12 Constructive Methods

One of the major tasks in the design of a network is the selection of an architecture and its configuration. How many layers should the network have, with how many nodes in each layer? Should every node in a layer connect to every node in the following layer? Obviously the best structure depends on the problem and performance may be poor if the structure is inappropriate. If there are too few units, the final error is likely to be large; training may not converge or may be very sensitive to initial conditions. If there are too many units, training times may be long and generalization may suffer.

Normally we just want to find a structure that works for the given problem. Although this is much easier than determining the theoretically optimum architecture, it may still be very difficult to decide a priori what architecture and size are appropriate for a problem. Some heuristics are available, but there are no dependable general rules for choosing a structure before training. A common ad hoc approach is to experiment with different configurations until one is found that works well. Unfortunately, this may be time consuming if many networks have to be tested before an adequate one is found.

Constructive methods attempt to adapt the size of the network to the problem by starting with a small network and adding layers and units as needed until a solution is found. The major advantage is that there is no need to make an a priori estimate of the correct network size. An unfortunate choice will not immediately condemn the network to failure and the trial-and-error search for a good size is avoided.

Theoretical Support There are good theoretical reasons for considering constructive algorithms. As noted in section 5.7, a common complaint about back-propagation is that it is too slow. The training time also appears to grow quickly as the problem size increases. Judd [204, 202] has shown that loading problem is NP-complete. That is, the training time scales exponentially with network size in the worst case. (The loading problem is the task of finding weight values such that the network imitates the training data, i.e., the task of "loading" the data into the network.) Baum and Rivest [35] showed that the problem is NP-complete even for networks containing as few as three neurons.

These results are an indication of the intrinsic difficulty of the computational problem, independent of the training algorithm. All algorithms, no matter how efficient, that deal with the task as it is posed face the same exponential scaling behavior and become impractical on very large problems. These results make it appear "unlikely that any algorithm which simply varies weights on a net of fixed size and topology can learn in polynomial time" [31: 201]. Baum [31], however, suggests that the difficulty is due to the constraint of a fixed network structure that only allows the algorithm to adjust existing weights. Algorithms with the freedom to add units and weights "can solve in polynomial time any learning problem that can be solved in polynomial time by any algorithm whatever. In this

sense, neural nets are universal learners, capable of learning any learnable class of concepts" [31: 201].

A trivial example is an algorithm that simply allocates a node for every example in the training set to create a network that functions as a lookup table. Of course, more efficient solutions are generally preferred. Several types of non-MLP neural networks, for example, ART and radial basis function networks, can be thought of as adding new units when necessary to fit new data. Most learn much faster than MLPs trained by back-propagation.

Constructive Methods vs. Pruning Methods Constructive methods complement pruning methods (chapter 13), which train a larger-than-necessary network and then remove unneeded elements. Both are means of adapting the network size to the problem at hand. Although pruning methods can be effective, they require an estimate of what size is "larger than necessary." Constructive methods can build a network without this estimate.

Because constructive methods sometimes add more nodes than necessary, it is often useful to follow with a pruning phase. In some algorithms the processes compete simultaneously, one attempting to add nodes while the other tries to remove them. At some point, the processes balance and the structure stabilizes.

When to Stop Adding Units An issue that must be considered with constructive methods is when to stop adding new units. In general, the training-set error can be made as small as desired by adding more units, but the law of diminishing returns predicts that each additional unit will produce less and less benefit. The question is whether the incremental error reduction is worth the cost of the additional units in processing time, storage requirements, and hardware costs. For continuous problems, an infinite number of units might be needed to achieve zero error. One generally must declare some nonzero error to be acceptably small and stop when it is achieved.

Aside from the question of efficiency, there is the problem of overfitting and generalization. Chapter 14 discusses a number of factors affecting generalization. Briefly, the problem is that when training on sampled data (which may contain noise and have other imperfections), the error on the training set is only an estimate of the true error. The two error functions tend to be similar but slightly different so a change that reduces one will not always reduce the other. Usually, they have large-scale similarities with small-scale differences. As the network fits the large scale features of the training-set in the initial stages of training, both errors tend to decrease together as learning progresses. At some point, however, the network starts to fit small-scale features where the two functions differ and additional training starts to have a detrimental effect on the true error. Improvements in the training error no longer correspond to improvements in the generalization error and the network begins to overfit the data. Thus, for good generalization, it is often desirable to stop

training before the training-set error reaches zero. Some implementations avoid the problem by passing it to the pruning algorithm; the constructive phase is allowed to continue well past the point of overfitting and then followed with pruning to satisfy generalization criteria.

Network Size versus Training Time A secondary advantage of constructive algorithms is that overall training times may be shorter because useful learning occurs when the network is still small. That is, even if a small network cannot satisfy the error criteria, it may learn the dominant characteristics of the target function and thereby simplify learning in later stages. With nonconstructive methods, an inadequate network would be abandoned and anything it learned would have to be relearned by the next network tested. With constructive methods, the learning is retained and finer details are picked up as more nodes are added.

There seems to be a trade-off between training time and network size with fast algorithms tending to produce larger, less efficient networks. Many algorithms train the network until the error stops decreasing and then add more units and resume training, repeating until the error is acceptably small. The problem is that plateaus in the error versus time curve are common with back-propagation training of MLP networks. The $E(t)$ error curve often has long flat intervals followed by a sharp drop. In a flat region, it may be difficult to tell if the error has reached its final minimum or if it will decrease further if we let it train longer. A constructive algorithm that does not wait long enough may add unnecessary units; one that waits too long just wastes time. Thus, if one is impatient, the resulting network may be larger than necessary and may not generalize as well as possible. This is another reason for combining constructive and pruning methods.

