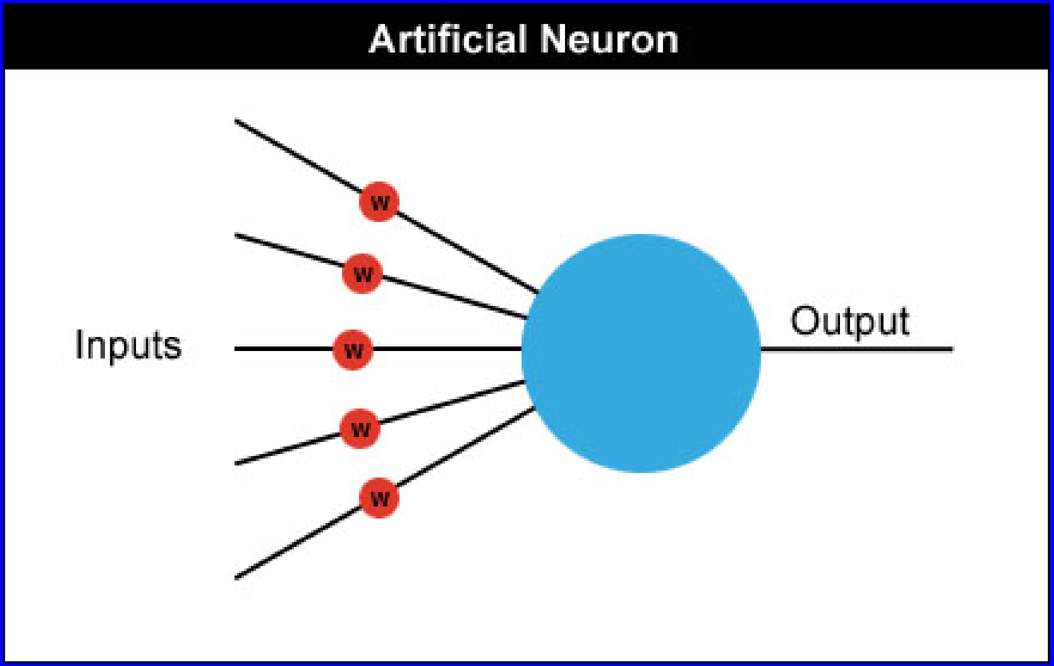
Finger Biometric
-- Neural networks

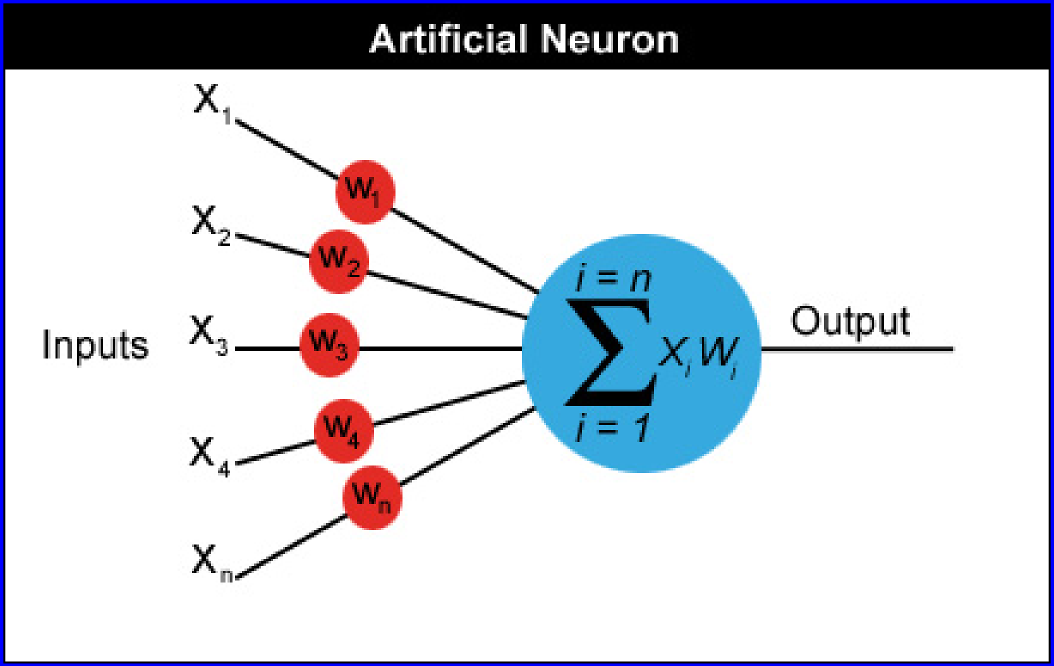
Neural networks

- Neural networks are made up of many artificial neurons.
- Each input into the neuron has its own weight associated with it illustrated by the red circle.
- A weight is simply a floating point number and it's these we adjust when we eventually come to train the network.



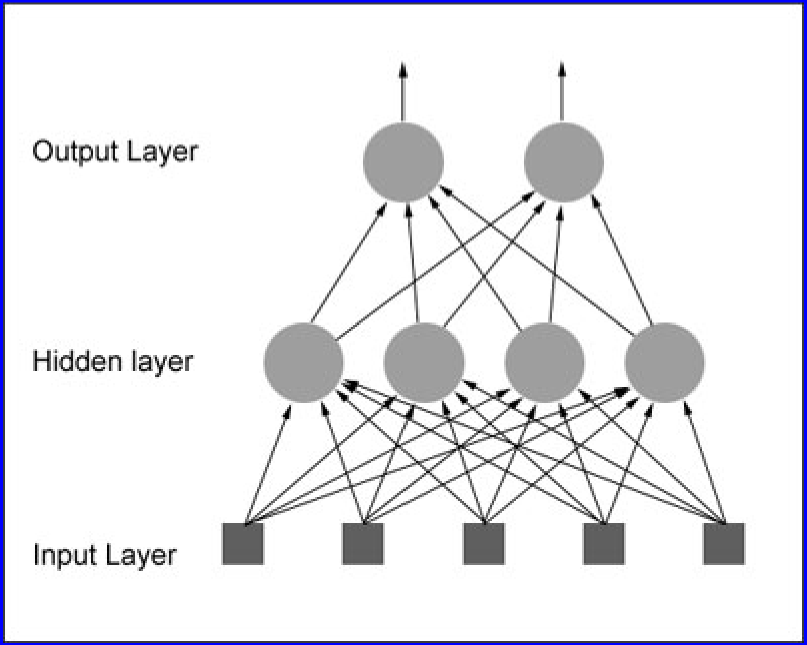
Neural networks

- A neuron can have any number of inputs from one to n, where n is the total number of inputs.
- The inputs may be represented therefore as x_1, x_2, x_3… x_n.
- And the corresponding weights for the inputs as w_1, w_2, w_3… w_n.
- Output $a = x_1w_1+x_2w_2+x_3w_3\ldots +x_nw_n$



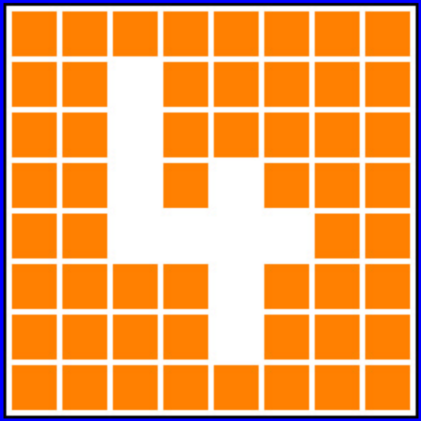
How do we actually *use* an artificial neuron?

- Feedforward network: The neurons in each layer feed their output forward to the next layer until we get the final output from the neural network.
- There can be any number of hidden layers within a feedforward network.
- The number of neurons can be completely arbitrary.



