Chapter 6

Multilayer Neural Networks

6.1 Introduction

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

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

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

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

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

BACKPROPAGATION

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

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

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

REGULARIZATION

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

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

6.2 Feedforward operation and classification

HIDDEN LAYER

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

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

BIAS

NEURON

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

RECALL

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

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

NET ACTIVATION

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

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

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

SYNAPSE

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

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

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

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

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

TRANSFER FUNCTION

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

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

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

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

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

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

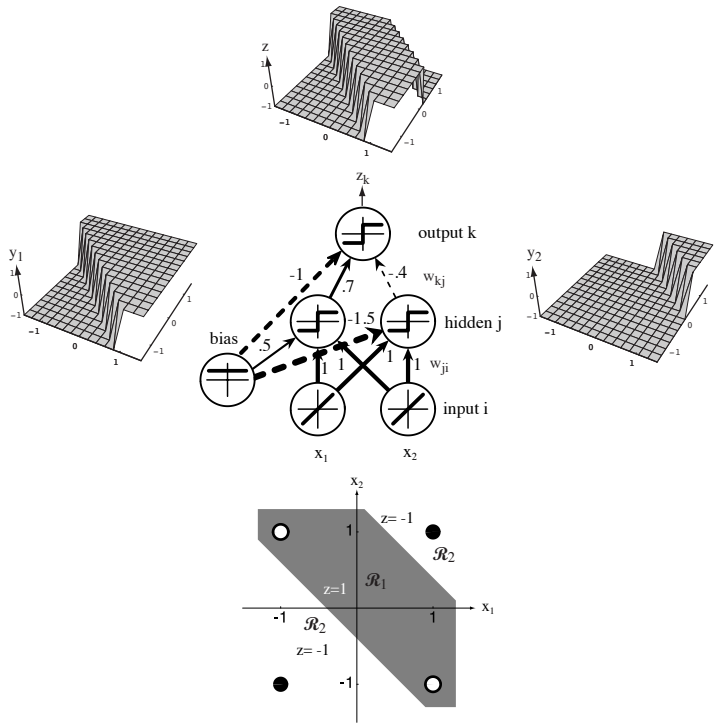


Figure 6.1: The two-bit parity or exclusive-OR problem can be solved by a three-layer network. At the bottom is the two-dimensional feature space $x_1 - x_2$, and the four patterns to be classified. The three-layer network is shown in the middle. The input units are linear and merely distribute their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

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

6.2.1 General feedforward operation

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

EXPRESSIVE POWER

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

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

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

6.2.2 Expressive power of multilayer networks

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

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

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

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

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

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

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

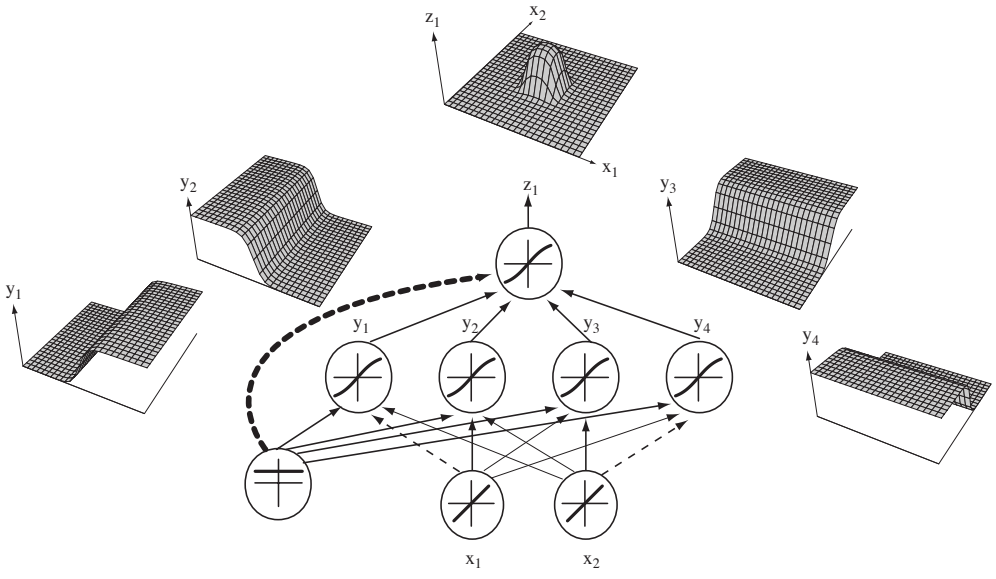


Figure 6.2: A 2-4-1 network (with bias) along with the response functions at different units; each hidden and output unit has sigmoidal transfer function $f(\cdot)$. In the case shown, the hidden unit outputs are paired in opposition thereby producing a "bump" at the output unit. Given a sufficiently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

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

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

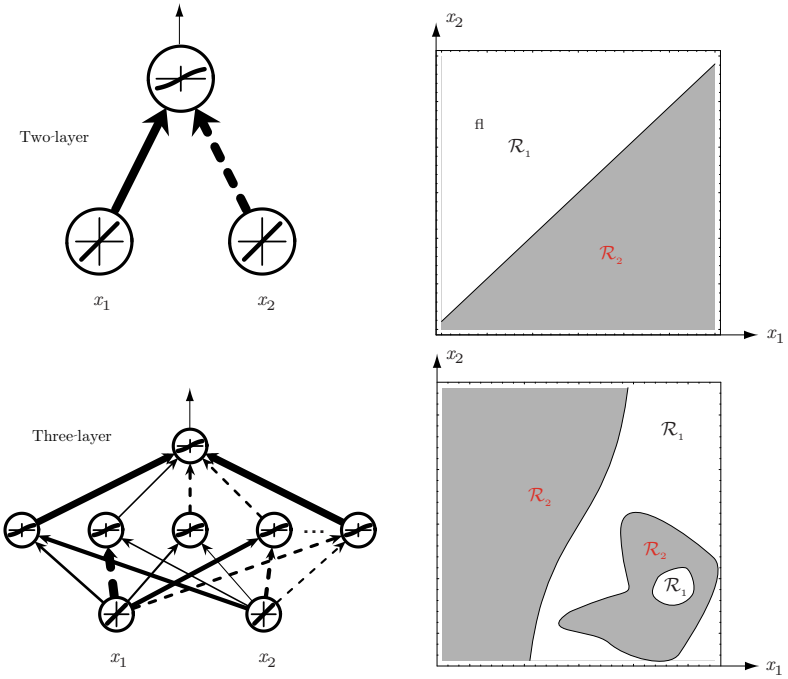


Figure 6.3: Whereas a two-layer network classifier can only implement a linear decision boundary, given an adequate number of hidden units, three-, four- and higher-layer networks can implement arbitrary decision boundaries. The decision regions need not be convex, nor simply connected.

6.3 Backpropagation algorithm

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

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 Chap. **??**. Other methods may be faster or have other desirable properties, but few are more instructive. The LMS algorithm worked for two-layer systems because we had an error (proportional to the square of the difference between the actual output and the desired output) evaluated at the output unit. Similarly, in a three-layer net it is a straightforward matter to find how the output (and thus 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.

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 as well as a desired, teaching or *target* pattern to the output layer and changing the network parameters (e.g., weights) in order to make the actual output more similar to the target one. 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 an input training pattern and determine the output. The error or criterion function is some scalar function of the weights that is minimized when the network outputs match the desired outputs. The weights are adjusted to reduce this measure of error. Here we present the learning rule on a per pattern basis, and return to other protocols later.

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:

TRAINING ERROR

$$J(\mathbf{w}) \equiv 1/2 \sum_{k=1}^{c} (t_k - z_k)^2 = 1/2(\mathbf{t} - \mathbf{z})^2, \tag{8}$$

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

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

$$\Delta \mathbf{w} = -\eta \frac{\partial J}{\partial \mathbf{w}}, \tag{9}$$

or in component form

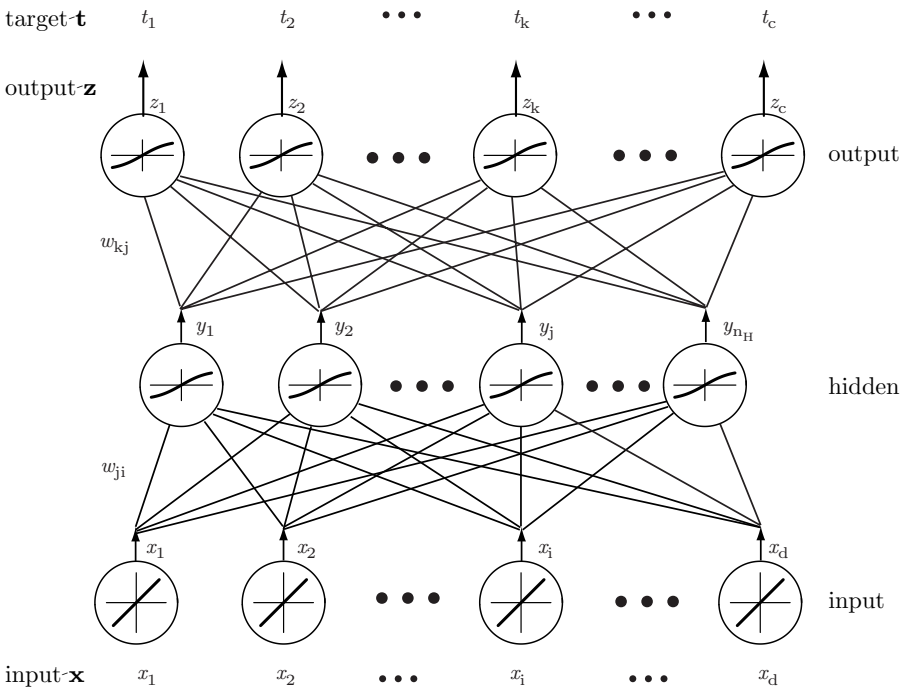


Figure 6.4: A d-n_H-c fully connected three-layer network and the notation we shall use (bias not shown). During feedforward operation, a d-dimensional input pattern $\mathbf{x}$ is presented to the input layer; each input unit then emits its corresponding component x_i. Each of the n_H hidden units 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 transfer 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.

$$\Delta w_{mn} = -\eta \frac{\partial J}{\partial w_{mn}}, \tag{10}$$

LEARNING RATE

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

$$\mathbf{w}(m+1) = \mathbf{w}(m) + \Delta\mathbf{w}(m), \tag{11}$$

where m indexes the particular pattern presentation (but see also Sect. 6.8).

