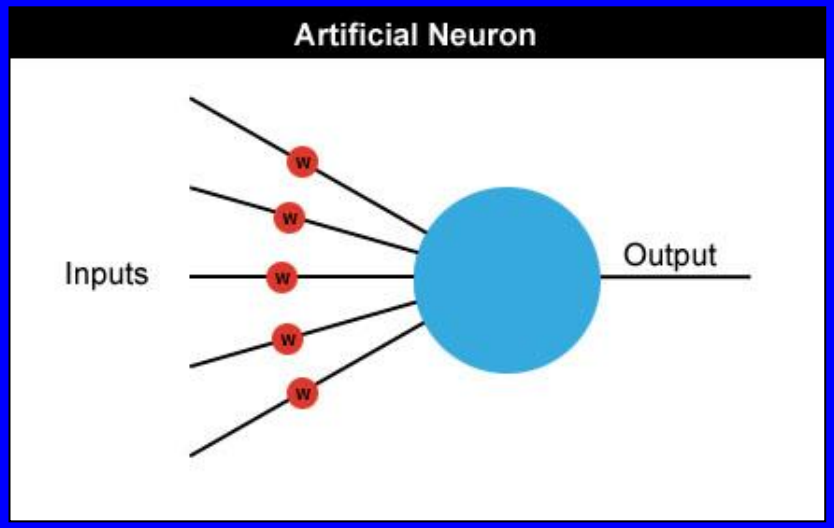
Neural networks

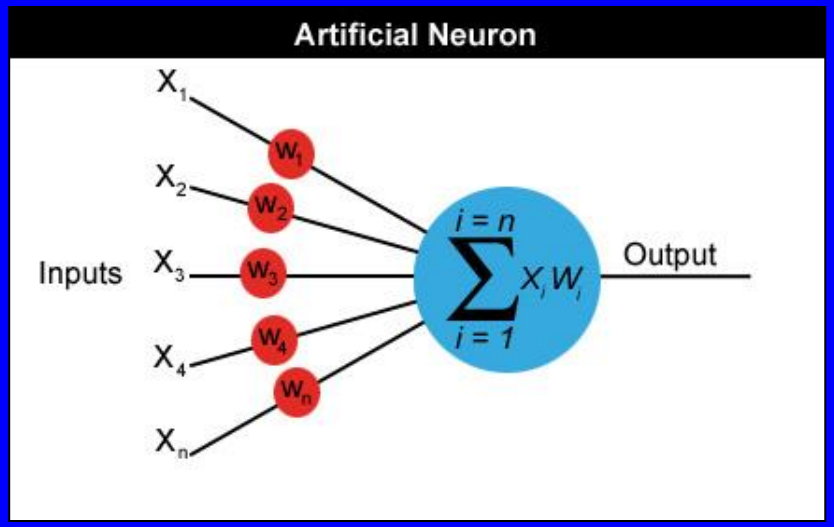
Neural networks

- 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.



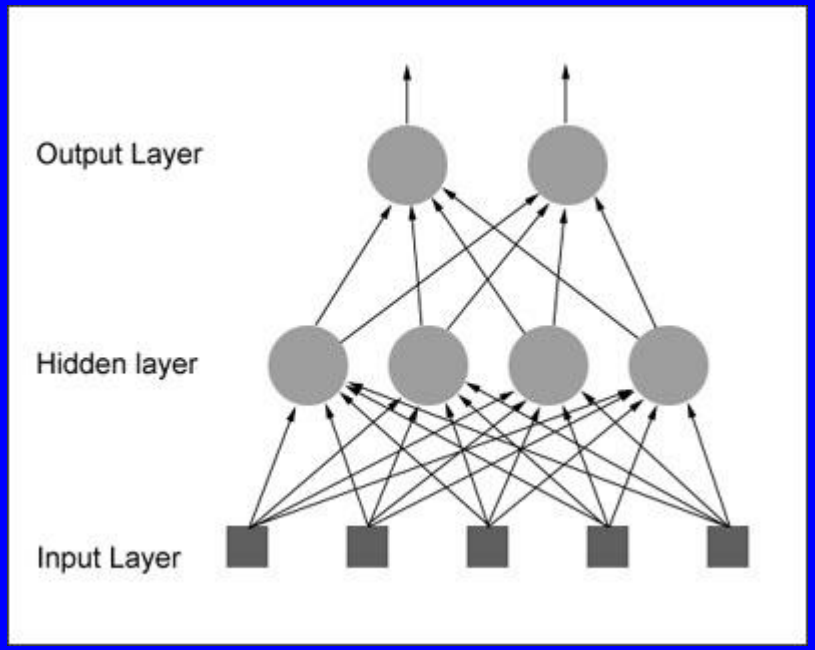
Neural networks

- 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 $a = x_1w_1+x_2w_2+x_3w_3... +x_nw_n$



How do we actually *use* an artificial neuron?

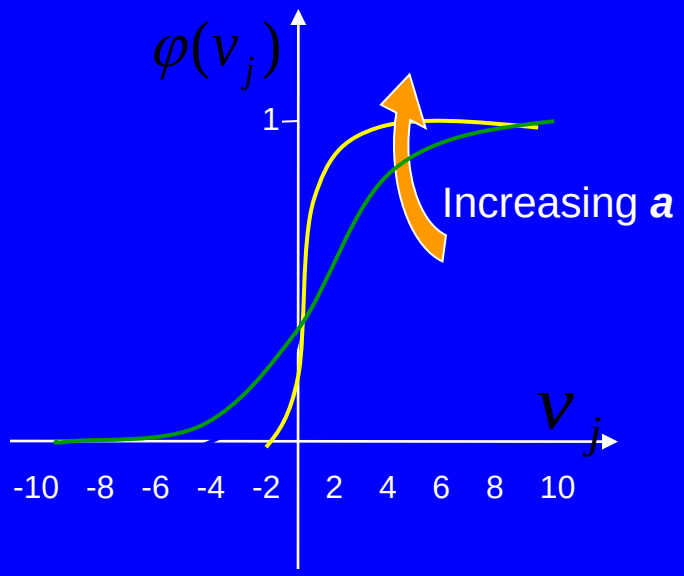
- feedforward network: The neurons in each layer feed their output forward to the next layer until we get the final output from the neural network.
- There can be any number of hidden layers within a feedforward network.
- The number of neurons can be completely arbitrary.



Neural Networks by an Example

- 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 adjust the weights accordingly.

NEURON MODEL

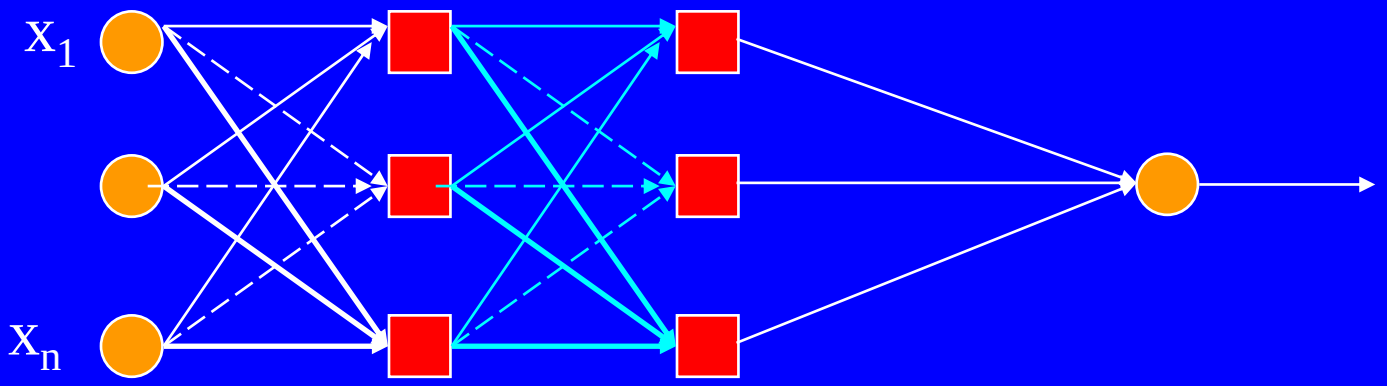


$$\varphi(\mathrm{v}_\mathrm{j}) = \frac{1}{1+e^{-av_j}}$$

$$\mathrm{v}_\mathrm{j} = \sum_{i=0,\ldots,m} w_{\mathrm{ji}} y_\mathrm{i}$$

- v_j induced field of neuron j
- Most common form of activation function
- a $\rightarrow\infty \Rightarrow \varphi \rightarrow$ threshold function

