# 13 Pruning Algorithms

A rule of thumb for obtaining good generalization is to use the smallest system that will fit the data. Unfortunately, it usually is not obvious what size is best so a common approach is to train a series of networks of various sizes and choose the smallest one that will learn the data. The problem is that this can be time consuming if many networks must be trained before an acceptable one is found. Even if the optimum size were known in advance, the smallest network just complex enough to fit the data may 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.

The idea behind pruning algorithms is to train an oversized network and then remove the unnecessary parts. The excess degrees of freedom of the large initial network allow it to learn reasonably quickly with less sensitivity to initial conditions while the reduced complexity of the trimmed system favors improved generalization. A side benefit is that the small resulting networks have other advantages such as economy and speed of operation. It may also be easier to interpret the logic behind their decisions as the network has less opportunity to spread functions over many nodes. (Some work has been done using pruning to help extract rules from a trained net, e.g., [374].)

The following sections summarize some pruning algorithms for feedforward networks like the multilayer perceptron, but the idea can also be applied to other systems such as associative networks [381] or tree structures [335].

**Example** Figure 13.1 illustrates the benefit of pruning. Figure 13.1(a) shows the boundary formed by an intentionally overtrained 2/50/10/1 network (after about 1800 epochs of RProp training). There are 671 weights in the network, but only 31 data points, so the network is very underconstrained. Although the data are nearly linearly separable (with some overlap near the boundary), the classification boundary found by the network is very nonlinear and would probably not generalize well on additional data from the same function. Figure 13.1(b) shows the same network after pruning by a simple algorithm. 659 of the original 671 weights and 56 of the 64 original nodes are removed. The network is reduced to 2/2/2/1 with 12 weights and the boundary is much smoother. Fig. 13.1(c) shows the response after 500 more epochs of training. The tuning smooths the response further and rotates the decision surface so the second input is almost ignored. It would probably be pruned in additional iterations. The simple algorithm used here had no way to remove hidden layers. A more sophisticated method could reduce the network to a one-dimensional solution with just 2 weights. (This illustrates pruning as a method of feature selection. If certain inputs are irrelevant to the problem, the algorithm should remove their connections to the network.)

---

1. Substantial parts of this chapter were published as [308].

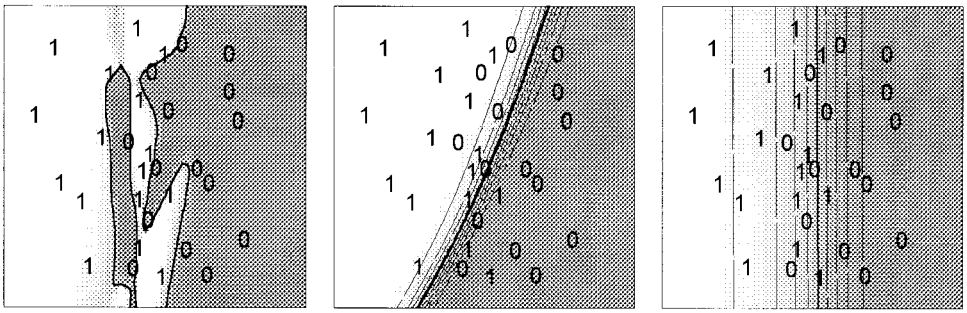


**Figure 13.1**
Effect of pruning: (a) response of an overtrained 2/50/10/1 network, (b) response after pruning (659 of the 671 original weights and 56 of the 64 original nodes have been removed to produce a 2/2/2/1 network—8 nodes including the bias node), (c) response after further training.

## 13.1 Pruning Algorithms

A brute-force pruning method is: for every weight, set the weight to zero and evaluate the change in the error; if it increases too much then restore the weight, otherwise remove it. On a serial computer, each forward propagation takes $O(W)$ time, where $W$ is the number of weights. This is repeated for each of the weights and each of $M$ training patterns resulting in $O(MW^2)$ time for each pruning pass. A number of passes are usually required. An even more cautious method would evaluate the change in error for all weights and patterns and then delete just the one weight with the least effect. This would be repeated until the least change in error reaches some threshold and could take $O(MW^3)$ time. This would be very slow for large networks, so most of the methods described in the following take a less direct approach.

Many of the algorithms can be put into two broad groups. One group estimates the sensitivity of the error function to removal of elements and deletes those with the least effect. The other group adds terms to the objective function that penalize complex solutions. Most of these can be viewed as forms of regularization. Many penalize large connection weights and are similar to weight decay rules. A term proportional to the sum of all weight magnitudes, for example, favors solutions with small weights; those that are nearly zero are unlikely to influence the output much and can be eliminated. There is some overlap in these groups because the objective function could include sensitivity terms.

In general, the sensitivity methods operate on a trained network. That is, the network is trained, sensitivities are estimated, and then weights or nodes are removed. The penalty-term methods, on the other hand, modify the cost function so that minimization drives

unnecessary weights to zero and, in effect, removes them during training. Even if the weights are not actually removed, the network acts like a smaller system.

## 13.2 Sensitivity Calculation Methods

### 13.2.1 Sensitivity Calculations I (Skeletonization)

Mozer and Smolensky [275] describe a method that estimates which units are least important and deletes them during training. A measure of the relevance, $\rho$, of a unit is the error when the unit is removed minus the error when it is left in place. Instead of calculating this directly for each and every unit, $\rho$ is approximated by introducing a gating term $\alpha$ in the node function

$$o_i = f(\sum_j w_{ij}\alpha_j o_j), \tag{13.1}$$

where $o_j$ is the activity of unit $j$, $w_{ij}$ is the weight to unit $i$ from unit $j$, and $f$ is the node nonlinearity. If $\alpha_j = 0$, unit $j$ has no influence on the network and can be removed; if $\alpha = 1$, the unit behaves normally. The importance of a unit is then approximated by the derivative

$$\hat{\rho}_i = -\left.\frac{\partial E^\ell}{\partial \alpha_i}\right|_{\alpha_i=1}, \tag{13.2}$$

which can be computed by back-propagation. This is evaluated at $\alpha = 1$ so $\alpha$ is merely a notational convenience rather than a parameter that must be implemented in the net. When $\hat{\rho}_i$ falls below a certain threshold, the unit can be deleted.

