Processes for Automated and Machine Assisted Knowledge Discovery [1]©

Chris Lacher

1 Introduction

In previous disclosures [Lacher et al, 1992; Lacher, 1993; and US Patent 5,649,066] the author introduced a process called Expert Network Backpropagation (which we will refer to as ENBP1) that enables a form of machine learning. The context in which ENBP1 operates is an expert system whose knowledge base consists of rules of the form

```
if
  {
    H1 and H2 and ...
  }
then
  {
    C1 and C2 and ...
  }
  {
    cf = x
  }
```

where `H1`, `H2`, ... are hypotheses, `C1`, `C2`, ... are conclusions, and `cf` is a certainty factor associated with the rule; and whose inference engine reasons using forward chaining, the Stanford/EMYCIN certainty factor algebra, and fuzzy logic definitions of logical operations.

The ENBP1 process is used to adjust the certainty factors of rules in the expert system so that system reasoning is in agreement with a set of known correct input/output assertions in the knowledge domain, or optimally close to agreement if complete

agreement is not possible. ENBP1 may be used to correctly set the certainty factors of rules using I/O data without envolving a human domain expert in the otherwise tedious and time-consuming task of finding appropriate certainty factor values for all rules. Thus ENBP1 can reduce the cost of building expert systems significantly, especially in knowledge domains where reasoning chains involve several intermediate concepts, while at the same time increase the accuracy of such systems.

In the current private white paper, we discuss more machine learning processes that lead to knowledge discovery, with or without a human expert in the discovery loop. Knowledge may be discovered in two forms: new rules and new concepts.

Two public domain concepts that are used extensively in this paper are *computational network* and *supervised learning*. In order to establish common terminology and notation, we present a brief introduction and review of these here (Sections 2 and 3). Next we review the original patented ENBP1 process (Section 4).

The remainder of the paper consists of descriptions of new discoveries and processes that enable an entire family of ENBP-like processes, applicable to a commicant family of expert systems, as well as new processes for knowledge discovery in the form of rule discovery and concept discovery, the latter depending on various instances of the ENBP family.

2 Computational Networks

2.1 Terminology and Notation

Discrete time computational networks were introduced in [Lacher, 1992] based on ideas that evolved over several years of investigation. The concepts were developed further in [Lacher and Nguyen, 1994]. For our purposes here, computational networks can be thought of as providing a convenient notational framework for discussing network activation and learning that specializes both to artificial neural networks and expert networks.

A *computational network* (CN) consists of *nodes* and *connections* that are organized by an underlying directed graph model whose vertices correspond to nodes and directed edges correspond to connections. In addition, the components in a CN have functionality, as detailed in the next two paragraphs.

Each node j has a combining or gathering function (denoted here by Γ_j) and an output or firing function (denoted by φ_j). The combining function receives post-synaptic input from the nodes that are connected into node j and produces a value y_j representing the internal state of node j. The firing function takes the internal node state and produces an external firing value z_j for the node. (See Figure **??**.)

Further, each connection, say from node i to node j, has a synaptic function $\sigma_{j|i}$ that processes the output value of node i into an input value $x_{j|i}$ for node j. For this paper we will consider only computational networks with linear synapses, meaning

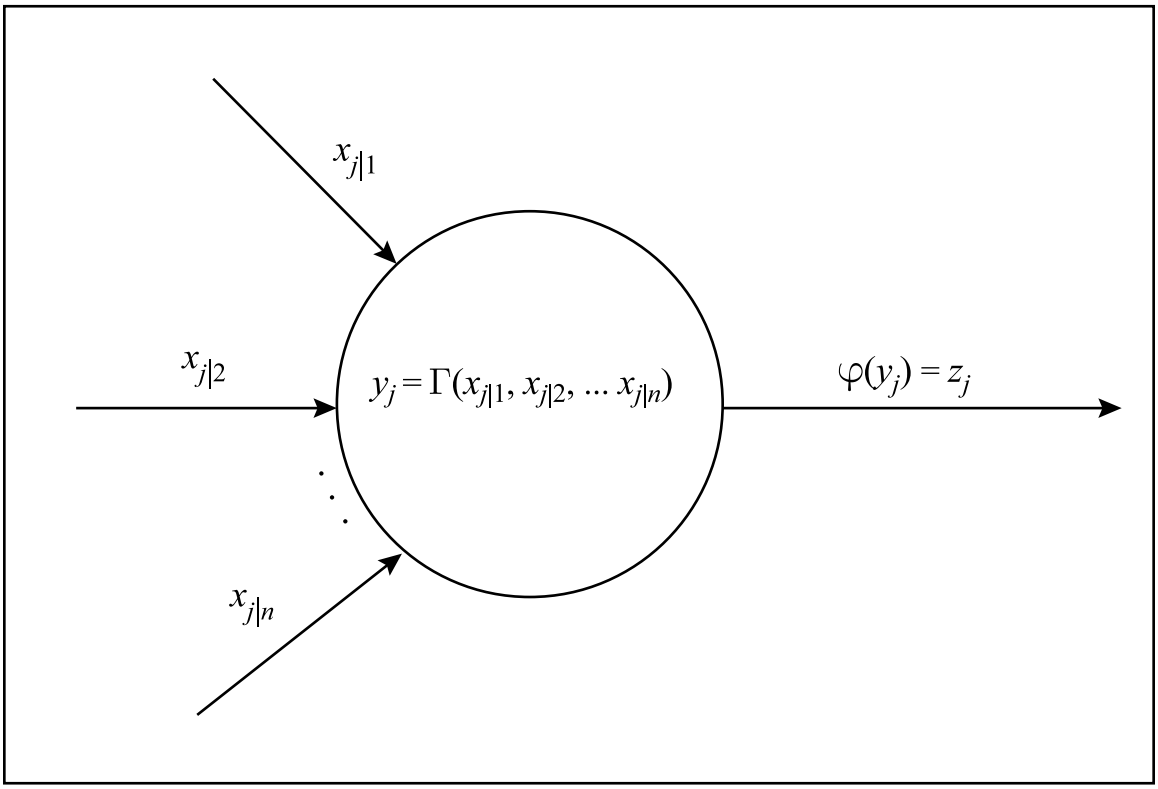

FIGURE 1. Node j in a computational network.

that $\sigma_{j|i}(z) = w_{j|i} \times z$, that is, the synaptic functions consist of multiplication by a numerical weight $w_{j|i}$ associated with the connection. (See Figure **??**.)

2.2 Activation

Nodes in a computational network may be designated as *input* in order to supply external values to begin an activation and as *output* in order to observe the result of activation. (The sets of input and output nodes may overlap, indeed in some cases each may consist of all nodes in the network.) Any nodes that are not designated as input or output are called *internal*.

Activation of a computational network consists of supplying values for input nodes, applying the component update functions repeatedly, and observing the output values of the output nodes.

Activation dynamics in general may be quite complex and are beyond the scope of our discussion here. However, in the case where the network is a directed acyclic graph (DAG) and time is discrete, the activation dynamics are straightforward: a steady state is reached after a finite number of activation steps. We will consider only discrete computational networks with the DAG property. Activation may be event driven or syncronized, the results are identical.

