Classification and Approximation with Rule-Based Networks

Charles M. Higgins, Jr.

Department of Electrical Engineering
California Institute of Technology
Pasadena, California

1993

CLASSIFICATION AND APPROXIMATION WITH RULE-BASED NETWORKS

Thesis by
Charles M. Higgins, Jr.

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1993

(Defended May 12, 1993)

Publication History:

- Thesis approved by defense committee (defense May 12, 1993):

Dr. Edward Posner, EE	May 12
Dr. Joel Franklin, AMa	May 12
Dr. Yaser Abu-Mostafa, EE	May 12
Dr. Rodney Goodman, EE (Chair)	May 19
Dr. Richard Murray, ME	May 25

- Final thesis delivered to Graduate Office May 27, 1993
- First printing, June '93 - 15 copies
- Second printing - special hardback edition, June '93 - 3 copies (to author's parents, author, and advisor)

I would never have written this thesis without the love and support of my parents through my ten-year college ordeal. They have consistently supported me both monetarily and emotionally, and it is to them that I truly owe this achievement.

This thesis is dedicated to my parents, with love.

Acknowledgements

First, I would like to thank my advisor, Rod Goodman, for his practical advice on difficult problems, his endless enthusiasm for research, and his uncritical support. I would also like to thank Rod for providing me with the best equipment and virtually unlimited resources for pursuing my many ideas over the last four years.

I would like to thank the members of the MicroSystems Group, both past and present, for enlightening research discussions, mathematical consultations, late-night computer game playing, snack runs to the stockroom, and a generally pleasant atmosphere in which to work. I thank you all for your tolerance of the noise and disturbance my experimentation made.

Every member of the group has at some point helped me to work on the model car experiment (which didn't make it into this thesis!), but I would never have gotten as far as I did on it without the insightful suggestions of Bhusan Gupta, Chris Ulmer, Jeff Dickson and Andrew Lundsten.

My special thanks to Jeff Dickson for helping me get started on the ball-and-beam hardware.

I would like to thank Padhraic Smyth, on whose research all my work is based, for four years of consultation on a wide range of issues, culminated by a 15-minute discussion in which he suggested a solution to all the theoretical issues I had been unable to resolve for three years.

Part two of this thesis, and especially chapter nine, benefitted greatly from the comments of Dr. Richard Murray. I am indebted to him not only for his practical advice, but also for helping me to see my work in a control systems perspective.

Finally, I would like to thank Michelle Stratton for endless spins on the dance floor and for helping me to release all of the pressures of graduate life. Your love and devotion makes this all worthwhile.

Preface
Systems that can *Explain*

We live in an age in which most of the difficult problems – including pattern classification and function approximation, the two problems addressed in this thesis – have been studied in depth. The breadth of research in any given field is absolutely astounding. There is a large body of theory which tells us how well we can do and offers a multitude of methods for achieving nearly that performance. What, then, is left to be researched in these problems? The availability of computing power is at an all time high, so optimizing the speed of a solution is no longer of primary concern. Are not existing methods good enough for any application?

If the only concern is performance, the answer is yes. We can achieve near-optimal classification performance with a number of algorithms; the same can be said for function approximation. However, these algorithms may be unsatisfactory for many applications because it is difficult to understand what the system is doing; they lack the ability to *explain* to the user what has been learned about the problem.

Can we retain near-optimal performance while making systems easier to understand and use? We can, and that philosophy pervades this thesis. Systems which express their problem-specific knowledge in the form of rules seem quite intuitive to humans and, as shown in the pages to follow, can achieve performance comparable with other paradigms. These efforts are a major step towards computing systems which can not only solve a difficult problem, but also explain how they have done it.

Abstract

This thesis describes the architecture of learning systems which can explain their decisions through a rule-based knowledge representation. Two problems in learning are addressed: pattern classification and function approximation.

In Part I, a pattern classifier for discrete-valued problems is presented. The system utilizes an information-theoretic algorithm for constructing informative rules from example data. These rules are then used to construct a computational network to perform parallel inference and posterior probability estimation. The network can be extended incrementally; that is, new data can be incorporated without repeating the training on previous data. It is shown that this technique performs comparably with other techniques including the backpropagation network while having unique advantages in incremental learning capability, training efficiency, and knowledge representation. Examples are shown of rule-based classification and explanation.

In Part II, we present a method for the learning of fuzzy logic membership functions and rules to predict a numerical function from examples of the function and its independent variables. This method uses a three-step approach to building a complete function approximation system: first, learning the membership functions and creating a cell-based rule representation; second, simplifying the cell-based rules using an information-theoretic approach for induction of rules from discrete-valued data; and finally, constructing a computational network to compute the function value given its independent variables. Applications of the system to adaptive control are suggested, including a method for learning a complete control system for an unknown plant. Experimental validation of the suggested methods using a ball-and-beam system is shown.

Table of Contents

List of Figures

Part I

Pattern Classification with Rule-Based Networks

Chapter 1
Introduction to Classification

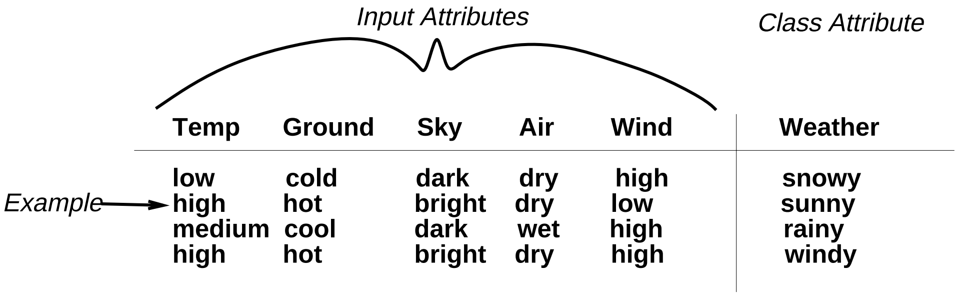
For decades now, computer systems have been able to capture visual images, record sound, and even detect such esoteric quantities as barometric pressure and humidity. Why, then, can't your computer recognize you when you walk up to it? Why can't it listen to thunder and tell you it's going to rain?

The answer is that computers can't do these things because the data which must be processed to make such decisions is incredibly complex. These decisions not only arise from a large number of factors, but also these factors are subject to random variation and environmental situations such as lighting and background noise. Any of today's computers can outperform a human in taking square roots, but in face recognition a human will win every time; sensory problems are among the hardest problems that exist for computing systems today.

There are a number of promising approaches to sensory problems. Existing systems can generally be broken into two parts: sensor preprocessing and higher-level processing. Sensor preprocessing involves processing what sensors we have in such a way as to remove the noise and reduce the complexity as much as possible without losing their predictive power. These processed sensors are often referred to as 'features,' and are input to higher-level systems. It is believed that higher-level systems which could solve realistic sensory problems will be hierarchical in nature. For instance, a face recognizer might have a processing layer including an ear recognizer, a nose recognizer and an eye recognizer, and then another layer which examines the ears, nose, and eyes to determine the particular identity of the face. In general, any a priori structure which we can give to the problem will improve our results. Most of the efforts in the past several decades have been focused on solving the lower-level portions of these sensory problems with a single system; usually, some form of pattern classifier.

Pattern Classification may be defined as "machine recognition of meaningful regularities in noisy or complex environments"[DH73]. In context, pattern classification is taking as input the preprocessed sensors and making a decision as to the type or 'class' in which it falls. Like the previous step of sensor preprocessing, it serves to reduce the complexity of the input in such a way that a higher-level system can deal with it more easily. The processing of sensory data into useful features is largely a black art, and is highly dependent on the problem at hand; thus classification is the lowest-level system that can be made completely problem independent. We will consider classifier systems which take as input preprocessed features; we assume that the input is in some way predictive of the class. The limits of our performance will rely largely upon the quality of feature preprocessing.

A large amount of pattern classification research has been done not in the pursuit of sensory problems, but with application to database analysis. Physicians wish to predict the likelihood of a new patient's falling ill given his symptoms by analyzing a database of previous patient symptoms and outcomes. Teachers wish to determine which students need extra help given their aptitude test scores by analyzing a database of past student test scores and performance. Manufacturers wish to determine which parts will fail under stress


Input Attributes
Class Attribute
Example

Temp	Ground	Sky	Air	Wind	Weather
low	cold	dark	dry	high	snowy
high	hot	bright	dry	low	sunny
medium	cool	dark	wet	high	rainy
high	hot	bright	dry	high	windy

Figure 1.1: Example classifier input file

by analyzing a database of manufacturing parameters and previous part failures. In these problems, the data can often be input directly to the classifier; little or no preprocessing is required.

In Part I of this thesis, we consider a special (but important) case of pattern classification, applicable to both sensory problems and database analysis.

1.1 Defining Classification

The input to a classification system is a database such as that shown in Figure 1.1. Each row of the database is an independent *example*; the columns will be referred to as *attributes* of the data, and each value of the output attribute is a *class* in which the data may be placed. Let us pose the classification problem in the following manner: learn to predict the value of the output (class) attribute given the input attributes by looking at examples of all the attributes. This form of learning is known as *supervised* because for each example, the correct class is provided.

We restrict ourselves to the *discrete* classification problem. A real-world classification problem may have continuous inputs, instead of discrete ones. While discretizing the inputs must result in some loss of information, there exist techniques for doing this in such a way as to maximize the predictive power of the input. (See Section 5.3 for suggestions on this problem.)

We assume no knowledge of the probability distributions of the input attributes, in order to tackle the most general (and difficult) classification problem possible. If knowledge of the distributions of one or more of the input attributes is known, a parameter estimation technique will likely be more effective than the more general systems we will describe.

It is important to note that the classifier will be tested on examples it has never seen before; thus it must have the ability to *generalize* from the examples it has seen. Just remembering the examples it has already seen is not sufficient. A basic assumption of pattern classification is that the set of examples shown to the classifier (referred to as the *training set*) contains examples which are incorrect in some way; that is, they don't follow the underlying pattern of the data set. We refer to these examples as 'noise,' since they tend to distract us from the actual pattern we are seeking. Thus just remembering the training set examples will lead to learning the noise as well, and the performance on the set of examples used to test the classifier (referred to as the *test set*) will suffer.

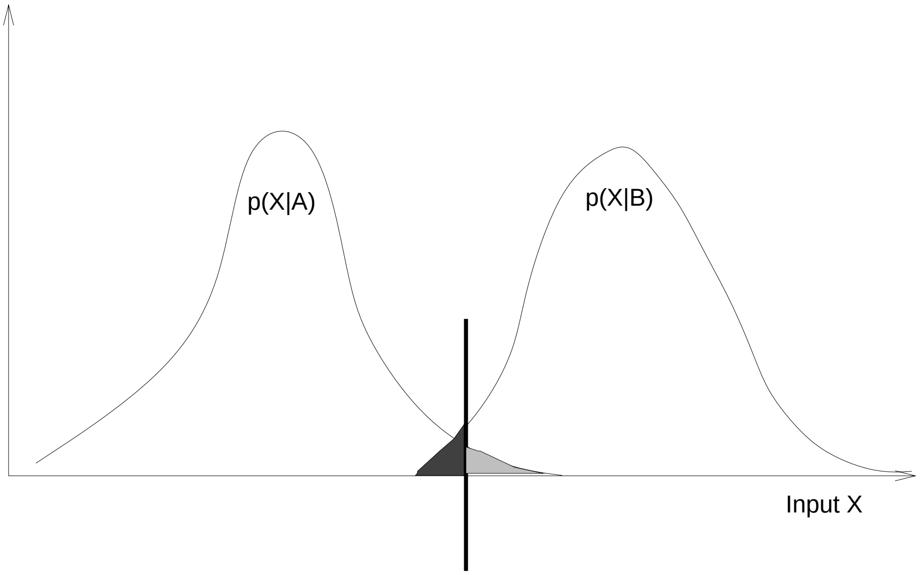


Figure 1.2: The concept of the Bayes risk

1.2 Choosing a Classifier

There are literally dozens of techniques for analyzing such databases as described above (see [DH73]). Springing from statistical theory are techniques such as principal component analysis, factor analysis, and Fisher's linear discriminant analysis, all of which serve to reduce the dimensionality of the data. There also exist statistically-based classifier techniques such as Bayesian classifiers, nearest-neighbor classifiers, and Parzen window density estimators. From modern neural network theory, there is the backpropagation classifier [RHW86].

Before motivating a decision as to which method is best for a specific problem, we must formalize the concept of the 'best' we can do in predicting a class output from a database. From statistical theory, there is the concept of the *Bayes risk*. This concept may be understood by looking at Figure 1.2. Given each class, there is a probability distribution of the inputs. Let us draw a decision boundary at a given point on the input axis. To the left side of this point, we will conclude that the class is A; to the right, B. The probability that we will make an error is the sum of all probability of class B to the left of our decision boundary and all probability of class A to the right. The *optimal* decision boundary may be drawn at the place at which the sum of these probabilities is minimum. If the conditional probability distributions overlap at all, then even with the optimal decision boundary drawn, there will still be some probability that we will make an error. This probability is called the Bayes risk, and its obverse, the Bayes rate, is the best classification performance that you can expect from any system. Of course, we cannot know the exact probability distribution of our inputs for a real problem and thus we cannot always *calculate* the Bayes rate; however, it is important to realize that in every case there is a 'best' we can do which may not be perfect.

Each of the pattern classification methods mentioned above has its own advantages and disadvantages, but on most problems one or more of these methods will prove successful in predicting the class output at a performance approaching the Bayes rate. In this case, it is clear that the choice of a classifier should not depend only on its performance, but upon other factors. In the remaining chapters of Part I of this thesis, we will present a rule-based classifier system whose performance approaches the Bayes rate, and yet has the advantage over the aforementioned methods that it can explain what it has learned.

Chapter 2 Rule-Based Systems

In order to construct a classifier system which can explain itself, we will make use of a rule-based knowledge representation. In this chapter, we will motivate the use of rules in a probabilistic framework. We begin with a history of rule-based systems.

2.1 A Brief History of Rule-Based Systems

In this section, we will review the origins and applications of rule-based systems and review rule-learning techniques to shed light on the novelty of our approach.[1]

Rule-based systems began with *production systems*, a general form of rule-based inference system proposed by Post [Pos43] in the 1940's. Because of the intuitive similarity of a production system to the patterns of human thought, production systems soon caught the interest of cognitive scientists. Newell and Simon [NS65, New68] researched the application of these systems to psychological modeling in the 1960's, and in the early 1970's [NS72], proposed them as a good candidate for human cognitive modeling in general. More recent work by Holland [HHNT86] continues to support the use of production systems for cognitive models.

The most widespread practical application of rule-based systems has been in expert systems (often cited as the only useful thing to come out of the golden age of symbolic AI research), which use rules obtained by querying a human expert to solve complex decision problems. The idea is to obtain in rule-based form the entire knowledge of an expert for a particular problem. Avoiding contradictions and constructing a sufficiently complete rule set becomes quite difficult for a large problem. This difficulty has become known as the *knowledge acquisition bottleneck*. Still, there are many successful examples of expert systems constructed this way. Some famous examples are MYCIN [SAB$^+$73], which gives advice on diagnosis and therapy for infectious diseases, DENDRAL [LBFL80], which analyzes mass spectrograms to determine molecular structure, and XCON/R1 [McD80], which configures VAX systems for Digital Equipment Corporation. Expert systems are still widely used today, as can be seen by the literally hundreds of papers about them in the literature of various disciplines, for example, Civil Engineering [TTE91], Chemistry [MWM91, HBZ92], Chemical Engineering [GS93], and Nuclear Science [CCC93].

A major part of the symbolic AI effort was devoted to the learning of rules from domain examples. Mitchell's 'version spaces' [Mit77] algorithm learned by looking at examples one after the other and generalizing or specializing as necessary so as to arrive at a reasonable rule set. This system relied on a sequential analysis of carefully chosen examples. Meta-DENDRAL [BM78] is an extension of the DENDRAL expert system which was able to learn its own production rules in its specific domain. It was quite successful, even discovering rules previously unknown to chemists, but did not generalize well to other domains. Michalski's AQ11 [MC80] was a more general rule-learning algorithm that actually outperformed the expert in its domain! Unfortunately, while the above algorithms are fascinating from a cognitive science viewpoint, the symbolic AI approach tends to break down when given noisy examples and fails to scale up to problems of larger dimensionality.

[1] For a more detailed history of rule-based systems, see [Smy88].

More recently, interest in database analysis has generated methods for discovery of rules. Gaines and Shaw [GS86] use fuzzy logic to induce inference rules about a very specific type of database (a repertory grid). Piatetsky-Shapiro [Pia89] presents a method for learning 'strong' rules from a general database using an *ad hoc* rule interest measure. Schlimmer et al. [SMM91] suggest a method for improving an expert rule set by using examples to suggest refinements to the rules. McMillan et al. [MMS91] propose a neural network which learns explicit rules by a winner-take-all competition between all possible rules. See [PSF91] for an excellent survey of 'database mining' techniques.

Also quite recently, in the interest of explanation, there have been numerous attempts to extract linguistic rules from existing learning paradigms. For example, Quinlan [Qui87] shows how to extract rules from decision trees, and Tresp and Hollatz [TH93] present a method for embodying rule-based knowledge in a neural network and then extracting learned rules from the network.

2.2 A Novel Approach to Rule-Based Systems

Most of the rule-learning approaches mentioned above are based upon strategies which completely lack a theoretical basis. To arrive at a theoretically sound method for the learning of rules, we will apply probability and information theory.

2.2.1 A Probabilistic Framework

Since we are processing a database which contains noisy examples, we will use a probabilistic framework. Statistically-based systems are much more robust to noise than other systems. Even if the noise is severe, the underlying statistics of the data can often be determined.

We will make some simple probabilistic assumptions about the database which provide a theoretical backing for the calculations we will make later. We will first assume that the examples in the database are statistically independent identically distributed samples from an underlying probability distribution. Thus we can use each one as an equally valid sample for estimating probabilities via relative frequencies. We will also assume that the input attributes are independent conditioned on the value of the output attribute. That is, for a *given* class, the input attributes are independent. This assumption is usually reasonable, and will be needed later to compute the probability of each class given a set of inputs.

To use a probabilistic framework, our classifier must obtain estimates of probabilities from the database using relative frequencies. To function well, such a classifier should be given as input a database which accurately and completely represents the statistics of the data. Small-sample statistics may be necessary for the processing of extremely small databases.

2.2.2 Why a Rule-Based Knowledge Representation?

Assuming that we use a probabilistic framework, why should we use rules as a knowledge representation? To answer this question, we must delve again into a bit of history.

Probabilistic systems which learn from examples have been proposed since early in the latter half of this century. Uttley [Utt59] conceived of a network in which the probability of every combination of the inputs was stored. This network stores a number of probabilities exponential in the number of attributes, but contains the information necessary for calculating the conditional probability of *any* conjunction of attribute values given that *any* conjunction of attribute values is occurring. This system stores a very large amount of information, much of which may not be valuable. On the other extreme, the Bayesian networks of Kononenko [Kon89] store information only about the probabilities of each input and pair of inputs. Kononenko's network has the same number of nodes as there are attributes, but only storing information about joint probabilities must make the restrictive assumption that all attributes are independent in order to estimate conditional probabilities. This network stores too little information to predict more complex relationships. Somewhere between these two extremes lie systems which decide which higher-order conjunctive probabilities to store without storing them all. Networks which choose such higher-order connections randomly were among the first neural networks [Ros62]. Ekeberg and Lansner [EL88] show a method based on a correlation/dependency measure for deciding which such probabilities to store, but again must assume independence of attributes to estimate conditional probabilities.

We will show in this thesis that information theory provides a measure for deciding which higher-order relationships are predictive that is superior to simple correlational or random methods and may be used for conditional probability estimation with less stringent assumptions. In addition, this measure has the advantage that the relationships found may be interpreted as rules and used in explaining a decision of the system.

2.3 Limitations of Rule-Based Systems

With a rule-based knowledge representation, a system can represent any possible pattern of output classes in input space.[2] However, how simply can a rule-based system represent a given class boundary? This depends on the shape of the boundary in input space. Because a rule is expressed in the form **if** y **then** x, where y is a conjunction of input attributes, it is making a statement about the value of the output at a point, on a line, or in a box or hyperbox in input space the sides of which are *parallel to the input axes.* What happens if a class boundary occurs along a *diagonal*? In this case, it will take a larger number of rules to represent the class. For example, see Figure 2.1. Class three, the boundary of which lies along an input axis, can be simply represented. However, classes one and two, whose boundary lies on a diagonal, require a large number of rules to specify. We will refer to this problem as the *axis-parallel* problem. This problem can be simply stated as follows: a rule-based representation is much simpler if the output 'features' are parallel to the input axes. We will encounter this problem in both parts of this thesis.

[2] In the pathological case, there will be a rule for each example.

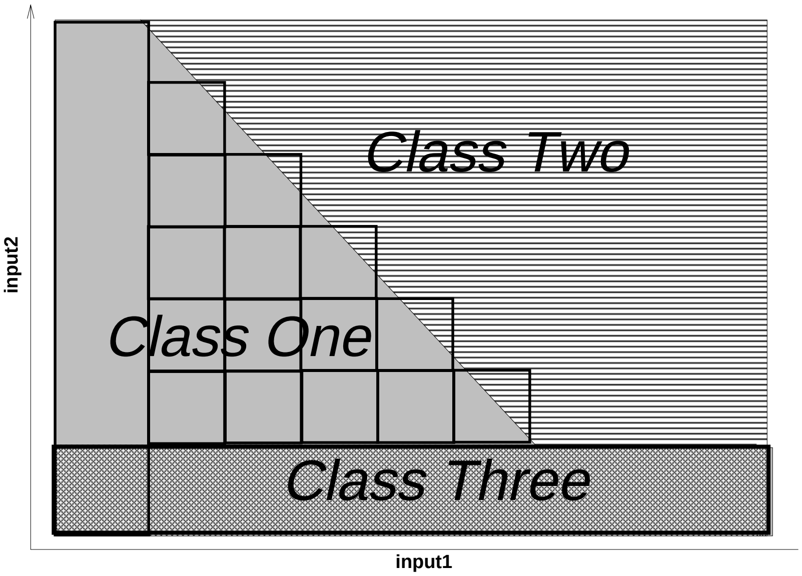


Figure 2.1: Axis-parallel problem example

Chapter 3 A Rule-Based Classifier System

In this chapter, we describe a classifier system based upon information and probability theory. The system uses a rule-based knowledge representation, and yet can achieve performance approaching the optimal. The final system is formulated in terms of a computational network for efficient output computation.

We begin by describing the research in Rod Goodman's lab at Caltech which led to the development of this system.

3.1 Previous Research

3.1.1 A Measure for Comparing Rules

Smyth and Goodman [GS88] have developed an information-theoretic measure of rule value with respect to a given discrete data set. This measure is known as the j-measure; defining a rule as *if y then X* where y is a conjunction of input variable values and X is a value of the output variable, the j-measure can be expressed as follows:

$$j(X|y) = p(X|y)\log_2(\frac{p(X|y)}{p(X)}) + p(\bar{X}|y)\log_2(\frac{p(\bar{X}|y)}{p(\bar{X})})$$

This measure is the average change in bits necessary to specify X between the *a priori* distribution ($p(X)$) and the *a posteriori* ($p(X|y)$) distribution. Smyth and Goodman have shown that this measure can be interpreted as a special case of the cross entropy of the two distributions and satisfies all the properties of an information measure. This measure can be thought of as a pure 'goodness' measure, since it values only the correctness of the rule. Any inconsistency with the data is harshly penalized.

[GS88] also suggests a modified rule measure, the J-measure:

$$J(X|y) = p(y)j(X|y).$$

The term $p(y)$ is a simplicity term, valuing the simpler, more common clauses over the complex, rarer ones. Together with the goodness term, this measure discounts rules which are not as useful in the data set in order to remove the effects of 'noise' or randomness. This has the effect of bringing out the underlying pattern in the data. (See Section 8.3.1 for a discussion of alternate simplicity terms.)

The probabilities in both measures are computed from relative frequencies counted in the given discrete data set. Thus, given a rule, we can go through a data set, count up matches with the left-hand side, right-hand side, both left- and right-hand sides, and so forth, to calculate a measure of the value of a rule. This allows us to objectively state which of two candidate rules is the better.

3.1.2 The ITRULE Algorithm

Now that we can rank rules, how can we search the space of all possible rules to find the best? The *Information Theoretic RULE induction* algorithm (ITRULE) [Smy88] was developed for just this purpose.[1] The user will specify two parameters to limit our search: first, the maximum order (number of conditions) d beyond which we will not search, and second, the number of rules R he wishes to obtain.

Since the best rules are more likely to be lower in order due to the preferences of the J-measure, we will start with first order rules and specialize from there. We begin by calculating the J-measure for all first-order rules. We save rules in a rank-ordered list R long, so that only the best R rules found so far are saved. We will now consider increasing the order of each of the first-order rules in a depth-first search of all possible specializations of each rule; we continually save the best rules in our list. (To avoid calculating the J-measure for any rule twice, we will begin with the first-order rule with the lowest-numbered attribute, and consider specializing each first-order rule only with attributes numbered greater than its own.)

A bound for the specialization of a rule has been developed to help speed up the search algorithm. It can be shown [Smy88] that the J-measure of a specialized rule J_s which adds any condition to a rule *if* y *then* X is bounded by

$$J_s \leq max\{p(X,y)\log(\frac{1}{p(X)}),p(\bar{X},y)\log(\frac{1}{p(\bar{X})})\}$$

This bound will be used to avoid specializing any rule for which no improvement could be achieved.

The ITRULE algorithm is *optimal*, in that it produces the best R rules (in a J-measure sense) of order d or less.

3.1.3 Pruning Rule Sets

By using the J-measure, ITRULE finds the best rules with each rule *in isolation*, not the best *rule set*. There are three reasons for which we will pick only some of these rules to use in our classifier. First, if the user is interested in data analysis, he probably doesn't want to look at *all* the best rules; just the smallest set of rules which are necessary to cover the data. Second, due to the assumptions we make in doing inference with the rules (see Section 3.1.4), the rule conditions should be independent given the class for the best performance. Finally, it is clear that by letting the user decide upon the number of rules R which he wants and then using those rules directly in a classifier, one will obtain performance extremely dependent upon the choice of R. The rules in the best rule set are among those with the highest J-measures, but may not include all of them. Thus both for understandability and for improved classification performance, we must set the R parameter to a large value and use a rule *pruning*[2] algorithm to pick the best rule set.

Pruning Algorithms

Many algorithms have been proposed for rule pruning. One of the simplest (and most time consuming) approaches to rule pruning is a *greedy search* for the rule set which leads to the best classification performance on the training set. This approach starts with an empty rule set and adds a rule only if it improves the overall classification performance. This

[1] While we will describe ITRULE in a classification context, it is actually capable of making more general expert system rules. See [Smy88] for details.

[2] This usage comes from the pruning of decision trees.

pruning method is very time consuming, but yields an excellent classification percentage *on the training set*. The performance on the test set varies.

Perhaps the most successful pruning algorithm is the independence pruning approach, which enforces the conditional independence assumption. This algorithm starts with all rules in the set and checks each pair with the same class output for dependence. Dependence is defined as follows: if the two rules have condition sides y_1 and y_2, the rules are dependent if

$$\frac{|p(y_1, y_2) - p(y_1)p(y_2)|}{p(y_1, y_2)} > T$$

where $0 < T < 1$ is a user-set threshold. The dependent rule with the lower J-measure is removed from the rule set. This algorithm is not only more computationally efficient than the greedy approach, but also has much better generalization performance.

3.1.4 Estimating the Posterior Probability

Once the rule set is constructed, the rules may be used in parallel to compute the posterior probability of each class. This approach to rule-based classification is described in detail in [GHMS92].

Let the subset of all the rules whose condition sides are satisfied by an example E in the training set and whose conclusion side conclude $X = x_j$ be called R_j. Then we estimate the probability that $X = x_j$ as

$$\log(\hat{p}(x_j|E)) = \log(p(x_j)) + \sum_{i=1}^{|R_j|} W_{i,j} \text{ where } W_{i,j} = \log \frac{p(x_j|y_i)}{p(x_j)}$$

This formula can be easily calculated by assuming conditional independence of the y_i's. Once the posterior probabilities are estimated, we need only choose the largest probability to make our classification decision.

The above formula provides a simple method of constructing a computational network. Consider the network of Figure 3.1. The input layer contains one node for each attribute except the class attribute. Each node in the second layer represents a rule generated by the algorithm. Nodes in this layer are connected to the input nodes of the attributes in the condition side of the rule which they represent; they output a 1 if the condition side of the rule is satisfied. The third layer contains a node for each value of the class attribute. Each second-layer node representing rule i is connected to third-layer node j with a multiplicative weight $W_{i,j}$. The bias of each third-layer node is $-\log(p(x_j))$. Each third-layer node sums its inputs, subtracts the bias and *exponentiates* the result. Thus, by the above formula, the output of each third-layer node is the posterior probability of the class it represents. If desired, a winner-take-all stage can be added to decide upon the most likely class.

3.2 SQUEEZE: A New Algorithm for Learning Rules

In practice, the ITRULE search algorithm often ends up searching every rule in the space to the maximum order it is allowed; the bound on the search is not very tight. For larger data sets, this can mean weeks of computation time. In addition, rule pruning can be quite time consuming in itself and often does not yield a satisfactory set of rules. Several

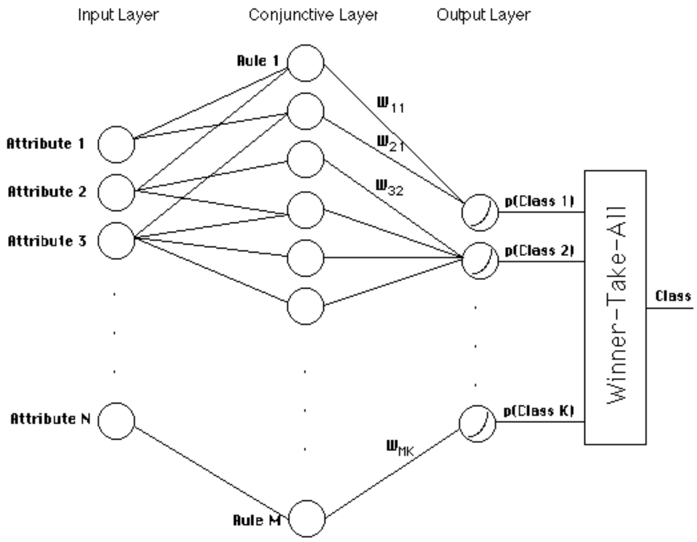


Figure 3.1: Rule-based network

alternative search algorithms have been tried, including genetic algorithms. The following algorithm was designed to generate rules more quickly than ITRULE, not require rule pruning at all, and create a concise, understandable explanation of a data set in the form of rules. This algorithm searches a smaller subset of the search space than other algorithms by using the examples directly as templates for rules.

