ACI050 Artificial Intelligence

Introduction to cluster analysis

Mario González
Facultad de Ingeniería y Ciencias Ambientales
Centro de Investigación, Estudios y Desarrollo de Ingeniería (CIEDI)

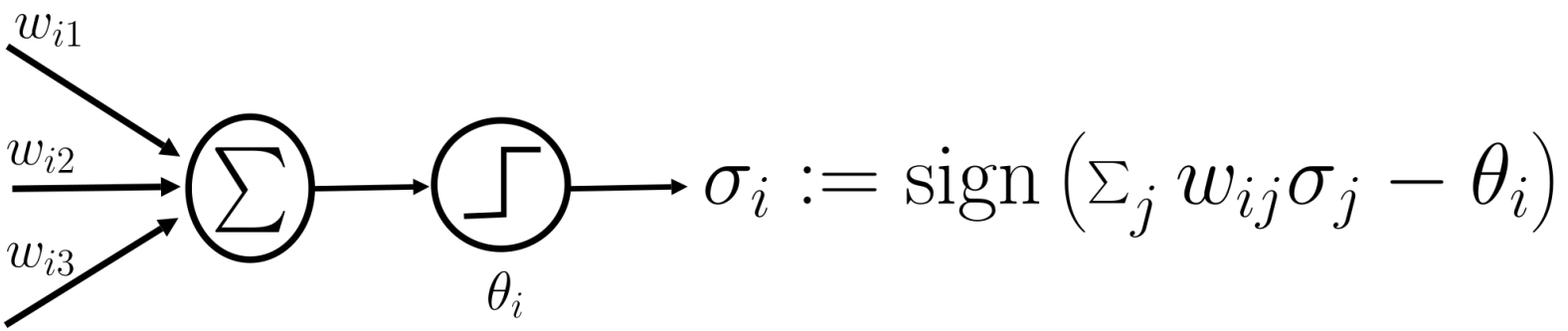
June 1, 2016

Introduction

- Artificial Neural Networks (ANN) where many simple units, called neurons, are interconnected by weighted links into larger structures are of remarkably high performance when performing classification and regression tasks.

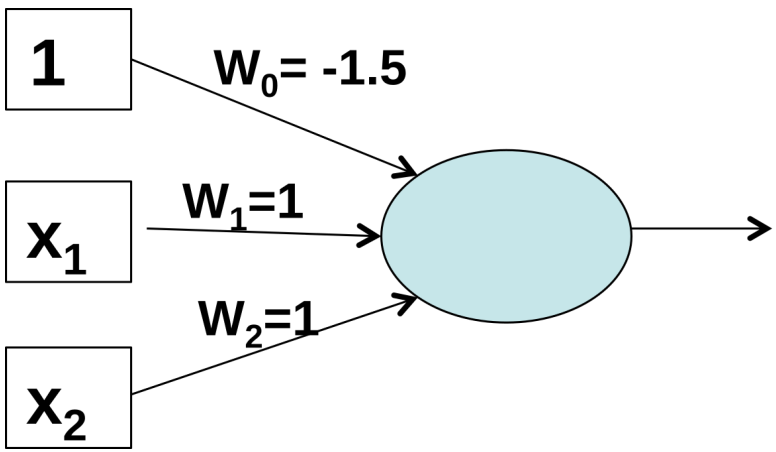
The basic unit of computation (MCP neuron)

- The basic unit of computation in ANN is the McCulloch–Pitts (MCP) neuron:



$$\mathrm{sign(z)} = 1 \text{ if } \mathrm{z} \geq 0, \text{ and } \mathrm{sign(z)} = -1 \text{ if } \mathrm{z} < 0$$

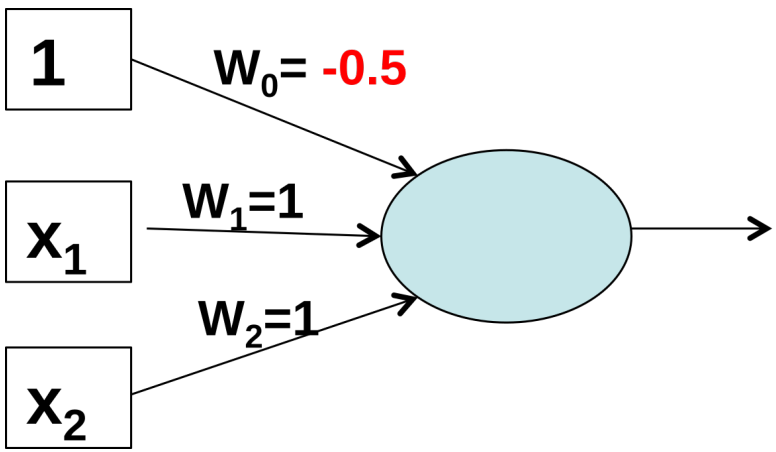
What does this do?