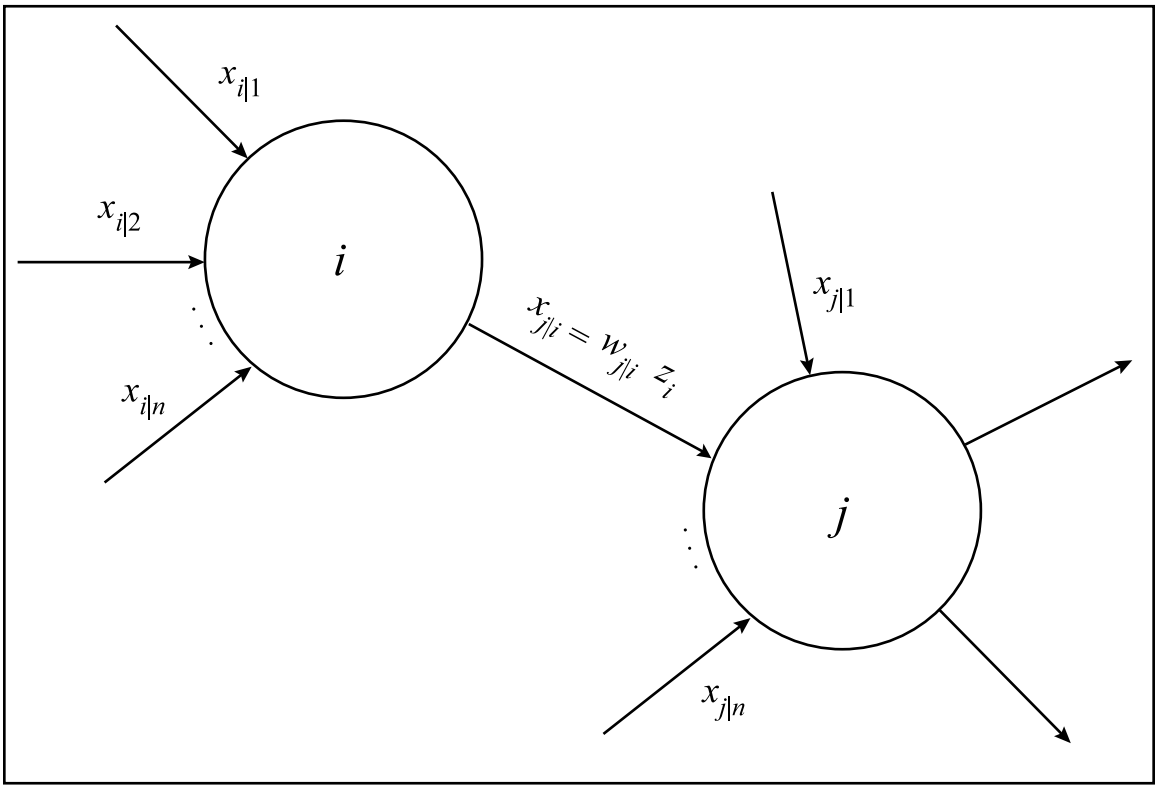

FIGURE 2. The connection from node i to node j in a computational network.

In the following discussion, we will assume a discrete time acyclic computational network with linear synapses. We will also assume that the input nodes are those without incoming connections in the network and that output nodes are those without outgoing connections in the network: in graph terminology, the input nodes are those with in-degree zero and the output nodes are those with out-degree zero. Given these assumptions (which hold for feed-forward neural networks as well as expert networks) it is possible to order the N nodes in such a way that the input nodes are subscripted $1, \ldots, K$, the internal nodes are subscripted $K+1, \ldots, K+L$, and the output nodes are subscripted $K+L+1, \ldots, K+L+M$. Thus there are K input nodes, L internal nodes, and M output nodes, and the total number of nodes is $N = K+L+M$. We also assume that all predecessors of node j have subscripts less than j.

Note that such an ordering, called a topological sort, is always attainable when the network over which activation is needed is a DAG. The assumption that the nodes are enumerated by a topological sort is not necessary, but the notation is simplified and hence the exposition is more readable.

Suppose, then, that $I_1, \ldots, I_K$ are values for the input nodes of the network. Activation is initiated by setting

$$z_j := \varphi_j(I_j) \tag{1}$$

for $j = 1, \ldots, K$ and then applying the following update rules

$$\begin{aligned} x_{j|i} &:= w_{j|i} z_i \quad , \quad i = 1, \ldots, j-1 \\ y_j &:= \Gamma_j(x_{j|1}, \ldots, x_{j|j-1}) \\ z_j &:= \varphi_j(y_j) \end{aligned} \tag{2}$$

for all remaining nodes $j = K+1, \ldots, N$. When this finite set of local activations has been completed, the entire network has been activated to a stable state. At that point the network output consists of the activation values of the output nodes: $O_j = z_{K+L+j}$, $j = 1 \ldots M$.

Thus activation of the computational network defines an input/output mapping

$$\mathbf{I} = (I_1, \ldots, I_K) \mapsto \mathbf{O} = (O_1, \ldots, O_M)$$

that is defined for any viable inputs $\mathbf{I}$ to the network.

3 Supervised Learning

Learning algorithms may be classified as one of three types: supervised, reinforcement, and unsupervised, depending on what and how data is used in the learning process. *Unsupervised learning* describes learning in which no correct result is known in advance, and the learner is simply presented with a collection of data and asked to learn whatever it might from the data. Pattern classifiers typically are unsupervised learners. In *reinforcement learning*, the learner is given feedback as to the correctness of results but knowledge of what the correct result may be is not given. Classicial conditioning is architypically a reinforcement learning process. *Supervised learning* requires that knowledge of correct results is known in advance, as in a student/teacher classroom setting.

Suppose we have a learner L that should be taught to reproduce a given set of known correct I/O data. The data is described in terms of numerical input patterns of the form $\mathbf{I} = (I_1, \ldots, I_K)$ and output patterns of the form $\mathbf{O} = (O_1, \ldots, O_M)$. The learner has the capability of computing an output pattern $L(\mathbf{I}) = \mathbf{O}$, and we also have knowledge of what the *correct* output should be for that pattern. Let's use the notation $\hat{\mathbf{O}} = (\hat{O}_1, \ldots, \hat{O}_M)$ to denote the correct output pattern that should be associated with $\mathbf{P}$. Supervised learning gives the learner access to a set of correct pattern associations

$$\mathbf{I}^p \mapsto \hat{\mathbf{O}}^p, p = 1 \ldots, P$$

and asks the learner to reproduce the pattern association, within a specified margin of error. In other words, the learner is asked to *compute* the correct output values:

$$L(\mathbf{I}^p) = \hat{\mathbf{O}}^p$$

to within specified error margin. To put it yet another way, we require that $\mathbf{O}^p = \hat{\mathbf{O}}^p$ for $p = 1, \ldots, P$, to within specified error margin. Known correct pattern associations are called *exemplars*, and a collection of exemplars to be used for learning is called a *training set*. The following meta-algorithm describes the general supervised learning process in the context established above. The meta-algorithm requires definitions of how learner modifications are implemented to obtain a specific concrete algorithm.

```
algorithm: supervised learning meta-algorithm
import:    exemplars (I[p],O[p]), p = 1 .. P // training set
           stop conditions
modify:    learner L
export:    stop_state: success/unsuccess

repeat
{
  for (p = 1 ... P) // one epoch
  {
    calculate L(I[p])
    calculate modifications for L
    if (online mode)
      modify L
    else // batch mode
      accumulate modifications separately
  }
  if (batch mode) modify L using accumulated modifications
}
until stop condition is met
return success/unsuccess
```

Note that there are two commonly used versions of supervised learning that differ based on when modifications to the learner are made. In *on line* mode, modifications are made immediately as soon as one exemplar is processed. In *batch* mode, proposed modifications are accumulated off line and made after each exemplar is processed one time. Batch mode is sometimes called *off line* mode. The stop conditions are typically an error threshold and a maximum on the number of epochs (the number of iterations of the outer `repeat` loop). The process terminates successfully if the error margin is attained and unsuccessfully if the maximum number of iterations is reached.

In this meta-algorithm, processing each exemplar in the training set one time (i.e., one trip through the `for` loop) is called a learning *epoch*. Thus batch mode makes corrective modifications at the end of each epoch, while on line makes modifications after each exemplar is processed.

Supervised learning has been studied in many contexts, including artificial neural networks (ANNs). There is a lot of published (public domain) wisdom accumulated about how to test for correctness and for the ability to generalize to exemplars that are not in a training set, including *cross validation*. Some of this applies to CNBP and ENBP learning, discussed below.

4 Computational Network Backpropagation

One of the processes patented in [USP 5649066] is the Expert Network Backpropagation learning algorithm, abbreviated ENBP1. Much if the theory behind ENBP1 can

be elevated to the context of computational networks. Our exposition continues in this section with the Computational Network Backpropagation learning meta-algorithm (called CNBP) that is made concrete to ENBP1 with certain definitions and calculations. The essence of this process is covered in [USP 5649066]. Again, the equations are derived under the assumption that the network has been topologically sorted, so that the network predecessors of node j all have index less than j. This assumption is not essential, because the network may be activated as event-driven, but it does make the notation simpler and hence the equations are easier to understand.

4.1 Influence Factors

Whenever a computational network is activated with a set of input values, assuming that the combining and firing functions for the nodes are differentiable almost everywhere, an influence factor may be calculated for each connection in the network using the following formula (which assumes that the connection runs from node j to node k):

$$E_{k|j} = w_{k|j} \times \frac{\partial \Gamma_k}{\partial x_{k|j}}(x_{k|1}, \ldots, x_{k|N}) \times \varphi'_k(y_k) \tag{3}$$

where $w_{k|j}$, φ_k, and Γ_k are as in section 2 above, $y_k = \Gamma_k(x_{k|1}, \ldots, x_{k|N})$ is the internal activation value of node k, and the partial derivative is evaluated at the vector of input values $(x_{k|1}, \ldots, x_{k|N})$ (which are the post-synaptic activation values $x_{k|j} = w_{k|j} \times z_j$ of nodes connected into node k – see Figure **??**). This influence factor is a measure of the influence, or responsibility, that node j has on the error associated with a successor node k. Note that $E_{k|j}$ is a number that depends on both the computational network structure and the activation values of the network using a particular input.

