Constructive learning: Inducing grammars and neural networks

by

Rajesh Girish Parekh

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Major Professor: Vasant G. Honavar

Iowa State University

Ames, Iowa

1998

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

Rajesh Girish Parekh

has met the dissertation requirements of Iowa State University

Committee Member

Committee Member

Committee Member

Committee Member

Major Professor

For the Major Program

For the Graduate College

Dedication

To my parents Smt. Usha Parekh and Shri Girish Parekh.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

At the outset, I express my heart-felt gratitude to my advisor Dr. Vasant Honavar for his encouragement and guidance all through these years. He has been a profound influence in molding my career. He has not only inspired me to pursue several challenging problems in machine learning but also encouraged me to establish my personal views and opinions about the research I undertook. He has been exemplary in all the three roles of a research advisor, teacher, and teaching assistant supervisor. I thank him for being accessible always, giving a patient ear to all my half-baked ideas, providing invaluable feedback on my work, and going out of his way to offer me reassurance and candid advice every so often when I have found myself dejected or confused. The research funding he provided enabled me to focus exclusively on my dissertation research. Most importantly, he has been a good friend all along.

I would like to thank Dr. Jack Lutz for introducing me to *Kolmogorov Complexity* and related topics which have played a central role in my work on learning DFA from *simple* examples. I thoroughly enjoyed each of the three courses I took with Jack. His influence on me as a model teacher and researcher is immeasurable. I am indebted to him for leading by example and teaching me several lessons in personal and professional ethics.

I enjoyed very much the discussions I had with Dr. Giora Slutzki on grammar inference. His frank critique and insightful comments enabled me to improve my work considerably. I thank Dr. Johnny Wong, Dr. Tom Barta, and Dr. Sarah Nusser for being on my committee. The *Operating Systems* course I took with Dr. Wong is among the most exciting courses I have taken at Iowa State. My interaction with Dr. Tom Barta and Dr. Sarah Nusser has also been very fruitful.

I owe a lot to my colleagues from the AI research group. Some of my most pleasant

memories are in working jointly with Jihoon Yang on the constructive algorithms project. The research results described in chapters 8 and 9 are part of joint work with Jihoon. In Jihoon I have found not only an able research partner but also a caring and considerate friend. Karthik Balakrishnan is also a good friend. Together, Jihoon and Karthik have ever so often lifted my flagging spirits and urged me to keep pursuing my goals. Codrin Nichitiu collaborated with me on the design and development of the incremental regular grammar inference algorithm presented in chapter 5 of this dissertation. He showed tremendous enthusiasm at a time when almost every idea we explored seemed to be leading to a dead-end. I enjoyed working with Carl Pecinovsky who helped in writing code for the *IID* algorithm and Jeremy Ludwig who designed the GUI for our constructive neural networks toolbox.

I am especially thankful to my closest friends Radhika Woodruff, Arun Narayanan, and Ravi Godbole for without their affection and unfailing support, life in Ames would just not have been the same. Thanks to Radhika also for carefully proof reading portions of this dissertation. To my current room-mates Raghu and Venky, I owe special gratitude. With their constant humor and jokes they have so often showed me the lighter side of life. Being with them is always a lot of fun. I hold Rajeev Thakur and Kishore Dhara in high esteem and have always counted on them for giving me frank advice. I am extremely lucky to have several wonderful friends and well-wishers both in Ames and in Mumbai. My sincere thanks to each of my friends whom I will not be able to mention individually.

I am grateful to Dr. Arthur Oldehoeft and the faculty and staff of the Computer Science Department at Iowa State University for the wonderful time I had during my graduate student career. I thank Mrs. and Mr. Tomlinson and Dr. Oldehoeft for giving me the opportunity to teach CS 103. Teaching CS 103 has been a great experience. I would like to thank the computer science office staff for all their assistance. In particular, the exuberance of Judy Kubera, Trish Stauble, Lynn Bremer and now Melaine Eckhart has made my interaction with them a very pleasant one. I am indebted to my current managers Mr. Gary Kerr and Mr. Tom Warden at Allstate Research and Planning Center for the patience they showed as I worked towards completing my dissertation.

Above all, thanks to my parents for it is only because of their constant encouragement, loving support, innumerable sacrifices, and abundant blessings that I have reached this far in life.

ABSTRACT

This dissertation focuses on two important areas of machine learning research – *regular grammar inference* and *constructive neural network learning algorithms.*

Regular grammar inference is the process of learning a target regular grammar or equivalently a *deterministic finite state automaton* (DFA) from labeled examples. We focus on the design of efficient algorithms for learning DFA where the learner is provided with a *representative* set of examples for the target concept and additionally might be guided by a teacher who answers *membership queries.* DFA learning algorithms typically map a given *structurally complete* set of examples to a *lattice* of finite state automata. Explicit enumeration of this lattice is practically infeasible. We propose a framework for implicitly representing the lattice as a *version space* and design a provably correct search algorithm for identifying the target DFA. Incremental or online learning algorithms are important in scenarios where all the training examples might not be available to the learner at the start. We develop a provably correct polynomial time incremental algorithm for learning DFA from labeled examples and membership queries. PAC learnability of DFA under restricted classes of distributions is an open research problem. We solve this problem by proving that DFA are efficiently PAC learnable under the class of *simple distributions.*

Constructive neural network learning algorithms offer an interesting approach for incremental construction of near minimal neural network architectures for pattern classification and inductive knowledge acquisition. The existing constructive learning algorithms were designed for two category pattern classification and assumed that the patterns have binary (or bipolar) valued attributes. We propose a framework for extending constructive learning algorithms to handle multiple output classes and real-valued attributes. Further, with carefully designed

experimental studies we attempt to characterize the *inductive bias* of these algorithms. Owing to the limited training time and the inherent representational bias, these algorithms tend to construct networks with redundant elements. We develop *pruning* strategies for elimination of redundant neurons in *MTiling* based constructive networks. Experimental results show that pruning brings about a modest to significant reduction in network size. Finally, we demonstrate the applicability of constructive learning algorithms in the area of *connectionist theory refinement*.

1 INTRODUCTION

The ability to learn is one of the central characteristics of intelligent entities. *Machine Learning* concerns the design and analysis of computational processes that learn from experience [Hon94, Lan95, RN95, Mit97]. A typical machine learning system is characterized by its ability to interact with its environment, observe the effects of its own actions, and improve its performance over time. *Inductive learning* or learning from examples is perhaps the most widely studied framework in the field of machine learning. The success of any intelligent system is based on the availability of adequate knowledge. Knowledge engineering which refers to the task of translating expert knowledge into a form that is accessible to an intelligent system is tedious and often prohibitively expensive. Inductive learning provides a framework for acquiring the necessary knowledge from examples alone and is useful in situations where the available expert knowledge is scarce or it is hard to encode the expert knowledge in the form of rules. Inductive learning systems have been successfully used in a variety of application domains including autonomously steering a vehicle on public highways [Pom89], automatically learning users' preferences and assisting them in coping with the information overload [Mae95], and discovering interesting new rules from large databases [FPSS96].

The goal of a typical inductive learning system is to construct a concise model that correctly explains the observed examples. We specifically study inductive learning systems for pattern classification tasks where the system learns to classify examples into one of M output categories (where $M \geq 2$). Formally, an example is an ordered pair $(x, c(x))$ where x is a description of an instance in a suitably chosen *instance language* and $c(x)$ is the class label assigned to the instance. For example, in pattern classification systems x is typically a vector of attribute values. A *concept* c is a function that assigns the appropriate class label to an instance x.

Inductive learning involves identifying a description $\hat{c}$ of an unknown concept c from a set of labeled examples $S = \{(x_1, c(x_1)), (x_2, c(x_2)), \ldots (x_N, c(X_N))\}$. The description $\hat{c}$ (also called the *hypothesis*) of the target concept c must ideally satisfy the following properties: $\hat{c}$ must be a good *approximation* of c in the sense that it should make very few errors (if at all) in predicting formerly unseen instances x, it must be a *concise* description of c, and it must be easily *comprehensible* in the sense that human beings can understand the rationale behind decisions made by $\hat{c}$. Often these goals are conflicting in the sense that the most concise description of the target might not necessarily be easily comprehensible and similarly, an easily understandable description of the concept might not be a very good approximation. Inductive learning systems must therefore attempt to strike a suitable balance between these potentially conflicting goals.

A crucial decision in the design of efficient inductive learning systems involves the choice of an appropriate language for describing the learned hypothesis. A restrictive choice of the *hypothesis language* severely limits the types of concepts that can be successfully learned. The set of all legally representable hypotheses in the chosen hypothesis language is called the *hypothesis class*. Thus, the hypothesis language must be expressive enough so that the hypothesis class includes at least the representations of all concepts that are of interest to the system. For example, the hypothesis class of *regular grammars* is restrictive in the sense that it cannot represent concepts describing *palindromes* (which require an advanced representation such as *context free grammars*). The choice of a suitable hypothesis class alone does not guarantee that the inductive learning system will be able to learn the target concept (or a suitable approximation of it) efficiently. The learning system must have the ability to efficiently search the space of candidate hypotheses and identify a suitable hypothesis based on the examples it is provided. Although on one hand, a highly expressive hypothesis class can obviously represent complex concept descriptions, it can also make it computationally infeasible (sometimes even impossible) to identify a suitable hypothesis from the space of candidate hypotheses. If in the above example the hypothesis class is chosen to be the set of *unrestricted grammars* then clearly the concepts describing palindromes can be suitably represented by the hypothe-

sis class. However, decision problems such as whether or not a string is in the language of an unrestricted grammar are unsolvable [HU79, Mar91]. Owing to such inherent difficulties there exists no algorithm for identifying a suitable hypothesis for the concept describing palindromes in the space of unrestricted grammars.

To make learning tractable, practical inductive learning systems have several *biases* built into them [Mit80, Mit97]. A *language bias* enables the system to focus on only one suitably chosen hypothesis description language. A strong language bias thus restricts the hypothesis class that would be considered by the system. For example, the language bias of *perceptrons* limits them to a hypothesis class of linear discriminant functions [MP69]. Since the size of the chosen hypothesis class could be very large or even infinite, a *search bias* is designed to specify how the system would search the elements of the class to determine a suitable hypothesis and which hypothesis it would prefer among a set of suitable hypotheses. For example, most inductive learning algorithms initially try to fit simple hypotheses to the training data and then progressively explore more complex ones.

1.1 Overview of the Dissertation

We present this dissertation in two parts: Part 1 describes the design and analysis of efficient algorithms for learning regular grammars and part 2 studies constructive neural network learning algorithms for pattern classification and inductive knowledge acquisition.

- *Regular grammar inference* is the process of learning rules of a target regular grammar from a set of labeled examples. The regular grammar inference problem is equivalently posed as one of learning a *deterministic finite state automaton* (DFA) corresponding to the target regular grammar. In this case, the language bias is chosen to restrict the hypothesis class to deterministic finite state automata. We attempt to design efficient algorithms that exploit appropriate search biases for learning DFA.

- *Constructive neural network learning algorithms* offer an incremental approach for the construction of near-minimal networks of *threshold logic units* (TLUs) for tasks such as pattern classification and inductive knowledge acquisition. In this case, we consider

a language bias that restricts the hypothesis class to the class of networks of TLUs. Constructive neural network learning algorithms employ a search bias of parsimonious representation and attempt to find compact networks (in terms of the number of neurons) that correctly classify all the training examples and at the same time generalize well on formerly unseen examples.

1.1.1 Inductive Learning of DFA

A *grammar* is defined as a set of rules for generating valid sentences of a particular language. *Grammar inference* is the process of learning a target grammar from a set of labeled examples [BF72, FB75, MQ86, Lan95]. It finds applications in *syntactic pattern recognition* [Fu82], *intelligent autonomous agents* [CM96], and *language acquisition* [FLSW90]. Regular grammars represent the simplest class in the Chomsky hierarchy of formal language grammars [Cho56, HU79] and describe the class of languages (*regular languages*) that can be generated (and recognized) by DFA. Since regular grammars represent a widely used subset of formal grammars, considerable research has focused on regular grammar inference (or equivalently, identification of the corresponding DFA). An understanding of the issues and pitfalls of learning regular grammars might provide insights into the problem of learning more general classes of grammars such as context free grammars.

The problem of learning the target DFA from an arbitrary set of labeled examples is known to be hard to solve [Gol78]. Efficient algorithms for learning DFA assume that some additional information is available to the learner. The learner might be provided with a representative set of labeled examples. Further, the availability of a knowledgeable teacher might facilitate learning by allowing the learner to pose queries about the target DFA. This in effect provides the learning system with additional and possibly more powerful biases that make learning tractable. We explore the learnability of DFA under the various learning biases obtained by restricting the set of labeled examples to include a *structurally complete* set (see chapter 4), a *live complete* set (see chapter 5), and a set of *simple* representative examples for the target DFA (see chapter 6). With this goal in mind, we attempt to design efficient learning algorithms that

in polynomial time output a hypothesis that is either exactly equivalent to the target or a good approximation of the target. A hypothesis is said to be equivalent to the target *iff* it makes no errors while predicting examples and counterexamples of the target. A good approximation of the target is one that guarantees an upper bound on the probability of errors made while predicting the examples and counterexamples of the target. We discuss some specific problems in the area of DFA learning and outline our solutions in chapter 2. Chapter 3 introduces some preliminary concepts. Chapter 4 presents a version space based learning algorithm for exactly learning the target DFA from a structurally complete set of examples and membership queries. Chapter 5 describes a polynomial time incremental learning algorithm for exactly learning the target DFA from labeled examples and membership queries. Chapter 6 analyzes the problem of learning DFA from simple examples and shows that DFA are approximately learnable under the *probably approximately correct* (PAC) model of learning when the class of probability distributions is restricted to the class of *simple* distributions.

1.1.2 Constructive Neural Network Learning

Artificial Neural Networks (ANN) are massively parallel systems of simple processing units that are interconnected via trainable connection weights. ANN have been successfully used in the design of pattern classification, function approximation, and knowledge acquisition systems. A variety of neural network architectures exist in the literature. These differ chiefly in terms of the choice of the mathematical functions implemented by the individual neurons (processing units), the network topology (fixed or dynamic), the network architecture (number of layers and neurons), the network interconnections (connectivity among the existing neurons), and the training methodology (one-shot or iterative) [Day90, Gal93, MMR97]. Traditional ANN algorithms such as *backpropagation* [RHW86], although successful on several pattern classification tasks, suffer from drawbacks such as restriction to an *a-priori* fixed network topology, use of the expensive gradient descent based *error backpropagation* training rule, and susceptibility to *local minima*.

Constructive (or *generative*) neural network learning algorithms offer an attractive frame-

work for automatic construction of near-minimal networks for pattern classification and inductive knowledge acquisition systems [Hon90, HU93, Gal93]. Most constructive learning algorithms are based on simple TLUs that implement a hard-limiting function of their inputs. These algorithms start out by training a single TLU using some variant of the *perceptron learning rule* [Ros58]. If the TLU is not successful in correctly classifying all the training patterns then an additional TLU (or a group of TLUs) is added and trained to correct some of the errors made by the network. Constructive learning algorithms incorporate the bias of parsimonious or compact networks (in terms of the number of neurons) in their search for an appropriate network topology for the given pattern classification task. Smaller networks are preferred to more complex networks for reasons such as: simpler digital hardware implementation, ease of extracting knowledge rules from the trained network, potential for matching the intrinsic complexity of the given classification task, and capability for superior generalization. In addition, theoretical results on learnability have shown that certain concept classes can be efficiently learned provided the hypothesis space is restricted to a set of compact representations [Nat91, KV94]. Constructive learning algorithms also provide guaranteed convergence (under certain assumptions) to zero classification errors on any finite non-contradictory data set, facility for trading off certain performance measures such as guaranteed convergence and training time versus others such as robust generalization capability, and approaches for incorporation of problem specific domain knowledge into the initial network configuration. Chapter 7 outlines some interesting issues and problems in constructive neural network learning algorithms. Chapter 8 presents provably correct extensions of several constructive neural network learning algorithms to handle multiple ($M > 2$) output classes and real-valued pattern attributes. Chapter 9 designs techniques for incorporating pruning in constructive neural network algorithms. Chapter 10 analyzes a framework for constructive theory refinement in knowledge based artificial neural networks.

Finally, we conclude in chapter 11 with a summary of the research contributions of this dissertation and highlight some interesting directions for future research.

PART I

LEARNING DETERMINISTIC FINITE AUTOMATA

2 INTRODUCTION TO REGULAR GRAMMAR INFERENCE

Regular Grammar Inference [BF72, FB75, MQ86, Lan95, PH98a] is defined as the process of learning the rules of a target regular grammar from a set of labeled examples. More formally, it is defined as follows: Given a finite non-empty set of positive examples and possibly a finite non-empty set of negative examples corresponding to an unknown regular grammar (called the *target grammar*) determine a grammar that is equivalent to the target grammar. The class of regular grammars is the simplest in the Chomsky hierarchy of formal language grammars. Their simplicity and ease of understanding makes them a widely used class of grammars for modeling several practical grammar inference tasks. Regular grammar inference has been applied in fields such as *syntactic pattern recognition*, *intelligent autonomous agents*, *language acquisition*, *computational biology*, *speech recognition*, and the like (see [GT78, Fu82, MQ86, FLSW90, CM96]). Regular grammar inference is a difficult problem to solve. It has been actively investigated for over two decades. While there do exist several practically useful heuristic solutions to the problem, we have not yet discovered an efficient general algorithm for learning the target regular grammar. On the other hand, negative results abound in the literature. Under the standard complexity theoretic assumption $P \neq NP$, it is known that no efficient algorithm exists for exactly learning a target regular grammar from an arbitrary set of labeled examples [Gol78]. Further, it has also been demonstrated that approximate learning of DFA under the PAC learning model is a hard problem [PW89, KV89]. These challenges make the regular grammar inference problem an attractive one. An understanding of the issues and pitfalls of learning regular grammars might provide insights into the problem of learning more general classes of grammars in the formal language hierarchy.

In what follows, we will attempt to briefly outline the key results to date in regular grammar

inference. We will discuss the different models of learning under which this problem has been attacked and the solutions or negative results that have resulted from these. Finally, we will provide an overview of the results of our research in this area.

2.1 Representation Classes of Regular Grammars

An important question in the regular grammar inference problem concerns the choice of the representation of the target grammar i.e., the selection of an appropriate language bias. Deterministic finite state automata (DFA), non-deterministic finite state automata (NFA), and regular expressions (REX) are equivalent representations for regular grammars. DFA are perhaps the simplest to understand and can be pictorially depicted using state transition diagrams. More importantly, decision problems such as the equivalence of two DFA, minimization of a DFA, determining if the language of a DFA is a superset/subset of the language of another, and such can be solved using efficient (i.e., polynomial time) algorithms [HU79, Mar91]. Thus, DFA is the popular representation choice for regular grammars in the area of regular grammar inference. The regular grammar inference problem is formulated equivalently as one of identifying a DFA corresponding to the target regular grammar from a given set of labeled examples.

2.2 Exact Identification of DFA

Exact learning of the target DFA from an arbitrary presentation of labeled examples is a hard problem. Gold showed that the problem of identifying the minimum state DFA consistent with a presentation S comprising of a finite non-empty set of positive examples S^+ and possibly a finite non-empty set of negative examples S^- is NP-hard [Gol78]. Under the standard *complexity theoretic* assumption $P \neq NP$, Pitt and Warmuth showed that there exists no polynomial time algorithm which when presented with a set of labeled examples corresponding to a DFA with N states is guaranteed to produce a DFA that is at most polynomially larger than the target DFA [PW88].

Efficient learning algorithms for exact identification of DFA assume that some additional

information is provided to the learner. Trakhtenbrot and Barzdin described a polynomial time algorithm for constructing the smallest DFA consistent with a *complete labeled sample* i.e., a sample that includes all strings up to a particular length and the corresponding label that states whether the string is accepted by the target DFA or not [TB73]. Thus, their algorithm computes, in polynomial time, the smallest DFA that correctly accepts all the positive examples and correctly rejects all the negative examples of the complete labeled sample. However, Angluin showed that even if a vanishingly small fraction of the strings from the complete labeled sample is missing then the problem of finding the smallest consistent DFA is NP-hard [Ang78]. Oncina and García recently proposed the *regular positive and negative inference* (RPNI) algorithm that in polynomial time identifies a DFA consistent with a given sample S [OG92]. Further, if S is a superset of a *characteristic set* for the target DFA (see section 3.3.7) then the DFA output by the RPNI algorithm is guaranteed to be equivalent to the target [OG92, Dup96b].

A set of labeled examples that satisfy certain properties is one form of additional information that makes the DFA learning problem tractable. Additionally, one may assume the existence of a knowledgeable teacher who responds to queries posed by the learner. Pao and Carr proposed a framework for learning the target DFA from a *structurally complete* set of positive examples that in essence describes all the transitions and the accepting states of the target DFA (see section 3.3.5) [PC78]. Additionally, their algorithm assumes the availability of a knowledgeable teacher capable of answering *membership queries*. Their algorithm maps the structurally complete set of examples to an ordered lattice of *finite state automata* (FSA)[1]. This lattice is guaranteed to contain the target DFA. The algorithm searches for the target DFA with the help of membership queries. A membership query is posed to ask the teacher whether an example string belongs to the language of the target DFA or not. Under this framework, the target DFA is shown to be exactly identifiable [PC78].

Angluin showed that given a *live complete* set of examples that contains a representative string for each live state of the target DFA (see section 3.3.6) and a knowledgeable teacher

[1]Note that a FSA is either a DFA or a NFA

to answer *membership queries* it is possible to exactly learn the target DFA [Ang81]. Later, Angluin refined this idea to design an algorithm L^* that infers the target DFA with the help of a *minimally adequate teacher* [Ang87]. A minimally adequate teacher (MAT) is one who is knowledgeable about the target concept and is able to answer the learner's queries. The L^* algorithm allows the learner to pose two types of queries viz. *membership* and *equivalence* queries. Unlike the above approaches, the L^* algorithm does not search the lattice of FSA. Instead it constructs a hypothesis DFA by posing membership queries to the teacher. Once an appropriate hypothesis is constructed, the learner poses an equivalence query to the teacher to inquire whether the current hypothesis is equivalent to the target DFA or not. If the hypothesis is indeed equivalent to the target the algorithm outputs the hypothesis and halts. Otherwise, the teacher provides a counterexample. The learner modifies the hypothesis using the counterexample and additional membership queries and poses another equivalence query. This interaction between the learner and the teacher continues until the teacher's answer to an equivalence query is *yes*. This algorithm runs in polynomial time and is guaranteed to converge to the target.

The L^* algorithm tacitly assumes that the learner has the capacity to *reset* the DFA to the start state before posing each membership query. This assumption might not be realistic in certain situations. For example, consider a robot trying to explore its environment. This environment may be modeled as a finite state automaton with the different states corresponding to the different situations the robot might find itself in and the transitions corresponding to the different actions taken by the robot in each situation. Once the robot has made a sequence of moves it might find itself in a particular state (say facing an obstacle). However, the robot has no way of knowing where it started from or of retracing its steps to the start state. The robot has to continue from its current state and explore the environment further. Rivest and Schapire proposed a learning method based on *homing sequences* [RS93]. Assuming that each state of the DFA has an output (the output could simply be 1 for accepting state and 0 for non-accepting state), a homing sequence is defined as a sentence (string) whose output always uniquely identifies the final state the DFA is in even if we do not know where the DFA started

from. Rivest and Schapire's algorithm runs N copies of the L^* algorithm in parallel (one copy for each of the N states of the target DFA) to overcome the limitation that the start state is unknown.

2.3 Incremental Learning of DFA

All the DFA learning algorithms discussed thus far require that the labeled training examples be available to the learner in advance. In many practical learning scenarios, the entire training set might not be available at the start. Instead, a sequence of examples is provided intermittently and the learner is required to construct an approximation of the target DFA based on the examples it has seen until then. In such scenarios, an online or incremental model of learning that is guaranteed to eventually converge to the target DFA in the limit is of interest. In the online learning framework the learner constructs a consistent hypothesis based on the initially provided set of examples. When additional examples become available, the learner must incrementally modify the current hypothesis to make it consistent with the new examples without having to re-start from scratch. Ideally, an online framework should be designed such that the learner does not have to store the examples it sees during learning. The current hypothesis along with the next labeled example should be sufficient to guarantee that the modified hypothesis is consistent with the new example as well as all the examples seen previously.

Dupont proposed an incremental version of the *RPNI* algorithm for regular grammar inference [Dup96a]. This algorithm is also based on the idea of a lattice of finite state automata constructed from a set of positive examples. It uses information from a set of negative examples to guide the ordered search through the lattice and is guaranteed to converge to the target DFA when the set of examples seen by the learner include a *characteristic set* (see section 3.3.7) for the target DFA as a subset. The algorithm runs in time that is polynomial in the sum of lengths of the training examples. However, it requires storage of all the examples that are seen by the learner during training to ensure that each time the representation of the target is modified, it stays consistent with all the previous examples.

Porat and Feldman have proposed an incremental algorithm for inference of regular grammars from a *complete ordered sample* (one that includes all the strings over the alphabet of the DFA up to a certain length) [PF91]. The algorithm maintains a current hypothesis that is consistent with all the examples seen thus far. If this hypothesis is inconsistent with the next labeled example then it is modified appropriately to ensure consistency with the new example and also with all the previous examples. It is guaranteed to converge in the limit provided the examples appear in strict lexicographic order. Further, it works with only a finite working storage which is an advantage over the incremental extension of the *RPNI* algorithm. However, it requires strict lexicographic order of presentation of examples which may not be feasible in some practical learning situations. Also, it requires a consistency check with all the previous examples each time the current representation of the target is modified.

2.4 Approximate Identification of DFA

Valiant's distribution independent model of learning (also called the PAC model) [Val84] is widely used for approximate learning of concept classes. When adapted to the problem of learning DFA, the goal of a PAC learning algorithm is to obtain in polynomial time, with high probability, a DFA that is a good approximation of the target DFA. Even approximate learnability of DFA was proven to be a hard problem. Pitt and Warmuth showed that the problem of polynomially approximate predictability of the class of DFA is hard [PW89]. Using *prediction preserving reductions* they showed that if DFA are polynomially approximately predictable then so are other known hard to predict concept classes such as *boolean formulas*. Kearns and Valiant showed that an efficient algorithm for learning DFA would entail efficient algorithms for solving problems such as breaking the *RSA* cryptosystem, factoring *Blum* integers, and detecting *quadratic residues* [KV89]. Under cryptographic assumptions it is known that these problems are known to be hard to solve. Thus, they showed that DFA learning is a hard problem.

The PAC model's requirement of learnability under all conceivable distributions is often considered too stringent for practical learning scenarios. Several researchers have explored the

PAC learnability of concepts under some known distributions such as the *uniform distribution* or under some restricted families of distributions such as *product distributions*. It has been shown that several concept classes are efficiently PAC learnable under restricted classes of distributions while their learnability under the distribution free model is not known. Pitt's seminal paper surveyed several approaches to approximate learning of DFA and identified the following open research problem: "*Are DFA PAC-identifiable if examples are drawn from the uniform distribution or some other known simple distribution?* " [Pit89]. Using a variant of Trakhtenbrot and Barzdin's algorithm, Lang empirically demonstrated that random DFA are approximately learnable from a sparse uniform sample [Lan92]. However, no theoretical results on PAC learnability of DFA were derived and exact identification of the target DFA was not possible even in the average case with a randomly drawn training sample.

2.5 Overview of Research Results

As is evident from the above discussion, the problem of learning DFA from labeled examples is computationally hard. The problem is made tractable when the learner is provided with some sort of a representative sample and is perhaps allowed access to a teacher who answers queries. We were inspired by the challenge posed by the DFA learning problem and have attempted to address some of the difficulties and the open research problems outlined by researchers in the field.

2.5.1 Exactly Learning DFA using a Version Space based Approach

Pao and Carr's algorithm maps the structurally complete set of examples to an ordered lattice of finite state automata which constitutes the hypothesis space. The lattice is guaranteed to contain a representation of the target DFA and the goal of the learning algorithm is to search the lattice for the target DFA using membership queries. Their algorithm explicitly enumerates the entire lattice. The size of the lattice is prohibitively large even when the structurally complete set contains only a few short strings. Thus, explicit enumeration of the hypothesis space is not practical. We propose the use of a version space for compactly representing

the hypothesis space [PH93, PH96]. The version space implicitly represents the entire lattice using two sets of DFA called $\mathcal{S}$ and $\mathcal{G}$ respectively. $\mathcal{S}$ is initialized to the most *special* DFA called the *prefix tree automaton* obtained from the structurally complete set of examples. $\mathcal{G}$ is initialized to the most *general* DFA that accepts all strings over a pre-specified alphabet. These two sets together capture the entire lattice implicitly. An efficient bidirectional search strategy is employed to locate the target DFA in the lattice. Query strings are generated by comparing two DFA (one from each of $\mathcal{S}$ and $\mathcal{G}$) for equivalence. The teacher's response to the membership queries is used to prune the hypothesis space. Elements of the set $\mathcal{S}$ that do not accept positive examples are progressively generalized by state merging. Similarly, elements of the set $\mathcal{G}$ that accept negative examples are progressively specialized by state splitting. The set $\mathcal{S}$ ($\mathcal{G}$) becomes progressively more general (special) and the algorithm eventually converges when $\mathcal{S}$ and $\mathcal{G}$ are exactly the same and contain equivalent DFA. We discuss this algorithm and give its correctness proof in chapter 4.

2.5.2 Polynomial Time Incremental Learning of DFA

In chapter 5 we study an approach for online learning of DFA using labeled examples and membership queries. The new algorithm *IID* (incremental *ID*) extends Angluin's *ID* algorithm to an incremental framework. The learning algorithm is intermittently provided with labeled examples and has access to a knowledgeable teacher capable of answering membership queries. Based on the observed examples and the teacher's responses to membership queries, the learner constructs a hypothesis DFA. This DFA is guaranteed to be consistent with all observed examples. When an additional example is provided, the learner determines if the new example is consistent with the current hypothesis in which case no further action is required. If however, the new example is not consistent with the current hypothesis then the learner incrementally modifies the hypothesis suitably to encompass the information provided by the new example. In the limit this algorithm is guaranteed to converge to a minimum state DFA corresponding to the target DFA. We describe this algorithm, prove its convergence, and analyze its time and space complexities in chapter 5.

2.5.3 Learning DFA from Simple Examples

In chapter 6 we address the issue of PAC learning of DFA. PAC learning of DFA is known to be a hard problem [PW89, KV89]. An interesting open research question (due to [Pit89]) is whether DFA can be learned approximately under restricted classes of distributions. Li and Vitányi proposed a model for PAC learning with *simple* examples wherein the examples are drawn according to the *Solomonoff-Levin universal distribution* (universal distribution). This model is referred to as the simple PAC learning model. They showed that learnability under the universal distribution implies learnability under a broad class distributions known as *simple distributions* provided the examples are drawn according to the universal distribution. Thus, this model is quite general. Recently, this learning model was extended to a framework where a teacher might intelligently choose examples based on the knowledge of the target concept [DDG96]. This is called the PAC learning with simple examples (PACS learning) model.

We answer the above open research question in the affirmative by proving that DFA are efficiently learnable from simple examples. In particular, by using the RPNI algorithm for learning DFA and the universal distribution $\mathbf{m}$ for drawing a labeled sample at random we show that the class of *simple DFA* (see section 6.4) is learnable under the simple PAC learning model. Further, we demonstrate that it is possible to efficiently learn the entire class of DFA under the PACS learning model [PH97]. Finally, we show the applicability of the PACS learning model in a more general setting by proving that all concept classes that are polynomially identifiable from characteristic samples according to Gold's model and semi-polynomially T/L teachable according to Goldman and Mathias' model are also probably exactly learnable under the PACS model.

3 PRELIMINARIES

In this chapter we introduce the basic definitions and notation used throughout part 1 of the thesis. Readers who are familiar with the concepts of finite state automata may choose to go over to the next chapter and refer to this chapter whenever some notation is unclear.

3.1 Strings and Sets of Strings

Let Σ be a finite set of symbols called the *alphabet*. A concatenation of symbols from Σ represents a string α. Σ^* denotes the set of all possible strings over Σ. Let α, β, γ be strings in Σ^* and $|\alpha|$ be the length of the string α. λ is a special string called the *null* string and has length 0. A *language* L is a subset of Σ^*. Given a string $\alpha = \beta\gamma$, $\beta \in \Sigma^*$ is the *prefix* of α and $\gamma \in \Sigma^*$ is the *suffix* of α. Let $Pref(\alpha)$ denote the set of all prefixes of α. The set $Pref(L) = \{\alpha \mid \alpha\beta \in L\}$ is the set of *prefixes* of the language L. The set $L_\alpha = \{\beta \mid \alpha\beta \in L\}$ is the set of *tails* of α in L. The *standard order* of strings over the alphabet Σ is denoted by $<$. The enumeration of strings over $\Sigma = \{a, b\}$ in standard order is $\lambda, a, b, aa, ab, ba, bb, aaa, \ldots$ The set of *short prefixes* $S_p(L)$ of a language L is defined as $S_p(L) = \{\alpha \in Pref(L) \mid \nexists \beta \in \Sigma^*$ such that $L_\alpha = L_\beta$ and $\beta < \alpha\}$. The *kernel* $N(L)$ of a language L is defined as $N(L) = \{\lambda\} \cup \{\alpha a \mid \alpha \in S_p(L), a \in \Sigma, \alpha a \in Pref(L)\}$. Let $S_1 \backslash S_2$ and $S_1 \oplus S_2$ denote the *set difference* and the *symmetric difference* respectively of the sets S_1 and S_2.

3.2 Formal Language Grammars

A *formal language grammar* G is a 4-tuple $G = (V_N, V_T, P, S)$ where V_N is the set of non-terminals, V_T is the set of terminals, P is the set of production rules for generating valid

sentences of the language, and $S \in V_N$ is a special symbol called the start symbol. The production rules are of the form $\alpha \longrightarrow \beta$ where α, β are strings belonging to $(V_N \cup V_T)^*$ and α contains at least one non-terminal symbol. Valid strings of the language are sequences of terminal symbols V_T and are obtained by repeatedly applying the production rules as shown below. The language $L(G)$ is the set of all strings generated by the grammar.

Different classes of formal grammars are obtained by placing different restrictions on the types of production rules. A *Regular Grammar* (G) is a finite set of rewrite (production) rules of the form $A \longrightarrow aB$ or $A \longrightarrow b$ where A and B are *non-terminals* and a and b are *terminals*. Consider the regular grammar G where $V_N = \{S, A, B\}$, $V_T = \{a, b\}$, and $P = \{S \longrightarrow b, S \longrightarrow aA, A \longrightarrow aB, A \longrightarrow a, B \longrightarrow aA\}$. Example strings generated by this grammar include b (obtained by applying the rule 1), aa (obtained by applying the rules 2 and 4 in that order), $aaaa$ (obtained by applying the rules 2, 3, 5, and 4 in that order), and so on.

3.3 Deterministic Finite State Automata

Finite State Automata (FSA) are recognizing devices for regular grammars. A *deterministic* finite state automaton (DFA) is a quintuple $A = (Q, \delta, \Sigma, q_0, F)$ where, Q is a finite set of states, Σ is the finite alphabet, $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of accepting states, and δ is the transition function: $Q \times \Sigma \longrightarrow Q$. A state $d_0 \in Q$ such that $\forall a \in \Sigma$, $\delta(d_0, a) = d_0$ is called a *dead* state. If there exists a state $q \in Q$ such that $\delta(q, a)$ is not defined for some $a \in \Sigma$ then the transition function is said to be incompletely specified. It may be fully specified by adding transitions of the form $\delta(q, a) = d_0$ when $\delta(q, a)$ is undefined. The extension of δ to handle input strings is denoted by δ^* and is defined as follows: $\delta^*(q, \lambda) = q$ and $\delta^*(q, a\alpha) = \delta^*(\delta(q, a), \alpha)$ for $q \in Q$, $a \in \Sigma$ and $\alpha \in \Sigma^*$. The set of all strings accepted by A is its language, $L(A)$. $L(A) = \{\alpha | \delta^*(q_0, \alpha) \in F\}$. The language accepted by a DFA is called a *regular language*.

A *non-deterministic* finite automaton (NFA) is defined just like the DFA except that the transition function δ defines a mapping from $Q \times \Sigma \longrightarrow 2^Q$. NFA and DFA are equivalent in their representation power in that for any NFA A' there exists a DFA A such that $L(A') = L(A)$. In general, a *finite state automaton* (FSA) refers to either a DFA or a NFA.

FSA are represented using state transition diagrams. The start state q_0 is indicated by the symbol $>$ attached to it. Accepting states are denoted using concentric circles. The state transition $\delta(q_i, a) = q_j$ for any letter $a \in \Sigma$ is depicted by an arrow labeled by the letter a from the state q_i to the state q_j. Fig. 3.1 shows the state transition diagram for a sample DFA. This DFA corresponds to the regular grammar G described in section 3.2 in that $L(A) = L(G)$.

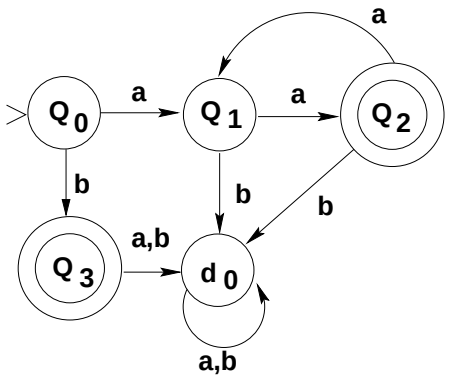

Figure 3.1 Deterministic Finite State Automaton.

A labeled example $(\alpha, c(\alpha))$ for A is such that $\alpha \in \Sigma^*$ and $c(\alpha) = +$ if $\alpha \in L(A)$ (i.e., α is a positive example) or $c(\alpha) = -$ if $\alpha \notin L(A)$ (i.e., α is a negative example). Thus, $(a, -)$, $(b, +)$, $(aa, +)$, $(aaab, -)$, and $(aaaa, +)$ are labeled examples for the DFA of Fig. 3.1. S^+ will be used to denote a set of positive examples of A i.e., $S^+ \subseteq L(A)$. Similarly, S^- will be used to denote a set of negative examples of A i.e., $S^- \subseteq \Sigma^* \backslash L(A)$. A sample S will be defined as $S = S^+ \cup S^-$. A is consistent with a *sample* S if it accepts all the positive examples (i.e., all examples in S^+) and rejects all negative examples (i.e., all examples in S^-).

3.3.1 Canonical DFA

Given any FSA A', there exists a minimum state DFA (also called the *canonical DFA*) A, such that $L(A) = L(A')$. Without loss of generality, we will assume that the target DFA being learned is a canonical DFA. Let N denote the number of states of A. It can be shown that any canonical DFA has at most one dead state. One can define a standard encoding of DFA as binary strings such that any DFA with N states is encoded as a binary string of length $O(N \lg N)$.

3.3.2 Prefix Tree Automaton

Given a set S^+ of positive examples, let $PTA(S^+)$ denote the *prefix tree acceptor* for S^+. $PTA(S^+)$ is a DFA that contains a path from the start state to an accepting state for each string in S^+ modulo common prefixes. Clearly, $L(PTA(S^+)) = S^+$. Learning algorithms such as the RPNI (see section 6.3) require the states of the PTA to be numbered in standard order. If we consider the set $Pref(S^+)$ of prefixes of the set S^+ then each state of the PTA corresponds to a unique element in the set $Pref(S^+)$ i.e., for each state q_i of the PTA there exists exactly one string α_i in the set $Pref(S^+)$ such that $\delta^*(q_0, \alpha_i) = q_i$ and vice-versa. The strings of $Pref(S^+)$ are sorted in standard order $<$ and each state q_i is numbered by the position of its corresponding string α_i in the sorted list. The PTA for the set $S^+ = \{b, aa, aaaa\}$ is shown in Fig. 3.2. Note that its states are numbered in standard order.

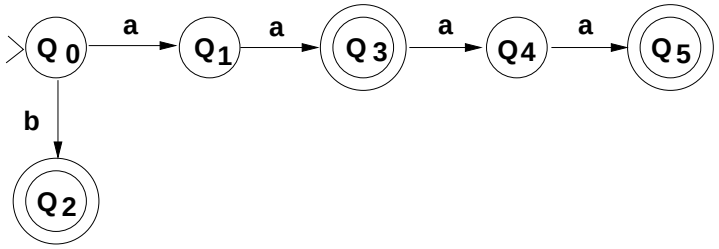

Figure 3.2 Prefix Tree Automaton.

3.3.3 Quotient FSA

Given a FSA A, consider a partition π on the set of states Q of A. Assume that the dead state d_0 and its associated transitions are ignored. Define $\pi = \{B_1, B_2, \ldots, B_k\}$ where $k \leq N$ and for $1 \leq i \leq k$ $B_i \subseteq Q$. Further, $\bigcup_{i=1}^{k} B_i = Q$. The block of the partition π to which a state $q \in Q$ belongs is denoted by $B(q, \pi)$. We define the *quotient automaton* (or equivalently *derived automaton*) $A_\pi = (Q_\pi, \delta_\pi, \Sigma, B(q_0, \pi), F_\pi)$ obtained by merging the states of A that belong to the same block of the partition π as follows: $Q_\pi = \{B(q, \pi) \mid q \in Q\}$ is the set of states. Essentially, each block of the partition π corresponds to a state in Q_π. $F_\pi = \{B(q, \pi) \mid q \in F\}$ is the set of accepting states. $\delta_\pi : Q_\pi \times \Sigma \longrightarrow 2^{Q_\pi}$ is the transition function such that $\forall B(q_i, \pi), B(q_j, \pi) \in Q_\pi, \forall a \in \Sigma$, $B(q_j, \pi) = \delta_\pi(B(q_i, \pi), a)$ *iff* $q_i, q_j \in Q$ and

$q_j = \delta(q_i, a)$. Note that a quotient automaton of a DFA might be a NFA and vice-versa. For example, the quotient automaton corresponding to the partition $\pi = \{\{Q_0, Q_1\}, \{Q_2\}, \{Q_3\}\}$ of the set of states of the DFA in Fig. 3.1 is shown in Fig. 3.3.

The set of all quotient automata obtained by systematically merging the states of A represents a *lattice* of FSA [PC78]. This lattice is ordered by the *grammar cover* relation $\preceq$. Given two partitions $\pi_i = \{B_{i_1}, B_{i_2}, \ldots, B_{i_r}\}$ and $\pi_j = \{B_{j_1}, B_{j_2}, \ldots, B_{j_k}\}$ of the states of A, we say that π_i covers π_j (written $\pi_j \preceq \pi_i$) if $|\pi_i| = |\pi_j| - 1$ and for some $j_1 \leq j_l, j_m \leq j_k$, $\pi_i = \pi_j \backslash \{B_{j_l}, B_{j_m}\} \cup \{B_{j_l} \cup B_{j_m}\}$. The *transitive closure* of $\preceq$ is denoted by $\ll$. By convention we will represent the quotient FSA corresponding to a partition π_i by M_i. We say that $\pi_j \ll \pi_i$ *iff* $L(M_j) \subseteq L(M_i)$. It is easy to see that the language of the quotient automaton in Fig. 3.3 is a superset of the language of the DFA in Fig. 3.1. Further, given a canonical DFA A and a set S^+ that is structurally complete with respect to A (see section 3.3.5), the lattice $\Omega(S^+)$ derived from $PTA(S^+)$ is guaranteed to contain A [PC78, PH93, DMV94].

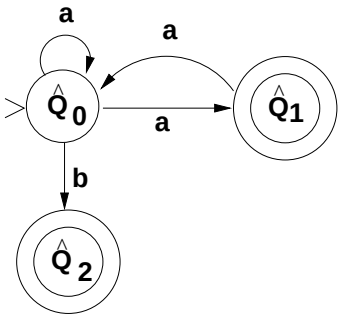

Figure 3.3 Quotient Automaton.

3.3.4 Sub-automaton

A *sub-automaton* A_x for a DFA A (ignoring its dead state d_0 and its associated transitions) is a quintuple $A_x = (Q_x, \delta_x, \Sigma, q_0, F_x)$ where $Q_x \subseteq Q$, $q_0 \in Q_x$, $F_x \subseteq F$, and δ_x is defined as follows: $\forall q_i, q_j \in Q_x$ and $a \in \Sigma$, $\delta_x(q_i, a) = q_j \Rightarrow \delta(q_i, a) = q_j$. An example sub-automaton of the DFA in Fig. 3.1 is shown in Fig. 3.4.

The interested reader is referred to [HU79, LP81, Mar91] for a detailed description of the theory of finite state automata, regular grammars, and languages.

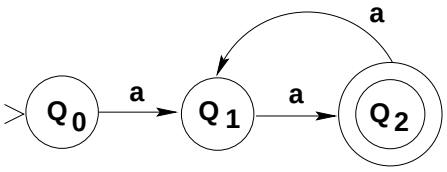

Figure 3.4 Sub-Automaton.

3.3.5 Structurally Complete Sample

Given a regular grammar G, a structurally complete set of examples S^+ is one that covers each production rule of G at least once. In other words, S^+ is a structurally complete set of examples with respect to G if each production rule of G is used at least once in generating the strings of S^+. Equivalently, if A is a canonical DFA corresponding to the regular grammar G then the set S^+ is said to be *structurally complete* with respect to A if S^+ covers each transition of A (except the transitions associated with the dead state d_0) and uses every element of the set of final states of A as an accepting state [PC78, PH93, DMV94]. More formally, S^+ is structurally complete with respect to A if it satisfies the following two properties:

1. $\forall a \in \Sigma,\ q_i, q_j \in Q - \{d_0\},$ if $\delta(q_i, a) = q_j$ is a transition of A then $\exists \alpha \in S^+$ where $\alpha = \beta a \gamma$ $\beta, \gamma \in \Sigma^*$ such that $\delta^*(q_0, \beta) = q_i$ and $\delta^*(q_j, \gamma) \in F$

2. $\forall q_i \in F,\ \exists \alpha \in S^+$ such that $\delta^*(q_0, \alpha) = q_i$

It can be verified that the set $S^+ = \{b, aa, aaaa\}$ is structurally complete with respect to the DFA in Fig. 3.1. Note that in general the structurally complete set is not unique for a given DFA. Further, any set of positive examples of the DFA A that includes a structurally complete set of examples with respect to A as a subset is also structurally complete.

3.3.6 Live Complete Sample

A state q_i of a DFA A is *live* if there exist strings α and β such that $\alpha\beta \in L(A)$, $\delta^*(q_0, \alpha) = q_i$, and $\delta^*(q_i, \beta) \in F$. A state that is not live is called *dead*. As stated earlier, a canonical DFA can have at most one dead state and we use d_0 to denote this dead state. Given A, a finite set of strings P is said to be *live complete* if for every live state q_i of A there exists a string $\alpha \in P$

such that $\delta^*(q_0, \alpha) = q_i$ [Ang81]. For example, $P = \{\lambda, a, b, aa\}$ is a live complete set for the DFA in Fig. 3.1. Any superset of a live complete set is also live complete. In order to have a representation for the start state of A we assume that the string λ is part of any live complete set. The set $P' = P \cup \{d_0\}$ represents all the states of A. To account for the state transitions, define a function $f : P' \times \Sigma \longrightarrow \Sigma^* \cup \{d_0\}$ as follows:

$$\begin{aligned} f(d_0, a) &= d_0 \\ f(\alpha, a) &= \alpha a \end{aligned}$$

Note that $f(\alpha, a)$ denotes the state reached upon reading an input letter $a \in \Sigma$ from the state represented by the string $\alpha \in \Sigma^*$. We will let $T' = P' \cup \{f(\alpha, a) | (\alpha, a) \in P' \times \Sigma\}$ and $T = T' \backslash \{d_0\}$. Thus, given the live complete set $P = \{\lambda, a, b, aa\}$ corresponding to the DFA in Fig. 3.1 we obtain the set $T = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab\}$.

3.3.7 Characteristic Sample

Consider a regular grammar G with the corresponding canonical acceptor A. Let L denote the language of G (and equivalently the language of A). A sample $S = S^+ \cup S^-$ is said to be *characteristic* with respect to a regular language L if it satisfies the following two conditions [OG92]:

1. $\forall \alpha \in N(L)$, if $\alpha \in L$ then $\alpha \in S^+$ else $\exists \beta \in \Sigma^*$ such that $\alpha\beta \in S^+$
2. $\forall \alpha \in S_p(L), \forall \beta \in N(L)$, if $L_\alpha \neq L_\beta$ then $\exists \gamma \in \Sigma^*$ such that ($\alpha\gamma \in S^+$ and $\beta\gamma \in S^-$) or ($\beta\gamma \in S^+$ and $\alpha\gamma \in S^-$)

Intuitively, $S_p(L)$, the set of short prefixes of L is a live complete set with respect to A in that for each live state $q \in Q$, there is a string $\alpha \in S_p(L)$ such that $\delta^*(q_0, \alpha) = q$. The kernel $N(L)$ includes the set of short prefixes as a subset. Thus, $N(L)$ is also a live complete set with respect to A. Further, $N(L)$ covers every transition between each pair of live states of A. i.e., for all live states $q_i, q_j \in Q$, for all $a \in \Sigma$, if $\delta(q_i, a) = q_j$ then there exists a string $\beta \in N(L)$ such that $\beta = \alpha a$ and $\delta^*(q_0, \alpha) = q_i$. Thus, condition 1 above which identifies

a suitably defined suffix $\beta \in \Sigma^*$ for each string $\alpha \in N(L)$ such that the augmented string $\alpha\beta \in L$, implies structural completeness with respect to A. Condition 2 implies that for any two distinct states of A there is a suffix γ that would correctly distinguish them. In other words, for any $q_i, q_j \in Q$ where $q_i \not\equiv q_j$, $\exists\gamma \in \Sigma^*$ such that $\delta^*(q_i, \gamma) \in F$ and $\delta^*(q_j, \gamma) \notin F$ or vice-versa. Given the language L corresponding to the DFA A in Fig. 3.1, the set of short prefixes is $S_p(L) = \{\lambda, a, b, aa\}$ and the kernel is $N(L) = \{\lambda, a, b, aa, aaa\}$. It can be easily verified that the set $S = S^+ \cup S^-$ where $S^+ = \{b, aa, aaaa\}$ and $S^- = \{\lambda, a, aaa, baa\}$ is a characteristic sample for L.

4 A VERSION SPACE BASED APPROACH TO LEARNING DFA

4.1 Introduction

In this chapter we describe a version space approach to learning the target DFA. We will assume that the learner is provided with a structurally complete set of examples and is allowed access to a knowledgeable teacher who answers membership queries. The problem of learning the target DFA under this framework was originally studied by Pao and Carr [PC78]. Their algorithm maps the structurally complete set of examples to a lattice of finite state automata. The lattice represents the entire search space and is guaranteed to contain a representation of the target DFA[1]. The learner uses membership queries to search the lattice for the target. Though provably correct their approach has the following limitations:

1. The entire lattice is enumerated explicitly. The size of the lattice grows exponentially in the sum of the lengths of the strings provided in the structurally complete set (see section 4.2). Even for small structurally complete sets it is practically infeasible to explicitly enumerate each element of the lattice.

2. Membership queries are generated by comparing two finite state automata (from the lattice) for equivalence. Some of the elements in the lattice represent NFA. Pao and Carr's algorithm requires that NFA be converted to equivalent DFA and then used for generating the query string. However, the algorithm for converting NFA to DFA has exponential time complexity in the worst case [HU79].

[1]Note that Pao and Carr define a structurally complete set as one that covers all the transitions of the target DFA. However, the correct definition of structural completeness states that each production rule of the target grammar must be covered by at least one string in the structurally complete set. This requires that each transition and each accepting state of the target DFA must be covered by the strings in the structurally complete set [PH93, DMV94, PH96].

We present an improved learning algorithm based on the version space representation of the lattice of FSA [PH93, PH96]. The version space implicitly represents the entire lattice using two sets of FSA called $\mathcal{S}$ and $\mathcal{G}$ respectively. The operations on the version space take time polynomial in the size of the $\mathcal{S}$ and $\mathcal{G}$ sets. The efficiency of the algorithm thus relies on the fact that the size of these sets at any time is substantially smaller than the size of the entire lattice. The proposed algorithm uses an efficient bidirectional search strategy inspired by Mitchell's version space algorithm [Mit82]. Further, we formulate the search procedure such that the problem of converting NFA to DFA is totally avoided. Thus, our approach overcomes both the limitations encountered in Pao and Carr's algorithm.

The rest of this chapter is organized as follows: Section 4.2 explains the mapping of the given structurally complete set to a lattice of FSA. Section 4.3 outlines the implicit representation of this lattice in the form of a version space. Section 4.4 describes the query aided bidirectional search of the lattice. Section 4.5 proves the correctness of this algorithm. Finally, section 4.6 concludes with a discussion of the algorithm's merits and demerits.

4.2 Lattice of Finite State Automata Specified by S^+

Given a structurally complete set of examples S^+, the learner constructs a prefix tree automaton (PTA) that accepts exactly the strings in S^+. For example, consider that the DFA in Fig. 4.1 is the target DFA. It is easy to verify that the set $S^+ = \{abb\}$ is structurally complete with respect to the target DFA. Fig. 4.2 shows the corresponding prefix tree automaton $PTA(S^+)$.

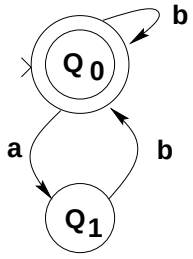

Figure 4.1 Target DFA.

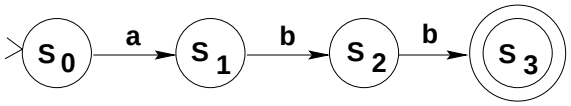

Figure 4.2 PTA Corresponding to $S^+ = \{abb\}$.

The set of all partitions of the set of states of $PTA(S^+)$ forms a lattice Ω. Fig. 4.3 shows the lattice Ω constructed from the PTA in Fig. 4.2 [2]. Each partition corresponds to a quotient automaton (as described in section 3.3.3) of $PTA(S^+)$. By definition, the language of each quotient automaton is a superset of $L(PTA(S^+))$.

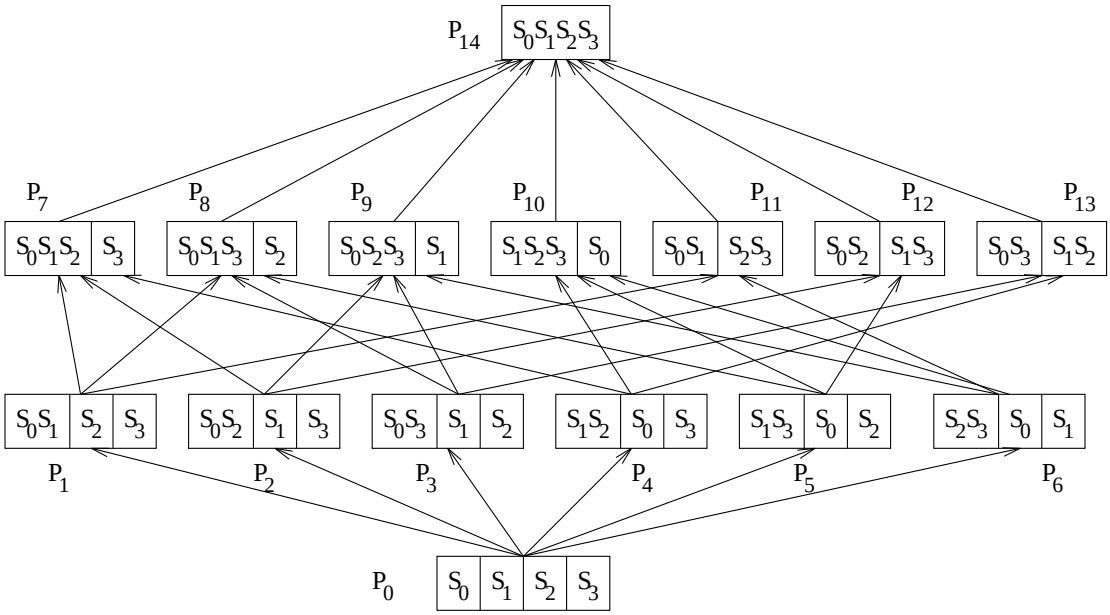

Figure 4.3 Lattice Ω.

The lattice is partially ordered by the grammar covers relation (see section 3.3.3). By the grammar covers property we know that if a partition π_i covers a partition π_j then the corresponding FSA M_i and M_j are such that $L(M_j) \subseteq L(M_j)$. The fact that a partition π_i covers a partition π_j (i.e., $\pi_j \preceq \pi_i$) is depicted in Fig. 4.3 by an arrow from π_j to π_i. If $\pi_j \preceq \pi_i$ then π_j is said to be an immediate *specialization* of π_i and correspondingly π_i is said to be an immediate *generalization* of π_j. The *minimal generalization* of a partition π_j is defined

[2] Note that the individual partitions of the set of states of $PTA(S^+)$ are denoted as $\mathrm{P}_0, \mathrm{P}_1, \ldots$ in Fig. 4.3 and are referred to as $\pi_0, \pi_1, \ldots$ in the text.

as the set of all immediate generalizations of π_j i.e., $\{\pi_k | \pi_j \preceq \pi_k\}$. Similarly, the *minimal specialization* of a partition π_i is defined as the set of all immediate specializations of π_i i.e., $\{\pi_k | \pi_k \preceq \pi_i\}$. In general, if $\pi_j \ll \pi_i$ then π_j is said to be *more special than or equal to* (MSE) π_i and correspondingly π_i is said to be *more general than or equal to* (MGE) π_j. The MSE (MGE) test can be performed efficiently by simply examining the two partitions π_i and π_j. If π_i and π_j are two partitions with corresponding FSA M_i and M_j then by virtue of the grammar covers relation $\pi_j \ll \pi_i \; iff \; L(M_j) \subseteq L(M_i)$.

