CHAPTER

6

MULTILAYER NEURAL NETWORKS

6.1 INTRODUCTION

We saw in Chapter 5 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—comprising hyperplane decision boundaries—while surprisingly good on a range of 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 decision regions, in particular those 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 such as all polynomials, but this will not work; such a classifier would have too many free parameters to be determined from a limited number of training patterns (Chapter 9). Alternatively, we may have prior knowledge relevant to the classification problem and this might guide our choice of nonlinearity. However, we have seen no principled or automatic method for finding the nonlinearities in the absence of such information. 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 or 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- and four-layer nets overcome those drawbacks—indeed

*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*.

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; 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 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.

BACK-PROPAGATION

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 having hundreds or thousands of parameters. In part because of the intuitive graphical representation and the simplicity of design of these models, designers can test different models quickly and easily; neural networks are thus a flexible heuristic 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, make it a mainstay in adaptive pattern recognition.

While the basic theory of backpropagation is simple, a number of heuristics—some a bit subtle—are often used to improve performance and increase training speed. Guided by an analysis of networks and their function we can make informed choices of the scaling of input values and initial weights, desired output values, and more. We shall also discuss alternative 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 by setting the network topology and estimating parameters via backpropagation enables classifier designers to try out alternative models fairly simply.

REGULARIZATION

A deep problem in the use of neural network techniques involves regularization, 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 we will return in Chapter 9 to consider their theoretical foundations.

It is crucial to remember that neural networks do not exempt designers from intimate and detailed 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.

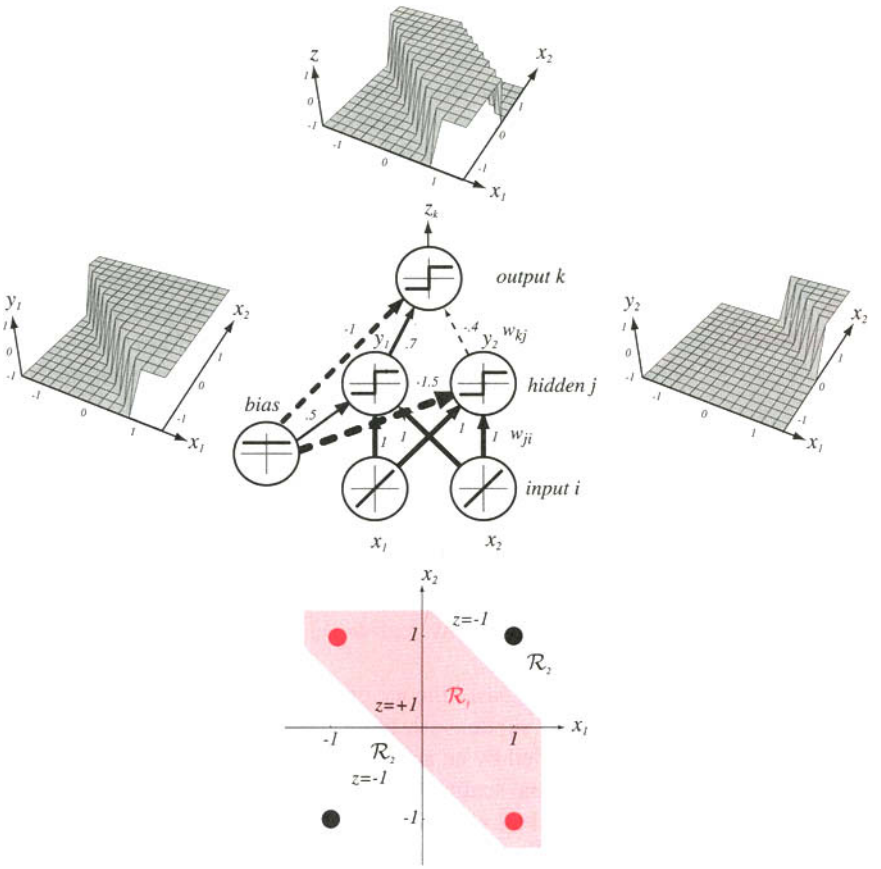


FIGURE 6.1. The two-bit parity or exclusive-OR problem can be solved by a three-layer network. At the bottom is the two-dimensional feature $x_1 x_2$-space, along with the four patterns to be classified. The three-layer network is shown in the middle. The input units are linear and merely distribute 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 to yield *net*, and emits a +1 if this *net* is greater than or equal to 0, and −1 otherwise, as shown by the graphs. Positive or "excitatory" weights are denoted by solid lines, negative or "inhibitory" weights by dashed lines; each weight magnitude is indicated by the line's thickness, and is labeled. The single output unit sums the weighted signals from the hidden units and bias to form its *net*, and emits a +1 if its *net* is greater than or equal to 0 and emits a −1 otherwise. Within each unit we show a graph of its input-output or activation function—$f(net)$ versus *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.

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, a *hidden layer*,* and an output layer, interconnected by modifiable weights,

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

BIAS UNIT

NEURON

represented by links between layers. There is, furthermore, a single *bias unit* that is connected to each unit other than the input units. Cleary, such a network is an extension of the two-layer networks we studied in Chapter 5. 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 and where signals emitted by output units will be the values of the discriminant functions used for classification.

We can clarify our notation and describe the feedforward 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.

NET ACTIVATION

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 unit computes the weighted sum of its inputs to form its scalar *net activation* which we denote simply as *net*. That is, the net activation is the inner product of the inputs with the weights at the hidden unit. As in Chapter 5, for simplicity we augment both the input vector, by appending a feature value $x_0 = 1$, as well as the weight vector, by appending a value w_0, and we can then write

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

SYNAPSE

where the subscript i indexes units in the input layer, j in the hidden; w_{ji} denotes the input-to-hidden layer weights at the hidden unit j. In analogy with neurobiology, such weights or connections are sometimes called "synapses" and the values of the connections the "synaptic weights." Each hidden unit emits an output that is a nonlinear function of its activation, $f(net)$, that is,

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

Figure 6.1 shows a simple threshold or *sign* (read "signum") function,

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

ACTIVATION FUNCTION

but as we shall see, other functions have more desirable properties and are hence more commonly used. This $f(\cdot)$ is sometimes called the *activation function* or merely "nonlinearity" of a unit, and it serves as a φ function discussed in Chapter 5. 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 and n_H denotes the number of hidden units. We have mathematically treated the bias unit as equivalent to one of the hidden units whose output is always $y_0 = 1$. In this example, there is only one output unit. However, anticipating a more general case, we shall refer to its output as

z_k. An output unit 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. Clearly, the output z_k can also be thought of as a function of the input feature vector $\mathbf{x}$. When there are c output units, we can think of the network as computing c discriminant functions $z_k = g_k(\mathbf{x})$, and can classify the input according to which discriminant function is largest. In a two-category case, it is traditional to use a single output unit and label a pattern by the sign of the output z.

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 two-layer Perceptron, and it computes the boundary $x_1 + x_2 + 0.5 = 0$; input vectors for which $x_1 + x_2 + 0.5 \geq 0$ lead to $y_1 = 1$, and 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 $y_1 = +1$ and $y_2 = +1$. Using the terminology of computer logic, the units are behaving like gates, where the first hidden unit is an OR gate, the second hidden unit is an AND gate, and the output unit implements

$$\begin{aligned} z_k &= y_1 \text{ AND NOT } y_2 = (x_1 \text{ OR } x_2) \text{ AND NOT } (x_1 \text{ AND } x_2) \\ &= x_1 \text{ XOR } x_2, \end{aligned} \tag{6}$$

giving rise to the appropriate nonlinear decision region shown in the figure—the XOR problem is solved.

6.2.1 General Feedforward Operation

EXPRESSIVE POWER

From the above example, it should be clear that nonlinear multilayer networks—that is, 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 Section 6.2.2 that given sufficient number of hidden units of a general type *any* function can be so represented.

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, and 5, to express such discriminant functions as

$$g_k(\mathbf{x}) \equiv z_k = f\left(\sum_{j=1}^{n_H} w_{kj}\, f\left(\sum_{i=1}^{d} w_{ji} x_i + w_{j0}\right) + w_{k0}\right). \tag{7}$$

This, then, is the class of functions that can be implemented by a three-layer neural network. In general, as we elaborate in Section 6.8.1, the activation function does not have to be a sign function. Indeed, we shall often require the activation functions to be continuous and differentiable. We can even allow the activation functions in the output layer to be different from the activation functions in the hidden layer, or indeed have different activation functions for each individual unit. Although we will have cause to use such networks later, to simplify the mathematical analysis and reveal

the essential concepts, we will temporarily assume that the activation functions are all the same.

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 as described by Eq. 7. 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 by a three-layer net. Just as we did in Chapter 2, for this c-category classification case we can merely apply a max[·] function to the set of network outputs, 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

$$g(\mathbf{x}) = \sum_{j=1}^{2n+1} \Xi_j \left(\sum_{i=1}^{d} \psi_{ij}(x_i) \right) \tag{8}$$

for properly chosen functions Ξ_j and ψ_{ij}. In any particular probem, 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 8 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 a network 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 S-shaped activation functions grouped appropriately (Fig. 6.2). The Fourier analogy and bump constructions are conceptual tools, and 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 (Section 6.3) simple networks build a Fourier-like representation, nor do they learn to group S-shaped functions to get component bumps. However, these analogies help to explain the reasons for the expressive power of multilayer networks.

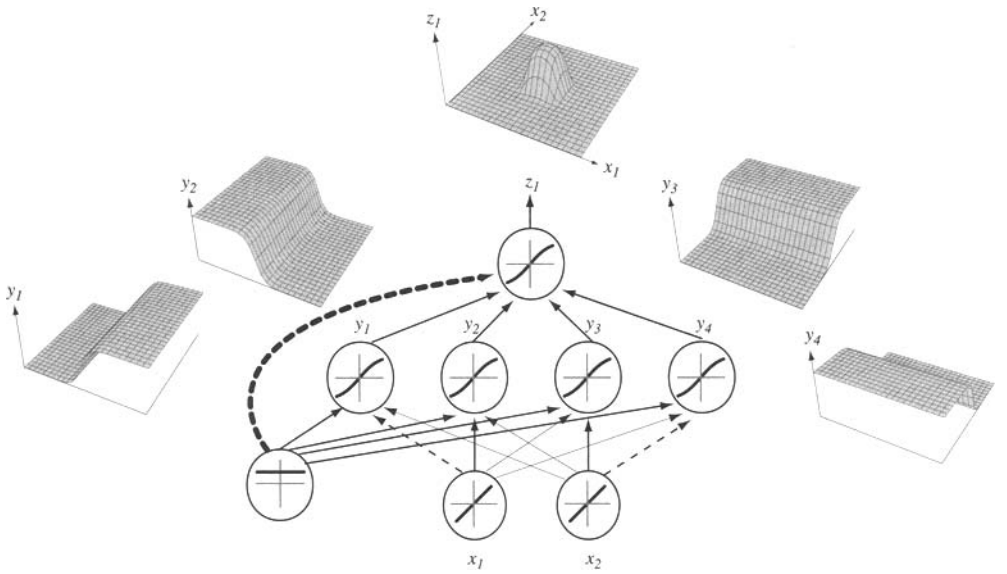


FIGURE 6.2. A 2-4-1 network (with bias) along with the response functions at different units; each hidden output unit has sigmoidal activation function $f(\cdot)$. In the case shown, the hidden unit outputs are paired in opposition thereby producing a "bump" at the output unit. Given a sufficiently large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

These latter constructions, showing that any desired function can be implemented by a three-layer network, are of greater theoretical than practical interest because the constructions give 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 because 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).

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 the desired output.

Backpropagation is one of the simplest and most general methods for supervised training of multilayer neural networks—it is the natural extension of the LMS algorithm for linear systems we saw in Chapter 5. Other methods may be faster or have other desirable properties, but few are more instructive. The LMS algorithm worked for two-layer systems because the error, proportional to the square of the difference between the actual output and the desired output, could be evaluated for each output unit. Similarly, in a three-layer net it is a straightforward matter to find how the output, and thus the error, depends on the hidden-to-output layer weights. In fact, this dependency is the same as in the analogous two-layer case, and thus the learning rule is the same.

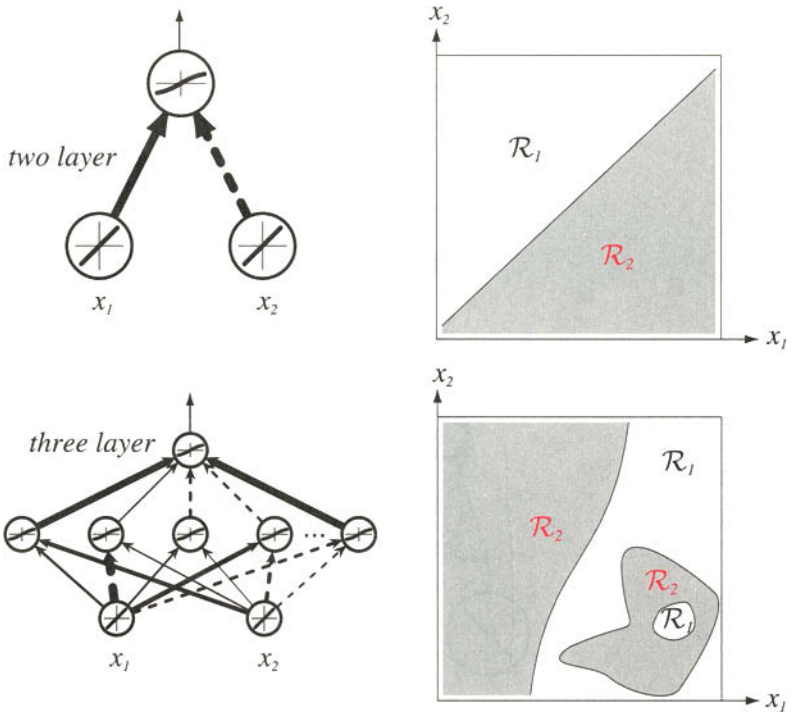


FIGURE 6.3. Whereas a two-layer network classifier can only implement a linear decision boundary, given an adequate number of hidden units, three-, four- and higher-layer networks can implement arbitrary decision boundaries. The decision regions need not be convex or simply connected.

But how should the input-to-hidden weights be learned, the ones governing the nonlinear transformation of the input vectors? If the "proper" outputs for a hidden unit were known for any pattern, the input-to-hidden weights could be adjusted to approximate it. However, there is no explicit teacher to state what the hidden unit's output should be. This is called the *credit assignment* problem. The power of backpropagation is that it allows us to calculate an effective error for each hidden unit, and thus derive a learning rule for the input-to-hidden weights.

CREDIT ASSIGNMENT

Networks have two primary modes of operation: feedforward and learning. Feedforward operation, such as illustrated in our XOR example above, consists of presenting a pattern to the input units and passing the signals through the network in order to yield outputs from the output units. Supervised learning consists of presenting an input pattern and changing the network parameters to bring the actual outputs closer to the desired teaching or *target* values. Figure 6.4 shows a three-layer network and the notation we shall use.

TARGET PATTERN

6.3.1 Network Learning

The basic approach in learning is to start with an untrained network, present a training pattern to the input layer, pass the signals through the net and determine the output at the output layer. Here these outputs are compared to the target values; any difference corresponds to an error. This error or criterion function is some scalar function of the weights and is minimized when the network outputs match the de-

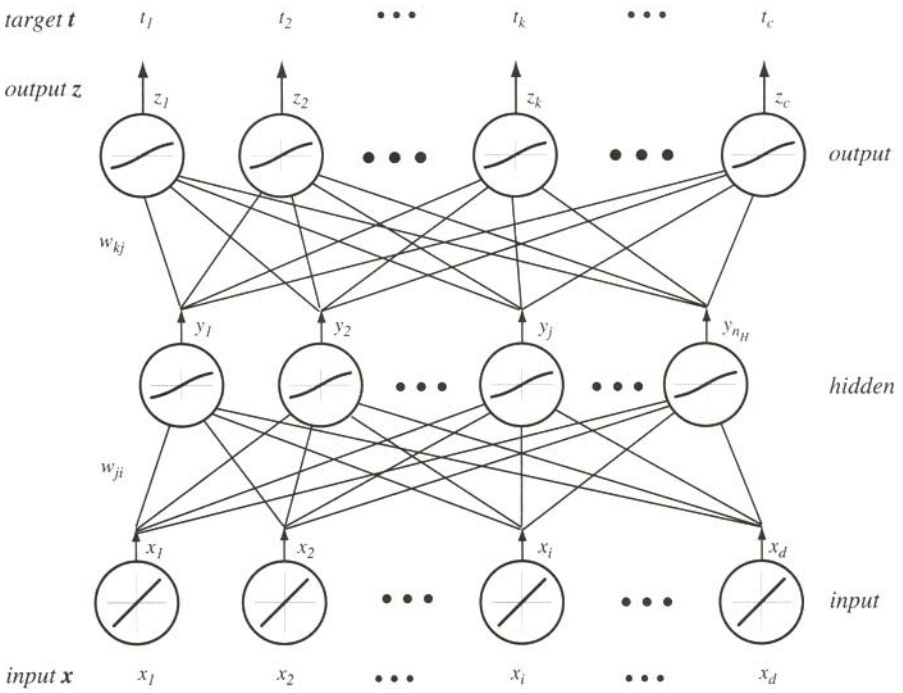


FIGURE 6.4. A d-n_H-c fully connected three-layer network and the notation we shall use. During feedforward operation, a d-dimensional input pattern $\mathbf{x}$ is presented to the input layer; each input unit then emits its corresponding component x_i. Each of the n_H hidden units computes its net activation, net_j, as the inner product of the input layer signals with weights w_{ji} at the hidden unit. The hidden unit emits $y_j = f(net_j)$, where $f(\cdot)$ is the nonlinear activation function, shown here as a sigmoid. Each of the c output units functions in the same manner as the hidden units do, computing net_k as the inner product of the hidden unit signals and weights at the output unit. The final signals emitted by the network, $z_k = f(net_k)$, are used as discriminant functions for classification. During network training, these output signals are compared with a teaching or target vector $\mathbf{t}$, and any difference is used in training the weights throughout the network.

sired outputs. Thus the weights are adjusted to reduce this measure of error. Here we present the learning rule on a per pattern basis, and we return to other protocols later.

TRAINING ERROR

We consider the *training error* on a pattern to be the sum over output units of the squared difference between the desired output t_k given by a teacher and the actual output z_k, much as we had in the LMS algorithm for two-layer nets:

$$J(\mathbf{w}) \equiv \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2 = \frac{1}{2}\|\mathbf{t} - \mathbf{z}\|^2, \tag{9}$$

where $\mathbf{t}$ and $\mathbf{z}$ are the target and the network output vectors of length c and $\mathbf{w}$ represents all the weights in the network.

The backpropagation learning rule is based on gradient descent. The weights are initialized with random values, and then they are changed in a direction that will reduce the error:

$$\Delta\mathbf{w} = -\eta\frac{\partial J}{\partial \mathbf{w}}, \tag{10}$$

or in component form

$$\Delta w_{pq} = -\eta \frac{\partial J}{\partial w_{pq}}, \tag{11}$$

LEARNING RATE

where η is the *learning rate*, and merely indicates the relative size of the change in weights. The power of Eqs. 10 and 11 is in their simplicity: They merely demand that we take a step in weight space that lowers the criterion function. It is clear from Eq. 9 that the criterion function can never be negative; furthermore, the learning rule guarantees that learning will stop, except in pathological cases. This iterative algorithm requires taking a weight vector at iteration m and updating it as

$$\mathbf{w}(m+1) = \mathbf{w}(m) + \Delta \mathbf{w}(m), \tag{12}$$

where m indexes the particular pattern presentation.

We now turn to the problem of evaluating Eq. 11 for a three-layer net. Consider first the hidden-to-output weights, w_{kj}. Because the error is not explicitly dependent upon w_{jk}, we must use the chain rule for differentiation:

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = -\delta_k \frac{\partial net_k}{\partial w_{kj}}, \tag{13}$$

SENSITIVITY

where the *sensitivity* of unit k is defined to be

$$\delta_k = -\partial J / \partial net_k, \tag{14}$$

and describes how the overall error changes with the unit's net activation. Assuming that the activation function $f(\cdot)$ is differentiable, we differentiate Eq. 9 and find that for such an output unit, δ_k is simply

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} = (t_k - z_k) f'(net_k). \tag{15}$$

The last derivative in Eq. 13 is found using Eq. 4:

$$\frac{\partial net_k}{\partial w_{kj}} = y_j. \tag{16}$$

Taken together, these results give the weight update or learning rule for the hidden-to-output weights:

$$\Delta w_{kj} = \eta \delta_k y_j = \eta (t_k - z_k) f'(net_k) y_j. \tag{17}$$

In the case that the output unit is linear—that is, $f(net_k) = net_k$ and $f'(net_k) = 1$—then Eq. 17 is simply the LMS rule we saw in Chapter 5.

The learning rule for the input-to-hidden units is more subtle, indeed, it is the crux of the solution to the credit assignment problem. From Eq. 11, and again using the chain rule, we calculate

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}. \tag{18}$$

The first term on the right-hand side involves all of the weights w_{kj}, and requires just a bit of care:

$$\begin{aligned}\frac{\partial J}{\partial y_j} &= \frac{\partial}{\partial y_j}\left[\frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2\right] \\ &= -\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial y_j} \\ &= -\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial y_j} \\ &= -\sum_{k=1}^{c}(t_k - z_k)f'(net_k)w_{kj}. \end{aligned} \tag{19}$$

For the second step above we had to use the chain rule yet again. The final sum over output units in Eq. 19 expresses how the hidden unit output, y_j, affects the error at each output unit. This will allow us to compute an effective target activation for each hidden unit. In analogy with Eq. 15 we use Eq. 19 to define the sensitivity for a hidden unit as

$$\delta_j \equiv f'(net_j)\sum_{k=1}^{c} w_{kj}\delta_k. \tag{20}$$

Equation 20 is the core of the solution to the credit assigment problem: The sensitivity at a hidden unit is simply the sum of the individual sensitivities at the output units weighted by the hidden-to-output weights w_{kj}, all multiplied by $f'(net_j)$. Thus the learning rule for the input-to-hidden weights is

$$\Delta w_{ji} = \eta x_i \delta_j = \eta \underbrace{\left[\sum_{k=1}^{c} w_{kj}\delta_k\right] f'(net_j)}_{\delta_j} x_i. \tag{21}$$

Equations 17 and 21, together with training protocols described below, give the backpropagation algorithm, or more specifically the "backpropagation of errors" algorithm. It is so-called because during training an error must be propagated from the output layer *back* to the hidden layer in order to perform the learning of the input-to-hidden weights by Eq. 21 (Fig. 6.5). At base then, backpropagation is just gradient descent in layered models where application of the chain rule through continuous functions allows the computation of derivatives of the criterion function with respect to all model weights.

As with all gradient-descent procedures, the exact behavior of the backpropagation algorithm depends on the starting point. While it might seem natural to start by setting the weight values to 0, Eq. 21 reveals that that would have very undesirable consequences. If the weights w_{kj} to the output layer were ever all zero, the backpropagated error would also be zero and the input-to-hidden weights would never change! This is the reason we start the process with random values for weights, as discussed further in Section 6.8.8.

