Pruning Algorithms—A Survey

Russell Reed, *Student Member, IEEE*

***Abstract*—A rule of thumb for obtaining good generalization in systems trained by examples is that one should use the smallest system that will fit the data. Unfortunately, it usually is not obvious what size is best; a system that is too small will not be able to learn the data while one that is just big enough may learn very slowly and be very sensitive to initial conditions and learning parameters. This paper is a survey of neural network pruning algorithms. The approach taken by the methods described here is to train a network that is larger than necessary and then remove the parts that are not needed.**

I. Introduction

When a system is trained by examples, an important issue is how well it generalizes to patterns outside the training set. For continuous domains, or large discrete ones, it is usually impossible to provide examples of every possible input. If the system simply memorizes the training patterns, it may do quite well during training but fail miserably when presented with similar but slightly different inputs. One would like the system to generalize from the training samples to the underlying function and give reasonable answers to novel inputs.

A rule of thumb for obtaining good generalization is to use the smallest system that will fit the data. Unfortunately, it usually isn't obvious what size is best so a common approach is to train successively smaller networks until the smallest one is found that will learn the data. This can be time consuming, however, since a number of networks must be trained and the smallest feasible networks may be sensitive to initial conditions and learning parameters and be more likely to become trapped in local minima.

The approach taken by the algorithms described in this paper is to train a network that is larger than necessary and then remove parts that are not needed. The large initial size allows the network to learn reasonably quickly with less sensitivity to initial conditions while the reduced complexity of the trimmed system favors improved generalization. The focus of the paper is on algorithms for feedforward networks such as the multilayer perceptron, but the idea can also be applied to other systems such as associative networks [33] or tree structures [24].

II. Overtraining and Generalization

When a neural network is trained, the weights are modified in order to decrease the error on the training patterns. If the network is tested on a slightly different set of examples of the same task, the error on the test set tends to decrease in step with the training error as the network generalizes from the training data to the underlying function. Fig. 1 illustrates the situation schematically. If the training data is incomplete, however, it may contain spurious and misleading regularities due to sampling. At some point, usually in the later stages of learning, the network starts to take advantage of these idiosyncrasies in the training data and the test error starts to increase again even though the training error continues to decrease. Chauvin describes an example of this type of overtraining in [6].

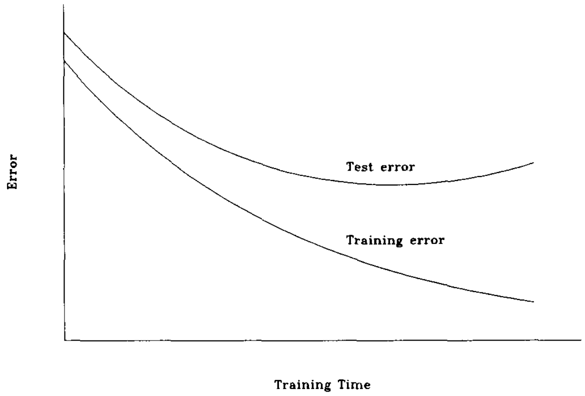


Fig. 1. In the early stages of training, the error on both the training and test sets tends to decrease with time as the network generalizes from the examples to the underlying function. At some point, however, the error on the test set reaches a minimum and begins to increase again as the network starts to adapt to artifacts in the training data.

One approach to avoid overfitting is to estimate the generalization ability during training and stop when it begins to decrease. The simplest method is to divide the data into a training set and a validation set, as above. The training set is used to modify the weights, the validation set is used to estimate the generalization ability, and training is stopped when the error on the validation set begins to rise. This technique, however, may not be practical when only a small amount of data is available, since the validation data cannot be used for training. In some recent work, based on a theoretical study of generalization [18], [31], [25], the generalization ability of the network is estimated based on its pre– and posttraining performance on previously unseen training data.

