Submitted to the *Future Generation Computer Systems* special issue on Data Mining.

# Using Neural Networks for Data Mining

Mark W. Craven
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
mark.craven@cs.cmu.edu

Jude W. Shavlik
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706-1685
shavlik@cs.wisc.edu

**Abstract**

Neural networks have been successfully applied in a wide range of supervised and unsupervised learning applications. Neural-network methods are not commonly used for data-mining tasks, however, because they often produce incomprehensible models and require long training times. In this article, we describe neural-network learning algorithms that are able to produce comprehensible models, and that do not require excessive training times. Specifically, we discuss two classes of approaches for data mining with neural networks. The first type of approach, often called *rule extraction*, involves extracting symbolic models from trained neural networks. The second approach is to directly learn simple, easy-to-understand networks. We argue that, given the current state of the art, neural-network methods deserve a place in the tool boxes of data-mining specialists.

**Keywords:** machine learning, neural networks, rule extraction, comprehensible models, decision trees, perceptrons

## 1 Introduction

The central focus of the data-mining enterprise is to gain insight into large collections of data. Often, achieving this goal involves applying machine-learning methods to inductively construct models of the data at hand. In this article, we provide an introduction to the topic of using neural-network methods for data mining. Neural networks have been applied to a wide variety of problem domains to learn models that are able to perform such interesting tasks as steering a motor vehicle, recognizing genes in uncharacterized DNA sequences, scheduling payloads for the space shuttle, and predicting exchange rates. Although neural-network learning algorithms have been successfully applied to a wide range of supervised and unsupervised learning problems, they have not often been applied in data-mining settings, in which two fundamental considerations are the comprehensibility of learned models and the time required to induce models from large data sets. We discuss new developments in neural-network learning that effectively address the comprehensibility and speed issues which often are of prime importance in the data-mining community. Specifically, we describe algorithms that are able to extract symbolic rules from trained neural networks, and algorithms that are able to directly learn comprehensible models.

Inductive learning is a central task in data mining since building descriptive models of a collection of data provides one way of gaining insight into it. Such models can be learned by either supervised or unsupervised methods, depending on the nature of the task. In supervised learning, the learner is given a set of instances of the form $\langle \vec{x}, y \rangle$, where $y$ represents the variable that we want the system to predict, and $\vec{x}$ is a vector of values that represent features thought to be relevant to determining $y$. The goal in supervised learning is to induce a general mapping from $\vec{x}$ vectors to

$y$ values. That is, the learner must build a model, $\hat{y} = f(\vec{x})$, of the unknown function $f$, that allows it to predict $y$ values for previously unseen examples. In unsupervised learning, the learner is also given a set of training examples but each instance consists only of the $\vec{x}$ part; it does not include the $y$ value. The goal in unsupervised learning is to build a model that accounts for regularities in the training set.

In both the supervised and unsupervised case, learning algorithms differ considerably in how they represent their induced models. Many learning methods represent their models using languages that are based on, or closely related to, logical formulae. Neural-network learning methods, on the other hand, represent their learned solutions using real-valued parameters in a network of simple processing units. We do not provide an introduction to neural-network models in this article, but instead refer the interested reader to one of the good textbooks in the field (e.g., Bishop, 1996). A detailed survey of real-world neural-network applications can be found elsewhere (Widrow et al., 1994).

The rest of this article is organized as follows. In the next section, we consider the applicability of neural-network methods to the task of data mining. Specifically, we discuss why one might want to consider using neural networks for such tasks, and we discuss why trained neural networks are usually hard to understand. The two succeeding sections cover two different types of approaches for learning comprehensible models using neural networks. Section 3 discusses methods for *extracting* comprehensible models from trained neural networks, and Section 4 describes neural-network learning methods that directly learn simple, and hopefully comprehensible, models. Finally, Section 5 provides conclusions.

## 2 The Suitability of Neural Networks for Data Mining

Before describing particular methods for data mining with neural networks, we first make an argument for why one might want to consider using neural networks for the task. The essence of the argument is that, for some problems, neural networks provide a more suitable *inductive bias* than competing algorithms. Let us briefly discuss the meaning of the term inductive bias. Given a fixed set of training examples, there are infinitely many models that could account for the data, and every learning algorithm has an inductive bias that determines the models that it is likely to return. There are two aspects to the inductive bias of an algorithm: its *restricted hypothesis space bias* and its *preference bias*. The restricted hypothesis space bias refers to the constraints that a learning algorithm places on the hypotheses that it is able to construct. For example, the hypothesis space of a perceptron is limited to linear discriminant functions. The preference bias of a learning algorithm refers to the preference ordering it places on the models that are within its hypothesis space. For example, most learning algorithms initially try to fit a simple hypothesis to a given training set and then explore progressively more complex hypotheses until they find an acceptable fit.

In some cases, neural networks have a more appropriate *restricted hypothesis space* bias than other learning algorithms. For example, sequential and temporal prediction tasks represent a class of problems for which neural networks often provide the most appropriate hypothesis space. *Recurrent networks*, which are often applied to these problems, are able to maintain state information from one time step to the next. This means that recurrent networks can use their hidden units to learn derived features relevant to the task at hand, and they can use the state of these derived features at one instant to help make a prediction for the next instance.

In other cases, neural networks are the preferred learning method not because of the class of hypotheses that they are able to represent, but simply because they induce hypotheses that generalize better than those of competing algorithms. Several empirical studies have pointed out that there are some problem domains in which neural networks provide superior predictive accuracy to commonly used symbolic learning algorithms (e.g., Shavlik et al., 1991).

Although neural networks have an appropriate inductive bias for a wide range of problems, they are not commonly used for data mining tasks. As stated previously, there are two primary explanations for this fact: trained neural networks are usually not comprehensible, and many neural-network learning methods are slow, making them impractical for very large data sets. We discuss these two issues in turn before moving on to the core part of the article.

The hypothesis represented by a trained neural network is defined by (a) the topology of the network, (b) the transfer functions used for the hidden and output units, and (c) the real-valued parameters associated with the network connections (i.e., the weights) and units (e.g., the biases of sigmoid units). Such hypotheses are difficult to comprehend for several reasons. First, typical networks have hundreds or thousands of real-valued parameters. These parameters encode the relationships between the input features, $\vec{x}$, and the target value, $y$. Although single-parameter encodings of this type are usually not hard to understand, the sheer number of parameters in a typical network can make the task of understanding them quite difficult. Second, in multi-layer networks, these parameters may represent nonlinear, nonmonotonic relationships between the input features and the target values. Thus it is usually not possible to determine, in isolation, the effect of a given feature on the target value, because this effect may be mediated by the values of other features.

These nonlinear, nonmonotonic relationships are represented by the hidden units in a network which combine the inputs of multiple features, thus allowing the model to take advantage of dependencies among the features. Hidden units can be thought of as representing higher-level, "derived features." Understanding hidden units is often difficult because they learn *distributed representations*. In a distributed representation, individual hidden units do not correspond to well understood features in the problem domain. Instead, features which are meaningful in the context of the problem domain are often encoded by *patterns* of activation across many hidden units. Similarly each hidden unit may play a part in representing numerous derived features.

Now let us consider the issue of the learning time required for neural networks. The process of learning, in most neural-network methods, involves using some type of gradient-based optimization method to adjust the network's parameters. Such optimization methods iteratively execute two basic steps: calculating the gradient of the error function (with respect to the network's adjustable parameters), and adjusting the network's parameters in the direction suggested by the gradient. Learning can be quite slow with such methods because the optimization procedure often involves a large number of small steps, and the cost of calculating the gradient at each step can be relatively expensive.

