Chapter 6

Multilayer Neural Networks

6.1 Introduction

In the previous chapter we saw a number of methods for training classifiers consisting of input units connected by modifiable weights to output units. The LMS algorithm, in particular, provided a powerful gradient descent method for reducing the error, even when the patterns are not linearly separable. Unfortunately, the class of solutions that can be obtained from such networks — hyperplane discriminants — while surprisingly good on a range or real-world problems, is simply not general enough in demanding applications: there are many problems for which linear discriminants are insufficient for minimum error.

With a clever choice of nonlinear φ functions, however, we can obtain arbitrary decisions, in particular the one leading to minimum error. The central difficulty is, naturally, choosing the appropriate nonlinear functions. One brute force approach might be to choose a complete basis set (all polynomials, say) but this will not work; such a classifier would have too many free parameters to be determined from a limited number of training patterns (Chap. **??**). Alternatively, we may have prior knowledge relevant to the classification problem and this might guide our choice of nonlinearity. In the absence of such information, up to now we have seen no principled or automatic method for finding the nonlinearities. What we seek, then, is a way to *learn* the nonlinearity at the same time as the linear discriminant. This is the approach of multilayer neural networks (also called multilayer Perceptrons): the parameters governing the nonlinear mapping are learned at the same time as those governing the linear discriminant.

We shall revisit the limitations of the two-layer networks of the previous chapter,* and see how three-layer (and four-layer...) nets overcome those drawbacks — indeed how such multilayer networks can, at least in principle, provide the optimal solution to an arbitrary classification problem. There is nothing particularly magical about multilayer neural networks; at base they implement *linear* discriminants, but in a space where the inputs have been mapped nonlinearly. The key power provided by such networks is that they admit fairly simple algorithms where the form of the nonlinearity

* Some authors describe such networks as *single* layer networks because they have only one layer of modifiable weights, but we shall instead refer to them based on the number of layers of *units*.

can be learned from training data. The models are thus extremely powerful, have nice theoretical properties, and apply well to a vast array of real-world applications.

BACKPROPAGATION

One of the most popular methods for training such multilayer networks is based on gradient descent in error — the *backpropagation algorithm* (or generalized delta rule), a natural extension of the LMS algorithm. We shall study backpropagation in depth, first of all because it is powerful, useful and relatively easy to understand, but also because many other training methods can be seen as modifications of it. The backpropagation training method is simple even for complex models (networks) having hundreds or thousands of parameters. In part because of the intuitive graphical representation and the simplicity of design of these models, practitioners can test different models quickly and easily; neural networks are thus a sort of "poor person's" technique for doing statistical pattern recognition with complicated models. The conceptual and algorithmic simplicity of backpropagation, along with its manifest success on many real-world problems, help to explain why it is a mainstay in adaptive pattern recognition.

While the basic theory of backpropagation is simple, a number of tricks — some a bit subtle — are often used to improve performance and increase training speed. Choices involving the scaling of input values and initial weights, desired output values, and more can be made based on an analysis of networks and their function. We shall also discuss alternate training schemes, for instance ones that are faster, or adjust their complexity automatically in response to training data.

Network architecture or topology plays an important role for neural net classification, and the optimal topology will depend upon the problem at hand. It is here that another great benefit of networks becomes apparent: often knowledge of the problem domain which might be of an informal or heuristic nature can be easily incorporated into network architectures through choices in the number of hidden layers, units, feedback connections, and so on. Thus setting the topology of the network is heuristic model selection. The practical ease in selecting models (network topologies) and estimating parameters (training via backpropagation) enable classifier designers to try out alternate models fairly simply.

REGULARIZATION

A deep problem in the use of neural network techniques involves regularization, complexity adjustment, or model selection, that is, selecting (or adjusting) the complexity of the network. Whereas the number of inputs and outputs is given by the feature space and number of categories, the total number of weights or parameters in the network is not — or at least not directly. If too many free parameters are used, generalization will be poor; conversely if too few parameters are used, the training data cannot be learned adequately. How shall we adjust the complexity to achieve the best generalization? We shall explore a number of methods for complexity adjustment, and return in Chap. **??** to their theoretical foundations.