It is important to understand the notation used above. In the partial derivative $\partial \Gamma_k / \partial x_{k|j}$, $x_{k|j}$ represents the *variable* with respect to which the derivative is taken, whereas in the evaluation $(x_{k|1}, \ldots, x_{k|N})$ the various components represents a *specific set of postsynaptic values* for node k.

4.2 Calculating Influence Factors

In order to apply this characterization of influence factor it is necessary to have a way of calculating the functions and their derivatives as used in equation (**??**) above. The functions are always given by formula, and the derivatives can be calculated numerically. However, it is much better and more practical to have formulas for the derivatives as well. Closed forms for the various functional components in an expert network have been derived [USP 5649066], [Lacher et al, 1992], [Lacher and Nguyen, 1994] and are given in Section 5.3 below.

Influence factor values are calculated during a forward activation of the network. Assuming the network (or the portion of the network being activated) is acyclic, as above, the influence factors of all connections can be calculated during activation, either as an event-driven process or taking advantage of the topological sort ordering

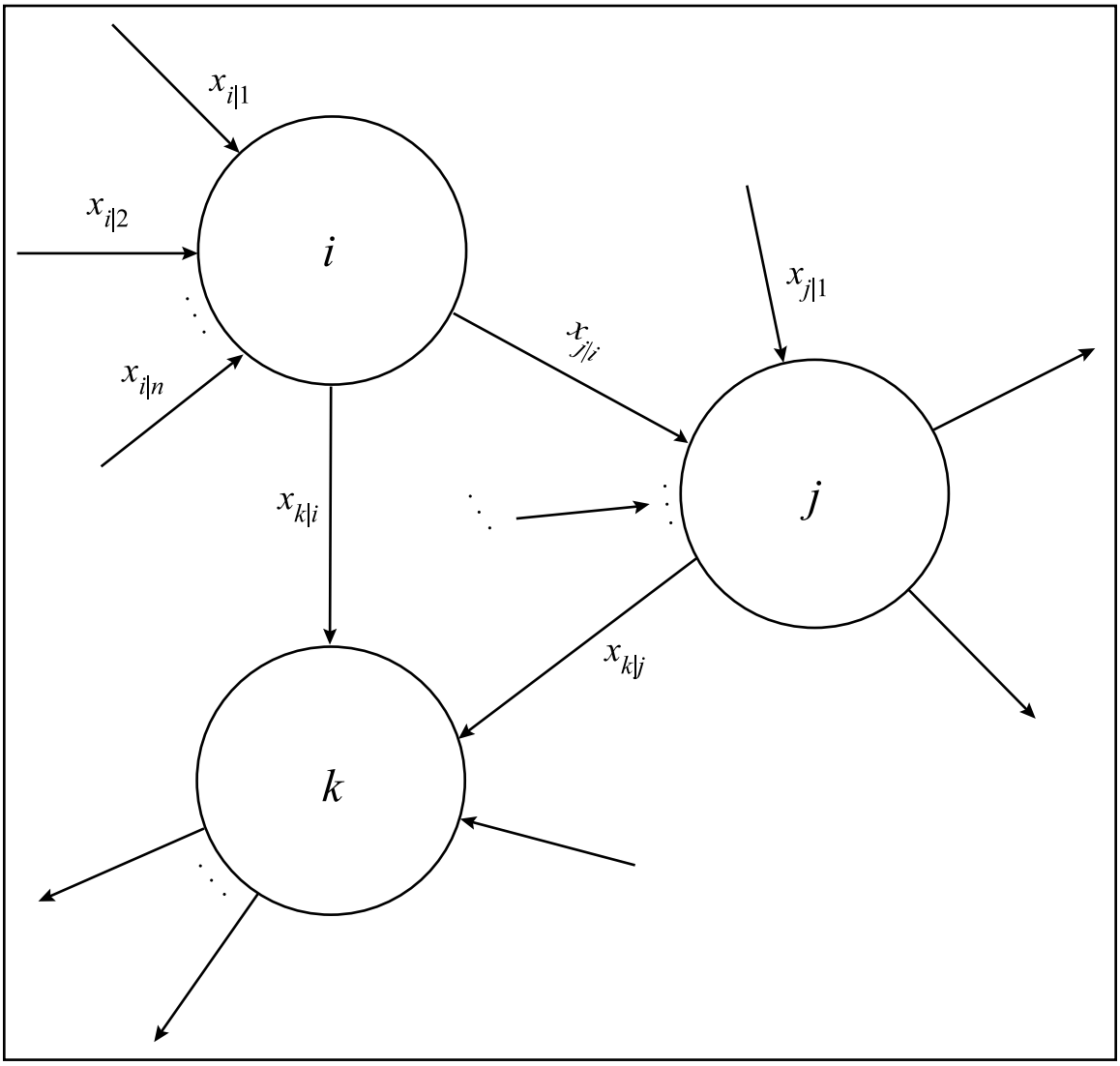

FIGURE 3. Nodes i, j, and k in a computational network. Node i connects to nodes j and k, and node j also connects to node k.

assumed in Section 4.2 above. All that is required is that equation (**??**) be applied for node k after the activation values $z_1, \ldots, z_k$ have been calculated.

4.3 Error Assignment

Having defined and calculated the relative influence each in-connecting node has on the error associated with a node, error can be assigned to each node in the network in two steps: First calculate the actual error at each output node, and second propagate these error values backward using relative influence (or responsibility) to all non-output nodes in the network.

The setting for this error assignment process is that of supervised learning, wherein we are given an *exemplar* consisting of input values $I_1, \ldots, I_K$ and associated correct

output values $\hat{O}_1, \ldots, \hat{O}_M$ obtained from some external source. Then error at the output nodes is the difference between correct value and computed value: $e_{K+L+j} = \hat{O}_j - O_j$ for $j = 1 \ldots M$. And error may be back-assigned for all non-output nodes by apportioning according to responsibility for downstream error:

$$e_j := \hat{O}_{j-K-L} - z_j \ , \ j = N \ldots N - M + 1 \tag{4}$$

$$e_j := \sum_{k>j} E_{k|j} e_k \ , \ j = K + M \ldots 1 \tag{5}$$

(Recall that $O_j = z_{K+L+j}$ in our notation.)

Error assignment is analogous to activation of a dual, or adjoint, network, whose nodes are the same as the original but whose connections have reversed direction with weights equal to the influence factors. Equation (**??**) is an initialization, analogous to equation (**??**), and equation (**??**) is an activation, analogous to equation (**??**).

4.4 CNBP Learning

Equations (**??**) and (**??**) provide an assignment of error to each node in the network. Note that it is necessary to assign error to all nodes, even those for which learning may not be desireable, because an upstream node can be assigned error only after the nodes it connects into have been assigned error.

Once having assigned error to node j, a learning rule for connections into that node may be derived using local steepest descent of the square error surface at that node. Note that local steepest descent amounts to applying the classic Widrow-Hoff learning rule at j. By calculating the gradient of square error at node j with respect to the weights on incoming connections to node j, we obtain the formula:

$$\Delta w_{j|i} = \eta e_j \varphi_j'(y_j) \frac{\partial \Gamma_j}{\partial x_{j|i}}(x_{j|1}, \ldots, x_{j|j-1}) z_i + \mu \Delta w_{j|i}^{prev} \tag{6}$$

Equation (**??**) defines one learning step with *learning rate* η and *momentum* μ in the direction of steepest descent of square error at node j.

Using equation (**??**) as the modification step in supervised learning yields the following framework learning algorithm, known as Computational Network Backpropagation, or CNBP:

```
algorithm: CNBP meta-algorithm
import:    exemplars (I[p],O[p]), p = 1 .. P // training set
           max_epochs
           max_error
modify:    weights in computational network
export:    stop_state success/unsuccess

repeat
{
  for (p = 1 ... P) // one epoch
```

```
  {
    present I[p] to network inputs                    // (1)
    activate network                                  // (2)
      and calculate influence factors                 // (3)
    calculate error at output nodes and accumulate    // (4)
    assign error to all other nodes                   // (5)
    calculate weight changes                          // (6)
    if (online mode)
      make weight changes
    else // batch mode
      accumulate weight changes separately
  }
  if (batch mode) make accumulated weight changes
}
until error <= max_error or epochs >= max_epochs
if (error <= max_error) return success
else return unsuccess
```

All that remains to have a concrete learning algorithm is concrete definitions of the computational network node funtions Γ_j and φ_j and a closed form calculation of their derivatives. These formulas are plugged in to the various named functional components in equations (**??**) ... (**??**).