One appealing aspect of many neural-network learning methods, however, is that they are *on-line* algorithms, meaning that they update their hypotheses after every example is presented. Because they update their parameters frequently, on-line neural-network learning algorithms often converge much faster than *batch* algorithms. This is especially the case for large data sets. Often, a reasonably good solution can be found in only one pass through a large training set! For this reason, we argue that training-time performance of neural-network learning methods may often be acceptable for data-mining tasks, especially given the availability of high-performance, desktop computers.

## 3 Extraction Methods

One approach to understanding a hypothesis represented by a trained neural network is to translate the hypothesis into a more comprehensible language. Various approaches using this strategy have been investigated under the rubric of *rule extraction*. In this section, we give an overview of various rule-extraction approaches, and discuss a few of the successful applications of such methods.

The methods that we discuss in this section differ along several primary dimensions:

- **Representation language:** the language that is used by the extraction method to describe the neural network's learned model. The languages that have been used by various methods include conjunctive (*if-then*) inference rules, $m$-of-$n$ rules, fuzzy rules, decision trees, and finite state automata.
- **Extraction strategy:** the strategy used by the extraction method to map the model represented by the trained network into a model in the new representation language. Specifically, how does the method explore a space of candidate descriptions, and what level of description does it use to characterize the given neural network. That is, do the rules extracted by the method describe the behavior of the network as a whole, the behavior of individual units in the network, or something in-between these two cases. We use the term *global methods* to refer to the first case, and the term *local methods* to refer to the second case.
- **Network requirements:** the architectural and training requirements that the extraction method imposes on neural networks. In other words, the range of networks to which the method is applicable.

Throughout this section, as we describe various rule-extraction methods, we will evaluate them with respect to these three dimensions.

### 3.1 The Rule-Extraction Task

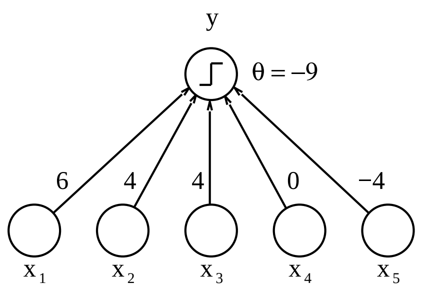
Figure 1 illustrates the task of rule extraction with a very simple network. This one-layer network has five Boolean inputs and one Boolean output. Any network, such as this one, which has discrete output classes and discrete-valued input features, can be exactly described by a finite set of symbolic *if-then* rules, since there is a finite number of possible input vectors. The extracted symbolic rules specify conditions on the input features that, when satisfied, guarantee a given output state. In our example, we assume that the value $false$ for a Boolean input feature is represented by an activation of 0, and the value $true$ is represented by an activation of 1. Also we assume that the output unit employs a threshold function to compute its activation:

$$a_y = \begin{cases} 1 & if \ \sum_i w_i a_i + \theta > 0 \\ 0 & otherwise \end{cases}$$

where $a_y$ is the activation of the output unit, $a_i$ is the activation of the $i$th input unit, $w_i$ is the weight from the $i$th input to the output unit, and $\theta$ is the threshold parameter of the output unit. We use $x_i$ to refer to the value of the $i$th feature, and $a_i$ to refer to the activation of the corresponding input unit. For example, if $x_i = true$ then $a_i = 1$.

Figure 1 shows three conjunctive rules which describe the most general conditions under which the output unit has an activation of unity. Consider the rule:

$$y \leftarrow x_1 \ \wedge \ x_2 \ \wedge \ \neg x_5.$$



**extracted rules:** $y \leftarrow x_1 \wedge x_2 \wedge x_3$
$y \leftarrow x_1 \wedge x_2 \wedge \neg x_5$
$y \leftarrow x_1 \wedge x_3 \wedge \neg x_5$

Figure 1: **A network and extracted rules.** The network has five input units representing five Boolean features. The rules describe the settings of the input features that result in the output unit having an activation of 1.

This rule states that when $x_1 = true$, $x_2 = true$, and $x_5 = false$, then the output unit representing $y$ will have an activation of 1 (i.e., the network predicts $y = true$). To see that this is a valid rule, consider that for the cases covered by this rule:

$$a_1 w_1 + a_2 w_2 + a_5 w_5 + \theta = 1.$$

Thus, the weighted sum exceeds zero. But what effect can the other features have on the output unit's activation in this case? It can be seen that:

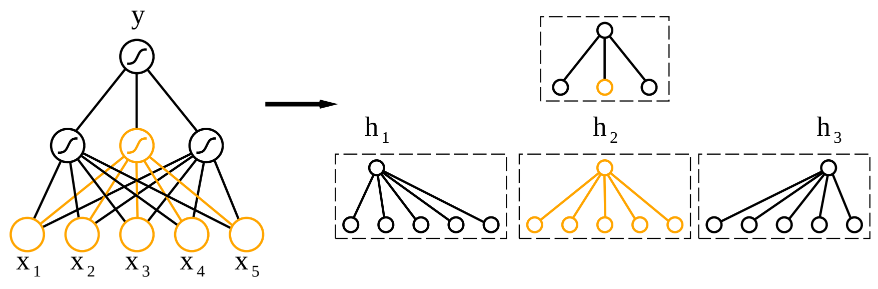
$$0 \leq a_3 w_3 + a_4 w_4 \leq 4.$$

No matter what values the features $x_3$ and $x_4$ have, the output unit will have an activation of 1. Thus the rule is *valid*; it accurately describes the behavior of the network for those instances that match its antecedent. To see that the rule is *maximally general*, consider that if we drop any one of the literals from the rule's antecedent, then the rule no longer accurately describes the behavior of the network. For example, if we drop the literal $\neg x_5$ from the rule, then for the examples covered by the rule:

$$-3 \leq \sum a_i w_i + \theta \leq 5$$

and thus the network does not predict that $y = true$ for all of the covered examples.

So far, we have defined an extracted rule in the context of a very simple neural network. What does a "rule" mean in the context of networks that have continuous transfer functions, hidden units, and multiple output units? Whenever a neural network is used for a classification problem, there is always an implicit decision procedure that is used to decide which class is predicted by the network for a given case. In the simple example above, the decision procedure was simply to predict $y = true$ when the activation of the output unit was 1, and to predict $y = false$ when it was 0. If we used a logistic transfer function instead of a threshold function at the output unit, then the decision procedure might be to predict $y = true$ when the activation exceeds a specified value, say 0.5. If we were using one output unit per class for a multi-class learning problem (i.e., a problem with more than two classes), then our decision procedure might be to predict the class associated with the output unit that has the greatest activation. In general, an extracted rule (approximately) describes a set of conditions under which the network, coupled with its decision procedure, predicts a given class.



**extracted rules:**
$$\begin{aligned} y &\leftarrow h_1 \vee h_2 \vee h_3 \\ h_1 &\leftarrow x_1 \wedge x_2 \\ h_2 &\leftarrow x_2 \wedge x_3 \wedge x_4 \\ h_3 &\leftarrow x_5 \end{aligned}$$

Figure 2: **The local approach to rule extraction.** A multi-layer neural network is decomposed into a set of single layer networks. Rules are extracted to describe each of the constituent networks, and the rule sets are combined to describe the multi-layer network.

