Complex Systems **1** (1987) 877–922

Large Automatic Learning, Rule Extraction, and Generalization

John Denker
Daniel Schwartz
Ben Wittner
Sara Solla
Richard Howard
Lawrence Jackel
AT&T Bell Laboratories, Holmdel, NJ 07733, USA

John Hopfield
AT&T Bell Laboratories, Murray Hill, NJ 07974, USA
and
California Institute of Technology, Pasadena, CA 91125, USA

Abstract. Since antiquity, man has dreamed of building a device that would "learn from examples", "form generalizations", and "discover the rules" behind patterns in the data. Recent work has shown that a highly connected, layered network of simple analog processing elements can be astonishingly successful at this, in some cases. In order to be precise about what has been observed, we give definitions of *memorization*, *generalization*, and *rule extraction*.

The most important part of this paper proposes a way to measure the *entropy* or *information* content of a learning task and the *efficiency* with which a network extracts information from the data.

We also argue that the way in which the networks can compactly represent a wide class of Boolean (and other) functions is analogous to the way in which polynomials or other families of functions can be "curve fit" to general data; specifically, they *extend the domain*, and *average noisy data*. Alas, finding a suitable representation is generally an ill-posed and ill-conditioned problem. Even when the problem has been "regularized", what remains is a difficult combinatorial optimization problem.

When a network is given more resources than the minimum needed to solve a given task, the symmetric, low-order, local solutions that humans seem to prefer are not the ones that the network chooses from the vast number of solutions available; indeed, the generalized delta method and similar learning procedures do not usually hold the "human" solutions stable against perturbations. Fortunately, there are

ways of "programming" into the network a preference for appropriately chosen symmetries.

1. Overview of the contents

Section 2 gives several examples that illustrate the importance of automatic learning from examples. Section 3 poses a test-case problem ("`clumps`") which will be used throughout the paper to illustrate the issues of interest. Section 4 describes the class of networks we are considering and introduces the notation. Section 5 presents a proof by construction that a two-layer network can represent any Boolean function, and section 6 shows that there is an elegant representation for the `clumps` task, using very few weights and processing units. Sections 7 and 8 argue that the objective function $E(W)$ has a complicated structure: good solutions are generally not points in W space, but rather parameterized families of points. Furthermore, in all but the simplest situations, the E surface is riddled with local minima, and any automatic learning procedure must take firm measures to deal with this. Section 9 shows that our `clumps` task is a very simple problem, according to the various schemes that have been proposed to quantify the complexity of network tasks and solutions. Section 10 shows that a general network does *not* prefer the simple solutions that humans seem to prefer. Section 11 discusses the crucial effect of changes of representation on the feasibility of automatic learning. We prove that "automatic learning will always succeed, given the right preprocessor," but we also show that this statement is grossly misleading since there is no automatic procedure for constructing the required preprocessor. Sections 12 and 13 propose definitions of rule extraction and generalization and emphasize the distinction between the two. Section 14 calculates the entropy budget for rule extraction and estimates the information available from the training data and from the "programming" or "architecture" of the network. This leads to an approximate expression for the efficiency with which the learning procedure extracts information from the training data. Section 16 presents a simple model which allows us to calculate the error rate during the learning process. Section 17 discusses the relationship between rule extraction in general and associative memory in particular. In section 18, we argue that when special information is available, such as information about the symmetry, geometry, or topology of the task at hand, the network must be provided this information. We also discuss various ways in which this information can be "programmed" into the network. Section 19 draws the analogy between the family of functions that can be implemented by networks with limited amounts of resources and other families of functions such as polynomials of limited degree. Appendix A contains details of the conditions under which our data was taken.

2. Why learn from examples?

Automatic learning from examples is a topic of enormous importance. There are many applications where there is no other way to approach the task.

For example, consider the problem of recognizing hand-written characters. The raw image can be fed to a preprocessor that will detect salient features such as straight line segments, arcs, terminations, etc., in various parts of the field. But what then? There is no mathematical expression that will tell you what features correspond to a "7" or a "Q". The task is defined purely by the statistics of what features conventionally go with what meaning—there is no other definition. There is no way to program it; the solution must be learned by examples [6,11].

Another example is the task of producing the correct pronunciation of a segment of written English. There are patterns and rules of pronunciation, but they are so complex that a network that could "discover the rules" on its own would save an enormous amount of labor [37].

Another example concerns clinical medicine: the task of mapping a set of symptoms onto a diagnosis. Here the inputs have physical meaning—they are not purely conventional as in the previous examples—but we are still a long way from writing down an equation or a computer program that will perform the task *a priori*. We must learn from the statistics of past examples [41].

Other examples include classifying sonar returns [10], recognizing speech [5,16,30,23], and predicting the secondary structure of proteins from the primary sequence [42].

In the foregoing examples, there was really no alternative to learning from examples. However, in order to learn more about the power and limitations of various learning methods and evaluate new methods as they are proposed, people have studied a number of "test cases" where there was an alternative—that is, where the "correct" solution was well understood. These include classifying input patterns according to their parity [33], geometric shape [33,35], or spatial symmetry [36].

3. Example: `two-or-more clumps`

The test case that we will use throughout this paper is a simple geometric task which an adaptive network ought to be able to handle. The network's input patterns will be N-bit binary strings. Sometimes we will treat the patterns as numbers, so we can speak of numerical order; sometimes we will also treat them as one-dimensional images, in which *false* bits (Fs) represent white pixels and *true* bits (Ts) represent black pixels. A contiguous clump of Ts represents a solid black bar. We then choose the following rule to determine the desired output of the network, as shown in table 1: if the input pattern is such that all the Ts appear in one contiguous clump, then the output should be F, and if there are two or more clumps, then the

Input pattern	Output	Interpretation
fftttttfff	F	1 clump
fftttftfff	T	2 clumps
ftttttttft	F	1 clump
tttfftfft	T	3 clumps
fffffffff	F	no clumps

Table 1: Examples of the `two-or-more clumps` predicate.

output should be T. We call this the `two-or-more clumps` predicate.[1] We will consider numerous variations of this problem, such as `three-versus-two clumps` and so forth. The `one-versus-two clumps` version is also known as the `contiguity` predicate [25]. Questions of connectedness have played an important role in the history of networks and automatic learning: Minsky and Papert devoted a sizable portion of their book [27] to this sort of question.

There are a host of important questions that immediately arise, some of which are listed below. In some cases, we give summary answers; the details of the answers will be given in following sections.

> Can any network of the type we are considering actually represent such a function? (Yes.) This is not a trivial result, since Minsky and Papert [27] showed that a Perceptron (with one layer of adjustable weights) absolutely could not perform a wide class of functions, and our function is in this class.
>
> Can it perform the function efficiently? (Yes.) This is in contrast, say, to a solution of the parity function using a standard programmable logic array (PLA) [26], which is possible but requires enormous numbers of hardware components ($O(2^N)$ gates).
>
> Can the network learn to perform this function, by learning from examples? (Yes.)
>
> How quickly can it learn it? (It depends; see below.)
>
> How many layers are required, and how many hidden units in each layer? How do the answers to the previous questions depend on the architecture (i.e. size and shape) of the network?
>
> How sensitive are the results to the numerical methods and other details of the implementation, such as the analog representation of T and F, "momentum terms", "weight decay terms", step size, etc.?
>
> Does the solution (i.e. the configuration of weights) that the network finds make sense? Is it similar to the solutions that humans would choose, given the task of designing a set of weights to perform the assigned task? Is its solution logical and systematic, or is it a kludge?

[1] A predicate is a Boolean-valued function; that is, its range is the set $\{T, F\}$. The domain can be anything you like.

Does the network discover the rule? That is, when it is trained on a *subset* of the possible input patterns, does it find a configuration that yields the "correct" output for the *remaining* input patterns?

4. Basics

The purpose of this section is to establish the notation. We assume that the reader has a basic familiarity with layered analog networks. This is a brutally brief review of a standard, vanilla model; there are many possible refinements and extensions that will not be mentioned here.

Information on the history of these ideas can be found in reference [29]. A good survey of many successful applications of adaptive networks can be found in reference [39]. A sophisticated and detailed analysis, dealing mainly with one-layered networks, appears in reference [27]. A collection of recent work on parallel networks and learning from examples appears in reference [32]. Papers on related topics, including the feasibility of building such networks, can be found in reference [8].

Figure 4 shows a typical layered analog network. It has two layers of weights ($L = 2$), and, equivalently, two stages of processing. There are $N^{[2]}$ processing units in the output layer, and $N^{[1]}$ units in the hidden layer; we use a superscript in square brackets to indicate layer number. There are $N^{[0]}$ input wires, which are not considered processing units, although we sometimes refer to them as the 0^{th} layer. The input bits have value $V_i^{[0]}$. The weights in layer l have values $W_{ji}^{[l]}$, for $l = 1$ to L. The empty triangles are amplifiers with a sigmoid transfer function $g()$ such as $g(x) = \tanh(x)$. The amplifiers do not have a separate "bias" or "threshold" input; the bias function is implemented by the weights connected to the "stuck bits". The stuck bits are provided by the triangles with no inputs; a "+" or "−" inside indicates that the output is stuck at T or F respectively. In each layer l, we number the unstuck bits from 0 to $N^{[l]} - 1$, and designate the stuck bit as number $N^{[l]}$ for a total of $N^{[l]} + 1$ connections to each unit in layer $l + 1$.

The input to each amplifier in layer l is given by:

$$u_j^{[l]} = \sum_{i=0}^{N^{[l-1]}} W_{ji}^{[l]} V_i^{[l-1]} \tag{4.1}$$

and the corresponding output is:

$$V_j^{[l]} = g(u_j^{[l]}) \tag{4.2}$$

Equations 4.1 and 4.2 define the input-output relation of the network, depending on the W values of course. Denoting the input of the network as $I_i \equiv V_i^{[0]}$ and the actual output as $A_k \equiv V_k^{[L]}$, we write this overall input-output relation as:

$$A = A(W; I). \tag{4.3}$$

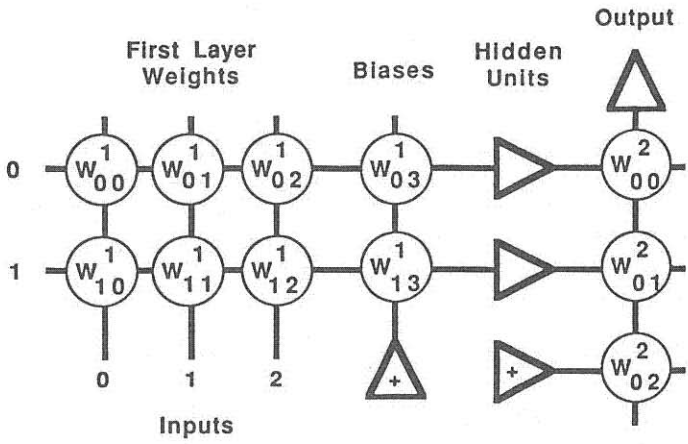

Figure 1: Typical network.

Suitable W values could be chosen by hand, but we are mainly interested in choosing them automatically—learning from examples. The training examples are ordered pairs: (input, desired output), denoted (I^α, D^α), where α runs over all examples. The traditional measure of how well the network has memorized the training examples is:

$$E(W) = \sum_\alpha |D^\alpha - A(W; I^\alpha)|^2, \tag{4.4}$$

that is, what you wanted minus what you got, norm squared. This makes the task of learning equivalent to the task of searching W space for a minimum of $E(W)$, hence the name "least mean squares" (LMS) learning method. In the case when $E(W)$ is differentiable, there are several powerful methods for descending the E surface. Notable among these is the "generalized delta method," which can be efficiently calculated by the "back propagation" algorithm.

5. Representability

Here is a proof that a two-layer network can implement any Boolean function. Note that we are discussing representability, not learning, so it suffices to construct by hand a representation.

The discussion here essentially parallels the proof in reference [27] that any predicate can be expressed in terms of masks. Also, for those readers with a background in digital circuits, there is a one-line proof: the two-layer network contains the PLA as a special case, a PLA contains a ROM as a special case, and an N-input ROM can implement an arbitrary Boolean function of N bits.