unit turns on when

$$\Sigma_j w_j x_j > 0$$

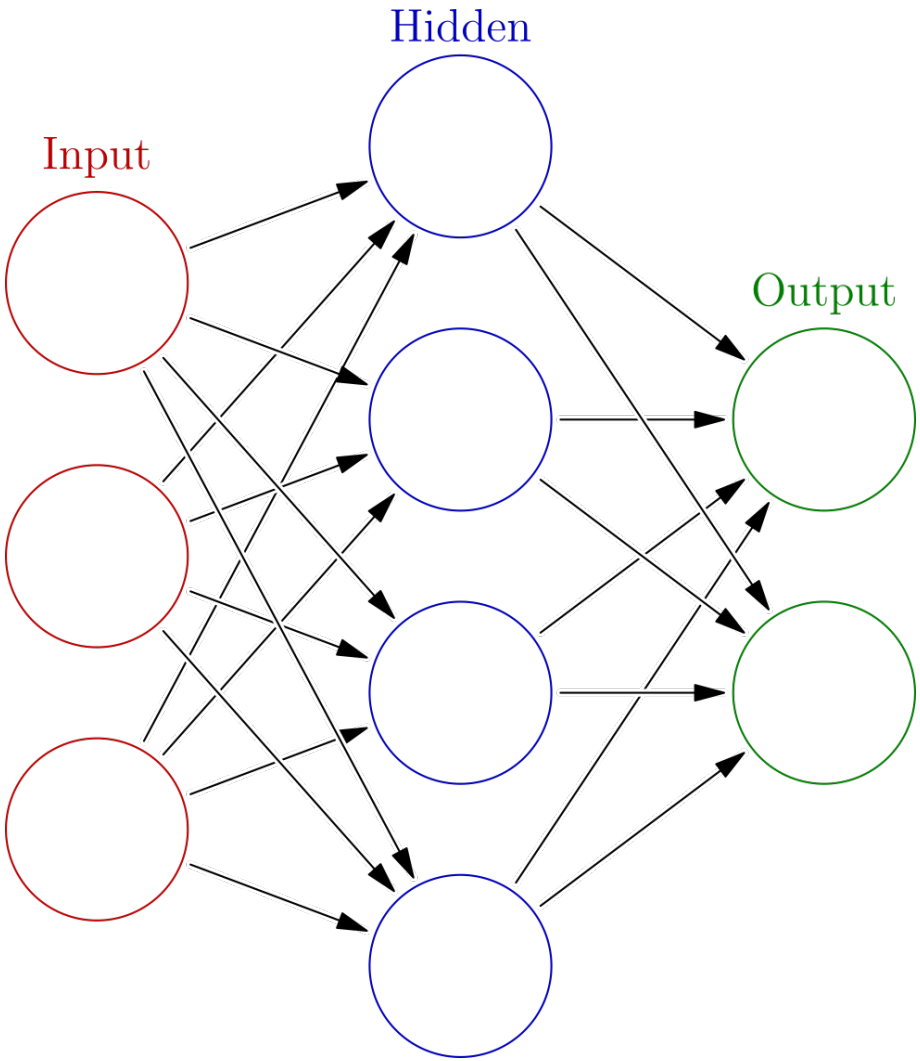
What about this one?



unit turns on when

$$\Sigma_j w_j x_j > 0$$

Artificial Neural Network (Feed-forward)



Notation: a perceptron has

- a current output or activation level, o_j
- a target or desired output level, t_j
- links from input values of weight w_{ij}
- a threshold or bias θ at which it activates
- treat the threshold as constant input of -1 connected by weight θ

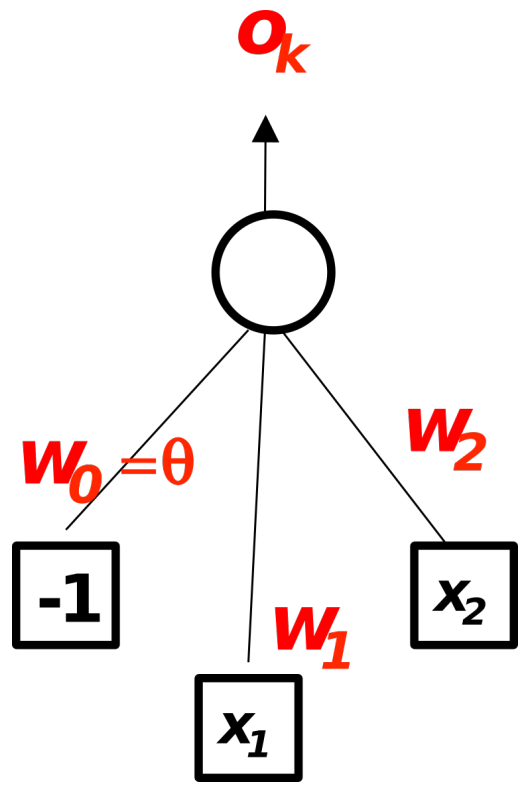
Cell Input and Output

- each cell's output o_k is determined by:

$$o_k = step(net_k)$$

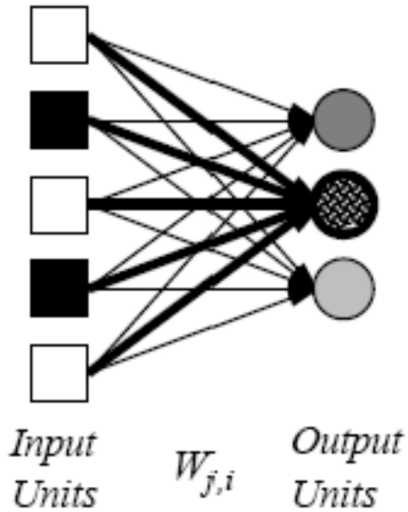
- where the net input net_k to cell k is:

$$net_k = \sum_j w_j o_j$$

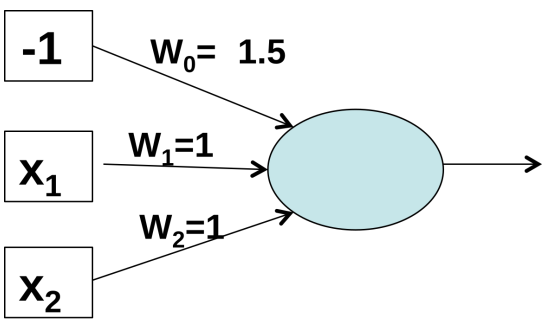


Perceptrons

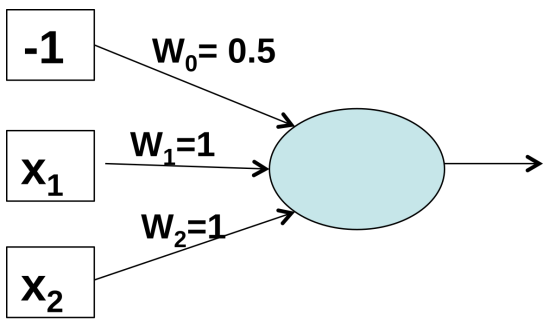
- **Perceptrons:** Single layer feed-forward networks



