NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Machine Learning of Logical Inference Rules

Michael Assels

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

April 1991

ISBN 0-315-64704-3

Canada

ABSTRACT

Machine Learning of Logical Inference Rules

Michael Assels

We develop a "neural network" method for learning logical inference rules, using a technique closely related to Pao's functional extension of pattern vectors.

We consider logical inference in a purely syntactical way as a two-step process: structural pattern recognition and classification, followed by vector transformation. Propositions are represented as vectors of integers, functionally extended in the manner of Pao. We show how a flat neural network consisting of linear threshold units, using a simple and strictly local learning rule, can learn to distinguish between suitable and unsuitable premises for any propositional inference rule and to infer the correct conclusion for most propositional rules — the exceptions being the *conjunction* and *addition* rules. A simulator has been written which builds and trains a collection of such networks, and uses them successfully to construct direct proofs.

Dedication

To Irene

Θάλαττα! Θάλαττα!

- Xenophon, *The Anabasis*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Problem in Brief

Consider the following inferences:

1. It's raining, and if it's raining then the streets are wet. Therefore the streets are wet.

2. It's snowing, and if it's snowing then it's not July. Therefore it's not July.

3. It's sunny, and if it's sunny then the sky is blue. Therefore the sky is blue.

These inferences clearly share a common pattern, and it is this common pattern that is the essence of an inference rule — *Modus Ponens* in this case. The Modus Ponens rule is traditionally represented as a template to be matched:

$$\frac{P \wedge (P \supset Q)}{Q} \tag{1.1}$$

The three inferences *fit* the template in the appropriate way — each is a substitution instance of the template — so they are justified by it; *i.e.*, if Modus Ponens is a valid rule then these are valid inferences. This kind of pattern matching is well understood, and forms the basis of programming languages such as Lisp and Prolog.

This is easy enough if the template is given, with certain symbols clearly marked as variables to be instantiated, and others as logical constants. But what happens if we approach from the other direction? Suppose we are given a collection of concrete inferences and we are required to abstract a rule from the inferences. How is this done? In particular, how can it be done by a "neural network" with no built-in inference rules, and with no built-in semantics?

In this work we will look at propositions as concrete patterns, devoid of any meaning whatever, and we will look at inference as consisting of two steps:

1. A pattern classification process in which it is verified that the input pattern — the premise — has the correct structure, and

2. A pattern transformation process in which an output pattern — the conclusion — is generated from the input.

We will discuss some difficulties associated with some known neural network techniques, and then show how an important class of inference rules can be learned by a neural network with a very simple two-layer architecture, using entirely local learning and activation rules. The locality of rules is important in two respects: First, neural networks are intended to be construed as networks of simple and independent parallel processors, operating without global control. Local rules have the advantage of minimizing communication traffic by requiring communication only among immediate neighbors.

We have implemented a simulation of this network. Its code is listed in Appendix A, and some sample deductions are shown in Appendix B.

1.2 An Important Disclaimer

Where neural networks are concerned, it is always important to stake out one's ground carefully. In particular, one must be state clearly whether one is doing psychology or computer science, since work that is useful in one area is often of little value in the other. The earliest work in the field (McCulloch and Pitts, [13]) was openly and unashamedly an attempt to provide a mathematical formalism for the description of biological neurons, and its authors had no claims to make about such issues as industrial applications or efficiency. They were doing theoretical work in the cause of an empirical science, so the value of their work is to be judged on the basis of its success or failure as a modeling tool for neurologists.

On the other hand, the backpropagation (BP) technique of Rumelhart, Hinton and Williams [17], has enjoyed considerable success in many applications, but it is open to serious challenge on at least three grounds if it is held out as a plausible model of biological processes:

First, BP's learning rule may be spatially local, but it is *temporally* non-local. The modification Δw_{ij} of the connection strength between two processing elements i and j which occurs as a consequence of the activity of i and j is not computable immediately following that activity (unless j is an output unit); the computation must be deferred until the completion of analogous updates for all processing elements lying between j and the output layer.

Second, the BP algorithm requires the computation and propagation of error signals through the network. Grossberg [5] observes that the backward pass of the BP algorithm can have no plausible physical instantiation in the brain; it would

require a duplicate nervous system, operating in reverse, to transport error signals through the network. Schmidhuber [19] argues that a mechanism can be postulated to avoid this problem, but even with this addition, BP would too complex to be plausible as a neurological hypothesis.

Third, BP is a form of supervised learning whose "teacher" can have no place in a model of genuine biological neurons. For each processing element i in the output layer, the teacher must compare i's actual output o_i with its *target* output t_i, and compute an error signal δ_i which is a function of the difference $t_i - o_i$ and the first derivative of the activation function of element i. This is obviously a nontrivial computation, and it is not credible that it could be accomplished by element i itself, simply upon being presented with the target t_i. If BP is to be taken seriously as a biological model, some physical mechanism must be postulated to play the role of teacher.

We explicitly disavow any claim that the neural network architecture presented here represents any structure in a human brain. On the contrary, what is presented here is concerned only with syntactical manipulation, and it seems to us highly unlikely that human reasoning is fundamentally syntactical. Johnson-Laird [7], and Johnson-Laird, Byrne and Tabossi [8] offer persuasive arguments to suggest that what goes on in natural reasoning has more to do with model-building or semantic inference than with syntactical manipulation of propositional symbols. The models of Grossberg [5], Carpenter and Grossberg, [2], [3], and Kohonen [10] are possible candidates for psychological reality, but ours is not. Our motivation is simply to investigate the possiblity of *machine* learning of inference rules with a local learning rule.

Chapter 2

Background Work

In this chapter we examine some neural network techniques that are relevant to our approach: backpropagation, adaptive resonance, and the functional link network. There are numerous other architectures that present interesting possibilities, especially Kohonen's "self-organizing feature maps" ([10]) and Kosko's "adaptive bidirectional associative memory" ([11]), but we have limited ourselves to these three.

2.1 The Delta Rule and Backpropagation

The "perceptron," a single-layer network of linear threshold units (Rosenblatt [16]), was an early machine learning architecture that used the "delta rule" for modifying the weight w_{ij} of a connection between an input unit i and with output o_i and an output unit j with output o_j when the target output of j was t_j:

$$\Delta w_{ij} = \eta(t_j - o_j)o_i \tag{2.1}$$

where η is the *learning rate* — a constant. In 1969, Minsky and Papert [14] showed that certain simple problems were inherently insoluble by perceptrons. The classical example of a perceptron-uncomputable function is the XOR function of two binary

inputs. These functions *can*, in principle, be computed by multi-layered perceptrons, but no general learning rule exists for these networks.

In 1985, Rumelhart, Hinton and Williams [17] showed that it is possible to formulate a general learning rule for multi-layered networks of *semi*-linear threshold units (*i.e.*, threshold units with differentiable and nondecreasing activation functions). This rule, called the *generalized delta rule*, minimizes the error (in the sense of least squares) at the output layer by performing gradient descent on the error surface in weight space, and is given by

$$\Delta w_{ij} = \eta \delta_j o_i \tag{2.2}$$

where δ_j is the error signal at unit j, given by

$$\delta_j = \begin{cases} f'(\sum_i o_i w_{ij} + \theta_j)(t_j - o_j) & \text{if } j \text{ is an output unit} \\ f'(\sum_i o_i w_{ij} + \theta_j) \sum_k \delta_k w_{jk} & \text{otherwise} \end{cases} \tag{2.3}$$

where f' is the first derivative of the semilinear activation function f, t_j is the desired output of unit j, o_j is the actual output of unit j, and θ_j is the threshold of unit j. A commonly used activation function is

$$o_j = f(\sum_i o_i w_{ij} + \theta_j) = \frac{1}{1 + e^{-(\sum_i o_i w_{ij} + \theta_j)}} \tag{2.4}$$

The first derivative of this function is

$$f'(\sum_i o_i w_{ij} + \theta_j) = o_j(1 - o_j) \tag{2.5}$$

Substituting equation (2.5) in equation (2.3), we get

$$\delta_j = \begin{cases} o_j(1-o_j)(t_j - o_j) & \text{if } j \text{ is an output unit} \\ o_j(1-o_j)\sum_k \delta_k w_{jk} & otherwise \end{cases} \tag{2.6}$$

From equations 2.2 and 2.6, it is evident that the Δw_{ij} are recursively computable, beginning from the output layer. This is the basis of the well-known backpropagation technique in which activation is propagated in a forward pass, and the error signal used for weight adjustment is propagated in a separate backward pass.

Backpropagation networks are generally capable of a settling into a weight assignment which minimizes the average error of the network's output as against the target, and this is indeed a highly desirable property, but there is a price paid: As mentioned in the introduction, backpropagation is temporally nonlocal, requiring a great deal of communication traffic within the network at training time to pass error signals from the output layer to the input layer. Moreover, for each layer in the net there is a propagation delay, and it would be desirable to eliminate this if possible.

2.2 Adaptive Resonance Theory

The work of Grossberg and Carpenter on Adaptive Resonance Theory (ART) follows an entirely different strategy. ART networks are unsupervised; *i.e.*, the model does not require a teacher to present "correct" patterns. On the contrary, the notion of correctness plays no real part in the model; the dominant concept is similarity. An ART network learns to classify patterns according to their resemblance to previously presented patterns.

Figure 2.2 shows an ART1 architecture, which is significantly simpler than its ART2 counterpart, but classifies only binary-valued patterns. Broad white arrows indicate excitatory connections, striped arrows indicate inhibitory connections, the

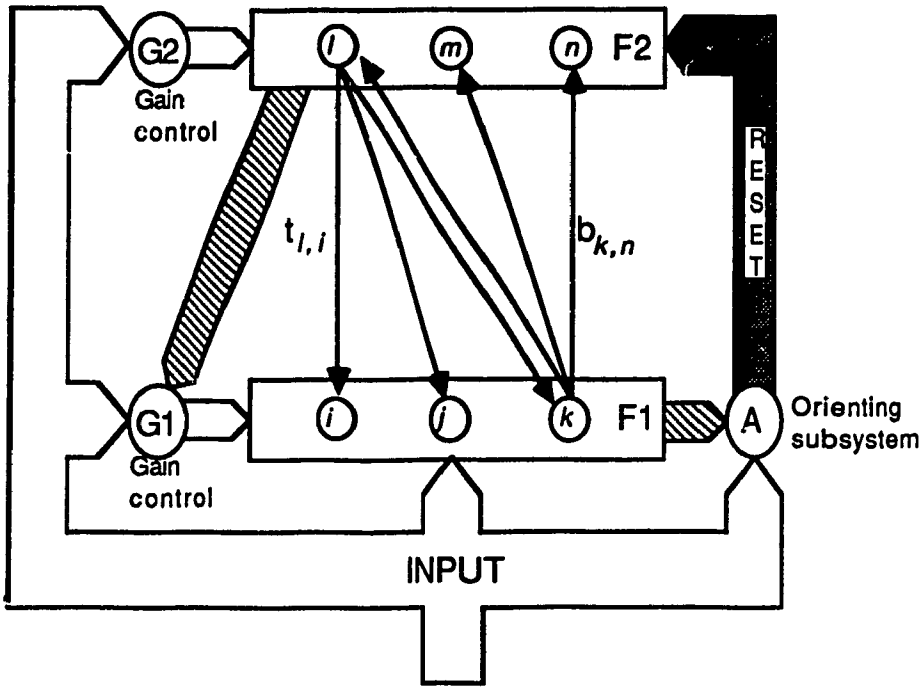

Figure 2.1: An ART1 architecture. Top-down and bottom-up LTM connections are shown only for one node in each direction.

"reset" connection enduringly inhibits all active F2 nodes. A rough sketch of the model's behavior follows:

1. A pattern is presented at the input "bus," exciting the corresponding STM nodes in F1, as well as the gain control units and the orienting subsystem (whose internal details we can ignore). All of the F1 units also receive excitation from gain control ($G1$). STM units, in both F1 and F2, obey a "2/3 Rule": They become active if, and only if, they receive excitatory input from two sources. In the present case, this means that F1 nodes copy the input pattern.

2. The activity in F1 *inhibits* the "orienting subsystem" A, counterbalancing the excitation of it receives from the input and effectively preventing a reset signal.

At the same time, F1 stimulates F2 by way of the bottom-up LTM connections.

3. F2 nodes compete amongst themselves by means of mutual inhibition (not shown in Fig. 2.2) until only one — the one that received the greatest net excitation from F1 — remains active. This node represents the class to which the input pattern has been tentatively assigned, subject to verification in subsequent steps.

4. The active F2 node, say l, stimulates an F1 node, say i, in proportion to the strength of the top-down LTM connection $t_{l,i}$ between them. The top-down LTM connections from l to the nodes of F1 constitute a "template" pattern for node l. Because the gain control unit $G1$ is inhibited by l, the 2/3 Rule ensures that an F1 node, say j, will be active if, and only if, it represent a feature that is present in both the input pattern and node l's template pattern.

5. Some number δ (possibly zero) F1 nodes will be turned off by the top-down template, and the inhibitory signal to the orienting subsystem A will be reduced by an amount proportional to δ. Note that δ is the *Hamming distance* between the input vector and the top-down template.

6. If $\delta < \rho$, where ρ is an arbitrary "vigilance" parameter representing a threshold for the orienting subsystem A, then A will remain inactive. In this case, equilibrium has been reached, and the system resonates in its current state until the input pattern is removed. During this period, both the top-down and the bottom-up LTM connections learn quickly, effectively adding the current input pattern to the class recognized by node l, and slightly altering the top-down template for l so as to reflect the new influence of the current pattern.

7. If $\delta \geq \rho$, then A is activated, and a "reset" signal is sent to F2. This is an enduring inhibitory stimulus that affects only the active node of F2 — in this case, l. The effect is to turn l off, and keep it off for a relatively long time. This extinguishes both the top-down template from l and the inhibition of $G1$, thus allowing the reestablishment of the input pattern in F1. The process returns to step 2, and continues until an equilibrium is reached, possibly by creating a new recognition class for the present input pattern alone, if no suitable template is generated by any of the committed nodes of F2.

Given that our problem is at least partly a pattern classification problem, it might seem at first as though an ART network would be an appropriate choice. Unfortunately, there is a problem here. ART networks learn to group patterns together into clusters whose members are all similar. But what does *similar* mean? Inevitably, it means "within ρ distance units of the top-down template," where ρ is the vigilance parameter. In the case of ART1, described above, distance is the Hamming distance; in ART2 it can be, *e.g.*, the Euclidean distance. In any case, ART nets will depend implicitly on some metric as the classification criterion for patterns. Moreover, the metric must be built-in rather that learned, since the classification decision is effectively made at A, based on signals received along non-LTM paths from F1 and from input.

For many pattern classification purposes this may well be quite adequate, but for classifying premises according to applicable inference rules, it is not. Consider the premises

1. $(A \supset B) \wedge A$

2. $(P \supset Q) \wedge P$

3. $((A \vee P) \supset (Q \wedge B)) \wedge (A \vee P)$

4. $(A \supset B) \wedge B$

5. $(P \supset Q) \vee P$

6. $((A \vee P) \supset (Q \wedge B)) \wedge ((A \vee P) \supset (Q \wedge B))$.

Is there a *metric* that will allow us to classify 1, 2, and 3 in a cluster that does not contain any of 4, 5, and 6? The question is perhaps premature, because we have not yet settled on any particular vector representation of propositions, but we may at least say that it will be a formidable task to find a metric that puts 1 closer to 3 than to 4.

Moreover, distance seems an inappropriate concept where inference is concerned. A proposition *is* or *is not* correct as a premise for a given rule, so distance from a cluster center ought not to be significant. Instead of a *distance metric*, what we really need is a *structural identity criterion* for propositions that is implicit in their representation; *i.e.*, we need a representation for propositions that carries with it a simple indication of the proposition's structure, such that a network can compare the structure with a template and determine whether they are identical. Such a representation can be obtained by extending a basic vector representation to express "higher order" information about the proposition.