These learning rules make intuitive sense. Consider first the rule for learning weights at the output units (Eq. 17). The weight update at unit k should indeed be

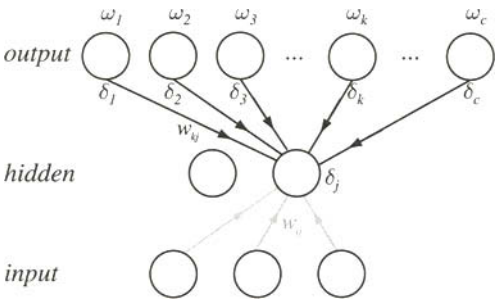


FIGURE 6.5. The sensitivity at a hidden unit is proportional to the weighted sum of the sensitivities at the output units: $\delta_j = f'(net_j) \sum_{k=1}^{c} w_{kj}\delta_k$. The output unit sensitivities are thus propagated "back" to the hidden units.

proportional to $(t_k - z_k)$: If we get the desired output $(z_k = t_k)$, then there should be no weight change. For the typical sigmoidal $f(\cdot)$ we shall use most often, $f'(net_k)$ is always positive. Thus if y_j and $(t_k - z_k)$ are both positive, then the actual output is too small and the weight must be increased; indeed, the proper sign is given by the learning rule. Finally, the weight update should be proportional to the input value; if $y_j = 0$, then hidden unit j has no effect on the output and hence the error, and thus changing w_{ji} will not change the error on the pattern presented. A similar analysis of Eq. 21 yields insight of the input-to-hidden weights (Problem 5).

Although our analysis was done for a special case of a particularly simple three-layer network, it can readily be extended to much more general networks. With moderate notational and bookkeeping effort (see Problems 7 and 11), the backpropagation learning algorithm can be generalized directly to feed-forward networks in which

- Input units include a bias unit.
- Input units are connected directly to output units as well as to hidden units.
- There are more than three layers of units.
- There are different nonlinearities for different layers.
- Each unit has its own nonlinearity.
- Each unit has a different learning rate.

Because stability problems arise, it is a more subtle matter to incorporate learning into networks having connections *within* a layer, or feedback connections from units in higher layers back to those in lower layers. We shall consider such *recurrent networks* in Section 6.10.5. First, however, we take a closer look at the convergence of the backpropagation algorithm in simpler situations.

6.3.2 Training Protocols

TRAINING SET

STOCHASTIC TRAINING

BATCH TRAINING

In broad overview, supervised training consists in presenting to the network those patterns whose category label we know—the *training set*—finding the output of the net and adjusting the weights so as to make the actual output more like the desired target values. The three most useful training protocols are: stochastic, batch, and on-line. In *stochastic training* patterns are chosen randomly from the training set, and the network weights are updated for each pattern presentation. This method is called stochastic because the training data can be considered a random variable. In *batch training*, all patterns are presented to the network before learning takes place.

ON-LINE PROTOCOL

In virtually every case we must make several passes through the training data. In *on-line* training, each pattern is presented once and only once; there is no use of memory for storing the patterns.*

EPOCH

We describe the overall amount of pattern presentations by *epoch*—where one epoch corresponds to a single presentations of all patterns in the training set. For other variables being constant, the number of epochs is an indication of the relative amount of learning.† The basic stochastic and batch protocols of backpropagation for n patterns are shown in the procedures below.

■ Algorithm 1. (Stochastic Backpropagation)

```
1 begin initialize n_H, w, criterion θ, η, m ← 0
2         do m ← m + 1
3            x^m ← randomly chosen pattern
4            w_ji ← w_ji + η δ_j x_i;  w_kj ← w_kj + η δ_k y_j
5         until ||∇J(w)|| < θ
6   return w
7 end
```

In the on-line version of backpropagation, line 3 of Algorithm 1 is replaced by sequential selection of training patterns (Problem 9). Line 5 makes the algorithm end when the change in the criterion function $J(\mathbf{w})$ is smaller than some preset value θ.

STOPPING CRITERION

While this is perhaps the simplest meaningful *stopping criterion*, others generally lead to better performance, as we shall discuss in Section 6.8.14.

In the batch training protocol, all the training patterns are presented first and their corresponding weight updates summed; only then are the actual weights in the network updated. This process is iterated until some stopping criterion is met.

So far we have considered the error on a single pattern, but in fact we want to consider an error defined over the entirety of patterns in the training set. While we have to be careful about ambiguities in notation, we can nevertheless write this total training error as the sum over the errors on n individual patterns:

$$J = \sum_{p=1}^{n} J_p. \tag{22}$$

In stochastic training, a weight update may reduce the error on the single pattern being presented, yet *increase* the error on the full training set. Given a large number of such individual updates, however, the total error as given in Eq. 22 decreases.

■ Algorithm 2. (Batch backpropagation)

```
1 begin initialize n_H, w, criterion θ, η, r ← 0
2         do r ← r + 1 (increment epoch)
3            m ← 0; Δw_ji ← 0; Δw_kj ←
```

*In Chapter 9 we shall discuss a fourth protocol, *learning with queries*, where the output of the network is used to *select* new training patterns.

†The notion of epoch does not apply to on-line training, where the number of pattern presentations is the more appropriate measure.

```
        do m ← m + 1
           x^m ← select pattern
           Δw_ji ← Δw_ji + η δ_j x_i;  Δw_kj ← Δw_kj + η δ_k y_j
        until m = n
        w_ji ← w_ji + Δw_ji;  w_kj ← w_kj + Δw_kj
     until ‖∇J(w)‖ < θ
  return w
end
```

In batch backpropagation, we need not select patterns randomly, because the weights are updated only after all patterns have been presented once. We shall consider the merits and drawbacks of each protocol in Section 6.8.

6.3.3 Learning Curves

Before training has begun, the error on the training set is typically high; through learning the error becomes lower, as shown in a *learning curve* (Fig. 6.6). The (per pattern) training error ultimately reaches an asymptotic value which depends upon the Bayes error, the amount of training data, and the expressive power (e.g., the number of weights) in the network: The higher the Bayes error and the fewer the number of such weights, the higher this asymptotic value is likely to be. Because batch backpropagation performs gradient descent in the criterion function, if the learning rate is not too high the training error tends to decrease monotonically. The average error on an independent test set is virtually always higher than on the training set, and while it generally decreases, it can increase or oscillate.

In addition to the use of the training set, there are two conceptual uses of independently selected patterns. One is to state the performance of the fielded network; for this we use the *test* set. Another use is to decide when to stop training; for this we use a *validation set*. As we shall discuss in Section 6.8.14, we stop training at a minimum of the error on the validation set.

VALIDATION SET

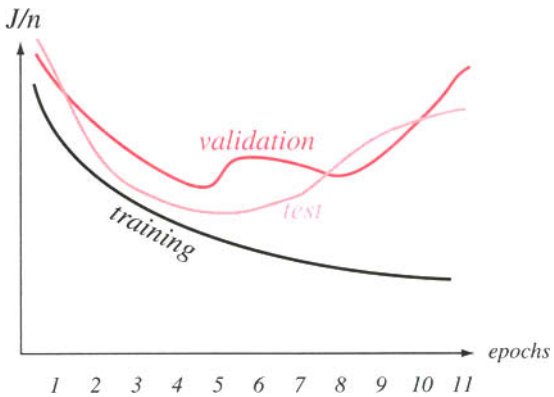


FIGURE 6.6. A learning curve shows the criterion function as a function of the amount of training, typically indicated by the number of epochs or presentations of the full training set. We plot the average error per pattern, that is, $1/n \sum_{p=1}^{n} J_p$. The validation error and the test or generalization error per pattern are virtually always higher than the training error. In some protocols, training is stopped at the first minimum of the validation set.

Figure 6.6 also shows the average error on a *validation set*—patterns not used directly for gradient descent training, and thus indirectly representative of novel patterns yet to be classified. The validation set can be used in a stopping criterion in both batch and stochastic protocols; gradient descent learning of the training set is stopped when a minimum is reached in the validation error (e.g., near epoch 5 in the figure). We shall return in Chapter 9 to understand in greater depth why this technique of *validation*, or more general *cross-validation*, often leads to networks having improved recognition accuracy.

CROSS-VALIDATION

6.4 ERROR SURFACES

Because backpropagation is based on gradient descent in a criterion function, we can gain understanding and intuition about the algorithm by studying error surfaces themselves—the function $J(\mathbf{w})$. Of course, such an error surface depends upon the classification task; nevertheless, there are some general properties of error surfaces that seem to hold over a broad range of real-world pattern recognition problems. One of the issues that concerns us is local minima; if many local minima plague the error landscape, then it is unlikely that the network will find the *global* minimum. Below we shall see whether this necessarily leads to poor performance. Another issue is the presence of plateaus—regions where the error varies only slightly as a function of weights. If such plateaus are plentiful, we can expect training according to Algorithms 1 and 2 to be slow. Because training typically begins with small weights, the error surface in the neighborhood of $\mathbf{w} \simeq \mathbf{0}$ will determine the general direction of descent. What can we say about the error in this region? Most interesting real-world problems are of high dimensionality. Are there any *general* properties of high-dimensional error functions?

We now explore these issues in some illustrative systems.

6.4.1 Some Small Networks

Consider the simplest three-layer nonlinear network, here solving a two-category problem in one dimension; this 1-1-1 sigmoidal network (and bias) is shown in Fig. 6.7. The data shown are linearly separable, and the optimal decision boundary, a point near $x_1 = 0$, separates the two categories. During learning, the weights descend to the global minimum, and the problem is solved.

Here the error surface has a *single* (global) minimum, which yields the decision point separating the patterns of the two categories. Different plateaus in the surface correspond roughly to different numbers of patterns properly classified; the maximum number of such misclassified patterns is four in this example. The plateau regions, where weight change does not lead to a change in error, here correspond to sets of weights that lead to roughly the same decision point in the input space. Thus as w_1 increases and w_2 becomes more negative, the surface shows that the error does not change, a result that can be informally confirmed by looking at the network itself.

Now consider the same network applied to another, harder, one-dimensional problem—one that is not linearly separable (Fig. 6.8). First, note that overall the error surface is slightly higher than in Fig. 6.7 because even the best solution attainable with this network leads to one pattern being misclassified. As before, the different plateaus in error correspond to different numbers of training patterns properly learned. However, one must not confuse the squared error measure with

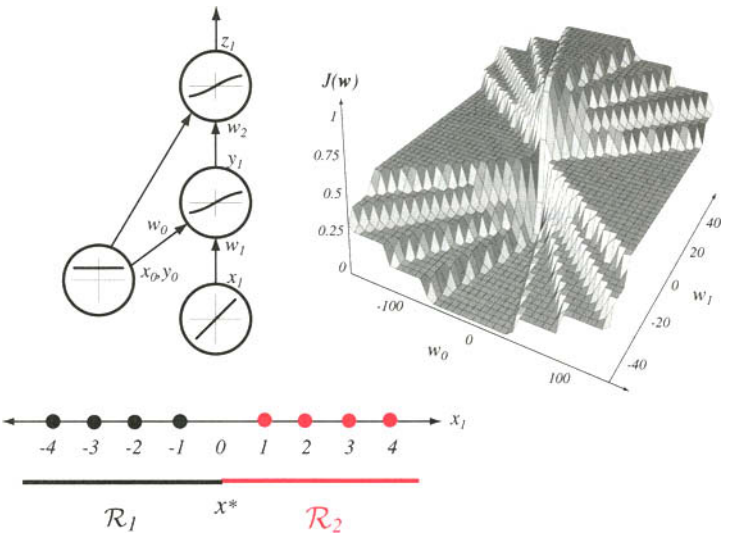


FIGURE 6.7. Eight one-dimensional patterns (four in each of two classes) are to be learned by a 1-1-1 network with steep sigmoidal hidden and output units with bias. The error surface as a function of w_0 and w_1 is also shown, where the bias weights are assigned their final values. The network starts with random weights; through stochastic training, it descends to a global minimum in error. Note especially that a low error solution exists, which indeed leads to a decision boundary separating the training points into their two categories.

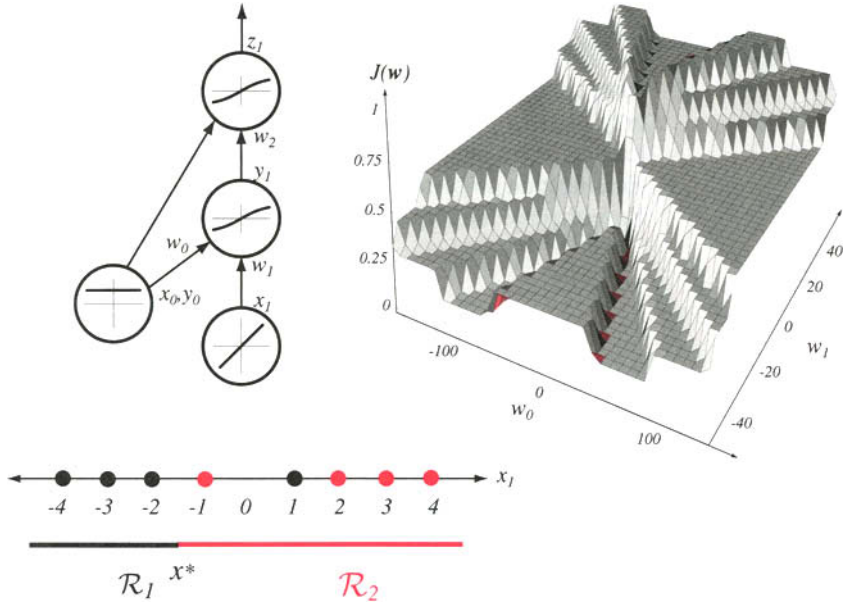


FIGURE 6.8. As in Fig. 6.7, except here the patterns are not linearly separable; the error surface is slightly higher than in that figure. Note too from the error surface that there are two forms of minimum error solution; these correspond to $-2 < x^* < -1$ and $1 < x^* < 2$, in which one pattern is misclassified.

classification error. For instance, here there are two general ways to misclassify exactly two patterns, but these have different errors. Incidentally, a 1-3-1 network (but not a 1-2-1 network) can solve this problem (Computer exercise 3).

From these very simple examples, where the correspondences among weight values, decision boundary, and error are manifest, we can see how the error of the global minimum is lower when the problem can be solved and that there are plateaus corresponding to sets of weights that lead to nearly the same decision boundary. Furthermore, the surface near $\mathbf{w} \simeq \mathbf{0}$, the traditional region for starting learning, has high error and happens in this case to have a large slope; if the starting point had differed somewhat, the network would descend to the same final weight values.

6.4.2 The Exclusive-OR (XOR)

A somewhat more complicated problem is the XOR problem we have seen before. Figure 6.9 shows several two-dimensional slices through the nine-dimensional weight space of the 2-2-1 sigmoidal network with bias. The slices shown include a global minimum in the error.

Notice first that the error varies a bit more gradually as a function of a *single* weight than does the error in the networks solving the problems in Figs. 6.7 and 6.8. This is because in a large network any single weight has on average a smaller relative contribution to the output. Ridges, valleys, and a variety of other shapes can all be seen in the surface. Several local minima in the high-dimensional weight space exist, which here correspond to solutions that classify three (but not four) patterns. Although it is hard to show it graphically, the error surface is invariant with respect to certain discrete permutations. For instance, if the labels on the two hidden units are exchanged, and the weight values changed appropriately, the shape of the error surface is unaffected (Problem 13).

6.4.3 Larger Networks

Alas, the intuition we gain from considering error surfaces for small networks gives only hints of what is going on in large networks, and at times can be quite misleading. For a network with many weights solving a complicated high-dimensional classification problem, the error varies quite gradually as a single weight is changed. Nevertheless, the surfaces can have troughs, valleys, canyons, and a host of shapes.

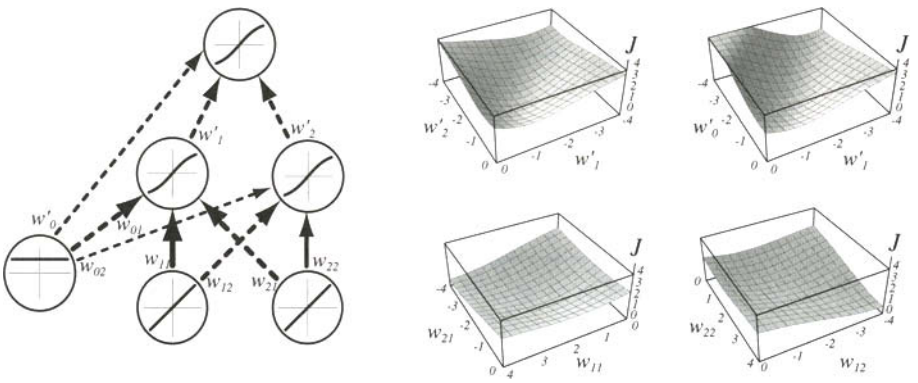


FIGURE 6.9. Two-dimensional slices through the nine-dimensional error surface after extensive training for a 2-2-1 network solving the XOR problem.

Whereas in low-dimensional spaces, local minima can be plentiful, in high dimension, the problem of local minima is different: The high-dimensional space may afford more ways (dimensions) for the system to "get around" a barrier or local maximum during learning. The more superfluous the weights, the less likely it is a network will get trapped in local minima. However, networks with an unnecessarily large number of weights are undesirable because of the dangers of overfitting, as we shall see in Section 6.11.

6.4.4 How Important Are Multiple Minima?

The possibility of the presence of multiple local minima is one reason that we resort to iterative gradient descent—analytic methods are highly unlikely to find a single global minimum, especially in high-dimensional weight spaces. In computational practice, we do not want our network to be caught in a local minimum having high training error because this usually indicates that key features of the problem have not been learned by the network. In such cases it is traditional to reinitialize the weights and train again, possibly also altering other parameters in the net (Section 6.8).

In many problems, convergence to a nonglobal minimum is acceptable, if the error is nevertheless fairly low. Furthermore, common stopping criteria demand that training terminate even before the minimum is reached, and thus it is not essential that the network be converging toward the *global* minimum for acceptable performance. In short, the presence of multiple minima does not necessarily present difficulties in training nets, and a few simple heuristics can often overcome such problems (Section 6.8).

6.5 BACKPROPAGATION AS FEATURE MAPPING

Because the hidden-to-output layer leads to a linear discriminant, the novel computational power provided by multilayer neural nets can be attributed to the nonlinear warping of the input to the representation at the hidden units. Let us consider this transformation, again with the help of the XOR problem.

Figure 6.10 shows a three-layer net addressing the XOR problem. For any input pattern in the x_1x_2-space, we can show the corresponding output of the two hidden units in the y_1y_2-space. With small initial weights, the net activation of each hidden unit is small, and thus the *linear* portion of their activation function is used. Such a linear transformation from $\mathbf{x}$ to $\mathbf{y}$ leaves the patterns linearly *in*separable (Problem 1). However, as learning progresses and the input-to-hidden weights increase in magnitude, the nonlinearities of the hidden units warp and distort the mapping from input to the hidden unit space. The linear decision boundary at the end of learning found by the hidden-to-output weights is shown by the straight dashed line; the nonlinearly separable problem at the inputs is transformed into a linearly separable at the hidden units.

We can illustrate such distortion in the three-bit parity problem, where the output is equal to $+1$ if the number of 1s in the input is odd, and -1 otherwise—a generalization of the XOR or two-bit parity problem (Fig. 6.11). As before, early in learning the hidden units operate in their linear range and thus the representation after the hidden units remains linearly *in*separable—the patterns from the two categories lie at alternating vertexes of a cube. After learning causes the weights to become larger, the nonlinearities of the hidden units are expressed and patterns have been moved and are linearly separable, as shown.

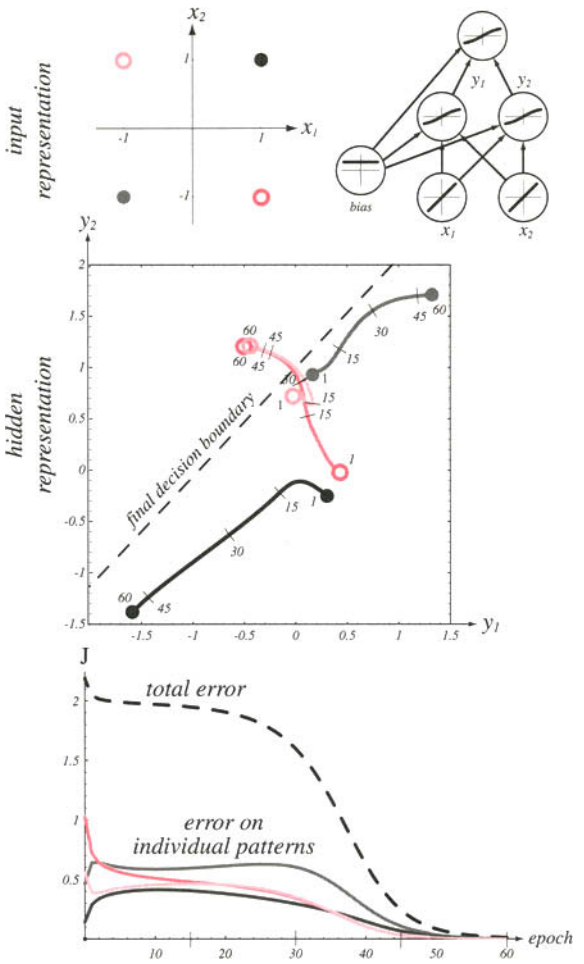


FIGURE 6.10. A 2-2-1 backpropagation network with bias and the four patterns of the XOR problem are shown at the top. The middle figure shows the outputs of the hidden units for each of the four patterns; these outputs move across the $y_1 y_2$-space as the network learns. In this space, early in training (epoch 1) the two categories are not linearly separable. As the input-to-hidden weights learn, as marked by the number of epochs, the categories become linearly separable. The dashed line is the linear decision boundary determined by the hidden-to-output weights at the end of learning; indeed the patterns of the two classes are separated by this boundary. The bottom graph shows the learning curves—the error on individual patterns and the total error as a function of epoch. Note that, as frequently happens, the total training error decreases monotonically, even though this is not the case for the error on each individual pattern.

Figure 6.12 shows a different two-dimensional two-category problem and the pattern representations in a 2-2-1 and in a 2-3-1 network of sigmoidal hidden units. Note that in the two-hidden-unit net, the categories are separated somewhat, but not enough for error-free classification; the expressive power of the net is not sufficiently high. In contrast, the three-hidden-unit net *can* separate the patterns. In general, given sufficiently many hidden units in a sigmoidal network, any set of different patterns can be learned in this way.

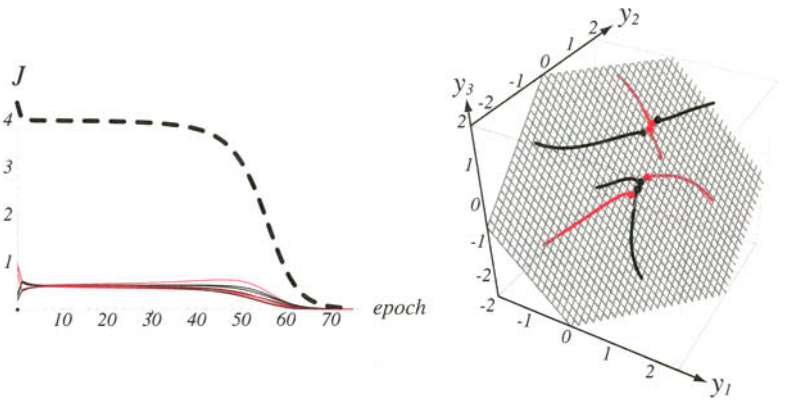


FIGURE 6.11. A 3-3-1 backpropagation network with bias can indeed solve the three-bit parity problem. The representation of the eight patterns at the hidden units ($y_1 y_2 y_3$-space) as the system learns and the planar decision boundary found by the hidden-to-output weights at the end of learning. The patterns of the two classes are indeed separated by this plane, as desired. The learning curve shows the error on individual patterns and the total error J as a function of epoch.

