Contents

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.

In this chapter 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 learning algo-

* 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*.

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

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 and 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 for 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 optimized 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 data used to train the net.

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 in question, the 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.

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 by choices in the number of hidden layers, units, feedback connections, and so on. Thus setting the topology of the network is a sort of 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.

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

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

HIDDEN LAYER

BIAS

NEURON

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.

RECALL

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

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

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

SYNAPSE

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

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 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.

TRANSFER FUNCTION

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

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

* 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.

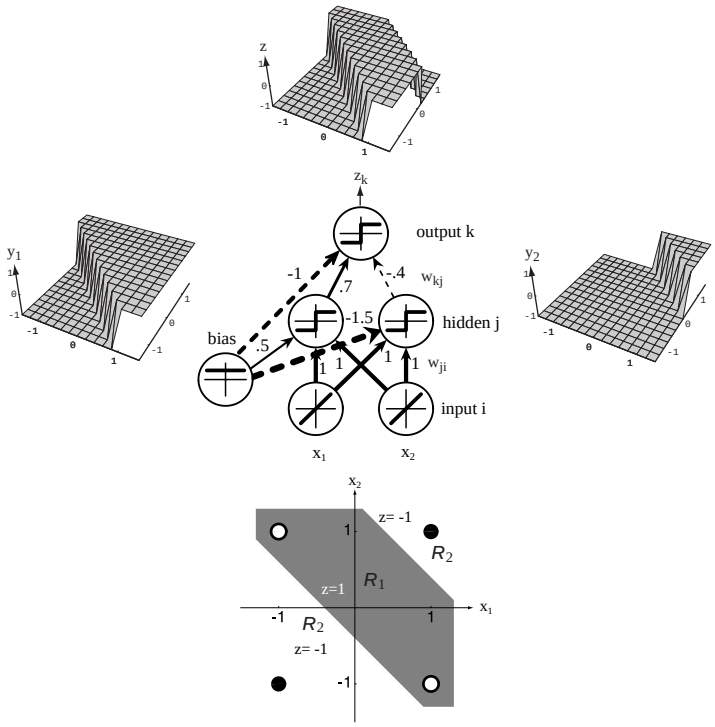

Figure 6.1: The two-bit parity or exclusive-OR problem and a three-layer network that solves it. At the bottom is the two-dimensional feature space $x_1 - x_2$, and the four patterns to be classified. In the middle a three-layer network. The input units are linear and merely distribute their (feature) 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 0, and -1 otherwise, as shown by the graphs. Positive ("excitatory") weights are denoted by by solid lines, negative ("inhibitory") weights by dashed lines; the weight magnitude is indicated by the relative thickness and is labelled. The single output unit sums the weighted signals from the hidden units (and bias) and emits a +1 if that sum is greater than 0 and a -1 otherwise. Within each unit we show a graph of its input-output 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.

where the subscript k indexes units in the output layer 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. 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 of 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 of the output unit.

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 $x_1 + x_2 + 0.5 = 0$; likewise hidden unit y_2 forms $x_1 + x_2 - 1.5 = 0$. The final output unit emits a $+1$ if and only if *both* y_1 and y_2 have value $+1$. This leads 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 our categories, and the signal from the output unit is the discriminant function $g(\mathbf{x})$. 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 different transfer functions at the hidden layer than the output layer, or indeed even different functions at each 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 two-category classification case, one can merely apply a threshold to the output of the single output unit and thereby obtain any decision region.

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 proof does not limit us. 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 (indeed cannot be smooth). As we shall soon see, though, smoothness is benefitial for 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 have smooth sigmoidal (rather than threshold) φ functions, and furthermore that such units are "paired" in opposition, so each pair implements a "bump" (or generalized "ridge" in higher dimensional space) mapping from input to output (Fig. 6.2). A sufficient number of these bumps could be put together to implement a portion of a sine wave of an arbitrarily large range of inputs. Another set of hidden units could implement different sine wave of a different frequency. Proper hidden-to-output weights corresponding to the coefficients in a Fourier synthesis would then enable the full network to implement the desired function.

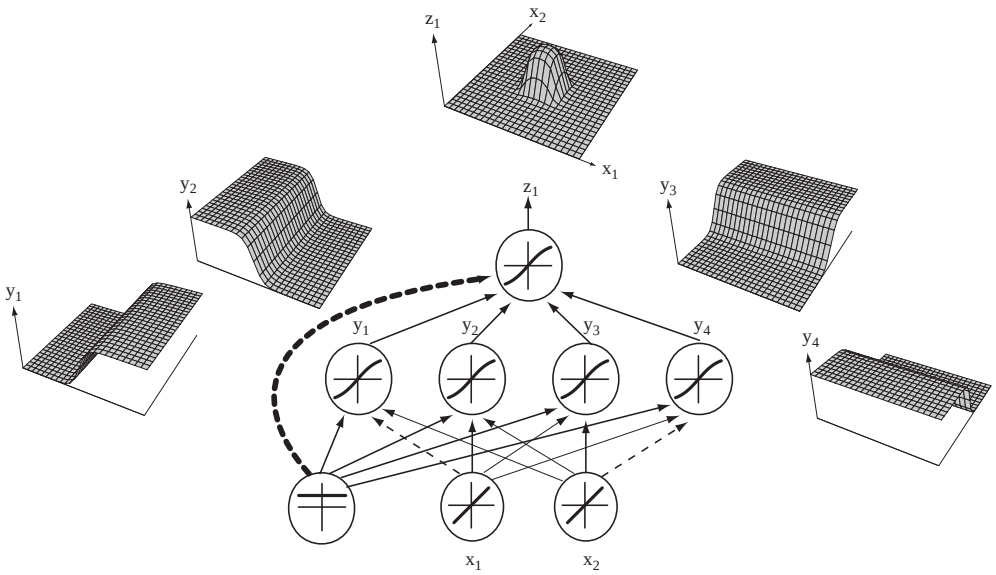

Figure 6.2: A 2-4-1 network along with the response functions at different units; each hidden and output unit has sigmoidal transfer function $f(\cdot)$. In the case shown, the hidden unit outputs are paired in opposition to yield a "bump" at the output unit, yielding $g(\mathbf{x})$. Given a sufficiently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

In a related analysis, one conceptually pairs sigmoids together to make ridges, and two of these to make bumps and constructs any well-behaved function by suffi-

cient number of such bumps without first constructing harmonic functions explicitly (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 pair sigmoids to get component bumps.

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 (such as sigmoids), can represent any continuous function too so long as the component sigmoids can be shifted and scaled. Thus we generally use a single functional form for the nonlinearities.

While these latter proof methods show that any desired function can be implemented by a three-layer network, they are not particularly practical because for most problems we do not know ahead of time the number of hidden units required, or 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.

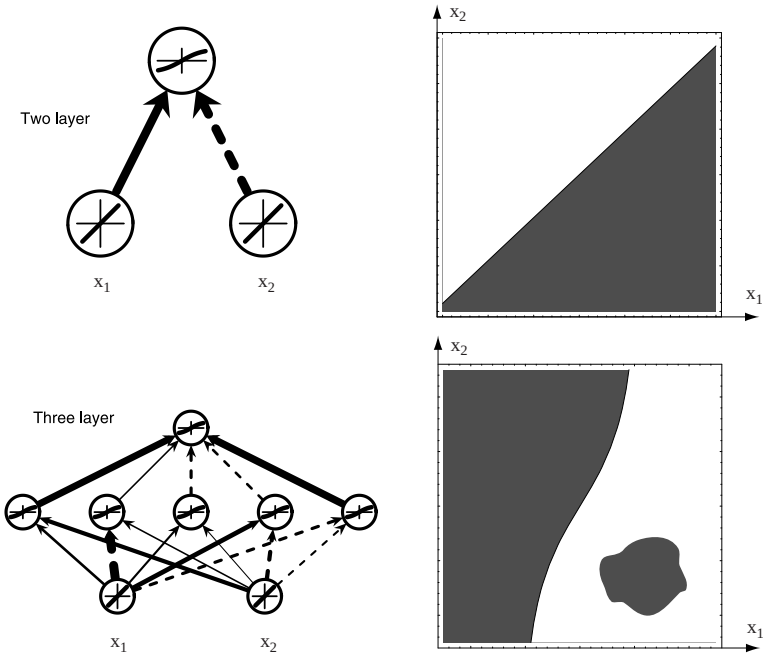

Figure 6.3: Whereas a two-layer network can only implement a linear decision boundary, given an adequate number of hidden units, three-, four- and higher-layer networks can implement arbitrary decision boundaries. The decision regions need not be convex, nor simply connected.

In sum, the above shows that given lax conditions, any function from input to output can be implemented by a three-layer feedforward network. Thus arbitrary discriminant functions can be implemented, and thus any decision boundaries, even in a c-category problem.

6.3 Backpropagation algorithm

We have just seen that any function from input to output can be implemented as a three-layer neural network. However, the above are not a constructive proofs, that is, they do not tell us how to set the weights to implement a particular desired function. Likewise, we have not shown how the weights in Fig. 6.1 were chosen (or learned). 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 training such 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 upon a 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? 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 activation should be. This is called the *credit assignment* problem. The power of backpropagation and related methods is that it allows us to calculate an effective error for each hidden unit, and this leads to 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. 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 **??** 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 (for instance, having weights whose values are assigned randomly), present an input training pattern and determine the output. The error or criterion function is some function of the actual and the outputs, and 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 methods later.

One common and useful measure of error on a pattern is the sum over output units of the squared difference between the desired output t_k (given by a teacher) and the actual ouput z_k — the *training error*. One of the most useful criterion functions is the sum over all the output units (indexed by k) of the square of the difference between the actual output and the desired or teaching value, 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 vectors of length c at the output units and $\mathbf{w}$ represents all the weights in the network.

The learning rule is based on gradient descent. The weights will start with a random value, 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

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

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 with a weight vector at iteration t and updating it as:

LEARNING RATE

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

where t 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. We first consider 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:

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

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. Differentiating in Eq. 8 shows 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. **??**:

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

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

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

The learning rule for the input-to-hidden units is a bit 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

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

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. Identifying Eq. 13 with Eq. 18, we can define the sensitivity for a hidden unit as:

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

Equation 19 is the core of the solution to the credit assigment problem: the sensitivity at a hidden unit is simply related to the sum of the individual sensitivities at the output units weighted by the hidden-to-output weights w_{jk}. Thus the learning rule for the input-to-hidden weights is:

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

Equations 16 & 20, together with training protocols such as described below, is called the backpropagation algorithm, or more specifically the "backpropagation of errors" algorithm, because during training an "error" (actually, the sensivities 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. 6.4).

These learning rules make intuitive sense. Consider first the rule for learning weights at the output units (Eq. 16). The weight update 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 we shall use most often, $f'(net_k)$ is always positive, and thus if $(t_k - z_k) > 0$ (for positive y_j), 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 indeed 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_{ij} will not change the performance on the pattern presented. A similar analysis of Eq. 20 yields insight of the input-to-hidden weights (Problem 5).

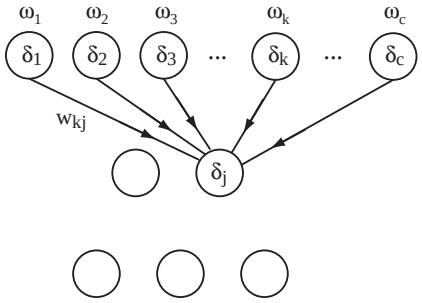

Figure 6.4: The sensitivity at a hidden unit is the weighted sum of the sensitivities at the output units: $\delta_j = f'(net_j) \sum_{k=1}^{c} w_{kj}\delta_k$.