It is crucial to remember that neural networks do not exempt designers from intimate knowledge of the data and problem domain. Networks provide a powerful and speedy tool for building classifiers, and as with any tool or technique one gains intuition and expertise through analysis and repeated experimentation over a broad range of problems.

6.2 Feedforward operation and classification

HIDDEN LAYER

Figure 6.1 shows a simple three-layer neural network. This one consists of an input layer (having two input units), a *hidden layer* with (two hidden units)* and an output

layer (a single unit), interconnected by modifiable weights, represented by links between layers. There is, furthermore, a single *bias unit* that is connected to each unit other than the input units. The function of units is loosely based on properties of biological neurons, and hence they are sometimes called "neurons." We are interested in the use of such networks for pattern recognition, where the input units represent the components of a feature vector (to be learned or to be classified) and signals emitted by output units will be discriminant functions used for classification.

BIAS

NEURON

* We call any units that are neither input nor output units "hidden" because their activations are not directly "seen" by the external environment, i.e., the input or output.

RECALL

We can clarify our notation and describe the feedforward (or classification or recall) operation of such a network on what is perhaps the simplest nonlinear problem: the exclusive-OR (XOR) problem (Fig. 6.1); a three-layer network can indeed solve this problem whereas a linear machine operating directly on the features cannot.

Each two-dimensional input vector is presented to the input layer, and the output of each input unit equals the corresponding component in the vector. Each hidden

NET ACTIVATION

unit performs the weighted sum of its inputs to form its (scalar) *net activation* or simply *net*. That is, the net activation is the inner product of the inputs with the weights at the hidden unit. For simplicity, we augment both the input vector (i.e., append a feature value $x_0 = 1$) and the weight vector (i.e., append a value w_0), and can then write

$$net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} \equiv \mathbf{w}_j^t \mathbf{x}, \tag{1}$$

where the subscript i indexes units on the input layer, j for the hidden; w_{ji} denotes the input-to-hidden layer weights at the hidden unit j. In analogy with neurobiol-

SYNAPSE

ogy, such weights or connections are sometimes called "synapses" and the value of the connection the "synaptic weights." Each hidden unit emits an output that is a nonlinear function of its activation, $f(net)$, i.e.,

$$y_j = f(net_j). \tag{2}$$

The example shows a simple threshold or *sign* (read "signum") function,

$$f(net) = Sgn(net) \equiv \begin{cases} 1 & \text{if } net \geq 0 \\ -1 & \text{if } net < 0, \end{cases} \tag{3}$$

but as we shall see, other functions have more desirable properties and are hence

TRANSFER FUNCTION

more commonly used. This $f()$ is sometimes called the *transfer function* or merely "nonlinearity" of a unit, and serves as a φ function discussed in Chap. **??**. We have assumed the *same* nonlinearity is used at the various hidden and output units, though this is not crucial.

Each output unit similarly computes its net activation based on the hidden unit signals as

$$net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} y_j w_{kj} = \mathbf{w}_k^t \mathbf{y}, \tag{4}$$

where the subscript k indexes units in the output layer (one, in the figure) and n_H denotes the number of hidden units (two, in the figure). We have mathematically treated the bias unit as equivalent to one of the hidden units whose output is always $y_0 = 1$. Each output unit then computes the nonlinear function of its *net*, emitting

$$z_k = f(net_k). \tag{5}$$

where in the figure we assume that this nonlinearity is also a sign function. It is these final output signals that represent the different discriminant functions. We would typically have c such output units and the classification decision is to label the input pattern with the label corresponding to the maximum $y_k = g_k(\mathbf{x})$. In a two-category case such as XOR, it is traditional to use a single output unit and label a pattern by the sign of the output z.

