Invariance Hints and the VC Dimension

Thesis by
William John Andrew Fyfe

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

California Institute of Technology
Pasadena, California

1992
(Submitted 26 May 1992)

©1992
William John Andrew Fyfe
All rights reserved

Acknowledgements

Many people have had a hand in this thesis, in many different ways.

The content was influenced enormously by my advisor, Dr. Yaser Abu-Mostafa, who, over the years, assisted me along those research paths that led to this thesis, as well as those that did not. The thesis was reviewed by Doctors Alan Barr, Carver Mead, Edward Posner and Richard Wilson, and I am grateful for their comments and suggestions. My research group, over the years, provided extremely useful feedback: Amir Atiya, Ruth Erlanson, Allen Knutson, Jack Lutz, David Schweizer.

Funding for my graduate studies came from a number of sources: the Natural Sciences and Engineering Research Council of Canada, the Air Force Office of Scientific Research, Hughes, and of course Caltech itself, which provided not only financial support, but a place to work.

Essential to the thesis was the well-being of its author, and essential to that were the many people around me. Some, John, David, Steve and Steve, have left to move on to other things; others, Pat, Barry and Rob, remain, and have struggled with me.

Finally, graduating with a thesis implies having started here some time ago. Getting here depended on many people who taught and inspired me along the way. Among the many, the names Ostlund, Ponzo, and Wilson stick out. And, of course, above all, my parents.

Abstract

We are interested in having a neural network learn an unknown function f. If the function satisfies an invariant of some sort, such as f is an odd function, then we want to be able to take advantage of this information and not have the network deduce the invariant based on an example of f.

The invariant might be defined in terms of an explicit transformation of the input space under which f is constant. In this case it is possible to build a network that necessarily satisfies the invariant.

In general, we define the invariant in terms of a partition of the input space such that if x, x' are in the same partition element then $f(x) = f(x')$. An example of the invariant would be a a pair (x, x') taken from a single partition element. We can combine examples of the invariant with examples of the function in the learning process. The goal is to substitute examples of the invariant for examples of the function; the extent to which we can actually do this depends on the appropriate VC dimensions. Simulations verify, at least in simple cases, that examples of the invariant do aid the learning process.

Contents

Acknowledgements iii
Abstract v
Chapter 1. Introduction 1
1. Outline 2
2. Notation 3
Chapter 2. Invariants 5
1. Perceptrons 5
2. Feed-Forward Networks 7
3. Tangent Prop 8
Chapter 3. The VC Dimension 9
1. Boolean-Valued Functions 11
2. Real-Valued Functions 14
Chapter 4. Invariance Hints 21
1. Boolean-Valued Functions 22
2. Real-Valued Functions 24
3. Graph View 25
4. Examples 31
Chapter 5. Learning 37
1. Perceptrons 37
2. Feed-Forward Networks 40
3. Group Invariance and Feed-Forward Networks 47

Chapter 6. Simulations **55**

Chapter 7. Conclusions **67**

1. Further Study **67**

Bibliography **69**

CHAPTER 1

Introduction

Consider a simple problem, in statement if not in solution: Is the number p prime? We begin with a naïve approach. We appeal to a dictionary, such as the Oxford English Dictionary, which defines a prime number as "Having no integral factors except itself and unity," and use the definition to establish a test of primality. Our solution, then, is to take each of the integers between 1 and p, and check to see if it divides p. If none divides p, then p is prime.

Our solution is correct, but it is not particularly efficient. A bit of simple mathematics can speed up our solution. We observe that factors must come in pairs, and both such factors can not be greater than $\sqrt{p}$. Thus if p is not prime, it has a factor no larger than $\sqrt{p}$, and we can thus restrict our search to those integers between 2 and $\sqrt{p}$, inclusive.

Heuristics can further speed up our solution. For example, even numbers, except for 2, are not prime, and odd numbers can not have even divisors. Thus we need not test p for divisibility by any even number other than 2.

In special cases, we can do much better. In the case of Mersenne primes, numbers of the form $2^q - 1$, where q is prime, there is the Lucas-Lehmer test. Colquitt and Welsh combined this test with other heuristics to find the 29th Mersenne prime, $2^{110503} - 1$ and to verify that $2^{132049} - 1$ is the 30th, in order of increasing size [4]. These primes have 33,265 and 39,751 digits respectively; it would be impossible to verify their primality by checking divisibility, using our solution above.

Consider another simple problem, again in statement if not in solution: Does this picture contain a tree? Appealing to a dictionary does not give us the solution in this case. The Oxford English Dictionary defines tree as, among other things, "A perennial plant having a self-supporting woody main stem or *trunk* (which usually develops woody branches at some distance from the ground), and growing to a considerable height and size." It is, in some sense, accurate without being helpful.

There is a certain irony in these two problems. If asked if a random number

was prime, one would probably apply some mental heuristics, such as divisibility by 2, 3 and 5, and, failing those, simply guess. On the other hand, if asked if a picture contained a tree, one would very likely answer directly. Yet the former problem has a straightforward solution, while the latter does not appear to.

Problems, such as recognizing trees, often fall in the domain of neural networks. Such a network is an abstraction of a brain. Networks are programmed by learning from example, not unlike their biological counterparts. It is hoped that by using example pictures that do and do not contain trees, the network can eventually deduce the rules that define a tree. This becomes the naïve solution for our tree recognition problem.

As was the case with the prime number problem, there is additional information that we can use besides example pictures, to aid in recognizing trees. For example, suppose we are given a pair of different pictures, each of a Joshua tree. We may be uncertain whether a Joshua tree is actually a tree or not, but we will know that they either both are, or both are not. There are properties that can vary without changing the final result. In the case of recognizing trees, we can assume, for example, that variations in position, orientation, colour, and so forth, do not change the answer.

In the case of prime numbers, we improved our solution by eliminating unnecessary tests of divisibility. In the case of recognizing trees, we want to eliminate the need for the network to deduce invariants from the examples, such as position independence, when we know in advance what these invariants are. This is the aim of this thesis.

1. Outline

We begin by reviewing, in chapter 2, how invariants have been used. In the case of perceptrons, invariants are incorporated directly into the structure of the network, and thus need not be learned. In the case of feed-forward networks, it is possible to learn the invariant, from examples of the invariant, while simultaneously learning the solution to the given problem. Such examples of the invariant can be viewed as a "hint."

Since learning is done from examples, we need to know that the examples are sufficient in quantity to be able to extrapolate based on them. It is not enough to remember the examples; we want to learn the rules that define the examples. This ability to generalize from the examples depends on a quantity known as the VC dimension; it is the subject of chapter 3.

Next we turn to hints, or examples of an invariant. We wish to use these hints to learn the rule that defines the invariant. Again we want the network to generalize, to learn the rule that defines the invariant. In chapter 4, we apply the results about the VC dimension to our hints in order to establish the conditions under which generalization occurs.

With the theoretical foundations established, we turn, in chapter 5, to learning algorithms. For perceptrons, where the invariant is incorporated into the network

directly, we need to develop an algorithm for learning that maintains the network structure. For feed-forward networks, we need to introduce examples of the invariant into the learning process, taking care to satisfy the conditions establish by the appropriate VC dimensions. We also investigate transferring the structures used with perceptrons over to feed-forward networks, and the implications for the free-forward learning algorithm.

Finally, in chapter 6, we look at simulation results for learning a simple function, using hints, on a feed-forward network. This presents us with an opportunity to explore some of the questions raised in earlier chapters.

2. Notation

Our problem, in its most general form, is to find a function g_α, from a class of functions G, that approximates a particular function f. The index α ranges over some index set A.

The class of functions we will typically use are those computed by a class of neural networks. Within such a class, the function g_α computed by a given network will be defined by the assignments to the parameters for each neuron, namely the weights $\mathbf{w}$ and the threshold τ. In this case, we might choose for our index variable α a vector of all these parameters. We will identify a network by the function g_α that it computes.

The inputs come from the set X. This set may be n-bit Boolean strings, a subset of n-dimensional Euclidean space, or just a set of n points. We will want to partition this set X into disjoint subsets; we will use β to index these subsets.

CHAPTER 2

Invariants

In this chapter we will show examples of how invariants are used. In particular, we will look at the perceptron model, where group invariance is asserted by the structure of the network. In the feed-forward model, the invariance is asserted during the learning process.

1. Perceptrons

Perceptrons are defined by Minsky and Papert [12]; they are simple, two layer networks. The first layer is made up of functions φ, taken from the set Φ. The second layer is a single, output neuron that computes a linear threshold function of the outputs of the first layer

$$g_\alpha(x) = \left[\sum_{\varphi\in\Phi} w_\varphi \varphi(x) > 0\right].$$

The notation $[\cdots]$ will be used to represent 1 if the enclosed condition is true, and 0 otherwise. To avoid subscripts, we will write w_φ as $w(\varphi)$. Also, we will abbreviate the sum by an inner product

$$\sum_{\varphi\in\Phi} w(\varphi)\varphi(x) = \mathbf{w}\cdot\Phi(x).$$

Suppose we have a function, such as the parity of an n-bit binary number. We know that this function is independent of the ordering of the bits in the string. Parity, then, is invariant under reordering of the input string. Alternatively, suppose we are doing pattern recognition, and the function is independent of the relative position of the image. In this case the function is invariant under translation of the input image. These invariants are formalized by a set of transformations $T = \{t\colon X \to X\}$ that forms a group under the operator $\circ$, composition of functions.

For the parity example, the group of transformations would be all those functions that return the n bits of the input vector $\mathbf{x}$ in a different order. This group

is the group of all permutations of n objects. For the pattern recognition example, we would use the group of translations in the plane, or, in order to keep everything finite, translations on the torus.

Two inputs x and $\hat{x}$ are equivalent if there is some $t \in T$ such that $\hat{x} = t(x)$. Similarly, two input functions φ and $\hat{\varphi}$ are equivalent if there is some t such that $\hat{\varphi}(x) = \varphi \circ t(x)$. The group divides the input set X and input functions Φ into equivalence classes. Finally, a function f is invariant under such a group of transformations T, or T-invariant, if, $\forall t \in T$, $f \circ t(x) = f(x)$.

In [12], Minsky and Papert prove the group invariance theorem. Suppose we have a perceptron, and the function g_α it computes is invariant under the group T. Suppose also that the group is finite, and that the set of functions Φ is closed under T. In the original perceptron g_α, we assumed the weights $w(\varphi)$ depended only on the corresponding input function φ; however, we can generate an equivalent perceptron where the weights $w(\varphi)$ depend only on the equivalence class containing φ, that is if φ and $\hat{\varphi}$ are equivalent, $w(\varphi) = w(\hat{\varphi})$.

The proof of this theorem is straightforward. Since the set Φ is closed under T, we have $\{ \varphi \circ t \mid \varphi \in \Phi \} = \Phi$. This means that

$$\mathbf{w} \cdot \Phi(x) = \sum_{\varphi \in \Phi} w(\varphi)\varphi(x) = \sum_{\varphi \in \Phi} w(\varphi \circ t)\varphi \circ t(x).$$

Since the function g_α is invariant under T, we have $g_\alpha(x) = g_\alpha(t^{-1}(x))$. Suppose $g_\alpha(x) = 1$, and thus $\mathbf{w} \cdot \Phi(x) > 0$. Then, $g_\alpha(t^{-1}(x)) = 1$ and

$$\sum_{\varphi \in \Phi} w(\varphi \circ t)\varphi \circ t(t^{-1}(x)) = \sum_{\varphi \in \Phi} w(\varphi \circ t)\varphi(x) > 0.$$

This holds for all $t \in T$, so we have

$$\frac{1}{|T|} \sum_{t \in T} \sum_{\varphi \in \Phi} w(\varphi \circ t)\varphi(x) = \sum_{\varphi \in \Phi} \left(\frac{1}{|T|} \sum_{t \in T} w(\varphi \circ t) \right) \varphi(x) > 0.$$

Let $v(\varphi) = |T|^{-1} \sum_{t \in T} w(\varphi \circ t)$, and $\hat{g_\alpha}(x) = [\mathbf{v} \cdot \Phi(x) > 0]$. We have $\mathbf{v} \cdot \Phi(x) > 0$ and hence $\hat{g_\alpha}(x) = 1$. If instead we started with $g_\alpha(x) = 0$, we would obtain the corresponding result, namely $\mathbf{v} \cdot \Phi(x) \leq 0$ and $\hat{g_\alpha}(x) = 0$.

The new weight $v(\varphi)$ is simply the average value of the weights $w(\hat{\varphi})$ where $\hat{\varphi}$ is equivalent to φ. This weight $v(\varphi)$ is constant on each equivalence class of Φ.

We have shown that any network that computes a T-invariant function can be replaced by an equivalent network where the weights are constant on each equivalence class of Φ. The converse also holds, that is, any network whose weights are constant on each equivalence class is T-invariant. This follows directly, again using the closure of Φ over T, since

$$\sum_{\varphi \in \Phi} w(\varphi)\varphi(t(x)) = \sum_{\varphi \in \Phi} w(\varphi \circ t)\varphi \circ t(x) = \sum_{\varphi \in \Phi} w(\varphi)\varphi(x),$$

and thus $g_\alpha \circ t(x) = g_\alpha(x)$.

Those functions g_α that satisfy the invariant T can be realized by a network with the restriction that certain weights must be equal, and in any network where this restriction holds, the function computed must satisfy the invariant. This, then, suggests a technique for building networks that satisfy an invariant. And in this case, the invariant itself is enforced by the structure imposed on the network.

2. Feed-Forward Networks

A feed-forward network is a neural network that is divided into layers, with the inputs for a layer coming strictly from the layer before, and the outputs going to the layer after. The exceptions are the boundaries; the first, or input layer, gets its inputs from the environment, and the last, or output layer, returns its output to the environment.

Each neuron computes a simple function $\sigma(\mathbf{w} \cdot \mathbf{x} + \tau)$ where $\mathbf{x}$ is the vector of inputs to the neuron, $\mathbf{w}$ is the corresponding vector of weights, τ is the threshold, and σ is a smooth sigmoid function such as $\sigma(x) = (2/\pi)\arctan(x)$ or $\sigma(x) = (1 + e^{-x})^{-1}$.

Suppose we are given such a network. As before, let the function computed be g_α. Suppose also that we wish to have the network learn an unknown function f. That is, we wish to find a network such that its function g_α is close to f.

Suppose further that we are given examples (x, y) of the function f. We don't require that y be a function of x. If it is, then we assume that x is chosen according to a probability distribution $P(x)$, and $y = f(x)$. If it is not, then we assume that $y = f(x) + \gamma(x)$ where the error $\gamma(x)$ has a mean of zero. In this case examples are pairs (x, y) chosen according to a joint probability distribution $P(x, y)$, and the function we seek is the average value for y given x, that is $f(x) = E(y|x)$. For any pair (x, y), we want the output of the network, $g_\alpha(x)$, to be close to $f(x)$.

We can define an error function on n examples (x_i, y_i) by

$$E(\alpha) = \frac{1}{n} \sum_{i=1}^{n} \bigl(y_i - g_\alpha(x_i)\bigr)^2.$$

We can then minimize this error by gradient descent. For feed-forward networks this is the back propagation algorithm [11].

Suppose our function f satisfies an invariant. Earlier, we defined an invariant by a group of transformations T. This group induces a partition of the input set X into equivalence classes. We will, therefore, define an invariant in more general terms by a partition of the input set $X = \bigcup_\beta X_\beta$, where we have $f(x) = f(x')$ whenever $x, x' \in X_\beta$ for some β. If a network function g_α is close to f, then we expect, when $f(x) = f(x')$, to have $g_\alpha(x)$ close to $g_\alpha(x')$, or, equivalently, we expect $\bigl(g_\alpha(x) - g_\alpha(x')\bigr)^2$ to be small.

If we were given a series of n such pairs (x_i, x_i'), we could, as above, define an error function

$$E_I(\alpha) = \frac{1}{n}\sum_{i=1}^{n}\left(g_\alpha(x_i) - g_\alpha(x_i')\right)^2$$

and apply gradient descent to this. Note that we don't actually need to know the value of the function f at any of these points. This method of learning an invariant was suggested by Abu-Mostafa [1].

Minimizing the error functions $E(\alpha)$ and $E_I(\alpha)$ is only guaranteed to give us a network that performs well on the given examples. Given a sufficiently large number of examples, however, we expect the values given by such an error function to become representative, in probability, of the limiting value over all examples. What we would like to do is substitute examples of the invariant for examples of the function, and meet the sufficiency requirements with fewer examples of the function than had we not used the invariant at all.

Here we don't impose any structure on the network, but rather incorporate the invariant into the learning process, so that the network learns the invariant as it learns the function. And, since we don't need to know the value of the function for our examples of the invariant, we can generate arbitrary numbers of examples of the invariant, as required.

3. Tangent Prop

Simard and others [15] propose a technique for handling certain types of invariant, which they call tangent prop. Their technique generalizes back propagation to learn both the unknown function and its derivative.

Suppose we define an invariant by a function $t(a, x)$ that transforms an input x according to a parameter a. For horizontal translation, for example, the parameter would be the amount of the translation. For rotation, it would be the angle. We assume that this transformation is differentiable with respect to a and x, and that $t(0, x) = x$.

Since the function f satisfies an invariant, it is constant as we transform an input x according to t. Thus, in the direction of this transformation $t(a, x)$, we have $f'(x) = 0$. Any network g_α approximating f should also have $g_\alpha'(x) \approx 0$, again in the direction of the transformation t. Tangent prop, then, modifies back propagation by also minimizing the derivative of the network, in the direction of the transformation.

This technique is in some ways similar to our earlier modification to back propagation. Again we are incorporating the learning of the invariant directly into the learning process. To do this we need not know the actual desired output for a given input x; we only consider a directional derivative at x. We will not consider this technique further.

CHAPTER 3

The VC Dimension

Our task is to select a network g_α that approximates an unknown function f, based on a series of examples, drawn at random.

We have seen, with feed-forward networks and the back propagation algorithm, that we can define an error function and minimize it using gradient descent. In particular, for a series of examples $\big((x_1, y_1), \ldots, (x_n, y_n)\big)$, we define the error function

$$E(\alpha) = \frac{1}{n} \sum_{i=1}^{n} \big(y_i - g_\alpha(x_i)\big)^2.$$

Minimizing this error function gives us a network that performs well on the n examples given. What we want, though, is a network which computes a function close to f. What we need, then, are the circumstances under which we can be fairly certain that the performance of the network on those examples is representative of the network's performance overall. We want the observed performance to *generalize.*

Corresponding to $E(\alpha)$, we have the functional

$$I(\alpha) = \iint_{X \times Y} \big(y - g_\alpha(x)\big)^2 P(x, y)\, dx\, dy.$$

The error function $E(\alpha)$ is an estimate of $I(\alpha)$. We need to know the circumstances under which we can be confident that the estimate is an accurate one, and know that minimizing $E(\alpha)$, and hence $I(\alpha)$, gives us a network where g_α is close to f.

First, let's verify that minimizing $I(\alpha)$ is equivalent to finding the function g_α closest to f. Should y be a function of x, then we can substitute $f(x)$ for y to obtain

$$I(\alpha) = \int_X \big(f(x) - g_\alpha(x)\big)^2 P(x)\, dx,$$

and hence we are minimizing the distance between f and g_α, as desired. Alternatively, suppose y contains some error. In this case the function f we seek is

defined by

$$f(x) = E(y|x) = \int_Y yP(y|x)\,dy,$$

where $P(y|x) = P(x,y)/P(x)$. Thus we get

$$\begin{aligned}
I(\alpha) &= \iint_{X\times Y} \big(y - f(x) - g_\alpha(x) + f(x)\big)^2 P(x,y)\,dx\,dy \\
&= \iint_{X\times Y} \big(y - f(x)\big)^2 P(x,y)\,dx\,dy + \iint_{X\times Y} \big(f(x) - g_\alpha(x)\big)^2 P(x,y)\,dx\,dy \\
&\qquad - 2\iint_{X\times Y} \big(f(x) - g_\alpha(x)\big)\big(y - f(x)\big) P(x,y)\,dx\,dy \\
&= \iint_{X\times Y} \big(y - f(x)\big)^2 P(x,y)\,dx\,dy + \iint_{X} \big(f(x) - g_\alpha(x)\big)^2 P(x)\,dx \\
&\qquad - 2\iint_{X} \big(f(x) - g_\alpha(x)\big) \left[\int_Y \big(y - f(x)\big) P(y|x)\,dy\right] P(x)\,dx.
\end{aligned}$$

The last term is 0, since the inner integral of $y - f(x)$ is 0, by our definition of $f(x)$. The second term is again the distance between f and g_α. The first term is independent of α, and hence minimizing $I(\alpha)$ is the same as minimizing just the second term, that is, minimizing the distance between f and g_α.

Next, let's verify that minimizing $E(\alpha)$ gives us a function g_{α_E} that isn't too far from optimal, as defined by $I(\alpha)$. Suppose we know that $\big|I(\alpha) - E(\alpha)\big| < \epsilon$, in probability. The optimal function g_α, the one closest to f, is the one that minimizes $I(\alpha)$. Let this function be g_{α_I}.

We have

$$\begin{aligned}
I(\alpha_I) &\le I(\alpha_E), \\
E(\alpha_E) &\le E(\alpha_I),
\end{aligned}$$

by the choice of g_{α_I} and g_{α_E}. We also have

$$\begin{aligned}
I(\alpha_E) &\le E(\alpha_E) + \epsilon \\
E(\alpha_I) &\le I(\alpha_I) + \epsilon
\end{aligned}$$

since we assume $E(\alpha)$ is close to $I(\alpha)$. Combining these we have

$$I(\alpha_I) \le I(\alpha_E) \le E(\alpha_E) + \epsilon \le E(\alpha_I) + \epsilon \le I(\alpha_I) + 2\epsilon,$$

that is,

$$I(\alpha_I) \le I(\alpha_E) \le I(\alpha_I) + 2\epsilon.$$

Thus we see that the function g_{α_E} that minimizes our estimate $E(\alpha)$ is at most 2ϵ farther from f than the actual, optimal, function g_{α_I} is from f, in probability.

We assumed that $\big|I(\alpha) - E(\alpha)\big| < \epsilon$, that we have generalization. Even if we fail to pick the network that minimizes $E(\alpha)$, we still want generalization. We need to be confident that the performance of our network on the examples, good

or bad, is typical of its performance overall, since we are interested in how f compares to our chosen function g_α over all inputs X, not just over the examples we have seen.

If we had a single network, we would get the required number of examples for generalization from Bernoulli's theorem, for it gives a bound, in terms of the number of examples, on the probability that the observed behaviour differs from the overall behaviour by more than some constant δ.

Our task, though, is to choose a single network from a class of networks, a class that may not be finite. Further, we must assume that we may ultimately choose any one of these networks. We need, therefore, a uniform result, that gives us a bound, given a number of examples, on the probability that *any* network has a difference between observed and overall behaviour of more than δ. This result is due to Vapnik and Chervonenkis [16, 17, 18].

