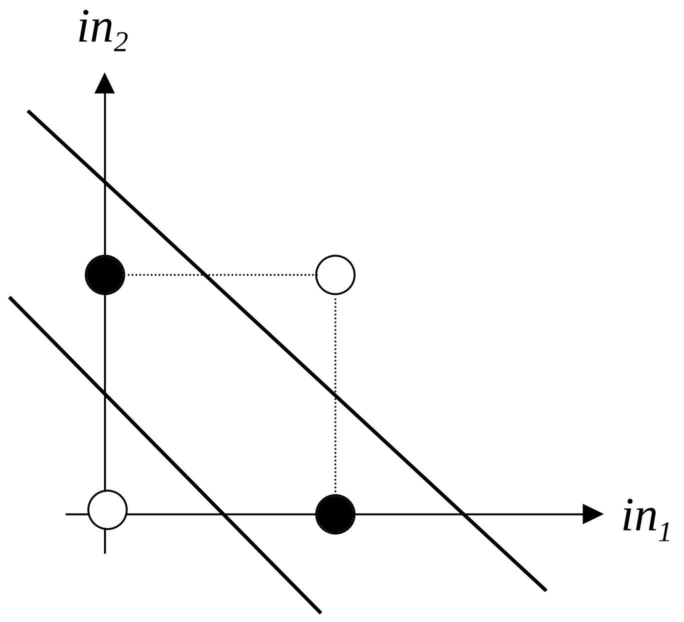
Learning in Multi-Layer Perceptrons, Back-Propagation

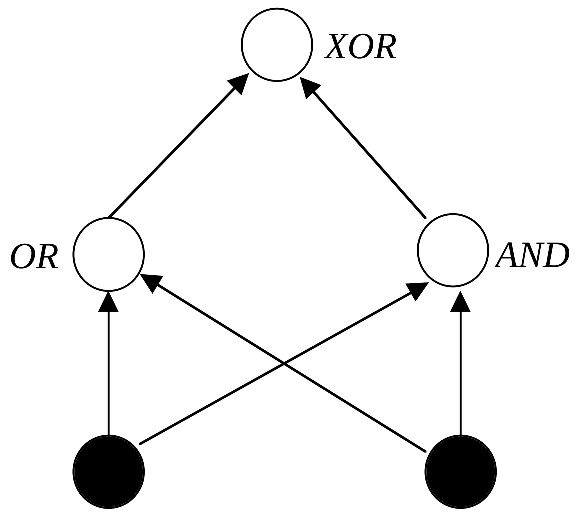
Introduction to Neural Networks : Lecture 7

XOR and Linear Separability Revisited

Remember that it is not possible to find weights that enable Single Layer Perceptrons to deal with non-linearly separable problems like XOR:



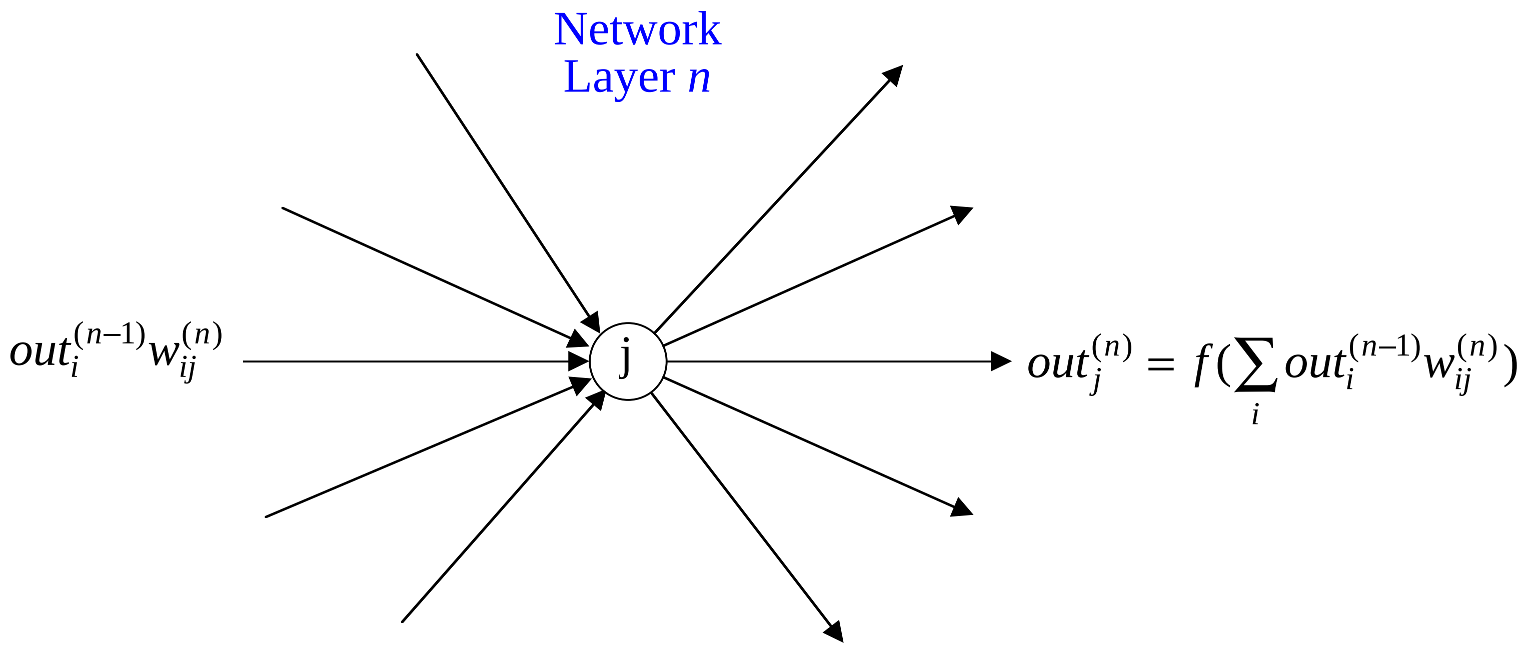
$\Rightarrow$ *e.g.*



However, Multi-Layer Perceptrons (MLPs) are able to cope with non-linearly separable problems. Historically, the problem was that there were no learning algorithms for training MLPs. Actually, it is now quite straightforward.

Notation for Multi-Layer Networks

Dealing with multi-layer networks is easy if we use a sensible notation. We simply need another label *(n)* to tell us which layer in the network we are dealing with:


Network
Layer n
$out_i^{(n-1)} w_{ij}^{(n)}$
j
$out_j^{(n)} = f(\sum_i out_i^{(n-1)} w_{ij}^{(n)})$

Each unit j in layer n receives activations $out_i^{(n-1)} w_{ij}^{(n)}$ from the previous layer of processing units and sends activations $out_j^{(n)}$ to the next layer of units.

Multi-Layer Perceptrons (MLPs)

Conventionally, the input layer is layer 0, and when we talk of an *N* layer network we mean there are *N* layers of weights and *N* non-input layers of processing units. Thus a two layer Multi-Layer Perceptron takes the form:

$$\begin{tikzcd}
\boxed{noutputs} \\
\boxed{nhidden} \arrow[u, "w_{jk}^{(2)}"'] \\
\boxed{ninputs} \arrow[u, "w_{ij}^{(1)}"']
\end{tikzcd}$$

$$out_k^{(2)} = f\Big(\sum_j out_j^{(1)} w_{jk}^{(2)}\Big)$$

$$out_j^{(1)} = f\Big(\sum_i out_i^{(0)} w_{ij}^{(1)}\Big)$$

$$out_i^{(0)} = in_i$$

It is clear how we can add in further layers, though for most practical purposes two layers will be sufficient. Note that there is nothing stopping us from having different activation functions *f(x)* for different layers, or even different units within a layer.