3.2.1 Statement of the Algorithm

Given a training set of examples, an obvious way to classify is to retain all the examples and match an incoming example to be classified to an example in storage. This is equivalent to regarding the examples as very high-order specific rules. However, these rules will not match any example not explicitly contained in the training set and also model the noise in the training set. Consider now if we could decide which attributes in each example to remove in order to generalize the examples to rules which cover more examples and remove the statistically insignificant noise in the data set; the J-measure provides just such a way.

The proposed algorithm for rule generation is as follows. If there are N attributes excluding the class attribute, each initial rule is of order N. For each rule independently, do the following:

1. Calculate the J-measure for the rule. Call this rule the parent-rule.

2. For each of the child-rules generated by removing a single attribute from the parent-rule, calculate the J-measure (If the parent-rule was order K, each of the K child-rules is order $K-1$).

3. Choose the rule among the parent rule and the set of child-rules with the greatest J-measure. Special cases:

   (a) If two rules have the same J-measure, choose the one with the lower order.

   (b) If two rules of the same order have the same J-measure, choose a random one.

4. If the chosen rule is not the parent rule, the chosen rule becomes a new parent rule; repeat the process starting at step 2. If the chosen rule is the parent rule, terminate.

It should be noted that each example may be expanded in parallel, since each is independent of all others. Before conversion to a computational network, duplicate rules can be removed to reduce the complexity of the network generated. However, the weight of each removed rule should be added to its duplicate which remains. Thus if a rule had five duplicates, it will have six times its original weight. This serves to preserve the statistics of the original data set.

A simple example of the application of this algorithm is given in Figure 3.2. The data set is shown at the top of the figure — it is the truth table for a logical AND. The inputs (A and B) and output (C) are binary. The tree of rules is shown for each example; the J-measure is shown for each possible rule. In each case, the algorithm progresses down the tree, taking the highest J-measure at each step, until none of the children are an improvement. The resulting rules are shown at the bottom of the figure.

Incremental Learning

What if all of the training data is not available at one time? Imagine we are learning to predict a medical diagnosis and patient data is coming in every day. We wish to be able to train our classifier system on the data we have now and update it with later data without repeating all the work. This is known as *incremental learning* and is an important feature of any real learning system [Qui91]. It turns out that the SQUEEZE algorithm may be simply extended to learn incrementally.

To learn rules incrementally, perform the SQUEEZE algorithm on each example in the initial training set. Retain for each rule the original example which generated it. When more training data is available, use the following modification to the SQUEEZE algorithm. For each rule generated from the initial training set:

1. Calculate the J-measure for the rule. Call this rule the parent-rule.

2. For each of the child-rules generated by removing a single attribute from the parent-rule, or *adding back any single previously removed attribute to the parent-rule*, calculate the J-measure (If the parent-rule was order K, the child-rules may be order $K-1$ or $K+1$).

3. Choose the rule among the parent rule and the set of child-rules with the greatest J-measure. Special cases:

   (a) If two rules have the same J-measure, choose the one with the lower order.

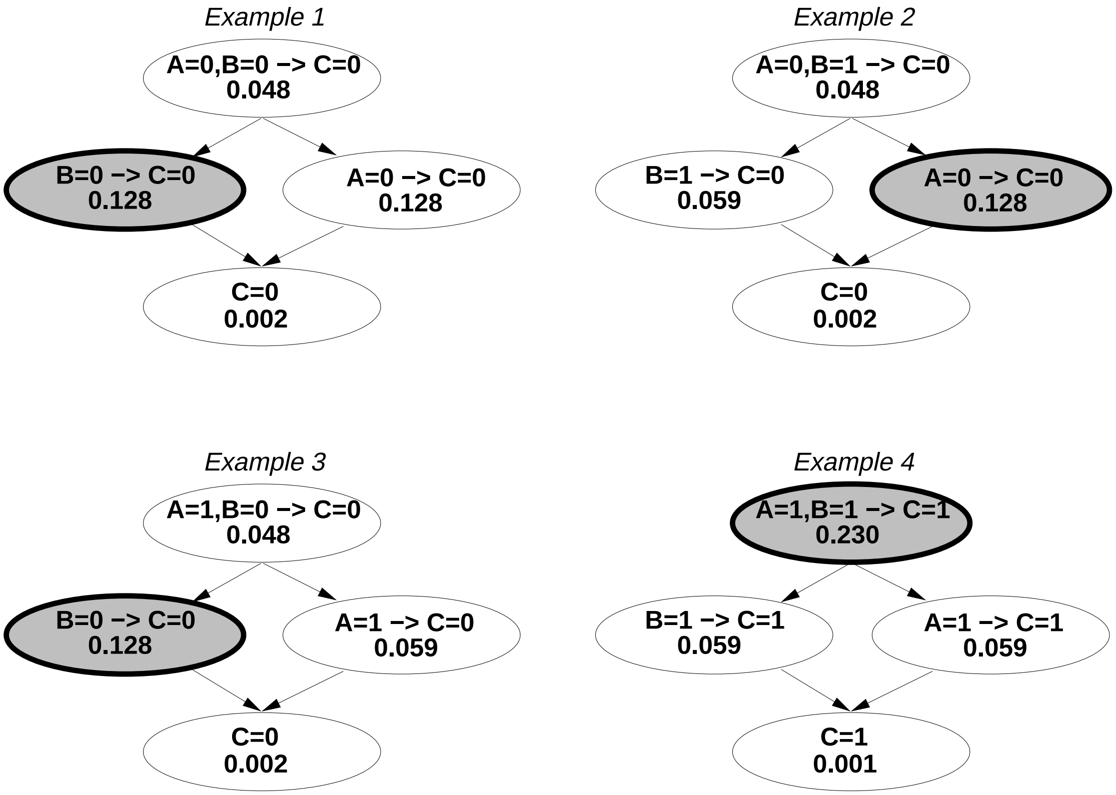
   (b) If two rules of the same order have the same J-measure, choose a random one.

4. If the chosen rule is not the parent rule, the chosen rule becomes a new parent rule; repeat the process starting at step 2. If the chosen rule is the parent rule, terminate.

For each example from the new training set, run the original SQUEEZE algorithm. The incremental algorithm is the same as SQUEEZE except that child-rules now include those generated by *adding back each previously removed attribute*; thus child-rules now have

Data Set:

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1



Final Rule Set:

1. **IF B=0 THEN C=0**
2. **IF A=0 THEN C=0**
3. **IF A=1 AND B=1 THEN C=1**

Figure 3.2: Simple example of SQUEEZE operation

order one more *or one less* than the parent rule. If the second training set was statistically similar to the first, the rules generated from the first set will not change. However, if the second training set changes the statistics of the examples, the rules generated from the first training set may have overgeneralized or overspecialized. This modification allows the rules to become either more general or more specific if necessary. J-measures in all cases are calculated with respect to the concatenation of the two training sets. Note that the algorithm cannot oscillate, since if the J-measures are the same, we favor the lower-order rule.

An example of the operation of the incremental algorithm is shown in Figures 3.3 and 3.4. Given only the first three examples of the AND truth table, the incremental algorithm in Figure 3.3 overgeneralizes to the belief that the output is always zero. When given the final (and pivotal) example in Figure 3.4, it is able to respecialize to find a good rule set.

3.2.2 Theoretical Justification

Will SQUEEZE find a 'good' rule set?

If the object of the rule-search algorithm is to find the R rules with the highest J-measures, a straightforward search such as the ITRULE algorithm is the only way to ensure success. However, we have already stated (Section 3.1.3) that while the set of rules which is best for classification is *among* those with the highest J-measure, some pruning of this set is required. Therefore, what we are really searching for is a set of rules which describes the data set completely and yet concisely. The SQUEEZE algorithm provides just such a set of rules. This can be seen as follows: since each example has a rule as a subset of its conditions, each example is covered. Thus the data set is described completely. In addition, each rule is generalized as much as possible by the use of the J-measure. Any duplicate rules are removed. This helps to fulfill the requirement that the rule representation be as concise as possible.

Optimality of SQUEEZE

The search for the best rule which can be made from a particular example goes on by picking the best among the child rules made by removing a single conjunct from the current parent rule. All the rules which can be made from an example can be found in a tree in which each level represents a different order of rule and each transition down represents the removal of a single conjunct. The search technique we have proposed makes the implicit assumption that the best rule in this tree is found along the path which has the highest J-measure at each step. But is this a valid assumption? Is is possible for the algorithm to get 'stuck' at a high-order rule and fail to find a better lower-order rule? We would *like* to prove the following conjecture:

> **Conjecture 1** The best rule among all the possible rules which can be made from an example is found along the path which has the highest J-measure at each step.

Unfortunately, this conjecture is false. It is possible, under certain conditions of the data, for the algorithm to get stuck at a high-order rule.

> **Counterexample** See Figures 3.5 and 3.6.

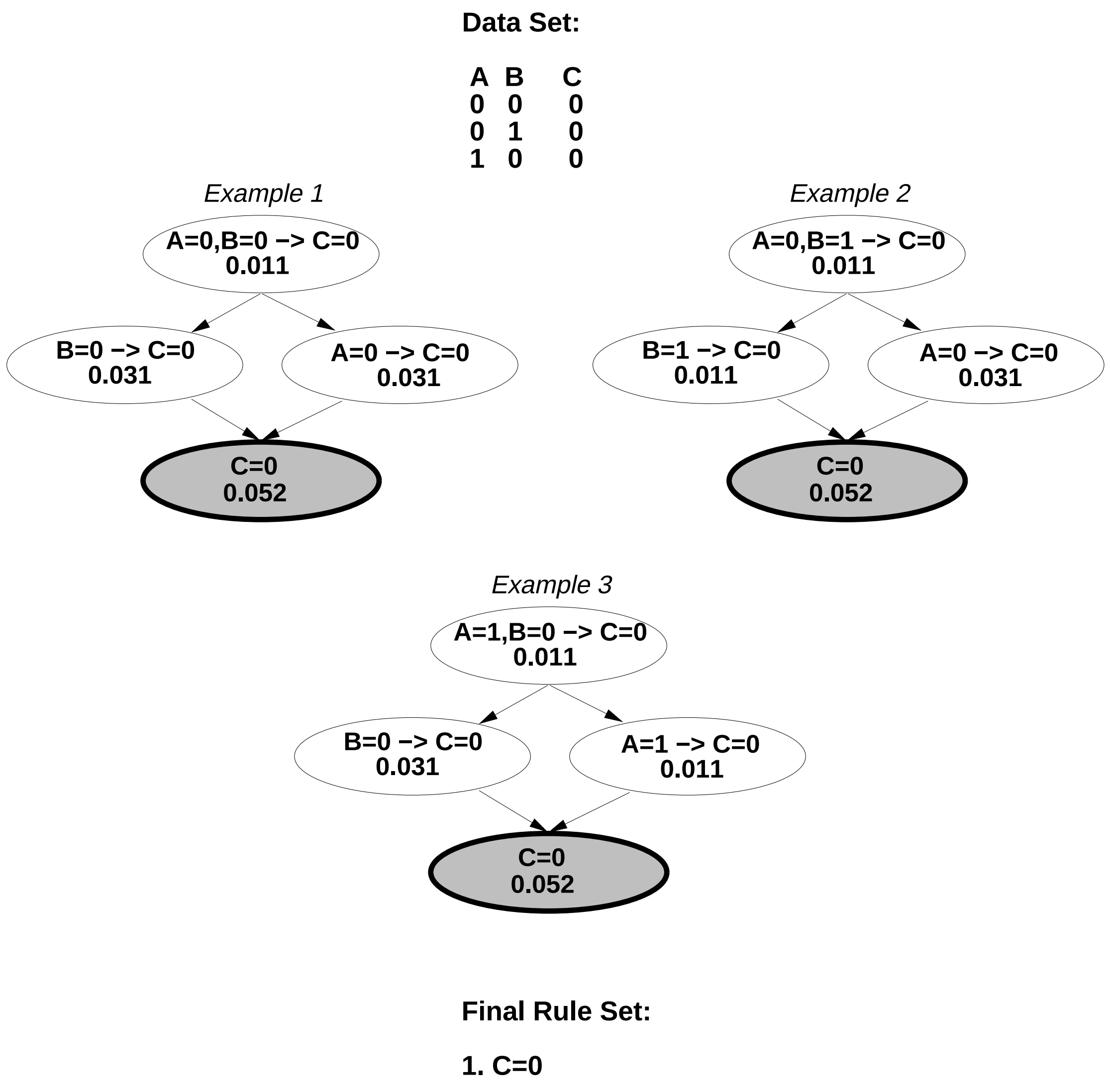
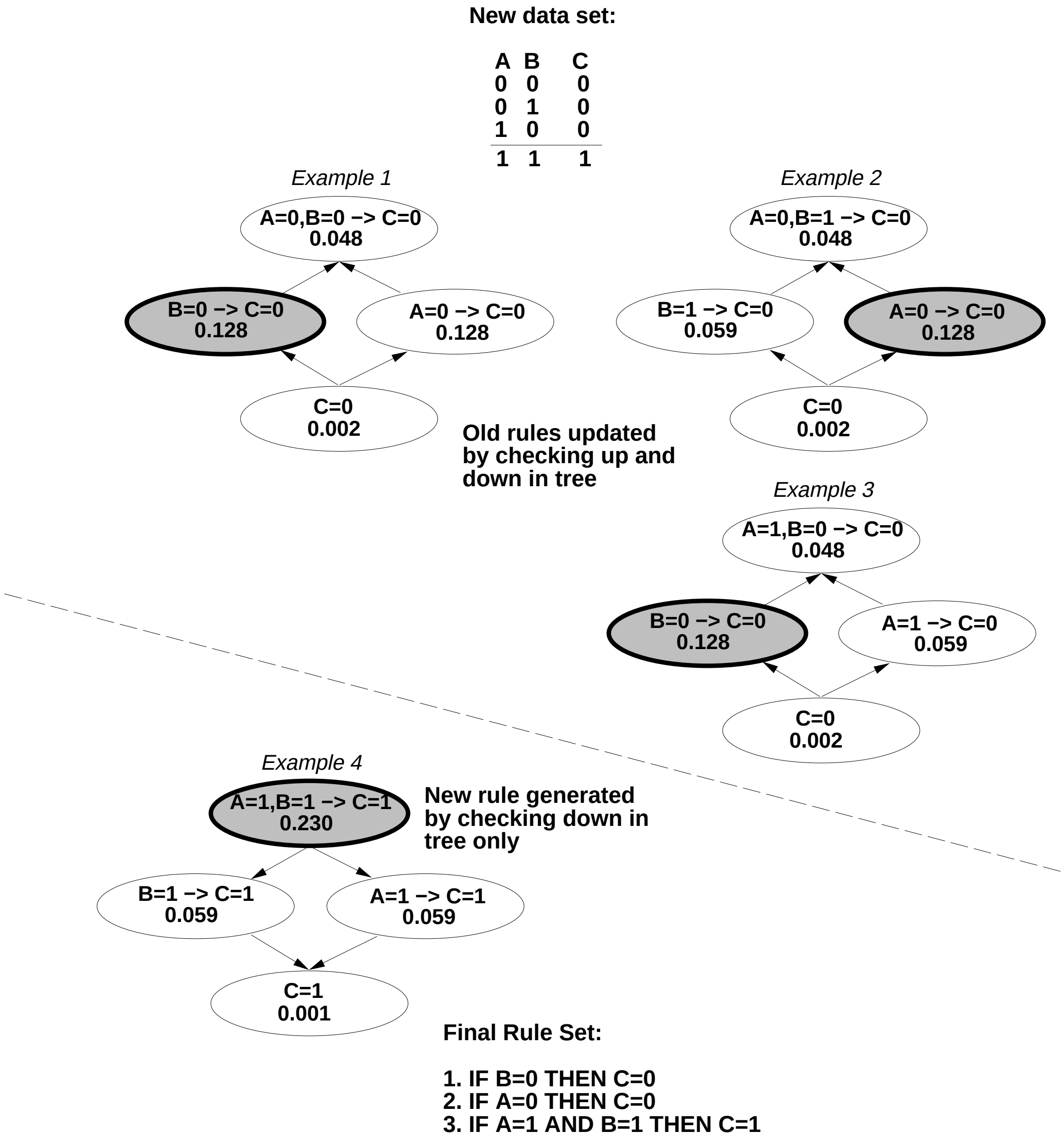


Figure 3.3: Incremental SQUEEZE overgeneralizing



New data set:

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Final Rule Set:

1. IF B=0 THEN C=0
2. IF A=0 THEN C=0
3. IF A=1 AND B=1 THEN C=1

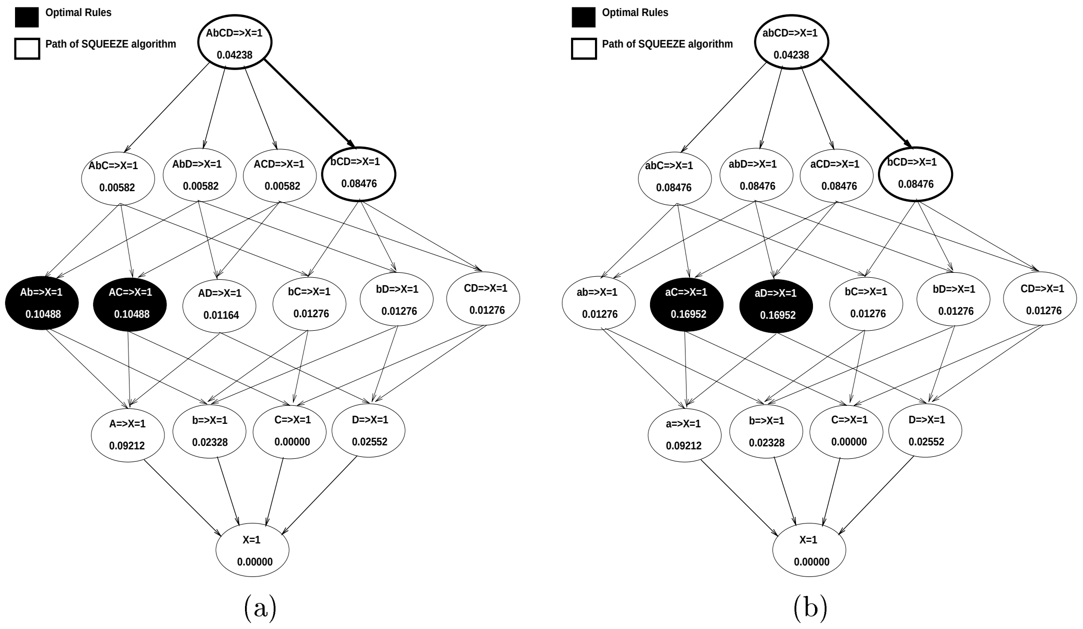
Figure 3.4: Incremental SQUEEZE recovering from overgeneralization

In Figure 3.5(a), the algorithm gets stuck because the J-measure actually misleads us. On the top level of the tree, the highest J-measure is that of the only rule which is not an ancestor of an optimal rule. In Figure 3.5(b), the algorithm gets stuck simply because it makes the wrong choice out of a number of equally valid rules. Does the algorithm get stuck simply because it is looking ahead only one step? The example of Figure 3.6 shows that even looking ahead two levels in the tree will not solve the problem. In general, we would argue that it is *always* possible to construct a pathological data set with N input attributes in which it is necessary to look ahead nearly N steps to find the optimal rule. Therefore, at least some of the rules we will find from this algorithm will be sub-optimal.

It is tempting to think of this problem as a tree search problem, as addressed in classical AI, but in truth it is an optimization problem. Tree search algorithms such as branch-and-bound and the A* algorithm [Nil80] are only useful when there is a *goal state* in the tree. In finding the 'best' rule, we are in truth optimizing our rule measure in a discrete space. Since no amount of lookahead less than the whole tree will work in every case, we choose the algorithm which takes the least amount of computation and accept sub-optimal performance. As it turns out, this is usually good enough.

	3	d	d	d	d	d	d	d	d	D	D	D	D	D	D	D	D
Input	2	c	c	c	c	C	C	C	C	c	c	c	c	C	C	C	C
	1	b	b	B	B	b	b	B	B	b	b	B	B	b	b	B	B
	0	a	A	a	A	a	A	a	A	a	A	a	A	a	A	a	A
Output		0	1	1	1	1	1	1	1	0	0	0	1	1	1	0	0

The data set



(a)

(b)

Figure 3.5: Counterexamples to Conjecture 1

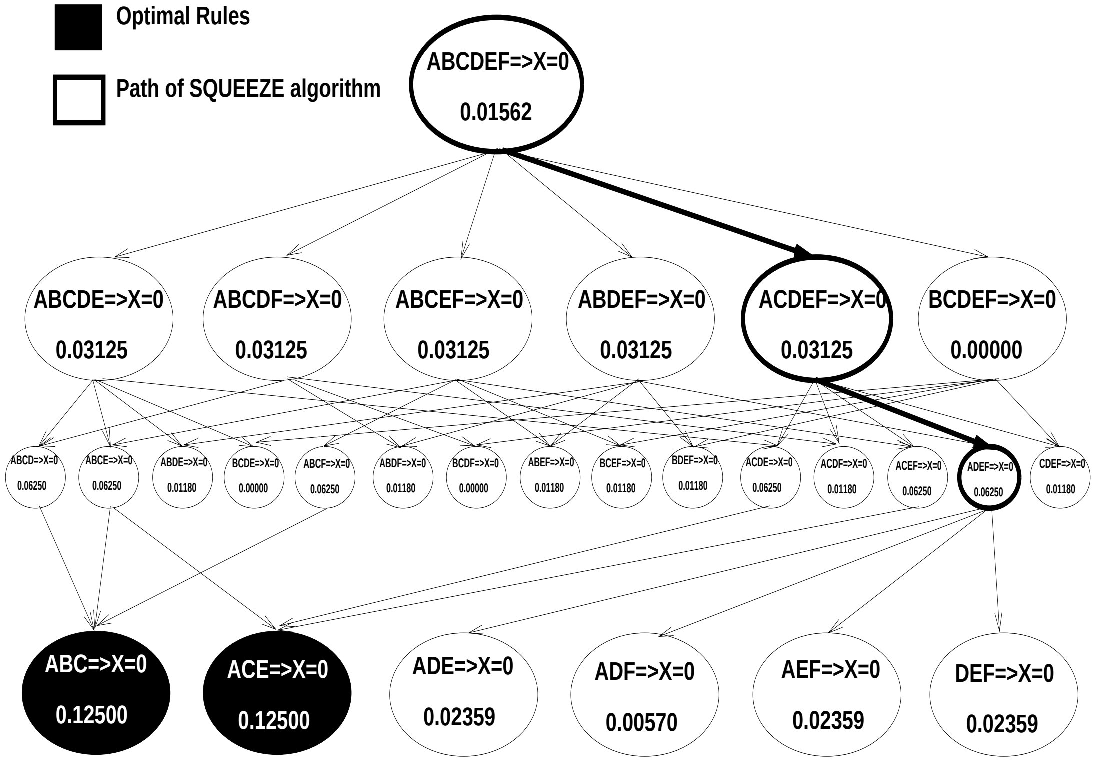
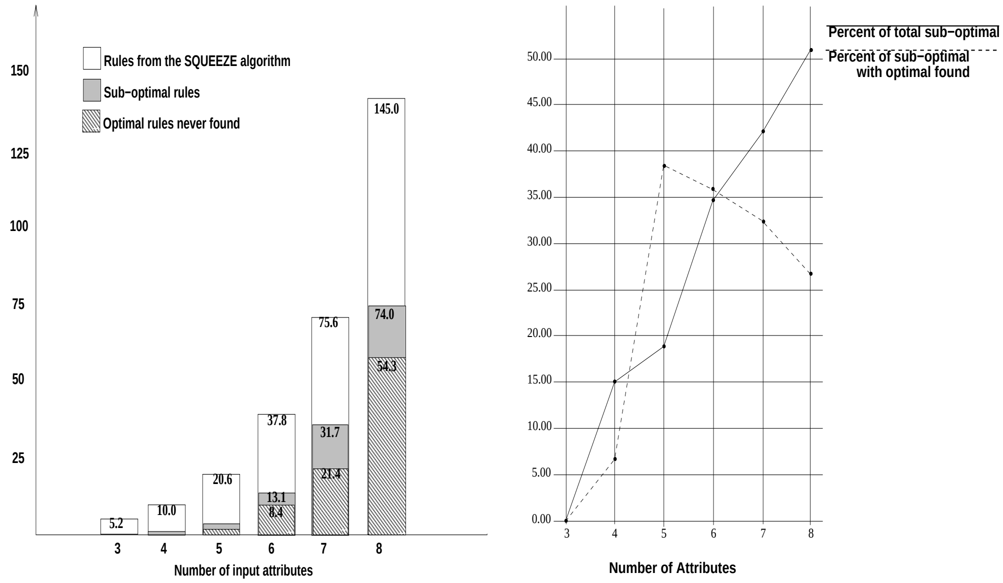


Figure 3.6: Even looking ahead two levels will not help

Looking ahead to Chapter 4, we can see that we still achieve near-optimal classification performance even with sub-optimal rules. How is this possible? There are several extenuating circumstances which help the algorithm to find a good rule set. First, there are a number of examples which can lead to any particular good rule. The lower the order of the rule, the more examples could yield it. Even if the expansion of one example fails to find the rule, it is possible that another will. Second, since the algorithm picks a random child if they have the same J-measure, it is likely that at least one example's expansion will lead to the good rule, solving the problem presented in Figure 3.5(b). Finally, a sub-optimal rule will likely have a lower transition probability. Thus, it will have less effect on the outcome of the classification.

When, and under what conditions, can this 'getting stuck' occur? It happens whenever, at some level in the search, the rule with the highest J-measure is not an ancestor of the best rule. The conditions on the data under which this can occur are too complex to be informative without making unrealistic assumptions about the data. The real questions are: will this happen often enough to impact our performance? Will it grow more or less severe as the number of attributes increases? Since a theoretical analysis does not yield any useful insight, it is worthwhile to conduct a simulation study to develop some practical insight into these questions.

The results of such a study are shown in Figure 3.7. A binary data set was generated with all possible combinations of input attributes and a random output (no functional dependence on the inputs) with an equal probability of being one and zero. The number of input attributes was varied from three to eight. For each data set, the number of



(a) (b)

Figure 3.7: Results of simulations with SQUEEZE

unique rules to come out of the SQUEEZE algorithm and the number of these which were sub-optimal were recorded.[3] In addition, a check was made to see which of the unfound optimal rules were found by the expansion of another example. The average over ten random data sets for each number of input attributes is shown in Figure 3.7(a). It is more informative, however, to look at the percentages shown in Figure 3.7(b); as a percentage, it is clear that the number of sub-optimal rules rises with the number of attributes and the number of optimal rules found by accident is steadily declining.

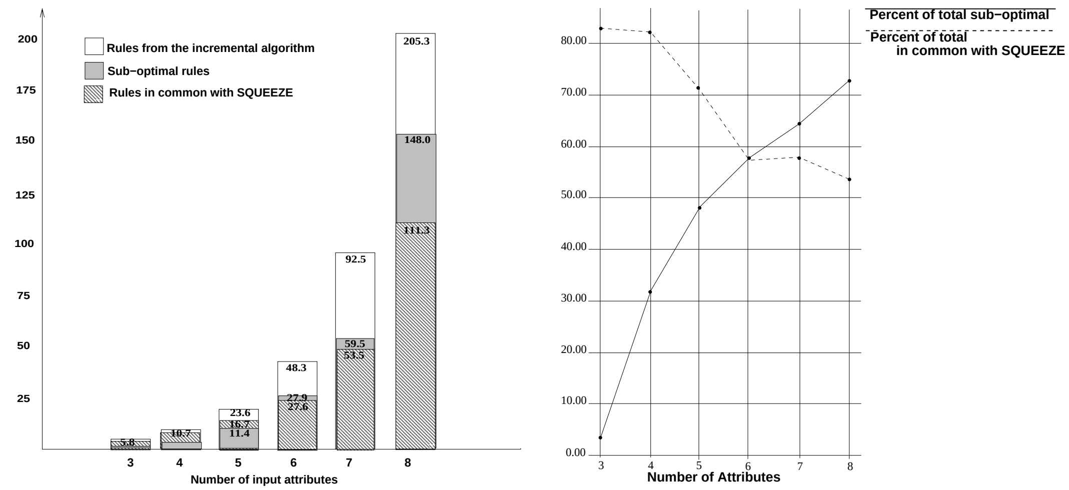
From this study, we can conclude that the problem of ‘getting stuck’ gets worse as the number of attributes increases. It should be expected that performance will worsen as the number of attributes gets very large. However, this rate at which this condition gets worse depends completely upon the data set.

Optimality of Incremental SQUEEZE

Suppose we are learning incrementally and, given a very small amount of input data, we seriously overgeneralize. Will the incremental algorithm get ‘stuck’ at a lower-order rule and fail to specialize? As discussed for regular SQUEEZE, it is *always* possible to construct a pathological data set such that this condition will occur. Since we are specializing as well as generalizing in this case, the performance study for the normal algorithm gives us little information. Again, the conditions on the data under which this can occur are too complex to be useful without unrealistic assumptions about the data. And so again, we must conduct a simulation study to determine how often the best rules are found.

The same binary data sets were used as in the first study, with all possible combinations of input attributes and an output with an equal probability of being one and zero. Again,

[3] It should be noted that SQUEEZE was predicting the output 100% correctly until the experiment reached 8 attributes; at 8 attributes, it averaged 99.6%.



(a) (b)

Figure 3.8: Results of simulations with Incremental SQUEEZE

the number of input attributes was varied from three to eight. The incremental algorithm was given each data set in ten steps, starting with one tenth of the data and working up to the whole set. For each data set, the number of unique rules to come out of the incremental algorithm and the number of these which were sub-optimal were recorded. These rules were matched with those from the regular SQUEEZE algorithm on the same data set to see how well the two matched. The average over ten random data sets for each number of input attributes is shown in Figure 3.8(a). Again, the percentages shown in Figure 3.7(b) are more informative; the number of sub-optimal rules rises with the number of attributes and the number of rules in common with SQUEEZE is steadily decreasing. Comparing the two studies, this algorithm produces not only a higher percentage of sub-optimal rules, but a higher number of rules.