The usual sum of squared errors is used for training. To measure relevance, the function $E^\ell$

$$E^\ell = \sum \left|t_{pj} - o_{pj}\right| \tag{13.3}$$

is used rather than the sum of squared errors because it provides a better estimate of relevance when the error is small. Exponential averaging is used to suppress fluctuations

$$\hat{\rho}_i(t+1) = 0.8\hat{\rho}_i(t) + 0.2\frac{\partial E(t)}{\partial \alpha_i}. \tag{13.4}$$

Segee and Carter [338] study the effect of this pruning method on the fault tolerance of the system. Interestingly, they found that the pruned system is not significantly more

sensitive to damage even though it has fewer parameters. When the increase in error is plotted as a function of the magnitude of a weight deleted by a fault, the plots for the pruned and unpruned networks are essentially the same. They report that the variance of the weights into a node is a good predictor of the node's relevance and that the relevance of a node is a good predictor of the increase in RMS error expected when the node's largest weight is deleted.

### 13.2.2 Sensitivity Calculations II

Karnin [207] measures the sensitivity of the error function with respect to the removal of each connection and prunes the weights with low sensitivity. The sensitivity of weight $w_{ij}$ is given as

$$S_{ij} = -\frac{E(w^f) - E(0)}{w^f - 0} w^f \tag{13.5}$$

where $w^f$ is the final value of the weight after training, 0 is its value upon removal, and $E(0)$ is the error when it is removed.

Rather than actually removing the weight and calculating $E(0)$ directly, they approximate $S$ by monitoring the sum of all the changes experienced by the weight during training. The estimated sensitivity is

$$\hat{S}_{ij} = -\sum_{n=0}^{N-1} \frac{\partial E}{\partial w_{ij}} \Delta w_{ij}(n) \frac{w_{ij}^f}{w_{ij}^f - w_{ij}^i} \tag{13.6}$$

where $N$ is the number of training epochs and $w^i$ is the initial weight. All of these terms are available during training so the expression is easy to calculate and does away with the need for a separate sensitivity calculation phase. When $\Delta w$ is calculated by back-propagation, this becomes

$$\hat{S}_{ij} = \sum_{n=0}^{N-1} [\Delta w_{ij}(n)]^2 \frac{w_{ij}^f}{\eta(w_{ij}^f - w_{ij}^i)}. \tag{13.7}$$

If momentum is used, the general expression in equation 13.6 should be used.

After training, each weight has an estimated sensitivity and the lowest sensitivity weights can be deleted. Of course, if all output connections from a node are deleted, the node itself can be removed. If all input weights to a node are deleted, the node output will be constant so the node can be deleted after adjusting for its effect on the bias of following nodes.

### 13.2.3 Sensitivity Calculations III (Optimal Brain Damage)

Le Cun, Denker, and Solla [91] describe the optimal brain damage (OBD) method in which the saliency of a weight is measured by estimating the second derivative of the error with respect to the weight. They also reduce network complexity significantly by constraining groups of weights to be equal.

When the weight vector $\mathbf{w}$ is perturbed by an amount $\delta\mathbf{w}$, the change in error is approximately

$$\delta E = \sum_i g_i \delta w_i + \frac{1}{2}\sum_i h_{ii}\delta w_i^2 + \frac{1}{2}\sum_{i\neq j} h_{ij}\delta w_i \delta w_j + O(\|\delta W\|^3) \tag{13.8}$$

where $\delta w_i$ is the $i^{\text{th}}$ components of $\delta\mathbf{w}$, $g_i$ is component $i$ of the gradient of $E$ with respect to $\mathbf{w}$, and the $h_{ij}$ are elements of the Hessian matrix $\mathbf{H}$

$$g_i = \frac{\partial E}{\partial w_i}$$

$$h_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j}.$$

Because pruning is done on a well-trained network, the gradient is nearly zero and the first term in equation 13.8 can be ignored. When the perturbations are small, the last term will also be negligible. Because $\mathbf{H}$ may be a very large matrix, they make the simplifying assumption that the off-diagonal terms are zero. This leaves

$$\delta E \approx \frac{1}{2}\sum_i h_{ii}\delta w_i^2. \tag{13.9}$$

The second derivatives $h_{kk}$ can be calculated by a modified back-propagation rule in about the same amount of time as one back-propagation epoch. The saliency of weight $w_k$ is then

$$s_k = h_{kk} w_k^2/2. \tag{13.10}$$

Pruning is done iteratively, that is, train to a reasonable error level, compute saliencies, delete low saliency weights, and resume training. Application to pruning of tapped-delay networks is described in [365], in which the decision to stop pruning is based on the estimated generalization ability as determined by a modified AIC.

### 13.2.4 Sensitivity Calculations IV (Optimal Brain Surgeon)

For simplicity, the optimal brain damage method (section 13.2.3) assumes the Hessian is diagonal, but this is rarely true. In the optimal brain surgeon (OBS) method, Hassibi

and Stork [157, 159, 158] use a linear approximation of the full Hessian to obtain better estimates of the saliencies.