We now turn to the problem of evaluating Eq. 10 for a three-layer net. Consider first the hidden-to-output weights, w_{jk}. Because the error is not explicitly dependent upon w_{jk}, we must use the chain rule for differentiation:

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = \delta_k \frac{\partial net_k}{\partial w_{kj}}, \tag{12}$$

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

$$\delta_k \equiv -\partial J/\partial net_k, \tag{13}$$

and describes how the overall error changes with the unit's activation. We differentiate Eq. 8 and find that for such an output unit δ_k is simply:

$$\delta_k \equiv -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} = (t_k - z_k)f'(net_k). \tag{14}$$

The last derivative in Eq. 12 is found using Eq. 4:

$$\frac{\partial net_k}{\partial w_{kj}} = y_j. \tag{15}$$

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

$$\Delta w_{kj} = \eta \delta_k y_j = \eta (t_k - z_k) f'(net_k) y_j. \tag{16}$$

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. 10, and again using the chain rule, we calculate

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}. \tag{17}$$

The first term on the right hand side requires just a bit of care:

$$\begin{aligned}
\frac{\partial J}{\partial y_j} &= \frac{\partial}{\partial y_j}\left[1/2\sum_{k=1}^{c}(t_k - z_k)^2\right] \\
&= -\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial y_j} \\
&= -\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial y_j} \\
&= -\sum_{k=1}^{c}(t_k - z_k)f'(net_k)w_{jk}.
\end{aligned} \tag{18}$$

For the second step above we had to use the chain rule yet again. The final sum over output units in Eq. 18 expresses how the hidden unit output, y_j, affects the error at each output unit. In analogy with Eq. 13 we use Eq. 18 to define the sensitivity for a hidden unit as:

$$\delta_j \equiv f'(net_j)\sum_{k=1}^{c} w_{kj}\delta_k. \tag{19}$$

Equation 19 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_{jk}, all multiplied by $f'(net_j)$. Thus the learning rule for the input-to-hidden weights is:

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

Equations 16 & 20, together with training protocols such as described below, give the backpropagation algorithm — or more specifically the "backpropagation of errors" algorithm — so-called because during training an "error" (actually, the sensitivities δ_k) 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. 20 (Fig. 6.5). At base then, backpropagation is "just" gradient descent in layered models where the chain rule through continuous functions allows the computation of derivatives of the criterion function with respect to all model parameters (i.e., weights).

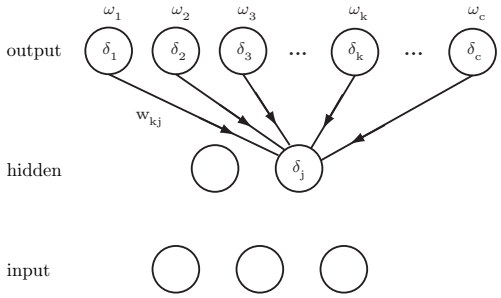


Figure 6.5: The sensitivity at a hidden unit is proportional to the weighted sum of the sensitivities at the output units: $\delta_j = f'(net_j) \sum_{k=1}^{c} w_{kj} \delta_k$. The output unit sensitivities are thus propagated "back" to the hidden units.

These learning rules make intuitive sense. Consider first the rule for learning weights at the output units (Eq. 16). The weight update at unit k should indeed be proportional to $(t_k - z_k)$ — if we get the desired output ($z_k = t_k$), then there should be no weight change. For a 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. 20 yields insight of the input-to-hidden weights (Problem 5).

Problem 7 asks you to show that the presence of the bias unit does not materially affect the above results. Further, with moderate notational and bookkeeping effort (Problem 11), the above learning algorithm can be generalized directly to feed-forward networks in which

- 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.

It is a more subtle matter to perform 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 Sect. **??**.

6.3.2 Training protocols

In broad overview, supervised training consists in presenting to the network 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 or teaching signal. The three most useful training protocols are: stochastic, batch and on-line. In *stochastic training* (or pattern 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 (weight update) takes place. 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.*

TRAINING SET

STOCHASTIC TRAINING

BATCH TRAINING

ON-LINE PROTOCOL

A fourth protocol is *learning with queries* where the output of the network is used to *select* new training patterns. Such queries generally focus on points that are likely to give the most information to the classifier, for instance those near category decision boundaries (Chap. **??**). While this protocol may be faster in many cases, its drawback is that the training samples are no longer independent, identically distributed (i.i.d.), being skewed instead toward sample boundaries. This, in turn, generally distorts the effective distributions and may or may not improve recognition accuracy (Computer exercise **??**).

LEARNING WITH QUERIES

We describe the overall amount of pattern presentations by *epoch* — the number of presentations of the full 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.

EPOCH

Algorithm 1 (Stochastic backpropagation)

```
1 begin initialize  network topology (# hidden units), w, criterion θ, η, m ← 0
2   do m ← m + 1
3       x^m ← randomly chosen pattern
4       w_ij ← w_ij + ηδ_j x_i;   w_jk ← w_jk + ηδ_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 pre-set value θ. While this is perhaps the simplest meaningful *stopping criterion*, others generally lead to

STOPPING CRITERION

* Some on-line training algorithms are considered models of biological learning, where the organism is exposed to the environment and cannot store all input patterns for multiple "presentations."

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

better performance, as we shall discuss in Sect. 6.8.14.

In the batch version, 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. With minor infelicities in notation we can write this total training error as the sum over the errors on n individual patterns:

$$J = \sum_{p=1}^{n} J_p. \tag{21}$$

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. 21 decreases.

Algorithm 2 (Batch backpropagation)