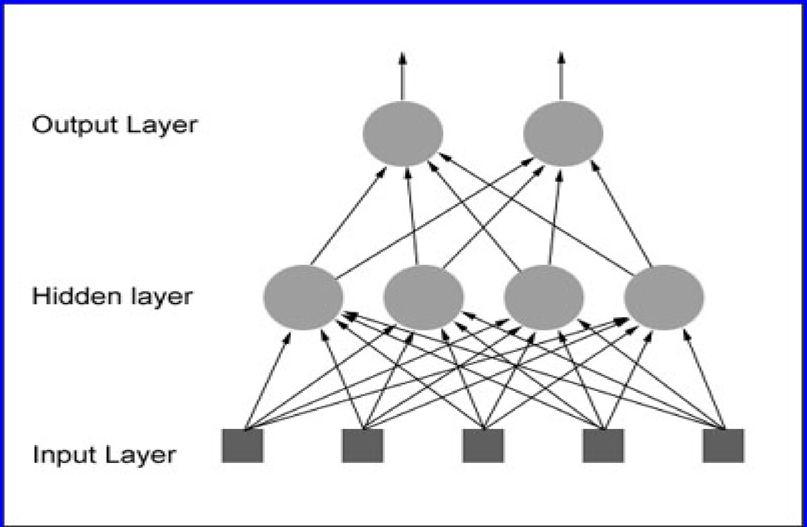
Neural Networks by an Example

- let's design a neural network that will detect the number '4'.
- Given a panel made up of a grid of lights which can be either on or off, we want our neural net to let us know whenever it thinks it sees the character '4'.
- The panel is eight cells square and looks like this:
- the neural net will have 64 inputs, each one representing a particular cell in the panel and a hidden layer consisting of a number of neurons (more on this later) all feeding their output into just one neuron in the output layer

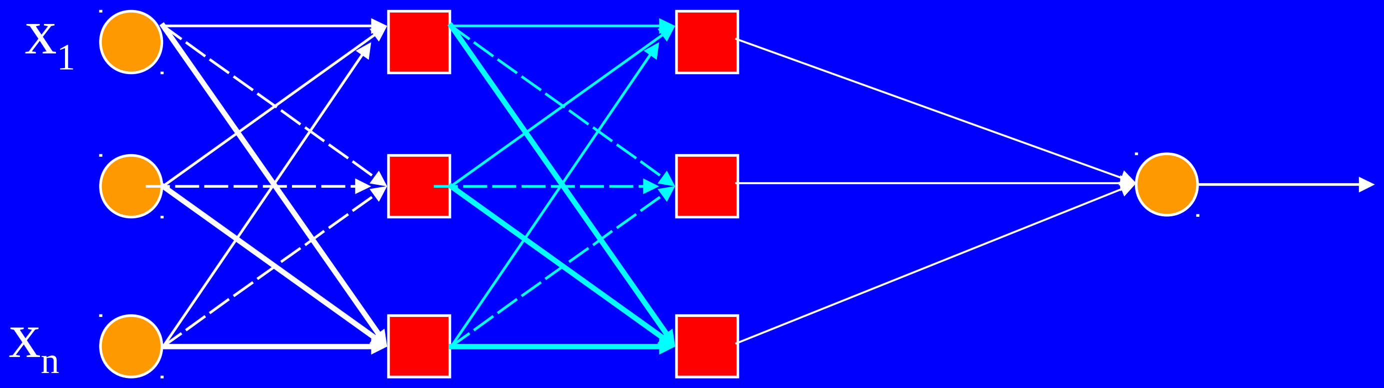


Neural Networks by an Example

- initialize the neural net with random weights
- feed it a series of inputs which represent, in this example, the different panel configurations
- For each configuration we check to see what its output is and <u>adjust the weights accordingly </u>so that whenever it sees something looking like a number 4 it outputs a 1 and for everything else it outputs a zero.
- More: http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html



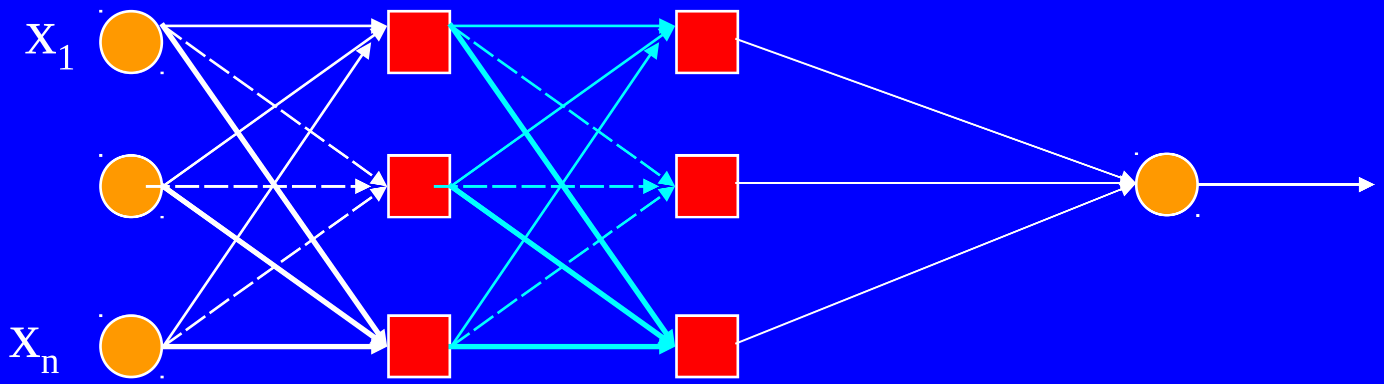
Multi-Layer Perceptron (MLP)



We will introduce the MLP and the backpropagation algorithm which is used to train it

MLP used to describe any general feedforward (no recurrent connections) network

However, we will concentrate on nets with units arranged in layers



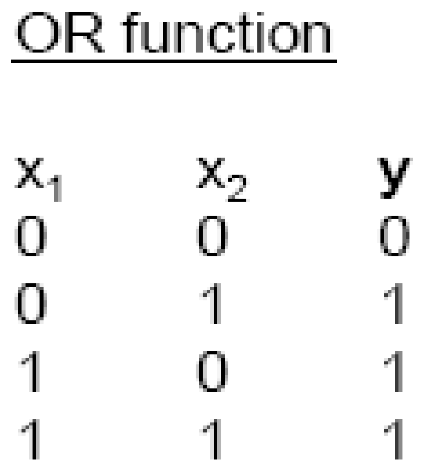
Different books refer to the above as either 4 layer (no. of layers of neurons) or 3 layer (no. of layers of adaptive weights). We will follow the latter convention

1st question:

what do the extra layers gain you? Start with looking at what a single layer can't do

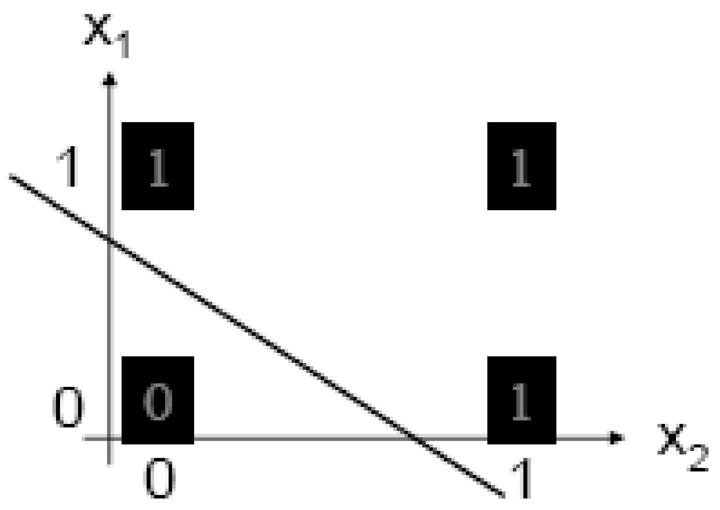
Perceptron Learning Theorem

- *Recap*: A perceptron (threshold unit) can *learn* anything that it can *represent* (i.e. anything separable with a hyperplane)