Another way of avoiding overtraining is to limit the ability of the network to take advantage of spurious correlations in the data. Overfitting is thought to happen when the network has more degrees of freedom (the number of weights, roughly) than the number of the training samples—when there are not enough examples to constrain the network. Even though it may give exactly the right output at the training points, it may be very inaccurate at other points. An example is a high–order

Manuscript received March 7, 1992; revised September 20, 1992.
The author is with the Department of Electrical Engineering, University of Washington, FT–10, Seattle, WA 98195.
IEEE Log Number 9204673.

1045–9227/93$03.00 © 1993 IEEE

Authorized licensed use limited to: Rutgers University. Downloaded on July 23,2020 at 19:04:32 UTC from IEEE Xplore. Restrictions apply.

polynomial fitted through a small number of points. As with polynomial approximations, a rule of thumb is to use the smallest system that will fit the data. If the system has only a limited number of degrees of freedom, it will use them to adapt to the largest regularities in the data and ignore the smaller (possibly spurious) ones.

Small networks have other advantages besides better expected generalization; they are also usually 1 faster and cheaper to build. Their operation may also be 2 easier to understand since there is less opportunity for the network to spread functions over many nodes. This may be important in critical applications where the user needs to know how the system works.

Formal learning theory [32], [3], [8] has been used to estimate the necessary size of a system. It relates the complexity of a learning system and the number of examples required to learn a particular function from a given class of functions. If the number of examples is small relative to the complexity of the system, the generalization error is expected to be high. The theory has been used to put bounds on the appropriate size of networks of linear threshold elements [1], [2]. These bounds, however, do not apply to networks with multiple continuous outputs and they do not say how to choose a suitable network given a particular set of examples to be learned, so choosing an appropriate network architecture is still something of an art.

Not knowing the optimum size, one can train a number of networks of various sizes and choose the smallest one that will learn the data. This approach, although straightforward, is rather inefficient since many networks may have to 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 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 approach taken by the algorithms described in this paper is to train a network that is larger than necessary and then remove the parts that are not needed. 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.

Example: Figs. 2 and 3 illustrate the effect of pruning. Fig. 2 shows the boundary formed by an intentionally overtrained network with 2 inputs, two layers of 50 and 10 hidden units, and a single ouput. There are 671 weights in the network, but only 31 data points, so the network is very underconstrained. Although the data is nearly linearly separable (with some overlap near the boundary), the network classification boundary is very nonlinear and will probably not generalize well on additional data from the same function. Fig. 3 shows the same network after pruning. The size of the network is reduced to 2/2/2/1 with 12 weights and the boundary is much smoother. A simple algorithm was used which had no way to remove hidden layers. A more sophisticated method could reduce the network to a one-dimensional solution with just 2 weights.

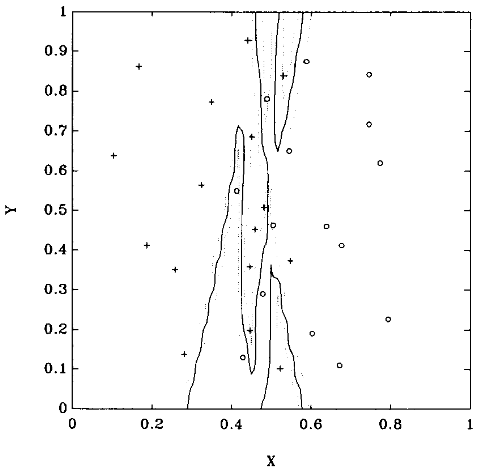


Fig. 2. Effect of overtraining. A 2/50/10/1 network with 671 weights trained on 31 points is very underconstrained. Although the points are nearly linearly separable, with some overlap, the decision surface is very nonlinear and is unlikely to generalize well. $X$and Y are the inputs to the network. + represent positive targets and o represent negative targets. The solid line shows the network decision surface, the 0.5 contour; the dotted lines show the 0.1 and 0.9 contours.