2.3 The Functional Link Net

The idea of extending a basic vector is due to Pao and Klassen [15] [9], and is supported by the work of Sobajic [20], who showed that the difficulties exposed by

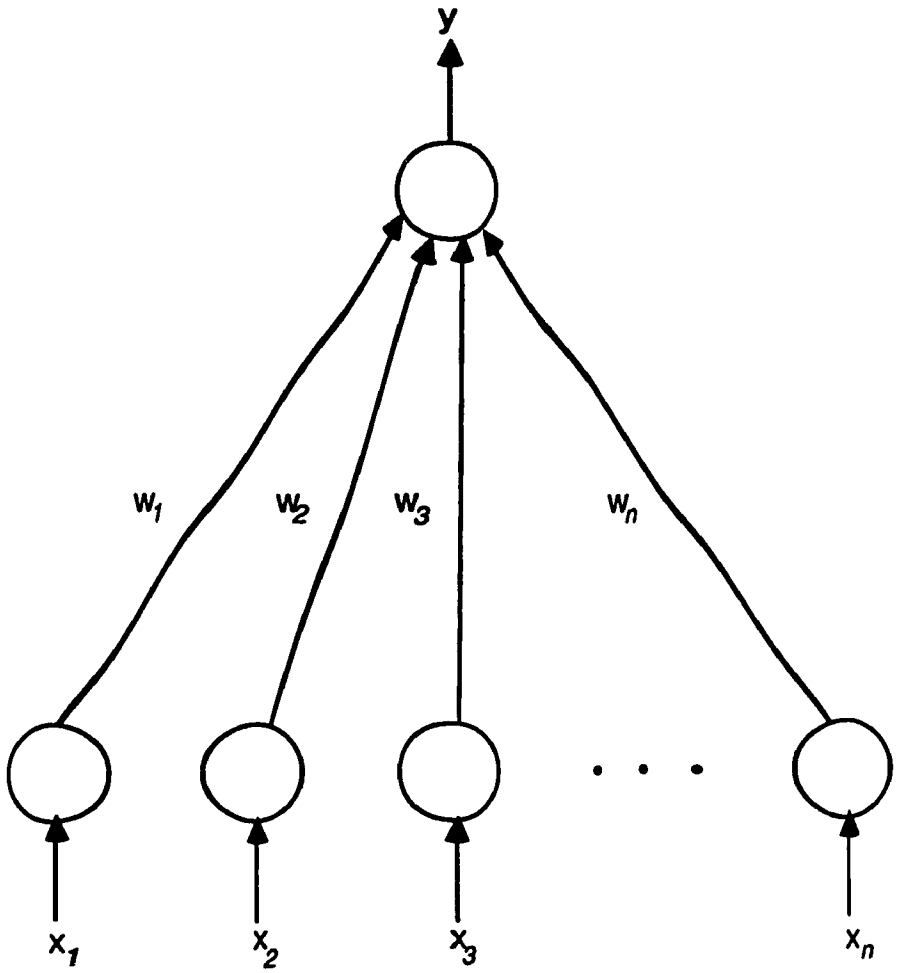

Figure 2.2: Flat network with single output unit

Minsky and Papert [14] could be overcome in a flat network whose input nodes were "functionally extended" so as to increase the dimensionality of the pattern. Consider a set of P independent patterns, each with N features. Let each pattern $\vec{x}^{(i)}$ be associated with a single value $y^{(i)}$. The task is to train a flat network like the one shown in figure 2.2 to produce the output $y^{(i)}$ when presented with the input $\vec{x}^{(i)}$.

For simplicity, we assume that the input units are simply placeholders (*i.e.*, their output is identical to their input), and that the output unit has the linear activation function

$$y = \sum_{i=1}^{N} x_i w_i \tag{2.7}$$

Then our problem is to find values for the w_i such that

$$\begin{aligned} y^{(1)} &= \sum_{i=1}^{N} w_i x_i^{(1)} \\ y^{(2)} &= \sum_{i=1}^{N} w_i x_i^{(2)} \\ &\vdots \\ y^{(p)} &= \sum_{i=1}^{N} w_i x_i^{(p)} \end{aligned}$$

This is essentially the problem of solving the linear matrix equation

$$\begin{bmatrix} x_1^{(1)} & \cdots & x_n^{(1)} \\ \vdots & \cdots & \vdots \\ x_1^{(p)} & \cdots & x_n^{(p)} \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(p)} \end{bmatrix} \tag{2.8}$$

or more briefly

$$X\vec{w} = \vec{y}. \tag{2.9}$$

Solving for $\vec{w}$ is trivial when $P \leq N$. The case where $P > N$ is both more interesting and more likely to arise in a real pattern recognition context. In this case, we can increase the number of input nodes in the network by $P - N$, and present not just the original pattern vectors $\vec{x}^{(i)}$, but vectors of size P consisting of $\vec{x}^{(i)}$ and $P - N$ orthogonal functions of the elements of $\vec{x}^{(i)}$. (The orthogonal functions can be taken from the Fourier expansion, for example.) Then we solve a new equation

$$X_{\text{ext}}\vec{w}_{\text{ext}} = \vec{y} \tag{2.10}$$

for $\vec{w}_{\text{ext}}$. In this equation, X_{ext} has dimensionality $P \times P$, $\vec{w}_{\text{ext}}$ has dimensionality $P \times 1$, and $\vec{y}$ has dimensionality $P \times 1$, so

$$\vec{w}_{\text{ext}} = X_{\text{ext}}^{-1}\vec{y}. \tag{2.11}$$

Since equation 2.11 is a linear matrix equation, it's solution can be found by a flat network, similar to the network of figure 2.2, but with more input nodes. Moreover, this result is quite general: Anything that can be learned by a multilayer network using the generalized delta rule of equation 2.2 can, in principle, be learned a flat network with functional extension of the input vectors. In practice, however, it is unreasonable to follow this method strictly. In most interesting cases, there will be many more patterns (P) than features (N), or perhaps an indefinite number of patterns, so the addition of $P - N$ functions to each input vector will be either too expensive or impossible. The practical approach is to compromise: Use data compression techniques where possible to reduce P; extend the input vectors by fewer than $P - N$ functions, and settle for a "best fit" solution instead of an exact solution.

As an example of suitable functional extensions, Pao takes the vector "outer product", in which each pair of elements of the vector is multiplied together. This can be iterated as often as seems appropriate. Thus, for an original pattern vector

$$[x_1, x_2, x_3],$$

a fully extended vector after one iteration would be

$$[x_1, x_2, x_3, x_1x_1, x_1x_2, x_1x_3, x_2x_1, x_2x_2, x_2x_3, x_3x_1, x_3x_2, x_3x_3].$$

We can remove those vector elements with repeated subscripts, and since multiplication is commutative, we can immediately eliminate obvious redundancies to maintain a manageable vector size:

$$[x_1, x_2, x_3, x_1x_2, x_1x_3, x_2x_3].$$

In a typical pattern recognition context, we might further reduce the size of the vector by using straightforward statistical methods to weed out higher-order features that are uncorrelated with the target outputs.

The advantages of the functional link net over standard backpropagation (BP) are fairly clear:

1. Reduced learning time: In effect, the BP net's hidden nodes must learn to *compute* the functions that the functional link net takes as input.

2. Reduced network size: If a net has M inputs and N outputs, then adding one hidden node adds $M + N$ weights, whereas adding one input node adds only M weights.

3. Reduced complexity: Since functional link nets are flat, they can be trained using the delta rule of equation 2.1 rather than the generalized delta rule of equation 2.2, which is required for BP. This makes coding easier, and may even strengthen the claim to biological plausibility.

There is one main drawback to the function link approach: It's cheating! In reality, only the easiest aspect of the learning problem is being learned. The rest —

the part that requires hidden layers in BP nets — is handled algorithmically by the preprocessing that computes the functions that extend the input vector. Of course, if the objective is pattern recognition, then cheating of this kind is to be encouraged; nothing in the rules of the pattern recognition game requires that problems be solved by learning. Our objective is different, however: We *are* interested in learning *per se*, and wish to avoid "hard-wired" computation wherever possible.

2.4 Neural Networks and Logic

Some attention has been paid in the literature to issues relating neural nets and logic. For example, Ballard and Hayes [1] looked at unification in neural networks, using a local representation; Touretzky and Hinton [21] presented a distributed architecture capable of performing rudimentary inferences on clauses possibly containing a single variable in a fixed position. (Both of these studies used *built-in* inference rules.) Williams [22] studied the logical aspects of activation functions. There does not seem to be much literature on the learning of inference rules.

Chapter 3

Neural Network Architecture for Inference Learning

3.1 Input Units, Output Units, and the Bad Flag

For learning a single inference rule, we use an architecture (see Figure 3.1) that is similar in some respects to those of Carpenter and Grossberg [2], [3], and in other respects to the functional link architecture of Pao [15]. It consists of an array of input units completely linked to an array of output units by connections of variable strength. Following a long tradition, we sometimes refer to the activity of the units as "short term memory (STM)," and to the variable connection strengths as "long term memory (LTM)." There is also a special classifying unit, which we call the "bad flag." It is very much like an output unit, but it has a strong inhibitory connection to the input units, which effectively shuts down the network when it becomes active. We interpret the activity of the bad flag as a measure of the *inappropriateness* of the premise (the input vector) for the inference rule.

Each array has one unit for each element of the propositional vector. Each unit in the input array has a connection of variable strength with each output unit, and with the classifying unit. The LTM connection strengths, often called "weights," vary continuously on the unit interval.

All units in the network are linear threshold units with threshold zero, except the bad flag, whose threshold may be slightly higher to allow some error.

The input units receive excitatory stimuli from sources outside the network, which may be other networks, or perhaps simply a bus from a connected source of propositions, as in our simulation. The function of these stimuli is straightforward: They cause a proposition to become active in the input STM. The STM of the input units is propagated by way of plastic (*i.e.*, modifiable) connections to both the output and the bad flag units.

The output units are functionally identical in all respects to the input units, but receive different stimuli: External control (active only during the "lecture" phase of learning), and excitatory stimuli from the input STM, filtered by the LTM connections. External control stimuli serve to hold the correct proposition in STM during learning, but after learning has been completed, this function is taken over by the input STM. It is assumed that the output STM will be propagated beyond the network by some means, as appropriate (*e.g.*, by bus to a store, as in our simulation).

Like the output units, the bad flag receives stimuli from external control during lecture learning only, but the control stimuli are always overwhelmingly inhibitory, forcing the bad flag's STM to zero. It also receives excitatory stimuli from the input units (filtered through LTM connections). The bad flag's special purpose is to turn off the input units if their STM does not represent a legitimate premise for the inference rule that has been learned by the network. This is accomplished by means of a simple "reset" mechanism, similar to that "orienting subsystem" of the ART1 architecture of Carpenter and Grossberg ([2]): The strong inhibitory connections from the bad flag to the input units overwhelm the excitatory control signals and

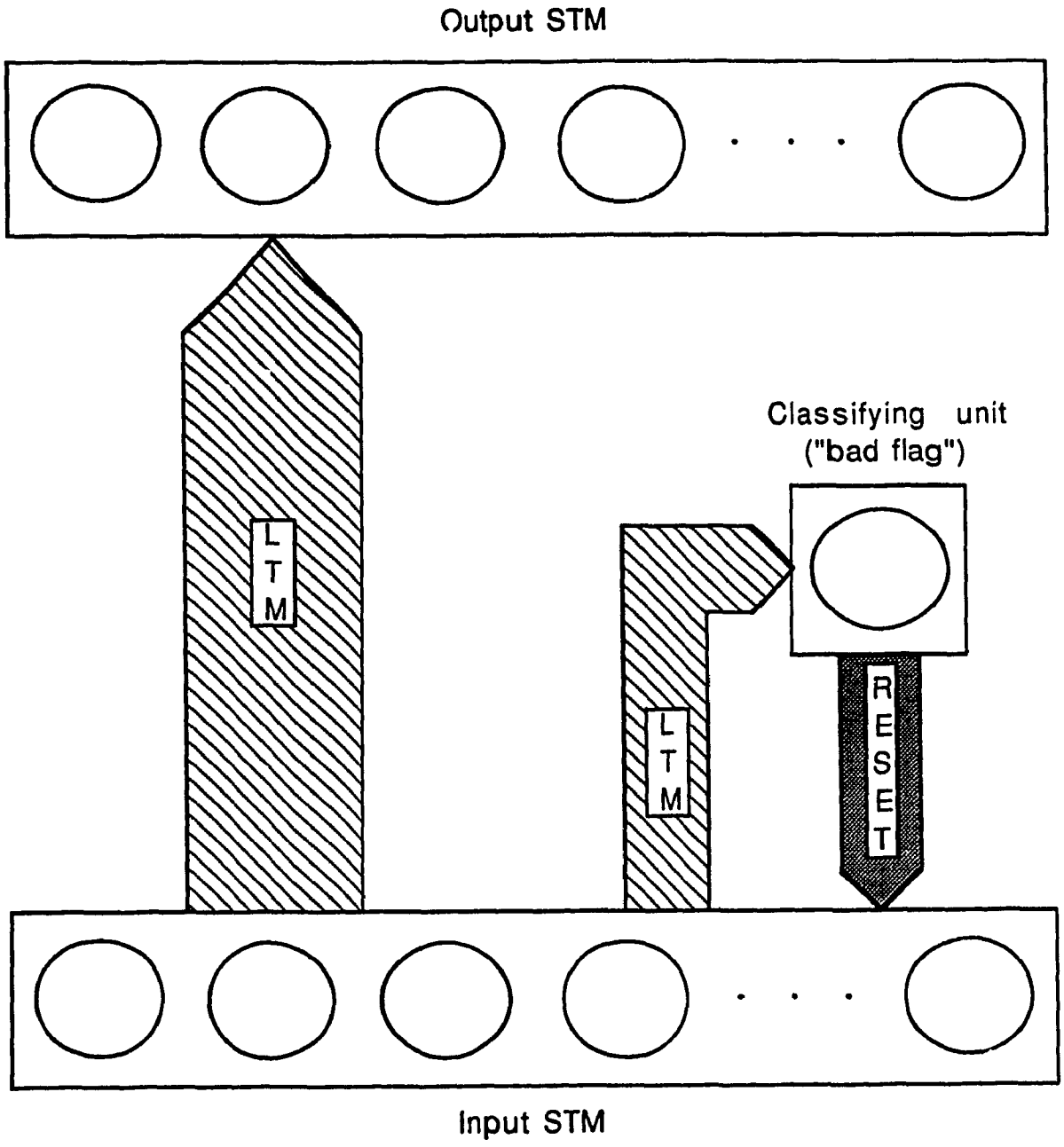

Figure 3.1: Network for learning an arbitrary inference rule (external control signals omitted)

reduce their activation below their threshold whenever the bad flag is active.

The inhibitory connections emanating from the bad flag are not LTM connections; *i.e.*, their strengths do not change with time or "experience." But the connections from the input units to the bad flag *are* LTM connections, so the bad flag can learn how to classify properly.

3.2 The Vector Representation of Propositions

For representing propositions, we take an approach that is closely related to Pao's: We augment the representation of propositions by adding strictly redundant information that is functionally dependent on the original information. However, we are not subject to the restriction that binds Pao. His functional link network architecture is intended to be extremely general in its applicability to pattern recognition problems, but we are concerned only with inference rules. Consequently, we do not need to respect the requirement that the extending functions be mutually orthogonal. We need only show that the extensions will be appropriate for our particular purpose.

Pao uses the outer product expansion for binary-valued data or the Fourier expansion for continuous data. We choose to use only the absolute difference function, since this function provides a convenient measure of "sameness" for components of the basic vector:

$$|x - y| = 0 \Leftrightarrow x = y$$

In this scheme, a basic vector is extended by adding the absolute difference of each pair of basic vector elements.

Our scheme differs from Pao's in another important respect: Pao's scheme makes explicit use of preprocessing to generate a functional extension for each basic vector presented to the network. We take a different approach. We assume that the extended vector *is* the only data structure presented to the network. When the network learns, we expect it to learn how to deduce a proposition's extension as well as its basic vector. This leaves the preprocessing task only to the input system.

Note that the distinction between "basic" and "extended" parts of a vector is not essential to the internal functioning of the system, but is useful in communication between the network and the user, as well as in the exposition of the representation scheme. The network need not at any point explicitly extend a "basic" vector by computing the absolute differences of pairs of its elements. On the contrary, the basic and extended parts are treated in precisely the same way.

As an aside, it might be noted that the basic part of the vector could, in principle, be derived from the extension. The entire network could function quite correctly using the extension alone. However, it is convenient, and not too costly, to use the basic vector for easier communication with the user.

