Neural Networks

History of NN

- Begining
  - Mc Culloch & Pitts (1943) : First model of formal neuron.
  - Rule of Hebb (1949) : Reinforcement learning of connexion.
- First realisations
  - ADALINE (Widrow-Hoff, 1960)
  - PERCEPTRON (Rosenblatt, 1958-1962)
  - Analysis of Minsky & Papert (1969)
- New models
  - Kohonen (Competitive learning)
  - Hopfield (1982) (Loop network)
  - Multi-Layers Perceptron (1985)
- Analysis and development
  - Control theory, generalization (Vapnik), ...

Why the neural networks?

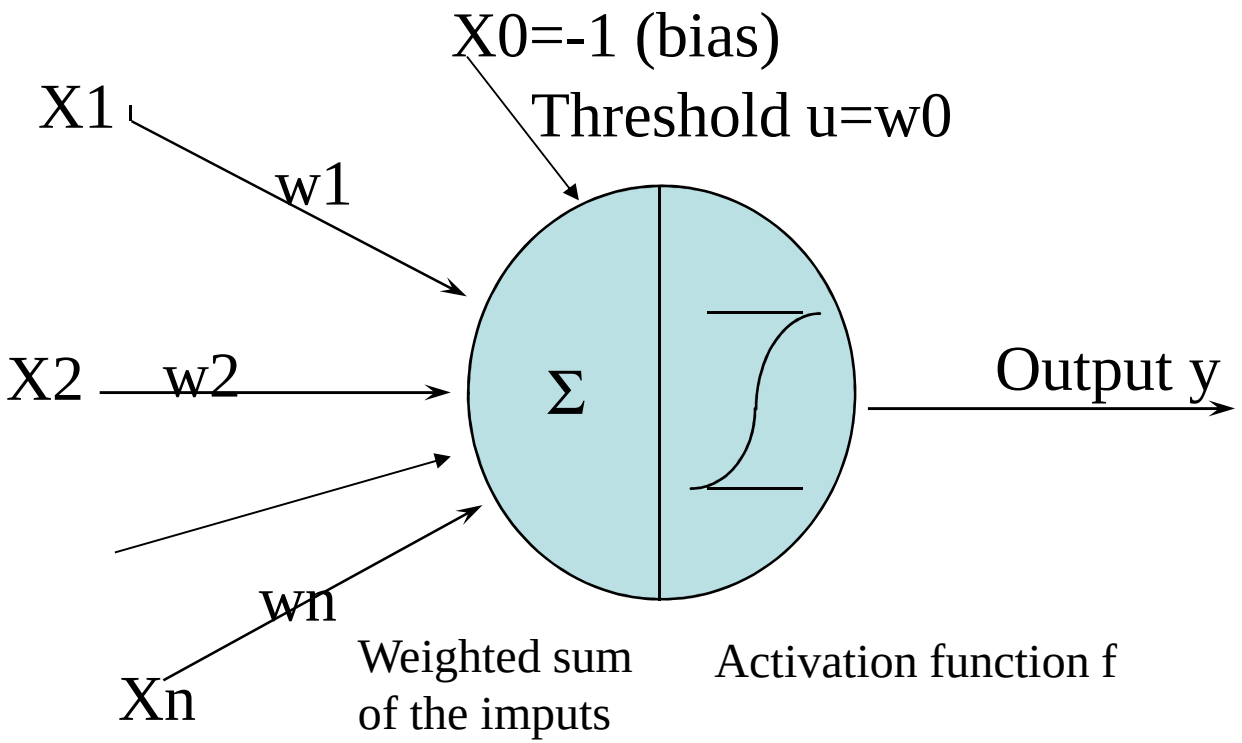
Biological inspiration

- The human brain: a very interesting model
  - Robuste and tolerant to errors
  - Flexible. Easily ajustable.
  - Deal with incomplete and noisy data
  - Massive parrallel computing
  - Learning skill
- Neurons
  - ≈ 10^{11} neurons in human brain
  - ≈ 10^{4} connexions / neurone
  - Activation potential
  - Excitation / inhibition signal

Why the neural networks?

- Advantages
  - Parrallel computing
  - Direct implementation on chips
  - Robust and tolerent to errors
  - Simples algorithme
  - Generability of the use
- Drawbacks
  - Black box
  - Interpretation of the results

Formal Neuron

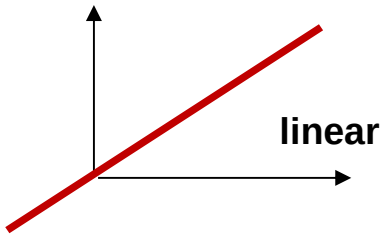


$$y = g\left(\sum_{i=0}^{n} w_i x_i\right) = g(a)$$

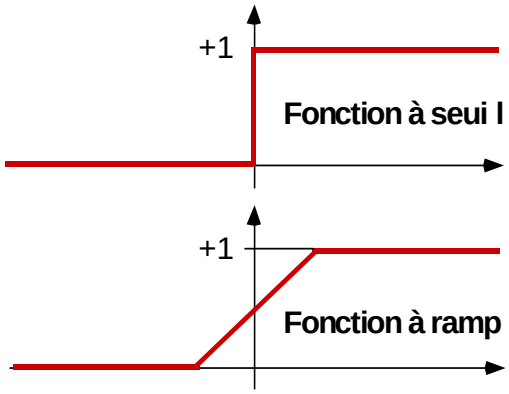
Formal neuron function

Types of fonction:

$$g(a) = a$$

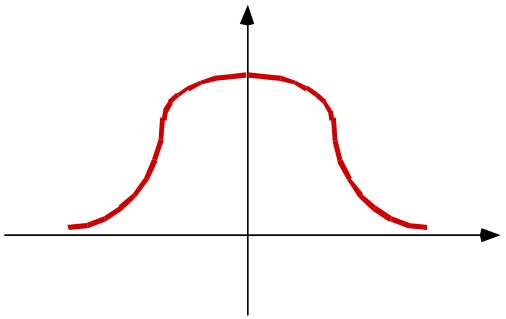


$$g(a) = I_{a>t}$$



$$g(a) = \frac{1}{1 + e^{-a}}$$

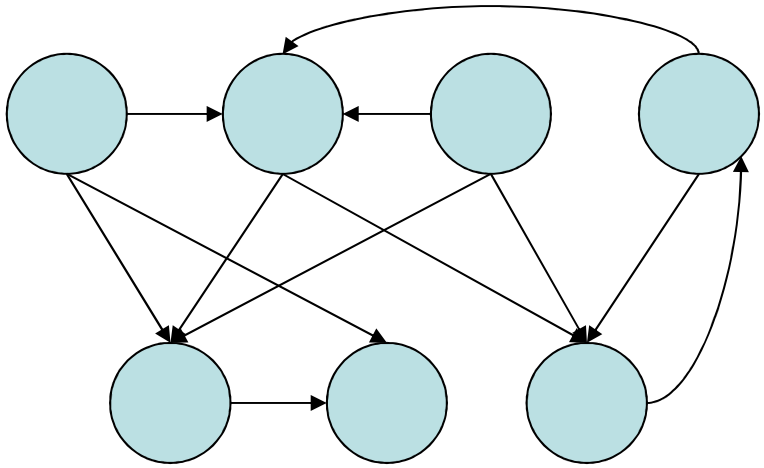
$$g(a) = g(\mathrm{a})(1\text{-}g(a))$$

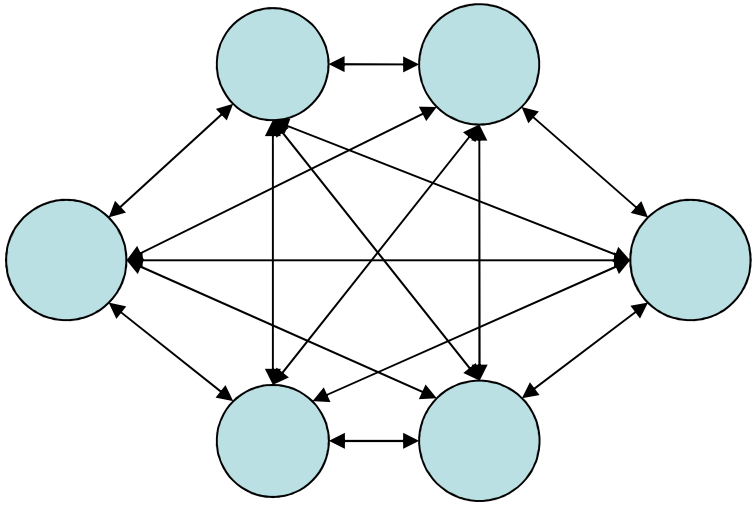


Fonction à base radial e

$$f(a) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left[-\frac{1}{2}\left(\frac{a-\mu}{\sigma}\right)\right]$$