This process, along with the formulae for influence factors and calculation of various partial derivatives, forms the fundamental core of the ENBP learning algorithm, patented in [US Patent 5,649,066]. In this exposition we have merely elevated the process up to the level of computational network.

5 ENBP1 Learning as Instance of CNBP

There are three broad areas discussed and claimed in [US Patent 5649066]: Translation of an expert system into a special kind of computational network called an *expert network*; a supervised learning algorithm for expert networks called *expert network backpropagation*, abbreviated ENBP1; and a reinforcement learning algorithm for expert networks called *goal directed Monte Carlo learning*, abbreviated GDMC. GDMC is used as an initializer for ENBP. In this section, we present a newly developed exposition of expert networks and the ENBP algorithms. The new way of understanding ENBP reveals insights into the various sub-processes of ENBP, making them applicable to new algorithms for knowledge discovery. The new knowledge discovery algorithms are discussed in Section 5.

5.1 Translation

The first step in applying Expert Network Backpropagation (ENBP) learning to an expert system is to translate the expert system into a certain type of computational network called an *expert network*.

The nodes in an expert network are classified as either regular nodes or logic nodes. Regular nodes are direct translates of hypothesis and/or conclusion components of rules. Thus regular nodes represent concepts from the knowledge domain that is modelled by the originating expert system. Logic nodes represent logical processing and are obtained from logical operators embedded in rule hypotheses and/or conclusions. The logical processing nodes under consideration in the present context are fuzzy NOT, AND, and NOR (also called UNK). Other logical operators can be added, requiring only that the fuzzy logical function and its partial derivatives be used to define the node type. We will assume also that rule conclusions are not compounded.

Translation of the original expert system into its expert network model proceeds in two steps [USP 5649066], [Kuncicky et al, 1992]: first create a network whose nodes represent the rule hypotheses and conclusions as they exist in the expert system, and second expand rule hypotheses into a subnetwork consisting of non-compound (aka atomic) regular, or concept, nodes feeding into a logic node, which in turn feeds into the conclusion concept node of the rule.

Weights on connections into logic nodes are set to the value 1.0 and are not permitted to change during learning, while weights of connections into concept nodes have the value of the certainty factor associated with the originating rule.

Nodes in the expert network are categorized as *input* if they are concept nodes that have no predecessor in the network and as *output* if they have no successor in the network. More concept nodes may be designated as input or output by the user of the system, where convenient. In essence, input nodes are the translates of concepts where reasoning begins and output nodes are translates of concepts that are final conclusions in the expert system reasoning process. All nodes not designated as input or output are called *internal*.

5.2 Activation

Activation of the expert network is defined by virtue of its being a discrete time acyclic computational network, so that the discussion of Section 2 applies. In particular, we assume that a topological sort order has been assigned to the nodes. Such an ordering is always attainable when the expert network over which activation is needed is a directed acyclic graph (DAG). This assumption simplifies the exposition, but is not essential to the actual working of expert networks, where the activation is event-driven.

Note that input queries to the expert system correspond directly to input values for the expert network. Therefore we have an input/output mapping

$$\mathbf{I} = (I_1, \ldots, I_K) \mapsto \mathbf{O} = (O_1, \ldots, O_M)$$

that is defined by the expert network, as in Section 2 above. Because the input values $\mathbf{I} = (I_1, \ldots, I_K)$ also represent an input query for the originating expert system, which produces a set of final conclusions using forward chaining, another input/output mapping is defined using the expert system. It is shown in the references that these

two mappings, one defined by the expert system and forward chaining, the other by the expert network and activation, are identical. Thus the expert network and the original expert system are isomorphic, that is, they reason identically.

5.3 Expert Network Functions and Derivatives

We collect here closed forms for the combining and firing functions for the Stanford/EMYCIN evidence accumulator and various fuzzy logic operators used by expert networks, along with their derivatives. These provide the plug-ins that create the ENBP1 algorithm from the CNBP meta-algorithm.

In a typical expert network, there will be many instances of all node types but only a few node types. Domain concept nodes (denoted by REG) will use the Stanford evidence accumulator, and logic nodes will use one of the fuzzy logic operators AND, OR, NAND, NOR, or NOT. We use the following notation for a node type: $x_1, \ldots, x_n$ are the post-synaptic inputs, $y = \Gamma(x_1, \ldots, x_n)$ is the internal state, and $z = \varphi(y)$ is the activation value. Note that a specific node subscript is omitted.

5.3.1 Regular Nodes

Positive and negative evidence values for regular node are given by

$$\begin{aligned} y^+ &= +1 - \prod_{x_i>0}(1 - x_i) \quad \text{and} \\ y^- &= -1 + \prod_{x_i<0}(1 + x_i), \end{aligned}$$

respectively. Positive and negative evidence are then reconciled, yielding the internal state of the node as the value of the REG combining function:

$$y := \Gamma_{\text{REG}}(x_1, \ldots, x_n) \equiv \frac{y^+ + y^-}{1 - \min\{y^+, -y^-\}}. \tag{7}$$

Note that Γ_{REG} is a symmetric function of the variables $x_1 \ldots x_n$. The only input variables which affect the values of Γ_{REG} are those from predecessors in the network.

The partial derivatives of the evidence combining function, evaluated at a specific input vector $\mathbf{x}$, are given by

$$\frac{\partial \Gamma_{\text{REG}}}{\partial x_j}(\mathbf{x}) = \begin{cases} \frac{1}{1-x_j}\frac{1-y^+}{1+y^-}, & \text{if } y^+ \geq |y^-| \text{ and } x_j > 0; \\ \frac{1}{1-x_j}\frac{1+y^-}{1-y^+}, & \text{if } y^+ < |y^-| \text{ and } x_j > 0; \\ \frac{1}{1+x_j}\frac{1-y^+}{1+y^-}, & \text{if } y^+ \geq |y^-| \text{ and } x_j < 0; \\ \frac{1}{1+x_j}\frac{1+y^-}{1-y^+}, & \text{if } y^+ < |y^-| \text{ and } x_j < 0 \end{cases} \tag{8}$$

provided $x_j \neq \pm 1$. Again it should be emphasized that on the left-hand side of equation (**??**) x_j represents the variable with respect to which we are differentiating, whereas on the right side all of the symbols represent specific values calculated by activating the network at the specific input $\mathbf{x} = (x_1, \ldots, x_j, \ldots)$.

The firing function φ_{REG} for a regular node is defined by equation

$$z := \varphi_{\text{REG}}(y) \equiv \begin{cases} y\ , & \text{if } y \geq 0.2; \\ 0\ , & \text{otherwise.} \end{cases} \tag{9}$$

so that its derivative is

$$\varphi'_{\text{REG}}(y) \equiv \begin{cases} 1\ , & \text{if } y \geq 0.2; \\ 0\ , & \text{otherwise.} \end{cases} \tag{10}$$

5.3.2 Logic Nodes

There are three logic node types used in the original expert network technology of [USP 5649066]: AND, UNK, and NOT. The combining and firing functions for these node types, along with their derivatives, are given in this section.

AND

The combining function for AND is given by

$$y = \Gamma_{\text{AND}}(x_1, \ldots, x_k) \equiv \min_i \{x_i\} \tag{11}$$

where x_i is post-synaptic input. The partial derivatives are given by

$$\frac{\partial \Gamma_{\text{AND}}}{\partial x_j}(\mathbf{x}) = \begin{cases} 1, & \text{if } x_j = \min_i \{x_i\} \\ 0, & \text{otherwise} \end{cases} \tag{12}$$

The firing function for AND nodes is the same threshold function used for REG nodes (equations **??** and **??**).

NOT

The NOT node is restricted to one input value, say x. Its combining function is given by

$$y = \Gamma_{\text{NOT}}(x) \equiv 1 - x \tag{13}$$

and its derivative is given by

$$\frac{\partial \Gamma_{\text{NOT}}}{\partial x}(\mathbf{x}) = -1 \tag{14}$$

The firing function for NOT is

$$z := \varphi_{\text{NOT}}(y) \equiv \begin{cases} 1, & \text{if } y \geq 0.8; \\ 0, & \text{otherwise.} \end{cases} \tag{15}$$

Note that the derivative of this function is zero.

