EXTRACTING COMPREHENSIBLE MODELS FROM TRAINED NEURAL NETWORKS

By

Mark W. Craven

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1996

To Mom, Dad, and Susan,
for their support and encouragement.

Abstract

Although neural networks have been used to develop highly accurate classifiers in numerous real-world problem domains, the models they learn are notoriously difficult to understand. This thesis investigates the task of extracting comprehensible models from trained neural networks, thereby alleviating this limitation.

The primary contribution of the thesis is an algorithm that overcomes the significant limitations of previous methods by taking a novel approach to the task of extracting comprehensible models from trained networks. This algorithm, called TREPAN, views the task as an inductive learning problem. Given a trained network, or any other learned model, TREPAN uses queries to induce a decision tree that approximates the function represented by the model. Unlike previous work in this area, TREPAN is broadly applicable as well as scalable to large networks and problems with high-dimensional input spaces. The thesis presents experiments that evaluate TREPAN by applying it to individual networks and to ensembles of neural networks trained in classification, regression, and reinforcement-learning domains. These experiments demonstrate that TREPAN is able to extract decision trees that are comprehensible, yet maintain high levels of fidelity to their respective networks. In problem domains in which neural networks provide superior predictive accuracy to conventional decision tree algorithms, the trees extracted by TREPAN also exhibit superior accuracy, but are comparable in terms of complexity, to the trees learned directly from the training data.

A secondary contribution of this thesis is an algorithm, called BBP, that constructively induces simple neural networks. The motivation underlying this algorithm is similar to that for TREPAN: to learn comprehensible models in problem domains in which neural networks have an especially appropriate inductive bias. The BBP algorithm, which is based on a hypothesis-boosting method, learns perceptrons that have relatively few connections. This algorithm provides an appealing combination of strengths: it provides learnability guarantees for a fairly natural class of target functions; it provides good predictive accuracy in a variety of problem domains; and it constructs syntactically simple models, thereby facilitating human comprehension of what it has learned.

These algorithms provide mechanisms for improving the understanding of what a trained neural network has learned.

Acknowledgements

This thesis would not exist without the support and guidance of many others. Happily, I can now thank some of the people who played an important role in my graduate career.

My greatest professional debt is to Jude Shavlik, who has been an excellent advisor and mentor. Jude taught me a great deal about machine learning, artificial intelligence, science in general, writing, hyphenation, and numerous other important aspects of conducting a career in research. Jude provided me with the ideal balance of freedom to pursue my own ideas, and guidance to steer me in the right direction when I (often) needed it. Importantly, he also acted as a good friend along the way.

I would also like to thank the other members of my thesis committee: Chuck Dyer, Wei-Yin Loh, Olvi Mangasarian, and Yu Hen Hu. They provided insightful feedback on my thesis work, and answered numerous questions over the years. Chuck, Wei-Yin and Olvi played another important role in my education, as instructors for interesting courses. Thanks also to Grace Wahba, who served as something of an adjunct committee member. Grace influenced my work through her interesting seminar, and she has always been more than willing to answer my naive questions.

The members of the Machine Learning Research Group over the years have also played an important role in the development of this research. Carolyn Allex, Paul Bradley, Kevin Cherkauer, Rich Maclin, Dave Opitz, Nick Street, and Geoff Towell have patiently answered my questions, reviewed my papers, suffered through my practice talks, and served as sounding boards for my goofy ideas. Rich and Nick also served ably as encyclopedias of knowledge about bad movies and bad pop songs, respectively.

Ed Uberbacher and Richard Mural provided a stimulating and fun work environment at Oak Ridge National Lab one summer. In a short time, they taught me a great deal about molecular biology, bioinformatics, and the way in which good science is practiced. I also enjoyed a long-distance research relationship with Jeff Jackson. Jeff was a collaborator for the work presented in Chapter 7 in this thesis, and he also provided thoughtful comments on two other chapters. This collaboration with Jeff has been very educational and enjoyable for me, and I hope we are able to continue our work together. Andreas Weigend is also partly responsible for some of the research presented in this thesis. Andreas suggested that I apply TREPAN to his exchange-rate network, and he graciously supplied his network, data, and encouragement for the task. Bob Crites and Andy Barto kindly shared their elevator-control neural network and the accompanying simulator which was developed by James Lewis and Christos Cassandras. Bob also patiently and promptly answered a deluge of e-mail while I was setting up this domain as a testbed for TREPAN.

Thankfully, I haven't missed many meals during my stint as a graduate student. In part, this good fortune is due to the fact that the research presented herein was supported at various times by Office of Naval Research Grants N00014-93-1-0998 and N00014-90-J-1941, Department of Energy Grant DE-FG02-91ER61129, National Science Foundation Grant IRI-9002413, and a grant from NYNEX Corporation.

In addition to the folks who have contributed to my professional development while at the University of Wisconsin, I owe a great debt of gratitude to those who have helped to make my life during this period happy, rewarding and fun.

One of the best aspects of my time in Madison is the great friends that I've had here. Kurt Brown, Susan Hert, Tia Newhall, Brad Richards (I'll refrain from using his nickname here[1]), and Martha Townsend deserve special mention for being great companions in dining, disturbing movies, and other fun.

Aside from one or two incidents as a teenager, my family has always been extremely supportive of my choices in life. My tenure in graduate school has been no exception, and they have constantly provided encouragement and love. In addition to forming a supportive family, Mom, Dad, Kendra, Dave and Chloe are among my best friends.

Closer to Madison, Susan's family played an important role as my second family. It has been great fun living near them for however many years it has been. I thank them for their love and support, as well as the steady stream of cookies and golf outings.

Most importantly, I would not have been able to accomplish my goals in graduate school without the love, patience, support, and joy that Susan has provided. In addition to these intangible contributions, she also kept me in constant supply of fabulous food, and is completely responsible for all semblance of organization in my life. In our future life together, I hope that I can support her as much in her endeavors as she has supported me in mine.

[1] I'll mention it here instead: it's Corn Boy.

Contents

Abstract **iv**

Acknowledgements **v**

1 Introduction **1**

1.1 Inductive Learning 2
1.2 Evaluation Criteria for Learned Models 3
1.3 Inductive Bias and Hypothesis Representations 6
1.4 Why Use Neural Networks? 8
1.5 Rule Extraction 9
1.6 Thesis Statement 10
1.7 Thesis Overview 10

2 Background **12**

2.1 Statistical Methodology 12
2.1.1 Estimation 13
2.1.2 Hypothesis Testing 14
2.2 Decision Trees 17
2.2.1 Decision-Tree Classification 17
2.2.2 Decision-Tree Learning 18
2.3 Neural Networks 20
2.3.1 Neural-Network Classification 20
2.3.2 Neural-Network Learning 22
2.3.3 Neural Networks and Comprehensibility 24
2.4 Related Work in Rule Extraction 25
2.4.1 The Rule-Extraction Task 25
2.4.2 Global Methods 29
2.4.3 Local Methods 33
2.4.4 Limitations of Related Methods 38

3 The Trepan Algorithm **42**

3.1 Overview of the Approach 42
3.2 The Algorithm 43
3.2.1 Membership Queries and the Oracle 44
3.2.2 Drawing Query Instances 46

3.2.3 Tree Expansion 53
3.2.4 Splitting Tests 54
3.2.5 Stopping Criteria 58
3.2.6 Pruning 60
3.3 Discussion 61
3.4 Chapter Summary 63

4 Empirical Evaluation of TREPAN 64
4.1 Evaluation Criteria 64
4.2 TREPAN Applied in Classification Domains 66
4.2.1 Problem Domains 66
4.2.2 Algorithms 67
4.2.3 Results 71
4.3 TREPAN Applied in a Regression Domain 73
4.3.1 Problem Domain 74
4.3.2 Algorithms 74
4.3.3 Results 76
4.4 TREPAN Applied in a Reinforcement-Learning Domain 80
4.4.1 Problem Domain 81
4.4.2 Algorithms 82
4.4.3 Results 83
4.5 Evaluating the Components of TREPAN 86
4.5.1 The Value of M-of-N Splitting Tests 86
4.5.2 The Value of an Instance Model 88
4.5.3 The Value of Best-First Tree Growing 92
4.6 Evaluating the Enhancements to ID2-of-3+ 94
4.7 Chapter Summary 97

5 Analytical Evaluation of TREPAN 101
5.1 Computational Complexity of TREPAN 101
5.1.1 Computational Complexity of DRAWSAMPLE 102
5.1.2 Computational Complexity of CONSTRUCTMOFNTEST 103
5.1.3 Putting It All Together 103
5.2 TREPAN and Learnability 104
5.2.1 PAC Learning 105
5.2.2 Weak Learning 109
5.2.3 Agnostic PAC Learning 110
5.2.4 Discussion 112
5.3 Generality of TREPAN 113
5.4 Chapter Summary 115

6 The MOFN-SWS Algorithm: A Local Method for Extracting M-of-N Rules 116
6.1 The MOFN Algorithm 117
6.2 Extending MOFN with Soft Weight-Sharing 118
6.3 Empirical Evaluation 120

6.3.1 Algorithms 121
6.3.2 Results 123
6.4 Chapter Summary 125

7 The Boosting-Based Perceptron Learning Algorithm 127
7.1 Hypothesis Boosting and the AdaBoost Algorithm 128
7.2 The BBP Algorithm 131
7.3 Empirical Evaluation 133
7.3.1 Algorithms 134
7.3.2 Problem Domains 140
7.3.3 Measuring Predictive Accuracy 141
7.3.4 Measuring Hypothesis Complexity 142
7.3.5 Discussion 145
7.4 Analytical Evaluation 147
7.4.1 Sparse Perceptrons 148
7.4.2 PAC Learning Sparse k-Perceptrons 149
7.5 Chapter Summary 152

8 Additional Related Work 153
8.1 Finite State Automata Extraction 153
8.2 Sensitivity and Hidden-Unit Analysis 155
8.3 Visualization Methods 157
8.4 Active Learning 158
8.5 Work Related to the BBP Algorithm 160
8.6 Chapter Summary 161

9 Conclusions 163
9.1 Contributions 164
9.2 Limitations of Trepan and Future Work 167
9.3 Limitations of BBP and Future Work 171
9.4 Final Remarks 172

A Representative Trees Extracted by Trepan 173
A.1 The `Coding` Domain 174
A.2 The `Heart` Domain 175
A.3 The `NETtalk` Domain 177
A.4 The `Promoter` Domain 179
A.5 The `Voting` Domain 180
A.6 The `Exchange-Rate` Domain 182
A.7 The `Elevator-Control` Domain 184

Bibliography 186

List of Tables

1 Characteristics of the classification domains 67
2 Network sizes for the classification domains 69
3 Test-set accuracy (%) for the classification domains 71
4 Test-set fidelity (%) for the classification domains 72
5 Tree complexity (# internal nodes) for the classification domains 73
6 Tree complexity (# feature references) for the classification domains 73
7 Test-set accuracy (%) for the `exchange-rate` domain 77
8 Tree complexity for the `exchange-rate` domain 80
9 TREPAN test-set fidelity (%) for the `elevator-control` domain 84
10 TREPAN tree complexity for the `elevator-control` domain 84
11 Tree accuracy (%) for ID2-of-3+ experiments 95
12 Tree complexity (# internal nodes) for ID2-of-3+ experiments 95
13 Tree complexity for (# feature references) ID2-of-3+ experiments 95
14 Test-set accuracy (%) for the MOFN-SWS experiments 124
15 Rule-set complexity for the MOFN-SWS experiments 124
16 Test-set fidelity (%) for MOFN-SWS and TREPAN 125
17 The effect of soft weight-sharing in the `NETtalk` domain 125
18 Test-set accuracy for the BBP experiments 142
19 Hypothesis complexity (# features used) for the BBP experiments 144
20 Hypothesis complexity (# weights used) for the BBP experiments 144
21 Summary of BBP Hypotheses 145

List of Figures

1 The table considered by the McNemar test 15
2 A decision tree 18
3 A neural network 21
4 Sigmoidal transfer functions 22
5 A network and extracted rules 26
6 The local approach to rule extraction 28
7 A rule search space 29
8 A search tree for the network in Figure 5 35
9 The TREPAN algorithm 43
10 The DRAWSAMPLE function 45
11 Sampling constraints determined by tree structure 46
12 Local instance models 49
13 The DRAWINSTANCE function 50
14 Example run of DRAWINSTANCE 52
15 The CONSTRUCTTEST function 55
16 The CONSTRUCTMOFNTEST function 55
17 TREPAN fidelity and accuracy for the `exchange-rate` domain 78
18 Profit-and-loss curves for the `exchange-rate` domain 79
19 Profit-and-loss curves for the `exchange-rate` domain 79
20 Q values for states in which the TREPAN tree agrees with the network 85
21 Q values for states in which the TREPAN tree disagrees with the network . . 85
22 Fidelity curves for TREPAN and TREPAN without m-of-n splitting tests . . . 87
23 Fidelity curves for TREPAN and TREPAN with simpler instance models . . . 90
24 Fidelity curves for TREPAN and TREPAN without using query instances . . . 91
25 Fidelity curves for TREPAN and TREPAN without best-first tree expansion . 93
26 Extracting rules using the MOFN method 118
27 The AdaBoost algorithm 130
28 The WEAKLEARN function used by BBP 132
29 The Relief algorithm 137
30 Using golden-section search to find a cut-point for Relief 137
31 The Winnow algorithm 140
32 The correspondence between an FSA and a recurrent network 154
33 A representative tree for the `coding` domain 174
34 A representative tree for the `heart` domain 176
35 A representative tree for the `NETtalk` domain 178

36 A representative tree for the `promoter` domain 179
37 A representative tree for the `voting` domain 181
38 A representative tree for the `exchange-rate` domain 183
39 A representative tree for the `elevator-control` domain 185

Chapter 1

Introduction

The ability to learn from examples is an important facet of intelligence, and in the last decade has been an especially fertile area of study for researchers in artificial intelligence, statistics, cognitive science, and related fields. Algorithms that are able to learn inductively from examples have been applied to numerous difficult, real-world problems of practical interest (Widrow et al., 1994; Langley & Simon, 1995). The application of inductive learning algorithms is usually driven by two underlying goals: performance and discovery. In the first case, the goal is to use a learning method to induce a model that can be used to perform some task of interest. For example, the ALVINN system (Pomerleau, 1993) has learned to steer a motor vehicle, and the GRAIL system (Uberbacher et al., 1993) has learned to recognize genes in uncharacterized DNA sequences. As illustrated by these cases, inductive learning methods can often learn to perform tasks that we do not know how to program explicitly, or that are too difficult and time-consuming to program explicitly. Another reason to make use of inductive learning methods is for the purpose of gaining insight into a collection of data by building a descriptive model of it. In many cases the models constructed by inductive learning algorithms are humanly comprehensible and thus can lead to a better understanding of the problem domain. Inductive learning with a focus on comprehensibility is a central activity in the growing field of *knowledge discovery in databases* and *data mining* (Fayyad et al., 1996). Of course, it is often the case that a learning method is applied in a given domain for both of these purposes: to construct a system that can perform a useful task, and to get a better understanding of the available data.

Given the goals of performance and discovery, two of the criteria that are most often used to evaluate learning systems are the *predictive accuracy* and the *comprehensibility* of their learned models. Predictive accuracy (also termed *generalization*), which is usually the predominant criterion, refers to how well a given model accounts for examples that were not used in inducing the model. Comprehensibility refers to how easily we can inspect and understand a model constructed by the learning system. It is often the case, however, that the learning method which constructs the model with the best predictive accuracy is not the method that produces the most comprehensible model. Neural networks, for example, provide good predictive accuracy in a wide variety of problem domains, but produce models that are notoriously difficult to understand. In many of the problem domains for which neural networks (and other similarly opaque systems) provide good predictive accuracy, however, it is desirable to be able to understand the model induced by the learning system. This thesis explores the following question: can we take an arbitrary, incomprehensible model produced by a learning algorithm, and re-represent it (or closely approximate it) in a language that better facilitates comprehensibility?

1.1 Inductive Learning

Generally speaking, there are three types of inductive learning settings:[1] *supervised learning*, *unsupervised learning*, and *reinforcement learning*. In supervised learning, the learner is given a set of training examples 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.[2] 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. An induced model is often referred to as a *hypothesis*.

In unsupervised learning, the learner is given a set of training examples but each example

[1]In addition to inductive learning, there is also a field of study called *analytical* or *speedup* learning. Unlike inductive learning, which is concerned with systems that learn fundamentally new things, analytical learning concentrates on systems that learn to improve the efficiency of what they already know how to do.

[2]There are also supervised learning algorithms that induce *relations* (Quinlan, 1990; Muggleton & DeRaedt, 1994), as opposed to functions, from examples.

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. The nature of the models constructed by unsupervised algorithms varies greatly from method to method. For example, there are unsupervised methods that explain for their training data by estimating probability distribution functions (e.g., Silverman, 1986), constructing hierarchical categorizations (e.g., Fisher, 1987), and reducing the data to a lower dimensional space that accounts for most of its variance (e.g., Jolliffe, 1986).

Reinforcement learning involves a task that lies in between supervised and unsupervised learning. A reinforcement learner operates in a dynamic environment, and it may take actions that influence the environment. The learner may also measure its environment, and thus it has access to a set of $\vec{x}$ vectors. The learner, however, is not given a y value for each $\vec{x}$, but instead is periodically given a scalar *reinforcement signal* that is indicative of its performance. The goal is learn what action to take (a *policy*) for any given $\vec{x}$ in order to maximize some long-run measure of reinforcement.

For the most part, the work presented in this thesis is concerned with supervised learning. More specifically, the focus of this thesis is on supervised *classification learning* (or *concept learning*). In classification learning, the task is to predict a discrete-valued $\hat{y}$ for a given $\vec{x}$. In other words, each $\vec{x}$ is assigned to one of a specified set of classes. In *regression learning*, in contrast, the task is to predict a continuous-valued $\hat{y}$ for a given $\vec{x}$. Although the primary focus is on classification, the tasks of unsupervised learning, regression, and reinforcement learning all make an appearance in this thesis. The algorithm presented in Chapter 3 has a component that uses an unsupervised learning method, and the empirical evaluation of this algorithm presented in Chapter 4 presents experiments in the context of regression and reinforcement-learning tasks.

1.2 Evaluation Criteria for Learned Models

Typically the most important consideration in inductive learning is to induce a model that has a high level of predictive accuracy. For supervised learning, this means that we want a model that is able to accurately predict the y value for an $\vec{x}$ drawn from the underlying distribution of examples in the world. Of course, it is trivially easy to predict the correct y value (or the Bayes-optimal y value in noisy problem domains) for examples that were in the learner's training set. Thus, we are usually concerned with how well the model does at predicting y values for examples that were not in the training set.

Often the comprehensibility of learned models is also an important consideration. That is, does the learning algorithm encode its model in such a way that it may be inspected and understood by humans? The importance of this criterion is argued by Michalski (1983) in his *comprehensibility postulate*:

> *The results of computer induction should be symbolic descriptions of given entities, semantically and structurally similar to those a human expert might produce observing the same entities. Components of these descriptions should be comprehensible as single 'chunks' of information, directly interpretable in natural language, and should relate quantitative and qualitative concepts in an integrated fashion.* (pg. 122)

There are a number of reasons why comprehensibility is an important criterion.

- **Validation**. In order to gain confidence in the performance of a learning system, its users often want to know how it arrives at its decisions. The ability to inspect a learned hypothesis is important in such domains. It is an especially important criterion in domains, such as medical diagnosis (e.g., Wolberg et al., 1994), in which the system occupies a position of trust.

- **Discovery**. Learning systems may also play an important role in the process of scientific discovery. A system may discover salient features and relationships in the training data whose importance was not previously recognized. If the hypotheses formed by the learner are comprehensible, then these discoveries can be made accessible to human review (e.g., Hunter & Klein, 1993).

- **Explanation**. In some domains, it is not necessary to have a complete description of the learning system's induced model, but it is desirable to be able to explain classifications of individual examples (Gallant, 1993). If the learned hypothesis is understandable in such a domain, then it can be used to produce explanations of classifications made for particular cases.

- **Improving predictive accuracy**. The feature representation used for a learning task can have a significant impact on how well an algorithm is able to learn and generalize (Flann & Dietterich, 1986; Craven & Shavlik, 1993c). Learned models that can be understood and analyzed may provide insight into devising better feature representations.

- **Refinement**. Learning algorithms have been used to refine approximately-correct *domain theories* (Pazzani & Kibler, 1992; Ourston & Mooney, 1994; Towell & Shavlik, 1994). An approximate domain theory is a incomplete description of how to solve the problem at hand. In order to complete the theory-refinement process, it is important to be able to express, in a comprehensible manner, the changes that have been imparted to the theory during learning.

Another consideration that is sometimes important when selecting a learning method is the flexibility of the language used by the algorithm to represent its hypotheses. Specifically, it is desirable for learning methods to represent their models using *declarative representations* in which the model is not restricted to being used by a particular procedure for a particular task, but instead can be used by different procedures in different contexts. There are several reasons why the flexibility of learned hypotheses is sometimes an important consideration. First, one might want to *transfer* some part of the solution learned for one task to the process of learning a solution for a related task (Pratt et al., 1991; Caruana, 1996; Thrun, 1996). Such a transfer can sometimes speed the rate of learning, or lead to better solutions in the second task. The flexibility of a representation is important in the context of transfer because it can determine the selectivity with which learned knowledge can be transferred.

A second reason to prefer learning algorithms which produce flexible representations is that one might want to selectively edit and use parts of a learned hypothesis. For example, the calendar-management system developed by Mitchell et al. (1994) operates in an environment in which the target concepts vary with time. For this reason, the authors chose to use learning methods that produce hypotheses that can be decomposed into distinct rules representing predictions for specific contexts. Such a representation enables them to evaluate the performance of learned hypotheses at a fine level of granularity, and to selectively replace parts of learned hypotheses as time passes and the target concepts evolve.

A third reason to be concerned with representational flexibility is that one might want to integrate learned knowledge with existing software systems. For example, in order to apply a learned model to a large body of data, one might want to translate a learned hypothesis (or part of one) into a standard query language so that it can be used to query a database (Fayyad et al., 1996; Pazzani et al., 1996).

In addition to predictive accuracy, comprehensibility, and representational flexibility, there are other criteria that are often important considerations when evaluating a learning algorithm or the hypotheses it produces. These criteria include the efficiency of the learning algorithm in inducing hypotheses, the efficiency of the algorithm in classifying new examples, and the ease with which the algorithm can adapt a current hypothesis to newly acquired training examples (i.e., is the algorithm *incremental*). For the most part, I am not concerned about these criteria when evaluating learning algorithms in this thesis, and instead focus primarily on predictive accuracy and comprehensibility.

1.3 Inductive Bias and Hypothesis Representations

Given a fixed set of training examples, there are infinitely many models that could account for the data. Every learning algorithm has an *inductive bias* that determines the models that it is likely to return (Mitchell, 1980). There are two aspects of 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 (Minsky & Papert, 1969). 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.

There is no universally "best" learning algorithm (Wolpert, 1995). That is, there is no learning method that provides superior predictive accuracy for all problems. Therefore, often one of the tasks involved in tackling a learning problem is to try to select the algorithm with the most suitable inductive bias for the problem. This is typically done by estimating the predictive accuracy of induced classifiers using a method such as cross validation (more on this in Chapter 2).

As suggested by this discussion of inductive bias and the earlier discussion of representational flexibility, learning algorithms differ considerably in how they represent induced hypotheses. For example, there are learning algorithms that represent their hypotheses as decision trees (Breiman et al., 1984; Quinlan, 1993), decision lists (Rivest, 1987; Clark & Niblett, 1989), inference rules (Michalski, 1983; Quinlan, 1993), neural networks (Rumelhart et al., 1986), hidden Markov models (Rabiner, 1989), Bayesian networks (Heckerman, 1995), and stored lists of examples (Stanfill & Waltz, 1986; Aha et al., 1991).

The various representations for expressing learned hypotheses differ greatly in how readily they can be inspected and understood by humans. Hypothesis languages that have a logic-like syntax (e.g., decision trees, inference rules, and decision lists), are often termed *symbolic representations* (Michalski, 1986). One purported advantage of symbolic representations is that they are usually easily understood. For example, in a decision tree it is easy to see which features are incorporated into the hypothesis, and to see the important relationships among features in the hypothesis. In contrast to symbolic learners, other learning methods (e.g., neural networks, hidden Markov models, Bayesian networks) represent their hypotheses using graph structures with real-valued parameters. Some of these hypotheses can also be quite comprehensible. In Bayesian networks, for example, the nodes of the graph usually correspond to specific features of the problem domain, and the edges correspond to known dependencies among the features. In neural networks, on the other hand, many of the nodes and edges do not have such crisp correspondences to the vocabulary of the problem domain, and neural networks therefore rarely learn comprehensible hypotheses. It is often not easy to tell by inspection which features make important contributions to a neural network's predictions, and it is often very difficult to ascertain how features interact within the hypothesis.

As with comprehensibility, learning methods also differ greatly in the flexibility of their learned models. Typically, models learned by neural networks, as well as being incomprehensible, afford little flexibility in how they can be applied. Most symbolic learning algorithms, on the other hand, represent their hypotheses using representations which are declarative in nature, and thus quite flexible.

1.4 Why Use Neural Networks?

Although learning systems other than neural networks may produce hypotheses that are hard to understand, neural networks are the learning approach most notorious for producing incomprehensible hypotheses. This and the following chapter will focus on the problem of understanding hypotheses learned by neural networks. As discussed in Chapter 2, this is a topic that has been addressed by quite a few different research groups. In Chapters 3 and 4, however, I will broaden the scope of the discussion to other types of learned hypotheses that are difficult to understand.

One may wonder, if comprehensibility is a concern in a given domain, then why apply neural networks to this problem? Instead, why not use learning methods which produce models that are more amenable to human understanding? The answer to this question is that, for some problems, neural networks have a more suitable inductive bias than competing algorithms.

In some cases, neural networks have a more appropriate *restricted hypothesis space* bias than other learning algorithms. For example, the *Q-learning* method for reinforcement learning problems (Watkins, 1989) requires that the learner represent hypotheses as continuous-valued functions, and it requires that these hypotheses be updated after each training example. Few if any symbolic learning algorithms are able to meet both of these requirements. Sequential and temporal prediction tasks provide another type of problem for which neural networks often provide the most appropriate hypothesis space. *Recurrent networks* (Jordan, 1986; Pineda, 1987), 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.

Again, this is a representational ability that few if any symbolic learning algorithms have.

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 (Atlas et al., 1989; Fisher & McKusick, 1989; Weiss & Kapouleas, 1989; Shavlik et al., 1991).

1.5 Rule Extraction

One approach to understanding a hypothesis represented by a trained neural network is to try to translate the hypothesis into a more comprehensible language. Various approaches using this strategy have been investigated under the rubric of *rule extraction*. The name *rule extraction* reflects the fact that this body of work has largely concentrated on translating neural-network hypotheses into inference-rule languages.

For the purposes of this thesis, I will define the rule-extraction task as follows:

> *Given a trained neural network and the data on which it was trained, produce a description of the network's hypothesis that is comprehensible yet closely approximates the network's predictive behavior.*

The goal of the work described in this thesis has been to develop extraction algorithms that are effective and widely applicable. There are several desiderata that have guided the development of the algorithms presented herein. I argue that algorithms for extracting symbolic representations from neural networks should address the following concerns:

- **Comprehensibility:** They should generate symbolic representations that are comprehensible by experts in the domain.

- **Fidelity:** They should produce symbolic representations that accurately model the networks from which they were extracted.

- **Scalability:** They should be scalable to networks that have large input spaces and large numbers of units and weighted connections.

- **Generality:** They should require neither special training regimes, nor restrictions on network architecture. Specifically, they should apply to networks with arbitrary topologies and activation functions.

The algorithms presented in later chapters will be evaluated with respect to these criteria. Although the primary focus of this thesis is on rule extraction for the purpose of eliciting comprehensible representations from neural networks, some of the algorithms developed in this research are also well suited to the task of rule extraction for the purpose of obtaining flexible, declarative representations of learned neural-network hypotheses.

1.6 Thesis Statement

In this thesis, I present and evaluate novel algorithms for the task of extracting comprehensible descriptions from hard-to-understand learning systems such as neural networks. The hypothesis advanced by this thesis research is that it is possible to develop algorithms for extracting symbolic descriptions from trained neural networks that: (i) produce comprehensible, high-fidelity descriptions of trained networks, (ii) scale to large networks, and (iii) apply to a broad class of network types.

1.7 Thesis Overview

The remaining chapters of this thesis are organized as follows:

- Chapter 2 provides background material for the rest of the thesis. The first section describes the statistical methods used in the thesis, and the next two sections give a brief introduction to decision trees and neural networks. The latter section also discusses why the hypotheses learned by neural networks are difficult to understand. The final section in this chapter provides an in-depth review of related work in rule extraction. This section provides context for understanding the contributions of the novel work presented in subsequent chapters.

- Chapter 3 presents the Trepan algorithm, which is the main contribution of this thesis. Trepan is an algorithm for extracting decision trees from hard-to-understand classifiers. Unlike previous approaches to rule extraction, it frames the task as an inductive learning problem. This novel perspective alleviates many of the limitations of previous approaches.

- Chapter 4 presents a detailed empirical evaluation of the Trepan approach. In the reported experiments, Trepan is applied to neural networks that were trained to solve classification, regression, and reinforcement-learning tasks.

- Chapter 5 provides an analytical evaluation of Trepan. This analysis presents a formal discussion of the scalability of the algorithm, as well as a discussion of its generality. The scalability of Trepan is considered in terms of computational complexity and in the context of formal models of learnability.

- Chapter 6 describes and evaluates a rule-extraction method that I developed in the early stages of this thesis research. Unlike Trepan, this algorithm does not approach the rule-extraction task as an inductive learning problem. As a consequence, it is not as widely applicable as Trepan.

- Chapter 7 describes and evaluates the Boosting-Based Perceptron (BBP) learning algorithm which constructively builds simple, comprehensible neural networks. BBP is not a rule-extraction method, but its underlying motivation is closely related to the motivation for the rule-extraction task: to induce comprehensible models in domains in which neural networks have a well-suited inductive bias.

- Chapter 8 describes related work not already discussed in Chapter 2.

- Finally, Chapter 9 discusses the contributions of this thesis, limitations of the work presented, and proposed future work.

Chapter 2

Background

This chapter provides background material for the remainder of the thesis. The first section describes the statistical methods used in the algorithms and experiments of this thesis. The next section gives a brief overview of decision trees. This material is relevant since the Trepan algorithm, presented in Chapter 3, induces decision trees to describe trained neural networks. Moreover, the empirical evaluation of Trepan presented in Chapter 4 involves experimental comparisons to two standard decision-tree learning methods. The third section in this chapter provides a brief introduction to neural networks, and discusses why it is difficult to understand the hypotheses learned by them. The final section surveys other work that has been done in the area of extracting rules from neural networks. The purpose of this discussion is to provide enough context to appreciate the novel aspects of the Trepan algorithm presented in the succeeding chapter; Chapter 7 provides a broader discussion of related research.

2.1 Statistical Methodology

Statistical methods play a role in this thesis in two important ways. First, they are used in various components of the algorithms described in Chapters 3, 6, and 7. Second, I use statistical methods in the experimental evaluation of these algorithms. This section briefly discusses two types of statistical methods that are important in later chapters: *estimation* and *hypothesis testing*.

2.1.1 Estimation

The task in an estimation problem is to determine the value of some parameter of interest. The experiments in this thesis, in particular, are concerned with estimating the predictive accuracy of various inductive learning algorithms, and with estimating the fidelity of representations that have been extracted from trained neural networks.

The basic method for estimating the predictive accuracy of a learning algorithm is to measure its accuracy on a set of examples that it is not allowed to access when constructing its hypothesis. Such a set is called a *test set* or a *holdout set*. Unless the size of the available data set is quite large, or unless the nature of the data somehow precludes it, a preferred method for accuracy estimation is to use *cross validation* (Stone, 1974). In k-fold cross validation, the available data is partitioned into k separate sets of approximately equal size. The cross-validation procedure involves k iterations in which the learning method is given $k - 1$ of the subsets to use as training data, and is tested on the set left out. Each iteration leaves out a different subset so that each is used as the test set exactly once. The *cross-validation accuracy* of the given algorithm is simply the average of the accuracy measurements from the individual folds. In the experiments in this thesis, accuracy and fidelity are measured using cross validation, as well as the single-test-set method.

In some cases, a learning algorithm has one or more parameters that affect how well it solves a given task. A common method for setting such parameters is to estimate the resulting accuracy (or whatever metric is of interest) for different values, and then to choose the value that provides in the best accuracy. It is not fair for the learning algorithm to use the test set for these estimates, but the learning algorithm can set aside part of its training set for this purpose. A set of data that is used by the learning algorithm to estimate the effect of such a decision, but not the overall accuracy of the algorithm, is called a *validation set* or a *tuning set*. Similarly, the method of cross validation can be applied for this purpose. When cross validation is used to set the parameter of a learning algorithm, however, it is run using *only* the data in the training set. It is imperative that the learning algorithm does not have access to the data in the test set. Otherwise, test-set estimates of the accuracy of the learned hypotheses are sure to be biased.

2.1.2 Hypothesis Testing

Hypothesis testing involves evaluating an assertion about the distribution of a random variable. Such an assertion is termed a *statistical hypothesis* (not to be confused with the hypothesis, or model, produced by a learning algorithm). Hypothesis-testing methods are used in this thesis when evaluating the performance of learning algorithms, and in some cases, when making decisions within the algorithms themselves.

One type of hypothesis test that is commonly used when evaluating inductive learning algorithms is whether two or more algorithms have significantly different performance. For example, in Chapters 4, 6, and 7, I will test the hypothesis that the models produced by one algorithm in some domain are more accurate than those produced by another algorithm. I use two types of statistical tests to compare the accuracy of one algorithm to another. The first, which is used with cross-validation runs, is the *paired-sample t-test* (Sachs, 1984). In the paired-sample t-test, we first calculate the average of the differences in accuracy measurements for algorithms A and B for k folds:

$$\overline{diff} = \frac{1}{k}\sum_{i=0}^{k}\left(accuracy_A^i - accuracy_B^i\right)$$

and the standard deviation, s, of this value. Here, $accuracy_A^i$ is the measured accuracy for algorithm A on the ith fold, and $accuracy_B^i$ is the accuracy of algorithm B on the same fold. In order for this test to be valid, the two learning algorithms must have used the same partition for cross validation. The test statistic is then:

$$t = \frac{\overline{diff}}{s/\sqrt{k}}.$$

The null hypothesis (that the two algorithms have the same level of accuracy) is rejected with $100(1-\alpha)\%$ confidence if:

$$|t| \geq t_{\alpha/2,k-1}$$

where $t_{\alpha/2,k-1}$ defines the *rejection region* for the test.[1] Note that this is a two-tailed test, meaning that the null hypothesis can be rejected either if algorithm A is more accurate than algorithm B, or vice versa. It is proper to use a two-tailed test in this situation since we have no *a priori* reason to believe that one algorithm cannot be less accurate than the other.

[1]The denominator of the t-statistic tends to underestimate the true standard deviation because cross-validation samples are correlated. Therefore the values of the t-statistic may be somewhat inflated.

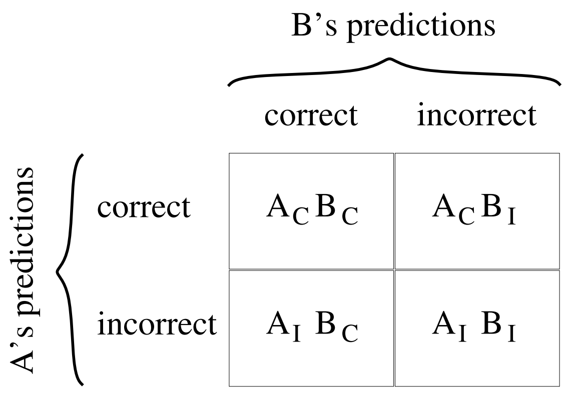


Figure 1: **The table considered by the McNemar test.** The table considers four possible outcomes when each test-set example is classified by algorithm A and algorithm B: they both produced the correct answer ($A_C B_C$); they both produced the incorrect answer ($A_I B_I$); only one algorithm made a correct prediction ($A_C B_I$ and $A_I B_C$). Each entry in the table indicates the number of test-set examples for one possible outcome.

The other method used in this thesis to test hypotheses about the predictive accuracy of learning algorithms is the *McNemar* χ^2 *test* (Sachs, 1984). This test, which I use when there is only a single test set, involves analyzing a four-entry table like that shown in Figure 1. The rows of the table indicate the correct/incorrect predictions made by one algorithm, and the columns indicate the correct/incorrect predictions made by another algorithm. Each entry in the table holds the number of test-set examples that satisfy the case represented by the entry. The null hypothesis in this test is that the entries $A_C B_I$ (representing cases in which A predicted the correct answer and B predicted the incorrect answer) and $A_I B_C$ (representing cases in which A predicted the incorrect answer and B predicted the correct one) have the same expected value: $(A_C B_I + A_I B_C)/2$. The hypothesis is tested by considering the χ^2 statistic with one degree of freedom:

$$\chi^2 = \frac{(A_C B_I - A_I B_C)^2}{A_C B_I + A_I B_C + 1}.$$

The null hypothesis (that the two algorithms have the same level of accuracy) is rejected with $100(1-\alpha)\%$ confidence if:

$$\chi^2 > \chi^2_{\alpha,1}$$

where $\chi^2_{\alpha,1}$ defines the rejection region for the test with significance level α. Note that this is a one-tailed test.

So far the discussion has focused on testing hypotheses about the accuracy of learned models. Hypothesis testing is also used within the Trepan algorithm presented in Chapter 3 in order to make several decisions. The nature of these tests is to determine whether two sets of data (for a single variable) come from different distributions or the same one. Two different tests are used for this purpose depending on the type of the data being considered. For discrete-valued variables, I use a χ^2 test, and for real-valued variables I use the *Kolmogorov-Smirnov test*.

For a discrete-valued variable with v possible values, the following χ^2 statistic is computed (Sachs, 1984; Press et al., 1992):

$$\chi^2 = \sum_{i=1}^{v} \frac{\left(\sqrt{m_B/m_A}\, m_A^i - \sqrt{m_A/m_B}\, m_B^i\right)^2}{m_A^i + m_B^i}$$

where m_A is the total number of elements in set A, m_A^i is the number of elements in set A that have the ith value for the variable, and likewise for set B. The null hypothesis (that the two sets of data come from the same distribution) is rejected with $100(1-\alpha)\%$ confidence if:

$$\chi^2 > \chi^2_{\alpha,v}$$

where $\chi^2_{\alpha,v}$ defines the rejection region for the test with v degrees of freedom.

The Kolmogorov-Smirnov test (Sachs, 1984; Press et al., 1992) compares two sets of real-valued data by first sorting each set and then converting each list into an estimator of the cumulative distribution function from which the set was drawn. Specifically, if we have m points in set A with values $x_i, i = 1, ..., m$, then $S_A(x)$ is the function giving the fraction of points to the left of a given value x. For comparing two different data sets, the Kolmogorov-Smirnov test statistic is simply:

$$D = \max_{-\infty < x < \infty} \mid S_A(x) - S_B(x) \mid .$$

The null hypothesis (that the two sets of data came from the same distribution) is rejected with $100(1-\alpha)\%$ confidence if:

$$Q_{KS}\left(\left[\sqrt{m_e} + 0.12 + \frac{0.11}{\sqrt{m_e}} \right] D \right) > \alpha$$

where m_e is the *effective* number of data points:

$$m_e = \frac{m_A\, m_B}{m_A + m_B}$$

and the function approximating the significance level is (Press et al., 1992):

$$Q_{KS}(\lambda) = 2\sum_{j=1}^{\infty}(-1)^{j-1}e^{-2j^2\lambda^2}.$$

2.2 Decision Trees

Decision-tree induction algorithms are among the most widely used methods in machine learning. Whereas neural networks are perhaps the most popular representative of the non-symbolic class of learning algorithms, decision-tree methods are the most widely used symbolic algorithms. In this section, I first describe how decision trees classify examples, and then discuss how they can be induced from training examples.

2.2.1 Decision-Tree Classification

Figure 2 depicts an example decision tree for the problem domain of heart-disease diagnosis which is explored as a testbed in Chapter 4.[2] As shown in the figure, a decision tree is a rooted, directed acyclic graph consisting of a set of internal nodes (depicted as rectangles) and a set of leaves (depicted as ovals). Each internal node in a decision tree has an associated logical test based on the features in the domain. When classifying an example, the role of an internal node is to send the example down one of the outgoing branches of the node. The decision as to which branch an example is sent down is determined by the logical expression at the node. In the simplest case, this expression considers one feature, and thus the outcome of the test is determined by the value of that feature in the given example. In some decision trees, the test may be a function of several features. In Figure 2, each internal node tests a single feature, and the outgoing branches are labeled with the possible outcomes for a given test. For example, the root of the tree looks at the feature `rest ECG`, which has three possible values: `abnormal`, `normal`, and `hypertrophy`. Similarly, the leftmost child of the root node tests the real-valued feature `cholesterol` against a threshold of 200.

The classification procedure involves starting at the root of the tree, and then traversing a path through the tree that is determined by the outcomes of the tests at the internal

[2]This tree was not produced by any decision-tree learning algorithm, but is simply made-up for pedagogical purposes.

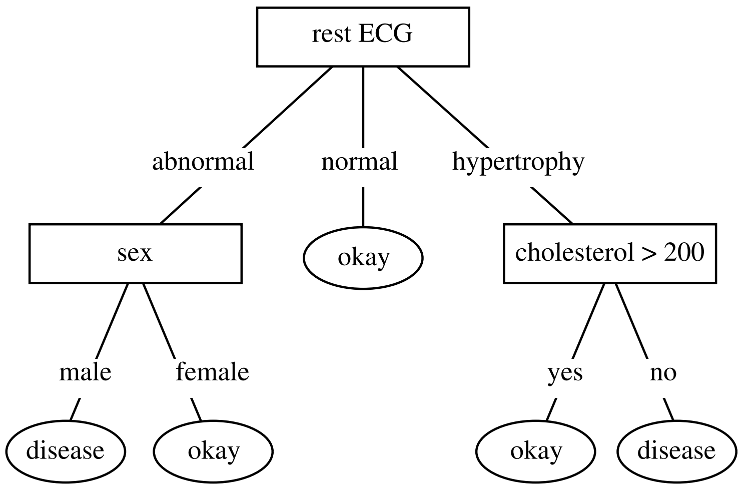


Figure 2: **A decision tree.** Internal nodes, depicted by rectangles, represent single-feature tests. Each outgoing branch of an internal node represents a possible outcome for its associated test. Leaves, depicted by ovals, correspond to class predictions.

nodes encountered along the path. The leaves of a decision tree do not have logical tests, but instead have associated class labels; in the figure, leaves are labeled either with the class `disease` or with `okay`. When an example reaches a leaf, the class associated with the leaf is the prediction made by the decision tree for that example.

2.2.2 Decision-Tree Learning

The two most widely used decision-tree induction algorithms are C4.5 (Quinlan, 1993), which arose in the artificial intelligence community, and CART (Breiman et al., 1984) which was developed in the statistics community. C4.5 is the successor to the ID3 algorithm (Quinlan, 1986). These two algorithms, and numerous variants of them, are similar in their overall structure, but differ somewhat in details. Here I focus mainly on C4.5, since it used extensively in the experiments reported in later chapters.

Decision-tree learning involves constructing a tree by recursively partitioning the training examples. Each time a node is added to the tree, some subset of the training examples are used to pick the logical test at that node. All of the training examples are used to determine the test for the root of the tree. After this test has been picked, it is used to partition the examples, and the process is continued recursively. That is, from the subset of training examples that reach a given internal node, only the examples that have the ith outcome for the test at that node are used to determine the test for the ith child of the node.

One of the key aspects of any decision-tree algorithm is selecting the test for partitioning the subset of training examples, S, that reaches a given node. C4.5 uses tests that are based on a single feature. For a discrete-valued feature with v values, C4.5 considers partitioning based on a test with v outcomes – one for possible each value. For a real-valued feature, C4.5 considers binary tests that compare the feature against various thresholds. The outcomes in this case are either that (1) the value is less than or equal to the threshold, or (2) the value is greater than the threshold. The thresholds considered by C4.5 for a real-valued split at a node are determined by the values that occur in the training examples that reach that node.

In order to pick a splitting test from a set of candidates, C4.5 uses an evaluation measure called *information gain*.[3] The information-gain criterion picks the test, T, that maximizes the information gained about the class labels of the examples in S:

$$gain(T) = info(S) - info_T(S).$$

Here $info(S)$ is the amount of information needed to identify the class of an example in S, and $info_T(S)$ is the corresponding measurement after S has been partitioned according to T. Specifically:

$$info(S) = -\sum_{j=1}^{k} \frac{freq(C_j, S)}{|S|}\, log_2\left(\frac{freq(C_j, S)}{|S|}\right)$$

where j ranges over the classes and $freq(C_j, S)$ is the number of examples in S that belong to class C_j. The information needed to identify the class of an example, given the partition T, is defined as the expected value of the information over the subsets induced by T:

$$info_T(S) = \sum_{i=1}^{n} \frac{|S_i|}{|S|} info(S_i)$$

Here i ranges over the outcomes of T and S_i is the subset of examples in S that have the ith outcome.

Another key aspect of a decision-tree algorithm is determining when to stop growing a tree. C4.5 uses several stopping criteria to decide when to make a node into a leaf. First, if

[3]Actually by default, C4.5 employs a minor variation of the information gain criterion called the *gain ratio*. For pedagogical purposes, however, I simply describe the information-gain measure.

the subset of examples that reaches a node are all members of the same class then C4.5 will not split the subset any further. Second, if C4.5 cannot find a test that results in at least two outcomes having a minimum number of examples in them, then it will stop splitting at this node. Finally, if the list of candidate tests available to use at a node is empty, then C4.5 will not partition this node further.

After C4.5 has grown a tree, it then tries to simplify it by pruning away various subtrees and replacing them with leaves. This strategy is a method for avoiding *over-fitting*. Over-fitting is the term used to describe the situation in which a learned model has been fit too closely to the training data. It is a concern because it can lead to poor predictive accuracy if the training data is noisy or if the training sample is small (which is almost always the case). C4.5's pruning method considers replacing each internal node by either a leaf or one of the node's branches. In order to decide if a change should be made, C4.5 computes a confidence interval around the resubstitution error rate[4] of the node. A change is made to a subtree if the resulting resubstitution error rate for the modified subtree is within a $C\%$ confidence interval of the unmodified subtree's error rate, where C is a parameter of the algorithm that determines how conservative the pruning process should be.

2.3 Neural Networks

There is a wide variety of neural-network architectures and learning methods for both unsupervised and supervised inductive learning tasks. The work in this thesis focuses on feed-forward neural networks applied to classification tasks, and therefore the discussion below is restricted to this particular type of neural-network approach.

2.3.1 Neural-Network Classification

As illustrated in Figure 3, a feed-forward neural network is composed of several layers of simple processing *units*. The state of a unit at any given time is represented by its *activation*, which is a real-valued number, typically in the range [0, 1] or in the range [-1, 1]. The *input layer* of a network contains units whose activations represent values for the features of the

[4] *Resubstitution error* refers to resulting error rate when a model is used to classify its training data.

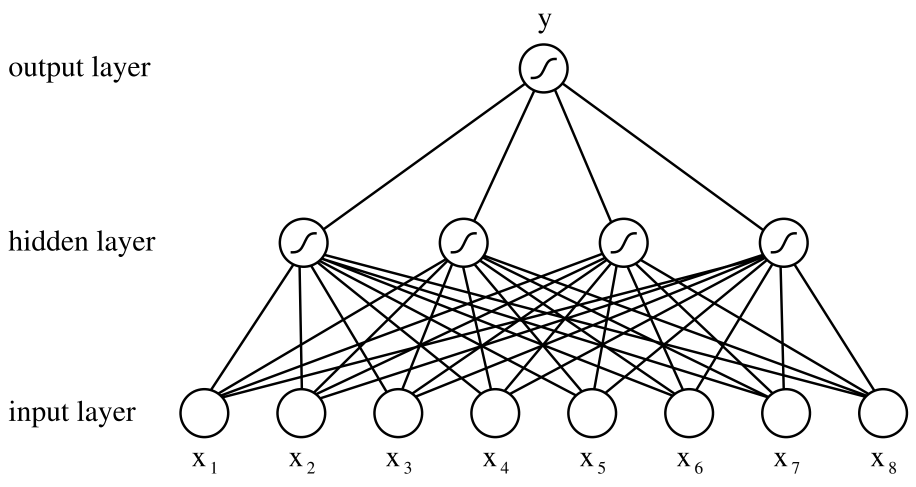


Figure 3: **A neural network.** The units in the input layer represent features in the problem domain. The unit in the output layer represents the network's predictions. Units in the hidden layer enable the network to learn and make use of "derived" features.

problem domain in which the network is being applied. Typically, a real-valued feature is represented by a single input unit, and a discrete feature with n possible values is represented by n input units. The units in the *output layer* of a network represent the decisions made by the network. Interposed between the input units and the output units, there can be a number of *hidden layers* of units. The units of a network are related by weighted connections.