1. Boolean-Valued Functions

First let us consider a probability model. We have a collection of events $\{E_\alpha\}$, with each event E_α a subset of some set X. Also, we have a series of samples $x_1, x_2, \ldots, x_n$, chosen independently, according to the distribution $P(x)$ over X.

For each event E_α, we have its probability π_α, and, given the n samples, an observed frequency $\nu_\alpha^{(n)}$. What we want are the conditions under which $\nu_\alpha^{(n)}$ tends to π_α uniformly, that is, for any $\delta > 0$, we want

$$\lim_{n\to\infty} \operatorname{Prob}\left\{\sup_\alpha \left|\pi_\alpha - \nu_\alpha^{(n)}\right| > \delta\right\} = 0.$$

Theorem 3.1 (Vapnik and Chervonenkis [16]). For a series of n samples, $n > 2/\delta^2$,

$$\operatorname{Prob}\left\{\sup_\alpha \left|\pi_\alpha - \nu_\alpha^{(n)}\right| > \delta\right\} \le 4m(2n)e^{-\delta^2 n/8}.$$

The growth function $m(n)$ is defined in terms of the events $\{E_\alpha\}$. Consider a sequence of n samples, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. If two events E_α, $E_{\alpha'}$ should agree on these samples, that is, if $S = \{x_1, \ldots, x_n\}$ and $E_\alpha \cap S = E_{\alpha'} \cap S$, then the two events are indistinguishable by the samples. We will refer to $E_\alpha \cap S$ as the projection of E_α onto the samples. The set of all projections $\{\, E_\alpha \cap S \mid \text{all events } E_\alpha \,\}$ is a subset of the power set of S, $\mathscr{P}(S)$. Let $m(\mathbf{x})$ equal the total number of distinct projections, which is independent of the ordering of the samples. Note that $m(\mathbf{x}) \le 2^n$, since this is the size of $\mathscr{P}(S)$. The function $m(n)$ is defined as the maximum number of projections for any choice of $\mathbf{x}$, that is

$$m(n) = \max_{|\mathbf{x}|=n} m(\mathbf{x}).$$

Suppose for some n that $m(n) = 2^n$. Then there are n samples $x_1, \ldots, x_n$ that give rise to all 2^n possible projections. If we take any $k < n$ of these samples, say $x_{i_1}, \ldots, x_{i_k}$, then these samples must give rise to all 2^k possible projections, and

hence $m(k) = 2^k$ for all $k < n$. If at some point $n = d + 1$ we have $m(n) < 2^n$, then it must be that $m(n) < 2^n$ for all $n > d + 1$. It turns out that if there is such a point d, then $m(n) < n^{d+1} + 1$. The VC dimension is defined as d, the last value for which the growth function was 2^n. If there is no such d, then we consider the VC dimension to be infinite.

In the case of a finite VC dimension, we can derive stronger bounds on the growth function $m(n)$.

THEOREM 3.2 (SAUER [14]). Suppose the VC dimension of a set of events $\{E_\alpha\}$ is d. For $n > d$,

$$m(n) \leq \sum_{i=0}^{d} \binom{n}{i}.$$

COROLLARY 3.3. For $n > d$ and $n \geq 4$, $m(n) \leq 1.5n^d/d!$.

Sauer's result also follows from the following theorem.

THEOREM 3.4 ([6]). Suppose $G = \{g_\alpha\}$ is a finite set of Boolean functions on n inputs. Then

(1) $\exists G$ with $|G| = \sum_{i=0}^{d} \binom{n}{i}$ and the VC dimension of G is d,
(2) if $|G| > \sum_{i=0}^{d} \binom{n}{i}$ then the VC dimension of G is strictly greater than d.

Recall the statement of theorem 3.1. It gives a bound on the probability that $E(\alpha)$ differs from $I(\alpha)$ by more than δ. Suppose we want to consider a particular probability of this happening, say ϵ, and solve for δ. We will assume a finite VC dimension, and use the result of corollary 3.3. What we find is δ depends on the ratio of the number of samples to the VC dimension. We reduce δ by increasing this ratio. We can do this either by increasing n, that is, using more samples, or by reducing d, that is, using fewer events.

The condition that $m(n)$ be bounded by a polynomial is sufficient to ensure uniform convergence. It is not necessary, however. It is always possible that those samples $\mathbf{x}$ that give rise to large values of $m(\mathbf{x})$ are extremely unlikely, and the typical value for $m(\mathbf{x})$ is smaller.

We defined $m(n)$ as the maximum number of projections for any choice of n samples. We can also define an average number of projections $N(n)$. The n samples $\mathbf{x} = (x_1, \ldots, x_n)$ are chosen independently, so the probability of $\mathbf{x}$ is $P(\mathbf{x}) = \prod_{i=1}^{n} P(x_i)$. Thus we have

$$N(n) = E\big(m(\mathbf{x})\big) = \int_{X^n} m(\mathbf{x}) P(\mathbf{x})\, d\mathbf{x}.$$

Finally, let $H(n) = E\big(\log_2 m(\mathbf{x})\big)$.

We can use $H(n)$ to strengthen theorem 3.5.

THEOREM 3.5 (VAPNIK AND CHERVONENKIS [16]). For

$$\lim_{n\to\infty} \text{Prob}\left\{\sup_{\alpha}\left|\pi_\alpha - \nu_\alpha^{(n)}\right| > \delta\right\} = 0,$$

it is necessary and sufficient that

$$\lim_{n\to\infty} \frac{H(n)}{n} = 0.$$

The difficultly in using this definition is the fact that the quantity $H(n)$ depends on the probability distribution, which we assume we do not know. By using the worst case $m(n)$, we make the result independent of the distribution.

Now let us consider neural networks and Boolean functions. We will assume that the unknown function f, and the network functions g_α are all Boolean. We used

$$E(\alpha) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - g_\alpha(x_i)\right)^2$$

to measure the error on the given examples. Since y_i and $g_\alpha(x_i)$ are both Boolean, the square of their difference is 0 if the two are equal, and 1 if not. Thus $E(\alpha)$ is the number of examples where the network erred divided, by the total number of examples. In other words, $E(\alpha)$ is the frequency of error. Similarly,

$$I(\alpha) = \int_X \left(f(x) - g_\alpha(x)\right)^2 P(x)\,dx$$

is the probability of error. If we let

$$E_\alpha = \{\, x \in X \mid f(x) \neq g_\alpha(x) \,\},$$

then we have $\pi_\alpha = I(\alpha)$ and $\nu_\alpha^{(n)} = E(\alpha)$. Uniform convergence of $\nu_\alpha^{(n)}$ to π_α thus gives us uniform convergence of $E(\alpha)$ to $I(\alpha)$, and generalization, as required. Note that in translating our neural network model to a collection of events, we have not changed the meaning of x_i or X. We still are choosing x_i according to the probability $P(x)$. We don't carry y_i over to the events directly; however, y_i is a function of x_i so nothing is lost.

There is another way to look at the projections above. Again we will assume we have a fixed, but arbitrary, sequence of n samples, in this case $(x_1, y_1), \ldots, (x_n, y_n)$, where $y_i = f(x_i)$. For each event E_α, we will define an n-bit binary string $\mathbf{b}_\alpha$ where bit i is 1 if $x_i \in E_\alpha$, or, equivalently, recalling our correspondence with neural networks, if $y_i \neq g_\alpha(x_i)$. If we let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, $\mathbf{y} = (y_1, y_2, \ldots, y_n)$, and $g_\alpha(\mathbf{x}) = (g_\alpha(x_1), \ldots, g_\alpha(x_n))$, then we have $\mathbf{b}_\alpha = \mathbf{y} \oplus g_\alpha(\mathbf{x})$ where $\oplus$ is the exclusive or operator. The projections are mapped one-to-one onto the n-bit vectors; the vectors $\mathbf{b}_\alpha$ are characteristic strings for the projections $E_\alpha \cap S$.

Recall that our sequence of samples, though arbitrary, is fixed, so the vector $\mathbf{y} = f(\mathbf{x})$ is constant. Also, since $\mathbf{b}_\alpha = \mathbf{y} \oplus g_\alpha(\mathbf{x})$, we have $g_\alpha(\mathbf{x}) = \mathbf{y} \oplus \mathbf{b}_\alpha$. Thus we have $\mathbf{b}_{\alpha_1} = \mathbf{b}_{\alpha_2}$ if and only if $g_{\alpha_1}(\mathbf{x}) = g_{\alpha_2}(\mathbf{x})$, and the number of distinct

vectors $\mathbf{b}_\alpha$ is equal to the number of distinct vectors $g_\alpha(\mathbf{x})$. Thus we need not consider $\mathbf{y}$ at all in computing the VC dimension of the set $\{g_\alpha\}$.

We can also view $\mathbf{b}_\alpha$ as an n-dimensional vector. Coordinate i of $\mathbf{b}_\alpha$ is equal to $\big(y_i - g_\alpha(x_i)\big)^2$. The vector $\mathbf{b}_\alpha$ is thus a vector of values for $\big(y - g_\alpha(x)\big)^2$ evaluated at each of the n examples (x_i, y_i). Each of these vectors represents the coordinates of one of the vertices of an n-dimensional unit hypercube, and the growth function $m(n)$ is thus the maximum number of vertices that can be covered.

Let's now construct an example. Suppose our input set X has n elements, and the functions g_α are all those Boolean functions that assign to at most d elements the value 1, and 0 to the rest. This set of functions has VC dimension d.

To see that the VC dimension is at least d, we need to show that there is some sequence of d distinct samples $\mathbf{x}$ such that we get all 2^d distinct vectors $g_\alpha(\mathbf{x})$. Suppose the d samples are $\mathbf{x} = (x_1, x_2, \ldots, x_d)$, and suppose we have some d-bit vector $\mathbf{b}$. We need to find a function g_α so that $g_\alpha(\mathbf{x}) = \mathbf{b}$. Suppose that for $i = 1, \ldots, d$ we have $g(x_i)$ equal to bit i of $\mathbf{b}$, and for $i = d+1, \ldots, n$ we have $g(x_i) = 0$. Then this function g can assign at most d inputs the value 1, and hence $g \in \{g_\alpha\}$.

To see that the VC dimension is at most d, we need to show that any sequence of $d+1$ distinct samples $\mathbf{x}$ can not give all 2^{d+1} vectors $\mathbf{b}_\alpha(\mathbf{x})$. In particular, we can not get the vector $\mathbf{b} = 1 \ldots 1$ since this would require some function g_α that assigns to those $d+1$ samples the value 1.

This set of functions has $\sum_{i=0}^{d} \binom{n}{i}$ elements. Thus this set can be used to prove part 1 of theorem 3.4. Furthermore, this set gives an instance where the inequality $m(n) \le \sum_{i=0}^{d} \binom{n}{i}$ from theorem 3.2 can be an equality.

2. Real-Valued Functions

Vapnik and Chervonenkis generalized their result for probabilities and frequencies of events, to the following [17]. Suppose you have a family of uniformly bounded functions $0 \le F_\alpha(z) \le c$. We want to know the conditions under which the empirical average of these functions tends to the expected value uniformly, that is when

$$\lim_{n\to\infty} \operatorname{Prob}\left\{ \sup_\alpha \left| E\big(F_\alpha(z)\big) - \frac{1}{n}\sum_{i=1}^{n} F_\alpha(z_i) \right| > \delta \right\} = 0.$$

Note that the variable z may be a vector, and, if so, the expected value of $F_\alpha(z)$ is computed over the appropriate joint distribution.

We can use this result directly: in our case the functions $F_\alpha(z)$ are simply $\big(y - g_\alpha(x)\big)^2$, where $z = (x, y)$. It is not necessary that y be a function of x; the choice of a pair (x, y) depends only on the joint distribution $P(x, y)$.

Uniform convergence once again depends on a growth function. As we did in the Boolean case, we will take each function $\big(y - g_\alpha(x)\big)^2$ and evaluate it for the

n pairs (x_i, y_i). Again let us call these vectors $\mathbf{b}_\alpha$, though now the individual components are real numbers, not just 0 or 1.

The range of values for $\big(y - g_\alpha(x)\big)^2$ depends on the transfer function σ we use in our network; we will assume that values for y are within the same range. If we use $\sigma(x) = (1 + e^{-x})^{-1}$ then we have a range of $[0, 1]$ and the vectors $\mathbf{b}_\alpha$ are points within the n dimensional hypercube $[0, 1]^n$. If we use instead $\sigma(x) = (2/\pi)\arctan(x)$ then we have a range of $[-1, 1]$ and the hypercube becomes $[0, 4]^n$. The uniform bound c on the functions $F_\alpha(z)$ is the edge length of the hypercube.

Let B be the set of all the vectors $\mathbf{b}_\alpha$. In the Boolean case the set B was a subset of the vertices of the hypercube. Now it is a possibly uncountable collection of points within the hypercube. Since we cannot count these points we need to use another method to quantify the size of B.

We will use one of two roughly equivalent techniques; both, as well as a third, are investigated in [9]. First, an ϵ-net for B is a finite set of points U in $\mathbf{R}^n$ such that for every $\mathbf{b} \in B$, there is some $\mathbf{u} \in U$ such that $\rho(\mathbf{b}, \mathbf{u}) \leq \epsilon$. We may additionally require that the points $\mathbf{u}$ are elements of B; in this case we have a proper ϵ-net.

Alternatively, an ϵ-separation is a set of points $U \subset B$ such that for any $\mathbf{u}_1, \mathbf{u}_2 \in U$, we have $\rho(\mathbf{u}_1, \mathbf{u}_2) > \epsilon$.

What we seek, for a given set B, is the smallest ϵ-net, or the largest ϵ-separation. Let $\mathscr{N}(\epsilon, B)$ be the size of the smallest ϵ-net, $\mathscr{N}_0(\epsilon, B)$ be the size of the smallest proper ϵ-net, and $\mathscr{M}(\epsilon, B)$ be the size of the largest ϵ-separation. We have

$$\mathscr{M}(2\epsilon, B) \leq \mathscr{N}(\epsilon, B) \leq \mathscr{N}_0(\epsilon, B) \leq \mathscr{M}(\epsilon, B).$$

We will verify this chain of inequalities from right to left. The rightmost inequality holds since a maximal ϵ-separation must also be a proper ϵ-net. The next holds since relaxing the condition on the choice of the points in the ϵ-net can only help. Finally, the last inequality holds because of the pigeon hold principle. To see this, take any maximal 2ϵ-separation, and any minimal ϵ-net. There can be at most 1 point from the 2ϵ-separation within ϵ of any point in the ϵ-net, since if there were 2, then these two points would be as most 2ϵ apart. It is this chain of inequalities that allows us to choose between an ϵ-net, a proper ϵ-net, and an ϵ-separation based on which is most convenient; we will use an ϵ-net.

For our ϵ-nets, we will follow Vapnik and Chervonenkis and use the L^∞ distance $\rho(\mathbf{b}, \mathbf{u}) = \max_i |b_i - u_i|$. Others, such as Haussler [7], use the L^1 distance $\rho(\mathbf{b}, \mathbf{u}) = \frac{1}{n}\sum_{i=1}^n |b_i - u_i|$. In [17], Vapnik and Chervonenkis investigate the implications of using the L^1 norm, relative to the L^∞ norm.

The hypercube has a finite edge length c, and hence a finite ϵ-net must exist. Such an ϵ-net need have no more than $\big(1 + \lfloor c/(2\epsilon)\rfloor\big)^n$ elements. Also recall that the elements of the net do not need to come from our set B — they can be arbitrary points within the hypercube.

For any choice of n pairs $(\mathbf{x}, \mathbf{y}) = \big((x_1, y_1), \ldots, (x_n, y_n)\big)$, let $m\big(\epsilon, (\mathbf{x}, \mathbf{y})\big)$ be the size of the smallest ϵ-net that covers the corresponding set $\{\mathbf{b}_\alpha\}$, that is,

$m(\epsilon, (\mathbf{x}, \mathbf{y})) = \mathscr{N}(\epsilon, \{\mathbf{b}_\alpha\})$. For real-valued functions we have the size of the smallest ϵ-net playing the same role as the number of distinct projections did in the Boolean case. Note that, as in the Boolean case, the order of the samples is independent of the size of the ϵ-net, since reordering the samples corresponds simply to relabelling the coordinate axes. We let $m(\epsilon, n)$ be the maximum value for $m(\epsilon, (\mathbf{x}, \mathbf{y}))$, and $N(\epsilon, n)$ be the expected value, over all possible choices of n pairs. We also define $H(\epsilon, n) = E(\log_2 m(\epsilon, (\mathbf{x}, \mathbf{y}))$, as before. Finally, we define

$$c(\epsilon) = \lim_{n \to \infty} \frac{H(\epsilon, n)}{n},$$

and this limit does exist. Since $N(\epsilon, n)$ is bounded by $\left(1 + \lfloor c/(2\epsilon) \rfloor\right)^n$, we have

$$0 \leq c(\epsilon) \leq \log_2 \left(1 + \left\lfloor \frac{c}{2\epsilon} \right\rfloor\right).$$

Further, $c(\epsilon)$ is non-decreasing as we decrease ϵ, since a smaller ϵ can not provide for a smaller ϵ-net.

Theorem 3.6 (Vapnik and Chervonenkis [17]). For

$$\lim_{n \to \infty} \text{Prob} \left\{ \sup_\alpha \left| E\big(F_\alpha(z)\big) - \frac{1}{n} \sum_{i=1}^{n} F_\alpha(z_i) \right| > \delta \right\} = 0,$$

it is necessary and sufficient that $c(\epsilon) = 0$ for all $\epsilon > 0$.

Let us return briefly to the Boolean case. Here the vectors $\mathbf{b}_\alpha$ are the vertices of the unit hypercube. For $\epsilon < \frac{1}{2}$, any single point in the ϵ-net can cover at most one vertex. Thus to cover $\{\mathbf{b}_\alpha\}$ we need a point in the ϵ-net for every vertex represented by $\mathbf{b}_\alpha$, and hence we must have $m(\epsilon, n) = m(n)$, provided $\epsilon < \frac{1}{2}$. The same applies to N and H, and we see that the real-valued case does generalize the Boolean-valued one.

Theorem 3.6 gives the necessary and sufficient conditions for uniform convergence in the real-valued case, just as theorem 3.5 does for the Boolean case. However, we do not yet have an analogue for the VC dimension.

Vapnik [18] defines a VC dimension of a class of real-valued functions to be equal to that of a corresponding class of Boolean functions. These Boolean functions are those created by taking each real-valued function g_α, and applying an arbitrary threshold to it. Specifically, we define the functions $\hat{g}_{\alpha\gamma}(x, y) = \left[\big(y - g_\alpha(x)\big)^2 > \gamma\right]$ where $\gamma \in \mathbf{R}$. For this extended class of functions $\{\hat{g}_{\alpha\gamma}\}$, we apply our results for Boolean functions. Given n samples (x_i, y_i), we have, for each function $\hat{g}_{\alpha\gamma}$, the n-bit vector $\hat{\mathbf{b}}_{\alpha\gamma}$. Let the number of distinct vectors $\hat{\mathbf{b}}_{\alpha\gamma}$ be $\hat{m}(\mathbf{x}, \mathbf{y})$, and, as before let $\hat{m}(n) = \max_{(\mathbf{x}, \mathbf{y})} \hat{m}(\mathbf{x}, \mathbf{y})$. Finally, associated with the growth function $\hat{m}(n)$, we have a VC dimension $\hat{d}$, which is then also the VC dimension for the set of functions g_α.

THEOREM 3.7 (VAPNIK [18]). For n samples, where $n > 2/\delta$,

$$\text{Prob}\left\{\sup_{\alpha}\left|E\left(\left(y-g_\alpha(x)\right)^2\right)-\frac{1}{n}\sum_{i=1}^{n}\left(y_i-g_\alpha(x_i)\right)^2\right|>c\delta\right\}<6\hat{m}(2n)e^{-\delta^2 n/4}.$$

There are other ways to define a VC dimension. In general, for real-valued functions, the size of a minimal ϵ-net replaces the growth function. If this quantity fails to dominate the negative exponential $e^{-\gamma n}$, where γ is a constant that depends on the details of a theorem such as 3.7, then, for n sufficiently large, we get uniform convergence. If we can bound the size of an ϵ-net by a polynomial in n and ϵ^{-1} of degree at most d, then we will have the required uniform convergence, and we may use d as an estimate of the VC dimension.

Suppose we have a 3-dimensional volume, and cover it with an ϵ-net. If we decrease ϵ by a factor of 2, then each point in the net covers 2^{-3} less volume, and thus we expect to need about 2^3 times as many points to continue to cover the volume. The size of such an ϵ-net will be about $(a/\epsilon)^3$ points, for some constant a. The exponent 3 corresponds to our volume being 3-dimension. We can define a metric dimension [9] as

$$\dim(B)=\lim_{\epsilon\to 0}\frac{\log\mathcal{N}(\epsilon,B)}{\log(\epsilon^{-1})},$$

where this limit exists. For our volume, the metric dimension will be 3. This quantity need not be integral; if it is fractional then the set B is fractal [10, 5].

Suppose that for any n, and n samples (x_i, y_i), the set $B = \{\mathbf{b}_\alpha\}$ has its metric dimension bounded by d. Then, for any δ we have ϵ_0 such that for $\epsilon < \epsilon_0$, we have

$$\frac{\log\mathcal{N}(\epsilon,B)}{\log(\epsilon^{-1})}<d+\delta,$$

and thus $\mathcal{N}(\epsilon, B) < \epsilon^{-(d+\delta)}$. In such a case, we have uniform convergence, and d becomes our VC dimension.

There are additional ways to attempt to bound the size of the minimal ϵ-net. Haussler [7], following Pollard [13], uses the combinatorial dimension. This approach is similar to that of Vapnik. Here we are given our set of points B, and translate each by some constant vector $\mathbf{b}^*$. We seek the translation that results in a maximal number of occupied orthants of $\mathbf{R}^n$. The number of orthants occupied is the number of distinct Boolean vectors $[\mathbf{b}_\alpha - \mathbf{b}^* \geq 0] = [\mathbf{b}_\alpha \geq \mathbf{b}^*]$.

Let $\hat{m}(n)$ be the maximum number of orthants that can be occupied given any translation $\mathbf{b}^*$. This function behaves like our growth function $m(n)$. It is either identically 2^n, or there is some $n = d + 1$ such that $\hat{m}(n) < 2^n$. In this case, Haussler bounds the size of an ϵ-separation, and hence an ϵ-net, by a function of ϵ^{-1} raised to the exponent d. Again we can call d the VC dimension, and we have uniform convergence.

Let's now turn to an example. Our functions g_α are the lines $g_\alpha(x) = mx + b$, where $m, b, x \in [0, 1]$. Our index α will be the ordered pair (m, b). The range of

these functions is $[0, 2]$. Our samples (x_i, y_i) will have $x_i \in [0, 1]$ and $y_i \in [0, 2]$. For a choice of n samples, we want to investigate the structure of the points $\mathbf{b}_\alpha = \left(\mathbf{y} - (m\mathbf{x} + b)\right)^2$.