The grammar covers property is used to prune the search space by eliminating candidate FSA that do not correspond to the target. Suppose it is determined that the FSA M_j corresponding to the partition π_j accepts a negative example. Clearly, M_j cannot be the target. Further, all FSA M_k where $\pi_j \ll \pi_k$ will also accept the same negative example and hence can be eliminated from consideration. Similarly, if it is determined that the FSA M_i corresponding to a partition π_i does not accept a positive example then clearly M_i cannot be the target. Further, all FSA M_k where $\pi_k \ll \pi_i$ will also fail to accept the same positive example and hence can be eliminated from consideration.

4.3 Version Space Representation of the Lattice Ω

The total number of partitions contained in a lattice (Ω) obtained from a PTA with m states is $E_m = \sum_{j=0}^{m-1} \binom{m-1}{j} E_j$ where $E_0 = 1$. Clearly, explicit enumeration of Ω is not feasible even for moderately large values of m. We propose an implicit representation of Ω using a version space $\Theta = [\mathcal{S}, \mathcal{G}]$ where $\mathcal{S}$ represents the most *special* partitions and $\mathcal{G}$ represents the most *general* partitions of Ω that are consistent with the data gathered by the learner at any time.

$\mathcal{S}$ is initialized to $\{\pi_0\}$ where π_0 is the most special element of Ω (i.e., π_0 is the partition representing the $PTA(S^+)$. $\mathcal{G}$ is initialized to $\{\pi_{E_{m-1}}\}$ where $\pi_{E_{m-1}}$ is the most general element of Ω (i.e., $\pi_{E_{m-1}}$ is the partition representing the DFA obtained by merging all the states of $PTA(S^+)$ into a single block). Note that E_m is the total number of partitions in the lattice Ω. Since the entire lattice can be enumerated by listing the set of quotient FSA of $PTA(S^+)$ (i.e., by systematically merging the states of $PTA(S^+)$) it is clear that Θ implicitly represents the

entire lattice Ω. As the search progresses, the elements of $\mathcal{S}$ are made progressively more general and those of $\mathcal{G}$ are made progressively more special. Fig. 4.4 depicts a typical bidirectional search of the lattice. In order to guarantee convergence to the target DFA the version space must satisfy the following properties at all times [Mit82]:

1. The elements of $\mathcal{S}$ must be maximally special in the sense that $\forall \pi_i \in \mathcal{S}, \not\exists \pi_j \in \mathcal{S}$ such that $\pi_j \ll \pi_i$. Analogously, the elements of $\mathcal{G}$ must be maximally general i.e., $\forall \pi_k \in \mathcal{G}, \not\exists \pi_l \in \mathcal{G}$ such that $\pi_l \ll \pi_k$.

2. Every element in $\mathcal{S}$ must have a corresponding more general element in $\mathcal{G}$ and vice-versa. i.e., $\forall \pi_i \in \mathcal{S}, \exists \pi_j \in \mathcal{G}$ such that $\pi_i \ll \pi_j$ and similarly, $\forall \pi_k \in \mathcal{G}, \exists \pi_l \in \mathcal{S}$ such that $\pi_l \ll \pi_k$.

3. The elements of $\mathcal{S}$ and $\mathcal{G}$ are consistent with all the examples observed by the learner. If $\hat{S}^+$ and $\hat{S}^-$ represent the set of positive and negative examples observed by the learner at any time then each partition π_k belonging to $\mathcal{S}$ or $\mathcal{G}$ must be such that M_k accepts every example in $\hat{S}^+$ and rejects every example in $\hat{S}^-$.

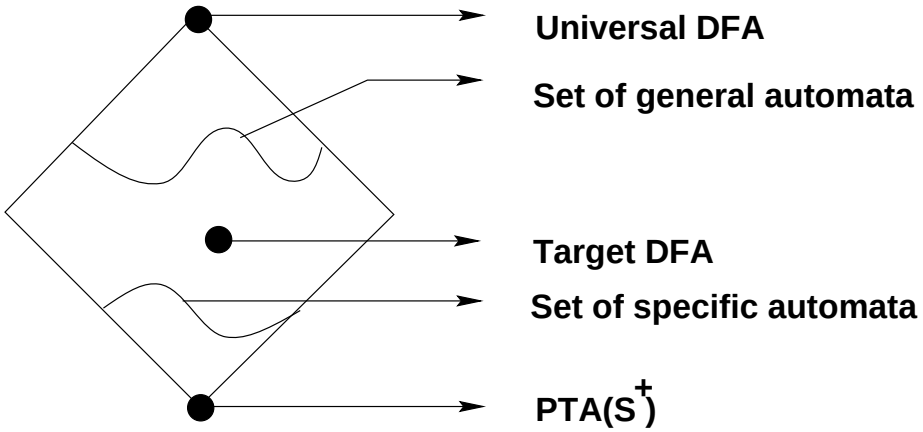

Figure 4.4 Bidirectional Search of Ω.

4.4 Query Aided Bi-Directional Search of the Lattice

The lattice Ω is implicitly represented by $\Theta = [\mathcal{S}, \mathcal{G}]$. $\mathcal{S}$ is initialized to $\{\pi_0\}$ and $\mathcal{G}$ is initialized to $\{\pi_{E_m-1}\}$. Θ implicitly represents the entire lattice. The search for the target

DFA now proceeds with the help of membership queries that are posed to the teacher.

The version space search for the target DFA is described in Fig. 4.5. At each step, a partition π_j from $\mathcal{S}$ and a partition π_i from $\mathcal{G}$ are picked and their corresponding automata M_j and M_i are compared for equivalence. If $M_i \not\equiv M_j$ then there exists a string y such that $y \in L(M_j)$ but $y \notin L(M_i)$ or vice-versa (in which case the roles of M_j and M_i are simply reversed). In other words, if $M_i \not\equiv M_j$ then there exists a string $y \in L(M_j) \oplus L(M_i)$. If we denote the FSA M_i and M_j respectively by $(Q_i, \delta_i, \Sigma, q_{0_i}, F_i)$ and $(Q_j, \delta_j, \Sigma, q_{0_j}, F_j)$ then the shortest string belonging to the difference automaton $M_i - M_j = (Q_d, \delta_d, \Sigma, q_{0_d}, F_d)$ where $Q_d = Q_i \times Q_j$, $\delta_d((q_i, q_j), a) = (\delta_i(q_i, a), \delta_j(q_j, a))$ for all $q_i \in Q_i$, $q_j \in Q_j$, and $a \in \Sigma$, $q_{0_d} = (q_{0_i}, q_{0_j})$, and $F_d = \{(q_i, q_j) \mid q_i \in F_i \text{ and } q_j \in Q_j \backslash F_j\}$ is selected as the query string y. The teacher's response to the query "$y \in L(A)$?" (where A is the target DFA) determines whether y is a positive example or a negative example. The version space is modified based on the teacher's response and the elements of $\mathcal{S}$ and $\mathcal{G}$ are made progressively more general and more special respectively. Specifically, when y is a positive example, partitions $\pi_i \in \mathcal{G}$ such that M_i rejects y are eliminated from $\mathcal{G}$. Further, partitions $\pi_j \in \mathcal{S}$ where M_j rejects y are *generalized* i.e., replaced by a set of partitions that are MGE π_j and whose corresponding FSA accept y. The procedure **Generalize** shown in Fig. 4.6 shows how a partition π_j whose corresponding FSA rejects a positive example is generalized. The operations on Θ when y is a negative example are analogous. The procedure **Specialize** shown in Fig. 4.7 shows how a partition π_i whose corresponding FSA accepts a negative example is specialized.

The version space algorithm (shown in Fig. 4.5) maintains two additional sets of partitions $\mathbf{S}^-$ and $\mathbf{G}^+$ to improve the efficiency of the search. $\mathbf{S}^-$ contains partitions belonging to $\mathcal{S}$ whose corresponding FSA were found to accept negative examples. Prior to adding any partition (say π_j) to $\mathcal{S}$ it is determined whether there is some $\pi_k \in \mathbf{S}^-$ such that $\pi_k \ll \pi_j$. If this is the case then by the grammar covers property we know that M_j would accept a negative example and hence π_j need not be considered any further. Similarly, $\mathbf{G}^+$ contains partitions from $\mathcal{G}$ whose corresponding FSA were found to reject a positive examples.

A partition of $\mathcal{S}$ whose corresponding DFA does not accept a positive example y is gen-

Algorithm: Version-Space-Search

Input: A structurally complete set S^+ and a teacher capable of answering membership queries.
Output: A DFA equivalent to the canonical representation of the target DFA A

begin
1) Construct $PTA(S^+)$
2) Initialize $\mathcal{S} = \{\pi_0\}$, $\mathcal{G} = \{\pi_{E_m-1}\}$, and $\mathbf{S}^- = \mathbf{G}^+ = \phi$
3) **while** (there exists $\pi_j \in \mathcal{S}$ and $\pi_i \in \mathcal{G}$ such that $M_j \not\equiv M_i$) **do**
   Generate the shortest string $y \in L(M_j) \oplus L(M_i)$
   Pose the membership query "$y \in L(A)$?"
   if $(y \in L(A))$
   then
      for (each $\pi_k \in \mathcal{G}$ such that $y \notin L(M_k)$) **do**
         $\mathcal{G} = \mathcal{G} \backslash \{\pi_k\}$
         $\mathbf{G}^+ = \mathbf{G}^+ \cup \{\pi_k\}$
      if ($\exists \pi_l \in \mathcal{S}$ such that $y \notin L(M_l)$)
      then perform **Generalize**$(\pi_l, y, \mathcal{S}, \mathcal{G}, \mathbf{S}^-)$
      end if
   else
      for (each $\pi_l \in \mathcal{S}$ such that $y \in L(M_l)$) **do**
         $\mathcal{S} = \mathcal{S} \backslash \{\pi_l\}$
         $\mathbf{S}^- = \mathbf{S}^- \cup \{\pi_l\}$
      if ($\exists \pi_k \in \mathcal{G}$ such that $y \in L(M_k)$)
      then perform **Specialize**$(\pi_k, y, \mathcal{S}, \mathcal{G}, \mathbf{G}^+)$
      end if
   end if
4) **if** $(\mathcal{S} = \phi$ or $\mathcal{G} = \phi)$
   then return error
   else return M_j corresponding to the smallest partition $\pi_j \in \mathcal{S}$
   end if
end

Figure 4.5 Version Space Search Algorithm.

eralized by the procedure **Generalize** (shown in Fig. 4.6) whose execution is described as follows:

1. Initially π_l is deleted from $\mathcal{S}$.
2. The set of immediate generalizations G_l of the partition π_l is computed.
3. The set G_l is processed as follows: Partitions that are MGE some partition already in $\mathcal{S}$ are removed for they violate the version space property that the set $\mathcal{S}$ should be maximally special. Partitions that do not have a corresponding more general partition in $\mathcal{G}$ are removed for they violate the version space property that each partition in $\mathcal{S}$ must be MSE than some partition in $\mathcal{G}$. Partitions that are MGE some partition in $\mathbf{S}^-$ are removed as explained earlier.
4. $\mathcal{S}$ is augmented with the remaining partitions in G_l.
5. Partitions in $\mathcal{S}$ whose corresponding FSA are either NFA or do not accept y are recursively generalized.

The termination of this recursive generalization procedure is easy to guarantee. Note that M_{E_m-1} corresponding to the partition π_{E_m-1} is the universal DFA and hence accepts any positive example. In the worst case a sequence of generalizations of a partition π_l is guaranteed to terminate with π_{E_m-1}. Thus, at the end of the recursive generalization it is guaranteed that all the partitions in set $\mathcal{S}$ will represent DFA and further, the DFA corresponding to each partition will accept the positive example y.

A partition of $\mathcal{G}$ whose corresponding DFA accepts a negative example is specialized as described by the procedure **Specialize** in Fig 4.7. The operations in **Specialize** are analogous to those described in the procedure **Generalize**.

4.5 Proof of Correctness

Let the target DFA be represented by A. The correctness of the algorithm follows from the following two theorems.

Procedure Generalize $(\pi_l, y, \mathcal{S}, \mathcal{G}, \mathbf{S}^-)$

begin
1) Delete π_l from $\mathcal{S}$
2) Let G_l be the set of immediate generalizations of π_l
3) **for** (each partition $\pi_x \in G_l$) **do**
   if $((\exists\, \pi_y \in \mathcal{S}$ such that π_x MGE $\pi_y)$ **or**
   $(\nexists\, \pi_y \in \mathcal{G}$ such that π_x MSE $\pi_y)$ **or**
   $(\exists\, \pi_y \in \mathbf{S}^-$ such that π_x MGE $\pi_y))$
   then remove π_x from G_l
   end if
4) $\mathcal{S} = \mathcal{S} \cup G_l$
5) **if** $(\exists \pi_x \in \mathcal{S}$ such that M_x is a NFA **or** $y \notin L(M_x))$
   then perform **Generalize** $(\pi_x, y, \mathcal{S}, \mathcal{G}, \mathbf{S}^-)$
   end if
6) **return** $\mathcal{S}$
end

Figure 4.6 Algorithm for Generalizing a Partition $\pi_l \in \mathcal{S}$.

Procedure Specialize $(\pi_k, y, \mathcal{S}, \mathcal{G}, \mathbf{G}^+)$

begin
1) Delete π_k from $\mathcal{G}$
2) Let S_k be the set of immediate specializations of π_k
3) **for** (each partition $\pi_x \in S_k$) **do**
   if $((\exists\, \pi_y \in \mathcal{G}$ such that π_x MSE $\pi_y)$ **or**
   $(\nexists\, \pi_y \in \mathcal{S}$ such that π_x MGE $\pi_y)$ **or**
   $(\exists \pi_y \in \mathbf{G}^+$ such that π_x MSE $\pi_y))$
   then remove π_x from S_k
   end if
4) $\mathcal{G} = \mathcal{G} \cup S_k$
5) **if** $(\exists \pi_x \in \mathcal{G}$ such that M_x is a NFA **or** $y \in L(M_x))$
   then perform **Specialize** $(\pi_x, y, \mathcal{S}, \mathcal{G}, \mathbf{G}^+)$
   end if
6) **return** $\mathcal{G}$
end

Figure 4.7 Algorithm for Specializing a Partition $\pi_k \in \mathcal{G}$.

Theorem 4.1 *The lattice Ω constructed from a structurally complete set of examples with respect to the target DFA A is guaranteed to contain a partition π_A such that M_A is exactly A.*

Proof:

The proof of this theorem is originally due to Pao and Carr [PC78] and has been reworked in [PH93]. It was also independently proven by Miclet (see [DMV94]). □

Theorem 4.2 *The following invariance condition holds at all times during the execution of the algorithm.*

$$\exists \pi_\beta \in \mathcal{G} \text{ and } \exists \pi_\alpha \in \mathcal{S} \text{ such that } \pi_\alpha \ll \pi_A \ll \pi_\beta$$

Proof:

We prove this theorem by induction.

Base Case:

Initially, $\mathcal{S} = \pi_0$ and $\mathcal{G} = \pi_{E_m-1}$. Therefore, the hypothesis space $\Theta = [\mathcal{S}, \mathcal{G}]$ implicitly includes a representation of the entire lattice Ω. Further, by construction, each partition $\pi_i \in \Omega$ is such that $\pi_0 \ll \pi_i \ll \pi_{E_m-1}$. Theorem 4.1 guarantees that $\pi_A \in \Omega$. Thus, the invariance condition holds if we set π_0 to π_α and π_{E_m-1} to π_β.

Induction Hypothesis:

Assume that the invariance condition holds just before processing a membership query.

Induction Proof:

We prove that the invariance condition continues to hold after processing the membership query. If the query string y is a positive example:

1. Any $\pi_k \in \mathcal{G}$ such that M_k rejects y is removed. No such π_k could be π_β or else, since $\pi_A \ll \pi_\beta$, by the grammar covers property A would also reject the positive example y which is a contradiction.

2. Consider that the designated partition π_α is such that M_α rejects y. π_α will thus be generalized. We now show that the modification of $\mathcal{S}$ in the procedure **Generalize** (shown in Fig. 4.6) does not violate the invariance.

(a) Step 1 deletes π_α from $\mathcal{S}$.

(b) Step 2 computes G_α, the set of minimum generalizations of π_α. Since, initially $\pi_\alpha \ll \pi_A$, it is clear that there will be at least one partition π_x (where $\pi_\alpha \preceq \pi_x$) such that $\pi_\alpha \ll \pi_x \ll \pi_A$. So, π_x can take over the role of π_α which was deleted in step 1.

(c) Step 3 removes elements from π_α that do not satisfy one (or more) of the version space properties. First, those partitions of G_α that are MGE some partition in $\mathcal{S}$ are removed. Clearly, if π_α is removed then the partition π_x in $\mathcal{S}$ where $\pi_x \ll \pi_\alpha$ can take over as the new π_α. Partitions in G_α that are not MSE some partition in $\mathcal{G}$ are removed. We have established above that $\pi_\alpha \ll \pi_A$. Further, from the invariance we know that there exists $\pi_\beta \in \mathcal{G}$ such that $\pi_A \ll \pi_\beta$. Thus, $\pi_\alpha \ll \pi_\beta$ and hence none of the partitions thus removed could correspond to π_α. Finally, π_α cannot be MGE any partition $\pi_x \in \mathbf{S}^-$ or else by the grammar covers property both M_α and M_A would accept the negative example accepted by M_x.

(d) Next, $\mathcal{S}$ is augmented with the remaining partitions in G_α. Thus, $\mathcal{S}$ contains a partition $\pi_\alpha \ll \pi_A$. If π_α is such that M_α does not accept the positive example y or M_α is a NFA then **Generalize** is invoked recursively and π_α generalized further. We know that $\pi_\alpha \ll \pi_A$. There is a sequence of one or more generalizations $\pi_\alpha \preceq \pi_{\alpha_1} \preceq \pi_{\alpha_2} \preceq \pi_{\alpha_3} \ldots \preceq \pi_A$. Clearly, M_A itself would accept the positive example y and is a DFA. Thus, the sequence of recursive generalizations would definitely yield a partition π_{α_i} (which could be π_A itself) where π_{α_i} would take over the role of π_α to satisfy the invariance.

At the end of the procedure, the modified set $\mathcal{S}$ contains a partition $\pi_\alpha \ll \pi_A$. Further, there is a partition $\pi_\beta \in \mathcal{G}$ such that $\pi_A \ll \pi_\beta$. This proves that the invariance continues to hold after a positive example y is processed.

The arguments for the case when y is a negative example are analogous to those presented above. This proves that the invariance holds at all times. □

Theorem 4.1 guarantees the existence of the target partition in the lattice. Theorem 4.2 shows that at each step during the search process the target partition is implicitly maintained in the search space represented by the current states of $\mathcal{S}$ and $\mathcal{G}$ respectively. The algorithm terminates when $\mathcal{S}$ and $\mathcal{G}$ each contain the same set equivalent FSA. At this time, by virtue of the invariance condition of theorem 4.2 $\pi_\alpha = \pi_A = \pi_\beta$ and the algorithm correctly identifies the target partition.

Example

Consider the DFA A in Fig. 4.1. $S^+ = \{abb\}$ is a structurally complete set with respect to A. $PTA(S^+)$ is depicted in Fig. 4.2. The corresponding lattice Ω is shown in Fig. 4.3. The version space is initialized to $\Theta = [\mathcal{S}, \mathcal{G}]$ where $\mathcal{S} = \{\pi_0\}$ and $\mathcal{G} = \{\pi_{14}\}$. The execution of the version space search is summarized in Table 4.1.

Table 4.1 Version Space Search.

Step	Θ	$M_i \equiv M_j$?	Query y	Modified Θ
1	$\mathcal{S} = \{\pi_0\}; \mathcal{G} = \{\pi_{14}\}$	$M_0 \not\equiv M_{14}$	$\lambda \in$ L(A)	$\mathcal{S} = \{\pi_3\}; \mathcal{G} = \{\pi_{14}\}$
2	$\mathcal{S} = \{\pi_3\}; \mathcal{G} = \{\pi_{14}\}$	$M_3 \not\equiv M_{14}$	$a \notin$ L(A)	$\mathcal{S} = \{\pi_3\}; \mathcal{G} = \{\pi_9\}$
3	$\mathcal{S} = \{\pi_3\}; \mathcal{G} = \{\pi_9,\}$	$M_3 \not\equiv M_9$	$b \in$ L(A)	$\mathcal{S} = \{\pi_9\}; \mathcal{G} = \{\pi_9\}$

In the first step, M_0 (the PTA) and M_{14} (the universal DFA) are compared for equivalence. $M_0 \not\equiv M_{14}$ and $y = \lambda$ is the shortest string belonging to $L(M_0) \oplus L(M_{14})$. Since $\lambda \in L(A)$, λ is a positive example. $y \in L(M_{14})$ so $\mathcal{G}$ does not change. Since, $y \notin L(M_0)$, the procedure **Generalize** is invoked with partition π_0. The generalization of π_0 proceeds as shown in Fig. 4.6. π_0 is removed from $\mathcal{S}$ thereby making $\mathcal{S} = \phi$. The set of immediate generalizations of π_0 is $G_0 = \{\pi_1, \pi_2, \ldots, \pi_6\}$. No partition in G_0 is MGE some partition in $\mathcal{S}$ because $\mathcal{S}$ is empty. All partitions in G_0 are MSE $\pi_{14} \in \mathcal{G}$. No partition in G_0 is MGE some partition in $\mathbf{S}^-$ because $\mathbf{S}^-$ is empty at this point. After performing $\mathcal{S} = \mathcal{S} \cup G_0$ we get $\mathcal{S} = \{\pi_1, \pi_2, \ldots, \pi_6\}$. Since M_1 does not accept $y = \lambda$, π_1 is generalized by invoking the procedure **Generalize** recursively. Continuing with the execution of **Generalize** we can see that $\mathcal{S} = \{\pi_3\}$ is eventually returned by the procedure. This completes step 1 of Table 4.1.

In step 2, M_3 and M_{14} are compared for equivalence. $M_3 \not\equiv M_{14}$ and $y = a$ is the query generated. Since y is a negative example π_{14} is specialized (as described in Fig. 4.7) resulting in $\mathcal{G} = \{\pi_9\}$. The final step compares M_3 with M_9 and poses the query $y = b$ to the teacher. Since y is a positive example π_3 is generalized resulting in $\mathcal{S} = \{\pi_9\}$. At this time, $\mathcal{S} = \mathcal{G}$ and the partitions in $\mathcal{S}$ and $\mathcal{G}$ are equivalent to each other. The search terminates returning M_9 as the inferred DFA. It is easy to see that M_9 is indeed the target DFA shown in Fig. 4.1.

4.6 Discussion

We have presented a provably correct method for inference of a target DFA from a structurally complete set of examples and membership queries. The version space is a compact representation of the lattice of candidate FSA. It implicitly represents all elements of the hypothesis space that are consistent with data observed by the learner. An efficient bidirectional search strategy is used to identify the target DFA. Our algorithm has the following advantages when compared to the approach suggested by Pao and Carr [PC78]:

1. *Implicit representation of the hypothesis space.*

   Pao and Carr's algorithm explicitly constructs the entire lattice Ω. Even for moderately small structurally complete sets of examples the size of the lattice is prohibitively large for explicit enumeration.

2. *Restricting the search to DFA alone.*

   Pao and Carr's approach allows the search to consider both DFA and NFA as candidate solutions to the inference problem and requires that a NFA be converted to a fully specified DFA before comparing it for equivalence with another FSA. The process of converting a NFA to an equivalent DFA has exponential time complexity in the worst case [HU79]. Our method restricts the search to DFA alone thereby circumventing the problem.

3. *Partial inference using the version space.*

   The properties of the version space allow the algorithm to make partial inferences even

before the algorithm has converged to the target DFA. If an example y is accepted by all the FSA belonging to the set $\mathcal{S}$ then y can be unambiguously classified as a positive example. Similarly, an example y that is rejected by all FSA in $\mathcal{G}$ can be unambiguously classified as a negative example. The explicit enumeration of the hypothesis space as in Pao and Carr's algorithm does not permit the learner to make such partial inferences without actually testing whether or not each FSA in the hypothesis space accepts the example.

VanLehn and Ball [VB87] have proposed a version space based approach to learning context free grammars from a set of positive and negative examples. Their algorithm returns a set of grammars consistent with the given sample set. Their algorithm is also based on a lattice of partitions. The version space is represented by a triple $[\mathrm{S}^+, \mathrm{S}^-, \mathcal{G}]$ where S^+ and S^- are sets of positive and negative examples respectively and $\mathcal{G}$ is the set of generalizations. The learner is required to store all the examples seen earlier for future reference. By restricting our approach to inference of regular grammars the version space is finite and compactly represented by $[\mathcal{S}, \mathcal{G}]$. Our algorithm does not store the previous examples and is guaranteed to converge to the desired target instead of a set of candidate solutions as is the case for VanLehn and Ball's method.

The dense inter-connectivity among the lattice elements poses a limitation for our algorithm. Each partition in the lattice can be realized by *generalizing* (*specializing*) several different partitions. For example, in Fig. 4.3, π_9 can be obtained as a result of generalizing partitions $\pi_0, \pi_2, \pi_3,$ and π_6. Thus, it is likely that several partitions will be generated and evaluated multiple times during the search which makes this approach inefficient. A trivial upper bound on the number of membership queries required in the version space based search is exponential in the number of states of $PTA(S^+)$. It is difficult to perform an analysis of the average case performance of the algorithm. Angluin proposed an algorithm (ID) to infer the target grammar from a *live complete set* of examples (which can be constructed from a structurally complete set) using a polynomial number of membership queries [Ang81]. Our approach offers an alternative to the ID procedure when a structurally complete set of sam-

ples is available. However, given the limitations of our algorithm in terms of the worst case performance it is clear that the ID algorithm is preferable to our method for learning the target DFA from a structurally complete set of examples and membership queries.

It is of interest to see if a more efficient search strategy that guarantees a polynomial worst case bound on the number of membership queries can be designed for our version space based learning framework. This search could perhaps generate more informative queries or possibly use the results of a polynomial number of queries posed simultaneously to speed up learning. An extension of the proposed approach for learning regular *tree* and *attributed* grammars [Fu82] also merits further investigation.

5 AN INCREMENTAL ALGORITHM FOR LEARNING DFA FROM LABELED EXAMPLES AND MEMBERSHIP QUERIES

5.1 Introduction

In chapter 4 we studied a version space based approach for learning the target DFA from a structurally complete set of examples and membership queries. Angluin's *ID* algorithm presents a framework for learning the target DFA from a live complete set of examples (see section 3.3.6) and membership queries [Ang81]. In many practical learning scenarios, a structurally complete set or a live complete set might not be available to the learner at the outset. Instead, a sequence of labeled examples is provided intermittently and the learner is required to construct an approximation of the target DFA based on the examples and the queries answered by the teacher. In such scenarios, an online or incremental model of learning that is guaranteed to eventually converge to the target DFA in the limit is of interest.

An incremental extension of the version space based learning algorithm of chapter 4 was described in [PH96]. The algorithm assumes that positive examples needed to construct a structurally complete set are intermittently presented to the learner. The learner constructs an initial version space representation of the lattice from the set of positive examples S_0 available to it at the start. The version space is denoted by $\Theta_0 = [\mathcal{S}_0, \mathcal{G}_0]$. S_0 is structurally complete with respect to a sub-automaton A_0 of the target. Using an argument similar to that in Theorem 4.1 it can be showed that Θ_0 is guaranteed to contain a representation of A_0. A bidirectional version space search strategy based on membership queries is used to search the lattice for A_0. In order to guarantee that the representation of A_0 is always implicitly contained in Θ_0, the modification of the sets $\mathcal{S}_0$ and $\mathcal{G}_0$ must be based on *safe membership queries* as explained below. The search is continued until no further elimination of elements of

Θ_0 is possible using safe queries. When an additional positive example is provided the current version space Θ_i is extended using a technique called *incremental version space merging* [Hir90] to give the modified version space Θ_{i+1}. The set $S_{i+1} = S_i \cup \{s\}$, where s is the new positive example provided to the learner, is guaranteed to be structurally complete with respect to a sub-automaton A_{i+1} of the target DFA. The incremental version space merging ensures that Θ_{i+1} implicitly contains a representation of A_{i+1}. The bidirectional search continues in the augmented space Θ_{i+1} using safe queries. This alternate lattice expansion and candidate elimination continues until a point when the set S_k is structurally complete with respect to the target DFA. The current version space Θ_k is then searched by treating all queries as *safe queries* (just as in the standard version space based algorithm).

The version space Θ_i is guaranteed to contain a representation of a sub-automaton A_i of A. By definition, $L(A_i) \subseteq L(A)$. A negative example of A is clearly also a negative example of A_i. However, the same is not true of a positive example of A. In order to ensure that the representation of A_i is not eliminated from Θ_i, the incremental algorithm requires that a query string that is a positive example of A is deemed unsafe if its length is greater than that of all examples in the set S_i. Further, the algorithm requires that strings belonging to the structurally complete set be provided in increasing order by length. Thus, given a bound on the number of states of A, the learner can determine when the set of examples S_k is structurally complete with respect to A. At this point, the version space Θ_k is guaranteed to contain a representation of A and no further lattice expansions would be required. All queries can then be treated as safe queries. The algorithm is guaranteed to converge to the target. However, it has the following drawbacks. The algorithm requires that the strings belonging to the structurally complete set be provided in increasing order by length. The learner must be given a bound on the number of states of the target DFA or must be explicitly signaled when it has seen a structurally complete set of examples. Further, this algorithm shares the same limitations in terms of worst case number of queries as the non-incremental one.

In this chapter we present an extension of *ID* to an incremental setting. The proposed algorithm *IID* (incremental *ID*) is a polynomial time interactive algorithm for learning the

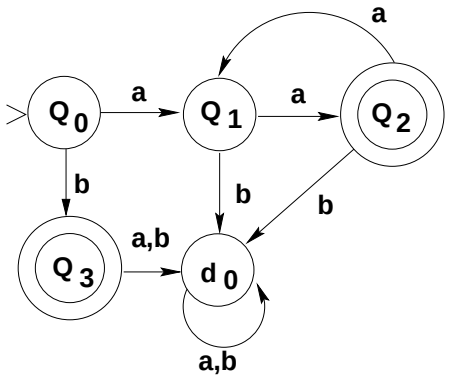

Figure 5.1 Target DFA A.

target DFA from labeled examples and membership queries [PNH97]. *IID* overcomes the limitations of the version space based incremental algorithm in that it runs in polynomial time and does not require either the knowledge of a bound on the number of states of the target DFA or the presentation of examples in increasing order by length.

Section 5.2 briefly reviews the *ID* algorithm. The interested reader is referred to [Ang81] for a complete description of the algorithm and its correctness proof. Section 5.3 describes *IID* together with its correctness proof and an analysis of its time and space complexities. Section 5.4 concludes with a brief discussion of how *IID* relates with other incremental algorithms for learning the target DFA.

5.2 The *ID* Algorithm

In order to keep the discussion in this chapter self-contained we will briefly review some pertinent definitions and the *ID* algorithm. Let A denote the target DFA. A live complete set for A is a set of strings such that for every live state q_i of A there is a string α in P such that $\delta^*(q_0, \alpha) = q_i$. Thus, if we consider the DFA in Fig. 5.1 to be the target DFA A, then $P = \{\lambda, a, b, aa\}$ is a live complete set for A. The set $P' = P \cup \{d_0\}$ represents all the states of the target DFA including the dead state d_0.

The function $f : P' \times \Sigma \longrightarrow \Sigma^* \cup \{d_0\}$ essentially represents the transitions going out of each state of the target DFA. By definition $f(d_0, a) = d_0$ which states that all transitions out of the dead state must be self-loops. $f(\alpha, a) = \alpha a$ specifies the destination state of a transition,

on a letter $a \in \Sigma$, out of the state represented by the string α. Thus, the set $T' = P' \cup \{f(\alpha, a) | (\alpha, a) \in P' \times \Sigma\}$ collectively represents the set of all states and the destination states of each transition of the target DFA. Given a live complete set $P = \{\lambda, a, b, aa\}$ corresponding to A the set $T' = \{d_0, \lambda, a, b, aa, ab, ba, bb, aaa, aab\}$. The set T is defined as $T = T' \backslash \{d_0\}$.

ID constructs a partition of the set T' such that the elements of T' that represent the same state of A are grouped together in the same block of the partition. In the process a set of distinguishing suffixes V is constructed such that no two distinct states of A have the same behavior on all strings in V i.e., for any two distinct states q_i and q_j of A there exists a string $\alpha \in V$ such that $\delta(q_i, \alpha) \in F$ and $\delta(q_i, \alpha) \notin F$ or vice-versa. When the set V has i elements, define function $E_i : T' \longrightarrow 2^V$ as follows $E_i(d_0) = \phi$ and $E_i(\alpha) = \{v_j | v_j \in V, 0 \leq j < i, \alpha v_j \in L(A)\}$. $E_i(\alpha)$ is a subset of $L(A)_\alpha$, the set of tails of the string α in $L(A)$. Fig. 5.2 describes the algorithm in detail. Step 1 performs the initialization. The set T_0 represents a trivial partition with all elements belonging to a single block. The first distinguishing suffix v_0 considered is λ. The function E_0 which is computed in step 2 partitions T' into two blocks, representing the accepting and non-accepting states of A respectively. Step 3 refines the individual blocks of the partition of T' based on the behavior of the elements on the distinguishing suffixes $v_0, v_1, \ldots, v_i$. Intuitively, if two elements of T', say α and β, have the same behavior on the current set V (i.e., $E_i(\alpha) = E_i(\beta)$) then α and β appear to represent the same state of the target DFA. However, if the transitions out of the states represented by $E_i(\alpha)$ and $E_i(\beta)$ on some letter of the alphabet lead to different states (i.e., $E_i(f(\alpha, a)) \neq E_i(f(\beta, a))$ for some $a \in \Sigma$) then clearly, α and β cannot correspond to the same state in the target DFA. A distinguishing suffix v_{i+1} is constructed to refine the partition of T' such that α and β appear in separate blocks of the partition. Step 3 terminates when the set V contains a distinguishing suffix for each pair of elements in T' that represent non-equivalent states of A. Step 4, finally constructs the hypothesis DFA M.

Algorithm: ID

Input: A live complete set P and a teacher to answer membership queries.
Output: A DFA M equivalent to the target DFA A.

begin

1) *// Perform initialization*
   $i = 0$, $v_i = \lambda$, $V = \{\lambda\}$, $T = P \cup \{f(\alpha, b) \mid (\alpha, b) \in P \times \Sigma\}$ and $T' = T \cup \{d_0\}$
2) *// Construct function E_0 for $v_0 = \lambda$*
   $E_0(d_0) = \phi$
   $\forall \alpha \in T$ pose the membership query "$\alpha \in L(A)$?"
      if the teacher's response is *yes*
      then $E_0(\alpha) = \{\lambda\}$
      else $E_0(\alpha) = \phi$
      end if
3) *// Refine the partition of the set T'*
   while ($\exists \alpha, \beta \in P'$ and $b \in \Sigma$ such that
      $E_i(\alpha) = E_i(\beta)$ but $E_i(f(\alpha, b)) \neq E_i(f(\beta, b))$)
   do
      Let $\gamma \in E_i(f(\alpha, b)) \oplus E_i(f(\beta, b))$
      $v_{i+1} = b\gamma$
      $V = V \cup \{v_{i+1}\}$ and $i = i + 1$
      $\forall \alpha \in T$ pose the membership query "$\alpha v_i \in L(A)$?"
         if the teacher's response is *yes*
         then $E_i(\alpha) = E_{i-1}(\alpha) \cup \{v_i\}$
         else $E_i(\alpha) = E_{i-1}(\alpha)$
         end if
   end while
4) *// Construct the representation of the DFA M*
   The states of M are the sets $E_i(\alpha)$, where $\alpha \in T$
   The initial state q_0 is the set $E_i(\lambda)$
   The accepting states are the sets $E_i(\alpha)$ where $\alpha \in T$ and $\lambda \in E_i(\alpha)$
   The transitions of M are defined as follows:
      $\forall \alpha \in P'$
         if $E_i(\alpha) = \phi$
         then add self loops on the state $E_i(\alpha)$ for all $b \in \Sigma$
         else $\forall b \in \Sigma$ set the transition $\delta(E_i(\alpha), b) = E_i(f(\alpha, b))$
         end if

end

Figure 5.2 Algorithm *ID*.

5.2.1 Example

We now demonstrate the use of *ID* to learn the target DFA in Fig. 5.1. $P = \{\lambda, a, b, aa\}$ is a live complete set and $T' = \{d_0, \lambda, a, b, aa, ab, ba, bb, aaa, aab\}$. Table 5.1 shows the computation of $E_i(\alpha)$ for the strings $\alpha \in T'$. The leftmost column lists the elements α of the set T'. Each successive column represents the function E_i corresponding to the string v_i (indicated in the second row of the table).

Table 5.1 Execution of ID.

i	0	1	2	3
v_i	λ	b	a	aa
$E(d_0)$	ϕ	ϕ	ϕ	ϕ
$E(\lambda)$	ϕ	$\{b\}$	$\{b\}$	$\{b, aa\}$
$E(a)$	ϕ	ϕ	$\{a\}$	$\{a\}$
$E(b)$	$\{\lambda\}$	$\{\lambda\}$	$\{\lambda\}$	$\{\lambda\}$
$E(aa)$	$\{\lambda\}$	$\{\lambda\}$	$\{\lambda\}$	$\{\lambda, aa\}$
$E(ab)$	ϕ	ϕ	ϕ	ϕ
$E(ba)$	ϕ	ϕ	ϕ	ϕ
$E(bb)$	ϕ	ϕ	ϕ	ϕ
$E(aaa)$	ϕ	ϕ	$\{a\}$	$\{a\}$
$E(aab)$	ϕ	ϕ	ϕ	ϕ

Note that the DFA returned by the procedure is exactly the DFA in Fig. 5.1. The number of membership queries posed by the learner is at most $O(|\Sigma| \cdot N \cdot |P|)$. Further, the time and space complexities of the algorithm are polynomial in $|\Sigma|$, N, and $|P|$ [Ang81].

5.3 *IID* - An Incremental Extension of *ID*

We now present an incremental version of the *ID* algorithm. As stated earlier, this algorithm does not require that the live complete set of examples be available to the learner at the start. Instead the learner is intermittently presented with labeled examples. The learner constructs a model of the target DFA based on the examples it has seen and gradually refines it as new examples become available. Our learning model assumes the availability of a teacher to answer membership queries. Let M_t denote the DFA that corresponds to the learner's current model after observing t examples. Initially, M_0 is a *null* automaton with only one state (the dead

state) and it rejects every string in $\Sigma^\star$. Clearly, every negative example encountered by the learner at this point is consistent with M_0. Without loss of generality we assume that the first example seen by the learner is a positive example. When the first positive example, α, is seen M_0 is modified to accept the positive example. With each additional observed example, α, it is determined whether α is consistent with M_t in which case $M_{t+1} = M_t$. Otherwise M_t is suitably modified such that α is consistent with the resulting DFA, M_{t+1}. A detailed description of the algorithm appears in Fig. 5.3.

5.3.1 Example

We now demonstrate how the incremental algorithm learns the target DFA of Fig. 5.1. The learner starts with a model M_0 equivalent to the *null* DFA accepting no strings. Suppose the example $(b, +)$ is encountered. The following actions are taken. $P_0 = \{\lambda, b\}$ and $P_0^{'} = \{d_0, \lambda, b\}$, $T_0 = \{\lambda, a, b, ba, bb\}$, and $T_0^{'} = \{d_0, \lambda, a, b, ba, bb\}$. The computation of the functions E_i is shown in Table 5.2. At this point the learner constructs a model M_1 of the target DFA (Fig. 5.4).

Table 5.2 Execution of *IID* ($k = 0$).

i	0	1
v_i	λ	b
$E(d_0)$	ϕ	ϕ
$E(\lambda)$	ϕ	$\{b\}$
$E(a)$	ϕ	ϕ
$E(b)$	$\{\lambda\}$	$\{\lambda\}$
$E(ba)$	ϕ	ϕ
$E(bb)$	ϕ	ϕ

Suppose the next example observed by the learner is $(a, -)$. Since, M_1 correctly rejects a, $M_2 = M_1$ and the learner waits for additional examples. Let $(aa, +)$ be the next observed example. Since $aa \notin L(M_2)$ the learner takes the following steps to update M_2. $P_1 = \{\lambda, a, b, aa\}$, $P_1^{'} = \{d_0, \lambda, a, b, aa\}$, $T_1 = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab\}$, and $T_1^{'} = \{d_0, \lambda, a, b, aa, ab, ba, bb, aaa, aab\}$. The function E_1 is extended to cover the new elements belonging to $T_1 \backslash T_0$. The resulting computation of the various E_i's is depicted in Table 5.3.

The revised model of the target DFA (M_3) is exactly the DFA we are trying to learn

Algorithm: IID

Input: A stream of labeled examples and a teacher to answer membership queries.
Output: A DFA M_t consistent with all t examples observed by the learner.

begin

1) *// Perform initialization*
   $i = 0$, $k = 0$, $t = 0$, $P_k = \phi$, $T_k = \phi$, $V = \phi$
   Initialize M_t to the *null DFA*
2) // Process the first positive example
   Wait for a positive example $(\alpha, +)$
   $P_0 = Pref(\alpha)$ and $P_0^{'} = P_0 \cup \{d_0\}$
   $T_0 = P_0 \cup \{f(\alpha, b) | (\alpha, b) \in P_0 \times \Sigma\}$ and $T_0^{'} = T_0 \cup \{d_0\}$
   $v_0 = \lambda$ and $V = \{v_0\}$
   $E_0(d_0) = \phi$
   $\forall \alpha \in T_0$ pose the membership query "$\alpha \in L(A)$?"
      if the teacher's response is *yes*
      then $E_0(\alpha) = \{\lambda\}$
      else $E_0(\alpha) = \phi$
      end if
3) *// Refine the partition of the set* $T_k^{'}$ (step 3 of Fig. 5.2)
4) *// Construct the current representation* M_t *of the target DFA* (step 4 of Fig. 5.2)
5) *// Process a new labeled example*
   Wait for a new example $(\alpha, c(\alpha))$
   if α is consistent with M_t
   then
      $M_{t+1} = M_t$
      $t = t + 1$
      goto step 5
   else
      $P_{k+1} = P_k \cup Pref(\alpha)$ and $P_{k+1}^{'} = P_{k+1} \cup \{d_0\}$
      $T_{k+1} = T_k \cup Pref(\alpha) \cup \{f(\alpha, b) | (\alpha, b) \in (P_{k+1} \backslash P_k) \times \Sigma\}$ and $T_{k+1}^{'} = T_{k+1} \cup \{d_0\}$
      $\forall \alpha \in T_{k+1} \backslash T_k$ fill in the entries for $E_i(\alpha)$ by posing membership queries:
         $E_i(\alpha) = \{v_j | 0 \leq j < i, \alpha v_j \in L(A)\}$
      $k = k + 1$
      $t = t + 1$
      goto step 3
   end if

end

Figure 5.3 Algorithm *IID*.

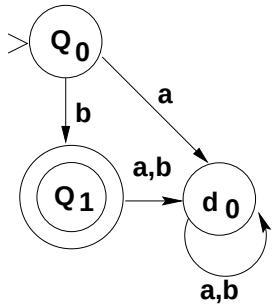

Figure 5.4 Model M_1 of the Target DFA.

Table 5.3 Execution of *IID* ($k = 1$).

i	1	2	3
v_i	b	a	aa
$E(d_0)$	ϕ	ϕ	ϕ
$E(\lambda)$	$\{b\}$	$\{b\}$	$\{b, aa\}$
$E(a)$	ϕ	$\{a\}$	$\{a\}$
$E(b)$	$\{\lambda\}$	$\{\lambda\}$	$\{\lambda\}$
$E(ba)$	ϕ	ϕ	ϕ
$E(bb)$	ϕ	ϕ	ϕ
$E(aa)$	$\{\lambda\}$	$\{\lambda\}$	$\{\lambda, aa\}$
$E(ab)$	ϕ	ϕ	ϕ
$E(aaa)$	ϕ	$\{a\}$	$\{a\}$
$E(aab)$	ϕ	ϕ	ϕ

(Fig. 5.1). Note also that at this time the set P_1 is live complete with respect to the target DFA.

5.3.2 Correctness Proof

The correctness of *IID* is a direct consequence of the following two theorems.

Theorem 5.1 *IID converges to a canonical representation of the target DFA when the set P_k includes a live complete set for the target as a subset.*

Proof:

Consider an execution of *ID* given a live complete set P_l. First we demonstrate that the execution of *ID* can be made to track that of *IID* in that the set V generated during the

execution of both the algorithms is the same and hence $\forall \alpha \in T_l$ the values $E_i(\alpha)$ are the same. We prove this claim by induction.

Base Case:

Both *ID* and *IID* start with $v_0 = \lambda$. At $k = 0$, *IID* has the set $P_0 \subseteq P_l$ available to it. Clearly, for all strings $\alpha, \beta \in P_0$ such that $E_0(\alpha) = E_0(\beta)$ but $E_0(f(\alpha, b)) \neq E_0(f(\beta, b))$ in the case of *IID* it is also the case that the same strings $\alpha, \beta \in P_l$ for *ID* such that $E_0(\alpha) = E_0(\beta)$ but $E_0(f(\alpha, b)) \neq E_0(f(\beta, b))$. Assume that one such pair α, β is selected by both *ID* and *IID*. The string $\gamma \in E_0(f(\alpha, b)) \oplus E_0(f(\beta, b))$ can only be λ. Thus, the string $v_1 = b\gamma$ is the same for both the executions.

Induction Hypothesis:

Assume that after observing t examples, at some value of k $(0 \leq k < l)$, when $P_k \subseteq P_l$ is available to *IID*, the sequence of strings $v_0, v_1, \ldots, v_i$ and $\forall \alpha \in T_k$ the values $E_i(\alpha)$ are the same for the executions of both *ID* and *IID*.

Induction Proof:

We now show that the same string v_{i+1} is the generated by both *ID* and *IID*. Following the reasoning presented in the base case, and given the induction hypothesis, we can state that for all strings $\alpha, \beta \in P_k$ such that $E_i(\alpha) = E_i(\beta)$ but $E_i(f(\alpha, b)) \neq E_i(f(\beta, b))$ in the case of *IID* it is also the case that the same strings $\alpha, \beta \in P_l$ for *ID* such that $E_i(\alpha) = E_i(\beta)$ but $E_i(f(\alpha, b)) \neq E_i(f(\beta, b))$.

Assume that one such pair α, β is selected by both executions. By the induction hypothesis $E_i(f(\alpha, b)) \oplus E_i(f(\beta, b))$ is identical for both. Thus, given that the same string $\gamma \in E_i(f(\alpha, b)) \oplus E_i(f(\beta, b))$ is selected, the string $v_{i+1} = b\gamma$ is identical for both executions. When the live complete set P_l is available to *IID*, $\forall \alpha \in T_l$ the values of $E_i(\alpha)$ are exactly the same as the corresponding values of $E_i(\alpha)$ for *ID*.

Given a live complete set of examples, *ID* outputs a canonical representation of the target DFA A [Ang81]. From above we know that at $k = l$ the current model (M_t) of the target automaton maintained by *IID* is identical to one arrived at by *ID*. Thus, we have proved that *IID* converges to a canonical representation of the target DFA. □

Theorem 5.2 *At any time during the execution of* IID*, all the t examples observed by the learner are consistent with M_t, the current representation of the target.*

Proof:

Consider an example α that is not consistent with M_t. *IID* modifies M_t and constructs M_{t+1} a new representation of the target. From step 5 in Fig. 5.3 we see that $\alpha \in P_{k+1}$ and hence $\alpha \in T_{k+1}$. E_i is extended to all elements of $T_{k+1} \backslash T_k$. Thus, $\lambda \in E_i(\alpha)$ if α is a positive example of A and $\lambda \notin E_i(\alpha)$ if α is a negative example of A. M_{t+1} is constructed from E_j for some $j \geq i$. Since, $E_i(\alpha) \subseteq E_j(\alpha)$ it is clear that α will be accepted (rejected) by M_{t+1} if it a positive (negative) example of A. We now show that all strings μ that were consistent with M_t are also consistent with M_{t+1}.

The set $\{E_i(\beta) \mid \forall \beta \in T_k\}$ represents the set of states of M_t as shown in step 4 of the algorithm (see Fig. 5.3). Thus, for any string $\mu \in \Sigma^*$ and a state q of M_t there is a corresponding string $\beta \in T_k$ such that $\delta^*(q_0, \mu) = \delta^*(q_0, \beta) = q$. We say that μ is consistent with M_t if either μ is a positive example of A and $\lambda \in E_i(\beta)$ or μ is a negative example of A and $\lambda \notin E_i(\beta)$. Now assume that the algorithm observes a string α that is not consistent with M_t. T_k is modified to T_{k+1} and the function E_i is extended to the elements of $T_{k+1} \backslash T_k$. The algorithm then proceeds to refine the partition of T_{k+1} by generating the distinguishing suffixes $v_{i+1}, v_{i+2}, \ldots, v_j$ and constructing the functions $E_{i+1}, E_{i+2}, \ldots, E_j$. Consider that there exists a string $\gamma \in T_{k+1}$ such that $E_l(\beta) = E_l(\gamma)$ but $E_{l+1}(\beta) \neq E_{l+1}(\gamma)$ for some l where $i \leq l \leq j-1$. Clearly, $E_{l+1}(\beta) \oplus E_{l+1}(\gamma) = v_{l+1}$. Further, $v_{l+1} \neq \lambda$ because $v_0 = \lambda$ is already chosen as the first distinguishing suffix. Thus, $\lambda \notin E_{l+1}(\beta) \oplus E_{l+1}(\gamma)$. The string μ that originally corresponded to the state represented by β would now correspond either to the state represented by β or to the state represented by γ. Further λ will either belong to both $E_{l+1}(\beta)$ and $E_{l+1}(\gamma)$ or to neither depending on whether λ was a member of both $E_l(\beta)$ and $E_l(\gamma)$ or not. Continuing with the argument we can see that there is some string $\kappa \in T_{k+1}$ where $\kappa = \beta$ or $E_j(\beta) \oplus E_j(\kappa) \subseteq \{v_{i+1}, v_{i+2}, \ldots, v_j\}$ such that μ corresponds to κ in that $\delta^*(q_0, \mu) = \delta^*(q_0, \kappa) = q$ for some state q in M_{t+1}. Further, since $\lambda \in E_j(\kappa)$ *iff* $\lambda \in E_j(\beta)$ or equivalently $\lambda \in E_j(\kappa)$ *iff* $\lambda \in E_i(\beta)$ we see that μ is consistent with M_{t+1}. This proves that

all strings that were consistent with M_t continue to be consistent with M_{t+1}. □

5.3.3 Complexity Analysis

Assume that at some $k = l$ the set P_l includes a live complete set for the target DFA A as a subset. From the correctness proof of the algorithm, the current representation of the target M_t is equivalent to the target A.

The size of T_l is at most $|\Sigma| \cdot |P_l| + 1$. Also, the size of V is no more than N (the number of states of A). Thus, the total number of membership queries posed by the learner is $O(|\Sigma| \cdot |P_l| \cdot N)$. Searching for a pair of strings α, β to distinguish two states in the current representation of the target takes time that is $O(T_l^2)$. Thus, the incremental algorithm runs in time polynomial in N, $|\Sigma|$, and $|P_l|$. Since the size of T_l is at most $|\Sigma| \cdot |P_l| + 1$ and the size of V is no more than N the space complexity of the algorithm is $O(|\Sigma| \cdot |P_l| \cdot N)$.

5.4 Discussion

Incremental or online learning algorithms play an important role in situations where all the training examples are not available to the learner at the start. We have proposed an incremental version of the *ID* algorithm for identifying the target DFA from a set of labeled examples and membership queries. The algorithm is guaranteed to converge to the target DFA and has polynomial time and space complexities. One practical application of incremental learning of DFA is in modeling the behavior of intelligent autonomous agents [CM96]. The behavior of agents such as robots can be modeled using a DFA. Incremental approaches to learning DFA provide these agents with a framework to learn from experience in unfamiliar environments. Given its efficiency and guaranteed consistency with all examples, our algorithm can provide an effective tool for agent learning, especially in an interactive setting.

The L^* algorithm for learning the target DFA is based on *membership* and *equivalence* queries [Ang87]. The equivalence queries can be replaced by a polynomial number of calls to an oracle that supplies labeled examples to give an efficient PAC algorithm for learning DFA from labeled examples and membership queries. The *IID* algorithm differs from L^* in

the following respects: *IID* is guaranteed to learn the target DFA exactly and it uses only labeled examples and membership queries whereas L^* makes use of equivalence queries in addition to labeled examples and membership queries to guarantee exact learning of the target DFA. In contrast, the PAC version of L^* only guarantees that the target would be learned probabilistically i.e., with a very high probability, the DFA output by the algorithm would make very low error (when compared to the target).

Two prominent incremental algorithms for learning DFA are due to Porat and Feldman [PF91] and Dupont [Dup96a] respectively. Porat and Feldman's algorithm learns the target DFA in the limit from a *complete ordered sample*. A complete ordered sample includes all the strings in Σ^* in strict lexicographic order. The algorithm uses only finite working storage. At each step the algorithm tests the next labeled example for consistency with its current hypothesis. The current hypothesis is modified if necessary to ensure that it is consistent with the example. Each modification of the hypothesis requires a consistency check with all the previous examples seen by the algorithm. The algorithm is guaranteed to converge to the target DFA in the limit.

The *regular positive and negative inference* (RPNI) is a polynomial time algorithm for learning a DFA that is consistent with a given set of positive and negative examples [OG92] (see section 6.3 for more information). The algorithm maps the set of positive examples to a lattice of finite state automata and uses the information from the set of negative examples to conduct an ordered search through the lattice. The algorithm is guaranteed to return a DFA that is consistent with the given set of examples. Further, if the set of examples provided to the learner includes a *characteristic set* of examples (see section 3.3.7) as a subset then the algorithm is guaranteed to return a canonical representation of the target DFA. RPNI2 extends the RPNI algorithm to an incremental setting where the characteristic set of examples might not be available to the learner at the start [Dup96a]. The operation of RPNI2 is summarized as follows: The initially available set of positive examples is mapped to a lattice of FSA. An ordered search of the lattice is conducted using the initial set of negative examples. The learner maintains a current hypothesis that is consistent with all the examples

it has observed thus far. The lattice is incrementally extended when a new positive example becomes available. A new curent hypothesis is then found in the augmented search space. New negative examples require a modification of the current hypothesis to ensure that it is consistent with the example. Further, a consistency check is required to make sure that the revised hypothesis stays consistent with all previously seen negative examples. The algorithm runs in time that is polynomial in the sum of lengths of the positive and negative examples and is guaranteed to converge to the target DFA in the limit when the set of examples seen by the learner includes a characteristic set corresponding to the target DFA as a subset.

IID differs from Porat and Feldman's algorithm and RPNI2 mainly in the fact that *IID* assumes the availability of a knowledgeable teacher to answer membership queries whereas the latter two algorithms use only labeled examples. Unlike RPNI2, *IID* does not require the learner to store all the examples. Only those examples that are inconsistent with the current representation of the target are required to be stored by the learner. Unlike Porat and Feldman's algorithm, *IID* does not require any specific ordering of the labeled examples. Furthermore, the incremental modification of the learner's representation of the target DFA is guaranteed to be consistent with all the examples observed by the learner until then and no explicit consistency check is required.

The learner's reliance on the teacher to provide accurate responses to membership queries poses a potential limitation in applications where a reliable teacher is not available. We are exploring the possibility of learning in an environment where the learner does not have access to a teacher. The algorithms due to Dupont [Dup96a] and Porat & Feldman [PF91] operate in this framework. Some open problems include whether the limitations of these algorithms (e.g., the need to store all the examples, the requirement of complete lexicographic ordering of examples, etc.) can be overcome without sacrificing efficiency and guaranteed convergence to the target. Porat and Feldman proved a strong negative result stating that there exists no algorithm which operating with finite working storage can incrementally learn the target DFA from an arbitrary presentation. In this context, it is of interest to explore alternative models of learning that relax the convergence criterion (for example, allow approximate learning of the

target within a given error bound), provide for some additional hints to the learning algorithm (like a bound on the number of states of the target DFA), or include a helpful teacher that carefully guides the learner (perhaps by providing *simple* examples first).

6 LEARNING DFA FROM SIMPLE EXAMPLES

6.1 Introduction

As we have seen thus far, exact learning of the target DFA from an arbitrary set of labeled examples is a hard problem. This problem is made more tractable when the learner is provided with examples that satisfy certain properties such as *structural completeness* or *live completeness* and possibly is allowed access to a knowledgeable teacher capable of answering queries. In chapter 5 we addressed incremental methods for learning the target DFA which are extremely useful in scenarios where the entire training set might not be available to the learner at the start. Incremental algorithms are guaranteed to converge to the target DFA in the limit. However, these approaches have certain restrictions. RPNI2 requires the learner to store all the examples [Dup96a], Porat and Feldman's algorithm mandates a *complete ordered presentation* of the labeled examples [PF91], and the *IID* algorithm described in chapter 5 is based on the availability of a knowledgeable teacher to answer membership queries [PNH97].