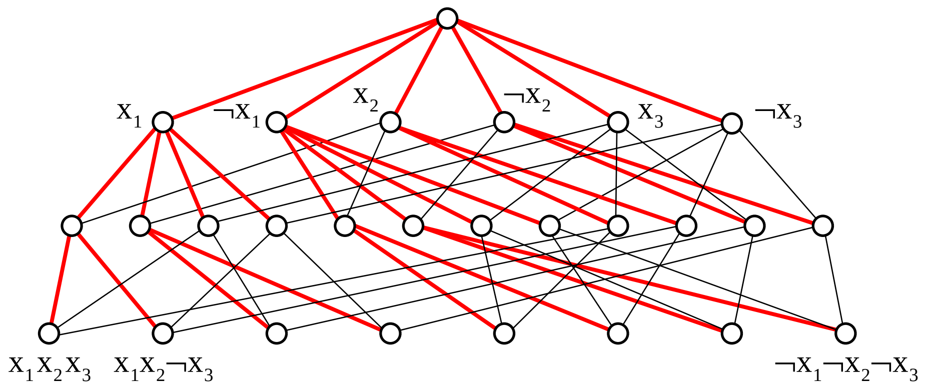


Figure 3: **A rule search space.** Each node in the space represents a possible rule antecedent. Edges between nodes indicate specialization relationships (in the downward direction). The thicker lines depict one possible *search tree* for this space.

As discussed at the beginning of this section, one of the dimensions along which rule-extraction methods can be characterized is their level of description. One approach is to extract a set of *global* rules that characterize the output classes directly in terms of the inputs. An alternative approach is to extract *local* rules by decomposing the multi-layer network into a collection of single-layer networks. A set of rules is extracted to describe each individual hidden and output unit in terms of the units that have weighted connections to it. The rules for the individual units are then combined into a set of rules that describes the network as a whole. The local approach to rule extraction is illustrated in Figure 2.

## 3.2 Search-Based Rule-Extraction Methods

Many rule-extraction algorithms have set up the task as a search problem which involves exploring a space of candidate rules and testing individual candidates against the network to see if they are valid rules. In this section we consider both global and local methods which approach the rule-extraction task in this way.

Most of these algorithms conduct their search through a space of conjunctive rules. Figure 3 shows a rule search space for a problem with three Boolean features. Each node in the tree corresponds to the antecedent of a possible rule, and the edges indicate specialization relationships (in the downward direction) between nodes. The node at the top of the graph represents the most general rule (i.e., all instances are members of the class $y$), and the nodes at the bottom of the tree represent the most specific rules, which cover only one example each. Unlike most search processes which continue until the first goal node is found, a rule-extraction search continues until all (or most) of the maximally-general rules have been found.

Notice that rules with more than one literal in their antecedent have multiple ancestors in the graph. Obviously when exploring a rule space, it is inefficient for the search procedure to visit a node multiple times. In order to avoid this inefficiency, we can impose an ordering on the literals thereby transforming the search graph into a tree. The thicker lines in Figure 3 depict one possible search tree for the given rule space.

One of the problematic issues that arises in search-based approaches to rule extraction is that the size of the rule space can be very large. For a problem with $n$ binary features, there are $3^n$ possible conjunctive rules (since each feature can be absent from a rule antecedent, or it can occur as a positive or a negative literal in the antecedent). To address this issue, a number of heuristics have been employed to limit the combinatorics of the rule-exploration process.

Several rule-extraction algorithms manage the combinatorics of the task by limiting the number of literals that can be in the antecedents of extracted rules (Saito & Nakano, 1988; Gallant, 1993). For example, Saito and Nakano's algorithm uses two parameters, $k_{pos}$ and $k_{neg}$, that specify the maximum number of positive and negative literals respectively that can be in an antecedent. By restricting the search to a depth of $k$, the rule space considered is limited to a size given by the following expression:

$$\sum_{i=0}^{k} \left( \begin{array}{c} n \\ k \end{array} \right) 2^k .$$

For fixed $k$, this expression is polynomial in $n$, but obviously, it is exponential in the depth $k$. This means that exploring a space of rules might still be intractable since, for some networks, it may be necessary to search deep in the tree in order to find valid rules.

The second heuristic employed by Saito and Nakano is to limit the search to combinations of literals that occur in the training set used for the network. Thus, if the training set did not contain an example for which $x_1 = true$ and $x_2 = true$, then the rule search would not consider the rule $y \leftarrow x_1 \wedge x_2$ or any of its specializations.

Exploring a space of candidate rules is only one part of the task for a search-based rule-extraction method. The other part of the task is testing candidate rules against the network. The method developed by Gallant operates by propagating *activation intervals* through the network. The first step in testing a rule using this method is to set the activations of the input units that correspond to the literals in the candidate rule. The next step is to propagate activations through the network. The key idea of this second step, however, is the assumption that input units whose activations are not specified by the rule could possibly take on any allowable value, and thus intervals of activations are propagated to the units in the next layer. Effectively, the network computes, for the examples covered by the rule, the range of possible activations in the next layer. Activation intervals are then further propagated from the hidden units to the output units. At this point, the range of possible activations for the output units can be determined and the procedure can decide whether to accept the rule or not. Although this algorithm is guaranteed to accept only rules that are valid, it may fail to accept maximally general rules, and instead may return overly specific rules. The reason for this deficiency is that in propagating activation intervals from the hidden units onward, the procedure assumes that the activations of the hidden units are independent of one another. In most networks this assumption is unlikely to hold.

Thrun (1995) developed a method called *validity interval analysis* (VIA) that is a generalized and more powerful version of this technique. Like Gallant's method, VIA tests rules by propagating activation intervals through a network after constraining some of the input and output units. The key difference is that Thrun frames the problem of determining *validity intervals* (i.e., valid activation ranges for each unit) as a linear programming problem. This is an important insight because it allows activation intervals to be propagated *backward*, as well as forward through the network, and it allows arbitrary linear constraints to be incorporated into the computation of validity intervals. Backward propagation of activation intervals enables the calculation of tighter validity intervals than forward propagation alone. Thus, Thrun's method will detect valid rules that Gallant's algorithm is not able to confirm. The ability to incorporate arbitrary linear constraints into the extraction process means that the method can be used to test rules that specify very general conditions on the output units. For example, it can extract rules that describe when one output unit has a greater activation than all of the other output units. Although the VIA approach is better at detecting general rules than Gallant's algorithm, it may still fail to confirm maximally general rules, because it also assumes that the hidden units in a layer act independently.

The rule-extraction methods we have discussed so far extract rules that describe the behavior of the output units in terms of the input units. Another approach to the rule-extraction problem is to decompose the network into a collection of networks, and then to extract a set of rules describing each of the constituent networks.

There are a number of local rule-extraction methods for networks that use sigmoidal transfer functions for their hidden and output units. In these methods, the assumption is made that the hidden and output units can be approximated by threshold functions, and thus each unit can be described by a binary variable indicating whether it is "on" (activation $\approx 1$) or "off" (activation $\approx 0$). Given this assumption, we can extract a set of rules to describe each individual hidden and output unit in terms of the units that have weighted connections to it. The rules for each unit can then be combined into a single rule set that describes the network as a whole.

If the activations of the input and hidden units in a network are limited to the interval [0, 1], then the local approach can significantly simplify the rule search space. The key fact that simplifies the search combinatorics in this case is that the relationship between any input to a unit and its output is a monotonic one. That is, we can look at the sign of the weight connecting the $i$th input to the unit of interest to determine how this variable influences the activation of the unit. If the sign is positive, then we know that this input can only push the unit's activation *towards* 1, it cannot push it *away* from 1. Likewise, if the sign of the weight is negative, then the input can only push the unit's activation away from 1. Thus, if we are extracting rules to explain when the unit has an activation of 1, we need to consider $\neg x_i$ literals only for those inputs $x_i$ that have negative weights, and we need consider non-negated $x_i$ literals only for those inputs that have positive weights. When a search space is limited to including either $x_i$ or $\neg x_i$, but not both, the number of rules in the space is $2^n$ for a task with $n$ binary features. Recall that when this monotonicity condition does not hold, the size of the rule space is $3^n$.

Figure 4 shows a rule search space for the network in Figure 1. The shaded nodes in the graph correspond to the extracted rules shown in Figure 1. Note that this tree exploits the monotonicity condition, and thus does not show all possible conjunctive rules for the network.

