

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} and the w_{ji}

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

$$\Delta w_{kj} = -\eta \frac{\partial E_k}{\partial w_{kj}} = -\eta \frac{\partial E_k}{\partial z_k} \frac{\partial z_k}{\partial w_{kj}}$$

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:

$$\Delta w_{ji} = -\eta \frac{\partial E_j}{\partial w_{ji}} = -\eta \frac{\partial E_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ji}}$$

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

$$E_k = \frac{1}{n} \sum_{j=1}^n \frac{A_k}{Y_j} = \frac{1}{n} \sum_{j=1}^n \frac{1}{Y_j} \sum_{i=1}^m w_{ki} Y_{ij}$$

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

$$E_k = \frac{1}{n} \sum_{j=1}^n \frac{1}{Y_j} \sum_{i=1}^m w_{ki} Y_{ij} = \frac{1}{n} \sum_{j=1}^n \frac{1}{Y_j} \sum_{i=1}^m w_{ki} \frac{A_i}{Y_j} = \frac{1}{n} \sum_{j=1}^n \frac{1}{Y_j^2} \sum_{i=1}^m w_{ki} A_i$$

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

$$A_k = \sum_{j=1}^n w_{kj} Y_j = \sum_{j=1}^n w_{kj} \frac{1}{Y_j} \sum_{i=1}^m w_{ji} A_i = \sum_{j=1}^n \frac{w_{kj}}{Y_j} \sum_{i=1}^m w_{ji} A_i$$

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:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial w_{ji}}$$

so that

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial w_{ji}} = \frac{\partial E}{\partial \text{net}_j} \cdot \frac{\partial}{\partial w_{ji}} \left(\sum_k w_{jk} \text{net}_k \right)$$

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,

$$E = \frac{1}{2} \sum_j (\text{net}_j - \text{target}_j)^2$$

we get the learning rule

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial w_{ji}} = \frac{\partial E}{\partial \text{net}_j} \cdot \text{net}_k$$

The Backpropagation 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  $O_w$ 
                    Apply  $O_w$ 
                }
            }
        }
        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  $O_w$ 
                    Accumulate  $O_w$ 
                }
            }
        }
        Apply accumulated  $O_w$  to the
        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.