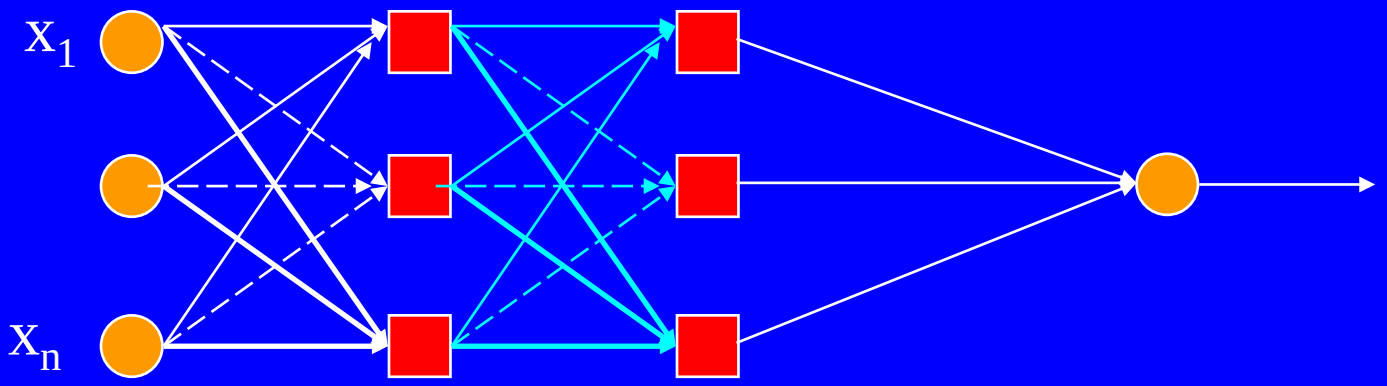
Multi-Layer Perceptron (MLP)



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



NB 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:

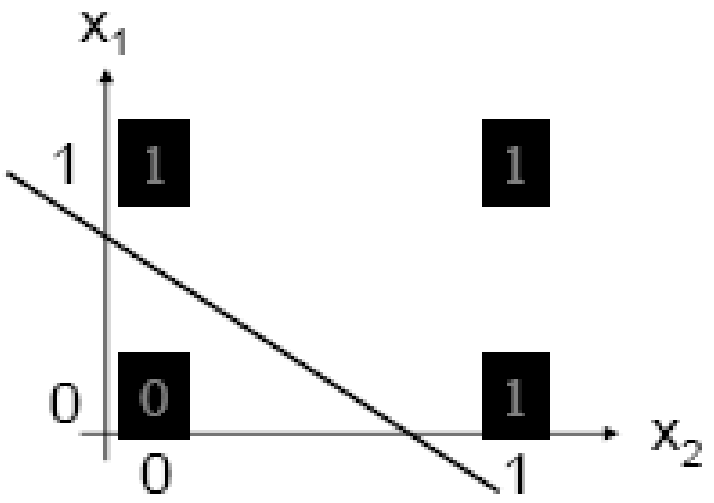
what do the extra layers gain you? Start with looking at what a single layer can't do