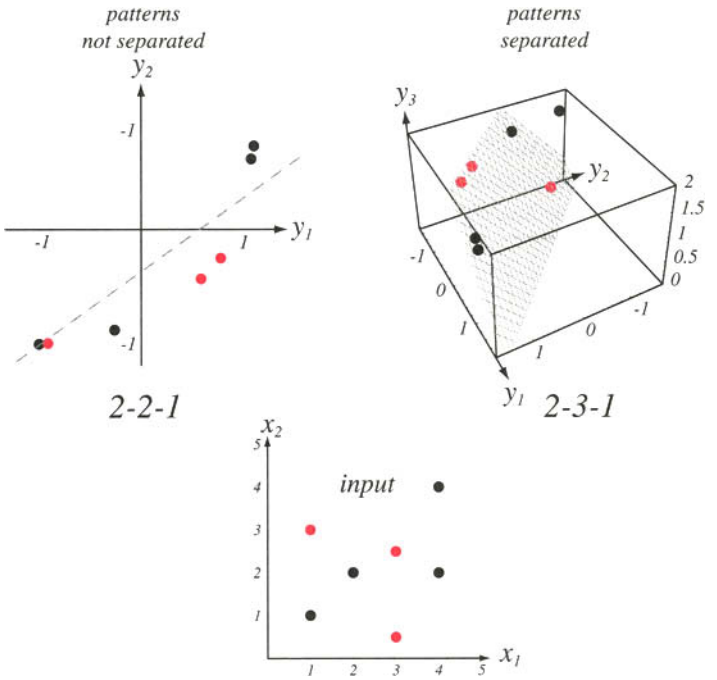


FIGURE 6.12. Seven patterns from a two-dimensional two-category nonlinearly separable classification problem are shown at the bottom. The figure at the top left shows the hidden unit representations of the patterns in a 2-2-1 sigmoidal network with bias fully trained to the global error minimum; the linear boundary implemented by the hidden-to-output weights is marked as a gray dashed line. Note that the categories are almost linearly separable in this $y_1 y_2$-space, but one training point is misclassified. At the top right is the analogous hidden unit representation for a fully trained 2-3-1 network with bias. Because of the higher dimension of the hidden layer representation, the categories are now linearly separable; indeed the learned hidden-to-output weights implement a plane that separates the categories.

6.5.1 Representations at the Hidden Layer—Weights

In addition to visualizing on the transformation of *patterns* in a network, we can also consider the representation of learned *weights*. Because the hidden-to-output weights merely lead to a linear discriminant, it is instead the input-to-hidden weights that are most instructive. In particular, such weights at a single hidden unit describe the input pattern that leads to maximum activation of that hidden unit, analogous to a "matched filter" (Section 6.10.3). Because the hidden unit activation functions are nonlinear, the correspondence with classical methods such as matched filters is not exact, however. Nevertheless, it is occasionally convenient to think of the hidden units as finding feature groupings useful for the linear classifier implemented by the hidden-to-output layer weights.

MATCHED FILTER

Figure 6.13 shows the input-to-hidden weights, displayed as images, for a simple task of character recognition. Note that one hidden unit seems "tuned" or "matched" to a pair of horizontal bars while the other is tuned to a single lower bar. Both of these feature groupings are useful building blocks for the patterns presented. In complex, high-dimensional problems, however, the pattern of learned weights may not appear to be simply related to the features we suspect are appropriate for the task. This could be because we may be mistaken about which are the true, relevant feature groupings; nonlinear interactions between features may be significant in a problem and such interactions are not manifest in the patterns of weights at a single hidden unit; or the network may have too many weights, and thus the feature selectivity is low. Thus, while analyses of learned weights may be suggestive, the whole endeavor must be approached with caution.

It is generally much harder to represent the hidden-to-output layer weights in terms of input features. Not only do the hidden units themselves already encode a somewhat abstract pattern, there is moreover no natural ordering or arrangement of the hidden units analogous to that of the input units in Fig. 6.13. Together with the

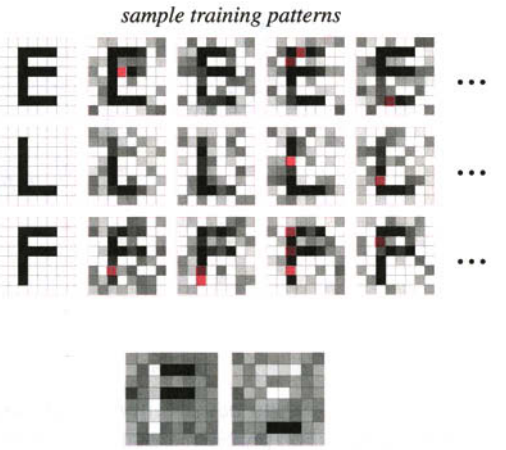


FIGURE 6.13. The top images represent patterns from a large training set used to train a 64-2-3 sigmoidal network for classifying three characters. The bottom figures show the input-to-hidden weights, represented as patterns, at the two hidden units after training. Note that these learned weights indeed describe feature groupings useful for the classification task. In large networks, such patterns of learned weights may be difficult to interpret in this way.

fact that the output of hidden units are nonlinearly related to the inputs, this makes analyzing hidden-to-output weights somewhat problematic. Often the best we can do is list the patterns of input weights for hidden units that have strong connections to the output unit in question (Computer exercise 9).

6.6 BACKPROPAGATION, BAYES THEORY AND PROBABILITY

While multilayer neural networks may appear to be somewhat ad hoc, we now show that when trained via backpropagation on a sum-squared error criterion, they provide a least squares fit to the Bayes discriminant functions.

6.6.1 Bayes Discriminants and Neural Networks

As we saw in Chapter 5, the LMS algorithm computed the approximation to the Bayes discriminant function for two-layer nets. We now generalize this result in two ways: to multiple categories and to nonlinear functions implemented by three-layer neural networks. We use the network of Fig. 6.4 and let $g_k(\mathbf{x};\ \mathbf{w})$ be the output of the kth output unit—the discriminant function corresponding to category ω_k. Recall first Bayes formula,

$$P(\omega_k|\mathbf{x}) = \frac{p(\mathbf{x}|\omega_k)P(\omega_k)}{\sum_{i=1}^{c} p(\mathbf{x}|\omega_i)p(\omega_i)} = \frac{p(\mathbf{x}, \omega_k)}{p(\mathbf{x})}, \tag{23}$$

and the Bayes decision for any pattern $\mathbf{x}$: Choose the category ω_k having the largest discriminant function $g_k(\mathbf{x}) = P(\omega_k|\mathbf{x})$.

Suppose we train a network having c output units with a target signal according to

$$t_k(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \omega_k \\ 0 & \text{otherwise.} \end{cases} \tag{24}$$

(In practice, teaching values of ± 1 are to be preferred, as we shall see in Section 6.8; we use the values 0–1 in this derivation for analytical simplicity.) The contribution to the criterion function based on a single output unit k for finite number of training samples $\mathbf{x}$ is

$$\begin{aligned} J(\mathbf{w}) &= \sum_{\mathbf{x}} [g_k(\mathbf{x};\ \mathbf{w}) - t_k]^2 \qquad (25) \\ &= \sum_{\mathbf{x}\in\omega_k} [g_k(\mathbf{x};\ \mathbf{w}) - 1]^2 + \sum_{\mathbf{x}\notin\omega_k} [g_k(\mathbf{x};\ \mathbf{w}) - 0]^2 \\ &= n\left\{ \frac{n_k}{n}\frac{1}{n_k}\sum_{\mathbf{x}\in\omega_k} [g_k(\mathbf{x};\ \mathbf{w}) - 1]^2 + \frac{n-n_k}{n}\frac{1}{n-n_k}\sum_{\mathbf{x}\notin\omega_k} [g_k(\mathbf{x};\ \mathbf{w}) - 0]^2 \right\}, \end{aligned}$$

where n is the total number of training patterns, n_k of which are in ω_k. In the limit of infinite data we can use Bayes formula to express Eq. 25 as (Problem 17)

$$\lim_{n\to\infty} \frac{1}{n} J(\mathbf{w}) \equiv \tilde{J}(\mathbf{w}) \tag{26}$$

$$= P(\omega_k) \int [g_k(\mathbf{x};\ \mathbf{w}) - 1]^2\, p(\mathbf{x}|\omega_k) d\mathbf{x} + P(\omega_{i\neq k}) \int g_k^2(\mathbf{x};\ \mathbf{w}) p(\mathbf{x}|\omega_{i\neq k})\, d\mathbf{x}$$

$$= \int g_k^2(\mathbf{x};\ \mathbf{w}) p(\mathbf{x}) d\mathbf{x} - 2 \int g_k(\mathbf{x};\ \mathbf{w}) p(\mathbf{x}, \omega_k) d\mathbf{x} + \int p(\mathbf{x}, \omega_k)\, d\mathbf{x}$$

$$= \int [g_k(\mathbf{x};\ \mathbf{w}) - P(\omega_k|\mathbf{x})]^2\, p(\mathbf{x})\, d\mathbf{x} + \underbrace{\int P(\omega_k|\mathbf{x}) P(\omega_{i\neq k}|\mathbf{x}) p(\mathbf{x}) d\mathbf{x}}_{\text{independent of } \mathbf{w}}.$$

The backpropagation rule changes weights to minimize the left-hand side of Eq. 26, and thus it minimizes

$$\int [g_k(\mathbf{x};\ \mathbf{w}) - P(\omega_k|\mathbf{x})]^2 p(\mathbf{x})\, d\mathbf{x}. \tag{27}$$

Because this is true for each category ω_k ($k = 1, 2, \ldots, c$), backpropagation minimizes the sum (Problem 22):

$$\sum_{k=1}^{c} \int [g_k(\mathbf{x};\ \mathbf{w}) - P(\omega_k|\mathbf{x})]^2\ p(\mathbf{x})\, d\mathbf{x}. \tag{28}$$

Thus in the limit of infinite data the outputs of the trained network will approximate the true *a posteriori* probabilities in a least-squares sense; that is, the output units represent the *a posteriori* probabilities,

$$g_k(\mathbf{x};\ \mathbf{w}) \simeq P(\omega_k|\mathbf{x}). \tag{29}$$

We must be cautious in interpreting these results, however. A key assumption underlying the argument is that the network can indeed represent the functions $P(\omega_k|\mathbf{x})$; with insufficient hidden units, this will not be true. Nevertheless, the above results show that neural networks can have very desirable limiting properties.

6.6.2 Outputs as Probabilities

In the previous subsection we saw one way to make the c output units of a trained net represent probabilities by training with $0 - 1$ target values. While indeed given infinite amounts of training data, the outputs will represent probabilities. If, however, these conditions do not hold—in particular we have only a *finite* amount of training data—then the outputs will not represent probabilities; for instance, there is not even a guarantee that they sum to 1.0. In fact, if the sum of the network outputs differs significantly from 1.0 within some range of the input space, it is an indication that the network is not accurately modeling the posteriors. This, in turn, may suggest changing the network topology, number of hidden units, or other aspects of the net (Section 6.8).

One approach toward approximating probabilities is to choose the output unit nonlinearity to be exponential rather than sigmoidal—$f(net_k) \propto e^{net_k}$—and for each pattern to normalize the outputs to sum to 1.0,

$$z_k = \frac{e^{net_k}}{\sum_{m=1}^{c} e^{net_m}}, \tag{30}$$

SOFTMAX
WINNER-TAKE-ALL

while training using $0-1$ target signals. This is the *softmax* method—a smoothed or continuous version of a *winner-take-all* nonlinearity in which the maximum output is transformed to 1.0, and all others reduced to 0.0. The softmax output finds theoretical justification if for each category ω_k the hidden unit representations $\mathbf{y}$ can be assumed to come from an exponential distribution (Problem 20 and Computer exercise 10).

As we have seen, then, a neural network classifier trained in this manner approximates the posterior probabilities $P(\omega_i|\mathbf{x})$, and this depends upon the prior probabilities of the categories. If such a trained network is to be used on problems in which the priors have been changed, it is a simple matter to rescale each network output, $g_i(\mathbf{x}) = P(\omega_i|\mathbf{x})$, by the ratio of such priors, though this need not ensure minimal error. The use of softmax is approriate when the network is to be used for estimating probabilities. While probabilities as network outputs can certainly be used for classification, other representations—for instance, where outputs can be positive or negative and need not sum to 1.0—are preferable, as we shall see in Section 6.8.4.

*6.7 RELATED STATISTICAL TECHNIQUES

PROJECTION PURSUIT

While the graphical, topological representation of networks is useful and a guide to intuition, we must not forget that the underlying mathematics of the feedforward operation is governed by Eq. 7. A number of statistical methods bear similarities to that equation. For instance, *projection pursuit regression*, or simply projection pursuit, implements

$$z = \sum_{j=1}^{j_{max}} w_j f_j(\mathbf{v}_j^t \mathbf{x} + v_{j0}) + w_0. \tag{31}$$

RIDGE FUNCTION

Here each $\mathbf{v}_j$ and v_{j0} together define the projection of the input $\mathbf{x}$ onto one of j_{max} different d-dimensional hyperplanes. These projections are transformed by nonlinear functions $f_j(\cdot)$ whose values are then linearly combined at the output; traditionally, sigmoidal or Gaussian functions are used. The $f_j(\cdot)$ have been called *ridge functions* because for peaked $f_j(\cdot)$, the system output appears as a ridge in a two-dimensional input space. Equation 31 implements a mapping to a scalar function z; in a c-category classification problem there would be c such outputs. In computational practice, the parameters are learned in groups minimizing an LMS error—for example, first the components of $\mathbf{v}_1$ and v_{10}, then $\mathbf{v}_2$ and v_{20} up to $\mathbf{v}_{j_{max}}$ and $v_{j_{max}0}$; then the w_j and w_0, iterating until convergence.

Such models are related to the three-layer networks we have seen in that the $\mathbf{v}_j$ and v_{j0} are analogous to the input-to-hidden weights at a hidden unit and the effective output unit is linear. The class of functions $f_j(\cdot)$ at such hidden units are more general and have more free parameters than do sigmoids. Moreover, such a model can have an output much larger than 1.0, as might be needed in a general regression task. In the classification tasks we have considered, a saturating output such as a sigmoid is more appropriate.

GENERALIZED ADDITIVE MODEL

Another technique related to multilayer neural nets is *generalized additive models*, which implement

$$z = f\left(\sum_{i=1}^{d} f_i(x_i) + w_0\right), \tag{32}$$

where again $f(\cdot)$ is often chosen to be a sigmoid, and the functions $f_i(\cdot)$ operating on the input features are nonlinear, and sometimes chosen to be sigmoidal. Such models are trained by iteratively adjusting parameters of the component nonlinearities $f_i(\cdot)$. Indeed, the basic three-layer neural networks of Section 6.2 implement a special case of general additive models (Problem 24), though the training methods differ.

MULTIVARIATE ADAPTIVE REGRESSION SPLINE

An extremely flexible technique having many adjustable parameters is *multivariate adaptive regression splines* (MARS). In this technique, localized spline functions (polynomials adjusted to ensure continuous derivative) are used in the initial processing. Here the output is the weighted sum of M products of splines:

$$z = \sum_{k=1}^{M} w_k \prod_{r=1}^{r_k} \phi_{kr}(x_{q(k,r)}) + w_0, \tag{33}$$

where the kth basis function is the product of r_k one-dimensional spline functions ϕ_{kr}, and w_0 is a scalar offset. The splines depend on the input values x_q, such as the feature component of an input, and this index is denoted $q(k, r)$. Naturally, in a c-category task, there would be one such output for each category.

In broad overview, training in MARS begins by fitting the data with a spline function along each feature dimension in turn. The spline that best fits the data, in a sum-squared-error sense, is retained. This is the $r = 1$ term in Eq. 33. Next, each of the other feature dimensions is considered, one by one. For each such dimension, candidate splines are selected based on the data fit using the *product* of that spline with the one previously selected, thereby giving the product $r = 1 \rightarrow 2$. The best such second spline is retained, thereby giving the $r = 2$ term. In this way, splines are added incrementally up to some value r_k, where some desired quality of fit is achieved. The weights w_k are learned via gradient descent on an LMS criterion.

For several reasons, multilayer neural nets have all but supplanted projection pursuit, MARS, and earlier related techniques in practical pattern recognition research. Backpropagation is simpler than learning in projection pursuit and MARS, especially when the number of training patterns and the dimension is large; heuristic information can be incorporated more simply into nets (Section 6.8.12); nets admit a variety of simplification or regularization methods (Section 6.11) that have no direct counterpart in those earlier methods. It is, moreover, usually simpler to refine a trained neural net using additional training data than it is to modify classifiers based on projection pursuit or MARS.

6.8 PRACTICAL TECHNIQUES FOR IMPROVING BACKPROPAGATION

Up to this point we have ignored a number of practical considerations for the sake of simplicity. Although the above analyses are correct, a naive application of the procedures can lead to very slow convergence, poor performance or other unsatisfactory results. Thus we now turn to a number of practical suggestions for training networks

by backpropagation. While it is difficult to give mathematical proofs for them, these suggestions can be based on a number of plausible heuristics and have been found to be useful in many practical applications.

6.8.1 Activation Function

There are a number of properties we seek for $f(\cdot)$, but we must not lose sight of the fact that backpropagation will work with virtually any activation function, given that a few simple conditions such as continuity of $f(\cdot)$ and its derivative are met. In any given classification problem we may have a good reason for selecting a particular activation function. For instance, if we have prior information that the distributions arise from a mixture of Gaussians, then Gaussian activation functions are appropriate.

When not guided by such problem dependent information, what general properties might we seek in $f(\cdot)$? First, of course, $f(\cdot)$ must be nonlinear—otherwise the three-layer network provides no computational power above that of a two-layer net (Problem 1). A second desirable property is that $f(\cdot)$ saturate—that is, have some maximum and minimum output value. This will keep the weights and activations bounded, and thus keep the training time limited. Saturation is a particularly desirable property when the output is meant to represent a probability. It is also desirable for models of biological neural networks, where the output represents a neural firing rate. It may not be desirable in networks used for regression, where a wide dynamic range may be required.

A third property is continuity and smoothness—that is, that $f(\cdot)$ and $f'(\cdot)$ be defined throughout the range of their argument. Recall that the fact that we could take a derivative of $f(\cdot)$ was crucial in the derivation of the backpropagation learning rule. The rule would not, therefore, work with the threshold or sign function of Eq. 3. Backpropagation can be made to work with *piecewise* linear activation functions, but with added complexity and few benefits.

Monotonicity is another convenient, but nonessential, property for $f(\cdot)$—we might wish that the derivative have the same sign throughout the range of the argument, for example, $f'(\cdot) \geq 0$. If f is *not* monotonic and has multiple local maxima, additional and undesirable local extrema in the error surface may become introduced. Nonmonotonic activation functions such as radial basis functions can be used if proper care is taken (Section 6.10.1). Another desirable property is linearity for a small value of *net*, which will enable the system to implement a linear model if adequate for yielding low error.

SIGMOID

One class of functions that has all the above desired properties is the *sigmoid*, such as a hyperbolic tangent. The sigmoid is smooth, differentiable, nonlinear, and saturating. It also admits a linear model if the network weights are small. A minor benefit is that the derivative $f'(\cdot)$ can be easily expressed in terms of $f(\cdot)$ itself (Problem 10).

POLYNOMIAL CLASSIFIER

We mention in passing that *polynomial classifiers* use activation functions of the form $x_1, x_2, \ldots, x_d, x_1^2, x_2^2, \ldots, x_d^2, x_1x_2, \ldots, x_1x_d$, and so forth—all terms up to some limit; training is via gradient descent too. One drawback here is that the outputs of the hidden units (φ functions) can become extremely large even for realistic problems (Problem 29). By employing saturating sigmoidal activation functions, neural networks avoid this problem.

DISTRIBUTED REPRESENTATION

A hidden layer of sigmoidal units affords a *distributed* or *global* representation of the input. That is, any particular input $\mathbf{x}$ is likely to yield activity throughout *several* hidden units. In contrast, if the hidden units have activation functions that have

LOCAL REPRESENTATION

significant response only for inputs within a small range, then an input $\mathbf{x}$ generally leads to *fewer* hidden units being active—a *local representation*. (Nearest neighbor classifiers employ local representations, of course.) It is often found in practice that when there are few training points, distributed representations are superior because more of the data influences the posteriors at any given input region.

The sigmoid is the most widely used activation function for the above reasons, and in most of the following we shall employ sigmoids.

6.8.2 Parameters for the Sigmoid

Given that we will use the sigmoidal activation function, there remain a number of parameters to set. It is best to keep the function centered on zero and antisymmetric—or as an "odd" function, that is, $f(-net) = -f(net)$—rather than one whose value is always positive. Together with the data preprocessing described in Section 6.8.3, antisymmetric sigmoids lead to faster learning. Nonzero means in the input variables and activation functions make some of the eigenvalues of the Hessian matrix large (Section 6.9.1), and this slows the learning, as we shall see.

Sigmoid functions of the form

$$f(net) = a \tanh(b\, net) = a\left[\frac{e^{+b\, net} - e^{-b\, net}}{e^{+b\, net} + e^{-b\, net}}\right] \tag{34}$$

work well. The *overall* range and slope are not important, because it is their relationship to parameters such as the learning rate and magnitudes of the inputs and targets that affect learning (Problem 23). For convenience, though, we choose $a = 1.716$ and $b = 2/3$ in Eq. 34—values that ensure $f'(0) \simeq 0.5$, that the linear range is $-1 < net < +1$, and that the extrema of the second derivative occur roughly at $net \simeq \pm 2$ (Fig. 6.14).

6.8.3 Scaling Input

Suppose we were using a two-input network to classify fish based on the features of mass (measured in grams) and length (measured in meters). Such a representation will have serious drawbacks for a neural network classifier: The numerical value of the mass will be orders of magnitude larger than that for length. During training, the network will adjust weights from the "mass" input unit far more than for the "length" input—indeed the error will hardly depend upon the tiny length values. If, however, the same physical information were presented but with mass measured in kilograms and length in millimeters, the situation would be reversed. Naturally we do not want our classifier to prefer one of these features over the other, because they differ solely in the arbitrary representation. The difficulty arises even for features having the same units but differing in overall magnitude, of course—for instance, if a fish's length and its fin thickness were both measured in millimeters.

In order to avoid such difficulties, the input patterns should be shifted so that the average over the training set of each feature is zero. Moreover, the full data set should then be scaled to have the same variance in each feature component—here chosen to be 1.0 for reasons that will be clear in Section 6.8.8. That is, we *standardize* the training patterns. This data standardization plays a role similar to a whitening transformation applied to the training set (Chapter 2), and need be done once, before the start of network training. It represents a small one-time computational burden (Problem 27). Standardization can only be done for stochastic and batch learning

STANDARDIZE

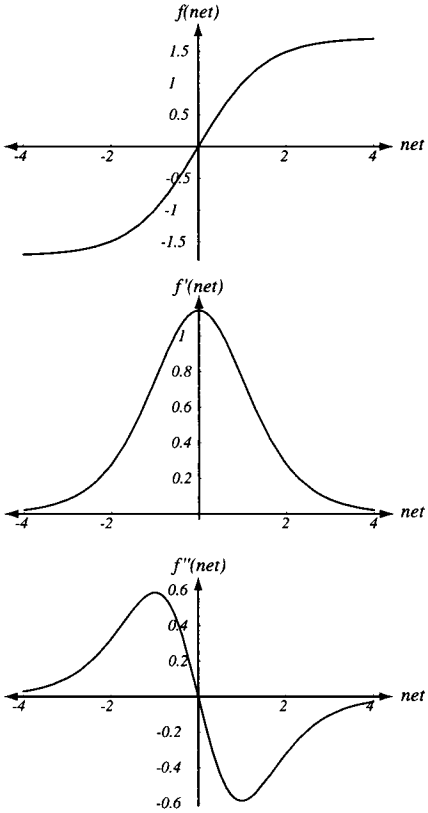


FIGURE 6.14. A useful activation function $f(net)$ is an anti-symmetric sigmoid. For the parameters given in the text, $f(net)$ is nearly linear in the range $-1 < net < +1$ and its second derivative, $f''(net)$, has extrema near $net \simeq \pm 2$.

protocols, but not on-line protocols where the full data set is never available at any one time. Naturally, any test pattern must be subjected to the same transformation before it is to be classified by the network.