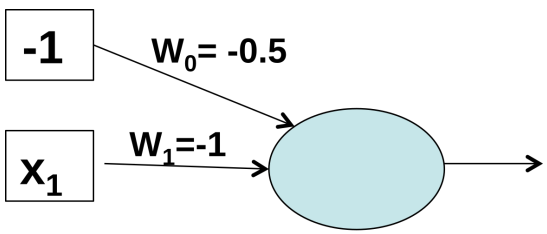
  - response given by: $o_k = step(\sum_j w_j x_j)$



AND



OR



NOT

Perceptron Learning (Delta Rule, Widrow and Hoff; Rosenblatt 1960)

- Recall

$$o_k = \sum_j w_j x_j$$

- LMS expresses error as sum of squared errors:

$$E = 1/2 Err^2 = 1/2 \sum (t - o)^2$$

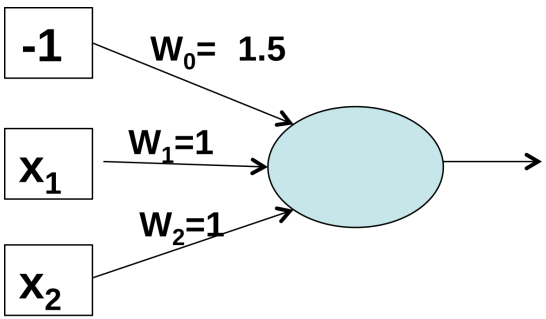
- Calculate partial of Error with respect to each weight:

$$\frac{\partial E}{\partial w_j} = \left(\frac{\partial E}{\partial o}\right)\left(\frac{\partial o}{\partial w_j}\right) = -(t - o)x_j$$

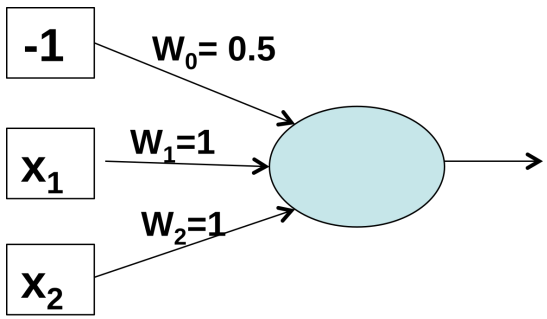
- Now modify weights along negative of error gradient, for some learning rate,α:

$$\Delta w_j = \alpha(t - o)x_j$$

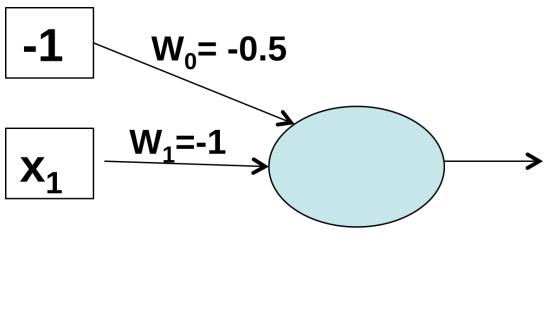
How do you build XOR? I



AND



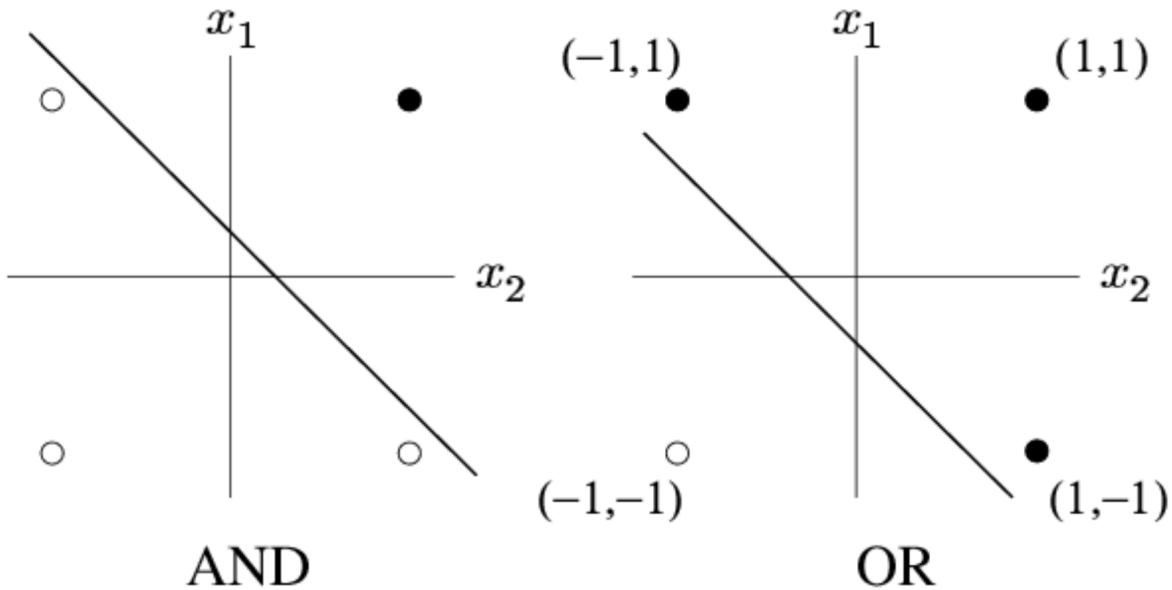
OR



NOT

How do you build XOR? II

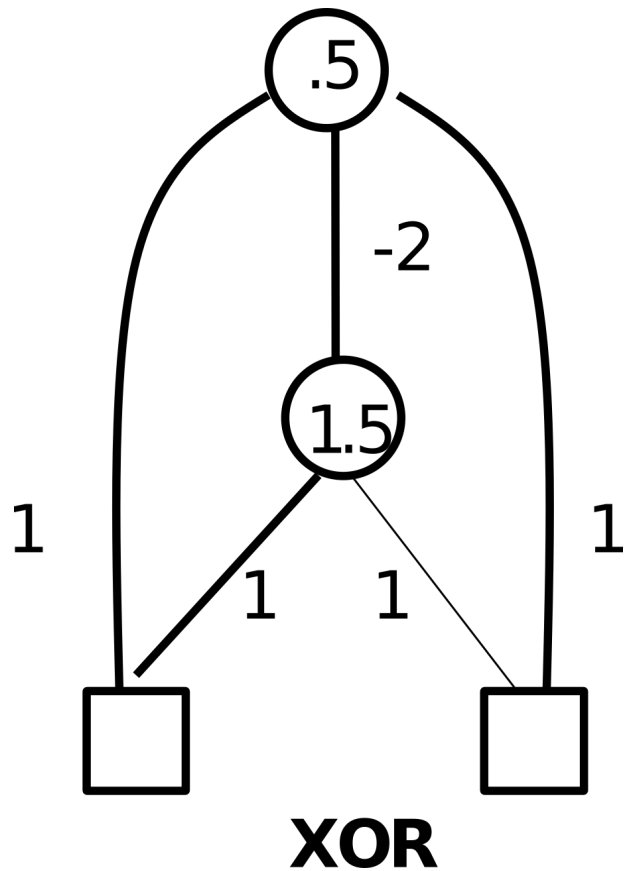
- Linear non-separability problem (Minsky and Papert, 1969)



- What about XOR?

How do you build XOR? III

- Two-layer feedforward nets
- **Multilayer perceptron** can solve XOR



Minsky and Papert's Criticism

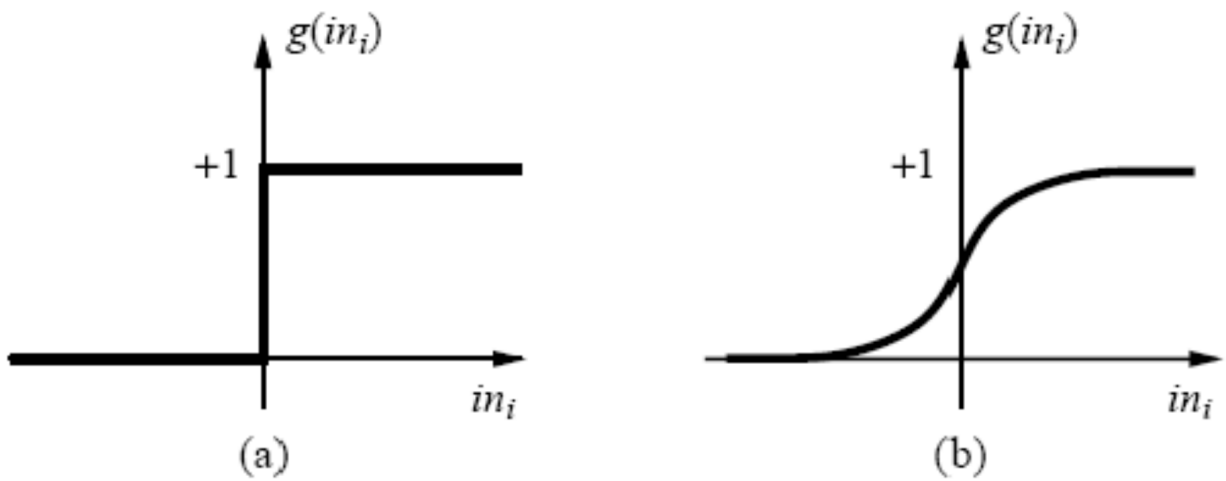
- No solution for how to learn weights for hidden units:
- How to give credit (or blame) to the hidden layer units for their contribution to the error?

Backpropagation (Werbos 1974; Rumelhart, Hinton & Williams 1986, etc.)

- Recursive method for weight adjustment in multilayer feed-forward networks
- Based on generalization of the delta rule for non-linear activation functions
- Assigns error value to output in hidden layers

Activation Functions

- Typically non-linear, monotonic (or discontinuous at one point)
- Common options:
- $tanh()$
- $sigmoid(x) = 1/(1 + e^{-x})$



Building a Multilayer Perceptron (MLP) I

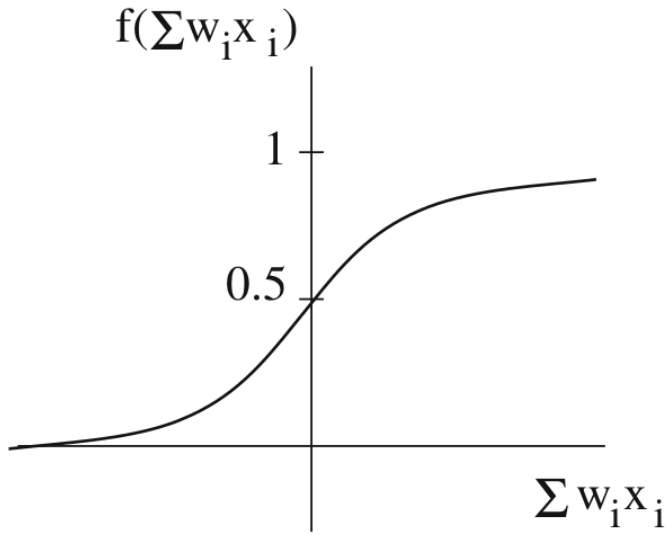
- We will suppose that all attributes are continuous. More- over, it is practical (though not strictly necessary) to assume that they have been normalized so that their values always fall in the interval $[-1, 1]$.
- **Neurons.** The function of a neuron, the basic unit of a multilayer perceptron, is

$$f\left(\sum\right) = \frac{1}{1 + e^{-\sum}}$$

- $\sum$ is the weighted sum of inputs.