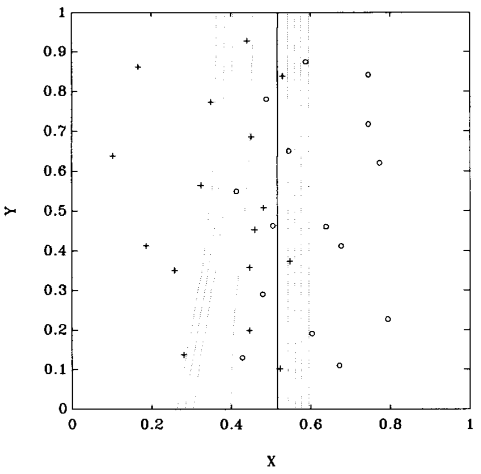


Fig. 3. Effect of pruning. The network of Fig. 2 was pruned to obtain a 2/2/2/1 network with 12 weights and this response. The boundary is much smoother and better generalization can be expected. X and Y are the inputs to the network. + represent positive targets and o represent negative targets. The solid line shows the network decision surface, the 0.5 contour; the dotted lines show other contours on 0.1 intervals.

The example also illustrates the point that pruning might be used as a method of feature selection. If certain inputs are irrelevant to the problem, the algorithm will remove their connections to the network.

III. Pruning Algorithms

A brute-force pruning method for every weight is, 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 $O(W)$

Authorized licensed use limited to: Rutgers University. Downloaded on July 23,2020 at 19:04:32 UTC from IEEE Xplore. Restrictions apply.

time, where W is the number of weights, and 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 conservative method would evaluate the change in error for all weights and patterns and then just delete 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. Since this can be very slow for large networks, most of the methods described below take a less direct approach.

Many of the algorithms can be put into two broad groups. ① One group estimates the sensitivity of the error function to removal of an element; the elements with the least effect can then be removed. The other group adds terms to the objective function that ② reward the network for choosing efficient solutions. A term proportional to the sum of all weight magnitudes, for example, favors solutions with small weights; those that are nearly zero are not likely to influence the output much and so can be eliminated. There is some overlap in these groups since the objective function could include sensitvity terms.

In general, the sensitivity methods modify a trained network, i.e., 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 back-propagation based on the function 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.

To find the least important methods.

IV. Sensitivity Calculation Methods

A. Sensitivity Calculations I

Mozer and Smolensky [19] describe a method which estimates which units are least important and deletes them during training.

A measure of the relevance ρ 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, ρ is approximated by introducing a gating term α for each unit such that

$$o_j = f(\sum_i w_{ji}\alpha_i o_i) \tag{1}$$

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

$$\hat{\rho}_i = -\left.\frac{\partial E^\ell}{\partial \alpha_i}\right|_{\alpha_i=1} \tag{2}$$

which can be computed by back-propagation. Since this is evaluated at $\alpha = 1$, α is merely a notational convenience rather than a parameter that must be implement 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. The error used to measure relevance is

$$E^\ell = \sum |t_{pj} - o_{pj}| \tag{3}$$

rather than the sum of squared errors, because this provides a better estimate of relevance when the error is small. An exponentially decaying average is used to suppress fluctuations

$$\hat{\rho}_i(t+1) = .8\hat{\rho}_i(t) + .2\frac{\partial E(t)}{\partial \alpha_i}. \tag{4}$$

Segee and Carter [26] 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 weighted deleted by a fault, the plots for the pruned and unpruned networks are essentially the same. They also found 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.

B. Sensitivity Calculations II

Karnin [13] 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

$$S_{ij} = -\frac{E(w^f) - E(0)}{w^f - 0} w^f \tag{5}$$

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

$$\hat{S}_{ij} = -\sum_{n=0}^{N-1} \frac{\partial E}{\partial w_{ij}} \Delta w_{ij}(n) \frac{w_{ij}^f}{w_{ij}^f - w_{ij}^i} \tag{6}$$

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 Δw is calculated by back-propagation, this becomes

$$\hat{S}_{ij} = \sum_{n=0}^{N-1} [\Delta w_{ij}(n)]^2 \frac{w_{ij}^f}{\eta(w_{ij}^f - w_{ij}^i)}. \tag{7}$$

If momentum is used, the general expression in (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, it will have a constant output and can be deleted after adjusting for it's effect on the bias of following nodes.

Authorized licensed use limited to: Rutgers University. Downloaded on July 23,2020 at 19:04:32 UTC from IEEE Xplore. Restrictions apply.