```
1  begin initialize  network topology (# hidden units), w, criterion θ, η, r ← 0
2    do r ← r + 1 (increment epoch)
3       m ← 0; Δw_ij ← 0; Δw_jk ← 0
4       do m ← m + 1
5          x^m ← select pattern
6          Δw_ij ← Δw_ij + η δ_j x_i;   Δw_jk ← Δw_jk + η δ_k y_j
7       until m = n
8       w_ij ← w_ij + Δw_ij;   w_jk ← w_jk + Δw_jk
9    until ∇J(w) < θ
10 return w
11 end
```

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

6.3.3 Learning curves

Because the weights are initialized with random values, error on the training set is large; 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 (Chap. **??**). Since batch backpropagation performs gradient descent in the criterion function, these training error decreases 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.

VALIDATION ERROR

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 training on the training set is stopped when a minimum is reached in the validation error (e.g., near epoch 5 in

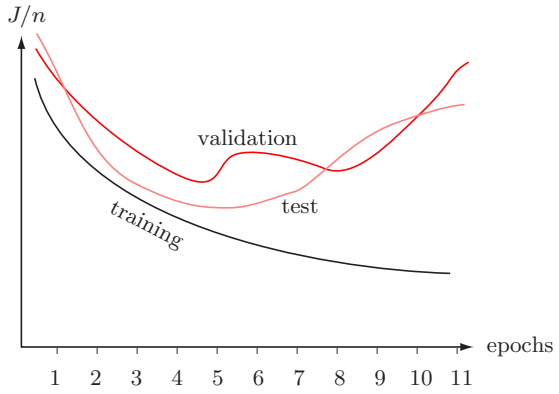


Figure 6.6: A learning curve shows the criterion function as a function of the amount of training, typically indicated by the number of epochs or presentations of the full training set. We plot the average error per pattern, i.e., $1/n \sum_{p=1}^{n} J_p$. The validation error and the test (or generalization)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

the figure). We shall return in Chap. **??** to understand in greater depth why this version of *cross validation* stopping criterion often leads to networks having improved recognition accuracy.

CROSS VALIDATION

6.4 Error surfaces

Since 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 training and 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 are local minima; if many local minima plague the error landscape, then it is unlikely that the network will find the *global* minimum. Does this necessarily lead 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 & 2 to be slow. Since 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 somewhat below $x_1 = 0$) separates the two categories. During learning, the weights descends to the global minimum, and the problem is solved.

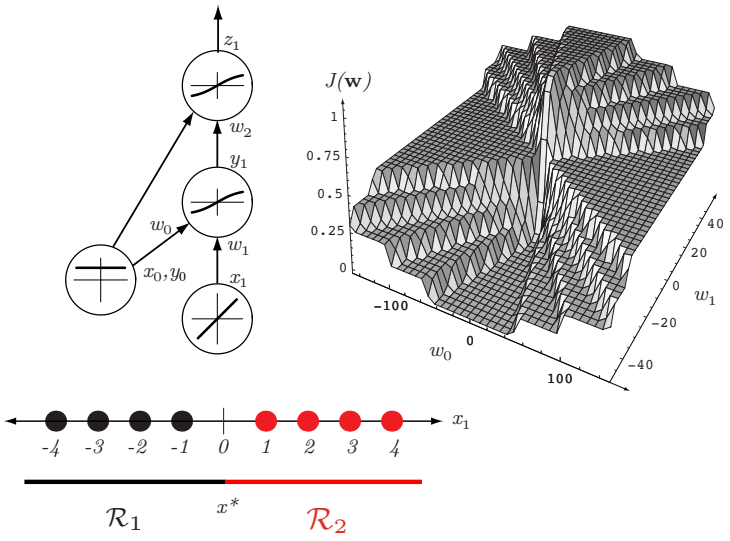


Figure 6.7: Six one-dimensional patterns (three in each of two classes) are to be learned by a 1-1-1 network with sigmoidal hidden and output units (and bias). The error surface as a function of w_1 and w_2 is also shown (for the case where the bias weights have their final values). The network starts with random weights, and through (stochastic) training descends to the global minimum in error, as shown by the trajectory. Note especially that a low error solution exists, which in fact leads to a decision boundary separating the training points into their two categories.

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 three 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 classification error (cf. Chap. **??**, Fig. **??**). 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

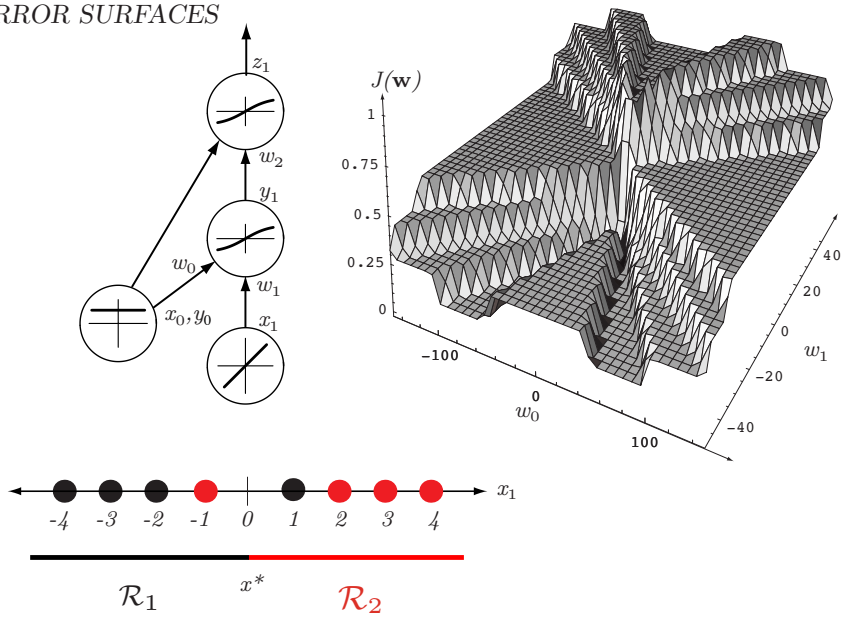


Figure 6.8: As in Fig. 6.7, except here the patterns are not linearly separable; the error surface is slightly higher than in that figure.

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 XOR

A somewhat more complicated problem is the XOR problem we have already considered. Figure **??** 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 & 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 **??**).

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. Figure 6.10 shows a network with many weights solving a complicated high-dimensional two-category pattern classification problem. Here, the error varies quite gradually as a single weight is changed though we can get troughs, valleys, canyons, and a host of

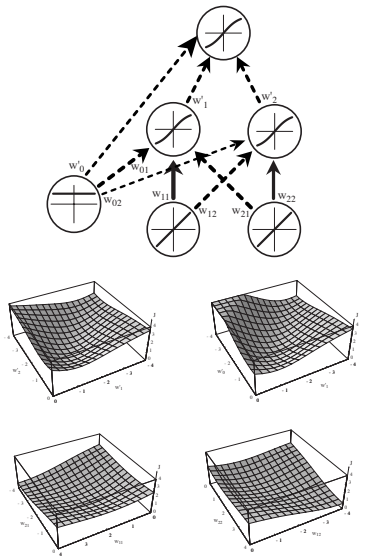


Figure 6.9: Two-dimensional slices through the nine-dimensional error surface after extensive training for a 2-2-1 network solving the XOR problem.

shapes.

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. In networks with many superfluous weights (i.e., more than are needed to learn the training set), one is less likely to get into local minima. However, networks with an unnecessarily large number of weights are undesirable because of the dangers of overfitting, as we shall see in Sect. 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 since this usually indicates that key features of the problem have not been learned by the network. In such cases it is traditional to re-initialize the weights and train again, possibly also altering other parameters in the net (Sect. 6.8).

In many problems, convergence to a non-global 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 (Sect. 6.8.14).

Figure 6.10: A network with xxx weights trained on data from a complicated pattern recognition problem xxx.

6.5 Backpropagation as feature mapping

Since 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 6.11 shows a three-layer net addressing the XOR problem. For any input pattern in the $x_1 - x_2$ space, we can show the corresponding output of the two hidden units in the $y_1 - y_2$ space. With small initial weights, the net activation of each hidden unit is small, and thus the *linear* portion of their transfer 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 = +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. 6.12). 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 and the weights have become larger, the nonlinearities of the hidden units are expressed and patterns have been moved and can be linearly separable, as shown.

Figure 6.13 shows a 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

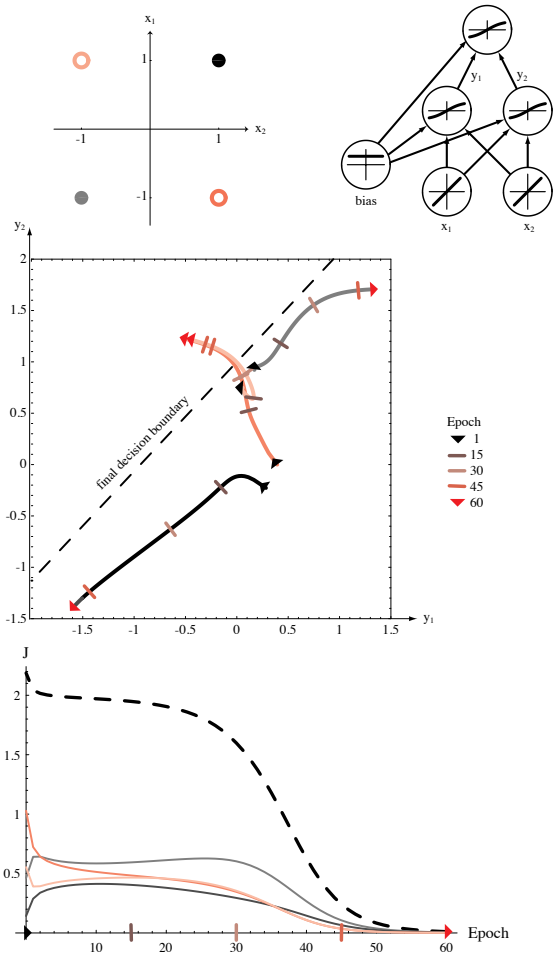


Figure 6.11: A 2-2-1 backpropagation network (with bias) and the four patterns of the XOR problem are shown at the top. The middle figure shows the outputs of the hidden units for each of the four patterns; these outputs move across the $y_1 - y_2$ space as the full network learns. In this space, early in training (epoch 1) the two categories are not linearly separable. As the input-to-hidden weights learn, the categories become linearly separable. Also shown (by 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. While the error on each individual pattern does not decrease monotonically, the total training error does decrease monotonically.

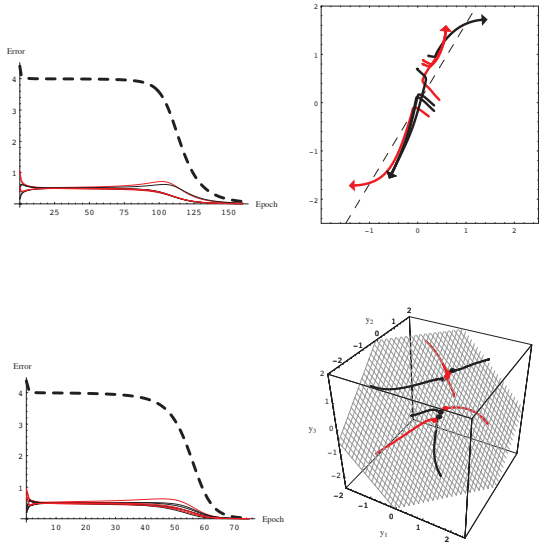


Figure 6.12: A 3-3-1 backpropagation network (plus bias) can indeed solve the three-bit parity problem. The representation of the eight patterns at the hidden units ($y_1 - y_2 - y_3$ space) as the system learns and the (planar) decision boundary found by the hidden-to-output weights at the end of learning. The patterns of the two classes are separated by this plane. The learning curve shows the error on individual patterns and the total error as a function of epoch.

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.

6.5.1 Representations at the hidden layer — weights

In addition to focusing on the transformation of patterns, we can also consider the representation of learned weights themselves. Since the hidden-to-output weights merely leads 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." Because the hidden unit transfer functions are nonlinear, the correspondence with classical methods such as matched filters (and principal components, Sect. **??**) is not exact; nevertheless it is often 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 6.14 shows the input-to-hidden weights (displayed as patterns) for a simple task of character recognition. Note that one hidden unit seems "tuned" for a pair of horizontal bars while the other 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

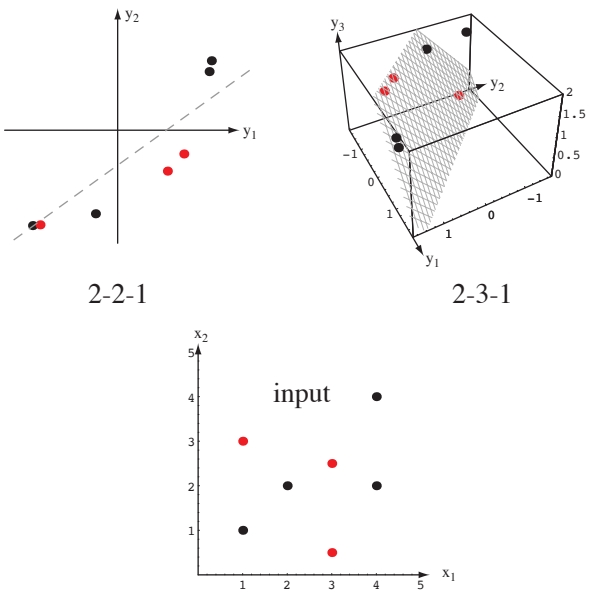


Figure 6.13: Seven patterns from a two-dimesional two-category nonlinearly separable classification problem are shown at the bottom. The figure at the top left shows the hidden unit representations of the patterns in a 2-2-1 sigmoidal network (with bias) fully trained to the global error minimum; the linear boundary implemented by the hidden-to-output weights is also shown. 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.

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 (degrees of freedom), and thus the feature selectivity is low.

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 of the hidden units. Together with the 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 form a least squares fit to the Bayes discriminant functions.

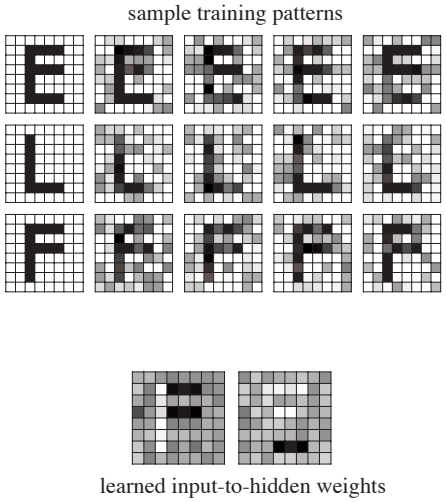


Figure 6.14: The top images represent patterns from a large training set used to train a 64-2-3 sigmoidal network for classifying three characters. The bottom figures show the input-to-hidden weights (represented as patterns) at the two hidden units after training. Note that these learned weights indeed describe feature groupings useful for the classification task. In large networks, such patterns of learned weights may be difficult to interpret in this way.

6.6.1 Bayes discriminants and neural networks

As we saw in Chap. **??** Sect. **??**, 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,

$$P(\omega_k|\mathbf{x}) = \frac{P(\mathbf{x}|\omega_k)P(\omega_k)}{\sum_{i=1}^{c} P(\mathbf{x}|\omega_i)P(\omega_i)} = \frac{P(\mathbf{x},\omega_k)}{P(\mathbf{x})}, \tag{22}$$

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:

$$t_k(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \omega_k \\ 0 & \text{otherwise.} \end{cases} \tag{23}$$

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

$$J(\mathbf{w}) \quad = \quad \sum_{\mathbf{x}} [g_k(\mathbf{x};\ \mathbf{w}) - t_k]^2 \tag{24}$$

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

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 (Eq. 22) to express Eq. 24 as (Problem 17):

$$
\begin{aligned}
\lim_{n\to\infty}\frac{1}{n}J(\mathbf{w}) &\equiv \tilde{J}(\mathbf{w}) \qquad (25)\\
&= P(\omega_k)\int\left[g_k(\mathbf{x};\ \mathbf{w}) - 1\right]^2 p(\mathbf{x}|\omega_k)d\mathbf{x} + P(\omega_{i\neq k})\int g_k^2(\mathbf{x};\ \mathbf{w})p(\mathbf{x}|\omega_{i\neq k})d\mathbf{x}\\
&= \int g_k^2(\mathbf{x};\ \mathbf{w})p(\mathbf{x})d\mathbf{x} - 2\int g_k(\mathbf{x};\ \mathbf{w})p(\mathbf{x},\omega_k)d\mathbf{x} + \int p(\mathbf{x},\omega_k)d\mathbf{x}\\
&= \int\left[g_k(\mathbf{x};\ \mathbf{w}) - P(\omega_k|\mathbf{x})\right]^2 p(\mathbf{x})d\mathbf{x} + \underbrace{\int P(\omega_k|\mathbf{x})P(\omega_{i\neq k}|\mathbf{x})p(\mathbf{x})d\mathbf{x}}_{\text{independent of }\mathbf{w}}.
\end{aligned}
$$

The backpropagation rule changes weights to minimize the left hand side of Eq. 25, and thus it minimizes

$$\int\left[g_k(\mathbf{x};\ \mathbf{w}) - P(\omega_k|\mathbf{x})\right]^2 p(\mathbf{x})d\mathbf{x}. \qquad (26)$$

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

$$\sum_{k=1}^{c}\int\left[g_k(\mathbf{x};\ \mathbf{w}) - P(\omega_k|\mathbf{x})\right]^2\ p(\mathbf{x})d\mathbf{x}. \qquad (27)$$

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

$$g_k(\mathbf{x};\ \mathbf{w}) \simeq P(\omega_k|\mathbf{x}). \qquad (28)$$

Figure 6.15 illustrates the development of the learned outputs toward the Bayes discriminants as the amount of training data and the expressive power of the net increases.

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 (Problem **??**). Moreover, fitting the discriminant function does not guarantee the optimal classification *boundaries* are found, just as we saw in Chap. **??**.

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 (and assuming the net can express the discriminants, does

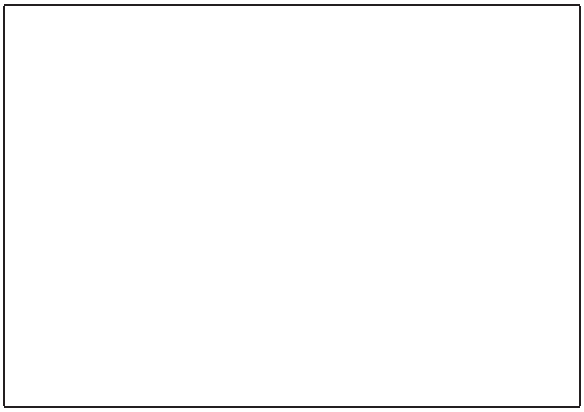


