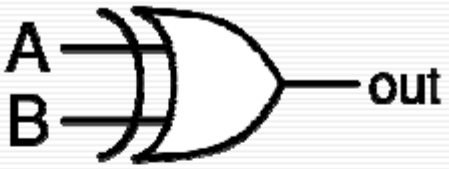
MULTI LAYER PERCEPTRON

Overview

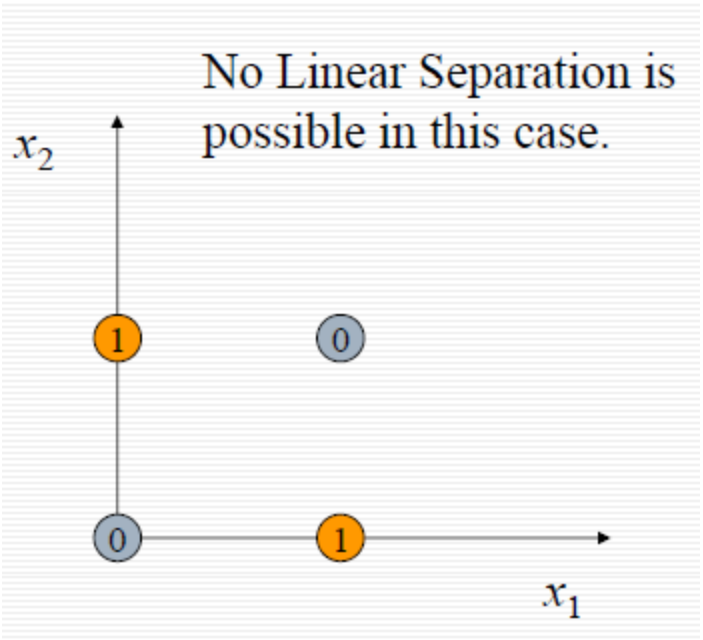
- Simple perceptrons can be trained to solved simple linearly separable problems.
- Composite “multi-layer” perceptrons can potentially solve non-linearly separable problems.
- However, the perceptron training algorithm (as described) does not generalize to training systems containing several neurons.

Linear Separability

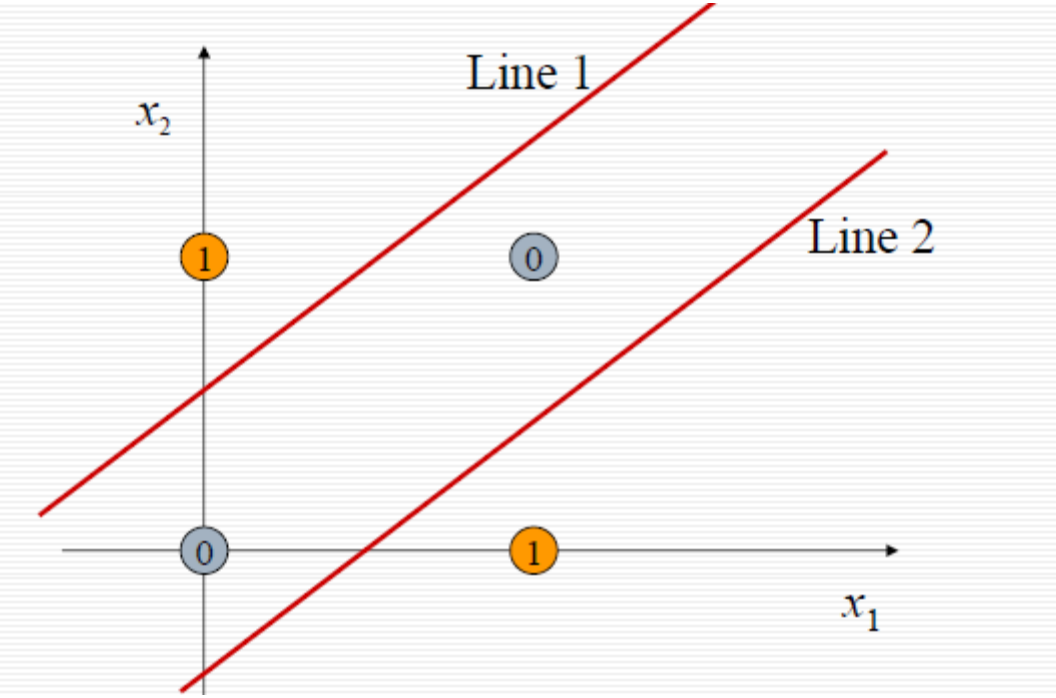
- Consider an XOR gate:



A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0



Two lines solves XOR problem



Composite Perceptron

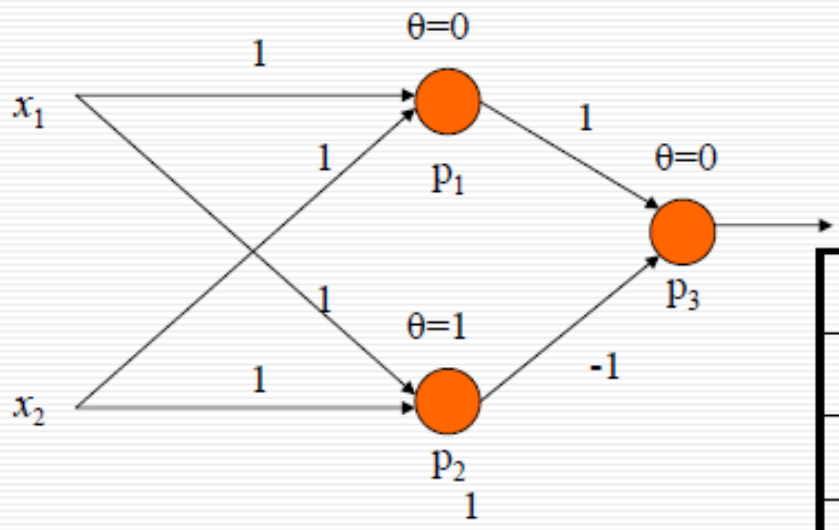
We can represent each line with a perceptron

We can then write down the following algorithm:

> if the point lies below line 1 AND above line 2 then it is in class 0
>
> else the point is in class 1

This algorithm is in itself linearly separable and can be represented by a third perceptron, which takes its inputs from the outputs of the other two perceptrons.