Types of network: feedback network

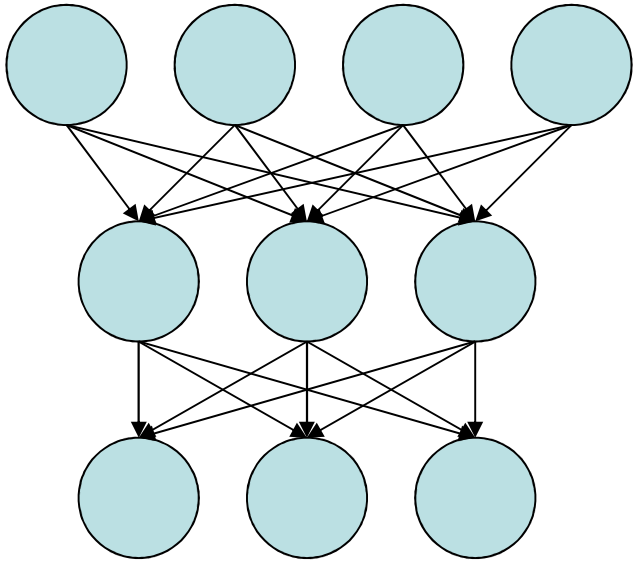




propagation of activation:

- synchronous : All neurons are updated simultaneously
- asynchronous : The neurons are updated sequentially

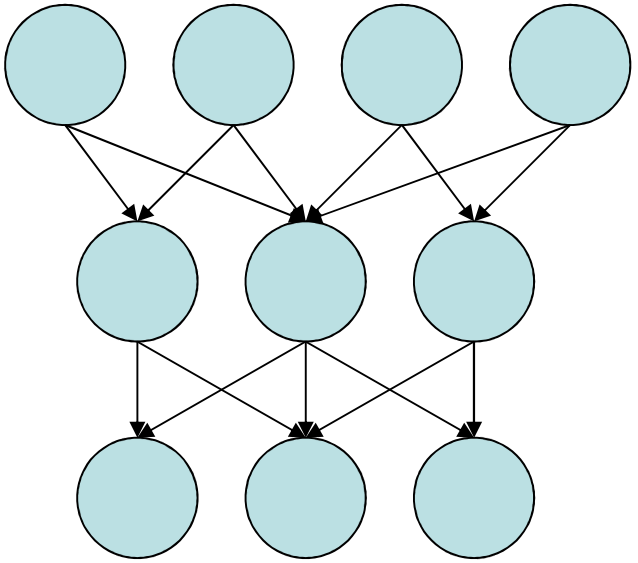
Types de réseaux: feedforward



Imput layer

Hidden layer

Output layer



Multilayer network

Local connexion network

propagation of activations : From imput to output

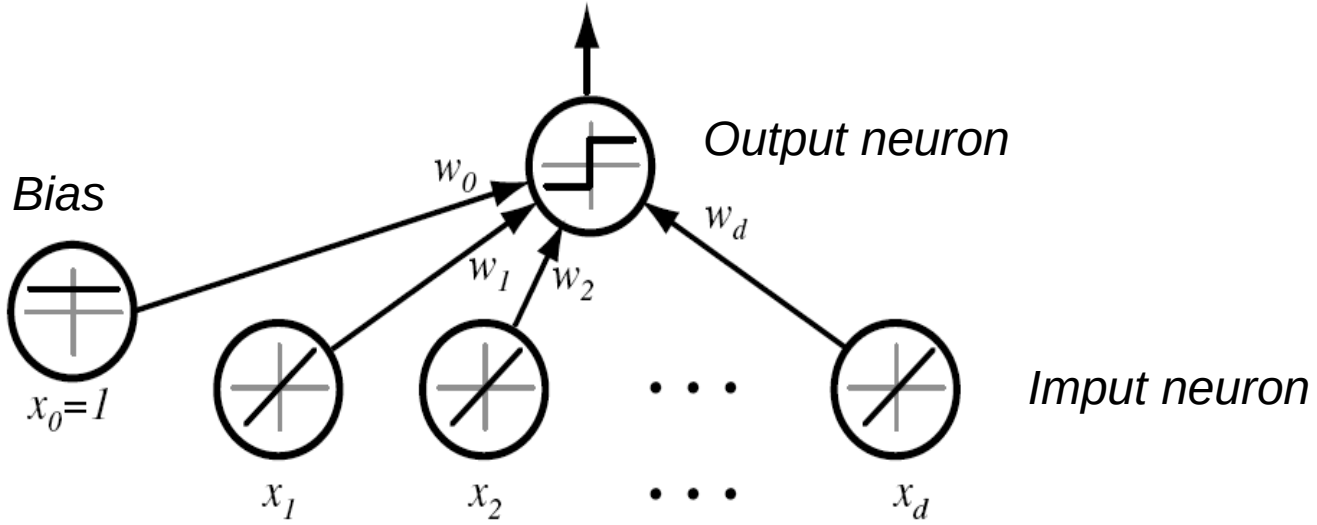
Linear discriminant : Perceptron

[Rosenblatt, 1957,1962]

$$g(x) = w_0 + \sum_{i=1}^{d} w_i x_i$$

- Decision function

$$f(\mathbf{x}) = \begin{cases} C_1 & \text{si } g(\mathbf{x}) > 0, \\ C_2 & \text{si } g(\mathbf{x}) \leq 0 \end{cases} = \begin{cases} C_1 & \text{si } \mathbf{w}^t\mathbf{x} > -w_0, \\ C_2 & \text{si } \mathbf{w}^t\mathbf{x} < -w_0 \end{cases}$$

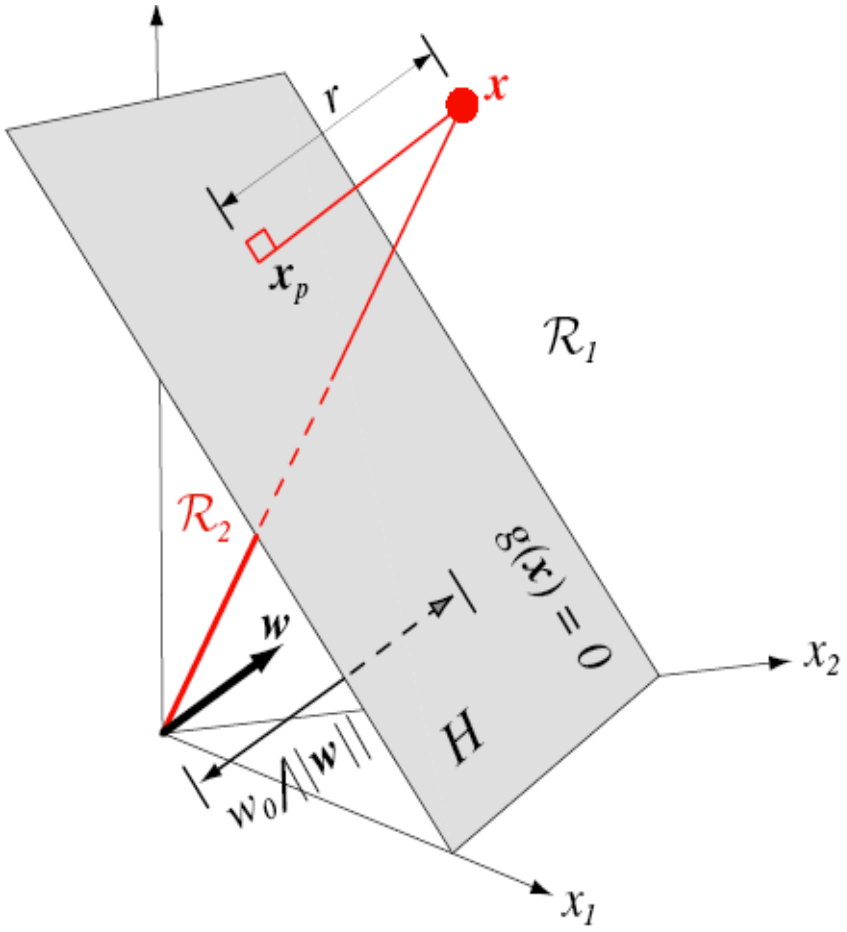


Linear discriminant : Perceptron

- Geometry – 2 classes
  - The decision function H is a hyperplan g(x)=0
  - $x_1, x_2 \epsilon H : w^t(x_1 - x_2) = 0$
  - R = algebric distance from x to H:

$$\begin{aligned} \mathbf{x} &= \mathbf{x}_p + r\frac{\mathbf{w}}{\|\mathbf{w}\|} \\ g(\mathbf{x}) &= \mathbf{w}^t\mathbf{x} + w_0 = r\|\mathbf{w}\| \\ r &= \frac{g(\mathbf{x})}{\|\mathbf{w}\|} \end{aligned}$$