From this study, we can conclude that the incremental algorithm will produce significantly less optimal rules than SQUEEZE (which is not unexpected). In addition, the rules from the incremental algorithm diverge from those from SQUEEZE as the number of attributes increase.

3.2.3 Algorithm Complexity

In this section, we compute the time complexity of the SQUEEZE algorithm to compare it to that of the original ITRULE algorithm and find out in what cases it is likely to be more efficient. In both algorithms, the actual time complexity depends largely upon the data set, so we will perform a worst-case analysis. In both analyses, let there be N m-ary input attributes and E examples. Let the arity of the output[4] also be m. The basic unit of time which we will use is an integer compare; everything else will be scaled appropriately.

We begin with the ITRULE algorithm, assuming (the worst case) that the J-measure bounds never prevent the algorithm from searching the entire tree of possible rules to the depth d, the maximum order specified by the user. To compute the J-measure for an i-th

[4] We define the ‘arity’ of a discrete-valued variable as the cardinality of the set of possible values of the variable.

order rule, it takes

$$J(i) = (i+1)E + CE + D$$

where the first term is comparisons for rule matching, the second term is a fixed number of adds to update the counters, and the third term is the J-measure calculation, which takes constant time. The ITRULE algorithm does a depth-first search, expanding each of the Nm^2 first order rules into a tree of higher-order rules. In doing this, it calculates the J-measures for every possible rule up to d-th order. To simplify our analysis, let us assume that $d < \lfloor \frac{N}{2} \rfloor$; this is a reasonable assumption for all but the smallest of data sets. Given this assumption, the dominating time in this expansion is in the computation for the d-th order rules, due to their large number. The time it takes to calculate the J-measures for every d-th order rule is

$$T_{d\text{-}th\ order} = m^{d+1} \left(\begin{array}{c} N \\ d \end{array} \right) J(d) = m^{d+1} \frac{N(N-1)...(N-(d-1))}{d!}((d+1)E + CE + D).$$

In fact, the whole algorithm is dominated by this term and the worst-case time complexity of ITRULE is

$$T_{ITRULE} = O\left(\frac{Em^{d+1}N^d}{(d-1)!}\right).$$

Thus ITRULE is linear in the number of examples (since only the J-measure calculation depends upon E) and exponential in the number of attributes and the arity of each attribute. In the maximum rule order, the time complexity of ITRULE goes up somewhat less than geometrically.

The SQUEEZE algorithm calculates the J-measure in the same manner as ITRULE, so $J(i)$ is still valid. To calculate the best rule for a particular example in the worst case takes

$$\begin{aligned} T_{best\ rule} &= NJ(N-1) + (N-1)J(N-2) + ... + 2J(1) + J(0) \\ &\quad +FN + F(N-1) + ... + 2F + F \end{aligned}$$

where the first N terms are the calculation of J-measures for each successive step in the algorithm, and the second N terms are comparisons to find the maximum for each expansion step. Expanding this, we find that $T_{best\ rule}$ is dominated by the J-measure calculations, and

$$\begin{aligned} T_{\text{best rule}} &\geq N(NE + CE + D) + (N-1)((N-1)E + CE + D) + ... + (E + CE + D) \\ &= O\left(EN^3\right). \end{aligned}$$

In order to calculate the best rules for each example in the data set sequentially, we must multiply this time by E, and thus

$$T_{SQUEEZE} = O\left(E^2N^3\right).$$

Of course, there are two ways to avoid this final multiplicative term. First, the examples may be expanded in parallel, since each one is expanded independently. Second, if there are a large number of examples, you need not make a rule from every one; there are only so many rules. This algorithm scales cubically with the number of attributes, linearly or as the square of the number of examples (depending upon the implementation), and does not depend at all on the arity of each attribute.

Incremental SQUEEZE does N J-measure computations and compares at each step, but the above complexity computation holds for it also. Thus also

$$T_{INCREMENTAL} = O\left(E^2 N^3\right).$$

Quinlan [Qui91] suggests that an algorithm of order $O(E^2)$ or more is unlikely to scale up. He suggests that $O(E\log(E))$ is the highest order which is practical for large data sets. By this criterion, both ITRULE and SQUEEZE are practical under certain conditions. The reader should also keep in mind that the time complexities computed above are worst-case. In practice, the ITRULE algorithm will often take its worst-case time, while the SQUEEZE algorithm will seldom proceed all the way down the tree to a zero'th-order rule.

Given these complexities, when should one algorithm be chosen over the other? Of course, this choice depends upon many other factors; but based upon complexity alone one should choose ITRULE over SQUEEZE when the maximum rule order needed is less than three. One should choose SQUEEZE over ITRULE when the arity of the attributes is high, or higher-order rules are necessary.

3.2.4 Summary

The SQUEEZE algorithm presented in this chapter does offer significant advantages to the straightforward ITRULE search technique. It allows the induction of a concise yet complete rule set without any need for rule pruning. In the case of data sets of higher arity or in which a rule order greater than two is necessary, SQUEEZE presents a computationally superior method for finding rules. However, as the number of attributes in the data set increases, the number of sub-optimal rules resulting from the SQUEEZE algorithm also increases. It should be expected that as the number of attributes grows very large, SQUEEZE generalization performance will become unsatisfactory. The incremental extension to the SQUEEZE algorithm, while still providing a reasonable rule set, produces even more sub-optimal rules than the SQUEEZE algorithm. As the number of attributes increases, the rules from incremental SQUEEZE diverge from the rules of regular SQUEEZE. To maximize the future performance of incremental SQUEEZE, it should be given as good an initial idea of the statistics of the data as possible.

Chapter 4
Experimental Results

In this chapter, we show the results of experiments with the rule-based classifier network to verify its effectiveness as a classifier, test the ability of the incremental algorithm, and demonstrate the explanation ability we have touted.

4.1 Comparison with Other Approaches

To verify the effectiveness of SQUEEZE as a classifier, we will compare its performance with several other classification schemes. The classical first-order Bayes classifier was chosen for comparison as an example of a system which does not use higher-order information. However, this system is very simple and performs quite well for many data sets. The backpropagation network was chosen as a very successful 'black box' approach to neural network learning which uses higher-order information but is opaque to the user. Also shown for comparison is a trivial method in which the class that appears most in the examples is always picked (i.e., the most likely *a priori* class).

Three data sets were chosen for training. The first of these is the well-known LED digits problem in which the system must decide which digit is being shown on a 7-segment display given the value of each segment. Noise has been added to the examples so that the optimal classification rate is about 74%. The second data set consists of 435 voting records from a session of the 1984 US Congress [vot85]. Each example corresponds to a particular congressman and the attributes correspond to their votes on 16 different issues. The system must decide on the party affiliation of each congressman. (Not surprisingly, it is possible to predict this very well.) The third and final data set has been generated from the following Boolean function:

$$x = (y1 \oplus y2) + (y3 \cdot y4) + (y5 \cdot y6)$$

where $\oplus$ is exclusive OR, $\cdot$ is AND, and $+$ is OR. To introduce noise, the class variable x has a 10% chance of being 'reversed' from its true state. Thus the optimal rate on this data set is 90%. This data set is designed to require the use of second-order information.

The data sets were divided into disjoint training/testing sets for generating comparative results. The splits were LED: 154/846, Voting: 200/235, and Boolean: 576/64. In each case, the training examples were chosen at random from the entire data set and the testing set was the remainder. Ten random runs were averaged to obtain the results shown.

	Optimal	Maximum	Back	First-order	Rule-based
	Rate	a priori	Propagation	Bayes	Neural Network
LED	74.0%	17.0%	68.4%	68.2%	69.5%
Voting	96.0%	53.3%	93.6%	91.1%	89.3%
Boolean	90.0%	67.3%	90.0%	67.5%	89.7%

Figure 4.1: Performance comparison

It is clear that the rule-based neural network compares with the other classification schemes. It performs about as well as the backpropagation algorithm on all data sets, and succeeds in finding the higher-order representation that the Boolean data set demands and that the first-order Bayes classifier is incapable of generating.

4.2 Performance of the Incremental Algorithm

We now explore the incremental algorithm's performance and the evolution of the rule-network's size as it is 'grown' example by example.

For this experiment the Boolean data set described in the previous section was used. While testing on 640 examples, training examples were given one at a time to the learning algorithm. Classification performance and network size were checked periodically.

The plot of Figure 4.2 shows that the classification performance quickly approaches the optimal classification rate of 90%. We can see in Figure 4.3 the growth of the network's conjunctive second layer as more examples are presented to it. Notice that the network quickly approaches a size of about 30 units after 200 examples, at the same time that the classification performance reaches 90%. It slowly builds to about 35 units after all 640 examples.

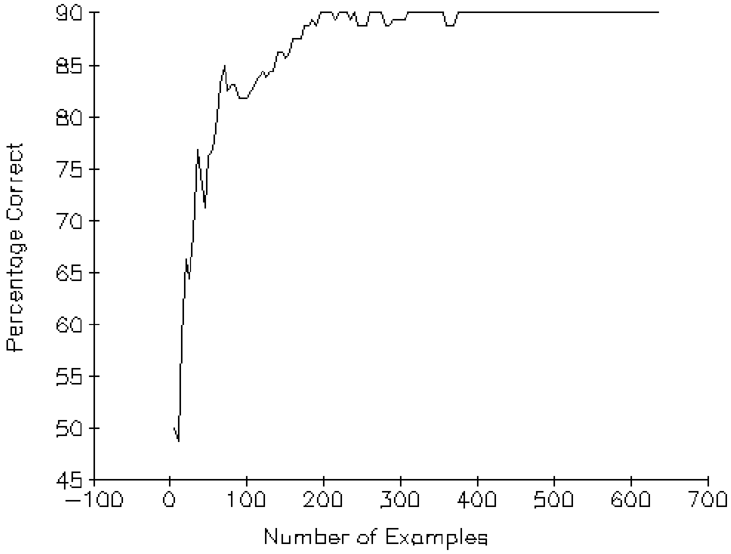


Figure 4.2: Incremental performance

4.3 Explanation Ability

In this section, we will demonstrate with a simple example the explanation ability that we have motivated rule-based systems with. Let us consider a simple data set which describes 90 animals in a zoo with 18 attributes including whether they have hair, feathers, teeth, a backbone, fins, a tail, venom, wings, and so forth. A small sample of the data set is

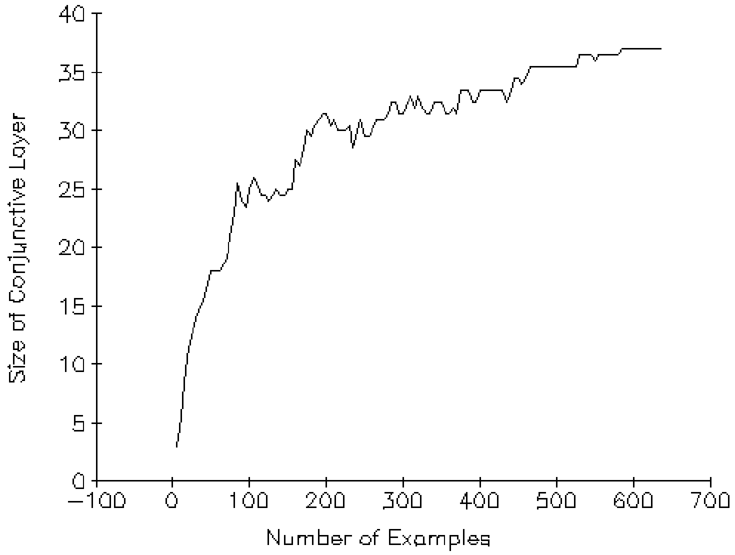


Figure 4.3: Growing the network incrementally

shown in Figure 4.4. The animals are separated into seven classes: mammal, bird, snake, fish, amphibian, insect and crustacean.

On this data set, the SQUEEZE algorithm classifies 89 out of 90 animals correctly. Let us look at a sample of the criteria it uses for fish and crustaceans.

See Figure 4.5 for the two rules about fish. These rules are both perfectly correct and have the same J-measure. The system has discovered that egg-laying is the most important criterion for determining fish. If, in addition, the creature has fins or has a tail but does not breathe air, it is a fish. (Dolphins and sea lions have fins, but are not fish.)

For crustaceans, the five rules may be found in Figure 4.6. In this case, there are very few examples so the system must make more complex rules. There is an interesting special case (rule 5). There are only two eight-legged creatures in the data set, and both are classed as crustaceans. Thus the system creates this rule, but gives it a low J-measure, since it is only valid for two examples.

Not only do the rules explain the data set as a whole, but they allow the explanation of each individual decision; for example, let us consider the case of the ostrich, which doesn't fly, yet is classified as a bird. The rules which fire are shown in Figure 4.7. Rule 4 argues that since the creature doesn't fly and yet breathes air, it must be a mammal. Luckily, there are several other criteria which mark it as a bird, including feathers and egg-laying.

Overall, the data set can be quite well understood from the rules describing it. This is the true hope of the rule-based systems we have proposed: achieving excellent classification performance while allowing explanation of these two kinds.

Name	Attributes																Class
aardvark	T	F	F	T	F	F	T	T	T	T	F	F	4	F	F	T	mammal
antelope	T	F	F	T	F	F	F	T	T	T	F	F	4	T	F	T	mammal
chicken	F	T	T	F	T	F	F	F	T	T	F	F	2	T	T	F	bird
crow	F	T	T	F	T	F	T	F	T	T	F	F	2	T	F	F	bird
pitviper	F	F	T	F	F	F	T	T	T	T	T	F	0	T	F	F	snake
seasnake	F	F	F	F	F	T	T	T	T	F	T	F	0	T	F	F	snake
herring	F	F	T	F	F	T	T	T	T	F	F	T	0	T	F	F	fish
pike	F	F	T	F	F	T	T	T	T	F	F	T	0	T	F	T	fish
frog	F	F	T	F	F	T	T	T	T	T	T	F	4	F	F	F	amphibian
newt	F	F	T	F	F	T	T	T	T	T	F	F	4	T	F	F	amphibian
flea	F	F	T	F	F	F	F	F	F	T	F	F	6	F	F	F	insect
gnat	F	F	T	F	T	F	F	F	F	T	F	F	6	F	F	F	insect
crayfish	F	F	T	F	F	T	T	F	F	F	F	F	6	F	F	F	crustacean
lobster	F	F	T	F	F	T	T	F	F	F	F	F	6	F	F	F	crustacean

Figure 4.4: The zoo data set

Number	Conditions			Conclusion			$p(X\vert y)$	J-measure
1	IF	eggs	T	THEN	Class	fish	1.000	0.388
	AND	breathes	F					
	AND	tail	T					
2	IF	eggs	T	THEN	Class	fish	1.000	0.388
	AND	fins	T					

Figure 4.5: Rules about fish

Number	Conditions			Conclusion			$p(X\vert y)$	J-measure
1	IF	breathes	F	THEN	Class	crustacean	1.000	0.258
	AND	tail	F					
2	IF	toothed	F	THEN	Class	crustacean	1.000	0.258
	AND	breathes	F					
3	IF	airborne	F	THEN	Class	crustacean	0.818	0.252
	AND	backbone	F					
4	IF	backbone	F	THEN	Class	crustacean	0.562	0.168
5	IF	legs	8	THEN	Class	crustacean	1.000	0.074

Figure 4.6: Rules about crustaceans

Number	Conditions			Conclusion			$p(X \vert y)$	J-measure
4	IF	airborne	F	THEN	Class	mammal	0.720	0.157
	AND	breathes	T					
7	IF	eggs	T	THEN	Class	bird	1.000	0.464
	AND	legs	2					
8	IF	toothed	F	THEN	Class	bird	0.947	0.405
	AND	venomous	F					
	AND	tail	T					
9	IF	milk	F	THEN	Class	bird	0.900	0.367
	AND	venomous	F					
	AND	fins	F					
	AND	tail	T					
10	IF	feathers	T	THEN	Class	bird	1.000	0.464
11	IF	milk	F	THEN	Class	bird	1.000	0.464
	AND	legs	2					

Figure 4.7: Rules which fire for an ostrich

Chapter 5
Future Work

In this chapter, we describe possible improvements to the systems we have proposed and promising avenues of research in classification with rule-based systems.

5.1 Improvements to the SQUEEZE Algorithm

The SQUEEZE algorithm has been implemented in the most straightforward way possible. It is likely that there is a way to perform this algorithm while calculating fewer probabilities. The hashing techniques suggested by John Miller [Mil93] might provide a starting point.

SQUEEZE does tend to get stuck in local minima in data sets with a large number of attributes. Perhaps some sort of improved search technique would allow the algorithm to skip over such minima without becoming computationally impractical.

5.2 Improvements to the Probabilistic Framework

If one of the classes is very likely *a priori*, it is very difficult to achieve classification performance better than just taking the most likely *a priori* class. This is due to our probabilistic handling of the data set. In general, it is found that our algorithms perform better if all values of all attributes are equally likely. Duplication of the examples for less common classes has been discussed to solve this problem, but may throw off the statistics of the input attributes.

5.3 Numeric Inputs

A number of techniques have been tried for dealing with numeric inputs. The first is clustering the inputs; this usually requires the choice of some ad hoc parameters such as the number of cluster centers or maximum cluster size. The second is quantizing the input space using an algorithm which optimizes the signal-to-quantization noise ratio (SQNR). We have tried combining these two approaches by doing a k-means clustering varying k from 1 to 10, and then picking the quantization with the highest SQNR.

A new approach which is gathering momentum is the use of fuzzy logic to partition the input space [Bez92]. Perhaps the idea of SQNR can be combined with the fuzzy approach.

5.4 Learning Important Features

Quite often, a classifier is given far more inputs than are actually necessary to make a classification decision. It would be nice if the classifier system could make it obvious that certain inputs are not being used and simply ignore them. The rule set learned for a classification problem gives a strong indication as to which inputs are useful. If an input is never (or very seldom) used, perhaps it can be dropped altogether.

5.5 Non-box Rules

To overcome the axis-parallel problem (see Section 2.1), perhaps rules could be formulated in a slightly different manner, such that they no longer specify structures parallel to the input axes. Rules with forms such as

```
if (a>b) then class=1
if (0.5*a-1.5*b > 2.0) then class=1
if a=red and b=blue then class=f(c*red+d*blue)
```

might be attempted. Of course, this will require a reevaluation of the techniques used for rule valuation and search.

5.6 Multi-Layered Rule Networks

The rule-based classifier systems we have described draw conclusions directly from the inputs about the class. It is possible that for some data sets superior performance might be obtained by drawing conclusions about an intermediate variable which could then be used to make rules about the class. In some data sets, these intermediate variables are obvious; but in general, discovering such variables is an unsolved problem.

Part II

Function Approximation with Rule-Based Networks

Chapter 6
Introduction to Function Approximation

You can't pass a single day without seeing one, and yet most people don't even realize they exist. Automatic control systems are everywhere, from the thermostat in your living room to the cruise control system on your car to the autopilot on every airplane. Today, they are even springing up in such unexpected places as clothes drying machines, cameras, and even toasters.

The more we demand from our devices, the more control systems become a necessity. Control systems allow a physical system to come to a desired state and maintain that state as the conditions on the system vary. Typically for an industrial control system, a mathematical model of the process being controlled (referred to as *the plant*) is formulated. Given this model, a great body of theory exists to allow an *optimal* controller to be designed. An optimal controller will control the plant as well as it can be controlled, given the limitations of the physical system. The parameters of a real plant will change, of course, as the system gets older or if it becomes damaged. There is another body of theory for *adaptive* control, which allows a controller to change to fit the plant it is controlling.

What happens if a sufficiently good mathematical model for the plant cannot be obtained because the plant is too complex or not well understood? In the interesting case of autonomous robotic systems, this is often a problem. In this case, control theory fails us and we are left with *learning control systems.* Learning control systems try to optimize some performance criterion and obtain numerically a controller which will perform as well as possible. It is usually not possible to obtain an optimal controller, but adaptive control is a necessity.

Function Approximation is synthesizing a complete function from samples of the function and its independent variables. As a general mathematical problem, function approximation has been studied for centuries. However, some knowledge of the function to be approximated is usually assumed. In control systems, a function approximator is handy when one has only examples of the output of a good control system. Usually one knows only that the function is smooth; aside from this, no knowledge of the function is assumed. The ability to take examples of good control outputs under these conditions and form a complete control surface from them is essential to many learning control paradigms.

In Part II of this thesis, we consider the extension of our rule-based philosophy to the function approximation problem and its application to control.

6.1 Defining Function Approximation

The problem of function approximation is similar to that of classification, except that both the inputs and the output are numerical. The problem is still to learn to predict the output given the inputs by looking at examples of the inputs and output. However, in this case the question arises as to the behavior of the approximation with noisy input data. Recall in the classifier case that we found it necessary to ignore statistically insignificant examples in order to bring out the underlying patterns in the data. What assumptions should we make about the numerical data to allow the smoothing of noise? For the type of problems to which we will apply our function approximator, the assumption that the input data is *noiseless* is sufficient. We will assume the data is perfectly correct and attempt to approximate it as closely as possible. For a discussion of the application of our function approximation method to noisy data, see Section 10.1.4.

Another difficulty arises because the attributes involved are numerical: characterizing a 'good' solution to this problem is more difficult than in the case of classification. The most common goodness measure is RMS error from the example points. Similar to our assumption in Part I, we will assume no knowledge of the mathematical form of the function to be approximated. Because of this assumption, the only information we have about the function to be learned comes from the example points. We will assume that a linear interpolation between example points would yield the true function, i.e., that the function does not 'jump' between example points; any large gap in the examples will simply be smoothed over. Thus since our approximation interpolates smoothly between example points, the RMS error in our approximation from the example points is the best measure of 'goodness.' Because of this assumption, the input to the system must be representative of the structure of the function to be approximated. Where there is more complexity in the function, more example points should be located.

6.2 Choosing an Approximator

When one makes no assumptions about a model of the function to be approximated, mathematical theory can only provide interpolation techniques such as splines for function approximation. Under this assumption, we are left with so-called 'model-free' systems. These systems include neural networks, memory-based systems, and fuzzy systems. Neural networks have provided at least two good function approximator systems: the radial basis function network [PG90] and the backpropagation network [RHW86]. The radial basis function network works by smoothing between carefully chosen exemplars, but does not scale well to higher dimensions. The backpropagation network performs a gradient descent to find the best approximation, but the necessity to specify the network structure beforehand and the difficulties in convergence make this network difficult to use. Memory-based systems, which operate by remembering every example presented to them [MA92], accomplish the function approximation task without learning. Fuzzy systems are capable of function approximation and also have other advantages over the other systems mentioned.

A fuzzy system expresses the function approximated in terms of linguistic rules; this allows a knowledge of what has been learned which is far in advance of what one may learn from looking at the weights of a neural network. In control systems, this becomes a definite advantage. At any time, the output of the controller can be traced back to a set of rules which govern its behavior in that state. A particular undesirable behavior of a controller can be modified simply by changing or deleting a rule which leads to that behavior. In addition, due to the use of an independent membership function representation for each variable, fuzzy systems tend to scale well to higher dimensions. Fuzzy logic, the basis for fuzzy systems, is described in the next chapter.

Chapter 7
Fuzzy Logic

In order to extend our rule-based philosophy to function approximators, we need a method for expressing numerical quantities in terms of linguistic rules. Fuzzy logic, proposed by Zadeh in 1965, provides an ideal framework for doing just that.[1]

It was Aristotle who formalized in his "Laws of Thought" [Kor67] the binary-valued logic which is prevalent in computer systems today. However, this logic was not universally accepted. The Greek philosopher Plato provided an alternative by indicating that there was a third region between true and false where the opposites "tumbled about." Multivalued logic surfaced again with the Heisenberg uncertainty principle of quantum mechanics, from which again arose the idea of a three-valued logic allowing a value between true and false. Lukasiewicz [Lej67, 1930's] first formalized a three-valued logic, and also explored four- and five-valued logics. Black [Bla37] was the first to use a continuous-valued logic, calling the uncertainty of his constructions 'vagueness'; but it was Lotfi Zadeh [Zad65] who first formalized this idea by extending traditional set theory and calling it *fuzzy set theory.*

In traditional set theory, an object x is either a member of a set S or not. We may define a characteristic function $\mu_S(y)$ which is one if y is a member of S and zero if not. In fuzzy set theory $\mu_S(y)$ may take on any value between zero and one (inclusive). Thus x has a *degree of membership* in S equal to $\mu_S(x)$. $\mu_S(y)$ is called the *membership function* for the fuzzy set S.

It is important to understand the difference between probability theory and fuzzy set theory. Probability expresses the likelihood, over repeated trials, that an event will occur. Fuzzy set theory allows the expression that, *in a single trial*, some vagueness exists about whether an event has occurred. Probability theory is *objective*, in that it can be tested by experience [KB78], whereas fuzzy set theory is *subjective*, based upon the belief which a given person has that an event has occurred. For example, suppose we are told a probability:

$$p(\text{John is tall}) = 0.7.$$

Then we expect that if we look at 100 Johns, about 70 of them will be tall. Suppose, however, that we are told a membership:

$$\mu(\text{John is tall}) = 0.7.$$

In this case, we look at a single John, and we see that he is about six feet tall. He is not extremely tall, but neither is he short. Thus he has about 0.7 membership in the set of tall people.

Fuzzy logic seems to model well the patterns and vagueness of human thought; ideas such as a warm day, a fast car, and a high price can all be elegantly expressed in terms of fuzzy sets. This has lead to widespread interest in the use of fuzzy logic for expert systems [Gra91]. Even more prevalent is the use of fuzzy logic in control systems. A fuzzy logic control system is designed so as to model the *operator* of a system rather than the system itself, as control theory does. This fascinating idea has led to millions of dollars

[1] The history section relies heavily on those of Brulé [Bru92] and Kosko [Kos92].

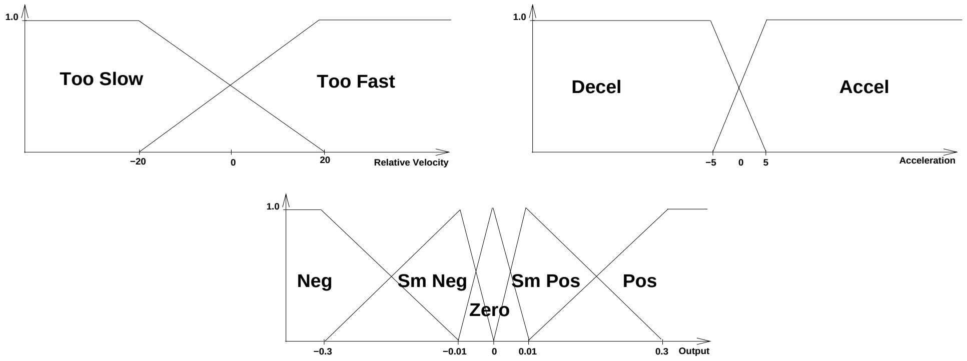


Figure 7.1: Membership functions for fuzzy speed control system

of products, particularly in Japan, and is the motivation behind the use of fuzzy logic in this thesis.

In a fuzzy control system, each variable is assigned a set of membership functions which partition its range into overlapping regions. Rules can then be made about these regions and a smooth interpolation between these rules will lead to a continuous output. Membership functions, fuzzy rules, and methods for doing inference with the rules are discussed below.

7.1 Membership Functions

We will formalize the concept of a membership function by using the concept of a *fuzzy number* [KG85]. A linguistic example of a fuzzy number is "around seven." In order for a fuzzy set to qualify as a fuzzy number, its membership function must satisfy two criteria: *normality* and *monotonicity.* The normality property simply requires that a membership function take its maximum value at one. The monotonicity property requires that we define a center C around which the fuzzy number is based. The membership function must then be monotonically non-decreasing from $-\infty$ to C, and monotonically non-increasing from C to ∞. Any membership function which satisfies these properties will be suitable for our purposes.

There are several membership function shapes which are commonly used. Most common is the triangular shape, which increases linearly from some point to the center C and then decreases linearly to some point. Almost as common are trapezoidal membership functions, which are triangular except for a flat area in the center. Continuous functions such as Gaussians are also used, but usually for theoretical analysis rather than actual practice.

In function approximation, a set of membership functions is specified by the designer of the system for each attribute, input and output. The set of membership functions (each of which can be given a descriptive name) define fuzzy sets that the attribute can be in. For example, see Figure 7.1; the membership functions for a speed control system are shown. The fuzzy sets for the output are {neg, sm_neg, zero, sm_pos, pos}. Thus an output of 0.02 has membership 0.9 in sm_pos, 0.1 in pos, and 0.0 in all the other sets. Due to the overlap in the areas shown for each set, an attribute is never *entirely* in one set or another.

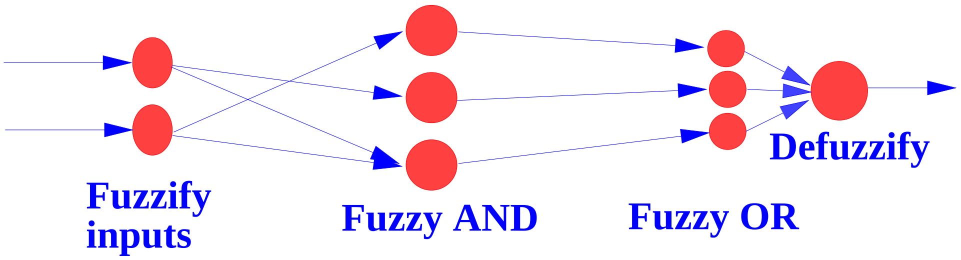


Figure 7.2: A fuzzy system

7.2 Fuzzy Rules

A fuzzy rule may be defined as

$$\text{if } y \text{ then } X$$

where y (the condition side) is specified as a conjunction in which each clause specifies an input attribute and one of the fuzzy sets associated with it, and X (the conclusion side) specifies an output variable fuzzy set. There may be at most one clause for each input variable. Thus a possible rule for the speed control system would be

```
IF relative_velocity=too_slow AND acceleration=decel THEN output=pos
```

Such rules as these specify the value of the approximator output at a point, line, plane, or hyperplane in the input space, depending on the number of conditions. If a set of rules has a clause for **every** input attribute on the condition side of **each** rule, it is referred to as a *cell-based* rule set, because any combination of membership functions for every input attribute defines a *cell* in the input space. The input membership functions partition the input space into disjoint cells; each cell corresponds to a point in the input space and the region of influence of the rule centered there.