First lets consider the points $\hat{\mathbf{b}}_\alpha = \mathbf{y} - m\mathbf{x} - b$. Suppose we have 2 samples (x_1, y_1) and (x_2, y_2). Then our points $\hat{\mathbf{b}}_\alpha$ are $(y_1 - mx_1 - b, y_2 - mx_2 - b) \in [-2, 2]^2$. Here x_1, y_1, x_2, y_2 are constants, and we have $m, b \in [0, 1]$ Provided $x_1 \neq x_2$, these points $\hat{\mathbf{b}}_\alpha$ form a parallelogram in the plane.

The points $\mathbf{b}_\alpha$ also form a region in the plane. This region is related to the parallelogram in the following way. We transform the points $\hat{\mathbf{b}}_\alpha$ to $\mathbf{b}_\alpha$ by first taking the absolute value of each coordinate, and then squaring it. By taking the absolute value, we reflect the parallelogram in the two axes, folding it into a single quadrant. Squaring then replaces the boundary lines by segments of a parabola. Since the points $\mathbf{b}_\alpha$ form a region of the plane, they have metric dimension 2. And, if we were to move the origin anywhere inside this region, then we would occupy all 4 quadrants, and thus the combinatorial dimension is at least 2.

Now let's add a third sample (x_3, y_3). Since the ordering of the samples does not effect the size of an ϵ-net, we will assume that x_3 lies between x_1 and x_2, that is, $x_1 \leq x_3 \leq x_2$. Again we will consider first the points $\mathbf{b}_\alpha = \mathbf{y} - m\mathbf{x} - b$.

Consider any function $g_\alpha(x) = mx + b$. Let

$$\begin{aligned} s &= y_1 - mx_1 - b \\ t &= y_2 - mx_2 - b \end{aligned}$$

and thus

$$\begin{aligned} x_1 &= -\frac{s + b - y_1}{m} \\ x_2 &= -\frac{t + b - y_2}{m}. \end{aligned}$$

Let $x_3 = \lambda x_1 + (1 - \lambda)x_2$. Hence we get

$$\begin{aligned} y_3 - mx_3 - b &= y_3 + \lambda(s + b - y_1) + (1 - \lambda)(t + b - y_2) \\ &= \lambda s + (1 - \lambda)t + y3 - \lambda y_1 + (1 - \lambda)y_2. \end{aligned}$$

Hence the points $\hat{\mathbf{b}}_\alpha$ are of the form $(s, t, \lambda s + (1 - \lambda)t + \gamma)$, where $\gamma = y3 - \lambda y_1 + (1 - \lambda)y_2$ and λ are constants, and thus the points form a planar region. The points $\mathbf{b}_\alpha$ now form a 2-dimensional surface, and again their metric dimension is 2.

The bits $\hat{\mathbf{b}}_\alpha$ form a planar region, and since a plane can not occupy all 8 octants of 3 space, these points have combinatorial dimension 2. The bits $\mathbf{b}_\alpha$, however, form a curved surface in 3 space. Such a surface can occupy all 8 octants, and thus the combinatorial dimension is at least 3. These dimensions are not required

to agree, as they represent different techniques for bounding the size of a minimal ϵ-net.

CHAPTER 4

Invariance Hints

Suppose we have an invariant defined by a partition $X = \bigcup_\beta X_\beta$. We have seen that we can "learn" this invariant by using gradient descent on the error function

$$E_I(\alpha) = \frac{1}{n}\sum_{i=1}^{n}\big(g_\alpha(x_i) - g_\alpha(x_i')\big)^2.$$

As with learning an unknown function, we need to know that $E(\alpha)$ is, in probability, a good estimate of the actual error

$$I_I(\alpha) = \iint_{X\times X}\big(g_\alpha(x) - g_\alpha(x')\big)^2 P(x,x')\,dx\,dx'.$$

Note that the range of the functions g_α is the same as the range of f, and thus we also have $0 \le \big(g_\alpha(x) - g_\alpha(x')\big)^2 \le c$. Since these functions are uniformly bounded, we can apply the general VC result directly. We have only to define the distribution $P(x,x')$, and investigate the growth function for this system.

One might imagine picking examples of the invariant (x,x') by choosing x at random, and then choosing x' from X_β where $x \in X_\beta$. This gives us the distribution $P(x,x') = P(x)P(x'|X_\beta)$ where $x \in X_\beta$,

$$P(x'|X_\beta) = \begin{cases} P(x')/P_\beta & \text{if } x' \in X_\beta, \\ 0 & \text{otherwise,} \end{cases}$$

and $P_\beta = \int_{X_\beta} P(x)\,dx$.

As we've defined the probability distribution $P(x,x')$ it is defined over all of $X \times X$. However, the only interesting part of $X \times X$ is $\bigcup_\beta X_\beta \times X_\beta$ since outside this subset, $P(x,x')$ is identically 0.

We are left now with the growth function. What we want to do is bound this growth function in terms of the growth function for the set of functions $\{g_\alpha\}$. We will look at the growth function in two parts; first, for Boolean-valued functions, and then for real-valued functions.

1. Boolean-Valued Functions

Suppose we have n examples of the invariant (x_i, x_i'). For each function g_α we define a function $h_\alpha(x, x') = \big(g_\alpha(x) - g_\alpha(x')\big)^2$. Again, since the functions g_α are Boolean, we have $h_\alpha(x_i, x_i') = [g_\alpha(x_i) \neq g_\alpha(x_i')] = g_\alpha(x_i) \oplus g_\alpha(x_i')$. Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ and $\mathbf{x}' = (x_1', x_2', \ldots, x_n')$. Just as $m(n)$ is the maximum number of distinct vectors $g_\alpha(\mathbf{x})$, we will define the growth function for the invariant, $m_I(n)$, to be the maximum number of distinct vectors $h_\alpha(\mathbf{x}, \mathbf{x}') = \big(h_\alpha(x_1, x_1'), \ldots, h_\alpha(x_n, x_n')\big)$.

Note that there could be as many as $m(n)$ different binary vectors $g_\alpha(\mathbf{x})$, and similarly as many as $m(n)$ different vectors $g_\alpha(\mathbf{x}')$. Thus there could be as many as $m(n) \cdot m(n)$ expressions $g_\alpha(\mathbf{x}) \oplus g_\alpha(\mathbf{x}')$. In the worst case, each expression would lead to a distinct vector $h_\alpha(\mathbf{x}, \mathbf{x}')$, in which case we have $m_I(n) \leq \big(m(n)\big)^2$. This worst case can not actually arise, however.

THEOREM 4.1. If $m(n) > 1$ then $m_I(n) < \big(m(n)\big)^2$.

PROOF. The function $m(n)$ is non-decreasing, so if $m(n) = 1$ for some n, then we must have $m(1) = 1$. This means that there can be only a single function g_α, for if there were two, they would necessarily differ somewhere, and using such a point, we would find $m(1) = 2$. Thus the restriction that $m(n)$ be greater than 1 is simply eliminating a degenerate case.

Also, we have $m_I(n) \leq 2^n$, so if $m(n) > \sqrt{2^n}$ then the theorem trivially holds. Thus we will consider the case $1 < m(n) \leq \sqrt{2^n}$. Note that this implies that $n \geq 2$.

What we will show is that if we have $m(n)$ vectors $g_\alpha(\mathbf{x})$ and $m(n)$ vectors $g_\alpha(\mathbf{x}')$, then having all possible $\big(m(n)\big)^2$ combinations will imply that the growth function for $\{g_\alpha\}$ is greater than $m(n)$, which is a contradiction.

We will first look at the vectors $\mathbf{b}_\alpha = g_\alpha(\mathbf{x})$, and show that we can remove one of the n bits, projecting the vectors to an $(n-1)$-dimensional space, and be left with m vectors, where m is more than half the original number of vectors.

To find such a bit we will proceed iteratively. We will use the notation $\mathbf{b}_\alpha[k]$ to indicate the first k bits of $\mathbf{b}_\alpha$. Let $m_0 = m(n)$, and $\big|\{\mathbf{b}_\alpha[n]\}\big| = m_0$.

We remove the last bit, and let $m_1 = \big|\{\mathbf{b}_\alpha[n-1]\}\big|$. Note that we must have $m_1 \geq \frac{1}{2}m_0$. If $m_1 > \frac{1}{2}m_0$, then we are are done. If $m_1 = \frac{1}{2}m_0$, then each vector $\mathbf{b}_\alpha[n-1]$ can be extended both by a 0 and a 1, and we continue, now removing the last bit from each vector $\mathbf{b}_\alpha[n-1]$.

If, at some point we have $m_{i+1} > \frac{1}{2}m_i$ then we are done. Bit $n-i$ is the required bit. Since we had reductions of exactly half through $m_i = \big|\{\mathbf{b}_\alpha[n-i]\}\big|$, we have $m_i = 2^{-i}m_0$ and each vector $\mathbf{b}_\alpha[n-i]$ can be extended by all possible 2^i i-bit vectors. Thus deleting only bit $n-i$ will give us $m_{i+1} \cdot 2^i > \frac{1}{2}m_i \cdot 2^i = \frac{1}{2}m_0$ vectors, as required.

We can continue this process only as far as reducing the set of vectors to a single vector. This must happen, if we don't stop earlier, since we can reduce

$n-1$ times, and we have at most $\sqrt{2^n} \leq 2^{n-1}$ vectors to begin with. Suppose this happened after j reductions. Since $j \leq n-1$, we have $n-j \geq 1$ and hence the remaining vector is at least 1 bit long. This means that $\{\mathbf{b}_\alpha[n-j]\}$ is a singleton, and the 2^j original vectors are all possible j-bit extensions to this single vector. Thus we can choose any bit from $\mathbf{b}_\alpha[n-j]$ as our required bit; removing that bit alone will leave the number of vectors $\mathbf{b}_\alpha$ unchanged.

Removing a single bit is equivalent to removing a point x_i from the vector $\mathbf{x}$. Thus we have shown that $m(n-1) > \frac{1}{2}m(n)$. We will now pick a point x'_j from $\mathbf{x}'$, and show that, along with the $n-1$ points from $\mathbf{x}$, we have the growth function $m(n) \geq 2m(n-1) > m(n)$, a contradiction.

We assumed that each vector $\mathbf{b}_\alpha$ was paired up with each vector $\mathbf{b}'_\alpha$. We also assumed that $m(n) > 1$, so there are at least 2 distinct vectors $\mathbf{b}'_\alpha$, and hence they differ in at least one bit. Let this bit be j, and the corresponding point x'_j is the bit we sought.

Thus for each of the m distinct vectors on $n-1$ points from $\mathbf{x}$, we can extend this vector by both a 0 and a 1 by adding the point x'_j. The number of distinct vectors is thus $2m > m(n)$, which is a contradiction. □

In showing that $m_I(n) < \big(m(n)\big)^2$, we relied only on demonstrating that you can not actually generate all $\big(m(n)\big)^2$ possible expressions of the form $g_\alpha(\mathbf{x}) \oplus g_\alpha(\mathbf{x}')$. We can obtain a stronger result by considering the vectors $\mathbf{x}$ and $\mathbf{x}'$ together. Let $\mathbf{xx}'$ be the concatenation of $\mathbf{x}$ and $\mathbf{x}'$, and consider the $2n$-bit vectors $g_\alpha(\mathbf{xx}')$. We earlier suggested, in effect, that there could be as many as $\big(m(n)\big)^2$ distinct vectors $g_\alpha(\mathbf{xx}')$. However, the number is necessarily bounded by $m(2n)$. Hence we get $m_I(n) \leq m(2n)$.

We have shown that $m_I(n)$ may be larger than $m(n)$. How much larger can the corresponding VC dimension be?

Suppose the VC dimension of $\{g_\alpha\}$ is d. Then, for $n \leq d$, $m(n) = 2^n$, and for $n > d$, $m(n) < 2^n$. Also, for all n, we have $m(n) < n^{d+1} + 1$. We want to find an n such that $m_I(n) < 2^n$, or, using our inequality, $m(2n) < 2^n$. Such a solution will have $n > d$, so we may use the inequality $m(2n) < 1.5(2n)^d/d!$. We have $\frac{2}{3}d! > (d/e)^d$, by Stirling's approximation, so we have $m(2n) < e^d \cdot (2n/d)^d$. Solving for $e^d \cdot (2n/d)^d < 2^n$, we get $(n/d)^d < 2^{n-\gamma d}$ where $\gamma = 1 + \log_2 e$, and, taking logarithms, $\log_2(n/d) < (n/d) - \gamma$. Solving this, we find that for $n/d \gtrapprox 4.7$, or $n \gtrapprox 4.7d$, we have $m_I(n) < 2^n$. Thus the VC dimension of $\{h_\alpha\}$ can be no more than about 4.7 times larger than that of $\{g_\alpha\}$.

Let's return briefly to our other inequality, $m_I(n) < \big(m(n)\big)^2$, and find an n' such that $m_I(n') < 2^{n'}$. Here we would want to find an n' such that $\big(m(n')\big)^2 < 2^{n'}$, or, equivalently, such that $m(n') < 2^{\frac{1}{2}n'}$. Above we found an n such that $m(2n) < 2^n$, so the required n' is exactly double, or $n' \gtrapprox 9.4d$.

2. Real-Valued Functions

Again we consider n examples of the invariant (x_i, x_i') and, as in the Boolean case, we define the functions $h_\alpha(x_i, x_i') = \big(g_\alpha(x_i) - g_\alpha(x_i')\big)^2$, and as before, we will consider the vectors $h_\alpha(\mathbf{x}, \mathbf{x}')$. This time, however, these vectors are points within an n-dimensional hypercube. For what follows, let's assume that the functions g_α map X to $[-1, 1]$, and thus the hypercube is $[0, 4]^n$. The particular choices for these ranges is not important, so long as there is a uniform bound on $h_\alpha(x_i, x_i')$ as required for the generalized VC result.

Our goal will be to construct an ϵ-net for the vectors $h_\alpha(\mathbf{x}, \mathbf{x}')$. As suggested by the Boolean case, we want to look at the concatenated vector $\mathbf{xx}'$, and the corresponding vectors $g_\alpha(\mathbf{xx}')$. For these vectors $g_\alpha(\mathbf{xx}')$, let $\mathbf{g}_1, \mathbf{g}_2, \ldots, \mathbf{g}_k$ be an ϵ-net. Let $g_{i,j}$ refer to the jth component of the vector $\mathbf{g}_i$. Define the vectors $\mathbf{h}_i$ in terms of the vectors $\mathbf{g}_i$ by

$$h_{i,j} = \big(g_{i,j} - g_{i,(j+n)}\big)^2,$$

for $j = 1, \ldots, n$. We know that for any vector $g_\alpha(\mathbf{xx}')$, it is within ϵ of one of the vectors $\mathbf{g}_i$. We will show that the corresponding vector $h_\alpha(\mathbf{x}, \mathbf{x}')$ is within 12ϵ of the corresponding vector $\mathbf{h}_i$. Hence the vectors $\mathbf{h}_i$ form a (12ϵ)-net for the vectors $h_\alpha(\mathbf{x}, \mathbf{x}')$, and thus an upper bound for the minimum possible (12ϵ)-net. Then $N_I(12\epsilon, n) \leq N(\epsilon, 2n)$, where $N_I(\epsilon, n)$ is the expected value for the minimum size of an ϵ-net for $\{h_\alpha(\mathbf{x}, \mathbf{x}')\}$, and, as before, $H_I(\epsilon, n) = \log_2 N_I(\epsilon, n)$.

Since the vectors $\mathbf{g}_i$ form an ϵ-net for $\{g_\alpha(\mathbf{xx}')\}$, we have the following, for $j = 1, \ldots, n$

$$\begin{aligned} g_{i,j} &= g_\alpha(x_j) + \delta_j \\ g_{i,(j+n)} &= g_\alpha(x_j') + \gamma_j, \end{aligned}$$

where $|\delta_j|, |\gamma_j| \leq \epsilon$. Thus we get, for $\epsilon \leq 1$,

$$\begin{aligned} \big|h_\alpha(x_j, x_j') - h_{i,j}\big| &= \Big|\big(g_\alpha(x_j) - g_\alpha(x_j')\big)^2 - \big(g_{i,j} - g_{i,(j+n)}\big)^2\Big| \\ &= \big|2\big(g_\alpha(x_j) - g_\alpha(x_j')\big)(\delta_j - \gamma_j) + (\delta_j - \gamma_j)^2\big| \\ &\leq 2\big|g_\alpha(x_j) - g_\alpha(x_j')\big| \cdot |\delta_j - \gamma_j| + |\delta_j - \gamma_j|^2 \\ &\leq 2 \cdot 2 \cdot 2\epsilon + (2\epsilon)^2 \\ &\leq 12\epsilon, \end{aligned}$$

as required.

Recall that the necessary and sufficient condition for uniform convergence is

$$\lim_{n \to \infty} \frac{H(\epsilon, n)}{n} = 0$$

for all $\epsilon > 0$. Then, since $N_I(\epsilon, n) \leq N(\frac{1}{12}\epsilon, 2n)$ we must have uniform convergence within our set of functions $\{h_\alpha\}$ if we have uniform convergence within our set of functions $\{g_\alpha\}$.

3. Graph View

Our results so far for the growth function have not taken into account any specific information about the invariant itself. Indeed, about the only piece of information that we have used is that n examples of the invariant (x_i, x_i') collectively contain $2n$ points from X, without any regard for the partition on X. In this section we will attempt to remedy this situation.

Our goal is to choose n examples of the invariant so that the number of different vectors $h_\alpha(\mathbf{x}, \mathbf{x}')$ is maximal.

We know that for any function h_α, we have $h_\alpha(x, x) = 0$. Thus we would never want to choose such a pair among the n examples. The vectors $h_\alpha(\mathbf{x}, \mathbf{x}')$ would all have 0 in some coordinate, and so the number of vectors would be the same as the number on $n - 1$ examples, where we drop the example (x, x).

For similar reasons, we would not want to repeat any example (x, x'). Again, dropping the second instance of the pair would leave exactly the same number of vectors $h_\alpha(\mathbf{x}, \mathbf{x}')$, but on $n - 1$ examples.

Further, the functions h_α are symmetric, that is, $h_\alpha(x, x') = h_\alpha(x', x)$. Thus the only interesting choices for examples of the invariant are unordered pairs (x, x') where $x \neq x'$.

Let's now look at the simplest case. We will assume that the set X is finite, and all our functions g_α are Boolean.

First, we will look at a simple example, where $X = \{x_1, \ldots, x_7\}$, and the invariant is defined by the partition $X_1 = \{x_1, x_2, x_3, x_4\}$ and $X_2 = \{x_5, x_6, x_7\}$.

Let us consider the elements of X to be vertices of a graph, and examples (x, x') of the invariant to be edges in the graph. Thus, for our example, the graph with all possible edges is shown in figure 4.1. This graph, $\mathscr{K}$, is a pair of cliques, one on 4 vertices and the other on 3.

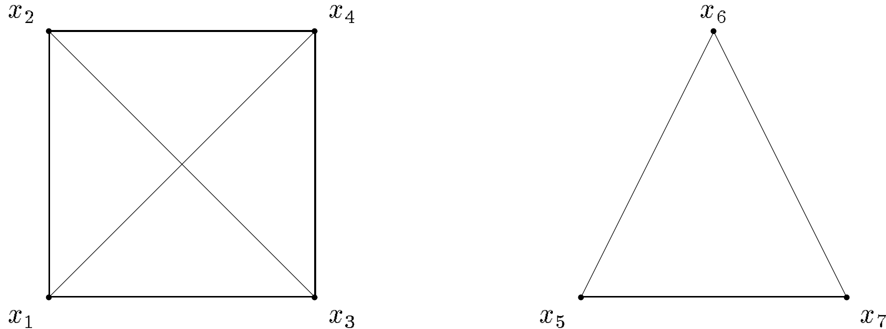

FIGURE 4.1. The graph $\mathscr{K}$

Suppose we choose 4 examples of the invariant (x_1, x_2), (x_2, x_3), (x_3, x_1), and (x_6, x_7). We consider these 4 pairs to be edges in a graph; this graph $\mathscr{G}$ will be a subgraph of $\mathscr{K}$. This subgraph is shown in figure 4.2.

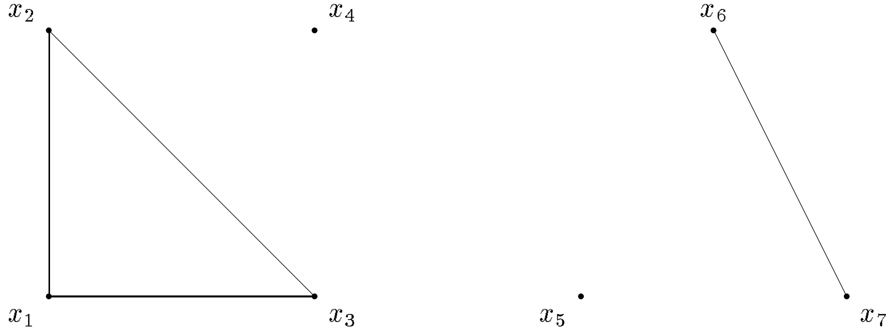

FIGURE 4.2. The subgraph $\mathscr{G}$

Now, suppose we choose some function $g \in \{g_\alpha\}$; let this function be

$$g(x_i) = \begin{cases} 1 & \text{if } i \text{ is odd,} \\ 0 & \text{if } i \text{ is even.} \end{cases}$$

Corresponding to this function g is the function $h(x, x') = \big(g(x) - g(x')\big)^2 = g(x) \oplus g(x')$. We can use these two functions g and h to label the vertices and edges, respectively. In figure 4.3, we use a solid circle to represent a vertex that was labelled 1, and an open circle for a vertex labelled 0.