When the weight vector $\mathbf{w}$ is perturbed by an amount $\delta\mathbf{w}$, the change in the error is approximately

$$\delta E = \left(\frac{\partial E}{\partial \mathbf{w}}\right)^T \cdot \delta\mathbf{w} \; + \; \frac{1}{2}\delta\mathbf{w}^T \cdot \mathbf{H} \cdot \delta\mathbf{w} \; + \; O(\|\delta\mathbf{w}\|^3) \tag{13.11}$$

where $\mathbf{H} \equiv \partial^2 E/\partial\mathbf{w}^2$ is the Hessian matrix of second derivatives. Again, the first term is assumed to be near zero since $E$ is at a minima.

For each weight $w_q$ that could be set to zero, there is an associated adjustment of the remaining weights needed to reminimize the error. The algorithm combines these steps and seeks the weight adjustment vector $\delta\mathbf{w}$, which sets one weight $w_q$ to zero and causes the least increase in the error. The constraint that $\delta\mathbf{w}$ sets element $w_q$ to zero is expressed as $\mathbf{e}_q^T \cdot \delta\mathbf{w} + w_q = 0$ where $\mathbf{e}_q$ is the unit vector corresponding to weight $w_q$. The goal is to solve

$$\min_q \left\{ \min_{\delta\mathbf{w}} \left\{ \frac{1}{2}\delta\mathbf{w}^T \cdot \mathbf{H} \cdot \delta\mathbf{w} \right\} \quad \text{such that } \mathbf{e}_q^T \cdot \delta\mathbf{w} + w_q = 0 \right\}. \tag{13.12}$$

Using the Lagrangian

$$L = \frac{1}{2}\delta\mathbf{w}^T \cdot \mathbf{H} \cdot \delta\mathbf{w} + \lambda(\mathbf{e}_q^T \cdot \delta\mathbf{w} + w_q), \tag{13.13}$$

the optimal adjustment vector and resulting change in error are

$$\delta\mathbf{w} = -\frac{w_q}{\left[\mathbf{H}^{-1}\right]_{qq}}\mathbf{H}^{-1}\mathbf{e}_q \tag{13.14}$$

and

$$L_q = \frac{1}{2}\frac{w_q^2}{\left[\mathbf{H}^{-1}\right]_{qq}}. \tag{13.15}$$

Starting with a trained network, the procedure is: calculate $\mathbf{H}^{-1}$ and find the $q$ with the smallest saliency $L_q = w_q^2/(2[\mathbf{H}^{-1}]_{qq})$; if the change in error, $L_q$, is acceptable then delete weight $q$ and adjust the remaining weights with $\delta\mathbf{w}$ (equation 13.14). Recalculate $\mathbf{H}^{-1}$ and repeat. Stop when $L_q$ becomes too large and no more weights can be deleted without causing an unacceptable increase in error. Additional training may allow more weights to be deleted later.

The advantage of this method is that the remaining weights are readjusted automatically without the need for incremental retraining so the error introduced by eliminating the weight is generally lower. Unlike most other sensitivity methods, this method can account for correlated elements. Possible drawbacks are the time and memory needed to calculate and store $\mathbf{H}^{-1}$. Although the matrix inversion can be avoided by recursive calculations (using the assumption that $\frac{\partial E}{\partial \mathbf{w}} \approx 0$), the matrix may be large; $O(W^2)$ elements are needed for a network with $W$ weights. The network of figure 14.5, for example, with 671 weights, would require 449,000 elements. In practice, some approximation of the Hessian is needed.

Some of the storage and computational objections are addressed in [158]. The recursive procedure for generating $\mathbf{H}^{-1}$ is generalized to any twice-differentiable error function and a dominant eigenspace decomposition of the inverse Hessian is described.

One possible objection to the optimal brain damage and optimal brain surgeon methods is that they are based on a Taylor series approximation, which is reasonable only for *small* perturbations. Complete removal of a weight is not necessarily a small perturbation, so the calculated saliency may be a poor predictor of the actual change in error. It is easy to generate plots which show that the error surface is usually not well-approximated by a quadratic at large scales.

A minor related problem is that the gradient is assumed to be zero because the network is at a minima, but this is true only if the network is trained to complete convergence. With algorithmic variations such as early stopping and weight clipping, this may not be true. (Weight clipping puts limits on the maximum allowed weight magnitudes.) Many of the networks described in this text were trained until certain error levels were reached (e.g., all outputs correct to within a tolerance band) and exact disappearance of the gradient was not achieved. This is a minor point, however, because it is easy to include the gradient term of equation 13.8 in the salience calculation. Another practical consequence of lack of convergence to a minimum is that the Hessian may not be positive definite.

These problems do not appear to be too serious for initial pruning of an underconstrained network. Figure 13.2 shows a plot of the actual change in error observed when a weight was deleted *vs.* the 'optimal brain damage' saliencies for a 10/3/10 network trained on the simple autoencoder problem (10 1-of-10 patterns). For this network, the small saliencies seem to be reasonably good predictors of the change in error. A similar pattern is observed on the 2/50/10/1 networks used elsewhere in these notes. After the network is partially pruned, however, the correspondence between saliency and change is error is not as good (even after retraining and recalculation of the saliencies) and it is possible to find small saliency weights which produce much larger increases in error than larger saliency weights (e.g., saliencies greater than about 0.1 in the figure).