Figure 6.15: As a network is trained via backpropagation (under the assumptions given in the text), its outputs more closely approximate posterior probabilities. The figure shows the outputs of a 1-3-2 and a 1-8-2 sigmoidal network after backpropagation training with $n = 10$ and $n = 1000$ points from two categories. Note especially the excellent agreement between the large net's outputs and the Bayesian discriminant functions in the regions of high $p(x)$.

not fall into an undesirable local minimum, etc.), then 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 no guarantee that they will 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 (Sect. 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 normalize the outputs to sum to 1.0,

$$z_k = \frac{e^{net_k}}{\sum_{m=1}^{c} e^{net_m}}, \tag{29}$$

and to train 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, Computer exercise 10).

SOFTMAX

WINNER-TAKE-ALL

A neural network classifier trained in this manner approximates the posterior probabilities $P(\omega_i|\mathbf{x})$, whether or not the data was sampled from unequal priors $P(\omega_i)$. 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 (Computer exercise 11).

6.7 *Related statistical techniques

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. 6. A number of statistical methods bear similarities to that equation. For instance, *projection pursuit regression* (or simply projection pursuit) implements

PROJECTION PURSUIT

$$z = \sum_{j=1}^{j_{max}} w_j f_j(\mathbf{v}_j^t \mathbf{x} + u_{j0}) + w_0. \tag{30}$$

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 $f_j(\cdot)$, one obtains ridges in two dimensions. Equation 30 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 instance 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.

RIDGE FUNCTION

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.

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

GENERALIZED ADDITIVE MODEL

$$z = f\left(\sum_{i=1}^{d} f_i(x_i) + w_0\right), \tag{31}$$

where again $f(\cdot)$ is often chosen to be a sigmoid, and the functions $f_i()$ 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 Sect. 6.2 implement a special case of general additive models (Problem 24), though the training methods differ.

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

MULTIVARIATE ADAPTIVE REGRESSION SPLINE

$$z = \sum_{k=1}^{M} w_k \prod_{r=1}^{r_k} \phi_{kr}(x_{q(k,r)}) + w_0, \tag{32}$$

where the kth basis function is the product of r_k one-dimensional spline functions ϕ_{kr}; w_0 is a scalar offset. The splines depend on the input values x_q, such as the feature component of an input, where the index is labeled $q(k, r)$. 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. 32. 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 using 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 (Sect. 6.8.12); nets admit a variety of simplification or regularization methods (Sec. 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

When creating a multilayer neural network classifier, the designer must make two major types of decision: selection of the architecture and selection of parameters (though the distinction is not always crisp or important). Our goal here is to give a principled basis for making such choices based on learning speed and optimal recognition performance. In practice, while parameter adjustment is problem dependent several rules of thumb emerge from an analysis of networks.

6.8.1 Transfer function

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

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, i.e., have some maximum and minimum output value. This will keep the weights and activations bounded, and thus keep training time limited. (This property is less desirable in networks used for regression, since there we may seek outputs values greater than any saturation level selected before training.) A third property is continuity and smoothness, i.e., 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 transfer functions, but with added complexity and few benefits.

Monotonicity is another convenient (but non-essential) property for $f(\cdot)$ — we might wish the derivative have the same sign throughout the range of the argument, e.g., $f'(\cdot) \geq 0$. If f is *not* monotonic, additional (and undesirable) local extremum in the error surface may become introduced (Computer Exercise **??**). Non-monotonic transfer functions such as radial basis functions can be used if proper care is taken (Sect. 6.10.1). Another desirable property is linearity for small value of *net*, which will enable the system to implement a linear model if adequate for low error. A property that is might occasionally be of importance is computational simplicity — we seek a function whose value and derivative can be easily computed.

POLYNOMIAL CLASSIFIER

We mention in passing that *polynomial classifiers* use transfer functions of the form $x_1, x_2, ..., x_d, x_1^2, x_2^2, ..., x_d^2, x_1x_2, ..., x_1x_d$, and so forth — all terms up to some limit; training is via gradient descent too. One drawback is that the outputs of the hidden units (ϕ functions) can become extremely large even for realistic problems (Problem 29, Computer exercise **??**). Instead, standard neural networks employ the same nonlinearity at each hidden unit.

SIGMOID

One class of function that has all the above 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). One last benefit of the sigmoid is that it maximizes information transmission for features that are normally distributed (Problem 25).

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 transfer functions that have 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 (Computer exercise 14).

LOCAL REPRESENTATION

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

6.8.2 Parameters for the sigmoid

Given that we will use the sigmoidal form, there remain a number of parameters to set. It is best to keep the function centered on zero and anti-symmetric, i.e., $f(-net) = -f(net)$, rather than one whose value is always positive. Together with the data preprocessing described in Sec. 6.8.3, anti-symmetric sigmoids speed learning by eliminating the need to learn the mean values of the training data. Thus, sigmoid functions of the form

$$f(net) = a \tanh(b\ net) = a\left[\frac{1 - e^{b\ net}}{1 + e^{b\ net}}\right] = \frac{2a}{1 + e^{-b\ net}} - a \tag{33}$$

work well. The *overall* range and slope are not important, since it is their relationship to parameters such as the learning rate and magnitudes of the inputs and targets that determine learning times (Problem 23). For convenience, though, we choose $a = 1.716$ and $b = 2/3$ in Eq. 33 — values which insure $f'(0) \simeq 1$, that the linear

range is $-1 < net < +1$, and that the extrema of the second derivative occur roughly at $net \simeq \pm 2$ (Fig. 6.16).

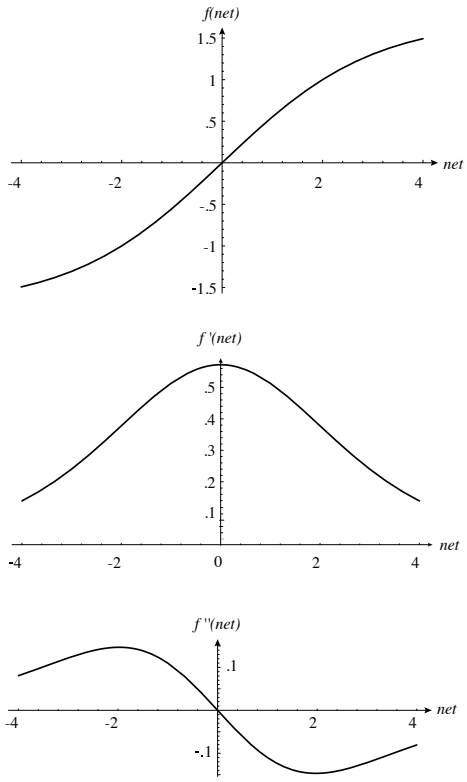


Figure 6.16: A useful transfer function $f(net)$ is an anti-symmetric sigmoid. For the parameters given in the text, $f(net)$ is nearly linear in the range $-1 < net < +1$ and its second derivative, $f''(net)$, has extrema near $net \simeq \pm 2$.

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, since they differ solely in the arbitrary representation. The difficulty arises even for features having the same units but differing 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 Sect. 6.8.8. That is, we *standardize* the STANDARDIZE

training patterns. This data standardization is done once, before actually network training, and thus represents a small one-time computational burden (Problem 27, Computer exercise 15). Standardization can only be done for stochastic and batch learning protocols, but not on-line protocols where the full data set is never available at any one time.

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. Since 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 would be less than the saturation values, and thus there would be error. Full training would never terminate as 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: $\mathbf{t} = (-1, -1, +1, -1)$. Of course, this target representation yields efficient learning for categorization — the outputs here do not represent posterior probabilities (Sec. 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 Sect. 6.8.3, the variance of the added noise should be less than 1.0 (e.g., 0.1) and the category label 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 (Problem 30).

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 (Sec. 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, since 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 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 hiddens are needed. Thus without further information there is no foolproof method for setting the number of hidden units before training.

Figure 6.17 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 data. Nevertheless, in this regime, the test error is unacceptably high, an example of overfitting we shall study again in Chap. **??**. 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.

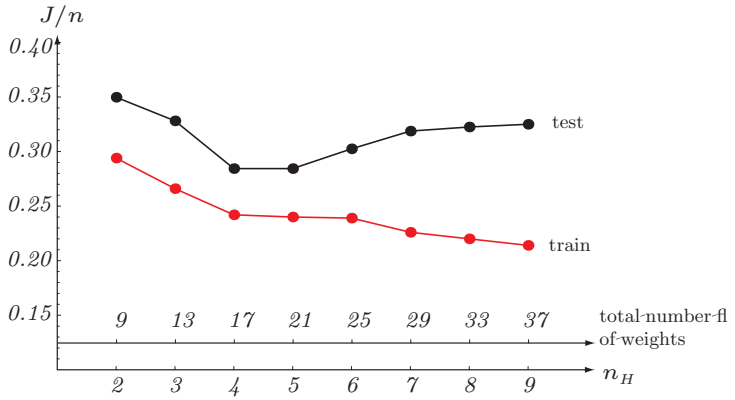


Figure 6.17: The error per pattern for networks fully trained but differing in the numbers of hidden units, n_H. Each 2–n_H–1 network (with bias) was trained with 90 two-dimensional patterns from each of two categories (sampled from a mixture of three Gaussians); thus $n = 180$. The minimum of the test 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 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. 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 hiddens and prune or eliminate weights — techniques we shall study in Sect. **??** and Chap. **??**.

6.8.8 Initializing weights

Suppose we have fixed the network topology, and thus set the number of hidden units. We now seek to set the initial weight values in order to have fast and *uniform learning*, i.e., all weights reach their final equilibrium values at about the same time.

UNIFORM LEARNING

One form of non-uniform learning occurs when category ω_i is learned well before ω_j. 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 insure uniform learning.)

In setting weights in a given layer, we choose weights randomly from a *single* distribution to help insure 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. Hence we set $\tilde{w}$ such that the net activation at a hidden unit is in the range $-1 < net_j < +1$, since $net_j \simeq \pm 1$ are the limits to its linear range (Fig. 6.16).

In order to calculate $\tilde{w}$, consider a hidden unit having a fan-in of d inputs. Suppose too that all weights have the same value $\tilde{w}$. 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 $\tilde{w} = 1/\sqrt{d}$ and 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 the fan-in is n_H; hidden-to-output weights should initialized with values chosen in the range $-1/\sqrt{n_H} < w_{kj} < +1/\sqrt{n_H}$.

6.8.9 Learning rates

In principle, so long as the learning rate is small enough to assure 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 fully trained to a training error minimum (Sect. 6.8.14), the learning rate can affect the quality of the final network. If some weights converge significantly earlier than others (non-uniform learning) then the network may not perform equally well throughout the full range of inputs, or equally well for patterns in each category. Figure 6.18 shows the effect of different learning rates on convergence in a single dimension.

NONUNIFORM LEARNING

The optimal learning rate is the one which 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 which thus gives (Fig. 6.19):

$$\frac{\partial^2 J}{\partial w^2}\Delta w = \frac{\partial J}{\partial w}. \tag{34}$$

The optimal rate is found directly to be

$$\eta_{opt} = \left(\frac{\partial^2 J}{\partial w^2}\right)^{-1}. \tag{35}$$

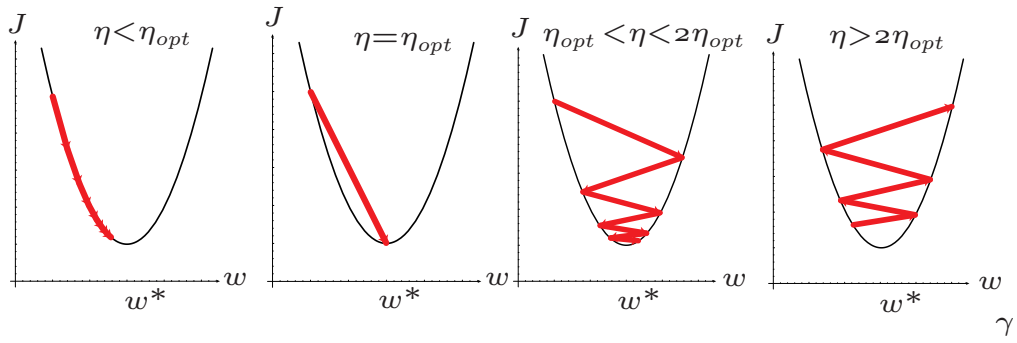


