National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

On Efficient Learning Algorithms for Neural Networks

by

Mostefa Golea

Thesis
submitted to the University of Ottawa
in partial fulfillment of the
requirements for the degree of
Ph.D. in Physics

Ottawa-Carleton Institute for Physics
University of Ottawa
Ottawa, Canada

ISBN 0-315-82518-9

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

Inductive Inference Learning can be described in terms of finding a good approximation to some unknown classification rule f, based on a pre-classified set of training examples $< \mathbf{x}, f(\mathbf{x}) >$. One particular class of learning systems that has attracted much attention recently is the class of neural networks. But despite the excitement generated by neural networks, learning in these systems has proven to be a difficult task. In this thesis, we investigate different ways and means to overcome the difficulty of training feedforward neural networks. Our goal is to come up with efficient learning algorithms for new classes (or architectures) of neural nets. In the first approach, we relax the constraint of fixed architecture adopted by most neural learning algorithms. We describe two constructive learning algorithms for two-layer and tree-like networks. In the second approach, we adopt the "probably approximately correct" (PAC) learning model and we look for positive learnability results by restricting the distribution generating the training examples, the connectivity of the networks, and/or the weight values. This enables us to identify new classes of neural networks that are efficiently learnable in the chosen setting. In the third and final approach, we look at the problem of learning in neural networks from the average case point of view. In particular, we investigate the average case behavior of the well known clipped Hebb rule when learning different neural networks with binary weights. The arguments given for the "efficient learnability" range from extensive simulations to rigorous mathematical proofs.

Acknowledgments

I would like to express my gratitude to my supervisor, Dr. Mario Marchand, under whom it has been a great pleasure to study. I am grateful to Mario for introducing me to the field of neural networks, for arranging my financial support, and for his invaluable guidance, encouragement and patience throughout the course of this work. In addition I also thank him for putting up with the tortuous task of reading the preliminary drafts of this thesis.

Some of the work in this thesis is collaborative, and I take this opportunity to thank my co-authors Thomas Hancock and Paul Ruján. Working with Tom has been a great pleasure, and my deep thanks go to him for sharing with me his invaluable knowledge of the PAC model.

I am very grateful to Dr. Gary Slater and to IBM System Engineering Representative in Ottawa, Toby Taylor, for giving us precious CPU time on their IBM RS/6000 workstations, and to Dr. Robert Holte for providing us with the data sets to test our algorithms.

I thank the faculty members at the Physics Department (U.O), particularly the professors whose classes I attended. I also thank Bei Wah Chan (T.A.'s best friend!), Lorainne Gravelle, Bob Hart, Mike Jackson, Roger Chagnon, Robert Parent, and all the graduate students and the staff of the department for their friendship.

Thanks are due to my Algerian friends who helped to make my life as a foreign student a little more bearable. Special thanks go to the Boucekkine family for its open door policy!

Last but not the least, I take this opportunity to dedicate this thesis to my parents who must have a great deal of knowledge about "learning" because, after more than two decades in school and hundreds of classes, I still believe they have been my best teachers and that our home was the best class I have ever attended.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

The Scope of the Thesis

Despite the excitement generated recently by neural networks, from both the theoretical and practical perspectives, learning in these systems has proven to be a difficult task. The most popular learning algorithm for neural networks, known as *back-propagation*, rests on very shaky theoretical foundations as there is no guarantee it will converge to a good solution. In practice, it often runs very slowly and gets trapped in local minima. Another serious drawback of the back-propagation algorithm is that it gives no hint whatsoever about how big a network should be to learn a given task.

In this thesis, we will look at different ways and means to overcome the difficulty of training feedforward neural networks.

The first approach we will take is to try to come up with efficient *constructive* learning algorithms for different types of neural architectures. By constructive learning we mean that the architecture of the network is not assumed to be known a *priori*. Rather, it is the task of the algorithm to infer, from the training sample, both the architecture and the weight values necessary to solve the given task.

The second approach we will take is to restrict the distribution generating the training examples, the connectivity of the network, and/or the values of the weights. This will enable us to identify new classes of neural networks that are *efficiently* learnable, either in a *worst case* or an *average case* sense. The arguments given for the efficient learnability of these new classes will range from extensive simulations to rigorous mathematical proofs.

The technical results of this thesis are divided into three parts. The first part (chapters 4

and 5) deals with the problem of *constructing* neural network architectures from training samples.

The second part deals with the problem of learning different types of neural architectures within the "probably approximately correct" (PAC) model in the case where the distribution of examples is a product or the uniform one (for the definition of of the PAC model, see chapter 6). More specifically, in chapter 7 we investigate the PAC learnability of halfspace intersections and neural decision lists under the uniform distribution. In chapters 8 and 9, we describe efficient learning algorithms for single perceptrons and special classes of nonoverlapping perceptron networks, when the weights are binary valued and the distribution generating the examples is a product distribution or the uniform distribution. Chapter 10 deals with the problem of learning networks with binary weights in noisy or uncertain environments.

The last part (chapters 11 and 12) presents an average case analysis of a particular algorithm (the clipped Hebb rule) when learning single perceptrons and layered nonoverlapping perceptron networks with binary weights.

Finally, chapter 13 contains the main conclusions of this research and some open problems.

The remainder of the present part provides the background necessary to understand this thesis. Chapter 2 describes the basic model for *concept learning*. A more sophisticated model of learning (the PAC model [1]) will be introduced in chapter 6. Chapter 3 serves both as an introduction and a review of the field of neural networks.

[1] We did postpone the introduction of the PAC model to chapter 6 for two reasons. First, we thought it would be too much to introduce both a new field and a new model. Second, we will not make use of the PAC model in the Part on constructive algorithms.

Publication Notes

Much of the results reported in this thesis appears or will appear as publications:

- The results of chapter 4 are joint work with Mario Marchand and Paul Ruján, which appear in *Europhysics Letters*, Vol. 11, p. 487, 1990.

- Chapter 5 is joint work with Mario Marchand and appears in *Europhysics Letters*, Vol. 12, p. 205, 1990.

- Chapter 7 is joint work with Mario Marchand that will appear in *Network: Computation in Neural Systems*, Vol. 4, p. 67, 1993 (in press). Two short versions of this chapter will appear in the *Proceedings of the World Congress on Neural Networks*, WCNN'93.

- Chapters 8 and 11 are joint work with Mario Marchand and will appear in *Neural computation*.

- Chapter 9 is joint work with Thomas Hancock and Mario Marchand and part of it appears in *Advances in Neural Information Processing Systems*, Vol. 5, p. 591, 1993.

- Chapter 12 is joint work with Mario Marchand, currently submitted for publication in *Journal of Physics A*. A short version of this chapter will appear in *Computational Learning Theory*, 1993.

- The extension mentioned in chapter 13 is joint work with Thomas Hancock and Mario Marchand, which appears as Technical Report-26-91, Center for Research in Computing Technology, Harvard University, and it is accepted for publication in *Machine Learning*.

Chapter 2

Learning about Learning

2.1 Introduction

An old dream of philosophers, science-fiction writers, poets, and fools is to design a machine that acts like a human, thinks like a human, behaves and works like a human, but faster, stronger, smoother, and smarter. Think of Frankenstein's monsters, of the miraculous chess-playing machines of the 19^{th} century, and of the many other legends and fake attempts.

With the advances in the computing technologies, science got involved in the matter. The search for "intelligent machines" intensified, but now in a much "down-to-earth" approach. Realistic goals were set. The "human-like behavior" was gradually specified. Basic problems were identified, fundamental concepts defined, and working prototypes demonstrated. The emphasis was on machines that can perform tasks which normally require human intelligence. Such tasks include vision, natural language understanding, problems solving, decision-making, and many more. The fields of *Artificial Intelligence* and *Machine Learning* were born.

But what happened to the old dream? Well, it is still an *ola* dream.

2.2 Machine Learning

If a machine is to behave intelligently, it must be able to interact efficiently with its environment and to learn from this interaction. Here, we will concentrate on the learning

process.

An infant learns how to discern three-dimensional pictures during the first days of life, but try to teach a computer to do the same. A parent needs a few minutes and two pictures to show a child the difference between a tiger and a lion. However, nobody has yet succeeded in explaining the difference to a computer.

Like *intelligence* itself, the concept of *learning* is easier to recognize than to define [1]. Loosely speaking, a learning system must have the ability to *improve* its "performance" as more and more information become available to it.

Making machines and computers learn by themselves as humans do is the main topic of the field of "machine learning". Learning methodologies investigated within this field include learning from examples, learning by analogy, learning by observation, deductive learning, to name but few. Here, we focus on one type of learning, namely learning from examples.

2.3 Learning Concepts from Examples

2.3.1 Generalities

One major learning task is *classification*, also referred to as *prediction* or *pattern recognition*. This task involves the ability to classify data into different categories/classes. One example of data classification is the weather prediction. A forecast is based on certain weather measurements, say atmospheric pressure values at a number of different stations. Suppose one wishes to predict whether or not it will rain tomorrow. In effect one must be able to place a given set of measured weather data into one of two classes: (1) those data that indicate rain tomorrow, and (2) those data that do not. To be successful, the classification must be performed in such a way that the resulting forecast and the actual outcome are in close agreement. Other examples of classification tasks are:

- the diagnosis of a medical condition from symptoms, in which the classes could be the various disease states.

[1] The "Potter Stewart" mode of concept definition: "I know it when I see it" may be of some help here!

- determining the game-theoretic value of a chess position, with the classes *won for white*, *lost for white*, and *drawn*.

- predicting whether the stock market will go up or down over a period of time. Here the two classes are: stock average goes up and stock average goes down.

It might appear that classification tasks are only a tiny subset of the learning phenomena, but many learning situations can be recast as classification problems [84].

In a typical classification problem, the learning system has available to it a finite set of solved examples: a training sample. The data for each example consists of a pattern of measurements/observations and a corresponding correct class. The set of potential measurements relevant to a particular problem are referred to as *features*, *attributes*, or *input variables*. The goal of the learning system is to extract from the training sample a decision rule that can determine the class of any pattern of measurements. The decision rule is expressed using a pre-chosen classifier, such as a logical formula, a decision tree, or a neural net. *Learning* consists in choosing, through training, the parameters within the classifier that work well for the examples at hand and, more importantly, for new patterns of measurements. The classifier works well if it can

1. predict accurately the class of any pattern of measurements in the training sample. This is called **memorization**.

2. predict accurately the class of any *new* pattern of measurements. This is called **generalization**.

The distinction between memorization and generalization is of fundamental importance. Unfortunately, some researchers do not realize this. The ability to *generalize* requires some kind of memorization but the ability to *memorize* does not imply by itself the ability to generalize. To see that, assume that each pattern in the training sample is stored in a table along with its correct class. This simple procedure, often called the look-up table solution, solves efficiently the memorization problem. Unfortunately, this solution has no generalization ability whatsoever. This cannot be called learning. To be able to generalize, the learning system must be able to go beyond mere memorization. It must be able to extract, from the

training sample, a "sufficient" amount of information about the classification problem. That is, to capture the *correlations* between the patterns and the corresponding classifications. As we will see, this amounts generally to some kind of data-compression or memorization with minimum resources. The ability to generalize is *the* fundamental property of any learning system.

Before we go further, let us say a few words about how one goes about the business of training and testing classifiers. For a given classification problem, one is given a sample of solved examples. This sample is then partitioned, at random, into two disjoint groups. The first group is used to train the classifier using a learning system (the learning phase). The learning system is usually an algorithm run on a conventional computer or implemented on specialized hardware. Once the training is done, the second group is used to estimate the the classifier accuracy, *i.e.* its generalization ability (the testing phase). The process of training and testing may be repeated many times, with different partitions of the sample, to get an average estimate of the generalization. The classification problem maybe a real one, like weather prediction, or an artificial one, *e.g.* computer-generated for test purposes.

In what follows, we will assume that there are only *two classes* in which the patterns are to be classified. This special case is the most extensively investigated and is referred to, in the field of machine learning, as *concept learning.* Loosely speaking, the concept is associated with one of the two classes and one wants to learn to predict whether or not a given pattern is in the concept. For example, if one wants to classify animals into two classes: "dog" and "not a dog", the concept to learn is "dog".

2.3.2 A Model for Concept Learning

In this section, we try to quantify the problem of concept learning and express it in somewhat mathematical terms.

We will denote by n the number of features. Each feature will be represented by an input variable x_i $(i = 1, \ldots, n)$, and a pattern of measurements will be represented by a vector (or a point) $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. We will sometimes refer to patterns as *input vectors* or *instances.* The set of all allowed patterns $\mathbf{x}$ is called the *input space*, denoted by $\mathcal{X}^n$. For example, if each input variable takes on values in $\mathcal{R}$ (the set of real numbers), the input space

is the n-dimensional Euclidean space E^n. On the other hand, if each input variable takes on values in $\{0,1\}$ or $\{-1,1\}$, the input space is the n-dimensional *hypercube* [2]. Note that we are using x_i to denote both the input variable and the values it takes on. The meaning will be clear from the context.

As we said earlier, we will be dealing mainly with two-class classification problems. We label the first class *positive* and we denote it by $+1$, or simply $(+)$. We label the second class *negative* and we denote it by 0, -1, or simply $(-)$. The concept to learn will be associated with the positive class. An input vector/instance $\mathbf{x}$ is labeled a *positive example* if it is in the concept (the positive class) and is labeled a *negative example* otherwise.

We assume that the learning system receives *examples* labeled by a process that we call the *teacher* [3] already in possession of the *target concept*, *i.e.* the concept the system is trying to learn. We may simulate the labeling process by an underlying function f, called the target function, such that

$$f(\mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{x} \text{ is in the target concept} \\ -1 & \text{if } \mathbf{x} \text{ is not in the target concept} \end{cases} \tag{2.1}$$

$$\tag{2.2}$$

using the ± 1 class notation. We will abuse the terminology somewhat and use f to refer to the target concept itself. Then $\mathbf{x}$ is said to be a positive example if $f(\mathbf{x}) = +1$ and is said to be a negative example if $f(\mathbf{x}) = -1$.

The decision rule (or the classifier) found by the learning system is referred to as the *hypothesis concept* or simply the hypothesis. We may simulate the labeling process done by the hypothesis concept by a function h, called the hypothesis function, such that

$$h(\mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{x} \text{ is in the hypothesis concept} \\ -1 & \text{if } \mathbf{x} \text{ is not in the hypothesis concept} \end{cases} \tag{2.3}$$

$$\tag{2.4}$$

Again, we will abuse the terminology and use h to refer to the hypothesis concept itself.

We are now ready to state the problem of *concept learning* in mathematical terms:

---

[2] The hypercube is an generalization of the cube for $n > 3$.

[3] Because of the presence of the teacher, this kind of learning is called *supervised learning*.

- Given a classifier with adaptive parameters.
- Given a training sample that consists of a set of examples labeled according to some unknown target concept f:

$$< \mathbf{x}^{(1)}, f(\mathbf{x}^{(1)}) >, \quad < \mathbf{x}^{(2)}, f(\mathbf{x}^{(2)}) >, \ \ldots, \quad < \mathbf{x}^{(m)}, f(\mathbf{x}^{(m)}) > .$$

- Find, in a reasonable time, a setting of the parameters of the classifier in such a way that the resulting classifier (the hypothesis concept h) and the target concept f agree on the labeling of most (if not all):
  - patterns in the training sample (memorization).
  - new unseen patterns (generalization).

This is a very simplistic model. In fact, we have omitted many details like how the training sample is generated, how many training examples are needed, how much time the learning system is allowed to produce its hypothesis, and many more. Nevertheless, this model captures the main ideas of concept learning that will help us understand learning in neural networks. Later on, we will switch to a more formal learning model (the PAC model). For now, we are more than satisfied with our model (if it is not broken, why fix it!).

In this thesis, we will confine our attention to one specific classifier model: *Neural Networks*. An introduction to this interesting topic is given in the next chapter.

Chapter 3

Learning about Neural Networks

> ...as is being argued more and more in the neural network community, the field will undoubtedly have to get over a period of extreme "hype", where almost everyone is attempting to apply the technique to something. Hardly ever before have so many people started to use a system they have understood as little as neural networks. This is not surprising because with the help of self-organization, neural networks are systems "one does not have to understand to get results"!
> Georg Dorffner [26]

3.1 Introduction

In this chapter, we introduce the reader to the fi[illegible] of neural networks. The results included here are of previous work of other researchers.

In section 3.3 we review the perceptron model, its learning algorithm, and its limitations. The notion of neural networks is introduced in section 3.4 and the special class of feedforward neural networks in section 3.5. As the latter is the topic of this study, we focus our attention on it. We review its geometric properties in section 3.5.1. Back-propagation, the most popular learning algorithm for this class of networks, is presented in section 3.5.2, along with a thorough discussion of its shortcomings. The negative results about training neural networks with fixed architectures are given in section 3.6. An alternative approach to training networks based on constructive algorithms is introduced in section 3.7. Finally, section 3.8

is a glance at what is coming in this thesis.

This chapter provides the background necessary to understand this thesis.

3.2 From Biological Neurons to Artificial Neurons

The most sophisticated and powerful problem solving machine is the human brain. From a circuit point of view, the brain is a huge network of interconnected processing elements, called *neurons*, that exchange signals through synaptic links. The number of neurons in the brain is roughly 10^{10}–10^{11}. On average, each neuron is connected to some 10^3–10^4 neighbors and can fire around 10^3 electrical pulses/second, which is not that impressive. However, because neurons can operate in parallel, the brain is able to perform some 10^{16}–10^{18} operations/second (for more information on biological and physiological issues, see [24]). From a learning point of view, the brain is the most sophisticated adaptive, learning machine we do have up to now. So it is not surprising that people, in their quest for learning systems, turned first to this amazing device for a role model. Unfortunately (or fortunately!), the brain is too complicated to be fully understood. So, the approach was to abstract out some simple features of the brain, investigate them, and go from there. The best place to start from is obviously the single neuron level.

From the biological inspiration of nerve cells in the brain, McCulloch and Pitts proposed, in 1943, a mathematical model of a neuron as a binary threshold unit [71]. The model neuron, called *artificial neuron*, is equipped with a finite number of input ports through which signals may be received. A weight factor is associated with each input port indicating its importance. Both the inputs and the weights are represented by integer or real numbers. The first step performed by the artificial neuron is the calculation of the weighted sum of its inputs. Then, the result is used as argument for a function which describes the activity of the artificial neuron. This function is normally a binary threshold function that outputs 1 or 0 according to whether the weighted sum is above or below a certain threshold. This crude model was called later the *perceptron* [86]. In the next section, we describe in mathematical terms the functioning of the perceptron.

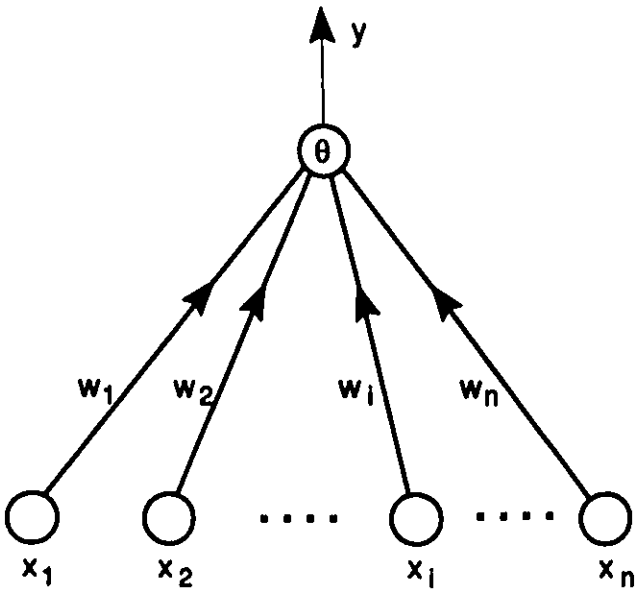

Figure 3.1: The general model of a perceptron: $(x_1, \ldots, x_n)$ are the input variables, $(w_1, \ldots, w_n)$ are the weights, θ is the threshold, and y is the output which may take on values in either $\{0, 1\}$ or $\{-1, 1\}$.

3.3 The Perceptron

As we said earlier, the perceptron is an abstraction of the biological neuron. Its goes by many names including threshold unit, linear threshold function, halfspace, (artificial) neuron.

The perceptron computes a weighted sum of its inputs, and outputs [1] 1 or -1 according to whether this sum is above or below a certain threshold θ. The weights and the threshold may take on any values in $\mathcal{R}$. A schematic representation of the perceptron is shown in fig. 3.1.

Let $\mathbf{w}$ denotes the weight vector $(w_1, \ldots, w_n)$ and let θ denotes the threshold. Let g denotes the function (mapping) performed by the perceptron. Then, for an input vector $\mathbf{x}$

$$y \equiv g(\mathbf{x}) = \begin{cases} +1 & \mathbf{w} \cdot \mathbf{x} \geq \theta \\ -1 & \mathbf{w} \cdot \mathbf{x} < \theta \end{cases} \tag{3.1}$$

[1] Depending on the situation, we will switch back and forth between the two representations $\{-1, +1\}$ and $\{0, +1\}$ for the perceptron's output.

where $\mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{n} w_i x_i$. Equation 3.1 can be also written as:

$$y = \text{sgn}(\mathbf{w} \cdot \mathbf{x} + w_0) \tag{3.2}$$

where $w_0 = -\theta$ is referred to as the bias, and $\text{sgn}(a) = +1$ if $a \geq 0$ and -1 otherwise.

The perceptron can be viewed as just another type of classifier (section 2.3). It produces an output indicating x's membership in class $+1$ (if $y = +1$) or class -1 (if $y = -1$). The weights and the threshold are the adaptive parameters of this model. As with other classifiers, the main task is to train the perceptron in such a way as to produce the right output (class) for any input vector x. This amounts to finding a setting of $(w_1, \ldots, w_n)$ and θ such that $g(\mathbf{x}) = +1$ wnenever $\mathbf{x}$ belongs to the positive class and $g(\mathbf{x}) = -1$ whenever $\mathbf{x}$ belongs to the negative class. Fortunately, there is a simple training procedure, called the *perceptron training rule*, that finds such setting whenever it does exist [27, 74, 76, 86]. In order to describe this procedure, we appeal to our model for learning concepts from examples (section 2.3.2).

We are given a sample of solved (classified) examples

$$< \mathbf{x}^{(1)}, f(\mathbf{x}^{(1)}) >, < \mathbf{x}^{(2)}, f(\mathbf{x}^{(2)}) >, \ldots, < \mathbf{x}^{(m)}, f(\mathbf{x}^{(m)}) > .$$

and our task is to find a setting of $\mathbf{w}$ and θ that classifies correctly every example in the training sample. The simple procedure given in fig. 3.2 finds such setting.

The perceptron learning rule is an *error-correcting* procedure because adjustments are made only when an error occurs in the classification. We state the following well known theorem (see [27, 74, 76] for a proof):

Theorem 3.1 The Perceptron Learning Rule *converges to a solution, if one exists, in a finite number of steps.*

This convergence theorem may appear to be quite strong, but it has its downs. First, if no solution exists, the procedure will not converge and will keep on cycling through the sample forever. Second, the "finite number of steps" the theorem is talking about may be a huge number and so, the time needed for convergence may be very long. Nevertheless, in light of what is coming, it is indeed a remarkable theorem!

The Perceptron Learning Rule

1. Initialization: $w_i = 0$ (for $i = 1, \ldots, n$) and $\theta = 0$.

2. Cycle through the training sample: for $j = 1, \ldots, m$,

   (a) If $\mathbf{x}^{(j)}$ is correctly classified, then do nothing.

   (b) Otherwise, set

$$w_i \leftarrow w_i + x_i^{(j)} f(\mathbf{x}^{(j)}) \quad \text{for } i = 1, \ldots, n$$

   and

$$\theta \leftarrow \theta - f(\mathbf{x}^{(m)})$$

3. If during a cycle no changes are made to $\mathbf{w}$ and θ, stop and output $\mathbf{w}$ and θ. Else go to step 2.

Figure 3.2: The perceptron learning rule.

There exists other, more efficient methods for training perceptrons. For example, the training problem can be formulated as a linear programming instance for which there exist very powerful algorithms [56].

As to the generalization ability of the perceptron, it is enough here to say that if the training sample is sufficiently large, the trained perceptron will be able to generalize. This will become clear from the results of chapter 6.

The perceptron model saw its best days in the 50's and 60's. Several researchers experimented then with this model. Problems tackled included computer vision, weather prediction, speech recognition, medical diagnosis, and image classification. Unfortunately, those researchers did not pay enough attention to some of the fundamental theoretical issues associated with this model: the most important of these issues is what kinds of functions (problems) are solvable by perceptrons. This inattention was to prove fatal.

The following geometric interpretation of eq. 3.1 will help us understand the limitations of perceptrons in terms of what types of functions they can represent.

Let S be the surface in the n-dimensional input space defined by

$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = \theta \tag{3.3}$$

For $n = 2$, S is a line; for $n = 3$, S is a plane; and for $n \geq 4$, S is a *hyperplane*. Looking back now to eq. 3.1, it is easy to see that a perceptron assigns $+1$ to input vectors above the hyperplane S (*i.e.* $\mathbf{w} \cdot \mathbf{x} \geq \theta$) and -1 to input vectors below the hyperplane S (*i.e.* $\mathbf{w} \cdot \mathbf{x} < \theta$). In other words, a perceptron defines a hyperplane that partitions the input space into two regions (halfspaces):

$$R_1 = \{\mathbf{x} : \mathbf{w} \cdot \mathbf{x} \geq \theta\}$$

$$R_2 = \{\mathbf{x} : \mathbf{w} \cdot \mathbf{x} < \theta\}$$

All inputs from R_1 are mapped into $+1$ and all inputs from R_2 are mapped into -1. This is shown in fig. 3.3 for $n = 2$.

From this geometric point of view, we can see that a function f is learnable by a perceptron *if and only if* (iff) there exists a hyperplane $\mathbf{w} \cdot \mathbf{x} = \theta$ that *separates* f's positive examples from f's negative examples. In this case f is said to be *linearly separable*. If no such hyperplane exists, f is said to be not linearly separable.

The most famous (or, rather, infamous) example of a non-linearly separable function is the "exclusive or" XOR (fig. 3.4): $f(x_1, x_2) = +1$ iff $x_1 \neq x_2$.

Unfortunately, as the number of inputs n increases, the probability of a function f being linearly separable decreases *exponentially* [22, 76]. Thus, in high dimensions most functions are not linearly separable and so, not learnable by perceptrons. Worse, Minsky and Papert [74] proved some stronger negative results about what kinds of functions can be computed using *finite-order* perceptrons[2].

The natural way to overcome the limitations of single perceptrons is to hook up many perceptrons to form a network. For example, the outputs of several perceptrons could be used as inputs to another perceptron whose output will represent the network's output. Such *neural networks* are potentially more powerful than single perceptrons. This is the topic of the next section.

[2]Instead of a weighted sum of the individual variables, a finite-order perceptron uses a weighted sum of combinations, for example products, of the input variables. For details, see [74].

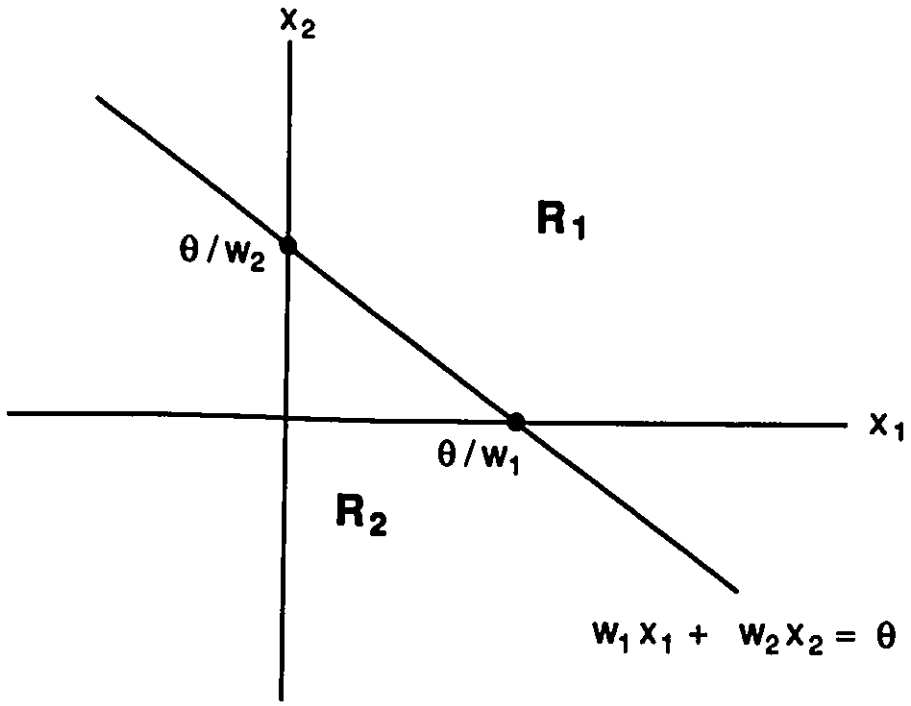

Figure 3.3: In two-dimensions, a perceptron defines a line $w_1x_1 + w_2x_2 = \theta$ that divides the input space into two regions, R_1 and R_2. All points in R_1 are mapped to $+1$ and all points in R_2 are mapped to -1.

3.4 Neural Networks

There are many ways to hook up many perceptrons to form a network and depending on the way this is done, a variety of different architectures is possible. The most commonly studied architectures are:

1. Recursive neural networks: here a neuron may be connected to any other neuron in the network. The special case of fully connected networks is referred to as the *Hopfield model* [50]. For details on this type of architectures, see [32].

2. Feedforward neural networks: here the connections flow in the forward direction starting from the inputs, and no weight connection cycles back to an input or previous

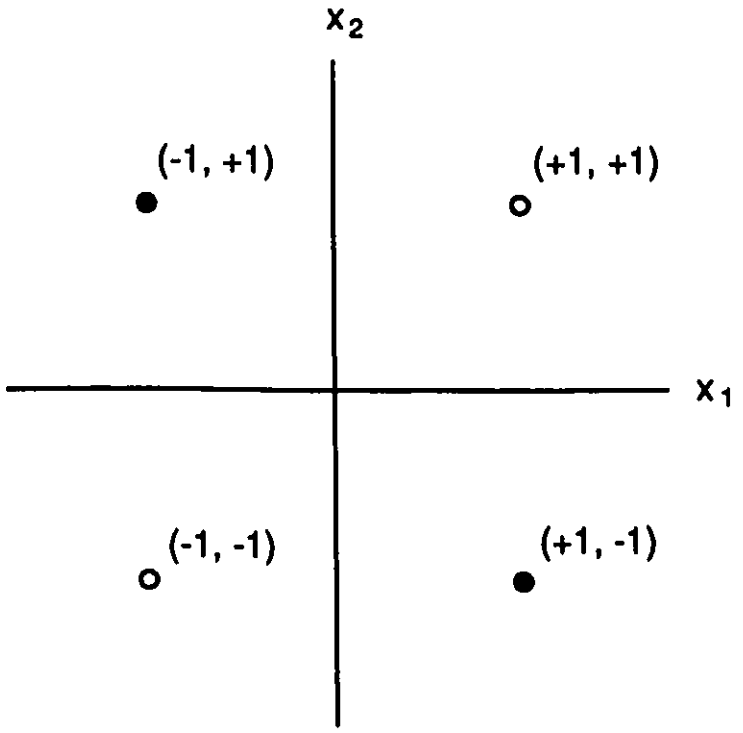

Figure 3.4: The XOR problem: no straight line can separate the positive examples (black circles) from the negative examples (white circles).

neuron. Some interesting special cases are:

(a) Layered feedforward neural networks: here the network is organized in layers (fig. 3.5). The neurons in the first layer receive their inputs from the input variables/nodes. The outputs of the first layer are used as the inputs to the neurons in the second layer, and so on. The neurons in the second and subsequent layers receive as their inputs the output of the neurons in the preceding layer *only*. The output of the single neuron in the final layer is the output of the network. Neurons (layers) other than the output are often called *hidden neurons* (*layers*).

(b) Tree-like feedforward neural networks (see chapter 5).

(c) Cascade feedforward neural networks (see chapter 7).

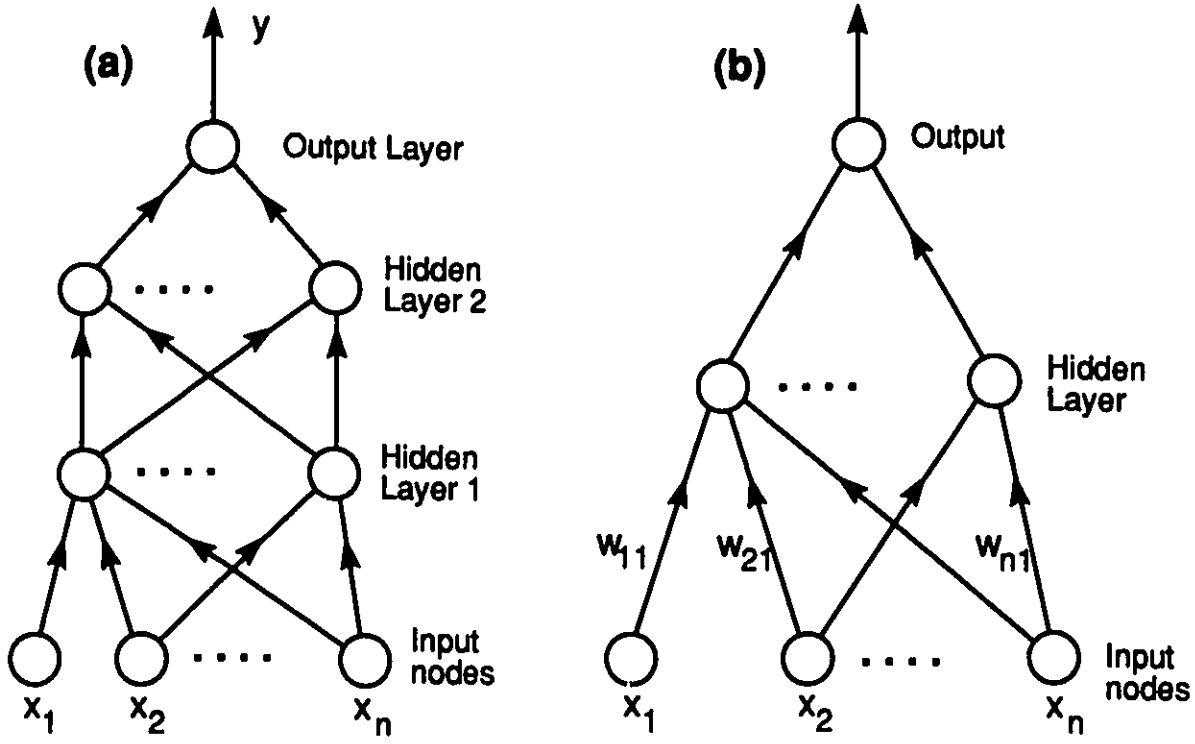

Figure 3.5: (a) A layered feedforward neural network with n input variables/nodes and $L = 3$ layers. The neurons in the first layer receive their inputs from the input variables/nodes. The outputs of the first layer are used as the inputs to the neurons in the second layer, and so on. The neurons in the second and subsequent layers receive as their inputs the output of the neurons in the preceding layer *only*. The output of the single neuron in the final layer is the output of the network. Neurons (layers) other than the output are called *hidden neurons* (*layers*). (b) A two-layer feedforward neural network.

In this thesis, we will deal exclusively with feedforward neural networks.

3.5 Feedforward Neural Networks

Again, we view feedforward neural networks as just another type of classifier. When an input vector $\mathbf{x}$ is presented to the network, the latter produces an output indicating $\mathbf{x}$'s membership in class $+1$ (if $y = +1$) or class -1 (if $y = -1$). The weights and the threshold are the adaptive parameters of the model. As with other classifiers, the main task is to train the network in such a way that it produces the right output (class) for any input vector $\mathbf{x}$. This amounts to finding a setting of the weights and the threshold of each neuron such that the network's output is $+1$ whenever $\mathbf{x}$ is a positive example and -1 whenever $\mathbf{x}$ is a negative example.

The following geometric point of view [76] will help us understand how feedforward neural networks function and why they are hard to train.

3.5.1 A Geometric Approach to Feedforward Networks

Consider the two-layer feedforward network [3] shown in fig. 3.5b. Each hidden neuron (perceptron) defines a hyperplane, which divides the input space into two regions: the neuron's output is $+1$ for points on one side of the hyperplane and -1 for points on the other side of the hyperplane. For each input vector $\mathbf{x}$, the binary outputs of the hidden neurons can be regarded as the components of a vector. If there are h neurons in the hidden layer, these neurons will transform (map) the n-dimensional input vector $\mathbf{x}$ into a h-dimensional vector with binary components called the *internal representation* of $\mathbf{x}$. Thus, each point (vector) in the input space $\mathcal{X}^n$ is mapped into one of the vertices of a h-dimensional hypercube. This hypercube, denoted $\mathcal{I}^h$, is called the internal representation space. This mapping between the input space $\mathcal{X}^n$ and $\mathcal{I}^h$ depends on the values of the weights and thresholds of the neurons in the hidden layer.

The output neuron defines a hyperplane, which divides the internal representation space

[3] The arguments generalize easily to feedforward networks of arbitrary depth L.

$\mathcal{I}^h$ into two regions: the neuron's output is $+1$ for points on one side of the hyperplane and -1 for points on the other side of the hyperplane. Thus, each point in $\mathcal{I}^h$ is mapped into one of the two values $+1$ or -1. These two values represent the two possible responses of the network.

A two-layer feedforward network that solves the XOR problem and the mapping it realizes are shown in fig. 3.6.

Let T^+ (T^-) be the subset of positive (negative) examples in our training sample. In light of what we said, T^+ (T^-) is first mapped into a set Y^+ (Y^-) in $\mathcal{I}^h$ and then into $\{-1, +1\}$. The training problem for the network can be viewed as a problem of finding a setting of the various weights and thresholds such that each example in T^+ is mapped into $+1$ and each example in T^- into -1. An obvious requirement is to keep the internal representations *faithful*: two inputs with different desired outputs must be represented by different points in $\mathcal{I}^h$, since otherwise the output neuron has no chance to distinguish between them. This condition is equivalent to $Y^+ \cap Y^- = \emptyset$. A more stringent requirement is that Y^+ must be linearly separable from Y^-. This is necessary because the output is a single perceptron which can solve only linearly separable problems [4].

The important question is how to choose the various weights and thresholds in the network in order to satisfy the above requirements. The answer is simply that nobody knows how to do it. What makes this training problem hard is the *credit assignment problem* associated with it: whereas we know what should be the network's output for each training example, we do not know what should be the output of a given hidden neuron for a given training example. Should it be -1 or should it be $+1$?

From the historical point of view, the possibility of using networks of connected perceptrons was actively pursued in the 60's [76, 108]. It turned out that these networks are not only more powerful than single perceptrons but also, unfortunately, harder to train.

The limitations of single perceptrons and the inability to derive training procedures for neural networks were significant factors in the demise of the first stage of neural networks

[4] The process of mapping from the input space into the internal representation space may be viewed as *change of variables*. This must be done in such a way that a problem that is not linearly separable in the input representation becomes linearly separable in the internal representation.

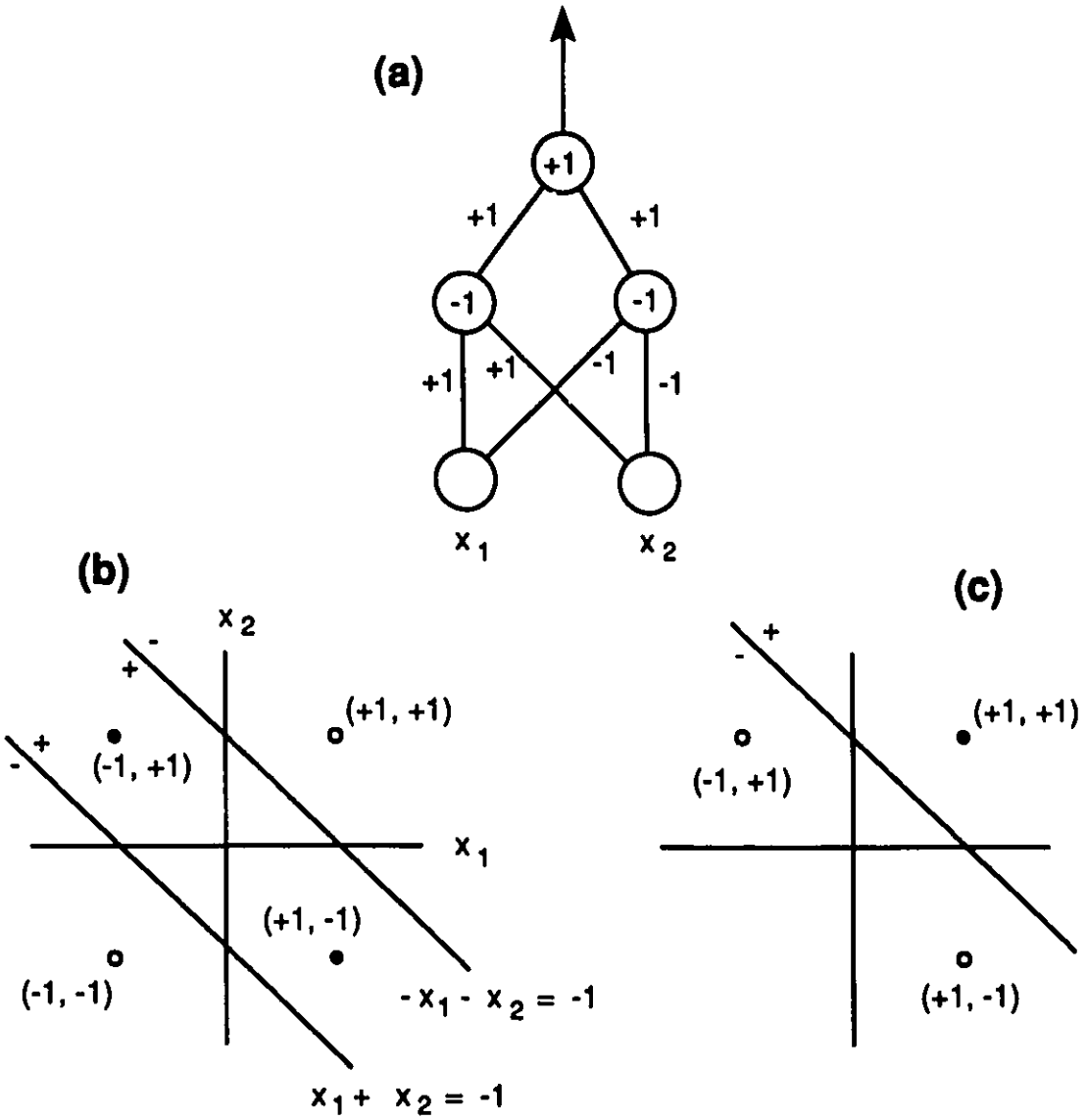

Figure 3.6: (a) a two-layer network that solves the XOR problem. Thresholds are written in the nodes, weights beside edges. (b) The input space: the positive examples are shown as black circles and the negative examples as white circles. Also shown are the two lines defined by the hidden units. (c) The internal representation space. Shown are the internal representations and the line defined by the output unit.

research in the late 60's. Minsky and Papert delivered the field's death certificate in their famous book *Perceptrons* [74]. For people who did not believe in *life after death*, this was the end of it. For the hard core of neural network researchers, it was time to go underground and wait for better times!

The better times came in the mid-80's when Rumelhart and the PDP group [88, 89] found the missing training algorithm, called *back-propagation* [5]. But is back-propagation really the missing piece we were all waiting for? The next section is intended to answer that.

3.5.2 The Back-Propagation Learning Algorithm

Minsky and Papert argued all along that a neural network can not be trained to do anything but the most trivial tasks that are probably easier to solve without a neural network [74]. So, it was a real challenge for the neural network community to prove them wrong. Rumelhart and the PDP group [88, 89] raised to the occasion and came up with the back-propagation; a training algorithm for multilayered neural networks. Back-propagation is essentially a nonlinear optimization method that uses gradient descent as the basis for improving its approximation (solution) at each iteration.

