networks of [21] 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—it describes the basic LIT BP mapping and architecture, and gives an informal discussion of execution and learning modes. Section 4 discusses the transformation of BP networks with more than two weight layers, building on section 3. Section 5 gives the formal algorithms for the transformed network. Section 6 adds the steps to the learning algorithm that handle dynamic topologies based on the model of [17]. 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 [3-5], [11-12], [17], [25]. The reader should not assume that LIT supports only the dynamic model of [17]—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 is found in [17], [21].

2. The Transformed Network

A transformation maps the original network onto a LIT network such that each node (in the LIT network) 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 *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 does not affect 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). The construction of LI-nodes is essentially determined by how the computations and behaviors of the original network can be executed efficiently in the transformed network. Consequently, the details of the construction/transformation differ across models. It seems helpful, in determining the efficacy of a proposed mapping, to express the original equations in an equivalent form that reflects the resulting structure and behavior of the transformed network. Formal definitions of such "transformed" equations also aid in precisely specifying network behaviors for implementation. An example transformation of an original BP network to an equivalent LIT BP network is given in section 3.2. Transformed equations are given in section 5.

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 figure 1. 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 (section 5) show explicitly how the required communication can take place.

The example transformation of section 3.2 will provide a concrete example for BP of how LIT supports dynamic topologies. However, the general idea is the following: Assume the original network in figure 1 has (as is typical) each hidden layer fully connected to adjacent layers. If a hidden node were added to some arbitrary layer *i*, one connection to each node in layer *i-1* and one connection to each node in layer *i+1* would also be added. This is accomplished in the transformed network by allocating a free node in the tree, and initializing it with the desired weights on inputs and weights on outputs. The deletion of a hidden node in the original network is accomplished in the transformed network by marking the corresponding node as free. In the transformed network, only the inserted (or deleted) node is affected by the change—no other nodes are affected.

3. Transforming 2-layer BP Networks

This section begins with a brief review of the standard BP model. It is assumed that the reader is familiar with the BP model [23], so section 3.1 really provides a frame of reference between the original and transformed networks. Section 3.2 describes the process for mapping an original 2-layer BP network onto a transformed network. Section 3.3 gives an informal

description of execution mode for the transformed network. Section 3.4 describes learning mode in the transformed network in like manner.

The transformation for 2-layer networks forms the basis of the transformation for multilayer networks (section 4). In this section and section 4, variables are introduced, used and defined informally. Formal definitions of variables are given in section 5 with the formal algorithms.

3.1. Review of Standard BP Architecture

In a BP network, there are typically three groups of nodes—input nodes, output nodes, and hidden nodes (see figure 2). 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 original *node* layer as *j* are referred to collectively as the *current layer*. The nodes in the original node layer labeled with *k* are referred to as the *subsequent layer*. The nodes in the original 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 designations 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 (vector)$\boldsymbol{u_j}$, and the weights on its output as (vector)$\boldsymbol{w_j}$.

Figure 2. Hidden node *j* in a BP network.

With reference to the original network, equations, values, and symbols are used as typically defined: net_j is the sum-of-products of node *j*'s weights-on-inputs and node *j*'s corresponding inputs. $f(net_j)$ is the sigmoid of each net_j, or the scalar output O_j for a node *j* in the original

network. The vector of inputs to the network is denoted by $\boldsymbol{X}$, the vector of computed outputs for the network is $\boldsymbol{Z}$, the vector of target values for learning is $\boldsymbol{T}$, error values are denoted by δ.

The notation for the transformed network uses the symbols and values in the same way. For example, net_j is exactly the same value in the transformed network as in the original network. However, the transformed network also requires some additional notation, such as the use of the vector $\boldsymbol{O_j}$ (to denote the output vector of a transformed node j) in addition to the scalar O_j. (which has its original meaning). Where possible, the transformed notation is extended from the original in this manner in order to make behavioral correspondences more obvious.

3.2. Mapping 2-Layer BP Networks

There are a variety of potential location-independent sets of nodes that could be constructed for BP. The mapping 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 (i.e. transformed network) reflects that symmetry, as well as achieving the other goals of LIT. Once a basic mapping is determined, the number and size and weight values within the transformed nodes can be precisely determined.