It is thus natural to ask whether DFA can be learned approximately. Valiant's distribution-independent model of learning, also called the *probably approximately correct* (PAC) learning model [Val84], is a widely used framework for approximate learning of concept classes. PAC learning models natural learning in that it is fast (learning takes place in polynomial time) and it suffices to learn approximately [KV94]. When adapted to the problem of learning DFA, the goal of a PAC learning algorithm is to obtain in polynomial time, with high probability, a DFA that is approximately correct when compared to the target DFA. We define PAC learning of DFA more formally in section 6.2. Angluin's L^* algorithm [Ang87] that learns DFA in polynomial time using *membership* and *equivalence* queries can be recast under the PAC framework to learn by posing membership queries alone. However, the approximate learnability

of DFA from labeled examples alone remains a hard problem [PW89, KV89].

The PAC model's requirement of learnability under all conceivable distributions is often considered too stringent. Pitt's seminal paper identified the following open research problem: "*Are DFA's PAC-identifiable if examples are drawn from the uniform distribution, or some other known simple distribution?*" [Pit89]. Several efforts have been made to study the learnability of concept classes under restricted classes of distributions. Li and Vitányi proposed a model for PAC learning with *simple* examples called the *simple PAC* model wherein the class of distributions is restricted to *simple* distributions (see section 6.4). Denis *et al* proposed a model of learning from simple examples where a knowledgeable teacher might choose the examples based on the knowledge of the target concept [DDG96]. This model is known as the PACS learning model. In this chapter, we present a method for efficient PAC learning of DFA from simple examples thereby answering Pitt's open research question in the affirmative. More specifically, we will prove that the class of *simple* DFA (see section 6.4) is learnable under the simple PAC model and the entire class of DFA is learnable under the PACS model. Further, we demonstrate how the model of learning from simple examples naturally extends the model of *learning concepts from representative examples* [Gol78] and the *polynomial teachability* model [GM93] to a probabilistic framework.

This chapter is organized as follows: Section 6.2 briefly introduces some of the concepts that are used in the results described in this chapter. This includes a discussion of the PAC learning model, Kolmogorov complexity, and the universal distribution. Section 6.3 reviews the RPNI algorithm for learning DFA. Section 6.4 discusses the PAC learnability of the class of *simple* DFA under the simple PAC learning model. Section 6.5 demonstrates the PAC learnability of the entire class of DFA under the PACS learning model. Section 6.6 analyzes the PACS model in relation with other models for concept learning. Section 6.7 concludes with a summary of our contributions and discussion of several intersting directions for future research.

6.2 Preliminaries

In this section we present a brief overview of some of the important concepts that are used in the results described later in this chapter. Specifically, we discuss PAC learning in the context of learning DFA, Kolmogorov complexity, and the universal distribution.

6.2.1 PAC Learning of DFA

Let $\mathcal{X}$ denote the *sample space* defined as the set of all strings Σ^*. Let $x \subseteq \mathcal{X}$ denote a *concept*. For our purpose, x is a *regular language*. We identify the concept with the corresponding DFA and denote the class of all DFA as the *concept class* $\mathcal{C}$. The *representation* $\mathcal{R}$ that assigns a name to each DFA in $\mathcal{C}$ is defined as a function $\mathcal{R} : \mathcal{C} \longrightarrow \{0,1\}^*$. $\mathcal{R}$ is the set of standard encodings of the DFA in $\mathcal{C}$ (see section 3.3.1). Assume that there is an unknown and arbitrary but fixed distribution $\mathcal{D}$ according to which the examples of the target concept are drawn. In the context of learning DFA, $\mathcal{D}$ is restricted to a probability distribution on strings of Σ^* of length at most m.

Definition 6.1 *(due to [Pit89])*
DFAs are PAC-identifiable iff *there exists a (possibly randomized) algorithm $\mathcal{A}$ such that on input of any parameters ϵ and δ, for any DFA M of size N, for any number m, and for any probability distribution $\mathcal{D}$ on strings of Σ^* of length at most m, if $\mathcal{A}$ obtains labeled examples of M generated according to the distribution $\mathcal{D}$, then $\mathcal{A}$ produces a DFA M' such that with probability at least $1 - \delta$, the probability (with respect to distribution $\mathcal{D}$) of the set $\{\alpha \mid \alpha \in L(M) \oplus L(M')\}$ is at most ϵ. The run time of $\mathcal{A}$ (and hence the number of randomly generated examples obtained by $\mathcal{A}$) is required to be polynomial in N, m, $1/\epsilon$, $1/\delta$, and $|\Sigma|$.*

If the learning algorithm $\mathcal{A}$ produces a DFA M' such that with probability at least $1 - \delta$, M' is equivalent to M i.e., the probability (with respect to distribution $\mathcal{D}$) of the set $\{\alpha \mid \alpha \in L(M) \oplus L(M')\}$ is exactly 0 then $\mathcal{A}$ is said to be a *probably exact* learning algorithm for the class of DFA and the class of DFA is said to be *probably exactly learnable* by the algorithm $\mathcal{A}$.

6.2.2 Kolmogorov Complexity

Note that the definition of PAC learning requires that the concept class (in this case the class of DFA) must be learnable under any arbitrary (but fixed) probability distribution. This requirement is often considered too stringent in practical learning scenarios where it is not unreasonable to assume that a learner is first provided with *simple* and *representative* examples of the target concept. Intuitively, when we teach a child the rules of *multiplication* we are more likely to first give simple examples like 3×4 than examples like 1377×428. A *representative set* of examples is one that would enable the learner to identify the target concept exactly. For example, the characteristic set of a DFA would constitute a suitable representative set. The question now is whether we can formalize what simple examples mean. *Kolmogorov complexity* provides a machine independent notion of *simplicity* of objects. Intuitively, the Kolmogorov complexity of an object (represented by a binary string α) is the length of the shortest binary program that computes α. Objects that have regularity in their structure (i.e., objects that can be easily compressed) have low Kolmogorov complexity. For example, consider the string $s_1 = 010101...01 = (01)^{500}$. On a particular machine M, a program to compute this string would be "*Print 01 500 times*". On the other hand consider a totally random string $s_2 = 110011010 \ldots 00111$ where $|s_2| = 500$. Unlike s_1, it is not possible to compress the string s_2 which means that a program to compute s_2 on M would be "*Print 1100111010000 ... 00111*" i.e., the program would have to explicitly specify the string s_2. The length of the program that computes s_1 is shorter than that of the program that computes s_2. Thus, we could argue that s_1 has lower Kolmogorov complexity than s_2 with respect to the machine M.

We will consider the *prefix* version of the Kolmogorov complexity that is measured with respect to prefix Turing machines and denoted by K. Consider a prefix Turing machine that implements the partial recursive function $\phi : \{0,1\}^* \stackrel{partial}{\longrightarrow} \{0,1\}^*$. For any string $\alpha \in \{0,1\}^*$, the Kolmogorov complexity of α relative to ϕ is defined as $K_\phi(\alpha) = min\{|\pi| \mid \phi(\pi) = \alpha\}$ where $\pi \in \{0,1\}^*$ is a program input to the Turing machine. Prefix Turing machines can be effectively enumerated and there exists a *Universal Turing Machine* (U) capable of simulating every prefix

Turing machine. Assume that the universal Turing machine implements the partial function ψ. The *Optimality Theorem* for Kolmogorov Complexity guarantees that for any prefix Turing machine ϕ there exists a constant c_ϕ such that for any string α, $K_\psi(\alpha) \leq K_\phi(\alpha) + c_\phi$. Note that we use the name of the Turing Machine (say M) and the partial function it implements (say ϕ) interchangeably i.e., $K_\phi(\alpha) = K_M(\alpha)$. Further, by the *Invariance Theorem* it can be shown that for any two universal Turing machines ψ_1 and ψ_2 there is a constant $\eta \in \mathcal{N}$ (where $\mathcal{N}$ is the set of natural numbers) such that for all strings α, $|K_{\psi_1}(\alpha) - K_{\psi_2}(\alpha)| \leq \eta$. Thus, we can fix a single universal Turing machine U and denote $K(\alpha) = K_U(\alpha)$. Note that there exists a Turing machine that computes the identity function $\chi : \{0,1\}^* \longrightarrow \{0,1\}^*$ where $\chi(\alpha) = \alpha \; \forall \alpha$. Thus, it can be shown that the Kolmogorov complexity of an object is bounded by its length i.e., $K(\alpha) \leq |\alpha| + K(|\alpha|) + \eta$ where η is a constant independent of α.

Suppose that some additional information in the form of a string β is available to the Turing machine ϕ. The conditional Kolmogorov complexity of any object α given β is defined as $K_\phi(\alpha \mid \beta) = min\{|\pi| \mid \phi(\langle \pi, \beta \rangle) = \alpha\}$ where $\pi \in \{0,1\}^*$ is a program and $\langle x, y \rangle$ is a standard pairing function[1]. Note that the conditional Kolmogorov complexity does not charge for the extra information β that is available to ϕ along with the program π. Fixing a single universal Turing machine U we denote the conditional Kolmogorov complexity of α by $K(\alpha|\beta) = K_U(\alpha|\beta)$. It can be shown that $K(\alpha|\beta) \leq K(\alpha) + \eta$ where η is a constant independent of α.

6.2.3 Universal Distribution

The set of programs for a string α relative to a Turing machine M is defined as $PROG_M(\alpha) = \{\pi \mid M(\pi) = \alpha\}$. The algorithmic probability of α relative to M is defined as $\mathbf{m}_M(\alpha) = \Pr(PROG_M)$. The algorithmic probability of α with respect to the universal Turing machine U is denoted as $\mathbf{m}_U(\alpha) = \mathbf{m}(\alpha)$. $\mathbf{m}$ is known as the Solomonoff-Levin distribution. It is the universal enumerable probability distribution, in that, it multiplicatively dominates all enumerable probability distributions. Thus, for any enumerable probability distribution

[1] Define $\langle x, y \rangle = bd(x)01y$ where bd is the bit doubling function defined as $bd(0) = 00$, $bd(1) = 11$, and $bd(ax) = aabd(x), a \in \{0,1\}$.

P there is a constant $c \in \mathcal{N}$ such that for all strings α, $c\,\mathbf{m}(\alpha) \geq P(\alpha)$. The *Coding Theorem* due independently to Schnorr, Levin, and Chaitin [LV93, LV97] states that $\exists \eta \in \mathcal{N}$ such that $\forall \alpha$ $\mathbf{m}_M(\alpha) \leq 2^{\eta - K(\alpha)}$. Intuitively this means that if there are several programs for a string α on some machine M then there is a short program for α on the universal Turning machine (i.e., α has a low Kolmogorov complexity). By optimality of $\mathbf{m}$ it can be shown that: $\exists \eta \in \mathcal{N}$, such that $\forall \alpha \in \{0,1\}^*$, $2^{-K(\alpha)} \leq \mathbf{m}(\alpha) \leq 2^{\eta - K(\alpha)}$. We see that the universal distribution $\mathbf{m}$ assigns higher probability to simple objects (objects with low Kolmogorov complexity). Given a string $r \in \Sigma^*$, the universal distribution based on the knowledge of r, $\mathbf{m}_r$, is defined as is defined as $\mathbf{m}_r(\alpha) = \lambda_r 2^{-K(\alpha|r)}$ where λ_r is a constant such that $\lambda_r \sum_{\alpha \in \Sigma^*} 2^{-K(\alpha|r)} = 1$ (i.e., $\lambda_r \geq 1$) [DDG96]. Further, $\mathbf{m}_r$ is such that $2^{-K(\alpha|r)} \leq \mathbf{m}_r(\alpha) \leq 2^{\eta - K(\alpha|r)}$ where η is a constant.

The interested reader is referred to [LV93, LV97] for a thorough treatment of Kolmogorov complexity, universal distribution, and related topics.

6.3 The RPNI Algorithm

The *regular positive and negative inference* (RPNI) algorithm [OG92] is a polynomial time algorithm for identification of a DFA consistent with a given set $S = S^+ \cup S^-$. Further, if the sample is a characteristic set for the target DFA then the algorithm is guaranteed to return a canonical representation of the target DFA. Our description of the RPNI algorithm is based on the explanation given in [Dup96a].

A labeled sample $S = S^+ \cup S^-$ is provided as input to the algorithm. It constructs a prefix tree automaton $PTA(S^+)$ and numbers its states in the standard order (see section 3.3.2). Then it performs an ordered search in the space of partitions of the set of states of $PTA(S^+)$ under the control of the set of negative examples S^-. The partition, π_0, corresponding to the automaton $PTA(S^+)$ itself is $\{\{0\},\{1\},\ldots,\{\overline{N}-1\}\}$ where $\overline{N}$ is the number of states of the PTA. Note that $\overline{N} \leq ||S^+||$ where $||S^+||$ is the sum of the lengths of the strings in S^+. $M_{\pi_0} = PTA(S^+)$ is consistent with all the training examples and is treated as the initial hypothesis. The current hypothesis is is denoted by M_π and the corresponding partition is

denoted by π.

The algorithm is outlined in Fig. 6.1. The nested *for* loop refines the partition π by merging the states of $PTA(S^+)$ in order. At each step, a partition $\tilde{\pi}$ is obtained from the partition π by merging the two blocks that contain the states i and j respectively. The function *derive* obtains the quotient automaton $M_{\tilde{\pi}}$, corresponding to the partition $\tilde{\pi}$. $M_{\tilde{\pi}}$ might be a NFA in which case the function *deterministic_merge* determinizes it by recursively merging the states that cause non-determinism. For example, if q_i, q_j, and q_k are states of $M_{\tilde{\pi}}$ such that for some symbol $a \in \Sigma$, $\delta(q_i, a) = \{q_j, q_k\}$ then the states q_j and q_k are merged together. This recursive merging of states can go on for at most $\overline{N} - 1$ steps and the resulting automaton $M_{\hat{\pi}}$ is guaranteed to be a DFA [Dup96a]. Note that since $\tilde{\pi} \ll \hat{\pi}$ we know by the grammar covers relation that if $M_{\tilde{\pi}}$ accepts a negative example in S^- then so would $M_{\hat{\pi}}$. The function, $consistent(M_{\hat{\pi}}, S^-)$ returns *True* if $M_{\hat{\pi}}$ is consistent with all examples in S^- and *False* otherwise. If a partition $\hat{\pi}$ is found such that the corresponding DFA $M_{\hat{\pi}}$ is consistent with S^- then $M_{\hat{\pi}}$ replaces M_{π} as the current hypothesis.

Let $||S^+||$ and $||S^-||$ denote the sums of the lengths of examples in S^+ and S^- respectively. $PTA(S^+)$ has $O(||S^+||)$ states. The nested *for* loop of the algorithm performs $O(||S^+||^2)$ state merges. Further, each time two blocks of the partition π are merged, the routine *deterministic_merge* in the worst case would cause $O(||S^+||)$ state mergings and the function *consistent* that checks for the consistency of the derived DFA with the negative examples would incur a cost of $O(||S^-||)$. Hence the time complexity of the RPNI algorithm is $O((||S^+|| + ||S^-||) \cdot ||S^+||^2)$.

Example

We demonstrate the execution of the RPNI algorithm on the task of learning the DFA in Fig. 3.1. Note that for convenience we have shown the target DFA in Fig. 6.2 without the dead state d_0 and its associated transitions. Assume that a sample $S = S^+ \cup S^-$ where $S^+ = \{b, aa, aaaa\}$ and $S^- = \{\lambda, a, aaa, baa\}$. It can be easily verified that S is a characteristic sample for the target DFA (see section 3.3.7). The FSA $M = PTA(S^+)$ is depicted in Fig. 6.3

Algorithm RPNI

Input: A sample $S = S^+ \cup S^-$
Output: A DFA compatible with S

begin
  // Initialization
  $\pi = \pi_0 = \{\{0\}, \{1\}, \ldots, \{\overline{N} - 1\}\}$
  $M_\pi = PTA(S^+)$
  // Perform state merging
  for i = 1 **to** $\overline{N} - 1$
    for j = 0 **to** $i - 1$
      // Merge the block of π containing state i with the block containing state j
      $\tilde{\pi} = \pi \backslash \{B(i,\pi), B(j,\pi)\} \cup \{B(i,\pi) \cup B(j,\pi)\}$
      // Obtain the quotient automaton $M_{\tilde{\pi}}$
      $M_{\tilde{\pi}} = derive(M, \tilde{\pi})$
      // Determinize the quotient automaton (if necessary) by state merging
      $\hat{\pi} = deterministic\_merge(M_{\tilde{\pi}})$
      // Does $M_{\hat{\pi}}$ reject all strings in S^-?
      if $consistent(M_{\hat{\pi}}, S^-)$
      then
        // Treat $M_{\hat{\pi}}$ as the current hypothesis
        $M_\pi = M_{\hat{\pi}}$
        $\pi = \hat{\pi}$
        break
      end if
    end for
  end for
  return M_π
end

Figure 6.1 RPNI Algorithm.

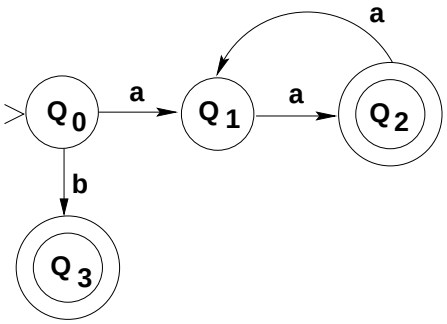

Figure 6.2 Target DFA A.

where the states are numbered in the standard order. The initial partition is $\pi = \pi_0 = \{\{0\},\{1\},\{2\},\{3\},\{4\},\{5\}\}$.

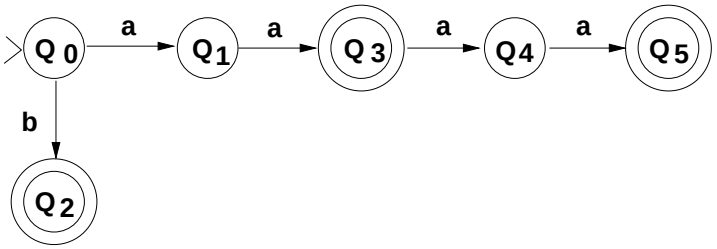

Figure 6.3 Prefix Tree Automaton.

The algorithm attempts to merge the blocks containing states 1 and 0 of the partition π. The quotient FSA $M_{\tilde{\pi}}$ and the FSA $M_{\hat{\pi}}$ obtained after invoking *deterministic_merge* are shown in Fig. 6.4. The DFA $M_{\hat{\pi}}$ accepts the negative example $\lambda \in S^-$. Thus, the current partition π remains unchanged.

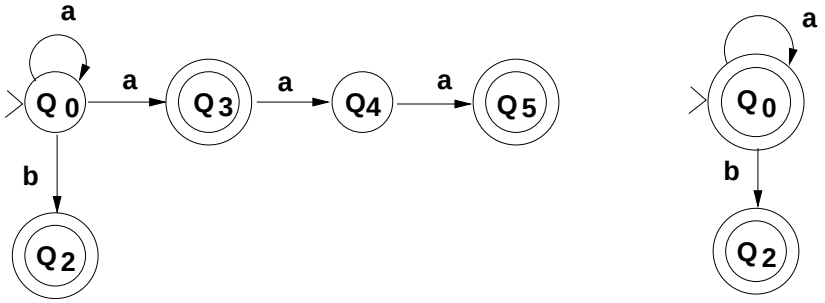

Figure 6.4 $M_{\tilde{\pi}}$ Obtained by Fusing Blocks Containing the States 1 and 0 of π and the Corresponding $M_{\hat{\pi}}$.

Next the algorithm merges the blocks containing states 2 and 0 of the partition π. The quotient FSA $M_{\tilde{\pi}}$ is depicted in Fig. 6.5. Since $M_{\tilde{\pi}}$ is a DFA, the procedure *deterministic_merge* returns the same automaton i.e., $M_{\hat{\pi}} = M_{\tilde{\pi}}$. $M_{\hat{\pi}}$ accepts the negative example $\lambda \in S^-$ and hence the partition π remains unchanged.

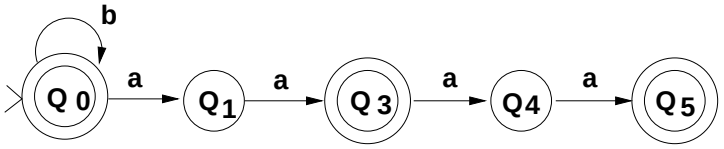

Figure 6.5 $M_{\tilde{\pi}}$ (same as $M_{\hat{\pi}}$) Obtained by Fusing Blocks Containing the States 2 and 0 of π.

Table 6.1 lists the different partitions $\tilde{\pi}$ obtained by fusing the blocks of π_0, the partitions $\hat{\pi}$ obtained by *deterministic_merge* of $\tilde{\pi}$, and the negative example (belonging to S^-), if any, that is accepted by the quotient FSA $M_{\tilde{\pi}}$. The partitions marked $*$ denote the partition π for which M_π is consistent with all examples in S^- and hence is the current hypothesis. It is easy to see that the DFA corresponding to the partition $\pi = \{\{0\},\{1,4\},\{2\},\{3,5\}\}$ is exactly the target DFA we are trying to learn (Fig. 6.2).

Table 6.1 Sample Run of the RPNI Algorithm.

Partition $\tilde{\pi}$	**Partition $\hat{\pi}$**	**Negative Example**
$\{\{0,1\},\{2\},\{3\},\{4\},\{5\}\}$	$\{\{0,1,3,4,5\},\{2\}\}$	a
$\{\{0,2\},\{1\},\{3\},\{4\},\{5\}\}$	$\{\{0,2\},\{1\},\{3\},\{4\},\{5\}\}$	λ
$\{\{0\},\{1,2\},\{3\},\{4\},\{5\}\}$	$\{\{0\},\{1,2\},\{3\},\{4\},\{5\}\}$	a
$\{\{0,3\},\{1\},\{2\},\{4\},\{5\}\}$	$\{\{0,3\},\{1,4\},\{2\},\{5\}\}$	λ
$\{\{0\},\{1,3\},\{2\},\{4\},\{5\}\}$	$\{\{0\},\{1,3,4,5\},\{2\}\}$	a
$\{\{0\},\{1\},\{2,3\},\{4\},\{5\}\}$	$\{\{0\},\{1\},\{2,3\},\{4\},\{5\}\}$	baa
$\{\{0,4\},\{1\},\{2\},\{3\},\{5\}\}$	$\{\{0,4\},\{1,5\},\{2\},\{3\}\}$	a
$\{\{0\},\{1,4\},\{2\},\{3\},\{5\}\}$	$\{\{0\},\{1,4\},\{2\},\{3,5\}\}^*$	—
$\{\{0,3,5\},\{1,4\},\{2\}\}$	$\{\{0,3,5\},\{1,4\},\{2\}\}$	λ
$\{\{0\},\{1,3,4,5\},\{2\}\}$	$\{\{0\},\{1,3,4,5\},\{2\}\}$	a
$\{\{0\},\{1,4\},\{2,3,5\}\}$	$\{\{0\},\{1,4\},\{2,3,5\}\}$	baa
$\{\{0\},\{1,4\},\{2\},\{3,5\}\}$	$\{\{0\},\{1,4\},\{2\},\{3,5\}\}^*$	—
$\{\{0\},\{1,3,4,5\},\{2\}\}$	$\{\{0\},\{1,3,4,5\},\{2\}\}$	a

6.4 Learning Simple DFA under the Simple PAC model

Li and Vitányi proposed the simple PAC learning model where the class of probability distributions is restricted to *simple* distributions [LV91]. A distribution is simple if it is multiplicatively dominated by some enumerable distribution. All computable distributions including the distributions that we commonly use in statistics such as the *uniform distribution*, *normal distribution*, *geometric distribution*, and *Poisson distribution* are simple. Simple distributions thus include a broad range of distributions. Further, the *simple distribution independent learning theorem* due to Li and Vitanyí says that that a concept class is learnable under universal distribution **m** *iff* it is learnable under the entire class of *simple distributions* provided the training examples are drawn according to the universal distribution [LV91]. Thus, the simple PAC learning model is sufficiently general. Concept classes such as $\log n$*-term DNF* and *simple k-reversible DFA* are learnable under the simple PAC model whereas their PAC learnability in the standard sense is unknown [LV91]. We show that the class of *simple* DFA is polynomially learnable under the simple PAC learning model.

A DFA with low Kolmogorov complexity is called a simple DFA. More specifically, a DFA A with N states and a standard encoding (or canonical representation) r is simple if $K(A) = O(\lg N)$. For example, a DFA which accepts all strings of length N is a simple DFA. Note however that this DFA contains a path for every string of length N and hence it has a path of Kolmogorov complexity N. In general, simple DFA might actually have very random paths. We saw in section 6.2.2 that a natural learning scenario would typically involve learning from a *simple* and *representative* set of examples for the target concept. We adopt Kolmogorov complexity as a measure of simplicity and define simple examples as those with low Kolmogorov complexity i.e., with Kolmogorov complexity $O(\lg N)$. Further, a characteristic set for the DFA A can be treated as its representative set.

We demonstrate that for every simple DFA there exists a characteristic set of simple examples S_c.

Lemma 6.1 *For any N state simple DFA (with Kolmogorov complexity $O(\lg N)$) there exists a characteristic set of simple examples S_c such that the length of each string in this set is at*

most $2N - 1$.

Proof:

Consider the following enumeration of a characteristic set of examples for a DFA $A = (Q, \delta, \Sigma, q_0, F)$ with N states[2].

1. Fix an enumeration of the shortest paths (in standard order) from the state q_0 to each state in Q except the dead state. This is the set of short prefixes of A. There are at most N such paths and each path is of length at most $N - 1$.

2. Fix an enumeration of paths that includes each path identified above and its extension by each letter of the alphabet Σ. From the paths just enumerated retain only those that do not lead to a dead state of A. This represents the kernel of A. There are at most $N(|\Sigma| + 1)$ such paths and each path is of length at most N.

3. Let the characteristic set be denoted by $S_c = S_c^+ \cup S_c^-$.

   (a) For each string α identified in step 2 above, determine the first suffix β in the standard enumeration of strings such that $\alpha\beta \in L(A)$. Since $|\alpha| \leq N$, and β is the shortest suffix in the standard order it is clear that $|\alpha\beta| \leq 2N - 1$. Each such $\alpha\beta$ is a member of S_c^+.

   (b) For each pair of strings (α, β) in order where α is a string identified in step 1, β is a string identified in step 2, and α and β lead to different states of A, determine the first suffix γ in the standard enumeration of strings such that $\alpha\gamma \in L(A)$ and $\beta\gamma \notin L(A)$ or vice versa. Since $|\alpha| \leq N - 1$, $|\beta| \leq N$, and γ is the shortest distinguishing suffix for the states represented by α and β it is clear that $|\alpha\gamma|, |\beta\gamma| \leq 2N - 1$. The accepted string from among $\alpha\gamma$ and $\beta\gamma$ is a member of S_c^+ and the rejected string is a member of S_c^-.

   Trivial upper bounds on the sizes of S_c^+ and S_c^- are $|S_c^+| \leq N^2(|\Sigma| + 1) + N(|\Sigma|)$, $|S_c^-| \leq N^2(|\Sigma| + 1) - N$. Thus, $|S_c| = O(N^2)$. Further, the length of each string in S_c is less than $2N - 1$.

[2]This enumeration strategy applies to any DFA and is not restricted to simple DFA alone.

The strings in S_c can be ordered in some way such that individual strings are identified by an index of length at most $\lg(3|\Sigma|N^2)$ bits. There exists a Turing machine M that implements the above algorithm for constructing the set S_c. M can take as input an encoding of a simple DFA of length $O(\lg N)$ bits and an index of length $\lg(3|\Sigma|N^2)$ bits and output the corresponding string α belonging to S_c. Thus, $\forall \alpha \in S_c$,

$$\begin{aligned} K(\alpha) &\leq k_1 \lg N + \lg(3|\Sigma|N^2) \\ K(\alpha) &\leq k_1 \lg N + k_2 \lg N \\ &= O(\lg N) \end{aligned} \tag{6.1}$$

This proves the lemma. □

Lemma 6.2 *Suppose a sample S is drawn according to $\mathbf{m}$. For $0 < \delta \leq 1$, if $|S| = O(N^k \lg(\frac{1}{\delta}))$ then with probability greater than $1 - \delta$, $S_c \subseteq S$ where k is a constant.*

Proof:

From lemma 6.1 we know that $\forall \alpha \in S_c$, $K(\alpha) = O(\lg N)$. Further, $|S_c| = O(N^2)$. By definition, $\mathbf{m}(\alpha) \geq 2^{-K(\alpha)}$. Thus, $\mathbf{m}(\alpha) \geq 2^{-k_1 \lg N}$ or equivalently $\mathbf{m}(\alpha) \geq N^{-k_1}$ where k_1 is a constant.

$$\begin{aligned} \Pr(\alpha \in S_c \text{ is not sampled in one random draw}) &\leq (1 - N^{-k_1}) \\ \Pr(\alpha \in S_c \text{ is not sampled in } |S| \text{ random draws}) &\leq (1 - N^{-k_1})^{|S|} \\ \Pr(\text{ some } \alpha \in S_c \text{ is not sampled in } |S| \text{ random draws}) &\leq |S_c|(1 - N^{-k_1})^{|S|} \\ &\leq k_2 N^2 (1 - N^{-k_1})^{|S|} \\ &\quad \text{since } |S_c| = O(N^2) \\ \Pr(S_c \not\subseteq S) &\leq k_2 N^2 (1 - N^{-k_1})^{|S|} \end{aligned}$$

We want this probability to be less than δ.

$$\begin{aligned} k_2 N^2 (1 - N^{-k_1})^{|S|} &\leq \delta \\ k_2 N^2 (e^{-N^{-k_1}})^{|S|} &\leq \delta \text{ since } 1 - x \leq e^{-x} \end{aligned}$$

$$\begin{aligned}
\ln(k_2) + \ln(N^2) - N^{-k_1}|S| &\leq \ln(\delta) \\
|S| &\geq N^{k_1}(\ln(\frac{1}{\delta}) + \ln(k_2) + \ln(N^2)) \\
&= O(N^{k_1} \lg(\frac{1}{\delta})) \\
&= O(N^k \lg(\frac{1}{\delta})) \text{ where } k \text{ replaces } k_1
\end{aligned}$$

Thus, $\Pr(S_c \subseteq S) \geq 1 - \delta$. □

We now prove that the class of simple DFA is polynomially learnable under $\mathbf{m}$.

Theorem 6.1 *For all N, the class $\mathcal{C}^{\leq N}$ of simple DFA whose canonical representations have at most N states is probably exactly learnable under the simple PAC model.*

Proof:

Let A be a simple DFA with at most N states. Let S_c be a characteristic sample of A enumerated as described in lemma 6.1 above. Recall, that the examples in S_c are simple (i.e., each example has Kolmogorov complexity $O(\lg N)$). Now consider the algorithm $\mathcal{A}_1$ in Fig. 6.6 that draws a sample S with the following properties.

1. $S = S^+ \cup S^-$ is a set of positive and negative examples corresponding to the target DFA A.

2. The examples in S are drawn at random according to the distribution $\mathbf{m}$.

3. $|S| = O(N^k \lg(\frac{1}{\delta}))$.

Lemma 6.1 showed that for every simple DFA A there exists a characteristic set of simple examples S_c where the length of each example in S_c is $< 2N - 1$. Lemma 6.2 showed that if a labeled sample S of size $O(N^k \lg(\frac{1}{\delta}))$ is randomly drawn according to $\mathbf{m}$ then with probability greater than $1 - \delta$, $S_c \subseteq S$. The RPNI algorithm is guaranteed to return a canonical representation of the target DFA A if the set of examples S provided is a superset of a characteristic set S_c. Since the size of S is polynomial in N and $1/\delta$ and the length of each string in S is restricted to $2N - 1$, the RPNI algorithm, and thus the algorithm $\mathcal{A}_1$ can be implemented to run in time polynomial in N and $1/\delta$. Thus, with probability greater than $1 - \delta$, $\mathcal{A}_1$ is

Algorithm $\mathcal{A}_1$

Input: $N, 0 < \delta \leq 1$
Output: A DFA M

begin
Randomly draw a labeled sample S according to $\mathbf{m}$
Retain only those examples in S that have length at most $2N - 1$
$M = RPNI(S)$
return M
end

Figure 6.6 A Probably Exact Algorithm for Learning Simple DFA.

guaranteed to return a canonical representation of the target DFA A. This proves that the class $\mathcal{C}^{\leq N}$ of simple DFA whose canonical representations have at most N states is exactly learnable with probability greater than $1 - \delta$ under the simple PAC learning model. □

6.5 Learning DFA under the PACS model

In section 6.4 we proved that the class of simple DFA is learnable under the simple PAC model where the underlying distribution is restricted to the universal distribution $\mathbf{m}$. Denis *et al* proposed the PACS learning model for learning from simple examples where a teacher might use knowledge of the target concept in selecting representative examples [DDG96]. Under this model, examples with low conditional Kolmogorov complexity given a representation r of the target concept are called simple examples. Specifically, for a concept with representation r, the set $S^r_{sim} = \{\alpha \mid K(\alpha|r) \leq \mu lg(|r|)\}$ (where μ is a constant) is the set of simple examples for the concept. Further, $S^r_{sim,rep}$ is used to denote a set of simple and representative examples of r. The PACS model restricts the underlying distribution to $\mathbf{m}_r$. Formally, the probability of drawing an example α for a target concept with representation r is given as $\mathbf{m}_r(\alpha) = \lambda_r 2^{-K(\alpha|r)}$ where λ_r is a constant. Representative examples for the target concept are those that enable the learner to exactly learn the target. As explained earlier, the characteristic set corresponding to a DFA can be treated as a representative set for the DFA. The Occam's Razor theorem proved by Denis *et al* states that if there exists a representative set of simple examples for each

concept in a concept class then the concept class is PACS learnable [DDG96].

We now demonstrate that the class of DFA is efficiently learnable under the PACS model. Lemma 6.3 proves that for any DFA A with standard encoding r there exists a characteristic set of simple examples $S^r_{sim,rep}$.

Lemma 6.3 *For any N state DFA with standard encoding r ($|r| = \mathcal{O}(N \lg(N))$), there exists a characteristic set of simple examples (denoted by $S^r_{sim,rep}$) such that each string of this set is of length at most $2N - 1$.*

Proof:

Given a DFA $A = (Q, \delta, \Sigma, q_0, F)$, it is possible to enumerate a characteristic set of examples S_c for A as described in lemma 6.1 such that $|S_c| = O(N^2)$ and each example of S_c is of length at most $2N - 1$. Individual strings in S_c can be identified by specifying an index of length at most $\lg(3|\Sigma|N^2)$ bits. There exists a Turing machine M that implements the above algorithm for constructing the set S_c. Given the knowledge of the target concept r, M can take as input an index of length $\lg(3|\Sigma|N^2)$ bits and output the corresponding string belonging to S_c. Thus $\forall \alpha \in S_c$,

$$\begin{aligned} K(\alpha|r) &\leq \lg(3|\Sigma|N^2) \\ &\leq \mu \lg(|r|) \text{ where } \mu \text{ is a constant} \end{aligned}$$

We define the set S_c to be the characteristic set of simple examples $S^r_{sim,rep}$ for the DFA A. This proves the lemma. □

Lemma 6.4 *(Due to [DDG96])*

Suppose that a sample S is drawn according to $\mathbf{m}_r$. For an integer $l \geq |r|$, and $0 < \delta \leq 1$, if $|S| = O(l^\mu \lg(\frac{1}{\delta}))$ then with probability greater than $1 - \delta$, $S^r_{sim} \subseteq S$.

Proof:

Claim 6.1 $\forall \alpha \in S^r_{sim},\ \mathbf{m}_r(\alpha) \geq l^{-\mu}$

$$\begin{aligned}
\mathbf{m}_r(\alpha) &\geq 2^{-K(\alpha|r)} \\
&\geq 2^{-\mu \lg |r|} \\
&\geq |r|^{-\mu} \\
&\geq l^{-\mu}
\end{aligned}$$

Claim 6.2 $|S^r_{sim}| \leq 2l^\mu$

$$\begin{aligned}
|S^r_{sim}| &\leq |\{\alpha \in \{0,1\}^* \mid K(\alpha|r) \leq \mu \lg(|r|)\}| \\
&\leq |\{\alpha \in \{0,1\}^* \mid K(\alpha|r) \leq \mu \lg(l)\}| \\
&\leq |\{\beta \in \{0,1\}^* \mid |\beta| \leq \mu \lg(l)\}| \\
&\leq 2^{\mu \lg(l)+1} \\
&\leq 2l^\mu
\end{aligned}$$

Claim 6.3 *If* $|S| = O(l^\mu \lg(\frac{1}{\delta}))$ *then* $\Pr(S^r_{sim} \subseteq S) \geq 1 - \delta$

$$\begin{aligned}
\Pr(\alpha \in S^r_{sim} \text{ is not sampled in one random draw}) &\leq (1 - l^{-\mu}) \\
&\quad \text{(claim 6.1)} \\
\Pr(\alpha \in S^r_{sim} \text{ is not sampled in } |S| \text{ random draws}) &\leq (1 - l^{-\mu})^{|S|} \\
\Pr(\text{some } \alpha \in S^r_{sim} \text{ is not sampled in } |S| \text{ random draws}) &\leq 2l^\mu (1 - l^{-\mu})^{|S|} \\
&\quad \text{(claim 6.2)} \\
\Pr(S^r_{sim} \not\subseteq S) &\leq 2l^\mu (1 - l^{-\mu})^{|S|}
\end{aligned}$$

We would like this probability to be less than δ.

$$\begin{aligned}
2l^\mu (1 - l^{-\mu})^{|S|} &\leq \delta \\
2l^\mu (e^{-l^{-\mu}})^{|S|} &\leq \delta, \quad \text{since } 1 - x \leq e^{-x} \\
\ln(2) + \ln(l^\mu) - |S| l^{-\mu} &\leq \ln(\delta) \\
|S| &\geq l^\mu \left(\ln(2) + \ln(l^\mu) + \ln(1/\delta)\right) \\
|S| &\geq O(l^\mu \lg(1/\delta))
\end{aligned}$$

Thus, $\Pr(S^r_{sim} \subseteq S) \geq 1 - \delta$ □

Corollary 6.1 *Suppose that a sample S is drawn according to $\mathbf{m}_r$. For an integer $l \geq |r|$, and $0 < \delta \leq 1$, if $|S| = O(l^\mu \lg(1/\delta))$ then with probability greater than $1 - \delta$, $S^r_{sim,rep} \subseteq S$.*

Proof:

Follows immediately from Lemma 6.3 since $S^r_{sim,rep} \subseteq S^r_{sim}$. □

We now prove that the class of DFA is polynomially learnable under $\mathbf{m}_r$.

Theorem 6.2 *For all N, the class $\mathcal{C}^{\leq N}$ of DFA whose canonical representations have at most N states is probably exactly learnable under the PACS model.*

Proof:

Let A be a canonical DFA with at most N states and r be its standard encoding. We define the simple representative sample $S^r_{sim,rep}$ to be the characteristic sample of A enumerated as described in lemma 6.3 above. Recall that the length of each example in $S^r_{sim,rep}$ is at most $2N - 1$. Now consider the algorithm $\mathcal{A}_2$ that draws a sample S with the following properties

1. $S = S^+ \cup S^-$ is a set of positive and negative examples corresponding to the target DFA A
2. The examples in S are drawn at random according to the distribution $\mathbf{m}_r$
3. $|S| = O(l^\mu \lg(\frac{1}{\delta}))$

Lemma 6.3 showed that for every DFA A there exists a characteristic set of simple examples $S^r_{sim,rep}$ such that the length of each example in $S^r_{sim,rep}$ is $\leq 2N - 1$. Corollary 6.1 showed that if a labeled sample S of size $O(l^\mu \lg(\frac{1}{\delta}))$ is randomly drawn according to $\mathbf{m}_r$ then with probability greater than $1 - \delta$, $S^r_{sim,rep} \subseteq S$. The RPNI algorithm is guaranteed to return a canonical representation of the target DFA A if the set of examples S is a superset of a characteristic set for A. Since the size of S is polynomial in N and $1/\delta$ and the length of each string in S is restricted to $2N - 1$, the RPNI algorithm, and thus the algorithm $\mathcal{A}_2$ can be implemented to run in time polynomial in N and $1/\delta$. Thus, with probability greater than

Algorithm $\mathcal{A}_2$

Input: $N, 0 < \delta \leq 1$
Output: A DFA M

begin
Randomly draw a labeled sample S according to $\mathbf{m}_r$
Retain only those examples in S that have length at most $2N-1$
$M = RPNI(S)$
return(M)
end

Figure 6.7 A Probably Exact Algorithm for Learning DFA.

$1-\delta$, $\mathcal{A}_2$ is guaranteed to return a canonical DFA equivalent to the target A. This proves that the class $\mathcal{C}^{\leq N}$ of DFA whose canonical representations have at most N states is exactly learnable with probability greater than $1-\delta$. □

Since the number of states of the target DFA (N) might not be known in advance we present a PAC learning algorithm $\mathcal{A}_3$ that iterates over successively larger guesses of N. At each step the algorithm draws a random sample according to $\mathbf{m}_r$, applies the RPNI algorithm to construct a DFA, and tests the DFA using a randomly drawn test sample. If the DFA is consistent with the test sample then the algorithm outputs the DFA and halts. Otherwise the algorithm continues with the next guess for N.

Theorem 6.3 *The concept class $\mathcal{C}$ of DFA is learnable in polynomial time under the PACS model.*

Proof: Fig. 6.8 shows the PAC learning algorithm for DFA.

In algorithm $\mathcal{A}_3$ the polynomial p is defined such that a sample S of size $p(N, \frac{1}{\delta})$ contains the characteristic set of simple examples $S^r_{sim,rep}$ with probability greater than $1-\delta$. It follows from corollary 6.1 that $p(N, \frac{1}{\delta}) = O(l^{\mu}\ \lg(1/\delta))$ will satisfy this constraint. The polynomial q is defined as $q(i, \frac{1}{\epsilon}, \frac{1}{\delta}) = \frac{1}{\epsilon}[2\ln(i+1) + \ln(\frac{1}{\delta})]$.

Consider the execution of the algorithm $\mathcal{A}_3$. At any step i where $i \geq N$, the set S will include the characteristic set of simple examples $S^r_{sim,rep}$ with probability greater than $1-\delta$ (as

Algorithm $\mathcal{A}_3$

Input: ϵ, δ
Output: A DFA M

begin
1) $i = 1$, $EX = \phi$, $p(0, 1/\delta) = 0$
2) **repeat**
Draw $p(i, 1/\delta) - p(i-1, 1/\delta)$ examples according to $\mathbf{m}_r$
Add the examples just drawn to the set EX
Let S be the subset of examples in EX of length at most $2i-1$
$M = RPNI(S)$
Draw $q(i, 1/\epsilon, 1/\delta)$ examples according to $\mathbf{m}_r$ and call this set T
if $consistent(M, T)$
then Output M and **halt**
else $i = i + 1$
end if
until eternity
end

Figure 6.8 A PAC Algorithm for Learning DFA.

proved in lemma 6.4). In this case the RPNI algorithm will return a DFA M that is equivalent to the target A and hence M will be consistent with the test sample T. Thus, with probability at least $1 - \delta$, the algorithm will halt and correctly output the target DFA.

Consider the probability that the algorithm halts at some step i and returns a DFA M with an error greater than ϵ.

$$\begin{aligned}
\Pr(M \text{ and } A \text{ are consistent on some } \alpha) &\leq 1 - \epsilon \\
\Pr(M \text{ and } A \text{ are consistent on all } \alpha \in T) &\leq (1-\epsilon)^{|T|} \\
&\leq (1-\epsilon)^{\frac{1}{\epsilon}[2\ln(i+1)+\ln(\frac{1}{\delta})]} \\
&\leq e^{-[2\ln(i+1)+\ln(\frac{1}{\delta})]} \text{ since } 1 - x \leq e^{-x} \\
&\leq \frac{\delta}{(i+1)^2}
\end{aligned}$$

The probability that the algorithm halts at step i and returns a DFA with error greater than ϵ is at most $\sum_{i=1}^{\infty} \frac{\delta}{(i+1)^2}$ which can be shown to be strictly less than δ. Thus, we have shown that with probability greater than $1 - \delta$ the algorithm returns a DFA with error at most

ϵ. Further, the running time of the algorithm is polynomial in N, $|\Sigma|$, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and m (where m is the length of the longest test example seen by the algorithm). Thus, the class of DFA is efficiently PAC learnable under the PACS model. □

6.6 Relating the PACS Model with other Learning Models

In this section we analyze the PACS model in relation with Gold's model of *polynomial identifiability from characteristic samples* [Gol78] and Goldman and Mathias' *polynomial teachability* model [GM93] and explain how the PACS learning model naturally extends these two models to a probabilistic framework. In the discussion that follows we will let $\mathcal{C}$ be a concept class and $\mathcal{R}$ be the set of representations of the concepts in $\mathcal{C}$ (see section 6.2.1).

6.6.1 Polynomial Identifiability from Characteristic Samples

Gold's model for polynomial identifiability of concept classes from characteristic samples is based on the availability of a polynomial sized characteristic sample for any concept in the concept class and an algorithm which when given a superset of a characteristic set is guaranteed to return, in polynomial time, a representation of the target concept.

Definition 6.2 *(due to [Hig96])*
$\mathcal{C}$ is polynomially identifiable from characteristic samples iff there exist two polynomials $p_1()$ and $p_2()$ and an algorithm $\mathcal{A}$ such that

1. *Given any sample $S = S^+ \cup S^-$ of labeled examples, $\mathcal{A}$ returns in time $p_1(||S^+|| + ||S^-||)$ a representation $r \in \mathcal{R}$ of a concept $c \in \mathcal{C}$ such that c is consistent with S.*

2. *For every concept $c \in \mathcal{C}$ with corresponding representation $r \in \mathcal{R}$ there exists a characteristic sample $S_c = S_c^+ \cup S_c^-$ such that $||S_c^+|| + ||S_c^-|| = p_2(|r|)$ and if $\mathcal{A}$ is provided with a sample $S = S^+ \cup S^-$ where $S_c^+ \subseteq S^+$ and $S_c^- \subseteq S^-$ then $\mathcal{A}$ returns a representation r' of a concept c' that is equivalent to c.*

Using the above definition Gold's result can be restated as follows:

Theorem 6.4 *(due to Gold [Gol78])*

The class of DFA is polynomially identifiable from characteristic samples.

The problem of identifying a minimum state DFA that is consistent with an arbitrary labeled sample $S = S^+ \cup S^-$ is known to be NP-complete [Gol78]. This result does not contradict the one in theorem 6.4 because the characteristic set is not any arbitrary set of examples but a special set that enables the learning algorithm to correctly infer the target concept in polynomial time (see the RPNI algorithm in section 6.3).

6.6.2 Polynomial Teachability of Concept Classes

Goldman and Mathias developed a teacher-student based model for efficient learning of target concepts [GM93]. Their model takes into account the quantity of information that a good teacher must provide to the student (or learner) during learning. An additional player called the *adversary* is introduced in this model to ensure that there is no collusion whereby the teacher gives the student an encoding of the target concept. A typical teaching session proceeds as follows:

1. The adversary selects a target concept and gives it to the teacher.
2. The teacher computes a set of examples called the *teaching set*.
3. The adversary adds correctly labeled examples to the teaching set with the goal of complicating the learner's task.
4. The learner computes a hypothesis from the augmented teaching set.

Under this model, a concept class for which the computations of both the teacher and the learner takes polynomial time and the learner always learns the target concept is called *polynomially T/L teachable*. Without the restrictive assumption that teacher's computations be performed in polynomial time, the concept class is said to be *semi-polynomially T/L teachable*. Goldman and Mathias prove that this model avoids collusion [GM93]. When this model is adapted to the framework of learning DFA the length of the examples seen by the learner

must be included as a parameter in the model. In the context of learning DFA the number of examples is infinite (it includes the entire set Σ^*) and further the lengths of these examples grow unboundedly. A scenario in which the teacher constructs a very small teaching set whose examples are unreasonably long is clearly undesirable and must be avoided. This is explained more formally in the following definition.

Definition 6.3 *(due to [Hig96])*
A concept class $\mathcal{C}$ is semi-polynomially T/L teachable iff there exist polynomials $p_1()$, $p_2()$, and $p_3()$, a teacher T, and a learner L, such that for any adversary ADV and any concept c with representation r that is selected by ADV, after the following teaching session the learner returns the representation r' of a concept c' that is equivalent to c.

1. *ADV gives r to T.*
2. *T computes a teaching set S of size at most $p_1(|r|)$ such that each example in the teaching set has length at most $p_2(|r|)$.*
3. *ADV adds correctly labeled examples to this set, with the goal of complicating the learner's task.*
4. *The learner uses the augmented set S to compute a hypothesis r' in time $p_3(||S||)$.*

Note that from Gold's result (theorem 6.4) it follows that DFA are semi-polynomially T/L teachable. Further, we demonstrated in lemma 6.1 that for any DFA there exists a procedure to enumerate a characteristic set corresponding to that DFA. This procedure can be implemented in polynomial time thereby proving a stronger result that DFA are polynomially T/L teachable. Colin de la Higuera proved that the model for polynomial identification from characteristic samples and the model for polynomial teachability are equivalent to each other. More specifically, by identifying the characteristic set with the teaching sample it was shown that a concept class is polynomially identifiable from characteristic samples *iff* it is semi-polynomially T/L teachable [Hig96].

We now show how the PACS model for learning from simple examples extends the above two models to a probabilistic setting.

Lemma 6.5 *Let $c \in \mathcal{C}$ be a concept with corresponding representation $r \in \mathcal{R}$. If there exists a characteristic sample S_c for c and a polynomial $p_1()$ such that S_c can be computed from r and $||S_c|| = p_1(|r|)$ then each example in S_c is simple in the sense that $\forall \alpha \in S_c,\ K(\alpha|r) \leq \mu \lg(|r|)$ where μ is a constant.*

Proof:

Fix an ordering of the elements of S_c and define an index to identify the individual elements. Since $||S_c|| = p_1(|r|)$ an index that is $\lg(p_1(|r|)) = \mu \lg(|r|)$ bits long is sufficient to uniquely identify each element of S_c[3]. Since S_c can be computed from r we can state that there exists a Turing machine which given r reads as input an index of length $\mu \lg(|r|)$ and outputs the corresponding string of S_c. Thus, $\forall \alpha \in S_c,\ K(\alpha|r) \leq \mu \lg(|r|)$ where μ is a constant independent of α. □

Let us designate the characteristic set of simple examples S_c identified above to be the set of simple representative examples $S^r_{sim,rep}$ for the concept c represented by r. Lemma 6.4 and corollary 6.1 together show that for an integer $l \geq |r|$ and $0 < \delta < 1$ if a sample S of size $|S| = O(l^\mu \lg(\frac{1}{\delta}))$ is drawn at random according to $\mathbf{m}_r$ then with probability greater than $1 - \delta$, $S^r_{sim,rep} \subseteq S$.

Theorem 6.5 *If a concept class is polynomially identifiable from characteristic samples or equivalently semi-polynomially T/L teachable then it is probably exactly learnable under the PACS model.*

Proof:

The proof follows directly from the results of lemma 6.5, lemma 6.4, and corollary 6.1. □

6.7 Discussion

The problem of exactly learning the target DFA from an arbitrary set of labeled examples and the problem of approximating the target DFA from labeled examples under Valiant's PAC

[3]Note that if the sum of the lengths of the examples belonging to a set is k then clearly, the number of examples in that set is at most $k + 1$.

learning framework are both known to be hard problems. Thus, the question as to whether DFA are efficiently learnable under some restricted yet fairly general and practically useful classes of distributions was clearly of interest. In this chapter, we have answered this question in the affirmative by providing a framework for efficient PAC learning of DFA from simple examples.

In particular, we have demonstrated that the class of simple DFA is polynomially learnable under the universal distribution $\mathbf{m}$. Further, the class of DFA is shown to be learnable under the universal distribution $\mathbf{m}_r$ where a benign teacher might use the knowledge of the target concept to draw representative examples of the target. When an upper bound on the number of states of the target DFA is unknown, the algorithm for learning DFA under $\mathbf{m}_r$ can be used iteratively to efficiently PAC learn the concept class of DFAs for any desired error and confidence parameters. Finally, we have shown the applicability of the PACS learning model in a more general setting by proving that all concept classes that are polynomially identifiable from characteristic samples according to Gold's model and semi-polynomially T/L teachable according to Goldman and Mathias' model are also probably exactly learnable under the PACS model.

The class of simple distributions includes a large variety of probability distributions (such as all computable distributions). Li and Vitanyí have shown that a concept class is efficiently learnable under the universal distribution if and only if it is efficiently learnable under each simple distribution provided the sampling is done according to the universal distribution [LV91]. This raises the possibility of using sampling under the universal distribution to learn under all computable distributions. However, the universal distribution is not computable. Whether one can instead get by with a polynomially computable approximation of the universal distribution remains an open question. It is known that the universal distribution for the class of polynomially-time bounded simple distributions is computable in exponential time [LV91]. This opens up a number of interesting possibilities for learning under simple distributions. In a recent paper Denis and Gilleron have proposed a new model of learning under *helpful distributions* [DG97]. A helpful distribution is one in which examples belonging to the charac-

teristic set for the concept (if there exists one) are assigned non-zero probability. A systematic characterization of the class of helpful distributions would perhaps give us a more practical framework for learning from simple examples.

A related question of interest has to do with the nature of environments that can be modeled by simple distributions. In particular, if Kolmogorov complexity is an appropriate measure of the intrinsic complexity of objects in nature and if nature (or the teacher) has a propensity for simplicity, then it stands to reason that the examples presented to the learner by the environment are likely to be generated by a simple distribution. Against this background, empirical evaluation of the performance of the proposed algorithms using examples that come from natural domains is clearly of interest.

In the RPNI2 learning algorithm for incremental learning of the target DFA the learner maintains a hypothesis that is consistent with all labeled examples seen thus far and modifies it whenever a new inconsistent example is observed [Dup96a]. The convergence of this algorithm relies on the fact that sooner or later, the set of labeled examples seen by the learner will include a characteristic set. If in fact the stream of examples provided to the learner is drawn according to a simple distribution, our results show that in an incremental setting the characteristic set would be made available relatively early (during learning) with a sufficiently high probability and hence the algorithm will converge quickly to the desired target.

Some of the negative results in approximate identification of DFA are derived by showing that an efficient algorithm for learning DFA would entail algorithms for solving known hard problems such as *learning boolean formulae* [PW88] and *breaking the RSA cryptosystem* [KV89]. It would be interesting to explore the implications of our results on efficient learning of DFA from simple examples on these problems.

PART II

CONSTRUCTIVE NEURAL NETWORKS

7 INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks (ANN) are biologically inspired models of computation. ANN (also referred to simply as *neural networks*) are networks of elementary processing units that are interconnected by trainable connections. Each processing unit (or *neuron*) computes an elementary function of its inputs. The connections among neurons are responsible for transmitting signals between the neurons. Each connection has an associated *strength* or *weight* which prescribes the magnitude by which it amplifies or suppresses the signals it carries. ANN are typically represented using weighted directed graphs. The nodes of the graph correspond to the neurons, the edges refer to the connections, and the weights on the edges indicate the strengths of the corresponding connections. Each neuron performs its computation locally on the inputs it receives from the neurons to which it is connected. These computations are independent of each other. Thus, ANN have a potential for massive parallelism and fault tolerance.

Artificial neural networks can be trained to learn tasks such as *pattern classification*, *pattern matching and associative memories*, *function approximation*, *optimization*, and the like. Learning in ANN involves training the connection weights. A neural network learning algorithm is a systematic procedure for setting or updating the weights. Learning in ANN draws from the analogy with biological neural networks (animal brains) which we know are not "hard-wired" but instead capable of learning with experience. Typical learning scenarios involve a set of prototypical *patterns* called the *training set*. A pattern is a description of an object and is usually represented by a vector of attributes values. Connection weights are modified based on the network's response to each pattern in the training set and the direct or indirect feedback given to the network by the environment. It should be noted that although much of the terminology in ANN is inspired by their biological counterparts it would be incorrect to claim that ANN

are models of the animal brain. As suggested by the results of several biological experiments, there seems to be a vast difference in the computations modeled by the ANN and the actual physical and chemical processes that occur in the brain. The simplifying assumptions made in the ANN models have led researchers to question their biological plausibility. Further, the capabilities of the current ANN are considerably limited as compared to capabilities of the brain.

7.1 A Brief History

The history of ANN traces back to McCulloch and Pitts' mathematical model of a biological neuron [MP43]. Perhaps the earliest learning algorithm which is still widely used is the Hebbian learning rule [Heb49]. It states that connection weights between neurons that are simultaneously *on* or simultaneously *off* on similar inputs are reinforced. Rosenblatt proposed a simple iterative strategy called the *perceptron* learning rule for training the weights of threshold neurons [Ros58]. The perceptron algorithm attempts to find a separating linear hyperplane that partitions the pattern space into two half-planes. It acts as a binary classifier giving an output of 1 for patterns on one side of the hyperplane and and output of -1 for patterns on the other side. The simplicity of this rule was also its nemesis. Minsky and Pappert demonstrated the limits of the single perceptrons [MP69]. In particular, they showed that there are certain datasets (such as the **XOR**) that cannot be separated by a linear hyperplane. The perceptron algorithm obviously fails in these situations. It thus became clear that single neurons were incapable of learning all classification tasks and some form of internal representation (hidden neurons) would be required to correctly learn these tasks.

Several researchers actively searched for suitable training algorithms for multi-layer networks of neurons. Early learning algorithms for training such networks were proposed independently by Dreyfus [Dre62], Bryson and Ho [BH69], and Werbos [Wer74]. These algorithms use a gradient descent approach for training the multi-layer networks of neurons. The *backpropagation* learning algorithm proposed by Rumelhart *et al* [RHW86] made the gradient decent based approach popular in the neural networks community. The success of the backpropaga-

tion algorithm rekindled the excitement among researchers and led to fervent activity in both the theory and applications of neural networks. The widespread interest in ANN is evident from the number of conferences and journals that are dedicated exclusively to this field and the variety of practical applications in which ANN are used. In the past decade since the publication of the backpropagation algorithm, several hundred neural network learning algorithms (either new or variants of the existing algorithms) have appeared in the literature.

