→ We can use gradient techniques. One commonly used gradient technique is backpropagation training:

Backpropagation Networks

Backpropagation networks are multilayer networks in which the various layers are cascaded. Figure 14.11 shows a typical three-layer backpropagation network.

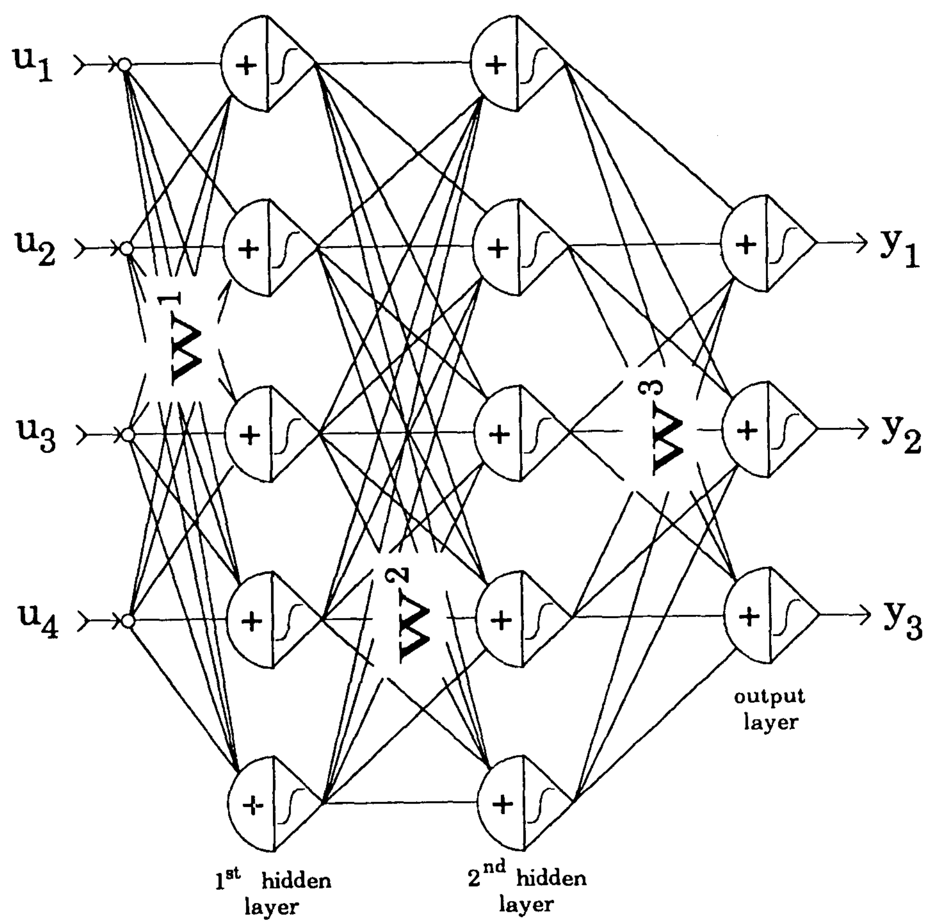

Figure 14.11. Three-layer backpropagation network.

The sigmoid function is particularly convenient because of its simple partial derivative:

$$\frac{\partial y}{\partial x} = y \cdot (1.0 - y) = \text{logistic}(y) \tag{14.17}$$

The partial derivative of the output y with respect to state x does not depend on x explicitly. It can be written as a logistic function of the output y.

We shall train the output layer in basically the same manner as in the case of the single–layer network, but we shall modify the formula for δ_j. Instead of simply using the difference between the desired output $\hat{y}_j$ and the true output y_j, we multiply this difference by the activation gradient:

$$\delta_j = \frac{\partial y}{\partial x} \cdot (\hat{y}_j - y_j) = y_j \cdot (1.0 - y_j) \cdot (\hat{y}_j - y_j) \tag{14.10^{alt}}$$

Therefore, the matrix version of the learning algorithm for the output layer can now be written as:

$$\mathbf{W}^{\mathbf{n}}_{\mathbf{k+1}} = \mathbf{W}^{\mathbf{n}}_{\mathbf{k}} + g * \left(\mathbf{y}^{\mathbf{n}}_{\mathbf{k}} \;.\!*\; (\text{ONES}(\mathbf{y}^{\mathbf{n}}_{\mathbf{k}}) - \mathbf{y}^{\mathbf{n}}_{\mathbf{k}}) \;.\!*\; (\hat{\mathbf{y}} - \mathbf{y}^{\mathbf{n}}_{\mathbf{k}})\right) * \mathbf{u}^{\mathbf{n}\prime}_{\mathbf{k}} \tag{14.13^{alt}}$$

