6.8.5 Training with Noise

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

6.8.6 Manufacturing Data

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

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

6.8.7 Number of Hidden Units

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

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

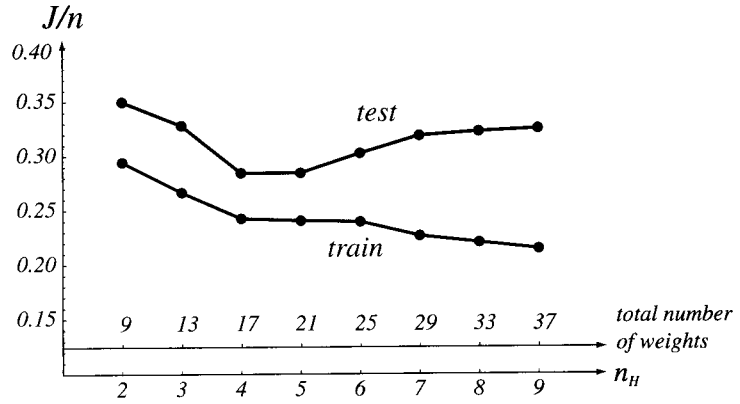

FIGURE 6.15. The error per pattern for networks fully trained but differing in the numbers of hidden units, n_H. Each $2 - n_H - 1$ network with bias was trained with 90 two-dimensional patterns from each of two categories, sampled from a mixture of three Gaussians, and thus $n = 180$. The minimum of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

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

6.8.8 Initializing Weights

UNIFORM LEARNING

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

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

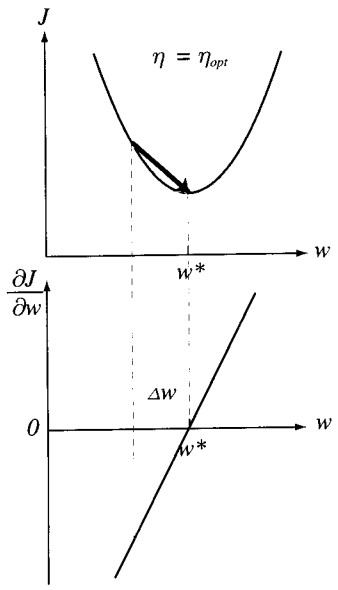

FIGURE 6.17. If the criterion function is quadratic (above), its derivative is linear (below). The optimal learning rate η_{opt} ensures that the weight value yielding minimum error, w^*, is found in a single learning step.

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

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

6.8.10 Momentum

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

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

represents learning with momentum.

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

6.8.9 Learning Rates

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

NONUNIFORM LEARNING

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

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

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

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

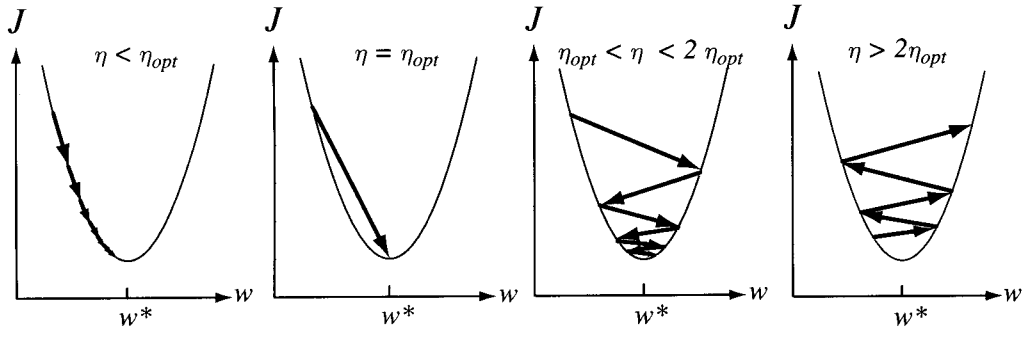

FIGURE 6.16. Gradient descent in a one-dimensional quadratic criterion with different learning rates. If $\eta < \eta_{opt}$, convergence is assured, but training can be needlessly slow. If $\eta = \eta_{opt}$, a single learning step suffices to find the error minimum. If $\eta_{opt} < \eta < 2\eta_{opt}$,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

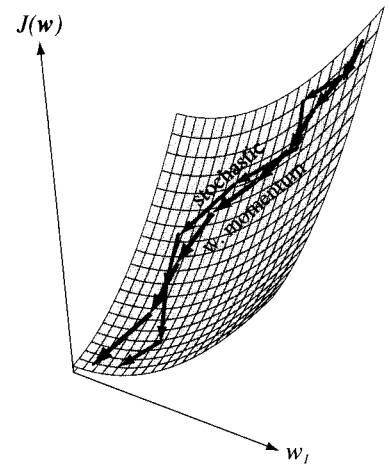

FIGURE 6.18. The incorporation of momentum into stochastic gradient descent by Eq. 37 (red arrows) reduces the variation in overall gradient directions and speeds learning.

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

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

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

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

6.8.11 Weight Decay

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

6.8.12 Hints