4 MLP Representational Capabilities

The standard multilayer perceptron (MLP) is a cascade of single-layer perceptrons (figure 4.1). There is a layer of input nodes, a layer of output nodes, and one or more intermediate layers. The interior layers are sometimes called "hidden layers" because they are not directly observable from the system inputs and outputs. Each node has a response $f(\mathbf{w}^T\mathbf{x})$ where $\mathbf{x}$ is the vector of output activations from the preceding layer, $\mathbf{w}$ is a vector of weights, and f is a bounded nondecreasing nonlinear function such as the sigmoid. Normally, one of the weights acts as a bias by virtue of connection to a constant input. Nodes in each layer are fully connected to nodes in the preceding and following layers. There are no connections between units in the same layer, connections from one layer back to a previous layer, or "shortcut" connections that skip over intermediate layers. Although back-propagation can be applied to more general networks, this is the most commonly used structure.

The following sections summarize some properties and limitations that result from this structure, independent of methods used to set the weights.

How to Count Layers? A minor digression: there is some disagreement about how to count layers in a network. Some say a network with one hidden layer is a three-layer network because there are three layers of nodes: the inputs, the hidden units, and the outputs. Others say this is a two-layer network because there are only two layers of active nodes, the hidden units and outputs. Inputs are excluded because they do no computation. We tend to follow this convention and say that an L-layer network has L active layers; that is, $L-1$ hidden layers and an output layer. Conveniently, this is also the number of weight layers. Not everyone uses the same convention, however, so it is often simplest to explicitly specify the number of hidden layers. The network in figure 4.1, for example, would be called a two-hidden-layer network. In spite of the convention, it is natural to refer to the input layer at times; we did so in the first paragraph of this chapter.

The notation $N_1/N_2/\cdots/N_L$ is sometimes used to describe the structure of a layered network. This is simply a list of the number of nodes in each layer. A $10/3/2$ network, for example, has 10 inputs, 3 nodes in a hidden layer, and 2 outputs. A $16/10/5/1$ network would have 16 inputs, 10 nodes in the first hidden layer, 5 nodes in the second hidden layer, and 1 output. Unless otherwise specified, each layer is presumed to be fully connected to the preceding and following layers with no short-cut or feedback connections. Figure 4.1 illustrates a $5/5/3/4$ structure.

4.1 Representational Capability

The representational capability of a network can be defined as the range of mappings it can implement when the weights are varied. A particular mapping is within the representational

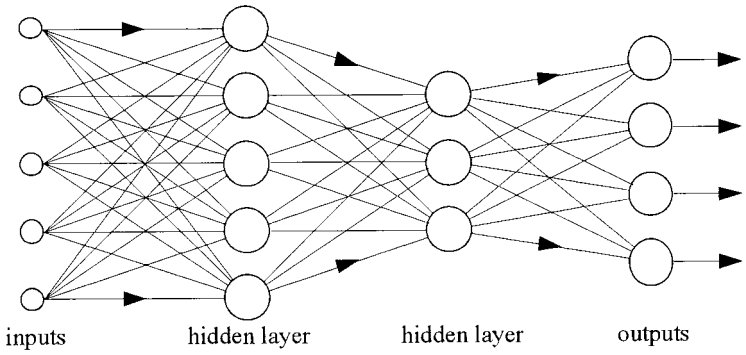

Figure 4.1
MLP structure. A multilayer perceptron is a cascade of single-layer perceptrons. There is a layer of input nodes and a layer of output nodes with one or more hidden layers in between.

capability of a net if there is a set of weights for which the net performs the mapping. Theories about representational capability answer questions like: Given a set of inputs and associated target outputs, is there a set of weights that allow the network to perform the desired mapping? Is the network capable of generating the desired outputs from the given inputs? In some cases, an exact solution may be required while in others, an approximation may be allowed. Results say whether a particular problem might be solvable by a particular network without necessarily producing the weights that generate the solution. If the answer is negative, we know the network cannot solve the problem and this saves us the expense of a futile search. If the answer is positive, we know a solution exists but these results don't guarantee that it will be easy to find or that a chosen optimization method will be able to find it. Results of this kind provide broad guidelines in selecting an architecture for a problem. One well-known representational result, for example, is that single-layer networks are capable of representing only linearly separable functions.

Two Hidden Layers Are Sufficient When designing a layered network, an obvious first question is how many layers to use. Lippmann [247] shows that two hidden layers are sufficient to create classification regions of any desired shape. In figure 4.2, linear threshold units in the first hidden layer divide the input space into half-spaces with hyperplanes, units in the second hidden layer AND (form intersections of) these half-spaces to produce convex regions, and the output units OR (form unions of) the convex regions into arbitrary, possibly unconnected, shapes. Given a sufficient number of units, a network can be formed that divides the input space in any way desired, producing 0 when the input is in one region and 1 when it is in the other. The boundaries are piecewise linear, but any smooth boundary can be approximated with enough units.