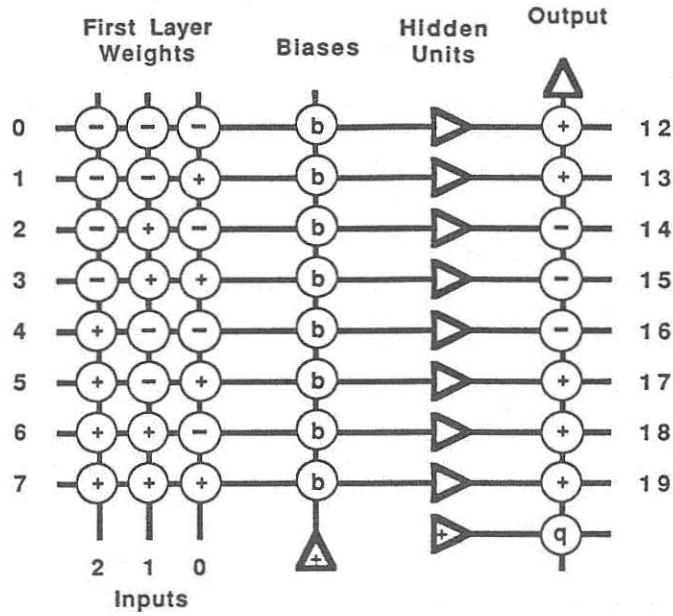

Figure 2: Network to represent an arbitrary predicate.

In fancy network language, the proof goes like this: construct a two-layer network with $N \equiv N^{[0]}$ inputs and 2^N hidden units. There are exactly 2^N possible input patterns, and we dedicate one hidden unit to each input pattern. You can call the hidden units "match filters" or "grandmother cells" if you like. The weight matrix $W^{[1]}$ for the first layer is chosen as follows: the weight connecting input bit i to hidden unit j will be called $W_{ji}^{[1]}$, for $i = 0$ to $N-1$ and $j = 0$ to $2^N - 1$. $W_{ji}^{[1]}$ is set to $+b$ if the i^{th} bit is set in the binary representation of the number j, and is set to $-b$ otherwise. This ensures that when input pattern number j is present at the input, the j^{th} hidden unit will receive a contribution of strength $+Nb$ from the matrix, and all other units will receive lesser contributions (by steps of $2b$). We give each hidden unit a threshold of $(N-1)b$. We make b very large, which is equivalent to making the gain of the transfer function very large (so the transfer function becomes a step function). Then the j^{th} unit will be turned on, practically to saturation, and all the other hidden units will be turned off, practically saturated in the negative direction.

The rule for choosing the second layer of weights is similarly straightforward: if the k^{th} output bit is supposed to be a T when the j^{th} input pattern is presented, then the weight $W_{kj}^{[2]}$ is set to $+p$, and set to $-p$ otherwise. The bias of output unit k is set to S_k, the algebraic sum of all the weights feeding that unit.

The circuit diagram in figure 5 is practically a picture of the truth table of the chosen input-output relation. The weights in the first layer are an enumeration of the possible input patterns, and the weights in the second layer are the corresponding output patterns. The device can also be thought of as a fully decoded Read Only Memory (ROM). The first layer of weights forms the decoder, and the weights in the second layer are the stored memories.

This demonstrates that the two-layer network can represent any Boolean function.[2] This is in stark contrast with the single-layer network, which can only represent linearly separable functions [27], which are not very interesting. The term refers to the fact that a one-layer network can only separate the input space into simple categories: the separation boundary can only be a function linear in the input variables, i.e., a hyperplane.

This solution is quite general, but it is also quite inefficient. The number of weights in the network grows exponentially as the number of input bits is increased.

The astute reader may have noticed that in the case of a single output bit, the number of hidden units can be reduced by a factor of at least two, by using a wired-or (or wired-and). This trick doesn't work for $A \gg 1$.

6. Efficient representations

For many tasks of practical interest, there exist efficient representations. Figure 6 shows an efficient representation for our clump-counting task. The solution capitalizes on the idea that clumps have edges. The first layer of weights is used as an array of (falling, right-hand) edge detectors, and the second layer counts the number of edges that were detected.

Figure 6 is laid out in a non-standard way in order to clarify a special case: what happens if the edge of a clump coincides with the edge of the input field? To eliminate this, we embed the N real input bits using $N+2$ wires, where the rightmost and leftmost wires are stuck in the F state. Now every edge of every clump can be seen a transition from T to F or vice versa. The N real inputs are labeled 0 to $N-1$; the special edge wires are labeled -1 and N. The ordinary bias line is stuck at T and is labeled $N+1$. Of course, when actually building such a network, the function of the edge bits would be combined with the ordinary bias line.

The weights are assigned as follows: $W_{ji}^{[1]}$ is set to $+b$ if $j = i$, $-b$ if $j = i+1$, and zero otherwise. The biases are all set to $-b$. This ensures that hidden unit j will be "high" if and only if the corresponding input bit is in

[2]We are quite aware that in many practical cases, the inputs are not Boolean, but can take on a range of values. That situation is quite a bit more complex. Consider a network with a single input wire, taking values x in the half-open interval $(0, 1]$. The simple function which yields the parity of the largest integer less than $1/x$ will obviously require an arbitrarily large number of hidden units. Furthermore, Lippmann (private communication; see also [23]) has shown that in the case of continuous-valued (non-Boolean) *output* levels, there are functions that cannot be represented with two layers, no matter how many hidden units are used.

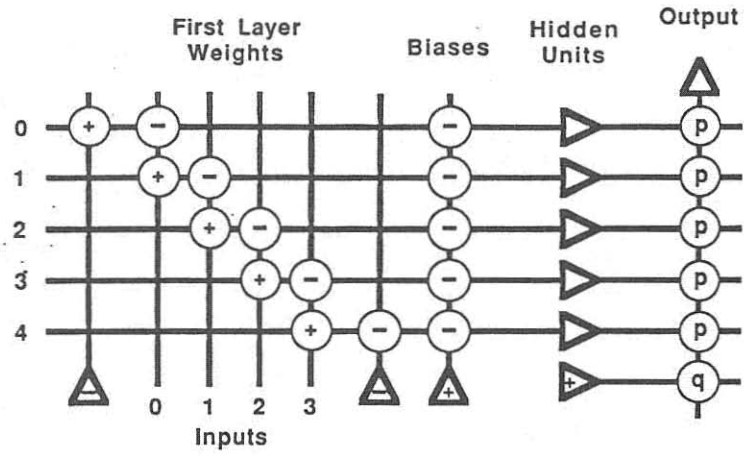

Figure 3: Efficient network for clump counting.

the T state *and* its neighbor to the right is in the F state. Specifically, the input to hidden unit j will have one of three possible values:

$$u_j = \begin{cases} +b & \text{if } (I_{j-1}, I_j) = (T, F) \\ -b & \text{if } (I_{j-1}, I_j) = (T, T) \text{ or } (F, F) \\ -3b & \text{if } (I_{j-1}, I_j) = (F, T) \end{cases} \tag{6.1}$$

The weights in the second layer are $+p$, and the bias unit in the second layer is q. Hence, for a pattern containing C clumps, the input signal to the output unit will be

$$u = Cp\,g(+b) + Cp\,g(-b) + (N+1-2C)p\,g(-3b) + q. \tag{6.2}$$

We want the network to return T when there are C clumps and F when there are $C-1$ clumps. This leads to the following equation, which determines the optimal value of p:

$$\begin{aligned} g^{-1}(T) - g^{-1}(F) &= pg(+b) + pg(-b) - 2pg(-3b) \\ &\approx pg(+b) - pg(-b) \quad \text{for large } b \end{aligned} \tag{6.3}$$

This construction shows that it is possible to represent the clump-counting problem using a two-layer network with only $H = N + 1$ hidden units. This proves that at least one efficient representation exists, for all N; we do not claim that this is the only such representation or even the best representation. We call this the *geometric* solution. We also refer to it as the *human*

solution, since when asked to design a network to solve this task, people tend to choose this solution.

We emphasize that in the case where all the training patterns contain either C or $C-1$ clumps, this is an exact, $E=0$ solution.

Since $g()$ is bounded and monotonic, $b \to \infty$ implies $g(-b) \to g(-3b)$. One consequence is that in this limit, the output of the $j=0$ hidden unit is nearly constant. That hidden unit can be eliminated entirely and its output weight lumped in with the ordinary bias unit, simplifying the construction of the network. The problem of searching for a minimum of $E(W)$ must be stated properly in this case, since for finite b there is no solution with $E=0$, or even with E equal to a minimum. Yet in all situations that we care about, $E = E_{\min}$ is not required; $E < E_{\min} + \epsilon$ is the proper criterion for the solution, and ϵ can be made as small as you like by increasing b. We will return to this point in section 8 (see also [40]).

The `clumps` predicate can also be represented by a *one*-layer network of sigma-pi units, which has been explored by reference [25].

7. The structure of weight space

Solutions should not be thought of as a single configuration of weights, but as a class of equivalent configurations, differing only by certain symmetries. One important symmetry comes from our freedom to choose b; we call this the b-symmetry. Also, in the large b limit it is possible to choose a different b_j for each hidden unit j; we call this the b_j-symmetry. This means that the solution set includes a large region of a H-dimensional subspace of weight space.

This phenomenon is not confined to our `clumps` example. Clearly, whenever a unit is being used as a linear threshold element, uniformly increasing its input weights leaves the result unchanged.

Another important symmetry comes from the fact that the ordering of the hidden units is arbitrary. In each hidden layer, we can relabel the units in $H!$ ways, where H is the number of units in that layer. Specifically, if we set $W'^{[l]}_{j,i} = W^{[l]}_{P(j),i}$ and $W'^{[l+1]}_{k,j} = W^{[l+1]}_{k,P(j)}$ for any hidden layer l and for any permutation P, then the behavior of the network is absolutely unchanged.

Yet another symmetry comes from the fact that in most networks that have been considered the polarity of the hidden units is arbitrary. In the representation where the transfer function is odd (which is typical when $T=1$ and $F=-1$), this symmetry is very simple and easy to visualize: for each hidden unit, if we change the sign of every weight on its input *and* every weight on its output, then the behavior of the network is unchanged. In other representations where the transfer function is an odd function plus a constant (which is typical when $T=1$ and $F=0$), the symmetry is still there, but requires an adjustment to the bias terms in the next layer.[3] The number of equivalent configurations in each hidden layer is $2^H H!$. Whenever

[3]One can construct networks in which the transfer function has no particular symmetries, but we have seen no advantage in doing so.

one wishes to compare two solutions, one must remove these symmetries by converting the configuration to some standard form.

These symmetries imply a certain periodicity in W space. This leads us to visualize the $E(W)$ surface as resembling a sombrero, or as a phono record that has been warped in certain symmetric ways: near the middle ($W = 0$) all configurations have moderately bad E values. Radiating out from the center are a great number of ridges and valleys. The valleys get deeper as they go out, but asymptotically level out. In the best valleys, E is exactly or asymptotically zero; other valleys have higher floors. This picture is, of course, an oversimplification.

8. Search methods

When the weights are very large (or $g()$ is a sharp threshold function), the task of "learning" the best set of weights is an unvarnished combinatorial optimization problem, and the network formulation does not offer any discernible advantage over conventional combinatorial optimization procedures. However, by varying the gain (steepness) of the transfer function, Hopfield and Tank were able to improve the results of a similar search. It is possible for the system to express temporary compromises, moving through weight space along complex paths such that many components (i.e. the weights) have intermediate values. In favorable situations, this allows the system to circumnavigate small barriers in the $E(W)$ surface, exploiting the high dimensionality of W space.

In order to promote this sort of behavior, one can introduce an additional term in the learning process which we refer to as *deterministic weight decay*. (Other types of weight decay will be discussed shortly.) This additional term effectively defines a new surface $E'(W) = E(W) + E_2(W)$, where E_2 is large when $|W|$ is large. Of course, searching the new surface is a different task than searching the original surface. If E_2 is too large, then the solutions to the new task do not correspond to the solutions of the original task, and we have accomplished nothing. Indeed, if E_2 is extremely large, it is possible to have only one solution, the parasitic solution $W = 0$. On the other hand, if E_2 is very small, it doesn't do any harm, but then it doesn't do any good either. A better procedure is to use a term $E_2(W, t)$, which is an explicitly decreasing function of time. Then, the real solutions will gradually emerge out of the parasitic solutions. It is hoped that the best solution will be the first to emerge, whereupon the system will find it and follow it as it evolves to its final form. This is one example of a class of techniques we call *simulated ironing*, because they remove wrinkles from the E surface. It is a nice vision of what *might* happen, but there is no guarantee that it *will* happen in a particular case. It is not a bad heuristic, but it is not guaranteed, either.