6.8.4 Target Values

For pattern recognition, we typically train with the pattern and its category label, and thus we use a one-of-c representation for the target vector. Because the output units saturate at ± 1.716, we might naively feel that the target values should be those values; however, that would present a difficulty. For any finite value of net_k, the output could never reach these saturation values, and thus there would be error. Full training would never terminate because weights would become extremely large as net_k would be driven to $\pm \infty$.

This difficulty can be avoided by using teaching values of +1 for the target category and -1 for the non-target categories. For instance, in a four-category problem if the pattern is in category ω_3, the following target vector should be used: $\mathbf{t} = (-1, -1, +1, -1)^t$. Of course, this target representation yields efficient learning for categorization—the outputs here do not represent *a posterior* probabilities (Section 6.6.2).

6.8.5 Training with Noise

When the training set is small, one can generate virtual or surrogate training patterns and use them as if they were normal training patterns sampled from the source distributions. In the absence of problem-specific information, a natural assumption is that such surrogate patterns should be made by adding d-dimensional Gaussian noise to true training points. In particular, for the standardized inputs described in Section 6.8.3, the variance of the added noise should be less than 1.0 (e.g., 0.1) and the category label should be left unchanged. This method of training with noise can be used with virtually every classification method, though it generally does not improve accuracy for highly local classifiers such as ones based on the nearest neighbor.

6.8.6 Manufacturing Data

If we have knowledge about the sources of variation among patterns, for instance, due to geometrical invariances, we can "manufacture" training data that conveys more information than does the method of training with uncorrelated noise (Section 6.8.5). For instance, in an optical character recognition problem, an input image may be presented rotated by various amounts. Hence during training we can take any particular training pattern and rotate its image to "manufacture" a training point that may be representative of a much larger training set. Likewise, we might scale a pattern, perform simple image processing to simulate a bold face character, and so on. If we have information about the range of expected rotation angles, or the variation in thickness of the character strokes, we should manufacture the data accordingly.

While this method bears formal equivalence to incorporating prior information in a maximum-likelihood approach, it is usually much simpler to implement, because we need only the forward model for generating patterns. As with training with noise, manufacturing data can be used with a wide range of pattern recognition methods. A drawback is that the memory requirements may be large and overall training may be slow.

6.8.7 Number of Hidden Units

While the number of input units and output units are dictated by the dimensionality of the input vectors and the number of categories, respectively, the number of hidden units is not simply related to such obvious properties of the classification problem. The number of hidden units, n_H, governs the expressive power of the net—and thus the complexity of the decision boundary. If the patterns are well-separated or linearly separable, then few hidden units are needed; conversely, if the patterns are drawn from complicated densities that are highly interspersed, then more hidden units are needed. Without further information there is no foolproof method for setting the number of hidden units before training.

Figure 6.15 shows the training and test error on a two-category classification problem for networks that differ solely in their number of hidden units. For large n_H, the training error can become small because such networks have high expressive power and become tuned to the particular training set. Nevertheless, in this regime, the test error is unacceptably high, an example of overfitting we shall study again in Chapter 9. At the other extreme of too few hidden units, the net does not have enough free parameters to fit the training data well, and again the test error is high. We seek some intermediate number of hidden units that will give low test error.

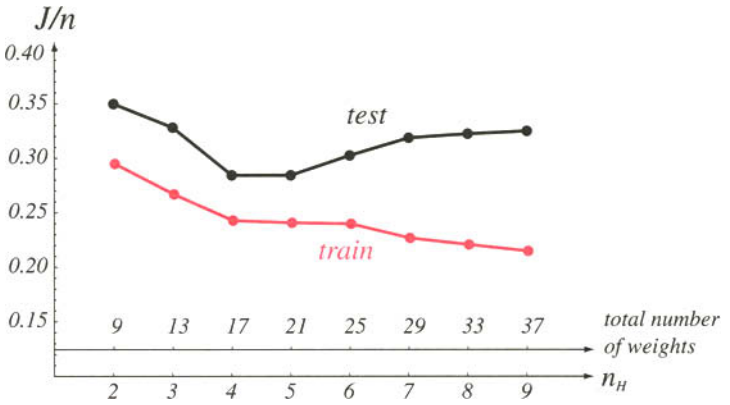


FIGURE 6.15. The error per pattern for networks fully trained but differing in the numbers of hidden units, n_H. Each $2 - n_H - 1$ network with bias was trained with 90 two-dimensional patterns from each of two categories, sampled from a mixture of three Gaussians, and thus $n = 180$. The minimum of the test error occurs for networks in the range $4 \leq n_H \leq 5$, i.e., the range of weights 17 to 21. This illustrates the rule of thumb that choosing networks with roughly $n/10$ weights often gives low test error.

The number of hidden units determines the total number of weights in the net—which we consider informally as the number of degrees of freedom—and thus it is plausible that we should not have more weights than the total number of training points, n. A convenient rule of thumb is to choose the number of hidden units such that the total number of weights in the net is roughly $n/10$. This seems to work well over a range of practical problems. It must be noted, however, that many successful systems employ more than this number. A more principled method is to adjust the complexity of the network in response to the training data, for instance, start with a "large" number of hidden units and "decay," prune, or eliminate weights—techniques we shall study in Section 6.11 and in Chapter 9.

6.8.8 Initializing Weights

UNIFORM LEARNING

First, we can see from Eq. 21 that we cannot initialize the weights to 0, otherwise learning cannot take place. Thus we must confront the problem of choosing their starting values. Suppose we have fixed the network topology and thus have set the number of hidden units. We now seek to set the initial weight values in order to have fast and *uniform learning*, that is, all weights reach their final equilibrium values at about the same time. One form of nonuniform learning occurs when one category is learned well before others. In this undesirable case, the distribution of errors differs markedly from Bayes, and the overall error rate is typically higher than necessary. (The data standarization described above also helps to ensure uniform learning.)

In setting weights in a given layer, we choose weights randomly from a *single* distribution to help ensure uniform learning. Because data standardization gives positive and negative values equally, on average, we want positive and negative weights as well; thus we choose weights from a uniform distribution $-\tilde{w} < w < +\tilde{w}$, for some $\tilde{w}$ yet to be determined. If $\tilde{w}$ is chosen too small, the net activation of a hidden unit will be small and the linear model will be implemented. Alternatively, if $\tilde{w}$ is too large, the hidden unit may saturate even before learning begins. Because $net_j \simeq \pm 1$ are the limits to its linear range, we set $\tilde{w}$ such that the net activation at a hidden unit is in the range $-1 < net_j < +1$ (Fig. 6.14).

In order to calculate $\tilde{w}$, we consider a hidden unit accepting input from d input units. Suppose too that the same distribution is used to initialize all the weights, namely, a uniform distribution in the range $-\tilde{w} < w < +\tilde{w}$. On average, then, the net activation from d random variables of variance 1.0 from our standarized input through such weights will be $\tilde{w}\sqrt{d}$. As mentioned, we would like this net activation to be roughly in the range $-1 < net < +1$. This implies that $\tilde{w} = 1/\sqrt{d}$; thus input weights should be chosen in the range $-1/\sqrt{d} < w_{ji} < +1/\sqrt{d}$. The same argument holds for the hidden-to-output weights, where here the number of connected units is n_H; hidden-to-output weights should be initialized with values chosen in the range $-1/\sqrt{n_H} < w_{kj} < +1/\sqrt{n_H}$.

6.8.9 Learning Rates

NONUNIFORM LEARNING

In principle, so long as the learning rate is small enough to ensure convergence, its value determines only the speed at which the network attains a minimum in the criterion function $J(\mathbf{w})$, not the final weight values themselves. In practice, however, because networks are rarely trained fully to a training error minimum (Section 6.8.14), the learning rate can indeed affect the quality of the final network. If some weights converge significantly earlier than others (nonuniform learning) then the network may not perform equally well throughout the full range of inputs, or equally well for the patterns in each category. Figure 6.16 shows the effect of different learning rates on convergence in a single dimension.

The optimal learning rate is the one that leads to the local error minimum in one learning step. A principled method of setting the learning rate comes from assuming the criterion function can be reasonably approximated by a quadratic, and this gives

$$\frac{\partial^2 J}{\partial w^2}\Delta w = \frac{\partial J}{\partial w}, \tag{35}$$

as illustrated in Fig. 6.17. The optimal rate is found directly to be

$$\eta_{opt} = \left(\frac{\partial^2 J}{\partial w^2}\right)^{-1}. \tag{36}$$

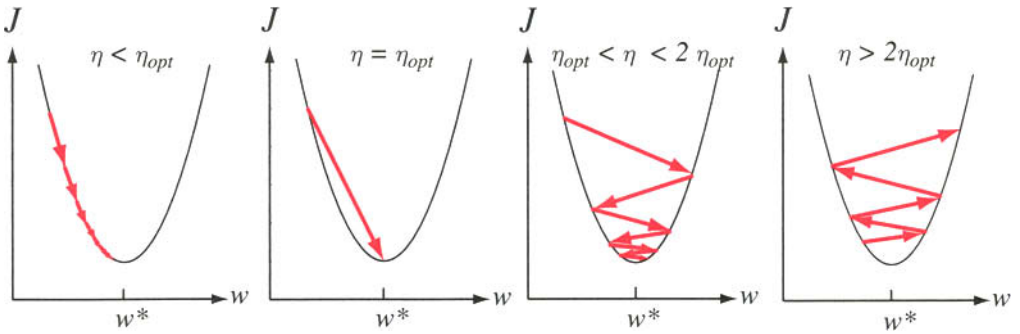


FIGURE 6.16. Gradient descent in a one-dimensional quadratic criterion with different learning rates. If $\eta < \eta_{opt}$, convergence is assured, but training can be needlessly slow. If $\eta = \eta_{opt}$, a single learning step suffices to find the error minimum. If $\eta_{opt} < \eta < 2\eta_{opt}$, the system will oscillate but nevertheless converge, but training is needlessly slow. If $\eta > 2\eta_{opt}$, the system diverges.

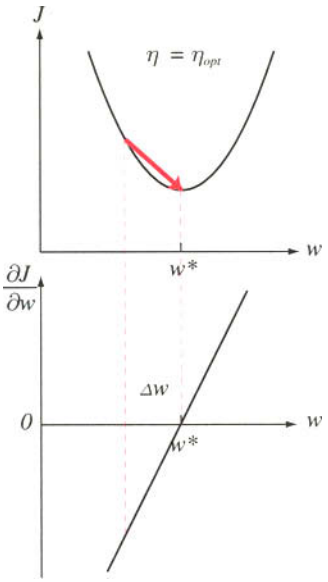


FIGURE 6.17. If the criterion function is quadratic (above), its derivative is linear (below). The optimal learning rate η_{opt} ensures that the weight value yielding minimum error, w^*, is found in a single learning step.

Of course the maximum learning rate that will give convergence is $\eta_{max} = 2\eta_{opt}$. It should be noted that a learning rate η in the range $\eta_{opt} < \eta < 2\eta_{opt}$ will lead to slower convergence (Computer exercise 8).

Thus, for rapid and uniform learning, we should calculate the second derivative of the criterion function with respect to *each* weight and set the optimal learning rate separately for each weight. We shall return in Section 6.9 to calculate second derivatives in networks, and to alternative descent and training methods. For typical problems addressed with sigmoidal networks and parameters discussed throughout this section, it is found that a learning rate of $\eta \simeq 0.1$ is often adequate as a first choice. The learning rate should be lowered if the criterion function diverges during learning, or instead should be raised if learning seems unduly slow.

6.8.10 Momentum

Error surfaces often have plateaus—regions in which the slope $dJ(\mathbf{w})/d\mathbf{w}$ is very small. These can arise when there are "too many" weights and thus the error depends only weakly upon any one of them. Momentum—loosely based on the notion from physics that moving objects tend to keep moving unless acted upon by outside forces—allows the network to learn more quickly when plateaus in the error surface exist. The approach is to alter the learning rule in stochastic backpropagation to include some fraction α of the previous weight update. Let $\Delta\mathbf{w}(m) = \mathbf{w}(m) - \mathbf{w}(m-1)$, and let $\Delta\mathbf{w}_{bp}(m)$ be the change in $\mathbf{w}(m)$ that would be called for by the backpropagation algorithm. Then

$$\mathbf{w}(m+1) = \mathbf{w}(m) + (1-\alpha)\Delta\mathbf{w}_{bp}(m) + \alpha\Delta\mathbf{w}(m-1) \tag{37}$$

represents learning with momentum.

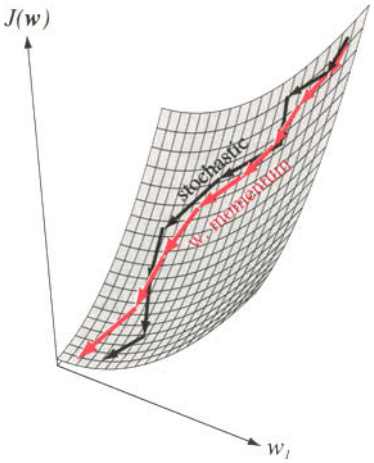


FIGURE 6.18. The incorporation of momentum into stochastic gradient descent by Eq. 37 (red arrows) reduces the variation in overall gradient directions and speeds learning.

Those who are familiar with digital signal processing will recognize this as a recursive or infinite-impulse-response low-pass filter that smooths the changes in $\mathbf{w}$. Obviously, α should not be negative, and for stability α must be less than 1.0. If $\alpha = 0$, the algorithm is the same as standard backpropagation. If $\alpha = 1$, the change suggested by backpropagation is ignored, and the weight vector moves with constant velocity. The weight changes are response to backpropagation if α is small, and sluggish if α is large. (Values typically used are $\alpha \simeq 0.9$.) Thus, the use of momentum "averages out" stochastic variations in weight updates during stochastic learning. By increasing stability, it can speed the learning process, even far from error plateaus (Fig. 6.18).

Algorithm 3 shows one way to incorporate momentum into gradient descent.

■ **Algorithm 3. (Stochastic Backpropagation with Momentum)**

```
1 begin initialize n_H, w, α(< 1), θ, η, m ← 0, b_ji ← 0, b_kj ← 0
2         do m ← m + 1
3            x^m ← randomly chosen pattern
4            b_ji ← η(1 − α)δ_j x_i + αb_ji; b_kj ← η(1 − α)δ_k y_j + αb_kj
5            w_ji ← w_ji + b_ji; w_kj ← w_kj + b_kj
6         until ‖∇J(w)‖ < θ
7   return w
8 end
```

6.8.11 Weight Decay

One method of simplifying a network and avoiding overfitting is to impose a heuristic that the weights should be small. There is no principled reason why such a method of "weight decay" should always lead to improved network performance (indeed there are occasional cases where it leads to *degraded* performance), but it is found in most

cases that it helps. The basic approach is to start with a network with "too many" weights and "decay" all weights during training. Small weights favor models that are more nearly linear (Problems 1 and 41). One of the reasons weight decay is so popular is its simplicity. After each weight update, every weight is simply "decayed" or shrunk according to

$$w^{new} = w^{old}(1 - \epsilon), \tag{38}$$

where $0 < \epsilon < 1$. In this way, weights that are not needed for reducing the error function become smaller and smaller, possibly to such a small value that they can be eliminated altogether. Those weights that *are* needed to solve the problem will not decay indefinitely. In weight decay, then, the system achieves a balance between pattern error (Eq. 67) and some measure of overall weight. It can be shown (Problem 42) that the weight decay is equivalent to gradient descent in a new effective error or criterion function:

$$J_{ef} = J(\mathbf{w}) + \frac{2\epsilon}{\eta}\mathbf{w}^t\mathbf{w}. \tag{39}$$

The second term on the right-hand side of Eq. 39 is sometimes called a regularization term (cf. Eq. 67) and in this case preferentially penalizes a single large weight. Another version of weight decay includes a decay parameter that depends upon the value of the weight itself, and this tends to distribute the penalty throughout the network:

$$J_{ef} = J(\mathbf{w}) + \frac{2\epsilon}{\eta}\sum_{i,j}\frac{w_{ij}^2/(\mathbf{w}^t\mathbf{w})}{1 + w_{ij}^2/(\mathbf{w}^t\mathbf{w})}. \tag{40}$$

6.8.12 Hints

Often we have insufficient training data for the desired classification accuracy and we would like to add information or constraints to improve the network. The approach of learning with *hints* is to add output units for addressing an ancillary problem, one different from but related to the specific classification problem at hand. The expanded network is trained on the classification problem of interest *and* the ancillary one, possibly simultaneously. For instance, suppose we seek to train a network to classify c phonemes based on some acoustic input. In a standard neural network we would have c output units. In learning with hints, we might add two ancillary output units, one which represents vowels and the other consonants. During training, the target vector must be lengthened to include components for the hint outputs. During classification, the hint units are not used; they and their hidden-to-output weights can be discarded (Fig. 6.19).

The benefit provided by hints is in improved feature selection. So long as the hints are related to the classification problem at hand, the feature groupings useful for the hint task are likely to aid category learning. For instance, the feature groupings useful for distinguishing vowel sounds from consonants in general are likely to be useful for distinguishing the /b/ from /oo/ or the /g/ from /ii/ categories in particular. Alternatively, one can train just the hint units in order to develop improved hidden unit representations.

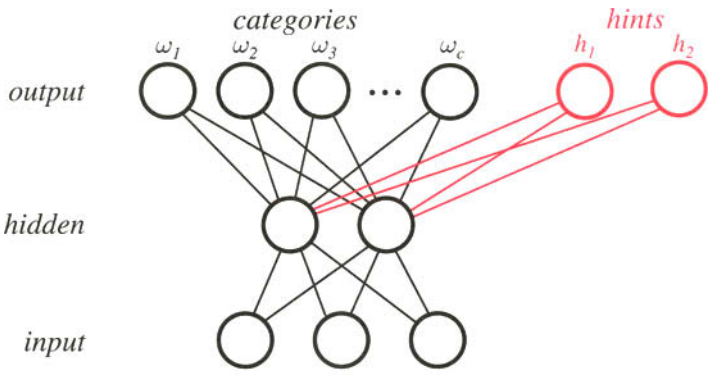


FIGURE 6.19. In learning with hints, the output layer of a standard network having c category units is augmented with hint units. During training, the target vectors are also augmented with signals for the hint units. In this way the input-to-hidden weights learn improved feature groupings. The hint units are not used during classification, and thus they and their hidden-to-output weights are removed from the trained network (red).

Learning with hints illustrates another benefit of neural networks: Hints are more easily incorporated into neural networks than into classifiers based on other algorithms, such as the nearest-neighbor or MARS.

6.8.13 On-Line, Stochastic or Batch Training?

Each of the three leading training protocols described in Section 6.3.2 has strengths and drawbacks. On-line learning is to be used when the amount of training data is so large, or when memory costs are so high, that storing the data is prohibitive. Most practical neural network classification problems are addressed instead with batch or stochastic protocols.

Batch learning is typically slower than stochastic learning. To see this, imagine a training set of 50 patterns that consists of 10 copies each of five patterns $(\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^5)$. In batch learning, the presentations of the duplicates of $\mathbf{x}^1$ provide as much information as a single presentation of $\mathbf{x}^1$ in the stochastic case. For example, suppose that in the batch case the learning rate is set optimally. The same weight change can be achieved with just a *single* presentation of each of the five different patterns in the batch case, so long as the learning rate is set correspondingly higher. Of course, true problems do not have exact duplicates of individual patterns; nevertheless, actual data sets are generally highly redundant, and the above analysis holds.

For most applications—especially ones employing large redundant training sets—stochastic training is hence to be preferred. Batch training admits some second-order techniques that cannot be easily incorporated into stochastic learning protocols and hence in some problems should be preferred, as we shall see in Section 6.9.

6.8.14 Stopped Training

In three-layer networks having many weights, excessive training can lead to poor generalization as the net implements a complex decision boundary "tuned" to the specific training data rather than the general properties of the underlying distributions. In training the two-layer networks of Chapter 5, we can usually train as long

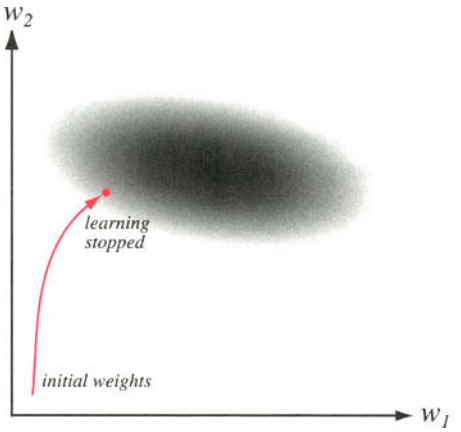


FIGURE 6.20. When weights are initialized with small magnitudes, stopped training leads to final weights that are smaller than they would be after extensive training. As such, stopped training behaves much like a form of weight decay.

as we like without fear that it would degrade final recognition accuracy because the *complexity* of the decision boundary is not changed—it is always simply a hyperplane. For this reason the general phenomenon should be called "overfitting," not "overtraining."

Because the network weights are initialized with small values, the units operate in their linear range and the full network implements linear discriminants. As training progresses, the nonlinearities of the units are expressed and the decision boundary warps. Qualitatively speaking, then, stopping the training before gradient descent is complete can help avoid overfitting. In practice, it is hard to know beforehand what stopping criterion θ should be set in line 5 of Algorithm 1. A far simpler method is to stop training when the error on a separate validation set reaches a minimum (Fig. 6.6). In Chapter 9 we shall explore the theory underlying this technique of validation and more generally cross-validation. We note in passing that weight decay behaves much like a form of stopped training (Fig. 6.20).

6.8.15 Number of Hidden Layers

The backpropagation algorithm applies equally well to networks with three, four, or more layers, so long as the units in such layers have differentiable activation functions. Because, as we have seen, three layers suffice to implement any arbitrary function, we would need special problem conditions or requirements to recommend the use of more than three layers.

One possible such requirement is translation, rotation or other distortion invariances. If the input layer represents the pixel image in an optical character recognition problem, we generally want such a recognizer to be invariant with respect to such transformations. It is easier for a four-layer net to learn translations than for a three-layer net. This is because each layer can generally easily learn an invariance within a limited range of parameters—for instance, a lateral shift of just two pixels. Stacking multiple layers, then, allows the full network to learn shifts of up to four pixels as the full invariance task is distributed throughout the net. Naturally, the weight initialization, learning rate, and data preprocessing arguments apply to these networks too. Some functions can be implemented more efficiently (i.e., with fewer total units) in

networks with more than one hidden layer. It has been found empirically that networks with multiple hidden layers are more prone to getting caught in undesirable local minima, however.

In the absence of a problem-specific reason for multiple hidden layers, then, it is simplest to proceed using just a single hidden layer, but also to try two hidden layers if necessary.

6.8.16 Criterion Function

The squared error of Eq. 9 is the most common training criterion because it is simple to compute, is nonnegative, and simplifies the proofs of some theorems. Nevertheless, other training criteria occasionally have benefits. One popular alternative is the cross entropy which measures a "distance" between probability distributions. The cross entropy for n patterns is of the form:

$$J_{ce}(\mathbf{w}) = \sum_{m=1}^{n}\sum_{k=1}^{c} t_{mk}\ln(t_{mk}/z_{mk}), \tag{41}$$

where t_{mk} and z_{mk} are the target and the actual output of unit k for pattern m. Of course, to interpret J as an entropy, the target and output values must be interpreted as probabilities and must fall between 0 and 1.

