Ada Design of a Neural Network

Jeffrey R. Carter
Boeing Computer Services
CV-70
7990 Boeing Court
Vienna, VA 22182
(703) 827-2522
72030.677@compuserve.com

Bo I. Sanden
George Mason University
Information and Software Systems Engineering
Department
Fairfax, VA 22030-4444
(703) 993-1651
bsanden@gmu.edu

Abstract

A neural network is a computer program structured as a simplified model of a brain. It contains nodes (analogous to neurons) and connections between nodes (analogous to synapses). Neural networks can solve difficult pattern-matching problems. A node sums the inputs it receives from other nodes and passes the result through a transfer function to produce its output. A modifiable weight is associated with each connection. A network is trained on a given training set of inputs. During training, the weights are successively adjusted to produce the desired output.

Classical design and implementation of neural networks are based on arrays that hold the node values and connection weights. The control structure consists of nested loops through these arrays. This paper suggests instead an object-based design where the nodes are modeled as objects to be operated on. This design models the conceptual network more closely and makes the software more understandable and maintainable. A generic Ada package representing a neural network is presented in some detail.

Introduction

A neural network is a simplified model of a brain. The primary components are nodes and connections between nodes. Nodes are analogous to neurons; connections are analogous to synapses. The values output by a node are analogous to the pulse firing rates of a neuron. Neural networks can solve difficult pattern-matching problems which resist conventional algorithmic and symbolic AI approaches.

A node sums all its inputs and passes the result through a transfer function to obtain its output. Commonly-used transfer functions include the hyperbolic tangent, used by Recursive Error Minimization networks [1]:

$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$,

and the logistic function, used by Back Propagation networks [2]:

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

All of these functions are sigmoid functions, which means they are nonlinear, S-shaped functions. They asymptotically approach a minimum value for negative values of x with large magnitudes, and asymptotically approach a maximum value for large positive values of x. They smoothly transition from the minimum to the maximum value for values of x near zero.

The output of a node passes over connections to become the input of other nodes. A modifiable weight is associated with each connection, and the value passing over the connection is multiplied by the weight of the connection to obtain the input value delivered to the receiving node. Input nodes, analogous to nerve endings, do not receive input from other nodes and do not pass values through a transfer function; instead, they distribute input values from an external source over connections to other nodes. Output nodes do not pass their output values over connections to other nodes, but provide their output values directly to an external sink.

Between the input and output nodes are intermediate nodes, which obtain their inputs from other nodes, and distribute their outputs over connections to other nodes. Intermediate nodes are sometimes called hidden nodes; this allows the letters I, H, and O to abbreviate Input, Hidden, and Output.

Neural networks are trained by presenting them with a training set of input values for which desired output values are known. A training algorithm converts the difference between the desired outputs and the network's actual outputs into adjustments to the weights on the network's connections which improve the network's response. The training set is repeatedly presented to the network until it gives an acceptable response to the entire training set.

A neural network with no hidden nodes, in which the input nodes connect directly to the output nodes, is called a perceptron. Perceptrons are limited in the types of problems they can solve. Although it is possible to use a linear transfer function in a neural network, any problem which can be solved by a neural network with a linear transfer function can be solved by a perceptron.

Similarly, with a nonlinear transfer function, it is possible to use more than one layer of hidden nodes between the input and output nodes of a neural network, but any problem which can be solved by a network with multiple hidden layers can be solved by a network with one hidden layer.

Back Propagation

Back propagation was the first successful algorithm for training neural networks with hidden nodes. It was popularized by McClelland and Rumelhart in the 1980's [2], which revitalized neural-network research after more than a decade of stagnation. For these reasons, it deserves a special place in the history of neural networks.

Because it was the first, back propagation is also the best-known algorithm. To many, "back propagation" is synonymous with "neural network," so developers who want to use a neural network in an application use back propagation. Unfortunately, back propagation suffers from a number of drawbacks:

- Back propagation is notoriously slow. The literature is filled with reports of complex back-propagation networks being trained for very large numbers of experiences, with very long elapsed times, even on fast computers. LeCun et al. report training a complex back-propagation network for three days on a Sun SPARCstation [3].
- Back propagation is not robust. Back propagation will not always reach a solution, even on very simple problems. Converging to an incorrect result is known as reaching a local minimum.
- Back propagation is difficult to use. Successful use of back propagation requires understanding the network's internals, the mathematics underlying the algorithm, and extensive experimentation with the momentum and learning-rate parameters.