Weight decay also improves the performance of certain learning methods by preventing the method from wasting time seeking the $b \to \infty$ solutions discussed in section 6. Most versions of the generalized delta rule learn very slowly when the weights are large. Weight decay is a rather artificial way of

introducing stable local minima at finite values of b. We have shown that a better choice of objective function (i.e. other than LMS) greatly reduces the $b \to \infty$ problem [40].

Another heuristic that is very easy to implement involves using Euler's method with a step size that is a little too large. This situation was analyzed by Jeffrey and Rosner [19]. Basically, it can be thought of as introducing into the equation of motion some higher-order terms, notably terms proportional to the curvature of the E-surface. This penalizes *narrow* basins of attraction, relative to wide basins of comparable depth. The resulting trajectory in W space is very complex, and $E(W(t))$ is not monotonic. We call this behavior "thrashing". It is a quick and dirty expedient that helps when a clean, accurate descent of the E surface would have gotten stuck in narrow local minima.

There are several methods for escaping local minima which depend on introducing an element of randomness into the search procedure. The classical way is to simply restart the search many times from different random initial conditions and choose the best of these trial solutions [31]. Another well-known technique is to use *simulated annealing* [20], which systematically allows the system to take steps uphill on the E surface, with a probability depending on the "temperature", which is an explicitly decreasing function of time. One can introduce a free energy surface, F, and show that temperature smoothes the F surface just as simulated ironing smoothes the E surface. We wish to emphasize a point that is widely misunderstood: the two procedures, although similar, are not completely equivalent. Specifically, a probabilistic system will eventually cross a small barrier in the F space by activated hopping, while a deterministic system that always goes downhill can *never* cross a barrier, however small, in the E space.[4]

Another way in which an element of randomness can be injected is by using "incomplete smoothing". Recall that the E function is defined with respect to (a sum over) all the items in the memorization set M. It is possible, however, to consider partial contributions to E, defined with respect to subsets (or even single elements) of M. This leads one to compute motions in W space that minimize the various partial contributions, and the choosing of the subsets can introduce enough randomness to help the search escape from local minima. Global minima will remain stable if $E = 0$ there; otherwise, they might not.

Another technique is known as "stochastic weight decay" [43]. Rather than decaying all the weights a little bit, one may decay a (random) subset of the weights somewhat more. This, too, allows the system to escape local minima and encourages solutions that are robust against this sort of perturbation.

We use the term "jostling" to refer collectively to ironing, annealing,

[4]This depends somewhat on the order of limits: here we have taken the limit of long time before taking the limit of small fluctuations and averaging over large ensembles. We are conducting "slow" annealing. To do otherwise would defeat much of the purpose of annealing.

Measure[5]	Parity	Clumps
Layers	2	2
Hidden Units	N	N
Order of Predicate	N	2
Number of Weights	N^2	N
Dynamic Range	N	2
Bits Required	$N^2 \log N$	$N \log N$

Table 2: Comparison of complexity.

stochastic weight decay, and similar procedures. There are a number of questions associated with such procedures; for example, it is difficult to decide how quickly the amplitude of the random forces should be decreased. This scheduling problem first arose in the context of simulated annealing and has been extensively studied. Perhaps it should not be called the "annealing schedule" problem, but rather the "jostling schedule" problem, since it crops up in numerous methods that are distinctly not equivalent to annealing.

All of these methods for escaping local minima incur an enormous time penalty, and the penalty increases as the size (N) of the problem increases. This leads to the strong suspicion that the learning process is formally intractable in general. The similar problem of finding the minimum number of minterms for a PLA is known to be NP-complete [9], but we know of no proof one way or the other for networks.

9. Measures of complexity

In the general case, a network having $H = N$ hidden units would require roughly $H \times N = N^2$ weights, but our geometric solution (section 6) is very sparse. The number of (non-zero) weights is proportional to N to the first power only. In fact, each hidden unit has only two input weights, so this is what Minsky and Papert [27] called a second-order predicate. (We know that no first-order solution exists.) Also, note that the weights need only take on small integer values, so the number of bits needed to specify the entire configuration is very small, proportional to $N \log N$. Finally, note that the input weights have the property of short range, or locality: the two nonzero weights are not distributed at random, but are in fact adjacent.

We will argue that not all of these concepts are accurate measures of the complexity or difficulty of a task, but in this case, the preponderance of the evidence indicates that this is a very easy task. The order of the predicate, the number of layers, the number of weights, the number of bits of specification, and the range are all small. If they got much smaller, the problem would be reduced to a triviality. Table 2 contrasts this task with the `parity` predicate [33], which is often cited as a demonstration of the power of this sort of network.

The network can indeed learn to solve the `clumps` task, learning from examples. Table 3 shows how the learning time depends on the width of the

N	H	Median # Passes
6	2	1484
6	3	422
6	4	390
6	5	393
6	6	414
6	7	439
10	6	1747
10	10	1180
10	11	1072

Table 3: Learning time versus network size and shape.

hidden layer for two different sizes of input. Figure 9 is similar, showing how the learning time depends on the width of the hidden layer for fixed input size.

The data reported in the table and figure represent the *median* number of passes, where each pass consists of one presentation of each of the patterns in the training set M. Note that the median is a better measure than the mean, because sometimes the network fails to memorize the M data perfectly, even after arbitrarily long training time. It would be impossible to compute a mean in such cases, yet the median is still well behaved.

The discussion of intrinsic complexity and learnability will be continued below in section 18.

10. Stability of the human, geometric solution

It is interesting that table 3 indicates that the network was able to solve the task in those cases where the number of hidden units was less than the number of input bits. This answers one of the questions posed in section 3: the network is quite happy to find solutions that don't correspond to human designs. Humans seem to have a very hard time designing a solution in which $H < N$. Actually, our observations go further: even in the case where $H \geq N$, the network does not find the geometric solution.

To confirm and extend this result, we conducted a perturbation analysis as follows: we constructed a network with the geometric solution as the initial condition and then proceeded to further train it with the generalized delta method. Since the network was already at a solution, $E = 0$, no further adaptation took place. We then perturbed the system, moving the weights to a point W_S in weight space, and re-trained it. We found that the system was quite able to re-solve the task, returning to $E = 0$, but did not do so by undoing the perturbation. In fact, it moved in some other direction and settled on a *new* point W_F.

The learning procedure can be considered a dynamical system, describing the motion of the point W in weight space. Each local minimum of E is an

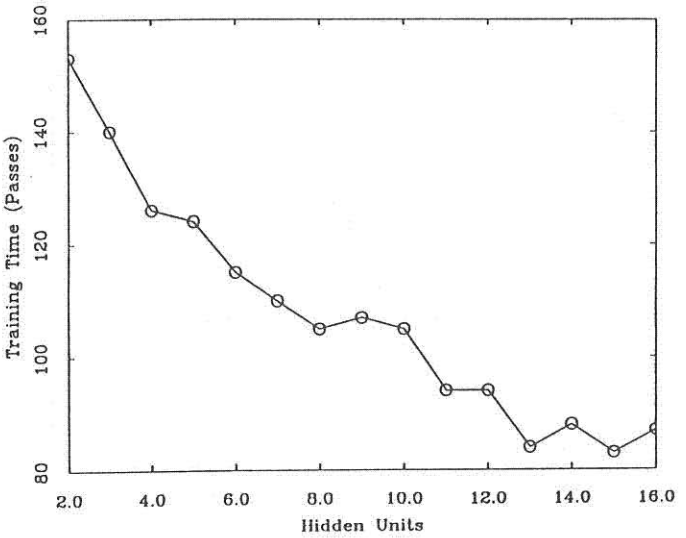

Figure 4: Learning time versus width of the hidden layer.

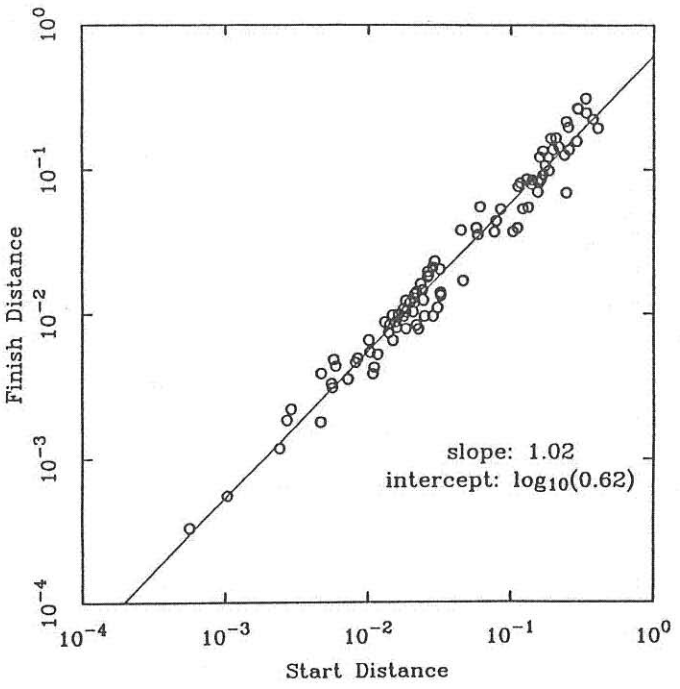

Figure 5: Perturbation of the geometric solution.

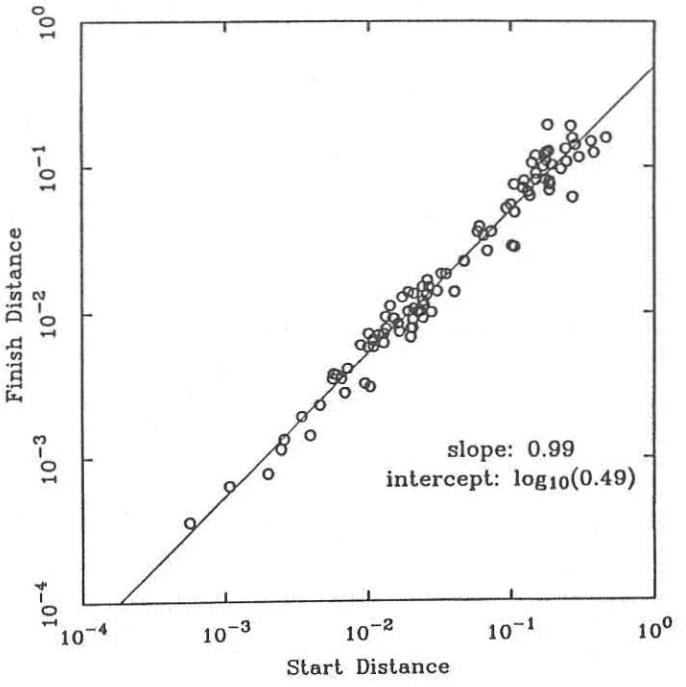

Figure 6: Perturbation with more symmetries removed.

attractor, and each will have its own basin of attraction. The "b symmetry" means that the attractor associated with the geometric solution is not pointlike but is at least one-dimensional.

To make the data in figure 10 more meaningful, we "projected out" this dimension by using the following slightly peculiar distance measure: first, we ignore the output weights and all the bias weights, so that only $H \times N$ weights remain. Then, we normalize by dividing by the biggest weight. Finally, we calculate the distances, using what we call the RMS metric: square each component of weight-difference, divide by the number of weights, and take the square root. We denote $D_S \equiv |W_S - W_0|$ and $D_F \equiv |W_F - W_0|$, where W_0 represents the geometric solution. In this projected space, the geometric solution is not a sole point, but a set of isolated points, because of the discrete symmetries discussed above.