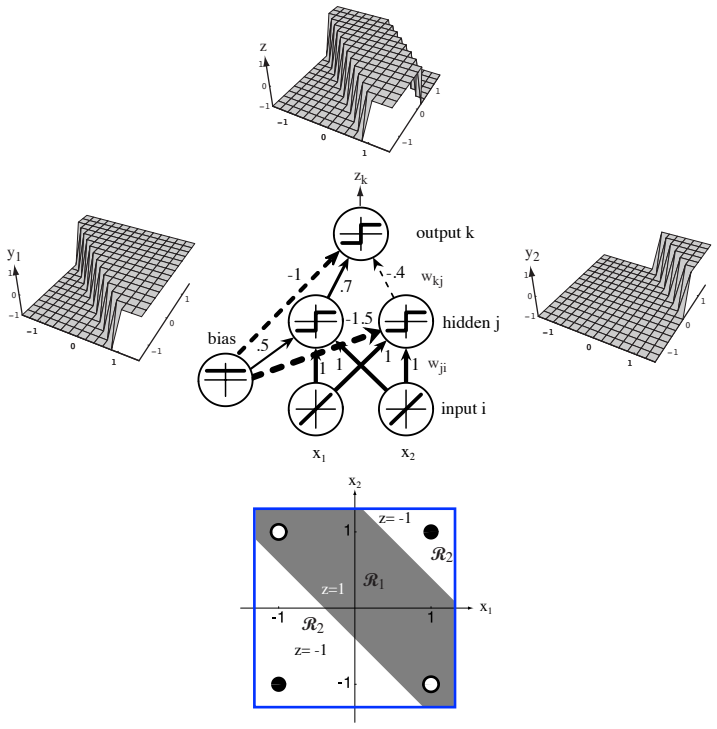

Figure 6.1: The two-bit parity or exclusive-OR problem can be solved by a three-layer network. At the bottom is the two-dimensional feature space $x_1 - x_2$, and the four patterns to be classified. The three-layer network is shown in the middle. The input units are linear and merely distribute their (feature) values through multiplicative weights to the hidden units. The hidden and output units here are linear threshold units, each of which forms the linear sum of its inputs times their associated weight, and emits a $+1$ if this sum is greater than or equal to 0, and -1 otherwise, as shown by the graphs. Positive ("excitatory") weights are denoted by solid lines, negative ("inhibitory") weights by dashed lines; the weight magnitude is indicated by the relative thickness, and is labeled. The single output unit sums the weighted signals from the hidden units (and bias) and emits a $+1$ if that sum is greater than or equal to 0 and a -1 otherwise. Within each unit we show a graph of its input-output or transfer function — $f(net)$ vs. net. This function is linear for the input units, a constant for the bias, and a step or sign function elsewhere. We say that this network has a 2-2-1 fully connected topology, describing the number of units (other than the bias) in successive layers.

It is easy to verify that the three-layer network with the weight values listed indeed solves the XOR problem. The hidden unit computing y_1 acts like a Perceptron, and computes the boundary $x_1 + x_2 + 0.5 = 0$; input vectors for which $x_1 + x_2 + 0.5 \geq 0$ lead to $y_1 = 1$, all other inputs lead to $y_1 = -1$. Likewise the other hidden unit computes the boundary $x_1 + x_2 - 1.5 = 0$. The final output unit emits $z_1 = +1$ if and only if *both* y_1 and y_2 have value $+1$. This gives to the appropriate nonlinear decision region shown in the figure — the XOR problem is solved.

6.2.1 General feedforward operation

From the above example, it should be clear that nonlinear multilayer networks (i.e., ones with input units, hidden units and output units) have greater computational or *expressive power* than similar networks that otherwise lack hidden units; that is, they can implement more functions. Indeed, we shall see in Sect. 6.2.2 that given sufficient number of hidden units of a general type *any* function can be so represented.

EXPRESSIVE POWER

Clearly, we can generalize the above discussion to more inputs, other nonlinearities, and arbitrary number of output units. For classification, we will have c output units, one for each of the categories, and the signal from each output unit is the discriminant function $g_k(\mathbf{x})$. We gather the results from Eqs. 1, 2, 4, & 5, to express such discriminant functions as:

$$g_k(\mathbf{x}) \equiv z_k = f\left(\sum_{j=1}^{n_H} w_{kj}\ f\left(\sum_{i=1}^{d} w_{ji}x_i + w_{j0}\right) + w_{k0}\right). \tag{6}$$

This, then, is the class of functions that can be implemented by a three-layer neural network. An even broader generalization would allow transfer functions at the output layer to differ from those in the hidden layer, or indeed even different functions at each individual unit. We will have cause to use such networks later, but the attendant notational complexities would cloud our presentation of the key ideas in learning in networks.