Problem 6 asks you to show that the presence of the bias unit does not materially affect the above results. 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
- there are different nonlinearities at each unit
- each unit has a different learning rate.

It is a more subtle matter to incorporate 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, training consists in presenting patterns whose category label we know — the *training set* — to the network and adjusting the weights so as to reduce error. The three most useful training protocols are: stochastic, batch and on-line. In *stochastic training*, patterns are chosen randomly from the training set, and the network weights are updated for each pattern presentation. This method is called stochastic because the training data represent a random variable. In *batch training*, all patterns are presented to the network before learning (a weight update) takes place. Here too, 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 TRAINING

A forth protocol is *learning with queries* where the output of the network is used to *select* or sample new training patterns. Such queries generally focus on points that are likely to give the most information to the classifier, for instance samples near

LEARNING WITH QUERIES

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

category decision boundaries (Chap. **??**). The drawback with this protocol is that the training samples are no longer independent, identically distributed (i.i.d.), being skewed instead toward sample boundaries.

EPOCH

We describe the overall amount of pattern presentations by *epoch* — the number of presentations of the full training data. 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 are shown in the procedures below.

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 better performance, as we shall discuss below.

STOPPING CRITERION

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.

Algorithm 2 (Batch backpropagation)

```
1  begin initialize  network topology (# hidden units), w, criterion θ, η, r = 0
2    do r ← r + 1
3        m ← 0; Δw_ij ← 0; Δw_jk ← 0
4        do m ← m + 1
5            x^m ← select pattern
6            Δw_ij ← Δw_ij + ηδ_j x_i;   Δw_jk ← Δw_jk + ηδ_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 method in Sect. 6.8.

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:

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

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

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.

6.3.3 Learning curves

Because the weight are initialized randomly, error on the training set is large; through learning the error becomes lower, as shown in a *learning curve* (Fig. 6.5). The (per pattern) training error decreases through learning, ultimately reaching 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. **??**).

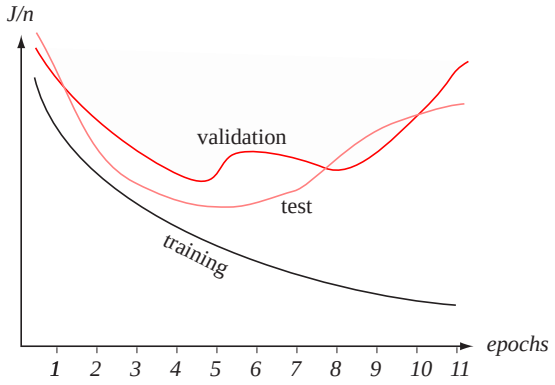

Figure 6.5: A learning curve shows the criterion function as a function of the amount of training, typically indicated by the number of epochs, or presentations of the full training set. It is traditional to plot the average error per pattern, i.e., $1/n \sum_{p=1}^{n} J_p$. The validation error and the test (or generalization) error per pattern are virtually always higher than the training error. In some training protocols, a set of patterns is used as a validation set, and training is stopped at the minimum of this set.

Since bach backpropagation performs gradient descent in the criterion function, these training error decreases monotonically. The average error on an independent test set is virtually alway higher than on the training set, and whil it geterally decreases, it can increase or oscillate.

VALIDATION ERROR

Figure 6.5 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 for forming 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 that figure. We shall return in Chap. **??** to understand in greater depth why this method of *cross validation* often leads to 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 such learning by studying error surfaces themselves — the function $J(\mathbf{w})$. Of course, such an error surface depends upon the data at hand; 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 concern 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 $\mathbf{w} = 0$ 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. 6.6. The data shown are linearly separable, and the optimal decision boundary (a point near $x = 0$) separates the two categories. During learning, the weights descends to the global minimum, and the problem is solved.

Notice that 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.

Now consider the same network applied to another, harder, one-dimensional problem — one that is not linearly separable (Fig. 6.7). First, note that overall the error surface is higher than in Fig. 6.6 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 exacly two patterns, but these have different errors. Note too that there are multiple error minima; these correspond to the solutions that place either four or two of the ω_1 points in one category. Incidentally, a 1-3-1 network (but not a 1-2-1 network) can solve this problem (Computer exercise 2).

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, that there are plateaus corresponding to sets of weights that lead to nearly the same error, and finally how the existence of local minima can correspond to different partial or approximate solutions. Furthermore, the surface near $\mathbf{w} \simeq 0$ (the traditional region for starting learning) has high error and happens in this case to have a large slope; here if the starting point had been different, the network would descend to the same final weight values.

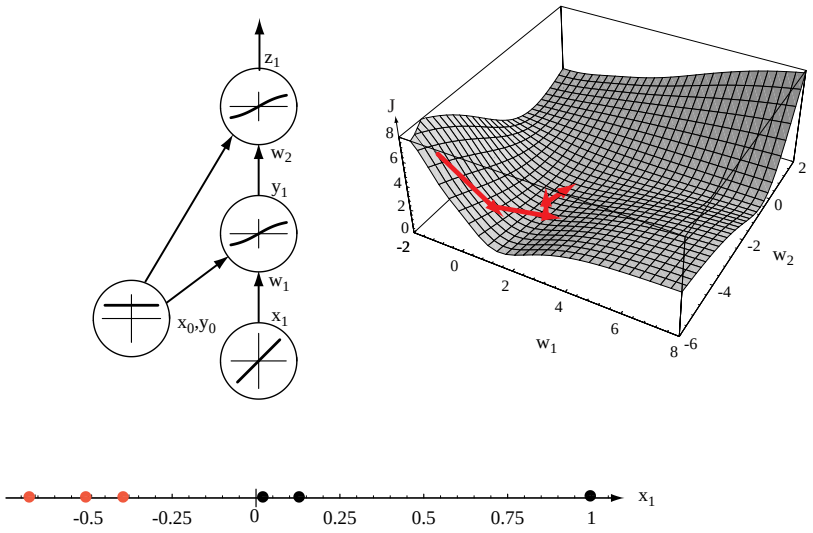

Figure 6.6: One-dimensional input x_1 and six patterns (three in each of two classes) to be learned. The 1-1-1 network with sigmoidal hidden and output units (and bias) is shown at the left. The error surface as a function of w_1 and w_2 is also shown (for the case where the bias weights have their final 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.

6.4.2 XOR

A somewhat more complicated problem is the XOR problem we have already considered. Figure 6.8 shows several two-dimensional slices through the nine-dimensional weight space of the 2-2-1 sigmoidal network (with bias) having the topology in Fig. 6.1. The slices shown include a global minimum in the error.

First of all, notice 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. 6.6 & 6.7. This is because on average in a large network any single weight has a smaller relative contribution to the error. 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

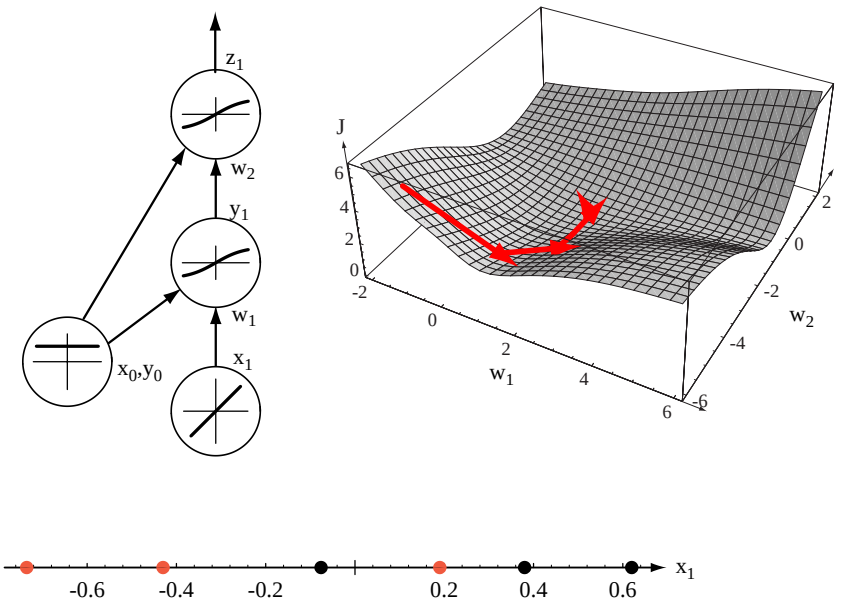

Figure 6.7: As in Fig. 6.6, except here the patterns are not linearly separable; the error surface is slightly higher than in that figure.

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 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 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 overfitting, as we shall see in Sect. 6.11.

6.4.4 How important are multiple minima?

Training error surfaces can have multiple minima, and depending upon initial conditions, gradient descent will take the system into different minima having different final errors. Generally speaking, we would prefer the gloabal minimum — the lowest test error given our network topology. In practice, if the asymptotic training error is not sufficiently low and we feel the net is caught in a non-global minimum, we can re-initialize the weights with different random values and train again.

In many problems, a local minimum is acceptable. Furthermore, when training with cross-validation stopping criterion, the training error never reaches the global minimum anyway.

6.5 Backpropagation as feature mapping

Since the hidden-to-output layer merely implements a linear discriminant, the novel computational power provided by multilayer neural nets can be attributed to the

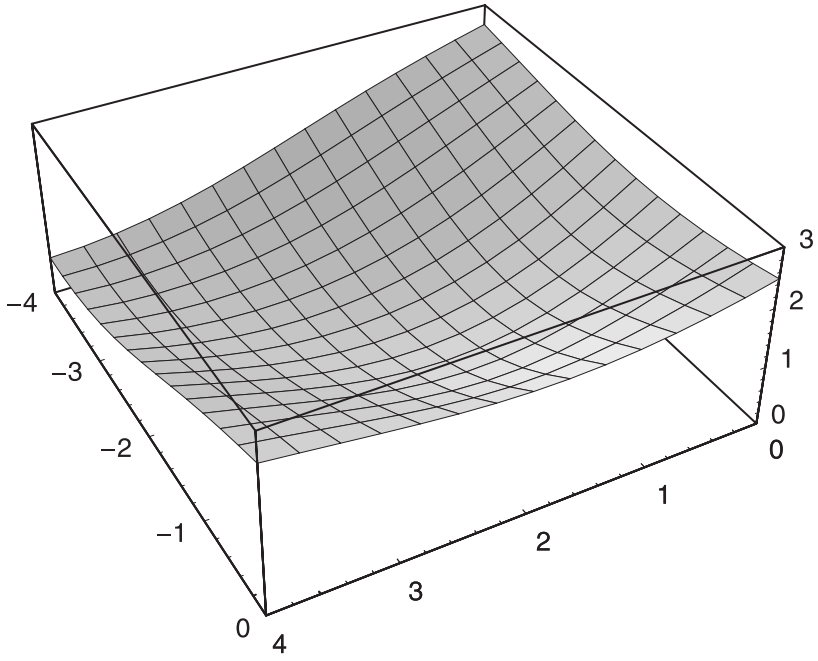

Figure 6.8: Two-dimensional slices through the nine-dimensional error surface after extensive training for a 2-2-1 network solving the XOR problem. a) $w_{??} - w_{??}$, b) $w_{??} - w_{??}$, c) $w_{??} - w_{??}$.

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.

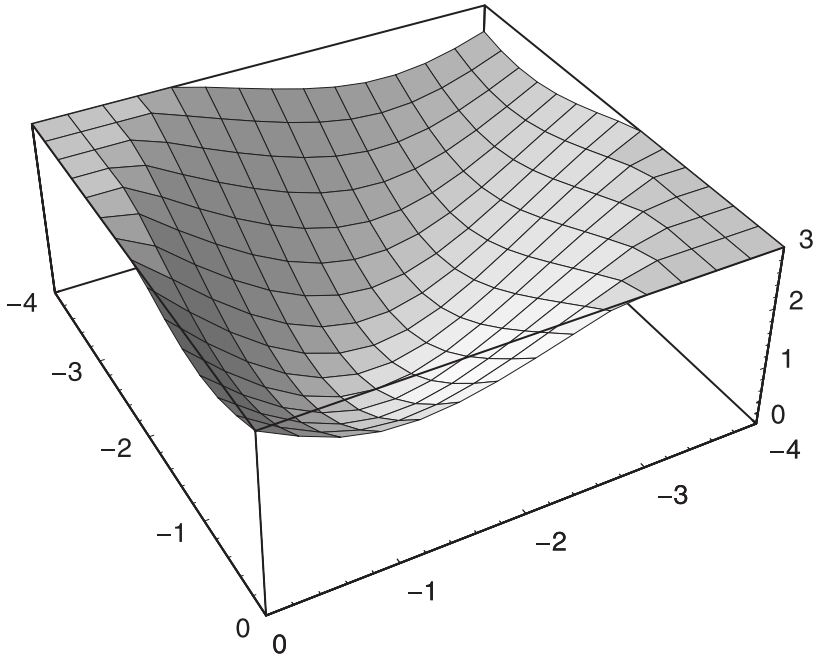