There are, of course, more sophisticated methods of testing for (near) convergence than thresholding the $E(t)$ slope. Second derivative information in the form of the Hessian matrix (or at least its diagonal elements) may be useful, but will not entirely solve the problem because of the nonlinearity of the problem.

12.1 Dynamic Node Creation

For networks trained by back-propagation and similar methods, the most common procedure is to add new units when the error E reaches a (nonzero) plateau and stops decreasing, presumably because it's in a local minimum. The triggering condition used by Ash [11] (summarized in [277]) is

$$\frac{|E(t) - E(t-\delta)|}{E(t_o)} < \Delta_T \qquad (t \geq t_o + \delta) \tag{12.1}$$

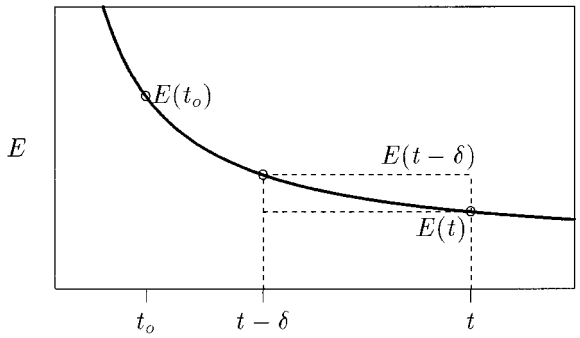

Figure 12.1
In many constructive algorithms, new units are added when the rate of improvement of the training error becomes small. Normalization by the magnitude of the error $E(t_o)$ obtained after the previous unit was added makes the triggering condition less dependent on the size of the error.

where t_o is the time at which the previous new node was added, δ is an interval over which the slope is measured, and Δ_T is the trigger value (figure 12.1). The requirement $(t \geq t_o + \delta)$ allows the network some time to adapt to the addition of the previous node before another is added.

The decision to stop is made by observing the size of the largest errors. The reasoning is that with cost functions like MSE, the error can often be reduced by making small improvements on many easy cases while ignoring large errors on a few difficult cases. As a result, the worst error may increase even as the average error decreases. Both errors tend to jump discontinuously when new units are added. The average error is normally small in later stages of learning so discontinuities in it might not be obvious, but the worst error is usually relatively large and often drops significantly when the critical number of units is reached. (This assumes that the data is consistent, however. If two training patterns with identical inputs have different targets, then no number of units will be able to reduce the worst error to zero.)

Hirose, Yamashita, and Hijiya [170] describe a similar method for adding nodes to a single-hidden-layer network when training stalls. The total error is evaluated periodically, for example, after every 100 weight updates. If it has not decreased significantly, for example, by at least 1%, then a new node is added. Note that these values probably depend on problem size and difficulty. For networks with many weights or problems with many training patterns, it may be necessary to wait longer before deciding that the network is stalled. An algorithm that does not wait long enough could add nodes continuously.

New nodes are added with small random initial weights. This perturbs the solution and usually causes the error to increase. The error normally decreases in subsequent training,

however, which prevents the algorithm from immediately adding another new node in the next cycle. The perturbation could be avoided by initializing the new node with zero-valued weights, but zero-valued weights tend to remain near zero under back-propagation so $E(t)$ would remain flat and a new node would be added at the next opportunity, leading to a proliferation of redundant nodes with very small weights.

Since this procedure can only add nodes, the network may become very large. This is generally undesirable so the training phase is followed by a pruning phase which removes unnecessary nodes. Any reasonable pruning algorithm would probably work; the following method is used by Hirose, Yamashita, and Hijiya [170]. One node is removed at a time and the network is retrained; if it converges, then another hidden node is removed and so on. At some point, the network will not converge after removal of a node; the node is restored and the process halted. In the simulations described, the most recently added hidden nodes were removed first.

12.2 Cascade-Correlation

Fahlman and Lebiere [120] describe the cascade-correlation algorithm. New hidden units are added one at a time with each new unit receiving connections from the external inputs and *all* existing hidden units. Figure 12.2 shows the resulting (nonlayered) structure. Input

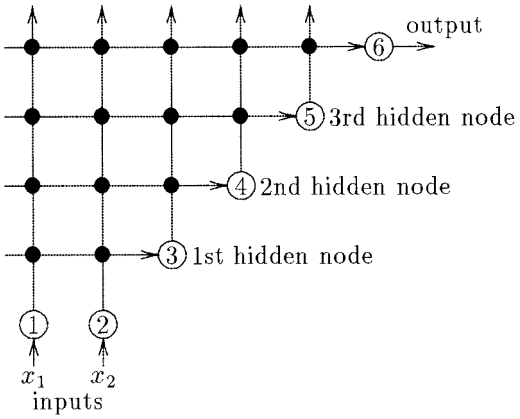

Figure 12.2
Cascade-correlation adds hidden units one at a time. Each new unit receives connections (shown as solid dots) from the external inputs and all existing hidden units. Its input weights are chosen to maximize the covariance of its response with the residual output error. Weights into existing hidden units remain unchanged. Then the weights from all the hidden units (new and old) to the output node are retrained to minimize the error. If the resulting error is acceptable, the network is complete; otherwise a new hidden unit is added and the process repeated.

weights to the new unit are chosen to maximize the correlation (covariance) of its response with the remaining output error. The unit is then added to the network and the weights from all the hidden units (new and old) to the output node are retrained to minimize the error. If the resulting error is acceptable, the network is complete; otherwise a new hidden unit is added and the process repeated.