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

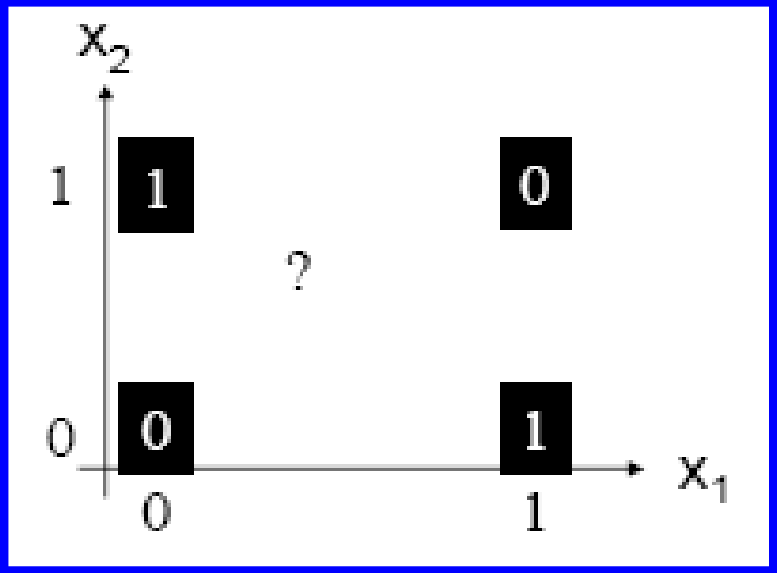


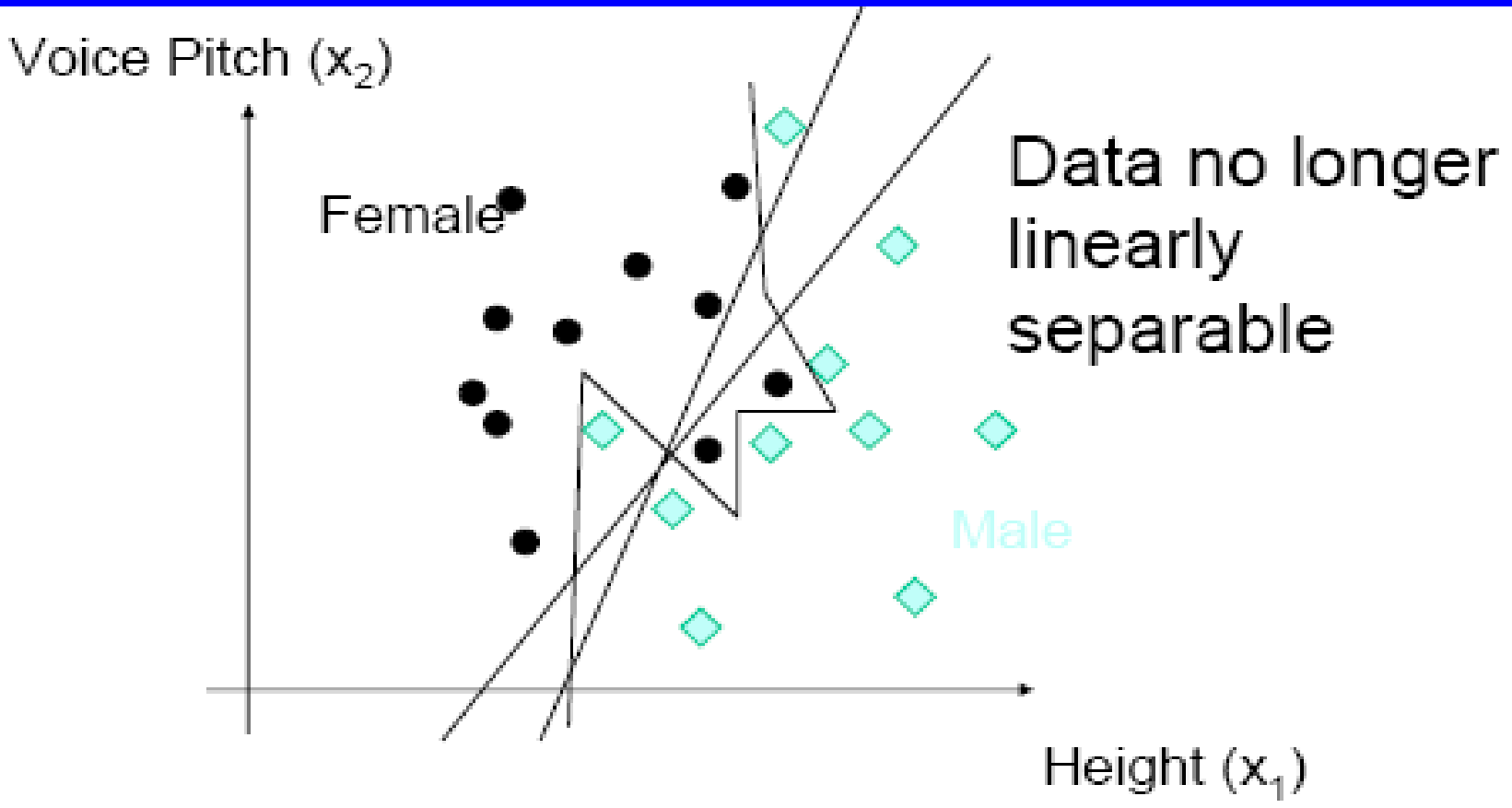
The Exclusive OR problem

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

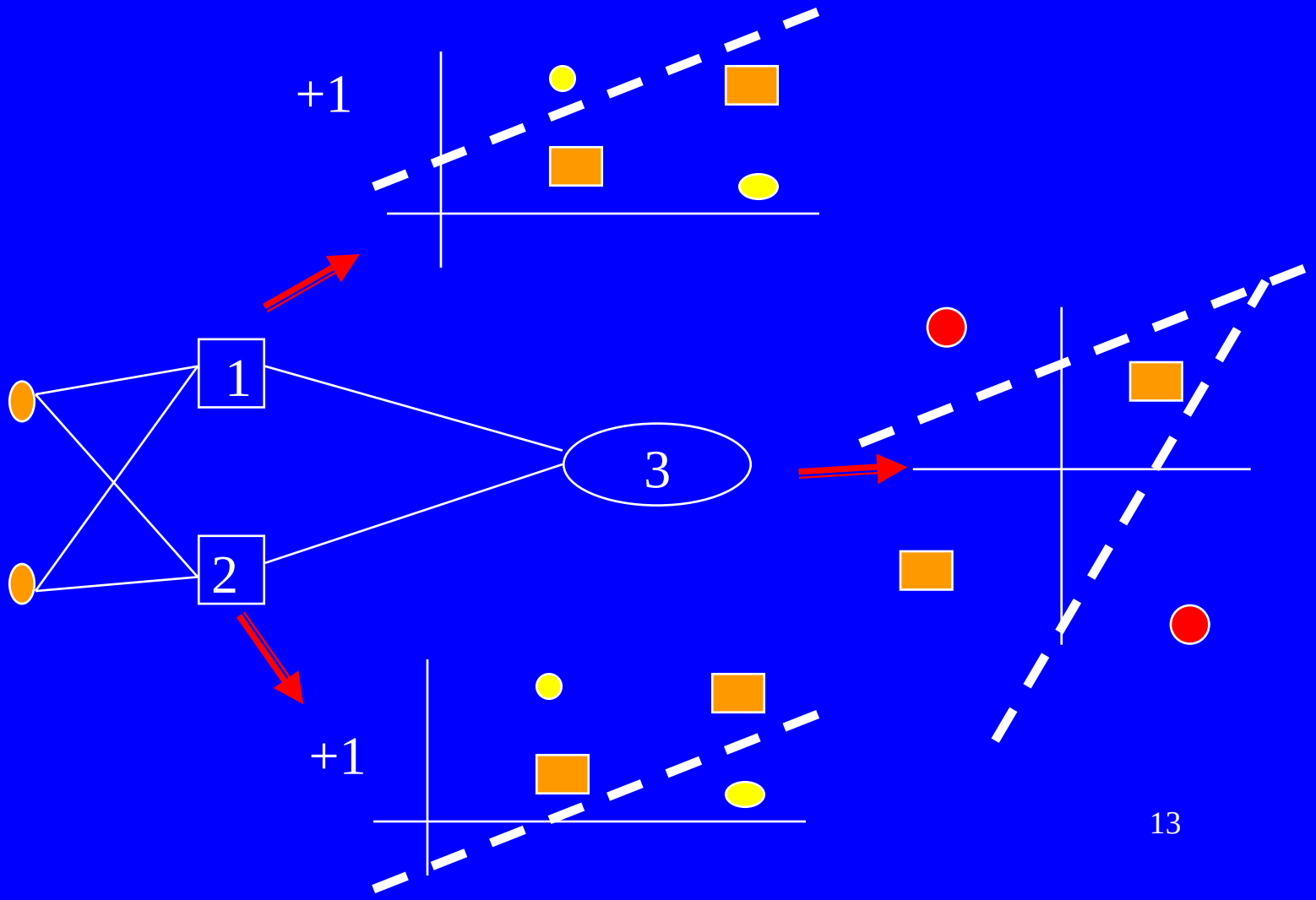




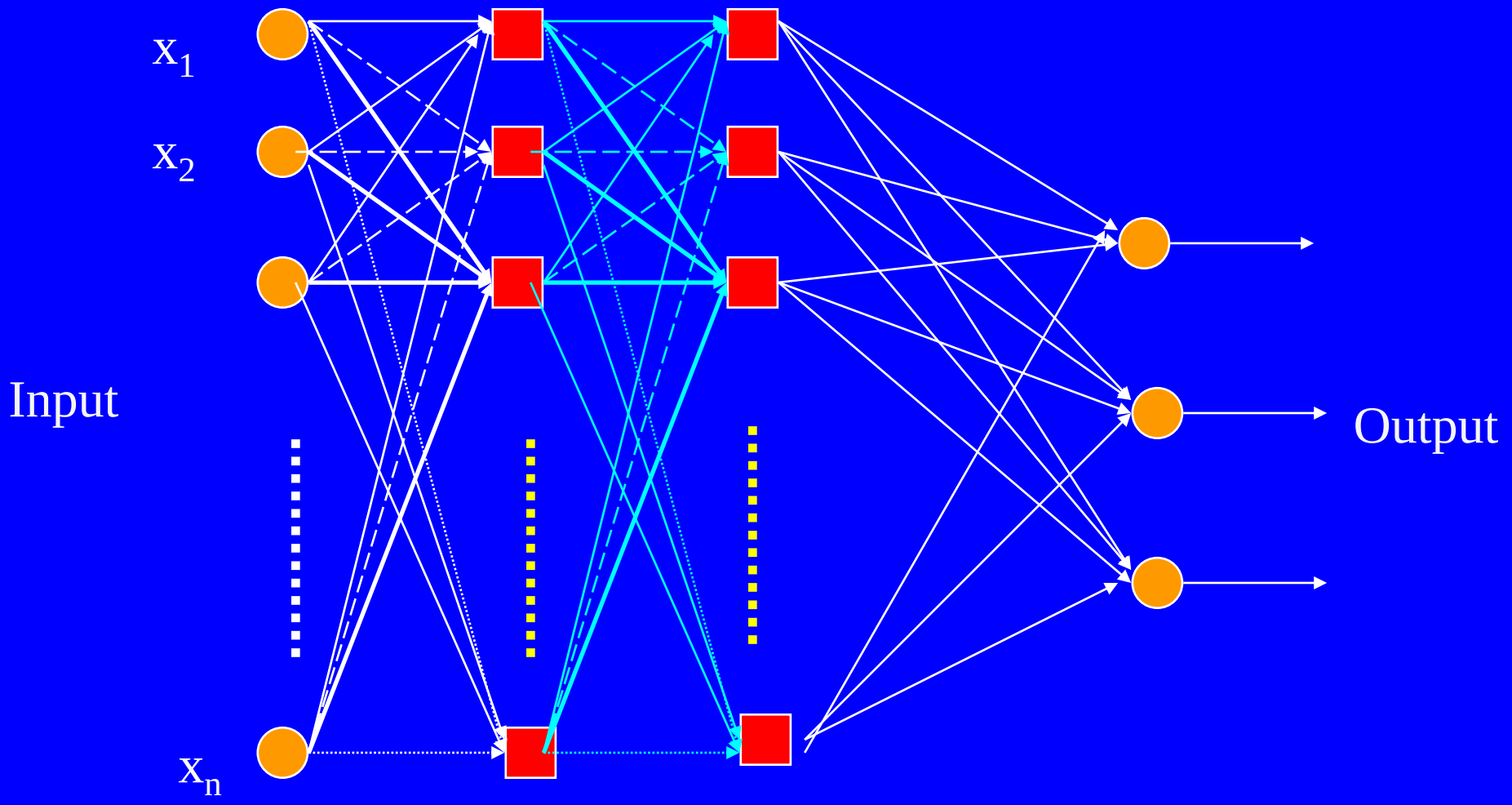
Data no longer linearly separable

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

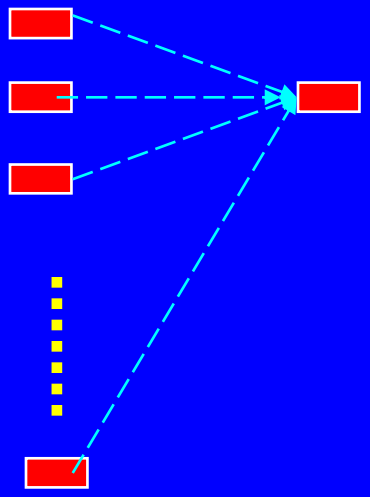


Three-layer networks



Properties of architecture

- No connections within a layer

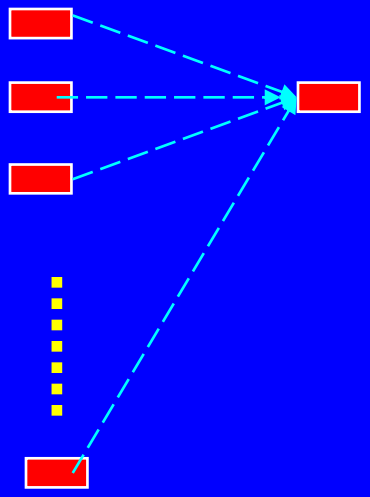


Each unit is a perceptron

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

Properties of architecture

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

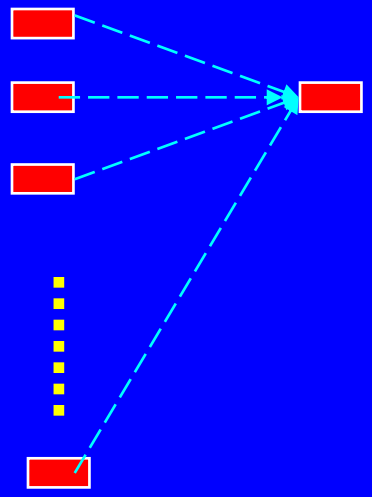


Each unit is a perceptron

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

Properties of architecture

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