3.2.1. Transformed Node Structure

Figure 3 shows the transformation of an original network with 13 nodes into a transformed network of six nodes. The original network has four input nodes, five hidden nodes, and three output nodes—denoted as a 4x5x3 original network. (Assume that the hidden nodes have bias weight θ_j, and the output nodes have bias weight θ_k). The original network maps into a transformed network with five "whole" 4x3 nodes and a "special" bias node. This is denoted by a 5:4x3+1:x3 transformed network, meaning that each whole node has a vector of four weights-on-input (plus an input bias weight, which is assumed) and a vector of three weights-on-output,

while the single bias node has its three bias weights-on-output. The activation value of the bias node is always 1.

Figure 3. Transforming a 2-layer BP Network

Let the weight values between the original input nodes and the hidden nodes be denoted by u_{ij}, where $1 \le i \le 4$ and $1 \le j \le 5$. Let the weights between the hidden nodes and the output nodes be denoted by w_{jk}, where $1 \le j \le 5$ and $1 \le k \le 3$. Here is how the transformed network is determined. For each hidden node j in the original network:

1. Consider the weights-on-input u_{ij} that are connected to node j from the input nodes as being stored in the node j.
2. Consider the weights w_{jk} that are connected to node j from each of the output nodes as being stored in the node j. (This may be confusing to the reader at first, because these weights are weights-on-inputs to the original output nodes, and thus involved in computations at different nodes—nevertheless, this is not a mistake.)

As a concrete example, consider node 2 in the original network of figure 3. According to the above mapping, node 2 would store weights u_{12}, u_{22}, u_{32}, u_{42} as $\boldsymbol{u_j}$, and weights w_{21}, w_{22}, w_{23}, w_{24}, as $\boldsymbol{w_j}$. The other nodes and weights are mapped similarly.

The results of this mapping to this point are the five transformed "whole" nodes mentioned above. However, at this point, the bias values in the hidden nodes (θ_j) and output node (θ_k) need to be accounted for in the transformed network. Each θ_j can be considered a weight-on-input for the respective node j, and treated accordingly. The θ_k weights, on the other hand, are mapped in a fashion analogous to the w_{jk} weights. This leads to the creation of the bias node in the transformed network, whose weights-on-output values w_{jk} are the respective θ_k values.

Having accounted for all the weights in the original network, the mapping is complete. As figure 4 shows, a transformed BP node therefore stores two vectors (layers) of weights—one on

inputs (hereafter $\boldsymbol{u_j}$) and one on outputs (hereafter $\boldsymbol{w_j}$). 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 "special" bias node has no $\boldsymbol{u_j}$, because its activation is always 1. Accordingly, the θ_k vector of values from the original output nodes essentially become the output of the transformed bias node—its $\boldsymbol{w_j}$ vector. The bias node is then treated uniformly with other nodes for execution and learning for this vector.

Figure 4. Two Views of a Node in the Transformed Network. (The top shows a node operating in execution mode and/or the forward phase of learning. The bottom shows a node during the backpropagation phase of learning.)

This mapping scheme, as will be evident from subsequent discussions, allows uniform behavior in the transformed network for executing what would correspond to forward and backward propagation of values in the original network.

3.2.2. Determining the Size of the Transformed Network

Given the mapping scheme above, the number of transformed nodes, and the size(of the vectors of weights on inputs and outputs) of each transformed node, and the specific initial weights (assuming some snapshot of the original network) can be determined precisely.

In the 2-layer network, such as the one referred to previously in figure 3, let the number of original input nodes be n, the number of original hidden nodes be m and the number of original output nodes be p. The number of transformed nodes required in the transformed network is

$$m+1. \tag{1}$$

There is one "whole" transformed node for each of the original m hidden nodes, plus a transformed bias node.

The size of each whole transformed node is

$$n+p+1, \tag{2}$$

where n and p are as defined above. $n+1$ units are needed because each original hidden node is connected to all n original input nodes, and also has a bias value (hence the + 1). p units needed because each hidden node is also connected to all p original output nodes.

The example given above, showing the weights which transformed node 2 in figure 3 would store, should be sufficient to indicate the specific mappings of original weight values to transformed nodes' weight values for the other transformed nodes.

3.2.3. How LIT BP Supports Dynamic Topologies

Suppose, according to some specified criteria, it was necessary to add a node to the hidden layer of the original network in figure 3. As suggested by the example at the end of section 2, four connections to the input nodes and three connections to the output nodes would also be added. This node addition is accomplished in the transformed network by allocating a free node, such as the right child of transformed node 3, and initializing it with the corresponding four weights on inputs (five counting the bias value) and three weights on outputs. The deletion of a hidden node in the original network is accomplished in the transformed network by marking the corresponding node as free. In the transformed network, only the inserted (or deleted) node is affected by the change—no other nodes are affected.

