Chapter 6

Revision and Extension of Inference Networks

In Chapter 5, we considered methods for learning disjunctions of logical, threshold, and competitive concept descriptions, which considerably expanded the representational power of these formalisms. In particular, the resulting decision lists were able to represent concepts that involve complex decision boundaries among different classes. However, the concept descriptions we examined there remained simple in structure, and they had few implications for the *organization* of conceptual knowledge in memory.

The current chapter focuses on learning in *inference networks*, a widely used organizational framework that occurs throughout artificial intelligence.[1] Most work on this topic has emphasized the role of background knowledge in learning, and has cast induction as a problem of refining or extending an existing inference network.

We will examine three variations of this idea – extending an incomplete network, specializing a complete network, and revising an incorrect network. Finally, we will examine approaches to the more challenging problem of constructing entire inference networks from training data in the absence of background knowledge. However, we should first examine some more basic issues raised by this framework.

1. The literature often refers to such inference networks as *domain theories*. However, one can store domain knowledge in other organizational frameworks, so we will usually rely on the more specific term.

Table 6-1. A simple inference network, stated as propositional Horn clauses, for recognizing instances of the disease `lethargia` from characteristics of cells. Note the use of the two nonterminal symbols `simploid` and `neoplasm`.

```
lethargia :- simploid, neoplasm.
simploid  :- one-tail, thin-wall.
simploid  :- two-tails.
neoplasm  :- one-nucleus, thick-wall.
neoplasm  :- two-nuclei.
```

6.1 General issues surrounding inference networks

As we suggested above, an inference network provides an organization for a set of concept descriptions, and this organization has implications for representational ability, performance, and learning. In this section we consider some basic issues that arise in this framework, focusing first on the structure of inference networks, then on their use, and finally on their acquisition.

6.1.1 The structure of inference networks

An inference network consists of nodes and directional links. Each node denotes either a primitive feature or some defined concept, and each link connects a lower-term feature or concept to some higher-order concept. Taken together, the set of nodes A with links pointing to concept C can be viewed as the antecedents of a 'rule', and C can be viewed as the rule's consequent. In general, the presence of nodes in A constitute evidence for the presence of C, although the nature of evidence combination varies widely. The notion of directional links is central to inference networks, and contrasts with the organizational framework we will consider in Chapter 7. Although one can view the nodes in an inference network as partially ordered, this ordering does *not* correspond to the one involving the extension of concepts discussed in previous chapters.

Table 6-1 presents a simple inference network composed of logical concepts for the disease `lethargia`. We have stated this network as a set of Horn clauses, but Figure 6-1 (a) shows an alternative depiction in terms of nodes and links. In this representation, each AND node (with an arc connecting incoming links) corresponds to the antecedent of a

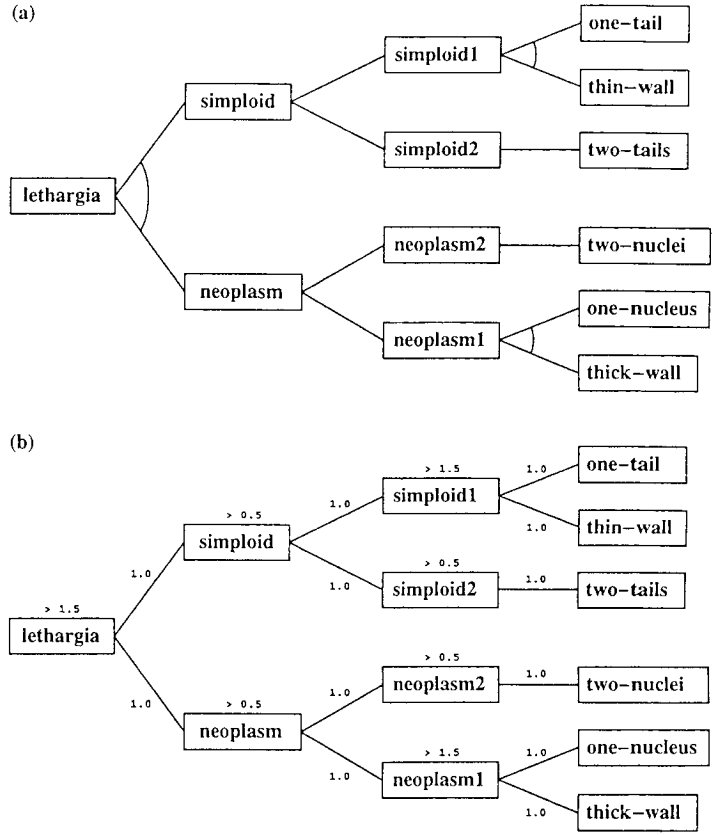

Figure 6-1. Two inference networks for the concept `lethargia` with the same extensional definition. Network (a), which maps directly onto the rules in Table 6-1, contains logical units that alternate between AND nodes (with arcs) and OR nodes. Network (b) contains threshold units with link weights and thresholds that mimic the logical network, although other numbers produce different behavior. The figure only shows links with nonzero weights.

rule in the table, whereas each OR node (with no arc on incoming links) corresponds to the head of one or more rules. The only exception is the AND node for `lethargia`, which occurs as the head of only one rule.

In general, a logical inference network alternates between AND nodes and OR nodes. In this example, the network defines a `lethargia` cell as the conjunction of a `simploid` cell and a `neoplasm` cell. It defines

the concept simploid in turn as either a cell that has one-tail and a thin-wall or as one that has two-tails. Similarly, the network defines neoplasm as occurring in cells that either have one-nucleus and a thick-wall or that have two-nuclei.[2]

Figure 6-1 (b) shows a network of threshold units, often called a *multilayer neural network*, with the same structure as the logical one. Here each link has an associated weight, and each non-input node has an associated threshold. As we saw in Chapter 3, the extension of such a concept definition is a function of the weighted sum of its antecedent links and its threshold. One can use these numbers to mimic the behavior of OR nodes and AND nodes, as shown in the figure, but threshold units are not limited to such semantics. Also, in some cases they allow a simpler network structure. In this example, we could remove the nodes simploid1, simploid2, neoplasm1, and neoplasm2, connecting thin-wall and two-tails directly to simploid, and linking thick-wall and two-nuclei directly to neoplasm. The appropriate weights and thresholds give the same extension as the network shown.

One can achieve similar effects with networks of m-of-n concepts, although they provide less flexibility than arbitrary threshold units. Networks of spherical units can also produce useful decision boundaries, especially when numeric attributes are involved. Mixtures of different types of threshold units are also possible. One combination, with spherical units at the internal layer and linear units at the output level, is commonly known as a *radial basis function*. However, this organizational scheme is typically used not for classification but for numeric prediction, which we discuss in Chapter 8.

Another instantiation of inference networks involves probabilistic concept descriptions. This approach, often referred to as *Bayesian networks*, specifies one node for each attribute and one link for each dependency between one attribute and another. Stored with each node is a table of conditional probabilities that, for each value V of the dependent attribute and for each combination of values C for its parent attributes, specifies the probability that V will occur given C. For a Boolean feature X with two parents Y and Z, this means storing entries

2. Given only the possibility of one or two nuclei, we could remove the one-nucleus condition from the first neoplasm rule; similarly, we might remove the one-tail literal from the first simploid conjunction. However, their presence serves a function in later examples, and they would be necessary if other values were possible for the tail and nucleus attributes.

Table 6-2. A partial set of conditional probabilities for a probabilistic inference network with the same structure as in Figure 6-1, and with causal dependencies going from right to left. Each value corresponds to a prior probability or to an entry in a conditional probability table.

P(one-tail) = 0.5,	P(simploid1 | one-tail) = 0
P(thick-wall) = 0.5,	P(simploid1 | $\overline{\text{one-tail}}$) = 1
P(simploid1 | one-tail, thick-wall) = 0	
P(simploid1 | $\overline{\text{one-tail}}$, thick-wall) = 0	
P(simploid1 | one-tail, $\overline{\text{thick-wall}}$) = 1	
P(simploid1 | $\overline{\text{one-tail}}$, $\overline{\text{thick-wall}}$) = 0	
P(simploid | simploid1, simploid2) = 1	
P(simploid | $\overline{\text{simploid1}}$, simploid2) = 1	
P(simploid | simploid1, $\overline{\text{simploid2}}$) = 1	
P(simploid | $\overline{\text{simploid1}}$, $\overline{\text{simploid2}}$) = 0	
P(lethargia | simploid, neoplasm) = 1	
P(lethargia | $\overline{\text{simploid}}$, neoplasm) = 0	
P(lethargia | simploid, $\overline{\text{neoplasm}}$) = 0	
P(lethargia | $\overline{\text{simploid}}$, $\overline{\text{neoplasm}}$) = 0	

for $P(X|Y,Z)$, $P(X|\bar{Y},Z)$, $P(X|Y,\bar{Z})$, and $P(X|\bar{Y},\bar{Z})$, with analogous entries for $\bar{X}$ (though these are redundant for Boolean features).