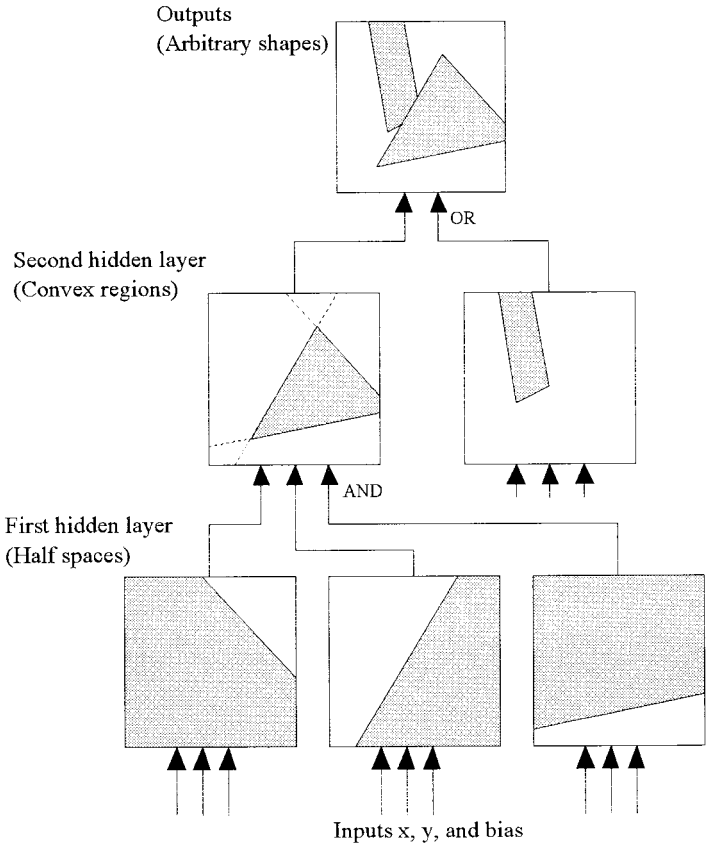

Figure 4.2
Three layers of linear threshold units are sufficient to define arbitrary regions. Units in the first hidden layer divide the input space with hyperplanes, units in the second hidden layer can form convex regions bounded by these hyperplanes, and output units can combine the regions defined by the second layer into arbitrarily shaped, possibly unconnected, regions. Here, the boundaries are piecewise linear, but any smooth boundary can be approximated with enough units (based on [247].)

To approximate continuous functions, one can add and subtract (rather than logically OR) convex regions with appropriate weighting factors so two hidden layers are also sufficient to approximate any desired bounded continuous function [234].

One-hidden-layer Nets Can Represent Nonconvex, Disjoint Regions Units in the second hidden layer of figure 4.2 respond to convex regions in the input space. Because these would be the outputs in a single-hidden-layer network, it is sometimes mistakenly assumed that single-hidden-layer networks can recognize only convex decision regions.

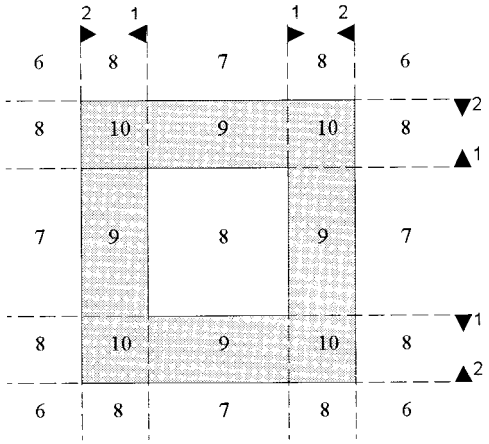

Figure 4.3
Single-hidden-layer networks can recognize nonconvex decision regions. Here, eight hidden units in a single layer create a nonconvex "square donut." Each dashed line is the decision boundary of one of the hidden units. The hidden unit is active (1) on the side of the line indicated by the pointer and connects to the output unit with the weight indicated next to the pointer. The other numbers show the summed input to the output unit in different regions. Thresholding at 8.5 creates the square donut.