Figure 6.18: Gradient descent in a one-dimensional quadratic criterion with different learning rates. If $\eta < \eta_{opt}$, convergence is assured, but training can be needlessly slow. If $\eta = \eta_{opt}$, a single learning step suffices to find the error minimum. If $\eta_{opt} < \eta < 2\eta_{opt}$, the system will oscillate but nevertheless converge, but training is needlessly slow. If $\eta > 2\eta_{opt}$, the system diverges.

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).

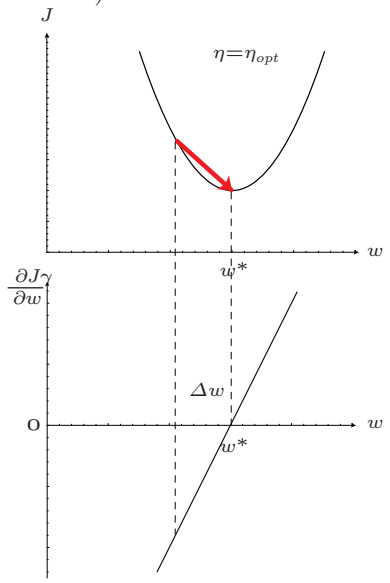


Figure 6.19: If the criterion function is quadratic (above), its derivative is linear (below). The optimal learning rate η_{opt} insures that the weight value yielding minimum error, w^* is found in a single learning step.

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 Sect. **??** to calculate second derivatives in networks, and to alternate descent and training methods such as Quickprop that give fast, uniform learning. 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, and lowered if the criterion function diverges, or 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 — for instance because of "too many" weights. 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:

$$\mathbf{w}(m+1) = \underbrace{\mathbf{w}(m) + \Delta\mathbf{w}(m)}_{\substack{\text{gradient} \\ \text{descent}}} + \underbrace{\alpha\Delta\mathbf{w}(m-1)}_{\text{momentum}} \tag{36}$$

Of course, α must be less than 1.0 for stability; typical values are $\alpha \simeq 0.9$. It must be stressed that momentum rarely changes the final solution, but merely allows it to be found more rapidly. Momentum provides another benefit: effectively "averaging out" stochastic variations in weight updates during stochastic learning and thereby speeding learning, even far from error plateaus (Fig. 6.20).

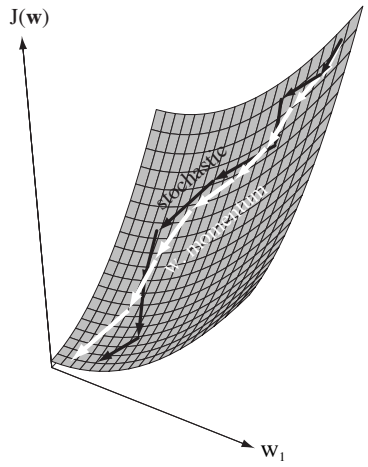


Figure 6.20: The incorporation of momentum into stochastic gradient descent by Eq. 36 (white arrows) reduces the variation in overall gradient directions and speeds learning, especially over plateaus in the error surface.

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

Algorithm 3 (Stochastic backpropagation with momentum)

1 **begin initialize** topology (# hidden units), $\mathbf{w}$, criterion, $\alpha(<1), \theta, \eta, m \leftarrow 0, b_{ji} \leftarrow 0, b_{kj} \leftarrow 0$
2 **do** $m \leftarrow m + 1$
3 $\mathbf{x}^m \leftarrow$ randomly chosen pattern
4 $b_{ji} \leftarrow \eta\delta_j x_i + \alpha b_{ji}$; $b_{kj} \leftarrow \eta\delta_k y_j + \alpha b_{kj}$
5 $w_{ji} \leftarrow w_{ji} + b_{ji}$; $w_{kj} \leftarrow w_{kj} + b_{kj}$
6 **until** $\boldsymbol{\nabla} J(\mathbf{w}) < \theta$

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 (or hidden units) and "decay" all weights during training. Small weights favor models that are more nearly linear (Problems 1 & 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:

$$w^{new} = w^{old}(1 - \epsilon), \tag{37}$$

where $0 < \epsilon < 1$. In this way, weights that are not needed for reducing the criterion 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 cannot decay indefinitely. In weight decay, then, the system achieves a balance between pattern error (Eq. 60) and some measure of overall weight. It can be shown (Problem 43) that the weight decay is equivalent to gradient descent in a new effective error or criterion function:

$$J_{ef} = J(\mathbf{w}) + \frac{2\epsilon}{\eta}\mathbf{w}^t\mathbf{w}. \tag{38}$$

The second term on the right hand side of Eq. 38 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:

$$\epsilon_{mr} = \frac{\gamma\eta/2}{(1 + w_{mr}^2)^2}. \tag{39}$$

We shall discuss principled methods for setting ϵ, and see how weight decay is an instance of a more general regularization procedure in Chap. **??**.

6.8.12 Hints

Often we have insufficient training data for adequate 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 related to the 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. 6.21).

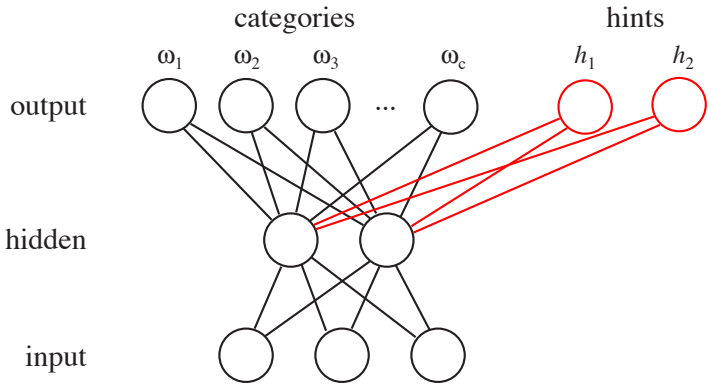


Figure 6.21: In learning with hints, the output layer of a standard network having c units (discriminant functions) is augmented with hint units. During training, the target vectors are also augmented with signals for the hint units. In this way the input-to-hidden weights learn improved feature groupings. During classification the hint units are not used, and thus they and their hidden-to-output weights are removed from the trained network.

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 (Computer exercise 16).

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 Sect. 6.3.2 has strengths and drawbacks. On-line learning is to be used when the amount of training data is so large, or that 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 $(\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^5)$. 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 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 (with learning rate correspondingly greater). Of course, true problems do not have exact duplicates of individual patterns; nevertheless, true 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 in some problems should be preferred, as we shall see in Sect. **??**.

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 Chap. **??**, we could train as long 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. This example shows that the general phenomenon should be called "overfitting," and 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, stopping the training before gradient descent is complete can help avoid overfitting. In practice, the elementary criterion of stopping when the error function decreases less than some preset value (e.g., line **??** in Algorithm **??**), does not lead reliably to accurate classifiers as it is hard to know beforehand what an appropriate threshold θ should be set. A far more effective method is to stop training when the error on a separate validation set reaches a minimum (Fig. **??**). We shall explore the theory underlying this version of cross validation in Chap. **??**. We note in passing that weight decay is equivalent to a form of stopped training (Fig. 6.22).

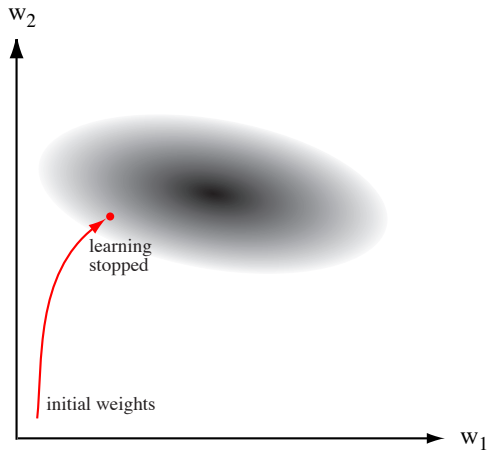


Figure 6.22: When weights are initialized with small magnitudes, stopped training is equivalent to a form of weight decay since the final weights are smaller than they would be after extensive training.

6.8.15 How many 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 transfer functions. Since, as we have seen, three layers suffice to implement any arbitrary function, we

would need special problem conditions or requirements 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 three-layer net to accept *small* translations than to accept large ones. In practice, then, networks with several hidden layers distribute the invariance task throughout the net. Naturally, the weight initialization, learning rate, data preprocessing arguments apply to these networks too. The Neocognitron network architecture (Sec. 6.10.7) has many layers for just this reason (though it is trained by a method somewhat different than backpropagation). It has been found empirically that networks with multiple hidden layers are more prone to getting caught in undesirable local minima.

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

6.8.16 Criterion function

The squared error criterion of Eq. 8 is the most common training criterion because it is simple to compute, non-negative, and simplifies the proofs of some theorems. Nevertheless, other training criteria occasionally have benefits. One popular alternate is the cross entropy which for n patterns is of the form:

$$J(\mathbf{w})_{ce} = \sum_{m=1}^{n}\sum_{k=1}^{c} t_{mk}\ln(t_{mk}/z_{mk}), \tag{40}$$

where t_{mk} and z_{mk} are the target and the actual output of unit k for pattern m. Of course, this criterion function requires both the teaching and the output values in the range $(0, 1)$.

Regularization and overfitting avoidance is generally achieved by penalizing complexity of models or networks (Chap. **??**). In regularization, the training error and the complexity penalty should be of related functional forms. Thus if the pattern error is the sum of squares, then a reasonable network penalty would be squared length of the total weight vector (Eq. 38). Likewise, if the model penalty is some description length (measured in bits), then a pattern error based on cross entropy would be appropriate (Eq. 40).

MINKOWSKI ERROR

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

$$J_{Mink}(\mathbf{w}) = \sum_{m=1}^{n}\sum_{k=1}^{c} |z_{mk}(\mathbf{x}) - t_{mk}(\mathbf{x})|^R, \tag{41}$$

much as we saw in Chap. **??**. It is a straightforward matter to derive the backpropagation rule for the this error (Problem **??**). While in general the rule is a bit more complex than for the ($R = 2$) sum squared error we have considered (since it includes a $Sgn[\cdot]$ function), 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 with others. 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. One can use second-order information more fully in other ways.

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. 16 & 20. We now turn to second-order derivatives, which find use in rapid learning methods, as well as some pruning or regularization algorithms. For our criterion function,

$$J(\mathbf{w}) = \frac{1}{2}\sum_{m=1}^{n}(t_m - z_m)^2, \tag{42}$$

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

$$\frac{\partial^2 J(\mathbf{w})}{\partial w_{ji}\partial w_{lk}} = \sum_{m=1}^{n}\frac{\partial J}{\partial w_{ji}}\frac{\partial J}{\partial w_{lk}} + \sum_{m=1}^{n}(z-t)\frac{\partial^2 J}{\partial w_{ji}\partial w_{lk}} \tag{43}$$

where we have used the subscripts to refer to *any* weight in the network — thus 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. 43 is often neglected as ; this approximation guarantees that the resulting approximation is positive definite.

The second term is of order $O(\|\mathbf{t} - \mathbf{o}\|)$; using Fisher's method of scoring we set this term to zero. This gives the expected value, a positive definite matrix thereby guaranteeing that gradient descent will progress. In this so-called *Levenberg-Marquardt* or *outer product approximation* our Hessian reduces to:

LEVENBERG-MARQUARDT APPROXIMATION

The full exact calculation of the Hessian matrix for a three-layer network such as we have considered is (Problem 31):

If the two weights are both in the hidden-to-output layer:

$$outout \tag{44}$$

If the two weights are both in the input-to-hidden layer:

$$inin \tag{45}$$

If the weights are in different layers:

$$inout \tag{46}$$

6.9.2 Newton's method

xxx

$$\begin{aligned}\boldsymbol{\Delta} J(\mathbf{w}) &= J(\mathbf{w}+\boldsymbol{\Delta}\mathbf{w}) - J(\mathbf{w}) \\ &\simeq \left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}\right)^t \boldsymbol{\Delta}\mathbf{w} + \frac{1}{2}\boldsymbol{\Delta}\mathbf{w}^t\mathbf{H}\boldsymbol{\Delta}\mathbf{w}, \end{aligned} \tag{47}$$

where $\mathbf{H}$ is the Hessian matrix. We differentiate Eq. 47 with respect to $\boldsymbol{\Delta}$ and find that $\boldsymbol{\Delta} J(\mathbf{w})$ is minimized for

$$\left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}\right) + \mathbf{H}\boldsymbol{\Delta}\mathbf{w} = \mathbf{0}, \tag{48}$$

and thus the optimum change in weights can be expressed as

$$\boldsymbol{\Delta}\mathbf{w} = -\mathbf{H}^{-1}\left(\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}\right). \tag{49}$$

Thus, if we have an estimate for the optimal weights $\mathbf{w}(m)$, we can get an improved estimate using the weight change given by Eq. 49, i.e.,

$$\mathbf{w}(m+1) = \mathbf{w}(m) + \boldsymbol{\Delta}\mathbf{w} = \mathbf{w}(m) - \mathbf{H}^{-1}(m)\left(\frac{\partial J(\mathbf{w}(m))}{\partial \mathbf{w}}\right), \tag{50}$$