The Algorithm The algorithm begins with a network with no hidden units; inputs are connected directly to the outputs. The output nodes are usually sigmoidal, but linear nodes might be used for continuous mapping problems. The following steps are repeated until the error is acceptably small:

1. Train the weights of the output node(s) using any appropriate algorithm. Single-layer training rules such as the Widrow-Hoff delta rule or the perceptron learning algorithm may be used. There is no need to back-propagate errors through the hidden units since their input weights are frozen.

2. When no significant error improvement has occurred after a certain number of training cycles, evaluate the error. If it is small enough, stop.

3. Otherwise create a new unit with connections from the inputs and all pre-existing hidden units. Select its weights to maximize S, the magnitude of the covariance of the new unit's response V with the residual error

$$S = \sum_o \left| \sum_p (V_p - \overline{V})(E_{p,o} - \overline{E_o}) \right| \tag{12.2}$$

where o indexes the output units and p indexes the training patterns. The quantities $\overline{V}$ and $\overline{E_o}$ are the averages of V and E_o over all patterns.

S can be maximized iteratively by gradient ascent. The derivative of S with respect to the ith weight is

$$\frac{\partial S}{\partial w_i} = \sum_{p,o} \sigma_o (E_{p,o} - \overline{E_o}) f'_p I_{i,p} \tag{12.3}$$

where σ_o is the sign of the correlation between the candidate's value and the output o, f'_p is the derivative of the candidate unit's activation function with respect to the sum of its inputs for pattern p, and $I_{i,p}$ is the input the unit receives from unit i for pattern p.

When S stops improving, the new unit is added to the network and its input weights are frozen.

4. Go to step 1.

Only the magnitude of the new unit's correlation with the residual error is important, hence the absolute value in the formula for S. If the correlation is positive, a negative output weight can be chosen to decrease the error; if the correlation is negative, a positive output weight will do.

A variation of the algorithm is to allocate a pool of candidate units with random initial input weights. Let each maximize S individually and select the best for addition to the network. This decreases the possibility of adding a useless unit that got stuck during training and it explores more of the weight space simultaneously. Other types of units besides sigmoidal (e.g., radial Gaussian) may also be included in the candidate pool.

When the output weights are being trained, all other weights are frozen. Because the activations of the hidden units depend only on the input pattern and do not change when the output weights change, there is no need to recalculate their response to each input pattern. If sufficient memory is available, the hidden unit responses can be stored in an array for quick retrieval rather than recalculated with each pattern presentation. This can significantly speed up simulations of large networks.

Remarks

- There is no need to guess the best architecture before training. Cascade-correlation builds reasonable, but not optimal, networks automatically.
- Input weights for new hidden nodes are chosen to maximize S, the covariance of the node response with the remaining output error. This is not the same as minimizing the error and won't be optimal in general.
- The procedure generally doesn't find the smallest possible network for a problem and has a tendency to create deep networks so a final pruning phase may be desirable.
- Cascade-correlation learns quickly. Unlike back-propagation, training doesn't slow down dramatically as the number of hidden layers increases because only the output weights are retrained each time. For the problems studied, the learning time in epochs grows approximately as $N \log N$ where N is the number of hidden nodes finally needed to solve the problem [120]. On the two-spirals problem (figure 12.3), an average of just 1700 epochs was needed.
- Although cascade-correlation learns quickly, it can overfit the data [88]. Pruning or early-stopping based on cross-validation may be necessary to avoid overfitting; however, reasonably good generalization as measured by insensitivity to input noise was found in [149]. There, nets created by cascade-correlation tended to have saturated hidden nodes whose values change little when small amounts of noise are added to the inputs.

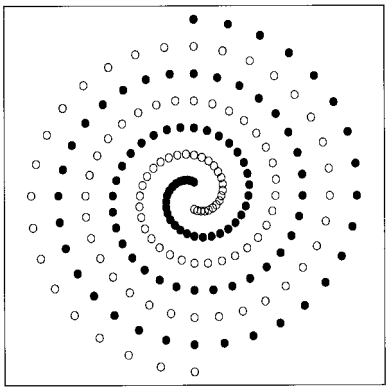

Figure 12.3
The two-spirals problem [233] is sometimes used as a benchmark for constructive algorithms because it requires careful coordination of many hidden nodes and is difficult to learn with simple back-propagation in a MLP network. (It is not representative of most real-world problems, however.) In a single-hidden-layer architecture, 40 or more hidden nodes are generally needed and training times are long. Most successful solutions use more than one hidden layer; some use short-cut connections.

• Hidden-unit input weights are frozen after the unit is added, so features detected by the unit will not be unlearned if the network is retrained with new data. This helps stabilize learning if the training data changes over time.

• "Divide and conquer," a similar method, is described in [322]. Unlike some other algorithms, it can create multiple hidden layers. Unlike cascade-correlation, it can create hidden layers with more than one node and it doesn't use a correlation measure. Like cascade-correlation, nodes are trained one-at-a-time with weights in other nodes held constant. As a result, back-propagation through hidden nodes is never necessary.

• A cascade-correlation architecture for recurrent networks is described by Fahlman [119]. A procedure similar to cascade-correlation, but using error minimization rather than covariance maximization is described by Littmann and Ritter [248]; benchmarks of cascade-correlation are also included.

12.3 The Upstart Algorithm

Frean [128] describes the upstart algorithm for learning binary mappings with layered networks of linear threshold units. It will converge to zero error for any Boolean mapping, including problems that require hidden units. The networks that result are often smaller than those of the tiling algorithm (see later discussion).