Perceptron which solves XOR



x_1	x_2	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

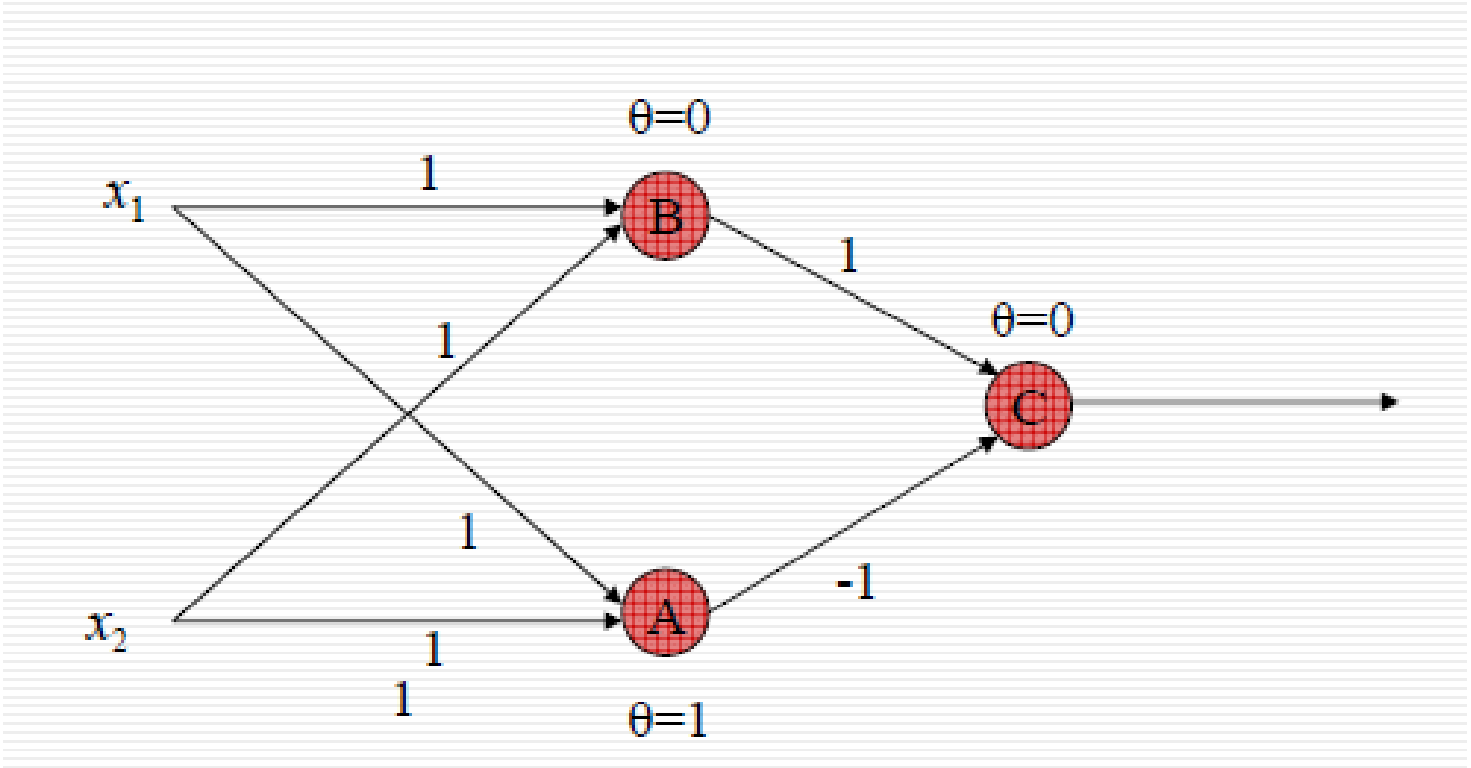
No Learning

- Although I presented a solution to the XOR problem, I didn't say where it came from…
- The problem is that it wasn't derived automatically, it was thought out by hand
- i.e. it wasn't learnt by the network
- The whole point about neural networks is that we want them to learn!
- Otherwise we could just as well write an ordinary program in the first place

Credit Assignment Problem

- Why does the anti-Hebbian learning algorithm presented earlier not work?
- Because of the “credit assignment problem”
- The neurons in the network have either an output or a input which is “inside” the neural network, i.e. not immediately comparable to input or output.
- The algorithm doesn’t specify how to solve this situation.
- It’s not clear how to divide the “global” error for the network up into a “local” error for each individual neuron

A multi layer perceptron



Idea

- An idea for solving the credit assignment problem is to adjust the weights on each neuron in turn
- Suppose we try adjusting neuron C. The problem is that we don't know whether the inputs coming from A and B are "right" or not.
- Suppose we try adjusting neuron A or B. Again we don't know whether the outputs from A or B are right or wrong.
- This is the "credit assignment problem".

Where is the problem?