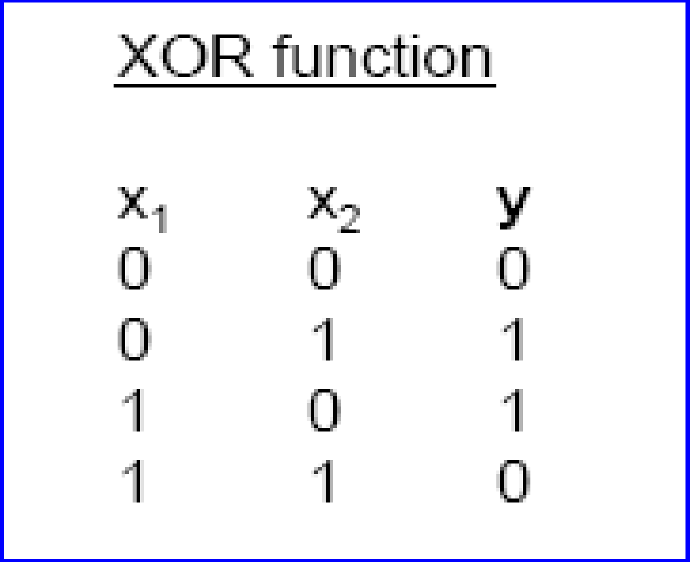
OR function

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



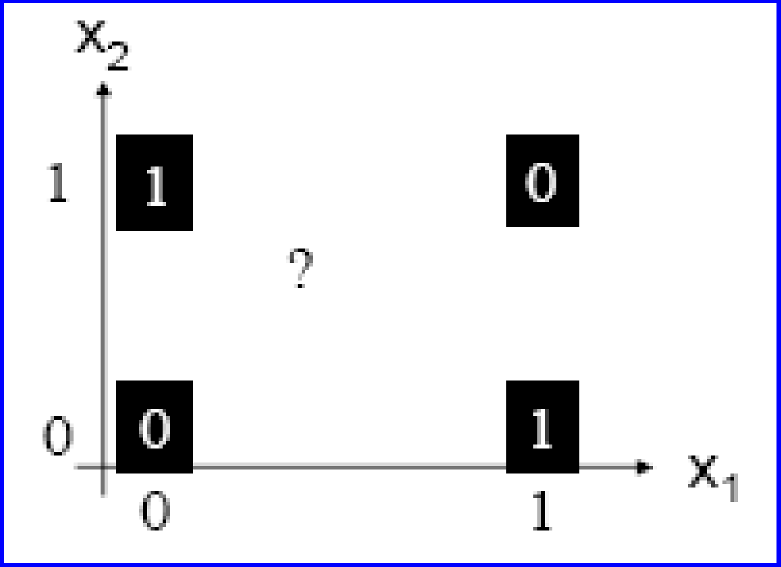
The Exclusive OR problem

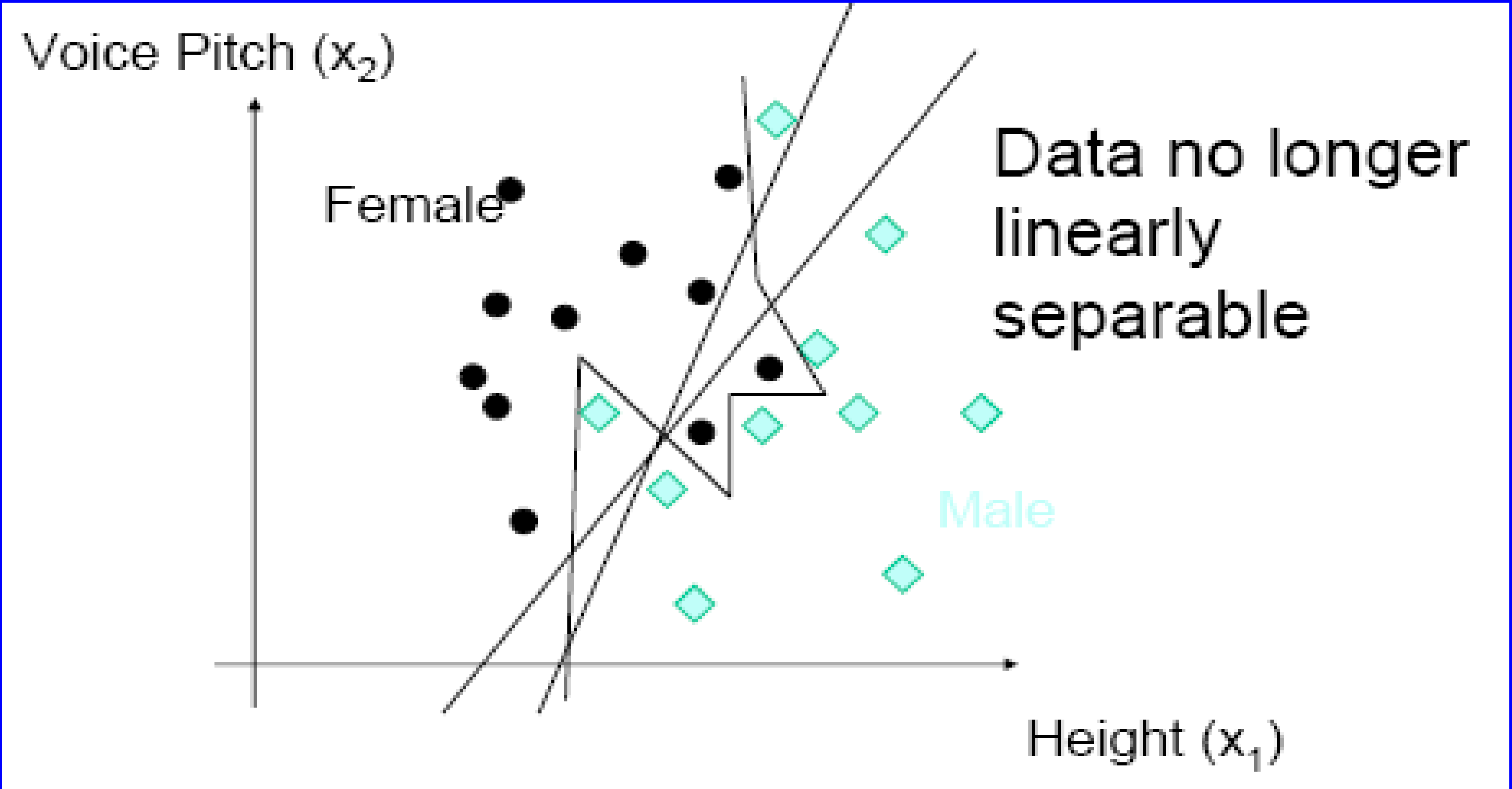
A Perceptron cannot represent Exclusive OR since it is not linearly separable.



XOR function

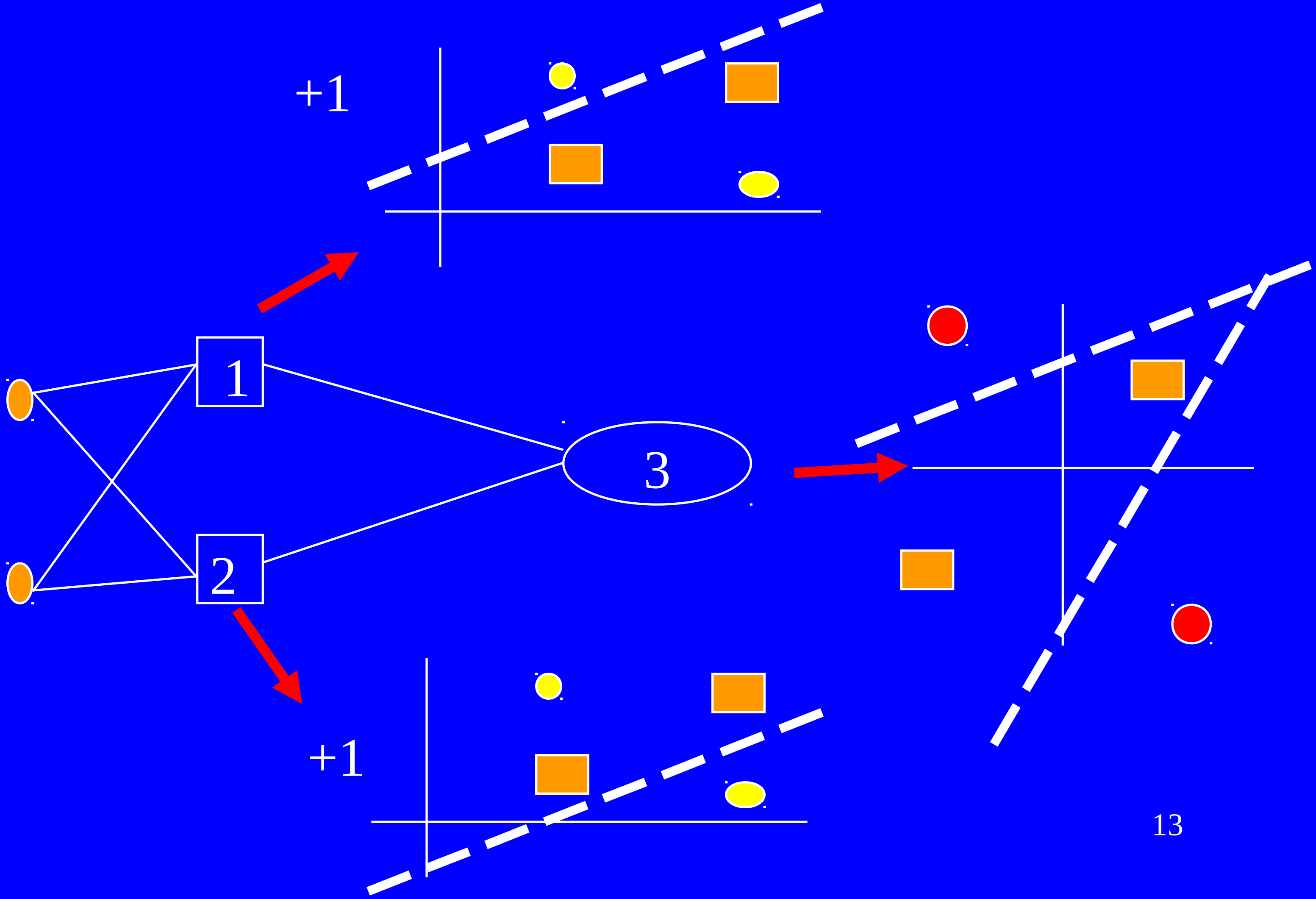
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



