Consider some sort of network connectivity model with outputs F_i. Let our total error function look like

$$E=\frac{1}{2}\sum_I \omega_I^2 \text{ with } \omega_I=\left(T_I-F_I\right)$$

where

F_I - Output Activation (what we got): I is the target node index
T_I - Target Output Activation (what we want): I is the target-node index
E - Error function for a single training-set model

Where the ½ is there to remove an unneeded factor of two from the derivative and we want to find the minimum with respect to all the weights w (index labels α, β) for all the layers (index label γ), $w_{\alpha\beta\gamma}$, across all the training set data. I am using Greek index variables so as to not confuse this general indexing with the more specific version to be used later. To find a minimum we generally would take the gradient with respect to $w_{\alpha\beta\gamma}$, set it equal to zero and then solve for all the weights through the simultaneous equations. The problem is that there may not be a true zero. In general, we need to find the minimum in E with respect to the weights. We thus have a minimization problem. There are many ways to find a minimum in an N-dimensional phase space. The simplest approach is gradient (steepest) decent, where we go "down hill" in weight space. We modify each weight by the negative derivative and put in a "learning factor", which I am calling λ, to allow us to control how far things can move down hill in each step:

$$\Delta w_{\alpha\beta\gamma}=-\lambda\frac{\partial E}{\partial w_{\alpha\beta\gamma}}.$$

Note that you can also make λ adaptive by making it dependent on the value of the error function E or the individual weights. You could also apply the **Newton–Raphson method** to converge on the correct weights. For a single variable function we can find a zero using

$$x^1=x^0-\frac{f(x)}{f'(x)},$$

so if the error function is E then we can find the minimum by looking for zeros in the first derivative (which is therefore the $f(x)$ used in the method) to locate a minimum:

$$w_{\alpha\beta\gamma}^1=w_{\alpha\beta\gamma}^0-\frac{\partial E/\partial w_{\alpha\beta\gamma}}{\partial^2 E/\partial w_{\alpha\beta\gamma}^2}.$$

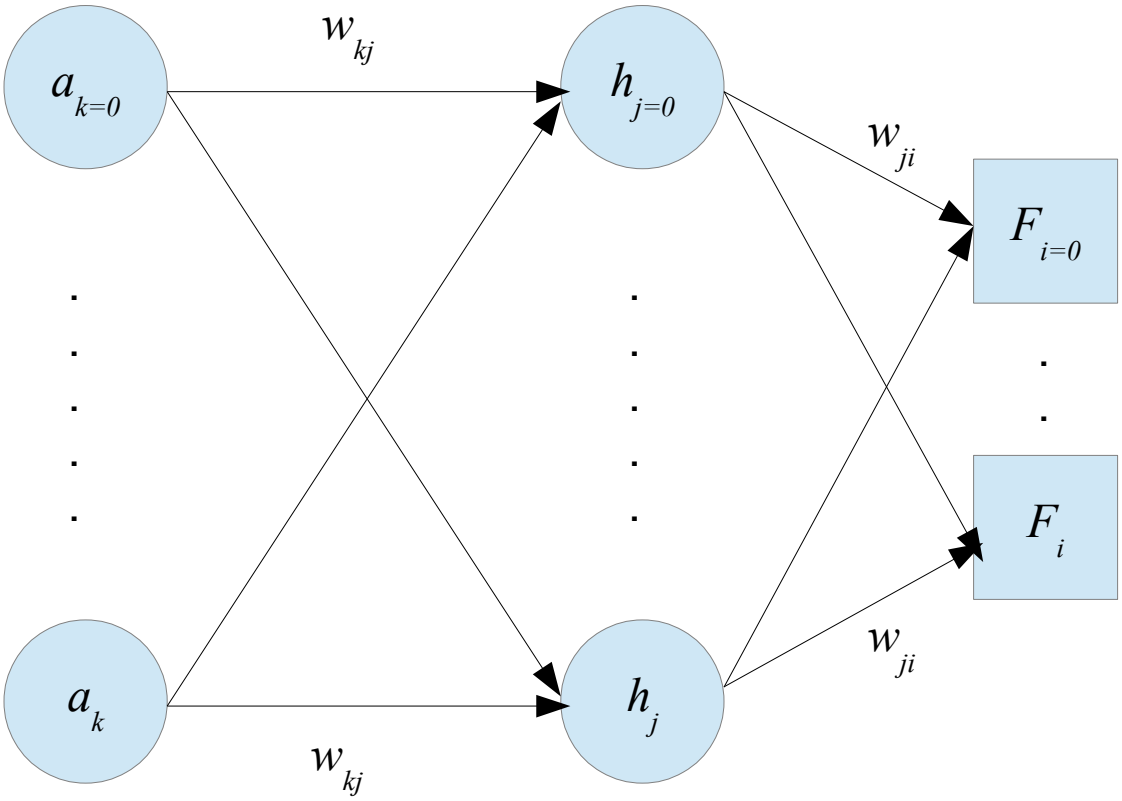
Or you could use the method of Davidon-Fletcher-Powell

http://en.wikipedia.org/wiki/Davidon-Fletcher-Powell_formula.