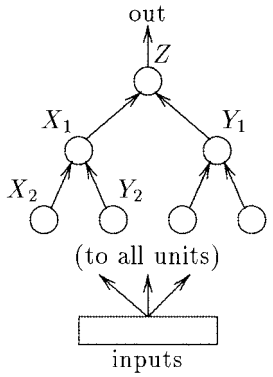

Figure 12.4
The upstart algorithm [128] starts with a single output unit Z. If errors remain after training, then daughter units X and Y are inserted to correct it. Ideally, X corrects Z when it is wrongly ON and Y corrects Z when it is wrongly OFF. If either X or Y cannot correct all the errors assigned to them, additional subunits are introduced to correct their errors and so on. The result is a tree structure for which there is an equivalent single-hidden-layer network.

The idea is that a unit Z can make two types of errors: "wrongly ON" and "wrongly OFF." A wrongly ON error can be corrected by adding a negative connection from a unit X, which is ON only when Z is wrongly ON. Likewise, a wrongly OFF error can be corrected by adding a strong positive connection from a unit Y which is ON only when Z is wrongly OFF.

The algorithm starts with a single output unit Z with weights chosen to separate as many of the training points as possible. If errors remain, daughter units X and Y are created to correct Z when it is wrong. Ideally, X corrects all the wrongly ON errors and Y corrects all the wrongly OFF errors. If either cannot correct all the errors assigned to them, additional subunits are introduced to correct their errors and so forth. Each new unit can always correct at least one error so the number of errors decreases at each step and the process eventually terminates when all patterns are classified correctly. The result is a tree structure (figure 12.4) for which there is an equivalent single-hidden-layer network.

The Algorithm The weights from the inputs to the output Z are trained to minimize the error and then frozen. Perceptron learning [325, 284] can be used, but will not converge to a stable solution if the patterns are not linearly separable. The "pocket" algorithm [133] can also be used. The following steps are then repeated recursively, first for Z, then for each of the daughter units X and Y, then for their daughters, and so on.

1. If Z makes any wrongly ON mistakes, create a new unit X. The targets for X are $\{o_z^\mu \wedge \neg t_z^\mu\}$ where o_z^μ and t_z^μ are the output and target for node Z on pattern μ. (The symbols

$\wedge$ and $\neg$ indicate the logical AND and NOT operations.) That is, X is designed to turn ON when Z is wrongly ON. When Z is ON and its target value is OFF, the target for X is ON; otherwise the target for X is OFF. The patterns for which both Z and the target are OFF can be eliminated from X's training set although this is not necessary.

Similarly, if Z makes any wrongly OFF mistakes, create a new unit Y with targets $\{\neg o_z^\mu \wedge t_z^\mu\}$. The patterns for which Z and the target are both ON can be eliminated from Y's training set.

2. Connect the outputs of X and Y to Z. The weight from X is large-negative and the weight from Y is large-positive. The size of the $X(Y)$ weight needs to exceed the sum of Z's positive (negative) input weights. These weights can be set explicitly, or by an appropriate training procedure.

3. Go to 1 and repeat recursively. That is, correct the errors of X and Y by generating two daughters for each.

The algorithm builds a binary tree of units from the output down to the inputs. The daughter nodes X and Y have an easier problem to solve than does Z. Each new unit can separate at least one of the incorrect patterns so they will always make fewer errors than their parent and will reduce the number of errors made by the parent if connected to it by appropriate weights. Daughter units are created only if the parent makes errors. The number of errors decreases with every branching, so at some stage none of the daughters will make any errors. This means their parents will not either and so on up to the top of the tree. As noted, some of the patterns for X and Y can be eliminated. This is not necessary, but speeds up training by a factor of about two [128].

Frean [128] shows that the resulting tree structure can be converted to an equivalent single-hidden-layer structure if the unnecessary patterns are not eliminated. The original Z unit and all the hidden units are placed in a single layer with the connections between them eliminated and a new output unit is created. The weights from the hidden units to the new output unit can be found by the perceptron algorithm, or by inspection of the tree structure.

12.4 The Tiling Algorithm

Mézard and Nadal [265] describe the tiling algorithm (figure12.5) for constructing multi-layer networks of linear threshold units to solve binary mapping problems. It should also be suitable for mappings from continuous inputs to binary outputs.

Units are added a layer at a time from the inputs upward. The first unit in each hidden layer is called the master unit. It attempts to classify as many of the training patterns as possible based on the input from the previous layer. It is always possible to create a new master unit on the current layer that will make at least one less error than the master unit of

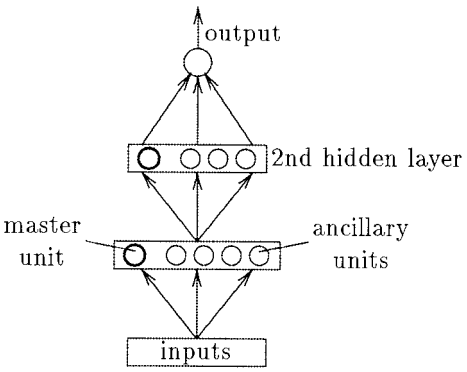

Figure 12.5
The tiling algorithm [265] adds a new hidden layer at each iteration. A master unit in each layer attempts to classify as many patterns as possible based on input from the preceding layer. If it cannot classify all patterns correctly, then 'ancillary' units are added until the layer produces a 'faithful' representation such that any two input patterns with different targets produce a different pattern of activity on the layer. This then serves as input for the next layer. It is always possible to create a master unit which makes fewer errors than the master unit in the previous layer so convergence is guaranteed if enough layers are added.

the preceding layer. Thus, if enough layers are created, the final master unit will not make any errors. Convergence is guaranteed because the number of layers required is limited by the number of patterns to be learned.