- Back propagation requires the manual selection of the optimum network architecture. If the network has too few hidden nodes, it cannot solve the problem. If it has too many hidden nodes, the solution it finds will be too specific to the training set, and will not generalize to independent data.

Any one of these problems would be acceptable alone. For example, if back propagation were robust, easy to use, and could adjust the network architecture as it learned, speed would not be an issue. The combination of these problems makes back propagation, and therefore neural networks, seem too complex and unusable, suited only for experts. Neural networks are considered a curiosity.

Recursive Error Minimization

Simon and Carter presented the Recursive Error Minimization (REM) training algorithm in 1989 [1]. Unlike back propagation, REM uses second-derivative information to reach a solution. REM addresses all the problems encountered with back propagation:

- REM is very fast. Depending on the complexity of the problem and the desired level of final error, REM is one or more orders of magnitude faster than back propagation [4].
- REM is robust. Even starting from a known local minimum, REM has successfully found the true solution.
- REM is easy to use. Although REM has more parameters than back propagation, default values may be calculated for all of REM's parameters from the network architecture. These default parameters are conservative and will usually require longer training than would be necessary with more carefully-selected parameters, but the training is still faster than back propagation with optimal parameters.
- REM includes REM Thinning, a technique which allows the network to adjust its architecture during training [5]. It is easy to choose a complex architecture with too many hidden nodes for the problem; REM Thinning will then simplify the architecture to one which is appropriate to the problem.

Because REM addresses all these factors, it becomes possible for the first time to view a neural network as a component. The user need not understand the internal workings of the network to obtain satisfactory results; the user must only understand the problem to be solved. REM transforms neural networks from a curiosity to a tool.

REM Neural-Network Component Interface

A useful neural-network component must be easy for the typical client to use. The client must not need to understand the internal workings of the network to select values for parameters. But the component should provide sufficient flexibility that experienced clients may specify the network's parameters if they wish. Ada's ability to specify default values for parameters provides a convenient way to allow the typical client to ignore the network parameters, while still allowing the experienced client to provide values for specific parameters when desired. These considerations lead to the following interface for an Ada component:

```
with min_max, system;
package REM_NN_wrapper is
    type    real          is digits system.max_digits; -- inputs and outputs of
                                                       -- the network
    subtype natural_real  is real range 0.0        .. real'large;
    subtype positive_real is real range real'small .. real'large;
    type    node_set      is array (positive range <>) of real;
```

```
package real_min_max is new min_max (item => real);
use real_min_max;

generic -- REM_NN
    num_input_nodes  : positive; -- network architecture
    num_hidden_nodes : natural;
    num_output_nodes : positive;

    num_patterns : positive; -- # of different desired output sets to save

    new_random_weights          : boolean := true; -- false to reuse weights
                                                    -- for testing, use, etc.
    input_to_output_connections : boolean := true; -- must be true if
                                                    -- num_hidden_nodes = 0
    thinning_active             : boolean := true;

    -- network parameters
    beta     : positive_real := 0.1; -- learning rate; 0.1 has always been
                                     -- satisfactory so far
    -- recursive means' characteristic lengths:
    P        : positive_real := 16.0 * max (max (real (num_patterns),
                                                  real (num_input_nodes) ),
                                             real (num_output_nodes) );
        -- error & denominator of learning rule
    Q        : positive_real := 0.25 * P; -- momentum
    R        : positive_real := Q;        -- transition of desired output
    S        : positive_real := 4.0 * P;  -- G, for thinning
    -- power-law recursive means: the corresponding recursive mean will
    -- change from an exponential to power-law mean when
    -- the parameter * current experience # > corresponding parameter:
    k_P      : natural_real := 0.0;   -- P
    k_Q      : natural_real := 0.0;   -- Q
    k_S      : natural_real := 0.0;   -- S
    -- thinning parameters:
    EC       : natural_real := 0.001; -- a connection will be inactivated
                                      -- when its G value < EC
    delta_EC : natural_real := 0.001; -- a connection will be reactivated
                                      -- when its G value > EC + delta_EC

    -- random ranges: random values will be selected from the range -X .. X,
    -- where X is one of:
    random_weight_range : natural_real := 0.1;   -- initial values for
                                                 -- weights
    random_E_star_range : natural_real := 0.001; -- if > 0, the network will
                                                 -- add random noise to E*
    random_H_star_range : natural_real := 0.001; -- ditto for H*

    -- file name: store, & possibly read, network values from this file
    weight_file_name : string := "rem.wgt";

    with procedure get_input (pattern : in     positive;
                              input   :    out node_set;
                              desired :    out node_set);
    -- gets an input pattern & associated desired output pattern for this
    -- pattern #
    -- called during initialization & by respond
    -- IMPORTANT:
    -- the actual procedure associated with get_input must have been
    -- elaborated before this package is instantiated
package REM_NN is
    subtype output_id  is positive range 1 .. num_output_nodes;
```

```
         subtype output_set is node_set (output_id);

         procedure respond (pattern : in positive; output : out output_set);
         -- calls get_input for this pattern #, and propagates the input through
         -- the network to obtain the network's response

         procedure train;
         -- propagates error & derivative backward through the network, & updates
         -- the network's weights

         procedure save_weights;
         -- saves the network's values in the file with name supplied during
         -- instantiation (weight_file_name)

         invalid_architecture : exception;
         -- this package can be initialized with num_hidden_nodes = 0 and
         -- input_to_output_connections = false
         -- that combination represents an invalid network architecture
         -- the initialization of this package checks for this condition, and
         -- raises invalid_architecture if it exists
      end REM_NN;
   end REM_NN_wrapper;
```