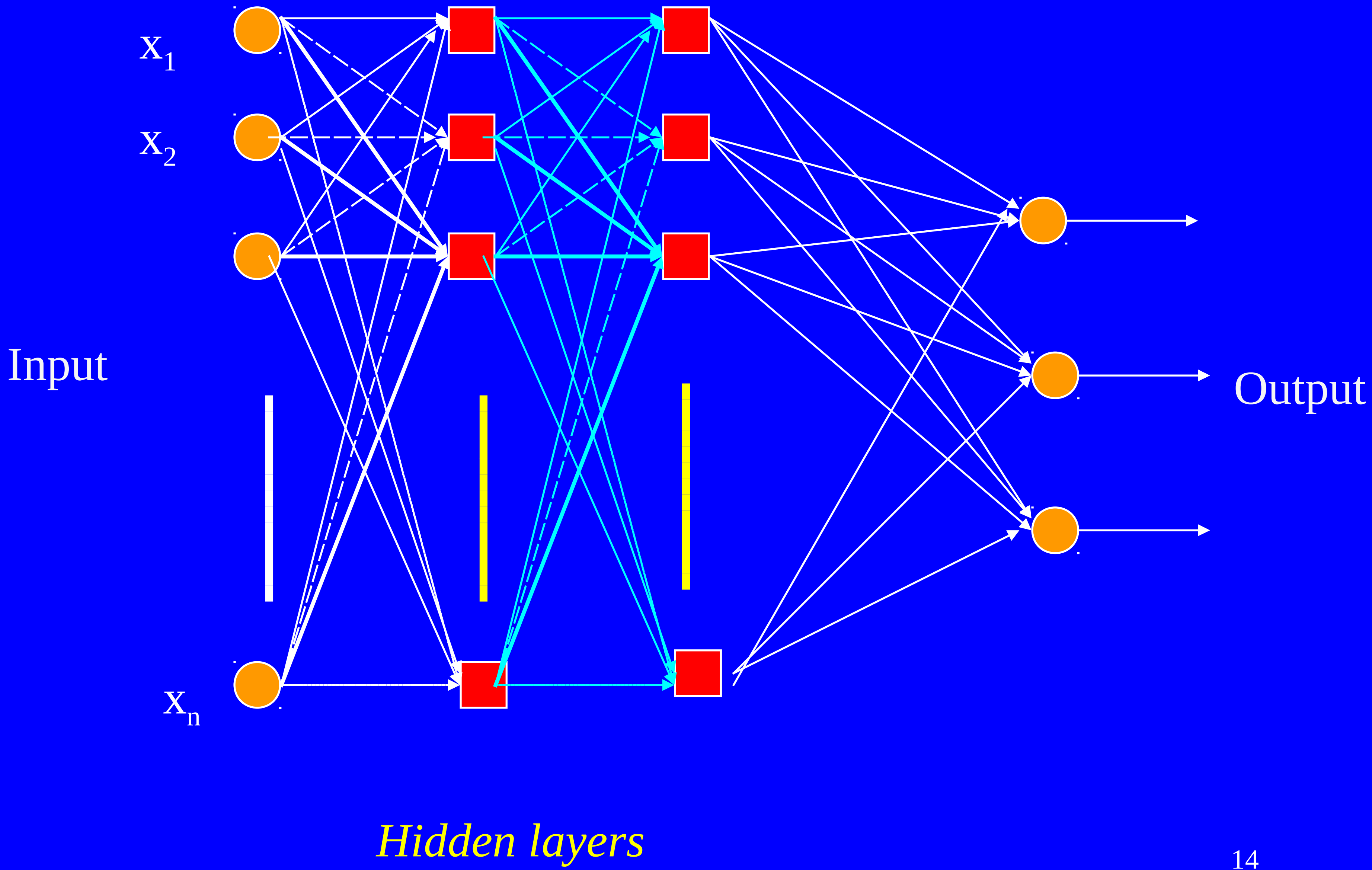


What is a good decision boundary ?

Minsky & Papert (1969) offered solution to XOR problem by combining perceptron unit responses using a second layer of Units. Piecewise linear classification using an MLP with threshold (perceptron) units

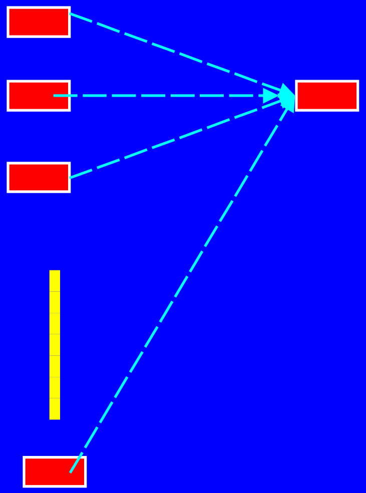


Three-layer networks



Properties of architecture

- No connections within a layer

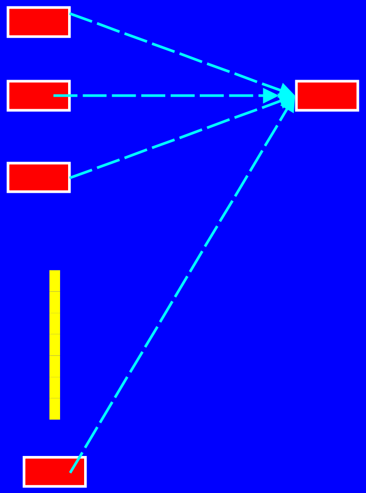


Each unit is a perceptron

$$y_i = f\left(\sum_{j=1}^{m} w_{ij} x_j + b_i\right)$$

Properties of architecture

- No connections within a layer
- No direct connections between input and output layers
-

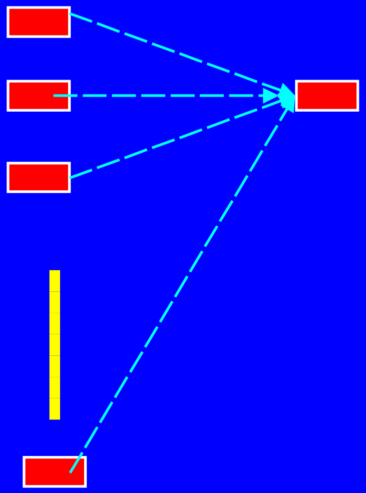


Each unit is a perceptron

$$y_i = f\left(\sum_{j=1}^{m} w_{ij} x_j + b_i\right)$$

Properties of architecture

- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers
-

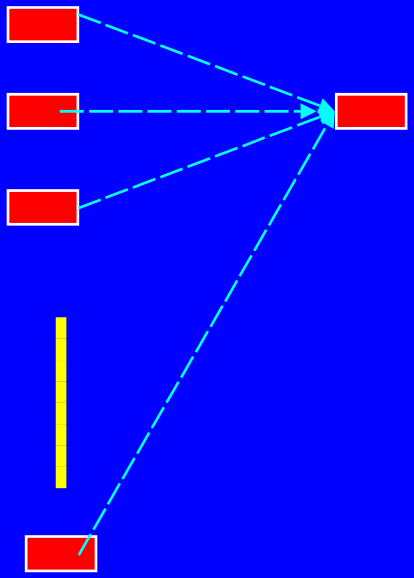


Each unit is a perceptron

$$y_i = f\left(\sum_{j=1}^{m} w_{ij} x_j + b_i\right)$$

Properties of architecture

- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers
- Often more than 3 layers
- Number of output units need not equal number of input units
- Number of hidden units per layer can be more or less than input or output units

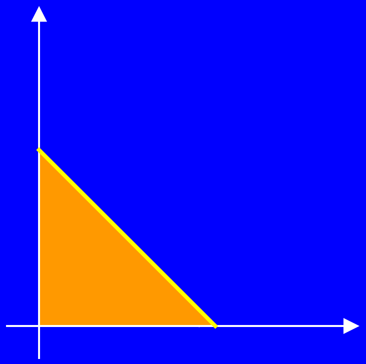


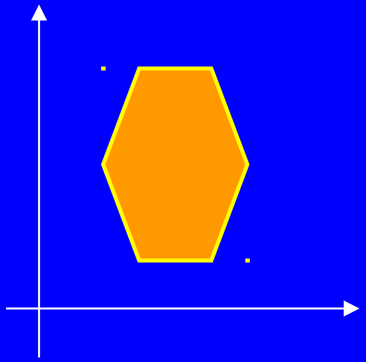
Each unit is a perceptron

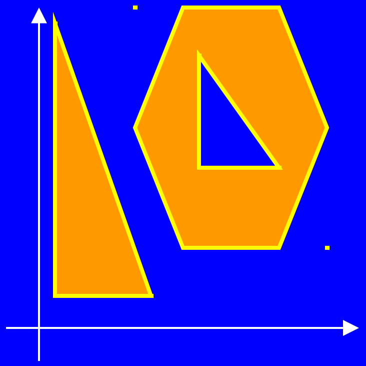
$$y_i = f\left(\sum_{j=1}^{m} w_{ij} x_j + b_i\right)$$

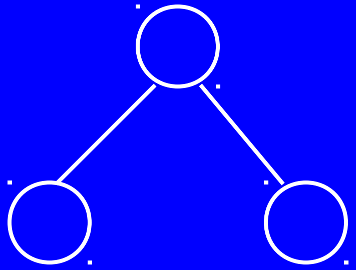
Often include bias as an extra weight

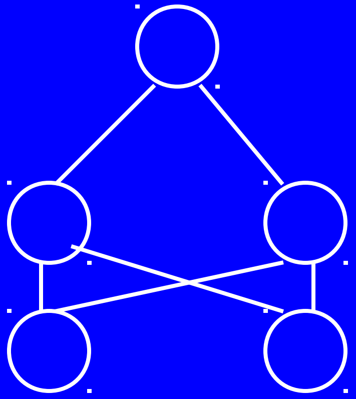
What do each of the layers do?