If (in the projected space) the geometric solution had consisted of isolated pointlike attractors, figure 10 would have shown a characteristic flat region: $D_F = 0$ for all $D_S < D^*$, where D^* represents the radius of the basin of attraction. Since the figure shows no sign of a flat, zero region, we conclude that either the basin of attraction is exceedingly small, or that the attractor is not pointlike in the projected space, i.e. the attractor in the full W space has dimensionality greater than one. To say it another way, the b symmetry is not the only continuous transformation that leaves E invariant.[6]

Since we know that the b_j symmetry exists, we repeated the above experiment using a metric that projected out this larger symmetry. Specifically, for *each* hidden unit we normalized its weights to make the largest one equal to unity. This experiment checked the stability of a wider class of solutions, which have the same topology but not the same translational symmetry of the human solution. The results are shown in figure 10. Once again, there is no indication of any basin of attraction, and so we conclude that the $E = 0$ solution set is multidimensional in the projected space, and more than H dimensional in the original W space.

The slope in figures 10 and 10 are remarkably close to unity. We conjecture that this is essentially a consequence of the law of similar triangles. That is, the learning procedure can be seen roughly as a projection operator, which projects the whole W space onto the solution set. That would explain why doubling the perturbation D_S would double its projection D_F. We believe the intercept in these figures reflects the dimensionality of the solution set, relative to the number of dimensions collapsed by the learning-projection.

These experiments show that the human solution does not have positive stability—it is a neutrally unstable subset of a larger attractor. This is evidence against an important conjecture [42], namely that "the network is lazy; it will find the lowest-order predicate that is consistent with the data." We showed in section 6 that a second-order solution exists, yet the

[6] We reserve the word symmetry to refer to guaranteed universal symmetries such as the ones discussed in section 7. We use the broader term invariance to refer to anything that leaves E unchanged, including things which depend critically on the training data for this particular task.

generalized delta method does *not* show any sign of preferring this or any other second-order solution. In fact, it seems greatly to prefer N^{th}-order solutions (perhaps because they are more numerous).

We do not mean to imply that networks never find human-type solutions. Rumelhart, Hinton, and Williams [33] report that a back propagation network of precisely the type we are considering found a sensible "human" solution to the `parity` task, and Maxwell et al. [25] report that a one-layer network of sigma-pi units found a sensible "human" solution to the `clumps` task.

The concept of the order of a predicate is useful for some purposes but not all. For a task with (uniformly) bounded order, the number of connections required is less than it would be in a task of order N, and this could affect the practicality of building a system to solve the task. On the other hand, if a network is given the resources to build representation of a given order, we see no reason why it should not use all its resources. Perhaps one should use the notion of order to discuss restrictions on the resources available (or resources needed), rather than the resources used.

11. Preprocessors, representations, and feasibility

Although the network's ability to memorize and recall data is impressive, the thing that really stirs the imagination is the hope that the network could extend this behavior to "similar" data it had never seen. Indeed, in the early days of network research, it was hoped that the network would be able to generalize in several dramatic ways. Some of these powers have already been demonstrated; others remain topics of research, and others we believe to be unachievable. It might be hoped that:

> The network builds a sensible internal representation.
>
> The network serves as a "rule-finding" system (in contrast to conventional AI programs, which are referred to as "rule-following systems").
>
> The network behaves "as if it knew the rules."
>
> The learning process is largely unhindered by local minima.
>
> The network is good for "discovering hidden symmetries."
>
> The network generalizes.

In discussing these ideas, it is important to be clear about the meaning of the terms. Also, if we claim to be designing a network, we must be careful to specify in advance just what the network is expected to do. All too often, people build a network and then retrospectively discover what it is good for. This sort of analysis is useful, but should not be confused with synthesis.

It is also important not to make intemperate claims. It is clear that some networks can discover exceedingly general solutions to some tasks. For instance, let the input consist of binary numbers x and consider the predicate

"x is an odd number." Even a one-layer network can learn this predicate, using a very modest number of examples. It does in fact find an approximation of the "human" solution—it connects the output to the low-order input bit and ignores the other input bits. The network will then generalize from those few examples to every representable integer, which is a high ratio of generalization indeed.

On the other hand, consider a different predicate, namely "the number x has an odd number of prime factors." We can't prove that there is no network that can learn this predicate, but such a thing seems too good to be true. Factorization is considered an intractable computer-science problem, and the network is surely no more powerful than a standard computer.

Obviously, blatantly superficial structural properties (like the oddness of binary numbers) are easier to learn than deep abstract properties (like primality). But note that the question of what is superficial and what is deep is very sensitive to change in representation. In the base 3 representation, oddness is *not* such a simple structural property. Similarly, it is easy to imagine a representation in which a number is stored in terms of its factors, which would make factoring easy (and make addition hard).

In many cases, the practicality of solving a problem hinges on constructing a preprocessor that transforms the data into a reasonable representation. Indeed, here is a proof that "automatic learning will always succeed, given the right preprocessor": let the preprocessor accept the raw input and "tag" it with the desired answer. (By this we mean combining the two using the Cartesian product, i.e. concatenating the bit strings.) Feed this processed input into a one-layer perceptron. It is guaranteed to learn to ignore the raw data part of its input and copy the desired output to the actual output, just as it learned to copy the low-order bit in the `odd-even` task. This settles the debate concerning the importance of preprocessors; they are all-important. It is, alas, completely wrong to conclude from this that "automatic learning will always succeed," since we have no automatic procedure for generating the required preprocessor.

The questions of generalization, learnability, and representability are also sensitive to the architecture of the network. For example, the solution to the `parity` task using $H = N$ hidden units depends on the analog accuracy of the weights. If the weights could take on only binary values (which is the way it is in standard digital VLSI processes), then a two-layer network would need $O(2^N)$ hidden units. That would be a very silly representation, since a digital solution using $\log N$ layers of N units is much more efficient. For that matter, there is a digital solution using only a total of N XOR gates (although no simple learning method is known for such an architecture). The point is that the resources needed to do a good job on a given task are very sensitive to the shape of the network, the accuracy of the weights, the form of the objective function $E()$, and other architectural specifications.

Therefore, we emphasize that the question of generalization must be answered quantitatively, not categorically. Some network architectures, some learning methods, and some representations are suitable for some problems.

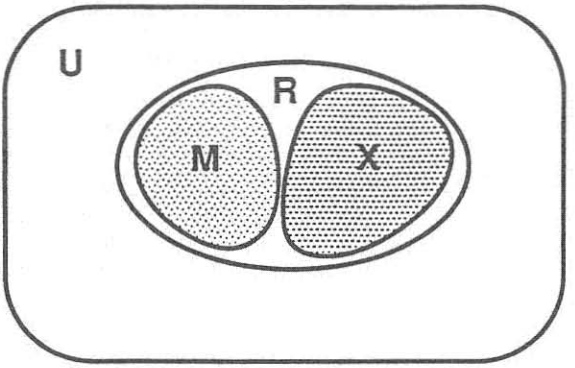

Figure 7: Rule extraction.

12. Definition of "rule extraction"

We hereby propose the following definition of rule extraction. This is, we believe, what most people mean by "finding the rules" or "discovering the symmetries" or "induction", and what some people mean by generalization. Reference works include Angluin and Smith [2], Holland et al. [14], Packel and Traub [28], and references therein.

Let I be the set of all possible inputs to the network, and let A be the set of all possible outputs. Let the set $U = I \times A$ be the universal set of all possible ordered pairs (input, output). All functions and other relations[7] are subsets of U. As indicated in figure 12, let us choose a particular function, a rule $R \subset U$, and see how well the network can discover it. In our specific example, the `two-or-more clumps` predicate defines the rule of interest. Now we identify a subset of R which we call the memorization set, M, and another set X which is disjoint from M, i.e. $X \subseteq R - M$. We call X the extraction set or the extension set, since the idea is to extract the rule from the data M, and extend it to the testing data X.

The network adaptively learns the data in the M set. The error function E which the generalized delta method seeks to minimize is defined as a mea-

[7]The `two-or-more clumps` predicate is, mathematically speaking, a function, since for each input pattern there is a unique output. The same is true for `parity` and most other predicates considered in the literature. Since all networks in the class we are considering are deterministic, the network's actual input-output relation is a function, too. In a real-world situation, the training data M may contain a certain amount of noise. By that, we mean two things. For one, we extend our definitions to include the case where a few of the elements of M may be inconsistent with the rule we are trying to find (i.e., they lie outside R). Second, in the presence of noise, M could easily be a relation that is not a function. Learning from noisy data is perhaps the most important use of networks that we can foresee. We will return to this topic below.

sure of how accurately the network's *actual* input-output relation matches the *desired* relation M. We define the network's *extraction score* (for the rule R) to be the the accuracy with which the network's input-output relation agrees with the X relation—that is, the data that it did *not* see during learning.

Occasionally, kibitzers suggest that we could improve the extraction score of our networks by including a "few" of the points from X in the learning process; we emphasize that to do so would by definition defeat the idea of rule extraction. The network's ability to produce the desired output, given input data that it saw during training, we call memorization; the ability to produce the desired output corresponding to input data that it has *never seen* before, we sometimes call generalization of the rule R, but we prefer to call it rule extraction.

We emphasize that rule extraction is a rather slippery concept, since it is possible to change a network's extraction score (without changing the network) simply by changing one's mind about what rule was "supposed" to be extracted.

In order for rule extraction to make sense, we require the property we call *representativity*; that is, the M and X sets must be representative samples of R. It generally suffices to construct M and X by the same random process. (Figure 12 should not be taken to mean that M and X are systematically different.) The point is to rule out nonsense of the following sort: suppose we were seeking to extract the `odd-even` predicate. Imagine that the subsets were manipulated so that the abscissas in M were all equal to 0 or 7 mod 8, while the abscissas in X were equal to 1 or 6 mod 8. That is, in one set there is a positive correlation between the bit we care about and its two neighbors, while in the other set there is a negative correlation. In such a perverse case, we should not expect good rule extraction.

We have performed experiments to see how well the generalized delta method and variants thereof can extract the `clumps` rule. In some circumstances, the rule extraction score is quite disappointing, and in other circumstances, it is rather good. Our struggle to reconcile these two results led to most of the ideas in the following sections.

For smallish networks ($H \approx N \approx 10$ or less), and even using a substantial fraction of all available data for training, we observed rather poor rule extraction. To confirm this result, we performed a perturbation analysis similar to the one described above in section 10.

The network used in these experiments had $H = N$, not $H = N + 1$, so the best performance (measured with respect to the testing set $X = R - M$) would occur for large b; specifically, $b \geq 4$ suffices. For lesser values of b, the initial condition does not constitute a solution, but is only a hint, having the same symmetry and topology as the true solution. Given an exceedingly strong hint, the performance was 100 percent, even with no training data, as can be seen in figure 12. Given no hint at all, the performance was poor even after training. Given a moderately strong hint, training with one or two hundred examples led to good performance. The amount of hint required

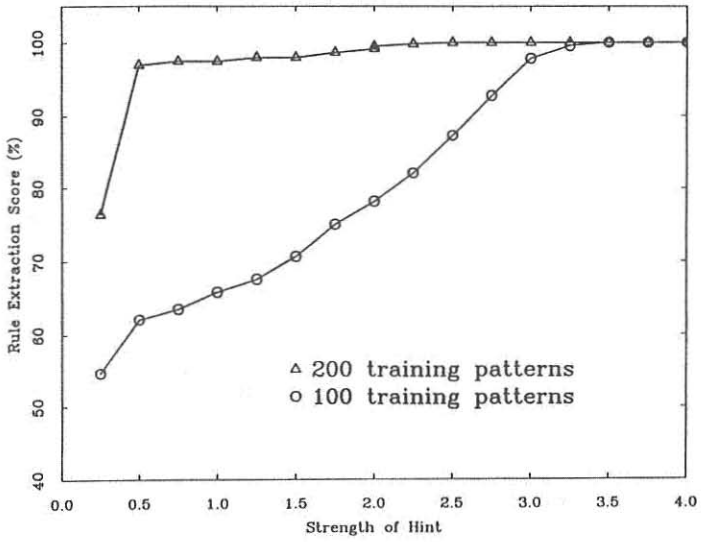

Figure 8: Rule extraction with weak hints.