6.2.2 Expressive power of multilayer networks

It is natural to ask if *every* decision can be implemented by such a three-layer network (Eq. 6). The answer, due ultimately to Kolmogorov but refined by others, is "yes" — any continuous function from input to output can be implemented in a three-layer net, given sufficient number of hidden units n_H, proper nonlinearities, and weights. In particular, any posterior probabilities can be represented. In the c-category classification case, we can merely apply a max[$\cdot$] function to the set of network outputs (just as we saw in Chap. **??**) and thereby obtain any decision boundary.

Specifically, Kolmogorov proved that any continuous function $g(\mathbf{x})$ defined on the unit hypercube I^n ($I = [0, 1]$ and $n \geq 2$) can be represented in the form

$$g(\mathbf{x}) = \sum_{j=1}^{2n+1} \Xi_j \left(\sum_{i=1}^{d} \psi_{ij}(x_i)\right) \tag{7}$$

for properly chosen functions Ξ_j and ψ_{ij}. We can always scale the input region of interest to lie in a hypercube, and thus this condition on the feature space is not limiting. Equation 7 can be expressed in neural network terminology as follows: each of $2n + 1$ hidden units takes as input a sum of d nonlinear functions, one for each

input feature x_i. Each hidden unit emits a nonlinear function Ξ of its total input; the output unit merely emits the sum of the contributions of the hidden units.

Unfortunately, the relationship of Kolmogorov's theorem to practical neural networks is a bit tenuous, for several reasons. In particular, the functions Ξ_j and ψ_{ij} are not the simple weighted sums passed through nonlinearities favored in neural networks. In fact those functions can be extremely complex; they are not smooth, and indeed for subtle mathematical reasons they cannot be smooth. As we shall soon see, smoothness is important for gradient descent learning. Most importantly, Kolmogorov's Theorem tells us very little about how to find the nonlinear functions based on data — the central problem in network based pattern recognition.

A more intuitive proof of the universal expressive power of three-layer nets is inspired by Fourier's Theorem that any continuous function $g(\mathbf{x})$ can be approximated arbitrarily closely by a (possibly infinite) sum of harmonic functions (Problem 2). One can imagine networks whose hidden units implement such harmonic functions. Proper hidden-to-output weights related to the coefficients in a Fourier synthesis would then enable the full network to implement the desired function. Informally speaking, we need not build up harmonic functions for Fourier-like synthesis of a desired function. Instead a sufficiently large number of "bumps" at different input locations, of different amplitude and sign, can be put together to give our desired function. Such localized bumps might be implemented in a number of ways, for instance by sigmoidal transfer functions grouped appropriately (Fig. 6.2). The Fourier analogy and bump constructions are conceptual tools, they do not explain the way networks in fact function. In short, this is not how neural networks "work" — we never find that through training (Sect. 6.3) simple networks build a Fourier-like representation, or learn to group sigmoids to get component bumps.