A number of research groups have developed local rule-extraction methods that search for conjunctive rules (Fu, 1991; Gallant, 1993; Sethi & Yoo, 1994). Like the global methods described previously, the local methods developed by Fu and Gallant manage search combinatorics by limiting the depth of the rule search. When the monotonicity condition holds, the number of rules considered

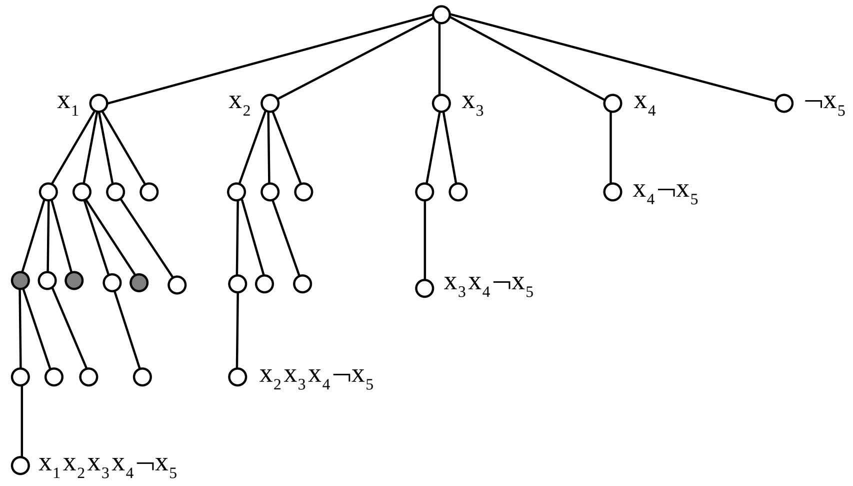


Figure 4: **A search tree for the network in Figure 1.** Each node in the space represents a possible rule antecedent. Edges between nodes indicate specialization relationships (in the downward direction). Shaded nodes correspond to the extracted rules shown in Figure 1.

in a search of depth $k$ is bounded above by:

$$\sum_{i=0}^{k} \binom{n}{k}.$$

There is another factor that simplifies the rule search when the monotonicity condition is true. Because the relationship between each input and the output unit in a perceptron is specified by a single parameter (i.e., the weight on the connection between the two), we know not only the sign of the input's contribution to the output, but also the possible magnitude of the contribution. This information can be used to order the search tree in a manner that can save effort. For example, when searching the rule space for the network in Figure 1, after determining that $y \leftarrow x_1$ is not a valid rule, we do not have to consider other rules that have only one literal in their antecedent. Since the weight connecting $x_1$ to the output unit is larger than the weight connecting any other input unit, we can conclude that if $x_1$ alone cannot guarantee that the output unit will have an activation of 1, then no other single input unit can do it either. Sethi and Yoo (1994) have shown that, when this heuristic is employed, the number of nodes explored in the search is:

$$O\left(\sqrt{\frac{2n}{\pi}}\,\frac{2^n}{n}\right).$$

Notice that even with this heuristic, the number of nodes that might need to be visited in the search is still exponential in the number of variables.

It can be seen that one advantage of local search-based methods, in comparison to global methods, is that the worst-case complexity of the search is less daunting. Another advantage of local methods, is that the process of testing candidate rules is simpler.

A local method developed by Towell and Shavlik (1993) searches not for conjunctive rules, but instead for rules that include $m$-of-$n$ expressions. An $m$-of-$n$ expression is a Boolean expression that is specified by an integer threshold, $m$, and a set of $n$ Boolean literals. Such an expression is satisfied when at least $m$ of its $n$ literals are satisfied. For example, suppose we have three Boolean features, $x_1$, $x_2$, and $x_3$; the $m$-of-$n$ expression *2-of-*$\{x_1, \neg x_2, x_3\}$ is logically equivalent to $(x_1 \wedge \neg x_2) \vee (x_1 \wedge x_3) \vee (\neg x_2 \wedge x_3)$.

There are two advantages to extracting $m$-of-$n$ rules instead of conjunctive rules. The first advantage is that $m$-of-$n$ rule sets are often much more concise and comprehensible than their conjunctive counterparts. The second advantage is that, when using a local approach, the combinatorics of the rule search can be simplified. The approach developed by Towell and Shavlik extracts $m$-of-$n$ rules for a unit by first clustering weights and then treating weight clusters as equivalence classes. This clustering reduces the search problem from one defined by $n$ weights to one defined by $(c \ll n)$ clusters. This approach, which assumes that the weights are fairly well clustered after training, was initially developed for *knowledge-based neural networks* (Towell & Shavlik, 1993), in which the initial weights of the network are specified by a set of symbolic inference rules. Since they correspond the symbolic rules, the weights in these networks are initially well clustered, and empirical results indicate that the weights remain fairly clustered after training. The applicability of this approach was later extended to ordinary neural networks by using a special cost function for network training (Craven & Shavlik, 1993).

## 3.3 A Learning-Based Rule-Extraction Method

In contrast to the previously discussed methods, we have developed a rule-extraction algorithm called TREPAN (Craven & Shavlik, 1996; Craven, 1996), that views the problem of extracting a comprehensible hypothesis from a trained network as an inductive learning task. In this learning task, the target concept is the function represented by the network, and the hypothesis produced by the learning algorithm is a decision tree that approximates the network.

TREPAN differs from other rule-extraction methods in that it does not directly test hypothesized rules against a network, nor does it translate individual hidden and output units into rules. Instead, TREPAN's extraction process involves progressively refining a model of the entire network. The model, in this case, is a decision tree which is grown in a best-first manner.

The TREPAN algorithm, as shown in Table 1, is similar to conventional decision-tree algorithms, such as CART (Breiman et al., 1984) and C4.5 (Quinlan, 1993), which learn directly from a training set. These algorithms build decision trees by recursively partitioning the input space. Each internal node in such a tree represents a splitting criterion that partitions some part of the input space, and each leaf represents a predicted class.

As TREPAN grows a tree, it maintains a queue of leaves which are expanded into subtrees as they are removed from the queue. With each node in the queue, TREPAN stores (i) a subset of the training examples, (ii) another set of instances which we shall refer to as *query instances*, and (iii) a set of constraints. The stored subset of training examples consists simply of those examples that reach the node. The query instances are used, along with the training examples, to select the splitting test if the node is an internal node, or to determine the class label if it is a leaf. The constraint set describes the conditions that instances must satisfy in order to reach the node; this information is used when drawing a set of query instances for a newly created node.

Although TREPAN has many similarities to conventional decision-tree algorithms, it is substantially different in a number of respects, which we detail below.

**Membership Queries and the Oracle.** When inducing a decision tree to describe the given network, TREPAN takes advantage of the fact that it can make *membership queries*. A membership query is a question to an *oracle* that consists of an instance from the learner's instance space. Given a membership query, the role of the oracle is to return the class label for the instance. Recall that, in this context, the target concept we are trying to learn is the function represented by the network. Thus, the network itself serves as the oracle, and to answer a membership query it simply classifies the given instance.