7.2 Taxonomy

We describe a taxonomy of the different neural network approaches based on the properties of the individual neurons, the architecture and topology of the neural network, the characteristics of the learning algorithm, and the application domain.

7.2.1 Neuron Properties

A neuron comprises of N input connections (typically outputs of other neurons) and a single output (see Fig. 7.1). Associated with each input connection is a trainable weight that scales the input signals. Neurons typically have an additional input connection called the *bias* or the *threshold* whose input signal is held constant at 1 and connection strength is denoted by θ. It is assumed that all the input signals arrive at the same instant of time and remain active at least until the neuron has computed its activation. The key feature of each neuron is its activation function (i.e., the mathematical function it implements). Typically, the activation is a function of the weighted sum of the inputs of the neuron. If the input signals to the neuron are designated by $x_0, x_1, \ldots, x_N$ and the corresponding weights are designated by $w_0, w_1, \ldots, w_N$ then the neuron's net input is given as $net = \sum_{i=0}^{N} w_i x_i$. The neuron's output is computed as $o = f(net)$. Note that the bias input x_0 (whose corresponding weight is $w_0 = \theta$) is permanently set to 1.

Neurons are distinguished by the activation functions they implement. The following are some typical activation functions described in ANN literature:

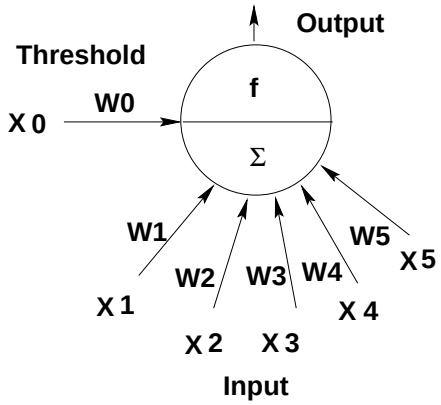

Figure 7.1 A Neuron.

- *Linear* function

$$f(net) = net$$

- *Threshold* function

$$f(net) = \begin{cases} a \text{ if } net < c \\ b \text{ if } net \geq c \end{cases}$$

- *Ramp* function

$$f(net) = \begin{cases} a \text{ if } net < c \\ b \text{ if } net \geq d \\ a + \frac{(net-c)(b-a)}{(d-c)} \text{ otherwise} \end{cases}$$

- *Sigmoid* function

$$f(net) = \frac{1}{1 + e^{-net*\beta}} \text{ where } \beta \text{ is a scaling constant .}$$

- *Gaussian* function

$$f(net) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}(\frac{net-\mu}{\sigma})^2} \text{ where } \mu \text{ is the mean and } \sigma \text{ is the standard deviation.}$$

7.2.2 Network Architecture

The different neurons in the network together with their interconnections determine the network's architecture or topology. In a typical neural network, a subset of the neurons is

designated as the *input* neurons, another subset of the neurons (not including the input neurons) is designated as the *output* neurons, and the remaining neurons (if any) are the *hidden* neurons. The input neurons have a single input connection and implement a linear activation function. Patterns are input to the network through the input neurons. The number of input neurons is equal to the total number of attributes of the training patterns. The choice of the number of hidden and output neurons and their activation functions depends on the learning algorithm and the task for which the network is being trained. For example, in classification tasks that involve assigning input patterns to one of several output categories, the number of output neurons is chosen equal to the number of output categories with one neuron per output category and the threshold activation function is chosen for each output neuron. On the other hand, in the case of function approximation tasks where the network is trained to model an unknown target function of several variables (the input attributes), a single output neuron implementing either the linear or the sigmoid activation function is used. Most practical applications require the neural network to have a non-empty set of hidden neurons. However, it is generally not possible to determine the optimal number of hidden neurons for any given task. This depends critically on the inherent complexity of the task and the number of training examples available. A common *rule-of-thumb* in neural networks is to choose the number of hidden units to be one-half of the total number of input and output units. Another approach adopted by *constructive neural network learning algorithms* (see chapter 8) is to allow the learning algorithm to dynamically determine the appropriate number of hidden neurons during training.

The most general neural network topology is the *fully connected* topology where each neuron is connected to every other neuron. This architecture is seldom used in practice since it has a large number of free parameters (n^2 connection weights and n thresholds for a network with n neurons). Besides, in practice, it is hardly ever the case that each neuron in the network has a direct influence on every other neuron. Typical neural network topologies are special cases of the fully connected network. For most practical applications it helps to organize the neurons in layers. Thus, a neural network is said to comprise of a single *input* layer, a single *output*

layer, and zero or more *hidden* layers. If the interconnections among the neurons are such that the input layer neurons are connected only to the neurons of the first hidden layer, the first hidden layer neurons are connected to the neurons of the second hidden layer, and so on with the final hidden layer neurons connected only to the neurons of output layer then the network is said to be a *strictly feed forward* network. An example of such a network is shown in Fig. 7.2. If the neurons in each layer are connected to neurons in any layer above the current layer then this type of network is called a general *feed forward* network. Finally, if the interconnections among the neurons form cyclic paths then the network is called a *recurrent* neural network.

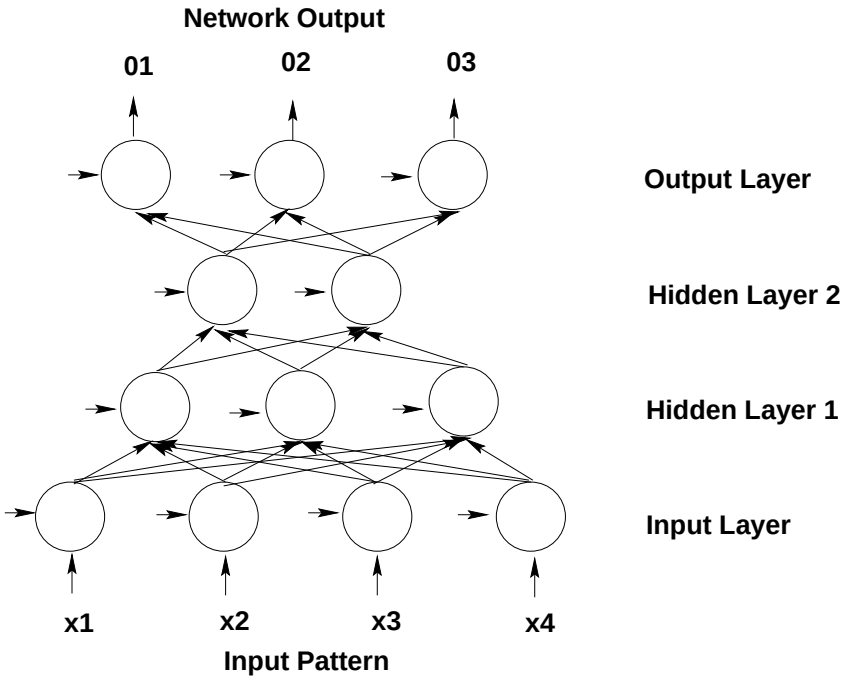

Figure 7.2 Strictly Feed Forward Network.

7.2.3 Learning Algorithms

A learning algorithm specifies how the network's connection weights and the thresholds are updated. The following three weight update strategies are prominently used in neural network learning algorithms:

- *Correlation Learning*: It is based on the Hebbian learning rule [Heb49] which states that the strength of the connection between two neurons must be gradually reinforced when the neurons have similar outputs in response to similar inputs.

- *Competitive Learning*: When an input pattern is presented to the network the neurons engage in a competitive process that involves self-excitation and mutual inhibition until a single neuron emerges as the *winner* (i.e., the neuron has the highest output magnitude among all neurons). The strength of the connections between the input nodes and the winner are boosted to increase the likelihood that the winner continues to win in future on similar patterns. This results in the development of networks where each neuron specializes on subsets of training patterns that share similar characteristics. Kohonen's self organizing maps (SOM) [Koh88] and Carpenter and Grossberg's adaptive resonance theory networks (ART) [CG88] are examples of neural networks that use competitive learning.

- *Feedback Learning*: Here the weights of the network are modified based on the feedback received by the learner from its environment. In *supervised* learning schemes the feedback is available instantly. For example, in pattern classification applications a comparison of the network's output with the desired output for an input pattern enables the learner to determine whether or not it has correctly classified that input pattern. The goal of the learning algorithm is typically to modify the network's weights so as to minimize an objective function such as sum squared error over the entire training set, the total number of misclassifications, etc. In certain learning scenarios the environment's feedback might not be available instantly. Here the learner receives a delayed reward or punishment, perhaps at the end of the task. This is *reinforcement* learning where the learner's goal is to assign appropriate credit or blame to the intermediate steps it took to complete the assigned task (after which it received a reward or punishment). This would enable the learner to determine an appropriate sequence of steps that would maximize its reward. Q-learning is a widely used algorithm for reinforcement learning [Wat89, WD92].

7.2.4 Applications

ANN have been successfully applied to problems in the areas of *pattern classification*, *clustering*, *vector quantization*, *pattern association*, *function approximation*, *optimization*, *control*, and *search* [Day90, Gal93, MMR97]. In this dissertation we focus exclusively on the use of neural networks for *pattern classification*. Pattern classification involves assigning patterns to one of several *a-priori* fixed classes. In typical classification tasks a set of training examples together with the corresponding class label is made available to the learner. It is the goal of the learner (in this case the neural network learning algorithm) to train a suitable neural network to learn a mapping from the input patterns to the output classes. The trained neural network can then be used to classify formerly unseen patterns.

7.3 Threshold Logic Units

A single threshold logic unit (TLU, also known as *perceptron*) can be trained to classify a set of input patterns into one of two classes. A TLU is an elementary processing unit that computes the threshold (hard-limiting) function of the weighted sum of its inputs. Assuming that the patterns are drawn from an N-dimensional Euclidean space, the output O^p, of a TLU with weight vector $\mathbf{W} = (w_0, w_1, \ldots, w_N)$, in response to a pattern $\mathbf{X}^p = (x_0^p, x_1^p, \ldots, x_N^p)$ is

$$\begin{aligned} O^p &= 1 \text{ if } \mathbf{W} \cdot \mathbf{X}^p \geq 0 \\ &= -1 \text{ otherwise} \end{aligned}$$

A TLU that implements the bipolar hard-limiting function (i.e., the TLU's outputs are 1 and -1) is called a *bipolar* TLU as against the TLU that implements the binary hard-limiting function (with outputs 1 and 0) which is referred to as a *binary* TLU. Unless explicitly stated otherwise, we will work with bipolar TLUs. A TLU implements a $(N-1)$-dimensional hyperplane given by $\mathbf{W} \cdot \mathbf{X} = 0$ which partitions the N-dimensional Euclidean pattern space defined by the coordinates $x_1, \cdots, x_N$ into two regions (or two classes). Given a set of *examples* $S = S^+ \cup S^-$ where $S^+ = \{(\mathbf{X}^p, C^p) \,|\, C^p = 1\}$ and $S^- = \{(\mathbf{X}^p, C^p) \,|\, C^p = -1\}$ (C^p is the desired output for the input pattern $\mathbf{X}^p$), it is the goal of a TLU training algorithm to attempt

to find a weight vector $\hat{\mathbf{W}}$ such that $\forall \mathbf{X}^p \in S^+$, $\hat{\mathbf{W}} \cdot \mathbf{X}^p \geq 0$ and $\forall \mathbf{X}^q \in S^-$, $\hat{\mathbf{W}} \cdot \mathbf{X}^q < 0$. If such a weight vector ($\hat{\mathbf{W}}$) exists for the pattern set S then S is said to be *linearly separable*. Consider the **OR** pattern set $S = \{[(-1\ -1),\ -1], [(-1\ 1),\ 1], [(1\ -1),\ 1], [(1\ 1),\ 1]\}$. This pattern set is linearly separable and a separating hyperplane defined by the weight vector $\hat{\mathbf{W}} = [1\ 1\ 1]$ is shown in Fig. 7.3. Note that the first component of the weight vector is the threshold term.

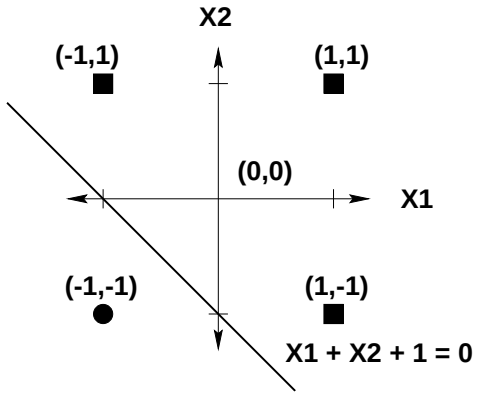

Figure 7.3 **OR** Dataset.

Several iterative algorithms are available for finding such a $\hat{\mathbf{W}}$, if one exists [Ros58, MP69, Nil65, DH73]. Most of these are variants of the *perceptron weight update rule*:

$$\mathbf{W} \leftarrow \mathbf{W} + \eta(C^p - O^p)\mathbf{X}^p \text{ where } \eta > 0 \text{ is the learning rate}$$

The perceptron weight update rule is guaranteed to find a separating hyperplane if one exists. However, since a TLU can implement only a linear hyperplane in the pattern space it will be unable to correctly separate pattern classes that are not linearly separable. The **XOR** pattern set shown in Fig. 7.4 is an example of non-linearly separable dataset.

In the case of non-linearly separable datasets, the perceptron algorithm behaves poorly i.e., the classification accuracy on the training set can fluctuate wildly from one training epoch to next. Several modifications to the perceptron weight update rule e.g., *pocket algorithm* with *ratchet modification* [Gal90], *thermal perceptron algorithm* [Fre90a, Fre92], *Loss minimization algorithm* [Hry92], and the *barycentric correction procedure* [Pou95] are proposed to find a reasonably good weight vector that correctly classifies a large fraction of the training set

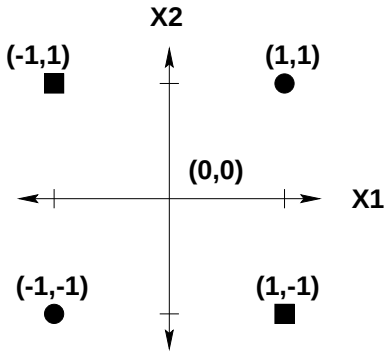

Figure 7.4 **XOR** Dataset.

S when S is not linearly separable and to converge to zero classification errors when S is linearly separable. Siu *et al* have established the necessary and sufficient conditions for a training set S to be non-linearly separable [SRK95]. They have also shown that the problem of identifying a largest linearly separable subset S_{sep} of S is NP-complete. It is widely conjectured that no polynomial time algorithms exist for NP-complete problems [GJ79]. Thus, we rely on heuristic algorithms (such as the *pocket algorithm* with *ratchet modification*) to correctly classify as large a subset of training patterns as possible within the given constraints like limited training time. We briefly summarize the *pocket algorithm* with *ratchet modification*, the *thermal perceptron algorithm*, and the *barycentric correction procedure*. The interested reader is referred to [YPH98a] for a detailed description of these algorithms and an empirical comparison of their performance on several artificial and real world datasets.

7.3.1 Pocket Algorithm with Ratchet Modification

The *pocket algorithm* with *ratchet modification* essentially uses the perceptron weight update rule. To improve the performance on non-linearly separable datasets the algorithm maintains an additional weight vector $\mathbf{W}_{pocket}$ which records the best weight setting encountered during training. The best weight setting is defined as one which results in the minimum classification error over the set of training patterns. Each time a weight $\mathbf{W}$ that correctly classifies a larger fraction of training samples as compared to the current pocket weight $\mathbf{W}_{pocket}$ is encountered, $\mathbf{W}_{pocket}$ is replaced by $\mathbf{W}$. Given enough training time, the algorithm is guaranteed

to find a weight setting W_{pocket} that will correctly classify as large a subset of the training set as possible [Gal90, Gal93].

7.3.2 Thermal Perceptron Learning Algorithm

The rationale behind the *thermal perceptron algorithm* [Fre90a] is to control the weight updates during learning and prevent drastic weight changes in response to patterns that might be outliers. The standard perceptron algorithm treats all misclassifications the same irrespective of the magnitude of the error. This can cause severe fluctuations in the classification rate for non-linearly separable datasets. To stabilize learning, a damping factor is introduced in the weight update equation:

$\mathbf{W} \leftarrow \mathbf{W} + \eta \frac{1}{T}(D^p - O^p)\mathbf{X}^p e^{-|\phi|/T}$ where ϕ is the net input and T is the temperature

T is set to an initial value T_0 at the start of learning and gradually annealed to 0 as the training progresses. Since the exponent effectively decays the learning rate, the probability of undoing previous work is reduced with time. In effect, the algorithm behaves like the perceptron algorithm at the start and avoids any large weight changes towards the end of training. Note that the performance of this algorithm is heavily dependent on the initial temperature. This difficulty can be overcome to a significant extent if at the end of each epoch the initial temperature T_0 is set to the average net input over that particular epoch [Bur94]. Training is performed for a fixed maximum number of epochs where an epoch is defined as $|S|$ presentations of randomly chosen patterns from the training set S ($|S|$ is the number of patterns in the training set).

7.3.3 Barycentric Correction Procedure

The *barycentric correction procedure* is an efficient algorithm for training a single TLU. It features separate methods for computing the weights and the threshold of the TLU. The training patterns belonging to the two classes are separated into two sets S^+ and S^- respectively. Each pattern is associated with a weighting coefficient which is initially set to 1. The weight vector $\mathbf{W} = (w_1, \ldots, w_N)$ is determined as the difference between the barycenters of

the patterns belonging to the two classes. The *barycenter* of a set of patterns is defined as a weighted mean of the patterns where each pattern is scalar multiplied by its corresponding weighting coefficient. The threshold θ is then chosen to optimize the classification accuracy in the sense that if the pattern set is linearly separable, the threshold will be set to a value such that the resulting weight vector $\mathbf{W} = (w_0, w_1, \ldots, w_N)$ will be a separating hyperplane for the two classes. However, if the pattern set is not linearly separable then the threshold will be selected to maximize the classification accuracy. Like the *pocket algorithm* with *ratchet modification*, the *barycentric correction procedure* also maintains a pocket weight $\mathbf{W}_{pocket}$ to record the best classification accuracy obtained during training. At the end of the epoch, the weighting coefficients of the patterns that are still misclassified are boosted up by a positive weighting modification. Intuitively, this causes the misclassified patterns of the two classes to be weighted more heavily in the future computation of the barycenters. Training is performed for a prespecified number of epochs at the end of which the best weight vector represented by $\mathbf{W}_{pocket}$ is returned.

7.3.4 Multiclass Discrimination

A single TLU is suitable for two category pattern classification tasks. Several practical real world problems involve classification of the given data into M ($M > 2$) output categories. A layer of M TLUs can be used to solve a M category classification task. A pattern is said to belong to class i if the i^{th} TLU outputs 1 and all the other TLUs output -1. If more than one TLU outputs 1 or if none of the TLUs output 1 then the pattern is treated as misclassified.

The group of M TLUs can be trained by *independent* training or as a *winner-take-all* (WTA) group. In independent training, the M TLUs are trained independently and in parallel. The i^{th} TLU is trained to output 1 for patterns belonging to class i and -1 for all other patterns. However, independent training does not take into account the inter-relationships among the different pattern classes. In practical classification tasks the class assignment is crisp in that a pattern assigned to class i cannot possibly belong to any other class as well. The WTA training strategy exploits this fact and gears the weight changes so that the i^{th} TLU has the

highest net input among the group of M TLUs in response to a pattern belonging to class i. The winner (i.e., the neuron with the highest net input) is assigned an output of 1 while all other neurons are assigned outputs of -1. In the event of a tie for the highest net input all neurons are assigned outputs of -1. This is described more formally as follows: Let $\mathbf{X}^p$ be a pattern and $\mathbf{C}^p = [C_1^p, C_2^p, \ldots, C_M^p]$ be its desired output. Further, let $\mathbf{O}^p = [O_1^p, O_2^p, \ldots, O_M^p]$ denote the output of the M TLUs in response to $\mathbf{X}^p$ and let $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_M$ be the current weight vectors of the M TLUs. $\mathbf{O}^p$ is computed as follows: If $\exists j \in \{1, \ldots, M\}$ such that $\mathbf{W}_j \cdot \mathbf{X}^p > \mathbf{W}_i \cdot \mathbf{X}^p \; \forall i \neq j$, then $O_j^p = 1$ and $O_i^p = -1$, $\forall i \neq j$. If $\exists j_1, j_2, \ldots, j_k \in \{1, \ldots, M\}$ such that $\mathbf{W}_{j_1} \cdot \mathbf{X}^p = \mathbf{W}_{j_2} \cdot \mathbf{X}^p = \ldots = \mathbf{W}_{j_k} \cdot \mathbf{X}^p$ and $\mathbf{W}_{j_1} \cdot \mathbf{X}^p > \mathbf{W}_i \cdot \mathbf{X}^p \; \forall i \notin \{j_1, j_2, \ldots, j_k\}$ then $O_j^p = -1$, $\forall j \in 1, \ldots, M$. In the event of a classification error (i.e., when $\mathbf{C}^p \neq \mathbf{O}^p$) the weight vector $\mathbf{W}_i$ of each TLU i for which $C_i^p \neq O_i^p$ can be modified using the perceptron weight update rule (or one of its variants). WTA training offers a potential advantage over independent training in that pattern classes that are only pairwise separable from each other can be correctly classified using WTA training while in independent training only pattern classes that are independently separable from all the other classes can be correctly classified [Gal93].

7.4 Multi-Layer Networks

A single layer of TLUs is incapable of correctly classifying pattern sets that are not linearly separable. In these situations, multi-layer networks that allow some internal representation in the form of hidden neurons are needed to learn the non-linear decision boundary required to correctly classify all training examples. A direct extension of the perceptron learning rule to multi-layer networks of TLUs is not easy to realize. We will study constructive learning algorithms that dynamically add neurons during training and train them using the perceptron learning rule (or its variants) in chapter 8. The *backpropagation* learning algorithm is a systematic procedure for training multi-layer feed forward networks. We briefly summarize the backpropagation algorithm below.

7.4.1 Backpropagation Learning Algorithm

The backpropagation algorithm is an iterative *gradient descent* based technique for learning in feed forward networks. The goal of backpropagation training is to minimize a suitably chosen objective function. In order to perform a gradient descent it is mandatory that the chosen objective function be a differentiable function. A typical objective function is the mean squared error over the set of training patterns. Gradient descent is performed iteratively. Each iteration involves a two phase weight update process. In the *forward* phase the patterns are presented to the network and the network's output in response to the pattern is determined. The error of the network is the difference between the target output for the pattern and the network's output. This enables the learner to compute the mean squared error over all the patterns in the training set. In the *backward* pass the error is propagated back through the network and the network's weights are modified in a direction that corresponds to the negative gradient of the error measure. This iterative weight update procedure is continued until the objective function encounters a local minimum. It should be noted that since the gradient term involves computation of the derivative of the neuron's output activation it is necessary that the individual neurons implement a differentiable activation function. Backpropagation networks thus cannot use threshold neurons. Instead they use neurons implementing the sigmoid activation function. The interested reader is referred to [RHW86] or any popular textbook on neural network learning (such as [Day90, Gal93, MMR97]) for a derivation of the backpropagation weight update rule.

The backpropagation algorithm and its extensions have been successfully used in several practical applications. However, the backpropagation like training algorithms suffer from the following important drawbacks:

- *A-priori* fixed network topology. The number of hidden layers and the number of neurons in each hidden layer must be fixed ahead of time. As explained earlier this poses a serious problem since there is no efficient way of determining the optimal network topology for a particular task. In backpropagation learning the network topology is either selected in an *ad-hoc* manner or by trial-and-error.

- Expensive error backward propagation. The procedure of updating weights by error backward propagation is computationally expensive and requires extensive fine tuning of parameters such as the *learning rate* and the *momentum* term to obtain a satisfactory performance.

- Though these methods are based on the mathematically well-founded principle of error minimization by gradient descent, they are susceptible to local minima which prevent the network from converging to the desired solution.

7.4.2 Constructive Learning Algorithms

Constructive (or *generative*) neural network learning algorithms offer an attractive framework for automatic construction of near-minimal networks for pattern classification and inductive knowledge acquisition systems [Hon90, HU93, Gal93]. Most constructive learning algorithms are based on simple *threshold logic units* (TLUs) that implement a hard-limiting function of their inputs. These algorithms start out by training a single TLU (using some variant of the perceptron learning rule [Ros58]) to learn to classify the set of training patterns. If the unit is not successful in correctly classifying all patterns, an additional TLU (or a group of TLUs) is added and trained to correct some of the errors made by the network. These algorithms incorporate a bias of parsimonious (or compact) networks (in terms of the number of neurons and neuron interconnections) in their search for an appropriate neural network for the given pattern classification task. Parsimonious or compact networks are preferred to more complex networks for reasons such as: simpler digital hardware implementation, ease of extracting knowledge rules from the trained network, potential for matching the intrinsic complexity of the given classification task, and capability for superior generalization. In addition, theoretical results on learnability have shown that certain concept classes can be efficiently learned provided the hypothesis space is restricted to a set of compact representations [Nat91, KV94]. Constructive learning algorithms offer the following advantages over the conventional backpropagation style learning algorithms:

- They obviate the need for an *ad-hoc*, *a-priori* choice of the network topology. Instead,

an appropriate network topology is dynamically determined during training. Thus, these algorithms have the potential for generating a near minimal network for the given task.

- They provide guaranteed convergence to zero classification errors on any finite, non-contradictory data set (under certain assumptions).
- They use elementary threshold neurons that are trained using the simple perceptron style weight update rules.
- No extensive parameter fine tuning is involved. A fixed learning rate of 1 is typically gives satisfactory performance across a variety of datasets.
- They provide a natural framework for incorporation of problem specific domain knowledge into the initial network configuration.

The *cascade correlation* learning algorithm due to Fahlman and Lebiere [FL90] differs from other constructive neural network learning algorithms in that it is based on gradient ascent training of neurons that implement a continuous differentiable activation function such as the sigmoid. As the name suggests, *cascade correlation* features cascade architecture development and correlation based training. The network starts with a single input layer and a single output layer. The cascade architecture development involves successively adding new hidden layers to the network. Each hidden layer comprises of a single neuron which is connected to all the neurons in the input layer and to all the previously added hidden neurons. The input weights of each newly added neuron are trained using gradient ascent to maximize the correlation of its output with the residual error in the network. Once the hidden neuron is trained it is connected to the neuron(s) in the output layer, the output layer weights are retrained. Fig. 7.5 shows the various stages in the execution of the cascade correlation algorithm. The solid lines indicate the network's weights that are being trained and the dotted lines indicate the weights that remain frozen.

Fahlman and Lebiere propose using the *Quickprop* learning algorithm [Fah88] to accelerate the learning process. The algorithm's performance can be further improved by training a pool of 4 or 8 neurons each time a new hidden neuron is to be added and selecting from

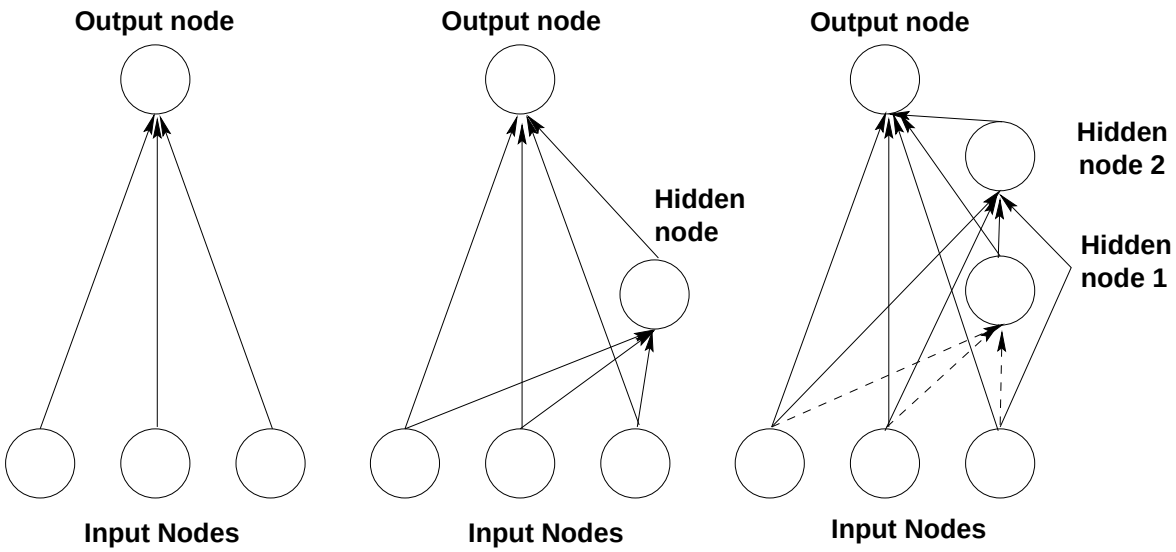

Figure 7.5 Cascade Correlation Network.

this pool a neuron that maximizes the correlation with the network's residual error. For an experimental study of the cascade correlation algorithm see [YH98]. Though the cascade correlation algorithm is considerably faster than the backpropagation algorithm, it still uses an expensive weight update scheme. In this dissertation we will focus on constructive neural network learning algorithms that use threshold logic units and simple perceptron style weight update rules.

7.5 Overview of Research Results

7.5.1 Multi-Category Real-Valued Pattern Classification

A number of algorithms that incrementally construct networks of threshold neurons for 2-category pattern classification tasks have been proposed in the literature. These include, among others, the *tower*, *pyramid* [Gal90], *tiling* [MN89], *upstart* [Fre90b], *perceptron cascade* [Bur94], and *sequential* [MGR90]. With the exception of the *upstart* and the *perceptron cascade* algorithms all the constructive learning algorithms require the input attributes to be either *binary* or *bipolar* valued.

Pattern classification tasks often require assigning patterns to one of M ($M > 2$) classes.

Although in principle an M-category classification task can be reduced to an equivalent set of M 2-category classification tasks (each with its own training set constructed from the given M-category training set), a better approach might be one that takes into account the inter-relationships between the M output classes. Additionally, practical classification tasks often involve patterns with real-valued attributes. The extensions of constructive learning algorithms to handle patterns with real-valued attributes have only been studied only for the *upstart* [ST91] and the *perceptron cascade* [Bur94] algorithms.

For each of the constructive learning algorithms mentioned above we have designed provably correct extensions to handle tasks involving multiple output categories and real-valued pattern attributes (see [PYH95, YPH96, PYH97b, PYH97a]). The convergence proofs for these algorithms outline a general framework for proving the convergence of constructive learning algorithms. Experiments on several artificial and real world datasets have demonstrated the practical applicability of these constructive learning algorithms. On most datasets the algorithms converged to fairly compact networks (in terms of the number of neurons) with zero training errors and demonstrated reasonably good generalization accuracy on the test set. Additionally, the influence of several other factors such as the TLU weight training algorithm (*pocket algorithm* with *ratchet modification*, *thermal perceptron algorithm*, or *barycentric correction procedure*), the output computation strategy (independent or WTA), and preprocessing of the dataset (normalization) on the performance of the constructive learning algorithms was borne out by the experiments we performed. We discuss the multi-category real-valued constructive learning algorithms along with their theoretical proofs of convergence and experimental results in chapter 8.

7.5.2 Network Pruning

Constructive neural network learning algorithms strive to attain parsimonious network topologies. However, in order to achieve a near-minimal network architecture, it is required that each added neuron be able to classify as large a subset of its training patterns as possible. Since the TLU weight training algorithm is only allowed limited training time, often the added

TLU might not satisfy this requirement. Further, the training of individual TLUs is based on local information in the sense that during training the weights of the remainder of the network are frozen. These factors might result in the construction of larger networks than is actually necessary for the given task.

Other things being equal, smaller (more compact) networks are desirable because of lower classification cost; potentially superior generalization performance; and transparency of the acquired knowledge in applications which involve extraction of rules from trained networks. Network pruning involves elimination of connection elements (i.e., weights or neurons) that are deemed unnecessary in that their elimination does not degrade the network's performance. In [PYH97c] we described the application of three simple neuron pruning strategies to the *MTiling* networks. Experimental results demonstrate a significant reduction in the network size without compromising the network's convergence properties or the generalization performance. We present these network pruning strategies in chapter 9.

7.5.3 Constructive Theory Refinement in Knowledge Based Neural Networks

Inductive learning systems attempt to learn a concept description from a sequence of labeled examples. The presence of domain specific knowledge (i.e., domain theories or knowledge about the concept being learned) can potentially enhance the performance of the inductive learning system both in terms of training speed and generalization ability. However, in practice the domain theory is often incomplete and even inaccurate. Inductive learning systems that use information from training examples to modify an existing domain theory by either augmenting it with new knowledge or by refining the existing knowledge are called *theory refinement* systems.

Neural network based systems for theory refinement typically operate by first embedding the knowledge rules into an appropriate initial neural network topology. This domain knowledge is then refined by training the neural network on a set of labeled examples. Constructive learning algorithms lend themselves well to the design of knowledge based neural networks for theory refinement. New rules can be incorporated and inaccuracies in the existing rules (if

any) can be corrected by dynamically adding new neurons to the neural network representing the domain theory. In chapter 10 we describe a constructive learning based approach to connectionist theory refinement [PH98b]. Specifically, we use a novel hybrid *Tiling-Pyramid* algorithm to augment the original network topology. The hybrid learning algorithm efficiently combines an adaptive *vector quantization* scheme based on the *MTiling* algorithm with the existing constructive learning algorithms to overcome some of the practical limitations of the constructive learning algorithms that prevent them from converging to zero training errors (see section 10.3.2 for more details).

8 CONSTRUCTIVE NEURAL NETWORK LEARNING ALGORITHMS FOR MULTI-CATEGORY REAL-VALUED PATTERN CLASSIFICATION

8.1 Introduction

Constructive (or generative) learning algorithms offer an attractive approach for incremental construction of potentially near-minimal neural network architectures for pattern classification tasks. These algorithms help overcome the need for *ad-hoc* and often inappropriate choice of network topology in the use of algorithms that search for a suitable weight setting in an *a-priori* fixed network architecture. The focus of this chapter is on learning algorithms that incrementally construct networks of threshold logic units (see chapter 7) to correctly classify a given (typically non-linearly separable) pattern set. Some of the motivations for studying such algorithms [Hon90, HU93] include:

- *Limitations of learning by weight modification alone within an a-priori fixed network topology:* Weight modification algorithms typically search for a solution weight vector that satisfies some desired performance criterion (e.g., classification error). In order for this approach to be successful, such a solution must lie within the weight-space being searched, and the search procedure employed must in fact, be able to locate it. This means that unless the user has adequate problem specific knowledge that could be brought to bear upon the task of choosing an appropriate network topology, the process is reduced to one of trial and error. Constructive algorithms can potentially offer a way around this problem by extending the search for a solution, in a controlled fashion, to the space of network topologies.

- *Complexity of the network should match the intrinsic complexity of the classification task:* It is desirable that a learning algorithm construct networks whose complexity (in terms of relevant criteria such as number of nodes, number of links, connectivity, etc.) is commensurate with the intrinsic complexity of the classification task (implicitly specified by the training data). Smaller networks yield efficient hardware implementations. Everything else being equal, the more compact the network, the more likely it is to exhibit better generalization properties. Constructive algorithms can potentially discover near-minimal networks for correct classification of a given dataset.

- *Estimation of expected case complexity of pattern classification tasks:* Many pattern classification tasks are known to be computationally hard. However, little is known about the *expected* case complexity of classification tasks that are encountered and successfully solved by living systems. This is primarily due to the difficulty in mathematically characterizing the properties of such problem instances. Constructive algorithms, if successful, can provide useful empirical estimates of the expected case complexity of real world pattern classification tasks.

- *Trade-offs among performance measures:* Different constructive learning algorithms offer natural means of trading off certain performance measures (like learning time) against others (like network size and generalization accuracy).

- *Incorporation of prior knowledge:* Constructive algorithms provide a natural framework for incorporating problem specific knowledge into the initial network configuration and augmenting the network to encompass additional information from the new examples seen.

A number of algorithms that incrementally construct networks of threshold neurons for 2-category pattern classification tasks have been proposed in the literature. These include the *tower*, *pyramid* [Gal90], *tiling* [MN89], *upstart* [Fre90b], *perceptron cascade* [Bur94], and *sequential* [MGR90]. With the exception of the *sequential* learning algorithm, constructive

learning algorithms are based on the idea of transforming the task of determining the necessary network topology and weights to two subtasks:

- Incremental addition of one or more threshold neurons to the network when the existing network topology fails to achieve the desired classification accuracy on the training set.
- Training the added threshold neuron(s) using some variant of the perceptron training algorithm.

In the case of the *sequential* learning algorithm, hidden neurons are added and trained by an appropriate weight training rule to exclude as many patterns belonging to the same class as possible from the currently unexcluded patterns. The constructive algorithms differ in terms of their choices regarding: restrictions on input representation (e.g., binary, bipolar, or real-valued inputs); when to add a neuron; where to add a neuron; connectivity of the added neuron; weight initialization for the added neuron; how to train the added neuron (or a subnetwork affected by the addition); and so on. The interested reader is referred to [CPY+95] for an analysis (in geometrical terms) of the decision boundaries generated by some of these constructive learning algorithms. Each of these algorithms can be shown to converge to networks which yield zero classification errors on any given training set wherein the patterns belong to one of two output classes (i.e., 2-category classification). The convergence proof is based on the ability of the TLU weight training algorithm to find a weight setting for each newly added neuron(s) such that the number of pattern misclassifications is reduced by at least one each time a neuron (or a set of neurons) is added and trained and the network's outputs are recomputed. The convergence proof of the *sequential* learning algorithm is based on the ability of the TLU weight training algorithm to exclude at least one formerly unexcluded pattern from the training set each time a new hidden neuron is trained. We will refer to such a TLU weight training algorithm as $\mathcal{A}$ and assume that it will correspond to an appropriate choice depending on the constructive algorithm being considered. In practice, the performance of the constructive algorithm depends partly on the choice of $\mathcal{A}$ and its ability to find weight settings that reduce the total number of misclassifications (or to exclude at least one formerly unexcluded pattern from the training set) each time new neurons are added to the network and trained. Some possible choices

for $\mathcal{A}$ when the desired task is to maximize classification accuracy are the *pocket algorithm* with *ratchet modification*, the *thermal perceptron algorithm*, and the *barycentric correction procedure*. A variant of the *barycentric correction procedure* can be used to efficiently exclude patterns as desired by the *sequential* learning algorithm [Pou95].

8.1.1 Multi-Category Pattern Classification

Pattern classification tasks often require assigning patterns to one of M ($M > 2$) classes. Although in principle, an M-category classification task can be reduced to an equivalent set of M 2-category classification tasks, a better approach might be one that takes into account the inter-relationships between the M output classes. For instance, the knowledge of the membership of a pattern $\mathbf{X}^p$ in category Ψ_i can be used by the learning algorithm to effectively rule out its membership in a different category Ψ_j ($j \neq i$) and any internal representations learned in inducing the structure of Ψ_i can therefore be exploited in inducing the structure of some other category Ψ_j ($j \neq i$). In the case of most constructive learning algorithms, extensions to multiple output classes have not been explored. In other cases, only some preliminary ideas (not supported by detailed theoretical or experimental analysis) for possible multi-category extensions of 2-category algorithms are available in the literature. A preliminary analysis of the extension of constructive learning algorithms to handle multi-category classification tasks is presented in [PYH95].

For pattern sets that involve multiple output classes, training can be performed either *independently* or by means of the *winner-take-all* (WTA) strategy. In the former, each output neuron is trained independently of the others using one of the TLU weight training algorithms mentioned earlier. The fact that the membership of a pattern in one class precludes its membership in all the other class can be exploited to compute the outputs using the WTA strategy wherein, for any pattern, the output neuron with the highest net input is assigned an output of 1 and all other neurons are assigned outputs of -1. In the case of a tie for the highest net input all neurons are assigned an output of -1, thereby rendering the pattern incorrectly classified. The WTA strategy succeeds in correctly classifying patterns belonging to multi-

ple output classes that are only pairwise separable from each other whereas the traditional method of computing the output of each neuron independently succeeds in correctly classifying all patterns only if the patterns belong to classes that are independently separable from each other [Gal93]. It is thus of interest to apply the WTA strategy for computing the outputs in constructive learning algorithms. For details on the adaptation of the TLU training algorithms to the WTA strategy see [YPH98a]. In this chapter we present the multi-category versions of the popular constructive learning algorithms.

8.1.2 Real-Valued Attributes

Practical classification tasks often involve patterns with real-valued attributes. The TLU weight training algorithms like the *pocket algorithm* with *ratchet modification*, *thermal perceptron algorithm*, and *barycentric correction procedure* are able to handle patterns with real-valued attributes. The original constructive learning algorithms were designed specifically to work with binary (or bipolar) valued pattern attributes. One way to deal with real-valued attributes is to use a *quantization* scheme to map the real-valued attributes to an equivalent representation of discrete valued vectors. The original constructive learning algorithm can then be applied using the quantized representations of the pattern vectors. Several quantization algorithms have been proposed in the literature [DKS95, YH96]. We will study a novel adaptive *vector quantization* technique in chapter 10.

Extensions of constructive learning algorithms to handle patterns with real-valued attributes have only been studied for the *upstart* and *perceptron cascade* algorithms (see [ST91, Bur94]). In this chapter, we present a general framework for the design of constructive learning algorithms that are capable of handling real-valued attributes. In order to guarantee convergence to zero classification errors on datasets with real-valued pattern attributes algorithms such as *tower*, *pyramid*, *upstart*, and *perceptron cascade* require a preprocessing of the dataset. Although the *tiling* and the *sequential* algorithms do not need the projection of the pattern set to guarantee convergence, such a projection would not hamper the convergence properties of these two algorithms. The following two forms of preprocessing techniques are commonly

used:

- *Projection*

  Individual patterns are projected onto a parabolic surface by appending an additional attribute to each pattern. This attribute takes on a value equal to the sum of squares of the values of all the attributes of the pattern Thus, a pattern $\mathbf{X}^p = \{X_1^p, \ldots, X_N^p\}$ is projected to a parabolic surface by augmenting an attribute $\mathbf{X}_{N+1}^p = \sum_{i=1}^{N} (X_i^p)^2$ to give the projected pattern $\hat{\mathbf{X}}^p = \{X_1^p, \ldots, X_N^p, X_{N+1}^p\}$.

- *Normalization*

  Individual patterns are normalized by dividing each attribute of the pattern by the square root of the sum of the squares of the individual attributes. Thus, a pattern $\mathbf{X}^p = \{X_1^p, \ldots, X_N^p\}$ is normalized by dividing each attribute of $\mathbf{X}^p$ by $(\sum_{i=1}^{N} (X_i^p)^2)^{1/2}$. Each normalized pattern thus has a euclidian norm of 1.

8.1.3 Notation

The following notation is used in the description of the algorithms and their convergence proofs:

Output categories: $\Psi_1, \Psi_2, \ldots, \Psi_M$

Number of pattern attributes: N

Number of output neurons (equal to the number of categories): M[1].

Input layer index: I

Indices for other layers (hidden and output): 1, 2, $\cdots$, L

Number of neurons in layer A: U_A

Indexing of neurons in layer A: $A_1, A_2, \ldots, A_{U_A}$

Threshold (or bias) of neuron i in layer A: $W_{A_i,0}$

Connection weight between neuron i in layer A and neuron j in layer B: W_{A_i,B_j}

Pattern p: $\mathbf{X}^p = < X_1^p, \ldots, X_N^p >$ where $X_i^p \in \mathcal{R}$ for all i

[1] Note that for two category classification a single output neuron with outputs 1 and -1 respectively for the two classes will suffice.

Augmented pattern p: $\mathbf{X}^p = < X_0^p, X_1^p, \ldots, X_N^p >$, $X_0^p = 1$ for all p,

and $X_i^p \in \mathcal{R}$ for all i

Projected pattern p: $\hat{\mathbf{X}}^p = < X_0^p, X_1^p, \ldots, X_N^p, X_{N+1}^p >$,

$X_{N+1}^p = \sum_{i=0}^{N} (X_i^p)^2$

Net input of neuron A_j in response to pattern $\mathbf{X}^p$: $n_{A_j}^p$

Target output for pattern $\mathbf{X}^p$: $\mathbf{C}^p = < C_1^p, C_2^p, \ldots, C_M^p >$,

$C_i^p = 1$ if $\mathbf{X}^p \in \Psi_i$ and $C_i^p = -1$ otherwise

Layer A's output in response to the pattern $\mathbf{X}^p$: $\mathbf{O}_A^p = < O_{A_1}^p, O_{A_2}^p, \ldots, O_{A_k}^p >$ where $k = U_A$

Number of patterns incorrectly classified at layer A: e_A

A pattern is said to be correctly classified at layer A when $\mathbf{C}^p = \mathbf{O}_A^p$. Define a function $sgn : \mathcal{R} \longrightarrow \{-1, 1\}$ as $sgn(x) = -1$ if $x < 0$ and $sgn(x) = 1$ if $x \geq 0$. Note that bipolar TLUs implement the *sgn* function. As is standard in neural networks literature we will assume that the input layer neurons are *linear* neurons with a single input (whose weight is set to 1). Thus, for an N-dimensional pattern set the input layer would have N linear neurons (one for each attribute of the pattern vector). The patterns are input to the neural network through the these neurons. Similarly, for projected pattern sets the input layer would have $N + 1$ linear neurons. Layers 1 through L have threshold neurons. In the following figures (for example, see Fig. 8.2) the threshold (or bias) of each TLU is depicted by a separate arrow attached to the respective TLU.

Against this background, the focus of this chapter is on provably convergent multi-category learning algorithms for construction of networks of threshold neurons for pattern classification tasks with real-valued attributes. These results are based on the work described earlier in [PYH95, YPH96, PYH97b, PYH97a]. The remainder of this chapter is organized as follows: Sections 8.2 through 8.7 explore the multi-category versions of the *tower*, *pyramid*, *upstart*, *perceptron cascade*, *tiling* and *sequential* learning algorithms respectively. In each case, convergence to zero classification errors is established for both the independent and the WTA output strategies. Note that in the following discussion we have assumed that the preprocessing of the dataset where necessary is performed by projecting each pattern to a parabolic surface

as explained in section 8.1.2. In appendix A we show how the convergence proofs (presented in sections 8.2 through 8.7) can be modified to deal with datasets having normalized pattern vectors. Section 8.8 presents preliminary results of experiments involving several artificial and real world classification tasks. Section 8.9 concludes with a summary and a discussion of future research directions.

8.2 Tower Algorithm

The 2-category *tower* algorithm [Gal90] constructs a tower of TLUs. The bottom-most neuron receives inputs from each of the N input neurons. The tower is built by successively adding neurons to the network and training them using $\mathcal{A}$ until the desired classification accuracy is achieved. Each newly added neuron receives input from each of the N input neurons and the output of the neuron immediately below itself and takes over the role of the network's output.

To handle patterns with real-valued attributes it is necessary to consider the projection of the patterns onto a parabolic surface. The extension of the 2-category *tower* algorithm to deal with multiple (M) output categories is accomplished by simply adding M neurons each time a new layer is added to the tower. Each neuron in the newly added layer (which then serves as the network's output layer) receives inputs from the $N+1$ input neurons as well as the M neurons in the preceding layer. This algorithm is described in Fig. 8.1 and the resulting *tower* network is shown in Fig. 8.2.

8.2.1 Convergence Proof

Theorem 8.1 *There exists a weight setting for neurons in the newly added layer L of the multi-category tower network such that the number of patterns misclassified by the network with L layers is less than the number of patterns misclassified prior to the addition of the L^{th} layer (i.e., $\forall L > 1,\ e_L < e_{L-1}$).*

Proof:

Define $\kappa = \max_{p,q} \sum_{i=1}^{N} (X_i^p - X_i^q)^2$. For each pattern $\hat{\mathbf{X}}^p$, define ϵ_p as $0 < \epsilon_p < \min_{p,q \neq p} \sum_{i=1}^{N} (X_i^p - X_i^q)^2$.

Algorithm: MTower (Multi-Category Real-Valued Tower Algorithm)

Input: A training set S
Output: A trained *tower* network

begin
1) Set the current output layer index $L = 0$
2) **repeat**
   // Construct a new output layer and train it
   a. $L = L + 1$
   b. Add M output neurons to the network at layer L
   c. Connect each newly added neuron to all the input neurons and to each neuron in the preceding layer $L - 1$, if one exists
   d. Train the weights of the newly added neurons using the algorithm $\mathcal{A}$ (Note that all other weights of the network remain frozen)
   until (*current_accuracy* $\geq$ *DESIRED_ACCURACY* **or** $L \geq$ *MAX_LAYERS*)
end

Figure 8.1 **MTower** Algorithm.

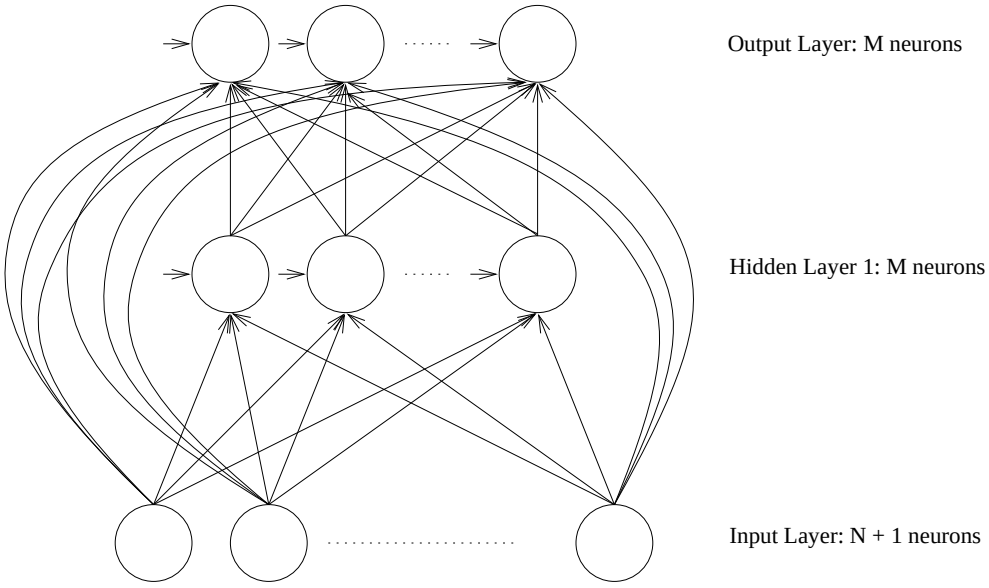

Figure 8.2 MTower Network.

It is clear that $0 < \epsilon_p < \kappa$ for all patterns $\hat{\mathbf{X}}^p$. Assume that a pattern $\hat{\mathbf{X}}^p$ was not correctly classified at layer $L-1$ (i.e., $\mathbf{C}^p \neq \mathbf{O}^p_{L-1}$). Consider the output neuron L_j $(j = 1 \ldots M)$ shown in Fig. 8.3 with the following weight setting.

$$\begin{aligned} W_{L_j,0} &= C^p_j(\kappa + \epsilon_p - \sum_{i=1}^{N}(X^p_i)^2) \\ W_{L_j,I_i} &= 2C^p_j X^p_i \text{ for } i = 1 \ldots N \\ W_{L_j,I_{N+1}} &= -C^p_j \\ W_{L_j,L-1_j} &= \kappa \\ W_{L_j,L-1_k} &= 0 \text{ for } k = 1 \ldots M, k \neq j \end{aligned} \tag{8.1}$$

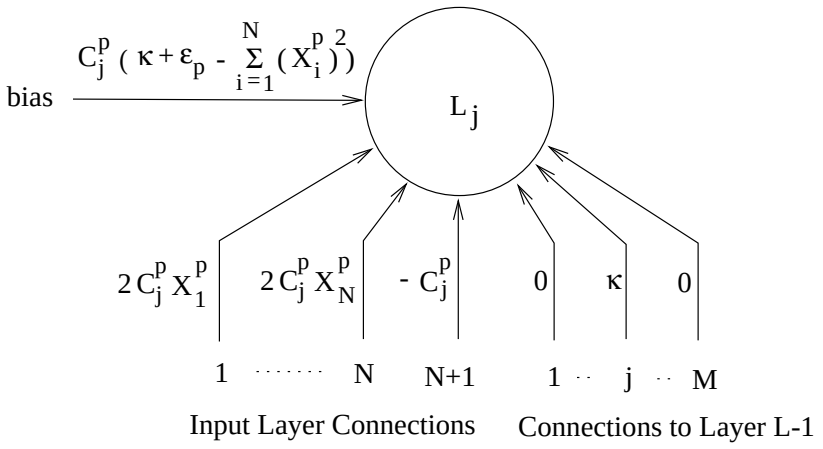

Figure 8.3 Weight Setting for the Output Neuron L_j of the MTower Network.

For the pattern $\hat{\mathbf{X}}^p$ the net input $n^p_{L_j}$ of neuron L_j is:

$$\begin{aligned} n^p_{L_j} &= W_{L_j,0} + \sum_{i=1}^{N+1} W_{L_j,I_i} X^p_i + \sum_{i=1}^{M} W_{L_j,L-1_i} O^p_{L-1_i} \\ &= C^p_j(\kappa + \epsilon_p - \sum_{i=1}^{N}(X^p_i)^2) + 2C^p_j \sum_{i=1}^{N}(X^p_i)^2 - C^p_j \sum_{i=1}^{N}(X^p_i)^2 + \kappa O^p_{L-1_j} \\ &= C^p_j(\kappa + \epsilon_p) + \kappa O^p_{L-1_j} \end{aligned} \tag{8.2}$$

If $C^p_j = -O^p_{L-1_j}$:

$$n^p_{L_j} = C^p_j \epsilon_p$$

$$\begin{aligned} O^p_{L_j} &= sgn(n^p_{L_j}) \\ &= C^p_j \text{ since } \epsilon_p > 0 \end{aligned}$$

If $C^p_j = O^p_{L-1_j}$:

$$\begin{aligned} n^p_{L_j} &= (2\kappa + \epsilon_p)C^p_j \\ O^p_{L_j} &= sgn(n^p_{L_j}) \\ &= C^p_j \text{ since } \kappa, \epsilon_p > 0 \end{aligned}$$

Thus we have shown that the pattern $\hat{\mathbf{X}}^p$ is corrected at layer L. Now consider a pattern $\hat{\mathbf{X}}^q \neq \hat{\mathbf{X}}^p$.

$$\begin{aligned} n^q_{L_j} &= W_{L_j,0} + \sum_{i=1}^{N+1} W_{L_j,I_i} X^q_i + \sum_{i=1}^{M} W_{L_j,L-1_i} O^q_{L-1_i} \\ &= C^p_j(\kappa + \epsilon_p - \sum_{i=1}^{N}(X^p_i)^2) + 2C^p_j \sum_{i=1}^{N}(X^p_i)(X^q_i) - C^p_j \sum_{i=1}^{N}(X^q_i)^2 + \kappa O^q_{L-1_j} \\ &= C^p_j(\kappa + \epsilon_p) + \kappa O^q_{L-1_j} - C^p_j \sum_{i=1}^{N}[(X^p_i)^2 - 2(X^p_i)(X^q_i) + (X^q_i)^2] \\ &= C^p_j(\kappa + \epsilon_p) + \kappa O^q_{L-1_j} - C^p_j [\sum_{i=1}^{N}(X^p_i - X^q_i)^2] \\ &= C^p_j(\kappa + \epsilon_p - \epsilon^{'}) + \kappa O^q_{L-1_j} \text{ where } \epsilon^{'} = \sum_{i=1}^{N}(X^p_i - X^q_i)^2;\ \text{note } \epsilon^{'} > \epsilon_p \\ &= \kappa^{'} C^p_j + \kappa O^q_{L-1_j} \text{ where } \kappa + \epsilon_p - \epsilon^{'} = \kappa^{'} \qquad (8.3) \\ O^q_{L_j} &= sgn(n^q_{L_j}) \\ &= O^q_{L-1_j} \text{ since } \kappa^{'} < \kappa \end{aligned}$$

Thus, for all patterns $\hat{\mathbf{X}}^q \neq \hat{\mathbf{X}}^p$, the outputs produced at layers L and $L-1$ are identical. We have shown the existence of a weight setting that is guaranteed to yield a reduction in the number of misclassified patterns whenever a new layer is added to the *tower* network. We rely on the TLU weight training algorithm $\mathcal{A}$ to find such a weight setting. Since the training set is finite in size, eventual convergence to zero errors is guaranteed. □

8.2.1.1 WTA Output Strategy

We now show that even if the output of the *tower* network is computed according to the WTA strategy, the weights for the output neurons in layer L given in equation (8.1) will ensure that the number of misclassifications is reduced by at least one.

Assume that the output vector $\mathbf{O}^p_{L-1}$ for the misclassified pattern $\hat{\mathbf{X}}^p$ is such that $O^p_{L-1_\beta} = 1$ and $O^p_{L-1_k} = -1,\ \forall k = 1 \ldots M, k \neq \beta$; whereas the target output $\mathbf{C}^p$ is such that $C^p_\gamma = 1$ and $C^p_l = -1,\ \forall l = 1 \ldots M, l \neq \gamma,$ and $\gamma \neq \beta$.