3.3. Execution Mode in the Transformed Network

The network executes as follows: The Control Unit sends the inputs to the root transformed node, which passes those values onto each of its children, and they to theirs, and so on. Each transformed node receives the broadcast (the bias node always has input clamped to 1), 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* $\boldsymbol{O_j}$, which is the output of transformed node 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 from $\boldsymbol{Y}$ in parallel fashion in two ways:

1. by a pipelined computation (send each component y_k through a one-element pipe that computes z_k on its way to the Control Unit) 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_k from a component y_k, 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 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, $\boldsymbol{Z}$ 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 ***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 *p*attern *s*um-*s*quared *e*rror for the current pattern. This first step of backpropagation is accomplished by a pipe like the one at the end of execution, but with some additional computations:

1. ***Z*** is computed from ***Y*** as above.
2. ***Z*** and ***T*** are used to compute the error vector $\boldsymbol{\delta w}$ (see section 3.4.1 below), which will be sent back to the nodes.
3. δw_k is squared to produce δw_k^2 (for calculating pattern sum-squared error).
4. 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.)

The second step of the backpropagation phase propagates the error backward (see the right half of figure 4). 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, δu_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.

3.4.1. Computation of the Vector $\boldsymbol{\delta w}$

Some discussion of how δw is computed in the original 2-layer and transformed networks is appropriate before proceeding. 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. Each original node produces a single error value δ_k which is passed back to the previous layer. In the original output layer however, each δ_k is computed by multiplying t_k-z_k by the first derivative of the corresponding component of the output vector $\boldsymbol{Z}$, z'_k.

Each δ_k corresponds to δw_k in the transformed network, which are the components of the error vector δw. Since a transformed node stores two layers of weights, the error information in the transformed node becomes available through a single broadcast (assuming target outputs $\boldsymbol{T}$ are at the Control Unit, and therefore directly available to the pipe.) In the transformed network, just as each node outputs a vector of values for the forward phase of learning, each transformed node receives and uses not a single error value, but instead the vector δw. This description is modified and extended in section 4.5 to apply to networks with more than two layers of weights.

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 transformed network is given by

$$\psi = ceiling(\frac{q}{2}).$$

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 Section 4.4 gives a formal discussion of the execution algorithm, which is essentially an iteration over the algorithm of section 3.3. Section 4.5 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 5 illustrates the transformation of a 4-layer network. The weight layers, nodes and waves 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 second wave handles the third and fourth weight layers.

Figure 5. Example Transformation for a 4-layer Network (number of weight layers is even).

The first wave maps to a 3:5x4+1:x4 set of transformed nodes (nodes **1**-**3** in figure 5, plus bias node **4**), which accounts for the first two layers of weights and the adjacent nodes. The second wave maps to a 2:4x2+1:x2 set of transformed nodes (nodes **5** and **6** in figure 5, plus bias node **7**), which accounts for the **third** and **fourth** layer of weights and the adjacent nodes.

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.

4.2. Mapping an Odd Number of Weight Layers

Figure 6 illustrates the transformation of a 3-layer network. As in the example above, the weight layers, nodes and waves 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).

Figure 6. Example Transformation for a 3-layer Network (number of weight layers is odd).

The first wave maps to a set of 4:5x4+1:x4 transformed nodes (nodes **1**-**3** in figure 6, 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 as a set of 2:4x2+0 nodes (nodes **5**-**6** in figure 6, note that there is no bias node for wave 2).

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 Control Unit 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 its 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. Determining the Size of the Transformed q-Layer Network

Section 3.2.2 discussed determining the size of a transformed 2-layer BP network. Those ideas and notations are extended here to the q-layer network. Once the q-layer network is subdivided into 2-layer subnets as indicated in the previous subsections, determining the size of the network is basically an iteration over the steps given in section 3.2.2. One caveat to this is that if q is odd, there is no bias node required in the final wave. Having stated this, here is a formal definition. Let m_i be the number of nodes in node layer i such that $1 \le i \le \psi$. Let M be the total number of nodes in the transformed network. Then,

$$b = \begin{cases} \psi, & \text{if } q \text{ is even} \\ \psi - 1, & \text{if } q \text{ is odd} \end{cases} \tag{3}$$

$$M = \left(\sum_{i=1}^{\psi} m_{2i}\right) + b. \tag{4}$$

Let λ be the current wave, such that $1 \le \lambda \le \psi$. Then the size of a node in the current wave is the following:

$$m_{2\lambda-1} + m_{2\lambda+1} + 1. \tag{5}$$

These equations are essentially elaborations on those given in section 3.2.2.