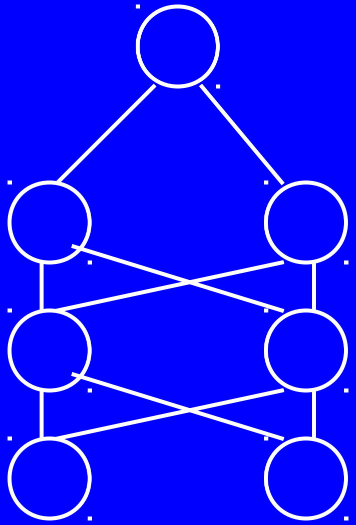












1st layer draws linear boundaries

2nd layer combines the boundaries

3rd layer can generate arbitrarily complex boundaries

Backpropagation learning algorithm ‘BP’

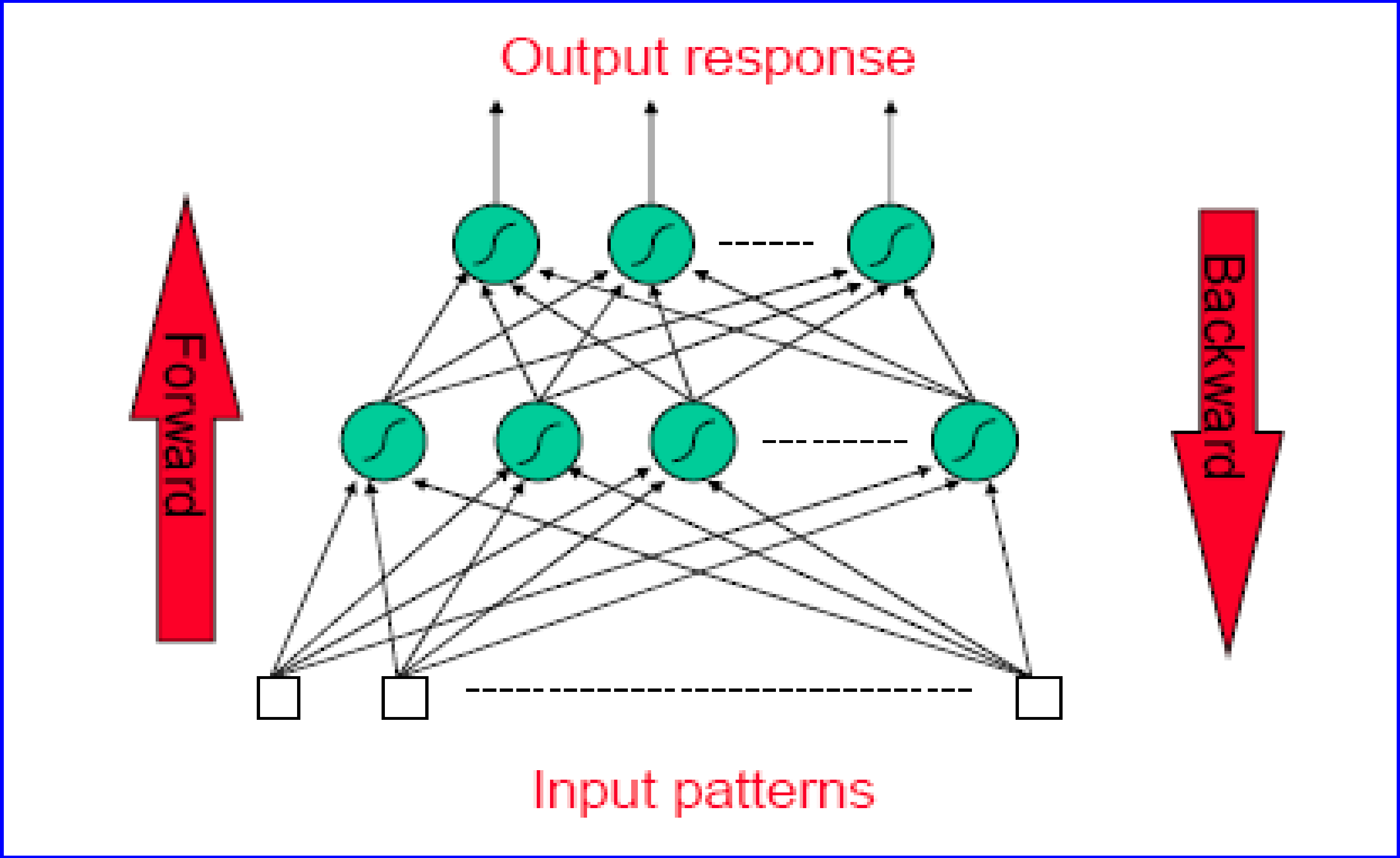
Solution to credit assignment problem in MLP. *Rumelhart, Hinton and Williams (1986) (*though actually invented earlier in a PhD thesis relating to economics)

BP has two phases:

Forward pass phase: computes ‘functional signal’, feed forward propagation of input pattern signals through network

Backward pass phase: computes ‘error signal’, *propagates* the error *backwards* through network starting at output units (where the error is the difference between actual and desired output values)

Conceptually: Forward Activity - Backward Error



Forward Propagation of Activity

- Step 1: Initialise weights at random, choose a learning rate η
- Until network is trained:
- For each training example i.e. input pattern and target output(s):
- Step 2: Do forward pass through net (with fixed weights) to produce output(s)
  - i.e., in Forward Direction, layer by layer:
    - Inputs applied
    - Multiplied by weights
    - Summed
    - ‘Squashed’ by sigmoid activation function
    - Output passed to each neuron in next layer
  - Repeat above until network output(s) produced

Step 3. Back-propagation of error

• Compute error (delta or local gradient) for each output unit δk

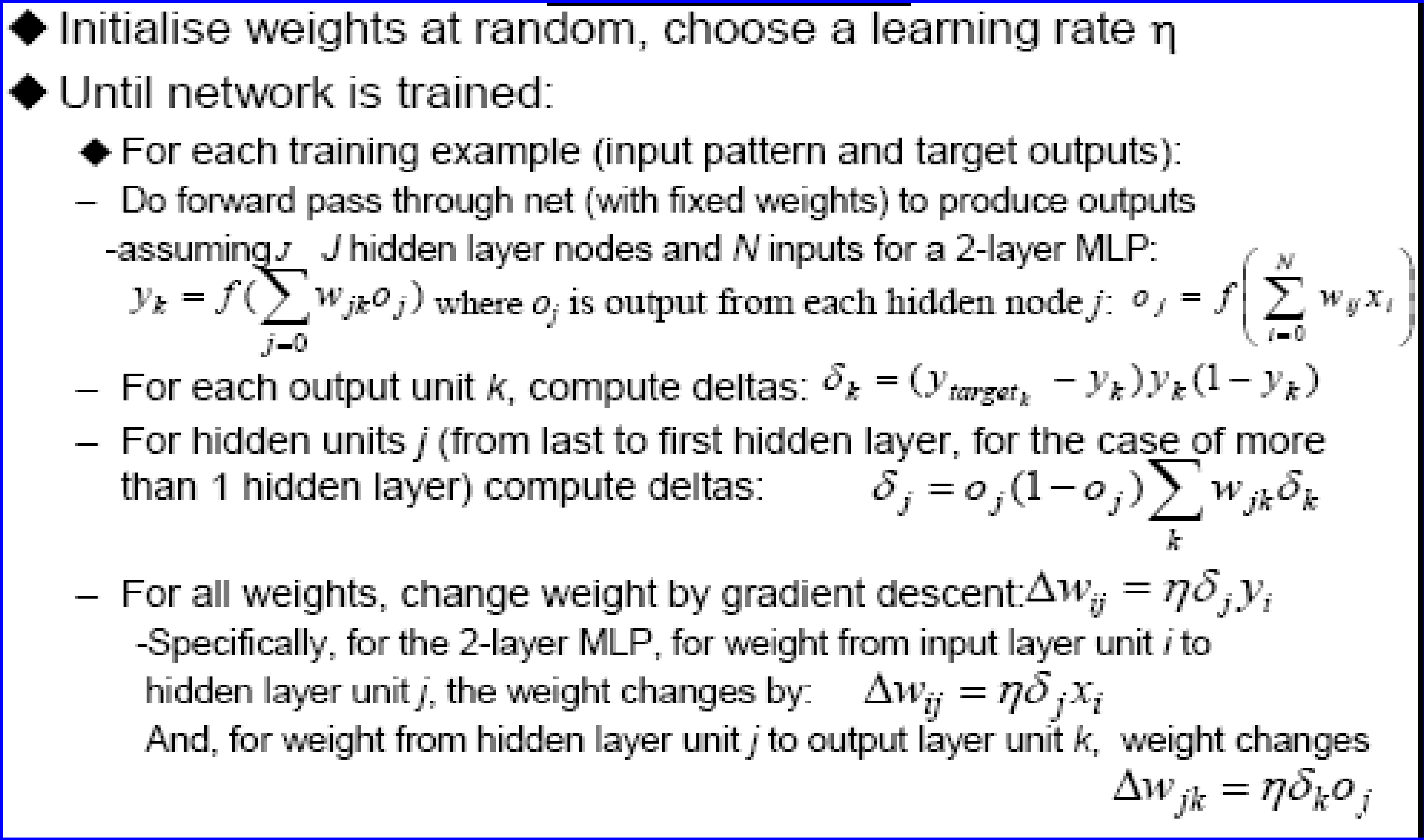
• Layer-by-layer, compute error (delta or local gradient) for each hidden unit δj by backpropagating errors (as shown previously)