Linear discriminant : Perceptron



Perceptron : performance criteria

Optimisation criteria (erreur function) :

- We do not use the error rate
- Criteria : $R_{Emp}(\boldsymbol{w}) = -\sum_{\boldsymbol{x}_j \in \mathrm{M}} \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_j \,.\, y_j$

Objective:

Find (w) minimizing R_{emp}

Direct learning

- Requirement
  - Knowledge of all pairs (x_p, y_p)
  - Inversion of the matrix

- Direct learning can be use only on a linear network and a quadratic error function

- We need an iterative method without matrix inversion

  → **Gradient descent**

Perceptron : algorithm

- Heuristic of exploration of error surface
  - Gradient descent
    - Minimization of the error function
    - Principle : Computation of the error in function of the connexion weight
    - Learning step by step in the gradient direction
  - Pseudo-Algorithm :

    Repeat

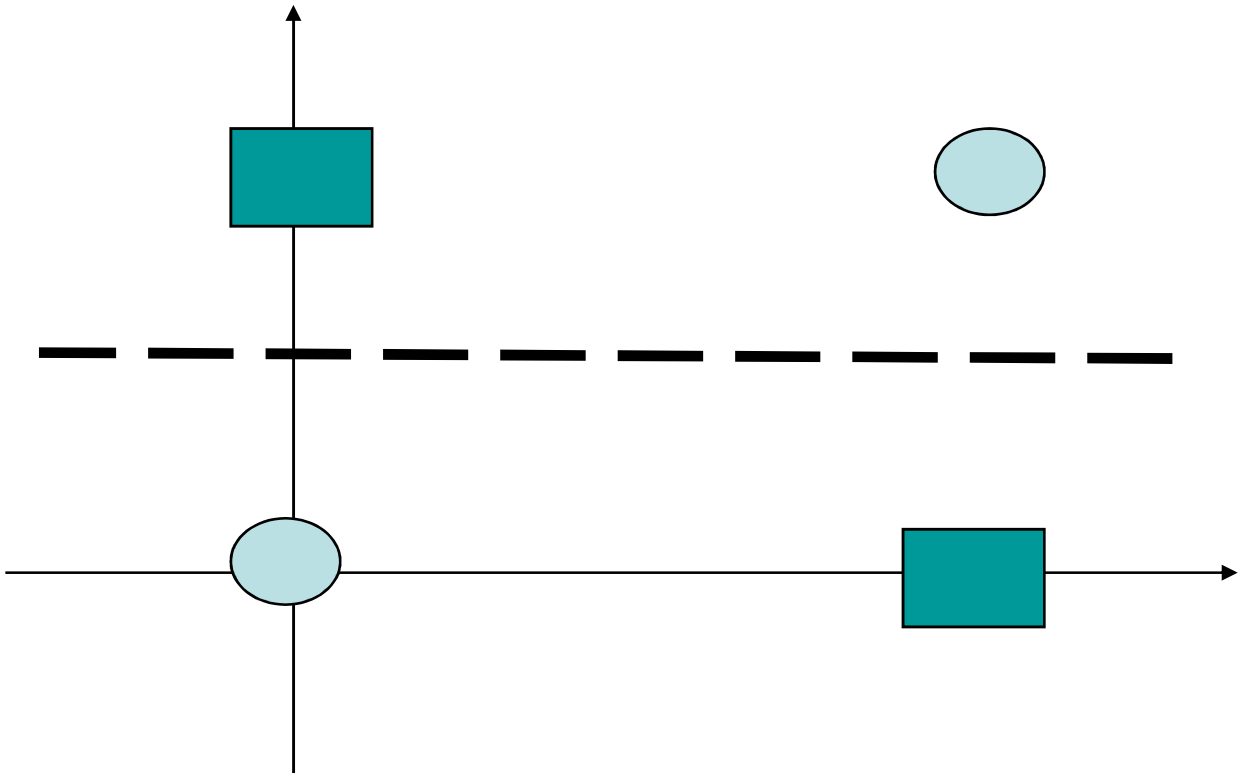
    If the example is not correctly classified

    Then updated the weight $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \eta \nabla_t$

    Until the error function reaches a given threshold

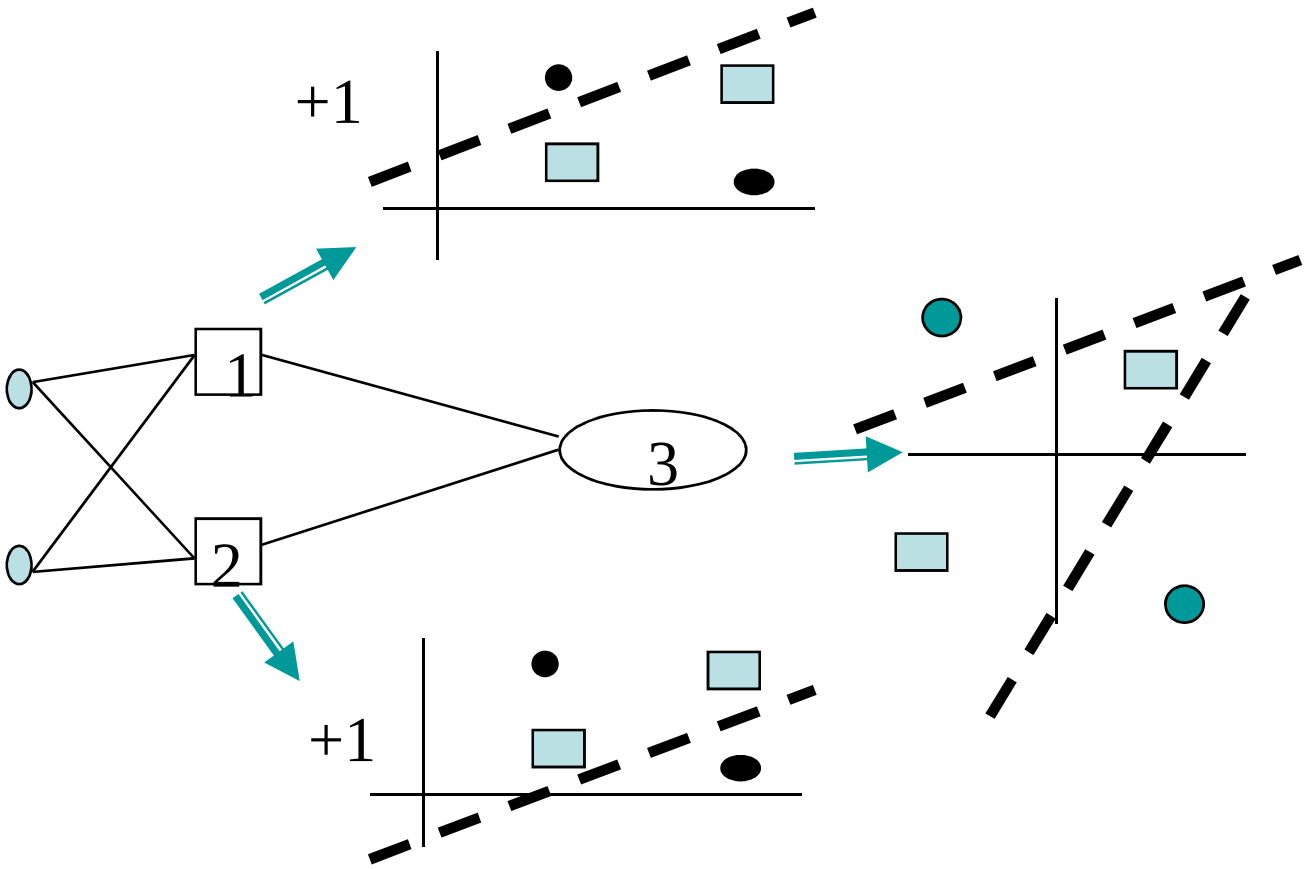
  - Convergence ?