From equation (8.2) the net input for the neuron L_j is:

$$n^p_{L_j} = C^p_j(\kappa + \epsilon_p) + \kappa O^p_{L-1_j}$$

The net inputs for the output neurons L_γ, L_β, and L_j where $j = 1 \ldots M; j \neq \gamma, j \neq \beta$ are given by

$$\begin{aligned} n^p_{L_\gamma} &= C^p_\gamma(\kappa + \epsilon_p) + \kappa O^p_{L-1_\gamma} \\ &= \epsilon_p \\ n^p_{L_\beta} &= C^p_\beta(\kappa + \epsilon_p) + \kappa O^p_{L-1_\beta} \\ &= -\epsilon_p \\ n^p_{L_j} &= C^p_j(\kappa + \epsilon_p) + \kappa O^p_{L-1_j} \\ &= -2\kappa - \epsilon_p \end{aligned}$$

Since the net input of neuron L_γ is higher than that of every other neuron in the output layer, we see that by the WTA strategy $O^p_{L_\gamma} = 1$ and $O^p_{L_j} = -1,\ \forall j \neq \gamma$. Thus pattern $\hat{\mathbf{X}}^p$ is correctly classified at layer L. Consider that as a result of a tie for the highest net input the output in response to pattern $\hat{\mathbf{X}}^p$ at layer $L-1$ is $O^p_{L-1_j} = -1,\ \forall j = 1 \ldots M$. It is easy to see that given the weight setting for neurons in layer L, $\hat{\mathbf{X}}^p$ would still be correctly classified at layer L.

Consider the pattern $\hat{\mathbf{X}}^q \neq \hat{\mathbf{X}}^p$ that is correctly classified at layer $L-1$ (i.e., $\mathbf{O}^q_{L-1} = \mathbf{C}^q$). From equation (8.3), the net input for neuron L_j is:

$$n^q_{L_j} = C^p_j(\kappa + \epsilon_p - \epsilon^{'}) + \kappa O^q_{L-1_j}$$

Since, $\kappa + \epsilon_p - \epsilon' < \kappa$, it is easy to see that the neuron L_γ such that $O^q_{L-1_\gamma} = 1$ has the highest net input among all output neurons irrespective of the value assumed by C^p_γ. Thus, $\mathbf{O}^q_L = \mathbf{O}^q_{L-1} = \mathbf{C}^q$ i.e., the classification of previously correctly classified patterns remains unchanged.

We have thus proved the convergence of the *tower* algorithm when the outputs are computed according to the WTA strategy.

8.3 Pyramid Algorithm

The 2-category *pyramid* algorithm [Gal90] constructs a network in a manner similar to the *tower* algorithm, except that each newly added neuron receives input from each of the N input neurons as well as the outputs of all the neurons in each of the preceding layers. The newly added neuron becomes the output of the network. As in the case of the *tower* algorithm, the extension of the 2-category *pyramid* algorithm to handle M output categories and real-valued pattern attributes is quite straightforward. Each pattern is modified by appending the extra attribute (X^p_{N+1}). Each newly added layer of M neurons receives inputs from the $N+1$ input neurons and from each neuron in each of the previously added layers. The algorithm is described in Fig. 8.4 and the resulting *pyramid* network is shown in Fig. 8.5.

8.3.1 Convergence Proof

Theorem 8.2 *There exists a weight setting for neurons in the newly added layer L of the multi-category pyramid network such that the number of patterns misclassified by the network with L layers is less than the number of patterns misclassified prior to the addition of the L^{th} layer (i.e., $\forall L > 1,\ e_L < e_{L-1}$).*

Proof:

Define $\kappa = \max_{p,q} \sum_{i=1}^{N} (X^p_i - X^q_i)^2$. For each pattern $\hat{\mathbf{X}}^p$, define ϵ_p as $0 < \epsilon_p < \min_{p,q \neq p} \sum_{i=1}^{N} (X^p_i - X^q_i)^2$. It is clear that $0 < \epsilon_p < \kappa$ for all patterns $\hat{\mathbf{X}}^p$. Assume that a pattern $\hat{\mathbf{X}}^p$ was not correctly classified at layer $L-1$ (i.e., $\mathbf{C}^p \neq \mathbf{O}^p_{L-1}$). Consider the output neuron L_j $(j = 1 \ldots M)$ shown

Algorithm: MPyramid (Multi-Category Real-Valued Pyramid Algorithm)

Input: A training set S
Output: A trained *pyramid* network

begin
1) Set the current output layer index $L = 0$
2) **repeat**
// Construct a new output layer and train it
a. $L = L + 1$
b. Add M output neurons to the network at layer L
c. Connect each newly added neuron to all the input neurons and to each neuron in each of the preceding layers, if there exist any.
d. Train the weights of the newly added neurons using the algorithm $\mathcal{A}$ (Note that all other weights of the network remain frozen)
until (*current_accuracy* $\geq$ *DESIRED_ACCURACY* **or** $L \geq$ *MAX_LAYERS*)
end

Figure 8.4 **MPyramid** Algorithm.

in Fig. 8.6 with the following weight setting.

$$\begin{aligned} W_{L_j,0} &= C_j^p(\kappa + \epsilon_p - \sum_{i=1}^{N}(X_i^p)^2) \\ W_{L_j,I_i} &= 2C_j^p X_i^p \text{ for } i = 1 \ldots N \\ W_{L_j,I_{N+1}} &= -C_j^p \\ W_{L_j,L-i_k} &= 0 \text{ for } i = 2 \ldots L-1, \text{ and } k = 1 \ldots M \\ W_{L_j,L-1_j} &= \kappa \\ W_{L_j,L-1_k} &= 0 \text{ for } k = 1 \ldots M, k \neq j \end{aligned} \tag{8.4}$$

This choice of weights for the output layer L reduces the multi-category *pyramid* network to a multi-category *tower* network. The convergence proof (for both the independent and WTA output strategies) follows directly from the convergence proof of the *tower* algorithm. □

8.4 Upstart Algorithm

The 2-category *upstart* algorithm [Fre90b] constructs a binary tree of threshold neurons. A simple extension of this idea to deal with M output categories would be to construct M

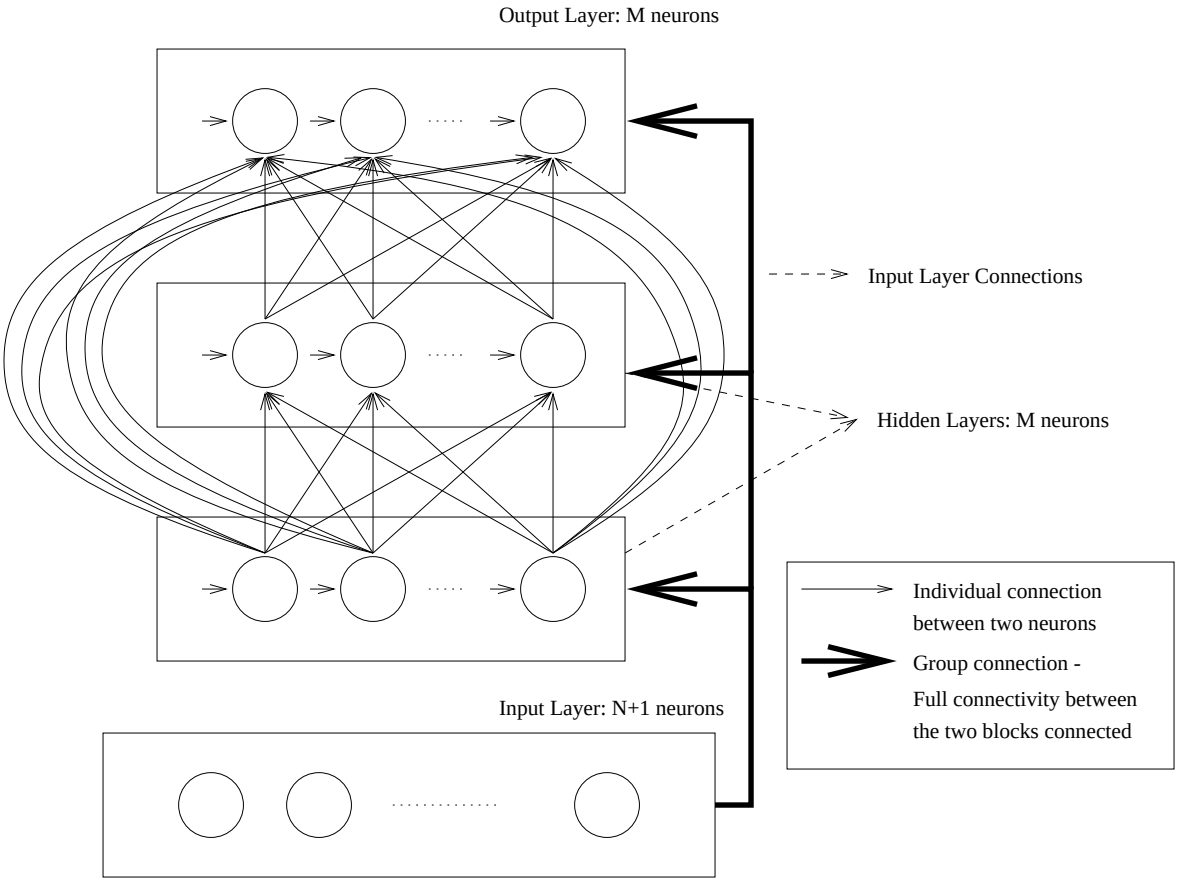

Figure 8.5 MPyramid Network.

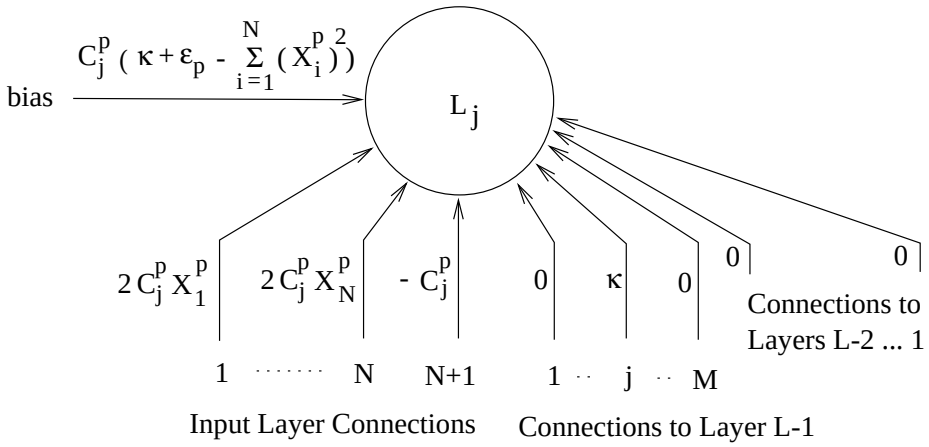

Figure 8.6 Weight Setting for the Output Neuron L_j of the MPyramid Network.

independent binary trees (one for each output class). This approach fails to exploit the inter-relationships that might exist between the different outputs. We therefore follow an alternative approach using a single hidden layer instead of a binary tree [Fre90b]. Since the original *upstart* algorithm was designed for binary valued patterns and used binary TLUs, we will present our extension of this algorithm to M classes under the same binary valued framework[2]. Again, to handle patterns with real-valued attributes we consider the projection of the pattern vectors[3].

The extension of the *upstart* algorithm to handle multiple output categories is described as follows[4]. First, an output layer of M neurons is trained using the algorithm $\mathcal{A}$. If all the patterns are correctly classified, the procedure terminates without the addition of any hidden neurons. If that is not the case, an output neuron L_k that makes at least one error in the sense $C_k^p \neq O_{L_k}^p$ on some pattern $\mathbf{X}^p$ is identified. Depending on whether the neuron k is *wrongly-on* (i.e., $C_k^p = 0, O_{L_k}^p = 1$) or *wrongly-off* (i.e., $C_k^p = 1, O_{L_k}^p = 0$) more often on the training patterns, a wrongly-on corrector daughter (X) or a wrongly-off corrector daughter (Y) is added to the hidden layer and trained to correct for some of the errors made by neuron L_k. For each pattern $\hat{\mathbf{X}}^p$ in the training set, the target outputs (C_X^p and C_Y^p) for the X and Y daughters are determined as follows:

- If $C_k^p = 0$ and $O_{L_k}^p = 0$ then $C_X^p = 0$, $C_Y^p = 0$.
- If $C_k^p = 0$ and $O_{L_k}^p = 1$ then $C_X^p = 1$, $C_Y^p = 0$.
- If $C_k^p = 1$ and $O_{L_k}^p = 0$ then $C_X^p = 0$, $C_Y^p = 1$.
- If $C_k^p = 1$ and $O_{L_k}^p = 1$ then $C_X^p = 0$, $C_Y^p = 0$.

The daughter is trained using the algorithm $\mathcal{A}$. It is then connected to each neuron in the output layer and the output weights are retrained. This algorithm is described in Fig. 8.7 and the resulting *upstart* network is shown in Fig. 8.8.

[2]The modification to handle bipolar valued patterns is straightforward with the only change being that instead of adding a X daughter or a Y daughter, a pair of X and Y daughters must be added at each time.

[3]An extension of the *upstart* algorithm to handle patterns with real-valued attributes using stereographic projection was originally proposed in [ST91].

[4]An earlier version of this algorithm appeared in [PYH97b].

Algorithm MUpstart (Multi-Category Real-Valued Upstart Algorithm)

Input: A training set S
Output: A trained *upstart* network

begin
1) Train a single layer network with M output neurons using the algorithm $\mathcal{A}$
2) Let $L = 2$ designate the above layer (it is the network's output layer)
and $H = 0$ be the number of hidden neurons
3) **while** ($current\_accuracy < DESIRED\_ACCURACY$ **and**
$H < MAX\_HIDDEN\_NEURONS$) **do**
a. Randomly pick a neuron L_k among the output neurons that make at least one error
b. Depending on whether L_k is wrongly-off or wrongly-on more often determine whether a X or a Y daughter is required
c. Increment H and add the daughter neuron to the hidden layer $L - 1$
d. Connect the new daughter neuron to the input neurons
e. Determine the training set for the daughter neuron
f. Train the daughter neuron using the algorithm $\mathcal{A}$
g. Connect the daughter neuron to all the output neurons and retrain the weights associated with the output neurons
end while
end

Figure 8.7 **MUpstart** Algorithm.

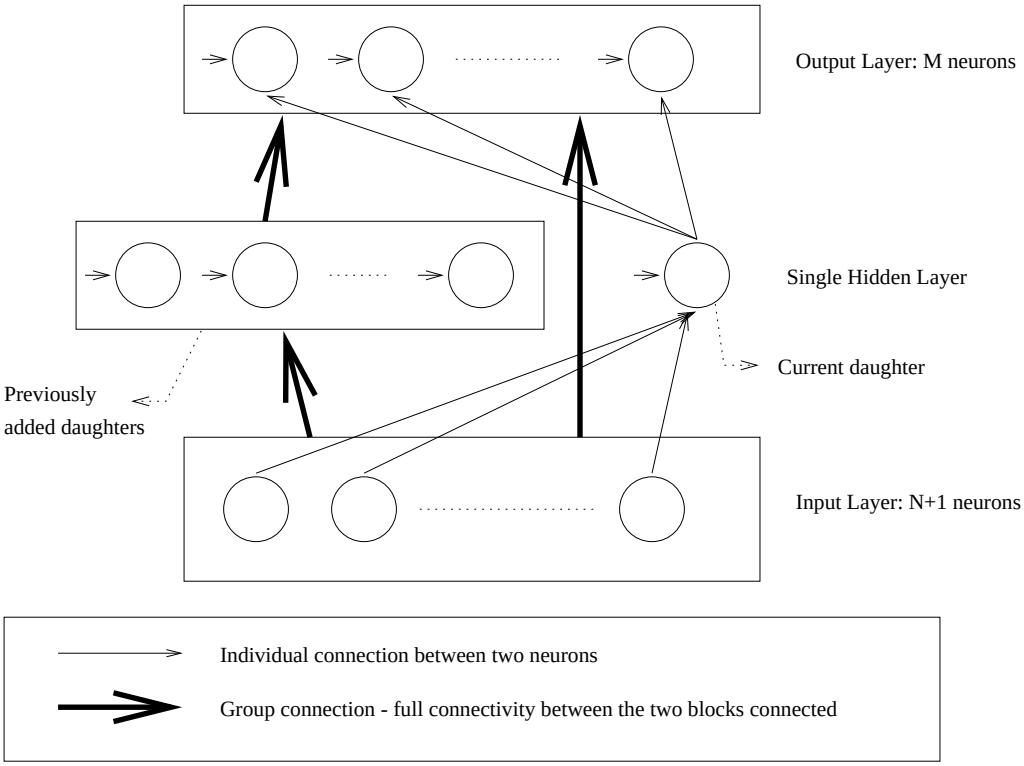

Figure 8.8 MUpstart Network.

8.4.1 Convergence Proof

Theorem 8.3 *There exists a weight setting for the X daughter neuron and the output neurons of the multi-category upstart network such that the number of patterns misclassified by the network after the addition of the X daughter and the retraining of the output weights is less than the number of patterns misclassified prior to that.*

Proof:

Assume that at some time during the training there is at least one pattern that is not correctly classified at the output layer L of M neurons[5]. Thus far, the hidden layer comprises of U_{L-1} daughter neurons. Assume also that an output neuron L_k $(1 < k < M)$ is wrongly-on for a training pattern $\hat{\mathbf{X}}^p$ (i.e., it produces an output of 1 when the desired output is in fact 0). Let $\lambda > \sum_{j=1}^{M} abs(W_{L_j,0} + \sum_{k=1}^{N+1} W_{L_j,I_k} X_k^p + \sum_{k=1}^{U_{L-1}} W_{L_j,L-1_k} O_{L-1_k}^p)$ (i.e., λ is greater than the sum of the absolute values of the net inputs of all the neurons in the output layer L in response to the pattern $\hat{\mathbf{X}}^p$). A X daughter neuron is added to the hidden layer and trained so as to correct the classification of $\hat{\mathbf{X}}^p$ at the output layer. The daughter neuron is trained to output 1 for pattern $\hat{\mathbf{X}}^p$, and to output 0 for all other patterns. Next the newly added daughter neuron is connected to all output neurons and the output weights are retrained. Consider the following weight setting for the X daughter neuron shown in Fig. 8.9.

$$\begin{aligned} W_{X,0} &= -\sum_{k=1}^{N} (X_k^p)^2 \\ W_{X,I_i} &= 2X_i^p \text{ for } i = 1 \ldots N \\ W_{X,I_{N+1}} &= -1 \end{aligned} \tag{8.5}$$

For pattern $\hat{\mathbf{X}}^p$:

$$\begin{aligned} n_X^p &= W_{X,0} + \sum_{k=1}^{N+1} W_{X,I_k} X_k^p \\ &= -\sum_{k=1}^{N} (X_k^p)^2 + \sum_{k=1}^{N} (2X_k^p) X_k^p - \sum_{k=1}^{N} (X_k^p)^2 \\ &= 0 \\ O_X^p &= 1 \text{ by definition of the binary threshold function} \end{aligned}$$

[5] In the case of the multi-category *upstart* algorithm where only two layers – the output layer and the hidden layer are constructed, the output layer index is $L = 2$ and the hidden layer index is $L - 1 = 1$.

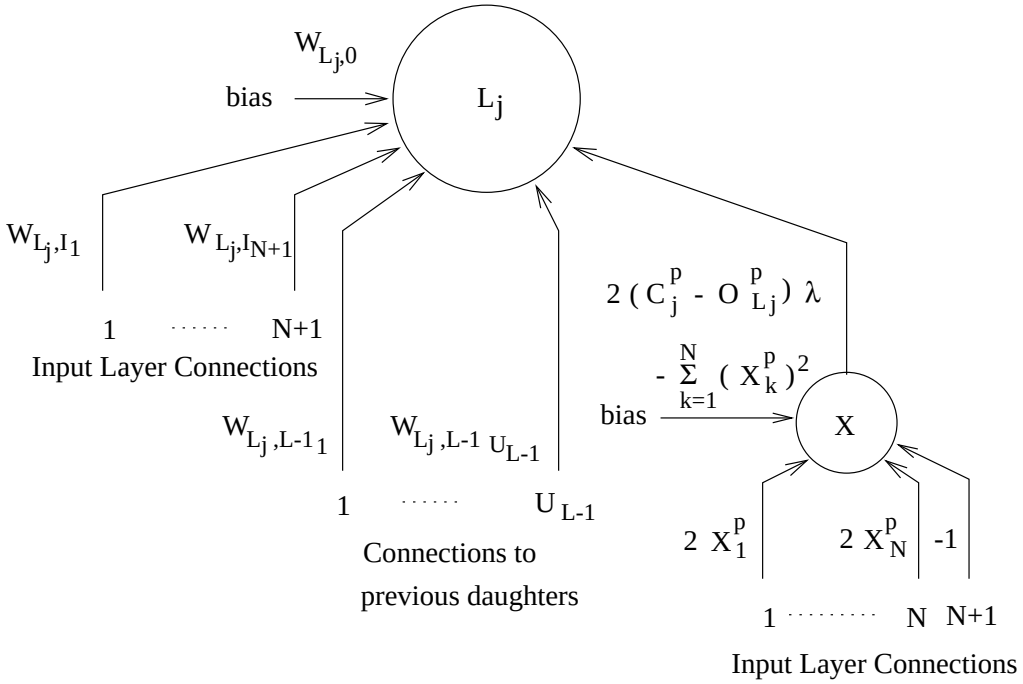

Figure 8.9 Weight Setting for the Output Neuron L_j of the MUpstart Network.

For any other pattern $\hat{\mathbf{X}}^q \neq \hat{\mathbf{X}}^p$:

$$\begin{aligned} n_X^q &= W_{X,0} + \sum_{k=1}^{N+1} W_{X,I_k} X_k^q \\ &= -\sum_{k=1}^{N} (X_k^p)^2 + \sum_{k=1}^{N} 2X_k^p X_k^q - \sum_{k=1}^{N} (X_k^q)^2 \\ &= -\sum_{k=1}^{N} (X_k^p - X_k^q)^2 \\ &< 0 \end{aligned}$$

$$O_X^q = 0 \text{ by definition of the binary threshold function}$$

Consider the following weight setting for connections between each output layer neuron and the newly trained X daughter (also shown in Fig. 8.9).

$$W_{L_j,X} = 2(C_j^p - O_{L_j}^p)\lambda$$

$O_{L_j}^p$ is the original output of neuron L_j in the output layer in response to the pattern $\hat{\mathbf{X}}^p$. (before adding the X daughter neuron). Let us consider the output of each neuron L_j in the

output layer in response to pattern $\hat{\mathbf{X}}^p$ after adding the X daughter neuron.

$$\begin{aligned} n^p_{L_j} &= W_{L_j,0} + \sum_{i=1}^{N+1} W_{L_j,I_i} X^p_i + \sum_{k=1}^{U_{L-1}} W_{L_j,L-1_k} O^p_{L-1_k} + 2(C^p_j - O^p_{L_j})\lambda O^p_X \\ &= W_{L_j,0} + \sum_{i=1}^{N+1} W_{L_j,I_i} X^p_i + \sum_{k=1}^{U_{L-1}} W_{L_j,L-1_k} O^p_{L-1_k} + 2(C^p_j - O^p_{L_j})\lambda(1) \end{aligned} \tag{8.6}$$

By the definition of λ we know that

$$-\lambda \;\leq\; \max_j [W_{L_j,0} + \sum_{i=1}^{N+1} W_{L_j,I_i} X^p_i + \sum_{k=1}^{U_{L-1}} W_{L_j,L-1_k} O^p_{L-1_k}] \;\leq\; \lambda \tag{8.7}$$

- If $C^p_j = O^p_{L_j}$ we see that the net input for neuron L_j remains the same as that before adding the daughter neuron and hence the output remains the same i.e., C^p_j.

- If $C^p_j = 0$ and $O^p_{L_j} = 1$, the net input for neuron L_j is $n^p_{L_j} \leq \lambda - 2\lambda$. Since $\lambda \geq 0$, the new output of L_j is 0 which is C^p_j.

- If $C^p_j = 1$ and $O^p_{L_j} = 0$, the net input for neuron j is $n^p_{L_j} \geq -\lambda + 2\lambda$. Since $\lambda \geq 0$, the new output of L_j is 1 which is C^p_j.

Thus, the pattern $\hat{\mathbf{X}}^p$ is corrected by the addition of the X daughter neuron. Consider any other pattern $\hat{\mathbf{X}}^q$. We know that $O^q_X = 0$.

$$\begin{aligned} n^q_{L_j} &= W_{L_j,0} + \sum_{i=1}^{N+1} W_{L_j,I_i} X^q_i + \sum_{k=1}^{U_{L-1}} W_{L_j,L-1_k} O^q_{L-1_k} + 2(C^p_j - O^p_{L_j})\lambda O^q_X \\ &= W_{L_j,0} + \sum_{i=1}^{N+1} W_{L_j,I_i} X^q_i + \sum_{k=1}^{U_{L-1}} W_{L_j,L-1_k} O^q_{L-1_k} \end{aligned} \tag{8.8}$$

We see that the X daughter neuron's contribution to the output neurons in the case of any patterns other than $\hat{\mathbf{X}}^p$ is zero. Thus, the net input of each neuron in the output layer remains the same as it was before the addition of the daughter neuron and hence the outputs for patterns other than $\hat{\mathbf{X}}^p$ remain unchanged.

A similar proof can be presented for the case when a wrongly-off corrector (i.e., a Y daughter) is added to the hidden layer. Thus, we see that the addition of a daughter neuron ensures that the number of misclassified patterns is reduced by at least one. Since the number of patterns in the training set is finite, the number of errors is guaranteed to eventually become zero. □

8.4.1.1 WTA Output Strategy

The mapping of the convergence proof for the *upstart* algorithm to the case when the output neurons are trained using the WTA strategy is straightforward. In response to the pattern $\hat{\mathbf{X}}^p$, for which a wrongly-off corrector X is trained, the net input of neuron L_j is calculated as in equation (8.6). Given this and equation (8.7), it is easy to see that the neuron L_j for which $C_j^p = 1$ has the maximum net input among all output neurons and hence pattern $\hat{\mathbf{X}}^p$ is correctly classified.

For any other pattern $\hat{\mathbf{X}}^q \neq \hat{\mathbf{X}}^p$, the net input of all the output neurons is exactly the same as the net input prior to training the new X daughter neuron (see equation (8.8)). Thus, the classification of pattern $\hat{\mathbf{X}}^q$ remains unchanged. This proves the convergence of the *upstart* algorithm when the outputs are computed according to the WTA strategy.

8.5 Perceptron Cascade Algorithm

The *perceptron cascade* algorithm [Bur94] draws on the ideas used in the *upstart* algorithm and constructs a neural network that is topologically similar to the one built by the *cascade correlation algorithm* [FL90] (see chapter 7). However, unlike the *cascade correlation algorithm* the *perceptron cascade* algorithm uses TLUs. Initially an output neuron is trained using the algorithm $\mathcal{A}$. If the output neuron does not correctly classify the training set, a daughter neuron (wrongly-on or wrongly-off as desired) is added and trained to correct some of the errors. The *perceptron cascade* algorithm differs from the *upstart* algorithm in that each newly added daughter neuron receives inputs from each of the previously added daughter neurons. As shown in Fig. 8.11 each daughter neuron is added to a new hidden layer during the construction of the *perceptron cascade* network. The targets for the daughter are determined exactly as in the case of the *upstart* network.

The extension of the *perceptron cascade* algorithm to handle M output classes is relatively straight forward. First, an output layer of M neurons is trained. If all the patterns are correctly classified, the procedure terminates without the addition of any hidden neurons. If that is not the case, an output neuron L_k that makes at least one error in the sense that $C_k^p \neq O_{L_k}^p$ is

Algorithm MCascade (Multi-Category Real-Valued Perceptron Cascade Algorithm)

Input: A training set S
Output: A trained *perceptron cascade* network

begin
1) Train a single layer network with M output neurons using the algorithm $\mathcal{A}$
2) Designate the above layer to be the output layer L and
let the number of hidden layers H be 0
3) **while** (*current_accuracy* $<$ *DESIRED_ACCURACY* **and**
$H <$ *MAX_HIDDEN_NEURONS*) **do**
a. Randomly pick a neuron L_k among the output neurons that make at least one error
b. Depending on whether L_k is wrongly-off or wrongly-on more often
determine whether a X or a Y daughter is required
c. Increment H and add the daughter neuron to a new hidden layer
(Note that the new hidden layer appears immediately below the output layer)
d. Connect the newly added daughter neuron to the input neurons and
to all the previously added daughter neurons
e. Determine the training set for the daughter neuron
f. Train the daughter neuron using the algorithm $\mathcal{A}$
g. Connect the daughter neuron to all the output neurons and
retrain the weights associated with the output neurons
end while
end

Figure 8.10 **MCascade** Algorithm.

identified and a daughter neuron (an X daughter if the neuron is wrongly-on more often or a Y daughter if the neuron is wrongly-off more often) is added to a new hidden layer and trained to correct some of the errors made by the output neurons. For each pattern $\hat{\mathbf{X}}^p$ in the training set, the target outputs for the daughter neuron are determined as described in the *upstart* algorithm. The daughter receives its inputs from each of the input neurons and from each of the previously added daughters. After the daughter neuron is trained it is connected to each of the M output neurons and the output weights are retrained. The algorithm is described in Fig. 8.10 and the resulting *perceptron cascade* network is depicted in Fig. 8.11.

8.5.1 Convergence Proof

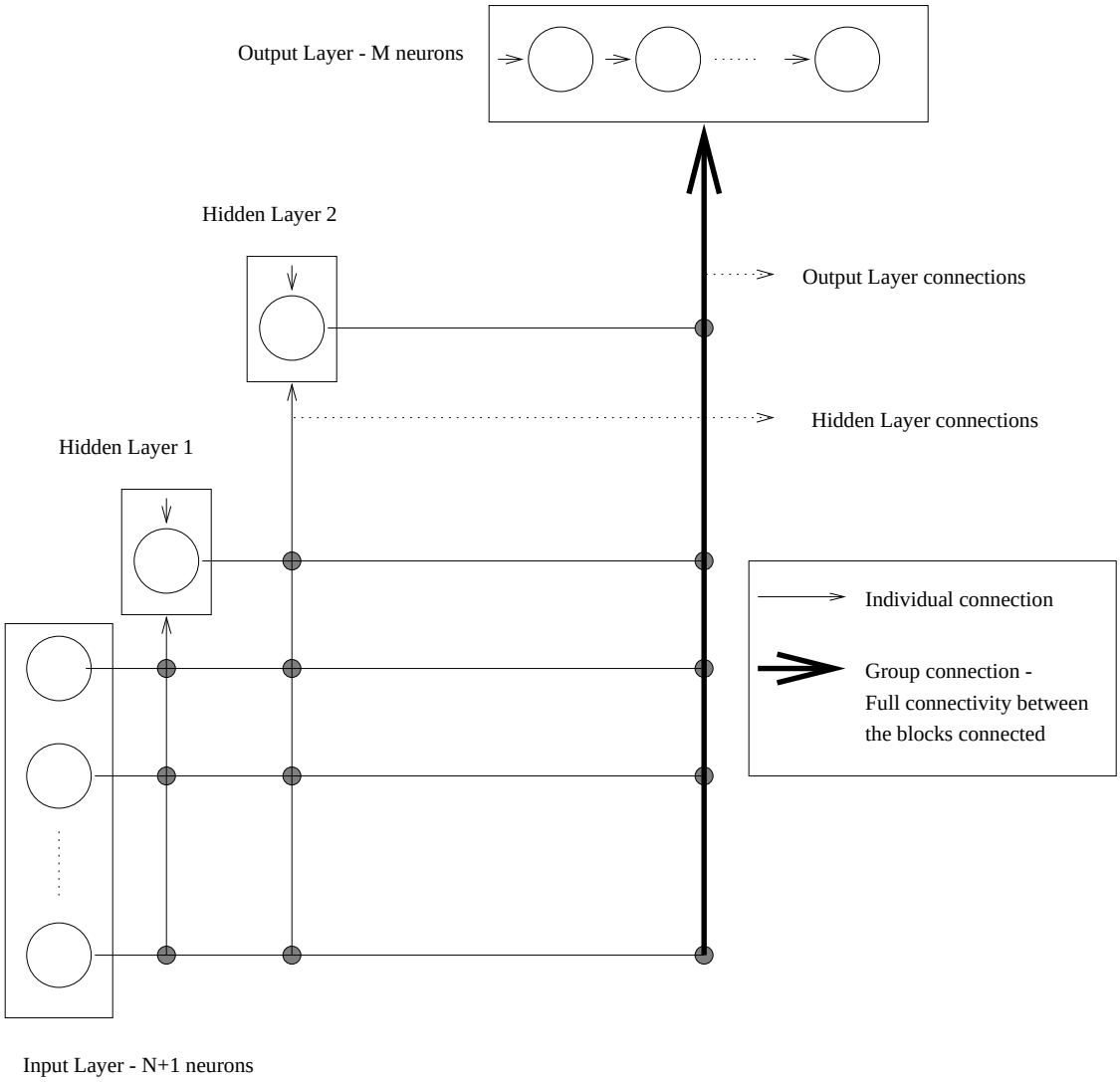

Figure 8.11 MCascade Network.

Theorem 8.4 *There exists a weight setting for the X daughter neuron and the output neurons of the multi-category perceptron cascade network such that the number of patterns misclassified by the network after the addition of the X daughter and the retraining of the output weights is less than the number of patterns misclassified prior to that.*

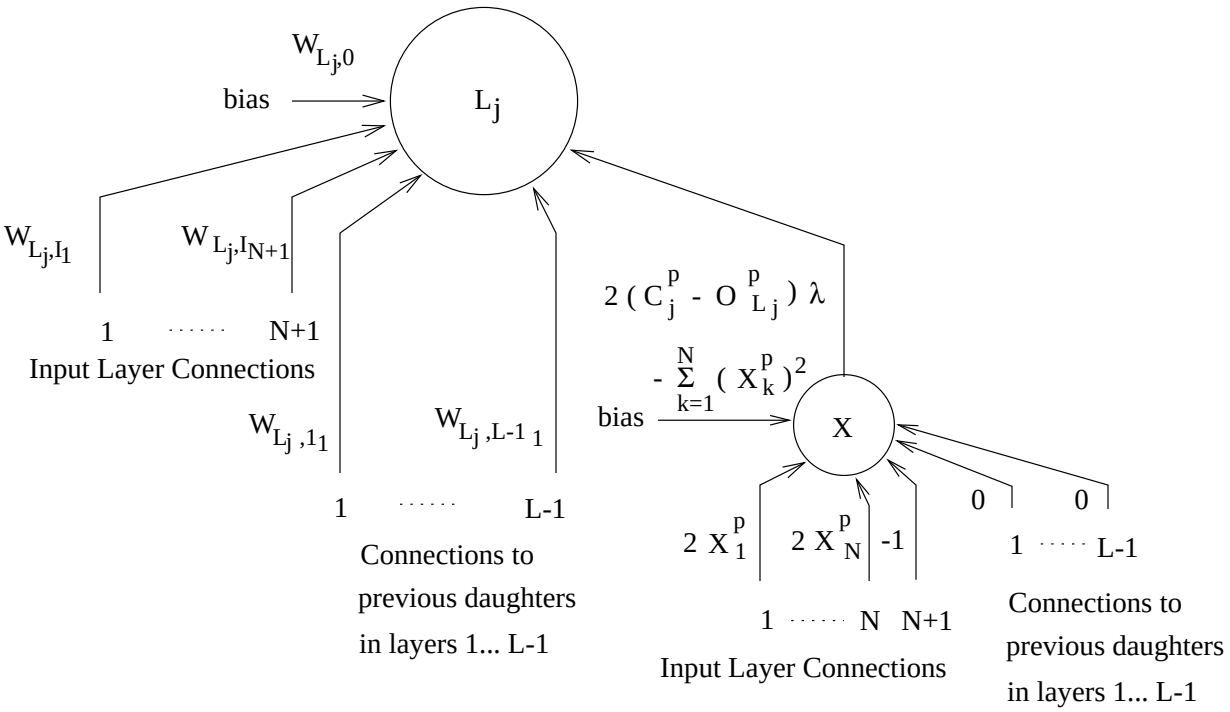

Figure 8.12 Weight Setting for the Output Neuron L_j of the MCascade Network.

Proof:

The *perceptron cascade* algorithm is similar to the *upstart* algorithm except that each newly added daughter neuron is connected to all the previously added daughter neurons in addition to all the input neurons. If we set the weights connecting the new daughter neuron to all the previous daughter neurons to zero (see Fig. 8.12), the *perceptron cascade* algorithm would behave exactly as the *upstart* algorithm. The convergence proof for the *perceptron cascade* algorithm (both in the case of the independent and WTA output strategies) thus follows directly from the proof of the *upstart* algorithm. □

8.6 Tiling Algorithm

The *tiling* algorithm [MN89] constructs a strictly layered network of threshold neurons. The bottom-most layer receives inputs from each of the N input neurons. The neurons in each subsequent layer receive inputs from those in the layer immediately below itself. Each layer maintains a *master neuron*. The network construction procedure ensures that the master neuron in a given layer correctly classifies more patterns than the master neuron of the previous layer. Each layer maintains a (possibly empty) set of ancillary neurons that are added and trained to ensure a *faithful representation* of the training patterns. The *faithfulness* criterion states that no two training examples belonging to different classes should produce identical output at any given layer. Faithfulness is clearly a necessary condition for convergence in strictly layered networks [MN89].

The proposed extension to multiple output classes involves constructing layers with M master neurons (one for each of the output classes)[6]. Unlike the other algorithms seen before, it is not necessary to preprocess the dataset using projection or normalization. Sets of one or more ancillary neurons are trained at a time in an attempt to make the current layer faithful. The algorithm is described in Fig. 8.13 and a sample *tiling* network is shown in Fig. 8.14.

8.6.1 Convergence Proof

The convergence of the multi-category *tiling* algorithm is proved in two parts: first we show that it is possible to obtain a faithful representation of the training set (with real-valued attributes) at the first hidden layer. We then show that with each additional layer the number of classification errors is reduced by at least one.

In the *tiling* algorithm each hidden layer contains M master neurons plus K ($K \geq 0$) ancillary neurons that are trained to achieve a faithful representation of the patterns in the layer. Let $\overline{S}$ be a subset of the training set S such that for each pattern $\mathbf{X}^p$ belonging to $\overline{S}$ the outputs $O_1^p, O_2^p, \ldots, O_{M+K}^p$ are exactly the same. We designate this output vector $< O_1^p, O_2^p, \ldots, O_{M+K}^p >$ as a prototype $\mathbf{\Pi}^p =< \pi_1^p, \pi_2^p, \ldots, \pi_{M+K}^p >$. $\pi_i^p = \pm 1$ for all $i =$

[6] An earlier version of this algorithm appeared in [YPH96].

Algorithm MTiling (Multi-Category Real-Valued Tiling Algorithm)

Input: A training set S
Output: A trained *tiling* network

begin
1) Train a single layer network with M output neurons using the algorithm $\mathcal{A}$
(Note that these M neurons are designated as the *master* neurons)
2) Let $L = 1$ denote the number of layers in the network
3) **while** ($current\_accuracy < DESIRED\_ACCURACY$ **and** $L < MAX\_LAYERS$) **do**
a. **while** (layer L is not faithful) **do**
// Make the current layer faithful
Let O_L be the set of outputs of layer L for the patterns in S
For each $v \in O_L$ let $S_v \subseteq S$ be the set of patterns that produced output v
and let v_k be the number of output classes to which the patterns in S_v belong
// If $v_k > 1$ *then the output vector* v *is unfaithful*
Randomly pick a v for which $v_k > 1$
Add v_k ancillary neurons to the layer L
Train the ancillary neurons using the algorithm $\mathcal{A}$
to separate the patterns in S_v
end while
b. $L = L + 1$
c. Add M master neurons to the new output layer L
d. Connect the neurons in layer L to all neurons in layer $L - 1$
e. Train the layer L on the patterns of S using the algorithm $\mathcal{A}$
end while
end

Figure 8.13 **MTiling** Algorithm.

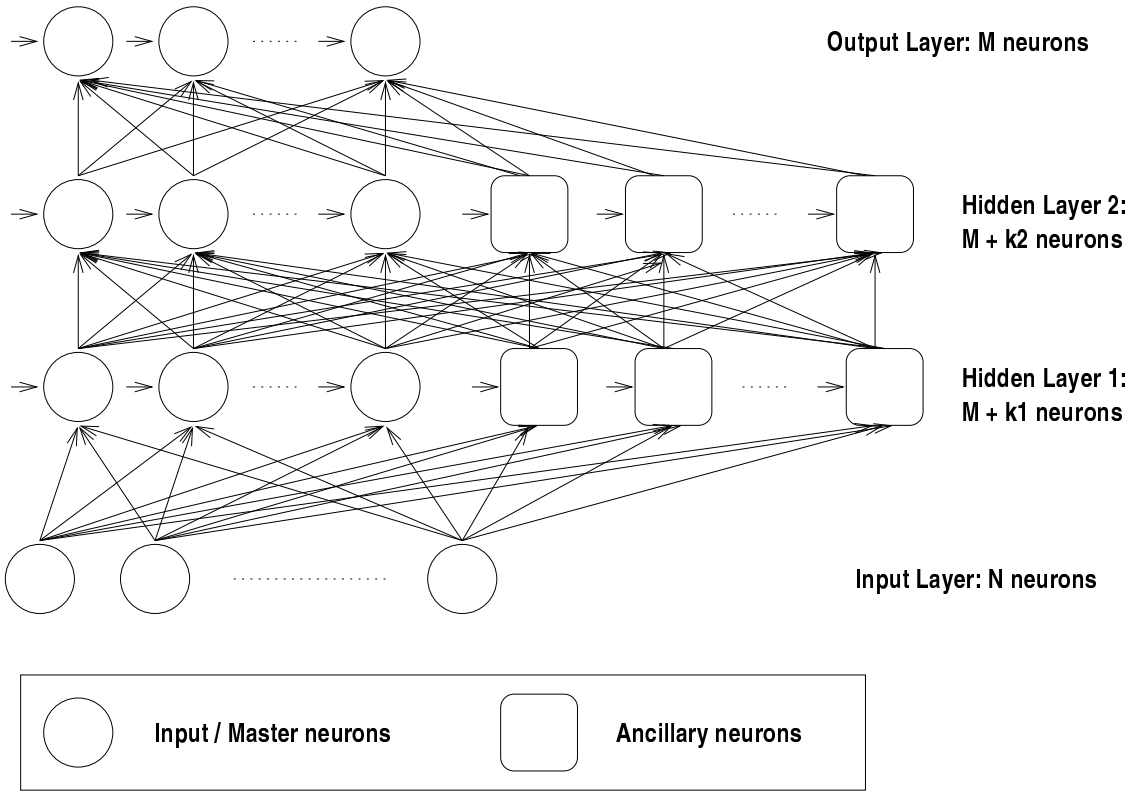

Figure 8.14 MTiling Network.

$1 \ldots (M+K)$. If all the patterns of $\overline{S}$ belong to exactly one class (i.e., they have the same desired output) then the prototype $\mathbf{\Pi}^p$ is a faithful representation of the patterns in $\overline{S}$. Further, if $< \pi_1^p, \pi_2^p, \ldots, \pi_M^p > = < C_1^p, C_2^p, \ldots, C_M^p >$ (i.e., the observed output for the patterns is the same as the desired output) then the patterns in $\overline{S}$ are said to be correctly classified.

Theorem 8.5 *For any finite non-contradictory dataset it is possible to train a layer of threshold neurons such that the outputs of these neurons provide a faithful representation of the entire training set.*

Proof:

Consider a training set S comprising of N-dimensional pattern vectors. Assume that the M master neurons are unsuccessful in correctly classifying all the patterns and that all patterns are assigned to the same output class. Thus, the representation of the set S is unfaithful. We now show that it is possible to add ancillary neurons (with appropriately set weights) that would result in a faithful representation of S for this layer of threshold neurons. Let $\mathbf{W} = \{W_0, W_1, \ldots, W_N\}$ designate the weight vector of a single TLU T.

If there exists a pattern $\mathbf{X}^p$ belonging to the *convex hull*[7] of the set S such that for some attribute i $(i = 1, \ldots, N)$ $|X_i^p| > |X_i^q|$ for all $\mathbf{X}^q \in S$ and $\mathbf{X}^q \neq \mathbf{X}^p$ then with a weight setting $\mathbf{W} = \{-(X_i^p)^2, 0, \ldots, 0, X_i^p, 0, \ldots, 0\}$ (i.e., all weights except W_0 and W_i set to 0), T will output 1 for $\mathbf{X}^p$ and -1 for all other patterns.

If however, the set S is such that there is a tie for the highest value of each attribute then the above method for excluding a single pattern will not work. In this case, there must exist a pattern $\mathbf{X}^p$ in the convex hull of S that dominates all others in the sense that for each attribute i, $X_i^p \geq X_i^q$ for all $\mathbf{X}^q$ in S. Clearly, $\mathbf{X}^p \cdot \mathbf{X}^p > \mathbf{X}^p \cdot \mathbf{X}^q$. The weights for T can be set to $\mathbf{W} = \{-\sum_{l=1}^{N}(X_l^p)^2, X_1^p, \ldots X_N^p\}$. With this weight setting T will output 1 for $\mathbf{X}^p$ and -1 for all other patterns.

Thus, the output of the layer in response to the pattern $\mathbf{X}^p$ is made faithful. Note that this output is distinct from the outputs for all the other patterns in the entire training set S.

[7] The convex hull for a set of points Q is the smallest convex polygon P such that each point in Q lies either on the boundary of P or in its interior. The interested reader is referred to [CLR91] for a detailed description of convex hulls and related topics in computational geometry.

In effect, the pattern $\mathbf{X}^p$ has been *excluded* from the remaining patterns in the training set. Similarly, using additional TLUs (up to $|S|$ TLUs in all) it can be shown that the outputs of the neurons in the layer provide a faithful representation of the entire training set S. □

Of course, in practice, by training a groups of one or more ancillary neurons using the algorithm $\mathcal{A}$ it is possible to attain a faithful representation of the input pattern set at the first hidden layer using far fewer TLUs as compared to the number of training patterns.

Theorem 8.6 *There exists a weight setting for the master neurons of the newly added layer L in the multi-category tiling network such that the number of patterns misclassified by the network with L layers is less than the number of patterns misclassified prior to the addition of the L^{th} layer (i.e., $\forall L > 1$, $e_L < e_{L-1}$).*

Proof:

Consider a prototype $\mathbf{\Pi}^p$ for which the master neurons in layer $L-1$ do not yield the correct output. i.e., $< \pi_1^p, \pi_2^p, \ldots, \pi_M^p > \neq < C_1^p, C_2^p, \ldots, C_M^p >$. The following weight setting for the master neuron L_j $(j = 1 \ldots M)$ shown in Fig. 8.3 results in the correct classification of the prototype $\mathbf{\Pi}^p$. Also, this weight setting ensures that the outputs of all other prototypes $\mathbf{\Pi}^q$ for which the master neurons of layer $L-1$ produce correct outputs (i.e., $< \pi_1^q, \pi_2^q, \ldots, \pi_M^q > = < C_1^q, C_2^q, \ldots, C_M^q >$), are unchanged.

$$\begin{aligned} W_{L_j,0} &= 2C_j^p \\ W_{L_j,L-1_k} &= C_j^p \pi_k^p \text{ for } k = 1 \ldots U_{L-1}, k \neq j \\ W_{L_j,L-1_j} &= U_{L-1} \end{aligned} \tag{8.9}$$

For the prototype $\mathbf{\Pi}^p$:

$$\begin{aligned} n_{L_j}^p &= W_{L_j,0} + \sum_{k=1}^{U_{L-1}} W_{L_j,L-1_k} \pi_k^p \\ &= 2C_j^p + U_{L-1}\pi_j^p + \sum_{k=1,k\neq j}^{U_{L-1}} C_j^p \pi_k^p \pi_k^p \\ &= 2C_j^p + U_{L-1}\pi_j^p + (U_{L-1} - 1)C_j^p \\ &= U_{L-1}\pi_j^p + (U_{L-1} + 1)C_j^p \end{aligned} \tag{8.10}$$

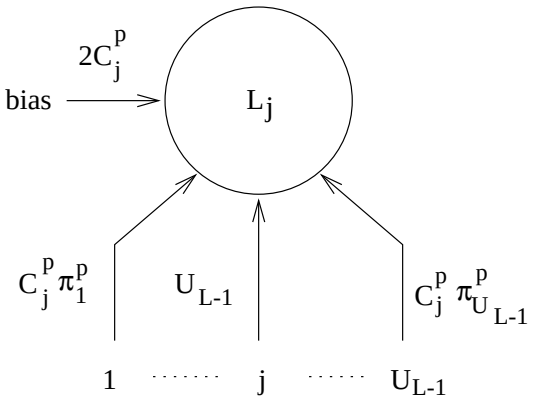

Figure 8.15 Weight Setting for the Output Neuron L_j of the MTiling Network.

$$\begin{aligned} O^p_{L_j} &= sgn(n^p_{L_j}) \\ &= C^p_j \end{aligned}$$

For the prototype $\mathbf{\Pi}^q$ (as described above) where $\mathbf{\Pi}^q \neq \mathbf{\Pi}^p$:

$$\begin{aligned} n^q_{L_j} &= W_{L_j,0} + \sum_{k=1}^{U_{L-1}} W_{L_j,L-1_k}\pi^q_k \\ &= 2C^p_j + U_{L-1}\pi^q_j + \sum_{k=1,k\neq j}^{U_{L-1}} W_{L_j,L-1_k}\pi^q_k \\ &= 2C^p_j + U_{L-1}\pi^q_j + \sum_{k=1,k\neq j}^{U_{L-1}} C^p_j\pi^p_k\pi^q_k \end{aligned} \tag{8.11}$$

CASE I:

$\pi^q_j \neq \pi^p_j$ and $\pi^q_k = \pi^p_k$ for $1 \leq k \leq U_{L-1}, k \neq j$.

For example,

$$\mathbf{\Pi}^p = < \underbrace{-1, \overbrace{+1}^{l}, -1, \ldots, \overbrace{+1}^{j}, \ldots, -1}_{M}, \underbrace{-1, \ldots, +1}_{K} >$$

$$\mathbf{\Pi}^q = < \underbrace{-1, \overbrace{+1}^{l}, -1, \ldots, \overbrace{-1}^{j}, \ldots, -1}_{M}, \underbrace{-1, \ldots, +1}_{K} >$$

Since π^q is correctly classified at layer $L-1$ whereas π^p is not, $\pi^q_j = C^p_j$ (this follows from the

fact that $\pi_j^q = -\pi_j^p$ and $C_j^p = -\pi_j^p$).

$$\begin{aligned} n_{L_j}^q &= 2C_j^p + U_{L-1}\pi_j^q + \sum_{k=1,k\neq j}^{U_{L-1}} C_j^p \pi_k^p \pi_k^q \text{ (from equation 8.11)} \\ &= 2C_j^p + U_{L-1}\pi_j^q + (U_{L-1}-1)C_j^p \\ &= U_{L-1}\pi_j^q + (U_{L-1}+1)C_j^p \\ &= (2U_{L-1}+1)\pi_j^q \text{ since } \pi_j^q = C_j^p \\ O_{L_j}^q &= sgn(n_{L_j}^q) \\ &= \pi_j^q \end{aligned}$$

CASE II:

$\pi_l^q \neq \pi_l^p$ for some l, $1 \leq l \leq U_{L-1}, l \neq j$ and $\pi_k^q = \pi_k^p$ for all k, $1 \leq k \leq U_{L-1}, k \neq j, k \neq l$

For example,

$$\begin{aligned} \mathbf{\Pi}^p &= < \underbrace{-1,-1,-1,\ldots,\overbrace{+1}^{j},\ldots,-1}_{M}, \underbrace{-1,\ldots,\overbrace{-1}^{l},\ldots,+1}_{K} > \\ \mathbf{\Pi}^q &= < \underbrace{-1,-1,-1,\ldots,\overbrace{+1}^{j},\ldots,-1}_{M}, \underbrace{-1,\ldots,\overbrace{+1}^{l},\ldots,+1}_{K} > \end{aligned}$$

In this case $\sum_{k=1,k\neq j}^{U_{L-1}} C_j^p \pi_k^p \pi_k^q \leq (U_{L-1}-3)C_j^p$

$$\begin{aligned} n_{L_j}^q &= 2C_j^p + U_{L-1}\pi_j^q + \sum_{k=1,k\neq j}^{U_{L-1}} C_j^p \pi_k^p \pi_k^q \text{ from equation 8.11} \\ &\leq 2C_j^p + U_{L-1}\pi_j^q + (U_{L-1}-3)C_j^p \\ &\leq (U_{L-1}-1)C_j^p + U_{L-1}\pi_j^q \\ O_{L_j}^q &= sgn(n_{L_j}^q) \\ &= \pi_j^q \text{ since } U_{L-1}\pi_j^q \text{ dominates } (U_{L-1}-1)C_j^p \end{aligned}$$

Once again we rely on the algorithm $\mathcal{A}$ to find the appropriate weight setting. With the above weights the previously incorrectly classified prototype $\mathbf{\Pi}^p$ would be corrected and all other prototypes that were correctly classified would remain unaffected. This reduces the number of incorrect prototypes by at least one (i.e., $e_L < e_{L-1}$). Since the training set is

finite, the number of prototypes must be finite, and with a sufficient number of layers the *tiling* algorithm would eventually converge to zero classification errors. □

8.6.1.1 WTA Output Strategy

For the incorrectly classified prototype $\mathbf{\Pi}^p$ described earlier assume that $\pi^p_\beta = 1, 1 \leq \beta \leq M$ and $\forall j = 1 \ldots M, j \neq \beta\ \pi^p_j = -1$. Clearly, $C^p_\beta = -1$ and $\exists \gamma\ 1 \leq \gamma \leq M, \gamma \neq \beta$ such that $C^p_\gamma = 1$. Given the weight settings for the master neurons in layer L in equation (8.9), the net input of neuron L_j in response to the prototype $\mathbf{\Pi}^p$ as given in equation (8.10) is

$$\begin{aligned}
n^p_{L_j} &= U_{L-1}\pi^p_j + (U_{L-1}+1)C^p_j \\
n^p_{L_\gamma} &= U_{L-1}(-1) + (U_{L-1}+1)(1) \\
&= 1 \\
n^p_{L_\beta} &= U_{L-1}(1) + (U_{L-1}+1)(-1) \\
&= -1 \text{ where } 1 \leq \beta \leq M, \beta \neq \gamma \\
n^p_{L_k} &= U_{L-1}(-1) + (U_{L-1}+1)(-1) \text{ for } k = 1 \ldots M, k \neq \gamma, k \neq \beta \\
&= -2U_{L-1} - 1
\end{aligned}$$

The master neuron L_γ has the highest net input among all master neurons in layer L which means that $O^p_{L_\gamma} = 1$ and $O^p_{L_j} = -1, \forall j = 1 \ldots M, j \neq \gamma$ and $\mathbf{C}^p = \mathbf{O}^p_L$. Thus, the prototype $\mathbf{\Pi}^p$ is now correctly classified.

Now consider the prototype $\mathbf{\Pi}^q$ that is correctly classified at layer $L-1$ (as described earlier). Since $\mathbf{\Pi}^q \neq \mathbf{\Pi}^p$, it is clear that $\pi^q_\beta = -1$ and $\exists \alpha\ 1 \leq \alpha \leq M, \alpha \neq \beta$ such that $\pi^q_\alpha = 1$. The net input of the master neurons at layer L in response to the prototype $\mathbf{\Pi}^q$ as calculated in equation (8.11) is

$$n^q_{L_j} = 2C^p_j + U_{L-1}\pi^q_j + \sum_{k=1, k\neq j}^{U_{L-1}} C^p_j \pi^p_k \pi^q_k$$

CASE I: Assume that $\alpha = \gamma$ (where $C^p_\gamma = 1$). For example,

$$\mathbf{\Pi}^p = < \underbrace{-1, \overbrace{-1}^{\alpha=\gamma}, -1, \ldots, \overbrace{+1}^{\beta}, \ldots, -1}_{M}, \underbrace{-1, \ldots, +1}_{K} >$$

$$\mathbf{\Pi}^q = < \underbrace{-1, \overbrace{+1}^{\alpha=\gamma}, -1, \ldots, \overbrace{-1}^{\beta}, \ldots, -1}_{M}, \underbrace{-1, \ldots, +1}_{K} >$$

In this case $(2M - U_{L-1} - 3) \leq [\sum_{k=1,k\neq\alpha}^{U_{L-1}} \pi_k^p \pi_k^q] \leq (U_{L-1} - 3)$. The net input for the output neuron L_α is

$$\begin{aligned} n_{L_\alpha}^q &= 2C_\alpha^p + U_{L-1}\pi_\alpha^q + C_\alpha^p[\sum_{k=1,k\neq\alpha}^{U_{L-1}} \pi_k^p \pi_k^q] \\ &= 2(1) + U_{L-1}(1) + (1)[\sum_{k=1,k\neq\alpha}^{U_{L-1}} \pi_k^p \pi_k^q] \\ &\geq 2 + U_{L-1} + 2M - U_{L-1} - 3 \\ &\geq 2M - 1 \end{aligned}$$

Similarly, the net input for any neuron j (other than α) in the output layer is given by

$$\begin{aligned} n_{L_j}^q &= 2C_j^p + U_{L-1}\pi_j^q + C_j^p[\sum_{k=1,k\neq j}^{U_{L-1}} \pi_k^p \pi_k^q] \text{ for } j = 1 \ldots M, j \neq \alpha \\ &= 2(-1) + U_{L-1}(-1) + (-1)[\sum_{k=1,k\neq\alpha}^{U_{L-1}} \pi_k^p \pi_k^q] \\ &\leq -2 - U_{L-1} + (-1)(2M - U_{L-1} - 3) \\ &\leq 1 - 2M \end{aligned}$$

Since $M \geq 3$ we see that the net input of neuron L_α is higher than the net input of any other master neuron in the output layer. Thus, $O_{L_\alpha}^q = 1$ and $O_{L_j}^q = -1 \; \forall j = 1 \ldots M, j \neq \alpha$ which means that $\mathbf{C}^q = \mathbf{O}_L^q$ as desired.