The problem with these methods is that you must calculate the error function three times for each weight because of the derivative, which is computationally intensive. If you have a fast system, then it is not a problem. On the other hand, although they might be slow they can be used for ANY connectivity model.

For all these methods, at first it looks like we still need to make three determinations of the error function for each weight because we have the derivative to deal with, but with a judicious choice of the connectivity model and activation rule along with a little grunt work shows how we can optimize the learning into a "back propagation" algorithm which significantly reduces the amount of computation. Back propagation is just an optimized version of steepest decent, as we shall soon see.

First we need to formalize our notation and our connectivity model. The initial use of our index letters was to help see the relations between the elements as they moved through the network. We now want to modify our notation to view things from right to left and start our index lettering on the output side:



What we want to know is how the error depends on the weights in each layer so we can determine how to modify the weights to minimize the error "on the fly". We are going to limit our derivation to a network with only a single hidden activation layer which I shall call a two-layer network since it has two layers of weights. The hidden layer is fully connected to both its input and output layer.

We start with the expressions for the output values and an error function which is now simply defined for only a single training event. $F_i = f\left(\sum_J h_J w_{Ji}\right)$, $E = \frac{1}{2}\sum_I \left(T_I - F_I\right)^2$, where $f(x)$ is the activation function. We should also note that by symmetry, the value of the hidden activation nodes looks a lot like the output activations in that $h_j = f\left(\sum_K a_K w_{Kj}\right)$.

First we want to see the effects of the error on the right-most set of weights coming out of the hidden layer (the *ji* weights)

$$\frac{\partial E}{\partial w_{ji}} = \sum_I \left(T_I - F_I\right)\frac{\partial\left(T_I - F_I\right)}{\partial w_{ji}} .$$

The T_i values are constants so we have

$$\frac{\partial E}{\partial w_{ji}} = -\sum_i \left(T_I - F_I\right)\frac{\partial F_I}{\partial w_{ji}} ,$$

but we know that $F_i = f\left(\sum_J h_J w_{Ji}\right)$, so we have

$$\frac{\partial E}{\partial w_{ji}} = -\sum_I \left(T_I - F_I\right)\frac{\partial f\left(\sum_J h_J w_{JI}\right)}{\partial w_{ji}} , \text{ or}$$

$$\frac{\partial E}{\partial w_{ji}} = -\sum_I \left(T_I - F_I\right) f'\left(\sum_J h_J w_{JI}\right)\frac{\partial \sum_J h_J w_{JI}}{\partial w_{ji}} .$$

The values of the activations in the hidden layer, h_j, are not dependent on the values of the weights to the right that feed into the output layer, w_{ji}, so the h_j terms look like constants to the derivative. We also need to note that we are going to differentiate for a single value of the index variables *i* and *j* on both sides, and since there are no cross-dependencies in our connectivity model then both the summations outside of the activation function reduce to only one term since all the other derivative terms will be zero, so we can see that

$\frac{\partial \sum_J h_J w_{JI}}{\partial w_{ji}} = h_j$, when $I = i$ and $J = j$ and is zero otherwise, so we have

$$\frac{\partial E}{\partial w_{ji}} = -\left(T_i - F_i\right) f'\left(\sum_J h_J w_{Ji}\right) h_j .$$

Let us define a new term

$$\psi_i = (T_i - F_i) f'\left(\sum_J h_J w_{Ji}\right),$$

as well as defining

$\Theta_i = \sum_J h_J w_{Ji}$ and $\omega_i = (T_i - F_i)$, which allows us to write

$$\psi_i = \omega_i f'(\Theta_i).$$

You will find that defining these collections will make coding the back propagation algorithm much simpler.

We thus finally have

$$\frac{\partial E}{\partial w_{ji}} = -h_j \psi_i .$$

We therefore have the expressions for updating the right-most set of weights:

$$\Delta w_{ji} = \lambda h_j \psi_i \text{ where } \psi_i = \omega_i f'(\Theta_i),\ \omega_i = (T_i - F_i),\ F_i = f(\Theta_i) \text{ and } \Theta_i = \sum_j h_J w_{Ji}.$$

This expression is how the weights in the right most connectivity layer will change to minimize the error function. Note that you will need to save Θ_i and h_j when the network is evaluated. Now let's move to the left most connectivity layer. We want to see the effects of the error on the first set of weights coming out of the initial input layer (the *kj* weights):

$$\frac{\partial E}{\partial w_{kj}} = \sum_I (T_I - F_I) \frac{\partial (T_I - F_I)}{\partial w_{kj}} .$$