Building a Multilayer Perceptron (MLP) II

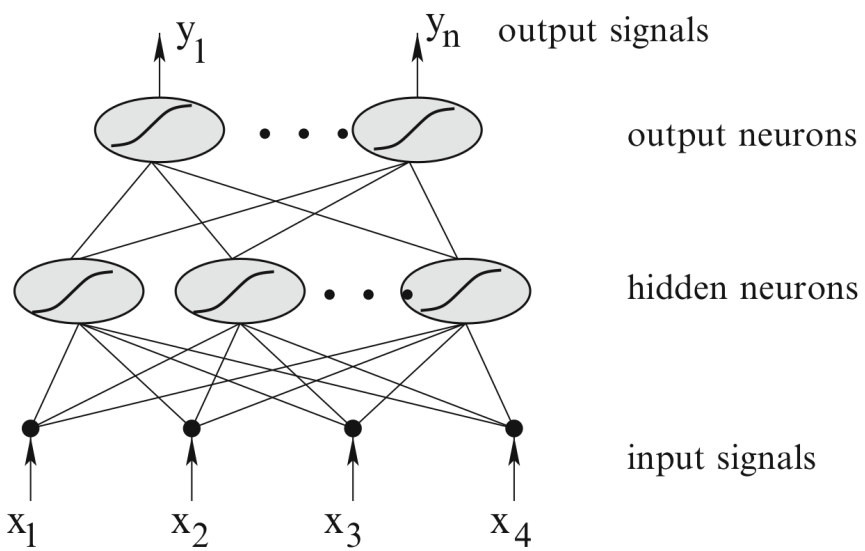
- A popular transfer function: the sigmoid



- $f(\sum)$ grows monotonically with the increasing value of $\sum$
- $f(\sum) \in (0, 1)$, $f(-\infty) = 0$, and $f(\infty) = 1$
- The vertical axis is intersected at $f(0) = 0.5$.
- It is assumed that each neuron has the same transfer function.

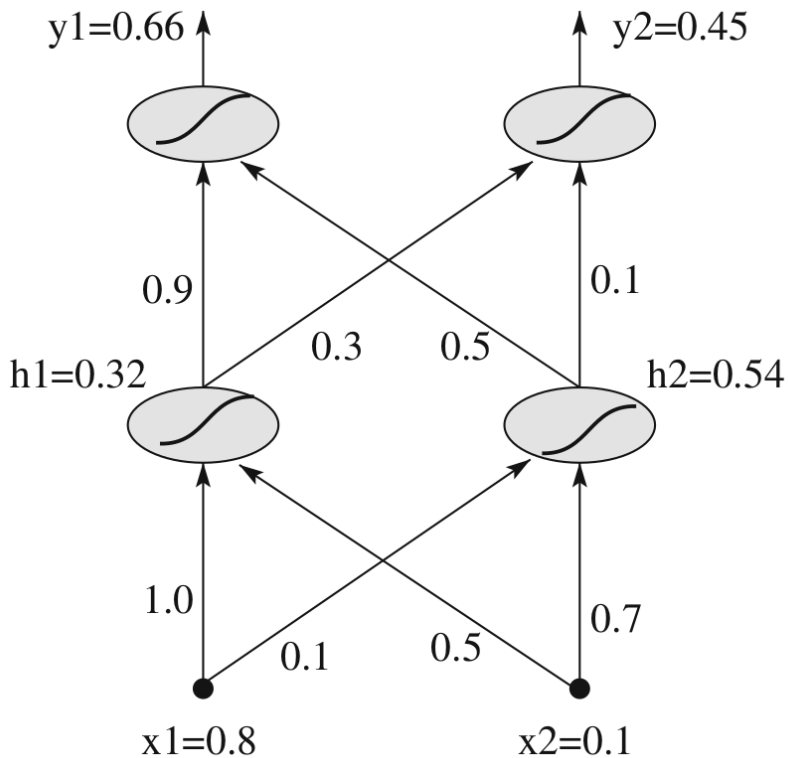
Building a Multilayer Perceptron (MLP) III

- **Multilayer perceptron.**
  - The neurons, represented by ovals, are arranged in the output layer and the hidden layer.
  - There is no communication between neurons of the same layer, adjacent layers are fully interconnected.
  - The weight of the link from the j-th hidden neuron to the i-th output neuron is denoted as w_{ji} , and the weight of the link from the k-th attribute to the j-th hidden neuron as w_{kj}.
  - Note that the first index always refers to the link's beginning; the second, to its end.



Building a Multilayer Perceptron (MLP) IV

- **Forward propagation** Forward-propagate $x = (x_1, x_2) = (0.8, 0.1)$ through the network below.



Building a Multilayer Perceptron (MLP) V

- **Error rate**.
  - Suppose we have two different networks to choose from, each with three output neurons corresponding to classes denoted by C_1, C_2, C_3.
  - For some example x, the first network outputs $y1(x) = (0.5, 0.2, 0.9)$ and the second $y2(x) = (0.6, 0.6, 0.7)$
  - This means that both will label x with the third class, $h1(x) = h2(x) = C_3$. If the correct answer is $c(x) = C_2$, both have erred, but the error does not appear to be the same.
- This is the circumstance can be captured by the error function known as the mean square error (MSE).
- **Target vector**. For instance, suppose the existence of three different classes, C_1, C_2, C_3, and let x be known to belong to C_2
  - In the ideal case, the second neuron should output 1, and the two other neurons should output 0.
  - The target is therefore $t(x) = (t_1, t_2, t_3) = (0, 1, 0)$.

