

9.2.2 Defining a Network Topology

“How can I design the neural network’s topology?” Before training can begin, the user must decide on the network topology by specifying the number of units in the input layer, the number of hidden layers (if more than one), the number of units in each hidden layer, and the number of units in the output layer.

Normalizing the input values for each attribute measured in the training tuples will help speed up the learning phase. Typically, input values are normalized so as to fall between 0.0 and 1.0. Discrete-valued attributes may be encoded such that there is one input unit per domain value. For example, if an attribute A has three possible or known values, namely $\{a_0, a_1, a_2\}$, then we may assign three input units to represent A . That is, we may have, say, I_0, I_1, I_2 as input units. Each unit is initialized to 0. If $A = a_0$, then I_0 is set to 1 and the rest are 0. If $A = a_1$, then I_1 is set to 1 and the rest are 0, and so on.

Neural networks can be used for both classification (to predict the class label of a given tuple) and numeric prediction (to predict a continuous-valued output). For classification, one output unit may be used to represent two classes (where the value 1 represents one class, and the value 0 represents the other). If there are more than two classes, then one output unit per class is used. (See Section 9.7.1 for more strategies on multiclass classification.)

There are no clear rules as to the “best” number of hidden layer units. Network design is a trial-and-error process and may affect the accuracy of the resulting trained network. The initial values of the weights may also affect the resulting accuracy. Once a network has been trained and its accuracy is not considered acceptable, it is common to repeat the training process with a different network topology or a different set of initial weights. Cross-validation techniques for accuracy estimation (described in Chapter 8) can be used to help decide when an acceptable network has been found. A number of automated techniques have been proposed that search for a “good” network structure. These typically use a hill-climbing approach that starts with an initial structure that is selectively modified.

9.2.3 Backpropagation

“How does backpropagation work?” Backpropagation learns by iteratively processing a data set of training tuples, comparing the network’s prediction for each tuple with the actual known *target* value. The target value may be the known class label of the training tuple (for classification problems) or a continuous value (for numeric prediction). For each training tuple, the weights are modified so as to minimize the mean-squared error between the network’s prediction and the actual target value. These modifications are made in the “backwards” direction (i.e., from the output layer) through each hidden layer down to the first hidden layer (hence the name *backpropagation*). Although it is not guaranteed, in general the weights will eventually converge, and the learning process stops. The algorithm is summarized in Figure 9.3. The steps involved are expressed in terms of inputs, outputs, and errors, and may seem awkward if this is your first look at

Algorithm: Backpropagation. Neural network learning for classification or numeric prediction, using the backpropagation algorithm.

Input:

- D , a data set consisting of the training tuples and their associated target values;
- l , the learning rate;
- $network$, a multilayer feed-forward network.

Output: A trained neural network.

Method:

```

(1) Initialize all weights and biases in  $network$ ;
(2) while terminating condition is not satisfied {
(3)   for each training tuple  $X$  in  $D$  {
(4)     // Propagate the inputs forward:
(5)     for each input layer unit  $j$  {
(6)        $O_j = I_j$ ; // output of an input unit is its actual input value
(7)     for each hidden or output layer unit  $j$  {
(8)        $I_j = \sum_i w_{ij} O_i + \theta_j$ ; // compute the net input of unit  $j$  with respect to
           the previous layer,  $i$ 
(9)        $O_j = \frac{1}{1+e^{-I_j}}$ ; } // compute the output of each unit  $j$ 
(10)    // Backpropagate the errors:
(11)    for each unit  $j$  in the output layer
(12)       $Err_j = O_j(1 - O_j)(T_j - O_j)$ ; // compute the error
(13)    for each unit  $j$  in the hidden layers, from the last to the first hidden layer
(14)       $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$ ; // compute the error with respect to
           the next higher layer,  $k$ 
(15)    for each weight  $w_{ij}$  in  $network$  {
(16)       $\Delta w_{ij} = (l) Err_j O_i$ ; // weight increment
(17)       $w_{ij} = w_{ij} + \Delta w_{ij}$ ; } // weight update
(18)    for each bias  $\theta_j$  in  $network$  {
(19)       $\Delta \theta_j = (l) Err_j$ ; // bias increment
(20)       $\theta_j = \theta_j + \Delta \theta_j$ ; } // bias update
(21)  } }
```

Figure 9.3 Backpropagation algorithm.

neural network learning. However, once you become familiar with the process, you will see that each step is inherently simple. The steps are described next.