4.4. 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 5, five input values are broadcast to the network in wave 1. Nodes 1-4 respond, each producing a four element vector of partial values, summed as described in section 3. Nodes 5-7 do not respond, or can be considered to produce $\boldsymbol{0}$ vectors of the proper size (firstly, they belong to wave 2, and therefore do not themselves contribute to wave 1 output, and secondly, they have no children that contribute to wave 1). $\boldsymbol{Z}$ for wave 1, then, is a four-element vector. Nodes 5 and 6, in wave 2, require four inputs—the $\boldsymbol{Z}$ from wave 1. During the second wave, nodes 1-4 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 2, and the result is passed to node 1, etc. The three-element $\boldsymbol{Z}$ for wave 2 is the output of the network.

This example can also be used to point out one of the main ideas of location-independence. Suppose the nodes in the network above were permuted such that the current node 1 was switched with node 4, and the current node 3 was switched with node 5. The overall output of the network would be unaffected by the change—the result computed at the root node is the same, even though some nodes' relative positions have changed.

4.5. q-Layer Learning

Section 3.4 described learning for a transformed 2-layer network. Section 3.4.1 described how $\boldsymbol{\delta w}$ is computed. The behavior of the *q*-layer transformed network is very similar to the behavior of the original BP network—unlike the 2-layer case of section 3, in which there was a single wave, each wave must now produce a vector of error values and pass it back to the previous wave.

Informally, $\boldsymbol{\delta w}$ and $\boldsymbol{w_j}$ are used to compute δu_j. δu_j and $\boldsymbol{u_j}$ are then used to produce 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 $\boldsymbol{\delta w}$ 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. Formal Execution and Learning Algorithms

5.1. Execution Algorithm

The following definitions 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.

Definitions

Note: All of the definitions below, except for q and ψ are relative to the current wave λ.. Some definitions have either explicit or parenthetical references to the current wave. This is done for clarity, and to indicate how other similar definitions are to be read.

- q — number of weight layers in the original network. If q is initially odd, modify the network per section 4.2.
- ψ — $ceiling(\frac{q}{2})$
- λ — designates the current wave: its domain is $1 \le \lambda \le \psi$
- n — number of nodes in the original node layer $2\lambda-1$, size of $\boldsymbol{u_j}$.
- m — number of nodes in the original node layer 2λ

p number of nodes in the original node layer $2\lambda+1$, 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 $\boldsymbol{u_j}$.

θ_j bias value for (original and transformed) node j.

f sigmoid function.

$\boldsymbol{X}$ pattern clamped on inputs (for the current wave), has size n.

x_i ith real-valued component of $\boldsymbol{X}$.

$\boldsymbol{u_j}$ vector of weights on inputs in a transformed 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 (node i in layer $2\lambda-1$) to hidden node j (node j in layer 2λ).

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 transformed 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

$$f(a) = \frac{1}{1+e^{-a}} \tag{6}$$

$$O_j = \begin{cases} f(net_j)\ , \text{ if } olbit_j \text{ is } 0 \\ net_j\ , \text{ if } olbit_j \text{ is } 1 \end{cases} \tag{7}$$

$$net_j = \sum_{i=1}^{n} x_i u_{ij} + \theta_j \tag{8}$$

$$o_{jk} = O_j w_{jk} \tag{9}$$

$$y_k = \sum_{j=1}^{m} o_{jk} \tag{10}$$

$$z_k = f(y_k) \tag{11}$$

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 (8), and then computes O_j according to equation (7). 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 (9)].

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 (7). 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 and definitions 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. As in the previous section, all of the definitions below are relative to the current wave λ.. Some definitions have either explicit or parenthetical references to the current wave. This is done for clarity, and to indicate how other similar definitions are to be read.

