SYMBOLIC KNOWLEDGE AND NEURAL NETWORKS: INSERTION, REFINEMENT AND EXTRACTION

by
Geoffrey G. Towell

B.A., Hamilton College, 1983
M.S., University of Wisconsin — Madison, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy
(Computer Science)

at the
UNIVERSITY OF WISCONSIN — MADISON
1991

Contents

Appendices

ACKNOWLEDGEMENTS

As with any document that is a long time in the writing, there are probably more people to thank than there are pages. So, rather than trying to thank everyone, I mention only four groups of people.

The first group has but a single member, my wife Betsy. A dedicated “morning person”, her choice of hours has a significant effect both upon the content of this work and the time frame in which it was completed. She pushed me when it was important, but also had the wisdom to let me relax.

The second group consists of Mike Litzkow and everyone involved in the development of Condor [Litzkow88]. Condor is a system which allows jobs to run on any idle workstation in the department. (Even as I am writing this sentence, I have tasks executing on more than 40 machines.) Much of the work in this thesis was only possible because of Condor; in the past six months, I have used more than three CPU years of computer time through the Condor system.

The third group consists of Richard Maclin, Eric Gutstein, and Charles Squires Jr. Fellow grad students, they all took time out from their own work to review, and substantially improve, this thesis.

Finally, my advisor, Jude Shavlik and my thesis committee: Richard Lehrer (reader), Charles Dyer (reader), Leonard Uhr (nonreader) and Olvi Mangasarian (nonreader). Now that I see the light at the end of the tunnel that is my thesis, I can see that Jude consistently guided me along a course towards that light. Then, to my amazement, the committee saw that light despite my best efforts at obfuscation.

In addition to thanking individuals, this work is partially supported by Office of Naval Research Grant N00014-90-J-1941, National Science Foundation Grant IRI-9002413, Department of Energy Grant DE-FG02-91ER61129 and the University of Wisconsin Graduate School.

ABSTRACT

Explanation-based and empirical learning are two largely complementary methods of machine learning. These approaches to machine learning both have serious problems which preclude their being a general purpose learning method. However, a "hybrid" learning method that combines explanation-based with empirical learning may be able to use the strengths of one learning method to address the weaknesses of the other method. Hence, a system that effectively combines the two approaches to learning can be expected to be superior to either approach in isolation. This thesis describes a hybrid system called KBANN which is shown to be an effective combination of these two learning methods.

KBANN *(Knowledge-Based Artificial Neural Networks)* is a three-part hybrid learning system built on top of "neural" learning techniques. The first part uses a set of approximately-correct rules to determine the structure and initial link weights of an artificial neural network, thereby making the rules accessible for modification by neural learning. The second part of KBANN modifies the resulting network using essentially standard neural learning techniques. The third part of KBANN extracts refined rules from trained networks.

KBANN is evaluated by empirical tests in the domain of molecular biology. Networks created by KBANN are shown to be superior, in terms of their ability to correctly classify unseen examples, to a wide variety of learning systems as well as techniques proposed by experts in the problems investigated. In addition, empirical tests show that KBANN is robust to errors in the initial rules and insensitive to problems resulting from the presence of extraneous input features.

The third part of KBANN, which extracts rules from trained networks, addresses a significant problem in the use of neural networks — understanding what a neural network learns. Empirical tests of the proposed rule-extraction method show that it simplifies understanding of trained networks by reducing the number of: consequents (hidden units), antecedents (weighted links), and possible antecedent weights. Surprisingly, the extracted rules are often more accurate at classifying examples not seen during training than the trained network from which they came.

Chapter 1

INTRODUCTION

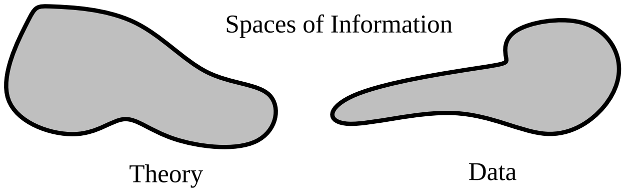
Suppose you are tying to teach someone who has never seen some class of objects (e.g., cups) to recognize the members of that class. One approach is to tell your student everything about the category. That is, you could state a "domain theory" [1] that describes how to recognize individual, critical components of the class members and how those components interact. Using this domain theory of a class of objects, your student could then distinguish between members and nonmembers of the class. For instance, in teaching someone to recognize cups, you might have that person learn: what a handle is, how material that a cup of made of affects its properties, and the utility of the picture on the side of some cups (i.e., none). Using this information, the person could determine what is, and is not, a cup.

A different approach to teaching a person to recognize a class of objects is to show the person lots of examples. As each example is shown, you would tell your student whether the example is, or is not a member of the class, and nothing else. After seeing sufficient examples, your student could classify new examples by comparison to those already seen.

A more reasonable way of teaching is to combine these two approaches. That is, teach your student to recognize cups by showing examples and providing a domain theory. The student could then use examples to fill gaps in the theory and theory to fill gaps in the set of examples.

The first two methods of teaching roughly characterize the two main approaches to machine learning: *explanation-based learning* [DeJong86, Mitchell86] and *empirical learning* [Michalski83, Quinlan86, Mitchell82, Rumelhart86, Holland86b]. Explanation-based learning corresponds to teaching by giving a person a domain theory without an extensive set of examples. Conversely, empirical learning involves giving a person lots of examples without any explanation of why the examples are members of a particular class. Unfortunately, for reasons described in the following section, neither of these approaches to machine learning is completely satisfactory.

[1] A *domain theory* is a collection of rules that describes the interactions of facts in a system. For classification problems, a domain theory can be used to prove whether or not an object is a member of the class in question.



Blobs indicate the complete space of information.

Fill represents the amount of information avaliable within that space. So, a completely filled blob indicates complete infomation while an empty blob indicates no information.

Figure 1.1: The two spaces of information.

They each suffer from flaws that preclude either from being a general method of learning.

The flaws of each learning method are, for the most part, complementary. Hence, a "hybrid" system that effectively combines learning from theory with learning from data might be like the hypothetical student taught using the combination of theoretical information and examples. The student, given both kinds of information, would have been able to combine it such that the whole was greater than the sum of its parts. Similarly, a hybrid learning system can be expected to find synergies that make it more effective that either purely empirical or explanation-based learning systems.

The KBANN *(Knowledge-Based Artificial Neural Networks)* system described and evaluated in this thesis is such a system. Briefly, the approach taken by KBANN is to *insert* a set of symbolic rules into a neural network. The network is then *refined* using standard neural learning algorithms and a set of classified training examples. Finally, symbolic rules that reflect modifications to the initial rules as a result of neural learning are *extracted* from the network.

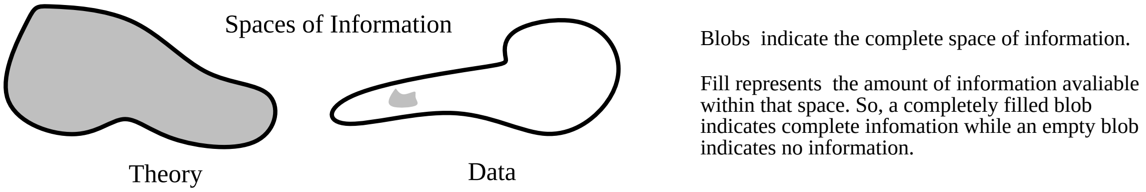
Empirical tests in Chapter 3 show that KBANN benefits from its combination of explanation-based and empirical learning. The result is that KBANN is better at classifying examples not seen during training than methods that learn purely from examples as well as methods which, like KBANN, learn from both theory and examples (see Section 3.2).

1.1 Learning from Theory and Data

Consider dividing information about a given task into two non-overlapping parts – theory and data – as represented by Figure 1.1. The theory part contains only rules that describe how pieces of information fit together. Thus, the theory part might contain rules that define what it is to be a cup. The data section contains only instances of objects labeled by their category. Hence, it might hold lots of examples of cups and non-cups. This theory/data split is exactly the division assumed by explanation-based and empirical learning systems.

1.1.1 Explanation-based learning

The information used by explanation-based learning (EBL) systems is shown in Figure 1.2. That is, EBL systems learn almost exclusively from theoretical knowledge of a problem —



Blobs indicate the complete space of information.

Fill represents the amount of information avaliable within that space. So, a completely filled blob indicates complete infomation while an empty blob indicates no information.

Figure 1.2: Information used by explanation-based learning systems.

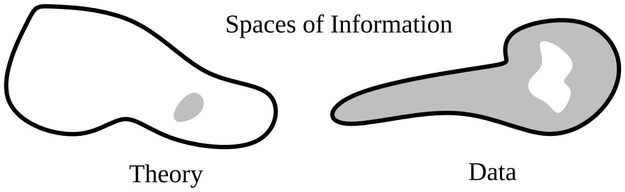
knowledge is assumed to be complete and correct. (This assumption is made only by basic EBL algorithms.) The principle advantage of EBL is that it requires very little data; often a single example is sufficient for learning. However, EBL systems are fraught with difficulties, of which the following is a partial list:

1. *The basic EBL algorithms assume that the domain theory is both complete and correct* [Mitchell86]. Thus, anything not meeting the definition of a cup is not a cup and anything meeting the definition must be a cup. However, writing domain theories is very difficult because the task often requires formalizing "functional categories" (i.e., categories defined only in terms of their functions [Brunner56]). Furthermore, in many domains writing complete and correct domain theories may not be merely difficult, but impossible [Bennett88, Rajamoney87]. For instance, it is impossible to make a perfect model of motion for a robot because the world constantly changes.

2. *Basic EBL systems do not learn at the "knowledge level"* [Dietterich86]. The knowledge level is an abstraction which captures everything an entity knows how to do. (I.e., the knowledge level encompasses the deductive closure of an entity's knowledge.) Hence, when an entity learns a new way to represent something that it already knew, it has not learned at the knowledge level. The implication of this for EBL systems is that they cannot correct mistakes in their initial theory. So, they are forever doomed to repeat their mistakes.

3. *Domain theories can be "brittle"* [Holland86a]. That is, they only apply to a very narrow domain with very sharp boundaries. As a result, correct application of knowledge does not gracefully degrade as the boundaries are crossed, but drops immediately to zero. Unfortunately, it can be difficult to make EBL systems aware of having crossed the boundaries of their knowledge. As a result, systems that are normally reliable may provide incorrect answers without warning.

4. *Domain theories can be intractable to use* [Mitchell86, Rajamoney87]. Constructing proofs (which is essentially how EBL systems make decisions) using a domain theory may exceed time and/or memory bounds of the computer. Thus, an EBL system may



Blobs indicate the complete space of information.

Fill represents the amount of information avaliable within that space. So, a completely filled blob indicates complete infomation while an empty blob indicates no information.

Figure 1.3: Knowledge assumed by empirical learning systems.

be unable to provide an answer despite having the knowledge to correctly answer the question at hand. For instance, given a board position and a proposed move, an EBL system for chess could potentially determine if is that move is optimal. However, doing so is computationally impossible.

5. *Domain theories must be supplied by some outside agency.* The creation of a domain theory suitable for use in an EBL system is itself, a significant learning problem [Pazzani88]. More generally, domain theory construction is one of the significant limiting factors in the development of "expert systems" [Waterman86].

1.1.2 Empirical learning

Empirical learning (EL) systems learn by generalizing from examples. Thus, as represented by Figure 1.3 they require little theoretical knowledge; instead they require a large, but possibly incomplete, library of examples. Their complete ignorance of theory has several disadvantages. Some of the most significant are:

1. *Spurious correlations in the examples can lead to incorrect classifications.* For instance, if all of the cups a learner has ever seen are colored red, it might conclude that all cups must be colored red. This is the classic problem of induction [Goodman83].

2. *Even when a large set of examples are available, small sets of exceptions may be either unrepresented or very poorly represented.* As a result, datasets with many "small disjuncts" tend to be poorly handled by EL systems [Holte89, Rendell90]. (Techniques are being developed to help empirical learning systems handle small disjuncts [Quinlan91].)

3. *Features relevant to classification are context dependent* [Schank86]. For example, that a $10 bill is made of flammable paper is unimportant when the bill is in your hand, but is quite significant when it is in a bank that is on fire.