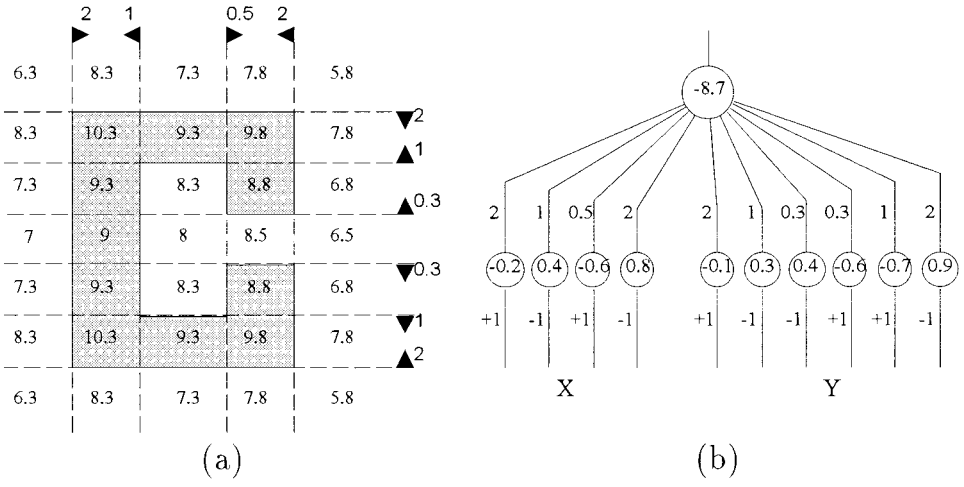

Figure 4.4
Weiland and Leighton [407] provide this example of a nonconvex region recognized by a single-hidden-layer network. Thresholding at 8.7 creates the 'C' shaped region: (a) a nonconvex decision region, and (b) the network. There are 10 hidden nodes. Each has one connection with weight $w = \pm 1$ from either the x or y input. Numbers inside the circles are the node thresholds.

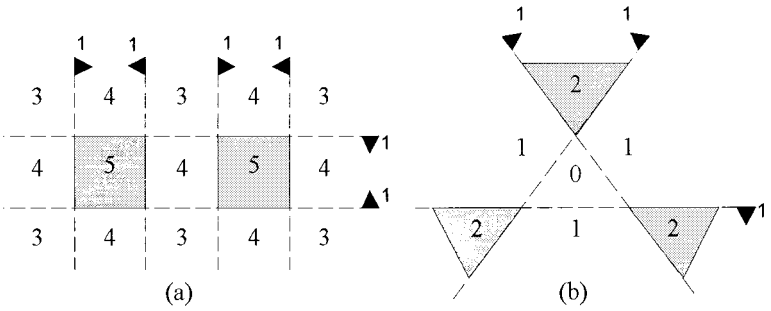

Figure 4.5
Disjoint regions recognized by single-hidden-layer networks. In addition to nonconvex regions, single-hidden-layer networks can also recognize disjoint (unconnected) regions.

Counter-examples are shown in figures 4.3 through 4.5. Figures 4.3 and 4.4 illustrate nonconvex regions recognized by three-layer nets. (Figure 4.4 is derived from Weiland and Leighton [407].) Figure 4.5 shows examples of disjoint (unconnected) regions recognizable by three-layer networks. Other examples can be found in [180, 255, 249].

4.2 Universal Approximation Capabilities

The examples above suggest that any bounded function can be approximated with arbitrary accuracy if enough hidden units are available. These ideas have been extended theoretically to show that multilayer perceptrons are *universal approximators*. That is, they are capable of arbitrarily accurate approximation of essentially arbitrary continuous mappings from the $[-1, +1]^n$ hypercube to the $(-1, 1)$ interval. This is an important result because it says neural networks, as a class, are powerful enough to implement essentially any function we require.

4.2.1 Kolmogorov's Theorem, One Hidden Layer Is Sufficient

A somewhat surprising result is that two hidden layers are not necessary for universal approximation; one hidden layer is sufficient. Kolmogorov [219, 220] showed that a continuous function of several variables can be represented *exactly* by the superposition of continuous one-dimensional functions of the original input variables. A refined proof by Sprecher [355] is often cited. Hecht-Nielsen [161] introduced the result to the neural network community and showed an implementation in the form of a single-hidden-layer network. Briefly, the result is that any continuous function mapping an n-dimensional input, $n \geq 2$, to an m-dimensional output can be implemented exactly by a network with one

hidden layer. For $\phi : \mathbf{I}^n \rightarrow \mathbf{R}^m$, $\phi(\mathbf{x}) = \mathbf{y}$, where $\mathbf{I}$ is the closed interval [0, 1] and therefore $\mathbf{I}^n$ is an n-dimensional hypercube, the function ϕ can be implemented *exactly* by a single-hidden-layer neural network having n elements in the input layer, $(2n + 1)$ elements in the middle layer, and m elements in the output layer. That is,

$$z_k = \sum_{j=1}^{n} \lambda^k \psi(x_j + \epsilon k) + k \tag{4.1}$$

$$y_i = \sum_{k=1}^{2n+1} g_i(z_k), \tag{4.2}$$

where λ is a real constant, ψ is a continuous real monotonic increasing function independent of ϕ but dependent on n, and ϵ is a rational number, $0 < \epsilon \leq \delta$, where δ is an arbitrarily chosen positive constant. The output node functions g_i are real and continuous and depend on ϕ and ϵ. Unfortunately, the proof is not constructive; it does not say how weights should be chosen to implement a particular mapping or specify the functions g_i (different for each i), which have unknown and often extremely nonlinear shapes.