Step 4: Next, update all the weights Δwij
By gradient descent, and go back to Step 2

- The overall MLP learning algorithm, involving forward pass and backpropagation of error (until the network training completion), is known as the Generalised Delta Rule (GDR), or more commonly, the Back Propagation (BP) algorithm

‘Back-prop’ algorithm summary

(with Maths!) (Not Examinable)



- Initialise weights at random, choose a learning rate η
- Until network is trained:
  - For each training example (input pattern and target outputs):
  - – Do forward pass through net (with fixed weights) to produce outputs

    -assuming J J hidden layer nodes and *N* inputs for a 2-layer MLP:

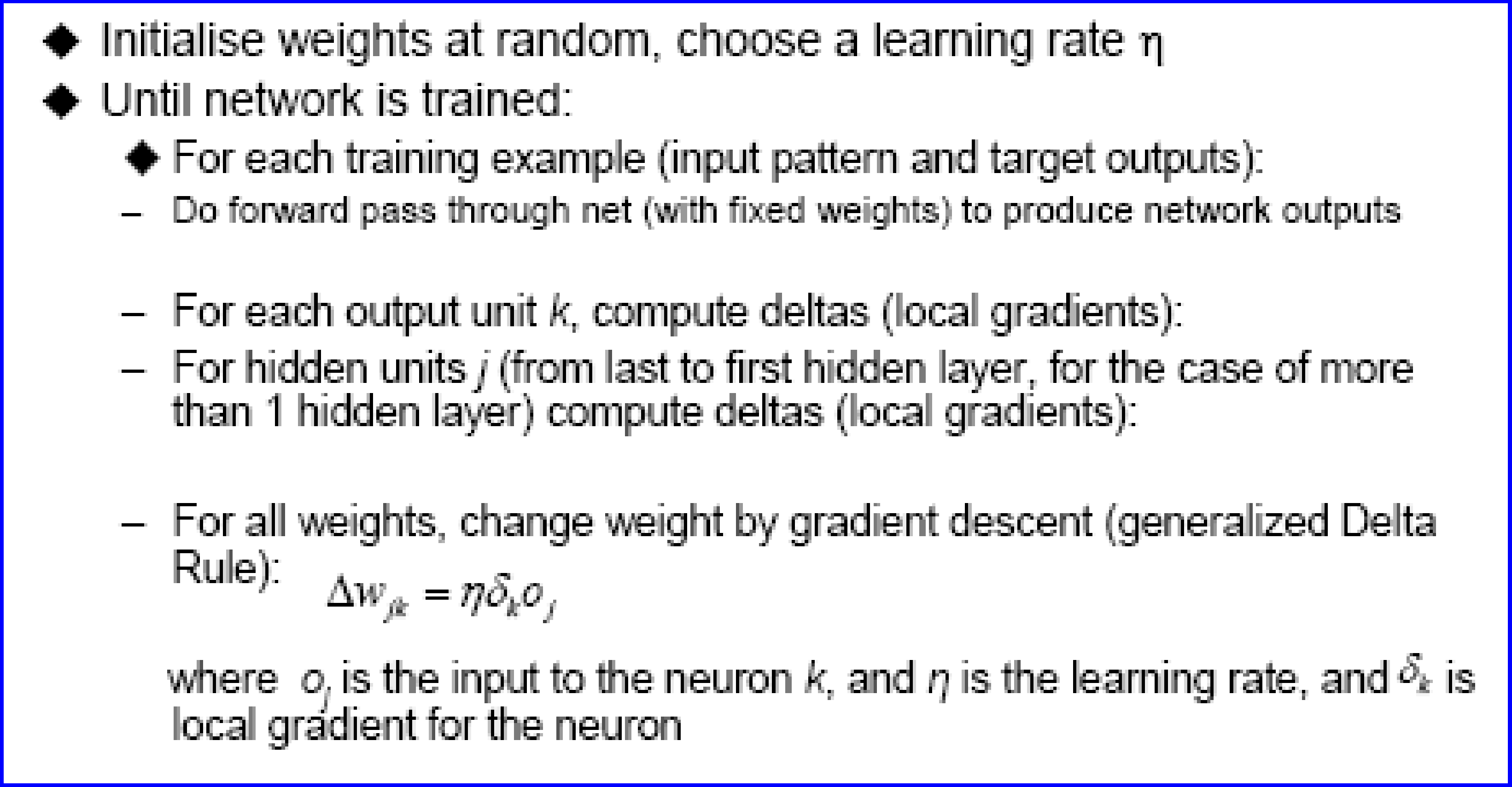
    $y_k = f(\sum_{j=0} w_{jk} o_j)$ where o_j is output from each hidden node j: $o_j = f\left(\sum_{i=0}^{N} w_{ij} x_i\right)$
  - – For each output unit *k*, compute deltas: $\delta_k = (y_{target_k} - y_k) y_k (1 - y_k)$
  - – For hidden units *j* (from last to first hidden layer, for the case of more than 1 hidden layer) compute deltas: $\delta_j = o_j (1 - o_j) \sum_k w_{jk} \delta_k$
  - – For all weights, change weight by gradient descent: $\Delta w_{ij} = \eta \delta_j y_i$

    -Specifically, for the 2-layer MLP, for weight from input layer unit *i* to hidden layer unit *j*, the weight changes by: $\Delta w_{ij} = \eta \delta_j x_i$

    And, for weight from hidden layer unit *j* to output layer unit *k*, weight changes $\Delta w_{jk} = \eta \delta_k o_j$

‘Back-prop’ algorithm summary

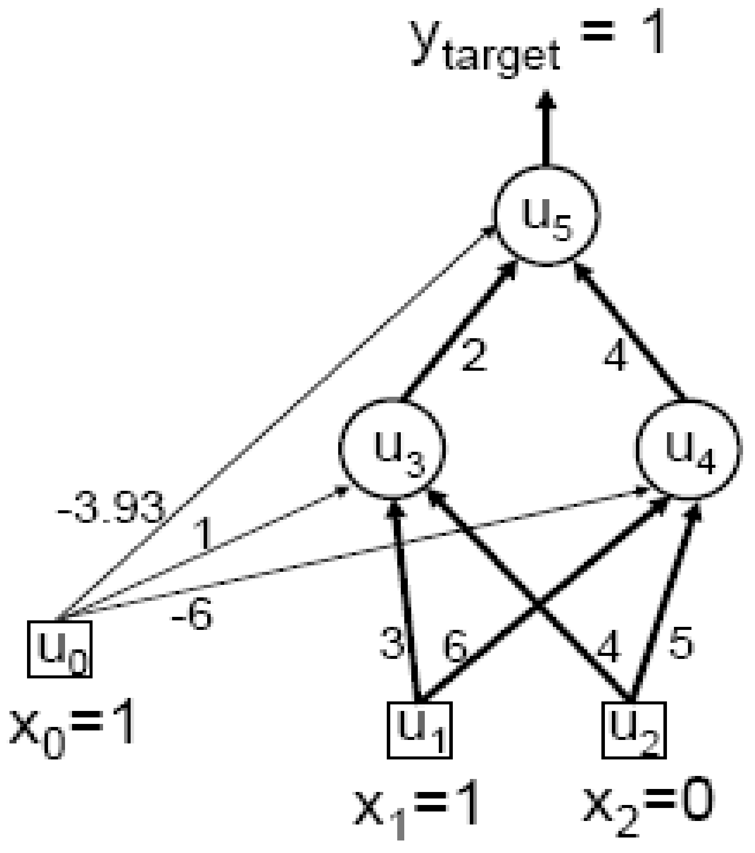
(with NO Maths!)