Problem XOR



The XOR problem cannot be solve by a linear perceptron

Problem XOR

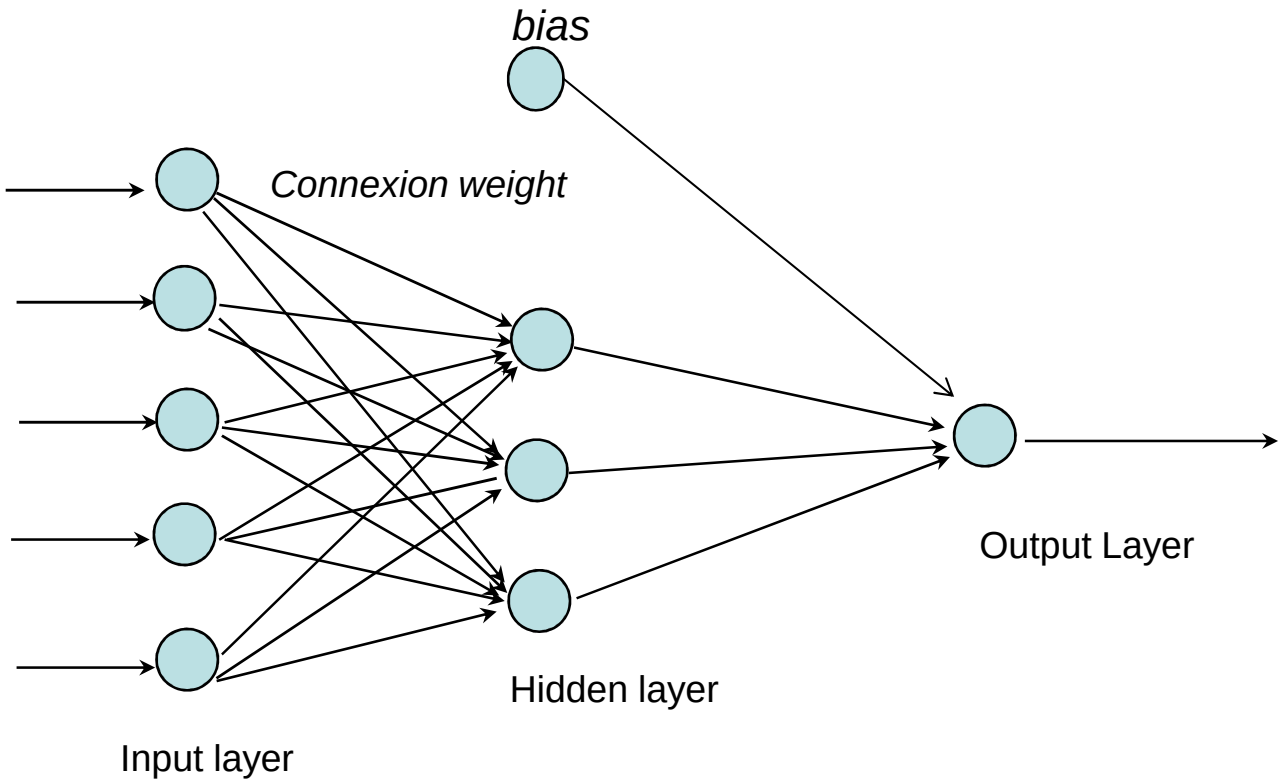


Minsky & Papert (1969) found the solution to the XOR problem.
Combination of several perceptrons in using a hidden neuron layer.

Multilayer Perceptron

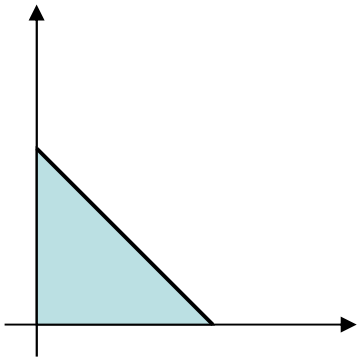
- Network where neurons are organizes in successive layers
- Input layer: Variables of the problem
- Output layer: Value to estimate
- Hidden layer: All neurons of a layer are the imput of the next layers and the output of the previous layer.

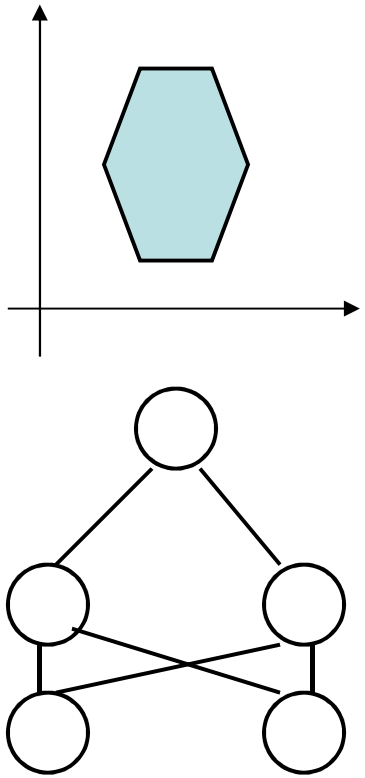
Multilayer Perceptron

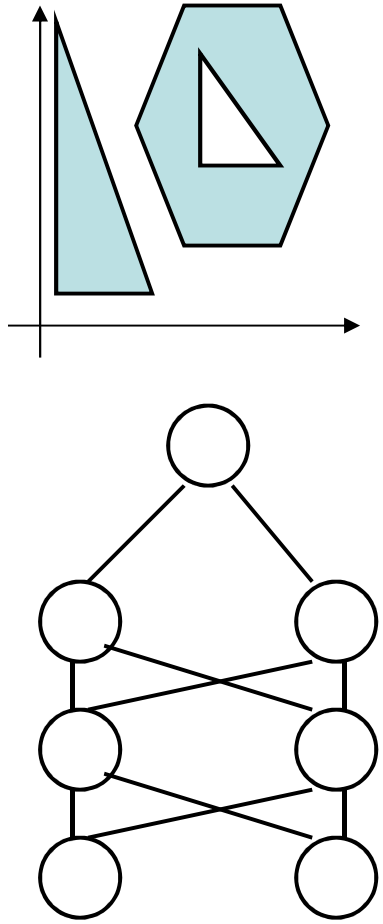


$$y = w_0 + \sum_{i=1}^{m} w_i g\left(w_{0j} + \sum_{j=1}^{n} w_{ji} x_j \right)$$