Girosi and Poggio [139] point out that the theorem might not be relevant because the inner functions ψ are extremely nonsmooth. (The theorem fails when restricted to smooth functions.) The functions are very unlike the sigmoids normally used in neural networks and not likely to be learnable because of their extreme roughness. Further, the functions g_k depend on ϕ, are all different, and do not form a parameterized family. They are likely to be at least as complex as ϕ, if not more so, and therefore no easier to learn than ϕ itself. In effect, the complexity of approximating the original function is shifted into the task of finding the internal node functions.

In more recent work, Sprecher [356] describes a stronger version of the theorem using translates of a single function in place of the multiple internal functions g_k. The replacement function remains nonsmooth, however, so the objections remain.

Kurkova [231] notes that although exact representation is not possible with smooth internal functions, approximation is, and goes on to show universal approximation properties of sigmoidal networks with two hidden layers. But other work [244] suggests that approximation of the internal functions does not yield approximation of the target function.

4.2.2 Other Proofs of Universal Approximation Capability

Several proofs are based on showing that single-hidden-layer networks can implement Fourier transforms and thus have the same approximation capabilities. The idea is that a sinusoid can be implemented by sums and differences of many shifted sigmoids, one for each half cycle. Irie and Miyake [186] give a proof based on Fourier integrals where the

number of nodes required for exact approximation is infinite. Gallant and White [132] give another Fourier transform proof using monotone "cosine-squasher" sigmoids. Funahashi [131] approximates the integral representation of Irie and Miyake with a discrete sum and shows that networks with at least one hidden layer of sigmoid units are universal approximators.

Cybenko [95, 96] provided one of the first rigorous proofs of universal approximation capability, showing that a network with linear output units and a single hidden layer of sigmoid units is sufficient for uniform approximation of any continuous function in the unit hypercube. Hornik, Stinchcombe, and White [177] and Stinchcombe and White [358] show that standard multilayer networks using arbitrary squashing functions are universal approximators; single-hidden-layer nets are included as a special case. Approximation of a function and its derivatives using arbitrary bounded nonconstant squashing functions is discussed by Hornik [175]. Other work on approximation includes [162, 48, 188, 226, 176].

In most cases, these are existence proofs rather than constructive recipes for finding a solution to a particular problem. These results do not guarantee that a chosen optimization method will find the necessary weights or that the solution found will be efficient. A constructive proof based on the Radon transform is given by Carroll and Dickson [64].

It should be noted that universal approximation is not a rare property. Many other systems have similar capabilities: polynomials, trigonometric polynomials (e.g., Fourier series), kernel regression systems, wavelets, and so on. By itself, this property does not make neural networks special. The results are important because they show that neural networks are powerful enough to approximate most functions that people find interesting. The lack of a universal approximation capability, on the other hand, would be bad news; neural networks would then be too weak for many problems and therefore much less appealing.

Given that there are many universal approximation systems, the choice of one over another depends on other factors such as efficiency, robustness, and so on. Barron [21, 22, 23] addresses the problem of how the MLP approximation error scales with the number of training samples and the number of parameters. One important result is that the error decreases like $O(1/\sqrt{N})$ as the number of training samples N increases. Another result is that the error decreases like $O(1/M)$ as a function of M, the number of hidden nodes. Unlike other systems (e.g., polynomials), this is independent of the input dimension and appears to avoid the "curse of dimensionality" (see [317: 178] for a skeptical view, however). These results can be used to put bounds on the necessary number of hidden nodes (in one hidden layer) and provide another justification for the rule of thumb that the number of training samples should be larger than the number of parameters divided by the desired approximation error, $N > O(Mp/\epsilon)$. Here N is the number of samples, M is the number of hidden nodes, p is the input dimension (so Mp is approximately the number of parameters), and ϵ is the desired approximation error.

One Hidden Layer Is Not Always Enough The limits of these proofs are sometimes forgotten. They say that a sufficiently large one-hidden-layer network is capable of essentially arbitrarily accurate approximation of continuous functions over compact regions. Although this covers most functions we would like to approximate, it may not include them all. Sontag [350, 351] points out that there are certain functions, important in inverse control, which cannot be approximated by single-hidden-layer nets with any number of units but which can be implemented rather simply with two hidden layers. That is, there are control systems that can be stabilized by a two-hidden-layer network but not by a one-hidden-layer net. The difference depends not on the number of units needed to achieve a certain numerical accuracy but rather on the need to approximate discontinuous functions that may arise as one-sided inverses of continuous functions.

4.3 Size versus Depth

Since a single sufficiently large hidden layer is adequate for approximation of most functions, why would anyone ever use more? One reason hangs on the words "sufficiently large." Although a single hidden layer is optimal for some functions, there are others for which a single-hidden-layer solution is very inefficient compared to solutions with more layers.