The instances that TREPAN uses for membership queries come from two sources. First, the

Table 1: **The Trepan algorithm.**

```
TREPAN
Input: ORACLE(), training set $S$, feature set $F$, $min\_sample$

    initialize the root of the tree, $R$, as a leaf node

    /* get a sample of instances */
    use $S$ to construct a model $M_R$ of the distribution of instances covered by node $R$
    $q := \max(0, min\_sample - |S|)$
    $query\_instances_R$ := a set of $q$ instances generated using model $M_R$

    /* use the network to label all instances */
    for each example $x \in (S \cup query\_instances_R)$
        class label for $x$ := ORACLE($x$)

    /* do a best-first expansion of the tree */
    initialize $Queue$ with tuple $\langle R, S, query\_instances_R, \{\} \rangle$
    while $Queue$ is not empty and global stopping criteria not satisfied

        /* make node at head of $Queue$ into an internal node */
        remove $\langle$ node $N, S_N, query\_instances_N, constraints_N \rangle$ from head of $Queue$
        use $F$, $S_N$, and $query\_instances_N$ to construct a splitting test $T$ at node $N$

        /* make children nodes */
        for each outcome, $t$, of test $T$
            make $C$, a new child node of $N$
            $constraints_C := constraints_N \cup \{T = t\}$

            /* get a sample of instances for the node $C$ */
            $S_C$ := members of $S_N$ with outcome $t$ on test $T$
            construct a model $M_C$ of the distribution of instances covered by node $C$
            $q := \max(0, min\_sample - |S_C|)$
            $query\_instances_C$ := a set of $q$ instances generated using model $M_C$ and $constraints_C$
            for each example $x \in query\_instances_C$
                class label for $x$ := ORACLE($x$)

            /* make node $C$ a leaf for now */
            use $S_C$ and $query\_instances_C$ to determine class label for $C$

            /* determine if node $C$ should be expanded */
            if local stopping criteria not satisfied then
                put $\langle C, S_C, query\_instances_C, constraints_C \rangle$ into $Queue$

Return: tree with root $R$
```

examples that were used to train the network are used as membership queries. Second, TREPAN also uses the training data to construct models of the underlying data distribution, and then uses these models to generate new instances – the query instances – for membership queries. The ability to make membership queries means that whenever TREPAN selects a splitting test for an internal node or selects a class label for a leaf, it is able to base these decisions on large samples of data.

**Tree Expansion.** Unlike most decision-tree algorithms, which grow trees in a depth-first manner, TREPAN grows trees using a best-first expansion. The notion of the best node, in this case, is the one at which there is the greatest potential to increase the *fidelity* of the extracted tree to the network. By fidelity, we mean the extent to which the tree agrees with the network in its classifications. The function used to evaluate node $N$ is:

$$f(N) = reach(N) \times (1 - fidelity(N))$$

where $reach(N)$ is the estimated fraction of instances that reach $N$ when passed through the tree, and $fidelity(N)$ is the estimated fidelity of the tree to the network for those instances. The motivation for expanding an extracted tree in a best-first manner is that it gives the user a fine degree of control over the size of the tree to be returned: the tree-expansion process can be stopped at any point.

**Splitting Tests.** Like some of the rule-extraction methods discussed earlier, TREPAN exploits $m$-of-$n$ expressions to produced more compact extracted descriptions. Specifically, TREPAN uses a heuristic search process to construct $m$-of-$n$ expressions for the splitting tests at its internal nodes in a tree.

**Stopping Criteria.** TREPAN uses three criteria to decide when to stop growing an extracted tree. First, TREPAN uses a statistical test to decide if, with high probability, a node covers only instances of a single class. If it does, then TREPAN does not expand this node further. Second, TREPAN employs a parameter that allows the user to place a limit on the size of the tree that TREPAN can return. This parameter, which is specified in terms of internal nodes, gives the user some control over the comprehensibility of the tree produced by enabling a user to specify the largest tree that would be acceptable. Third, TREPAN can use a validation set, in conjunction with the size-limit parameter, to decide on the tree to be returned. Since TREPAN grows trees in a best-first manner, it can be thought of as producing a nested sequence of trees in which each tree in the sequence differs from its predecessor only by the subtree that corresponds to the node expanded at the last step. When given a validation set, TREPAN uses it to measure the fidelity of each tree in this sequence, and then returns the tree that has the highest level of fidelity to the network.

The principal advantages of the TREPAN approach, in comparison to other rule-extraction methods, are twofold. First, TREPAN can be applied to a wide class of networks. The generality of TREPAN derives from the fact that its interaction with the network consists solely of membership queries. Since answering a membership query involves simply classifying an instance, TREPAN does not require a special network architecture or training method. In fact, TREPAN does not even require that the model be a neural network. TREPAN can be applied to a wide variety of hard-to-understand models including *ensembles* (or committees) of classifiers that act in concert to produce predictions.

The other principal advantage of TREPAN is that it gives the user fine control over the complexity of the hypotheses returned by the rule-extraction process. This capability derives from the fact that TREPAN represents its extracted hypotheses using decision trees, and it expands these trees in a best-first manner. TREPAN first extracts a very simple (i.e., one-node) description of a trained network, and then successively refines this description to improve its fidelity to the network. In

Table 2: **Test-set accuracy (%) and tree complexity (# feature references).**

| | accuracy | | | tree complexity | |
|---|---|---|---|---|---|
| problem domain | networks | TREPAN | C4.5 | TREPAN | C4.5 |
| protein-coding region recognition | 94.1 | 93.1 | 90.4 | 70.5 | 153.3 |
| heart-disease diagnosis | 84.5 | 83.2 | 74.6 | 24.4 | 15.5 |
| promoter recognition | 90.6 | 87.4 | 85.0 | 105.5 | 7.0 |
| telephone-circuit fault diagnosis | 65.3 | 63.3 | 60.7 | 26.3 | 35.0 |
| exchange-rate prediction | 61.6 | 60.6 | 54.6 | 14.0 | 53.0 |

this way, TREPAN explores increasingly more complex, but higher fidelity, descriptions of the given network.

TREPAN has been used to extract rules from networks trained in a wide variety of problem domains including: gene and *promoter* identification in DNA, telephone-circuit fault diagnosis, exchange-rate prediction, and elevator control. Table 2 shows test-set accuracy and tree complexity results for five such problem domains. The table shows the predictive accuracy of feed-forward neural networks, decision trees extracted from the networks using TREPAN, and decision trees learned directly from the data using the C4.5 algorithm (Quinlan, 1993). It can be seen that, for every data set, neural networks provide better predictive accuracy than the decision trees learned by C4.5. This result indicates that these are domains for which neural networks have a more suitable inductive bias than C4.5. Indeed, these problem domains were selected for this reason, since it is in cases where neural networks provide superior predictive accuracy to symbolic learning approaches that it makes sense to apply a rule-extraction method. Moreover, for all five domains, the trees extracted from the neural networks by TREPAN are more accurate than the C4.5 trees. This result indicates that in a wide range of problem domains in which neural networks provide better predictive accuracy than conventional decision-tree algorithms, TREPAN is able to extract decision trees that closely approximate the hypotheses learned by the networks, and thus provide superior predictive accuracy to trees learned directly by algorithms such as C4.5.

The two rightmost columns in Table 2 show tree complexity measurements for the trees produced by TREPAN and C4.5 in these domains. The measure of complexity used here is the number of *feature references* used in the splitting tests in the trees. An ordinary, single-feature splitting test, like those used by C4.5, is counted as one feature reference. An $m$-of-$n$ test, like those used at times by TREPAN, is counted as $n$ feature references, since such a split lists $n$ feature values. We contend that this measure of syntactic complexity is a good indicator of the comprehensibility of trees. The results in this table indicate that, in general, the trees produced by the two algorithms are roughly comparable in terms of size. The results presented in this table are described in greater detail elsewhere (Craven, 1996).