- Initialise weights at random, choose a learning rate η
- Until network is trained:
  - For each training example (input pattern and target outputs):
  - Do forward pass through net (with fixed weights) to produce network outputs
  - For each output unit *k*, compute deltas (local gradients):
  - For hidden units *j* (from last to first hidden layer, for the case of more than 1 hidden layer) compute deltas (local gradients):
  - For all weights, change weight by gradient descent (generalized Delta Rule): $\Delta w_{jk} = \eta \delta_k o_j$

where o_j is the input to the neuron k, and η is the learning rate, and δ_k is local gradient for the neuron

MLP/BP: A worked example



Current state:

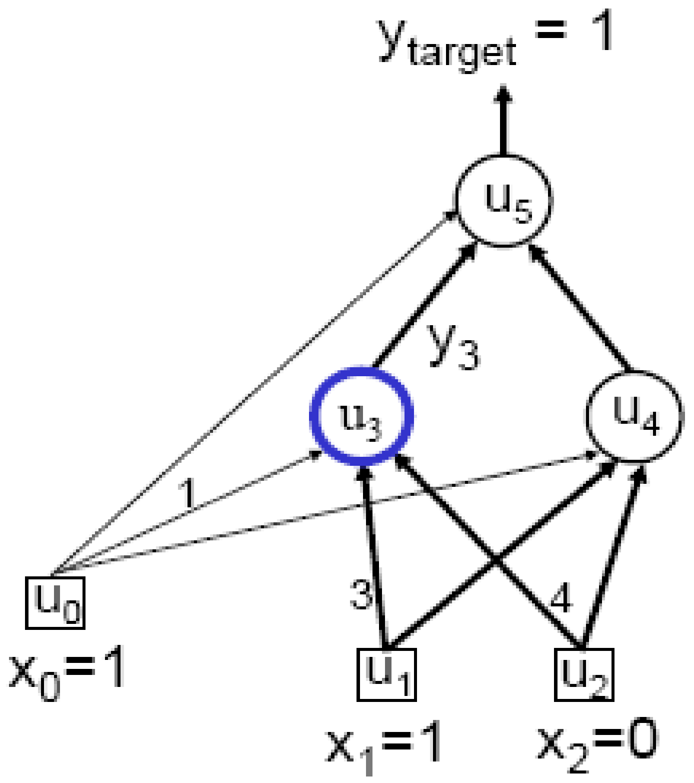
- Weights on arrows e.g. $w_{13} = 3$, $w_{35} = 2$, $w_{24} = 5$
- Bias weights, e.g.

bias for unit 4 (u_4) is $w_{04}= -6$

Training example (e.g. for logical OR problem):

- Input pattern is $x_1=1$, $x_2=0$
- Target output is $y_{target}=1$

Worked example: Forward Pass



Output for any neuron/unit j can be calculated from:

$$a_j = \sum_i w_{ij} x_i$$

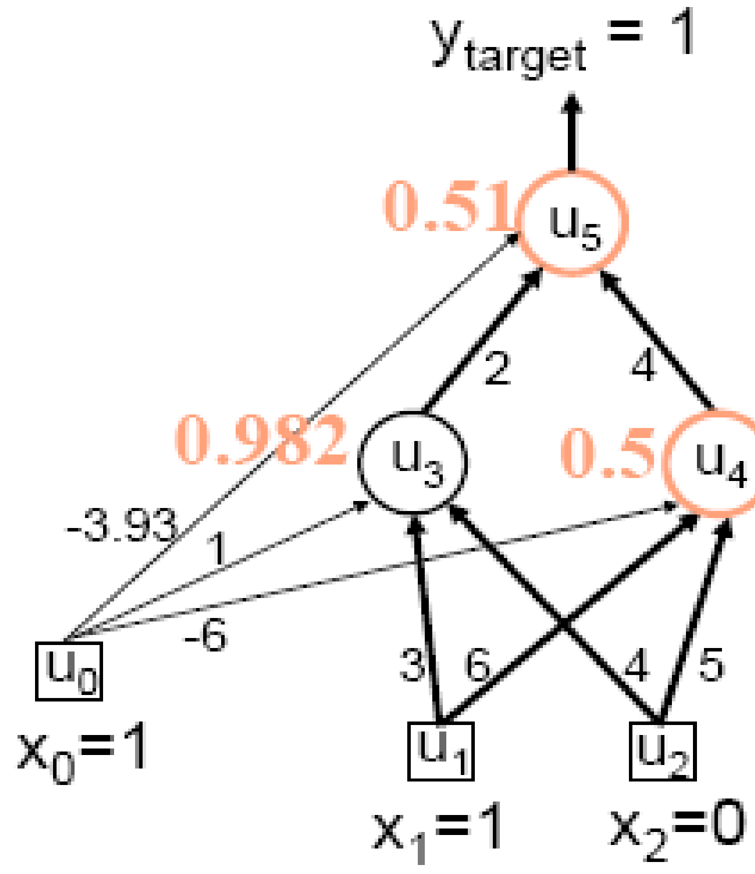
$$y_j = f(a_j) = \frac{1}{1+e^{-a_j}}$$

e.g Calculating output for Neuron/unit 3 in hidden layer:

$$a_3 = 1*1 + 3*1 + 4*0 = 4$$

$$y_3 = f(4) = \frac{1}{1+e^{-4}} = 0.982$$

Worked example: Forward Pass

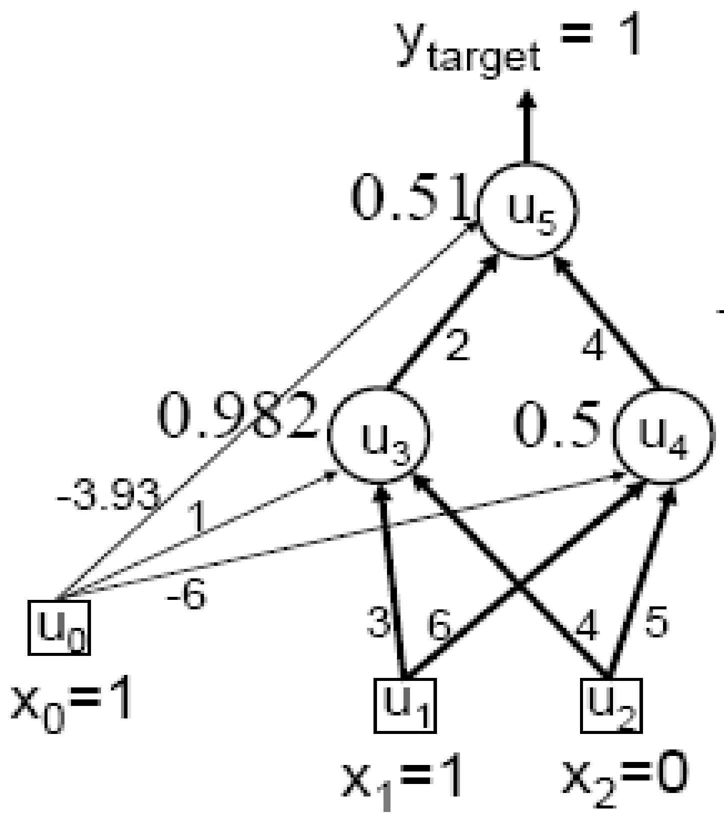


Unit	activation a_j	output y_j
u_3	4.00	0.982
u_4	0.00	0.500
u_5	0.04	**0.510**

(network output)

So the error for this training example is: $(y_{target} - y_5) = (1 - 0.510) = 0.490$

Worked example: Backward Pass



Now compute delta values starting at the output:

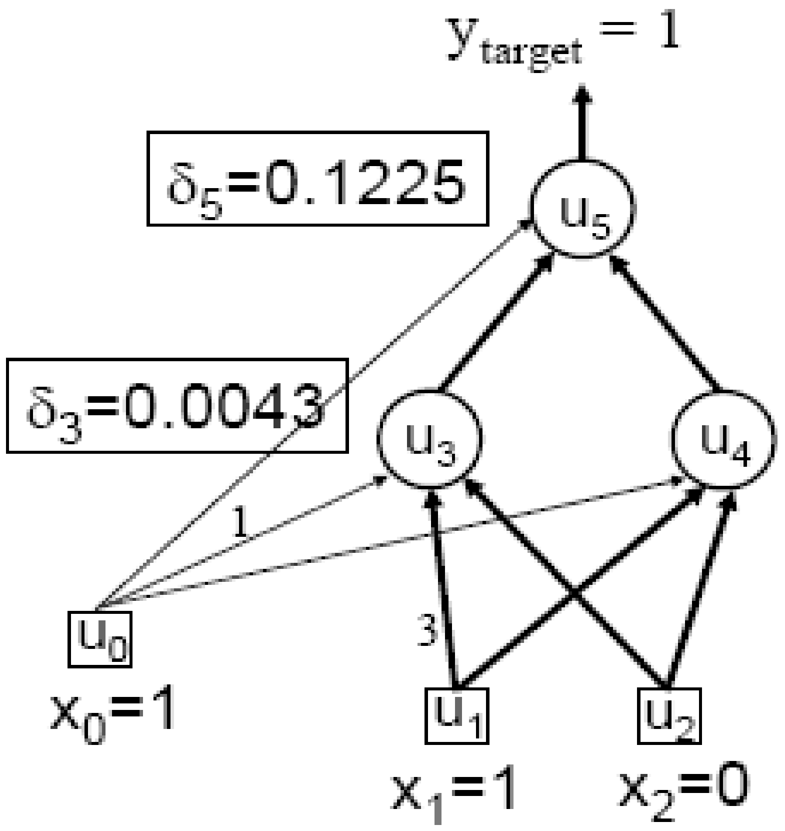
$\delta_5 = y_5(1 - y_5)\,(y_{target} - y_5)$
$= 0.51(1 - 0.51) \times 0.49$
$= \mathbf{0.1225}$