C. Sensitivity Calculations III

Le Cun *et al.* [7] measure the "saliency" of a weight by estimating the second derivative of the error with respect to the weight. They also reduce the network complexity by a large factor by constraining certain weights to be equal.

When the weight vector W is perturbed, the change in the error is approximately

$$\delta E = \sum_i g_i \delta w_i + \frac{1}{2} \sum_i h_{ii} \delta w_i^2 + \frac{1}{2} \sum_{i \neq j} h_{ij} \delta w_i \delta w_j + O(\|\delta W\|^2) \qquad (8)$$

where the δw_i's are the components of δW, g_i are the components of the gradient of E with respect to W, and the h_{ij} are elements of the Hessian matrix H

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

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

Since pruning is done on a well-trained network the first term in (8) will be zero because E is at a minimum. When the perturbations are small, the last term will be negligible. Since H is a very large matrix, they make the simplifying assumption that the off-diagonal terms are zero. This leaves

$$\delta E \approx \frac{1}{2} \sum_i h_{ii} \delta w_i^2. \qquad (9)$$

It turns out that the second derivatives h_{kk} can be calculated by a modified back-propagation rule. The saliency of weight w_k is then

$$s_k = h_{kk} w_k^2 / 2. \qquad (10)$$

Pruning is done iteratively: i.e., train to a reasonable error level, compute saliencies, delete low saliency weights, and resume training.

V. 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 below modify the error function so that normal back-propagation 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, the network still acts somewhat like a smaller system.

A. Penalty Terms I

In [4], Chauvin uses the cost function

$$C = \mu_{er} \sum_j^P \sum_i^O (d_{ij} - o_{ip})^2 + \mu_{en} \sum_j^P \sum_i^H e(o_{ij}^2) \qquad (11)$$

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 back-propagation error term, the second term measures the average "energy" expended by the hidden units. The parameters μ_{er} and μ_{en} balance the two terms. The "energy" expended by a unit—how much its activity varies over the training patterns—is an indication of its importance. If the unit changes a lot, it probably encodes significant information; if it does not change much, it probably does not carry much information.

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

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

where n is an integer. For $n = 0$, e is linear and high and low energy units receive equal penalties. For $n = 1$, e is logarithmic and 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 in [9].

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

$$C = \mu_{er} \sum_j^P \sum_i^O (d_{ij} - o_{ip})^2 + \mu_{en} \sum_j^P \sum_i^H e(o_{ij}^2) + \mu_w \sum_{ij}^W w_{ij}^2. \qquad (12)$$

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 which are not essential to the solution decay to zero and can be removed.

Simulations are described in [5], [6] which use the cost function

$$C = \mu_{er} \sum_{ip}^{OP} (t_{ip} - o_{ip})^2 + \mu_{en} \sum_{ip}^{HP} \frac{o_{ip}^2}{1 + o_{ip}^2} + \mu_w \sum_{ij} \frac{w_{ij}^2}{1 + w_{ij}^2}. \qquad (13)$$

No overtraining effect was observed despite long training times ($\mu_{er} = 0.1$, $\mu_{en} = 0.1$, $\mu_w = 0.001$) and analysis showed that the network was reduced to an optimal number of hidden units independently of the starting size.

B. Penalty Terms II

Weigend *et al.* [34]–[36] minimize the following cost function:

$$\sum_{k \in T} (t_k - o_k)^2 + \lambda \sum_{i \in C} \frac{w_i^2 / w_o^2}{1 + w_i^2 / w_o^2} \qquad (14)$$

where T is the set of training patterns and C is the set of all connections. The second term represents the complexity of the network as a function of the weight magnitudes relative to the constant w_o. For $|w_i| \gg w_o$, the cost of a weight approaches λ. For $|w_i| \ll w_o$, the cost is nearly zero.

When λ is large, this is similar to weight-decay methods. The value of λ requires some tuning and depends on the

Authorized licensed use limited to: Rutgers University. Downloaded on July 23,2020 at 19:04:32 UTC from IEEE Xplore. Restrictions apply.

problem. If it is too small, it won't have any significant effect; if it is too large, all the weights will be driven to zero. Heuristics for modifying λ dynamically are given.