Thus in this Newton's algorithm, we iteratively recompute $\mathbf{w}$.

Alas, the computation of the Hessian can be expensive, and there is no guarantee that the Hessian is nonsingular.

xxx

6.9.3 Quickprop

The simplest method 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 (i.e., a parabola) and the coefficients for the 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. 6.23). It can be shown (Problem 34) that this approach leads to the following weight update rule:

$$\Delta w(m+1) = \frac{\frac{dJ}{dw}\big|_m}{\frac{dJ}{dw}\big|_{m-1} - \frac{dJ}{dw}\big|_m}\Delta w(m). \tag{51}$$

If the third- and higher-order terms in the error are non-negligible, 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 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.

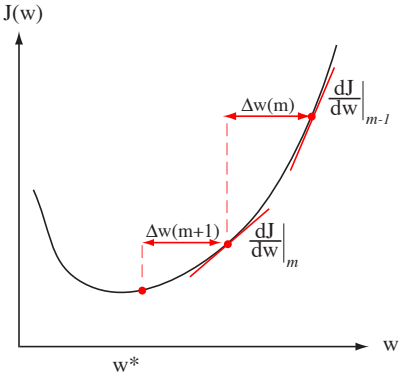


Figure 6.23: The quickprop weight update takes the error derivatives at two points separated by a known amount, and by Eq. 51 makes its next weight value. If the error can be fully expressed as a second-order function, then the weight update leads to the weight (w^*) leading to minimum error.

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, determined by the gradient) and moves along that direction until the 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. **??**).

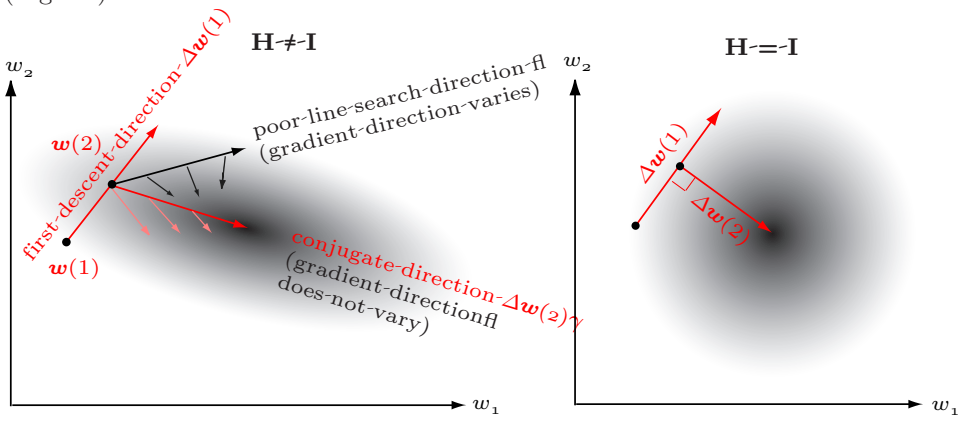


Figure 6.24: Conjugate gradient descent in weight space employs a sequence of line searches. If $\Delta\mathbf{w}(1)$ is the first descent direction, the second direction obeys $\Delta\mathbf{w}^t(1)\mathbf{H}\Delta\mathbf{w}(2) = 0$. Note especially that along this second descent, the gradient changes 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.

More specifically, if 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 is determined by the line search). We demand that the subsequent direction, $\Delta\mathbf{w}(m)$, obey

$$\Delta\mathbf{w}^t(m-1)\mathbf{H}\Delta\mathbf{w}(m) = 0, \tag{52}$$

where $\mathbf{H}$ is the Hessian matrix. Pairs of descent directions that obey Eq. 52 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, since 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:

$$\Delta\mathbf{w}(m) = -\nabla J(\mathbf{w}(m)) + \beta_m\Delta\mathbf{w}(m-1), \tag{53}$$

and the relative proportions of these contributions is governed by β. This proportion can be derived by insuring that the descent direction on iteration m does not spoil that from direction $m-1$, and indeed all earlier directions. It is generally calculated in one of two ways. The first formula (Fletcher-Reeves) is

$$\beta_m = \frac{[\nabla J(\mathbf{w}(m))]^t\ \nabla J(\mathbf{w}(m))}{[\nabla J(\mathbf{w}(m-1))]^t\ \nabla J(\mathbf{w}(m-1))} \tag{54}$$

A slightly preferable formula (Polak-Ribiere) is more robust in non-quadratic error functions is:

$$\beta_m = \frac{[\nabla J(\mathbf{w}(m))]^t\ [\nabla J(\mathbf{w}(m)) - \nabla J(\mathbf{w}(m-1))]}{[\nabla J(\mathbf{w}(m-1))]^t\ \nabla J(\mathbf{w}(m-1))}. \tag{55}$$

Equations 53 & 36 show that conjugate gradient descent algorithm is analogous to calculating a "smart" momentum, where β 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 miminimum of a simple quadratic criterion function centered on the origin of weight space, $J(\mathbf{w}) = 1/2(.2w_1^2 + w_2^2) = \mathbf{w}^t\mathbf{H}\mathbf{w}$, where by simple differentiation the Hessian is found to be $\mathbf{H} = \left(\begin{smallmatrix}.2 & 0\\ 0 & 1\end{smallmatrix}\right)$. We start descent descent at a randomly selected position, which happens to be $\mathbf{w}(0) = \left(\begin{smallmatrix}-8\\ -4\end{smallmatrix}\right)$, as shown in the figure. The first descent direction is determined by a simple gradient, which is easily found to be $-\Delta J(\mathbf{w}(0)) = -\left(\begin{smallmatrix}4w_1(0)\\ 2w_2(0)\end{smallmatrix}\right) = \left(\begin{smallmatrix}3.2\\ 8\end{smallmatrix}\right)$. 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 be calculus. We let s represent the distance along the first descent direction, and 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

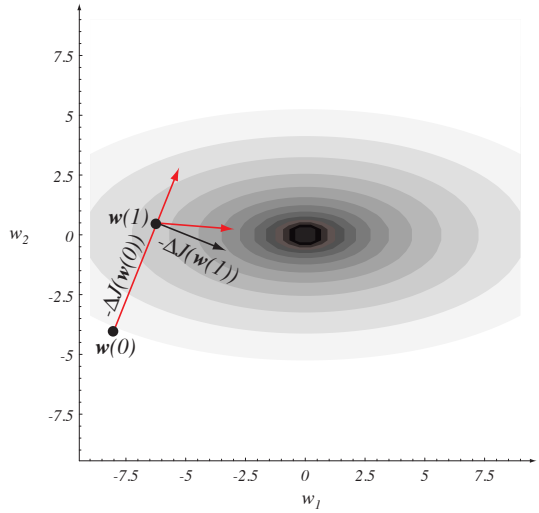
$$-\Delta J(\mathbf{w}(1)) = -\begin{pmatrix}.4w_1(1)\\2w_2(1)\end{pmatrix} = \begin{pmatrix}2.48\\-0.99\end{pmatrix}.$$

(It is easy to verify that this direction, shown as a black arrow in the figure, does not point toward the global minimum at $\mathbf{w} = \binom{0}{0}$.) We use the Fletcher-Reeves formula (Eq. 54) to construct the conjugate gradient direction:

$$\beta_1 = \frac{[\Delta J(\mathbf{w}(1))]^t \Delta J(\mathbf{w}(1))}{[\Delta J(\mathbf{w}(0))]^t \Delta J(\mathbf{w}(0))} = \frac{(-2.48\ .99)\binom{-2.48}{.99}}{(-3.2\ 8)\binom{-3.2}{8}} = \frac{7.13}{74} = 0.096.$$

Incidentally, for this quadratic error surface, the Polak-Ribiere formula (Eq. 55) would give the same value. Thus the conjugate descent direction is

$$\Delta\mathbf{w}(1) = -\Delta J(\mathbf{w}(1)) + \beta_1\begin{pmatrix}3.2\\8\end{pmatrix} = \begin{pmatrix}2.788\\-.223\end{pmatrix}.$$



Conjugate gradient descent in a quadratic error landscape, shown in contour 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 properly passes through the global error minimum at $\mathbf{w} = \binom{0}{0}$.

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

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

which has solution $s = 2.231$. This yields the next minimum to be

$$\mathbf{w}(2) = \mathbf{w}(1) + s\Delta\mathbf{w}(1) = \begin{pmatrix}-6.202\\0.496\end{pmatrix} + 2.231\begin{pmatrix}2.788\\-.223\end{pmatrix} = \begin{pmatrix}0\\0\end{pmatrix}.$$

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

The elementary method of gradient descent used by backpropagation can be slow, even with straightforward improvements. We now consider some alternate networks and training methods.

6.10.1 Radial basis function networks (RBF)

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 unit implements

$$z_k(\mathbf{x}) = \sum_{j=0}^{n_H} w_{kj}\phi_j(\mathbf{x}). \tag{56}$$

where we have included a $j = 0$ bias unit. If we define a vector $\boldsymbol{\phi}$ whose components are the hidden unit outputs, and a matrix $\mathbf{W}$ whose entries are the hidden-to-output weights, then Eq. 56 can be rewritten as: $\mathbf{z}(\mathbf{x}) = \mathbf{W}\boldsymbol{\phi}$. Minimizing the criterion function

$$J(\mathbf{w}) = \frac{1}{2}\sum_{m=1}^{n}(\mathbf{y}(\mathbf{x}^m;\,\mathbf{w}) - \mathbf{t}^m)^2 \tag{57}$$

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

$$\boldsymbol{\Phi}^t\boldsymbol{\Phi}\mathbf{W}^t = \boldsymbol{\Phi}^t\mathbf{T}, \tag{58}$$

and the solution can be written directly: $\mathbf{W}^t = \boldsymbol{\Phi}^\dagger\mathbf{T}$. Recall that $\boldsymbol{\Phi}^\dagger$ is the pseudoinverse 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

$$z_k(\mathbf{x}) = f\left(\sum_{j=0}^{n_H} w_{kj}\phi_j(\mathbf{x})\right) \tag{59}$$

rather than Eq. 56, then standard backpropagation can be used. One need merely take derivatives of the localized transfer 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 at the hidden units functions is absent, but this presents no conceptual difficulties (Problem **??**).

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 transfer 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 Chap. **??**. For instance, if we know that each underlying distribution comes from a mixture of two Gaussians, naturally we would use Gaussian transfer functions and use a learning rule that set the parameters (such as the mean and covariance).

6.10.3 Time delay neural networks (TDNN)

One can also 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 6.25 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 inputs from the input layer that are similarly shifted. Training proceeds as in standard backpropagation, but with the added constraint that corresponding weights are forced to have the same value — an example of *weight sharing*. Thus, the weights learned do not depend upon the position of the pattern (so long as the full pattern lies in the domain of the input layer). WEIGHT SHARING

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. 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.

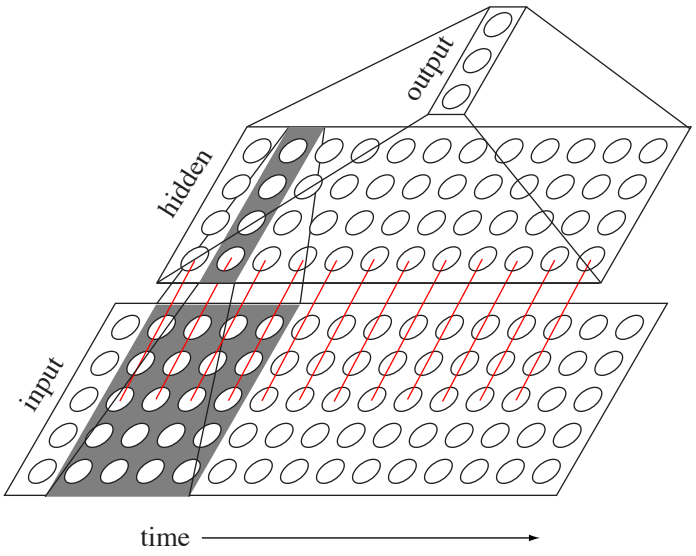


Figure 6.25: A time delay neural network (TDNN) uses weight sharing to insure that patterns are recognized regardless of shift in one dimension; in practice, this dimension generally corresponds to time. In this example, there are five input units at each time step. Because we hypothesize that the input 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.

6.10.4 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 6.26 illustrates such an architecture, one in which the output unit values are fed back and duplicated as auxiliary inputs, augmenting the traditional feature values. During classification, a static pattern $\mathbf{x}$ is presented to the input units, the feedforward flow computed, and the outputs 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 right of the figure, where it must be emphasized that many sets of weights are constrained to be the same (weight sharing), as indicated.

This unfolded representation shows that recurrent networks can be trained via standard backpropagation, but with the weight sharing constraint imposed, as in TDNNs.

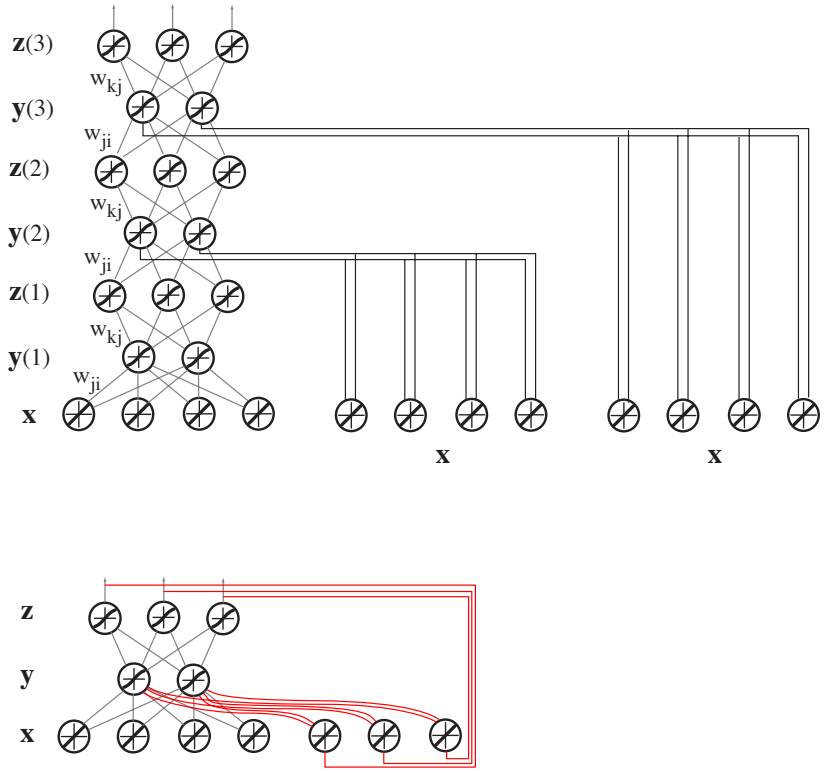


Figure 6.26: The form of recurrent network most useful for static classification has the architecture shown at the bottom, with the recurrent connections in red. It is functionally equivalent to a static network with many hidden layers and extensive weight sharing, as shown above. Note that the input is replicated.

6.10.5 Counterpropagation

Occasionally, one wants a rapid prototype of a network, yet one that has expressive power greater than a mere two-layer network. Figure 6.27 shows a three-layer net, which consists of familiar input, hidden and output layers.* When one is learning the weights for a pattern in category ω_i,

In this way, the hidden units create a Voronoi tesselation (cf. Chap. **??**), and the hidden-to-output weights pool information from such centers of Voronoi cells. The processing at the hidden units is competitive learning (Chap. **??**).

The speedup in counterpropagation is that only the weights from the single most active hidden unit are adjusted during a pattern presentation. While this can yield suboptimal recognition accuracy, counterpropagation can be orders of magnitude faster than full backpropagation. As such, it can be useful during preliminary data exploration. Finally, the learned weights often provide an excellent starting point for refinement by subsequent full training via backpropagation.

* It is called "counterpropagation" for an earlier implementation that employed five layers with signals that passed bottom-up as well as top-down.

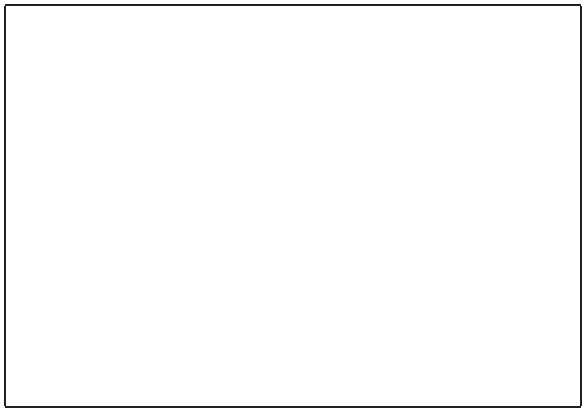


Figure 6.27: The simplest version of a counterpropagation network consists of three layers. During training, an input is presented and the most active hidden unit is determined. The only weights that are modified are the input-to-hidden weights leading to this most active hidden unit and the single hidden-to-output weight leading to the proper category. Weights can be trained using an LMS criterion.

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 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. Further, 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 via 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. 6.28).

The benefit is that often faster than strict backprop since fewer weights are updated at any time (Computer exercise 18).

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
```

