16 Heuristics for Improving Generalization

Given that data are limited and may be sampled in nonrandom ways, and that little is known about the "complexity" of the target function, the problem is to produce a system that fits the data as accurately as possible. One of the first tasks is to choose a network architecture. Even when generalization is not explicitly mentioned, the intent is usually to find a network that is powerful enough to solve the problem but simple enough to train easily and generalize well. Generalization criteria usually favor choosing the smallest network that will do the job, but in small networks back-propagation, for example, may be more likely to become trapped by local minima and may be more sensitive to initial conditions. If the algorithm cannot find a solution that does well on the training set, the solution it does find is not likely to do well on the test set either and generalization will be poor. Given limitations in the learning algorithm, a network that learns the problem reliably may be more complex than absolutely necessary and may not generalize as well as possible. Thus, additional techniques are often needed to aid generalization.

The following sections discuss some specific techniques that have been suggested as ways to improve generalization. Some are based on theoretical principles while others are more heuristic. Purely numerical techniques are considered first, followed by techniques using domain-dependent prior information.

16.1 Early Stopping

Figure 14.9 shows that generalization performance can vary with time during training. When the network is underconstrained, the generalization error may reach a minimum but then increase as the network fits peculiarities of the training set that are not characteristic of the target function. One approach to avoid overfitting is to monitor the generalization error and stop training when the minimum is observed. The generalization error is commonly estimated by simple cross-validation with a holdout set although more sophisticated estimates may be used. In [243, 371, 337], the generalization ability of the network is estimated based on its pre- and post-training performance on previously unseen training data. Early-stopping is compared to a number of other nonconvergent training techniques by Finnoff, Hergert, and Zimmermann [123, 122]. A practical advantage of early stopping is that it is often faster than training to complete convergence followed by pruning.

Although early stopping can be effective, some care is needed in deciding when to stop. As noted in section 14.5.3, the validation error surface may have local minima that could fool simple algorithms into stopping too soon [16]. The generalization *vs.* time curve may also have long flat regions preceding a steep drop-off [16]. It should also be noted that figure 14.9 represents an idealized situation; the training curves are often noisy and may need to be filtered. A simple way to avoid many of these problems is to train until

the network is clearly overfitting, retaining the best set of weights observed along the trajectory.

Although early stopping helps prevent overfitting, the results apply only to the chosen network. To achieve the best possible generalization, it is still necessary to test other network configurations and additional criteria will probably be needed to choose among them. The fact the overtraining is not observed in one training trial does not mean that it will not occur in another and is not proof that a suitable network size has been selected.

It can be argued that part of the reason for the relative success of back-propagation with early stopping is that it has a built-in bias for simple solutions because, when initialized with small weights, the network follows a path of increasing complexity from nearly constant functions to linear functions to increasingly nonlinear functions. Training is normally stopped as soon as some nonzero error criterion is met, so the algorithm is more likely to find a simple solution than a more complex solution that gives the same result. Cross-validation is a method for comparing solutions, but *stopping* when the validation error is minimum takes advantage of these special dynamics. It may be less effective for systems initialized with large weights or second-order algorithms that make large weight changes at each iteration.

16.2 Regularization

A problem is said to be *ill-posed* if small changes in the given information cause large changes in the solution. This instability with respect to the data makes solutions unreliable because small measurement errors or uncertainties in parameters may be greatly magnified and lead to wildly different responses. In contrast, a problem is *well-posed* if (*i*) it has a solution, (*ii*) the solution is unique, and (*iii*) the solution varies continuously with the given data. Violation of any of these conditions makes the problem ill-posed [370].

The idea behind regularization is to use supplementary information to restate an ill-posed problem in a stable form. The result will be a well-behaved, but approximate, solution of the original problem. Ideally, the bias introduced by the approximation will be more than offset by the gain in reliability. In general, domain-specific knowledge will be needed to stabilize a problem without changing it fundamentally.