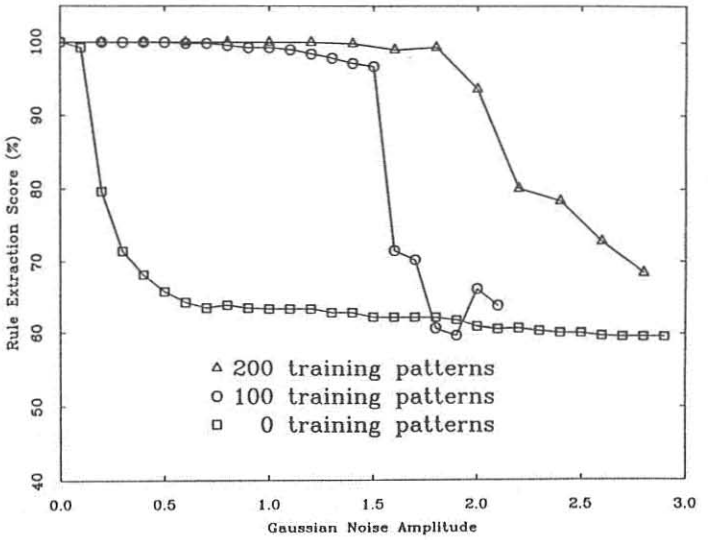

Figure 9: Rule extraction with noisy hints.

decreased dramatically when the size of the training set was increased.

In similar experiment, the network was always given a strong hint ($b = 3.5$), but the weights were perturbed by adding random noise. This is shown in figure 12.

The data in figures 12 and 12 shows a distinct basin of attraction. That is, there is a special region in W space surrounding the geometric solution. When started from any point within this region, the learning procedure leads to a final point with a rather good rule extraction score (X score). Starting outside that region produces a final point with a rather poor X score (even though it has $E \approx 0$).

This is in contrast to figure 10, which did not show any basin of attraction. From this, we conclude that the geometric solution, indeed the whole multidimensional solution set related to the geometric solution, is a very small subset of the region of W space that has a good X score.

On the other hand, figures 12 and 12 show that beyond a certain distance, the learning procedure does *not* lead to a point with a good X score, geometric or otherwise. This is the basis for our strongest negative conclusion: we have reason to believe that our implementation of the learning procedure is not faulty (since it does exhibit learning and rule extraction for some starting points), yet for general starting points it does *not* find a set of weights with a reasonable rule extraction score.

This result applies under the stated conditions of H, N, m, etc.; we emphasize that under other conditions we have achieved a very good X score, as will be discussed below.

13. Definition of "generalization"

To pursue these ideas, we must now define what we mean by true generalization (as distinct from rule extraction). As before, let U be the universe of relations, and let M be the memorization data, a subset of U. Now consider the sets G_1, G_2, G_3, etc. (For simplicity, let us restrict M and G_i to be functions for now.) We say that G_i is a *generalization* of M if M is a proper subset of G_i, as shown in figure 13. That is, the relation G_i has a larger domain than M, and the two relations agree wherever their domains overlap.

Note that we do not speak of *the* generalization, but one of many generalizations. There are, in fact, an enormous number of generalizations, and it is interesting to calculate the number. Consider a network, a definite network that is not, for now, undergoing training. This network performs some definite Boolean function, and we can tabulate this function in a truth table. Since there are N input bits, the truth table will have 2^N rows. We now ask how many distinct functions can exist. If there are a output bits, there are 2^a possible output symbols for each row of the truth table and a total of $2^N \times a$ independent bits on the output side of the table. We choose values for those bits in all possible ways and calculate how many different truth tables exist; namely, $\#(\mathcal{F}) = (2^a)^{2^N}$ (where $\mathcal{F}$ denotes the set of all functions, i.e. truth

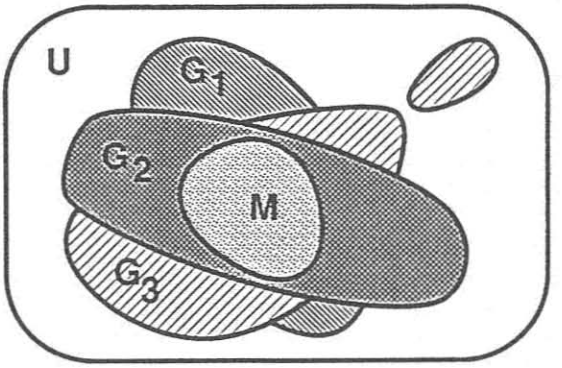

Figure 10: Many generalizations are possible.

tables). In the case of a single output bit, this simplifies[8] to $\#(\mathcal{F}) = 2^{2^N}$ possible Boolean functions.

It is sometimes useful to imagine the space of all possible truth tables as a hypercube with 2^N dimensions. Each dimension is a row of the truth table, and each corner is one of the 2^{2^N} distinct functions.

Now in quest of generalizations, we ask which functions are consistent with our training data M. Let $m = \#(M)$ be the number of data items (ordered pairs) in the M set. This data dictates the output values for m of the rows in the truth table, freezing out m of the dimensions in the truth hypercube. There remain $(2^N - m) \times a$ undetermined bits, and hence the number of possible generalizations is

$$\#(\mathcal{G}) = (2^a)^{2^N}/(2^a)^m \tag{13.1}$$

where $\mathcal{G}$ denotes the set of all G_i. This calculation is similar in spirit to the definition of Kolmogorov complexity [22]. The numbers in this equation are fantastically large. As a modest example, take a single output bit ($a = 1$), thirty input bits ($N = 30$), and a thousand training examples ($m = 1000$). Then there are $\#(\mathcal{G}) = 2^{10^9}/2^{10^3} = 2^{10^9}$ generalizations. These are all perfectly valid generalizations, in the sense that they are perfectly consistent with the training data M.

14. Entropy and information

We can use these ideas of function-counting to discuss rule extraction, and at first glance, the numbers seem very discouraging. The learning procedure

[8]We will henceforth discuss only the $a = 1$ case; the interested reader will have no difficulty in deriving the corresponding general expressions.

can result in any one of a vast number of possible generalizations. The data does not provide any reason to prefer any of these over any other. The set of networks that compute the rule we are seeking is a vanishingly small fraction of the set of all possible networks. It is inconceivable that any automatic learning procedure would be able to stumble onto the "correct" extension.

Therefore, we must not search the space of all possible functions or all possible networks. We must perform a constrained search. There are many ways of doing this.

One crucial idea is to search the space of all *efficient* networks. In some cases, we may know *a priori* that the task that has been posed *can be solved* by an efficient network. This special knowledge may come, for instance, from considerations of the geometry, symmetry, or intrinsic complexity of the task.

In other cases, we may not have such special knowledge, and we must in principle search all possible functions. These are what Abu-Mostafa [1] calls "random problems". However, all is not lost, because we have a choice: the crucial idea is that we choose to search the efficient network functions first. We have powerful search techniques for such functions. If, after efficiently searching the efficient networks, we still do not have a solution, we can then decide whether or not further searching is worth the trouble.

14.1 Preview

We are now almost ready to derive the main results of this paper. The discussion is a bit complex, so we will briefly list the main ideas and then proceed to derive them in detail.

1. We will be searching through the space of networks (i.e. weight space). Note that the previous section discussed searching through the space of functions; we will exploit the connection between the two spaces.

2. Since we will be calculating a probability, we need some notion of a prior distribution. We call this "throwing darts at weight space." It allows us to assign a probability $P(W)dW$ to each volume element dW in weight space.

3. We will extend the idea of exactly correct rule extraction by accepting nearly correct extractions with some error tolerance f. You can of course set $f = 0$ if you want to recover strict rule extraction.

4. We use the training data M and all other information at our disposal to restrict as much as possible the portion of weight space that we need to consider; call this portion $\widehat{W}$. We then calculate the probability that a point in $\widehat{W}$ will meet our f-acceptance criterion. If this probability is near 100 percent, we conclude that this network can be expected to perform rule extraction with the required accuracy f. On the other hand, if this probability is small, we conclude that the network can not be relied upon to perform rule extraction.

We remind the reader that one is *not* allowed to search $\widehat{W}$ space to find the "correct" rule extracting network. That cannot be done without using data from the testing set X, which defeats the purpose, by definition. That would be like betting on the winning horse after the race is over. We are only allowed to play the probabilities in $\widehat{W}$ space.

14.2 Derivation

The task of choosing a probability distribution in W space is a bit tricky. The choice depends on just what method is used for "learning", i.e. for searching W space. Fortunately, the exact form of the distribution is not important for our argument. You could, for instance, use a probability density proportional to $e^{|W|/\omega}$, for some "radius" ω. We will for most purposes use a distribution that is uniform inside a hypercubical volume (ω on a side) and zero elsewhere. We choose ω to be big enough to enclose reasonable weight values, but not too much bigger than that.

We can map weight space onto function space as follows: for each configuration of weights, W, build a network with those weights. Present it all possible binary inputs. Observe the corresponding outputs, and convert to binary. This mapping associates a definite truth table, i.e. a definite Boolean function, with each point in W space. To say it the other way, the inverse image of a function is a region in weight space.

By integrating over weight space, we can assign a probability P_i to each function. If ω is large enough, and if there are enough hidden units ($H \propto 2^N$), there will be non-zero probability assigned to every function, according to the discussion in section 5. On the other hand, we are particularly interested in the case where there are very few hidden units, perhaps $H \propto N^2$ or N^3. In that case, we expect many functions to have zero probability.

It is interesting to consider the quantity we call the "functional entropy", namely

$$S = \sum_{i \in \mathcal{F}} -P_i \log P_i \tag{14.1}$$

where $\mathcal{F}$ is the set of all functions. All logarithms in this paper are base 2, so entropy is measured in *bits*. It has its maximal value when all functions are equally likely, in which case $S = 2^N$. If some functions are less likely than others (or ruled out completely), the value of S is reduced.

We define S_0 to be the initial value of S, measured before any training has taken place. A large S_0 means that the network is capable of solving a large class of problems; a small S_0 means that the architecture has restricted the class of problems that this network can handle.

Now we get to use the training data M. The training data applies directly to function space, but we can use the mapping to "black out" the regions of W space that are (inverse images of functions that are) inconsistent with the training data. We can also define a reduced functional entropy,

$$S_m = \sum_{i \in \mathcal{F}^*} -P_i^* \log P_i^* \tag{14.2}$$

where the "*" indicates that we have removed from consideration all those functions that are inconsistent with the training data M and normalized the remaining probabilities. If all functions were equally likely, we might hope that each element of training data would reduce the entropy by one bit, so that $S_m = S_0 - m$. Alas, we forsee that this will not always be the case, so we define η to be the *average efficiency*[9] *with which the learning procedure extracts information from the training data*, defined by the expression

$$\eta = \frac{S_0 - S_m}{m} \tag{14.3}$$

It is sometimes useful to treat m as an independent variable and define the *local* efficiency,

$$\eta_m = -\mathrm{d}S_m/\mathrm{d}m. \tag{14.4}$$

There are a number of possible reasons why η might be less than unity. The most obvious possibility is that the training data might contain duplicate points. The first copy of that point would freeze out one axis of the truth-hypercube, reducing the entropy by one bit, but succeeding copies of that point would contribute nothing. We have obscured this issue by referring to the training data (M) as a set—and a set cannot have duplicate points. In the real world, however, the training data is treated as a list, not a set, so it can have duplicate points. We also point out that in the case where M is not a function, the phrase "rule out" is too strong a term. Perhaps "provide evidence against" would be better. Conflicting evidence, like duplicate evidence, will lower the efficiency.

A more profound and interesting case occurs when a training item cuts the truth-hypercube across an axis such that the entropy is not equally distributed among the two halves. It is well known that the optimum strategy for playing the game of twenty questions is to use questions that divide the number of possible objects in half each time. This strategy returns one bit of useful information per question; any other type of question returns less.

In the other direction, it might seem possible to construct a network with an artificially large η, even greater than unity, by hindsightfully choosing a rule R that coincides with whatever function the network initially produced. In such a case, the network could seem to "learn" the function with no data whatsoever! We insist, however, that this cannot be considered a rule extraction. Any claim of rule extraction which implies an efficiency greater than unity must be viewed with extreme skepticism.

It is possible, of course, every so often to stumble onto a good rule extraction based on very little data, just by dumb luck and waiting for coincidences.

[9]This is not to be confused with other forms of efficiency, such as efficiency of representation mentioned in section 6.

For instance, consider the following unusual game of 20 questions: Q: "Is your name Rumplestiltskin?" A: "Yes." This outcome is technically possible, but any strategy that asks such questions is a very poor one. It can only search 20 names, whereas a good strategy can search 2^{20} names. An apparent $\eta > 1$ will only occur if the rule R is very special, and/or the initial value of the weights W is very special. This sort of nonsense can be detected and prevented by evaluating η for a number of different, sensible rules and/or using an ensemble of randomized initial weights.