UNK

The combining function for UNK is given by

$$y := \Gamma_{\text{UNK}}(x_1, \ldots, x_k) \equiv 1 - \max_i\{x_i\} \tag{16}$$

where x_i is post-synaptic input. The partial derivatives are given by

$$\frac{\partial \Gamma_{\text{UNK}}}{\partial x_j}(\mathbf{x}) = \begin{cases} -1, & \text{if } x_j = \max_i\{x_i\} \\ 0, & \text{otherwise} \end{cases} \tag{17}$$

The firing function for UNK is the same as for NOT (equation **??**). A defect in this system, which we will correct in the new discoveries, is that the zero derivatives for NOT and UNK firing functions effectively prevent error assignment back through nodes of these types.

5.4 ENBP1 Learning

Now ENBP1 learning can be re-described as the concrete CNBP algorithm obtained using the specific formulas from section 5.3.

```
algorithm: ENBP1
import:    exemplars (I[p],O[p]), p = 1 .. P // training set
           max_epochs
           max_error
modify:    weights in expert network
export:    stop_state success/unsuccess

epochs = 0
repeat
{
  for (p = 1 ... P) // one epoch
  {
    present I[p] to network inputs         // (1) + (9)...(17)
    activate network                       // (2) + (9)...(17)
      and calculate influence factors      // (3) + (9)...(17)
    calculate error at output nodes        // (4) + (9)...(17)
      and accumulate into total_error
    assign error to all other nodes        // (5) + (9)...(17)
    calculate weight changes               // (6) + (9)...(17)
    if (online mode)
      make weight changes
    else // batch mode
      accumulate weight changes separately
  }
  epochs = epochs + 1
  if (batch mode) make accumulated weight changes
}
until total_error <= max_error or epochs >= max_epochs
```

```
if (total_error <= max_error) return success
return unsuccess
```

This process, along with the formulae for influence factors and calculation of various partial derivatives, forms the fundamental core of the ENBP1 learning algorithm, patented in [US Patent 5,649,066].

This concludes the review of public domain background and processes patented in [US Patent 5,649,066]. The material in all remaining sections is new.

6 New Facts Revealed

Several facts are discovered by our innovative exposition of CNBP technology. The following are important for the new processes discussed in following Sections 8 (ENBP2) and 9 (Knowledge Discovery).

The first fact follows from inspection of equations (**??**) and (**??**), the places where derivatives are used in CNBP:

6.1 CNBP can be instantiated with any set of node functionalities whose derivatives exist almost everywhere.

Thus given any set of node functionalities whose functionalities are differential almost everywhere determines a concrete instantiation of the CNBP meta-algorithm.

The second fact follows from the local nature of weight changes given by equation (**??**) together with the separation of learning from error assignment, given by equations (**??**) and (**??**):

6.2 CNBP learning may be turned off for any set of connections.

Another consequence of separation of error assignment (**??**,**??**) and learning (**??**) is that reverse error assignment is independent of the learning step in the CNBP meta-algorithm, which then implies:

6.3 Error assignment can be made through all connections, even those for which learning has been turned off.

Similarly, there is no reason why learning must be invoked at any connection at all:

6.4 Error assignment can be made stand-alone, without a subsequent learning step.

We have stated these observations in terms of the CNBP meta-algorithm to emphasize that they apply to any of its concrete instantiations, such as the ENBP family discussed in Section 8.

7 New Components for Expert Networks

We introduce here a more numerous and upgraded collection of node types, defined in terms of their combining functions, firing functions, and their derivatives. For simplicity of reference, we will create a catalog that includes the functionalities covered in Section 5.3.

Notation is consistently as follows: y_b denotes the internal state node b, and z_b denotes the firing value of node b, and $x_{b|a}$ denotes post-synaptic input to node b from node a. Thus $x_{b|a}$ represents either external input or $x_{b|a} = w_{b|a}z_a$, where z_a is the firing value for node a. The weight $w_{b|a}$ is the certainty factor associated with the connection from a to b. Sometimes subscripts for these entities may be omitted where context is clear.

We will also use Γ and φ to denote the closed form combining and firing functions, respectively. Thus

$$\begin{aligned} x_{b|a} &= w_{b|a}z_a \\ y_b &= \Gamma(x_{b|1}, \ldots, x_{b|n}) \\ z_b &= \varphi(y_b) \end{aligned}$$

The type of a node is generally determined by the combining functionality of the node, and it has become clear that varying the firing function of a node has only a minor efefct on reasoning but a highly significant effect on learning. Therefore we catalog ours nodes according to combining function and reserve separate names for firing functions.

7.1 Combining Functions

One property that any combining function must have is symmetry with respect to its independent variables. We say that a function $y = \Gamma(x_1, \ldots, x_n)$ is *symmetric* if for any permutation $i_1, \ldots, i_n$ of its independent variables, $\Gamma(x_1, \ldots, x_n) = \Gamma(x_{i_1}, \ldots, x_{i_n})$. Symmetry is essential for combining functions to ensure that the internal states of nodes are not dependent on the order assigned to the nodes. For example, it may be convenient to use a topological sort order for the nodes, and this order may be different from the order chosen by the expert system developer, so this new order must not affect the internal state values of nodes. All of the combining functions in this white paper are symmetric.

7.1.1 Stanford Evidence Accumulator

The Stanford Evidence Accumulator is the same evidence accumulator used in the original patent, under a new name. In accumulator mode, which is used during

reasoning in an expert system, it is defined by these update equations:

$$y := \begin{cases} y + x(1-y), & \text{if both } y \text{ and } x \text{ are positive} \\ y + x(1+y), & \text{if both } y \text{ and } x \text{ are negative} \\ \frac{x+y}{1-\min(|y|,|x|)}, & \text{otherwise} \end{cases}$$

Here, y is the accumulating internal state of the node, z is the firing value for an upstream node, and $x = cf \times z$ is the input value from the upstream node.

We will call nodes defined with this functionality *stanford* nodes. The closed form combining function for stanford nodes is given by:

$$\begin{aligned} y^+ &= +1 - \prod_{x_i>0}(1-x_i), \\ y^- &= -1 + \prod_{x_i<0}(1+x_i), \\ \Gamma(\mathbf{x}) &= \frac{y^+ + y^-}{1 - \min(y^+, -y^-)} \end{aligned}$$

where $\mathbf{x} = (x_1, \ldots, x_n)$ is the vector of postsynaptic inputs to the node. The partial derivatives for the stanford combining function are given by:

$$\frac{\partial \Gamma}{\partial x_j}(\mathbf{x}) = \begin{cases} \frac{1}{1-x_j}\frac{1-y^+}{1+y^-}, & \text{if } y^+ \geq |y^-| \text{ and } x_j > 0; \\ \frac{1}{1-x_j}\frac{1+y^-}{1-y^+}, & \text{if } y^+ < |y^-| \text{ and } x_j > 0; \\ \frac{1}{1+x_j}\frac{1-y^+}{1+y^-}, & \text{if } y^+ \geq |y^-| \text{ and } x_j < 0; \\ \frac{1}{1+x_j}\frac{1+y^-}{1-y^+}, & \text{if } y^+ < |y^-| \text{ and } x_j < 0 \end{cases}$$

Commentary. The stanford evidence accumulator was introduced in the MYCIN expert system and formed the basis for reasoning under uncertainty for the EMYCIN expert systems shell (first commercialized in M1). This system has been in continuous use in textbooks and public domain software ever since. Stanford nodes with the linear threshold firing function (see 7.2.2) were called *regular* nodes in the original patent of ENBP1.

In the new algorithms introduced below, the firing function used for stanford nodes varies as required, depending on context and on the intended use of the algorithm. The partial derivative calculation, and its use in learning, were part of the original patent [USP 5649066].

7.1.2 Fuzzy AND

The combining function for fuzzy AND is given by

$$\Gamma(\mathbf{x}) = \min_i\{x_i\}$$

where $\mathbf{x} = (x_1, \ldots, x_n)$ is post-synaptic input. The partial derivatives are given by

$$\frac{\partial \Gamma}{\partial x_j}(\mathbf{x}) = \begin{cases} 1, & \text{if } x_j = \min_i\{x_i\} \\ 0, & \text{otherwise} \end{cases}$$

The firing function for AND nodes is again a variable, depending on the targeted purpose of a particular algorithm.