Figure 6.9: a) Error as a function of two input-to-hidden weights in a ??-??-?? network solving a categorization problem. b) As in a), but for two hidden-to-ouput weights. Because of the smaller numbers of such weights, error is more strongly dependent upon hidden-to-output weights than upon input-to-hidden weights. c) Error as a function of an input-to-hidden and a hidden-to-output weights involving different hidden units. In general, the dependence of the error shows negligible cross-terms in such weights. d) As in c), but for weights involving the same hidden unit. The error depends upon the two weights in a non-independent manner.

Figure 6.11 shows a three-layer backpropagation net for solving the XOR problem. The input space is denoted $x_1 - x_2$, while the outputs of the two hidden units represent an $y_1 - y_2$ space. At any stage in learning, each of the four input patterns has a particular representation in $y_1 - y_2$ 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 (cf., 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 figure shows the hidden representation of the four patterns as learning progresses. As just mentioned, before learning the patterns are not linearly separable; after learning they are. The linear decision boundary at the end of learning found by the hidden-to-output weights is shown by the straight dashed line. In the case shown in Fig. 6.11 the nonlinearly separable problem at the inputs is transformed into a linearly separable by the hidden units.

We can repeat the above analysis but for 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

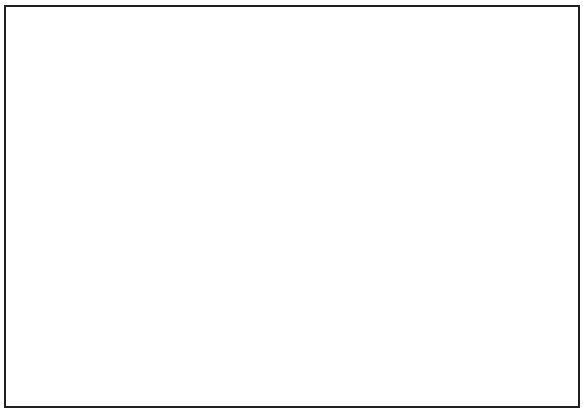

Figure 6.10: a) – c) Slices through the error surface ??-??-?? network solving a hard problem in pattern classification.

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, though, the nonlinearities of the hidden units are expressed and patterns have been moved to be linearly separable by a plane learned by the hidden-to-output weights.

6.5.1 Example

The nonlinear transformation of pattern space for typical pattern recognition problems is more complicated. 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 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.

6.5.2 Hidden unit representations — weights

A method complementary to the representation of patterns is the representation of learned weights themselves. Since the hidden-to-output weights merely implement a linear discriminant, it is the input-to-hidden weights that are most instructive. In particular, the input-to-hidden weights at a single hidden unit describe a sort of "matched filter," i.e., the input pattern that leads to maximum activation of that hidden unit. Because the hidden unit transfer functions are nonlinear, the correspondence with classical methods such as principal components is not exact (Sect. **??.??**); nevertheless it is often convenient to think of the hidden units as finding relevant features for the hidden-to-ouput perceptron.

MATCHED FILTER

Figure 6.14 shows the input-to-hidden weights (displayed as patterns) for a simple task of character recognition. Note that one of the hidden units seems "tuned" for loops, while another for "bars" — both useful building blocks for the patterns presented.

Despite the fact that the hidden-to-output layer acts as a standard Perceptron, it is generally much harder to represent weights in terms of input features. This is because not only do the hidden units themselves already encode a somewhat abstract

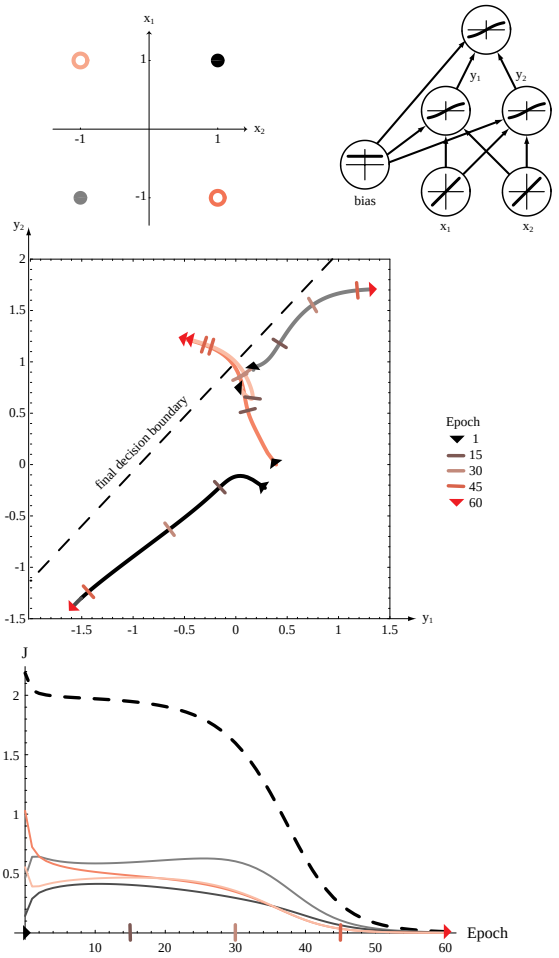

Figure 6.11: A 2-2-1 backpropagation network (plus bias) and the four patterns of the XOR problem are shown at the top. The middle figure shows the outputs of the hidden units (in $y_1 - y_2$ space) for each of the four patterns; these outputs change as the full network learns. Notice that at the beginning of training, the two categories are not linearly separable in the $y_1 - y_2$ space; however as the input-to-hidden weights learn, the categories become separable in the hidden unit space. Also shown 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 indeed separated by this boundary. The bottom graph shows 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.

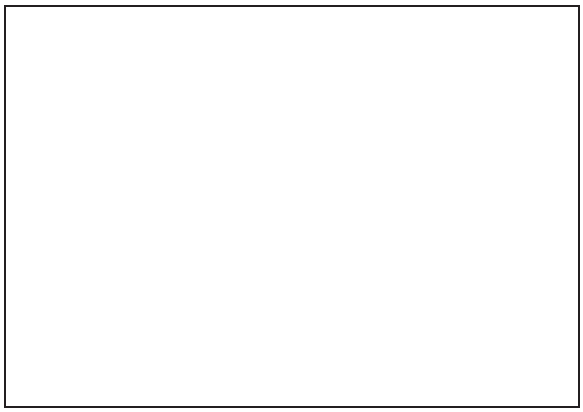

Figure 6.12: a) A 3-3-1 backpropagation network (plus bias) for solving the three-bit parity problem. b) The representation at the hidden units ($y_1 - y_2 - y_3$ space) as the system learns and the (planar) decision boundary found by the hidden-to-output weights at the end of learning. The patterns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

pattern, but also there is no natural representation of the hidden units since their ordering and placement is arbitrary. This, along with the fact that the output of hidden units are nonlinearly related to the inputs makes analyzing hidden-to-output weights somewhat problematic. Of the best we can do is describe the set of hidden units taht contribute significantly to a particular discriminant function.

6.6 Backpropagation and Bayes theory

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

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 in two ways: to multiple categories and to nonlinear functions implemented by three-layer neural networks.

We use the network of Fig. **??** 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 the Bayes decision criterion,

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

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

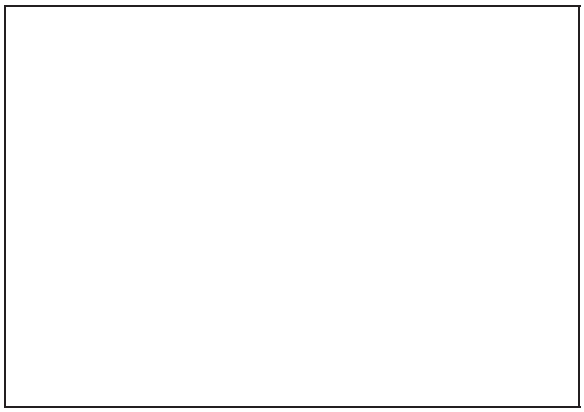

Figure 6.13: A two-dimesional two-category problem and the solutions learned by two networks. a) A 2-2-1 network (with bias) distorts the representation so that at the hidden units the categories are almost linearly separable. The straight line in $y_1 - y_2$ space represents the (linear) discriminant learned by the hidden-to-output weights. b) A 2-3-1 network (with bias) distorts the same patterns, but because of the higher dimension of the hidden layer representation, *can* separate the patterns of the two categories, and perfect categorization of the training data.

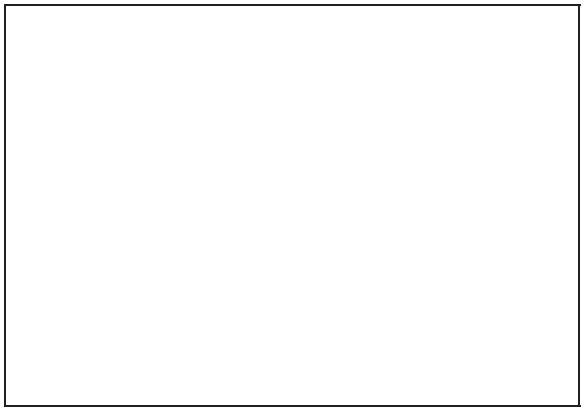

Figure 6.14: Hidden unit representations (input-to-hidden weights) for a large character recognition problem. Note especially that the weights describe features that are useful in classifying the patterns.

Suppose we train a network having c output units (one for each category) with a target signal according to:

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

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

$$\begin{aligned} J(\mathbf{w}) &= \sum_{\mathbf{x}} \left[g_k(\mathbf{x},\mathbf{w}) - t_k\right]^2 \qquad (24)\\ &= \sum_{\mathbf{x}\in\omega_k} \left[g_k(\mathbf{x},\mathbf{w}) - 1\right]^2 + \sum_{\mathbf{x}\notin\omega_k} \left[g_k(\mathbf{x},\mathbf{w}) - 0\right]^2 \\ &= n\left\{\frac{n_k}{n}\frac{1}{n_k}\sum_{\mathbf{x}\in\omega_k}\left[g_k(\mathbf{x},\mathbf{w}) - 1\right]^2 + \frac{n-n_k}{n}\frac{1}{n-n_k}\sum_{\mathbf{x}\notin\omega_k}\left[g_k(\mathbf{x},\mathbf{w}) - 0\right]^2\right\}, \end{aligned}$$

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 16):

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

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 21):

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

Thus the outputs of our trained network will, in the limit of infinite data, approximate (in a least-squares sense) the true a posteriori probabilities. In other words, if we train our network using the signals implied by Eq. 23, then the output units represent the a posteriori probabilities, i.e.,

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

Figure 6.15 illustrates the development of the learned discriminant functions toward the Bayes discriminants as the amount of training data 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 decision *boundaries* are found, just as we saw in Chap. **??**.

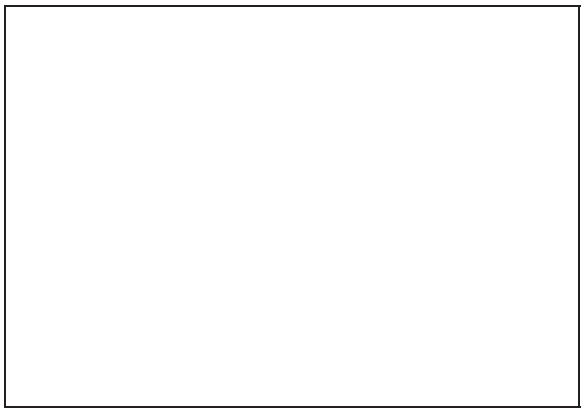

Figure 6.15: As a backpropagation network learns (under the assumptions given in the text), its outputs more closely approximate posterior probability densities. The figure shows the output of a 1-3-1 sigmoidal backpropagation network after training. Note especially the excellent agreement with the discriminant functions in the regions of high $p(x)$; the agreement is worse at the extreme values of x, where $p(x)$ is small. A 1-8-1 sigmoidal network has greater expressive power, and its outputs approximate the posterior densities quite well.

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, including training with 0–1 target values. While indeed given infinite amounts of training data (and assuming the net can express the discriminants, does not fall into a 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.

