The Back-Propagation Algorithm

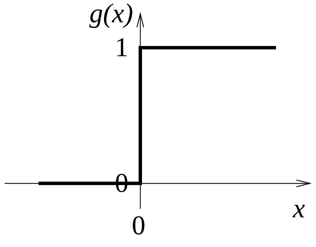
1 Introduction

Having looked at learning in a single TLU, we now look at the learning algorithm for a fully connected, layered, feedforward network. The algorithm we look at is called the *back-propagation algorithm* (or the *back-prop algorithm)* for reasons that will become clear below.

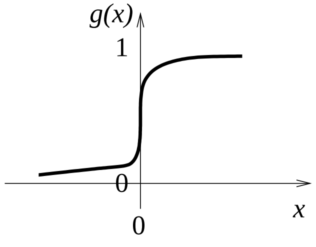
There is a change we have to make to the TLUs in our network in order to make this algorithm work properly. We have to redefine g, the activation function. We need g to be a function that can be differentiated. At the moment, g is a step-function, and these cannot be continuously differentiated. We replace our step-function by an s-shaped function called a *sigmoid*. We will use:

$$g(x) =_{\text{def}} \frac{1}{1+e^{-x}}$$

Note that previously TLUs (using step-functions) could output only 0 or 1. But now, TLUs (using sigmoid-functions) can output any real numbers between 0 and 1. (In fact, from this definition, the outputs can never quite reach 0 or 1.)



Step function
(threshold at zero)



Sigmoid function

This function *can* be differentiated. Its partial derivative with respect to x , $\frac{\delta g}{\delta x}$, also written g', is as follows:

$$g'(x) = g(x) \times (1 - g(x))$$

The back-prop learning algorithm is basically the same as the algorithm for a single TLU. But there's a problem that we have to solve. It is called the *blame assignment problem* (or, in some circumstances, the *credit assignment problem*). In our case, we need to decide by how much each existing weight is to blame for any error, and divvy-up the adjustment among those weights proportionately.

Why is this a problem (especially given that it wasn't a problem in TLU learning)? Updating weights between the hidden layer and the output layer is simple enough. We can easily compute the error between the actual net outputs and the target outputs, and we can divide this up among the weights. But how do we know the error at the hidden layer? In other words, we know what the hidden units *do* produce; but how do we know what the hidden units *ought* to produce. Without knowing their target outputs, we don't know their error, and so we can't share it out.

The back-prop algorithm solves this problem. It relies on having the nice uniform architecture of a fully connected, layered, feedforward network. (There are variants of the back-prop algorithm that can be used for networks that are not fully-connected and/or layered. There are alternative learning algorithms and even, for some cases, variants of back-prop that can be used for recurrent networks.)

As you might guess from its name, this algorithm propagates error back through the layers of the network. So first it computes the error at the output units, and adjusts the weights on the lines entering those output units. Then, *on the basis of those adjustments*, it can compute error at the hidden units and adjust the weights on the lines entering the hidden units.

For simplicity, we'll be sticking to two-layer networks. But in general, the error will be propagated back, layer by layer, through the whole network.

The neat thing about back-prop is that a unit's weights will change in a way that can be calculated using only information that is local to that unit.

2 Back-Propagation

Suppose we're presenting some training example to the network. The example comprises an input vector, $\mathbf{s}$ (the vector of incoming values) and the target output vector, $\mathbf{target}$ (the values we'd like to see appearing on the output lines of the units in the output layer). We activate the network, using $\mathbf{s}$, and we observe the *actual* output values from the output units, $\mathbf{output}$.

Let's look at one of these output units, o_k. Its input activation is the weighted sum of the outputs of all the hidden units, and we'll call this in_{o_k}. Its actual output is $g(\text{in}_{o_k})$ and we'll call this $\mathbf{output}_{o_k}$. Its target output is $\mathbf{target}_{o_k}$. The error at this output node is, of course:

$$\text{Err}_{o_k} =_{\text{def}} \text{target}_{o_k} - \text{output}_{o_k}$$

.

This output node receives weighted input from the hidden units. Suppose there are m hidden units, $\langle h_0, \ldots, h_m \rangle$. A given hidden unit, h_j, has an output of output_{h_j}, which it feeds into each of the output units, including o_k. The weight on the line between h_j and o_k will be denoted by w_{h_j,o_k}.