3.2.1 The basic vector

Propositions are represented in a prefix form, using the propositional connectives $\wedge$, $\vee$, and $\supset$. Negation is represented internally in terms of implication:

$$\neg P \stackrel{\text{def}}{=} P \supset \bot.$$

The size of the basic vector used to represent a proposition is determined by the maximum permitted depth of nesting of propositional connectives, which we will call the "complexity":

$$\text{basic vector size} = 2^{c+1} - 1 \tag{3.1}$$

where c is the maximum complexity. We will generally work with $c = 3$, which is sufficient to permit expression of the premise of the Destructive Dilemma rule—the most complex rule we consider. Hence, our basic vector size is generally 15.

To represent atomic propositions, we use the ASCII codes of uppercase letters A (65) to Z (90). (This choice is for computational convenience only, and does not represent any fundamental limitation to a small range of atoms.) The other symbols are given the following arbitrary encodings:

$$\begin{aligned} \bot &= 0 \\ \supset &= 10 \\ \wedge &= 20 \\ \vee &= 30 \end{aligned}$$

An atomic proposition is represented by filling all vector elements with its code. For example, the representation of Z (ASCII code 90) in a vector of size 15 is

[90,90,90,90,90,90,90,90,90,90,90,90,90,90,90].

For notational convenience in our discussion, we will usually write the symbol itself in place of its code, so we write the above vector as

[Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z].

Compound propositions are represented as follows: (1) the first vector element is given the code of the connective; (2) the first operand is represented in the first half of the remaining vector; and (3) the second operand is represented in the last half. For example:

$$A \wedge B$$

becomes

$$[\wedge,A,A,A,A,A,A,A,B,B,B,B,B,B,B],$$

$$\neg A \ (\stackrel{\text{def}}{=} A \supset \bot)$$

becomes

$$[\supset,A,A,A,A,A,A,A,\bot,\bot,\bot,\bot,\bot,\bot,\bot],$$

and

$$(\neg C \vee \neg D) \wedge ((A \supset C) \wedge (B \supset D))$$

becomes

$$[\wedge,\vee,\supset,C,\bot,\supset,D,\bot,\wedge,\supset,A,C,\supset,B,D].$$

3.2.2 The extended vector

The extended vector is a function of the basic vector (and conversely, but we ignore this fact). Its elements are

1. all elements of the basic vector,
2. the logical connectives ($\supset$, $\wedge$ and $\vee$), and
3. the absolute differences of pairs from 1 and 2.

It is computed by the following algorithm:

```
ALGORITHM extend
INPUT a basic vector b of size sb
OUTPUT the extended vector e of b
BEGIN extend
        FOR i := 1 TO sb DO
                e[i] := b[i]
        e[sb + 1] := ⊃
        e[sb + 2] := ∧
        e[sb + 3] := ∨
        k := sb + 4
        FOR i := 2 TO sb + 3 DO
                FOR j := 1 TO i DO
                        e[k] := |e[i] − e[j]|
                        k := k + 1
END extend
```

Thus, for each element of the basic vector, the extension contains elements "comparing" it with each other element of the basic vector, as well as with the three connectives. For convenience, when we show an extended vector, we will show basic vector and the three connectives on the first line, and we will break the remaining elements of the extension into rows of different length, so that it will be possible to tell at a glance which pair of elements were compared to produce a given element. For example, for the proposition A $\supset$ B, the basic vector of size 3 is

[$\supset$,A,B],

and the extended vector is

[$\supset$, A, B, $\supset$, $\wedge$, $\vee$,
55,
56, 1,
0, 55, 56,
10, 45, 46, 10,
20, 35, 36, 20, 10].

The size of the extended vector is $(n^2 + 7n + 12)/2$, where n is the size of the basic vector.

3.3 Network Training

We use the following notation:

w_{ij} : The weight of the connection from input unit i to output unit j.

w_{ib} : The weight of the connection from input unit j to the bad flag.

o_i : The output signal of unit i.

o_b : The output signal of the bad flag.

η : The learning rate — a parameter chosen by the user.

The activities of the input units are set by the user by presenting a propositional vector. The activity o_i is deemed to be equal to the value of the i^{th} element of the input vector, and ranges between 0 and 90.

The output units, as well as the bad flag, are forced to the desired values during training time, but otherwise their activities are determined by the activation rule

$$o_j = \begin{cases} \sum_i o_i w_{ij} & \text{if } \sum_i o_i w_{ij} < 90 \\ 90 & \text{otherwise} \end{cases} \tag{3.2}$$

where 90 (= Z) is (arbitrarily) the maximum output.

All weights w_{ij} are initialized to 1 before learning takes place. Learning occurs according to the rule

$$\Delta w_{ij} = -\eta w_{ij} |o_i - o_j| \tag{3.3}$$

That is, connections between input and output units are initially very strong, but are progressively weakened by "unharmonious" activity. Note that this rule differs significantly from both the "delta rule" (Equation 2.1) and the "generalized delta rule" (Equation 2.2). In our view, this difference is justifiable for several reasons:

1. It is entirely local, depending only on the current weight w_{ij}, and the current outputs of the connected units i and j.

2. It does not permit weights to vary in sign.

3. It constrains weights to take a values from a limited range.

4. It does not require comparison of output against a "target" value.

5. It can remain in effect at all times; not only during an formal learning period.

6. It works.

The learning strategy we use is to present *correct* premise-conclusion pairs at random to the input and output arrays, while forcing the output of the appropriate classifying unit to zero. If a sufficiently large number of random pairs are presented, most weights will fall to zero, leaving non-zero weights only between units whose activities are always equal when the inference rule is used correctly. In particular, the classifying unit will receive non-zero weights only from units that always have zero activity.

When this process is completed, weights will generally be either 1 or 0. This, in itself, is not sufficient to guarantee correct performance, since there will usually be cases in which several input units have the same activity as the target output unit. Given the activation rule above, the inputs will be summed, resulting in an excessive output. This problem can be solved by "practice," however: By allowing the learning rule to remain in effect at all times, even after training, we ensure that the weights will soon fall off until $o_i = o_j$ whenever $w_{ij} \neq 0$.

3.3.1 The "lecture" learning phase

Although there is only a single learning rule, the training regimen is divided into two distinct phases, which we will call "lecture" and "lab". In the "lecture" phase, an input vector is presented to the input array, and the *correct* output vector is presented to the output array. At the same time, the "bad flag" is held at zero, so that inhibitory learning will take place at any link where the input and output units have different activities. This learning obeys the usual rule given in Equation 3.3, above. Observe that Δw_{ij} will be zero if, and only if, $o_i = o_j$. After a sufficiently

large number of training presentations, clearly all the w_{ij} will be reduced to zero, with the exception of those cases where $o_i = o_j$ in *every* presentation; in these cases, w_{ij} retains its original value.

Since the "bad flag" unit is qualitatively identical to the output units, links from the input array to the bad flag learn in the same way. Since o_b is forced to zero by control stimuli, however, the learning rule in this case reduces to:

$$\Delta w_{ib} = -\eta w_{ib} o_i. \tag{3.4}$$

Here, the Δw_{ib} will be zero if, and only if, $o_i = 0$. So after training, the w_{ib} will be zero if the o_i have taken nonzero values during training, and will retain their original values whenever the o_i have had zero values throughout training. The latter cases are of special interest, since they effectively identify the cases where the input vector represents a premise that is suitable for the inference rule being learned.

Consider, for example, the *Modus Ponens* inference rule as it applies to premises with complexity ≤ 3. Here are some examples:

$$(\mathrm{P} \supset \mathrm{Q}) \wedge \mathrm{P}$$
$$(\mathrm{A} \supset \mathrm{Z}) \wedge \mathrm{A}$$
$$((\mathrm{A} \vee \mathrm{B}) \supset \neg \mathrm{C}) \wedge (\mathrm{A} \vee \mathrm{B})$$

The basic vectors representing these examples are, respectively:

(∧, ⊃, P, P, P, Q, Q, Q, P, P, P, P, P, P, P)
(∧, ⊃, A, A, A, Z, Z, Z, A, A, A, A, A, A, A)
(∧, ⊃, ∨, A, B, ⊃, C, ⊥, ∨, A, A, A, B, B, B)

These generate the extended vectors:

[∧, ⊃, P, P, P, Q, Q, Q, P, P, P, P, P, P, P, ⊃, ∧, ∨,
10,
60, 70,
60, 70, 0,
60, 70, 0, 0,
61, 71, 1, 1, 1,
61, 71, 1, 1, 1, 0,
61, 71, 1, 1, 1, 0, 0,
60, 70, 0, 0, 0, 1, 1, 1,
60, 70, 0, 0, 0, 1, 1, 1, 0,
60, 70, 0, 0, 0, 1, 1, 1, 0, 0,
60, 70, 0, 0, 0, 1, 1, 1, 0, 0, 0,
60, 70, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
60, 70, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
60, 70, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0,
10, 0, 70, 70, 70, 71, 71, 71, 70, 70, 70, 70, 70, 70, 70,
0, 10, 60, 60, 60, 61, 61, 61, 60, 60, 60, 60, 60, 60, 60, 10,
10, 20, 50, 50, 50, 51, 51, 51, 50, 50, 50, 50, 50, 50, 50, 20, 10]

[∧, ⊃, A, A, A, Z, Z, Z, A, A, A, A, A, A, A, ⊃, ∧, ∨,
10,
45, 55,
45, 55, 0,
45, 55, 0, 0,
70, 80, 25, 25, 25,
70, 80, 25, 25, 25, 0,
70, 80, 25, 25, 25, 0, 0,
45, 55, 0, 0, 0, 25, 25, 25,
45, 55, 0, 0, 0, 25, 25, 25, 0,
45, 55, 0, 0, 0, 25, 25, 25, 0, 0,
45, 55, 0, 0, 0, 25, 25, 25, 0, 0, 0,
45, 55, 0, 0, 0, 25, 25, 25, 0, 0, 0, 0,
45, 55, 0, 0, 0, 25, 25, 25, 0, 0, 0, 0, 0,
45, 55, 0, 0, 0, 25, 25, 25, 0, 0, 0, 0, 0, 0,
10, 0, 55, 55, 55, 80, 80, 80, 55, 55, 55, 55, 55, 55, 55,
0, 10, 45, 45, 45, 70, 70, 70, 45, 45, 45, 45, 45, 45, 45, 10,
10, 20, 35, 35, 35, 60, 60, 60, 35, 35, 35, 35, 35, 35, 35, 20, 10]

[∧, ⊃, ∨, A, B, ⊃, C, ⊥, ∨, ∧, A, A, B, B, B, ⊃, ∧, ∨,
10,
10, 20,
45, 55, 35,
46, 56, 36, 1,
10, 0, 20, 55, 56,
47, 57, 37, 2, 1, 57,
20, 10, 30, 65, 66, 10, 67,
10, 20, 0, 35, 36, 20, 37, 30,
45, 55, 35, 0, 1, 55, 2, 65, 35,
45, 55, 35, 0, 1, 55, 2, 65, 35, 0,
45, 55, 35, 0, 1, 55, 2, 65, 35, 0, 0,
46, 56, 36, 1, 0, 56, 1, 66, 36, 1, 1, 1,
46, 56, 36, 1, 0, 56, 1, 66, 36, 1, 1, 1, 0,
46, 56, 36, 1, 0, 56, 1, 66, 36, 1, 1, 1, 0, 0,
10, 0, 20, 55, 56, 0, 57, 10, 20, 55, 55, 55, 56, 56, 56,
0, 10, 10, 45, 46, 10, 47, 20, 10, 45, 45, 45, 46, 46, 46, 10,
10, 20, 0, 35, 36, 20, 37, 30, 0, 35, 35, 35, 36, 36, 36, 20, 10]

The boldface zeros mark the positions in the input vectors where zeros *must* occur in any Modus Ponens premise by virtue of the rules for representation of compound propositions given on page 22. Recall now the assertion of Equation 3.4: that the Δw_{ib} vary with the o_i. Since all weights are initialized to 1 before training, this means that in these positions we will always have $w_{ib} = 1$, as long as the network is trained only for Modus Ponens. If the output threshold of the bad flag unit is sufficiently low, then, the bad flag unit will fire a reset wave each time the a non-Modus Ponens proposition is presented at the input array.

3.3.2 The "lab" learning phase

The lab learning phase involves no teaching — only practice on correct examples. That is, correct premises for the rule being learned are presented at the input array, and no external control is applied to the output array or the bad flag. The only influences on the output and bad flag units, therefore, are the STM signals from the input array, filtered by the LTM that has just finished the lecture learning phase.

The LTM weights, as we observed in the last section, will have been either reduced to zero or left undisturbed in their initialized state (*i.e.*, = 1). The zero weights are of no concern to us now, as they have already acquired their desired state, but many of the other weights are still too strong. Specifically, whenever an inference rule requires that an atom (or connective) occur at least once in the conclusion — say at position j — and at least twice in the premise — say at positions $i_1, \ldots, i_n$, the weights between the corresponding units will be excessive. The activation rule of Equation 3.2 reduces to

$$o_j = \begin{cases} \sum_{k=1}^{n} o_{i_k} w_{i_k j} & \text{if } \sum_{k=1}^{n} o_{i_k} w_{i_k j} < 90 \\ 90 & \text{otherwise} \end{cases} \tag{3.5}$$

and, since all the $w_{i_k j} = 1$ and all the o_{i_k} are equal, this is further reducible to

$$o_j = \begin{cases} n o_{i_1} & \text{if } n o_{i_1} < 90 \\ 90 & \text{otherwise} \end{cases} \tag{3.6}$$

This is incompatible in general with the requirement of the hypothesis that after learning, $o_j = o_{i_k}$ for each k.

The solution is simple, however. We just allow the activation rule and the learning rule to act in tandem. The result: after a sufficiently large number of iterations, each of the n weights $w_{i_k j}$ decays to $1/n$.

3.4 Unlearnable Rules

When the bad flag does not fire, the trained network infers a conclusion from its premise by *copying* components of the premise vector to the appropriate location in the conclusion vector. This points to a limitation of the method: If the conclusion

of an inference rule has content that is absent in the premise, then the rule cannot be learned. What rules does this exclude? Surprisingly few. One might think, for example, that DeMorgan's law

$$\frac{\neg(P \vee Q)}{\neg P \wedge \neg Q}$$

would be unlearnable because the conclusion contains the symbol '$\wedge$,' which is absent in the premise. However, it is the *extended* vector, not the basic vector, that we must examine. Recall that the extended vector always contains all of the logical connectives.

Only rules that introduce new non-connective elements are unlearnable. Among the usual propositional rules, only *addition* and *conjunction*

$$\frac{P}{P \vee Q}(\text{Add.}) \qquad \frac{P \quad Q}{P \wedge Q}(\text{Conj.})$$

fail on this account. Addition fails for the obvious reason that Q is missing. Conjunction fails because, although P and Q are both present in the premise, their absolute difference $|P - Q|$ is not.

The inability of the architecture to learn addition is perhaps not a serious drawback, but conjunction is a problem that cannot be swept aside so easily. In principle, we could resolve it by changing the architecture so as to use only *binary* threshold units, with each atom being represented by a binary code spread over six units. This would leave only '1's and '0's to be copied, and we are already assured that some of each will occur in every extended vector. However, this would give an extended vector size of 4371 units — an unacceptably large number in practice. It make better sense to treat conjunction in our simulations as a special case of a hard-wired rule.

3.5 The Simulator

To test the ideas discussed above, we have developed a program that creates an array of networks of the kind just described, trains them to perform any of several inference rules, and then uses the array to do proofs in a bottom-up fashion. The program effectively simulates the architecture shown in Figure 3.2, but to save both space and time, it actually simulates the equivalent architecture of Figure 3.3. The difference is that the purely local reset mechanism has been replaced by global control: When the bad flag is activated for a given rule, rather than use the computationally expensive reset to prevent a conclusion from being drawn, the simulator allows the conclusion to be inferred, but does not copy it to memory.

The memory store contains propositions represented in their fully extended form, so it is not necessary to extend them explicitly (except in the conjunction unit). This resolves the "cheating" problem that we pointed out with respect to Pao's functional link net; the networks themselves take care of the extension as part of the inference process.

The code for the simulator, in "C," is reproduced in Appendix A, and some sample proofs are shown in Appendix B.