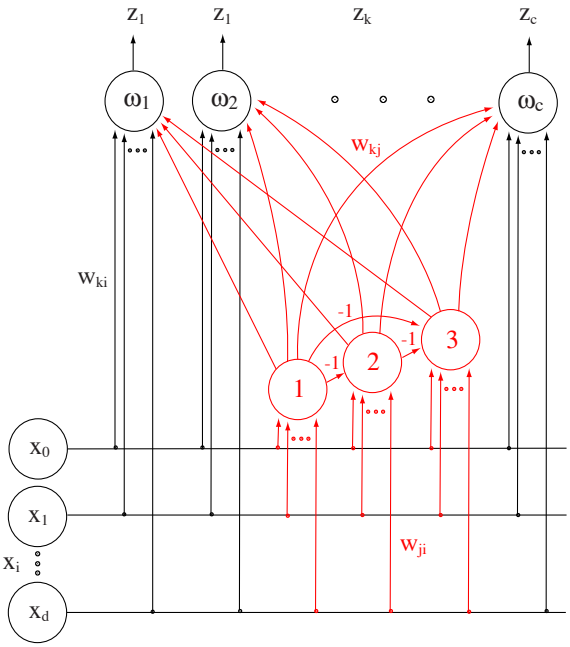


Figure 6.28: The training of a multilayer network via cascade-correlation begins with the input later fully connected to the output layer (black). Such weights, w_{ki} are trained using an LMS criterion, as discussed in Chap. **??**. If the resulting training error is not sufficiently low, a first hidden unit (labeled 1, in red) is introduced, fully interconnected from the input layer and to the output layer. These new red weights are 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.