MINKOWSKI ERROR

Yet another criterion function is based on the *Minkowski error*:

$$J_{Mink}(\mathbf{w}) = \sum_{m=1}^{n}\sum_{k=1}^{c} |z_{mk}(\mathbf{x}) - t_{mk}(\mathbf{x})|^R, \tag{42}$$

much as we saw in Chapter 4. It is a straightforward matter to derive the backpropagation rule for this error (Problem 30). While in general the rule is a bit more complex than for the ($R = 2$) sum-squared error we have considered, the Minkowski error for $1 \leq R < 2$ reduces the influence of long tails in the distributions—tails that may be quite far from the category decision boundaries. As such, the designer can adjust the "locality" of the classifier indirectly through choice of R; the smaller the R, the more local the classifier.

Most of the heuristics described in this section can be used alone or in combination. While they may interact in unexpected ways, all have found use in important pattern recognition problems and classifier designers should have experience with all of them.

*6.9 SECOND-ORDER METHODS

We have used a second-order analysis of the error in order to determine the optimal learning rate. We can use second-order information more fully in other ways, including the elimination of unneeded weights in a network.

6.9.1 Hessian Matrix

We derived the first-order derivatives of a sum-squared-error criterion function in three-layer networks, summarized in Eqs. 17 and 21. We now turn to second-order derivatives, which find use in rapid learning methods as well as in some pruning

or regularization algorithms. We treat the familiar sum-squared-error criterion for a network with a single output,

$$J(\mathbf{w}) = \frac{1}{2}\sum_{m=1}^{n}(t_m - z_m)^2, \tag{43}$$

where t_m and z_m are the target and output signals, and n the total number of training patterns. The elements in the Hessian matrix are thus

$$\frac{\partial^2 J(\mathbf{w})}{\partial w_{ji}\partial w_{lk}} = \frac{1}{n}\left(\sum_{m=1}^{n}\frac{\partial J}{\partial w_{ji}}\frac{\partial J}{\partial w_{lk}} + \underbrace{\sum_{m=1}^{n}(z-t)\frac{\partial^2 J}{\partial w_{ji}\partial w_{lk}}}_{O(\|\mathbf{t}-\mathbf{z}\|)}\right), \tag{44}$$

where we have used the subscripts to refer to *any* weight in the network; hence i, j, l, and k could all take on values that describe input-to-hidden weights, or that describe hidden-to-output weights, or mixtures. Of course the Hessian matrix is symmetric. The second term in Eq. 44 is of order $O(\|\mathbf{t} - \mathbf{z}\|)$, which is generally small and thus often neglected. This approximation guarantees that the resulting approximation is positive definite and thus gradient descent will progress. In this *outer product approximation* our Hessian reduces to:

$$\mathbf{H} = \frac{1}{n}\sum_{m=1}^{n}\mathbf{X}^{[m]t}\mathbf{X}^{[m]} \tag{45}$$

where the superscript $[m]$ indexes the pattern and $\mathbf{X} = \partial J/\partial \mathbf{w}$ can be broken into two parts as

$$\mathbf{X} = \begin{pmatrix} \mathbf{X}_u \\ \mathbf{X}_v \end{pmatrix}. \tag{46}$$

Here $\mathbf{X}_v$ refers to derivatives with respect to the hidden-to-output weights, and $\mathbf{X}_u$ refers to the input-to-hidden weights. For a $d - n_H - 1$ neuron three-layer network, these vectors of derivatives can be written (Problem 31)

$$\mathbf{X}_v^t = (f'(net)y_1, \ldots, f'(net)y_{n_H}) \tag{47}$$

and

$$\mathbf{X}_u^t = (f'(net)f'(net_1)y_1x_1, \ldots, f'(net)f'(net_{n_H}x_d)) \tag{48}$$

Equations 45, 47 and 48, enable the approximation to the Hessian be computed in a straightforward fashion.

6.9.2 Newton's Method

We can use a Taylor series to describe the change in the criterion function due to a change in weights $\Delta\mathbf{w}$ as

$$\begin{aligned}\Delta J(\mathbf{w}) &= J(\mathbf{w}+\Delta\mathbf{w}) - J(\mathbf{w}) \\ &\simeq \left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}\right)^t \Delta\mathbf{w} + \frac{1}{2}\Delta\mathbf{w}^t\mathbf{H}\Delta\mathbf{w},\end{aligned} \tag{49}$$

where $\mathbf{H}$ is the Hessian matrix. We differentiate Eq. 49 with respect to $\Delta\mathbf{w}$ and find that $\Delta J(\mathbf{w})$ is minimized for

$$\left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}\right) + \mathbf{H}\Delta\mathbf{w} = \mathbf{0}. \tag{50}$$

Therefore, the optimum change in weights can be expressed as

$$\Delta\mathbf{w} = -\mathbf{H}^{-1}\left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}\right), \tag{51}$$

as we saw in Fig. 6.17. Thus, in gradient descent using Newton's algorithm, if we have an estimate for the weights at iteration m (that is, $\mathbf{w}(m)$), we can get an improved estimate using the weight change given by Eq. 51, that is,

$$\begin{aligned}\mathbf{w}(m+1) &= \mathbf{w}(m) + \Delta\mathbf{w} \\ &= \mathbf{w}(m) - \mathbf{H}^{-1}(m)\left(\frac{\partial J(\mathbf{w}(m))}{\partial \mathbf{w}}\right),\end{aligned} \tag{52}$$

In Newton's algorithm we iteratively recompute $\mathbf{w}$ according to Eq. 52.

Alas, there are several drawbacks to this simple version of Newton's algorithm. The first is that for a network having N weights, the algorithm requires computing, storing and inverting the $N \times N$ Hessian matrix, which has complexity $O(N^3)$, making it impractical for all but small problems. Second, and more severe, is the fact that the algorithm need not converge in nonquadratic error surfaces, which often occur in practice. Nevertheless, an understanding of Newton's algorithm is excellent background for understanding more sophisticated methods, such as conjugate gradient descent (Section 6.9.4).

6.9.3 Quickprop

One of the simplest methods for using second-order information to increase training speed is the Quickprop algorithm. In this method, the weights are assumed to be independent, and the descent is optimized separately for each. The error surface is assumed to be quadratic and the coefficients for the particular parabola are determined by two successive evaluations of $J(w)$ and $dJ(w)/dw$. The single weight w is then moved to the computed minimum of the parabola (Fig. 6.21). It can be shown (Problem 35) that this approach leads to the following weight update rule,

$$\Delta w(m+1) = \frac{\left.\frac{dJ}{dw}\right|_m}{\left.\frac{dJ}{dw}\right|_{m-1} - \left.\frac{dJ}{dw}\right|_m}\Delta w(m). \tag{53}$$

where the derivatives are evaluated at iterations m and $m-1$, as indicated.

If the third- and higher-order terms in the error are nonnegligible, or if the assumption of weight independence does not hold, then the computed error minimum will not equal the true minimum, and further weight updates will be needed. When a

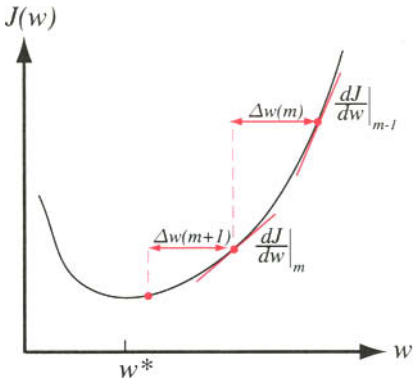


FIGURE 6.21. The quickprop weight update takes the error derivatives at two points separated by a known amount, and by Eq. 53 computes the next weight value. If the error can be fully expressed as a second-order function, then the weight update leads to the weight (w^*) leading to minimum error.

number of obvious heuristics are imposed—to reduce the effects of estimation error when the surface is nearly flat, or the step actually increases the error—the method can be significantly faster than standard backpropagation. Another benefit is that each weight has, in effect, its own learning rate, and thus weights tend to converge at roughly the same time, thereby reducing problems due to nonuniform learning.

6.9.4 Conjugate Gradient Descent

Another fast learning method is conjugate gradient descent, which employs a series of line searches in weight or parameter space. One picks the first descent direction (for instance, the simple gradient) and moves along that direction until the local minimum in error is reached. The second descent direction is then computed: This direction—the "conjugate direction"—is the one along which the gradient does not change its *direction*, but merely its magnitude during the next descent. Descent along this direction will not "spoil" the contribution from the previous descent iterations (Fig. 6.22).

More specifically, we let $\Delta\mathbf{w}(m-1)$ represent the *direction* of a line search on step $m-1$. Note especially that this is not an overall *magnitude* of change, which will be determined by the line search. We demand that the subsequent direction, $\Delta\mathbf{w}(m)$, obey

$$\Delta\mathbf{w}^t(m-1)\mathbf{H}\Delta\mathbf{w}(m) = 0, \tag{54}$$

where $\mathbf{H}$ is the Hessian matrix. Pairs of descent directions that obey Eq. 54 are called "conjugate." If the Hessian is proportional to the identity matrix, then such directions are orthogonal in weight space. Conjugate gradient requires batch training, because the Hessian matrix is defined over the full training set.

The descent direction on iteration m is in the direction of the gradient plus a component along the previous descent direction:

$$\Delta\mathbf{w}(m) = -\nabla J(\mathbf{w}(m)) + \beta_m \Delta\mathbf{w}(m-1); \tag{55}$$

the relative proportions of these contributions is governed by β_m. This proportion can be derived by ensuring that the descent direction on iteration m does not spoil that

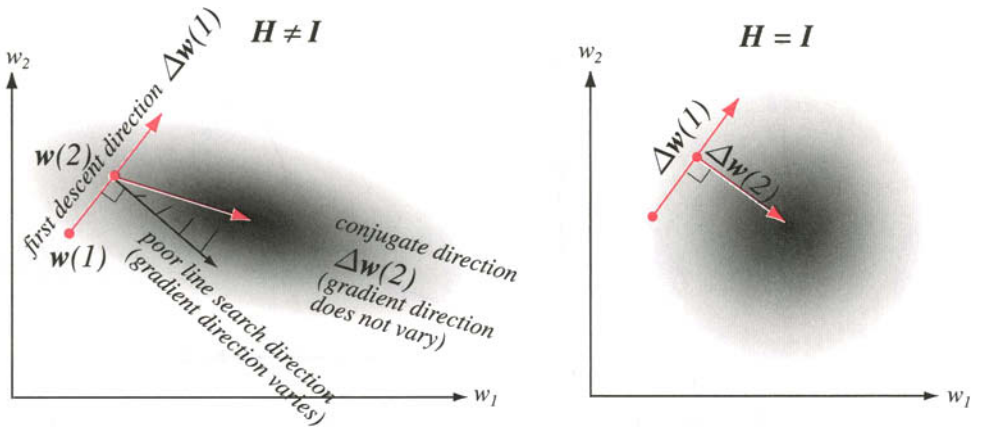


FIGURE 6.22. Conjugate gradient descent in weight space employs a sequence of line searches. If $\Delta\mathbf{w}(1)$ is the first descent direction, the second direction obeys $\Delta\mathbf{w}^t(1)\mathbf{H}\Delta\mathbf{w}(2) = 0$. Note especially that along this second descent, the gradient changes only in magnitude, not direction; as such, the second descent does not "spoil" the contribution due to the previous line search. In the case where the Hessian is diagonal (right), the directions of the line searches are orthogonal.

from direction $m - 1$, and indeed all earlier directions. It is generally calculated in one of two ways. The first formula (Fletcher-Reeves) is

$$\beta_m = \frac{\nabla J^t(\mathbf{w}(m))\nabla J(\mathbf{w}(m))}{\nabla J^t(\mathbf{w}(m-1))\nabla J(\mathbf{w}(m-1))}. \tag{56}$$

A slightly preferable formula (Polak-Ribiere) that is more robust in nonquadratic error functions is

$$\beta_m = \frac{\nabla J^t(\mathbf{w}(m))[\nabla J(\mathbf{w}(m)) - \nabla J(\mathbf{w}(m-1))]}{\nabla J^t(\mathbf{w}(m-1))\nabla J(\mathbf{w}(m-1))}. \tag{57}$$

Equations 55 and 37 show that conjugate gradient descent algorithm is analogous to calculating a "smart" momentum, where β_m plays the role of a momentum. If the error function is quadratic, then the convergence of conjugate gradient descent is guaranteed when the number of iterations equals the total number of weights.

EXAMPLE 1 Conjugate Gradient Descent

Consider finding the minimum of a simple quadratic criterion function centered on the origin of weight space, $J(\mathbf{w}) = 1/2(.2w_1^2 + w_2^2) = 1/2\mathbf{w}^t\mathbf{H}\mathbf{w}$, where by simple differentiation the Hessian is found to be $\mathbf{H} = \begin{pmatrix} .2 & 0 \\ 0 & 1 \end{pmatrix}$. We start descent at a randomly selected position, which for this figure happens to be $\mathbf{w}(0) = \begin{pmatrix} -8 \\ -4 \end{pmatrix}$. The first descent direction is determined by a simple gradient, which is easily found to be $-\nabla J(\mathbf{w}(0)) = -\begin{pmatrix} .4w_1(0) \\ 2w_2(0) \end{pmatrix} = \begin{pmatrix} 3.2 \\ 8 \end{pmatrix}$. In typical complex problems in high dimensions, the minimum along this direction is found using a line search; in this simple case the minimum can be found by calculus. We let s represent the distance along the first descent direction, and we find its value for the minimum of $J(\mathbf{w})$ according to

$$\frac{d}{ds}\left[\left[\begin{pmatrix}-8\\-4\end{pmatrix}+s\begin{pmatrix}3.2\\8\end{pmatrix}\right]^t\begin{pmatrix}.2&0\\0&1\end{pmatrix}\left[\begin{pmatrix}-8\\-4\end{pmatrix}+s\begin{pmatrix}3.2\\8\end{pmatrix}\right]\right]=0,$$

which has solution $s = 0.562$. Therefore the minimum along this direction is

$$\begin{aligned}\mathbf{w}(1) &= \mathbf{w}(0) + 0.562(-\Delta J(\mathbf{w}(0)))\\ &= \begin{pmatrix}-8\\-4\end{pmatrix} + 0.562\begin{pmatrix}3.2\\8\end{pmatrix} = \begin{pmatrix}-6.202\\0.496\end{pmatrix}.\end{aligned}$$

Now we turn to the use of conjugate gradients for the next descent. The simple gradient evaluated at $\mathbf{w}(1)$ is

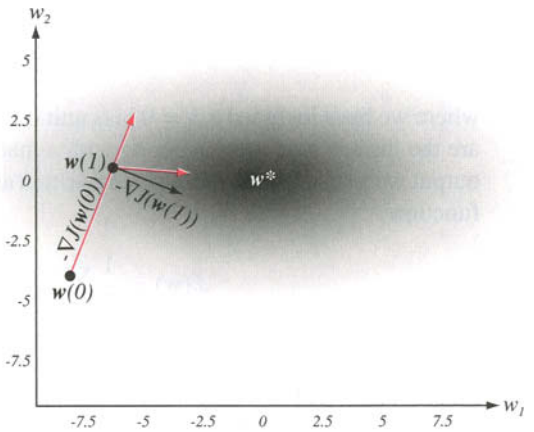
$$-\nabla J(\mathbf{w}(1)) = -\begin{pmatrix}.4w_1(1)\\2w_2(1)\end{pmatrix} = \begin{pmatrix}2.48\\-0.99\end{pmatrix}.$$

(It is easy to verify that this direction, shown as a black arrow in the figure, does not point toward the global minimum at $\mathbf{w}^* = \mathbf{0}$.) We use the Fletcher-Reeves formula (Eq. 56) to construct the conjugate gradient direction:

$$\beta_1 = \frac{\nabla J^t(\mathbf{w}(1))\nabla J(\mathbf{w}(1))}{\nabla J^t(\mathbf{w}(0))\nabla J(\mathbf{w}(0))} = \frac{(-2.48\ .99)\begin{pmatrix}-2.48\\.99\end{pmatrix}}{(-3.2\ 8)\begin{pmatrix}-3.2\\8\end{pmatrix}} = \frac{7.13}{74} = 0.096.$$

For this and all quadratic error surfaces, the Polak-Ribiere formula (Eq. 57) would give the same value as the Fletcher-Reeves formula. Thus the conjugate descent direction is

$$\nabla\mathbf{w}(1) = -\nabla J(\mathbf{w}(1)) + \beta_1\begin{pmatrix}1.6\\4\end{pmatrix} = \begin{pmatrix}2.788\\-.223\end{pmatrix}.$$



Conjugate gradient descent in a quadratic error landscape (indicated by the shading), shown in the density plot, starts at a random point $\mathbf{w}(0)$ and descends by a sequence of line searches. The first direction is given by the standard gradient and terminates at a minimum of the error—the point $\mathbf{w}(1)$. Standard gradient descent from $\mathbf{w}(1)$ would be along the black vector, "spoiling" some of the gains made by the first descent; it would, furthermore, miss the global minimum. Instead, the conjugate gradient (red vector) does not spoil the gains from the first descent, and it properly passes through the global error minimum at $\mathbf{w}^* = \begin{pmatrix}0\\0\end{pmatrix}$.

As above, rather than perform a traditional line search, we use calculus to find the error minimum along this second descent direction:

$$\frac{d}{ds}\left[[\mathbf{w}(1)+s\Delta\mathbf{w}(1)]^t\,\mathbf{H}\,[\mathbf{w}(1)+s\Delta\mathbf{w}(1)]\right]=$$

$$\frac{d}{ds}\left[\left[\begin{pmatrix}-6.202\\0.496\end{pmatrix}+s\begin{pmatrix}2.788\\-.223\end{pmatrix}\right]^t\begin{pmatrix}.2&0\\0&1\end{pmatrix}\left[\begin{pmatrix}-6.202\\0.496\end{pmatrix}+s\begin{pmatrix}2.788\\-.223\end{pmatrix}\right]\right]=0,$$

which has solution $s = 2.23$. This yields the next minimum to be

$$\mathbf{w}(2)=\mathbf{w}(1)+s\Delta\mathbf{w}(1)=\begin{pmatrix}-6.202\\0.496\end{pmatrix}+2.23\begin{pmatrix}2.788\\-.223\end{pmatrix}=\begin{pmatrix}0\\0\end{pmatrix}.$$

Indeed, the conjugate gradient search finds the global minimum in this quadratic error function in two search steps—the number of dimensions of the space.

*6.10 ADDITIONAL NETWORKS AND TRAINING METHODS

We now consider some alternative networks and training methods that have proven effective in special classes of problems.

6.10.1 Radial Basis Function Networks (RBFs)

We have already considered several classifiers, such as Parzen windows, that employ densities estimated by localized basis functions such as Gaussians. In light of our discussion of gradient descent and backpropagation in particular, we now turn to a different method for training such networks. A radial basis function network with linear output units implements

$$z_k(\mathbf{x})=\sum_{j=0}^{n_H}w_{kj}\phi_j(\mathbf{x}), \tag{58}$$

where we have included a $j = 0$ bias unit. If we define a vector $\boldsymbol{\phi}$ whose components are the hidden unit outputs, along with a matrix $\mathbf{W}$ whose entries are the hidden-to-output weights, then Eq. 58 can be rewritten as $\mathbf{z}(\mathbf{x}) = \mathbf{W}\boldsymbol{\phi}$. Minimizing the criterion function

$$J(\mathbf{w})=\frac{1}{2}\sum_{m=1}^{n}\|\mathbf{y}(\mathbf{x}^m;\,\mathbf{w})-\mathbf{t}^m\|^2 \tag{59}$$

is formally equivalent to the linear problem we saw in Chapter 5. We let $\mathbf{T}$ be the matrix consisting of target vectors and let $\boldsymbol{\phi}$ be the matrix whose columns are the vectors $\boldsymbol{\phi}$, then the solution weights obey

$$\boldsymbol{\phi}^t\boldsymbol{\phi}\mathbf{W}^t=\boldsymbol{\phi}^t\mathbf{T}, \tag{60}$$

and the solution can be written directly: $\mathbf{W}^t = \boldsymbol{\Phi}^\dagger\mathbf{T}$. Recall that $\boldsymbol{\phi}^\dagger$ is the pseudo-inverse of $\boldsymbol{\phi}$. One of the benefits of such radial basis function or RBF networks with linear output units is that the solution requires merely such standard linear techniques. Nevertheless, inverting large matrices can be computationally expensive, and thus the above method is generally confined to problems of moderate size.

If the output units are nonlinear, that is, if the network implements

$$z_k(\mathbf{x}) = f\left(\sum_{j=0}^{n_H} w_{kj}\phi_j(\mathbf{x})\right) \tag{61}$$

rather than Eq. 58, then standard backpropagation can be used. One need merely take derivatives of the localized activation functions. For classification problems it is traditional to use a sigmoid for the output units in order to keep the output values restricted to a fixed range. Some of the computational simplification afforded by sigmoidal activation functions in the hidden units functions is lost, but this presents no conceptual difficulties (Problem 38).

6.10.2 Special Bases

Occasionally we may have special information about the functional form of the distributions underlying categories, and then it makes sense to use corresponding hidden unit activation functions. In this way, fewer parameters need to be learned for a given quality of fit to the data. This is an example of increasing the bias of our model and thereby reducing the variance in the solution, a crucial topic we shall consider again in Chapter 9. For instance, if we know that each underlying distribution comes from a mixture of two Gaussians, naturally we would use Gaussian activation functions and use a learning rule that set the parameters, for instance as the mean and entries of the covariance matrix. This is very closely related to the model-dependent maximum-likelihood techniques we saw in Chapter 3.

6.10.3 Matched Filters

In Chapter 2 we treated the problem of designing a classifier in the ideal case that the full probability structure is known. Now we consider the design of a detector for a specific known pattern. This will lead us to the concept of a matched filter. While it comes as no surprise that the optimal detector for a pattern must "match" that pattern (in a way that will become clear), our treatment here lends deeper understanding of why this should be so. Matched filters appear in a wide range of detection problems, particularly those of time-varying signals. Although not a neural network, such filters have close relations to convolutional neural networks we will consider in Section 6.10.4. Furthermore, the maximum response in a hidden unit in a traditional three-layer network occurs when the input pattern "matches" the pattern of input weights leading to that hidden unit (Section 6.5.1).

IMPULSE RESPONSE

Consider the problem of detecting a continuous signal $x(t)$ by a linear detector. We describe such a detector by its *impulse response*, $h(t)$, or, better yet, by its time-reversed impulse response $w(t) = h(-t)$. The output of a linear detector to an arbitrary input $x(t)$ is given by the integral

$$z(T) = \int_{-\infty}^{+\infty} x(t)w(t - T)\, dt. \tag{62}$$

where T is the relative offset of the signal.

Our goal in designing an optimal detector is thus to find the response function that gives the maximum output z; we denote this unknown filter function $\hat{w}(t)$. Of course the output can be made arbitrarily large by choosing a large $w(t)$. Such a solution is uninformative; we are interested in the *shape* of $w(t)$. For this reason, we impose a constraint

$$\int_{-\infty}^{+\infty} w^2(t)\, dt = const \tag{63}$$

ENERGY

where the constant is sometimes called the *energy* of the filter, in analogy with the total energy of a physical signal such as a sound wave or light wave.