7.3 Fuzzy Function Approximation

The key to fuzzy systems is that no single rule is used to determine the proper course of action at any time; rather all rules are used *to some degree*. It is the tradeoff between rules with different conclusions that allows a smooth variation of the output.

To determine the (numerical) output, the memberships of each attribute are determined. (See Figure 7.2 for a graphical representation of the dataflow in this system.) This 'fuzzifies' all numerical inputs. Each rule in the system fires to a certain degree, determined a fuzzy AND of the memberships in the conditions of the rule. For example, the above example rule for the speed control system will fire to the degree of the fuzzy AND of the membership of relative velocity in 'too slow' and acceleration in 'decel.' When a rule fires, it adds a weight proportional to its degree of firing to the conclusion that the output is in the set it concludes. The weight for each output set is calculated as a fuzzy OR of the weights from all the rules which conclude that set. The numerical output is calculated as a function of these weights. This step is known as defuzzification, because it goes from a weight in each fuzzy set to a crisp numerical value, the opposite of the initial fuzzification step.

The specifics of fuzzy AND, fuzzy OR, and the output calculation are discussed below.

7.3.1 Firing the Rules: Fuzzy AND

A conjunction of two fuzzy numbers should be nonzero only when both of the fuzzy numbers are nonzero. The conjunction should be low when either fuzzy number is low and high only when both are high. A class of functions which satisfy these properties are the triangular norms [Lee90]. The greatest triangular norm is the minimum, and the least is the drastic product. Some triangular norms are:

$$\begin{array}{ll}
\textbf{Minimum} & \mathrm{AND}(x, y) = \min(x, y) \\
\textbf{Algebraic Product} & \mathrm{AND}(x, y) = x * y \\
\textbf{Bounded Product} & \mathrm{AND}(x, y) = \max(0, x + y - 1) \\
\textbf{Drastic Product} & \mathrm{AND}(x, y) = \begin{cases} x & y = 1 \\ y & x = 1 \\ 0 & x, y < 1 \end{cases}
\end{array}$$

In his seminal paper, Zadeh [Zad65] suggested the use of the minimum function for fuzzy conjunction. While all of the other functions have been used at some time, the second most common is the product.

7.3.2 Coming to Consensus: Fuzzy OR

The disjunction of two fuzzy numbers should be high when either one is high, and should be low only when both are low. It should be nonzero when either number is nonzero. A class of functions satisfying these properties are the triangular co-norms [Lee90]. Some triangular co-norms are:

$$\begin{array}{ll}
\textbf{Maximum} & \mathrm{OR}(x, y) = \max(x, y) \\
\textbf{Algebraic Sum} & \mathrm{OR}(x, y) = x + y - xy \\
\textbf{Bounded Sum} & \mathrm{OR}(x, y) = \min(1, x + y) \\
\textbf{Drastic Sum} & \mathrm{OR}(x, y) = \begin{cases} x & y = 0 \\ y & x = 0 \\ 1 & x, y > 0 \end{cases} \\
\textbf{Disjoint Sum} & \mathrm{OR}(x, y) = \max(\min(x, 1 - y), \min(1 - x, y))
\end{array}$$

Zadeh suggested the use of the maximum function for fuzzy conjunction. This is still the most commonly (almost exclusively) used fuzzy disjunction function.

7.3.3 Calculating the Output: Defuzzification

Given a weight w_i for each output fuzzy set, which can be interpreted as a set of fuzzy memberships for the output variable, how should we calculate a crisp numerical output? Again, there are a large number of possibilities.

The most popular techniques require the construction of an output membership function graph, which is then used to calculate the crisp output. Typically, each membership function is cut off (see Figure 7.3) at the height of its weight. From this graph, there are two common ways to calculate the output. The first is the *mean of maximum* method, which takes the average output value of the points assuming the maximum membership. This method tends to result in a jerky mode-based output. A more robust method is the *centroid* method, which calculates the center of mass of the entire membership graph.

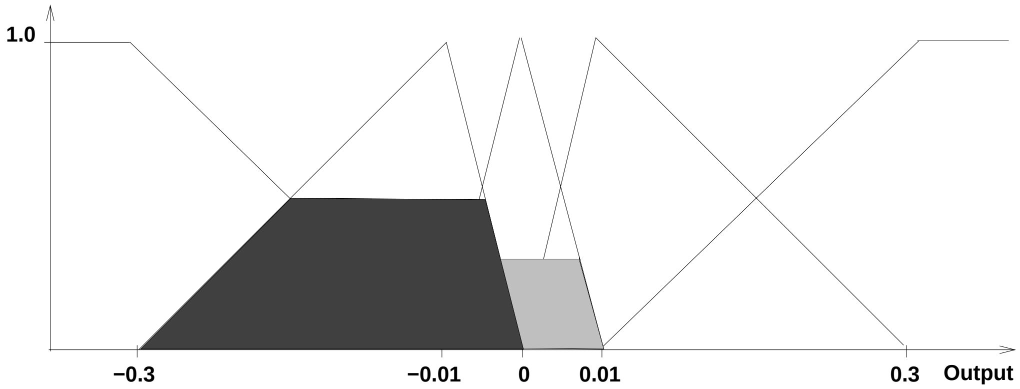


Figure 7.3: Output membership graph construction

The *singleton* defuzzification method, proposed by Sugeno [Sug92], utilizes only the weights w_i for each output fuzzy set and the peaks P_i of each fuzzy set membership function. The output is calculated as

$$O = \frac{\sum w_i P_i}{\sum w_i}.$$

Note that the shape of the output membership functions is not used in output computation – only the peaks; thus the output membership functions can be considered as 'spikes,' or *fuzzy singletons.*

7.4 Learning in Fuzzy Systems

Learning in fuzzy systems is still in its infancy. A number of competing approaches exist, but no well-accepted paradigms have emerged. The approach of Kosko [Kos92] learns only the rules using a clustering technique. This approach requires the membership functions to be set up by hand from expert knowledge. Lin [LL91] and, later, Horikawa [HFU92] and d'Alché-Buc [dBAN92] start with a fixed number of rules and membership functions and perturb them by backpropagation until they fit the data. The required *a priori* choice of the number of rules and membership functions limits the practicality of these approaches. Similar to the approach of the radial basis function network, Wang [WM92] learns to express a function in terms of fuzzy basis functions. The explanation ability of the rule-based system is mostly lost, however.

In the next chapter, we present a method for learning a fuzzy system to approximate example data. The membership functions and a minimal set of rules are constructed automatically from the example data, and the final system is expressed as a computational network for efficient parallel computation of the function value. No *a priori* choice of the number of rules or membership functions is necessary. This method does not require the convergence of an iterative energy search algorithm, as in backpropagation methods, and retains the explanation ability of fuzzy systems by expressing the example data in terms of a single set of membership functions for each variable.

Chapter 8
A Rule-Based Function Approximator

In this chapter, we present an entirely new approach to learning a complete fuzzy system from example data. There are three steps in our method for constructing a fuzzy system: first, learn the membership functions and an initial rule representation; second, simplify (compress) the rules as much as possible using information theory; and finally, construct a computational network with the rules and membership functions to calculate the function value given the independent variables. We begin with definitions and justifications of our fuzzy logic primitives, and then proceed to describe each of the three stages in detail.

8.1 Definitions of Fuzzy Logic Primitives

Membership Functions

As described in Section 7.1, there is considerable leeway in the choice of membership function shape and overlap. No clearly optimal choices exist; however, the following assumptions make the learning process much more well-posed. We will use triangular membership functions rather than Gaussian or other continuous functions; such membership functions are simple to implement and computationally efficient. We will also specify that membership functions are *fully overlapping*; that is, at any given value of the variable the total membership sums to one. (See Figure 8.1 for an example of both properties.) Given these two properties of the membership functions, we need only specify the positions of the peaks of the membership functions to completely describe them. Another benefit of these choices for membership functions is that they allow the interpretation of the system as a simple interpolation between points in the input space. If all rules had a value for every input variable on their condition side, then each rule would specify the value of the output at a single point in the input space, and the system would interpolate smoothly between these points to determine the entire output surface.

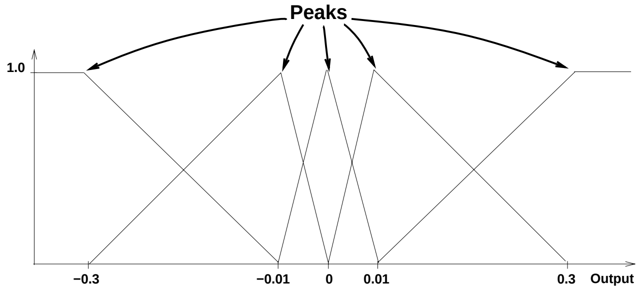
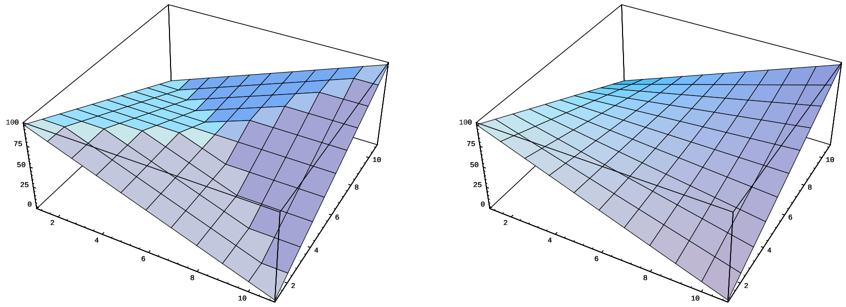


Figure 8.1: Piecewise linear fully overlapping membership functions

Fuzzy AND

We will define fuzzy AND as a product. A product gives a smoother tradeoff between rules than using the minimum, more common in the fuzzy literature. Use of the



Maximum

Summation

Figure 8.2: Output membership layer

minimum results in a sharp corner in the output where the minimum stops following one input and begins to follow the other. The smoother response of the product is better for a simple interpolative function approximation system.

Fuzzy OR

We will define fuzzy OR as a (normalized) sum. A more common approach in the fuzzy literature is to use the maximum rule weight, but consider the following rule set:

```
1. if i1=low  and i2=low  then out=high
2. if i1=low  and i2=high then out=low
3. if i1=high and i2=low  then out=low
4. if i1=high and i2=high then out=high
```

The surface produced by these rules with summation and with maximum are shown in Figure 8.2. Summing the weights rather than taking the maximum results in a smoother output surface. Again, this is better for a function approximation system.

Defuzzification

For defuzzification, we will employ the singleton method, defined in Section 7.3.3. This method is computationally efficient, allows a simple network implementation, and since our output membership functions have a symmetric tradeoff, the most often used centroid method would result in very little difference.

8.1.1 Inference with Dependent Rules

There is a problem with the standard fuzzy inference techniques when used with dependent rules. Consider the example rule set below:

```
1. output = low
2. IF input1 = low                   THEN output = high
3. IF input1 = low AND input2 = low THEN output = med
```

Such a rule set might well be specified by a human expert or come out of a learning system such as the one we will describe. Each of the three rules is valid, but not always applicable. What the ideal inference technique would do is the following: in the corner input1=low and input2=low, we know from rule 3 that the output should be medium. We are not interested in the contribution of the other two rules. Similarly, if we are along the input1=low axis (but not too near input2=low), we wish the output to be high because of rule 2. If we are not too near the input1=low axis, only rule 1 applies and the output should be low. We also expect a smooth tradeoff between these regions, in keeping with the basic principles of fuzzy logic.

If we use the standard fuzzy techniques described in Section 7.3 to compute the output, rule 1 will add its contribution to rules 2 and 3 to drive the output lower than it should be even though we know that along the input1=low axis, the output should be high. Similarly, rule 2 will pull the output higher than it should be at the input1=low and input2=low corner. We know specifically what the value at this corner should be, and the interference of the other rules is unsatisfactory.

What we really want is that a more general rule dependent on a more specific rule should only be allowed to fire *to the degree that the more specific rule is not firing.* Thus the degree of firing of rule 3 should gate the maximum firing allowed for rule 2. Both rules should have a similar effect on rule 1. Let the raw degree of firing of rule i (the fuzzy AND of its conditions) be called f_i, and the adjusted degree of firing be called o_i. Then we can express this relationship as

$$o_1 = f_1(1 - f_3)(1 - f_2)$$

$$o_2 = f_2(1 - f_3)$$

$$o_3 = f_3$$

Thus at the corner specified by rule 3, it alone is allowed to fire, while rules 1 and 2 are completely shut off. In general, the degree of firing of each rule should be gated by the degree of firing of each more specific rule on which it depends.

8.2 The Initial Representation

In this first step, the membership functions and an initial set of cell-based rules are generated automatically from the example data. Before learning, two parameters must be specified. First, the maximum allowable RMS error of the approximation from the example data; second, the maximum number of membership functions for each variable. The system will not exceed this number of membership functions, but may use fewer if the error is reduced sufficiently before the maximum number is reached.

8.2.1 Learning by Successive Approximation to the Target Function

The following steps are performed to construct membership functions and a set of cell-based rules to approximate the given data set. Initially, there are no membership functions.

An example is provided in Figure 8.4 of learning the function in Figure 8.3.

1. **Add input membership functions at input extrema.**

   We add membership functions for each input variable at its maximum and minimum in the data set. Figure 8.4(a) shows the input membership functions representing the input extrema in our example.

2. **Add output membership functions at the corner points.**

   A 'corner' of the input space is a point at which each of the input variables is at its maximum or minimum value in the data set. The closest example point to each corner is found and a membership function for the output is added at its value at the corner point. Figure 8.4(a) shows the three membership functions obtained for the output by looking at the corners of the example function.

3. **Create the initial rule set.**

   The initial cell-based rule set contains a rule for each corner, specifying the closest output membership function to the actual value at that corner. Each rule effectively represents the point that was closest to that corner. Thus we begin with a (hyper)planar model of the data. Note that this is **not** the best planar approximation to the data, but merely the plane correct at the corners. This provides us with a good starting point from which to learn membership functions. Figure 8.4(a) shows the initial planar approximation to the example function.

4. **Add membership functions at the point of maximum error.**

   We compare the current model to the function to be learned and find the example point with the maximum absolute error. We then add a membership function *for each variable* at its value at the point of maximum error. This allows us to completely specify this point, totally eliminating its error. Figures 8.4(b)-(e) show the membership functions added at the point of maximum error for four iterations, gradually improving the approximation to the example function.

5. **Construct a new cell-based rule set; update output membership functions.**

   In this step, we construct a new set of rules to approximate the function. Constructing rules simply means determining the output membership function to associate with each cell. While constructing this rule set, we will also add any output membership functions which are lacking in the data; note that when we add a single new membership function, we add a number of rules to the cell-based set. The correct output value for any point which was not explicitly added may not be among the output membership functions.

   The best rule for a given cell is found by finding the closest example point to the rule (recall each rule specifies a point in the input space). If the output value at this point is 'too far'[1] from the closest output membership function value, this output value is added as a new output membership. After this addition has been made, if necessary, the closest output membership function to the value at the closest point is used as the conclusion of the rule.

6. **If error threshold has been reached or the maximum number of membership functions has been reached in all input variables, exit. Otherwise, go back to step 4.**

[1] Defined as a fixed percentage of the range of the output variable.

> By Figure 8.4(e), the RMS error of the model from the example function is so small that the algorithm can terminate.

In order to make the above algorithm work properly on realistic data, a numerical trick must be employed. To avoid having membership functions for a variable at 10.1, 10.05, 9.995 and 10.0 at the same time, we set a threshold T such that if a candidate membership function is closer to another than T, it will be considered already there and not added. T is typically set at a fixed percentage of the range of the variable.

8.2.2 Control System Considerations

In a general function approximation system, we are concerned with reducing the error in all parts of the input space equally. However, if we are learning a control system we are more concerned with accuracy in the approximation near the 'zero-error' or 'goal' state. It is acceptable if the approximation is less precise far away from the goal state, as long as the control system is able to get the plant near the goal state. Near the goal state, we require more precision in order to have a satisfactory result. This uneven requirement for precision is usually expressed by fuzzy system designers by putting more membership functions near the goal state. To allow for this requirement, in finding the point with the maximum error in the algorithm given above, we multiply the error calculated for each point by a weighting factor which is greater if the point is closer to the goal state and less if not. This will result in more membership functions near the goal state. We typically use a function which decreases exponentially with distance from the goal state; the severity of this decrease is set high if the function to be learned contains much complexity irrelevant to the control problem, and is set low if most of the details of the example function are important. The following function has been used successfully:

$$W(D, D_{max}, M) = e^{D \log(M)/D_{max}}$$

where D is the Euclidean distance from the goal state, D_{max} is the maximum distance from the goal state, and M is the desired weight at the maximum distance.

As an additional measure to assure a precise response at the goal state, we add membership functions before learning at the goal state in each variable. This assures that the system will know *exactly* what to do at the goal state, rather than bouncing back and forth between points on either side.

8.2.3 Theoretical Considerations

Note that given an infinitely dense concentration of data points, the error in approximation to any bounded (not necessarily continuous) function within a closed region of the input space can be driven arbitrarily small by not restricting the number of membership functions. This is equivalent to specifying a larger and larger number of points in the input space to be interpolated between. However, for a finite concentration of data, a limit of membership functions will be reached past which the error will begin to increase. Obviously, if the number of membership functions increases enough, there will be some cells in which no data points lie. The closest point to the rule will be in some other cell, creating an unpredictable conclusion for the rule. This will cause approximation error, whereas without this membership function, the data points on either side might be smoothly interpolated between. Thus the membership functions can be no closer (on average) than

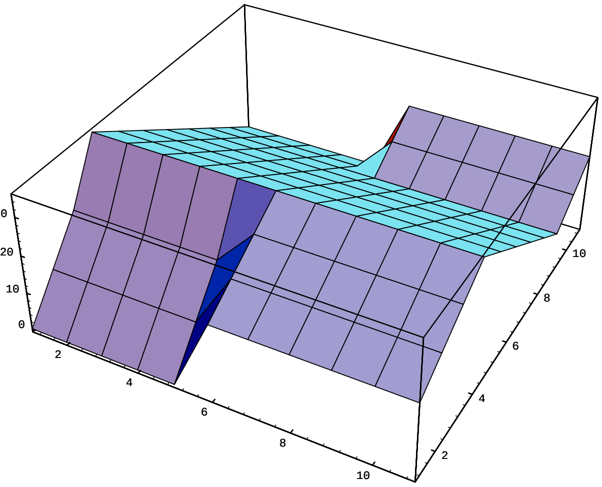


Figure 8.3: Function to be learned

the data points themselves. Ideally, we wish the data to be much more dense in every region than the membership functions need to be in that region.

8.3 Compressing the Rules

In order to have as simple a fuzzy system as possible, we would like to use the minimum possible number of rules. The cell-based rule set resulting from the membership function learning step contains many more rules than are necessary to represent the data. These rules can be compressed into a set of rules which are not cell-based – that is, they may have less conditions than there are input variables.

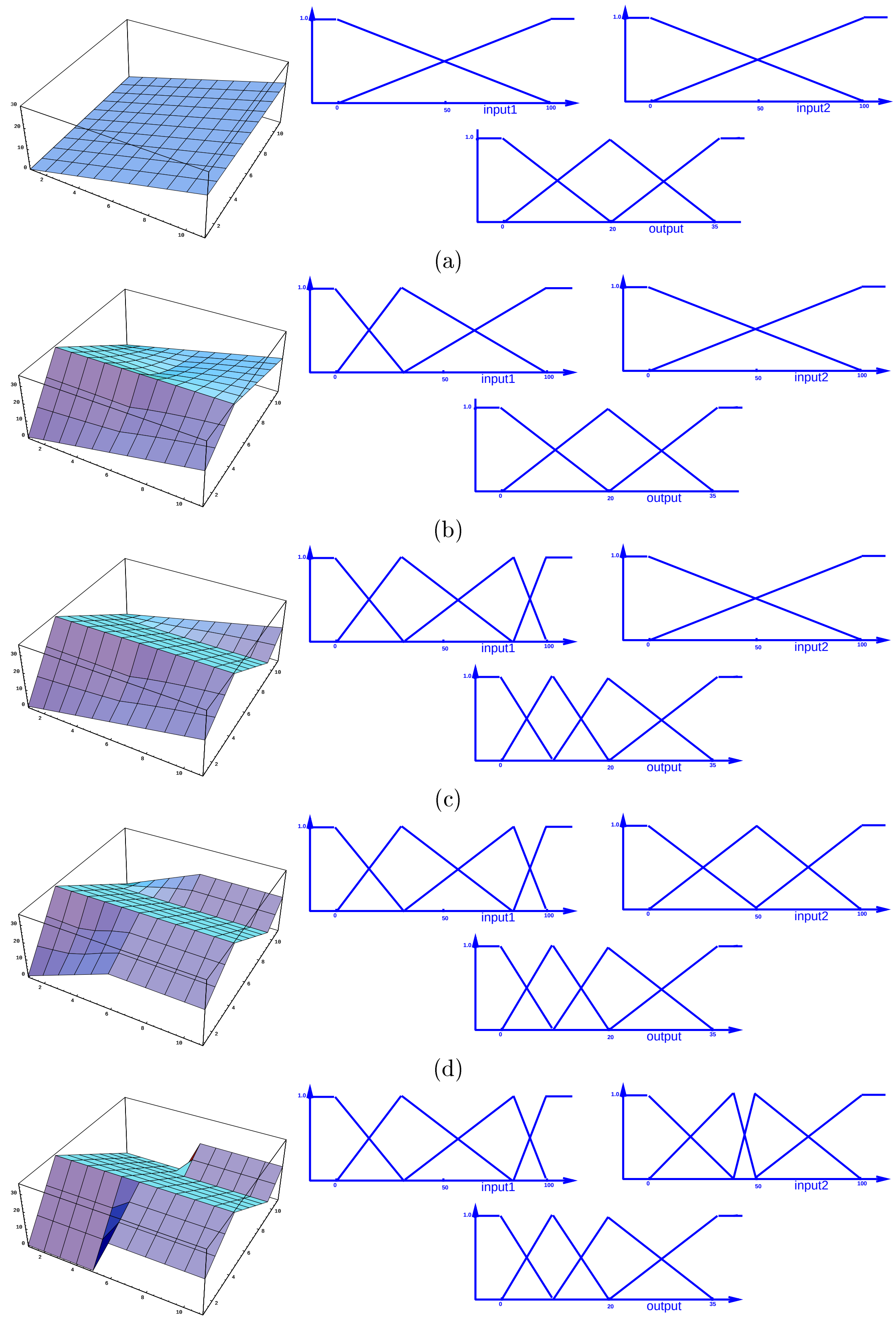
We propose the use of the SQUEEZE algorithm discussed in Part I of this thesis for this purpose. The key to the use of this method is the interpretation of each of the original cell-based rules as a discrete example. The cell-based rule set becomes a discrete data set which is input to the rule-learning algorithm. This algorithm learns the best rules to describe the data set. This new rule set will approximate the same function as the original cell-based rule set with fewer rules.

In order to compress the rules without inducing too much error, we must introduce a modified rule measure.

8.3.1 A Modified Rule Measure

We have previously (Section 3.1.1) discussed information-theoretic measures of rule value with respect to a given discrete data set. The first measure introduced is known as the j-measure. The j-measure is a pure 'goodness' measure, in that it values only the correctness of the rule. The other measure discussed, the J-measure, uses a multiplicative simplicity term to discount rules which are not as useful in the data set in order to remove the effects of 'noise' or randomness.

Using SQUEEZE to learn rules with the j-measure is like a truth-table simplification of the data set – examples will be combined only when no error is caused in the prediction of the data set. This compression is 'lossless' in that it always allows perfect prediction



(a)

(b)

(c)

(d)

(e)

Figure 8.4: Successive approximations to target function

of the data set. Using the J-measure, on the other hand, we ignore noise in the data to a large extent and will combine examples even if some prediction ability of the data is lost. This compression is ‘lossy,’ in that some of the data set may be unpredictable from the resulting rules.

If we simply use the j-measure to compress our original cell-based rule set, we don’t get significant compression. However, for a controls application, we can only tolerate a certain margin of error in prediction of our original cell-based rule set and maintain the same performance. In order to obtain compression, we wish to allow some ‘lossy-ness,’ but not so much as the J-measure will create. The J-measure was designed for a data set in which some underlying pattern existed along with noisy examples (a typical model for a classifier). In this data set, on the other hand, every example is meaningful, and we don’t wish to discount as many uncommon examples as noise. We thus propose the following measure, referred to as the *L-measure*, which allows a gradual variation of the amount of noise tolerance by modifying the rule simplicity term (see Figure 8.5):

$$L(X|y) = f\left(p(y), \alpha\right) j(X|y)$$

where

$$f(x, \alpha) = \frac{1 - e^{-\alpha x}}{1 - e^{-\alpha}}$$

The parameter α may be set at 0^+ to obtain the J-measure since

$$\lim_{\alpha \to 0} f(x, \alpha) = x$$

or at ∞ to obtain the j-measure, since

$$\lim_{\alpha \to \infty} f(x, \alpha) = 1 \qquad (x > 0)$$

Any value of α between 0 and ∞ will result in an amount of compression between that of the J-measure and the j-measure; thus if we are able to tolerate some error in the prediction of the original rule set, we can obtain more compression than the j-measure could give us, but not as much as the J-measure would require. Consider the rule compression example shown in Figure 8.6. This figure shows that as we vary the parameter α in the L-measure from large to small, the error in predicting the original rule set (treated as a discrete data set) holds at near zero for some time before increasing. By the time the error has reached 5%, more than 30% compression of the original rules has been obtained.

8.3.2 The Rule-Compression Algorithm

Clearly we can vary α until we get as much error as we can tolerate, but is there a way to calculate the α which will lead to this error? Since the ‘compression’ that is our criterion is calculated from the number of rules coming out of the SQUEEZE algorithm, there is very little that theory can tell us about it. In addition, predicting the error on a discrete data set beforehand from theory is impractical. However, the second best thing would be to have an algorithm to calculate α for a given desired compression. Figure 8.6 suggests an elegant method.

By use of the incremental SQUEEZE algorithm, it is possible to vary α until the desired compression is reached without rerunning the compression from the beginning. This is accomplished as follows:

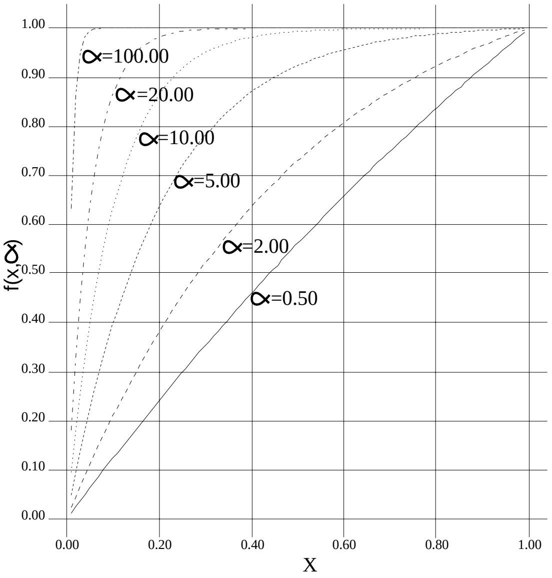


Figure 8.5: Graph of $f(x, \alpha)$ for different α's

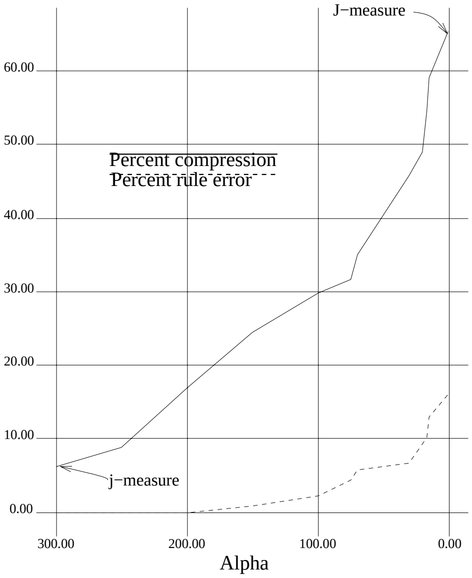
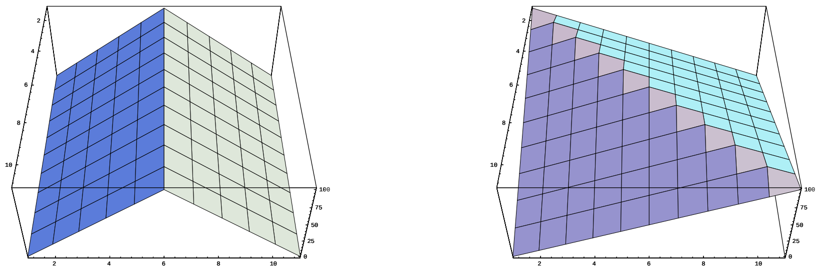


Figure 8.6: Compression vs. error in rule prediction



(a) Compressible

(b) Not compressible

Figure 8.7: Example functions

1. Start α at a very high value (so that the rule measure is the j-measure).

2. Run incremental SQUEEZE to compress the rules as much as possible.

3. Check if the stopping criterion has been reached, either by counting the rules which would come out of the algorithm or checking the error on the training set. If so, exit.

4. Decrease α by a constant factor.

5. Go back to step 2.

At each run of the incremental SQUEEZE algorithm, the rules will generalize as much more as the new rule measure will allow. Since the simplicity criterion is being constantly weakened, the algorithm will proceed *only down* the rule trees as far as it can. When the stopping criterion is reached, the desired α has been found.

8.3.3 Theoretical Considerations

Interpretation of the L-measure

With the introduction of the L-measure, we have a generalized simplicity criterion in our rule measure. What does this mean in terms of the J-measure simplicity criterion?

Using $p(y)$ as a simplicity criterion, as in the J-measure, has a pleasing theoretical interpretation: when the J-measure is summed over all possible y, the mutual information between the output X and the input Y is obtained. It is also the easiest probabilistic simplicity criterion to calculate. However, is it the 'right' simplicity criterion? Will it always yield the appropriate tradeoff between simplicity and j-measure? The answer is, of course, **no**. Like everything else in the data analysis realm, the importance of simplicity is dependent upon the data. The J-measure criterion seems to be a good choice for a large number of data sets, but cannot be theoretically shown to be the best.