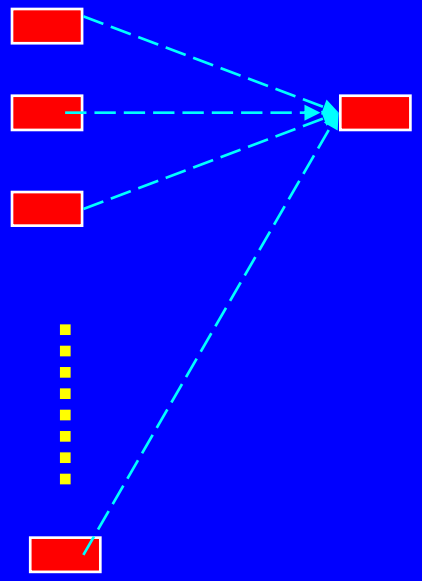


Each unit is a perceptron

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

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

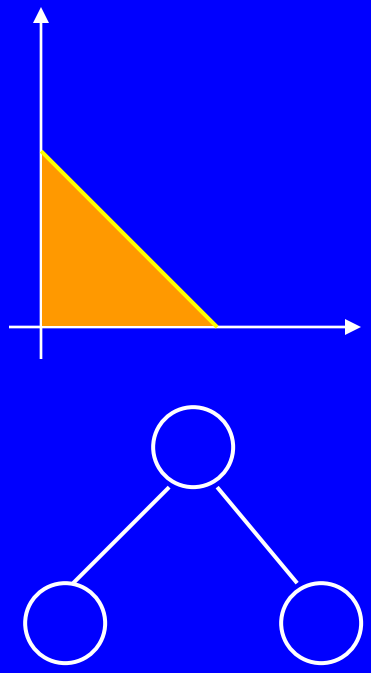


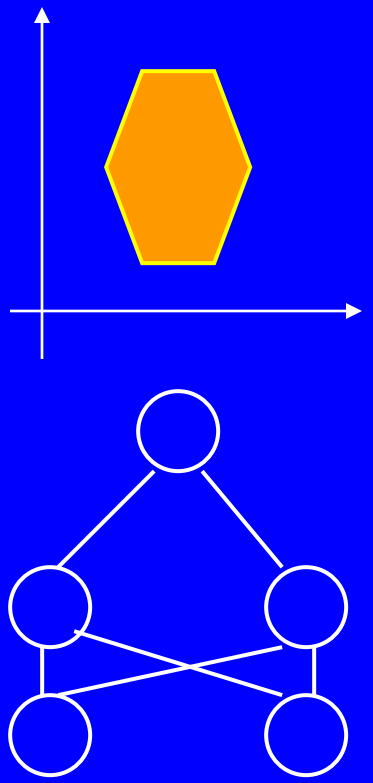
Each unit is a perceptron

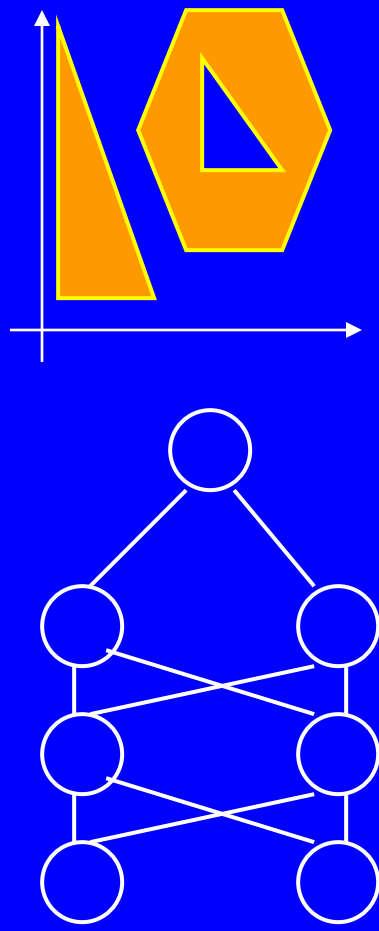
$$y_i = f\left(\sum_{j=1}^{m} w_{ij} x_j + b_i\right)$$

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'

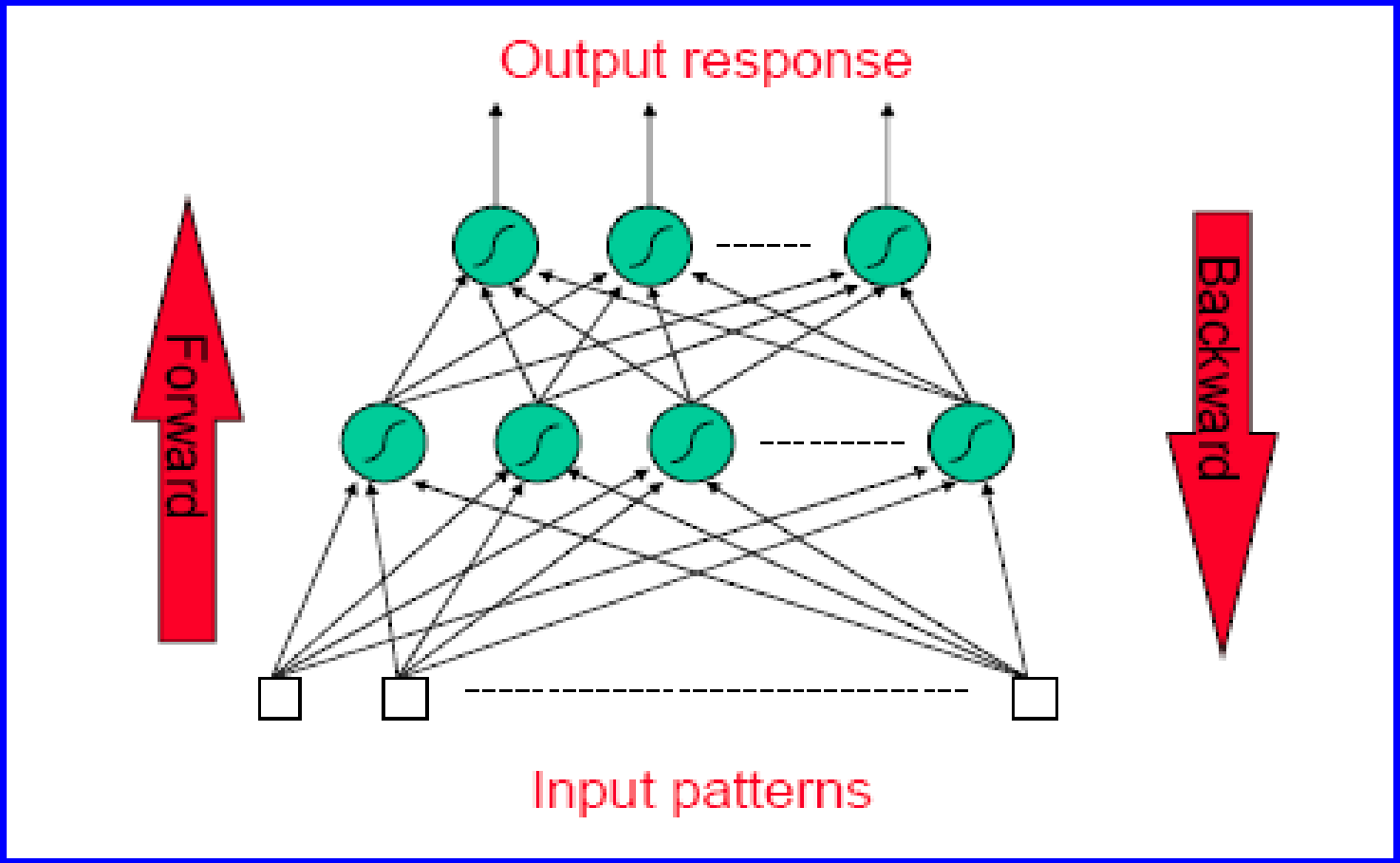
Solution to credit assignment problem in MLP. *Rumelhart, Hinton and Williams (1986) (*though actually invented earlier in a PhD thesis relating to economics)

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 δ *k*

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

Step 4: Next, update all the weights Δ*wij*

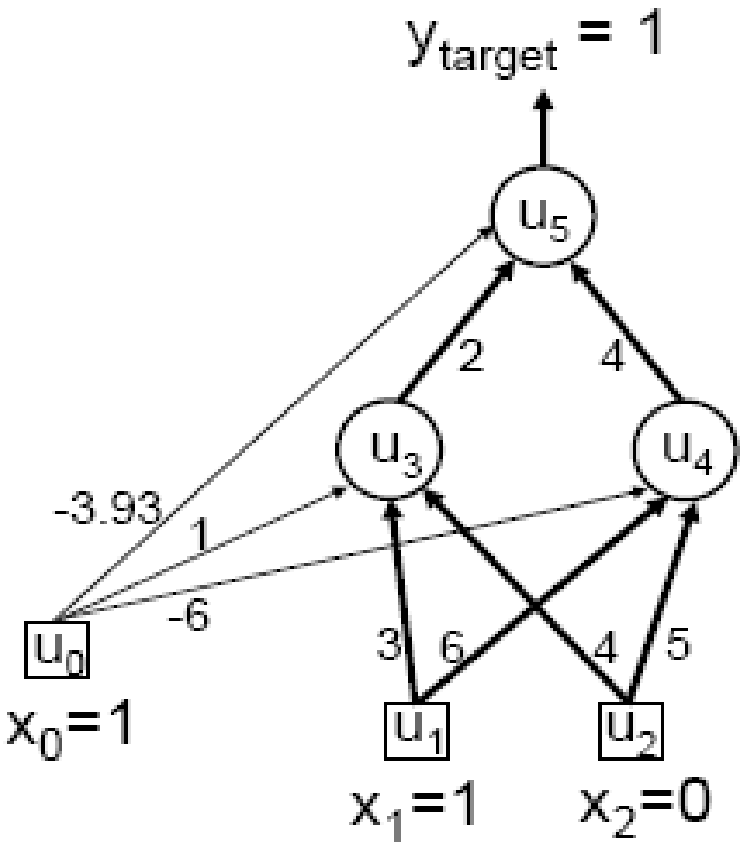
By gradient descent, and go back to Step 2