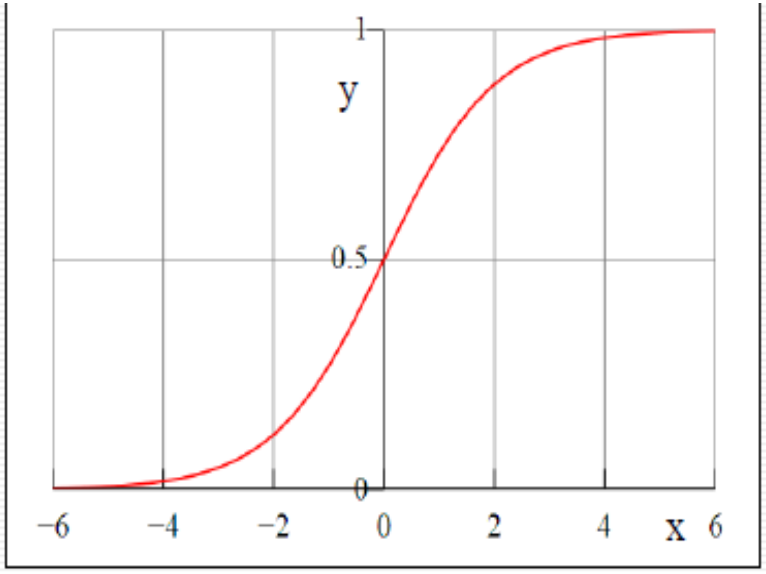
- The problem lies in the threshold function
- Because the neurons fire “all or nothing”, it means that for many changes in the weights, nothing changes (with respect to the other units)
- No information is transmitted
- This leads to an attempt to sub-optimize each individual perceptron, but this does not yield an overall solution.

Possible Solution

- A possible solution is to use a “smooth” transfer function, rather than the step function.

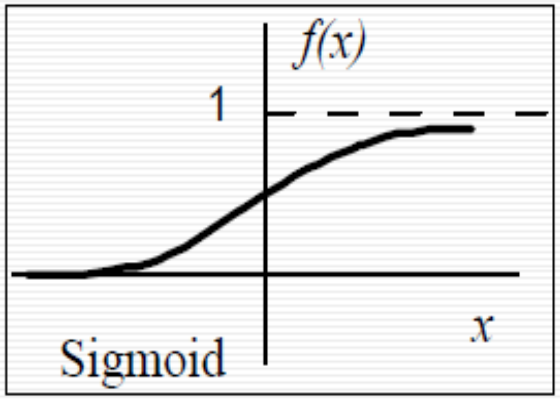
- In particular, the function has to be:
  - The same general shape as the step function.
  - Differentiable at all points.
  - Monotonic.
  - Easy to work with (i.e. not too complicated)
  - A possible candidate is the sigmoid (or logistic) function.

Sigmoid Function



- ✓As x goes to minus infinity, y goes to 0 (tends not to fire).
- ✓As x goes to infinity, y goes to 1 (tends to fire).
- ✓At x=0, y=1/2

$$y = f(x) = \frac{1}{1+e^{-x}}$$



Properties of Sigmoid Function

- The Sigmoid function can be used to approximate a step function
- It is monotonic and simple to work with.
- Sigmoid function is the solution of the first-order non linear differential equation.

$$y = \frac{1}{1+e^{-x}}$$

$$\frac{dy}{dx} = -1(1+e^{-x})^{-2}(-1)e^{-x}$$

$$= \frac{e^{-x}}{\left(1+e^{-x}\right)^2}$$

Note that:

$$1-y = 1-\frac{1}{1+e^{-x}}$$

$$= \frac{1+e^{-x}-1}{1+e^{-x}} = \frac{e^{-x}}{1+e^{-x}}$$

$$\therefore \frac{dy}{dx} = \frac{1}{(1+e^{-x})} \cdot \frac{e^{-x}}{(1+e^{-x})} = y(1-y)$$