The typical user need only supply values for num_input_nodes, num_hidden_nodes, num_output_nodes, and num_patterns. The first three represent the network architecture; the number of input and output nodes is defined by the problem. Num_patterns is also defined by the problem. For example, an OCR problem would have num_patterns defined by the number of different characters to be recognized.

The following two rules will help the client instantiate package REM_NN:

1. Set num_hidden_nodes => max (num_input_nodes, num_output_nodes). In very rare cases, this will not work. In such cases, or to use a very conservative architecture, set num_hidden_nodes => num_input_nodes * num_output_nodes.
2. Sometimes the problem does not define a value for num_patterns, or the structure of the training set makes it difficult to select values for a specific pattern class on demand. In such cases, set num_patterns => 1 and R => 1.0.

To train a network:

```
train : loop
   network.respond (...);
   network.train;
   exit train when results_are_acceptable;
end loop train;
network.save_weights;
```

Once the network has been trained, it may be used. Instantiate package REM_NN with the same network architecture, num_patterns => 1, and new_random_weights => false. Each call to procedure respond will provide the network's output for the input pattern supplied. Continuing the OCR example, an input pattern represents a character to be recognized, and the network's output classifies the character.

Classical Component Design and Implementation

McClelland and Rumelhart not only popularized back propagation as a training algorithm for neural networks, but also produced early implementations of back propagation. They represent node values by one-dimensional arrays, indexed by node identifier. These hold the output values of all the nodes in the network, for example. Two-dimensional arrays hold values such as the weights of connections.

McClelland and Rumelhart use such a scheme in the C source code for their "bp" program, which implements back propagation [6]. An example of the kind of code which uses such an implementation is

```
compute_output () {
   for (i = ninputs; i < nunits; i++) {
      netinput [i] = bias [i];
      for (j = first_weight_to [i]; j < last_weight_to [i]; j++) {
         netinput [i] += activation [j] * weight [i] [j];
      }
      activation [i] = logistic (netinput [i]);
   }
}
```

So common is this form of implementation that most neural networks are implemented in this manner, effectively reusing McClelland and Rumelhart's design, without considering any other representations. We refer to this as the classical design of a neural network because of the sheer number of networks implemented this way.

The C approach numbers nodes from zero to $N - 1$, where N is the total number of nodes in the network. In Ada it is more convenient to use a separate range for each type of node:

```
subtype input_id  is positive range 1 .. num_input_nodes;
subtype hidden_id is positive range 1 .. num_hidden_nodes;
subtype output_id is positive range 1 .. num_output_nodes;
```