6.10.7 Neocognitron

The cognitron and its descendent, the Neocognitron, address the problem of recognition of characters in pixel input. The networks are noteworthy not for the learning method, but instead for their reliance on a large number of layers for translation, scale and rotation invariance.

The first layer consists of hand tuned feature detectors, such as vertical, horizontal and diagonal line detectors. Subsequent layers consist of slightly more complex features, such as Ts or Xx, and so forth — weighted groupings of the outputs of units at earlier layers. The total number of weights in such a network is enormous (Problem 35).

6.11 Regularization and complexity adjustment

Whereas the number of inputs and outputs of a backpropagation network are determined by the problem itself, we do not know *a priori* the number of hidden units, or weights. If we have too many degrees of freedom, we will have overfitting. This will depend upon the number of training patterns and the complexity of the problem

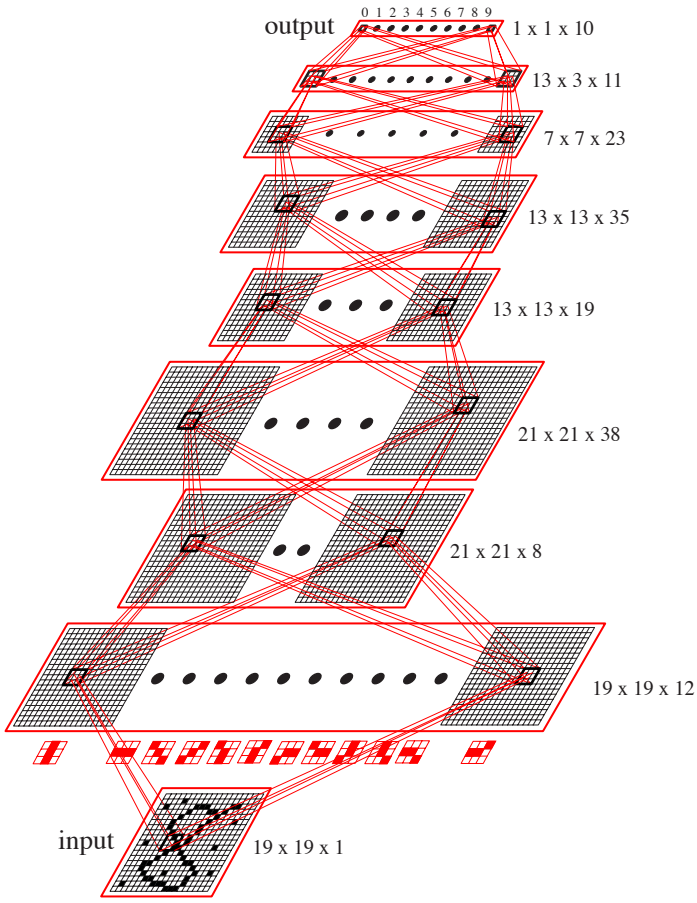


Figure 6.29: The neocognitron consists of a 19×19 pixel input layer, seven intermediate layers, and an output layer consisting of 10 units, one for each digit. The earlier layers consist of relatively fixed feature detectors (as shown); units in successively layer respond to a spatial range of units in the previous layer. In this way, shift, rotation and scale invariance is distributed throughout the network. The network is trained one-layer at a time by a large number of patterns.

itself.

We could try different numbers of hidden units, apply knowledge of the problem domain or add other constraints. The error is the sum of an error over patterns (such as we have used before) plus a regularization term, which expresses constraints or desirable properties of solutions:

$$J = J_{pat} + \lambda J_{reg}. \tag{60}$$

The parameter λ is adjusted to impose the regularization more or less strongly.

Because a desirable constraint is *simpler* networks (i.e., simpler models), regularization is often used to adjust complexity, as in weight decay.

6.11.1 Complexity measurement

xxx

6.11.2 Wald statistics

The fundamental theory of generalization favors simplicity. For a given level of performance on observed data, models with fewer parameters can be expected to perform better on test data. For instance weight decay leads to simpler decision boundaries (closer to linear). Likewise, training via cascade-correlation adds weights only as needed.

The fundamental idea in Wald statistics is that we can estimate the importance of a parameter in a model, such as a weight, by how much the training error increases if that parameter is eliminated. To this end the Optimal Brain Damage method (*OBD*) seeks to delete weights by keeping the training error as small as possible. *OBS* extended *OBD* to include the off-diagonal terms in the network's Hessian, which were shown to be significant and important for pruning in classical and benchmark problems.

OBD and Optimal Brain Surgeon (*OBS*) 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:

$$\delta J = \underbrace{\left(\frac{\partial J}{\partial \mathbf{w}}\right)^T \cdot \delta\mathbf{w}}_{\simeq 0} + \frac{1}{2}\,\delta\mathbf{w}^T \cdot \underbrace{\frac{\partial^2 J}{\partial \mathbf{w}^2}}_{\equiv \mathbf{H}} \cdot \delta\mathbf{w} + \underbrace{O(\|\delta\mathbf{w}\|^3)}_{\simeq 0}\,, \tag{61}$$

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 **??**):

$$\delta\mathbf{w} = -\frac{w_q}{[\mathbf{H}^{-1}]_{qq}}\,\mathbf{H}^{-1}\cdot\mathbf{u}_q \quad \text{and} \quad L_q = \frac{1}{2}\,\frac{w_q^2}{[\mathbf{H}^{-1}]_{qq}}\,. \tag{62}$$

Here, $\mathbf{u}_q$ is the unit vector along the qth direction in weight space and L_q is the *saliency* of weight q — an estimate of the increase in training error if weight q is pruned and the other weights updated by the left equation in Eq. 62 (Problem 42).

We define $\mathbf{X}_k \equiv \frac{\partial g(\mathbf{x}^m;\,\mathbf{w})}{\partial \mathbf{w}}$ and $a_k \equiv \frac{\partial^2 d(\mathbf{t}^m,\mathbf{z}^m)}{\partial \mathbf{z}^2}$, and can easily show that the recursion for computing the inverse Hessian becomes:

$$\begin{aligned}
\mathbf{H}_{m+1}^{-1} &= \mathbf{H}_m^{-1} - \frac{\mathbf{H}_m^{-1}\cdot\mathbf{X}_{m+1}\cdot\mathbf{X}_{m+1}^T\cdot\mathbf{H}_m^{-1}}{\frac{P}{a_k} + \mathbf{X}_{m+1}^T\cdot\mathbf{H}_m^{-1}\cdot\mathbf{X}_{m+1}}, \\
\mathbf{H}_0^{-1} &= \alpha^{-1}\mathbf{I} \qquad (63)\\
\mathbf{H}_n^{-1} &= \mathbf{H}^{-1}\,, \qquad (64)
\end{aligned}$$

where α is a small parameter — effectively a weight decay constant (Problem 38). Note how different error measures $d(\mathbf{t},\mathbf{z})$ scale the gradient vectors $\mathbf{X}_k$ forming the Hessian (Eq. **??**). For the squared error $d(\mathbf{t},\mathbf{z}) = (\mathbf{t}-\mathbf{z})^2$, we have $a_k = 1$, and all gradient vectors are weighted equally.

Problem: repeat for cross-entropy (Problem 36).

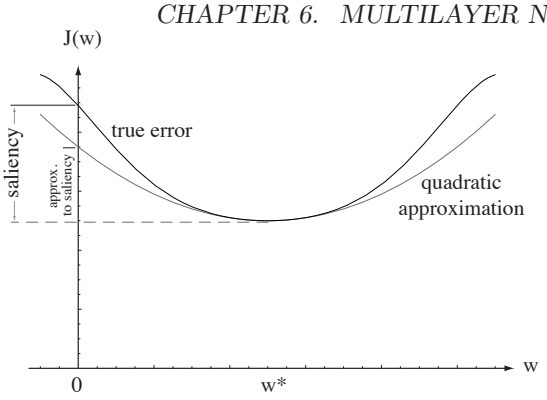


Figure 6.30: The saliency of a parameter, such as a weight, is the increase in the training error when that weight is set to zero. One can approximate the saliency by expanding the true error around a local minimum, w^*, and setting the weight to zero. In this example the approximated saliency is smaller than the true saliency; this is typically, but not always the case.

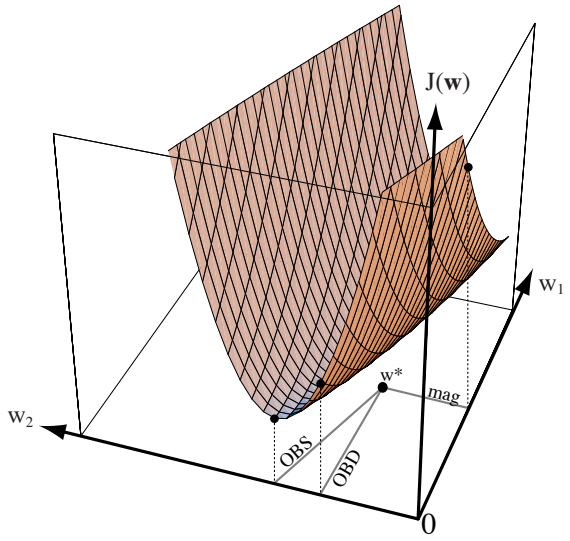


Figure 6.31: In the second-order approximation to the criterion function, optimal brain damage assumes the Hessian matrix is diagonal, while Optimal Brain Surgeon uses the full Hessian matrix.

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 (parameters) in the model defined by the network topology. 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 constraints by means such as weight decay. Discrete pruning algorithms such as Optimal Brain Surgeon and Optimal Brain Damage correspond to priors favoring *few* weights, and can help avoid overfitting.

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

Complexity adjustment: weight decay, Wald statistic, which for networks is optimal brain damage and optimal brain surgeon, which use the second-order approximation to the true saliency as a pruning criterion.