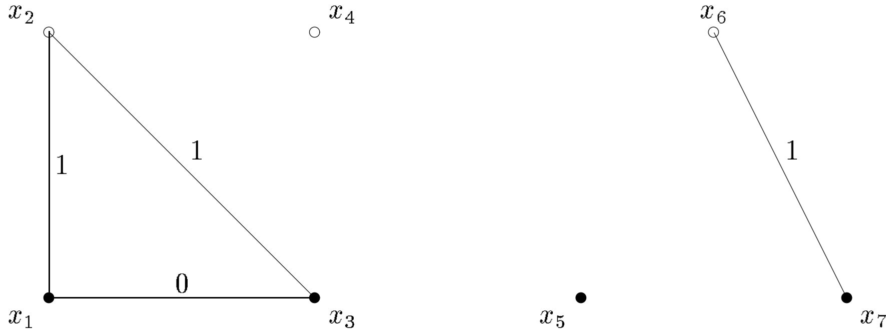

FIGURE 4.3. The subgraph $\mathscr{G}$, labelled by g and h

In our graph, we have a cycle x_1, x_2, and x_3. In this cycle we have 2 edges labelled 1; in general, for any function $g \in \{g_\alpha\}$, the number of edges labelled 1 must be even. If we start at, say, x_1, then move to x_2, the label on x_2 is the

same as that on x_1 if and only if the edge label was 0. This follows directly from $h(x_1, x_2) = g(x_1) \oplus g(x_2)$, or,

$$g(x_2) = h(x_1, x_2) \oplus g(x_1).$$

Similarly, continuing to x_3 and back to x_1 we have

$$g(x_3) = h(x_2, x_3) \oplus g(x_2),$$
$$g(x_1) = h(x_3, x_1) \oplus g(x_3).$$

Combining these we get

$$g(x_1) = h(x_3, x_1) \oplus h(x_2, x_3) \oplus h(x_1, x_2) \oplus g(x_1),$$

and hence, cancelling $g(x_1)$,

$$0 = h(x_3, x_1) \oplus h(x_2, x_3) \oplus h(x_1, x_2).$$

Thus, around our cycle, the exclusive or of the edge labels must be zero. This is not particular to our cycle; in the above, all we have assumed is that we return to where we started, allowing us to cancel $g(x)$ where x is the starting vertex. So for any cycle, the exclusive or of the edge labels must be zero, and thus we must have an even number of edges labelled 1, as required.

For our choice, then, of 4 edges, we have one edge which is redundant, in that for the vectors $h_\alpha(\mathbf{x}, \mathbf{x}')$, the coordinate corresponding to any one edge in the cycle depends strictly on the coordinates of the other two, and hence we have exactly as many different vectors $h_\alpha(\mathbf{x}, \mathbf{x}')$ as we would if we deleted that redundant edge.

The restriction that a cycle contain an even number of edges labelled with 1 covers our earlier restrictions against loops (x, x) on a single vertex, and duplicated edges. In the case of loops, we have a cycle containing one edge. Our requirement of an even number of ones then means that the edge must be labelled 0, and hence is redundant. In the case of a duplicated edge, we have a cycle containing two edges. Thus we must have neither or both labelled with 1, and thus either depends on the other, making one of the two edges redundant.

Thus we do not want a cycle of any kind, whether it's a loop on a single vertex, a pair of edges between the same two vertices, or a cycle on 3 or more vertices. Let's return to a more general setting.

Let the m points of X be vertices of a graph. Our input set is divided into k parts, $X = \bigcup_{j=1}^{k} X_j$. We will place an edge between two points x and x' if they are both from a single X_j and if $x \neq x'$. Since there are k elements in our partition of X, the graph is a collection of k cliques. Let us call this graph $\mathscr{K}$.

Again we will choose n pairs (x_i, x_i'), and we will generate a subgraph $\mathscr{G}$ of $\mathscr{K}$ that has only the edges (x_i, x_i'). We seek n pairs that maximize the number of distinct vectors $h_\alpha(\mathbf{x}, \mathbf{x}')$.

As before, we will not permit any loops (x_i, x_i), any duplicated pair (x_i, x_j), nor any cycles in the subgraph $\mathscr{G}$. All of these lead to a redundant edge. We will again label this subgraph, based on a given, but arbitrary, function g_α. We label each vertex x_i by $g_\alpha(x_i)$, and each edge (x_i, x_i') by $h_\alpha(x_i, x_i')$.

Each connected component of $\mathscr{G}$ must be cycle free, and hence a tree. Thus $\mathscr{G}$ is a collection of trees, or a forest. This sets an upper bound on the number of edges we can have in the subgraph $\mathscr{G}$. Recall that the graph $\mathscr{K}$ is a collection of cliques. Take any one of those cliques, and suppose it has j vertices. We know that if we have as many as j edges in a graph with j vertices, we must have created a cycle. Further, with $j-1$ edges, we can produce a spanning tree. Thus, in our subgraph $\mathscr{G}$, we can have no more than $j-1$ edges on those j vertices from our designated clique. Since we have a clique on these j vertices, we have no other restrictions, beyond avoiding cycles, when choosing up to $j-1$ of the $\binom{j}{2}$ edges. $\mathscr{K}$ contains k cliques, and a total of m vertices, so the maximum number of edges in our subgraph $\mathscr{G}$ is $m-k$. This gives an absolute upper bound of 2^{m-k} on the growth function $m_I(n)$.

Consider the boundary cases. Suppose that the number of cliques is m. This means there is a single vertex x in each X_j. Since we don't permit loops, our graph $\mathscr{K}$ has no edges, and hence so must our subgraph. In this case, however, the only examples of the invariant we would be able to generate would be of the form (x, x), and we know that $h_\alpha(x, x) = 0$. Hence the growth function $m_I(n)$ would be identically 1, and the corresponding VC dimension would be 0. This extreme example corresponds to a null invariant, since it doesn't make any assertions about pairs (x, x') where $x \neq x'$.

On the other extreme is a partition of a single element, $X = X_1$. This corresponds to a universal invariant, that asserts that everything is equivalent to everything else.

In this case we have m vertices, and at most $m-1$ edges. Suppose we have $m-1$ edges, which must form a spanning tree. Let us label the vertices by some function g_α. This gives rise to a unique labelling of the edges, where edge (x, x') is labelled by $h_\alpha(x, x') = g_\alpha(x) \oplus g_\alpha(x')$. Let the function g_α^c be the complement of g_α, so $g_\alpha^c(x) = 1 \oplus g_\alpha(x)$, and now label the vertices by g_α^c. Then we have

$$\begin{aligned} h_\alpha^c(x, x') &= g_\alpha^c(x) \oplus g_\alpha^c(x') \\ &= 1 \oplus g_\alpha(x) \oplus 1 g_\alpha(x') \\ &= g_\alpha(x) \oplus g_\alpha(x') \\ &= h_\alpha(x, x'), \end{aligned}$$

and the edge labels remain the same. We can look at this from the other direction. Suppose we have a labelling of the edges. Then either one of two labellings of the vertices could give rise to this. To see this, we designate any vertex as the root, and, since we have a spanning tree, there is a unique path to every other vertex. If the number of edges labelled 1 in the path is odd, the the vertex is

labelled opposite to the label on the root, otherwise, they are labelled the same. The only remaining variable is the choice of the label for the root, either 0 or 1. This gives us the two labellings of the vertices.

Suppose that we have n pairs, or edges, (x_i, x_i'), rather than $m-1$, as above, and that these edges make up k trees. Then the number of vertices is $n+k$. For those $n+k$ vertices, let the maximum number of labellings induced by the functions g_α be $p \leq m(n+k)$, and, for the n edges, let the maximum number of labellings induced by the functions h_α be $q \leq m_I(n)$. If $k=1$, the n edges form a single tree, and, as above, each labelling of the edges will correspond to either one or two labellings of the vertices, and thus we must have $\frac{1}{2}p \leq q \leq p$. In general, for k trees, we will have $2^{-k}p \leq q \leq p$. Note that the number of trees can not exceed n; having exactly n would require n isolated edges.

Suppose for these particular n pairs (x_i, x_i') we actually attain the growth function, that is $q = m_I(n)$. The number of distinct labellings for the $n+k$ vertices can be as large as $m(n+k)$, and must be at least $m_I(n)$. This tells us that $m_I(n) \leq m(k+n)$. Note that the growth function is non-decreasing, so we have $m_I(n) \leq m(k+n) \leq m(2n)$, as we derived earlier.

Earlier we found that for $m_I(n) \leq m(2n) < 2^n$, we needed to have n about 4.7 times the VC dimension d of the set of functions $\{g_\alpha\}$. We have shown that if the n pairs form k trees, then we have $m_I(n) \leq m(n+k)$. We can, as we did before, solve $m(n+k) < 2^n$ for any value of $1 \leq k \leq n$, this time requiring that $\log_2((n+k)/d) < (n/d) - \log_2(e)$, and thus find the maximum number of times that the VC dimension for $\{h_\alpha\}$ may be larger than the VC dimension for $\{g_\alpha\}$. This is plotted in figure 4.4.

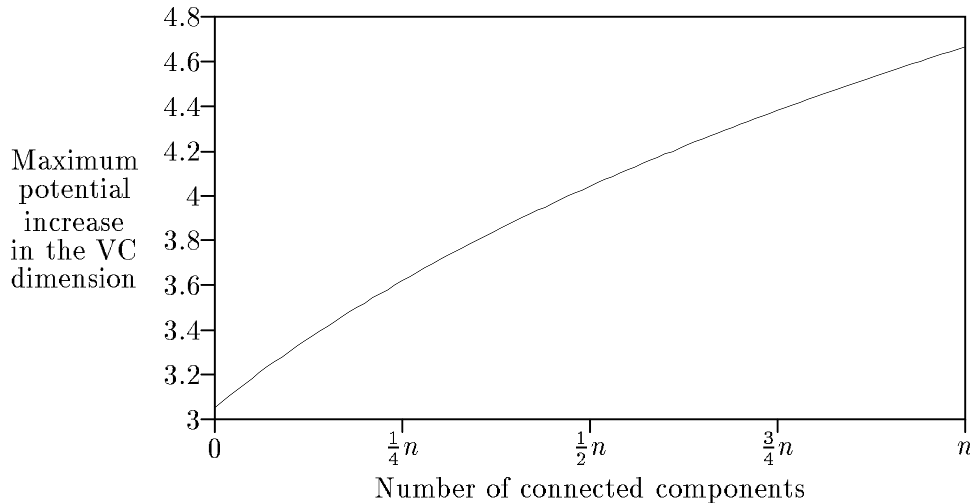

FIGURE 4.4

Figure 4.4 follows our intuition. The number of vertices is equal to the total

number of edges plus the number of connected components. The more vertices we have, the greater the number of possible assignments to the vertices we have, and thus a greater possibility of maximizing the number of assignments to the edges.

The graph suggests with a single component, we might be able to get a VC dimension of $\{h_\alpha\}$ that is about 3 times larger than the VC dimension of $\{g_\alpha\}$. This is surprising. The graph of course represents an upper bound, and may not be a particularly tight one. We will show later that we can have the VC dimension of the set $\{h_\alpha\}$ twice that of $\{g_\alpha\}$.

In the case, then, of the universal invariant, we find that what turns out to be important is the nature of the n pairs we choose, and, in particular, how many trees they produce. Having a single invariant does give us complete latitude in choosing our pairs, since all possible pairings are available.

Suppose our input set X has 6 elements. Disregarding any partition on X, suppose we have 3 disjoint pairs (x_i, x'_i) that maximize the growth function. The pairs are consistent with a partition of X if for each pair (x_i, x'_i) we have $x_i, x'_i \in X_\beta$. These pairs are necessarily consistent with the universal invariant. They are also consistent with a partition of X into 3 subsets, each of 2 elements, provided that each subset contained exactly one pair. There are 15 ways to partition X into 3 subsets of 2 elements each, yet only 1 is consistent with the 3 pairs (x_i, x'_i). No partition of X into 2 subsets of 3 elements each is consistent with our 3 pairs. Yet a partition into 2 subsets of 4 and 2 elements may be.

We saw that with the null invariant, the growth function $m_I(n)$ was identically 1. With the universal invariant, we do attain the maximum possible value for $m_I(n)$. However, given the pairs (x, x') that maximize $m_I(n)$, it may be possible to partition X into more than a single subset such that we also attain this maximum value for $m_I(n)$, by simply constructing the partition around those given pairs. Thus we find that the nature of the invariant, except in extreme limiting cases, does not provide for a bound on the growth function $m_I(n)$.

Suppose now that our set X is infinite. Our graph $\mathscr{K}$ also becomes infinite; either an infinite number of cliques, or cliques on an infinite number number of vertices, or both. The subgraphs $\mathscr{G}$ will remain finite, containing n edges, and no more than $2n$ vertices. We still reject loops, and duplicate edges. Loops are defined as a pair (x, x). With an infinite input set, we might have a pair (x, x') where $\rho(x, x')$ is arbitrarily small. Would such a pair still be considered a loop?

The question is moot unless we can actually have the pair (x, x'), and thus we will require that both x and x' be in the same subset X_β. If our functions are Boolean, then they can not be continuous, unless they happen to be constant. Thus there will be points x and x' where $\rho(x, x')$ is arbitrarily small, yet there will be a function $g \in \{g_\alpha\}$ such that $|g(x) - g(x')| = 1$. Such a pair (x, x') could only be considered equivalent to a loop if all functions g_α have $|g_\alpha(x) - g_\alpha(x')| = 1$. This will not be typical.

If our functions are real, then they may be continuous. We want, given (x, x')

with $\rho(x, x')$ small, that $|g_\alpha(x) - g_\alpha(x')|$ be small, for all network functions g_α. Suppose we have a Lipschitz condition $|g_\alpha(x) - g_\alpha(x')| \le a\rho(x, x')$ where a is a constant that is uniform over all $\{g_\alpha\}$. Such a condition holds, for example, on our feed-forward networks, provided that we have a limit on the size of the weights.

Then, if $\rho(x, x') < \delta$, we have $\left(g_\alpha(x) - g_\alpha(x')\right)^2 < (a\delta)^2$. Suppose such a pair was one of the n examples of the invariant. Then all the vectors $h_\alpha(\mathbf{x}, \mathbf{x}')$ would be close to one of the coordinate hyperplanes. We are trying instead to fill n space, if possible.

Similarly, if we have two vectors (x_1, x_1') and (x_2, x_2') with $\rho(x_1, x_2) < \delta$ and $\rho(x_1', x_2') < \delta'$, both small, then these distinct edges are close to a duplicate edge. We have $|g_\alpha(x_1) - g_\alpha(x_2)| < a\delta$, or $g_\alpha(x_2) = g_\alpha(x_1) + \gamma$ where $|\gamma| < a\delta$. Similarly, we have $g_\alpha(x_2') = g_\alpha(x_1') + \gamma'$ where $|\gamma'| < a\delta'$. Then

$$\begin{aligned} |h_\alpha(x_1, x_1') - h_\alpha(x_2, x_2')| &= \left(g_\alpha(x_1) - g_\alpha(x_1')\right)^2 - \left(g_\alpha(x_1) + \gamma - g_\alpha(x_1') + \gamma'\right)^2 \\ &\le 2|\gamma + \gamma'| \cdot |g_\alpha(x_1) + g_\alpha(x_2)| + |\gamma + \gamma'|^2. \end{aligned}$$

Here we have the values of the functions $h_\alpha(x_1, x_1')$ and $h_\alpha(x_2, x_2')$ close to one another, and thus, with such a pair of pairs among our n examples, we would again have the vectors $h_\alpha(\mathbf{x}, \mathbf{x}')$ close to a hyperplane, rather than filling n space.

The nature of the invariant has little role here. If the invariant splits up two points x and x' that are close to one another, then we can't have the pair (x, x'), and if it does not, we don't want such a pair anyway. Similarly, two nearly equal pairs are redundant whether they can both be made from a single subset X_β, or if each comes from a different subset of X.

Note that now that we have moved to real-valued functions, we no longer have a restriction on non-trivial cycles. In the Boolean case, as we moved around the cycle, we knew that we must have an even number of edges labelled 1. In the real-valued case, the edge labels are non-negative real numbers, and there is no constraint on the sum around a cycle. The only restrictions we can assert are the ones mentioned above.

4. Examples

Let's look at a few specific examples. We will begin, as always, with the simplest case, a finite number of Boolean functions.

Suppose we have four functions g_1, g_2, g_3 and g_4, defined on the three element set $X = \{x_1, x_2, x_3\}$. We will define these functions so that they have a VC dimension of 1. When we take two pairs, (x_1, x_2), and (x_1, x_3) we will find that the corresponding functions $h_i(x, x') = \left(g_i(x) - g_i(x')\right)^2$ have a VC dimension of 2. This demonstrates that, in general, the VC dimension of the functions h_α can be greater than that of the corresponding functions g_α. In the following table, we list the functions g_i on the left, and the functions h_i on the right.

	x_1	x_2	x_3	x_1 x_2	x_1 x_3	
g_0	0	0	0	0	0	h_0
g_1	0	0	1	0	1	h_1
g_2	0	1	0	1	0	h_2
g_3	1	0	0	1	1	h_3

We will now generalize this example.

Suppose our input set X contain n points, and the functions g_α will be those that assign 1 to at most d of those n points, and assign 0 to the rest. For the above example, we had $n = 3$ and $d = 1$. We will assume further that $n = 2d+1$. This set of functions has been seen earlier; we know that it has VC dimension d. The number of functions is

$$\sum_{i=0}^{d} \binom{n}{i} = \sum_{i=d+1}^{n} \binom{n}{n-i} = \sum_{i=d+1}^{n} \binom{n}{i} = \frac{1}{2} 2^n = 2^{2d}.$$

We can choose as many as $n-1$ pairs (x_i, x_i'), and for those pairs we wish to find the VC dimension of the corresponding functions h_α. We will show for any non-redundant choice of $n-1$ pairs, each of the functions h_α is unique. There are 2^{2d} possible functions on $n-1 = 2d$ input pairs, so this would imply a VC dimension of $2d$.

Let us return to our graph model. The n points in X become vertices, and our pairs (x_i, x_i') edges. Suppose the edges form any spanning tree of these n vertices. This gives us $n-1$ edges, as required. As before, we will label the vertices with $g_\alpha(x_i)$ and the edges with $h_\alpha(x_i, x_i')$. Recall that if two labellings of the vertices give rise to the same labelling of the edges, then the labellings of the vertices must be complements of one another. We therefore can have two functions from $\{h_\alpha\}$ equal to one another only if we have two functions from $\{g_\alpha\}$ that are complements of one another. Each function g_α assigns 1 to at most d points. The complement must therefore assign 1 to at least $n-d = d+1$ points. Therefore, a function and its complement can not both appear, and we have shown that the functions h_α are unique, as required.

What if we had $n < 2d+1$? We can choose as many as $n-1$ pairs (x_i, x_i'), and $n-1 < 2d$. Since the functions h_α are on fewer than $2d$ inputs, we can not obtain a VC dimension of $2d$.

Finally what if $n > 2d+1$? If we choose $2d$ pairs in a single spanning tree, then the above still holds. If we attempt $r > 2d$ pairs in a single spanning tree, then we fail to get all 2^r functions h_α. The r pairs are on $r+1$ vertices, since we have a single spanning tree. The number of distinct functions g_α on these $r+1$ inputs is

$$k = \sum_{i=0}^{d} \binom{r+1}{i} = \sum_{i=r+1-d}^{r+1} \binom{r+1}{r+1-i} = \sum_{i=r+1-d}^{r+1} \binom{r+1}{i}.$$

The first sum is from 0 to d, and the last from $r-d+1$ to $r+1$. Since $r > 2d$, we have $r - d \geq d + 1$, so we have some binomial coefficients missing in the middle, namely $d+1$ through $r-d$, and hence

$$\sum_{i=0}^{r+1} \binom{r+1}{i} = 2k + \sum_{i=d+1}^{r-d} \binom{r+1}{i} = 2^{r+1},$$

giving us $k < \frac{1}{2}2^{r+1}$. Thus we do not have enough functions to obtain a VC dimension of $r > 2d$.

We have shown, given these functions g_α, that the VC dimension of the corresponding functions h_α is $2d$, provided that $n \geq 2d+1$. However, in demonstrating this, we have used a single spanning tree, and hence assumed the universal invariant. We will now generalize this example further, to demonstrate that the VC dimension of $\{h_\alpha\}$ is at most $2d$, for a variety of choices of pairs (x_i, x_i'). To demonstrate a VC dimension of $2d$ for $\{h_\alpha\}$, we need $2d$ edges, and possibly as many as twice that number of vertices. So we will assume that $n \geq 4d$.

Suppose we have r pairs (x_i, x_i') taken from the set X, and these form k trees. Let the size of each tree be r_i. We will show that in order to obtain all 2^r functions h_α, we must have $d \geq \frac{1}{2}r$, or $r \leq 2d$.

Consider any one of these trees. There are two cases to consider. Either r_i is even, or it is odd. If it is even, then we have a tree of $r_i = 2k$ edges, and $2k+1$ vertices. We've looked at this case earlier. If we use labelling of the vertices using at most k ones, then we can generate precisely the 2^{2k} distinct labellings of the edges. Since each labelling appears exactly once, there must be one that requires the use of at least $k = \frac{1}{2}r_i$ ones in the labelling of the vertices.

Suppose r_i is odd, and thus we have a tree with $r_i = 2k-1$ edges and $2k$ vertices. If we use labellings of the vertices up to $k-1$ ones, then we fail to get all possible labellings, since we have $2k-1 > 2(k-1)$, as we saw earlier. If we allow up to k ones, then we do get all 2^{2k-1} possible labelling of the edges. Some labellings of the edges appear twice; those must be the ones that use k ones in the labellings of the vertices since the complement of such a labelling also uses $2k-k=k$ ones. In any event, we once again have a labelling of the edges that requires we use at least $k = \lceil \frac{1}{2}r_i \rceil$ ones in the labelling of the vertices.

We have, for each tree, a particular labelling of the edges that requires at least $\lceil \frac{1}{2}r_i \rceil$ ones in the labelling of the vertices. Combining these, we get a labelling of all the edges that requires at least $\sum_{i=1}^{k} \lceil \frac{1}{2}r_i \rceil \geq \frac{1}{2}r$ ones in the labelling of the vertices. We have available as many as d ones, so we must have $d \geq \frac{1}{2}r$. If all of the r_i are even, that is, if all the trees have an even number of edges, then the total number of edges r must also be even. In this case, all 2^r labellings of the edges are possible if we have $2d = r$. This shows we can have the VC dimension double. If one of the r_i is odd, then we have $\sum_{i=1}^{k} \lceil \frac{1}{2}r_i \rceil > \frac{1}{2}r$ and thus $d > \frac{1}{2}r$.

The VC dimension of the functions h_α is twice that of the functions g_α, provided that the invariant allows us to choose pairs in such a way that the trees

corresponding to them all have an even number of edges. Here we find what was suggested earlier, that the particular nature of the invariant itself has a limited effect.

In this example, we found that it is disadvantageous to use r disjoint pairs (x_i, x_i'), since these form trees with an odd number of edges. In such a case, we have $r_i = 1$ for all i, and thus the number of ones required in the labelling of the vertices is at least $\sum_{i=1}^{k} \lceil \frac{1}{2} \rceil = k = r$. In this case we would only obtain 2^r distinct labellings of the edges, far fewer than the 2^{2r} we could get if we used, say, pairs of pairs with a common point between them.

This example suggests a number of questions. Is it always better if we don't take disjoint pairs? Since this particular set of function $\{g_\alpha\}$ is maximal in the sense that you can not have a set with more functions and still have a VC dimension of d, is it also maximal with respect to the change in the VC dimension of $\{h_\alpha\}$ over $\{g_\alpha\}$, giving us a bounding factor of twice rather than the factor of about 4.7 we derived earlier? Is it only the number of edges within each tree that matters? The answers are no, no, and yes.

