7 Weight-Initialization Techniques

The following sections summarize some techniques for initializing weights in sigmoidal networks. The basic motivation is to speed up learning by choosing better initial solutions. A survey and empirical comparison of a number of techniques is given by Thimm and Fiesler [368].

There are two clusters of methods. One consists of methods for choosing parameters controlling the distribution of random initial weights. The motivation here is to avoid sigmoid saturation problems that cause slow training. Most of these methods do not use domain-specific information. The other cluster consists of techniques for initializing the system from an approximate solution found by another modeling system; common choices include rule-based systems, decision trees, or nearest-neighbor classifiers. The motivation here is to reduce training times and probability of convergence to poor minima by starting the system near a good solution. An advantage of these methods, besides faster training, is that they provide ways to embed domain-dependent information in a network.

7.1 Random Initialization

The normal initialization procedure is to set weights to "small" random values. The randomness is intended to break symmetry while small weights are chosen to avoid immediate saturation.

Symmetry breaking is needed to make nodes compute different functions. If all nodes had identical weight vectors, then all nodes in a layer would respond identically and the layer would function as if it contained just one node. Likewise, each node would receive identical error information during back-propagation so weight changes would be identical and the weights would never have a chance to become different.

Small weights are needed to avoid immediate saturation because large weights could amplify a moderate input to produce an extremely large weighted sum at the inputs of the next layer. This would push the nodes well into the flat regions of their nonlinearities and learning would be very slow because of the small derivatives [241]. The weights should not be too small, however, because learning speed would then be limited by the small δ values back-propagated through the weights. Another factor is that the origin is a saddle point on many error surfaces.

Typically, weights are randomly selected from a range such as $(-A/\sqrt{N}, A/\sqrt{N})$, where N is the number of inputs (fan-in) to the node and A is a constant between 2 and 3. Division by the fan-in compensates for the increase in variance of the weighted-input sum with the number of inputs; without it, the sum could sometimes be large for large N

and the node would saturate often. The range $(-2.4/N, 2.4/N)$, where N is the number of node inputs, is another commonly cited choice [236].

Suppose the weights are selected from a range $[-w_o, +w_o]$. Many studies, for example [121, 235, 240, 393, 110, 241, 321, 368], have observed that an intermediate set of w_o values work best. Extreme values either do not converge or converge to poor solutions. Very small initial weights make it hard to escape from the $\mathbf{w} = \mathbf{0}$ weight vector, which is a poor local minimum or saddle point in many problems. With large w_o values, on the other hand, many nodes saturate, derivatives are small, and the net does not converge to a good solution in a reasonable amount of time. Within the range of values that work, the exact value is usually not critical. Thimm and Fiesler [368] suggest that it is better to choose too small a value rather than one that is too big because performance deteriorates very quickly once the upper threshold is crossed.

7.1.1 Calculation of the Initial Weight Range

One basis for selecting an initial weight distribution is to assume the inputs have some statistical distribution and select the initial weight distribution so that the probability of saturating the node nonlinearity is small.

Assume that the weights are independent of the inputs

$$E\left[w_i x_i\right] = E\left[w_i\right] E\left[x_i\right] \tag{7.1}$$

and that the weights are zero-mean, independent, and identically distributed

$$E\left[w_i\right] = 0 \quad \text{for all } i, \tag{7.2}$$

$$E\left[w_i w_j\right] = \sigma_w^2 \delta_{ij} \tag{7.3}$$

where $\delta_{ij} = 1$ if $i = j$ and 0 otherwise.

The weighted sum into a node with N inputs is

$$u = \sum_{i=1}^{N} w_i x_i \tag{7.4}$$

and by independence of w and x the expected value is

$$E\left[u\right] = \sum_{i=1}^{N} E\left[w_i\right] E\left[x_i\right] = 0. \tag{7.5}$$

Because the expected value is 0, the variance is then

$$\begin{aligned} E\left[u^2\right] &= E\left[\left(\sum_{i=1}^{N} w_i x_i\right)^2\right] \\ &= \sum_{i,j=1}^{N} E\left[w_i w_j x_i x_j\right] \\ &= \sum_{i,j=1}^{N} E\left[w_i w_j\right] E\left[x_i x_j\right] \\ &= \sum_{i,j=1}^{N} \sigma_w^2 \delta_{ij} E\left[x_i x_j\right] \\ &= \sigma_w^2 \sum_{i=1}^{N} E\left[x_i^2\right]. \end{aligned}$$