If the newly created master unit still makes some errors then additional "ancillary" units are added to the layer until it produces a "faithful representation" of the training patterns such that any two input patterns with distinct targets produce different patterns of activity on the layer.

The Algorithm

1. Create a master unit for the new hidden layer and train it to separate as many of the input patterns as possible.
2. If the new unit produces the correct responses for all patterns, then it is the final output unit. Stop.
3. Otherwise, add ancillary units to create a faithful representation on the current layer.
4. Go to 1.

These steps are explained in more detail in the following.

Generating the Master Unit Assume there are p_o patterns to be learned and that the preceding layer $L-1$ has been established. Layer $L=0$ is taken to be the input layer. At layer L, let $\tau^\mu = (\tau_j^\mu)$, be the vector of activity patterns, or "prototypes," generated

in the preceding layer. $\mu = 1, \ldots, p_{L-1}$ indexes activity patterns and $j = 0, \ldots, N_{L-1}$ indexes nodes in the previous layer ($j = 0$ is the index of a bias node and $j = 1$ is the index of the master unit in the preceding layer). A number V_μ of different input patterns may be represented by the same prototype τ^μ, $\sum_\mu V_\mu = p_o$. Let μ_o be the index of one of the patterns for which the master unit of the preceding layer makes an error, that is $\tau_1^{\mu_o} = -s^{\mu_o}$, where $s^{\mu_o} = \pm 1$ is the desired target.

The set of weights $w_1 = 1$ and $w_{j\neq 1} = \lambda s^{\mu_o} \tau_j^{\mu_o}$, $1/N_{L-1} < \lambda < 1/(N_{L-1} - 2)$ ensures that this master unit will make at least one less error than the master unit in the preceding layer. Let $\lambda = 1/(N_{L-1} - 1)$. When pattern μ_o is again presented, the unit output will be

$$\begin{aligned} m^{\mu_o} &= \operatorname{sgn}\left(\sum_{j=0}^{N_{L-1}} w_j \tau_j^{\mu_o}\right) \\ &= \operatorname{sgn}\left(\tau_1^{\mu_o} + s^{\mu_o} \lambda N_{L-1}\right) \\ &= s^{\mu_o} \quad \text{if } \lambda > 1/N_{L-1} \end{aligned} \tag{12.4}$$

so the pattern μ_o is stabilized. When another pattern μ is presented for which $\tau_1^\mu = s^\mu$ the unit output will be

$$m^\mu = \operatorname{sgn}\left(\tau_1^\mu + s^{\mu_o} \lambda \sum_{j\neq 1, j=0}^{N_{L-1}} \tau_j^{\mu_o} \tau_j^\mu\right). \tag{12.5}$$

If $\mu \neq \mu_o$ then the sum is less than or equal $N_{L-1} - 2$ so if $\lambda < 1/(N_{L-1} - 2)$ then $m^\mu = \operatorname{sgn}(\tau_1^\mu)$ and the classification of pattern μ is preserved. Thus, with this set of weights, the current layer will also stabilize prototype μ_o in addition to all the prototypes μ stabilized in layer $L - 1$,

If the unit is initialized with this set of weights and trained by the pocket algorithm [133] then the final set of weights will be at least as good. If training converges to zero error, then the new unit is the desired output. This happens when the targets are linearly separable in terms of the preceding layer activities. Otherwise ancillary units must be created.

Creating Ancillary Units If the master unit still makes errors, then ancillary units must be created so that the layer generates a unique pattern of activity for all input patterns with different targets. Assume the preceding layer generates $p = p_{L-1}$ distinct prototypes $\tau^\mu = \tau_i^\mu$ where $i = 0, \ldots, N$ and $N = N_{L-1}$. The p prototypes are a faithful representation by construction. Each τ^μ is the prototype of one faithful class of the $(L - 1)$th layer. The current layer must produce a mapping from these p patterns.

Suppose $1 + N'$ units have already been created and they produce p' distinct representations. In general, $p' < p$. If the p' patterns are not a faithful representation, then at least one activity pattern on this layer doesn't have a unique target. One of the unfaithful classes is selected and the next unit is trained to produce the mapping $\tau^\mu \rightarrow s^\mu$ for patterns μ belonging only to the unfaithful class. In the best case, the mapping will be learned perfectly and the unfaithful class will be broken into two faithful classes. Often, however, the mapping will not be linearly separable. In such cases it is possible to break the unfaithful class into two classes—one faithful and one unfaithful. In the worst case, the faithful class may consist of just one prototype.

This is repeated until the layer generates a faithful representation. In practice, if more than one unfaithful class exists, the smallest is selected first. If the new unit is able to separate this class successfully, then the next largest unfaithful class is also attempted with the same unit. As a result, each new unit breaks at least one class into two classes and at most p units are needed to create a faithful representation.

Other Notes

- For multiple output problems, a master unit can be created in each layer for each of the outputs.
- Results of simulations are described by Mézard and Nadal [265] for N-bit parity $N \leq 10$ and random Boolean functions $N \leq 8$. Random Boolean functions with $N = 8$ required an average of 7 hidden layers and about 55 hidden neurons. In comparison, a single-hidden-layer AND-OR network would require on the order of 2^8 hidden units to compute random functions. A typical function would require about 128 units.
- Simulations show that, in general, the number of units per layer decreases with each successive layer.