A network for classification that has only input units and output units is capable of representing only linear decision boundaries in its input space (Minsky & Papert, 1969). In order to represent more complex boundaries, it necessary to add hidden units to the network. The role of hidden units is to transform the input space into another space in which it is more profitable for the output units to make linear discriminations.

Computation in a feed-forward network proceeds by setting the activation values of the input units to represent a particular instance in the problem domain. The activation of the inputs feeds forward through the weighted connections to the units at the hidden layers and then to units at the output layer. The answer provided by the network is determined by the resultant activations on the output units.

For a particular example, the net input to a unit in a hidden or output layer is given by:

$$net_i = \sum_j w_{ij} a_j + \theta_i \tag{1}$$

where w_{ij} is the weight from unit j to unit i, a_j is the activation of unit j in response to the

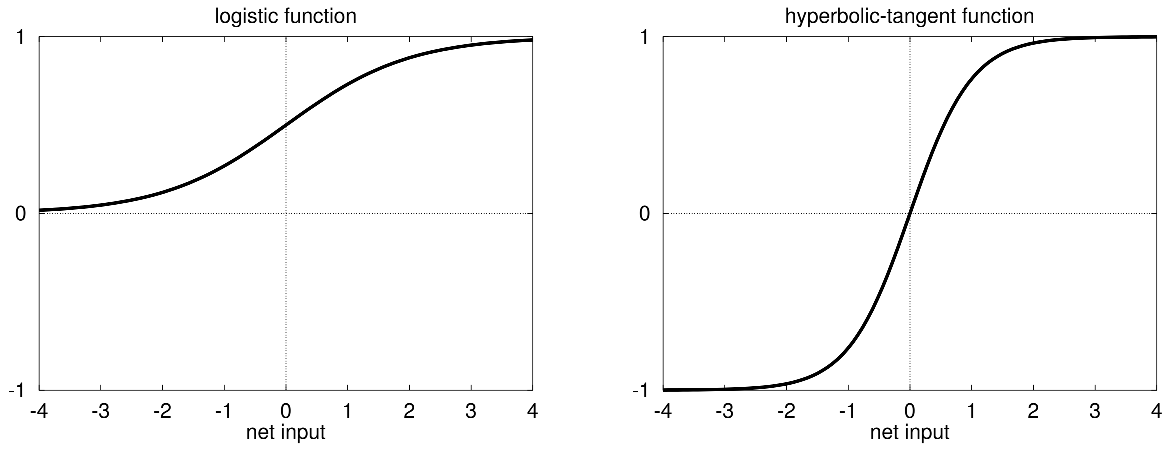


Figure 4: **Sigmoidal transfer functions.** On the left is the logistic function, which squashes its net input into the range [0, 1]. On the right is the hyperbolic-tangent function, which squashes its net input into the range [-1, 1].

example, and θ_i is the *bias* for unit i. The bias of a unit, which is an adjustable parameter, can be thought of as the unit's predisposition to have a high (or low) activation before it receives any activation signals from other units. The activation of a hidden or an output unit is determined by passing its net input through a *transfer function* (sometimes called an *activation function*). One commonly used transfer function is the *logistic function*:

$$a_i = \frac{1}{1 + e^{-net_i}}.$$

This function "squashes" the unit's net input to an activation value in the range [0, 1]. A similar transfer function is the *hyperbolic tangent function* which squashes a unit's net input into an activation value in the range [-1, 1]:

$$a_i = \frac{e^{net_i} - e^{-net_i}}{e^{net_i} + e^{-net_i}}.$$

Both the logistic function and the hyperbolic tangent function, illustrated in Figure 4, are *sigmoidal functions*. As can be seen in the figure, they are continuous approximations of a threshold function.

2.3.2 Neural-Network Learning

The most widely used neural-network learning method is the *backpropagation algorithm* (Rumelhart et al., 1986). Learning in a neural network involves modifying the weights and biases of the network in order to minimize a *cost function*. The cost function always includes an error term – a measure of how close the network's predictions are to the class labels for the examples in the training set. Additionally, it may include a complexity term that reflects a prior distribution over the values that the parameters can take (Rumelhart et al., 1995). An appropriate cost function for classification problems is the *cross-entropy function* (Hinton, 1989):

$$C = -\sum_i \sum_j \left[t_j \, ln(a_j) + (1 - t_j)\, ln(1 - a_j)\right]$$

Here i ranges the examples in the training set, j ranges over the output units of the network, t_j is the *target value* for the jth output unit for a given example, and a_j is the activation of the jth output unit in response to the example. The target value for an output unit is the activation value that it should have for a given example.

In almost all neural-network methods, the function implemented by the network is continuous and differentiable. Therefore, the cost function can be minimized by calculating its partial derivatives with respect to each of the network's parameters, and making changes to the parameters as follows:

$$\Delta\vec{w} \propto -\nabla_{\vec{w}} C$$

where $\vec{w}$ represents the vector of weights and biases in the network.

For a particular weight from unit j to unit i, the partial derivative is given by:

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial net_i}\frac{\partial net_i}{\partial w_{ij}} = \frac{\partial C}{\partial a_i}\frac{\partial a_i}{\partial net_i}\frac{\partial net_i}{\partial w_{ij}}.$$

Since hidden and output units use differentiable transfer functions, $\partial a_i / \partial net_i$ is easy to calculate. From Equation 1, we can see that the derivative $\partial net_i / \partial w_{ij}$ is simply a_j. For the output units, the term $\partial C / \partial a_i$ can be calculated directly from the cost function. For hidden units, however, it must be calculated by "backpropagating" error through the network. Specifically, to determine $\partial C / \partial a_i$ for the hidden unit i, we compute the sum:

$$\frac{\partial C}{\partial a_i} = \sum_k \left[\frac{\partial C}{\partial a_k}\frac{\partial a_k}{\partial net_k} w_{ki}\right]$$

where k ranges over the units to which unit i is connected. The process of backpropagating errors is one of "blame" assignment. The activations of the output units are determined by the activations of hidden units, which are in turn determined by the activations of the input units (or a lower level of hidden units). Thus, error at the output units may be due, not only to the weights directly connected to the outputs, but also to weights farther down in the network (the weights impinging on the hidden units).

Various optimization algorithms can be used to minimize the cost function, the most popular one being *on-line backpropagation*. This method can be thought of as a stochastic form of *gradient descent* in that weight changes do not follow the gradient of the cost function for the entire training set, but instead they follow the gradient for a single training example (White, 1989a; 1989b). The method also has a deterministic interpretation (Mangasarian & Solodov, 1993; 1994) in which the cost function does not decrease monotonically. In order to avoid large oscillations, these weight changes usually also incorporate a *momentum* term (Rumelhart et al., 1986), which is a time-decaying average of previous weight changes. Other optimization methods can be used to minimize the cost function. Standard gradient descent augmented with a momentum term is sometimes used, as is the *conjugate-gradient* method (Kramer & Sangiovanni-Vincentelli, 1989), and even second-order methods (Becker & Le Cun, 1988).

Often, network training is stopped before a local minimum in the cost function is reached. The motivation underlying this technique of *early stopping* is that over-fitting may occur if the network is trained to fit the training data too closely. One method for estimating a good stopping point is to use a validation set to monitor the predictive accuracy of the network as it is being trained. Instead of saving the network weights corresponding to the cost-function minimum, this procedure saves the weights from the iteration of the optimization method that results in the highest validation-set accuracy.

2.3.3 Neural Networks and Comprehensibility

A hypothesis learned by a 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 (i.e., the biases of sigmoid units).

Hypotheses learned by neural networks are difficult to comprehend for several reasons.

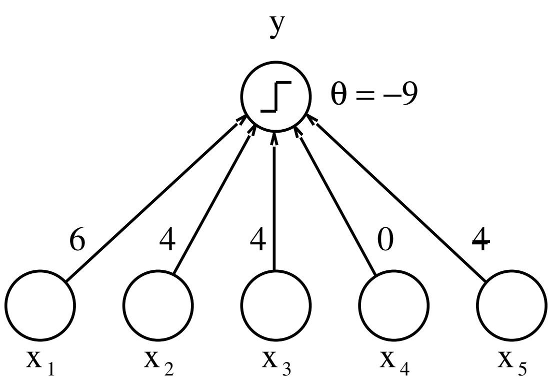
First, typical networks have hundreds or thousands of real-valued parameters. These parameters encode the relationships between the input features, $\vec{x}$, and the target value, y. Although single-parameter encodings of this type are usually not hard to understand, the sheer number of parameters in a typical network can make the task of understanding them quite difficult. Moreover, 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. Understanding the hidden units themselves is often difficult because these units often learn *distributed representations* (Hinton, 1986). Hidden units can be thought of as representing higher-level, "derived features." In a distributed representation, however, these derived features may 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.

2.4 Related Work in Rule Extraction

In this section, I review previous and contemporaneous work in the area of rule extraction from neural networks. The purpose of this discussion is to provide suitable background and context for the novel work introduced in succeeding chapters. Chapter 8 discusses additional related research that is not covered here. This chapter begins by first further defining the task of rule extraction.

2.4.1 The Rule-Extraction Task

Figure 5 illustrates the task of rule extraction with a very simple network. This one-layer network (i.e., perceptron) 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



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 5: **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.

be exactly described by a finite set of symbolic *if-then* rules since there is a finite number of possible input vectors. The symbolic rules specify conditions on the input features that, when satisfied, guarantee a given output state. In the example, I 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 I 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 ith input unit, w_i is the weight from the ith input to the output unit, and θ is the threshold parameter of the output unit. For the remainder of this chapter, I will use x_i to refer to the value of the ith 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 5 shows three conjunctive rules[5] which describe the most general conditions under

[5]A conjunctive rule is one in which the antecedent of the rule is a logical conjunction (i.e., an “and” of literals).

which the output unit has an activation of unity. Consider the rule:

$$y \leftarrow x_1 \wedge x2 \wedge \neg x_5.$$

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 that:

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

Thus, the weighted sum of these input values causes the output unit's threshold to be exceeded. But what effect can the other features have on the output unit's activation in this case? It can be seen that:

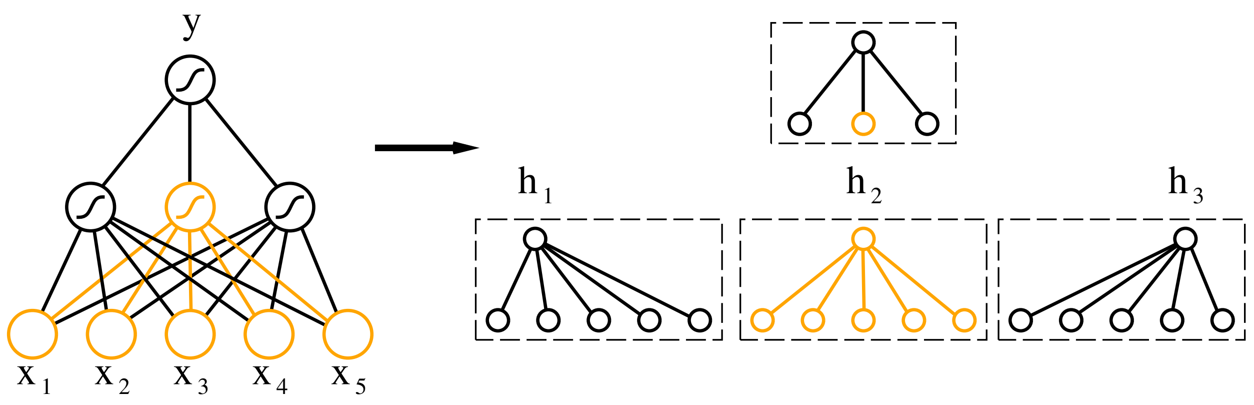
$$0 \leq a_3 w_3 + a_4 w_4 \leq 6.$$

Therefore, 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, I 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



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 6: **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.

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.

Generally speaking, there are two types of approaches to extracting rules from multi-layer networks. One approach is to extract a set of *global* rules that characterize the output classes directly in terms of the inputs. The alternative 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 6. Although *local* methods usually are less generally applicable than global methods, they are usually less computationally expensive as well.

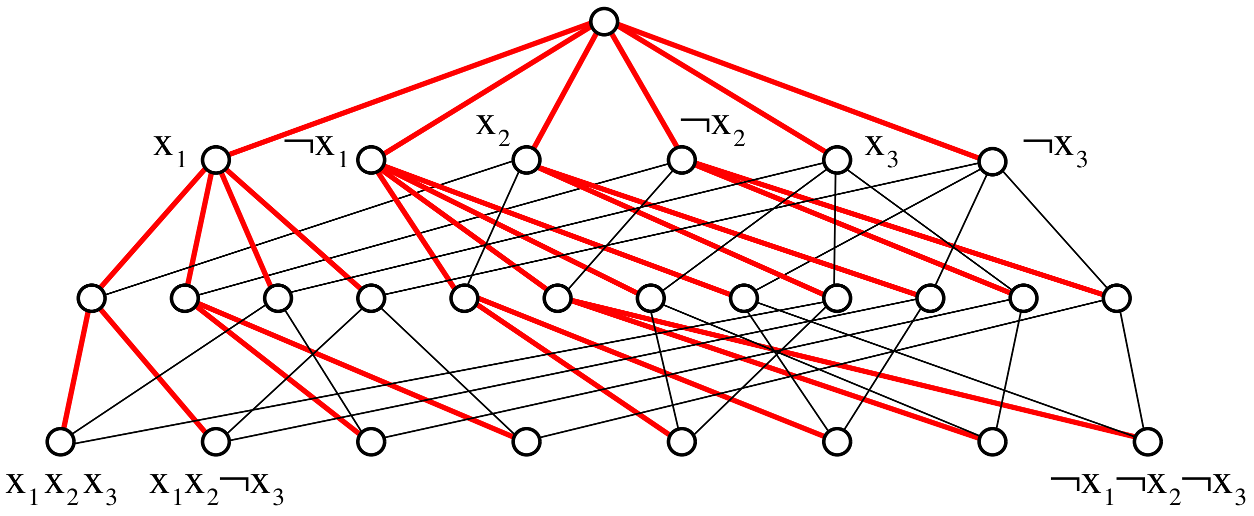


Figure 7: **A rule search space.** Each node in the space represents a possible rule antecedent. Edges between nodes indicate specialization relationships (in the downward direction). The thicker lines depict one possible *search tree* for this space.

2.4.2 Global Methods

Many previous approaches to rule extraction have set up the task as a search problem. This search problem involves exploring a space of candidate rules, and testing individual candidates against the network to see if they are valid rules.

Several research groups have investigated rule-extraction methods which operate by conducting a search through a space of possible conjunctive rules. Figure 7 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 a 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 two literals 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 7 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. One of the first rule-extraction methods developed (Saito & Nakano, 1988) employs a breadth-first search process to extract conjunctive rules in binary problem domains. To deal with the combinatorics of the problem, Saito and Nakano use two heuristics. The first heuristic limits the number of literals in the antecedents of extracted rules. Specifically, their 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. 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.

Given a candidate rule, Saito and Nakano use the following method to test the rule against the network. The input units corresponding to the positive literals are set to an activation of 1, and all other input units are set to an activation of 0. Activations are forward-propagated through the network and the decision procedure is used to classify the instance. Next, the input units corresponding the negated literals are given an activation of 1, the other inputs are not changed, and once again the instance is classified by the network. If in the first case the network classified the instance as a member of the class of interest, and in the second case it did not, then the rule is accepted. Because the depth of the search for negative literals is limited, this procedure may accept rules that do not agree with the network. For example, if the search is limited to antecedents with two negated literals, and one of the valid rules describing the network is $y \leftarrow x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4$, then this procedure might accept the overly general rule $y \leftarrow x_1 \wedge \neg x_2 \wedge \neg x_3$.

Gallant (1993) developed a similar rule-extraction technique, which like the method of Saito and Nakano, manages the combinatorics of searching for rules by limiting the search depth. The principal difference between the two approaches is the procedure used to test rules against the network. Unlike Saito and Nakano's method, Gallant's rule-testing procedure is guaranteed to accept only rules that are valid. This method 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 that it is assumed that input units whose activations are not specified by the rule could possibly take on any allowable value, and thus an interval of activations is propagated to the units in the next layer. Effectively, the network is computing, 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, given the conditions specified by the rule, the range of possible activations for the output units can be determined, and thus the procedure can decide whether to accept the rule or not. This algorithm is guaranteed to accept only rules that are valid. However, 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 the 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. Linear constraints can also be applied to the input units. For example, Thrun shows how VIA can be used to test m-of-n rules. An m-of-n rule is a Boolean expression that is specified by an integer threshold, m, and a set of n Boolean literals. An m-of-n expression is satisfied when at least m of its n literals are satisfied. 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.

Thrun uses two methods to explore a space of rules. As with the previously discussed techniques, he uses a breadth-first method to search for rules with discrete-valued features. For tasks with real-valued features, he uses a method that starts with training examples as seeds for rules. Each of these initial rules describes a point in the instance space. For example, one initial rule might be the following:

$$y \;\leftarrow\; x_1 = 0.5 \wedge \neg x_2 = 0.2 \wedge \neg x_3 = 0.8.$$

The rules are iteratively generalized by converting one of the rule's literals into an interval, or by increasing the bounds of an interval in an existing rule. For example, one possible generalization of the rule above is:

$$y \;\leftarrow\; x_1 \in [0.3, 0.5] \wedge \neg x_2 = 0.2 \wedge \neg x_3 = 0.8.$$

Each proposed generalization is tested against the network using VIA. Thrun has not described a method for exploring a rule spaces in domains that involve both discrete and real-valued features.

Another global rule-extraction method is the algorithm developed by Tchoumatchenko and Ganascia (1994) for extracting *majority-vote rules* from trained networks. A majority-vote rule is a list of literals that can be considered as either evidence for, or evidence against, a particular class. Each literal is a (possibly negated) value of a discrete-valued feature. An example is classified by a set of majority-vote rules by determining the rule that has the greatest number of its literals satisfied by the example. Tchoumatchenko and Ganascia's method, which extracts one majority-vote rule per class, uses a special cost function to encourage the network weights to cluster around the values in the set {-1, 0, 1} during training. The algorithm then determines the literals for each rule by estimating whether each input unit, considered independently, tends to act as evidence for a particular class, evidence against the class, or tends to be neutral with respect to the class. This determination is made by considering the signs of the weights along each path in the network from the input to the output unit representing the class.

The rules extracted by this method may provide only a rough approximation of a network, because they do not take into account how the hidden units combine evidence from multiple input units. That is, the method assumes that the relationship between each input unit and each output unit is monotonic, which is not necessarily the case in networks with hidden units.

2.4.3 Local Methods

The rule-extraction methods discussed in the previous section 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. The nature of these methods is quite different depending upon what type of transfer function the units in the network employ. I shall first discuss methods for networks that use sigmoidal transfer functions, and then cover methods for networks that employ *local basis functions* (Moody & Darken, 1988; Poggio & Girosi, 1990) instead of sigmoids.

There are a number of local rule-extraction methods for networks that use sigmoidal activation transfer 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 ≈ 1) or "off" (activation ≈ 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. Since the local approach assumes that the network's hidden and output units can be approximated by a threshold function, it is applicable only if the hidden units use either logistic or hyperbolic-tangent transfer functions.

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 ith 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 can push the activation of the output unit away from 1. Similarly, we need consider non-negated x_i literals only for those inputs that can push the activation of output unit towards 1. 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 8 shows a rule search space for the network in Figure 5. The shaded nodes in the graph correspond to the extracted rules shown in Figure 5. 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 in the previous section, the local methods of Fu and Gallant manage search combinatorics by limiting the depth of the rule search. When the monotonicity condition holds,

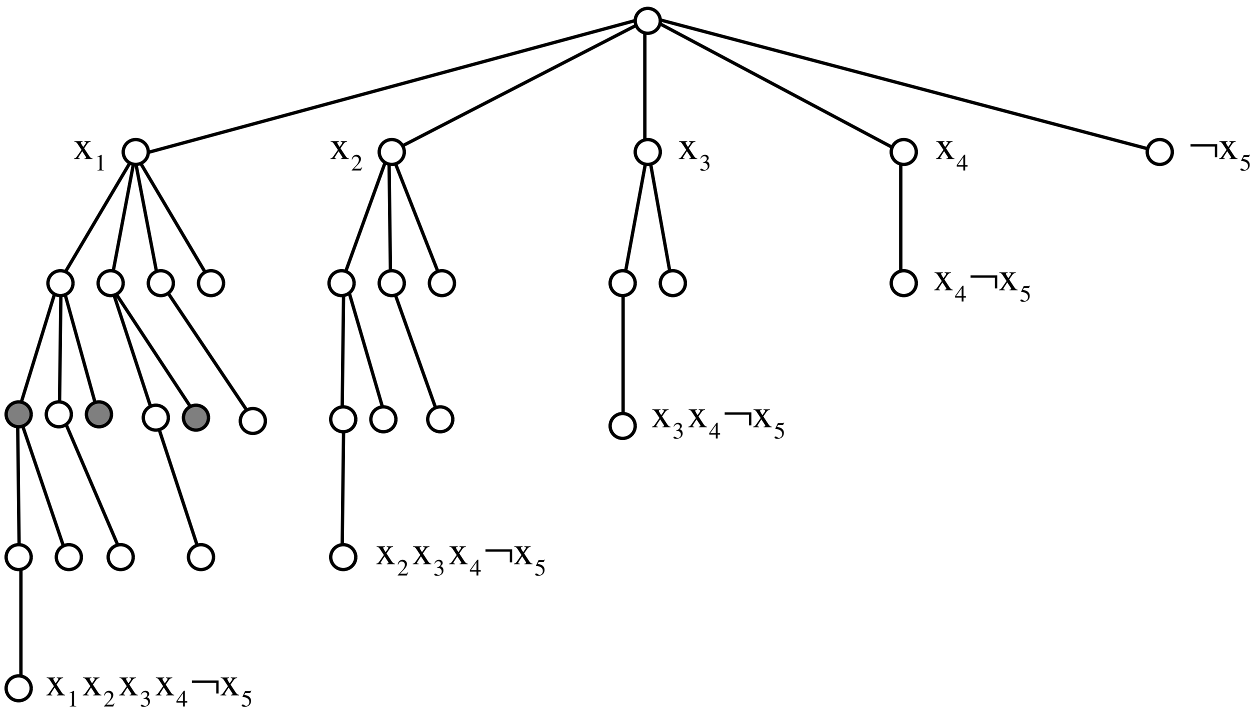


Figure 8: **A search tree for the network in Figure 5.** Each node in the space represents a possible rule antecedent. Edges between nodes indicate specialization relationships (in the downward direction). Shaded nodes correspond to the extracted rules shown in Figure 5.

the number of rules considered in a search of depth k is bounded above by:

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

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 we also know 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 5, 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 antecedents. 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.

Hayashi (1991) has described a method for extracting *fuzzy* rules from a specialized network designed for the task. Hayashi's extraction method is quite similar to the local search procedures described above. The principal difference is that the literals in the rules can represent fuzzy conditions (Zadeh, 1965). A fuzzy condition is one which has graded, as opposed to Boolean, degrees of satisfaction.

A local method developed by Towell and Shavlik (1993) searches not for conjunctive rules, but instead for rules that include m-of-n expressions. Recall that 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, 1994), 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 are initially well clustered, and empirical results indicate that the weights of knowledge-based neural networks 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, 1993a); this work is described in detail in Chapter 6.

Blasig (1994) also developed a method that uses a special cost function to encourage hidden and output units to stay in rule-like states during training. The cost function used by his method pushes weights toward the discrete values in the set $\{-6, 0, 6\}$, and pushes biases toward values that are odd multiples of 3. When the network parameters are in such a state after training, each unit can be directly translated into a single rule.

McMillan et al. (1992) devised a local rule-extraction method that simplifies the search process using a template-matching scheme. The basic idea of this approach is to extract rules by matching a unit's weight vector against of a set of canonical weight templates. Each one of these weight templates corresponds to a symbolic rule. The extraction process involves finding the weight template, and hence symbolic rule, that best fits the weight vector for a given unit. Alexander and Mozer (1995) extended this basic approach to be able to extract m-of-n rules. Their method finds the best m-of-n expression to describe a unit by considering only $O(n^2)$ templates. Moreover, it can find the best set of *nested* m-of-n expressions to describe a unit by considering $O(n^3)$ templates.

Several of the methods described here modify the network-training process in order to facilitate rule-extraction. For example, Craven and Shavlik use a special cost function, Fu uses sigmoidal functions with a steep slope, and McMillan et al. periodically adjust weights to match templates during training. Another rule-extraction method in this vein is that of Setiono and Liu (1995). Their approach actively modifies the network-training process in order to simplify the task of rule extraction. First, the approach tries to minimize the number of weights in a network by iteratively pruning weights and by using a cost function that encourages weights to "decay" towards 0. Next, the hidden units are discretized and the network is re-trained to compensate for this discretization. The discrete states for each hidden unit are determined by clustering the activation values that occur for the unit when the network is used to classify the training data. Finally, a local method is used to extract rules for the hidden and output units. Like the method of Saito and Nakano, the search for rules is guided using training examples as seeds. Moreover, like the method of Saito and Nakano, this method may not find all of the rules that describe a unit. Setiono and Liu have demonstrated that their method is able to extract concise rule-sets from networks that have a small number of hidden units. The primary limitation of their approach is that it lacks generality because it makes so many assumptions about the network and how it was trained.

The local rule-extraction methods discussed so far are designed for networks that use sigmoidal transfer functions for their hidden and output units. Another class of local methods has been developed for networks that use *local basis functions* (Moody & Darken, 1988; Poggio & Girosi, 1990) for their hidden units. Local basis functions are so named because they are designed to respond to localized patterns in their input space. Rule extraction in these networks is greatly simplified because there is no need for a search process. Each local-basis function can be directly translated into a conjunctive rule. Tresp and Hollatz (1992) have investigated a method for networks that use the most common type of local basis function: multivariate Gaussians. Andrews and Geva (1995) developed a method for networks that employ what they term "local bumps." These local-bump functions are actually composed from sigmoid functions, but rules are extracted directly from the local bumps, instead of the constituent sigmoids. As with the method of Tresp and Hollatz, extracting rules from these networks involves a straightforward process of translating each local function directly into a rule.

Finally, Tan (1994) has presented a rule extraction method for complex neural-network architecture called the Fuzzy ARTMAP (Carpenter et al., 1992). As is the case for networks with local basis functions, rule-extraction in this context involves directly translating parts of the network architecture into rules. One novel aspect of this approach is that each extracted rule has an associated *certainty factor* which is used in classifying test cases.

2.4.4 Limitations of Related Methods

Although quite a few researchers have invested effort in developing methods for extracting symbolic representations from neural networks, the current arsenal of algorithms for this task suffers from significant limitations. Broadly, the most significant of these limitations are that current methods lack generality, scalability, or both.

As discussed in Section 1.5, the term *generality* refers to the breadth of the class of networks to which an extraction method can be applied. Some methods have limited applicability in that they require that a special training procedure be used for the network (e.g., Blasig, 1994; Craven & Shavlik, 1993a; McMillan et al., 1992; Setiono & Liu, 1995;

Tchoumatchenko & Ganascia, 1994; Towell & Shavlik, 1993). Some methods are limited in their applicability because they impose restrictions on the network architecture (e.g., Andrews & Geva, 1995; Hayashi, 1991; Tan, 1994; Tresp et al., 1992), or because they require that hidden units use sigmoidal transfer functions (e.g., Alexander & Mozer, 1995; Fu, 1991; Gallant, 1993; Saito & Nakano, 1988; Sethi & Yoo, 1994). In fact, some of the abovementioned methods place restrictions on both the network's architecture and training regime.

It is important to note that a neural network with a special architecture or training procedure will likely have a different inductive bias than a network with a "standard" architecture trained using backpropagation. Consider a situation in which such a "standard" network has a more suitable inductive bias than a specialized method. Although we can run the specialized method to get comprehensible hypotheses in the given problem domain, we have really dodged the issue of how to get comprehensible hypotheses from the learning method that has the best inductive bias in the domain.

Although the restrictions imposed by some of these rule-extraction methods are relatively mild (e.g., the requirement that hidden units use sigmoids), many of the algorithms also make assumptions about the behavior of the trained network. For example, all of the local methods assume that the hidden and output units in the network behave like threshold functions, and thus can be closely approximated by them. Hidden units in trained networks often do not adhere to this assumption; they often make use of graded activation values. It is often the case that this is a poor assumption even for output units, especially in the case of multi-class problems in which the decision procedure does not threshold the output activations, but simply predicts the class associated with the most active output unit. Some local methods also assume that hidden units in the network are locally meaningful, and thus that rules describing the behavior of individual hidden units will be comprehensible. This assumption is also often not valid. In some cases, very large, complex networks provide the best predictive accuracy (e.g., Opitz, 1995). In such networks, local descriptions of hidden units make for very large, complicated rule sets. Another problem with local descriptions of hidden units is that trained networks often employ a large set of hidden units to represent a smaller set of "derived" features (Weigend, 1994). In such cases, local hidden-unit descriptions do not correspond very well to the derived features that the network has learned.

Another limitation of many rule-extraction methods is that they are designed for problems that have only discrete-valued features. Many, if not most, inductive learning applications involve both real-valued and discrete-valued features.

A final comment regarding generality: although this chapter has focused on the task of extracting symbolic rules from neural networks, there are other types of classifiers that learn hypotheses which are difficult to understand. For example, instance-based classifiers (which represent hypotheses as lists of examples) and ensembles of classifiers (in which a collection of classifiers represents the hypothesis) are often quite opaque. Unlike the approach presented in the next chapter, none of the rule-extraction methods discussed in this chapter is able to extract symbolic descriptions from an arbitrary classifier.

The other dimension along which many rule-extraction methods are lacking is scalability. Specifically, there are two scalability issues that are of concern. The first issue is how does the efficiency of the method change as a function of the size of the problem. The measures of size that are relevant here are the dimensionality of the instance space for the problem, and the number of units and weights in the network. As discussed throughout this chapter, the size of a rule space grows exponentially with input dimensionality. Some methods are able to efficiently explore large rule spaces, and some are not. I will return to this point momentarily.

The second issue of scalability is how the complexity of extracted rule sets grows with the size of the instance space and the complexity of the network's hypothesis. Although it is more difficult to evaluate rule-extraction algorithms along this dimension than along the dimension of efficiency, we can make a few general observations. The first is that local methods that extract rule sets for each hidden unit do not scale well with increasing network size. The sizes of the rule sets extracted by these methods tends to increase with network size, even though the hypotheses represented by larger networks may not be any more complex than those represented by smaller networks. A second observation is that methods which extract only conjunctive rules do not scale well to difficult problems. There is a fair amount of empirical evidence that indicates that conjunctive-rule descriptions of network hypotheses are often very large and incomprehensible (Saito & Nakano, 1988; Towell & Shavlik, 1993; Craven & Shavlik, 1994; Thrun, 1995). Finally, most of the rule-extraction algorithms discussed in this chapter are not able control the trade-off between rule-set fidelity and comprehensibility. In difficult problem domains, it may not be possible to describe a network with a rule set that characterizes the network to a high degree of fidelity, yet is simple and comprehensible. In such cases, it is desirable to have a rule-extraction method that is able to trade off the complexity of the extracted rule set with its fidelity to the network. None of the techniques described in this section enable this trade-off in a controlled manner.

It is interesting to note that the rule-extraction methods that measure up best along the dimension of generality are the ones that suffer most along the dimension of scalability. For example, Thrun's VIA algorithm for testing candidate rules is perhaps the most general of the described methods. It is suitable for problems with real-valued and discrete-valued features, and the only restriction it places on networks is that they use transfer functions that are continuous and monotonic. The method suffers with respect to scalability, however, because it has not been coupled with an efficient search technique and because it often extracts overly-specific rules. On the other hand, Alexander and Mozer's algorithm for extracting m-of-n rules is an exemplar of an algorithm that is efficiently able to explore large rule spaces, but it is rather limited in generality. It requires that the network use sigmoidal transfer functions, it assumes that the network has learned local representations, and it is only applicable to problems with discrete-valued features.

In summary, although there is a wide array of algorithms for extracting symbolic descriptions from neural networks, this body of algorithms suffers from serious limitations in terms of generality and scalability. In the following chapter, I present a novel approach to rule extraction that is aimed at addressing these limitations.

Chapter 3

The Trepan Algorithm

As discussed in the previous chapter, most rule-extraction algorithms suffer from a lack of generality, lack of scalability, or both. In this chapter, I present a rule-extraction algorithm, called Trepan[1], that is designed to address these limitations. Trepan does not assume that the networks given to it have any particular architecture, nor that they were trained in any special way. In fact, Trepan is general enough that it can be applied to a wide variety of learned models – not just to neural networks. Moreover, Trepan scales well to tasks with large instance spaces and to large networks.

3.1 Overview of the Approach

Trepan is a novel approach to rule extraction in that it 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. However, unlike most inductive learning problems, in this task we have access to an *oracle* that can be used to answer *queries* during the learning process. Specifically, since the target function is simply the concept represented by the network, the network itself (or whatever model we have) can be used as the oracle. The advantage of learning with an oracle is that

[1]Although the name Trepan was really inspired by its metaphorical connotation, it also stands for TREes PArroting Networks.

```
TREPAN
Input: ORACLE(), training set $S$, feature set $F$, $min\_sample$ parameter, stopping criteria
 1.  for each example $x \in S$
 2.      class label for $x$ := ORACLE($x$)
 3.  initialize the root of the tree, $R$, as a leaf node
 4.  construct a model $M$ of the distribution of instances covered by node $R$
 5.  $query\_instances_R$ := DRAWSAMPLE( $\{\}$, $min\_sample - |S|$ , $M$ )
 6.  use $S$ and $query\_instances_R$ to determine class label for $R$
 7.  initialize $Queue$ with tuple $\langle$ $R$, $S$, $query\_instances_R$, $\{\}$ $\rangle$
 8.  while $Queue$ not empty and global stopping criteria not satisfied
 9.      remove $\langle$ node $N$, $S_N$, $query\_instances_N$, $constraints_N \rangle$ from head of $Queue$
10.      $T$ := CONSTRUCTTEST($F$, $S_N \cup query\_instances_N$)
11.      make $N$ an internal node with test $T$
12.      for each outcome, $t$, of test $T$
13.          make $C$, a new child node of $N$
14.          $constraints_C$ := $constraints_N \cup \{T = t\}$
15.          $S_C$ := members of $S_N$ with outcome $t$ on test $T$
16.          construct a model $M$ of the distribution of instances covered by node $C$
17.          $query\_instances_C$ := DRAWSAMPLE($constraints_C$,
                                             $min\_sample - |S_C|$, $M$)
18.          use $S_C$ and $query\_instances_C$ to determine class label for $C$
19.          if local stopping criteria not satisfied then
20.              put $\langle C$, $S_C$, $query\_instances_C$, $constraints_C \rangle$ in $Queue$
Return: tree with root $R$
```

Figure 9: **The TREPAN algorithm.** Extract a tree that approximates a given learned model.

the learner can make as many queries as desired, and it can make queries for those instances that provide the most information about the target function.

3.2 The Algorithm

The TREPAN algorithm is shown in Figure 9. Like conventional decision-tree induction algorithms, such as CART (Breiman et al., 1984), and C4.5 (Quinlan, 1993), TREPAN builds a decision tree by recursively partitioning the instance space. Unlike these algorithms, however, TREPAN constructs a decision tree in a best-first manner (the notion of "best" is described shortly). 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 I 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.

The process of expanding a node in TREPAN is much like it is in conventional decision-tree algorithms: a splitting test is selected for the node, and a child is created for each outcome of the test. Each child is either made a leaf of the tree or put into the queue for future expansion.

Although TREPAN has many similarities to conventional decision-tree algorithms, it is substantially different in a number of respects. The key aspects of TREPAN are described in detail below.

3.2.1 Membership Queries and the Oracle

The generality of TREPAN derives from the fact that its interaction with the network consists solely of *membership queries* (Angluin, 1988). 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 oracle returns 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. Hence, the network serves as the oracle, and answering a membership query simply involves using the network to classify an instance.

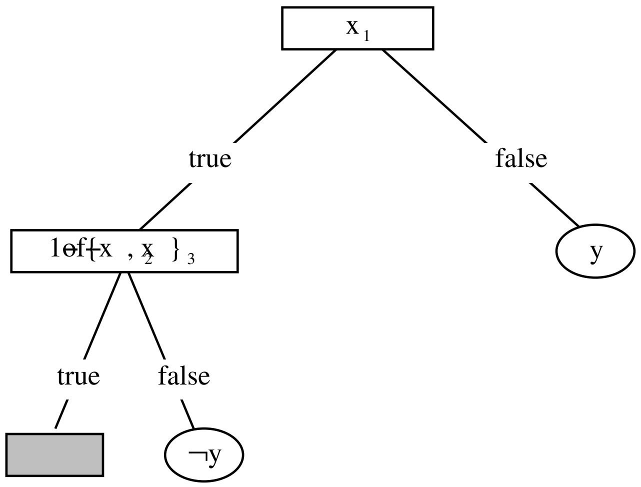
Membership queries are used in two different ways in TREPAN. Initially, they are used to get class labels for the network's training examples. Note that these class labels are not necessarily the "true" class labels, but instead they are determined by the network's classification of the instances. Since we are interested in inducing a description of the trained network, we treat the network's classifications as ground truth. TREPAN is not limited to using only the network's training data, however, it makes membership queries for other instances as well.

```
DrawSample
Input: $constraints$, sample size $m'$, model $M$

1.  $sample := \{\}$
2.  if $m' > 0$ then for $m'$ iterations do
3.        $x :=$ DrawInstance($constraints$, $M$)
4.        class label for $x :=$ Oracle($x$)
5.        $sample := sample \cup \{x\}$

Return: $sample$
```

Figure 10: **The DrawSample function.** Use model of a data distribution to draw a sample of instances satisfying the given constraints.

A major limitation of conventional tree-induction algorithms is that the amount of training data used to select splitting tests and to label leaves decreases with the depth of the tree. Thus tests and class labels near the bottom of a tree are often poorly chosen because such decisions are based on little data. In contrast, because Trepan uses query instances in addition to training examples, it always makes these decisions based on large samples. Specifically, Trepan ensures that it has at least *min_sample* instances at a node before giving a class label to the node or choosing a splitting test for it. Thus, if m training examples reach a node and $m <$ *min_sample*, then Trepan will draw and make membership queries for another (*min_sample* $- m$) instances before making any decisions at the node.

As can be seen in Figure 9, Trepan calls the DrawSample routine to get a set of query instances to use for membership queries. DrawSample is shown in Figure 10. In order to make a membership query for an instance that is not in the training set, Trepan must first select an instance to use as the query. Usually when Trepan is selecting such an instance, it is subject to a set of constraints that are determined by the location of the node in the tree. Specifically, the constraints state that instances must have outcomes for the tests at nodes higher in the tree that cause the instance to follow the path from the root of the tree to the given node. Figure 11 illustrates this point. Suppose that we want to draw a sample of instances that would reach the shaded node in the figure. The instances in such a sample must satisfy the constraints that ($x_1 = true$) and ($x_2 = true$ or $x_3 = true$). The first constraint is attributable to the test at the root node in the tree, and the second constraint is



sampling constraints for shaded node: $(x_1 = true) \wedge (x_2 = true \vee x_3 = true)$

Figure 11: **Sampling constraints determined by tree structure.** When drawing a sample of query instances at a node in the tree, the tests between the root and the given node impose constraints that the instances in the sample must satisfy. Shown here are the constraints that must be satisfied by instances selected at the shaded node.

due to the m-of-n test at the root node's left child (refer to Chapter 2 or to Section 3.2.4 for a description of m-of-n tests). The process of converting a set of constraints into an instance is handled by the DrawInstance routine which is discussed in the following section.

3.2.2 Drawing Query Instances

A key issue for Trepan is how to convert a set of constraints into an instance that can be used as a membership query. One approach would be to sample the instance space uniformly given a set of constraints. In other words, such an approach would randomly draw instances from a uniform distribution over the part of the instance space that satisfies the constraints. This is a reasonable approach if the goal of the rule-extraction task is to get a description of the network that has a uniformly high level of fidelity across the entire instance space.

Another approach to drawing query instances is to take into account the actual distribution of instances in the problem domain. The underlying motivation for this approach is that there are many domains in which it may not be possible to find a concise decision tree (or rule set) that describes the network to a high level of fidelity. Therefore, the extraction algorithm should focus on constructing a tree that has an especially high level of fidelity in those parts of the instance space where examples are most likely to be found. This is the approach Trepan takes.

There are two ways in which the underlying distribution of data may be used in the querying process. The first method is to use known *unlabeled* examples as membership queries. In some domains, although there may be few labeled examples that can be used for inducing a classifier, there are plenty of unlabeled examples – that is, examples for which the class labels are not known. For example, in the field of molecular biology, although there is an abundance of DNA sequence data, much of this data has not yet been characterized. The data that is poorly understood usually cannot be used for supervised learning tasks because examples from it cannot be given class labels reliably. However, such data sets could be drawn upon in order to get instances for membership queries. Likewise, in many reinforcement learning tasks it is easy to collect large sets of realistic instances (for example, by saving the sensor measurements of a robot). Although this data cannot be used for reinforcement learning unless the reward signal is collected as well, such a signal is not needed in order to use the data for query instances. Similarly, for some reinforcement learning tasks there exists a good computational model of the problem domain that could be used to generate instances (e.g., Crites & Barto, 1996). This situation is discussed further in Chapter 5.

The second distribution-based method to querying – which is investigated in depth in the following chapter – is to construct a model of the underlying distribution of data, and to use this model in a generative manner to draw instances. Although Trepan could employ sophisticated domain-specific models for this purpose, by default it uses a fairly simple approach based on modeling the marginal distributions[2] of individual features.

Trepan uses *empirical distributions* to model discrete-valued features, and *kernel density estimates* (Silverman, 1986) to model continuous features. The empirical distribution of a feature is simply the distribution of values that occurs in a given sample of the feature. Thus, the empirical distribution for a discrete-valued feature is represented by a parameter for each possible value of the feature indicating the frequency of that value in the training set. The kernel density estimation method used by Trepan models the probability density

[2]The *marginal distribution* of a random variable is its distribution independent of any other variables.

function for real-valued feature x as:

$$f(x) \; = \; \frac{1}{m} \sum_{j}^{m} \left[\frac{1}{\sqrt{2\pi}\sigma} \, e^{-\left(\frac{x-\mu_j}{2\sigma}\right)^2} \right]$$

where m is the number of training examples used in the estimate, μ_j is the value of the feature for the jth example, and σ is the width of the Gaussian kernel. Trepan sets σ set to $1/\sqrt{m}$. This kernel density estimation procedure has the property of *consistency*, meaning that as the size of training set tends to infinity, the estimate of the density function converges to the true function (Devroye, 1983; John & Langley, 1995). Since the value of σ is inversely proportional to the available data, however, the method produces smooth, near-Gaussian estimates when training data is scarce.

This method of modeling the underlying data distribution suffers from one significant limitation: since it estimates only marginal distributions, it does not take into account dependencies among features. Trepan partially overcomes this limitation by estimating the marginal distributions *locally* as it grows a decision tree. That is, instead of just estimating the marginal distributions once using all of the training data, Trepan constructs these estimates specific to certain nodes in the tree using only the training examples that reach those nodes. This scheme is illustrated in Figure 12. The advantage of constructing these estimates locally is that some of the conditional dependencies are captured, and thus a more accurate model of the true distribution is formed. A local estimate of a feature's distribution represents some feature dependencies because it is conditionally dependent on the outcomes of the tests that lie between the root of the tree and the node of interest.

Although local models may provide more accurate estimates than a global model by taking into account feature dependencies, in some cases they may instead provide worse estimates because they are based on less data. To handle this trade-off, Trepan uses a statistical test to decide whether or not to use the local model for a node. In this test, Trepan compares the distribution of training examples at the node of interest to the distributions at the next highest ancestor in the tree at which a local model was used. If the distributions are significantly different, then Trepan uses the newly computed distributions as a local model, otherwise it uses the ancestor's model.

To decide if the distributions are significantly different, Trepan compares the marginal

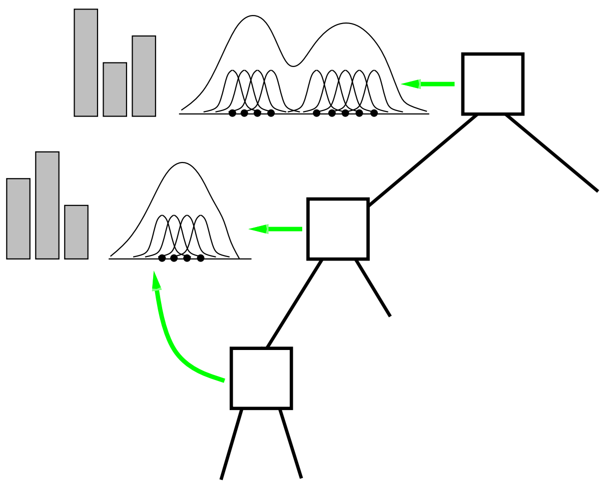


Figure 12: **Local instance models.** The figure shows local instance models for a problem with a three-valued feature and a real-valued feature. The distributions are significantly different at the top two nodes shown. The lowest node uses the local model of its parent because its distribution (not shown) is not significantly different than its parent's.

distributions for each unconstrained feature separately using a χ^2 *test* for discrete-valued features and the *Kolmogorov-Smirnov test* for real-valued features (see Chapter 2 for descriptions of these tests). An unconstrained feature is one whose value is not constrained by tests at internal nodes that are ancestors in the tree. Trepan rejects the null hypothesis (that the distributions at the two nodes are the same) if the marginal distributions for any feature are significantly different. Since each feature presents an opportunity to spuriously reject the null hypothesis, Trepan uses the *Bonferroni correction*[3] (Rice, 1995) to adjust the significance level of the overall test downward for the individual tests. Note that if a node has very little data on which to base its model, then it is unlikely that the null hypothesis will be rejected. Similarly, if two distributions are statistically indistinguishable, then Trepan will not reject the null hypothesis, and instead will prefer the distribution that is based on more data.

Figure 13 shows the DrawInstance routine which uses a model to draw instances. The algorithm takes as input a model of the marginal distribution $f(x_i)$ for each feature x_i along with the set of constraints that define allowable instances. Recall that these constraints are

[3]Suppose that we are testing n hypotheses at significance level α, and the null hypotheses are true for all of them, then the probability that one of the null hypotheses is falsely rejected is as high as $n\alpha$. The Bonferroni correction involves testing each hypothesis at the level n/α to ensure that the overall significance level is less than or equal to α.