Note that this does not require that the inputs be independent. Independence of w_i and w_j suppresses the effect of correlations between x_i and x_j on $E\left[u^2\right]$. If the inputs are zero-mean, identically distributed so $E\left[x_i^2\right] = \sigma_x^2$, then

$$\sigma_u^2 = N\sigma_w^2\sigma_x^2 \tag{7.6}$$

and

$$\sigma_u = \sigma_x\sigma_w\sqrt{N}. \tag{7.7}$$

For $y = \tanh(u)$ nonlinearities, the input u needed to produce an output y is

$$u = \ln\frac{1+y}{1-y}. \tag{7.8}$$

Let us say the node is saturated for $|y| > 0.9$ so $u_{sat} = \ln 19$. For sigmoid nodes, the constant is the same if saturation is taken to occur at $\text{sigmoid}(u_{sat}) = 0.95$.

We want the probability that $u > u_{sat}$ to be small. This can be achieved by selecting the initial weight distribution so u_{sat} is several times σ_u. With, for example,

$$u_{sat} > 2\sigma_u \tag{7.9}$$

the probability that a given node will be saturated will be about 5%. This assumes a Gaussian distribution for u, which is reasonable when N is large because of the central-limit theorem.

Uniform Weights For weights initialized from a uniform distribution over the interval $[-w_o, +w_o]$,

$$\sigma_w = w_o/\sqrt{3} \tag{7.10}$$

$$\sigma_u = w_o\sqrt{N/3}\sigma_x \tag{7.11}$$

$$u_{sat} > 2w_o\sqrt{N/3}\sigma_x \tag{7.12}$$

$$w_o < \frac{u_{sat}}{2\sigma_x}\sqrt{\frac{3}{N}}. \tag{7.13}$$

For bipolar inputs $x_i \in \{-1, +1\}$ with equal probability for either value, $\sigma_x = 1$. Then, for $u_{sat} = \ln 19$

$$w_o < 2.55/\sqrt{N}. \tag{7.14}$$

For bipolar inputs with probability $P\ [x = 1] = p$, $\sigma_x = 2\sqrt{p(1-p)}$ and

$$w_o < 1.28/\sqrt{Np(1-p)}. \tag{7.15}$$

For binary inputs $x_i \in \{0, 1\}$ with probability $P\ [x = 1] = p$, $\sigma_x = \sqrt{p(1-p)}$ and

$$w_o < 2.55/\sqrt{Np(1-p)} \tag{7.16}$$

and

$$w_o < 5.1/\sqrt{N} \quad (\text{for } p = 1/2). \tag{7.17}$$

For uniform inputs in the range $[-a, +a]$, $\sigma_x = a/\sqrt{3}$ and

$$w_o < 4.4/(a\sqrt{N}). \tag{7.18}$$

For Gaussian $N(0, \sigma_x)$ inputs,

$$w_o < 2.55/(\sigma_x\sqrt{N}). \tag{7.19}$$

Gaussian Weights Similarly, for weights initialized from a Gaussian $N(0, \sigma_w)$ distribution,

$$\sigma_u = \sigma_w \sigma_x / \sqrt{N} \tag{7.20}$$

$$u_{sat} > 2\sigma_w \sigma_x / \sqrt{N} \tag{7.21}$$

$$\sigma_w < \frac{u_{sat}}{2\sigma_x \sqrt{N}}. \tag{7.22}$$

For $u_{sat} = \ln 19$,

$$\sigma_w < 1.47/(\sigma_x \sqrt{N}). \tag{7.23}$$

For bipolar inputs $x_i \in \{-1, +1\}$ with equal probability for either value, $\sigma_x = 1$ and

$$\sigma_w < 1.47/\sqrt{N}. \tag{7.24}$$

For bipolar inputs with probability $P\ [x = 1] = p$, $\sigma_x = 2\sqrt{p(1-p)}$ and

$$\sigma_w < 0.74/\sqrt{Np(1-p)}. \tag{7.25}$$

For binary inputs $x_i \in \{0, 1\}$ with probability $P\ [x = 1] = p$, $\sigma_x = \sqrt{p(1-p)}$ and