MLP: decison function complexity







1 layer

2 layers

3 layers

MLP: decison function complexity

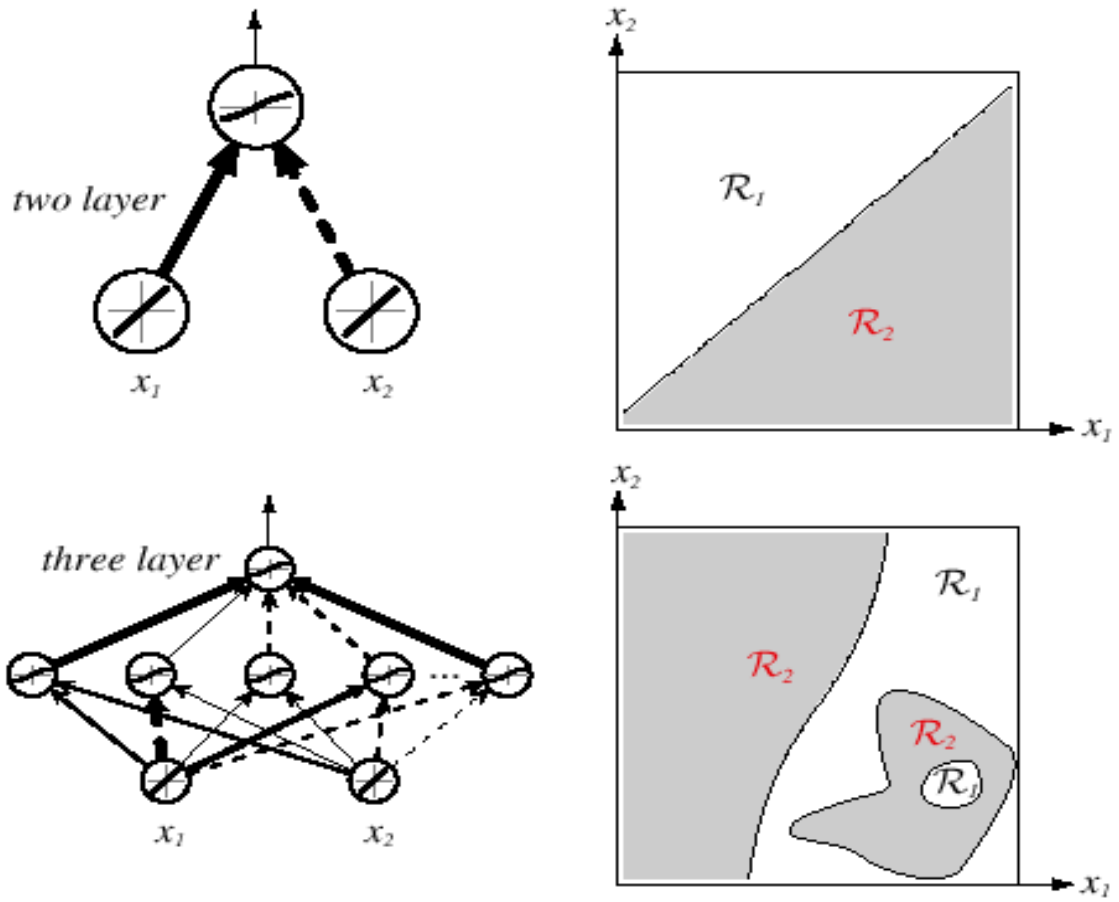


FIGURE 6.3. Whereas a two-layer network classifier can only implement a linear decision boundary, given an adequate number of hidden units, three-, four- and higher-layer networks can implement arbitrary decision boundaries. The decision regions need not be convex or simply connected. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

MLP: learning

- Find the weights that verify the relation input-output specified by the training set

- Learning:
  - Minimization of the error function $E(w,\{x^l,y^l\})$ in function of the parameter w
  - Use the method of Gradient descent

$$\Delta w_{ij} \propto -\partial E / \partial w_{ij}$$

(algorithm of *gradient retro-propagation*)

Learning: Gradient descent

- Learning = exploration of the parameter space in order to minimize the error function
- Gradient descent method:
  - Optimal solution *w** tq. :

$$\nabla E(\boldsymbol{w}^*) = \mathbf{0}$$

$$\nabla = \left[\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \ldots, \frac{\partial}{\partial w_N}\right]^T$$

$$E^{(\tau+1)} = E^{(\tau)} - \nabla_{\boldsymbol{w}} E$$

$$w_{ij}^{(\tau+1)} = w_{ij}^{(\tau)} - \eta \left.\frac{\partial E}{\partial w_{ij}}\right|_{\boldsymbol{w}^{(\tau)}}$$

Learning: Gradient descent

Objective : $$\boldsymbol{w}^* = \arg\min_{w} \frac{1}{m} \sum_{l=1}^{m} [\boldsymbol{y}_l - f(\boldsymbol{x}_l, w)]^2$$

Iterative Algorithm

Algorithme itératif : $$\boldsymbol{w}^{(t)} = \boldsymbol{w}^{(t-1)} - \eta \nabla_E \boldsymbol{w}^{(t)}$$

Off-line case

$$w_{ij}(t) = w_{ij}(t-1) - \eta(t) \frac{1}{m} \sum_{k=1}^{m} \frac{\partial R_E(x_k, \boldsymbol{w})}{\partial w_{ij}}$$

where:

$$R_E(x_k, \boldsymbol{w}) = [y_k - f(x_k, \boldsymbol{w})]^2$$

On-line case

$$w_{ij}(t) = w_{ij}(t-1) - \eta(t) \frac{\partial R_E(x_k, \boldsymbol{w})}{\partial w_{ij}}$$

Learning: Gradient descent

- Presentation of an training exemple

  Sequential, random, criteria…
- Compute the output of the network
- Compute the error = f(output, target output)
- Compute the gradients

  Algorithm of gradient retro-propagation
- Modification of the weights
- Stop criteria

  Error function. Number of iterations, ...

Learning: Gradient descent

- Question:

  Which connexion and how much is responsable of the error*?*

- Principle :

  Computable of the error on a connexion in function of the error on the next layer.

- Two steps :

  1. Evaluation of the error derivate in function of the weight
  2. Use this derivate to compute the modification of the weight

Learning: Gradient descent

1. Evaluation of the error E^j from each connexion : $\frac{\partial E^l}{\partial w_{ij}}$

Idea : compte the error on the connexion Wji in function on the error beyond the neuron j

$$\frac{\partial E^l}{\partial w_{ij}} = \frac{\partial E^l}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}} = \delta_j \, z_i$$

– For the neurons in the output layer:

$$\delta_k = \frac{\partial E^l}{\partial a_k} = f'(a_k) \frac{\partial E^l}{\partial y_k} = f'(a_k) \cdot \left(F_k(\mathbf{x}_l) - y_k \right)$$

– For the neurons in the hidden layer :

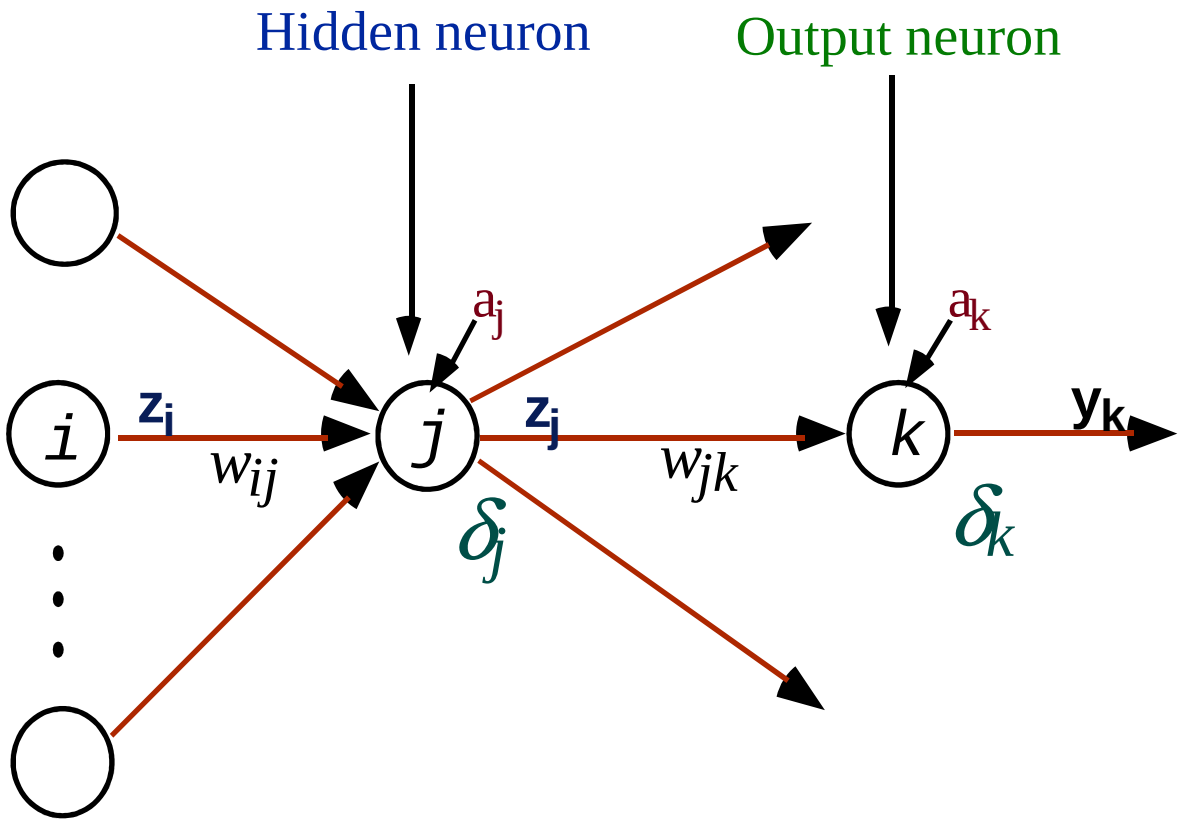
$$\delta_j = \frac{\partial E^l}{\partial a_j} = \sum_k \frac{\partial E^l}{\partial a_k} \frac{\partial a_k}{\partial a_j} = \sum_k \delta_k \frac{\partial a_k}{\partial z_j} \frac{\partial z_j}{\partial a_j} = f'(a_j) \cdot \sum_k w_{jk} \delta_k$$