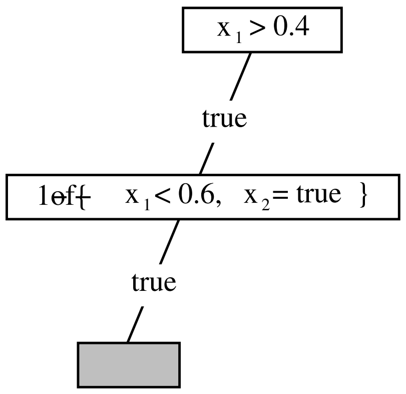
```
DrawInstance
Input: $constraints$, model which specifies probability function $f(x_i)$ for each feature $x_i$

 1.  for each feature $x_i$
 2.      $hard\_constraints_i$ := non-disjunctive constraints on feature $x_i$
 3.      $g(x_i) := f(x_i \mid hard\_constraints_i)$
 4.  for each disjunctive constraint (i.e., $m$-of-$n$ test), $T$
 5.      while $T$ not satisfied                              /* satisfy another constraint */
 6.          for each literal $l_{ij}$ on feature $x_i$ in $T$
 7.              $Pr(\text{selecting } l_{ij}) := \frac{Pr_{g(x_i)}(l_{ij})}{\sum_{ij} Pr_{g(x_i)}(l_{ij})}$
 8.          $s_{ij}$ := literal randomly selected according to $Pr(\text{selecting } s_{ij})$
 9.          $hard\_constraints_i := hard\_constraints_i \cup s_{ij}$
10.          $g(x_i) := f(x_i \mid hard\_constraints_i)$
11.  for each feature $x_i$                                    /* pick a value for each feature */
12.      $x_i$ := value randomly selected from distribution $g(x_i)$

Return: instance $x$
```

Figure 13: **The DrawInstance function.** Use a model of a data distribution to draw an instance satisfying the given constraints.

determined by the location of the node in the tree where the instances are being drawn. When there are no disjunctive constraints (i.e., when there are no disjunctive m-of-n tests), the process of drawing an instance is very simple. Trepan determines the distribution of each feature $g(x_i)$ conditioned on the constraints for that feature, and then randomly samples each of these conditional distributions. When there are disjunctive constraints, however, Trepan must ensure that each of these constraints is also satisfied. To do this, Trepan enters a loop in which it randomly selects literals of the disjunctive constraint to satisfy, until the constraint itself is satisfied. In other words, to ensure that an m-of-n expression is satisfied, Trepan needs to ensure that m of the literals in the expression are satisfied. Similarly, in cases where Trepan wants to draw an instance that *does not* satisfy a given m-of-n expression, it needs to ensure that negations of $(n-m+1)$ of the literals are satisfied. Each iteration of this loop involves first calculating the probability, $Pr_{g(x_i)}(l_{ij})$, that each literal, l_{ij}, in the constraint will be satisfied. Trepan then randomly selects a literal to be satisfied, based upon the relative probabilities of the literals being true. The selected literal s_{ij} becomes a new constraint on its associated feature x_i and Trepan updates the conditional distribution of the feature $g(x_i)$, given this constraint. This procedure ensures that instances are drawn from a distribution that is defined to be the joint distribution of the features given that the constraints are satisfied. The final step of the procedure is to draw a random value from the conditional distribution of each feature. This step is trivial for discrete-valued features. For real-valued features, TREPAN adapts a standard algorithm for generating values from kernel density estimates (Silverman, 1986, pg. 143). The adaptation of this algorithm ensures that the values it generates fall within the range specified by the conditional distribution of the feature.

When constructing m-of-n tests in its trees, TREPAN enforces a constraint that the same feature cannot be used in more than one m-of-n test lying on any path between the root and a leaf of the tree. The primary purpose of this restriction is that it prevents DRAWINSTANCE from having to solve a difficult satisfiability problem to ensure that all of the m-of-n constraints were satisfied. TREPAN avoids this situation for the sake of efficiency since the general satisfiability problem is NP-hard (Garey & Johnson, 1979).

Figure 14 illustrates the DRAWINSTANCE procedure with an example. Suppose that TREPAN is drawing a sample for the shaded node in the figure, where x_1 is a real-valued feature in the range $[0, 1]$, and x_2 is a Boolean feature. Moreover, to make things simple, suppose that TREPAN models both x_1 and x_2 with uniform distributions, where $f(x_1)$ indicates the probability density function for x_1, and $f(x_2)$ indicates the probability distribution of x_2. The first step of the algorithm is to determine the hard (i.e., the non-disjunctive) constraints on the features; in this case, only x_1 has a hard constraint. After calculating the conditional distribution, $g(x_i)$, of each feature given its constraints, DRAWINSTANCE enters the loop in which it selects literals to satisfy until the m-of-n test itself is satisfied. In this example, only one literal needs to be selected. The procedure first determines the probability that each literal would be satisfied given that values were drawn randomly from the conditional distributions. Then, since DRAWINSTANCE must select one of these literals to satisfy, it calculates the probability that each will be selected based on its likelihood relative to the other literals. In the example, we assume that the more likely literal ($x_2 = true$) is selected. DRAWINSTANCE adds this literal to the constraints for x_2, and updates the conditional distribution of the feature. Finally, since the m-of-n test is satisfied, DRAWINSTANCE uses the



Input:	$f(x_1) = \begin{cases} \frac{1}{1-0}, & 0 \le x_1 \le 1 \\ 0, & \text{otherwise} \end{cases}$	$f(x_2) = \begin{cases} \frac{1}{2}, & x_2 = true \\ \frac{1}{2}, & x_2 = false \end{cases}$
2.	$hard\_constraints_1 = \{(x_1 > 0.4)\}$	$hard\_constraints_2 = \{\}$
3.	$g(x_1) = \begin{cases} \frac{1}{1-0.4}, & 0.4 < x_1 \le 1 \\ 0, & \text{otherwise} \end{cases}$	$g(x_2) = \begin{cases} \frac{1}{2}, & x_2 = true \\ \frac{1}{2}, & x_2 = false \end{cases}$
7.	$Pr_{g(x_1)}(x_1 < 0.6) = \frac{1}{3}$	$Pr_{g(x_2)}(x_2 = true) = \frac{1}{2}$
	$Pr(\text{selecting } (x_1 < 0.6)) = \frac{2}{5}$	$Pr(\text{selecting } (x_2 = true)) = \frac{3}{5}$
8.	assume $(x_2 = true)$ randomly selected	
9.	$hard\_constraints_1 = \{(x_1 > 0.4)\}$	$hard\_constraints_2 = \{(x_2 = true)\}$
10.	$g(x_1) = \begin{cases} \frac{1}{1-0.4}, & 0.4 < x_1 \le 1 \\ 0, & \text{otherwise} \end{cases}$	$g(x_2) = \begin{cases} 1, & x_2 = true \\ 0, & x_2 = false \end{cases}$
12.	randomly draw $x_1 = 0.65$	$x_2 = true$

Figure 14: **Example run of DrawInstance.** The task is to draw an instance that will reach the shaded node in the tree. The leftmost column indicates the corresponding step in the algorithm shown in Figure 13. The middle column shows information maintained about x_1, and the rightmost column shows information maintained about x_2.

conditional distributions to randomly draw a value for each feature. In the example, the value 0.65 is drawn for x_1, and the value *true* is drawn for x_2 (although the latter feature was constrained such that there was no choice to make).

3.2.3 Tree Expansion

Unlike most decision-tree algorithms, which grow trees in a depth-first manner, Trepan grows trees using a best-first expansion. In the conventional decision-tree learning setting, the order in which the tree is expanded does not make much difference. The learning algorithm is given a fixed set of training data and grows a tree until either the training examples are sufficiently separated by class, or until some other locally evaluated stopping criteria are met. By locally evaluated stopping criteria, I mean criteria that can be decided by considering only the node that is currently being expanded (as opposed to the current state of the entire tree). Such criteria are invariant to the order in which the tree is expanded.

The rule-extraction setting differs in several respects, however. First, there is not a fixed set of data, since in addition to the training examples, we have the ability to draw query instances. Second, we assume that decision trees that provide complete models of their counterpart networks are often too complex to be comprehensible, or even impossible to attain (in cases where the networks have learned decision boundaries that are not parallel to the axes of an instance space with real-valued features). For these reasons, Trepan uses a best-first expansion so that as it adds each node it tries to maximize the gain in fidelity of the tree to the network that it is trying to model.

The notion of the best node is the one at which there is the greatest potential to increase the fidelity of the extracted tree to the network. The function used to evaluate node N is:

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

where $reach(N)$ is the estimated fraction of training examples and query 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. As the tree is being constructed, for each internal node Trepan keeps track of the fraction of instances that are sent down each branch leaving the node. The value for $reach(N)$ is then calculated as the product of these branch frequencies for all of the branches that lie on the path between the root of the tree and node N. The value for $fidelity(N)$ – where *fidelity* refers to the fraction of instances for which the tree and the network agree in their predictions – is calculated using the training examples and query instances that reach node N.

3.2.4 Splitting Tests

Selecting a test for an internal node in a decision tree involves deciding how to partition the part of the instance space covered by the internal node. Figure 15 shows the ConstructTest function which Trepan uses to determines the splitting test for a node. Whereas C4.5 and CART use single-feature tests for their splitting criteria, Trepan uses m-of-n expressions for its tests. An m-of-n expression is a Boolean expression that is specified by an integer threshold, m, and a set of n Boolean literals. An m-of-n 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 $(x_1 \wedge \neg x_2) \vee (x_1 \wedge x_3) \vee (\neg x_2 \wedge x_3)$. Murphy and Pazzani (1991) introduced the idea of using m-of-n expressions as splitting criteria in decision trees, and Trepan's method for constructing such tests is patterned after their ID2-of-3 algorithm. The function that Trepan uses to construct m-of-n tests, ConstructMofNTest, is shown in Figure 16.

Like the ID2-of-3 algorithm, Trepan uses a heuristic search process to construct its m-of-n tests. The search process begins by first selecting the best binary test at the current node measured using the *information gain* criterion (Quinlan, 1993)[4] to evaluate candidate tests. For two-valued features, a binary test separates examples according to their values for the feature. For discrete features with more than two values, Trepan considers binary tests based on each allowable value of the feature (e.g., *color=red?, color=blue?, ...*). For real-valued features, Trepan considers binary tests on thresholds (e.g., $x_1 < 0.75$, $x_1 \geq 0.75$).

Like the C4.5 algorithm, the thresholds considered by Trepan for tests on a real-valued feature are determined by the values of the feature that occur in the training set. Specifically,

[4]C4.5, on the other hand, uses a variant of information gain called the *gain ratio* criterion. This modified version of the information-gain measure is designed to control for information gain's bias toward many-valued tests. Because Trepan forms only binary tests, it does not use the gain-ratio measure.

```
ConstructTest
Input: $features$, $instances$

 1.  $C$ := MakeCandidateTests($features$, $instances$)
 2.  $best\_test$ := $true$
 3.  for each $c \in C$
 4.      if gain($c, instances$) $>$ gain($best\_test, instances$) then
 5.          $best\_test$ := $c$
 6.  $best\_m\text{-}of\text{-}n\_test$ := ConstructMofNTest($best\_test$, $C$, $instances$)

Return: $best\_m\text{-}of\text{-}n\_test$
```

Figure 15: **The** ConstructTest **function.** Determine a splitting test for an internal node in a Trepan tree.

```
ConstructMofNTest
Input: $best\_test$, $C$, $instances$

 1.  initialize $Beam$ to contain only $best\_test$
 2.  repeat
 3.      $beam\_changed$ := $false$
 4.      for each $t \in Beam$
 5.          for each $c \in C$
 6.              for each $operator \in$ { $m$–$of$–$n$+$1$, $m$+$1$–$of$–$n$+$1$ }
 7.                  $t'$ := $operator(t, c)$
 8.                  if $t'$ and $t$ are significantly different then
 9.                      if gain($t'$) $>$ gain($\tilde{t}$) where $\tilde{t}$ is worst test in $Beam$ then
10.                          replace $\tilde{t}$ by $t'$ in $Beam$
11.                          $beam\_changed$ := $true$
12.  until $beam\_changed = false$

Return: best test in $Beam$
```

Figure 16: **The** ConstructMofNTest **function.** Construct an m-of-n test for an internal node in a Trepan tree.

the set of candidate thresholds for a node is determined by sorting the values that occur in the training examples that reach the node, and then making candidates for midpoints between adjacent values. TREPAN does not admit a candidate for every one of these midpoints, however, but only for those midpoints that are between examples with different class labels. Fayyad and Irani (1992) proved that information gain is always maximized on a cut-point that falls between examples of different classes, and therefore only these cut-points need to be considered as candidates.

The selected binary test serves as a seed for TREPAN's m-of-n search process. This search uses information gain as its heuristic evaluation function, and uses the following two operators (Murphy & Pazzani, 1991):

- *m–of–n+1* : This operator adds a new literal to the set, holding m constant.

  Here are some examples of this operator being applied:

  $\textit{2-of-}\{x_1,\ x_2\} \Longrightarrow \textit{2-of-}\{x_1,\ x_2, x_3\},$

  $\textit{2-of-}\{x_1,\ x_2\} \Longrightarrow \textit{2-of-}\{x_1,\ x_2,\ x_4 > 0.5\},$

  $\textit{2-of-}\{x_1,\ x_6 = \textit{blue}\} \Longrightarrow \textit{2-of-}\{x_1,\ x_6 = \textit{blue},\ x_6 = \textit{red}\}.$

- *m+1–of–n+1*: This operator adds a new literal to the set, and increments m.

  Here are some examples of this operator being applied:

  $\textit{2-of-}\{x_1,\ x_2\} \Longrightarrow \textit{3-of-}\{x_1,\ x_2,\ \neg x_3\},$

  $\textit{2-of-}\{x_1,\ x_4 > 0.5\} \Longrightarrow \textit{3-of-}\{x_1,\ x_4 > 0.5,\ x_4 \leq 0.9\}.$

TREPAN's search permits a limited form of backtracking in that it allows these operators to add all possible values of a feature to a test. For example, the *m–of–n+1* operator might modify a test in the following way:

$$\textit{2-of-}\{x_1, x_2, x_3\} \Longrightarrow \textit{2-of-}\{x_1, x_2, x_3, \neg x_3\}.$$

In such a case, TREPAN detects that the x_3 and $\neg x_3$ literals are superfluous, and it performs the following truth-preserving modification to the test:

$$\textit{2-of-}\{x_1, x_2, x_3, \neg\ x_3\} \Longrightarrow \textit{1-of-}\{x_1, x_2\}.$$

Similarly, TREPAN allows a test with real-valued literals to be modified so that a pair of literals are complementary. For example, the *m–of–n+1* operator might modify a test in the following way:

$$\textit{2-of-}\{x_1, x_2, x_3 \leq 0.8\} \implies \textit{2-of-}\{x_1, x_2, x_3 \leq 0.8, x_3 > 0.8\}.$$

As with the analogous discrete-valued situation, TREPAN performs the following truth-preserving modification in this case:

$$\textit{2-of-}\{x_1, x_2, x_3 \leq 0.8, x_3 > 0.8\} \implies \textit{1-of-}\{x_1, x_2\}.$$

One restriction that TREPAN places on real-valued feature literals in m-of-n tests is that it does not allow a literal that is implied by another literal. Thus, TREPAN never constructs a test such as:

$$\textit{2-of-}\{x_1, x_3 > 0.5, x_3 > 0.8\}.$$

TREPAN's heuristic search uses a beam-search method with a beam width of two. The ID2-of-3 algorithm, on the other hand, uses a simple hill-climbing search, but there are potential pitfalls to this approach. For example, suppose that we were addressing an induction task where the target concept was *2-of-*$\{\neg x_1, \neg x_2, \neg x_3\}$ where the three features are Boolean variables. Suppose also that we are using a decision-tree algorithm that uses a hill-climbing method to construct m-of-n tests and that we have sufficient training data to learn this concept. Since this target concept can be concisely expressed as an m-of-n expression, we would hope that our decision-tree learner would solve the task by inducing a tree with one internal node that represented exactly this expression. Furthermore, suppose that the feature x_1 is the single feature that results in the greatest information gain. The question then is do we start the hill-climbing search using x_1 or $\neg x_1$ as the seed? If we pick the former, then we will miss finding the target concept with a single m-of-n expression. Since TREPAN uses a beam width of two, it is not forced to make an arbitrary choice in this situation, but instead can further consider both alternatives.

Another difference between ID2-of-3 and TREPAN is that TREPAN constrains m-of-n tests so that the same feature is not used in more than one m-of-n test that lies on the same path between the root and a leaf of the tree. As discussed in Section 3.2.2, the primary purpose of this restriction is that it prevents TREPAN from having to solve difficult satisfiability problems when drawing instances for membership queries. An additional reason to disallow features occurring in multiple m-of-n tests along the same path is that it enhances tree comprehensibility. This restriction obviates the need to understand complex interactions among tests in order to understand a tree.

Using a heuristic search to find m-of-n tests, it can be easy to over-fit a given set of instances. Over-fitting can occur if either the set of instances is large, or if there are many possible operator applications (because there is a large set of candidate tests), since these conditions provide many opportunities for very small increases in the information gain of an m-of-n test. To avoid this pitfall, TREPAN places an additional restriction on operator applications. Namely, to decide whether or not a given operator application is admissible, a χ^2 test is used to determine if the proposed change to the m-of-n test results in a significantly different partitioning of the instances than the partition induced by the test before the proposed change. If not, then the particular operator application is disallowed.

Finally, after TREPAN has constructed an m-of-n test, it uses a simple literal-pruning procedure to try to simplify the test. This procedure involves making one pass through the literals in the m-of-n test to see if any of them can be deleted without the information gain of the test being reduced. The literals are considered in the order that they were added to the m-of-n test, and for each literal, TREPAN considers two modifications to the test. The first candidate modification is to drop the literal from the test while holding m constant, and the second is to decrement m and drop the literal from the test.

3.2.5 Stopping Criteria

Most decision-tree induction algorithms use what I refer to as *local* stopping criteria. That is, when deciding whether to expand a node into a subtree or to make it a leaf, the decision is local in that it is based only characteristics of that node, such as the distribution of training examples that reach it. TREPAN, on the other hand, uses both local and *global* stopping criteria. A global stopping criterion is one that considers the state of the entire tree, not just the state of the node currently being expanded.

The local criterion that TREPAN uses is based on the "purity" of the set of examples covered by the node. The node becomes a leaf in the tree if, with high probability, it covers only instances of a single class. To make this decision, TREPAN first estimates the proportion of instances, $prop_c$, covered by the node that fall into the most common class, c, at the node.

If $\widehat{prop}_c = 1$, then TREPAN calculates the number of instances it needs to consider, m_L, in order to ensure that $Pr(prop_c < 1 - \epsilon) < \alpha$. In other words, m_L specifies how many instances are needed to get a sufficiently tight confidence interval around $\widehat{prop}_c$. Here, α is the significance level for the test, and ϵ specifies how tight the confidence interval around $\widehat{prop}_c$ must be. The value for m_L is calculated by considering that a confidence interval around a proportion, $\hat{p}$, specifies:

$$p \geq \frac{\hat{p} + z_\alpha^2/2n - z_\alpha\sqrt{\hat{p}(1-\hat{p})/n + z_\alpha^2/4n^2}}{1 + z_\alpha^2/n}$$

with $100(1-\alpha)\%$ confidence (Hogg & Tanis, 1983)[5]. Here, z_α represents the value such that the integral of the standard normal density from z_α to ∞ equals α. Since we want to ensure that $Pr(prop_c < 1 - \epsilon) < \alpha$, we set:

$$1 - \epsilon = \frac{\hat{p} + z_\alpha^2/2n - z_\alpha\sqrt{\hat{p}(1-\hat{p})/n + z_\alpha^2/4n^2}}{1 + z_\alpha^2/n}$$

Substituting $(\widehat{prop}_c = 1)$ for $\hat{p}$, m_L for n, and solving for m_L, we see that:

$$m_L = \frac{z_\alpha^2(1-\epsilon)}{\epsilon}.$$

If m_L instances have already reached the node of interest, then TREPAN makes it a leaf. Otherwise, TREPAN draws instances and makes membership queries until either $\widehat{prop}_c < 1$, or m_L instances have been seen and $\widehat{prop}_c = 1$. The node is made a leaf only if the latter condition is met.

TREPAN also employs global stopping criteria. The first is simply a limit on the size of the tree that TREPAN returns. This parameter, which is specified in terms of internal nodes, gives the user some control over the comprehensibility of the trees produced by enabling a user to specify the largest tree that would be acceptable for TREPAN to return.

TREPAN is able to 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

[5]Commonly, this confidence interval is calculated using $z_{\alpha/2}$ instead of z_α. However, in the situation described here, $\widehat{prop}_c = 1$, and thus we need calculate only a one-sided confidence interval instead of the usual two-sided interval.

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 target network.

Unlike Trepan, the CART and C4.5 algorithms do not rely primarily on stopping criteria to control the size of induced trees. Instead, the philosophy of these algorithms is to grow oversized trees and then to use a pruning method to find "right-sized" trees. The argument for using a pruning strategy instead of *early stopping* to control tree size is that early stopping methods are too sensitive to plateaus in the function used to decide when to stop growing the tree. For example, when growing a tree we may add a node that itself does not result in a significant increase in information gain, but that when extended by a few nodes into a subtree does result in a significant increase. Suppose also that the larger tree provides a better description of the problem at hand. In this kind of situation where patience pays off, an early-stopping method would have been likely to prematurely stop growing the tree after the initial node failed to exhibit a gain.

Despite this argument against an early stopping policy, Trepan uses one for several reasons. A primary motivation underlying Trepan is to produce comprehensible (and presumably small) decision trees. Therefore, its notion of how large a tree can be considered a "right-sized tree" is bounded by the user's limit on acceptable tree size. The other reason that Trepan uses an early-stopping method is that, empirically, it does not seem to exhibit the short-horizon problem described above. One explanation for this fact is that m-of-n tests are not as susceptible to this problem as are single-feature tests.

3.2.6 Pruning

After the stopping criteria are met, Trepan employs a very simple form of pruning before returning the final tree. The purpose of this pruning step is to detect subtrees that predict the same class at all of their leaves, and to collapse each such a subtree into a single leaf. This pruning procedure is done using a recursive, post-order traversal of the tree, so that the tree is simplified as much as possible. Note that the modifications made to a tree by this process do not change the predictive behavior of the tree at all, they simply delete extraneous internal nodes from the tree.

3.3 Discussion

Unlike most rule-extraction approaches, the symbolic descriptions produced by Trepan are decision trees instead of sets of inference rules. This distinction may not seem important, since a decision tree can easily be converted into a set of inference rules (Quinlan, 1993). There are, however, a couple of advantages to using decision trees as the extracted representation, instead of inference rules. First, the decision-tree representation provides the extraction algorithm with some degree of control over the extracted representation's complexity and fidelity. 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 this way, Trepan explores increasingly more complex, but higher fidelity, descriptions of the given network. Note, however, that this property is a function not only of Trepan's decision-tree representation, but also of its best-first strategy.

A second advantage of a decision-tree representation is that a decision tree provides a complete covering of the instance space. That is, a decision tree predicts a class for every point in the space of possible instances. This is a desirable property for an extraction algorithm because it avoids the problem of identifying which parts of the instance space have not yet been covered. For example, two of rule-extraction methods for real-valued problem domains discussed in the previous chapter (Saito & Nakano, 1988; Thrun, 1995) try to cover the instance space with extracted rules by successively generalizing training examples and testing these generalizations against the network. A problem that these approaches encounter is that as more and more rules are extracted, it becomes increasingly difficult to identify the parts of the instance that have not yet been covered by rules. In contrast, a method such as Trepan, which uses decision trees as its representation, always maintains a complete cover of the instance space. It trades off the problem of identifying uncovered parts of the instance space with the much simpler problem of identifying the parts of its representation that do not provide adequate fidelity.

Another key aspect of the Trepan algorithm is that it uses the training set and carefully selected query instances to induce a decision tree. One might wonder why not use the following approach: classify every instance in the instance space using the network and then run an ordinary decision-tree algorithm on this exhaustive set of instances. In effect, this approach would use a decision-tree method to *summarize* the given set of data. In fact, such an approach might be effective for rule extraction in problem domains with small, finite instance spaces. However, there are few real-world problems in which it is possible to exhaustively enumerate all of the instances. The instance spaces for most real-world problems are either not small (because they involve many features) or not finite (because they have real-valued features). Thus, Trepan does not assume that it is given an exhaustive set of instances, but instead selectively draws instances and makes membership queries as needed.

In earlier work (Craven & Shavlik, 1994), I developed a precursor to Trepan that also treated the rule-extraction task as an inductive learning problem. This algorithm, however, extracted inference rules instead of decision trees, and therefore suffered from the limitations of using rules discussed above. My experience with this algorithm led to the adoption of decision trees as the representation language for Trepan.

Another notable aspect of this earlier algorithm is that it used *subset queries* (Angluin, 1988) instead of membership queries. A subset query specifies a region of the instance space, and asks an oracle if all of the instances in the region are members of a given class. In other words, a subset query is equivalent to testing a hypothesized rule to see if it agrees with a network. It is easy to implement an oracle that answers subset queries for perceptrons, but more difficult to do so for multi-layer networks. Since answering a subset query, however, is equivalent to testing a rule, one could use an algorithm such as Thrun's (1995) *validity interval analysis* (described in Chapter 2) to implement a subset-query oracle. Trepan opts to use membership queries instead of subset queries for three reasons. First, membership queries are simpler and less expensive to handle. Second, they can be applied to almost any type of learned model. And finally, as mentioned in Chapter 2, the available methods for answering subset queries (such as validity interval analysis) may be able to verify only overly specific rules.

3.4 Chapter Summary

This chapter presented the Trepan algorithm which extracts decision trees from neural networks and other learned models that are hard to understand. Trepan is novel in that, unlike previous approaches, it treats the rule-extraction problem as an inductive learning task. In this learning task, the target concept is simply the function represented by the given learned model. Trepan makes use of membership queries to the model in order to induce a decision tree that describes it. The primary advantages of this approach are that it is general in its applicability, and it is scalable to large networks and problem domains.

The design of Trepan is predicated on the conjecture that, in many problem domains, it is not possible to extract concise representations that have perfect fidelity to their target models. Trepan therefore tries to extract trees that maximize the tradeoff between fidelity and concision. It does this in two ways. First, it uses a network's training set to construct models of the data distribution in the problem domain, and uses these models to generate instances for membership queries. In this way, Trepan is able to focus its description of the given model on the parts of the instance space where data is likely to occur. Secondly, Trepan grows decision trees in a best-first manner, attempting the maximize the gain in fidelity each time it expands a node in the tree.

Another key aspect of Trepan is that it uses m-of-n expressions as splitting tests in the trees it induces. These expressions often result in more concise and comprehensible trees.

The idea of viewing rule extraction as an inductive learning task was introduced in an earlier publication (Craven & Shavlik, 1994). A description of the Trepan algorithm and some of experiments reported in the following chapter were also previously published (Craven & Shavlik, 1996).

Chapter 4

Empirical Evaluation of Trepan

This chapter provides an empirical evaluation of the Trepan approach. The experiments presented here illustrate the application of Trepan to neural networks trained to solve both supervised and reinforcement learning tasks. The chapter begins by first describing the evaluation criteria considered in these experiments.

4.1 Evaluation Criteria

Chapter 1 introduced the dimensions along which rule-extraction methods should be evaluated:

- **Comprehensibility:** They should generate symbolic representations that are humanly comprehensible.

- **Fidelity:** They should produce symbolic representations that accurately model the networks from which they were extracted.

- **Scalability:** They should be scalable to networks with large input spaces and large numbers of units and weighted connections.

- **Generality:** They should require neither special training regimes, nor restrictions on network architecture.

The experiments presented in this chapter test the Trepan algorithm with respect to these desiderata. Additionally, the experiments evaluate the predictive accuracy of the trees extracted by Trepan. I now discuss each of these evaluation measures in more detail.

Measuring the comprehensibility of learned hypotheses is a problematic issue. An underlying premise of the experiments in this chapter is that *syntactic complexity* is a good indicator of comprehensibility. For a given representation language, I contend that, other things being equal, simpler descriptions are better than complex descriptions. Not surprisingly, the psychological literature supports the notion that humans prefer simple concepts (Neisser & Weene, 1962; Pinker, 1979; Medin et al., 1987). The specific measures of syntactic complexity that I use to assess the comprehensibility of decision trees are: (i) the number of internal (i.e., non-leaf) nodes in the tree, and (ii) the number of *feature references* used in the splitting tests in the tree. An ordinary, single-feature splitting test is counted as one feature reference. An m-of-n test is counted as n feature references, since such a split lists n feature values.

Another key metric is the *fidelity* of extracted decision trees. In the experiments reported in this chapter, both accuracy and fidelity are measured using test-set examples. Whereas *accuracy* is defined as the percentage of test-set examples that are correctly classified, *fidelity* is defined as the percentage of test-set examples for which the classification made by a tree agrees with its neural-network counterpart. Using held-aside test sets is the standard methodology for measuring the predictive accuracy of a learning method. It is used here to measure fidelity as well since it ensures that fidelity is measured using examples that (i) are held aside from Trepan during the extraction process, and (ii) come from the true, underlying distribution of data in the domain. The former condition is important because it means that the estimates of fidelity are unbiased. The latter condition is important because it means that fidelity is being measured with respect to the frequency that instances actually occur in the domain.

The experiments in this chapter do not directly measure the scalability of Trepan. Instead, scalability is evaluated indirectly by applying the algorithm in a variety of domains, some of which have large instance spaces. Moreover, in many of the domains, the neural networks to which Trepan is applied are quite large, and in one domain Trepan is used to extract decision trees from ensembles of large neural networks.

The generality of the TREPAN approach is also assessed indirectly. The experiments in this chapter illustrate the application of TREPAN in both supervised and reinforcement learning problems. Within the supervised learning setting, TREPAN is applied to extract rules in several classification domains, and in one regression domain. These problem domains involve both discrete and real-valued features. Moreover, the experiments in this chapter illustrate the application of TREPAN to networks with logistic, hyperbolic-tangent, and linear activation functions. And finally, as mentioned above, TREPAN is used to extract rules from neural-network ensembles in one domain.

4.2 TREPAN Applied in Classification Domains

This section evaluates TREPAN in the context of six problem domains that involve supervised classification learning. Two conventional decision-tree algorithms are also applied in these problem domains. Unlike TREPAN, which extracts decision trees from trained networks, these algorithms induce trees directly from the given training data. These two algorithms provide a baseline for (i) evaluating the predictive accuracy of the neural networks and the trees extracted from them by TREPAN, and (ii) evaluating the syntactic complexity (and hence comprehensibility) of the trees produced by TREPAN.

4.2.1 Problem Domains

The problem domains used in these experiments are summarized in Table 1. They include: recognizing protein-coding regions in *E. coli* DNA sequences (Craven & Shavlik, 1993b), diagnosing the presence of heart disease in patients (Detrano et al., 1989), mapping English text into its pronunciation (Sejnowski & Rosenberg, 1987), recognizing promoters in *E. coli* DNA sequences (Towell et al., 1990), diagnosing faults in local telephone loops (Provost & Danyluk, 1995), and predicting the party affiliation of members of the U.S. House of Representatives given their voting records (Schlimmer & Fisher, 1986). I shall refer to these as the `coding`, `heart`, `NETtalk`, `promoter`, `telephone`, and `voting` domains, respectively.

A few notes about the data sets used here are in order. The `NETtalk` task in these

Table 1: **Characteristics of the classification domains.**

domain	# examples	# features		# classes
		discrete	continuous	
protein-coding regions	20,000	64	0	2
Cleveland heart disease	303	8	5	2
NETtalk stresses	5,438	7	0	5
promoters	468	57	0	2
telephone loop diagnosis	2,686	10	13	3
Congressional voting	435	15	0	2

experiments is a simplified version of the one addressed by Sejnowski and Rosenberg. The scaled-down version used here involves learning only the stresses (but not the phonemes[1]) from a corpus of the 1,000 most common English words. The `promoter` data set used here is a larger and more complex set than the original one (Towell et al., 1990). Following Buntine and Niblett (1992), the `physician-fee-freeze` feature is not used in the `voting` domain in order to make the problem more difficult.

For the `heart`, `NETtalk`, `telephone`, `promoter`, and `voting` domains, experiments are conducted using a 10-fold cross validation methodology. Because of certain domain-specific characteristics of the `coding` data set,[2] four-fold cross-validation is used for the experiments with it. Appendix A includes additional information about these data sets.

4.2.2 Algorithms

For the `coding`, `heart`, `NETtalk`, `promoter`, and `voting` domains, I train neural networks with a single layer of hidden units (or no hidden units at all). The hidden and the output units use logistic transfer functions, and the number of hidden units used for each network (0, 5, 10, 20 or 40) is chosen using cross validation *within* each network's training set. The networks are trained using a conjugate-gradient learning method (Kramer & Sangiovanni-Vincentelli, 1989), and training continues until either (i) all of the training-set examples are correctly classified, (ii) a local minimum in the error surface is reached, or (iii) 50 search

[1]The task of learning stresses is of comparable predictive difficulty to the task of learning phonemes. The standard approach to the phoneme task, however, is to predict 21 output bits encoding the 54 possible classes. This representation is unwieldy to work with in experiments involving decision-tree (or rule-based) classifiers.

[2]This data set is partitioned so that DNA sequences from the same, and similar, genes are in the same fold.

directions have been tried.

For the `telephone` domain, TREPAN is not applied to individual networks, but instead to neural-network *ensembles*. An ensemble is a set of separately trained classifiers whose predictions are combined through some weighting scheme. Here, the ensembles are induced using the ADDEMUP algorithm (Opitz & Shavlik, 1996). The ADDEMUP algorithm uses a *domain theory* and genetic search techniques to generate an ensemble of knowledge-based neural networks (Towell & Shavlik, 1994). As mentioned in Chapter 2, a domain theory is a set of symbolic inference rules that represents an approximately correct solution to the task at hand. A knowledge-based network is a neural network in which the topology and initial weights are specified by a domain theory. The domain theory used in the `telephone` experiments was derived from the inference rules of an expert system that performs the local-loop diagnosis task. The ADDEMUP algorithm initially maps the rules of a domain theory into a set of networks using the KBANN algorithm (Towell & Shavlik, 1994), and then significantly alters the architecture of the networks by interleaving a genetic search algorithm with training. In the experiments reported here, ADDEMUP produced ensembles consisting of twenty networks each. Although ADDEMUP is often able to find solutions that have a high level of predictive accuracy, it typically represents these solutions using large, complex networks.

Table 2 summarizes the sizes of the networks used in the experiments reported here. The table lists the average number of units and parameters (weights and biases) for the networks used in each domain, and the standard deviations of these values. The figures for the `telephone` domain indicate the sizes of the ensembles, as opposed to the 20 individual networks, produced by ADDEMUP.

TREPAN is applied to each of the trained networks (or ensembles in the `telephone` domain). TREPAN employs three statistical tests: the χ^2 test that is used when constructing m-of-n splitting tests, the proportion test that is used for deciding whether a node covers instances of only one class, and the test to decide whether a local model should be used for drawing query instances. A significance level of 0.05 is used for the first test. For the stopping test, the significance level is set to 0.01. A stringent significance level is used for this test because it does not hurt to be conservative in making this decision. The philosophy

Table 2: **Network sizes for the classification domains.**

domain	# units	# parameters
`coding`	105.0 ± 0.0	2,641.0 ± 0.0
`heart`	37.5 ± 8.8	267.2 ± 253.2
`NETtalk`	205.5 ± 7.8	2,342.0 ± 1,397.4
`promoter`	238.5 ± 12.6	2,277.2 ± 2,816.4
`telephone`	942.0 ± 214.3	10,839.0 ± 7,158.8
`voting`	52.0 ± 14.3	673.0 ± 457.5

of Trepan is to use other stopping criteria to control the size of the returned tree. For the third test, the significance level is set to 0.10. This relatively high value is used because this test actually involves a series of tests, and the Bonferroni correction (Rice, 1995) is used to determine the significance level of each of these constituent tests. The Bonferroni correction is conservative in nature (it makes a worst-case assumption about the outcomes of the tests), and thus it is fairly common to use a somewhat relaxed significance level with it (Loh, 1996). The other parameter of Trepan is *min_sample*, which specifies how many instances must be considered before giving a class label or choosing a splitting test for a node. For all domains, the value of this parameter is set to 10,000. Trees are grown to a maximum size of 31 internal nodes, which is the size of a complete binary tree of depth five. Trepan holds aside 10%and from the sequence of nested trees from a given run, returns the one that has the highest level of training-set validation to the network.

As a baseline for evaluating the accuracy and comprehensibility of the trees extracted by Trepan, I also run two conventional decision-tree algorithms: C4.5 (Quinlan, 1993) and an enhanced version of ID2-of-3 (Murphy & Pazzani, 1991). C4.5 (the successor to ID3) is one of the most widely used inductive learning algorithms, and is perhaps the most popular inductive algorithm for learning "symbolic" hypotheses. ID2-of-3 is a variant of ID3 that differs from C4.5 primarily in the nature of the splitting tests it uses. Whereas C4.5 uses single-feature splitting tests, ID2-of-3 uses m-of-n expressions for tests at its internal nodes. (Chapter 3 provides a detailed discussion of how ID2-of-3 and Trepan construct m-of-n tests.)

The version of ID2-of-3 used in the experiments here, which I will refer to as ID2-of-3+, includes a number of enhancements on the original algorithm:

1. *M*-of-*n* tests can include literals on real-valued features.

2. The heuristic search process that constructs *m*-of-*n* tests uses a significance test to decide when to stop the search.

3. The *m*-of-*n* search process uses a literal-pruning routine to simplify tests after they are constructed.

4. The heuristic search process uses a beam search (with a beam width of two) instead of a hill-climbing search.

5. After trees are grown, they are pruned using the same pruning process as C4.5.

The first four of these modifications are common to TREPAN also, and are described in detail in Chapter 3. The final enhancement is described in detail elsewhere (Chapter 4 of Quinlan, 1993).

The primary parameter of C4.5 that is varied in the experiments is the pruning level. As discussed in Chapter 2, C4.5's pruning method is parameterized by a *confidence level* that specifies how liberally trees should be pruned. I evaluate unpruned trees and trees pruned with confidence levels ranging from 5% (liberal) to 95% (conservative). In the same way that the number of hidden units is selected for each neural network, cross-validation (within the training set) is used to select the pruning level for each training set. For all domains except `coding`, the default settings are used for other parameters of C4.5. In the `coding` domain, C4.5's `minobjs` parameter[3] is set to ten instead of the default value of two, since the training sets are so large.

I use the same experimental methodology for ID2-of-3+ as for C4.5. A `minobjs` parameter is used to determine when to stop growing a tree. The ID2-of-3+ experiments use the same settings for this parameter as do the C4.5 runs. Similarly, in the ID2-of-3+ experiments, pruning levels are set using cross validation within each training set as is done with C4.5. Recall that, like TREPAN, ID2-of-3+ employs a χ^2 test to decide when to stop the search process that constructs *m*-of-*n* tests. The significance level of this test is set to 0.05 as it is for TREPAN.

[3]When selecting a splitting test at a node, C4.5 requires that a split send at least `minobjs` examples down two or more branches.

Table 3: **Test-set accuracy (%) for the classification domains.** The symbol '$*$' marks results in cases where the accuracy of an algorithm is inferior to the accuracy of the neural networks at the $p \leq 0.05$ level of significance. Similarly, the symbols '$\bullet$' and '$\diamond$' mark results that are inferior to Trepan and ID2-of-3+, respectively, at $p \leq 0.05$.

method	`coding`	`heart`	`NETtalk`	`promoter`	`telephone`	`voting`
networks	94.1	84.5	87.2	90.6	65.3	92.2
C4.5	$*\bullet$ 90.4	$*\bullet$ 74.6	$*\bullet$ 80.9	$*$ 85.0	$*\bullet$ 60.7	$*$ 89.7
ID2-of-3+	$*\bullet$ 91.9	$*\bullet$ 76.2	$*$ 85.3	$*$ 82.7	$*\bullet$ 60.4	90.8
Trepan	$*$ 93.1	83.2	$*\diamond$ 83.9	87.4	$*$ 63.3	91.3

4.2.3 Results

Table 3 shows the test-set accuracy results for the experiments in the classification domains. Each of these values represents the average of the test-set results for the cross-validation run in each domain. It can be seen that, for every data set, neural networks provide better predictive accuracy than the decision trees learned by C4.5 and ID2-of-3+. This result indicates that these are domains for which neural networks have a more suitable inductive bias than either C4.5 or ID2-of-3+. 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 (such as these decision-tree algorithms) that it makes sense to apply a rule-extraction method. I test the statistical significance of differences in accuracy using a paired-sample, two-tailed t-test. In all of the domains, the pairwise differences in accuracy between the neural networks and the C4.5 trees are statistically significant at $p \leq 0.05$. In five of the six domains, the pairwise differences in accuracy between the neural networks and the ID2-of-3+ trees are significant at this level. These cases (for both C4.5 and ID2-of-3+) are indicated in the table by the symbol '$*$' next to the inferior accuracy result.

The results in Table 3 show that the trees extracted by Trepan are more accurate than the C4.5 and ID2-of-3+ trees in five of the domains (`coding`, `heart`, `promoter`, `telephone`, and `voting`). I test the significance of the accuracy differences between Trepan and the other algorithms using a paired-sample, two-tailed t-test. Cases in which the accuracy of another algorithm is less than Trepan at $p \leq 0.05$ are indicated in the table by the symbol '$\bullet$' next to the inferior result. In three out of the five cases mentioned above, the Trepan trees

Table 4: **Test-set fidelity (%) for the classification domains.**

method	`coding`	`heart`	`NETtalk`	`promoter`	`telephone`	`voting`
Trepan	94.1	96.0	91.0	89.1	89.3	96.8

are significantly more accurate than both the C4.5 trees and the ID2-of-3+ trees at $p \leq 0.05$. In the `NETtalk` domain, the trees extracted by Trepan are more accurate than the C4.5 trees, but not as accurate as the ID2-of-3+ trees. In fact, the difference in accuracy between ID2-of-3+ and Trepan in this domain is statistically significant; this result is marked by the symbol '$\diamond$' in the table. As we shall see in Section 4.5, however, this result is largely due to the fact that the `NETtalk` trees were not allowed to be grown large enough.

Table 4 shows the fidelity of the Trepan trees to their respective neural networks. As with the accuracy measurements, these values represent averages for the cross-validation run in each domain. In four of the domains, the fidelity level exceeds 90%. The results in this table indicate that the trees extracted by Trepan provide good approximations to their respective neural networks.

In summary, for a wide range of problem domains in which neural networks provide better predictive accuracy than conventional decision-tree induction algorithms, Trepan is able to extract decision trees that represent their learned hypotheses to a high degree of fidelity. In many cases, because the trees extracted by Trepan provide a faithful representation of the neural networks' hypotheses, they provide superior predictive accuracy to the trees learned by the conventional algorithms.

Tables 5 and 6 show tree-complexity measurements for C4.5, ID2-of-3+, and Trepan. The measurements for the conventional decision-tree algorithms are taken for pruned trees. Broad trends are hard to discern from these results. In two of the domains (`heart` and `voting`), the trees learned by all three algorithms are comparable in size. In two other domains (`NETtalk` and `telephone`), the Trepan trees are comparable in size to the C4.5 trees but significantly smaller than the ID2-of-3+ trees. And finally in the `coding` domain, the Trepan trees are much smaller than those induced by the other two algorithms, while in the `promoter` domain the situation is reversed. Based on these results, I argue that the trees extracted by Trepan are generally as comprehensible as the trees learned by

Table 5: **Tree complexity (# internal nodes) for the classification domains.**

method	coding	heart	NETtalk	promoter	telephone	voting
C4.5	153.3	15.5	87.8	7.0	35.0	11.2
ID2-of-3+	63.3	13.0	137.0	13.2	215.0	7.1
Trepan	9.0	10.7	26.2	11.7	9.5	11.5

Table 6: **Tree complexity (# feature references) for the classification domains.**

method	coding	heart	NETtalk	promoter	telephone	voting
C4.5	153.3	15.5	87.8	7.0	35.0	11.2
ID2-of-3+	344.3	37.2	258.0	41.3	109.9	14.5
Trepan	70.5	24.4	68.8	105.5	26.3	29.7

conventional decision-tree algorithms. With Trepan, however, there is often a trade-off between the fidelity and the comprehensibility of extracted trees. That is, in some cases the comprehensibility of the trees extracted by Trepan can be improved by requiring the algorithm to return smaller trees. Similarly, the fidelity of extracted trees can sometimes be improved by allowing Trepan to consider larger trees. This trade-off is illustrated in the experiments reported in Section 4.5.

Representative trees extracted by Trepan for each problem domain are shown in Appendix A.

4.3 Trepan Applied in a Regression Domain

This section investigates using Trepan to describe a neural network that is trained to perform a regression task. Specifically, the network is trained to predict the Dollar-Mark exchange rate (Weigend et al., 1995). This network was trained by Weigend et al. without any intention of later applying Trepan to it. The network and associated data were shared with me by the authors, who were interested in gaining an understanding of their trained network.

4.3.1 Problem Domain

The data for this time-series problem consists of daily values from the period of January 15, 1985 through January 27, 1994. The last 216 days were set aside as a test set before the neural network was trained. From the remaining data, every fourth day was held aside to form a validation set (535 days), and the rest of the data (1607 days) was used as a training set. Twelve of the 69 features for this domain represent information derived from the time series itself (relative strength index, skewness, point and figure chart indicators, etc.), and the other 57 inputs represent fundamental information beyond the series itself (indicators dependent on exchange rates between different countries, interest rates, stock indices, currency futures, etc.). Forty-eight of the features are real-valued and the remaining 21 are discrete-valued.

The network predicts three real-valued outputs. The first output unit predicts a normalized version of the *return*: the logarithm of the ratio of tomorrow's price to today's price is divided by the standard deviation of this ratio computed over the last 10 trading days. The second output unit predicts the number of days to the next *turning point*: the point at which the daily change in the exchange-rate will reverse direction. The third output unit predicts the return between the current day and the next turning point.

Although the network was trained to predict three separate continuous quantities, TREPAN is applied here to extract a description of only the *return* output. Moreover, although the *return* output of the network is a continuous variable, the extraction task here is framed as one of predicting whether the current day's price is going *up* or *down*. In other words, the *return* output is trained to perform a regression task, but the extraction process is set up as a classification task – to describe the qualitative behavior of the *return* output.

4.3.2 Algorithms

The network used in the experiments has 69 input units, a single layer of 15 hyperbolic-tangent hidden units, and three output units with linear transfer functions. The network was trained using the technique of *clearning* (Weigend et al., 1995). The *clearning* method involves simultaneously *cleaning* the data and *learning* the underlying structure of the data.

The idea behind cleaning is that the learned model can be used to adjust training instances so that their values are more likely, given that the model represents the true structure of the problem. Specifically, *clearning* uses a cost function that consists of two terms (shown here for a single training example):

$$C = \frac{1}{2}\eta\left(y - y^d\right)^2 + \frac{1}{2}\kappa\left(x - x^d\right)^2.$$

The first term, which corresponds to the *learning* aspect of *clearning*, is the squared error between the network's output y and the target value y^d. The second term, which corresponds to the *cleaning* aspect, is the squared deviation between the cleaned input x and the actual input x^d. The parameters η and κ are the learning rate and the cleaning rate respectively. In addition to *clearning*, a pruning process is also applied to the network during training. *Clearning* is run until the network starts over-fitting the training data (as indicated by accuracy on the validation set), and then some of the weights from the input-to-hidden layer are pruned. This process is iterated until overpruning is detected. At this point, the last set of pruned weights is restored and the *clearning* step is run a final time. Out of the original 69 features, the *clearned* network used in these experiments retained connections to 15 of the real-valued ones and 5 of the discrete-valued features.

As in the classification experiments earlier in this chapter, C4.5 and ID2-of-3+ serve as baselines for comparison in the experiments presented in this section. The induction problem for these algorithms is to learn the classification task of predicting whether the exchange rate will go up or down on the next day. I run C4.5 and ID2-of-3+ using two different feature sets. In the first run, both algorithms are given the entire set of 69 features. In the second run, they are given only the 20 features that were incorporated into the hypothesis of the *clearned* network. The latter configuration allows us to test the hypothesis that any performance differences between the trees extracted by TREPAN and the trees induced by C4.5 and ID2-of-3+ are explained by the fact that TREPAN is effectively working with a reduced feature set (the one selected by the *clearned* network).

Pruning confidence levels ranging from 5% to 95% are applied to the trees produced by C4.5 and ID2-of-3+, and accuracy measurements on the validation set are used to determine which tree to return for each algorithm. The other parameters of these algorithms are set as in the previous experiments. In addition to C4.5 and ID2-of-3+, a naive prediction algorithm is also considered as a baseline. This algorithm simply predicts that the exchange rate will do the same thing it did yesterday (i.e., if it went up yesterday then predict that it will go up today).

Unlike the classification experiments presented earlier in this chapter, six runs of Trepan are made varying two parameters: the beam-width parameter for the m-of-n search, and the *min_sample* parameter which specifies how many instances Trepan considers at a node before giving a class label to the node or choosing a splitting test for it. For the beam-width parameter, values of 2, 5, and 10 are tried. For the *min_sample* parameter, values of 1,000, 5,000, and 10,000 are tried. For each configuration of parameters, Trepan grows a tree until it has 31 internal nodes (recall that this is the size of a complete binary tree of depth five). After each node is added to the tree, the validation set is used to measure the fidelity of the tree to the network. From this sequence of nested trees for a given run, Trepan returns the one that has the highest validation-set fidelity. Finally, from the pool of trees extracted with different parameter settings, the one with the best validation-set fidelity is retained.