## 3.4 Finite State Automata Extraction Methods

One specialized case of rule extraction is the extraction of finite state automata (FSA) from *recurrent* neural networks. A recurrent network is one that has links from a set of its hidden or output units to a set of its input units. Such links enable recurrent networks to maintain state information from one input instance to the next. Like a finite state automaton, each time a recurrent network is presented with an instance, it calculates a new state which is a function of both the previous state and the given instance. A "state" in a recurrent network is not a predefined, discrete entity, but instead corresponds to a vector of activation values across the units in the network that have

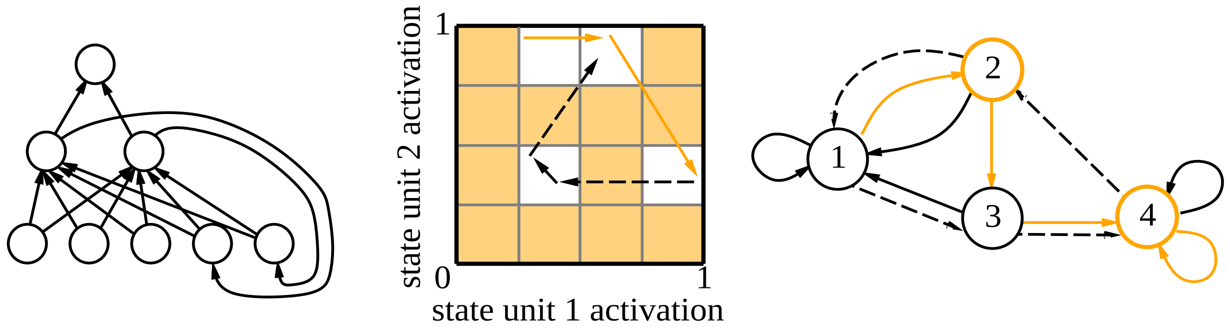


Figure 5: **The correspondence between a recurrent network and an FSA.** Depicted on the left is a recurrent network that has three input units and two state units. The two units that are to the right of the input units represent the activations of the state units at time $t-1$. Shown in the middle is the two-dimensional, real-valued space defined by the activations of the two hidden units. The path traced in the space illustrates the state changes of the hidden-unit activations as the network processes some sequence of inputs. Each of the three arrow styles represents one of the possible inputs to the recurrent network. Depicted on the right is a finite state automaton that corresponds to the network when the state space is discretized as shown in the middle figure. The shade of each node in the FSA represents the output value produced by the network when it is in the corresponding state.

outgoing recurrent connections – the so-called *state units*. Another way to think of such a state is as a point in an $s$-dimensional, real-valued space defined by the activations of the $s$ state units.

Recurrent networks are usually trained on *sequences* of input vectors. In such a sequence, the order in which the input vectors are presented to the network represents a temporal order, or some other natural sequential order. As a recurrent network processes such an input sequence, its state-unit activations trace a path in the $s$-dimensional state-unit space. If similar input sequences produce similar paths, then the continuous-state space can be closely approximated by a finite state space in which each state corresponds to a region, as opposed to a point, in the space. This idea is illustrated in Figure 5, which shows a finite state automaton and the two-dimensional state-unit space of a recurrent network trained to accept the same strings as the FSA. The path traced in the space illustrates the state changes of the state-unit activations as the network processes a sequence of inputs. The non-shaded regions of the space correspond to the states of the FSA.

Several research groups have developed algorithms for extracting finite state automata from trained recurrent networks. The key issue in such algorithms, is deciding how to partition the $s$-dimensional real-valued space into a set of discrete states. The method of Giles et al. (Giles et al., 1992), which is representative of this class of algorithms, proceeds as follows. First, the algorithm partitions each state unit's activation range into $q$ intervals of equal width, thus dividing the $s$-dimensional space into $q^S$ partitions. The method initially sets $q = 2$, but increases its value if it cannot extract an FSA that correctly describes the network's training set. The next step is to run the input sequences through the network, keeping track of (i) the state transitions, (ii) the input vector associated with each transition, and (iii) the output value produced by the network. It is then a simple task to express this record of the network's processing as a finite state automaton. Finally, the FSA can be minimized using standard algorithms.

Recently, this approach has been applied to the task of exchange-rate prediction. Lawrence et al. (1997) trained recurrent neural networks to predict the daily change in five foreign exchange rates, and then extracted finite state automata from these networks in order to characterize the learned models. More specifically, the task they addressed was to predict the next (log-transformed) change in a daily exchange rate $x(t + 1)$, given the previous four values of the same time series:

$$X(t) = (x(t), x(t-1), x(t-2), x(t-3)).$$

Their solution to this task involves three main components. The first component is a neural network called a self-organizing map (SOM) (Kohonen, 1995) which is trained by an unsupervised learning process. An SOM learns a mapping from its input space to its output space that preserves the topological ordering of points in the input space. That is, the similarity of points in the input space, as measured using a metric, is preserved in the output space. In their exchange-rate prediction architecture, Lawrence et al. used SOMs to map from a continuous, four-dimensional input space into a discrete output space. The input space, in this case, represents $X(t)$, and the output space represents a three-valued discrete variable that characterizes the trend in the exchange rate.

The second component of the system is a neural network that has a set of recurrent connections from each of its hidden units to all of the other hidden units. The input to the recurrent network is a three-dimensional vector consisting of the last three discrete values output by the self-organizing map. The output of the network is the predicted probabilities that the next daily movement of the exchange rate will be upward or downward. In other words, the recurrent network learns a mapping from the SOM's discrete characterization of the time series to the predicted direction of the next value in the time series.

The third major part of the system is the rule-extraction component. Using the method described above, finite state automata are extracted from the recurrent networks. The states in the FSA correspond to regions in the space of activations of the state units. Each state is labeled by the corresponding network prediction (*up* or *down*), and each state transition is labeled by the value of the discrete variable that characterizes the time series at time $t$.

After extracting automata from the recurrent networks, Lawrence et al. compared their predictive accuracy to that of the neural networks and found that the FSA were only slightly less accurate. On average, the accuracy of the recurrent networks was 53.4%, and the accuracy of the finite state automata was 53.1% (both of which are statistically distinguishable from random guessing).

## 3.5 Discussion

As we stated at the beginning of this section, there are three primary dimensions along which rule-extraction methods differ: representation language, extraction strategy, and network requirements. The algorithms that we have discussed in this section provide some indication of the diversity of rule-extraction methods with respect to these three aspects.

The representation languages used by the methods we have covered include conjunctive inference rules, $m$-of-$n$ inference rules, decision trees with $m$-of-$n$ tests, and finite state automata. In addition to these representations, there are rule-extraction methods that use fuzzy rules, rules with confidence factors, "majority-vote" rules, and condition/action rules that perform rewrite operations on string-based inputs. This multiplicity of languages is due to several factors. One factor is that different representation languages are well suited for different types of networks and tasks. A second reason is that researchers in the field have found that it is often hard to concisely describe the concept represented by a neural network to a high level of fidelity. Thus, some of the described representations, such as $m$-of-$n$ rules, have gained currency because they often help to simplify extracted representations.

The extraction strategies employed by various algorithms also exhibit similar diversity. As discussed earlier, one aspect of extraction strategy that distinguishes methods is whether they extract global or local rules. Recall that global methods produce rules which describe a network as a whole, whereas local methods extract rules which describe individual hidden and output units in the network. Another key aspect of extraction strategies is the way in which they explore a space of rules. In this section we described (i) methods that use search-like procedures to explore rule spaces, (ii) a method that iteratively refines a decision-tree description of a network, and (iii) a method that extracts finite state automata by first clustering unit activations and then mapping the clusters into an automaton. In addition to these rule-exploration strategies, there are also algorithms that extract rules by matching the network's weight vectors against templates representing canonical rules, and methods that are able to directly map hidden units into rules when the networks use transfer functions, such as radial basis functions, that respond to localized regions of their input space.

Another key dimension we have considered is the extent to which methods place requirements on the networks to which they can be applied. Some methods require that a special training procedure be used for the network. Other methods impose restrictions on the network architecture. or require that hidden units use sigmoidal transfer functions. Some of the methods we have discussed place restrictions on both the network's architecture and its training regime. Another limitation of many rule-extraction methods is that they are designed for problems that have only discrete-valued features. The tradeoff that is involved in these requirements is that, although they may simplify the rule-extraction process, they reduce the generality of the rule-extraction method.