The optimization problem, then, is to find the response that maximizes the output of Eq. 62 subject to the constraint of Eq. 63. Because we are searching for the *function* that leads to the maximum output, we employ calculus of variations. Specifically, we take the functional derivative of Eq. 63 and add Eq. 62 times an undetermined multiplier λ, and set it to zero. Without loss of generality, we can arbitrarily assign the offset to be $T = 0$, and thus we find

$$\delta z(0) = \delta \left(\int_{-\infty}^{+\infty} [w^2(t) + \lambda x(t) w(t)]\, dt \right) = 0 \tag{64}$$

or

$$\int_{-\infty}^{+\infty} \underbrace{[2w(t)\delta w(t) + \lambda x(t)\delta w(t)]}_{0 \text{ for extremum}}\, dt = 0. \tag{65}$$

Because Eq. 65 is true for all $\delta w(t)$, the integrand must vanish and this gives our solution: the optimal filter response is

$$\hat{w}(t) = \frac{-\lambda}{2} x(t). \tag{66}$$

In short, the optimal detector has a time-reversed impulse response proportional to the target signal (Fig. 6.23). The overall magnitude is determined by the energy constant, as conveyed by λ; it is easy to show that λ is negative. Finally, we should mention that technically speaking the above the derivation proves only that we have an *extremum*; nevertheless, it can be shown that indeed this solution gives a *maximum* (Problem 36).

6.10.4 Convolutional Networks

TIME DELAY NEURAL NETWORK

We can incorporate prior knowledge into the network architecture itself. For instance, if we demand that our classifier be insensitive to translations of the pattern, we can effectively replicate the recognizer at all such translations. This is the approach taken in time delay neural networks (or TDNNs).

Figure 6.24 shows a typical TDNN architecture; while the architecture consists of input, hidden and output layers, much as we have seen before, there is a crucial difference. Each hidden unit accepts input from a restricted spatial range of positions in the input layer. Hidden units at "delayed" locations (i.e., shifted to the right) accept

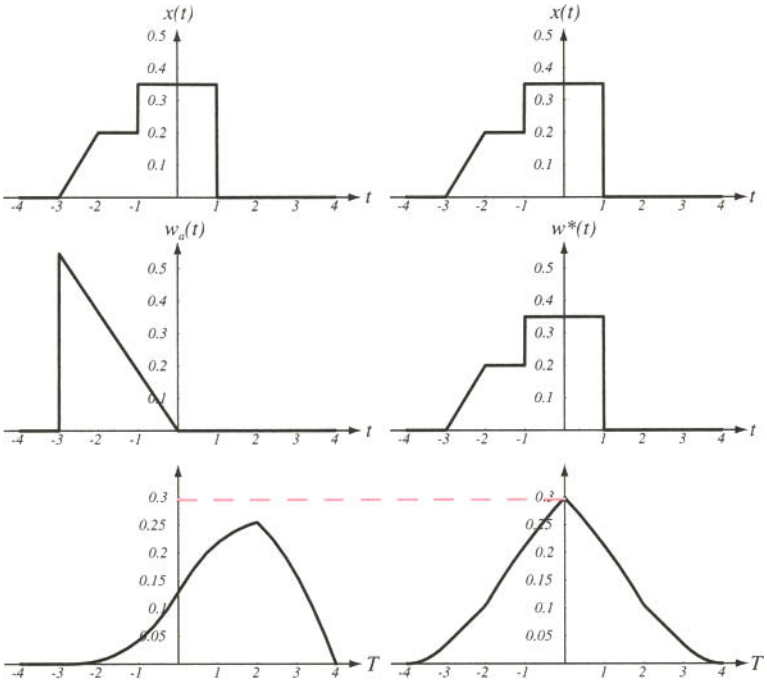


FIGURE 6.23. The left column shows a signal $x(t)$; beneath it is an arbitrary response function $w_a(t)$. At the bottom is the response of the filter as a function of the offset T, as given by Eq. 62. The right column shows the the case where the input and response function "match." The two response functions, $w_a(t)$ and $w^*(t)$ here have the same energy. Note particularly at the bottom that the maximum output in this case is greater than in the nonmatching case at the left.

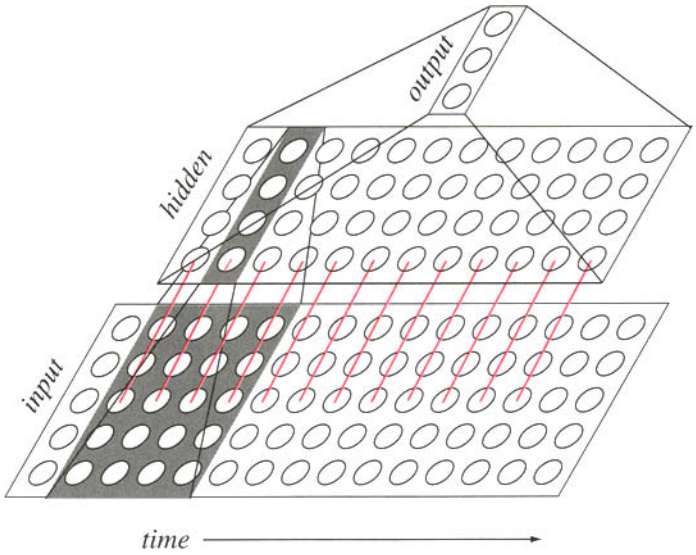


FIGURE 6.24. A time delay neural network (TDNN) uses weight sharing to ensure that patterns are recognized regardless of shift in one dimension; in practice, this dimension generally corresponds to time. Thus all weights shown in red are forced to have the same value. In this example, there are five input units at each time step. Because we hypothesize that the input patterns are of four time steps or less in duration, each of the hidden units at a given time step accepts inputs from only $4 \times 5 = 20$ input units, as highlighted in gray. An analogous translation constraint is also imposed between the hidden and output layer units.

WEIGHT SHARING

inputs from the input layer that are similarly shifted. Training proceeds as in standard backpropagation, but with the added constraint that corresponding weights (that is, shifted to the right or left) are forced to have the same value—an example of *weight sharing*. In this way, the weights learned do not depend upon the position of the training pattern so long as the full pattern lies in the domain of the input layer.

The feedforward operation of the network during recognition is the same as in standard three-layer networks, but because of the weight sharing, the final output does not depend upon the position of the input pattern. The network gets its name from the fact that it was developed for and finds greatest use in speech and other temporal phenomena, where the shift corresponds to delays in time. Such weight sharing can be extended to translations in an orthogonal *spatial* dimensions, and has been used in optical character recognition systems, where the location of an image in the input space is not precisely known.

6.10.5 Recurrent Networks

Up to now we have considered only networks which use feedforward flow of information during classification; the only feedback flow was of error signals during training. Now we turn to feedback or *recurrent* networks. In their most general form, these have found greatest use in time series prediction, but we consider here just one specific type of recurrent net that has had some success in static classification tasks.

Figure 6.25 illustrates such a recurrent architecture, one in which the output unit values are fed back and duplicated as auxiliary inputs, augmenting the traditional

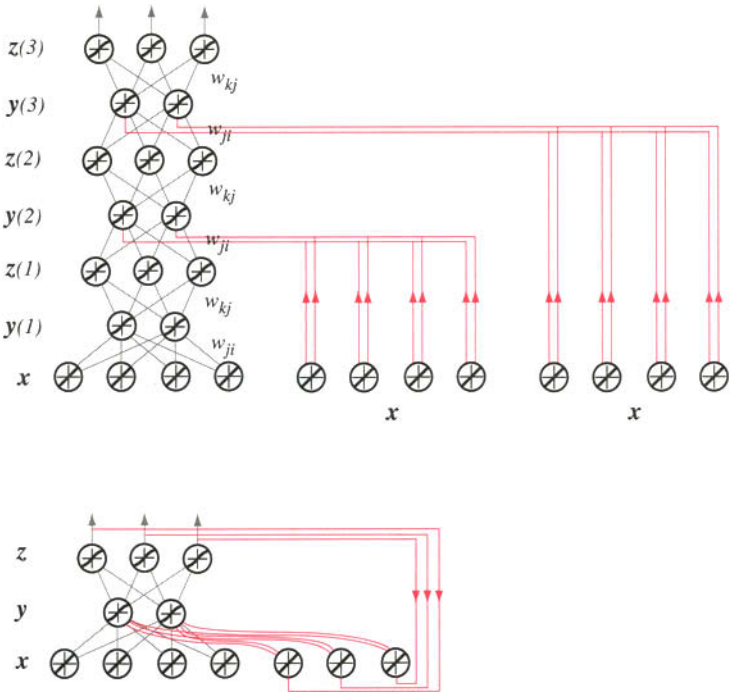


FIGURE 6.25. The form of recurrent network most useful for static classification has the architecture shown at the bottom, with the recurrent connections in red. It is functionally equivalent to a static network with many hidden layers and extensive weight sharing, as shown above.

feature values. During classification, a static pattern **x** is presented to the input units, the feedforward flow is computed, and the outputs are fed back as auxiliary inputs. This, in turn, leads to a different set of hidden unit activations, new output activations, and so on. Ultimately, the activations stabilize, and the final output values are used for classification. As such, this recurrent architecture, if "unfolded" in time, is equivalent to the static network shown at the top of the figure, where it must be understood that many sets of weights are constrained to be the same, as indicated.

Recurrent networks have proven effective in learning time-dependent signals whose structure varies over farily short periods. Such nets are less successful when applied to problems where the structure is longer term, since during training the error gets "diluted" when passed back through the layers many times.

6.10.6 Cascade-Correlation

The central notion underlying the training of networks by cascade-correlation is quite simple. We begin with a two-layer network and train to minimum of an LMS error. If the resulting training error is low enough, training is stopped. In the more common case in which the error is not low enough, we fix the weights but add a single hidden unit, fully connected from inputs and to output units. Then these new weights

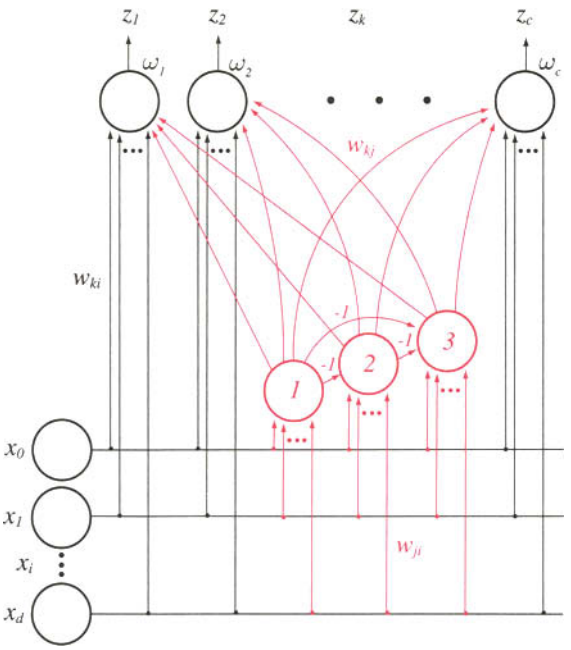


FIGURE 6.26. The training of a multilayer network via cascade-correlation begins with the input later fully connected to the output layer (black). Such weights, w_{ki}, are fully trained using an LMS criterion, as discussed in Chapter 5. If the resulting training error is not sufficiently low, a first hidden unit (labeled 1, in red) is introduced, fully interconnected from the input layer and to the output layer. These new red weights are fully trained, while the previous (black) ones are held fixed. If the resulting training error is still not sufficiently low, a second hidden unit (labeled 2) is likewise introduced, fully interconnected; it also receives a the output from each previous hidden unit, multiplied by −1. Training proceeds in this way, training successive hidden units until the training error is acceptably low.

are trained using an LMS criterion. If the resulting error is not sufficiently low, yet another hidden unit is added, fully connected from the input layer and to the output layer. Furthermore, the output of each previous hidden unit is multiplied by a fixed weight of -1 and presented to the new hidden unit; this prevents the new hidden unit from learning function already represented by the previous hidden units. Then the new weights are trained on an LMS criterion. Thus training proceeds by alternatively training weights, then (if needed) adding a new hidden unit, training the new modifiable weights, and so on. In this way the network grows to a size that depends upon the problem at hand (Fig. 6.26). The benefit of cascade-correlation is that it is often faster than traditional backpropagation since fewer weights are updated at any time.

■ **Algorithm 4. (Cascade-Correlation)**

```
1  begin initialize a, criterion θ, η, k ← 0
2    do m ← m + 1
3       w_ki ← w_ki − η∇J(w)
4    until ‖∇J(w)‖ < θ
5    if J(w) > θ then add hidden unit else exit
6      do m ← m + 1
7         w_ji ← w_ji − η∇J(w); w_kj ← w_kj − η∇J(w)
8      until ‖∇J(w)‖ < θ
9    return w
10 end
```

6.11 REGULARIZATION, COMPLEXITY ADJUSTMENT AND PRUNING

Whereas the number of inputs and outputs of a three-layer network are determined by the problem itself, we do not know ahead of time the number of hidden units, or weights. If we have too many weights and thus degrees of freedom and train too long, there is a danger of overfitting. If we have too few weights, the training set cannot be learned.

One general approach of regularization is to make a new criterion function that depends not only on the classical training error we have seen, but also on classifier complexity. Specifically the new criterion function penalizes highly complex models; searching for the minimum in this criterion is to balance error on the training set with complexity. Formally, then, we can write the new error as the sum of the familiar error on the training set plus a regularization term, which expresses constraints or desirable properties of solutions:

$$J = J_{pat} + \lambda J_{reg}. \tag{67}$$

The parameter λ is adjusted to impose the regularization more or less strongly. Clearly weight decay (Section 6.8.11) can be cast in this form, where J_{reg} is large for large weights.

Another approach is to eliminate—*prune*—weights that are least needed. It is natural to imagine that after training it is the weights with the smallest magnitude

that can be eliminated. Such *magnitude-based pruning* can work, but is provably nonoptimal; sometimes weights with small magnitudes are important for learning the training data.

WALD STATISTIC

The fundamental idea in *Wald statistics* is that we can estimate the importance of a parameter in a model, and then eliminate the parameter having the least such importance. In a network, such a parameter would be a weight. The *Optimal Brain Damage* (OBD) algorithm and its descendent, *Optimal Brain Surgeon* (OBS), use a second-order approximation to predict how the training error depends upon a weight, and eliminate a weight that leads to the smallest increase in the training error.

OPTIMAL BRAIN DAMAGE
OPTIMAL BRAIN SURGEON

Optimal Brain Damage and Optimal Brain Surgeon share the same basic approach of training a network to local minimum in error at weight $\mathbf{w}^*$, and then pruning a weight that leads to the smallest increase in the training error. The predicted functional increase in the error for a change in full weight vector $\delta\mathbf{w}$ is

$$\delta J = \underbrace{\left(\frac{\partial J}{\partial \mathbf{w}}\right)^t \cdot \delta\mathbf{w}}_{\simeq 0} + \frac{1}{2}\,\delta\mathbf{w}^t \cdot \underbrace{\frac{\partial^2 J}{\partial \mathbf{w}^2}}_{\equiv \mathbf{H}} \cdot \delta\mathbf{w} + \underbrace{O(\|\delta\mathbf{w}\|^3)}_{\simeq 0} \tag{68}$$

where $\mathbf{H}$ is the Hessian matrix. The first term vanishes because we are at a local minimum in error; we ignore third- and higher-order terms. The general solution for minimizing this function given the constraint of deleting one weight is (Problem 44):

$$\delta\mathbf{w} = -\frac{w_q}{[\mathbf{H}^{-1}]_{qq}}\,\mathbf{H}^{-1}\cdot\mathbf{u}_q \qquad \text{and} \qquad L_q = \frac{1}{2}\,\frac{w_q^2}{[\mathbf{H}^{-1}]_{qq}}\,. \tag{69}$$

Here, $\mathbf{u}_q$ is the unit vector along the qth direction in weight space and L_q is approximation to the *saliency* of weight q—the increase in training error if weight q is pruned and the other weights updated by the left equation in Eq. 69 (Problem 45).

SALIENCY

We saw in Eq. 45 the outer product approximation for calculating the Hessian matrix. Note that Eq. 69 requires the *inverse* of $\mathbf{H}$. One method to calculating this inverse matrix is to start with a small value, $\mathbf{H}_0^{-1} = \alpha^{-1}\mathbf{I}$, where α is a small parameter—effectively a weight decay constant (Problem 43). Next we update the matrix with

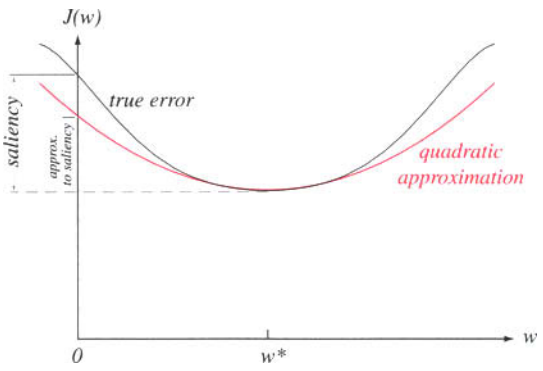


FIGURE 6.27. The saliency of a parameter, such as a weight, is the increase in the training error when that weight is set to zero. One can approximate the saliency by expanding the true error around a local minimum, w^*, and setting the weight to zero. In this example the approximated saliency is smaller than the true saliency; this is typically, but not always, the case.

each pattern according to

$$\mathbf{H}_{m+1}^{-1} = \mathbf{H}_m^{-1} - \frac{\mathbf{H}_m^{-1}\mathbf{X}_{m+1}\mathbf{X}_{m+1}^T\mathbf{H}_m^{-1}}{\frac{n}{a_m} + \mathbf{X}_{m+1}^T\mathbf{H}_m^{-1}\mathbf{X}_{m+1}}, \tag{70}$$

where the subscripts correspond to the pattern being presented and a_m decreases with m. After the full training set has been presented, the inverse Hessian matrix is given by $\mathbf{H}^{-1} = \mathbf{H}_n^{-1}$. Figure 6.28 illustrates the effects of pruning based on OBD, OBS, and the magnitude of a weight in a network.

In algorithmic form, the Optimal Brain Surgeon method is:

■ **Algorithm 5. (Optimal Brain Surgeon)**

```
1 begin initialize n_H, w, θ
2         train a reasonably large network to minimum error
3         do compute H^{-1} by Eq. 70
4             q* ← arg min_q w_q^2/(2[H^{-1}]_qq)        (saliency L_q)
5             w ← w − (w_q*/[H^{-1}]_q*q*) H^{-1} e_q*
6         until J(w) > θ
7   return w
8 end
```

Optimal Brain Damage method is computationally simpler because the calculation of the inverse Hessian matrix in line 3 is particularly simple for a diagonal matrix. The above algorithm terminates when the error is greater than a criterion initialzed to be θ. Another approach is to change line 6 to terminate when the *change* in $J(\mathbf{w})$ due to the elimination of a weight is greater than some criterion value.

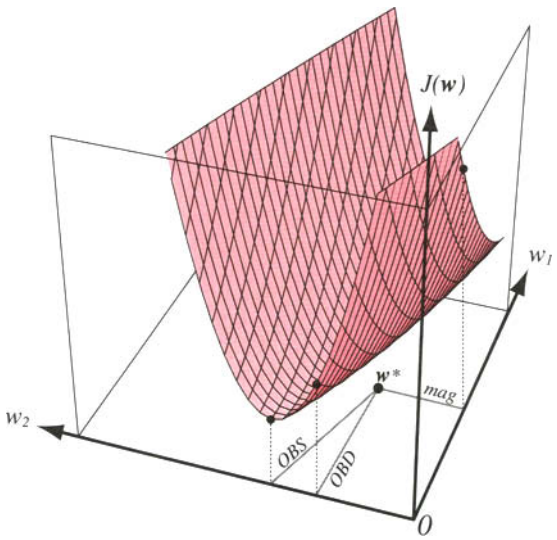


FIGURE 6.28. The figure shows a quadratic error surface as a function of weights, $J(\mathbf{w})$, and the global minimum at $\mathbf{w}^*$. In the second-order approximation to the criterion function, Optimal Brain Damage assumes the Hessian matrix is diagonal, while Optimal Brain Surgeon uses the full Hessian matrix.

SUMMARY

Multilayer nonlinear neural networks—nets with two or more layers of modifiable weights—trained by gradient descent methods such as backpropagation perform a maximum-likelihood estimation of the weight values in the model defined by the network topology. The nonlinear activation function $f(net)$ at the hidden units allows such networks to form an arbitrary decision boundary, so long as there are sufficiently many hidden units.

One of the great benefits of learning in such networks is the simplicity of the learning algorithm, the ease in model selection, and the incorporation of heuristic information and constraints. Such heuristics include weight decay (which penalizes large weights and thus effectively simplifies the classifier) and learning with hints (in which the network also learns an ancillary related task). The parameters of the nonlinear activation function $f(net)$, preprocessing of patterns, target values and weight initialization can all be founded on statistical principles that will ensure fast and uniform learning. The three primary learning protocols are stochastic, batch, and online. Discrete pruning algorithms such as Optimal Brain Surgeon and Optimal Brain Damage correspond to priors favoring *few* weights and can help avoid overfitting.

Alternative networks and training algorithms have benefits. For instance, radial basis functions are most useful when the data clusters. Cascade-correlation networks are generally faster than backpropagation.

BIBLIOGRAPHICAL AND HISTORICAL REMARKS

One of the earliest discussions of what we would now recognize as a neural network was due to the pioneer of modern computer science, Alan Turing, who in a 1948 paper described his "B-type unorganized machine," which consisted of networks of NAND gates [79]. (This forward-looking work was dismissed as a "schoolboy essay" by the head of the lab where Turing worked, Sir Charles Darwin, the grandson of the great English biologist and naturalist.) McCulloch and Pitts provided the first principled mathematical and logical treatment of the behavior of networks of simple neurons [51]. This early work addressed nonrecurrent as well as recurrent nets (those possessing "circles" in their terminology), but not learning. Its concentration on all-or-none or threshold function of neurons indirectly delayed the consideration of continuous valued neurons that would later dominate the field. These authors later wrote an extremely important paper on featural mapping, invariances, and learning in nervous systems and thereby advanced the conceptual development of pattern recognition [58].

Rosenblatt's work on the (two-layer) perceptron (cf. Chapter 5) [64, 65] was some of the earliest to address learning and was the first to include proofs about convergence. A number of stochastic methods, including Pandemonium [69, 70], were developed for training networks with several layers of processors, though in keeping with the preoccupation with threshold functions, such processors generally computed logical functions (AND or OR), rather than some continuous functions favored in later neural network research. The limitations of networks implementing linear discriminants—linear machines—were well known in the 1950s and 1960s and discussed by both their promoters [65] and their detractors [53].

A popular early method was to design by hand three-layer networks with fixed input-to-hidden weights, and then train the hidden-to-output weight; reference [86] provides a review. Much of the difficulty in finding learning algorithms for all lay-

ers in a multilayer neural network came from the prevalent use of linear threshold units. Because these do not have useful derivatives throughout their entire range, the current approach of applying the chain rule for derivatives and the resulting "back-propagation of errors" did not gain more adherents earlier.

The development of backpropagation was gradual, with several steps, not all of which were appreciated at the time they were introduced. The earliest application of adaptive methods that would ultimately become backpropagation came from the field of control [6]. Kalman filtering from electrical engineering used an analog error (difference between predicted and measured output) for adjusting gain parameters in predictors, as described in references [30] and [39]. Bryson, Denham, and Dreyfus showed how Lagrangian methods could train multilayer networks for control, as presented in reference [7]. We saw in the last chapter the work of Widrow, Hoff, and their colleagues [87, 88] in using analog signals and the LMS training criterion applied to pattern recognition in two-layer networks. Werbos [83, 84], too, discussed a method for calculating the derivatives of a function based on a sequence of samples, which, if interpreted carefully, carried the key ideas of backpropagation. Parker's early "Learning logic" [55, 56], developed independently, showed how layers of linear units could be learned by a sufficient number of input-output pairs. This work lacked simulations on representative or challenging problems and terminology in the field, and alas was not appreciated adequately. Le Cun independently developed a learning algorithm for three-layer networks [10, in French] in which target values are propagated, rather than derivatives; the resulting learning algorithm is equivalent to standard backpropagation, as was pointed out shortly thereafter [11].