Building a network

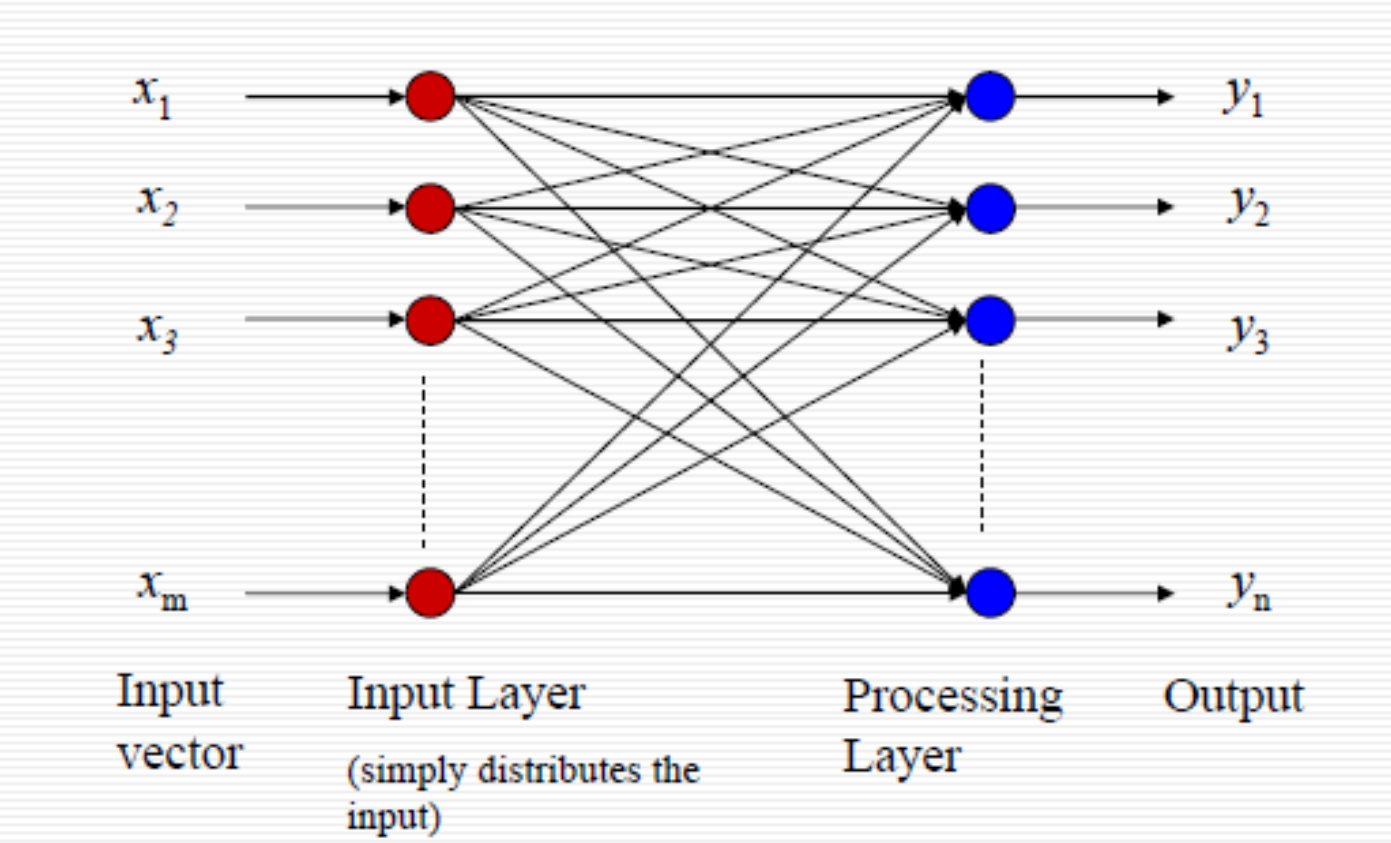
- We are now ready to start building a network composed of these “new and improved” neurons.

- Assume we have a set of *m*-D input patterns $\mathbf{x}_1 \ldots \mathbf{x}_M$ that we wish to associate with a set of *n*-D desired patterns $\mathbf{z}_1 \ldots \mathbf{z}_N$
  - Note that this means we have *N* patterns in our trainin set in total.