– 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

- 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 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 $\Delta w_{jk} = \eta \delta_k o_j$

MLP/BP: A worked example



Current state:

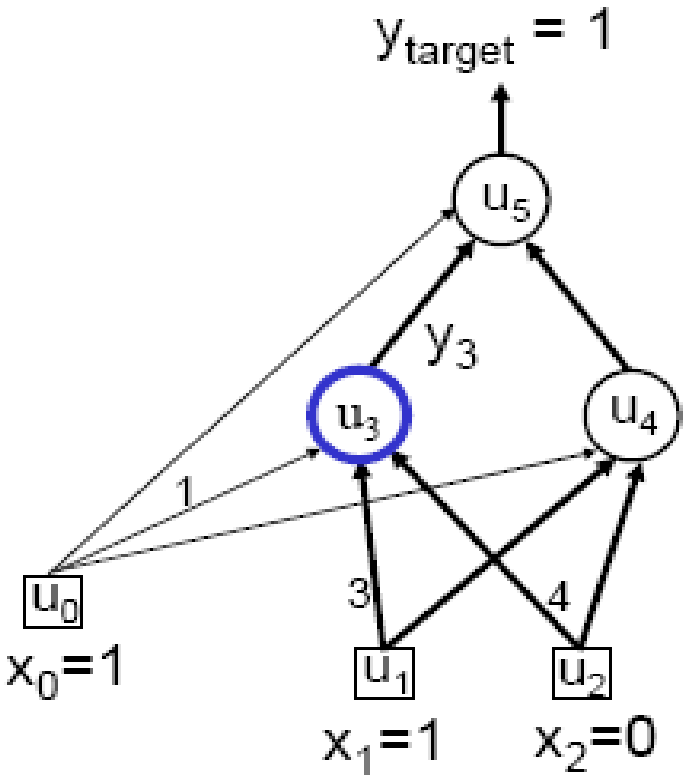
- Weights on arrows e.g. $w_{13} = 3$, $w_{35} = 2$, $w_{24} = 5$
- Bias weights, e.g.

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:

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

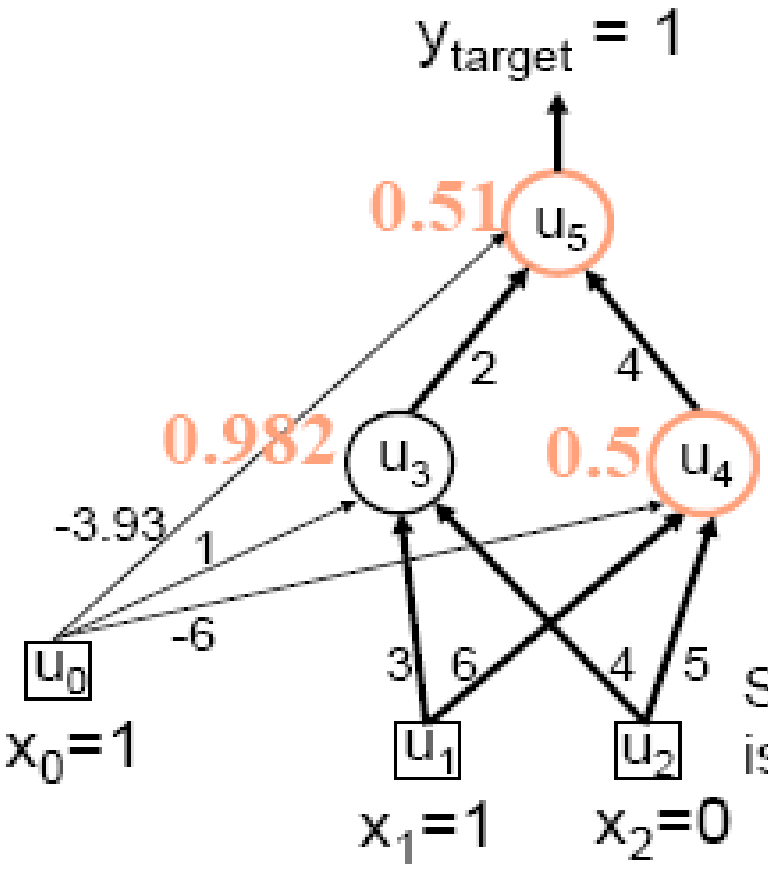
$$y_j = f(a_j) = \frac{1}{1+e^{-a_j}}$$

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

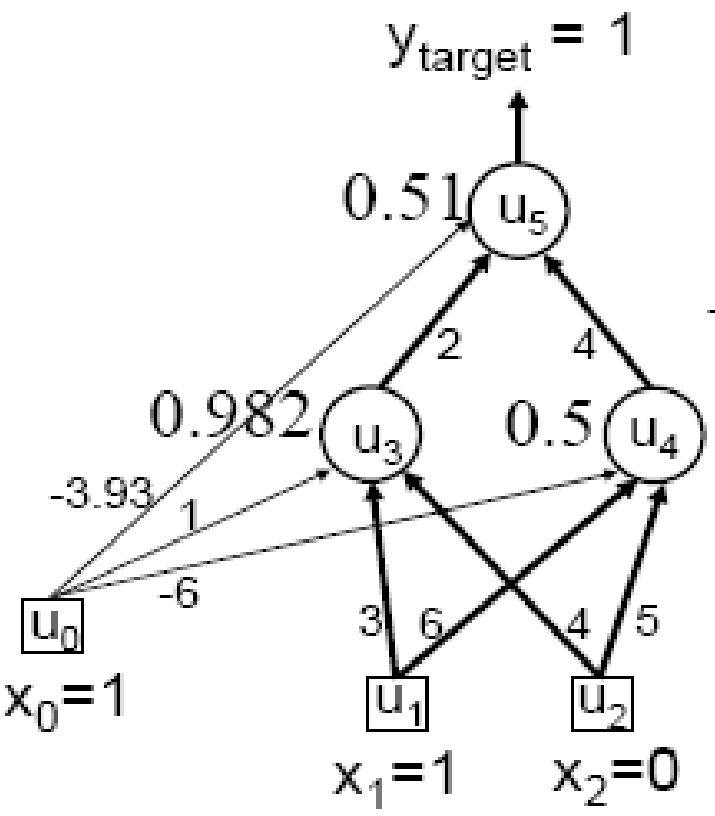


Unit	activation a_j	output y_j
u_3	4.00	0.982
u_4	0.00	0.500
u_5	0.04	**0.510**

(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:

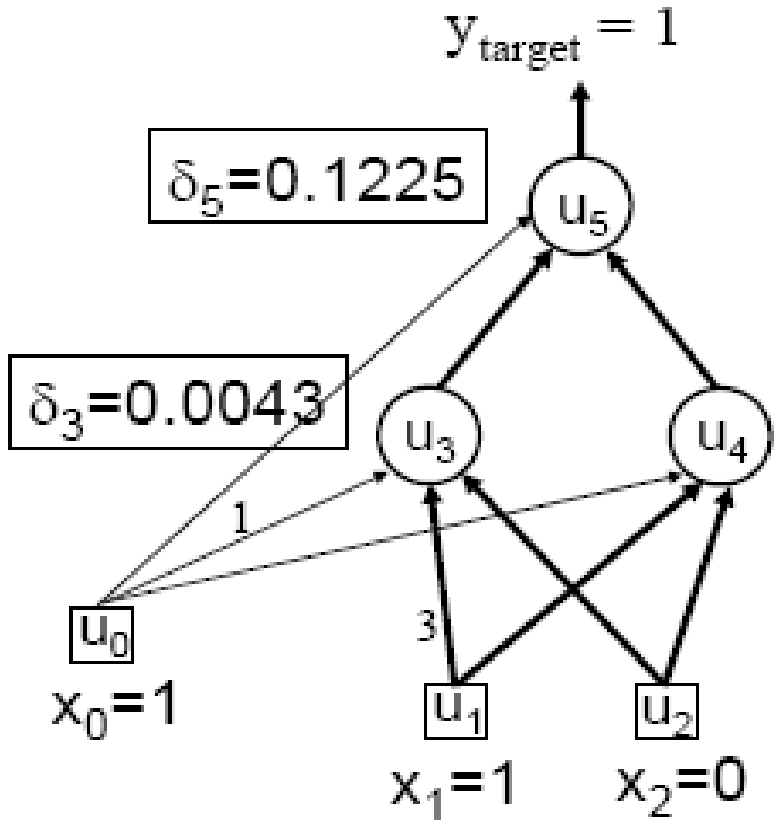
δ_5 $= y_5(1 - y_5)\,(y_{target} - y_5)$
$= 0.51(1 - 0.51) \times 0.49$
$=$ **0.1225**