15. Calculating the efficiency

By introducing some approximations and restrictions, we can arrive at a convenient way of estimating the initial entropy in practical situations.

In the network of figure 4, each hidden unit has $\approx N$ weights connected to its input wire. If a signal of size u is significant to the hidden unit's input wire, a roundoff error of size $u/\sqrt{N}$ per weight would add up to be significant, if the errors added in quadrature, which is characteristic of random numbers. (Errors of size u/N would add up to be significant if they all added in the same direction.) From this, we conclude that each weight needs roughly $.5 \log N$ (or perhaps $\log N$) bits of precision. The fact that our solution, figure 6, has only three non-zero weights per hidden unit does not count, since we don't know that fact until after learning is completed.

Let us calculate the number of bits B needed to specify the configuration of the network. A network with $H \approx N$ hidden units has $O(N^2)$ weights, in which case $B = .5N^2 \log N$. Therefore, there are of order $2^{.5N^2 \log N}$ different networks that we can build. That is a large number, but it is quite a reduction from 2^{2^N}.

In general, to obtain a value for B, one must know L_i, the number of (significantly) different levels that each weight i in the network can take. Specifically,

$$B = \sum_{\text{weights}} \log L_i. \tag{15.1}$$

It is difficult to obtain an exact value for L_i, but it easy to obtain a good estimate, and only its logarithm matters anyway. The estimate can be performed by iteration: start with a small L and increase it, building a series of networks. Stop when you get a network that is capable of learning.

Our emphasis on networks with the minimum number of specification bits B is important not only for information-theoretic reasons, but also because there are important technological limits to the precision with which weights can be fabricated in real-world networks. The notion of an efficient representation, which was introduced back in section 6, can now be made precise: we require that B grow no faster than some polynomial in N, the size of the problem.

We take as our prior distribution the notion that all of our 2^B networks are equally likely.

We tried to arrange by construction that each of the B bits is "significant". Imagine for a moment that each bit was sufficiently significant that each time we change a bit the network implements a different Boolean function. In that case, there would be 2^B different functions, all equally likely, and the functional entropy would be $S_0 = B$.

This is clearly an overestimate for S_0, for a number of reasons. For one thing, the actual number of different functions implemented by our B-bit network will be less than 2^B, because of the various symmetries discussed in section 7. Furthermore, there will most likely be some low-order bit in the network specification somewhere that does not change the input output relation $A()$. This means that the 2^B imaginary networks are "folded" onto some smaller number of actual networks. We can get an expression for S by taking into account as many of these foldings as possible.

$$\begin{aligned} 2^{S_0} &= \frac{2^B}{H!2^H(F_1)(F_2)(F_3)\cdots} \\ &\leq \frac{2^B}{H!2^H} \end{aligned} \tag{15.2}$$

The first factor in the denominator accounts for the permutation symmetry of the hidden units. The second factor accounts for the polarity symmetry. The factor F_1 represents the fact that although our construction of B guarantees that most of the bits will be significant, quite likely not *all* of them will be. The fudge factor F_2 represents the b-symmetry; we do not know how much of this symmetry survives the coarse-graining involved in our construction of B. The factors $F_3\cdots$ represent any other symmetries that have simply escaped our attention. Setting these unknown factors to 1 gives us an upper bound on S_0.

We believe that whenever the final rule extraction score is significantly better than 50/50, the final entropy must be very small, for reasons given in section 16. In that case, we can estimate the efficiency as $\eta \approx S_0/m$, using the value of S_0 estimated from equation 2.

Other Viewpoints

Equation 14.3 defines an average efficiency η that was actually achieved in a particular case. Another viewpoint would be to imagine that there is a definite expected efficiency $\hat{\eta}$ that the network is capable of. We can then consider the expression

$$\delta S = S_0 - S_m - \hat{\eta} m \tag{15.3}$$

If δS is positive, then the number of consistent generalizations (consistent with the training data M) is large compared to the number of f-acceptable extractions, and we would expect that the rule extraction score would be very poor. On the other hand, if we increase m (the amount of training

data), then we can presumably winnow the number of valid generalizations down to the point ($\delta S \leq 0$) where they are all f-acceptable extractions.

Similarly, we could hold $\delta S = 0$ and hold η fixed, and solve equation 15.3 for the expected error rate f as a function of m.

Yet another possibility would be to consider everything but S_0 fixed. This equation would then tell us how much we would need to restrict the space of *a priori* likely functions, by limiting the number of hidden units or their connectivity, etc.

Our experiments indicate that the network can learn the `2 versus 3 clumps` predicate with reasonably high efficiency, as shown in table 4. For instance, using $N = 16$, the network was able to extract the `three-or-more clumps` rule with a $18:1$ gain factor ($\epsilon = 5.4$ percent), and 97 percent correct rule extraction ($f = 3.3$ percent). This gives $\eta = 56$ percent (assuming $S_m = 0$).

# Inputs N	Data Used m	Error Rate f	Efficiency η
16	800	.033	.56

Table 4: Rule extraction score versus N.

16. A model system

There is a model system which captures the our main ideas about rule extraction, yet is simple enough to be soluble. Consider the following feat: a particular card in a deck of 52 cards is marked, and a blindfolded mentalist offers to deduce which one it is. Now the simplest way is for him to ask a series of questions: Is the marked card in the top half of the deck? Is it in the first or third quarter? Is it in an odd-numbered eighth? and so on. After six questions, there is no doubt as to which of the cards is the marked one. The analogy to automatic learning is this: the deck is an ensemble of functions—imagine that a truth table is written on the back of each card. The distribution is weight space is such that it gives each of these functions equal probability and all other functions zero probability. The questions (with answers) are the training data.

This version of the model is not very realistic, since in real applications the training data generally has not been constructed to be optimal and orthogonal. Therefore, consider a second version of the model, in which the questions divide the cards in a random way. It is easy to visualize randomizing the cards, which is equivalent to randomizing the questions. The mentalist first asks if the marked card is in the top half of the deck. He then punches a hole through every card in the half that does *not* contain the marked card. Then the deck is shuffled. Again, he asks if the card is in the top half, and punches all the cards in the excluded half. This continues, with the deck being shuffled before each question. In this version, six questions do not suffice, but

after many questions have been asked it becomes exponentially unlikely for any card except the marked one to remain unpunched.

We now come to the third and final version of the model. We must take into account the fact that we do not expect the questions to divide the deck in a nice 50/50 way—the shuffle will not be perfect. What really happens is that each training item (i.e. ordered pair) focuses attention on a particular row of the truth tables that are written on the cards, namely, the row that matches the abscissa of that training item. The deck is then divided into two pieces, according to whether the ordinate on the card agrees or disagrees with the marked card. The ones that disagree get punched. The fraction that disagree is denoted ϕ; this is the principal free parameter of this model. A perfect shuffle would give $\phi = 1/2$.

Let the expected number of unpunched cards (after m questions) be Z; then,

$$Z = 1 + (2^{S_0} - 1) \prod_m (1 - \phi_m). \tag{16.1}$$

The first term, 1, represents the marked card, and the second term represents all the other unpunched cards, which are reduced by a factor of $(1-\phi_m)$ each time. For simplicity, we will treat ϕ as a constant from now on; it is easy to generalize the formulas.

If we pick a card at random from the set of unpunched cards, the probability of *not* picking the marked card is $(Z-1)/Z$, and therefore the total probability of picking up a card that disagrees with the marked card is

$$f = \phi \frac{Z-1}{Z}. \tag{16.2}$$

We have used the principle of representativity (discussed back in section 12) to connect ϕ, which is a property of the M set, with f, which is a property of the X set.

Now, suppose that we wish to train the network until it does better than some specified error rate f^*. How much data do we expect it to require? We assume that the network learns as efficiently as possible given adverse training data. Within this model, the worst ϕ that could be produced by any training set would be $\phi \approx f^*$, because if ϕ were smaller than this, equation 16.2 would imply that f was less than f^*, and the task would be complete, while if ϕ were larger than this, the network would learn faster according to equation 16.1. For small f^*, we expect

$$m \approx S_0 \ln 2 / f^*. \tag{16.3}$$

The network can learn faster than this if the error rate stays up near 50/50 until near the end of training, and it can, of course, learn more slowly if it does not live up to its information-theoretic potential.

The entropy is

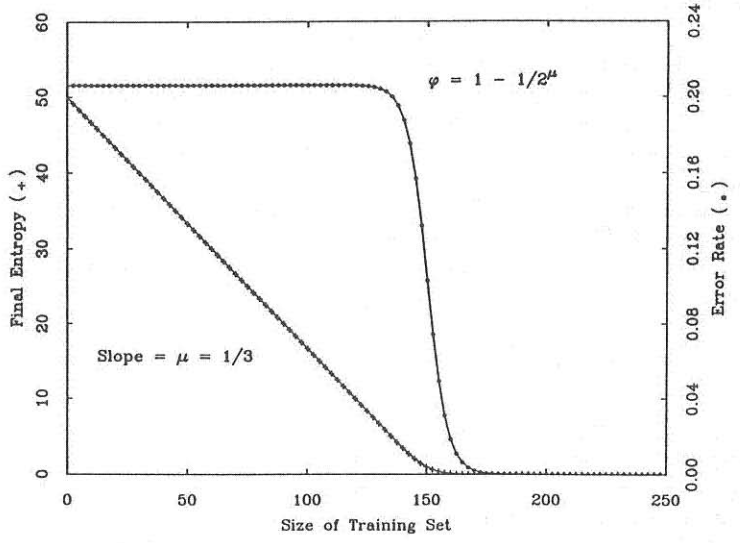

Figure 11: Entropy and error rate versus m.

$$S_m = \log Z. \tag{16.4}$$

It is useful to introduce the pre-efficiency $\mu \equiv -\log(1-\phi)$ and recall that the local efficiency $\eta_m \equiv -\mathrm{d}S_m/\mathrm{d}m$. We assume ϕ and hence μ will be constant, or at least relatively constant. Initially, $f = \phi$ and $\eta_m = \mu$, but f and η_m decrease exponentially after the entropy has been squeezed out of the network ($\mu m > S_0$). Figure 16 shows the entropy and error rate for $S_0 = 50$, $\mu = 1/3$. Changing S_0 would rigidly shift the curves horizontally, and changing μ would just rescale the horizontal axis by a uniform factor.

17. Associative memory and clustering

The ability of massively parallel networks to perform associations, and to categorize the inputs into clusters, has received enormous amounts of attention—for a review, see reference [32]. Sometimes people get carried away and take associative memory as the definition of what a network ought to do, or even as the definition of computation[10] in general. We take the opposite view,

[10]The Turing machine, although not a practical device, is widely used as a formal, theoretical model of "computation in general." An associative memory is clearly less

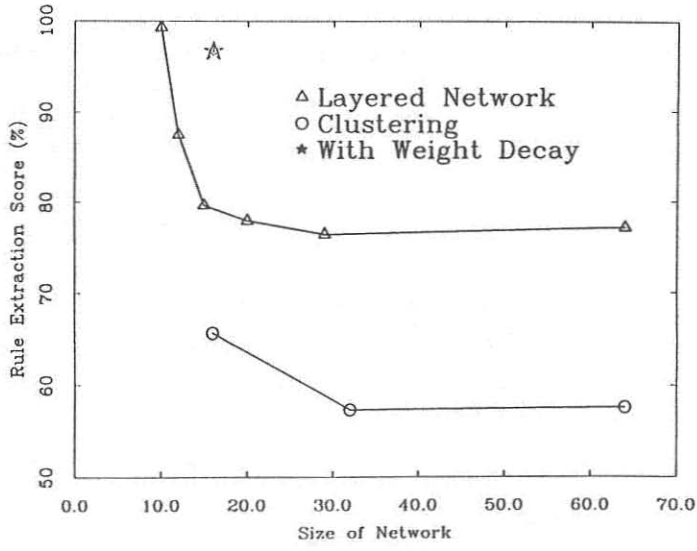

Figure 12: Rule extraction under various conditions.

namely that clustering is a particular case of rule extraction in which the rule is (roughly) of the form "nearby inputs should produce nearby outputs." The definition of "nearness" depends on the specific task in question.