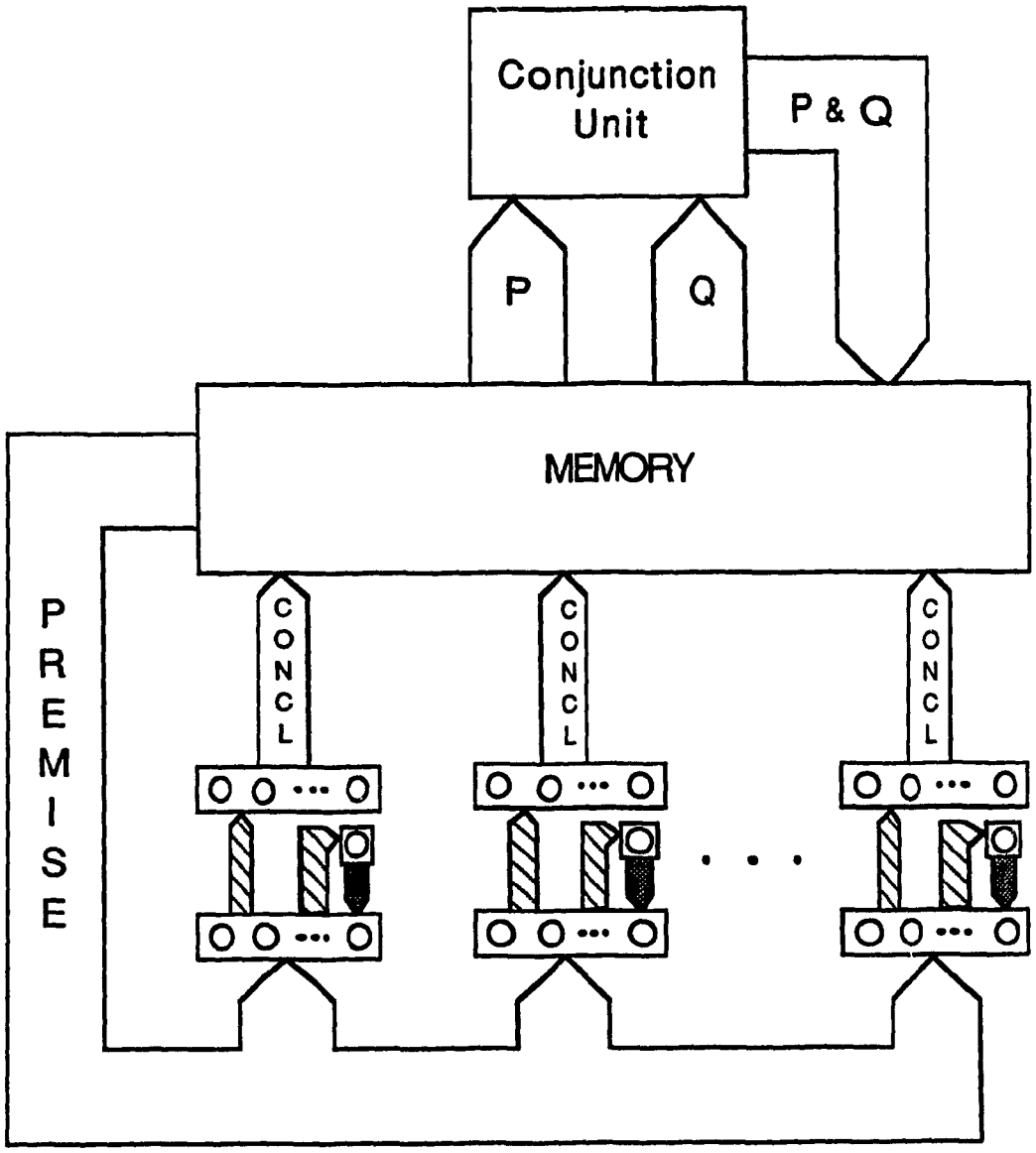

Figure 3.2: The simulated architecture with local "reset"

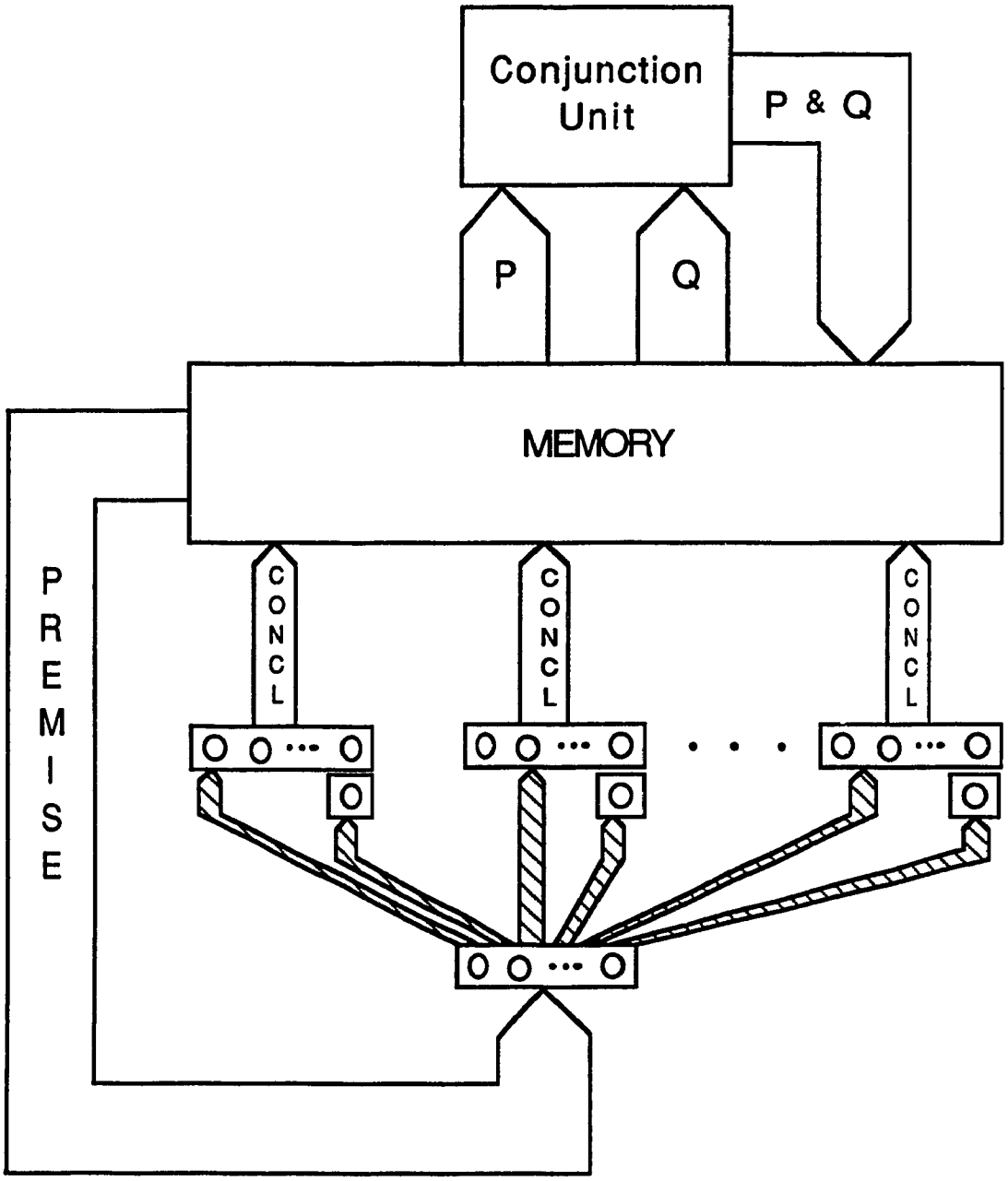

Figure 3.3: The simulated architecture with global control instead of "reset"

Chapter 4

Conclusion and Discussion

We have shown how it is possible, for a flat neural network to learn most inference rules of propositional logic, the exceptions being *Addition* and *Conjunction.*

Propositions are represented as vectors of integers, extended by adding logical connectives, and the absolute differences of the basic vector components. The network itself, for a single rule, consists of two arrays of linear threshold units with the usual linear activation rule. It uses a local learning rule which we have proposed especially for this purpose.

Simulations show that the network is indeed capable of learning and performing interesting inferences.

Several interesting issues remain to be addressed. First, it would be a valuable step to find a way around the disappointing exceptions. This may require some alterations to the particular representation we have used, or to the learning rule, but it is at least an open question whether the problem is soluble. Secondly, it would be interesting to know whether the same technique (or a similar one) can be extended to deal with first-order logic with individual variables and quantifiers.

Finally, since the network does not depend on any semantic aspects of propositional logic, it can be viewed more generally as a machine for learning "pattern

inference." We may well ask whether there are applications other than propositional logic to which this technique could be applied.

4.1 Another Possible Approach

The failure of our architecture to learn the *Conjunction* rule is, as we have mentioned, a consequence of our choice of learning rule (Equation 3.3) and activation rule (Equation 3.2), which together ensure that inference consists in the copying of appropriate vector elements from premise to conclusion. This turns out to be quite adequate in most cases, effectively blocks successful learning whenever the conclusion vector contains new elements.

Can this obstacle be overcome by using different rules? In view of the fact that our representation of propositions contains a great deal of information about pairwise *joint* activities of units, it is reasonable to enquire whether the delta rule, or perhaps the generalized delta rule, might be capable of producing better results. We have taken some preliminary steps towards answering this question, which we describe here briefly.

In our first experiment, we simulated a flat two-layer network using our activation rule (Equation 3.2) and the simple delta rule (Equation 2.1) for learning. The network was trained for a given rule by presenting correct premise/conclusion pairs at the input and output layers. No attempt was made to train the network with incorrect pairs; nevertheless, the network was unable to learn any rule satisfactorily; *i.e.*, in every case, there was at least one element of the conclusion vector whose value was incorrect. Essentially the same result was obtained when the generalized delta rule (Equation 2.1) was substituted for the delta rule.

We interpret these results as indicating that the delta rules are inappropriate for use in networks where the output of an individual network element is given a nonbinary interpretation. Unfortunately, a binary interpretation would require us to use 4371 units to represent a proposition of complexity 3 with its full functional extension, so this avenue would seem closed.

Our second experiment, however, presents an interesting prospect. In this case, the idea of functional extension was abandoned in favor of a simple binary representation of the basic vector: The basic vector was formed as described in Section 3.2.1, above, but each element was further encoded into five binary units, giving a 75-element vector for a proposition of complexity 3. Again, a two-layer flat network was used, but this time with the generalized delta rule and a sigmoid activation function. This network was trained with correct premise/conclusion vector pairs, intermingled in equal proportions with random/zero vector pairs to see whether the network could perform the dual functions of recognition and transformation of vectors.

The results are summarized in Table 4.1. The error frequencies seem to indicate that the flat delta rule network can learn effectively when the rule being learned requires the transcription of only a portion of the premise to the conclusion. In particular, the *Repetition* and *Conjunction* rules, which require complete copying of the input vector, have very high error rates compared with *Double Negation Elimination* and *Destructive Dilemma*, which transcribe only relatively small portions.

Some remarks are in order: First, the apparently small error rates are misleading in that they reflect only the network's ability to discriminate between correct premises and random bit vectors. A more complete study, which is beyond our

Inference Rule	Proposition Errors (%)
Repetition	6.84
Conjunction	6.40
Double Negation Introduction	5.49
Implication Elimination	1.60
OR-Commutativity	0.51
AND-Commutativity	0.50
Modus Ponens	0.14
Constructive Dilemma	0.05
Hypothetical Syllogism	0.03
Implication Introduction	0.02
Simplification	0.002
Modus Tollens	0
Disjunctive Syllogism	0
Contraposition	0
AND-OR DeMorgan's Law	0
OR-AND DeMorgan's Law	0
Double Negation Elimination	0
Destructive Dilemma	0

Table 4.1: Error frequencies for a flat network

present scope, would need to compare correct premises with incorrect premises that are legitimate propositions. Second, an error rate of 1% is too high for most practical purposes, especially if a chain of inferences is required. It should also be noted that the performance of the network for *Conjunction* is less than spectacular, in view of the fact that we first turned to it in the hope of learning that rule. Nevertheless, these results suggest that it might be possible, with some further work, to train delta rule networks to perform probable inference in contexts where some error in results can be tolerated. Other preliminary trials indicate that the introduction of a small random error (one inverted bit in every hundredth premise) in the *correct* premises does not significantly impair the capacity of the network to learn, so networks of this kind might be useful where the training set is imperfect.

This approach is in a sense, complementary to the one which forms the basis of

avoid catastrophic failure. The other is slower and subject to residual error, but has at least some robustness in the presence of flawed training data. It is not impossible that both should find applications.

Bibliography

[1] Ballard, D, H. and Hayes, P. J., "Parallel Logical Inference," *Proceedings of the Sixth Annual Conference of the Cognitive Science Society*, Boulder, Co., June, 1988.

[2] Carpenter, G. and Grossberg, S., "A massively parallel architecture for a self-organizing neural pattern recognition machine," *Computer Vision, Graphics, and Image Processing*, **37**, pp. 54-115, 1987.

[3] Carpenter, G. and Grossberg, S., "ART2: Self-organization of stable category recognition codes for analog input patterns," *Applied Optics*, **26**, pp. 4919-4930, 1987.

[4] Copi, I. M., *Introduction to Logic.* New York: Macmillan Publishing, 3rd ed., 1978.

[5] Grossberg, S. "Competitive Learning: From Interactive Activation to Adaptive Resonance," *Cognitive Science*, **11**, pp. 23-63, 1987.

[6] D. O. Hebb. *The Organization of Behaviour.* New York: Wiley, 1949.

[7] Johnson-Laird, P. N. *Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness.* Cambridge, England: Cambridge University Press, 1983.

[8] Johnson-Laird, P. N., Byrne, R. M. J. and Tabossi, P. "Reasoning by Model: The Case of Multiple Quantification," *Psychological Review*, **96**, No. 4, pp. 658-673, 1989.

[9] Klassen, M. S. and Pao, Y. H., "Characteristics of the functional link net: A higher order delta rule net," in *IEEE Proceedings of 2nd Annual International Conference on Neural Networks*, 1988.

[10] Kohonen, T., *Self-Organization and Associative Memory.* Heidelberg: Springer-Verlag, 1984, 3rd ed., 1989.

[11] Kosko, B., "Adaptive bidirectional associative memories," *Applied Optics*, **26**, No. 23, pp. 4947-4960, 1987.

[12] Kurfess, F. and Reich, M., "Logic and Reasoning with Neural Models," in Pfeiffer, R., Schreter, Z., and Fogelman-Soulié, F., Eds., *Connectionism in Perspective.* North-Holland: Elsevier Science Publishers, pp. 365-376, 1989.

[13] McCulloch, W. S. and Pitts, W. H., "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, 7, pp. 89-93, 1913.

[14] Minsky, M. and Papert, S., *Perceptrons: An Introduction to Computational Geometry.* Cambridge, MIT Press, 1969. (Expanded edition, 1988).

[15] Pao, Y.-H., *Adaptive Pattern Recognition and Neural Networks.* Reading, Mass.: Addison-Wesley, 1989.

[16] Rosenblatt, F., *Principles of Neurodynamics.* New York: Spartan, 1962.

[17] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Learning Internal Representations by Error Propagation," in Rumelhart, D. E. and McClelland, J. L., Eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Cambridge, Mass.: MIT Press, pp. 318-362, 1986.

[18] Rumelhart, D. E. and McClelland, J. L., Eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Cambridge, Mass.: MIT Press, 1986.

[19] Schmidhuber, J., "The Neural Bucket Brigade," in Pfeiffer, R., Schreter, Z., and Fogelman-Soulié, F., Eds., *Connectionism in Perspective.* North-Holland: Elsevier Science Publishers, pp. 429-437, 1989.

[20] Sobajic, D., *Neural Nets for Control of Power Systems*, Ph.D. Thesis, Computer Science Dept., Case Western Reserve University, Cleveland, Ohio, 1988.

[21] Touretzky, D. S. and Hinton G. E. "Symbols among the neurons: Details of a connectionist inference architecture," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 238-243, 1985.

[22] Williams, R. J., "The Logic of Activation Functions," in Rumelhart, D. E. and McClelland, J. L., Eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Cambridge, Mass.: MIT Press, pp. 423-443, 1986.