The subscript k denotes the k^{th} iteration, whereas the superscript n denotes the n^{th} stage (layer) of the multilayer network. I assume that the network has exactly n stages. Equation (14.13^{alt}) is written in a pseudo–CTRL–C (pseudo–MATLAB) style. The '*' operator denotes a regular matrix multiplication, whereas the '.*' operator denotes an elementwise multiplication. The vector $\mathbf{u}_\mathbf{k}^\mathbf{n}$ is obviously identical to $\mathbf{y}_\mathbf{k}^{\mathbf{n}-1}$. Let:

$$\vec{\delta}_k^n = \mathbf{y}_\mathbf{k}^\mathbf{n} \; .* \; (\mathrm{ONES}(\mathbf{y}_\mathbf{k}^\mathbf{n}) - \mathbf{y}_\mathbf{k}^\mathbf{n}) \; .* \; (\hat{\mathbf{y}} - \mathbf{y}_\mathbf{k}^\mathbf{n}) \qquad (14.18)$$

denote the k^{th} iteration of the $\vec{\delta}$ vector for the n^{th} (output) stage of the multilayer network. Using Eq.(14.18), we can rewrite Eq.(14.13^{alt}) as follows:

$$\mathbf{W}_{\mathbf{k}+1}^\mathbf{n} = \mathbf{W}_\mathbf{k}^\mathbf{n} + g \; * \; \vec{\delta}_k^n \; * \; \mathbf{u}_\mathbf{k}^{\mathbf{n}\prime} \qquad (14.19)$$

Unfortunately, this algorithm will work for the output stage of the multilayer network only. We cannot train the hidden layers in the same fashion since we don't have a desired output for these stages. Therefore, we replace the gradient by another (unsupervised) updating function. The $\vec{\delta}$ vector of the ℓ^{th} hidden layer is computed as follows:

$$\vec{\delta}_k^\ell = \mathbf{y}_\mathbf{k}^\ell \; .* \; (\mathrm{ONES}(\mathbf{y}_\mathbf{k}^\ell) - \mathbf{y}_\mathbf{k}^\ell) \; .* \; (\mathbf{W}_\mathbf{k}^{\ell+1\prime} \; * \; \vec{\delta}_k^{\ell+1}) \qquad (14.20)$$

Instead of weighing the $\vec{\delta}$ vector with the (unavailable) difference between the desired and the true output of that stage, we propagate the weighted $\vec{\delta}$ vector of the subsequent stage back through the network. We then compute the next iteration of the weighting matrix of this hidden layer using Eq.(14.19) applied to the ℓ^{th} stage, i.e.:

$$\mathbf{W}_{\mathbf{k}+1}^\ell = \mathbf{W}_\mathbf{k}^\ell + g \; * \; \vec{\delta}_k^\ell \; * \; \mathbf{u}_\mathbf{k}^{\ell\prime} \qquad (14.21)$$

In this fashion, we proceed backward through the entire network.

The algorithm starts by setting all weighting matrices to small random matrices. We apply the true input to the network and propagate the true input forward to the true output, generating the first iteration on all signals in the network. We then propagate the gradients backward through the network to obtain the first iteration on all the weighting matrices. We then use these weighting matrices to propagate the same true input once more forward through the network to obtain the second iteration on the signals and then propagate the modified gradients backward through the entire network to obtain the second iteration on the weighting matrices. Consequently, the $\mathbf{u}^\ell$ and $\mathbf{y}^\ell$ vectors of the ℓ^{th} stage are updated on the forward path, while the $\vec{\delta}^\ell$ vector and the $\mathbf{W}^\ell$ matrix are updated on the backward path. Each iteration consists of one forward path followed by one backward path.

The backpropagation algorithm was made popular by Rumelhart *et al.* [14.32]. It presented the artificial neural network research community with the first systematic (although still heuristic) algorithm for training multilayer networks. The backpropagation algorithm has a fairly benign stability behavior. It will converge on many problems provided the gain g has been properly selected. Unfortunately, its convergence speed is usually very slow. Typically, a backpropagation training session may require several hundred thousand iterations for convergence.