C. Penalty Terms III

Ji *et al.* [12] 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 computed

$$g(x; w, \theta) = \sum_{i=1}^{N} v_i f(u_i x - \theta_i) \tag{15}$$

where u_i and v_i are, respectively, the input and output weights of hidden unit i, θ_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

$$S_i = \sigma(u_i)\sigma(v_i) \tag{16}$$

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

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.

$$\begin{aligned} \mathcal{E}(w, \theta) &= \eta\mathcal{E}_o(w, \theta) + \lambda\mathcal{E}_1(w) && (17) \\ &= \eta \sum_{\pi=1}^{M} [g(x^\pi; w, \theta) - y^\pi]^2 + \lambda \sum_{i=1}^{N} \sum_{j=1}^{i-1} S_i S_j && (18) \end{aligned}$$

where π indexes the training patterns and x^π and y^π are the input and desired output for pattern π, and η and λ 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, λ can be made a function of $\mathcal{E}_o$ such as

$$\lambda = \lambda_o e^{-\beta\mathcal{E}_o}. \tag{19}$$

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

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

$$\begin{aligned} w_i^{n+1} &= w_i^n - \eta \frac{\partial \mathcal{E}_o}{\partial w_i}(w^n, \theta^n) - \lambda \frac{\partial \mathcal{E}_1}{\partial w_i}(w^n) \\ &\quad - \mu \tanh(w_i^n) && (20) \\ \theta_i^{n+1} &= \theta_i^n - \eta \frac{\partial \mathcal{E}_o}{\partial w_i}(w^n, \theta^n) - \mu \tanh(\theta_i^n). && (21) \end{aligned}$$

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

$$\mu = \mu_o \big| \mathcal{E}_o(w^n, \theta^n) - \mathcal{E}_o(w^{n-1}, \theta^{n-1}) \big|. \tag{22}$$

This reduces μ 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. They note that the modified error functions increase the training time.

D. Weight Decay

Many of the penalty-term methods include terms which effectively introduce weight decay into the learning process, although the weights do not always decay at a constant rate. The third term in (12), 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 *et al.* [22].

Ishikawa [11] proposed another simple cost function

$$C = \sum_{k \in T} (t_k - o_k)^2 + \lambda \sum_{i,j} |w_{ij}|. \tag{23}$$

The second term adds $-\lambda \operatorname{sgn}(w_{ij})$ to the weight update rule. If $w_{ij} > 0$, the weight is decremented by λ, otherwise, if $w_{ij} < 0$, then it is incremented by λ.

A drawback of the $\sum_i w_i^2$ penalty term is that it tends to favor weight vectors with many small components over ones with a single large component, even when this is an effective choice. Nowlan and Hinton [20] describe a more complex penalty term that models the 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 [20].

VI. Other Methods

A. Interactive Pruning

Sietsma and Dow [27], [28] describe an interactive method in which the designer inspects a trained network and decides which nodes to remove. Several heuristics are used to identify units which don't contribute to the solution. A network of linear threshold elements is considered.

- If a unit has a constant output over all the training patterns, then it is not participating in the solution and can be removed. It may contribute to the bias of units in following layers, so their thresholds should be adjusted.
- If a number of units have highly correlated responses (e.g., identical or opposite) over all patterns, 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 judgement.

A second stage of pruning removes nodes that are linearly independent from other nodes in the same layer, but which aren't strictly necessary. They describe an example with four training patterns and a layer of three binary units. Since two

Authorized licensed use limited to: Rutgers University. Downloaded on July 23,2020 at 19:04:32 UTC from IEEE Xplore. Restrictions apply.

units are sufficient to encode the four patterns, 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, they were able to find relatively small networks that solved the problem and generalized well. For comparison, they attempted training random networks of the same size and found that they were unable to learn the problem reliably.