4.3.3 Results

Using validation-set accuracy, the pruning levels selected for the various runs of C4.5 and ID2-of-3+ were as follows: 15% for C4.5, 5% for C4.5 with the reduced feature set, 5% for ID2-of-3+, and 25% for ID2-of-3+ with the reduced feature set.

Recall, that although the network was trained for a regression task (predicting the continuous *return* value), we are interested in characterizing its behavior in terms of a classification task (predicting whether the exchange rate will go up or down).

Table 7 shows test-set accuracy results for the neural network, the Trepan tree extracted from the network, the decision trees induced by the two conventional algorithms, and the naive prediction strategy. It can be seen that the *clearned* network provides the most accurate predictions and the naive rule (predicting same as previous day) and C4.5 produce the worst predictions. Although the ID2-of-3+ trees are more accurate than the C4.5 trees, they are not as accurate as the tree extracted by Trepan, which makes predictions that are nearly as accurate as the *clearned* network. These results suggest that, by exploiting the concept

Table 7: **Test-set accuracy (%) for the `exchange-rate` domain.**

method	accuracy
naive rule	52.8
C4.5	52.8
C4.5 (reduced feature set)	54.6
ID2-of-3+	59.3
ID2-of-3+ (reduced feature set)	57.4
TREPAN	60.6
clearned network	61.6

representation learned by the neural network, TREPAN is able to find a better decision tree than either C4.5 and ID2-of-3+. I test the statistical significance of these accuracy differences using the sign test known as the McNemar χ^2 test (Sachs, 1984). Because the test set for this problem is small, none of the differences are significant at $p \leq 0.05$. I note, however, that TREPAN is more accurate than the naive algorithm at $p = 0.10$, than C4.5 at $p = 0.09$, and than C4.5 with the reduced feature set at $p = 0.11$. The *clearned* network is more accurate than the naive algorithm at $p = 0.07$, than C4.5 at $p = 0.07$, and than C4.5 with the reduced feature set at $p = 0.09$.

Figure 17 shows the test-set fidelity and accuracy of the intermediate trees produced by TREPAN as it add nodes to the tree being extracted. It can be seen in this figure that the accuracy of the intermediate trees is fairly constant after the first node is added to the tree. A disappointing result is that the fidelity of the tree is nearly constant as well. The first node in the tree quickly attains fidelity near 80%, but successive nodes added to the tree do not significantly improve this value. This result seems to suggest that TREPAN is failing to find a good decision-tree representation of the network. However, it is interesting to consider the profit-and-loss curves produced by the *clearned* network and the TREPAN tree extracted from it.

Figure 18 shows the profit-and-loss curves that result from trading based on the predictions made by the network and the TREPAN tree extracted from it.[4] Also shown as a

[4]Following Weigend (1995), these curves are calculated by assuming that the *full position* is taken each day, meaning that the trader's daily gain/loss is equal to the percentage change in the exchange rate. Additionally, a small transaction cost (0.001) is charged whenever the trader changes its position, meaning that the direction of today's prediction is the opposite of yesterday's prediction.

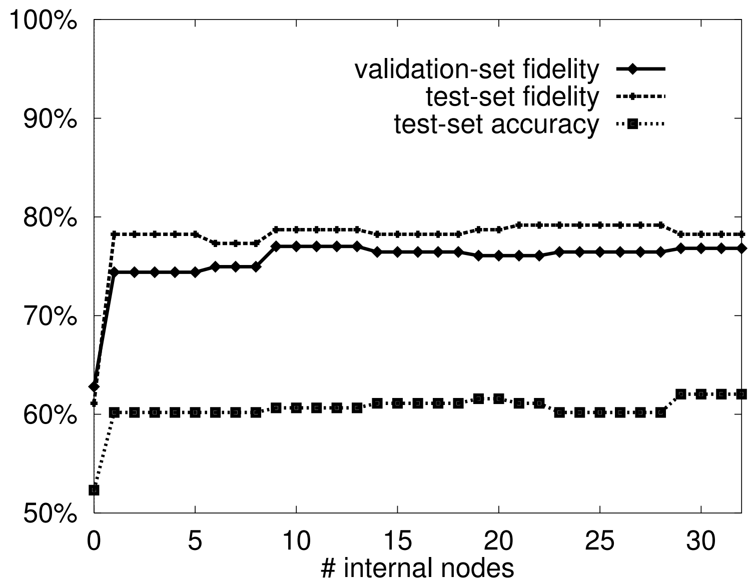


Figure 17: TREPAN **fidelity and accuracy for the** `exchange-rate` **domain.** The x-axis indicates the number of internal nodes the extracted tree, and the y-axis indicates test-set fidelity and accuracy.

reference point is the profit-and-loss curve for trading based on the naive strategy. This figure shows that, although the fidelity of the TREPAN tree (when measured as the percentage of discrete predictions in agreement) is not very high, the tree has captured much of the underlying structure of the solution represented by the network. The profit-and-loss curve for TREPAN closely mirrors that of the neural network. An explanation for the seeming dissonance between this result and the result shown in Figure 17 is that because the predictions made by the network are clustered around zero, the regression surface represented by the network has many crossings of the plane that divides *up* predictions from *down* predictions. Although TREPAN has a hard time fitting a decision tree to all of these fluctuations in the decision surface, it does a good job of fitting the overall structure of the surface. That is, it more accurately models the regions of the surface that correspond to *way up* or *way down* than it does those regions that represent small values of *up* and *down*.

Figure 19 shows the profit-and-loss curves that result from trading based on the C4.5 and ID2-of-3+ trees. None of these trees performs as well as the network or TREPAN. The best tree in this group is the one induced by ID2-of-3+ using the full feature set. Its total profit is about ten percentage points less than the profit earned by the neural network and the TREPAN tree extracted from it. Moreover, it obviously does not model the gross behavior

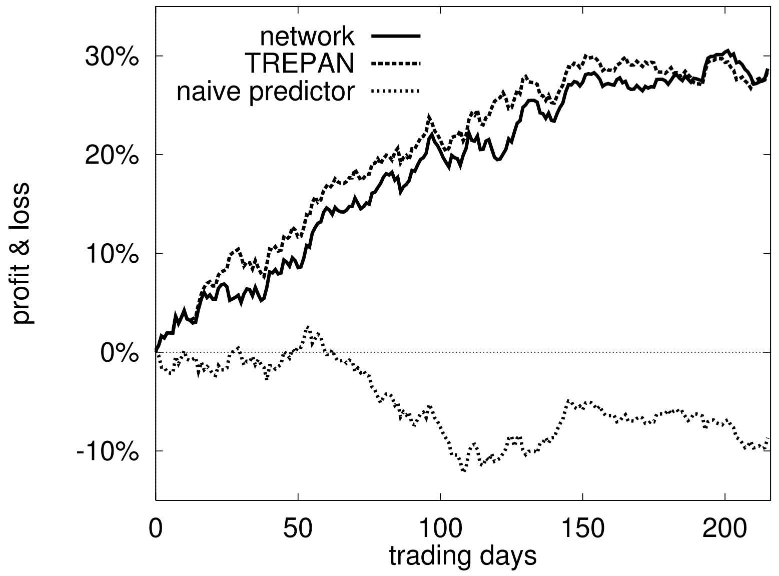


Figure 18: **Profit-and-loss curves for the `exchange-rate` domain.** The x-axis indicates the trading days in the test set, and the y-axis indicates the return on investment that would result from trading based on the predictions made by each method.

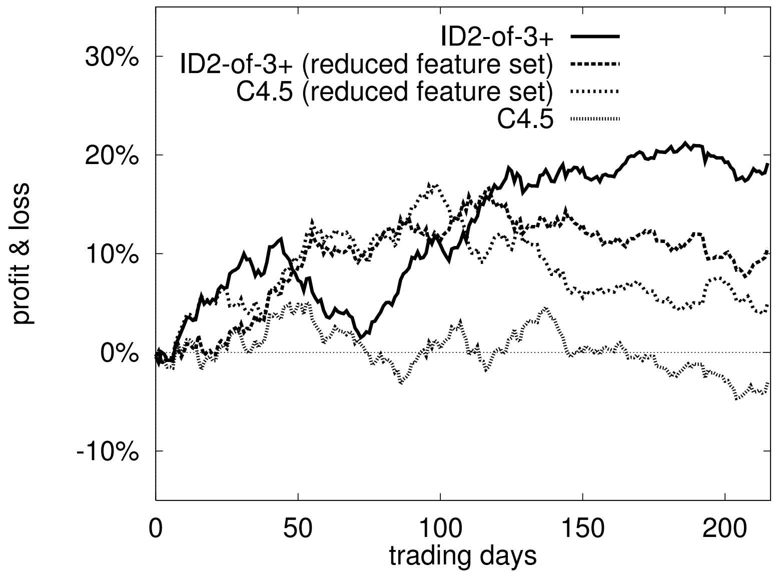


Figure 19: **Profit-and-loss curves for the `exchange-rate` domain.** The x-axis indicates the trading days in the test set, and the y-axis indicates the return on investment that would result from trading based on the predictions made by each method.

Table 8: **Tree complexity for the exchange-rate domain.** The middle column indicates the number of internal nodes in the tree, and the rightmost column indicates the total number of feature references in the splitting tests used in the tree.

method	# internal nodes	# feature references
C4.5	103	103
C4.5 (selected)	53	53
ID2-of-3+	78	303
ID2-of-3+ (selected)	103	358
TREPAN	5	14

of the network nearly as closely as the TREPAN tree.

Table 8 shows tree-complexity measurements for the C4.5, ID2-of-3+, and TREPAN trees. For both complexity measures, the TREPAN trees are considerably simpler than the C4.5 and ID2-of-3+ trees, even though the latter trees were pruned. Based on these results, I argue that the tree extracted by TREPAN is more comprehensible than the trees learned by the conventional decision-tree algorithms. The tree produced by TREPAN is depicted in Appendix A.

4.4 TREPAN Applied in a Reinforcement-Learning Domain

The experiments in this section investigate the application of TREPAN in a reinforcement-learning setting. Recall from Chapter 1 that in reinforcement learning, the task is to learn what action an agent should take for any given observed state such that the agent maximizes some long-run measure of reward. The reinforcement-learning agent that is used in these experiments is one that controls elevator cars in a simulated, but realistic, environment. This learning agent is especially interesting because it has demonstrated performance that surpasses the best heuristic, elevator-control algorithms.

4.4.1 Problem Domain

The reinforcement-learning system that is investigated here is a neural network that addresses the real-world problem of elevator dispatching (Crites & Barto, 1996). As with the exchange-rate network investigated in the previous section, this learning system was developed without any intention of later applying a rule-extraction method to it.

The elevator-dispatching agent operates in a simulated 10-story building with four elevator cars (Lewis, 1991; Bao et al., 1994). The state of the system that is presented to the neural network includes information about:

- which hall buttons have been pushed (indicating waiting passengers) and the elapsed time since each button was pushed,
- the location and direction of the elevator car being controlled,
- the locations and speeds of the other cars in the system,
- whether or not the car is at the highest floor with a waiting passenger,
- whether or not the car is at the floor with the passenger who has been waiting for the longest amount of time.

This information is encoded using 19 real-valued and 12 discrete-valued features. Parts of the system state are not observable by the reinforcement learner. For example, the learner does not know the number of passengers waiting at each floor nor their desired destinations. The elevator car has a default control policy which specifies constraints such as do not stop at a floor unless a passenger wants to get off there, and given a choice between moving up or down, prefer to move up. The task for the reinforcement learner is to decide, given the current measurable state, whether a car should stop or continue on its current path.

Crites and Barto trained their elevator-dispatching network for 60,000 hours of simulated elevator time. To evaluate the trained network's performance, they used it to control elevator cars under three different traffic profiles. For each profile, they measured the network's performance during 30 hours of simulated time. The first traffic profile, which was used for training the network, involves only downward traffic in the building. The second profile involves both up and down traffic, and the third profile has twice as much up traffic as the second one. Recall that in reinforcement learning, unlike supervised learning, there is not a fixed set of training data. The learner is able to partially measure the state of the world, thus garnering input vectors, but because its actions affect the world, its control policy will influence which input vectors it encounters as it operates. Recall also, that unlike the supervised learning setting, each input vector, $\vec{x}$, is not associated with a target output value, y. But instead, the learner periodically receives a scalar reinforcement signal, and its goal is to learn a control policy that maximizes the long-term reward.

4.4.2 Algorithms

The reinforcement learner for the elevator control task is a neural network with 47 input units, 20 hidden units with logistic transfer functions, and two output units with linear transfer functions. The network is trained using a method called *Q-learning* (Watkins, 1989). In *Q*-learning, the learner estimates an evaluation function, $Q(s, a)$, that specifies the discounted, cumulative reinforcement that can be achieved starting from state s, applying a as the first action, and assuming that optimal actions are followed thereafter. The optimal action is specified by a policy, π^*, defined as follows:

$$\pi^*(s) = \arg\max_a Q(s, a).$$

Through a process of iterative approximation, the learner refines its estimate of the Q function.

In the context of the elevator-control problem, the discounted cumulative reinforcement is defined as:

$$-\int_0^\infty e^{-\beta\tau} r_\tau d\tau,$$

where r_τ is the instantaneous cost at time τ, and β is a parameter that specifies the rate of exponential decay. The instantaneous cost, r_τ, is defined to be the sum of squared wait times for the passengers in the building.

Although Q-learning involves training the network in the context of a regression problem (predicting continuous Q values), once the network is trained, it is used to predict discrete values – which action to take in a given state. Thus, the extraction task for TREPAN is a supervised learning problem: describe the conditions under which the trained network will select each action. A training set for Trepan is acquired by running the elevator-control network in its environment for 30 hours of simulated time, saving a set of 16,623 state-action pairs. This run is performed employing the traffic profile that was used for training the network – down traffic only. Three other sets of state-action pairs are collected by running the network for 30 simulated hours under the three different traffic profiles used by Crites and Barto to evaluate the performance of the network. I use these sets as test sets to get unbiased estimates of the fidelity of the trees extracted by Trepan under various distributions of data.

I apply Trepan to the elevator-control network using the same parameter settings as in Section 4.2. I set aside 10% of the training data as a validation set, and then allow Trepan to grow a tree of up to 31 internal nodes. From this sequence of nested trees, Trepan returns the one that has the highest level of fidelity, measured using the validation set.

4.4.3 Results

Table 9 shows the test-set fidelity measurements for the tree extracted by Trepan. Each line in the table represents the fidelity of the tree to the network when measured under different traffic conditions. Recall that the training set given to Trepan consisted of data only from the first distribution – down traffic only. The results in this table indicate that the action selected by the tree agrees with the network in approximately 90% of encountered states, and furthermore this level of fidelity is fairly consistent across states that come from different types of traffic conditions.

Table 10 summarizes the syntactic complexity of the tree extracted by Trepan. The tree, which is shown in Appendix A, consists of only four internal nodes and a total of 14 feature references.

Figures 20 and 21 plot the network's predicted Q values for a random sample of test-set instances from the downward traffic profile. The x-axes in these figures represent predicted Q values for the *stop* action, and the y-axes represent predicted Q values for the *continue* action. Whereas Figure 20 shows Q values for states in which the network and the Trepan tree agree in their selected actions, Figure 21 represents states in which the network and the extracted tree disagree. It is interesting to note that the cases of disagreement are tightly

Table 9: TREPAN **test-set fidelity (%) for the** `elevator-control` **domain.**

traffic profile	fidelity
down traffic only	89.4
down and up traffic	90.8
down and $2\times$ up traffic	91.9

Table 10: TREPAN **tree complexity for the** `elevator-control` **domain.**

method	# internal nodes	# feature references
TREPAN	4	14

clustered around the line $Q(continue) = Q(stop)$. That is, in most of the states in which TREPAN fails to accurately model the network, the expected utility of the two actions is about the same. By comparing Figures 20 and 21, it can be seen that TREPAN nearly always agrees with the network in cases where one action is predicted to be clearly superior to the other.

As in the case of the `exchange-rate` task, which also involves predicting continuous values, this result suggests that the tree extracted by TREPAN accurately represents the gross structure of the network's solution. The fidelity of TREPAN's solution is lowest in those parts of the instance where the network does not make "strong" predictions. In problem domains such as this, it might make sense to have TREPAN describe the target network using three (or perhaps more) classes (e.g., *stop, continue*, or *either*). By framing the extraction task as a multi-class problem, the extracted trees would explicitly represent cases that do not correspond to strong predictions.

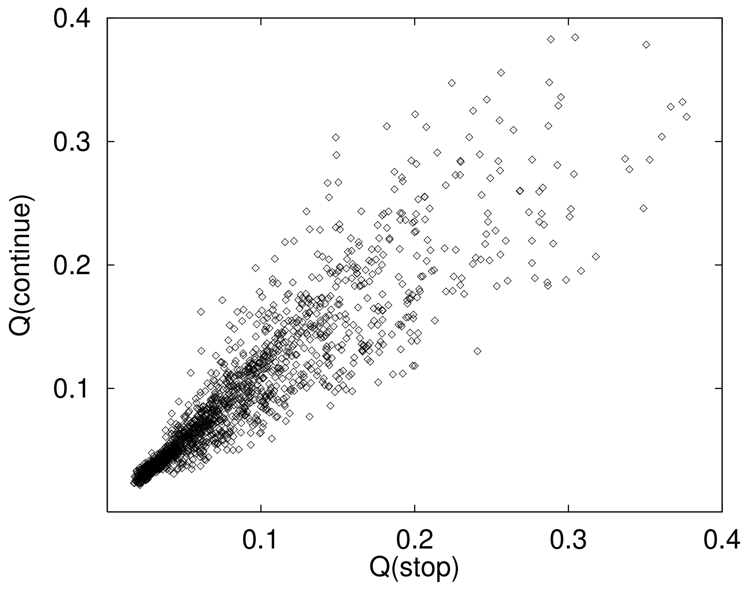


Figure 20: Q **values for states in which the** TREPAN **tree agrees with the network.** The x-axis represents predicted Q values for the *stop* action, and the y-axis represents predicted Q values for the *continue* action.

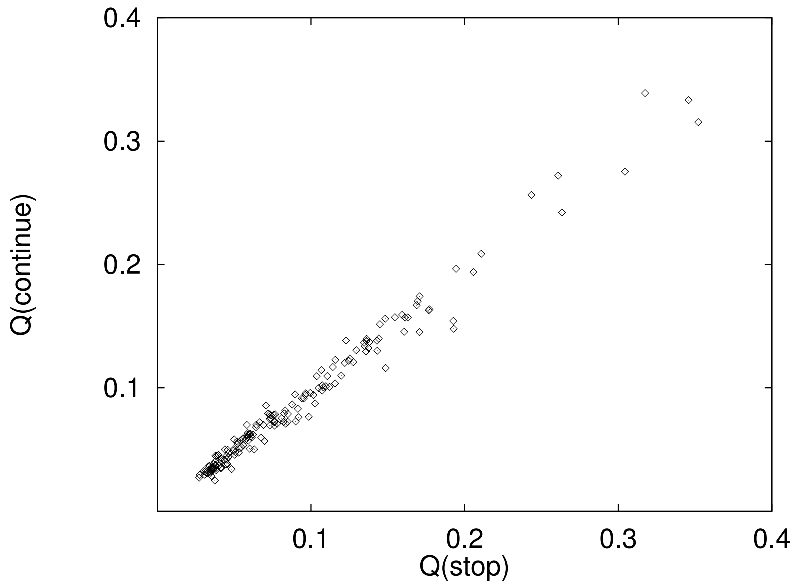


Figure 21: Q **values for states in which the** TREPAN **tree disagrees with the network.** The x-axis represents predicted Q values for the *stop* action, and the y-axis represents predicted Q values for the *continue* action.

4.5 Evaluating the Components of Trepan

This section presents experiments that are designed to elucidate the value of various components of the Trepan algorithm. In particular, I investigate the value of (i) m-of-n splitting tests in Trepan, (ii) Trepan's instance-modeling method, and (iii) Trepan's best-first tree expansion strategy. The methodology used here is to conduct *lesion experiments* (Kibler & Langley, 1988), in which some component of Trepan is left out or modified, and the resulting performance is compared to that of the unmodified algorithm.

4.5.1 The Value of M-of-N Splitting Tests

The first experiment investigates the utility of m-of-n tests in trees grown by Trepan. Using the six classification domains presented in Section 4.2, this experiment compares Trepan to a modified version of Trepan that uses only single-feature tests at its internal nodes. The latter version of the algorithm is identical to the first in every respect, except for its avoidance of m-of-n splitting tests. The only stopping criterion employed in this experiment is a limit on the maximum tree size. Trees are grown up to 31 internal nodes, and as each node is added, the test-set fidelity of the current tree is measured. As in the experiments presented in Section 4.2, all of the results here represent averaged values over a cross-validation run in each domain.

Figure 22 shows the *fidelity curves* that result from this experiment. Each curve plots test-set fidelity as a function of the number of internal nodes in a tree. There are two interesting conclusions to be drawn from these plots. The first is that, in most domains, Trepan trees with m-of-n tests are able to attain a higher level of fidelity than trees without m-of-n tests for a fixed number of internal nodes. It is important to note, however, that since the splitting tests of the m-of-n trees may be complex expressions, that the x-axes of the plots do not control for overall tree complexity. In some cases though, an m-of-n expression or a small set of m-of-n expressions may be able to concisely represent the same function as a very large tree that uses single-feature tests. This effect is especially pronounced in the `promoter` domain where Trepan trees with only a single node attain a higher level of fidelity than is seen for much larger trees extracted by the variant of Trepan that uses only single-feature

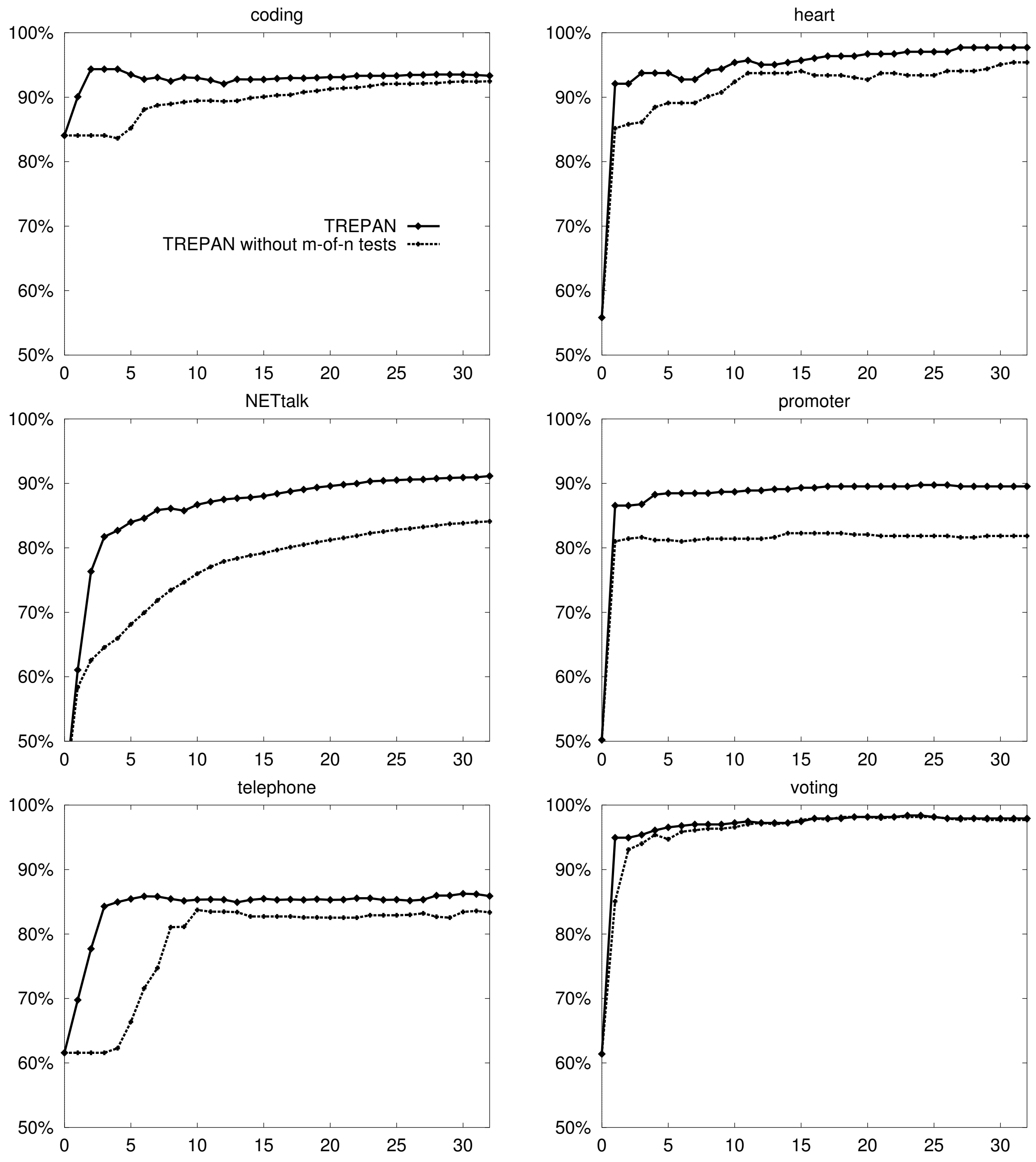


Figure 22: **Fidelity curves for** Trepan **and** Trepan **without** m**-of-**n **splitting tests.** The x-axis in each figure indicates the number of internal nodes in extracted trees, and the y-axis indicates the test-set fidelity of the extracted trees to the trained networks.

tests. This effect can also be seen in the curves for the `coding` and `NETtalk` domains.

These plots also illustrate the fidelity-complexity trade-off that is often faced when applying Trepan. High fidelity often comes at the price of large, and presumably hard-to-understand, trees. The trade-off is best illustrated by the experiments in the `NETtalk` domain, where the fidelity of the trees extracted by Trepan is clearly proportional to their size. When applying Trepan in practice, the user must decide what size of extracted tree provides an acceptable compromise between desired fidelity and comprehensibility. One of the primary advantages of Trepan's strategy of growing trees in a best-first manner is that it gives the user a fine level of control over the size of the tree that is returned.

4.5.2 The Value of an Instance Model

In this section, I evaluate the utility of the instance models that Trepan constructs and uses to draw instances for membership queries. As in the previous experiment, fidelity curves are plotted to show how the quality of the trees produced by various instantiations of Trepan change as a function of tree size. As before, the results reported for each domain represent averages across a cross-validation run. In this section's experiments, Trepan is contrasted to three modified versions of the algorithm.

In the first variant, Trepan uses global instance models instead of local ones. Recall that a local instance model is one that is constructed at a particular node in a tree using only the data that reaches that node. The purported advantage of local models is they can potentially provide better estimates of the underlying distribution of data by capturing some of the conditional dependencies that exist among features. Thus, the purpose of considering this variant to see if local instance models enable Trepan to extract higher-fidelity trees by more accurately representing the underlying distribution of data.

The second variant of Trepan considered in this experiment does not use an instance model at all, but instead draws instances assuming a uniform distribution over the instance space. The purpose of including this variant in the experiment is to test the hypothesis that the use of instance models enables Trepan to extract higher-fidelity trees.

The third variant of Trepan used in this experiment does not use query instances at all; it uses only the training data to induce a tree. Recall that the term *query instances* refers to randomly drawn instances used for membership queries. Although this variant does make membership queries to get the class labels for examples in the training set, it does not make any additional membership queries. This variant is included in the experiment to test the hypothesis that TREPAN's ability to make queries for large numbers of instances leads to higher-fidelity trees.

Figure 23 shows the fidelity curves for TREPAN and the first two variants considered in this experiment (TREPAN without local models and TREPAN with uniform sampling). Somewhat surprisingly, in all of the problem domains, the fidelity curves for TREPAN and the version of TREPAN without local instance models are nearly indistinguishable. This result suggests that, in the domains considered here, either the local instance models are not providing better estimates of the data distributions, or that better estimates are not really of value in extracting decision trees. Note, however, that the use of local instance models carries little cost. The computational expense of constructing the models is small, and in none of the domains considered here did local models lead to worse results than global models.

The fidelity-curve results for the other TREPAN variant, however, illustrate the value of using *some* model of the underlying data distribution. In three of the domains (`heart`, `NETtalk`, and `telephone`), the fidelity curves are considerably worse for the version of TREPAN that samples the instance space uniformly. The degradation is especially dramatic in the case of the `telephone` domain. Without an instance model in this domain, TREPAN fails to focus its effort in the parts of the instance space where data is likely to occur, but instead refines the tree in sparsely populated regions of the instance space.

Figure 24 shows the fidelity curves for TREPAN and the variant of TREPAN that does not use query instances. The curves for the variant do not extend through 32 nodes in the `heart` and the `promoter` domains because smaller trees are able to completely cover the training sets in these domains. In three of the problem domains (`heart`, `NETtalk`, and `promoter`) TREPAN without query instances clearly performs worse than TREPAN. When query instances are not used in the `heart` and `promoter` domains, the fidelity of the extracted trees does not steadily improve with increasing tree size, but often gets worse. In the `NETtalk` domain, the fidelity of the extracted trees for this variant does not decline with increasing

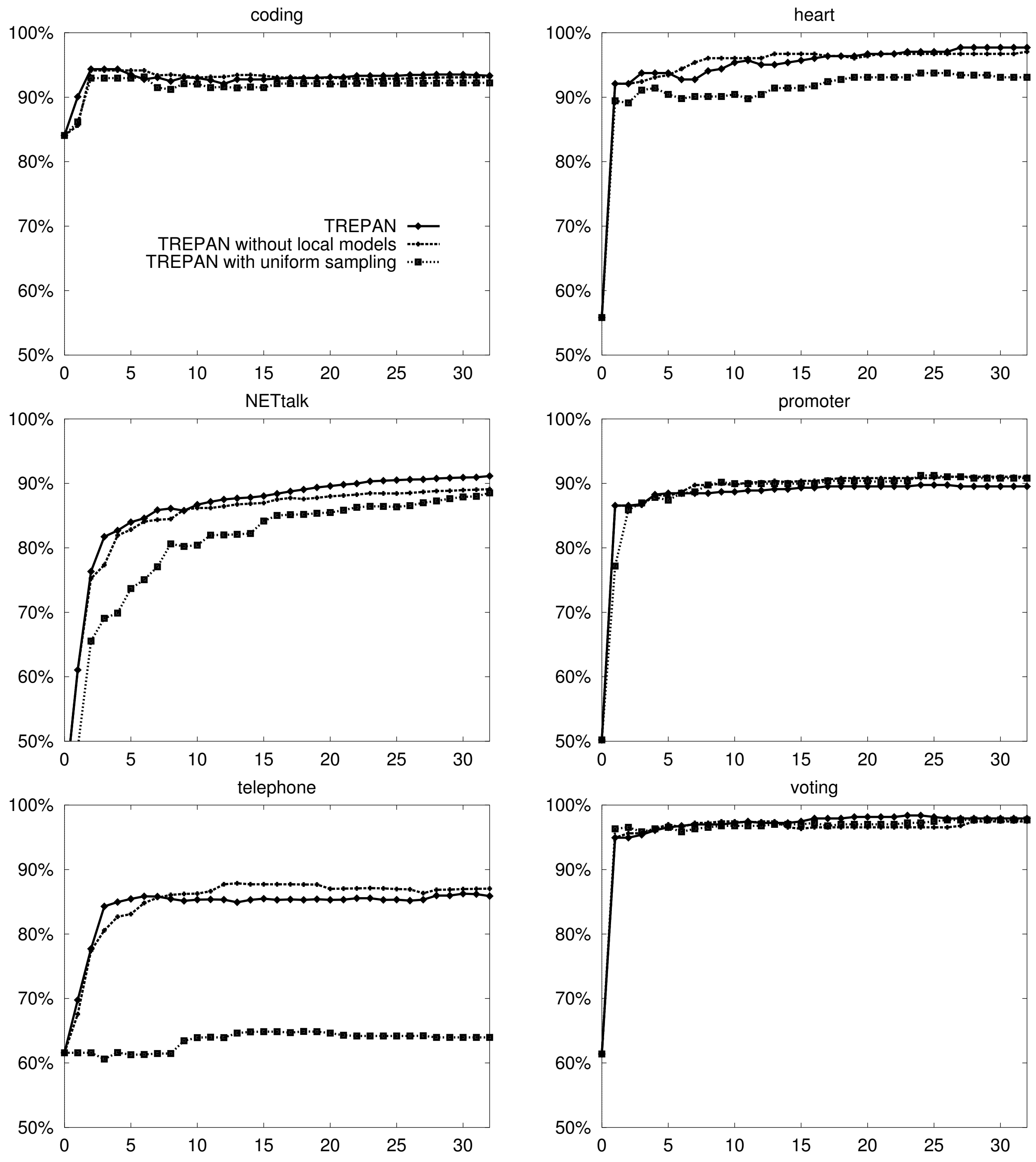


Figure 23: **Fidelity curves for Trepan and Trepan with simpler instance models.** The x-axis in each figure indicates the number of internal nodes in extracted trees, and the y-axis indicates the test-set fidelity of the extracted trees to the trained networks.

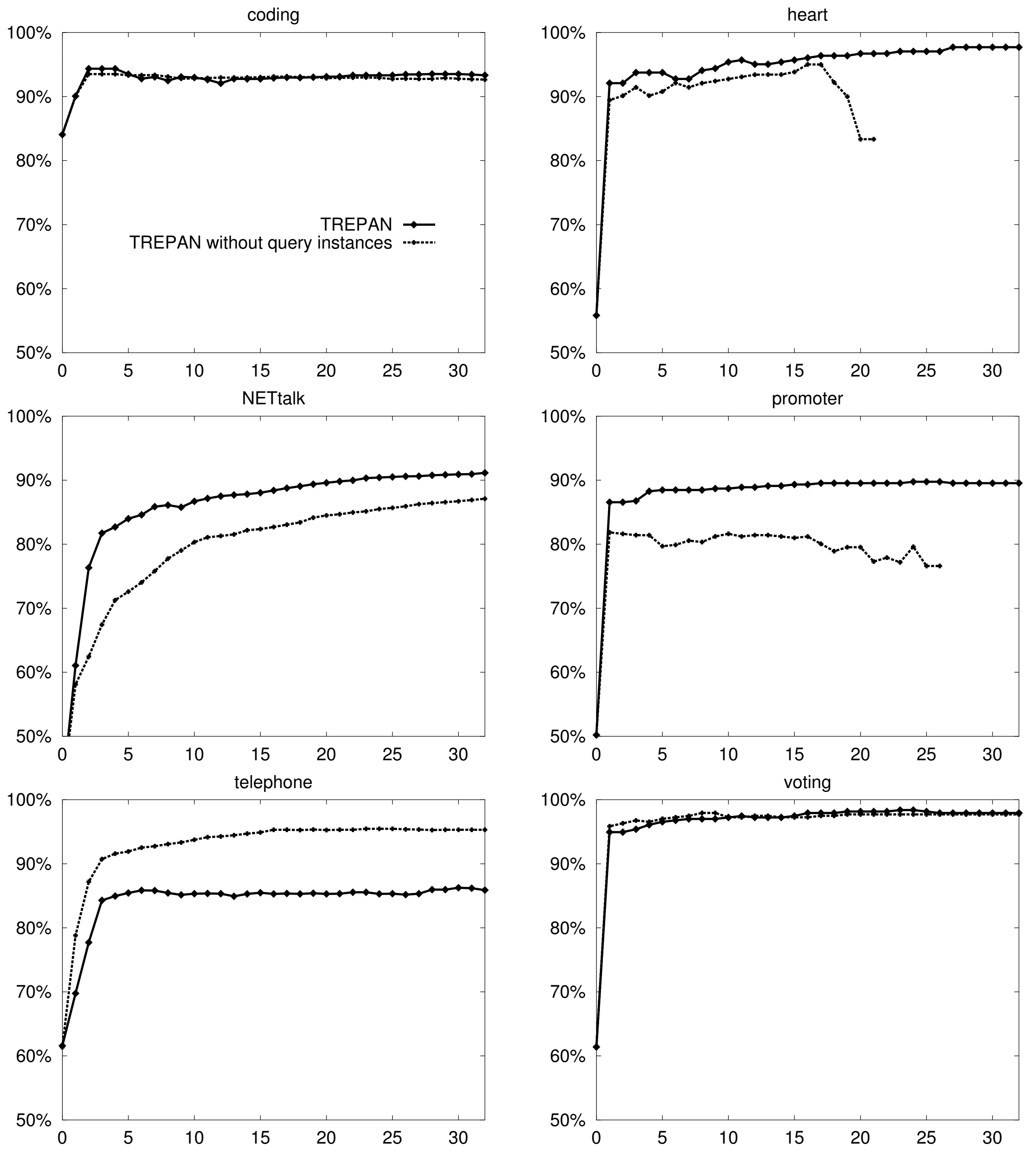


Figure 24: **Fidelity curves for Trepan and Trepan without using query instances.** The x-axis in each figure indicates the number of internal nodes in extracted trees, and the y-axis indicates the test-set fidelity of the extracted trees to the trained networks.

tree size, but the trees have consistently lower fidelity than their counterparts extracted by the non-lesioned version of TREPAN. In the `coding` and `voting` domains, there is little difference between the trees extracted by TREPAN and the variant. This result in the `coding` domain is most likely explained by the fact that the training sets for this task are very large, and thus query instances are of less value than in most domains.

The most surprising result occurs in the `telephone` domain. Here the lesioned variant of TREPAN actually performs considerably better than TREPAN. This result can be explained by the fact that there are numerous known dependencies among the features in this domain. The simple instance model used by TREPAN fails to adequately represent these dependencies, thereby generating misleading samples of query instances.

From the experiments presented in this section we see that, for a learning-based approach to rule extraction such as TREPAN, it is often of significant value to use query instances, as well as the training data, to induce descriptions of trained networks. Moreover, it is often of significant value to employ models of the underlying data distribution for the purpose of generating these query instances.

4.5.3 The Value of Best-First Tree Growing

The final experiment in this section investigates the effect of TREPAN's best-first tree growing method on the fidelity of returned trees. In this experiment, TREPAN is compared to a variant that grows trees one level at a time. In other words, the first intermediate tree produced by this variant consists of a single node, the second intermediate tree is a tree with three nodes (a binary tree two levels deep), the third intermediate tree has seven nodes, etc. Whereas TREPAN produces a sequence of intermediate trees in one-internal-node increments, the variant produces a sequence of intermediate trees in one-level increments.

Figure 25 shows fidelity curves in the six classification domains for TREPAN and the level-at-a-time variant. It can be seen that the two curves are close to each other in every domain. This result indicates that, in the domains considered here, there is no significant advantage to being able to return an unbalanced tree (i.e., one that has most of its structure distributed to one side of the tree). For these six domains, the primary difference between the performance of TREPAN and the level-at-a-time variant is that TREPAN offers a finer

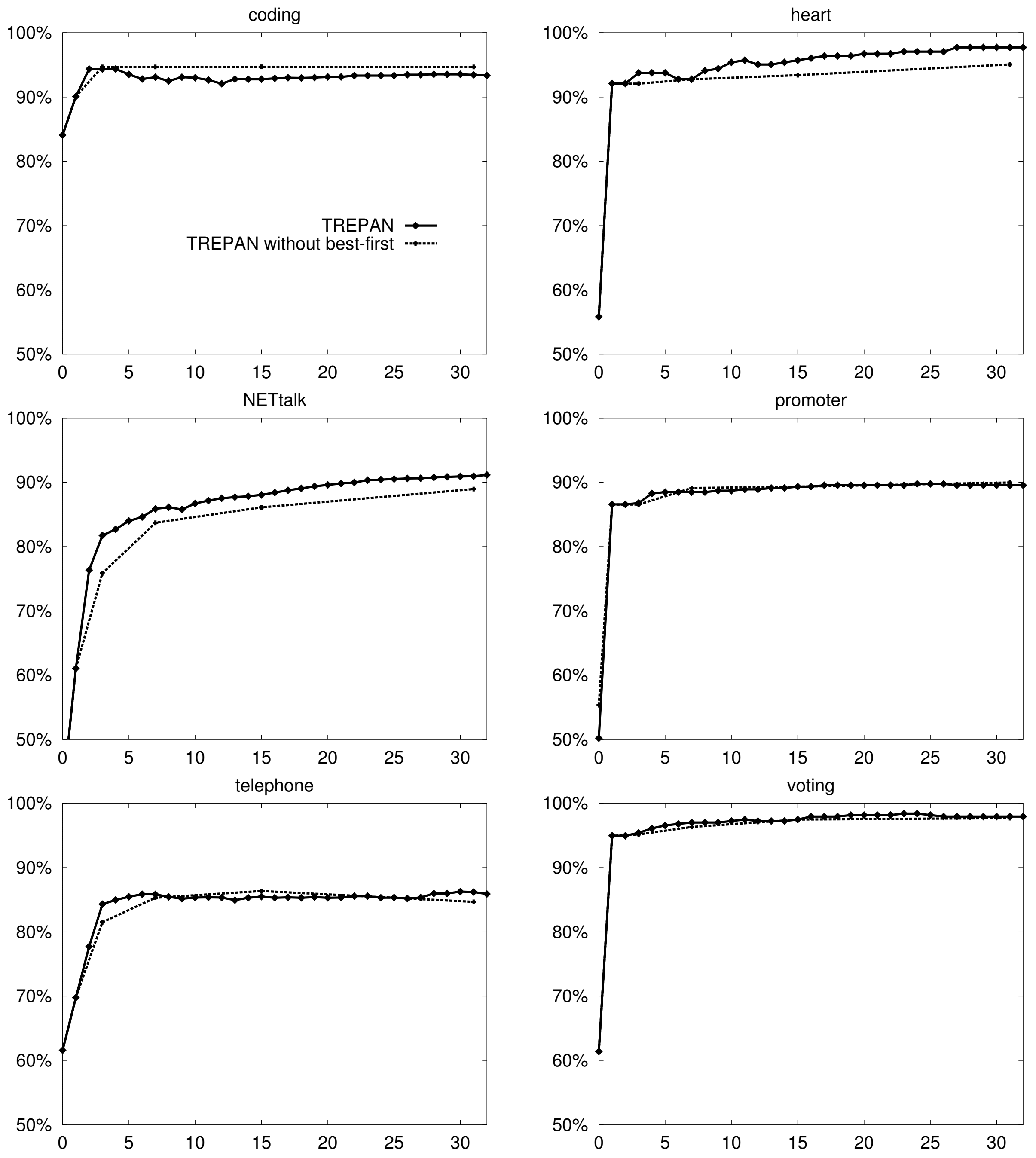


Figure 25: **Fidelity curves for Trepan and Trepan without best-first tree expansion.** The x-axis in each figure indicates the number of internal nodes in extracted trees, and the y-axis indicates the test-set fidelity of the extracted trees to the trained networks.

degree of control over the size of the tree returned. With best-first TREPAN, the user can select a tree of arbitrary size (in terms of the number of internal nodes) to describe a given network.

4.6 Evaluating the Enhancements to ID2-of-3+

One of the minor contributions of this thesis is an improved version of Murphy and Pazzani's ID2-of-3 algorithm. As detailed in Section 4.2.2, the version of ID2-of-3 I developed and used in the experiments in this chapter, ID2-of-3+, incorporates five enhancements not included in the original algorithm: the application of a χ^2 test, literal pruning, and a beam search when constructing m-of-n tests; the use of C4.5's pruning method after tree induction; and the generalization of m-of-n tests to real-valued features. In this section, I present a set of lesion experiments designed to assess the value of the first four of these modifications.

These experiments involve running four variants of ID2-of-3+ in the six classification domains used in Section 4.2. Each of these variants is identical to ID2-of-3+ except that it leaves out one of the four enhancements. The methodology used for these experiments is the same as that used in Section 4.2. The key parameter of the algorithm that is varied is the pruning confidence level (except in the variant that leaves out pruning). As before, cross validation within the training set is used to select the pruning level for that training set.

Table 11 reports test-set accuracy results for ID2-of-3+ and its four lesioned variants. Tables 12 and 13 show tree complexity results for these trees in terms of numbers of internal nodes and numbers of feature references, respectively. The top row in Tables 12 and 13 display the complexity results for ID2-of-3+, and the other rows show the changes from the ID2-of-3+ numbers for each variant. I test the statistical significance of accuracy differences using a two-tailed, paired-sample, t-test. The symbol '$*$' is used in the table to mark the inferior result in cases where ID2-of-3+ is more accurate than a lesioned variant, and the difference is statistically significant at $p \leq 0.05$. The symbol '$\diamond$' marks the ID2-of-3+ result for the only case in which a lesioned variant (the one with the beam width set to one) more accurate, and the difference is statistically significant at this same level.

It can be seen from these results that the incorporation of C4.5's pruning mechanism into

Table 11: **Tree accuracy (%) for ID2-of-3+ experiments.** The symbol '$*$' marks the inferior results in cases where ID2-of-3+ is more accurate than a lesioned variant at the $p \leq 0.05$ level of significance. The symbol '$\diamond$' indicates the inferior result in the case where a lesioned variant (w/ beam=1) is more accurate than ID2-of-3+ at this same level.

method	`coding`	`heart`	`NETtalk`	`promoter`	`telephone`	`voting`
ID2-of-3+	91.9	76.2	85.3	82.7	$\diamond$ 60.4	90.8
w/o pruning	91.6	75.6	$*$ 83.7	82.5	$*$ 58.9	91.0
w/o χ^2	91.7	73.3	$*$ 83.9	84.8	60.9	$*$ 88.5
w/o literal pruning	92.0	75.2	85.6	82.5	60.6	90.3
w/ beam=1	91.7	74.3	85.1	82.5	62.4	89.2

Table 12: **Tree complexity (# internal nodes) for ID2-of-3+ experiments.** The top row shows the number of internal nodes for ID2-of-3+, and subsequent rows report the change from this baseline.

method	`coding`	`heart`	`NETtalk`	`promoter`	`telephone`	`voting`
ID2-of-3+	63.3	13.0	137.0	13.2	215.0	7.1
w/o pruning	+19.2	+10.0	+116.8	+3.1	+152.6	+12.7
w/o χ^2	+0.2	-4.0	-71.0	-5.9	-71.8	+2.4
w/o literal pruning	-2.0	+0.7	+7.1	-0.4	-2.6	-0.3
w/ beam=1	+12.7	+2.8	+33.9	+2.9	-40.9	+1.1

Table 13: **Tree complexity for (# feature references) ID2-of-3+ experiments.** The top row shows the number of feature references for ID2-of-3+, and subsequent rows report the change from this baseline.

method	`coding`	`heart`	`NETtalk`	`promoter`	`telephone`	`voting`
ID2-of-3+	344.3	37.2	258.0	41.3	475.5	14.5
w/o pruning	+55.5	+12.5	+147.4	+3.2	+199.6	+16.6
w/o χ^2	+149.2	+19.1	+372.1	+1.2	+261.5	+32.4
w/o literal pruning	-6.3	+0.7	+197.6	-0.7	+11.7	+0.2
w/ beam=1	-61.3	-3.4	-37.5	-7.7	-128.0	-0.9

ID2-of-3+ is clearly beneficial. In five of the six domains, the ID2-of-3+ variant without pruning is less accurate, and in two of these cases the accuracy difference is statistically significant. Moreover, as expected, the unpruned trees are considerably more complex than the pruned trees in all six domains.

Similarly, the use of a χ^2 test to limit the complexity of m-of-n tests appears to be of significant benefit. The trees induced by the ID2-of-3+ variant that leaves out this mechanism sometimes have fewer nodes than the ID2-of-3+ trees, but the m-of-n tests at the nodes of this variant are considerably more complex in all six domains. In four of the six domains, these complex m-of-n tests lead to lower levels of accuracy, and in two of these domains the differences are statistically significant.

The benefit of using a beam search in ID2-of-3+ as opposed to a hill-climbing search is less clear. In five of the six domains, the variant of ID2-of-3+ that had its beam width set to one (i.e., a hill-climbing variant) produced less accurate trees than ID2-of-3+, but in none of these cases was the accuracy difference statistically significant. In the `telephone` domain, on the other hand, the hill-climbing variant had significantly higher accuracy than ID2-of-3+. In terms of tree complexity, the trees induced by the hill-climbing variant tend to have more nodes than the ID2-of-3+ trees, but the m-of-n tests at these nodes are less complex. Tests that result in high information-gain values are often complex, and because beam search considers a greater number of candidate tests, it is more likely to find such a test.