Without question, the paper by Rumelhart, Hinton and Williams [67], later expanded into a full and readable chapter [68], brought the backpropagation method to the attention of the widest audience. These authors clearly appreciated the power of the method, demonstrated it on key tasks (such as XOR), and applied it to pattern recognition more generally. An enormous number of papers and books of applications—speech perception, optical character recognition, data mining, finance, game playing, and much more—continues unabated. One view of the history of backpropagation is presented in reference [84]; two collections of key papers in the history of neural processing more generally, including many in pattern recognition, are presented in references [2] and [3]. One novel class of such networks includes generalization in the *production* of new patterns [22, 23].

Clear elementary papers on neural networks can be found in [37] and [48], and several good textbooks, which differ from the current one in their emphasis on neural networks over other pattern recognition techniques, can be recommended, particularly textbooks [4, 29, 31, 61] and [63]. An extensive treatment of the mathematical aspects of networks, much of which is beyond that needed for mastering the use of networks for pattern classification, can be found in reference [21]. There is continued exploration of the strong links between networks and more standard statistical methods; White presents an overview [85], and books such as references [9] and [71] explore a number of close relationships. The important relation of multilayer perceptrons to Bayesian methods and probability estimation can be found in references [5, 14, 25, 45, 54, 62] and [66]. Original papers on projection pursuit and MARS can be found in references [16] and [35], respectively, and a good overview can be found in reference [63].

Shortly after its wide dissemination, the backpropagation algorithm was criticized for its lack of biological plausibility; in particular, Grossberg [24] discussed the non-local nature of the algorithm—that is, that synaptic weight values were transported without physical means. Somewhat later, Stork devised a local implementation of

backpropagation [47, 75], and pointed out that it was nevertheless highly implausible as a biological model.

The discussions and debates over the relevance of Kolmogorov's theorem [40] to neural networks—such as references [13, 20, 34, 38, 41, 42] and [44]—have centered on their expressive power. The proof of the universal expressive power of three-layer nets based on bumps and Fourier ideas appears in reference [32]. The expressive power of networks having non-traditional activation functions was explored in references [76], [77], and elsewhere. The fact that three-layer networks can have local minima in the criterion function was explored in reference [52], and some of the properties of error surfaces were illustrated in reference [36].

The outer product approximation to the Hessian matrix and deeper analysis of second-order methods can be found in references [26, 46, 50] and [60]. Three-layer networks trained via cascade-correlation have been shown to perform well compared to standard three-layer nets trained via backpropagation [15]. Although there was little new from a learning theory presented in Fukushima's neocognitron [17, 18], its use of many layers and mixture of hand-crafted feature detectors and learning groupings showed how networks could address shift, rotation, and scale invariance. The method of matched filters predates neural networks; Stork and Levinson used matched filters to explore human visual response functions, as described in references [74] and [78].

A simple method of weight decay was introduced in reference [33], and it gained greater acceptance due to the work of Weigend and others [82]. The method of hints was introduced in reference [1]. While the Wald test [80, 81] has been used in traditional statistical research [72], its application to multilayer network pruning began with the work of Le Cun et al.'s Optimal Brain Damage method [12], later extended to include nondiagonal Hessian matrices [28, 26, 27], including some speedup methods [73]. An extensive review of the computation and use of second-order derivatives in networks can be found in reference [8], and a good review of pruning algorithms can be found in reference [60].

PROBLEMS

Section 6.2

1. Show that if the activation function of the hidden units is linear, a three-layer network is equivalent to a two-layer one. Use your result to explain why a three-layer network with linear hidden units cannot solve a nonlinearly separable problem such as XOR or d-bit parity.

2. Fourier's theorem can be used to show that a three-layer neural net with sigmoidal hidden units can approximate to arbitrary accuracy any posterior function. Consider two-dimensional input and a single output, $z(x_1, x_2)$. Recall that Fourier's theorem states that, given weak restrictions, any such function can be written as a possibly infinite sum of cosine functions, as

$$z(x_1, x_2) \approx \sum_{f_1} \sum_{f_2} A_{f_1 f_2} \cos(f_1 x_1)\ \cos(f_2 x_2),$$

with coefficients $A_{f_1 f_2}$.

(a) Use the trigonometric identity

$$\cos(\alpha)\ \cos(\beta) = \frac{1}{2}\cos(\alpha+\beta) + \frac{1}{2}\cos(\alpha-\beta)$$

to write $z(x_1, x_2)$ as a linear combination of terms $\cos(f_1 x_1 + f_2 x_2)$ and $\cos(f_1 x_1 - f_2 x_2)$.

(b) Show that $\cos(x)$ or indeed any continuous function $f(x)$ can be approximated to any accuracy by a linear combination of sign functions as:

$$f(x) \approx f(x_0) + \sum_{i=0}^{N} [f(x_{i+1}) - f(x_i)] \left[\frac{1 + \mathrm{Sgn}(x - x_i)}{2}\right],$$

where the x_i are sequential values of x; the smaller $x_{i+1} - x_i$, the better the approximation.

(c) Put your results together to show that $z(x_1, x_2)$ can be expressed as a linear combination of step functions or sign functions whose arguments are themselves linear combinations of the input variables x_1 and x_2. Explain, in turn, why this implies that a three-layer network with sigmoidal hidden units and a linear output unit can approximate any function that can be expressed by a Fourier series.

(d) Does your construction guarantee that the derivative $df(x)/dx$ can be well approximated too?

Section 6.3

3. Consider a $d - n_H - c$ network trained with n patterns for m_e epochs.

(a) What is the space complexity in this problem? (Consider both the storage of network parameters as well as the storage of patterns, but not the program itself.)

(b) Suppose the network is trained in stochastic mode. What is the time complexity? Because this is dominated by the number of multiply-accumulates, use this as a measure of the time complexity.

(c) Suppose the network is trained in batch mode. What is the time complexity?

4. Prove that the formula for the sensitivity δ for a hidden unit in a three-layer net (Eq. 21) generalizes to a hidden unit in a four- (or higher-) layer network, where the sensitivity is the weighted sum of sensitivities of units in the next higher layer.

5. Explain in words why the backpropagation rule for training input-to-hidden weights makes intuitive sense by considering the dependency upon each of the terms in Eq. 21.

6. One might guess that the backpropagation learning rules should be *inversely* related to $f'(net)$—that is, that weight change should be *large* where the output does not vary. In fact, as shown in Eq. 17, the learning rule is linear in $f'(net)$. Explain intuitively why the learning rule should be linear in $f'(net)$.

7. Show that the learning rule described in Eqs. 17 and 21 works for bias, where $x_0 = y_0 = 1$ is treated as another input and hidden unit.

8. Consider a standard three-layer backpropagation net with d input units, n_H hidden units, c output units, and bias.

   (a) How many weights are in the net?

   (b) Consider the symmetry in the value of the weights. In particular, show that if the sign is flipped on every weight, the network function is unaltered.

   (c) Consider now the hidden unit exchange symmetry. There are no labels on the hidden units, and thus they can be exchanged (along with corresponding weights) and leave network function unaffected. Prove that the number of such equivalent labelings—the exchange symmetry factor—is thus $n_H 2^{n_H}$. Evaluate this factor for the case $n_H = 10$.

9. Write pseudocode for on-line version of backpropagation training, being careful to distinguish it from stochastic and batch procedures.

10. Express the derivative of a sigmoid in terms of the sigmoid itself in the following two cases (for positive constants a and b):

    (a) A sigmoid that is purely positive: $f(net) = \frac{1}{1+e^{a\ net}}$.

    (b) An antisymmetric sigmoid: $f(net) = a \tanh(b\ net)$.

11. Generalize the backpropagation to four layers, and individual (smooth, differentiable) activation functions at each unit. In particular, let x_i, y_j, v_l and z_k denote the activations on units in successive layers of a four-layer fully connected network, trained with target values t_k. Let f_{1i} be the activation function of unit i in the first layer, f_{2j} in the second layer, and so on. Write a program, with greater detail than that of Algorithm 1, showing the calculation of sensitivities, weight update, and so on, for your general four-layer network.

Section 6.4

12. Explain why the input-to-hidden weights must be different from each other (e.g., random) or else learning cannot proceed well (cf. Computer exercise 2). Specifically, what happens if the weights are initialized so as to have identical values?

13. If the labels on the two hidden units are exchanged (and the weight values changed appropriately), the shape of the error surface is unaffected. Consider a $d - n_H - c$ three-layer network. How many equivalent relabellings of the units (and their associated weights) are there?

14. Show that proper preprocessing of the data will lead to faster convergence, at least in a simple network 2-1 (two-layer) network with bias. Suppose the training data come from two Gaussians, $p(x|\omega_1) \sim N(-.5, 1)$ and $p(x|\omega_2) \sim N(+.5, 1)$. Let the teaching values for the two categories be $t = \pm 1$.

    (a) Write the error as a sum over the n patterns of a function of the weights, inputs, and other parameters.

    (b) Differentiate twice with respect to the weights to get the Hessian matrix $\mathbf{H}$.

    (c) Consider two data sets drawn from $p(\mathbf{x}|\omega_i) \sim N(\boldsymbol{\mu}_i, \mathbf{I})$ for $i = 1, 2$ and $\mathbf{I}$ is the 2-by-2 identity matrix. Calculate your Hessian matrix in terms of $\boldsymbol{\mu}_i$.

    (d) Calculate the maximum and minimum eigenvalues of the Hessian in terms of the components of $\boldsymbol{\mu}_i$.

    (e) Suppose $\boldsymbol{\mu}_1 = (1, 0)^t$ and $\boldsymbol{\mu}_2 = (0, 1)^t$. Calculate the ratio of the eigenvalues, and hence a measure of the convergence time.

(f) Now standardize your data, by subtracting means and scaling to have unit covariances in each of the two dimensions. That is, find two new distributions that have overall zero mean and the same covariance. Check your answer by calculating the ratio of the maximum to minimum eigenvalues.

(g) If T denotes the total training time for the unprocessed data, express the time required for the preprocessed data (cf. Computer exercise 12).

15. Consider the derivation of the bounds on the convergence time for simple gradient descent. Assume that the error function can be well described by a Hessian matrix $\mathbf{H}$ having d eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_d$, where λ_{max} and λ_{min} are the maximum and minimum values in this set. Assume that the learning rate has been set to the optimum for one of the dimensions, as based on Eq. 36.

(a) Express the optimum rate in terms of the appropriate eigenvalue, that is, either λ_{max} or λ_{min}.

(b) State a convergence criterion for learning.

(c) In terms of the variables given, compute the time for your system to meet your convergence criterion.

16. Assume that the criterion function $J(\mathbf{w})$ is well-described to second order by a Hessian matrix $\mathbf{H}$.

(a) Show that convergence of learning is ensured if the learning rate obeys $\eta < 2/\lambda_{max}$, where λ_{max} is the largest eigenvalue of $\mathbf{H}$.

(b) Show that the learning time is thus dependent upon the ratio of the largest to the smallest nonnegligible eigenvalue of $\mathbf{H}$.

(c) Explain why "standardizing" the training data can therefore reduce learning time.

(d) What is the relation between such standardizing and the whitening transform described in Chapter 2?

Section 6.6

17. Fill in the steps in the derivation leading to Eq. 26.

18. Confirm that one of the solutions to the minimum squared-error condition yields outputs that are indeed posterior probabilities. Do this as follows:

(a) To find the minimum of $\tilde{J}(\mathbf{w})$ in Eq. 28, calculate its derivative $\partial \tilde{J}(\mathbf{w})/\partial \mathbf{w}$; this will consist of the sum of two integrals. Set $\partial \tilde{J}(\mathbf{w})/\partial \mathbf{w} = 0$ and solve to obtain the natural solution.

(b) Apply Bayes rule and the normalization $P(\omega_k|\mathbf{x}) + P(\omega_{i\neq k}|\mathbf{x}) = 1$ to prove that the outputs $z_k = g_k(\mathbf{x};\ \mathbf{w})$ are indeed equal to the posterior probabilities $P(\omega_k|\mathbf{x})$.

19. In the derivation that backpropagation finds a least squares fit to the posterior probabilities, it was implicitly assumed that the network could indeed represent the true underlying distribution. Explain where in the derivation this was assumed, and what in the subsequent steps may not hold if that assumption is violated.

20. Show that the softmax output (Eq. 30) indeed approximates posterior probabilities if the hidden unit outputs, $\mathbf{y}$, belong to the family of exponential distributions

as

$$p(\mathbf{y}|\omega_k) = \exp[A(\tilde{\mathbf{w}}_k) + B(\mathbf{y}, \phi) + \tilde{\mathbf{w}}_k^t \mathbf{y}]$$

for n_H-dimensional vectors $\tilde{\mathbf{w}}_k$ and $\mathbf{y}$, and scalar ϕ and scalar functions $A(\cdot)$ and $B(\cdot, \cdot)$. Proceed as follows:

(a) Given $p(\mathbf{y}|\omega_k)$, use Bayes theorem to write the posterior probability $P(\omega_k|\mathbf{y})$.

(b) Interpret the parameters $A(\cdot)$, $\tilde{\mathbf{w}}_k$, $B(\cdot, \cdot)$, and ϕ in light of your results.

21. Consider a three-layer network for classification with output units employing softmax (Eq. 30), trained with 0-1 signals.

(a) Derive the learning rule if the criterion function (per pattern) is sum squared error, that is,

$$J(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2.$$

(b) Repeat for the criterion function is cross-entropy, that is,

$$J_{ce}(\mathbf{w}) = \sum_{k=1}^{c} t_k \ln \frac{t_k}{z_k}.$$

22. Clearly if the discriminant functions $g_{k_1}(\mathbf{x};\ \mathbf{w})$ and $g_{k_2}(\mathbf{x};\ \mathbf{w})$ were independent, the derivation of Eq. 28 would follow from Eq. 27. Show that the derivation is nevertheless valid despite the fact that these functions are implemented using the same input-to-hidden weights. Are the discriminant functions independent?

Section 6.7

23. Show that the slope of the sigmoid and the learning rates together determine the learning time.

(a) That is, show that if the slope of the sigmoid is increased by a factor of γ, and the learning rate decreased by a factor $1/\gamma$, the total learning time remains the same.

(b) Must the input be rescaled for this relationship to hold?

24. Show that the basic three-layer neural networks of Section 6.2 are special cases of general additive models by describing in detail the correspondences between Eqs. 7 and 32.

25. Show that the sigmoidal activation function acts to transmit the maximum information if its inputs are distributed normally. Recall that the entropy (a measure of information) is defined as $H = \int p(y)\ln[p(y)]\, dy$.

(a) Consider a continuous input variable x drawn from the density $p(x) \sim N(0, \sigma^2)$. What is entropy for this distribution?

(b) Suppose samples x are passed through an antisymmetric sigmoidal function to give $y = f(x)$, where the zero crossing of the sigmoid occurs at the peak of the Gaussian input, and the effective width of the linear region equal to

the range $-\sigma < x < +\sigma$. What values of a and b in Eq. 34 ensure this?

(c) Calculate the entropy of the output distribution $p(y)$.

(d) Suppose instead that the activation function were a Dirac delta function $\delta(x - 0)$. What is the entropy of the resulting output distribution $p(y)$?

(e) Summarize your results of (c) and (d) in words.

Section 6.8

26. Consider the sigmoidal activation function:

$$f(net) = a \tanh(b\, net) = a\left[\frac{1 - e^{-2b\, net}}{1 + e^{-2b\, net}}\right] = \frac{2a}{1 + e^{-2b\, net}} - a.$$

(a) Show that its derivative $f'(net)$ can be written simply in terms of $f(net)$ itself.

(b) What are $f(net)$, $f'(net)$ and $f''(net)$ at $net = -\infty? 0? +\infty?$

(c) For which value of net is the second derivative $f''(net)$ extremal?

27. Consider the computational burden for standardizing data, as described in the text.

(a) What is the computational complexity of standardizing a training set of n d-dimensional patterns?

(b) Estimate the computational complexity of training. Use the heuristic for choosing the size of the network (i.e., number of weights) described in Section 6.8.7. Assume that the number of training epochs is nd.

(c) Use your results from (a) and (b) to express the computational burden of standardizing as a ratio. (Assume unknown constants have value 1.0.)

28. Derive the gradient descent learning rule for a three-layer network with linear input units and sigmoidal hidden and output units for the Minkowski error of Eq. 42 with arbitrary R. Confirm that your answer reduces to Eqs. 17 and 21 for $R = 2$.

29. Consider a $d - n_H - c$ three-layer neural network whose input units are linear and output units are sigmoidal but each hidden unit implements a particular polynomial function, trained on a sum-square error criterion. Specifically, let the output of hidden unit j be given by

$$o_j = w_{ji}x_i + w_{jm}x_m + q_j x_i x_m$$

for two prespecified inputs, i and $m \neq i$.

(a) Write gradient descent learning rule for the input-to-hidden weights and scalar parameter q_j.

(b) Does the learning rule for the hidden-to-output unit weights differ from that in the standard three-layer network described in the text?

(c) What might be some of the strengths and the weaknesses of such a network and its learning rule?

30. Derive the backpropagation learning rules for the Minkowski error defined in Eq. 42. Confirm that your result reduces to the standard learning rules for the case $R = 2$. Repeat your derivation for the Manhattan metric directly, that is, without referring to R.

Section 6.9

31. Show intermediate steps in the derivation of the full Hessian matrix for a sum-square-error criterion in a three-layer network, as given in Eqs. 47 and 48.

32. Repeat Problem 31 but for a cross-entropy error criterion.

33. Suppose a Hessian matrix for an error function is proportional to the identity matrix, that is, $\mathbf{H} \propto \mathbf{I}$.

(a) Prove that in this case both the Polak-Ribiere and the Fletcher-Reeves formulas, Eqs. 57 and 56, used in conjugate gradient descent give $\beta = 0$.

(b) Interpret your result. In particular state why for a Hessian matrix proportional to the identity matrix the two methods would give the same β, and moreover why the resulting value for β will be 0.

34. Consider an error surface that can be well approximated by a positive definite Hessian matrix associated with a sum-square error on the training set. Let $\mathbf{w}^*$ denote a network's optimal weight vector, that is, at the global error minimum, and let $\Delta\mathbf{w} = \mathbf{w} - \mathbf{w}^*$ denote the difference between an arbitrary weight vector and this global minimum. Prove that moving along the direction given in conjugate gradient descent initially shrinks $||\Delta\mathbf{w}||$, and thus the process converges.

35. Based on the discussion in the text, derive the quickprop learning rule in Eq. 53. Be sure to state any assumptions you need to make.

Section 6.10

36. Prove that the matched filter given in Eq. 66 gives a maximum (not merely an extremum) as follows. Throughout the fixed signal desired is $x(t)$.

(a) Let $w(t) = w^*(t) + h(t)$ be a trial weight function, where $w^*(t)$ is the matched filter. We demand that the constraint of Eq. 63 be obeyed for this trial weight function, that is,

$$\int_{-\infty}^{+\infty} w^2(t)\, dt = \int_{-\infty}^{+\infty} w^{*2}(t)\, dt.$$

Expand the left-hand side of this equation to prove

$$-2\int_{-\infty}^{+\infty} h(t)w^*(t)\, dt = \int_{-\infty}^{+\infty} h^2(t)dt \geq 0. \tag{71}$$

(b) Let z^* be the output of the matched filter to the target input, according to Eq. 62. Use your result in (a) to show that the output for the trial weights is

$$z = z^* - \frac{1}{-\lambda}\int_{-\infty}^{+\infty} h^2(t)\, dt.$$

(c) The sign of λ is determined by the sign of z^*. Use Eq. 66 to show that λ has the opposite sign to that of z^*.

(d) Use all your above results to prove that $z^* = z$ if and only if $h(t) = 0$ for all t, i.e., that the matched filter $w^*(t)$ ensures the *maximum* output, and moreover $w^*(t)$ is unique.

37. Derive the central equations (i.e., analogous to Eq. 69) for OBD and OBS in a three-layer sigmoidal network for a cross-entropy error.

38. Derive the learning rule for a three-layer radial basis function neural network in which the hidden units are spherical Gaussians whose mean $\boldsymbol{\mu}$ and amplitude are learned in response to data.

Section 6.11

39. Consider a general constant matrix $\mathbf{K}$ and variable vector parameter $\mathbf{x}$.

(a) Use summation notation with components explicit to derive the formula for the derivative:

$$\frac{d}{d\mathbf{x}}[\mathbf{x}^t\mathbf{K}\mathbf{x}] = (\mathbf{K} + \mathbf{K}^t)\mathbf{x}.$$

(b) Show simply that for the case where $\mathbf{K}$ is symmetric (as for instance the Hessian matrix $\mathbf{H} = \mathbf{H}^t$), we have

$$\frac{d}{d\mathbf{x}}[\mathbf{x}^t\mathbf{H}\mathbf{x}] = 2\mathbf{H}\mathbf{x}$$

as was used in the derivation of the Optimal Brain Damage and Optimal Brain Surgeon methods.

40. Weight decay is equivalent to doing gradient descent on an error that has a "complexity" term.

(a) Show that in the weight decay rule $w_{ij}^{new} = w_{ij}^{old}(1-\epsilon)$ amounts to performing gradient descent in the error function $J_{ef} = J(\mathbf{w}) + \frac{2\epsilon}{\eta}\mathbf{w}^t\mathbf{w}$ (Eq. 39).

(b) Express γ in terms of the weight decay constant ϵ and learning rate η.

(c) Likewise, show that if $w_{mr}^{new} = w_{mr}^{old}(1 - \epsilon_{mr})$ where $\epsilon_{mr} = 1/(1 + w_{mr}^2)^2$, that the new effective error function is $J_{ef} = J(\mathbf{w}) + \gamma \sum_{mr} w_{mr}^2/(1 + w_{mr}^2)$. Find γ in terms of η and ϵ_{mr}.

(d) Consider a network with a wide range of magnitudes for weights. Describe qualitatively how the two different weight decay methods affect the network.

41. Show that the weight decay rule of Eq. 38 is equivalent to a prior on models that favors small weights.

42. Prove that the weight decay rule of Eq. 38 leads to the J_{reg} of Eq. 39.

43. Equation 69 for Optimal Brain Surgeon requires the inverse of $\mathbf{H}$. One method to calculating this inverse matrix is to start with a small value, $\mathbf{H}_0^{-1} = \alpha^{-1}\mathbf{I}$ and iteratively estimate $\mathbf{H}^{-1}$ by Eq. 70. In this context, show that α acts as a weight decay constant.

44. Derive the key equation in the Optimal Brain Surgeon algorithms, Eq. 69, from the Taylor expansion of the criterion function expressed in Eq. 68.

45. Consider the computational demands of the Optimal Brain Surgeon procedure as follows:

(a) Find the space and the time computational complexities for one step in the nominal OBS method.

(b) Find the space and the time computational complexities for pruning the *first* weight in OBS. What is it for pruning subsequent weights, if one uses Shur's decomposition method?

(c) Find the space and the time computational complexities for one step of OBD (without retraining).

(d) Find the saliency in Optimal Brain Damage, where one assumes $\mathbf{H}_{ij} = 0$ for $i \neq j$.

COMPUTER EXERCISES

Several exercises will make use of the following three-dimensional data sampled from three categories, denoted ω_i.