Commentary. The effect of this derivative is apparent in error assignment. Consider equation (**??**), repeated here, which controls how error is backpropagated in the CNBP meta-algorithm:

$$E_{j|i} = w_{j|i} \times \frac{\partial \Gamma}{\partial x_{j|i}}(x_{j|1}, \ldots, x_{j|n}) \times \varphi'(y_j)$$

The partial derivative factor acts as a multiplexer, sending error back through the connection associated with the smallest (minimum) post-synaptic input. This is intuitively appropriate, since the smallest input determines the internal state (and firing value) of the node, and hence is completely responsible for any error at the node.

7.1.3 Fuzzy OR

The combining function for fuzzy OR is given by

$$\Gamma(\mathbf{x}) = \max_i \{x_i\}$$

where $\mathbf{x} = (x_1, \ldots, x_n)$ is post-synaptic input. The partial derivatives are given by

$$\frac{\partial \Gamma}{\partial x_j}(\mathbf{x}) = \begin{cases} 1, & \text{if } x_j = \max_i\{x_i\} \\ 0, & \text{otherwise} \end{cases}$$

The firing function for OR nodes is again a variable, depending on the targeted purpose of a particular algorithm.

Commentary. Just as with AND, this derivative acts as a multiplexer, sending error back through the connection associated with the largest (maximum) post-synaptic input. OR nodes are rarely needed in expert systems, however.

7.1.4 Fuzzy NOR

$$\Gamma(\mathbf{x}) = 1 - \max_i \{x_i\}$$

$$\frac{\partial \Gamma}{\partial x_j}(\mathbf{x}) = \begin{cases} -1, & \text{if } x_j = \max_i\{x_i\} \\ 0, & \text{otherwise} \end{cases}$$

Commentary. Fuzzy NOR is another node type introduced in EMYCIN. When used with the bivalent threshold firing function it has been called the *unknown* node type. This choice for firing function was unfortunate, as it effectively prevents reverse error assignment through the node. In the new ENBP2 below, we rectify this defect by changing firing functions to the anchored logistic. (See 7.2.1 and 7.2.3 below.)

Note that NOR is equivalent to chaining NOT (with linear threshold firing function, $\tau = 0$) and OR.

7.1.5 Fuzzy NAND

$$\Gamma(\mathbf{x}) = 1 - \min_i\{x_i\}$$

$$\frac{\partial\Gamma}{\partial x_j}(\mathbf{x}) = \begin{cases} -1, & \text{if } x_j = \min_i\{x_i\} \\ 0, & \text{otherwise} \end{cases}$$

Commentary. The NAND node type is equivalent to chaining NOT (with linear threshold firing function, $\tau = 0$) and AND.

7.1.6 Fuzzy NOT

The fuzzy NOT node is restricted to one input value, say x. Its combining function is given by

$$\Gamma(x) = 1 - x$$

and its derivative is given by

$$\frac{\partial\Gamma}{\partial x}(\mathbf{x}) = -1$$

Commentary. Along with REG, AND, and UNK, NOT is the fourth (and last) node type introduced in EMYCIN. As with UNK type, NOT was paired with the bivalent threshold firing function, with the same problems for learning as those discussed for UNK. We rectify the situation in ENBP2 by disengaging the firing function from the node type, using the anchored logistic to fire UNK and NOT.

7.2 Firing Functions

We catalog previously used and one new firing function in this section. To keep the notation consistent with the use of firing functions in a computational network, we use y to denote the independent variable and $z = \varphi(y)$ the dependent variable for a firing function φ.

7.2.1 Bivalent Threshold

The bivalent threshold firing function (with $\tau = 0.2$) was originally associated with UNK and NOT nodes by EMYCIN:

$$\varphi(y) = \begin{cases} 1, & \text{if } y \geq \tau; \\ 0, & \text{otherwise.} \end{cases}$$

The parameter: τ is called the *threshold*. The derivative is constant zero:

$$\varphi^{'}(y) = 0$$

Commentary. Clearly, when inserted into the definition of influence factor given by equation (**??**), the result is zero, which prevents reverse error propagation through these nodes.

7.2.2 Linear Threshold

The linear threshold firing function was originally associated with REG and AND nodes by EMYCIN:

$$\varphi(y) = \begin{cases} y, & \text{if } y \geq \tau; \\ 0, & \text{otherwise.} \end{cases}$$

Again the parameter τ is called the *threshold*. The derivative is given by

$$\varphi^{'}(y) = \begin{cases} 1, & \text{if } y > \tau; \\ 0, & \text{if } y < \tau. \end{cases}$$

Commentary. This firing function indeed makes reverse error assignment, and hence learning, operate appropriately. Because most nodes in typical EMYCIN-based expert network are of either REG or AND type, learning with ENBP1 has been adequate. The problem of blocking error assignment through UNK and NOT nodes remains, and is solved below using the anchored logistic.

7.2.3 Anchored Logistic

The logistic function $L(y)$ has been used for many years as a firing function in artificial neural networks. It is also called the *sigmoid* function in that context. The definition of $L(y)$, along with the well known formula giving its derivative in terms of itself, are as follows:

$$\begin{aligned} L(y) &= \frac{1}{1 + \exp(-\lambda(y - \tau))} \\ L^{'}(y) &= \lambda L(y)(1 - L(y)) \end{aligned}$$

Note that there are two parameters in the logistic function: τ plays the role of *threshold*, and λ determines the maximum slope. The logistic function is strictly increasing with one inflection point at $y = \tau$. The maximum slope occurs at $y = \tau$ and has value $\lambda/4$.

We introduce the anchored logistic as a linear change of dependent variable of the logistic, arranged so that the curve $z = \varphi(y)$ passes through the points $(0, 0)$ and $(1, 1)$, that is, $\varphi(0) = 0$ and $\varphi(1) = 1$, which makes the anchored logistic appropriate as a firing function for nodes in an expert network. Here are equations defining the

anchored logistic and its derivative:

$$\begin{aligned} \varphi(y) &= \frac{L(y) - L(0)}{L(1) - L(0)} \\ \varphi^{'}(y) &= \frac{1}{L(1) - L(0)} L^{'}(y) \\ &= \frac{\lambda}{L(1) - L(0)} L(y)(1 - L(y)) \end{aligned}$$

The two parameters are inherited from the logistic function: τ plays the role of *threshold*, and λ determines the maximum slope. The anchored logistic function is strictly increasing with one inflection point at $y = \tau$. The maximum slope occurs at $y = \tau$ and has value $\lambda/4(L(1) - L(0))$.

Commentary. The anchored logistic firing function replaces the bivalent firing function in the new ENBP2 algorithm. The tunable slope parameter λ is used explicitly in other knowledge discovery algorithms.

8 ENBP2

As predicted in the discussions of Section 7, we can now state a new version of ENBP. Due to the wide variety of node types available, we will adopt the convention of establishing a *context* by citing specific node types and firing functions for an instantiation of CNBP meta-algorithm.

```
algorithm: ENBP2
meta-algorithm: CNBP
context:   expert network with the following node types:
           stanford nodes (7.1.1)
             with linear threshold firing functions (7.2.2)
           AND nodes (7.1.2)
             with linear threshold firing functions (7.2.2)
           NOR nodes (7.1.4)
             with anchored logistic firing functions (7.2.3)
           NOT nodes (7.1.6)
             with anchored logistic firing functions (7.2.3)
import:    exemplars (I[p],O[p]), p = 1 .. P // training set
           max_epochs
           max_error
modify:    weights in expert network
export:    stop_state success/unsuccess

epochs = 0
repeat
{
  for (p = 1 ... P) // one epoch
  {
```

```
    present I[p] to network inputs
    activate network
      and calculate influence factors
    calculate error at output nodes
      and accumulate into total_error
    assign error to all other nodes
    calculate weight changes
    if (online mode)
      make weight changes
    else // batch mode
      accumulate weight changes separately
  }
  epochs = epochs + 1
  if (batch mode) make accumulated weight changes
}
until total_error <= max_error or epochs >= max_epochs
if (total_error <= max_error) return success
return unsuccess
```

Note that there are many choices (contexts) one can make in defining an expert system with certainty factors and its associated expert network. Each set of choices defines an explicit expert network backpropagation learning process, as in ENBP2.

Henceforth we will refer to the original expert network backpropogation, as patented in [USP 5649066] and also discussed in Section 5.4 above, as ENBP1. In the following sections, ENBP is used to mean any of the entire family of ENBP algorithms defined by a consistent set of choices of logic nodes and their explicit functionality. Thus ENBP is taken to mean ENBP1, ENBP2, or any other consistent set of choices from the possibilities discussed in Section 7.