A general learning machine can learn to do clustering, but the reverse is *not* true: there are huge categories of tasks a specialized cluster analysis machine cannot do. At worst, the general machine's task can be discussed in two phases: first the network discovers that clustering is appropriate, and then it learns where the cluster boundaries should be. A cluster analysis machine has less work to do, since it already "knows" that clustering is the way to go. There are a number of quite efficient clustering methods ([21] and references therein), and they should be used whenever they are appropriate. More general rule extraction procedures must be used in the remaining cases.

The upper trace figure 17 shows the rule extraction score for a very simple layered network—no particular effort was made to optimize its performance. The lower trace shows the analogous results for the most powerful clustering analysis we could think of, as described in appendix A. The layered network was clearly superior in this case. The figure also displays the data from table

powerful than a Turing machine; for starters, it cannot perform loops.

4, showing the importance of stochastic weight decay.

18. Symmetry, easiness, and programming

At first glance, the amount of data required for learning, as indicated in table 4, seems rather large. Humans seem to "discover the rule" using far fewer training examples. To understand this phenomenon, consider the following modification of the task: let us permute the bits of the input pattern, leaving unchanged the numerical value of the output. (There is only one permutation, applied to all patterns.) An example of this is shown in table 5, in which the permutation (0123456789) → (3120459786) was applied. This is a very simple permutation (two pairs are exchanged: 0 with 3, and 6 with 9), but it has a drastic effect on the appearance of the input patterns.

Old Input	Output	Permuted Input
fffttffff	F	tfffftffff
fffftftfff	T	tfffftftfff
ftttttttt	F	tttftttttt
tttfftfft	T	ftttftttff
fffffffff	F	fffffffff

Table 5: Permutation of the input.

This change has a similarly drastic effect on the rate at which humans learn the rule. First of all, the humans must guess that the task requires treating the input as a geometric pattern, which is not an obvious assumption—for all they know, the task might require treating the inputs as binary numbers and factoring them! Even if they guess that permuted geometry is important, they must guess what particular permutation has been applied and then discover the clump-counting or edge-counting rule. The humans would require a stupendous number of examples in order to achieve this.

Permutation of the input bits is an exact symmetry of the architecture; that is, the first step in the operation of the network is a calculation of the form

$$\sum_i W_{ji} I_i \tag{18.1}$$

in which the symbol i is a dummy index. This is in addition to the similar symmetry of the hidden units, described above.

Humans have a very strong prejudice in favor of geometric solutions. They consider the original task easy and the permuted task hard. The network has no such prejudice. It has no built-in notion of geometry or topology. For the network, the original task is just as difficult as the permuted task.

We feel that this difficulty is artificial. It arises when one asks an overgeneral network to solve an under-specified task. In real-world situations, the data does not exist in a vacuum, but exists together with important

ancillary information such as symmetry, geometry, topology, and so forth. When we presented the `two-or-more clumps` predicate back in section 3, we presented the topological information to the reader, along with the data. Throwing away the ancillary information makes the problem more abstract and certainly makes it less tractable. Why should we ask the network to solve a very hard general task, when all we cared about was a rather easier special task?

Learning from examples and related techniques will never replace programming—they will supplement it. Someone who understands a task will always do better than someone who does not. It is important to realize that search techniques are useful when you have an *intermediate* amount of knowledge about a task. There are some tasks (such as finding the minimum of a parabola) that are so well understood that the answer is obvious, or obtainable by by conventional analysis. At the other extreme, in the case of a truly random function, sophisticated procedures will not do any better than simple procedures. (You use grandmother cells to memorize the M data, and guess at the rest.)

In an artificial example such as the `three-or-more clumps` predicate, it is a question of taste as to what constitutes "giving ancillary information" and what constitutes "giving away the whole answer." Real-world tasks are so much more complicated that giving all available ancillary information still leaves plenty of work for the network to do.

Having decided to provide ancillary information to the network, we need sensible techniques for doing so. These techniques can be thought of as a strange sort of "programming language" for networks. One way is to change the architecture of the network, restricting the "receptive field" of each of the input units. For example, we could require that

$$W_{ij} = 0 \qquad \text{whenever } |i - j| > \rho \tag{18.2}$$

where ρ represents the radius of the receptive field.

A gentler and more general way to implement limited receptive fields would be to add to the E function a term of the form

$$E_3 = \lambda_3 \sum W_{ji}^2 K(|i - j|, \rho). \tag{18.3}$$

In the case where the multiplier λ_3 is large and the kernel K is a suitable step function, this method becomes effectively identical to the previous restriction. The idea of "programming" the network by adding terms to the E function is traceable to the traveling salesman network paper of Hopfield and Tank [15].

Either of these methods provide hints that the solution should be sparse, topologically one-dimensional, and local along that dimension. They break the permutation symmetry of i. To say it another way, we give the network a notion of neighborhoods (each containing $\approx 2\rho$ input bits), and these neighborhoods induce a topology on the input space.

The work on "feature spin" systems [24,3,4,21] used carefully constructed input and output spaces to perform automatic clustering. Hinton [43] used a network with three layers, one of which had one-dimensional limited receptive fields, to good advantage. Our studies of the representations found by networks when given such hints will be reported elsewhere [34].

A different type of ancillary information can be provided. We notice that the solution presented in section 6 has a high degree of translational invariance. It is easy to concoct a term analogous to equation 18.3 that penalizes solutions that are not translationally invariant. It is also possible to impose a rigorous restriction, analogous to equation 18.2, but one must be careful since the solution is not *exactly* invariant.

The human prejudice in favor of geometric solutions is not an accident; it is the result of thousands of centuries of accumulated information about the world. For any particular geometric task, there may be a very efficient *ad hoc*, non-geometric representation, which has nothing in common with the solution to any other task. The geometric solution is useful for many, many tasks. In order to test this idea, we are checking the stability of the human solution when a network with several output units is required to perform the combined `3/4/5/6/...` `clumps` task [34].

19. NERFs, regularization, and curve fitting

Once again, let us consider a collection of data and assume we have a reason to believe that it adheres to some simple rule. For instance, the data might be obtained by measuring some very simple physical system. This does *not* give us any reason to believe that the data can be represented efficiently by a network (i.e. by a network with $H \ll 2^N$ hidden units). What's more, in cases where a representation exists, the solution may well not be unique, and (as discussed back in section 8) the learning process may be NP-complete, in which case there is no general automatic learning procedure that will find the solution appreciably faster than an exhaustive search of weight space.

It is quite important to reconcile these sobering points with the observation that automatic learning procedures in general, and layered networks in particular, are capable of doing real-world tasks, sometimes astonishingly well (see examples and references listed in section 2). It is also imperative that we understand what the limitations are. Fortunately, there is a powerful analogy that sheds considerable light on this situation, namely the analogy to curve fitting.

Consider the task of fitting a smooth curve through the four data points shown in figure 19a. One way to do this would be to minimize the error function

$$E = \sum_i \frac{|y_i - f(x_i)|^2}{\sigma_i^2} \tag{19.1}$$

where (x_i, y_i) are the experimental data points, σ_i is the uncertainty in y_i, and $f()$ is the theoretical function (actually a family of functions, depend-

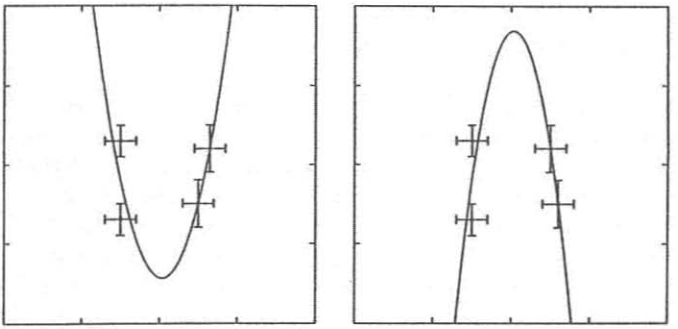

Figure 13: Curve fitting.

ing on some adjustable parameters) to be fitted so we can extrapolate and interpolate over some interval D. For concreteness, let us consider the case where $f()$ is restricted to be a parabola. This is the most elementary form of curve fitting.

The resulting fit function $f()$ can be considered a generalization of the data points (x_i, y_i), in the sense of extending the domain. The data can be considered a function (or relation) on a very small domain—just the isolated points $\{x_i\}$. The function $f()$ is defined on the whole interval D. That is a fantastic degree of generalization, a huge extension of the domain.

Curve fitting is also generalization in the sense of averaging noisy data. The data could be quite unsmooth, yet a low-degree polynomial that fits the data cannot possibly have very much curvature on the interval D.

Curve fitting can, however, be very tricky. In figure 19, the data has uncertainties in x_i as well as y_i. Only small horizontal shifts, much less than the length of an error bar, would be needed to make the four points form two pairs, each pair having a single abscissa. In that case, there would be an infinite number of parabolas that could be fit through the points, all having the same E value or "chi-square" or "goodness of fit". Also, when the data is *nearly* paired, small changes in the data can lead to arbitrarily large changes in the parameters of the "best" parabola, as illustrated by going from figure 19a to figure 19b. This is a double-purpose example of what Hadamard [12] classified as ill-posed or ill-conditioned problems; that is, the solution is not uniquely determined by the data, or the solution depends discontinuously on the data.

Tikhonov [38] showed how to deal with such problems; in this case, we just need to remember that what we originally wanted was a smooth curve through the data, i.e. a function with low curvature. This is expressed in the

following formula by the second term, which explicitly and precisely penalizes functions that have a large curvature.

$$E = \sum_i \frac{|y_i - f(x_i)|^2}{\sigma_i^2} + \lambda \int_D |\frac{d^2 f}{dr^2}|^2 \tag{19.2}$$

Of course, this is just an example of a possible Tikhonov regularization term. One chooses a penalty term that is appropriate to the actual task. The idea is that the penalty term expresses an estimate of the *a priori* implausibility of each possible solution $f()$. There is a vast literature on maximum-likelihood estimation—see Press et al. [31]. and references therein.

Note that we have invoked two subtly different ways of limiting the curvature. One is what we call *structural* stabilization, in which we explicitly limit the degree of the polynomial so that it cannot have much curvature. The second case we call *formal* stabilization, such as equation 19.2, in which it is not really necessary to require that $f()$ have a small degree. The Tikhonov regularization term will automatically select a function of low curvature, whether or not it can be expressed as a low-order polynomial.

We emphasize that neither stabilization by structural restrictions nor formal, Tikhonov-style regularization is the exclusive, universal solution. The latter is more elegant, and it provides a mathematical language for discussing things that would otherwise be "teleological and anthropomorphic", as George Furnas put it [44]. On the other hand, there are important practical reasons why structural restrictions are often more appropriate; for instance, they can greatly reduce the number of free parameters, making the search more efficient and making the answer more readily describable and understandable.

Practically all of these ideas can be applied equally well to layered networks. Just as a polynomial with high enough degree can closely approximate most reasonable functions, a network with enough hidden units can represent any Boolean function. Low-degree polynomials can only represent relatively smooth functions, and small networks can only represent a certain class of relatively simple Boolean functions. We call these particular functions *network efficiently representable functions*, or NERFs.

If the training data is noisy, and only defined on a sparse set of points, fitting a NERF to it will average out the noise and provide us an extension that covers the entire input space. If the data is inadequate to specify a unique NERF, we can add to the optimization equation a stabilization term such as in equation 18.3, which is quite analogous to the second term in equation 19.2. It expresses the designer's preference among solutions that would otherwise have equivalent E-functions.

The question immediately arises: what is so special about NERFs? For that matter, what is so special about polynomials?

Polynomials are special for several reasons. Perhaps the main reason is that polynomials occur in nature. To a good approximation, the path of a thrown stone draws a parabola in the sky. A second reason is that polynomials are reasonably easy to evaluate by elementary methods. Third, a

polynomial has a small number of adjustable parameters, so even if the fit is purely phenomenological and fortuitous, it is a handy way of describing the data. Finally, people just have a lot of experience dealing with polynomials and can readily visualize what they look like and how (as the degree is increased) they come to resemble other functions.