Appendix A

The Simulator Code

```
/*****************************************************************/
/******************                               ******************/
/******************        neurnets.h             ******************/
/******************                               ******************/
/*****************************************************************/

#include  <stdio.h>
#include  <ctype.h>
#include  <strings.h>
#include  <memory.h>

/*==================== CONSTANTS AND MACROS ======================*/

#ifdef __TURBOC__
#include  <stdlib.h>
#define RANDOM_INT(x)       ( (random(32767)>>8) % (x) )
#define SEED_RANDOM         ( randomize() )
#else
#define RANDOM_INT(x)       ( (random()>>16) % (x) )
#define SEED_RANDOM         ( srandom(getpid()) )
#endif

#define MAX_COMPLEXITY      2 /* should be 2 or 3 */
#define PROP_SZ             ( (1 << (MAX_COMPLEXITY+1)) - 1 )
#define EXT_SZ              ( (((PROP_SZ+3)*(PROP_SZ+2)) >> 1)+3 )
#define VEC_SZ              ( PROP_SZ + EXT_SZ )
#define WT_SCALE            17
#define ETA                 8
#define THRESHOLD           1
#define MAX_ACT             90
#define SQR(x)              ( (x) * (x) )
#define MAX(x,y)            ( ((x)>(y)) ? (x) : (y) )
#define ABSDIF(x,y)         ( ((x)>(y)) ? ((x)-(y)) : ((y)-(x)) )
#define MAX_RULES           20
```

```
#define ERROR              -1
#define FAILURE            0
#define SUCCESS            1
#define TRUE               1
#define FALSE              0
#define LPAR               1
#define RPAR               2
#define BOTTOM             0
#define CONTRADICTION      40
#define NEG                0
#define IMPL               10
#define CONJ               20
#define DISJ               30
#define NUM_CONNECTIVES    4
#define MAX_CONNECTIVE     DISJ
#define ATOM(x)            ( !(((x) < 'A') || ((x) > 'Z')) )
#define IS_CONNECTIVE(x)   ( !(ATOM(x) || ((x)%10)) )
#define RANDOM_CONNECTIVE  ( RANDOM_INT(NUM_CONNECTIVES) * 10 )
#define RANDOM_ATOM        ( RANDOM_INT(26) + 'A' )
#define FIFTY_FIFTY        ( RANDOM_INT(2) )
#define CLEAR_SCREEN       printf( "%c", (char) 12 )

/*====================== TYPE DEFINITIONS ======================*/

typedef    struct rule_struct
      {
        int   (*premise)(), (*conclusion)();
        char  *name;
      } RULE;


typedef   int  PROPOSITION[VEC_SZ];
#define   SAME_PROP(p1,p2)   !memcmp( (char *)(p1), \
                              (char *)(p2), sizeof(PROPOSITION) )


typedef  struct prop_list
      {
        PROPOSITION       prop;
        struct prop_list  *next;
      } *PROP_LIST;

/*=================== EXTERNAL DECLARATIONS ===================*/

/* defined in rules.c */
extern  void  extend();
extern  void  conjoin();
extern  int   select_rules();
```

```
extern  int   read_net();
extern  RULE  *get_rule();

/* defined in parse.c */
extern  int  str_prop();
extern  int  prop_str();

/* defined in memarch.c */
extern  void  flush_memory();
extern  int   get_prop();
extern  void  next_prop();
extern  void  memorize_premises();
extern  int   memorize_inference();
extern  int   print_proof();
extern  void  print_memory();

/* defined in utility.c */
extern  int   yes();
extern  int   get_option();
extern  void  pause();
extern  int   all_white();
extern  char  *all_scan();
```

```
/*****************************************************************/
/*****************                                  *****************/
/*****************            sim.c                 *****************/
/*****************                                  *****************/
/*****************************************************************/

#include    "neurnets.h"

\begin{verbatim}
/*-------------------- global variables ----------------------*/

PROPOSITION    in, out[MAX_RULES];   /* input & output vectors */
int     bad_flag[MAX_RULES];         /* one "bad_flag per rule */
int     w[MAX_RULES][VEC_SZ][VEC_SZ];/* weights from in to out */
int     bf_w[MAX_RULES][VEC_SZ];/* weights from in to bad_flag */

RULE    rules[MAX_RULES];           /* array of rules to learn */
int     rule_count;                /* number of rules to learn */

/*------------------------- init_wts ---------------------------*/
/* Set all weights for rule to 1 (times a scaling factor).     */
/*-----------------------------------------------------------*/

void   init_wts( rule )
int    rule;
{
    int i, j;

    for ( i=0 ; i<VEC_SZ ; i++ ) {
        bf_w[rule][i] = 1<<WT_SCALE;
        for ( j=0 ; j<VEC_SZ ; j++ )
            w[rule][i][j] = 1<<WT_SCALE;
    }
}

/*--------------------------- learn ---------------------------*/
/* Adjust the weights between "in" and "out[rule]" and its     */
/* bad_flag according to the standard learning rule.           */
/*-----------------------------------------------------------*/

void learn( rule )
int  rule;
{
    int i, j;

    for ( i=0 ; i<VEC_SZ ; i++ ) {
        bf_w[rule][i] -= (bf_w[rule][i] * in[i]) >> ETA;
        for ( j=0 ; j<VEC_SZ ; j++ )
```

```
            w[rule][i][j] -= (w[rule][i][j] *
                              ABSDIF(in[i],out[rule][j]))
                             >> ETA;
    }
}

/*------------------------ update ----------------------------*/
/* Compute new activation values for out[rule] and its       */
/* bad_flag according to the standard activation rule.  If the */
/* bad_flag is activated above threshold, suppress learning.  */
/*-----------------------------------------------------------*/

void    update( rule )
int    rule;
{
    int    i, j, sum, bf_sum;

    bf_sum = 0;
    for ( i=0 ; i<VEC_SZ ; i++ ) {
        bf_sum += in[i]*bf_w[rule][i];
        sum = 0;
        for ( j=0 ; j<VEC_SZ ; j++ )
            sum += in[j] * w[rule][j][i];
        out[rule][i] = sum >> WT_SCALE;
        if ( out[rule][i] > MAX_ACT )
            out[rule][i] = MAX_ACT;
    }
    bad_flag[rule] = bf_sum >> WT_SCALE;
    if ( bad_flag[rule] > MAX_ACT )
        bad_flag[rule] = MAX_ACT;
    if ( bad_flag[rule] < THRESHOLD )
        learn(rule);
}

/*------------------------ train ----------------------------*/
/* For each rule being learned, present premise/conclusion   */
/* pairs at random for "lecture" learning, then random       */
/* premises alone for "lab' learning.                        */
/*-----------------------------------------------------------*/

void    train()
{
    int    i, rule, cycles;
    char    str[50];
    PROPOSITION temp;

    printf( "Training cycles: " );
    fgets( str, 50, stdin );
```

```
    sscanf( str, "%d", &cycles );
    for ( rule=0 ; rule<rule_count ; rule++ ) {
        printf( "\n%s\n", rules[rule].name );
        printf( "\tLecture...\n" );
        bad_flag[rule] = 0;
        for ( i=0 ; i++<cycles ;  ) {
            (*(rules[rule].premise))(in);
            (*(rules[rule].conclusion))(in, out[rule]);
            learn(rule);
            if ( !(i%100) ) printf("\t\t%d cycles\n", i);
        }
        printf( "\tLab...\n" );
        for ( i=0 ; i++<cycles ;  ) {
            (*(rules[rule].premise))(in);
            update(rule);
            if ( !(i%100) ) printf("\t\t%d cycles\n", i);
        }
    }
}

/*------------------------ test -----------------------------*/
/* Activate all rules and print conclusions for those rules    */
/* whose bad_flags remain below threshold.  Called either by   */
/* auto_test() or by manual_test().                            */
/*-------------------------------------------------------------*/

void    test()
{
    int    i, j, r;
    char   str[50];

    for ( r=0 ; r<rule_count ; r++ ) {
        update(r);
        if ( bad_flag[r] < THRESHOLD ) {
            if ( prop_str(out[r], str) == FAILURE )
                printf( "Unprintable conclusion by %s\n",
                        rules[r].name );
            else
                printf( "Conclusion by %s: %s\n",
                        rules[r].name, str );
        }
    }
}

/*---------------------- auto_test --------------------------*/
/* Get random premise for some active rule, then call test()   */
/*-------------------------------------------------------------*/
```

```
void    auto_test()
{
    char    str[50];

    do {
        (*(rules[RANDOM_INT(rule_count)].premise))(in);
        prop_str( in, str );
        printf( "Premise: %s\n", str );
        test();
    } while ( yes("\nAnother?") );
}

/*---------------------- manual_test -------------------------*/
/* Get a premise from the keyboard and call test()            */
/*------------------------------------------------------------*/

void    manual_test()
{
    int    i;
    char    str[50];

    do {
        printf("Enter a premise: ");
        fgets( str, 50, stdin );
        i = strlen(str);
        if ( str[--i] == '\n' ) str[i] = '\0';
        if ( str_prop(str, in) == FAILURE ) {
            printf( "Cannot parse %s\n", str );
            continue;
        }
        test();
    } while ( yes("\nAnother?") );
}

/*---------------------- show_vectors ------------------------*/
/* Display both the actual output vector and the target vector */
/* for each rule whose bad_flag is below threshold.  Otherwise */
/* display bad_flag activation.                               */
/*------------------------------------------------------------*/

void    show_vectors()
{
    int    i, j, k, rule;
    char    str[50];

    for ( rule=0 ; rule<rule_count ; rule++ ) {
        if ( prop_str(in, str) == FAILURE )
            printf( "\nUnprintable input proposition.\n" );
```

```
        else
            printf( "\nInput proposition: %s\n", str );
        if ( bad_flag[rule] >= THRESHOLD ) {
            printf( "%s: badness = %d\n",
                    rules[rule].name, bad_flag[rule] );
            pause("Press <return> to continue...");
            continue;
        }
        printf("\nOutput vector for %s:\n", rules[rule].name );
        printf("==========================================\n");
        k = 0;
        for ( i=0 ; i<(PROP_SZ+3) ; i++ )
            printf("%6d", out[rule][k++]);
        printf("\n");
        for ( i=0 ; i<(PROP_SZ+3) ; i++ ) {
            for ( j=0 ; j<i ; j++ )
                printf("%6d", out[rule][k++]);
            printf("\n");
        }
        (*(rules[rule].conclusion))(in, out[rule]);
        printf("\nTarget vector for %s:\n", rules[rule].name);
        printf("==========================================\n");
        k = 0;
        for ( i=0 ; i<(PROP_SZ+3) ; i++ )
            printf("%6d", out[rule][k++]);
        printf("\n");
        for ( i=0 ; i<(PROP_SZ+3) ; i++ ) {
            for ( j=0 ; j<i ; j++ )
                printf("%6d", out[rule][k++]);
            printf("\n");
        }
        pause("Press <return> to continue...");
    }
}

/*------------------------ save_net -------------------------*/
/* Save all weights for all active rules to a file.          */
/*-----------------------------------------------------------*/

void    save_net()
{
    FILE    *fp;
    char    filename[50], *p;
    int     i, j, rule;

    if ( yes("Save weights?") ) {
        printf("Enter file name: ");
        fgets( filename, 50, stdin );
```

```
        p = (char *) strchr(filename,'\n');
        if ( p != NULL ) *p = '\0';
        fp = fopen( filename, "w" );
        fprintf( fp, "%d\n", rule_count );
        for ( rule=0 ; rule<rule_count ; rule++ ) {
            fprintf( fp, "%s\n", rules[rule].name );
            for ( i=0 ; i<VEC_SZ ; i++ ) {
                fprintf( fp, "%d\n", bf_w[rule][i] );
                for ( j=0 ; j<VEC_SZ ; j++ )
                    fprintf( fp, "%d\n", w[rule][i][j] );
            }
        }
        fclose( fp );
    }
}

/*---------------------- get_premises_kbd ----------------------*/
/* Get premises from the keyboard, and insert them in          */
/* simulated memory, along with conjunctions.                  */
/*-------------------------------------------------------------*/

void    get_premises_kbd()
{
    PROP_LIST    head, temp;
    char        p_str[50], *end;

    printf("Enter propositions (blank line to exit):\n");
    head = (PROP_LIST) malloc( sizeof(struct prop_list) );
    head->next = NULL;
    while( (fgets(p_str,50,stdin) != NULL)
            && !all_white(p_str) ) {
        end = (char *) strchr(p_str, '\n');
        if ( end != NULL ) *end = '\0';
        end = all_scan(p_str);
        if ( *end != '\0' ) {
            printf( "Cannot scan %s.  Try again.\n", p_str );
            continue;
        }
        if ( str_prop(p_str, head->prop) == FAILURE ) {
            printf( "Cannot parse %s.  Try again.\n", p_str );
            continue;
        }
        temp = head;
        head = (PROP_LIST) malloc( sizeof(struct prop_list) );
        head->next = temp;
    }
    memorize_premises(head->next);
    while ( head != NULL ) {
```

```
        temp = head->next;
        free(head);
        head = temp;
    }
}

/*-------------------------- saturate ------------------------*/
/* Fire all active rules while new premises are available in  */
/* simulated memory queue, adding conclusions to queue.       */
/*------------------------------------------------------------*/

void    saturate()
{
    int    r;

    while ( get_prop(in) ) {
        for ( r=0 ; r<rule_count ; r++ ) {
            update( r );
            if ( bad_flag[r] < THRESHOLD )
                memorize_inference( out[r], rules[r].name );
        }
        next_prop();
    }
}

/*-------------------------- prove ---------------------------*/
/* Look for goal proposition in memory queue while saturating */
/* memory queue.  If found, print proof; otherwise fail.      */
/*------------------------------------------------------------*/

void    prove()
{
    PROPOSITION p;
    char        p_str[50], *end;
    int         r;

    printf("Enter proposition to prove: ");
    fgets(p_str, 50, stdin);
    end = (char *) strchr(p_str, '\n');
    if ( end != NULL ) *end = '\0';
    if ( str_prop(p_str, p) == FAILURE )
        printf( "Parse error.  " );
    else {
        while ( get_prop(in) ) {
            if ( SAME_PROP(p,in) ) {
                print_proof( p );
                return;
            }
```

```
            for ( r=0 ; r<rule_count ; r++ ) {
                update( r );
                if ( bad_flag[r] < THRESHOLD ) {
                    memorize_inference( out[r], rules[r].name );
                    if ( SAME_PROP(p,out[r]) ) {
                        print_proof( p );
                        return;
                    }
                }
            }
            next_prop();
        }
        pause( "Proof not found.  Press <return>" );
    }
}

/*-------------------------- main ---------------------------*/
/* Main menu driver for neural net simulator.                */
/*-----------------------------------------------------------*/

main()
{
    int    rule, old_count, option;

    rule_count = 0; /* default network has no rules */
    SEED_RANDOM;
    printf("\n");

    do {
        CLEAR_SCREEN;
        printf("             Options\n");
        printf("=================================\n");
        printf("1...Create a new network\n");
        printf("2...Read a network from a file\n");
        printf("3...Add rules to current network\n");
        printf("4...Train current network\n");
        printf("5...Display current output vectors\n");
        printf("6...Automatic network test\n");
        printf("7...Manual network test\n");
        printf("8...Enter propositions from keyboard\n");
        printf("9...Prove proposition\n");
        printf("A...Saturate memory\n");
        printf("B...Flush memory\n");
        printf("C...Print memory\n");
        printf("D...Save current network\n");
        printf("Q...Quit\n");
        printf("=================================\n");
        option = get_option("Choose...", "123456789ABCDQ");
```

```
CLEAR_SCREEN;
switch ( option ) {
    case '1' : /* Create a new network */
        rule_count = select_rules( rules );
        for ( rule=0 ; rule<rule_count ; rule++ )
            init_wts(rule);
        break;
    case '2': /* Read a network from a file */
        rule_count = read_net( rules, bf_w, w );
        if ( rule_count == ERROR ) rule_count = 0;
        break;
    case '3' : /* Add rules to current network*/
        old_count = rule_count;
        rule_count = add_rules( rules, rule_count );
        for ( rule=old_count ; rule<rule_count ; rule++ )
            init_wts(rule);
        break;
    case '4' : /* Train current network */
        train();
        break;
    case '5' : /* Display current output vectors */
        show_vectors();
        break;
    case '6' : /* Automatic network test */
        auto_test();
        break;
    case '7' : /* Manual network test */
        manual_test();
        break;
    case '8' : /* Enter propositions from keyboard */
        get_premises_kbd();
        break;
    case '9' : /* Prove proposition */
        prove();
        break;
    case 'A' : /* Saturate memory */
        printf("Saturating memory...\n");
        saturate();
        pause("Memory saturated. Press <return>...");
        break;
    case 'B' : /* Flush memory */
        flush_memory();
        break;
    case 'C' : /* Print memory */
        print_memory();
        break;
    case 'Q' : /* Quit: fall through save option first */
```

```
            case 'D' : /* Save current network */
                save_net();
                break;
        }
    } while ( option != 'Q' );
}
```

```
/*****************************************************************/
/*****************                                  **************/
/*****************          rules.c                 **************/
/*****************                                  **************/
/*****************************************************************/

#include    "neurnets.h"

/*------------------------ make_prop ------------------------*/
/* Create a random propositional vector "vec" of length "size" */
/*-----------------------------------------------------------*/

static void    make_prop( vec, size )
int    vec[];
int    size;
{
    int    i, start, newsize, atom, connective;

    if ( size == 1 )
        vec[0] = RANDOM_ATOM;
    else if ( FIFTY_FIFTY ) {
        atom = RANDOM_ATOM;
        for ( i=0 ; i<size ; i++ )
            vec[i] = atom;
    }
    else {
        connective = RANDOM_CONNECTIVE;
        newsize = size >> 1;
        make_prop( &vec[1], newsize );
        start = newsize + 1;
        if ( connective == NEG ) { /* -P <=> P -> BOTTOM */
            vec[0] = IMPL;
            for ( i=start ; i<size ; i++ )
                vec[i] = BOTTOM;