9 Knowledge Discovery

The ENBP processes, outlined in previous sections, are designed to adjust certainty factors on rules in a forward chaining expert system of the type described in the introduction. The process works, as has been demonstrated by careful mathematical derivation as well as emperical trials. By replacing the human expert with a process that learns from data, ENBP can result in both large savings in development time and greater accuracy in these systems.

Convergence of the ENBP algorithm is not ensured, however, and there are certainly cases where incomplete rule systems will make convergence impossible. This section of the paper addresses the question of what to do when ENBP is unable to converge.

The ENBP algorithms terminates with either `success` or `unsuccess`. Due to the possibility of sub-optimal local minima in the manifold representing square error, it is not unlikely that random restarts of ENBP may attain convergence. In fact, a monte carlo learning process called Goal Directed Monte Carlo (GDMC) learning is

designed to provide better than random restarts for ENBP. GDMC is also part of the original patent [US Patent 5,649,066]. However, again, it may be that even with random restarts assisted by GDMC, ENBP never converges satisfactorily, leaving the result: `unsuccess`.

Such a lack of successful convergence may in fact be due to failure to capture sufficient complexity in the rule base itself, so that it is a mathematical impossibility to set the certainty factors in such a way that the system reasons correctly on all exemplars in the training set.

In this section we introduce algorithms for discovering new knowledge in the form of new rules and new concepts for an expert network. We start with a discussion of the differences between expert networks and neural networks and measures of complexity for such systems. Then we address the problem of knowledge discovery in three stages: diagnosing problems with the completeness of existing knowledge captured in an expert network, discovering new rules for an expert network, and finally discovering new concepts for an expert network.

9.1 Expert Networks vs. Neural Networks

Expert networks (ENs) differ from artificial neural networks (ANNs) in two important ways:

1. ENs represent knowledge symbolically, while ANNs represent knowledge subsymbolically.

The nodes in an EN represent concepts (logic or domain), whereas a node in an ANN has no external meaning: knowledge in the EN is local, whereas knowledge in an ANN is represented globally by patterns of activation. A single activation value in an EN has external meaning, whereas in an ANN a single activation value is analogous to a pixel in an image: totally without meaning by itself, but part of a bigger picture that represents knowledge.

2. EN's tend to be sparsely connected, whereas ANNs are inherently densely connected.

An ANN requires many nodes in order to represent complex patterns, just as a digital image requires many pixels to represent a scene. Moreover, experience has shown that successful training of ANNs requires very high connectivity, often "full" connectivity, wherein each node on one layer is connected to every node in the next layer.

These distinctions are of course related, and it is a mistake to try move either technology to a state that is more like the other: ANNs need large numbers of nodes to represent complex patterns and high connectivity for successul learning, whereas ENs need to keep the number of nodes relatively small (because each represents a concept

in either logic or the knowledge domain) and the connectivity sparse (because connections represent rules, and the fewer rules that are needed to characterize a knowledge domain, the better, all things considered).

Therefore the obvious way to make ENBP converge, by adding lots of new nodes and connections, would work, but would lead to an expert network that is trending to an ANN system, which in turn would defeat the purpose of having an expert network in the first place: we want systems whose concepts and rules are accepted by human domain experts and for which logical traces are comprehensible explanations of how given conclusions are reached. Adding many nodes and connections would allow ENBP to begin to represent knowledge sub-symbolically, at which point we would have a "black box" solution.

9.2 Measures of Complexity: Rule and Connection Density

One can define the *connection density* of a network as the total number of connections divided by the total number of nodes. The connection density can be calculated and maintained in a computer display at all times during knowledge discovery.

A similar ratio is the *rule density*, consisting of the number of rules divided by the number of domain concepts in the expert system. While these two ratios each quantify a measure of the expert system complexity, the connection density is preferred due to its somewhat higher sensitivity. For example, suppose we have a rule base consisting of two rules:

```
if A and B and C then D (cf = x)
if C then E             (cf = y)
```

The rule density of this simple system is two rules divided by five concepts (`A`, `B`, `C`, `D`, `E`), or 0.40, while the connection density is five connections divided by six nodes (`A`, `B`, `C`, `D`, `E`, `AND`), or 0.83. (See Figure **??**.) Removing `C` as a conjunct in the first rule changes the connection density to $4/6 = 0.67$ but does not change the rule density.

Another advantage of using connection density is that it applies to both expert networks and ANNs. A typical ANN with n nodes has connection density $\Theta(n)$, whereas a good rule of thumb for expert networks is to keep the connection density bounded by a constant value (2.0 being a good target).

9.3 Diagnostics

In Section 9.1 above, we make the point that rules and concepts must be added to en expert network with great judiciousness in order to preserve the very benefits expert systems have over black box solutions. A first and most judicious step in solving an ENBP non-convergence problem is to provide diagnostic data that can be examined by a human expert using standard spreadsheet technology.

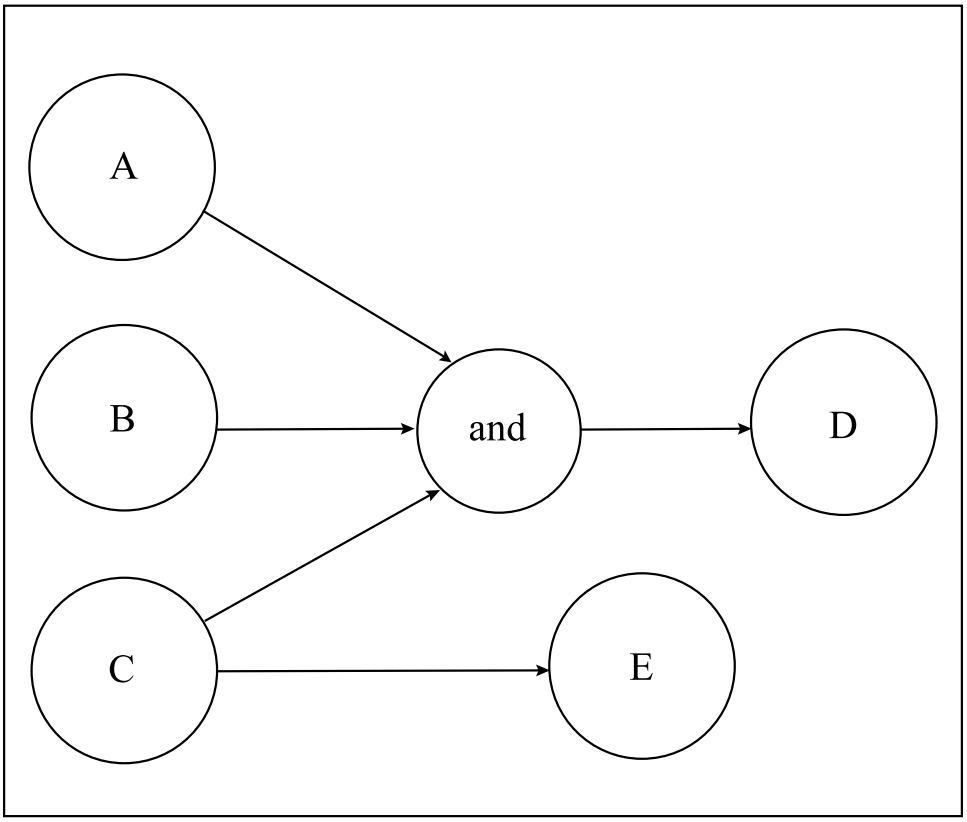

FIGURE 4. Expert network associated with simple two-rule expert system.

Suppose that we have an expert network and that we have a failure to converge by ENBP. The first question that a human domain expert might ask is: where are the problems?

Here is an algorithm that answers this question:

```
algorithm: diagnose_1
context:   expert network
import:    exemplars I[p],O[p], p = 1..P
export:    error at each node for each exemplar

for (p = 1 .. P)
{
  activate the network with I[p]                  // (1),(2)
    and calculate influence factors               // (3)
  e[p,j] = error assigned to node j by exemplar p // (4),(5)
}
return: error matrix e
```

This algorithm returns a matrix, which can be output in comma-delimited form and read by a spreadsheet with rows corresponding to exemplars and columns corresponding to nodes. Sorts and other analysis techniques can help in diagnosing where error is high which may be helpful in discovering where knowledge is "missing" from the system.