NERFs have some analogous virtues. It is hard to say just how common NERFs are in nature, but it is not easy to concoct a meaningful, natural function that is not a NERF. This is in contrast to single-layer perceptrons, which Minsky and Papert [27] showed could not solve the `parity` problem, `T/C` problem, or any of various connectedness problems. For the N-layer networks, which are much more powerful, it would be nice to have an analogous *pons asinorum* which the network cannot solve, as well as a simple analytic concept (analogous to linear separability) to help us understand the limitations.

It might be said the NERFs exist in nature in another, quite different sense: recall that the reason that this family of functions came originally to be considered is that they seem to be a modest model of the computational operations that are implemented by the neural circuits in the brain.

As for the second virtue, there is no doubt that NERFs are especially easy to evaluate. Analog integrated circuits have been built which perform the required products and sums using far less time and space than would be required for other, seemingly similar, functions [17,18].

As for the third and fourth virtues, NERFs are not widely used for phenomenological data reduction, precisely because most people do not have a keen intuitive understanding of what they look like. To a certain extent, this defines the present and future task of everyone in this field: our job is to understand NERFs, until they become as familiar as polynomials.

Of course, neither polynomials nor NERFs are a panacea. Suppose that rather than the points in figure 19, the data consisted of a thousand points with very small error bars, closely fitting ten cycles of a sine wave. It would be insane to fit that data with a high-degree polynomial; it would be much more sensible to use a low order Fourier series. The set of low-order Fourier series and the set of low-degree polynomials are both subsets of the set of all smooth functions. Any given smooth function might belong to one, or the other, or neither of these subsets.

Similarly, the set of low-order NERFs is an important but still limited subset of the set of all "reasonable" Boolean functions. Surely for some applications we must turn to other families of functions to smooth and extrapolate our data. Various extensions to the basic network models, such as sigma-pi networks and higher-order terms (numerous references in [32]; also [7]) are analogous to the way in which rational functions are an extension of the family of polynomials.

20. Summary

A generalization is a function that *extends the domain* of the training data. This is very useful for *averaging noisy data*. Practically all of the problems (and the power) of this sort of generalization can be understood by comparing it with the well-studied discipline of curve fitting.

Given enough training data, the generalization will be unique. More commonly, though, there are an enormous number of perfectly valid generalizations, i.e. functions that are perfectly consistent with the training data. The amount of data needed to determine the result allows us to define a measure of efficiency with which the network extracts information from its training data. Giving the network too many resources (e.g. too many hidden units) increases the initial entropy and hinders rule extraction.

Rule extraction involves comparing the generalization found by the network with some prechosen rule. The distinction between the memorization set and the extraction set must be scrupulously respected. The language of regularization theory is important because it allows us to speak quantitatively of "good" generalizations and "bad" generalizations.

A layered network is capable of extracting the `two-or-more clumps` predicate from the training data. Indeed, it extracts it with rather high efficiency. The internal representation that it uses does not have any discernible degree of symmetry, sparseness, or locality; the symmetric, low-order, local solution that humans prefer is of course a fixed point of the training process, but is not stable against perturbations.

A general, *tabula rasa* network is a fine subject for abstract, formal studies, but one should not try to use it to solve practical problems. Automatic learning will never replace programming—it will *supplement* it. One should pre-program the network with all available information about the structure of the problem, especially information about the symmetry and topology of the data.

21. Acknowledgements

We thank Arno Penzias for calling our attention to the `two-or-more clumps` predicate and the questions of generalization and "human" representation that it raises. Our preliminary results were intensively discussed on the computerized bulletin board "connectionists@c.cs.cmd.edu" created by David Touretzky. Contributors to this discussion include Alexis Wieland, Christof Koch, David J. Burr, Terry Sejnowski, Thomas Maxwell, Scott E. Fahlman, Geoff Hinton, George W. Furnas, Jerry Feldman, Steven Nowlan, and Richard Golden. We are particularly indebted to David Burr for providing us with relevant "back issues" of the bulletin board.

We thank Lee Giles for making available the results of his group [25] prior to publication. It is a pleasure to acknowledge important discussions with Bill Jeffrey. Our friend and colleague Erik deBennedictis created the hypercube hardware and software that was indispensible to our work.

Some of us carried out part of this work at the Santa Fe Institute (JSD) and at the Institute for Theoretical Physics, UC Santa Barbara (JSD and SAS). We are grateful to these institutes and their participants for their hospitality, support and interest.

Appendix A. Conditions of the experiments

The data in table 3 was taken using the LMD objective function [40]. The predicate was `2 versus 3 clumps`. The memorization set was exhaustive, using complete smoothing. The logical levels were $T = 1$ and $F = -1$, and the convergence criterion was 100 percent categorically correct (i.e. closer to the right target than the wrong target). The initial distribution of weights was uniform on the interval $[-.2, .2]$. Each data point is the median of an ensemble of fifteen trials. All hidden units in all the trials reported in this paper were fully connected.

The data in figure 9 was taken using the LMS objective function. The predicate was `1 versus 2 clumps`. The memorization set included 40 items (20 of each class), and there was extreme incomplete smoothing—one by one. The logic levels were $T = 1$ and $F = 0$, and the convergence criterion was RMS error $\leq .005$. The initial distribution of weights was Gaussian, with unit variance. Each data point is the median of an ensemble of ten trials.

For figures 12 and 12, the memorization data was the same as in the previous paragraph, except that the predicate was `2 versus 3 clumps`, and the size of the memorization set was varied as indicated in the figure (always half in each class). The extraction set consisted of all the remaining patterns, so the rule extraction gain ratio $(1 - \epsilon)/\epsilon$ is a strong function of N, and the extraction set was not half in each class. Note: the number of patterns with C clumps is $\jmath\binom{N+1}{2C}$. The criterion for correctness during the testing phase was being with .2 of the target.

In figure 17, the predicate was `2 versus 3 clumps`. The memorization set included $50N$ items and the extraction set consisted of another $50N$ patterns (half in each class). The data in the upper curve was taken using the LMS objective function, using the generalized delta method with incomplete smoothing (one by one). The criterion for correctness during the testing phase was being with .2 of the target. The high isolated point represents the data from table 4, which was taken under the same conditions, with the addition of stochastic weight decay, according to the following scheme: For each weight, with probability P, multiply by $(1 - \alpha^K)$, where K is the number of completed passes through the data, and in this case $P = 1/256$ and $\alpha = 0.1$.

The lower curve in figure 17 was generated as follows: All elements of the M set were used as "prototypes", i.e. the centers of clusters. Each element of the X set was compared with each of the prototypes, and a histogram of distance was made. All prototypes in the minimal bin (closest distance) voted on what class the X element should be in.

References

[1] Y. Abu-Mostafa, article in *Complexity in Information Theory*, Y. Abu-Mostafa, ed., (Springer-Verlag, New York, 1986); see also article in *Neural Networks for Computing, (Snowbird 1986)*, AIP Conference Proceedings #151, (American Institute of Physics, New York, 1986).

[2] Dana Angluin and Carl Smith, *Computing Surveys*, **15:3** (1983) 237.

[3] David J. Burr, "A dynamic model of image registration", *Computer Graphics and Image Processing*, **15** (1981) 102.

[4] David J. Burr, "Matching elastic templates", in *Proceedings of the Royal Society Symposium on Physical and Biological Processing of Images*, (Springer Verlag, 1982).

[5] David J. Burr, "A Neural Network Digit Recognizer", *Proceedings of the IEEE Conference on Systems, Man, and Cybernetics*, (Atlanta, 1986) 1621.

[6] David J. Burr, "Experiments with a Connectionist Text Reader", to appear in *Proceedings of the First International Conference on Neural Networks*, (San Diego, 1987).

[7] H. H. Chen, Y. C. Lee, G. Z. Sun, H. Y. Lee, Tom Maxwell, and C. Lee Giles, in *Neural Networks for Computing, (Snowbird 1986)*, AIP Conference Proceedings #151, (American Institute of Physics, New York, 1986).

[8] John S. Denker, ed., *Neural Networks for Computing, (Snowbird 1986)*, AIP Conference Proceedings #151, (American Institute of Physics, New York, 1986).

[9] Michael R. Garey and David S. Johnson, *Computers and Intractability—A Guide to the Theory of NP-Completeness*, (W. H. Freeman, New York, 1979).

[10] P. Gorman and T. H. Sejnowski, article in JPL Document #D-4406, (1987) 224–237.

[11] H. P. Graf, W. Hubbard, and L. D. Jackel, to be published.

[12] J. Hadamard, *Lectures on the Cauchy Problem in Linear Partial Differential Equations*, (Yale University Press, 1923).

[13] G. E. Hinton and T. J. Sejnowski, "Learning and Relearning in Boltzmann Machines", in *Parallel Distributed Processing—Explorations in the Microstructure of Cognition*, (MIT Press, 1986).

[14] J. H. Holland, K. J. Holyoak, R. E. Nisbett, and P. R. Thagard, *Induction*, (MIT Press, Cambridge, MA, 1986).

[15] J. J. Hopfield and D. W. Tank, *Biological Cybernetics*, **52** (1985) 141.

[16] J. J. Hopfield and D. W. Tank, to appear in *Proceedings of the IEEE International Conference on Neural Networks*, (San Diego, 1987).

[17] R. E. Howard, *Proceedings of the 13th European Solid State Circuits Conference*, (1987).

[18] L. D. Jackel, H. P. Graf, and R. E. Howard, "Electronic Neural Network Chips", *Applied Optics* (1987).

[19] Jeffrey W. and R. Rosner, *Astrophysics J.*, **310** (1986) 473; see also article in *Neural Networks for Computing, (Snowbird 1986)*, AIP Conference Proceedings #151, (American Institute of Physics, New York, 1986).

[20] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing", *Science*, **220** (1983) 671.

[21] Teuvo Kohonen, *Self-Organization and Associative Memory*, (Springer-Verlag, New York, 1984).

[22] A. N. Kolmogorov, "Logical basis for information theory and probability theory", *IEEE Transactions on Information Theory*, **IT-14** (1968) 662.

[23] Richard P. Lippmann, *IEEE ASSP*, 4 (1987).

[24] Christof von der Malsburg, "Self-organizing of orientation sensitive cells in the striate cortex", *Kybernetik*, **14**, (19873) 85–100.

[25] Tom Maxwell, C. Lee Giles, and Y. C. Lee, "Generalization in Neural Networks: The Contiguity Problem", to appear in the *Proceeding of the IEEE International Conference on Neural Networks* (San Diego, 1987).

[26] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, (Addison Wesley, Reading, MA, 1980).

[27] Marvin Minsky and Seymour Papert, *Perceptrons*, (MIT Press, Cambridge, MA, 1969).

[28] Edward W. Packel and J. F. Traub, "Information-Based Complexity", *Nature*, **328** (1987) 29.

[29] David Parker, to appear in the *Proceedings of the 1987 AAAI Conference.*

[30] R. W. Prager, T. D. Harrison, and F. Fallside, *Computer Speech and Language*, **1**, (1986) 3.

[31] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, *Numerical Recipes—The Art of Scientific Computing*, (Cambridge University Press, 1986).

[32] David E. Rumelhart and James E. McClelland, *Parallel Distributed Processing—Explorations in the Microstructure of Cognition*, (MIT Press, 1986).

[33] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation" in *Parallel Distributed Processing—Explorations in the Microstructure of Cognition*, (MIT Press, 1986).

[34] S. Ryckebusch, S. A. Solla, J. S. Denker, B. S. Wittner, D. B. Schwartz, to be published.

[35] R. Scalettar and A. Zee, ITP preprint (1987).

[36] Terry J. Sejnowski, P. K. Kienker, and G. E. Hinton, "Learning symmetry groups with hidden units: beyond the perceptron", *Physica*, **D22** (1986).

[37] Terry J. Sejnowski and C. R. Rosenberg, "NETtalk: a parallel network that learns to read aloud," Johns Hopkins technical report JHU/ EECS-86/01 (1986).

[38] A. N. Tikhonov and V. Y. Arsenin, *Solutions of Ill-Posed Problems*, (Wiley, New York, 1977).

[39] Bernard Widrow and Samuel D. Stearns, *Adaptive Signal Processing*, (Prentice-Hall, Englewood Cliffs, NJ, 1985).

[40] B. S. Wittner, J. S. Denker, et al., to be published.

[41] Yann LeCun, private communication.

[42] Terry J. Sejnowski, private communication.

[43] Geoffrey Hinton, private communication.

[44] George Furnas, private communication.