Let $\mathbf{w}$ be the vector representing all the weights and thresholds in the network [6]. Let $h(\mathbf{x}, \mathbf{w})$ denote the network's output for an input vector $\mathbf{x}$ and a given setting of $\mathbf{w}$. As before, we are given a training sample of m classified examples

$$< \mathbf{x}^{(1)}, f(\mathbf{x}^{(1)}) >, < \mathbf{x}^{(2)}, f(\mathbf{x}^{(2)}) >, \ldots, < \mathbf{x}^{(m)}, f(\mathbf{x}^{(m)}) > .$$

and the task is to find a setting of $\mathbf{w}$ in such a way that the network classifies correctly every example in the training sample. For a given setting of $\mathbf{w}$ the network makes an error in classifying a vector $\mathbf{x}^{(j)}$ iff: $h(\mathbf{x}^{(j)}, \mathbf{w}) \neq f(\mathbf{x}^{(j)})$. The quadratic error for a single example $< \mathbf{x}^{(j)}, f(\mathbf{x}^{(j)}) >$ is defined to be

$$E_j(\mathbf{w}) = \left(h(\mathbf{x}^{(j)}, \mathbf{w}) - f(\mathbf{x}^{(j)})\right)^2$$

[5] The back-propagation technique has apparently been discovered independently by several researchers, including P. Werbos (1974), D. Parker (1982), Y. LeCun (1985), and the PDP group (1986).

[6] The thresholds can be treated as ordinary weights.

The total quadratic error for the training sample is then given by

$$E(\mathbf{w}) = \sum_{j=1}^{m} E_j(\mathbf{w}) = \sum_{j=1}^{m} \left(h(\mathbf{x}^{(j)}, \mathbf{w}) - f(\mathbf{x}^{(j)}) \right)^2 \tag{3.4}$$

The total quadratic error $E(\mathbf{w})$ is referred to as the cost function or the energy function of the network. It is a highly nonlinear function of $\mathbf{w}$. Learning may be achieved by minimization of this energy function. This is precisely what back-propagation tries to do using a gradient descent method.

The method of back-propagation allows the $\partial E(\mathbf{w})/\partial w_i$ to be calculated [7]. This gives the $\mathbf{w}$-space gradient of the energy function $\nabla E(\mathbf{w})$. The back-propagation algorithm tries then to minimize $E(\mathbf{w})$ by moving along the gradient $\nabla E(\mathbf{w})$: in its simplest version, each weight is changed by an amount proportional to $\partial E(\mathbf{w})/\partial w_i$

$$\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} - \epsilon \nabla E(\mathbf{w}) \tag{3.5}$$

The parameter ϵ is referred to as the *learning rate*. A sketch of the simplest version of the back-propagation algorithm is given in fig. 3.7. Of course, there are many improved or "accelerated" variations of it, but the main idea is the same.

The discovery of the back-propagation algorithm has dramatically revived the general interest in the neural network field. This algorithm is by far the most popular training algorithm for feedforward neural networks. This is so not because of its solid theoretical foundations, as it has almost none, but because of some spectacular applications. Here we mention only a few:

- The famous NETtalk [92]: the network that could be taught to read aloud by training it on examples of written English text and its correct pronunciation. Upon convergence, the network achieved over 95% accuracy on the training sample (of 1000 words) and 77% accuracy on a testing sample of about 20,000 words.

- Predicting the secondary structure of Globular Proteins [83]: the network could learn from existing protein structures how to predict the secondary structure of local se-

[7] The detailed algebra will not be given here, but we should mention that the name *back-propagation* comes from the fact that energy derivatives are propagated back, in the conceptual sense, from the output layer to the first hidden layer.

The Back-Propagation Algorithm

1. Initialization: Starting from a random setting of $\mathbf{w}$.
2. Evaluate $E(\mathbf{w})$ (eq. 3.4).
3. If $E(\mathbf{w})$ is smaller than a *prespecified* value E_0, STOP and output $\mathbf{w}$.
4. Evaluate $\nabla E(\mathbf{w})$.
5. Change the weights according to eq. 3.5.
6. Go to step 2.

Figure 3.7: A sketch of the back-propagation algorithm.

quences of amino acids. The average success rate on a testing sample was 64.3%, higher than those of previous methods.

It might appear that back-propagation has made a breakthrough in learning for neural network as it seems to have solved our famous credit assignment problem. Many people at the beginning thought so, and many still do. But in reality, what back-propagation is telling us is to forget about the credit assignment problem and all that stuff, and things will take care of themselves (remember the quote at the beginning of this chapter!). In this sense, it is at best a general-purpose method that cannot be expected to find any intelligent way to solve a particular problem.

More specifically, gradient descent methods are known to have problems in finding solutions to minimization problems and back-propagation is no exception to that. Its main problems are:

- It may not converge at all.
- It may converge to the wrong answer.
- It may converge to the right answer, but it may take a life time.

In general there is no a priori knowledge about the *shape* of the energy function $E(\mathbf{w})$ (eq. 3.4). Typically, it is characterized by a large dynamic range of gradients and a large

number of local minima. There is a significant chance of becoming trapped in bad local minima and the algorithm is very slow to converge in areas of relatively shallow gradient. Moreover, the algorithm is very sensitive to 1) the initial setting of the weights, 2) the learning rate ϵ, and 3) the stopping conditions. Even the slightest variation can make the difference between good and bad solution. The only route to getting the best results is through repeated experimentation with varying experimental conditions [8].

In practice, the back-propagation often runs very slowly and gets trapped in local minima [100]. Worse, a recent paper[17] provided examples in which the back-propagation failed to train even single perceptrons.

Although you will find people that swear by it, back-propagation is certainly not the algorithm we have been waiting for. The question now is whether or not we have been waiting in vain. In other words, does such an algorithm really exist? Maybe Minsky and Papert were right after all (a terrifying thought indeed!). This is what we will try to answer in the next section.

3.6 Negative Results about Training Neural Networks

In the previous section, we raised the natural question of whether there are *intrinsic* computational difficulties associated with training neural networks, or whether good training algorithms might exist. Before answering this question, we restate the training problem for neural networks as a decision problem: [9]

Given a *fixed* network architecture and given a training sample, is there a setting of the weights such that the network classifies correctly every example in the training sample?

Note that the network is not required to generalize to new examples but simply to *memorize* or *load* the training examples. This is referred to as the "loading problem". Obviously, if a network can not even memorize the training set, we do not expect it to generalize to unseen examples. So before worrying about the generalization, we should first solve the loading problem.

[8] Another problem that we will talk about later is how to choose an appropriate network architecture.

[9] Note that the search problem (finding the weights) is at least equally hard.

A common method for demonstrating a problem to be intrinsically hard is to show it to be NP-complete[31]. The most famous NP-complete problem is the traveling salesman. It is widely believed that no polynomial time algorithm exists for the class of NP-complete problems. In other words, an algorithm that solves an NP-complete problem will run in time exponential in the *size* of the problem, in the worst case. So, in practice, only small instances of the problem can be solved efficiently: for the traveling salesman, only instances with small numbers of cities can be solved. For details on the theory of NP-completeness, see[31].

Judd [55] proved the loading problem for neural networks to be NP-complete. He proved also that the problem remains NP-complete even if the network is required to produce the correct output for only two-thirds of the training examples. This implies that even "approximate training" is intrinsically difficult, in the worst case. However, Judd's proofs were based on some complicated architectures and do not apply to the typical networks with say one output. This has been addressed by Blum and Rivest [14] who showed the NP-completeness of loading a very simple architecture: a two-layer network with n inputs, two hidden neurons, and one output neuron. More NP-completeness results for loading neural networks can be found in [65]. Thus, the NP-completeness is the profound reason for the failure to produce an efficient training algorithm for neural networks.

These negative results may appear somewhat surprising since *memorization problems* are generally very easy to solve, by constructing look-up tables for example. But the loading problem as stated above is a "constrained memorization problem": memorize a set of examples in a *fixed* architecture. In fact, if we relax the *fixed* architecture constraint, any given memorization task can be trivially solved by a Grand-Mother type network [10]. Unfortunately, this is equivalent to a look-up table and as such, has no generalization ability whatsoever.

Nevertheless, if we avoid the trivial Grand-Mother solution and alike, the idea of *constructing* neural networks to fit a given set of examples is quite interesting.

[10]For a training set of m examples, one can construct a two-layer network with m hidden neurons in such a way that the j^{th} hidden neuron outputs $+1$ *only* for the j^{th} training example. The weights of the output neuron can then be chosen to get the right classification for every training example. See [87].

3.7 Constructive Learning Algorithms for Feedforward Neural Networks

Given two trained networks that both memorize exactly a training sample, there are strong intuitive reasons to expect the simpler network (fewest hidden units and weights) to generalize better to new, unseen examples. This principle, called *Occam's Razor*, is confirmed by both experimental [25] and theoretical [13, 16] studies. So, it is important to identify the *appropriate* network to solve a given problem. The appropriate network is the one that can memorize the training sample and yields the best generalization: a too-large network will memorize the training sample but will generalize very poorly, and a too-small network will not be able even to memorize the training sample [11].

A serious drawback of the back-propagation algorithm is that it gives no hint whatsoever about how many and how big layers should be to learn a given task. A naive approach to this problem would be to start with a large network and hope that the back-propagation is smart enough to eliminate the unnecessary weights/neurons. But back-propagation is a simple minimization procedure that uses all the weight space available to it. A less naive approach is to modify the energy function E (eq. 3.4) to include a term representing the sum of the weights (a weight decay term) or the squared activations of the hidden units [20]. But in general, this only complicates the shape of the energy function and makes it harder to minimize. Thus, the only route to getting the *appropriate* network is through repeated experimentation with different architectures, a priori knowledge of the task, and/or intuition. These methods may work for small size problems but there is no hope they do for large, complicated tasks.

To address this problem and to get out of the NP-completeness trap, an alternative approach to learning in neural network was proposed by Ruján and Marchand [87], and Mézard and Nadal [73], among others. The main idea of this pioneer approach is that, rather than fixing (guessing) the network in advance, the appropriate network is *constructed*

[11] The same happens in polynomial curve fitting. If the degree of the polynomial is too small, we may not be able to fit the data (underfitting). If the degree of the polynomial is too large, we may be able to fit the data but the extrapolation will be very poor (overfitting).

by the learning algorithm. In other words, the learning algorithm must be able to infer from the training sample both the appropriate architecture and the appropriate weight values.

The learning algorithm given by Ruján and Marchand [87] builds a two-layer network from a given set of training examples. Their idea is based on the geometric point of view described in section 3.5.1. They realized that one obtains a faithful and linearly separable internal representation if the hyperplanes associated with the hidden units do not intersect inside the input hypercube [12]. They gave a greedy algorithm that constructs the hidden units (hyperplanes) one by one, in such a way that this condition is verified. Unfortunately, the condition imposed on hyperplanes is too restrictive and the greedy algorithm is computationally expensive. A substantial improvement of this approach will be given in the next chapter.

3.8 About this thesis

In this thesis, we will look at different ways to overcome the difficulty of training feedforward neural networks. The first approach we will take is to try to come up with efficient constructive learning algorithms for different types of neural architectures. The second approach will be to restrict the distribution generating the training examples, the connectivity of the network, and/or the values of the weights. This will enable us to identify new classes of neural networks that are efficiently learnable, either in a worst case or an average case sense. The arguments given for the efficient learnability of those new classes will range from extensive simulation results to rigorous mathematical proofs.

[12] This is a sufficient but not a necessary condition.

Part II

Constructive Algorithms for Neural Networks

Chapter 4

A Convergence Theorem for Sequential Learning in Two Layer Networks

4.1 Introduction

Usually in real applications the complexity of a classification problem is not known *a priori*. So identifying the appropriate network to obtain good performance is of fundamental importance. The approach we adopt here is to let the learning algorithm determine the appropriate network during the training process. This approach has the additional advantage of avoiding the NP-completeness trap due to the fixed architecture constraint. However, if the algorithm is given too much freedom in choosing the architecture, we risk to fall in another trap: trivial solutions (networks) with no generalization abilities whatsoever. So we should somehow direct the learning algorithm toward good solutions (networks).

As a guideline for choosing the appropriate network we adopt *Occam's Razor* principle: choose the *smallest* network (fewest hidden units and weights) that memorizes exactly the training sample. There are strong experimental [25] and theoretical [13, 16] evidences that the simplest network will generalize better to new, unseen examples. Unfortunately, finding the smallest network is an NP-complete problem in the worst case [65]. But this is not a

real setback since a *not-too-large* network will do the job almost as good as the smallest one [16]. So we will settle for a small, but maybe not the smallest, network that memorizes the training examples.

In this chapter, we focus on learning algorithms that construct a two-layer network from a given training sample [1]. The approach is to start with a network with zero hidden units and gradually add hidden units. At the end of the building process, the network must be able to memorize all the training examples. The goal is to achieve this with the minimum possible number of hidden units.

To solve this problem, we propose a class of learning procedures we call "sequential learning algorithms". By sequential learning we mean that hidden units are added to the network, one at a time. Each added unit separates a group of training examples, pertaining to the same class, from the rest of the examples. The process ends when all the remaining examples are all from the same class. We prove that the internal representations obtained by such procedures are linearly separable.

Recently, two different heuristic approaches have been proposed for building neural networks from training examples [73, 87]. The class of learning procedures introduced here differs in several respects from these approaches. The "Tiling strategy" of Mézard and Nadal [73] generally needs several hidden layers before convergence and, in each layer, there exists two kinds of hidden units: a master unit obtained by minimizing an error criteria and ancillary units generated in order to have faithful representations. In our approach, only one type of units is necessary. Furthermore, our convergence theorem is quite different and applies already for the first hidden layer. Compared to Ruján and Marchand's strategy [87], the present approach is a substantial improvement. First, we now go beyond regular partitions [87] by relaxing the condition that hyperplanes should not intersect inside the hypercube (and hence permit a larger set of internal representations). Second, we do not restrict the values of the weights to $\{-1, 0, +1\}$ but allow for any real value. Third, instead of using an exhaustive search procedure among a limited set of solutions, we use iteratively the perceptron learning rule (chapter 3).

[1] It is well known that, given a sufficient number of hidden neurons, such networks are able in principle to memorize any set of training examples [74].

4.2 Definitions

Let the input space $\mathcal{X}^n$ be the set $\{-1,+1\}^n$. We are interested in learning a boolean function f that maps from $\mathcal{X}^n$ into $\{-1,+1\}$. An input vector $\mathbf{x} \in \mathcal{X}^n$ is said to be classified positive (or a positive example) if $f(\mathbf{x}) = +1$, and is said to be classified negative (or a negative example) if $f(\mathbf{x}) = -1$. We will sometimes refer to $f(\mathbf{x})$ as the target classification of input vector $\mathbf{x}$. To simplify the notation, let $\sigma \equiv f(\mathbf{x})$.

We are given a set of training examples $< \mathbf{x}^\mu, \sigma^\mu >_{\mu=1,\ldots,m}$. We want to find a two-layer network of n input units [2], h hidden units (each connected to all n input units), and one output unit (connected to each hidden unit but not to the input units), that reproduces the given training examples. For an input vector $\mathbf{x} \in \mathcal{X}^n$, we take x_j to be the state (or setting) of the j^{th} input unit of the network.

We denote by $\mathbf{w}_k = (w_{k,1}, \ldots, w_{k,j}, \ldots, w_{k,n})$ the weight vector connecting the k^{th} hidden unit to the input units and by $w_{k,0}$ the bias of this hidden unit. We denote by $\mathbf{u} = (u_1, \ldots, u_1, \ldots, u_h)$ the weight vector connecting the output unit to the hidden units and by u_0 the bias of the output unit.

When the input vector $\mathbf{x}^\mu$ is presented to the network, the activation state of the k^{th} hidden unit, denoted S_k^μ, is given by the threshold rule:

$$S_k^\mu = \text{sgn}(\sum_{j=1}^{n} w_{k,j} x_j^\mu + w_{k,0}). \tag{4.1}$$

where $\text{sgn}(z) = +1$ if $z \geq 0$ and -1 otherwise.

If there are h units in the hidden layer, these units will transform (map) the n-dimensional input vector $\mathbf{x}^\mu$ into a h-dimensional vector with binary components. This vector, denoted $\mathbf{S}^\mu$, is called the *internal representation* of $\mathbf{x}^\mu$.

Likewise, the state of the output unit, denoted O^μ, is given by:

$$O^\mu = \text{sgn}(\sum_{k=1}^{h} u_k S_k^\mu + u_0). \tag{4.2}$$

The network reproduces (or is consistent with) the given training sample if

$$O^\mu = \sigma^\mu \quad \text{for } \mu = 1, \ldots, m$$

[2]The number of input units is obviously determined by the dimension of the input space.

4.3 The Sequential Learning Procedure

The sequential procedure to build the layer of hidden units works as follows:

Let $s_1 = \pm 1$. Find any setting of $\mathbf{w}_1$ such that

- $S_1^\mu = \sigma^\mu$ for *all* examples of target $\sigma^\mu \neq s_1$ *and*
- $S_1^\mu = \sigma^\mu$ for a *certain* number $m_1 \geq 1$ of examples of target $\sigma^\mu = s_1$.

Note that this is always possible because of the existence of the "Grand-Mother neuron" solution [25, 87]. This gives us the first hidden unit. We shall say that this hidden unit "excludes" a cluster of m_1 examples from the training set. Note that the target of all examples in this cluster is s_1.

Next, remove these m_1 examples from the training set and call the remaining set of $m - m_1$ examples the *working set*. Now, let $s_2 = \pm 1$ and find a setting of $\mathbf{w}_2$ such that

- $S_2^\mu = \sigma^\mu$ for all examples (in the working set) of target $\sigma^\mu \neq s_2$ *and*
- $S_2^\mu = \sigma^\mu$ for a *certain* number $m_2 \geq 1$ of examples (in the working set) of target $\sigma^\mu = s_2$.

This gives us the second hidden unit. We shall say that this hidden unit "excludes" a cluster of m_2 examples from the working set. Note that the target of all examples in this cluster is s_2. Remove these m_2 examples from the working set and repeat the procedure until only examples with the same target remain in the working set.

In this way, the input space is partitioned into a number of regions, each of them containing at least one example $\mathbf{x}^\mu$. Moreover, all the examples belonging to the same region are of the same target and are identified by a single internal representation vector $\mathbf{S}$ (see Fig. 4.1a). The number of hidden units h , the cluster sizes $m_1, \ldots, m_h, m_{h+1}$, and the cluster targets $s_1, \ldots, s_h, s_{h+1}$, are not fixed in advance. Their determination is the task of the particular algorithm implementing the sequential learning. The $(h+1)^{\text{th}}$ cluster, with a target $s_{h+1} = -s_h$, is made of the examples remaining at the end of the process. Note that the number of different internal representations $N_{ir} \geq h+1$ (see Fig. 4.1a), thus generalizing the result obtained for regular partitions [87], where $N_{ir} = h + 1$.

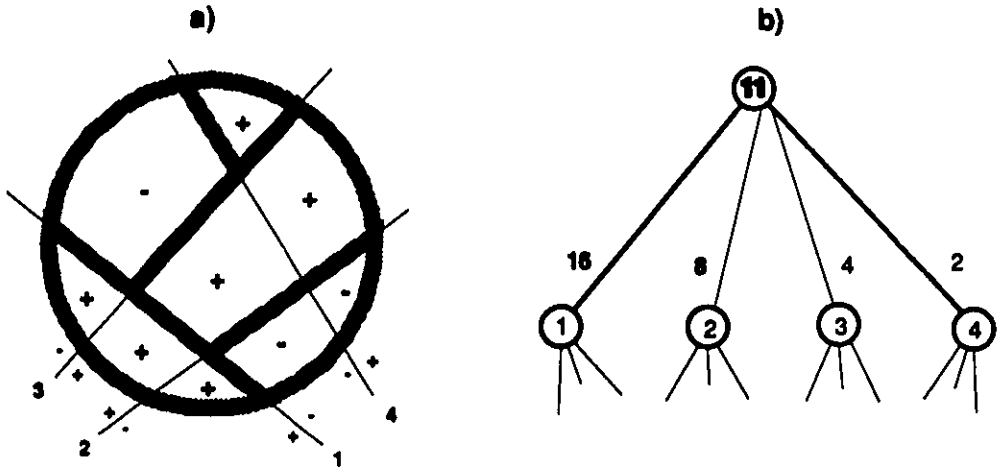

Figure 4.1: An example of a) the partition of the input space realized by the sequential procedure and b) the values of the weights going to the output unit. Ir a), the big circle represents the input space and the hidden units are represented by straight lines with their outputs indicated by a + or − on each side. The hidden units divide the input space into regions. All examples in the same region have the same target which is indicated by + or −. Here we have nine different internal representations but only five "excluded" clusters (indicated by shaded regions). The corresponding hidden units with the values of their outgoing connection are illustrated in b). The bias w_0 is indicated inside the output circle.

The original training sample has been now mapped into a set of internal representations. We prove the following theorem:

Theorem 4.1 *The set of internal representations obtained by any procedure satisfying the conditions of the sequential learning is linearly separable.*

Proof:

The proof of this theorem relies on the fact that since we have build *sequentially* h hidden units, each of them "excludes" from the working space a cluster of examples of the same target. Hence, upon the presentation of any example $\mathbf{x}^\mu$ from the k^{th} cluster, the internal

representation vector on the hidden layer will have the form:

$$S^\mu = (-s_1, -s_2, \ldots, -s_{k-1}, s_k, *, \ldots, *). \tag{4.3}$$

where $*$ indicates either $+1$ or -1.

This suggests that the correct classification of this example, s_k, can be obtained if we choose the weights of the output unit such that $u_k > \sum_{j=k+1}^{h} |u_j|$ and the bias u_0 such as to cancel out the activity coming from the hidden units $1, \ldots, k-1$. The following solution satisfies both conditions:

$$\begin{aligned} u_j &= 2^{h-j+1} \quad \text{for } j = 1, \ldots, h \\ u_0 &= \sum_{i=1}^{h} s_i u_i - s_h \end{aligned} \tag{4.4}$$

Indeed, one can easily verify (using eqs. 4.2, 4.3, and 4.4) that $O^\mu = s_k$ upon presentation of any example $\mathbf{x}^\mu$ from k^{th} cluster. This completes the proof.□

A close look at the solution given in eq. 4.4 reveals that, as h increases, the weights of the output unit grow exponentially large. This may cause some problems in practice, for example in hardware implementation. However, this solution is not the only one possible. In fact, we have found that running the perceptron algorithm on the obtained set of internal representations will not generally yield this solution, where the weights increase exponentially. In chapter 7 we present a way to eliminate completely the problem of the exponential increase in the weights.

4.4 A Particular Implementation: the Maximum Cluster Size Strategy

According to the sequential learning procedure, the weight vector going to a given hidden unit is chosen such as to exclude, from the working set, a certain number of examples with the same target. Since our goal is to construct as few hidden units as possible, it seems natural to choose at each step the weight vector excluding the maximal number of examples with the same target. Unfortunately, finding such a weight vector is extremely difficult (see chapter 7). So we will settle for the following greedy procedure.

Assume that when constructing hidden unit k, the working space has N^+ positive examples and N^- negative examples. Our implementation tries first to find two weight vectors: one that excludes a maximum number of positive examples, say M^+, and another one that excludes a maximum number of negative examples, say M^-. Among these two weight vectors, we choose as $\mathbf{w}_k$ the one that corresponds to the largest fraction $M^{\pm}/N^{\pm}$.

To find, for each hidden unit k, the weight vector that maximizes M^+, first we pick a positive example $\mathbf{x}^\mu$ and choose the weights such that the output of this unit is $+1$ for this example and -1 for all the others [3]. Next, a misclassified [4] positive example, say $\mathbf{x}^{\mu'}$, is chosen and the perceptron learning rule is used to change the weights as to classify correctly *also* this example. If we succeed, we keep both $\mathbf{x}^\mu$ and $\mathbf{x}^{\mu'}$ and try to add another positive example. If we fail to classify correctly $\mathbf{x}^{\mu'}$, we simply drop it and try with another positive example.

After trying to include all the misclassified positive examples, we end up with a weight vector that excludes a certain number v_1 of positive examples. But unless we have been working on a linearly separable set, there are still some misclassified positive examples left. So we store the weight vector found, we remove the properly classified positive examples from the working set, and repeat the same procedure starting from one of the misclassified positive examples. This will yield another weight vector. To compute what number v_2 of positive examples this new weight vector does exclude, we bring back, temporarily, the v_1 positive examples into the working set. Depending on the values of v_2 and v_1, we keep only the best of the previous weight vectors. We continue, keeping always the best weight vector found, until $\sum_i v_i = N^+$. We proceed similarly to find the weight vector that maximizes M^-.

The major computational load of this procedure is the use of the perceptron learning rule to decide whether or not a given set of N examples (in which only one example is misclassified) is linearly separable. We have found however that, in this case, the perceptron requires few iterations to find a solution when the set is linearly separable. For example, the maximum number of iterations found in 625 tests on $m = 1000$ for $n = 20$ was 105, which

[3] For example, $w_{k,j} = x_j^\mu$ for $j = 1, \ldots, n$ and $w_{k,0} = 1 - n$.

[4] Hidden unit k is said to misclassify an input vector $\mathbf{x}^\mu$ if $S_k^\mu \neq \sigma^\mu$.

is much smaller than m. In addition, one can take advantage of the fact that if the set is not linearly separable, the sequence of weights given by the Perceptron rule will eventually cycle [74].

4.5 Simulation Results

New learning algorithms are usually first tested on standard problems that have been previously used as *bench-marks* in the neural network literature. The effectiveness of the new algorithm can then be measured in terms of how close its solutions are to the known optimal ones.

In this section, we describe the experimental tests performed on some bench-mark problems. We investigate both the memorization and the generalization abilities of the algorithm.

4.5.1 Memorization

The learning algorithm, as described in the previous section, was first tested on completely specified functions, *i.e.* the training samples consisted of all the 2^n possible examples.

Random Boolean Functions

A boolean function f is said to be a random one if the input vectors are labeled positive or negative at random: $\text{Prob}(\sigma^\mu = +1) = \text{Prob}(\sigma^\mu = -1) = 1/2$. To test the robustness of the algorithm, we have generated 100 random Boolean functions on n input variables. As expected, the algorithm was always able to find a network with one hidden layer. For $n = 6$ and 8, the average number of hidden units found was $< h >= 7.28 \pm 0.82$ and 18.3 ± 1.2, respectively. This is significantly lower than the results reported in [73, 87].

Parity Function

The parity function is defined as follows: $f(\mathbf{x}) = +1$ if the number of input variables set to $+1$ is odd, and $f(\mathbf{x}) = -1$ otherwise [89]. For $n = 2, 3, \ldots, 12$, the algorithm was always able to find the optimal solution: one layer of $h = n$ hidden units with weights similar to those found in [89, 73, 87].

Mirror Symmetry Function

The mirror symmetry function is defined as follows: $f(\mathbf{x}) = +1$ if $\mathbf{x}$ is symmetric about its center and $f(\mathbf{x}) = -1$ otherwise [89]. For $n = 2, 4, \ldots, 12$, we have always found the optimal solution of only two hidden units with weights similar to those found in [89].

We note here that it is impossible to obtain this optimal solution using the Tiling strategy [73]. In building the first master unit the latter strategy may eventually find the weight vector that minimizes the classification error. But since there are only $2^{n/2}$ positive examples, the weight vector that minimizes the error will classify properly all the negative examples and *only* one positive example (a Grand-Mother type solution). This shows that a least error strategy, as adopted by the Tiling algorithm, is not always appropriate.

4.5.2 Generalization

Here, we investigate the generalization abilities of networks built from partially specified functions, *i.e.* from training samples that do not contain all the 2^n possible examples.

In each test, the learning algorithm is run on a training sample of m randomly chosen examples. The resulting network is then tested on the $2^n - m$ unseen examples. A classification error occurs whenever the network's output is different from the example's target classification. The *generalization* is simply defined as the fraction of examples, in the test set, classified correctly by the network.

We are interested in how the generalization and the size of the network (number of hidden units) vary with the fraction of examples memorized ($m/2^n$). As it turned out, these two quantities are closely related.

Parity Function

Fig. 4.2 is a plot of (a) the average number of hidden units in the resulting network and (b) the average generalization, both as a function of the fraction of examples memorized, for $n = 10$. Each point on the graph is the average over 20 tests.

We note that as the fraction of examples memorized increases, the average number of hidden units first increases to a value above the optimal one, then starts decreasing (fig. 4.2a).

This phenomena is quite ir resting as it seems that for small values of m, the algorithm is simply memorizing the training examples by allocating more resources (hidden units) whenever m increases. But at some value of m (corresponding roughly to 0.5 on fig. 4.2a), the algorithm starts taking advantage of the correlations that exist between the training examples. The algorithm realizes that large *groups* of examples can be assigned to the same hidden unit. This "change of mind" is quite abrupt.

Looking at fig. 4.2b, we can see that for small values of m the generalization ability improves slowly. But as we reach the value of m mentioned above, the generalization starts improving rapidly.

To see whether these trends are intrinsic to this algorithm or simply the results of the particular problem (parity), we look at another problem: mirror symmetry.

Mirror Symmetry Function

Fig. 4.3 is a plot of the network's average size and the generalization as a function of the fraction of examples memorized, for $n = 10$. Again, each point on the graph is the average over 20 tests. Since the number of positive examples is substantially lower than the number negati . . amples, we need to consider the generalization for the two type of examples *separately*. In fact, the fraction of correctly classified negative examples is always exceeding 90%. It is however much more difficult to classify correctly the positive examples [5].

The same trends are observed again. Below a certain value of m (corresponding roughly to 0.2 on fig. 4.3), the number of hidden units increases and the generalization improves only slowly. Above this value of m, the number of hidden units starts decreasing and the generalization improves very rapidly. Again this change of mind is quite abrupt.

Thus, the trends are the same for the parity and the mirror functions. This raises the following interesting question: is it possible to tell whether the algorithm is learning or memorizing by observing *only* the variations of the size of the network? We conjecture that the answer is yes. The next example is a strong indication that it is so.

[5] Any dummy network which classifies every example as negative will do extremely well if we do not consider the generalization for the two type of examples separately.

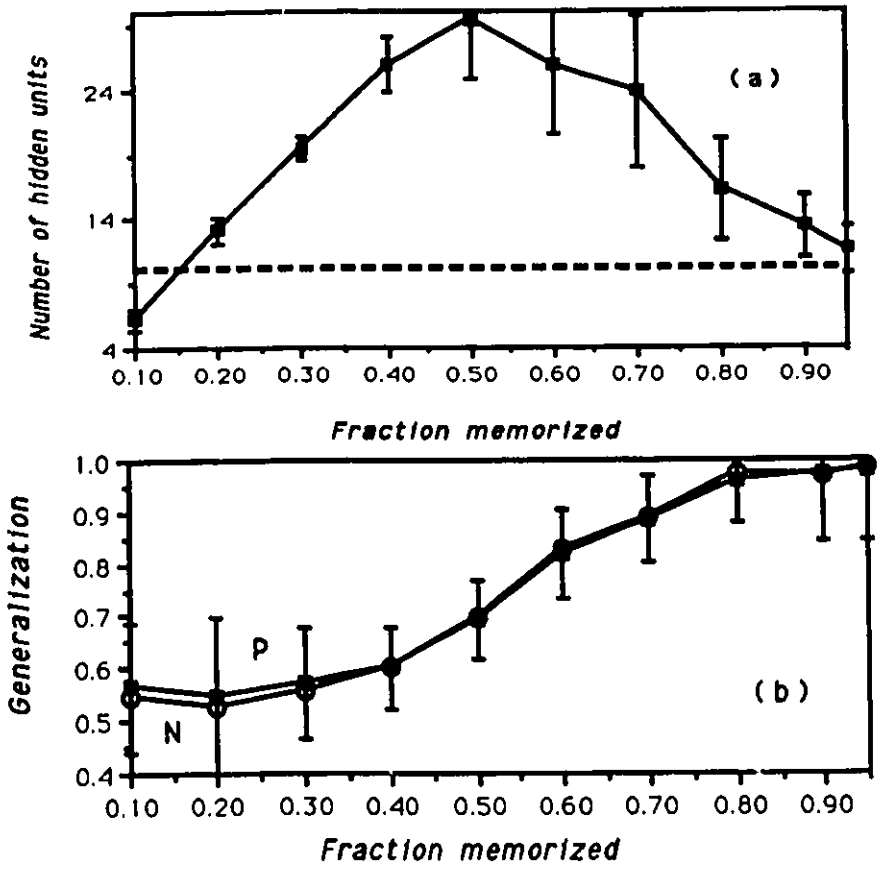

Figure 4.2: Learning the Parity Function for $n = 10$. (a) The average number of hidden units found by the algorithm as a function of the number of examples memorized. The dashed horizontal line represents the number of hidden units in the optimal solution. (b) The average generalization for the negative (N) and the positive (P) examples as a function of the number of examples memorized. The error bars, shown only for one curve in (b) for clarity, denote the standard deviations.

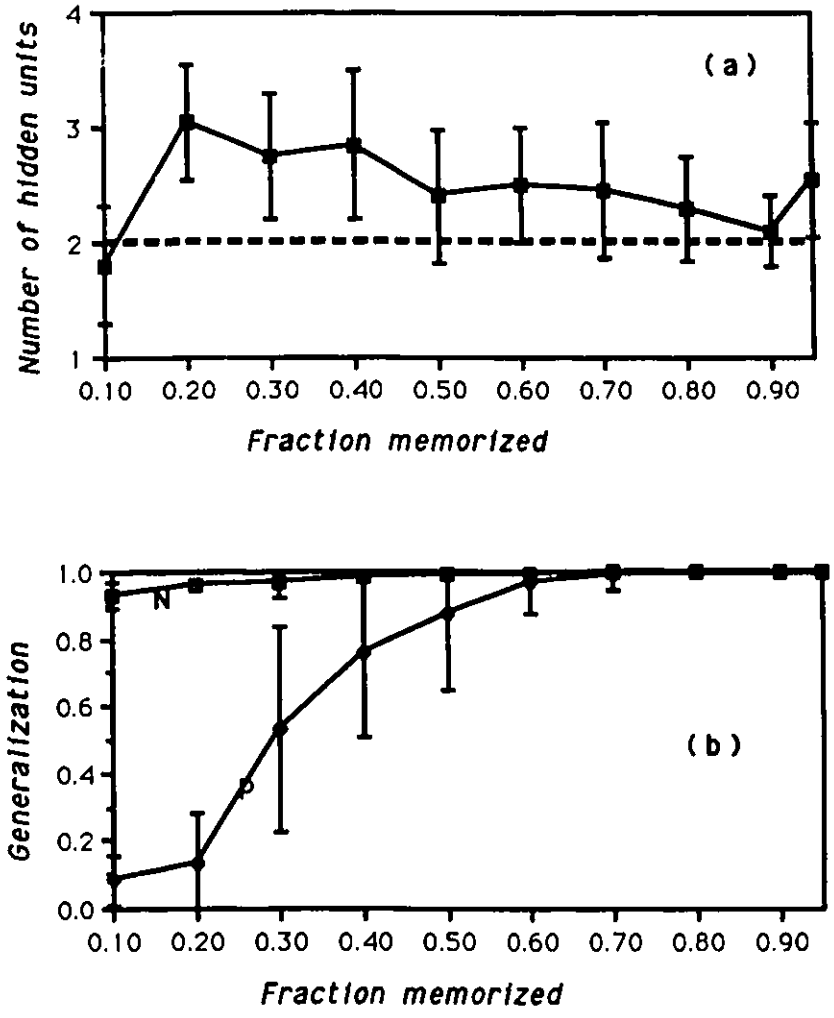

Figure 4.3: Learning the Mirror Symmetry Function for $n = 10$. (a) The average number of hidden units found by the algorithm as a function of the number of examples memorized. The dashed horizontal line represents the number of hidden units in the optimal solution. (b) The average generalization for the negative (**N**) and the positive (**P**) examples as a function of the number of examples memorized. The error bars, shown only for one curve in (b) for clarity, denote the standard deviations.

Cyclic Permutation Function

The cyclic permutation function is defined as follows: $f(\mathbf{x}) = +1$ if the second half of $\mathbf{x}$ is a cyclic permutation of the first half, and $f(\mathbf{x}) = -1$ otherwise. The optimal solution for this function is a two-layer network with n hidden units. However, due to the particular symmetry of this problem, the internal representations generated in the optimal solution are *totally* different from the kind of internal representations used by our algorithm. So we do not expect our algorithm to be able to learn this function, although it can always memorize any set of examples classified according to this function.

Fig. 4.4 is a plot of the network's average size and the generalization as a function of the fraction of examples memorized, for $n = 10$. Each point on the graph is the average over 20 tests. Again, since the number of positive examples is substantially lower than the number negative examples, we need to consider the generalization for the two types of examples *separately*. As with the mirror symmetry problem, the fraction of correctly classified negative examples is always exceeding 90%. It is however much more difficult to classify correctly the positive examples

Here, the trends change dramatically. First, the number of hidden units increases monotonously, almost linearly, with the fraction of examples memorized. Second, the network exhibits a very poor generalization ability [6]. The algorithm is simply memorizing the training examples.

This confirms our conjecture that the variation of the size of the network is a good indication of whether or not the algorithm is actually learning.

4.6 Summary and Discussion

We have presented a class of learning algorithms that construct a two-layer network from the given training sample. Starting from a network with zero hidden units, the algorithm adds units, one by one. Each added hidden unit excludes from the training sample a subset of positive (negative) examples. The building process ends when the network is able to mem-

[6] We should not be fooled by the network's performance on the negative examples.

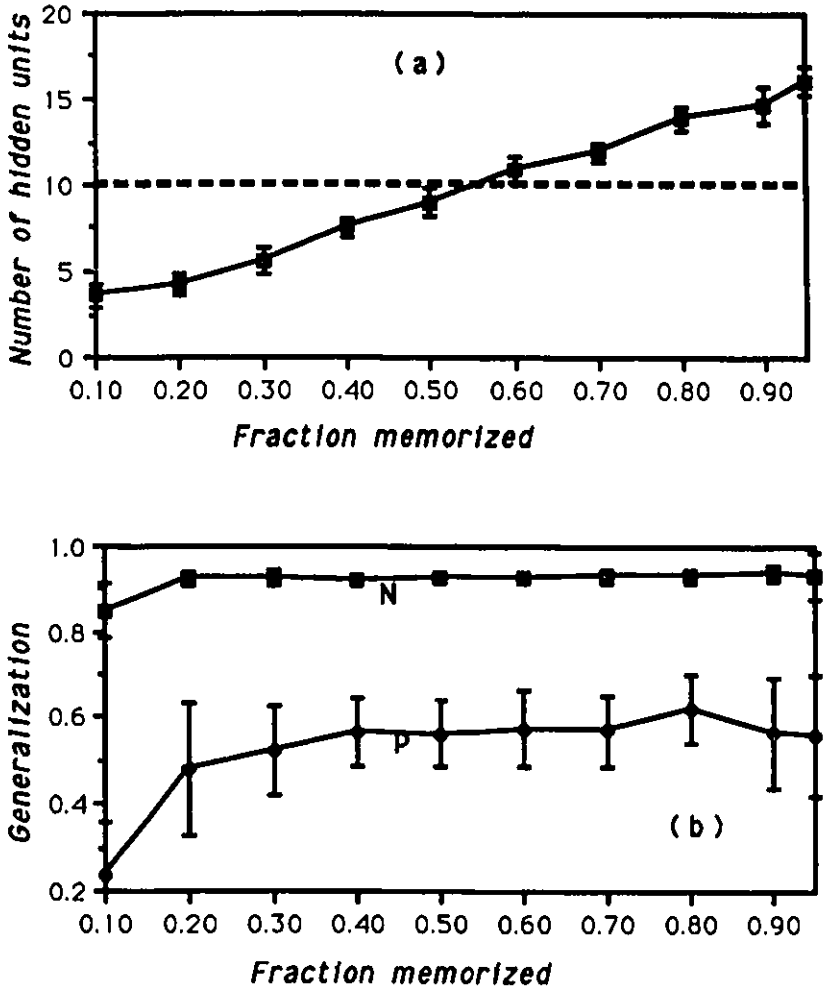

Figure 4.4: Learning the Cyclic Permutation Function for $n = 10$. (a) The average number of hidden units found by the algorithm as a function of the number of examples memorized. The dashed horizontal line represents the number of hidden units in the optimal solution. (b) The average generalization for the negative (N) and the positive (P) examples as a function of the number of examples memorized. The error bars, shown only for one curve in (b) for clarity, denote the standard deviations.

orize all the training examples. We have proved that the internal representations obtained by such procedures are linearly separable.

The *maximum cluster size* strategy is a particular implementation of the sequential learning algorithm that tries to add the fewest possible number of hidden units. We have tested this strategy on some bench-marks problems. From the memorization point of view, the algorithm is always able to reproduce any given training sample. From the generalization point of view, the simulation results exhibit two different trends as a function of the size of the training sample m.

- When the algorithm is able to learn the classification task, the number of hidden units added increases with m up to a certain value m_0, then starts decreasing. This change of mind is quite abrupt and may be interpreted as a transition between the memorization phase and the learning phase. This interpretation is confirmed by the variations in the generalization ability: whereas the generalization improves only slowly for $m \leq m_0$, it starts improving very rapidly above m_0.

- When the algorithm is not able to learn the classification task, the number of hidden units added increases monotonously, almost linearly, with m. In this case, the algorithm exhibits a very poor generalization ability.

We conjecture that these trends are an intrinsic characteristic of this learning algorithm. An interesting question is whether other constructive algorithms exhibit some kind of trends. If so, these trends can be used as good indicators in real applications.

In the next chapter, we present a learning algorithm that constructs tree-like networks from training examples. Fortunately enough, the algorithm does exhibit the same trends observed here.

Chapter 5

A Constructive Algorithm for Neural Network Decision Trees

5.1 Introduction

The sequential learning algorithm presented in the previous chapter builds two-layer networks from training examples. Unfortunately, some classification problems have no efficient (small) two-layer network representation. For these problems the sequential algorithm will build wide two-layer networks that, although they memorize the training examples, have poor generalization abilities. To overcome this difficulty, we are lead naturally to consider other kinds of network architectures.

Since many classification problems are easily represented by multi-branched decision procedures, the design of efficient hierarchical classifiers, more commonly known as decision trees, is currently an important topic in several fields [18, 84] [1]. As the name implies, these classifiers arrive at a decision through a hierarchy of stages. In the simplest decision tree, each non-terminal (or internal) node consists of a test on *one* input variable, of the form "$x_i > b_i$?", with one subtree for each possible outcome of the test. Each terminal node (or leaf) has an assigned class label. The decision process for an input vector $\mathbf{x}$ starts at the bottom node (the root) and proceeds up the tree until a terminal node is reached. The class

[1] A reader who is unfamiliar with decision trees is advised to take a look at Ref. [18] before proceeding with this chapter.

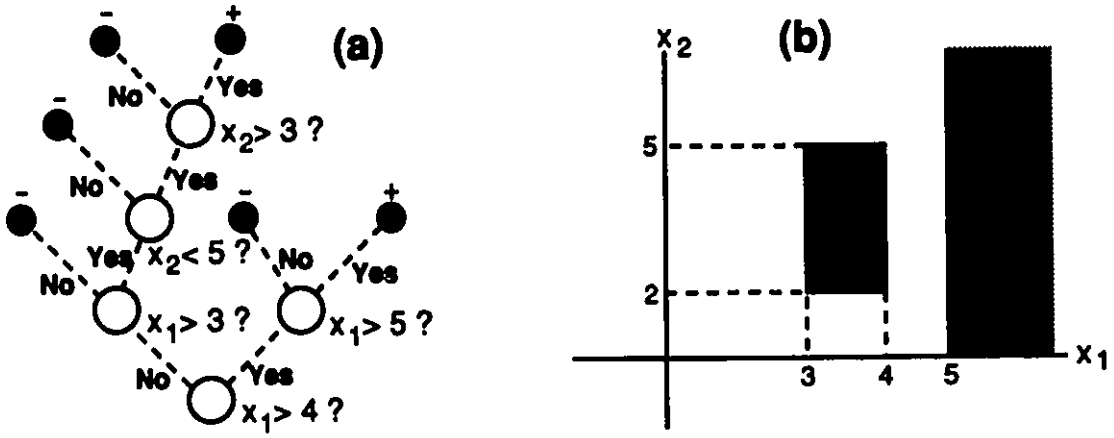

Figure 5.1: (a) An example of of a binary decision tree. The internal nodes are shown as open circles and the leaves as labeled black circles. (b) The partition induced by this binary tree. The shaded region represents the positive examples.

of the input vector x is then the label attached to that terminal node.