Applying the classical design to the specification presented above results in an Ada version of procedure respond (corresponding to the C example):

```
procedure respond (pattern : in positive; output : out output_set) is
    net_input : real;

    -- transfer: applies the node transfer function to a weighted summed input
    --           value;
    --           calculates node output & derivative
    procedure transfer (net_input : in real; output : out real; deriv : out real)
    is
        A : real := real_math.exp (0.5 * net_input);
        B : real := 1.0 / A;
    begin -- transfer
        output := (A - B) / (A + B);      -- hyperbolic tangent (tanh)
        deriv  := 2.0 / ( (A + B) ** 2); -- derivative of tanh
    end transfer;
begin -- respond
    current_pattern := pattern;
    get_input (pattern => pattern, input => input, desired => target);

    -- calculate output & derivatives for hidden & output nodes
    -- for hidden nodes
    all_hidden : for receiver in hidden_id loop
        if active.bias.hidden (receiver) then
            net_input := weight.bias.hidden (receiver);
        else
            net_input := 0.0;
        end if;
        input_to_hidden : for sender in input_id loop
```

```
            if active.ih_value (sender) (receiver) then
                net_input := net_input + input (sender) *
                                         weight.ih_value (sender) (receiver);
            end if;
        end loop input_to_hidden;
        transfer (net_input => net_input,
                  output    => output_all.hidden (receiver),
                  deriv     => deriv.hidden (receiver) );
    end loop all_hidden;

    - for output nodes
    all_output : for receiver in output_id loop
        if active.bias.output (receiver) then
            net_input := weight.bias.output (receiver);
        else
            net_input := 0.0;
        end if;
        hidden_to_output : for sender in hidden_id loop
            if active.ho_value (sender) (receiver) then
                net_input := net_input + output_all.hidden (sender) *
                                         weight.ho_value (sender) (receiver);
            end if;
        end loop hidden_to_output;
        if input_to_output_connections then
            input_to_output : for sender in input_id loop
                if active.io_value (sender) (receiver) then
                    net_input := net_input + input (sender) *
                                             weight.io_value (sender) (receiver);
                end if;
            end loop input_to_output;
        end if;
        transfer (net_input => net_input,
                  output    => output_all.output (receiver),
                  deriv     => deriv.output (receiver) );
    end loop all_output;

    output := output_all.output;
end respond;
```

There are three main reasons why this code is longer than the equivalent C presented above:

- The possibility of inactive connections (because of REM Thinning) adds a check to each addition. The bp program has no equivalent to this.
- The Ada implementation allows connections from input nodes directly to output nodes, which are not considered by bp.
- The use of separate subtypes for each type of node makes the code more readable by expanding the two nested loops in the C version into a separate pair of loops for each pair of node types.

Object-Based Component Design and Implementation

The description of neural networks given in the introduction contains many references to the nodes of a network, but the code resulting from the classical design does not mention nodes at all. The variables in a classical implementation contain the values produced and consumed by the network. The structure of these values is not encapsulated with the subprograms which operate on them.

Object-based design encourages an implementation to model its variables on the objects in the problem, such as the nodes in a neural network, to encapsulate the types defining these objects with the operations on them, and to hide the implementation of the objects from the rest of the system [7]. A decade of experience with object-based systems has shown that they are more readable and more easily modified than systems designed using other methods, such as stepwise refinement or data-structure design methods.

Applying these principles to the design of a neural network, we see that nodes are the primary objects in the problem, and so should be central to the implementation. Once this choice is made, the designer must choose the operations applicable to a node. Clearly, nodes produce outputs in response to their inputs, and adjust the weights on the network's connections when training, so respond and train operations are appropriate.

There are also implementation decisions to make: Does a node send its output to other nodes, and so passively receive its inputs, or does a node obtain its inputs, and passively provide its output? Does the sending or receiving node adjust the weight on the connection between the two? These implementation decisions may need additional operations on nodes.

If a node obtains its inputs, it is much easier to determine when the node has received all of its inputs than if the node passively receives its inputs. The weight-updating algorithm needs many values from the receiving node, but only the output of the sending node, so the implementation does less data transfer between nodes if the receiving node updates the weights on the connections to it.