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 kth 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 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 $\boldsymbol{\delta w}$ 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

$$f'(a) = \frac{df(a)}{da} = f(a)(1-f(a)) \tag{12}$$

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

$$\Delta u_{ij}(c) = \eta x_i \delta u_j + \alpha \Delta u_{ij}(c-1) \tag{13}$$

$$\delta w_k = \begin{cases} (t_k - z_k) f'(y_k), & \text{if output wave,} \\ v_k z'_k, & \text{otherwise.} \end{cases} \tag{14}$$

$$\delta u_j = \left(\sum_{k=1}^{p} (\delta w_k w_{jk})\right) f'(O_j) \tag{15}$$

$$tsse = \sum_{r=1}^{s} psse_r \tag{16}$$

$$psse_r = \sum_{k=1}^{p} \delta w_k{}^2 \tag{17}$$

$$v_i = \sum_{j=1}^{m} d_{ij} \tag{18}$$

$$d_{ij} = \sum_{j=1}^{m} \delta u_j u_{ij} \tag{19}$$

As with the execution algorithm above, the steps of the learning algorithm are numbered for convenient reference. The main steps refer to the (static) standard BP algorithm. 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 (11)], and received by the Control Unit.

2.1.2 z_k and t_k are used to compute δw_k [equation (14)], 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 (17)].

(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 (15).

3.3 $\boldsymbol{D_j}$ is calculated according to equation (19).

3.4 $\boldsymbol{V}$ is calculated according to equation (18).

3.5 $\boldsymbol{\delta w}$ for the previous wave is calculated according to equation (14).

end

(4) Update the weights

4.1 Calculate Δw_{jk} and Δu_{ij} according to equation (13).

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 (16)

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 from figure 7). 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 [13]. 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. [17]. 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 [21]. 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 $\boldsymbol{u_j=0}$, 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

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* nodes, where *M* counts all the transformed nodes as defined previously in section 4.3. 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 section 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		***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 δw	*O(N)*

3.1	Broadcast δw	*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 transformed node(s)	*O(N+logM)*
total		***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 transformed node—at most *N* input variables and at most *N* output variables have to be sent to each transformed 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))*. An upper bound on the complexity of a single wave, in either execution or learning has been shown to be *O(N+logM)*. If there are *q* layers in the original network, then we are guaranteed that the algorithm requires exactly ceiling($\frac{q}{2}$) waves—but *q* is an upper bound on this as well. Hence, *O(q(N+logM))* is a reasonable upper bound on the complexity of the algorithms.

The authors have not yet implemented a BP model in hardware using the LIT scheme. That effort is a subject for potential future research. However, results for hardware implementations such as [8], support the predicted complexity analysis for LIT given here, even though such models differ in significant ways form LIT. [8] uses a tree, but only for communication—computing nodes exist only at the leaves. The architecture of [8] either apparently cannot, or does not, take advantage of pipelining during the storage of weights into the computing nodes or the broadcast of input values to the nodes. While the architecture and algorithms of [8] might support dynamic topologies, the authors do not discuss the subject. Taking into account architectural and algorithmic differences between the complexity analysis of [8] and the predicted complexity given here for the LIT approach, the results are nearly identical. This is encouraging.

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 helping to overcome some of these problems. The Location-Independent Transformation (LIT) is a general strategy for implementing variations of ANNs that use both static and 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:

- **Efforts to develop LITs for other ANNs, aside from those for which transformations have been developed.** LIT research to date has covered a few models with a broad spectrum of characteristics. Existing ANNs continue to evolve and new ANNs continue to be developed. As one considers potential approaches to hardware implementations of these models, the current set of LIT models may suggest an efficient approach to implementing additional models in hardware. This would, in turn, expand the applicability of the LIT strategy.
- **VLSI design and fabrication of LIT models for embedded systems and real-world applications.** Stout and Rudolph have completed a proof-of-concept implementation of a PASOCS network using the LIT approach [26]. The results were encouraging, however the project was subject to a variety of constraints that greatly influenced the results. What might be achieved using state-of-the-art technology is a subject for further research. While VLSI technology itself continues to evolve, another area of pursuit is using other emergent technologies to the implement ANNs via the LIT approach.
- **The use of LIT models in solving real-world applications.** The authors, as do others, envision ANNs in a variety of embedded systems as a "black-box" learning machine. Such embedded machines, unlike typical software simulations for example, may not require full underlying operating systems and peripherals support. Direct hardware support for neural computation (and dynamic topologies) therefore has the potential to

enhance and simplify the creation and use of embedded neural hardware. ANN models, like Backpropagation, have defined sets of applications. LIT research hopes to use LIT to enhance existing applications and to enable the development of new, unknown applications.

While the authors present LIT as a strategy for helping to enable more direct support for neural computation in hardware, the strategy could also be useful for other forms of implementation as well, such as on general-purpose parallel machines.

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 (1992) 4979-4984.