A test of the form "$x_i > b_i$?" defines a line $x_i = b_i$ that partitions the input space into two regions. Consequently, a decision tree induces a partition of the input space into decision regions formed by lines that are parallel to the axes of the input space. An example of a decision tree and the corresponding partition induced by it is shown in fig. 5.1.

Training procedures for decision trees do not minimize a global cost function, as back-propagation for example tries to do. Instead, they gradually build a tree by minimizing at each node a *local* cost function that reflects the success of the split of the training sample by the node. The optimal test minimizing this local cost function is selected. The procedures use a top-down, divide-and-conquer strategy to build the tree: select the optimal test, divide the training sample into subsets according to the possible outcomes of the test, and follow

the same procedure with each subset until no subset contains examples from both classes. These single-class (or pure) subsets correspond then to the leaves of the tree, where each leaf is labeled with its corresponding class.

One shortcoming of traditional decision trees is the condition that each test involves only one input variable. This condition does not allow to take advantage of correlations between input variables. Moreover, this condition limits quite severely the type of decision regions that can be implemented. Many attempts have been made to correct this problem, for example by allowing tests that involve two or more input variables [18].

In this chapter, we will extend the concept of decision trees to *neural decision trees*. For that, we define each node to be a perceptron, *i.e.* the test performed by each node is now an inequality that depends on all the input variables. Moreover, to keep up the spirit of *networks*, we depart from the usual organization of decision trees [2]; we add inter-node connections and replace the leaves by a single output node representing the network's decision. The resulting network is a tree-like feedforward net (see below).

As in the previous chapter, we adopt Occam's Razor as the guideline for choosing the appropriate neural tree: choose the *smallest* neural decision tree (fewest nodes and weights) that memorizes exactly the training sample. But, as one may expect by now, finding the smallest tree is an NP-complete problem [51]. So we are led again to consider heuristic algorithms for constructing small, but maybe not the smallest, neural trees from training examples.

In the subsequent sections, we describe a learning algorithm that builds (hopefully small) neural decision trees from training examples. First, a top-down, divide-and-conquer strategy is used to build a neural tree without inter-node weights. Then, the inter-node weights are added in a way such that the internal representations are linearly separable. Finally, an output unit representing the network's decision is suitably connected to the tree.

We should mention here that the idea of constructing neural decision trees has been investigated, independently, by a number of people [29, 97, 102, 93].

[2] A traditional decision tree is a *functional* organization but not a *structural* organization.

5.2 The Learning Algorithm

Let the input space $\mathcal{X}^n$ be the set $\{-1,+1\}^n$ or a subset of $\mathcal{R}^n$. We are interested in learning a boolean function f that maps from $\mathcal{X}^n$ into $\{-1,+1\}$. An input vector $\mathbf{x} \in \mathcal{X}^n$ is said to be classified positive (or a positive example) if $f(\mathbf{x}) = +1$, and is said to be classified negative (or a negative example) if $f(\mathbf{x}) = -1$. To simplify the notation, let $\sigma \equiv f(\mathbf{x})$.

We are given a set S of training examples $< \mathbf{x}^\mu, \sigma^\mu >_{\mu=1,\ldots,m}$. We want to find a neural decision tree of n input units, h internal nodes (each connected to all n input units and to all its ancestors, and one output unit (connected to a suitably chosen subset of internal nodes) that reproduces the given training examples.

We divide the learning process into two phases:

- Using a top-down, divide-and-conquer strategy to build the decision tree without inter-node connections and without the output node.
- Adding the inter-node connections and the output unit to get a tree-like feedforward network.

5.2.1 Learning the Decision Tree

If we forget for a while about inter-node connections, then a neural decision tree (hereafter NDT) is much like a traditional decision tree except that every internal node is a perceptron connected to all input variables (fig. 5.2). The decision process for an input vector $\mathbf{x}$ starts at the root (node 1). At each internal node, the right (respectively left) edge to a child is taken if the node's output is $+1$ (respectively -1). The class of the input vector $\mathbf{x}$ is then the label attached to the terminal node reached.

Let $\mathbf{w}_k$ be the weight vector connecting node (perceptron) k to the input variables and let $w_{k,0}$ be its bias. Geometrically, node k defines a hyperplane H_k which divides the input space into two halfspaces or decision regions:

$$H_k^+ = \{\mathbf{x} : \sum_{i=1}^{n} w_{k,i}x_i + w_{k,0} \geq 0\}$$

$$H_k^- = \{\mathbf{x} : \sum_{i=1}^{n} w_{k,i}x_i + w_{k,0} < 0\}$$

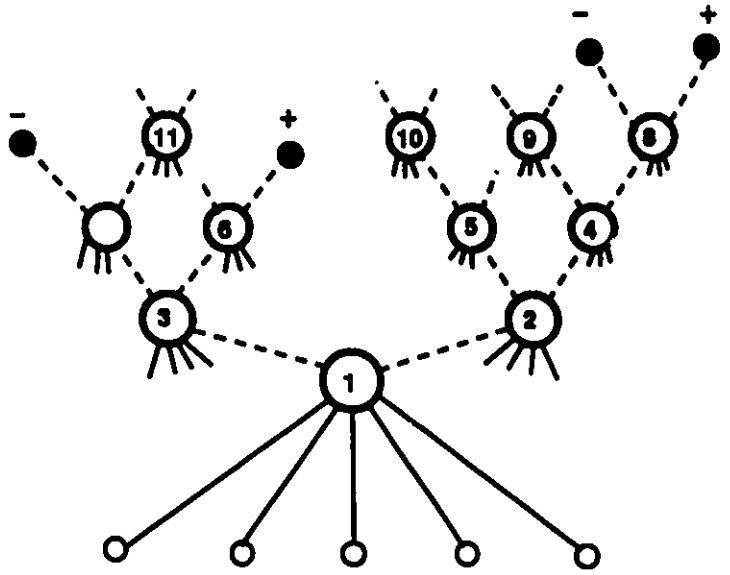

Figure 5.2: The neural network tree considered in the first phase of learning. The small open circles represent the input variables. The large open circles represent the internal (non-terminal) nodes. Each internal node is a perceptron connected to all inputs. The small filled circles represent the leaves (terminal nodes) which are labeled + or −. For clarity, the connections coming from the input units have been interrupted, except for the root.

Our top-down, divide-and conquer strategy works as follows. Initially, we start with a NDT with one node (the root) and a set $S_1 \equiv S$ of training examples which consists of a subset S_1^+ of positive examples and a subset S_1^- of negative examples. Depending on the setting of $(\mathbf{w}_1, w_{1,0})$, this first node may classify incorrectly some positive and/or negative examples. We define the local cost (error) function associated with this first node to be:

$$E_1 = \frac{|H_1^+ \cap S_1^-|}{|S_1^-|} + \frac{|H_1^- \cap S_1^+|}{|S_1^+|} \tag{5.1}$$

where $|\ldots|$ is the cardinality, or the number of elements, in the set. The first term on the right hand side of the above equation is the fraction of incorrectly classified negative

examples. The second term is the relative number of incorrectly classified positive examples. Thus, this cost function reflects the success of the split of the training sample by this first node.

Our approach is to find a setting of $(\mathbf{w}_1, w_{1,0})$ which minimizes the cost function E_1, *i.e.* a setting which yields decision regions as pure as possible. Unfortunately, if S_1^+ and S_1^- are not linearly separable, finding the *global* minimum of this cost function is a very difficult task. To find an approximate solution, we use a variant of the perceptron learning rule called the *pocket* algorithm[30]: we run the perceptron learning rule on the training sample, with a random presentation of the examples, and at each updating we keep in our "pocket" the setting of $(\mathbf{w}_1, w_{1,0})$ which minimizes the cost function E_1. This algorithm generally yields "good" solutions if run for a sufficiently long time [3].

The setting found for $(\mathbf{w}_1, w_{1,0})$ splits the training set S_1 into two subsets:

$$S_2 = H_1^+ \cap S_1 \quad \text{and} \quad S_3 = H_1^- \cap S_1$$

The subset S_2 is assigned to the right subtree of the root node, S_3 is assigned to the left subtree, and the process is applied recursively to each subtree. In other words, the training set for the right subtree consists only of S_2 and the training set for the left subtree consists only of S_3. The subdivision process on one branch of the tree ends when its subset contains only positive (negative) examples. This single-class subset corresponds then to a leaf of the tree, labeled by that class.

More generally, for a node k with training set S_k, the task is to find a setting of $(\mathbf{w}_k, w_{k,0})$ which minimizes the cost function

$$E_k = \frac{|H_k^+ \cap S_k^-|}{|S_k^-|} + \frac{|H_k^- \cap S_k^+|}{|S_k^+|} \tag{5.2}$$

Once the minimization process is done, node k is said to be a (see fig. 5.3):

- **pure node** if $E_k = 0$, *i.e.* the hyperplane H_k divides the set S_k into two *pure* subsets. Obviously, in this case no subtree is needed for this node. Instead, two leaves are attached to this node indicating the corresponding classes.

[3] As we will see in chapter 7, a variant of the maximum cluster size strategy performs better than the pocket algorithm.

- **semi-pure node (type *I*)** if $|H_k^+ \cap S_k^-| \neq 0$ but $|H_k^- \cap S_k^+| = 0$, *i.e.* the hyperplane H_k divides the set S_k into two subsets: a mixed subset $(H_k^+ \cap S_k)$ and a pure subset of negative examples $(H_k^- \cap S_k)$. In this case, only a right subtree is needed for this node. A left leaf, labeled negative, is attached to this node.

- **semi-pure node (type *II*)** if $|H_k^+ \cap S_k^-| = 0$ but $|H_k^- \cap S_k^+| \neq 0$, *i.e.* the hyperplane H_k divides the set S_k into two subsets: a pure subset of positive examples $(H_k^+ \cap S_k)$ and a "mixed" subset $(H_k^- \cap S_k)$. In this case, only a left subtree is needed for this node. A right leaf, labeled positive, is attached to this node.

- **impure node** if none of the above, *i.e.* the hyperplane H_k divides the set S_k into two mixed subsets. In this case, both a left and a right subtrees are needed.

This terminology will be used in the next section to transform the tree to a tree-like feedforward neural network.

5.2.2 From the Decision Tree to the Tree-like Network

Up to this point, we have built a *tree of neurons*, not a *neural network*. To obtain a feedforward network, we make use of the following transformation.

The first transformation is to connect each node to all its ancestors in the tree. Let us denote by $v_{k,j}$ the weight connecting node k to its ancestor j. The output of node k for an example $\mathbf{x}^\mu$, denoted Y_k^μ, is then given by

$$Y_k^\mu = \mathrm{sgn}(\sum_{i=1}^{n} w_{k,i} x_i^\mu + \sum_{j \in A(k)} v_{k,j} Y_j^\mu + w_{k,0}) \tag{5.3}$$

where the summation over j runs over all ancestors of node k.

Suppose that node k has $L-1$ ancestors, which we label $1, 2, \cdots L-1$, starting with the youngest. Then, having found a setting for $(\mathbf{w}_k, w_{k,0})$ by the error-minimization procedure, we choose the $v_{k,i}$'s and renormalize the bias $w_{k,0}$ according to the following rules:

1. $|v_{k,j}| = 1 + \sum_{i=1}^{n} |w_{k,i}| + |w_{k,0}|, \quad \text{for } j = 1 \ldots L-1.$

2. $v_{k,j}$ is positive (excitatory) if unit k is located in the left subtree rooted at node j, and negative (inhibitory) otherwise.

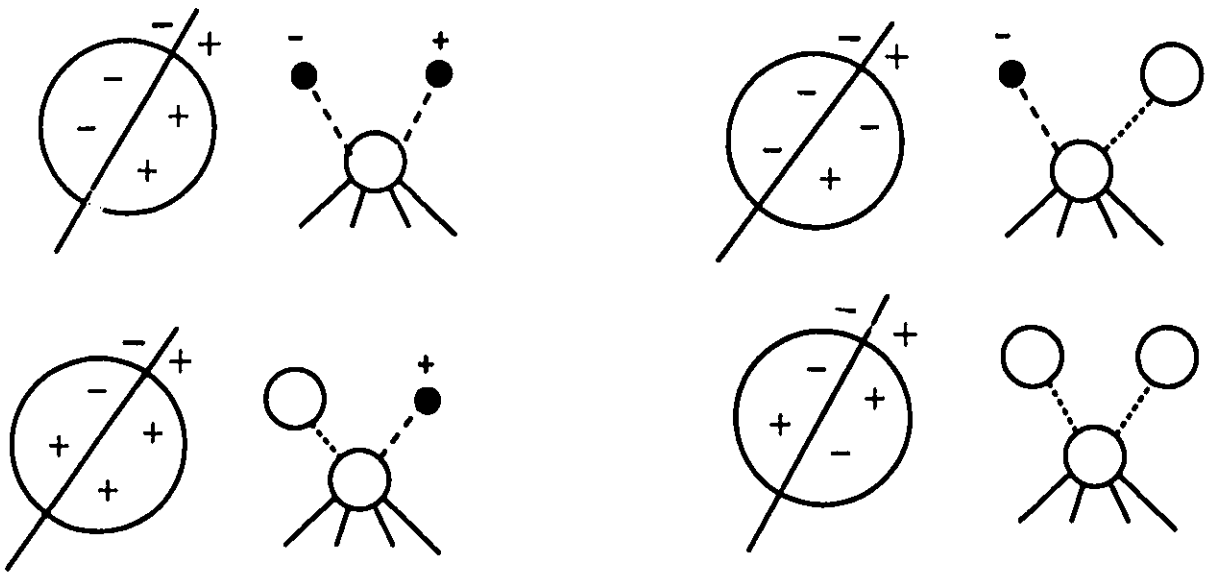

Figure 5.3: The types of nodes as defined in the text along with their decision regions. From left to right; a pure node, a semi-pure (type$I$), a semi-pure node (type II), and a impure node.

3. Renormalize the bias of node k according to: $w_{k,0} \rightarrow w_{k,0} + \sum_{j=1}^{L-1} |v_{k,j}|$.

This choice will ensure that $Y_k^\mu = +1$ whenever $\mathbf{x}^\mu$ does not belong to the training set of node k. This can easily be verified using eq. 5.3 and taking into account the way the tree was built. This property can be achieved by other choices than the one given here.

Now, let B be the set of pure and semi-pure (type I) nodes. The states (outputs) of these nodes for a given input vector $\mathbf{x}^\mu$ form a vector $(Y_k^\mu)_{k\in B}$ we call the *internal representation* of $\mathbf{x}^\mu$. It is easy to check that the internal representation of a positive example will be of the form $(+1,\ldots,+1,\ldots,+1)$ and the internal representation of a negative example will be of the form $(+1,\ldots,+1,-1,+1,\ldots,+1)$. As we will see below, these internal representations are linearly separable.

The second transformation is to get rid of all the leaves and replace them with a single output node (perceptron) connected only to nodes in B. Let us denote by u_k the weight

connecting the output node to the k^{th} node in B and by u_0 its bias. The state of the output node for input vector $\mathbf{x}^\mu$ is given by:

$$O^\mu = \text{sgn}(\sum_{k \in B} u_k V_k^\mu + u_0). \tag{5.4}$$

We now prove that the internal representations defined above are linearly separable. Better, we write down the solution for the weight vector $\mathbf{u}$ and the bias u_0:

$$u_k = 1 \quad k \in B \tag{5.5}$$

$$u_0 = -(N-1) \tag{5.6}$$

where N is the number of nodes in the set B. It is easy to check that, with this solution, $O^\mu = +1$ whenever $\mathbf{x}^\mu$ is a positive example and $O^\mu = -1$ whenever $\mathbf{x}^\mu$ is a negative example. Note that this would not have been the case if we had chosen to include *both* semi-pure (type I and II) nodes in the set B.

The resulting network is a tree-like feedforward net (fig 5.4). Other ways to map a decision tree into a feedforward network have been described subsequently and independently by many researchers [29, 97, 93] .

Finally, the major computational load of this constructive procedure is the use of the pocket algorithm to minimize the cost functions. This is complicated by the fact that we do not know for how long the pocket should be run to obtain a satisfactory solution. For each of the simulations reported below, a maximum number of iterations was set based on some preliminary runs.

5.3 Simulation Results

In this section, we describe the experimental tests performed on some bench-mark problems. We investigate both the memorization and the generalization abilities of the algorithm.

5.3.1 Memorization

As with the simulations of the previous chapter, the learning algorithm was first tested on completely specified functions, *i.e.* the training samples consisted of all the 2^n possible examples.

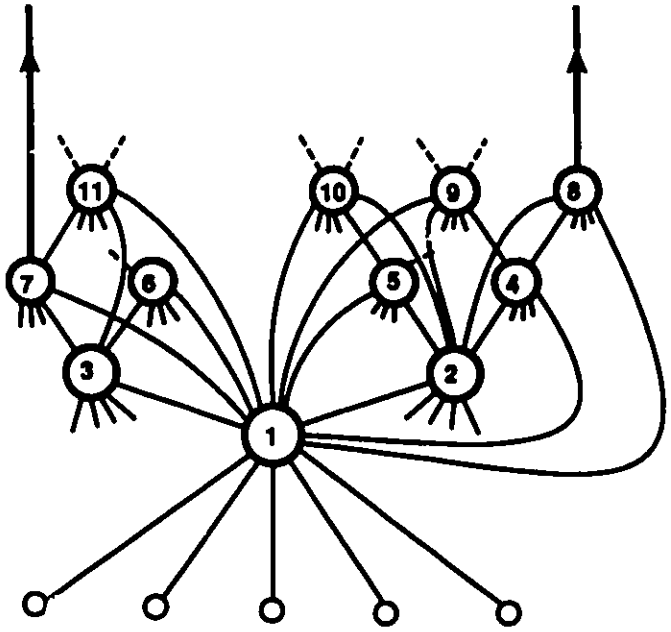

Figure 5.4: The final tree-like network. Each internal node is now connected to the input variables and to all its ancestors. Inter-node connections going to the left are excitatory and those going to the right are inhibitory. As an example: node 2, 3 and 4 are impure nodes, 8 is a pure node, 7 is a semi-pure (type I) node, and node 6 is a semi-pure (type II) node. The big arrows indicate the connections to the output node (not shown here).

Random Boolean Functions

To test the robustness of the algorithm, we have generated 100 random boolean functions on $n = 6$ inputs. As expected, the algorithm was always able to find a neural tree that memorizes all the training sample. The average number of nodes found was 20.5 ± 3.9. This can be compared to $20.27 \pm .44$ reported in [73] using the Tiling strategy, and 7.28 ± 0.82 reported in the previous chapter using the sequential learning strategy.

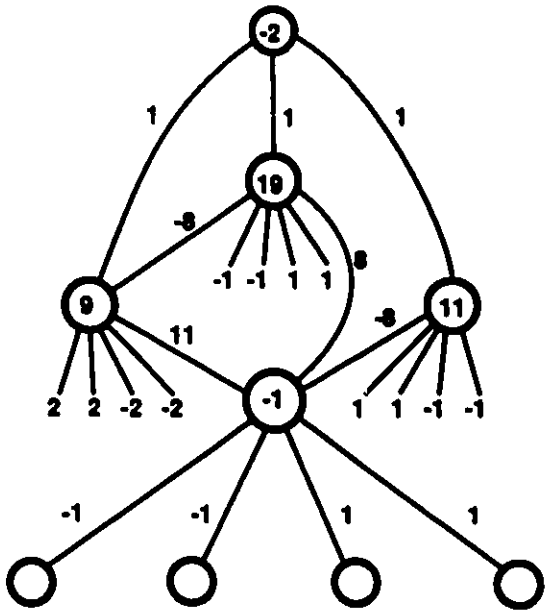

Figure 5.5: A tree-like network obtained by the algorithm for the parity task for $n = 4$. The bias values are indicated inside the circles. For clarity, the connections coming from the input units have been interrupted, except for the root.

Parity Function

For $n = 2, \ldots, 8$, the number of nodes generated by the algorithm was always optimal, *i.e.* equal to n. An example is shown in fig. 5.5 for $n = 4$. Notice that the hyperplanes defined by the weight vectors coming from the inputs are all parallel. This illustrates the fact that the boundaries between the positive and the negative regions are parallel planes [74, 87].

Mirror Symmetry Function

For $n = 2, 4, \ldots, 10$, the number of nodes generated by the algorithm was always optimal, *i.e.* equal to 2. The weight vectors coming from the inputs were similar to those found in [89].

5.3.2 Generalization

Here, we investigate the generalization abilities of networks built from partially specified functions, *i.e.* from training samples that do not contain all the 2^n possible examples.

In each test, the learning algorithm is run on a training sample of m randomly chosen examples. The resulting tree network is then tested on the $2^n - m$ unseen examples. A classification error occurs whenever the network's output is different from the example's target classification.

As in the previous chapter, we are interested in how the generalization and the size of the network (number of nodes) vary with the fraction of examples memorized ($m/2^n$). An important question is whether or not these two quantities exhibit the trends observed in the previous chapter.

Parity Function

Fig. 5.6 is a plot of (a) the average number of nodes in the resulting tree network and (b) the average generalization, both as a function of the fraction of examples memorized, for $n = 10$. Each point on the graph is the average over 20 tests.

It is clear form the fig. 5.6 that both the size of the network and the generalization exhibit the same trends observed when dealing with the sequential learning algorithm. For $0.2 \leq m/2^n \leq 0.4$, the average number of nodes increases up to a value above the optimal one and the generalization improves very slowly. For $m/2^n \geq 0.4$, the average number of nodes starts decreasing and the generalization starts improving rapidly. Again, this "change of mind" is quite abrupt.

Mirror Symmetry Function

The same behavior is observed when learning the mirror symmetry function. Fig. 5.7 reproduces the simulation results for $n = 10$. Each point on the graph is the average over 20 tests. Again, we should not be fooled by the performance of the algorithm on the negative examples.

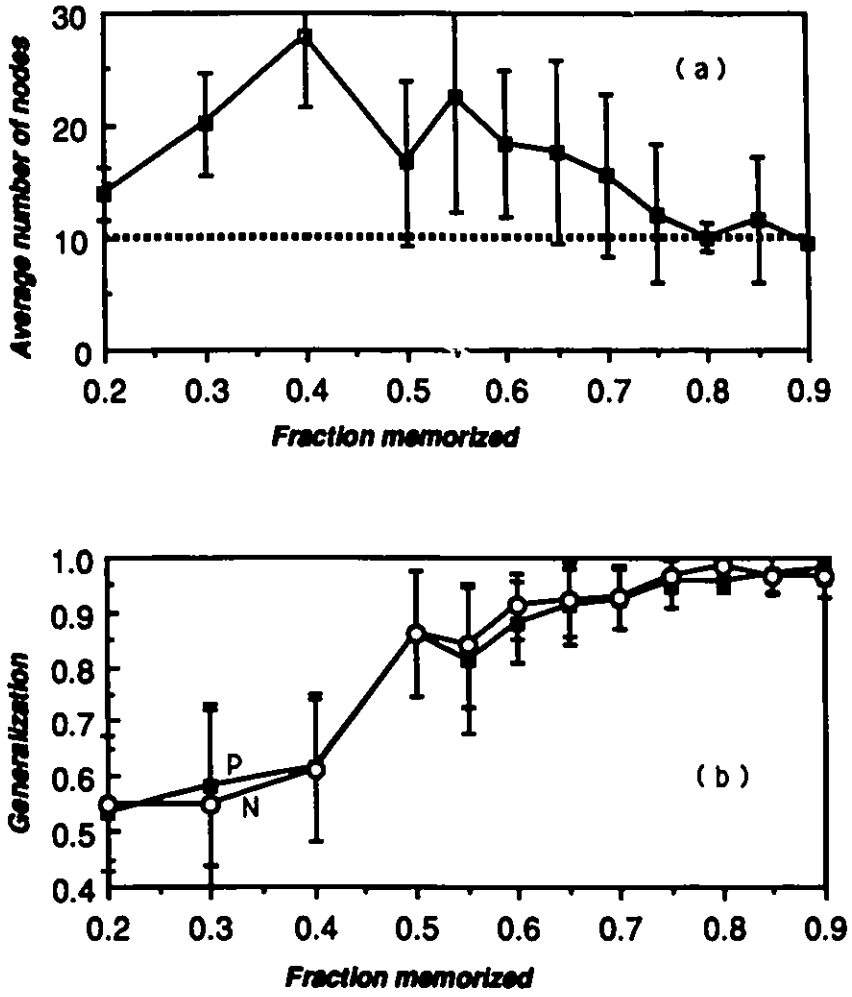

Figure 5.6: Learning the Parity Function for $n = 10$. (a) The average number of nodes found by the algorithm as a function of the number of examples memorized. The dashed horizontal line represents the number of nodes in the optimal solution. (b) The average generalization for the negative (N) and the positive (P) examples as a function of the number of examples memorized. The error bars denote the standard deviations.

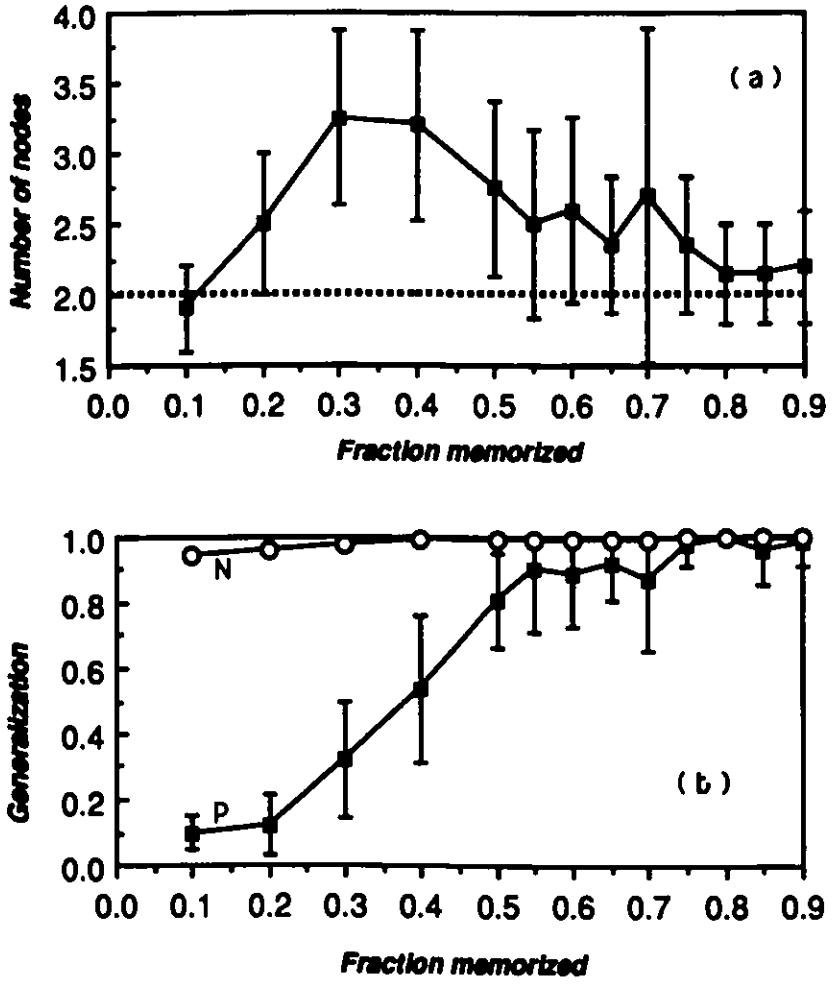

Figure 5.7: Learning the Mirror Symmetry Function for $n = 10$. (a) The average number of nodes found by the algorithm as a function of the number of examples memorized. The dashed horizontal line represents the number of nodes in the optimal solution. (b) The average generalization for the negative (N) and the positive (P) examples as a function of the number of examples memorized. The error bars denote the standard deviations.

The Domain Wall Problem

As an example for situations where the algorithm is not able to learn the target function, we look at the domain wall problem.

The problem is defined as follows [25]. The n input variables are organized in a one dimensional chain. By analogy with a chain of spins, a domain wall is defined as the presence of two adjacent variables having opposite values. The task is to identify those patterns whose number of domain walls is greater or equal to three. More specifically, $f(\mathbf{x}) = +1$ if $\mathbf{x}$ contains 3 or more domain walls and $f(\mathbf{x}) = -1$ otherwise.

Fig. 5.8 reproduces the simulation results for $n = 10$. Each point on the graph is the average over 20 tests. Here, we should not be fooled by the performance of the algorithm on the positive examples!

The trends change dramatically. They look more like the ones observed for the sequential algorithm when learning the cyclic permutation function (section 4.5.2). First, the number of nodes increases monotonically, almost linearly, with the fraction of examples memorized. Second, the network exhibits a very poor generalization ability. All in all, the algorithm is simply memorizing the training examples.

5.4 Conclusion

In summary, the algorithm outlined in this chapter identifies a new class of procedures that will build a tree-like feedforward network able to map any boolean function defined on a given sample data. This algorithm possesses a lot of freedom in the way the inter-node connections may be chosen; we have presented here only one possibility. In addition, learning occurs only at the single neuron level and our algorithm can accommodate any variants in the way to choose the incoming weight vectors $\mathbf{w}_k$. Here we have decided to find the $\mathbf{w}$ that minimizes an error function but it would be interesting to consider criteria like maximizing the mixing entropy [66] or the maximum cluster size strategy (chapter 4).

At the risk of repeating what we have said at the end of the previous chapter, the present algorithm exhibits two different trends:

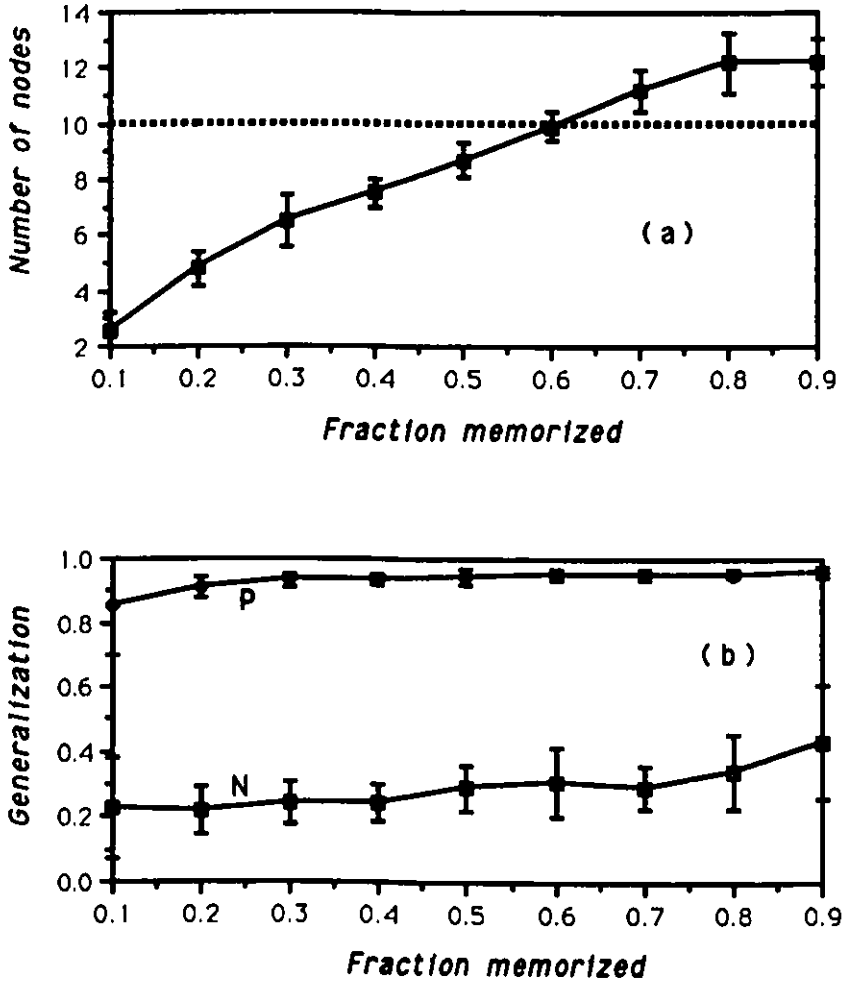

Figure 5.8: Learning the Domain Wall Problem for $n = 10$. (a) The average number of nodes found by the algorithm as a function of the number of examples memorized. The dashed horizontal line represents the number of nodes in the optimal solution. (b) The average generalization for the negative (N) and the positive (P) examples as a function of the number of examples memorized. The error bars denote the standard deviations.

- When the algorithm is able to learn the classification task, the number of nodes added increases with the size of the training set m up to a certain value m_0, then starts decreasing. This change of mind is quite abrupt and may be interpreted as a transition between the memorization phase and the learning phase. This interpretation is confirmed by the variations in the generalization ability: whereas the generalization improves only slowly for $m \leq m_0$, it starts improving very rapidly above m_0.

- When the algorithm is not able to learn the classification task, the number of nodes added increases monotonically, almost linearly, with m. In this case, the algorithm never gets out of the memorization phase and as such, it exhibits a very poor generalization ability.

An analytical investigation of these trends seems beyond reach, but it would be interesting to see whether other *constructive algorithms* exhibits the same trends (or any trend at all).

Part III

Learning Neural Networks within the PAC Model

Chapter 6

Introduction to the "Probably Approximately Correct" (PAC) Model of Learning

6.1 Introduction

In our model of learning, introduced in section 2.3.2, a learning algorithm is given positive and negative examples of some unknown target concept f to be learned and must produce an hypothesis concept h, such that f and h are likely to agree on the classification of new unseen examples. Whereas this model captures the main ideas of learning, it lacks the necessary details for a *quantitative* analysis of the performances of learning algorithms. To be able to do that and prove some interesting results about learning in neural networks, we switch to another, more sophisticated model of learning.

6.2 The PAC Model

In a seminal paper [103], Valiant (1984) introduced an elegant framework for learning concepts from examples, with a view towards probabilistic performance analysis. This framework, now called the "Probably Approximately Correct" (PAC) model, has been subsequently developed by a number of researchers to yield some impressive results and tools for

analysis [16, 15, 45, 46].

In this model, the unknown target concept f is a member of a *prespecified* class $\mathcal{F}$ of concepts. The learning algorithm may request a number of positive and negative examples of the target concept. These examples are generated according to a *fixed* but unknown distribution D on the input space $\mathcal{X}^n$. The learning algorithm then produces its hypothesis about the identity of the concept being learned. If the learning algorithm needs relatively few examples and little running time to produce an hypothesis h that is likely to be *close* to the correct concept, then the class $\mathcal{F}$ is said to be efficiently learnable. Here, for fairness, the *same* probability distribution that is used to generate the training examples is used to judge the error of the hypothesis concept. Thus the algorithm is not penalized if its hypothesis classifies incorrectly examples that occur only rarely according to D.

More specifically, when executed on the target concept $f \in \mathcal{F}$, a learning algorithm is given access to an oracle $EX(f, D)$ that, on each call, draws an input $\mathbf{x}$ randomly and independently according to a fixed but unknown probability distribution D, and returns the labeled example $< \mathbf{x}, f(\mathbf{x}) >$. The error of the hypothesis concept h with respect to f and D is defined to be

$$error(h) = \mathbf{Pr}_{\mathbf{x} \in D}[h(\mathbf{x}) \neq f(\mathbf{x})]$$

where $\mathbf{Pr}_{\mathbf{x} \in D}[\ldots]$ denotes the probability when $\mathbf{x}$ is drawn according to D. The following is the formal definition of PAC learnability.

Definition 6.1 *Let $\mathcal{F}$ be a class of concepts over $\mathcal{X}^n$, and let $\mathcal{H}$ be a class of representations of concepts over $\mathcal{X}^n$. $\mathcal{F}$ is said to be PAC learnable from examples using $\mathcal{H}$ if there exists an algorithm A such that for any target concept $f \in \mathcal{F}$, for any distribution D, and for any $0 < \epsilon < 1$ and for any $0 < \delta < 1$, the following holds:*
Given n, ϵ, δ as inputs, and access to oracle $EX(f, D)$, the algorithm runs in time polynomial in $(n, 1/\epsilon, 1/\delta)$ and produces an hypothesis $h \in \mathcal{H}$ such that $error(h) < \epsilon$ with probability at least $1 - \delta$.
If $\mathcal{H} \equiv \mathcal{F}$, then $\mathcal{F}$ is said to be properly *PAC learnable.*

The parameter ϵ, called the *accuracy*, reflects the fact that the algorithm is not required to identify exactly the target concept, but simply to produce an accurate approximation.

The parameter δ, called the *confidence*, reflects the fact that the algorithm A is not expected to be *always* accurate since it may suffer from bad luck in drawing examples according to D, but it is expected to be accurate very *often*.

The *sample complexity* of the algorithm is the number of training examples needed in the worst case over all concepts in $\mathcal{F}$. The *time complexity* is the worst computation time required to produce an hypothesis over all concepts in $\mathcal{F}$. Note that running in polynomial time implies drawing a polynomial number of training examples.

Valiant called this model *distribution-free* learning, in reference to the requirement that the learning algorithm works for any distribution of examples. The term "probably approximately correct" (PAC) is a reference to the requirement that with high probability $(1 - \delta)$ the algorithm outputs an hypothesis that is approximately correct (error less than ϵ).

To get an intuitive feeling of the difficulty of learning using algorithms that are not polynomial in n, consider an algorithm that is trained by looking at binary pictures of size $n = d \times d$. Assume that the training requires only 1% of all 2^n possible examples, and that a new training example becomes available every 1 msec. Then, the training takes only 5.12 msec for $d = 3$, but takes more than 10^{10} million years for $d = 10$ [96].

There are many variants of the basic definition of PAC learning [47]. One important variant defines a notion of complexity or size of the target concept [15]. The complexity of f, denoted by $s(f)$, measures the size of the *smallest* representation of the target concept f in the representation $\mathcal{H}$ used by the algorithm. In this variant, the number of training examples and the computation time are also allowed to grow polynomially in the complexity $s(f)$ of the target concept.

6.3 Methods for Proving PAC Learnability

There are mainly two methods for proving a class of concepts $\mathcal{F}$ to be PAC learnable:

1. proving directly and explicitly that there is a polynomial time algorithm that, for any concept in the class, finds a hypothesis that satisfy the PAC criteria [103].

2. showing that there is a polynomial time algorithm that finds, for any concept in the class, a hypothesis in a particular hypothesis class that is *consistent* with a given training sample, *i.e.* agrees with every example in the sample. If the hypothesis representation is small enough and the sample large enough, this achieves a form of data compression that is shown, by statistical arguments, to be sufficient to achieve PAC learnability [16, 15].

The first method is problem-dependent in the sense there is no general principle behind it. Most of the learning algorithms we will describe later have been developed through this method.

Algorithms developed using the second method are known as *Occam Algorithms* [16, 15], in reference to Occam's razor. In this case, the size of the sample is related to the expressive power (complexity) of the hypothesis class: the higher this power is, the larger the size of the sample will be. The expressive power of the hypothesis class can be measured in terms of its cardinality (if it contains a finite number of concepts) or by a combinatorial parameter known as the Vapnic-Chervonenkis (VC) dimension.

Definition 6.2 *The VC dimension of an hypothesis class $\mathcal{H}$, denoted by VCdim($\mathcal{H}$), is defined to be the maximum number d of instances that can be labeled as positive and negative in all 2^d possible ways, such that each labeling is consistent with some hypothesis in $\mathcal{H}$. In other words, VCdim($\mathcal{H}$) is the maximum number d of points for which one can perform all the 2^d possible dichotomies using concepts from $\mathcal{H}$.*

The following lemma, from Blumer *et al.* [15], shows that PAC learning can be reduced to that of finding an hypothesis consistent with a polynomial number of examples.

Lemma 6.3 ([15]) *Let $\mathcal{H}$ be a non-trivial, well-behaved, class of functions mapping $\Re^n$ to $\{-1, +1\}$. Then for any $0 < \epsilon, \delta < 1$ and any sample of at least*

$$m_0 = \max\left(\frac{4}{\epsilon}\log\frac{2}{\delta}, \frac{8\,VCdim(\mathcal{H})}{\epsilon}\log\frac{13}{\epsilon}\right)$$

randomly selected examples labeled according to some target function f, the probability that any function from $\mathcal{H}$ that is consistent with those examples has error at most ϵ with respect to f is at least $1 - \delta$.

Thus, to PAC learn a target class $\mathcal{F}$ using a hypothesis class $\mathcal{H}$, all that we need is to draw a sample of size m_0 and use a polynomial time algorithm to find an hypothesis in $\mathcal{H}$ consistent with this sample. Note that we have interest in keeping $\mathcal{H}$ as small as possible: for a given number m_0 of examples, the lower the VCdim(H) is, the higher the confidence $(1-\delta)$ will be. This is the essence of Occam's razor: *choose the simplest explanation.*

Obviously, the smallest class of concepts $\mathcal{H}$ that can PAC learn a class $\mathcal{F}$ is $\mathcal{F}$ itself. In this case, finding a consistent hypothesis is synonym of finding the smallest (minimum) consistent hypothesis. Unfortunately, the minimum consistency problem for many interesting concept classes is NP-complete, which implies that these classes are not *properly* PAC learnable [14, 58, 81, 65]. For example, learning two layer networks with n inputs, k hidden units, and one output unit using the same kind of network is NP-complete [14]. This however does not imply the non PAC learnability of these classes. To achieve PAC learnability, it is sufficient to find, in polynomial time, a consistent hypothesis *polynomially* larger than the smallest one. More precisely, if the size of the smallest consistent hypothesis is s and the size of the training sample is m, it suffices to find a consistent hypothesis of size $s^c m^\alpha$ for some constant $c \geq 1$ and $\alpha < 1$. Note the factor m^α: α is strictly less than one, in order to have some data compression. To continue with our example, to learn the networks described above, it is sufficient to find consistent networks say with n^2 inputs, $10 \times k$ hidden units, and one output. But even this relaxation does not solve the consistency problem for many target classes.

Up to date, the PAC model (definition 6.1) has accumulated more negative (non-learnability) results than positive ones [81]. This is due mainly to the model's distribution-free characteristic which many people now think it is too demanding.
One natural variation is to make the model distribution-specific [79]:

Definition 6.4 *The class $\mathcal{F}$ is said to be PAC learnable from examples on distribution D if there is an algorithm A that achieves the PAC criterion for that particular distribution.*

The most commonly considered specific distributions are the uniform and product distributions.

Another variation is to add active learning to the model by allowing *membership queries* [2, 103]. In a membership query, the learning algorithm presents an input vector $\mathbf{x}$ and is told, by an oracle, its classification $f(\mathbf{x})$:

Definition 6.5 *The class $\mathcal{F}$ is said to be PAC learnable from examples and membership queries if there is an algorithm A that achieves the PAC criterion using examples and membership queries.*

This extension seems natural as humans tend to use queries in learning. Many classes known to be not PAC learnable in the distribution-free setting, become PAC learnable under specific distributions and/or using membership queries [2, 19, 41, 79, 90].

6.4 PAC Learning and Neural Networks

To discuss the learnability of neural networks within the PAC model we need to upgrade our terminology somewhat.

Clearly, every network architecture defines a class of concepts: each concept in the class is associated with a given setting of the weights and the thresholds in the network. For example, the single perceptron architecture defines the class of single perceptrons (or halfspaces): each set of values for the weight vector and the threshold defines a particular concept–perceptron. In this sense, we may talk of learning the class of concepts defined by a network or simply learning the network. In the same way, we may think of the training examples as being labeled by the target concept or simply as being labeled by a network with a given setting of the weights and thresholds, representing the target concept.

If we adopt distribution-free PAC's point of view, the list of neural networks that are proven learnable is deceptively small:

- Single halfspaces (perceptrons with real weights) are properly PAC learnable under any distribution [15]. The VC dimension of perceptrons is $(n+1)$. All we need to do to learn a target perceptron is to draw $m_0(\epsilon, \delta, n+1)$ examples (lemma 6.3) and use the polynomial time Karmarkar's algorithm [56] to find a separating hyperplane. Note that the perceptron learning rule is not polynomial in the worst case.