B. Local Bottlenecks

Kruschke [15] describes a method in which the 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 does not back-propagate any error to preceeding 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

$$\Delta g_i^s = -\gamma \sum_{j\neq i} \cos^2 \angle(w_i^s, w_j^s) \cdot g_j^s \tag{24}$$

$$= -\gamma \sum_{j\neq i} \langle \hat{w}_i^s, \hat{w}_j^s \rangle^2 \cdot g_j^s, \tag{25}$$

where γ 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. Since the gains are always positive, this rule can only decrease them. (If (25) results in negative gains, they are set to 0.) Once a gain becomes zero, it remains zero so the unit can be removed.

The gain competition is interleaved with back-propagation. Since back-propagation modifies the weights, the gains are updated and the weights renormalized after each back-propagation cycle.

This method effectively prunes nodes by driving their gains to zero. The parameter γ sets the relative importance of the gain competition and back-propagation. As in other methods, some tuning may be needed since, if γ is large, competition will dominate error reduction and too many nodes may be removed.

C. Distributed Bottlenecks

Kruschke proposes another solution that puts constraints on the weights rather than pruning them [15], [16]. For example, a network can start with a large hidden layer with random weights and then reduce the dimensionality of the weight space during training by introducing constraints. 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 making vectors that are farther apart than average even farther apart, and making vectors that are closer together than average even closer together. 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

$$\bar{d} = \frac{2}{H(H-1)} \sum_{i<j} d_{ij}. \tag{26}$$

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

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

$$\Delta w_i = \beta \sum_{j\neq i} (d_{ij} - \bar{d})(w_i - w_j). \tag{27}$$

If $d_{ij} > \bar{d}$, then w_i is shifted away from w_j. If $d_{ij} < \bar{d}$, 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 given constraint surface.

In (27), β is a small positive constant that controls the relative importance of back-propagation and dimensional compression. If β is too large, all the vectors collapse into two antiparallel bundles—a single dimension—and effectively act like one node.

D. Pruning by the Genetic Algorithm

In a different approach, Whitley and Bogart [37] describe the use of the genetic algorithm to prune a trained network. Each individual in the population represents a pruned version of the original network. A binary representation can be used, with bits set to 0 or 1 depending if 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 connections from input to output—something perfectly valid for back-propagation, but often not allowed by experimenters—and suggest that this speeds up learning and makes it less likely to be trapped in local minima. This also allows hidden layers to be removed if they are not needed and could be applied to most of the other methods described. The simple example in Section II, for instance, would benefit from this.

Authorized licensed use limited to: Rutgers University. Downloaded on July 23,2020 at 19:04:32 UTC from IEEE Xplore. Restrictions apply.

VII. Discussion

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

One 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 which cancel each other out at the output. As a pair, they have no effect on the output, but individually each has a large effect so neither will be removed. Partially correlated nodes are a less extreme, but more common, example.

In the original problem, there is the question of when to stop training. With the 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. The sensitivity methods delete the elements with the smallest sensitivities first and there may be a natural stopping point where the sensitivity jumps suddenly. The 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 of the 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.

Acknowledgment

The author thanks the reviewers for their helpful suggestions and R. J. Marks II and S. Oh for many helpful discussions on this and other topics.

References

[1] E. B. Baum, "When are k–nearest neighbor and back propagation accurate for feasible sized sets of examples?," in *Neural Networks, Proc. EURASIP Workshop*, L. B. Almeida and C. J. Wellekens, Eds., Feb. 1990, pp. 2–25.

[2] E. B. Baum and D. Haussler, "What size net gives valid generalization?," *Neural Computation*, vol. 1, pp. 151–160, 1989.

[3] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth, "Learnability and the Vapnik–Chervonenkis dimension," *J. Ass. Comput. Mach.*, vol. 36, no. 4, pp. 929–965, 1989.

[4] Y. Chauvin, "A back-propagation algorithm with optimal use of hidden units," in *Advances in Neural Information Processing (1)*, D. S. Touretzky, Ed. (Denver 1988), 1989, pp. 519–526,

[5] Y. Chauvin, "Dynamic behavior of constrained back-propagation networks," in *Advances in Neural Information Processing (2)*, D.S. Touretzky, Ed. (Denver 1989), 1990, pp. 642–649.