4. *An unbounded number of features can be used to describe any object.* As a result, any two objects can appear similar or different by considering appropriate sets of features [Wantanabe69, the theorem of the ugly duckling]. This point implies that, at a minimum,

empirical learning systems require knowledge that indicates the relevant features. So, the small blob in the theory space of Figure 1.3 represents knowledge of the features that are probably necessary and sufficient for learning the solution to a problem.

5. *Irrelevant features in the description of examples can negatively affect learning* [Breiman84]. For example, consider the problem of seeing only red cups discussed above in the context of spurious correlations. The color of a cup is irrelevant, but spurious correlations between positive examples of cups and their color make red appear to be relevant. The presence of some irrelevant features in the description of an example is generally unavoidable since the set of relevant features may not be known. However, as the number of irrelevant features increases, so does the probability that there will be a spurious correlation making an irrelevant feature appear relevant.

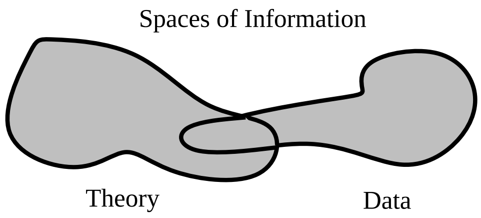
6. *Complex features that can be constructed from the initial features may considerably simplify learning* [Rendell90]. EL systems must either independently discover these derived features or do without them. As most EL systems operate using some sort of hill-climbing heuristic, they may be unable to discover derived features that are only valuable when fully constructed.

1.1.3 Artificial neural networks

Artificial neural networks (ANNs), which form the basis of KBANN, are a particular method for empirical learning. ANNs have proven to be equal, or superior, to other empirical learning systems over a wide range of domains, when evaluated in terms of their ability to correctly classify examples not seen during training (i.e., their ability to generalize) [Shavlik91, Fisher89, Weiss89, Atlas89, Dietterich90, Tsoi90, Ng90]. However, they have a set of problems unique to their style of empirical learning. Among these problems are:

1. *Training times are lengthy.* Experiments [Shavlik91, Fisher89, Weiss89] indicate that ANNs require 100 to 1000 times as much training time as some symbolic learning algorithms (e.g., ID3 [Quinlan86]). That is, ANNs learn with the ponderous grace of a glacier.

2. *The initial parameters of the network can greatly effect how well concepts are learned* [Ahmad88, Kolen90]. Because training ANNs usually involves a hill-climbing algorithm, local minima may be a problem. For example, in some experiments involving the conversion of text to speech, correctness on a large dictionary of examples ranged from 1% to 31% [Shavlik91]. The only difference between the runs was the randomly-chosen initial set of weights.



Blobs indicate the complete space of information.

Fill represents the amount of information avaliable within that space. So, a completely filled blob indicates complete infomation while an empty blob indicates no information.

Figure 1.4: Interactions between the two spaces of information.

3. *There is not yet a problem-independent way to choose a good network topology.* The performance of an ANN can be greatly affected by the layout and number of hidden units, as well as the specification of connectivity. Many researchers have suggested methods for addressing this problem [Honavar88, Diederich88, Chauvin88, Hanson88, Kruschke88, Fahlman89]. However, no method eliminates it.

4. *After training, a neural network is normally used as a "black box".* That is, given input, it produces output, but there is no explanation how they relate. Yet, without the ability to explain their decisions, it is hard to be confident in the reliability of a network that addresses a real-world problem. Moreover, the extraction of accurate, comprehensible, symbolic knowledge from networks is important if the results of neural learning are to used in related problems. Work has only just begun on the problem of shining light into the black box that is a trained neural network [Saito88, Hanson90, Fu91, Towell91].

1.2 Thesis Statement

Two of the disadvantages of empirical learning (i.e., that feature relevance is context dependent and that an unbounded number of features can be used to describe any object) point towards the conclusion that the distinction between theory and data is largely artificial. Theory and data are not independent as is suggested by Figure 1.1. Rather, as pictured in Figure 1.4, theory and data overlap. At a minimum, examples affect theory by pointing to its relevant parts (as well as identifying inconsistent parts). Likewise, theory affects examples by pointing out some of their relevant features.

Therefore, systems that make a hard distinction between theory and data are, in some sense, bankrupt. The distinction does not, and cannot, exist in the manner required by either explanation-based or empirical learning systems. *Instead learning systems must be able to learn from both theory and data.*

Moreover, the problems of explanation-based and empirical learning systems are complementary. *Hence, a hybrid learning system that is able to use strengths of one system to offset the weaknesses of the other may prove to be a powerful learning method.* Table 1.1 highlights the complementary nature of the weaknesses of explanation-based and empirical learning systems. This table lists each of the weaknesses of empirical and explanation-based learning discussed previously. With each item in the list is a brief description of why the other learning method is unaffected by the weakness.

The preceding discussion of the complementary nature of theory and data – as well as explanation-based and empirical learning – is the basis for the following:

> **Thesis:** *A system capable of learning effectively and efficiently must be able to draw on the available knowledge from both theoretical understanding of problems as well as examples of solved problems. A good approach to building such a system is to combine explanation-based learning with empirical learning. The resulting "hybrid" learning system should be able to profitably learn from theory and data; it should be able to use the strengths of one learning mechanism to overcome the weaknesses of the other.*

1.3 Review of Learning in Neural Networks

This section is a brief digression into a review of artificial neural networks; they are the basis of KBANN. Readers familiar with neural networks may skip this section without trepidation.

Artificial neural networks (ANNs) are a class of learning systems inspired by the architecture of the brain and theories of learning in the brain [Rosenblatt62, Hebb49, McCulloch43]. They are composed of cell-like entities (referred to as *units*) and connections between the units corresponding to dendrites and axons. (The connections are referred to as *links.*) Links are *weighted*; hence, the signal each link carries is the product of the link weight and the signal sent through the link. Abandoning the brain metaphor, an ANN can be thought of as a directed graph with weighted connections.

Units do only one thing – they compute a real-numbered output (known as an *activation*) which is a function of real-numbered inputs. Inputs either come from the environment or are received on links from other units. For instance, Figure 1.5 shows a single unit with three incoming links. The activation of the unit is shown to be a function of the sum of the incoming signals.

Units in neural networks are commonly sorted into three groups: *input*, *hidden* and *output.* These groups are shown graphically in Figure 1.6. Input units are so named because they receive signals from the environment. Similarly, output units are so named because their activation is available to the environment. (Generally the activation of the output units is the answer computed by the network.) Finally hidden units are so named because they have no direct interaction with the environment. They are purely internal to the network.

Table 1.1: Complementary strengths and weaknesses of learning systems.

Weaknesses of EBL are offset by strengths of EL.

- Basic EBL systems require a complete and correct theory,
  but EL requires little or no theory.
- Basic EBL systems do not learn at the "knowledge level",
  but EL systems do.
- Rules can be "brittle" in application,
  but EL systems such as neural networks gracefully degrade,
- EBL domain theories may be intractable to use,
  but the results of EL learning can often take little time to use.
- EBL systems must be provided with a knowledge base. However, they are incapable of its origination,
  but EL systems require little or no domain knowledge. Also, EL systems may be used to create domain knowledge.

Weaknesses of EL are offset by strengths of EBL.

- Spurious correlations may reduce the accuracy of empirical learning,
  but EBL systems do not rely upon correlations in the data.
- Small disjuncts can be difficult or impossible to learn accurately,
  but domain knowledge can specify arbitrarily small subsets.
- Contextual dependency of features is difficult to capture,
  but context may be a part of the domain knowledge.
- Every object can be described using an unbounded number of features,
  but domain knowledge may specify relevant features.
- Irrelevant features can interfere with learning,
  but domain knowledge may specify relevant features.
- Derived features, necessary for correctly acquiring a concept, may be impossible to learn,
  but intermediate conclusions in rules can specify arbitrarily complex derived features.

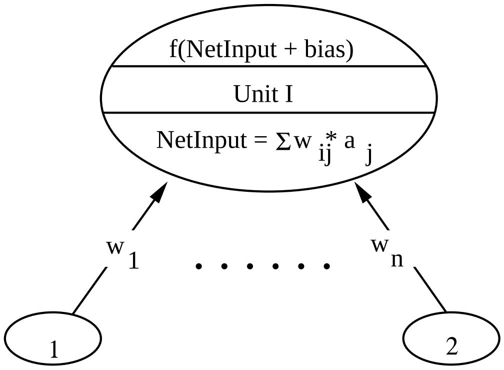


Figure 1.5: A single unit of an ANN.

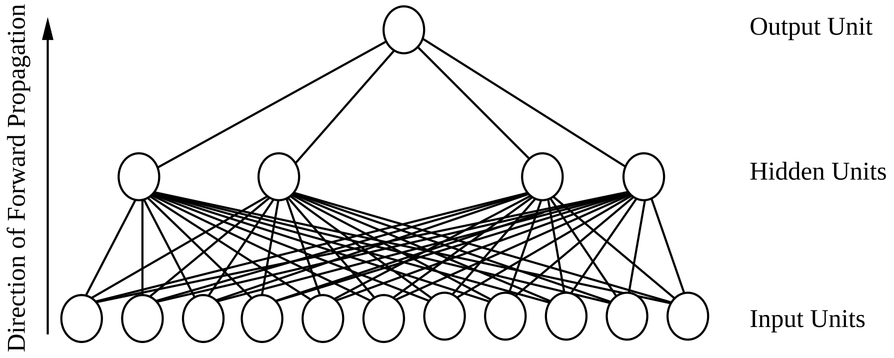


Figure 1.6: A prototypical artificial neural network.

Table 1.2: The backpropagation algorithm.

1. Activations of input units are set by the environment;
2. Activation is propagated forward along the directed connections (links) possibly through hidden units to the output units;
3. Errors are determined as a function of the difference between the computed activation of the output units and the desired activation of the output units;
4. Errors are propagated backward along the same links used to carry activations;
5. Changes are made to link weights to reduce the difference between the actual and desired activations of the output units.

The general process of learning in a neural network, using the standard backpropagation model [Rumelhart86], is given by the five-step algorithm in Table 1.2. The equations that underlie this algorithm appear in Table 1.3. These equations define how activations are computed, errors are propagated and weighted are changed following the standard approach of backpropagation.

Equation 1.2 defines the activation function for units in a backpropagation network. As illustrated in Figure 1.7, this standard "sigmoid" function squashes net incoming activation to a unit (Equation 1.1), which ranges over $[-\infty, +\infty]$, into the range [0, 1]. Note that the effect of θ_i, the "bias" term, in Equation 1.1 is to increase (or decrease) the net input to a unit. One way of looking at this is that the bias shifts the position of the sloping part of the curve in Figure 1.7. (In this case, the X-axis would be only the weighted sum of the incoming activations.) Hence, the bias acts like a threshold for the activation function.