Building a Multilayer Perceptron (MLP) VI

- **Mean Square Error (MSE)**. The mean square error is defined using the differences between the elements of the output vector and the target vector:

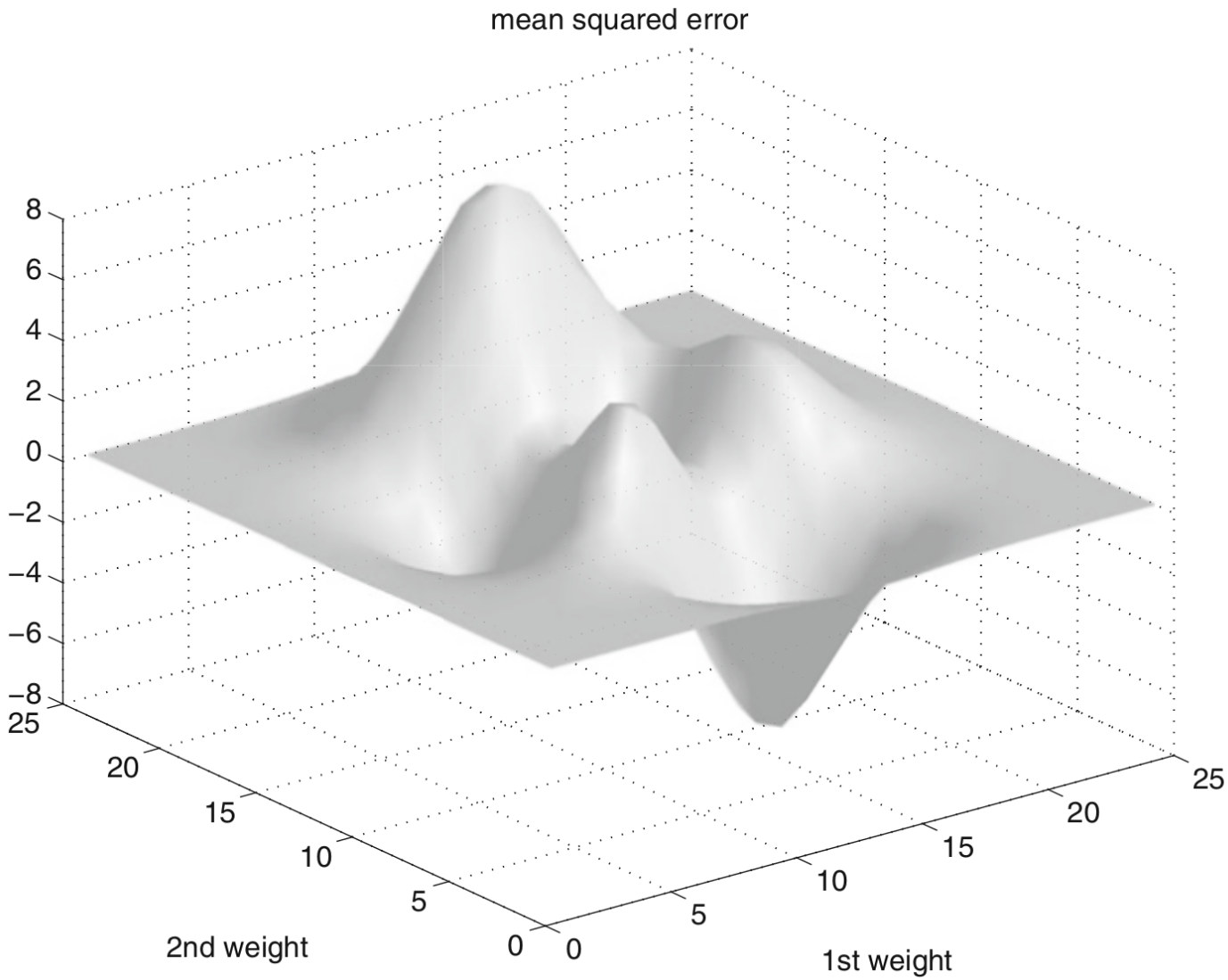
$$MSE = \frac{1}{m}\sum_{i=1}^{m}(t_i - y_i)^2$$

  - Returning to the example of the two networks mentioned before, if the target vector is $t(x) = (0, 1, 0)$, then these are the mean square errors:

$$MSE_1 = \frac{1}{3}[(0-0.5)^2 + (1-0.2)^2 + (0-0.9)^2] = 0.57$$
$$MSE_2 = \frac{1}{3}[(0-0.6)^2 + (1-0.6)^2 + (0-0.7)^2] = 0.34$$

Building a Multilayer Perceptron (MLP) VII

- **Gradient descent method.**



Building a Multilayer Perceptron (MLP) VIII

- **Backpropation of error.**

1. Present example $\mathbf{x}$ to the input layer and propagate it through the network.
2. Let $\mathbf{y} = (y_1, \ldots y_m)$ be the output vector, and let $\mathbf{t}(\mathbf{x}) = (t_1, \ldots t_m)$ be the target vector.
3. For each output neuron, calculate its responsibility, $\delta_i^{(1)}$, for the network's error:
   $\delta_i^{(1)} = y_i(1 - y_i)(t_i - y_i)$
4. For each hidden neuron, calculate its responsibility, $\delta_j^{(2)}$, for the network's error. While doing so, use the responsibilities, $\delta_i^{(1)}$, of the output neurons as obtained in the previous step.
   $\delta_j^{(2)} = h_j(1 - h_j) \sum_i \delta_i^{(1)} w_{ji}$
5. Update the weights using the following formulas, where η is the learning rate:

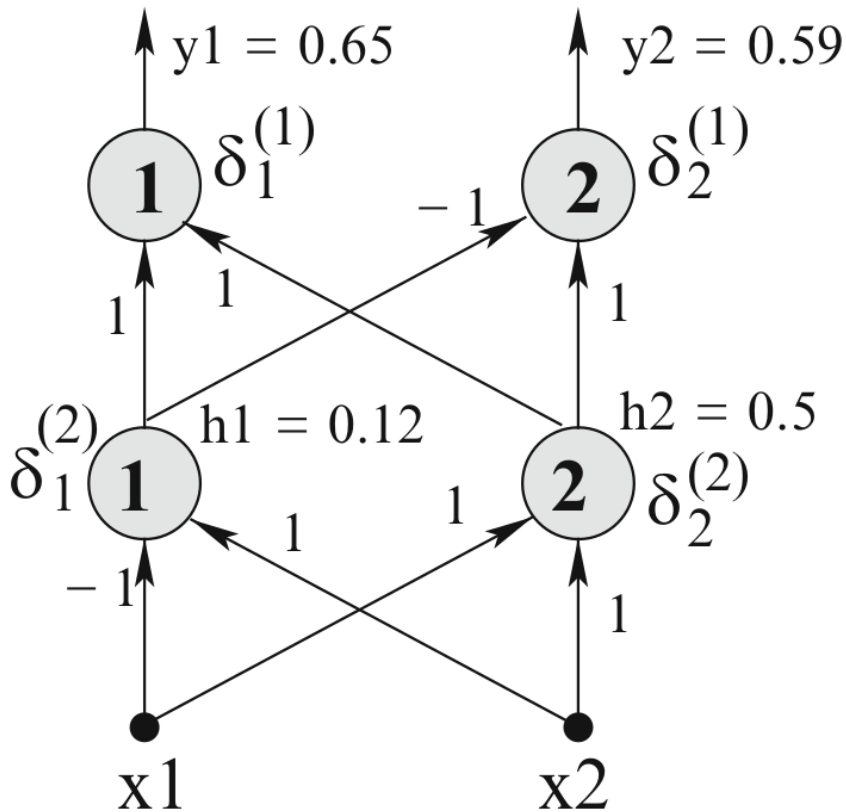
   output layer: $w_{ji}^{(1)} := w_{ji}^{(1)} + \eta \delta_i^{(1)} h_j$; h_j: the output of the j-th hidden neuron
   hidden layer: $w_{kj}^{(2)} := w_{kj}^{(2)} + \eta \delta_j^{(2)} x_k$; x_k: the value of the k-th attribute

6. Unless a termination criterion has been satisfied, return to step 1.

Building a Multilayer Perceptron (MLP) IX

- **Example of backpropagation.**

Task. In the neural network below, let the transfer function be $f(\Sigma) = \frac{1}{1+e^{-\Sigma}}$. Using backpropagation of error (with $\eta = 0.1$), show how the weights are modified after the presentation of the following example: $[\mathbf{x}, \mathbf{t}(\mathbf{x})] = [(1, -1), (1, 0)]$



Special Aspects of Multilayer Perceptrons I

- **Computational costs.** Backpropagation of error is computationally expensive. Upon the presentation of an example, the responsibility of each individual neuron has to be calculated, and the weights then modified accordingly. This has to be repeated for all training examples, usually for many epochs.
- **Target values.** For simplicity, we have so far assumed that each target value is either 1 or 0. This may not be the best choice. More appropriate values for the target are recommended: for instance, $t_i(x) = 0.8$ if the example belongs to the i-th class, $t_i(x) = 0.2$. Both 0.8 and 0.2 find themselves in regions of relatively high sensitivity of the sigmoid function, and as such will eliminate most of the concerns raised in the previous paragraph.

Special Aspects of Multilayer Perceptrons II

- **Local minima.** The main drawback of the gradient-descent approach when adopted by multilayer perceptron training is that the weights are changed in a way that guarantees descent along the steepest slope. But once the bottom of a local minimum has been reached, there is nowhere else to go which is awkward: after all, the ultimate goal is to reach the global minimum. Techniques to overcome this difficulty usually rely on adaptive learning rates. Generally speaking, the problem is less critical in networks with many hidden neurons. Also, local minima tend to be shallower, and less frequent, if all weights are very small, say, from the interval (-0.01,0.01).

Special Aspects of Multilayer Perceptrons III

- **Adaptive learning rate.** Very often, the training starts with a large η which then gradually decreases in time. The motivation is easy to guess. At the beginning, greater weight-changes reduce the number of epochs, and they may even help the learner to jump over some local minima. Later on, however, this large η might lead to overshooting the global minimum, and this is why its value should be decreased. We can define the learning rate as a function of time, $\eta(t)$ as:

$$\eta(t) = \eta(0)e^{-\alpha t}$$

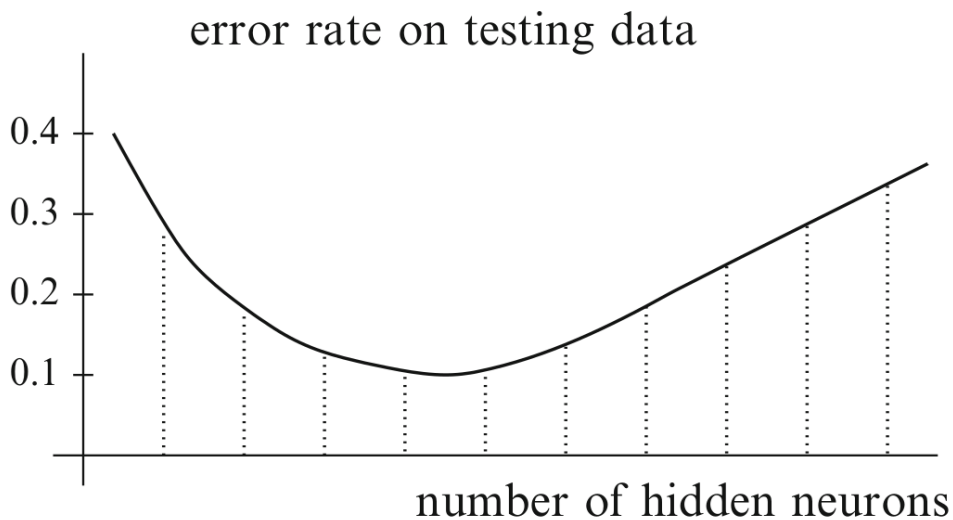
α is the slope of the negative exponential, and $\eta(0)$ is the learning rate's initial value.

Special Aspects of Multilayer Perceptrons IV

- **Overtraining.** Multilayer perceptrons share with polynomial classifiers one unpleasant property. Theoretically speaking, they are capable of modeling any decision surface, and this makes them prone to overfitting the training data. The reader remembers that overfitting typically means perfect classification of noisy training examples, which is inevitably followed by disappointing performance in the future.
  - For small multilayer perceptrons, this problem is not as painful; they are not flexible enough to overfit. But as the number of hidden neurons increases, the network gains in flexibility, and overfitting can become a real concern. However, this does not mean that we should always prefer small networks!
  - There is a simple way to discover whether the training process has reached the overfitting stage. If the training set is big enough, we can afford to leave aside some 10–20 % examples and using them as test.