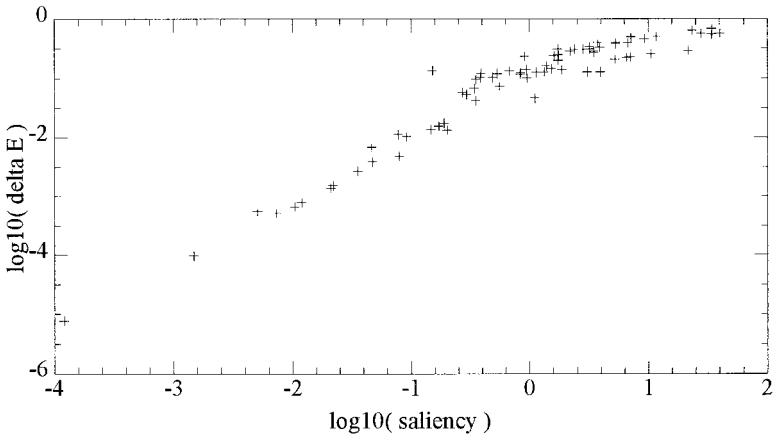


**Figure 13.2**
Pruning error versus weight saliency of optimal brain damage. Shown are the changes in error due to deletion of a weight versus the OBD saliency for a 10/3/10 network trained on the simple autoencoder problem.

## 13.3 Penalty-Term Methods

The methods described so far attempt to identify nonessential elements by calculating the sensitivity of the error to their removal. The methods described next modify the error function so that error minimization effectively prunes the network by driving weights to zero during training. The weights may be removed when they decrease below a certain threshold; even if they are not actually removed, the network still acts somewhat like a smaller system.

As an aside, it should be noted that the simple heuristic of deleting small weights is just a heuristic. Obviously, if a weight is exactly zero, it can be deleted without affecting the response. In many cases, it may also be safe to delete small weights because they are unlikely to have a large effect on the output. This is not guaranteed, so some sort of check should be done before actually removing the weight. Input weights may be small because they're connected to an input with a large range; weights to output nodes may be small because the targets have a small range. These objections are less important when inputs and outputs are normalized to unit ranges, but there are cases where small weights may be necessary; a small nonzero bias weight may be useful, for example, to put a boundary near but not exactly at the origin. A more cautious heuristic would be to use the weight magnitude to choose the order in which to evaluate weights for deletion.

Figure 13.3 shows the change in error due to deletion of a weight versus the weight magnitude for a 10/3/10 network trained on the simple autoencoder problem (10 patterns where 1 of 10 bits is set). The graph gives some support to the heuristic because the

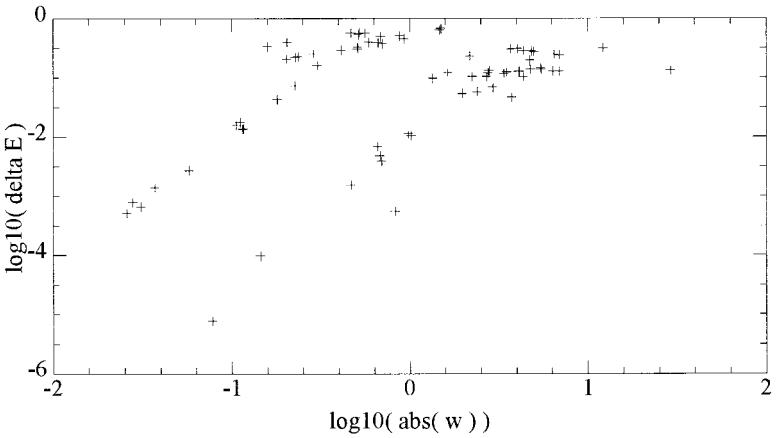


**Figure 13.3**
Pruning error versus weight magnitude. One of the simplest pruning heuristics is that small weights can be deleted since they are likely to have the least effect on the output. Shown are the errors due to deletion of a weight versus weight magnitude for a 10/3/10 network trained on the simple autoencoder problem. There are two separate, approximately linear trends. The upper group contains input-to-hidden weights while the lower group contains hidden-to-output weights.

smaller weights tend to give smaller changes in the error when the weight is deleted, but it also shows cases where smaller weights have more effect on the error than larger weights. The graph shows two separate approximately linear trends. The upper-left group contains mostly input-to-hidden weights, while the lower-right group contains mostly hidden-to-output weights. In many networks, weights close to the output may be larger than weights close to the input because the delta-attenuation effect of back-propagating through intermediate node nonlinearities causes input weights to grow slower.

### 13.3.1 Penalty Terms I

Chauvin [70], uses the cost function

$$C = \mu_{er} \sum_{j}^{P} \sum_{i}^{O} (d_{ij} - o_{ij})^2 + \mu_{en} \sum_{j}^{P} \sum_{i}^{H} e(o_{ij}^2) \tag{13.16}$$

where $e$ is a positive monotonic function. The sums are over the set of output units $O$, the set of hidden units $H$, and the set of patterns $P$. The first term is the normal sum of squared errors, the second term measures the average "energy" expended by the hidden units. The parameters $\mu_{er}$ and $\mu_{en}$ balance the two terms. The energy expended by a unit—how much its activity varies over the training patterns—is taken as an indication of its importance.

If the unit activity has a wide range of variation, the unit probably encodes significant information; if the activity does not change much, the unit probably does not carry much information.

Qualitatively different behaviors are seen depending on the form of $e$. Various functions are examined that have the derivative

$$e' = \frac{\partial e(o^2)}{\partial o_i^2} \quad = \quad \frac{1}{(1+o^2)^n}$$

where $n$ is an integer. For $n = 0$, $e$ is linear so high and low energy units receive equal differential penalties. For $n = 1$, $e$ is logarithmic so low energy units are penalized more than high energy units. For $n = 2$, the penalty approaches an asymptote as the energy increases so high energy units are not penalized much more than medium energy units. Other effects of the form of the function are discussed by Hanson and Pratt [154].