Regularization has been studied extensively for linear systems. The book by Tikhonov and Arsenin [370] is a classic reference. In the context of learning from limited data, generalization is an unrealistic goal unless additional information is available beyond the training samples. One of the least restrictive assumptions is that the target function is smooth, that is, that small changes in the input do not cause large changes in the output. Given two functions that fit the data equally well, we tend to prefer the

smoother one because it is somehow simpler or more efficient. This bias is embedded in the learning algorithm by adding terms to the cost function to penalize nonsmooth solutions. In addition to the usual term E_o measuring the approximation error, we add terms $\Omega(y)$ which measure how well the approximation function $y(\mathbf{x})$ conforms to our preferences

$$E = E_o + \lambda\Omega(y). \tag{16.1}$$

The regularizing parameter λ balances the trade-off between minimizing the approximation error and conforming to the external constraints. A regularizer favoring smooth functions is [300]

$$E = E_o + \lambda\|Py\|^2 \tag{16.2}$$

where the regularizer P is a differential operator. This rewards smooth functions (whose derivatives are small, on average) and penalizes nonsmooth functions (those with large derivatives).

Regularization can be fit into a Bayesian approach [274, 140]. Equation 16.2, for example, corresponds to a prior in equation 15.6

$$P\left[f = y\right] \propto \exp(-\lambda\|Py\|^2). \tag{16.3}$$

Approximation with radial basis functions (which are linear in their output weights) is equivalent to classical regularization under certain conditions [274, 140]. Radial basis functions, however, form mostly local internal representations and therefore usually do not generalize as well as sigmoid networks (e.g., [56]). Curvature-driven smoothing using second derivative information as a means of improving generalization in radial basis function nets is discussed by Bishop [42].

Regularization provides a way of biasing the learning algorithm, but its success depends on the choice of an appropriate value for the regularization parameter λ to determine how strong the bias should be. In many of the other proposed heuristics there is a similar parameter balancing the need to minimize training error with other constraints. The parameter has an important effect on the eventual solution and is usually determined by criteria such as cross-validation. Although not discussed here, it is often useful to change the parameter dynamically because overfitting usually is not a problem until the later stages of learning. In many cases, it helps to impose the constraints only after the network has made some progress in reducing the initial error. In difficult problems, for example, there may be long periods before the network makes any significant progress. If a strong weight decay rule were in force during this period, the network might never escape from the initial set of weights around $\mathbf{w} = 0$.

16.3 Pruning Methods

Pruning algorithms are surveyed in chapter 13. The following paragraphs outline a few main points. Because the target function is unknown, it is difficult to predict ahead of time what size network will learn the data without overtraining. Not knowing the optimum network configuration, one can train many networks and choose the smallest or least complex one that learns the data. Although simple, this approach can be inefficient if many networks must be trained before an acceptable one is found. Even if the optimum size is known, the smallest networks just complex enough to fit the data may (depending on the learning algorithm) be sensitive to initial conditions and learning parameters. It may be hard to tell if the network is too small to learn the data, if it is simply learning very slowly, or if it is stuck in a local minima due to an unfortunate set of initial conditions or parameters. Thus, even if one finds a small network that will reliably learn the data, there might be a still smaller network that would work but is very difficult to train.

The pruning approach is to train a network that is somewhat larger than necessary and then remove unnecessary elements. The large initial size allows the network to learn reasonably quickly with less sensitivity to initial conditions and local minima while the reduced complexity of the trimmed system favors improved generalization. In several studies, e.g., [345, 344], pruning techniques produced solutions for small networks that generalized well and were not reliably obtainable by training the reduced network with random weights.

Although pruning techniques provide a means to simplify a network, they must be guided by other criteria to decide how simple the network should be. That is, there is still a need for external information and theoretical criteria to decide when to stop pruning.

16.4 Constructive Methods

Pruning methods train a larger-than-necessary network and then remove unneeded elements. The opposite approach is to build a network incrementally, adding elements until a suitable configuration is found. The two approaches are complementary and often used together. Like pruning, constructive techniques are a means of adjusting the size of a network rather than a method for deciding what size is appropriate. Other criteria are still necessary to decide when to stop adding elements. A number of constructive methods are discussed in chapter 12. Cascade-correlation [120] is often cited as an example.