Then for hidden units:

δ_4 $= y_4(1 - y_4)\, w_{45}\, \delta_5$
$= 0.5(1 - 0.5) \times 4 \times 0.1225$
$=$ **0.1225**

δ_3 $= y_3(1 - y_3)\, w_{35}\, \delta_5$
$= 0.982(1-0.982) \times 2 \times 0.1225$
$=$ **0.0043**

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



- Set learning rate η = 0.1

  Change weights by:

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

- e.g.bias weight on u_3:

$\Delta w_{03} = \eta \delta_3 x_0$

$=0.1*0.0043*1$

$= 0.0004$

So, new w_{03} ⌧

$w_{03}(old)+\Delta w_{03}$

$=1+0.0004=1.0004$

- and likewise:

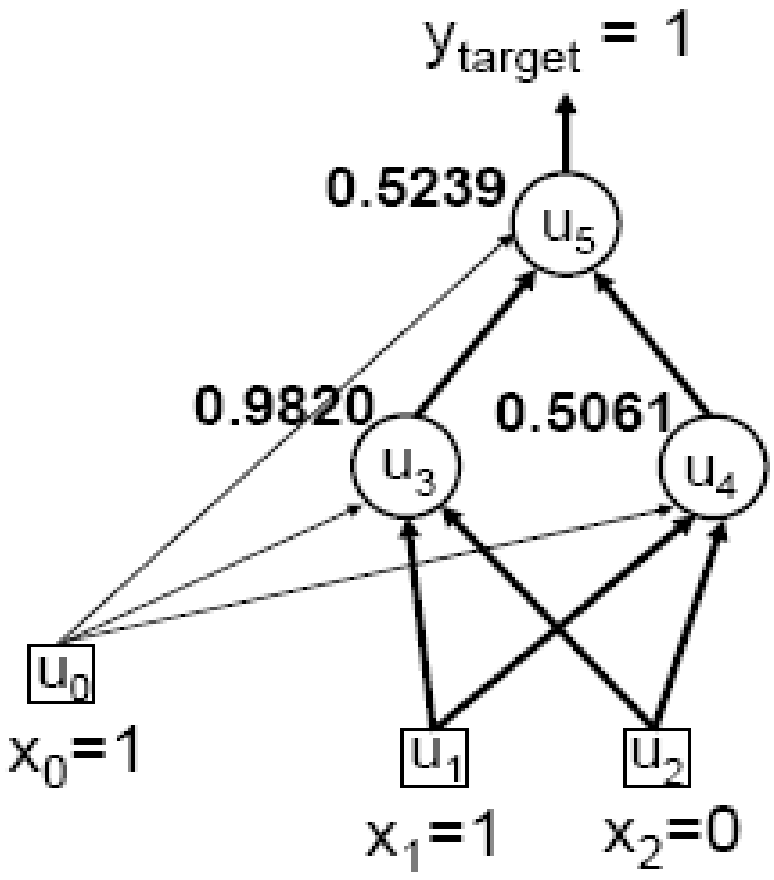
w_{13} ⌧ 3 + 0.0004

= 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 next forward pass:

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 new error

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

has been reduced by **0.014**

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

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

Momentum

Method of reducing problems of instability while increasing the rate of convergence

Adding term to weight update equation term effectively exponentially holds weight history of previous weights changed

Modified weight update equation is

$$w_{ij}(n+1) - w_{ij}(n) = \eta \delta_j(n) y_i(n) + \alpha[w_{ij}(n) - w_{ij}(n-1)]$$

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

α is momentum constant and controls how much notice is taken of recent history

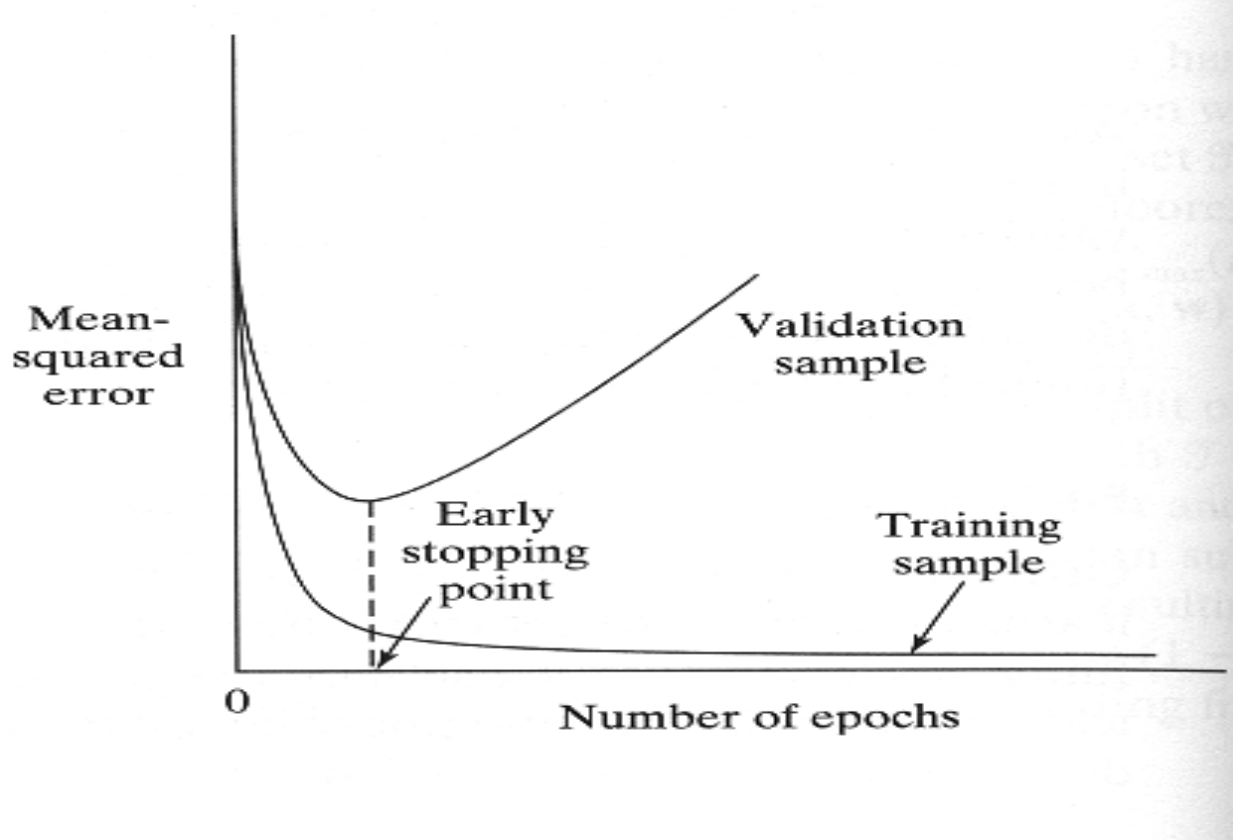
Effect of momentum term

- If weight changes tend to have same sign
  momentum terms increases and gradient decrease
  speed up convergence on shallow gradient
- If weight changes tend have opposing signs
  momentum term decreases and gradient descent slows to reduce oscillations (stablizes)
- Can help escape being trapped in local minima

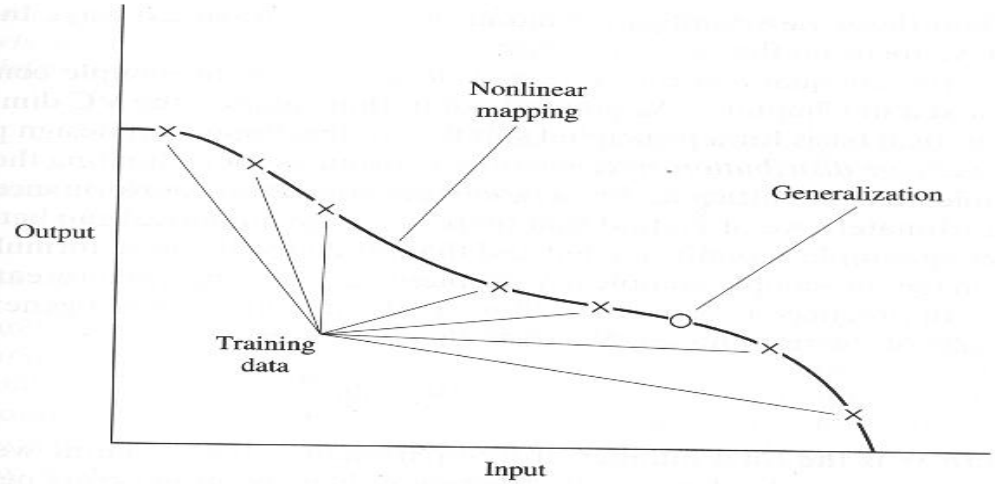
Stopping criteria

- Sensible stopping criteria:
  - Average squared error change: Back-prop is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small (in the range [0.1, 0.01]).
  - Generalization based criterion: After each epoch the NN is tested for generalization. If the generalization performance is adequate then stop.