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

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

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

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

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

[8] Hämäläinen, T., H. Klapuri, Jukka Saarinen and Kimo Kaski, Mapping of Multilayer Perceptron Networks to Tree Shape Parallel Neurocomputer, In *Proceedings of the 1996 World Conference on Neural Networks*, (IEEE 1996) 962-967.

[9] 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).

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

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

[12] C. Lee, A Simple Procedure for Pruning Back-Propagation Trained Neural Networks, *Neural Networks*, *Vol. 6, #3*, (Pergamon Press Ltd, New York, 1993) 385-392.

[13] Lippman, R.P., An Introduction to Computing with Neural Networks, *IEEE ASSP Magazine, Vol. 3,* (1987) 4-22.

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

[15] T. Martinez, D. Campbell, B. Hughes. Priority ASOCS, *Journal of Artificial Neural Networks*, *Vol. 1, #3*, (1994) 403-429.

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

[17] S. Odri, D. Petrovacki, G. Krstonosic, Evolutional Development of a Multilevel Neural Network, *Neural Networks*, *Vol. 6, #4*, (Pergamon Press Ltd, New York, 1993), 583-595.

[18] 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*, (INNS Press, 1993) 775-778.

[19] D. Reilly, L. Cooper, C. Erlbaum, A Neural Model Category Learning, *Biological Cybernetics*, Vol. 45 (1981) 35-41.

[20] G. Rudolph, T. Martinez, An Efficient Static Topology for Modeling ASOCS, *Artificial Neural Networks,* (Elsevier Publishers, North Holland, 1991) 729-734.

[21] Rudolph, G., T. Martinez, An Efficient Transformation for Implementing Two-Layer FeedForward Networks, *Journal of Artificial Neurasl Networks, Vol. 2, #3*, (1995) 263-282.

[22] G. Rudolph, T. Martinez, A Transformation for Implementing Localist Neural Networks, *Neural Parallel and Scientific Computations, Vol. 3, #2*, (1995) 173-188.

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

[24] 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).

[25] A. Sperduti, A. Starita, Speed Up Learning and Network Optimization With Extended Back Propagation, *Neural Networks*, *Vol. 6, #3*, (Pergamon Press Ltd, New York, 1993) 365-383.

[26] Stout, M., G. Rudolph, T.R. Martinez, L. Salmon, A VLSI Implementation of a Parallel, Self-Organizing Learning Model, *Proceedings of the International Conference on Pattern Recognition*, (1994) 373-376.

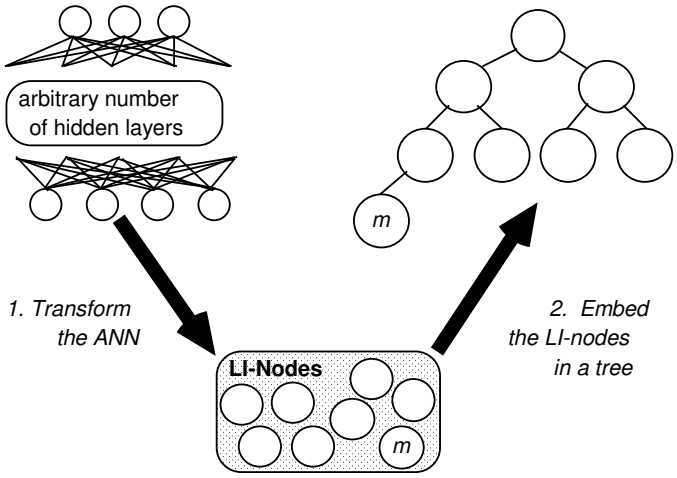

Figure 1. General LIT Transformation

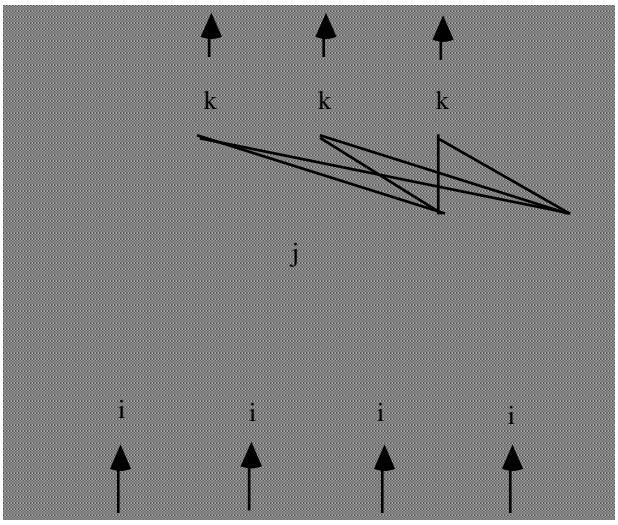

Figure 2. Hidden node *j* in a BP network.