A magnitude-of-weights term may also be added to the cost function, giving

$$C = \mu_{er} \sum_{j}^{P} \sum_{i}^{O} (d_{ij} - o_{ij})^2 + \mu_{en} \sum_{j}^{P} \sum_{i}^{H} e(o_{ij}^2) + \mu_w \sum_{ij}^{W} w_{ij}^2. \tag{13.17}$$

Since the derivative of the third term with respect to $w_{ij}$ is $2\mu_w w_{ij}$, this effectively introduces a weight decay term into the back-propagation equations. Weights that are not essential to the solution decay to zero and can be removed.

Simulations described in [72, 73] use the cost function

$$C = \mu_{er} \sum_{ip}^{OP} (t_{ip} - o_{ip})^2 + \mu_{en} \sum_{ip}^{HP} \frac{o_{ip}^2}{1 + o_{ip}^2} + \mu_w \sum_{ij} \frac{w_{ij}^2}{1 + w_{ij}^2}. \tag{13.18}$$

No overtraining effect was observed despite long training times (with $\mu_{er} = .1$, $\mu_{en} = .1$, $\mu_w = .001$). Analysis showed that the network was reduced to an optimal number of hidden units independent of the starting size.

### 13.3.2 Penalty Terms II (Weight Elimination)

Weigend, Rumelhart, and Huberman [386, 387, 388] minimize the following cost function

$$\sum_{k \in T} (t_k - o_k)^2 + \lambda \sum_{i \in C} \frac{w_i^2 / w_o^2}{1 + w_i^2 / w_o^2} \tag{13.19}$$

where $T$ is the set of training patterns and $C$ is the set of connection indices. The second term (plotted in figure 13.4) represents the complexity of the network as a function of the weight magnitudes relative to the constant $w_o$. At $|w_i| \ll w_o$, the term behaves like

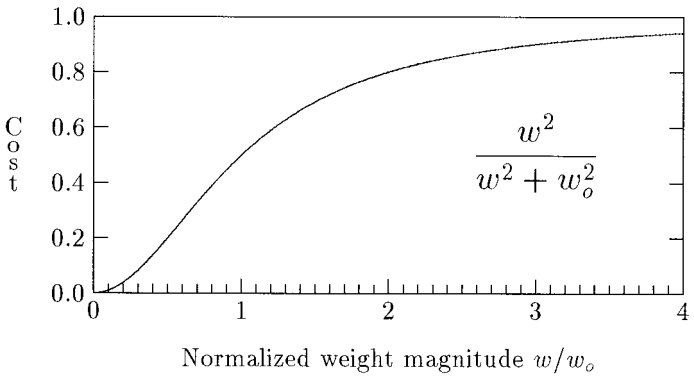


**Figure 13.4**
The normal weight-decay penalty term penalizes large weights heavily, which discourages their use even when they might be helpful. The weight-elimination penalty term in equation 13.19 saturates so large weights do not incur excess penalties once they grow past a certain value. This allows large weights when needed, while still encouraging the decay of small weights.

the normal weight decay term and weights are penalized in proportion to their squared magnitude. At $|w_i| \gg w_o$, however, the cost saturates so large weights do not incur extra penalties once they grow past a certain value. This allows large weights which have shown their value to survive while still encouraging the decay of small weights.

The value of $\lambda$ requires some tuning and depends on the problem. If it is too small, it will not have any significant effect; if it is too large, all the weights will be driven to zero. Heuristics for modifying $\lambda$ dynamically are described.

### 13.3.3 Penalty Terms III

Ji, Snapp, and Psaltis [195] modify the error function to minimize the number of hidden nodes and the magnitudes of the weights. They consider a single-hidden-layer network with one input and one *linear* output node. Beginning with a network having more hidden units than necessary, the output is

$$g(x; w, \theta) = \sum_{i=1}^{N} v_i f(u_i x - \theta_i) \tag{13.20}$$

where $u_i$ and $v_i$ are, respectively, the input and output weights of hidden unit $i$, $\theta_i$ is the threshold, and $f$ is the sigmoid function.

The significance of a hidden unit is computed by a function of its input and output weights

$$S_i = \sigma(u_i)\sigma(v_i) \tag{13.21}$$

where $\sigma(w) = w^2/(1 + w^2)$. This is similar to terms in the methods previously described.

The error is defined as the sum of $\mathcal{E}_o$, the normal sum of squared errors, and $\mathcal{E}_1$, a term measuring node significances.

$$\mathcal{E}(w, \theta) = \eta \mathcal{E}_o(w, \theta) + \lambda \mathcal{E}_1(w) \tag{13.22}$$

$$= \eta \sum_{\pi=1}^{M} [g(x^\pi; w, \theta) - y^\pi]^2 + \lambda \sum_{i=1}^{N} \sum_{j=1}^{i-1} S_i S_j \tag{13.23}$$

where $\pi$ indexes the training patterns, $x^\pi$ and $y^\pi$ are the input and desired output for pattern $\pi$, and $\eta$ and $\lambda$ are learning rate parameters. The $\mathcal{E}_1(w)$ term makes the algorithm favor solutions with fewer significant hidden units.

Conflict between the two error terms may cause local minima, so it is suggested the second term be added only after the network has learned the training set sufficiently well. Alternatively, $\lambda$ can be made a function of $\mathcal{E}_o$ such as

$$\lambda = \lambda_o e^{-\beta \mathcal{E}_o}. \tag{13.24}$$

When $\mathcal{E}_o$ is large, $\lambda$ will be small and vice versa.