- Border augmented symmetric differences of halfspaces (an XOR of halfspaces) are PAC learnable under any distribution [104, 14]. This is due to the fact that any concept in this class can be reduced to a single halfspace concept.

Moreover, the general class of multilayer networks has been proven to be not PAC learnable in the distribution-free setting. This remains true even if the learning algorithm is allowed to represent its hypothesis in ways other than a network [4, 60]. Blum and Rivest [14] proved that the very simple class of two-layer two-hidden-unit networks are not properly PAC learnable in the distribution-free setting. More negative results can be found in [14, 65, 81].

Very little is known about the PAC learnability of neural networks under specific distributions (definition 6.4) or using membership queries (definition 6.5). Baum [11] proved that intersections of two homogeneous halfspaces (AND of two perceptrons with zero thresholds) are PAC learnable under distributions symmetric with respect to the origin. The same author described a query-based algorithm for PAC learning halfspace intersections and two-layer four-hidden-unit networks when the input space is $\mathcal{R}^n$ [12]. However, the latter algorithm breaks down if the input space is discrete [1].

In the coming chapters, we will look for positive PAC learnability results for neural networks by

- restricting the network connectivity,
- restricting the values of the weights,
- restricting the distributions of examples,
- or any combination of the above.

This will enable us to identify some new classes of neural networks that are PAC learnable in the chosen setting.

[1] Baum's query-based algorithm can be hardly called a learning algorithm. In fact, it is a binary search similar to the one usually used to track and find the roots of a real function on a given interval.

Chapter 7

On Learning Halfspace Intersections and Neural Decision Lists

7.1 Introduction

As we said in the previous chapter, if we adopt the distribution-free PAC's point of view, the list of neural networks that are learnable is deceptively small. To our knowledge, only single halfspaces [15] and border augmented symmetric differences of halfspaces [14, 104] have been proven to be learnable. Even simple neural concepts like the intersection of two halfspaces are known to be not properly learnable under an arbitrary distribution of examples [14].

Many people believe now that the distribution free aspect of the PAC is too demanding. Bartlett and Williams [8] have proposed to allow only *reasonable* distributions (*i.e.* bounded distributions which are nonzero everywhere in the domain). One such distribution is the uniform one. Hence, a meaningful question to ask is whether or not simple neural concepts are PAC learnable on the uniform distribution. By simple neural concepts we mean concepts that can be represented as simple combinations of perceptrons (halfspaces). In this chapter (and in following chapters), we set to address this question.

First, we investigate the PAC learnability of *halfspace intersections* [1] under the uniform distribution of examples. The task here is to find, in polynomial time, a halfspace intersec-

[1] The union is the complement of the intersection and can be treated similarly.

tion hypothesis network that approximates the most the halfspace intersection target net. Furthermore, we allow for a possibly larger number of halfspaces in the hypothesis net. By Occam's razor principle (section 6.3), if the hypothesis network is not too large compared to the target network, we are guaranteed to learn. We stress here the fact that algorithms such as backpropagation [89] and Cascade-Correlation [28] do not solve our problem because there is no guarantee that they do converge to a solution in polynomial time.

We formalize the problem of learning halfspace intersections as a *set covering problem*. This leads us to consider the following sub-problem: given a set of non linearly separable examples, find the largest linearly separable subset of it. We give an approximation algorithm for this NP-hard sub-problem. Simulations, on both linearly and non linearly separable functions, show that this approximation algorithm works well under the uniform distribution, outperforming the Pocket algorithm used by many constructive neural algorithms.

Based on this approximation algorithm, we present a greedy method for learning halfspace intersections. Although we are not yet able to prove its PAC correctness, this greedy method does very well experimentally under the uniform distribution of examples up to 50 dimensions. To our knowledge, the tests reported here go beyond any in the literature in terms of testing generalization by a greedy method in high dimensions.

The greedy method for halfspace intersections is extended to a class of concepts we call *neural decision lists*. These are a generalization of the decision lists of Rivest [85] by allowing each node to be a halfspace (perceptron). This class of functions is strictly richer than halfspace intersections (unions).

Both greedy methods for halfspace intersections and neural decision lists are tried on real-world data with very encouraging results. Their performance is comparable to C4, a "state of the art" tree-induction algorithm [84]. This shows that these simple neural concepts are not only important from the theoretical point of view, but also in practice.

This chapter is organized as follows. In section 7.2 we present some definitions. Our greedy method for halfspace intersections is presented in section 7.3. In section 7.4, we present our approximation algorithm for training single neurons. Our approach is generalized to neural decision lists in section 7.5. In section 7.6, we present the numerical results of our extensive simulation on both random nets and real-world data. The conclusions are

summarized in section 7.6.

7.2 Definitions

Let the input space $\mathcal{X}^n$ be the set $\{-1,+1\}^n$ or a subset of $\mathcal{R}^n$.

An *example* of a boolean function, $f : \mathcal{X}^n \rightarrow \{-1,+1\}$, is an an ordered pair $< \mathbf{x}, f(\mathbf{x}) >$ where $\mathbf{x} \in \mathcal{X}^n$. Point $\mathbf{x}$ is said to be a positive example if $f(\mathbf{x}) = +1$, otherwise it is said to be a negative example. A *sample* is a set of examples. We assume that the distribution D generating the examples is uniform on $\mathcal{X}^n$.

A *linear threshold function* on a set of n variables is specified by a vector of n real valued weights w_i and a single real valued bias w_0. The output of the function is $+1$ or -1 depending on whether the following inequality holds

$$\sum_{i=1}^{n} w_i x_i + w_0 > 0$$

Such functions are also referred to as *perceptrons* or *halfspaces*. We denote by H the positive halfspace $\{\mathbf{x} : \mathbf{w} \cdot \mathbf{x} + w_0 > 0\}$ and by $\overline{H}$ its complement. Halfspace H is said to *cover* example $\mathbf{x}$ if $\mathbf{x} \in H$. Halfspace H is said to be *consistent* with a sample S if all positive examples of S are covered by H and and all negative examples of S are covered by $\overline{H}$.

A function $f : \mathcal{X}^n \rightarrow \{-1,+1\}$ is said to be a *halfspace intersection* if it can be written as a conjunction (AND) of halfspaces. These functions have an obvious neural net representation: a feedforward network (FFN) made of one layer of hidden units connected to a single output unit that performs the "AND" operation. (see fig. 7.1).

We generalize the notion of *decision lists* introduced by Rivest [85] to *neural decision lists* (hereafter NDL). A NDL (fig. 7.2) is a list $\mathcal{L}$ of pairs

$$(H_1, v_1), (H_2, v_2), ..., (H_r, v_r)$$

where each H_i is a halfspace and v_i is a value in $\{-1,+1\}$. The last halfspace H_r is the constant function [2] $+1$. This defines a function as follows: for any $\mathbf{x}$, $\mathcal{L}(\mathbf{x})$ is defined to be

[2] We may think of the constant (halfspace) function as the halfspace covering the whole input space. A perceptron with zero weights and a positive bias will do the trick.

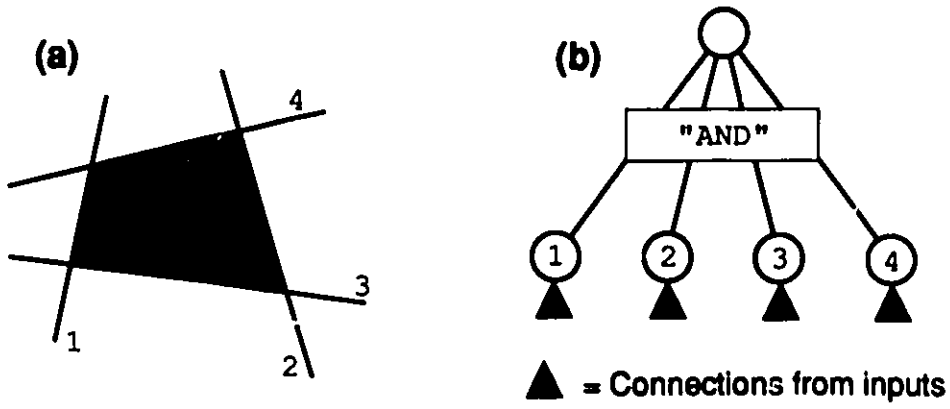

Figure 7.1: (a) The function represented by the intersection of four halfspaces. The shaded region represents the set of positive examples. (b) The equivalent FFN. The input units are not shown.

equal to v_j where j is the first (least) index for which $\mathbf{x} \in H_j$. As in [85], we may think of a NDL as an extended **"if–then–elseif–...else–"** rule (see fig. 7.2). Compared to Rivest's decision lists, NDL's have the same structure, but the complexity of the decision allowed at each node is greater.

This class of representations is strictly richer than halfspace intersections (unions). Indeed, any boolean function on a boolean (or discrete) domain has a NDL representation. Moreover, there always exists a NDL consistent with any finite sample of a boolean function on a continuous domain (i.e. a subset of $\mathcal{R}^n$). We will present numerical results on "real-world" data sets that have an efficient NDL representation.

NDLs have a simple FFN representation (see fig. 7.2): it is a type of FFN known as a *cascade net* [28, 30, 65] because hidden units need to be updated one after the other (in "cascade"), starting from the first. The reduction from NDLs to cascade FFNs is given in section 7.8.

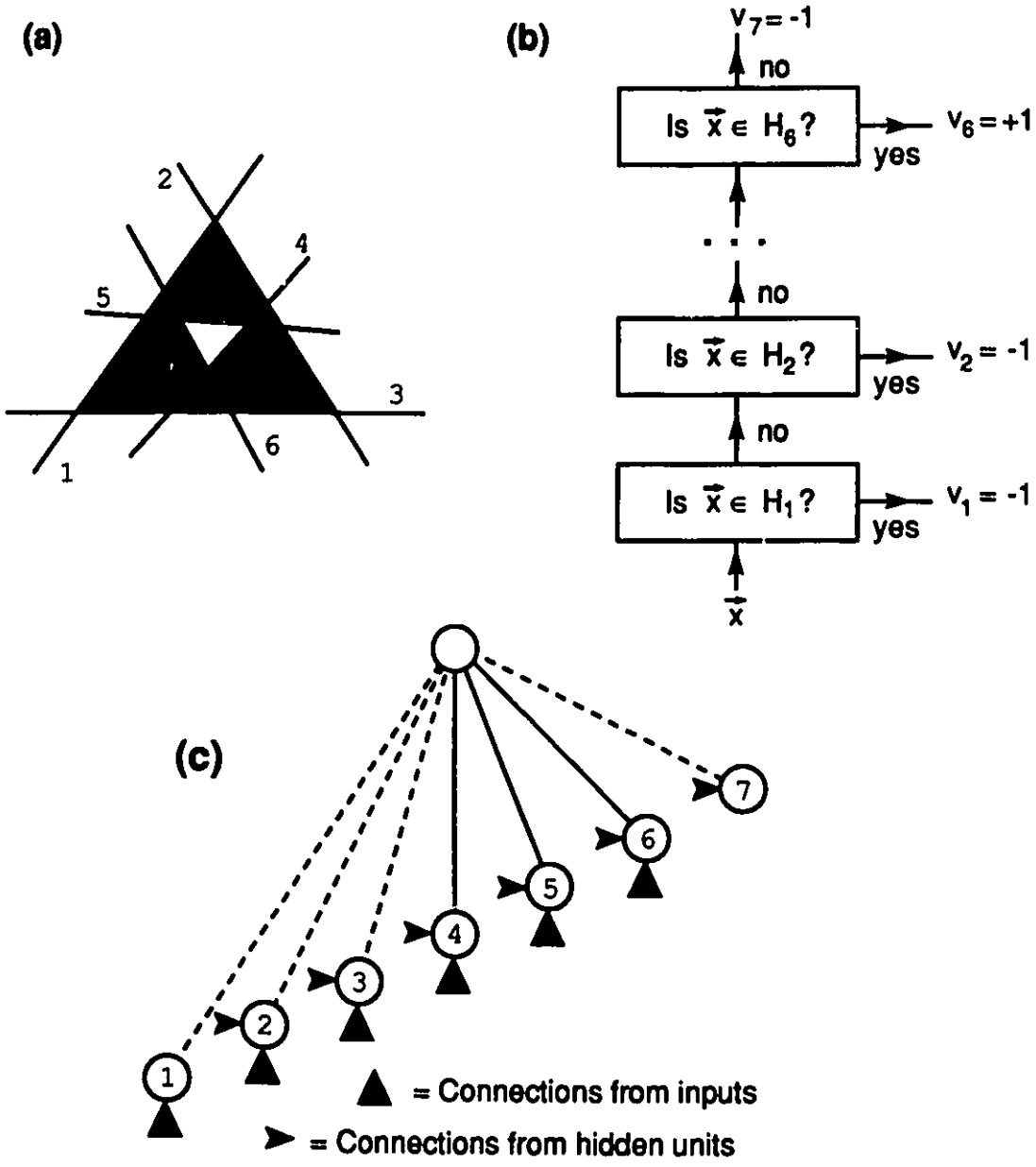

Figure 7.2: (a) A function to be learned; the shaded region represents the set of positive examples. (b) A NDL performing this function. (c) The equivalent cascade FFN. Each hidden unit receives connections from the input units (not shown) and from the other hidden units below it. Excitatory connections going to the output are indicated by full lines whereas inhibitory connections are shown by dashed lines. See section 7.8.

7.3 Learning Halfspace Intersections

When given a training sample from a halfspace intersection target function, the goal of the learner is to find a halfspace intersection that best approximates the target function. Here, each positive example must be consistent with each and every halfspace whereas each negative example needs only to be consistent with one halfspace. The problem, often called the credit assignment problem (CAP) [9], is to decide which halfspace must be consistent with a given negative example. One way to bypass the CAP is to adopt the following greedy method:

1. Let S^- be the set of all negative examples.
2. If S^- is empty, Halt.
3. Find the halfspace H that is consistent with the largest subset of S^- and all the positive examples.
4. Add H to the hypothesis halfspace intersection. Remove this subset from S^-.
   Goto 2.

Obviously, this greedy method will build as hypothesis a halfspace intersection consistent with all the training examples.

Let us call the optimization problem encountered in step 3 the *densest halfspace covering problem*. Suppose for a moment that we have a way to solve this optimization problem *exactly*. It is easy to imagine some distributions of examples for which the halfspace consistent with the largest number of negative examples is quite different from any one of the target halfspaces—thus causing the above greedy method to give a larger number of halfspaces than the minimum. This is not a real setback because our goal is not to find the *minimum* number of halfspaces, which is an NP-complete problem, but to find a *good* approximation. Under the above assumption (that step 3 can be solved exactly), the greedy method is equivalent to the standard greedy algorithm for set covering [21, 53]. Hence, it is guaranteed to find a halfspace intersection with a number of halfspaces not greater than $h \ln(m) + 1$ in the worst case, where m is the number of examples and h is the smallest number of

halfspaces in any halfspace intersection consistent with all the examples. By the Occam's razor principle [16, 15] (see section section 6.3), this would be sufficient to PAC learn this class of concepts.

Unfortunately, the densest halfspace covering problem contains, as a particular case, a known NP-Complete problem: the densest hemisphere problem [31, 54]—finding the largest linearly separable subset (positive or negative) from a data set. Hence, for the greedy method to PAC learn the class of halfspace intersections, one needs to find some approximation algorithm for the densest halfspace covering problem with a good performance guarantee in the *worst case* [31]. Unfortunately, we are not aware at present of any such algorithm that runs in polynomial time [3]. We present, in the next section, an approximation algorithm for the densest halfspace covering problem which runs in polynomial time and does extremely well experimentally on both real and artificial data, although we are not able yet to prove its correctness under the uniform distribution.

7.4 Approximation Algorithm for the Densest Halfspace Covering Problem

We have seen in the previous section that finding an approximation algorithm for the densest halfspace covering problem is of fundamental importance to PAC learning halfspace intersections (and probably for other learning problems). Moreover, all constructive (or growth) neural net algorithms need such an approximation algorithm for training at the single neuron level. Here we present our approach to this problem.

7.4.1 Description of the Approximation Algorithm

We are of course concerned with the case where the data is not linearly separable because the linearly separable case is directly solvable by Linear Programming (LP). One way to approach this problem is to try to minimize the perceptron criterion function [27] which can

[3] The only known algorithm for the densest hemisphere problem is the *tree algorithm* [38] which runs in time exponential in n.

be converted to a linear cost function and hence, solvable by LP. This however has nothing to do with minimizing the number of misclassifications, which is a truly nonlinear cost function of the weights. As a consequence, a set of weights that minimize the perceptron criterion function will, in general, contain too many misclassified examples; each being close to the separating hyperplane.

Another way to approach this problem is to use, incrementally, a LP procedure to try to incorporate one example at a time into a linearly separated data set. This is the approach we adopt here. Let us look at the following greedy heuristic; call it the Incremental Linear Programming (IncLP) algorithm:

IncLP(S^+, S^-, L, H)

Parameters

S^+: the set of positive examples.

S^-: the set of negative examples.

L: an initial set of negative examples separable from S^+ (may be empty).

H: an initial halfspace consistent with S^+ and L.

Output: (L, H) where L is subset of negative examples separable from S^+ and H is a halfspace consistent with S^+ and L.

Description: The algorithm builds on the set L by adding to it negative examples from S^-.

1. Set $R = S^-$.
2. If R is empty, Return (L, H).
3. Choose (possibly at random) an example $\vec{x}$ from R. Set $R = R - \{\mathbf{x}\}$.
4. By using your favorite LP procedure, try to find a halfspace H^a consistent with $L \cup \{\mathbf{x}\}$ and S^+.
5. **If** such halfspace exists **Then** set $L = L \cup \{\vec{x}\}$ and $H = H^a$.
6. Goto step 2.

The **IncLP** will return a halfspace H consistent with a subset L of negative examples and all positive examples. It may happen however that a bad sequence of negative examples will be chosen such that the resulting subset L is small. Since the number of linearly separable dichotomies increases exponentially with n [22], it is not feasible to find all of them. So we must instead look for ways to find, with high probability, large linearly separable subsets.

The first procedure one may try is what we call **mult_IncLP**. It consists simply of running **IncLP** a certain number of times on the same training set. After each pass we record both the halfspace and the size of the subset found, and then change randomly the order of the examples in the training set for the next pass. At the end, **mult_IncLP** returns the halfspace that covers the largest subset found from these multiple attempts. However, it may happen that only a small subset will be found by **IncLP** at each attempt, thus causing **mult_IncLP** to return a small subset. In fact, the simulation results show that this is the case.

Note that each time **IncLP** returns a small subset, this means that it has run into a "bad sequence" of negative examples. Thus a good "rule of thumb" would be to remove these negative examples from a working set W and use **IncLP** to find another subset from the *remaining* examples in W. This can be repeated until no negative example is left in W. At the end, we just retain the largest subset found. Hence, instead of **mult_IncLP**, we propose the following heuristic for our approximation algorithm:

Find_large_neg_subset(S^+, S^-)

Parameters

S^+: the set of positive examples.

S^-: the set of negative examples.

Output: (L_m, H_m) where L_m is a (hopefully large) subset of negative examples linearly separable from S^+ and H_m is a halfspace consistent with L_m and S^+.

Description:

1. Set $W = S^-$ (W will hold the remaining negative examples).
   Set $U = \emptyset$ (U will hold the removed negative examples).
   Set $L_m = \emptyset$ (L_m will hold the largest subset found).

2. If W is empty, Return L_m.

3. Set $L = \emptyset$ and $H = +1$ (i.e. H is the constant halfspace).

4. $(L, H) =$ **IncLP**(S^+, W, L, H).

5. Set $W = W - L$.

6. $(L, H) =$ **IncLP**(S^+, U, L, H).

7. **If** $|L| > |L_m|$ **Then** $(L_m, H_m) = (L, H)$.

8. Set $U = U \cup L$. Goto step 2.

Note that, in addition to what has been said above, **Find_large_neg_subset** tries (in step 6) to build a larger subset L of negative examples by trying to include the negative examples belonging to the subsets already removed. More generally, it is clear that we can use **IncLP** in a similar way to find a large linearly separable subset of $S^+ \cup S^-$ with the constraint that a certain subset (not necessarily S^+) must be present in the solution. We will present, in section 7.6, numerical results that indicate the superiority of **Find_large_neg_subset** over the **mult_IncLP**.

Baum [9] has suggested that no "good" incremental approximation algorithm exists for the densest halfspace covering problem. His arguments can be summarized as follows: for n large, with very high probability, any random set of n negative examples (on n inputs) will be separable from the positive examples. Because of that, it provides little information that one is getting a "good" halfspace. So essentially no further negative examples can be added to this set. Indeed, it is easy to imagine different scenarios where this approximation algorithm would give only a very small linearly separable subset. For example, suppose that the target function is the intersection of two halfspaces. Suppose also that the distribution of the m^- negative examples is correlated to the distribution of positive examples as follows. Let Π be the polytope formed by the positive examples. For each face of Π, we have two (and only two) negative examples such that any hyperplane that separates *both* of them from Π, cannot separate any other negative example. Hence, if IncLP always starts with two such examples, the approximation algorithm will return subsets containing only

two examples. This will cause the greedy method (section 7.3) to give $m^-/2$ halfspaces. However, all such catastrophic scenarios that we can think of have one thing in common that makes them very unlikely to occur under the uniform distribution or in practice: the distribution of positive examples is *correlated* with the distribution of negative examples which is also correlated with our (random) strategy for choosing the examples in the IncLP algorithm. In other words, all these catastrophic scenarios are based on some malicious distributions controlled by adversaries [10]. Thus, we think that our approximation algorithm will do well for simple functions like halfspace intersections under the uniform distribution of examples and probably also under other reasonable distributions [8]. Our numerical simulations (section 7.6) suggest that this is indeed the case for both real and artificial data.

In the worst case, our approximation algorithm runs in polynomial time if we use Karmarkar's polynomial time algorithm for LP [56]. We have, however, used the Simplex algorithm [23] since, although its running time can increase exponentially with n in the worst case, it is reputed to be very efficient in practice. In fact, Smale [98] showed that the number of pivot operations needed to solve a LP problem is almost always no larger than the number of variables or the number of constraints, whichever is the larger. We have implemented the algorithm described in [80, chapter 2], which uses Bland's rule to avoid cycling, and have encountered no problems.

Finally, there exists another approximation algorithm, known as the pocket algorithm [30], to find the largest separable subset of examples. One feature that might make this algorithm attractive is its "convergence" theorem: *Given a set of examples and a probability $p < 1$, there exists an N such that after $l \geq N$ iterations of the pocket algorithm, the probability of finding the largest separable subset of examples exceeds p.* However, this theorem is not a strong statement since it gives no upper bound on the number N of iterations needed. Indeed, as Gallant himself emphasizes [30], the proof relies on the fact that there is a finite (but very small) probability that the examples from the largest (or a largest) linearly separable subset will be picked repeatedly by the perceptron which, following the perceptron convergence theorem, will then find the corresponding optimal set of weights. Of course, we have also a similar "convergence" theorem if we use our approximation algorithm repeatedly, each time changing (at random) the order of the examples. But in practice, we find

it sufficient to use our approximation algorithm only once. An experimental comparison of the two approaches is given in section 7.6.1.

7.5 Learning Neural Decision Lists

7.5.1 Two-Class Problems

Here we extend the greedy method given in section 7.3 to the case of neural decision lists. The additional problem that arises now is that the set of positive examples is not necessarily convex (see fig. 7.2a). Hence, the greedy method will need to alternate between covering (cutting) subsets of positive and negative examples.

Consider the example of fig. 7.2a. Each time a halfspace covers a subset of examples, these are removed from the training set. For the first 3 halfspaces, only negative examples can be covered. Then, the positive examples will be covered by the next three halfspaces. Finally, the last halfspace will cover the remaining examples which are all of the same target: negative in this case.

We therefore propose the following greedy method for learning NDLs:

Learn_NDL(S^+, S^-)

1. Let S^+ be the set of positive examples and S^- be the set of negative examples.
2. If $S^+ = \emptyset$, append the pair $(H_r = 1, -1)$ to the decision list and stop.
3. If $S^- = \emptyset$, append the pair $(H_r = 1, +1)$ to the decision list and stop.
4. Find the halfspace H^+ that covers the largest number of examples in S^+ and none of the examples in S^-. Let f^+ be the *fraction* of examples from S^+ covered by H^+.
5. Similarly, find the halfspace H^- that covers the largest number of examples in S^- and none of the examples in S^+. Let f^- be the *fraction* of examples from S^- covered by H^-.

6. **If** $f^+ > f^-$ **then** append the pair $(H^+, +1)$ to the decision list and remove from S^+ the examples that are covered by H^+.
   Else append the pair $(H^-, -1)$ to the decision list and remove from S^- the examples that are covered by H^-.

7. Go to step 2.

Note that, at each step, we have decided to retain the halfspace that covers the largest *fraction* of the remaining positive (negative) examples. The reason is that the fraction of examples reflects more the probability measure we are covering than the actual number of examples [4].

7.5.2 Multi-class Problems

There are several ways one can extend this approach to multiple class functions [5]. Say that we have Q classes so that the function to learn $\tau(\vec{x})$ can take any value fron $1, \cdots Q$. One way to build the multiple class NDL is to find, at each step, the halfspace that covers (cut) the largest fraction of examples *from one class only*. The following heuristic uses this idea to build NDLs for multiclass problems:

1. For $\tau = 1, \cdots Q$, let S^τ be the set of examples of the class τ.

2. If *all* but one of the S^τ are empty, append the pair $(H_\tau = 1, \tau_0)$ to the decision list and stop (τ_0 is the class of the non-empty set).

3. For $\tau = 1, \cdots Q$:
   if $S^\tau \neq \emptyset$, find the halfspace H^τ that covers the largest fraction f^τ of S^τ and none of the examples in $\cup_{\sigma \neq \tau} S^\sigma$.

[4] The **Learn_NDL** procedure is in fact an improved version of the sequential learning of chapter 4. We can now understand the reason the latter works well for the parity and the mirror symmetry functions, but not for cyclic permutation function: the first two have very small NDL representations, the third one does not.

[5] This is the only place in the thesis where we touch on multi-class problems.

4. Choose the H^σ that covers the largest fraction f^σ and remove these examples from S^σ. Append to the decision list the pair (H^σ, σ).

5. Go to step 2.

We can think of other, and probably better, ways to use these halfspace covering heuristics in multi-class situations by incorporating some knowledge about the dispersion of the data. We could, for example, perform $\lceil \log_2(Q) \rceil$ different dichotomies of the Q classes and then use, in parallel, the **Learn_NDL** algorithm of the last section to create $\lceil \log_2(Q) \rceil$ different binary classification NDL's. These dichotomies could be done by performing a principal component analysis on the data or by using any other criteria like those presented in [97]. The point we want to make is that our approach is flexible enough to be applied in a rich variety of ways to multi-class problems.

7.6 Experimental Results

7.6.1 The Approximation Algorithm vs. the Pocket Algorithm

Because most of the constructive algorithms use the pocket algorithm at the single neuron level to find the "optimal" halfspace, we include here a numerical comparison of our approximation algorithm with the pocket algorithm on both linearly and non linearly separable functions. We took great care in implementing the pocket as explained in the references cited above.

The Linearly Separable Case:

In this case, the approximation algorithm requires only one pass of IncLP. For each test, we generate at random a hyperplane in the $[-1, +1]^n$ region. The training examples were drawn randomly in $[-1, +1]^n$ and classified according to this target function. Because both algorithms are guaranteed to converge to a solution, the performance criteria will be the time needed to find it. The average CPU time taken on a 1.4 MFLOPS computer (YARC's NuSuper accelerator board for the Macintosh) is reported in fig. 7.3 for $n = 10$. Although the CPU time depends on both the machine and the code written, it is a good measure

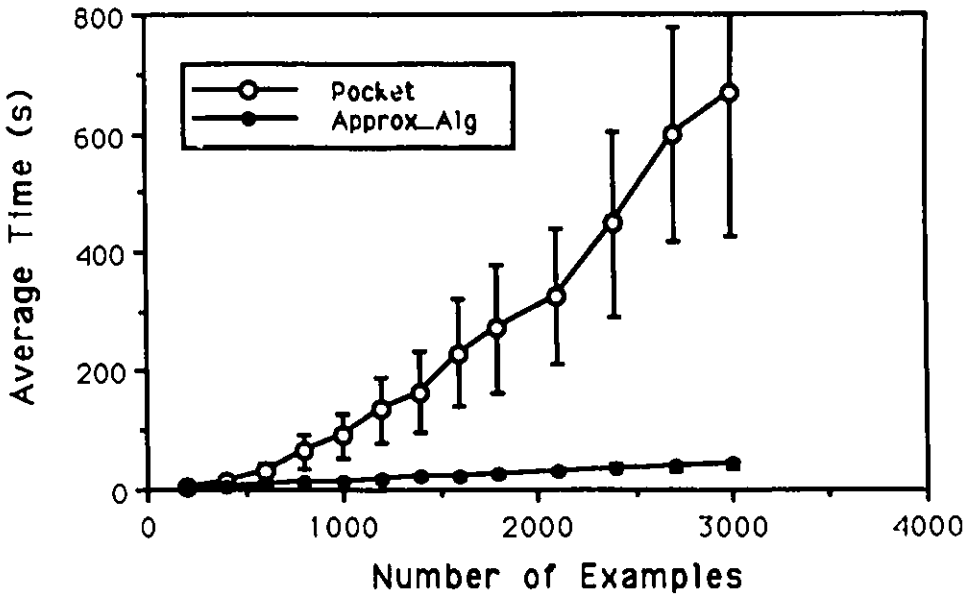

Figure 7.3: Comparison of the approximation algorithm with the pocket algorithm on linearly separable functions for $n = 10$. Each point on the graph is the average over 100 tests. Also shown are the standard deviations.

of the *relative* efficiency of the algorithms. One can see that the approximation algorithm outperforms the pocket by many orders of magnitude. Whereas the approximation algorithm scales linearly with the number of examples in the training set, the pocket clearly scales super-linearly with the number of examples. This can be explained by the fact that because the pocket chooses examples at random, it will spend most of its time checking examples already well classified.

The Non Linearly Separable Case:

One of the simplest non linearly separable function is the intersection of two halfspaces defined by parallel hyperplanes. So we used this problem to compare:
i) our approximation algorithm **Find_large_neg_subset**, ii) the pocket algorithm with rules (each positive example is taken as a rule that must not be violated), and iii) **mult_IncLP**

described in section 7.4.1.

The task is to classify the largest possible fraction of the negative examples, keeping all the positive examples well classified. The relevant question is: given the same amount of time and the same training set, which algorithm returns the largest cluster (subset) of negative examples? To answer this, we first run **Find_large_neg_subset** on a training set to find a subset of negative examples and note both the time spent by the algorithm and the size of the subset returned. Then, we run the pocket with rules for the same amount of time and on the same training set and we note the size of the subset returned. Finally we run **mult_IncLP** for the same amount of time taken by **Find_large_neg_subset** and record the size of the largest subset found (among the multiple passes of **IncLP**).

The results for the three algorithms are shown in fig. 7.4 for $n = 16$. Each point on the graph is the average over 50 tests: five different random target networks, each being tested on ten different training sets. Each target network is an intersection of two halfspaces defined by two parallel hyperplanes centered at the origin of the input space, a distance of 0.5 apart, and randomly oriented. Each example of the training set is generated uniformly in $[-1, +1]^n$ and classified according to the target function. Note that under these conditions the optimal plane covers half of the total negative measure.

Clearly, **Find_large_neg_subset** outperforms the pocket algorithm. The fact that it also outperforms **mult_IncLP** provides strong numerical evidence of the importance of removing bad sequences of negative examples when they are found. This is the basic difference between **Find_large_neg_subset** and **mult_IncLP**. Note also the large error bars (standard deviation of the cluster size returned) of **mult_IncLP** and the very small ones for **Find_large_neg_subset**. This shows that **mult_IncLP** is much less reliable and consistent than **Find_large_neg_subset**. Also, we must mention that, for a training set of 300 or more negative examples, **Find_large_neg_subset** returned, on average, an optimal cluster containing half of the total number of negative examples in the training set.

From these results, we conclude that **Find_large_neg_subset** is superior, on average (and almost always), to both the pocket and the **mult_IncLP** algorithms.

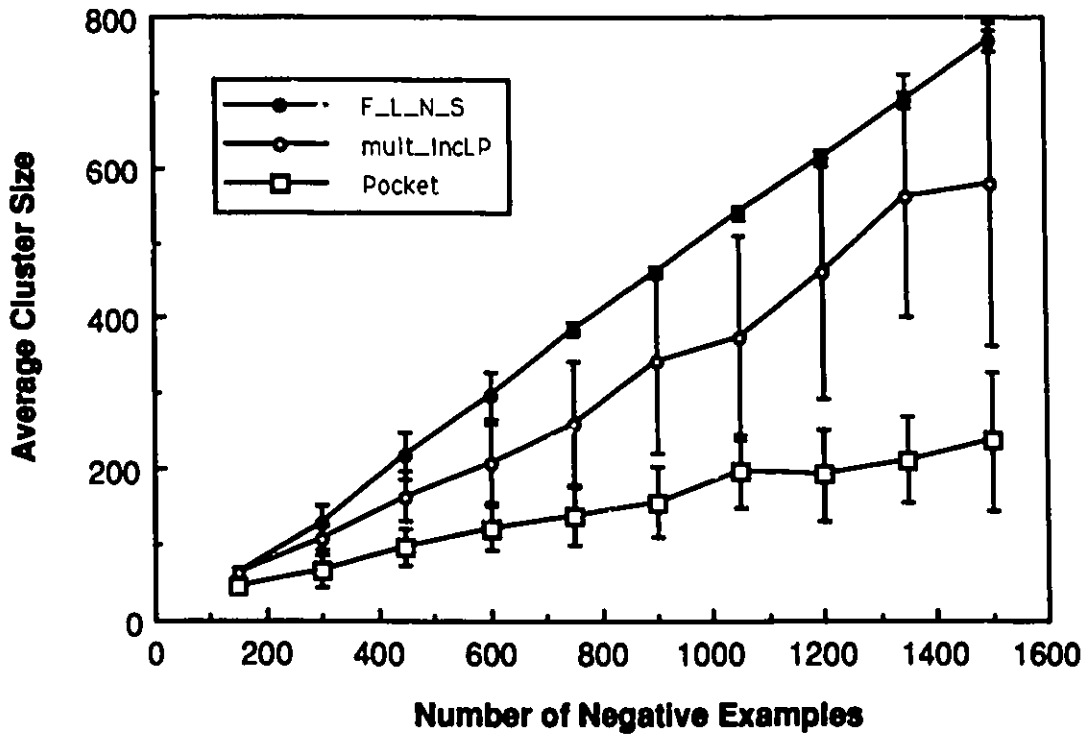

Figure 7.4: Comparison of the approximation algorithm **Find_large_neg_subset** the pocket algorithm, and **mult_IncLP** (see text) on non linearly separable functions for $n = 16$. The average "cluster size" is the average number of negative examples in the largest linearly separable subset found. Each point on the graph is the average over 50 tests. The error bars indicate the standard deviations.

7.6.2 Learning Halfspace Intersections

In this section, we describe the results of the extensive experimental tests performed using the greedy method of section 7.3, coupled with the approximation algorithm, to learn random halfspace intersections.

Our class of functions to be learned will be the class of h halfspace intersections or equivalently, the FFN containing one layer of h hidden units and one output unit implementing an "AND" of the hidden layer. We want to find out how the performance of the algorithm scales with the number of examples in the training set, the dimension of the input space and the number of halfspaces in the target net function.

For each test, the target net function is generated as follows: the value of each connection going from an input unit to a hidden unit is chosen uniformly and randomly in the interval $[-1, +1]$. The bias of the hidden units are also chosen randomly in the interval $[-1, +1]$. The output unit is hardwired to compute an AND of its inputs. We make sure that the region defined by the halfspace intersection is not void. This gives our target net function. The training examples were drawn randomly in $[-1, +1]^n$ and classified according to the target function. Another separate set of examples is drawn according to the same distribution, classified according to the target function, and used as a test set for the generalization ability of our algorithm.

We tried target nets with $n = 6, 8, 10, 12$ and $h = 2, 4, 6$. For each pair of values (n, h), the number of examples in the training set was varied approximately between 100 and 2000. The number of examples in the test set was kept constant at around 2000 examples. Typical results are shown in fig. 7.5 and fig. 7.6. We present both the average number of halfspaces in the hypothesis net returned by the algorithm and its generalization ability for different values of (n, h). Each point of these figures represents the average over 100 tests.

From fig. 7.5 and fig. 7.6, one can see that, on average, the algorithm is doing well. Not only the number of halfspaces returned by the algorithm is small enough to allow good generalization, but it also does scale very nicely with the size of the problem.

We have looked also at the worst case behavior of the algorithm over the set of tests performed. For each pair (n, h), we looked at the *maximum* number of halfspaces returned

by the algorithm at any test. This number was 4 for $(8,2)$, 5 for $(10,2)$ and 10 for $(10,4)$. This indicates that cases where the algorithm will perform poorly will be encountered very rarely. It would be interesting to investigate this question theoretically.

To avoid the objection that these are moderate size problems, we ran tests on $(20,2)$, $(30,2)$, $(35,2)$, $(40,2)$, and $(50,2)$. The results are presented in table 7.1. These tests were done as follows. We first generate at random a halfspace intersection function made of two n-dimensional halfspaces, orthogonal to each other and passing through the origin[6] of the continuous input space $[-1,+1]^n$ (note that for these target functions, the positive measure is $1/4$ of the total measure). Then a training sample consisting of $m/2$ positive examples and $m/2$ negative examples is generated uniformly at random. When the greedy algorithm returns its hypothesis function with k halfspaces, the generalization is tested separately on $M/2$ positive and $M/2$ negative new examples generated uniformly. The average (Gen) of these two scores is reported. Since the last few halfspaces of the hypothesis often cut only a few points, we also report the number of halfspaces r for which the hypothesis exhibits the best generalization (Opt-Gen).

One can see (table 7.1) that the number of examples needed to get a generalization rate of 98-99% is consistent with a polynomial increase with respect to the dimension of the input space (for $n \leq 50$). The results do not show any evidence of an exponential increase, as a function of n, in the number of examples needed to learn halfspace intersections, as it was conjectured by Baum [9]. On the contrary, these experimental results *strongly* suggest that the greedy method learns halfspace intersections under the uniform distribution.

An interesting function that has been the subject of controversy is the **mirror symmetry detector** [74, 89]. This function is defined as follows. The output is $+1$ if and only if the second half of the input bits is a mirror reflection of the first half [74]. This function is known to have at least two different neural net representations: an intersection of two parallel halfspaces and a majority function of n order-two-disjuncts. Since the latter is not a NDL, the greedy algorithm always finds the former one. The training was done on $m/2$ positive and $m/2$ negative examples obtained from the uniform distribution on $\{-1,+1\}^n$. Testing was done on $M/2$ positive and $M/2$ negative examples obtained from the same distribution.

[6] Of course, this information is not coded in the learning algorithm

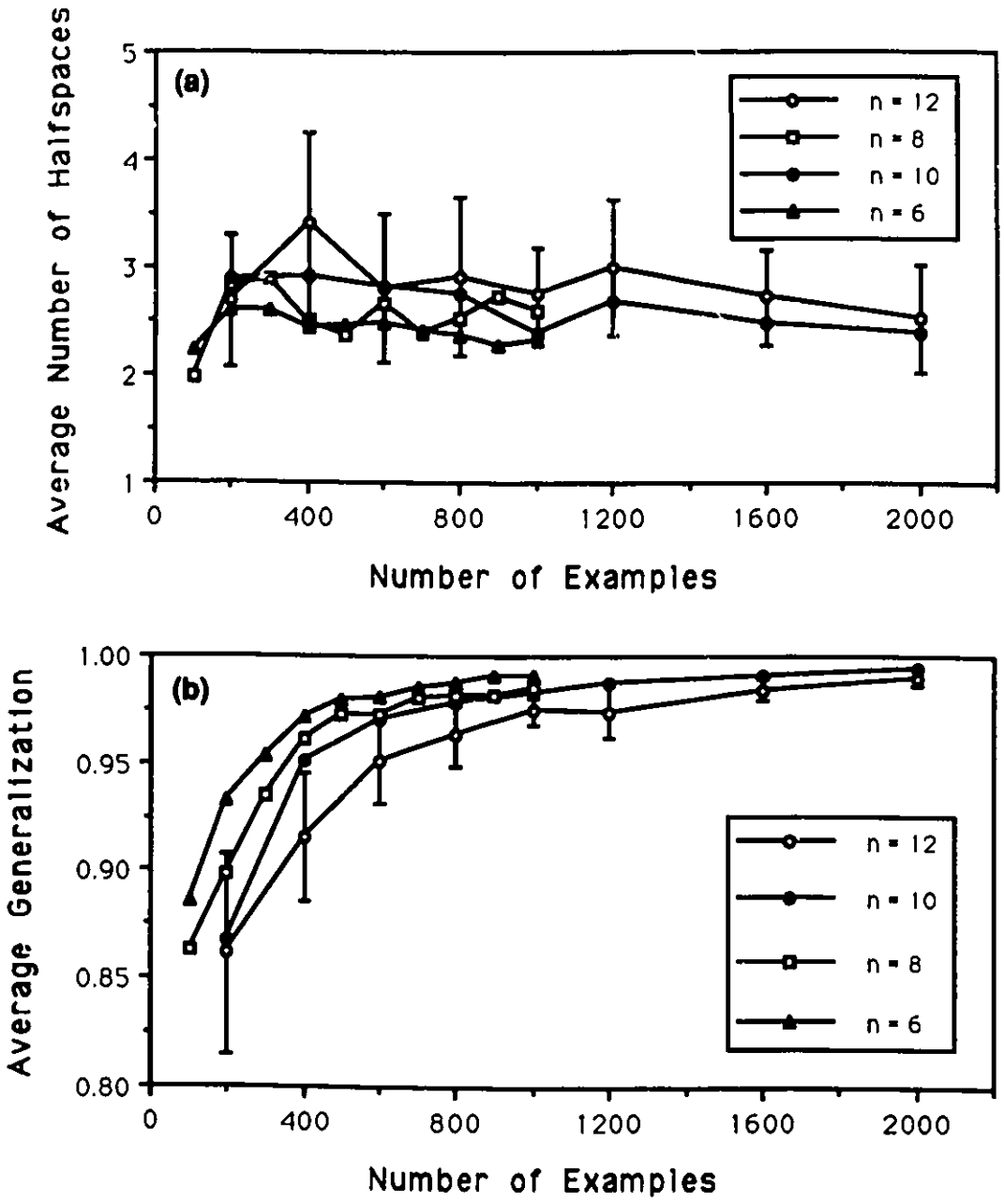

Figure 7.5: (a) The average number of halfspaces and (b) the average generalization of the hypothesis net when learning intersections of two halfspaces for different values of n. For clarity, only standard deviations for $n = 12$ are shown.