16.5 Weight Decay

One way to implement a bias for simple or smooth functions is to favor networks with small weights over those with large weights. Large weights tend to cause sharp transitions in the node functions and thus large changes in output for small changes in the inputs. A simple way to obtain some of the benefits of pruning without complicating the learning algorithm much is to add a decay term like $-\beta w$ to the weight update rule. Weights that are not essential to the solution decay to zero and can be removed. Even if they aren't removed, they have no effect on the output so the network acts like a smaller system. Weight decay rules have been used in many studies, for example, [299, 388, 387, 227]. Several methods are compared by Hergert, Finnoff, and Zimmermann [165].

Weight decay can be considered as a form of regularization (e.g., [227]). Adding a $\beta \sum_i w_i^2$ regularizing term to the cost function, for example, is equivalent to addition of a $-\beta w_i$ decay term to the weight update rule. A drawback of the $\sum_i w_i^2$ penalty term is that it tends to favor weight vectors with many small components over those with a few large components, even when this is an effective choice. An alternative [386, 387, 388] is

$$\lambda \sum_i \frac{w_i^2/w_o^2}{1 + w_i^2/w_o^2}. \tag{16.4}$$

When λ is large, this is similar to weight decay methods. For $|w_i| \ll w_o$, the cost is small but grows like w_i^2 while, for $|w_i| \gg w_o$, the cost of a weight saturates and approaches a constant λ. (The developers call this form 'weight elimination' to differentiate it from simple weight decay.)

Soft weight sharing [286, 285] is another method that allows large weights when they are needed by using a penalty term that models the prior likelihood of the weights as a mixture of Gaussians. In practice, a number of Gaussians are used and their centers and widths are adapted to minimize the cost function. This reduces the complexity of the network by increasing the correlation among weight values.

Hard weight sharing is commonly used in image processing networks where the same kernel is applied repeatedly at different positions in the input image. In a neural network, separate hidden nodes may be used to compute the kernel at different locations and the number of weights could be huge. Constraining nodes that compute the same kernel to have the same weights greatly reduces the network complexity [91].

Example Figure 16.1 illustrates effects of weight decay. A 2/50/10/1 network was trained on 31 points using normal batch back-propagation (learning rate 0.01, momentum and weight decay 0). The network is very underconstrained. After 200 epochs the weight

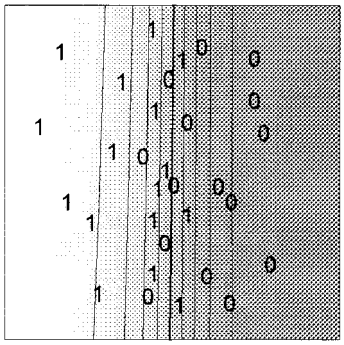

Figure 16.1
Effect of training with weight decay. A 2/50/10/1 network was trained using normal back-propagation for 200 epochs. Then weight decay was set to 1E-4 and training resumed for a total of 5000 epochs. Unlike figure 14.5, the decision surface is very simple and does not show obvious signs of overtraining.

decay was set to 1E-4 and training resumed for a total of 5000 epochs. Unlike figure 14.5 the decision surface is simple and smooth and doesn't show obvious signs of overtraining in spite of long training times. The response is basically that of a single sigmoid unit.

Figure 16.2 shows another example. Figure 16.2(a) shows the response of a network trained by normal batch back-propagation (learning rate 0.03, momentum and weight decay 0) until all patterns were correctly classified (error less than 0.1) at about 11,000 epochs. The network is underconstrained and the boundary is complex with steep transitions. Another net was trained with the same initial weights and learning rate but with weight decay increasing from 0 to 1E-5 at 1200 epochs, to 1E-4 at 2500 epochs, and to 1E-3 at 4000 epochs after which it was held constant. Figure 16.2(b) shows the response after 20,000 epochs. The surface is smoother and transitions are more gradual, but it could be argued that the data are still somewhat overfitted. Figure 16.2(c) shows the response after the learning rate was reduced to 0.01 and training resumed for another 1000 epochs. Further smoothing occurs because of the shift in balance between error minimization and weight decay.