Equations 1.3–1.6 define the learning mechanism of neural networks. Intuitively, this

Table 1.3: The mathematics underlying backpropagation.

$$NetInput_i = \sum_j w_{ji} * a_j + \theta_i \tag{1.1}$$

$$a_i = \frac{1}{1 + e^{-NetInput_i}} \tag{1.2}$$

$$E = \frac{1}{2}\sum_{i=1}^{n}(d_i - a_i)^2 \tag{1.3}$$

$$\begin{aligned} eOutput_i &= -\frac{\partial E}{\partial NetInput_i} \\ &= -\frac{\partial E}{\partial A} * \frac{\partial A}{\partial NetInput_i} \\ &= [d_i - a_i] * [(1 - a_i) * a_i] \end{aligned} \tag{1.4}$$

$$\begin{aligned} eHidden_i &= \frac{\partial A}{\partial NetInput_i} * \sum_k e_k * w_{ik} \\ &= [a_i * (1 - a_i)] * \sum_k e_j * w_{ik} \end{aligned} \tag{1.5}$$

$$\Delta w_{ij}(t) = \alpha * e_j * a_i + \mu * \Delta w_{ij}(t-1) \tag{1.6}$$

where:	i	is the index number of a unit
	$NetInput_i$	is the net incoming activation to unit i
	j	is a index ranging over every unit from which unit i receives activation
	w_{ij}	is the weight on a connection from unit j to unit i
	θ_i	is the threshold term associated with unit i
	A(x)	is the activation function
	a_i	is the activation of unit i
	E	is the total error of the network
	n	is the number of output units
	d_i	is the desired activation (training signal) for unit i
	$eOutput_i$	is the error of unit i, when the units is an output unit
	$eHidden_i$	is the error of unit i, when the units is a hidden unit
	k	is an index ranging over units to which unit i sends activation
	α	is the learning rate, a real number typically from [0. . . 1]
	μ	is the momentum term, a real number typically from [0. . . 1]
	t	is a time index

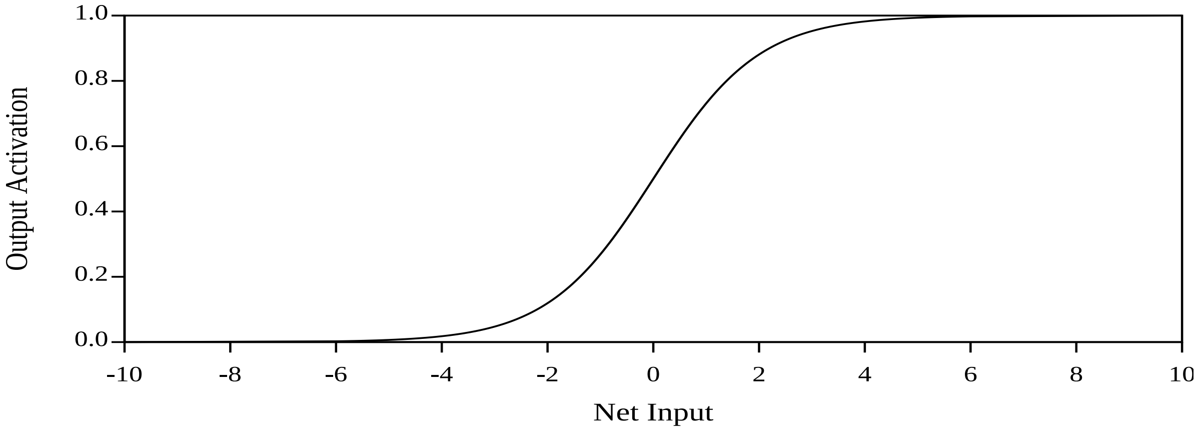


Figure 1.7: Output from a unit as a function of its net input.

method works by determining where the network went wrong, and making changes to address the mistake. Equation 1.3 defines the total error of the network as the sum of the squared differences between the actual and desired activations of the output units. Defining error as a sum of squared differences is common in statistics. For example, this measure underlies standard deviation.

Equations 1.4 and 1.5 define the propagation of error through a network. Equation 1.4 sets the error at an output unit. This function, the partial derivative of the error with respect to the net input, can be computed by applying the chain rule to Equations 1.2 and 1.3. Equation 1.5 defines the error at each hidden unit to be a function of the activation of that unit and the errors of all the units to which that unit sends activation. So, Equation 1.5 is the mechanism whereby credit or blame is assigned to units in the network. In other words, Equation 1.5 defines the way backpropagation addresses the "credit assignment problem" [Minsky63]. (The credit assignment problem is a classic problem of determining who should be rewarded for taking a correct action, or blamed for taking an incorrect action.)

Finally, Equation 1.6 shows that the adjustment of a link's weight is dependent upon the error at the receiving end of the link and the activation at the sending end of the link. In addition, this equation contains a "momentum" term that is dependent upon the previous change in the link weight. The momentum term is intended to damp out oscillations in the weight adjustments.

1.4 Overview of this Thesis

The rest of this thesis describes KBANN, a system that demonstrates the hypothesis that systems can learn from both theory and data by combining aspects of explanation-based and empirical learning systems. Chapter 2 contains a complete specification of KBANN. Briefly, KBANN has three, largely separate parts: a mechanism for inserting domain knowledge into a neural network, a mechanism for refining the network, and a mechanism for extracting refined rules from a trained network. In combination, these three parts form a powerful and general learning program.

Two real-world problems from the area of molecular biology are used to test the generality of learning in KBANN. (See Section 3.1 for their descriptions.) These problems are a small subset of the growing number of problems in computational biology due to the Human Genome Project [Alberts88]. In addition, KBANN has been used as the basis of a psychological model of the development of geometric reasoning in children (see Chapter 4).

The empirical results presented in Section 3.2 verify the hypothesis that a system that learns from both theory and data can learn more effectively than a system that learns from data alone. Further tests show KBANN makes more effective use of the combination of theory and data than other hybrid systems.

Tests in the remainder of Chapter 3 characterize the conditions under which it is useful to learn from both theory and data. For example, one set of tests adds noise to theories to determine the limits on how poor a domain theory can be while still providing useful information. A second set of tests investigates the properties of data sets which make theoretical information useful.

Chapter 4 continues the testing of KBANN, but in a very different way than the previous chapter. Rather than evaluating KBANN as a classifier, this chapter also describes the use of KBANN as the basis of a model of the development of geometric reasoning in children.

Chapter 5 describes two enhancements to KBANN. The first extension is the DAID preprocessor, a method for determining slight adjustments to the weight of links in a network created by KBANN. These adjustments, based upon the errors that the domain knowledge makes on a set of training examples are shown to both improve generalization and reduce training effort. The second extension to KBANN described in Chapter 5 is the ability to add hidden units not specified by the normal KBANN network construction method. These hidden units provide KBANN with the ability to grow beyond the vocabulary defined by the initial domain knowledge.

Overall, KBANN supports the thesis that a system which learns from both theory and data can outperform a system that learns from theory or data alone. The tests reported in this thesis show that this hypothesis is true under a wide variety of conditions on three different problems. Moreover, KBANN is shown to be an effective learning system by comparison to other hybrid approaches. Hence KBANN is a powerful and general approach to machine learning that empirically verifies this work's thesis.

Chapter 2

KBANN

This chapter describes the organization and algorithms of KBANN. Figure 2.1 depicts KBANN as a series of three algorithms. The first algorithm *inserts* approximately-correct, symbolic rules into a neural network. This step creates networks that make the same responses as the rule upon which they are based. Details of this first algorithm are given in Section 2.2.

The second algorithm of KBANN *refines* networks using the backpropagation learning algorithm [Rumelhart86]. While the learning mechanism is essentially standard backpropagation, the network it operates on is not standard. Instead, the network is constructed by the first algorithm of KBANN and thus, makes the same responses as the provided domain knowledge. This property of the networks created by KBANN has implications for training which are discussed in Section 2.3.

The final algorithm of KBANN *extracts* refined rules from trained networks. This algorithm makes it possible to explain the responses of trained networks.As a result, the information learned by the network is made available for human review. Moreover, the extracted rules can

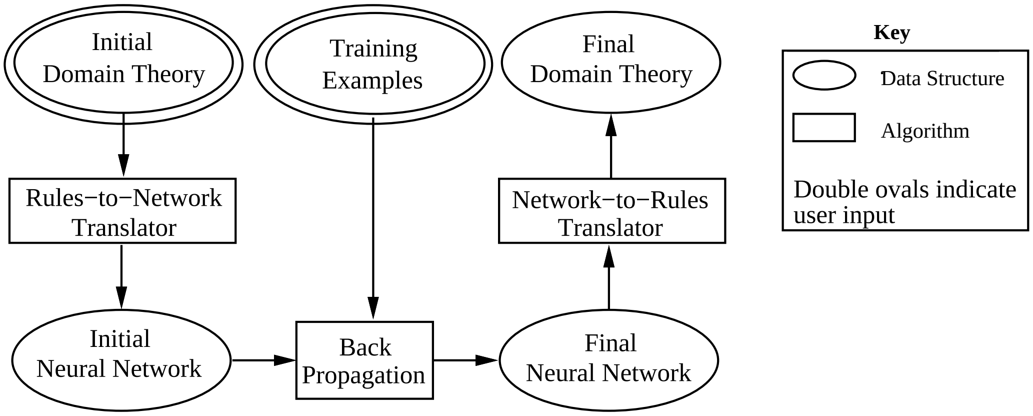


Figure 2.1: The flow of information through KBANN**.**

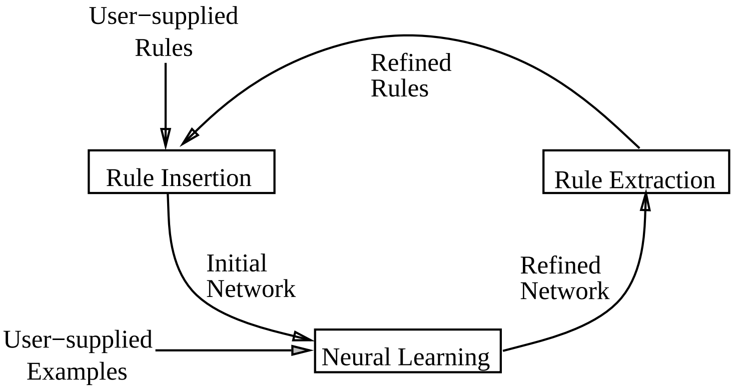


Figure 2.2: Information feeds back upon itself in KBANN.

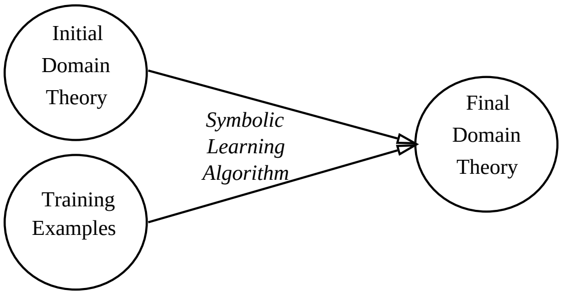


Figure 2.3: Flow chart of "all-symbolic" theory-refinement.

be used as a part of the initial knowledge for related problems. Hence, this algorithm allow KBANN to feed results back to itself in much the manner illustrated by Figure 2.2.

Before continuing with the description of KBANN, consider the difference between Figures 2.1 and 2.3. These figures present two alternative architectures for systems that learn form both theory and examples. As described above, Figure 2.1 shows the tri-algorithm architecture of KBANN. By contrast, Figure 2.3 represents the architecture is of EITHER [Ourston90] and Labyrinth-k [Thompson91], two "all-symbolic" hybrid learning systems. Whereas KBANN requires three algorithms, these all-symbolic systems require only a single algorithm because their underlying empirical learning mechanism operates directly upon the rules rather than their re-representation as a neural network. (See Chapter 6 for a descriptions of EITHER, Labyrinth-k, and other hybrid learning systems.) Tests reported in Chapter 3 show that this extra effort is well rewarded, as KBANN consistently outperforms these all-symbolic systems.

The next section presents a high-level overview of KBANN which gives a feel for the approach. The subsequent three sections contain in-depth descriptions of each of the three algorithmic steps of the KBANN.

Table 2.1: Correspondences between knowledge-bases and neural networks.

Knowledge Base	*Neural Network*
Final Conclusions	Output Units
Supporting Facts	Input Units
Intermediate Conclusions	Hidden Units
Dependencies	Weighted Connections

2.1 Overview of KBANN

As shown in Figure 2.1, KBANN consists of three largely independent modules: a rules-to-network translator, a refiner, and a network-to-rules translator. Briefly, rules-to-networks translation is accomplished by mapping between a rule set and a neural network. This mapping, specified by Table 2.1, defines the topology of networks created by KBANN (KBANN-nets) as well as its initial link weights (see Section 2.2).

By defining KBANN-nets in this way, many of the problems inherent to neural networks and empirical learning are either eliminated or significantly reduced. For example, rules identify features that are very likely to be relevant. Hence, problems of spurious correlations and irrelevant features are significantly reduced. (See Section 3.4 for empirical evidence in support of this contention.) In addition, as KBANN admits rule sets that are not perfectly correct, it addresses several of the problems of explanation-based learning (EBL). For example, complete and correct rule sets are not required as the networks created by KBANN can (and do) learn at the knowledge level. This procedure also indirectly addresses other problems of EBL such as intractable theories because approximately correct theories are often quite brief.

The second major step of KBANN is to refine the KBANN-net using standard neural learning algorithms and a set of classified training examples. At the completion of this step, the trained KBANN-net can be used as a classifier that is more accurate than those derived by other machine learning methods. (Section 3.2 contains empirical evidence that supports this contention.)

The final, network-to-rules translation step of KBANN extracts a set of symbolic rules from the trained KBANN-net that retains its accuracy. This part of KBANN, described in Section 2.4, directly addresses problems that occur because trained neural networks are effectively "black boxes". Rule extraction shines light into the box, a process with three principle benefits: (1) the information learned by the KBANN-net during training is accessible for human review; (2) the rules can be used to give a semantically meaningful explanation for the responses of the network; (3) the modified rules can be used as part of knowledge bases of related problems. Hence, the extraction of rules allows information transfer to other neural networks [Pratt91] as well as other, symbolically-oriented learning systems.