A second modification to the weight update rule explicitly favors small weights

$$w_i^{n+1} = w_i^n - \eta \frac{\partial \mathcal{E}_o}{\partial w_i}(w^n, \theta^n) - \lambda \frac{\partial \mathcal{E}_1}{\partial w_i}(w^n) - \mu \tanh(w_i^n) \tag{13.25}$$

$$\theta_i^{n+1} = \theta_i^n - \eta \frac{\partial \mathcal{E}_o}{\partial w_i}(w^n, \theta^n) - \mu \tanh(\theta_i^n). \tag{13.26}$$

The new $\tanh(\cdot)$ term is modulated by $\mu$:

$$\mu = \mu_o \left| \mathcal{E}_o(w^n, \theta^n) - \mathcal{E}_o(w^{n-1}, \theta^{n-1}) \right|. \tag{13.27}$$

This reduces $\mu$ gradually and makes it go to zero when the target-value component $\mathcal{E}_o$ of the error function ceases to change.

Once an acceptable level of performance is achieved, small magnitude weights can be removed and training resumed. It was noted that the modified error functions increase the training time.

### 13.3.4 Weight Decay

Many of the penalty-term methods include terms that effectively introduce weight decay into the learning process, although the weights don't always decay at a constant rate. The

third term in equation 13.17, for example, adds a $-2\mu_w w_{ij}$ term to the update rule for $w_{ij}$. This is a simple way to obtain some of the benefits of pruning without complicating the learning algorithm much. A weight decay rule of this form was proposed by Plaut, Nowlan, and Hinton [299].

Ishikawa [187] proposed another simple cost function

$$C = \sum_{k \in T} (t_k - o_k)^2 + \lambda \sum_{i,j} |w_{ij}|. \tag{13.28}$$

The second term adds $-\lambda \operatorname{sgn}(w_{ij})$ to the weight update rule. Positive weights are decremented by $\lambda$ and negative weights are incremented by $\lambda$.

A drawback of the $\sum_i w_i^2$ penalty term is that it tends to favor weight vectors with many small components over vectors with a single large component, even when this is an effective choice. The weight elimination term in equation 13.19 addresses this by making the penalty saturate past a certain value. Nowlan and Hinton [286] describe soft weight sharing, which uses a more complex penalty term modeling the prior probability distribution of the weights as a mixture of Gaussians. Unlikely sets of weights under this distribution have a higher cost, so the weights tend to conform to the distribution during training. If the distribution consists of two Gaussians, for example, one narrow and one broad, both centered at zero with approximately equal mixing proportions, then the narrow Gaussian exerts a strong force attracting the small weights to zero. Larger weights, however, are less influenced by the narrow Gaussian and so only feel the weaker force of the wider Gaussian. In practice, more than two Gaussians are used and their centers and spreads are also adapted to minimize the cost function [286].

**Bias Weights** Some authors suggest that bias weights should not be subject to weight decay. In a linear regression $y = \mathbf{w}^T\mathbf{x} + \theta$, for example, the bias weight $\theta$ compensates for the difference between the mean value of the output target and the average weighted input. There is no reason to prefer small offsets, so $\theta$ should have exactly the value needed to remove the mean error.

In a sigmoidal node however, $y = f(\mathbf{w}^T\mathbf{x} + \theta)$, the bias $\theta$ has the effect of shifting the boundary sideways in the input space. If both the normal weights and the bias are scaled by the same factor $\lambda$, then the location of the boundary remains fixed while slope of the sigmoid transition varies. If the bias weight is not subject to equal decay, the boundary shifts as the weights change. The distance of the boundary from the origin is $\theta/\|\mathbf{w}\|$, where $\mathbf{w}$ excludes the bias weight; reduction of $\|\mathbf{w}\|$ by weight decay would shift the boundary away from the origin at the same time it reduces the slope. This suggests that bias weights should be subject to decay, at least for sigmoidal hidden nodes.

## 13.4 Other Methods

### 13.4.1 Interactive Pruning

Sietsma and Dow [345, 344] describe an interactive method in which the designer inspects a trained network and decides which nodes to remove. A network of linear threshold elements is considered. Several heuristics are used to identify noncontributing units:

- If a unit has a constant output over all training patterns, then it is not participating in the solution and can be removed. (Thresholds of units in following layers may need to be adjusted to compensate for its constant output.)
- If a number of units have highly correlated responses (e.g., identical or opposite), then they are redundant and can be combined into a single unit. All their output weights should be added together so the combined unit has the same effect on following units.

Units are unlikely to be exactly correlated (or have exactly constant output if sigmoid nodes are used), so application of the heuristics calls for some judgment.

A second stage of pruning removes nodes that are linearly independent of other nodes in the same layer, but which are not strictly necessary. The paper describes an example with four training patterns and a layer of three binary units. Two units are sufficient to encode the four binary patterns, so one of the three can be eliminated. It is possible for this to introduce linear inseparability (by requiring the following layer to do an XOR of the two units, for example), so a provision for adding hidden layers is included. This tends to convert short, wide networks to longer, narrower ones.

In a demonstration problem, the procedure was able to find relatively small networks that solved the test problem and generalized well. In comparison, randomly initialized networks of the same size (after pruning) were unable to learn the problem reliably. It was also observed that training with noise tends to produce networks which are harder to prune in the sense that fewer elements can be removed.

### 13.4.2 Local Bottlenecks

Kruschke [228, 230] describes a method in which hidden units "compete" to survive. The degree to which a unit participates in the function computed by the network is measured by the magnitude of its weight vector. This is treated as a separate parameter, the gain, and the weight vector is normalized to unit length. A unit with zero gain has a constant output; it contributes only a bias term to following units and doesn't back-propagate any error to preceding layers.