In addition to showing the smoothing effects of weight decay, these examples show that the results may be hard to predict a priori. As in other regularization or penalty-term methods, there is a complex interaction between error minimization and constraint satisfaction. The particular value of the weight decay parameter (or regularization parameter in general) determines where equilibria occur, but it is difficult to predict ahead of time what value is needed to achieve desired results. The value 0.001 was chosen rather arbitrarily because it is a typically cited round number, but figure 16.2(b) is still perhaps somewhat overfitted.

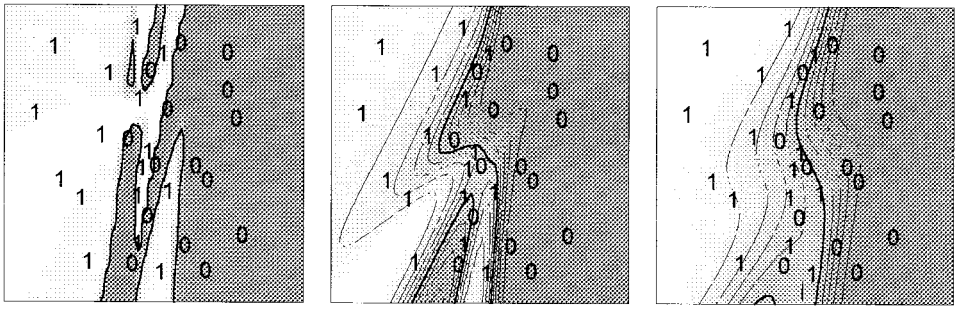

Figure 16.2
Effects of weight decay: (a) response of a 2/50/10/1 network trained by batch back-propagation until all patterns were correctly classified at about 11,000 epochs; (b) response after 20,000 epochs of a network trained from the same starting point with weight decay increasing to 0.001 at 4000 epochs; and (c) response of the network in (b) after 1000 more epochs with the learning rate decreased to 0.01.

16.6 Information Minimization

A heuristic for improving generalization based on the idea of information minimization is described by Kamimura, Takagi, and Nakanishi [205]. The uncertainty of a sigmoidal node is taken to be maximum when its activation is 0.5. A pseudo-entropy of the network for a particular set of patterns is defined as

$$H = -\sum_{k}^{K}\sum_{i}^{M}\left[v_i^k \log v_i^k + (1 - v_i^k)\log(1 - v_i^k)\right] \tag{16.5}$$

where K is the number of input patterns, M is the number of hidden units, and v_i^k is the activation of unit i for pattern k. The information in the network is given as

$$\begin{aligned} I &= H_{max} - H \\ &= KM \log 2 + H. \end{aligned}$$

The entropy is used as a penalty function to minimize the information contained in the network so the augmented error function is

$$E' = \beta E_o + \alpha H$$

where E_o is the standard sum of squared errors. Minimizing E' adds the term

$$\phi_i^k = v_i^k(1 - v_i^k)\log\frac{1 - v_i^k}{v_i^k}$$

to the weight adjustment rule, giving

$$\Delta w_{ij} = -\frac{\partial E}{\partial w_{ij}}$$
$$= \sum_k (\alpha \phi_i^k + \beta \delta_i^k)\, v_j^k.$$

Here, $\delta_i^k = \partial E_k / \partial a_i$ is the back-propagation delta term calculated in section 5.2. The use of H as a penalty term makes this an example of a regularization method. This also has effects similar to weight decay because (*i*) the entropy of a sigmoidal node is maximum when its output is 0.5, (*ii*) the output is 0.5 when the input is 0, and (*iii*) the input is 0 when the input weights are 0; that is, minimizing the weights would tend to minimize $-H$.

16.7 Replicated Networks

Another idea for improving generalization is to combine the outputs of several systems that differ in how they classify novel examples [245, 298, 189, 238, 63, 36, 111]. (Though not about neural networks per se [83] surveys many methods for combining forecasts.) The subsystems may differ due to variations in configuration, size, initialization, variations in the learning algorithm, differences in training data, and so on, or because they use completely different approximation models. The important factor is that they represent a variety of solutions to the same problem. There is no benefit in evaluating multiple models that all predict the same thing, after all.