Table 6-2 shows a partial set of table entries for a Bayesian network that is equivalent to our logical inference network, assuming the structure in Figure 6-1 (a) and assuming that causal dependencies point from right to left. In this example, all probabilities are either 1 or 0 to mimic AND nodes and OR nodes. However, probabilistic inference networks are like threshold networks in that they support a continuum of relations between the AND/OR extremes, which they represent with conditional probabilities between 1 and 0.

Although our examples have assumed Boolean features, one can extend the three formalisms to handle other types of domains. We will see an example involving numeric attributes in a logical network in Section 6.2, and the threshold and probabilistic schemes handle such tasks as well. As normally formulated, Horn clauses support predicates with arguments, and much of the recent work on learning with logical networks has dealt with such formalisms. However, we delay treatment of this topic until Chapter 8 and focus on Boolean tasks in this chapter, both for the sake of simplicity and because there has been little work on threshold and probabilistic methods for relational domains.

6.1.2 The use of inference networks

The term 'inference network' suggests the use of such knowledge structures – to make inferences about some unobserved concepts. Typically, the performance system is presented with features or literals that describe some situation in the world, and the inference network must conclude whether this constitutes an instance of some concept specified in the network. In other words, it must chain through the network, concluding consequents from antecedents, until inferring (or failing to infer) the concept in question. This is simply another form of the classification task we have encountered in previous chapters.

The details of the inference process can vary considerably. For example, the performance component associated with many logical inference networks (such as PROLOG) chains backward from the concept C it is attempting to infer. If the algorithm can construct an AND tree that proves the observed literals satisfy C's definition, it concludes that C is present; otherwise it infers by default that C is absent.[3] However, a forward-chaining control structure is also possible, with rules matching against terms that have been observed or inferred, which lead to new inferences that let other rules match. Provided one is concerned only with accuracy and not efficiency, the inference mechanism does not matter as long as it considers all possible paths before concluding it has failed.

The performance techniques used for threshold inference networks typically apply in the forward direction, computing a weighted sum of the evidence from each node at level N to determine the 'activation' of each node at level $N+1$. Some methods simply infer an activation of 1 if the sum S exceeds the threshold θ and 0 (or in some versions -1) if it does not. Other approaches employ a sigmoid function to compute the consequent activation from the antecedent activations, giving a smooth transition rather than a step function. One commonly used scheme calculates the weighted sum $S = \sum_k w_k x_k$ of the inputs, then uses the logistic function

$$\frac{1}{1 + e^{-(S-\theta)}}$$

to determine the activation of the consequent. In this approach, nodes in the network receive an activation anywhere from 0 to 1, but the overall

3. When applied to Horn clauses that include predicates with multiple variables, this approach can accomplish much more than classify instances. We will see examples of this power in Chapters 10 and 11.

classification task still requires an all-or-none decision. Typically, such methods conclude that the instance satisfies a concept C if the activation of C is 0.5 or greater.

One can use a very similar method for some probabilistic inference networks, provided the dependency links point from the concept C being inferred toward the observable terms. Starting from the observable terms O, one uses Bayes' rule to compute the probabilities of nodes N that only have links directly from O nodes. One then uses the results to compute the probabilities of nodes that only have dependency links directly from nodes in $O \cup N$, and so on, until reaching the target node C. However, in some domains it is reasonable to assume that the observed features cause the unobserved concept C. In this case, one can compute the probability of C from the observed terms O using more sophisticated methods, which we do not have the space to discuss here. Because probabilistic networks produce a probability for a concept rather than a decision, the same issue arises as with threshold networks. One natural response is to conclude that an instance I satisfies concept C if $P(C|I) > 0.5 > P(\bar{C}|I)$.

Earlier we emphasized the directional nature of inference networks, but the above use of probabilistic structures shows that their use is not limited to propagating beliefs in the direction of the links. A similar flexibility exists with logical networks, which one can use to make default assumptions that, if present, would let one explain the presence of some observed literal. This approach, which is often termed *abduction*, considerably extends the uses of inference networks. Nevertheless, in this chapter we will limit our attention to simpler uses of this framework.

6.1.3 The task of revising inference networks

Now that we have clarified the organization and use of inference networks, we can specify the learning task associated with them:

- *Given*: A set of training instances and their associated classes;
- *Given* (optional): An initial inference network that incorporates some knowledge about the domain;
- *Find*: An inference network that, to the extent possible, makes accurate predictions about novel test instances.

As we mentioned earlier, much of the work on inference networks has examined the role of background knowledge. In this view, learning in-

volves adding or removing nodes or links to an existing network or modifying the weights on existing nodes or links. However, such background knowledge is not essential, and one can also construct an entire inference network entirely from the training data, although such methods can usually be characterized as starting with a degenerate network.

In either case, one can view learning as carrying out search through the space of inference networks. The operators used in this search can include actions for adding or removing nodes, adding or removing links, and altering the weights on links, and we will see examples of each in the sections that follow. Because they include nonterminal symbols, the space of inference networks (even logical ones, which include no weights) is much larger than the space of simple concept descriptions we considered in previous chapters. However, much of the work in this area assumes that the background knowledge is nearly correct and complete, so that relatively few steps are required to reach the target network.

Even when this assumption holds, learning still requires some search through the space of inference networks. Most work on the topic relies on a greedy or gradient-descent control structure that uses the entire training set to evaluate alternative changes to the network, although some incremental methods exist. Many approaches employ operators that can make the network either more specific or more general, but this is separate from whether they include operators for adding and removing structure from the network. The latter combination is associated primarily with algorithms for network revision, whereas methods for extending and constructing networks typically only add structure.

6.2 Extending an incomplete inference network

Perhaps the simplest approach to learning in the current framework is to extend an inference network that includes correct knowledge but remains *incomplete*. In particular, work in this framework typically assumes the network lacks the top-level links that connect the rest of the organization to the target concept. Thus, learning involves the induction of one or more 'rules' that specify the connections necessary for accurate classification.

Table 6-3. Disjunctive rules needed to accurately classify instances of `lethargia` without the nonterminal symbols `simploid` and `neoplasm` from Table 6-1.

```
lethargia :- one-tail, thin-wall, two-nuclei.
lethargia :- two-tails, one-nucleus, thick-wall.
lethargia :- two-tails, two-nuclei.
```

6.2.1 Extending a logical network

Most work on extending inference networks emphasizes the role of background knowledge, even when incomplete, in simplifying the induction task. In previous chapters we saw that the easiest induction tasks involve limited languages for representing concepts, such as logical conjunctions, linear threshold units, and simple probabilistic summaries. Unfortunately, many learning problems require more sophisticated representations and more complex induction algorithms.

The use of background knowledge stated as inference networks can alter this situation. For instance, reconsider the Horn clauses in Table 6-1, which characterize the concept `lethargia`. We can rewrite this description as the three rules in Table 6-3, which directly relate the top-level concept to observable features.[4] Clearly, one needs a reasonably powerful induction algorithm, such as NSC or NEX from Chapter 5, to acquire this description. A technique like HGS or HSG from Chapter 2 cannot handle disjunctive target concepts of this sort.

However, suppose we provide the learning system with background knowledge in the form of an incomplete inference network, in this case the last four rules in Table 6-1. Given training instances of the concept `lethargia`, the system can apply these rules to each case, generating an augmented instance description that may include the nonterminal symbols `simploid` and `neoplasm`. In this example, both of these higher-level features would be present in all positive instances of `lethargia`, but their combination would be absent in all negative instances. Thus, an algorithm like HGS or HSG could easily acquire the conjunctive description `lethargia :- simploid, neoplasm`. The presence of an ap-

4. We have omitted the fourth possible rule, `lethargia :- one-tail, thin-wall, one-nucleus, thick-wall`, because it contains two contradictory literals.

Table 6-4. A partial inference network, defining the intermediate concepts `odd-girth` and `odd-height`, which can be used as background knowledge to aid induction of the target concept `odd-size`.

```
odd-girth :- girth(G), 2 < G, G < 4.
odd-girth :- girth(G), 6 < G, G < 8.
odd-height :- height(H), 2 < H, H < 4.
odd-height :- height(H), 6 < H, H < 8.
```

propriate partial network has transformed a disjunctive induction task into a simpler conjunctive one.