Early stopping



Generalization



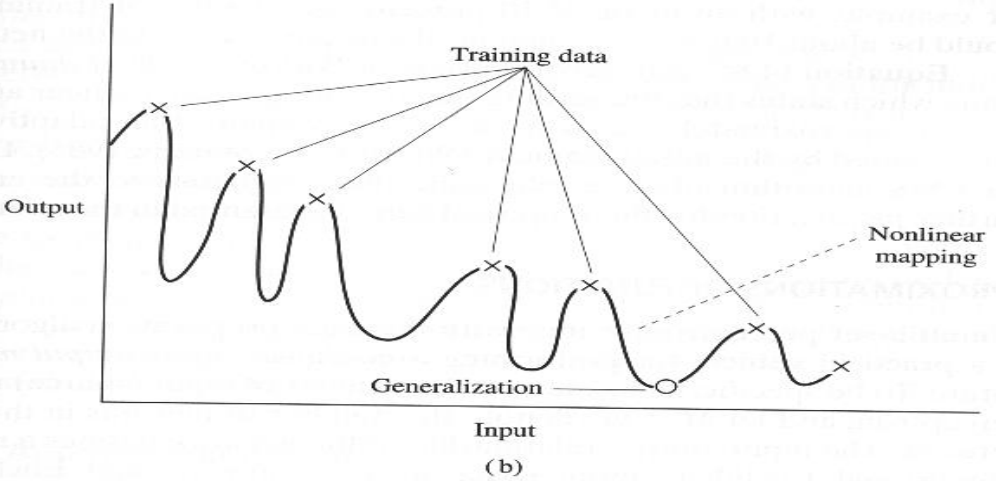


FIGURE 4.19 (a) Properly fitted data (good generalization) (b) Overfitted data (poor generalization).

GMDH

- Introduced by Ivakhnenko in 1966
- Group Method of Data Handling is the realization of inductive approach for mathematical modeling of complex systems.
- A data-driven modeling method that approximates a given variable y (output) as a function of a set of input variables

The General Form of GMDH

$$y = a_0 + \sum_{i=1}^{m} a_i x_i + \sum_{i=1}^{m}\sum_{j=1}^{m} a_{ij} x_i x_j + \sum_{i=1}^{m}\sum_{j=1}^{m}\sum_{k=1}^{m} a_{ijk} x_i x_j x_k \ldots$$

where $X(x_1,x_2,\ldots,x_m)$ - input variables vector; $A(a_1,a_2,\ldots,a_m)$ - vector of coefficients or weights

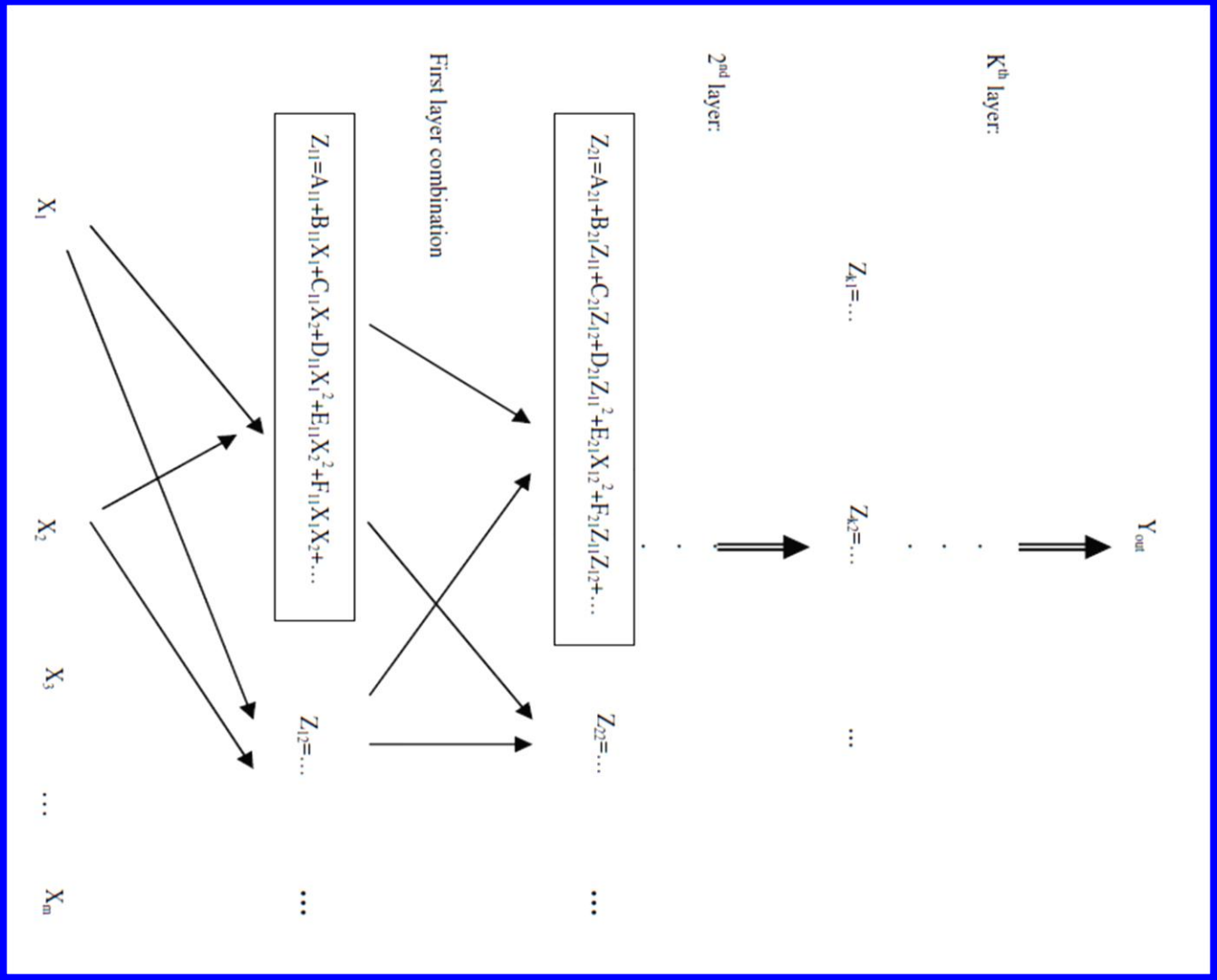
$$\mathrm{SSE} = 1/N \sum_{i=1}^{N} (y_i - y_p)^2 \rightarrow \min$$

Second Order Polynomial Form

$$y = a_0 + a_1 x_i + a_2 x_j + a_3 x_i x_j + a_4 x_i^2 + a_5 x_j^2$$

where xi and xj are input variables and y is the corresponding output value. The data points are divided into training and checking sets. The coefficients of the polynomial are found by regression on the training set and its output is then evaluated

Network Structure



Inductive Learning Algorithm

(1) Given a learning data sample including a dependent variable y and independent variables x1, x2,... ,xm split the sample into a training set and a checking set.

(2) Feed the input data of m input variables and generate combination (w,2) units from every two variable pairs at the first layer.

(3) Estimate the weights of all units (a to/in formula (la) or (lb)) using training set. Regression method can be employed.

(4) Compute mean square error between y and prediction of each unit using checking data.

(5) Sort out the unit by mean square error and eliminate bad units.

(6) Set the prediction of units in the first layer to new input variables for the next layer, and build up a multi-layer structure by applying Steps (2) (5).

(7) When the mean square error become larger than that of the previous layer, stop adding layers and choose the minimum mean square error unit in the highest layer as the final model output.

Learning A Neuron

- Matrix of data: inputs and desired value

  $u_1, u_2, u_3, \ldots, u_n, \quad y$ sample 1

  $u_1, u_2, u_3, \ldots, u_n, \quad y$ sample 2

  …. sample m

- A pair of two *u*'s are neuron's inputs x_1, x_2
- *m* approximating equations, one for each sample

  $$a\,x_1^2 + b\,x_1 x_2 + c\,x_2^2 + d\,x_1 + e\,x_2 + f = y$$

- Matrix $X\,\beta = Y$ $\beta = (a, b, c, d, e, f)^{\mathbf{t}}$
  - Each row of X is $x_1^2+x_1x_2+x_2^2+x_1+x_2+1$
- LMS solution $\beta = (X^{\mathbf{t}}X)^{-\mathbf{1}}X^{\mathbf{t}}Y$
- If $X^{\mathbf{t}}X$ is singular, we omit this neuron

GRNN

- GRNN, as proposed by Donald F. Specht, falls into the category of probabilistic neural networks.
- It needs only a fraction of the training samples a backpropagation neural network would need.
- Its ability to converge to the underlying function of the data with only few training samples available.

Algorithm

The calculations performed in each pattern neuron of GRNN are *exp(-Dj 2/2s2), the normal distribution centered* at each training sample.