Once again, the T_i values are constants so we have

$$\frac{\partial E}{\partial w_{kj}} = -\sum_I (T_I - F_I) \frac{\partial F_I}{\partial w_{kj}},$$

but we know that $F_i = f\left(\sum_J h_J w_{Ji}\right)$, so we have

$$\frac{\partial E}{\partial w_{kj}} = -\sum_I (T_I - F_I) \frac{\partial f\left(\sum_J h_J w_{JI}\right)}{\partial w_{kj}}, \text{ so}$$

$$\frac{\partial E}{\partial w_{kj}} = -\sum_I (T_I - F_I) f'\left(\sum_J h_J w_{JI}\right) \frac{\partial \sum_J h_j w_{JI}}{\partial w_{kj}} .$$

In the previous case we only had the *i* and *j* index values on both sides, so only a single differential element would survive the summations. In this case however, we have the *k* index and there are dependencies of h_j on w_k, so we cannot drop the all the sums up front this time. Recall that we defined $\psi_i = (T_i - F_i) f'\left(\sum_J h_J w_{Ji}\right)$, so we have

$$\frac{\partial E}{\partial w_{kj}} = -\sum_I \psi_I \frac{\partial \sum_J h_J w_{JI}}{\partial w_{kj}} .$$

The weights on the right connectivity layer are independent of the weights in the left connectivity layer and each activation h_j will depend on only one w_k for the matching values of *j*, so the right summation reduces to a single term, therefore

$$\frac{\partial \sum_J h_J w_{Ji}}{\partial w_{kj}} = w_{ji} \frac{\partial h_j}{\partial w_{kj}}$$, so we have

$$\frac{\partial E}{\partial w_{kj}} = -\sum_I \psi_I w_{jI} \frac{\partial h_j}{\partial w_{kj}} ,$$

but we know from the diagram that $h_j = f\left(\sum_K a_K w_{Kj}\right)$, so we have

$$\frac{\partial E}{\partial w_{kj}} = -\sum_I \psi_I w_{jI} \frac{\partial f\left(\sum_K a_K w_{Kj}\right)}{\partial w_{kj}}$$, or

$$\frac{\partial E}{\partial w_{kj}} = -\sum_I \psi_I w_{jI} f'\left(\sum_K a_K w_{Kj}\right) \frac{\partial \sum_K a_K w_{Kj}}{\partial w_{kj}} .$$

We have seen things like these relations before, and we should see that

$$\frac{\partial \sum_K a_K w_{Kj}}{\partial w_{kj}} = a_k$$, so

$$\frac{\partial E}{\partial w_{kj}} = -\sum_I \psi_I w_{jI} f'\left(\sum_K a_K w_{Kj}\right) a_k ,$$

or after pulling terms outside the summation over *i*,

$$\frac{\partial E}{\partial w_{kj}} = -a_k f'\left(\sum_K a_K w_{Kj}\right) \sum_I \psi_I w_{jI} .$$

If we now make the following definitions

$$\Psi_j = f'\left(\sum_K a_K w_{Kj}\right)\sum_I \psi_I w_{jI},\ \Theta_j = \sum_K a_K w_{Kj} \text{ and } \Omega_j = \sum_I \psi_I w_{jI},$$

so we can write

$$\Psi_j = \Omega_j f'(\Theta_j).$$

There will be a Θ_j and Ω_j for each hidden layer when it is acting as an output. We therefore have

$$\frac{\partial E}{\partial w_{kj}} = -a_k \Psi_j.$$

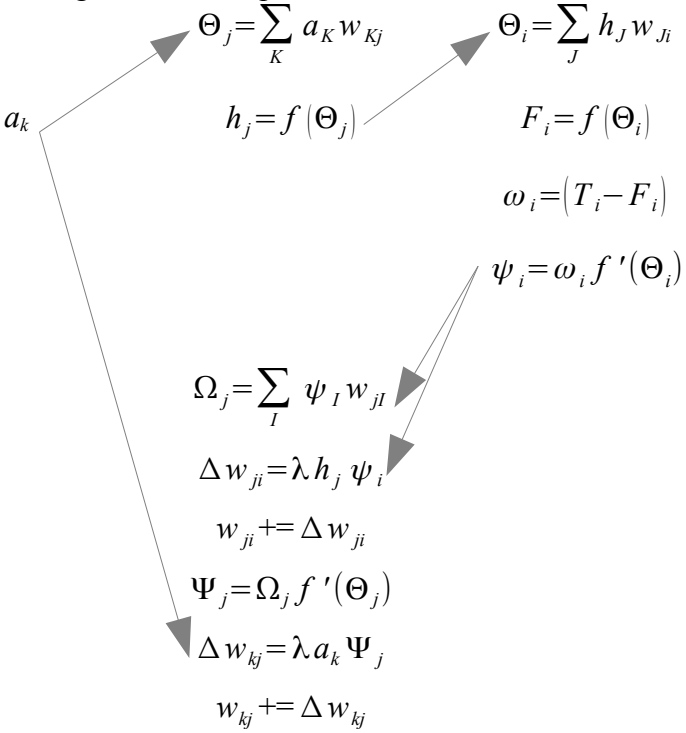
So we can update the weights in the left most layer with