2.2 Inserting Knowledge into a Neural Network

The first step of KBANN is to translate a set of approximately-correct rules into a knowledge-based neural network (KBANN-net). This translation has several important benefits. First, the translation specifies the features that are probably relevant to making a correct decision. Feature specification addresses problems inherent to empirical learning such as spurious correlations, irrelevant features, and the unboundedness of the set of possible features. Second, the translation of rules can specify important "derived" features, thereby simplifying the learning problem [Rendell90]. Moreover, these derived features can capture contextual dependencies in an example's description. Finally, the translated rules can refer to arbitrarily small regions of feature space. Hence, the rules reduce the need for the empirical learning system to learn about "small disjuncts." (Section 1.1 contains a definition of this problem.)

Rules to be translated into KBANN-nets are expressed as Horn clauses using the notation described in Appendix A. There are two constraints on the rule set. First, the rules must be propositional. This constraint results from the use of neural learning algorithms which are, at present, unable to handle variables. Second, the rules must be acyclic (i.e., no rule, or combination of rules, in which the consequent of the rule is an antecedent of that rule). This constraint simplifies the translation of rules and training of the resulting networks. However, the constraint does not represent a fundamental limitation on KBANN as there exist algorithms based upon backpropagation which can be used to train networks with cycles [Pineda87]. (Section 7.2 describes plans for relaxing these constraints.)

In addition to these constraints, rule sets are usually hierarchically structured. That is, some rules do not map directly form inputs to outputs. Rather, the rules provide intermediate conclusions that describe useful conjunctions of the input features. These intermediate conclusions may be used by other rules to either determine the final conclusion or other intermediate conclusions. It is the hierarchical structure of a set of rules that creates derived features for use by the empirical learning system. Hence, if the domain knowledge is not hierarchically structured, then the networks created by KBANN will have no derived features to specify contextual dependencies or other useful conjunctions within example descriptions.

The rules-to-network translator is described in the next three subsections. The first of these subsections provides a high-level description of the translation; the second contains an example of the translation process; and the third contains a set of mathematical proofs of the correctness of the translator. Appendix A contains a complete specification of the information accepted by the translator along with descriptions of how information is translated into a network.

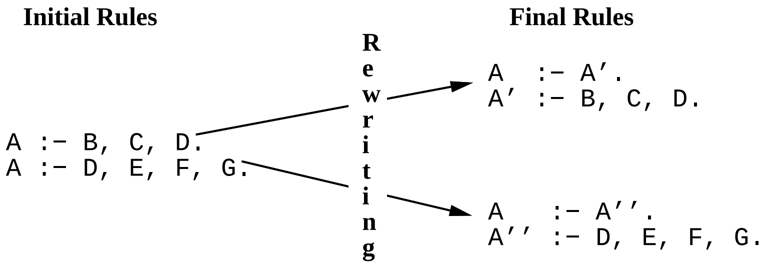


Figure 2.4: Rewriting rules to eliminate disjucts with more than one term.

2.2.1 The rules-to-network algorithm

Table 2.2 is an abstract specification of the rules-to-network translation algorithm. This algorithm initially translates a set of rules into a neural network. It then augments the network so that it is able to learn concepts not provided by the initial rules.

Table 2.2: The rules-to-networks algorithm of KBANN.

1. Rewrite rules so that disjuncts are in rules which have only one literal on the right-hand side.
2. Map the rule structure into a neural network.
3. Label units in the KBANN-net according to their "level".
4. Add hidden units to the network at user-specified levels. (*optional*)
5. Add units for known features not used in the rules.
6. Add links not specified by translation between all units in topologically-contiguous levels.
7. Perturb the network by adding near-zero random numbers to all link weights and biases.

Step 1, rewriting. The first step of the algorithm transforms the set of rules into a format which clarifies its hierarchical structure and makes it possible to directly translate the rules into a neural network. This step involves scanning every rule with the same consequent to determine if any have more than one antecedent. (The only form of disjuncts allowed by KBANN is multiple rules with the same consequent.) If there is more than one rule to a consequent, then every rule with more than one antecedent, is rewritten as two rules. One of the rules has the original consequent and a single, newly-created term as an antecedent. The other rule has that same newly-created term as its consequent and the antecedents of the original rule as its antecedents. For instance, Table 2.4 shows the transformation of two rules into format required by the next steps of KBANN. (See Theorem 5 in Appendix C for a proof that this rewriting is required.)

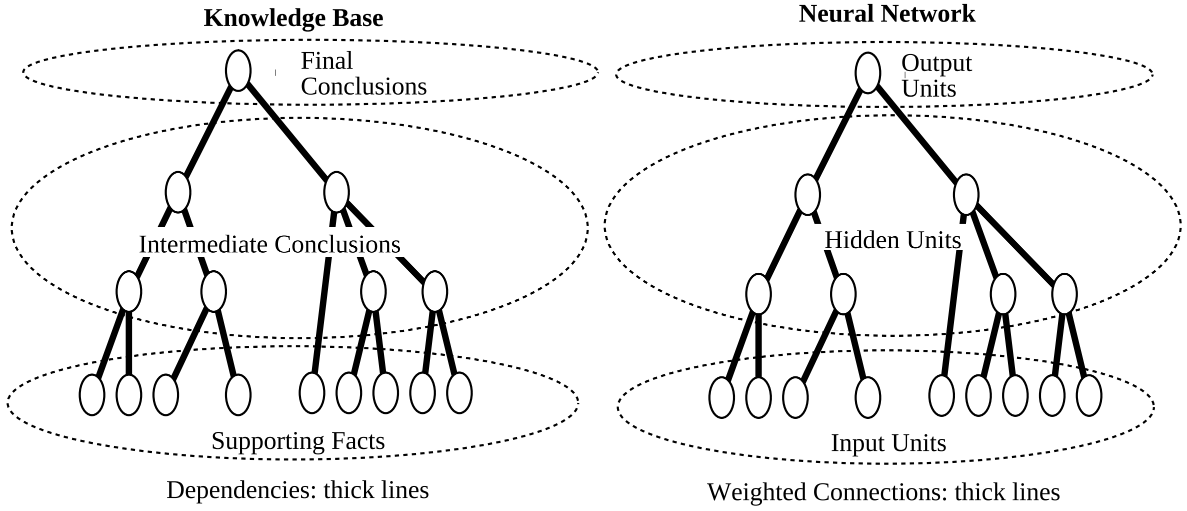


Figure 2.5: Correspondences between a rule set and a neural network.

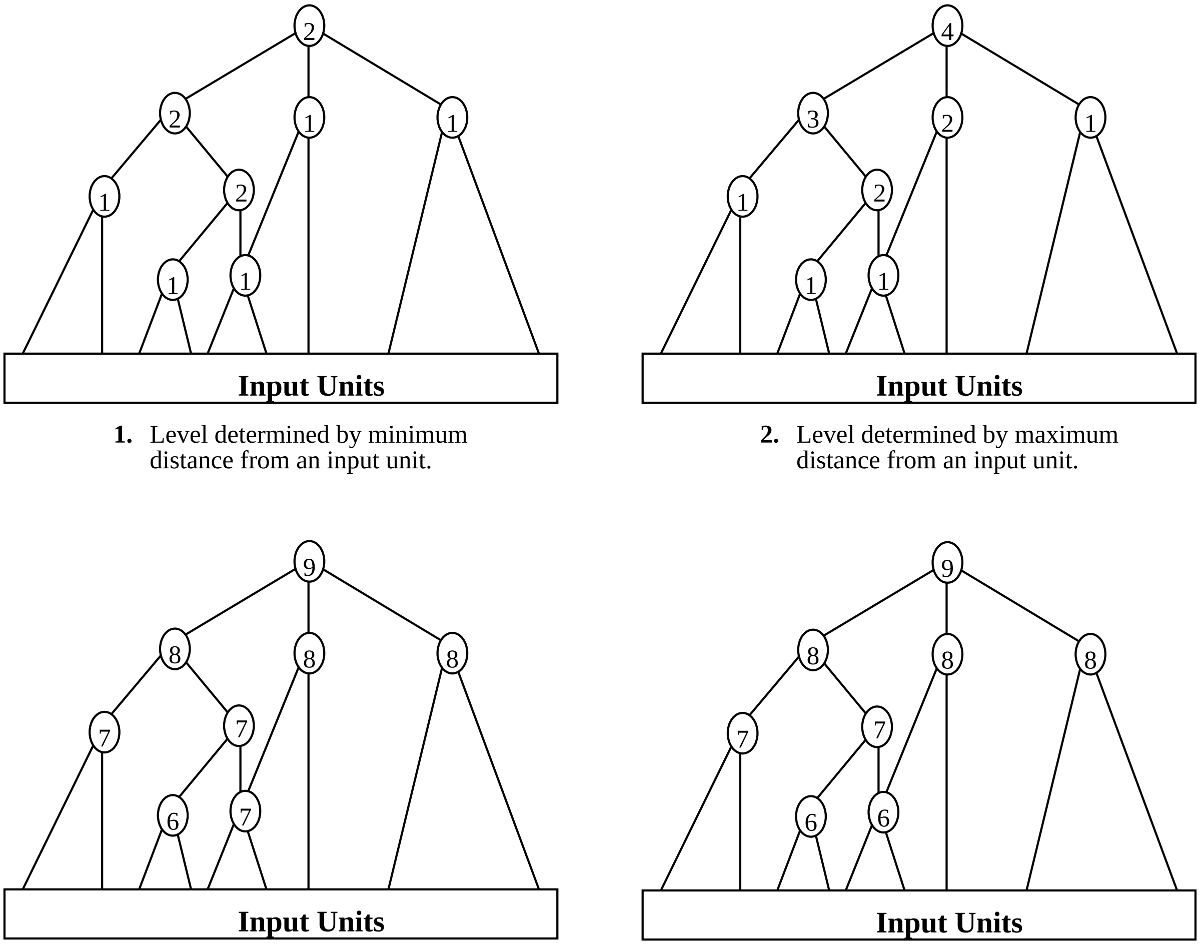
Step 2, mapping. The second step of the rules-to-network algorithm establishes a mapping between a transformed set of rules and a neural network. This mapping, shown in Figure 2.5, creates a KBANN-net which has a one-to-one correspondence with elements of the rule set. Weights on all links specified by the rule set are set to $-\omega$ or ω depending on whether the antecedent is negated or not. Biases are set to $-\frac{\omega}{2}$ for disjunctive rules and $-\frac{2N-1}{2}\omega$ otherwise (where N is the number of unnegated terms).[1] Using this scheme for setting weights, units in a KBANN-net are active (i.e., have activation near 1) only when the corresponding consequent in the rule set is satisfied. Thus, the KBANN-net mimics the responses of the set of rules upon which it is based. (Section 2.2.3 contains proofs that this method of setting weights creates networks that accurately mimic the rules upon which they are based.)

At the completion of this step, a KBANN-net has been created that has the information from the set of rules concerning relevant, and derived, features. However, there is no guarantee that the set of rules refers to all of the relevant features or provides a significant collection of derived features. Hence the next four steps expand the KBANN-net by adding links, inputs units, and possibly hidden units.

Step 3, labeling. In this step, units in the KBANN-nets are labeled by their "level". This label is not useful in itself, but is a necessary precursor to several of the following steps. Level may be defined in any of the following four ways:

1. Length of the *minimum* length path connecting a unit to an *input* unit. The upper-left graph in Figure 2.6 shows an example of this labeling method.

[1] KBANN uses $\omega = 4.0$, a number empirically found to work well.



1. Level determined by minimum distance from an input unit.

2. Level determined by maximum distance from an input unit.

3. Level determined by maximum distance from an output unit.

4. Level determined by minimum distance from an output unit.

Figure 2.6: Four methods of labeling units in a KBANN-net.

2. Length of the *maximum* length path connecting a unit to an *input* unit. This method is used in the sample translation that appears later in this chapter. The upper-right graph in Figure 2.6 shows an example of this labeling method.

3. N less the length of the *minimum* length path connecting a unit to an *output* unit. The lower-right graph in Figure 2.6 shows an example of this labeling method ($N = 9$).

4. N less length of the *maximum* length path connecting a unit to an *output* unit. The lower-left graph in Figure 2.6 shows an example of this labeling method ($N = 9$).