- By *associate* we could think of several tasks which need a generalized mapping between input and output

- Classification, prediction, auto-association, etc.

A General Layered Network



MLP Networks

- MLP networks is a NN which consists of
  - A set of sensory units (source nodes) that constitute the *input layer,*
  - One or more *hidden layers* of computation nodes,
  - An *output layer* of computation nodes.

- The input signal propagates through the network in a forward direction, on a layer-by-layer basis.

- They have been applied successfully to solve some difficult and diverse problems by training them in a supervised manner with a highly popular algorithm known as the *error back-propagation algorithm.*

- This algorithm is based on the *error-correction learning rule.*

Characteristics of MLP

- The model of each neuron in the network includes a *nonlinear activation function.*
  - The same transfer function for all neurons in a layer.

- The network contains one or more layers of *hidden neurons* that are not part of the input or output of the network.
  - These hidden neurons enable the network to learn complex tasks by extracting progressively more meaningful features from the input patterns (vectors).

- The network exhibits a high degrees of *connectivity,* determined by the synapses of the network.

Characteristics of MLP

- Notice that building the network in layers like this implies several important restrictions:
  - No circular (feedback) paths in the network.
  - Complete interconnection between layers.
  - Synchronous timing (all units in a layer fire simultaneously)
- These assumptions are not realistic for a real brain, but are made for mathematical convenience.
- Later we will relaxe some of the restrictions

Back Propagation

- Error back-propagation learning consists of two passes through the different layers of the network:
  - Forward Pass:
    - An input vector is applied to the sensory nodes of the network.
    - Its effect propagates through the network layer by layer.
    - A set of outputs is produced as the actual response of the network.
    - During this pass the synaptic weights of the networks are all *fixed.*
  - Backward Pass:
    - An error signal is calculated and propagated backward through the network against the direction of synaptic connections.
    - The synaptic weights are adjusted to make the actual response of the network move closer to the desired response in a statistical sense.
    - During this pass, the synaptic weights are all *adjusted* in accordance with an error-correction rule.

Types of Signals in MLP

- Function signal
  - It is the input signal that comes in at the input end of the network
  - It propagates forward through the network, and emerges at the output end of the network as an output signal.
- Error signal
  - An error signal originates at an output neuron of the network
  - It propagates backward through the network.
  - It is called as "error signal" because its computation by every neuron of the network involves an error-dependent function.

The need for a metric

- All the connections have weights associated with them
  - In the beginning, the weights are allocated random values.
  - Thus at the beginning, the network will not perform a useful function
    - i.e. the outputs of the network $\mathbf{y}_1 \ldots \mathbf{y}_N$ in response to input patterns $\mathbf{x}_1 \ldots \mathbf{x}_N$ will not be anything like equal to what we want, which is $\mathbf{z}_1 \ldots \mathbf{z}_N$
  - But what we can do is to compare the output with the desired output and form a *metric* which measures the error.

Back Propagation

- The sigmoid function is used which is defined as follows:
  - $y = \frac{1}{1 + e^{-x}}$
  - dy/dx = y(1-y) as proved before
- As with the single perceptron, the back propagation algorithm starts by initializing the weights in the network to random values, which are usually set to small values, say in the range of -0.5 to 0.5
- Each iteration of the algorithm involves first feeding data through the network from the inputs to the outputs.
- The next phase, which gives the algorithm its name, involves feeding errors back from the outputs to the inputs