Learning in Multi-Layer Perceptrons

We can use the same ideas as before to train our N-layer neural networks. We want to adjust the network weights $w_{ij}^{(n)}$ in order to minimise the sum-squared error function

$$E(w_{ij}^{(n)}) = \tfrac{1}{2}\sum_p \sum_j \left(targ_j^p - out_j^{(N)}(in_i^p)\right)^2$$

and again we can do this by a series of gradient descent weight updates

$$\Delta w_{kl}^{(m)} = -\eta \frac{\partial E(w_{ij}^{(n)})}{\partial w_{kl}^{(m)}}$$

Note that it is only the outputs $out_j^{(N)}$ of the final layer that appear in the error function. However, the final layer outputs will depend on all the earlier layers of weights, and the learning algorithm will adjust them all.

The learning algorithm automatically adjusts the outputs $out_j^{(n)}$ of the earlier (hidden) layers so that they form appropriate intermediate (hidden) representations.

Computing the Partial Derivatives

For a two layer network, the final outputs can be written:

$$out_k^{(2)} = f\left(\sum_j out_j^{(1)}.w_{jk}^{(2)}\right) = f\left(\sum_j f\left(\sum_i in_i w_{ij}^{(1)}\right).w_{jk}^{(2)}\right)$$

We can then use the chain rules for derivatives, as for the Single Layer Perceptron, to give the derivatives with respect to the two sets of weights $w_{hl}^{(1)}$ and $w_{hl}^{(2)}$:

$$\frac{\partial E(w_{ij}^{(n)})}{\partial w_{hl}^{(m)}} = -\sum_p \sum_k \left(targ_k - out_k^{(2)}\right).\frac{\partial out_k^{(2)}}{\partial w_{hl}^{(m)}}$$

$$\frac{\partial out_k^{(2)}}{\partial w_{hl}^{(2)}} = f'\left(\sum_j out_j^{(1)} w_{jk}^{(2)}\right).out_h^{(1)}.\delta_{kl}$$

$$\frac{\partial out_k^{(2)}}{\partial w_{hl}^{(1)}} = f'\left(\sum_j out_j^{(1)} w_{jk}^{(2)}\right).f'\left(\sum_i in_i w_{il}^{(1)}\right).w_{lk}^{(2)}.in_h$$

Deriving the Back Propagation Algorithm

All we now have to do is substitute our derivatives into the weight update equations

$$\Delta w_{hl}^{(2)} = \eta \sum_p \left(targ_l - out_l^{(2)}\right).f'\left(\sum_j out_j^{(1)} w_{jl}^{(2)}\right).out_h^{(1)}$$

$$\Delta w_{hl}^{(1)} = \eta \sum_p \sum_k \left(targ_k - out_k^{(2)}\right).f'\left(\sum_j out_j^{(1)} w_{jk}^{(2)}\right).f'\left(\sum_i in_i w_{il}^{(1)}\right).w_{lk}^{(2)}.in_h$$

Then if the transfer function *f(x)* is a Sigmoid we can use $f'(x) = f(x).(1 - f(x))$ to give

$$\Delta w_{hl}^{(2)} = \eta \sum_p \left(targ_l - out_l^{(2)}\right).out_l^{(2)}.\left(1 - out_l^{(2)}\right).out_h^{(1)}$$

$$\Delta w_{hl}^{(1)} = \eta \sum_p \sum_k \left(targ_k - out_k^{(2)}\right).out_k^{(2)}.\left(1 - out_k^{(2)}\right).w_{lk}^{(2)}.out_l^{(1)}.\left(1 - out_l^{(1)}\right).in_h$$

These equations constitute the basic Back-Propagation Learning Algorithm.