With a mean-square error function, the best generalization would be expected when the system generates the expected value of all possible consistent functions, weighted by their probability of occurrence. That is

$$f^*(x) = \int f(x)\, p_f(f)\, df. \tag{16.6}$$

Averaging the output of different systems is a simple approximation to this expected value and tends to damp out extreme behaviors that might not be justified by the data. Additional advantages are improved fault-tolerance and the ability to retrain poorly performing subsystems using the ensemble average as the target.

Although more sophisticated combination methods are possible, a simple average may do as well as other methods in many cases [83]. A weighted average is often suggested

$$\sum_k c_k f_k(x). \tag{16.7}$$

The weighting factors c_k may be determined by a linear regression or depend on how well each subsystem performs on its training data; there are other possibilities. Because similar systems trained on similar data are likely to make similar predictions, colinearity of the $f_k(x)$ could make the linear regression ill-conditioned and result in a bad choice of c values. (This is one suggestion for why a simple average often does as well as more complicated methods.) Use of a convex linear combination in which $\sum_k c_k = 1$ is suggested in [60] for this reason.

Stacked-generalization [409] is a related method for improving generalization. Rather than simply averaging the outputs of several systems, the outputs are combined in more complex ways to maximize generalization.

The idea that replicating networks could help generalization might seem counterintuitive because N replicated networks would have N times as many weights and thus might need many more examples to constrain. The networks are trained independently, however, so the number of examples needed to train each does not change. If identical networks are trained on different subsets of the data (each net having a different holdout set used to control overfitting) and their outputs averaged to obtain the global output, this is similar to doing k-fold cross-validation or bootstrapping in parallel.

In general, a training set can contain regularities on many scales. Different subsystems with different biases, but trained with the same goals, are likely to agree about the large scale regularities that are obviously "supported by the data" while disagreeing mostly on smaller factors. An overtrained subsystem could choose a very idiosyncratic solution that is unlikely to match the real target function, but there are a huge number of ways to overfit the data and independent subsystems are likely to choose different ones. By averaging many responses, the total system expresses the consensus about obvious regularities recognized by most subsystems while avoiding extreme solutions in areas where there is disagreement.

A problem with this approach is that the number of systems that may need to be averaged in order to improve generalization significantly could be very large, particularly when the systems are complex; that is, the estimated mean in equation 16.6 could have a high variance. There is also still a need for external information to bias the learning algorithm to produce subnetworks that share the bias $p_f(f)$.

16.8 Training with Noisy Data

Many studies (e.g., [299, 118, 310, 387, 345, 246, 287, 267]) have noted that adding small amounts of input noise (jitter) to the training data often aids generalization and fault tolerance. Training with small amounts of added input noise embodies a smoothness assumption because we assume that slightly different inputs give approximately the same

output. If the noise distribution is smooth, the network will interpolate among training points in relation to a smooth function of the distance to each training point.

With jitter, the effective target function is the result of convolution of the actual target with the noise density [307, 306]. This is typically a smoothing operation. Averaging the network output over the input noise gives rise to terms related to the magnitude of the gradient of the transfer function and thus approximates regularization [307, 306, 45].

Training with jitter helps prevent overfitting in large networks by providing additional constraints because the effective target function is a continuous function defined over the entire input space whereas the original target function is defined only at the specific training points. This constrains the network and forces it to use excess degrees of freedom to approximate the smoothed target function rather than forming an arbitrarily complex surface that just happens to fit the sampled training data. Even though the network may be large, it models a simpler system.

Training with noisy inputs also gives rise to effects similar to weight decay and gain scaling. Gain scaling [228, 171] is a heuristic that has been proposed as a way of improving generalization. (Something like gain scaling is also used in [252] to "moderate" the outputs of a classifier.) Effects similar to training with jitter (and thus similar to regularization) can be achieved in single-hidden-layer networks by scaling the sigmoid gains [305, 306]. This is usually much more efficient than tediously averaging over many noisy samples. The scaling operation is equivalent to

$$\mathbf{w} \rightarrow \frac{\mathbf{w}}{\sqrt{\|\mathbf{w}\|^2\sigma^2 + 1}}$$

where σ^2 is the variance of the input noise. This has properties similar to weight decay. The development of weight decay terms as a result of training single-layer linear perceptrons with input noise is shown in [167]. Effects of training with input noise and its relation to target smoothing, regularization, gain scaling and weight decay are considered in more detail in chapter 17.