We can demonstrate a "no" by providing a counterexample; we will answer the first two questions with the following example. We have a set of 32 functions g_i on 10 inputs $x_0, \ldots, x_9$. These functions were chosen to have a VC dimension of 2. The largest set of functions on 10 inputs with a VC dimension of 2 is $\binom{10}{0} + \binom{10}{1} + \binom{10}{2} = 56$, so our set $\{g_i\}$ is not maximal. We will choose 5 disjoint pairs for the corresponding functions h_i, and use these pairs to generate 32 distinct vectors $h_i(\mathbf{x}, \mathbf{x}')$, demonstrating that the functions h_i have a VC dimension of 5, 2.5 times larger. These functions are listed in table 4.1.

The final answer is "yes." Consider a tree of n edges on $n+1$ vertices. We seek the VC dimension, so we are required to generate all 2^n possible labellings of the edges. We know that each such labelling corresponds to one of two labellings of the vertices, and those two are complements of each other. So we consider the set of all 2^{n+1} labellings of the vertices and pair each up with its complement, giving us exactly 2^n pairs. We must have at least one of each of these 2^r pairs of labellings, and collectively, they must generate all possible labelling of the edges, no matter what the nature of the spanning tree.

Now let's return to our example of real-valued functions $g_\alpha(x) = mx + b$, where $m, b, x \in [0, 1]$. These functions are a poor choice for fitting data that satisfies an invariant. Suppose for our unknown function f, we have $f(x) = f(x')$, where $x \neq x'$. Suppose a function g_α satisfies this invariant. Then we have

$$mx + b = mx' + b,$$
$$mx = mx',$$

and since $x \neq x'$, we must have $m = 0$ and hence $g_\alpha(x) = b$. Nevertheless, we will compute the VC dimension of the set of functions $h_\alpha(x, x') = \big(g_\alpha(x) - g_\alpha(x')\big)^2 = m^2(x - x')^2$. Since we have a single parameter here, namely m, rather than 2,

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_0 x_1	x_2 x_3	x_4 x_5	x_6 x_7	x_8 x_9	
g_0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	h_0
g_1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	h_1
g_2	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	h_2
g_3	1	1	0	0	0	0	0	1	1	0	0	0	0	1	1	h_3
g_4	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	h_4
g_5	0	0	0	0	0	1	0	0	0	1	0	0	1	0	1	h_5
g_6	0	0	0	0	0	1	1	0	0	0	0	0	1	1	0	h_6
g_7	0	0	0	0	1	0	1	0	1	0	0	0	1	1	1	h_7
g_8	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	h_8
g_9	0	0	1	0	0	0	0	0	1	0	0	1	0	0	1	h_9
g_{10}	0	0	1	0	0	0	1	0	0	0	0	1	0	1	0	h_{10}
g_{11}	0	0	1	0	0	0	0	1	1	0	0	1	0	1	1	h_{11}
g_{12}	0	0	0	1	0	1	0	0	0	0	0	1	1	0	0	h_{12}
g_{13}	0	0	0	1	1	0	0	0	1	0	0	1	1	0	1	h_{13}
g_{14}	0	0	0	1	1	0	1	0	0	0	0	1	1	1	0	h_{14}
g_{15}	0	0	0	1	1	0	1	0	0	1	0	1	1	1	1	h_{15}
g_{16}	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	h_{16}
g_{17}	0	1	0	0	0	0	0	0	1	0	1	0	0	0	1	h_{17}
g_{18}	0	1	0	0	0	0	1	0	0	0	1	0	0	1	0	h_{18}
g_{19}	1	0	0	0	0	0	0	1	0	1	1	0	0	1	1	h_{19}
g_{20}	1	0	0	0	0	1	0	0	0	0	1	0	1	0	0	h_{20}
g_{21}	1	0	0	0	1	0	0	0	1	0	1	0	1	0	1	h_{21}
g_{22}	0	1	0	0	1	0	1	0	0	0	1	0	1	1	0	h_{22}
g_{23}	0	1	0	0	1	0	1	0	0	1	1	0	1	1	1	h_{23}
g_{24}	1	0	1	0	0	0	0	0	1	1	1	1	0	0	0	h_{24}
g_{25}	1	0	1	0	0	0	0	0	1	0	1	1	0	0	1	h_{25}
g_{26}	0	1	1	0	0	0	1	0	0	0	1	1	0	1	0	h_{26}
g_{27}	1	0	1	0	0	0	1	0	1	0	1	1	0	1	1	h_{27}
g_{28}	1	0	0	1	0	1	0	0	0	0	1	1	1	0	0	h_{28}
g_{29}	1	0	1	0	1	0	0	0	1	0	1	1	1	0	1	h_{29}
g_{30}	1	0	0	1	0	1	0	1	0	0	1	1	1	1	0	h_{30}
g_{31}	1	0	1	0	1	0	0	1	1	0	1	1	1	1	1	h_{31}

TABLE 4.1

both m and b, in the case of the functions g_α, we will find the VC dimension to be 1.

Suppose we have a single example of the invariant (x_1, x_1'). The range of the functions $h_\alpha(x_1, x_1')$ is the interval $[0, (x_1 - x_1')^2]$. This is a line segment, provided $x_1 \neq x_1'$, and thus the metric dimension is 1.

Now let's add a second example (x_2, x_2'). The functions h_α give us the points $\left(m^2(x_1 - x_1')^2, m^2(x_2 - x_2')^2\right)$. Suppose $(x_2 - x_2') = \lambda(x_1 - x_1')$. We will assume that $x_2 \neq x_2'$. Then the points become

$$\left(m^2(x_1 - x_1')^2, m^2(x_2 - x_2')^2\right) = \left(m^2(x_1 - x_1')^2, m^2\lambda^2(x_1 - x_1')^2\right) = (s, \lambda^2 s),$$

where $s \in [0, (x_1 - x_1')^2]$. Here again we have a line segment, and again the metric dimension is 1.

A line segment can occupy both the positive and negative segments of 1 space, but at most 3 of the 4 quadrants in 2 space. Thus the points $\mathbf{b}_\alpha$ have combinatorial dimension 1, and, in this case, the combinatorial dimension coincides with the metric dimension.

CHAPTER 5

Learning

We will now investigate specifically how we can incorporate invariants into learning algorithms. We will start with our original two examples, perceptrons and feed-forward neural networks.

1. Perceptrons

Minsky and Papert provide a learning algorithm for perceptrons. Our goal will be to modify this algorithm to learn functions that are invariant under a group of transformations T.

Suppose we have an unknown function f that is T-invariant, and we are given examples (x_i, y_i) of this function. We will split these examples into two sets, those for which the required output y_i is 1 and those for which it is 0. We will consider only the former class; the latter is handled in the same way save for a change of sign.

Recall that a perceptron is a linear threshold function

$$g_\alpha(x) = \left[\sum_{\varphi \in \Phi} w_\varphi \varphi(x) > 0\right] = \mathbf{w} \cdot \Phi(x).$$

The learning algorithm of Minsky and Papert is the following. Take an example x and use this as input to our perceptron. If the output $g_\alpha(x) = 1$, we do nothing. If it is 0, we modify the network. Since the output is 0, we must have had $\mathbf{w} \cdot \Phi(x) < 0$. We want $\mathbf{w} \cdot \Phi(x) > 0$; we move in this direction by adding the vector $\Phi(x)$ to $\mathbf{w}$, since

$$(\mathbf{w} + \Phi(x)) \cdot \Phi(x) = \mathbf{w} \cdot \Phi(x) + \Phi(x) \cdot \Phi(x) > \mathbf{w} \cdot \Phi(x).$$

Minsky and Papert prove that this algorithm will converge after a finite number of changes to the weight vector $\mathbf{w}$.

We want to modify this algorithm to take advantage of the knowledge that the unknown function is T-invariant. We will assume that the set of input functions $\Phi = \{\varphi\}$ is closed under T. The group invariance result of Minsky and Papert

then applies. The set T divides the input functions Φ into equivalence classes. If we have a perceptron $g_\alpha(x)$ that is invariant under T, then there is an equivalent perceptron where the weights $w(\varphi)$ are equal within the equivalence class containing φ, that is, if φ is equivalent to $\hat{\varphi}$, then $w(\varphi) = w(\hat{\varphi})$. This will refer to this as the *equal weight property.*

We know that if we have an example x, then we have another example $t(x)$ for all transformations $t \in T$. Further, we know that if the perceptron is T-invariant, we can assume the weights are equal within the equivalence classes of Φ. What we will do is assume that the perceptron starts out with equal weights within each equivalence class, and ensure that each time we update the network we preserve this property.

Suppose for some x we have $\mathbf{w} \cdot \Phi(x) < 0$. Then, we have $\mathbf{w} \cdot \Phi(t(x)) < 0$ and hence

$$\sum_{t \in T} \mathbf{w} \cdot \Phi(t(x)) = \mathbf{w} \cdot \sum_{t \in T} \Phi(t(x)) < 0.$$

Since scaling won't affect the sign, we also have

$$\mathbf{w} \cdot \frac{1}{|T|} \sum_{t \in T} \Phi(t(x)) < 0.$$

Using the same argument as above, if we add the vector $\mathbf{v} = |T|^{-1} \sum_{t \in T} \Phi(t(x))$ to the vector $\mathbf{w}$, we will increase the value of the dot product. We have then two things to show: first, that this update preserves the equal weight property, and second, that we will still converge to a solution if one exists.

Suppose $\hat{\varphi}$ is equivalent to φ, that is, that $\hat{\varphi}(x) = \varphi \circ s(x)$ for some $s \in T$. Then

$$\begin{aligned}
\sum_{t \in T} \hat{\varphi}(t(x)) &= \sum_{t \in T} \varphi \circ s(t(x)) \\
&= \sum_{t \in T} \varphi(s \circ t(x)) \\
&= \sum_{t \in T} \varphi(t(x))
\end{aligned}$$

since composing each element t of the group of transformations with a fixed transformation s simply returns the group, possibly in a different order. Thus for any two equivalent functions φ and $\hat{\varphi}$, the increment to the weight vector $\mathbf{w}$ is the same. Thus we have preserved the equal weight property of $\mathbf{w}$.

So show convergence, we will follow the proof of Block and Levin [3]. We want to show that if there is a solution, then we will converge to some solution, after a finite number of updates to the perceptron. By a solution, we mean a set of assignments to the weights such that the perceptron correctly classifies all of the examples given. Suppose there is a solution vector $\mathbf{w}^*$. We can assume, by the group invariance theorem, that the weights are equal with the equivalence

classes of Φ. We start with a vector $\mathbf{w}_1$, also with the equal weight property. This vector is otherwise arbitrary. Suppose for some example x, $\mathbf{w}_i \cdot \Phi(x) < 0$. Then $\mathbf{w}_{i+1} = \mathbf{w}_i + \mathbf{v}$, where $\mathbf{v} = |T|^{-1} \sum_{t \in T} \Phi(t(x))$, as before.

Let $a = \min_j(\mathbf{w}^* \cdot \Phi(x_j))$ and $b = \max_j |\Phi(x_j)|$. Note that $\Phi(t(x_j))$ is a permutation of $\Phi(x_j)$, where the elements are reordered within equivalence classes. Thus we have $|\Phi(t(x_j))| = |\Phi(x_j)|$, and, since the weights in $\mathbf{w}^*$ are equal within equivalence classes, we also have $\mathbf{w}^* \cdot \Phi(x_j) = \mathbf{w}^* \cdot \Phi(t(x_j))$. Let $\gamma = b^2/a$. Then

$$\begin{aligned} |\mathbf{w}_{i+1} - \gamma \mathbf{w}^*|^2 &= |\mathbf{w}_i - \gamma \mathbf{w}^* + \mathbf{v}|^2 \\ &= |\mathbf{w}_i - \gamma \mathbf{w}^*|^2 + 2\mathbf{w}_i \cdot \mathbf{v} - 2\gamma \mathbf{w}^* \cdot \mathbf{v} + |\mathbf{v}|^2. \end{aligned}$$

We have $\mathbf{w}_i \cdot \mathbf{v} < 0$. Also,

$$\begin{aligned} |\mathbf{v}|^2 &= |T|^{-2} \sum_{s,t \in T} \Phi(s(x_j)) \cdot \Phi(t(x_j)) \\ &\leq |T|^{-2} \sum_{s,t \in T} |\Phi(s(x_j))| \cdot |\Phi(t(x_j))| \\ &\leq b^2, \end{aligned}$$

and,

$$\mathbf{w}^* \cdot \mathbf{v} = |T|^{-1} \sum_{t \in T} \mathbf{w}^* \cdot \Phi(t(x_j)) \geq a.$$

Combining these we get

$$|\mathbf{w}_{i+1} - \gamma \mathbf{w}^*|^2 \leq |\mathbf{w}_i - \gamma \mathbf{w}^*|^2 - 2\frac{b^2}{a} a + b^2 = |\mathbf{w}_i - \gamma \mathbf{w}^*|^2 - b^2,$$

and thus

$$0 \leq |\mathbf{w}_{i+1} - \gamma \mathbf{w}^*|^2 \leq |\mathbf{w}_1 - \gamma \mathbf{w}^*|^2 - ib^2.$$

Hence we can only update the perceptron a finite number of times before we converge on a solution.

What if we had used the sum of the vectors $\Phi(t(x))$ rather than the average as our update vector $\mathbf{v}$? In this case everything would simply have been scaled by the size of the group $|T|$. In particular, we would have had $|\mathbf{v}|^2 \leq |T|^2 b^2$, $\mathbf{w}^* \cdot \mathbf{v} \geq |T| a$, and the scale factor $\gamma = |T| b^2/a$. The use of the average rather than the sum of the vectors $\Phi(t(x))$ is very much suggestive of the proof of group invariance, however.

Suppose, as before, we have a weight vector $\mathbf{w}$ with elements that are equal within each equivalence class of Φ, and we have an example x_j for which $\mathbf{w} \cdot \Phi(x_j) < 0$. We then obtain a new weight vector $\mathbf{w}'$ by adding $\Phi(x_j)$ to $\mathbf{w}$, according to the original learning algorithm. We cannot assume, now, that the function computed by the new network is T-invariant. Nevertheless, suppose we apply the averaging of the group invariance proof.

We want to compute, then,

$$|T|^{-1}\sum_{t\in T} w'(\varphi\circ t) = |T|^{-1}\sum_{t\in T}\Big(w(\varphi\circ t)+\varphi\circ t(x_j)\Big)$$
$$= w(\varphi) + |T|^{-1}\sum_{t\in T}\varphi\circ t(x_j)$$

and hence we have $\mathbf{w}' = \mathbf{w}+\mathbf{v}$, where $\mathbf{v}$ retains its earlier definition.

2. Feed-Forward Networks

We have defined functionals that measure the distance between our network function g_α and the unknown function f, namely

$$I(\alpha) = \int_X \big(f(x) - g_\alpha(x)\big)^2 P(x)\,dx$$

as well as a functional that measures how closely our network function g_α satisfies the invariant, namely

$$I_I(\alpha) = \iint_{X\times X} \big(g_\alpha(x) - g_\alpha(x')\big)^2 P(x,x')\,dx\,dx'.$$

We have corresponding estimates $E(\alpha)$ and $E_I(\alpha)$ for these, based on examples, and the circumstances under which the estimates are probably accurate. We can apply gradient descent to both estimates to get a learning algorithm. What we need to do is combine all the pieces.

The unknown function f is known to strictly satisfy the invariant. We expect that any function g_α that approximates f well, will also nearly satisfy the invariant. This is proven with the following.

THEOREM 5.1. If $I(\alpha) < \epsilon$, then $I_I(\alpha) < 2\epsilon$.

PROOF. We will show that $I_I(\alpha) \le 2I(\alpha)$, over an arbitrary invariant class X_β. That is, we will show that

$$\iint_{X_\beta\times X} \big(g_\alpha(x) - g_\alpha(x')\big)^2 P(x,x')\,dx\,dx' \le 2\int_{X_\beta}\big(f(x) - g_\alpha(x)\big)^2 P(x)\,dx.$$

We will consider the left and right hand sides of this equation separately. Let

$$L = \iint_{X_\beta\times X} \big(g_\alpha(x) - g_\alpha(x')\big)^2 P(x,x')\,dx\,dx',$$

and

$$R = 2\int_{X_\beta}\big(f(x) - g_\alpha(x)\big)^2 P(x)\,dx.$$

We will show that $R - L \ge 0$.

First we consider the right hand side R. Since f satisfies strictly the invariant, $f(x) = a$ for all $x \in X_\beta$. Hence we get

$$\begin{aligned} R &= 2 \int_{X_\beta} \big(a - g_\alpha(x)\big)^2 P(x)\, dx \\ &= 2 \int_{X_\beta} \big(a^2 - 2a g_\alpha(x) + g_\alpha^2(x)\big) P(x)\, dx \\ &= 2a^2 P_\beta - 4a \int_{X_\beta} g_\alpha(x) P(x)\, dx + 2 \int_{X_\beta} g_\alpha^2(x) P(x)\, dx. \end{aligned}$$

Next we consider the left hand side L. $P(x, x') = P(x)P(x'|X_\beta)$. Since $x \in X_\beta$, we have $P(x'|X_\beta) = P(x')/P_\beta$ for $x' \in X_\beta$, and $P(x'|X_\beta) = 0$ otherwise. Thus we can reduce the integration from $X_\beta \times X$ to $X_\beta \times X_\beta$ and the left hand side becomes

$$\begin{aligned} L &= \frac{1}{P_\beta} \iint_{X_\beta \times X_\beta} \big(g_\alpha^2(x) - 2 g_\alpha(x) g_\alpha(x') + g_\alpha^2(x')\big) P(x)P(x')\, dx\, dx' \\ &= \frac{1}{P_\beta} \iint_{X_\beta \times X_\beta} g_\alpha^2(x) P(x)P(x')\, dx\, dx' \\ &\quad - \frac{2}{P_\beta} \iint_{X_\beta \times X_\beta} g_\alpha(x) g_\alpha(x') P(x)P(x')\, dx\, dx' \\ &\quad + \frac{1}{P_\beta} \iint_{X_\beta \times X_\beta} g_\alpha^2(x') P(x)P(x')\, dx\, dx'. \end{aligned}$$

Let's take each of these three terms in turn. The first term becomes

$$\begin{aligned} \frac{1}{P_\beta} \iint_{X_\beta \times X_\beta} g_\alpha^2(x) P(x)P(x')\, dx\, dx' &= \frac{1}{P_\beta} \int_{X_\beta} g_\alpha^2(x) P(x)\, dx \int_{X_\beta} P(x')\, dx' \\ &= \int_{X_\beta} g_\alpha^2(x) P(x)\, dx. \end{aligned}$$

Similarly, the third term becomes

$$\frac{1}{P_\beta} \iint_{X_\beta \times X_\beta} g_\alpha^2(x') P(x)P(x')\, dx\, dx' = \int_{X_\beta} g_\alpha^2(x') P(x')\, dx'.$$

And the second term becomes

$$\begin{aligned} \frac{2}{P_\beta} \iint_{X_\beta \times X_\beta} g_\alpha(x) g_\alpha(x') P(x)P(x')\, dx\, dx' &= \frac{2}{P_\beta} \int_{X_\beta} g_\alpha(x) P(x)\, dx \int_{X_\beta} g_\alpha(x') P(x')\, dx' \\ &= \frac{2}{P_\beta} \left[\int_{X_\beta} g_\alpha(x) P(x)\, dx \right]^2. \end{aligned}$$

Combining these, the left hand side becomes

$$L = 2\int_{X_\beta} g_\alpha^2(x)P(x)\,dx - \frac{2}{P_\beta}\left[\int_{X_\beta} g_\alpha(x)P(x)\,dx\right]^2.$$

Finally, subtracting the left hand side from the right hand side, we obtain

$$\begin{aligned} R - L &= 2a^2P_\beta - 4a\int_{X_\beta} g_\alpha(x)P(x)\,dx + \frac{2}{P_\beta}\left[\int_{X_\beta} g_\alpha(x)P(x)\,dx\right]^2 \\ &= \frac{2}{P_\beta}\left(aP_\beta - \int_{X_\beta} g_\alpha(x)P(x)\,dx\right)^2. \end{aligned}$$

Thus we have the $2I(\alpha) - I_I(\alpha) \geq 0$, as required. □

We are using $I_I(\alpha)$ to measure the distance between a network function g_α and the invariant. Let's consider an alternative measure — the minimum distance between g_α and any function that strictly satisfies the invariant. First, we define the distance between an arbitrary function f and the function g_α by

$$\rho(f, g_\alpha) = \int_X \big(f(x) - g_\alpha(x)\big)^2 P(x)\,dx.$$

This has the same form as our functional $I(\alpha)$. Then the alternate distance between g_α and the invariant becomes

$$\rho_I(g_\alpha) = \inf\{\, \rho(f, g_\alpha) \mid f \text{ strictly satisfies the invariant } \}.$$

Each candidate function f strictly satisfies the invariant, and thus we may use the theorem 5.1 to get

$$I_I(\alpha) \leq 2\rho(f, g_\alpha).$$

From the proof, we find that we have strict equality if the constant value a of the function within each invariant class is

$$a = \frac{1}{P_\beta}\int_{X_\beta} g_\alpha(x)P(x)\,dx$$

that is, if a equals the mean value of g_α over X_β. This defines the closest function that strictly satisfies the invariant, and hence we have

$$\rho_I(g_\alpha) = \frac{1}{2} I_I(\alpha).$$

We want to find a network that approximate the unknown function f. We have seen that if $I(\alpha) < \epsilon$, then $I_I(\alpha) < 2\epsilon$. We can use this to restrict the set of networks that we need to search. Let

$$G_{2\epsilon} = \{\, g_\alpha \in G \mid I_I(\alpha) < 2\epsilon \,\}.$$

Then for any g_α such that $I(\alpha) < \epsilon$, we have $g_\alpha \in G_{2\epsilon}$.

The set $G_{2\epsilon}$ is defined in terms of $I_I(\alpha)$ which we cannot compute directly. We can compute $E_I(\alpha)$, and, subject to having enough examples, we know that these two quantities can differ by at most δ, in probability. Thus we define

$$G^*_{2\epsilon+\delta} = \{\, g_\alpha \in G \mid E_I(\alpha) < 2\epsilon + \delta \,\},$$

and for any $g_\alpha \in G_{2\epsilon}$, we have $g_\alpha \in G^*_{2\epsilon+\delta}$, in probability. While we could assert that for any function g_α with $I(\alpha) < \epsilon$, we had $g_\alpha \in G_{2\epsilon}$, we can only assert $g_\alpha \in G^*_{2\epsilon+\delta}$, in probability.

Note that we need to pick examples of the invariant according to the probability distribution $P(x, x')$. The results for uniform convergence of $E_I(\alpha)$ to $I_I(\alpha)$ make this assumption. It can be relaxed, under some circumstances, though there is a penalty involved in the number of required examples. Haussler and other investigate this in [8]. Further, the ability to restrict our set of networks relies on the observation that $I(\alpha) < \epsilon$ implies that $I_I(\alpha) < 2\epsilon$. The proof of this implication assumed that the probability distribution $P(x, x')$ was derived from the distribution $P(x)$ on the input set X.