Then for hidden units:

$\delta_4 = y_4(1 - y_4)\, w_{45}\, \delta_5$
$= 0.5(1 - 0.5) \times 4 \times 0.1225$
$= \mathbf{0.1225}$

$\delta_3 = y_3(1 - y_3)\, w_{35}\, \delta_5$
$= 0.982(1-0.982) \times 2 \times 0.1225$
$= \mathbf{0.0043}$

Worked example: Update Weights Using Generalized Delta Rule (BP)



- Set learning rate $\eta = 0.1$

  Change weights by:

  $$\Delta w_{ij} = \eta \delta_j y_i$$

- e.g.bias weight on u_3:

Δw_{03} = $\eta \delta_3 x_0$

=0.1*0.0043*1

= 0.0004

So, new w_{03} ⌫

w_{03}(old)+Δw_{03}

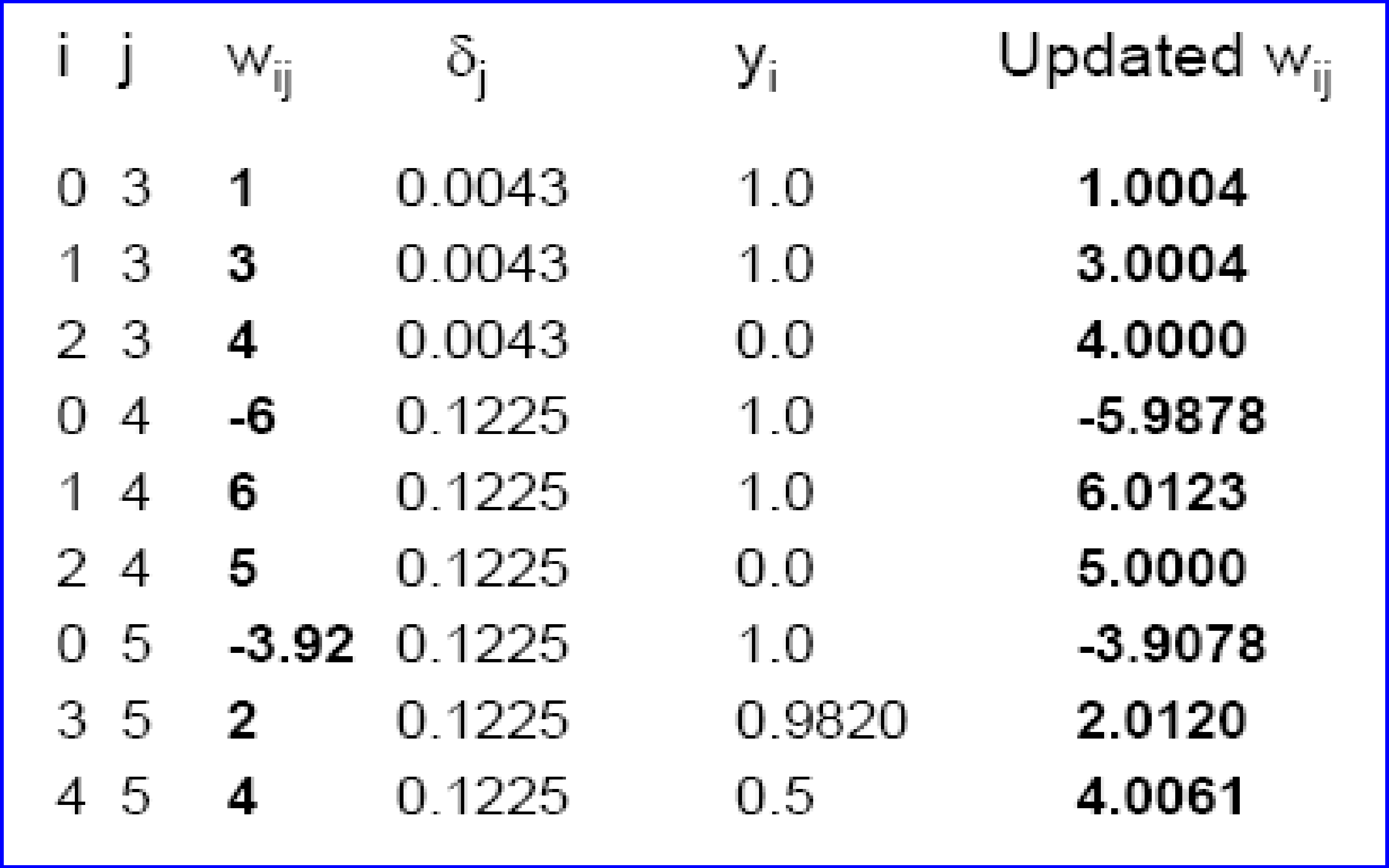
=1+0.0004=1.0004

- and likewise:

  w_{13} ⌫ 3 + 0.0004

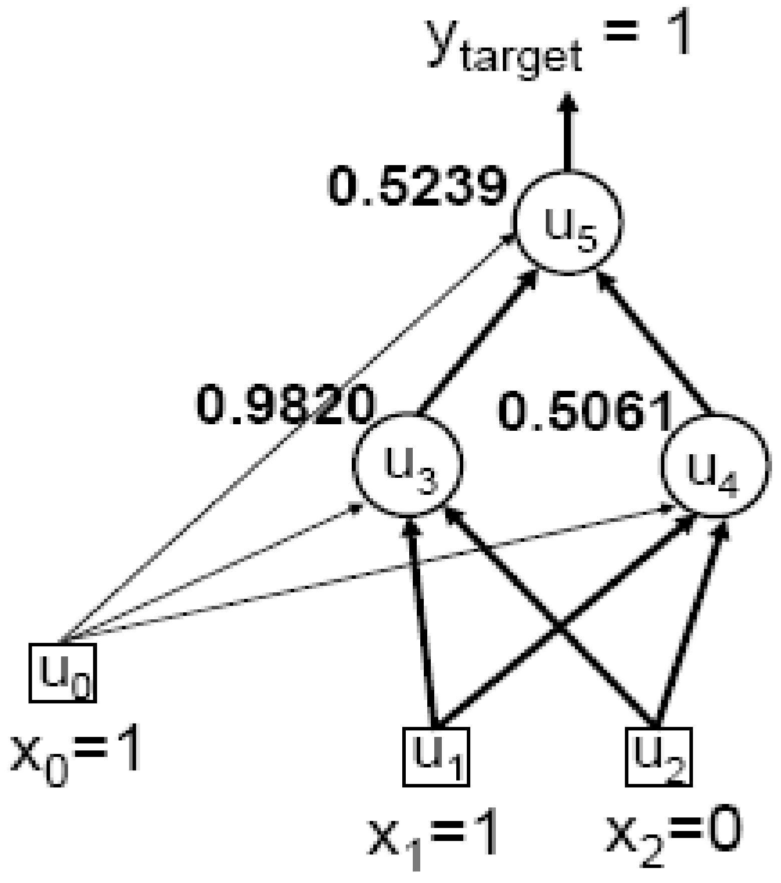
  = 3.0004

Similarly for the all weights wij:



i	j	w_{ij}	δ_j	y_i	Updated w_{ij}
0	3	**1**	0.0043	1.0	**1.0004**
1	3	**3**	0.0043	1.0	**3.0004**
2	3	**4**	0.0043	0.0	**4.0000**
0	4	**-6**	0.1225	1.0	**-5.9878**
1	4	**6**	0.1225	1.0	**6.0123**
2	4	**5**	0.1225	0.0	**5.0000**
0	5	**-3.92**	0.1225	1.0	**-3.9078**
3	5	**2**	0.1225	0.9820	**2.0120**
4	5	**4**	0.1225	0.5	**4.0061**

Verification that it works



On <u>next forward pass</u>:

The new activations are:

$y_3 = f\ (4.0008) = 0.9820$

$y_4 = f\ (0.0245) = 0.5061$

$y_5 = f\ (0.0955) = \mathbf{0.5239}$

Thus the <u>new error</u>

$(y_{target} - y_5) = (1 - 0.5239) = 0.476$

<u>has been reduced</u> by **0.014**

(from **0.490** to **0.476**)

Ref: "Neural Network Learning & Expert Systems" by Stephen Gallant

Training

- This was a single iteration of back-prop
- Training requires many iterations with many training examples or *epochs* (one epoch is entire presentation of complete training set)
- It can be slow !
- Note that computation in MLP is local (with respect to each neuron)
- Parallel computation implementation is also possible

Training and testing data

- How many examples ?
  - The more the merrier !
- Disjoint training and testing data sets
  - learn from training data but evaluate performance (generalization ability) on unseen test data
- **Aim**: minimize error on *test* data