Each of these labeling methods results associating a number with every hidden or output unit in the network. Every labeling method gives input units the same label.

All of these labeling techniques implicitly assume that every chain of reasoning is complete; that is every intermediate conclusion is a part of a directed path from the one or more inputs to one or more outputs. However, there is no guarantee that every chain will be complete. Hence, the rules-to-network translator has the ability of recognize and handle incomplete chains. For instance, rules leading to a consequent may not reach the input units. In this case, the translator may attach the "dangling" consequents directly to every input unit (with low-weight links). Alternately, the translator may attach dangling consequents to one or more added hidden units. One the other hand, when a reasoning chain does not reach any output, the translator may attach the end of the reasoning chain directly to the output units or to some intermediate conclusions. For both types of incomplete theories, the defaults of the translator can be overridden by users.

Step 4, adding hidden units. This step adds hidden units to KBANN-nets thereby giving KBANN-nets the ability to learn derived features not specified in the initial rule set. In other words, added hidden units give KBANN-nets the ability to expand the vocabulary of the initial rules.

In many cases, the initial rules provide a vocabulary sufficient to obviate the need for adding hidden units. Hence, this step is optional. Because added hidden units are not always needed, they are only added upon specific instruction from a user. This instruction must specify the number and distribution among the levels established in the previous step of the added units.

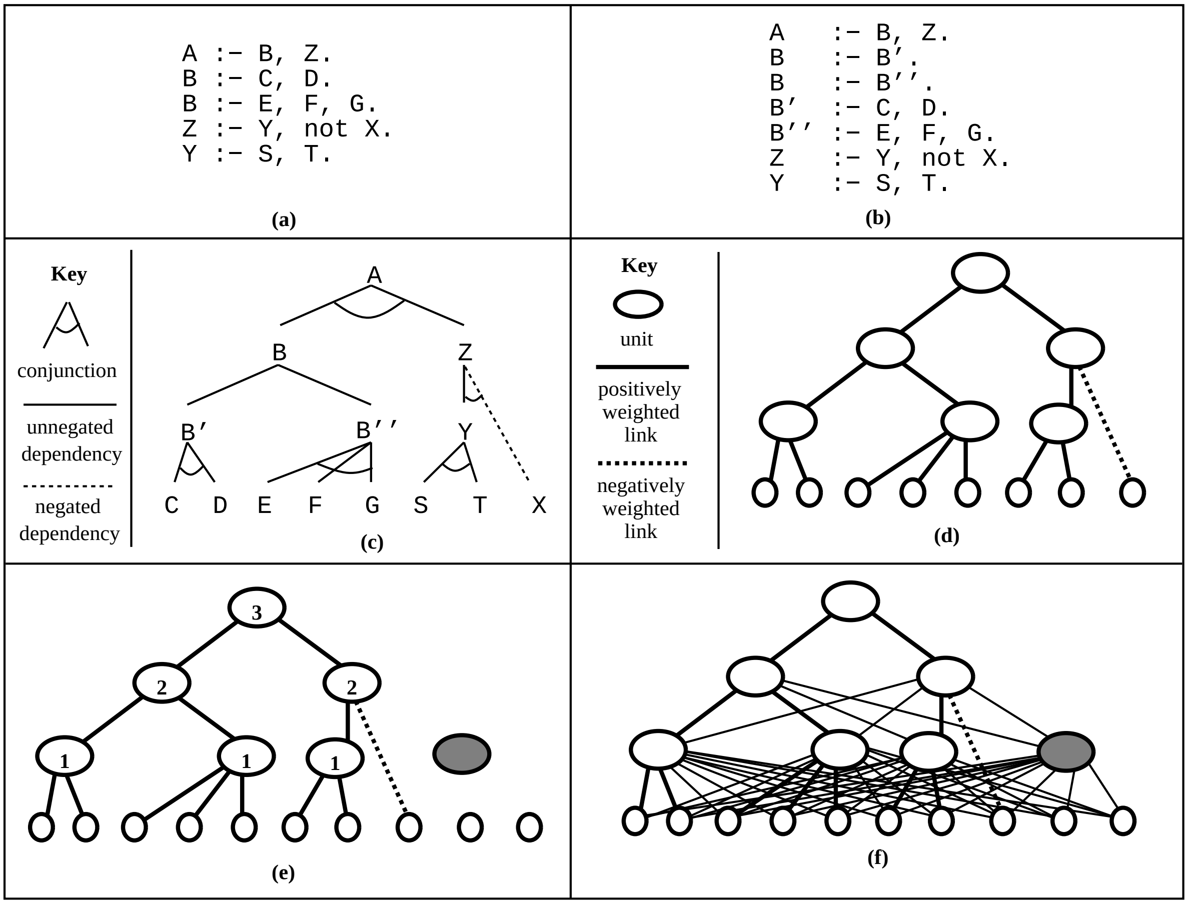
The addition of hidden units to KBANN-nets is a subject that has been only partially explored. For instance, Section 5.2 presents one method of hidden unit addition.

Step 5, adding input units. In this step, the KBANN-net is augmented with input features not referred to by the rule set. This addition is necessary because a set of rules that is only approximately correct may not identify every input feature that is required for correctly learning a concept.

Step 6, adding links. In this step links with weight zero are added to the network using the labeling of units established in step 4. Methods for adding links include:

1. Add links connecting each unit with label $n-1$ to each unit with label n.
2. Add links connecting each unit with label m to each unit with label n such that $n > m$.
3. Add links connecting every input unit to every hidden or output unit.

Adding links using the first method in conjunction with labeling according to the maximum distance form an input unit has proven slightly superior to the other methods on problems whose rule set has a good hierarchical structure. Note that neither addition of links nor addition of units have an immediate impact upon the KBANN-net because weights of the added links are so low that they do not affect the activation of any of the network's units.



```
A :- B, Z.
B :- C, D.
B :- E, F, G.
Z :- Y, not X.
Y :- S, T.
```

```
A   :- B, Z.
B   :- B′.
B   :- B′′.
B′  :- C, D.
B′′ :- E, F, G.
Z   :- Y, not X.
Y   :- S, T.
```

Figure 2.7: Sample rules-to-network translation.

Step 7, perturbing. The final step in the network-to-rules translation is to perturb all the weights in the network by adding a small random number to each. This perturbation is too small to have an effect on the KBANN-net's computations. However, it is sufficient to avoid training problems that occur in symmetrical networks [Rumelhart86].

2.2.2 Sample rules-to-network translation

Figure 2.7 shows a step-by-step translation of a simple set of rules into a KBANN-net. Panel (a) shows a set of rules in PROLOG-like notation. Panel (b) is the same set of rules after they have been rewritten in step 1 of the translation algorithm. The only rules affected by rewriting are `B :- C, D` and `B :- E, F, G`
which together form a disjunctive definition of the consequent B.

Panel (c) is a graphical representation of the rules in panel (b) that shows the hierarchical structure of the rules. In this figure, dotted lines represent negated antecedents while solid lines represent unnegated antecedents. Arcs connecting antecedents indicate conjuncts (i.e., this a standard AND/OR tree).

The next step of the translation algorithm (step 2 in Table 2.2) is to create a neural network by mapping the hierarchical structure of the rules into a network. As a result, there is little visual difference between the representations of the initial KBANN-net in panel (d) and the hierarchical structure of the rules in panel (c).

Panels (e) and (f) illustrate the process whereby links, input units and hidden units not specified in the set of rules are added to the KBANN-net (steps 3–6 of the algorithm). Panel (e) shows units in the KBANN-net labeled by their "level" where level is defined to be the maximum length path to an input unit. In addition, panel (e) shows a hidden unit (it is shaded) added to the network at level one. (For the purposes of this example, assume that the user instructed the network-to-rules translator to add a single hidden unit at level one.)

Panel (f) shows the network after links with zero weight have been added to connect all units that are separated by one level. Note that in addition to providing existing units with access to information not specified by the domain knowledge, the low-weighted links connect the added hidden units to the rest of the network.

There is no illustration of the final step of the rules-to-network translation algorithm because the perturbation of link weights results only in minute changes that never affect the decisions of the initial network.

2.2.3 Translation of rules into KBANN-nets

This section describes how KBANN translates rules containing the logical connectives AND, OR, and NOT into a KBANN-net. Recall that individual rules are assumed to be conjunctive, nonrecursive, and variable-free; disjuncts among the antecedents of rules are encoded (without loss of generality) as multiple rules.

To simplify discussion in this section, only binary-valued features are assumed to exist. Nominally-valued features are handled by simple recoding each nominal value as a binary decision. For example, consider a feature *color*, having three values *red*, *blue* and *green*. This feature would be encoded in a KBANN-net by three input units corresponding to the binary decisions: *color-is-red*, *color-is-blue*, and *color-is-green*. While this is far from the most parsimonious encoding scheme, Shavlik *et al.* suggest that this scheme improves generalization by some empirical learning algorithms [Shavlik91]. Methods for translating additional types of features into networks are presented in Appendix A.

The rules-to-network translator sets weights on the links and the biases of units so that units have significant activation (i.e., activation near one) only when the corresponding deduction can be made using the domain knowledge.[2] Likewise, when the deduction cannot be made

[2] The translation of rules into neural structures has been described by McCulloch and Pitts [McCulloch43] and Kleene [Kleene56] for units with threshold functions. Thus, the important and original contribution of this

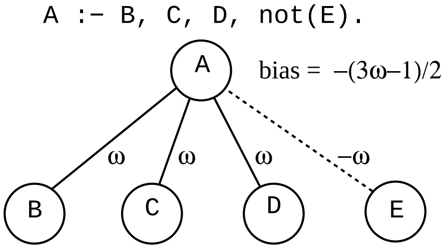


Figure 2.8: Translation of a conjunctive rule into a KBANN-net.

using the knowledge base, then the corresponding unit should be inactive (i.e., have activation near zero).

The descriptions and proofs in this section use the following terms:

C_a	the minimum activation for a unit to be considered *active*
C_i	the maximum activation for a unit to be considered *inactive*
$\mathcal{F}(x)$	$1/(1+e^{-x})$, the standard logistic activation function
w_p	the weight on links corresponding to positive dependencies
w_n	the weight on links corresponding to negated dependencies
P	the number of positive antecedents to the rule
N	the number of negated antecedents to the rule
K	the total number of antecedents to a rule ($K = P + N$)

Appendix C contains proofs of generalized versions of the theorems in this section.

Translation of conjunctive rules

Conjunctive rules are translated into a neural network by setting weights on all links corresponding to positive (i.e., unnegated) antecedents to ω, weights on all links corresponding to negated antecedents to $-\omega$, and the bias on the unit corresponding to the rule's consequent to $\frac{-2P+1}{2}\omega$. Using the terms defined above: $w_p = \omega$, $w_n = -\omega$, and $\theta = \frac{-2P+1}{2}\omega$. KBANN commonly uses a setting of $\omega = 4$, a value empirically found to work well. For example, Figure 2.8 shows a network which encodes:

```
A:- B, C, D, ¬E.
```

Intuitively, this translation method is reasonable. The input from the links plus the bias can only exceed zero when all of the mandatory antecedents are true and none of the prohibitory antecedents are true. (Units are only active when the net incoming activation plus

thesis is the idea of training networks that have been so constructed, rather than simply the construction of these networks.

the bias exceeds zero.) Hence, the unit will only be significantly active when all positively-weighted links carry a signal near one and all negatively-weighted links carry a signal near zero.

Theorem 1: *Mapping conjunctive rules into a neural network using:*

1. $w_p = -w_n = \omega > 0$
2. $\theta = \frac{-2P+1}{2}\omega$

creates a network that accurately encodes rules given that the rules have a "sufficiently small" number of antecedents. (The conditions on "sufficiently small" are given below.)

Proof of Theorem 1

This theorem is proved by showing that these mappings are correct in the following two situations:

A) The unit encoding the consequent of the rule is active when the rule's antecedents are satisfied.

B) The unit encoding the consequent of the rule is inactive when the rule's antecedents are not satisfied.

The proofs of each of these situations takes advantage of the monotonically-increasing property of the logistic activation function. This allows the analysis to focus on the boundary cases. This proof assumes only $C_i = 1 - C_a$ which provides nicely symmetric results. Also, $C_i = 1 - C_a$ means that $ln(\frac{1}{C_a} - 1) = -ln(\frac{1}{C_i} - 1)$.