The benefit of using a literal-pruning procedure when searching for m-of-n tests is not apparent in most of the domains. Accuracy values with and without the procedure are about the same, and tree sizes are also similar, except in the `NETtalk` domain. In this domain, however, the literal-pruning method is clearly beneficial. The ID2-of-3+ variant without literal pruning produces m-of-n tests that are considerably more complex than the ID2-of-3+ trees. Although one might expect that literal pruning would always result in simpler trees, this is not always the case. The reason for this is that once a test is modified by the literal-pruning routine, the tree structure induced from the corresponding node on down may be different than it would have been if literal pruning had not been applied. Sometimes this tree structure may be simpler, and sometimes it may not.

4.7 Chapter Summary

This chapter presented a series of experiments that empirically evaluated Trepan in the context of classification, regression, and reinforcement learning tasks. The purpose of the experiments was to evaluate Trepan along the dimensions of comprehensibility, fidelity, scalability, and generality. I discuss the experimental results with respect to each of these criteria in turn:

- **Comprehensibility:** In order to assess the comprehensibility of the trees extracted by Trepan, I compared their syntactic complexity to the complexity of trees induced by two conventional decision-tree algorithms run in the same problem domains. Overall, the trees extracted from trained networks by Trepan were of comparable complexity to the trees learned directly from the data by the ordinary decision-tree methods, and in some cases, the Trepan trees were much simpler. Moreover, the experiments in Section 4.5.1 showed that in many problem domains, Trepan can extract trees that provide good approximations to their target networks, but use only a few internal nodes.

- **Fidelity:** In the six classification domains considered in this chapter, the trees extracted by Trepan had fidelity levels ranging from 89.1% (`promoter` domain) to 97.7% (`voting` domain). As indicated by the experiments in Section 4.5, in some of these domains (e.g., `heart` and `NETtalk`) there is a trade-off between fidelity and tree complexity, and the level of fidelity can be increased by extracting larger trees. In other domains (e.g., `promoter` and `telephone`), the fidelity does not increase appreciably with larger trees. This somewhat disappointing result points to several directions for future research. As indicated in the experiments presented in Section 4.5.2, the level of fidelity in the `telephone` domain can be improved from 89.3% to over 95% by not using query instances. The reason for this anomalous behavior is that Trepan's simple instance model fails to adequately represent known feature dependencies in the domain, and thus misleading samples are formed from the instances generated by these models. Fidelity in domains such as this might be improved by exploiting more sophisticated instance models that are able to explicitly represent known dependencies.

Another explanation for the disappointing levels of fidelity in some domains is that the hypothesis language of decision trees is not well suited to concisely representing the functions learned by some networks. For example, in the `promoter` domain, the trained networks give significant weight to a large number of the 57 features. In such cases, a reasonably large decision tree is required to represent the target model. Thus, another line of future research that might improve the fidelity of extracted representations is to develop versions of Trepan that use different languages to represent their hypotheses.

Although the exchange-rate problem explored in Section 4.3 is a regression task, Trepan was applied to extract a tree describing the discretized behavior of the trained network. The fidelity of this tree to the network's discretized behavior was about 80%. However, although Trepan has a hard time fitting a decision tree to the small fluctuations in the regression network's decision surface, it does a good job of fitting the overall structure of the surface. This is evidenced by the profit-and-loss behavior of the extracted tree which closely modeled that of the network.

In the reinforcement learning task considered in Section 4.4, the fidelity of the extracted tree ranged from 89% to 92%, depending on the traffic distribution in the simulated system.

- **Scalability:** This chapter indirectly evaluated the scalability of Trepan by applying it in a variety of problem domains, some of which involve large feature sets and large networks. In one domain (`telephone`), Trepan was applied to ensembles of networks that, on average, involved more than 10,000 parameters per ensemble.

- **Generality:** The generality of Trepan was evaluated by applying it in classification, regression, and reinforcement-learning domains that involved both discrete and real-valued features. Moreover, the networks varied in their topologies, transfer functions, and training methods. Finally, as mentioned above, in one problem domain, decision trees were extracted to describe ensembles of classifiers, as opposed to individual neural networks.

This chapter also evaluated some of the key components of the Trepan algorithm by conducting a series of lesion experiments. The first lesion experiment measured the fidelity of trees extracted by TREPAN to those extracted by a variant of TREPAN that did not use m-of-n tests at its internal nodes. This experiment showed that, in general, trees that use m-of-n tests require fewer nodes to achieve a given level of fidelity.

The second set of lesion experiments compared the fidelity of trees extracted by TREPAN to three variants that involved modifications to TREPAN's mechanism for drawing query instances and using them for membership queries. First, TREPAN was compared to a variant that did not use local instance models. This experiment showed that, although local instance models have the potential to more accurately represent underlying distributions than do global models, this capability did not result in trees with higher levels of fidelity. The second experiment compared TREPAN to a variant that did not construct instance models, but instead selected samples assuming a uniform distribution over the instance space. This experiment illustrated the value of using instance models by showing that, in some domains, drawing samples uniformly leads to trees with poor levels of fidelity. The third experiment compared trees extracted by TREPAN to those extracted by a variant of TREPAN that did not use any query instances. In several of the problem domains, TREPAN without query instances produced trees that were clearly inferior to those extracted by TREPAN. In the `telephone` domain, however, the variant of TREPAN that did not use query instances extracted trees with noticeably better fidelity. This result, which is most likely due to the fact that TREPAN's instance models are failing to represent the numerous, significant feature dependencies in this domain, points out the importance of using more sophisticated, domain-specific instance models for some tasks.

The final lesion experiment compared the fidelity of trees extracted by TREPAN to those extracted by a variant of TREPAN that expanded trees one level at time. The results of the experiment indicated that, in the domains considered, there was not a noticeable advantage to being able to return an unbalanced tree. The primary value of TREPAN's method of best-first expansion is that it offers a finer degree of control over the size of tree to be returned by TREPAN.

Finally, this chapter also described and evaluated a variant of the ID2-of-3 decision-tree algorithm, which was used as a baseline for comparison in some of this chapter's experiments. This variant of ID2-of-3, which I developed, incorporates five enhancements over the original algorithm. The final set of experiments in this chapter used a set of lesion experiments to evaluate these enhancements, and showed that, in the six domains considered, they generally improved the predictive accuracy and syntactic complexity of induced trees.

Chapter 5

Analytical Evaluation of Trepan

Chapter 1 argued that rule-extraction algorithms should be evaluated with respect to the criteria of comprehensibility, fidelity, scalability, and generality. The experiments presented in the previous chapter directly measured the fidelity and comprehensibility of extracted representations. Additionally, by applying Trepan to a wide variety of learned models, the experiments provided indirect evaluation of Trepan along the dimensions of scalability and generality. In this chapter, however, I provide a formal discussion of scalability, as well as a more detailed discussion of the generality of Trepan. The first section in this chapter addresses the issue of scalability by analyzing the computational complexity of the Trepan algorithm. The second section frames the discussion of scalability in various learning-theoretic models, and the final section provides a discussion of the generality of the Trepan algorithm.

5.1 Computational Complexity of Trepan

This section considers the computational complexity of the Trepan algorithm. Following conventions used in the machine-learning literature, throughout this chapter I use m to refer to the size of a set of instances, and n to refer to the number of features in a domain. To avoid confusion about what m and n indicate, I refer to m-of-n tests as r-of-k tests throughout this chapter.

To begin this discussion, assume that we are expanding a node in the tree, that we have a sample (of training examples and query instances) of size m, and that there are n features in the problem domain. Furthermore, assume that the node being expanded is at depth d in the tree. I will begin the analysis by considering the complexity of the DrawSample and ConstructTest subroutines used by Trepan. Recall that DrawSample randomly selects a sample of instances from the part of instance space covered by a given node, and ConstructTest selects a splitting test to be used at an internal node in the tree.

5.1.1 Computational Complexity of DrawSample

The DrawSample routine (previously shown in Figure 10) involves calling DrawInstance (Figure 13) at most m times, and using the oracle to label each instance returned by DrawInstance. The complexity of the DrawInstance routine is dominated by the potential cost of satisfying the r-of-k constraints along the path from the root of the tree to the given node.

Consider the cost of satisfying one r-of-k test. Let v be the maximum number of values for a discrete-valued feature; in domains with only real-valued features, let $v = 2$. This is an upper bound on the number of references to a particular feature that may occur in an r-of-k test. Since there are n features, there are $O(nv)$ literals in an r-of-k test that need to be satisfied. On each iteration of the loop that selects literals to satisfy, Trepan needs to keep track of the probability that each literal will be selected; the complexity of this task is $O(nv)$. Additionally, Trepan needs to randomly select one of these literals and to update the conditional distribution for the corresponding feature. The complexity of this task is $O(m)$ in the more expensive case of selecting a literal on a real-valued feature. Therefore, the complexity of satisfying an r-of-k test is $O(mnv + n^2v^2)$.

The number of r-of-k tests that need to be satisfied is at most d (the depth of the tree). Hence, the complexity of DrawInstance is $O(dmnv + dn^2v^2)$. Since DrawSample must call DrawInstance up to m times, its complexity is $O(dm^2nv + dmn^2v^2)$, assuming it takes constant time to classify an instance (this assumption will be reconsidered shortly).

5.1.2 Computational Complexity of ConstructMofNTest

The time complexity of the ConstructTest routine (Figure 15) is dominated by the complexity of the search for an r-of-k test, and thus the discussion here focuses on the complexity of the ConstructMofNTest function (Figure 16).

The cost of evaluating an r-of-k test (by measuring its information gain) is $O(mnv)$ because the test is used to classify m instances, and the cost of classifying one instance with the test is $O(nv)$, since this bounds the number of literals it may have.

Each iteration of the beam search considers $O(mn)$ modifications to the tests in the beam: there are n features, and in the more expensive case of a real-valued feature, there are $O(m)$ possible thresholds to consider on the feature. Since the search can possibly continue until the test includes nearly all of the features, and as many as v references per feature, there are $O(nv)$ possible iterations in the search. Therefore, assuming that the beam width is a constant, the complexity of constructing an r-of-k test is $O(m^2n^3v^2)$.

5.1.3 Putting It All Together

The two previous sections analyzed the functions that dominate the cost of expanding a node in Trepan: the complexity of drawing a sample of instances is $O(dm^2nv + dmn^2v^2)$, and the complexity of constructing a test at an internal node is $O(m^2n^3v^2)$. Making the reasonable assumption that $d \leq n$ (i.e., the depth of our tree is less than the number of features in the domain), the complexity of expanding a node is dominated by the complexity of constructing an r-of-k test at the node. Thus the overall complexity of expanding a node is $O(m^2n^3v^2)$. The significant conclusion of this discussion is that the computational complexity of expanding a node in Trepan is polynomial in the sample size, the dimensionality of the instance space, and the maximum number of values for a discrete feature. Therefore, in terms of computational complexity, Trepan measures up well along the dimension of scalability.

Note that this discussion has analyzed the cost of expanding one node in a tree, not the cost of growing an entire tree. The size of the tree to be returned by Trepan, however, is a parameter of the algorithm, and thus can be viewed as a constant.

One assumption made earlier in this analysis was that the cost of classifying an instance with the oracle is constant. This may not be a reasonable assumption, depending on the type of learned hypothesis that Trepan is trying to describe. For example, if Trepan is extracting a decision tree to describe a nearest-neighbor classifier, then the cost of classifying an instance is $O(\tilde{m})$, where $\tilde{m}$ is the size of the training set. Since nearest-neighbor classifiers have an especially high cost of classifying examples, let us consider how the complexity of Trepan changes if we assume $O(\tilde{m})$ instead of constant complexity to classify an instance. Assuming that the sample size we are drawing for each node, m, is approximately equal to the size of the training set for the task, $\tilde{m}$, then the complexity of classifying the examples in the drawn sample is $O(m^2)$. Note that this is dominated by the complexity of drawing the instances in the first place, and thus the overall complexity of the DrawSample routine is still $O(dm^2nv + dmn^2v^2)$.

If Trepan is not allowed to use r-of-k tests when extracting a tree, then the complexity of expanding a node is only $O(dm^2n)$. The complexity of DrawInstance in this case is $O(dmn)$ since there are n features, as many as d conjunctive constraints for each one, and the cost of updating the conditional distribution for (the more expensive case of) a real-valued feature is $O(m)$. The cost of DrawSample, which must be called m times, is therefore $O(dm^2n)$. The complexity of ConstructTest when r-of-k tests are not used is $O(m^2n)$, since there are $O(mn)$ tests to evaluate, and the cost of evaluating each one is $O(m)$. Thus, the complexity of Trepan in this situation is determined by the complexity of DrawSample, not by ConstructTest as when r-of-k tests are allowed. The reader is referred back to Section 4.5.1 for an empirical view of Trepan without r-of-k tests.

5.2 Trepan and Learnability

The previous section analyzed scalability in terms of the computational effort required to extract a decision tree of a given size. In this section I consider a different view of scalability: how computational effort scales with the fidelity of the extracted tree. We expect that it would be computationally inexpensive to produce a decision tree (or whatever extracted representation) that has a low level of fidelity to a given network, but that it would be reasonably expensive to extract a tree that has a high level of fidelity. The question addressed in this section is the following: can we formally characterize how the fidelity of a tree extracted by Trepan scales as a function of computational effort? This is a more difficult issue to analyze than computational complexity, but it is one that is quite pertinent to the task of rule extraction. Since Trepan treats the rule-extraction task as a learning problem, this section discusses Trepan in the context of analytical frameworks that have developed in the field of *computational learning theory* (Kearns & Vazirani, 1994). Specifically, I discuss results for learning decision trees that have been proven for the PAC-learning, weak-learning, and agnostic PAC-learning models.

5.2.1 PAC Learning

The notion of *probably approximately correct (PAC) learning* was pioneered by Valiant (1984), and since has served as the dominant formal model of learning, as well as the foundation for numerous variants. In the PAC model, it is assumed that there is an unknown target function f and an arbitrary probability distribution D over the instance space, and the goal of a learning algorithm is to infer a hypothesis h that closely approximates f. The learner is given access to examples of the target function through an oracle, $EX(f, D)$, that randomly draws instances from distribution D and labels them using the target function f. The error of a hypothesis is defined as:

$$error(h) = Pr_D\left[f(x) \neq h(x)\right]$$

where Pr_D represents the probability with respect to the distribution D. A class of functions $\mathcal{F}$ is said to be *PAC learnable* if there is an algorithm $\mathcal{A}$, such that for any $f \in \mathcal{F}$, any distribution D, any $0 < \epsilon < 1/2$, and any $0 < \delta < 1/2$, then with probability at least $1 - \delta$, $\mathcal{A}$ produces a hypothesis h such that $error(h) \leq \epsilon$. The probability in this definition is taken over the random choices made by the EX oracle as well as any random choices made by the learning algorithm. The algorithm is said to be *efficiently PAC learnable* if it runs in time polynomial in the number of features n, the size of f, $1/\epsilon$, and $1/\delta$.

The two parameters, ϵ (sometimes called the *error parameter*) and δ (sometimes called the *confidence parameter*), represent two types of failure that are unavoidable in induction.

The error parameter reflects the fact that, without having complete knowledge of the data distribution, the learner is likely to produce a hypothesis that is only approximately correct. The confidence parameter reflects the fact that there is a small chance that the learner will draw an unrepresentative sample, and thus its hypothesis will not be as approximately correct as expected.

This formal model of learning has a number of properties that make it an appealing analog to the usual empirical-learning setting. The goal is to learn an unknown target function, and the learner is given randomly drawn training examples from some underlying distribution. The learned hypothesis is evaluated with respect to the same distribution from which the training data was selected, and we are concerned with being able to efficiently infer a hypothesis.

The key question addressed in this section, however, is what applicability does the PAC model have to the task of rule extraction? The answer is that, since Trepan views the extraction task as one of learning, the PAC model may provide an appropriate model for characterizing the scalability of Trepan in terms of fidelity. If we can show efficient PAC learnability for a rule-extraction algorithm, then we have a reasonable bound on how hard it is extract a representation of a given level of fidelity with a specified level of confidence. In this context, the rule-extraction algorithm is the learner, and the target function is the concept represented by the trained network. The goal in this task is to infer a representation of the target function that closely approximates it. It is important to note that in this discussion we are concerned with how accurately the extracted representation approximates the network; that is, the notion of accuracy in this discussion is what I have referred to as fidelity throughout this thesis.

A number of researchers have analyzed the learnability of decision trees in the PAC model (e.g., Rivest, 1987; Ehrenfeucht & Haussler, 1989; Hancock, 1990; Kushilevitz & Mansour, 1991; Bshouty, 1993). The upshot of these analyses is that, in general, decision trees are not known to be efficiently PAC learnable. There are a number of special cases in which decision trees are efficiently learnable, however. I discuss two of them, and their implications for Trepan, below.

Ehrenfeucht and Haussler

Ehrenfeucht and Haussler (1989) showed that in Boolean domains (i.e., domains in which all of the features are Boolean), decision trees of fixed *rank* [1] are efficiently PAC learnable.

What does this result say about the rule-extraction task, and specifically the scalability of Trepan? Unfortunately, the applicability of this result to the rule-extraction problem is somewhat limited. First, note the restriction on the target class; it applies only to trees of fixed rank. The complexity of Ehrenfeucht and Haussler's algorithm is exponential in the rank of the target tree, so it is really only practical for trees of small rank.[2] In practice, however, we do not know *a priori* how complex of a decision tree is required to describe the function represented by some trained network. The result is also limited in that it applies to problems with discrete-valued features only.

Another limitation on the applicability of this result is that we may not be able to get a large enough training set to satisfy specified ϵ and δ values. In order to ensure that a hypothesis satisfies the PAC criterion for specified values of ϵ and δ, we need to provide the decision-tree algorithm with a large enough training sample from the underlying distribution. In the rule-extraction setting, however, although we can use a network to label as many examples as necessary, we may not have access to a sufficient number of examples from the *true* distribution. Trepan addresses this problem by constructing estimates of the underlying distribution and sampling from these estimates. Although this is a reasonable heuristic, the PAC guarantees do not hold unless training data is drawn from the actual distribution over the instance space. Note, however, that to meet this requirement we need only to have a sufficient set of *unlabeled* data since the network (the target function) is used to label examples. In many problem domains there is plenty of unlabeled data, even though labeled examples may be scarce.

[1]The *rank* of a decision tree is defined recursively as follows. The rank of a leaf is 0. First, let r_L denote the rank of a tree's left subtree, and let r_R denote the rank of its right subtree. If $r_L = r_R$ for some tree T, then the rank of T is $r_L + 1$, otherwise T's rank is $\max(r_L, r_R)$. The rank of a tree is a function of its depth and the extent to which it is unbalanced.

[2]However, the rank is only logarithmic in the size of a tree. The algorithm may be practical, therefore, as long as the target tree is not very deep.

Bshouty

A second result pertaining to decision trees and PAC learnability is due to Bshouty (1993). He proved that decision trees in Boolean domains are *exactly learnable* when the learner is allowed to make membership and *equivalence queries* (Angluin, 1988). Recall that 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 oracle returns the class label for the instance. An equivalence query, on the other hand, involves giving the oracle a hypothesis, and asking if the hypothesis is equivalent to the target function. If it is equivalent, the oracle returns *true*, otherwise the oracle returns a counterexample – that is, an instance for which the supplied hypothesis and the target function disagree in their classifications.

Note that the learning model considered here is different from the standard PAC model in two ways. The first difference is that the goal of the learning task is exact identification of the target function. The second difference is that the learner is able to make membership and equivalence queries, whereas the standard PAC model assumes only that the learner has access to a source of randomly drawn examples.

Bshouty's result is interesting in the context of rule extraction, however, because it implies that decision trees in Boolean domains are efficiently PAC learnable using *only* membership queries. This fact is a corollary of a result by Angluin (1988) showing that if a function class is exactly learnable by equivalence queries, then it is PAC learnable by membership queries alone.

Although Bshouty's algorithm is able to efficiently learn the class of decision trees, it does not actually represent its hypotheses as decision trees. Instead, it represents them as depth-three AND/OR circuits. These hypotheses, however, may be reasonably comprehensible.

The important distinction, in the context of rule extraction, between this algorithm and that of Ehrenfeucht and Haussler, is that the Bshouty algorithm is able to efficiently learn a broad target class. The requirement that the learner make membership queries is not a realistic one for most learning problems, but in the context of rule extraction, it is quite appropriate – as illustrated by the Trepan algorithm.

The applicability of the Bshouty algorithm to the task of rule-extraction is still rather limited, however. First, as with the Ehrenfeucht-Haussler result, it applies only to discrete-valued problem domains. Second, the algorithm might not find a small hypothesis in cases where the target concept can be closely approximated by a small decision tree. That is, the running time of the algorithm is guaranteed to be polynomial in the size of smallest tree that exactly represents the target function – but such a tree may be extremely large in some cases.

5.2.2 Weak Learning

Recently, decision-tree learning has been analyzed in the context of the *weak learning* framework (Schapire, 1990; Freund & Schapire, 1995). The concept of a *weak learner* is one that is able to predict just slightly more accurately than random guessing. It is known that if we have a weak-learning method for some problem, then we can construct an algorithm that combines hypotheses produced the weak learner into a hypothesis that achieves arbitrary accuracy for the problem (Schapire, 1990). An algorithm that combines weak hypotheses in such a way is called a *hypothesis-boosting algorithm.*

More formally, let $\mathcal{F}$ be a any class of Boolean functions over the instance space of some problem domain. Then we say that a target function, f, satisfies the *weak-hypothesis assumption* with respect to $\mathcal{F}$ if, for any distribution D over the instance space, there is an $h \in \mathcal{F}$ such that $Pr_D\left[h(x) \neq f(x)\right] \leq 1/2 - \gamma$, where $\gamma \in (0, 1/2]$ and depends on the particular distribution.

Kearns and Mansour (1996) showed that slightly modified versions of standard decision-tree algorithms (including C4.5 and CART) are boosting algorithms when the target function satisfies the weak-hypothesis assumption with respect to the class of functions used as splitting tests. In other words, if for any distribution over the instance space, there is a splitting test (in the class of tests considered by the decision-tree algorithm) that has a slight predictive advantage over random guessing, then the algorithm can construct a decision tree that has arbitrarily high accuracy in the domain.

This result of Kearns and Mansour has some appealing aspects with respect to rule-extraction task. First, unlike the PAC-learning results discussed in the previous section, it applies to problem domains with both real-valued and discrete features, and it does not assume that the target function is a member of some predefined class. The only assumption made in this model is that the decision-tree learner will be able to find a weak hypothesis for any distribution.

It is important to point out that, although this analysis shows that common decision-tree algorithms are boosting methods, decision trees are not known to be *efficiently* learnable in the weak-learning model. Thus, although the analysis of Kearns and Mansour can be applied to show that TREPAN is a boosting algorithm, it fails to establish that TREPAN will efficiently learn the concept represented by a trained network (assuming the concept can be expressed as a finite decision tree).

5.2.3 Agnostic PAC Learning

The PAC-learning and the weak-learning models do not fully address the issues that are important in rule extraction. These frameworks are concerned with efficiently inducing an arbitrarily accurate model with arbitrarily high confidence. In the rule-extraction task, on the other hand, we are also concerned with the comprehensibility of the learned model. Thus, having a guarantee that we can efficiently extract a decision-tree of arbitrarily good fidelity may not be especially interesting if such a tree might be too complex to understand. In this section, I consider the model of *agnostic PAC learning* (Haussler, 1992; Kearns et al., 1992). In several respects the agnostic PAC model is more appropriate to analyzing the rule-extraction task than are the ordinary PAC and the weak-learning models.

Agnostic PAC learning differs from standard PAC learning in two important ways. First, in ordinary PAC learning it is assumed that training examples are drawn from a fixed distribution over the instance space and classified according to a target concept from a known concept class. In agnostic PAC learning, on the other hand, there is no notion of the target concept or class; it is simply assumed that instances are drawn from a fixed distribution over the instance space and then given class labels by some process. Thus, instead of learning a target function from a fixed class, the task in agnostic PAC learning is to learn from an arbitrary *target distribution* over the product of the instance space and the set of possible classes.

The second key difference between the two models is the notion of successful learning.

Recall that in the PAC model, the success of the learner is defined as follows: with probability at least $1 - \delta$, algorithm $\mathcal{A}$ outputs a hypothesis satisfying $error(h) \leq \epsilon$. In the agnostic PAC model, on the other hand, success means that with probability at least $1-\delta$, $\mathcal{A}$ outputs a hypothesis satisfying $\mid error(h) - \inf_{g \in H} error(g) \mid \leq \epsilon$, where H is the class of hypotheses considered by $\mathcal{A}$, and $\inf_{g \in H} error(g)$ indicates the greatest lower bound on the error of any hypothesis $g \in H$. Informally, this success criterion means that algorithm $\mathcal{A}$ produces a hypothesis that approximates the target function nearly as closely as possible, given the algorithm's hypothesis class.

As in the weak-leaning model, the lack of an assumption about the target function coming from a known class is an appealing aspect of the agnostic model. In the rule-extraction task, we do not usually have *a priori* knowledge that the function represented by a given network can be expressed as a small decision tree.

The second key aspect of the agnostic PAC model – the notion of learning success – closely parallels what I argue is the right measure of success for rule extraction. Namely, given a class of hypotheses, the learner should return the member of the class that provides the best description of the given network. This concept of success enables comprehensibility considerations to be incorporated into the formal model of extraction. For example, we might want our rule-extraction method to return the decision tree with no more than five nodes that has greatest level of fidelity to the given network.

The key question then, is what bearing do learnability results in the agnostic PAC model have on the rule extraction task? There are relatively few positive results for the agnostic model, but there is one that is of interest with regard to rule extraction. There is an algorithm called T2 (Auer et al., 1995) that is an agnostic PAC-learning algorithm for the class of decision trees that have two levels of internal nodes.

Could T2 be profitably used for rule extraction? The answer is a qualified yes. If run on a set of examples labeled by a target network, T2 would produce the two-level decision tree that minimizes error on the training set. Thus in order to ensure a hypothesis that satisfies the agnostic PAC criterion for specified values of ϵ and δ, we simply would need to give T2 a large enough training sample from the underlying distribution. As discussed in Section 5.2.1, however, in practice we might not be able to draw a large enough sample from the true distribution of data in the domain.

Another limitation of using T2 as a rule-extraction algorithm is that its hypothesis space – two-level trees – is rather limited. Auer et al. compared the predictive accuracy of T2 to C4.5 using 15 problem domains. T2 provided superior accuracy in only five of the domains, and in several others its accuracy was much worse than that of C4.5. Thus, the theoretical guarantees of T2 did not manifest themselves in especially good results in practice. Auer et al. showed that in several of the domains in which T2 performed poorly its performance is explained by its inability to form a hypothesis capable of representing the target concept. Although Trepan and ID2-of-3 often provide good solutions with decision trees that are only a few levels deep, they do so by using m-of-n tests at their internal nodes, as opposed to the single-feature tests that T2 uses.

Although the hypothesis space of two-level trees is quite limited, Auer et al. show that versions of T2 that consider deeper trees can be constructed, and that these versions are also agnostic PAC-learning algorithms. The complexity of finding a hypothesis for algorithm Td, which considers trees with d levels of internal nodes, is $O(n^d m^{d-1} \log m)$. The complexity increases exponentially in the depth considered, and thus such an approach is practical only for fairly shallow depths.

5.2.4 Discussion

In this section, I have argued that formal models of learning provide an appropriate framework for evaluating the scalability of rule-extraction algorithms. These models are especially relevant to Trepan since it views rule extraction as a learning task. The goal of considering the rule-extraction task in terms of formal learnability models is twofold: (i) to establish bounds on the computational expense of extracting a hypothesis with a specified level of fidelity to its target model, and (ii) to use insight gained from such analysis to improve rule-extraction algorithms such as Trepan. As evidence that the latter goal is reasonable, consider that a recently proposed improvement to C4.5 was based on analysis of the algorithm within the weak-learning model (Kearns & Mansour, 1996; Dietterich et al., 1996).

This section discussed learnability results for decision trees in three formal learning models. In the standard PAC model, decision trees are known to be efficiently learnable in Boolean domains only if the target class is restricted to trees of a fixed rank. Bshouty showed that if the learner is allowed to make membership queries, then arbitrary decision trees are efficiently learnable in Boolean domains. This algorithm, however, may not find a small hypothesis in cases where a small tree closely approximates the target concept. In the weak-learning model, decision trees are learnable when the weak-hypothesis assumption is met, but they are not known to be efficiently learnable. And finally, in the agnostic PAC model, decision trees of fixed depth are efficiently learnable. The complexity of Auer et al.'s algorithm for this task, however, increases exponentially in the depth considered. Empirical evidence indicates that some problem domains require a hypothesis class that includes trees of such a depth that this algorithm may not be practical.

This discussion might seem to suggest that decision trees are not an appropriate representation language for the rule-extraction task, given the dearth of poignant learnability results for them. However, learnability results for other "realistic" hypothesis languages are noticeably scarce as well.

In summary, it is still an open research issue as to which model of learnability provides the most appropriate context for evaluating rule-extraction algorithms such as Trepan. Another open question is under what minimally restrictive set of assumptions can Trepan, or some suitably-modified version of it, be shown to be efficiently learnable.

5.3 Generality of Trepan

As discussed in Chapter 2, many rule-extraction approaches suffer from a lack of generality. Some require a special training regime for networks, some place restrictions on the network architectures to which they can be applied, and others restrict both the training method and the architecture. Trepan, on the other hand, can be applied to a wide range of classifiers. In this section, I discuss in more detail the generality of the Trepan algorithm.

The assumptions made by Trepan about a given learned model are that (i) the input representation used for instances is a vector of feature values, and (ii) the model predicts discrete output categories. In fact, as illustrated by experiments in the previous chapter, it is not necessary that the model actually predict discrete values, as long as TREPAN can characterize its predictions in terms of discrete values. In comparison to most rule-extraction methods, these restrictions are very weak. Since TREPAN's interface to a learned model is solely through membership queries, it does not place any restrictions on the architecture (beyond the two mentioned above) of the model or the training method used for it. 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 instance-based learners and ensembles of classifiers, as illustrated by Chapter 4's experiments in the `telephone` domain.

The discussion of TREPAN so far, and the experiments evaluating the algorithm, have assumed that the rule-extraction task is to produce a description of the complete function represented by a learned model. Note, however, that TREPAN can also be applied to extract a description of a learned model in just part of its instance space. For example, if we wanted to extract a description of what a model predicts in a specific context, then we could simply hold constant certain feature values to represent the context, and generate query instances by varying values for the remaining features. This type of approach might be useful in cases where complete descriptions of learned models are too complex to comprehend. Alternatively, this approach might be useful for knowledge-transfer tasks (recall the discussion of this issue in Chapter 1) in which the entire model was not relevant to the target task in the transfer process.

In addition to extracting descriptions for specific contexts, TREPAN could also be applied to extract descriptions for specific components of a learned model. For example, TREPAN could be used to extract local descriptions for each individual unit in a neural network. Similarly, TREPAN could be applied to describe the discretized behavior of a group of units in a network (e.g., describe the conditions under which all the members of a set of hidden units had high activations). This capability might be useful in one were especially interested in *how* a hypothesis was represented, instead of *what* the hypothesis represented.

5.4 Chapter Summary

This chapter provided a discussion of the scalability and generality of the Trepan algorithm. The first part of the chapter analyzed Trepan in terms of computational complexity, and showed that Trepan scales well along this dimension. The computational complexity of expanding a decision-tree node in Trepan is $O(m^2n^3v^2)$, where m is the sample size, n is the number of features, and v is the maximum number of values for a discrete feature. When Trepan is not allowed to use r-of-k tests, then the complexity of expanding a node is only $O(dm^2n)$, where d is the depth of the tree.

The second part of the chapter argued that formal models of learning provide an appropriate setting for evaluating the scalability of rule-extraction algorithms. In this context, scalability refers to the computational expense of extracting a hypothesis with a specified level of fidelity to its target model. The conclusion of this section was twofold. First, it is still an open question as to which model of learnability provides the best framework for analyzing rule-extraction algorithms like Trepan. Second, another open issue is under what minimally restrictive set of assumptions can Trepan, or a modified version of it, be shown to be efficiently learnable.

The final section in this chapter discussed the generality of Trepan, arguing that the algorithm can be applied to a wide range of learned models that use feature-value input representations. In addition to extracting descriptions of entire learned concepts, it can be used to extract descriptions of learned models in localized regions of their instance spaces, and it can be applied to extract descriptions of specific components of learned models.

Chapter 6

The MofN-sws Algorithm: A Local Method for Extracting *M*-of-*N* Rules

This chapter presents an algorithm that belongs to the family of local methods for rule extraction. Recall from Chapter 2 that a local rule-extraction method is one that extracts a set of rules to describe each hidden and output unit in a network. The method I present here is based on the MofN algorithm which was developed by Towell and Shavlik (1993) for extracting *m*-of-*n* rules from knowledge-based neural networks. As discussed in Chapter 2, Towell and Shavlik's method is interesting because it simplifies the combinatorics involved in extracting local rules, and because it tends to produce relatively concise rule sets. Since the MofN algorithm was designed to be applied to knowledge-based neural networks (Towell & Shavlik, 1994), however, it makes certain assumptions about the distribution of weights in trained networks. The method I present in this chapter, called MofN-sws, is designed to extend the applicability of the MofN algorithm to ordinary neural networks.

I evaluate the accuracy and complexity of rules extracted by the MofN-sws method by comparing them to rules induced by the C4.5 system (Quinlan, 1993). Additionally, I compare MofN-sws, which was developed in the early stages of this thesis, to the more recent Trepan algorithm.

6.1 The MofN Algorithm

Towell and Shavlik's MofN algorithm comprises six steps:

1. **Clustering.** The weights impinging on each hidden and output unit of the trained network are grouped into clusters. Initially, each weight is treated as a cluster. The two nearest clusters are successively merged until no pair of clusters is closer than a pre-selected distance. The distance metric used for clustering is the difference in the means of the weight magnitudes for two clusters.

2. **Averaging.** The magnitude of each weight is set to the average value of the weights in its cluster.

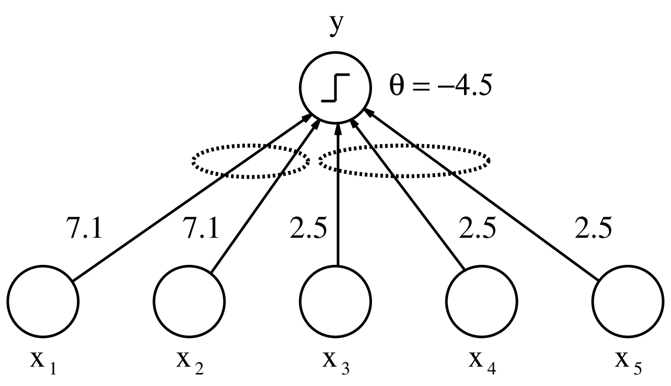
3. **Elimination.** Each unit's logistic transfer function is approximated by a threshold function, and weight clusters that do not have an effect on the threshold unit's activation are eliminated. Two elimination procedures are applied: one algorithmic and one heuristic. The algorithmic elimination procedure identifies clusters that *cannot* affect the activation of the unit. The heuristic elimination step eliminates clusters that do not have such an effect for any of the training examples.

4. **Optimization.** The unit biases are retrained to adapt the network to the changes that been imparted by the previous steps.

5. **Extraction.** Each hidden and output unit is translated into a rule with a threshold and weighted literals in its antecedent such that the consequent is true if the sum of the weighted literals causes the threshold to be exceeded.

6. **Simplification.** Weights and thresholds are eliminated and rules are expressed in the m-of-n format.

Figure 26 provides a simple example of the MofN method. The weights in this one-layer network are grouped into two clusters, resulting in two m-of-n rules. The *elimination* and *optimization* steps are not depicted in this example.



extracted rules: $y \leftarrow$ *1-of-*$\{x_1, x_2\}$
$y \leftarrow$ *2-of-*$\{x_3, x_4, x_5\}$

Figure 26: **Extracting rules using the** MofN **method.** The dotted ovals illustrate how the weights have been grouped into clusters. Each weight has been set to the average value of its cluster. The first extracted rule states that the output unit representing y will have an activation of 1 (i.e., the network predicts $y = true$), if either x_1 is true or x_2 is true. The second rule states that the output unit will have an activation of 1 if any two of x_3, x_4, or x_5 has a value of true.

6.2 Extending MofN with Soft Weight-Sharing

An underlying assumption of the MofN method is that the distribution of weights in the network will be conducive to forming a small number of clusters for each hidden and output unit. For knowledge-based neural networks, this is a reasonable assumption since the weights are clustered before training. For example, using the Kbann algorithm (Towell & Shavlik, 1994) to map a set of symbolic rules into a knowledge-based network, the weights that are specified by the domain theory have values of approximately 4 and -4, whereas the rest of the weights in the network have values near 0. Experimental evidence indicates that the weights tend to be fairly well clustered after training as well (Towell, 1991).

The applicability of the MofN method might seem to be limited to knowledge-based networks, since in conventional neural networks there is usually not an inductive bias leading weight values to be clustered after training. In fact, Towell (1991) reported that MofN did not extract small sets of accurate rules from conventional networks. My extension to the MofN algorithm, however, does not rely on the network weights initially being clustered, but instead encourages clustering *during* network training. In my MofN-sws approach, a method developed by Nowlan and Hinton (1992), termed *soft weight-sharing* (SWS), is used to encourage weights to form clusters during the training process. Although soft weight-sharing was motivated by the desire for better predictive accuracy, it is explored here as a means for facilitating rule extraction.

In the spirit of the minimum-description-length principle (Rissanen, 1978), soft weight-sharing uses a cost function that penalizes network complexity. Thus, during training the network tries to find an optimal trade-off between data-misfit (i.e., the error rate on the training examples) and complexity. The complexity term models the distribution of weights in the network as a mixture of multiple Gaussians. A set of weights is considered to be simple if the weights have high likelihood values under the mixture model. Specifically, the cost function in soft weight sharing is the following:

$$C = \lambda E - \sum_{i \in wgts} \log \left[\sum_{j \in Gauss} \pi_j \; p_j(w_i) \right]$$

where E is the usual error expression, λ is a parameter used to balance the trade-off between data misfit and complexity, w_i is a weight in the network, $p_j(w_i)$ is the density value of w_i under the jth Gaussian, and π_j is the mixing proportion of the jth Gaussian. The mixing proportions, which are parameters determining the influence of each Gaussian, are constrained to sum to one.

The partial derivative of this cost function with respect to each weight is the sum of the usual error derivative plus a term due to the complexity cost of the weight:

$$\frac{\partial C}{\partial w_i} = \lambda \frac{\partial E}{\partial w_i} - \sum_{j \in Gauss} r_j(w_i) \; \frac{(\mu_j - w_i)}{\sigma_j^2}.$$

Here μ_j and σ_j^2 are the mean and variance, respectively, of the jth Gaussian, and $r_j(w_i)$ is the conditional probability that w_i is explained by the jth Gaussian:

$$r_j(w_i) = \frac{\pi_j \; p_j(w_i)}{\sum_{k \in Gauss} \pi_k \; p_k(w_i)}.$$

Thus, the effect of each Gaussian is to pull each weight toward its mean with a force proportional to the density of the Gaussian at the value of the weight. When weights are pulled tightly around the means of the Gaussians, the network is similar to one that has fewer free parameters than connections (i.e., the situation in ordinary weight sharing, Rumelhart et al., 1986). The parameters of each Gaussian – the mean μ_j, standard deviation σ_j, and mixing proportion π_j – are learned simultaneously with the weights during training.

My MofN-sws method involves training networks using a variant of soft weight-sharing and then applying the MofN algorithm to the trained networks. Although the MofN method was designed for knowledge-based neural networks, my hypothesis was that it could be successfully applied to conventional networks, provided that the weights of the networks were grouped into clusters during training.

Whereas the MofN algorithm works best when the weights impinging on *each unit* form clusters, standard soft weight-sharing tends to *globally* cluster network weights. Hence, MofN-sws instead assigns a local set of Gaussians to each unit. The complexity cost of a given weight is calculated with respect to only the Gaussians associated with the unit to which the weight connects.

6.3 Empirical Evaluation

In this section, I present experiments that address the hypothesis that the MofN-sws approach is able to extract small sets of accurate rules. I evaluate the method by comparing the predictive accuracy and comprehensibility of rules extracted from neural networks by MofN-sws to rules learned directly from the training data using C4.5 (Quinlan, 1993). Additionally, I compare the MofN-sws method to Trepan by applying Trepan to the same networks.

Following the experimental design used in Chapter 4, I measure the syntactic complexity of the rule sets and use this as a proxy for comprehensibility. In the next section I discuss the specific aspects of syntactic complexity considered.

The experiments reported here use the `NETtalk` and `promoter` domains that were used in Chapter 4. As in those experiments, I conduct 10-fold cross-validation runs for both data sets.

6.3.1 Algorithms

The neural networks used in the experiments have fully-connected hidden units in a single layer. I use cross validation to select the number of hidden units to be used for the network for each training set. That is, using just the data in the training set, a cross-validation run is performed for networks with 20, 15, 10, 5, and no hidden units. The network architecture that results in the highest cross-validation accuracy is selected and then trained on all of the data in the training set. After the number of hidden units is selected for each network, I use a similar cross-validation procedure to determine the λ parameter for soft weight-sharing. I use a conjugate-gradient learning algorithm (Kramer & Sangiovanni-Vincentelli, 1989) to train the weights and the Gaussian parameters of the networks. Each hidden and output unit has five local Gaussians which act on the weights feeding into it.

I use the C4.5 system to induce decision trees and to convert the trees into rule sets. C4.5 generates a rule set from a decision tree by creating a conjunctive rule for each leaf in the tree. The literals in a rule's antecedent are determined by the splitting tests that lie on the path from the root to the leaf. C4.5 then simplifies the rule set by pruning literals from rule antecedents, and by pruning entire rules from the set. This pruning procedure may result in rules that are not mutually exclusive and exhaustive, meaning that a given example might be covered by rules for more than one class, or that an example might not be covered by any of the rules. C4.5 handles this problem in two ways: (1) rules are ordered by class, and the first rule to match a given example determines the predicted class; (2) a default rule is used to classify instances that are not covered by any other rule. C4.5's rule-generation algorithm is described in detail elsewhere (Chapter 5 of Quinlan, 1993).

The key parameter of C4.5's rule-pruning method is a confidence level analogous to the one used for decision-tree pruning (discussed in Chapter 2 of this thesis). I use cross-validation within each training set to determine the confidence levels for both tree pruning and rule pruning. The confidence level selected for tree pruning does not affect C4.5's tree-to-rule procedure since it operates on unpruned trees and performs its own pruning independently. For each training set, I test confidence levels ranging from 5% to 95%, and separately select tree-pruning and rule-pruning levels.

I apply the MofN algorithm to extract rules from the networks that are trained with soft weight-sharing. In the `promoter` domain (which is a two-class problem), I extract rules only for the positive (i.e., promoter) class, and employ the closed-world assumption (i.e., a default rule) to classify negative examples. Because local rule-extraction methods, such as MofN-sws, consider each output unit independently, the extracted rules may not be mutually exclusive and exhaustive when they are applied to networks trained for multi-class problems. As mentioned above, this problem is also faced by C4.5's rule-generation algorithm. For the `NETtalk` domain, which is a multi-class problem, MofN-sws uses the same strategy as C4.5 to handle this issue: it orders the rules by class, and uses a default rule to classify examples not covered by other rules. MofN-sws sets its rule ordering so that false-positive errors on the training set are minimized. Its default rule specifies the class which has the most training examples not covered by any rule.

I also apply Trepan to the networks trained with soft weight-sharing. I use the same parameter settings and methodology for Trepan as in Chapter 4's experiments. In order to compare the syntactic complexity of Trepan trees to the MofN-sws and C4.5 rule sets, I convert extracted trees to rule sets in a straightforward manner: a rule is created for each leaf in a tree, where the antecedent of the rule is the conjunction of splitting tests that occur on the path from the root to the leaf. Following the procedure used for MofN-sws, I retain rules only for the positive class in the `promoter` domain, and in the `NETtalk` domain, I retain rules for four of the classes, and use a default rule for the fifth class. Note that, unlike C4.5's tree-to-rule algorithm, the rules produced by this method implement the same function as their respective Trepan tree.

To compare the comprehensibility of the hypotheses produced by C4.5, MofN-sws, and Trepan, I consider three measures of syntactic complexity. I compare the complexity of rule sets by counting the number of rules and the total number of literals in rule antecedents in the sets. I also compare Trepan's trees to MofN-sws rule sets by counting the total number of *feature references* in each representation. As in Chapter 4, an m-of-n test or an m-of-n rule is counted as n feature references, since such an expression lists n feature values.

I use this feature-references metric, in addition to the number-of-literals measure, because the latter is biased against Trepan. When a Trepan tree is converted to a rule set, much of the structure of the tree redundantly appears in numerous rules. For example, an m-of-n expression at the root of a Trepan tree is replicated in every rule generated from the tree. The feature-references metric enables a direct comparison of Trepan's trees to the rule sets extracted by MofN-sws.

6.3.2 Results

Table 14 shows test-set accuracy results for the various algorithms considered here. As in the experiments reported in Chapter 4, neural networks perform significantly better on this task than C4.5 trees (and the rules generated from them) in both domains. Additionally, the predictive accuracy of the symbolic rules extracted by MofN-sws is fairly close to the accuracy of the networks themselves, and better than the C4.5 rules and trees in both domains. Similarly, the accuracy of the Trepan hypotheses is close to the networks and better than both C4.5 rules and trees. I test the statistical significance of accuracy differences using a paired-sample, two-tailed t-test. The symbol '$*$' marks results in cases where the accuracy of an algorithm is inferior to the accuracy of the neural networks at the $p \leq 0.05$ level of significance. Similarly, the symbols '$\bullet$' and '$\diamond$' mark results that are inferior to MofN-sws and Trepan, respectively, at $p \leq 0.05$.

Table 15 shows the average number of rules, the average number of literals in rule antecedents, and the average number of feature references for the C4.5, MofN-sws, and Trepan hypotheses. The rule sets extracted by MofN-sws and Trepan are comparable in terms of number of rules, and both methods produce significantly fewer rules than C4.5. The three methods differ a fair amount, however, in terms of the complexity of the individual rules they extract. The rules produced by C4.5 are purely conjunctive rules that tend to have few literals. The rule sets extracted by MofN-sws and Trepan, on the other hand, contain far fewer rules, but have far more literals in their antecedents. In terms of literals, Trepan produces the least complex rule sets in the `NETtalk` domain, and C4.5 produces the simplest rule sets in the `promoter` domain.

The Trepan rules appear to be more complex than the MofN-sws rules in the `promoter` domain. However, when the complexity of the hypotheses is measured in terms of feature references, they are nearly identical. As argued in the previous section, the number-of-literals

Table 14: **Test-set accuracy (%) for the MofN-sws experiments.** The symbol '$*$' marks results in cases where the accuracy of an algorithm is inferior to the accuracy of the neural networks at the $p \leq 0.05$ level of significance. Similarly, the symbols '$\bullet$' and '$\diamond$' mark results that are inferior to MofN-sws and Trepan, respectively, at $p \leq 0.05$.

method	`NETtalk`	`promoter`
neural networks	87.0	92.1
MofN-sws	$*$ 83.0	88.9
Trepan	$*$ 84.8	89.1
C4.5 trees	$* \bullet \diamond$ 80.9	$* \bullet \diamond$ 83.1
C4.5 rules	$* \bullet \diamond$ 79.9	$* \bullet \diamond$ 86.5

Table 15: **Rule-set complexity for the MofN-sws experiments.**

	# rules		# literals		# feature references	
method	`NETtalk`	`promoter`	`NETtalk`	`promoter`	`NETtalk`	`promoter`
MofN-sws	17.5	8.2	661.9	119.6	661.9	119.6
Trepan	15.2	6.7	337.0	217.7	71.7	119.7
C4.5 rules	233.5	23.2	466.5	47.3	466.5	47.3

metric is biased against Trepan because the complexity of Trepan's hypotheses increases when the trees are converted into rules.

In summary, Trepan clearly produces the least complex hypotheses in the `NETtalk` domain. In the `promoter` domain, the results are not as definite. Whereas, C4.5 induces rule sets that contain many simple rules, MofN-sws and Trepan produce rule sets that contain fewer, more complex rules. I argue that the three methods are roughly comparable in terms of the comprehensibility of their rule sets in this domain. However, it is worth noting that whatever additional complexity the network-extracted rules exhibit, results in significant gains in accuracy.