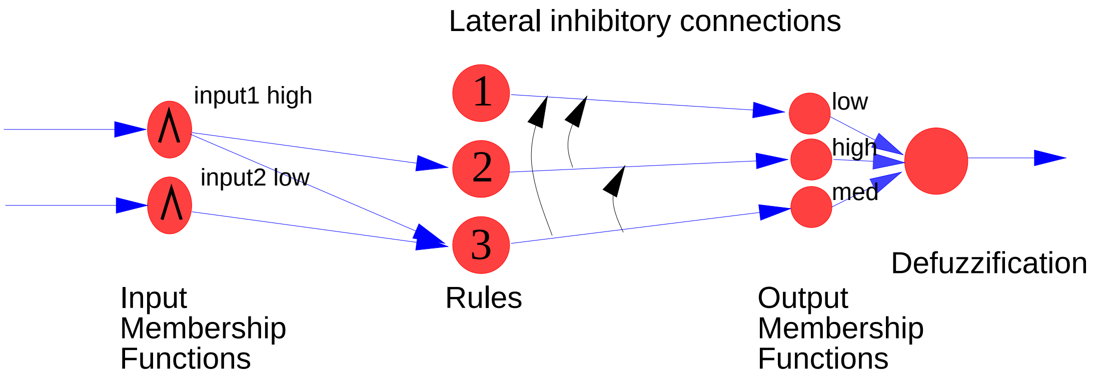

Lateral inhibitory connections
input1 high
input2 low
∧
∧
1
2
3
low
high
med
Input Membership Functions
Rules
Output Membership Functions
Defuzzification

Figure 8.8: Computational network constructed from fuzzy system

How much compression can be obtained?

The axis-parallel problem (Section 2.1) limits the amount of rule compression we can obtain. The amount that it is possible to compress the rules without error is directly proportional to how much the 'features' of the function to be approximated are parallel to the input axes. Rule-based approximation systems break a function down into rectangular hyperboxes in the input variable space. A cell-based rule set specifies a value in each box where data exists. The rule compression system works by combining similar boxes, but can only combine boxes if there is a similar pattern along an input variable axis. For example, consider Figure 8.7. The function in (a) has a ridge parallel to the input1 axis. This function can be simply expressed in terms of fuzzy rules. However, consider the function in (b). This is the same function rotated 45°. This function has a rather complex description in terms of fuzzy rules, and little compression can be obtained.

8.4 Expressing the Rules in Network Form

Constructing a computational network to represent a given fuzzy system can be accomplished as shown in Figure 8.8.[2] From input to output, layers represent input membership functions, rules, output membership functions, and finally defuzzification. A novel feature of our network is the lateral links shown in Figure 8.8 between the outputs of various rules; these links allow inference with dependent rules. Each layer is described in detail below.

The Input Membership Layer

> This layer merely implements the input membership functions by generating a value between zero and one given a numerical input. A connection is made into each node in this layer from the input variable for which it is a membership; a connection is made out of each node in this layer to each rule which has this input membership as a condition.

The Rule Layer

[2] The rules represented here are the dependent example rules from Section 8.1.1.

This layer contains a node for each rule, receiving inputs from the appropriate input layer membership functions, and connecting to exactly one output membership function node. Each node performs a product (fuzzy AND) of its inputs.

The links between the rule layer and the layers before and after it have unit weight. (See Section 10.1.2).

The Output Membership Layer

Each node in this layer takes inputs from all rules that conclude this output membership function and outputs the sum (fuzzy OR) of the weights for that output fuzzy set.

The Defuzzification Layer

This layer performs a defuzzification by normalizing the weights from each output membership function and performing a convex combination with the peaks of the output membership functions. This implements the singleton method.

Lateral Inhibitory Links

The lateral arrows in Figure 8.8 are inhibitory connections which take the value at their input, invert it (i.e., subtract it from one), and multiply it by the value at their output. More generally, each rule has a lateral inhibitory link coming to it from every higher-order rule which contains all of its conditions. This allows inference with dependent rules, as discussed in Section 8.1.1.

8.5 Summary

In this chapter, we have presented a method which, given examples of a function and its independent variables, can construct a computational network based on fuzzy logic to predict the function given the independent variables. The user must only specify the maximum number of membership functions for each variable and the maximum RMS error from the example data.

There are three innovative aspects of this system, each of which is valuable independently:

- Membership functions are learned by successive approximation.

  Membership functions are often generated by hand. This scheme allows the membership functions to be chosen based only upon an error criterion by an algorithm which must terminate in a small number of steps.

- Cell-based rules are compressed into a minimal rule set.

  Many systems exist using cell-based rule sets. The ability to compress such rule sets and retain the same performance will lead to more manageable, understandable rule sets.

- The problem of inference with dependent rules is solved.

  When a system designer sets up a fuzzy system, he may well want to use dependent rules. This inference scheme allows the rule system to perform as he would expect it to.

The final fuzzy system's actions can be explicitly explained in terms of rule firings. If a system designer does not like some aspect of the learned system's performance, he can simply change the rule set and the membership functions to his liking. This is in direct contrast to a neural network system, in which he would have no recourse but another round of training.

Chapter 9
Application to Control Systems

In this chapter, we describe the application of our proposed function approximator to multiple-input, single-output, discrete-time, closed-loop automatic control systems. We begin by demonstrating the conversion of a neural controller to a fuzzy controller on a simulated problem, and then discuss the learning of a controller for a completely unknown plant.

9.1 Converting Any Controller into a Fuzzy Controller

Given examples of the output of a working controller sufficient to cover the input space, we can learn a fuzzy approximation to the control function which will also function as a controller. We assume that the prototype controller simply implements a nonlinear function of the input variables; any controller internal state variables will render this approach ineffective. We will demonstrate this application of our proposed function approximator with the problem of truck-and-trailer backing.

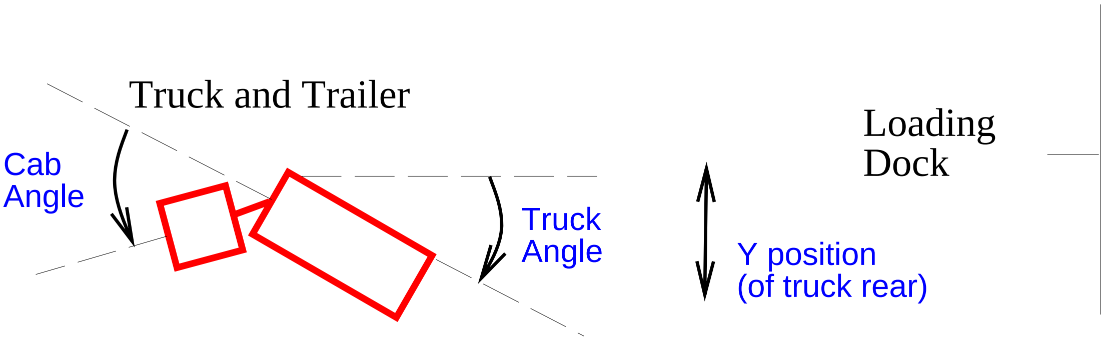
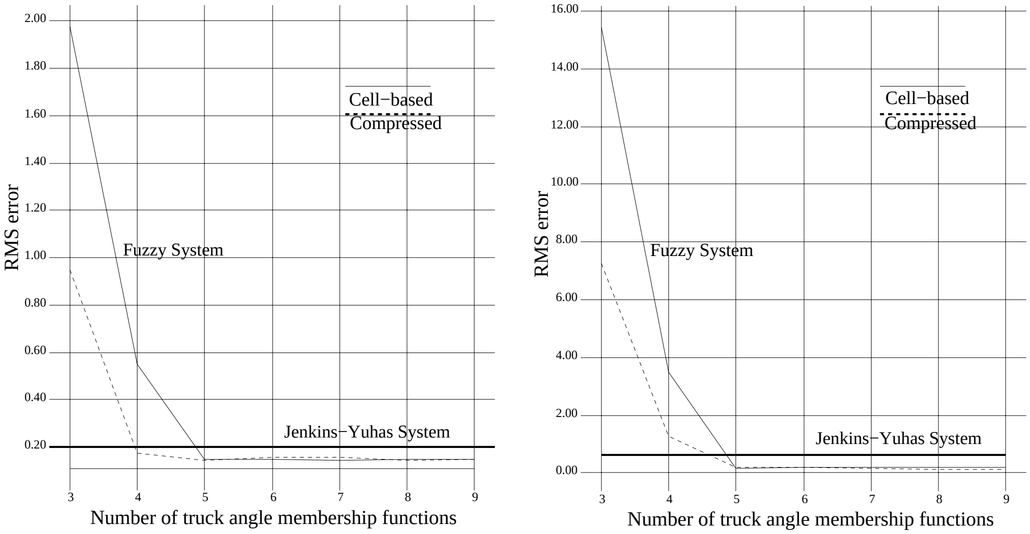


Figure 9.1: The truck and trailer backer-upper problem

The problem of backing up a truck and trailer to a loading dock, as posed by Nguyen and Widrow [NW89], is quite difficult, sometimes requiring that the truck actually drive *away* from the loading dock in order to turn around. We will specify that the truck may only drive in a reverse direction. Jenkins and Yuhas [JY92] have hand-crafted a very efficient neural network solution to this problem with only two hidden units. Its trajectory is quite efficient, especially considering the simplicity of its implementation. We have chosen to approximate this system because it is highly nonlinear[1] and its features are non-parallel to the axes; this makes a more challenging approximation problem for our system. The truck and trailer backer-upper problem is parameterized in Figure 9.1.

The function approximator system was trained on 225 example runs of the Jenkins-Yuhas controller, with initial positions distributed symmetrically about the field in which the truck operates. In order to show the effect of varying the number of membership

[1] This problem is only nonlinear if posed as in Figure 9.1. Geva et al. [GSW92] have shown that if polar coordinates are used, a linear controller can solve the problem.



a) Control error: final y position

b) Control error: final truck angle

Figure 9.2: Results of experiments with the truck backer-upper

		Number of truck angle membership functions						
		3	4	5	6	7	8	9
Number of Rules	Cell-Based	75	100	125	150	175	200	225
	Compressed	48	67	86	100	114	138	154
Compression		36%	33%	31%	33%	35%	31%	32%

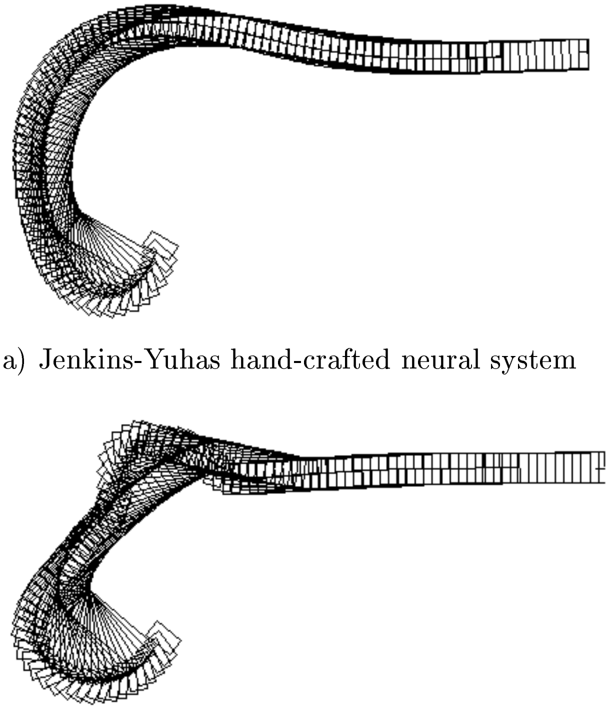
Figure 9.3: Number of rules and compression figures for learned TBU systems

functions, we have fixed the maximum number of membership functions for the y position and cab angle at 5 and set the maximum allowable error to zero, thus guaranteeing that the system will fill out all of the allowed membership functions. We varied the maximum number of truck angle membership functions from 3 to 9. The effects of this are shown in Figure 9.2. Note that the error decreases sharply and then holds constant, reaching its minimum at 5 membership functions. The Jenkins-Yuhas network performance is shown as a horizontal line. At its best, the fuzzy system performs slightly better than the system it is approximating.[2]

For this experiment, we set a goal of 33% rule compression. We varied the parameter α in the L-measure for each rule set to get the desired compression. (In each case, the prediction error of the original rule set was 5% or less.) Note in Figure 9.2 the performance of the system with compressed rules. The performance is in every case almost identical to that of the original cell-based rule sets. This validates the effectiveness of our rule compression and dependent rule inference schemes. The number of rules and the amount of rule compression obtained can be seen in Figure 9.3.

One thing we have not quantified in this example is the *smoothness* of the truck trajec-

[2] This is due to the fact that the Jenkins-Yuhas system approaches zero truck angle at the loading dock *asymptotically*, whereas the fuzzy system turns sharply to zero truck angle and backs up straight to the dock.



(a) Jenkins-Yuhas hand-crafted neural system

(b) Learned fuzzy system

Figure 9.4: Demonstration of mode-based behavior of fuzzy system

tory (see Figure 9.4). Although the learned fuzzy system with 5 truck angle membership functions actually performs better in RMS docking error than the original Jenkins-Yuhas network, its path is sometimes not as smooth. The fuzzy truck backer-upper has 'modes' of operation: the truck will first turn around, then back up in a straight line at a diagonal angle, then change direction sharply and back towards the loading dock. This is directly related to the piecewise approximation to the original function.

9.2 Learning a Controller for an Unknown Plant

In the last section, we showed the application of our proposed fuzzy function approximator to the learning of a control function for a simulated system given samples of the actions of a good controller. While this is certainly useful in converting an existing controller to a fuzzy rule-based one, it begs the question, "What will we do when faced with a *new* system, for which there exists no good controller?"

In this section, we will suggest a technique for learning a complete control system for an unknown plant. In addition, we will show the results of experiments with a ball-and-beam system which bear out the usefulness of this method.

9.2.1 The Loop-Back Approach

Learning a new controller for a physical system presents a dilemma. To learn given no initial knowledge of the plant, a learning controller must experiment with the system and make mistakes until it reaches its goals. To quote renowned learning researcher Tom Mitchell at the recent NIPS '92 robot learning workshop, "*Tabula rasa* learning will never scale up." In other words, we expect that as the difficulty of the control problem increases, this experimentation will grow more and more complex until it becomes intractable. In many cases, it may simply be impractical to experiment while learning a new controller; how many helicopter crashes per day can you stand?

Luckily, it is not necessary to learn from nothing. In any system, there are basic rules that we humans know intuitively about the controller. These rules may not form a complete controller, but at least give some idea of the control of the system in certain cases; for example, "If you want it to nose up, pull the stick back." What we don't know is *exactly how much* you must pull the stick if you want to level the vehicle. The ideal learning system would be able to make use of this common-sense knowledge in boot-strapping itself to a good controller. By employing fuzzy logic, we can do just that.

Because our system is based on fuzzy logic, we can embody prior knowledge of the controller into a 'boot-strap' control system in terms of membership functions and rules. We can then *adapt* this controller to the real system. We accomplish this as follows (see Figure 9.5 for a block diagram):

1. Create a fuzzy controller from whatever common-sense rules may be available. Membership functions should be chosen via the best knowledge that the designer has about the system parameters.

2. Convert this fuzzy controller into a table-based controller by calculating its output on a grid of points.

3. Adapt the table-based controller to fit the physical system until the performance is satisfactory (by some pre-specified criterion).

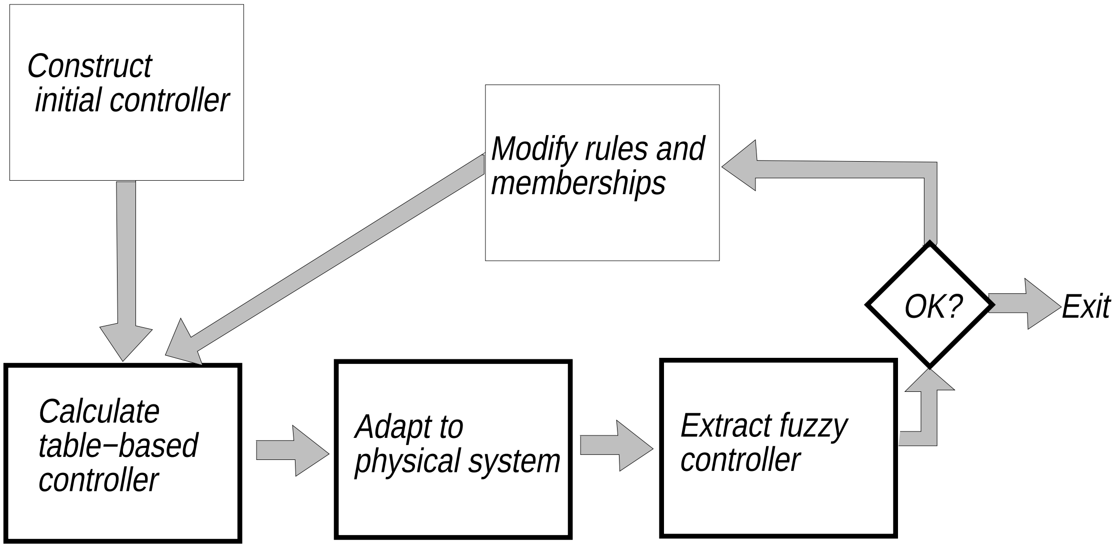

Construct initial controller
Modify rules and memberships
OK?
Exit
Calculate table−based controller
Adapt to physical system
Extract fuzzy controller

Figure 9.5: The Loop-Back Approach

4. Give the adapted table-based controller as input to the function approximator presented in the last chapter, and a fuzzy controller which approximates its performance will result.

5. If performance is found at a later time to be unsatisfactory in some way due to a change in the plant, the final fuzzy system can be quickly adapted to the new plant by using it as an initial controller in step 2.

Step two, creating a table-based controller from a fuzzy controller, is no trick — this is just calculating the output at a grid of points in input space. However, this is likely to drastically increase the amount of memory required to specify the controller. Step four, learning a fuzzy controller from the adapted table-based controller, we have already discussed in Chapter 8. The learned fuzzy system will represent the table-based controller using far less memory; in addition, we regain the inherent explanation ability of a rule-based system. Step three, adapting the initial table to a physical system, is the backbone of this approach, and requires some explanation.

9.2.2 Adaptive Table-Based Controllers

There are a large number of algorithms for table-based adaptive control.[3] Perhaps the earliest such attempts were made by Waltz and Fu [WF65] who partitioned the system input space and tried to find the optimal control for each sub-region by a reinforcement scheme. Michie and Chambers [MC68] chose a similar representation scheme, but use an algorithm for determining appropriate control actions when 'payoff' is delayed. There are several neural network approaches which could be interpreted as table-based control: Albus' CMAC [Alb75] generates control actions using several overlapping input space regions; Rumelhart et al. [RM86] again encode an input in terms of its membership in several overlapping regions and feed this encoding to a neural network; Rosen et al.

[3] This genealogy of table-based adaptive control follows that of Gonzalez [Gon93].

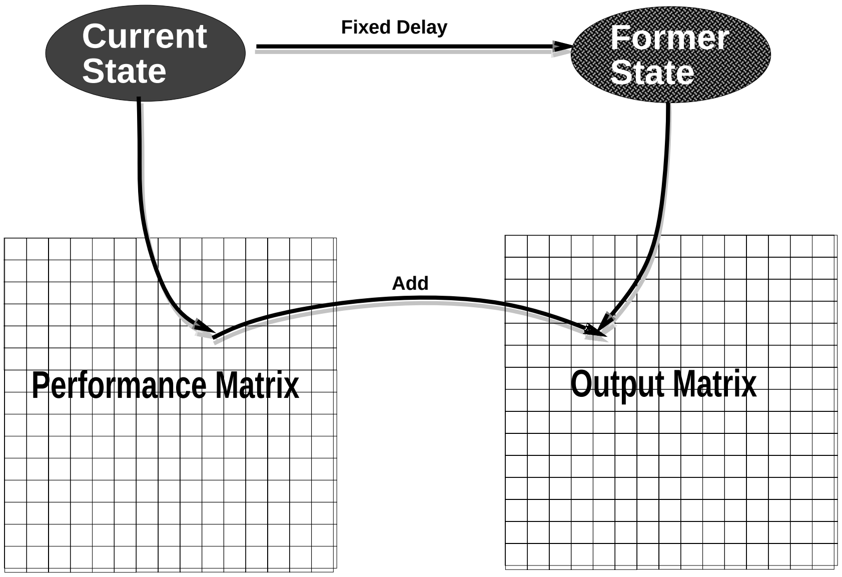


Figure 9.6: The Extended Self-Organizing Controller

[RGV92] propose the use of non-overlapping regions which shift according to the system dynamics. Gonzalez [Gon93] refines an initial coarse partition of the input space at need, learning the best control action for each region. From a fuzzy control perspective, Scharf and Mandic [SM85] proposed an algorithm which partitions the input space and modifies the control actions incrementally by a input space ‘goodness’ criterion.

The time complexity of such algorithms can be very high, so due to the lack of computational power available and our need for a real-time implementation of the adaptive controller, we have chosen the extremely simple algorithm of Scharf and Mandic, described below.

The Extended Self-Organizing Controller

Scharf and Mandic of Queen Mary College proposed an adaptive ‘fuzzy’ controller in 1985 and applied it to the learning of a controller for a robotic arm. This controller was called the SOC (Self-Organizing Controller). We will describe an extension to their controller, which used discrete outputs and updates.

The extended controller (ESOC, see Figure 9.6) operates by partitioning the input space into a rectangular grid of possible states, referred to as the output matrix. The controller learns an appropriate control response at each grid point, and smoothly varies between these control values between grid points. The learning is accomplished by the use of a performance matrix, which is the same dimension as the output matrix. The performance matrix tells the numerical desirability of each state in the input space, and gives an incremental direction to change the control output if this state has been arrived at. The performance matrix is fixed before learning; the output matrix is the only data modified during learning. Learning takes place while the controller dynamically controls the plant. Learning proceeds on each iteration of the controller as follows (let there be d controller inputs):

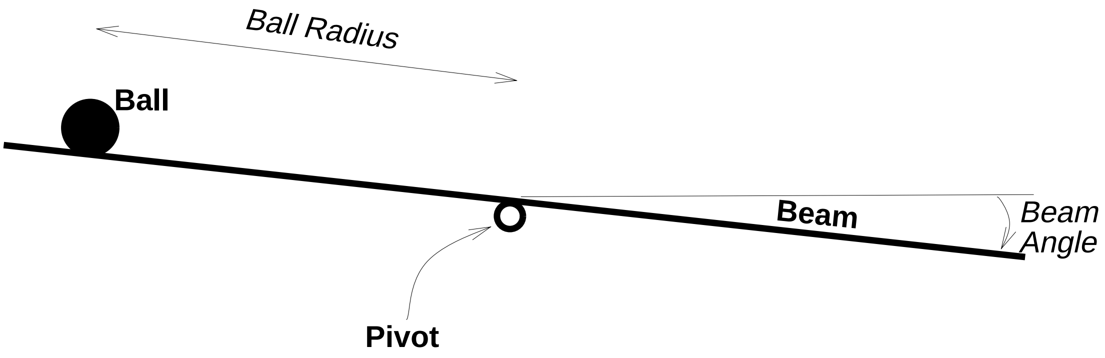


Figure 9.7: The ball-and-beam problem

1. The performance increment is calculated by smoothly varying between the 2^d performance matrix values nearest the current state.

2. The controller looks into the past a fixed delay and finds out the system's former state.

3. The 2^d points in the output matrix nearest the former state are updated by adding the performance increment. Each grid point gets a portion of the increment proportional to its relative distance from the former state.

We assume implicitly in this adaptation scheme that the current state depends only upon the former state and the output at that time (Markov property). This way, we can attribute any 'badness' at the current state to the output at the former state. As the learning proceeds, control outputs which lead to undesired states are changed, and those which lead to desired states are unchanged. Eventually, the learning algorithm should converge to the controller with the desired performance criterion. However, the success of this method depends on the choice of the performance matrix; the performance matrix is set up by trial and error, which limits this approach.

9.2.3 Experimental Results with the Ball-and-Beam Problem

In order to validate our suggested loop-back approach, we have performed experiments with the ball-and-beam problem. This control problem, as diagrammed in Figure 9.7, is to bring a ball to a desired location on a horizontal beam by controlling the angle of the beam. This problem is quite nonlinear for a large error in the ball position. We will begin by demonstrating the effect of our learning system on a simulated ball-and-beam system, and conclude with the same experiments on a real ball-and-beam system.

Simulated Ball-and-Beam

If we choose the radius of the ball from the center of the beam r and the angle of the beam θ as our state variables, the dynamics of the system can be expressed parametrically as shown below [Cas90]:

$$\begin{aligned}
\dot{x_1} &= x_2 \\
\dot{x_2} &= x_1 x_4^2 - G\sin(x_3) \\
\dot{x_3} &= x_4 \\
\dot{x_4} &= u \\
y &= h(x) = x_1
\end{aligned}$$

where $(x_1, x_2, x_3, x_4) = \left(r, \dot{r}, \theta, \dot{\theta}\right)$ and G represents the acceleration of gravity: $G = 9.8m/s^2$. In this canonical statement of the problem, we desire to bring the ball radius $y = r$ to some set point s by controlling the torque on the beam $u = \ddot{\theta}$. The set point s may be constant or a function of time. For our simulations, we will assume that the motor driving the beam operates in such a fashion that it can be modeled with a maximum angular velocity $\dot{\theta}_{max}$; that is, given a maximum input, the beam motor will provide only enough torque to move the beam at a constant angular velocity. This assumption is consistent with the design of a servo-motor, such as the one used in our actual hardware implementation. Thus our real command to the simulation will be $u' = x_4 = \dot{\theta}$.

In order to simulate the system, we must specify some parameters. Let us specify that the beam is one meter long; if the ball reaches the end of the beam it will be stopped. We will also specify that the maximum angular velocity of the beam is $\dot{\theta}_{max} = 0.9m/s^2$. Almost solely, this parameter determines the difficulty of the problem in simulation. This parameter has been set just higher than the minimum value that will allow a solution of the 3 set point problem discussed below. We will simulate with a time step $T_{step} = 1$ms, and our controller will give a new output every $T_{loop} = 20$ms. The maximum angle that the beam will be able to take on will be specified as $\theta_{max} = \pi/4$, or 45 degrees.

Our controller will take as input the error from the desired set point

$$E(t) = r(t) - s$$

and the change in the error (referred to as *D-error*)

$$\dot{E}(t) = \frac{E(t) - E(t-1)}{T_{loop}}$$

The controller will output a desired beam angle θ_{des}.[4] We will calculate the desired beam angular velocity as

$$\dot{\theta}_{des} = \frac{\theta_{des} - \theta(t)}{T_{step}}$$

and the commanded beam angular velocity as

$$u' = \dot{\theta}(t) = \min(\dot{\theta}_{des}, \dot{\theta}_{max})$$

The performance matrix used in ESOC was designed as specified in Figure 9.8. The performance matrix describes the input space path that we wish the controller to follow in getting the ball to the set point. This path requires a constant speed when the absolute error is above some threshold E_{far}, a gradual slowing down towards zero with slope S when the error is between E_{far} and E_{near}, and zero speed when error is less than E_{near}.

[4] The beam angle is output because it is much easier to design a reinforcement controller for beam angle than for beam angular velocity.

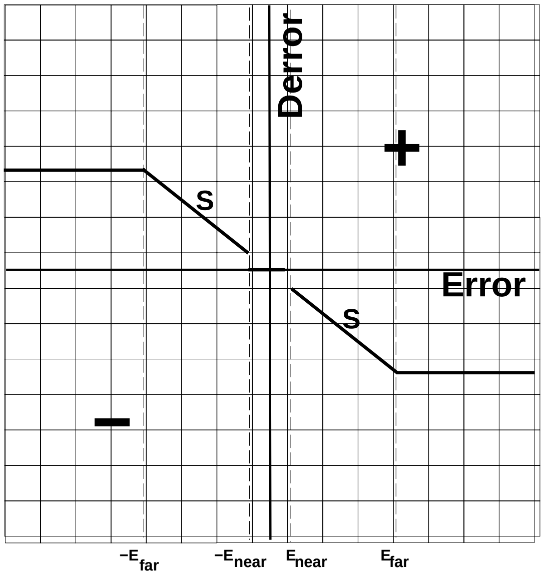


Figure 9.8: Performance matrix design

The constant speed requirement at large error assures that the ball will not go faster than the system can control; the gradual slowdown assures that the ball will reach the set point at zero speed; and the zero speed requirement near zero error is used to damp oscillation near the set point. The performance matrix values are zero on the ideal path, positive above it, and negative below it. The values get larger the farther the distance from the ideal path. A gain parameter on the performance matrix allows a variation of the learning rate. With the proper choice of the parameters E_{far}, E_{near}, and S, this path will get the ball to within E_{near} of the set point with no overshoot and maintain it there with a minimum of oscillation.

The performance matrix is implemented as a nonlinear grid of points in the input space. To avoid cluttering the discussion of experiments, all of the parameters necessary to specify the performance matrix for each experiment to be shown are listed in Section A.2.

Simulation A: Three Set Points

For our first simulated experiment, we will designate three set points s on the beam: at $s_1 = -0.375$m, $s_2 = 0.0$m, and $s_3 = 0.375$m. We begin learning with the output matrix set to all zero (such that the default output is to balance the beam). We train the system by changing the set point between s_1, s_2, and s_3 and waiting for the ball to stop. In the beginning, the ball follows an input space path as shown in Figure 9.9(a). The ball has significant overshoot and oscillates before reaching the set point. As we train the controller, its performance improves until (a simulated time of 50 seconds later) the input space paths look as shown in Figure 9.9(b). This controller has no overshoot and will not oscillate at the set point. It follows a path close to the one designated by the performance matrix. Ordinarily, we would stop training here. What will happen if we continue training the system? The results of overtraining on this system are shown in Figure 9.9(c), after an additional simulated minute of training. The system now tries to *too* closely follow the desired path; this results in overshoot and oscillation, mostly as the system attempts to reach the desired path as quickly as possible and is unable to compensate for its speed.

The best controller (Figure 9.9(b)) is now expressed as a table of numbers representing the control output at a grid of points in the input space. The nonlinear surface learned is shown in Figure 9.10.[5] We can see near small error a gentle movement of the ball towards the goal of zero error and zero speed, and at larger error a wave-like surface which speeds the ball up and then slows it down as it nears the set point. Note also that there are some areas of the input space around the larger values that have never been visited.