16.9 Use of Domain-Dependent Prior Information

The methods considered so far are mostly numerical techniques that make no use of problem-specific information. Another powerful way of favoring good generalization is through the use of domain-dependent prior information.

As noted earlier, samples alone are not enough to uniquely specify the target function in the absence of other constraints. In many applications where neural nets are considered, there is significant human knowledge that could be useful even though it is incomplete or

only partially reliable. There may be existing techniques that give reasonable but imperfect solutions or we may know certain rules that should be satisfied by any correct solution. When the goal is to develop a working application, it makes sense to use as much of this information as possible.

The following sections review some ways of using domain-dependent prior information in a neural network. Some are based on the idea of adapting a good non-neural solution to provide the starting point for further fine tuning in a neural network structure. It should be noted that whether or not this leads to good generalization depends on many factors; in some cases it may merely accelerate learning by giving the network a good headstart, without really improving generalization.

16.10 Hint Functions

One way to provide additional constraints is through the use of "hints" [361, 416]. In addition to outputs for the function of interest, extra output nodes are added to the network and trained to learn certain hint functions. The hint functions should be related to the function of interest and are usually designed to be easier to learn. The extra functions may speed convergence by generating nonzero derivatives in regions where the original function has plateaued. They may also aid generalization by providing additional constraints and removing certain local minima of the original function. They discourage the choice of a solution that somehow matches the original function on the training samples but does not include intermediate concepts embedded in the hints. After training, the hint output nodes can be removed because they usually are not of interest in the overall system.

The term hints is usually used to refer to augmented outputs, but hint information can also be provided in the form of targets for the (normally) hidden nodes. Hints can also be provided by shaping the target function dynamically [193]. The initial target function is an easy to learn, coarse approximation of the desired function which is gradually made more similar to the desired function as the learner masters each stage. This is a standard technique in animal training.

16.11 Knowledge-Based Neural Nets

Rule-based systems, such as expert systems, have been used quite successfully in many applications. These systems use human information efficiently and there is interest in developing hybrid systems combining the high-level information processing abilities of symbolic systems with the adaptability of neural nets. A useful feature of expert systems which neural networks generally lack is the ability to explain the reasoning behind its conclusions.

One approach [342, 375] is to embed symbolic rules in the initial structure of a neural network by translating the AND, OR, and NOT terms into corresponding network structures with appropriate weights. (Simple variable-free propositional rules are easily translated to neural network structures.) Additional links with small random weights are provided to let the system add other terms that may be useful. The network is then trained from examples to improve its performance. Because the embedded symbolic rules are often classifications, the cross-entropy error function may work better than the mean-squared-error function [342].

Besides faster training due to a good initial solution, improved generalization has been observed in spite of imperfect embedded rules. This is attributed to "(1) focusing attention on relevant input features, and (2) indicating useful intermediate conclusions (which suggest a good network topology)" [342]. Given a sufficient number of examples, a standard network initialized with random weights should converge to the same asymptotic performance, but the knowledge-based networks generalize better when examples are sparse. Evidently "the initial knowledge is 'worth' some number of training examples" [342]. Some references for ways of using forms of prior knowledge other than symbolic rules are provided by Shavlik [342].

16.12 Physical Models to Generate Additional Data

When there is no theoretical understanding of the target function, training from examples is one of few options. In many cases, however, there may be a physical model that can provide useful information even if it is not completely accurate. Possibilities include

- a rough model exists that accounts for the main variables only and ignores small details;
- an accurate model exists, but is too cumbersome to use in practice; or
- an exact model exists, but it is difficult or expensive to measure all the variables needed by the model.

Models can be useful to generate artificial training data for cases where it is difficult to obtain real training data. In physical control systems, for example, it may not be practical to obtain data for unusual operating modes such as process faults. Use of a model to generate additional artificial data for unusual operating modes of a steel rolling mill is described by Röscheisen, Hofmann, and Tresp [323].