Table 16 shows test-set fidelity results for the MofN-sws rules and Trepan trees. As in Chapter 4, I measure fidelity using examples in the test sets. Recall that fidelity is defined as the percentage of test-set examples for which the classification made by an extracted model agrees with its neural-network counterpart. The results in the table indicate that, in both domains, the models extracted by Trepan provide better descriptions than MofN-sws of their corresponding networks. In the `promoter` domain, the difference in fidelity between the two algorithms is significant at $p \leq 0.05$ using a paired-sample, two-tailed t-test.

Table 16: **Test-set fidelity (%) for MofN-sws and Trepan.**

method	NETtalk	promoter
MofN-sws	91.5	89.3
Trepan	92.1	91.7

Table 17: **The effect of soft weight-sharing in the NETtalk domain.**

method	accuracy	# rules	# literals
networks with soft weight-sharing	83.0	17.5	616.1
ordinary networks	83.1	16.0	724.5

In order to evaluate the contribution of soft weight-sharing to the MofN-sws results presented in this section, I apply MofN-sws to networks trained without soft weight-sharing. Surprisingly, the results obtained for the `promoter` domain, in terms of rule accuracy and comprehensibility, are essentially the same as in the first experiment. The results for the `NETtalk` domain are presented in Table 17. Although soft weight-sharing does not seem to affect the predictive accuracy ability of the extracted rules, it has a significant impact on the concision of the rules. The rules extracted from the networks that employed soft weight-sharing had, on average, 108 fewer literals in their antecedents. This difference is significant at the 0.05 level using a paired-sample, two-tailed t-test.

6.4 Chapter Summary

The experiments in this section demonstrated that small sets of accurate, reasonably concise symbolic rules can be extracted from ordinary artificial neural networks by a local rule-extraction algorithm. The MofN-sws method presented here involves exploiting the effectiveness of the MofN algorithm by encouraging weight clustering during training. For two difficult problem domains, this approach was able to induce rules that resulted in better predictive accuracy than rules learned using C4.5.

The primary limitations of this approach are twofold. First, the MofN-sws method is limited in its applicability because it assumes that networks use logistic transfer functions, and that these sigmoids can be closely approximated by threshold functions. Second, like all local methods, the algorithm extracts a set of rules for each hidden and output unit. It thus implicitly assumes that individual units correspond to meaningful concepts. Moreover, the complexity of extracted rule sets may grow as a function of network size instead of as a function of the complexity of the concept represented by the network.

This chapter also compared the (more recent) TREPAN algorithm to MOFN-SWS. In the two domains considered here, the predictive accuracy of the trees extracted by TREPAN was slightly better than the accuracy of the MOFN-SWS rules. In the `promoter` domain, the syntactic complexity of the hypotheses produced by the two algorithms was comparable, but in the `NETtalk` domain, TREPAN produced hypotheses that were considerably more concise. TREPAN's extracted models also had higher levels of fidelity to their target networks than did the rule sets extracted by MOFN-SWS. Additionally, unlike the MOFN-SWS method, TREPAN does not assume that a special training procedure is used for the networks. In conclusion, although the experiments presented in this chapter demonstrated the value of the MOFN-SWS rule extraction method, they also showed the superiority of the more recently developed TREPAN algorithm.

The MOFN-SWS algorithm, and portions of the experimental evaluation of the algorithm reported in this chapter were previously published in a conference paper (Craven & Shavlik, 1993a).

Chapter 7

The Boosting-Based Perceptron Learning Algorithm

Unlike the methods presented previously in this thesis, the algorithm introduced in this section is not a method for extracting rules from neural networks. Rather, the Boosting-Based Perceptron (BBP) learning algorithm is a method for learning *sparse perceptrons.* A sparse perceptron is a linear discriminant function over relatively few terms, where the terms are either individual features or conjunctions of features.

The underlying motivation for the BBP algorithm, however, is closely related to the motivation for the rule-extraction task: to induce comprehensible models in domains in which neural networks have a well-suited inductive bias. In contrast to the rule-extraction approach, where one tries to extract a comprehensible representation from a trained network, the BBP approach is to directly induce a network that itself is simple and comprehensible. BBP tries to keep a learned hypothesis comprehensible by incorporating as few features as possible into the hypothesis, and by ensuring that the relationship between each feature and the target function is relatively simple.

The BBP algorithm is similar to C4.5 (Quinlan, 1993), and other decision-tree learning algorithms, in that it incrementally adds features to its hypothesis during learning. On the other hand, since BBP learns perceptrons, its inductive bias is more similar to that of multi-layer neural networks. An important difference between BBP and ordinary neural-network learning algorithms, however, is that BBP has a bias towards hypotheses that are easy to understand. The hypotheses learned by ordinary multi-layer networks are typically difficult to understand because they involve hundreds or thousands of real-valued parameters that represent nonlinear, nonmonotonic relationships between the input features and the output categories. BBP, on the other hand, is a constructive algorithm and thus does not start with a fixed number of parameters. Instead, it adds weighted links to its hypotheses only as deemed necessary. Importantly, each parameter in a BBP-learned hypothesis describes a simple (i.e., linear) relationship between an input feature and an output category.

Because it learns hypotheses that are linear discriminant functions with relatively few parameters, the perceptrons produced by BBP are usually much more comprehensible than multi-layer networks. As evidence for this position that sparse perceptrons are comprehensible, consider that linear discriminant functions are commonly used to express domain knowledge in fields such as medicine (Spackman, 1988) and molecular biology (Stormo, 1987). In short, BBP is in some sense a hybrid between the decision tree and multi-layer network approaches to machine learning, combining some of the interpretability of one approach with the powerful inductive bias of the other.

This chapter is based on joint work with Jeffrey Jackson. The BBP algorithm was co-developed with Jackson, and he is responsible for the technical contributions presented in Section 7.4. A description of the BBP algorithm and some of the experiments reported in this chapter were previously published elsewhere (Jackson & Craven, 1996).

7.1 Hypothesis Boosting and the AdaBoost Algorithm

The BBP algorithm is based on a *hypothesis-boosting* method called AdaBoost (Freund & Schapire, 1995). Recall from Chapter 5 that a hypothesis-boosting algorithm is one that combines the hypotheses produced by a *weak learning* algorithm into a *strong hypothesis*. Informally the notion of a *weak learner* or a *weak hypothesis* is one that predicts just slightly better than random guessing, whereas a *strong learner* or a *strong hypothesis* is one that achieves arbitrarily high accuracy.

More formally, let $\mathcal{F}$ be any class of Boolean functions over the instance space of some problem domain, and assume that a learner is given access to an oracle, $EX(f, D)$, that randomly draws instances from distribution D and labels them using the target function f. A *strong learning* algorithm $\mathcal{A}$ for a class of functions $\mathcal{F}$ is one such that for any $f \in \mathcal{F}$, any distribution D, any $0 < \epsilon < 1/2$, and any $0 < \delta < 1/2$, with probability at least $1 - \delta$, $\mathcal{A}$ is able to produce a hypothesis h such that $Pr_D\,[f(x) \neq h(x)] \leq \epsilon$. Here, Pr_D represents the probability with respect to distribution D. Strong learning is synonymous with PAC learning, and thus this definition is the same as the one used for PAC learning in Chapter 5. An algorithm $\mathcal{A}$ is said to be a *weak learning* algorithm for $\mathcal{F}$ if, for any target function $f \in \mathcal{F}$ and any distribution D, $\mathcal{A}$ is able to produce a hypothesis h such that $Pr_D\,[h(x) \neq f(x)] \leq 1/2 - 1/p(n, size(f))$, where p is a fixed polynomial in the number of features n, and a measure of the size of the target function.

Schapire (1990) first showed the following surprising result: any class $\mathcal{F}$ that is weakly learnable is also strongly learnable. In particular, given a weak learning algorithm for a class $\mathcal{F}$, there are general mechanisms for boosting the weak learner into a strong learner for $\mathcal{F}$. Several such boosting algorithms have been developed (Schapire, 1990; Freund, 1990; Freund, 1992), and one called AdaBoost (Freund & Schapire, 1995) is the basis of the BBP algorithm.

Figure 27 shows the version of AdaBoost on which our BBP algorithm is based. We assume here that the target function we are trying to learn, f, is a mapping from a set of Boolean features to a Boolean output: $\{0, 1\}^n \rightarrow \{-1, +1\}$. AdaBoost is given a set S of m training examples of the function f, and a weak-learning algorithm WeakLearn capable of efficiently producing a hypothesis that has error $\epsilon \leq (\frac{1}{2} - \gamma)$ with respect to any distribution D over the instance space of f. AdaBoost runs for $T = \ln(m)/(2\gamma^2)$ stages, where at each stage, it creates a probability distribution D_i over S and invokes WeakLearn(S, D_i) to find a hypothesis h_i that has $\epsilon_i \leq (\frac{1}{2} - \gamma)$ with respect to D_i. Informally, the probability distribution at each stage is calculated such that if h_i correctly classifies an example, then the weight on that example is reduced in the distribution D_{i+1} used at stage $i+1$. The effect of this strategy is to focus the weak learner on those examples that the hypothesis from the previous stage classified incorrectly.

At the end of the T stages, AdaBoost returns a hypothesis h which is a weighted threshold over the hypotheses $\{h_i \mid 1 \leq i \leq T\}$. If WeakLearn succeeds in producing a hypothesis

AdaBoost

Input: training set S of m examples of function $f : \{0,1\}^n \rightarrow \{-1,+1\}$;
learning algorithm WeakLearn that produces classifier with $\epsilon \leq (\frac{1}{2} - \gamma)$ given S
and any distribution D over S

```
 1.  $T := \frac{1}{2\gamma^2} \ln(m)$                          /* determine number of stages */
 2.  for all $x \in S$, $weight(x) := 1/m$
 3.  for $i := 1$ to $T$ do
 4.      for all $x \in S$, $D_i(x) := \frac{weight(x)}{\sum_{j=1}^{m} weight(x)}$      /* set distribution for stage */
 5.      $h_i :=$ WeakLearn($S$, $D_i$)                          /* learn weak hypothesis */
 6.      $\epsilon_i := 0$
 7.      for all $x \in S$                                 /* determine error of weak hypothesis */
 8.          if $h_i(x) \neq f(x)$ then $\epsilon_i := \epsilon_i + D_i(x)$
 9.      $\beta_i := \epsilon_i \,/\, (1 - \epsilon_i)$
10.      for all $x \in S$
11.          if $h_i(x) = f(x)$ then $weight(x) := \beta_i \, weight(x)$
```

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

Figure 27: **The AdaBoost algorithm.** Given a weak learning algorithm, boost a set of weak hypotheses into a strong hypothesis.

that has $\epsilon_i \leq (\frac{1}{2} - \gamma)$ at each stage, then AdaBoost's final hypothesis will be consistent with the training set (Freund & Schapire, 1995).

The hypothesis returned by AdaBoost is a thresholded, weighted sum over the outputs of the weak hypotheses. In other words, it is a perceptron over the weak hypotheses. The summed outputs of the weak hypotheses are thresholded using the sign function defined as follows:

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

7.2 The BBP Algorithm

AdaBoost is a general algorithm for boosting a weak learner into a strong learner. This section describes the BBP algorithm which is an instantiation of AdaBoost intended to be applied in practice. The key issues discussed in this section are how BBP: (i) finds a weak hypotheses, (ii) determines the number of boosting stages, and (iii) handles multi-class problems.

The basic version of the BBP algorithm that we describe is designed to work in Boolean classification domains with Boolean features. It can easily be applied to problem domains with nominal features by first replacing each v-valued nominal feature with a set of v Boolean features, each one representing a possible value of the feature.

Recall that AdaBoost forms a hypothesis from a set of constituent hypotheses that are produced by a weak learning algorithm, WeakLearn. The weak learning algorithm employed by BBP is shown in Figure 28. The basic idea of the WeakLearn algorithm is very simple: it tries to find a small conjunction of features that correlates well with the target function with respect to the given distribution. This algorithm begins by first generating the set of all non-trivial[1] conjunctions, C_k, of at most k of the given Boolean features. This set includes the function that is identically one, which can be thought of as the conjunction of zero features. WeakLearn treats these conjunctions as functions that map to $\{-1, +1\}$, corresponding to whether the conjunction is *false* or *true* in a given case, respectively. In

[1]We use the term *non-trivial conjunctions* to refer to those conjunctions that are not trivially false. For example, the conjunction $(x_1 = true \ \wedge \ x_1 = false)$ is a trivial conjunction.

```
WeakLearn
Input: training set $S$ of $m$ examples of function $f : \{0,1\}^n \rightarrow \{-1,+1\}$; distribution $D$ over $S$

1.  let $C_k$ be the set of non-trivial conjunctions of at most $k$ features
2.  for each $c_i \in C_k$
3.      $correlation(c_i) := E_D\left[f \cdot c_i\right]$          /* correlation of conjunction with target function */
4.      $value(c_i) := \frac{\mid correlation(c_i) \mid}{\text{size}(c_i)}$

Return: $c_i \in C_k$ with maximum $value(c_i)$
```

Figure 28: **The WeakLearn function used by BBP.** Find a conjunction that is a weak hypothesis for the target function with respect to the given distribution.

order to find a weak hypothesis, WeakLearn first calculates the correlation of each conjunction c_i in C_k to the target function f with respect to the given distribution D. WeakLearn then determines the value of each conjunction by taking its absolute value and dividing by the *size* of the conjunction. The size of a conjunction is simply the number of features in it (except that the identically one conjunction has a size of one).

The step of dividing $|correlation(c_i)|$ by $size(c_i)$ introduces a bias that is intended to make BBP prefer simple features over higher-order conjunctive features. The motivation underlying this heuristic is that higher-order conjunctive features make a hypothesis more difficult to understand. This heuristic is related to Rissanen's Minimum Description Length principle (Rissanen, 1989) in that it involves trading off a possible gain in accuracy (at least in the weak hypothesis) for reduction in the complexity of the final hypothesis.

Because the BBP algorithm uses exhaustive search over all conjunctions of size k, learning time depends exponentially on the choice of k. In the experiments reported here, we chose to use $k \leq 2$ throughout, since this choice results in reasonable learning times and classifiers with good predictive accuracy.

Another key aspect of the BBP algorithm is deciding when the boosting process should terminate. The AdaBoost algorithm uses the size of the training set and the weak-hypothesis advantage γ to determine the number of boosting stages. Since γ is not known *a priori*, however, it is not practical to determine the number of stages before the boosting procedure begins. One possible approach would be to continue the boosting process until either the overall hypothesis was consistent with all of the examples in the training set, or until a weak hypothesis could not be found. Such an approach, however, would be likely to produce an overly complex hypothesis and would be susceptible to over-fitting. The approach BBP takes is to use 10-fold cross-validation, within the training set, to determine an appropriate termination point. That is, before running BBP on the entire training set, it is run on each of the training subsets induced by the cross-validation partition. At each stage of each of these cross-validation runs, the predictive accuracy of the current hypothesis is measured using the data in the validation fold. The stage that results in the best averaged predictive accuracy during this cross-validation procedure is noted, and then BBP is run for this number of stages on the entire training set. To facilitate comprehensibility, BBP is limited to run for at most n stages, where n is the number of weights that would be in an ordinary perceptron for the task (i.e., one weight for each value of each feature).

So far the discussion has focused on learning concepts that involve only two classes. BBP handles multi-class problems by learning a sparse perceptron for each class. That is, the ith perceptron is trained to distinguish examples that are members of the ith class from those that are not. To classify a test example, BBP considers the value computed by each perceptron before the *sign* operator is applied. That is, BBP identifies the perceptron that computes the greatest weighted sum, and then predicts the class corresponding to this perceptron. In neural-network terminology, this can be thought of as picking the output unit with the greatest activation.

7.3 Empirical Evaluation

The underlying hypothesis of this research is that the BBP algorithm will provide a combination of good predictive accuracy and comprehensible models in a wide variety of real-world domains. To test this hypothesis, we evaluate the algorithm by comparing it to four other inductive learning methods in five problem domains. The models learned by BBP are compared to those learned by the other algorithms in terms of both predictive accuracy and syntactic complexity. As in previous experiments in this thesis, we use syntactic complexity as a substitute for comprehensibility since the latter is problematic to measure objectively.

The task of measuring comprehensibility in the experiments here is further complicated by the fact that the algorithms we compare have different hypothesis languages. For this reason, two different measures of syntactic complexity are used in the experiments. First, for all algorithms, we count the number of features that are incorporated into learned hypotheses. Second, for algorithms, such as BBP, that form hypotheses composed of weighted connections, we count the number of connections in the learned hypotheses.

7.3.1 Algorithms

The four other learning algorithms to which we compare BBP are:

1. multi-layer neural networks (Rumelhart et al., 1986) trained using a conjugate-gradient method (Kramer & Sangiovanni-Vincentelli, 1989);
2. decision trees induced using C4.5 (Quinlan, 1993);
3. the Relief feature-selection algorithm (Kira & Rendell, 1992a; 1992b) used in conjunction with C4.5;
4. the *Balanced* version of the Winnow algorithm (Littlestone, 1989; 1995).

The reasons for selecting each of these algorithms are as follows. Multi-layer networks and C4.5 are selected because they are two of the most commonly used inductive learning algorithms, and they have provided good predictive performance in a number of real-world domains. C4.5 is also interesting because it has a bias toward using features sparingly in its hypotheses. We include Relief because it is an example of a feature-selection method; it is designed to improve predictive accuracy and reduce hypothesis complexity by limiting the number of features used by a learning algorithm. Relief itself is not a learning method, but is used in tandem with one. In the experiments reported here, it is used with C4.5. Finally, Winnow is selected because, like BBP, it learns hypotheses that are linear-threshold functions, and because it has provably good performance when given irrelevant features. Additionally, Winnow has recently shown its practical value by demonstrating excellent predictive performance in an interesting, real-world application (Blum, 1995). We use the

Balanced version of Winnow, rather than one of the other versions (Littlestone, 1989), largely due to its empirical success.

We now describe these algorithms in more detail, and discuss how they are applied in the experiments.

Multi-layer Neural Networks

The multi-layer neural networks used in the experiments have logistic output units and one (or no) layer of logistic hidden units. For two-class problems, we use networks with one output unit; for problems with more than two classes, we use one output unit per class. The networks are fully connected between layers. For each training set, networks with 0, 5, 10, 20, and 40 hidden units are tried, and we use 10-fold cross validation *within* the training set to select the topology to be used for that training set. The networks are trained using the cross-entropy error function (Hinton, 1989) and a conjugate-gradient minimization algorithm (Kramer & Sangiovanni-Vincentelli, 1989). They are trained for a maximum of 50 search directions. During training, 10% of the training data is held aside as a validation set in order to decide when the weights should be saved (i.e., when training should be effectively stopped).

C4.5

We use C4.5 in the experiments to learn decision trees. Except for the pruning confidence level, the default settings are used for C4.5's parameters. To choose the pruning level for each training set, 10-fold cross validation is used. For each training set, we perform cross-validation runs using pruning levels of 5-95%, in 10% increments.

Relief

Although learning methods such as C4.5 and multi-layer neural networks are designed to distinguish relevant from irrelevant features during learning, a number of empirical studies have found that predictive accuracy can sometimes be improved by using an explicit *feature-selection method* in conjunction with an ordinary learning algorithm (Almuallim & Dietterich, 1991; Kira & Rendell, 1992b; Caruana & Freitag, 1994; John et al., 1994; Moore & Lee, 1994;

Skalak, 1994). In the experiments below, we use one such method – Relief (Kira & Rendell, 1992a; 1992b) – with C4.5 as the learning algorithm. Relief is a *filter*-type method for feature selection. Filter approaches involve a two-stage process: first a feature-selection algorithm is used to select a subset of candidate features, and then a learning algorithm is applied using only the selected subset of features.

Wrapper-type methods are an alternative to filter-based methods for feature selection. In a wrapper method, the learning algorithm is intrinsic to the feature-selection process. A wrapper method conducts a search through a space of feature subsets, using the learning algorithm in a cross-validation procedure to evaluate each subset. A purported advantage of wrapper methods over filter methods is that they use the inductive bias of the learning algorithm to evaluate feature subsets, whereas filter methods typically employ a different inductive bias than the learning algorithm. Filter methods, however, are typically much less computationally expensive than wrapper methods. Evaluating a single node in a wrapper-method search space can be expensive since it involves running a cross-validation procedure. Even if the search algorithm used in the wrapper approach is a greedy method, it can still involve considering a large number of nodes in the search space. For example, the problem domains used in these experiments have as many as 464 features; thus the branching factor at each node in the search can be as high as 464. For this reason, we do not use a pure wrapper-style method in the experiments, but instead use a hybrid filter-wrapper approach.

The Relief algorithm, which is shown in Figure 29, calculates a weight for each feature, and then returns the subset of features whose weights (after a normalization step) exceed a threshold τ. The main loop of Relief, which calculates the feature weights, involves comparing the feature values of a selected example, x, to randomly selected near-neighbors of x. In the experiments here, we randomly select among the two nearest neighbors of a given example x, and use the same function as Kira and Rendell to calculate the distance between the values of discrete-valued features when comparing x to its neighbor:

$$\text{Diff}(x_f, z_f) \equiv \begin{cases} 0 & \text{if } x_f \text{ and } z_f \text{ are the same} \\ 1 & \text{if } x_f \text{ and } z_f \text{ are different.} \end{cases}$$

An issue that arises when applying Relief in practice is deciding how to select an appropriate value for τ. The Relief algorithm defines a total ordering over the feature set, and τ

```
Relief
Input: training set $S$, number $m'$ of examples to randomly draw, feature set $F$, threshold $\tau$

 1. separate $S$ into $S^+$ (positive examples) and $S^-$ (negative examples)
 2. for all $f \in F$, $w_f := 0$
 3. for $m'$ iterations do
 4.     $x :=$ example randomly selected from $S$
 5.     $z^+ :=$ randomly selected example from $S^+$ that is close to $x$
 6.     $z^- :=$ randomly selected example from $S^-$ that is close to $x$
 7.     if $x$ is a positive example then
 8.         $\textit{near-hit} := z^+$, $\textit{near-miss} := z^-$
 9.     else
10.         $\textit{near-hit} := z^-$, $\textit{near-miss} := z^+$
11.     for all $f \in F$, $w_f := w_f - \mathrm{Diff}(x_f, \textit{near-hit}_f)^2 + \mathrm{Diff}(x_f, \textit{near-miss}_f)^2$
12. for all $f \in F$, $\textit{Relevance}_f := w_f / m'$

Return: the set of relevant features $F' \equiv \{f \mid \textit{Relevance}_f \geq \tau\}$
```

Figure 29: **The Relief algorithm.** Select a subset of the given features to use for a learning task.

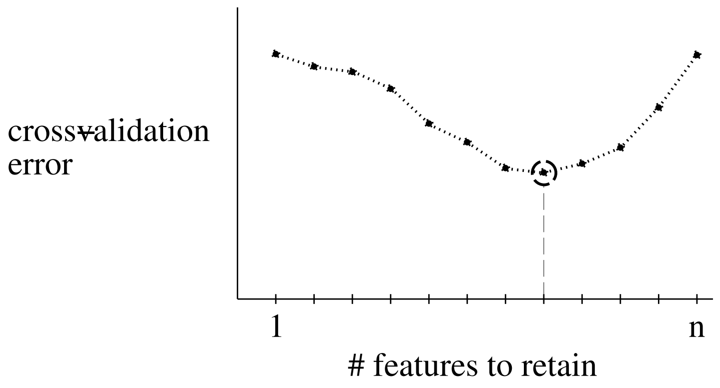


Figure 30: **Using golden-section search to find a cut-point for Relief.** An ordering is imposed on the features by the Relief algorithm. The x-axis indicates the number of features to be selected, and the y-axis indicates the corresponding cross-validation error. Golden-section search is used to find a local minimum in this function.

specifies where a cut-point should be made in this ordering to separate the features that will be retained from those that will be discarded. For the experiments here, we developed an adaptation of Relief that employs a simple search method to find this cut-point. As illustrated in Figure 30, this method views the task as a one-dimensional optimization problem, where the x-axis represents the number of features to retain and the y-axis represents the estimated predictive accuracy for the corresponding subset. In order to estimate accuracy for a given feature subset (i.e., to evaluate the function at a given point), we use a 10-fold cross-validation procedure. The optimization method we use is a modified version of *golden-section search* (Press et al., 1992). Our modification to this method is simple: whereas ordinary golden-section search assumes a continuous domain for the x-axis, this problem involves a discrete-valued domain. Thus, whenever the search method wants to evaluate the function (i.e., do a cross-validation run) at a given abscissa, the abscissa is first rounded to correspond to one of the cut-points. Golden-section search is guaranteed to find a *local* minimum in the function being optimized. For this application, this means that given an ordering over the full feature set, golden-section search will find a cut-point that results in a local minimum in cross-validation error; however, it may fail to find the cut-point that provides the global minimum.

Our variant of Relief is actually a hybrid filter-wrapper method. Like a conventional wrapper method, it conducts a search through a space of feature subsets and uses cross validation to evaluate each subset. However, the search conducted by this method is very limited. It is first constrained by using the Relief algorithm to specify a total ordering over the features. It is then further constrained by using an optimization method to consider a relatively small number of cut-points along this ordering.

Finally, this application of Relief requires one more twist. Whereas Relief is designed to be used for problems with two classes, several of the data sets used here are multi-class problems. For these problem domains, we run Relief once for each class, treating examples of the selected class as *positive* examples and the remaining examples as *negative* examples. This procedure produces a set of feature weights for each class. In order to obtain a combined feature ranking, we rank according to its *largest* weight.

With this modified Relief method, the only parameter that needs to be set is m – the number of examples that are randomly drawn. We set m to the training-set size in all of the experiments.

Winnow

The final algorithm used in the experiments is the *Balanced* version of Winnow (Littlestone, 1989; 1995). This algorithm learns a hypotheses that are like perceptrons, except that they use two thresholds, θ^+ and θ^-, and pair of weights, w_f^+ and w_f^-, for each feature f. The version of Winnow used here, which is shown in Figure 31, involves two modifications to the original algorithm. First, although Winnow is an *on-line* algorithm, we evaluate it in a *batch* setting in the experiments. An on-line algorithm is one that may update its current hypothesis after seeing each training example. Although Winnow is allowed to update its hypotheses after each example in the experiments, the learned hypothesis is not applied to a test set until the algorithm has made several passes through its training set. Thus, Winnow is trained much like the Boosting-Based Perceptrons and the multi-layer networks. The Winnow hypotheses are trained for a maximum of 50 passes through the training set, where 10-fold cross validation is used to select the number of training passes for each training set.

Winnow, like BBP, is designed to operate on Boolean features. Thus, before running Winnow in domains that have nominal features, each nominal-valued feature is first transformed to a set of Boolean features as is done with BBP.

We also adapt Winnow to handle problems involving more than two classes. As with the Boosting-Based Perceptrons, a hypothesis is trained for each class, and then the one with the highest "activation" is picked to classify a test example. For Winnow hypotheses, the activation is measured as:

$$(\theta^+ + \sum_{f=1}^{n} w_f^+ x_f) - (\theta^- + \sum_{f=1}^{n} w_f^- x_f).$$

Finally, following Littlestone (1995), we set the parameter α to 1.2 for all of the experiments.

Winnow

Input: training set S of m examples, set of n Boolean features F, learning rate α,
number of training passes P

```
1.   $\theta^+ := 1,$   $\theta^- := 1$                              /* initialize weights and thresholds */
2.   for all $f \in F$,  $w_f^+ := 1,$   $w_f^- := 1$
3.   for $P$ iterations do
4.       for each example $x \in S$ do
5.           $h(x) := sign\left(\left(\theta^+ + \sum_{f=1}^{n} w_f^+ x_f\right) - \left(\theta^- + \sum_{f=1}^{n} w_f^- x_f\right)\right)$      /* classify $x$ */
6.           if $h(x) \neq f(x)$ then          /* update weights & thresholds if $x$ misclassified */
7.               if $f(x) = -1$ then
8.                   $\theta^+ := \theta^+/\alpha,$   $\theta^- := \theta^-\alpha$
9.                   for all $f \in F$,  $w_f^+ := w_f^+ \alpha^{-x_f},$   $w_f^- := w_f^- \alpha^{x_f}$
10.              else
11.                  $\theta^+ := \theta^+\alpha,$   $\theta^- := \theta^-/\alpha$
12.                  for all $f \in F$,  $w_f^+ := w_f^+ \alpha^{x_f},$   $w_f^- := w_f^- \alpha^{-x_f}$
```

Return: $h(x) \equiv sign\left(\left(\theta^+ + \sum_{f=1}^{n} w_f^+ x_f\right) - \left(\theta^- + \sum_{f=1}^{n} w_f^- x_f\right)\right)$

Figure 31: **The Winnow algorithm.** Learn a hypothesis that consists of a pair of linear threshold functions.

7.3.2 Problem Domains

We use five problem domains to compare BBP and the other algorithms. The `promoter` and `voting` domains are used here again, as well as a version of the `coding` data set that has more features than the one used in Chapter 4. Recall that the `coding` data set involves classifying DNA sequences as to whether they come from a protein-coding region (i.e., a gene) or not. The `coding` data set used in Chapter 4 has 64 features representing the possible three-character DNA "words"; each of these features indicates whether or not the corresponding word occurs "in-frame" in the sequence. The `coding` data set used in the experiments in this chapter includes another 400 features which represent each possible pair of amino acids that are encoded by the sequence (each three-character word encodes an amino acid). Each of these features indicates whether or not the corresponding amino acid pair is found "in-frame" in the sequence. The experiments in this chapter use the larger feature set because we are interested in testing the ability of the BBP algorithm to sparingly incorporate features into its learned hypotheses.

In addition to the `coding`, `promoter`, and `voting` data sets the experiments here also use the `splice-junction` (Noordewier et al., 1991) and the `lung-cancer` (Hong & Yang, 1991) domains. The `splice-junction` data set is a three-class problem comprising 3,190 examples represented using 60 features. Like the `promoter` domain, each feature corresponds to a position in a DNA sequence and thus can take on one of four values. The `lung-cancer` data set is unusual in that it uses 56 four-valued features but includes only 32 examples. It also is a three-class problem. Note that all five of the data sets are classification tasks that involve only discrete-valued features.

7.3.3 Measuring Predictive Accuracy

The first hypothesis underlying these experiments is that the BBP algorithm is able to induce accurate models in a variety of problem domains. In order to test this hypothesis, we compare the test-set accuracy of BBP to the other algorithms in the study. For all five problem domains, we use a cross-validation methodology to measure predictive accuracy. We use ten-fold cross validation for the `voting`, `promoter`, `splice-junction`, and `lung-cancer` domains. As in Chapter 4, only four training and test sets are used for the `coding` data set because of certain domain-specific characteristics of the data. The experiments in this domain use a modified four-fold cross-validation methodology. As would be the case in ordinary cross validation, four independent test sets consisting of 5,000 examples each are used. For reasons of computational time, however, each classifier is not trained on the entire remaining set of 15,000 examples. Instead four training sets are formed by randomly selecting 5,000 examples from each set of 15,000.

Recall that the BBP algorithm considers incorporating into its hypotheses not just terms that correspond to individual features, but also terms that represent conjunctions of input features. Specifically, BBP considers all terms that represent non-trivial conjunctions up to size k. In the experiments, k is set to two for the `voting`, `promoter`, `splice-junction`, and `lung-cancer` domains. For the `coding` domain, where 400 of the features actually represent interesting conjunctions (amino-acid pairs), k is set to one. For the `voting`, `promoter`, `splice-junction`, and `lung-cancer` data sets, Winnow is run using two different feature representations. In the first, the original set of features for each domain is used; in the

Table 18: **Test-set accuracy for the BBP experiments.** The symbol '$*$' marks results in cases where the accuracy of an algorithm is inferior to the accuracy of BBP at the $p \leq 0.05$ level of significance. Similarly, the symbol '$\diamond$' marks the result where the accuracy of BBP is inferior to the accuracy of Winnow-Conj at this same level of significance.

method	`coding`	`lung`	`promoter`	`splice`	`voting`
BBP	93.6	56.2	92.7	$\diamond$ 94.6	91.6
multi-layer networks	93.6	46.9	90.6	95.4	92.2
C4.5	$*$ 84.9	31.2	$*$ 85.0	94.5	$*$ 89.7
Relief-C4.5	$*$ 88.4	53.1	$*$ 83.5	94.2	$*$ 89.2
Winnow	$*$ 86.7	53.1	90.8	$*$ 73.9	90.3
Winnow-Conj		43.7	92.1	95.0	90.3

second, the original feature set plus all non-trivial pairwise conjunctions are used. The latter configuration, which we refer to as Winnow-Conj, allows Winnow to explore the same hypothesis space as BBP. For the `coding` domain, we run Winnow only once using the ordinary feature set since this methodology parallels the application of BBP in this domain.

Table 18 displays the average test-set accuracy results for this experiment. In three of the domains (`coding`, `lung-cancer`, and `promoter`) BBP is as accurate, or more accurate, than all of the other learning methods. Only two other algorithms, in two domains, produce classifiers that are more accurate than those induced by BBP. Multi-layer networks are more accurate in the `voting` and `splice-junction` domains, and Winnow-Conj is more accurate in the `splice-junction` domain. We test the statistical significance of accuracy differences using a paired-sample, two-tailed t-test with a 0.05 level of significance. Cases in which the accuracy of another algorithm is less than BBP at $p \leq 0.05$ are indicated in the table by the symbol '$*$' next to the inferior result. There is one result in which the accuracy of BBP is less than another algorithm (Winnow-Conj) at $p \leq 0.05$. This case is indicated in the table by the symbol '$\diamond$' next to the BBP result.

7.3.4 Measuring Hypothesis Complexity

The second hypothesis tested by this chapter's experiments is that BBP learns models that are reasonably comprehensible. As stated previously, we use syntactic complexity as a proxy for comprehensibility. Specifically, we use two measures of hypothesis complexity. First, for all of the learning algorithms used in the experiments, we count the number of features incorporated into learned hypotheses. This measure is straightforward to determine for C4.5, which when building a decision tree, incrementally selects features to be added as splitting tests in the tree. Similarly, the BBP algorithm incrementally incorporates features into its hypotheses as it adds weighted connections to its hypotheses. However, for nominal features in a BBP hypothesis, each weight corresponds to a particular value of a feature, not to a feature itself. In counting the number of features used in a BBP hypothesis, we consider a feature to be included if at least one of its values has a weighted connection. For multi-layer networks, we consider all features to be incorporated into the learned model, since all features are part of the connected structure of the networks.[2] Like a multi-layer network, a Winnow hypothesis retains weighted connections to all features. However, when the positive weight for a given feature is the same as its negative weight in a Winnow hypothesis, then effectively the feature is not being considered. Like the BBP algorithm, Winnow expands nominal-valued features into a set of Boolean features, and thus each pair of weights corresponds to a particular value of the feature. To determine the features used in a Winnow hypothesis, we count the number of features for which at least one value has unequal positive and negative weights. For BBP and Winnow hypotheses which involve multiple perceptrons, we consider a feature to be used if any of the constituent perceptrons uses the feature.

Table 19 shows the average number of features each algorithm uses for each data set. In one domain (`voting`), BBP uses fewer features than all of the other algorithms. In the other four domains, C4.5 and Relief-C4.5 use fewer features than BBP, but the multi-layer networks and both versions of Winnow use more. Although C4.5 and Relief-C4.5 produce simpler hypotheses than BBP in these four domains, recall that their predictive accuracy was inferior to BBP in every domain.

The second measure of complexity considered is the number of weights used in learned hypotheses. This measure is not applicable to C4.5 and Relief-C4.5 since the hypotheses learned by these methods do not use weighted connections. However, it does enable a

[2]It would be possible to augment the neural-network training procedure with a pruning method that trimmed extraneous weights from the network during training (e.g., Le Cun et al., 1989). Such a method might produce networks that do not use all of the features. We do not consider this extension in the experiments, however.

Table 19: **Hypothesis complexity (# features used) for the BBP experiments.**

method	`coding`	`lung`	`promoter`	`splice`	`voting`
BBP	171	7	30	56	8
multi-layer networks	464	56	57	60	15
C4.5	150	5	10	22	10
Relief-C4.5	115	4	7	14	10
Winnow	464	56	57	60	15
Winnow-Conj		56	57	60	15

Table 20: **Hypothesis complexity (# weights used) for the BBP experiments.**

method	`coding`	`lung`	`promoter`	`splice`	`voting`
BBP	172	8	41	189	12
multi-layer networks	9,300	3,741	2,267	2,934	651
Winnow	464	308	215	718	28
Winnow-Conj		13,278	21,095	78,218	393

comparison of the complexity of the models learned by the BBP algorithm to the multi-layer neural networks and to the Winnow hypotheses.

Obviously, the number of weights used in each hypothesis is at least as large as the number of features used, and for most hypotheses, the number of weights is much larger. One reason for this fact is that each nominal feature has several possible values, and learned hypotheses often refer to several different values of the same feature. Moreover, the BBP and Winnow hypotheses may reference many more weights than features because they can have weights connected to conjunctive features. Similarly, although the multi-layer networks are not given conjunctive features, they have weights which connect the input features to hidden units, and the hidden units to output units.

Finally, recall that although cross validation is used to determine the number of boosting stages for BBP on each training set, an upper bound of at most n stages was set, where n is the number of weights that would be in an ordinary perceptron for the task. Thus, by design, the BBP hypotheses can have no more weights than the ordinary Winnow hypotheses.

Table 20 reports the average number of weights used by each algorithm for each data set. The results in this table show that, for all of the data sets considered, the BBP algorithm produces hypotheses that are much less complex than those learned by multi-layer networks

Table 21: **Summary of BBP Hypotheses.**

domain	features used (%)	weights used (%)	conjunctive weights (#)
`voting`	53	40	2.4
`promoter`	53	18	1.2
`splice-junction`	93	26	43.4
`lung-cancer`	13	4	0.3
`coding`	37	35	

and Winnow-Conj. The ordinary Winnow hypotheses are closer to the BBP hypotheses, in terms of complexity, but the BBP hypotheses are still less complex for every problem domain.

Table 21 summarizes some of the comprehensibility results for BBP and provides additional information about the hypotheses learned by the algorithm. The second column from the left lists, for each domain, the percentage of the domain's features that were used, on average, in BBP hypotheses. The next column lists the percentage of the possible weights that were used for each domain. Recall that BBP hypotheses were allowed to use up to n weights per perceptron, where n is the number of weights that would be used in an ordinary perceptron (or a Winnow hypothesis) for the domain. The final column lists the average number of conjunctive weights in the BBP hypotheses.

7.3.5 Discussion

From the results of the experiments, it can be seen that the BBP algorithm is able to induce classifiers that are accurate on a variety of interesting, real-world problems. Its predictive accuracy in the experiments was, with one exception, comparable to (and in some cases, better than) the other the algorithms evaluated.

The BBP algorithm is also strong in terms of comprehensibility. Unlike the multi-layer networks and Winnow hypotheses, BBP used only a subset of features in its hypotheses. In the `voting` domain, BBP even used fewer features than C4.5 and Relief-C4.5, although these decision-tree algorithms used fewer features in the other four domains. However, BBP was more accurate than C4.5 and Relief-C4.5 in every domain, and in three of the five domains these differences were statistically significant. Thus, although C4.5 and Relief-C4.5 may have found the most comprehensible models, they did so at the expense of reduced accuracy in several domains.

Besides incorporating fewer features into its hypotheses than the other perceptron-type algorithms (multi-layer nets, Winnow, and Winnow-Conj), in all domains, BBP was also superior in terms of the number of weights used. Winnow, without conjunctive features, was the closest to BBP in this measure. However, this version of Winnow was not as accurate as BBP in any domain, and in two domains, the differences in accuracy between Winnow and BBP were statistically significant.

It is important to note that if we allow the BBP algorithm to add weighted connections to all of its candidate terms, then its hypothesis space is the same as that of the Winnow-Conj method (or ordinary Winnow for the `coding` task). One advantage of the BBP algorithm, however, is that because it incrementally adds weights to its hypotheses, we can place a limit on the acceptable complexity of a hypothesis returned by the algorithm. As stated earlier, this restriction was in fact used in the experiments. An interesting result, however, is that the BBP hypotheses usually incorporate significantly fewer weights than permitted. As shown in Table 21, BBP usually used less than 40% of the possible weights. The Winnow hypotheses, on the other hand, use nearly all of the allowable connections in their hypotheses. The net result, we contend, is that the BBP hypotheses are much easier to understand than those produced by Winnow. It might be possible, however, to obtain more comprehensible Winnow hypotheses by pruning away some of the weights after training; this would be an interesting experiment to conduct in future work. One of the appealing aspects of the BBP algorithm in comparison to Winnow, however, is that its preference for simple hypotheses is built into the algorithm and does not require a special post-processing procedure.

Although Table 20 uses the same basis for comparing multi-layer networks and the other algorithms in this study, it should be emphasized that the weights in a multi-layer network are typically much more difficult to interpret than the weights in a BBP or Winnow hypothesis. Part of the reason that multi-layer networks are difficult to understand is that many weights may act together to encode nonlinear, nonmonotonic relationships between each feature and the problem classes. On the other hand, in a BBP or Winnow hypothesis that does not use any conjunctive features, each weight encodes a simple, linear relationship between a feature value and an output category. Conjunctive features may complicate a BBP or Winnow hypothesis because they describe feature interactions—and thus potential nonlinear relationships—between inputs and outputs. However, these interactions are made explicit in a BBP/Winnow hypothesis, unlike in a multi-layer network. Furthermore, we argue that perceptrons with conjunctive features are nearly as comprehensible as ordinary perceptrons if the use of conjunctive features is limited as it appears to be in BBP. The rightmost column in Table 21 shows that, for most domains, BBP includes few conjunctive weights in its perceptrons. The exception is the `splice-junction` domain, in which BBP uses 43 conjunctive weights on average. This result might explain why Winnow without conjunctive features generalized poorly in this domain.

Finally, we discuss the issue of how the BBP algorithm compares to TREPAN. In the experiments reported in this chapter, the BBP hypotheses had comparable accuracy to the trained multi-layer networks. Thus one might ask, why bother developing rule-extraction algorithms when accurate and comprehensible hypotheses can be learned using a method such as BBP? The first answer to this question is that, although it may not have been exhibited in this chapter's experiments, there are surely problem domains in which multi-layer networks will have better predictive accuracy than the hypotheses learned by BBP. The two algorithms have different inductive biases, and as stated in Chapter 1, no learning algorithm has a universally superior bias (Wolpert, 1995). A related argument is that there are other learning methods that produce hard-to-understand hypotheses. As argued elsewhere in this thesis, TREPAN is general enough that it can be used to understand a wide range of learned models. The third argument for the utility of rule-extraction algorithms is that the BBP algorithm is more limited in its range of applicability than are multi-layer neural networks. The BBP algorithm currently handles only discrete-valued features, and it is not applicable to regression or reinforcement learning tasks.

7.4 Analytical Evaluation

In addition to demonstrating good performance in practice, the BBP algorithm is interesting in that a slightly modified version of it is able to efficiently PAC learn a fairly natural class of target concepts. This section provides a formal definition of the class of *sparse perceptrons* and presents a proof that the (modified version of the) BBP algorithm is an efficient PAC learning algorithm for the class of sparse perceptrons.

The key difference between the practical version of BBP presented in Section 7.2 and the theoretical version considered here, is that the latter version continues the boosting process until it has found a hypothesis consistent with its training set. Recall that the practical version, on the other hand, may stop the boosting process before the induced perceptron is completely consistent with the training set. This heuristic is employed in the practical version for two reasons. First, it may not be possible to find a consistent hypothesis, and second, even if a consistent hypothesis is found, it may over-fit the training data.

7.4.1 Sparse Perceptrons

We begin the formal analysis of the BBP algorithm by giving a precise definition of sparse perceptrons. Consider the Boolean function $f : \{0,1\}^n \to \{-1,+1\}$. Let C_k be the set of all conjunctions over at most k of the n input features to f. For example, the Boolean function that outputs 1 if and only if the first two features both have value 1 is in C_k for all $k \geq 2$. For all k, C_k includes the "conjunction" of zero inputs, which is taken as the identically one function, and for $k \geq 1$, C_k includes "conjunctions" of one input (i.e., individual input features). As before, we assume that all of the functions in C_k map to $\{-1,+1\}$.

The function f is a *k-perceptron* if there is some integer s such that

$$f(x) = \text{sign}\left(\sum_{i=1}^{s} c_i(x)\right) \tag{2}$$

where for all i, $c_i \in C_k$, and the sign function is defined as before:

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

Note that while this definition does not explicitly indicate any weights, it implicitly assumes integer weights in that a particular $c_i \in C_k$ is allowed to appear more than once in the sum defining f.

We refer to a given collection of s conjunctions $c_i \in C_k$ as a *k-perceptron representation* of the corresponding function f, and we refer to s as the *size* of the representation. The size of a given k-perceptron function f is defined as the minimal size of any k-perceptron representation of f. An *s-sparse* k-perceptron is defined to be a k-perceptron f such that the size of f is at most s. Finally, we denote by $\mathcal{P}_k^n$ the set of Boolean functions over $\{0,1\}^n$ which can be represented as k-perceptrons, and we define $\mathcal{P}_k = \cup_n \mathcal{P}_k^n$.

In the following section, we will also be interested in the class of Boolean functions $\mathcal{H}_r$ defined as follows: a function $f : \{0,1\}^n \to \{-1,+1\}$ is in $\mathcal{H}_r$ if there is a linear function g (with real-valued coefficients) defined over at most r of the n input features to f such that $\text{sign}(g(x)) = f(x)$ for all $x \in \{0,1\}^n$. In other words, $\mathcal{H}_r$ is the class of Boolean functions over $\{0,1\}^n$ that can be represented as the sign of a real-weighted sum of at most r features (i.e., perceptrons with real-valued weights).

7.4.2 PAC Learning Sparse k-Perceptrons

We now show that the BBP algorithm is an efficient PAC algorithm for the class $\mathcal{P}_k$ using the hypothesis class $\mathcal{H}_r$. While there are already known algorithms for PAC learning the general class of perceptrons (e.g., Blumer et al., 1989), an atypical aspect of the BBP algorithm is that when learning target functions which are in the class of sparse perceptrons, it produces hypotheses that are relatively sparse themselves. In particular, if the target function f is an s-sparse k-perceptron then, with high probability, the learned hypothesis will be a real-weighted perceptron over at most $O(s^2)$ features (ignoring logarithmic factors) drawn from the set of conjunctions C_k. It should be noted that the proof that the BBP algorithm PAC learns $\mathcal{P}_k$ is implicit in more general work by Freund (1993), although he did not identify k-perceptrons as a class of particular interest, or present an algorithm for this class. The analysis presented here of the sample size needed to ensure that a hypothesis has accuracy of $(1-\epsilon)$ is also novel.

The first step in the proof that the BBP algorithm PAC learns the class $\mathcal{P}_k$is to show that the WeakLearn function can weakly learn $\mathcal{P}_k$. Following Freund, we begin with the following lemma (Goldmann et al., 1992).

Lemma 1 (Goldmann Hastad Razborov) *For $f : \{0,1\}^n \rightarrow \{-1,+1\}$ and H, any set of functions with the same domain and range, if f can be represented as*

$$f(x) = \text{sign}\left(\sum_{i=1}^{s} h_i(x)\right)$$

where $h_i \in H$, then for any probability distribution D over $\{0,1\}^n$, there is some h_i such that

$$\Pr_D[f(x) \neq h_i(x)] \leq \frac{1}{2} - \frac{1}{2s}.$$

If we specialize this lemma by taking $H = C_k$, then this implies that for any k-perceptron function f of size s, and any probability distribution D over the input features of f, there is some $c_i \in C_k$ that weakly approximates f with respect to D. Therefore, given a training set S and distribution D that has non-zero weight only on instances in S, the weak learning algorithm WeakLearn needs only to test each of the $O(n^k)$ possible conjunctions of at most k features until it finds a conjunction that has $\epsilon \leq (\frac{1}{2} - \frac{1}{2s})$ with respect to D. Any such conjunction can be returned as the weak hypothesis. The above lemma proves that if f is a k-perceptron then this exhaustive search will succeed at finding a weak hypothesis.

The next step in the analysis is to consider how many boosting stages are required to find a hypothesis consistent with a training set of size m. In terms of the parameter γ of the AdaBoost algorithm, the above result shows that the algorithm WeakLearn has $\gamma \geq 1/2s$ when the target is an s-sparse k-perceptron. Recall that the AdaBoost algorithm, as shown in Figure 27, runs for $T = \frac{1}{2\gamma^2}\ln(m)$ stages. Therefore, given a training set with m examples labeled by an s-sparse k-perceptron f, the BBP algorithm need only run for $2s^2\ln(m)$ stages before it will produce a classifier consistent with f over S. Because each stage adds one weak hypothesis (conjunction) to the hypothesis, the final hypothesis will be a linear threshold over at most $2s^2\ln(m)$ conjunctions in C_k.