Case A of Theorem 1 *Assume that the rule's antecedents are satisfied and show that the unit encoding the consequent is active.*

By assumption, the P positive antecedents are true and the N negative antecedents are false. Hence, the boundary condition given by Equation 2.1 obtains. This boundary condition expressed the minimum net incoming activation a unit could receive when all of its antecedents are satisfied. This occurs when all positive antecedents are at the minimum activation indicative of truth and all negative antecedents are at their maximum activation indicative of falsehood. Solving Equation 2.1 for the link weight yields Equation 2.2. Given the above assumptions about C_a and C_i and that $\omega > 0$, Equation 2.3 provides a bound on the relationship between the number of antecedents in a rule and the amount the activation of a unit may differ from 0 or 1. Given that this condition is met, the encoding scheme proposed by this theorem is correct for case A.

$$C_a \leq \mathcal{F}[PC_a\omega + NC_i\omega + \frac{-2P+1}{2}\omega] \tag{2.1}$$

$$\omega \geq \frac{ln(\frac{1}{C_a} - 1)}{K(1 - C_a) - \frac{1}{2}} \tag{2.2}$$

$$KC_i \leq \frac{1}{2} \tag{2.3}$$

Case B of Theorem 1 *Assume that the rule's antecedents are not satisfied and show that the unit encoding the consequent is inactive.*

A conjunctive rule is false when at least one positive antecedent is false or one negative antecedent is true. Hence, there are two boundary conditions for case B. The boundary conditions complement those of case A above as they capture the maximum net incoming activation a unit could receive when its antecedents are not satisfied. These conditions appear in Equations 2.4 and 2.5. Each of these equations, when solved for ω, yields Equation 2.6 (again, given the assumptions about C_a and C_i and that $\omega > 0$). So, the proposed encoding scheme is correct from case B when ω satisfies Equation 2.6.

Note that the denominator of Equation 2.6 requires $C_a > \frac{1}{2}$ to be true for $\omega > 0$ to hold. Thus, this constraint enforces $C_a > C_i$ under the assumption that $C_i = 1 - C_a$. This is a reasonable (and minor) constraint given the definitions of C_a and C_i.

$$C_i \geq \mathcal{F}[(P-1)\omega + C_i\omega + \frac{-2P+1}{2}\omega] \tag{2.4}$$

$$C_i \geq \mathcal{F}[P\omega - C_a\omega + \frac{-2P+1}{2}\omega] \tag{2.5}$$

$$\omega \geq \frac{ln(\frac{1}{C_i} - 1)}{C_a - \frac{1}{2}} \tag{2.6}$$

The scheme for translating conjunctive rules into networks has been show to correctly apply to both case A and case B. The final step to show that encoding scheme correct is to point out that the constraints on ω given by Equation 2.2 is compatible with the constraint given by Equation 2.6.

The only conditions on the correctness of this translation method are that K is "sufficiently small", where "sufficiently small" is given by Equation 2.3 and the very minor constraint that $C_a > 0.5$.

□

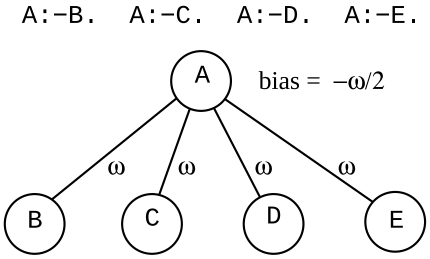


Figure 2.9: Translation of disjunctive rules into a KBANN-net.

Translation of disjunctive rules

To translate a set of rules encoding a disjunct, KBANN sets the weight of each link corresponding to a disjunctive antecedent to ω and the bias on the unit encoding the consequent to $-\frac{\omega}{2}$. For example, Figure 2.9 shows the network that results from the translation of four disjunctive rules into a KBANN-net. Intuitively, this is a reasonable strategy; the incoming activation overcomes the bias when one or more of the antecedents is true. The following theorem supports this intuition.

Theorem 2: *Mapping disjunctive rules into a neural network using:*

1. $w_p = \omega > 0$
2. $\theta = -\frac{1}{2}\omega$

creates a network that accurately encodes the disjunctive rules given that there are a "sufficiently small" number of rules. (Recall that each rule has a single antecedent. The conditions on "sufficiently small" are given below.)

Proof of Theorem 2

This theorem is proved by showing that this mapping is correct in the following two situations:

A The unit encoding the consequent of the set of rules encoding the disjunct is active when the one or more of the antecedents are satisfied.

B The unit encoding the consequent of the set of rules encoding the disjunct is inactive when the none of the antecedents are satisfied.

As in the proof of Theorem 1, the proofs of each of these situations takes advantage of the monotonically-increasing property of the logistic activation function. Also, as in Theorem 1, this proof assumes only $C_i = 1 - C_a$ (which provides nicely symmetric results). Recall that $C_i = 1 - C_a$ means that $ln(\frac{1}{C_a} - 1) = -ln(\frac{1}{C_i} - 1)$.

Case A of Theorem 2 *Assume that the consequent of the set of disjunctive rules is satisfied and show that the unit encoding that consequent is active.*

By assumption, at least one positive antecedent is true. (Disjunctive rules are assumed, without loss of generality, to have no negative antecedents. This assumption can be enforced by simply adding a rule that inverts any negated antecedents.) Hence, the boundary condition given by Equation 2.7 obtains. As in case A of Theorem 1, this boundary condition captures the minimum activation that a unit encoding the consequent of a set of disjunctive rules could receive when the consequent is satisfied. Solving Equation 2.7 for the link weight yields Equation 2.8. Note that this is the same lower bound on ω as determined in case B of Theorem 1. Also, note that this solution requires $C_a > \frac{1}{2}$ which is quite reasonable given its definition.

$$C_a \leq \mathcal{F}[C_a\omega - \frac{1}{2}\omega] \tag{2.7}$$

$$\omega \geq \frac{ln(\frac{1}{C_i} - 1)}{C_a - \frac{1}{2}} \tag{2.8}$$

Case B of Theorem 2 *Assume that the consequent of a set of disjunctive rules is not satisfied and show that the unit encoding that consequent is inactive.*

The consequent of a set of disjunctive rules is not satisfied only when the lone antecedent of each rule in the set is not true. Equation 2.9 expresses the boundary condition. Solving this equation for ω, yields Equation 2.10 (given the assumptions concerning C_a and C_i and that $\omega > 0$). Finally, this solution for ω is only correct when Equation 2.11, which defines "a sufficiently small number of antecedents", holds.

This solution parallels that of Theorem 1, case A. Both cases provide a lower bound on ω that is a function of the allowed deviation of activations from 0 and 1 as well as the total number of antecedents to the rule. Also, both solutions place an upper bound on KC_i.

$$C_i \geq \mathcal{F}[RC_i\omega - \frac{1}{2}\omega] \tag{2.9}$$

$$\omega \geq \frac{ln(\frac{1}{C_a} - 1)}{KC_i - \frac{1}{2}} \tag{2.10}$$

$$KC_i \leq \frac{1}{2} \tag{2.11}$$

The scheme for translating disjunctive rules into networks has been show to correctly apply to both case A and case B. The final step to show that this encoding scheme correct is to point out that the constraint on ω given by Equation 2.8 is compatible with the constraint given by Equation 2.10.

□

2.3 Refining KBANN-nets

KBANN refines its networks using backpropagation [Rumelhart86], a standard neural learning method. Unfortunately, KBANN-nets create problems for backpropagation because they start with confident answers (i.e., the output units have activation near zero or one), regardless of the correctness of those answers. This causes problems because under the standard formulation of backpropagation, when answers are confident little change is made to the network regardless of the correctness of the answer. Rumelhart *et al.* argue this is a desirable property for a standard neural networks because it means that their learning tends to be noise resistant [Rumelhart86, page 329]. However, when the outputs of the network are incorrect, this property makes it very difficult to unlearn the aspects of the network that cause the errors. This problem with unlearning mistakes has been termed the "flat spot" of backpropagation [Fahlman88].

In the general context of neural networks, several solutions that require only minor adjustments to backpropagation have been proposed for the flat spot problem [Franzini87, Hinton89]. Solutions involving more significant changes to backpropagation have also been proposed [Fahlman88, Barnard89]. The results for KBANN-nets described herein all use the solution suggested by Hinton [Hinton89]. Hence, the rest of this subsection describes that approach.

Hinton's [Hinton89] approach to eliminating the flat spot in error propagation is given by the equations in Table 2.3. The approach uses a different error function so that large error signals are propagated through the network when the difference between the actual and desired output is large. Specifically, Hinton replaces the standard error function (Equation 2.12) with the *cross-entropy* function (Equation 2.15).

Statistically, the cross-entropy function operates by treating the actual and desired activation of each output unit as the probability that the activation value is actually one, independent of any other output units. This statement is equivalent to Equation 2.14. The independence assumption is enforced in Equation 2.14 by raising the terms to either d_i or $(1 - d_i)$. As d_i is either zero or one, raising terms to these powers results in either the term or one. Thus, when the desired activation is one, Equation 2.14 only depends upon the difference between the actual activation and one.

Under this assumption of independence among the output values, the likelihood of producing the desired outputs is maximized when the *cross entropy* between actual and desired outputs is minimized. That is, to minimize the error of the network, maximize Equation 2.14. Equation 2.15 is an algebraic restatement of Equation 2.14 which takes the log of Equation 2.14

Table 2.3: Revisions to error propagation.

$$E = \frac{1}{2}\sum_{i=1}^{n}(d_i - a_i)^2 \tag{2.12}$$

$$\begin{aligned} e_i &= -\frac{\partial E}{\partial NetInput_i} \\ &= -\frac{\partial E}{\partial A} * \frac{\partial A}{\partial NetInput_i} \\ &= [d_i - a_i] * [(1 - a_i) * a_i] \end{aligned} \tag{2.13}$$

$$C = p(\vec{\mathbf{d}}|network) = \prod_{i=1}^{n}(a_i(1 - d_i) * (1 - a_i)^{d_i}) \tag{2.14}$$

$$C = -\sum_{i=1}^{n}[(1 - d_i) * log_2 a_i) + (d_i * log_2(1 - a_i)] \tag{2.15}$$

$$\begin{aligned} e_i &= \frac{\partial C}{\partial NetInput_i} \\ &= log(2) * \left(\frac{d_i}{a_i} - \frac{1 - d_i}{1 - a_i}\right) * a_i * (1 - a_i) \end{aligned} \tag{2.16}$$

where: E is the standard definition of error output unit error
A is the activation function
e_i is the error of unit i
a_i is the activation of unit i
d_i is the desired activation for unit i
n is the number of output units
$NetInput_i$ is the net incoming activation to unit i
$\vec{\mathbf{d}}$ is the vector of desired output activations
$p(\vec{\mathbf{d}}|network)$ is the probability of the desired outputs given the network
C is the cross-entropy function

to put it into a form that can be easily applied to neural learning.

Use of the cross-entropy error function changes only the error signal at output units, for which Equation 2.16 replaces Equation 2.13. Hidden units still use Equation 2.13. Equation 2.16 is considerably simpler when the desired activation is 0 and 1 (recall that this assumption underlies cross-entropy); it respectively reduces to $-o_i$ and $(1 - o_i)$ times a constant. (I.e., $\frac{\partial C}{\partial net_i} = (1 - o_i)$ when $d_i = 1$ and $\frac{\partial C}{\partial net_i} = -o_i$ when $d_i = 0$.) Hence, under the cross-entropy

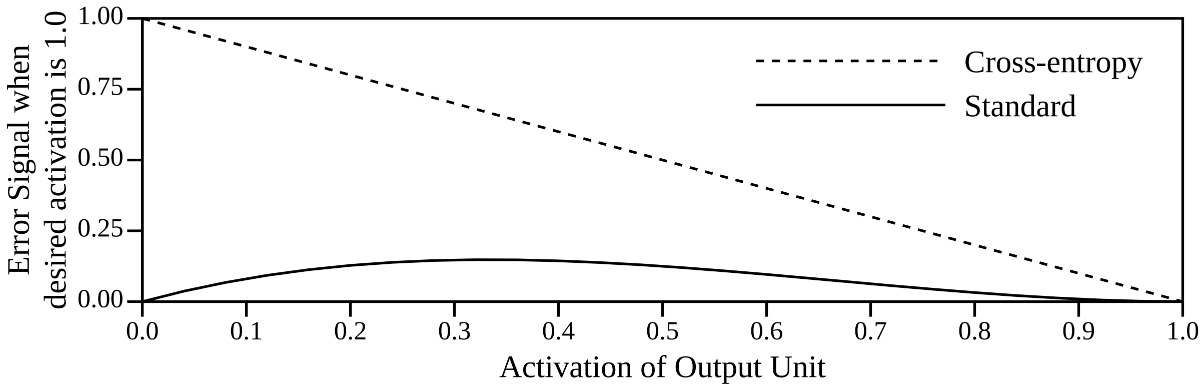


Figure 2.10: Error signal for two popular error functions.

error function, when the desired output is either zero or one, the error signal is proportional to the actual error.