12.5 Marchand's Algorithm

Marchand, Golea, and Ruján [256] describe a method for constructing a one-hidden-layer network of linear threshold units to solve binary mapping problems (figure 12.6). It is always possible to classify N input patterns by creating N hidden nodes, each of which recognizes one of the patterns. The network would act like a look-up table and the number of nodes needed would grow linearly with the size of the problem. But this fails to capture correlations in the training data and the resulting network does not generalize well. Usually it is better if each hidden unit recognizes as many patterns as possible.

The algorithm described by Marchand, Golea, and Ruján adds hidden units sequentially. The weights of each new unit are chosen to split a group of patterns with like targets from

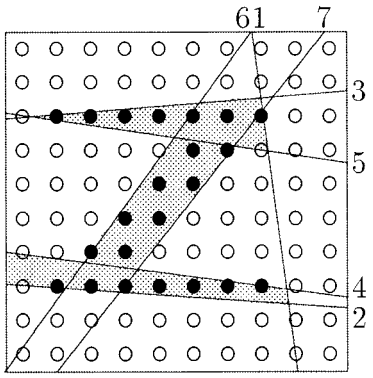

Figure 12.6
The algorithm of Marchand, Golea, and Ruján [256] creates a single-hidden-layer network of linear threshold units by adding hidden units sequentially. Each new unit slices off a group of training patterns that share the same target value. Filled and empty circles represent training points with positive and negative targets. Lines show the hidden unit decision surfaces; the numbers to the side indicate the order in which the hidden units were created. All points are classified correctly, although some are very close to the boundary.

the rest of the data. On one side of the hyperplane defined by the unit, all patterns have the same target value; on the other side, target values may be mixed. The separated patterns are then removed from the working set and the procedure repeated with additional hidden units. Each new hidden unit slices off another set of training patterns that share the same target. The procedure stops when all remaining patterns have the same target.

It is always possible to separate at least one pattern from the rest so in the worst case, no more than $N - 1$ nodes will be needed to recognize N patterns. In practice though, the patterns are often correlated because of regularities in the target function and clustering in the input distribution. Most slices can then remove more than one pattern so fewer than $N - 1$ nodes will be required in general.

The procedure sequentially creates h hidden units, which partition the input space into a number of regions, each containing one or more training patterns that share the same target. The resulting internal representation is linearly separable in that the desired target values can be generated from the hidden unit activities with no need for more hidden layers. The weight u_j from the jth hidden unit to the output unit can be found by the perceptron algorithm or it can be set as

$$u_j = 2^{h-j+1} \quad (\text{for } j = 1, \ldots, h) \tag{12.6}$$

$$u_o = \sum_{i=1}^{h} s_i u_i - s_h \qquad \text{bias weight} \tag{12.7}$$

where $-s_h$ is the target of the $(h+1)$th cluster. These weights increase exponentially; the perceptron algorithm will generally find a different set.

Selection of Hidden Unit Weights To obtain small networks, it is desirable for each hidden unit to slice off as many patterns as possible that share the same target. The following greedy procedure is simple, but not always optimal.

When adding hidden unit k, the working space has N^+ patterns with positive target $+1$ and N^- patterns with target -1. The procedure in [256] tries to find two weight vectors: one that excludes the largest number M^+ of positive patterns and one that excludes the largest number M^- of negative patterns. The vector with the largest ratio M^+/N^+ (or M^-/N^-) is then chosen.

To find the weight vector that maximizes M^+, a pattern ξ^μ with a positive target is chosen and weights are selected so that the unit has a $+1$ output for this pattern and a -1 output for all other patterns, $w_{kj} = \xi_j^\mu$, $j = 1, \ldots N_i$, with bias $w_{ko} = 1 - N_i$. At this point, all -1 patterns and pattern ξ^μ are correctly classified, but some of the other $+1$ patterns may be misclassified. One of the misclassified patterns is chosen and the perceptron rule is used to change the weights to also correctly classify the additional pattern without causing ξ^μ and the -1 patterns to be misclassified. If it succeeds, the change is accepted and the algorithm goes on to try another misclassified pattern. If it fails, another pattern is tried. After all the misclassified $+1$ patterns have been considered, the vector separates a certain number v_1 of the $+1$ patterns, but some are still misclassified (unless the patterns are linearly separable). This current weight vector is saved and all the properly classified $+1$ patterns are removed from the working set. The procedure is then repeated starting with another misclassified pattern to generate another weight vector. All $+1$ patterns, including the ones excluded by the first vector, are considered when computing the number v_2 excluded by the second vector. v_2 and v_1 are compared to choose the better weight vector. This is repeated until the best weight vector is found. A similar procedure maximizes M^-.

Remarks Simulation results for the parity function, random binary functions, and the mirror symmetry problem are discussed by Marchand, Golea, and Ruján [256]. (The target in the symmetry problem is $+1$ if a vector of binary bits is symmetric about its center.) The net found for the symmetry problem was optimal; a similar solution could not be found by the tiling algorithm.

The search for hidden unit weight vectors may take a long time because it searches to find the one vector that excludes the largest number of patterns. The procedure starts with one pattern, say p_{start}, and scans all the other (positive) patterns to see if they are also separable with p_{start}. Then it increments p_{start} to the next pattern. This is a loop over all patterns inside another loop over all patterns so there are $O(M^2)$ steps (where M is

the number of patterns), each of which calls the perceptron learning algorithm. For many problems, however, the running time seems reasonable compared to back-propagation.