[6] Y. Chauvin, "Generalization performance of overtrained back-propagation networks," in *Neural Networks, Proc. EUROSIP Workshop*, L. B. Almeida and C. J. Wellekens, Eds., Feb. 1990, pp. 46–55,

[7] Y. Le Cun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing (2)*, D.S. Touretzky, Ed. (Denver 1989), 1990, pp. 598–605.

[8] A. Ehrenfeucht, D. Haussler, M. Kearns, and L. Valiant, "A general lower bound on the number of examples needed for learning," in *Proc. 1988 Workshop Computational Learning Theory*, 1988.

[9] S. J. Hanson and L. Y. Pratt, "Comparing biases for minimal network construction with back-propagation," in *Advances in Neural Information Processing (1)*, D.S. Touretzky, Ed. (Denver 1988), 1989, pp. 177–185.

[10] F. Hergert, W. Finnoff, and H. G. Zimmermann, "A comparison of weight elimination methods for reducing complexity in neural networks," in *Proc. Int. Joint Conf. Neural Networks*, San Diego, CA, vol. III, 1992, pp. 980–987.

[11] M. Ishikawa, "A structural learning algorithm with forgetting of link weights," Tech. Rep. TR–90–7, Electrotechnical Lab., Tsukuba-City, Japan, 1990.

[12] C. Ji, R. R. Snapp, and D. Psaltis, "Generalizing smoothness constraints from discrete samples," *Neural Computation*, vol. 2, no. 2, pp. 188–197, 1990.

[13] E. D. Karnin, "A simple procedure for pruning back–propagation trained neural networks," *IEEE Trans. Neural Networks*, vol. 1, no. 2, pp. 239–242, 1990.

[14] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *Advances in Neural Information Processing (4)*, J. E. Moody, S. J. Hanson, and R. P. Lippmann, Eds., 1992, pp. 951–957.

[15] J. K. Kruschke, "Creating local and distributed bottlenecks in hidden layers of back-propagation networks," in *Proc. 1988 Connectionist Models Summer School*, D. Touretzky, G. Hinton, and T. Sejnowski, Eds., 1988, pp. 120–126,

[16] J. K. Kruschke, "Improving generalization in back–propagation networks with distributed bottlenecks," in *Proc. Int. Joint Conf. Neural Networks*, Washington DC, vol. I, 1989, pp. 443–447 ().

[17] S. Y. Kung and Y. H. Hu, "A Frobenius approximation reduction method (FARM) for determining optimal number of hidden units," in *Proc. Int. Joint Conf. Neural Networks*, vol. II, Seattle, 1991, pp. 163–168,

[18] E. Levin, N. Tishby, and S. A. Solla, "A statistical approach to learning and generalization in layered neural networks," *Proc IEEE*, vol. 78, no. 10, pp. 1568–1574, Oct. 1990.

[19] M. C. Mozer and P. Smolensky, "Skeletonization: A technique for trimming the fat from a network via relevance assessment," in *Advances in Neural Information Processing (1)*, D.S. Touretzky, Ed. (Denver 1988), 1989, pp. 107–115.

[20] S. J . Nowlan and G. E. Hinton, "Simplifying neural networks by soft weight-sharing," *Neural Computation*, vol. 4, no. 4, pp. 473–493, 1992.

[21] O. M. Omidvara and C. L. Wilson, "Optimizaton of neural network topology and information content using Boltzmann methods," in *Proc. Int. Joint Conf. Neural Networks (Baltimore)*, vol. IV, 1992, pp. 594–599.

[22] D. C. Plaut, S. J. Nowlan, and G. E. Hinton, "Experiments on learning by back propagation," Tech. Rep. CMU–CS–86–126, Carnegie-Mellon Univ., 1986.

[23] S. Ramachandran and L. Y. Pratt, "Information measure based skeletonisation," in *Advances in Neural Information Processing (4)*, J. E. Moody, S. J. Hanson, and R. P. Lippmann, Eds., 1992, pp. 1080–1087.