Thus if the network outputs do not sum to 1.0 for some range of the input space, it is an indication that the network is not accurately modelling the psteriors. This, in turn, may suggest changing the network topology, number of hidden units, or other aspects of the net, as described in Sect. 6.8.

One heuristic approach toward obtaining approximations to probabilities is to choose the output unit nonlinearity to be exponential — $f(net_k) \propto e^{net_k}$ — and for any pattern normalize the outputs to sum to 1.0, i.e.,

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

SOFTMAX

WINNER-TAKE-ALL

and to train using 0–1 signals. This is called *softmax* — a sort of smoothed or continuous version of a *winner-take-all* function, where 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 19, Computer exercise 8).

A neural network classifier trained in this manner approximates the 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 ration of such priors (Computer exercise 9).

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

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

Here each $\mathbf{u}_j$ and u_{j0} together define the projection of the input $\mathbf{x}$ onto j_{max} different d-dimensional hyperplanes. These projections are transformed by nonlinear functions $f_j(\cdot)$, which are then linearly combined at the output. Sigmoidal function are traditional, but other smooth function are occassionally used instead. The $f_j(\cdot)$ have been called *ridge functions* because for peaked $f_j(\cdot)$ one obtains ridges in two dimensions. Equation 30 would be used for implementating a mapping to a scalar function z and c-category classification problem there would be c such outputs. The parameters are set using an LMS criterion. In computational practice, the parameters are learned in groups, for instance first the components of $\mathbf{u}_1$ and u_{10}, then $\mathbf{u}_2$ and u_{20}; then the w_j and w_0, iterating until convergence.

RIDGE FUNCTION

Such models are related to the three-layer networs we have seen in that the $\mathbf{u}_j$ and u_{j0} are analogous to the input-to-hidden weights at a hidden unit. The class of functions $f_j(\cdot)$ at such hidden units are more general and have more free parameters than do sigmoids. The effective output unit is linear. Moreover, such a model can have an output much larger than 1.0, as m9ight be needed in a general regression task; in constrast, the outputs of a typical sigmoidal output unit saturate.

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 **??**), 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. Here the output is the weighted sum of M products of splines:

MULTIVARIATE ADAPTIVE REGRESSION SPLINE

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

where the kth basis function is the product of r_k one-dimensional spline functions ϕ_{kr}, and w_0 is a scalar offset. The splines depend on the input values x_q, such as the feature component of an input, and that feature index is the label $q(k, r)$.

In broad overview, training in MARS begins by fitting each feature dimension in turn with a spline function. 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 are considered, one by one. For each such dimension, candidate splines are selected based on the fit of the product of the spline and the one already selected, i.e., the product $r = 1 \rightarrow 2$. The best such spline is retained, thereby giving the $r = 2$ term. In thes way, splines are added incrementally up to come value r_k until a desired level of fit is achieved.

6.8 Practical techniques for improving backpropagation

When designing and training a multilayer neural network, the designer must make a number of decisions. The two major types of decision are the topology and the parameter selection; here we discuss parameter selection and return to topology in Sect. 6.12. Our goal here is to give a principled basis for making such choices in order to speed learning and give optimal recognition performance. In practice such parameter adjustment is problem dependent; nevertheless there are a number of useful choices based on a deeper understanding of these networks with only the barest assumptions as to the particular problems themselves.

6.8.1 Transfer function

We first turn to the choice of nonlinearity $f(\cdot)$. 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 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. 6.12.5).

There are several functional forms that have been used for the transfer function $f(\cdot)$, often motivated by knowledge of the problem domain. Barring such problem dependent design, what properties do 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 1.0.) A third property is continuity, 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 threshold or sign function of Eq. 3 could not be used in backpropagation.) Backpropagation can be made to work with *piecewise* linear transfer functions, but with added complexity and few benefits.

Monotonicity is another desirable property for $f(\cdot)$ — the derivative should 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 **??**). One can use functions that are non-monotonic, as in radial basis functions (Sect. 6.12.5) if proper care is taken. Another desirable property is linearity for small value of *net*. This will enable the system to implement a linear model if the linear model is adequate for the task. A property that is might occassionally be of importance is computational simplicity — we seek a function whose value and derivative can be easily computed.

One class of function that has all these properties is the *sigmoid*. The sigmoid is SIGMOID smooth, differentiable, nonlinear, and saturating. The sigmoid also has the benefit of allowing a linear model; if the weights are small, the linear region of the activation function is used. Another 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 24).

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

6.8.2 Parameters for the sigmoid

Given that we will use the sigmoidal form, there remain a number of parameters to choose. 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. As we shall see in Sect. **??**, anti-symmetric sigmoids speed learning by avoiding large covariances in network activations. 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 the learning rate, size of the input, etc., that determine learning times (Problem 22). For convenience, though, we choose $a = 1.716$ and $b = 2/3$ in Eq. 33; this insures $f'(0) \simeq 1$ and gives extrema of the second derivative at roughly $net \simeq \pm 2$ (Fig. 6.16).

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 networks: 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" unit — indeed the error will hardly depend upon the tiny length values at all. If however, the physically same data were presented but with mass measured in kilograms and length in millimeters, the situation would be reversed. Since we do not want our classifier to prefer one of these features over the other since they differ solely in the arbitrary representation. How shall we insure that the input features are properly scaled?*

In order to avoid such difficulties, the input patterns should be scaled so that the average (over the training set) of each feature is zero and equal variance — here chosen to be 1.0, for reasons that will be clear below. That is, we *standardize* the STANDARDIZE training patterns. Another benefit is that standardization of the inputs may help to symmetrize the Hessian, thereby speeding learning (Sect. 6.8.14).

* The difficulty arises even for features having the same units, but differing overall magnitude, of course, as in fish length fin thickness measured in millimeters.

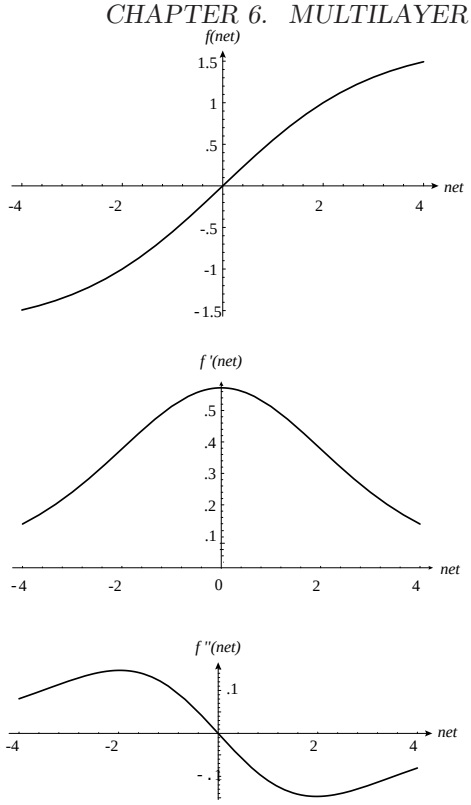

Figure 6.16: A useful transfer function $f(net)$ is a sigmoid, centered on zero and anti-symmetric. The transfer function is nearly linear in the range $-1 < net < +1$ and the second derivative, $f''(net)$, has extrema near $net = \pm 2$.

6.8.4 Target values

For pattern recognition, we typically seek a one-of-c representation for the c output units. Since the output units saturate at ± 1.716, we might naively feel that the target values should be those values. However, this presents a difficulty: weights would become extremely large as the activation for an output unit would be driven to $\pm \infty$. Instead, teaching values of +1 for the target category and -1 for the non-target categories allows rapid learning. For instance, if the pattern is in the category ω_3, the following teaching values (presented to the output units) 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 probabilities.