12.6 Meiosis Networks

Hanson [153] describes meiosis networks, which work by splitting nodes. (In biology, meiosis refers to a process of cell division.) The algorithm varies the sizes of layers in a given network but does not add new layers. The description in [153] assumes a single-hidden-layer net, but other forms might also be used. In principle, the target function can be either continuous or discrete; the description in [153] presents results for several classification problems.

The optimization procedure is stochastic in that the network weights have noisy values, which change randomly from one instant to the next. The mean and variance for each weight are adjusted during training. The specificity or certainty of a node is estimated by the variance of its weights relative to their means. Nodes with high relative variances are candidates for splitting.

Weight values change randomly from one instant to the next according to a probability distribution such as

$$P\left[w_{ij} = w_{ij}^*\right] \quad = \quad \phi\left(\frac{w_{ij}^* - \mu_{ij}}{\sigma_{ij}}\right) \tag{12.8}$$

where $\phi()$ is an $N(0, 1)$ Gaussian density function. μ_{ij} and σ_{ij} are, respectively, the mean and standard deviation for the fluctuations of weight w_{ij}. Because of this variability, successive presentations of the same pattern can result in different outputs.

The initial network contains one hidden unit whose weights are initialized with random means and variances The mean is adjusted by gradient descent

$$\mu_{ij}(n+1) = -\alpha\frac{\partial E}{\partial w_{ij}^*} + \mu_{ij}(n) \tag{12.9}$$

with a learning rate parameter α. The standard deviation changes depending on the magnitude of the gradient

$$\sigma_{ij}(n+1) = \beta\left|\frac{\partial E}{\partial w_{ij}^*}\right| + \sigma_{ij}(n). \tag{12.10}$$

β is a learning rate parameter. Values $0.1 < \beta < 0.5$ are suggested in [153]. This update mechanism can only increase σ_{ij}. Decreases occur by a decay process

$$\sigma_{ij}(n+1) = \zeta \sigma_{ij}(n), \qquad (\zeta < 1). \tag{12.11}$$

As errors approach zero during training, the standard deviations decay to zero and the network becomes deterministic. Low values of ζ, for example, < 0.7, produce little node splitting; large values, for example, > 0.99, produce continual node splitting. A value of 0.98 was used in simulations.

The standard deviation of a weight is considered to be a measure of its certainty or prediction value; large variances tend to mean low prediction value. This process above tends to assign small variances to weights that converge quickly and high variances to weights that converge slowly. Presumably, quick convergence indicates that the weights are clearly necessary and adequate while slow convergence indicates a delicate balance between opposing forces that the net is unable to resolve quickly. That is, a high variance reflects uncertainty in the proper weight value.

Nodes with many uncertain weights are candidates for splitting. Nodes split when the standard deviation becomes large relative to the mean for both the input and output weight vectors

$$\frac{\sum_i \sigma_{ij}}{\sum_i \mu_{ij}} > 1 \tag{12.12}$$

and

$$\frac{\sum_k \sigma_{jk}}{\sum_k \mu_{jk}} > 1. \tag{12.13}$$

(It may be preferable to use the sum of absolute mean values here.) Child node weights are initialized with the same mean as the parent node and half the variance.

One problem with this splitting criterion is that nodes whose weights have small mean values are more likely to be split than other nodes. A completely unnecessary node whose mean weights are all zero would be split many times.

12.7 Principal Components Node Splitting

A method of node splitting based on detection of oscillation in the weight update directions is described by Wynne-Jones [410]. The idea is that when a network is too small, the weight vectors of hidden units may oscillate between several competing solutions. Oscillation may occur because there are two clusters of data within a class or because a decision boundary is pulled one way by one set of patterns and the other way by another set of patterns. Figure 8.5 illustrates clustering of the weight update directions that could lead to oscillation.

A large amount of weight oscillation is taken as a measure of insufficiency. Nodes whose weights oscillate the most are identified and split in two. The "child" nodes are initialized based on a principal components analysis of the oscillation in the parent node or by examination of the Hessian matrix of the network error with respect to the weights. The Hessian method has the advantage that it can also be applied to the input nodes to determine their relative importance.

This is usually better than initializing child nodes with random weights because it uses the information in the existing weights and usually causes less perturbation in the error. In high dimensional spaces, random weights are unlikely to be well-placed initially and considerable learning may be needed to move them to where they are useful.

Splitting The network is allowed to train until it stops making progress. The weights are then frozen and the training set presented again to evaluate oscillation by computing the principal components of the covariance matrix of weight updates,

$$C = \sum_p \delta \mathbf{w}_p^T \delta \mathbf{w}_p. \tag{12.14}$$

C is the outer product of the weight updates $\delta \mathbf{w}_p = \partial E_p / \partial \mathbf{w}$ summed over the patterns p. The mean of $\delta \mathbf{w}$ is assumed to be zero. The largest eigenvalue and corresponding eigenvector of C give the magnitude of the oscillation and its direction. The node is split into two and the child nodes are initialized with weight vectors one standard deviation on either side of the parent vector along the direction of oscillation. This usually results in minimal perturbation of the existing solution but gives enough separation to break symmetry and allow the child nodes to converge to different solutions. Since computation of eigenvectors can be computationally expensive, more practical iterative techniques are mentioned by Wynne-Jones [410]. After splitting, the weights are unfrozen and training resumed.