This weight needs adjusting in the light of the error computed above. The amount it should be adjusted by is

$$\Delta_{o_k} =_{\text{def}} g'(in_{o_k}) \times \text{Err}_{o_k}$$

The derivation of this is beyond the scope of this course, but some explanation is given at the end of these notes.

The formula for making the adjustment is then:

$$w_{h_j,o_k} := w_{h_j,o_k} + \alpha \times \text{output}_{h_j} \times \Delta_{o_k}$$

As before, α is the learning rate.

The above update rule is used to update all the weights between all the hidden units and all the output units.

Next, we must update the weights between the input units and the hidden units.

For this, we need to compute the error in the outputs of the hidden units. (This is the back-propagation!) Consider a particular hidden unit h_j. The idea is that h_j is responsible for a fraction of the sum of the Δ_{o_k} for each of the output nodes $\langle o_1, \ldots, o_k, \ldots, o_l \rangle$. Specifically,

$$\Delta_{h_j} =_{\text{def}} g'(in_{h_j}) \times \sum_{k=1}^{k=l} w_{h_j,o_k} \Delta_{o_k}$$

(Again, the derivation of this is beyond the scope of this course.)

The formula for making weight adjustments is then much as before:

$$w_{s_i,h_j} := w_{s_i,h_j} + \alpha \times s_i \times \Delta_{h_j}$$

We use this update rule to update all the weights between all the input units and all the hidden units.

This is summarised in pseudocode below.

```
Algorithm: TRAIN(network, dataset)
initialise all TLUs with random weights;
do
{   for each example $e \in dataset$
    {   UPDATE(network, $e$);
    }
}
while network hasn't converged
```

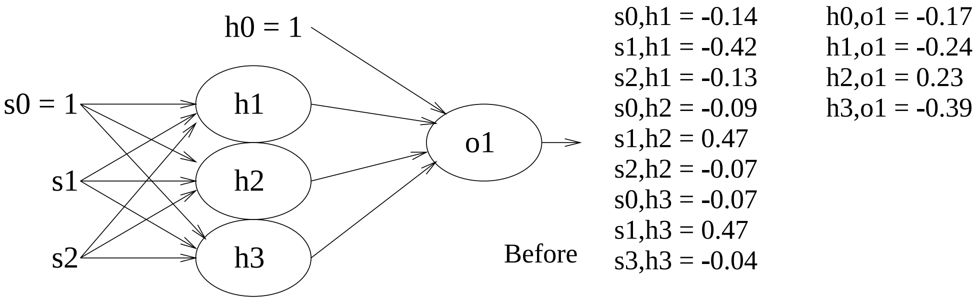
```
Algorithm: UPDATE(network, e)
inputVector := $e$'s input vector;
target := $e$'s output vector;
output := ACTIVATE(network, inputVector);
for each of network's output units $o_k$
{   $Err_{o_k} := target_{o_k} - output_{o_k}$;
    $\Delta_{o_k} := g'(in_{o_k}) \times Err_{o_k}$;
    $sumOfDeltas_{h_j} := 0$;
    for each of network's hidden units $h_j$
    {   $w_{h_j,o_k} := w_{h_j,o_k} + \alpha \times output_{h_j} \times \Delta_{o_k}$;
        $sumOfDeltas_{h_j} := sumOfDeltas_{h_j} + w_{h_j,o_k} \times \Delta_{o_k}$;
    }
    for each of network's hidden units $h_j$
    {   $\Delta_{h_j} := g'(in_{h_j}) \times sumOfDeltas_{h_j}$;
        for each of network's input units $s_i$
        {   $w_{s_i,h_j} := w_{s_i,h_j} + \alpha \times s_i \times \Delta_{h_j}$;
        }
    }
}
```

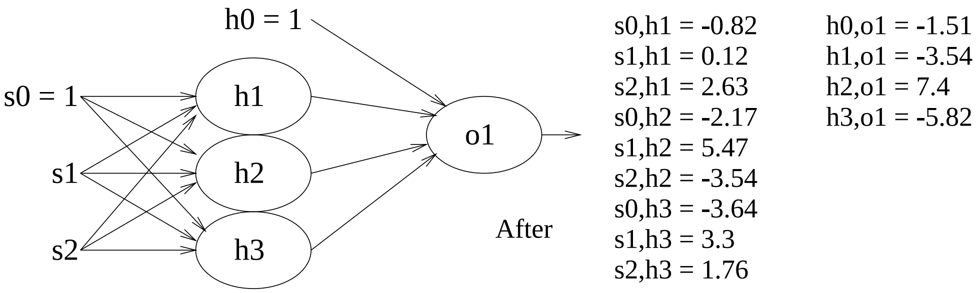
3 Example

I used the algorithm to learn a net to compute exclusive-or. The architecture of the net (excluding extra units in place of thresholds) was: two input units, three hidden units and one output unit.