Initialize the weights: The weights in the network are initialized to small random numbers (e.g., ranging from -1.0 to 1.0 , or -0.5 to 0.5). Each unit has a *bias* associated with it, as explained later. The biases are similarly initialized to small random numbers.

Each training tuple, X , is processed by the following steps.

Propagate the inputs forward: First, the training tuple is fed to the network's input layer. The inputs pass through the input units, unchanged. That is, for an input unit, j ,

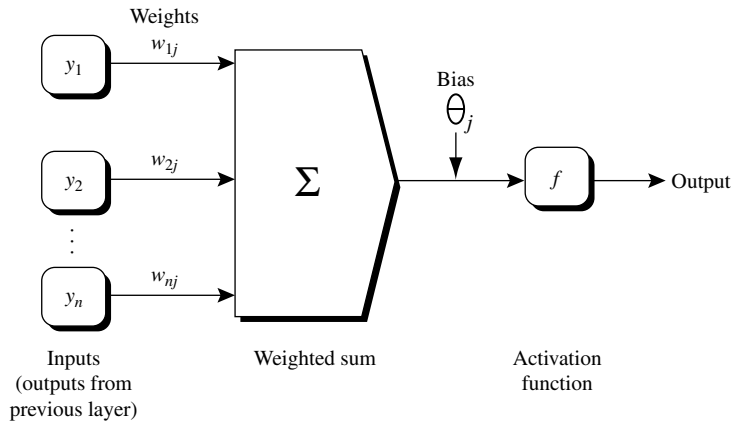


Figure 9.4 Hidden or output layer unit j : The inputs to unit j are outputs from the previous layer. These are multiplied by their corresponding weights to form a weighted sum, which is added to the bias associated with unit j . A nonlinear activation function is applied to the net input. (For ease of explanation, the inputs to unit j are labeled y_1, y_2, \dots, y_n . If unit j were in the first hidden layer, then these inputs would correspond to the input tuple (x_1, x_2, \dots, x_n) .)

its output, O_j , is equal to its input value, I_j . Next, the net input and output of each unit in the hidden and output layers are computed. The net input to a unit in the hidden or output layers is computed as a linear combination of its inputs. To help illustrate this point, a hidden layer or output layer unit is shown in Figure 9.4. Each such unit has a number of inputs to it that are, in fact, the outputs of the units connected to it in the previous layer. Each connection has a weight. To compute the net input to the unit, each input connected to the unit is multiplied by its corresponding weight, and this is summed. Given a unit, j in a hidden or output layer, the net input, I_j , to unit j is

$$I_j = \sum_i w_{ij} O_i + \theta_j, \quad (9.4)$$

where w_{ij} is the weight of the connection from unit i in the previous layer to unit j ; O_i is the output of unit i from the previous layer; and θ_j is the **bias** of the unit. The bias acts as a threshold in that it serves to vary the activity of the unit.

Each unit in the hidden and output layers takes its net input and then applies an **activation** function to it, as illustrated in Figure 9.4. The function symbolizes the activation of the neuron represented by the unit. The **logistic**, or **sigmoid**, function is used. Given the net input I_j to unit j , then O_j , the output of unit j , is computed as

$$O_j = \frac{1}{1 + e^{-I_j}}. \quad (9.5)$$

This function is also referred to as a *squashing function*, because it maps a large input domain onto the smaller range of 0 to 1. The logistic function is nonlinear and differentiable, allowing the backpropagation algorithm to model classification problems that are linearly inseparable.

We compute the output values, O_j , for each hidden layer, up to and including the output layer, which gives the network's prediction. In practice, it is a good idea to cache (i.e., save) the intermediate output values at each unit as they are required again later when backpropagating the error. This trick can substantially reduce the amount of computation required.