Suppose, then, that we limit our search for a function g_α close to f to those functions in $G^*_{2\epsilon+\delta}$. The number of examples of the function will depend now on the growth function for $G^*_{2\epsilon+\delta} \subseteq \{g_\alpha\}$, which can be no larger than the growth function for all functions $\{g_\alpha\}$, and, possibly, much smaller. We can use gradient descent to reduce the empirical functional $E_I(\alpha)$, over enough examples of the invariant to ensure that $E_I(\alpha)$ is close to $I_I(\alpha)$.

Let's look at the gradient descent algorithm for $E_I(\alpha)$. We will simplify things somewhat. We will assume we have two equivalent inputs, $\mathbf{x}_1$ and $\mathbf{x}_2$, and the outputs of the network for these inputs are y_1 and y_2. The error function will simply be $E = \frac{1}{2}(y_1 - y_2)^2$. To return to the actual error $E_I(\alpha)$ we will simply have to reintroduce the summation.

In general, the change in the weight Δw is proportional to $-\partial E/\partial w$. We will compute this partial derivative. First, let's suppose we have a single neuron, with a transfer function $\sigma\colon \mathbf{R} \to \mathbf{R}$. The input θ to the function σ is the inner product of the inputs $\mathbf{x}$ and the weights $\mathbf{w}$.

Let $\theta_1 = \mathbf{w} \cdot \mathbf{x}_1$ and $\theta_2 = \mathbf{w} \cdot \mathbf{x}_2$. Then $y_1 = \sigma(\theta_1)$ and $y_2 = \sigma(\theta_2)$. We get

$$\begin{aligned}
-\frac{\partial E}{\partial w_i} &= -\frac{1}{2}\frac{\partial}{\partial w_i}(y_1 - y_2)^2 \\
&= -(y_1 - y_2)\left(\frac{\partial y_1}{\partial w_i} - \frac{\partial y_2}{\partial w_i}\right) \\
&= -(y_1 - y_2)\left(\sigma'(\theta_1)\frac{\partial \theta_1}{\partial w_i} - \sigma'(\theta_2)\frac{\partial \theta_2}{\partial w_i}\right) \\
&= -(y_1 - y_2)\sigma'(\theta_1)x_{1i} - (y_2 - y_1)\sigma'(\theta_2)x_{2i}.
\end{aligned}$$

Let $\delta_1 = \sigma'(\theta_1)(y_2 - y_1)$ and $\delta_2 = \sigma'(\theta_2)(y_1 - y_2)$. Then $\Delta w_i \propto \delta_1 x_{1i} + \delta_2 x_{2i}$.

We now generalize this to an interior layer of a multilayer network. The previous derivation will apply to the output neuron, if we understand the vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ to be the inputs to the final layer, and not inputs to the network. The following figure will provide the framework for generalizing to an interior neuron. We will consider a single neuron from each of 3 adjacent layers, in order to avoid unnecessarily complex notation. We use u to indicate the output of a neuron. Note that layer k may be the output layer, and layer i may actually be the inputs, in which case u_i would refer to those inputs.

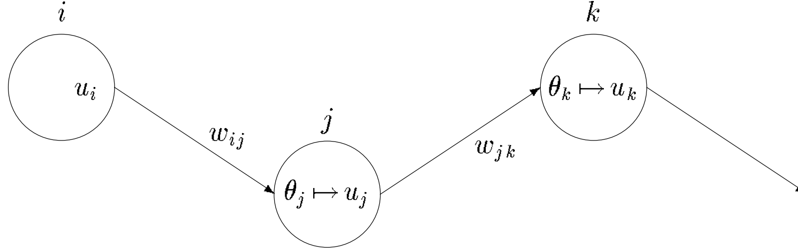

FIGURE 5.1

We will compute the change Δw_{ij}, by

$$\begin{aligned}
-\frac{\partial E}{\partial w_{ij}} &= -(y_1 - y_2)\left(\frac{\partial y_1}{\partial w_{ij}} - \frac{\partial y_2}{\partial w_{ij}}\right) \\
&= -(y_1 - y_2)\left(\frac{\partial y_1}{\partial u_{1j}}\frac{\partial u_{1j}}{\partial \theta_{1j}}\frac{\partial \theta_{1j}}{\partial w_{ij}} - \frac{\partial y_2}{\partial u_{2j}}\frac{\partial u_{2j}}{\partial \theta_{2j}}\frac{\partial \theta_{2j}}{\partial w_{ij}}\right) \\
&= -(y_1 - y_2)\left(\frac{\partial y_1}{\partial u_{1j}}\sigma'(\theta_{1j})u_{1j} - \frac{\partial y_2}{\partial u_{2j}}\sigma'(\theta_{2j})u_{2j}\right).
\end{aligned}$$

Let $\delta_{1j} = (y_2 - y_1)\partial y_1/\partial\theta_{1j}$ and $\delta_{2j} = (y_1 - y_2)\partial y_2/\partial\theta_{2j}$. Then we have

$$\begin{aligned}
\delta_{1j} &= (y_2 - y_1)\frac{\partial y_1}{\partial u_{1j}}\frac{\partial u_{1j}}{\partial \theta_{1j}} \\
&= (y_2 - y_1)\sigma'(\theta_{1j})\sum_k \frac{\partial y_1}{\partial \theta_{1k}}\frac{\partial \theta_{1k}}{\partial u_{1j}} \\
&= \sigma'(\theta_{1j})\sum_k \delta_{1k} w_{jk}.
\end{aligned}$$

Similarly we have $\delta_{2j} = \sigma'(\theta_{2j})\sum_k \delta_{2k} w_{jk}$. Thus we get $\Delta w_{ij} \propto \delta_{1j}u_{1j} + \delta_{2j}u_{2j}$. This turns out to be equivalent to running the standard back propagation algorithm twice, once assuming y_2 is the "correct" output and adjusting y_1 to match, followed by the reverse. The only difference is that the changes to the weights are summed, and applied after both passes through the network.

We have established a means, therefore, of minimizing the functional $E_I(\alpha)$. If we use enough examples, and we get $E_I(\alpha) < 2\epsilon + \delta$, then we can be confident

that our network is in the set $G_{2\epsilon}$. The examples of the function f have only to select a candidate from this reduced set. The number of examples required to do this will depend on the VC dimension of the set $G_{2\epsilon}$.

The set $G_{2\epsilon}$ contains those functions that nearly satisfy the invariant. The set G_0 is the set of all functions that exactly satisfy the invariant. We know that $G_0 \subseteq G_{2\epsilon} \subseteq G$, and hence we have $\mathrm{VCdim}(G_0) \leq \mathrm{VCdim}(G_{2\epsilon}) \leq \mathrm{VCdim}(G)$.

First let's consider G_0. Note that the constant function necessarily satisfies exactly any invariant. For our networks, we can construct a constant function by setting the weights in the output neuron to 0. The network then computes the function $\sigma(\tau)$, and thus we can obtain the constant function $g_\alpha(x) = a$ by setting the threshold in the output neuron neuron to $\sigma^{-1}(a)$, provided a is in the range of σ. The constant functions have VC dimension 1, and hence we have $\mathrm{VCdim}(G_0) \geq 1$.

Now let's move to the set $G_{2\epsilon}$. Here we allow functions that nearly satisfy the invariant. Such functions include, for example, those that are nearly constant. We can generate a pair of nearly constant functions by making the threshold in the output neuron very large in magnitude. This will force the output of the neuron near the limiting values of the sigmoid, independent of the values of the remaining weights and thresholds in the network.

We have $0 \leq \big(g_\alpha(x) - g_\alpha(x')\big)^2 \leq c$, so once $\epsilon \geq \frac{1}{2}c$, we have $G_{2\epsilon} = G$. The exact behaviour of the VC dimension as we vary ϵ between 0 and $\frac{1}{2}c$ will depend on the invariant and the functions g_α.

We cannot actually proceed by first minimizing $E_I(\alpha)$, to restrict ourselves to networks within $G_{2\epsilon}$, followed by minimizing $E(\alpha)$, to select a network that is close to our examples and hence close to f. The function $E_I(\alpha)$ does not depend in any way on the function f. While we know that if a network computes a function g_α close to f, then g_α is necessarily close to the invariant, the reverse does not hold — a function that is close to the invariant need not be close to f. If we use the invariant alone, we obtain a network that closely satisfies the invariant, and no more. As we've seen, a nearly constant function is close to the invariant, and to get such a function, we need only increase the magnitude of the threshold in the output neuron.

We can't use the examples of the invariant alone, yet we do want to use them. The solution is to simultaneously minimize $E(\alpha)$ and $E_I(\alpha)$.

However, this seems somewhat suspect. In minimizing $E_I(\alpha)$, for a given pair (x, x'), we are moving $g_\alpha(x)$ toward $g_\alpha(x')$, and vice versa. While they may become closer to one another, they are not necessarily close to $f(x)$. If we happen to also have the example (x, y), then we bring $g_\alpha(x)$ close to y. Now, bringing $g_\alpha(x')$ close to $g_\alpha(x)$ should also bring it close to y. In this case why not be more direct and simply use the pair of examples (x, y) and (x', y)?

Our results with the VC dimension say nothing about how we select a function g_α. Since the results are uniform, they apply to any function we might pick. Once our minimization is complete, we need to verify, using sufficient examples of the

invariant, that $E_I(\alpha) < 2\epsilon + \delta$. If this is the case, then we can be confident that our network is in the set $G_{2\epsilon}$. Knowing this, the value we compute for $E(\alpha)$ will be close, in probability, to $I(\alpha)$, assuming we compute $E(\alpha)$ using enough examples of f as dictated by the VC dimension of $G_{2\epsilon}$. If we simultaneously minimize $E_I(\alpha)$ and $E(\alpha)$, using adequate numbers of examples, then we directly address the required conditions above.

What if we were to not use examples of the invariant, but use the invariant to generate additional examples of the function. Suppose we have the example (x, y), where $x \in X_\beta$ for some β. Then, for all $x' \in X_\beta$, we also include the examples (x', y).

Within this framework, we have to be careful in applying the uniform convergence results. In order to say that the estimate $E(\alpha)$ is close to $I(\alpha)$ we need to have a sufficiently large number of examples, chosen independently, according to the distribution $P(x)$. Our extended class of examples does not satisfy these requirements. The original set of examples is chosen as required, but the number of examples needed is dictated by $\mathrm{VCdim}(G)$, not $\mathrm{VCdim}(G_{2\epsilon})$, as before.

Instead, however, we can look at the problem from a different perspective. Instead of thinking of each of the original examples as picking a point $x \in X$, according to $P(x)$, we can think of it as choosing a partition X_β, according to its probability P_β. The example (x, y) then carries all of (x', y) where $x' \in X_\beta$. Given this, assume we can compute

$$g_\alpha(\beta) = \int_{X_\beta} \big(y - g_\alpha(x)\big)^2 P(x|X_\beta)\, dx.$$

For the time being, we will ignore the distribution $P(x|X_\beta)$. If the partition is finite in size, we can replace integral with a sum. If the partition is reasonably small, we can sum over all elements; otherwise we can use the VC framework to estimate the integral using a subset of X_β. Corresponding to $g_\alpha(\beta)$ we have functional

$$I_P(\alpha) = \sum_\beta g_\alpha(\beta) P_\beta$$

and its estimate

$$E_P(\alpha) = \frac{1}{n} \sum_{i=1}^{n} g_\alpha(\beta_i).$$

Note that $I_P(\alpha) = I(\alpha)$. If we have uniform convergence of $E_P(\alpha)$ to $I_P(\alpha)$, then minimizing $E_P(\alpha)$ will give us a network with g_α close to the unknown function f, in probability. The number of required partitions, and hence original examples, is governed by the VC dimension of the set of functions $g_\alpha(\beta)$.

The remaining detail is the distribution $P(x|X_\beta)$, which appears in the definition of $g_\alpha(\beta)$. If we wish to compute $g_\alpha(\beta)$, we need to know $P(x|X_\beta)$, and

if we wish to estimate $g_\alpha(\beta)$ based on a random subset of X_β, we need to pick elements of X_β according to $P(x|X_\beta)$.

This problem has actually already come up, though it wasn't mentioned explicitly. We assumed that we could generate examples of the invariant (x, x') at random. Now we have always assumed that we can get an input x based on $P(x)$, but we now have to add to it a point x' based on $P(x|X_\beta)$.

If we know that inputs are equivalent, we may know how likely they are, given the subset X_β. For example, equivalent inputs may be equally likely. Even if we don't know the conditional probability, we can substitute another, at the expense of a penalty. The implications of such a substitution are investigated by Haussler and others in [8].

3. Group Invariance and Feed-Forward Networks

For perceptrons, we have seen how invariance under a group can be ensured. We would like to see if a similar result can be obtained for feed-forward neural networks. We will start with a simple example.

Suppose our unknown function $f\colon X^n \to \mathbf{R}$ is even, that is, $f(x_1, x_2, \ldots, x_n) = f(-x_1, -x_2, \ldots, -x_n)$. The group of transformations corresponding to this invariant is simple: $T = \{e, t\}$ where $e(\mathbf{x}) = \mathbf{x}$ is the identity and $t(\mathbf{x}) = -\mathbf{x}$.

For a perceptron, we would require that the functions computed by the input layer be closed under T. Suppose we make the same requirement on a feed-forward network.

Each neuron in the input layer computes $\varphi(\mathbf{x}) = \sigma(\mathbf{w}\cdot\mathbf{x} + \tau)$. We require that for every such function $\varphi(\mathbf{x})$ we also have the function $\hat{\varphi}(\mathbf{x}) = \varphi \circ t(\mathbf{x}) = \varphi(-\mathbf{x})$.

Let $\varphi(\mathbf{x}) = \sigma(\mathbf{w}\cdot\mathbf{x} + \tau)$ and $\hat{\varphi}(\mathbf{x}) = \sigma(\hat{\mathbf{w}}\cdot\mathbf{x} + \hat{\tau})$ Thus we have

$$\sigma(\mathbf{w}\cdot(-\mathbf{x}) + \tau) = \sigma(\hat{\mathbf{w}}\cdot\mathbf{x} + \hat{\tau})$$

and, since σ is one-to-one

$$\mathbf{w}\cdot(-\mathbf{x}) + \tau = \hat{\mathbf{w}}\cdot\mathbf{x} + \hat{\tau}.$$

We can satisfy this requirement by $\hat{\mathbf{w}} = -\mathbf{w}$ and $\hat{\tau} = \tau$.

Again, from the perceptron model, we further require that the weights that multiply the outputs of $\varphi(\mathbf{x})$ and $\hat{\varphi}(\mathbf{x})$ be equal for each neuron in the second layer. Thus, in the first two layers of our networks we would have the structure shown in figure 5.2.

The outputs of the second layer are even. This follows directly from the structure we imposed, just as it did with the perceptron. Take any neuron in the second layer. It computes $\sigma\big(\tau + \sum w(\varphi)\varphi(\mathbf{x})\big)$, where τ is the neuron's threshold and $w(\varphi)$ are the weights that multiply the outputs $\varphi(\mathbf{x})$ of the first layer. Since $w(\phi) = w(\phi \circ t)$ we have,

$$\sum w(\varphi)\varphi(-\mathbf{x}) = \sum w(\varphi \circ t)\varphi \circ t(\mathbf{x}) = \sum w(\varphi)\varphi(\mathbf{x}).$$

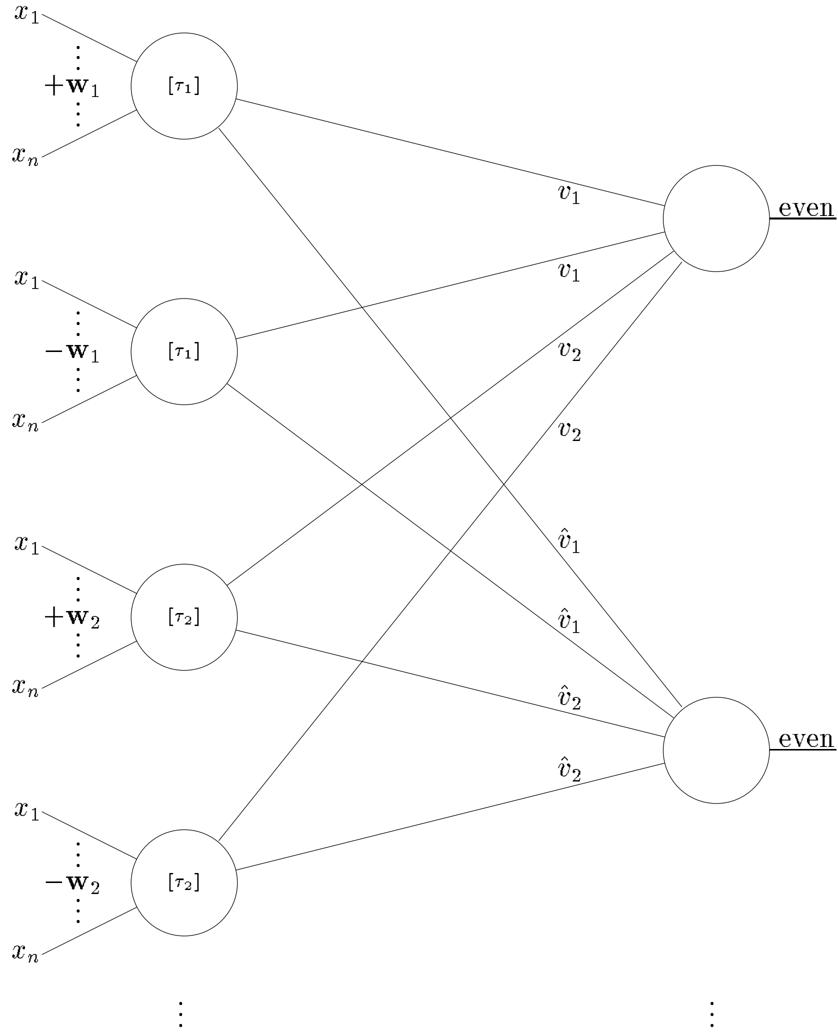

FIGURE 5.2

Thus

$$\sigma\big(\tau + \sum w(\varphi)\varphi(-\mathbf{x})\big) = \sigma\big(\tau + \sum w(\varphi)\varphi(\mathbf{x})\big)$$

and the output is even, as required. Further, from the second layer forward, the outputs of all neurons will be even, and thus the function computed by the network is also even.

This network is very similar to the network suggested by Abu-Mostafa in [1]. Instead of having $\hat{\varphi}(\mathbf{x}) = \varphi(-\mathbf{x})$, he uses $\hat{\varphi}(\mathbf{x}) = -\varphi(-\mathbf{x})$. This means that the weights multiplying the outputs $\varphi(x)$ and $\hat{\varphi}(x)$ are negations of one another, rather than identical as we have derived here.

In this network we have placed constraints on the weights in the first two layers,

and the thresholds in the input layer. These constraints reduce the number of free variables that determine each network. Baum and Haussler [2] have shown that the VC dimension of the networks can be bounded in terms of the number of weights and thresholds in the network. Our constraints effectively reduces this number, and thus tends to reduce the VC dimension of the networks. This then translates to a reduced number of examples needed for learning.

To take advantage of any reduced VC dimension due to the imposed structure, we need to modify the learning algorithm to maintain the structure. Our model for doing so will be based on our modified learning algorithm for perceptrons. We will start with an example, namely the even functioned discussed above. Let us first simplify the network. We will have 2 neurons in the first layer, and a single neuron in the second layer. The network is shown in figure 5.3.

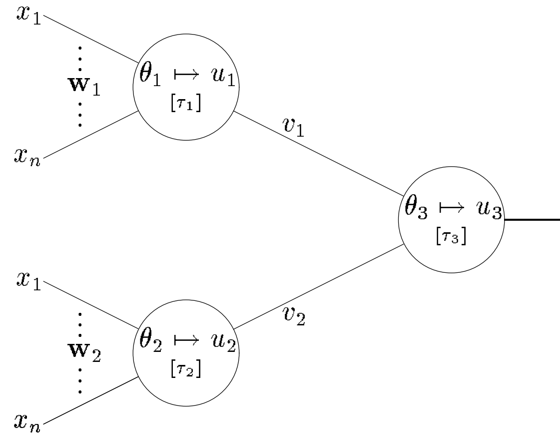

FIGURE 5.3

The equations that define the network are the following:

$$\begin{aligned} \theta_1 &= \mathbf{w}_1 \cdot \mathbf{x} + \tau_1 & \theta_2 &= \mathbf{w}_2 \cdot \mathbf{x} + \tau_2 & \theta_3 &= v_1 u_1 + v_2 u_2 + \tau_3 \\ u_1 &= \sigma(\theta_1) & u_2 &= \sigma(\theta_2) & u_3 &= \sigma(\theta_3). \end{aligned}$$

We will assume that the network has, initially, the structure we derived for even functions, as shown in figure 5.2. We will assume that the two input neurons are equivalent under T, and hence $\mathbf{w}_2 = -\mathbf{w}_1$, $\tau_1 = \tau_2$, and $v_1 = v_2$.

Suppose we have an example $(\mathbf{x}, y)$. The output of the network given $\mathbf{x}$ is u_3. We want u_3 to be close to y. We know that $(-\mathbf{x}, y)$ is another example, and because of the structure of our network, the output of the network given $-\mathbf{x}$ is also u_3.

The general back propagation algorithm, applied to our network, gives us the

following.

$$
\begin{array}{lll}
\delta_3 = \sigma'(\theta_3)(y - u_3) & \Delta v_1 \propto u_1 \delta_3 & \Delta \tau_3 \propto \delta_3 \\
 & \Delta v_2 \propto u_2 \delta_3 & \\
\delta_1 = \sigma'(\theta_1) v_1 \delta_3 & \Delta w_{1j} \propto x_j \delta_1 & \Delta \tau_1 \propto \delta_1 \\
\delta_2 = \sigma'(\theta_2) v_2 \delta_3 & \Delta w_{2j} \propto x_j \delta_2 & \Delta \tau_2 \propto \delta_2.
\end{array}
$$

We will apply this algorithm for the two examples $(\mathbf{x}, y)$ and $(-\mathbf{x}, y)$. First we will reduce the number of variables by taking advantage of what we require for this network.

Let $\mathbf{w}_1 = \mathbf{w}$, $\mathbf{w}_2 = -\mathbf{w}$, and $\tau_1 = \tau_2 = \tau$. The output for both inputs $\mathbf{x}$ and $-\mathbf{x}$ is $u_3 = u$. Also, as we observed earlier, θ_3 is equal for both inputs $\mathbf{x}$ and $-\mathbf{x}$, since, for $\mathbf{x}$ we have

$$\theta_3 = \sigma(\mathbf{x} \cdot \mathbf{w} + \tau) v + \sigma(\mathbf{x} \cdot (-\mathbf{w}) + \tau) v + \tau_3,$$

and for $-\mathbf{x}$ we have

$$\theta_3 = \sigma((-\mathbf{x}) \cdot \mathbf{w} + \tau) v + \sigma((-\mathbf{x}) \cdot (-\mathbf{w}) + \tau) v + \tau_3.$$

Hence let $\theta_3 = \theta$.