We have now shown that, given a training set of m examples from an s-sparse k-perceptron, the BBP algorithm finds consistent hypothesis in the hypothesis class $\mathcal{H}_r$, where $r = 2s^2\ln(m)$. Often, finding a hypothesis consistent with a large enough set of examples produced by an example oracle $EX(f, D)$ is sufficient to guarantee PAC learnability. For example, given a finite set of functions $\mathcal{F}$, it is straightforward to show the following (Haussler, 1988).

Lemma 2 (Haussler) *Let $\mathcal{F}$ be a finite set of functions over a domain X. For any function f over X, any probability distribution D over X, and any positive ϵ and δ, given a set S of m examples drawn consecutively from $EX(f, D)$, where*

$$m \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + \ln|\mathcal{F}|\right),$$

then

$$\Pr[\exists h \in \mathcal{F} \mid \forall x \in S\ f(x) = h(x)\ \&\ \Pr_D[f(x) \neq h(x)] > \epsilon] < \delta,$$

where the outer probability is over the random choices made by $EX(f, D)$.

In other words, given enough training data, any hypothesis $h \in \mathcal{F}$ that is completely consistent on the training data with the target function f is likely to have its error be less than ϵ with respect to D. Notice that the concept of "enough" training data is a sample size that is polynomial in the parameters describing the difficulty of the learning problem.

Now let us instantiate this lemma for the hypothesis class used by the BBP algorithm, $\mathcal{H}_r$. Each function $f \in \mathcal{H}_r$ can be defined as a linear threshold over a set R of Boolean features, where $|R| \leq r$. For a given such R there are at most $2^{(r+1)^2}$ distinct Boolean functions defined by linear thresholds over R (e.g., Bruck, 1990). Since there are at most n^r such sets R:

$$\ln|\mathcal{H}_r| \leq (r+1)^2 \ln 2 + r \ln n.$$

Finally, we consider how large m must be to ensure that if BBP learns a hypothesis consistent with the training set, that the hypothesis will, with high probability, have accuracy of $(1 - \epsilon)$ with respect to an arbitrary distribution D. If the training set is produced by drawing m examples from $EX(f, D)$, where f is an s-sparse k-perceptron and

$$m \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + 2s^2\ln((2n)^k + m) + (2s^2\ln m + 1)^2\ln 2\right),$$

then with probability at least $1 - \delta$ the BBP algorithm produces a hypothesis that has accuracy $(1 - \epsilon)$ with respect to D. The probability here is taken over the random draws by $EX(f, D)$. Note that m has only a logarithmic dependence on n, ignoring the small dependence contributed by the $\log m$ terms on the right-hand side of the above inequality.

In summary, the following is an efficient PAC algorithm for $\mathcal{P}_k$: calculate a sufficiently large m, create a training set by drawing m examples from $EX(f, D)$, and run BBP on this training set. The required sample size, m, is polynomial in the relevant parameters, as is the number of stages required by the boosting algorithm.

7.5 Chapter Summary

This section presented a boosting-based algorithm for learning sparse perceptrons; that is, perceptrons with relatively few non-zero weights. The BBP algorithm, which is based on Freund and Schapire's AdaBoost hypothesis-boosting method, is a constructive algorithm that incrementally adds weighted connections to its hypothesis. The BBP algorithm was evaluated in this chapter both theoretically and empirically. The theoretical results show that the BBP algorithm is able to efficiently PAC learn target functions that belong to the class of sparse perceptrons.

The empirical evaluation demonstrated that the algorithm works well in practice. In terms of predictive accuracy, BBP was competitive with the best of the algorithms in the study. Along the dimension of comprehensibility of learned models, BBP also compared well with the other algorithms. By the measures used here, the only algorithms that were potentially superior to BBP in terms of learning comprehensible hypotheses were C4.5 and Relief-C4.5. However, for most domains these algorithms were clearly inferior to BBP in terms of predictive accuracy.

We argue that the BBP algorithm provides an appealing combination of strengths. First, it provides learnability guarantees for a fairly natural class of target functions. Second, it has been shown to provide good predictive accuracy in a variety of real-world tasks. And finally, it has been shown to produce syntactically simple hypotheses, thereby facilitating human comprehension of what it has learned.

Chapter 8

Additional Related Work

Chapter 2 described related research in the area of rule extraction. This chapter discusses other areas of research related to the work presented in this thesis. The first three sections in this chapter describe approaches that are designed to help understand the hypotheses represented by trained neural networks. The penultimate section in this chapter discusses the topic of active learning, and the final section covers algorithms that are related to the BBP method presented in Chapter 7.

8.1 Finite State Automata Extraction

A task closely related to rule extraction is the extraction of finite-state automata (FSA) from recurrent neural networks. As mentioned in Chapter 1, a recurrent network has links from a set of hidden or output units to a set of input units, and thus it is able 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 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 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 often trained on simple grammatical tasks that involve either

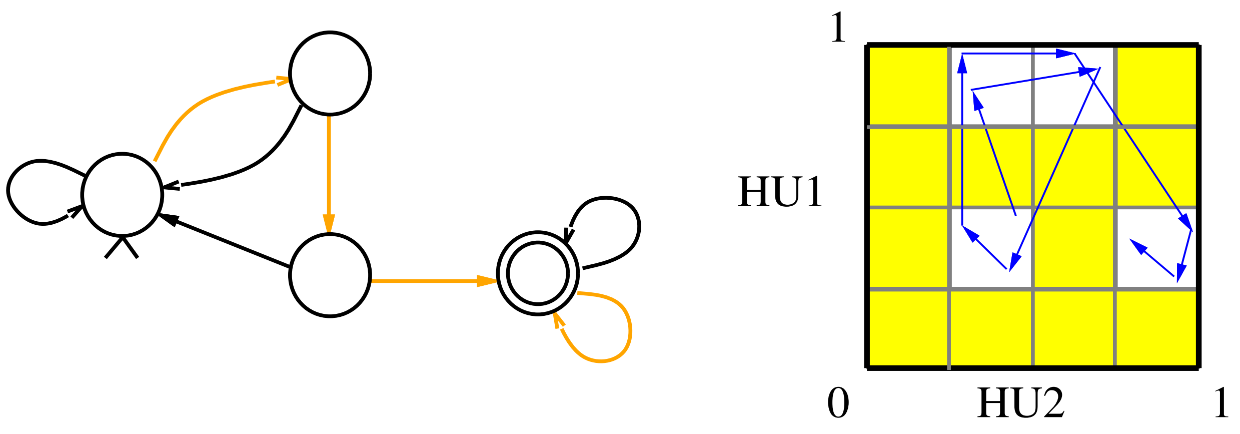


Figure 32: **The correspondence between an FSA and a recurrent network.** Depicted on the left is a finite-state automaton that implements a simple grammar. The two shades of arcs represent the two possible input symbols. Shown on the right is a two-dimensional, real-valued space defined by the activations of the two hidden units in a recurrent neural network. The path traced in the space illustrates the state changes of the hidden-unit activations as the network processes a sequence of inputs. The non-shaded regions of the space correspond to the states of the FSA.

recognizing strings of a target language, or predicting the next symbol in given strings. As a recurrent network processes such an input sequence, its hidden-unit activations trace a path in the s-dimensional hidden-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. A number of researchers have made this observation, and subsequently analyzed the behavior of trained recurrent networks as finite-state automata (Cleeremans et al., 1989; Pollack, 1991). This idea is illustrated in Figure 32, which shows a finite-state automaton and the two-dimensional hidden-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 hidden-unit activations as the network processes a sequence of inputs. The non-shaded regions of the space correspond to the states of the FSA.

Going a step further, several research groups have developed algorithms for extracting finite-state automata from trained recurrent networks (Giles et al., 1992; Watrous & Kuhn, 1992; Zeng et al., 1993; Das & Mozer, 1994). The key issue in such algorithms, is deciding how to partition the s-dimensional real-valued space into a set of discrete states. For example, the algorithm of Giles et al. partitions each hidden unit's activation range into q intervals of equal width, thus dividing the s-dimensional space into q^S partitions. Their method initially sets $q = 2$, but increases its value if it cannot extract an FSA that correctly describes the network's training set. Some FSA-extraction methods employ special training procedures that encourage the network to form discrete states during learning (Zeng et al., 1993; Das & Mozer, 1994). Although FSA-extraction algorithms have been able to extract accurate finite-state automata from networks trained on simple languages, they have not yet shown their usefulness for real-world applications.

8.2 Sensitivity and Hidden-Unit Analysis

Rule-extraction and FSA-extraction methods promote comprehensibility by translating the hypotheses represented by trained neural networks into a different, more perspicuous language. In contrast, there are several techniques for understanding neural-network hypotheses that do not provide complete descriptions of learned models, but instead provide local descriptions of some aspect of the neural models.

One such method that has been applied to neural networks is *sensitivity analysis* (Chatterjee & Hadi, 1988). The general idea of sensitivity analysis is to determine how the response of a variable (e.g., an output-unit activation) changes as the value of an input variable is modified. It is simple enough to measure and plot this relationship in a one or two-dimensional input space, but neural networks typically have high-dimensional input spaces. Thus, the key issue that arises in applying sensitivity analysis to neural networks is deciding at what points (i.e., instances) in a high-dimensional space to measure the variable of interest, and in what directions to perturb these points. Goh (1993) applied sensitivity analysis to characterize the influence of the features in neural networks trained for classification tasks. Goh describes three sensitivity-analysis methods that involve modifying the value for each feature in each training example while measuring the changes in output-unit activations. For example, one method characterizes the sensitivity of an output with respect to a particular feature by starting with a given training example, and varying the feature's value across its allowable range in fixed increments. For each tested value, the method records the absolute difference of the change in the value of the output unit. Goh's method runs this procedure on all of the training examples, and then plots the averaged results which show the sensitivity of the output as a function of the feature's value.

Pomerleau and Touretzky (1993) developed a method, called *hidden unit sensitivity analysis* that characterizes the role of individual hidden units in a network. Instead of describing the response of a particular output unit in terms of input activations, their method describes the response of the *correct* output unit as a function of a hidden unit's activation. Pomerleau and Touretzky investigated their technique using a network which takes images of roads as input, and produces as output steering directions for an autonomous driving vehicle. Given an image of a road, their sensitivity-analysis method systematically varies the position and the orientation of the road in the image while measuring the hidden-unit activation for each position-orientation pair. In order to understand the role of the hidden unit, they plot (i) its activation and (ii) its contribution to the correct output unit as a function of the road position-orientation. Although their network has a 960-dimensional input space (a 30×32 pixel "retina"), they are able to display the contribution of a hidden unit in a fairly natural and interpretable manner because their method uses only two degrees of freedom to vary an input image.

Gorman and Sejnowski (1988) also developed an approach for analyzing individual hidden units in a neural network. The first step in their method is to calculate, for each training example, the vector of activation signals that is sent to a given hidden unit. Each element in such a vector is simply the product of an input unit's activation and the weight from the input unit to the given hidden unit. Next, the method clusters these vectors and calculates the centroid of each cluster. These centroids can be thought of as prototypical instances. By displaying the prototypes and ordering them according to their resultant hidden-unit activations, the method aims to illustrate the types of input patterns to which the hidden unit responds. Like Pomerleau and Touretzky, Gorman and Sejnowski applied their method in a domain – classification of sonar signals – in which it is easy to visualize instances. The input features for an instance in this domain represent a time-slice of a sonar signal, and thus prototypes can be graphed as one-dimensional signals.

In contrast to the two previously discussed methods, which describe individual hidden units, several authors have investigated methods aimed at characterizing sets of hidden units (Sanger, 1989; Hanson & Burr, 1990; Dennis & Phillips, 1991; Elman, 1991). These methods are similar in spirit to the FSA-extraction algorithms in that they examine the space of hidden-unit activations, and try to identify regions in this space that are associated with particular network predictions. Whereas FSA-extraction algorithms analyze the s-dimensional activation space of the s state units, the methods considered here typically analyze the entire h-dimensional activation space of the h hidden units in a network. The premise of these techniques is that, by considering the similarities of examples that map to the same region of this space, the high-order features represented by the hidden units will be made apparent. Unlike the methods described earlier in this section, which analyze hidden units independently, these approaches recognize that the high-order features learned by networks may be represented by *patterns* of activation across their hidden units. The first step in these approaches is to classify the training data using the network, recording the vector of hidden-unit activations for each example. The next step is to apply a clustering method to these vectors in order to uncover structure in the patterns of hidden-unit activations. Hanson (1990) used a hierarchical-clustering method to identify clusters, while others have used principal-components analysis (Jolliffe, 1986) (and the related method of canonical-discriminant analysis) for this task (Dennis & Phillips, 1991; Elman, 1991). Sanger (1989) describes a slightly different approach that uses principal components analysis to identify patterns in hidden-unit *contributions*, which are the products of hidden-unit activations and the weights to an output unit.

8.3 Visualization Methods

A number of researchers have employed *scientific visualization* techniques to assist in understanding neural networks. Scientific visualization methods use graphical attributes such as color, size, and spatial organization to depict systems that are hard to understand because they involve a large number of elements or numerical parameters, or because they have time-varying behavior. Visualization techniques have been used to illustrate both the learning process and the prediction process in neural networks. Specifically, the following aspects of neural networks have been described using visualization methods:

- the weights and magnitudes of a network's connections (Hinton, 1986; Wejchert & Tesauro, 1989; Craven & Shavlik, 1992; Pomerleau & Touretzky, 1993),
- the decision boundaries formed by units in a network (Lang & Witbrock, 1988; Munro, 1991; Pratt et al., 1991),
- unit activations and the forward propagation of activation signals through the network (Craven & Shavlik, 1992),
- the sensitivity of hidden and output unit activations to smoothly varying inputs (Pomerleau & Touretzky, 1993),
- the backward propagation of error signals during learning (Craven & Shavlik, 1992),
- the trajectory of units in weight space during learning (Wejchert & Tesauro, 1989).

These visualization methods can sometimes provide insight into the learning and prediction behavior of a network. They seem to be especially effective in tasks where the input to the network is image data (e.g., Pomerleau & Touretzky, 1993). In such networks, various functions characterizing the network, such as the response of a hidden unit, can be readily visualized by projecting the function onto the network's input "retina" (i.e., its input units arranged into a planar image).

8.4 Active Learning

Another area of work related to the Trepan approach is the field of active learning. The term *active learning* refers to the situation in which a learner has some control over the training examples it receives. Trepan, for example, is an active learning algorithm in that it selectively draws instances and makes membership queries for these instances.

Under the rubric "learning with queries," active learning has been extensively investigated in the computational learning theory community. Angluin (1988) provides a taxonomy of queries, and elsewhere (Angluin, 1993) reviews learnability results for algorithms that use queries. These results are compelling because they demonstrate that some learning problems are efficiently learnable only if the learner is allowed to make queries. Hence, in some practical situations the ability to make queries should be a significant advantage. However, most of the algorithms developed to date in this community do not address the requirements of the rule-extraction task: specifically, learning comprehensible hypotheses in a language that applies to a broad range of tasks As discussed in Chapter 5, however, Bshouty's (1993) algorithm for learning decision trees using membership queries may have some practical implications for rule extraction.

Active learning has also been investigated in practical settings, where it potentially enables a learner to induce better models with fewer training instances. Cohn et al. (1996) provide a nice summary of the types of heuristics that have been used by practical active-learning algorithms to select training examples. These heuristics include selecting data where the learner:

- does not yet have any data,
- has low confidence in its model,
- expects to change its model,
- previously found data that resulted in changes to its model.

The heuristic used by Trepan is related to the first three of these: Trepan selects data in the part of the instance space in which it is currently refining its model.

In practical settings, active learning has gained the most attention in reinforcement-learning problems (e.g., Thrun & Moeller, 1992). Active learning is a natural aspect of reinforcement learning since, in these problems, the learner can influence its environment, thereby exerting some control over the states (i.e., training examples) it experiences. Practical algorithms for active learning have also been developed for regression tasks (e.g., MacKay, 1992; Cohn et al., 1996), and classification tasks (e.g., Baum & Lang, 1991; Cohn et al., 1994). The classification algorithms, however, are for learning in neural networks and thus they do not produce comprehensible hypotheses like Trepan.

Catlett (1992) and others (Musick et al., 1993) have investigated an issue related to active learning: selecting samples for decision-tree induction when there is a wealth of training data. The primary motivation for this work is to speed up learning by processing only as much training data as is needed to make good decisions when inducing trees. For example, one method developed in this line of work frames the question of when to select one splitting test over another as a decision-theory problem (Musick et al., 1993). This method estimates how much training data is needed to confidently decide that the information gain of one test is greater than the information gain of other candidate tests, and uses such estimates to select samples of appropriate size for making these decisions. Incorporating ideas such as this into Trepan would be an interesting topic for future research.

8.5 Work Related to the BBP Algorithm

Chapter 7 presented the Boosting-Based Perceptron (BBP) algorithm which constructively learns simple neural networks. As stated earlier, the BBP algorithm is similar to C4.5 and other decision-tree algorithms which learn by iteratively selecting features and incorporating them into their current hypotheses. The BBP algorithm has a much different inductive bias, however, since its hypotheses are perceptrons instead of decision trees. BBP is also similar to feature-selection algorithms (Almuallim & Dietterich, 1991; Kira & Rendell, 1992a; Caruana & Freitag, 1994; John et al., 1994; Moore & Lee, 1994; Skalak, 1994; Cherkauer & Shavlik, 1996) which attempt to find small feature subsets that are adequate for learning. Feature-selection methods, however, are not able to form hypotheses themselves, but instead must be used in conjunction with ordinary learning methods.

Several constructive neural-network approaches have been previously developed (Ash, 1989; Fahlman & Lebiere, 1989; Frean, 1990). Similarly, there are several algorithms that simplify learned networks by removing weights or hidden units (Le Cun et al., 1989; Mozer & Smolensky, 1988). Unlike BBP, however, these methods are not designed to learn comprehensible hypotheses, but instead are intended primarily to avoid over-fitting. Thus, the networks produced by these methods may still embody hard-to-understand feature interactions and nonlinear relationships between inputs and outputs.

The previous work most closely related to BBP is a constructive algorithm for learning polynomial approximations (Sutton & Matheus, 1991; Sanger et al., 1992). Like BBP, Sutton et al.'s algorithm incrementally builds one-layer networks. Unlike BBP, however, their algorithm is designed for regression problems instead of classification tasks. The terms in their networks are real-valued input variables and products of input variables. The output units in their networks do not perform a thresholding function, but instead output a linear combination of the inputs. A final difference is that, unlike BBP, their algorithm is not based on a hypothesis-boosting method.

Finally, other groups have used hypothesis-boosting methods to improve the performance of classifiers in applied settings (Drucker et al., 1993; Drucker & Cortes, 1996; Freund & Schapire, 1996; Quinlan, 1996). In most of these efforts, however, the "weak" learning algorithms being boosted were fairly strong learners, such as C4.5 or multi-layer neural networks, and therefore the hypotheses produced were quite complex. The exception to this trend is the recent study by Freund and Schapire (1996), which used the AdaBoost algorithm to boost a variety of weak-hypothesis types, including one-level decision trees and conjunctive rules. Still, these hypotheses are generally not as simple as those induced by the BBP algorithm, which is specifically designed to learn comprehensible hypotheses.

8.6 Chapter Summary

This chapter discussed three lines of research related to the work presented in this thesis: methods designed to aid in understanding trained neural networks, algorithms for active learning, and methods related to the BBP algorithm presented in Chapter 7.

The approaches to understanding neural-network hypotheses discussed in this chapter include FSA-extraction, sensitivity and hidden-unit analysis, and visualization methods. The task of extracting finite-state automata from recurrent neural networks is analogous to the task of rule extraction from non-recurrent networks in that learned hypotheses are translated into languages that are easier to understand. Although FSA-extraction algorithms have been investigated in depth, they have not found successful application to real-world problems.

A variety of methods have been developed to describe how the activation of a particular unit or set of units in a neural network responds to various types of input patterns. These methods can provide insight into how a network represents its solution, or how sensitive its predictions are to certain features. Unlike rule-extraction and FSA-extraction algorithms, however, these methods do not provide complete descriptions of learned models. They provide only local descriptions of some aspect of a trained network.

Active learning refers to the situation in which a learner is able to select its own training examples. Active learning has been investigated in both theoretical and practical contexts. Unlike Trepan, however, the existing suite of active-learning algorithms do not address the unique requirements of the rule-extraction task: learning comprehensible hypotheses in a language that applies to a broad range of tasks.

There is a broad set of approaches related to the BBP algorithm, including methods for feature selection, constructive neural-network learning, and practical hypothesis-boosting. The BBP algorithm has a valuable niche among these techniques, however, in that produces comprehensible hypotheses, has formal learnability guarantees for an interesting class of concepts, and has demonstrated good predictive performance in a variety of classification tasks.

Chapter 9

Conclusions

Neural networks are one of the most widely used approaches to inductive learning. They have been applied to classification, regression, and reinforcement learning tasks, and they have demonstrated good predictive performance in a wide variety of interesting problem domains. They suffer from a significant limitation, however, in that their learned hypotheses are usually incomprehensible. To address this limitation, a number of research groups have developed techniques for rule extraction. Rule extraction involves approximating the function represented by a trained network in a language, such as symbolic inference rules, that better facilitates comprehensibility. The primary focus of this thesis has been the development of a rule-extraction method, called TREPAN, that overcomes the significant limitations of previous algorithms.

A secondary focus of this thesis has been the development of an algorithm, called BBP, that constructively induces simple neural networks. The motivation underlying this algorithm is similar to the motivation behind TREPAN: to learn comprehensible models in problem domains in which neural networks have an especially appropriate inductive bias.

In this concluding chapter, I discuss the contributions and limitations of the research presented in this thesis, and propose several future research tasks aimed at addressing the limitations.

9.1 Contributions

This thesis has made several contributions to the state of the art in learning comprehensible models via neural networks. The following sections discuss these contributions in some detail.

A New Approach to the Rule-Extraction Task

One significant contribution of this thesis is a novel approach to the task of rule extraction. The approach pioneered in this research is to view the rule-extraction problem as an inductive learning task. A significant advantage of this approach is that it is widely applicable. Unlike many rule-extraction algorithms, which require special network architectures or training methods, the learning-based approach to rule extraction naturally extends to a broad class of learned models. A second significant advantage of this approach is that it is scalable to large networks and to problem domains with large feature spaces.

The Trepan Algorithm

Another major contribution of the thesis is the Trepan algorithm, which is an instantiation of the learning-based approach to rule extraction described above. Given a learned model, such as a trained neural network, Trepan induces a decision tree that describes the function represented by the model. Since Trepan interacts with a given model only through membership queries, it can be applied to a wide range of hard-to-understand models. Moreover, because Trepan represents its hypotheses as decision trees, and grows its trees in a best-first manner, it offers the user a fine level of control over the complexity of extracted trees. Trepan is also scalable to large networks and problem domains: the computational complexity of expanding a node in Trepan is polynomial in the number of features in the domain, and in the sample size used (i.e., the number of instances used to select tests and determine class labels at tree nodes).

Extensive Empirical Evaluation of a Rule-Extraction Method

In the experiments reported in Chapter 4, I applied Trepan to neural networks in eight problem domains. These experiments were designed to evaluate Trepan along the dimensions of comprehensibility, fidelity, generality, and scalability.

The experiments demonstrated the generality and scalability of Trepan by using the algorithm to extract decision trees in a wide variety of problem domains, including classification, regression, and reinforcement-learning tasks. These domains involved a wide range of feature types, network architectures, network sizes, and training-set sizes.

The fidelity of the trees extracted by Trepan was measured by comparing the predictions made by each tree to its counterpart network. In all of the domains, the extracted trees exhibited a high level of fidelity to their corresponding networks.

The experiments evaluated the comprehensibility of extracted trees by measuring their syntactic complexity. In the supervised-learning domains, the trees extracted by Trepan were compared to trees induced directly from the training data by two conventional decision-tree algorithms. In general, the trees extracted by Trepan were of comparable complexity to the trees learned by the ordinary decision-tree methods, and in some cases they were significantly less complex. Additionally, the Trepan trees had higher levels of predictive accuracy than their counterparts induced directly from the training data by the ordinary decision-tree algorithms.

This is the most extensive empirical evaluation of a rule-extraction method reported to date. Importantly, in three of the problem domains (telephone loop diagnosis, exchange-rate prediction, and elevator scheduling) the networks were not trained solely for the sake of testing a rule-extraction method, and in the two latter domains, the networks were trained by others without any forethought of later applying a rule-extraction method to them. To the best of my knowledge, this is the first case of a rule-extraction method being applied to independently developed neural networks.

Application of a Rule-Extraction Method to Ensembles

Although the historical context of this thesis is in the task of extracting rules from neural networks, the Trepan algorithm is general enough that it can be applied to nearly any learned model that uses a feature-value input representation. In fact, in one of the classification domains considered in Chapter 4, Trepan was used to extract decision trees from *ensembles* of neural networks. This is the first case of a rule-extraction method being applied to a learned model that is not simply an individual neural network, and it is one of the few cases of a learning method being used to induce a simple description of a more complicated model.

The BBP Algorithm

A secondary focus of this thesis was the development of an algorithm that learns simple, comprehensible neural networks. The Boosting-Based Perceptron (BBP) learning algorithm, which is based on a hypothesis-boosting method, learns simple networks by constructively adding weighted connections to a perceptron over individual features and high-order features.

The BBP algorithm provides an appealing combination of strengths. First, it provides formal learnability guarantees for a fairly natural class of target functions. Specifically, it is able to PAC-learn the class of sparse perceptrons. Second, it has demonstrated good predictive accuracy in a variety of problem domains. Along this dimension, BBP was competitive with the best of the algorithms in the empirical evaluation presented in Chapter 7. And finally, it has been shown to produce syntactically simple hypothesis, thereby facilitating human comprehension of what it has learned.

Enhancements to the ID2-of-3 Algorithm

A minor contribution of this thesis is that an improved version of the ID2-of-3 algorithm (Murphy & Pazzani, 1991) was developed. ID2-of-3 is a decision-tree induction method that uses m-of-n tests at internal nodes in the trees it learns. In course of investigating Trepan, I developed a version of ID2-of-3 that incorporates five enhancements not included in the original algorithm: the application of a χ^2 test, literal pruning, and a beam search when constructing m-of-n tests, the use of C4.5's pruning method (Quinlan, 1993) after tree induction, and the generalization of m-of-n tests to real-valued features. The first four enhancements were empirically evaluated in Chapter 4 and found to be of value in improving the concision and predictive accuracy of induced trees. The fifth capability clearly enhances the applicability of the algorithm.

A Fast Wrapper-Based Method for Feature Selection

Another minor contribution of this thesis is a fast wrapper-based method for feature selection. Feature selection methods, used in concert with inductive learning algorithms, are often able to improve the predictive accuracy of learned models. Broadly speaking, there are two types of feature-selection algorithms: wrapper methods and filter methods. Whereas wrapper methods exploit the inductive bias of the learning method when selecting features, they often require a large, expensive search. Filter methods, on the other hand, are often fast but do not take advantage of the learning method's inductive bias when selecting feature subsets. In Chapter 7, I presented a hybrid filter-wrapper method based on the Relief algorithm (Kira & Rendell, 1992b). Relief is a filter method that determines a total ordering over the domain features. My method employs a wrapper procedure to evaluate the possible subsets defined by this ordering. The method is especially fast because it does not consider every allowable subset, but instead uses a golden section search to quickly find a good one.

9.2 Limitations of Trepan and Future Work

The primary limitations of Trepan are threefold: (i) it cannot describe the real-valued predictions of models trained to perform regression tasks; (ii) it sometimes fails to find trees that are concise yet exhibit a high level of fidelity to their target models; (iii) it does not have strong theoretical guarantees regarding the amount of computational effort required to extract a tree with a specified level of fidelity. The proposed research tasks described below are designed to address these limitations.

Regression Trees

The rule-extraction work in this thesis has focused on understanding learned models in classification domains. Although Chapter 4 presented experiments illustrating that Trepan can be profitably applied in regression and reinforcement-learning domains, Trepan's approach to these problems was to represent the networks' hypotheses using decision (i.e., classification) trees. For some regression tasks, however, it would be desirable to have the extracted model represent the actual regression surface of the network. One way to do this would be to extend Trepan so that it can learn *regression trees* in addition to classification trees. A regression tree is a tree-structured model like a decision tree, except that its leaves are characterized by real-valued functions as opposed to predicted classes. The CART algorithm (Breiman et al., 1984), for example, learns regression trees in which a linear function is associated with each leaf.

Beyond Trees

The general approach underlying the Trepan algorithm is to view rule extraction as a learning task. As I have argued, this approach provides a very general rule-extraction method that scales well to large problems. This thesis investigated in detail the Trepan algorithm, which is a particular instantiation of this approach designed to extract decision trees. Trepan, like all other learning algorithms, has a particular inductive bias, and this bias might not be well suited to expressing some concepts. Therefore, a natural extension of the approach championed in this thesis is to develop extraction algorithms, similar in spirit to Trepan, which use languages other than decision trees to represent their hypotheses. The motivation here is to develop a suite of complementary extraction algorithms, each of which is well suited to a particular class of problems. The key research tasks in this endeavor will be to identify suitable hypothesis languages (which, importantly, promote comprehensibility) and to adapt learning algorithms for such languages to the special requirements of the rule-extraction task. One hypothesis representation that would be a good candidate to consider is the language of decision lists (Rivest, 1987). Algorithms for inducing decision lists are interesting in that they employ a different type of learning strategy, and hence have a different inductive bias, than decision-tree learning methods. Whereas decision-tree learning methods use a divide-and-conquer approach, decision-list algorithms employ a covering (i.e., separate-and-conquer) strategy.

Exploiting Other Sources of Inductive Bias

Often performance on a learning task can be improved by changing the inductive bias of the algorithm in order to exploit available knowledge about the problem domain. The TREPAN algorithm has a fairly weak inductive bias in that it considers a rich hypothesis space and uses training examples and membership queries as its only source of information about the target concept. An interesting line of inquiry for future research would be to investigate what sources of domain-specific bias can be exploited in order to improve the solutions produced by TREPAN.

Broadly, there are two types of information that TREPAN could use to guide its inductive process: background knowledge and model knowledge. I use the term *model knowledge* to refer to information that can be acquired by inspecting the learned model that TREPAN is trying to describe. For example, the given model might be used to suggest additional, higher-level features that TREPAN could use in its description of the model. A number of studies in the conventional learning setting have shown that the predictive performance of a learner can often be improved by constructing new features (e.g., Matheus, 1990). Such features might be discovered either by directly inspecting the structure of the model (i.e., the parameters and topology of a neural network), or by using membership queries to uncover feature interactions. For example, membership queries could be used to determine if a network's predictions were statistically dependent on particular feature expressions.

A related issue is how best to apply TREPAN to knowledge-based neural networks (Towell & Shavlik, 1994). Recall that a knowledge-based network has its topology and initial weights specified by a formal specification of available background knowledge (i.e., a domain theory). TREPAN currently treats a given model as a black box, interacting with it only via membership queries. In the case of a trained knowledge-based neural network, however, much of the network's structure may directly correspond to meaningful symbolic rules. In such cases, a rule-extraction algorithm might find better solutions using a local approach of extracting rules to describe the behavior of specific units in the network. This was the approach taken by Towell and Shavlik (1993) in their work on extracting rules from knowledge-based networks. In many cases, however, much of the structure of knowledge-based networks after training is as opaque as ordinary networks (Opitz, 1995), and thus a hybrid local-global approach might be most appropriate.

Domain-Specific Instance Models

A key aspect of Trepan is that it constructs models of the underlying data distribution in order to generate instances for membership queries. The models used in the experiments reported in the thesis were simple and domain independent. Although this approach works well for many tasks, in some problem domains there is evidence that such simple models do not adequately represent the underlying distribution (e.g., the `telephone` domain considered in Chapter 4). I hypothesize that, in some domains, Trepan could produce better hypotheses by exploiting domain knowledge to construct more accurate models of the data distribution. For example, in the `telephone` domain there are several known dependencies among features that are not adequately captured by Trepan's instance models. Such information about feature dependencies could be leveraged by using a Bayesian network, for example, as the instance model. The general research topic I advocate here is to explore the utility of more sophisticated instance models, and to extend the Trepan algorithm to handle such models. For example, to make queries to a Bayesian-network instance model Trepan would have to be integrated with an algorithm for inference in Bayesian networks (Pearl, 1988) to ensure that it sampled properly from the distribution.

Better Theoretical Guarantees

Chapter 5 argued that the scalability of rule-extraction methods, and Trepan in particular, should be analyzed in the context of formal models of learning. Such frameworks can be used to determine how much computational effort is required to extract a hypothesis that has a specified level of fidelity to its target model. As discussed in Chapter 5, it is still an open research issue as to which model of learnability provides the most appropriate context for evaluating rule-extraction algorithms such as Trepan. Moreover, another open issue is under what minimally restrictive set of assumptions can Trepan, or some suitably-modified version of the algorithm, be shown to be efficiently learnable. In addition to establishing learnability results for algorithms such as Trepan, another goal of this line of inquiry is to use insight gained from such analyses to improve rule-extraction algorithms, as has happened with a recent analysis of decision-tree induction algorithms (Kearns & Mansour, 1996).

Simplification of Extracted Models

Often the comprehensibility of a tree extracted by Trepan can be improved by a few truth-preserving modifications. One area for future research is to develop an algorithm that automatically simplifies extracted trees by performing such operations. One approach to this task would be to frame it as a heuristic search problem in which operators are used to transform given trees, and an evaluation function guides the transformations by measuring the syntactic complexity of resulting trees. Such a method might include operators for reordering tests in trees, expanding out m-of-n expressions, or even converting trees into rules. C4.5's rule-generation algorithm addresses this task to some extent. However, C4.5's method is not truth-preserving, and its task is simpler because C4.5 trees do not use m-of-n tests.

9.3 Limitations of BBP and Future Work

The BBP algorithm presented in Chapter 7 has several limitations that I plan to address in future work. Perhaps foremost among these limitations is that, to include conjunctive terms in its hypotheses, BBP exhaustively searches the set of conjunctions of up to k features. This approach does not scale well in the size of conjunctions considered, since learning time depends exponentially on k. I plan to investigate modifications to the algorithm that would enable a more efficient exploration of higher-order terms. One approach I plan to investigate is to use a heuristic-search strategy – like the one used to build m-of-n tests in Trepan– to construct higher-order terms on the fly.

A second significant limitation of the BBP algorithm is that it currently handles only discrete-valued features. An important task for future work is to generalize the algorithm to be applicable to problems with real-valued features. The AdaBoost algorithm (Freund & Schapire, 1995) can be applied to domains with real-valued features, so the task would involve generalizing the notion of high-ordered functions over the inputs, and the notion of correlation of a hypothesis with the target function.

Another limitation of BBP, in comparison to Winnow (Littlestone, 1989), is that it is a batch algorithm (i.e., it looks at all of its training data before updating its hypothesis). Although the capability was not exploited in the experiments, one of the appealing aspects of Winnow is that it is well suited for tasks that involve on-line learning. Thus, an interesting task for future research would be to modify BBP for the on-line setting as well.

9.4 Final Remarks

Although the idea of rule extraction has been around for a number of years, it is receiving increasing attention from researchers in machine learning, and from specialists applying neural networks and similarly opaque models in a wide variety of application domains. With the rapid proliferation of data-mining projects and fielded learning systems, the comprehensibility of learned models is increasingly a primary concern in the application of inductive methods. I believe that the ideas presented in this thesis have demonstrated significant and measurable advancement of the state of the art in gaining comprehensible models via neural-network learning. There are still many difficult, open research issues in the field, however, and I hope that this work will help point the way to solutions for many of them.

Appendix A

Representative Trees Extracted by Trepan

This appendix depicts representative trees extracted by Trepan in seven of the eight domains studied in this thesis. A tree for the `telephone` problem is not shown because the domain is proprietary. In cases where multiple trees were extracted for a given domain, I show trees that are typical, in terms of their syntactic complexity, of the trees extracted in the reported experiments.

A.1 The `Coding` Domain

The features in the `coding` domain (Craven & Shavlik, 1993b) represent the 64 *codons* (three-letter words) that can be formed from the DNA bases: {`a, c, g, t`}. Each Boolean feature indicates the presence or absence of a given codon "in-frame" in the sequence being classified. In the figure, negated features are preceded by a minus sign. The class labels indicate whether a given, fixed-length sequence is predicted to encode a protein or not.

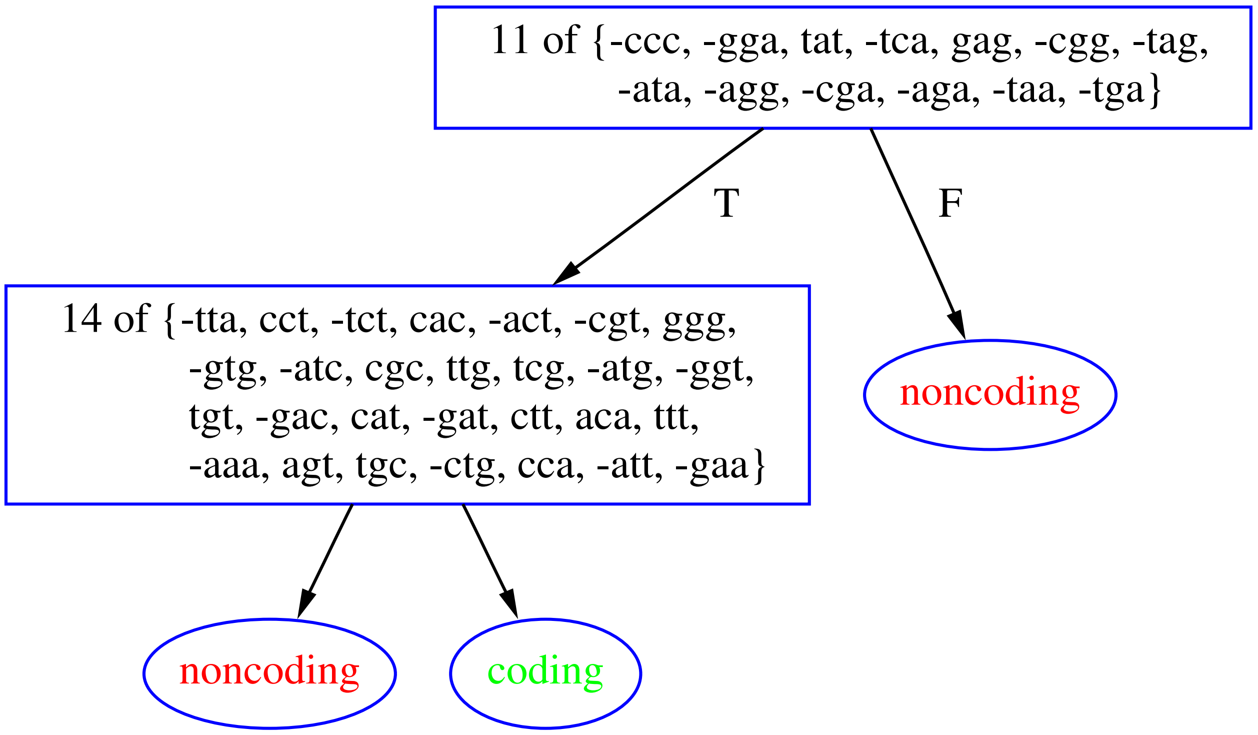

11 of {-ccc, -gga, tat, -tca, gag, -cgg, -tag, -ata, -agg, -cga, -aga, -taa, -tga}
T
F
14 of {-tta, cct, -tct, cac, -act, -cgt, ggg, -gtg, -atc, cgc, ttg, tcg, -atg, -ggt, tgt, -gac, cat, -gat, ctt, aca, ttt, -aaa, agt, tgc, -ctg, cca, -att, -gaa}
noncoding
noncoding
coding

Figure 33: **A representative tree for the `coding` domain.**

A.2 The `Heart` Domain

The 13 features in the `heart` domain (Detrano et al., 1989) represent measured descriptions of the patient under consideration. The table below lists the abbreviations used in the figure, and provides a lengthier description of each feature. The possible values for each feature are also listed. The class labels indicate whether or not a given patient is diagnosed as having heart disease.

abbreviation	description
`ca`	number of major vessels colored by fluoroscopy (`0, 1, 2, 3`)
`cp`	chest pain type (`typical_angina, atypical_angina, non_anginal, asymptomatic`)
`exang`	exercise-induced angina (`yes, no`)
`fbs`	fasting blood sugar $>$ 120mg/dl (`yes, no`)
`restecg`	resting electrocardiographic results (`normal, wave_abnormality, hypertrophy`)
`sex`	`male` or `female`
`slope`	slope of the peak exercise ST segment (`upsloping, flat, downsloping`)
`thal`	exercise test (`normal, fixed_defect, reversible_defect`)

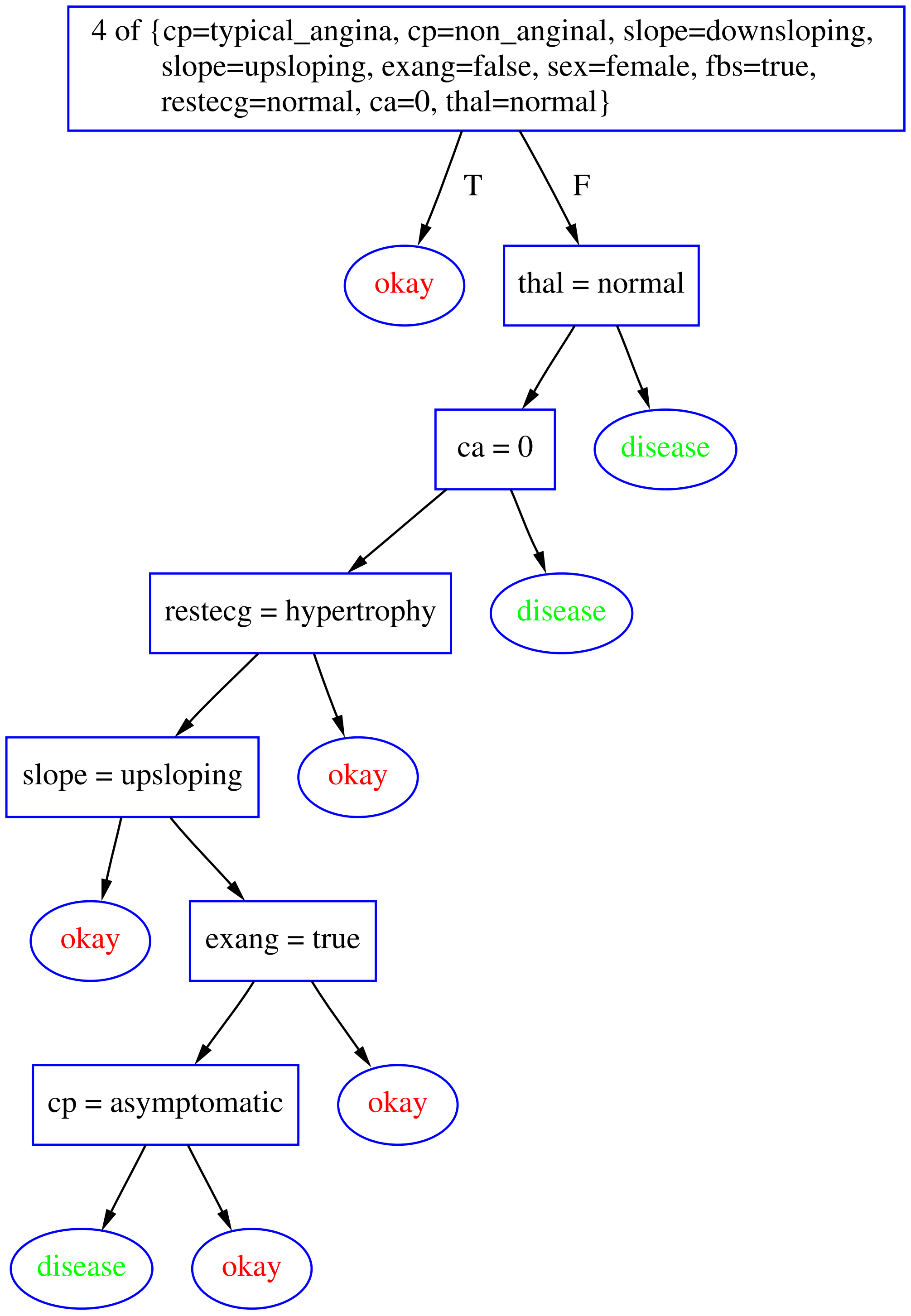

4 of {cp=typical_angina, cp=non_anginal, slope=downsloping, slope=upsloping, exang=false, sex=female, fbs=true, restecg=normal, ca=0, thal=normal}
T
F
okay
thal = normal
ca = 0
disease
restecg = hypertrophy
disease
slope = upsloping
okay
okay
exang = true
cp = asymptomatic
okay
disease
okay

Figure 34: **A representative tree for the `heart` domain.**

A.3 The `NETtalk` Domain

The seven features in the `NETtalk` domain (Sejnowski & Rosenberg, 1987) represent letters in a seven-letter window of English text. The features are named `p1, p2, p3, p4, p5, p6` and `p7`, indicating their position in the window. Each feature can either take on one of the 26 letters in the English alphabet, or it can have the value `dash` indicating the absence of a letter in the given position. The class labels indicate the stress assigned to the letter in position `p4` (i.e., the middle letter). Consonants are generally assigned a stress of either `left` or `right`, indicating whether the principal vowel in the syllable is to the left or right of the consonant. Vowels are generally assigned a stress of `pri`, `sec`, or `none`, denoting whether the vowel receives primary, secondary, or no stress, respectively.

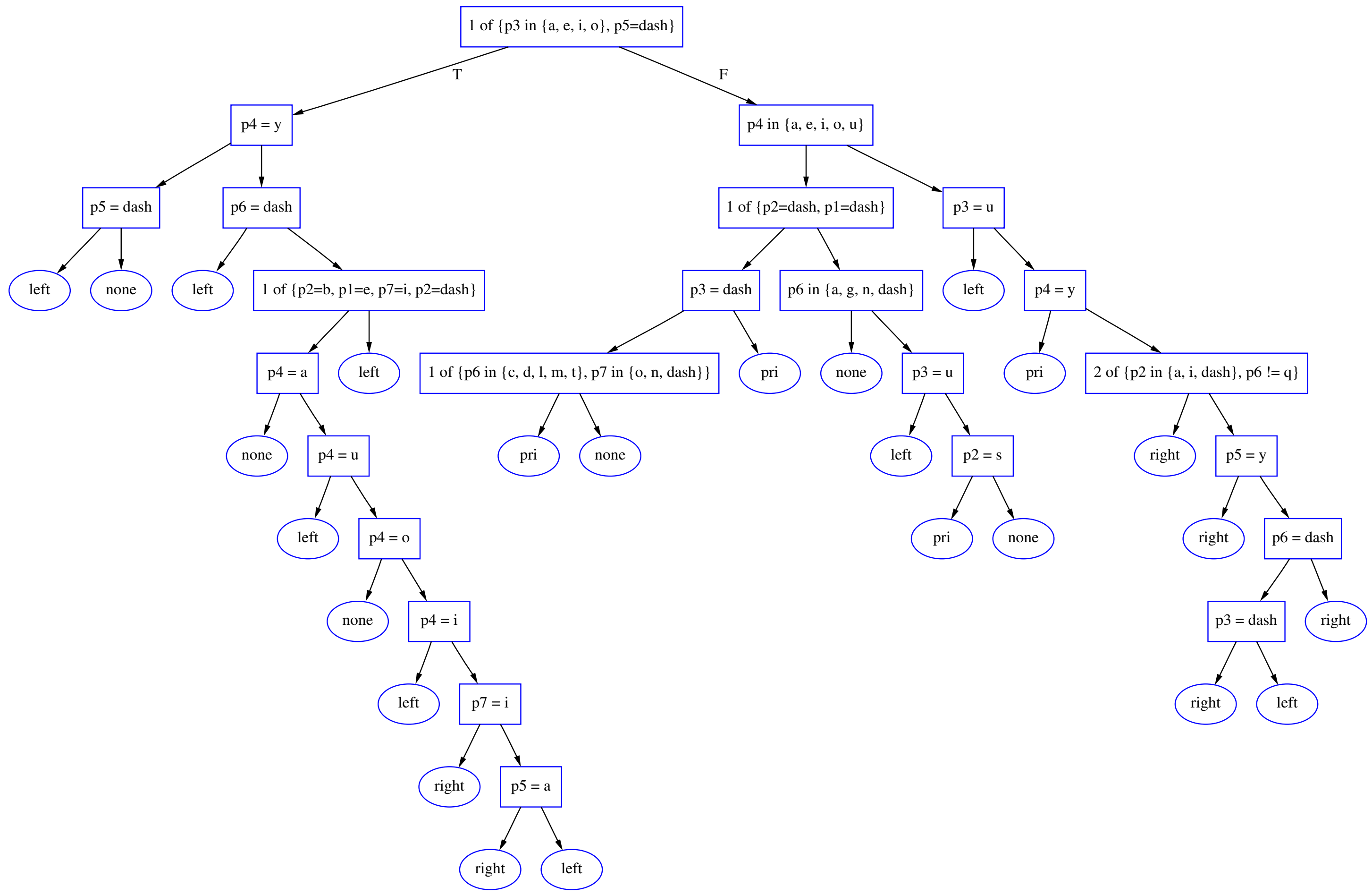


Figure 35: **A representative tree for the NETtalk domain.**

A.4 The `Promoter` Domain

The 57 features in the `promoter` domain (Towell et al., 1990) represent the DNA bases in a 57-base long sequence. Each feature is named pX, where X ranges from -50 to +7 (skipping 0) and indicates the position of the feature in the sequence. The value of each feature indicates the base that occurs at that position. The allowable values are: { `a`, `c`, `g`, `t`}. The class labels indicate whether a given sequence is predicted to be a promoter or not.