Backpropagate the error: The error is propagated backward by updating the weights and biases to reflect the error of the network's prediction. For a unit j in the output layer, the error Err_j is computed by

$$Err_j = O_j(1 - O_j)(T_j - O_j), \quad (9.6)$$

where O_j is the actual output of unit j , and T_j is the known target value of the given training tuple. Note that $O_j(1 - O_j)$ is the derivative of the logistic function.

To compute the error of a hidden layer unit j , the weighted sum of the errors of the units connected to unit j in the next layer are considered. The error of a hidden layer unit j is

$$Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}, \quad (9.7)$$

where w_{jk} is the weight of the connection from unit j to a unit k in the next higher layer, and Err_k is the error of unit k .

The weights and biases are updated to reflect the propagated errors. Weights are updated by the following equations, where Δw_{ij} is the change in weight w_{ij} :

$$\Delta w_{ij} = (l)Err_j O_i. \quad (9.8)$$

$$w_{ij} = w_{ij} + \Delta w_{ij}. \quad (9.9)$$

“What is l in Eq. (9.8)?” The variable l is the **learning rate**, a constant typically having a value between 0.0 and 1.0. Backpropagation learns using a gradient descent method to search for a set of weights that fits the training data so as to minimize the mean-squared distance between the network's class prediction and the known target value of the tuples.¹ The learning rate helps avoid getting stuck at a local minimum in decision space (i.e., where the weights appear to converge, but are not the optimum solution) and encourages finding the global minimum. If the learning rate is too small, then learning will occur at a very slow pace. If the learning rate is too large, then oscillation between

¹A method of gradient descent was also used for training Bayesian belief networks, as described in Section 9.1.2.

inadequate solutions may occur. A rule of thumb is to set the learning rate to $1/t$, where t is the number of iterations through the training set so far.

Biases are updated by the following equations, where $\Delta\theta_j$ is the change in bias θ_j :

$$\Delta\theta_j = (l)Err_j. \quad (9.10)$$

$$\theta_j = \theta_j + \Delta\theta_j. \quad (9.11)$$

Note that here we are updating the weights and biases after the presentation of each tuple. This is referred to as **case updating**. Alternatively, the weight and bias increments could be accumulated in variables, so that the weights and biases are updated after all the tuples in the training set have been presented. This latter strategy is called **epoch updating**, where one iteration through the training set is an **epoch**. In theory, the mathematical derivation of backpropagation employs epoch updating, yet in practice, case updating is more common because it tends to yield more accurate results.

Terminating condition: Training stops when

- All Δw_{ij} in the previous epoch are so small as to be below some specified threshold, or
- The percentage of tuples misclassified in the previous epoch is below some threshold, or
- A prespecified number of epochs has expired.

In practice, several hundreds of thousands of epochs may be required before the weights will converge.

“*How efficient is backpropagation?*” The computational efficiency depends on the time spent training the network. Given $|D|$ tuples and w weights, each epoch requires $O(|D| \times w)$ time. However, in the worst-case scenario, the number of epochs can be exponential in n , the number of inputs. In practice, the time required for the networks to converge is highly variable. A number of techniques exist that help speed up the training time. For example, a technique known as *simulated annealing* can be used, which also ensures convergence to a global optimum.

Example 9.1 Sample calculations for learning by the backpropagation algorithm. Figure 9.5 shows a multilayer feed-forward neural network. Let the learning rate be 0.9. The initial weight and bias values of the network are given in Table 9.1, along with the first training tuple, $X = (1, 0, 1)$, with a class label of 1.