Figure 2.10 graphs the error propagated backward from output units by cross-entropy and the standard error function when the desired activation is 1.0. Highlighted by Figure 2.10, the principle advantage of cross-entropy is that, unlike the standard error function, the magnitude of cross-entropy's error signal always varies with the difference between the desired and the actual activations.

In summary, cross-entropy makes more sense for KBANN-nets than the standard error function because it reduces problems that result from initially confident, but possibly-mistaken answers. Empirical comparisons (results not shown) of learning in KBANN-nets using both the standard and cross-entropy error functions show this analysis to be correct — cross-entropy results in shorter training times and slightly better generalization.

2.4 Extracting Rules from Trained Networks

After KBANN-nets have refined, they can be used as highly accurate classifiers. The results presented in the first parts of Chapter 3 demonstrate just that. However, trained KBANN-nets provide no explanation of how an answer was derived. Nor can the results of their learning be shared with humans or transferred to related problems. (Pratt's work on direct transfer between two neural networks is an exception to this [Pratt91].)

The extraction of symbolic rules directly addresses these problems. It makes the information learned by the KBANN-net during training accessible for human review and justification of answers. Moreover, the modified rules can be used as part of knowledge bases for the solution of related problems.

This section presents two approaches to the extraction of rules from trained KBANN-nets. The next section describes features shared by all approaches to the extraction of rules from neural networks and explains two assumptions about trained neural networks made by the algorithms described herein. The two algorithms appear in the subsequent sections.

2.4.1 Underpinnings of rule extraction

Assumptions

The methods of rule extraction described below make two assumptions about trained KBANN-nets. The first assumption is that training a KBANN-net does not significantly shift the meaning of its units. By making this assumption, the methods are able to attach labels to every consequent and antecedent of the extracted rules, thereby enhancing the comprehensibility of the rules. These labels correspond to consequents in the symbolic knowledge upon which the KBANN-net is based. The utility of the labels would be compromised if meaning shifted significantly.

Observation of trained networks indicates that meanings usually are quite stable. Yet, meanings can shift. Such shifts are most common at units associated with the least certain of the initial rules.[3] Hence, inappropriately labeled rules can be used as pointers to weak portions of the initial knowledge base.

The second assumption is that almost all of the units in a trained KBANN-net are either fully active (i.e., have activation near one) or inactive (i.e., have activation near zero). By making this assumption, each non-input unit in a trained KBANN-net can be treated as a step function or a Boolean rule. Therefore, the problem for rule extraction is to determine the situations in which the "rule" is true. Again, examination of trained KBANN-nets suggests that this assumption is valid.

This second assumption is not particularly restrictive; the standard logistic activation function can be slightly modified to ensure that units approximate step functions. In particular, Equation 2.17, in Table 2.4, is the logistic activation function to which a scaling parameter, σ, has been added. The standard value for this parameter is 1.0. However, as indicated by Figure 2.11, changing σ can make the logistic activation function resemble a straight line or a step function.

Adjusting σ so that the activation function resembles a step function must be done cautiously. Standard backpropagation looses effectiveness as the activation function steepens due to a widening of the "flat spot" [Fahlman88] in the error propagation mechanism. (The cross-entropy error-function discussed in the previous section only addresses this problem at the output units. When every unit resembles a step function, the flat spot is a problem throughout the network.) As a result, it is difficult and time-consuming to train networks that have

[3] Currently, certainty of rules is information known by the user but not provided to KBANN. Section 7.2 describes plans for adding the ability to handle assessments of rule confidence.

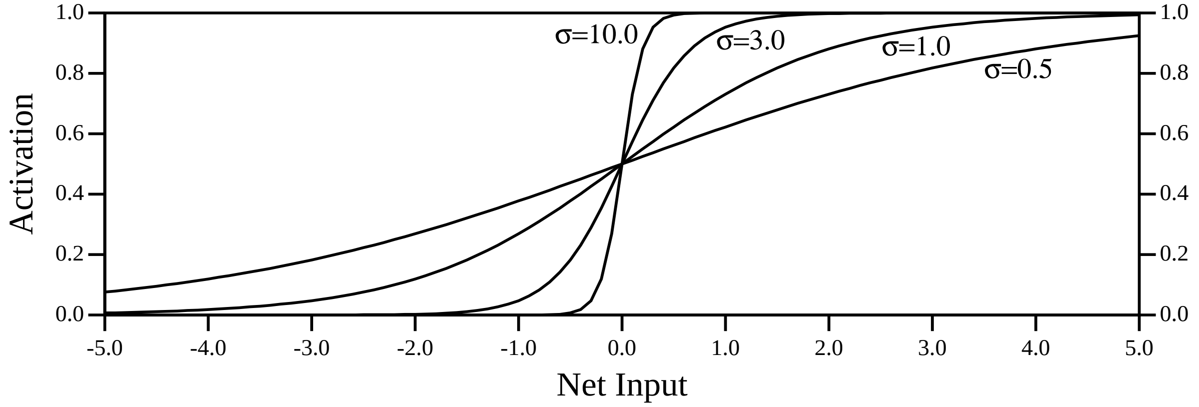


Figure 2.11: Activation of units when σ is varied.

activation functions which approximate step functions.[4] In addition, according to the unpublished lore of neural networks, some of the power of neural networks comes from their ability to make use of partially-activated units. Hence, adjusting the σ parameter so that units resemble step functions may impose some correctness penalty on the networks.

Commonalities

The methods described in this thesis for extracting rules from neural networks, as well as those in the literature, do so by trying to find combinations of the input values to a unit that result in it having an activation near one. Recall that units in neural networks normally have activations defined by Equations 2.17 and 2.18. Broadly speaking, the result of Equation 2.18 is that if the summed weighted inputs exceed the bias, then the activation of a unit will be near one. Otherwise the activation will be near zero. *Hence, rule extraction methods look for ways in which the weighted sum of inputs exceeds the bias.*

The second of the above assumptions, that all units in trained networks have activations near zero or one, simplifies this task by forcing links to carry a signal equal to their weight or no signal at all. That is, Equation 2.17 reduces to Equation 2.19 As a result, rule extraction need only be concerned with the weight of links entering a unit and may ignore the activation of the sending unit.

Rule extraction is further simplified by the fact that units always have non-negative activations. This property allows rule-extraction methods to take the sign of a link's weight as a perfect indicator of the way in which an antecedent will be used. That is, negatively-weighted links can only give rise to negated antecedents, while positively-weighted links can only give

[4] Kruschke and Movellan [Kruschke91] suggest that is possible to use backpropagation to train σ for each unit at the same time as standard training. Their technique would avoid some of the difficulties of training with very steep activation functions. However, their method does not guarantee that units assume a step-like character at the end of training.

Table 2.4: The logistic activation function.

$$a_i = f\left(\sum_j (w_{i,j} * a_j) + \theta_i\right) \tag{2.17}$$

$$f(x) = \frac{1}{1 + e^{-\sigma x}} \tag{2.18}$$

$$a_i = f\left(\sum_{\{j|a_j \approx 1\}} w_{i,j} - \theta_i\right) \tag{2.19}$$

where: a_i is the activation of unit i
$w_{i,j}$ is the weight on a link from unit j to unit i
θ_i is the "bias" of unit i
σ is a paramenter affecting the slope of the sigmoid.

rise to unnegated antecedents. This considerably reduces the size of the search space [Fu91].

2.4.2 The Subset method

The first method for rule extraction is referred to in this work as the Subset algorithm. It is so named because it operates by attempting to find subsets of the units to which a unit is connected whose summed weighted activations exceed the bias of that unit. The method was developed independently, though it is fundamentally similar to approaches described by Saito and Nakano [Saito88], Fu [Fu91] and Masuoka [Masuoka90].[5] Thus, Subset represents the state of the art in the published literature.

The Subset algorithm is summarized in Table 2.5. It relies heavily upon the assumption that units are either active or inactive, as this assumption allows the method to look only at link weights. Hence, steps 2 and 3 of the algorithm ignore activations of the units at the sending ends of the links.

As an example, consider the unit in Figure 2.12a. Given that the link weights and the bias are as shown, the four rules listed in Figure 2.12b are extracted by the algorithm.

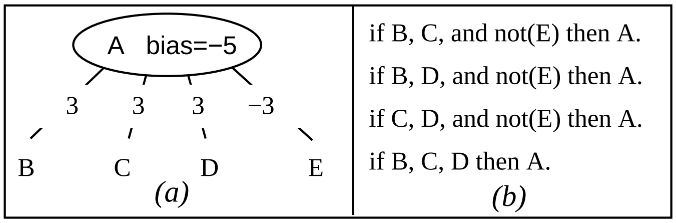
The major problem with the Subset method is that the cost of finding all subsets grows as the size of the power set of the links to each unit. Thus, the algorithm can only be expected to work on simple networks and toy domains. To sidestep these combinatorial problems, Saito and Nakano [Saito88] establish a ceiling on the number of antecedents in extracted rules.

[5] Empirical comparisons (results not reported) of the method described here to the algorithm described by Saito and Nakano indicate the algorithm described here is superior in terms of generalization accuracy.

Table 2.5: The SUBSET approach to rule extraction.

```
1. With each $U \in$ {all hidden and output units}
   Let $\theta$ be the bias of unit $U$

2. Find up to $\beta_p$ groups of positively-weighted links to unit $U$ such that
   $0 < \theta + \sum(link\ weights\ of\ group)$
   Call the set of groups found in this step $G_p$

3. For each $\mathcal{P} \in G_p$

    Find up to $\beta_n$ groups of negatively weighted links to unit $U$ such that
        $0 > \theta + \sum(link\ weights\ of\ \mathcal{P}) + \sum(link\ weights\ of\ group)$
        Call the set of groups found in this step $G_n$
    For each $\mathcal{N} \in G_n$
        Create a rule with the following form:
            ``if $\forall\mathcal{P}$ and $\neg\forall\mathcal{N}$ then (name of $U$)''

4. Remove any duplicate rules.
```

Appendix B.2.1 contains pseudocode for this algorithm.



if B, C, and not(E) then A.
if B, D, and not(E) then A.
if C, D, and not(E) then A.
if B, C, D then A.

Figure 2.12: Rule extraction using the SUBSET algorithm.

However, the initial rules for the real-world domains studied in this work involve large numbers of antecedents. As a result, the lowest possible ceiling on the number of antecedents might require considering more than 10^5 sets of antecedents.

Therefore, the implementation of SUBSET described here explicitly bounds the number of positive and negative groups, rather than setting a bound on the number of antecedents. While this approach is advantageous because it allows rules with an unlimited number of antecedents, it may discover up to $\beta_p * \beta_n$ rules for each non-input unit in the network to be translated. In practice, far fewer rules are extracted. Yet, the worst case is a significant concern because the accuracy of the set of extracted rules is expected to increase as the size of β_p and β_n increases. Thus, when using the SUBSET method a tension can be expected between the accuracy and potential size of the set of extracted rules. This tension is empirically investigated in Section 3.5.

The SUBSET method typically extracts about 300 rules from networks trained for the problems studied in this work. While 300 rules is a large set, it is smaller than many handcrafted expert systems [Fozzard88, McDermott82]. Hence, SUBSET provides sets of rules that are, at least potentially, tractable. However, the rules tend to hide significant structures in trained networks. For instance, in Figure 2.12a, the links to B, C and D all have the same weight while the link to E has the negative of that weight. Looking at the problem in this way suggests the rules in Figure 2.12b could be rewritten as the following rule, which provides a clearer statement of the conditions on A:

```
if (3 of {B, C, D, not(E)}) then A.
```

This consideration of structure shared among several rules led to the development of the algorithm presented next.