Certain functions containing disjoint decision regions cannot be realized exactly with only a single hidden layer of threshold units [255]. Certain functions can be implemented exactly by small networks with two hidden layers but require an infinite number of nodes to approximate with a single hidden layer network [255, 254]. Figure 4.6 shows an example from [255]. Gibson and Cowan [137] provide another example. Chester [80] describes a "pinnacle" function (1 at the origin and 0 elsewhere) for which a single-hidden-layer network needs $O(1/\epsilon)$ nodes to achieve $O(\epsilon)$ maximum error where a two-hidden-layer network needs only 4 nodes. This uses a maximum error criteria, however, rather than mean squared error. For certain functions, single-hidden-layer networks may require very large weights or extreme precision [180].

Although these results suggest that two-hidden-layer networks are more powerful, one-hidden-layer networks may be adequate for many functions encountered in practice. One- and two-hidden-layer networks are compared empirically by de Villiers and Barnard [104]. For fair comparison, the networks are configured to have approximately the same number of weights. Using conjugate gradient training and artificial clustered 2-dimensional data (mixtures of Gaussians), no significant difference was found in the best solution found by either architecture. The one-hidden-layer nets were reported to have lower average error and perhaps generalized better. For two-hidden-layer nets, the same authors report that

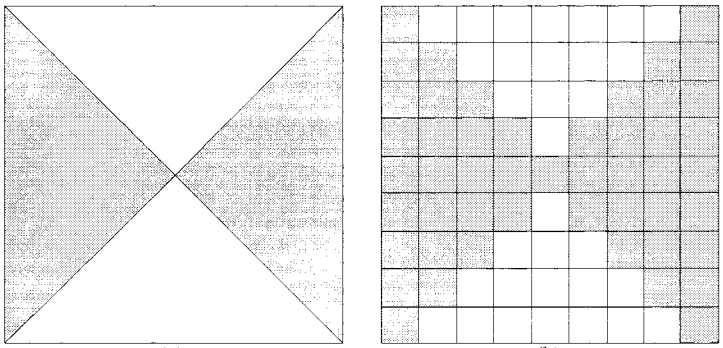

Figure 4.6
Certain functions can be implemented exactly by small networks with two hidden layers but require an infinite number of nodes to approximate if constrained to a single hidden layer [255, 254]: (a) a simple function that can be implemented with two hidden layers each containing two threshold units, (b) an approximation by a network with just one hidden layer, (from [255]).

training is easier if the hidden layers have approximately equal sizes. Similar results are reported by Huang and Lippmann [180]; no significant difference was seen in error rate or convergence time.

4.3.1 Size versus Depth for Boolean Functions

Many results exist for Boolean logic functions, e.g., [347]. (See figure 4.7 for the elementary gates.) It is obvious that a net with one hidden layer of 2^n threshold units can compute any Boolean function of n binary inputs since any Boolean function can be expressed in conjunctive (or disjunctive) normal form. Each hidden unit computes one of the 2^n product terms and the output unit ORs selected product terms to produce the desired result. An AND-OR array has this structure. This amounts to exhaustive enumeration of all positive cases and becomes impractical for large n.

More economical solutions can often be obtained by adding more layers. The number of nodes needed can be reduced by a factor of $O(\sqrt{n})$ because any Boolean function of n variables can be computed by a network with two hidden layers and $O(2^{n/2})$ threshold gates [347]. This is nearly optimal as an unbounded depth circuit must have size $O(2^{n/2}/\sqrt{n})$ [347: 125]. Notice that this is an exponential reduction in size.

Because a one-hidden-layer solution may be inefficient, there may be functions that cannot be implemented by a network of limited size using just one hidden layer. It is known that there are Boolean functions which cannot be computed by threshold networks with

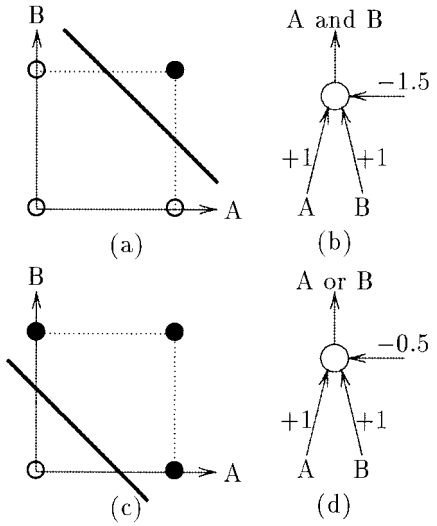

Figure 4.7
AND and OR logic functions. Any logic function can be implemented by a circuit of AND, OR, and NOT gates. Threshold units are at least as powerful as logic circuits because they can implement all the necessary elementary functions, as well as others: (a) the AND function, (b) linear threshold unit implementation of AND, (c) the OR function, and (d) linear threshold unit implementation of OR. The NOT function can be obtained from a single-input unit with a large negative weight and a zero threshold.

one hidden layer and a constant number of nodes, but which can be computed by threshold networks with two hidden layers and a constant number of nodes ([250]; cited in [99]).