A more focused diagnostic is the following:

```
algorithm: diagnose_2
context:   expert network
import:    exemplars I[p],O[p], p = 1..P
export:    worst performing exemplar
           associated activation values
           associated error values

for (p = 1 .. P)
{
  activate the network with I[p]     // (1), (2)
  rms[p] = RMS output error for I[p] // (4)
}
m = subscript of first largest value of rms[]
// I[m],O[m] is exemplar with worst performance
a[j] = activation value of node j with input I[m]     // (1),(2)
e[j] = error assigned to node j by exemplar I[m],O[m] // (3),(4),(5)
return: m,a,e
```

The Diagnose2 process first finds the (first) worst performing exemplar and then calculates and returns (a) the exemplar identifier, (b) the activation values for the exemplar, and (c) the error values associated with the exemplar.

It is worth emphasizing here that the error assigned to nodes is a signed number: positive error values indicate that the activation value is too low and negative error values indicate that the activation value is too high. To correct for a positive error, the activation value should increase, and to correct for a negative error the activation value should be decreased.

In a visual mode, the software would show the activation values as intensities in one color and the error values as intensities in a spectrum of two orthogonal colors. For example, activation value may be indicated in green, positive error in red, and negative error in blue, or alternatively some other graphical representation may be used. (See Figure **??**.) In text mode, a similar result is produced. In any case, the software shows the user where activation is stifled and where error is accumulated. The latter shows the most likely place for a new rule to terminate, while the former are prime candidates for rule hypothesis clauses.

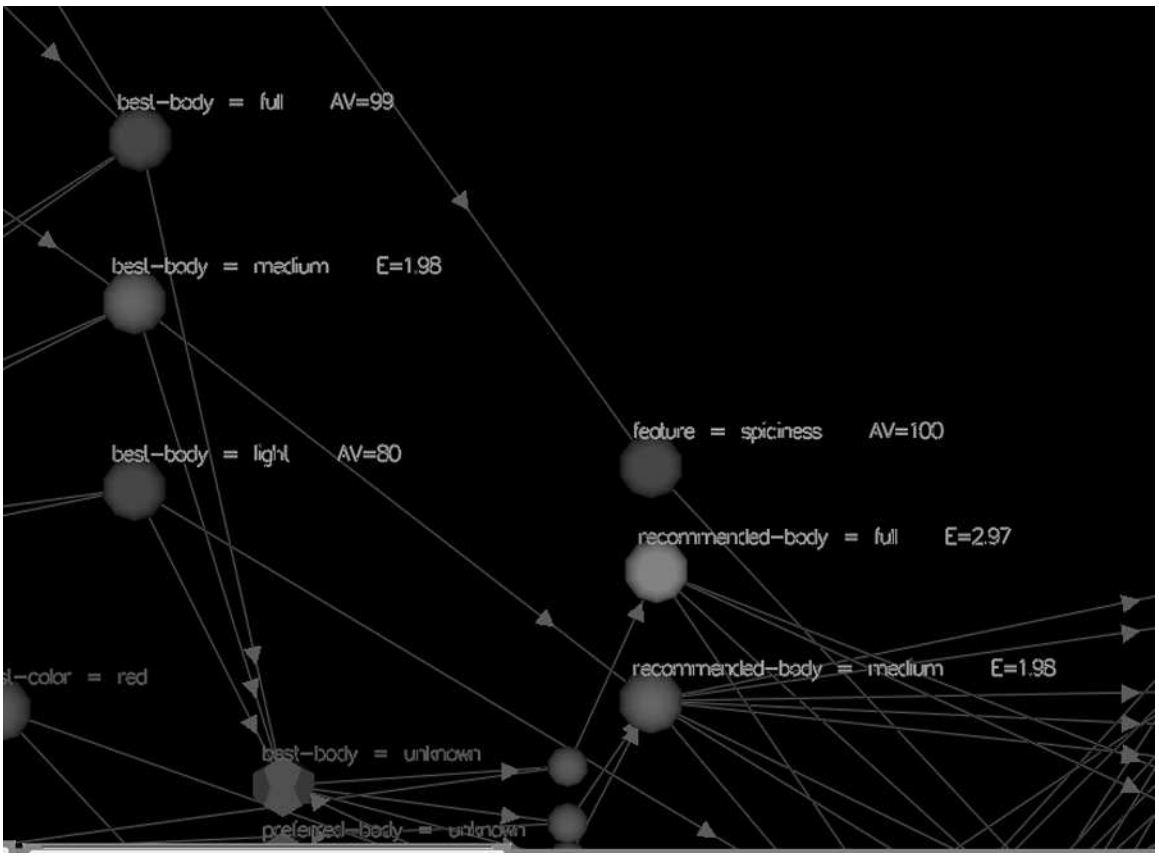

FIGURE 5. Portion of an expert network display, showing concept node with maximal error (black) and concept nodes with maximal activation (white).

9.4 Rule Discovery

The techniques of Section 9.3 can be automated to an algorithm that assembles a new rule from the places in the expert network where activation is maximally stifled and error is maximally assigned.

9.4.1 Algorithm CreateRule

...

9.4.2 Algorithm: BuildRuleBase

...

There are two modes for this rule building process. The first is automated, as decribed above. The second we refer to as the *human-in-the-loop* (HIL) mode. In HIL mode, the human domain expert is presented with the proposed new rules for evaluation and acceptance, possibly after some modification. Modifications could be asked for that would reduce complexity by eliminating any rules that are false or redundant and also eliminating hypothesis conjuncts where possible, resulting in simplified new

rules. Whenever feasible, it is recommended that a human expert review new rules created by CreateRule. The place where the processes pause to involve the human expert is marked in CreateRule.

9.4.3 Algorithm: Thinning

The rule discovery process can continue until ENBP converges successfully. At that point, especially when in automatic mode, it may be appropriate to start improving the structural characteristics of the system by decreasing the rule density (defined above in the introduction to this section). Reducing the connection density may be called "thinning". Note that thinning can be done over an extended time while the now correctly working expert system is being used. Thinning is not essential, but may be useful in cases where the rule structure built by CreateRule is complex. Thinning will not change the way the system reasons on the training set, but it may improve the clarity of explanations derived from the system.

...

9.5 Concept Discovery

Either because the connection density has grown too large or because the BuildRuleBase algorithm has been unsuccessful, there are circumstances where a given expert network needs to have its base of domain concepts increased.

The *discovery* of new concepts in a knowledge domain is an activity that is closely associated with human intelligence. Moreover, the *naming* of newly discovered concepts is closely associated with human language, communication, scholarship, and judgement. We present here an algorithm that can discover concepts that are new to a given expert network. However, there may be several equally usable choices for this new concept, distinguished by how it is connected into the network, and there will be no name for the new concept that has meaning outside the context of the original expert system. It is therefore highly recommended that the human-in-the-loop, or HIL, mode of the algorithm be applied. This will create a pause during which equally usable concepts can be vetted by a human expert, and a name may be supplied for the chosen concept.

...

9.5.1 Algorithm: CreateConcept

...

9.5.2 Algorithm: BuildKnowledgeBase

...

We conclude by re-emphasizing that unbridled growth in either concepts or connections, beyond the minimalist needs to adequately describe a knowledge domain, runs

counter to the usual goals of creating an expert system in the first place. Judiciousness in accepting new rules and concepts is called for under all circumstances.

Finally, note that the Thinning algorithm may be applied after successssfully building a knowledge base using any algorithm.

10 Application to Backward Chaining Systems

. . .

11 Application to Classical Logic Systems

. . .

References

US Patent 5,649,066, *Method and Apparatus for Refinement of Learning in Expert Networks*, Issued July 15, 1997, by US Patent Office (R.C. Lacher, David C. Kuncicky, and Susan I. Hruska, inventors).

D.C. Kuncicky, S.I. Hruska, and R.C. Lacher, The equivalence of expert system and neural network inference, *International Journal of Expert Systems* **4** (3) (1992) 281-297.

R.C. Lacher, S.I. Hruska, and D.C. Kuncicky, Backpropagation learning in expert networks, *IEEE Transactions on Neural Networks* **3** (1) (1992) 62-71.

R.C. Lacher, Expert networks: Paradigmatic conflict, technological rapprochement, *Minds and Machines* **3** (1993) 53-71.

R.C. Lacher and K.D. Nguyen, Hierarchical architectures for reasoning, Chapter 4 of *Computational Architectures for Integrating Neural and Symbolic Processes* (R. Sun and L. Bookman, eds.), Kluwer Academic Publishers, Boston, 1994, pp 117-150