Readers who are interested in more detailed descriptions of these rule-extraction methods, as well as pointers to the literature are referred elsewhere (Andrews et al., 1995; Craven, 1996).

## 4 Methods that Learn Simple Hypotheses

The previous section discussed methods that are designed to extract comprehensible hypotheses from trained neural networks. An alternative approach to data mining with neural networks is to use learning methods that directly learn comprehensible hypotheses by producing simple neural networks. Although we have assumed in our discussion so far that the hypotheses learned by neural networks are incomprehensible, the methods we present in this section are different in that they learn networks that have a single layer of weights. In contrast to multi-layer networks, the hypotheses represented by single-layer networks are usually much easier to understand because each parameter describes a simple (i.e., linear) relationship between an input feature and an output category.

### 4.1 A Supervised Method

There is a wide variety of methods for learning single-layer neural networks in a supervised learning setting. In this section we focus on one particular algorithm that is appealing for data-mining applications because it incrementally constructs its learned networks. This algorithm, called BBP (Jackson & Craven, 1996), is unlike traditional neural-network methods in that it does not involve training with a gradient-based optimization method. The hypotheses it learns, however, are perceptrons, and thus we consider it to be a neural-network method.

The BBP algorithm is shown in Table 3. The basic idea of the method is to repeatedly add new input units to a learned hypothesis, using different probability distributions over the training set to select each one. Because the algorithm adds weighted inputs to hypotheses incrementally, the complexity of these hypotheses can be easily controlled.

The inputs incorporated by BBP into a hypothesis represent Boolean functions that map to $\{-1, +1\}$. In other words, the inputs are binary units that have an activation of either $-1$ or $+1$. These inputs may correspond directly to Boolean features, or they may represent tests on nominal or numerical features (e.g., $color = red$, $x_1 > 0.8$), or logical combinations of features (e.g., $[color = red] \wedge [shape = round]$). Additionally, the algorithm may also incorporate an input representing the identically *true* function. The weight associated with such an input corresponds to the threshold of the perceptron.

Table 3: **The** BBP **algorithm.**

BBP
**Input:** training set $S$ of $m$ examples, set of candidate inputs $C$ that map to $\{-1, +1\}$,
number of iterations $T$

```
    /* set the initial distribution to be uniform */
    for all $x \in S$
        $D_1(x) := 1/m$

    for $t := 1$ to $T$ do
        /* add another feature */
        $h_t := \text{argmax}_{c_i \in C} |E_{D_t}[f(x) \cdot c_i(x)]|$

        /* determine error of this feature */
        $\epsilon_t := 0$
        for all $x \in S$
            if $h_t(x) \neq f(x)$ then $\epsilon_t := \epsilon_t + D_t(x)$

        /* update the distribution */
        $\beta_t := \epsilon_t \,/\, (1 - \epsilon_t)$
        for all $x \in S$
            if $h_t(x) = f(x)$ then
                $D_{t+1}(x) := \beta_t D_t(x)$
            else
                $D_{t+1}(x) := D_t(x)$

        /* re-normalize the distribution */
        $Z_t := \sum_x D_{t+1}(x)$
        for all $x \in S$
            $D_{t+1}(x) := D_{t+1}(x)/Z_t$
```

**Return:** $h(x) \equiv \text{sign}\left(\sum_{i=1}^{T} -\ln(\beta_i)\, h_i(x)\right)$

On each iteration of the BBP algorithm, an input is selected from the pool of candidates and added to the hypothesis under construction. BBP measures the correlation of each input with the target function being learned, and then selects the input whose correlation has the greatest magnitude. The correlation between a given candidate and the target function varies from iteration to iteration because it is measured with respect to a changing distribution over the training examples.

Initially, the BBP algorithm assumes a uniform distribution over the training set. That is, when selecting the first input to be added to a perceptron, BBP assigns equal importance to the various instances of the training set. After each input is added, however, the distribution is adjusted so that more weight is given to the examples that the input did not correctly predict. In this way, the learner's attention is focused on those examples that the current hypothesis does not explain well.

The algorithm stops adding weighted inputs to the hypothesis after a pre-specified number of iterations have been reached, or after the training set error has been reduced to zero. Since only one input is added to the network on each iteration, the size of the final perceptron can be controlled by limiting the number of iterations. The hypothesis returned by BBP is a perceptron in which the weight associated with each input is a function of the error of the input. The perceptron uses the *sign* function to decide which class to return:

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x \leq 0. \end{cases}$$

The BBP algorithm has two primary limitations. First, it is designed for learning binary classification tasks. The algorithm can be applied to multi-class learning problems, however, by learning a perceptron for each class. The other limitation of the method is that it assumes that the inputs are Boolean functions. As discussed above, however, domains with real-valued features can be handled by discretizing the features.

The BBP method is based on an algorithm called AdaBoost (Freund & Schapire, 1996) which is a *hypothesis-boosting* algorithm. Informally, a boosting algorithm learns a set of constituent hypotheses and then combines them into a composite hypothesis in such a way that, even if each of the constituent hypotheses is only slightly more accurate than random guessing, the composite hypothesis has an arbitrarily high level of accuracy on the training data. In short, a set of *weak* hypotheses are boosted into a *strong* one. This is done by carefully determining the distribution over the training data that is used for learning each weak hypothesis. The weak hypotheses in a BBP perceptron are simply the individual inputs. Although the more general AdaBoost algorithm can use arbitrarily complex functions as its weak hypotheses, BBP uses very simple functions for its weak hypotheses in order to facilitate comprehensibility.

Table 4 shows test-set accuracy and hypothesis-complexity results for the BBP algorithm, ordinary feed-forward neural networks, and C4.5 applied to three problem domains in molecular biology. As the table indicates, simple neural networks, such as those induced by BBP, can provide accuracy comparable to multi-layer neural networks in some domains. Moreover, in two of the three domains, the accuracy of the BBP hypotheses is significantly superior to decision trees learned by C4.5.

The three rightmost columns of Table 4 show one measure of the complexity of the models learned by the three methods – the total number of features incorporated into their hypotheses. These results illustrate that, like decision-tree algorithms, the BBP algorithm is able to selectively incorporate input features into its hypotheses. Thus in many cases, the BBP hypotheses use significantly fewer features, and have significantly fewer weights, than ordinary multi-layer networks. It should also be emphasized that multi-layer networks are usually much more difficult to interpret than BBP hypotheses because their weights may encode nonlinear, nonmonotonic relationships

Table 4: **Test-set accuracy (%) and hypothesis complexity (# features used ).**

| problem domain | accuracy | | | hypothesis complexity | | |
|---|---|---|---|---|---|---|
| | networks | BBP | C4.5 | networks | BBP | C4.5 |
| protein-coding region recognition | 93.6 | 93.6 | 84.9 | 464 | 171 | 150 |
| promoter recognition | 90.6 | 92.7 | 85.0 | 57 | 30 | 10 |
| splice-junction recognition | 95.4 | 94.6 | 94.5 | 60 | 56 | 22 |

between the input features and the class predictions. In summary, these results suggest the utility of the BBP algorithm for data-mining tasks: it provides good predictive accuracy on a variety of interesting, real-world tasks, and it produces syntactically simple hypotheses, thereby facilitating human comprehension of what it has learned. Additional details concerning these experiments can be found elsewhere (Craven, 1996).

## 4.2 An Unsupervised Method

As stated in the Introduction, unsupervised learning involves the use of inductive methods to discover regularities that are present in a data set. Although there is a wide variety of neural-network algorithms for unsupervised learning, we discuss only one of them here: *competitive learning* (Rumelhart & Zipser, 1985). Competitive learning is arguably the unsupervised neural-network algorithm that is most appropriate for data mining, and it is illustrative of the utility of single-layer neural-network methods.