Learning: Gradient descent

a_i : activation of the neuron i

z_i : output of the neuron i

δ_i : error in the neuron i



Learning: Gradient descent

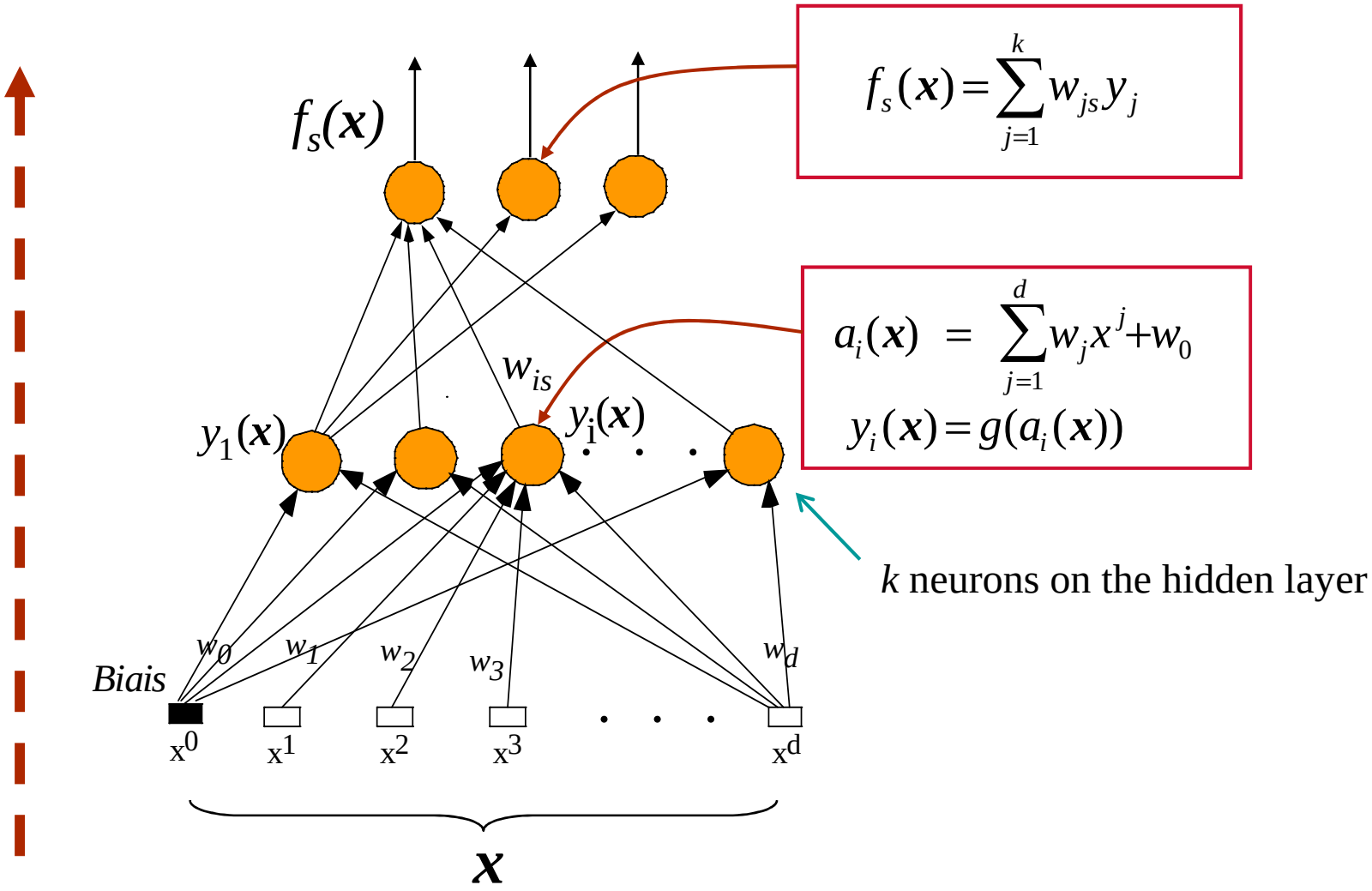
- 2. Modification of the weights
  - Constant step (constant or not): η(t)
  - If on-line learning (after each example)

$$\Delta w_{ji} = \eta(t)\,\delta_j\,a_i$$

  - If off-line learning (after all examples)