This data standardization is done once, before actualy network training, and represts a small one-time computational burden. Technically speaking, standardization can only be done for stochastic and batch learning protocols, but not on-line protocols, since the full data set is never available at any one time.

6.8.5 Number of hidden units

While the number of input units is given by the dimensionaly of the input vectors, and the number of output units is given by the number of categories, the number of hidden units, n_H, is not simply related to such obvious properties of the classification problem at hand. The number of hidden units tgoverns the expressive power of the

net — the complexity of the decision boundary. If the patterns are well separated, or linearly separable, for instance, then few hiddens are needed. Conversely, if the patterns are highly interspersed, more hiddens may be needed.

Figure 6.17 shows the training and test error on a two-category classification problem for networks as a function of the number of hidden units. Indeed, for large n_H, the training error can become small. Nevertheless, in this region of overfitting, the test error is high.

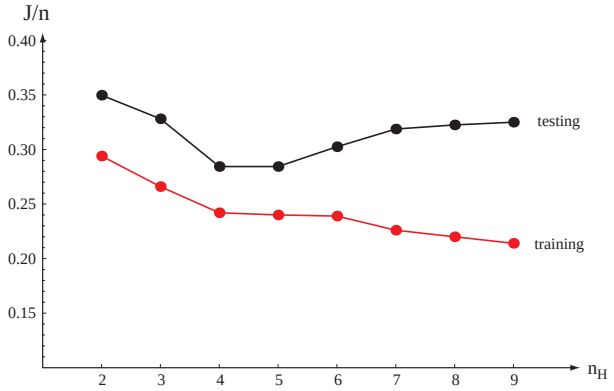

Figure 6.17: The average error on networks fully trained having different numbers of hidden units, n_H. This xxx–n_H–xxx was trained on a two-category problem with $n = xxx$ patterns.

The number of hidden units determines the total number of weights in the net — roughly the number of degrees of freedom — and in any case we should not have more weights than n, the total number of training points. A convenient rule of thumb is to choose the number of hidden units such that the total number of weights is roughly $n/10$. This seems to work well over a range of problems. A more principled method is to start with a "large" number of hiddens (i.e., such that the total number of weights is roughly $n/2$), and prune or eliminate weights — a technique we shall study in Sec. **??**.

6.8.6 Initializing weights

Suppose for the moment that we have fixed the network topology, and thus the number of hidden units. We now seek to set the initial weights in order to have fast and uniform learning. In this context, uniform learning means unbiased: if *part* of the network learns first (i.e., *some* of the weights reach equilibrium values), there will be a bias in the learning of the remaining weights. Biased learning arises, for instance, if category ω_i is learned well before ω_j, since the subsequent learning of ω_j will be affected.*

Consider a hidden unit having a fan-in of d inputs (we ignore the bias unit here). Suppose too that all weights have the same value w_0. On average, then, the net activation from d random variables of variance 1.0 from our standarized input through such weights will be $w_0\sqrt{d}$. We would like this net activation to be roughly in the range (-1, +1), so as to be near the cusp of the transfer function, i.e., neither too

* There should, of course, be no confusion between the statistical term "bias" as used here, and the term used elsewhere in this chapter to represent an offset.

small (linear model) nor too large (saturated). Thus we should choose our weights randomly and uniformly in the range $(-1/\sqrt{d}, +1/\sqrt{d})$. Such initialization also has the benefit of keeping the learning more uniform among weights, and thus less biased.

6.8.7 Hints

Often we have insufficient training data for adequate accuracy. We would like to add information or constraints to improve the network — technically called *bias* (Chap. **??**). Occassionally this information can be incorporated into the network topology (Sec. **??**), but more frequently it is in the form of related problems.

The approach of learning with *hints* is to add output units for an ancillary or related problem. We train the network simultaneously on the problem of interest *and* the ancillary one, by using a 2-out-of-n target representation.

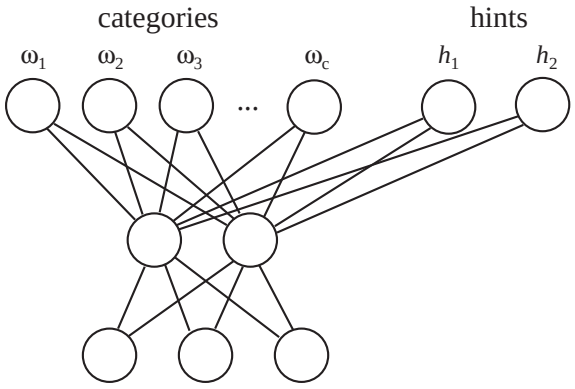

Figure 6.18: Network for training with hints.

For instance, suppose we seek to train a network to classify n phonemes based on some acoustic input. In standard methodology we would have n output units. In learning with hints, though, we might add two ancillary output units, one for vowels, one for consonants.

Learn other problems along the way. for instance, vowels

good feature selection

6.8.8 Training with noise

If insufficient data, manufacture data

add noise

If standardized data (Sec. **??**), indpendent Gaussian noise

Use standard deviation smaller than 1 (e.g., 0.1).

6.8.9 Learning rates

The ideal would be to adjust learning rates per weight to make them all complete learning at the same time, so no parameter reaches final value first. This, too, would help avoid biased learning. If possible, all weights should complete learning at the same time, though this can be somewhat difficult in practice since one does not know ahead of time how much training each will need. The optimal learning rate should

depend upon properties of the problem (Sect. **??**), and if the Hessian can be calculated, a good choice of η can be made. Without further analysis, a typical value of $\eta \simeq 0.1$ is a reasonable first choice.

In order to have the most rapid learning, we would like to set the learning constant η large. However, if η is too large, the learning will be unstable (Fig. 6.19).

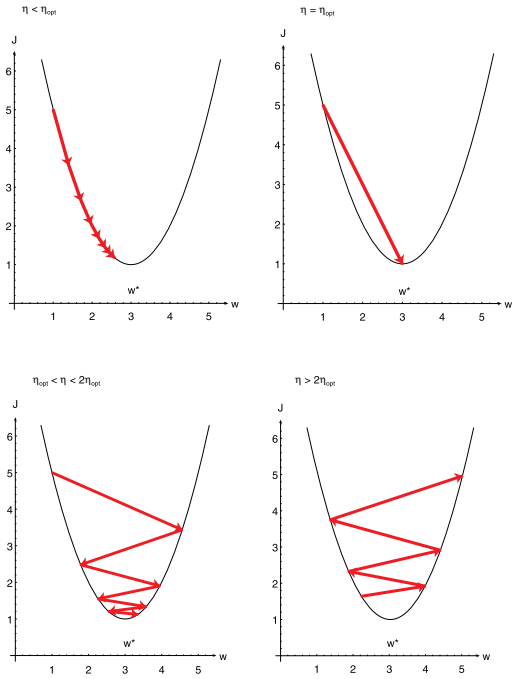

Figure 6.19: Gradient descent in one dimension with different learning rates. If $\eta < \eta_{opt}$, convergence is assured, but training can be needlessly slow. If $\eta = \eta_{opt}$, a single learning iteration suffices to find the error minimum. If $\eta_{opt} < \eta < 2\eta_{opt}$, the system converges, but will oscillate; convergence is needlessly slow. If $\eta > 2\eta_{\text{opt}}$, the system diverges.

The optimal learning rate is the one which leads to the local error minimum in one learning step. Assuming the error surface is quadratic we have (Fig. 6.20):

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

Solving for the optimal rate gives:

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

and of course $\eta_{\text{max}} = 2\eta_{\text{opt}}$. The system diverges if too large a learning rate is used. This derivation assumed the error surface is well described to second-order, and in practice higher order terms do not affect this rate significantly.

It should be emphasized that a learning rate greater than η_{opt} will lead to slower confergence (Computer exercise 7).

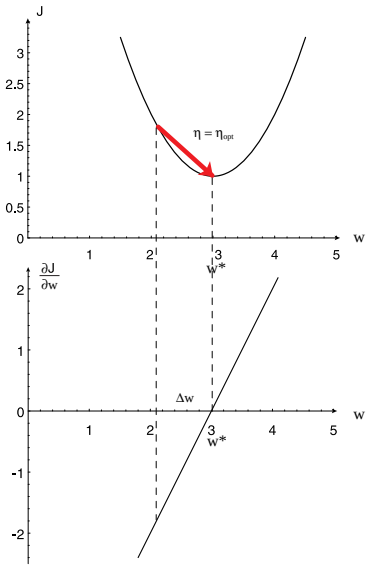

Figure 6.20: If the criterion function is quadratic (above), its derivative is linear (below). The optimal learning rate η_{opt} insures that the weight value yielding minimum error, $\mathbf{w}^*$ is found in a single learning step.

6.8.10 Momentum

In practice, error surfaces often have plateaus — regions in which the slope $dJ(\mathbf{w})/d\mathbf{w}$ is very small — for instance due to "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 backpropagation learning rule to include some fraction α of the previous weight update:

$$\mathbf{w}(t+1) = \underbrace{\mathbf{w}(t) + \Delta\mathbf{w}(t)}_{\substack{\text{standard} \\ \text{backprop}}} + \underbrace{\alpha\Delta\mathbf{w}(t-1)}_{\text{momentum}} \tag{36}$$

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" stochasic variations in weight updates during stochastic learning (Fig. 6.21).

Algorithm 3 (Stochastic backpropagation with momentum)

1 **begin** **initialize** topology (# hidden units), $\mathbf{w}$, criterion, $\alpha(<1), \theta, \eta, k = 0, m_{ij} = 0, m_{jk} = 0$
2 **do** $b \leftarrow b + 1$
3 $\mathbf{x}^m \leftarrow$ randomly chosen pattern
4 $m_{ij} \leftarrow \eta\delta_j x_i + \alpha m_{ij}$
5 $m_{jk} \leftarrow \eta\delta_k y_j + \alpha m_{jk}$
6 $w_{ij} \leftarrow w_{ij} + m_{ij}$
7 $w_{jk} \leftarrow w_{jk} + m_{jk}$
8 **until** $\nabla J(\mathbf{w}) < \theta$

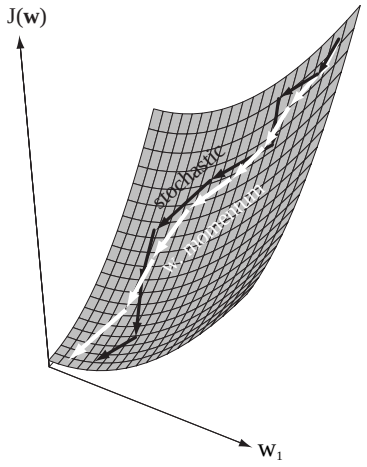

Figure 6.21: Stochastic gradient descent (black arrows) and with incorporating momentum (white arrows). Momentum tends to reduce random variation in the descent direction. It also speeds learning over plateaus in the error surface.