[24] A. Sankar and R. J. Mammone, "Optimal pruning of neural tree networks for improved generalization," in *Proc. Int. Joint Conf. Neural Networks*, vol. II (Seattle), 1991, pp. 219–224.

[25] D. B. Schwartz, V. K. Samalan, S. A. Solla, and J. S. Denker, "Exhaustive learning," *Neural Computation*, vol. 2, no. 3, pp. 374–385, 1990.

[26] B. E. Segee and M. J. Carter, "Fault tolerance of pruned multilayer networks," in *Proc. Int. Joint Conf. Neural Networks*, vol. II (Seattle), pp. 447–452, 1991.

[27] J. Sietsma and R. J. F. Dow, "Creating artificial neural networks that generalize," *Neural Networks*, vol. 4, no. 1, pp. 67–69, 1991.

[28] J. Sietsma and R. J. F. Dow, "Neural net pruning—why and how," in *Proc. IEEE Int. Conf. Neural Networks*, vol. I (San Diego), 1988, pp. 325–333,

[29] W. E. Simon and J. R. Carter, "Learning to identify letters with REM equations," in *Proc. Int. Joint Conf. Neural Networks*, vol. I (Washington, DC), 1990, pp. 727–730.

[30] W. E. Simon and J. R. Carter, "Removing and adding network connections with recursive error minimization (REM) equations," in *Applications of Artificial Neural Networks*, S.K. Rogers, Ed. 1990, pp. 600–606.

[31] N. Tishby, E. Levin, and S. A. Solla, "Consistent inference of probabilities in layered networks: Predictions and generalization," in *Proc. Int. Joint Conf. Neural Networks*, 1989, p. 403.

[32] L. G. Valiant, "A theory of the learnable," *Commun. Ass. Comput. Mach.*, vol. 27, no. 11, pp. 1134–1142, 1984.

[33] J.H. Wang, T. F. Krile, and J. F. Walkup, "Reduction of interconnection

Authorized licensed use limited to: Rutgers University. Downloaded on July 23,2020 at 19:04:32 UTC from IEEE Xplore. Restrictions apply.

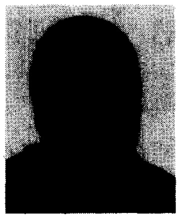
weights in higher order associative memory networks," in *Proc. Int. Joint Conf. Neural Networks*, vol. II (Seattle), 1991, pp. 177–182.

[34] A. S. Weigend, D. E. Rumelhart, and B. A. Huberman, "Back–propagation, weight-elimination and time series prediction," in *Proc. 1990 Connectionist Models Summer School*, D. Touretzky, J. Elman, T. Sejnowski, and G. Hinton, Eds., 1990, pp. 105–116.

[35] A. S. Weigend, D. E. Rumelhart, and B. A. Huberman, "Generalization by weight-elimination applied to currency exchange rate prediction," in *Proc. Int. Joint Conf. Neural Networks*, vol. I (Seattle), 1991, pp. 837–841.

[36] A. S. Weigend, D. E. Rumelhart, and B. A. Huberman, "Generalization by weight–elimination with application to forecasting," in *Advances in Neural Information Processing (3)*, R. Lippmann, J. Moody, and D. Touretzky, Eds., 1991, pp. 875–882.

[37] D. Whitley and C. Bogart, "The evolution of connectivity: Pruning neural networks using genetic algorithms," in *Proc. Int. Joint Conf. Neural Networks*, vol. I (Washington, DC), 1990, p. 134.



Russell Reed (S'79–M'81–S'90) received the B.S. and M.S. degrees in electrical engineering from Texas A&M University in 1981 and 1986, respectively. He is presently working towards the Ph.D. degree at the University of Washington, Seattle.

From 1986 to 1990 he worked at World Instruments in Longview, TX, designing microprocessor based instruments and PC software. His research interests include adaptive/learning systems, pattern recognition, and neural networks.

Mr. Reed is a member of the INNS.

Authorized licensed use limited to: Rutgers University. Downloaded on July 23,2020 at 19:04:32 UTC from IEEE Xplore. Restrictions apply.