This example shows the calculations for backpropagation, given the first training tuple, X . The tuple is fed into the network, and the net input and output of each unit

are computed. These values are shown in Table 9.2. The error of each unit is computed and propagated backward. The error values are shown in Table 9.3. The weight and bias updates are shown in Table 9.4. ■

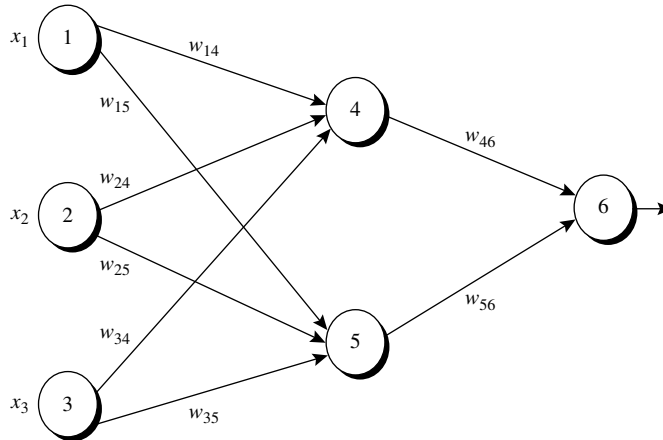


Figure 9.5 Example of a multilayer feed-forward neural network.

Table 9.1 Initial Input, Weight, and Bias Values

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6
1	0	1	0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2	-0.4	0.2	0.1

Table 9.2 Net Input and Output Calculations

Unit, j	Net Input, I_j	Output, O_j
4	$0.2 + 0 - 0.5 - 0.4 = -0.7$	$1/(1 + e^{0.7}) = 0.332$
5	$-0.3 + 0 + 0.2 + 0.2 = 0.1$	$1/(1 + e^{-0.1}) = 0.525$
6	$(-0.3)(0.332) - (0.2)(0.525) + 0.1 = -0.105$	$1/(1 + e^{0.105}) = 0.474$

Table 9.3 Calculation of the Error at Each Node

Unit, j	Err $_j$
6	$(0.474)(1 - 0.474)(1 - 0.474) = 0.1311$
5	$(0.525)(1 - 0.525)(0.1311)(-0.2) = -0.0065$
4	$(0.332)(1 - 0.332)(0.1311)(-0.3) = -0.0087$

Table 9.4 Calculations for Weight and Bias Updating

Weight or Bias	New Value
w_{46}	$-0.3 + (0.9)(0.1311)(0.332) = -0.261$
w_{56}	$-0.2 + (0.9)(0.1311)(0.525) = -0.138$
w_{14}	$0.2 + (0.9)(-0.0087)(1) = 0.192$
w_{15}	$-0.3 + (0.9)(-0.0065)(1) = -0.306$
w_{24}	$0.4 + (0.9)(-0.0087)(0) = 0.4$
w_{25}	$0.1 + (0.9)(-0.0065)(0) = 0.1$
w_{34}	$-0.5 + (0.9)(-0.0087)(1) = -0.508$
w_{35}	$0.2 + (0.9)(-0.0065)(1) = 0.194$
θ_6	$0.1 + (0.9)(0.1311) = 0.218$
θ_5	$0.2 + (0.9)(-0.0065) = 0.194$
θ_4	$-0.4 + (0.9)(-0.0087) = -0.408$

“How can we classify an unknown tuple using a trained network?” To classify an unknown tuple, X , the tuple is input to the trained network, and the net input and output of each unit are computed. (There is no need for computation and/or backpropagation of the error.) If there is one output node per class, then the output node with the highest value determines the predicted class label for X . If there is only one output node, then output values greater than or equal to 0.5 may be considered as belonging to the positive class, while values less than 0.5 may be considered negative.

Several variations and alternatives to the backpropagation algorithm have been proposed for classification in neural networks. These may involve the dynamic adjustment of the network topology and of the learning rate or other parameters, or the use of different error functions.

9.2.4 Inside the Black Box: Backpropagation and Interpretability

“Neural networks are like a black box. How can I ‘understand’ what the backpropagation network has learned?” A major disadvantage of neural networks lies in their knowledge representation. Acquired knowledge in the form of a network of units connected by weighted links is difficult for humans to interpret. This factor has motivated research in extracting the knowledge embedded in trained neural networks and in representing that knowledge symbolically. Methods include extracting rules from networks and sensitivity analysis.

Various algorithms for rule extraction have been proposed. The methods typically impose restrictions regarding procedures used in training the given neural network, the network topology, and the discretization of input values.

Fully connected networks are difficult to articulate. Hence, often the first step in extracting rules from neural networks is **network pruning**. This consists of simplifying