Simplifying the Computation

When implementing the Back-Propagation algorithm it is convenient to define

$$delta_k^{(2)} = \left(targ_k - out_k^{(2)}\right).f'\left(\sum_j out_j^{(1)} w_{jk}^{(2)}\right) = \left(targ_k - out_k^{(2)}\right).out_k^{(2)}.\left(1 - out_k^{(2)}\right)$$

which is a generalised output deviation. We can then write the weight update rules as

$$\Delta w_{hl}^{(2)} = \eta \sum_p delta_l^{(2)}.out_h^{(1)}$$

$$\Delta w_{hl}^{(1)} = \eta \sum_p \left(\sum_k delta_k^{(2)}.w_{lk}^{(2)}\right).out_l^{(1)}.\left(1 - out_l^{(1)}\right).in_h$$

So the weight $w_{hl}^{(2)}$ between units h and l is changed in proportion to the output of unit h and the *delta* of unit l. The weight changes at the first layer now take on the same form as the final layer, but the ‘error’ at each unit l is ***back-propagated*** from each of the output units k via the weights $w_{lk}^{(2)}$.

Networks With Any Number of Hidden Layers

It is now becoming clear that, with the right notation, it is easy to extend our gradient descent algorithm to work for any number of hidden layers. We define

$$delta_k^{(N)} = \left(targ_k - out_k^{(N)}\right).f'\left(\sum_j out_j^{(1)} w_{jk}^{(N)}\right) = \left(targ_k - out_k^{(N)}\right).out_k^{(N)}.\left(1 - out_k^{(N)}\right)$$

as the delta for the output layer, and then back-propagate the deltas to earlier layers using

$$delta_k^{(n)} = \left(\sum_k delta_k^{(n+1)}.w_{lk}^{(n+1)}\right).f'\left(\sum_j out_j^{(n-1)} w_{jk}^{(n)}\right) = \left(\sum_k delta_k^{(n+1)}.w_{lk}^{(n+1)}\right).out_k^{(n)}.\left(1 - out_k^{(n)}\right)$$

Then each weight update equation can be written as:

$$\Delta w_{hl}^{(n)} = \eta \sum_p delta_l^{(n)}.out_h^{(n-1)}$$

Suddenly the Back-Propagation Algorithm looks very simple and easily programmable!

The Cross Entropy Error Function

We have talked about using the ***Sum Squared Error (SSE)*** function as a measure of network performance, and as a basis for gradient descent learning algorithms. A useful alternative for classification networks is the ***Cross Entropy (CE)*** error function

$$E_{ce}(w_{ij}) = -\sum_p \sum_j \left[targ_j^p . \log\left(out_j(in_i^p)\right) + \left(1 - targ_j^p\right) . \log\left(1 - out_j(in_i^p)\right) \right]$$

This is appropriate if we want to interpret the network outputs as probabilities, and has several advantages over the *SSE* function. When we compute the partial derivatives for the gradient descent weight update equations, the sigmoid derivative cancels out leaving

$$\Delta w_{hl}^{(N)} = \eta \sum_p \left(targ_l - out_l^{(N)} \right) . out_h^{(N-1)}$$

which is easier to compute than the *SSE* equivalent, and no longer has the property of going to zero when the outputs are totally wrong (so no need for offsets, etc.).

The Need For Non-Linearity

We have noted that if the network outputs are non-binary, then it is appropriate to have a linear output activation function rather than a Sigmoid. So why not use linear activation functions on the hidden layers as well?

Recall that for an activation functions $f^{(n)}(x)$ at layer n, the outputs are given by

$$out_k^{(2)} = f^{(2)}\left(\sum_j out_j^{(1)}.w_{jk}^{(2)}\right) = f^{(2)}\left(\sum_j f^{(1)}\left(\sum_i in_i w_{ij}^{(1)}\right).w_{jk}^{(2)}\right)$$

so if the hidden layer activation is linear, i.e. $f^{(1)}(x) = x$, this simplifies to

$$out_k^{(2)} = f^{(2)}\left(\sum_i in_i.\left(\sum_j w_{ij}^{(1)} w_{jk}^{(2)}\right)\right)$$

But this is equivalent to a single layer network with weights $w_{ik} = \sum_j w_{ij}^{(1)} w_{jk}^{(2)}$ and we know that such a network cannot deal with non-linearly separable problems.