        }
        else {
            vec[0] = connective;
            make_prop( &vec[start], newsize );
        }
    }
}

/*------------------------ copy_prop ------------------------*/
/* Transform propositional vector "src" of length "src_sz" to  */
/* another propositional vector "dst" of length "dst_sz".      */
/* Caller must ensure that dst is large enough to accommodate  */
/* a copy of src.                                              */
```

```
/*-------------------------------------------------------------*/

static    void    copy_prop( src, src_sz, dst, dst_sz )
int    src[], src_sz, dst[], dst_sz;
{
    int    i;

    if ( (src_sz == 1) || (dst_sz == 1) )
        for ( i=0 ; i<dst_sz ; i++ ) dst[i] = src[0];
    else {
        dst[0] = src[0];
        src_sz >>= 1;
        dst_sz >>= 1;
        copy_prop( &src[1], src_sz,
               &dst[1], dst_sz );
        copy_prop( &src[src_sz+1], src_sz,
               &dst[dst_sz+1], dst_sz );
    }
}

/*------------------------- extend -------------------------*/
/* Functionally extend vector "vec", assumed to have PROP_SZ   */
/* significant elements, to VEC_SZ significant elements        */
/*-------------------------------------------------------------*/

void    extend( vec )
PROPOSITION    vec;
{
    int    i, j, k;


    k = PROP_SZ;
    vec[k++] = IMPL;
    vec[k++] = CONJ;
    vec[k++] = DISJ;
    for ( i=1 ; i<(PROP_SZ+3) ; i++ )
        for ( j=0 ; j<i ; j++ )
            vec[k++] = ABSDIF( vec[i], vec[j] );
}

/*---------------------- Inference rules ----------------------*/
/* The premise functions return random premise vectors         */
/* matching the templates of their inference rules.  The       */
/* conclusion functions return the correct conclusions for the */
/* argument premises.  Note that "conjoin" is different:  It   */
/* is not learned by the system; it is provided as a built-in  */
/* supplementary rule to be used by the proof system.          */
/*-------------------------------------------------------------*/
```

```
/*==================== CONJOIN (special case) ====================*/
                    /* P and Q => (P & Q) */

void    conjoin( p, q, p_and_q )
PROPOSITION    p, q, p_and_q;
{
    p_and_q[0] = CONJ;
    copy_prop( p,           PROP_SZ,
               &p_and_q[1], PROP_SZ>>1 );
    copy_prop( q,                          PROP_SZ,
               &p_and_q[(PROP_SZ>>1)+1], PROP_SZ>>1 );
    extend( p_and_q );
}

/*======================== REPETITION ========================*/
                          /* P => P */

static    int    premiseREP( vec )
PROPOSITION    vec;
{
    make_prop( vec, PROP_SZ );
    extend( vec );
}

static    int    conclusionREP( invec, outvec )
PROPOSITION    invec, outvec;
{
    copy_prop( invec,  PROP_SZ,
               outvec, PROP_SZ );
    extend( outvec );
}

/*======================== MODUS PONENS ========================*/
                  /* ( (P -> Q) & P ) => Q */

static int    premiseMP( vec )
PROPOSITION   vec;
{
    vec[0] = CONJ;
    vec[1] = IMPL;
    make_prop( &vec[2], PROP_SZ>>2 );
    copy_prop( &vec[2], PROP_SZ>>2,
               &vec[(PROP_SZ>>1)+1], PROP_SZ>>1);
    make_prop( &vec[(PROP_SZ>>2)+2], PROP_SZ>>2 );
    extend( vec );
```

```
}

static int    conclusionMP( invec, outvec )
PROPOSITION   invec, outvec;
{
    copy_prop( &invec[(PROP_SZ>>2)+2], PROP_SZ>>2,
           outvec,            PROP_SZ );
    extend( outvec );
}

/*==================== HYPOTHETICAL SYLLOGISM ====================*/
             /* ( (P -> Q) & (Q -> R) ) => (P -> R) */

static int    premiseHS( vec )
PROPOSITION   vec;
{
    vec[0] = CONJ;
    vec[1] = vec[(PROP_SZ>>1)+1] = IMPL;
    make_prop( &vec[2], PROP_SZ>>2 );
    make_prop( &vec[(PROP_SZ>>2)+2], PROP_SZ>>2 );
    copy_prop( &vec[(PROP_SZ>>2)+2], PROP_SZ>>2,
               &vec[(PROP_SZ>>1)+2], PROP_SZ>>2 );
    make_prop( &vec[PROP_SZ-(PROP_SZ>>2)], PROP_SZ>>2 );
    extend( vec );
}

static int    conclusionHS( invec, outvec )
PROPOSITION   invec, outvec;
{
    outvec[0] = IMPL;
    copy_prop( &invec[2], PROP_SZ>>2, &outvec[1], PROP_SZ>>1 );
    copy_prop( &invec[PROP_SZ-(PROP_SZ>>2)], PROP_SZ>>2,
               &outvec[(PROP_SZ>>1)+1],      PROP_SZ>>1 );
    extend( outvec );
}

/*==================== DISJUNCTIVE SYLLOGISM ====================*/
                   /* ( (P | Q) & ~P ) => Q */

static int    premiseDS( vec )
PROPOSITION   vec;
{
    int       i;

    vec[0] = CONJ;
    vec[1] = DISJ;
    vec[(PROP_SZ>>1)+1] = IMPL;
    make_prop( &vec[2], PROP_SZ>>2 );
```

```
    make_prop( &vec[(PROP_SZ>>2)+2], PROP_SZ>>2 );
    copy_prop( &vec[2],              PROP_SZ>>2,
               &vec[(PROP_SZ>>1)+2], PROP_SZ>>2 );
    for ( i=PROP_SZ-(PROP_SZ>>2) ; i<PROP_SZ ; i++ )
        vec[i] = BOTTOM;
    extend( vec );
}


static int    conclusionDS( invec, outvec )
PROPOSITION   invec, outvec;
{
    copy_prop( &invec[(PROP_SZ>>2)+2], PROP_SZ>>2,
               outvec,                 PROP_SZ );
    extend( outvec );
}

/*======================== MODUS TOLLENS ========================*/
                /* ( (P -> Q) & ~Q ) => ~P */

static int    premiseMT( vec )
PROPOSITION   vec;
{
    int       i;

    vec[0] = CONJ;
    vec[1] = vec[(PROP_SZ>>1)+1] = IMPL;
    make_prop( &vec[2],              PROP_SZ>>2 );
    make_prop( &vec[(PROP_SZ>>2)+2], PROP_SZ>>2 );
    copy_prop( &vec[(PROP_SZ>>2)+2], PROP_SZ>>2,
               &vec[(PROP_SZ>>1)+2], PROP_SZ>>2 );
    for ( i=PROP_SZ-(PROP_SZ>>2) ; i<PROP_SZ ; i++ )
        vec[i] = BOTTOM;
    extend( vec );
}

static int    conclusionMT( invec, outvec )
PROPOSITION   invec, outvec;
{
    int       i;

    outvec[0] = IMPL;
    copy_prop( &invec[2],  PROP_SZ>>2,
               &outvec[1], PROP_SZ>>1 );
    for ( i=(PROP_SZ>>1)+1 ; i<PROP_SZ ; i++ )
        outvec[i] = BOTTOM;
    extend( outvec );
}
```

```
/*================ DOUBLE NEGATION ELIMINATION ================*/
                        /* ~~P => P */

static int    premiseDNE( vec )
PROPOSITION   vec;
{
    int       i;

    vec[0] = vec[1] = IMPL;
    make_prop( &vec[2], PROP_SZ>>2 );
    for ( i=(PROP_SZ>>2)+2 ; i<PROP_SZ ; i++ )
        vec[i] = BOTTOM;
    extend( vec );
}

static int    conclusionDNE( invec, outvec )
PROPOSITION   invec, outvec;
{
    copy_prop( &invec[2], PROP_SZ>>2,
               outvec,    PROP_SZ );
    extend( outvec );
}

/*================ DOUBLE NEGATION INTRODUCTION ================*/
                        /* P => ~~P */

static int    premiseDNI( vec )
PROPOSITION   vec;
{
    PROPOSITION   p;

    make_prop( p, PROP_SZ>>2 );
    copy_prop( p, PROP_SZ>>2, vec, PROP_SZ );
    extend( vec );
}

static int    conclusionDNI( invec, outvec )
PROPOSITION   invec, outvec;
{
    int       i;

    outvec[0] = outvec[1] = IMPL;
    copy_prop( invec, PROP_SZ, &outvec[2], PROP_SZ>>2 );
    for ( i=(PROP_SZ>>2)+2 ; i<PROP_SZ ; i++ )
        outvec[i] = BOTTOM;
    extend( outvec );
}
```

```
/*======================== SIMPLIFICATION ========================*/
                        /* (P & Q) => P */

static int    premiseSIMP( vec )
PROPOSITION   vec;
{
    int       i;

    vec[0] = CONJ;
    make_prop( &vec[1], PROP_SZ>>1 );
    make_prop( &vec[(PROP_SZ>>1)+1], PROP_SZ>>1 );
    extend( vec );
}

static int    conclusionSIMP( invec, outvec )
PROPOSITION   invec, outvec;
{
    copy_prop( &invec[1], PROP_SZ>>1, outvec, PROP_SZ );
    extend( outvec );
}

/*====================== OR-COMMUTATIVE LAW ======================*/
                    /* (P | Q) => (Q | P) */

static int    premiseCOM_OR( vec )
PROPOSITION   vec;
{
    vec[0] = DISJ;
    make_prop( &vec[1], PROP_SZ>>1 );
    make_prop( &vec[(PROP_SZ>>1)+1], PROP_SZ>>1 );
    extend( vec );
}

static int    conclusionCOM_OR( invec, outvec )
PROPOSITION   invec, outvec;
{
    outvec[0] = DISJ;
    copy_prop( &invec[1],               PROP_SZ>>1,
               &outvec[(PROP_SZ>>1)+1], PROP_SZ>>1 );
    copy_prop( &invec[(PROP_SZ>>1)+1],    PROP_SZ>>1,
               &outvec[1],                PROP_SZ>>1 );
    extend( outvec );
}

/*===================== AND-COMMUTATIVE LAW =====================*/
                    /* (P & Q) => (Q & P) */

static int    premiseCOM_AND( vec )
```

```
PROPOSITION   vec;
{
    vec[0] = CONJ;
    make_prop( &vec[1], PROP_SZ>>1 );
    make_prop( &vec[(PROP_SZ>>1)+1], PROP_SZ>>1 );
    extend( vec );
}

static int    conclusionCOM_AND( invec, outvec )
PROPOSITION   invec, outvec;
{
    outvec[0] = CONJ;
    copy_prop( &invec[1],                PROP_SZ>>1,
               &outvec[(PROP_SZ>>1)+1], PROP_SZ>>1 );
    copy_prop( &invec[(PROP_SZ>>1)+1],    PROP_SZ>>1,
               &outvec[1],                PROP_SZ>>1 );
    extend( outvec );
}

/*==================== CONSTRUCTIVE DILEMMA ====================*/
      /* ((P | Q) & ((P -> R) & (Q -> S))) => (R | S) */

static int    premiseCD( vec )
PROPOSITION   vec;
{
    vec[0] = vec[(PROP_SZ>>1)+1] = CONJ;
    vec[1] = DISJ;
    vec[(PROP_SZ>>1)+2] = vec[PROP_SZ-(PROP_SZ>>2)] = IMPL;
    make_prop( &vec[(PROP_SZ>>1)+3], PROP_SZ>>3 );
    make_prop( &vec[(PROP_SZ>>1)+(PROP_SZ>>3)+3], PROP_SZ>>3 );
    make_prop( &vec[PROP_SZ-(PROP_SZ>>2)+1], PROP_SZ>>3 );
    make_prop( &vec[PROP_SZ-(PROP_SZ>>3)], PROP_SZ>>3 );
    copy_prop( &vec[(PROP_SZ>>1)+3],    PROP_SZ>>3,
               &vec[2],                 PROP_SZ>>2 );
    copy_prop( &vec[PROP_SZ-(PROP_SZ>>2)+1],PROP_SZ>>3,
               &vec[(PROP_SZ>>2)+2],        PROP_SZ>>2 );
    extend( vec );
}

static int    conclusionCD( invec, outvec )
PROPOSITION   invec, outvec;
{
    outvec[0] = DISJ;
    copy_prop( &invec[(PROP_SZ>>1)+(PROP_SZ>>3)+3], PROP_SZ>>3,
               &outvec[1],                          PROP_SZ>>1 );
    copy_prop( &invec[PROP_SZ-(PROP_SZ>>3)], PROP_SZ>>3,
               &outvec[(PROP_SZ>>1)+1],      PROP_SZ>>1 );
    extend( outvec );
```

```
}

/*==================== DESTRUCTIVE DILEMMA ====================*/
      /* ((~R | ~S) & ((P -> R) & (Q -> S))) => (~P | ~Q) */

static int    premiseDD( vec )
PROPOSITION   vec;
{
    int       i;

    vec[0] = vec[(PROP_SZ>>1)+1] = CONJ;
    vec[1] = DISJ;
    vec[2] = vec[(PROP_SZ>>1)+2] = vec[PROP_SZ-(PROP_SZ>>2)]
                    = vec[(PROP_SZ>>2)+2] = IMPL;
    for ( i=0 ; i<(PROP_SZ>>3) ; i++ )
        vec[(PROP_SZ>>3)+3+i] =
                    vec[(PROP_SZ>>2)+(PROP_SZ>>3)+3+i] = BOTTOM;
    make_prop( &vec[(PROP_SZ>>1)+3], PROP_SZ>>3 );
    make_prop( &vec[(PROP_SZ>>1)+(PROP_SZ>>3)+3], PROP_SZ>>3 );
    make_prop( &vec[PROP_SZ-(PROP_SZ>>2)+1], PROP_SZ>>3 );
    make_prop( &vec[PROP_SZ-(PROP_SZ>>3)], PROP_SZ>>3 );
    copy_prop( &vec[(PROP_SZ>>1)+(PROP_SZ>>3)+3], PROP_SZ>>3,
               &vec[3],                           PROP_SZ>>3 );
    copy_prop( &vec[PROP_SZ-(PROP_SZ>>3)], PROP_SZ>>3,
               &vec[(PROP_SZ>>2)+3],       PROP_SZ>>3 );
    extend( vec );
}

static int    conclusionDD( invec, outvec )
PROPOSITION   invec, outvec;
{
    outvec[0] = DISJ;
    outvec[1] = outvec[(PROP_SZ>>1)+1] = IMPL;
    copy_prop( &invec[(PROP_SZ>>1)+3], PROP_SZ>>3,
               &outvec[2],             PROP_SZ>>2 );
    copy_prop( &invec[PROP_SZ-(PROP_SZ>>2)+1], PROP_SZ>>3,
               &outvec[(PROP_SZ>>1)+2],        PROP_SZ>>2);
    copy_prop( &invec[(PROP_SZ>>3)+3],  PROP_SZ>>3,
               &outvec[(PROP_SZ>>2)+2], PROP_SZ>>2 );
    copy_prop( &invec[(PROP_SZ>>3)+3],        PROP_SZ>>3,
               &outvec[PROP_SZ-(PROP_SZ>>2)], PROP_SZ>>2 );
    extend( outvec );
}

/*==================== CONTRAPOSITION ====================*/
                /* ( ~P -> ~Q ) => (Q -> P) */

static int    premiseCONTRA( vec )
```

```
PROPOSITION   vec;
{
    int       i;

    vec[0] = vec[1] = vec[(PROP_SZ>>1)+1] = IMPL;
    for ( i=0 ; i<(PROP_SZ>>2) ; i++ )
        vec[(PROP_SZ>>2)+2+i] =
                          vec[PROP_SZ-(PROP_SZ>>2)+i] = BOTTOM;
    make_prop( &vec[2], PROP_SZ>>2 );
    make_prop( &vec[(PROP_SZ>>1)+2], PROP_SZ>>2 );
    extend( vec );
}

static int    conclusionCONTRA( invec, outvec )
PROPOSITION   invec, outvec;
{
    outvec[0] = IMPL;
    copy_prop( &invec[(PROP_SZ>>1)+2], PROP_SZ>>2,
               &outvec[1],             PROP_SZ>>1 );
    copy_prop( &invec[2],               PROP_SZ>>2,
               &outvec[(PROP_SZ>>1)+1], PROP_SZ>>1);
    extend( outvec );
}

/*================== IMPLICATION INTRODUCTION ====================*/
                  /* (~P | Q) => (P -> Q) */

static int    premiseIMPL1( vec )
PROPOSITION   vec;
{
    int      i;

    vec[0] = DISJ;
    vec[1] = IMPL;
    for ( i=0 ; i<(PROP_SZ>>2) ; i++ )
        vec[(PROP_SZ>>2)+2+i] = BOTTOM;
    make_prop( &vec[2], PROP_SZ>>2 );
    make_prop( &vec[(PROP_SZ>>1)+1], PROP_SZ>>1 );
    extend( vec );
}

static int    conclusionIMPL1( invec, outvec )
PROPOSITION   invec, outvec;
{
    outvec[0] = IMPL;
    copy_prop( &invec[(PROP_SZ>>1)+1],  PROP_SZ>>1,
               &outvec[(PROP_SZ>>1)+1], PROP_SZ>>1 );
    copy_prop( &invec[2],  PROP_SZ>>2,
```

```
                &outvec[1], PROP_SZ>>1);
    extend( outvec );
}

/*================= IMPLICATION ELIMINATION ====================*/
                    /* (P -> Q) => (~P | Q) */

static int    premiseIMPL2( vec )
PROPOSITION   vec;
{
    PROPOSITION   p;

    vec[0] = IMPL;
    make_prop( p, PROP_SZ>>2 );
    copy_prop( p, PROP_SZ>>2, &vec[1], PROP_SZ>>1 );
    make_prop( &vec[(PROP_SZ>>1)+1], PROP_SZ>>1 );
    extend( vec );
}

static int    conclusionIMPL2( invec, outvec )
PROPOSITION   invec, outvec;
{
    int       i;

    outvec[0] = DISJ;
    outvec[1] = IMPL;
    copy_prop( &invec[1],  PROP_SZ>>1,
               &outvec[2], PROP_SZ>>2 );
    for ( i=(PROP_SZ>>2)+2 ; i<(PROP_SZ>>1)+1 ; i++ )
        outvec[i] = BOTTOM;
    copy_prop( &invec[(PROP_SZ>>1)+1],  PROP_SZ>>1,
               &outvec[(PROP_SZ>>1)+1], PROP_SZ>>1 );
    extend( outvec );
}

/*================= DEMORGAN'S LAW (AND => OR) ==================*/
                    /* ~(P & Q) => (~P | ~Q)  */

static int    premiseDM1( vec )
PROPOSITION   vec;
{
    int       i;

    vec[0] = IMPL;
    vec[1] = CONJ;
    for ( i=(PROP_SZ>>1)+1 ; i<PROP_SZ ; i++ )
        vec[i] = BOTTOM;
    make_prop( &vec[2], PROP_SZ>>2 );
```

```
    make_prop( &vec[(PROP_SZ>>2)+2], PROP_SZ>>2 );
    extend( vec );
}

static int    conclusionDM1( invec, outvec )
PROPOSITION   invec, outvec;
{
    int       i;

    outvec[0] = DISJ;
    outvec[1] = outvec[(PROP_SZ>>1)+1] = IMPL;
    for ( i=0 ; i<(PROP_SZ>>2) ; i++ )
        outvec[(PROP_SZ>>2)+2+i] =
                    outvec[PROP_SZ-(PROP_SZ>>2)+i] = BOTTOM;
    copy_prop( &invec[2],  PROP_SZ>>2,
               &outvec[2], PROP_SZ>>2 );
    copy_prop( &invec[(PROP_SZ>>2)+2],  PROP_SZ>>2,
               &outvec[(PROP_SZ>>1)+2], PROP_SZ>>2);
    extend( outvec );
}

/*================ DEMORGAN'S LAW (OR => AND) ================*/
                  /* ~(P | Q) => (~P & ~Q)  */

static int    premiseDM2( vec )
PROPOSITION   vec;
{
    int       i;

    vec[0] = IMPL;
    vec[1] = DISJ;
    for ( i=(PROP_SZ>>1)+1 ; i<PROP_SZ ; i++ )
        vec[i] = BOTTOM;
    make_prop( &vec[2], PROP_SZ>>2 );
    make_prop( &vec[(PROP_SZ>>2)+2], PROP_SZ>>2 );
    extend( vec );
}

static int    conclusionDM2( invec, outvec )
PROPOSITION   invec, outvec;
{
    int       i;

    outvec[0] = CONJ;
    outvec[1] = outvec[(PROP_SZ>>1)+1] = IMPL;
    for ( i=0 ; i<(PROP_SZ>>2) ; i++ )
        outvec[(PROP_SZ>>2)+2+i] =
                    outvec[PROP_SZ-(PROP_SZ>>2)+i] = BOTTOM;
```

```
    copy_prop( &invec[2],  PROP_SZ>>2,
               &outvec[2], PROP_SZ>>2 );
    copy_prop( &invec[(PROP_SZ>>2)+2],  PROP_SZ>>2,
               &outvec[(PROP_SZ>>1)+2], PROP_SZ>>2);
    extend( outvec );
}

/*================== ARRAY OF INFERENCE RULES ==================*/
/* NOTE:  RULE_TOTAL and RULE must be updated whenever a new   */
/*        inference rule is added .                            */

#define RULE_TOTAL    17

static RULE    rule_set[] =
     {
      { premiseREP, conclusionREP, "Repetition" },
      { premiseMP, conclusionMP, "Modus Ponens" },
      { premiseMT, conclusionMT, "Modus Tollens" },
      { premiseDS, conclusionDS, "Disjunctive Syllogism" },
      { premiseHS, conclusionHS, "Hypothetical Syllogism" },
      { premiseDNE, conclusionDNE, "Double Negation Elimination" },
      { premiseDNI, conclusionDNI, "Double Negation Introduction" },
      { premiseSIMP, conclusionSIMP, "Simplification" },
      { premiseCOM_OR, conclusionCOM_OR, "OR-Commutativity" },
      { premiseCOM_AND, conclusionCOM_AND, "AND-Commutativity" },
      { premiseCD, conclusionCD, "Constructive Dilemma" },
      { premiseDD, conclusionDD, "Destructive Dilemma" },
      { premiseCONTRA, conclusionCONTRA, "Contraposition" },
      { premiseIMPL1, conclusionIMPL1, "Implication Introduction" },
      { premiseIMPL2, conclusionIMPL2, "Implication Elimination" },
      { premiseDM1, conclusionDM1, "AND=>OR DeMorg" },
      { premiseDM2, conclusionDM2, "OR=>AND DeMorg" }
     };

/*----------------------- select_rules ----------------------*/
/* Get user's choice from available inference rules.  Return  */
/* number of rules selected.                                  */
/*------------------------------------------------------------*/

int    select_rules( rule )
RULE    rule[];
{
    int    r, rcount;
    char    answer[50], *p;

    rcount = 0;
    for ( r=0 ; r<RULE_TOTAL ; r++ ) {
        do {
```

```
            printf( "Include rule \"%s\"? (Y/N) ",
                                            rule_set[r].name );
            fgets( answer, 50, stdin );
            p = (char *) strchr(answer,'\n');
            if ( p != NULL ) *p = '\0';
        } while ( (strlen(answer) != 1) ||
                                        !strspn(answer,"yYnN") );
        if ( strspn(answer,"yY") ) {
            rule[rcount].premise = rule_set[r].premise;
            rule[rcount].conclusion = rule_set[r].conclusion;
            rule[rcount].name = rule_set[r].name;
            rcount++;
        }
    }
    return( rcount );
}

/*------------------------ add_rules ------------------------*/
/* Let user select previously unselected inference rules.    */
/* Return number of rules added.                             */
/*-----------------------------------------------------------*/

int    add_rules( rule, old_rcount )
RULE    rule[];
{
    int    c, i, r, rcount, ok;
    char    answer[50], *p;

    rcount = old_rcount;
    for ( r=0 ; r<RULE_TOTAL ; r++ ) {
        do {
            ok = 1;
            for ( i=0 ; i<old_rcount  && ok ; i++ )
                if ( rule[i].premise == rule_set[r].premise )
                    ok = 0;
            if ( ok ) {