$$\sigma_w < 1.47/\sqrt{Np(1-p)} \tag{7.26}$$

and

$$\sigma_w < 2.94/\sqrt{N} \quad (\text{for } p = 1/2). \tag{7.27}$$

For uniform inputs in the range $[-a, +a]$, $\sigma_x = a/\sqrt{3}$ and

$$\sigma_w < 2.54/(a\sqrt{N}). \tag{7.28}$$

For multilayer networks, the inputs in the derivation above could actually be the outputs of a preceding layer. Because they have different statistical properties from the overall system inputs, each layer of weights will have a different ideal initialization range according to this approach. For large fan-ins, the weighted sum u into a node usually approaches a Gaussian distribution. If the initial weights into the node are chosen to avoid saturation, the distribution of the node outputs will also be approximately Gaussian but with a standard deviation multiplied by the slope s of the nonlinearity, $\sigma_{out} = s\sigma_u$. Similar adjustments are appropriate for nodes receiving weights from several different layers, as might happen in networks containing "leap-frog" weights.

It should be noted that the derivation depends on assumptions about the input distribution, which may not apply in a particular problem. In many problems, inputs will be

Table 7.1
Weight Initialization Parameters.

Input distribution	*Weight distribution*	
	Uniform($-w_o$,$+w_o$)	Gaussian N(0,σ_w)
Bipolar $\{-1, 1\}$	$w_o < 1.28/\sqrt{Np(1-p)}$	$\sigma_w < 0.74/\sqrt{Np(1-p)}$
for $p = 1/2$	$w_o < 2.55/\sqrt{N}$	$\sigma_w < 1.47/\sqrt{N}$
Binary $\{0, 1\}$	$w_o < 2.55/\sqrt{Np(1-p)}$	$\sigma_w < 1.47/\sqrt{Np(1-p)}$
for $p = 1/2$	$w_o < 5.1/\sqrt{N}$	$\sigma_w < 2.94/\sqrt{N}$
Uniform($-a$,a)	$w_o < 4.4/(a\sqrt{N})$	$\sigma_w < 2.54/(a\sqrt{N})$
Gaussian N(0,σ_x)	$w_o < 2.55/(\sigma_x\sqrt{N})$	$\sigma_w < 1.47/(\sigma_x\sqrt{N})$

clustered and the large fan-in assumption may not be valid for small networks. An alternative to relying on possibly invalid assumptions is to calculate an appropriate range numerically using the same basic procedure. The calculations are relatively simple and fast in most cases.

The objective of this initialization method is to minimize the probability that nodes will be saturated in the early stages of training. A potential problem, pointed out by Wessels and Barnard [393] (see section 7.1.4), is that this makes all nodes sensitive to all the training patterns so the decision boundaries of many nodes may move large distances before settling to a stable state. In some cases, hyperplanes may move completely out of the region occupied by the training data and produce stray nodes that contribute little useful information to the rest of the net. They suggest that occasional saturation helps to "pin the hyperplanes to the data." Because its derivatives are small when saturated, each node will be most sensitive to patterns near its hyperplane and relatively insensitive to more distant patterns. Initially at least, each node would be loosely specialized by sensitization to a different fraction of the data.

7.1.2 Initialization to Maximize BP Deltas

The derivation of section 7.1.1 provides criteria for selecting a weight initialization range for the input-to-hidden weights. The initialization range of the hidden-to-output weights can be selected in order to maximize the expected magnitude of the back-propagated deltas at the hidden nodes [321]. The expected magnitude of the back-propagated error is an increasing function of the weight range for small weight ranges. (If all the weights were zero, the back-propagated error would be zero.) But for large weight ranges, the output nodes saturate often so the back-propagated deltas are small. Rojas [321] reports that w_o values between 0.5 and 1.5 give similar results in empirical tests.

7.1.3 Initialization of Bias Weights

It was noted in section 3.1 that the distance d of a node's hyperplane from the origin is controlled by the bias weight

$$d = w_{bias}/\|\mathbf{w}\| \tag{7.29}$$

where $\mathbf{w}$ is the weight vector excluding the bias weight. When weights are initialized randomly, d will sometimes be large and the hyperplane may be far from the region containing the inputs. A remedy is to choose $w_{bias} < \|\mathbf{w}\|$ so the initial hyperplane always intersects the unit hypercube around the origin. This idea is mentioned in Palubinskas [294] among other places.