$$\Delta w_{kj} = \lambda a_k \Psi_j \text{ where } \Psi_j = \Omega_j f'(\Theta_j),\ \Omega_j = \sum_I \psi_I w_{jI},\ \Theta_j = \sum_K a_K w_{Kj} \text{ and}$$
$$\psi_i = \omega_i f'(\Theta_i) \text{ where } \omega_i = (T_i - F_i),\ F_i = f(\Theta_i) \text{ and } \Theta_i = \sum_J h_J w_{Ji}.$$

We now have everything we need for a full back propagation implementation for a two layer network. The network is evaluated with the feed-forward calculations:

$$F_i = f(\Theta_i) \text{ where } \Theta_i = \sum_J h_J w_{Ji} \text{ and } h_j = f(\Theta_j) \text{ where } \Theta_j = \sum_K a_K w_{Kj}.$$

A diagram of the dependencies shows how the calculations are related:



Note that the change in weights on the right most layer are calculated and immediately applied in the layer to the left based on the calculation of the ψ_i values which are done when the network is executed. You therefore only need one forward loop-construct and one back propagation nested loop-construct.

Note that we can continue this back propagation derivation for three or more layers and would have something to the effect of

$$\Delta w_{mk} = \lambda a_m \Psi_k \text{ where } \Psi_k = \Omega_k f'(\Theta_k),\ \Omega_k = \sum_J \Psi_J w_{kJ},\ \Theta_k = \sum_M a_M w_{Mk}\ .$$

for the next layer to the left that would have activations a_m. Note that Ψ_j (which is tied to the output layer) is not the same as Ψ_k (which is tied to a hidden layer). You should also notice that there will be an array of Θ values for every right-side activation layer, so there will be an array of values for every hidden layer and the output layer. These values can be collected when the activation/weight multiplications are being performed when the network is being evaluated. You will also need to maintain the activation values for every hidden layer.

Now we want to implement it. Remember that the point of the optimization is to minimize the number of loops needed to perform the training, so you will need to explicitly write out all the loops and then see how they can be combined, and the operations ordered, to minimize run time.

Evaluate the network: During the evaluation you will need to accumulate the items needed for the back propagation training (Θ and ψ values for example). You will need to identify them for yourself. You should only need three loops (over i, j, and k). You need to remember that at some point you will simply be running the network with a predetermined set of weights and some arbitrary inputs (the training will be all done), so do NOT make execution of the network dependent on also training it. The pseudocode for <u>simply</u> evaluating the network and collecting the information need for training looks <u>something</u> like:

```
for i = 1 to Ni
   Θi = 0
   for j = 1 to Nj
      Θj = 0
      for k = 1 to Nk
         Θj += ak * wkj
      next k
      hj = f(Θj)
      Θi += hj * wji
   next j
   Fi = f(Θi)
   ωi = Ti - Fi
   ψi = ωi f'(Θi)
next i
```

Train the network: The outer most loop will be over the training-set. The inner loops will again iterate over i, j, and k. **The ordering of these loops should be the reverse of what was done to evaluate the network** since you are now moving backwards though it. As you calculate the Ω values, you will find that you can apply the change in weights on-the-fly once you are past the dependencies. No extra loops or using duplicate copies of the weights are needed, so you will only need one set of nested loops. You should write out the training in pseudo code similar to that for evaluating the network before you try to code it.

The Activation Function

Now what does $f'(x)$ look like? Let us use a simple sigmoid so that $f(x)=\frac{1}{1+e^{-x}}$. In that case:

$$f'(x)=\frac{d}{dx}f(x)=\frac{d}{dx}\left(1+e^{-x}\right)^{-1}=-\left(1+e^{-x}\right)^{-2}e^{-x}=-f^2(x)\frac{d}{dx}e^{-x}=f^2(x)e^{-x},$$

but we can write $e^{-x}=\frac{1}{f(x)}-1$, so $\frac{d}{dx}f(x)=f^2(x)\left(\frac{1}{f(x)}-1\right)=f(x)\left(1-f(x)\right)$,

so we finally have

$$f'(x)=f(x)\left(1-f(x)\right) \text{ if } f(x)=\frac{1}{1+e^{-x}}.$$