Thus, for the input $\mathbf{x}$, we get

$$
\begin{aligned}
\delta_3 &= \sigma'(\theta)(y - u) \\
\delta_1 &= \sigma'(\mathbf{x} \cdot \mathbf{w} + \tau) v \delta_3 \\
\delta_2 &= \sigma'(-\mathbf{x} \cdot \mathbf{w} + \tau) v \delta_3,
\end{aligned}
$$

and for the input $-\mathbf{x}$ we get

$$
\begin{aligned}
\delta_3 &= \sigma'(\theta)(y - u) \\
\delta_1 &= \sigma'(-\mathbf{x} \cdot \mathbf{w} + \tau) v \delta_3 \\
\delta_2 &= \sigma'(\mathbf{x} \cdot \mathbf{w} + \tau) v \delta_3.
\end{aligned}
$$

Let $\delta = \sigma'(\theta)(y - u)$, $a = \sigma'(\mathbf{x} \cdot \mathbf{w} + \tau)$, and $b = \sigma'(-\mathbf{x} \cdot \mathbf{w} + \tau)$.

Here we will update the weights by the sum of the individual updates, rather than the average, as we did with the perceptrons. Thus we get

$$
\begin{array}{ll}
\Delta v_1 \propto \big(\sigma(\mathbf{x} \cdot \mathbf{w} + \tau) + \sigma(-\mathbf{x} \cdot \mathbf{w} + \tau)\big)\delta & \Delta \tau_3 \propto \delta \\
\Delta v_2 \propto \big(\sigma(-\mathbf{x} \cdot \mathbf{w} + \tau) + \sigma(\mathbf{x} \cdot \mathbf{w} + \tau)\big)\delta & \\
\Delta w_{1j} \propto x_j (a - b) v \delta & \Delta \tau_1 \propto (a + b) v \delta \\
\Delta w_{2j} \propto x_j (b - a) v \delta & \Delta \tau_2 \propto (a + b) v \delta.
\end{array}
$$

Note that we have $\Delta v_1 = \Delta v_2$ and $\Delta \tau_1 = \Delta \tau_2$, so that after updating, we will still have $v_1 = v_2$ and $\tau_1 = \tau_2$. Also note that $\Delta w_{1j} = -\Delta w_{2j}$, so we also maintain the relationship $\mathbf{w}_1 = -\mathbf{w}_2$.

We chose here to use the sum of the updates, rather than the average as we did with the perceptrons. The difference between the two is a factor of $|T|^{-1}$, or, in this case $\frac{1}{2}$. However, we have yet to define the constant of proportionality. Thus the difference between the sum and the average of the updates can be, at worst, incorporated into the constant of proportionality.

In this derivation, we have assumed that we start with certain relationships between weights and thresholds. Given those relationships, we've shown that they are maintained throughout the learning process. If we were to start with no such requirements on the weights and thresholds, it is always possible that the network would eventually move to such a result. Any single step, however, does not necessarily do so.

Consider updating the weights v_1 and v_2. We will use the notation, for example, $u_1(\mathbf{x})$ to indicate the output of neuron 1 given the network input $\mathbf{x}$. We have

$$\Delta v_1 \propto u_1(\mathbf{x})\sigma'\big(\theta_3(\mathbf{x})\big)\big(y - u_3(\mathbf{x})\big) + u_1(-\mathbf{x})\sigma'\big(\theta_3(-\mathbf{x})\big)\big(y - u_3(-\mathbf{x})\big),$$
$$\Delta v_2 \propto u_2(\mathbf{x})\sigma'\big(\theta_3(\mathbf{x})\big)\big(y - u_3(\mathbf{x})\big) + u_2(-\mathbf{x})\sigma'\big(\theta_3(-\mathbf{x})\big)\big(y - u_3(-\mathbf{x})\big).$$

The weights v_1 and v_2 appear only as part of θ_3 and u_3. Thus it is simple to establish conditions under which the update will increase the distance between v_1 and v_2. Assume the range of the sigmoid is $[-1, 1]$. Note that the derivative of σ is strictly positive. Assume further that the three thresholds are large in magnitude, making $u_1(\mathbf{x}) \approx 1$, $u_2(\mathbf{x}) \approx -1$, and $u_3(\mathbf{x}) \approx -1$. Also assume that $v_1 > 0$ and $v_2 < 0$. Assuming finally that $y > 0$, we have $\Delta v_1 > 0$ and $\Delta v_2 < 0$.

We want now to generalize this result to arbitrary networks. Suppose the input neurons are closed under T, that is, for each neuron φ, and each $t \in T$, there is also some neuron that computes $\varphi \circ t$. The set T divides the input neurons into equivalence classes. We will assume that the weights from all neurons within an equivalence class to any given neuron in the next layer are equal to one another. Thus the structure is as shown in figure 5.4.

Suppose we have a given example (x, y). The change in, say, one of the weights v_1, is proportional to the output of the neuron $\varphi(x)$ in the first layer times a constant δ. The constant δ depends on $\sigma'(\theta)$, and the value of constants in layers beyond the second layer. The value of θ is invariant under T, as we have already seen. Substituting $t(x)$ for the input x simply gives us the same output from the first layer but in a different order, within equivalence classes. Since the weights multiplying these outputs are equal with equivalence classes, the inner product and hence θ is invariant under T.

We accumulate all the changes for a given weight, over the equivalent inputs $t(x)$ for all $t \in T$. Thus the change in the weight on the output of $\varphi(x)$ is v_1 is $\delta \sum_{t \in T} \varphi(t(x))$.

Consider any other equivalent input neuron to $\varphi(x)$. We can use $\varphi \circ s(x)$ to indicate this neuron, for some $s \in T$. Then the change for the weight on this neuron's output is $\delta \sum_{t \in T} \varphi \circ s(t(x)) = \delta \sum_{t \in T} \varphi(t(x))$.

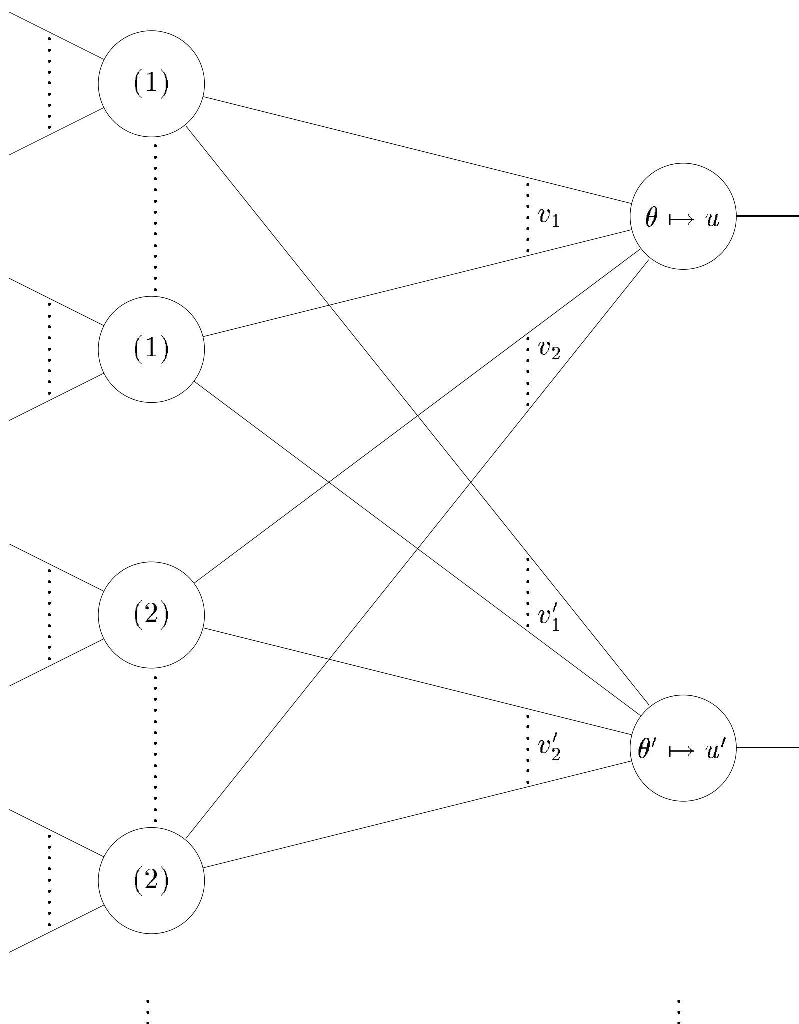

FIGURE 5.4

Note that each function $\varphi(t(x))$ is in the equivalence class containing $\varphi(x)$. Thus summing $\varphi(t(x))$ over all $t \in T$ is equivalent to summing the output of each member of the equivalence class, modulo an integral constant multiplier. Thus the update for the weights can be viewed in terms of the average update over the equivalence class, for a single input x.

These changes to the weights multiplying the outputs of the neurons from a single equivalence class are equal to one another, and so the corresponding weights remain equal to one another after updating. In showing this, we have only assumed that the outputs of the input layer were closed under T. Thus it is not necessary that the input layer be made up of single neurons. Each input function $\varphi(x)$ could represent a multiple layer neural network, so long as the

closure condition is met.

In the case of even functions, we were quite specific about the structure of the neurons in the first layer, and were able to show that the structure was preserved. Here we have no explicit structure, and indeed we should not even assume that the input functions $\varphi(x)$ are simply single neurons. Thus we can not explore this algorithm further, in the general case.

CHAPTER 6

Simulations

In this chapter we will explore learning a simple function on a feed-forward network. We will use the back propagation algorithm, exploring some of the variations discussed earlier in the thesis.

Our simple function $f\colon X^n \to \mathbf{R}$ is defined by

$$f(\mathbf{x}) = \begin{cases} +1 & \text{if } |\sum_{i=1}^{n} x_i| < \frac{1}{2}\sqrt{n/6}, \\ -1 & \text{if } |\sum_{i=1}^{n} x_i| > \frac{3}{2}\sqrt{n/6}, \\ \text{unspecified} & \text{otherwise.} \end{cases}$$

The network we will use is similarly simple. It is the same network we used when we considered even functions, and is shown in figure 5.3, on page 49. The network has three neurons, two in the input layer, numbered 1 and 2, and one in the output layer, numbered 3.

The network is capable of computing this function. We may rewrite f as

$$f(\mathbf{x}) = 1 \quad \text{iff} \quad \left(\sum_{i=1}^{n} x_i < \sqrt{n/6} \right) \wedge \left(\sum_{i=1}^{n} x_i > -\sqrt{n/6} \right) .$$

The "and" function $\wedge$ can be computed by the output neuron 3. The two terms are simple threshold functions, which can be computed by the two input neurons 1 and 2. In this case we have $\mathbf{w}_1 = (1, \ldots, 1)$, $\mathbf{w}_2 = (-1, \ldots, -1)$, $\tau_1 = \tau_2 = \sqrt{n/6}$, $v_1 = v_2 = 1$, and $\tau_3 = 1$. It may be necessary to adjust the gain of the neurons; this can be done by scaling the above weights and thresholds.

Let us first verify that the network behaves as expected under back propagation. We have a pool of 5000 examples, of which we choose, at random, a subset. The performance of the network on all 5000 examples is our estimate of $I(\alpha)$; the performance on the chosen subset is of course $E(\alpha)$. For $n = 6$, and 50 examples, we plot these two functions against the number of updates to the network. The result is shown in figure 6.1. We see here that the performance on the examples

is much better than the overall performance, so our network has not generalized well.

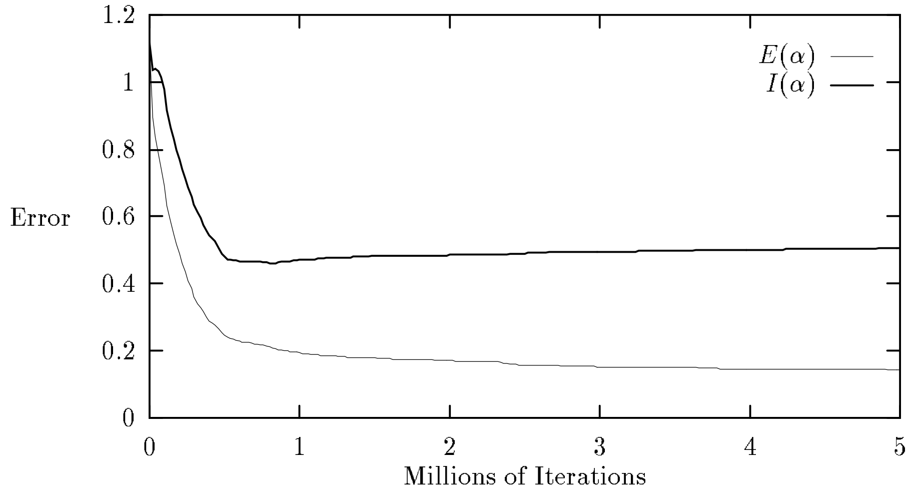

FIGURE 6.1. 6 inputs and 50 examples

For any particular choice of 50 examples, it might be that the network learned the function or it might have failed outright. The graph in figure 6.1 does not represent a single simulation with one choice of 50 examples, but is the average over 25 such simulations, each based on a different choice of examples. In figure 6.2, we plot the final values, after 5 million updates, of $E(\alpha)$ and $I(\alpha)$, for each of the 25 individual runs. Here we see that for 15, if not 17, of the 25 individual runs, that the network did generalize.

We define an update, or iteration, as a single forward/backward pass through the network. Such a pass is made for each of the examples, and the changes to the weights are not made, but rather accumulated on the side. After a pass has been made for all the examples, the weights are then updated. With, for example, 50 examples, a simulation will make 5,000,000 forward and backward passes through the network, but will only actually update the weights 100,000 times.

The behaviour of our network depends on the VC dimension of the set of all possible networks. To estimate the VC dimension, we again appeal to the results of Baum and Haussler [2]. There are a $2n + 2$ weights and 3 thresholds, for a total of $2n+5$ parameters. All of these are not independent; nevertheless, we will use $2n + 5$ as an estimate of the VC dimension. What we see from our estimate is that the VC dimension increases as we increase n, the number of inputs to the network.

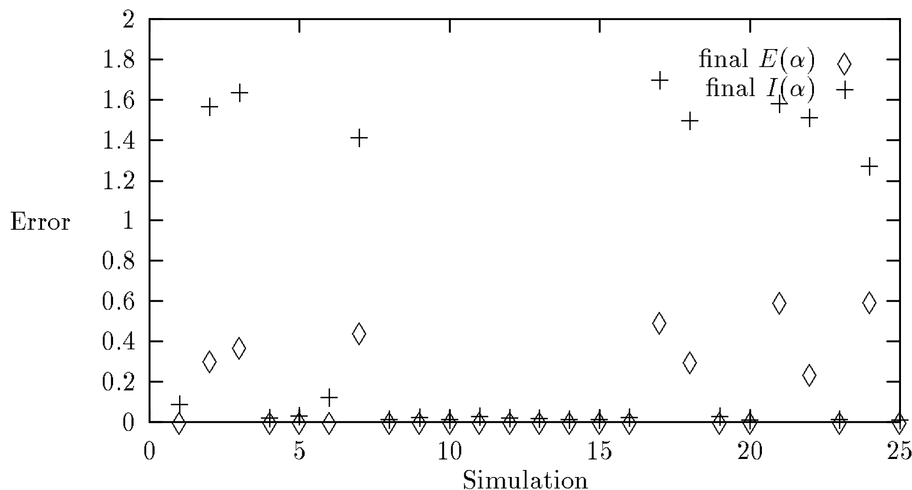

FIGURE 6.2. Final performance for each run with 6 inputs and 50 examples

Our concern is generalization, the difference between $E(\alpha)$ and $I(\alpha)$. For a network of fixed VC dimension, we expect that if we increase the number of examples, the difference will decrease, and vice versa. The following graph, figure 6.3, shows that this is the case. Here we plot $\left|I(\alpha) - E(\alpha)\right|$, based on the average values over 25 individual simulations, with the number of inputs fixed at 6, fixing the VC dimension. Thus the line corresponding to 50 inputs is the difference of the two lines in figure 6.1. We will refer to this difference as the generalization error.

We also expect that more of 25 the individual simulations will result in generalization as we increase the number of examples. In figure 6.4, we show the final value for the generalization error, for each of the 25 simulations that make up the average values plotted in figure 6.3. Here the errors are sorted by decreasing size, to make the graph more clear.

Now, instead of varying the number of examples while holding the VC dimension constant, we do the reverse. Here we fix the number of examples at 25, and vary the VC dimension by varying the number of inputs. As the number of inputs increase, we expect to see a reduced ability of the network to generalize. Figure 6.5 shows exactly this behaviour.

We have now established the framework for our simulations. The next step is to add examples of the invariant. We will refer to examples of the function f simply as examples, and use hints to refer to examples of the invariant. The function f is invariant under any permutation of the n inputs. It is also invariant under any transformation that is a subgroup of the permutations. We will consider two groups, the group of all permutations, and the group of cyclic shifts. The former

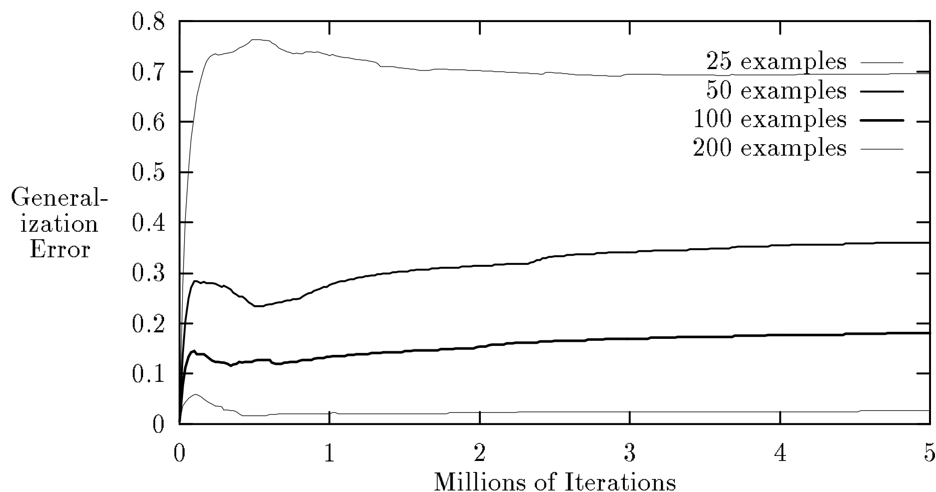

FIGURE 6.3. Average generalization error, using 6 inputs

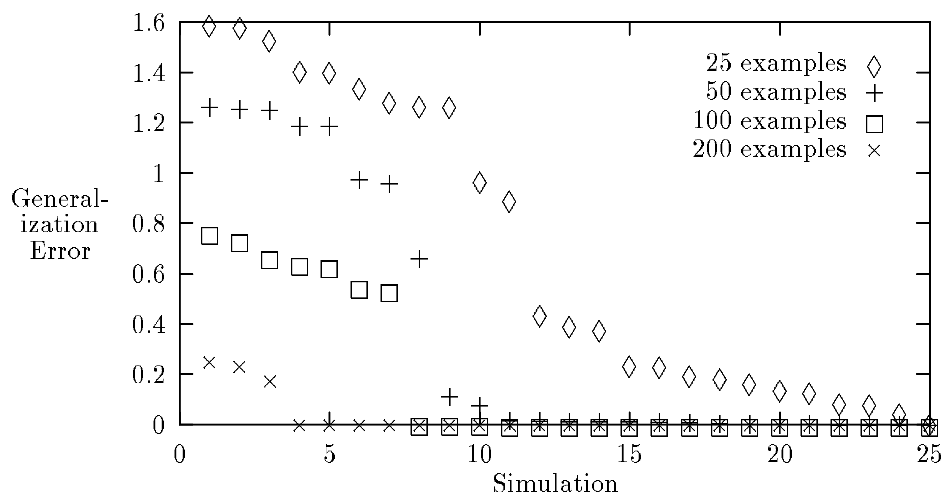

FIGURE 6.4. Final generalization error, using 6 inputs

has $n!$ elements while the latter has only n. The former carries more information with it; we expect to see this reflected in the relative performance of these two types of hint.

A hint is chosen at random. First an input $\mathbf{x}$ is picked at random, uniformly in $[-1,1]^n$. Then an element of the group is picked, again uniformly, and applied

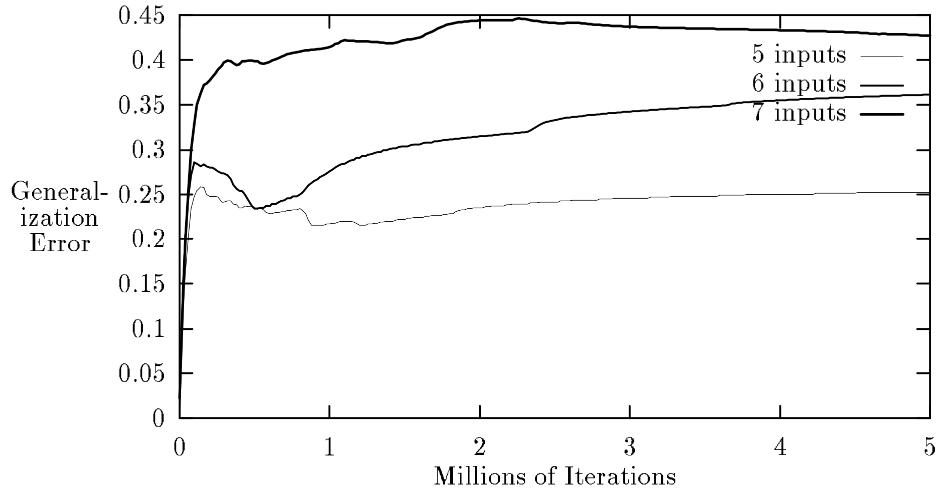

FIGURE 6.5. Generalization error, using 50 examples

to $\mathbf{x}$ producing $\mathbf{x}'$. The pair $(\mathbf{x}, \mathbf{x}')$ is our hint. Note that the identity element of the group may be picked, in which case we have a degenerate hint.

Let us now attempt to estimate the VC dimension of the set G_0, those networks that strictly satisfy the invariant. For this we will use the permutation group. Consider the principles of group invariance, which have us split the input neurons into equivalence classes. The obvious split is into two classes with a single neuron each. In this case, we must have each neuron invariant under permutations of the inputs, and hence the weights multiplying those inputs must be the same. Thus the neurons no longer have n independent weights, but a single value that is used for all weights. Thus the VC dimension of these networks is $4 + 3 = 7$, and we will use this to estimate the VC dimension of G_0. Note that we are not suggesting that these are all possible networks in G_0, nor are we suggesting that we should enforce any restriction on the weights in the network.