This approach provides some additional benefits as well. Let us consider another example involving the height/girth domain. Table 6-4 presents a partial inference network that recognizes examples of two intermediate concepts. The first two inference rules state that a person has `odd-girth` if her `girth` falls either between 2 and 4 or between 6 and 8. (We assume that no person ever has `girth` less than 2 or more than 8.) The last two rules classify a person as having `odd-height` if her `height` has the same ranges. Figure 6-2 shows the extensions of these two concepts, along with training instances for the higher-level concept `odd-size`, which we will define as the conjunction of `odd-girth` and `odd-height`. Thus, the extension of `odd-size` corresponds to the area within the four shaded squares that occur at the intersections of the two intermediate concepts.

Suppose a logical induction algorithm observes the positive and negative training instances the figure shows in black. As before, one can augment the instance descriptions to include the features `odd-girth` and `odd-height` when their definitions are satisfied. Again, a simple conjunction of these two features suffices to discriminate the positive training cases (black circles) from the negative ones (black squares). Now suppose we present the performance component with the positive test cases shown in white. These correspond to regions of the extensional definition that the induction algorithm has never seen, yet the acquired concept correctly classifies them as positive instances of the `odd-size` concept. The presence of a partial inference network can increase accuracy on unseen parts of the instance space, thus improving

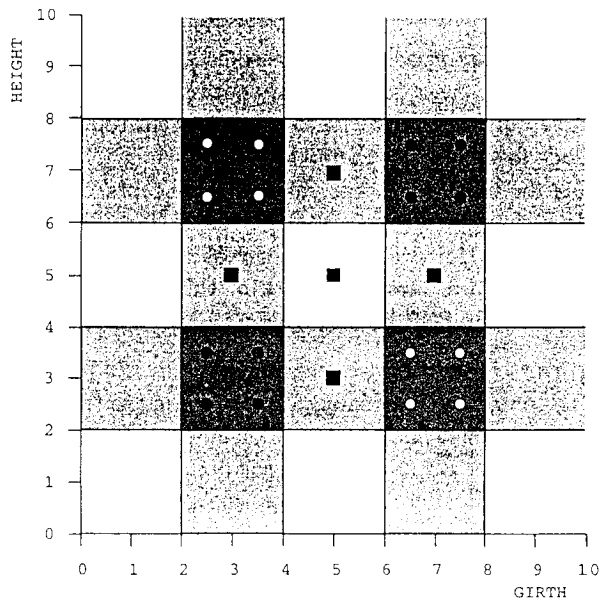

Figure 6-2. The presence of a partial inference network can increase learning rate. The two horizontal rectangles depict the extension of the `odd-height` concept defined in Table 6-4, the two vertical ones show the extension of `odd-girth`, and the four rectangles at their intersections give the extension of the target concept, `odd-size`. Small rectangles correspond to negative training cases and circles to positive ones. Training on the black instances, combined with the inference rules in Table 6-4, lets a simple conjunctive method induce a description that correctly classifies the unseen white instances.

the learning rate. This holds even for domains in which the partial network fails to eliminate the need for disjoint decision regions, and which thus require more powerful induction methods.

The above discussion assumes that the background knowledge is relevant to the current target concept. Naturally, inappropriate background knowledge will simply introduce irrelevant features, which will decrease the learning rate for any induction algorithm. Similar comments about incorrect or irrelevant structures hold for each of the methods we consider in this chapter, but we will typically assume that such structures are generally relevant and correct.

6.2.2 Extending a network of threshold units

Although the above approach is often associated with algorithms for inducing logical descriptions, it is certainly not limited to them. The same basic techniques apply equally well to other concept representations and to the methods for acquiring them. For example, in Chapter 3 we saw that methods for inducing linear threshold concepts can only master domains in which the classes are linearly separable. However, just as an incomplete set of Horn clauses can increase the power of conjunctive learning methods, so a partial network of threshold units can extend the range of threshold learning algorithms.

Let us return to the `lethargia` example. Figure 6-3 (c) shows the extension of this concept in dimensions corresponding to the three relevant attributes, which we treat here as the Booleans `one-tail`, `one-nucleus`, and `thick-wall`. One cannot separate the positive instances of this concept from the negatives by a single hyperplane, making the perceptron and LMS algorithms insufficient to induce it. But Figure 6-3 (a) and (b) show the extensions of the concepts `simploid` and `neoplasm`, which one *can* represent with single threshold units. Moreover, Figure 6-3 (d) gives the extension of `lethargia` in the instance space defined by the higher-level features `simploid` and `neoplasm`. In this context, the positive and negative instances of the concept are linearly separable.

The implication is that, given linear threshold units that correctly classify instances of `simploid` and `neoplasm`, one can use simple methods like the PCP and LMS algorithms to induce the `lethargia` concept. For instance, given background knowledge of the form

```
If −1.0 · one-tail + −1.0 · thin-wall > 0.5, then simploid

If −1.0 · one-nucleus + 1.0 · thick-wall > 0.5, then neoplasm
```

and training cases for the `lethargia` concept, these methods would induce a threshold unit along the lines of

```
If 1.0 · simploid + 1.0 · neoplasm > 1.5, then lethargia .
```

We have omitted the irrelevant features in each rule, which would be given zero weights in the first two cases and presumably would acquire small weights for the target concept.

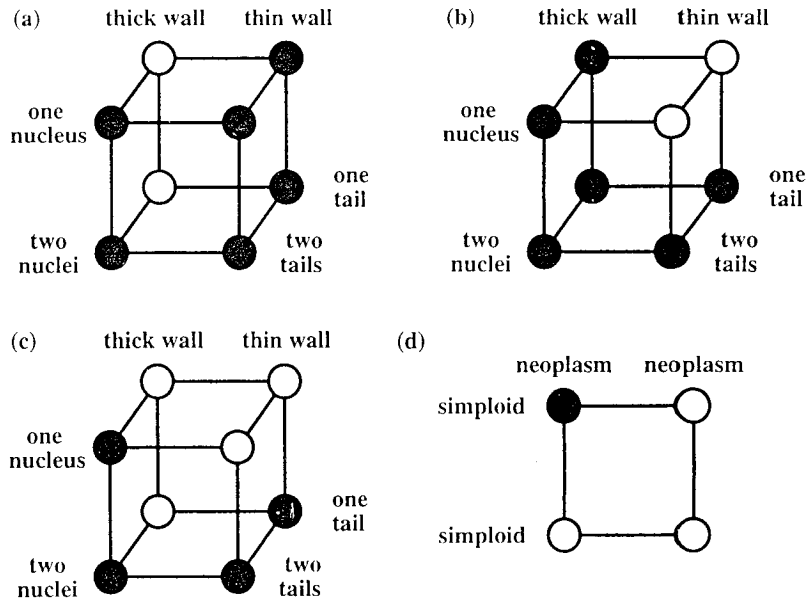

Figure 6-3. Extensional definitions of the concepts (a) `simploid`, (b) `neoplasm`, and (c) `lethargia` in the original instance space, ignoring the irrelevant feature involving cell color. Solid spheres represent positive instances of each concept, whereas open spheres indicate negative instances. Both (a) and (b) are linearly separable; this property does not hold for `lethargia` in the basic space, but (d) indicates that the concept is linearly separable in the augmented space that includes the `simploid` and `neoplasm` features.

Similar benefits occur even when the target concept is not linearly separable in the augmented instance space that includes the higher-level features. For example, suppose the extension of `lethargia` in Figure 6-3 (d) included two classes of instances – the conjunction of `simploid` and `neoplasm` and the conjunction of $\overline{\texttt{simploid}}$ and $\overline{\texttt{neoplasm}}$. Here one must invoke a more sophisticated algorithm for inducing threshold concepts, like NSC from Chapter 5, but this method should converge on the target considerably faster in the higher-order space than in the primitive one, just as should methods for inducing logical disjunctions.

6.2.3 Extending a probabilistic network

In Chapter 4 we encountered simple Bayesian classifiers, another induction algorithm that can acquire only a limited class of concepts. In this case, the learning method assumes the independence of attributes or features for instances within each class. As in the logical and threshold frameworks, one can use a partial probabilistic network to extend the power of this simple algorithm. Note that a probabilistic inference network represents exactly the correlations (stated as probabilistic causal links) that a Bayesian classifier assumes are absent. The probabilities on a link from attribute A to attribute B are *conditionally* independent of any attributes that affect A, and one can use connections in the network to factor out dependencies in a training set.

In particular, suppose one is given a probabilistic inference network that relates attributes A_1 through A_n, along with training cases of concept C that uses these attributes. Before using the training set to compute $P(A_k|C)$, one first runs the training instances through the network, producing a revised set that factors out nonindependence due to links that connect attributes. One then runs a simple Bayesian classifier on the modified data to calculate $P(A_k|C)$ for each attribute k, which together specify the probabilities on links from the concept node C to each attribute. If the initial network accurately represents the correlations among attributes, the revised training set will satisfy the independence assumption and the algorithm will have no difficulty.