This problem is learned too quickly in simulation to benefit from an initial fuzzy controller as specified in the loop-back approach, but let us construct a fuzzy controller from this table-based one using the methods from Chapter 8 and compare its performance with the table-based controller. If we specify beforehand that there will be a maximum of nine membership functions for each of the inputs (see Section A.1 for a complete list of function approximator parameters), we can use the fuzzy function approximator to obtain the fuzzy controller shown in Figure 9.11.[6] The performance of the original table-based controller is shown in Figure 9.12(a), the fuzzy controller with uncompressed rules in Figure 9.12(b), and finally the fuzzy controller with compressed rules[7] in Figure 9.12(c). The original table-based controller performs ideally, as we trained it to do. The fuzzy controller is an approximation to the table-based controller, and as such, does not perform quite as well. This controller exhibits some overshoot and damped oscillation. The fuzzy controller with compressed rules is an approximation to the *fuzzy* controller, and again we lose some performance. In general, we should expect our performance to decline with each successive level of approximation to the original controller.

Simulation B: Sinusoidal Set Point

Due to its non-linearity, the ball-and-beam problem has been addressed quite often in the control systems literature. The most common demonstration is that of tracking a sinusoidally moving set point. These results for our system are included for comparison with the published results of [Cas90, Tee91, HSK92, HR92] who use similar ball-and-beam dynamics. The sinusoidal set point which we will employ is

$$s(t) = 0.375 \sin(2\pi \frac{t}{T})$$

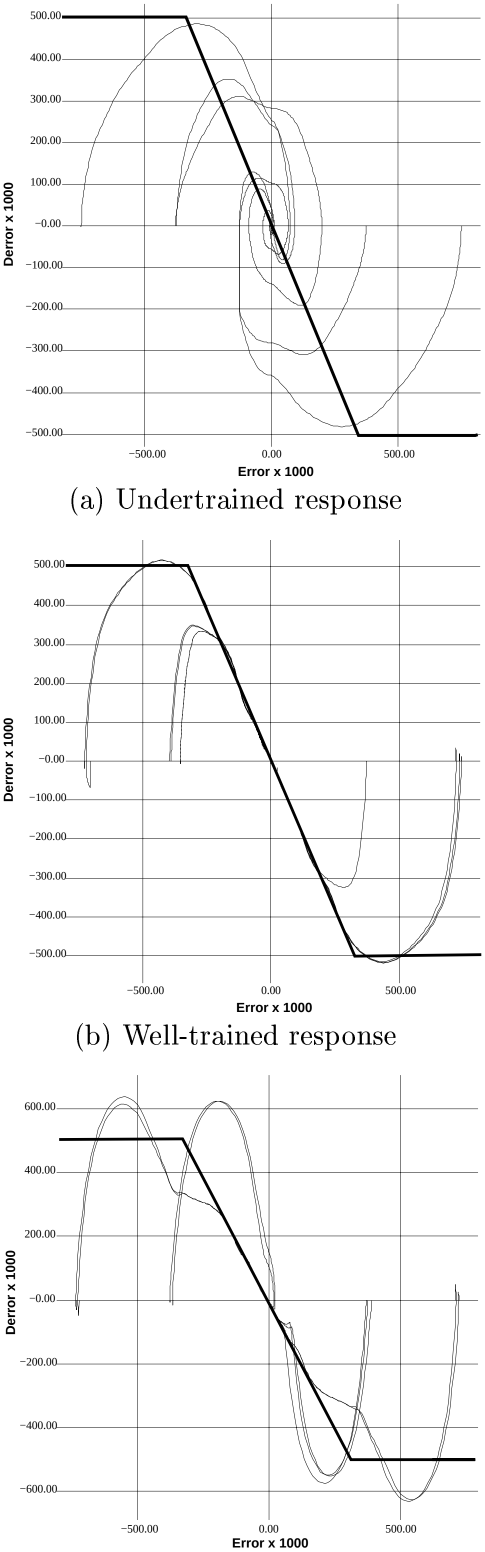
where T is the period of oscillation. Thus this sinusoid moves between s_1 and s_3 of the previous experiment.

Again, we start the output matrix at all zeros (balanced default) and train the system with a sinusoid of period $T = 5$ seconds. The ball cannot track sinusoids much faster than this due to our maximum assumptions on the beam angular velocity; this makes the problem about as difficult to learn as it can be. The performance of the system as it learns

[5] Even though the grid is nonlinearly spaced, the grid points have been plotted at regular intervals to facilitate observation of the detail near zero.

[6] The membership functions are numbered from most negative to most positive, starting with zero. The number of membership functions for each variable is given in parentheses. The rules are shown in a matrix indexed by the membership function number. A number in the matrix indicates a second-order rule; a vertical or horizontal box with a large number in it indicates a first-order rule.

[7] Given the dependent rule inference scheme, the best way to compress these rules would be to specify that the output is `memb11` unless otherwise specified. However, that rule is only good *in the context of the other rules*; alone, it is quite bad and so is not found by our rule compression algorithm.



(a) Undertrained response

(b) Well-trained response

(c) Overtrained response

Figure 9.9: Input space paths of simulated system while learning

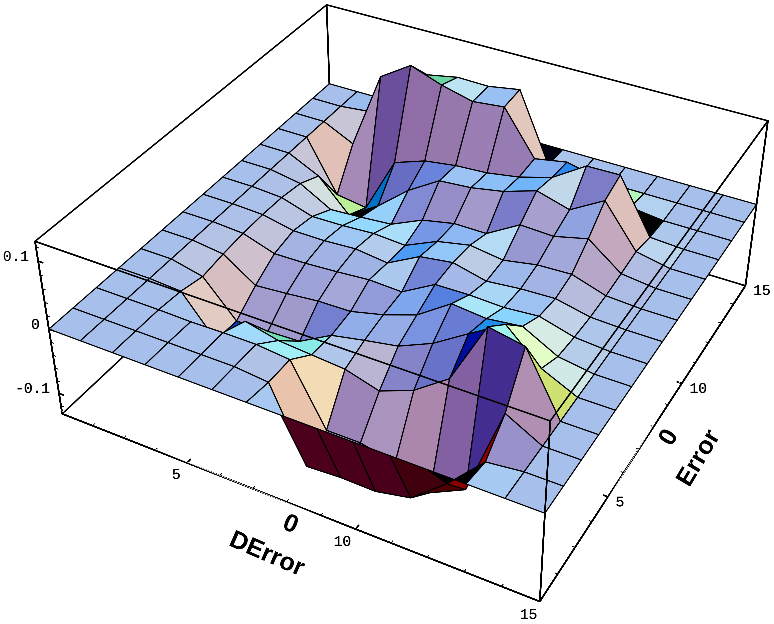


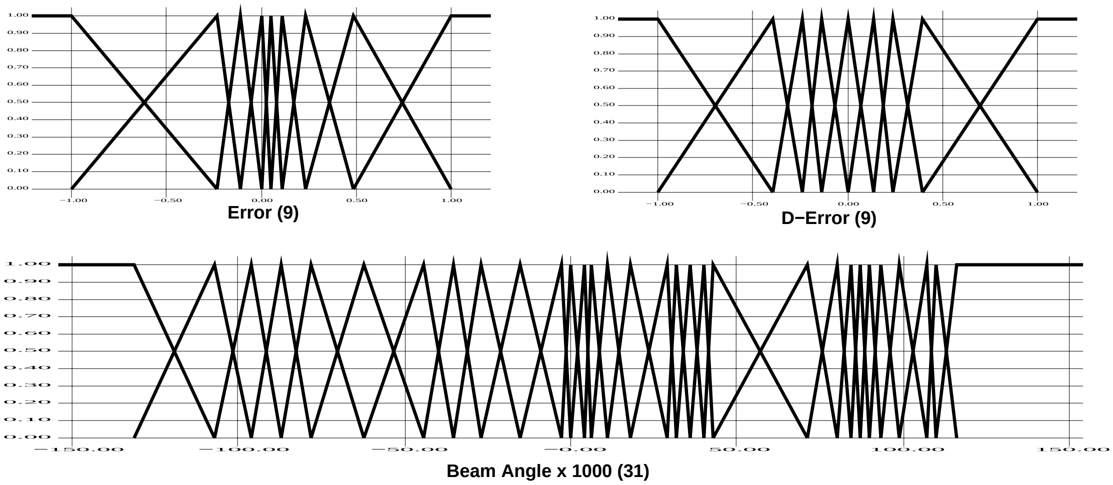
Figure 9.10: Control surface of learned controller

from nothing is shown in Figure 9.13.[8] By the end of this training, it is tracking the ball with very a small error. A plot of the performance of this controller as it begins tracking a sinusoid of period $T = 5, 10, 20$, and 30 seconds is shown in Figure 9.14(a). The ball is started at $s_1 = -0.375$ to cause a large initial error. The ball reaches the desired path within 3.5 seconds and tracks the sinusoid with an oscillating error that diminishes as the period T increases. There is no overshoot or oscillation in reaching the set point. For a detail of the steady-state error, see Figure 9.14(b). The error diminishes so far by $T = 20$s that the roundoff error from a four-digit floating point representation can be observed. The error is virtually constant at $T = 30$s; this small constant error is probably due to the fact that the system was trained only with a negative initial error (the only asymmetry in its training). This error is smallest when the sinusoid reaches its minimum peak, and largest when the sinusoid reaches its maximum peak.

Now let us compare the performance of a fuzzy controller on sinusoid-tracking with that of the table-based one. The fuzzy controller constructed from the learned sinusoid-tracking table-based controller is shown in Figure 9.15, its transient error (with compressed rules) in Figure 9.16(a), and its steady-state error (with compressed rules) in Figure 9.16(b). There is significant oscillation as the simplified fuzzy system tries to begin tracking the sinusoid, and the steady state error is an order of magnitude larger than before the fuzzy approximation. However, considering the wide range of speeds and the precision of output

[8] In this figure, between 15 and 25 seconds, the ball hits either end of the beam while the system is learning. Because our simulation stops the ball if it reaches the end of the beam, we can learn this problem faster. If we assume an infinite beam, the solution can still be learned, but the performance matrix gain must be reduced.

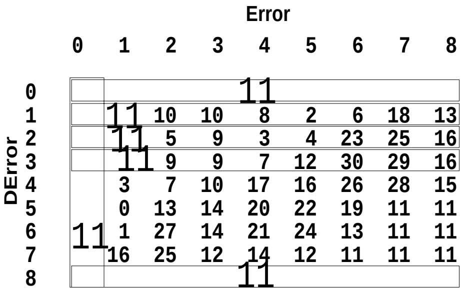
Membership Functions



Rules

DError \ Error	0	1	2	3	4	5	6	7	8
0	11	11	11	11	11	11	11	11	11
1	11	11	10	10	8	2	6	18	13
2	11	11	5	9	3	4	23	25	16
3	11	11	9	9	7	12	30	29	16
4	11	3	7	10	17	16	26	28	15
5	11	0	13	14	20	22	19	11	11
6	11	1	27	14	21	24	13	11	11
7	11	16	25	12	14	12	11	11	11
8	11	11	11	11	11	11	11	11	11

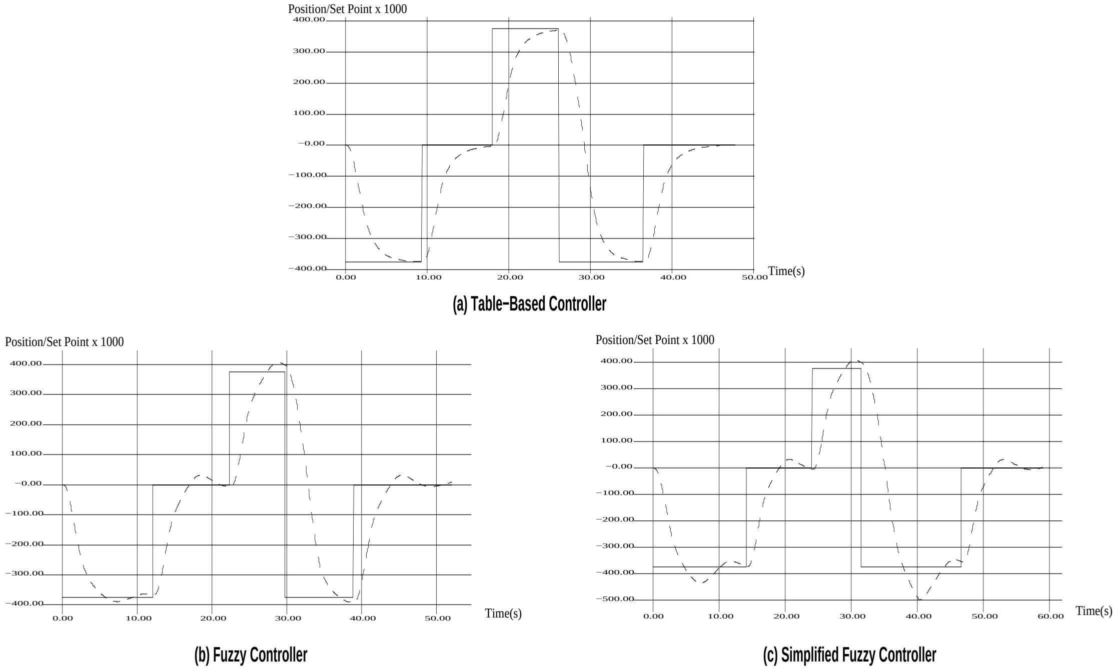
The 81 uncompressed rules



DError \ Error	0	1	2	3	4	5	6	7	8
0					11				
1		11	10	10	8	2	6	18	13
2		11	5	9	3	4	23	25	16
3		11	9	9	7	12	30	29	16
4	11	3	7	10	17	16	26	28	15
5		0	13	14	20	22	19	11	11
6		1	27	14	21	24	13	11	11
7		16	25	12	14	12	11	11	11
8					11				

The 59 compressed rules

Figure 9.11: Fuzzy controller for simulated three set point problem



(a) Table−Based Controller

(b) Fuzzy Controller

(c) Simplified Fuzzy Controller

Figure 9.12: Performance of three controllers for simulated three set point problem

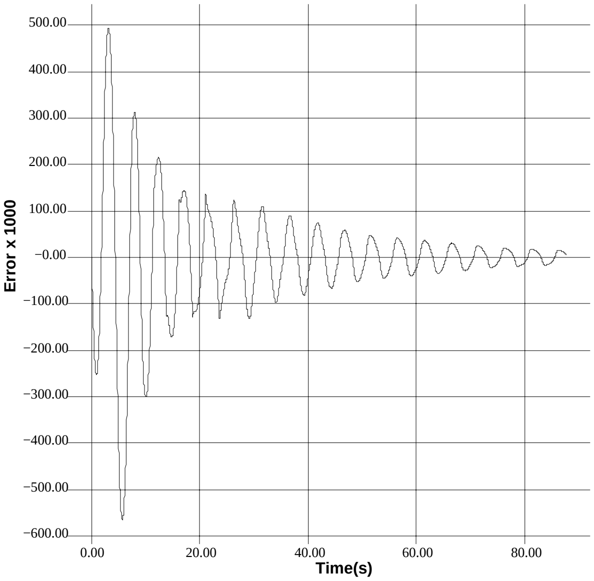
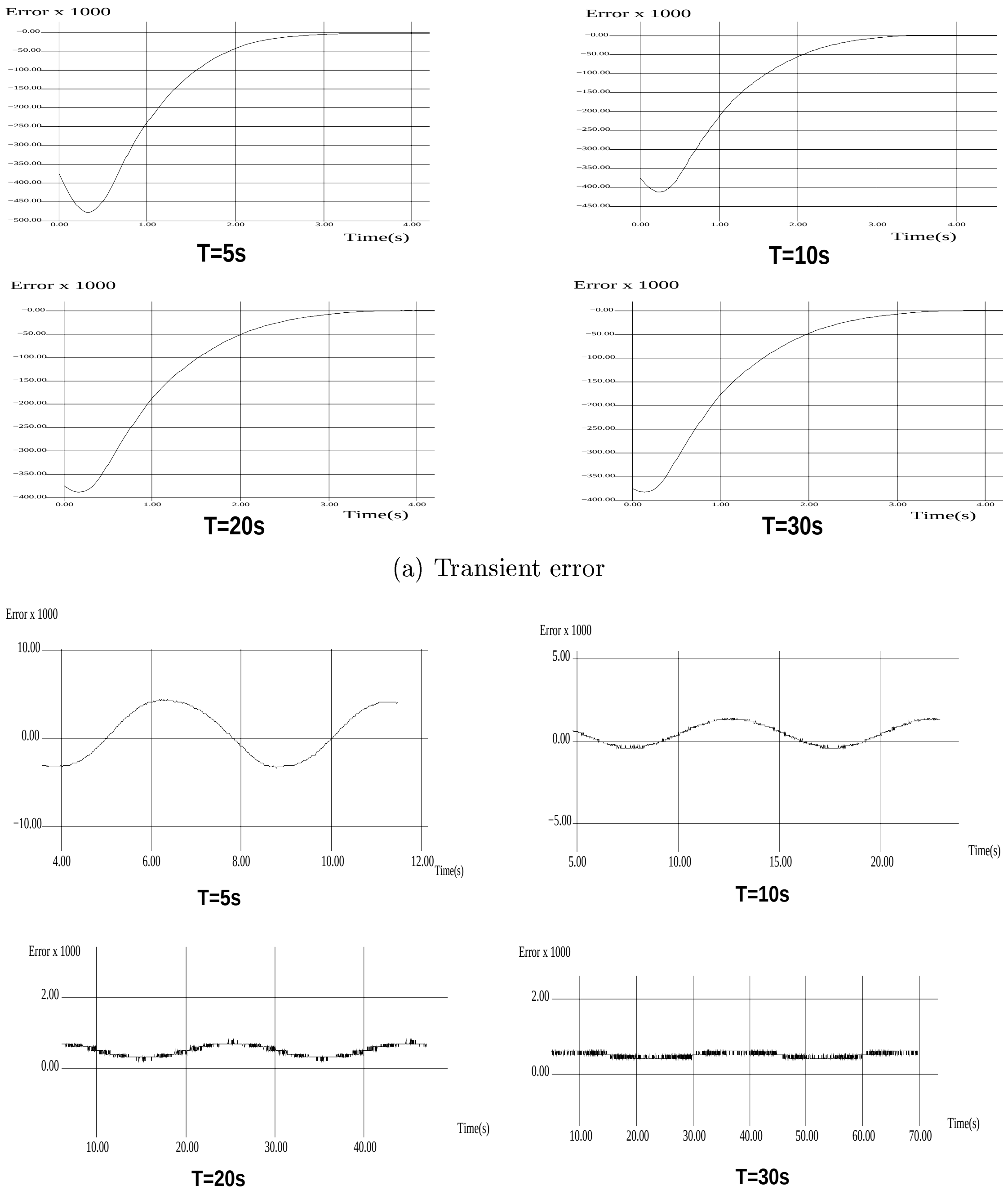


Figure 9.13: Learning sinusoid-tracking performance on simulated system



(a) Transient error

(b) Steady-state error

Figure 9.14: Error in sinusoid-tracking, $T = 5, 10, 20, 30$s

required for sinusoid-tracking, this fuzzy system with only 27 output values and 47 rules performs remarkably well.

Ball-and-Beam Hardware

To demonstrate the effectiveness of this strategy on a real control problem, a ball-and-beam system was built. A diagram and picture of the setup can be found in Figure 9.17. To form the beam, a stainless steel rod and a brass cylinder were used. The brass cylinder was insulated with shrink wrap and wound tightly with 90 feet of very thin tungsten wire. These two rods are set a fixed distance apart to form the beam. At one end, a pivot is attached, and at the other, a servo controls the height of the beam. This allows us to directly control the angle of the beam. A conducting 'ball' rests on the beam. Its position is sensed by applying a voltage to the tungsten wire winding of the brass cylinder, which is returned through the ball and the steel rod. From the resulting current, the position of the ball can be calculated. The 'ball' was actually implemented by an arrangement which looks more like an axle of a train, to provide more weight and contact area for a stronger current. Again, we desire to bring the ball to some desired point by controlling the angle of the beam.

This setup is a little different from the canonical control problem we have described and simulated. The main difference is that the pivot is at one end of the beam instead of in the center. This actually makes the control problem easier by eliminating a singularity at zero ball radius.

There are a number of important non-idealities of this system. These include:

1. **Static Friction**

   Because the ball rests upon a cylinder wrapped with coils of wire, it tends to stop and refuse to move without a large angle of the beam. This is a major non-ideality we must deal with to control the ball on this platform.

2. **Non-linearity of ball position detector**

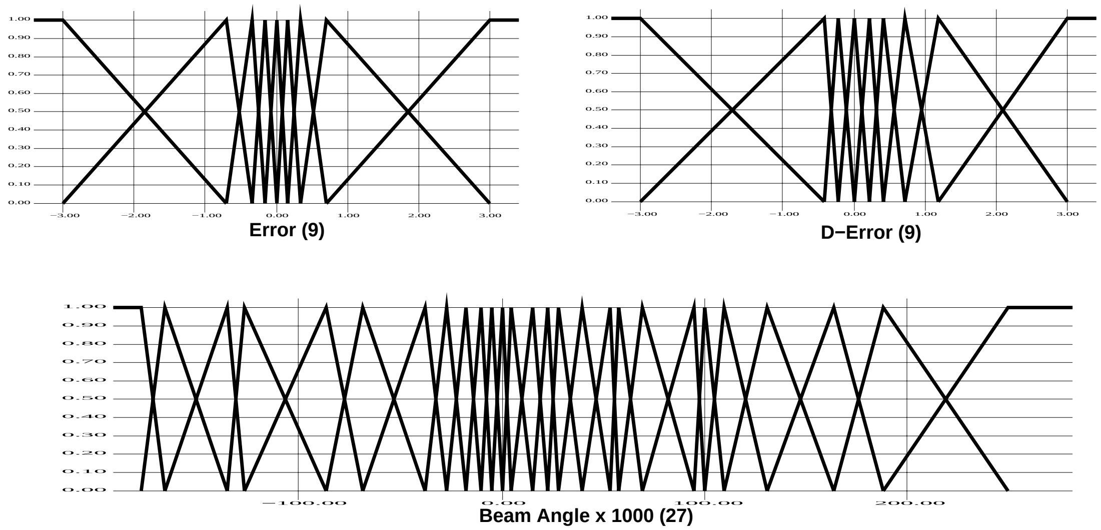
   The tungsten wire which is used to sense the ball position was wrapped by hand around the cylinder. This took approximately four hours and several sittings. Due to a varying concentration level, the spacing of the windings is not uniform. A rough measurement of the actual ball position versus the sensed ball position is shown in Figure 9.18. The piecewise-linear approximation to this function shown as a dotted line is sufficient to explain the effects on the control performance we will observe. This approximation reveals closely-spaced windings at the very beginning, medium-spaced windings in the middle, and an area of particularly widely-spaced windings towards the end. (Points 1, 2, and 3 marked on the diagram will be set points — see Experiment A.)

   If we gave the actual ball position, instead of the position error, as the input to the controller, the nonlinearity of the ball sensor would be unimportant. It is only important because it causes a difference between the responses at different set points to error.

3. **Ball position noise**

   As the ball rolls, the current which can be driven through it varies. In addition, at times it momentarily loses electrical contact with one of the beams, causing

Membership Functions



Rules

Error

DError	0	1	2	3	4	5	6	7	8
0	11	11	11	11	11	11	11	11	11
1	11	8	0	2	7	7	13	12	11
2	11	6	0	3	7	12	19	15	11
3	11	7	4	3	11	23	21	16	11
4	11	8	5	10	14	22	24	18	11
5	11	10	9	17	14	23	26	15	11
6	11	11	12	16	15	25	21	11	11
7	11	11	11	11	11	11	11	11	11
8	11	11	11	11	11	11	11	11	11

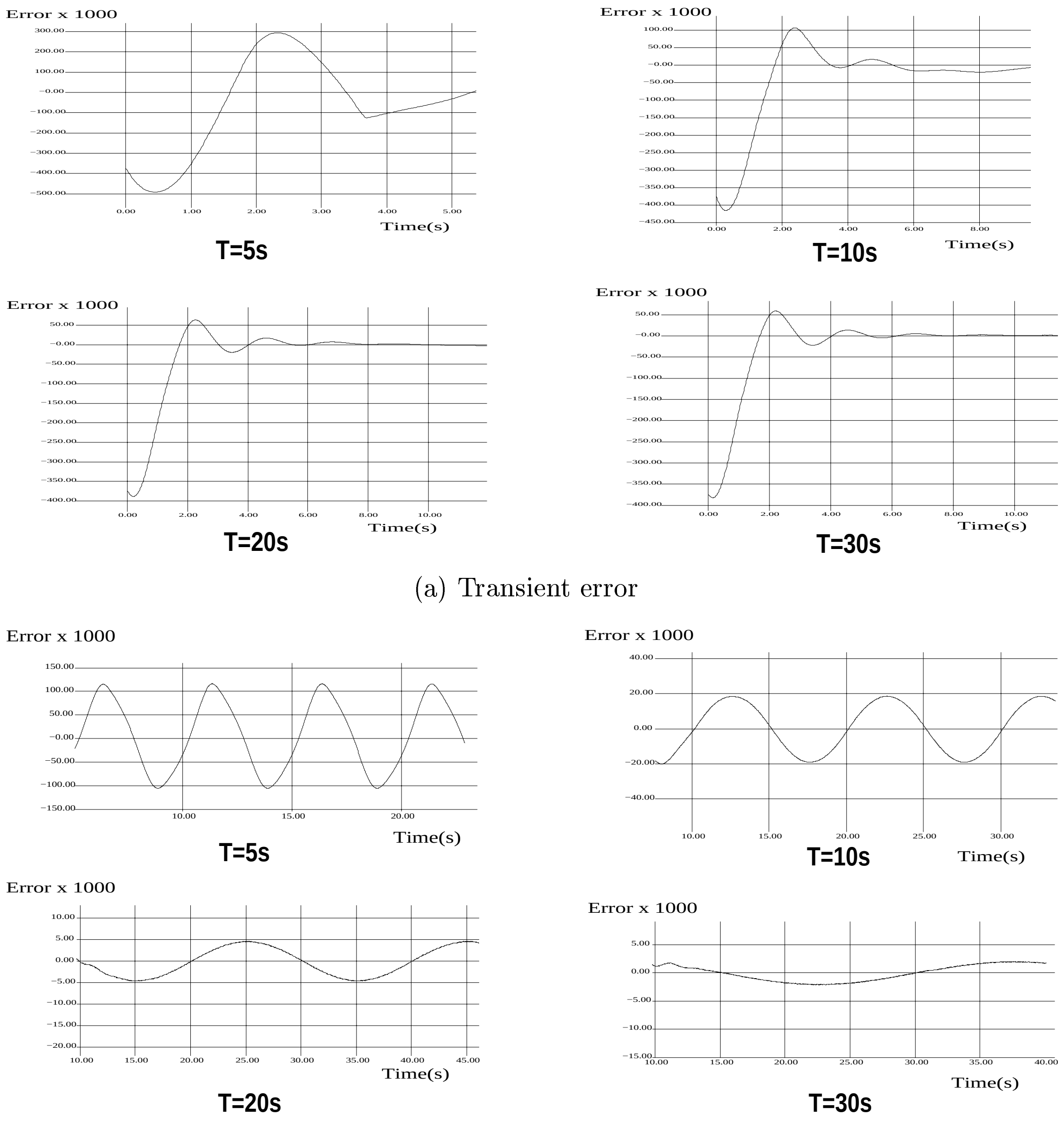
The 81 uncompressed rules

Error

DError	0	1	2	3	4	5	6	7	8
0					11				
1		8	0	2	7	7	13	12	
2		6	0	3	7	12	19	15	
3	11	7	4	3	11	23	21	16	11
4		8	5	10	14	22	24	18	
5		10	9	17	14	23	26	15	
6		11	12	16	15	25	21	11	
7					11				
8					11				

The 47 compressed rules

Figure 9.15: Fuzzy controller for simulated sinusoid-tracking problem



T=5s

T=10s

T=20s

T=30s

(a) Transient error

T=5s

T=10s

T=20s

T=30s

(b) Steady-state error

Figure 9.16: Error in fuzzy sinusoid-tracking, $T = 5, 10, 20, 30$s

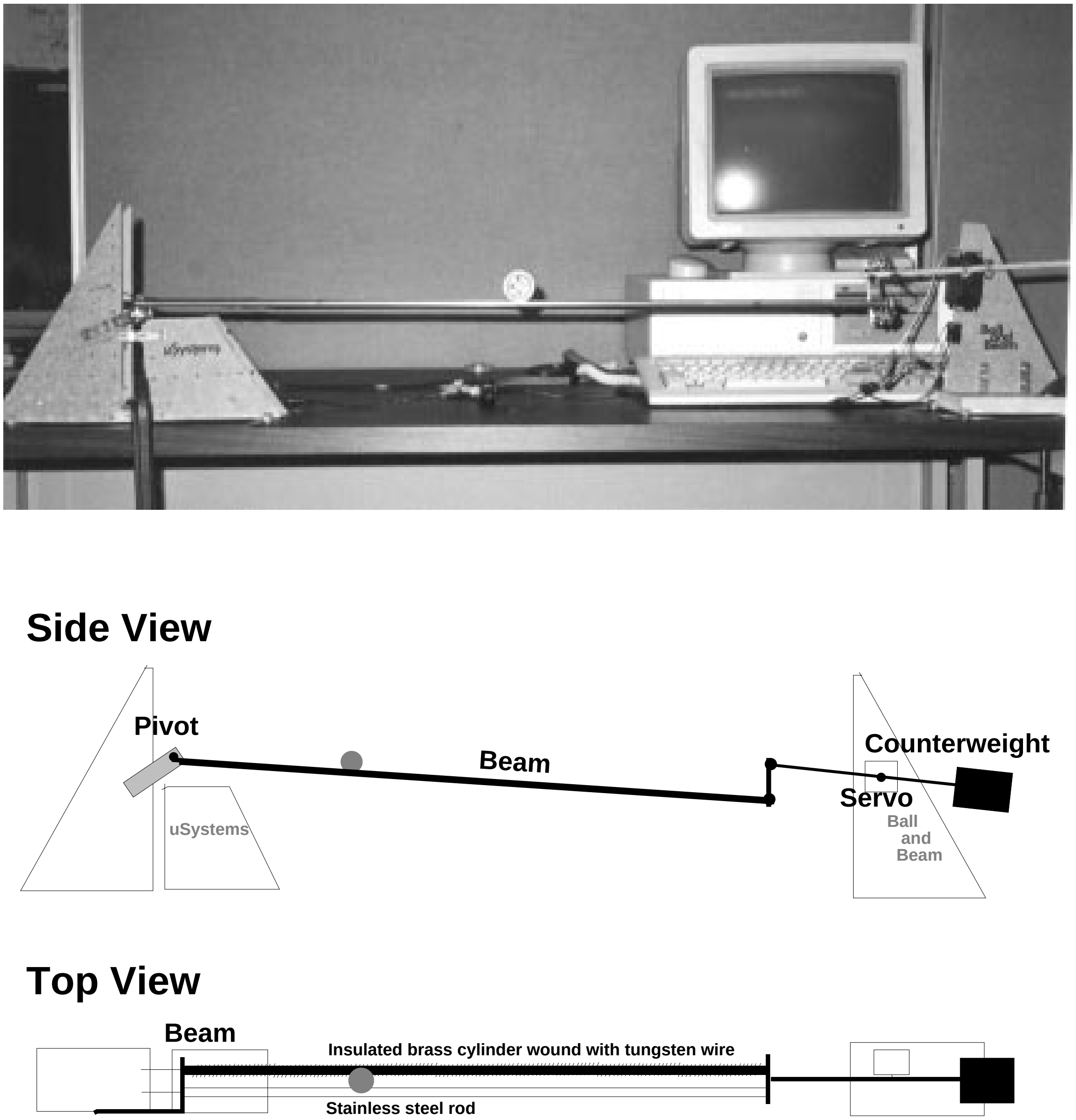


Figure 9.17: Actual ball-and-beam system

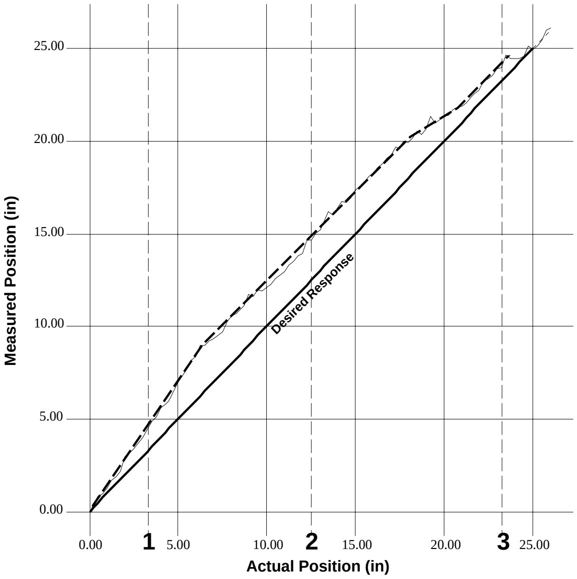


Figure 9.18: Nonlinearity of ball position sensor

'dropouts'. Even with a sound averaging scheme, ignoring dropouts, there is a significant amount of noise in the ball position.