These decisions allow creation of an object-based implementation:

```
-----------
- basics about nodes:
- a node maintains the weights & related values for the connections TO itself
- a node also calculates its output value and supplies it to other nodes (those
- to which it connects) on demand

package input is - definition of input nodes
    type node_handle is limited private;

    procedure set_input  (node : in out node_handle; value : in real);
      - the node accepts its external input value
    function  get_output (from : node_handle) return real;
      - the node provides its output on demand
private - input
    type node_handle is record - an input node just provides its input value as
       output : real := 0.0;  - its output
    end record;
end input;

type weight_group is record
    weight      : real    := 0.0;
    active      : boolean := true;
    G           : real    := 2.0 * EC;
    delta_W_RM  : real    := 0.0;
    deriv_RM    : real    := deriv_lim;
end record;
type weight_set is array (positive range <>) of weight_group;

package hidden is - definition of hidden nodes
    type node_handle is limited private;

    procedure respond    (node : in out node_handle);
```

```
      -- the node collects its input & calculates its output
    function  get_output (from : node_handle) return real;
      -- the node provides its output on demand
    procedure train      (node : in out node_handle; id : in hidden_id);
      -- the node updates weights on connections to it

    -- to use pre-calculated weights, the network has to be able to set weights
    -- to save weights, the network has to be able to obtain weights
    procedure set_weight
      (node : in out node_handle; from : in input_id; weight : in weight_group);
    function  get_weight (node : node_handle; from : input_id)
      return weight_group;
    procedure set_bias_weight
      (node : in out node_handle; weight : in weight_group);
    function  get_bias_weight (node : node_handle) return weight_group;
private -- hidden
    type node_handle is record
        output : real := 0.0;
        deriv  : real := 0.0;
        bias   : weight_group;
        weight : weight_set (input_id); -- weights from input nodes to this node
    end record;
end hidden;

type star_group is record
    E_star : real := 0.0;
    H_star : real := 0.0;
end record;
type star_set is array (hidden_id) of star_group;

package output is -- definition of output nodes
    type node_handle (input_to_output : boolean) is limited private;

    procedure respond   (node : in out node_handle; result : out real);
      -- the node collects its input & calculates its output, which is provided
      -- in result
    procedure train     (node : in out node_handle; id : in output_id);
      -- the node updates weights on connections to it
    function  get_stars (node : node_handle; from : hidden_id) return star_group;
      -- the node provides weighted values of E* & H* to hidden nodes on demand

    -- to use pre-calculated weights, the network has to be able to set weights
    -- to save weights, the network has to be able to obtain weights
    procedure set_input_weight
      (node : in out node_handle; from : in input_id; weight : in weight_group);
    function  get_input_weight  (node : node_handle; from : input_id)
      return weight_group;
    procedure set_hidden_weight
      (node : in out node_handle; from : in hidden_id; weight : in weight_group);
    function  get_hidden_weight (node : node_handle; from : hidden_id)
      return weight_group;
    procedure set_bias_weight
      (node : in out node_handle; weight : in weight_group);
    function  get_bias_weight (node : node_handle) return weight_group;
private -- output

    __________
    -- an output node has connections from hidden nodes, which have weights to
    -- update
    -- the hidden nodes require propagated values of E* & H*
    -- these values must be propagated BEFORE the weights on the connections are
    -- updated
    -- because an output node's TRAIN procedure is called before the hidden
    -- node's TRAIN, the output node stores the weighted values of E* & H* in
    -- hidden_star before updating the weights

    type node_handle (input_to_output : boolean) is record
        output        : real := 0.0;
        deriv         : real := 0.0;
        bias          : weight_group;
        hidden_weight : weight_set (hidden_id);
          -- weights from hidden nodes to this node
        hidden_star   : star_set;
          -- weighted E* & H* values; see comment block above
        case input_to_output is
        when false =>
            null;
        when true =>
            input_weight  : weight_set (input_id);
              -- weights from input nodes to this node
        end case;
    end record;
end output;
```

Combining these declarations with the node-number subtypes creates the node objects:

```
type input_node_set  is array (input_id)  of input.node_handle;
type hidden_node_set is array (hidden_id) of hidden.node_handle;
subtype output_node_handle is output.node_handle
  (input_to_output => input_to_output_connections);
type output_node_set is array (output_id) of output_node_handle;
```

The network-level respond operation is implemented in terms of the nodes' operations:

```
procedure respond (pattern : in positive; output : out output_set) is
    input_value : node_set (input_id);
begin -- respond
    current_pattern := pattern;
    get_input (pattern => pattern, input => input_value, desired => target);

    -- get network response
    -- send input to input nodes
    all_input : for node in input_id loop
        input.set_input
          (node => input_node (node), value => input_value (node) );
    end loop all_input;

    -- for hidden nodes
    all_hidden : for node in hidden_id loop
        hidden.respond (node => hidden_node (node) );
    end loop all_hidden;

    -- for output nodes
    all_output : for node in output_id loop
        REM_NN.output.respond
          (node => output_node (node), result => output (node) );
    end loop all_output;
end respond;
```