Units are redundant when their weight vectors are nearly parallel or antiparallel and they compete with others that have similar directions. The gains $g$ are adjusted according to

$$\Delta g_i^s = -\gamma \sum_{j \neq i} \cos^2 \angle(w_i^s, w_j^s) \cdot g_j^s \tag{13.29}$$

$$= -\gamma \sum_{j \neq i} \langle \hat{w}_i^s, \hat{w}_j^s \rangle^2 \cdot g_j^s, \tag{13.30}$$

where $\gamma$ is a small positive constant, $\hat{w}_i^s$ is the unit vector in the direction $w_i^s$, $\langle \cdot, \cdot \rangle$ denotes the inner product, and the superscript $s$ indexes the pattern presentations. If node $i$ has weights parallel to those of node $j$ then the gain of each will decrease in proportion to the gain of the other and the one with the smaller gain will be driven to zero faster. The gains are always positive so this rule can only decrease them. (If equation 13.30 results in negative gains, they are set to 0.) Once a gain becomes zero, it will remain zero so the unit can be removed.

Gain competition is interleaved with back-propagation. Because back-propagation modifies the weights, the gains are updated and the weights renormalized after each training cycle.

This method effectively prunes nodes by driving their gains to zero. The parameter $\gamma$ sets the relative importance of the gain competition and back-propagation. As usual, some tuning may be needed to balance error reduction and node removal. If $\gamma$ is large, competition will dominate error reduction and too many nodes may be removed.

Sperduti and Starita [354] describe a similar pruning method in conjunction with the use of gain-scaling for faster training. As with weight decay, gain decay lets the gains of unneeded nodes to decrease to zero; the node outputs become effectively constant and can be removed after adjusting biases of nodes to which they send output weights.

### 13.4.3 Distributed Bottlenecks

Kruschke proposes another solution that puts constraints on the weights rather than pruning them [228, 229]. The network starts with a large hidden layer of random weights and the dimensionality of the weight matrix is reduced during training. The number of nodes and weights remains the same, but the dimension of the space spanned by the weight vectors is reduced so the network behaves somewhat like a smaller network. The dimensionality reduction has an effect similar to pruning, but preserves redundancy and fault tolerance.

The method operates by spreading apart vectors that are farther apart than average and bringing together vectors that are closer together than average. Let $d_{ij} = \|w_i - w_j\|$ be the distance between vectors $w_i$ and $w_j$. The process starts with $H$ vectors with a mean of zero and an initial mean separation of $D$. At each step, the mean distance is

$$\overline{d} = \frac{2}{H(H-1)} \sum_{i<j} d_{ij}. \tag{13.31}$$

This calculation is nonlocal. The same paper describes a local method that works for the encoder problem but may not work for other problems.

After each back-propagation cycle, the weights are modified by

$$\Delta w_i = \beta \sum_{j \neq i} (d_{ij} - \overline{d})(w_i - w_j). \tag{13.32}$$

If $d_{ij} > \overline{d}$, then $w_i$ is shifted away from $w_j$. If $d_{ij} < \overline{d}$, then $w_i$ is shifted toward $w_j$. The vectors are then recentered and renormalized so that their mean is again zero and their mean separation is $D$, the initial mean distance. This is equivalent to doing gradient descent of the error function on the constraint surface.

In equation 13.32, $\beta$ is a small positive constant that controls the relative importance of back-propagation and dimensional compression. If $\beta$ is too large, all the vectors collapse into two antiparallel bundles—a single dimension—and effectively act like one node.

### 13.4.4 Principal Components Pruning

A similar idea is considered by Levin, Leen, and Moody [242]. Rather than actually eliminating links, the effective number of parameters is reduced by reducing the rank of the weight matrix for each layer.

The procedure starts with a trained network. For each layer starting with the first and proceeding to the output:

1. Calculate the correlation matrix $\Sigma$ for the input vector to the layer

$$\Sigma = \frac{1}{N} \sum_k u(k) u^T(k). \tag{13.33}$$

where $u(k)$ is the column vector of outputs of the previous layer for pattern $k$.

2. Diagonalize $\Sigma = C^T D C$ to obtain $C$ and $D$, the matrices of eigenvectors and eigenvalues of $\Sigma$. Rank the principal components in decreasing order.

3. Use a validation set to determine the effect of removing an eigennode. That is, set the least nonzero eigenvalue to zero and reevaluate the error. Keep the deletions that do not increase the validation error.

4. Project the weights onto the $l$ dimensional subspace spanned by the remaining eigenvectors

$$W \rightarrow W C_l C_l^T$$

where the columns of $C$ are the eigenvectors of the correlation matrix.

5. Continue with the next layer.

Advantages of the algorithm are that it is relatively easy to implement and reasonably fast. The dimensions of $\Sigma$ are determined by the number of nodes in a layer rather than the number of nodes or weights in the entire network so the matrix sizes may be more reasonable. Retraining after pruning is not necessary.

This procedure falls between OBD (optimal brain damage) and OBS (optimal brain surgeon) in terms of its use of the Hessian information. OBS uses a linearized approximation of the full Hessian matrix that improves accuracy but makes it impractical for large networks. OBD uses a diagonal approximation of the Hessian that is fast but inaccurate; errors may be large and subsequent retraining may be necessary. This method effectively uses a linear block-diagonal approximation of the Hessian and has a computational cost intermediate between OBD and OBS.

### 13.4.5 Pruning by the Genetic Algorithm

In a different approach, Whitley and Bogart [397] describe the use of the genetic algorithm to prune a trained network. Each population member represents a pruned version of the original network. A binary representation can be used with bits set to 0 or 1 to indicate whether a weight in the reference network is pruned or not. After mating, the offspring (probably) represent differently pruned networks. They are retrained for a small number of cycles to allow them to fix any damage that may have occurred. As a reward for using fewer weights, heavily pruned networks are given more training cycles than lightly pruned networks. The networks are then evaluated on the error achieved after training. This favors small networks, but not if they reduce size at the cost of increasing error.

As described, each pruned net begins retraining with weights from the original unpruned network. They suggest that it might be better to inherit the weights from the parents so that more drastically pruned networks don't have to adapt to such a large step in a single generation.