Training a Two-Layer Feed-forward Network

The training procedure for two layer networks is similar to that for single layer networks:

1. Take the set of training patterns you wish the network to learn

   $\{in_i^p, out_j^p : i = 1 \ldots ninputs, j = 1 \ldots noutputs, p = 1 \ldots npatterns\}$.

2. Set up your network with *ninputs* input units fully connected to *nhidden* non-linear hidden units via connections with weights $w_{ij}^{(1)}$, which in turn are fully connected to *noutputs* output units via connections with weights $w_{jk}^{(2)}$.

3. Generate random initial weights, e.g. from the range [–*smwt*, +*smwt*]

4. Select an appropriate error function $E(w_{jk}^{(n)})$ and learning rate η.

5. Apply the weight update equation $\Delta w_{jk}^{(n)} = -\eta \partial E(w_{jk}^{(n)}) / \partial w_{jk}^{(n)}$ to each weight $w_{jk}^{(n)}$ for each training pattern p. One set of updates of all the weights for all the training patterns is called one *epoch* of training.

6. Repeat step 5 until the network error function is ‘small enough’.

The extension to networks with more hidden layers should be obvious.

Practical Considerations for Back-Propagation Learning

Most of the practical considerations necessary for general Back-Propagation learning were already covered when we talked about training single layer Perceptrons:

1. Do we need to pre-process the training data? If so, how?
2. How do we choose the initial weights from which we start the training?
3. How do we choose an appropriate learning rate η?
4. Should we change the weights after each training pattern, or after the whole set?
5. Are some activation/transfer functions better than others?
6. How can we avoid flat spots in the error function?
7. How can we avoid local minima in the error function?
8. How do we know when we should stop the training?

However, there are also two important issues that were not covered before:

9. How many hidden units do we need?
10. Should we have different learning rates for the different layers?

How Many Hidden Units?

The best number of hidden units depends in a complex way on many factors, including:

1. The number of training patterns
2. The numbers of input and output units
3. The amount of noise in the training data
4. The complexity of the function or classification to be learned
5. The type of hidden unit activation function
6. The training algorithm

Too few hidden units will generally leave high training and generalisation errors due to under-fitting. Too many hidden units will result in low training errors, but will make the training unnecessarily slow, and will result in poor generalisation unless some other technique (such as *regularisation*) is used to prevent over-fitting.

Virtually all "rules of thumb" you hear about are actually nonsense. A sensible strategy is to try a range of numbers of hidden units and see which works best.

Different Learning Rates for Different Layers?

A network as a whole will usually learn most efficiently if all its neurons are learning at roughly the same speed. So maybe different parts of the network should have different learning rates η. There are a number of factors that may affect the choices:

1. The later network layers (nearer the outputs) will tend to have larger local gradients (*deltas*) than the earlier layers (nearer the inputs).
2. The activations of units with many connections feeding into or out of them tend to change faster than units with fewer connections.
3. Activations required for linear units will be different for Sigmoidal units.
4. There is empirical evidence that it helps to have different learning rates η for the thresholds/biases compared with the real connection weights.

In practice, it is often quicker to just use the same rates η for all the weights and thresholds, rather than spending time trying to work out appropriate differences. A very powerful approach is to use evolutionary strategies to determine good learning rates.

Overview and Reading

1. We started by revisiting the concept of linear separability and the need for multi-layered neural networks.
2. We then saw how the Back-Propagation Learning Algorithm for multi-layered networks could be derived ‘easily’ from the standard gradient descent approach.
3. We ended by looking at some practical issues that didn’t arise for the single layer networks.

Reading

1. Gurney: Sections 6.1, 6.2, 6.3, 6.4
2. Haykin: Sections 4.1, 4.2, 4.3, 4.4, 4.5, 4.6
3. Beale & Jackson: Sections 4.1, 4.2, 4.3, 4.4, 4.5
4. Bishop: Sections 4.1, 4.8
5. Callan: Sections 2.2, 2.3, 2.4, 2.5