	ω_1			ω_2			ω_3		
sample	x_1	x_2	x_3	x_1	x_2	x_3	x_1	x_2	x_3
1	1.58	2.32	−5.8	0.21	0.03	−2.21	−1.54	1.17	0.64
2	0.67	1.58	−4.78	0.37	0.28	−1.8	5.41	3.45	−1.33
3	1.04	1.01	−3.63	0.18	1.22	0.16	1.55	0.99	2.69
4	−1.49	2.18	−3.39	−0.24	0.93	−1.01	1.86	3.19	1.51
5	−0.41	1.21	−4.73	−1.18	0.39	−0.39	1.68	1.79	−0.87
6	1.39	3.16	2.87	0.74	0.96	−1.16	3.51	−0.22	−1.39
7	1.20	1.40	−1.89	−0.38	1.94	−0.48	1.40	−0.44	0.92
8	−0.92	1.44	−3.22	0.02	0.72	−0.17	0.44	0.83	1.97
9	0.45	1.33	−4.38	0.44	1.31	−0.14	0.25	0.68	−0.99
10	−0.76	0.84	−1.96	0.46	1.49	0.68	−0.66	−0.45	0.08

Section 6.2

1. Consider a 2-2-1 network with bias, where the activation function at the hidden units and the output unit is a sigmoid $y_j = a \tanh(b\, net_j)$ for $a = 1.716$ and $b = 2/3$.

(a) Suppose the matrices describing the input-to-hidden weights (w_{ji} for $j = 1, 2$ and $i = 0, 1, 2$) and the hidden-to-output weights (w_{kj} for $k = 1$ and $j = 0, 1, 2$) are, respectively,

$$\begin{pmatrix} 0.5 & -0.5 \\ 0.3 & -0.4 \\ -0.1 & 1.0 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1.0 \\ -2.0 \\ 0.5 \end{pmatrix}.$$

The network is to be used to place patterns into one of two categories, based on the sign of the output unit signal. Shade a two-dimensional x_1x_2 input space ($-5 \leq x_1, x_2 \leq +5$) black or white according to the category given by the network.

(b) Repeat part (a) with the following weight matrices:

$$\begin{pmatrix} -1.0 & 1.0 \\ -0.5 & 1.5 \\ 1.5 & -0.5 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 0.5 \\ -1.0 \\ 1.0 \end{pmatrix}.$$

Section 6.3

2. Create a 3-1-1 sigmoidal network with bias to be trained to classify patterns from ω_1 and ω_2 in the table above. Use stochastic backpropagation (Algorithm 1) with learning rate $\eta = 0.1$ and sigmoid as described in Eq. 34 in Section 6.8.2.

(a) Initialize all weights randomly in the range $-1 \leq w \leq +1$. Plot a learning curve—the training error as a function of epoch.

(b) Now repeat (a) but with weights initialized to be the *same* throughout each level. In particular, let all input-to-hidden weights be initialized with $w_{ji} = 0.5$ and all hidden-to-output weights with $w_{kj} = -0.5$.

(c) Explain the source of the differences between your learning curves (cf. Problem 12).

3. Consider the nonlinearly separable categorization problem shown in Fig. 6.8.

(a) Train a 1-3-1 sigmoidal network with bias by means of batch backpropagation (Algorithm 2) to solve that problem.

(b) Display your decision boundary by classifying points along the x-axis.

(c) Repeat with a 1-2-1 network.

(d) Inspect the decision boundary for your 1-3-1 network (or construct by hand an optimal one) and explain why no 1-2-1 network with sigmoidal hidden units can achieve it.

4. Write a backpropagation program for a 2-2-1 network with bias to solve the XOR problem (cf. Fig. 6.1).

(a) Show the input-to-hidden weights and analyze the function of each hidden unit.

(b) Plot the representation of each pattern as well as the final decision boundary in the $y_1 y_2$-space.

(c) Although it was not used as a training pattern, show the representation of $\mathbf{x} = \mathbf{0}$ in your $y_1 y_2$-space.

5. Write a basic backpropagation program for a 3-3-1 network with bias to solve the three-bit parity problem where each input has value ± 1. That is, the output should be $+1$ if the number of inputs that have value $+1$ is even, and should be -1 if the number of such inputs is odd.

(a) Show the input-to-hidden weights and analyze the function of each hidden unit.

(b) Retrain several times from new random points until you get a local (but not global) minimum. Analyze the function of the hidden units now.

(c) How many patterns are properly classified for your local minima? Explain.

6. Explore the effect of the number of hidden units on the accuracy of a $2 - n_H - 1$ neural network classifier (with bias) in a two-dimensional two-category problem with $p(\mathbf{x}|\omega_1) \sim N(\mathbf{0}, \mathbf{I})$ and $p(\mathbf{x}|\omega_2) \sim N\left(\begin{pmatrix}1\\.5\end{pmatrix}, \begin{pmatrix}3 & 1\\1 & 2\end{pmatrix}\right)$.

(a) Generate a training set of 100 points, 50 from each category, as well as an independent set of 40 points (20 from each category).

(b) Train your network fully with with different number of hidden units, $1 \leq n_H \leq 10$, and for each trained network plot the training and test error, as in Fig. 6.15. What number of hidden units gives the minimum training error?

What number of hidden units gives the minimum test error? (Call this last number n_H^*.)

(c) Re-initialize a $2 - n_H^* - 1$ network and train it. Plot learning curves, that is, the training error and the validation error as a function of the number of epochs. Suppose you were to stop training at the the minimum of this validation error. What is the value of the validation error at such a stopping point?

(d) Compare and explain the difference between the minimum of the validation error in part (c) with the corresponding validation error for n_H^* hidden units in part (b).

7. Train a 2-4-1 network having a different activation function at each hidden unit on a two-dimensional two-category problem with 2^k patterns chosen randomly from the unit square. Estimate k such that the expected error is 25%. Discuss your results.

Section 6.4

8. Construct a 3-1-3 network (with bias) having sigmoidal activation functions. Train your network on the data in the table above.

(a) Compute the Hessian matrix $\mathbf{H}$.

(b) Find the eigenvalues and eigenvectors of $\mathbf{H}$.

Section 6.5

9. Show that the hidden units of a network may find meaningful feature groupings in the following problem based on optical character recogntion.

(a) Let your input space consist of and 8×8 pixel grid. Generate a 100 training patterns for a B category in the following way. Start with a block-letter representation of the B where the "black" pixels have value -1.0 and the "white" pixels $+1.0$. Generate 100 different versions of this prototype B by adding independent random noise independently to each pixel. Let the distribution of noise be uniform between -0.5 and $+0.5$. Repeate the above for O and E. In this way you have created a set $\mathcal{D}$ of 300 training patterns.

(b) Train a 64-3-3 network (with bias) using your training set $\mathcal{D}$.

(c) Display the input-to-hidden weights as 8×8 images. separately for each hidden unit.

(d) Interpret the patterns of weights displayed in part (c).

Section 6.6

10. Consider training a three-layer network for estimating probabilities of four, equally probable three-dimensional distributions in the following.

(a) First generate a training set $\mathcal{D}$ of 4000 patterns, 1000 each from the following Gaussian distributions $p(\mathbf{x}|\omega_i) \sim N(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$ for $i = 1, 2, 3, 4$:

i	$\boldsymbol{\mu}_i$	$\boldsymbol{\Sigma}_i$
1	$\mathbf{0}$	$\mathbf{I}$
2	$\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 2 & 2 \\ 1 & 2 & 5 \end{pmatrix}$

i	$\boldsymbol{\mu}_i$	$\boldsymbol{\Sigma}_i$
3	$\begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}$	Diag[2,6,1]
4	$\begin{pmatrix} 0 \\ 0.5 \\ 1 \end{pmatrix}$	Diag[2,1,3]

(b) Train a 3-3-4 network (with bias) using the 4000 patterns in $\mathcal{D}$. Use the softmax target values as given in Eq. 30.

(c) Use your trained network to estimate the posterior probabilities of each of the following five test patterns: $\mathbf{x}_1 = (0, 0, 0)^t$, $\mathbf{x}_2 = (-1, 0, 1)^t$, $\mathbf{x}_3 = (.5, -.5, 0)^t$, $\mathbf{x}_4 = (-1, 0, 0)^t$, and $\mathbf{x}_5 = (0, 0, 1)^t$.

(d) Use your network to classify each of the five test patterns.

(e) Use the techniques of Chapter 2 to calculate the *a posteriori* probabilities of the five test points. Compare these answer to your answers in part (c).

Section 6.7

11. Consider several gradient descent methods applied to a criterion function in one dimension: simple gradient descent with fixed learning rate η, optimized descent, Newton's method, and Quickprop. Consider first the criterion function $J(w) = w^2$ which of course has minimum $J = 0$ at $w = 0$. In all cases, start the descent at $w(0) = 1$. For definiteness, we consider convergence to be complete when $J(\mathbf{w}) < 0.001$.

(a) Plot the number of steps until convergence as a function of η for $\eta = 0.01, 0.03, 0.1, 0.3, 1, 3$.

(b) Calculate the optimum learning rate η_{opt} by Eq. 36, and confirm that this value is consistent with your results in part (a).

(c) Calculate the weight update by the Quickprop rule of Eq. 53.

Section 6.8

12. Demonstrate that preprocessing data can lead to significant reduction in time of learning. Consider a single linear output unit for a two-category classification task, with teaching values ± 1 and squared-error criterion.

(a) Write a program to train the three weights based on training samples.

(b) Generate 20 samples from each of two categories $P(\omega_1) = P(\omega_2) = .5$ and $p(\mathbf{x}|\omega_i) \sim N(\boldsymbol{\mu}_i), \mathbf{I}$, where $\mathbf{I}$ is the 2×2 identity matrix and $\boldsymbol{\mu}_1 = (0, 1)^t$ and $\boldsymbol{\mu}_2 = (1, -1)^t$.

(c) Find the optimal learning rate empirically by trying a few values.

(d) Train to minimum error. Why is there no danger of overtraining in this case?

(e) Why can we be sure that it is at least possible that this network can achieve the minimum (Bayes) error?

(f) Generate 100 test samples, 50 from each category, and estimate the error rate.

(g) Now preprocess the data by subtracting off the mean and scaling standard deviation in each dimension.

(h) Repeat the above, and find the optimal learning rate.

(i) Find the error rate on the (transformed) test set.

(j) Verify that the accuracy is virtually the same in the two cases (any differences can be attributed to stochastic effects).

(k) Explain in words the underlying reasons for your results.

BIBLIOGRAPHY

[1] Yaser S. Abu-Mostafa. Learning from hints in neural networks. *Journal of Complexity*, 6(2):192–198, 1990.

[2] James A. Anderson, Andras Pellionisz, and Edward Rosenfeld, editors. *Neurocomputing 2: Directions for Research*. MIT Press, Cambridge, MA, 1990.

[3] James A. Anderson and Edward Rosenfeld, editors. *Neurocomputing: Foundations of Research*. MIT Press, Cambridge, MA, 1988.

[4] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Oxford, UK, 1995.

[5] John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Françoise Fogelman-Soulié and Jeanny Hérault, editors, *Neurocomputing: Algorithms, Architectures and Applications*, pages 227–236, Springer-Verlag, New York, 1990.

[6] Arthur E. Bryson, Jr., Walter Denham, and Stuart E. Dreyfus. Optimal programming problem with inequality constraints. I: Necessary conditions for extremal solutions. *American Institute of Aeronautics and Astronautics Journal*, 1(11):2544–2550, 1963.

[7] Arthur E. Bryson, Jr. and Yu-Chi Ho. *Applied Optimal Control*. Blaisdell, Waltham, MA, 1969.

[8] Wray L. Buntine and Andreas S. Weigend. Computing second derivatives in feed-forward networks: A review. *IEEE Transactions on Neural Networks*, 5(3):480–488, 1991.

[9] Vladimir Cherkassky, Jerome H. Friedman, and Harry Wechsler, editors. *From Statistics to Neural Networks: Theory and Pattern Recognition Applications*. NATO ASI. Springer, New York, 1994.

[10] Yann Le Cun. A learning scheme for asymmetric threshold networks. In *Proceedings of Cognitiva 85*, pages 599–604, Paris, France, 1985.

[11] Yann Le Cun. Learning processes in an asymmetric threshold network. In Elie Bienenstock, Françoise Fogelman-Soulié, and Gerard Weisbuch, editors, *Disordered Systems and Biological Organization*, pages 233–240, Les Houches, Springer-Verlag, France, 1986.

[12] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal Brain Damage. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 598–605. Morgan Kaufmann, San Mateo, CA, 1990

[13] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematical Control Signals Systems*, 2:303–314, 1989.

[14] John S. Denker and Yann Le Cun. Transforming neural-net output levels to probability distributions. In Richard Lippmann, John Moody, and David Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 3, pages 853–859. Morgan Kaufmann, San Mateo, CA, 1991.

[15] Scott E. Fahlman and Christian Lebiere. The Cascade-Correlation learning architecture. In David S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532. Morgan Kaufmann, San Mateo, CA, 1990.

[16] Jerome H. Friedman and Werner Stuetzle. Projection pursuit regression. *Journal of the American Statistical Association*, 76(376):817–823, 1981.

[17] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

[18] Kunihiko Fukushima, Sei Miyake, and Takayuki Ito. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):826–834, 1983.

[19] Federico Girosi, Michael Jones, and Tomaso Poggio. Regularization theory and neural networks architectures. *Neural Computation*, 7(2):219–269, 1995.

[20] Federico Girosi and Tomaso Poggio. Representation properties of networks: Kolmogorov's theorem is irrelevant. *Neural Computation*, 1(4):465–469, 1989.

[21] Richard M. Golden. *Mathematical Methods for Neural Network Analysis and Design*. MIT Press, Cambridge, MA, 1996.

[22] Igor Grebert, David G. Stork, Ron Keesing, and Steve Mims. Connectionist generalization for productions: An example from Gridfont. *Neural Networks*, 5(4):699–710, 1992.

[23] Igor Grebert, David G. Stork, Ron Keesing, and Steve Mims. Network generalization for production: Learning and producing styled letterforms. In John E. Moody, Stephen J. Hanson, and Richard P. Lippmann, editors, *Advances in Neural Information Processing Systems*, volume 4, pages 1118–1124. Morgan Kaufmann, San Mateo, CA, 1992.

[24] Stephen Grossberg. Competitive learning: From interactive activation to adaptive resonance. *Cognitive Science*, 11(1):23–63, 1987.

[25] John B. Hampshire, II and Barak A. Pearlmutter. Equivalence proofs for multi-layer Perceptron classifiers and

the Bayesian discriminant function. In David S. Touretzky, Jeffrey L. Elman, Terrence J. Sejnowski, and Geoffrey E. Hinton, editors, *Proceedings of the 1990 Connectionst Models Summer School*, pages 159–172. Morgan Kaufmann, San Mateo, CA, 1990.

[26] Babak Hassibi and David G. Stork. Second-order derivatives for network pruning: Optimal Brain Surgeon. In Stephen J. Hanson, Jack D. Cowan, and C. Lee Giles, editors, *Advances in Neural Information Processing Systems*, volume 5, pages 164–171. Morgan Kaufmann, San Mateo, CA, 1993.

[27] Babak Hassibi, David G. Stork, and Greg Wolff. Optimal Brain Surgeon and general network pruning. In *Proceedings of the International Conference on Neural Networks*, volume 1, pages 293–299. IEEE, San Francisco, CA, 1993.

[28] Babak Hassibi, David G. Stork, Gregory Wolff, and Takahiro Watanabe. Optimal Brain Surgeon: Extensions and performance comparisons. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 263–270. Morgan Kaufmann, San Mateo, CA, 1994.

[29] Mohamad H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, MA, 1995.

[30] Simon Haykin. *Adaptive Filter Theory*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1991.

[31] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Macmillan, New York, 1994.

[32] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Proceeding of the International Joint Conference on Neural Networks (IJCNN)*, volume 1, pages 593–605. IEEE, New York, 1989.

[33] Geoffrey E. Hinton. Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pages 1–12. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

[34] Kurt Hornik, Maxwell Stinchcombe, and Halbert L. White, Jr. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[35] Peter J. Huber. Projection pursuit. *Annals of Statistics*, 13(2):435–475, 1985.

[36] Don R. Hush, John M. Salas, and Bill G. Horne. Error surfaces for multi-layer Perceptrons. In *Proceedings of International Joint Conference on Neural Networks (IJCNN)*, volume 1, pages 759–764. IEEE, New York, 1991.

[37] Anil K. Jain, Jianchang Mao, and K. Moidin Mohiuddin. Artificial neural networks: A tutorial. *Computer*, 29(3):31–44, 1996.

[38] Lee K. Jones. Constructive approximations for neural networks by sigmoidal functions. *Proceedings of the IEEE*, 78(10):1586–1589, 1990.

[39] Rudolf E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME, Series D, Journal of Basic Engineering*, 82(1):34–45, 1960.

[40] Andreæi N. Kolmogorov. On the representation of continuous functions of several variables by superposition of continuous functions of one variable and addition. *Doklady Akademiia Nauk SSSR*, 114(5):953–956, 1957.

[41] Věra Kůrková. Kolmogorov's theorem is relevant. *Neural Computation*, 3(4):617–622, 1991.

[42] Věra Kůrková. Kolmogorov's theorem and multilayer neural networks. *Neural Computation*, 5(3):501–506, 1992.

[43] Chuck Lam and David G. Stork. Learning network topology. In Michael A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*. MIT Press, Cambridge, MA, second edition, 2001.

[44] Alan Lapedes and Ron Farber. How neural nets work. In Dana Z. Anderson, editor, *Advances in Neural Information Processing Systems*, pages 442–456. American Institute of Physics, New York, 1988.

[45] Dar-Shyang Lee, Sargur N. Srihari, and Roger Gaborski. Bayesian and neural network pattern recognition: A theoretical connection and empiricial results with handwritten characters. In Ishwar K. Sethi and Anil K. Jain, editors, *Artificial Neural Networks and Statistical Pattern Recognition: Old and New Connections*, chapter 5, pages 89–108. North-Holland, Amsterdam, 1991.

[46] Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly Journal of Applied Mathematics*, II(2):164–168, 1944.

[47] Daniel S. Levine. *Introduction to Neural and Cognitive Modeling*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1991.

[48] Richard Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4–22, April 1987.

[49] David Lowe and Andrew R. Webb. Optimized feature extraction and the Bayes decision in feed-forward classifier networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-13(4):355–364, 1991.

[50] Donald W. Marquardt. An algorithm for least-squares estimation of non-linear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[51] Warren S. McCulloch and Walter Pitts. A logical calculus of ideas imminent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[52] John M. McInerny, Karen G. Haines, Steve Biafore, and Robert Hecht-Nielsen. Back propagation error surfaces can have local minima. In *International Joint Conference on Neural Networks (IJCNN)*, volume 2, page 627. IEEE, New York, 1989.

[53] Marvin L. Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, 1969.

[54] Hermann Ney. On the probabilistic interpretation of neural network classifiers and discriminative training criteria. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-17(2):107–119, 1995.

[55] David B. Parker. Learning logic. Technical Report S81-64, File 1, Stanford University Office of Technology Licensing, 1982.

[56] David B. Parker. Learning logic. Technical Report TR-47, MIT Center for Research in Computational Economics and Management Science, 1985.

[57] Fernando Pineda. Recurrent backpropagation and the dynamical approach to adaptive neural computation. *Neural Computation*, 1(2):161–172, 1989.

[58] Walter Pitts and Warren S. McCulloch. How we know universals: The perception of auditory and visual forms. *Bulletin of Mathematical Biophysics*, 9:127–147, 1947.

[59] Tomaso Poggio and Federico Girosi. Regularization algorithms for learning that are equivalent to multilayer networks. *Science*, 247(4945):978–982, 1990.

[60] Russell Reed. Pruning algorithms—a survey. *IEEE Transactions on Neural Networks*, TNN-4(5):740–747, 1993.

[61] Russell D. Reed and Roberg J. Marks II. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press, Cambridge, MA, 1999.

[62] Michael D. Richard and Richard P. Lippmann. Neural network classifiers estimate Bayesian *a-posteriori* probabilities. *Neural Computation*, 3(4):461–483, 1991.

[63] Brian D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge, UK, 1996.

[64] Frank Rosenblatt. The Perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[65] Frank Rosenblatt. *Principles of Neurodynamics*. Spartan Books, Washington, DC, 1962.

[66] Dennis W. Ruck, Steven K. Rogers, Matthew Kabrisky, Mark E. Oxley, and Bruce W. Suter. The multilayer Perceptron as an approximation to a Bayes optimal discriminant function. *IEEE Transactions on Neural Networks*, TNN-1(4):296–298, 1990.

[67] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by back-propagating errors. *Nature*, 323(99):533–536, 1986.

[68] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In David E. Rumelhart, James L. McClelland, and the PDP Research Group, editors, *Parallel Distributed Processing*, volume 1, chapter 8, pages 318–362. MIT Press, Cambridge, MA, 1986.

[69] Oliver G. Selfridge. Pandemonium: A paradigm for learning. In *Mechanisation of Thought Processes: Proceedings of a Symposium held at the National Physical Laboratory*, pages 513–526, London, 1958. HMSO.

[70] Oliver G. Selfridge and Ulrich Neisser. Pattern recognition by machine. *Scientific American*, 203(2):60–68, 1960.

[71] Ishwar K. Sethi and Anil K. Jain, editors. *Artificial Neural Networks and Statistical Pattern Recognition: Old and New Connections*. North-Holland, Amsterdam, The Netherlands, 1991.

[72] Suzanne Sommer and Richard M. Huggins. Variables selection using the Wald test and a robust CP. *Applied Statistics*, 45(1):15–29, 1996.

[73] Achim Stahlberger and Martin Riedmiller. Fast network pruning and feature extraction using the unit-OBS algorithm. In Michael C. Mozer, Michael I. Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9, pages 655–661. MIT Press, Cambridge, MA, 1997.

[74] David G. Stork. *Determination of symmetry and phase in human visual response functions: Theory and Experiment*. Ph.D. thesis, University of Maryland, College Park, MD, 1984.

[75] David G. Stork. Is backpropagation biologically plausible? In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, pages II–241–246. IEEE, New York, 1989.

[76] David G. Stork and James D. Allen. How to solve the n-bit parity problem with two hidden units. *Neural Networks*, 5(6):923–926, 1992.

[77] David G. Stork and James D. Allen. How to solve the n-bit encoder problem with just one hidden unit. *Neurocomputing*, 5(3):141–143, 1993.

[78] David G. Stork and John Z. Levinson. Receptive fields and the optimal stimulus. *Science*, 216(4542):204–205, 1982.

[79] Alan M. Turing. Intelligent machinery. In Darrell C. Ince, editor, *Collected Works of A. M. Turing: Mechanical Intelligence*. Elsevier Science Publishers, Amsterdam, The Netherlands, 1992.

[80] Abraham Wald. Tests of statistical hypotheses concerning several parameters when the number of observations is large. *Transactions of the American Mathematical Society*, 54(3):426–482, 1943.

[81] Abraham Wald. *Statistical Decision Functions*. Wiley, New York, 1950.

[82] Andreas S. Weigend, David E. Rumelhart, and Bernardo A. Huberman. Generalization by weight-elimination with application to forecasting. In Richard P. Lippmann, John E. Moody, and David S. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 3, pages 875–882. Morgan Kaufmann, San Mateo, CA, 1991.

[83] Paul John Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph.D. thesis, Harvard University, Cambridge, MA, 1974.

[84] Paul John Werbos. *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. Wiley, New York, 1994.

[85] Halbert L. White, Jr. Learning in artifical neural networks: A statistical perspective. *Neural Computation*, 3(5):425–464, 1989.

[86] Bernard Widrow. 30 years of adaptive neural networks: Perceptron, Madaline, and Backpropagation. *Proceedings of IEEE*, 78(9):1415–1452, 1990.

[87] Bernard Widrow and Marcian E. Hoff, Jr. Adaptive switching circuits. *1960 IRE WESCON Convention Record*, pages 96–104, 1960.

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