"Symmetric" functions depend only on the sum of the inputs, $\sum_i x_i$; parity is one example. For arbitrary symmetric Boolean functions of n inputs, $O(n)$ units are needed in depth-2 (one hidden layer) networks of linear threshold units (LTU), but only $O(\sqrt{n})$ units are needed in depth-3 (two hidden layers) networks [348, 347]. For periodic symmetric functions such as parity, a depth/size tradeoff can be obtained at all depths. For parity(n), there is a depth($d+1$) LTU circuit of size $O(dn^{1/d})$ [347]. But increasing the depth beyond 3 does not decrease size much; $O(\sqrt{n/\log n})$ LTU gates are needed to compute general symmetric functions if there is no restriction on depth [347].

Another reason to use more than one hidden layer is to decrease the weight magnitudes. A one-hidden-layer implementation may require arbitrarily large weights. If permissible weight values are bounded (e.g., by hardware limitations), networks with more hidden layers may be more efficient. A depth-d circuit with exponential weights can be simulated by a depth-$(d+1)$ circuit with polynomially bounded weights at the cost of a polynomial increase in network size [347: 40]. A single linear threshold unit with exponential weights

can be exchanged for a depth-3 polynomial size circuit with polynomial weights [347: 41]. Anything computable by a depth-2 threshold circuit of polynomial size can be computed by depth-3 *small weight* threshold circuits of polynomial size ([143]; cited in [142]). Thus if the range of weights is limited, networks with more than one hidden layer may be able to realize functions that cannot be realized by single-hidden-layer networks.

Caveats The results just stated say there are functions for which single-hidden-layer networks are less efficient than networks with more hidden layers. Of course, there are still functions where small single-hidden-layer networks are optimal and additional hidden layers are not useful. Single-hidden-layer networks may need large numbers of nodes to compute arbitrary functions, but small networks may suffice for particular functions.

Many of these results are statements about the power of a class of networks rather than guarantees that a particular network will be able to learn a particular set of data and generalize accurately. Many are based on asymptotic analyses valid only for large data sets or large input dimensions. Many are statements about the existence of a solution rather than constructive statements about how to find the necessary set of weights. Most depend on particular assumptions that may be violated in a given problem.

In real problems, training sets may be rather small and nonrandomly sampled, data distributions may have arbitrary forms, and the target function is unknown. It may be problematic just to determine if the data fits the assumptions. Although representational capability results can be helpful in putting bounds on the size and configuration of a network, they do not guarantee that a particular network and training algorithm will be able to learn a particular set of samples of a given function. Even if it is guaranteed that a particular network can exactly classify N training points (using some set of weights), this does not necessarily imply that the system will generalize well to new points. Finally, many of the results for Boolean functions are for exact implementation and may not hold when approximation is allowed.

4.4 Capacity versus Size

Another big question in designing a network is how many nodes to place in each layer. Universal approximation results say that a sufficiently large network can fit almost any function we are likely to want to approximate, but they do not deal with the problem of training a network to fit a finite set of discrete data points. Obviously, we cannot have infinite numbers of nodes in practice so it is useful to have bounds on the number that will be needed to fit a particular function. Even if arbitrarily large networks were allowed, it might not be possible to use them effectively given the limited amount of information contained in the training samples. After the network grows past a certain size,

generalization criteria become the limiting factor on performance; a large network can often fit the training data exactly but is unlikely to do so in a way that fits the underlying function that generated the data.

Obviously, if we have m training points, a network with a single layer of $m - 1$ hidden units can learn the data exactly, since a line can always be found onto which the points project uniquely [80]. (Set each weight vector parallel to this line, make the magnitude large so the sigmoid approaches a step function, adjust the threshold of node k, $k = 1 \ldots M - 1$, so it separates points k and $k + 1$, and assign the output weights based on the difference in target values for points $k + 1$ and k.) Of course, this is inefficient and generalizes very badly; it uses as much storage as a nearest neighbor classifier, takes about the same time to evaluate on a serial computer, and probably generalizes worse. (Poor generalization could be expected just on the grounds that the number of weights would be much larger than the number of training samples.)

In general, more efficient solutions are sought. Most interesting functions have structure so each node should be able to account for more than just one training sample.

The sections that follow summarize some results on the number of patterns representable by multilayer networks of a given size. Results are simply listed in most cases; the reader should consult the references for details. Although these results may be used as guidelines in selecting a network, they should not be interpreted as inviolable rules. In many cases, they put bounds on the number of independent samples a given net can represent. The bounds may be loose, however, and actual data may be correlated so smaller networks may suffice for particular problems. Furthermore, we do not need the network to be able to fit *any* function on the samples, just the particular function that generated the data. We would like the network to fit the data and approximate the true target function, but the true target function is usually unknown and if we make the network powerful enough to represent any possible function on the data, it is probably too powerful. If the results say a given network can fit the m training points exactly, there is a danger of overfitting so smaller networks should probably be considered. Generalization depends on many factors so formulas based only on network size and number of samples cannot predict generalization performance.