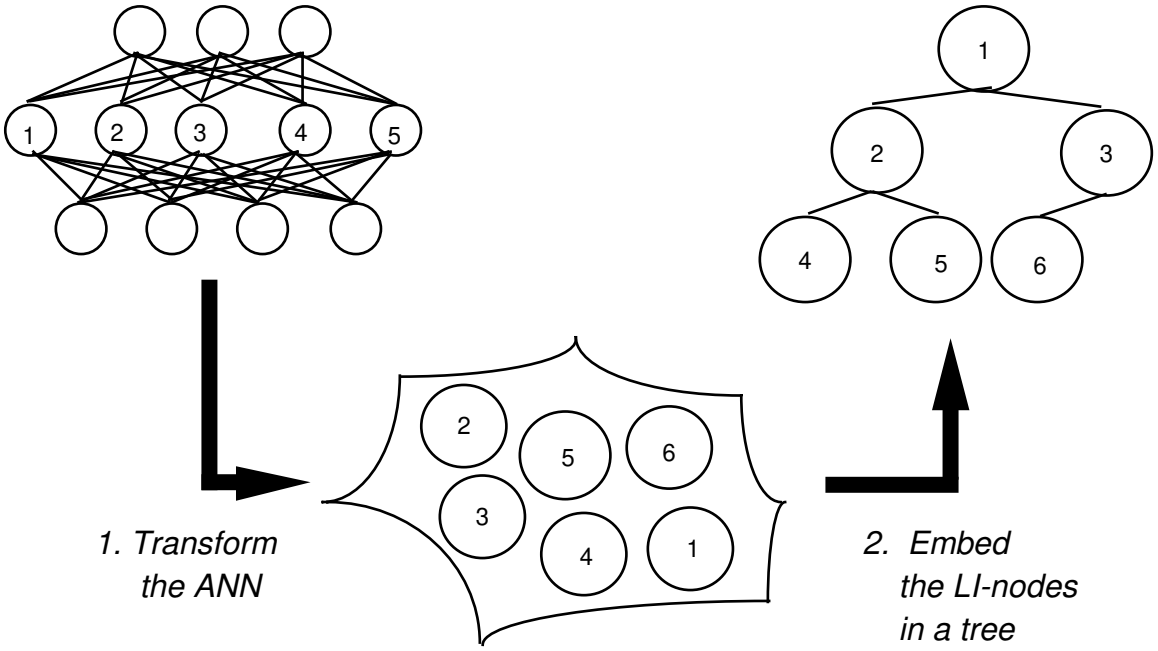

Figure 3. Transforming a 2-layer BP Network

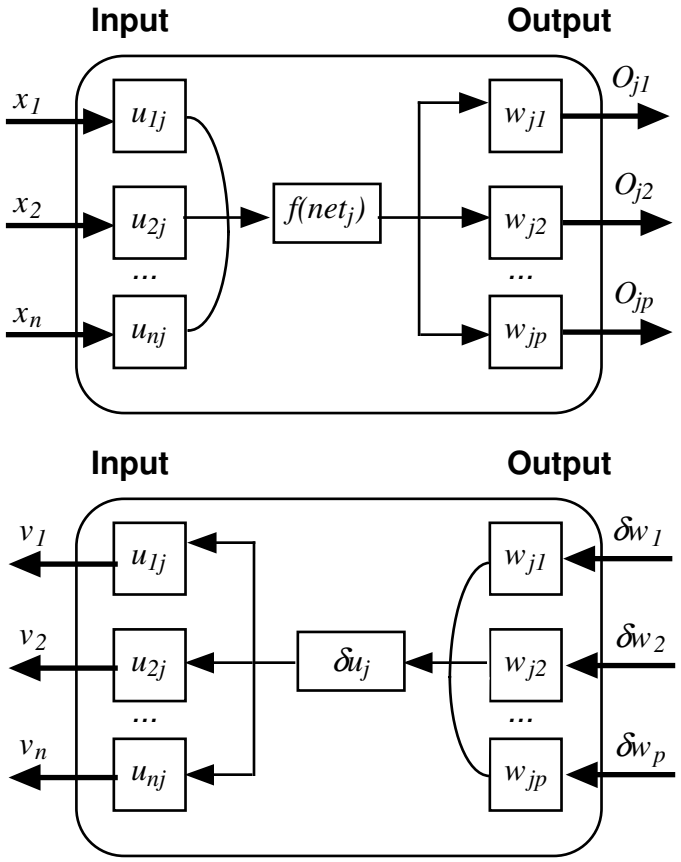

Figure 4. Two Views of a Node in the Transformed Network. (The top shows a node operating in execution mode and/or the forward phase of learning. The bottom shows a node during the backpropagation phase of learning.)