Selecting Nodes for Splitting The nodes most likely to benefit from splitting are those in which there are very pronounced directions of weight oscillation. Nodes can be compared on this basis by computing the ratio of the largest eigenvalue over the sum of the eigenvalues. This will be highest for nodes with a single dominant direction of oscillation. In high dimensional spaces, however, a node may have several directions with significant eigenvalues (in which case the node could be split along each direction). The ratio will be lower in this case so the ratio technique would not split these nodes until there are no other options. An alternative is to calculate the second derivative of the error with respect to a normalized parameter such as the node gating parameter α described by Mozer and Smolensky [275] (summarized in section 13.2.1). A high curvature of the error with respect to α_i indicates the error is sensitive to the weights of node i. The node is a good candidate for splitting if

the curvature of the error in weight space has a dominant direction as indicated by eigenvalues of the Hessian of $E(\mathbf{w})$. Nodes with a small second derivatives of the error with respect to α, on the other hand, have little differential effect on the error and are candidates for pruning. The same process can be used to estimate the sensitivity of the error to the presence or absence of input variables.

Backsliding A potential problem with this method is that the child nodes often revert back to the position of the parent node because of the global properties of the sigmoid activations. That is, a node that makes a strong contribution to part of the global decision boundary may be influenced by training patterns that are far from the boundary. If the node is split, the child nodes may feel similar influences from the distant patterns and choose the same solution, leaving the global boundary unchanged. Node splitting may be more successful, therefore, in local networks such as radial basis functions than in MLP networks. In this case, oscillation in the Gaussian centers is detected.

12.8 Construction from a Voronoi Diagram

A constructive method based on Voronoi tessellation of the training data is described by Bose and others [55, 53, 54]. A similar method is described by Murphy [279].

The Voronoi tessellation is related to the familiar nearest-neighbor-classifier partition. Figure 12.7 illustrates a two-dimensional example, but the principle applies in higher

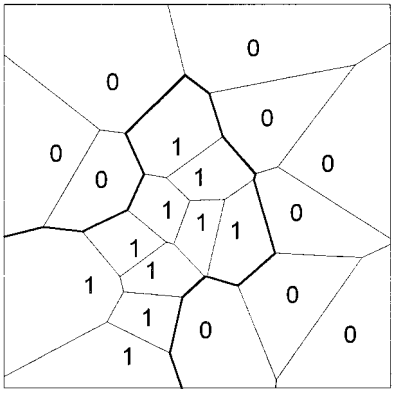

Figure 12.7
Constructive methods can be based on a Voronoi tessellation of the training points. The Voronoi diagram of a set of "base" points partitions the space into cells depending on which base point is closest. Each cell is a convex region bounded by hyperplanes. A layered network can be constructed which forms the same partition and generates the required outputs in each cell.

dimensions as well. Given a set of base points, the surrounding space is partitioned into regions, or cells, depending on which base point is closest. With a Euclidean metric, the resulting cells are convex regions bounded by hyperplanes. (Other tessellations using different metrics and different criteria are possible; this is the most common variation.) There are efficient algorithms for obtaining the partition in high dimensions. Most are based on its dual, the Delaunay tessellation.

Given a Voronoi diagram of the training points, a layered network can be generated to form the same partition and produce the required outputs in each cell. As noted in section 4.1, a network with two hidden layers would be sufficient. Nodes in the first hidden layer would implement the partition hyperplanes, nodes in the second hidden layer would combine these into the convex cell partitions, and the output node would combine these to generate the required output for each cell. It isn't necessary to implement every partition hyperplane, however, because many don't separate points with different labels.

The algorithm described by Bose and colleagues [55, 53, 54] automatically constructs a network to fit a given set of training data. It chooses the number of layers, number of nodes in each layer, and sets appropriate weights to realize the mapping. The layers are only partially connected in general. The process is completely automatic, so repetitive trial and error experiments with different structures and different training parameters are avoided, as well as long training times and uncertainties about convergence to local minima. The algorithm is rather involved, however, and will not be described here. Details can be found in [55, 53, 54].

Remarks This method designs networks for classification problems with binary or discrete-class targets. Generalization issues are not addressed; with consistent data, the resulting network will classify every training point correctly.

Because the design is based on a nearest-neighbor classifier, the resulting network has properties like a nearest-neighbor classifier. That is, accuracy can be good when training data are abundant, but may be poor when data are sparse. Class boundaries may be rather jagged in some cases. Advantages over a naive nearest-neighbor classifier are economy (it does not store every training point) and evaluation speed (it does not slowly search through every training point to classify a new input).

In general, only hyperplanes that separate differently labeled cells need to be realized by hidden nodes. In some cases, a single hidden node can fill in for several hyperplanes of the Voronoi diagram so the resulting network can be relatively small. The network may not be minimal though because the algorithm is not always smart enough to see when a single hidden node could do the job of several partition hyperplanes. In figure 12.7, for example, there are 17 planes separating the 0s and 1s, but as few as three or four hidden nodes would probably be enough.

12.9 Other Algorithms

The preceding sections list only a few of the many algorithms that have been proposed. The list is not exhaustive by any means so we encourage the reader to explore further for a more complete survey. Genetic algorithms, for example, have been proposed to both generate the network structure and find the appropriate weights. Some of the weight initialization techniques mentioned in chapter 7 construct networks based on solutions provided by other method, for example, decision trees (section 7.2.5) or rule-based knowledge (section 7.2.6). A polynomial time algorithm using clustering and linear programming techniques to generate classifier networks is described in [276]. Projection pursuit regression [129, 185], a well-known statistical procedure, creates a system similar to a single-hidden-layer network with a linear output node. It is constructive in the sense that it adds projection directions (corresponding to hidden units) sequentially until the error is sufficiently small.