Several enhancements of the algorithm have been proposed. Frequently, a bias vector is added, i.e., the state of an artificial neuron is no longer the weighted sum of its inputs alone, but is computed using the formula:

$$\mathbf{x} = \mathbf{W} \cdot \mathbf{u} + \mathbf{b} \tag{14.22}$$

Conceptually, this is not a true enhancement. It simply means that the neuron has an additional input, which is always '1.' Consequently, the bias term is updated as follows:

$$\mathbf{b_{k+1}} = \mathbf{b_k} + g \cdot \vec{\delta}_k \tag{14.23}$$

Also, a small momentum term is frequently added to the weights in order to improve the convergence speed [14.19]:

$$\mathbf{W_{k+1}} = (1.0 + m) \cdot \mathbf{W_k} + g \cdot \vec{\delta}_k \cdot \mathbf{u_k}' \tag{14.24a}$$

The momentum should obviously be added to the bias term as well:

$$\mathbf{b_{k+1}} = (1.0 + m) \cdot \mathbf{b_k} + g \cdot \vec{\delta}_k \tag{14.24b}$$

The momentum m is usually very small, $m \approx 0.01$.

Other references add a small percentage of the last change in the matrix to the weight update equation [14.12]:

$$\mathbf{\Delta W_k} = g \cdot \vec{\delta}_k \cdot \mathbf{u_k}' \tag{14.25a}$$

$$\mathbf{W_{k+1}} = \mathbf{W_k} + \mathbf{\Delta W_k} + m \cdot \mathbf{\Delta W_{k-1}} \tag{14.25b}$$

Finally, it is quite common to limit the amount by which the $\vec{\delta}$ vectors, the $\mathbf{b}$ vectors, and the $\mathbf{W}$ matrices can change in a single step. This often improves the stability behavior of the algorithm.

Example: $y = u_1 \otimes u_2$ ← exor

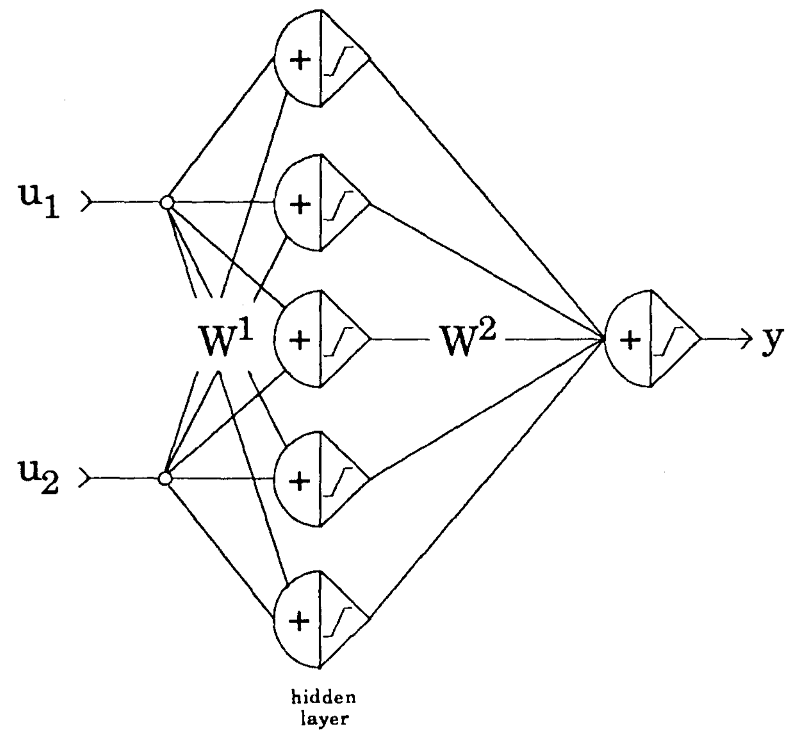

Figure 14.14. Backpropagation network for XOR.