4.4.1 Number of Cells Created by *m* Hyperplanes

For a layered network to realize an arbitrary dichotomy on m points in a d-dimensional space, each point must be uniquely represented in terms of the activities of the first hidden layer units.

Consider a network of linear threshold units. Each unit in the first hidden layer has an associated hyperplane that partitions the input space into two half-spaces. Two hyperplanes can divide the space into four quadrants, three hyperplanes may produce eight octants,

and so on. In general, h hyperplanes could produce up to 2^h different regions, or cells (assuming $h \leq d$, the input dimension). The hyperplanes form the cell boundaries. Within each cell, the vector of hidden layer outputs is constant (since the input must cross at least one hyperplane boundary for the output to change) so two points in the same cell are indistinguishable in terms of the hidden layer outputs. Thus, to realize an arbitrary dichotomy on m points, each point must lie in a different cell. If the network does not have enough hidden units to create at least m cells, it won't be able to realize some dichotomies.

An expression for the number of cells created by intersections of hyperplanes is derived by Makhoul, El-Jaroudi, and Schwartz [255]. The results are similar to Cover's [87] formula for the number of linearly separable dichotomies (section 3.3). Let $C(h, d)$ be the number of cells formed by h planes in general position in the input space of d dimensions. (h planes are in "general position" in d space if none of them are parallel and no more than d planes intersect at a point.) A recursive formula is

$$C(h, d) = C(h-1, d) + C(h-1, d-1). \tag{4.3}$$

with boundary conditions

$$C(0, d) = 1 \quad (d \geq 0) \tag{4.4}$$

$$C(h, 0) = 1 \quad (h \geq 1). \tag{4.5}$$

This gives [255]

$$C(h, d) = \begin{cases} 2^h & h \leq d \\ \sum_{i=0}^{d} \binom{h}{i} & h > d. \end{cases} \tag{4.6}$$

For $h < d$, each new hyperplane can be positioned to split all existing cells so up to 2^h cells can be created. For $h > d$, $C(h, d)$ grows more slowly because each new hyperplane cannot split every existing cell. For $h \gg d$, the last term $\binom{h}{d}$ dominates and

$$C(h, d) \approx \frac{h^d}{d!}, \quad (h \gg d). \tag{4.7}$$

The number of cells obtained by adding a new hyperplane (in terms of the number of existing cells) is

$$C(h, d) = 2C(h-1, d) - \binom{h-1}{d} \tag{4.8}$$

and the number of cells obtained when adding a new dimension is

$$C(h, d) = C(h, d-1) + \binom{h}{d}. \tag{4.9}$$

Expressions for the number of open (bounded at infinity) and closed cells are also given in [255]. Two hidden layers are required to implement all the $2^{C(h,d)}$ possible binary functions on the $C(h, d)$ cells [255].

Although this puts bounds on the number of nodes (hyperplanes) needed to form an arbitrary dichotomy on m points, in practical problems we have one specific dichotomy to implement. The data are likely to have structure and large savings can often be obtained because a single cell can contain entire clusters of points belonging to the same class. Also, when the input data are highly correlated, the effective dimension d' may be less than d.

4.4.2 MLP Capacity I

Baum [33, 32] studied the number of examples needed to train a network with N nodes and W weights for a given error rate ϵ. These results are independent of the training algorithm and are based on an estimate of the VC dimension of the network—a relation between a system's complexity and the amount of information that must be provided to constrain it (see section 15.4).

If a network can be trained with

$$m \geq O\left(\frac{W}{\epsilon} \log_2 \frac{N}{\epsilon}\right)$$

random examples of the desired mapping so that at least a fraction $1 - \epsilon/2$ are correctly classified, then it will almost certainly correctly classify a fraction $1 - \epsilon$ of test examples drawn from the same distribution (for $0 < \epsilon \leq 1/8$) [33, 32]. This is an upper bound for the number of training examples needed. A lower bound for MLPs with one hidden layer is $\Omega(W/\epsilon)$. That is, a network trained on fewer than $\Omega(W/\epsilon)$ will fail (some fraction of the time) to find a set of weights that classifies a fraction $1 - \epsilon$ of future examples correctly.

This agrees with the W/ϵ rule of thumb given by Widrow for Adaline networks. Similar results are also obtained in linear regression. Roughly, if the network has W adjustable weights and an error rate ϵ is desired, it is necessary to have on the order of W/ϵ samples.

Assuming that the function can be learned, this gives an estimate of the network size necessary to learn m training patterns. For input patterns chosen randomly from the domain $\{-1, +1\}^n$, a network with one hidden layer of $2\lceil m/\lfloor n(1 - \frac{10}{\ln n})\rfloor\rceil$ linear threshold units can learn the $m \leq 2^{n/3}$ training patterns.

4.4.3 MLP Capacity II

Baum [30] makes the following points:

• A MLP with one hidden layer of $N-1$ nodes is capable of computing an arbitrary dichotomy on N points [284].