7.1.4 Constrained Random Initialization I

Some heuristics for setting the initial weights in order to decrease the chance of the network becoming trapped in a local minimum are discussed by Wessels and Barnard [393]. The following types of irregularities are defined:

Type 1. Stray hidden nodes whose decision boundaries have drifted out of the region of the input space sampled by the examples. These nodes have nearly constant activation for all training inputs and contribute little useful information.

Type 2. Hidden nodes duplicating function due to failure of symmetry breaking.

Type 3. Hidden node configurations that result in all nodes being inactive in some regions of the input space, making the network insensitive to inputs there.

Type 1 and 2 errors are common with random initialization. As an alternative, initial weights can be chosen systematically so that the following occur [393]:

• The decision boundary of every hidden node crosses the region covered by the samples (to avoid type 1 errors).

• The decision boundaries have a wide range of different orientations (to avoid type 2 errors).

• The transition region of each hidden node covers about 20% of the input space (to avoid type 2 errors). When weights are initialized with small values, the nodes tend to compute nearly linear functions. Because the sum of two linear functions is also linear, the net might use two nodes to do what could be done by one. Initializing with weights large enough to make the node functions somewhat nonlinear helps to avoid this.

• Every part of the sampled region has at least one active hidden node (to avoid type 3 errors).

The initial hidden-to-output weights are set to the same small value, for example, 0.25. Random hidden-to-output weights are not required because symmetry is broken by the way the input-to-hidden weights are initialized. It might even be counterproductive if it masks the activities carefully set up in initializing the hidden node weights. Because the error back-propagated to a hidden node is proportional to its output weights, setting the weights equal makes each hidden node equally responsive to all the outputs. Very small values cause slow learning while values larger than 1 tend to cause saturation because large δ values are propagated back from the output nodes. The 0.25 value was based on empirical tests. Performance was said to be relatively insensitive to the exact value as long as it was not large enough to cause saturation (in which case performance dropped off drastically).

7.1.5 Constrained Random Initialization II

A similar method is described by Nguyen and Widrow [283]. Weight vectors are chosen with random directions, magnitudes are adjusted so each node is linear over a fraction of the input space with some overlap of linear regions between nodes with similar directions, and thresholds are set so the hyperplanes have random distances from the origin within the region occupied by the input data.

The following recipe gives similar results. Let $\mathbf{w}$ represent the weight vector excluding the bias and let θ denote the bias weight. The weighted sum into a node is $u = \mathbf{w}^T\mathbf{x} + \theta$.

1. First, set the weights so each vector has a random direction. A Gaussian or other spherically symmetric distribution should be used because this makes all directions equally likely; a uniform distribution tends to favor directions pointing to corners of the hypercube.

$\mathbf{w} \sim N(0, 1)$

2. Adjust the magnitude of $\mathbf{w}$ so the linear region covers a fraction of the input space. The best width for the linear region depends on the number of hidden nodes; with fewer hidden nodes, the linear region has to be wider so every point in the input space is covered by the linear region of some node.

The linear region of a sigmoid-like node roughly covers the region from ($-u_{sat}$ to $+u_{sat}$). For tanh nodes, $u_{sat} = \ln 19 = 2.94$. If the inputs lie in the interior of the unit hypersphere, the maximum weighted sum occurs when $\mathbf{x} = \mathbf{w}$

$$u_{max} = \|\mathbf{w}\|^2. \tag{7.30}$$

To make the linear region approximately 0.4 long (1/5 of the diameter of the input space), this should be about 5 times u_{sat}

$$u_{max} = 5u_{sat}. \tag{7.31}$$

Normalization of $\mathbf{w}$ to a magnitude $\|\mathbf{w}\| = \sqrt{5u_{sat}} = 3.84$ gives this result

$$\mathbf{w} \leftarrow 3.84 \frac{\mathbf{w}}{\|\mathbf{w}\|}.$$

3. Set the threshold so the distance of the hyperplane from the origin has a random distribution between 0 and 1 (again assuming inputs lie in the unit hypersphere). The distance of the hyperplane from the origin is $d = \theta / \|\mathbf{w}\|$ so choose

$$\theta = \|\mathbf{w}\| \tau \tag{7.32}$$

where τ is a random number between 0 and 1.

7.1.6 Remarks

• Many random weight initialization methods attempt to specify an appropriate range of initial weights. The equivalence between scaling weights by a constant factor and introducing a gain term in the sigmoid function means that similar results can be obtained by gain-scaling (section 8.7).