CASE II: Assume $\alpha \neq \gamma$ (where $C_\gamma^p = 1$). For example,

$$\mathbf{\Pi}^p = < \underbrace{-1, \overbrace{-1}^{\gamma}, -1, \ldots, \overbrace{+1}^{\beta}, \ldots, \overbrace{-1}^{\alpha}, \ldots, -1}_{M}, \underbrace{-1, \ldots, +1}_{K} >$$

$$\mathbf{\Pi}^q = < \underbrace{-1, \overbrace{-1}^{\gamma}, -1, \ldots, \overbrace{-1}^{\beta}, \ldots, \overbrace{+1}^{\alpha}, \ldots, -1}_{M}, \underbrace{-1, \ldots, +1}_{K} >$$

In this case $C_\alpha^q = 1$, $(2M - U_{L-1} - 3) \leq [\sum_{k=1,k\neq\alpha}^{U_{L-1}} \pi_k^p \pi_k^q] \leq (U_{L-1} - 3)$, and $(2M - U_{L-1} - 5) \leq [\sum_{k=1,k\neq\gamma}^{U_{L-1}} \pi_k^p \pi_k^q] \leq (U_{L-1} - 5)$.

The net input for output neuron L_α is

$$\begin{aligned} n^q_{L_\alpha} &= 2C^p_\alpha + U_{L-1}\pi^q_\alpha + C^p_\alpha[\sum_{k=1,k\neq\alpha}^{U_{L-1}} \pi^p_k\pi^q_k] \\ &= 2(-1) + U_{L-1}(1) + (-1)[\sum_{k=1,k\neq\alpha}^{U_{L-1}} \pi^p_k\pi^q_k] \\ &\geq -2 + U_{L-1} - 1(U_{l-1} - 3) \\ &\geq 1 \end{aligned}$$

Similarly, the net input for the output neuron L_γ is

$$\begin{aligned} n^q_{L_\gamma} &= 2C^p_\gamma + U_{L-1}\pi^q_\gamma + C^p_\gamma[\sum_{k=1,k\neq\gamma}^{U_{L-1}} \pi^p_k\pi^q_k] \\ &= 2(1) + U_{L-1}(-1) + 1[\sum_{k=1,k\neq\gamma}^{U_{L-1}} \pi^p_k\pi^q_k] \\ &\leq 2 - U_{L-1} + (U_{l-1} - 5) \\ &\leq -3 \end{aligned}$$

Finally, the net input of the output neuron L_j where $j \neq \alpha, j \neq \gamma$ is given by

$$\begin{aligned} n^q_{L_j} &= 2C^p_j + U_{L-1}\pi^q_j + C^p_j[\sum_{k=1,k\neq j}^{U_{L-1}} \pi^p_k\pi^q_k] \\ &= 2(-1) + U_{L-1}(-1) + (-1)[\sum_{k=1,k\neq j}^{U_{L-1}} \pi^p_k\pi^q_k] \\ &\leq -2 - U_{L-1} - (2M - U_{L-1} - 3) \\ &\leq -2M + 1 \end{aligned}$$

Again, since $M \geq 3$ we see that the net input of neuron L_α is higher than the net input of any other neuron in the output layer. Thus, $O^q_{L_\alpha} = 1$ and $O^q_{L_j} = -1\ \forall j = 1 \ldots M, j \neq \alpha$ which means that $\mathbf{C}^q = \mathbf{O}^q_L$ as desired. We have shown that if the output of the master neurons is computed according to the WTA strategy there is a weight setting for a newly added group of master neurons which will reduce the number of misclassifications by at least one.

8.7 Sequential Learning Algorithm

The *sequential* learning algorithm [MGR90] offers an alternative method for network construction where instead of training neurons to correctly classify a maximal subset of the training patterns, the idea is to train hidden neurons to sequentially exclude patterns belonging to one class from the remaining patterns. When all the patterns in the training set have been thus excluded, the internal representation of the patterns at the hidden layer is guaranteed to be linearly separable. A single output layer where the neurons are connected to all the hidden layer neurons can then be constructed to correctly classify all the patterns in the training set. Recently, Poulard has shown that a variation of the *barycentric correction procedure* can be used effectively in *sequential* learning to exclude as many patterns belonging to a single class as possible [Pou95].

The extension of the *sequential* learning algorithm to multiple output categories follows the same principles as the original version. Using a simple modification of the *barycentric correction procedure*, hidden neurons can be trained to exclude patterns belonging to one of the M classes from the remaining patterns. Once all the patterns in the training set have been excluded by the hidden layer neurons, the output layer with M TLUs can be constructed to correctly classify all patterns. As in the case of the *tiling* algorithm, it is not necessary to perform preprocessing of the training patterns to prove the convergence for patterns with real-valued attributes. The *sequential* learning algorithm is described in Fig. 8.16 and a sample network constructed by *sequential* learning is shown in Fig. 8.17.

8.7.1 Convergence Proof

We prove the convergence of this algorithm in two parts. Firstly, we show that it is possible to construct a hidden layer to sequentially exclude all patterns in the training set. Next we show that if the weights of the output layer neurons are set as described in the algorithm (see Fig. 8.16) then all the patterns in the training set are correctly classified.

Algorithm: MSequential (Multi-Category Real-Valued Sequential Learning Algorithm)

Input: A training set S
Output: A trained *sequential* network

begin
1) $i \leftarrow 1$
2) Initialize S to the entire set of training patterns
3) **while** $(S \neq \phi)$ **do**
   a. Train a pool of M neurons using the *barycentric correction procedure* (*sequential* learning version). Neuron k $(k = 1, \ldots, M)$ is trained to exclude as many patterns belonging to Ψ_k from the remaining patterns in S as possible
   b. Pick the neuron (trained in the previous step) that excludes the largest subset of patterns in S and designate it as neuron i in the hidden layer
   c. Let E^i be the set of patterns excluded by the hidden layer neuron i
   d. $S \leftarrow S - E^i$
   e. $i = i + 1$
   end while
4) Construct the output layer with M neurons
   Each output neuron is connected to all the neurons in the hidden layer
5) Set the weights for the output layer neurons as follows

$$W_{L_j,L-1_k} = \begin{cases} 2^{U_{L-1}+1-k} & \text{if neuron } L-1_k \text{ excludes } \Psi_j \\ -2^{U_{L-1}+1-k} & \text{otherwise} \end{cases}$$

$$W_{L_j,0} = \sum_{k=1}^{U_{L-1}} W_{L_j,L-1_k} \qquad (8.12)$$

end

Figure 8.16 **MSequential** Algorithm.

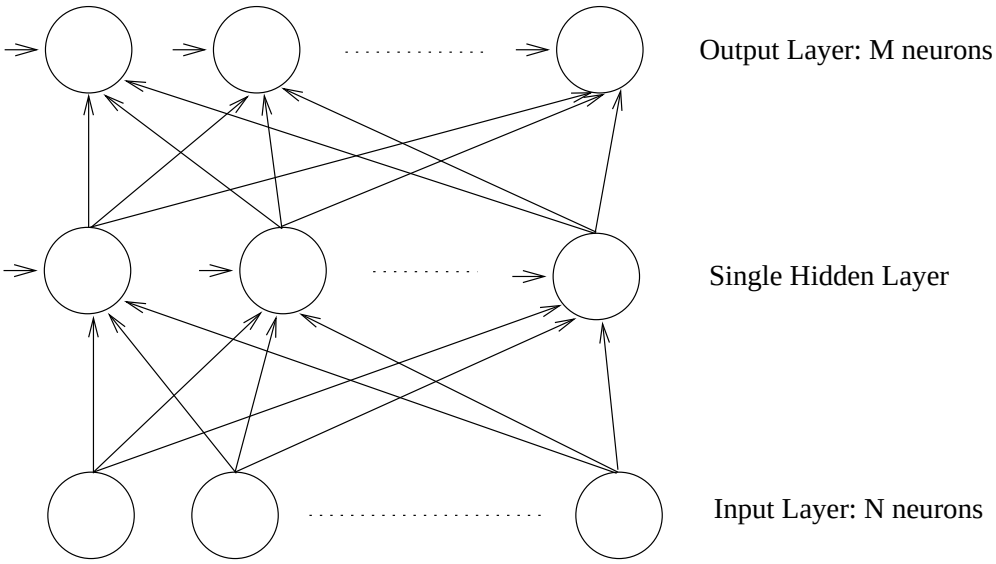

Figure 8.17 MSequential Network.

8.7.1.1 Construction of the Hidden Layer

Given the training set S for the neuron i of the hidden layer, intuitively it is clear that a weight setting exists for which one pattern belonging to the *convex hull* of the set S can be excluded from the rest. The proof of theorem 8.5 can be used directly to show that it is possible to construct a layer of threshold neurons that sequentially excludes patterns belonging to any finite dataset.

8.7.1.2 Construction of the Output Layer

Consider that the hidden layer $L-1$ with U_{L-1} neurons is trained to sequentially exclude all patterns. The output layer L with M neurons is constructed with each neuron connected to all the U_{L-1} neurons in the hidden layer. Given that the weights of the output layer neurons are set as described in the algorithm (see Fig. 8.16) we show that all patterns belonging to the training set are correctly classified by the network.

Theorem 8.7 *(Sequential Learning Theorem[8])*

The internal representation of the training patterns that are excluded sequentially by the neu-

[8] *A version of the sequential learning theorem for two category pattern classification was originally proposed by [MGR90].*

rons in the single hidden layer is linearly separable.

Proof:

Let $\mathbf{X}^p$ be a pattern belonging to Ψ_j $(1 \leq j \leq M)$ and excluded by neuron $L-1_k$ $(1 \leq k \leq U_{L-1})$. By construction, the hidden neurons $L-1_1, L-1_2, \ldots L-1_{k-1}$ output -1, the neuron $L-1_k$ outputs 1, and the hidden neurons $L-1_{k+1}, \ldots, L_{U_{L-1}}$ output 1 or -1 in response to pattern $\mathbf{X}^p$. The net input of the neuron L_j is:

$$\begin{aligned}
n^p_{L_j} &= W_{L_j,0} + \sum_{l=1}^{U_{L-1}} W_{L_j,L-1_l} O^p_{L-1_l} \\
&= \sum_{l=1}^{U_{L-1}} W_{L_j,L-1_l} + \sum_{l=1}^{k-1} W_{L_j,L-1_l}(-1) + W_{L_j,L-1_k}(1) + \sum_{l=k+1}^{U_{L-1}} W_{L_j,L-1_l} O^p_{L-1_l} \\
&= \sum_{l=k+1}^{U_{L-1}} W_{L_j,L-1_l} + 2W_{L_j,L-1_k} + \sum_{l=k+1}^{U_{L-1}} W_{L_j,L-1_l} O^p_{L-1_l} \\
&\geq 2W_{L_j,L-1_k} \\
&> 0 \\
O^p_{L_j} &= sgn(n^p_{L_j}) \\
&= 1
\end{aligned} \tag{8.13}$$

The net input of any other output neuron L_i $(i = 1, \ldots, M$ and $i \neq j$ $)$ is

$$\begin{aligned}
n^p_{L_i} &= W_{L_i,0} + \sum_{l=1}^{U_{L-1}} W_{L_i,L-1_l} O^p_{L-1_l} \\
&= \sum_{l=1}^{U_{L-1}} W_{L_i,L-1_l} + \sum_{l=1}^{k-1} W_{L_i,L-1_l}(-1) + W_{L_i,L-1_k}(1) + \sum_{l=k+1}^{U_{L-1}} W_{L_i,L-1_l} O^p_{L-1_l} \\
&= \sum_{l=k+1}^{U_{L-1}} W_{L_i,L-1_l} + 2W_{L_i,L-1_k} + \sum_{l=k+1}^{U_{L-1}} W_{L_i,L-1_k} O^p_{L-1_l} \\
&= \sum_{l=k+1}^{U_{L-1}} W_{L_i,L-1_l} - 2|W_{L_i,L-1_k}| + \sum_{l=k+1}^{U_{L-1}} W_{L_i,L-1_k} O^p_{L-1_l} (\text{ since } W_{L_i,L-1_k} < 0) \\
&\leq 2 \sum_{l=k+1}^{U_{L-1}} |W_{L_i,L-1_l}| - 2|W_{L_i,L-1_k}| \\
&< 0 (\text{ since } |W_{L_i,L-1_k}| > \sum_{l=k+1}^{U_{L-1}} |W_{L_i,L-1_l}|) \\
O^p_{L_i} &= sgn(n^p_{L_i})
\end{aligned}$$

$$= \quad -1 \tag{8.14}$$

Thus the network correctly classifies $\mathbf{X}^p$ as belonging to Ψ_j. Since each pattern is thus correctly classified we have demonstrated that the internal representation of the training patterns that are excluded sequentially by the neurons in the single hidden layer are linearly separable. □

8.7.1.3 WTA Output Strategy

In the case of the *sequential* learning algorithm, the weight assignment for the output weights from equation (8.12) ensures that for a pattern $\mathbf{X}^p$ belonging to Ψ_j, the net input of output neuron (L_j) is greater than 0 (see equation (8.13)) and the net input of all other neurons $(L_i, i = 1 \ldots M, i \neq j)$ is less than 0 (see equation (8.14)). Thus, we see that pattern $\mathbf{X}^p$ is correctly classified even if the output is computed using the WTA strategy.

8.8 Constructive Learning Algorithms in Practice

The preceding discussion has focused on provably convergent constructive learning algorithms to handle real-valued multi-category pattern classification problems. The algorithms differ from one another chiefly in the criteria used to decide when and where to add a neuron to an existing network, and the method used to train individual neurons. A systematic experimental study of the constructive algorithms aimed at a thorough characterization of their implicit inductive, representational, and search biases (that arise from the construction procedures employed by the different algorithms) is beyond the scope of this chapter. Such a study would entail, among other things, a careful experimental analysis of each constructive algorithm for different choices of the single neuron training algorithm (e.g., *pocket algorithm* with *ratchet modification*, *thermal perceptron algorithm*, *barycentric correction procedure*, and perhaps other variants designed for synergy with specific network construction strategies) and different output representations (e.g., independent output neurons versus WTA). We present a more systematic comparison of the performance of different constructive learning algorithms in appendix B. This is the subject of [PYH98]. In what follows, we explore some practical

issues that arise in the application of constructive learning algorithms and present the results of a few experiments designed to address the following key issues.

1. The convergence proofs presented here rely on two factors: The ability of the network construction strategy to connect a new neuron to an existing network so as to guarantee the existence of weights that will enable the added neuron to improve the resulting network's classification accuracy and the TLU weight training algorithm's ability to find such a weight setting. Finding an optimal weight setting for each added neuron such that the classification error is maximally reduced when the the data is non-separable is an NP-hard problem [SRK95]. Thus, practical algorithms for training threshold neurons are heuristic in nature. This makes it important to study the convergence of the proposed constructive algorithms in practice. We trained constructive networks on several non-linearly separable datasets that require highly nonlinear decision surfaces.

2. It is important to examine whether constructive algorithms yield in practice, networks that are significantly smaller than would be the case if a new neuron is recruited to memorize each pattern in the training set. A comparison of the size of the networks generated by the algorithms with the number of patterns in the training set would at least partially answer this question.

3. Regardless of the convergence of the constructive learning algorithms to zero classification errors, a question of practical interest is the algorithms' ability to improve generalization on the test set as the network grows in size. One would expect *over-fitting* to set in eventually as neurons continue to get added in an attempt to reduce the classification error, but we wish to examine whether the addition of neurons improves generalization before over-fitting sets in. Experiments were designed to examine the generalization behavior of constructive algorithms on non-linearly separable datasets.

Another important issue, especially in the case of large pattern sets, is that of training time. Since our experiments were not designed for optimal performance in terms of training time,

it is difficult to make a definitive statement comparing the training speeds of the different algorithms. We have identified some important factors that address this issue.

8.8.1 Datasets

We have conducted several experiments with constructive learning algorithms using a variety of *artificial* and *real world* datasets. A detailed specification of these datasets is given in appendix B (see Table B.1). In this chapter we describe experiments with the 5 bit random patterns (*r5*), three concentric circles (*3c*), ionosphere (*ion*), segmentation (*seg*), iris (*iris*), wine (*wine*), and sonar (*sonar*) datasets. Given that the *seg* and *wine* datasets involve attributes with high magnitudes, we used normalized versions of these datasets in our experiments.

8.8.2 Training Methodology

Any of the three TLU weight training schemes can fit the role of $\mathcal{A}$ for the *tower*, *pyramid*, *tiling*, *upstart*, and *perceptron cascade* algorithms. Initially, the *thermal perceptron algorithm* was used for training weights of the individual TLUs. The weights of each neuron were randomly initialized to values between -1 and $+1$. The number of training epochs was set to 500. Each epoch involves presenting a set of l randomly drawn patterns from the training set where l is the size of the training set. The initial temperature T_0 was set to 1.0 and was dynamically updated at the end of each epoch to match the average net input of the neuron(s) during the entire epoch [Bur94]. 25 runs were conducted for each experimental set up. Training was stopped if the network failed to converge to zero classification errors after adding either 100 hidden neurons in a given layer or after training a total of 25 hidden layers and that particular run was designated as a failure. Following the training step, the network's generalization performance was measured on a set of test patterns (if one was available).

For *sequential* learning, the variation of the *barycentric correction procedure* which is specifically designed for exclusion of patterns can be used for training. Each hidden neuron was trained for 500 epochs of the *barycentric correction procedure* with the initial weighting coefficients set to random values between 1 and 3.

In the case of the *upstart* and *perceptron cascade* algorithms, some runs failed to converge to zero classification errors. Upon closer scrutiny it was found that the training sets of the daughter neurons had very few patterns with a target output of 1 (compared to the patterns with a target output of 0). The *thermal perceptron algorithm* while trying to correctly classify the largest subset of training patterns ended up assigning an output of 0 to all patterns. Thus it failed to meet the requirements imposed on $\mathcal{A}$ in this case. This resulted in the added daughter neuron's failure to reduce the number of misclassified patterns by at least one and in turn caused the *upstart* and the *perceptron cascade* algorithms to keep adding daughter neurons without converging. To overcome this problem, a *balancing* of the training set for the daughter neuron was performed as follows: The daughter neuron's training set was balanced by replicating the patterns having target output 1 sufficient number of times so that the dataset has the same number of patterns with target 1 as with target 0. Given the tendency of the *thermal perceptron algorithm* to find a set of weights that correctly classify a near-maximal subset of its training set, it was now able to (with the modified training set) at least approximately satisfy the requirements imposed on $\mathcal{A}$.

8.8.3 Convergence Properties

Tables 8.1, 8.2, and 8.3 summarize the performance of the constructive algorithms on the *r5*, *3c* and *ion* datasets respectively. In each case, the networks were trained to attain 100% classification accuracy on the training set and the network size (number of neurons excluding the input neurons), training time (in seconds), and generalization accuracy (the fraction of the test set that was correctly classified by the network) were recorded. These tables demonstrate that the constructive algorithms are indeed capable of converging to zero classification errors while generating sufficiently compact networks.

Certain constructive algorithms experienced difficulty in successfully classifying the entire training set in the case of some datasets (e.g., *iris* and *seg*). In Tables 8.4 and 8.5 we describe the results of those constructive algorithms that did manage to converge successfully to zero classification errors on these training sets.

Table 8.1 Performance of the Constructive Algorithms on the *r5* Dataset.

Algorithm	Network Size	Time
Tower	11.03±0.73	10.90±0.95
Pyramid	10.58±0.88	10.63±1.21
Upstart	10.64±0.57	52.34±4.83
Cascade	9.78±0.4	49.33±2.68
Tiling	14.95±1.17	9.49±0.76
Sequential	10.92±0.62	65.23±6.98

Table 8.2 Performance of the Constructive Algorithms on the *3c* Dataset.

Algorithm	Network Size	Training Time	Test Accuracy
Tower	6.00±0.00	134.96±1.52	99.79±0.09
Pyramid	6.00±0.00	137.43±3.36	99.74±0.23
Upstart	19.00±15.53	1227.56±774.52	99.03±0.76
Cascade	8.38±5.52	690.28±533.57	99.14±1.15
Tiling	45.60±7.76	561.44±71.32	95.37±0.92
Sequential	44.68±6.26	1550.90±1282.35	94.44±1.13

Owing to the inherent bias of the network construction strategy, there might be a particular network construction strategies that are favorably disposed towards certain datasets. This fact is evident from the table 8.4 where we see that only the *tiling* and *sequential* algorithms have converged on the *iris* datasets and table 8.5 which shows that only the *perceptron cascade*, *tiling*, and *sequential* algorithms have been successful on the *seg* dataset.

Projecting individual patterns on to a parabolic surface by appending an additional attribute also causes some practical difficulties. Certain real world datasets have patterns with

Table 8.3 Performance of the Constructive Algorithms on the *ion* Dataset.

Algorithm	Network Size	Training Time	Test Accuracy
Tower	5.68±1.65	97.94±28.25	94.8±1.52
Pyramid	5.04±0.98	90.16±19.03	94.84±1.17
Upstart	3.04±0.45	133.77±28.39	94.12±1.89
Cascade	3.28±0.61	148.56±39.17	93.03±2.10
Tiling	8.76±1.48	86.99±9.43	89.64±3.46
Sequential	5.08±0.4	106.17±29.39	91.62±2.53

Table 8.4 Performance of the Constructive Algorithms on the *iris* Dataset.

Algorithm	**Network Size**	**Training Time**	**Test Accuracy**
Tiling	9.76±3.27	17.19±2.16	96.08±1.35
Sequential	7.0±0.0	80.8±26.24	90.32±0.75

Table 8.5 Performance of the Constructive Algorithms on the *seg* Dataset.

Algorithm	**Network Size**	**Training Time**	**Test Accuracy**
Cascade	20.96±2.72	490.67±106.98	74.43±2.15
Tiling	53.41±19.39	174.65±60.93	83.87±1.78
Sequential	29.48±2.18	1156.72±188.59	81.57±2.38

large magnitude attributes. Since the correctness proofs of the *tower*, *pyramid*, *upstart*, and *perceptron cascade* algorithms require augmentation of the dataset with an additional attribute representing the sum of squares of the individual attributes, this additional attribute is often very large in magnitude. Such high magnitude attributes would cause an excruciating slow down in the training of the constructive algorithms. One solution to this problem is to normalize the patterns so that each pattern vector has a magnitude of 1. In appendix A we show how the convergence proofs of the constructive algorithms can be modified to deal with normalized pattern vectors.

In practice, the success of constructive learning algorithms is critically dependent on the performance of the TLU weight training method ($\mathcal{A}$). The close interaction between the network construction process and the training of individual TLUs is demonstrated by our experiments with the *wine* dataset. When the *thermal perceptron algorithm* was used to play the role of $\mathcal{A}$ none of the constructive learning algorithms were able to converge. Replacing the *thermal perceptron algorithm* by the *barycentric correction procedure* produced entirely different results with all except the *pyramid* algorithm converging to zero classification errors fairly quickly. These results are summarized in Table 8.6.

Similarly, experiments with the *sonar* dataset revealed that a single TLU trained using the *pocket algorithm* with *ratchet modification* could correctly classify the entire training set

Table 8.6 Performance of the Constructive Algorithms on the *wine* Dataset.

Algorithm	**Network Size**	**Training Time**	**Test Accuracy**
Tower	12.24±0.83	56.38±5.66	92.76±1.72
Upstart	14.76±10.17	491.18±505.99	90.48±3.76
Cascade	16.36±3.9	557.97±190.84	89.52±5.67
Tiling	7.56±0.51	24.54±1.43	93.04±3.67
Sequential	7.4±1.12	129.04±29.71	93.24±4.75

i.e., the dataset is linearly separable. Even after training a TLU for 1000 epochs using the *thermal perceptron algorithm* and the *barycentric correction procedure* the separating weight vector was not found.

Another important factor which affects convergence of the constructive algorithms in the case of datasets with multiple output categories is the WTA training strategy. Tables 8.7 and 8.8 below summarize the performance of the constructive algorithms on the *iris* and the *seg* datasets using the WTA output strategy. We observe that for the *seg* dataset the *upstart* algorithm converges using the WTA output strategy whereas its convergence using the independent output computation was not possible (see Table 8.5).

Table 8.7 WTA Output Strategy on the *iris* Dataset.

Algorithm	**Network Size**	**Training Time**	**Test Accuracy**
Tiling	8.0±0.0	29.17±0.99	95.92±0.7
Sequential	7.0±0.0	126.06±43.2	90.4±0.82

Table 8.8 WTA Output Strategy on the *seg* Dataset.

Algorithm	**Network Size**	**Training Time**	**Test Accuracy**
Upstart	14.76±1.94	292.86±72.35	86.77±1.47
Cascade	13.88±1.13	269.12±44.26	86.79±1.52
Tiling	30.32±4.34	153.85±21.46	86.81±1.25
Sequential	30.16±3.2	1997.75±489.26	83.64±1.97

8.8.4 Network Size

A major motivation for exploring constructive learning algorithms is their ability to generate parsimonious networks. The convergence proofs for constructive algorithms are existence proofs and are based on the ability of each added neuron to reduce the classification error by at least one. A trivial network construction process of assigning one neuron per pattern would achieve zero classification errors. In this case, neither the network size nor the generalization performance of the resultant network would be satisfactory. We argue that in practice the algorithms we have presented perform much better. A comparison of the average network sizes (see Tables 8.1 — 8.8), in the cases where the networks generated actually converged to zero training errors, to the total size of the training set (see Table B) demonstrates that the networks generated were compact in the sense that the constructive algorithms did not simply memorize the training patterns by assigning a single hidden node to classify each pattern.

The average network sizes generated for the *seg* dataset with and without the WTA output strategy (see Tables 8.5 and 8.8 respectively) shows one case wherein the WTA output strategy yields substantially smaller networks.

8.8.5 Generalization Performance

Although convergence and network size are important parameters of constructive algorithms, generalization is a more meaningful yardstick for measuring their performance. A single layer of TLUs when trained has a certain generalization ability. Of course, this single layer of TLUs cannot converge to zero classification errors in the case of non-linearly separable training sets. A constructive algorithm can generate a network with zero classification errors on non-linearly separable sets. However, in cases where the size of the training set is small or there is noise in the training data the use of the constructive algorithm might result in over-fitting. The added neurons might effectively memorize a few patterns misclassified by the first layer of TLUs. When this happens, the generalization performance of the resulting networks can be worse than that of the single layer network.

We trained a single layer of TLUs using the *thermal perceptron algorithm* on each of the

datasets mentioned in section 8.8.1. The following parameters were used in the simulation runs: 500 training epochs, initial temperature $T_0 = 1$, adaptive re-scaling of T_0 following each epoch, random initial weights between $[-1..1]$, and a learning rate $\eta = 1$. We measured the total time it took to train the TLUs for 500 epochs and recorded the training and test accuracies at the end of 500 epochs. Table 8.9 describes the performance of a single layer of TLUs on the different datasets.

Table 8.9 Single Layer Training using the *thermal perceptron algorithm*.

Dataset	**Training Accuracy**	**Training Time**	**Test Accuracy**
r5	56.37±2.54	3.26±0.08	—
r5 (WTA)	75.78±1.18	4.26±0.05	—
3 circles	23.11±9.76	102.40±2.00	22.26 ± 9.48
3 circles (WTA)	44.86±4.13	126.70±1.86	42.46 ± 4.20
Iris	78.36±0.95	8.17±0.1	72.64±1.7
Iris (WTA)	99.0±0.0	11.9±0.17	98.0±0.0
Segmentation	82.59±0.7	45.95±0.94	71.44±0.67
Segmentation (WTA)	94.31±0.57	50.46±0.84	87.39±0.42
Ionosphere	95.42±0.59	17.46±0.15	92.99±1.93

A significant increase in generalization performance is observed for the *3c* (see Table 8.2), the *iris* (see Table 8.4), and the *seg* (see Table 8.5) datasets with independent training. The performance on the *ion* dataset improved only marginally (see Table 8.3). When the WTA training was employed the performance of the constructive algorithms on the *iris* and *seg* datasets actually deteriorated (see Tables 8.7 and 8.8).

In summary, given adequate training data, constructive algorithms can yield relatively compact networks that significantly outperform the single layer networks for the same task in terms of generalization accuracy. However, in practice, it might be necessary to terminate the network construction algorithm before over-fitting sets in.

8.8.6 Training Speed

The issue of network training time becomes critical for very large training sets. We have measured the average training time for each dataset (see Tables 8.1 — 8.9). Below we discuss

some factors that affect the training time. We must point out that our simulation programs did not contain any special optimization beyond the facilities provided by the compiler and standard techniques for enhancing the run time performance of the programs.

A comparison of the average training time across different algorithms clearly shows that the *tower*, *pyramid*, and *tiling* algorithms are able to learn relatively faster as compared to the *upstart*, *perceptron cascade*, and *sequential* algorithms. This can be explained in terms of the operational characteristics of the algorithms. The *upstart* and *perceptron cascade* algorithms require re-training of the output weights after each daughter neuron is added and trained. This computation is fairly time consuming especially since the fan in of the output neurons increases with the addition of each new daughter. The *sequential* learning algorithm is limited by the fact that the only suitable TLU weight training algorithm available to exclude patterns belonging to a single class is a variant of the *barycentric correction procedure*. The multi-category extension of this procedure involves running the two-category version for each of the output classes which explains why the *sequential* learning algorithm learns very slowly for pattern sets involving large number of output classes. Faster learning in the *tower* and *pyramid* is attributed to the fact that each layer of the network is trained just once and the weights are frozen. In the case of the *tiling* network, in addition to the fact that the neurons are trained only once, the training set sizes for the ancillary neurons progressively decrease as additional ancillary neurons get added. Since each neuron or group of neurons are trained for 500 epochs irrespective of the training set size, smaller training sets obviously require less training time than larger ones. The same advantage holds for *sequential* learning.

8.9 Summary and Discussion

Constructive algorithms offer an attractive approach for automated design of neural networks. In particular, they eliminate the need for *ad hoc*, and often inappropriate, *a-priori* choice of network architecture; potentially provide a means of constructing networks whose size (complexity) is commensurate with the complexity of the pattern classification task at hand; and offer natural ways to incorporate prior knowledge (e.g., in the form of classification

rules, decision trees, etc.) to guide learning. In this chapter, we have focused on a family of such algorithms that incrementally construct networks of threshold neurons. Although a number of such algorithms have been proposed in the literature, most of them are limited to 2-category pattern classification tasks with binary/bipolar valued input attributes. This chapter extends several existing constructive learning algorithms to handle multi-category classification for patterns having real-valued attributes. We have provided rigorous proofs of convergence to zero classification errors on finite, non-contradictory training sets for each of the multi-category algorithms proposed in this chapter. Our proof technique provides a sufficiently general framework to prove the convergence of several different constructive algorithms. This strategy will be useful in proving the convergence properties of constructive algorithms designed in the future.

The convergence of the proposed algorithm to zero classification errors was established by showing that each modification of the network topology guarantees the existence of a weight setting that would yield a classification error that is less than that provided by the network before the modification and assuming a weight modification algorithm $\mathcal{A}$ that would find such a weight setting. We do not have a rigorous proof that any of the graceful variants of perceptron learning algorithms that can in practice, satisfy the requirements imposed on $\mathcal{A}$, let alone find an *optimal* (in some suitable well-defined sense of the term - e.g., so as to yield minimal networks) set of weights. The design of suitable TLU training algorithms that (with a high probability) satisfy the requirements imposed on $\mathcal{A}$ and are at least approximately optimal remains an open research problem. Against this background, the primary purpose of the experiments described in section 8.8 was to explore the actual performance of such multi-category constructive learning algorithms on some non-linearly separable classification tasks if we were to use a particular variant of perceptron learning for non-linearly separable datasets. Detailed theoretical and experimental analysis of the performance of single threshold neuron training algorithms is in progress [YPH98a]. We expect this analysis to lead to the design of improved and possibly hybrid weight modification schemes that can dynamically adapt to the situation faced by the particular constructive algorithm on a given dataset. For example, in certain pattern configurations it might be appropriate to exclude as many patterns of one class

as possible whereas in other scenarios it might be better to correctly classify as large a subset of the training patterns as possible.

Simulation results have demonstrated the usefulness of the constructive algorithms in classification tasks. Some of the issues addressed in the preceding sections set the stage for a detailed evaluation of the design choices that affect the performance of the constructive learning algorithms and identify several avenues for further research. The impact of these issues on the training efficiency (network size and training time) and the generalization ability merit further investigation.

- A cross-validation based criterion for training constructive networks must be employed wherein the training is stopped when the network's generalization begins to deteriorate after the addition of a new neuron (or a group of neurons). It is likely to generate compact networks that exhibit good generalization properties with relatively little training as opposed to the current stopping-criterion of zero classification errors which might lead to over-fitting of the training set.

- Hybrid network training schemes that dynamically select an appropriate network construction strategy, an appropriate TLU weight training algorithm, an appropriate output computation strategy and such to obtain locally optimal performance at each step of the classification task are likely to yield superior performance across a variety of datasets.

- Post-processing techniques such as *pruning* of networks eliminate nodes and connections that do not adversely affect the network's performance. Pruning can potentially overcome the over-fitting problem by yielding more compact networks with superior generalization. An application of neuron pruning techniques to the *MTiling* networks is described in chapter 9.

- Various pre-processing techniques are responsible for transforming the training data in a manner that might simplify the learning task. Among these we have already seen the benefits of normalization. Another method of handling pattern sets with real-valued outputs is quantization of the training patterns. Preliminary results of applying quantization

are presented in [YH96]. In chapter 10 we describe a novel adaptive *vector quantization* scheme based on the *MTiling* algorithm.

- Constructive neural network learning algorithms provide a natural framework for incorporating domain specific prior knowledge in the network topology. This domain knowledge can be refined and/or augmented by dynamically adding more neurons to the original network. This framework for constructive theory refinement in knowledge based neural networks is studied in chapter 10.

- Each constructive algorithm has its own set of inductive and representational biases implicit in the design choices that determine when and where a new neuron is added and how it is trained. A systematic characterization of this bias would be useful in guiding the design of constructive algorithms that exhibit improved performance.

- The differences in the training time of the various constructive algorithms are striking. This may be due, among other things, to the differences in their inductive and representational biases. However, it might be possible in some cases to optimize each algorithm separately to reduce its training time.

- It is often the case that the generalization performance of inductive learning algorithms can be substantially improved by augmenting them with suitable algorithms for selecting a relevant subset of a much larger set of input attributes many of which might be irrelevant or noisy. A variety of feature subset selection algorithms have been proposed in the literature on pattern recognition [Rip96]. The effectiveness of genetic algorithms for feature subset selection has been demonstrated by [YH97]. Against this background, exploration of constructive learning algorithms augmented with suitable feature subset selection techniques might be of interest.

- The results of a more extensive experimental comparison of the different constructive learning algorithms is presented in appendix B. Yang *et al* have recently proposed an efficient inter-pattern distance based constructive learning algorithm DistAl [YPH98b]. Unlike the algorithms described in this chapter, DistAl does not use the perceptron style

iterative weight update procedure. Instead, it constructs spherical threshold neurons whose weights are determined by the inter-pattern distances to sequentially exclude patterns belonging to a single class from all others (as is the case in the *sequential* learning algorithm). A comparison of the constructive learning algorithms proposed in this chapter with the DistAI, the *cascade correlation* algorithm, and the *backpropagation* learning algorithm would be useful in gaining a better understanding of the advantages and disadvantages of each approach.

- Recent research has focussed on the use of neural networks for *lifelong learning* [Thr95] where networks are trained to learn multiple classification tasks one after the other. A goal of the multi-task learning system is to exploit (if possible) the prior knowledge acquired while learning the earlier tasks to make the learning of the later and possibly more difficult tasks easier. Constructive learning algorithms offer an interesting approach for the use of domain knowledge to learn multiple classification tasks. A network that has domain knowledge from the simpler task(s) built into its architecture (either by explicitly setting the values for the connection weights or by training them) can form a building block for a system that constructively learns more difficult tasks. The performance of constructive learning algorithms in this setting of lifelong learning merits further study.

9 PRUNING STRATEGIES FOR THE MTILING CONSTRUCTIVE LEARNING ALGORITHM

9.1 Introduction

Constructive neural network learning algorithms offer an interesting paradigm for incremental construction of near-minimal architectures for pattern classification problems [Gal90, Hon90, Gal93, HU93]. As we saw in chapter 8, constructive learning algorithms enjoy several advantages over the traditional algorithms for learning in multi-layer feed-forward networks. Several constructive learning algorithms have been proposed in the literature — *tower*, *pyramid* [Gal90], *tiling* [MN89], *upstart* [Fre90b], *perceptron cascade* [Bur94], and *sequential* [MGR90]. These algorithms differ from each other in the design choices viz. representation of input patterns (binary/bipolar valued or real-valued); when and where to add a new TLU (or a group of TLUs); connectivity of the newly added neuron(s); algorithms for training the TLUs; and the strategy for training the sub-network affected by the modification of the network topology. These differences in design choices result in constructive learning algorithms with different representational and inductive biases. Provably correct and practical extensions of these algorithms to handle real-valued pattern attributes and multiple output categories were described in chapter 8.

The success of a constructive learning algorithm depends partly on the algorithm used to train the individual TLUs because the convergence to zero classification errors is based on the fact that the TLU weight training algorithm can find a suitable weight setting such that the total number of mis-classifications is reduced by at least one each time a new neuron (or a group of neurons) is added to the network and trained. Algorithms such as the *pocket algorithm* with *ratchet modification* [Gal90], the *thermal perceptron algorithm* [Fre92], and the

barycentric correction procedure [Pou95] are commonly used for training individual TLUs (or groups of TLUs) in constructive learning algorithms. We denote such a suitable TLU training algorithm by $\mathcal{A}$.

Given a particular pattern classification task it is the goal of a neural network learning algorithm to search the space of neural network architectures to determine an architecture suitable for the task. An exhaustive search through the space of neural network architectures is computationally infeasible. Constructive learning algorithms adopt a greedy strategy in that each incrementally added neuron attempts to reduce as large a fraction of the network's residual classification error as possible. The training of individual TLUs is based on local information in the sense that during training the weights of the remainder of the network are frozen and the training set for these neurons is constructed with the objective of reducing the residual classification error. Owing to the *representation* and *inductive biases* introduced in the design choices incorporated in the constructive learning algorithm and the *locality of training*, it is possible that the incrementally grown networks are larger than necessary for the given classification task. Other things being equal, smaller (more compact) networks are desirable because of lower classification cost; potentially superior generalization performance; and transparency of the acquired knowledge in applications which involve extraction of rules from trained networks. These reasons motivate the study of pruning techniques in constructive learning algorithms.

Network pruning involves elimination of connection elements (i.e., weights or neurons) that are deemed unnecessary in that their elimination does not degrade the network's performance. Pruning can be performed either after the entire network is trained or can be integrated into the training process itself. In this chapter we study the application of pruning techniques to *MTiling*, an extension of the *tiling* algorithm to handle real-valued pattern attributes and multiple output classes (see section 8.6). The remainder of this chapter is organized as follows: Section 9.2 describes three elementary pruning strategies for eliminating unwanted neurons from a *MTiling* network. Section 9.3 presents the results of experiments with pruning using several artificial and real-world datasets. Finally, section 9.4 concludes with an analysis of the

experiments with pruning and suggests directions for future research.

9.2 Pruning Strategies

An excellent survey of neural network pruning strategies appears in [Ree93]. It outlines two types of pruning techniques for feed forward neural networks trained using the backpropagation algorithm — *sensitivity calculations* and *penalty terms*. The former investigates the sensitivity of the error function (or the objective function that is minimized) to the removal of a network element. Elements with the least sensitivity are pruned. The second group of techniques involves incorporating a penalty term in the error function which is minimized by the gradient descent based learning algorithm. For example, incorporating a term proportional to the sum of all the weight magnitudes in the error function favors solutions in which the individual weights are small in magnitude. Weights that are nearly zero are not likely to influence the output much and so can be pruned. In general, sensitivity based techniques modify the network topology (i.e., remove redundant nodes and weights based on sensitivity calculations) whereas the penalty term based methods modify the cost function so that the learning algorithm while minimizing the cost function forces the irrelevant connection weights to be driven toward zero. The group of constructive algorithms mentioned in section 9.1 does not explicitly define a cost (error) function for minimization during training. Thus, it is not clear whether penalty term based pruning techniques can be directly applied to these constructive learning algorithms. In this chapter, we focus on the sensitivity based pruning of neurons in *MTiling* networks.

9.2.1 Pruning in MTiling Networks

Recall from section 8.6 that the *MTiling* network is a strictly layered network of TLUs. Each layer has a group of M master neurons (where M is the number of output categories specified in the pattern classification task) and a set of 0 or more ancillary neurons that are trained to ensure that each layer attains a faithful output representation for all the patterns in the dataset. The topmost layer of the network (see Fig. 8.14) is the output layer and it contains only M master neurons.

The sensitivity of a neuron is defined as the *error* introduced in the network upon the removal of the neuron. The error can be defined in a manner that is most suited to the context. In the case of *MTiling* networks, since ancillary neurons are added and trained to ensure faithfulness of current layer, we assign a sensitivity value of 0 to neurons whose removal does not cause the layer to become unfaithful. More precisely, we define the sensitivity $S(i)$ of a neuron i in a layer L immediately after the layer has been made faithful as follows:

$$S(i) = 1 \text{ if eliminating } i \text{ renders } L \text{ unfaithful}$$

$$S(i) = 0 \text{ otherwise}$$

In *MTiling* networks we integrate pruning with the training process and invoke the pruning phase after each layer of the network is trained and made faithful. Since the master neurons of each layer are the output neurons for that layer we assign a sensitivity of 1 to the master neurons thereby preventing them from being pruned. The ancillary neurons are assigned sensitivity of 1 or 0 as described below. All the ancillary neurons that have sensitivity 0 are pruned from the network. The training of the network is then continued with the addition of a new layer with M master neurons.

Dead Neurons:

Ancillary neurons with exactly the same output (i.e., 1 or -1) for all patterns in the training set are called *dead neurons*. As seen in Fig. 9.1, pruning dead neurons does not affect the faithfulness of the current layer. Thus, dead neurons are assigned a sensitivity of 0.

Correlated Neurons:

Pairs of ancillary neurons that have either exactly the same or exactly the opposite output in response to each pattern in the training set (i.e., the product of the outputs of the two neurons is either 1 for all patterns or -1 for all patterns) are said to be correlated. As seen in Fig. 9.2, the first two neurons have exactly opposite outputs on all training patterns and thus satisfy the test for correlated neurons. One of these neurons can be safely pruned without affecting the faithfulness of the current layer. Ancillary neurons are taken two at a time and their outputs (for each pattern) are compared to determine if the neurons are correlated. One neuron from the correlated pair is dropped as soon as it is identified and is not considered any further in the

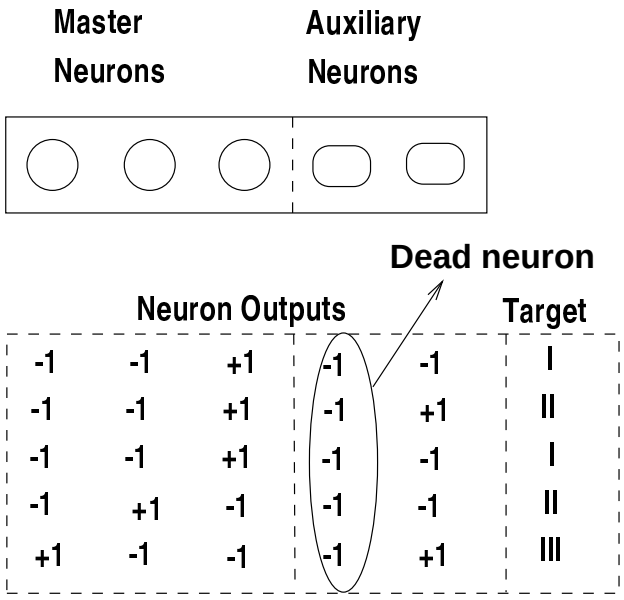

Figure 9.1 Dead Neurons.

search for correlated neuron pairs. Note that in this case we only consider perfect correlation among neurons i.e., neurons having exactly the same or exactly the opposite outputs on all the patterns belonging to the training set.

Redundant Neurons:

Determination of redundant neurons involves dropping ancillary neurons one at a time and comparing the remaining outputs for faithfulness. If the outputs are not faithful then the dropped neuron is restored. Otherwise the dropped neuron is redundant and is assigned a sensitivity $S(i) = 0$ (see Fig. 9.3). The redundant neuron is immediately pruned and the search for redundant neurons is continued starting with the first ancillary neuron. This identification and pruning of redundant neurons is continued until no further redundant neurons can be identified.

9.2.2 Pruning Cost

The search for ancillary neurons with sensitivity $S(i) = 0$ incurs an additional cost. The identification of neurons that are eventually pruned involves a comparison of the current layer's outputs in response to every training pattern. It should be noted that this output computation

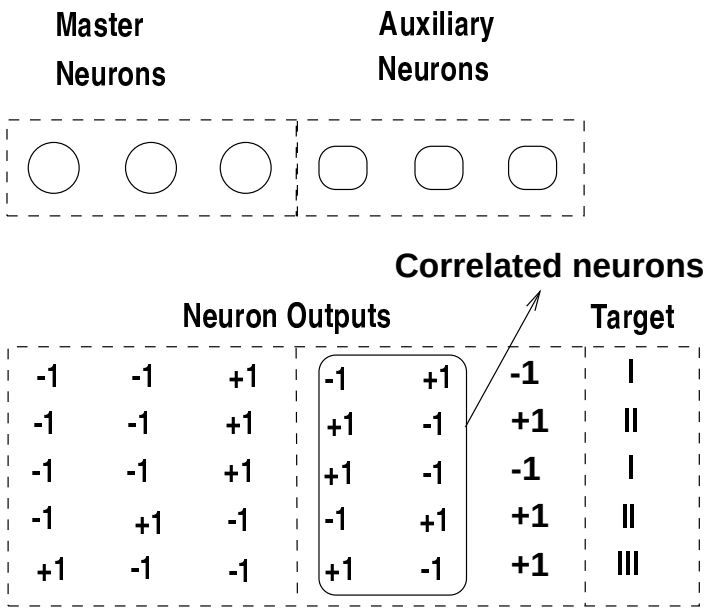

Figure 9.2 Correlated Neurons.

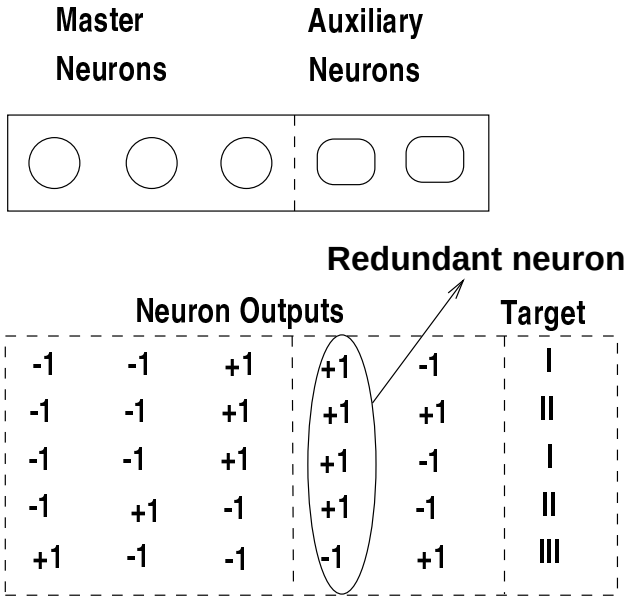

Figure 9.3 Redundant Neurons.

is performed by the *MTiling* algorithm while checking for faithfulness of the current layer. These output values can thus be made readily available to the pruning step. Let us now analyze the cost involved in searching for the ancillary neurons with $S(i) = 0$.

Let k be the number of ancillary neurons in the current layer and $|S|$ be the total number of training patterns. The search for dead neurons involves comparing the output of each neuron in response to each of the $|S|$ training patterns. This step therefore takes $O(k \cdot |S|)$ time. The identification of correlated neurons involves comparing the outputs of the neurons taken two at a time. Each such comparison takes $O(|S|)$ time. Thus, the worst case time complexity for the search for correlated neurons is $O(k^2 \cdot |S|)$. The process of determining redundant neurons involves dropping the ancillary neurons one at a time and testing the outputs of the remaining neurons for faithfulness. If a redundant neuron is found then it is immediately pruned and the search for additional redundant neurons is started again from the first ancillary neuron. Thus, in the worst case this search for redundant neurons makes $O(k^2)$ calls to the routine that checks for faithfulness of the current layer. Checking for faithfulness takes $O(k \cdot |S|)$. This means that the worst case time complexity for this final step is $O(k^3 \cdot |S|)$. It must be noted that the only operation involved in the above three search strategies is an equality comparison of integers which in practice can be performed very efficiently. We see in section 9.3 that in practical experiments the total time for pruning is a small fraction (about 10%) of the total training time for the *MTiling* network.

9.3 Experimental Results

We have conducted several experiments with pruning using a variety of *artificial* and *real-world* datasets. Specifically, we used the *3c*, *2sp*, *liver*, *seg*, *wdbc*, and *wine* datasets. A detailed specification of these datasets is given in Table B.1 (see appendix B).

We performed 10 runs of the *MTiling* algorithm on each of the above mentioned datasets with and without pruning. Individual TLUs were trained using the *thermal perceptron algorithm* for 500 epochs. The initial temperature T_0 was set at 1.0, the learning rate η was set at 1, and initial weights of each TLU were initialized to random values in the range $[-1..1]$. The

winner-take-all (WTA) strategy was used to compute the outputs for datasets involving more than two pattern classes. On each run the network was trained until it achieved zero classification errors on the training set. If a test set was available for the dataset then the network's generalization performance was determined by measuring network's classification accuracy on the test set. For runs with network pruning the number of neurons pruned by each of the three pruning strategies, the total time for pruning (in seconds), the network size (number of hidden and output neurons), the total training time (in seconds), and the generalization performance (classification accuracy on the test set) over the 10 runs were recorded. For runs without network pruning the network size, the training time, and the generalization performance over the 10 runs were recorded.

Table 9.1 reports the mean and standard deviation over 10 simulation runs of the number of neurons pruned by each of the three pruning strategies dead neurons, correlated neurons, and redundant neurons, the time expended on the pruning step in seconds and the total training time in seconds for the *MTiling* algorithm. The results show that a majority of the neurons pruned belong to the class of redundant neurons. Further, the total time spent in searching for neurons with $S(i) = 0$ is a small fraction of the total training time in each case.

Table 9.1 Results of Pruning using the *thermal perceptron algorithm*.

Dataset	**Dead neurons**	**Correlated neurons**	**Redundant neurons**	**Pruning Time**	**Training Time**
3c	0.1 ± 0.3	0.0 ± 0.0	5.2 ± 1.4	133.8 ± 22.9	856.3 ± 94.3
2sp	2.2 ± 1.6	0.6 ± 0.8	16.0 ± 5.8	13.3 ± 2.3	134.8 ± 16.1
liver	0.0 ± 0.0	0.1 ± 0.3	5.5 ± 3.1	7.2 ± 1.7	156.8 ± 23.3
seg	0.6 ± 0.5	0.0 ± 0.0	2.3 ± 1.1	4.01 ± 1.3	175.1 ± 21.1
wdbc	5.9 ± 4.6	0.0 ± 0.0	7.4 ± 4.0	23.21 ± 8.1	439.9 ± 67.8
wine	15.4 ± 14.0	4.8 ± 9.1	14.8 ± 16.1	8.45 ± 10.5	142.3 ± 90.3

Fig. 9.4 compares the average final network size of the *MTiling* algorithm with and without pruning. This demonstrates a modest to significant reduction in the size of the network with pruning. The generalization performance of the network is not affected by pruning as can be seen from Fig. 9.5.

On a few runs for the *wdbc* and the *wine* dataset the *MTiling* algorithm was unable to

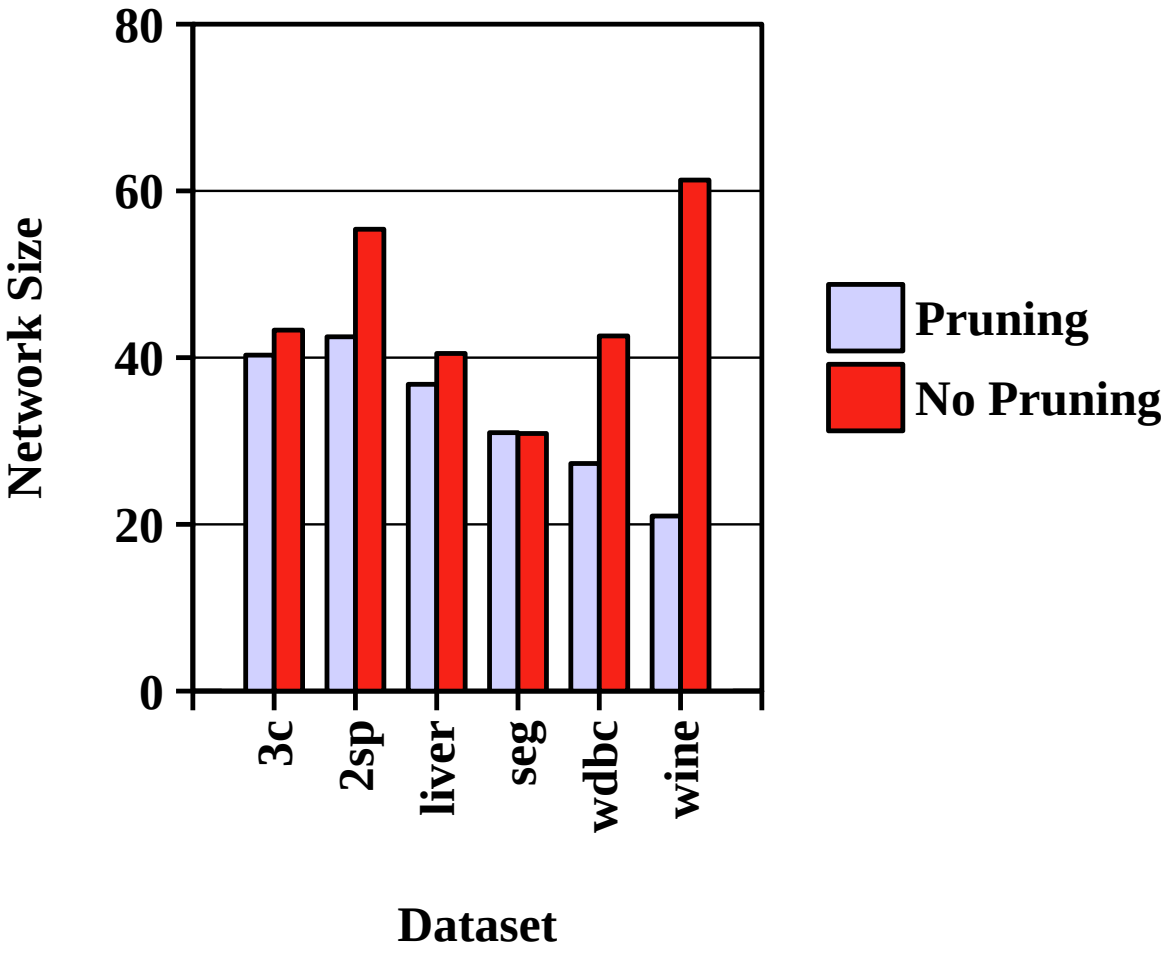

Figure 9.4 Comparing the Network Size with and without Pruning.