They also allow direct short-cut connections from input to output—something perfectly valid for back-propagation, but sometimes not considered by experimenters—and suggest that this speeds up learning and makes it less likely to be trapped in local minima. This also allows removal of unnecessary hidden layers and could be applied to most of the other methods described. The simple example in figure 13.1, for instance, would benefit from this.

## 13.5 Discussion

Pruning algorithms have been proposed as a way to exploit the learning advantages of larger systems while avoiding overfitting problems. Many of the methods listed either calculate the sensitivity of the error to the removal of elements or add terms to the error function that favor smaller networks.

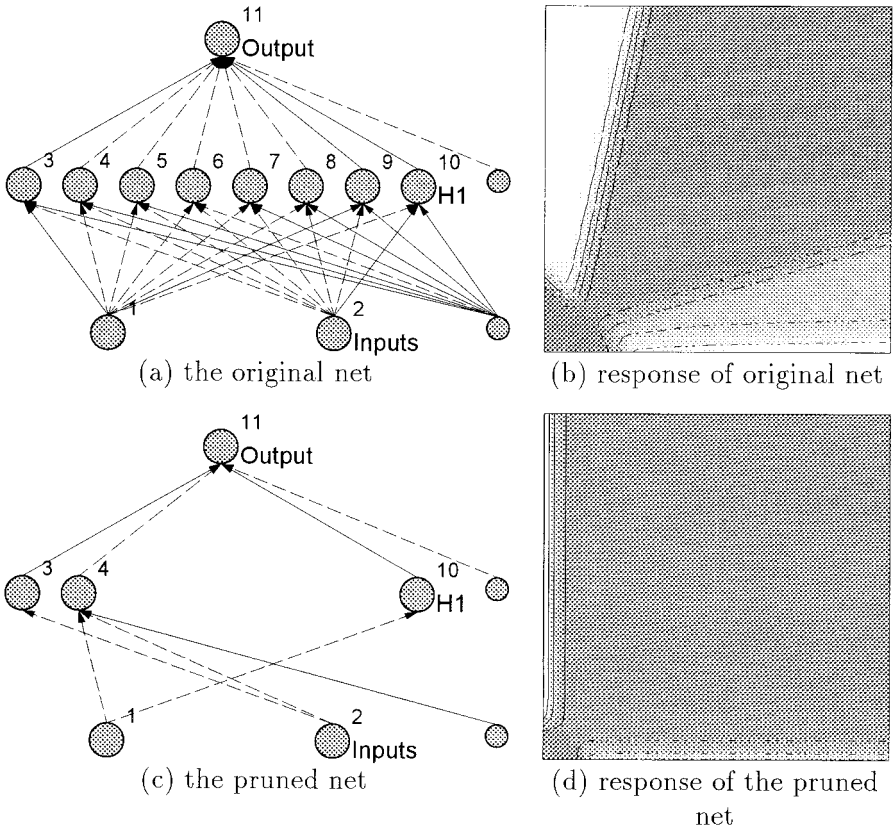


**Figure 13.5**
A pruning problem. The original network is underconstrained for the 2-input XOR problem and chooses a correct but unexpected solution. (The training points are the four corners of the square.) A naive pruning algorithm is able to remove many redundant elements, but the resulting network is unlikely to generalize better than the original.

A disadvantage of most of the sensitivity methods is that they do not detect correlated elements. The sensitivities are estimated under the assumption that $w_{ij}$ is the only weight to be deleted (or node $i$ is the only node to be deleted). After the first element is removed, the remaining sensitivities are not necessarily valid for the smaller network. An extreme example is two nodes whose effects cancel at the output. As a pair they have no effect on the output, but each has a strong effect individually so neither will be removed. Partially correlated nodes are a less extreme but more common example. The optimal brain surgeon method, which uses the full Hessian, is better able to deal with correlated weights.

In the original problem, there is the question of when to stop training. With pruning algorithms, there is the similar question of when to stop pruning. If separate training and validation sets are available, the choice may be clear; if not, it may be somewhat arbitrary. Sensitivity methods delete elements with the smallest sensitivities first and there may be a natural stopping point where the sensitivity jumps suddenly. Penalty-term methods control the amount of pruning by balancing the scaling factors of the error terms. This choice may be tricky, however, if it must be made before training begins, so some methods control these parameters dynamically. A compensating advantage of the penalty-term methods is that training and pruning are effectively done in parallel so the network can adapt to minimize errors introduced by pruning.

Although pruning and penalty term methods may often be faster than searching for and training a minimum size network, they do not necessarily reduce training times because larger networks may take longer to train due to sheer size and pruning takes some time itself. The goal, however, is improved generalization rather than the training speed.

It should be noted that pruning alone is not a sufficient tool to guarantee good generalization. Figure 14.11, for example, illustrates near minimal networks that probably generalize differently. (Except for some possibly unneeded bias connections, the 2/2/1 networks are minimal for the 2-bit XOR problem unless short-cut connections are used.) No connections can be removed so pruning is not an option.

It should also be noted that there is no guarantee that pruning will always lead to a network that generalizes better. It may be that the larger network allows an internal representation that is unavailable to a smaller network and it may not be possible to change from one form to the other by simply removing elements. Overfitting might be so severe that it cannot be corrected by pruning alone. It may also turn out that pruning simply removes redundant elements but leaves the overall response essentially unchanged. That is, elements may be removed because they have little effect on the output; removing them does not change the basic solution and does not lead to better generalization. If the pruning algorithm removes elements without adjusting remaining weights, the response of the pruned network is unlikely to be smoother than the original—especially when small weights are more likely to be removed than large ones. Figure 13.5 illustrates a case where the pruned network is likely to generalize worse than the original.