In addition, the voltage over the ball sensor goes as

$$V_{ball} = \frac{R_{ball}}{R_{ball} + 120\Omega} V_{supply}$$

(The 120Ω resistor is used to restrict the maximum current through the circuit.) Thus as R_{ball} goes from zero to its maximum (300Ω) roughly linearly with the ball position, V_{ball} increases sharply and then approaches its maximum asymptotically. This means that at large ball position, there is only a tiny difference in the voltage at slightly different positions. This translates into noise which grows worse at a large ball position. For this reason, precision in position at large ball position is virtually impossible.

4. **Beam sag**

   When the ball is nearer the end of the beam with the servo attached, the beam sags slightly under the weight of the ball. The precise control of the ball position depends strongly upon a constant balanced position of the beam at zero output. This makes controlling the ball at a large ball position even more difficult.

5. **Beam angle noise**

   The beam angle is controlled by a servo, which is positioned by a PWM pulse from the controlling computer. The width of this pulse is between one and three milliseconds. Due to the digital nature of the control and the low time resolution of counters available at this frequency, the output width tends to jump back and forth between two discrete values of pulse width. This results in noise in the controlled beam angle, causing random movement of the ball.

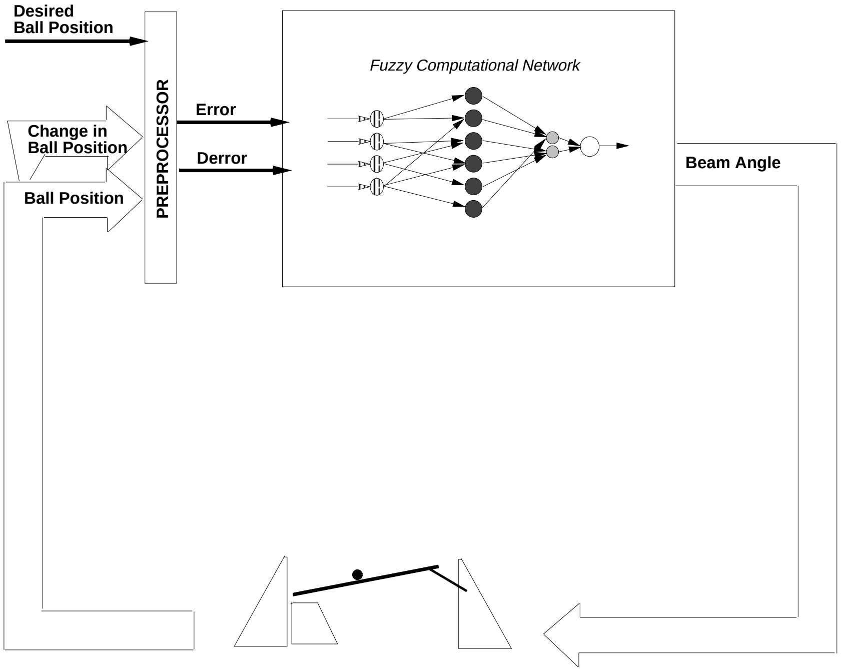


Figure 9.19: Ball-and-beam problem

The loop time of our controller will be 50 milliseconds. Again, our controller will take as input the error from the desired set point

$$E(t) = r(t) - s$$

and the change in error[9] (referred to as *D-error*).

$$\dot{E}(t) = E(t) - E(t-1)$$

A schematic of this control setup is shown in Figure 9.19.

For these experiments, the performance matrix was designed in the same way as for the simulations (Figure 9.8), but with different parameters. These parameters may be found in Section A.2.

Experiment A: Three Set Points

Similar to our first simulation, we will designate three set points s on the beam: at $s_1 = 50$, $s_2 = 150$, and $s_3 = 250$.[10] Due to the nonlinearity of the ball position sensor, we should expect some anomalies in the control response. Due to the closely-spaced windings near s_1, the ball *seems* to be moving farther (i.e., over more windings) than at the other set points. Thus since a disproportionate amount of time is spent training for stability at set point s_2 (this happens because s_2 is the only set point which the ball can overshoot significantly without running off the end of the beam), we should expect oscillation at s_1.

[9] This is only different by the scale factor $1/T_{loop}$ from the error velocity and requires no floating point calculations.

[10] Ball position is reported in sensor units, approximately 11 units per inch.

For the same reason, when the ball approaches s_1, the system will believe it has suddenly sped up and attempt to slow its progress. Conversely, due to the widely-spaced windings between set points two and three, the ball moving at a constant speed will seem to be moving slower (i.e., over less windings); thus the controller will try to speed it up going into this region and slow it down on exit.

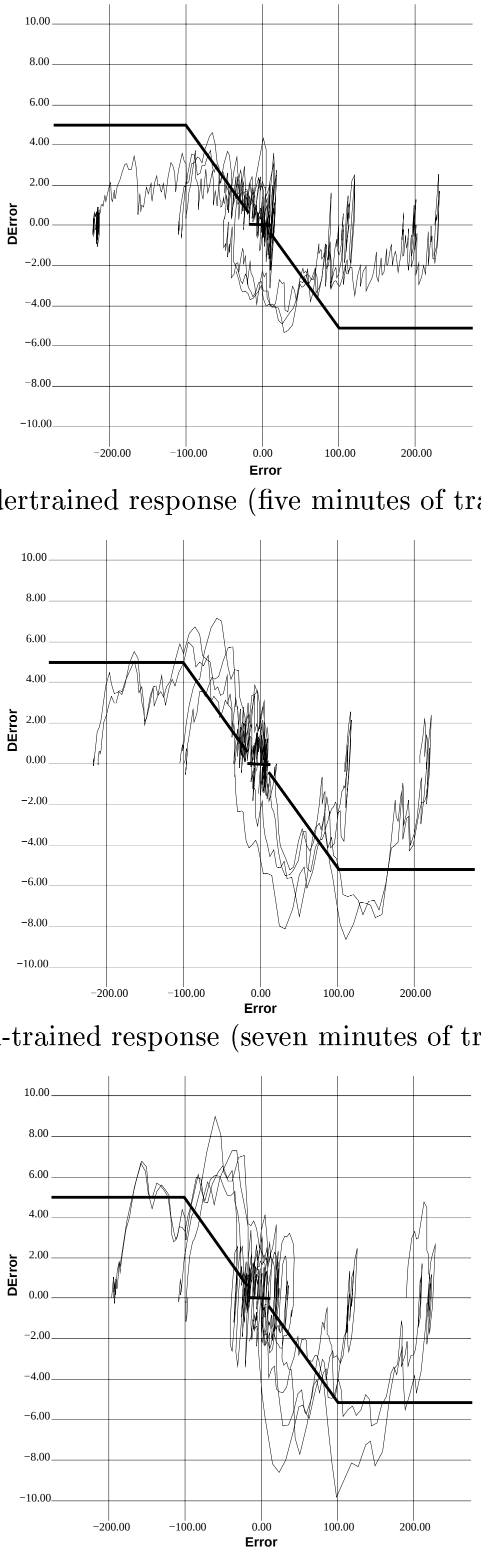
We begin learning with the output matrix set to all zero (such that the default output is to balance the beam). We train the system, as we did the simulated system, by changing the set point between s_1, s_2, and s_3 and waiting for the ball to stop. After five minutes of training, the system is following the input space paths shown in Figure 9.20(a). The ball has barely enough speed to reach each of the set points, taking more than five seconds to reach the next set point; the controller will often lose the ball off one end of the beam because it doesn't catch it fast enough. This controller has barely overcome static friction. After another two minutes of training, the controller has reached its optimum[11] performance (Figure 9.20(b)). This controller exhibits some overshoot and oscillation, but will not lose the ball off either end of the beam at any time, and overcomes the nonlinearity of the ball position sensor quite nicely. If we continue training this controller for only three minutes more, we see increased speed and oscillation (Figure 9.20(c)) as the system attempts to rigorously follow the performance criterion. The performance of the best controller as the set points are changed is plotted in Figure 9.21.

The control surface of the best controller is shown in Figure 9.22. The gentle slope near zero error gives stability about the set point. Note the wave-like shape at large error which speeds up the ball and then slows it down near zero error. Overall, the function bears a remarkable similarity to that of Figure 9.10.

In the first few minutes of training described above, the controller spends a lot of time figuring out just how to get the ball moving over the static friction barrier. Let us design a simple initial fuzzy controller to get the ball moving in an attempt to reduce training time. Such a controller is shown in Figure 9.23. The controller includes rules which will move the ball to the set point with a larger response for a larger error, and some simple rules to slow the ball down near the set point. It is just sufficient to get the ball moving in the right direction. To proceed, we convert this fuzzy controller into a table-based controller and initialize our output matrix with these values. An example of the abysmal performance of the initial table-based controller is shown in Figure 9.24(a). After three minutes of training (less than half the time of the best controller with no head start), we have reached the performance shown in Figure 9.24(b). This system gets the ball to the set point extremely quickly and exhibits a tightly damped oscillation. Taking this controller and learning a fuzzy controller from it results in a second fuzzy system capable of solving the problem (Figure 9.25). The performance of the fuzzy controller with uncompressed rules is shown in Figure 9.26(a), and with compressed rules in Figure 9.26(b). Again, we sacrifice some control accuracy for simplicity, but even the controller with compressed rules is sufficient to solve the problem quite satisfyingly.

It is interesting to compare the best controller learned from nothing with the best controller learned from a fuzzy head start. The best controller learned from nothing has rather slow and steady performance, reaching the set point with a minimum of oscillation.

[11] This controller is optimum only in the sense that earlier in training, the controller would be unable to stop the ball at the set point for a large initial error, and later in training the controller would be unable to maintain the ball at the set point without tremendous oscillation. It would certainly be possible to specify a performance criterion in terms of rise time, overshoot, and oscillation to determine *exactly* when in learning the controller actually reaches its 'best' performance.



(a) Undertrained response (five minutes of training)

(b) Well-trained response (seven minutes of training)

(c) Overtrained response (ten minutes of training)

Figure 9.20: Input space paths of actual system while learning

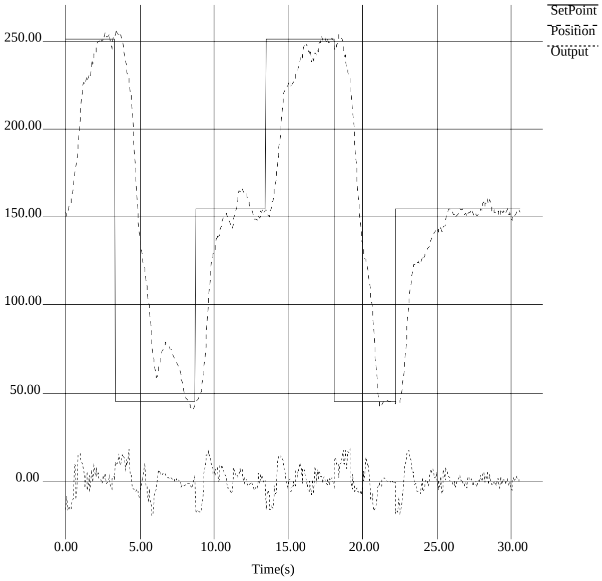


Figure 9.21: Well-trained performance

The best controller learned with a fuzzy head start reaches the set point quickly, and oscillates quickly into the set point. This is because the controller learned from nothing has approached the desired input space response from a slower speed than desired, while the fuzzy head start controller approached the desired input space response from a higher speed than desired.

Experiment B: Sinusoidal Set Point

Our final experiment is that of tracking a sinusoid

$$s(t) = \frac{s_3 + s_1}{2} + \frac{s_3 - s_1}{2}\sin(2\pi\frac{t}{T})$$

This sinusoid moves between s_1 and s_3 of the previous experiment. This experiment is much harder in a real system than in simulation, due to static friction. We began with the learned controller of Figure 9.24(b), which was given a fuzzy head start. This controller obtains the best performance we have seen. The physical system cannot learn to track a sinusoid of period $T = 5$ seconds; the speed required exceeds that which it is capable of controlling. By training this controller for 30 seconds on a sinusoid of period $T = 10$ seconds, we obtained the performance shown in Figure 9.27. At $T = 10$s, the ball is at a comfortable rolling speed most of the time (average speed 5.2 in/s) and only exhibits oscillation at low ball position due to the ball sensor nonlinearity. At $T = 20$s, the required ball speed is so low (average speed 2.6 in/s) that static friction begins to stop the ball periodically, requiring the system to provide a large output to get it moving again. At $T = 30$s, static friction has a strong hold on the ball (average speed 1.7 in/s); the system must constantly restart it. These results would improve if the maximum angular velocity provided by the servo were increased, so the ball could be restarted more sharply.

A fuzzy controller was learned from this table-based controller; it is shown in Figure 9.28 and is virtually identical to that of Figure 9.25. Its performance (with compressed

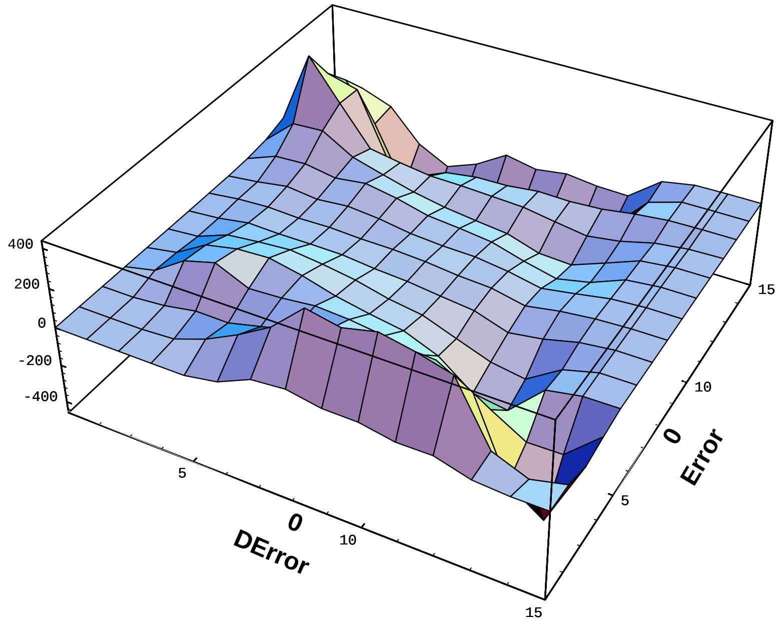


Figure 9.22: Control surface of best controller

rules) is shown in Figure 9.29. As expected, we see increased oscillation and overall a less precise response.

9.2.4 Summary

Simulation A showed us how the ESOC algorithm performs on a highly non-linear problem by demonstrating the input space response of an undertrained, well-trained, and overtrained controller. This gives us insight into the responses of the actual system, which are not as well-defined. We observed that the controllers resulting from the learning algorithm are quite nonlinear. This experiment also showed us that the fuzzy approximation to the table-based system does not perform quite as well, and that the simplified fuzzy system does not perform quite as well as the original fuzzy system.

Simulation B demonstrated learning to track a very fast sinusoid. This standard problem with the ball-and-beam system was solved easily and with very small error. The simplified fuzzy approximation to this controller was shown to operate, but with significantly more transient oscillation and steady-state error.

Experiment A showed similar performance to that of Simulation A on actual hardware. The undertrained, well-trained, and over-trained controllers are not as apparently different. By giving the system a very simple fuzzy system as a head start, we were able to cut the training time in half. The resulting controller was somewhat different from the controller learned from nothing. Again, the fuzzy system resulting from an approximation of the learned table-based controller exhibits inferior, but still satisfactory, performance.

Experiment B demonstrated learning to track a sinusoid with actual hardware. Static

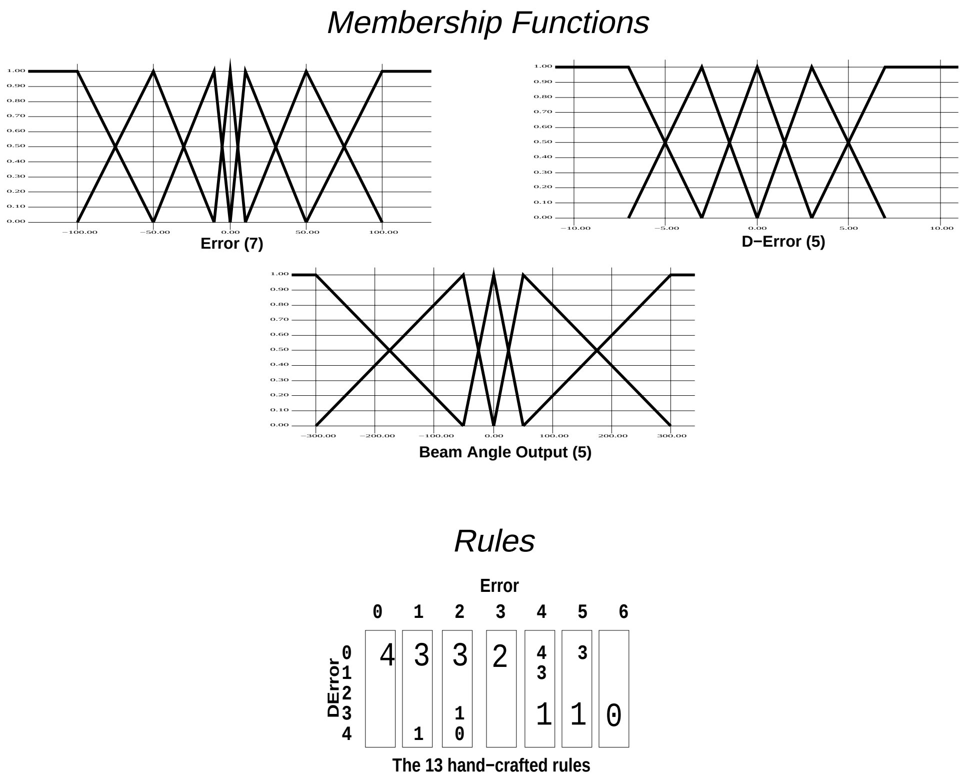
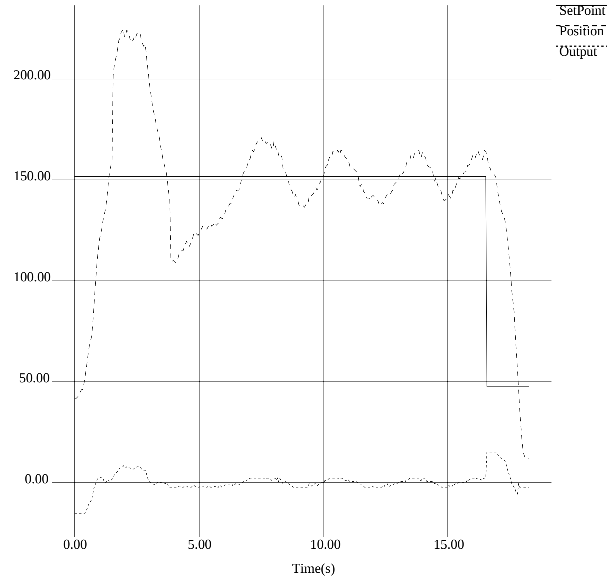
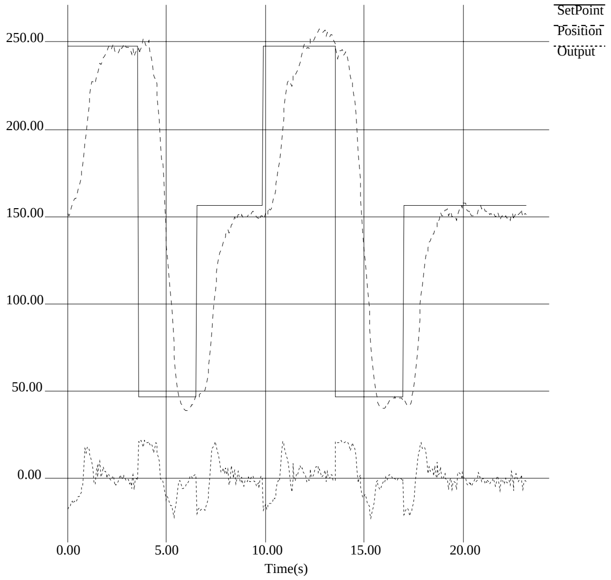


Figure 9.23: Initial fuzzy controller



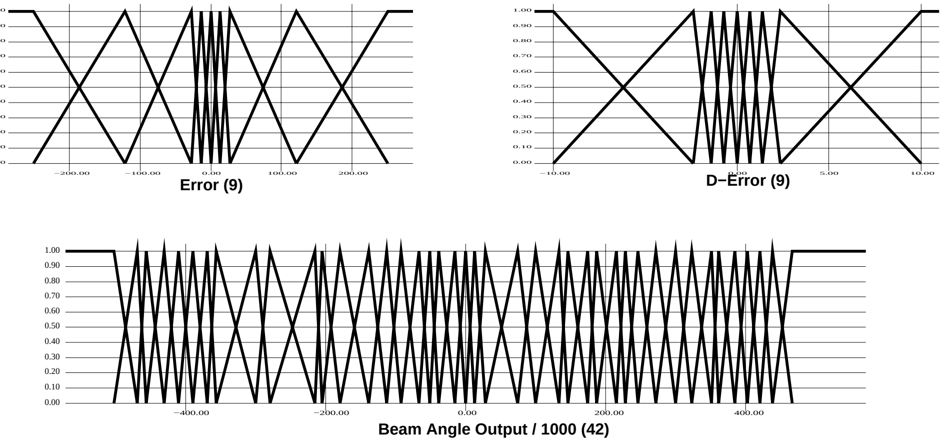
(a) Performance of initial table-based controller



(b) Performance of table-based controller after three minutes of training

Figure 9.24: Learning with a fuzzy head start

Membership Functions



Rules

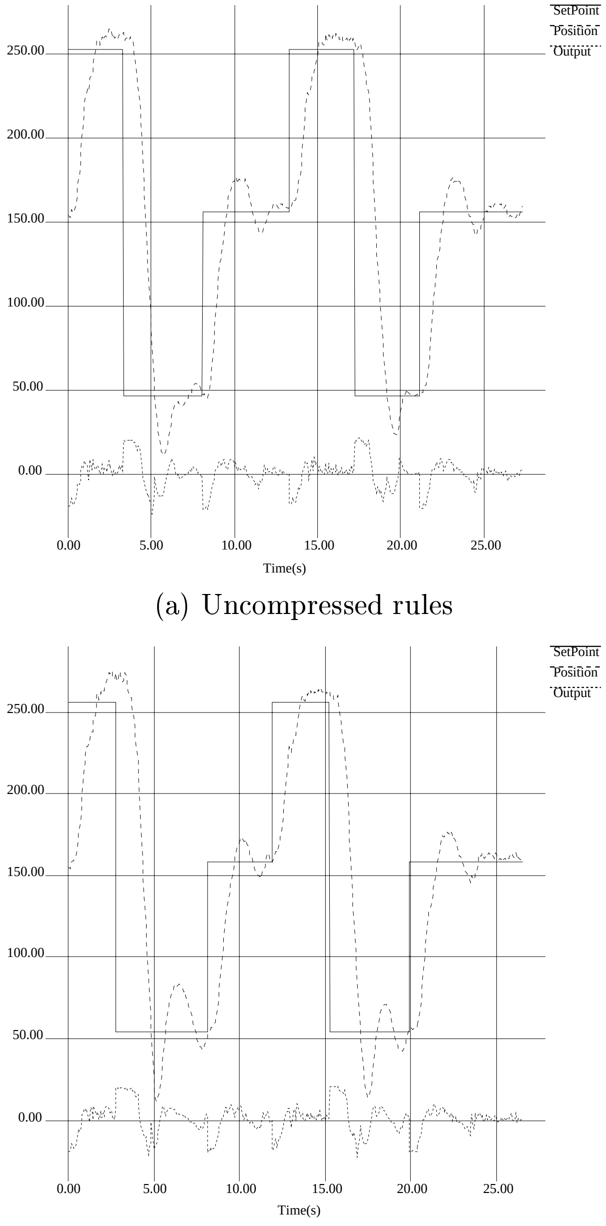
The 81 uncompressed rules (rows: DError; columns: Error)

DError \ Error	0	1	2	3	4	5	6	7	8
0	33	33	36	28	21	34	41	24	9
1	33	34	31	29	22	22	20	3	5
2	33	35	30	27	22	21	18	2	6
3	33	39	25	24	21	18	16	0	6
4	38	41	23	23	20	16	16	0	6
5	37	40	23	21	19	14	15	1	4
6	35	38	21	15	18	12	13	4	4
7	34	35	17	10	17	12	11	6	7
8	32	26	9	13	20	17	16	8	8

The 74 compressed rules (rows: DError; columns: Error)

DError \ Error	0	1	2	3	4	5	6	7	8
0	33	33	36	28	21	34	41	24	9
1		34	31	29	22	22	20	3	5
2		35	30	27	22	21	18	2	6
3		39	25	24	21	18	16	0	
4	38	41	23	23	20	16		0	
5	37	40	23	21	19	14	15	1	4
6	35	38	21	15	18	12	13	4	4
7	34	35	17	10	17	12	11	6	7
8	32	26	9	13	20	17		8	8