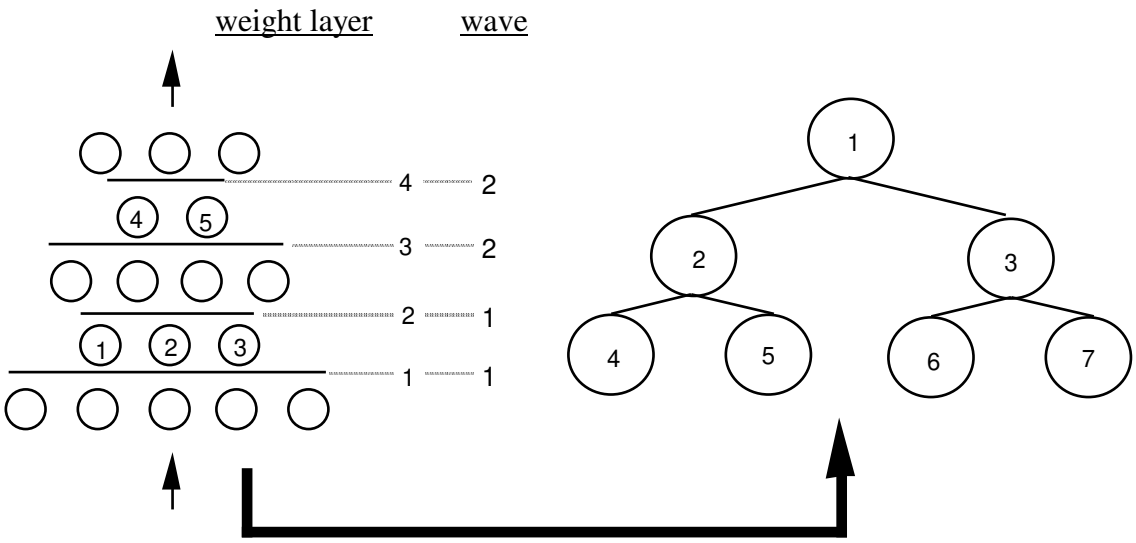

Figure 5. Example Transformation for a 4-layer Network (number of weight layers is even).

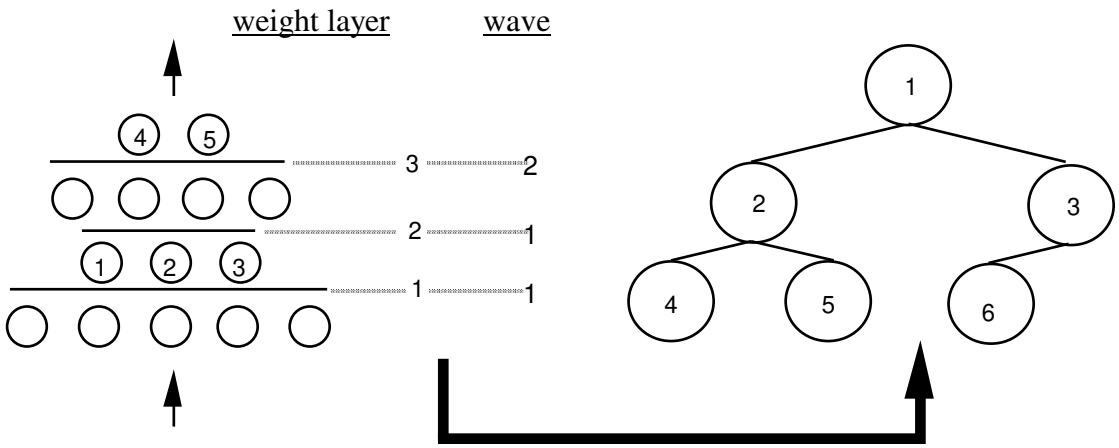

Figure 6. Example Transformation for a 3-layer Network (number of weight layers is odd).