If the probabilistic network specifies only some of the actual connections among the attributes, this approach should still aid a more sophisticated algorithm. By factoring out correlations between some attributes, it should let a method for inducing competitive disjunctions, like NCD from Chapter 5, acquire the target concept in fewer training instances than if this knowledge were not available.

6.3 Inducing specialized concepts with inference networks

An alternative approach to learning with inference networks assumes that one's knowledge base includes a complete and correct network for some concept C, but that the target concept T is a specialization of C. Moreover, it assumes that the intensional definition of T uses only those primitive features that occur in the definition of C, so the inference network for the latter (C) can be used to bias learning of the former (T).

Table 6-5. A revised version of the inference network from Table 6-1 that includes a rule for recognizing instances of the disease `doldroma`. This concept is a specialization of `lethargia` that makes use of the nonterminal symbol `neoplasm` but not `simploid`.

```
lethargia :- simploid, neoplasm.
simploid :- one-tail, thin-wall.
simploid :- two-tails.
neoplasm :- one-nucleus, thick-wall.
neoplasm :- two-nuclei.
doldroma :- two-tails, neoplasm.
```

This approach is often referred to as *explanation-based generalization*, but this term covers a variety of methods, some concerned with improving the efficiency of problem solving, which we discuss in Chapters 10 and 11. Here we are concerned with the improvement of classification accuracy, and in this context the term *theory-guided specialization* seems more appropriate.

6.3.1 Induction by adding compiled explanations

Consider the intensional definition of the concept `doldroma` given in Table 6-5, which is a specialization of the `lethargia` concept we studied in the previous section. The new concept makes use of the nonprimitive feature `neoplasm` but not `simploid`, replacing this latter term with the primitive feature `two-tails`. The result is that the instance space contains only six cases of `doldroma`, a subset of the eight `lethargia` instances.

However, there exists a simple mechanism for using the inference network to acquire the target concept `doldroma`, provided one is willing to assume its special case relation to `lethargia`. We will call this algorithm ICE, as it involves induction through the compilation of explanations. Briefly, given a positive training instance I of the target concept T, ICE uses the inference network to prove that I is a positive instance of the more general concept C. The method then takes the features that occur as terminal nodes in the proof tree and makes them the antecedents of a rule for which the target concept T is the

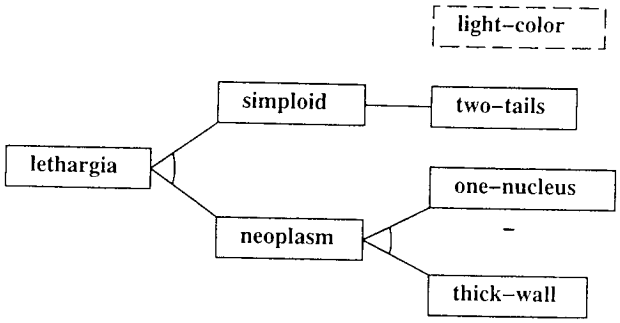

Figure 6-4. A proof tree showing that a training case described by the literals on the far right is a positive instance of the concept `lethargia`, using the inference network from Table 6-5. Literals that occur as terminal nodes in the AND tree become the antecedents of a new inference rule for the target concept, a specialization of `lethargia`.

consequent. The ICE algorithm repeats this process for each positive training instance, creating another rule in each case unless an equivalent one already exists.

Figure 6-4 illustrates the proof (shown as an AND tree) that a particular instance constitutes an example of the concept `lethargia`. In this case, the basic features include `two-tails`, `one-nucleus`, `thick-wall`, and `light-color`. The AND tree decomposes `lethargia` into `simploid` and `neoplasm` using the first rule in Table 6-5. It then uses the third rule to rewrite `simploid` as `two-tails`, which matches a literal in the instance description, and uses the fourth one to decompose `neoplasm` into `one-nucleus` and `thick-wall`, which also hold for the instance.

After generating this proof, ICE collects the literals that occur as terminal nodes[5] and uses them to construct a new inference rule. In this case, the result is

```
doldroma :- two-tails, one-nucleus, thick-wall.
```

in which the consequent refers to the target concept, `doldroma`, rather than to the more general concept `lethargia` used in the proof. Note that this rule makes no mention of the feature `light-color`, since it

5. In this chapter we focus on domains that one can describe using propositional clauses, which require no arguments. In Chapter 8 we consider the issues of variable binding and unification that arise with more powerful representations.

did not occur as a terminal node in the AND tree (due to its absence in the rules that define `lethargia`). As a result, the rule covers two cases in the instance space, giving some transfer beyond the training case on which it is based.

Of course, the above trace constitutes only the first step in the overall process; ICE must repeat the process for each new training instance encountered.[6] Thus, given another positive case of `doldroma` in which the literal `two-nuclei` was present, the algorithm would produce a different proof tree and the associated inference rule

```
doldroma :- two-tails, two-nuclei.
```

Note that this structure omits information about both the color and wall of the cell, letting it cover four cases in the instance space.

Taken together, these two inference rules have the same extension as the definition for `doldroma` given in Table 6-5. Thus, in a noise-free environment, we would not expect to observe any positive instances that would produce additional rules for this concept. Nor does this scheme require any negative instances to detect problems with the existing hypothesis, provided the specialization assumption is correct. The presence of noise complicates matters, in that some negative cases that have been falsely labeled as positive may be examples of the general concept but not the target. To handle this possibility, one can modify ICE to collect statistical evidence for each learned inference rule and require sufficient confidence before adding each to the network.

6.3.2 Induction over explanation structures

The above algorithm is only one among many that take advantage of a complete domain theory for one concept to bias learning about another. Returning to the terminology of Chapter 2, the approach begins with a very specific extension for the target concept (the empty set) and gradually broadens it, making the definition more general by adding new disjunctive rules. However, the existence of an inference network also suggests another method for moving toward a more general description of the target concept – replacing some of the terminal literals in learned rules with nonterminal symbols that appear in the rules' proofs.

6. The common claim that explanation-based methods "learn from a single instance" is misleading at best. Each instance does produce an inference rule with some generality, but this will seldom cover the entire target concept. More precisely, such methods learn one disjunctive rule from each positive training case.

For instance, given the two acquired doldroma rules we saw earlier, one might naturally collapse these into the single rule

```
doldroma :- two-tails, neoplasm.
```

since the term neoplasm occurred as parent of the literals one-nucleus and thick-wall in one proof and as the parent of two-nuclei in the other. In this case, the new rule has the same extension as the two original ones, but in situations where the nonterminal literal can be decomposed in ways not yet observed in the training data, the same operation can alter predictions on novel test instances.

This algorithm, which we will call IOE (induction over explanations), will gradually move up the inference network in some areas and retain terminal literals in others, depending on the training data it encounters. In some domains, it will even decide that the target concept is equivalent to the original concept defined by the inference network. For the current network, this might occur if IOE observed an instance of doldroma with one tail and a thin wall in addition to those given earlier. In this case, the method might replace the more specific rules with

```
doldroma :- simploid, neoplasm.
```

since all three possible explanation structures for the lethargia concept also appear to be valid for the target concept.

We have outlined a method that gradually adds compiled rules and then replaces them with more general rules as it gains experience. However, such an incremental hill-climbing strategy can lead to overly general hypotheses. A more robust algorithm would also include an operator for replacing nonterminal symbols with their decompositions if a negative instance gives evidence of overgeneralization. Such a technique would be equivalent to the ISC algorithm from Chapter 5, except that its search would be biased by the inference network that it uses as background knowledge.

Of course, one need not rely on incremental hill climbing to use an inference network to bias induction, nor need one first construct rules that contain only ground literals and then replace them with nonterminal symbols.[7] One can collect all observations at the outset and invoke a

7. Work in this framework is often called *multiple explanation-based generalization* to distinguish it from the simpler variety in which each disjunct is based on a single training case.

nonincremental algorithm instead, using the entire training set to compute evaluation metrics for deciding whether to collapse rules and move through the network. One can also start with the most general possible hypothesis (the definition of lethargia in our example), and invoke operators that replace nonterminal symbols with their decompositions as negative training cases reveal the need for such actions. Methods for the induction of specialized concepts using an inference network run the same gamut of variations that we saw in Chapter 2.

Historically, the above approach to induction has been almost exclusively associated with logical representations of knowledge. In principle, one might use an inference network composed of threshold units or competitive descriptions to bias learning of a specialized target concept, but few examples of such approaches exist in the literature.

6.4 Revising an incorrect inference network

A third approach to learning with inference networks assumes that one has some initial knowledge about the target concept C, but that the network for C is only partially correct. The knowledge may be overly specific, in that it classifies some positive instances as negative, or it may be overly general, in that it classifies some negative cases as positive. The task confronting the learner involves modifying the inference network so that it will accurately classify test cases. Work in this framework assumes that one already knows the *nodes* appropriate for the network, and that learning only involves modifying the *links*. This problem is often referred to as *theory revision*, especially when applied to logical formalisms.