                printf( "Include rule \"%s\"? (Y/N) ",
                                            rule_set[r].name );
                fgets( answer, 50, stdin );
                p = (char *) strchr(answer,'\n');
                if ( p != NULL ) *p = '\0';
            }
        } while ( ok && ( (strlen(answer) != 1) ||
                                        !strspn(answer,"yYnN") ) );
        if ( strspn(answer,"yY") && ok ) {
            rule[rcount].premise = rule_set[r].premise;
            rule[rcount].conclusion = rule_set[r].conclusion;
            rule[rcount].name = rule_set[r].name;
```

```
            rcount++;
        }
    }
    return( rcount );
}

/*------------------------ read_net -------------------------*/
/* Read rule array and network weights from a file. In case of */
/* success, return number of rules; otherwise, return error.  */
/*-----------------------------------------------------------*/

int     read_net( rule, badflag_wt, wt )
RULE    rule[];
int     badflag_wt[MAX_RULES][VEC_SZ];
int     wt[MAX_RULES][VEC_SZ][VEC_SZ];
{
    FILE    *fp;
    char    str[50], *p;
    int     i, j, r, rules_in_file;

    printf("Enter name of network file to read: ");
    fgets( str, 50, stdin );
    p = (char *) strchr(str,'\n');
    if ( p != NULL ) *p = '\0';
    if ( (fp = fopen(str, "r")) == NULL ) {
        printf("Error opening %s for reading\n", str);
        return( ERROR );
    }
    fscanf( fp, "%d\n", &rules_in_file );
    for( r=0 ; r<rules_in_file ; r++ ) {
        fgets( str, 50, fp );
        i = strlen(str);
        if ( str[--i] == '\n' ) str[i] = '\0';
        for ( i=0 ; strcmp(str, rule_set[i].name); i++ ) {
            if ( i==RULE_TOTAL ) {
                printf("Rule %s does not exist\n", str);
                return( ERROR );
            }
        }
        rule[r].premise = rule_set[i].premise;
        rule[r].conclusion = rule_set[i].conclusion;
        rule[r].name = rule_set[i].name;
        for ( i=0 ; i<VEC_SZ ; i++ ) {
            fscanf( fp, "%d\n", &badflag_wt[r][i] );
            for ( j=0 ; j<VEC_SZ ; j++ )
                fscanf( fp, "%d\n", &wt[r][i][j] );
        }
    }
```

```
    fclose( fp );
    return( rules_in_file );
}

/*------------------------ get_rule -------------------------*/
/* Return a pointer to the rule whose name is r_name if the  */
/* rule exists in rule_set; otherwise, return NULL.          */
/*-----------------------------------------------------------*/

RULE    *get_rule( r_name )
char    *r_name;
{
    int    i;

   -for ( i=0 ; i<RULE_TOTAL ; i++ )
        if ( !strcmp(r_name, rule_set[i].name) )
            return( &rule_set[i] );
    return( NULL );
}
```

```
/*****************************************************************/
/*****************                               *****************/
/*****************          parse.c              *****************/
/*****************                               *****************/
/*****************************************************************/

/* GRAMMAR:

    Sentence ---> atom
    Sentence ---> #              (# means "bottom")
    Sentence ---> ~ Sentence
    Sentence ---> ( Sentence Binary

    Binary ---> | Sentence )
    Binary ---> & Sentence )
    Binary ---> -> Sentence )
    Binary ---> )
*/


#include    "neurnets.h"

/* Forward declaration needed due to mutual recursion */

static    char    *sentence();

/*-------------------------- scan ---------------------------*/
/* Cut first token from str, returning the remaining string.  */
/* On return, token points to the token type, or FAILURE.     */
/*------------------------------------------------------------*/

static  char    *scan( str, token )
char    *str;
int     *token;
{
    if ( *str == '\0' ) {
        *token = FAILURE;
        return( str );
    }
    for ( ; isspace(*str) ; str++ );
    switch ( *str ) {
        case '(' : *token = LPAR;
                   break;
        case ')' : *token = RPAR;
                   break;
        case '~' : *token = NEG;
                   break;
```

```
        case '|' : *token = DISJ;
                   break;
        case '&' : *token = CONJ;
                   break;
        case '#' : *token = CONTRADICTION;
                   break;
        case '-' : *token = *(++str) == '>' ? IMPL : FAILURE;
                   break;
        default:   *token = isupper(*str) ? *str : FAILURE;
    }
    str++;
    return( str );
}

/*------------------------- binary ---------------------------*/
/* Transcribe string form proposition to vector form using the */
/* grammar above.  Mutually recursive with sentence().         */
/*-------------------------------------------------------------*/

static    char    *binary( vec, str, size, tokenp, result )
int       vec[];
char      *str;
int       size, *tokenp, *result;
{
    int   last;

    str = scan( str, tokenp );
    switch ( *tokenp ) {
        case RPAR : *result = SUCCESS; return( str );
        case DISJ :
        case CONJ :
        case IMPL : str = sentence( vec, str, size, result );
                    if ( *result == FAILURE )
                        return( str );
                    str = scan( str, &last );
                    if ( last == RPAR )
                        *result = SUCCESS;
                    else
                        *result = FAILURE;
                    return( str );
    }
    *result = FAILURE;
    return( str );
}

/*------------------------ sentence -------------------------*/
/* Transcribe string form proposition to vector form using the */
/* grammar above.  Mutually recursive with binary().           */
```

```
/*-------------------------------------------------------------*/
static    char    *sentence( vec, str, size, result )
int    vec[];
char    *str;
int    size, *result;
{
    int    i, token, newsize;

    str = scan( str, &token );
    if ( isupper(token) || token == CONTRADICTION ) {
        for ( i=0 ; i<size ; i++ ) vec[i] = token;
        *result = SUCCESS;
        return( str );
    }
    newsize = size>>1;
    if ( token == LPAR ) {
        str = sentence(&vec[1], str, newsize, result);
        if ( *result == FAILURE ) return( str );
        str = binary( &vec[newsize+1], str,
                                    newsize, &vec[0], result );
        return( str );
    }
    if ( token == NEG ) {
        vec[0] = IMPL;
        for ( i=newsize+1 ; i<size ; i++ ) vec[i] = BOTTOM;
        str = sentence( &vec[1], str, newsize, result );
        return( str );
    }
    *result = FAILURE;
    return( str );
}

/*---------------------- prop_to_string ----------------------*/
/* Transcribe the vector vec to string form str.               */
/*-------------------------------------------------------------*/

static    int    prop_to_string( vec, size, str )
int    vec[];
int    size;
char    *str;
{
    int    smaller, temp;

    if ( size < 1 ) return( FAILURE );
    if ( ATOM(vec[0]) ) {
        *(str++) = vec[0];
        *str = '\0';
        return ( SUCCESS );
```

```
    }
    if ( vec[0] == BOTTOM ) {
        *(str++) = '#';
        *str = '\0';
        return( SUCCESS );
    }
    smaller = size>>1;
    if ( (vec[0] == IMPL) && (vec[smaller+1] == BOTTOM) ) {
        *(str++) = '~';
        *str = '\0';
        return( prop_to_string(&vec[1], smaller, str) );
    }
    else {
        *(str++) = '(';
        *str = '\0';
        if ( !prop_to_string(&vec[1], smaller, str) )
            return( FAILURE );
        switch( vec[0] ) {
            case IMPL: strcat( str, " -> " ); break;
            case DISJ: strcat( str, " | " ); break;
            case CONJ: strcat( str, " & " ); break;
            default: return( FAILURE );
        }
        temp = prop_to_string( &vec[smaller+1],
                               smaller, strchr(str,'\0') );
        if ( temp ) {
            strcat( str, ")" );
            return( SUCCESS );
        }
        else
            return ( FAILURE );
    }
}

/*------------------------ str_prop -------------------------*/
/* Convert string form proposition (str) to vector form (vec). */
/*------------------------ -------------------------------*/

int     str_prop( str, vec )
char     *str;
int     vec[];
{
    int     result;

    str = sentence(vec,str,PROP_SZ,&result);
    if ( (result == SUCCESS) &&
                            (str != NULL) && (*str == '\0') ) {
        extend( vec );
```

```
        return( SUCCESS );
    }
    else
        return( FAILURE );
}

/*------------------------ str_prop -------------------------*/
/* Convert vector form proposition (vec) to string form (str). */
/*-----------------------------------------------------------*/

/* prop_str()  converts a proposition to a string */

int    prop_str( vec, str )
int    vec[];
char    *str;
{
    return( prop_to_string(vec, PROP_SZ, str) );
}
```

```
/*****************************************************************/
/*****************                                   *************/
/*****************          memarch.c                *************/
/*****************                                   *************/
/*****************************************************************/

#include    "neurnets.h"

#define     EMPTY(q)  ((q) == NULL)

typedef     struct mem_q {
                        PROPOSITION  prop;
                        char         *inf_rule;
                        struct mem_q *support1, *support2, *next;
            } *MEM_Q;



/*-------------------- global variables ----------------------*/

static      MEM_Q     head_q = NULL;
static      MEM_Q     crnt_q = NULL;
static      MEM_Q     tail_q = NULL;

/*---------------------- flush_memory -------------------------*/
/* Empty memory queue and return memory to system.             */
/*-------------------------------------------------------------*/

void    flush_memory()
{
    MEM_Q    temp_q;

    while( !EMPTY(head_q) ) {
        temp_q = head_q;
        head_q = head_q->next;
        free( temp_q );
    }
    crnt_q = tail_q = NULL;
}

/*------------------------ get_prop ---------------------------*/
/* Copy the current proposition in memory into parameter p.    */
/*-------------------------------------------------------------*/

int    get_prop( p )
PROPOSITION    p;
{
    if ( EMPTY(crnt_q) )
```

```
        return( FAILURE );
    else {
        memcpy( (char *)p, (char *)crnt_q->prop,
                                        sizeof(PROPOSITION) );
        return( SUCCESS );
    }
}

/*------------------------ next_prop -------------------------*/
/* Advance the current proposition pointer.                   */
/*------------------------------------------------------------*/

void    next_prop()
{
    if ( !EMPTY(crnt_q) ) crnt_q = crnt_q->next;
}

/*------------------------ complexity ------------------------*/
/* Return depth of nesting of connectives in proposition p.   */
/*------------------------------------------------------------*/

static    int    complexity( p, size )
int    p[], size;
{
    int    newsize, left, right;
    switch( p[0] ) {
        case IMPL :
        case DISJ :
        case CONJ :
            newsize = size >> 1;
            left = complexity( &p[1], newsize );
            right = complexity( &p[newsize+1], newsize );
            return( 1 + MAX(left,right) );
    }
    return( 0 );
}

/*------------------------ find_prop -------------------------*/
/* Return location of p in memory queue, or NULL if not found. */
/*------------------------------------------------------------*/

static    MEM_Q    find_prop( p )
PROPOSITION    p;
{
    MEM_Q    q;

    q = head_q;
```

```
    while( !EMPTY(q) && !SAME_PROP(q->prop, p) ) q = q->next;
    return( q );
}

/*-------------------------- insert -------------------------*/
/* Insert p in memory if not there. Record premises s1 and s2  */
/* from which p was derived, and the inference rule used.      */
/* Return a pointer to the proposition in memory.              */
/*-------------------------------------------------------------*/

static    MEM_Q    insert( p, s1, s2, rulename)
PROPOSITION    p;
MEM_Q        s1, s2;
char        *rulename;
{
    MEM_Q    q;

    if ( tail_q == NULL ) {
        tail_q = (MEM_Q) malloc(sizeof(struct mem_q));
        head_q = crnt_q = tail_q;
    }
    else {
        q = find_prop( p );
        if ( EMPTY(q) ) {
            tail_q->next = (MEM_Q) malloc(sizeof(struct mem_q));
            tail_q = tail_q->next;
        }
        else
            return( q );
    }
    if ( tail_q == NULL ) {
        pause("\nMemory insertion error.  Press <return>...");
        exit( -1 );
    }
    memcpy( (char *)tail_q->prop, (char *)p, sizeof(PROPOSITION) );
    tail_q->support1 = s1;
    tail_q->support2 = s2;
    tail_q->inf_rule = rulename;
    tail_q->next = NULL;
    return( tail_q );
}

/*----------------------- conjoin_all ----------------------*/
/* Form all possible conjunctions of p and propositions in the */
/* memory queue and add insert them into the queue.            */
/*-------------------------------------------------------------*/

static    void    conjoin_all( p )
```

```
MEM_Q    p;
{
    PROPOSITION    conj;
    MEM_Q          q;

    if ( complexity(p->prop, PROP_SZ) >= MAX_COMPLEXITY ) return;

    q = head_q;
    while( !EMPTY(q) ) {
        if ( complexity(q->prop, PROP_SZ) < MAX_COMPLEXITY ) {
            conjoin( p->prop, q->prop, conj );
            insert( conj, p, q, "Conjunction" );
        }
        q = q->next;
    }
}

/*---------------------- memorize_premises ----------------------*/
/* Record all premises (and conjunctions) in memory queue.      */
/*---------------------------------------------------------------*/

void    memorize_premises( p_list )
PROP_LIST    p_list;
{
    MEM_Q    p;

    while ( !EMPTY(p_list) ) {
        p = insert( p_list->prop, NULL, NULL, "Premise" );
        conjoin_all( p );
        p_list = p_list->next;
    }
}

/*--------------------- memorize_inference ---------------------*/
/* Record inference (and conjunctions) in memory queue.         */
/*---------------------------------------------------------------*/

int    memorize_inference( conclusion, rulestr )
PROPOSITION    conclusion;
char          *rulestr;
{
    MEM_Q    p;

    if ( EMPTY(crnt_q) ) return( FAILURE );
    p = insert( conclusion, crnt_q, NULL, rulestr );
    conjoin_all( p );
/*    crnt_q = crnt_q->next;  */
    return( SUCCESS );
```

```
}

/*------------------------ build_proof ------------------------*/
/* Print proof of q by preorder traversal of "support tree"    */
/*-------------------------------------------------------------*/


static    void    build_proof( q )
MEM_Q    q;
{
    char    str[50];

    if ( !EMPTY(q) ) {
        prop_str( q->prop, str );
        if ( q->support1 == NULL )
            printf( "%s is a premise.\n", str );
        else {
            printf( "%s follows from ", str );
            prop_str( (q->support1)->prop, str );
            if ( q->support2 == NULL )
                printf( "%s by %s.\n", str, q->inf_rule );
            else {
                printf( "%s and ", str );
                prop_str( (q->support2)->prop, str );
                printf( "%s by %s.\n", str, q->inf_rule );
            }
        }
        build_proof( q->support1 );
        build_proof( q->support2 );
    }
}

/*------------------------ print_proof ------------------------*/
/* If p is in memory, print its proof and return success;      */
/* otherwise return failure                                    */
/*-------------------------------------------------------------*/

int    print_proof( p )
PROPOSITION    p;
{
    MEM_Q    q;

    q = find_prop( p );
    if ( EMPTY(q) )
        return( FAILURE );
    else {
        build_proof( q );
        pause( "\nPress <return>." );
```

```
        return( SUCCESS );
    }
}

/*----------------------- print_memory -----------------------*/
/* Display the contents of the memory queue in string form     */
/*-------------------------------------------------------------*/

void    print_memory()
{
    char    str[50];
    MEM_Q    q;

    CLEAR_SCREEN;
    printf( "Contents of memory\n" );
    printf( "==================\n" );
    q = head_q;
    while ( !EMPTY(q) ) {
        prop_str( q->prop, str );
        printf( "%s\n", str );
        q = q->next;
    }
    pause( "Press <return>..." );
}
```

```
/*****************************************************************/
/*****************                                  **************/
/*****************          utility.c               **************/
/*****************                                  **************/
/*****************************************************************/

#include "neurnets.h"

/*------------------------ get_option ------------------------*/
/* Display prompt and wait for valid selection from options.  */
/*-----------------------------------------------------------*/

int get_option( prompt, options )
char *prompt, *options;
{
char str[50], *pos;

do {
printf( "%s ", prompt );
fgets( str, 50, stdin );
pos = (char *) strchr(str,'\n');
if ( pos ) *pos = '\0';
if ( (strlen(str) == 1) && islower(*str) )
*str = toupper(*str);
} while ( ( strlen(str) != 1) || !strspn(str,options) );
return( *str );
}

/*---------------------------- yes ----------------------------*/
/* Display prompt and wait for yes or no answer.              */
/*-------------------------------------------------------------*/

int yes( question )
char *question;
{
char prompt[50];

strcpy( prompt, question );
strcat( prompt, " (Y/N) " );
return( get_option(prompt, "YN") == 'Y' );
}

/*--------------------------- pause ---------------------------*/
/* Display prompt and wait for carriage return.               */
/*-------------------------------------------------------------*/

void pause( prompt )
```

```
char *prompt;
{
char str[50];

printf("%s", prompt);
fgets( str, 50, stdin );
}

/*------------------------ all_white ------------------------*/
/* TRUE iff string s consists entirely of whitespace.       */
/*-------------------------------------------------------------*/

int all_white( s )
char *s;
{
for ( ; *s ; s++ )
if ( !isspace(*s) ) return( FALSE );
return( TRUE );
}

/*------------------------ all_scan ------------------------*/
/* Return pointer to first invalid character in s.  If all  */
/* characters are valid return pointer to terminal null byte.  */
/*-------------------------------------------------------------*/

char *all_scan( s )
char *s;
{
return( s +
            strspn(s,"ABCDEFGHIJKLMNOPQRSTUVWXYZ()|&~-> \t\n") );
}
```