The set $G_{2\epsilon}$, those networks that are close to the invariant, will have a VC dimension between G_0 and G. It is the hints that define the particular subset $G_{2\epsilon}$. As we increase the number of hints, we decrease ϵ, and thus we move the VC dimension of $G_{2\epsilon}$ closer to that of G_0. As a result, we need fewer examples to select a network from $G_{2\epsilon}$.

We are able to vary the relative number of examples and hints. Hints are treated in a similar way to the examples. The simulation will choose a number of random hints and make a forward and backward pass over the network with each. The changes to the weights are again accumulated, and applied at the end. The number of hints used in making such an update is equal to the number of examples. The simulation might alternate an update based on examples followed

by one based on hints, thus giving equal number of each. Or it might alternate a single update based on examples followed by, say, 4 updates based on hints, making 80% of the updates based on hints.

We expect that as we increase the relative proportion of hints, we will decrease the generalization error. From figure 6.3, we see that with 6 inputs and 25 examples, we do fail to generalize. To these 25 examples, then, we add hints, so that of the 5,000,000 iterations, 20%, 50%, and 80% are hints. The generalization errors are plotted in figure 6.6. Here we see that increasing the relative proportion of hints does reduce the generalization error. Also, a comparison with figure 6.3 shows that using 25 examples with 50% hints, is roughly equivalent to using 100 examples alone. When we increase proportion of hints to 80%, we do better, on average, than 100 examples alone.

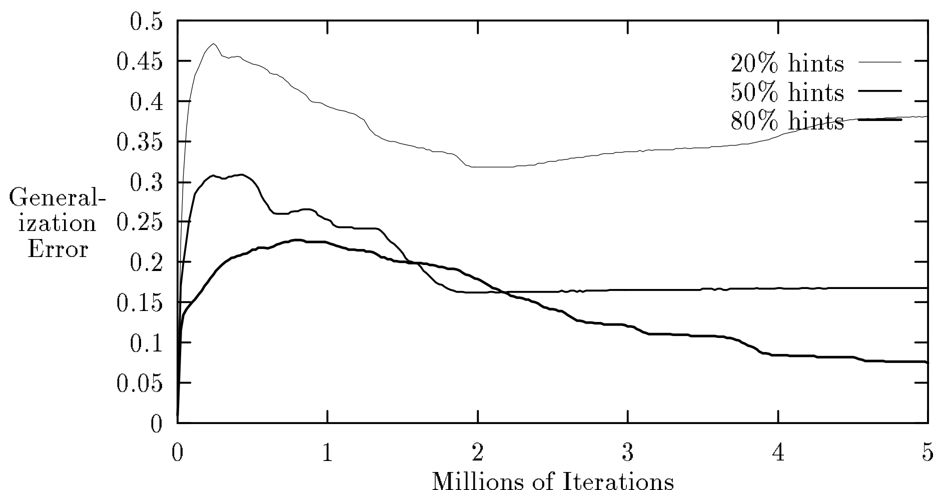

FIGURE 6.6. Generalization error, using 25 examples, and 6 inputs, with examples of the invariant

From the perspective of $G_{2\epsilon}$, adding hints always helps us. However, we have a fixed total number of iterations, and we need to reserve a sufficient number of them for examples, otherwise the network won't be able to select a network close to f. In the limit, of 100% hints, we have no examples and thus we can't expect to learn f. Nevertheless, even in this limiting case, the generalization error will still tend to zero. What happens, though, is that the performance on the examples tends to be uniformly bad, as it is overall. This is shown clearly in figure 6.7, where we plot $E(\alpha)$ in three cases where the proportion of hints is increasingly large.

We stated that our function f is invariant under any permutation of the inputs. A cyclic shift is a subset of these permutations. If we use only this subgroup, we

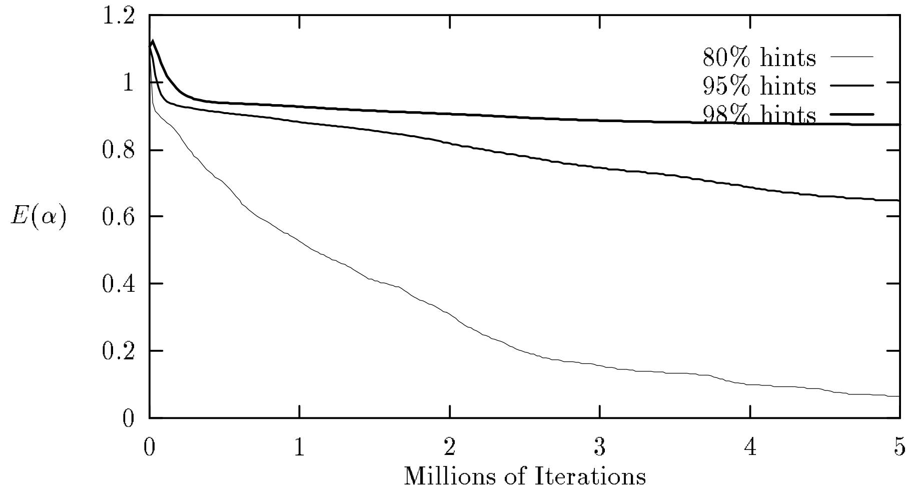

FIGURE 6.7. $E(\alpha)$, using 25 examples and 6 inputs, with a large proportion of hints

would not expect to do as well as we would taking advantage of the entire group. It turns out that for our example function, the performance is almost identical, though the entire group does appear to have a very small edge. In figure 6.8, we compare the two groups, using 6 inputs, 25 examples, and 20% of the iterations going to hints. These two curves are almost identical.

When we combine hints with our examples, the examples are only responsible for selecting a network from the restricted class of networks that nearly satisfy the invariant. We have seen already that the solution is found within a class of networks that explicitly satisfy the invariant, and whose VC dimension is estimated at 7, which is independent of n. Thus we might expect that there will be less variation in the generalization error as we vary n when using hints. In figure 6.9, we plot the results of using 25 examples, with 80% of the iterations hints, for 5, 6 and 7 inputs. There does appear to be a narrowing of these curves in contrast to figure 6.5. We don't see, in this case, that 5 is best and 7 worst, however.

Now let's turn to the alternative method suggested in the previous chapter. Rather than generating a hint, we transform the example. Thus, given (x, y), we pick a random transformation t and use $\big(t(x), y\big)$. The estimate $E_P(\alpha)$ suggests that we should take an example, and apply all possible transformations to it. We might choose to update the weights after each example, or wait and update after using all the examples. The results of these two choices are plotted in figure 6.10. We find that the curves are very jagged. There are $6! = 720$ permutations of the 6 inputs, and the 720 corresponding examples $(t(x), y)$ are likely to be highly

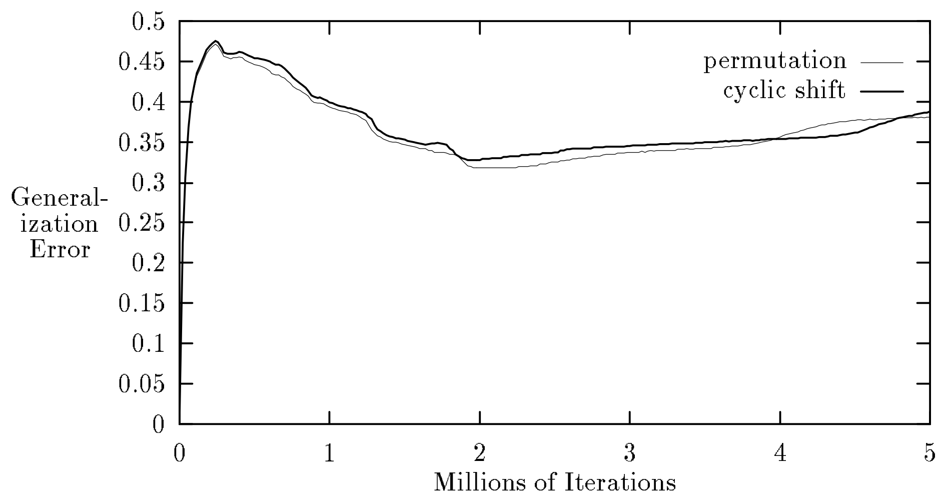

FIGURE 6.8. Generalization error, using 25 examples, and 6 inputs, comparing the underlying group

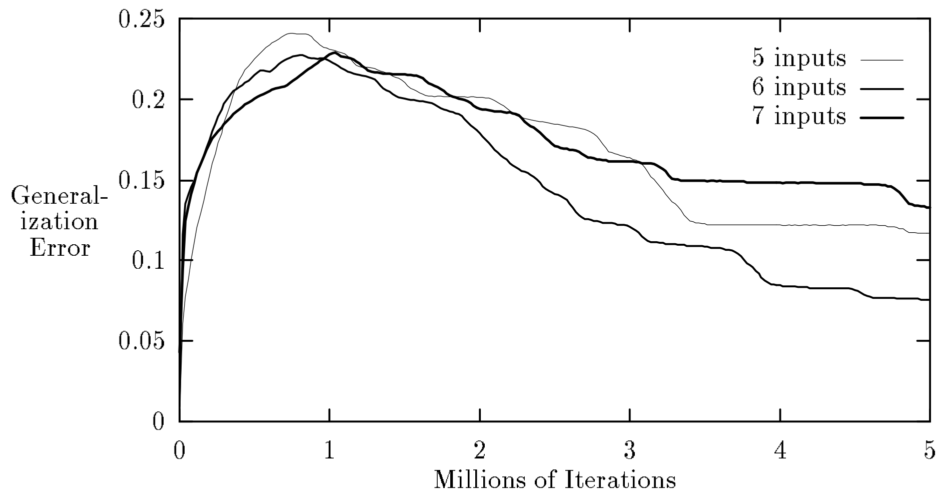

FIGURE 6.9. Generalization error, using 25 examples, and 80% hints

correlated. Such correlations hurt the gradient descent algorithm.

We get much better results instead by either picking a random example and applying random permutation, and updating the network based on this single iteration, or updating the network based on a single random permutation of all

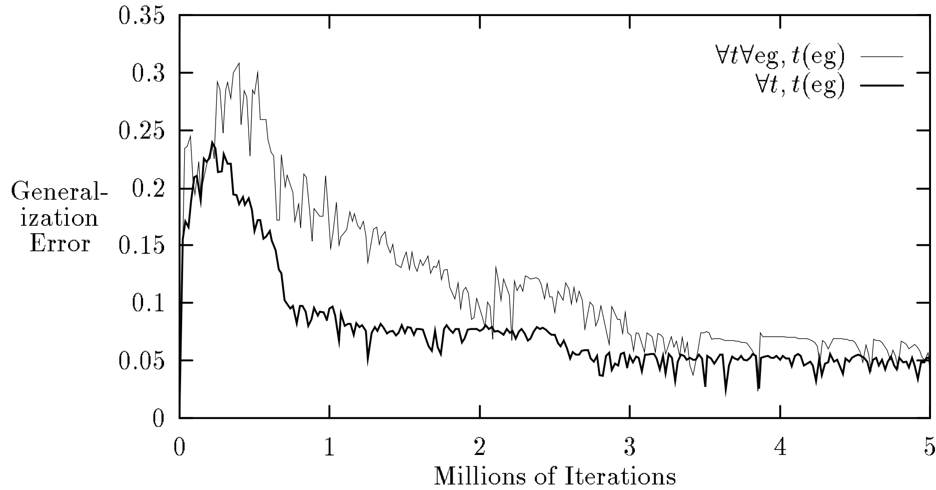

FIGURE 6.10. Generalization error, using all permutations on each example

examples. These curves are plotted in figure 6.11. Note that 25 examples, with 720 possible permutations, gives us an effective pool of some 18,000 examples. If we compare figures 6.11 and 6.3, we see that we generalize slightly better with 200 examples chosen independently.

The permutation group has $n!$ elements, while the cyclic subgroup has only n. Thus we wouldn't expect the same jagged lines with the smaller group. The corresponding curves from figures 6.10 and 6.11, but based on the cyclic subgroup, are plotted in figure 6.12, and are almost indistinguishable, in fact. Comparing these three figures, we see that the permuted examples do better, assuming we update based on a random example and random transformation.

Since we are incorporating the invariant, we expect the variation in the generalization error due to an increasing number of inputs to be less than it was with examples alone. In figure 6.13, we compare 5, 6 and 7 inputs, all using 25 examples, that are permuted randomly.

If we compare figures 6.9 and 6.13, we see that permuting the examples does better than using hints, at least at the 80% level. This remains true if we use cycled examples, rather than permuted ones. These results are suggestive, but are based on a single, particular example. The relative performance does suggest that a comparative study is well worth while.

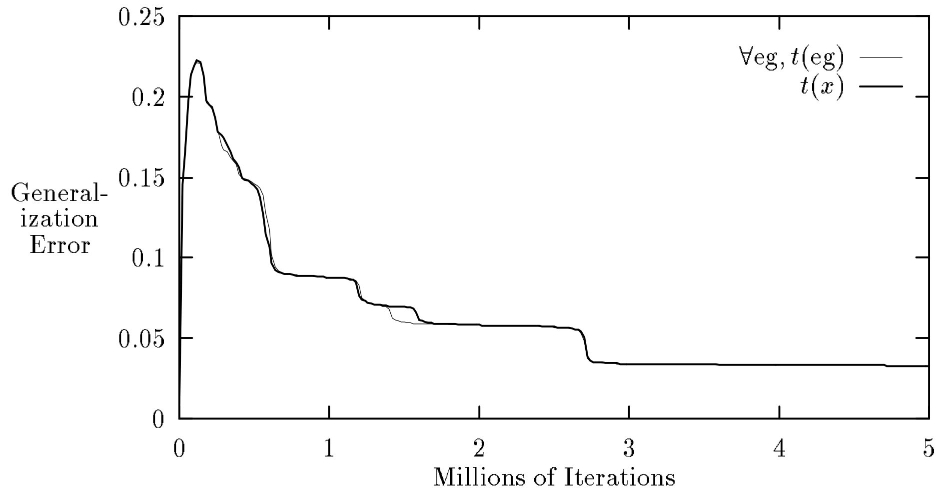

FIGURE 6.11. Generalization error, using a single random permutations on the examples

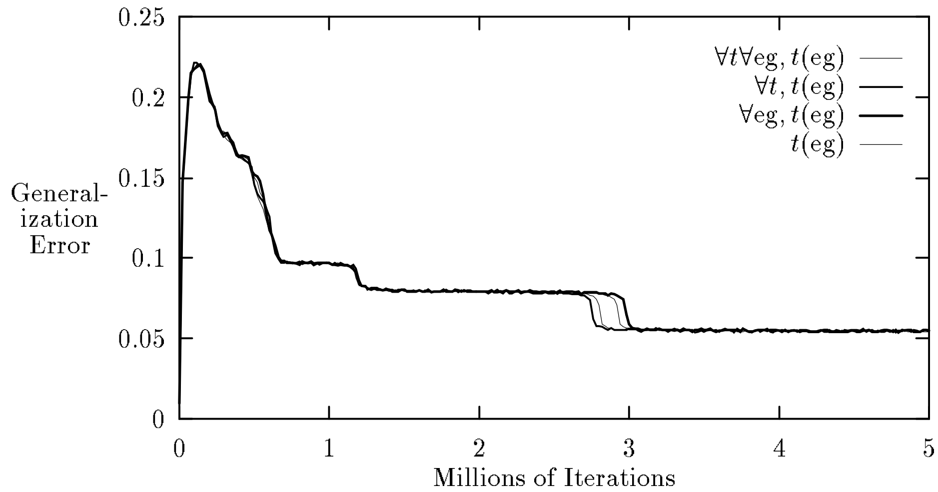

FIGURE 6.12. Generalization error, varying how we update based on the cycled examples

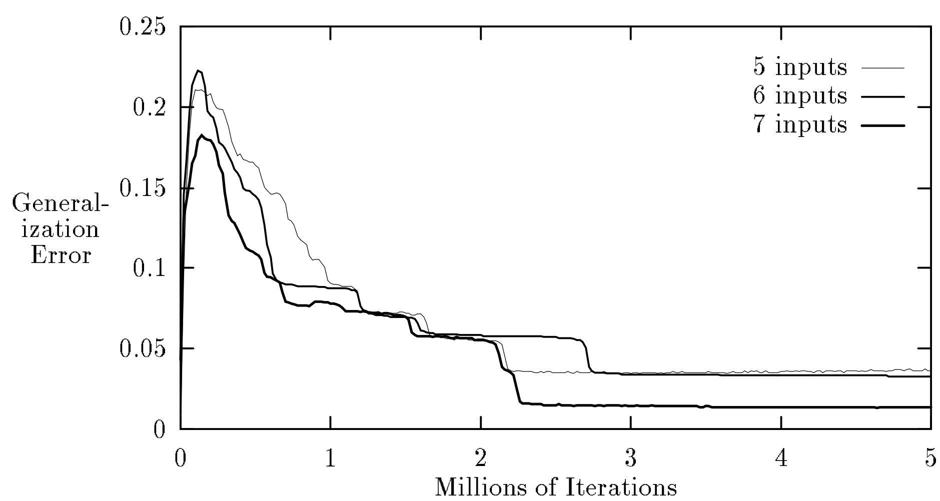

FIGURE 6.13. Generalization error, varying number of inputs

CHAPTER 7

Conclusions

In this thesis we have looked at invariants, inputs that are equivalent with respect to the unknown function f that we wish to learn. We have shown two ways we can take advantage of this extra information about f.

First we assumed that there was a group of transformations that defines the equivalent inputs. Armed with this group, it is possible to build a network that is guaranteed to satisfy the invariant. Further, the standard learning algorithms can be modified to preserve the structure imposed on the network.

More generally, we defined an invariant as a partition over the input space, that groups together inputs for which we know f gives the same output. We can use the partition to choose examples of the invariant, and use these, along with examples of the unknown function f, to learn f. In the case of back propagation, an example of the invariant turns out to be not unlike two mock examples of the function. The number of examples of the invariant that we must use is determined by the VC dimension of the functions $h_\alpha(x, x') = \big(g_\alpha(x) - g_\alpha(x')\big)^2$, which can be several times larger than the VC dimension of the functions $g_\alpha(x)$. We assume, however, that we are able to generate examples of the invariant at will, so a large required number is not a concern. Using the examples of the invariant does allow us to effectively reduce the set of functions $\{g_\alpha\}$ to only those that nearly satisfy the invariant, and it is now the VC dimension of this reduced set that determines the number of examples of f that are needed. Simulations of a simple network learning an equally simple function confirmed properties we would expect given the theory developed here.

1. Further Study

A number of topics for further study were raised. In computing the VC dimension of $\{h_\alpha\}$ we assumed the worst case. What happens in the average case? First, we largely ignored the partition that defined the invariant. Given a set of pairs that are used to define the VC dimension, we found it was likely possible to construct a variety of partitions around them, certainly a coarse division of X

but also probably some finer divisions. Thus, except in the limit, we found that a fine division of X could give rise to as large a VC dimension for X as a coarse division, even though a coarse division represents more information about f. On average, however, this may not be the case. Suppose the partition is given, and fixed. If we were to compute an expected growth function for $\{h_\alpha\}$, we might gain something from the probability distribution just as we would for $\{g_\alpha\}$. But we have additional requirements on our choice of pairs (x, x') if we want to avoid redundancies. These redundancies might be substantial in the average case.

The VC dimension of the subset $G_{2\epsilon}$ was not investigated. This is a difficult problem; it's not clear what networks are contained in the subset $G_{2\epsilon}$, or even in the smaller G_0, which contains only those functions that strictly satisfy the invariant. The structures defined here based upon a group of transformations might be used to provide a lower bound.

The use of transformed examples, as compared to using examples of the invariant needs further study. The error estimate $E_P(\alpha)$ suggests gradient descent within partitions X_β, while the simulations find it better to cut across the partitions. This is not a contradiction, since the uniform convergence results do not say anything about how we select a network.

Bibliography

[1] Yaser S. Abu-Mostafa, *Learning from hints in neural networks*, Journal of Complexity **6** (1990), no. 2, 192–198.

[2] Eric B. Baum and David Haussler, *What size net gives valid generalization?*, Neural Computation **1** (1989), no. 1, 151–160.

[3] H. D. Block and S. A. Levin, *On the boundedness of an iterative procedure for solving a system of linear inequalities*, Proceedings of the American Mathematical Society **26** (1970), no. 2, 229–235.

[4] W. N. Colquitt and L. Welsh, Jr., *A new Mersenne prime*, Mathematics of Computation **56** (1991), no. 194, 867–870.

[5] Gerald A. Edgar, *Measure, topology and fractal geometry*, Springer-Verlag, New York, 1990.

[6] Andrew Fyfe, *Properties of the V-C dimension*, Master's thesis, California Institute of Technology, 1990.

[7] David Haussler, *Generalizing the PAC model: sample size bounds from metric dimension-based uniform convergence results*, Proceedings of the 30th IEEE Symposium on Foundations of Computer Science, 1989, pp. 40–45.

[8] David Haussler, Michael Kearns, and Robert E. Schapire, *Bounds on the sample complexity of Bayesian learning using information theory and the VC dimension*, Proceedings of the Fourth Annual Workshop on Computational Learning Theory, Morgan Kaufmann Publishers, 1991, pp. 61–74.

[9] A. N. Kolmogorov and V. M. Tihomirov, *ϵ-entropy and ϵ-capacity of sets in functional spaces*, American Mathematical Society Translations, Series 2 **17** (1961), 277–364.

[10] Benoit B. Mandelbrot, *The fractal geometry of nature*, W. H. Freeman and Company, New York, 1977.

[11] James L. McClelland and David E. Rumelhart, *Explorations in parallel distributed processing*, The MIT Press, Cambridge, Massachusetts, 1988.

[12] Marvin L. Minsky and Seymour A. Papert, *Perceptrons*, expanded ed., The MIT Press, Cambridge, Massachusetts, 1988.

[13] David Pollard, *Convergence of stochastic processes*, Springer-Verlag, New York, 1984.

[14] N. Sauer, *On the density of families of sets*, Journal of Combinatorial Theory, Series A **13** (1972), no. 1, 145–147.

[15] Patrice Simard, Bernard Victorri, Yann Le Cun, and John Denker, *Tangent prop — a formalism for specifying selected invariances in an adaptive network*, Advances in Neural Information Processing Systems 4 (John E. Moody, Steven J. Hanson, and Richard P. Lippmann, eds.), Morgan Kaufmann Publishers, 1992, pp. 895–903.

[16] V. N. Vapnik and A. Ya. Chervonenkis, *On the uniform convergence of relative frequencies of events to their probabilities*, Theory of Probability and Its Applications **16** (1971), no. 2, 264–280.

[17] ———, *Necessary and sufficient conditions for the uniform convergence of means to their probabilities*, Theory of Probability and Its Applications **26** (1981), no. 3, 532–553.

[18] Vladimir Vapnik, *Estimation of dependences based on empirical data*, Springer-Verlag, New York, 1982.