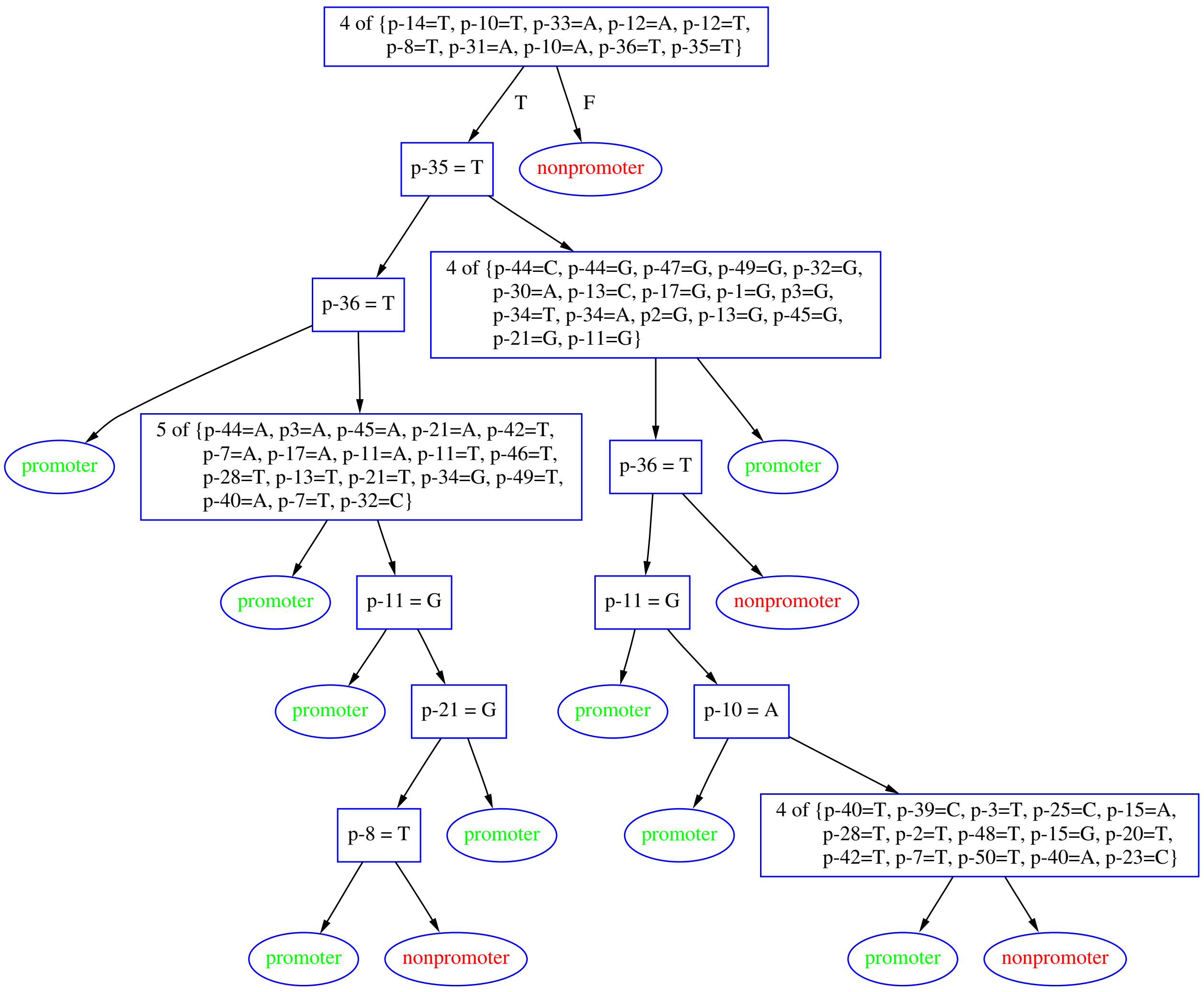

4 of {p-14=T, p-10=T, p-33=A, p-12=A, p-12=T, p-8=T, p-31=A, p-10=A, p-36=T, p-35=T}
T
F
p-35 = T
nonpromoter
p-36 = T
4 of {p-44=C, p-44=G, p-47=G, p-49=G, p-32=G, p-30=A, p-13=C, p-17=G, p-1=G, p3=G, p-34=T, p-34=A, p2=G, p-13=G, p-45=G, p-21=G, p-11=G}
promoter
5 of {p-44=A, p3=A, p-45=A, p-21=A, p-42=T, p-7=A, p-17=A, p-11=A, p-11=T, p-46=T, p-28=T, p-13=T, p-21=T, p-34=G, p-49=T, p-40=A, p-7=T, p-32=C}
p-36 = T
promoter
promoter
p-11 = G
p-11 = G
nonpromoter
promoter
p-21 = G
promoter
p-10 = A
p-8 = T
promoter
promoter
4 of {p-40=T, p-39=C, p-3=T, p-25=C, p-15=A, p-28=T, p-2=T, p-48=T, p-15=G, p-20=T, p-42=T, p-7=T, p-50=T, p-40=A, p-23=C}
promoter
nonpromoter
promoter
nonpromoter

Figure 36: **A representative tree for the `promoter` domain.**

A.5 The `Voting` Domain

The features in the `voting` domain (Schlimmer & Fisher, 1986) represent votes made by members of the U. S. House of Representatives in 1984. The value of each feature indicates whether a given representative voted yes, no or did not vote. The table below lists the abbreviations used in the figure, and provides a lengthier description of each feature. The class labels indicate whether a representative is predicted to be a member of the Democratic or Republican party.

abbreviation	description
`budget`	adoption of the budget resolution
`contras`	aid to Nicaraguan Contras
`crime`	crime
`education`	education spending
`el_salvador`	El Salvador aid
`exports`	duty free exports
`immigration`	immigration
`mx_missile`	MX missile
`synfuels`	Synfuels corporation cutback
`religious`	religious groups in schools
`satellite`	anti-satellite test ban

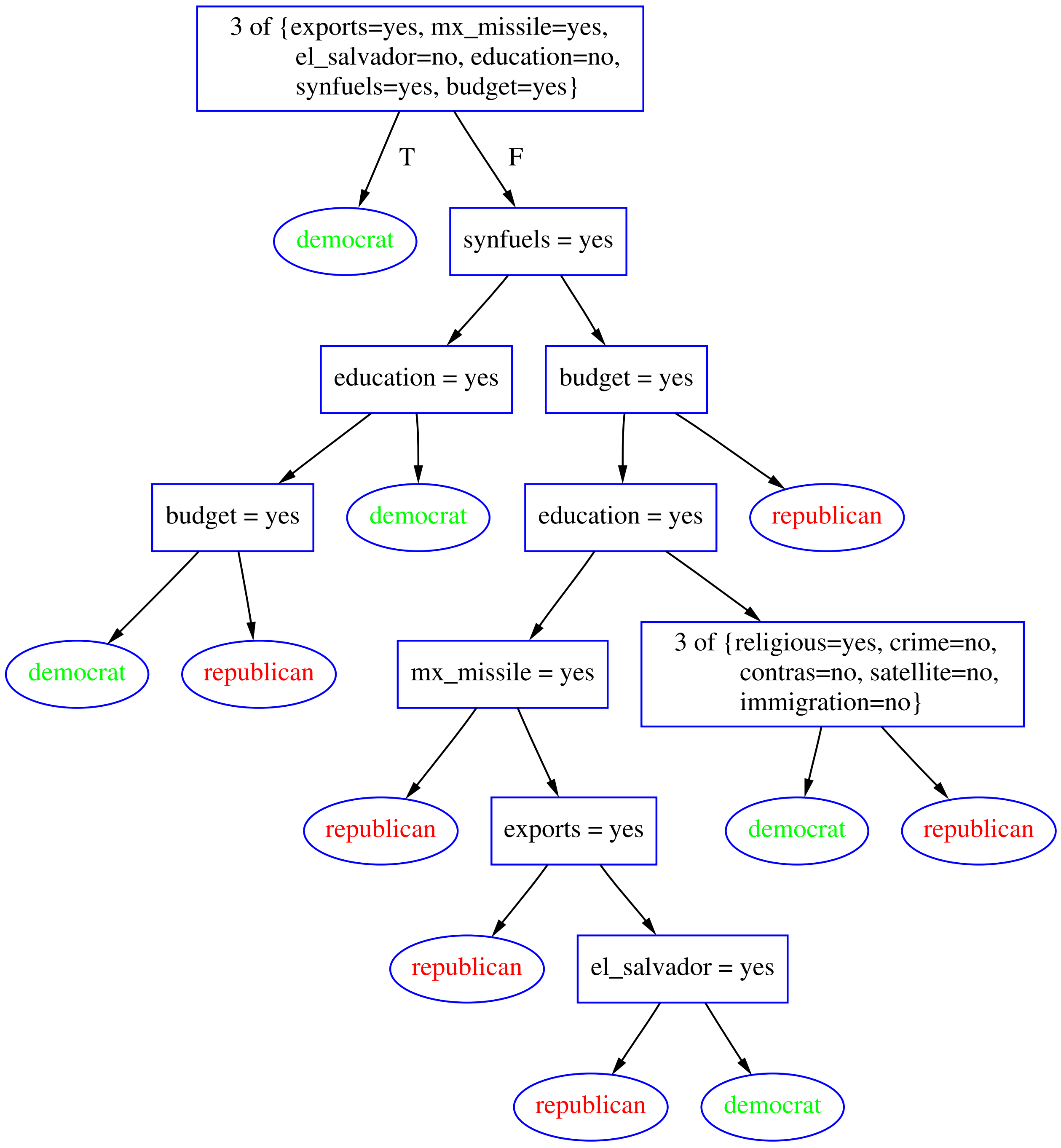


Figure 37: **A representative tree for the `voting` domain.**

A.6 The Exchange-Rate Domain

The following features from the exchange-rate domain (Weigend et al., 1995) are incorporated into the depicted tree. There are sets of features in this domain that involve multiple functions computed on the same underlying indicator; these features are indicated by subscripts in the table and the tree. For example, Dem_USD_ex[3] and Dem_USD_ex[8] represent different functions of the Mark-Dollar exchange rate.

abbreviation	description
Dem_future[5]	A function of Deutsche Mark futures.
Dem_USD_ex[3] Dem_USD_ex[8]	Functions of the Mark-Dollar exchange rate.
interest_rate[1]	A comparison of the Mark-Dollar exchange rate to the German interest rate.
ROI[13]	A comparison of the return on investment (ROI) of investing Marks in the U.S. stock market versus investing Marks in the French stock market.
vs_Dem_future[2] vs_Dem_future[3]	Measures of the Mark-Dollar exchange rate versus Mark futures.
vs_SFr_future	A measure of the Swiss Franc-Dollar exchange rate versus Swiss Franc futures.
vs_Yen_future	A measure of the Yen-Dollar exchange rate versus Yen futures.

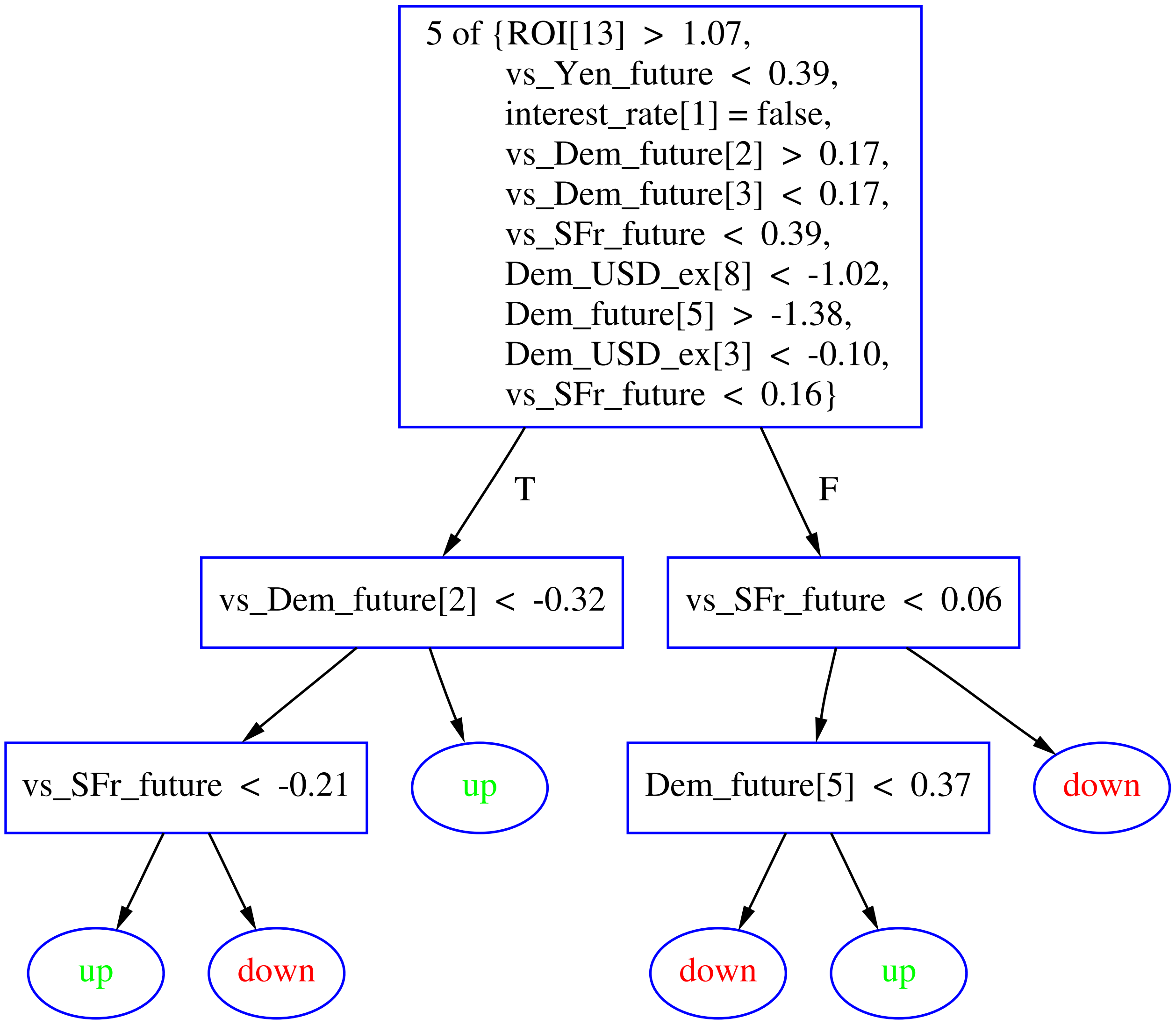


Figure 38: **A representative tree for the `exchange-rate` domain.**

A.7 The `Elevator-Control` Domain

The following features from the `elevator-control` domain (Crites & Barto, 1996) are incorporated into the depicted tree. The class labels indicate whether the elevator car should continue in its current direction or stop at the next floor.

abbreviation	description
`footprint_`X	These features correspond to the ten floors where the other elevator cars may be located. Each car has a "footprint" that depends on its direction and speed. For example, the footprint of a stopped car is localized to one floor, whereas the footprint of a moving car is spread across the floors it is approaching. The feature `footprint_`X represents the cumulative footprints of the other cars for floor X.
`floor`X`_pushed_time`	Time elapsed since the hall button at floor X was pushed.
`highest_queue`	Is the elevator at the highest floor with a waiting passenger?
`location_direction`	Describes the position and the direction of the elevator. For example `2u` indicates that the elevator is headed for the second floor going up.

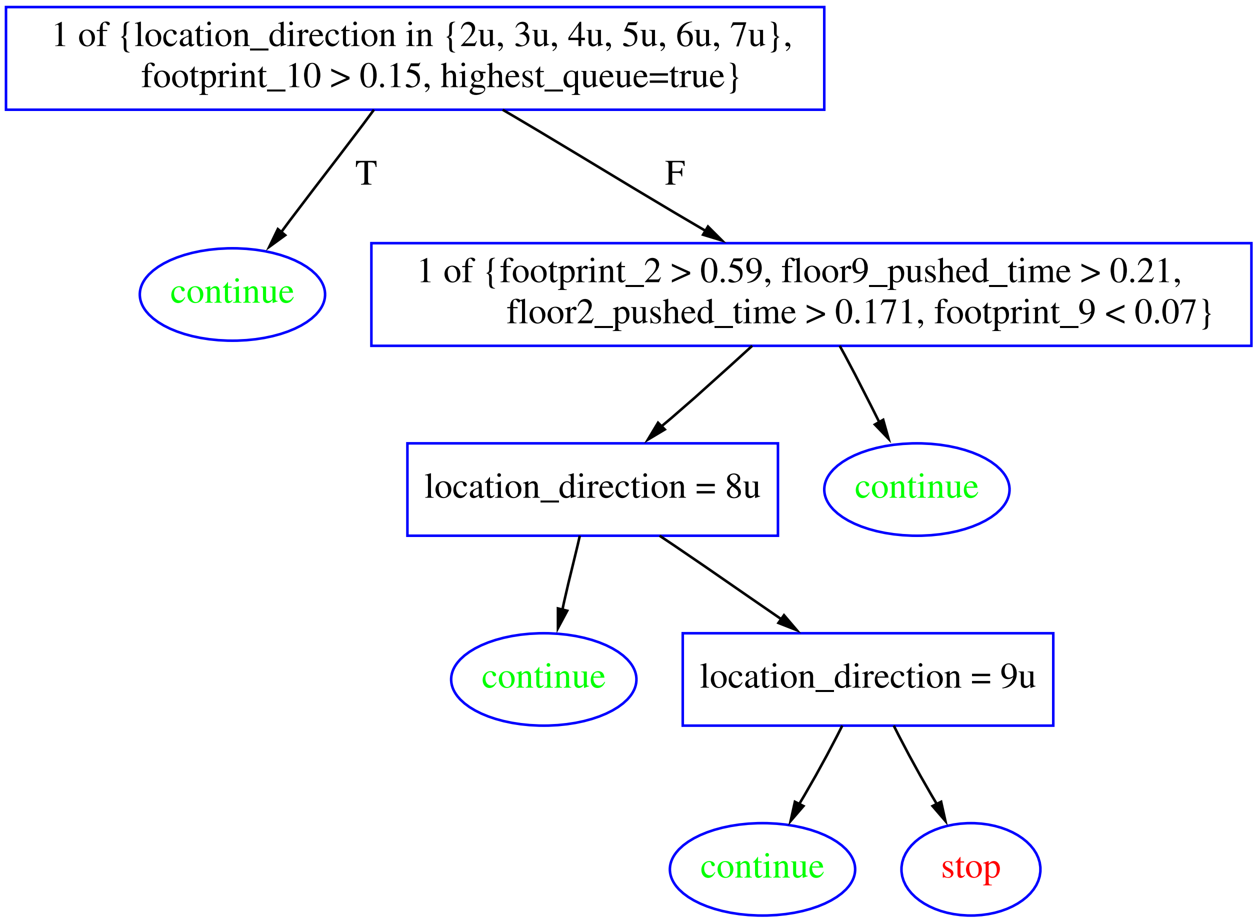

1 of {location_direction in {2u, 3u, 4u, 5u, 6u, 7u}, footprint_10 > 0.15, highest_queue=true}
T
F
continue
1 of {footprint_2 > 0.59, floor9_pushed_time > 0.21, floor2_pushed_time > 0.171, footprint_9 < 0.07}
location_direction = 8u
continue
continue
location_direction = 9u
continue
stop

Figure 39: **A representative tree for the `elevator-control` domain.**

Bibliography

Aha, D., Kibler, D., & Albert, M. (1991). Instance-based learning algorithms. *Machine Learning*, 6:37–66.

Alexander, J. A. & Mozer, M. C. (1995). Template-based algorithms for connectionist rule extraction. In Tesauro, G., Touretzky, D., & Leen, T., editors, *Advances in Neural Information Processing Systems (volume 7)*. MIT Press, Cambridge, MA.

Almuallim, H. & Dietterich, T. G. (1991). Learning with many irrelevant features. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, (pp. 547–552), Anaheim, CA. MIT Press.

Andrews, R. & Geva, S. (1995). Inserting and extracting knowledge from constrained error backpropagation networks. In *Proceedings of the Sixth Australian Conference on Neural Networks*, Sydney, Australia.

Angluin, D. (1988). Queries and concept learning. *Machine Learning*, 2:319–342.

Angluin, D. (1993). Learning with queries. In Baum, E. B., editor, *Computational Learning & Cognition: Proceedings of the Third NEC Research Symposium*. SIAM Books, Philadelphia, PA.

Ash, T. (1989). Dynamic node creation in backpropagation networks. *Connection Science*, 1(4):365–376.

Atlas, L., Cole, R., Connor, J., El-Sharkawi, M., Marks II, R., Muthusamy, Y., & Barnard, E. (1989). Performance comparisons between backpropagation networks and classification trees on three real-world applications. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*. Morgan Kaufmann, San Mateo, CA.

Auer, P., Holte, R. C., & Maass, W. (1995). Theory and applications of agnostic PAC-learning with small decision trees. In *Proceedings of the Twelfth International Conference on Machine Learning*, (pp. 21–29), Tahoe City, CA. Morgan Kaufmann.

Bao, G., Cassandras, C. G., Djaferis, T. E., Gandhi, A. D., & Looze, D. P. (1994). Elevator dispatchers for down peak traffic. Technical report, ECE Department, University of Massachusetts, Amherst, MA.

Baum, E. & Lang, K. (1991). Constructing hidden units using examples and queries. In Lippmann, R., Moody, J., & Touretzky, D., editors, *Advances in Neural Information Processing Systems (volume 3).* Morgan Kaufmann, San Mateo, CA.

Becker, S. & Le Cun, Y. (1988). Improving the convergence of back-propagation learning with second order methods. In Hinton, G. E., Sejnowski, T. J., & Touretzky, D. S., editors, *Proceedings of the 1988 Connectionist Models Summer School*, (pp. 29–37), San Mateo, CA. Morgan Kaufmann.

Blasig, R. (1994). GDS: Gradient descent generation of symbolic classification rules. In Cowan, J., Tesauro, G., & Alspector, J., editors, *Advances in Neural Information Processing Systems (volume 6).* Morgan Kaufmann, San Francisco, CA.

Blum, A. (1995). Empirical support for Winnow and Weighted-Majority based algorithms: results on a calendar scheduling domain. In *Proceedings of the Twelfth International Conference on Machine Learning*, (pp. 64–72), Tahoe City, CA. Morgan Kaufmann.

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. K. (1989). Learnability and the Vapnik-Chervonenkis dimension. *Journal of ACM*, 36(4):929–965.

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

Bruck, J. (1990). Harmonic analysis of polynomial threshold functions. *SIAM Journal of Discrete Mathematics*, 3(2):168–177.

Bshouty, N. H. (1993). Exact learning via the monotone theory. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, (pp. 302–311). IEEE Press.

Buntine, W. & Niblett, T. (1992). A further comparison of splitting rules for decision-tree induction. *Machine Learning*, 8:75–86.

Carpenter, G. A., Grossberg, S., Markuzon, N., Reynolds, J. H., & Rosen, D. B. (1992). Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps. *IEEE Transactions on Neural Networks*, 3:698–713.

Caruana, R. (1996). Algorithms and applications for multitask learning. In *Proceedings of the Thirteenth International Conference on Machine Learning*, Bari, Italy.

Caruana, R. & Freitag, D. (1994). Greedy attribute selection. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 28–36), New Brunswick, NJ. Morgan Kaufmann.

Catlett, J. (1992). Peepholing: Choosing attributes efficiently for megainduction. In *Proceedings of the Ninth International Conference on Machine Learning*, (pp. 49–54), Aberdeen, Scotland. Morgan Kaufmann.

Chatterjee, S. & Hadi, A. S. (1988). *Sensitivity Analysis in Linear Regression.* Wiley, New York, NY.

Cherkauer, K. & Shavlik, J. (1996). Growing simpler decision trees to facilitate knowledge discovery. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, (pp. 315–318), Portland, OR. AAAI Press.

Clark, P. & Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, 3:261–284.

Cleeremans, A., Servan-Schreiber, D., & McClelland, J. (1989). Finite state automata and simple recurrent networks. *Neural Computation*, 1:372–381.

Cohn, D., Atlas, L., & Ladner, R. (1994). Improving generalization with active learning. *Machine Learning*, 15(2):201–221.

Cohn, D. A., Ghahramani, Z., & Jordan, M. I. (1996). Active learning with statistical models. *Journal of Artificial Intelligence Research*, 4:129–145.

Craven, M. & Shavlik, J. (1992). Visualizing learning and computation in artificial neural networks. *International Journal on Artificial Intelligence Tools*, 1(2):399–425.

Craven, M. & Shavlik, J. (1993a). 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. & Shavlik, J. W. (1993b). Learning to predict reading frames in *E. coli* DNA sequences. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, (pp. 773–782), Wailea, HI. IEEE Press.

Craven, M. W. & Shavlik, J. W. (1993c). Learning to represent codons: A challenge problem for constructive induction. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, (pp. 1319–1324), Chambery, France. Morgan Kaufmann.

Craven, M. W. & Shavlik, J. W. (1994). Using sampling and queries to extract rules from trained neural networks. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 37–45), New Brunswick, NJ. Morgan Kaufmann.

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.

Crites, R. H. & Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In Touretzky, D., Mozer, M., & Hasselmo, M., editors, *Advances in Neural Information Processing Systems (volume 8)*. MIT Press, Cambridge, MA.

Das, S. & Mozer, M. (1994). A unified gradient-descent/clustering architecture for finite state machine induction. In Cowan, J., Tesauro, G., & Alspector, J., editors, *Advances in Neural Information Processing Systems (volume 6)*. Morgan Kaufmann, San Mateo, CA.

Dennis, S. & Phillips, S. (1991). Analysis tools for neural networks. Technical Report 207, Department of Computer Science, University of Queensland, Queensland, Australia.

Detrano, R., Janosi, A., Steinbrunn, W., Pfisterer, M., Schmid, J., Sandhu, S., Guppy, K., Lee, S., & Froelicher, V. (1989). International application of a new probability algorithm for the diagnosis of coronary artery disease. *American Journal of Cardiology*, 64:304–310.

Devroye, L. (1983). The equivalence of weak, strong, and complete convergence in L_1 for kernel density estimates. *The Annals of Statistics*, 11:896–904.

Dietterich, T., Kearns, M., & Mansour, Y. (1996). Applying the weak learning framework to understand and improve C4.5. In *Proceedings of the Thirteenth International Conference on Machine Learning*, Bari, Italy.

Drucker, H. & Cortes, C. (1996). Boosting decision trees. In Touretzky, D., Mozer, M., & Hasselmo, M., editors, *Advances in Neural Information Processing Systems (volume 8)*. MIT Press, Cambridge, MA.

Drucker, H., Schapire, R., & Simard, P. (1993). Improving performance in neural networks using a boosting algorithm. In Hanson, S. J., Cowan, J. D., & Giles, C. L., editors, *Advances in Neural Information Processing Systems (volume 5)*. Morgan Kaufmann, San Mateo, CA.

Ehrenfeucht, A. & Haussler, D. (1989). Learning decision trees from random examples. *Information and Computation*, 82:231–246.

Elman, J. L. (1991). Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning*, 7:195–225.

Fahlman, S. & Lebiere, C. (1989). The cascade-correlation learning architecture. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*, (pp. 524–532), San Mateo, CA. Morgan Kaufmann.

Fayyad, U., Piatetsky-Shapiro, G., & Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI Magazine*, 17(3).

Fayyad, U. M. & Irani, K. B. (1992). On the handling of continuous-valued attributes in decision tree generation. *Machine Learning*, 8:87–102.

Fisher, D. (1987). Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2:139–172.

Fisher, D. & McKusick, K. (1989). An empirical comparison of ID3 and back-propagation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, (pp. 788–793), Detroit, MI. Morgan Kaufmann.

Flann, N. & Dietterich, T. (1986). Selecting appropriate representations for learning from examples. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, (pp. 460–466), Philadelphia, PA. Morgan Kaufmann.

Frean, M. (1990). The upstart algorithm: A method for constructing and training feed-forward neural networks. *Neural Computation*, 2(2):198–209.

Freund, Y. (1990). Boosting a weak learning algorithm by majority. In *Proceedings of the Third Annual Workshop on Computational Learning Theory*, (pp. 202–216), Rochester, NY. Morgan Kaufmann.

Freund, Y. (1992). An improved boosting algorithm and its implications on learning complexity. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, (pp. 391–398), Pittsburgh, PA. ACM Press.

Freund, Y. (1993). *Data Filtering and Distribution Modeling Algorithms for Machine Learning*. PhD thesis, University of California at Santa Cruz. Available as technical report UCSC-CRL-93-37.

Freund, Y. & Schapire, R. E. (1995). A decision-theoretic generalization of on-line learning and its application to boosting. In *Proceedings of the Second European Conference on Computational Learning Theory*, Barcelona, Spain.

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

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.

Garey, M. R. & Johnson, D. S. (1979). *Computers and Intractability*. W. H. Freeman, New York, NY.

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.

Goh, T. H. (1993). Semantic extraction using neural network modelling and sensitivity analysis. In *Proceedings of the International Joint Conference on Neural Networks*, Nagoya, Japan.

Goldmann, M., Hastad, J., & Razborov, A. (1992). Majority gates vs. general weighted threshold gates. In *Proceedings of the Seventh IEEE Conference on Structure in Complexity Theory*, (pp. 2–13).

Gorman, R. P. & Sejnowski, T. J. (1988). Analysis of hidden units in a layered network trained to classify sonar targets. *Neural Networks*, 1:75–89.

Hancock, T. R. (1990). Identifying μ-formula decision trees with queries. In *Proceedings of the Third Annual Workshop on Computational Learning Theory*, (pp. 23–37), Rochester, NY. Morgan Kaufmann.

Hanson, S. & Burr, D. (1990). What connectionist models learn: Learning and representation in connectionist networks. *Behavioral and Brain Sciences*, 13:471–518.

Haussler, D. (1988). Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36:177–221.

Haussler, D. (1992). Decision theoretic generalizations of the PAC model for neural nets and other learning applications. *Information and Computation*, 100:78–150.

Hayashi, Y. (1991). A neural expert system with automated extraction of fuzzy if-then rules. In Lippmann, R., Moody, J., & Touretzky, D., editors, *Advances in Neural Information Processing Systems (volume 3)*. Morgan Kaufmann, San Mateo, CA.

Heckerman, D. (1995). A tutorial on learning Bayesian networks. Technical Report MSR-TR-95-06, Microsoft Research, Redmond, WA.

Hinton, G. (1986). Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, (pp. 1–12), Amherst, MA. Erlbaum.

Hinton, G. (1989). Connectionist learning procedures. *Artificial Intelligence*, 40:185–234.

Hogg, R. V. & Tanis, E. A. (1983). *Probability and Statistical Inference.* MacMillan Publishing, New York, NY.

Hong, Z. Q. & Yang, J. Y. (1991). Optimal discriminant plane for a small number of samples and design method of classifier on the plane. *Pattern Recognition*, 24(4):317–324.

Hunter, L. & Klein, T. (1993). Finding relevant biomolecular features. In *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology*, (pp. 190–197), Bethesda, MD. AAAI Press.

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.

John, G. H., Kohavi, R., & Pfleger, K. (1994). Irrelevant features and the subset selection problem. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 121–129), New Brunswick, NJ. Morgan Kaufmann.

John, G. H. & Langley, P. (1995). Estimating continuous distributions in Bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, (pp. 338–345), Montreal, Quebec. Morgan Kaufmann.

Jolliffe, I. T. (1986). *Principal Component Analysis.* Springer-Verlag, New York, NY.

Jordan, M. (1986). Serial order: A parallel distributed processing approach. Technical Report 8604, University of California, Institute for Cognitive Science, San Diego.

Kearns, M. J. & Mansour, Y. (1996). On the boosting ability of top-down decision tree learning algorithms. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, (pp. 459–468), Philadelphia, PA. ACM Press.

Kearns, M. J., Schapire, R. E., & Sellie, L. M. (1992). Toward efficient agnostic learning. In *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*, (pp. 341–352), Pittsburgh, PA. ACM Press.

Kearns, M. J. & Vazirani, U. V. (1994). *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA.

Kibler, D. & Langley, P. (1988). Machine learning as an experimental science. In *Proceedings of the Third European Working Session on Learning*, Glasgow, Scotland. Pitman.

Kira, K. & Rendell, L. A. (1992a). The feature selection problem: Traditional methods and a new algorithm. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, (pp. 129–134), San Jose, CA. AAAI/MIT Press.

Kira, K. & Rendell, L. A. (1992b). A practical approach to feature selection. In *Proceedings of the Ninth International Conference on Machine Learning*, (pp. 249–256), Aberdeen, Scotland. Morgan Kaufmann.

Kramer, A. H. & Sangiovanni-Vincentelli, A. (1989). Efficient parallel learning algorithms for neural networks. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 1)*, (pp. 40–48), San Mateo, CA. Morgan Kaufmann.

Kushilevitz, E. & Mansour, Y. (1991). Learning decision trees using the Fourier spectrum. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, (pp. 455–464), New Orleans, LA. ACM Press.

Lang, K. J. & Witbrock, M. J. (1988). Learning to tell two spirals apart. In *Proceedings of the 1988 Connectionist Models Summer School*, (pp. 52–59), Pittsburgh, PA. Morgan Kaufmann.

Langley, P. & Simon, H. A. (1995). Applications of machine learning and rule induction. *Communications of the ACM*, 38(11):54–64.

Le Cun, Y., Denker, J., & Solla, S. (1989). Optimal brain damage. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*, (pp. 598–605), San Mateo, CA. Morgan Kaufmann.

Lewis, J. (1991). *A Dynamic Load Balancing Approach to Control of Multiserver Polling Systems with Applications to Elevator System Dispatching*. PhD thesis, University of Massachusetts, Amherst, MA.

Littlestone, N. (1989). *Mistake Bounds and Logarithmic Linear-threshold Learning Algorithms*. PhD thesis, Department of Computer Science, University of California, Santa Cruz. Available as Technical Report UCSC-CRL-89-11.

Littlestone, N. (1995). Comparing several linear-threshold algorithms on tasks involving superfluous attributes. In *Proceedings of the Twelfth International Conference on Machine Learning*, (pp. 353–361), Tahoe City, CA. Morgan Kaufmann.

Loh, W.-Y. (1996). Personal communication.

MacKay, D. (1992). A practical Bayesian framework for backpropagation networks. *Neural Computation*, 4(3):448–472.

Mangasarian, O. L. & Solodov, M. V. (1993). Nonlinear complementarity as unconstrained and constrained minimization. *Mathematical Programming, Series B*, 62:277–297.

Mangasarian, O. L. & Solodov, M. V. (1994). Serial and parallel backpropagation convergence via nonmonotone perturbed minimization. *Optimization Methods and Software*, 4(2):103–116.

Matheus, C. J. (1990). *Feature Construction: An Analytic Framework and an Application to Decision Trees*. PhD thesis, University of Illinois at Urbana-Champaign.

McMillan, C., Mozer, M. C., & Smolensky, P. (1992). Rule induction through integrated symbolic and subsymbolic processing. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 4)*. Morgan Kaufmann, San Mateo, CA.

Medin, D. L., Wattenmaker, W. D., & Michalski, R. S. (1987). Constraints and preferences in inductive learning: An experimental study of human and machine performance. *Cognitive Science*, 11:299–339.

Michalski, R. (1983). A theory and methodology of inductive learning. *Artificial Intelligence*, 20:111–161.

Michalski, R. S. (1986). Understanding the nature of learning: Issues and research directions. In Michalski, R. S., Carbonell, J. G., & Mitchell, T. M., editors, *Machine Learning: An Artificial Intelligence Approach (volume II)*. Morgan Kaufmann, Los Altos, CA.

Minsky, M. & Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge.

Mitchell, T., Caruana, R., Freitag, D., McDermott, J., & Zabowski, D. (1994). Experience with a learning personal assistant. *Communications of the ACM*, 37(7):80–91.

Mitchell, T. M. (1980). The need for biases in learning generalizations. Technical Report CBM-TR-117, Department of Computer Science, Rutgers University, New Brunswick, NJ.

Moody, J. & Darken, C. (1988). Learning with localized receptive fields. In Hinton, G. E., Sejnowski, T. J., & Touretzky, D. S., editors, *Proceedings of the 1988 Connectionist Models Summer School*, (pp. 133–143), San Mateo, CA. Morgan Kaufmann.

Moore, A. W. & Lee, M. S. (1994). Efficient algorithms for minimizing cross validation error. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 190–198), New Brunswick, NJ. Morgan Kaufmann.

Mozer, M. & Smolensky, P. (1988). Skeletonization: A technique for trimming the fat from a network via relevance assessment. In *Advances in Neural Information Processing Systems (volume 1)*, (pp. 107–115), San Mateo, CA. Morgan Kaufmann.

Muggleton, S. & DeRaedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19:629–679.

Munro, P. (1991). Visualizations of 2-D hidden unit space. Technical Report LIS035/IS91003, School of Library and Information Science, University of Pittsburgh, Pittsburgh, PA.

Murphy, P. M. & Pazzani, M. J. (1991). ID2-of-3: Constructive induction of M-of-N concepts for discriminators in decision trees. In *Proceedings of the Eighth International Machine Learning Workshop*, (pp. 183–187), Evanston, IL. Morgan Kaufmann.

Musick, R., Catlett, J., & Russell, S. (1993). Decision theoretic subsampling for induction on large databases. In *Proceedings of the Tenth International Conference on Machine Learning*, (pp. 212–219), Amherst, MA. Morgan Kaufmann.

Neisser, U. & Weene, P. (1962). Hierarchies in concept attainment. *Journal of Experimental Psychology*, 64:640–645.

Noordewier, M., Towell, G., & Shavlik, J. (1991). Training knowledge-based neural networks to recognize genes in DNA sequences. In Lippmann, R., Moody, J., & Touretzky, D., editors, *Advances in Neural Information Processing Systems (volume 3)*, (pp. 530–536), San Mateo, CA. Morgan Kaufmann.

Nowlan, S. & Hinton, G. (1992). Simplifying neural networks by soft weight-sharing. *Neural Computation*, 4:473–493.

Opitz, D. (1995). *An Anytime Approach to Connectionist Theory Refinement: Refining the Topologies of Knowledge-Based Neural Networks.* PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI. Available as CS Technical Report 1281.

Opitz, D. W. & Shavlik, J. W. (1996). Generating accurate and diverse members of a neural-network ensemble. In Touretzky, D., Mozer, M., & Hasselmo, M., editors, *Advances in Neural Information Processing Systems (volume 8).* MIT Press, Cambridge, MA.

Ourston, D. & Mooney, R. (1994). Theory refinement combining analytical and empirical methods. *Artificial Intelligence*, 66(2):273–309.

Pazzani, M. & Kibler, D. (1992). The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94.

Pazzani, M. J., Muramatsu, J., & Billsus, D. (1996). Syskill & Webert: Identifying interesting Web sites. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, (pp. 54–59), Portland, OR. AAAI/MIT Press.

Pearl, J. (1988). *Probabalistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA.

Pineda, F. J. (1987). Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59:2229–2232.

Pinker, S. (1979). Formal models of language learning. *Cognition*, 7:217–283.

Poggio, T. & Girosi, F. (1990). Regularization algorithms for learning that are equivalent to multilayer networks. *Science*, 247:978–982.

Pollack, J. (1991). The induction of dynamical recognizers. *Machine Learning*, 7:227–252.

Pomerleau, D. A. (1993). *Neural Network Perception for Mobile Robot Guidance*. Kluwer, Boston, MA.

Pomerleau, D. A. & Touretzky, D. S. (1993). Understanding neural network internal representations through hidden unit sensitivity analysis. In *Proceedings of the International Conference on Intelligent Autonomous Systems*, Pittsburgh, PA. IOS Publishers.

Pratt, L. Y., Mostow, J., & Kamm, C. A. (1991). Direct transfer of learned information among neural networks. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, (pp. 584–589), Anaheim, CA. AAAI/MIT Press.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (1992). *Numerical Recipes in C*. Cambridge University Press, New York, NY, 2nd edition.

Provost, F. & Danyluk, A. (1995). Learning from bad data. In *Working Notes of the Workshop on Applying Machine Learning in Practice, Twelfth International Conference on Machine Learning*, Tahoe City, CA.

Quinlan, J. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.

Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning*, 5:239–2666.

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

Quinlan, J. R. (1996). Bagging, boosting, and C4.5. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, (pp. 725–730), Portland, OR. AAAI/MIT Press.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286.

Rice, J. A. (1995). *Mathematical Statistics and Data Analysis.* Wadsworth, Belmont, CA.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, 14:465–471.

Rissanen, J. (1989). *Stochastic Complexity in Statistical Inquiry.* World Scientific.

Rivest, R. (1987). Learning decision lists. *Machine Learning*, 2:229–246.

Rumelhart, D., Hinton, G., & Williams, R. (1986). Learning internal representations by error propagation. In Rumelhart, D. & McClelland, J., editors, *Parallel Distributed Processing: Explorations in the microstructure of cognition. Volume 1: Foundations.* MIT Press, Cambridge, MA.

Rumelhart, D. E., Durbin, R., Golden, R., & Chauvin, Y. (1995). Backpropagation: The basic theory. In Chauvin, Y. & Rumelhart, D. E., editors, *Backpropagation: Theory, Architectures, and Applications.* Lawrence Erlbaum, Hillsdale, NJ.

Sachs, L. (1984). *Applied Statistics: A Handbook of Techniques.* Springer-Verlag, New York, NY, 2nd edition.

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.

Sanger, D. (1989). Contribution analysis: A technique for assigning responsibilities to hidden units in connectionist networks. *Connection Science*, 1(2):115–138.

Sanger, T. D., Sutton, R. S., & Matheus, C. J. (1992). Iterative construction of sparse polynomial approximations. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 4).* Morgan Kaufmann, San Mateo, CA.

Schapire, R. E. (1990). The strength of weak learnability. *Machine Learning*, 5(2):197–227.

Schlimmer, J. C. & Fisher, D. (1986). A case study of incremental concept induction. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, (pp. 496–501), Philadelphia, PA. Morgan Kaufmann.

Sejnowski, T. & Rosenberg, C. (1987). Parallel networks that learn to pronounce English text. *Complex Systems*, 1:145–168.

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.

Setiono, R. & Liu, H. (1995). Understanding neural networks via rule extraction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, (pp. 480–485), Montreal, Quebec. Morgan Kaufmann.

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

Silverman, B. W. (1986). *Density Estimation for Statistics and Data Analysis.* Chapman and Hall, New York, NY.

Skalak, D. B. (1994). Prototype and feature selection by sampling and random mutation hill climbing algorithms. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 293–301), New Brunswick, NJ. Morgan Kaufmann.

Spackman, K. A. (1988). Learning categorical decision criteria. In *Proceedings of the Fifth International Conference on Machine Learning*, (pp. 36–46), Ann Arbor, MI. Morgan Kaufmann.

Stanfill, C. & Waltz, D. (1986). Toward memory-based reasoning. *Communications of the ACM*, 29(12):1213–1228.

Stone, M. (1974). Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society*, 36:111–147.

Stormo, G. (1987). Identifying coding sequences. In Bishop, M. J. & Rawlings, C. J., editors, *Nucleic Acid and Protein Sequence Analysis: A Practical Approach.* IRL Press, Oxford, England.

Sutton, R. S. & Matheus, C. J. (1991). Learning polynomial functions by feature construction. In *Proceedings of the Eighth International Machine Learning Workshop*, (pp. 208–212), Evanston, IL. Morgan Kaufmann.

Tan, A.-H. (1994). Rule learning and extraction with self-organizing neural networks. In *Proceedings of the 1993 Connectionist Models Summer School*, (pp. 192–199), Hillsdale, NJ. Lawrence Erlbaum Associates.

Tchoumatchenko, I. & Ganascia, J.-G. (1994). A Bayesian framework to integrate symbolic and neural learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 302–308), New Brunswick, NJ. Morgan Kaufmann.

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.

Thrun, S. (1996). *Explanation-Based Neural Network Learning: A Lifelong Approach.* Kluwer, Boston, MA.

Thrun, S. B. & Moeller, K. (1992). Active exploration in dynamic environments. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 4).* Morgan Kaufmann, San Mateo, CA.

Towell, G. (1991). *Symbolic Knowledge and Neural Networks: Insertion, Refinement, and Extraction.* PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI.

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

Towell, G. & Shavlik, J. (1994). Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1,2):119–165.

Towell, G., Shavlik, J., & Noordewier, M. (1990). Refinement of approximate domain theories by knowledge-based neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, (pp. 861–866), Boston, MA. AAAI/MIT Press.

Tresp, V., Hollatz, J., & Ahmad, S. (1992). Network structuring and training using rule-based knowledge. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 5)*. Morgan Kaufmann, San Mateo, CA.

Uberbacher, E. C., Einstein, J. R., Guan, X., & Mural, R. J. (1993). Gene recognition and assembly in the GRAIL system: Progress and challenges. In Lim, H., Fickett, J., Cantor, C., & Robbins, R., editors, *Proceedings of the Second International Conference on Bioinformatics, Supercomputing, and Complex Genome Analysis*. World Scientific, Singapore.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142.

Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge University, Cambridge, England.

Watrous, R. L. & Kuhn, G. M. (1992). Induction of finite state languages using second-order neural networks. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 4)*. Morgan Kaufmann, San Mateo, CA.

Weigend, A. (1994). On overfitting and the effective number of hidden units. In *Proceedings of the 1993 Connectionist Models Summer School*, (pp. 335–342), Hillsdale, NJ. Lawrence Erlbaum Associates.

Weigend, A. S., Zimmermann, H. G., & Neuneier, R. (1995). Clearning. Technical Report CU-CS-772-95, Computer Science Department, University of Colorado.

Weiss, S. M. & Kapouleas, I. (1989). An empirical comparison of pattern recognition, neural nets, and machine learning classification methods. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, (pp. 688–693), Detroit, MI. Morgan Kaufmann.

Wejchert, J. & Tesauro, G. (1989). Neural network visualization. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*, (pp. 465–472), San Mateo, CA. Morgan Kaufmann.

White, H. (1989a). Learning in artificial neural networks: a statistical perspective. *Neural Computation*, 1:425–461.

White, H. (1989b). Some asymptotic results for learning in single hidden-layer feedforward network models. *Journal of the American Statistical Association*, 84(408):1003–1013.

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.

Wolberg, W. H., Street, W. N., & Mangasarian, O. L. (1994). Machine learning techniques to diagnose breast cancer from fine needle aspirates. *Cancer Letters*, 77:163–171.

Wolpert, D. H. (1995). The relationship between PAC, the statistical physics framework, the Bayesian framework, and the VC framework. In Wolpert, D. H., editor, *The Mathematics of Generalization*. Addison-Wesley, Reading, MA.

Zadeh, L. A. (1965). Fuzzy sets. *Information and Control*, 8:338–353.

Zeng, Z., Goodman, R. M., & Smyth, P. (1993). Learning finite state machines with self-clustering recurrent networks. *Neural Computation*, 5(6):976–990.