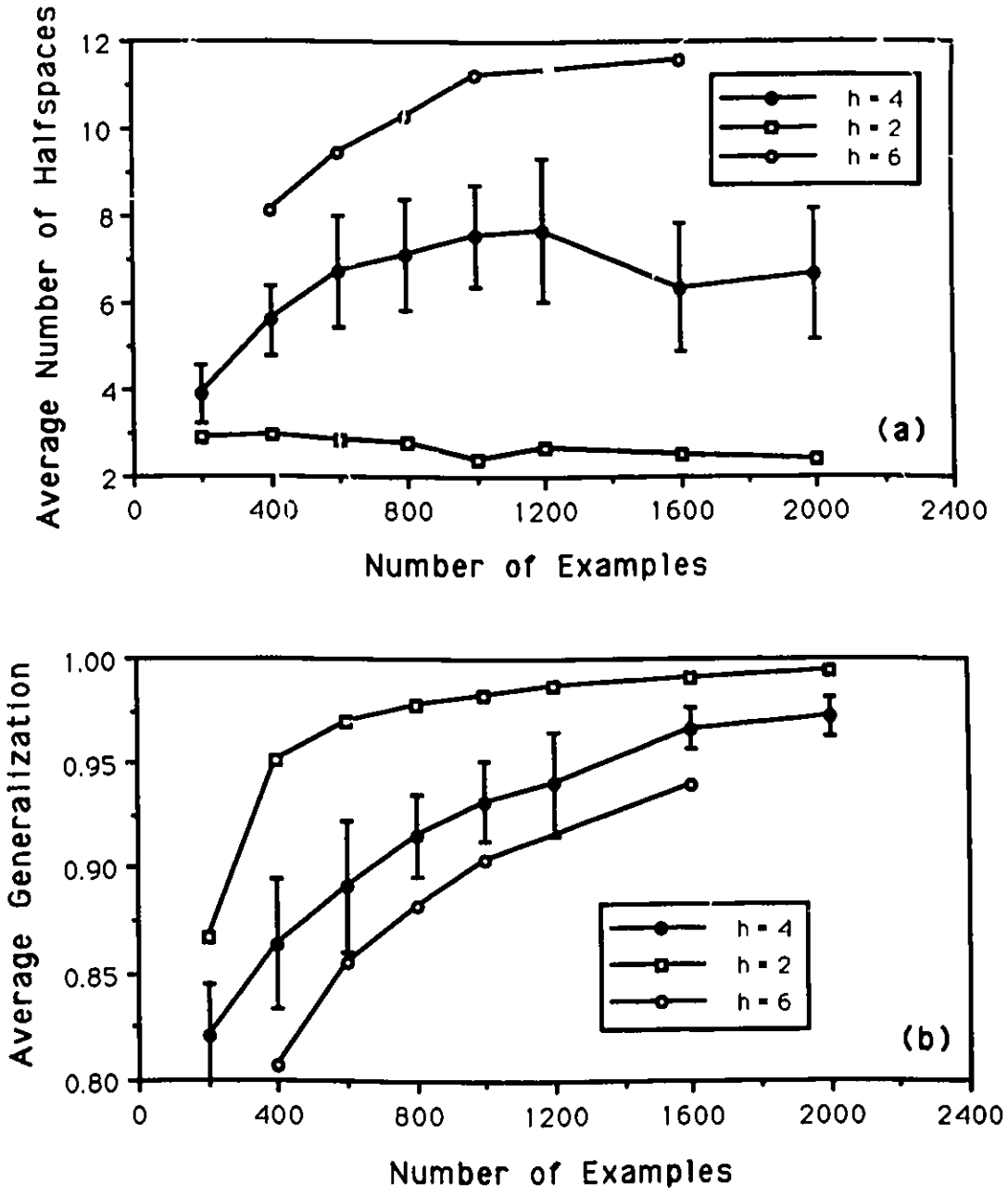

Figure 7.6: (a) The average number of halfspaces and (b) the average generalization of the hypothesis net when learning intersections of $h = 2, 4, 6$ halfspaces for $n = 10$. For clarity, only standard deviations for $h = 4$ are shown.

n	m	M	N	CPU time	k	Gen	r	Opt-Gen
10	3000	10000	10	$\sim$ 0.3 hrs	4.3	99.4% $\pm$ 0.1%	3.6	99.5 $\pm$ 0.2%
20	4000	20000	6	$\sim$ 0.7 hrs	5.0	98.1% $\pm$ 1.0%	3.3	98.8 $\pm$ 0.3
30	6000	20000	5	$\sim$ 1.6 hrs	4.3	98.6% $\pm$ 0.1%	3.6	98.9 $\pm$ 0.1%
40	10000	40000	1	$\sim$ 12 hrs	5	98.2%	4	98.6%
50	14000	40000	1	$\sim$ 36 hrs	5	98.8%	4	99.0%

Table 7.1: The simulation results for two halfspace intersections. n, m, and M are respectively the dimension of the input space, the size of the training set, and the size of the test set. N is the number of tests performed and (CPU time) is the average time needed to find a solution using an IBM RS/6000 machine. k is the number of halfspaces in the hypothesis intersection and (Gen) is the latter's generalization ability. r ($\leq k$) is the number of halfspaces in the hypothesis yielding the best generalization rate (Opt-Gen).

As the results show (see table 7.2), the greedy method was able to find a good approximation of this function after seeing very few examples. The function returned consisted always of two, almost parallel, halfspaces. For the sceptic, we have done five runs in 50 dimensions and have obtained an average generalization rate of 97.5% after training on only $m = 1600$ examples! Testing was done on $M = 20000$ examples. Note that it is important here to test separately on the positive and negative measure. Otherwise, a trivial net giving -1 to all examples would achieve almost perfect generalization on the uniform distribution since the positive region has only a fraction of $2^{-n/2}$ of the total measure. But here this trivial net gets only 50% generalization on the testing set. These results show how easy learning can be on a reasonable distribution: here the returned net needs only to be correct on inputs having roughly an equal amount of $+1$ and -1 entries.

To summarize, our numerical results strongly suggest that the algorithm is behaving as "Occam algorithm" and does learn halfspace intersections under the uniform distribution of examples.

m	100	200	400	600
<Gen>	69.7%	80.1%	90.8%	94.4%
σ	7.5%	3.5%	1.7%	0.9%

Table 7.2: The mirror symmetry problem in 30 dimensions. The training set consisted of m examples (half of them positive examples) generated uniformly on the hypercube. The generalization ability (Gen) was tested on 4000 examples (half of them positive), again generated uniformly. Each value represent an average over 20 different tests and σ is the standard deviation of the generalization.

7.6.3 Real Data Sets

We have tested our greedy heuristic for learning NDLs (section 7.5) on data sets taken from a collection distributed by the machine learning group of the University of California at Irvine[7]. The simulation results of our algorithm on these data are summarized in table 7.3. We have included, for comparison, the performance of C4 on these data sets, as reported recently by Holte [49]. C4 is a "state of the art" tree-induction algorithm [84] capable of producing very complex decision rules. Also indicated is the "default accuracy" one has when classifying all the testing examples according to the class containing the largest number of examples. We now comment our results:

CH: *Chess End-Game.* This data set was originally generated and described by Alen Shapiro [95]. The goal is to determine whether or not a given chess board configuration is a winning position for the white player. The white player has a king and rook whereas the black player has a king and pawn where the pawn is located on "a2" (just ready to be promoted to a Queen). The input consists of 35 binary features and one ternary feature, describing a board position of a chess end-game. As described in Shapiro's book[95], these features have been explicitly engineered for C4 by a chess expert working with a version of C4

[7]Contact person: Pat Murphy (pmurphy@ics.uci.edu). The data sets were kindly provided to us by Robert Holte (University of Ottawa).

built specially for this purpose. In view of the importance of representations, it is surprising to see that our algorithm achieves a 95% generalization score on this "C4-hand-crafted data".

G2: *Glass identification.* Here the goal is to determine if a given piece of glass is "float processed" or "non-float-processed". The study of classification of types of glass was motivated by criminological investigation. At the scene of the crime, the glass left can be used as evidence...if it is correctly identified! The input vector consists of 9 continuous valued attributes where each attribute indicates the concentration of a given element (Mg, Na, Al...) and one attribute indicates the refractive index. The performance of our greedy method was only slightly better (but not significantly) than C4. Although we think that this kind of problem is well suited for our approach (all attributes are continuous), the number of examples is not large enough to achieve higher accuracy.

IR *Iris data set.*This is perhaps the best known database in the pattern recognition literature [27]. The data set contains 3 classes of 50 instances each. The classes refer to types of iris plant: virginica, versicolor or setosa. One class (setosa) is linearly separable from the other two; the latter are not linearly separable from each other. The four inputs are continuous valued and correspond to the length and width of the sepal and petal. The present algorithm achieved good accuracy on this set; it is slightly better (but not significantly) than C4.

V0-V1: *United States Congressional Voting Records Database.* The V0 data set includes votes for each of the U.S. House of Representatives Congressmen on 16 key votes (Salvador aid, aid to Nicaraguan contras, education spending, MX-missiles,...). The result of each vote is expressed as a ternary attribute: yea, nay or "?" where this last value is taken when a congressmen did not vote, voted present, or voted present to avoid conflict of interest. This is a two-class problem since a congressmen is either Democrat or Republican. The V1 data set is identical to V0 except that the most informative attribute (physician fee-freeze) has been deleted, which makes the problem harder. The performance of our algorithm for these sets is slightly less (but not significantly) than that of C4.

Hence, the greedy method for constructing NDL's can handle with success these real data sets with roughly the same level of performance as C4. We must mention however that the version of C4 used in these tests incorporates pruning. We could probably also

Dataset	m	n	Def-acc	<Gen> of C4	<Gen>	$< k >$
CH	3196	36	52.2%	99.2% ± 0.3	95.2% ± 0.5%	4.0
G2	163	9	53.4%	74.3% ± 6.6	76.4% ± 6.7%	4.7
IR	150	4	33.3%	93.8% ± 3.0	95.1% ± 6.3%	3.4
V0	435	16	61.4%	95.6% ± 1.3	92.0% ± 2.8%	2.0
V1	435	15	61.4%	89.4% ± 2.5	87.3% ± 2.3%	3.4

Table 7.3: The simulation results of the algorithm for learning NDLs on real data sets. Columns from left to right: the dataset name, the total number of examples in the dataset, the number of inputs, the default accuracy, the average generalization of C4, the average generalization of the present algorithm, and the average number of halfspaces found by by it. For both C4 and the present algorithm, the training set consists of the 2/3 of the m examples and the test set consists of the remaining 1/3. Our results are the average over 20 trials.

improve our results by pruning the NDL and by using a cross-validation data set to test each new halfspace found during the learning process. Moreover, once a halfspace is found, we could "fine tune"it with a quadratic programming algorithm so as to find a hyperplane with maximal stability. In short, this approach is flexible enough to incorporate many engineering tricks that have proven useful in the past.

7.7 Summary

By formulating the problem of halfspace intersections as a set covering problem, we were able to bring it down to the much simpler, yet very difficult, sub-problem: given a set of non linearly separable examples, find the largest linearly separable subset of it. Short of solving the credit assignment problem, the only hope for PAC learning halfspace intersections is to find a good approximation algorithm for this NP-hard sub-problem.

The approximation algorithm given in this chapter seems to work well in practice. At the single neuron level, it outperforms the pocket algorithm used by many constructive neural algorithms on both linearly and non linearly separable functions. Whereas one can always think of malicious situations where the approximation algorithm will do poorly, the numerical results are very encouraging and indicate that this will occur very rarely in practice. The problem of whether or not there exists an approximation algorithm with a performance guarantee, in the worst case, is still open.

The greedy method that incorporates this approximation algorithm seems able to learn halfspace intersections under the uniform distribution of examples up to 50 dimensions. To our knowledge, the tests reported here go beyond any in the literature in terms of testing generalization by a greedy method in high dimensions. Again, the question of the PAC learnabililty of halfspace intersections is still open. Note that any algorithm that yields a number of halfspaces bounded by $h^{\beta}m^{\alpha}$, for $\alpha < 1$ and $\beta \geq 1$, will satisfy the Occam algorithm criteria (section 6.3).

We extended the greedy method to a larger and richer class of concepts, namely the class of neural decision lists. Again the approach seems to work well. Our results on real-data problems suggest that concepts like halfspace intersections (unions) and NDLs are not only important from the theoretical point of view, but also useful in practice.

7.8 From Neural Decision Lists to Cascade Feedforward Nets

Here we give the reduction from a NDL to a cascade FFN (fig. 7.2). Let

$$(H_1, \tau_1), (H_2, \tau_2), ..., (H_h, \tau_h)$$

be a NDL. We construct the cascade FFN as follows. To each halfspace H_k in the NDL corresponds a hidden unit with the same weight vector, call it $\mathbf{w}_k$, and the same bias, call it $w_{k,0}$. Our goal is to get a set of internal representations of the type:

$$-1, -1, \cdots, -1, +1, -1, \cdots, -1.$$

To achieve that, we connect hidden unit k to all hidden units $j > k$. If we denote by $v_{j,k}$ the weight from hidden unit k to hidden unit j, then the output of hidden unit j for an example $\mathbf{x}$, denoted S_j, is given by:

$$S_j = sgn\left(\sum_{i=1}^{n} w_{j,i}x_i + w_{j,0} + \sum_{k=1}^{j-1} v_{j,k}S_k\right), \tag{7.1}$$

where $sgn(a) = +1$ if $a > 0$, and -1 otherwise. The cascade architecture is reflected, in this equation, by the fact that hidden unit j is updated only after all hidden units $k < j$ have been updated.

It is easy to check that the following choice yields the desired set of internal representations:

$$v_{i,j} = -\sum_{k=0}^{n} |w_{i,k}| - 1, \quad \text{for } j = 1, \cdots h \quad \text{and} \quad i = j+1, \cdots h, \tag{7.2}$$

$$w_{i,0} \rightarrow w_{i,0} + \sum_{j=1}^{i-1} v_{i,j} \quad \text{for } i = 1, \cdots h \tag{7.3}$$

where h denotes the number of hidden units or, equivalently, the number of halfspaces in the NDL. Finally, if u_j is the weight connecting hidden unit j to the output and u_0 is output unit's bias, we set

$$u_j = \tau_j, \quad \text{for } j = 1, \cdots h, \tag{7.4}$$

$$u_0 = \sum_{j=1}^{h} \tau_j. \tag{7.5}$$

It is easy to check that the cascade FFN and the NDL agree on the classification of any example $\mathbf{x}$.

A different type of reduction, from NDLs to FFNs, is given in chapter 9.

Chapter 8

Learning Single Binary Perceptrons On Product Distributions

8.1 Introduction

The study of neural networks with binary weights is well motivated from both the theoretical and practical points of view. First, because the number of possible states in the weight space of a binary network is finite, its properties may differ drastically from those of a network with continuous weights [39, 94]. Second, the hardware realization of binary networks may prove simpler.

Although networks with binary weights have been the subject of intense analysis from the *capacity* point of view [7, 61, 62, 107], the question of the *learnability* of these networks remains largely unanswered. The reason for this state of affairs lies perhaps in the apparent strength of the following distribution-free result [81]: learning perceptrons with binary weights is equivalent to Integer Programming and so, it is an NP-Complete problem. However, this result does not rule out the possibility that this class of functions is learnable under some *reasonable distributions*.

In this chapter, we take a close look at this possibility. In particular, we investigate, within the distribution-specific PAC model, the learnability of single perceptrons with binary weights and arbitrary thresholds under the family of *product distributions*. A distribution of examples is a product distribution if the setting of each input variable is independent

of the settings of the other variables. The result of this investigation is a polynomial time algorithm that PAC learns binary perceptrons under any product distribution of examples. More specifically, the sample complexity of the algorithm is of $O((n/\epsilon)^4 \ln(n/\delta))$, and its running time is linear in the number of training examples. We note here that the algorithm produces hypotheses that are not necessarily consistent with all the training examples, but that nonetheless have very good generalization ability. Those types of algorithms are called "inconsistent algorithms" [72].

How does our algorithm relate to the learning rules proposed previously for learning binary perceptrons? We show that, under the *uniform distribution* and for binary perceptrons with zero thresholds, our algorithm reduces to the well known *clipped Hebb rule* [61] (called the majority rule in [107]).

8.2 Definitions

Let the input space $\mathcal{X}^n$ be the set $\{-1,+1\}^n$. A *perceptron* g on $\mathbf{x} = (x_1, x_2, ..., x_n)$ is specified by a vector of n *weight* values w_i and a single *threshold* value θ. For an input vector $\mathbf{x} \in I^n$, we have:

$$g(\mathbf{x}) = \begin{cases} +1 & \text{if } \sum_{i=1}^{i=n} w_i x_i > \theta \\ -1 & \text{if } \sum_{i=1}^{i=n} w_i x_i \leq \theta \end{cases} \tag{8.1}$$

A perceptron is said to be *positive* (or monotone) if $w_i \geq 0$ for $i = 1, ..., n$.

We are interested in the case where the weights are binary valued (± 1). We assume, without loss of generality (w.l.o.g.), that θ is integer and $-n - 1 \leq \theta \leq n$.

An *example* is an input-output pair $< \mathbf{x}, g(\mathbf{x}) >$. A *sample* is a set of examples. The examples are assumed to be generated randomly according to some unknown probability distribution D which can be any member of the family $\mathcal{D}$ of all *product distributions*. Distribution D belongs to $\mathcal{D}$ if and only if the setting of each input variable x_i is chosen *independently* of the settings of the other variables. The *uniform distribution*, where each x_i is set independently to ± 1 with probability $1/2$, is a member of $\mathcal{D}$.

We denote by $P(A)$ the probability of an event A and by $\hat{P}(A)$ its empirical estimate

based on a given finite sample. We denote by $P(A|B)$ the conditional probability of an event A given the fact that event B has been observed. All probabilities are taken with respect to the product distribution D on $\mathcal{X}^n$.

The *influence* of a variable x_i, denoted $Inf(x_i)$, is defined as

$$\begin{aligned} Inf(x_i) &\stackrel{\text{def}}{=} P(g=+1|x_i=+1) - P(g=+1|x_i=-1) \\ &- P(g=-1|x_i=+1) + P(g=-1|x_i=-1) \end{aligned} \tag{8.2}$$

Intuitively, the influence of a variable is positive (negative) if its weight is positive (negative).

8.3 PAC Learning Single Binary Perceptrons

8.3.1 The Learning Model

We adopt the distribution specific setting of the PAC learning model (definition 6.4). Here the methodology is to draw, according to D, a sample of a certain size labeled according to the unknown target perceptron g and then to find a "good" approximation g' of g. The error of the hypothesis perceptron g', with respect to the target g, is defined to be $P(g' \neq g) = P(g'(\mathbf{x}) \neq g(\mathbf{x}))$, where $\mathbf{x}$ is drawn according to the *same* distribution D used to generate the training sample.

An algorithm PAC learns from examples the class G of binary perceptrons, under a family $\mathcal{D}$ of distributions on $\mathcal{X}^n$, if for every $g \in G$, any $D \in \mathcal{D}$ and any $0 < \epsilon, \delta < 1$, the algorithm runs in time polynomial in (n, ϵ, δ) and outputs, with probability at least $1-\delta$, a hypothesis $g' \in G$ that makes an error at most ϵ with g.

8.3.2 The Learning Algorithm

We assume that the examples are generated according to a (unknown) product distribution D on $\{-1,+1\}^n$ and labeled according to a target binary perceptron g given by eq.(8.1). The learning algorithm proceeds in three steps:

1. Estimating, for each input variable x_i, the probability that it is set to $+1$. If this probability is too high (too low), the variable is set to $+1$ (-1). Note that setting a

variable to a given value is equivalent to neglecting this variable because any constant can be absorbed in the threshold.

2. Estimating the weight values (signs). This is done by estimating the *influence* of each variable.

3. Estimating the threshold value.

To simplify the analysis, we introduce the following notation. Let $\mathbf{y}$ be the vector whose components y_i are defined as

$$y_i = w_i \times x_i \tag{8.3}$$

Then eq.(8.1) can be written as

$$g(\mathbf{y}) = \begin{cases} +1 & \text{if } \sum_{i=1}^{i=n} y_i > \theta \\ -1 & \text{if } \sum_{i=1}^{i=n} y_i \leq \theta \end{cases} \tag{8.4}$$

In addition, we define $Inf(y_i)$ by:

$$\begin{aligned} Inf(y_i) &= P(g=+1|y_i=+1) - P(g=+1|y_i=-1) \\ &- P(g=-1|y_i=+1) + P(g=-1|y_i=-1) \end{aligned} \tag{8.5}$$

Note that if $D(\mathbf{x})$ is a product distribution on $\{-1,+1\}^n$, then so is $D(\mathbf{y})$.

Lemma 8.1 *Let g be a binary perceptron. Let x_i be a variable in g. Let $a \in \{-1,+1\}$. Let g' be a perceptron obtained from g by setting x_i to a. Then, if $P(x_i = -a) \leq \frac{\epsilon}{2n}$,*

$$P(g \neq g') \leq \frac{\epsilon}{2n}$$

Proof: follows directly from the fact that $P(g \neq g') \leq P(x_i = -a)$.□

Lemma 8.1 implies that we can neglect any variable x_i for which $P(x_i = \pm 1)$ is too high (too low). In what follows, we consider only variables that have not been neglected.

As we said earlier, intuition suggests that the influence of a variable is positive (negative) if its weight is positive (negative). The following lemma strengthens this intuition by showing that there is a measurable *gap* between the two cases. This gap will be used to estimate the weight values (signs).

Lemma 8.2 *Let g be a perceptron such that $P(g=+1)$, $P(g=-1) > \rho$, where $0 < \rho < 1$. Then for any product distribution D,*

$$Inf(x_i) \begin{cases} > \frac{\rho}{n+1} & \text{if } w_i = +1 \\ < -\frac{\rho}{n+1} & \text{if } w_i = -1 \end{cases}$$

Proof: We first note that from the definition of the influence and eq. 8.3 and 8.5, we can write:

$$Inf(x_i) = \begin{cases} +Inf(y_i) & \text{if } w_i = +1 \\ -Inf(y_i) & \text{if } w_i = -1 \end{cases}$$

We exploit the independence of the input variables to write

$$\begin{aligned} Inf(y_i) &= P(\sum_{j \neq i} y_j + 1 > \theta) - P(\sum_{j \neq i} y_j - 1 > \theta) \\ &\quad - P(\sum_{j \neq i} y_j + 1 \leq \theta) + P(\sum_{j \neq i} y_j - 1 \leq \theta) \\ &= 2P(\sum_{j \neq i} y_j = \theta) + 2P(\sum_{j \neq i} y_j = \theta + 1) \end{aligned} \tag{8.6}$$

One can also write

$$\begin{aligned} P(g=+1) &= P(\sum_j y_j > \theta) \\ &= P(y_i = +1) \times P(\sum_{j \neq i} y_j + 1 > \theta) + P(y_i = -1) \times P(\sum_{j \neq i} y_j - 1 > \theta) \\ &\leq P(\sum_{j \neq i} y_j + 1 > \theta) = \sum_{r=\theta}^{n} P(\sum_{j \neq i} y_j = r) \end{aligned} \tag{8.7}$$

Likewise,

$$\begin{aligned} P(g=-1) &= P(\sum_j y_j \leq \theta) \\ &= P(y_i = 1) \times P(\sum_{j \neq i} y_j + 1 \leq \theta) + P(y_i = -1) \times P(\sum_{j \neq i} y_j - 1 \leq \theta) \\ &\leq P(\sum_{j \neq i} y_j - 1 \leq \theta) = \sum_{r=-n}^{r=\theta+1} P(\sum_{j \neq i} y_j = r) \end{aligned} \tag{8.8}$$

Let $p(r)$ denote $P(\sum_{j \neq i} y_j = r)$. From the properties of the generating function associated with product distributions, it is well known [52, 67] that $p(r)$ is always *unimodal* and reaches its maximum at a given value of r, say r_{max}. We distinguish two cases:

$\underline{\theta \geq r_{max}}$: in this case, using eq (8.7)

$$\begin{aligned} P(g=+1) &\leq \sum_{r=\theta}^{n} P(\sum_{j\neq i} y_j = r) \\ &\leq (n-\theta+1) \times p(\theta) \end{aligned} \tag{8.9}$$

Using eq. (8.6) and eq. (8.9), it easy to see that

$$Inf(y_i) \geq \frac{2P(g=1)}{n-\theta+1} > \frac{\rho}{n+1}$$

$\underline{\theta \leq r_{max}-1}$: in this case, using eq (8.8)

$$\begin{aligned} P(g=-1) &\leq \sum_{r=-n}^{r=\theta+1} P(\sum_{j\neq i} y_j = r) \\ &\leq (n+\theta+2) \times p(\theta+1) \end{aligned} \tag{8.10}$$

Using eq. (8.6) and eq. (8.10), it easy to see that

$$Inf(y_i) \geq \frac{2P(g=-1)}{n+\theta+2} > \frac{\rho}{n+1}$$

□

So, if we estimate $Inf(x_i)$ to within a precision better than the gap established in lemma 8.2, we can determine the value of w_i with enough confidence. Note that if θ is too large (small), most of the examples will be negative (positive). In this case, the influence of any input variable is very weak. This is the reason we require $P(g=+1)$, $P(g=-1) > \rho$.

The weight values obtained in the previous step define the weight vector of our hypothesis perceptron g'. The next step is to estimate an appropriate threshold for g', using these weight values. For that, we appeal to the following lemma.

Lemma 8.3 *Let g be a perceptron with a threshold θ. Let g' be a perceptron obtained from g by substituting r for θ. Then, if $r \leq \theta$,*

$$P(g \neq g') \leq 1 - P(g=+1|g'=+1)$$

Proof:

$$\begin{aligned} P(g \neq g') &\leq P(g=-1|g'=+1) + P(g=+1|g'=-1) \\ &= 1 - P(g=+1|g'=+1) + P(g=+1|g'=-1) \\ &= 1 - P(g=+1|g'=+1) \end{aligned}$$

The last equality follows from the fact that $P(g = +1|g' = -1) = 0$ for $r \leq \theta$.□

So, if we estimate $P(g = +1|g' = +1)$ for $r = -n-1, -n, -n+1, ...$ and then choose as a threshold for g' the *least* r for which $P(g = +1|g' = +1) \geq (1-\epsilon)$, we are guaranteed to have $P(g \neq g') \leq \epsilon$. Obviously, such an r exists and is always $\leq \theta$ because $P(g = +1|g' = +1) = 1$ for $r = \theta$.

A sketch of the algorithm for learning single binary perceptrons is given in fig. 8.1.

Theorem 8.4 *The class of binary perceptrons is PAC learnable under the family of product distributions.*

Proof: Using Chernoff bounds [40], one can show that a sample of size $m = \frac{[160n(n+1)]^2}{\epsilon^4} \ln \frac{32n}{\delta}$ is sufficient to ensure that:

- $|\hat{P}(g = a) - P(g = a)| \leq \epsilon/4$ with confidence at least $1 - \delta/2$.
- $|\hat{P}(x_i = a) - P(x_i = a)| \leq \epsilon/4n$ with confidence at least $1 - \delta/4n$.
- $|\hat{Inf}(x_i) - Inf(x_i)| \leq \frac{\epsilon}{4(n+1)}$ with confidence at least $1 - \delta/8n$.
- $|\hat{P}(g = +1|g' = +1) - P(g = +1|g' = +1)| \leq \epsilon/4$ with confidence at least $1 - \delta/16n$.

Combining all these factors, it is easy to show that the hypothesis g' returned by the algorithm will make an error at most ϵ with the target g, with confidence at least $1 - \delta$.

Since it takes m units of time to estimate a conditional probability using a sample of size m, the running time of the algorithm will be of $O(m \times n)$.□

8.4 Reduction to the Clipped Hebb Rule

The perceptron with binary weights and zero-threshold has been extensively studied by many authors [62, 61, 107]. All these studies assume a uniform distribution of examples and zero threshold. So we come to ask how the algorithm of fig. 8.1 relates to the learning rules proposed previously.

To answer this, let us first rewrite the influence of a variable as:

$$Inf(x_i) = \frac{P(x_i = +1|g = +1)}{P(x_i = +1)} - \frac{P(x_i = -1|g = +1)}{P(x_i = +1)} + \frac{P(x_i = -1|g = -1)}{P(x_i = -1)} - \frac{P(x_i = +1|g = -1)}{P(x_i = +1)}$$

Algorithm *LEARN-BINARY-PERCEPTRON*(n,ϵ,δ)

Parameters: n is the number of input variables, ϵ is the accuracy parameter and δ is the confidence parameter.

Output: a binary perceptron g' defined by a weight vector $(w_1,\ldots,w_n)$ and a threshold r.

Description:

1. Call $m = \frac{[160n(n+1)]^2}{\epsilon^4} \ln \frac{32n}{\delta}$ examples. This sample will be used to estimate the different probabilities. Initialize g' to the constant perceptron -1.
2. (Are most examples positive?) If $\hat{P}(g=+1) \geq (1-\frac{\epsilon}{4})$ then set $g'=1$ and return g'.
3. (Are most examples negative?) If $\hat{P}(g=+1) \leq \frac{\epsilon}{4}$ then set $g'=-1$ and return g'.
4. Set $\rho = \frac{\epsilon}{2}$.
5. (Is $P(x_i=+1)$ too low or too high ?) For each input variable x_i:
   (a) Estimate $P(x_i=+1)$.
   (b) If $\hat{P}(x_i=+1) \leq \frac{\epsilon}{4n}$ or $1-\hat{P}(x_i=+1) \leq \frac{\epsilon}{4n}$, neglect this variable.
6. (Determine the weight values) For each input variable x_i:
   (a) If $\hat{Inf}(x_i) > \frac{1}{2}\frac{\rho}{n+1}$, set $w_i=1$.
   (b) Else if $\hat{Inf}(x_i) < -\frac{1}{2}\frac{\rho}{n+1}$, set $w_i=-1$.
   (c) Else set $w_i=0$ (x_i is not an influential variable).
7. (Estimating the threshold) Initialize r (the threshold of g') to $-(n+1)$.
   (a) Estimate $P(g=+1|g'=+1)$.
   (b) If $\hat{P}(g=+1|g'=+1) > 1-\frac{1}{4}\epsilon$, go to step 8.
   (c) $r=r+1$. Go to step 7a.
8. Return g' (that is $(w_1,\ldots,w_n;r)$).

Figure 8.1: An algorithm for learning single binary perceptrons on product distributions.

and observe that under the uniform distribution, $P(x_i = +1) = P(x_i = -1)$. Next, we notice that in the algorithm of fig. 8.1, each weight w_i is basically assigned to the sign of $\hat{Inf}(x_i)$. Hence apart from ϵ and δ, the algorithm can be summarized by the following rule:

$$\begin{aligned} w_i &= \text{sgn}\left(\hat{Inf}(x_i)\right) \\ &= \text{sgn}\left(\sum_{\nu} g(\mathbf{x}^{\nu}) x_i^{\nu}\right) \end{aligned} \tag{8.11}$$

where $\text{sgn}(x) = +1$ for $x > 0$ and -1 otherwise and x_i^{ν} denotes the ith component of the νth training example.

Equation 8.11 is simply the well known *clipped Hebb rule* [61], also called the *majority rule* in [107]. Since this rule is just the restriction of the learning algorithm of fig. 8.1 for uniform distributions, theorem 8.4 has the following corollary:

Corollary 8.5 *The clipped Hebb rule PAC learns the class of binary perceptrons with zero thresholds under the uniform distribution.*

The clipped Hebb rule will be the subject of our average case analysis later in this thesis.

8.5 Conclusion

We have presented a simple algorithm for PAC learning binary perceptrons on product distributions. The sample complexity of the algorithm is $O((n/\epsilon)^4 \ln(n/\delta))$ and its running time $O(m \times n)$. The conclusion that can be drawn from this chapter is that whereas learning under *arbitrary* distributions is generally hard, learning under some reasonable distributions may be quite easy.

We will come back to single binary perceptrons later in this thesis, when we look at the problem of learning from the *average case* point of view.

Chapter 9

On Learning Nonoverlapping Perceptron Networks with Binary Weights on the Uniform Distribution

9.1 Introduction

In this chapter, we investigate the problem of learning the class of *"nonoverlapping"* perceptron networks. These are loop-free neural nets in which each node, including the input units, has only one outgoing non-zero weight (fig. 9.1 a). This class of representations includes as a subclass nonoverlapping multilayer networks (fig.9.1 b) and nonoverlapping cascade networks. Such networks, in which each node has fan-out 1, are also referred to in the literature as *read-once formulas*. Our work is partly motivated by recent positive results for learning other classes of read-once formulas on specific distributions [44, 58, 79, 90]. In the terminology of that literature, nonoverlapping perceptron networks are read-once formulas (or μ formulas) over the basis of perceptrons.

One can think of this type of architecture as a network of *"decoupled"* perceptrons, which in terms of architecture complexity lies somewhere between the single perceptron and the traditional feedforward neural net. As such, studying this restricted class may shed some light on the gap that exists, in terms of computational complexity, between training single

perceptrons, which can be done in polynomial time [56], and training feedforward nets, which has been proven to be intrinsically hard [14, 55] (even if the algorithm is allowed to represent its hypothesis in ways other than as a network [4, 60]). Of fundamental importance is the question of whether or not removing the overlap between the receptive fields of the nodes makes the learning problem easier.

Standard techniques [58] show that the problem of learning nonoverlapping networks from examples drawn according to an arbitrary distribution is no easier than the problem where the input variables may have an arbitrary number of outgoing weights. Kearns and Valiant have shown that this (seemingly) more general problem is intractable [60]. Thus to achieve interesting results we must consider a slightly easier learning model. For that, we restrict ourselves to the case where the distribution of examples is uniform, and investigate the problem of learning concepts that can be represented as *simple combinations* of *nonoverlapping* perceptrons with binary weights and arbitrary thresholds.

More specifically, we investigate, within the distribution-specific PAC model, the learnability of the following concepts when the weights are *binary* valued (± 1) (see section 9.2 for precise definitions):

- *Nonoverlapping perceptron unions*, *i.e.* an OR function of binary perceptrons where each input unit is connected to one and only one perceptron (fig. 9.2) [1].
- *Nonoverlapping perceptron decision lists* (fig. 9.3a).
- *Generalized nonoverlapping perceptron decision lists* (fig. 9.3b).

Note that in contrast to most neural network learning algorithms, we do not assume that the architecture of the network is known in advance. Rather, it is the task of the algorithm to find both the architecture of the net and the weight values necessary to represent the function to be learned.

We give polynomial time algorithms for learning these special classes of nonoverlapping perceptron networks. The algorithms work by estimating various statistical quantities that yield enough information to infer, with high probability, the target concept. Because our

[1] The intersection is simply the complement of the union and can be treated similarly.

algorithms are statistical in nature, they are robust against a large amount of random classification noise. From another point of view, our results prove the learnability (under the uniform distribution) of the above classes in the new model of *learning from statistical queries* introduced by Kearns [57]. In that model, the standard oracle EX(f,D) providing the learner with examples is replaced by a weaker oracle STAT(f,D) providing accurate estimates of probabilities over the sample space [2].

The results of this chapter imply, as particular cases, simpler algorithms for learning μ-DNF [79] and read-once boolean formulas over the basis {AND,OR,NOT} [90].

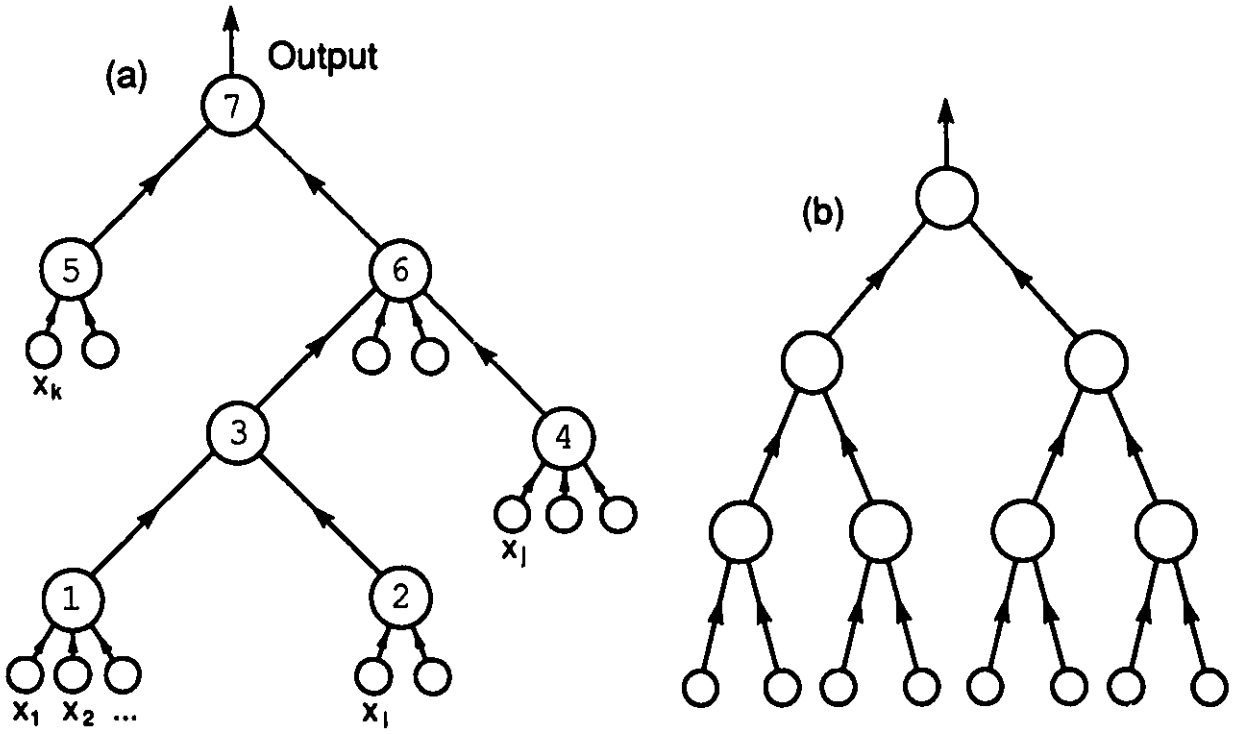

Figure 9.1: (a) Architecture of a nonoverlapping perceptron network. Note that each node, including the input variables but excluding the output, has only one outgoing connection. (b) A nonoverlapping layered network.

[2]These remarks apply also to the results of chapter 8.

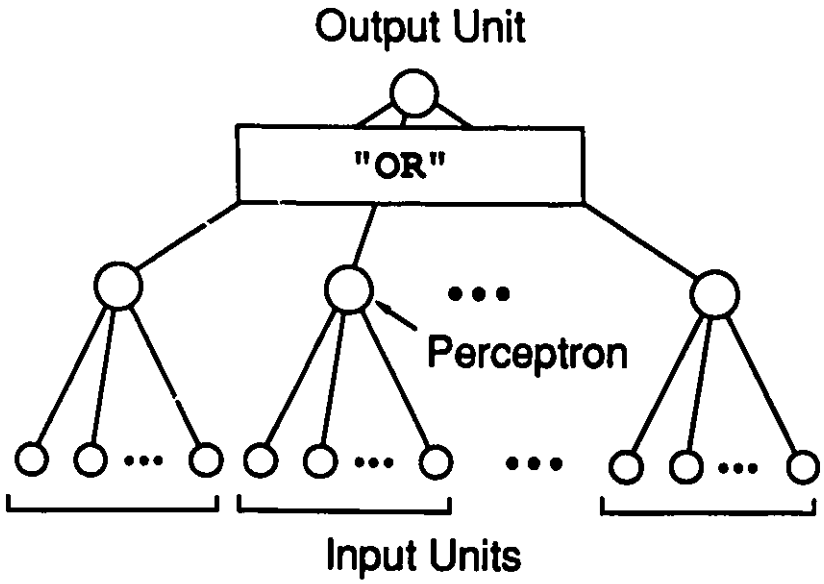

Figure 9.2: A two-layer network representing a μ-perceptron union. Note that each input unit is connected to one and only one perceptron (hidden unit). The output unit computes an OR function.

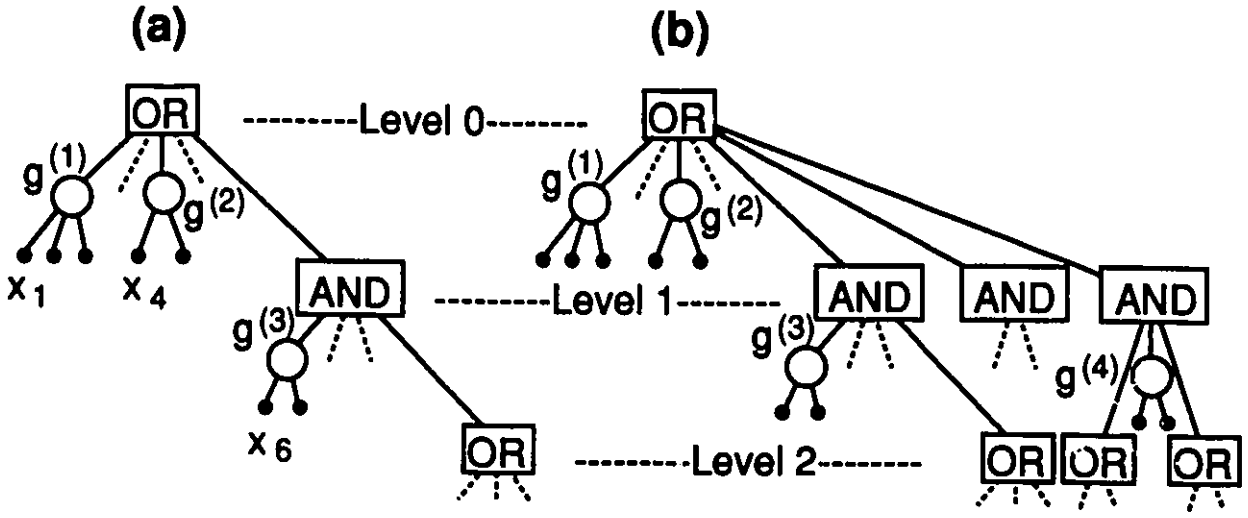

Figure 9.3: A network representing a) a perceptron decision list, b) a generalized perceptron decision list.

9.2 Definitions

Let $X = \{x_1, x_2..., x_n\}$ be the set of the n input variables and let the input space $\mathcal{X}^n$ be the set $\{0,1\}^n$. We assume throughout this chapter that the distribution D generating the examples is uniform on $\mathcal{X}^n$.

We denote by $P(A)$ the probability of an event A, and by $P(A|B)$ the conditional probability of an event A given the fact that event B has been observed. All probabilities are taken with respect to D.

Let f be a boolean function defined on X. The *influence* of a variable x_i on f, denoted $Inf(x_i)$, is defined as [3]:

$$Inf(x_i) \stackrel{\text{def}}{=} P(f = 1|x_i = 1) - P(f = 1|x_i = 0).$$

The *correlation* of a variable x_j with x_i, denoted $C(x_i, x_j)$, is defined as:

$$C(x_i, x_j) \stackrel{\text{def}}{=} \frac{P(f = 1|x_i = x_j = 1) - P(f = 1|x_i = 1, x_j = 0)}{P(f = 1|x_j = 1) - P(f = 1|x_j = 0)}.$$

Intuitively, $C(x_i, x_j)$ reflects the effect of setting x_i to 1 on the influence of x_j. Note that, in general, $C(x_i, x_j) \neq C(x_j, x_i)$.

As usual, a perceptron g on X is specified by a vector of n weights w_i and a single threshold θ. For $\mathbf{x} = (x_1, x_2, ..., x_n) \in I^n$, we have:

$$g(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_{i=1}^{i=n} w_i x_i \geq \theta \\ 0 & \text{if } \sum_{i=1}^{i=n} w_i x_i < \theta \end{cases} \tag{9.1}$$

A perceptron is said to be monotone (or positive) if $w_i \geq 0$ for $i = 1, ..., n$.

We are interested in the learnability of simple neural concepts built from nonoverlapping perceptrons with binary valued weights ($w_i = \pm 1$). For later reference, we define the concepts we shall consider along with some notation and terminology.

[3] This definition is equivalent to the one given in chapter 8.

Nonoverlapping perceptron unions

A function f is said to be a *perceptron union* if it can be written as a disjunction of perceptrons:

$$f = g^{(1)} \vee g^{(2)} \vee ... \vee g^{(r)} \quad 1 \leq r \leq n \tag{9.2}$$

If the perceptrons do not share any variables, f is said to be a *nonoverlapping* (or μ) *perceptron union* (hereafter μ-PU) (fig. 9.2).

The *parent* of a variable is the perceptron to which the variable is an immediate input.

We say that two variables are *siblings* if they share a common parent (perceptron), and not siblings otherwise.

Nonoverlapping perceptron decision lists and their generalization

We generalize the notion of *decision lists*, introduced by Rivest [85], to *perceptron decision lists* [4] (hereafter PDL). A PDL is a list $\mathcal{L}$ of pairs: $(g^{(1)}, v^{(1)}), \ldots, (g^{(i)}, v^{(i)}), \ldots, (g^{(r)}, v^{(r)})$, where each $g^{(i)}$ is a perceptron and $v^{(i)}$ is a value in $\{0, 1\}$. The last perceptron $g^{(r)}$ is the constant function $+1$. This defines a function $f : I^n \rightarrow \{0, 1\}$ as follows: for any $\mathbf{x}$, $f(\mathbf{x})$ is defined to be equal to $v^{(j)}$ where j is the least index for which $g^{(j)}(\mathbf{x}) = 1$. Compared to Rivest's decision lists, PDLs have the same structure but the complexity of the decision allowed at each node is greater.