```
9  return w
10 end
```

6.8.11 Weight decay

One method of regularization a network is to impose a heuristic that the weights should be small. There is no principled reason why this method should always lead to improved network performance (indeed there are cases where it leads to *degraded* performance). It corresponds to a bias favoring models with small parameters (Problem 32).

One of the reasons weight decay is so popular is the simplicity of its implementation. One needs merely to change the learning rule to be:

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

that is, to "decay" the weights. Intuitively, we might expect weights that are not needed would become smaller and smaller, whereas those that *are* needed to solve the problem would decay, but if the error increases, then the weight decay would stop. Thus, the system would achieve a balance between pattern error (Eq. eq:regularizdef) and an overall weight. It can be shown (Problem 35) that the weight decay is equivalent lead to gradient descent of a new effective error or cost function:

$$J = J_{\text{pat}} + \underbrace{\frac{2\epsilon}{\eta}\sum_{(ij)} w_{ij}^2}_{\lambda J_{\text{reg}}}. \tag{38}$$

Another version of weight decay involves a decay parameter that depends upon the value of the weight itself:

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

All of these are to reduce the exess degrees of freedom, to avoid overfitting. They form a bias on the types of solutions that can result.

6.8.12 On-line, stochastic or batch training?

Each of the three basic 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.

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. Suppose for the batch case the learning rate is set as high as possible while assuring convergence and avoiding instabilities. 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 ten times 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 — and especially ones employing large redundant training sets — stochastic training is to be preferred. Batch training admits some second-order methods (e.g., some pruning techniques) not available with stochastic training, however (Sec. **??**).

6.8.13 Stopped training

Avoid overfitting

In two-layer training we could train all the way because it did not change the *complexity* of the classifier — only the parameters. It still implemented a linear discriminant. In multilayer networks the issue is more subtle. Excessive training leads to a more complex classifier.

Call it overfitting, not over training.

Even if one does not know the right number of weights, one can start with a network that could have too many degrees of freedom and limit their deleterious effects by "stopping" the training, i.e., stop training before the minimum training error occurs (Fig. 6.22). Stopped training is a heuristic that keeps weights smaller, so that units are operating in their linear region, thereby reducing the expressive power of the network. This is the process of *cross validation* is related to regularization mehtods discussed in Sect.6.11, and a topic we shall revisit in Sect. **??**.**??**.

CROSS VALIDATION

6.8.14 Convergence of gradient descent in high-dimensions

Let us describe the error locally to second order around a local minimum $\mathbf{w}^*$.

$$J(\mathbf{w}) \simeq J(\mathbf{w}) + 1/2(\mathbf{w} - \mathbf{w}^*)^t \mathbf{H}(\mathbf{w} - \mathbf{w}^*) + ..., \tag{40}$$

HESSIAN where the *Hessian* is the symmetric square matrix of second derivatives of the error:

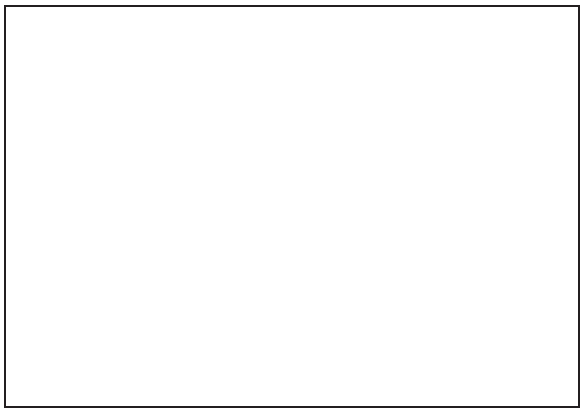

Figure 6.22: Typical form of learning curves in networks. The training error continues to decrease as a function of the amount of training (typically asymptoting at non-zero error). The validation error — the error on an independent set of samples *not* used for training — decreases, and then increases, indicating overtraining. (In fact, the validation error can oscillate many times as a function of training, but typically one stops training at the *first* minimum of the validation error.) Stopping training at the minimum of a validation error helps to reduce overfitting by keeping some weights small and hence the model simpler.

$$\mathbf{H}_{ij} \equiv \frac{\partial^2 J}{\partial w_m \, \partial w_r}. \tag{41}$$

We could be more careful and use $\mathbf{H}(\mathbf{w}^*)$ in the above, but our quadratic approximation assumes $\mathbf{H}$ is a constant in the region of interest, so we omit the argument. We note for completeness that the Hessian could refer to *any* parameters within the network or recognizer, we have written weights for clarity.

Gradient descent is thus:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \left.\frac{\partial J}{\partial \mathbf{w}}\right|_t = \mathbf{w}(t) - \eta \mathbf{H}(\mathbf{w} - \mathbf{w}^*), \tag{42}$$

where, as before, t denotes the iteration number. Rewriting Eq. 42, we get:

$$(\mathbf{w}(t+1) - \mathbf{w}^*) = \underbrace{(\mathbf{I} - \eta \mathbf{H})}_{\substack{\text{shrinks} \\ \text{any vector?}}} (\mathbf{w}(t) - \mathbf{w}^*), \tag{43}$$

where $\mathbf{I}$ is the identity matrix of the same rank as $\mathbf{H}$. Equation 43 shows that we will get convergence if the prefactor on the right hand side shrinks any vector.

An analysis of the structure of the Hessian can give us bounds on the convergence rate of gradient descent. We first rotate the Hessian to make it diagonal (though almost surely not proportional to the identity). Define a rotation matrix $\mathbf{\Theta}$ such that

$$\mathbf{\Theta}\mathbf{H}\mathbf{\Theta}^t = \mathbf{\Lambda}, \qquad \text{with } \mathbf{\Theta}^t\mathbf{\Theta} = \mathbf{I}. \tag{44}$$

The crucial point we take is that the more symmetric the Hessian — that is, the closer it is to being proportional to the identity matrix — the faster the convergence. We shall see how to aid in this by preprocessing data in Sect. **??**.

We can view gradient descent in r-dimensional weight space as r indpendent one-dimensional descents along the eigenvectors of the Hessian. Convergence is assured if $\eta < 2/\lambda_{max}$, where λ_{max} is the largest eigenvalue of $\mathbf{H}$ (Problem 15).

It can be shown (Problem 14) that convergence obtains if $\eta < 2/\lambda_{\max}$, where $\lambda_{\max}$ is the maximum eignevalue of the Hessian (Problem **??**).

6.8.15 Criterion function

We have concentrated almost exclusively on a (squared) error criterion (Eq. 8), though others occassionally have benefits. One popular alternate is the cross entropy which for n patterns is of the form:

$$J_{ce} = \sum_{i=1}^{n} \sum_{k=1}^{c} t_{ik} \log(t_{ik}/z_{ik}), \tag{45}$$

where t_{ik} and z_{ik} are the teaching and the actual output of unit k for pattern i. Of course, this criterion function requires both the teaching and the output values in the range $(0, 1)$ (Fig. 6.23).

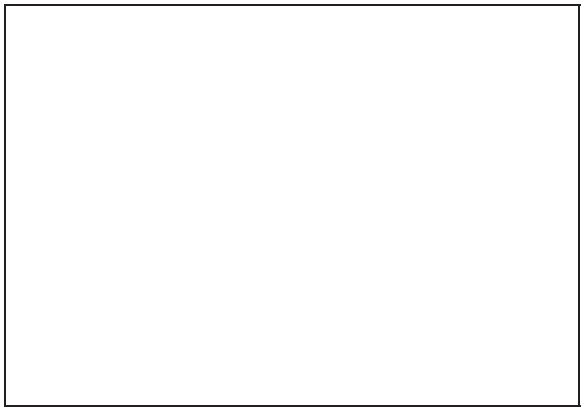

Figure 6.23: xxx

Benefits are

6.9 Transfer functions

DISTRIBUTED REPRESENTATION

sigmoid: *distributed representation*
localized: gaussian
distributed representation

LOCAL REPRESENTATION

6.9.1 Radial basis functions

xxx

6.9.2 Special bases

lksjdf

6.10 Additional training methods

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

6.10.1 Conjugate gradient descent

One method for speeding up the descent is conjugate gradient descent, which employs a series of line searches. 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 direction 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 itterations (Figure 6.24).

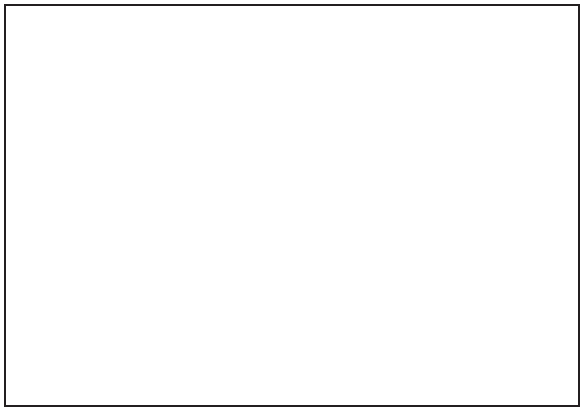

Figure 6.24: Conjugate gradient descent in m (weight) dimensions employs m line searches. Note especially that the descent does not "spoil" the descent from the previous line search.

The method is called "conjugate" because $\Delta\mathbf{w}^t(t)\mathbf{H}\Delta\mathbf{w}(t+1) = 0$, where now t indexes the successive line searches (Fig. 6.25). The descent directions are orthogonal in the space where the Hessian is proportional to the identity matrix.

If $\Delta\mathbf{w}(t)$ is the descent direction on iteration t, we have

$$\Delta\mathbf{w}(t) = -\nabla J(\mathbf{w}(t)) + \beta_\kappa \Delta\mathbf{w}(t-1). \tag{46}$$

There are two primary formulas for β. The first is

$$\beta_\kappa = \frac{[\nabla J(\mathbf{w}_\kappa)]^t \ \nabla J(\mathbf{w}_\kappa)}{[\nabla J(\mathbf{w}_\kappa) - \nabla J(\mathbf{w}_{\kappa-1})]^t \ \nabla J(\mathbf{w}_{\kappa-1})}. \tag{47}$$

A slightly preferable formula is more robust in non-quadratic (valid in the expansion)

$$\beta_\kappa = \frac{[\nabla J(\mathbf{w}_\kappa)]^t \ \nabla J(\mathbf{w}_\kappa)}{[\nabla J(\mathbf{w}_{\kappa-1})]^t \ \nabla J(\mathbf{w}_{\kappa-1})} \tag{48}$$

The use of conjugate gradient requires batch training.

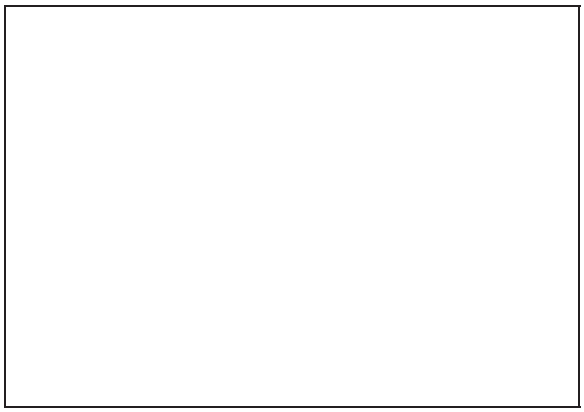