Appendix B

Some Simulator Proofs

The following proofs were constructed by the simulator. The problems were chosen from a set of exercises in Copi's *Introduction to Logic* ([4]), an undergraduate logic textbook. The proofs are nearly verbatim transcripts of the simulator output; the only changes are the deletion of intervening menus, and the addition of whitespace for greater legibility on paper.

```
========================================
Enter propositions (blank line to exit):
(A -> B)
(A & C)
Enter proposition to prove: B
B follows from
              ((A -> B) & A)
              by Modus Ponens.
((A -> B) & A) follows from
              (A -> B) and A
              by Conjunction.
(A -> B) is a premise.
A follows from
              (A & C)
              by Simplification.
(A & C) is a premise.
========================================

Enter propositions (blank line to exit):
(R -> S)
(R | T)
(T -> U)
Enter proposition to prove: (S | U)
```

```
(S | U) follows from
              ((R | T) & ((R -> S) & (T -> U)))
              by Constructive Dilemma.
((R | T) & ((R -> S) & (T -> U))) follows from
              (R | T) and ((R -> S) & (T -> U))
              by Conjunction.
(R | T) is a premise.
((R -> S) & (T -> U)) follows from
              (R -> S) and (T -> U)
              by Conjunction.
(R -> S) is a premise.
(T -> U) is a premise.
==========================================

==========================================
Enter propositions (blank line to exit):
((V&W) | (X->Y))
(Z -> X)
~(V & W)
Enter proposition to prove: (Z -> Y)
(Z -> Y) follows from
              ((Z -> X) & (X -> Y))
              by Hypothetical Syllogism.
((Z -> X) & (X -> Y)) follows from
              (Z -> X) and (X -> Y)
              by Conjunction.
(Z -> X) is a premise.
(X -> Y) follows from
              (((V & W) | (X -> Y)) & ~(V & W))
              by Disjunctive Syllogism.
(((V & W) | (X -> Y)) & ~(V & W)) follows from
              ((V & W) | (X -> Y)) and ~(V & W)
              by Conjunction.
((V & W) | (X -> Y)) is a premise.
~(V & W) is a premise.
==========================================

==========================================
Enter propositions (blank line to exit):
((A -> B) & (C -> D))
(B -> D)
((B -> D) -> (A | C))
Enter proposition to prove: (B | D)
(B | D) follows from
              ((A | C) & ((A -> B) & (C -> D)))
              by Constructive Dilemma.
((A | C) & ((A -> B) & (C -> D))) follows from
              (A | C) and ((A -> B) & (C -> D))
```

```
                by Conjunction.
(A | C) follows from
                (((B -> D) -> (A | C)) & (B -> D))
                by Modus Ponens.
(((B -> D) -> (A | C)) & (B -> D)) follows from
                ((B -> D) -> (A | C)) and (B -> D)
                by Conjunction.
((B -> D) -> (A | C)) is a premise.
(B -> D) is a premise.
((A -> B) & (C -> D)) is a premise.
========================================

========================================
Enter propositions (blank line to exit):
(E | ~F)
(F -> ~G)
~E
Enter proposition to prove: ((F -> ~G) & ~F)
((F -> ~G) & ~F) follows from
                (F -> ~G) and ~F
                by Conjunction.
(F -> ~G) is a premise.
~F follows from
                ((E | ~F) & ~E)
                by Disjunctive Syllogism.
((E | ~F) & ~E) follows from
                (E | ~F) and ~E
                by Conjunction.
(E | ~F) is a premise.
~E is a premise.
========================================

========================================
Enter propositions (blank line to exit):
(A -> B)
(A | C)
~B
Enter proposition to prove: (C & ~A)
(C & ~A) follows from
                C and ~A
                by Conjunction.
C follows from
                ((A | C) & ~A)
                by Disjunctive Syllogism.
((A | C) & ~A) follows from
                (A | C) and ~A
                by Conjunction.
(A | C) is a premise.
```

```
~A follows from
              ((A -> B) & ~B)
              by Hypothetical Syllogism.
((A -> B) & ~B) follows from
              (A -> B) and ~B
              by Conjunction.
(A -> B) is a premise.
~B is a premise.
~A follows from
              ((A -> B) & ~B)
              by Hypothetical Syllogism.
((A -> B) & ~B) follows from
              (A -> B) and ~B
              by Conjunction.
(A -> B) is a premise.
~B is a premise.
========================================

========================================
Enter propositions (blank line to exit):
 ((O|P) -> Q)
(~Q & ~O)
Enter proposition to prove: ~(O|P)
~(O | P) follows from
              (((O | P) -> Q) & ~Q)
              by Hypothetical Syllogism.
(((O | P) -> Q) & ~Q) follows from
              ((O | P) -> Q) and ~Q
              by Conjunction.
((O | P) -> Q) is a premise.
~Q follows from
              (~Q & ~O)
              by Simplification.
(~Q & ~O) is a premise.
========================================
```

Note the curious invocation of the *Hypothetical Syllogism* rule in the last proof. This can be explained with reference to the fact that $\neg Q$ is represented internally as $Q \supset \bot$, so that Modus Tollens becomes merely a special case of Hypothetical Syllogism.