$$\Delta w_{ji} = \eta(t)\sum_n \delta_j^n\,a_i^n$$

Summary: compute the output



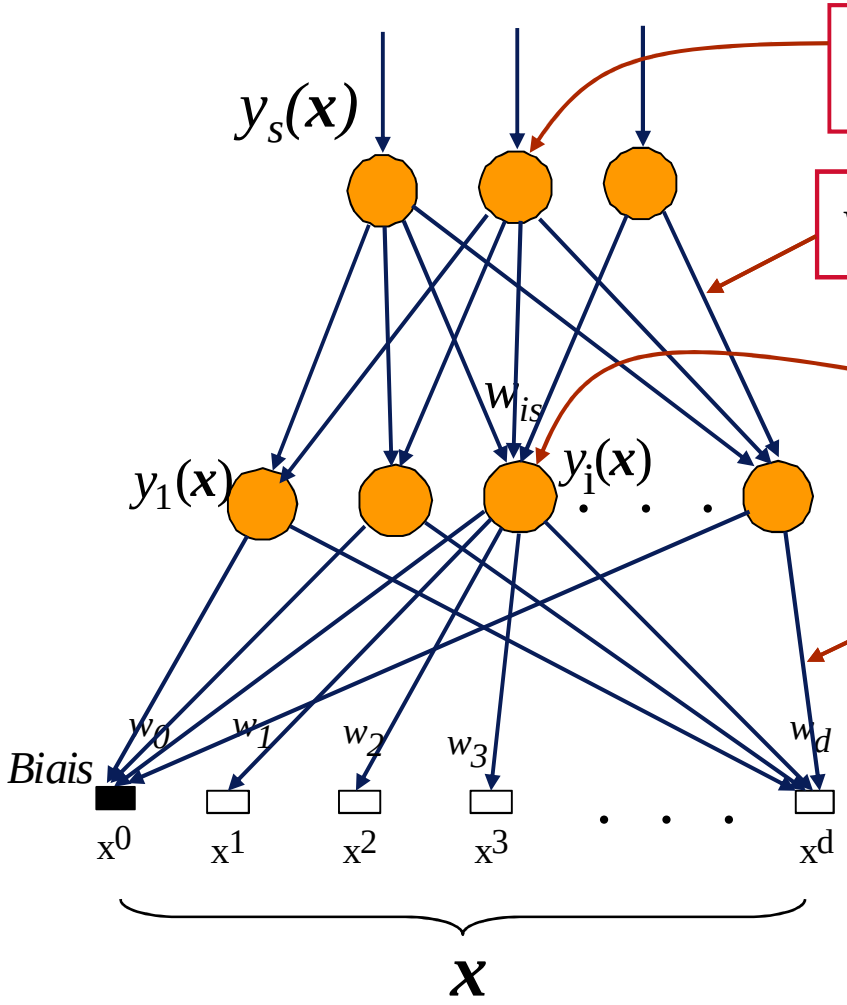
$$f_s(x) = \sum_{j=1}^{k} w_{js} y_j$$

$$a_i(x) = \sum_{j=1}^{d} w_j x^j + w_0$$

$$y_i(x) = g(a_i(x))$$

k neurons on the hidden layer

Summary: update the weights



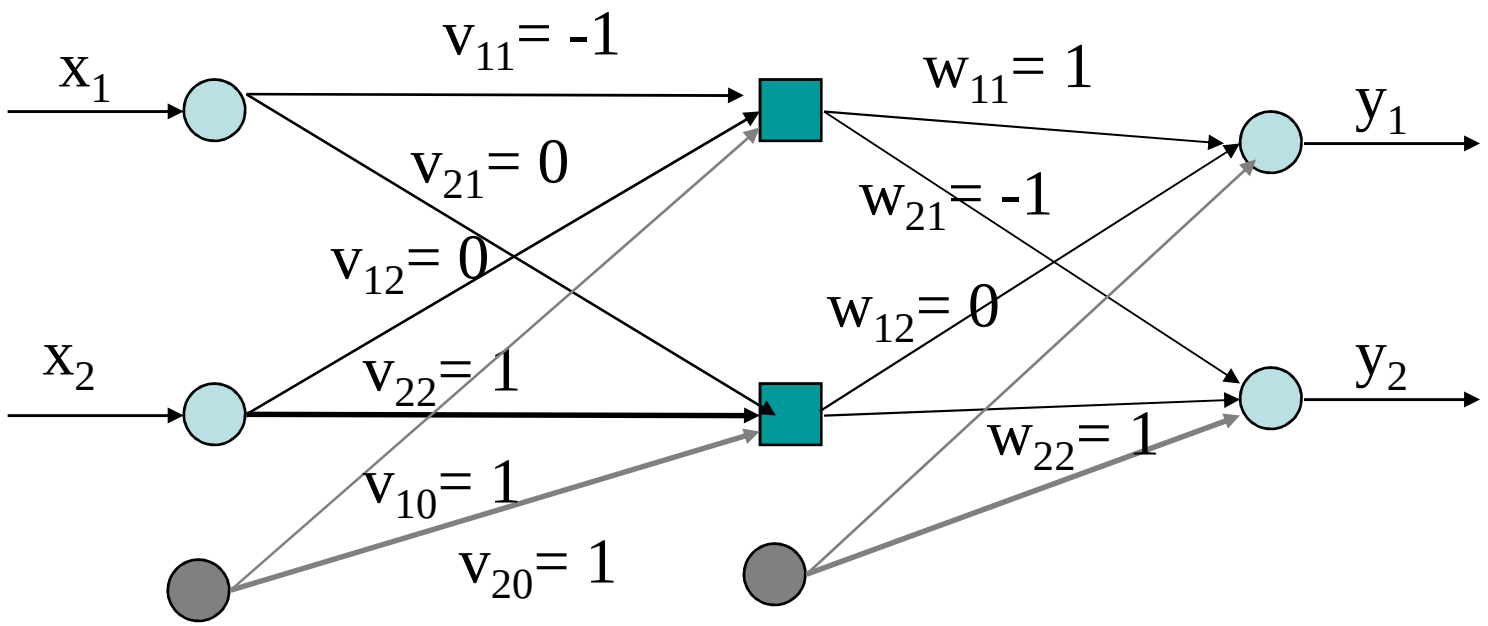
$$\delta_s = g'(a_s)(u_s - y_s)$$

$$w_{is}(t+1) = w_{is}(t) - \eta(t)\delta_s a_i$$

$$\delta_j = g'(a_j)\sum_{\substack{\text{neurons}\\ \text{next}\\ \text{layer}}} w_{js}\delta_s$$

$$w_{ei}(t+1) = w_{ei}(t) - \eta(t)\delta_i a_e$$

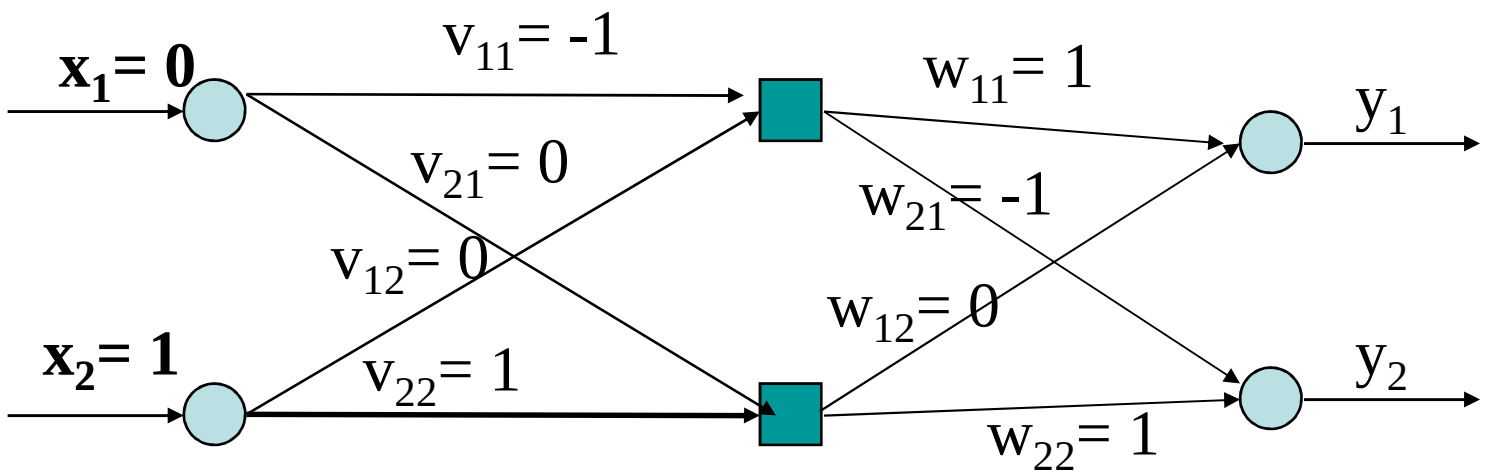
Once weight changes are computed for all units, weights are updated at the same time (bias included as weights here). An example:



Use identity activation function (ie f(a) = a)

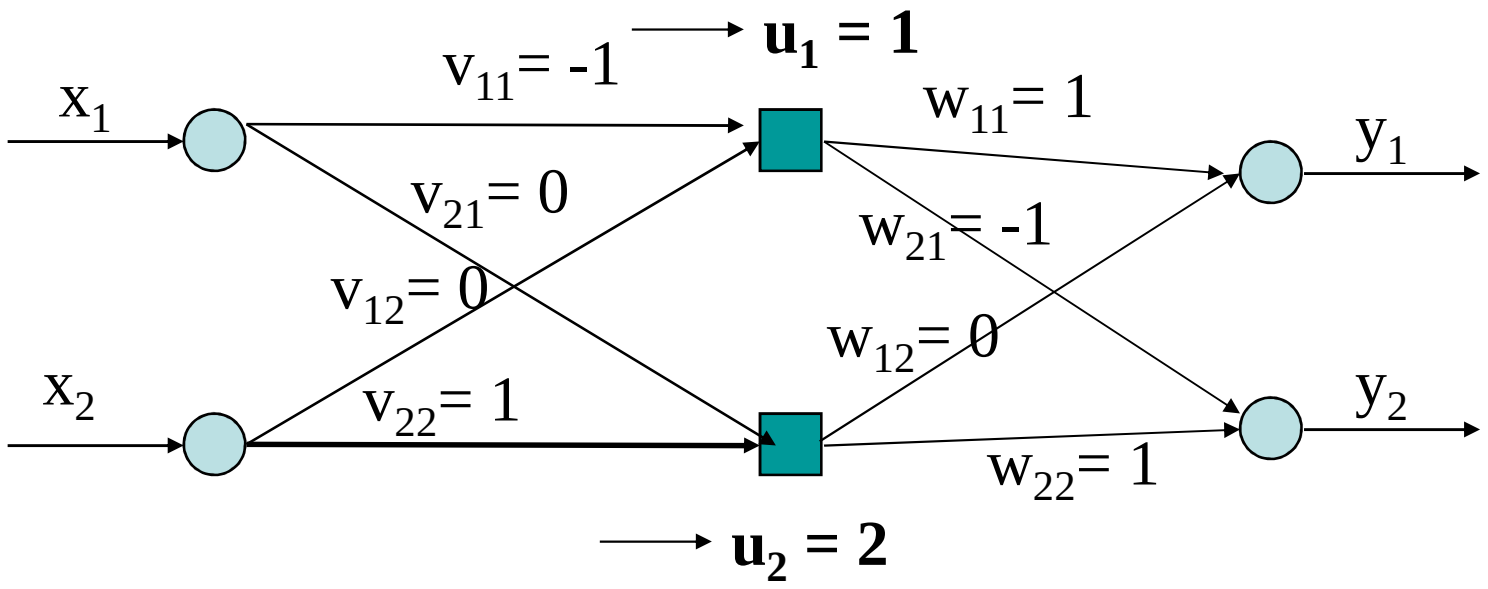
All biases set to 1. Will not draw them for clarity.