6.4.1 Revision of logical theories

Table 6-6 shows a mutilated version of the inference network for lethargia from Table 6-1. Inspection reveals three differences from the correct network. First, the second simploid rule contains an extra literal, thick-wall, which makes the inference network overly specific so that it misclassifies some positive instances. Second, one of the neoplasm rules is absent, which also causes the network to label some positive cases as negative. Finally, the remaining neoplasm rule lacks the literal thick-wall, which causes the network to classify some negative cases

Table 6-6. An incorrect inference network for the concept `lethargia`. This version differs from the one in Table 6-1 by an omitted `neoplasm` rule, an omitted literal in the remaining `neoplasm` rule, and an unnecessary literal in one of the `simploid` rules.

```
lethargia :- simploid, neoplasm.
simploid :- one-tail, thin-wall.
simploid :- two-tails, thick-wall.
neoplasm :- one-nucleus.
```

as positive. An extra rule would also produce an overly general network, although none are present in this example.

These four sources of error in inference networks suggest four natural learning operators for correcting them. Clearly, the learner can broaden the extension of a network either by removing a literal from an existing rule or by creating an entirely new rule. Similarly, it can reduce an inference network's extension either by adding a literal to an existing rule or by completely removing a rule. Taken together, these operators can transform any inference network into any other inference network involving the same nodes, moving through a space that is larger than that for DNF expressions. If one is willing to assume that the initial theory is *nearly* correct, the problem is not so daunting, but it still requires search through the space of networks, and effective search requires some constraints or heuristics to direct the process.

An obvious constraint involves the terms used in the network. In addition to limiting attention to known terms, one can provide the learner with a partial ordering, with the class name at the top and literals from the instance language on the bottom. In the current example, one might inform the learner that `lethargia` can be decomposed into any other terms, `simploid` and `neoplasm` can be decomposed into any terms except each other and `lethargia`, and none of the instance features can be further decomposed. This knowledge limits the literals the learner can add to an existing rule. For instance, given the network in Table 6-6, it ensures that this operator does not consider adding `lethargia`, `simploid`, and `neoplasm` to any rules. In other situations, the constraint will forbid adding a literal to some rules but not to others.

Table 6-7. Behavior of different inference networks on training data for the `lethargia` concept, ignoring the color of the cell. Column (a) indicates whether each instance is actually positive (+) or negative (-), with column (b) showing the predictions made by the faulty network in Table 6-6. The remaining columns depict predictions (c) after the RLT algorithm has added the rule `neoplasm :- two-nuclei`, (d) after it has removed `thick-wall` from the second `simploid` rule, and (e) after it has added `thick-wall` to the first `neoplasm` rule.

Instance description	(a)	(b)	(c)	(d)	(e)
one-tail, one-nucleus, thick-wall	-	-	-	-	-
one-tail, one-nucleus, thin-wall	-	+	+	+	-
one-tail, two-nuclei, thick-wall	-	-	-	-	-
one-tail, two-nuclei, thin-wall	+	-	+	+	+
two-tails, one-nucleus, thick-wall	+	+	+	+	+
two-tails, one-nucleus, thin-wall	-	-	-	+	-
two-tails, two-nuclei, thick-wall	+	-	+	+	+
two-tails, two-nuclei, thin-wall	+	-	-	+	+

One algorithm for revision of logical theories (RLT) relies on greedy search directed by an evaluation function. At each step in the iterative process, the method considers all possible ways (within the allowed constraints) of modifying the theory. RLT then evaluates each network in terms of its ability to correctly classify the training data, possibly taking other factors like simplicity into account. Removing literals from existing rules and deleting entire rules are straightforward operations, and we have outlined a scheme that limits the addition of literals. The simplest approach to creating an entirely new rule is to generate all allowed rules that contain only one literal per antecedent, then to rely on other operators for adding more literals if needed. After selecting the revision that leads to the greatest improvement, RLT cycles, continuing the process until no alternative scores as well as the current theory.

Consider the sample data presented in Table 6-7, which shows the actual classes (a) of eight instances[8] from the `lethargia` domain and the classes predicted (b) by the inference network in Table 6-6. Suppose that the evaluation function used to direct the theory revision process

8. For the sake of clarity, we will ignore the existence of features involving the cell color, giving only eight instances in the domain. A more realistic training set would include only some of the possible instances from the domain.

simply measures accuracy on the training set, giving $\frac{4}{8}$ as the score for the initial network. Given this theory, the RLT algorithm would consider all possible ways of adding rules, removing rules, adding literals, and removing literals.

In this case, the best choice involves adding the rule `neoplasm :- two-nuclei`, which gives the score $\frac{6}{8}$ and the predictions in (c). After making this alteration to the network, RLT repeats the process. This time the best action involves removing the literal `thick-wall` from the third rule in Table 6-6, which gives the score $\frac{6}{8}$ (no better than the current theory) and a strictly overgeneral network. Finally, the best change to this structure involves adding `thick-wall` to the fourth rule in Table 6-6, which gives an accuracy of $\frac{8}{8}$ and produces the target theory from Table 6-1.

Clearly, this example sidesteps some important issues. One can design training sets that lead the RLT scheme into local optima from which it cannot recover. However, this problem is common to all greedy methods, and one can reduce its likelihood in exchange for additional search. We have also ignored the possibility that the training data contain noise, which can suggest the addition or removal of inappropriate rules or literals. Earlier chapters considered some techniques for minimizing the effect of noise, and one can adapt these methods to the revision of inference networks. In addition, one can incorporate a bias toward simplicity into the evaluation function to reduce the chances of generating a complex theory that overfits the data.

Incremental approaches to logical theory revision are also possible. In this scheme, incorrect classification of a new training case invokes the modification process. If the network labels a positive instance as negative, one attempts to generalize the theory by creating a new rule, typically encoding the entire training case, or by removing those literals that kept a rule from matching the instance. Identifying such literals is closely related to the problem of abduction we mentioned in Section 6.1.2. If the network classifies a negative case as positive, one tries to specialize the theory by removing the responsible rule or by adding literals that will keep it from matching. This approach assumes an incremental hill-climbing process, in which decisions made early in learning may be overturned later as new evidence becomes available. Obviously, a useful evaluation metric or some other constraint on search is as important here as for a nonincremental method.

Table 6-8. A multilayer threshold network for the `lethargia` concept that makes the same predictions as the logical network in Table 6-6. Each rule only shows the features with nonzero weights, but features with zero weights are also present. In this framework, theory revision involves altering weights rather than adding and deleting rules and literals.

```
If  1.0 · simploid + 1.0 · neoplasm > 1.5, then lethargia .
If  1.0 · simploid1 + 1.0 · simploid2 > 0.5, then simploid .
If  1.0 · neoplasm1 + 1.0 · neoplasm2 > 0.5, then neoplasm .
If  1.0 · one-tail + 1.0 · thin-wall > 1.5, then simploid1 .
If -1.0 · one-tail + -1.0 · thin-wall > 1.5, then simploid2 .
If  1.0 · one-nucleus > 0.5, then neoplasm1 .
If  > 0.5, then neoplasm2 .
```

6.4.2 Revision of threshold networks

Naturally, one can also revise inference networks that are composed of threshold units. In this framework, a network can have the desired structure but still make faulty predictions because its weights are incorrect. For example, consider the threshold network in Table 6-8, which is equivalent to the logical network in Table 6-6. This includes three levels or layers (not counting the primitive features), with the top layer responsible for recognizing instances of `lethargia`, the middle layer handling `simploid` and `neoplasm`, and the lowest layer noting `simploid1` and `simploid2` (the two types of `simploid` cells), as well as `neoplasm1` and `neoplasm2` (the two forms of `neoplasm` cells). One could easily add another layer on top to handle multiple forms of `lethargia`. Note that the `neoplasm2` rule lacks inputs, yet it is necessary for the revision process to approximate the `lethargia` network in Figure 6-1 (b).

Each rule in this network includes every possible feature at the next lower level, although the table omits features with 0 weights. Thus, the structure of the network subsumes the structure necessary to correctly describe the `lethargia` concept; the only problem resides in the assigned weights. As a result, the natural theory revision operators for the threshold networks involve altering the weights on links. In some cases, such modifications have the same effect as adding or removing rules or literals. However, because any given threshold unit can rep-

resent a continuum from AND to OR, weight changes often have much more complex effects.