A PDL can be converted to an equivalent feedforward neural net in an obvious manner. The result is a network of *alternating levels* of disjunction (OR) and conjunction (AND) of perceptrons (fig. 9.3a).

A possible generalization of PDL is to allow multiple AND/OR nodes at each level (fig. 9.3b). Such networks represent arbitrary boolean formulas over the basis {AND,OR} whose inputs are perceptrons. We call such networks (functions) *generalized* PDL.

If each variable appears at most once, the PDL is referred to as *nonoverlapping* (or μ) *perceptron decision list* (hereafter μ-PDL). Similarly, we refer to generalized PDLs with the same property as *generalized* μ-PDLs. The equivalent network can be then viewed as a rooted tree. The root is the *output node* and each leaf in the tree is labeled with an input

[4] We referred to them as neural decision lists in chapter 7.

variable in a way such that no variable appears on more than one leaf. We will refer to the AND/OR nodes as *gates*.

We will be interested in learning μ-PDL and generalized μ-PDLs with binary weights. To simplify the analysis, we assume w.l.o.g. that the input variables feed the gates only through perceptrons (maybe single-variable perceptrons) and that each gate has, as input, at least one multi-variable perceptron or another gate (otherwise, it is a perceptron).

The *parent* of a variable is the perceptron to which the variable is an immediate input. We say that a perceptron g (or a gate Γ) is an *uncle* of a variable x_i if g (respec., Γ) is an immediate input to a gate fed by x_i, but g (respec., Γ) is not itself fed by x_i.

The *depth* of a gate Γ (or a variable x_i) is the number of gates on the path from Γ (respec., x_i) to the output gate. We say that x_i is a *zero-level* variable *with respect to* a gate Γ if x_i's parent is an immediate input to Γ.

The *least common ancestor* of two variables x_i and x_j, denoted $\mathrm{lca}(x_i,x_j)$, is the deepest gate fed by both x_i and x_j. We say that two variables are *siblings* if they share a common parent (perceptron), and not siblings otherwise.

Learning model

In what follows, we adopt the PAC learning model [103, 15] specified to the uniform distribution. Here the methodology is to draw a sample of a certain size labeled according to the unknown target function f and then to find a "good" approximation h of f. The error of the hypothesis function h, with respect to the target f, is defined to be $P(h \neq f) = P(h(\mathbf{x}) \neq f(\mathbf{x}))$, where $\mathbf{x}$ is distributed according to D. An algorithm learns from examples a target class F using an hypothesis class H under the distribution D on I^n, if for every $f \in F$, and any $0 < \epsilon, \delta < 1$, the algorithm runs in time polynomial in (n, ϵ, δ) and outputs an hypothesis $h \in H$ such that, with probability at least $1 - \delta$,

$$P(h \neq f) < \epsilon$$

In what follows, the hypothesis class H will be the same as the target class F.

9.3 The Learning Algorithms

As in the previous chapter, in order to simplify the analysis, we introduce the following notation. Let N be the number of negative weights in g and let $\mathbf{y}$ be defined as

$$y_i = \begin{cases} x_i & \text{if } w_i = 1 \\ 1 - x_i & \text{if } w_i = -1 \end{cases} \tag{9.3}$$

Then eq. 9.1 can be written as

$$g(\mathbf{y}) = \begin{cases} 1 & \text{if } \sum_{i=1}^{i=n} y_i \geq \Omega \\ 0 & \text{if } \sum_{i=1}^{i=n} y_i < \Omega \end{cases} \tag{9.4}$$

where the renormalized threshold Ω is related to the original threshold θ by: $\Omega = \theta + N$. We assume, w.l.o.g., that $1 \leq \Omega \leq n$. Note that $D(\mathbf{x}) = D(\mathbf{y})$.

We will make use of the following lemmas.

Lemma 9.1 *Let d and n be two integers. Then, if $\frac{n}{2} \leq d \leq n$,*

$$\sum_{i=d}^{n} \binom{n}{i} \leq \frac{1}{2} \binom{n}{d} \frac{1}{1-z}$$

where $z = \frac{1}{2}\frac{n+1}{d+1}$. And if $0 \leq d \leq \frac{n}{2}$,

$$\sum_{i=0}^{d} \binom{n}{i} \leq \frac{1}{2} \binom{n}{d} \frac{1}{1-z'}$$

where $z' = \frac{1}{2}\frac{n+1}{n-d+1}$.

Proof: follows directly from Badahur's expansion [5]. This lemma is used throughout this chapter to approximate binomial distributions.□

Lemma 9.2 *Let g be a perceptron with binary weights such that $P(g = 1)$, $P(g = 0) > \rho$, where $0 < \rho < 1$. Then,*

$$P(g(\mathbf{x}) = 1 | x_i = 1) - P(g(\mathbf{x}) = 1 | x_i = 0) \begin{cases} \geq \frac{\rho}{n+2} & \text{if } w_i = 1 \\ \leq -\frac{\rho}{n+2} & \text{if } w_i = -1 \end{cases}$$

Proof: We prove the lemma for $w_i = +1$. The case $w_i = -1$ follows by a similar argument. We exploit the equivalence between eq. 9.1 and eq. 9.4, and the fact that the distribution is uniform:

$$\begin{aligned} P(g(\mathbf{x}) = 1|x_i = 1) - P(g(\mathbf{x}) = 1|x_i = 0) &= P(g(\mathbf{y}) = 1|y_i = 1) - P(g(\mathbf{y}) = 1|y_i = 0) \\ &= P(\sum_{j \neq i} y_j = \Omega - 1) \\ &= \binom{n-1}{\Omega-1} / 2^{n-1} \\ &= 2\frac{\binom{n-1}{\Omega-1}}{\sum_{i=\Omega}^{n} \binom{n}{i}} P(g = 1) \qquad (9.5) \\ &= 2\frac{\binom{n-1}{\Omega-1}}{\sum_{i=0}^{\Omega-1} \binom{n}{i}} P(g = 0) \qquad (9.6) \end{aligned}$$

Eqs. 9.5 and 9.6 follow from the fact that, under the uniform distribution,

$$P(g = 1) = \sum_{i=\Omega}^{n} \binom{n}{i} / 2^n$$

and

$$P(g = 0) = \sum_{i=0}^{\Omega-1} \binom{n}{i} / 2^n$$

respectively.

Case 1: $\Omega \geq \frac{n}{2}$

Using eq. (9.5) and lemma 9.1, it is easy to see that

$$P(g(\mathbf{x}) = 1|x_i = 1) - P(g(\mathbf{x}) = 1|x_i = 0) \geq \frac{\binom{n-1}{\Omega-1}}{\frac{1}{2}\binom{n}{\Omega}} (1 - \frac{1}{2}\frac{n+1}{\Omega+1}) P(g = 1)$$

$$
\begin{aligned}
&= \frac{2\Omega}{n}(1 - \frac{1}{2}\frac{n+1}{\Omega+1})P(g=1) \\
&\geq \frac{\rho}{n+2}
\end{aligned}
$$

Case 2: $\Omega \leq \frac{n}{2}$

Using eq. (9.6) and lemma 9.1, it is again easy to see that

$$
\begin{aligned}
P(g(\mathbf{x})=1|x_i=1) - P(g(\mathbf{x})=1|x_i=0) &\geq \frac{\binom{n-1}{\Omega-1}}{\frac{1}{2}\binom{n}{\Omega-1}}(1 - \frac{1}{2}\frac{n+1}{n-\Omega+2})P(g=0) \\
&= \frac{2(n-\Omega+1)}{n}(1 - \frac{1}{2}\frac{n+1}{n-\Omega+2})P(g=0) \\
&\geq \frac{\rho}{n+2}
\end{aligned}
$$

□.

Lemma 9.3 *Let f be a μ-perceptron union. Let $g^{(a)}$ and $g^{(b)}$ be two perceptrons in f. Let m be the disjunction of all perceptrons in f except $g^{(a)}$ and $g^{(b)}$. Then*

$$
\begin{aligned}
P(f=1) &= (1 - P(m=1)) \times P(g^{(a)}=1) + (1 - P(m=1)) \times P(g^{(b)}=1) + P(m=1) \\
&- (1 - P(m=1)) \times P(g^{(b)}=1) \times P(g^{(a)}=1)
\end{aligned}
$$

Proof: follows from the inclusion-exclusion property and the fact that perceptrons in f do not share any variables.□

Lemma 9.4 *Let f be a μ-perceptron intersection. Let $g^{(a)}$ and $g^{(b)}$ be two perceptrons in f. Let m be the conjunction of all perceptrons in f except $g^{(a)}$ and $g^{(b)}$. Then*

$$
P(f=1) = P(m=1) \times P(g^{(b)}=1) \times P(g^{(a)}=1)
$$

9.3.1 Learning μ-Perceptron Unions

In this section, we describe a polynomial time algorithm for learning μ-PUs with binary weights from random examples drawn according to the uniform distribution D. Note that we do not assume that the architecture (connectivity and number of perceptrons) is known

in advance. Rather, it is the task of the learning algorithm to determine which variables are in a given perceptron.

Let us assume that the target function f is a μ-PU as in eq. 9.2. We shall assume w.l.o.g. that f is expressed with the maximum possible number of $g^{(i)}$'s, and has the minimum possible number of non-zero weights.

The learning algorithm proceeds in three steps:

1. In the first step, the algorithm determines the "relevant" input variables and the values (signs) of their weights. To achieve this, for each variable x_i, the algorithm estimates its *influence* $Inf(x_i)$ using examples drawn randomly according to D. We prove that if the variable x_i is relevant, then its influence is significantly greater or less than zero, depending on whether $w_i = 1$ or $w_i = -1$. If the influence is too small, the algorithm concludes that the variable is "irrelevant" and so, neglects it in later stages. Once the weights are estimated, the algorithm reduces the target function to a monotone μ-PDL by simply changing x_i to $1 - x_i$ whenever $w_i = -1$.

2. In the second step, the algorithm determines which variables are siblings, *i.e.* belong to the same perceptron. To do this, the algorithm estimates the correlation $c(x_i, x_j)$ of each ordered pair of variables using examples drawn randomly according to D. We show that these correlations contain enough information to infer the architecture of the network.

3. In the last step, the algorithm estimates the threshold value for each perceptron.

Intuition suggests that the influence of a variable is positive (negative) if its weight is positive (negative). The following lemma strengthens this intuition by showing that there is a measurable *gap* between the two cases. This gap will be used to estimate the weight values (signs).

Lemma 9.5 *Let f be a μ-perceptron union as in eq.9.2. Let $g^{(a)}$ be a perceptron in f and let $x_i \in g^{(a)}$. Assume that $P(f = 1) < 1-\gamma$ and $P(g = 1), P(g = 0) > \rho$ where $0 < \gamma, \rho < 1$. Then*

$$Inf(x_i) \begin{cases} > \frac{\gamma\rho}{n+2} & \text{if } w_i = 1 \\ < -\frac{\gamma\rho}{n+2} & \text{if } w_i = -1 \end{cases}$$

Proof: $\underline{w_i = +1}$

Let m be the disjunction of all perceptrons in f except g. Then, using lemma 9.3

$$\begin{aligned} Inf(x_i) &= (1 - P(m=1))(P(g=1|x_i=1) - (P(g=1|x_i=0)) \\ &> \gamma(P(g=1|x_i=1) - (P(g=1|x_i=0)) \quad (9.7) \\ &> \frac{\gamma\rho}{n+2} \quad (9.8) \end{aligned}$$

Inequality 9.7 follows from the fact that $1-P(m=1) > 1-P(f=1) > \gamma$, and inequality 9.8 follows from lemma 9.2. The case $w_i = -1$ can be treated similarly. Note that we can assume w.l.o.g. that $\gamma \geq \epsilon/2$ and $\rho \geq \epsilon/2n$, for otherwise we can neglect perceptron g without introducing much error. Under this assumption, the gap is wide enough to be estimated efficiently using a sample of polynomial size.□

Lemma 9.5 enables us to determine the weight values and reduce f to its monotone form. The next step is to determine which variables belong to the same perceptron. Starting with a variable, say x_i, the algorithm uses the correlation measure to decide whether or not another variable, say x_j, belongs to x_i's perceptron. We appeal to the following lemma where we assume that f is already reduced to its monotone form.

Lemma 9.6 *Let f be a μ-perceptron union (monotone form). Let $g^{(a)}$ be a perceptron in f. Let $x_i \in g^{(a)}$ and let x_j and x_k be two other influential variables in f. Then*

$$C(x_i,x_j) - C(x_i,x_k) = \begin{cases} 0 & \text{if } x_j \in g^{(a)} \text{ and } x_k \in g^{(a)} \\ 0 & \text{if } x_j \notin g^{(a)} \text{ and } x_k \notin g^{(a)} \\ \geq \frac{1}{n^2} & \text{if } x_j \in g^{(a)} \text{ and } x_k \notin g^{(a)} \end{cases}$$

Proof: It is easy to see that if $x_j, x_k \in g^{(a)}$, then $C(x_i,x_j) = C(x_i,x_k)$. Now, assume that $x_k \notin g^{(a)}$. Using the definition of correlation and lemma 9.3, one can show that

$$C(x_i,x_k) = C_I = \frac{1 - P(g^{(a)}=1|x_i=1)}{1 - P(g^{(a)}=1)} = 1 - \frac{P(g^{(a)}=1|x_i=1) - P(g^{(a)}=1)}{1 - P(g^{(a)}=1)} \quad (9.9)$$

which is independent of x_k. So, if $x_j \notin g^{(a)}$, then $C(x_i,x_j) = C(x_i,x_k)$. We are left with the case where $x_j \in g^{(a)}$ and $x_k \notin g^{(a)}$. In this case, $C(x_i,x_k)$ is given by eq. (9.9), and $C(x_i,x_j)$ is given by (again using lemma 9.3)

$$C(x_i,x_j) = C_{II} = \frac{P(g^{(a)}=1|x_i=x_j=1) - P(g^{(a)}=1|x_i=1,x_j=0)}{P(g^{(a)}=1|x_j=1) - P(g^{(a)}=1|x_j=0)} \quad (9.10)$$

Let p be the number of variables in $g^{(a)}$ and let v be its renormalized threshold. Combining eq. (9.9) and (9.10), we get after few manipulations

$$C(x_i,x_j) - C(x_i,x_k) = 2\frac{v-1}{p-1} + \frac{\binom{p-1}{v-1}}{\sum_{i=0}^{v-1}\binom{p}{i}} - 1 \tag{9.11}$$

Case 1: $p \geq v \geq \frac{p}{2}+1$

$$C(x_i,x_j) - C(x_i,x_k) \geq 2\frac{v-1}{p-1} - 1 \geq \frac{1}{p-1} \geq \frac{1}{n-1} \geq \frac{1}{n^2}$$

Case 2: $2 \leq v \leq \frac{p}{2}$

In this case, applying lemma 9.1 to $\sum_{i=0}^{v-1}\binom{p}{i}$ we get

$$\begin{aligned} C(x_i,x_j) - C(x_i,x_k) &\geq 2\frac{v-1}{p-1} + 2(\frac{p-v+1}{p})(1 - \frac{1}{2}\frac{p+1}{p-v+2}) - 1 \\ &\geq \frac{3p-1}{p^2(p-1)} \geq \frac{1}{p^2} \geq \frac{1}{n^2} \end{aligned} \tag{9.12}$$

Note that because we have assumed that f is expressed with the maximum possible number of $g^{(i)}$'s, the renormalized threshold v is different from 1 (in other words, $g^{(a)}$ is not itself an OR function).□

If we estimate the correlations to within a sufficient precision, the correlation gap, established in lemma 9.6, enables us to decide which variables are in the same perceptron.

The last step is to estimate the bias of each perceptron. Let g be a perceptron in f and let g' be the perceptron obtained from g by setting its renormalized bias to r. Estimating g's bias may be done by simply estimating $P(f=1|g'=1)$ for different values of the renormalized threshold, $r=1,2,3,...$, and choosing the least r such that $P(f=1|g'=1) \geq (1-\epsilon/n)$ (see lemma 8.3 and step 7 in fig. 8.1).

Theorem 9.7 *The class of μ-perceptron unions with binary weights are PAC learnable under the uniform distribution.*

Proof: By using the standard Chernoff bounds analysis [40], one can show that a sample polynomial in (n,ϵ,δ) is sufficient to ensure that the different probabilities are estimated to within a sufficient precision. Because each perceptron makes an error at most ϵ/n, their union will make an error at most ϵ. Finally, the algorithm runs in time polynomial in (n,ϵ,δ).□

9.3.2 Learning μ-Perceptron Decision Lists

In this section, we describe a polynomial time algorithm for learning μ-PDLs with binary weights from random examples drawn according to the uniform distribution D. Actually, what we will be learning is the equivalent network of the μ-PDL (fig. 9.3a). The next section will provide the necessary additional processing to handle generalized μ-PDLs.

We assume w.l.o.g. that the target network contains no two successive AND or two successive OR gates, as such nodes can always be merged. We also assume w.l.o.g. that the target network has the maximum possible number of perceptrons, the minimum possible number of weights, and it contains negative weights only for those inputs that lead directly from input variables.

The learning algorithm proceeds in three steps:

1. In the first step, the algorithm determines the "relevant" input variables and the values (signs) of their weights. To achieve this, for each variable x_i, the algorithm estimates its *influence* $Inf(x_i)$ using examples drawn randomly according to D. We prove that if the variable x_i is relevant, then its influence is significantly greater or less than zero, depending on whether $w_i = 1$ or $w_i = -1$. If the influence is too small, the algorithm concludes that the variable is "irrelevant" and so, neglects it in later stages. Once the weights are estimated, the algorithm reduces the target function to a monotone μ-PDL by simply changing x_i to $1 - x_i$ whenever $w_i = -1$.

2. In the second step, the algorithm infers the architecture of the network (fig. 9.3a), *i.e.* determines which variables appear in a given level, and among those, which are siblings. To do this, the algorithm estimates the correlation $c(x_i, x_j)$ of each ordered pair of variables using examples drawn randomly according to D. We show that these correlations contain enough information to infer the architecture of the network.

3. In the last step, the algorithm estimates a threshold value for each perceptron. We will see below that the correlations provide enough information to do that.

The following lemma shows that there is a measurable *gap*, in terms of influence, between the two cases $w_i = 1$ and $w_i = -1$. This gap is used to estimate the weight values (signs).

Lemma 9.8 *Let f be μ-PDL. Let g be a perceptron in f and let $x_i \in g$. Let $\{\lambda^{(s)}\}$ be the set of x_i's uncles and let $\{f^{(s)}\}$ be the set of subformulas computed by those uncles. Let $a^{(s)} = 1$ if $\lambda^{(s)}$ feeds immediately an AND gate and $a^{(s)} = 0$ if $\lambda^{(s)}$ feeds immediately an OR gate. Then, if $P(g = 1)$, $P(g = 0) > \rho$ and $\prod_s P(f^{(s)} = a^{(s)}) > \gamma$,*

$$Inf(x_i) \begin{cases} > \frac{\gamma\rho}{n+2} & \text{if } w_i = 1 \\ < -\frac{\gamma\rho}{n+2} & \text{if } w_i = -1 \end{cases}$$

Proof: Using lemmas 9.3 and 9.4 it is easy to show by induction on the depth of f that

$$Inf(x_i) = \prod_s P(f^{(s)} = a^{(s)})[\ P(g(\mathbf{x}) = +1 | x_i = +1) - P(g(\mathbf{x}) = +1 | x_i = -1)\ ]$$

The result follows then from the conditions of this lemma and lemma 9.2. Note that we can assume w.l.o.g. that $\rho,\ \gamma > \epsilon/2n$, for otherwise we can neglect perceptron g without introducing much error. Under this assumption, the gap is wide enough to be estimated efficiently using a sample of polynomial size.□

Once we determine the weight values, we reduce f to a monotone μ-PDL by changing x_i to $1 - x_i$ whenever $w_i = -1$. In the following, we concentrate on the monotone case.

The next step is to infer the architecture of the network. It turns out that the *correlation measure* contains enough information to determine which variables appear at a given level, and among those, which are siblings. First, some facts about the correlations.

Lemma 9.9 *Let f be a monotone μ-PDL. Let x_i and x_j be two influential variables in f. Assume that $x_i \in g$. Then*

1. *If x_i and x_j are siblings:*

$$C(x_i, x_j) = C_I = \frac{P(g = 1 | x_i = x_j = 1) - P(g = 1 | x_i = 1, x_j = 0)}{P(g = 1 | x_j = 1) - P(g = 1 | x_j = 0)}.$$

2. *If x_i and x_j are not siblings, x_i is not deeper than x_j, and lca(x_i,x_j) is an OR:*

$$C(x_i, x_j) = C_{II} = \frac{1 - P(g = 1|x_i = 1)}{1 - P(g = 1)}.$$

3. *If x_i and x_j are not siblings, x_i is not deeper than x_j, and lca(x_i,x_j) is an AND:*

$$C(x_i, x_j) = C_{III} = \frac{P(g = 1|x_i = 1)}{P(g = 1)}.$$

Moreover, if g contains p variables and has a renormalized threshold v:

$$C_I = 2\frac{v-1}{p-1} \quad ; \quad C_{II} = 1 - \frac{\binom{p-1}{v-1}}{\sum_{i=0}^{v-1}\binom{p}{i}} \quad ; \quad C_{III} = 1 + \frac{\binom{p-1}{v-1}}{\sum_{i=v}^{p}\binom{p}{i}} \tag{9.13}$$

$$C_I - C_{II} \geq \frac{1}{p^2} \geq \frac{1}{n^2} \qquad \text{for } v \geq 2 \tag{9.14}$$

$$C_I - C_{III} \leq -\frac{1}{2p} \leq -\frac{1}{2n} \qquad \text{for } v \leq p-1 \tag{9.15}$$

Proof: The different expressions of $C(x_i, x_j)$ can be easily derived by induction on the depth of f using lemmas 9.3 and 9.4. The derivation of eq. 9.13 is straightforward. Eqs. 9.14 and 9.15 follow using the approximation introduced in lemma 9.1 (see the proof of lemma 9.6). Note that the correlation gaps established in this lemma are wide enough to be estimated using a polynomial number of examples.□

Let $X' \subseteq X$. We call two variables x_i and x_j *OR-potential-siblings* (w.r.t. X') if $C(x_i, x_j) \geq C(x_i, x_k)$ for all $x_k \in X'$, and *AND-potential-siblings* (w.r.t. X') if $C(x_i, x_j) \leq C(x_i, x_k)$ for all $x_k \in X'$. The following lemma enables us to determine which variables appear in a given level, and among these, which are siblings.

Lemma 9.10 *Let f be a monotone μ-PDL and let $X' \subseteq X$. Assume that the deepest gate Γ fed by all variables in X' is an OR. Let $x_i \in X'$. Then, x_i is not a zero-level variable with respect to Γ iff there exist x_j, $x_k \in X'$ such that, for some permutation $\{i_1, i_2, i_3\}$ of $\{i, j, k\}$: x_{i_1} and x_{i_2} are OR-potential-siblings (w.r.t. X'),*

x_{i_2} and x_{i_3} are OR-potential-siblings (w.r.t. X'), but
x_{i_1} and x_{i_3} are not OR-potential-siblings (w.r.t. X').

Moreover, if x_i is a zero-level variable with respect to Γ, and x_i and x_j are OR-potential-siblings (w.r.t. X'), then x_i and x_j are siblings in f.
The same holds if we replace OR by AND in the lemma.

Proof idea: Consider the different possible situations and apply the facts established in lemma 9.9.□

Assume that the output gate Γ is an OR (we proceed similarly if the output is an AND). To decide whether a variable x_i is a zero-level w.r.t. Γ, we determine the set of its OR-potential-siblings (w.r.t. X), call it T_i. If x_i is a zero-level variable, then every pair of variables in T_i will themselves be OR-potential-siblings. But If x_i is not a zero-level variable, then some pair in T_i will not be OR-potential siblings. There may be several different sets of zero-level siblings determined in this manner, which will form the various perceptrons that feed *immediately* the output gate [5]. A similar process applies if the output gate is an AND, using the AND-potential-siblings technique. We repeat this recursively, removing from X the variables already used and noting that the gates switches from OR to AND (or vice versa).

The last step of the algorithm is to estimate the threshold of each perceptron g. Two cases are possible:
case 1: g is a single-variable perceptron. Then the only possible threshold value is 1.
case 2: g is a multi-variable perceptron. Let p be the number of variables in g and let v be its renormalized threshold. Let x_i, $x_j \in g$. Then (eq. 9.13)

$$C(x_i, x_j) = C_I = 2\frac{v-1}{p-1}$$

Thus, a good estimation of $C(x_i, x_j)$ yields a good estimation of the renormalized threshold v (and hence the original threshold).

By a standard Chernoff bounds analysis [40], one can show that a sample polynomial in (n,ϵ,δ) is sufficient to ensure that, with high probability, the different *influences correlations*,

[5] This argument can be phrased in terms of *type-graphs* that are usually used for read-once formulas [19].

and probabilities are estimated to within a sufficient precision. Finally, the learning algorithm runs in time polynomial in (n,ϵ,δ).

Theorem 9.11 *The class of μ-PDL with binary weights are PAC learnable under the uniform distribution.*

9.3.3 Learning Generalized μ-Perceptron Decision Lists

We extend the results of the previous section to handle the case where the target function f is a generalized μ-PDL. The basic ideas behind the algorithm are just like those of the preceding section. First, we note that lemmas 9.8, 9.9, and 9.10 hold also for generalized μ-PDLs. In fact, the problem of learning generalized μ-PDLs presents only one difficulty that is not encountered in learning μ-PDLs: because a given level in the network may contain more than one gate, we need to determine not only which variables appear at that level and which are siblings, but also which variables appear in *same* subtree rooted at one of the gates of that level. This extra difficulty can be solved fairly easily. The following subroutine, when called with a set X' and a variable x_i, returns the set $S \subseteq X'$ of all variables that feed some AND gate fed by x_i:

AND-test:

1. Set $S = \{x_i\}$.
2. For each variable $x_j \in X'$ and $x_j \notin S$:
   If there exists $x_k \in S$ such that $C(x_k, x_j) + C(x_j, x_k) > 2$, set $S = S \cup x_j$ and Go to 2.
3. If no variable can be added, return S.

Likewise, the following subroutine, when called with a set X' and a variable x_i, returns the set $S \subseteq X'$ of all variables that feed some OR gate fed by x_i:

OR-test:

1. Set $S = \{x_i\}$.
2. For each variable $x_j \in X'$ and $x_j \notin S$:
   If there exists $x_k \in S$ such that $C(x_k, x_j) + C(x_j, x_k) < 2$, set $S = S \cup x_j$ and Go to 2.

3. If no variable can be added, return S.

The intuition behind the above subroutines is that if two variables meet at an AND gate, setting one of them to 1 increases the influence of the other, whereas if they meet at an OR gate, setting one of them to 1 decreases the influence of the other. The following lemma strengthens this intuition by showing that there is a measurable gap between the two cases.

Lemma 9.12 *Let f be a monotone generalized μ-PDL. Let x_i and x_j be two variables in f. Then, if x_i and x_j are not siblings*

$$C(x_i, x_j) + C(x_j, x_i) > 2 + \frac{1}{2}\min(Inf(x_i), Inf(x_j)) \quad \text{if} \quad lca(x_i\ x_j) \text{ is an AND}$$

$$C(x_i, x_j) + C(x_j, x_i) < 2 - \frac{1}{2}\min(Inf(x_i), Inf(x_j)) \quad \text{if} \quad lca(x_i, x_j) \text{ is an OR}$$

Proof: We prove the case where the $lca(x_i, x_j)$ Γ is an AND. Assume w.l.o.g. that x_i is not deeper than x_j. Then x_i's parent g is an immediate input to Γ. Let f' be the subformula computed by gate Γ. Then f' can be decomposed as

$$f' = g \wedge f''$$

where the subformula f'' depends only x_j. Using lemma 9.4, it is easy to show that

$$\begin{aligned} C(x_i, x_j) = C_{III} &= \frac{P(g=1|x_i=1)}{P(g=1)} \\ &= 1 + \frac{P(g=1|x_i=1) - P(g=1)}{P(g=1)} \\ &> 1 + [\ P(g=1|x_i=1) - P(g=1)\ ] \\ &> 1 + \frac{1}{2}Inf(x_i) \end{aligned} \tag{9.16}$$

and that

$$C(x_j, x_i) = \frac{P(f''=1|x_j=1)}{P(f''=1)} > 1 \tag{9.17}$$

Combining eqs. 9.16 and 9.17 yields the desired result.

A similar argument applies to the case where the $lca(x_i, x_j)$ Γ is an OR (using lemma 9.3).□

Assume that the output gate Γ is an OR (a similar procedure applies when Γ is an AND). The zero-level variables w.r.t. Γ are determined as in the previous section, using the OR-potential-siblings technique. These variables are then removed and the AND-test is invoked

to determine which set of variables feed the *same* AND gate in the next level. There may be several different sets determined in this manner; each set S_i will be assigned an AND gate Γ_i in that level. Then, for each set S_i we use the AND-potential-siblings technique to determine which variables are zero-level w.r.t. Γ_i. This may yield several different subsets, which will form the various perceptrons that feed *immediately* the gate Γ_i. We repeat this subdivision process recursively, removing the variables already used and noting that the gates switches from OR to AND (or vice versa).

Finally, the threshold of each perceptron can be estimated by the same method used in the previous section.

Theorem 9.13 *The class of generalized μ-PDL with binary weights are PAC learnable under the uniform distribution.*

9.4 Conclusion and Open Problems

The techniques developed in the previous sections can be extended to learn other nonoverlapping neural concepts on the uniform distribution. For example, our approach can be extended to handle the class of unions (intersections) of nonoverlapping perceptrons with *real* weights but zero thresholds. That is, two-layer networks where the output computes an OR (AND) of its inputs and the hidden units are nonoverlapping perceptrons with real weights and zero thresholds. We do not know yet how to handle the case with non-zero thresholds [6].

The general class of nonoverlapping perceptron networks has recently been shown to be PAC learnable from examples and membership queries [42]. It is still an open problem whether this class is PAC learnable from examples only on the uniform distribution. A more tractable problem would be to extend the techniques of this chapter to handle the class of nonoverlapping perceptron networks with binary weights and arbitrary thresholds.

The hardness results of [14, 65] suggest that one can not avoid the training difficulties

[6] Our proof uses the fact that, under the uniform distribution, single perceptrons with zero thresholds are learnable using only *one type* of examples (positive or negative). It is an open question whether or not this holds also for perceptrons with arbitrary thresholds.

simply by considering only very simple neural networks. The results reported here suggest that the combination of simple networks and reasonable distributions may be needed to achieve any degree of success.

Chapter 10

On Learning Probabilistic Neural Concepts

10.1 Definition of Probabilistic Concepts

The PAC model has been criticized for the *deterministic* and *noise-free* view it takes of the target to be learned. In fact, in many real situations the target concept may exhibit *uncertain* or *probabilistic* behavior and thus, the same example may sometimes be classified as positive and sometimes as negative. The uncertainty may be due to a noisy environment, insufficient inputs/measurements (*e.g.* weather prediction), or *inherent* probabilistic process (*e.g.* predicting the orientation of the spin of a particle).

To handle such situations, Kearns and Schapire introduced a new variant of the PAC model in which the target concept is a probabilistic one [59]. Formally, A probabilistic concept (or p-concept), as defined in [59], is a mapping $c : \mathcal{X}^n \rightarrow [0,1]$. For each $\mathbf{x} \in \mathcal{X}^n$, $c(\mathbf{x})$ is interpreted as the probability that $\mathbf{x}$ is a positive example of the p-concept c. Thus, in the p-concept model, a labeled example is generated as follows: first, an instance $\mathbf{x}$ is chosen according to the target distribution on $\mathcal{X}^n$; then, with probability $c(\mathbf{x})$, the labeled example $< \mathbf{x}, 1 >$ is observed, and with probability $1 - c(\mathbf{x})$, the labeled example $< \mathbf{x}, 0 >$ is observed. Thus, the learning algorithm has no direct access to c : the only access it has is through labeled examples $< \mathbf{x}, \sigma >$.

10.2 Learning Probabilistic Majorities of Nonoverlapping Binary Perceptrons

In this chapter, we attempt to apply Kearns and Schapire's model to learning in neural networks. For that, we restrict ourselves to the uniform distribution of examples and investigate a class of neural p-concepts we call *probabilistic majorities* of nonoverlapping binary perceptrons, defined as:

$$c(\mathbf{x}) = \sum_{s=1}^{r} p_s g^{(s)}(x) + p_0 \tag{10.1}$$

where each $g^{(s)}$ is a perceptron with binary weights and arbitrary threshold, $p_s \geq 0$ for $s = 0, \ldots, r$, and $0 \leq \sum_{s=0}^{r} p_s \leq 1$.

Such p-concepts have a simple neural representation: a two-layer network with r hidden perceptrons and one (probabilistic) output unit that outputs 1 with a probability equal to the weighted sum of its inputs. Each probability p_s may be interpreted as the *relative confidence* we have in perceptron $g^{(s)}$'s decision, and p_0 as the network's bias toward a yes (or 1) decision.

As in [59, 90], we view the problem of learning a p-concept c as that of inferring a good approximation of c itself. This is called in [59] *learning with a model of probability.*

Definition 10.1 *Let C be a class of p-concepts. Then C is said to be learnable with a model of probability under the uniform distribution D if there is an algorithm A such that, for the distribution D, for any $c \in C$, and any $0 < \epsilon, \delta < 1$, A runs in time polynomial in (n, ϵ, δ) and outputs a real-valued hypothesis c' such that, with probability at least $(1 - \delta)$,*

$$E_{\mathbf{x} \in D} |c'(\mathbf{x}) - c(\mathbf{x})| \; < \; \epsilon$$

where $E_{\mathbf{x} \in D} | \ldots |$ denotes the expectation with respect to D.

10.3 The Learning Algorithm

Recall that, when learning a p-concept, we have no direct access to c: the only access we have is through labeled examples $< \mathbf{x}, \sigma >$. So in what follows, the different estimations are taken with respect to the label σ.

Let the target c be a probabilistic majority of nonoverlapping perceptrons defined by eq. 10.1. We are interested in learning, with a model of probability, the equivalent neural network of c. The learning algorithm will make use of the notions of *influence* $Inf(x_i)$, *correlation* $C(x_i, x_j)$, and *siblings* as defined in the previous chapter (see section 9.2).

The learning algorithm proceeds in four steps:

1. Estimating the weight values using the influences. Once this is done, the target p-concept is reduced to its monotone form by simply changing x_i to $1 - x_i$ whenever $w_i = -1$.

2. Inferring the architecture, *i.e.* which variables are siblings (appear in the same perceptron).

3. Estimating the threshold of each perceptron.

4. Estimating the different probabilities $p_0, p_1, \ldots, p_r$.

The following lemma establishes that there exists a measurable gap in terms of the influence between the two cases $w_i = 1$ and $w_i = -1$.

Lemma 10.2 *Let c be a probabilistic majority of nonoverlapping perceptrons as in eq. 10.1. Let $g^{(a)}$ be a perceptron in c and let $x_i \in g^{(a)}$. Then if $P(g^{(a)} = 1)$, $P(g^{(a)} = 0) > \rho$, and $p_a > \gamma$,*

$$Inf(x_i) \begin{cases} > \frac{\gamma\rho}{n+2} & \text{if } w_i = 1 \\ < -\frac{\gamma\rho}{n+2} & \text{if } w_i = -1 \end{cases}$$

Proof: $\underline{w_i = +1}$.

$$\begin{aligned} Inf(x_i) &\equiv P(\sigma = 1 | x_i = 1) - P(\sigma = 1 | x_i = 0) \\ &= [\sum_{s \neq a} p_s P(g^{(s)}(\mathbf{x}) = 1) + P(g^{(a)}(\mathbf{x}) = 1 | x_i = 1) + p_0] \\ &- [\sum_{s \neq a} p_s P(g^{(s)}(\mathbf{x}) = 1) + P(g^{(a)}(\mathbf{x}) = 1 | x_i = 0) + p_0] \\ &= p_a[\, P(g^{(a)}(\mathbf{x}) = 1 | x_i = 1) - P(g^{(a)}(\mathbf{x}) = 1 | x_i = 0)\,] \\ &> \frac{\gamma\rho}{n+2} \end{aligned}$$

The last inequality follows from the conditions of this lemma and lemma 9.2. The case where $w_i = -1$ can be proved similarly.□

We can assume w.l.o.g. that $\gamma > \epsilon/2n$, for otherwise we can neglect perceptron $g^{(a)}$ without introducing much error. Likewise, we can assume w.l.o.g. that $\rho > \epsilon/2n$, for otherwise we can either neglect perceptron $g^{(a)}$ (if $P(g^{(a)} = 1) < \epsilon/2n$) or replace it by the constant 1 (if $P(g^{(a)} = 0) < \epsilon/2n$). The latter amounts also to neglecting perceptron $g^{(a)}$ because any constant can be absorbed in the probability p_0. Under this assumption, the gap is wide enough to be estimated efficiently using a sample of polynomial size.

Once we have an estimate of the weight values, we reduce c to its monotone form by changing x_i to $1 - x_i$ whenever $w_i = -1$. The next step is to infer the architecture of the monotone p-concept c.

Lemma 10.3 *Let c be a monotone probabilistic majority of nonoverlapping perceptrons. Let $g^{(a)}$ be a perceptron in c with p variables and a renormalized threshold v. Let $x_i \in g^{(a)}$ and let x_j be another variable in c. Then*

$$C(x_i, x_j) = \begin{cases} 2\frac{v-1}{p-1} & \text{if } x_j \in g^{(a)} \\ 1 & \text{if } x_j \notin g^{(a)} \end{cases}$$

Proof: Recall that

$$C(x_i, x_j) \stackrel{\text{def}}{=} \frac{P(\sigma = 1 | x_i = x_j = 1) - P(\sigma = 1 | x_i = 1, x_j = 0)}{P(\sigma = 1 | x_j = 1) - P(\sigma = 1 | x_j = 0)}.$$

Then, if $x_j \in g^{(a)}$,

$$\begin{aligned} P(\sigma = 1 | x_i = x_j = 1) - P(\sigma = 1 | x_i = 1, x_j = 0) &= p_a[\, P(g^{(a)}(\mathbf{x}) = 1 | x_i = x_j = 1) \\ &- P(g^{(a)}(\mathbf{x}) = 1 | x_i = 1, x_j = 0)\,] \end{aligned} \tag{10.2}$$

and

$$P(\sigma = 1 | x_j = 1) - P(\sigma = 1 | x_j = 0) = p_a[\, P(g^{(a)}(\mathbf{x}) = 1 | x_j = 1) - P(g^{(a)}(\mathbf{x}) = 1 | x_j = 0)\,]$$

Hence

$$\begin{aligned} C(x_i, x_j) &= \frac{P(g^{(a)}(\mathbf{x}) = 1 | x_i = x_j = 1) - P(g^{(a)}(\mathbf{x}) = 1 | x_i = 1, x_j = 0)}{P(g^{(a)}(\mathbf{x}) = 1 | x_j = 1) - P(g^{(a)}(\mathbf{x}) = 1 | x_j = 0)} \\ &= 2\frac{v-1}{p-1} \end{aligned}$$

The last equality follows from eq. 9.10. The case where $x_j \notin g^{(a)}$ can be proved similarly.□

One implication of this lemma is that as long as g is not a majority function ($v \neq (p+1)/2$), we can determine whether or not x_i and x_j are siblings. But if g is a majority function ($v = (p+1)/2$), estimating $C(x_i, x_j)$ yields no information on whether or not x_i and x_j are actually siblings [1]. To overcome this difficulty, we introduce a new measure of correlation that depends on triplets of variables. More precisely, we define the correlation of a variable x_k with two variables x_i and x_j, denoted $C(x_i, x_j, x_k)$, as

$$C(x_i, x_j, x_k) \stackrel{\text{def}}{=} \frac{P(\sigma = 1 | x_i = x_j = x_k = 1) - P(\sigma = 1 | x_i = x_j = 1, x_k = 0)}{P(\sigma = 1 | x_k = 1) - P(\sigma = 1 | x_k = 0)}$$

The following lemma shows that this new correlation measure solves our problem.

Lemma 10.4 *Let c be a monotone probabilistic majority of nonoverlapping perceptrons. Let x_i, x_j, and x_k be three variables in c and assume that the parents of these variables compute majority functions. Assume that $x_i \in g$ where g is a perceptron with p variables. Then*

$$C(x_i, x_j, x_k) = \begin{cases} 1 - \frac{1}{p-2} & \text{if } x_j, x_k \in g \\ 1 & \text{otherwise} \end{cases}$$

Proof idea: follows by similar arguments to the ones used to prove lemma 10.3.□

Thus, to infer the architecture, we first estimate the correlation $C(x_i, x_j)$ of each ordered pair of variables, to within a sufficient precision. Whenever $|C(x_i, x_j)|$ is strictly different from one, we conclude that x_i and x_j are siblings. For the unsolved cases, we estimate the correlation $C(x_i, x_j, x_k)$ of each ordered triplet of variables, again to within a sufficient precision. We conclude that x_i, x_j, and x_k are siblings if $C(x_i, x_j, x_k)$ is strictly less than one, and not siblings otherwise.

The next step is to estimate the threshold of each perceptron. This can be done exactly as in section 9.3.2.

The final step is to estimate the different probabilities $p_0, \ldots, p_r$. For this, note that

$$p_s = P(c = 1 | g^{(s)} = 1) - P(\sigma = 1 | g^{(s)} = 0) \qquad \text{for} \quad s = 1, \ldots, r$$

[1] This is related to the fact that the amplification function is independent of the level at which the two variables meet[34].

and

$$p_0 = P(\sigma = 1) - \sum_{s=1}^{r} p_s P(g^{(s)} = 1)$$

Since we have already determined the different perceptrons $g^{(s)}$ $(s = 1, \ldots, r)$, the different probabilities $p_0, p_1, \ldots, p_r$ can be estimated easily using the above expressions,

A standard Chernoff bounds analysis [40] will show that a sample polynomial in (n,ϵ,δ) is sufficient to ensure that, with high probability, the different *influences*, *correlations*, and *probabilities* are estimated to within a sufficient precision, and consequently $E_{\mathbf{x}\in D}|c'(\mathbf{x}) - c(\mathbf{x})| < \epsilon$.

Theorem 10.5 *The class of probabilistic majorities of nonoverlapping perceptrons with binary weights is learnable with a model of probability under the uniform distribution.*

10.4 Extensions and Conclusion

The techniques developed in the previous sections can be extended to learn other probabilistic nonoverlapping neural concepts (with binary weights) on the uniform distribution. For example, let g be a binary perceptron with binary weights and arbitrary threshold, and let c be the p-concept defined by

$$c(\mathbf{x}) = p_1 g(\mathbf{x}) + p_2(1 - g(\mathbf{x}))$$

That is, if $g(\mathbf{x}) = 1$, then with probability p_1, the labeled example $< \mathbf{x}, 1 >$ is observed, and with probability $1 - p_1$, the labeled example $< \mathbf{x}, 0 >$ is observed. Similarly, if $g(\mathbf{x}) = 0$, then with probability p_2, the labeled example $< \mathbf{x}, 1 >$ is observed, and with probability $1 - p_2$, the labeled example $< \mathbf{x}, 0 >$ is observed. This p-concept models for example the behavior of a *noisy* perceptron where the classification label of each positive example is flipped independently with some probability $1 - p_1$ and the classification label of each negative example is flipped independently with some probability p_2.

Our techniques show that this class of p-concepts is learnable under the uniform distribution. Consequently, noisy perceptrons with binary weights and arbitrary thresholds are efficiently learnable under the uniform distribution.