Back Propagation

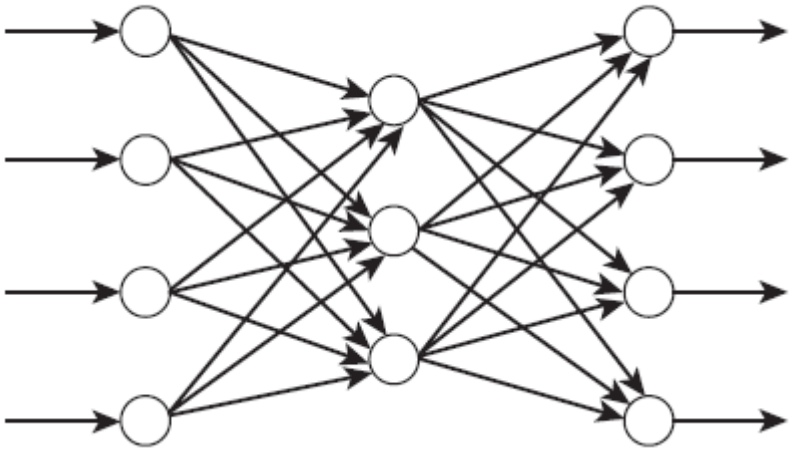
- These error values feed back through the network, making changes to the weights of nodes along the way

- The algorithm repeats in this way until the outputs produced for the training data are sufficiently close to the desired values—in other words, until the error values are sufficiently small

- Because the sigmoid function cannot actually reach 0 or 1, it is usual to accept a value such as 0.9 as representing 1 and 0.1 as representing 0

Back Propagation

- We will consider a network of three layers and will use *i* to represent nodes in the input layer, *j* to represent nodes in the hidden layer, and *k* to represent nodes in the output layer



- “i” “j” “k”

Back Propagation

- The function that is used to derive the output value for a node *j* in the network is as follows

$$X_j = \sum_{i=1}^{n} x_i \cdot w_{ij} - \theta_j$$

$$Y_j = \frac{1}{1 + e^{-X_j}}$$

- where *n* is the number of inputs to node *j*
- w_{ij} is the weight of the connection between each node *i* and node *j*
- Θ_j is the threshold value being used for node *j*
- x_i is the input value for input node *I*
- y_j is the output value produced by node *j*

Back Propagation

- Once the inputs have been fed through the network to produce outputs, an **error gradient** is calculated for each node *k* in the output layer.

- The error signal for k is defined as the difference between the desired value and the actual value for that node:

$$e_k = d_k - y_k$$

  - Where d_k is the desired value for node *k*, and y_k is the actual value

- The error gradient for output node *k* is defined as the error value for this node multiplied by the derivative of the activation function:

$$\delta_k = \frac{\partial y_k}{\partial x_k} \cdot e_k$$

Back Propagation

$$\delta_k = \frac{\partial y_k}{\partial x_k} \cdot e_k$$

- x_k is the weighted sum of the input values to the node *k*
- Because *y* is defined as a sigmoid function of *x*, therefore

$$\delta_k = y_k \cdot (1 - y_k) \cdot e_k$$

- Similarly, we calculate an error gradient for each node *j* in the hidden layer, as follows:

$$\delta_j = y_j \cdot (1 - y_j) \sum_{k=1}^{n} w_{jk} \delta_k$$

- where *n* is the number of nodes in the output layer, and thus the number of outputs from each node in the hidden layer

Back Propagation

- Each weight in the network, w_{ij} or w_{jk}, is updated according to the following formula:

$$w_{ij} \leftarrow w_{ij} + \alpha \cdot x_i \cdot \delta_j$$

$$w_{jk} \leftarrow w_{jk} + \alpha \cdot y_j \cdot \delta_k$$

- where x_i is the input value to input node i, and is the learning rate, which is a positive number below 1, and which should not be too high.

- This method is known as **gradient descent**
- As it involves following the steepest path down the surface that represents the error function to attempt to find the minimum in the error space
- It then represents the set of weights that provides the best performance of the network

Back Propagation- Termination

- The iterations of the back propagation algorithm is usually terminated when the sum of the squares of the errors of the output values for all training data in an epoch is less than some threshold, such as 0.001.