The most commonly used algorithm for modifying weights in a network of threshold units is known as *backpropagation*, which generalizes the LMS technique that we encountered in Chapter 3. Backpropagation carries out an incremental gradient descent, revising the weights in the network in response to each training instance.[9] For each node in the network, it first computes the level of activation for the given training case. Next, for each of the top-level nodes, it compares the desired activation (usually 1 or 0) with that predicted for the instance.

Using the LMS algorithm, backpropagation modifies the weights leading into the top-level node j to reduce the difference between the desired and the predicted value. In particular, it computes

$$\Delta w_{ij} = \eta \delta_j x_i'$$

as the change in the weight from node i to node j, where x_i' is the activation of node i, $0 < \eta < 1$ is a constant gain factor, and the error term for node j is

$$\delta_j = p_j(1 - p_j)(d_j - p_j) \quad ,$$

where d_j is the desired output for top-level node j and p_j represents its prediction on the training case.

After calculating the weight changes for the top level of the network, backpropagation proceeds to the next layer, using the δ values computed at the previous level to alter the weights leading into it. In particular, it uses

$$\delta_j = x_j'(1 - x_j') \sum_k \delta_k w_{jk} \quad ,$$

where x_j' is again the prediction of internal node j, w_{ij} is the weight from node j to i, and the summation occurs over all k nodes immediately above node j in the network. Backpropagation uses this scheme to modify weights at each level in turn, until it reaches the input nodes associated with primitive features. As usual, the values for these units are either 1 (if present in the instance) or 0 (if absent), and thresholds

9. Like many other methods for modifying weights in threshold units, backpropagation is often run through the same training set multiple times until it converges on some global or local optimum, giving the effect of a nonincremental scheme.

are simulated by special nodes with the constant value 1. Most work on the backpropagation algorithm assumes the use of a logistic function, which we discussed in Section 6.1, to compute node outputs, but similar algorithms are possible for use with other performance methods.

Although backpropagation is the most commonly used algorithm for weight modification in threshold networks, many other techniques also exist. Most such methods also carry out gradient descent through the weight space, trying to minimize some error metric on the training set. Some work has explored more sophisticated search techniques designed to avoid problems with local optima, but a more common response is to run the learning algorithm multiple times with different starting conditions, then select the resulting network with the smallest error.

This raises another issue about work with multilayer threshold networks. Although most research on this topic requires the programmer to specify the nodes and links that constitute network structure, the initial weights on links are typically assigned in a random fashion. We have ignored this trend in order to emphasize the similarity between work on revising networks of threshold units and on theory revision in logical networks. Moreover, some recent work in the former framework does let one specify initial weights based on knowledge of the domain. But most approaches to learning with threshold units adopt the random assignment scheme to initialize the weights and thus the extension of the initial network.

6.4.3 Revision of probabilistic networks

Research on the revision of probabilistic networks has been less common than for logical and threshold networks, but many of the same issues arise. Given the assumptions of conditional independence we discussed in Section 6.1, one can use methods based on probability theory to estimate the conditional probabilities on links for a specific network structure on a giving training set. Thus, the main problem involves determining which of the possible links between pairs of attributes should be included in the network and which should be omitted.

Fortunately, well-defined methods also exist for computing the probability of a particular network structure, given that one has observed a particular set of training cases. An induction algorithm can use the resulting probability as an evaluation function to direct search through the space of probabilistic network structures. However, the size of this

space encourages the introduction of constraints. One approach is to place a complete ordering on the attributes, such that there can be a link from an attribute high in the ordering to a lower one, but not vice versa. This constraint also ensures a clear causal semantics for the resulting networks.

Two natural operators arise in modifying a probabilistic network – removing an existing link or adding a new link. Since one can compute the best weights for a given structure, the only issue involves which links to eliminate and which to create. Perhaps the simplest approach uses a greedy technique that considers each possible addition and deletion, computes the probability of each alternative network given the training data, selects the most probable network, and repeats this process until no further improvement occurs. This scheme assumes one starts with an initial network that must be revised. However, one can use a similar method when no such probabilistic network is available. In this case, one can either start with the empty network, considering only the addition of links on each cycle, or start with the complete network, considering only the removal of links on each pass. As with the other frameworks, alternative search schemes are possible that reduce the chances of converging on a local optimum.

6.5 Network construction and term generation

In Sections 6.2 and 6.3, we assumed that the learner began its career with background knowledge stated in the form of an inference network. We relaxed this assumption in Section 6.4 by letting the learning process determine the links connecting the terms in such a network. Now we are ready to examine the more difficult problem of creating these terms.

The simplest approach to introducing new terms also makes the greatest demands on the developer. Briefly, if one wants the learner to acquire a network for concept C that includes intermediate terms for concepts $T_1, T_2, \ldots, T_k$, one first trains the learner on these intermediate concepts, specifying their names in each case. Thus, one would give the learner positive and negative instances of T_1 until it masters this concept, then provide it with instances of T_2, and so forth. If some terms are situated lower in the network than others, the learner must acquire the lower ones first.

One can use this strategy in conjunction with any of the representational formalisms we have examined, and it provides a practical interactive approach to building complex inference networks. However, from a theoretical perspective, it is much more interesting to require the learner to create new terms on its own initiative. This task goes by a variety of names, including *term generation*, *representation change*, and *constructive induction*. Much of the work on this problem attempts to construct an inference network from scratch, but it can also extend a partial network. As usual, we will consider induction methods that arise within a number of representational frameworks.

6.5.1 Generation of logical terms

The reader may have noted that Horn clauses have many similarities to grammars, with nonterminal symbols corresponding to parts of speech and phrasal classes, and with clauses corresponding to phrase definitions. Thus, one can view the task of term generation for a logical inference network as equivalent to the task of grammar induction from sample sentences. One difference between the two is that ground literals in the training instances are typically unordered sets rather than lists of words. Another is that negative training cases may be present to rule out overly general networks, whereas grammar induction typically assumes only legal sentences.

According to this view, there are two natural learning operators. The first creates a nonterminal symbol along with an associated rewrite rule that decomposes it into its constituents. This should be useful when certain combinations of literals tend to co-occur in positive instances of the training set. The second operator involves merging symbols, either terminal or nonterminal, into disjunctive classes. This should be useful when a set of symbols tends to occur in similar contexts. For domains in which one knows that literals fall into mutually exclusive sets (i.e., attributes), one can constrain new phrases to combine only terms of different types and new classes to include only terms of the same type.

Let us consider GLT (generation of logical theories), a nonincremental algorithm for inducing a logical inference network from positive and negative training cases. The method starts with a 'flat' network that contains only the class name and terminal symbols, with one inference rule for each positive case. Naturally, this network covers all positive and no negative training instances. The algorithm operates in a greedy

fashion. GLT starts in 'term creation' mode, in which it considers all legal ways of pairing existing symbols (except the class name) to define a new term.

After eliminating redundant rules in each inference network and removing unacceptable candidates, including networks that cover negative training cases, the algorithm uses an evaluation function to select the best alternative. If the score of the resulting network is good enough, GLT revises the network and repeats the process. If not, it changes to 'term merging' mode and considers all legal ways of merging pairs of existing terms, using the same evaluation metric to select the best candidate. If the result is good enough, GLT revises the network and iterates; otherwise it switches back to term generation. This alternation between term creation and merging continues until neither mode meets the criteria for continuation.

Table 6-9 shows the behavior of GLT on a variant of the cell domain that allows three values for each attribute, using the complexity of a network (its total number of symbols) as the evaluation metric.[10] The initial inference network includes nine instances of the class `somnogenic`, each described as a conjunction of four literals, giving 45 as the evaluation score. The algorithm begins by considering all new terms that it can construct from legal pairs of the primitive terms. The second part of the table shows the result after six iterations, a network in which there are six new rules defining nonterminal symbols (`A` through `F`) and in which these terms have replaced the combinations of primitive terms in the original rules. The complexity of this network is 45, the same as the original one.

Beyond this stage, additional terms only increase grammatical complexity, worsening the score, so GLT turns its attention to merging terms. The third part of Table 6-9 shows the network after two more iterations, during which the algorithm merges the terms `D` and `E` into the class `deprivitic`, and then adds `F` to this class. These actions lead to a reduced complexity of 27, since after substituting class names back into the `somnogenic` rules, only three unique ones remain. The final part of the table shows the network that results from two additional merges, creating the class `hypnagogic` from `A` and `B`, to which GLT then adds the phrasal term `C`. The new complexity is only 21, since substitutions have

10. Here we assume that an evaluation score S is 'good enough' to let GLT continue in the current mode if S is equal to or less than the current score.