Figure 6.25: Conjugate gradient descent definition. In the space of spherical Hessian, the directions of the line searches are orthogonal, or more specifically $\Delta \mathbf{w}^t(t)\mathbf{H}\Delta \mathbf{w}(t+1) = 0$.

Comparing Eqs. 46 and 36 shows that conjugate gradient descent algorithm amounts to calculating a sort of "smart" momentum, i.e., one that increases learning speed where β plays the role of a momentum. If the error function is quadratic in n variables (e.g., weights), then the convergence is guaranteed in n iterations.

QuickProp

quickprop

6.10.2 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.

6.10.3 Levenberg-Marquardt

The Levenberg-Marquardt method is appropriate for intermediate size networks.

6.10.4 Counterpropagation

Occasionally, one wants a rapid prototype of a network, yet one that has expressive power greater than a mere two-layer network. Preliminary studies, or perhaps an excellent starting point for a full backpropagation network. Figure 6.26 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, and the hidden-to-output weights pool information from such centers of Voronoi cells. The processing at the hidden units is competitive learning (Sect. **??**.**??**).

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.

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

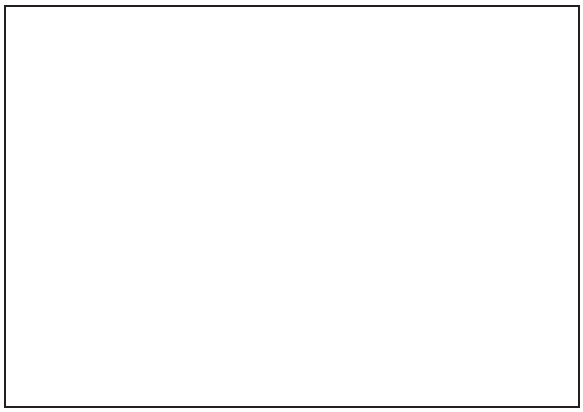

Figure 6.26: Counterpropagation network. The hidden units perform a winner-take-all, and the hidden-to-output weights are LMS.

6.10.5 Cascade Correlation

The central notion underlying cascade correlation networks is quite simple. One begins with a two-layer network and trains to minimum error (using, for example an LMS algorithm). If error is low enough, stop. In the far more common case in which the error is *not* low enough, one keeps the weights that already exist, but adds a single hidden unit, fully connected from inputs and to output units. In this way the network grows to a size that depends upon the problem at hand (Fig. 6.27).

The first unit picks up

The benefit is that often faster than strict backprop since weights are updated using local error signals (no (Computer exercise 11).

Algorithm 4 (Cascade-correlation)

1 **begin initialize** $\mathbf{a}$, criterion $\theta, \eta(\cdot), k = 0$
2 **do** $k \leftarrow k + 1$
3 $\mathbf{a} \leftarrow \mathbf{a} - \eta(k)\boldsymbol{\nabla} J(\mathbf{a})$
4 **until** $\eta(k)\boldsymbol{\nabla} J(\mathbf{a}) < \theta$
5 **return** $\mathbf{a}$
6 **end**$FIXthis$

Hence it seeks a linear model first, then progressively more complex ones.

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 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:

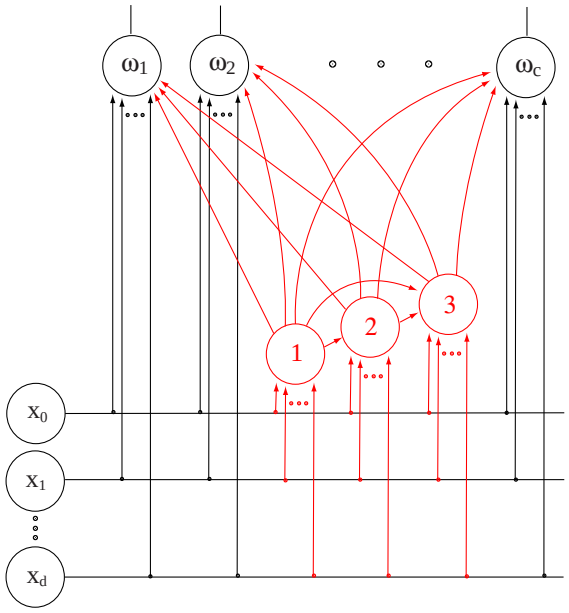

Figure 6.27: Cascade correlation

$$J = J_{pat} + \lambda J_{reg}. \tag{49}$$

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 Computational Complexity

Space complexity

Time complexity

6.11.2 Optimal Brain Damage/Surgeon

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. In practice, we find that neural networks with fewer weights typically generalize better than large networks with the same training error. 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:

$$\delta J = \underbrace{\left(\frac{\partial J}{\partial \mathbf{w}}\right)^T}_{\simeq 0} \cdot \delta \mathbf{w} + \frac{1}{2}\, \delta \mathbf{w}^T \cdot \underbrace{\frac{\partial^2 J}{\partial \mathbf{w}^2}}_{\equiv \mathbf{H}} \cdot \delta \mathbf{w} + \underbrace{O(\|\delta \mathbf{w}\|^3)}_{\simeq 0}\,, \tag{50}$$

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 **??**):

$$\delta \mathbf{w} = -\frac{w_q}{[\mathbf{H}^{-1}]_{qq}}\, \mathbf{H}^{-1} \cdot \mathbf{e}_q \quad \text{and} \quad L_q = \frac{1}{2}\, \frac{w_q^2}{[\mathbf{H}^{-1}]_{qq}}\,. \tag{51}$$

Here, $\mathbf{e}_q$ 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. 51 (Problem 33).

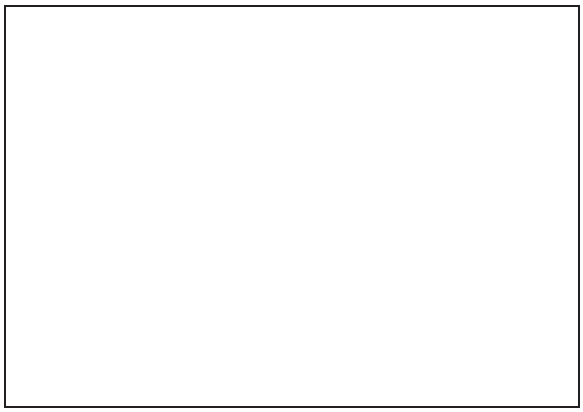

Figure 6.28: OBS caption.

Consider an arbitrary twice differentiable error $J_p(\mathbf{t}, \mathbf{o})$ where $\mathbf{t}$ is the desired output (teaching vector) and $\mathbf{o} = F(\mathbf{w}, \mathbf{x})$ the actual output. Given a weight vector $\mathbf{w}$, F maps the input vector $\mathbf{x}$ to the output; the total error over P patterns is $J = \frac{1}{P} \sum_{k=1}^{P} d(\mathbf{t}^{[k]}, \mathbf{o}^{[k]})$. It is straightforward to show that for a single output unit network the Hessian is:

$$\begin{aligned} \mathbf{H} &= \frac{1}{P} \sum_{k=1}^{P} \frac{\partial F(\mathbf{w}, \mathbf{x}^{[k]})}{\partial \mathbf{w}} \cdot \frac{\partial^2 d(\mathbf{t}^{[k]}, \mathbf{o}^{[k]})}{\partial \mathbf{o}^2} \cdot \frac{\partial F^T(\mathbf{w}, \mathbf{x}^{[k]})}{\partial \mathbf{w}} \\ &+ \frac{1}{P} \sum_{k=1}^{P} \frac{\partial d(\mathbf{t}^{[k]}, \mathbf{o}^{[k]})}{\partial \mathbf{o}} \cdot \frac{\partial^2 F(\mathbf{w}, \mathbf{x}^{[k]})}{\partial \mathbf{w}^2}\,. \end{aligned} \tag{52}$$

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 theeby guaranteeing that gradient descent will progress. Thus our Hessian reduces to:

$$\mathbf{H} = \frac{1}{P} \sum_{k=1}^{P} \frac{\partial F(\mathbf{w}, \mathbf{x}^{[k]})}{\partial \mathbf{w}} \cdot \frac{\partial^2 d(\mathbf{t}^{[k]}, \mathbf{o}^{[k]})}{\partial \mathbf{o}^2} \cdot \frac{\partial F^T(\mathbf{w}, \mathbf{x}^{[k]})}{\partial \mathbf{w}}\,. \tag{53}$$

We define $\mathbf{X}_k \equiv \frac{\partial F(\mathbf{w},\mathbf{x}^{[k]})}{\partial \mathbf{w}}$ and $a_k \equiv \frac{\partial^2 d(\mathbf{t}^{[k]},\mathbf{o}^{[k]})}{\partial \mathbf{o}^2}$, and can easily show that the recursion for computing the inverse Hessian becomes:

$$\mathbf{H}_{k+1}^{-1} = \mathbf{H}_k^{-1} - \frac{\mathbf{H}_k^{-1} \cdot \mathbf{X}_{k+1} \cdot \mathbf{X}_{k+1}^T \cdot \mathbf{H}_k^{-1}}{\frac{P}{a_k} + \mathbf{X}_{k+1}^T \cdot \mathbf{H}_k^{-1} \cdot \mathbf{X}_{k+1}} \ , \ \mathbf{H}_0^{-1} = \alpha^{-1}\mathbf{I}, \text{ and } \quad \mathbf{H}_P^{-1} = \mathbf{H}^{-1} \ , \quad (54)$$

where α is a small parameter — effectively a weight decay constant (Problem 28). Note how different error measures $d(\mathbf{t}, \mathbf{o})$ scale the gradient vectors $\mathbf{X}_k$ forming the Hessian (Eq. **??**). For the squared error $d(\mathbf{t}, \mathbf{o}) = (\mathbf{t} - \mathbf{o})^2$, we have $a_k = 1$, and all gradient vectors are weighted equally. The cross entropy or Kullback-Leibler distance,

$$d(\mathbf{t}, \mathbf{o}) = \mathbf{o} \log \frac{\mathbf{o}}{\mathbf{t}} + (1 - \mathbf{o}) \log \frac{(1 - \mathbf{o})}{(1 - \mathbf{t})} \ , \qquad 0 \leq \mathbf{o}, \mathbf{t} \leq 1 \quad (55)$$

yields $a_k = \frac{1}{\mathbf{o}^{[k]}(1-\mathbf{o}^{[k]})}$. Hence if $\mathbf{o}^{[k]}$ is close to zero or one, $\mathbf{X}_k$ is given a large weight in the Hessian; conversely, the smallest value of a_k occurs when $\mathbf{o}^{[k]} = 1/2$. This is desirable and makes great intuitive sense, since in the cross entropy norm the value of $\mathbf{o}^{[k]}$ is interpreted as the probability that the kth input pattern belongs to a particular class, and therefore we give large weight to $\mathbf{X}_k$ whose class we are most certain and small weight to those which we are least certain.

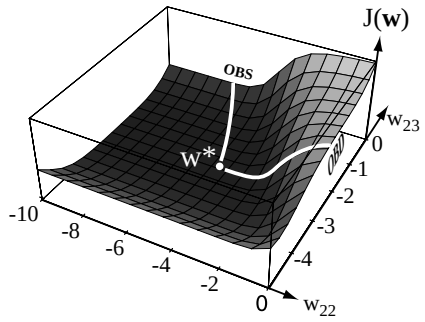

Figure 6.29: xxx

6.12 Architectural constraints

Architecture setting is generally problem dependent; nevertheless we can address some general topics without regard to a specific pattern recognition problem.

6.12.1 How many hidden layers?

The first general issue is how many layers. As we saw in Sect. 6.2.2, three suffice. There is empirical evidence that four-layer networks are more prone to being caught in local minima. Nevertheless, can use more for invariance, especially translation.

As a general rule, use three layers unless there is a principled reason for using more.

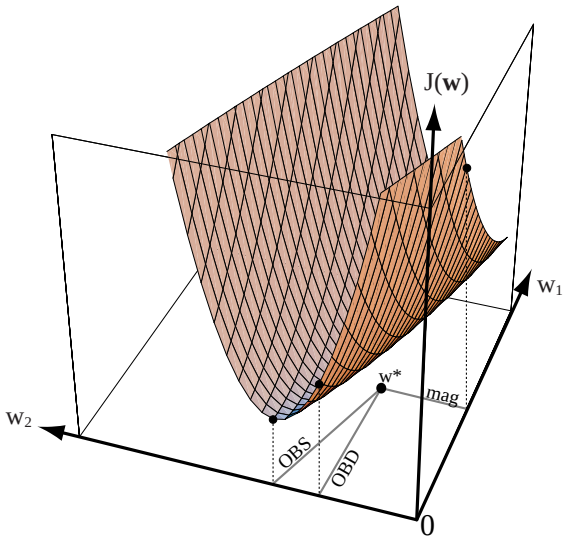

Figure 6.30: OBS OBD figure

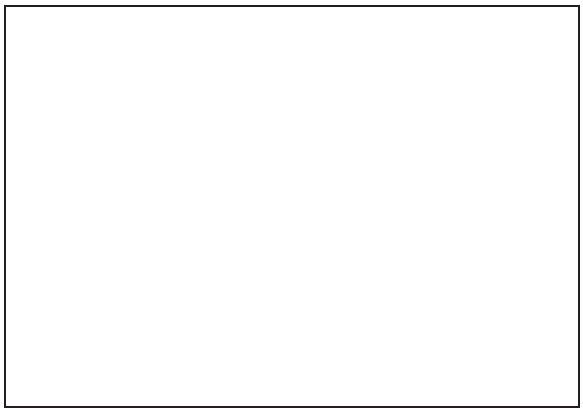

Figure 6.31: Three or four layers — how many is best?

6.12.2 Weight sharing

One of the great benefits of neural networks is that high-level constraints can be incorporated in the architecture, and the learning rule will guarantee that we get the maximum likelihood solution.

For instance, one may wish to have translation invariance. One way to insure this is to have the same pattern of weights covering one region of the input as another. We merely train with the patterns and force the pattern of weights to be the same over the input.

For instance, to implement translation invariance one explicitly forces the same pattern of weights covering one region of the input as another. We merely train with the patterns and force the pattern of weights to be the same over the input.

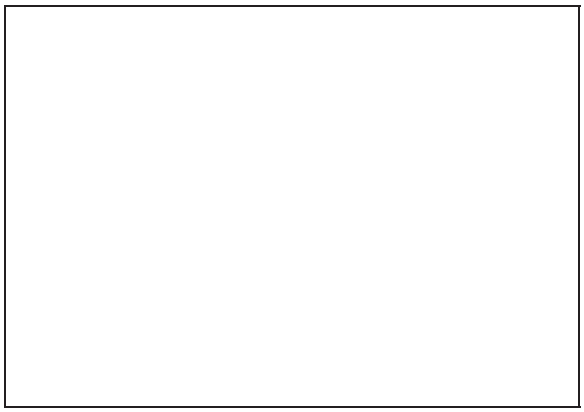

Figure 6.32: Weight sharing figure.

6.12.3 Time delay neural networks