2.4.3 The NOFM method

The second algorithm for rules extraction, referred to as NOFM, explicitly searches for antecedents of the form:

`if (`N` of the following `M` antecedents are true) then ...`

This approach was taken because, as discussed in the previous section, rule sets discovered by the SUBSET method often contain "N-of-M" style concepts. Furthermore experiments indicate that ANNs are good at learning N-of-M concepts [Fisher89] and that searching for N-of-M concepts is a useful inductive bias [Murphy91]. Finally, note that purely conjunctive rules result if $N = M$, while a set of disjunctive rules results when $N = 1$; hence, using N-of-M rules does not restrict generality.

The algorithm

The idea underlying NOFM, an abstracted version of which appears in Table 2.6, is that individual antecedents (links) do not have a unique importance. Rather, groups of antecedents form equivalence classes in which each antecedent has the same importance as, and is interchangeable with, other members of the class. This equivalence class idea is the key to the NOFM algorithm; it allows the algorithm to consider groups of links without worrying about the particular links within the group.

Step 1, clustering. Backpropagation training tends to group links of KBANN-nets into loose clusters rather than tight equivalence classes as assumed by the NOFM algorithm. Hence, the first steps of NOFM group links into equivalence classes. This grouping can be done in either of two ways.

Table 2.6: The NofM approach to rule extraction.

1. With each hidden and output unit,
   form groups of similarly-weighted links.
2. Set link weights of all group members to the average of the group.
3. Eliminate any groups that do not significantly affect whether the unit will be active or inactive.
4. Holding all links weights constant, optimize biases of all hidden and output units using the backpropagation algorithm.
5. Form a single rule for each hidden and output unit. The rule consists of a threshold given by the bias and weighted antecedents specified by the remaining links.
6. Where possible, simplify rules to eliminate weights and thresholds.

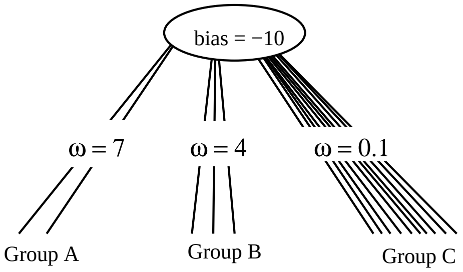
Appendix B.2.2 contains pseudocode for this algorithm.

First, clustering may be done using a standard clustering method such as the *join* algorithm [Hartigan75]. This method clusters by joining the two closest clusters starting with n clusters of size 1. Clustering stops when no pair of clusters is closer than a set distance (Kbann uses 0.25).

Alternately, clustering can be done by sorting all links and performing a single pass across the sorted links. Using this approach, items are added to the current cluster until either: (a) a new item differs from the mean of the cluster by more than a specified bound or (b) the addition of an item causes an item already in the cluster to differ from the cluster's mean by more than this same bound. As with the join approach to clustering, the bound used by Kbann for this approach is 0.25. (See Appendix B.2.2 for pseudocode of this clustering approach.)

Each of these two clustering methods work quite well for the NofM algorithm. However, the second method is used almost exclusively as its complexity is $O(n * log(n))$ while the complexity of the *join* procedure is $O(n^2)$.

Step 2, averaging. After groups are formed, the second step of the algorithm is to set the weight of all links in each group to the average of each group's weight. Thus, the first two steps of the algorithm force links in the network into equivalence classes as required by the rest of the algorithm.



Possible Combinations of A and B	Total weight of combo
1B	4
1A	7
2B	8
1A+1B	11
3B	12
2A	14
1A + 2B	15
etc	etc

Figure 2.13: Link weight combinations that indicate a cluster should be dropped.

Step 3, eliminating. With equivalence classes in place, the procedure next attempts to identify and eliminate those groups that are unlikely to have any bearing on the calculation of the consequent. Such groups generally have low link weights and few members. Elimination proceeds via two paths one heuristic, and one algorithmic.

The first elimination procedure algorithmically attempts to find clusters of links that cannot have any effect on whether or not the total incoming activation exceeds the bias. This is done by calculating the total possible activation that each cluster can send (taking into account properties of units such as that only one unit related to each nominal input feature may be active at any time). This total possible activation is then compared to the levels of activation that are reachable given link weights in the network. Clusters that cannot change whether the net input exceeds the bias are eliminated. Note that this procedure is very similar to SUBSET. However, it is possible to take advantage of the limited number of possible link weights to considerably reduces the combinatorics of the problem.

For instance, consider Figure 2.13 which illustrates a unit with a bias of -10 and three clusters of links: (A) two links of weight 7, (B) three links of weight 4, and (C) ten links of weight 0.1. In this case, the third cluster is eliminated. Its total possible activation is 1.0 and the only reachable activations that are less than the bias using the other two clusters are 7 and 8. Neither of these activation levels, when combined with the 1.0 total from group C, exceeds 10.0. Hence, the 0.1 weight links can have no effect on the unit into which they feed. So, the cluster is safely eliminated.

The heuristic elimination procedure is based explicitly upon whether the net input received from a cluster ever is necessary for the correct classification of a training example. This procedure operates by presenting a training example to the clustered network and sequentially zeroing the input from each cluster. If doing so results in a change in the activation of the unit receiving activation from the cluster, then the cluster is marked as necessary. After doing this for every example, any clusters that are not marked as necessary are eliminated.

Step 4, optimizing. With unimportant groups eliminated, the fourth step of NofM is to optimize the bias on the unit. This step is necessary because the averaging of the link weights in a cluster and elimination of links can shift the pattern of activation of a unit. As a result, prior to optimization, networks on which the first three steps of the NofM algorithm have been applied may have error rates that are significantly higher than they were at the end of training.

Optimization can be done by freezing the weights on the links so that the groups stay intact and retraining the network using backpropagation. To reflect the rule-like nature of the network, the activation function is modified by the addition of a slope term as was shown in Equation 2.17. Setting σ to a large positive value (e.g., 20) ensures that units in the network take on values near zero or one.

Step 5, extracting. This step of the NofM algorithm form rules that simply re-express the network. That is, rules are created by directly translating the bias and incoming weights to each unit into a rule with weighted antecedents such that the rule is true if the sum of the weighted antecedents exceeds the bias. Note that because of the equivalence classes and elimination of groups, these rules are considerably simpler than the original trained network; they have fewer antecedents and those antecedents tend to be in a few weight classes. (See the extracted rules in Section 3.5 for examples of unsimplified rules.)

Step 6, simplifying. Finally, rules are simplified whenever possible to eliminate the weights and thresholds. Simplification is accomplished by scanning each restated rule to determine the possible combinations of group items that exceed the rule's threshold (i.e., bias). This scan may result in more than one rule. Hence, there is a tradeoff in the simplification procedure between complexity of individual rules and complexity resulting from a number of simple rules.

For example, consider the rule[6]:

```
A :- 10.0 < 5.1*number-true{B, C, D, E} +
            3.5*number-true{X, Y, Z}
```

The simplification procedure would simplify this rule by rewriting it as the following three rules:

```
A :- 2 of {B, C, D, E}
A :- 1 of {B, C, D, E} and 2 of {X, Y, Z}
A :- X, Y, Z.
```

[6] The function `number-true` returns the number of antecedents in the following set that are true.

If the elimination of weight and biases requires rewriting a single rule with more than five rules,[7] then the rule is left in its original state.

Example of the NofM method

As an example of NofM, consider Figure 2.14. This figure illustrates the process through which a single unit with seven incoming links is transformed by the NofM procedure into a rule that requires two of three antecedents to be true.

The first two steps of the algorithm transform the unit with seven unique inputs into a unit with two classes of inputs, one with three links of weight 6.1 and one with for links of weight 1.1. The next step of the algorithm eliminates the group with weight 1.1 from consideration because there is no way that these links – either alone or in combination with links in the other group – can affect whether or not the sum of the incoming activation to unit Z exceeds the bias on Z. (Note this step takes advantage of the assumption that units have activations near either zero or one is necessary here.)

Figure 2.14 does not illustrate bias optimization. The lower left panel of Figure 2.14 shows the re-expression of the simplified unit as a rule. The final step of the algorithm is illustrated by the bottom right panel of Figure 2.14, in which the rule with weighted antecedents and a threshold is transformed into a simple N-of-M style rule. (The Subset algorithm would find the three rules that are the expansion of the 2-of-3 rule found by NofM. However, to find these three rules Subset would have had to consider as many as 125 possibilities.)

Algorithmic complexity of NofM

The complexity of NofM is difficult to precisely analyze as the bias optimization phase uses backpropagation. However, the problem addressed in bias-optimization is considerably simpler than the initial training of the network. Usually, networks have more than an order of magnitude fewer links during bias optimization than during initial training. Moreover, only the biases are allowed to change. As a result, this step takes a reasonably short time.[8] Each of the other steps requires O(n) time except for the initial clustering, which requires O($n * log(n)$) time using the faster of the two algorithms described above.

2.4.4 Subset and NofM: Summary

Both of these rule-extraction algorithms have strengths with respect to the other. For example, the individual rules returned by Subset are more easily understood than those returned by

[7] When there are more than five disjuncts to a single consequent it is difficult to keep understand the conditions under which the consequent is satisfied.

[8] Training neural networks has been proven NP-complete [Judd88, Blum88]. However, Hinton [Hinton89] suggests that in practice backpropagation usually runs in O(n^3) time.

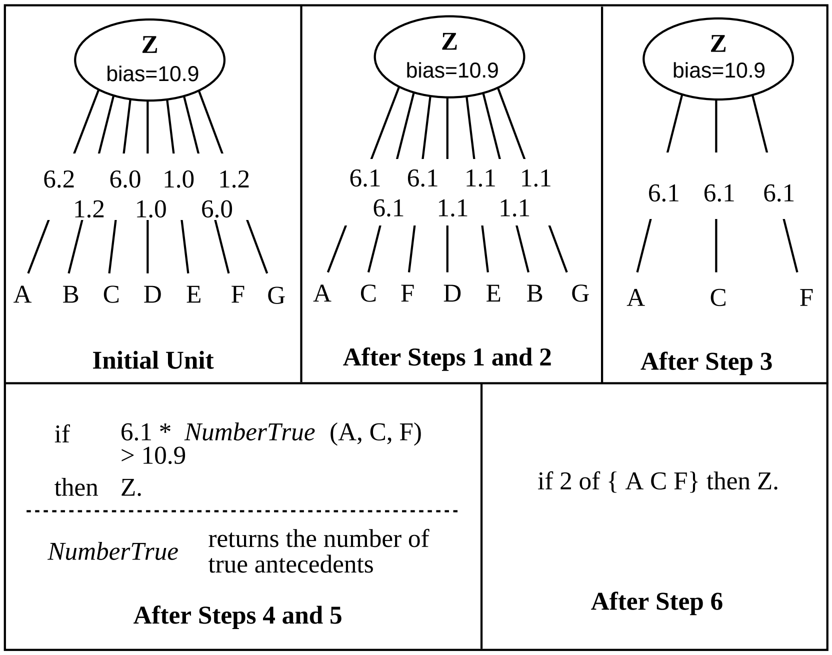


Figure 2.14: An example of rule extraction using NofM.

NofM. However, because Subset can be expected to return many more rules (which are often quite repetitive) than NofM, the rule sets returned by NofM may actually be easier to understand than those of Subset. More significantly, Subset is an exponential algorithm whereas NofM is, in practice [Hinton89], cubic. Finally, results presented in Section 3.5 indicate that the rule sets derived by NofM retain the accuracy of the networks from which they are extracted, while the rules extracted by Subset are significantly worse.

2.5 Review of Kbann's algorithms

This chapter has described the three algorithms that together comprises Kbann. The first algorithmic step is a rules-to-network translator. This step *inserts* domain information, in the form of sets of propositional rules, into a neural network. As a result, the network initially makes the same responses as the rules upon which it is based.

The second step of Kbann *refines* the networks created by the first step. Aspects of the insertion process render standard backpropagation less than efficient on the networks that Kbann creates. Hence, the refinement algorithm of Kbann uses a modified definition of error.

The final algorithmic step of Kbann is the *extraction* of symbolic rules from networks trained in the second step. Extraction is performed by a network-to-rules translator for which two methods are described. The extraction of rules makes it possible for trained networks to explain their actions. It also make possible the reuse of the extracted refined rules on related problems. Hence, rule extraction "closes the loop" by allowing rules refined by KBANN to be fed back to the rules-to-network translator (possibly after expert editing).