The learning task addressed by competitive learning is to partition a given set of training examples into a finite set of clusters. The clusters should represent regularities present in the data such that similar examples are mapped into similar classes.

The variant of competitive learning that we consider here, which is sometimes called *simple competitive learning*, involves learning in a single-layer network. The input units in such a network represent the relevant features of the problem domain, and the $k$ output units represent the $k$ classes into which examples are clustered.

The net input to each output unit in this method is a linear combination of the input activations:

$$net_i = \sum_j w_{ij} a_j .$$

Here, $a_j$ is the activation of the $j$th input unit, and $w_{ij}$ is the weight linking the $j$th input unit to the $i$th output. The name *competitive learning* derives from the process used to determine the activations of the hidden units. The output unit that has the greatest net input is deemed the winner, and its activation is set to one. The activations of the other output are set to zero:

$$a_i = \begin{cases} 1 & \text{if } \sum_j w_{ij} a_j > \sum_j w_{hj} a_j \text{ for all output units } h \neq i \\ 0 & \text{otherwise.} \end{cases}$$

The training process for competitive learning involves minimizing the cost function:

$$C = \frac{1}{2} \sum_i \sum_j a_i (a_j - w_{ij})^2$$

where $a_i$ is the activation of the $i$th output unit, $a_j$ is the activation of the $j$th input unit, and $w_{ij}$ is the weight from the $j$th input unit to the $i$th output unit. The update rule for the weights is then:

$$\Delta w_{ij} = -\eta \partial C \partial w_{ij} = \eta a_i (a_j - w_{ij}).$$

where $\eta$ is a learning-rate parameter.

The basic idea of competitive learning is that each output unit takes "responsibility" for a subset of the training examples. Only one output unit is the winner for a given example, and the weight vector for the winning unit is moved towards the input vector for this example. As training progresses, therefore, the weight vector of each output unit moves towards the centroid of the examples for which the output has taken responsibility. After training, each output unit represents a cluster of examples, and the weight vector for the unit corresponds to the centroid of the cluster.

Competitive learning is closely related to the statistical method known as *k-means clustering*. The principal difference between the two methods is that competitive learning is an *on-line* algorithm, meaning that during training it updates the network's weights after every example is presented, instead of after all of the examples have been presented. The on-line nature of competitive learning makes it more suitable for very large data sets, since on-line algorithms usually converge to a solution faster in such cases.

## 5 Conclusion

We began this article by arguing that neural-network methods deserve a place in the tool box of the data miner. Our argument rests on the premise that, for some problems, neural networks have a more suitable inductive bias (i.e., they do a better job of learning the target concept) than other commonly used data-mining methods. However, neural-network methods are thought to have two limitations that make them poorly suited to data-mining tasks: their learned hypotheses are often incomprehensible, and training times are often excessive. As the discussion in this article shows, however, there is a wide variety of neural-network algorithms that avoid one or both of these limitations.

Specifically, we discussed two types of approaches that use neural networks to learn comprehensible models. First, we described rule-extraction algorithms. These methods promote comprehensibility by translating the functions represented by trained neural networks into languages that are easier to understand. A broad range of rule-extraction methods has been developed. The primary dimensions along which these methods vary are their (i) representation languages, (ii) strategies for mapping networks into the representation language, and (iii) the range of networks to which they are applicable.

In addition to rule-extraction algorithms, we described both supervised and unsupervised methods that directly learn simple networks. These networks are often humanly interpretable because they are limited to a single layer of weighted connections, thereby ensuring that the relationship between each input and each output is a simple one. Moreover, some of these methods, such as BBP, have a bias towards incorporating relatively few weights into their hypotheses.

We have not attempted to provide an exhaustive survey of the available neural-network algorithms that are suitable for data mining. Instead, we have described a subset of these methods, selected to illustrate the breadth of relevant approaches as well as the key issues that arise in applying neural networks in a data-mining setting. It is our hope that our discussion of neural-network approaches will serve to inspire some interesting applications of these methods to challenging data-mining problems.

**Acknowledgements**

The authors have been partially supported by ONR grant N00014-93-1-0998 and NSF grant IRI-9502990. Mark Craven is currently supported by DARPA grant F33615-93-1-1330.

## References

Andrews, R., Diederich, J., & Tickle, A. B. (1995). A survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-Based Systems*, 8(6).

Bishop, C. M. (1996). *Neural Networks for Pattern Recognition.* Oxford University Press, Oxford, England.

Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). *Classification and Regression Trees.* Wadsworth and Brooks, Monterey, CA.

Craven, M. & Shavlik, J. (1993). Learning symbolic rules using artificial neural networks. In *Proceedings of the Tenth International Conference on Machine Learning*, (pp. 73–80), Amherst, MA. Morgan Kaufmann.

Craven, M. W. (1996). *Extracting Comprehensible Models from Trained Neural Networks.* PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI. Available as CS Technical Report 1326. Available by WWW as ftp://ftp.cs.wisc.edu/machine-learning/shavlik-group/craven.thesis.ps.Z.

Craven, M. W. & Shavlik, J. W. (1996). Extracting tree-structured representations of trained networks. In Touretzky, D., Mozer, M., & Hasselmo, M., editors, *Advances in Neural Information Processing Systems (volume 8).* MIT Press, Cambridge, MA.

Freund, Y. & Schapire, R. E. (1996). Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning*, (pp. 148–156), Bari, Italy. Morgan Kaufmann.

Fu, L. (1991). Rule learning by searching on adapted nets. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, (pp. 590–595), Anaheim, CA. AAAI/MIT Press.

Gallant, S. I. (1993). *Neural Network Learning and Expert Systems.* MIT Press, Cambridge, MA.

Giles, C. L., Miller, C. B., Chen, D., Chen, H. H., Sun, G. Z., & Lee, Y. C. (1992). Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, 4:393–405.

Jackson, J. C. & Craven, M. W. (1996). Learning sparse perceptrons. In Touretzky, D., Mozer, M., & Hasselmo, M., editors, *Advances in Neural Information Processing Systems (volume 8).* MIT Press, Cambridge, MA.

Kohonen, T. (1995). *Self-Organizing Maps.* Springer-Verlag, Berlin, Germany.

Lawrence, S., Giles, C. L., & Tsoi, A. C. (1997). Symbolic conversion, grammatical inference and rule extraction for foreign exchange rate prediction. In Abu-Mostafa, Y., Weigend, A. S., & Refenes, P. N., editors, *Neural Networks in the Capital Markets.* World Scientific, Singapore.

Quinlan, J. (1993). *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA.

Rumelhart, D. E. & Zipser, D. (1985). Feature discovery by competitive learning. *Cognitive Science*, 9:75–112.

Saito, K. & Nakano, R. (1988). Medical diagnostic expert system based on PDP model. In *Proceedings of the IEEE International Conference on Neural Networks*, (pp. 255–262), San Diego, CA. IEEE Press.

Sethi, I. K. & Yoo, J. H. (1994). Symbolic approximation of feedforward neural networks. In Gelsema, E. S. & Kanal, L. N., editors, *Pattern Recognition in Practice (volume 4)*. North-Holland, New York, NY.

Shavlik, J., Mooney, R., & Towell, G. (1991). Symbolic and neural net learning algorithms: An empirical comparison. *Machine Learning*, 6:111–143.

Thrun, S. (1995). Extracting rules from artificial neural networks with distributed representations. In Tesauro, G., Touretzky, D., & Leen, T., editors, *Advances in Neural Information Processing Systems (volume 7)*. MIT Press, Cambridge, MA.

Towell, G. & Shavlik, J. (1993). Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13(1):71–101.

Widrow, B., Rumelhart, D. E., & Lehr, M. A. (1994). Neural networks: Applications in industry, business, and science. *Communications of the ACM*, 37(3):93–105.