Time dealy neural networks (or TDNNs) is an example of weight sharing. Here, however, the pattern of weights is repeated along a single dimension, which typically represents *time*. TDNNs have had their greatest use in neural net speech recognition, where time invariance is crucial — the recognizer should be insensitive to the precise arrival time of the utterance.

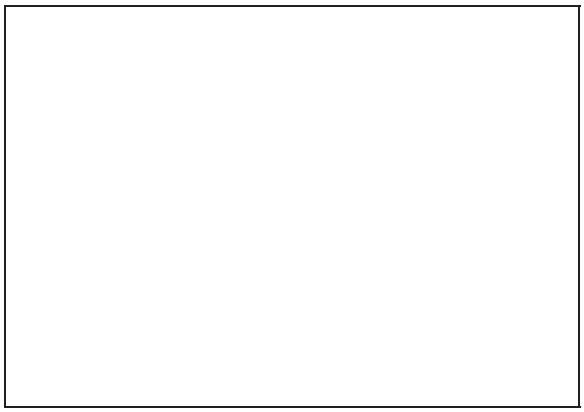

Figure 6.33: A time delay neural network (TDNN) uses weight sharing to impose time invariance.

6.12.4 Recurrent networks

Recurrent networks are another way to represent temporal information. The basic notion is that information can be fed back from higher layers to lower ones (Fig. 6.34).

One must be a bit more careful about the training rule for recurrent networks since the activation at a hidden unit is fed back upon itself. Thus its activation depends in a more complicated way upon the signals entering it.

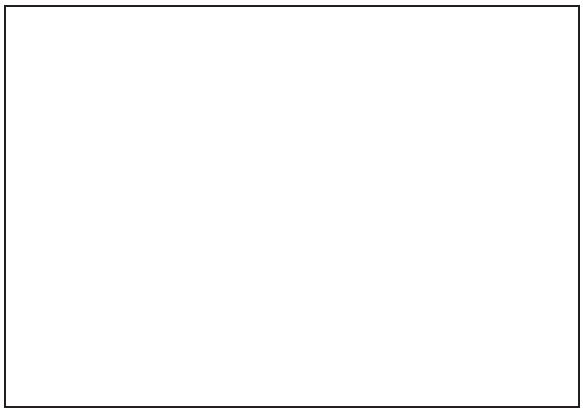

Figure 6.34: Recurrent networks

6.12.5 Functional link networks

Functional link networks (Fig. 6.35) replace the simple multiplication by weights we have seen before by a more complicated function.

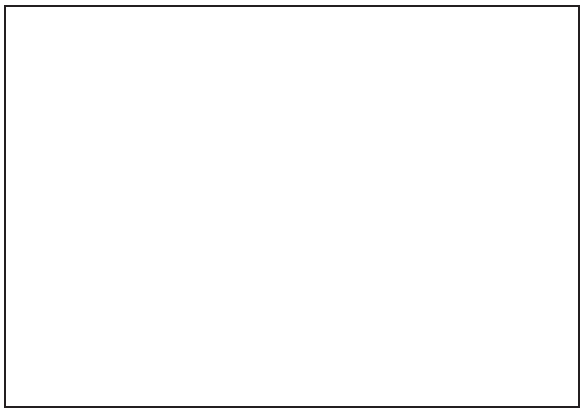

Figure 6.35: Functional link network.

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.

Architectural considerations such as the number of layers, interconnections of subnetworks and feedback connections can often be incorporated into networks easily to implement invariances such as time translation. A number of alternate training schemes such as conjugate gradient descent, the Levenberg-Marquardt algorithm and more architecturally motivated schemes such as cascade correlation and counterpropagation often yield acceptable performance.

Bibliographical and Historical Remarks

McCulloch and Pitts wrote the earliest paper providing a principled mathematical and logical treatment of the behavior of networks of simple neurons [46]. This pioneering work addressed non-recurrent as well as recurrent nets (those possessing "circles," in their terminology), but not learning. Its concentration on all-or-none or threshold function of neurons indirectly delayed the consideration of continuous valued neurons that would later dominate the field. These authors later wrote an extremely important paper on featural mapping (cf. Chap. **??**), invariances, and, obliquely, learning in nervous systems and thereby advanced the conceptual development of pattern recognition significantly [52].

Rosenblatt's work on the (two-layer) Perceptron (cf. Chap. **??**) [56, 57] was some of the earliest to address learning, and was the first to include rigorous proofs about convergence. A number of stochastic methods, including Pandemonium [61, 62], were developed for training networks with several layers of processors, though in keeping with the preoccupation with threshold functions, such processors generally computed logical functions (AND or OR), rather than some continuous functions favored in later neural network research. The limitations of networks implementing linear discriminants — linear machines — were well known in the 1950s and 1960s and discussed by both their promoters [57, cf., Chapter xx, "Summary of Three-Layer Series-Coupled Systems: Capabilities and Deficiencies"] and their detractors [48, cf., Chapter 5, "$\psi_{CONNECTED}$: A Geometric Property with Unbounded Order"].

A number of people trained weights at the output layer of three-layer networks [74, for a review]. Much of the difficulty in finding learning algorithms for all layers in a multilayer neural network (then called multilayer Perceptrons) came from the prevalent use of linear threshold units. Since these do not have useful derivatives throughout their entire range, the current approach of applying the chain rule for derivatives and the resulting "backpropagation of errors" did not gain more adherents earlier. Kalman filtering from electrical engineering [38, 28] uses an analog error (difference between predicted and measured output) for adjusting gain parameters in predictors.

Indeed, the work of Bryson, Denham and Dreyfus showed how Lagrangian methods could train multilayer networks for control (rather than pattern recognition), described in [5, Chapter 2].

An important related development from Widrow, Hoff and their colleagues [75, 76] was the use of analog signals and the LMS training criterion, applied to pattern recognition.

Werbos [71], too, discussed a method for backpropagation through time, which, if interpreted carefully carried the central ideas in backpropagation. Parker's early "Learning logic" [50, 51], developed independently, showed how layers of linear units could be learned by a sufficient number of input-output pairs. This work lacked simulations on representative or challenging problems (such as XOR) and too was not appreciated adequately. Le Cun independently developed a learning algorithm for three-layer networks [8, in French] in which target values are propagated, rather than derivatives; the resulting learning algorithm is equivalent to standard backpropagation, as pointed out shortly thereafter [9]. Without question, the paper by Rumelhart, Hinton and Williams [59], later expanded into a full and readable chapter [60], brought the backpropagation method to the attention of the widest audience. An enormous number of papers and books of applications — from speech production and perception, optical character recognition, data mining, finance, game playing and much more

— continues unabated. One novel class of for such networks includes generalization for production [20, 21]. One view of the history of backpropagation is [72], and two collections of key papers in the history of neural processing more generally, including many in pattern recognition, are [2, 1].

Clear elementary introduction to networks can be found in [45, 36], and several good textbooks, which differ from the current one in their emphasis on neural networks over other pattern recognition techniques, can be recommended [3, 55, 29, 27]. An extensive treatment of the mathematical aspects of networks, much of which is beyond that needed for mastering the use of networks for pattern classification, can be found in [19]. There is continued exploration of the strong links between networks and more standard statistical methods; White presents and overview [73], and proceedings such as [7] explore a number of close relationships. The important relation of multilayer Perceptrons to Bayesian methods and probability estimation can be found in [23, 54, 43, 4, 12, 58, 49]. Original papers on projection pursuit and MARS, can be found in [15] and [34], respectively, and a good overview in [55].

Shortly after its wide dissemination, the backpropagation algorithm was criticized for its lack of biological plausibility; in particular, Grossberg [22] discussed the non-local nature of the algorithm, i.e., that synaptic weight values were transported without physical means. Somewhat later Stork showed a local implementation of backpropagation was [65, 44], which was nevertheless highly implausible as a biological model.

The discussions and debates over the relevance of Kolmogorov's Theorem [39] to neural networks, e.g. [18, 40, 41, 33, 37, 11, 42], have centered on the expressive power. The proof of the univerasal expressive power of three-layer nets based on bumps and Fourier ideas appears in [31]. The expressive power of networks having non-traditional transfer functions was explored in [66, 67] and elsewhere.

The fact that three-layer networks can have local minima in the criterion function was explored in [47] and some of the properties of error surfaces illustrated in [35].

The simple method of weight decay was introduced in [32], and gained greater acceptance due to the work of Weigend and others [70]. While the Wald test [68, 69] has been used in traditional statistical research [63], its application to multilayer network pruning began with the work of Le Cun et al's Optimal Brain Damage method [10], later extended to include non-diagonal Hessian matrices [24, 25, 26], including some speedup methods [64]. A good review of the computation and use of second order derivatives in networks can be found in [6] and of pruning algorithms in [53].

Three-layer networks trained via cascade-correlation have been shown to perform well compared to standard three-layer nets trained via backpropagation [13]. Our presentation of counterpropagation networks focussed on just three of the five layers in a full such network [30]. Although there was little from a learning theory new presented in Fukushima's Neocognitron [16, 17], its use of many layers and mixture of hand-crafted feature detectors and learning groupings showed how networks could address shift, rotation and scale invariance. [14]

Problems

⨁ *Section 6.2*

1. Show that if the transfer function of the hidden units is linear, a three-layer network is equivalent to a two-layer one. Explain why, therefore, that a three-layer