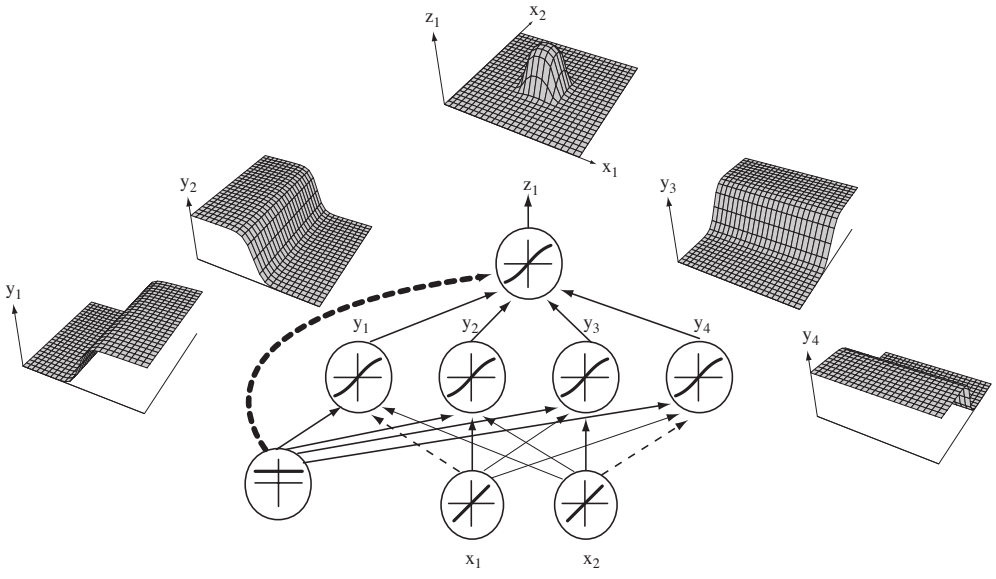

Figure 6.2: A 2-4-1 network (with bias) along with the response functions at different units; each hidden and output unit has sigmoidal transfer function $f(\cdot)$. In the case shown, the hidden unit outputs are paired in opposition thereby producing a "bump" at the output unit. Given a sufficiently large number of hidden units, any continuous function from input to output can be approximated arbitrarily well by such a network.

While we can be confident that a complete set of functions, such as all polynomials, can represent any function it is nevertheless a fact that a *single* functional form also suffices, so long as each component has appropriate variable parameters. In the absence of information suggesting otherwise, we generally use a single functional form for the transfer functions.

While these latter constructions show that any desired function can be implemented by a three-layer network, they are not particularly practical because for most problems we know ahead of time neither the number of hidden units required, nor the proper weight values. Even if there *were* a constructive proof, it would be of little use in pattern recognition since we do not know the desired function anyway — it is related to the training patterns in a very complicated way. All in all, then, these results on the expressive power of networks give us confidence we are on the right track, but shed little practical light on the problems of designing and *training* neural networks — their main benefit for pattern recognition (Fig. 6.3).

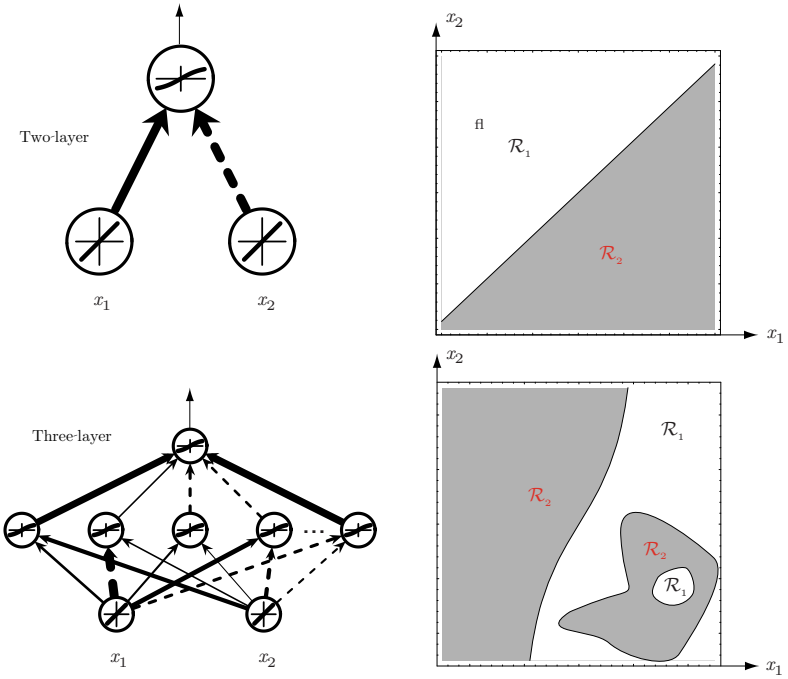

Figure 6.3: Whereas a two-layer network classifier can only implement a linear decision boundary, given an adequate number of hidden units, three-, four- and higher-layer networks can implement arbitrary decision boundaries. The decision regions need not be convex, nor simply connected.

6.3 Backpropagation algorithm

We have just seen that any function from input to output can be implemented as a three-layer neural network. We now turn to the crucial problem of setting the weights based on training patterns and desired output.