As a second example, let c be the *probabilistic* version of μ-PDL (chapter 9), that is a list $\mathcal{L}$ of pairs

$$(g^{(1)}, p_1), (g^{(2)}, p_2), ..., (g^{(r)}, p_r)$$

where each $p_i \in [0,1]$. For any input $\mathbf{x}$, $c(\mathbf{x})$ is defined to be p_j, where j is the least index for which $g^{(j)}(\mathbf{x}) = 1$. This probabilistic concept can also be written as:

$$c(\mathbf{x}) = \sum_{s=1}^{r} p_s g^{(s)}(\mathbf{x}) \prod_{j=1}^{s-1} (1 - g^{(j)}(\mathbf{x}))$$

We do not know yet how to handle general probabilistic μ-PDLs, but if we assume that $p_1 \geq \ldots \geq p_r$, then our approach applies to such p-concepts. Following [59], we call these concepts probabilistic μ-PDL *with decreasing probabilities.*

Finally, we should mention here that very little is known about the efficient learnability of probabilistic neural concepts. In fact, it is not even known whether single perceptrons (with real weights *and* arbitrary thresholds) are efficiently learnable from noisy examples on the uniform distribution. Obviously, a lot of work has to be done to understand the behavior of neural networks in noisy or uncertain environments. We hope the investigation reported in this chapter is a small step in this direction.

Part IV

On Learning Neural Networks with Binary Weights: an Average Case Analysis

Chapter 11

The Learning Curves of the Clipped Hebb Rule for Single Perceptrons with Binary Weights

11.1 General Introduction

Much of the recent progress in the theoretical understanding of learning algorithms has come within the PAC model and its variants. But while this approach has yielded some impressive results under very general conditions, it has not been clear how these results relate to specific situations, or to the *average case* performance. In particular, the PAC results provide upper bounds on the performance of *any* consistent learning algorithm. Thus, even the algorithm that always finds a *worst* hypothesis that is consistent with the training examples but that has the largest possible error with respect to the distribution D is covered by the VC dimension analysis (lemma 6.3). In practice, it seems unlikely that we would encounter such algorithms–reasonable algorithms should manage to find an *average* hypothesis, in terms of error with respect to D.

A second line of research, based on a statistical mechanical formulation of learning, has recently been investigated in the context of neural networks [25, 39, 64, 77, 78, 99]. This approach is concerned with the calculation of the *average case* or *typical* performance

of *specific* learning algorithms under some restricted conditions. More specifically, it is concerned with the analytical evaluation of the *average generalization* as a function of the size of the training sample, where the average is taken over all possible training samples of the given size. We will not go in details here, but we mention only that this approach is strongly influenced by ideas and tools from statistical physics [1].

The results obtained with the average case approach are so far limited to very simple networks, often single perceptrons, with simple probability distributions of examples. Nonetheless, the results obtained are elegant, to say the least, and are in very good agreement with the numerical simulations [72]. In the rest of this thesis we adopt this approach and try to evaluate the average generalization of the *clipped Hebb rule* (section 8.4) when learning different neural networks with binary weights.

11.2 Introduction

The generalization properties of neural networks with binary weights have been studied extensively using the *statistical mechanics approach* [39, 70, 94, 101]. Although those studies have yielded some impressive results, like the presence of a phase transition to perfect generalization, they have their shortcomings. In particular, they neglect the *computational* aspect of the learning process. To simplify the mathematical analysis, those studies assume a stochastic training algorithm, similar to a first monte carlo process, that leads at long times to a Gibbs distribution [94]. Unfortunately, stochastic training algorithms generally require prohibitively long convergence times. Here, we are interested in the average case behavior of *efficient* algorithms, in terms of time complexity, for learning this class of networks.

Perhaps the simplest algorithm that one may think of for learning binary networks is the *clipped Hebb rule* [61], also called the majority rule in [107] (see section 8.4). This rule is local, homogeneous, and simple enough to be biologically plausible and easy to implement. Moreover, it observes each training example only once, and so, its running time increases only linearly with the number of training examples.

[1] A first attempt to bridge the gap between the PAC model and the average case approach has been reported in [48].

In this chapter, we investigate the typical behavior of the clipped Hebb rule when learning single perceptrons with binary weights and zero threshold, under the uniform distribution. We calculate exactly its learning curve (*i.e.* the average generalization as a function of the number of training examples) in the limit of a large number of input variables. We find that the average generalization rate converges *exponentially* fast to perfect generalization. Consequently, the sample complexity in the average case is of $O(n \ln(1/\epsilon))$, a large improvement over the PAC learning analysis of chapter 8. We also calculate the average generalization rate when learning from noisy examples and show that the clipped Hebb rule is very robust with respect to classification noise. The results of the extensive simulations are in very good agreement with the theoretical ones, even for moderate values of n.

We note here that the clipped Hebb rule produces hypotheses that are not necessarily consistent with all the training examples, but that nonetheless have very good generalization ability. These type of algorithms are called "inconsistent algorithms" [72].

11.3 Definition

Let $\mathcal{X}^n$ (the input space) be the set $\{-1,+1\}^n$. We are interested in learning a *target* perceptron g^t, with a binary valued weight vector $\mathbf{w}^t$, that maps from the set X into $\{-1,+1\}$:

$$g^t(\mathbf{x}) = \mathrm{sgn}(\sum_{i=1}^{n} w_i^t x_i) = \begin{cases} +1 & \sum_{i=1}^n w_i^t x_i \geq 0 \\ -1 & \sum_{i=1}^n w_i^t x_i < 0 \end{cases} \tag{11.1}$$

where $w_i^t = \pm 1$ for $i = 1, \ldots, n$. We assume, w.l.o.g., that n is odd.

The training examples are input vectors $\{\mathbf{x}^l\}_{l=1,\ldots,m}$, generated according to the uniform distribution D on $\mathcal{X}^n$, and labeled according to the target perceptron g^t. An example is said to be positive (negative) if $g^t(\mathbf{x}) = +1$ (-1). Given the training examples, the goal of the learning algorithm is to find an hypothesis perceptron g, with a weight vector $\mathbf{w}$, that approximates the most the target perceptron.

Let $\sigma^l \equiv g^t(\mathbf{x}^l)$. The clipped Hebb rule can be simply written as

$$w_i = \mathrm{sgn}(\sum_{l=1}^{m} \sigma^l x_i^l) \quad i = 1, \ldots, n \tag{11.2}$$

where $\mathrm{sgn}(a) = +1$ if $a > 0$ and -1 otherwise.

We will calculate analytically the learning curve of the clipped Hebb rule (hereafter CHR), *i.e.* the curve of the average generalization as a function of the size m of the training set. The average is taken over all training sets of size m and all target weights $\mathbf{w}^t$. Formally the results are exact only in the thermodynamic limit, *i.e.* when the number of weights n approaches infinity. However, as our extensive simulations indicate, those results provide a very good approximation to the behavior of the CHR even for moderate values of n.

The correct thermodynamic limit requires the number of training examples to scale as $m = \alpha n$, where the constant α remains finite as n grows.

11.4 Average Case Behavior in the Limit of Large n

11.4.1 Zero Noise

The generalization rate G is defined as the probability that the hypothesis agrees with the target on a new random example $\mathbf{x}$ chosen according to the uniform distribution D.

We start by rederiving a very well known result that relates the generalization rate G to the angle between the target and the hypothesis weight vectors [78]. For that, let us define the following sums of random variables:

$$X = \sum_{i=1}^{n} w_i x_i \tag{11.3}$$

$$Y = \sum_{i=1}^{n} w_i^t x_i \ , \tag{11.4}$$

The generalization rate is then given by

$$G = P\left[g^t(\mathbf{x}) = g(\mathbf{x})\right] \tag{11.5}$$

$$= P\left[\mathrm{sgn}(X) = \mathrm{sgn}(Y)\right] \tag{11.6}$$

$$= P\left[XY > 0\right] \tag{11.7}$$

Since $\mathbf{x}$ is distributed uniformly, it is easily to see that:

$$E(X) = E(Y) = 0 \tag{11.8}$$

$$Var(X) = Var(Y) = n \tag{11.9}$$

$$E(XY) = \sum_{i=1}^{n} w_i w_i^t = n \times \rho \tag{11.10}$$

where $E(\ldots)$ and $Var(\ldots)$ denote respectively the expectation and the variance with respect to the distribution D, and ρ is the normalized *overlap* between the target and the hypothesis weight vector. That is,

$$\rho = \frac{\sum_{i=1}^{n} w_i w_i^t}{n} \tag{11.11}$$

According to the central limit theorem, in the limit $n \to \infty$, X and Y will be distributed according to a bivariate normal distribution with moments given by eq. 11.8, 11.9 and 11.10. Hence, for fixed $\mathbf{w}^t$ and $\mathbf{w}$, the generalization rate G is given by:

$$G = 2 \int_0^\infty dX \int_0^\infty dY \; p(X, Y)$$

where the joint probability distribution $p(X, Y)$ is given by

$$p(X, Y) = \frac{1}{2\pi n \sqrt{(1-\rho^2)}} \times \exp\left[-\frac{X^2}{2n} - \frac{(Y - \rho X)^2}{2n(1-\rho^2)} \right]$$

This integral easily evaluates to give [78]:

$$G = 1 - \frac{1}{\pi} \arccos \rho \tag{11.12}$$

That is, under our assumptions, the generalization rate depends only on the angle between the target and the hypothesis weight vectors.

Now, to average this result over all training samples of size m, we argue that for large n, the distribution of the random variable ρ becomes sharply peaked at its mean. This allows us to approximate $\overline{G(\rho)}$ by $G(\bar{\rho})$, where the overbar denotes the average over all training sets of size αn. Because the results will be independent of $\mathbf{w}^t$, we assume from now on that $w_i^t = 1$ for $i = 1, \ldots, n$.

Let

$$y_i^l = \sigma^l x_i^l \tag{11.13}$$

and let

$$Y_i = \sum_{l=1}^{\alpha n} y_i^l \tag{11.14}$$

Then, eq. 11.2 can be written as

$$w_i = \mathrm{sgn}(\sum_{l=1}^{\alpha n} y_i^l) = \mathrm{sgn}(Y_i) \tag{11.15}$$

The random variable Y_i is simply the sum of αn independent and identically distributed, ± 1 random variables. Let us define q such that

$$P(y_i^l = +1) = \frac{1}{2} + \frac{q}{\sqrt{n}} \quad \text{as } n \to \infty.$$

where, as we shall see, q is independent of i and l.

Note here that $\frac{q}{\sqrt{n}}$ reflects the correlation between the state of each input node and the output node (for a random function, $P(y_i = +1) = 1/2$ and $q = 0$). This correlation is positive if $w_i^t = +1$ and negative if $w_i^t = -1$. The clipped Hebb rule exploits this correlation to determine the sign (value) of w_i.

According to the central limit theorem, as $n \to \infty$, Y_i will be distributed according to a normal distribution with mean

$$\mu = 2\alpha n \frac{q}{\sqrt{n}}$$

and variance

$$\sigma = \sqrt{\alpha n}.$$

Hence,

$$\begin{aligned} \overline{w_i} &= P(Y_i > 0) - P(Y_i < 0) \\ &= \frac{2}{\sqrt{\pi}} \int_0^{\mu/\sqrt{2}\sigma} e^{-t^2} dt \\ &\equiv \text{erf}(q\sqrt{2\alpha}) \end{aligned}$$

where erf denotes the error function. This yields,

$$\overline{p} = \frac{\sum_{i=1}^{n} \overline{w_i}}{n} = \text{erf}(q\sqrt{2\alpha}) \tag{11.16}$$

We need to specify the value [2] of q. From the definition of y_i^l, it is easy to see that $P(y_i^l = +1)$ is simply the probability that an input variable is set to $+1$ when each negative training example $\mathbf{x}^l$ is replaced by $-\mathbf{x}^l$. Note that, once the negative examples are inverted, each example has at least $(n+1)/2$ of its inputs set to $+1$. Under the uniform distribution,

[2] The reason we did not do that earlier is that we will make use of eq. 11.16, in its general form, later in this chapter and in the next chapter.

$P(y_i^l = +1)$ is given by

$$P(y_i^l = +1) = \frac{\sum_{r=\frac{n+1}{2}}^{n} \binom{n}{r} \frac{r}{n}}{\sum_{r=\frac{n+1}{2}}^{n} \binom{n}{r}} = \frac{1}{2} + \frac{\binom{n-1}{\frac{n-1}{2}}}{2^n}$$

$$\sim \frac{1}{2} + \frac{1}{\sqrt{2\pi n}} \text{ as } n \to \infty \tag{11.17}$$

It follows that

$$q = \frac{1}{\sqrt{2\pi}} \tag{11.18}$$

Thus,

$$\overline{\rho} = \text{erf}(\sqrt{\alpha/\pi}) \tag{11.19}$$

$$G(\alpha) = 1 - \frac{1}{\pi}\cos^{-1}\left(\text{erf}[\sqrt{\alpha/\pi}]\right) \tag{11.20}$$

The average generalization rate and normalized overlap are plotted in fig. 11.1 and compared with numerical simulations. We see that the agreement with the theory is excellent, even for moderate values of n. Notice that the agreement is slightly better for $\overline{\rho}$ than it is for $G(\alpha)$. This illustrates the difference between $\overline{G(\rho)}$ and $G(\overline{\rho})$.

To compare this analytic result for the average case to the bounds given by PAC learning (chapter 8), we exploit the fact that we can bound $\text{erf}(z)$ by an exponential [1], and thus bound the error rate $1 - G = \epsilon$ by:

$$\epsilon < \exp\left(-\frac{\alpha}{2\pi}\right) \tag{11.21}$$

That is, the error rate decreases *exponentially* with the number of examples and, on average, a training set of size of $O(n \ln(1/\epsilon))$ is sufficient to produce an hypothesis perceptron with error rate ϵ. This is an important improvement over the bound given by our PAC learning analysis.

Thus, the CHR is a striking example of a very simple "inconsistent" algorithm that does not always produce hypotheses that agree with all the training examples, but nonetheless produce hypotheses with outstanding generalization ability. Moreover, the exponential convergence outlines the computational advantage of learning binary perceptrons using binary

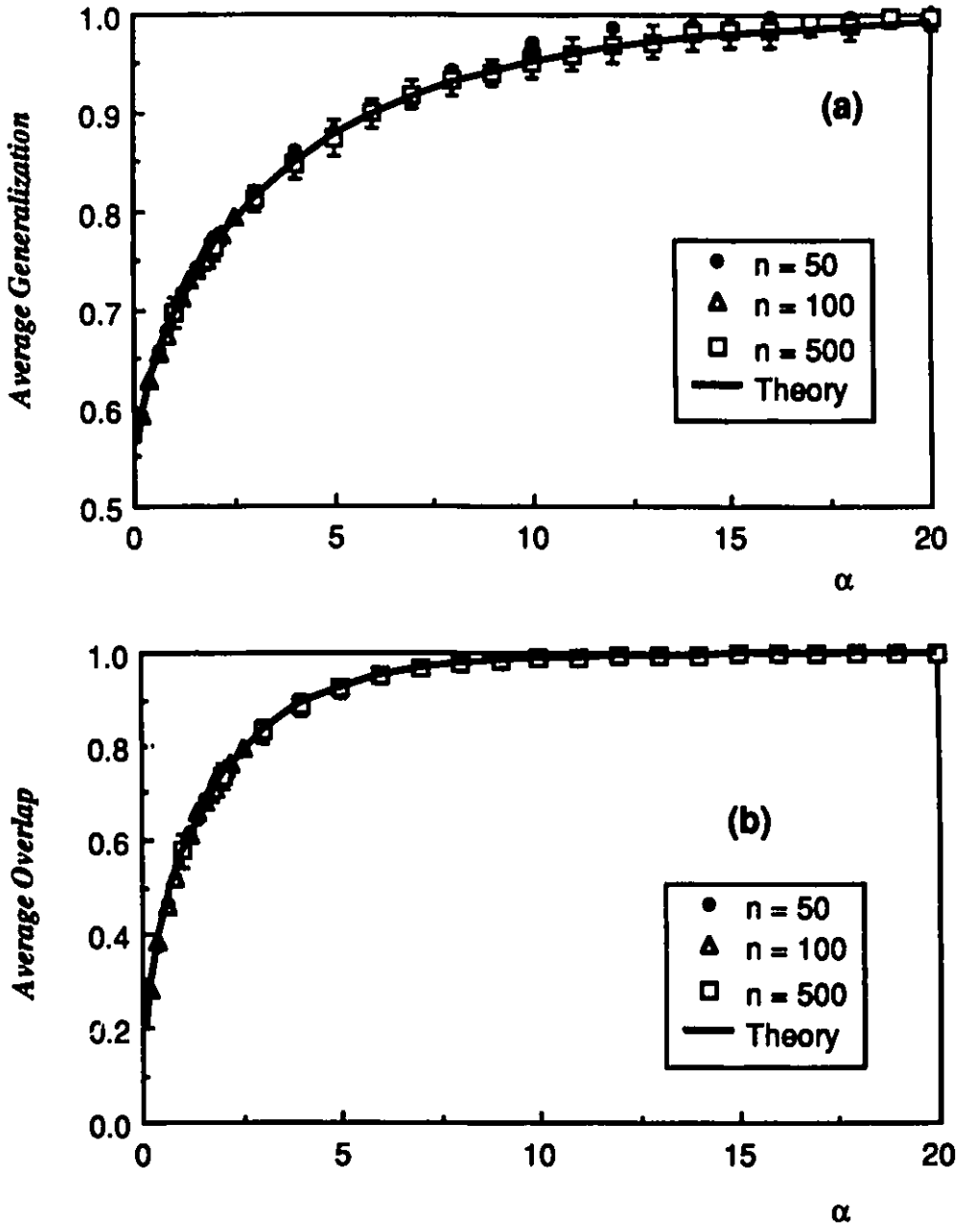

Figure 11.1: (a) The average generalization rate $G(\alpha)$ and (b) the average normalized overlap $\bar{\rho}$ as a function of normalized number of examples $\alpha = m/n$. Numerical results are shown for $n = 50, 100, 500$. Each point denotes an average over 50 different training samples and the error bars denote the standard deviations.

perceptrons. In fact, if one allows real weights, no algorithm can outperform the Bayes optimal algorithm[77]. The latter's error rate improves only *algebraically*, approximately as $0.44/\alpha$.

On the other hand, for consistent learning rules that produce perceptrons with binary weights, a phase transition to perfect generalization is known to take place at a critical value of α [39, 94, 99]. Thus, these rules have a slightly better sample complexity than the CHR. Unfortunately, they are much more computationally expensive (with a running time that generally increases exponentially with the number of inputs n). Since it is an "inconsistent" learning rule, the CHR does not exhibit a phase transition to perfect generalization. We think that the exponential convergence is the reminiscence of the "lost" phase transition.

An interesting question is how the CHR behaves when learning binary perceptrons on product distributions. To answer this, we first note that the CHR works by exploiting the *correlation* between the state of each input variable x_i and the classification label (eq. 11.2). Under the uniform distribution, this correlation is positive if $w_i^t = +1$ and negative if $w_i^t = -1$. This is no more true for product distributions: one can easily craft some malicious product distributions where, for example, this correlation is negative although $w_i^t = +1$. The CHR will be fooled by such distributions because it does no take into account the fact that the settings of the input variables do not occur with the same probability. The algorithm of chapter 8 fix this problem by taking this fact into consideration, through the conditional probabilities.

Finally, it is important to mention that binary perceptrons trained with the CHR on examples generated uniformly will perform well even when tested on examples generated by non-uniform distributions, as long as these distributions are *reasonable* (for a precise definition of reasonable distributions, see [8]).

11.4.2 Classification Noise

In this section we are interested in the generalization rate when learning from noisy examples. We assume that the classification label of each training example is flipped independently with some probability γ. Since the object of the learning algorithm is to construct an hypothesis $\mathbf{w}$ that agrees the most with the underlying target $\mathbf{w}^t$, the generalization rate G is defined

to be the probability that the hypothesis agrees with the *noise-free* target on a new random example x.

The generalization rate G as a function of ρ is still given by eq. 11.12. To calculate the effect of noise on $\overline{\rho}$, recall that in noise-free regime

$$P(y_i^l = +1) = \frac{1}{2} + \frac{q}{\sqrt{n}} \quad \text{as } n \to \infty.$$

Let us define q' such that, for a noise level γ

$$P(y_i^l = +1)_{\text{noise}} = \frac{1}{2} + \frac{q'}{\sqrt{n}} \quad \text{as } n \to \infty.$$

It is easy to see that

$$P(y_i^l = +1)_{\text{noise}} = (1-\gamma) \times P(y_i^l = +1) + \gamma \times (1 - P(y_i^l = +1))$$

This implies that q and q' are related by:

$$q' = q(1-2\gamma) \tag{11.22}$$

$$= \frac{1-2\gamma}{\sqrt{2\pi}} \text{ as } n \to \infty \tag{11.23}$$

where we have used eq. 11.18 for the last equality. This leads to the following expressions for the normalized overlap and the generalization rate, in the presence of noise:

$$\overline{\rho} = \text{erf}\left((1-2\gamma)\sqrt{\alpha/\pi}\right) \tag{11.24}$$

$$G(\alpha) = 1 - \frac{1}{\pi}\arccos\left[\text{erf}\left((1-2\gamma)\sqrt{\alpha/\pi}\right)\right] \tag{11.25}$$

One can see that the algorithm is very robust with respect to classification noise: the average generalization rate still converges exponentially to 1 as long as $\gamma < 1/2$. The only difference with the noise-free regime is the presence of the prefactor $(1-2\gamma)$.

The average generalization rate for different noise levels γ is plotted in fig. 11.2. We see that the numerical simulations are in excellent agreement with the theoretical curves.

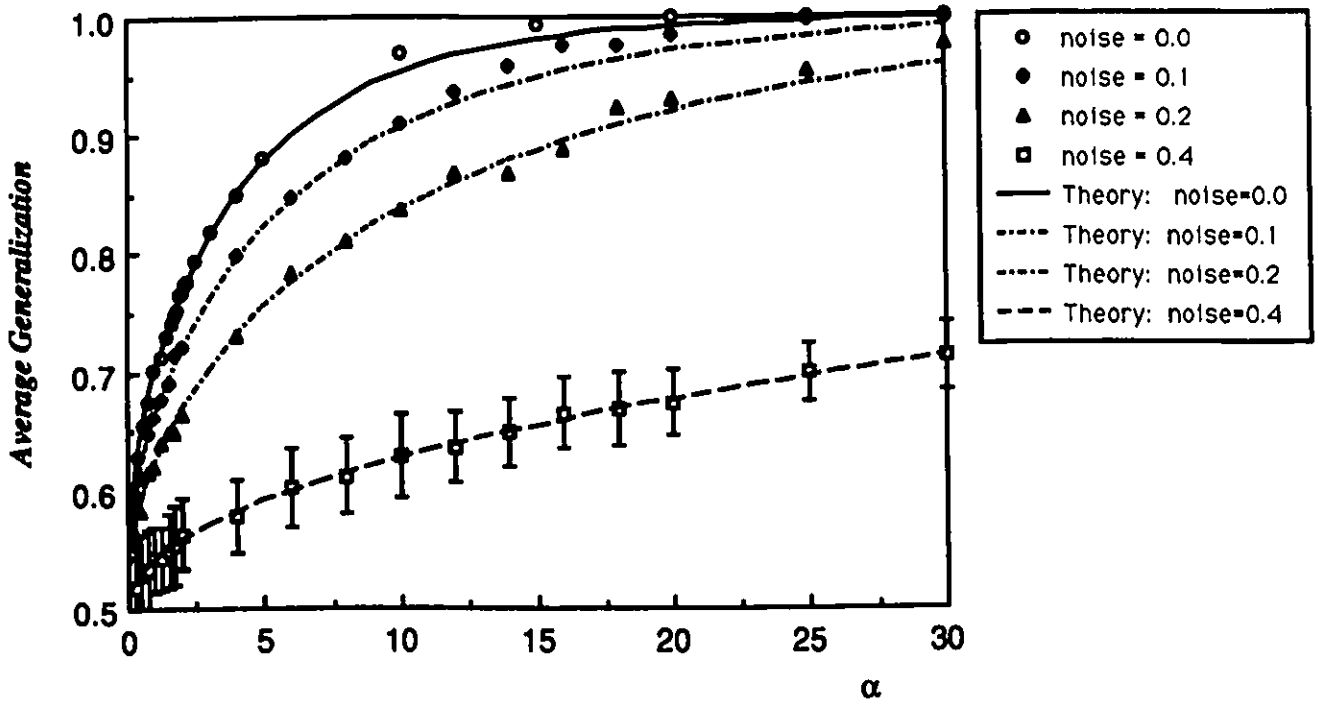

Figure 11.2: The average generalization rate $G(\alpha)$ for different noise levels γ. Numerical results are shown for $n = 100$. Each point denotes the average over 50 different simulations (*i.e.* 50 different noisy training sets). The error bars (indicated only for $\gamma = 0.4$ for clarity) denote the standard deviations.

11.5 Summary

We have calculated exactly the learning curve of the clipped Hebb rule in the limit $n \to \infty$ when learning a single perceptron with binary weights. We have found that the error rate converges exponentially to zero and have thus improved the sample complexity to $O(n \ln(1/\epsilon))$. The analytic expression of the learning curve is in excellent agreement with the numerical simulations, even for moderate values of n.

We have also calculated the learning curve of the clipped Hebb rule when learning from noisy examples. The results indicate that this rule is very robust with respect to random classification noise. Again, the analytic expressions of the learning curves for different noise levels are in excellent agreement with the numerical simulations.

Chapter 12

Learning Curves of the Clipped Hebb Rule for Nonoverlapping Networks with Binary Weights

12.1 Introduction

In the previous chapter, we calculated the learning curves of the clipped Hebb rule when learning single binary perceptrons under the uniform distribution. We found that the generalization rate converges *exponentially* to perfect generalization as a function of the number of training examples. These findings were confirmed by extensive simulations.

In this chapter, we take the investigation one step further. We look at the average behavior of the clipped Hebb rule when learning layered networks of *nonoverlapping* perceptrons with binary weights and *zero thresholds* (fig. 12.1). We recall that a network is nonoverlapping if each node, including the inputs, has one and only one outgoing connection. This is referred to, within the computational learning community, as the μ or the read-once restriction [79, 90] (see chapter 9).

We show that, under the uniform distribution of examples, the clipped Hebb rule does indeed learn this class of networks. We derive expressions for the average generalization rate of this rule, in the limit of large number of inputs, when learning 1) a union of nonoverlapping

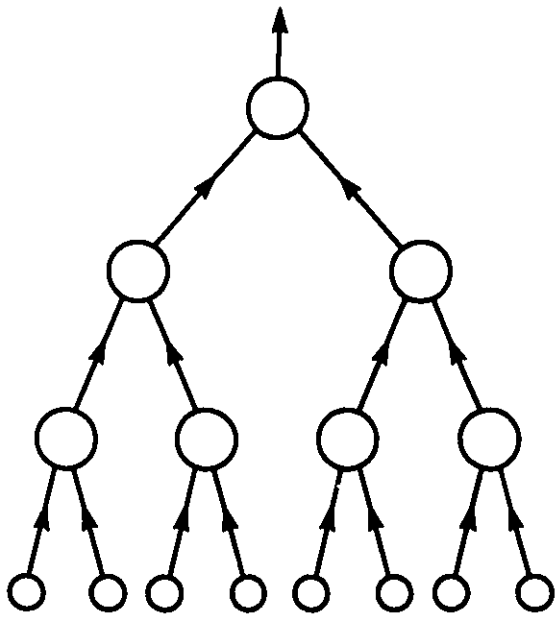

Figure 12.1: A multilayer network of nonoverlapping binary perceptrons. Note that each node has one and only one outgoing connection. All weights in the network are binary valued (± 1). The hidden nodes and the output node are binary valued perceptrons.

binary perceptrons, 2) a two-layer network of nonoverlapping binary perceptrons, and 3) a multilayer network of nonoverlapping binary perceptrons. This is done using only very simple counting arguments and the central limit theorem. We find that the generalization rates still converge extremely rapidly, often exponentially, to perfect generalization. The results of extensive simulations are in very good agreement with the theoretical predictions.

12.2 Definitions

Let $\mathcal{X}^n$ (the input space) be the set $\{-1,+1\}^n$. We are interested in learning a *target function* f^* that maps from the set $\mathcal{X}^n$ into $\{-1,+1\}$. We assume that f^* is a layered network of nonoverlapping perceptrons with binary weights and zero thresholds. (fig. 12.1). For an input vector $\mathbf{x} \in X$, we take x_i to be the state of the input node i of the network.

One interesting property of nonoverlapping networks is that one can assume, w.l.o.g., that all the weights, except these coming directly from the input nodes, are positive [42] [1].

[1] Any other case reduces to this one by an analog to De Morgan's laws that allows us to push negation of

From now on, we assume this is the case and concentrate only on learning the input level weights.

For nonoverlapping networks, each input node i has one and only one input level weight J_i. We define J to be the weight vector obtained from the collection of all these input level weights. Then, each possible setting of the weight vector J defines a mapping function f. We denote by $\mathbf{J}^*$ the weight vector associated with f^*. We call $\mathbf{J}^*$ the *target weight vector* and the corresponding network the *target network*. Each perceptron (hidden unit) in the target network is referred to as a *target perceptron*.

The training examples are input vectors $\{\mathbf{x}^l\}_{l=1,\ldots,m}$, generated according to the uniform distribution D on X, and labeled according to the target function (network) f^*. An example is said to be positive (negative) if $f^*(\mathbf{x}) = +1(-1)$.

Knowing the target network architecture [2] and using the training examples, the goal of the learning algorithm is to find a setting of J that approximate the most the target function. The network corresponding to the J found by the algorithm is called the *hypothesis network*. Each perceptron in the hypothesis network is referred to as an *hypothesis perceptron*.

Let $\sigma^l \equiv f^*(\mathbf{x}^l)$. The clipped Hebb rule, for the network, can be simply written as

$$J_i = \mathrm{sgn}(\sum_{l=1}^{m} \sigma^l x_i^l) \quad i = 1, \ldots, n \tag{12.1}$$

where $\mathrm{sgn}(a) = +1$ if $a > 0$ and -1 otherwise.

Because the results of this paper are independent of $\mathbf{J}^*$, we assume from now on that $J_i^* = 1$ for $i = 1, \ldots, n$.

Before we leave this section, we define new probabilistic quantities that will be useful later. We denote by $P(A)$ the probability that the event A occurs. We denote by $P(A \,\&\, B)$ the probability that events A and B do occur simultaneously. We denote by $P(A \mid B)$ the conditional probability that event A occurs given the fact that event B has been observed. All probabilities are taken with respect to the uniform distribution D on X.

---

weights down to the input level.

[2] Here, we assume that the architecture is known in advance.

12.3 Learning a Union of Nonoverlapping Binary Perceptrons

Let us assume that the target function (network) f^* is a union of k nonoverlapping binary perceptrons:

$$f^* = g_1^* \vee g_2^* \vee \ldots \vee g_k^*$$

In other words, the target network is a two-layer network of nonoverlapping binary perceptrons where the output node computes an OR function. In this case, a negative example is classified negative by each and every target perceptron. A positive example is classified positive by one or more target perceptrons.

We denote by $\mathbf{Ir}(\mathbf{x})$ the vector $(g_1(\mathbf{x}), \ldots, g_k(\mathbf{x}))$. This is called the *internal representation* of $\mathbf{x}$. Obviously, $\mathbf{Ir}(\mathbf{x})$ depends on the setting of $\mathbf{J}$. We denote by $\mathbf{Ir}^*(\mathbf{x})$ the *target* internal representation, *i.e.* the one corresponding to $\mathbf{J}^*$.

The calculation of the generalization curves is divided into two steps. First, we derive the analytical expression of the generalization rate of *one* perceptron in the hypothesis network. Then, we determine the overall generalization of the network.

Let $G_j(\alpha)$ denotes the generalization rate of the hypothesis perceptron g_j, *i.e.* the probability that g_j agrees with the *corresponding* target perceptron g_j^* on a new random example $\mathbf{x}$. Let R_j denotes the overlap between the weight vectors associated with g_j^* and g_j. Assume, for now, that each perceptron is connected to the same number of input variables, n/k. Then

$$R_j = \frac{\sum_{i \in S_j} J_i^* J_i}{n/k} \tag{12.2}$$

where S_j denotes the set of indices of variables connected to g_j^* $(j = 1, \ldots, k)$. It is easy to see that, under our assumptions, R_j and G_j are the same for all perceptrons. So, let us put

$$R_j \equiv R \quad \text{and} \quad G_j \equiv G$$

Obviously, $G(\alpha)$ is still related to the overlap by eq. 11.12

$$G(\alpha) = 1 - \frac{1}{\pi}\cos^{-1}(R) \tag{12.3}$$

Let y_i^l be defined as in the previous chapter. That is,

$$y_i^l = \sigma^l x_i^l \tag{12.4}$$

and let us write again

$$P(y_i^l = +1) = \frac{1}{2} + \frac{q_u}{\sqrt{n/k}} \quad \text{as } n/k \to \infty$$

for some q_u. The $\sqrt{k}$ factor comes from the fact that each perceptron is connected only to n/k inputs. For n and $n/k \to \infty$, $\overline{R}$ is still given by eq. 11.16, with q substituted for $q_u\sqrt{k}$. In other words,

$$\overline{R} = \text{erf}(q_u\sqrt{k}\sqrt{2\alpha}) \tag{12.5}$$

To specify the value of q_u, we first remind the reader that $q_u/\sqrt{n/k}$ reflects simply the correlation between the state of each input node and the state of the output node. We use the following intuitive argument to calculate q_u. Let $\mathbf{x}^l$ be a positive example with a target internal representation $\mathbf{Ir}^*(\mathbf{x}^l)$. If $\mathbf{Ir}^*(\mathbf{x}^l)$ has at least one of its components set to -1, then $-\mathbf{x}^l$ is also a positive example. Under the uniform distribution D, $\mathbf{x}^l$ and $-\mathbf{x}^l$ are equally probable to occur in the training set and so, their combined contribution to q_u is null. We are left with the positive examples for which $\mathbf{Ir}^*(\mathbf{x}^l)$ has all its components set to $+1$, and with the negative examples for which, obviously, $\mathbf{Ir}^*(\mathbf{x}^l)$ has all its components set to -1. It is easy to see that the contribution of these examples to q_u is exactly q (eq. 11.18). This is so because, for these examples, $f^*(\mathbf{x}^l) = g_j^*(\mathbf{x}^l)$ and $f^*(-\mathbf{x}^l) = g_j^*(-\mathbf{x}^l)$ $(j = 1, \ldots, k)$. Thus, q_u is given by

$$q_u = \frac{2}{2^k} \times q = \frac{2}{2^k} \times \frac{1}{\sqrt{2\pi}} \tag{12.6}$$

where $2/2^k$ is the probability of drawing an example $\mathbf{x}^l$ for which $\mathbf{Ir}^*(\mathbf{x}^l)$ has all its components set to $+1$ or all set to -1.

Substituting this value of q_u in eq. 12.5 , we get

$$\overline{R} = \text{erf}(\sqrt{\frac{\alpha}{\pi}}\frac{\sqrt{k}}{2^{k-1}}) \tag{12.7}$$

With that, eq. 12.3 reads

$$G(\alpha) = 1 - \frac{1}{\pi}\cos^{-1}\left(\text{erf}(\sqrt{\frac{\alpha}{\pi}}\frac{\sqrt{k}}{2^{k-1}})\right) \tag{12.8}$$

Comparing this to the case of single binary perceptrons (eq. 11.20), we see that only about a fraction of $1/2^{k-1}$ of the training examples contributes actually to the learning process. This is not due to the inefficiency of the clipped Hebb rule but simply due to the fact that, for most of the positive examples, there is no correlation whatsoever between the state of the output node and the state of the input nodes. In fact, we could use *only* the negative examples in the learning process without any significant loss! It is very likely that any learning algorithm for the union will experience the same difficulty.

We turn our attention now to the overall generalization rate of the network, $G_T(\alpha)$. This is defined as the probability that the hypothesis network agrees with the target network on a new random example $\mathbf{x}$ drawn according to D.

Let $G_T^+(\alpha)$ and $G_T^-(\alpha)$ denote the generalization rates for the positive and negative examples, respectively. The hypothesis network will classify correctly a random negative example if and only if each of its perceptrons does so. That is, for a negative example,

$$G_T^-(\alpha) = \prod_{j=1}^{k} G(\alpha) = G(\alpha)^k$$

The probability that the hypothesis network will classify correctly a positive example depends on its target internal representation, *i.e.* on how many target perceptrons classify this example as positive/negative. Let us consider a positive example $\mathbf{x}$ that is classified positive by r target perceptrons, say $g_1^*(\mathbf{x}) = 1, \ldots, g_r^*(\mathbf{x}) = 1$, and negative by the remaining $k - r$ target perceptrons, $g_{r+1}^*(\mathbf{x}) = -1, \ldots, g_k^*(\mathbf{x}) = -1$. The hypothesis network can fail to classify this example correctly only if

$$g_j(\mathbf{x}) \neq g_j^*(\mathbf{x}) \quad \text{for } j = 1, \ldots, r$$

and

$$g_j(\mathbf{x}) = g_j^*(\mathbf{x}) \quad \text{for } j = r+1, \ldots, k.$$

This can happen with a probability

$$(1 - G)^r G^{k-r}.$$

Taking into account the probability that a positive example is classified positive by r target

perceptrons, $G_T^+(\alpha)$ can be written as

$$G_T^+(\alpha) = \sum_{r=1}^{k} \frac{\binom{k}{r}}{2^k - 1} \left(1 - (1-G)^r G^{k-r}\right)$$

The overall generalization rate is thus given by

$$G_T(\alpha) \;=\; \frac{1}{2^k} \times G_T^-(\alpha) + (1 - \frac{1}{2^k}) \times G_T^+(\alpha) \tag{12.9}$$

where $1/2^k$ is the probability that a random drawn example is a negative example and $1 - 1/2^k$ is the probability that a random drawn example is a positive example.

After few manipulations, eq. 12.9 reduces to

$$G_T(\alpha) = 1 - \frac{1}{2^{k-1}} \left(1 - G(\alpha)^k\right) \tag{12.10}$$

The generalization rate $G(\alpha)$ (eq. 12.8), and the overall generalization rate $G_T(\alpha)$ (eq. 12.10) are plotted in fig. 12.2, for different values of k. Also shown there are the results of the numerical simulations. Again, we see that the agreement with the theory is excellent, even for moderate values of n. For small values of k, $G(\alpha)$ converges exponentially to 1 as a function of the number of training examples. But as k increases, $G(\alpha)$ drops very rapidly. On the other hand, the overall generalization, $G_T(\alpha)$, increases very rapidly with k. The latter is due to the fact that the *default* generalization, *i.e.* with no learning at all ($G(\alpha) = 1/2$), increases very rapidly with k.

Finally, it is easy to extend the results of this section to the case where the perceptrons are not connected to the same number of inputs. Assume that perceptron g_j^* is connected to n/k_j inputs, Then eqs. 12.7, 12.8, and 12.10 will read:

$$\overline{R_j} = \mathrm{erf}(\sqrt{\frac{\alpha}{\pi}} \frac{\sqrt{k_j}}{2^{k-1}}) \tag{12.11}$$

$$G_j(\alpha) = 1 - \frac{1}{\pi} \cos^{-1} \left(\mathrm{erf}(\sqrt{\frac{\alpha}{\pi}} \frac{\sqrt{k_j}}{2^{k-1}}) \right) \tag{12.12}$$

$$G_T(\alpha) = 1 - \frac{1}{2^{k-1}} \left(1 - \prod_{j=1}^{k} G_j(\alpha)\right) \tag{12.13}$$

This situation is illustrated in fig. 12.3, for $k = 3$.

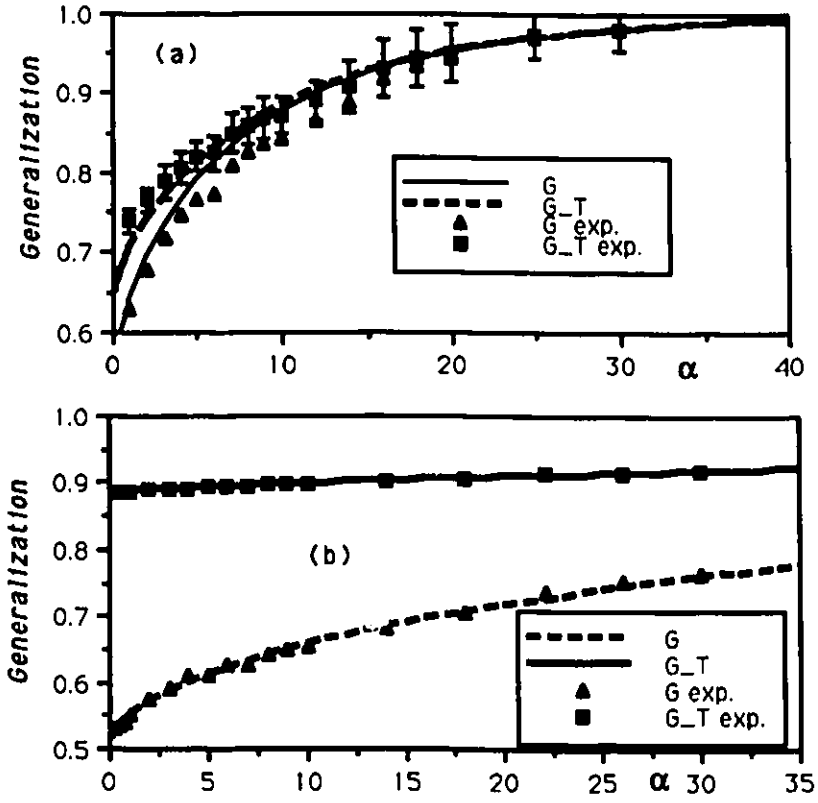

Figure 12.2: Learning a union of k nonoverlapping binary perceptrons connected to the same number of inputs. (a) $k = 2$, and (b) $k = 4$. Shown are the average generalization rate of one perceptron in the net, G, and the overall generalization rate, G_T. The points are the results of the simulations for $n = 100$. Each point denotes an average over 25 different training samples. The error bars, shown only for one curve for clarity, denote the standard deviations.

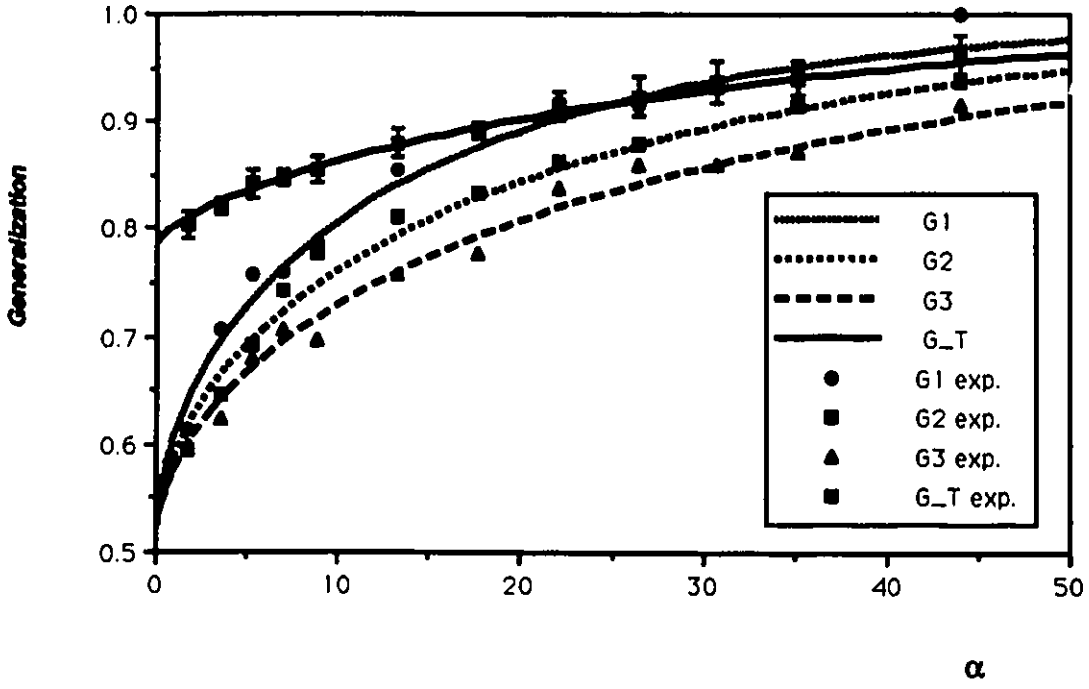

Figure 12.3: Learning a union of 3 nonoverlapping binary perceptrons connected respectively to n/k_1, n/k_2, and n/k_3 input units, where $k_1 = 4.45098$, $k_2 = 3.02666$, and $k_3 = 2.2475$. Shown are the generalization of each perceptron G_i $(i = 1, 2, 3)$, and the overall generalization G_T. The points are the results of the simulations for $n = 227$. Each point denotes an average over 25 different training samples. The error bars, shown only for one curve for clarity, denote the standard deviations.