Table 6-9. The GLT algorithm's transformation of a grounded inference network (initial group) into one involving nonterminal symbols. The second network introduces the symbols `A`, `B`, `C`, `D`, `E`, and `F` to represent different recurring pairs of ground literals; the third network eliminates some rewrite rules by replacing `D`, `E`, and `F` with `deprivitic`, and the final group eliminates more rules by replacing `A`, `B`, and `C` with `hypnagogic`.

```
somnogenic :- one-tail, thick-wall, one-nucleus, light-color.
somnogenic :- one-tail, medium-wall, one-nucleus, medium-color.
somnogenic :- one-tail, thin-wall, one-nucleus, dark-color.
somnogenic :- two-tails, thick-wall, two-nuclei, light-color.
somnogenic :- two-tails, medium-wall, two-nuclei, medium-color.
somnogenic :- two-tails, thin-wall, two-nuclei, dark-color.
somnogenic :- three-tails, thick-wall, three-nuclei, light-color.
somnogenic :- three-tails, medium-wall, three-nuclei, medium-color.
somnogenic :- three-tails, thin-wall, three-nuclei, dark-color.
```

```
somnogenic :- A, D.          A :- one-tail, one-nucleus.
somnogenic :- A, E.          B :- two-tails, two-nuclei.
somnogenic :- A, F.          C :- three-tails, three-nuclei.
somnogenic :- B, D.          D :- thick-wall, light-color.
somnogenic :- B, E.          E :- medium-wall, medium-color.
somnogenic :- B, F.          F :- thin-wall, dark-color.
somnogenic :- C, D.
somnogenic :- C, E.
somnogenic :- C, F.
```

```
somnogenic :- A, deprivitic.
somnogenic :- B, deprivitic.
somnogenic :- C, deprivitic.
A :- one-tail, one-nucleus.
B :- two-tails, two-nuclei.
C :- three-tails, three-nuclei.
deprivitic :- thick-wall, light-color.
deprivitic :- medium-wall, medium-color.
deprivitic :- thin-wall, dark-color.
```

```
somnogenic :- hypnagogic, deprivitic.
hypnagogic :- one-tail, one-nucleus.
hypnagogic :- two-tails, two-nuclei.
hypnagogic :- three-tails, three-nuclei.
deprivitic :- thick-wall, light-color.
deprivitic :- medium-wall, medium-color.
deprivitic :- thin-wall, dark-color.
```

left only one unique somnogenic rule. After this, neither term merging nor term creation leads to networks of equivalent or lower complexity, causing the algorithm to halt.

This example ignores a number of important issues. First, the final inference network does not move beyond the initial training data, in that it classifies the initial nine cases as positive and no others. However, the algorithm can produce such networks from alternative training sets, provided that they contain enough co-occurrences of terms to reduce the complexity score. The example also ignores the role of negative cases in pruning the space of alternative networks, which becomes important precisely when the learning operators produce hypotheses that move beyond the training data.

Incremental approaches to term generation are also possible. For example, upon observing the two positive instances `S :- A B D E` and `S :- A C D F`, one natural response would be to replace them with three rules, involving two nonterminal symbols (`S :- X Y`, `X :- A D`, `Y :- B E`, and `Y :- C F`) that reflect the terms held in common by the two instances (`X`) and those in which they differ (`Y`). Work on this topic is sometimes referred to as *inverse resolution*, because it involves generating clauses that would resolve with the original instances if only they were present.[11] The problem with this approach, as with most incremental methods, lies in directing search through the space of hypotheses with only a small subset of the training data. As we have seen, one solution is to include additional learning operators that support bidirectional search and a hill-climbing strategy. In the current framework, this means adding operators for removing and splitting terms, thus producing structures with fewer nonterminal symbols.

6.5.2 Generation of threshold units

The possibility of term generation also exists with networks of threshold units. Most methods of this sort carry out a nonincremental greedy search that starts with a simple network and introduces one node at a time into the network. However, they differ in the structure of the resulting network, in the order of node creation, and in whether training occurs at only one level or all of them.

11. Most work on inverse resolution deals with more powerful representations that support predicates with variables, typically using true Horn clauses.

One algorithm, which we call TSC (threshold separate-and-conquer), embodies a strategy similar to that of NSC in Chapter 5. The method begins with a single-layer threshold network with one output node, *N1*, for concept C. If training reveals the data are linearly separable, no further action is taken, but failure to correctly classify all training cases suggests a need for additional units in the threshold network. If the initial network misclassifies some negative instances as positive, TSC adds two new nodes. One of these, *N2*, occurs at the same level as *N1*, and is trained only on those training cases that *N1* classified as positive. Both *N1* and *N2* are connected to a second new node, *A1*, with link weights of 1 and threshold 1.5 that effectively specify the conjunction *N1* $\wedge$ *N2*. No training occurs for this AND node, only for those at the first level. If node *A1* still misclassifies some negative cases, TSC repeats the process, creating another node, *N3*, at the first level, connecting it to *A1* with weight 1 and increasing *A1*'s threshold by 1 to keep it conjunctive. This process continues until *A1* covers no negative cases.

However, the addition of a first-level node *Nk* may also have excluded some positive training cases. When this occurs, TSC creates another second-level node, *A2*, with *N1* through *Nk* as its initial inputs. As with the first AND node, the threshold on *A2* is $k-0.5$, but the weights are all -1. This effectively computes the conjunction $\neg N1 \wedge \ldots \wedge \neg Nk$, making the node active only when none of its inputs are present. The algorithm then takes all training instances covered by this node and repeats the process above, creating and training additional first-layer nodes and adding them as inputs to *A2*, this time with weights of 1, until *A2* covers only positive cases. If more positive instances remain uncovered, TSC repeats the initial strategy, and so forth. When this process is complete, there exists a two-layer threshold network, with AND nodes at the second level, that correctly classifies the entire training set.[12] Figure 6-5 depicts a network constructed in this manner, with four units in the first level and two in the second layer.

Other methods for constructing multi-layer networks are also possible. For instance, another algorithm begins with a two-layer network having only one node in the first (hidden) level. However, rather than storing only weights, each link also has a variance that reflects the amount of change it has undergone during training with a method like backpropa-

12. Some versions of this method include a single OR node in a third layer with threshold 0.5 and weights of 1 on links from the AND nodes. However, this is not necessary if one simply lets each AND node predict the class C.

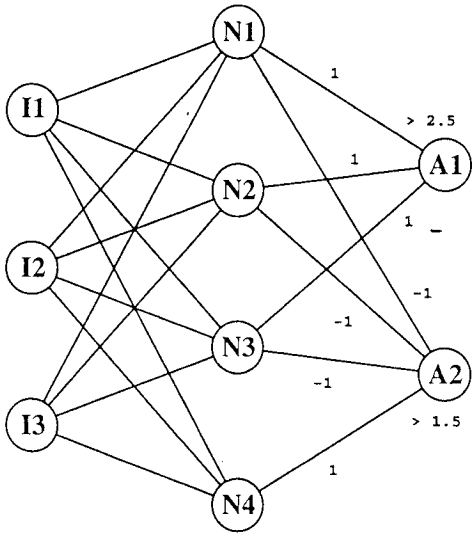

Figure 6-5. A multilayer network constructed by the TSC algorithm, with the second layer ($A1$, $A2$) defining conjunctions of the threshold units in the first layer ($N1, \ldots, N4$).

gation. If the total variance on a hidden node's links is high enough, the algorithm splits the node into two siblings with the same weights but half the variance. Continued training may reduce the variance on their links, but if this increases again, the splitting process continues, gradually increasing the number of hidden nodes until the network achieves a good fit to the training data and the weight changes stabilize.

Yet another method constructs a 'cascade' network, which may contain an arbitrary number of layers, but allows only one node at any layer but the inputs. This *cascade correlation* technique starts with a single-layer network with output node $N1$, but if training on this structure does not reduce errors to an acceptable level, the algorithm creates another node, $N2$, with incoming links from the input nodes and with an outgoing link to $N1$. The aim is to use the new node to predict the *errors* in the original network, and then to subtract them out.

To this end, cascade correlation sets the weight on the link connecting $N2$ and $N1$ to -1, fixes the weights on the original links to $N1$, and trains the new portion of the network (the weights into $N2$) on the difference between the desired outputs and the original network's predictions.

Training continues until these weights appear to have converged and, if sufficient errors remain, the algorithm introduces another node, $N3$, with incoming links from $N2$ and the input units, and with an output link (weighted -1) into $N1$. Again, the method fixes all but the new weights, which it trains on the remaining errors. This process continues until the scheme generates a network structure, and associated weights, that are sufficiently accurate on the training data.

6.5.3 Generation of probabilistic terms

Term generation for a probabilistic inference network raises the same basic issues as in the threshold and logical frameworks. Here the aim is to determine the number and location of 'latent' variables or attributes that mediate effects among observable attributes. Given two correlated attributes x and y, the methods outlined in Section 6.4.3 would consider the hypotheses that x directly influences y and that y influences x. However, they would not entertain the possibility that neither directly influences the other, and that an unobserved attribute, z, influences both and thus explains their correlation.