• There is some suggestion that on-line learning can tolerate saturation problems caused by large initial weights better than batch learning [241].

• In [368], no significant difference was found between uniform, normal, and unbalanced uniform distributions for initializing higher-order perceptrons. Empirical tests favored the method of section 7.1.4, but other methods gave similar results.

• The effects of initial weights on convergence time are examined by Kolen and Pollack [216, 217]. Plots of convergence time *vs.* initial weights (displayed as two-dimensional slices through the weight space) show fractal structure. Convergence regions are separated from nonconvergence regions by complex borders and certain mappings cannot be learned from initial weights in certain regions of the weight space.

7.2 Nonrandom Initialization

An alternative to random initialization is to base the initial state of the system on an approximate solution provided by another type of classifier or fitting system. The goal is to start the system at a reasonably good solution which can then be fine tuned by normal training methods. If things work as planned, the initial state will be in the basin of attraction of a good minimum so global search capability and convergence time are not critical issues and relatively simple training methods may therefore suffice. (However, techniques such as quasi-Newton methods or conjugate gradients may be preferred over back-propagation because final tuning is where they really outperform simpler methods.)

In many cases, the initialization time will be negligible compared to training times for randomly initialized networks. Even when the time is not negligible, it may be regained later because (1) the actual training time may be much shorter if the initial solution is good and (2) problems of convergence to poor local minima may be avoided because the system is already at an approximate solution when it starts. Often, only one fine-tuning run will be needed. With random initialization, in contrast, it is usually necessary to train a number of networks with different starting weights because of the possibility of poor local minima and the total training time for all networks should be considered when making comparisons.

Another reason for using nonrandom initializations is to embed domain-dependent information in a network. In many applications there is significant human knowledge that might be useful even though it might be incomplete or only partly reliable. Existing techniques may give reasonable but imperfect solutions or we may have partial knowledge about the form the solution should take. Pure training on examples in an unstructured network does not provide an efficient way to use this information. When training data are sparse, the external bias supplied by the initial solution may be a major factor in obtaining a solution that generalizes well.

7.2.1 Principal Components Initialization

Principal components analysis (PCA) attempts to identify the major axes of variation of a data set—the directions along which the data varies the most. The principal components of a data set are determined from the eigenvectors of the covariance matrix

$$\mathbf{R} = E\left[\mathbf{x}_k \mathbf{x}_k^T\right].$$

The eigenvalues measure the variance of the projection of the data onto the corresponding eigenvectors. The principal eigenvector (having the largest eigenvalue) indicates the direction along which the data varies the most; lesser eigenvectors (having smaller eigenvalues) indicate directions with lesser amounts of variation. PCA is discussed further in appendix B.

Going on the assumption that these are important directions for the function to be learned, one can initialize the input-to-hidden weight vectors along these directions. In a network with H hidden nodes, the H eigenvectors with the largest eigenvalues would be selected. A two-phase algorithm is described in [136]. In phase one, the input-to-hidden weights are fixed to the principal component directions while the output weights are trained. In the second phase, all weights are allowed to learn. The authors report a large drop in error after the first few iterations of the second phase and an overall speed up in learning times.

7.2.2 Discriminant Analysis Initialization

As an unsupervised method, principal components analysis ignores classification information when choosing its directions. Discriminant analysis, a related linear projection method, uses class information and so may yield a better set of directions for classification purposes (see section B.2).

7.2.3 Nearest Neighbor Classifier Initialization

A nearest neighbor classifier that performs well on a classification problem can be used to initialize a layered network solution. The following method is described by Smyth [349]. Similar ideas are suggested by Raudys and Skurikhina [303] and others.

The process starts with a set of cluster centers for each class. The k-means algorithm, or a similar method such as LVQ, may be used to select representatives. A bisecting hyperplane is then created for every pair of centers with different class memberships. For n cluster centers, there will be $n(n-1)/2$ hyperplanes; a culling algorithm selects the fraction of these that actually separate adjacent centers. Each remaining hyperplane determines the weight vector for a node in the hidden layer.

Next, the cluster centers are applied as inputs to calculate the hidden-to-output weights. The hidden unit responses are treated as fixed in this phase so the hidden-to-output weights can be found by solving a system of linear equations (e.g., by a pseudoinverse solution). The response can then be fine tuned by normal back-propagation training.