• A network that implements $f(x)$ on N points in general position in d dimensions, must have at least N/d units in the first hidden layer [30]. If the points are in nongeneral position (structured data), many fewer units may be necessary.

• A one-hidden-layer net with $\lceil N/d \rceil$ hidden units can compute an arbitrary $f(x)$ on N points in general position [30]. Small changes in the input vector may cause large changes in the output, however, so good generalization is not guaranteed.

• The number of linearly separable dichotomies of N points in d dimensions, for $N \geq 3d$, is less than $4N^d/d!$ [30].

• No feedforward net (of the type considered) can compute an arbitrary map from N d-dimensional vectors into the e dimensional hypercube unless it has a number of connections $N_c \geq \frac{eN}{\log_2 N}$.

• For a one-hidden-layer network with d inputs, G hidden units, and e outputs, the number of connections is $N_c = G(e+d)$. Thus, a one-hidden-layer network cannot compute arbitrary functions on a set of N vectors in d-dimensions unless it has at least $G = \frac{eN}{(e+d)\log_2 N}$ hidden units.

• No MLP can compute an arbitrary function, no matter how many layers it has, unless it has $O(\sqrt{\frac{Ne}{\log_2 N}})$ units. As for all the items in this section, many fewer units may be needed if the training data is structured.

• A one-hidden-layer net with N hidden units can represent an arbitrary mapping of N points into the e hypercube.

• A one-hidden-layer net with $G = \lfloor \frac{4N}{d} \rfloor \lceil \frac{e}{\lfloor \log_2 \frac{N}{d} \rfloor} \rceil$ hidden units is capable of arbitrary binary mappings.

• There are 2^{Ne} mappings of N d-dimensional vectors into the e-dimensional hypercube.

4.4.4 MLP Capacity III

Widrow and Lehr [403] consider a fully connected feedforward network of linear threshold units with N_x inputs (excluding the bias), N_h hidden nodes in a single layer, and N_y outputs. There are N_w weights and N_p patterns in general position to be learned.

A bound on the number of weights needed is

$$\frac{N_y N_p}{1+\log_2 N_p} \leq N_w < N_y \left(\frac{N_p}{N_x}+1\right)(N_x+N_y+1)+N_y. \tag{4.10}$$

A loose upper bound for the number of hidden nodes N_h required is

$$N_h \leq N_y \frac{N_p}{N_x} < N_y \left(\frac{N_p}{N_x}+1\right). \tag{4.11}$$

When N_x,$N_h >\sim 5N_y$ (when there are more inputs and hidden nodes than outputs), the deterministic capacity (the number of patterns that can certainly be stored) is bounded below by $\sim N_w/N_y$. When $N_w \gg N_y$ (i.e. $1000\times$), the capacity is bounded above by

$$\sim \frac{N_w}{N_y} \log_2 \frac{N_w}{N_y}.$$

Thus

$$\frac{N_w}{N_y} - k_1 \leq C_d \leq \frac{N_w}{N_y} \log_2 \frac{N_w}{N_y} + k_2 \tag{4.12}$$

where k_1 and k_2 are small constants when $N_x + N_h \gg N_y$.

For good generalization, the number of training patterns should be at least several times larger than the capacity of the network. Otherwise, the amount of data will be insufficient to constrain the network.

4.4.5 MLP Capacity IV

The results of section 3.3 show that, on average, a single linear threshold unit with n inputs can be made to correctly classify up to $m = 2n$ random binary patterns before the probability of error falls to 1/2. In other words, a linear threshold unit has a probabilistic capacity of $m = 2n$ patterns.

Mitchison and Durbin [269] study the capacity of a MLP with one hidden layer. As above, the capacity is defined as the number of random input/output patterns that can be stored before the probability of error on recall reaches 1/2. They find that for a MLP with n inputs, one layer of h hidden units, and s output units, where $s \leq h \leq n$, the capacity m satisfies

$$2n \leq m \leq nt \log t \tag{4.13}$$

where $t = 1 + h/s$, $n \geq 2$, and $t \geq 2$ [269].

If there is a single output that is a fixed Boolean function of the hidden units, then $m \leq O(nh \log h)$. Comparing this to the case above when $s = 1$ and thus $t = 1 + h$ shows that allowing the output to be a variable function of the hidden units has an effect on the capacity equivalent to adding one hidden unit. Because a one-hidden-layer network can be connected in such a way that it has the same response as a single-layer network with the same number of inputs and outputs, the lower bound of $m \geq 2n$ still applies. For the special case $s = h = n$, they find $2n \leq m \leq 9.329n$.

Note that the capacity bounds are defined probabilistically for random functions on a randomly chosen set of points; the actual number of example pairs of a particular function that can be learned by a particular network will depend heavily on the function and how the examples are chosen. These are limiting bounds for the capacity; m certainly exceeds the lower bound and is certainly less than the upper bound. The actual capacity that can be achieved is less than the upper bound, possibly by a large amount.