12.4 Learning a Two-layer Network of Nonoverlapping Binary Perceptrons

Let us assume that the target function (network) f^* is a two-layer network of k nonoverlapping binary perceptrons:

$$f^* = \mathrm{sgn}\left(\sum_{j=1}^{k} g_j^*\right)$$

In other words, the output of the target network computes a majority function. This is the so-called Nonoverlapping Committee Machine [70, 91]. We assume, w.l.o.g., that k is odd. Then, a negative (positive) example is classified negative (positive) by at least $(k+1)/2$ target perceptrons. Assume again, for simplicity, that each perceptron is connected to the same number of inputs, n/k.

Let $G_j(\alpha) \equiv G$ and $R_j \equiv R$ be defined as in the previous section. $G(\alpha)$ is still related to the overlap by eq. 11.12

$$G(\alpha) = 1 - \frac{1}{\pi}\cos^{-1}(R) \tag{12.14}$$

Let us again write

$$P(y_i^l = +1) = \frac{1}{2} + \frac{q_m}{\sqrt{n/k}} \quad \text{as } n/k \to \infty.$$

for some q_m.

For n and $n/k \to \infty$, R is still given by eq. 11.16, with q substituted for $q_m\sqrt{k}$. That is.

$$R = \mathrm{erf}(q_m\sqrt{k}\sqrt{2\alpha}) \tag{12.15}$$

Now, we determine the value of q_m. Assume that x_i is connected to perceptron g_j^*. First, we note that if $g_j^*(\mathbf{x}^l) = -1$, then $P(x_i^l = +1) < P(x_i^l = -1)$ (since we have assumed, w.l.o.g., that $J_i^* = 1$). Likewise, if $g_j^*(\mathbf{x}^l) = +1$, then $P(x_i^l = -1) < P(x_i^l = +1)$. Based on this observation, it is easy to see that the contributions to q_m from the following two sources will be negative:

- Positive examples $\mathbf{x}^l$ for which $g_j^*(\mathbf{x}^l) = -1$, and
- Negative examples $\mathbf{x}^l$ for which $g_j^*(\mathbf{x}^l) = +1$.

Similarly, the contributions to q_m from the following two sources will be positive:

- Positive examples $\mathbf{x}^l$ for which $g_j^*(\mathbf{x}^l) = +1$, and
- Negative examples $\mathbf{x}^l$ for which $g_j^*(\mathbf{x}^l) = -1$.

Moreover, the contribution in absolute value, of each of the these four possibilities to q_m is exactly q (eq. 11.18). Taking into account the probability that each of the the four possibilities mentioned above does occur, we get

$$\begin{aligned} q_m &= \Big(P(f^*(\mathbf{x}^l) = 1 \;\&\; g_j^*(\mathbf{x}^l) = 1) + P(f^*(\mathbf{x}^l) = -1 \;\&\; g_j^*(\mathbf{x}^l) = -1)\Big) \times q \\ &- \Big(P(f^*(\mathbf{x}^l) = 1 \;\&\; g_j^*(\mathbf{x}^l) = -1) + P(f^*(\mathbf{x}^l) = -1 \;\&\; g_j^*(\mathbf{x}^l) = 1)\Big) \times q \end{aligned} \tag{12.16}$$

Now,

$$\begin{aligned} P(f^*(\mathbf{x}^l) = 1 \;\&\; g_j^*(\mathbf{x}^l) = 1) &= P(f^*(\mathbf{x}^l) = 1) \times P(\, g_j^*(\mathbf{x}^l) = 1 \mid P(f^*(\mathbf{x}^l) = 1) \\ &= \frac{1}{2} \frac{\sum_{r=\frac{k+1}{2}}^{k} \binom{k}{r} \frac{r}{k}}{\sum_{r=\frac{k+1}{2}}^{k} \binom{k}{r}} \\ &= \frac{1}{2}\left(\frac{1}{2} + \frac{\binom{k-1}{\frac{k-1}{2}}}{2^k}\right) \end{aligned}$$

Similarly,

$$P(f^*(\mathbf{x}^l) = -1 \;\&\; g_j^*(\mathbf{x}^l) = -1) = \frac{1}{2}\left(\frac{1}{2} + \frac{\binom{k-1}{\frac{k-1}{2}}}{2^k}\right)$$

$$P(f^*(\mathbf{x}^l) = 1 \;\&\; g_j^*(\mathbf{x}^l) = -1) = \frac{1}{2}\left(\frac{1}{2} - \frac{\binom{k-1}{\frac{k-1}{2}}}{2^k}\right)$$

$$P(f^*(\mathbf{x}^l) = -1 \;\&\; g_j^*(\mathbf{x}^l) = 1) \;=\; \frac{1}{2}\left(\frac{1}{2} - \frac{\binom{k-1}{\frac{k-1}{2}}}{2^k}\right)$$

After few manipulations, this yields

$$q_m = 2\frac{\binom{k-1}{\frac{k-1}{2}}}{2^k}q = 2\frac{\binom{k-1}{\frac{k-1}{2}}}{2^k}\frac{1}{\sqrt{2\pi}} \tag{12.17}$$

Finally, we look at the overall generalization rate of the network, $G_T(\alpha)$. Deriving an expression for $G_T(\alpha)$ for an arbitrary k is a difficult task. In the following, we concentrate on the two limiting cases: the case $k = 3$ and the case of large k. Note that the learning curves for an arbitrary k will lay in between those corresponding to the two limiting cases.

12.4.1 The Case of $k = 3$

For $k = 3$, eq. 12.17 reduces to

$$q_m = \frac{1}{2}\frac{1}{\sqrt{2\pi}}.$$

Putting these values back in eqs. 12.15 and 12.14, we get

$$\overline{R} = \mathrm{erf}(\frac{\sqrt{3}}{2}\sqrt{\frac{\alpha}{\pi}}) \tag{12.18}$$

$$G(\alpha) = 1 - \frac{1}{\pi}\cos^{-1}\left(\mathrm{erf}(\frac{\sqrt{3}}{2}\sqrt{\frac{\alpha}{\pi}})\right) \tag{12.19}$$

Comparing this to the case of single binary perceptrons (eq. 11.20), we see that 3/4 of the training examples contribute to the learning process, and that $G(\alpha)$ still converges exponentially to 1.

The probability that the hypothesis network will classify correctly a new positive (negative) example depends on its target internal representation, more precisely on how many target perceptrons classify this example as positive/negative. For a positive example $\mathbf{x}$, we have two possibilities:

1) $\mathbf{x}$ is classified positive by 2 target perceptrons, say by g_1^*, g_2^*, and negative by the remaining target perceptron, g_3^*. The hypothesis network can fail to classify this example correctly only if

$$g_1(\mathbf{x}) \neq g_1^*(\mathbf{x}), \quad g_2(\mathbf{x}) \neq g_2^*(\mathbf{x}), \quad g_3(\mathbf{x}) = g_3^*(\mathbf{x})$$

or

$$g_1(\mathbf{x}) \neq g_1^*(\mathbf{x}), \quad g_2(\mathbf{x}) = g_2^*(\mathbf{x}), \quad g_3(\mathbf{x}) = g_3^*(\mathbf{x})$$

or

$$g_1(\mathbf{x}) = g_1^*(\mathbf{x}), \quad g_2(\mathbf{x}) \neq g_2^*(\mathbf{x}), \quad g_3(\mathbf{x}) = g_3^*(\mathbf{x})$$

or

$$g_j(\mathbf{x}) \neq g_j^*(\mathbf{x}) \quad \text{for } j = 1, 2, 3$$

This can happen with a probability

$$G(\alpha)[1 - G(\alpha)]^2 + 2G(\alpha)^2[1 - G(\alpha)] + [1 - G(\alpha)]^3$$

2) $\mathbf{x}$ is classified positive by all 3 target perceptrons. The hypothesis network can fail to classify this example correctly only if at least two of its perceptrons fail to do so. This can happen with a probability

$$3G(\alpha)[1 - G(\alpha)]^2 + [1 - G(\alpha)]^3$$

The same argument holds for negative examples. Taking into account the probability that an example is classified positive (negative) by r target perceptrons, we get

$$\begin{aligned} G_T(\alpha) = 1 &- \frac{3}{2}G(\alpha)^2[1 - G(\alpha)] \\ &- \frac{3}{2}G(\alpha)[1 - G(\alpha)]^2 \\ &- [1 - G(\alpha)]^5 \end{aligned} \tag{12.20}$$

The analytical expressions for $\overline{R}$, $G(\alpha)$, and $G_T(\alpha)$ are plotted in fig. 12.4 along with the simulation results. Again, the agreement is excellent. One can also see that $G_T(\alpha)$ tends exponentially to perfect generalization.

The arguments of this section may be used, in principle, to derive an expression for $G_T(\alpha)$ for any value of k. However, it becomes too complicated to follow for $k \geq 7$. Thus, we will look simply at the other end of the spectrum, *i.e.* large k.

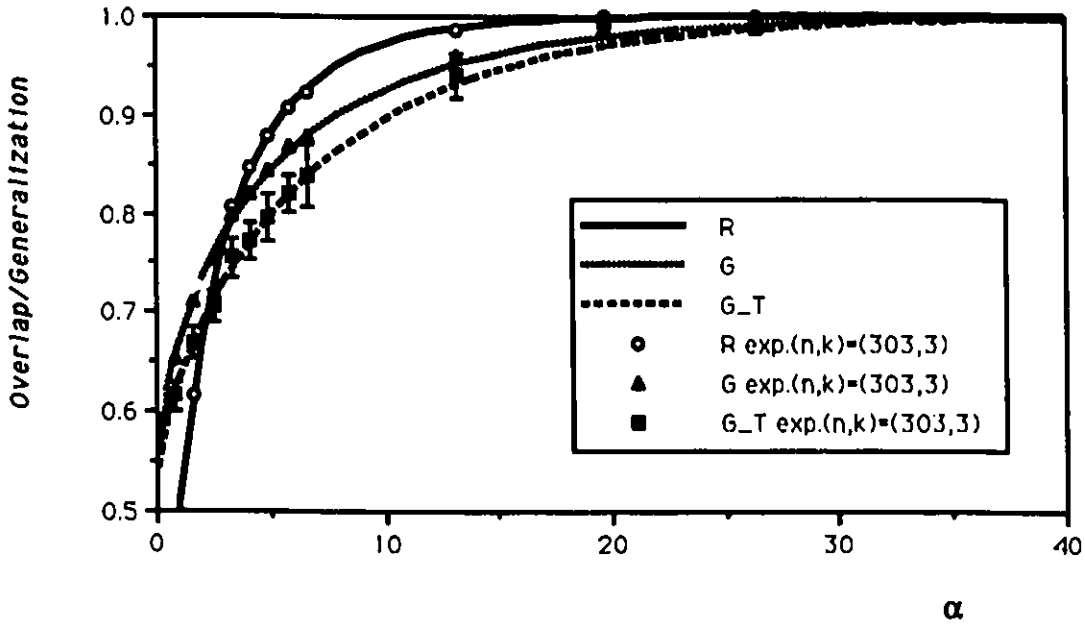

Figure 12.4: Learning a two-layer network of 3 nonoverlapping binary perceptrons connected to the same number of inputs. Shown are the average overlap R, the generalization rate of each perceptron G, and the overall generalization rate G_T. The points are the results of the simulations for $n = 303$. Each point denotes an average over 25 different training samples. The error bars, shown only for one curve for clarity, denote the standard deviations.

12.4.2 The Case of Large k

Here we are interested in the case where $k \to \infty$ (but n is still larger than k such that $n/k \to \infty$). In this case, eq. 12.17 reduces to

$$q_m \sim \frac{2}{\sqrt{2\pi k}} \frac{1}{\sqrt{2\pi}}. \tag{12.21}$$

Putting this value back in eqs. 12.15 and 12.14, we get

$$\overline{R} = \mathrm{erf}(\sqrt{\frac{2}{\pi}}\sqrt{\frac{\alpha}{\pi}}) \tag{12.22}$$

$$G(\alpha) = 1 - \frac{1}{\pi}\cos^{-1}\left(\mathrm{erf}(\sqrt{\frac{2}{\pi}}\sqrt{\frac{\alpha}{\pi}})\right) \tag{12.23}$$

Again, comparing this to the case of single binary perceptrons (eq. 11.20), we see that a fraction $2/\pi$ ($\simeq 0.63$) of the training examples contribute to the learning process, and that $G(\alpha)$ still converges exponentially to 1.

To determine the overall generalization, let $\mathbf{x}$ be a random input and let

$$a = \sum_{j=1}^{k} g_j^*(\mathbf{x}) \qquad b = \sum_{j=1}^{k} g_j(\mathbf{x})$$

For different inputs $\mathbf{x}$, a and b are correlated Gaussian variables with

$$\overline{a} = \overline{b} = 0$$

$$\overline{a^2} = \overline{b^2} = k$$

and

$$\overline{ab} = k \times \rho$$

where ρ, the overlap between $\mathbf{Ir}^*(\mathbf{x})$ and $\mathbf{Ir}(\mathbf{x})$, is given by

$$\rho = \frac{\sum_{j=1}^{k} g_j^*(\mathbf{x}) g_j(\mathbf{x})}{k}$$

By definition,

$$G_T(\alpha) \equiv P(ab > 0)$$

which, as for a single perceptron, depends only on the average overlap $\overline{\rho}$. So, $G_T(\alpha)$ is again given by eq. 11.12

$$G_T(\alpha) = 1 - \frac{1}{\pi}\cos^{-1}(\overline{\rho})$$

We now evaluate $\overline{\rho}$ (remember, the overbar denotes the average with respect to the training set). For that, let

$$h_j(\mathbf{x}) = g_j^*(\mathbf{x}) g_j(\mathbf{x}) \quad j = 1, \ldots, k$$

Then,

$$P(h_j(\mathbf{x}) = +1) \;=\; G(\alpha) \qquad \overline{h_j(\mathbf{x})} \;=\; 2G(\alpha) - 1$$

This yields

$$\begin{aligned} \overline{\rho} &= 2G(\alpha) - 1 \\ &= 2(1 - \frac{1}{\pi}\cos^{-1}(\overline{R})) - 1 \\ &= 1 - \frac{2}{\pi}\cos^{-1}(\overline{R}) \end{aligned} \tag{12.24}$$

So, the overall generalization is given by

$$\begin{aligned} G_T(\alpha) &= 1 - \frac{1}{\pi}\cos^{-1}(\overline{\rho}) \\ &= 1 - \frac{1}{\pi}\cos^{-1}\left(2G(\alpha) - 1\right) \qquad (12.25) \\ &= 1 - \frac{1}{\pi}\cos^{-1}\left(1 - \frac{2}{\pi}\cos^{-1}(\overline{R})\right) \qquad (12.26) \end{aligned}$$

It is interesting to see that, for large k, the generalization rate of a majority of nonoverlapping perceptrons behaves as that of a single perceptron, with a modified overlap $1 - \frac{2}{\pi}\cos^{-1}(\overline{R})$. Eq. 12.26 has been also derived in [70], using a different method.

The analytical expressions for $G(\alpha)$ and $G_T(\alpha)$ are plotted in fig. 12.5 along with the simulation results. There is a noticeable deviation from the theoretical predictions; the reason for this is that, in the simulations, k and n/k are not sufficiently large. On the other hand, one can see that as k and n/k become larger, the simulations results tend towards the theoretical curves. One can also see that $G_T(\alpha)$ tends exponentially to perfect generalization.

The nonoverlapping Committee Machine with binary weights has been studied extensively using the statistical mechanics approach [70, 91]. This approach assumes a stochastic training algorithm, similar to a finite Monté Carlo process, that leads at long times to a Gibbs distribution of weights [94]. Under this assumption, it is found that a phase transition to perfect generalization does occur at a critical value of α. Thus, such training algorithms

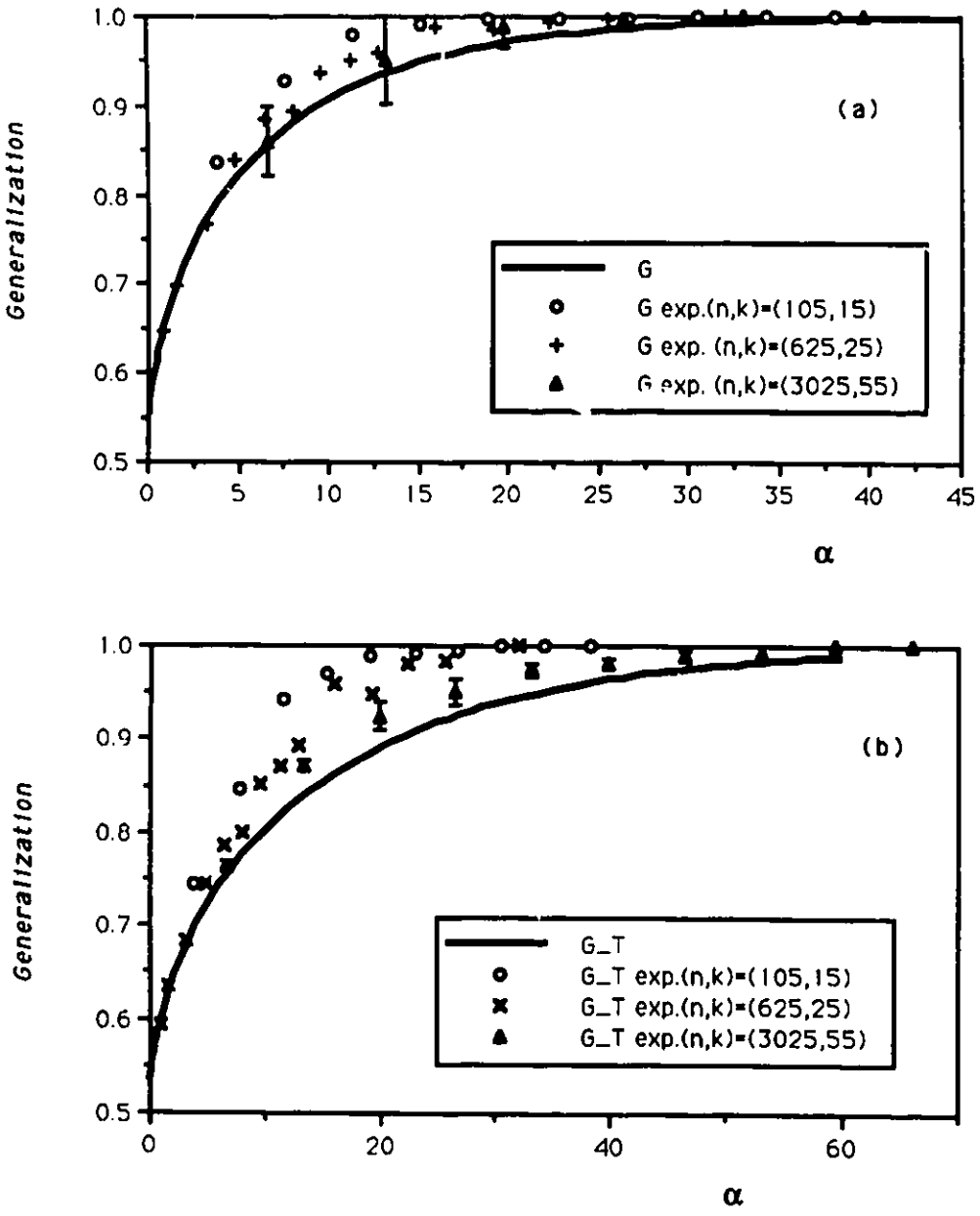

Figure 12.5: Learning a two-layer network of k nonoverlapping binary perceptrons connected to the same number of inputs. The case of large k. Shown are (a) the generalization rate of each perceptron G, and (b) the overall generalization rate G_T. The points are the results of the simulations for the indicated values of (n,k). Each point denotes an average over 25 different training samples. The error bars, shown only for one curve for clarity, denote the standard deviations.

have a slightly better sample complexity than the clipped Hebb rule. However, the time complexity of the clipped Hebb rule is only $O(n \times m)$, whereas stochastic training algorithms generally require prohibitively long convergence times.

12.5 Extension to Multilayer Networks of Nonoverlapping Binary Perceptrons

Let f^* be a layered network of nonoverlapping binary perceptrons (fig. 12.1). Let H denote the number of hidden layers and k_h the number of perceptrons in layer h ($h = 1, \ldots, H$). Assume that the number of nodes in layer $h-1$ is much greater than the number of nodes in layer h. That is

$$n \to \infty$$

$$n/k_1 \to \infty$$

$$k_{h-1}/k_h \to \infty \quad h = 2, \ldots, H$$

Assume, for simplicity, that all perceptrons in the same layer are connected to the same number of nodes in the previous layer. Let $G_h(\alpha)$ denotes the generalization rate of a perceptron (hidden unit) in layer h.

Using the arguments of the previous section that led to eq. 12.21, one can show that each hidden layer will contribute a factor $2/\sqrt{2\pi}$ to q_m. Thus,

$$q_m\sqrt{k_1} = \left(\frac{2}{\sqrt{2\pi}}\right)^H \frac{1}{\sqrt{2\pi}} \tag{12.27}$$

With this, eq. 12.22 reads now

$$\overline{R} = \mathrm{erf}\left(\left(\sqrt{\frac{2}{\pi}}\right)^H \sqrt{\frac{\alpha}{\pi}}\right) \tag{12.28}$$

Also, $G_1(\alpha)$ is still given by

$$G_1(\alpha) = 1 - \frac{1}{\pi}\cos^{-1}(\overline{R}) \tag{12.29}$$

Applying the arguments that led to eq. 12.25 recursively, that is from one layer to the next, we get

$$G_h(\alpha) = 1 - \frac{1}{\pi}\cos^{-1}\left(2G_{h-1}(\alpha) - 1\right) \qquad h = 2, \ldots, H. \tag{12.30}$$

$$G_T(\alpha) = 1 - \frac{1}{\pi}\cos^{-1}\left(2G_H(\alpha) - 1\right) \tag{12.31}$$

Finally, we note that eq. 12.28 can be written as

$$\overline{R} = \mathrm{erf}(\sqrt{\alpha_{\mathrm{eff}}/\pi}) \tag{12.32}$$

where

$$\alpha_{eff} = \left(\frac{2}{\pi}\right)^H \alpha$$

Compared to the single binary perceptron case (eq. 11.19), α_{eff} reflects the *effective* number of examples contributing to the learning process. This effective number decreases as $(2/\pi)^H$. This may explain the observation made in [70] that the critical value of α at which the phase-transition occurs scales as $(\pi/2)^H$.

12.6 Conclusion

We have investigated the clipped Hebb rule for learning different networks of nonoverlapping binary perceptrons under the uniform distribution. We have calculated exactly the learning curves of this rule in the limit $n \to \infty$, where the average behavior becomes the typical one.

Our results indicate that the clipped Hebb rule does indeed learn this class of networks. Specifically, the generalization rates converge extremely rapidly, often exponentially, to perfect generalization as a function of the number of training examples. The analytic expression of the learning curves are in excellent agreement with the numerical simulations. These results are very encouraging given the simplicity of the learning rule. In particular, this shows that simple neural networks with binary weights may be learnable under simple distributions of examples.

As we have mentioned earlier, the clipped Hebb rule produces hypotheses that are not necessarily consistent with all the training examples, but that nonetheless have very good generalization ability. These type of algorithms are called "inconsistent algorithms"[72]. Such algorithms are very important because, in many situations, there is no hypothesis consistent with all the training examples. This may be due to the intrinsic difficulty of the problem or to the examples being noisy. As we saw in the previous chapter, the clipped Hebb rule in particular is very robust with respect to random classification noise.

Finally, we leave the reader with the following important problem. Throughout this chapter, we have assumed that the architecture is known in advance. But whereas this is in line with most of the neural network research, it is hardly justifiable in practice. Is there a CHR-type algorithm that can learn networks of nonoverlapping binary perceptrons in terms of both the weight values and the network's architecture? Note that such algorithm can still use the clipped Hebb rule to determine the weight values.

Chapter 13

Conclusion and Open Problems

In this thesis, we have investigated different ways to overcome the difficulties of learning in neural networks. The main conclusions can be summarized as follows:

- Part 2:

  Usually in real applications the complexity of a classification problem is not known *a priori*. So identifying the appropriate network to obtain good performance is of fundamental importance. The approach we adopted in Part 2 is to let the learning algorithm determine the appropriate network during the training process. This approach has the additional advantage of avoiding the NP-completeness trap due to the fixed architecture constraint. As a guideline for choosing the appropriate network we adopted *Occam's Razor* principle: choose the *smallest* network (fewest hidden units and weights) that memorizes exactly the training sample. We came up with two constructive algorithms that build two-layer and tree-like neural networks. These algorithms exhibit two different trends, in terms of the *size of the network* and the *generalization ability*, depending on whether or not they are able to learn the target function. An interesting question is whether other constructive algorithms exhibit the same kind of trends (or any trend at all). If so, these trends can be used as good indicators in real applications.

- Part 3:

  The PAC framework is widely accepted as a reasonable model of learning. By restrict-

ing the distribution generating the training examples, the connectivity of the network, and/or the values of the weights, we were able to identify new important classes of neural networks that are *efficiently* PAC learnable. The simulation results on real data sets, reported in chapter 7, show that simple neural concepts such as halfspace intersections and neural decision lists are not only important from the theoretical point of view, but also in practice. An interesting problem is to extend the results of Part 3 to more general networks and/or distributions of examples. Another interesting direction is to allow the learning algorithm to ask questions, *e.g.* membership queries. We have shown recently that the general class of nonoverlapping perceptron networks (real valued weights and thresholds) are PAC learnable from examples and membership queries under an arbitrary distribution [42]. Can this be generalized to networks with partial overlap?

- Part 4:

  We were able to calculate the *learning curves* of the *clipped Hebb rule* when learning different neural networks with binary weights. The analytical expressions are in very good agreement with the numerical simulations. We assumed that the architectures are known *in advance*. But whereas this is in line with most of the neural network research, it is hardly justifiable in practice. Is there a clipped Hebb rule type algorithm that can learn networks of nonoverlapping binary perceptrons in terms of both the weight values and the network's architecture? More generally, all the results obtained with the average case approach are, so far, limited to fixed architectures. It would be interesting to apply this approach to the analysis of neural algorithms that modify the architecture of the network during learning.

Bibliography

[1] Abramowitz M. and Stegun I.A., Handbook of Mathematical Functions (Dover Publ.), 1972.

[2] Angluin D., "Queries and Concept Learning", *Machine Learning*, Vol. 2, p. 319, 1988.

[3] Angluin D., Hellerstein L., and Karpinski M., " Learning read-once formulas with queries". Technical Report N. UBC/CSD 89/528, Computer Science Division, University of California Berkeley, 1989. To appear in *J. ACM*.

[4] Angluin D. and Kharitonov M., "When won't membership queries help?", in *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, p. 444, 1991.

[5] Bahadur R.,"Some Approximations to the Binomial Distribution Function", *Annals Math. Stat.*, Vol.31, p. 43, 1960.

[6] Barkai E., Hansel D., and Kanter I., "Statistical mechanics of multilayer neural networks", *Physical Review Letters*, Vol. 65, p. 2312, 1990.

[7] Barkai E. and Kanter I., "Storage capacity of a multilayer network with binary weights", *Europhysics Letters*, Vol. 14, p. 107, 1991.

[8] Bartlett P.L. and Williamson R.C., "Investigating the Distribution assumptions in the PAC Learning Model", in *Proc. of the 4th Workshop on Computational Learning Theory*, Morgan Kaufman, p. 24, 1991.

[9] Baum E.B., "On learning a union of halfspaces", *Journal of Complexity*, Vol. 6, p. 67, 1990.

[10] Baum E.B., "The Perceptron Algorithm if Fast for Nonmalicious Distributions", *Neural Computation*, VOl. 2, p. 248, 1990.

[11] Baum E.B., "A polynomial time algorithm that learns two hidden unit nets", *Neural Computation*, Vol. 2, p. 510, 1990.

[12] Baum E.B., "Neural net algorithms that learn in polynomial time from examples and queries", *IEEE Trans. on Neural Networks*, vol. 2, p. 5, 1991.

[13] Baum E.B. and Haussler D., "What size net gives valid generalization", *Neural Computation*, Vol. 1, p. 151, 1989.

[14] Blum A. and Rivest R.L., "Training a 3-node neural network is NP-complete", in *Proc. of the 1st Workshop on Computational Learning Theory*, Morgan Kaufman, p. 9, 1988.

[15] Blumer A., Ehrenfeucht A., Haussler D., and Warmuth M., "Learnability and the Vapnik-Chervonenkis dimension", *J. ACM*, Vol. 36, p. 929, 1989.

[16] Blumer A., Ehrenfeucht A., Haussler D., and Warmuth M., "Occam's Razor", *Inf. Proc. Lett.*, Vol. 24, p. 377, 1987.

[17] Brady M., Raghavan R., and Slawny J., "Gradient descent fails to separate", in *Proc. 2nd Int. Conf. Neural Networks*, p. 649, 1988.

[18] Breiman L., Friedman J.H., Olshen R.A., and Stone C.J., Classification and Regression Trees (Wadsworth, Belmont CA), 1984.

[19] Bshouty N.H., Hancock T.R., and Hellerstein L., "Learning boolean read-once formulas with arbitrary symmetric and constant fan-in gates", in *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, p. 1, 1992.

[20] Chauvin Y., "A Back-Propagation Algorithm with Optimal Use of Hidden Units", in *Advances in Neural Information Processing Systems*, Vol. 1, p. 519, 1989.

[21] Chvatal V., "A Greedy Heuristic for the Set-Covering Problem", *Math. Oper. Res.* Vol. 4, p. 3, 1979.

[22] Cover T.M., "Goemetrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition", *IEEE Transac. Electron. Comput.* Vol. 14, p. 326, 1965.

[23] Dantzig G.B., Linear Programming and Extensions (Princeton NJ: Princeton University Press), 1963.

[24] Dayhoff J., Neural Network Architectures (Van Nostrand Reinhold), 1990.

[25] Denker J., Schwartz D., Wittner B., Solla S., Hopfield J., Howard R., and Jackel L., " Automatic Learning, Rule Extraction and Generalization", *Complex Systems*, Vol. 1, p. 877, 1987.

[26] Dorffner G., "An Introduction to Neurocomputing and its Possible Role in AI", *Lecture Notes in Artificial Iintelligence*, Vol. 617, p. 440, 1992.

[27] Duda R.O. and Hart P.E., Pattern Classification and Scene Analysis (New-York: Wiley), 1973.

[28] Fahlman S.E. and Lebière C., "The Cascade-Correlation Learning Architecture", in *Advances in Neural Information Processing Systems*, Vol. 2, p. 524, 1990.

[29] Frean M., "The Upstart Algorithm: A Method for Constructing and Training Neural Networks", *Neural Computation*, Vol.2, p. 198, 1990.

[30] Gallant S.I., "Perceptron-Based Learning Algorithms", *IEEE Trans. Neural Networks*, Vol. 1, p. 179, 1990.

[31] Gary M.R. and Johnson D.S., Computers and Intractability, A Guide to the Theory of NP-Completeness (New-York: Freeman), 1979.

[32] Kamp Y. and Hasler M., Recursive Neural Networks for Associative Memory (John Wiley & Sons), 1990

[33] Gibson G.J. and Cowan C.F.N., "On the Decision Regions of Multilayer Perceptrons", *Proceedings of The IEEE*, Vol. 78, p. 1590, 1990.

[34] Goldman S., Kearns M., and Schapire R., "Exact identification of circuits using fixed points of amplification functions", in *Proceedings of the 31st Symposium on Foundations of Computer Science*, 1990.

[35] Golea M., Marchand M., and Hancock T.R., "On Learning μ-Perceptron Networks with Binary Weights", *Advances in Neural Information Processing Systems*, Vol. 5, p. 591, 1993.

[36] Golea M. and Marchand M., "Average Case Analysis of the Clipped Hebb Rule for Nonoverlapping Perceptron Networks", to appear in *Computational Learning Theory*, 1993.

[37] Golea M. and Marchand M., "A Growth Algorithm for Neural Network Decision Trees", *Europhys. Lett.*, Vol. 12, p. 205, 1990.

[38] Greer R., "Trees and Hills: Methodology for Maximizing Functions of Systems of Linear Relations", *Annals of Discrete Mathematics* , Vol. 22, 1984.

[39] Gyorgyi G., "First-order transition to perfect generalization in a neural network with binary synapses", *Phys. Rev. A*, Vol. 41, p. 7097, 1990.

[40] Hagerup T. and Rub C., "A Guided Tour to Chernoff Bounds", *Info. Proc. Lett.*, Vol. 33, p. 305, 1989.

[41] Hancock T.R., Learning 2μ DNF formulas and $k\mu$ decision trees. In *The 1991 Workshop on Computational Learning Theory*, p. 199, 1991.

[42] Hancock T.R., Golea M., and Marchand M. "Learning Nonoverlapping Perceptron Networks from Examples and Membership Queries". TR-26-91, Center for Research in Computing Technology, Harvard University. to appear in *Machine Learning*.

[43] Hancock T.R. and Hellerstein L., "Learning read-once formulas over fields and extended bases", in *The 1991 Workshop on Computational Learning Theory*, p. 326, 1991.

[44] Hancock T. and Mansour Y.,"Learning Monotone $k\mu$-DNF Formulas on Product Distributions", in *Proc. of the 4st Workshop on Computational Learning Theory*, p. 179, 1991.

[45] Haussler D., "Quantifying Inductive Bias: AI Learning Algorithms and Valiant's Learning Framework", *Artif. Intell.*, Vol. 36, p. 177, 1988.

[46] Haussler D., "Decision Theoretic Generalizations of the PAC Model for Neural Net and Other Learning Applications", *Information and computation*, Vol. 100, p. 78, 1992.

[47] Haussler D., Kearns M., Littlestone N., and Warmuth M., "Equivalence of Models for Polynomial Learnability", *Information and computation*, Vol. 95, p. 129, 1991.

[48] Haussler D., Kearns M., and Schapire R., "Bounds on the Sample Complexity of Bayesian Learning Using Information Theory and the VC Dimension", in *The Fourth Workshop on Computational Learning Theory*, p. 61, 1991.

[49] Holte R.C., "Very Simple Classification Rules Perform Well on Most Data Sets". TR-16-91, Computer Science Depart., University of Ottawa. To appear in *Machine Learning*.

[50] Hopfield J., "Neural Networks and Physical Systems with Emergent collective Computational Abilities", *Proc. Nat. Acad. Sci. USA*, Vol. 79, p. 2554, 1982.

[51] Hyafil L. and Rivest R.L., " Constructing Optimal Binary Decision Trees is NP-complete", *Inform. Process. Lett.*, Vol. 5, p. 15, 1976.

[52] Ibragimov I.A., "On the composition of unimodal distributions", *Theory of Probability and its applications*, Vol. 1, p. 255, 1956.

[53] Johnson D.S., "Approximation Algorithms for Combinatorial Problems", *J. Comput. System Sci.*, Vol. 9, p. 256, 1974.

[54] Johnson D.S. and Preparata F.P., "The Densest Hemisphere Problem", *Theor. Comput. Sci.*, Vol. 6, p. 93, 1978.

[55] Judd S., "On the complexity of loading shallow neural networks", *Journal of Complexity*, Vol. 4, p. 177, 1988.

[56] Karmarkar N., "A new polynomial time algorithm for linear programming", *Combinatorica*, Vol. 4, p. 373, 1984.

[57] Kearns M., " Efficient Noise-Tolerant Learning from Statistical Queries", manuscript 1992.

[58] Kearns M., Li M., Pitt L., and Valiant L., "On the learnability of boolean formulas", in *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, New York, NY, 1987.

[59] Kearns M. and Schapire R.E., " Efficient Distribution-free Learning of Probabilistic Concepts", in *Proceedings of the 31st Symposium on Foundations of Computer Science*, p. 382, 1990.

[60] Kearns M. and Valiant L., "Cryptographic limitations on learning boolean formulae and finite automata", in *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, p. 433, 1989.

[61] Köhler H., Diederich S., Kinzel W., and Opper M., "Learning Algorithm for a Neural Network with Binary Synapses", *Z. Phys. B*, Vol. 78, p. 333, 1990.

[62] Krauth W. and Mézard M., "Storage Capacity of Mémory Networks with Binary Couplings", *J. Phys. France*, Vol. 50, p. 3057, 1989.

[63] Laird P.D., "A Survey of Computational Learning Theory", in *Formal Techniques in Artificial Intelligence*, Banerji R.B. (editor), (North-Holland), p. 173, 1990.

[64] Levin E., Tishby N., and Solla S., "A statistical Approach to Learning and Generalization in Layerd Neural Networks", *Proc. IEEE*, Vol. 78, p. 1568, 1990.

[65] Lin J.H. and Vitter J.S., "Complexity results on learning by neural nets", *Machine Learning*, Vol. 6, p. 211, 1991.

[66] Linsker R., " Self-Organization in a Perceptual Network", *Computer*, Vol. 21, p. 105, 1988.

[67] MacDonald D.R., "On local limit theorems for integer-valued random variables", *Theory of Probability and Statistics Acad. Nauk.*, Vol. 3, p. 607, 1979.

[68] Marchand M., Golea M., and Ruján P., "A convergence Theorem for Sequential Learning in Two-Layer Perceptrons", *Europhys. Lett.*, Vol. 11, p. 487, 1990.

[69] Marchand M. and Golea M., "On Learning Simple Neural Concepts: From Halfspace Intersections to Neural Decision Lists", *Network: Computation in Neural Systems*, Vol. 4, p. 67, 1993 (in press).

[70] Mato G. and Parga N., "Generalization Properties of Multilayerd Neural Networks", *J. Phys. A: Math. Gen.*, Vol. 25, (1992), 5047–5054.

[71] McCulloch W. and Pitts W., "A Logical Calculus of the Ideas Immanent in the Nervous Activity", *Bulletin of Mathematical Biophysics*, Vol. 5, p. 115, 1943.

[72] Meir R. and Fontanari J.F., "Calculation of Learning Curves for Inconsistent Algorithms", *Phys. Rev. A*, Vol. 92, p. 8874, 1992.

[73] Mézard M. and Nadal J.P., "Learning in Feedforward Neural Network: the Tiling Algorithm", *J. Phys. A*, Vol. 22, p. 2191, 1989.

[74] Minsky M. and Papert S., Perceptrons (Cambridge MA: MIT Press, second edition), 1988.

[75] Nadal J.P., "Study of a Growth Algorithm for a Feedforward Network", *Int. J. Neural Systems*, Vol. 1, p. 55, 1989.

[76] Nilsson N.J., Learning Machines: Foundations of Trainable Pattern-Classifying Systems (McGraw-Hill Book), 1965.

[77] Opper M. and Haussler H., "Generalization Performance of Bayes Optimal Classification Algorithm for Learning a Perceptron", *Phys. Rev. Lett.* Vol. 66, p. 2677, 1991.

[78] Opper M., Kinzel W., Kleinz J., and Nehl R., "On the Ability of the Optimal Perceptron to Generalize", *J. Phys. A: Math. Gen.*, Vol. 23, p. L581, 1990.

[79] Pagallo G. and Haussler D., "A greedy method for learning μDNF functions under the uniform distribution". Technical Report UCSC-CRL-89-12, Santa Cruz: Dept. of Computer and Information Science, University of California at Santa Cruz.

[80] Papadimitriou C.H. and Steiglitz K., Combinatorial Optimization (Englewood Cliffs NJ: Prentice-Hall), 1982.

[81] Pitt L. and Valiant L.G., "Computational Limitations on Learning from Examples", *J. ACM*, Vol. 35, p. 965, 1988.

[82] Preparata F.P. and Shamos M.I., Computational Geometry (New-York: Springer), 1985.

[83] Qian N. and Sejnowski T., "Predicting the Secondary Structure of Globular Proteins Using Neural Network Models", *Mol.Biol.*, Vol. 202, p.865, 1988.

[84] Quinlan J.R., "Induction of Decision Trees", *Machine Learning*Vol. 1, p. 81, 1986.

[85] Rivest R.L., 1987 "Learning Decision Lists" *Machine Learning*, Vol. 2, p. 229, 1987

[86] Rosenblatt F., Principles of Neurodynamics (Spartan New York), 1962.

[87] Ruján P. and Marchand M., "Learning by Minimizing Resources in Neural Networks", *Complex Systems*, Vol. 3, p. 229, 1989.

[88] Rumelhart D.E., Hinton G.E., and Williams R.J., "Learning Representations by Back-Propagating Errors", *Nature*, Vol. 323, p. 533, 1986.

[89] Rumelhart D.E. and McClelland J.L., Parallel Distributed Processing (Cambridge MA: MIT Press), 1986.

[90] Schapire R.E., "Learning probabilistic read-once formulas on product distributions", in *Proc. of the 4th Annual Workshop on Computational Learning Theory*, Morgan Kaufman, p. 184, 1991.

[91] Schwarze H. and Hertz J., "Generalization in Large Committee Machine", *Europhys. Lett.* , Vol. 20, p. 375, 1992.

[92] Sejnowski T. and Rosenberg C. "Parallel Networks that Learn to Pronounce English text", *Complex Systems*, Vol. 1, p.145, 1987.

[93] Sethi I.K., "Entropy Nets: From Decision Trees to Neural Networks", *Proc. IEEE*, Vol. 78, p. 1605, 1990.

[94] Seung H., Sompolinsky H., and Tishby N., "Statistical Mechanics of Learning from Examples", *Phys. Rev. A*, Vol. 45, p. 6056, 1992.

[95] Shapiro A.D., Structured Induction and Expert Systems (Wokingham, UK: Addison-Wesley), 1987.

[96] Shvaytser H., "Learnable and Nonlearnable Visual Concepts", *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 12, p. 459, 1990.

[97] Sirat J.A. and Nadal J.P., "Neural Trees: a New Efficient Tool for Classification", *Network*, Vol. 1, p. 423, 1990.

[98] Smale S., "On the Average Number of Steps of the Simplex Method of Linear Programming", *Math. Programm.*, Vol. 27, p. 241, 1983.

[99] Sompolinsky H., Tishby N., and Seung H.S., "Learning from Examples in Large Neural Networks", *Phys. Rev. Lett*, Vol. 65, p. 1683, 1990.

[100] Tesauro G. and Janssens B., "Scaling Relationships in Back-propagation Learning", *Complex Systems*, Vol. 2, p. 39, 1988.

[101] Timothy L.H. and Albrecht R., "Learning Unlearnable Problems with Perceptrons", *Phys. Rev. A*, Vol. 45, (1992), 4102–4110.

[102] Utgoff P.E., "Perceptron Trees: A Case Study in Hybrid Concept Representation", *Connection Science*, Vol. 1, p. 377, 1989.

[103] Valiant L.G., "A theory of the learnable", *Communications of the ACM*, Vol. 27, p. 1134, 1984.

[104] Valiant L.G. and Warmuth M.K., "The border augmented symmetric differences of halfspaces is learnable", *Unpublished manuscript*, 1989.

[105] Vallet F., "The Hebb Rule for Learning Linearly Separable Boolean Functions: Learning and Generalization.", *Europhys. Lett*, Vol. 8, p. 747, 1989.

[106] Vapnik V.N., Estimation of Dependences Based on Empirical Data (Springer Verlag, New-York, NY), 1982.

[107] Venkatesh S., "On Learning Binary Weights for Majority Functions", in *Proc. of the 4th Workshop on Computational Learning Theory*, Morgan Kaufman, p. 257, 1991.

[108] Widrow B., "Generalization and Information storage in Networks of Adaline Neurons", in *Self-organizing Systems*, Yovitz M. and Goldstein G. (editors), (Washington, DC: Spartan Books), 1962, p. 435.