It was noted that the method does not work for the XOR problem because of limitations in the algorithm used to cull redundant hyperplanes [349]. This is a limitation in the specific culling algorithm, however, not a fundamental problem.

The algorithm may allocate more hidden units than are actually needed because it ignores the fact that hyperplanes can often be shared to separate more than one pair of clusters. A post-training pruning step might address this problem.

Nearest neighbor initialization methods should work well if a nearest neighbor classifier performs well on the classification problem. There is a question of how many cluster centers to choose; the choice is presumably based on the performance of the nearest neighbor classifier. It should be noted that simple unsupervised clustering procedures such as the k-means algorithm use only the input patterns when selecting cluster centroids; the output classifications are ignored. When the data are naturally clustered in the input space and these clusters correspond to output classifications, unsupervised clustering methods may select an adequate set of centroids. If not, then other centroid selection methods will be required.

Also, nearest neighbor classifiers often require large data sets to achieve good accuracy so random initialization may give better results when data are limited. In an empirical test,

randomly initialized nets generalized better in the sense that the difference between the training and test set errors was smaller. Because the nearest-neighbor initialization starts the network at reasonably good solution, it may be easier to overtrain.

7.2.4 Prototype Initialization

Another nearest-neighbor initialization is discussed by Denoeux and Lengellé [105]. There is one hidden node for each prototype. Input patterns are normalized so each input vector has unit length, $\|\mathbf{x}\| = 1$. This can be done without loss of information by increasing the input dimension by 1 to encode the (suitably scaled) magnitude. When the weight vectors are also unit length, $\|\mathbf{w}\| = 1$, then $\mathbf{w}^T\mathbf{x} = 2 - \|\mathbf{w} - \mathbf{x}\|^2$ and a sigmoid unit has properties similar to the Gaussian units normally used in radial basis functions. (The normalized inputs lie on surface of an n-dimensional sphere which is divided into two sections by the hyperplane defined by $\mathbf{w}$.) The hidden units thus have localized responses in the input space and can be viewed as recognizing prototype patterns.

When gains are high, the hidden layer response approximates a nearest neighbor selector. That is, for nonborderline cases, the hidden unit whose weight vector is closest to the input pattern will respond strongly while more distant hidden units respond weakly, if at all. The response at the hidden layer thus approximates a 1-of-N representation so hidden-to-output weights are independent and can be set directly from the value of the target function at the prototype location. If hidden unit j responds to the jth prototype p_j, for example, then hidden-to-output weight $w_j = f^{-1}(t_j)$ where f is the node nonlinearity function and t_j is the target value for the prototype. When prototype p_j is presented, hidden unit j outputs a 1, the other hidden unit outputs are approximately 0, and the output is $f(w_j) \approx t_j$.

Several methods are suggested for choosing prototypes. For continuous target functions, the best prototypes are at local extremes (peaks and dips) of the function. An algorithm producing prototypes at these locations is described.

When gains are relaxed, the network interpolates between the prototypes more smoothly so the method can be used for continuous regression problems. As in the method of section 7.2.3 there is a question of how many prototypes to choose.

Successful learning of the two-spirals problem (illustrated in figure 12.3) in about 1000 epochs was demonstrated in a 3/20/1 network. (The easy success may be attributable to the change in input representation, however, since the normalization provides a simple function of the radius r as an additional input. This makes the problem simple enough to be solved easily by a randomly initialized network.)

7.2.5 Initialization from a Decision Tree Classifier

Initialization from a decision tree is considered by Sethi [340]. A decision tree classifier performs a hierarchical series of tests on an input pattern to determine its classification.