Four examples were presented to the net (see below). I used a learning rate of 0.35. I trained the net until its total error on the training examples was no greater than 0.1. The number of epochs required to achieve this was surprisingly high. Obviously, exactly what happens in any particular use of the algorithm depends on the initial randomly-chosen weights. But it seems that between 2500 and 5000 epochs are needed!

The diagram shows what happened on one occasion when I ran the algorithm. The first diagram shows the random weights. The second diagram shows the weights after training. On this occasion, training required 3378 epochs.





The table shows the actual outputs of the net after it has been trained.

example input	target output	actual output
$\langle 1, 1\rangle$	0.1	0.13
$\langle 1, 0\rangle$	0.9	0.88
$\langle 0, 1\rangle$	0.9	0.87
$\langle 0, 0\rangle$	0.1	0.12

Question. *Why did I use target outputs of 0.1 instead of 0 and 0.9 instead of 1?*

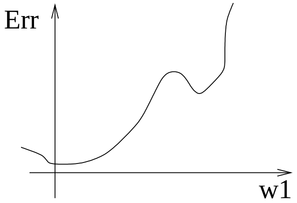
By the way, I got equally good results with a net that had only 2 units in its hidden layer. I also tried a net with only one hidden unit but I had to terminate the program — presumably it was not able to converge.

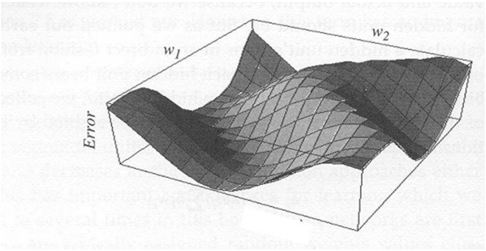
4 Gradient Descent Search

Where do the update rules come from? What's going on in the back-prop algorithm?

The idea of this algorithm is to minimise the total error. In fact, when deriving the update rules, we don't use the sum of the errors, we use the sum of the square of the errors. Why? Because some of our errors are positive numbers and others are negative numbers, and if we simply summed them, some would cancel out others, giving an underestimate of the total error. If we square them first, all numbers involved in the sum are positive. (And summing the squares makes the maths more convenient than summing the absolute differences.)

Imagine that, for each possible combination of weights, we could compute the total error and that we plotted these values on a graph. That's hard to visualise in general because, if there are n weights in the net, we need a $(n+1)$-dimensional graph. So, to help visualisation, in the left-hand diagram below, we pretend that our net has only one weight in it, and we plot the different error values for different weight values. (The right-hand diagram attempts a 3-dimensional effort where error is plotted against two different weights.)





If we are presently at some point on this line, then we want to change the weight in such a way as to reduce the error. The gradient of the line with respect to the weight shows how the error would change if we made a small change to that weight.

(In general, where we consider more than one weight, if we are presently at some point in this n-dimensional space, the gradient of the error surface with respect to each weight shows how the error would change if we changed that weight. And this is given by the partial derivative of the error function with respect to that weight.)

Each weight is changed by an amount proportional to the slope with respect to that weight and in such a way as to cause the error to decrease. This has the effect of moving the error in the direction of the steepest descent. Hence, we say that this learning algorithm carries out *gradient descent search* in the space of weights.

So, the sum of the squares of the errors is given by:

$$\text{Err} =_{\text{def}} \sum_{k=1}^{k=l} (\text{target}_{o_k} - \text{output}_{o_k})^2$$

We then derive (by differentiation) the change in the error with respect to a weight between the hidden layer and the output layer. This is what we get:

$$\frac{\delta \text{Err}}{\delta w_{h_j,o_k}} = -\text{output}_{h_j} \times \Delta_{o_k}$$

We then derive the change in the error with respect to a weight between the input layer and the hidden layer. This is what we get:

$$\frac{\delta \text{Err}}{\delta w_{s_i,h_j}} = -s_i \times \Delta_{h_j}$$

These are the quantities we use in the update rules.

(A technicality is that we are doing *stochastic gradient descent search* because we update the weights after each example, rather than finding the total error over all examples in the dataset. This technicality need not concern us in this module.)