The node operations used by respond are straightforward:

```
-- in package input:
procedure set_input (node : in out node_handle; value : in real) is
    -- null;
begin -- set_input
    node.output := value;
end set_input;

function get_output (from : node_handle) return real is
    -- null;
begin -- get_output
    return from.output;
end get_output;

-- in package hidden:
procedure respond (node : in out node_handle) is
    net_input : real := 0.0;
begin -- respond
    if node.bias.active then
        net_input := node.bias.weight;
    end if;

    sum_input : for i_id in input_id loop
        if node.weight (i_id).active then
            net_input := net_input + input.get_output (input_node (i_id) ) *
                                     node.weight (i_id).weight;
        end if;
    end loop sum_input;

    transfer
      (net_input => net_input, output => node.output, deriv => node.deriv);
end respond;

function get_output (from : node_handle) return real is
    -- null;
begin -- get_output
    return from.output;
end get_output;

-- in package output:
procedure respond (node : in out node_handle; result : out real) is
    net_input : real := 0.0;
begin -- respond
    if node.bias.active then
        net_input := node.bias.weight;
    end if;

    if node.input_to_output then
        sum_input : for i_id in input_id loop
            if node.input_weight (i_id).active then
                net_input := net_input + input.get_output (input_node (i_id) ) *
                                         node.input_weight (i_id).weight;
            end if;
        end loop sum_input;
    end if;

    sum_hidden : for h_id in hidden_id loop
        if node.hidden_weight (h_id).active then
            net_input := net_input + hidden.get_output (hidden_node (h_id) ) *
                                     node.hidden_weight (h_id).weight;
        end if;
    end loop sum_hidden;
```

```
        transfer
          (net_input => net_input, output => node.output, deriv => node.deriv);

        result := node.output;
    end respond;
```

The train operation has a similar implementation in terms of the node operations.

Comparison of Classical and Object-Based Versions

During execution, the primary difference between the classical and the object-based versions of the network is the extra layer of subprogram calls to the node operations in the object-based version. In absolute terms, the classical version should be faster than the object-based version. The object-based version, however, avoids array indexing to access node bias values, and substitutes one-dimensional arrays for the two-dimensional arrays of the classical version. If the subprogram calls to the node operations were eliminated by using pragma inline, the object-based version should be as fast as the classical version.

In practice, there is no speed advantage for either version. During training, applications usually display the network's progress for human review. The output operations are so much slower than the network operations that no speed difference between the two versions is noticeable. After training, during the use of the trained network, the respond operation is fast enough that the few microseconds required for each extra subprogram call in the object-based version are rarely of concern.

The other difference between the two versions concerns human understanding and modification of the network. The object-based version, unlike the classical version, contains node objects, which correspond to the reader's understanding of neural networks, so the object-based version is easier to read and understand than the classical version.

In the classical version, all the network's operations can see all the network's data. Because of this, a change to the network's hidden nodes may have an unexpected and undesired effect on the results of output-node operations. The object-based version encapsulates data with their operations, and hides the data from other operations, so a change to hidden nodes cannot affect output nodes. The object-based version is easier to modify than the classical version.

Possible Future Investigations

The nodes of a neural network are sometimes referred to as "processing elements," to reflect the parallel operation of neurons in brains. McClelland and Rumelhart use the term "parallel distributed processing" in place of "neural network" [2]. The human brain has about 10^{12} neurons, which all function in parallel.

This suggests that neural networks be implemented with concurrent nodes, to better model the characteristics of brains. In Ada terms, the network's nodes would be implemented as tasks. We expect such an implementation to be significantly slower than either of the versions presented here, although this needs verification through further investigation. On parallel systems, especially with a processor dedicated to each task, a tasking implementation should be significantly faster than a sequential implementation, especially for very large networks. The effect of a tasking implementation on single- and multiprocessor systems awaits further investigation.

References

1. Simon, W., and J. Carter, "Back Propagation Learning Equations from the Minimization of Recursive Error," *Proceedings of the IEEE International Conference on Systems Engineering*, IEEE, 1989
2. Rumelhart, D., J. McClelland, and the PDP Research Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, MIT Press/Bradford Books, 1986
3. LeCun, Y., B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, "Handwritten Digit Recognition with a Back-Propagation Network," *Advances in Neural Information Processing Systems II*, Morgan Kaufmann, 1990
4. Simon, W., and J. Carter, "Learning to Identify Letters with REM Equations," *Proceedings of the International Joint Conference on Neural Networks*, Vol. I, Lawrence Erlbaum Associates, 1990
5. Simon, W., and J. Carter, "Removing and Adding Network Connections with Recursive Error Minimization (REM) Equations," *Applications of Artificial Neural Networks*, SPIE, 1990
6. McClelland, J., and D. Rumelhart, *Explorations in Parallel Distributed Processing*, MIT Press/Bradford Books, 1988
7. Sanden, B., *Software Systems Construction with Examples in Ada*, Prentice-Hall, 1994