Architectural Issues I

- **Optimum number of neurons.** Typically, the error rate will be high in the case of very small networks because these lack adequate flexibility, and also suffer from the danger of getting stuck in local minima. These two weaknesses can be mitigated if we increase the number of hidden neurons. As shown in the graph, the larger networks then exhibit lower error rates. But then, networks that are too large are vulnerable to overfitting. This is why, after a certain point, the testing-set error starts growing again.



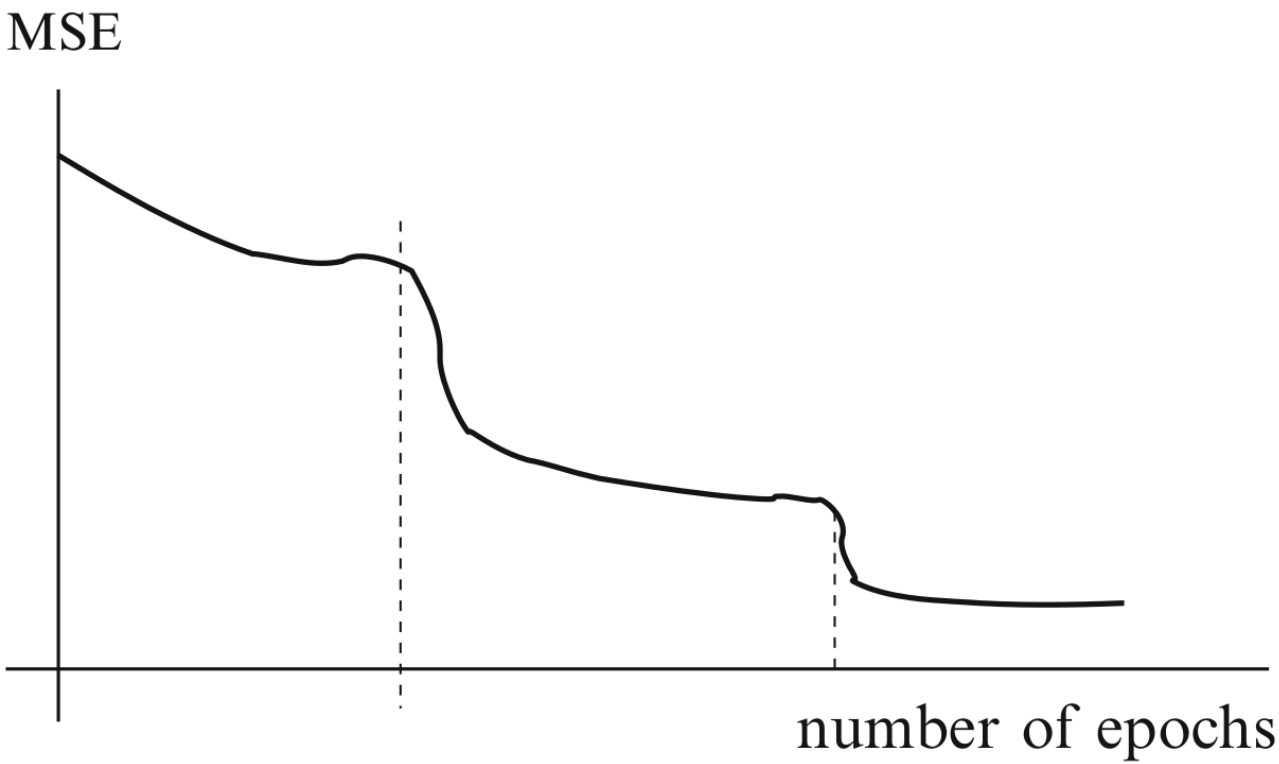
Architectural Issues II

- **Search for appropriate size.** The scenario described above is too expensive to be employed in practical applications. Instead, we would like to have at our disposal a technique capable of finding the appropriate size more efficiently as presented in the following pseudo-code:

  1. At the beginning, use only a few hidden neurons, say, five.
  2. Train the network until the mean square error no longer seems to improve.
  3. At this moment, add a few (say, three) neurons to the hidden layer, each with randomly initialized weights, and resume training.
  4. Repeat the previous two steps until a termination criterion has been satisfied. For instance, this can be the case when the new addition does not result in a significant error reduction, or when the hidden layer exceeds a user-set maximum size.

Architectural Issues III

- The following figure illustrate the behavior of the pseudo-code before:



Some final recommendations I

- Scaling the input: The input is scaled, that is, we standardize the training patterns so that the average (over the training set) of each feature is zero and to have the same variance in each feature component to be 1.
- Weights initialization: The weights are initialized using Nguyen-Widrow Randomization This technique is one of the most effective neural network weight initialization methods available.
- Learning rate: The learning rate value determines the speed at which the network attains a minimum. In practice, because networks are rarely fully trained to a training error minimum the learning rate can affect the quality of the final network.
- Number of hidden units: A convenient rule to choose the number of hidden units h is $h = \frac{numberofinstances \times 0.05}{InputUnits + OutputUnits}$.

Some final recommendations II

- Momentum allows the network to learn more quickly (as error surfaces often have plateaus). The approach is to alter the learning rule in stochastic backpropagation to include some fraction of the previous weight update.
- Weight decay: One method of simplifying a network and avoiding overfitting is to impose a heuristic that the weights should be small.

Sources and Resources

- McGill University Artificial Intelligence - ECSE 526.
- Kubat, M. (1992). Introduction to machine learning. In Advanced Topics in Artificial Intelligence (pp. 104-138). Springer Berlin Heidelberg.
- Duda, R. O., Hart, P. E., & Stork, D. G. (2012). Pattern classification. John Wiley & Sons. Chicago
- Pattern Recognition and Machine Learning. C. Bishop.