Proving the Backpropagated Delta Rule.

For simplicity, we will consider a single pattern, in a simple three-layer network, in which

k indexes the output units
j indexes the hidden units
i indexes the input "units"

We therefore need to alter the w_{kj}'s and the w_{ji}'s.

To achieve gradient descent when altering the w_{kj}'s we can simply apply the Delta Rule:

and this will decrease E_k, for small enough η.

If we had an expression for E_j, we would simply apply the Delta rule again:

However, we know that E_k depends on the A_k, and that the A_k depend on the Y_j. So we can write

$$\frac{\partial E}{\partial Y_j} = \sum_{k=1}^{s} \frac{\partial E}{\partial A_k} \frac{\partial A_k}{\partial Y_j}$$

where k indexes the s output units. Y_k depends only on A_k, so we can write:

$$\frac{\partial E}{\partial Y_j} = \sum_{k=1}^{s} \frac{\partial E}{\partial Y_k} \frac{dY_k}{dA_k} \frac{\partial A_k}{\partial Y_j}$$

A_k is calculated from the Y_j by weighted summation using the w_{kj} weights so that

$$\frac{\partial E}{\partial Y_j} = \sum_{k=1}^{s} \frac{\partial E}{\partial Y_k} \frac{dY_k}{dA_k} w_{kj}$$

We can then use the Delta rule to show us how the E depends on the w_{ji}, and hence how to alter the w_{ji} weights:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial Y_j}\frac{dY_j}{dA_j}\frac{\partial A_j}{\partial w_{ji}}$$

so that

$$\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{s} \frac{\partial E}{\partial Y_k}\frac{dY_k}{dA_k} w_{kj} . \frac{dY_j}{dA_j} . I_i$$

which gives us the learning rule.

And it is (reasonably!) local. Further, it can be applied

* for any well-behaved error measure
* for any strictly increasing and differentiable output function.

For the usual error measure,

$$\frac{\partial E}{\partial Y_k} = 2(D_k - Y_k). - 1$$

we get the learning rule

$$\Delta w_{ji} = \eta \sum_{k=1}^{s} E_k \frac{dY_k}{dA_k} w_{kj} . \frac{dY_j}{dA_j} . I_i$$

The Backpropogation Algorithm.

This algorithm may be clearer expressed programmatically.

```
Repeat
{
    for (pno=0;pno<N_Patterns;pno++)
    {
        /* forward pass */
        Apply Input[pno] to input units;
        Compute Y[pno] at output units ;
        /* Backward pass */
        For each layer, starting at output
        {
            For each unit in this layer
            {
                Compute the error at this unit
                For each weight to this unit
                {
                        Compute Δw
                        Apply Δw
                }
            }
        }
    }
    Increment epoch counter
    Compute total error
}
until (total error small enough or
       epoch count exceeded)
```

This form of the algorithm is known as *on-line update* as the weights are updated after each pattern-pair presentation.

There is another form, known as *batch update* in which the weights are updated only after a complete epoch presentation. In fact, the proof of the algorithm applies to the batch update version.

```
Repeat
{
    for (pno=0;pno<N_Patterns;pno++)
    {
        /* forward pass */
        Apply Input[pno] to input units;
        Compute Y[pno] at output units ;
        /* Backward pass */
        For each layer, starting at output
        {
            For each unit in this layer
            {
                Compute the error at this unit
                For each weight to this unit
                {
                        Compute $\Delta$w
                        Accumulate $\Delta$w
                }
            }
        }
    }
    Apply accumulated $\Delta$w’s
    Increment epoch counter
    Compute total error
}
until (total error small enough or
       epoch count exceeded)
```

The only difference between these is in when the weight changes are applied.