The length of the hidden layer is arbitrary. In our program, we made this a parameter, *lhid*, which can be chosen at will. The program is shown here:

```
// This procedure designs a backpropagation network for XOR
// Select the length of the hidden layer (LHID) first
//
deff limit -c
deff tri -c
//
// Define the input and target vectors
//
inpt = [ -1  -1   1   1
         -1   1  -1   1 ];
target = [ -1   1   1  -1 ];
```

```
// Set the weighting matrices and biases
//
W1 = 0.1 * (2.0*RAND(lhid, 2)- ONES(lhid, 2));
W2 = 0.1 * (2.0*RAND(1, lhid)- ONES(1, lhid));
b1 = ZROW(lhid, 1);   b2 = ZROW(1);
WW1 = ZROW(lhid, 2);   WW2 = ZROW(1, lhid);
bb1 = ZROW(lhid, 1);   bb2 = ZROW(1);
//
// Set the gains and momenta
//
g1 = 0.6;   g2 = 0.3;
m1 = 0.06;   m2 = 0.03;
//
// Set the termination condition
//
crit = 0.025;   error = 1.0;   count = 0;
//
// Learn the weights and biases
//
while error > crit, ...
  count = count + 1; ...
  ... //
  ... // Loop over all input/target pairs
  ... //
  error = 0; ...
  for nbr = 1:4, ...
    u1 = inpt(:, nbr); ...
    y2h = target(nbr); ...
    ... //
    ... // Forward pass
    ... //
    x1 = WW1 * u1 + bb1; ...
    y1 = LIMIT(x1); ...
    u2 = y1; ...
    x2 = WW2 * u2 + bb2; ...
    y2 = LIMIT(x2); ...
    ... //
    ... // Backward pass
    ... //
    e = y2h - y2; ...
    delta2 = TRI(y2) .* e; ...
    W2 = W2 + g2 * delta2 * (u2') + m2 * WW2; ...
    b2 = b2 + g2 * delta2 + m2 * bb2; ...
    delta1 = TRI(y1) .* ((WW2') * delta2); ...
    W1 = W1 + g1 * delta1 * (u1') + m1 * WW1; ...
    b1 = b1 + g1 * delta1 + m1 * bb1; ...
    error = error+ NORM(e); ...
  end, ...
```

```
    ... // Update the momentum matrices and vectors
    ... //
    WW1 = W1;   WW2 = W2; ...
    bb1 = b1;   bb2 = b2; ...
end
//
// Apply the learned network to evaluate the truth table
//
y = ZROW(target);
for nbr = 1:4, ...
    u1 = inpt(:, nbr); ...
    x1 = WW1 * u1 + bb1; ...
    y1 = LIMIT(x1); ...
    u2 = y1; ...
    x2 = WW2 * u2 + bb2; ...
    y2 = LIMIT(x2); ...
    y(nbr) = y2; ...
end
//
// Display the results
//
y
//
return
```

It took some persuasion to get this program to work. The first difficulty was with the activation functions. The sigmoid function is no longer adequate since the output varies between -1.0 and $+1.0$, and not between 0.0 and 1.0. In this case, the sigmoid function is frequently replaced by:

$$y = \frac{2}{\pi} \cdot \tan^{-1}(x) \tag{14.30}$$

which also has a very convenient partial derivative:

$$\frac{\partial y}{\partial x} = \frac{2}{\pi} \cdot \frac{1.0}{1.0 + x^2} \tag{14.31}$$

However, this function won't converge for our application either. Since we wish to obtain outputs of exactly $+1.0$ and -1.0, we would need infinitely large states, and therefore infinitely large weights.

Without the $2/\pi$ term, the network does learn, but converges very slowly. Therefore, we decided to eliminate the requirement of a continuous derivative and used a limit function as the activation function:

```
// [y] = LIMIT(x)
//
// This procedure computes the limit function
//
[n, m] = SIZE(x);
for i = 1:n, ...
   y(i) = MIN([MAX([x(i), -1.0]), 1.0]); ...
end
//
return
```

In this case, we cannot backpropagate the gradient. Instead, we make use of the fact that we know that all outputs must converge to either +1.0 or −1.0. We therefore punish the distance of the true output from either of these two points using the tri function [14.19]:

```
// [y] = TRI(x)
//
// This procedure computes the tri function
//
[n, m] = SIZE(x);
y = ONES(n, m)- ABS(x);
//
return
```

We call this type of network a *pseudobackpropagation network*.

In addition to the weighting matrices, we need biases and momenta. The optimization starts with a zero–weight matrix, but adds small random momenta to the weights and biases. After each iteration, the momenta are updated to point more toward the optimum solution.

The program converges fairly quickly. It usually takes less than 20 iterations to converge to the correct solution. The program is also fairly insensitive to the length of the hidden layer. The convergence is equally fast with $lhid = 8$, $lhid = 16$, and $lhid = 32$.

This discussion teaches us another lesson. The design of neural networks is still more an art than a science. We usually start with one of the classical textbook algorithms ... and discover that it doesn't work. We then modify the algorithm until it converges in a satisfactory manner for our application. However, there is little generality in this procedure. A technique that works in one case may fail when applied to a slightly different problem. The backpropagation algorithm, as presented in this section, was taken from