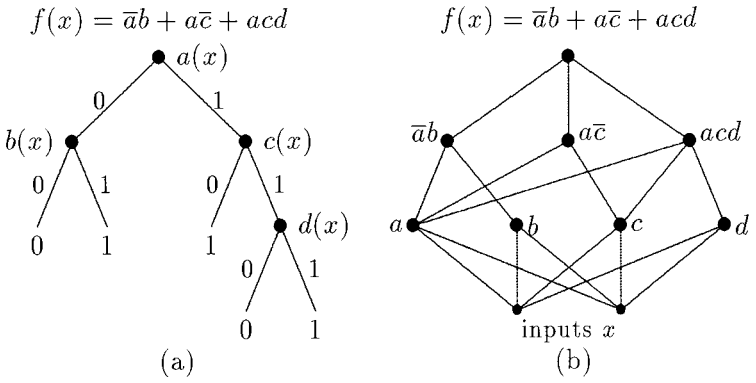

Figure 7.1
Initialization from a decision tree. The function computed by the decision tree can be duplicated by a network of linear threshold units [340]. The tree computes a Boolean function of the propositions evaluated at its decision nodes so a two-hidden-layer network is sufficient if the tree's node decisions can be computed by linear threshold elements. The network can be built by inspection of the tree. Training is not required; (a) is a decision tree and (b) the equivalent network.

This can be represented as a tree (figure 7.1). Each node represents a test that can have one of several outcomes. The result determines which outgoing path is taken and which test is performed next. The path terminates in a leaf node when the result is an unambiguous classification. For a given input pattern, evaluation starts at the root node and follows a path to the leaf node with the appropriate classification.

Since a decision tree computes a Boolean function of the propositions evaluated at its decision nodes, its function can be duplicated by a network of linear threshold units [340]. A two-hidden-layer network is sufficient if the tree's node decisions can be computed by linear threshold elements. The network can be built by inspection of the given decision tree (see [340] for details). Decision trees can usually be constructed quickly, so the overall time needed to design a tree classifier, construct the equivalent network, and fine tune it by training on examples may be much shorter than the time needed to train a network from random weights.

7.2.6 Initialization from Symbolic Rule Systems

Another method of initialization constructs a network from a rule-based solution. In [342, 375] symbolic rules are embedded in the initial structure of a neural network by translating the AND, OR, and NOT terms into corresponding network structures with appropriate weights. (See figure 4.7 for implementation of AND and OR functions.) Note that this may produce an irregular nonlayered network because the rules in the original system can have

arbitrary dependencies. Additional links with small random weights are provided to let the system add new terms that may be useful. Small values are used so the embedded rules dominate initially. The network is then trained from examples to improve its performance. Because the embedded symbolic rules are often classifications, the cross-entropy error function may be preferable to the mean-squared-error function [342]. The embedded rules represent the default knowledge of the system, so if weight-decay is used it may be desirable to let the weights decay to their initial values rather than zero.

7.2.7 Initialization from Fuzzy Rule Systems

For problems where the desired input-output relationship is well understood and expressible by a small set of rules, initialization based on a fuzzy logic implementation has been suggested, e.g., [291]. This is similar to the method of section 7.2.6, but the original rules are fuzzy.

As a very simple example, if we know that the output should be large when two input variables, A and B, are large, we can prewire one of the hidden nodes to implement (A AND B) and connect it to the output node with a positive weight. (Simple logical functions such as AND, OR, and NOT are easy to implement in a single node by appropriate selection of the weights and thresholds. A AND B, for example, can be realized by connecting large positive weights from A and B and selecting the threshold so that the node is active only when both A and B are active.) The information in the rules initializes the network near a reasonably good solution which is then improved by further training on examples. Even when we don't have complete knowledge about the desired solution, we may have knowledge about certain aspects of it that can be "injected" into the network in this way.

7.2.8 Remarks

Bias It should be noted that basing the initial state on a solution provided by another method introduces bias which may remain in the final network. Subsequent training may amount to mere second-order adjustments of the initial solution so a network initialized from a nearest-neighbor classifier, for example, may end up with classification properties similar to the nearest-neighbor classifier.

This is desirable in some cases because it provides some assurance that the network will behave properly in critical situations and it helps in understanding the trained network. When the initial solution is satisfactory, implementation as a neural network may be useful simply to gain properties such as parallel operation and some fault tolerance. In cases where the objective is simply to accelerate learning however, the bias may be a problem if the initial solution is only a poor guess.

Effect of the Optimization Procedure The training procedure has some effect on how much of the initial bias will be retained in the final network. In cases where it is desirable to preserve the basic form of the initial solution, small learning rates are appropriate to avoid leaving the initial basin of attraction. Local search methods (such as gradient descent) are appropriate in this case rather than truly global search techniques which ignore the initial state. If the initial solution is good, more powerful gradient techniques such as quasi-Newton methods or conjugate gradient descent may be preferable to back-propagation since their convergence behavior in the neighborhood of a solution is much better.