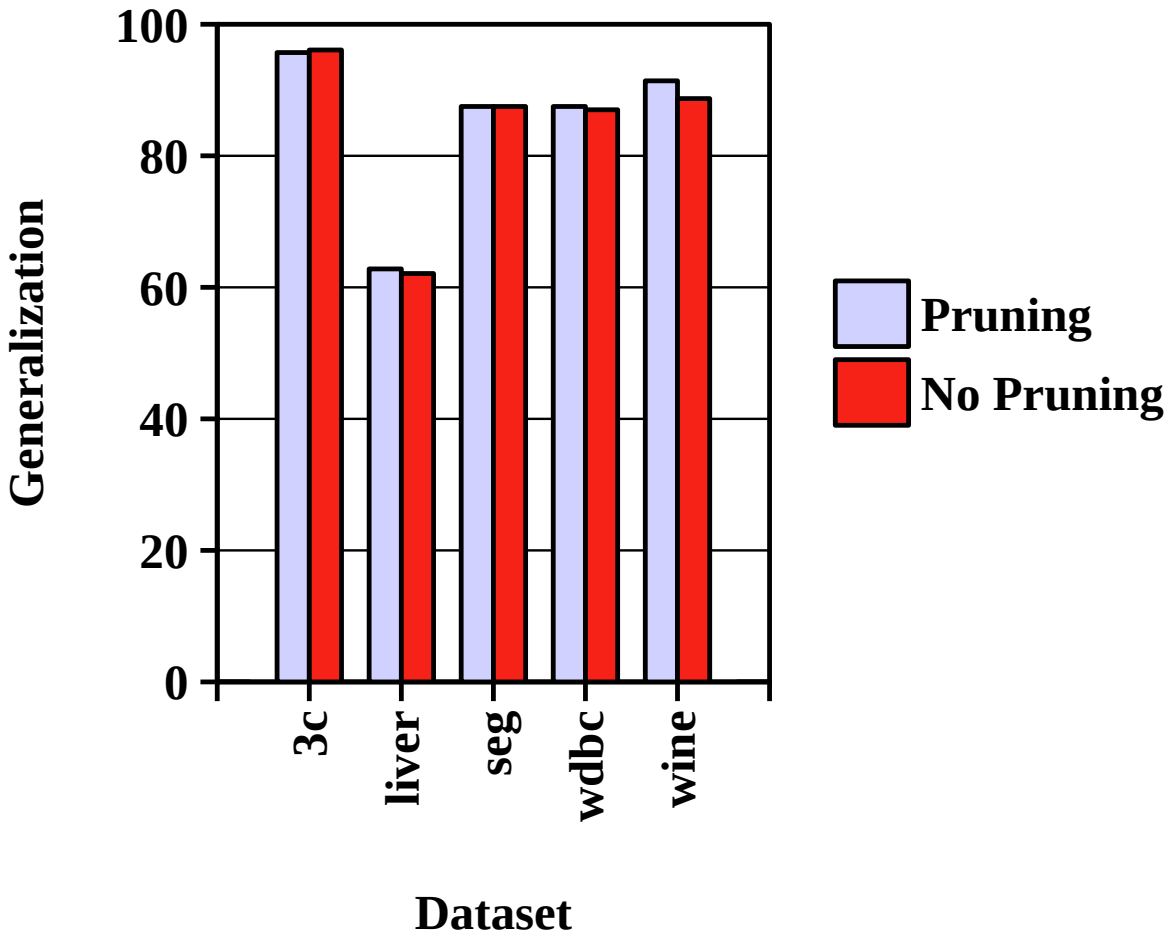

Figure 9.5 Comparing the Generalization with and without Pruning.

attain a faithful representation of the patterns at a layer even after the addition of 100 ancillary neurons. These runs were considered as failures and were not included in the results reported. We repeated the above experiments using the *barycentric correction procedure* instead of the *thermal perceptron algorithm* for training the individual TLUs. Since the extension of the *barycentric correction procedure* to WTA based output computation is extremely slow (see [YPH98a]) we performed these experiments using the independent output computation strategy. These results are summarized in Table 9.2. In the case of the *wdbc* and the *wine* datasets training using the *barycentric correction procedure* resulted in *MTiling* networks with practically no redundancy as opposed to the case of the *thermal perceptron algorithm* where a significant reduction in network size was attained as a result of pruning. The results of pruning on the other datasets were comparable to those obtained when the *thermal perceptron algorithm* was used for training TLUs.

Table 9.2 Results of Pruning using the *barycentric correction procedure*.

Dataset	Dead neurons	Correlated neurons	Redundant neurons	Pruning time	Training time
3c	0.0 ± 0.0	0.8 ± 0.8	3.8 ± 1.2	74.0 ± 22.5	779.0 ± 92.6
2sp	0.0 ± 0.0	0.0 ± 0.80	7.4 ± 3.1	9.3 ± 1.2	131.8 ± 17.3
liver	0.0 ± 0.0	0.0 ± 0.0	1.4 ± 0.8	2.5 ± 0.8	186.2 ± 29.5
seg	0.0 ± 0.0	0.0 ± 0.0	2.0 ± 0.9	2.6 ± 0.8	167.4 ± 27.9
wdbc	0.0 ± 0.0	0.0 ± 1.0	0.7 ± 0.0	20.6 ± 3.1	89.4 ± 1.5
wine	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	7.9 ± 1.5	94.0 ± 3.3

In Table 9.3 we present a comparison of the results of our experiments when the *thermal perceptron algorithm* is used to train the individual TLUs versus when the *barycentric correction procedure* is used. It lists the mean and standard deviation of the total number of neurons pruned (all the three strategies combined), the final network size, and the generalization accuracy on the test set (if one exists). These results are averaged over 10 simulation runs. Table 9.3 provides no conclusive evidence as to whether using *thermal perceptron algorithm* in *MTiling* networks is better than using *barycentric correction procedure* or vice-versa since results for both network size and generalization accuracy (test accuracy) show that the performance of the networks trained using *thermal perceptron algorithm* is superior on some

of the datasets whereas the performance of the networks trained using *barycentric correction procedure* is superior on other datasets. On all (except the *seg* dataset) the number of neurons pruned in networks trained using *thermal perceptron algorithm* is higher than that for networks trained using *barycentric correction procedure*. This points to the fact that in general networks trained using *barycentric correction procedure* tend to have much lesser redundancy.

Table 9.3 Comparing the Pruning Performance of Two TLU Training Algorithms.

Dataset	Thermal Perceptron			Barycentric Correction Procedure		
	Total Pruned	Network Size	Test Accuracy	Total Pruned	Network Size	Test Accuracy
3c	5.3 ± 1.5	40.3 ± 3.0	95.7 ± 0.7	4.6 ± 1.4	34.9 ± 2.9	96.7 ± 0.7
2sp	18.8 ± 7.0	42.5 ± 3.9	—	7.4 ± 3.1	46.6 ± 3.5	—
liver	5.6 ± 3.1	36.8 ± 4.7	62.8 ± 4.2	1.4 ± 0.8	52.3 ± 5.9	65.1 ± 3.9
seg	2.9 ± 1.1	31.0 ± 4.1	87.5 ± 1.2	2.0 ± 0.9	36.2 ± 8.2	84.1 ± 1.5
wdbc	13.3 ± 8.4	27.3 ± 2.9	87.5 ± 1.6	0.7 ± 1.1	20.6 ± 3.1	89.4 ± 1.5
wine	35.0 ± 38.6	21.0 ± 3.7	91.4 ± 3.2	0.0 ± 0.0	7.9 ± 1.5	94.0 ± 3.3

9.4 Discussion

Constructive neural network learning algorithms incrementally construct near minimal networks for pattern classification tasks. They employ a greedy search strategy in that each added neuron attempts to reduce as large a fraction of the network's residual classification error as possible. However, owing to the inherent biases of the network construction scheme, the limited training time allowed for each newly added TLU, and the locality of training which results from the fact that only the weights associated with the newly added TLU are trained whereas the rest of weights are kept frozen, it is possible that the eventual network constructed is larger than is actually necessary for the given classification task. Pruning techniques can be used to eliminate unwanted network elements (connection weights and neurons). Two broad categories of pruning strategies are the sensitivity based methods which estimate the sensitivity of a network element to the network's error and eliminate it if it has very low sensitivity and the penalty term methods which incorporate a penalty term in the error function that

forces unwanted connection elements to be driven to zero during training. Smaller (more compact) networks have several advantages such as lower classification cost, potentially superior generalization capability, and transparency of the acquired knowledge.

In this chapter we have designed three sensitivity based pruning strategies for eliminating unwanted neurons from *MTiling* networks. These pruning strategies are integrated with the network construction phase and are invoked after each layer is made faithful with respect to the set of training patterns. In particular, these methods identify *dead* neurons whose output is constant for the entire training set, *correlated* neurons whose outputs for each pattern are exactly the same or exactly the opposite, and *redundant* neurons whose elimination does not affect the faithfulness of the trained layer.

The experiments conducted on a variety of artificial and real-world datasets demonstrate a moderate to significant reduction in the network size as a result of pruning. The total time expended in identifying these unwanted neurons is roughly 10% of the total network training time. This approach thus presents a natural trade-off between training time and network size. We observed that the generalization performance of the networks with and without pruning did not differ significantly. This might be attributed to the fact that the pruning methods we studied simply eliminate the redundancy in the network. Other pruning strategies might however significantly affect the network's generalization performance.

The redundancy introduced in a *MTiling* network while learning to classify a particular dataset might actually depend on the choice of the TLU training algorithm (as seen in the results described in section 9.3). Specifically, we observed that *MTiling* networks trained using the *thermal perceptron algorithm* on the *wdbc* and the *wine* datasets contain a large number of irrelevant neurons that are eventually pruned. However, networks trained using *barycentric correction procedure* on the same datasets had very little or no redundancy. It is not clear whether there exists a single TLU training algorithm which when used for training TLUs in the *MTiling* network results in the construction of superior networks (in terms of network size and generalization ability) on all datasets. In the absence of any prior knowledge about the suitability of a particular TLU weight training algorithm for a given task it is advisable to

perform pruning to ensure that most of the redundancy in the network is eliminated. As we have noted earlier, the pruning operation increases the total training time by only a marginal amount.

Sensitivity based pruning of individual weights requires computing the network error after removing each weight independently. This is computationally infeasible even for moderately large networks. The characteristics of the *MTiling* algorithm can be used to identify *dominating* connection weights as follows. Since the *MTiling* network is strictly layered and uses bipolar TLUs (with outputs 1 or -1) in the hidden and output layers, the inputs for the TLUs in all the layers of the network starting with the second hidden layer are guaranteed to be bipolar valued (i.e., the inputs values can only be 1 or -1). Consider a TLU with the weight vector $\mathbf{W} = \{W_0, W_1, \ldots, W_n\}$. If this TLUs inputs are guaranteed to be bipolar valued, we say that the connection weight W_j is the dominating connection weight if $\mid W_j \mid > \sum_{i=0, i \neq j}^{n} \mid W_i \mid$. Note that if a TLU has a dominating connection weight then the output of the TLU is determined solely by the input to this dominating connection and does not depend on the inputs to the other connections. For example, if the input to the dominating connection is 1 and the dominating weight W_j is positive then it is easy to see that the output of the TLU will be 1 irrespective of the inputs to the other connections of the TLU. Thus, in the case of *MTiling* networks, if any TLU in the second hidden layer (or above) contains a dominating connection weight, then all connection weights except the dominating one for this TLU can be pruned.

The following are some interesting directions for further research:

- Design of more efficient pruning strategies for the *MTiling* networks that would potentially improve the generalization performance of the networks. Techniques that prune trained networks as well as those that integrate the network pruning with the training process merit further investigation.

- Design of a strategy to compute or approximate the sensitivity of the connection weights during the training itself might provide a more efficient method for pruning connection weights.

- Development of appropriate pruning techniques for the other constructive learning algorithms studied in chapter 8.

10 CONSTRUCTIVE THEORY REFINEMENT IN KNOWLEDGE BASED NEURAL NETWORKS

10.1 Introduction

Inductive learning systems attempt to learn a concept description from a sequence of labeled examples. The constructive neural network learning algorithms described in chapter 8 are typical inductive learning systems. Such systems have performed well in several application domains. However, these systems generalize from the labeled examples without knowing anything about why some particular example was assigned a given class label. Further, it is well known that the choice of the attributes to represent the examples can have a significant impact on the performance of the learning system [Rip96]. The presence of domain specific knowledge (domain theories) about the concept being learned can potentially enhance the performance of the inductive learning system. Hybrid learning systems that effectively combine domain knowledge with the inductive learning can potentially learn faster and generalize better than those based purely on inductive learning (learning from labeled examples alone). In practice the domain theory is often *incomplete* or even *inaccurate*. Inductive learning systems that use information from training examples to modify an existing domain theory by either augmenting it with new knowledge or by refining the existing knowledge are called *theory refinement* systems.

Theory refinement systems can be broadly classified into the following three categories:

- **Purely symbolic approaches**

  These methods use *symbolic* inductive learning algorithms (such as decision tree induction) for theory revision. Examples of such systems include RTLS [Gin90], EITHER [OM94], PTR [KFS94], and TGCI [DR95]. The EITHER system starts with the

given domain knowledge and a set of training examples. It divides the examples into two subsets depending on whether or not the rules in the domain theory are able to correctly classify them. It then uses the standard decision tree learning algorithm ID3 [Qui86] to invent new rules that correctly classify some of the previously misclassified training examples.

- **ILP based methods**

  Inductive Logic Programming (ILP) is an area of artificial intelligence research that combines techniques from machine learning with logic programming [Mug92]. It uses computational logic as the knowledge representation mechanism and extends the theory and practice of logic to the inductive (rather than the traditional deductive) model of inference. Theory refinement systems such as FOCL [PK92] and FORTE [RM95] use first-order logic as the representation scheme in theory revision and thus are said to belong to the class of ILP based techniques. FORTE (First-Order Revision of Theories from Examples) uses a hill-climbing search for refining first-order Horn-clause theories. It tackles new challenges such as logic program debugging and qualitative modeling that are presented by the first-order representation (and are beyond the reach of propositional systems). It identifies possible errors in the theory and calls on a library of operators to develop possible revisions. The best revision is implemented, and the process repeats until no further revisions are possible.

- **Connectionist strategies**

  Neural network based systems for theory refinement typically operate by first embedding the knowledge rules into an appropriate initial neural network topology. This domain knowledge is then refined by training the neural network on a set of labeled examples. Towell and Shavlik proposed the KBANN (knowledge based artificial neural network) learning algorithm for connectionist theory refinement [TSN90, TS94]. Their *rules-to-network* algorithm constructs an *AND-OR* graph representation of the initial domain knowledge and translates this graph to an appropriate neural network topology. KBANN then uses the standard backpropagation learning algorithm [RHW86] to refine the do-

main knowledge. The approaches described by Fu [Fu89] and Katz [Kat89] are similar to the KBANN algorithm. Unlike the symbolic and ILP based methods for theory refinement, the connectionist approaches require that the domain knowledge be translated into an appropriate initial neural network topology. This additional step is of merit as it allows KBANN to generalize better than systems that train from examples alone. In experiments involving datasets from the Human Genome Project[1], KBANN outperformed symbolic theory refinement systems (such as EITHER) and other learning algorithms such as backpropagation and ID3 [TS94]. KBANN is limited by the fact that it does not modify the network's topology and theory refinement is conducted solely by updating the connection weights. This prevents the incorporation of new rules and also restricts the algorithm's ability to compensate for inaccuracies in the domain theory.

As seen in chapter 8 constructive neural network learning algorithms offer an interesting approach for dynamically constructing near-minimal networks for pattern classification tasks. Further, constructive learning algorithms offer several advantages over the traditional backpropagation style learning algorithms (see chapter 7). Constructive learning algorithms thus lend themselves well to the design of knowledge based neural networks for theory refinement. The domain theory can be translated into an initial network topology as in the case of the KBANN algorithm. New rules can be incorporated and inaccuracies in the existing rules (if any) can be corrected by dynamically adding new neurons to the network. These new neurons can be trained using a sequence of labeled examples.

Against this background we discuss a constructive learning approach for theory refinement in knowledge based neural networks. In section 10.2 we describe some related constructive theory refinement systems and compare them with our proposed approach. In section 10.3 we outline the process of incorporating the domain theory into the initial network topology and the constructive learning algorithm that is used to dynamically grow the knowledge based network. Specifically, we discuss a new hybrid *Tiling-Pyramid* constructive learning algorithm that uses an adaptive vector quantization based on the *MTiling* algorithm in conjunction with

[1]These datasets are available from the University of Wisconsin Madison WWW site at (ftp://ftp.cs.wisc.edu/machine-learning/shavlik-group/datasets/).

the *pyramid* learning algorithm. In section 10.4 we present the results of our experiments with the *Financial Advisor Rule Base* [LS89, FO93] and two datasets from the Human Genome Project (*ribosome binding sites* and *DNA promoter sequences*). We conclude in section 10.5 with a summary and outline some promising directions for future research.

10.2 Related Work

Fletcher and Obradović [FO93] designed a constructive learning method for dynamically adding neurons to the initial knowledge based network. Their approach starts with an initial network representing the domain theory and modifies this theory by training a single hidden layer of TLUs using the labeled training data. The resultant network topology is depicted in Fig. 10.1. Their method uses the *hyperplane detection* from *examples* (HDE) algorithm [BL91] to construct the hidden layer. The HDE algorithm divides the feature space with hyperplanes. Each hyperplane is constructed by randomly selecting two points that belong to different output classes and localizing a suitable split between them. This process is repeated until a fixed number of hyperplanes is constructed. Fletcher and Obradović's algorithm maps these hyperplanes to a set of TLUs and then then trains the final output unit using the *pocket algorithm* with *ratchet modification* algorithm [Gal90].

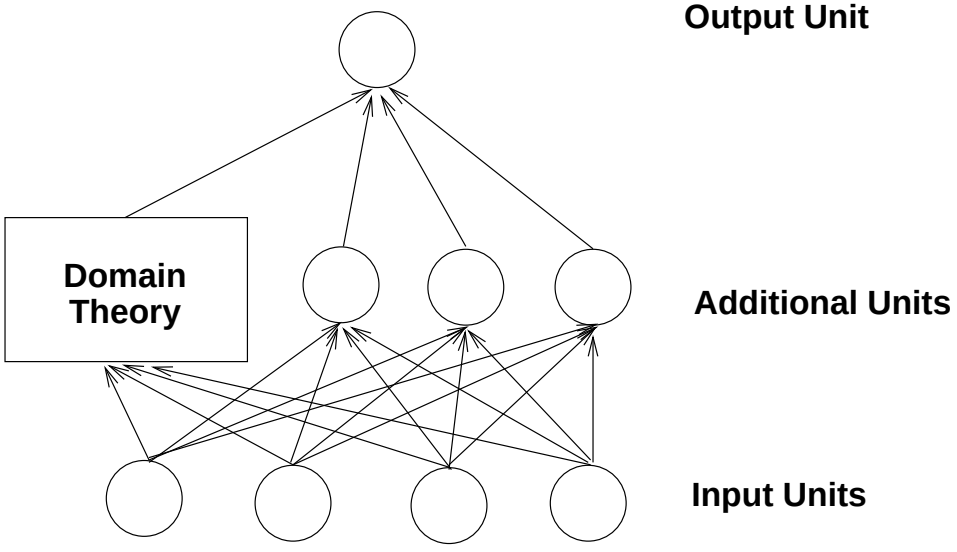

Figure 10.1 Knowledge Based Neural Network.

Our approach is similar to the one taken by Fletcher and Obradović. Instead of constructing a single hidden layer we allow the constructive learning algorithm to build a network of one or more hidden layers (if necessary) above the initial network representing the domain theory (see Fig. 10.2). This provides a more general framework for incorporating domain knowledge into any constructive neural network learning algorithm. Since the performance of various constructive learning algorithms often differs quite significantly for different datasets (see chapter 8) it might be advantageous to have a scheme that allows the construction of an appropriate network topology to augment the initial network instead of limiting the network construction to a single hidden layer.

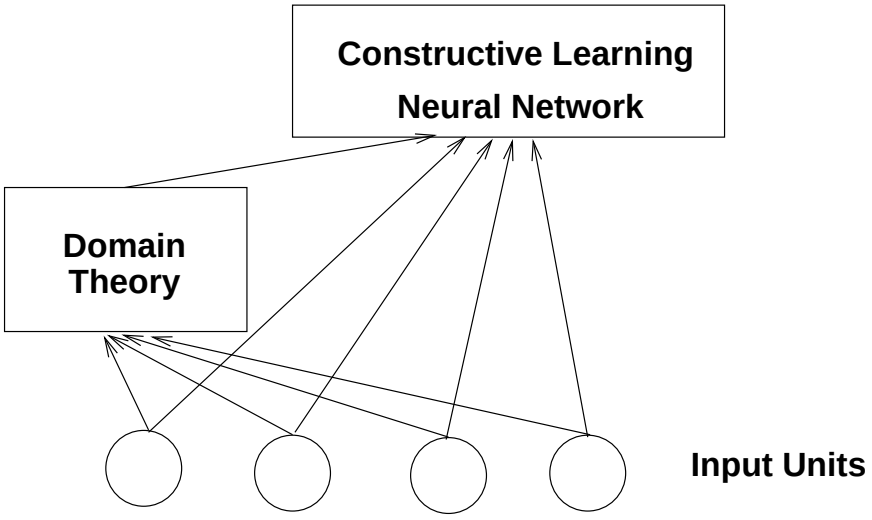

Figure 10.2 Constructive Learning in Knowledge Based Networks.

RAPTURE is a system for refining domain theories that contains probabilistic rules represented in the certainty-factor format [MM94]. It first translates the initial domain theory to an appropriate neural network architecture and then refines the domain theory by using backpropagation training on the network (just as in the case of KBANN). Further, it augments the network topology by adding new neurons using the *upstart* learning algorithm [Fre90b]. Apart from the fact that RAPTURE is designed for probabilistic rule bases, it differs from our approach of using a constructive learning algorithm to augment the initial network topology in the following manner: RAPTURE uses backpropagation based training in addition to

constructive learning. It also explicitly adds links to the existing network based on ID3's information gain heuristic. Our approach is simpler in that it just uses a constructive neural network algorithm that adds TLUs to augment the initial network topology and trains them using a perceptron style learning rule.

Opitz and Shavlik have extensively studied connectionist theory refinement systems that overcome the fixed topology limitation of the KBANN algorithm [OS95, OS97]. They have focussed on the design of systems that use abundant computational resources to yield theory revision systems with improved generalization performance. The TopGen algorithm [OS95] searches through the space of possible expansions of a KBANN network to determine the expansion that has the best generalization accuracy on the cross-validation set. More specifically, the algorithm first translates the domain theory to a KBANN (using the *rules-to-network* algorithm), trains the KBANN using backpropagation, and places the network on a queue of candidate hypotheses. At each step, the algorithm picks the best network (in terms of classification accuracy on the cross-validation set) and explores possible ways of expanding it. New networks are generated by strategically adding nodes at different locations within the best network selected. These networks are trained and placed on the queue. The best network on the queue after a prespecified number of epochs is returned.

The REGENT algorithm broadens the space of networks searched by TopGen by performing a genetic search in the space of all neural network architectures [OS97]. REGENT first creates a diversified population of networks from the initial KBANN. The network's error on a cross-validation set is selected as its measure of fitness. During each generation of the genetic evolution a subset of the population is selected for reproduction. Application of the genetic *mutation* and *crossover* operators results in the production of new candidate networks. The genetic operators are specialized for connectionist theory refinement. Specifically, mutation adds a node to the network using the TopGen algorithm and crossover attempts to maintain the network's rule structure. After each evolution, the network with the best fitness value is reported as the current best hypothesis. Both TopGen and REGENT were evaluated on datasets from the human genome project and found to perform better when compared with

the standard backpropagation or the KBANN algorithms.

Our approach is considerably simpler than both TopGen and REGENT. We construct a single network of TLUs as against a population of networks constructed by TopGen and REGENT. The impact of this is on the training time of knowledge based networks. TopGen and REGENT have reportedly taken several days to search 500 networks and report the best [OS97]. On the other hand our approach requires only a few minutes of CPU time for training. Related to this issue of training time is TopGen and REGENT's use of the expensive backpropagation style training as opposed to the simple perceptron type weight update rule used in our approach. Further, the backpropagation algorithm might not be very effective in networks with a large number layers as the propagated error tends to diffuse considerably from one layer to the next. TopGen and REGENT allow weight changes even to the part of the network that incorporates the original domain theory. There is a possibility that these weight changes would completely alter the original rules embedded in the neural network. Our approach leaves the initial neural network (representing the domain theory) unchanged. The domain theory revision is performed by constructively adding new neurons to the network and training them. Leaving the original domain theory intact might simplify the task of extracting refined knowledge from the trained neural network as in this case the knowledge extraction routine will only be required to focus on the newly added neurons. Additionally, the task of identifying which knowledge rules were newly added and which ones were constructed to offset inaccuracies in the original domain theory is simplified when the original theory is left intact.

10.3 Constructive Knowledge Based Neural Network Learning Algorithms

10.3.1 Embedding the Domain Theory in a Neural Network

We use a symbolic knowledge encoding procedure to translate the initial domain theory into a network of TLUs. This procedure is based on the *rules-to-networks* algorithm of Towell and Shavlik [TSN90, FO93]. It involves rewriting the knowledge rules into a format that highlights the hierarchical structure of the domain theory. In particular, the disjuncts are expressed as a set of rules that each have only one antecedent. This modified set of rules can be mapped to

an *AND-OR* graph which in turn can be directly translated into a network of TLUs.

For example, consider the following propositional rules of a domain theory:

$$A \ :- \ B, C, D$$

$$A \ :- \ D, \neg E$$

The rules are rewritten in the following format:

$$A \ :- \ A'$$

$$A' \ :- \ B, C, D$$

$$A \ :- \ A''$$

$$A'' \ :- \ D, \neg E$$

The *AND-OR* graph corresponding to the modified set of rules is shown in Fig. 10.3. The equivalent network of bipolar TLUs (with outputs 1 and -1) is shown in Fig. 10.4. Note that the TLUs A' and A'' implement the logical *AND* function and the TLU A implements the logical *OR* function.

Further, the magnitudes of the connection weights can be set appropriately to satisfy different rules as shown in Fig. 10.5. Note that this TLU implements the rule "*if* $a - 4b > 6$ *then* c".

Using the approach outlined above, the initial neural network topology corresponding to the simple financial advisor rule base (due to [LS89]) of Fig. 10.6 is shown in Fig. 10.7. Each TLU in the network computes a bipolar hardlimiting function (i.e., the TLU's outputs are 1 and -1) of the weighted sum of its inputs. The neurons in the first hidden layer encode the rules 6–9 of the rule base. The only TLU in the second hidden layer computes the logical *and* function and encodes rule 4 and so on.

10.3.2 Refining the Knowledge Rules

A constructive neural network learning algorithm is used to augment the initial network topology. For the purpose of our experiments with constructive theory refinement we have

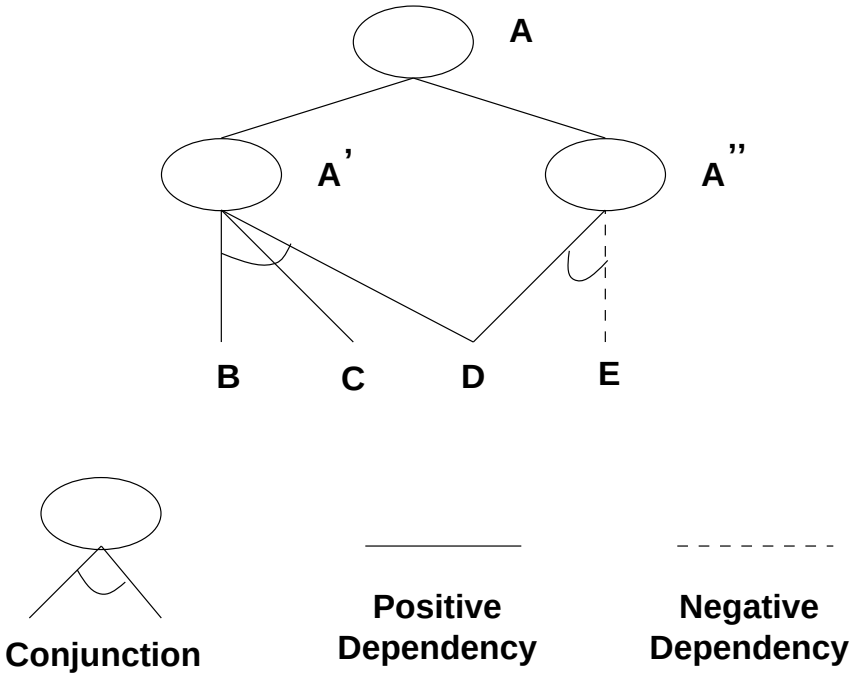

Figure 10.3 *AND-OR* Graph Representation of Knowledge Rules.

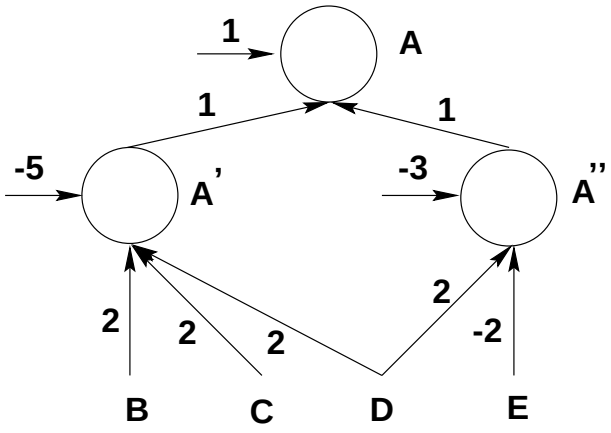

Figure 10.4 Neural Network Implementation of Knowledge Rules.

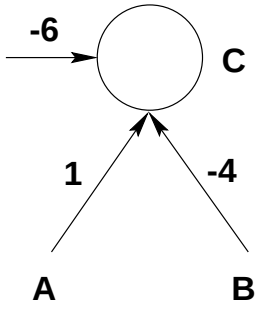

Figure 10.5 TLU Implementing an *If-Then* Propositional Rule.

1	if (sav_adeq and inc_adeq)	then	invest_stocks
2	if dep_sav_adeq	then	sav_adeq
3	if assets_hi	then	sav_adeq
4	if (dep_inc_adeq and earn_steady)	then	inc_adeq
5	if debt_lo	then	inc_adeq
6	if (sav $\geq$ dep * 5000)	then	dep_sav_adeq
7	if (assets $\geq$ income * 10)	then	assets_hi
8	if (income $\geq$ 25000 + dep * 4000)	then	dep_inc_adeq
9	if (debt_pmt $<$ income * 0.3)	then	debt_lo

Figure 10.6 Financial Advisor Rule Base.

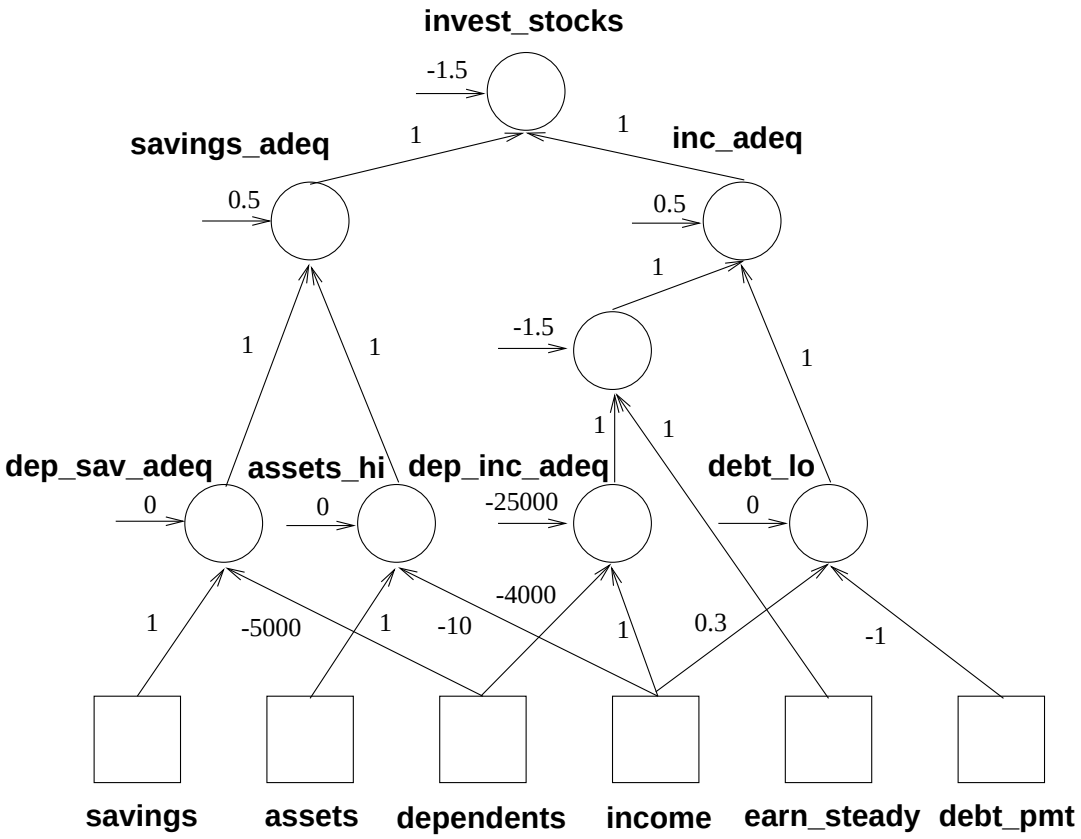

Figure 10.7 Embedding the Financial Advisor Domain Theory in a Neural Network.

used a novel hybrid algorithm that combines the features of the *tiling* [MN89] and the *pyramid* [Gal90] learning algorithms.

In chapter 8 we described provably correct extensions of several constructive learning algorithms to handle multiple output classes and patterns with real-valued attributes. In order to correctly process patterns with real-valued attributes the algorithms such as *tower*, *pyramid*, *upstart*, and *perceptron cascade* require a preprocessing of the dataset[2]. We explained the use of *projection* and *normalization* as two preprocessing techniques. It was observed that despite the data preprocessing the above algorithms were not converging to zero classification errors on certain datasets. Upon further analysis we determined the practical limitations of both the preprocessing techniques. Projection is achieved by appending an extra attribute whose value is equal to the sum of the squares of the values of the attributes in the pattern. In practice, this attribute tends to be very large and consequently hampers the progress of the TLU weight training algorithm (such as the *thermal perceptron algorithm*) which is only allowed to train the TLU for only a limited number of epochs. Normalization on the other hand tends to create the patterns having very low magnitude attributes. This similarly affects the operation of the TLU weight training algorithm. One alternative is to scale the high (or low) magnitude attributes to an acceptable range (say between 0 and 1) and then perform the projection of the modified dataset. Although scaling was observed to perform better we observed that simulation runs with certain datasets still did not converge to zero classification errors (see appendix B).

10.3.2.1 Discretization

A third alternative for handling patterns with real-valued attributes is to use a *discretization* (or *quantization*) algorithm to convert these patterns into equivalent bipolar or binary valued representations. Discretization methods have been extensively studied in conjunction with many different machine learning algorithms. A detailed survey of discretization algorithms appears in [DKS95]. Yang and Honavar's experiments with a simple randomized quantization algorithm and an entropy based quantization algorithm (see [YH96]) demonstrated the effec-

[2]Note that the *tiling* and *sequential* learning algorithms do not require any such preprocessing.

tiveness of quantization in cutting down on the training time and improving the generalization performance of single layer networks of TLUs. A majority of the discretization methods studied in literature are *feature* based schemes in that they independently discretize each real-valued feature of the input patterns. *Vector Quantization* on the other hand is a method of *instance* based discretization. It partitions the N-dimensional instance space into connected regions called *Voronoi regions* [OBS92] and represents each region using a discrete valued *code book vector*. The *learning vector quantizer* (LVQ) algorithm is one method for performing vector quantization [Koh89]. LVQ trains a single layer of k neurons each of which is assigned an arbitrarily chosen class label. The parameter k is chosen heuristically. The weights of these neurons are iteratively updated so that for each training pattern a single neuron whose assigned class label is the same as the training pattern's class label is turned on while all the other neurons remain off. The k-bit output vector produced in response to a pattern is treated as the code book vector representation for that pattern.

We now describe an adaptive quantization algorithm that dynamically constructs an appropriate sized layer of TLUs to perform vector quantization. Recall the operation of the *MTiling* algorithm described in chapter 8. It trains a group of M master neurons (where M is the number of output classes) using a perceptron style learning algorithm. Next it repeatedly adds and trains ancillary neurons to the layer until a faithful representation of the training patterns is obtained. A careful observation of this process shows that the *MTiling* algorithm is actually performing vector quantization. The output of each TLU is either 1 or -1 and the output vector (combined outputs of the M master neurons and the K ancillary neurons) can be treated as a code book vector representation of the input pattern. Note that the quantized outputs are a faithful representation of the training patterns which is an important feature of any good quantization algorithm [YH96]. Thus, the *MTiling* algorithm can be used as an efficient adaptive vector quantization algorithm. In particular, a single *MTiling* layer can be constructed using the set of training patterns and the output of this layer in response to each pattern can be treated as the quantized representation of that pattern. The quantized dataset can then be used to train a *tower*, a *pyramid*, or any other constructive network.

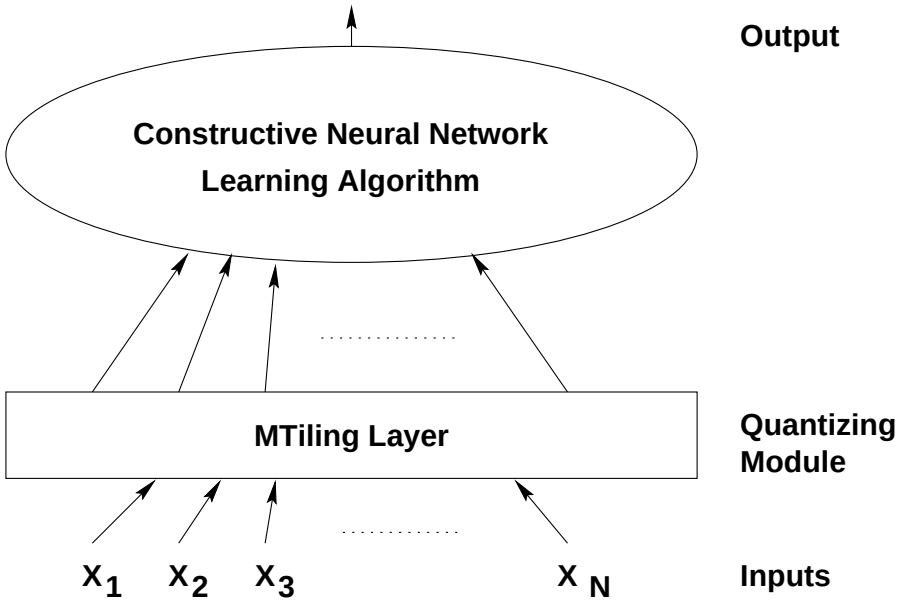

Figure 10.8 Block Diagram of a Hybrid Constructive Network.

10.3.2.2 Hybrid Constructive Learning Algorithms

The *MTiling* based adaptive vector quantization method provides an interesting approach for the design of hybrid constructive learning algorithms. Instead of using a separate *MTiling* based quantization routine first and then using one of the constructive learning algorithms on the quantized representation of the data we could combine these two steps into a single hybrid constructive learning algorithm. The hybrid algorithm reads in a real-valued training set and uses the *MTiling* algorithm to construct a single layer of TLUs that provide a faithful representation of the training patterns. The constructive network is then built on top of the *MTiling* layer. The block diagram of this hybrid constructive network is shown in Fig. 10.8. For the purpose of our experiments with constructive theory refinement we used the hybrid *Tiling-Pyramid* learning algorithm for connectionist theory refinement where a *MPyramid* network is built on top of the *MTiling* layer. Similarly, the hybrid *Tiling-Cascade* learning algorithm can be designed to construct a *MCascade* network on top of the *MTiling* layer.

We have conducted several experiments using the hybrid *Tiling-Pyramid* and the hybrid *Tiling-Cascade* learning algorithms and have compared the performance of these with the

MPyramid, the *MCascade*, and *MTiling* algorithms described in chapter 8. We observed that the two hybrid constructive learning algorithms were able to converge on several real world datasets where the *MPyramid* and *MCascade* algorithms failed to converge. Further, the average size of the networks created by the hybrid algorithms was often significantly smaller than the average size of the *MTiling* networks. A few representative results of these experiments are summarized in appendix B.

10.4 Experimental Results

We report the results of experiments on the *ribosome binding sites* and the *E. coli promoter sequence* datasets from the Human Genome Project and the *financial advisor* rule base. The former two datasets comprise of an imperfect domain theory and a set of labeled examples. In these domains the input is a short segment of DNA nucleotides and the goal is to learn to predict whether these DNA segments contain an important site (such as a ribosome binding site or a promoter) or not. The ribosome binding site dataset's domain theory contains 17 rules. Additionally, there are 1877 labeled training examples. The promoter dataset contains a set of 31 knowledge rules and 936 labeled training examples. The financial advisor rule base is shown in Table 10.6. A set of 5500 labeled examples (500 for training and 5000 for testing) were randomly generated as is the case for the experiments performed by Fletcher and Obradović [FO93]. Each of the 5500 examples are correctly classified by the rules of the financial advisor rule base.

We used the hybrid *Tiling-Pyramid* constructive learning algorithm (described in section 10.3.2.2) to augment the initial domain knowledge. The hybrid network was trained using the *thermal perceptron algorithm* [Fre92]. Each TLU was trained for 500 epochs with the initial weights chosen randomly between in the range $[-1..1]$, the learning rate η held constant at 1 and the initial temperature T_0 set to 1.0. The network was allowed to train to zero classification errors and the network size and generalization accuracy on the test set were recorded.

10.4.1 Human Genome Project Datasets

We performed ten-fold cross validation based training on the ribosome binding set and the promoter datasets using exactly the same folds that were used in the experiments performed by Opitz and Shavlik[3]. On each of the ten runs we first recorded the training and test accuracy of the original network representing the domain theory. The domain theory was refined using the hybrid *Tiling-Pyramid* algorithm to dynamically add new TLUs to the network. Training was continued until the network attained 100% accuracy on the training set. The network size and generalization accuracy were then measured. We report the average generalization accuracy and the average network size (along with the standard deviations where available) over the ten runs for the ribosome dataset in Table 10.1 and for the promoter dataset in Table 10.2.

Table 10.1 Experiments with the Ribosome Dataset.

Rules alone	**Tiling-Pyramid**		**TopGen**		**REGENT**	
Test %	*Test %*	*Size*	*Test %*	*Size*	*Test %*	*Size*
87.3 ± 2.0	90.3 ± 1.8	23 ± 0.0	90.9	42.1 ± 9.3	91.8	70.1 ± 25.1

Table 10.2 Experiments with the Promoters Dataset.

Rules alone	**Tiling-Pyramid**		**TopGen**		**REGENT**	
Test %	*Test %*	*Size*	*Test %*	*Size*	*Test %*	*Size*
77.5 ± 5.0	96.3 ± 1.8	34 ± 0.0	94.8	40.2 ± 3.3	95.8	74.9 ± 38.9

We see from the results described in the Tables 10.1 and 10.2 that the hybrid *Tiling-Pyramid* based constructive theory refinement method generalizes well from the labeled examples. The generalization performance of the refined domain theory (represented by the trained neural network) is significantly better than that of the original set of rules. Further, our approach compares favorably with TopGen and REGENT on both the datasets. In terms of generalization accuracy *Tiling-Pyramid* performs slightly worse than TopGen and REGENT on the ribosome dataset and slightly better on the promoter dataset. Our approach trains a single network as against a pool networks evaluated by TopGen and REGENT. As a result, our ex-

[3]These folds are available along with the dataset.

periments take only about 10 minutes of CPU time on the ribosome binding site dataset and less than 2 minutes of CPU time on the promoter dataset. When compared with the training times for TopGen and REGENT (which are reported to be several days of CPU time [OS97]) we see that our method offers a significant advantage over TopGen and REGENT. It must be kept in mind of course that TopGen and REGENT were designed to take advantage of the available computing resources and come up with hypotheses that have good generalization performance. Further, a comparison of the number of hidden nodes in the resulting networks constructed by the *Tiling-Pyramid*, TopGen, and REGENT shows that *Tiling-Pyramid* is able to come up with considerably smaller networks as compared to TopGen and REGENT. The advantages of smaller (more compact) networks have already been discussed in chapter 9 (see section 9.1).

10.4.2 Financial Advisor Dataset

As described earlier the financial advisor dataset comprises of 5500 patterns that are generated at random to satisfy the rules described in Fig. 10.6. Incomplete domain knowledge was modeled by pruning certain rules and their antecedents from the original rule base (as described in [FO93]). For example, if *sav_adeq* was selected as the pruning point, then the rules for *sav_adeq*, *dep_sav_adeq*, and *assets_hi* are eliminated from the rule base. In other words rules 2, 3, 6, and 7 are pruned. Further, rule 1 is modified to read "*if (inc_adeq) then invest_stocks*". The initial network is constructed from this modified rule base and is then augmented using constructive learning. Our experiments follow those performed by Fletcher and Obradović [FO93]. In Table 10.3 we summarize the average generalization (on the 5000 test patterns) and the average network size over 25 runs for 6 different pruning points. The generalization accuracy of the corresponding network prior to theory refinement (i.e., based on rules alone) is also reported. In all except for the *assets_hi* rule we see a substantial increase in generalization accuracy after theory refinement. Since the generalization accuracy of the network without the *assets_hi* rule is already significantly high, constructive theory refinement understandably does not improve the generalization any further. In Table 10.4 we present the

results for the experiments with HDE that were reported by Fletcher and Obradović[4] [FO93]. These results demonstrate that the performance of the hybrid *Tiling-Pyramid* algorithm compares favorably with that of the HDE algorithm on the financial advisor rule base in terms of both the generalization accuracy and the final size of the trained network.

Table 10.3 Financial Advisor Rule Base (*Tiling-Pyramid*).

Pruning point	**Tiling-Pyramid**		**Rules alone**
	Test %	Size	Test %
dep_sav_adeq	91.2 ± 1.7	28.2 ± 3.6	52.4
assets_hi	99.4 ± 0.2	10 ± 0.0	99.5
dep_inc_adeq	94.3 ± 1.5	21.0 ± 3.1	90.4
debt_lo	94.1 ± 2.0	22.1 ± 4.0	81.2
sav_adeq	90.8 ± 1.5	26.4 ± 3.3	87.6
inc_adeq	83.8 ± 2.2	32.7 ± 2.9	69.4

Table 10.4 Financial Advisor Rule Base (HDE).

Pruning point	**HDE**		**Rules alone**
	Test %	Hidden Units Constructed	Test %
dep_sav_adeq	92.7	31	75.1
assets_hi	92.4	23	93.4
dep_inc_adeq	85.8	25	84.5
debt_lo	84.7	30	61.7
sav_adeq	92.2	19	90.9
inc_adeq	81.2	32	64.6

10.5 Discussion

Connectionist theory refinement systems have been extensively studied in the literature and are found to perform well in several application domains. Most connectionist approaches to theory refinement translate the initial domain theory into an appropriate neural network architecture and then refine this theory by training the network. The KBANN learning algorithm is demonstrated to perform better than several other machine learning algorithms on domains

[4]Note that the standard deviations for the results with these experiments were not available.

such as the promoter and the splice-junction datasets [TSN90, TS94]. However, KBANN is limited by the fact that it does not modify the network topology. The TopGen and REGENT learning algorithms were designed to add new neurons to the KBANN network thereby extending the realm of network topologies considered by KBANN. TopGen heuristically determines effective places in the network where new nodes might be added and REGENT uses a genetic algorithm equipped with crossover and mutation operators (designed specifically for theory refinement) to search the space of neural network architectures. These algorithms use the available computing resources to search the space of network topologies in a bid to identify an expansion of the KBANN that generalizes well on test data. Results of experiments with TopGen and REGENT demonstrated a significant performance improvement over the standard KBANN algorithm [OS95, OS97].

We have presented an approach for constructive learning in knowledge based neural networks. Our method embeds the original domain theory into an initial neural network and then refines the theory by dynamically adding new TLUs to the network. This approach is similar to the one taken by Fletcher and Obradović in their algorithm for connectionist theory refinement [FO93]. The main difference being that our approach allows potentially any constructive neural network learning algorithm to be used for theory refinement whereas Fletcher and Obradović's approach is based on the specific HDE learning algorithm for training a single hidden layer of TLUs. In particular, the hybrid *Tiling-Pyramid* learning algorithm used in our experiments gives satisfactory results.

Experimental results demonstrate that the hybrid *Tiling-Pyramid* algorithm compares favorably with the HDE algorithm on the financial advisor rule base in terms of both the generalization accuracy and the network size. Further, the generalization accuracies after theory refinement exhibit a significant increase in performance when compared to the generalization accuracy based on the imperfect domain theory alone. The algorithm's generalization performance on the ribosome binding site dataset is slightly worse and on the promoter dataset it is slightly better when compared with the performance achieved by TopGen and REGENT. However, the hybrid *Tiling-Pyramid* algorithm generates significantly smaller networks as com-

pared to both TopGen and REGENT. Finally, since the algorithm trains a single network as opposed to a population of networks and since it uses a simple perceptron-style learning rule instead of the more expensive backpropagation learning rule, the training time is significantly lesser than the training times for TopGen and REGENT.

We now analyze some shortcomings of our approach and identify some interesting directions for future research:

- Constructive neural network learning algorithms typically train the network until the number of errors on the training set is reduced to zero. As was observed in the experiments described in chapter 8, training the network until zero classification errors is likely to result in over-fitting of the training data which in turn might hurt the generalization performance of the network. It is of interest to study the performance of the hybrid *Tiling-Pyramid* algorithm (and other constructive learning algorithms) when the network's generalization performance on a *hold-out* set of examples is used to determine when to stop training.

- Our current framework does not allow any explicit changes to the original rules of the domain theory. Instead theory refinement is performed indirectly by adding new rules to over-ride the effects of the existing ones. In some scenarios it might be beneficial to allow the theory refinement system to explicitly modify the original rules. TopGen uses heuristics to determine effective places within the network where new neurons could be added. RAPTURE uses the *upstart* algorithm to grow the existing network. The *upstart* algorithm (see section 8.4) specifically trains daughter neurons to correct some of the errors made by the existing neurons in the network. Design of a constructive learning scheme that allows for direct modification of the existing domain theory and also adds new knowledge rules is worth exploring in depth.

- Extraction of rules from the trained neural networks is an actively pursued area of research and finds direct applicability in datamining. We have not yet explored approaches for extracting the refined knowledge rules from the trained *Tiling-Pyramid* network. We

conjecture that our method for training constructive learning algorithms would make the knowledge extraction task simpler. Our approach uses elementary TLUs whose operation can more easily be translated into rules than the *sigmoid* neurons typically used in backpropagation type algorithms. Further, since the original rules are left uncorrupted in our approach, the comprehensibility of rules extracted from the trained network is likely to improve significantly. Of late there is significant interest in the study of efficient techniques for knowledge extraction from trained neural networks. The interested reader is referred to [TS93, Fu93, Cra96] for additional details.

- The types of domain theory rules that can be incorporated into the network are limited to propositional rules. Further, there is no mechanism for handling uncertainty in rules. An extension of the knowledge based neural networks to handle rules based on first order logic and to handle uncertainty by adjusting the weights of the individual connections merits further investigation.

11 SUMMARY

In this dissertation we have addressed two important machine learning research problems:

1. Efficient methods for learning deterministic finite state automata from labeled examples.
2. Design and analysis of constructive neural network learning algorithms that dynamically construct near-minimal networks of threshold logic units for applications such as pattern classification and inductive knowledge acquisition.

DFA are recognizing devices for regular grammars which form the simplest class of grammars in the Chomsky formal language hierarchy. The problem of learning DFA poses several intriguing challenges and has been actively pursued for over two decades. It is well known that DFA cannot be efficiently learned from arbitrary sets of labeled examples. Despite this, a significant amount of research effort is dedicated to designing efficient heuristic algorithms, establishing simpler (more helpful) models, and identifying promising new application areas for DFA learning. Another important reason for extensive interest in this area is that the experience gained from the various attempts to design efficient methods for learning DFA is likely to yield useful insights about the learnability of other more expressive classes of grammars (such as natural language grammars). The primary focus of part 1 of this dissertation was on the design of efficient algorithms for learning DFA where the learner has access to some representative set of labeled examples. Additionally, the learner might also avail of the facility of a knowledgeable teacher who guides the learning task by answering queries.

Constructive neural network learning algorithms offer an interesting approach for the incrementally generating near-minimal neural network architectures for a given task. They have several advantages when compared with the more traditional approaches such as the *backpropagation* learning algorithm which searches for a suitable weight setting in an otherwise *a-prior*

fixed network topology. Specifically, constructive algorithms have potential for guaranteed convergence to zero classification errors on any finite non-contradictory dataset, adaptively determine both the network topology and the weight settings of the individual neurons of the network, typically use only the elementary perceptron style learning rule, and provide a natural framework for incorporation of domain specific prior knowledge. Thus, constructive neural network learning algorithms are important tools in the design of automatic pattern classification and inductive knowledge acquisition systems.

We summarize the contributions of this dissertation research in the areas of regular grammar inference and constructive neural network algorithms and outline some promising directions for future research.

11.1 Contributions

11.1.1 Version Space Approach to Learning DFA

We have presented an approach for compactly representing the hypothesis space of candidate finite state automata using a version space. This representation overcomes the need for explicitly enumerating the entire hypothesis space. A bidirectional version space candidate elimination algorithm can search the hypothesis space using membership queries to identify the target DFA.

11.1.2 Incremental Interactive Algorithm for Learning DFA

The *IID* is an incremental algorithm for learning the target DFA from a set of labeled examples and membership queries. Salient features of the *IID* algorithm include guaranteed convergence to the target DFA in the limit, polynomial worst case time and space complexities, no requirement of storing all the examples encountered during learning, and no restriction of any specific order of presentation of the training examples.

11.1.3 Learning DFA from Simple Examples

Efficient learning of DFA is one of the most challenging research problems in the field of *grammatical inference*. It is known that both exact and approximate (in the PAC sense) identifiability of DFA is a hard problem. Pitt posed an open research problem that asked whether DFA are approximately learnable under some specific distribution or groups of distributions. We answered this problem in the affirmative by showing the learnability of DFA from simple examples. Specifically, we have demonstrated that the class of simple DFA is efficiently PAC learnable under the Solomonoff-Levin universal distribution. Additionally, if the labeled examples are drawn at random from the universal distribution where a knowledgeable teacher might pick representative examples of the target concept then it is proved that the entire class of DFA is efficiently PAC learnable. We have argued for the generality of the framework for learning from simple examples (called the PACS model) by proving that any concept that is learnable under Gold's model for learning from characteristic samples or equivalently under Goldman and Mathias' polynomial teachability model is also learnable under the PACS model.

11.1.4 Provably Correct Constructive Neural Network Learning Algorithms

We have proposed a general framework for extending constructive neural network learning algorithms to domains that involve multi-category pattern classification and real-valued pattern attributes. In particular, we have designed provably correct extensions of the *tower*, *pyramid*, *upstart*, *perceptron cascade*, *tiling*, and *sequential* learning algorithms for handling multiple output classes and real-valued pattern attributes. An experimental evaluation of the performance of these algorithms on several artificial and real-world datasets demonstrated the practical applicability of constructive learning algorithms as viable alternatives to the traditional methods such as backpropagation.

11.1.5 Pruning Strategies in MTiling Constructive Neural Networks

We have designed three efficient neuron pruning strategies to eliminate the redundancy in *MTiling* networks or in any hybrid networks that use the *MTiling* algorithm. These pruning

strategies were able to achieve moderate to significant reduction in the network size without in any way sacrificing the generalization performance of the network. An effective combination of network growing strategies with pruning techniques for eliminating redundancies will move constructive learning algorithms one step closer to the goal of designing the minimal network topology for a given task.

11.1.6 Constructive Theory Refinement in Knowledge Based Neural Networks

We have developed a framework for incorporating domain specific prior knowledge in a neural network where the original domain theory is progressively refined by dynamically adding TLUs to the network and training them. We have also designed a novel hybrid learning algorithm that combines the features of the *tiling* and the *pyramid* constructive learning algorithms. The hybrid *Tiling-Pyramid* learning algorithm compares favorably with other algorithms for connectionist theory refinement on several benchmark datasets.

11.2 Future Work

During the course of this dissertation research we have identified several interesting problems and avenues that merit further investigation. At the end of each chapter of this dissertation we have mentioned the relevant open research areas. In this section we seek to highlight some key future research directions.

11.2.1 Implications of Learning from Simple Examples

It is of interest to explore the applicability of the framework for learning from simple examples for learning higher classes of formal language grammars such as context free grammars. Very strong negative results in grammar inference were proved recently when it was demonstrated that the classes of context free grammars, linear grammars, simple deterministic grammars, and non-deterministic finite state automata are not learnable under Gold's model for polynomial identification from characteristic samples [Hig96]. We proved that any concept learnable under Gold's model is also learnable under the PACS model. The converse of this

theorem remains an open problem. If it turns out that the converse is not true then it would be intersting to prove whether some or all of the above four concept classes that are not learnable under Gold's model can be learned under the PACS framework.

The universal distribution is not computable. The applicability of the PACS model under some efficiently computable approximation of the universal distribution needs to be explored further. A systematic characterization of the framework for learning under helpful distributions (due to [DG97]) might give us a more practical framework for learning from simple examples.

In applications such as natural language learning it is not inconceivable that a teacher might provide simpler examples of the target concept first before providing the more complex ones. It is worth comparing the notion of simplicity that is implicit in these scenarios with Kolmogorov complexity that provides a measure of intrinsic complexity of an object.

11.2.2 Modeling the Behavior of Intelligent Autonomous Agents

Intelligent autonomous agents have been successfully applied in several domains such as personalized e-mail filtering, news weeding, electronic commerce, etc. [Mae95]. It is of interest to design a formal framework to model agent behavior. Regular grammars can be used to capture the behavior of *intelligent agents* like robots navigating in a finite world. Incremental regular grammar inference can provide a framework for these agents to learn from experience in an unfamiliar environment. Algorithms such as *IID* can provide efficient tools for modeling agent behavior in an interactive setting (where an agent is allowed to pose queries). A knowledgeable teacher might not always be available in all practical learning situations. Algorithms that are able to learn efficiently in environments where no teacher is available are of considerable interest.

11.2.3 Knowledge Extraction from Constructive Neural Networks

The field of knowledge discovery and data mining seeks to use machine learning techniques to extract interesting rules from large databases [FPSS96]. In these applications it is vital that the learned model be comprehensible to a human. Neural network models have been shown

to have better generalization capability on several domains than some of the symbolic or rule based machine learning approaches. Despite this, current data mining systems prefer to use the more traditional symbolic or rule based approaches because the models they produce are much more comprehensible. The task of extracting knowledge rules from a trained neural network is thus of significant practical value. We conjecture that our method for training constructive learning algorithms would make the knowledge extraction task simpler. We have already demonstrated the feasibility of using constructive neural network learning algorithms for theory refinement. It is worth exploring different methods for extracting the refined rules from the trained neural networks.