No great effort is required to modify algorithms for revising a probabilistic network to include an operator for introducing new latent attributes. As in the other frameworks, the problem is that there exist many such terms. Thus, one needs some evaluation metric to select the best term, to determine whether creating this term is more desirable than modifying the links among existing terms, and to decide when no additional terms are required. One also needs some means for estimating the value of a latent term for each training instance from the values of known attributes.

Research on this topic remains in the early stages. Methods for identifying latent variables in quantitative data exist, but they rely on an assumption of linear relations between variables in the network. Alternative techniques have been developed for nominal domains, but these methods typically assume the probabilistic network has a tree structure and they appear to be sensitive to noise. The development of robust methods for term generation in probabilistic, threshold, and logical networks remains an open and important problem for future research.

6.6 Summary of the chapter

In this chapter, we examined a variety of methods for refining and extending inference networks. We saw that this formalism can organize logical, threshold, and competitive concepts into knowledge structures that, when combined with particular inference methods, can produce complex decision regions. We also found that much of the learning work on this topic starts with some initial inference network and uses training data to modify the structure, rather than creating it from scratch.

The simplest task of this sort involves extending an *incomplete* network, in which the existing portions are correct but which are not connected to the node for the class attribute. In this case, one can use the existing network to redescribe the training instances at a higher level of description, then use an induction algorithm to formulate some connection between existing nodes and the node for the target concept. We noted that the presence of such background knowledge can transform a difficult induction task involving multiple decision regions into a easier one involving only one region per class. As a result, one can use simple induction methods like HSG, PCP, and the simple Bayesian classifier to good effect.

Another task involves learning a *specialization* of some concept for which a complete network is available. We considered one such method, ICE, that uses the inference network to construct a proof for each positive instance, then compiles the proof into a single rule that includes only observable features. A more sophisticated algorithm, IOE, generalizes over a number of such proof trees, producing rules that may incorporate nonterminals from the original network. If one knows that the target is a specialization of a known concept, these methods can focus attention on relevant features and speed the learning process.

A more challenging problem requires the *revision* of a complete but incorrect inference network. We examined RLT, an algorithm that carries out a hill-climbing search through the space of logical networks, using operators that add and delete rules and conditions. We also saw that backpropagation, a well-known method for connectionist learning, can be viewed as revising multilayer networks of threshold units, using gradient descent in its efforts to minimize error on the training data. In addition, we noted that one can apply similar ideas to the revision of Bayesian networks of probabilistic units.

Finally, we considered methods for *constructing* inference networks in the absence of significant background knowledge. We described one such algorithm for logical networks, GLT, which uses a simplicity metric to decide between creating new terms and merging existing ones. We also presented three related schemes for networks of threshold units, though in less detail, and we noted that similar methods apply to the generation of Bayesian networks with latent variables.

Exercises

1. Add the literal `thick-wall` to the right-hand side of the second `simploid` rule in Table 6-1, and similarly add the literal `thin-wall` to the second `neoplasm` rule. Draw the extensional definitions of `simploid` and `neoplasm` that result, using the cube notation from Figure 6-3. Also, show the extension of some target concept `lethargia'` that can be learned using these higher-level features using the HGS algorithm (Chapter 2) but not without them.
2. Assume a target concept `doldroma'` with the same definition as in Table 6-5, but in which the definitions for `simploid` and `neoplasm` have been altered as in Exercise 1. Show the proof tree generated by the ICE algorithm (Section 6.3.1) for some positive instance of `doldroma'`, along with the compiled rule that results. Show the compiled rules that result from all other instances of this concept.
3. Show the result of applying the IOE algorithm (Section 6.3.2) to the set of compiled rules for `doldroma'` from Exercise 2.
4. Remove the literal `one-tail` from the first `simploid` rule in Table 6-1 and add the literal `thin-wall` to the second `neoplasm` rule. Show the succession of inference networks generated by the RLT algorithm (Section 6.4.1) for the eight training cases in Table 6-7 (a), along with each network's prediction for these instances and their overall accuracies on the training set.
5. Transform the logical inference network from Exercise 4 into a multilayer threshold network that produces the same behavior, as in Figure 6-1. Assume a desired output of 1 for positive instances of `lethargia` and 0 for negative instances. Select a training case from Exercise 4 that this network misclassifies and use the backpropagation algorithm (Section 6.4.2), with 0.1 as the gain factor, to revise the network's weights to reduce its error on the instance. Show your calculations for each weight.

6. In Table 6-9, replace all occurrences of the literal `three-nuclei` in the training cases with `two-nuclei`, and replace all occurrences of `medium-color` with `dark-color`. Summarize the major steps taken by the GLT algorithm (Section 6.5.1) on these data, reporting the rules at each stage and each theory's overall simplicity score.

Historical and bibliographical remarks

Research on learning in inference networks has occurred within a number of distinct paradigms that seldom interact with each other. As we have seen, the simplest involves extending an incomplete inference network by inducing new conceptual structures that are described in terms of existing ones. Such constructive induction (coined by Michalski, 1983) has been explored with inference networks composed of logical, threshold, and competitive units. Sammut and Banerji's (1986) MARVIN, Elio and Watanabe's (1991) LAIR, and Drastal, Raatz, and Czako's (1989) MIRO extend logical networks, Gluck and Bower's (1988) configural cue model learns higher-order threshold concepts, and Kononenko (1991) and Pazzani (1995) use a naive Bayesian classifier in a similar manner. Some work on constructive induction (e.g., Matheus & Rendell, 1989; Pagallo, 1989) has also occurred with organizations other than inference networks.

The late 1980s saw considerable work on 'explanation-based' learning, following influential papers by Mitchell, Keller, and Kedar-Cabelli (1986), DeJong and Mooney (1986), and Laird, Rosenbloom, and Newell (1986). Early work in this area distinguished itself from other approaches to learning, especially earlier methods for logical induction, with some initial confusion about the performance task. Two paths subsequently emerged, one emphasizing the speeding up of problem solvers, which we discuss in Chapters 10 and 11, and the other focusing on supervised concept learning. Mitchell et al.'s (1986) EBG method (see the ICE algorithm in Section 6.3.1) formed the basis for much of the work in this latter area (e.g., Pazzani, 1988). Considerable attention also focused on clearly defining the notion of an operational term (e.g., Keller, 1988), while other work explored methods for combining evidence from multiple training cases (e.g., Cohen, 1988). Our formulation of explanation-based learning as theory-guided specialization comes directly from Flann and Dietterich (1989), and their IOE system is the source for our algorithm of the same name (Section 6.3.2).

Although the attention devoted to explanation-based learning has decreased in recent years, the idea of using background knowledge to aid learning has remained. Much of the work in theory revision, such as that by Hall (1989), Ourston and Mooney (1990), and Towell, Shavlik, and Noordeweier (1990), grew directly out of this tradition. Another source of interest in this topic was expert systems, which often require refinement when constructed manually. Ginsberg, Weiss, and Politakis (1988), Craw and Sleeman (1990), and Langley, Drastal, Rao, and Greiner (1994) present theory revision work with this motivation. The two approaches have produced quite similar methods, most using variations of the RLT algorithm (Section 6.4.1). Shapiro (1981) presents early work in this area, but took a more interactive approach than later methods.

By far the most active area covered in this chapter concerns weight revision in multilayer neural networks. This approach became popular in the mid-1980s, after an extended lapse of research on threshold units, as described in Chapter 3. An important cause for the new activity was the introduction of backpropagation (Section 6.4.2), by Rumelhart, Hinton, and Williams (1986), and related methods (e.g., Fahlman, 1988), which extended techniques like LMS, for gradient-descent search through a weight space, to networks with many levels. The literature reports many extensions to this basic approach, which can be found in proceedings of annual conferences like NIPS and IJCNN, and in journals such as *Neural Computation* and *International Journal of Neural Systems*. Lippman (1987), Hinton (1989), and Widrow, Rumelhart, and Lehr (1994) present overviews of work in this active area. Most research in this framework initializes weights randomly, but Towell et al. (1990) use backpropagation to revise an incorrect domain theory, and Thrun and Mitchell (1993) take a similar approach.

In general, research on determining the structure of inference networks from scratch has been much less common than work on modifying existing structure or altering weights. The GLT algorithm from Section 6.5.1 actually borrows from work on grammar induction (e.g., Langley, 1994b; Stolcke & Omohundro, 1994), which we discuss in Chapter 9. Muggleton's (1987) DUCE and its successor CIGOL (Muggleton & Buntine, 1988) introduced methods for inverse resolution, an approach to forming new terms in logical networks. There has been some research on determining the structure of threshold networks, but this constitutes only a small fraction of work in the area. The TSC algorithm (Sec-