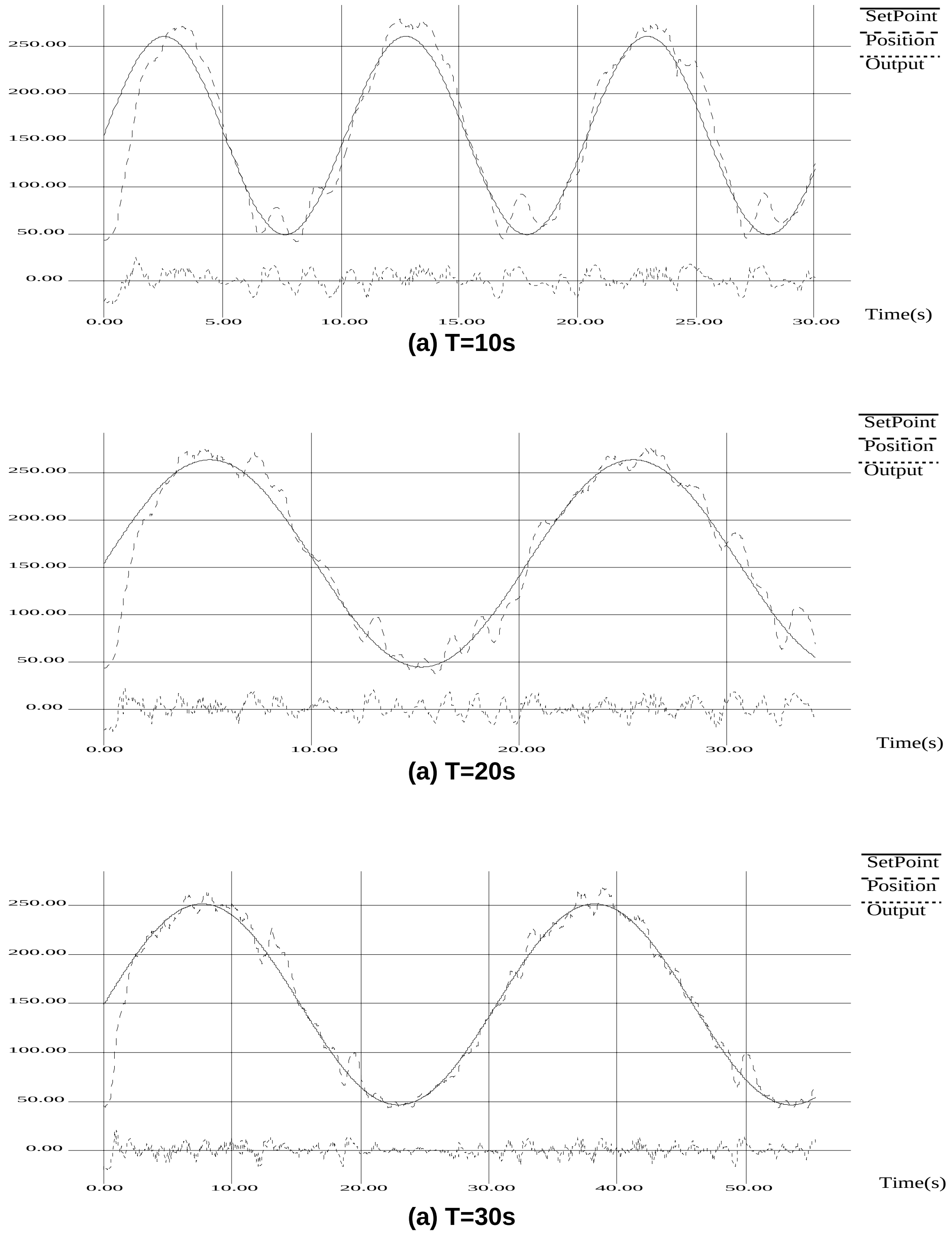
Figure 9.25: Learned fuzzy controller



(a) Uncompressed rules

(b) Compressed rules

Figure 9.26: Learned fuzzy controller performance

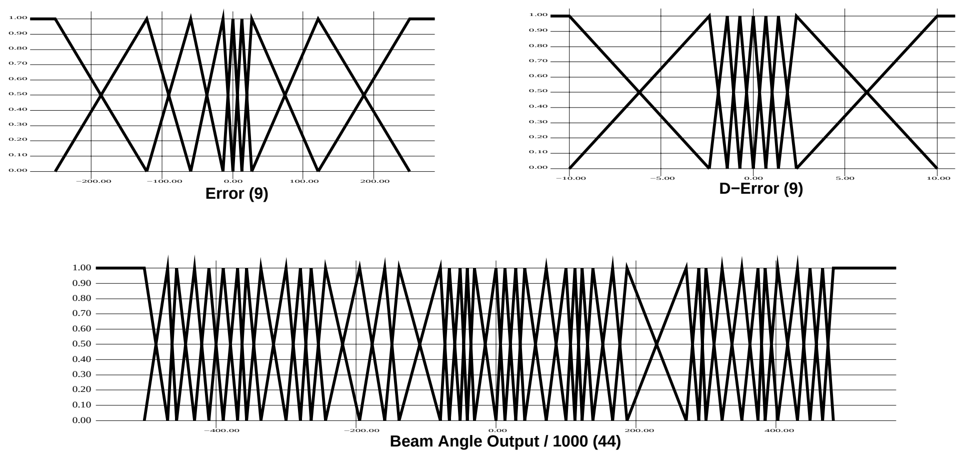


(a) T=10s

(a) T=20s

(a) T=30s

Figure 9.27: Performance of learned controller for sinusoid-tracking



Membership Functions

Rules

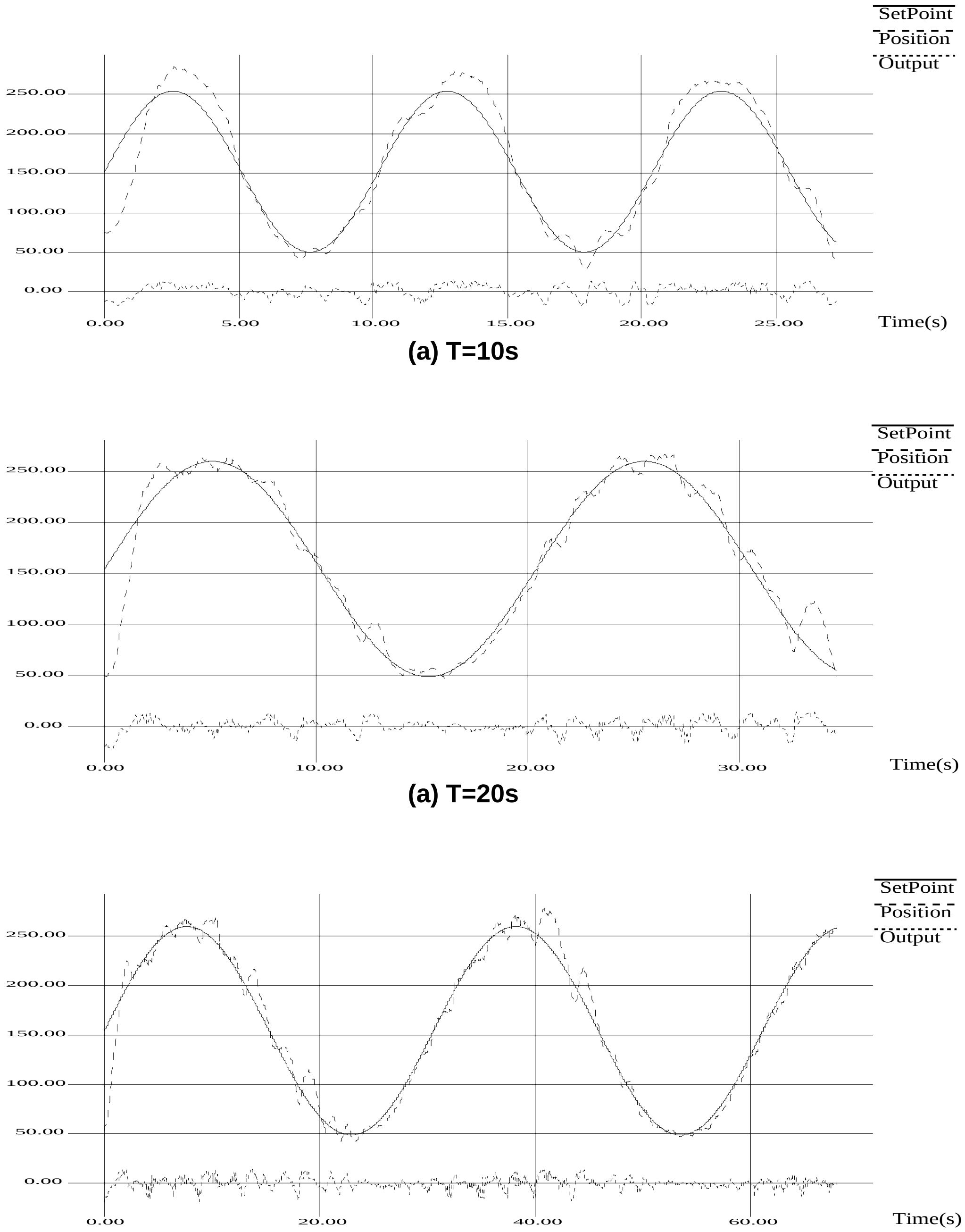
The 81 uncompressed rules (rows: DError; columns: Error)

DError \ Error	0	1	2	3	4	5	6	7	8
0	34	34	28	31	22	35	42	26	10
1	34	39	29	37	24	25	22	3	5
2	34	37	30	33	24	23	19	2	6
3	34	42	30	29	22	20	17	0	6
4	39	43	30	25	21	16	16	0	6
5	38	41	29	21	20	14	15	1	4
6	36	40	29	15	19	12	13	4	4
7	35	36	27	9	17	11	10	6	7
8	32	29	8	15	21	18	17	9	9

The 76 compressed rules (rows: DError; columns: Error)

DError \ Error	0	1	2	3	4	5	6	7	8
0	34	34	28	31	22	35	42	26	10
1		39	29	37	24	25	22	3	5
2		37	30	33	24	23	19	2	6
3		42		29	22	20	17	0	6
4	39	43		25	21	16	16	0	6
5	38	41	29	21	20	14	15	1	4
6	36	40	29	15	19	12	13	4	4
7	35	36	27	9	17	11	10	6	7
8	32	29	8	15	21	18	17	9	9

Figure 9.28: Learned fuzzy controller for sinusoid-tracking



(a) T=10s

(a) T=20s

(a) T=30s

Figure 9.29: Performance of learned fuzzy controller for sinusoid-tracking

friction and the relatively slow response of the hardware combined to make this a difficult experiment. The fuzzy approximation to the learned table-based controller exhibits more oscillation and a less precise response.

9.2.5 Comments and Conclusions

The ball-and-beam experiments we have shown verify the effectiveness of the loop-back approach we have presented for simple control problems. It should be clear that the adaptation algorithm used is not important to the approach in general. For a more complex problem, the ESOC algorithm would prove more difficult to use, due to the manual choice of the performance matrix. If more computing power were available, it would be possible to use a more robust adaptation algorithm to make this approach even more attractive.

The parameters we set for learning fuzzy controllers (Section A.1) determined the complexity and the precision of the fuzzy controllers we have shown. We intended to show that a simple fuzzy controller can approximate a complex control function and obtain adequate control performance. However, the merely *adequate* performance shown for fuzzy controllers in this chapter should not be construed to imply that they cannot function any better. Using the methods we have presented in Chapter 8, it is possible to represent our table-based systems to arbitrary precision. If better control performance is desired, the number of membership functions for each input can be increased, at the cost of a more complex fuzzy controller.

As the number of inputs increases, the difficulty of finding an effective table-based controller will also increase. Due to the geometry of high-dimensional spaces, it will be necessary to specify a larger and larger number of grid points in each dimension to encompass a function over the space. This will also be true of the number of membership functions for each input to the fuzzy system. However, these techniques will tend to scale with dimension better than other function representation techniques which do not have an independent representation for each dimension, including radial basis functions and memory-based systems.

One important comment to make about *learned* controllers is that they only know what they have been taught. For example, a controller trained on the three set point problem would likely not work for the sinusoid tracking problem, because there will be many places along the beam where the controller has never seen the ball stop. Conversely, a controller trained on the sinusoid tracking problem would likely not work for the three set point problem, because it has little experience with large initial errors. This is not to say that you could not train a controller to do either problem (the controller in Figure 9.28 began as a three set point controller and was trained additionally for sinusoid tracking), but it is a fact that one should be aware of in using learned controllers.

It is a testament to the power of nonlinear control systems that we are able to demonstrate actual performance of a ball-and-beam system while only sampling our inputs at 20Hz. A linearized controller for this problem would need to sample at hundreds or thousands of Hertz to maintain the problem within the linearizable region.

On a philosophical point, although we claim to give the controller prior information about the control system only through our initial fuzzy rule set, there is actually quite a lot of information about the system embedded in the performance matrix of ESOC. However, the particular choice of an adaptive learning algorithm is not part of the loop-back approach we have suggested. To provide the most general implementation of this system, a reinforcement learning controller which receives only a binary success/fail signal on each experiment could be used, such as [BSA83].

A final comment that should be made is that the systems we have proposed for control are still in the very early stages of research. The controllers learned are not always entirely robust — while they will usually succeed, sometimes they will encounter a condition they are not prepared for. Neither are they provably stable. However, they can sometimes cleanly solve problems which control theory can at best make an approximation to. There is much that is not well understood, and much left to be done, but learning control systems is clearly interesting.

Chapter 10 Future Work

In this final chapter, we discuss possible improvements to the fuzzy function approximator we have presented and its application to control.

10.1 The Function Approximator

10.1.1 Improvements to the MF Learning Algorithm

There are a number of difficulties with the membership function learning algorithm which need to be looked at.

The choice of a planar approximation to the data as a starting place is seemingly arbitrary. While this does lead to the point of maximum error being the point which will minimize the total RMS error the most, it often turns out that these initial membership functions can be *removed altogether* without increasing the error.[1] However, starting from nothing often leads to the choice of points which start at the greatest value of the function and work down towards zero – this is **not** what we want. Perhaps a better starting point than either of these choices can be found.

There are too many *ad hoc* parameters in the algorithm: they include the 'already-there' thresholds for each variable and the 'too-far' threshold for the output variable. While the values that have been chosen manually seem to be effective for a number of different function approximation problems, these parameters should really be chosen automatically.

Using a fixed percentage of the range of an input variable for the 'already-there' threshold causes problems when the dynamic range of an input is very large. To avoid getting two membership functions at 300.0 and 301.5, we must set the threshold larger than 1.5. However, it may be necessary to have this precision near zero in order to solve the control problem. Perhaps some form of nonlinearity could be introduced.

Finally, the choice of the error weighting function for control systems is crucial to learning just the part of the data that is important to your problem, but we have no insight on the choice of this parameter other than what we have stated.

10.1.2 Rule Weights

In the computational network described in Section 8.4, there are links from the rule nodes to the output nodes which have unit weight. It seems obvious that some performance improvement might be obtained by modifying these weights to modulate the effect of each rule on the output. We have experimented with backpropagating these weights to reduce the error as much as possible after learning the network as described. Unfortunately, this process is extremely time-consuming and resulted in little performance improvement.

[1] We experimented with 'shrinking' the membership functions after learning them by testing the error and decided it was too time-consuming to be worthwhile.

10.1.3 Rotating the Axes

Since we introduced the axis-parallel problem in Section 2.1, it must have been obvious to the reader that a good way to reduce this difficulty for a function approximator would be to include before its inputs a network which rotates the input axes in such a way as to maximize their predictive power. Principal component analysis can provide such a rotational transform, and there is even a neural network which can do such a calculation [BH89]. This rotational transform would lead to inputs which could describe the output in terms of the minimum number of fuzzy rules. However, if one includes this network before the inputs of the function approximator, what semantics is one to assign to the inputs? They will have no easily understood meaning with relation to the physical world from which they came. We have defeated the very purpose of our system, and we may as well use a backpropagation network for all the understandability we have. It seems as if the very ability for which we have valued fuzzy systems has been lost.

10.1.4 Approximation with Noise

As stated in Section 6.1, we have assumed that the input data is noiseless. For the control examples to which we have applied our system, this is a valid assumption since each example is coming from a good control system. However, what if we wanted to apply our fuzzy learning paradigm to examples which did contain noise? Immediately the membership function learning scheme, which picks the point of maximum error to add the next membership function, breaks down since there may be outliers which really need not be approximated. In order to detect such outliers, some assumptions about the data might be made; perhaps a maximum gradient condition would be effective.

10.1.5 Multi-Layered Rule Networks

The systems which we have discussed make rules about the inputs which are directly used to calculate the output. In control systems theory, there is the concept of an *observer* [O'R83]: an intermediate calculation made from the inputs which is then used to calculate the output. An analog to this in fuzzy control would be a layer of rules which do not draw conclusions about the output, but about an intermediate variable.

10.1.6 The Use of the L-measure in Classification

Since the J-measure does not seem to be the best simplicity criterion from any theoretical viewpoint, it may be that use of the L-measure will result in superior performance on some data sets. The L-measure could be used as suggested in Section 8.3.2: start with the alpha parameter very high and decrease it until the classification percentage on the training set reaches the Bayes rate. At this point, you should have the rule set which will generalize the best to the test set.

10.1.7 The Use of Lateral Weights in Classification

Rather than pruning rules for independence, perhaps some version of the lateral inhibitory weights could be used for classification. A version of this has been tried in the past, with mixed success. The benefits of this would be to allow greater redundancy and save rule pruning time.

10.2 Application to Control

10.2.1 A Better Compression Criterion

As a metric of the error caused by our rule compression, we have used the percentage classification error that SQUEEZE gets on our original cell-based rule set. This is somewhat misleading, since even with every original cell-based rule correctly classified, there can still be some degradation of the control performance. A better metric of compression error would take into account the difference in the function approximated by the uncompressed and compressed rules and how this difference might affect control performance.

10.2.2 Connection to Control Theory

There are a number of papers citing similar performance on the sinusoid-tracking problem with the same simulated ball-and-beam system we have presented [Cas90, Tee91, HSK92, HR92]. It would be interesting to determine if the learning system has discovered a strategy similar to one of those suggested by these authors.

10.2.3 Dependent Rule Inference Scheme

The fact that there is any difference at all between the performance of the compressed and uncompressed rule sets shown in Section 9.2 shows that the dependent rule inference scheme we have suggested does not work perfectly. The rule sets shown were compressed with *no error* (see Section A.1) from the original rule set. Looking at the rule tables, it can be seen that not a single rule has been lost. However, the surfaces created by the two rule sets are different. The trouble seems to happen when two first-order rules with the same conclusion overlap. In this case, there is twice as much weight for their mutual conclusion as there would be in the uncompressed rule set.

Appendix A
Details of Ball-and-Beam Experimentation

A.1 Function Approximation Parameters

The same parameters were used to learn all of the fuzzy controllers shown in Section 9.2 of the thesis.

To obtain the simplest possible fuzzy controllers, the RMS error maximum was set to zero and the number of membership functions varied by hand. Nine memberships for both error and derror was chosen after some experimentation. The examples were given a minimum weight $M = 1 \times 10^{-10}$ (see Section 8.2.2). The output membership functions were allowed to be no more than 1.0% of the output range from the actual output value, and the input membership functions were not allowed to be closer than 2.5% of their range.

The rules were compressed by varying the compression parameter α starting at 100 and working down in steps of 5 until the first rule prediction error was seen. This step is quite large, but resulted in a fast solution. Each compressed rule set shown is as compressed as it can be with absolutely no error in predicting the original cell-based rule set, to the stated granularity of α.

A.2 ESOC Parameters

Recalling the operation of ESOC, a fixed delay is used to attribute the current state to a former state's output. This delay was determined by looking at the impulse response of the system to a large number of input levels. The quick response of the hardware allowed us to set this parameter to one sample, or 50ms.

The following describes the parameters necessary to specify the output matrix and performance matrix for each experiment. The value of each parameter used for each experiment is listed at the end of this appendix.

Maxima

In order to construct the output matrix, it is necessary to choose a maximum value for each of the inputs. Let the value of the maximum for input j be called M_j.

Grid Size

Let the size of the grid for input j be called S_j. S_j must be odd, so that there is a point at zero.

Grid Spacing

The grids used for the output matrix and performance matrix were nonlinearly spaced, to allow more precision near zero. For attribute j, the formula used for placement of the grid points is as follows:

$$p(i,j) = M_j sgn(i) \frac{1 - e^{-\beta|i/s|}}{1 - e^{-\beta}} \qquad (-s \leq i \leq s)$$

where $s = \lfloor S_j/2 \rfloor$ and i is an integer.

The Performance Matrix

Refer to Figure 9.8. The performance matrix value at each grid point is calculated as a gain parameter G times the Euclidean distance to the nearest constraint line. The slope S and the error parameters E_{far} and E_{near} define these constraints.

A.2.1 Simulations A and B

- Grid:

	Maximum M	Grid Size S	Nonlinearity β
Error	1.0m	15	5.0
D-Error	1.0m/s	15	3.0

- Performance Matrix:

$$E_{near} = 0.0m \quad E_{far} = 0.33m \quad S = 1.5$$
$$G = 0.02$$

A.2.2 Experiments A and B

- Grid:[1]

	Maximum M	Grid Size S	Nonlinearity β
Error	250.0su	15	5.0
D-Error	10su/iteration	15	3.0

- Performance Matrix:

$$E_{near} = 10.0su \quad E_{far} = 100.0su \quad S = 0.05$$
$$G = 1.5$$

[1] su = sensor units

References

[Alb75] J. Albus, "A new approach to manipulator control: The Cerebellar Model Articulation Controller (CMAC)," *Transactions of the ASME*, pages 220–27, 1975.

[Bez92] J. Bezdek, *Fuzzy Models for Pattern Classification*, IEEE Press, New York, NY, 1992.

[BH89] P. Baldi and K. Hornik, "Neural networks and principal component analysis," *Neural Networks*, 2:53–58, 1989.

[Bla37] M. Black, "Vagueness: An exercise in logical analysis," *Philosophy of Science*, 4:427–55, 1937.

[BM78] B. Buchanan and T. Mitchell, "Model-directed learning of production rules," In D. Waterman and F. Hayes-Roth, editors, *Pattern-Directed Inference Systems*, pages 297–312. Academic Press, New York, NY, 1978.

[Bru92] J. F. Brulé, "Fuzzy systems – a tutorial," *Internet posting on COMP.AI*, January 2, 1992.

[BSA83] A. Barto, R. Sutton, and C. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–46, September/October 1983.

[Cas90] B. Castillo, "Almost tracking though singular points via the nonlinear regulator theory," *Preprint*, University of Roma, La Sapienza, 1990.

[CCC93] S. Cheon, S. Chang, and H. Chung, "Development strategies of an expert system for multiple alarm processing and diagnosis in nuclear-power-plants," *IEEE Transactions on Nuclear Science*, 40(1):21–30, 1993.

[dBAN92] F. d'Alché Buc, V. Andrès, and J. Nadal, "Learning fuzzy control rules with a fuzzy neural network," *Proceedings of the International Conference on Artificial Neural Networks*, 1992.

[DH73] R. Duda and P. Hart, *Pattern Classification and Scene Analysis*, John Wiley, New York, NY, 1973.

[EL88] O. Ekeberg and A. Lansner, "Automatic generation of internal representations in a probabilistic artificial neural network," *Proceedings of the First European Conference on Neural Networks*, June 6-9 1988.

[GHMS92] R. Goodman, C. Higgins, J. Miller, and P. Smyth, "Rule-based networks for classification and probability estimation," *Neural Computation*, 4(6), November 1992.

[Gon93] R. Gonzalez, "Learning by on-line state space refinement," *AAAI Spring Symposium on Incremental Learning*, March 1993.

[Gra91] I. Graham, "Fuzzy logic in commercial expert systems – results and prospects," *Fuzzy Sets and Systems*, 40(3):451–72, 1991.

[GS86] B. Gaines and M. Shaw, "Induction of inference rules for expert systems," *Fuzzy Sets and Systems*, 18:315–328, 1986.

[GS88] R. Goodman and P. Smyth, "An information-theoretic model for rule-based systems," *Proceedings of the International Symposium on Information Theory*, Kobe, Japan, 1988.

[GS93] M. Goring and H. Schecker, "Hazexpert — an integrated expert system to support hazard analysis in process plant-design," *Computers and Chemical Engineering*, 17(S):429–34, 1993.

[GSW92] S. Geva, J. Sitte, and G. Willshire, "A one neuron truck backer upper," *Proceedings of the International Joint Conference on Neural Networks*, II:850–6, 1992.

[HBZ92] G. Holzer, W. Bertsch, and Q. Zhang, "Design criteria of a gas-chromatography mass-spectrometry based expert system for arson analysis," *Analytica Chimica Acta*, 259(2):225–35, 1992.

[HFU92] S. Horikawa, T. Furuhashi, and Y. Uchikawa, "On fuzzy modeling using fuzzy neural networks with the back-propagation algorithm," *IEEE Transactions on Neural Networks*, 3(5):801–6, September 1992.

[HHNT86] J. Holland, K. Holyoak, R. Nisbett, and P. Thagard, *Induction: Processes of Inference, Learning and Discovery*, MIT Press, Cambridge, MA, 1986.

[HR92] J. Huang and W. Rugh, "An approximation method for the nonlinear servomechanism problem," *IEEE Transactions on Automatic Control*, 37(9):1395–8, September 1992.

[HSK92] J. Hauser, S. Sastry, and P. Kokotović, "Nonlinear control via approximate input-output linearization: The ball and beam example," *IEEE Transactions on Automatic Control*, 37(3):392–8, March 1992.

[JY92] R. Jenkins and B. Yuhas, "A simplified neural-network solution through problem decomposition: The case of the truck backer-upper," *Neural Computation*, 4(5), September 1992.

[KB78] A. Kandel and W. Byatt, "Fuzzy sets, fuzzy algebra, and fuzzy statistics," *Proceedings of the IEEE*, 66(12):1619–39, December 1978.

[KG85] A. Kaufmann and M. Gupta, *Introduction to Fuzzy Arithmetic*, Van Nostrand Reinhold, New York, NY, 1985.

[Kon89] I. Kononenko, "Bayesian neural networks," *Biological Cybernetics*, 61:361–370, 1989.

[Kor67] S. Korner, "Laws of thought," In *Encyclopedia of Philosophy*, volume 4, pages 414–17. MacMillan, New York, NY, 1967.

[Kos92] B. Kosko, *Neural Networks and Fuzzy Systems*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[LBFL80] R. Lindsay, B. Buchanan, E. Feigenbaum, and J. Lederberg, *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project*, McGraw-Hill, New York, NY, 1980.

[Lee90] C. C. Lee, "Fuzzy logic in control systems: Fuzzy logic controller," *IEEE Transactions on Systems, Man, and Cybernetics*, 20(2):404–35, March/April 1990.

[Lej67] C. Lejewski, "Jan Lukasiewicz," In *Encyclopedia of Philosphy*, volume 5, pages 104–7. MacMillan, New York, NY, 1967.

[LL91] C. Lin and C. Lee, "Neural-network-based fuzzy logic control and decision system," *IEEE Transactions on Computers*, 40:1320–36, December 1991.

[MA92] A. Moore and C. Atkeson, "An investigation of memory-based function approximators for learning control," *MIT Technical Report*, July 1992.

[MC68] D. Michie and R. Chambers, "Boxes: An experiment in adaptive control," In E. Dale and D. Michie, editors, *Machine Intelligence*. Oliver and Boyd, 1968.

[MC80] R. Michalski and R. Chilausky, "Learning by being told and learning from examples," *International Journal of Policy Analysis and Information Systems*, 4:125–161, 1980.

[McD80] J. McDermott, "R1: An expert in the computer systems domain," *Proceedings of AAAI 1*, pages 269–271, 1980.

[Mil93] J. Miller, *Building Probabilistic Models from Databases*, PhD thesis, California Institute of Technology, 1993.

[Mit77] T. M. Mitchell, "Version Spaces: an approach to rule revision during rule induction," *Proceedings of the International Joint Conference on Artificial Intelligence*, 5:305–10, 1977.

[MMS91] C. McMillan, M. Mozer, and P. Smolensky, "Learning explicit rules in a neural network," *Proceedings of the International Joint Conference on Neural Networks*, 2:83–8, Seattle, WA, 1991.

[MWM91] M. Mulholland, N. Walker, and F. Maris, "Expert system for repeatability testing of high-performance liquid- chromatographic methods," *Journal of Chromatography*, 550(1-2):257–66, 1991.

[New68] A. Newell, "On the analysis of human problem solving protocols," In J. Gardin and B. Jaulin, editors, *Calcul et Formalisation dans les Sciences de l'Homme*, pages 146–85. CNRS, Paris, 1968.

[Nil80] N. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Co., Palo Alto, CA, 1980.

[NS65] A. Newell and H. Simon, "An example of human chess play in the light of chess playing programs," In N. Weiner and J. Schade, editors, *Progress in Biocybernetics*. Elsevier Press, Amsterdam, 1965.

[NS72] A. Newell and H. A. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[NW89] D. Nguyen and B. Widrow, "The truck backer-upper: An example of self-learning in neural networks," *Proceedings of the International Joint Conference on Neural Networks*, 2:357–63, June 1989.

[O'R83] J. O'Reilly, *Observers for Linear Systems*, Academic Press, New York, NY, 1983.

[PG90] T. Poggio and F. Girosi, "Networks for approximation and learning," *Proceedings of the IEEE*, 78(9), September 1990.

[Pia89] G. Piatetsky, "Discovery of strong rules in databases," *Proceedings of the IJCAI Workshop on Knowledge Discovery in Databases*, pages 264–74, 1989.

[Pos43] E. L. Post, "Formal reductions of the general combinatorial decision problem," *American Journal of Mathematics*, 65:197–268, 1943.

[PSF91] G. Piatetsky-Shapiro and W. Frawley, editors, *Knowledge Discovery in Databases*, AAAI Press, Menlo Park, CA, 1991.

[Qui87] J. Quinlan, "Generating production rules from decision trees," *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 304–7, 1987.

[Qui91] J. Quinlan, "Foreword," In G. Piatetsky-Shapiro and W. Frawley, editors, *Knowledge Discovery in Databases*, pages ix–xii. AAAI Press, Menlo Park, CA, 1991.

[RGV92] B. Rosen, J. Goodwin, and J. Vidal, "Process control with adaptive range coding," *Biological Cybernetics*, 66(4), March 1992.

[RHW86] D. Rumelhart, G. Hinton, and R. Williams, "Learning internal representations by error propagation," In D. Rumelhart and J. McClelland, editors, *Parallel Distributed Processing*, chapter 8. MIT Press, Cambridge, MA, 1986.

[RM86] D. Rumelhart and J. McClelland, editors, *Parallel Distributed Processing*, pages 91–96, MIT Press, Cambridge, MA, 1986.

[Ros62] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan Books, Washington, DC, 1962.

[SAB$^+$73] E. Shortliffe, S. Axline, B. Buchanan, T. Merigan, and S. Cohen, "An artificial intelligence program to advise physicians regarding antimicrobial therapy," *Computers and Biomedical Research*, 6:544–560, 1973.

[SM85] E. Scharf and M. Mandic, "The application of a fuzzy controller to the control of a multi-degree of freedom robot arm," In M. Sugeno, editor, *Industrial Applications of Fuzzy Control*, pages 41–61. Elsevier Science, Palo Alto, CA, 1985.

[SMM91] J. Schlimmer, T. Mitchell, and J. McDermott, "Justification-based refinement of expert knowledge," In G. Piatetsky-Shapiro and W. Frawley, editors, *Knowledge Discovery in Databases*, pages 397–410. AAAI Press, Menlo Park, CA, 1991.

[Smy88] P. Smyth, *The Application of Information Theory to Problems in Decision Tree Design and Rule-Based Expert Systems*, PhD thesis, California Institute of Technology, 1988.

[Sug92] M. Sugeno, "Fuzzy control: Principles, practice and perspectives," *IEEE International Conference on Fuzzy Systems*, March 1992.

[Tee91] A. R. Teel, "Toward larger domains of attraction for local nonlinear control schemes," In *Proceedings of the European Control Conference*, 1991.

[TH93] V. Tresp and J. Hollatz, "Network structuring and training using rule-based knowledge," *Advances in Neural Information Processing Systems*, 5, 1993.

[TTE91] D. Thompson, T. Tomski, and S. Ellacott, "An expert system for the preliminary design of timber roofs," *Computers and Structures*, 40(1):115–25, 1991.

[Utt59] A. M. Uttley, "The design of conditional probability computers," *Information and Control*, 2:1–24, 1959.

[vot85] *Congressional Quarterly Almanac, 98th Congress*, Congressional Quarterly Inc., Washington, DC, Second session, 1984 1985.

[WF65] M. Waltz and K. Fu, "A heuristic approach to reinforcement learning control systems," *IEEE Transactions on Automatic Control*, AC-10:390–98, 1965.

[WM92] L. Wang and J. Mendel, "Fuzzy basis functions, universal approximation, and orthogonal least-squares learning," *IEEE Transactions on Neural Networks*, 3(5), September 1992.

[Zad65] L. Zadeh, "Fuzzy sets," *Information and Control*, 8:338–53, 1965.