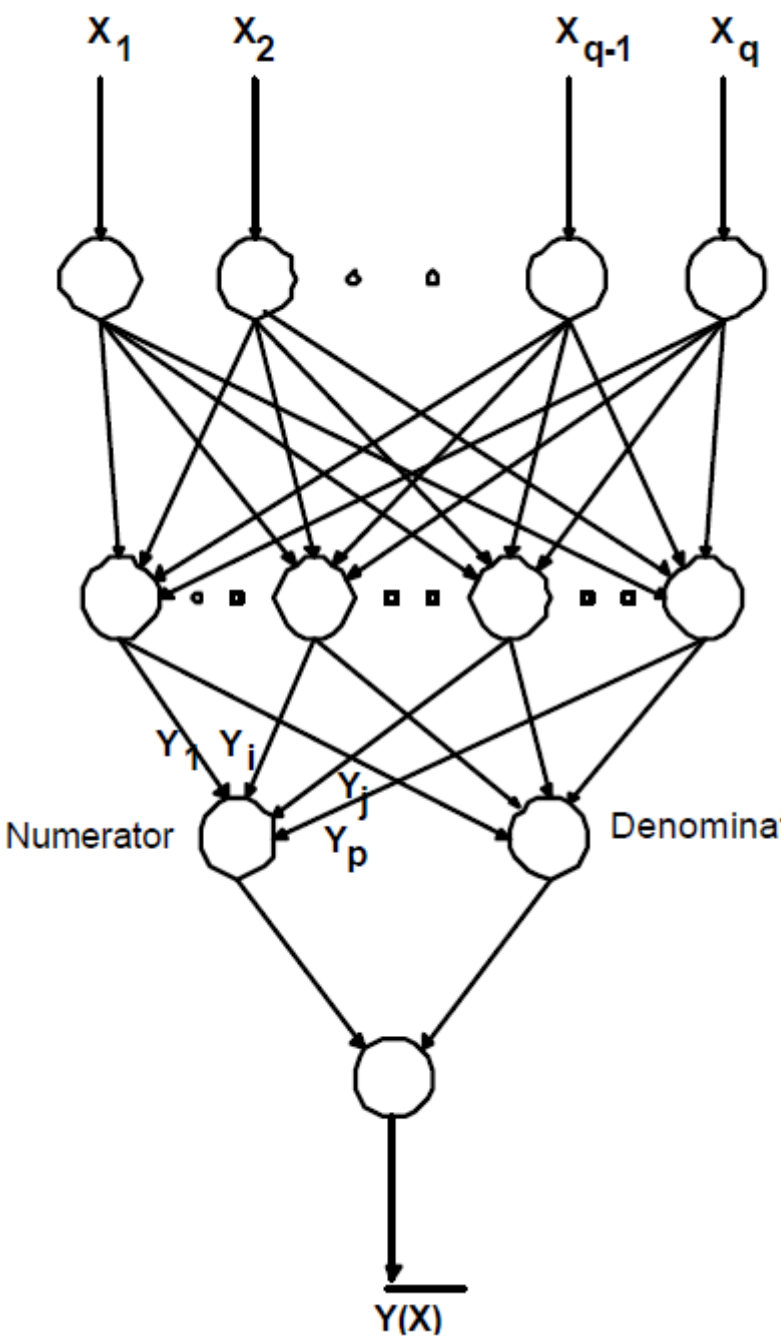
$$Y(X) = \frac{\sum_{i=1}^{n} Y_i \exp\left(-D_i^2 / 2\sigma^2\right)}{\sum_{i=1}^{n} \exp\left(-D_i^2 / 2\sigma^2\right)}$$

$$D_i^2 = \left(X - X_i\right)^T \cdot \left(X - X_i\right)$$

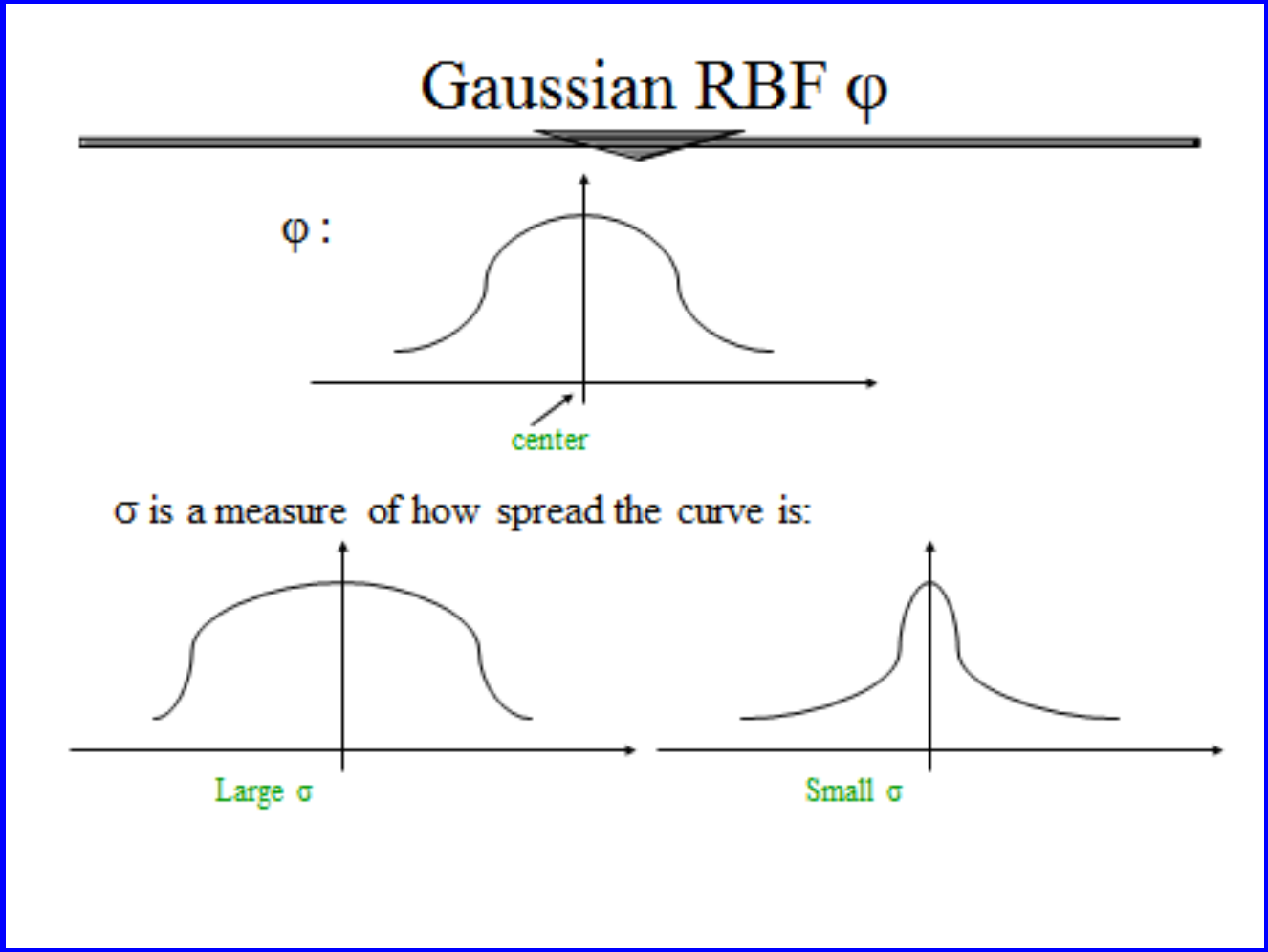
Structures of GRNN

- **Input layer** — There is one neuron in the input layer for each predictor variable. The input neurons then feed the values to each of the neurons in the hidden layer.
- **Hidden layer** — This layer has one neuron for each case in the training data set. The neuron stores the values of the predictor variables for the case along with the target value. When presented with the *x* vector of input values from the input layer, a hidden neuron computes the Euclidean distance of the test case from the neuron's center point and then applies the RBF kernel function using the sigma value(s). The resulting value is passed to the neurons in the pattern layer.

- **Pattern layer / Summation layer** — There are only two neurons in the pattern layer. One neuron is the denominator summation unit the other is the numerator summation unit. The denominator summation unit adds up the weight values coming from each of the hidden neurons. The numerator summation unit adds up the weight values multiplied by the actual target value for each hidden neuron.
- **Decision layer** — The decision layer divides the value accumulated in the numerator summation unit by the value in the denominator summation unit and uses the result as the predicted target value.


X1
X2
Xq-1
Xq
Input Neurons
Pattern Neurons with activation function: exp(-Di2/2σ2)
Y1
Yi
Yj
Yp
Numerator
Denominator
Summation Neurons
Output Neuron
Y(X)

How to choose the smooth parameter



Advantages and Disadvantages

- It is usually much faster to train a GRNN network than a multilayer perceptron network.
- GRNN networks often are more accurate than multilayer perceptron networks.
- GRNN networks are relatively insensitive to outliers (wild points).
- GRNN networks are slower than multilayer perceptron networks at classifying new cases.
- GRNN networks require more memory space to store the model.