11.2.4 Constructive Neural Networks in a Lifelong Learning Framework

Recent research has focussed on the use of neural networks for *lifelong learning* [Thr95] where networks are trained to learn multiple classification tasks one after the other. It is desirable to allow the network to exploit the knowledge acquired while learning one task to simplify the learning of a related (possibly more complicated) task. Constructive learning algorithms offer an interesting approach to the use of domain knowledge to learn multiple classification tasks. A network that has domain knowledge from the simpler task(s) built into its architecture (either by explicitly setting the values for the connection weights or by training them) can form a building block for a system that constructively learns more difficult tasks. The performance of constructive learning algorithms in this setting of lifelong learning merits further study.

11.2.5 Characterization of the Bias of Constructive Neural Networks

It is well known that a suitably designed bias can greatly simplify the task of the learning system. Each constructive algorithm has its own set of inductive and representational biases implicit in the design choices that determine when and where a new neuron is added and how it is trained. Towards this end, we have analyzed the performance of several different constructive neural network learning algorithms on a variety of artificial and real-world datasets

and have identified some biases exploited by these learning algorithms. A more systematic characterization of the inductive bias of these algorithms would be useful in guiding the design of new or hybrid constructive learning algorithms that build smaller networks with better generalization performance.

APPENDIX A CONVERGENCE OF CONSTRUCTIVE LEARNING ALGORITHMS ON NORMALIZED DATASETS

In chapter 8 we have demonstrated the convergence of different constructive learning algorithms for multi-category pattern classification of datasets that involve real-valued pattern attributes. We assumed that the preprocessing of the dataset where necessary would be performed by projecting each pattern to a parabolic surface. In this chapter we demonstrate the convergence of the *tower* algorithm in the case where preprocessing involves normalizing the patterns of the dataset. This involves a slight modification of the convergence proof discussed in section 8.1. The convergence proofs of the *pyramid*, *upstart*, and the *perceptron cascade* algorithms for the case of normalized input patterns can be worked out similarly.

Convergence Proof for the Tower Algorithm

Theorem A.1 *There exists a weight setting for neurons in the newly added layer L in the multi-category tower network such that the number of patterns misclassified by the tower with L layers is less than the number of patterns misclassified prior to the addition of the L^{th} layer (i.e., $\forall L > 1,\ e_L < e_{L-1}$).*

Proof:

Assume that all the patterns in the dataset are normalized. i.e., $\forall p\ \mathbf{X}^p$ is such that $\sum_{i=1}^{N}(X_i^p)^2 = 1$. Define $\kappa = \max_{p,q}\sum_{i=1}^{N}(X_i^p - X_i^q)^2$. For each pattern $\mathbf{X}^p$, define ϵ_p as $0 < \epsilon_p < \min_{p,q\neq p}\sum_{i=1}^{N}(X_i^p - X_i^q)^2$. It is clear that $0 < \epsilon_p < \kappa$ for all patterns $\mathbf{X}^p$. Assume that a pattern $\mathbf{X}^p$ was not correctly classified at layer $L-1$ of the tower network (i.e., $\mathbf{C}^p \neq \mathbf{O}_{L-1}^p$). Consider the

following weight setting for the neuron L_j:

$$\begin{aligned} W_{L_j,0} &= C_j^p(\kappa + \epsilon_p - \sum_{i=1}^{N}(X_i^p)^2) - C_j^p \\ W_{L_j,I_i} &= 2C_j^p X_i^p \text{ for } i = 1 \ldots N \\ W_{L_j,L-1_j} &= \kappa \\ W_{L_j,L-1_k} &= 0 \text{ for } k = 1 \ldots M, k \neq j \end{aligned} \tag{A.1}$$

For the pattern $\mathbf{X}^p$ the net input of neuron L_j is:

$$\begin{aligned} n_{L_j}^p &= W_{L_j,0} + \sum_{i=1}^{N} W_{L_j,I_i} X_i^p + \sum_{i=1}^{M} W_{L_j,L-1_i} O_{L-1_i}^p \\ &= C_j^p(\kappa + \epsilon_p - \sum_{i=1}^{N}(X_i^p)^2) - C_j^p + 2C_j^p \sum_{i=1}^{N}(X_i^p)^2 + \kappa O_{L-1_j}^p \\ &= C_j^p(\kappa + \epsilon_p) + \kappa O_{L-1_j}^p \text{ since } \sum_{i=1}^{N}(X_i^p)^2 = 1 \end{aligned} \tag{A.2}$$

If $C_j^p = -O_{L-1_j}^p$:

$$\begin{aligned} n_{L_j}^p &= C_j^p \epsilon_p \\ O_{L_j}^p &= sgn(n_{L_j}^p) \\ &= C_j^p \text{ since } \epsilon_p > 0 \end{aligned}$$

If $C_j^p = O_{L-1_j}^p$:

$$\begin{aligned} n_{L_j}^p &= (2\kappa + \epsilon_p) C_j^p \\ O_{L_j}^p &= sgn(n_{L_j}^p) \\ &= C_j^p \text{ since } \kappa, \epsilon_p > 0 \end{aligned}$$

Thus we have shown that the pattern $\mathbf{X}^p$ is corrected at layer L. Now consider a pattern $\mathbf{X}^q \neq \mathbf{X}^p$. Note that for normalized patterns $-C_j^p$ can be re-written as $-C_j^p(\sum_{i=1}^{N}(X_i^q)^2)$.

$$\begin{aligned} n_{L_j}^q &= W_{L_j,0} + \sum_{i=1}^{N} W_{L_j,I_i} X_i^q + \sum_{i=1}^{M} W_{L_j,L-1_i} O_{L-1_i}^q \\ &= C_j^p(\kappa + \epsilon_p - \sum_{i=1}^{N}(X_i^p)^2) - C_j^p + 2C_j^p \sum_{i=1}^{N}(X_i^p)(X_i^q) + \kappa O_{L-1_j}^q \end{aligned}$$

$$
\begin{aligned}
&= C_j^p(\kappa+\epsilon_p) + \kappa O^q_{L-1_j} - C_j^p \sum_{i=1}^{N}[(X_i^p)^2 - 2(X_i^p)(X_i^q) + (X_i^q)^2] \\
&= C_j^p(\kappa+\epsilon_p) + \kappa O^q_{L-1_j} - C_j^p [\sum_{i=1}^{N}(X_i^p - X_i^q)^2] \\
&= C_j^p(\kappa+\epsilon_p-\epsilon^{'}) + \kappa O^q_{L-1_j} \text{ where } \epsilon^{'} = \sum_{i=1}^{N}(X_i^p - X_i^q)^2;\ \text{note } \epsilon^{'} > \epsilon_p \\
&= \kappa^{'} C_j^p + \kappa O^q_{L-1_j} \text{ where } \kappa+\epsilon_p-\epsilon^{'} = \kappa^{'} \qquad \text{(A.3)} \\
O^q_{L_j} &= sgn(n^q_{L_j}) \\
&= O^q_{L-1_j} \text{ since } \kappa^{'} < \kappa
\end{aligned}
$$

Thus, for all patterns $\mathbf{X}^q \neq \mathbf{X}^p$, the outputs produced at layers L and $L-1$ are identical. We have shown the existence of a weight setting that is guaranteed to yield a reduction in the number of misclassified patterns whenever a new layer is added to the *tower* network. We rely on the TLU weight training algorithm $\mathcal{A}$ to find such a weight setting. Since the training set is finite in size, eventual convergence to zero training errors is guaranteed. □

WTA Output Strategy

We now show that even if the output of the *tower* network is computed according to the WTA strategy, the weights for the output neurons in layer L given in equation A.1 will ensure that the number of misclassifications is reduced by at least one. Assume that the output vector $\mathbf{O}^p_{L-1}$ for the misclassified pattern $\mathbf{X}^p$ is such that $O^p_{L-1_\beta} = 1$ and $O^p_{L-1_k} = -1$, $\forall k = 1 \ldots M, k \neq \beta$; whereas the target output $\mathbf{C}^p$ is such that $C^p_\gamma = 1$ and $C^p_l = -1$, $\forall l = 1 \ldots M, l \neq \gamma$, and $\gamma \neq \beta$.

From equation (A.2) the net input for the neuron L_j is:

$$n^p_{L_j} = C_j^p(\kappa+\epsilon_p) + \kappa O^p_{L-1_j}$$

The net inputs for the output neurons L_γ, L_β, and L_j where $j = 1 \ldots M; j \neq \gamma, j \neq \beta$ are given by:

$$n^p_{L_\gamma} = C_\gamma^p(\kappa+\epsilon_p) + \kappa O^p_{L-1_\gamma}$$

$$
\begin{aligned}
&= \epsilon_p \\
n^p_{L_\beta} &= C^p_\beta(\kappa + \epsilon_p) + \kappa O^p_{L-1_\beta} \\
&= -\epsilon_p \\
n^p_{L_j} &= C^p_j(\kappa + \epsilon_p) + \kappa O^p_{L-1_j} \\
&= -2\kappa - \epsilon_p
\end{aligned}
$$

Since the net input of neuron L_γ is higher than that of every other neuron in the output layer, we see that $O^p_{L_\gamma} = 1$ and $O^p_{L_j} = -1$, $\forall j \neq \gamma$. Thus pattern $\mathbf{X}^p$ is correctly classified at layer L. Even if the output in response to pattern $\mathbf{X}^p$ at layer $L-1$ had been $O^p_{L-1_j} = -1$, $\forall j = 1 \ldots M$, it is easy to see that given the weight setting for neurons in layer L, $\mathbf{X}^p$ would be correctly classified at layer L.

Consider the pattern $\mathbf{X}^q \neq \mathbf{X}^p$ that is correctly classified at layer $L-1$ (i.e., $\mathbf{O}^q_{L-1} = \mathbf{C}^q$). From equation (A.3), the net input for neuron L_j is:

$$
n^q_{L_j} = C^p_j(\kappa + \epsilon_p - \epsilon^{'}) + \kappa O^q_{L-1_j}
$$

Since $\kappa + \epsilon_p - \epsilon^{'} < \kappa$, it is easy to see that the neuron L_γ such that $O^q_{L-1_\gamma} = 1$ has the highest net input among all output neurons irrespective of the value assumed by C^p_γ. With this, $\mathbf{O}^q_L = \mathbf{O}^q_{L-1} = \mathbf{C}^q$. Thus, the classification of previously correctly classified patterns remains unchanged.

We have thus proved the convergence of the *tower* algorithm in the case of normalized patterns when the outputs are computed according to the WTA strategy.

APPENDIX B ADDITIONAL EXPERIMENTS WITH CONSTRUCTIVE LEARNING ALGORITHMS

In this appendix we describe the properties of the different datasets used in our experiments and summarize the results of a more systematic experimental evaluation of the performance of the different constructive learning algorithms.

Datasets

We have used an extensive cross section of artificial and real world datasets for our experiments with constructive neural network learning algorithms. These datasets are available either at the UCI Machine Learning Repository [MA94], the ELENA Classification Database [GD+93], the CMU Connectionist Benchmark[1], or are artificial datasets generated by us at Iowa State University. Table B.1 summarizes the characteristics of the datasets. **Train** and **Test** denote the size of the training and test sets respectively. **Inputs** indicates the total number of input attributes, **Outputs** represents the number of output classes, and **Attributes** describes the type of input attributes of the patterns. In the 5 bit random dataset (*r5*), the 32 training patterns are randomly assigned to one of three output classes. 5 such datasets representing 5 different random functions were generated. The concentric circles dataset (*3c*) considers points belonging to three concentric circles centered at the origin and having radii 2, 4, and 6 respectively. Points in the two dimensional Euclidean space are assigned to one of three output classes depending on the distance $d(x, O)$ of the point x from the origin O: $\forall x,\ 0 \leq d(x,O) \leq 2 \Longrightarrow x \in$ class 1 ; $2 < d(x,O) \leq 4 \Longrightarrow x \in$ class 2 ; and $4 < d(x,O) \leq 6 \Longrightarrow x \in$ class 3. 1800 points are uniformly drawn at random from the two dimensional euclidian space and are

[1] [ftp://ftp.cs.cmu.edu/afs/cs.cmu.edu/project/connect/bench/]

Table B.1 Datasets.

Dataset	**Train**	**Test**	**Inputs**	**Outputs**	**Attributes**
7 bit parity (*p7*)	128	–	7	2	bipolar
8 bit parity (*p8*)	256	–	8	2	bipolar
9 bit parity (*p9*)	512	–	9	2	bipolar
5 bit random (*r5*)	32	–	5	3	bipolar
2 spirals (*2sp*)	194	–	2	2	real
3 concentric circles (*3c*)	900	900	2	3	real
balance (*balance*)	416	209	4	3	int
glass identification (*glass*)	142	72	9	6	real
ionosphere structure (*ion*)	234	117	34	2	real, int
image segmentation (*seg*)	210	2100	19	7	real, int
iris plant (*iris*)	100	50	4	3	real
liver (*liver*)	230	115	6	2	real, int
pima indians diabetes (*pima*)	576	192	8	2	real, int
sonar (*sonar*)	104	104	60	2	real
vehicle silhouettes (*vhcl*)	846	–	18	4	int
wine recognition (*wine*)	120	58	13	3	real, int
wisconsin diagnostic breast cancer (*wdbc*)	390	189	30	2	real

assigned an appropriate class label. All the other datasets used are standard machine learning benchmarks.

Experimental Results

We have conducted extensive simulation runs to compare the performance of the constructive neural network learning algorithms on a variety of datasets. During some initial simulation runs we observed that the performance of the *MPyramid* and *MCascade* algorithms was comparable (or superior) to the performance of the *MTower* and *MUpstart* algorithms respectively. Further, the training speed of the *MSequential* algorithm was found to be much slower than that of any of the other algorithms. Based on these preliminary results we decided to restrict our attention to the *MPyramid*, *MCascade*, and *MTiling* algorithms alone. Further, we observed that the *MPyramid* and *MCascade* algorithms did not converge to zero training errors on several real world datasets. Thus, we included the hybrid *Tiling-Pyramid* and *Tiling-Cascade* algorithms (see chapter 10) in our experimental studies. We use the term *MTil-*

ing based algorithms to collectively refer to the *MTiling* algorithm and the *Tiling-Pyramid* and *Tiling-Cascade* algorithms that use the *MTiling* based adaptive vector quantization of real-valued attributes.

Data Preparation

We performed experiments using the datasets described in Table B.1. As noted in chapter 8 the *MPyramid* and *MCascade* algorithms require preprocessing of datasets that have real-valued pattern attributes. We performed preprocessing by projecting each pattern onto a parabolic surface (see section 8.1.2). Note that the *MTiling* based algorithms do not require any such preprocessing. Certain real world datasets (for example *vhcl*) have patterns with large attribute values. The additional attribute computed while projecting these patterns to a parabolic surface is thus extremely large in magnitude. This means that large weight changes would be needed to correct for incorrectly classified patterns. Since each TLU is trained only for a certain fixed number of epochs it is possible that large errors are not compensated for in the limited training time that is allowed. Thus, for such datasets we performed preprocessing as follows:

1. *Scaling*: The attributes having large magnitudes were scaled to values in the interval $[0, 1]$ as follows:

$$x_i^p \longleftarrow \frac{x_i^p - x_{i_{min}}}{x_{i_{max}} - x_{i_{min}}} \text{ where } 1 \leq i \leq N,\ 1 \leq p \leq |S| \tag{B.1}$$

   Note that $x_{i_{max}} - x_{i_{min}}$ represents the range of values of the i^{th} attribute. The projection was then taken on this scaled dataset. The resulting dataset thus had $N + 1$ attributes.

2. *Normalization*: Each pattern vector was normalized to have a magnitude of 1 (see section 8.1.2).

In the results reported below we indicate the scaled dataset by a suffix $-s$ (e.g., *vhcl-s*) and a normalized dataset with a suffix $-n$ (e.g., *vhcl-n*). To keep the comparison with *MTiling* and the two hybrid learning algorithms fair we used the same *scaled* (without the projection)

and *normalized* datasets for the experiments with the *MTiling*, *Tiling-Pyramid*, and *Tiling-Cascade* algorithms as well.

Training Methodology

We used the *thermal perceptron algorithm* for training individual TLUs. Each TLU was trained for 500 epochs. The weights for each TLU were initialized to values in the range $[-1..1]$. The learning rate η was set to 1. The temperature T_0 was initialized to 1 and was dynamically updated at the end of each epoch to match the average net input of the neuron(s) during the entire epoch [Bur94]. The pruning option (see chapter 9) was turned on in experiments involving the *MTiling*, *Tiling-Pyramid*, and *Tiling-Cascade* algorithms. For the experiments with the *MCascade* and *Tiling-Cascade* algorithms the training set computed for each daughter neuron was *balanced* if necessary (see section 8.8.2).

We performed a ten-fold cross validation based training on all datasets except the parity and the concentric circles. The combined dataset (including the training and test patterns if any) was randomly divided into 10 equal parts. Ten simulation runs were conducted for each algorithm. On each run, one of the ten parts of the dataset was held out as the test set and the remaining nine parts were used for training. The network was trained until it converged to zero training errors. The accuracy of the trained network on the test set was then measured. Training was stopped if the network failed to converge to zero classification errors after adding 100 hidden neurons in a given layer or after training a total of 25 layers and that particular run was designated as a failure[2]. The following results report the total number of failed runs together with the average network size and the generalization accuracy over the successful runs. 25 runs were used for each constructive learning algorithm in the experiments with the parity and *3c* datasets. One each run the network was trained until zero classification errors on the training set and in the case of the *3c* dataset the generalization accuracy of the trained network was measured using the set of test patterns. The number of failed runs (if any) are reported along with the average network size and the average generalization accuracy over the

[2]Failed runs were not included in the calculation of the averages.

Table B.2 Experiments with the *MPyramid* Algorithm.

Dataset	**Failed Runs**	**Network Size**	**Test Accuracy**
p7	0	4.1 ± 0.3	–
p8	0	5.2 ± 0.8	–
p9	0	5.0 ± 0.2	–
2sp	6	15.5 ± 4.2	92.11 ± 9.12
3c	0	3.0 ± 0.0	99.9 ± 0.2
balance	10	–	–
glass-s	10	–	–
glass-n	10	–	–
ion	0	5.3 ± 1.6	89.1 ± 2.6
iris	10	–	–
liver-s	10	–	–
liver-n	10	–	–
pima-s	10	–	–
pima-n	10	–	–
sonar	0	5.8 ± 0.9	76.0 ± 14.3
vhcl-s	10	–	–
vhcl-n	10	–	–
wdbc-s	9	18.0 ± 0.0	96.4 ± 0.0
wdbc-n	10	–	–
wine-s	10	–	–
wine-n	10	–	–

successful runs. Tables B.2, B.3, B.4, B.5, and B.6 summarize the results of our experiments with the *MPyramid*, *MCascade*, *MTiling*, *Tiling-Pyramid*, and *Tiling-Cascade* algorithms respectively.

To facilitate a comparison of the generalization performance of the constructive learning algorithms with that of the single layer networks we trained single layer networks (using the *thermal perceptron algorithm*) for each of the above datasets. These experiments used exactly the same parameter settings as the experiments with the constructive learning algorithms. In Table B.7 we report the average training and test accuracies achieved by the single layer networks.

Table B.3 Experiments with the *MCascade* Algorithm.

Dataset	**Failed Runs**	**Network Size**	**Test Accuracy**
p7	0	4.8 ± 0.4	–
p8	0	5.0 ± 0.2	–
p9	0	5.6 ± 0.6	–
2sp	0	12.9 ± 2.6	86.3 ± 7.1
3c	0	3.0 ± 0.0	99.9 ± 0.2
balance	5	25.0 ± 1.2	90.7 ± 1.4
glass-s	10	–	–
glass-n	10	–	–
ion	0	2.7 ± 0.7	90.6 ± 7.5
iris	1	12.9 ± 4.6	94.1 ± 7.8
liver-s	10	–	–
liver-n	10	–	–
pima-s	10	–	–
pima-n	10	–	–
sonar	0	3.8 ± 0.4	76.5 ± 10.6
vhcl-s	10	–	–
vhcl-n	10	–	–
wdbc-s	0	4.4 ± 0.8	96.8 ± 2.2
wdbc-n	0	18.1 ± 1.9	89.1 ± 4.1
wine-s	0	7.7 ± 0.7	95.9 ± 4.9
wine-n	0	11.1 ± 1.7	91.2 ± 5.0

Observations

We make the following key observations from the results summarized in the Tables B.2, B.3, B.4, B.5, and B.6.

Convergence Properties

Constructive learning algorithms performed quite well on several highly non-linear datasets. Non-linearly separable datasets cannot be correctly classified by the *perceptron* algorithm training a single layer of TLUs (see the average training accuracy in Table B.7). Constructive learning algorithms were able to converge to zero classification errors on these non-linear datasets. Further, a comparison of the size of the network constructed by the constructive algorithms with the size of the training set indicates that the constructive algorithms are not simply memorizing the classifications of the training patterns but are in fact attempting to

Table B.4 Experiments with the *MTiling* Algorithm.

Dataset	**Failed Runs**	**Network Size**	**Test Accuracy**
p7	0	8.0 ± 0.0	–
p8	0	9.1 ± 0.4	–
p9	0	11.0 ± 5.2	–
2sp	1	41.1 ± 2.8	49.7 ± 17.5
3c	1	40.3 ± 4.7	98.0 ± 10.0
balance	0	35.3 ± 6.6	91.0 ± 3.6
glass-s	0	48.4 ± 4.3	60.0 ± 10.3
glass-n	2	57.1 ± 6.3	60.7 ± 12.7
ion	0	6.5 ± 2.3	84.6 ± 6.5
iris	0	10.4 ± 3.3	95.3 ± 4.5
liver-s	0	44.2 ± 5.6	66.4 ± 10.4
liver-n	0	47.7 ± 6.2	64.4 ± 6.3
pima-s	1	78.4 ± 10.7	64.0 ± 2.6
pima-n	0	94.6 ± 9.5	61.3 ± 5.8
sonar	0	4.9 ± 1.5	77.0 ± 7.2
vhcl-s	0	91.9 ± 7.1	76.1 ± 4.0
vhcl-n	0	134.4 ± 10.5	66.0 ± 5.6
wdbc-s	0	5.9 ± 1.9	96.3 ± 2.1
wdbc-n	2	33.5 ± 4.9	90.0 ± 5.0
wine-s	0	3.0 ± 0.0	95.9 ± 4.0
wine-n	6	21.3 ± 2.2	86.8 ± 7.4

learn a suitable input-output mapping.

The *MPyramid* and *MCascade* algorithms do not converge to zero training errors on several real world datasets. This is the case despite the fact that datasets with large magnitude attributes were either scaled or normalized. On the other hand the hybrid *Tiling-Pyramid* and *Tiling-Cascade* algorithms that use the *MTiling* based adaptive vector quantization were able to converge to zero classification errors on all the datasets. The reason for this behavior is most likely due to the fact that the *MTiling* algorithm trains the ancillary neurons on progressively smaller subsets of the entire training set. In the *MPyramid* and *MCascade* algorithms the entire training set is input to each neuron trained. Since the datasets are inherently difficult to classify presenting the entire pattern set to each neuron results in the addition of new layers (in the case of *MPyramid*) or the addition of new daughter neurons (in the case of *MCascade*)

Table B.5 Experiments with the *Tiling-Pyramid* Algorithm.

Dataset	**Failed Runs**	**Network Size**	**Test Accuracy**
p7	0	8.0 ± 0.0	–
p8	0	9.0 ± 0.0	–
p9	0	10.0 ± 0.0	–
2sp	0	36.9 ± 2.2	59.0 ± 8.5
3c	1	38.7 ± 3.9	95.5 ± 0.7
balance	0	28.2 ± 2.9	91.9 ± 3.7
glass-s	0	48.4 ± 4.3	60.0 ± 10.3
glass-n	4	51.5 ± 3.0	54.8 ± 5.8
ion	0	5.1 ± 1.4	85.4 ± 3.3
iris	0	9.9 ± 2.1	96.0 ± 4.7
liver-s	0	29.6 ± 1.6	66.8 ± 11.4
liver-n	0	35.1 ± 2.1	58.8 ± 6.4
pima-s	1	41.2 ± 1.5	64.9 ± 6.6
pima-n	0	55.9 ± 2.8	61.5 ± 4.2
sonar	0	4.3 ± 0.7	72.0 ± 8.6
vhcl-s	0	76.0 ± 6.0	74.3 ± 5.5
vhcl-n	0	84.5 ± 4.9	64.0 ± 4.5
wdbc-s	0	5.6 ± 2.1	95.7 ± 2.6
wdbc-n	0	26.6 ± 2.5	89.8 ± 3.5
wine-s	0	3.0 ± 0.0	95.9 ± 4.8
wine-n	2	20.5 ± 3.7	83.8 ± 6.9

without any significant improvement in the performance. This in turn prevents the algorithms from converging to zero classification errors within the limited amount of training time that is allowed.

Network Size

The networks generated by the hybrid learning algorithms *Tiling-Pyramid* and *Tiling-Cascade* are smaller (in terms of average number of neurons) as compared to those generated by the *MTiling* algorithm. Further, the average network sizes of the networks generated by the *MPyramid* and *MCascade* (in cases where these algorithms actually converged) are smaller as compared to those generated by the *MTiling* based algorithms. This could suggest the fact that the *MPyramid* and *MCascade* algorithms are more strongly biased towards parsimonious

Table B.6 Experiments with the *Tiling-Cascade* Algorithm.

Dataset	**Failed Runs**	**Network Size**	**Test Accuracy**
p7	0	9.6 ± 3.3	–
p8	0	13.0 ± 7.3	–
p9	0	14.7 ± 10.0	–
2sp	0	38.9 ± 1.9	50.0 ± 17.8
3c	1	38.2 ± 3.9	95.5 ± 0.7
balance	0	25.6 ± 1.7	90.8 ± 4.1
glass-s	2	34.6 ± 1.2	59.5 ± 10.2
glass-n	0	36.5 ± 1.4	44.3 ± 10.3
ion	0	5.0 ± 1.1	86.6 ± 4.1
iris	0	9.1 ± 1.1	95.3 ± 6.3
liver-s	0	30.6 ± 1.3	64.7 ± 9.8
liver-n	0	35.7 ± 3.0	60.6 ± 10.0
pima-s	0	42.9 ± 2.0	68.4 ± 4.5
pima-n	0	58.1 ± 3.6	62.6 ± 8.3
sonar	0	5.9 ± 1.3	73.5 ± 6.3
vhcl-s	0	46.7 ± 2.5	76.0 ± 5.5
vhcl-n	0	59.7 ± 3.3	71.6 ± 6.9
wdbc-s	0	5.9 ± 1.3	96.4 ± 3.3
wdbc-n	0	26.5 ± 2.5	87.9 ± 4.2
wine-s	0	3.0 ± 0.0	97.1 ± 4.2
wine-n	1	20.0 ± 3.2	88.9 ± 10.1

network representations than the *MTiling* based algorithms.

Scaling versus Normalization

The performance of the constructive learning algorithms on the scaled version of the dataset was better than the performance on the normalized version of the dataset both in terms of network size and generalization. This is to be expected because as mentioned earlier normalization tends to produce patterns in which some attributes have extremely small magnitudes in comparison with some others. Consequently, the normalized datasets are significantly harder to classify than their scaled counterparts where the relative magnitudes of all attributes are nearly equal.

Table B.7 Experiments with the *perceptron* Algorithm.

Dataset	**Training Accuracy**	**Test Accuracy**
p7	65.2 ± 1.6	–
p8	63.5 ± 1.1	–
p9	63.7 ± 0.0	–
2sp	55.1 ± 2.9	48.4 ± 16.4
3c	46.3 ± 3.9	44.0 ± 3.7
balance	91.1 ± 0.6	87.9 ± 3.4
glass-s	74.6 ± 2.0	60.5 ± 8.4
glass-n	55.2 ± 2.2	46.7 ± 10.2
ion	97.0 ± 0.9	86.9 ± 5.3
iris	98.7 ± 0.7	97.3 ± 3.4
liver-s	74.1 ± 1.5	69.7 ± 10.9
liver-n	73.6 ± 1.0	67.1 ± 9.5
pima-s	79.4 ± 1.0	76.0 ± 5.8
pima-n	70.9 ± 1.2	68.7 ± 5.6
sonar	95.5 ± 1.9	74.0 ± 7.4
vhcl-s	85.0 ± 0.6	79.1 ± 2.1
vhcl-n	76.9 ± 0.9	73.2 ± 4.7
wdbc-s	99.0 ± 0.3	97.3 ± 1.9
wdbc-n	93.0 ± 0.4	92.1 ± 2.7
wine-s	100.0 ± 0.0	90.6 ± 4.6
wine-n	89.7 ± 3.1	80.6 ± 8.0

Generalization

On an average the constructive learning algorithms generalize well from the training data. A comparison of the test accuracy achieved by the constructive learning algorithms on the *2sp*, *3c*, *balance*, *glass*, *ion*, and *sonar* datasets with the test accuracy of the *perceptron* algorithm bears testimony to the superior generalization ability of the constructive learning algorithms.

The generalization performance of the *MPyramid* and *MCascade* algorithms on the *2sp*, *3c*, and *ion* datasets is better than that of the *MTiling* based algorithms. On other datasets (such as *iris*, *sonar*, *wine*, and *wdbc*) where at least one of the *MPyramid* and *MCascade* algorithms did converge their generalization performance was comparable to that of *MTiling* based algorithms. This strengthens the argument for preferring the *MPyramid* and *MCascade* algorithms over *MTiling* based algorithms. The practical difficulty however, is the fact that

MPyramid and *MCascade* do not converge on several datasets. The use of the *MTiling* based adaptive vector quantization does provide one framework that improves the convergence properties of the *MPyramid* and *MCascade* algorithms on real-valued datasets. It is of interest to explore quantization schemes that might also improve the generalization performance of these algorithms.

On the *iris*, *liver*, *pima*, *vhcl*, and *wdbc* datasets we observe that the *perceptron* algorithm generalizes better than all the constructive learning algorithms. This worsening of generalization could be attributed to one of the following factors:

1. *Inherent limitations of the datasets*:

   Some of the datasets we used were created at a time when algorithms for training multilayer networks were just gaining popularity. It is likely that these datasets contain a set of carefully engineered features that were selected by experts to work well with the algorithms existing at that time [MSTG89]. These inherent limitations of the datasets result in the scenario where it is not possible to improve the generalization on a particular dataset beyond what is achieved by a single layer network.

2. *Presence of irrelevant or noisy attributes*:

   It is possible that the presence of irrelevant or noisy attributes for a set of patterns actually complicates the task of the learning algorithm. The effectiveness of using an appropriate feature subset selection mechanism in culling unwanted input attributes and thereby simplifying the task of several inductive learning algorithms is well known [Rip96]. Experimental results have shown that using GA based feature subset selection algorithm significantly boosts the performance of the DistAl constructive learning algorithm [YH97].

3. *Over-fitting of the training set*:

   Constructive learning algorithms allow the network to train until all training patterns are correctly classified. This is likely to result in over-fitting of the training set where the algorithm spends a large fraction of its effort in trying to correctly classify a small fraction of hard to classify training patterns. A common approach for avoiding over-fitting in most

machine learning algorithms is to use separate training and cross-validation sets. The training is stopped when the generalization on the cross-validation set begins to decline.

Over-fitting

We now describe the results of some experiments conducted specifically to address the issue of over-fitting. We implemented the cross-validation based criterion for early termination of the training phase in our constructive learning algorithms. Tables B.8 and B.9 summarize the results of our ten fold cross-validation experiments with early termination on the *sonar* and *pimas* datasets. The size, training accuracy, and generalization accuracy of the network averaged over the ten runs is reported.

Table B.8 Cross-validation Experiments on the *sonar* Dataset.

Algorithm	**Network Size**	**Training Accuracy**	**Test Accuracy**
MPyramid	2.1 ± 1.0	92.6 ± 4.3	82.5 ± 4.9
MCascade	2.2 ± 1.3	94.3 ± 5.8	83.6 ± 6.7
MTiling	2.3 ± 1.7	96.7 ± 3.0	72.5 ± 10.6
Tiling-Pyramid	2.6 ± 1.7	97.3 ± 2.4	76.5 ± 10.8
Tiling-Cascade	2.5 ± 2.6	97.1 ± 2.4	77.5 ± 11.8

Table B.9 Cross-validation Experiments on the *pima-s* Dataset.

Algorithm	**Network Size**	**Training Accuracy**	**Test Accuracy**
MPyramid	8.1 ± 4.8	80.5 ± 1.7	78.3 ± 3.7
MCascade	6.9 ± 4.8	81.2 ± 2.1	79.2 ± 4.9
MTiling	12.8 ± 21.3	82.4 ± 6.0	76.7 ± 5.2
Tiling-Pyramid	4.3 ± 10.4	80.4 ± 3.9	76.7 ± 5.4
Tiling-Cascade	3.8 ± 8.9	79.8 ± 1.6	76.3 ± 4.3

The results in Tables B.8 and B.9 suggest the following.

- Constructive learning algorithms do indeed tend to overfit data. The average test accuracy of the networks that were trained using the cross-validation based stopping criterion is better than that of the networks that were trained until they correctly classied all training patterns on both the datasets. To further demonstrate the over-fitting we plot the

learning curve for a single run of the *MTiling* algorithm on the *pima-s* dataset in Fig. B.1. Over-fitting sets in very early during training with the best generalization being recorded at the first layer itself. We contrast this with the behavior of the *MTiling* algorithm on the *3c* dataset in Fig. B.2. In this case, there is no over-fitting and the train and test accuracies track each other very closely.

- The *MPyramid* and *MCascade* algorithms did not converge to zero classification errors on the *pima-s* dataset (see Tables B.2 and B.3). However, when the early stopping criterion is implemented for training their performance is comparable to that of the *MTiling* based algorithms.

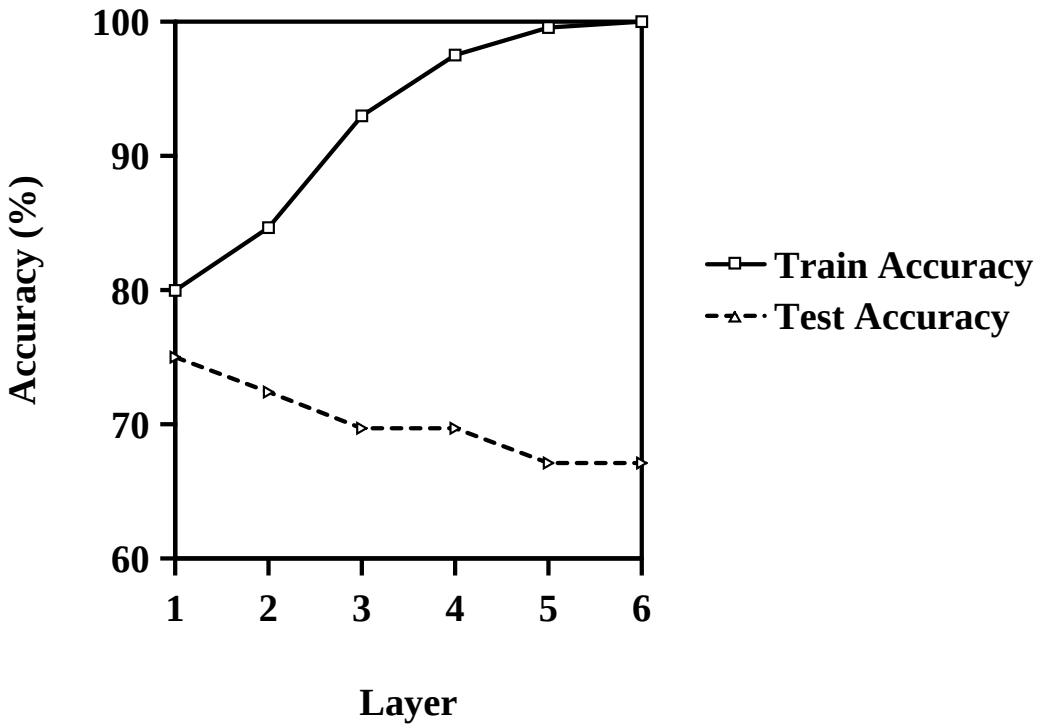

Figure B.1 Learning Curve of the *MTiling* Algorithm (*pima-s* Dataset).

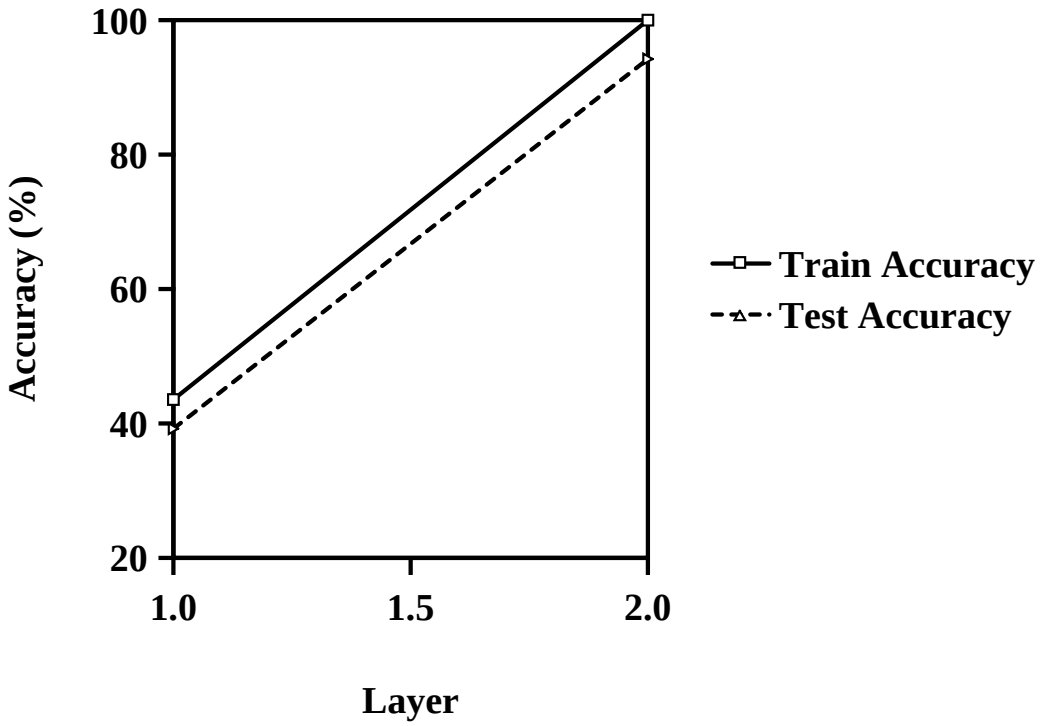

Figure B.2 Learning Curve of the *MTiling* Algorithm (*3c* Dataset).

BIBLIOGRAPHY

[Ang78] D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, 1978.

[Ang81] D. Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51:76–87, 1981.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[BF72] A. Biermann and J. Feldman. A survey of results in grammatical inference. In S. Watanabe, editor, *Frontiers of Pattern Recognition*, pages 31–54. Academic Press, New York, 1972.

[BH69] A. E. Bryson and Y. C. Ho. *Applied Optimal Control.* Blaisdell, New York, 1969.

[BL91] E. Baum and K. Lang. Constructing hidden units using examples and queries. In R. Lippmann, J. Moody, and D. Touretzky, editors, *Advances in Neural Information Processing Systems, vol. 3*, pages 904–910, Morgan Kaufmann, San Mateo, CA, 1991.

[Bur94] N. Burgess. A constructive algorithm that converges for real-valued input patterns. *International Journal of Neural Systems*, 5(1):59–66, 1994.

[CG88] G. Carpenter and S. Grossberg. The art of adaptive pattern recognition by a self-organizing neural network. *Computer*, (March): 77–88, 1988.

[Cho56] N. Chomsky. Three models for the description of language. *PGIT*, 2(3):113–124, 1956.

[CLR91] T. Cormen, C. Leiserson, and Rivest R. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1991.

[CM96] D. Carmel and S. Markovitch. Learning models of intelligent agents. In *Proceedings of the AAAI-96 (vol. 1), Portland, OR*, pages 62–67, AAAI Press/MIT Press, Menlo Park, CA, 1996.

[CPY+95] C-H. Chen, R. Parekh, J. Yang, K. Balakrishnan, and V. Honavar. Analysis of decision boundaries generated by constructive neural network learning algorithms. In *Proceedings of WCNN'95 (vol. 1), Washington D.C.*, pages 628–635, Lawrence Erlbaum Associates/INNS Press, Mahwah, NJ, 1995.

[Cra96] M. Craven. *Extracting Comprehensible Models from Trained Neural Networks.* PhD dissertation, University of Wisconsin, Madison, WI, 1996.

[Day90] J. Dayhoff. *Neural Network Architectures: An Introduction.* Van Nostrand Reinhold, New York, 1990.

[DDG96] F. Denis, C. D'Halluin, and R. Gilleron. PAC learning with simple examples. *STACS'96 - Proceedings of the* 13^{th} *Annual Symposium on the Theoretical Aspects of Computer Science*, pages 231–242, 1996.

[DG97] F. Denis and R. Gilleron. PAC learning under helpful distributions. In *Proceedings of the Eighth International Workshop on Algorithmic Learning Theory (ALT'97), Sendai, Japan, Lecture Notes in Artificial Intelligence 1316*, pages 132–145, 1997.

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

[DKS95] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In *Proceedings of the Twelfth International Conference on Machine Learning, San Francisco, CA*, pages 194–202, 1995.

[DMV94] P. Dupont, L. Miclet, and E. Vidal. What is the search space of the regular inference? In *Proceedings of the Second International Colloquium on Grammatical Inference (ICGI'94), Alicante, Spain*, pages 25–37, 1994.

[DR95] S. Donoho and L. Rendell. Representing and restructuring domain theories: A constructive induction approach. *Journal of Artificial Intelligence Research*, 2:411–446, 1995.

[Dre62] S. Dreyfus. The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45, 1962.

[Dup96a] P. Dupont. Incremental regular inference. In L. Miclet and C. Higuera, editors, *Proceedings of the Third ICGI-96, Montpellier, France, Lecture Notes in Artificial Intelligence 1147*, pages 222–237, 1996.

[Dup96b] P. Dupont. *Utilisation et Apprentissage de Modèles de Language pour la Reconnaissance de la Parole Continue.* PhD dissertation, Ecole Normale Supérieure des Télécommunications, Paris, France, 1996.

[Fah88] S. E. Fahlman. An empirical study of learning speed in backpropagation networks. Technical Report CMU-CS-88-162, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[FB75] K. S. Fu and T. L. Booth. Grammatical inference: Introduction and survey (part 1). *IEEE Transactions on Systems, Man and Cybernetics*, 5:85–111, 1975.

[FL90] S. E. Fahlman and C. Lebiere. The cascade correlation learning algorithm. In D.S. Touretzky, editor, *Neural Information Processing Systems 2*, pages 524–532. Morgan-Kauffman, San Mateo, CA, 1990.

[FLSW90] J. A. Feldman, G. Lakoff, A. Stolcke, and S. H. Weber. Miniature language acquisition: A touchtone for cognitive science. Technical Report TR-90-009, International Computer Science Institute, Berkeley, California, 1990.

[FO93] J. Fletcher and Z. Obradović. Combining prior symbolic knowledge and constructive neural network learning. *Connection Science*, 5(3,4):365–375, 1993.

[FPSS96] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. *Advances in Knowledge Discovery and Data Mining*. MIT Press, Cambridge, MA, 1996.

[Fre90a] M. Frean. *Small Nets and Short Paths: Optimizing Neural Computation*. PhD dissertation, Center for Cognitive Science, Edinburgh University, Edinburgh, Scotland, 1990.

[Fre90b] M. Frean. The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, 2:198–209, 1990.

[Fre92] M. Frean. A thermal perceptron learning rule. *Neural Computation*, 4:946–957, 1992.

[Fu82] K. Fu. *Syntactic Pattern Recognition and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Fu89] L. M. Fu. Integration of neural heuristics into knowledge-based inference. *Connection Science*, 1:325–340, 1989.

[Fu93] L. M. Fu. Knowledge-based connectionism for revising domain theories. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(1):173–182, 1993.

[Gal90] S. Gallant. Perceptron based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191, 1990.

[Gal93] S. Gallant. *Neural Network Learning and Expert Systems*. MIT Press, Cambridge, MA, 1993.

[GD+93] A. Guérin-Dugué et al. Deliverable R1-B1-P - Task B1: Databases. Technical report, Elena-NervesII "Enhanced Learning for Evolutive Neural Architecture", ESPRIT-Basic Research Project Number 6891, 1993.

[Gin90] A. Ginsberg. Theory reduction, theory revision, and retranslation. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 777–782, AAAI/MIT Press, Boston, MA, 1990.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability.* W. H. Freeman, New York, 1979.

[GM93] S. Goldman and H. Mathias. Teaching a smarter learner. In *Proceedings of the Workshop on Computational Learning Theory (COLT'93)*, pages 67–76, ACM Press, New York, 1993.

[Gol78] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[GT78] R.C. Gonzales and M.G. Thomason. *Syntactic Pattern Recognition - An Introduction.* Addison Wesley, Reading, MA, 1978.

[Heb49] D. Hebb. *The Organization of Behavior.* Wiley, New York, 1949.

[Hig96] Colin de la Higuera. Characteristic sets for polynomial grammatical inference. In L. Miclet and C. Higuera, editors, *Proceedings of the Third ICGI-96, Montpellier, France, Lecture Notes in Artificial Intelligence 1147*, pages 59–71, 1996.

[Hir90] H. Hirsh. *Incremental Version Space Merging: A General Framework for Concept Learning.* Kluwer Academic, Boston, MA, 1990.

[Hon90] V. Honavar. *Generative Learning Structures and Processes for Generalized Connectionist Networks.* PhD dissertation, University of Wisconsin, Madison, WI, 1990.

[Hon94] V. Honavar. Toward learning systems that integrate multiple strategies and representations. In V. Honavar and L. Uhr, editors, *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, pages 615–644. Academic Press, New York, 1994.

[Hry92] T. Hrycej. *Modular Learning in Neural Networks.* Wiley, New York, 1992.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, Reading, MA, 1979.

[HU93] V. Honavar and L. Uhr. Generative learning structures and processes for connectionist networks. *Information Sciences*, 70:75–108, 1993.

[Kat89] B. F. Katz. EBL and SBL: A neural network synthesis. In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society*, pages 683–689, 1989.

[KFS94] M. Kopel, R. Feldman, and A. Serge. Bias-driven revision of logical domain theories. *Journal of Artificial Intelligence Research*, 1:159–208, 1994.

[Koh88] T. Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, New York, 1988.

[Koh89] T Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, New York, 1989.

[KV89] M. Kearns and L. G. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. In *Proceedings of the* 21^{st} *Annual ACM Symposium on Theory of Computing*, pages 433–444, 1989.

[KV94] M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, Cambridge, MA, 1994.

[Lan92] K. J. Lang. Random DFAs can be approximately learned from sparse uniform sample. In *Proceedings of the 5th ACM workshop on Computational Learning Theory*, pages 45–52, 1992.

[Lan95] P. Langley. *Elements of Machine Learning.* Morgan Kaufmann, Palo Alto, CA, 1995.

[LP81] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[LS89] G. F. Luger and W. A. Stubblefield. *Artificial Intelligence and the Design of Expert Systems*. Benjamin Cummings, Redwood City, CA, 1989.

[LV91] M. Li and P. Vitányi. Learning simple concepts under simple distributions. *SIAM Journal of Computing*, 20:911–935, 1991.

[LV93] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer-Verlag, New York, 1993.

[LV97] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications, 2^{nd} edition*. Springer-Verlag, New York, 1997.

[MA94] P. Murphy and D. Aha. UCI repository of machine learning databases. Department of Information and Computer Science, University of California, Irvine, CA, 1994.

[Mae95] P. Maes. Agents that reduce work and information overload. *Communications of the ACM*, 38(11):108–114, 1995.

[Mar91] J. C. Martin. *Introduction to Languages and The Theory of Computation*. McGraw-Hill, New York, 1991.

[MGR90] M. Marchand, M. Golea, and P. Rujan. A convergence theorem for sequential learning in two-layer perceptrons. *Europhysics Letters*, 11(6):487–492, 1990.

[Mit80] T. Mitchell. The need for biases in learning generalizations. Technical Report CBM-TR-117, Rutgers University, New Brunswick, NJ, 1980.

[Mit82] T. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[Mit97] T. Mitchell. *Machine Learning*. McGraw Hill, New York, 1997.

[MM94] J. Mahoney and R. Mooney. Comparing methods for refining certainty-factor rule-bases. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 173–180, 1994.

[MMR97] K. Mehrotra, C. Mohan, and S. Ranka. *Elements of Artificial Neural Networks.* MIT Press, Cambridge, MA, 1997.

[MN89] M. Mézard and J. Nadal. Learning feed-forward networks: The tiling algorithm. *J. Phys. A: Math. Gen.*, 22:2191–2203, 1989.

[MP43] W. S. Mcculloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[MP69] M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA, 1969.

[MQ86] L. Miclet and J. Quinqueton. Learning from examples in sequences and grammatical inference. In G. Ferrate *et al*, editor, *Syntactic and Structural Pattern Recognition*, pages 153–171. NATO ASI Series Vol. F45, 1986.

[MSTG89] R. Mooney, J. Shavlik, G. Towell, and Alan Gove. An experimental comparison of symbolic and connectionist learning algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 775–780. Morgan Kauffman, San Mateo, CA, 1989.

[Mug92] S. Muggleton. *Inductive Logic Programming.* Academic Press, San Diego, 1992.

[Nat91] B. K. Natarajan. *Machine Learning: A Theoretical Approach.* Morgan Kauffman, San Mateo, CA, 1991.

[Nil65] N.J. Nilsson. *The Mathematical Foundations of Learning Machines.* McGraw-Hill, New York, 1965.

[OBS92] A. Okabe, B. Boots, and K Sugihara. *Spatial tessellations : concepts and applications of Voronoi diagrams.* Wiley and Sons, Chichester, England, 1992.

[OG92] J. Oncina and P. García. Inferring regular languages in polynomial update time. In N. Pérez *et al*, editor, *Pattern Recognition and Image Analysis*, pages 49–61, World Scientific, New Jersey, 1992.

[OM94] D. Ourston and R. J. Mooney. Theory refinement: Combining analytical and empirical methods. *Artificial Intelligence*, 66:273–310, 1994.

[OS95] D. W. Opitz and J. W. Shavlik. Dynamically adding symbolically meaningful nodes to knowledge-based neural networks. *Knowledge-Based Systems*, 8(6):301–311, 1995.

[OS97] D. W. Opitz and J. W. Shavlik. Connectionist theory refinement: Genetically searching the space of network topologies. *Journal of Artificial Intelligence Research*, 6:177–209, 1997.

[PC78] T. Pao and J. Carr. A solution of the syntactic induction-inference problem for regular languages. *Computer Languages*, 3:53–64, 1978.

[PF91] S. Porat and J. Feldman. Learning automata from ordered examples. *Machine Learning*, 7:109–138, 1991.

[PH93] R. G. Parekh and V. G. Honavar. Efficient learning of regular languages using teacher supplied positive examples and learner generated queries. In *Proceedings of the Fifth UNB Conference on AI, Fredricton, Canada*, pages 195–203, 1993.

[PH96] R. G. Parekh and V. G. Honavar. An incremental interactive algorithm for regular grammar inference. In L. Miclet and C. Higuera, editors, *Proceedings of the Third ICGI-96, Montpellier, France, Lecture Notes in Artificial Intelligence 1147*, pages 238–250, 1996.

[PH97] R. G. Parekh and V. G. Honavar. Learning DFA from simple examples. In *Proceedings of the Eighth International Workshop on Algorithmic Learning Theory (ALT'97), Sendai, Japan, Lecture Notes in Artificial Intelligence 1316*, pages 116–131, 1997. Also presented at the *Workshop on Grammar Inference, Automata Induction, and Language Acquisition (ICML'97), Nashville, TN*, 1997.

[PH98a] R. G. Parekh and V. G. Honavar. Grammar inference, automata induction, and language acquisition. In Dale, Moisl, and Somers, editors, *Handbook of Natural Language Processing*. Marcel Dekker, 1998. (To appear).

[PH98b] R. G. Parekh and V. G. Honavar. Constructive theory refinement in knowledge based neural networks. In *Proceedings of the International Joint Conference on Neural Networks'98, Anchorage, AK*, 1998. (To appear).

[Pit89] L. Pitt. Inductive inference, DFAs and computational complexity. In *Analogical and Inductive Inference, Lecture Notes in Artificial Intelligence 397*, pages 18–44. Springer-Verlag, New York, 1989.

[PK92] M. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9:57–94, 1992.

[PNH97] R. G. Parekh, C. Nichitiu, and V. G. Honavar. A polynomial time incremental algorithm for regular grammar inference. Technical Report ISU-CS-TR97-03, Iowa State University, Ames, Iowa, 1997.

[Pom89] D. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. Technical Report CMU-CS-89-107, Carnegie Mellon University, Pittsburgh, PA, 1989.

[Pou95] H. Poulard. Barycentric correction procedure: A fast method of learning threshold units. In *Proceedings of WCNN'95 (vol. 1), Washington D.C.*, pages 710–713, 1995.

[PW88] L. Pitt and M. K. Warmuth. Reductions among prediction problems: on the difficulty of predicting automata. In *Proceedings of the* 3^{rd} *IEEE Conference on Structure in Complexity Theory*, pages 60–69, 1988.

[PW89] L. Pitt and M. K. Warmuth. The minimum consistency dfa problem cannot be approximated within any polynomial. In *Proceedings of the* 21^{st} *ACM Symposium on the Theory of Computing*, pages 421–432, 1989.

[PYH95] R. Parekh, J. Yang, and V. Honavar. Constructive neural network learning algorithms for multi-category classification. Technical Report ISU-CS-TR95-15a, Iowa State University, Ames, IA, 1995.

[PYH97a] R. G. Parekh, J. Yang, and V. G. Honavar. Constructive neural network learning algorithms for multi-category real-valued pattern classification. Technical Report ISU-CS-TR97-06, Iowa State University, Ames, IA, 1997. (Submitted for review to the IEEE Transactions on Neural Networks).

[PYH97b] R. G. Parekh, J. Yang, and V. G. Honavar. Mupstart - a constructive neural network learning algorithm for multi-category pattern classification. In *Proceedings of the IEEE/INNS International Conference on Neural Networks, ICNN'97*, pages 1924–1929, 1997.

[PYH97c] R. G. Parekh, J. Yang, and V. G. Honavar. Pruning strategies for constructive neural network learning algorithms. In *Proceedings of the IEEE/INNS International Conference on Neural Networks, ICNN'97*, pages 1960–1965, 1997.

[PYH98] R. G. Parekh, J. Yang, and V. G. Honavar. An experimental comparison of constructive neural network learning algorithms, 1998. (In preparation).

[Qui86] R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Ree93] R. Reed. Pruning algorithms — a survey. *IEEE Transactions on Neural Networks*, 4(5):740–747, 1993.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations into the Microstructure of Cognition*, volume 1 (Foundations). MIT Press, Cambridge, MA, 1986.

[Rip96] B. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, New York, 1996.

[RM95] B. Richards and R. Mooney. Automated refinement of first-order horn-clause domain theories. *Machine Learning*, 19:95–131, 1995.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[RS93] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[SRK95] K-Y. Siu, V. Roychowdhury, and T. Kailath. *Discrete Neural Computation - A Theoretical Foundation*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[ST91] J. Saffery and C. Thornton. Using stereographic projection as a preprocessing technique for upstart. In *Proceedings of the International Joint Conference on Neural Networks*, pages II 441–446, IEEE Press, New York, 1991.

[TB73] B. Trakhtenbrot and Ya. Barzdin. *Finite Automata: Behavior and Synthesis*. North Holland Publishing Company, Amsterdam, 1973.

[Thr95] S. Thrun. Lifelong learning: A case study. Technical Report CMU-CS-95-208, Carnegie Mellon University, Pittsburgh, PA, 1995.

[TS93] G. Towell and J. Shavlik. Extracting rules from knowledge-based neural networks. *Machine Learning*, 13:71–101, 1993.

[TS94] G. Towell and J. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1–2):119–165, 1994.

[TSN90] G. G. Towell, J. W. Shavlik, and M. O. Noordwier. Refinement of approximate domain theories by knowledge-based neural networks. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 861–866, 1990.

[Val84] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[VB87] K. Vanlehn and W. Ball. A version space approach to learning context-free grammars. *Machine Learning*, 2:39–74, 1987.

[Wat89] C. J. Watkins. *Learning from Delayed Rewards*. PhD dissertation, King's College, Cambridge, UK, 1989.

[WD92] C. J. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[Wer74] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in Behavioral Sciences*. PhD dissertation, Harvard University, 1974.

[YH96] J. Yang and V. Honavar. A simple randomized quantization algorithm for neural network pattern classifiers. In *Proceedings of the World Congress on Neural Networks'96, San Diego, CA*, pages 223–228, 1996.

[YH97] J. Yang and V. Honavar. Feature subset selection using a genetic algorithm. In *Proceedings of the Genetic Programming'97, Stanford, CA*, pages 380–385, 1997.

[YH98] J. Yang and V. Honavar. Experiments with the cascade-correlation algorithm. *Microcomputer Applications*, 1998. (To appear).

[YPH96] J. Yang, R. Parekh, and V. Honavar. MTiling - a constructive neural network learning algorithm for multi-category pattern classification. In *Proceedings of the World Congress on Neural Networks'96, San Diego, CA*, pages 182–187, 1996.

[YPH98a] J. Yang, R. Parekh, and V. Honavar. Comparison of performance of variants of single-layer perceptron algorithms on non-separable datasets. (Submitted for review to the Journal of Artificial Neural Networks), 1998.

[YPH98b] J. Yang, R. Parekh, and V. Honavar. DistAI: An inter-pattern distance-based constructive learning algorithm. In *Proceedings of the International Joint Conference on Neural Networks'98, Anchorage, AK*, 1998. (To appear).