Learning rate $\eta = 0.1$



Have input [0 1] with target [1 0].

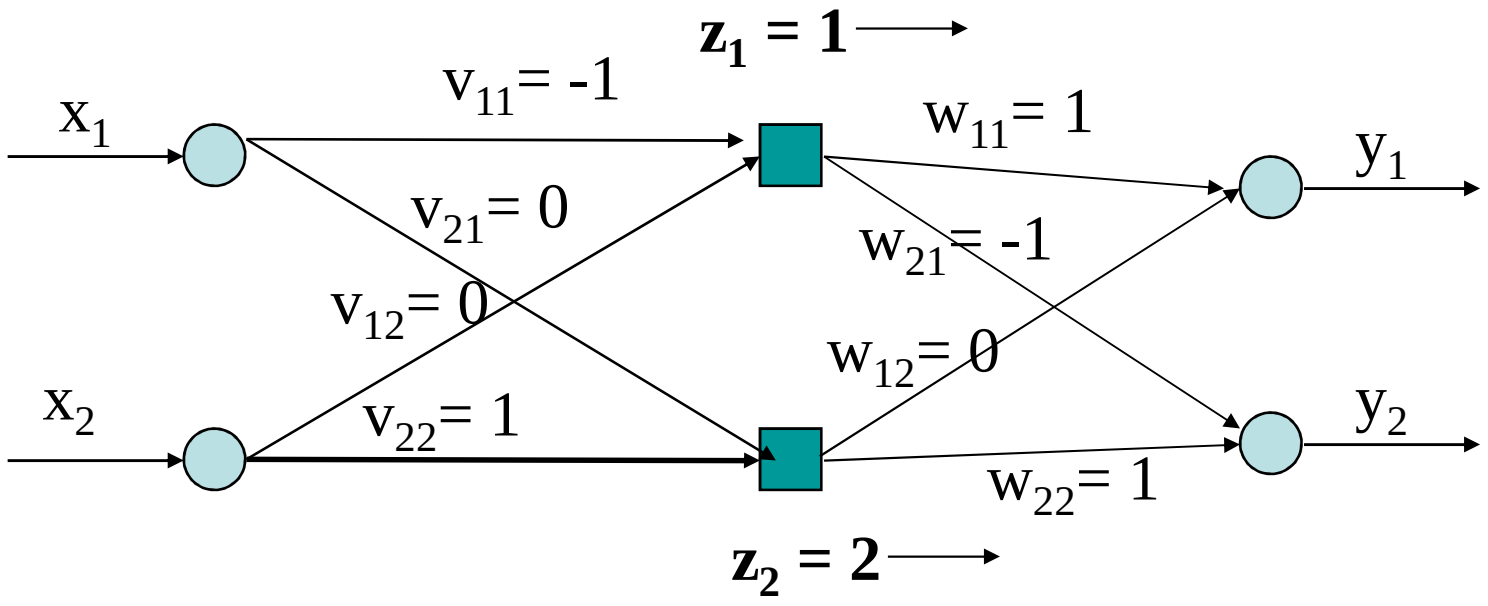
Forward pass. Calculate 1st layer activations:



$u_1 = -1x0 + 0x1 + 1 = 1$

$u_2 = 0x0 + 1x1 + 1 = 2$

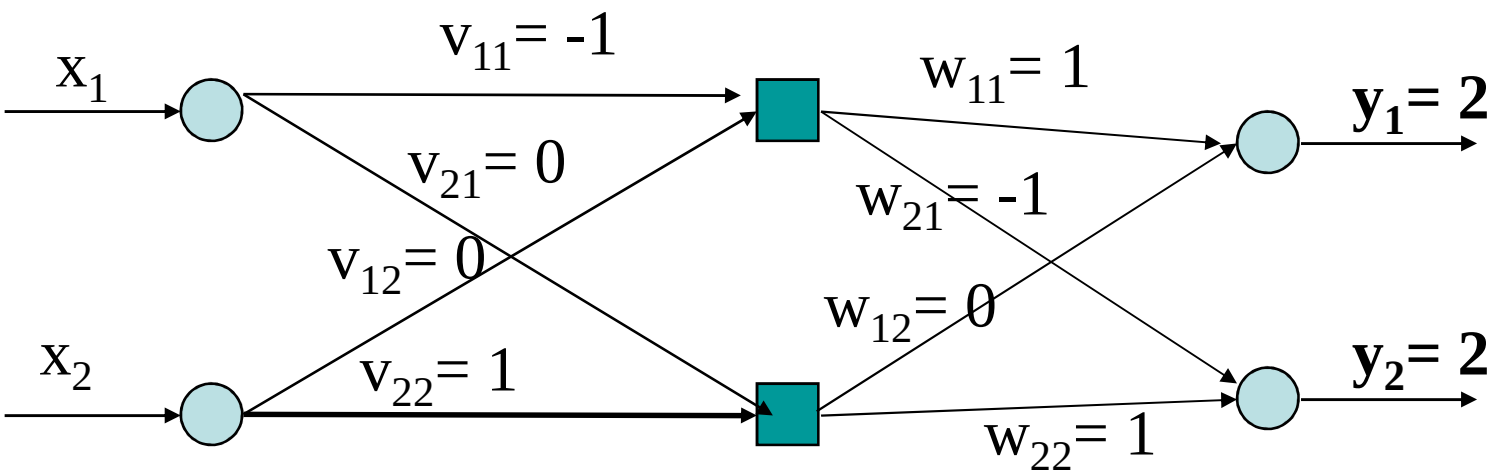
Calculate first layer outputs by passing activations thru activation functions



$z_1 = g(u_1) = 1$

$z_2 = g(u_2) = 2$

Calculate 2nd layer outputs (weighted sum thru activation functions):

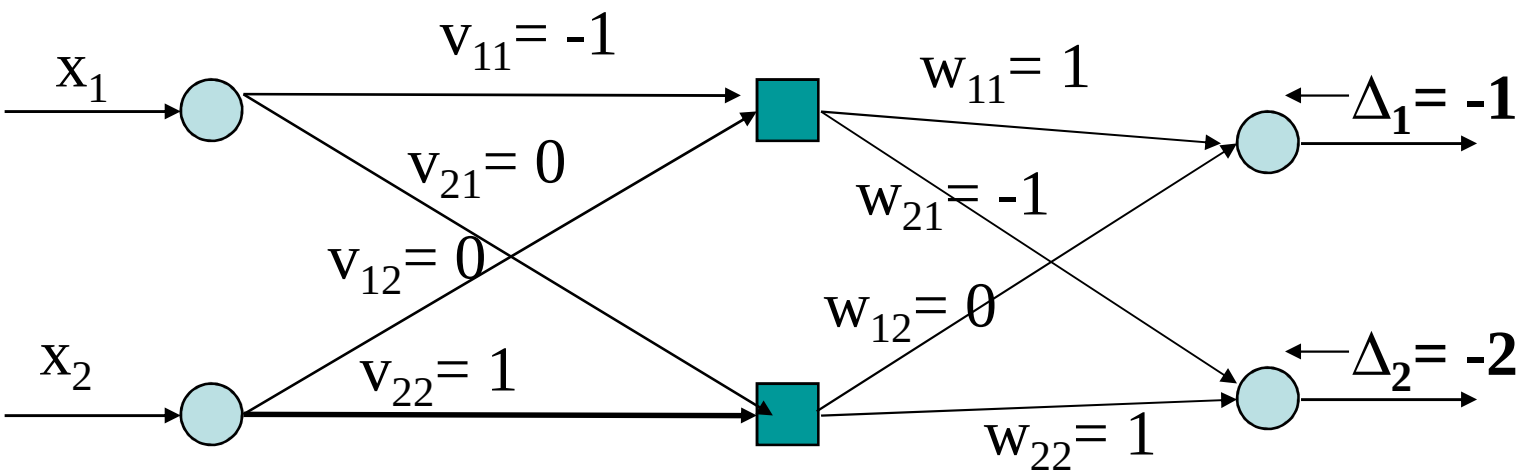


$y_1 = a_1 = 1x1 + 0x2 + 1 = 2$

$y_2 = a_2 = -1x1 + 1x2 + 1 = 2$

Backward pass:

$$w_{ij}(t+1) - w_{ij}(t) = \eta \Delta_i(t) z_j(t)$$
$$= \eta (d_i(t) - y_i(t)) g'(a_i(t)) z_j(t)$$



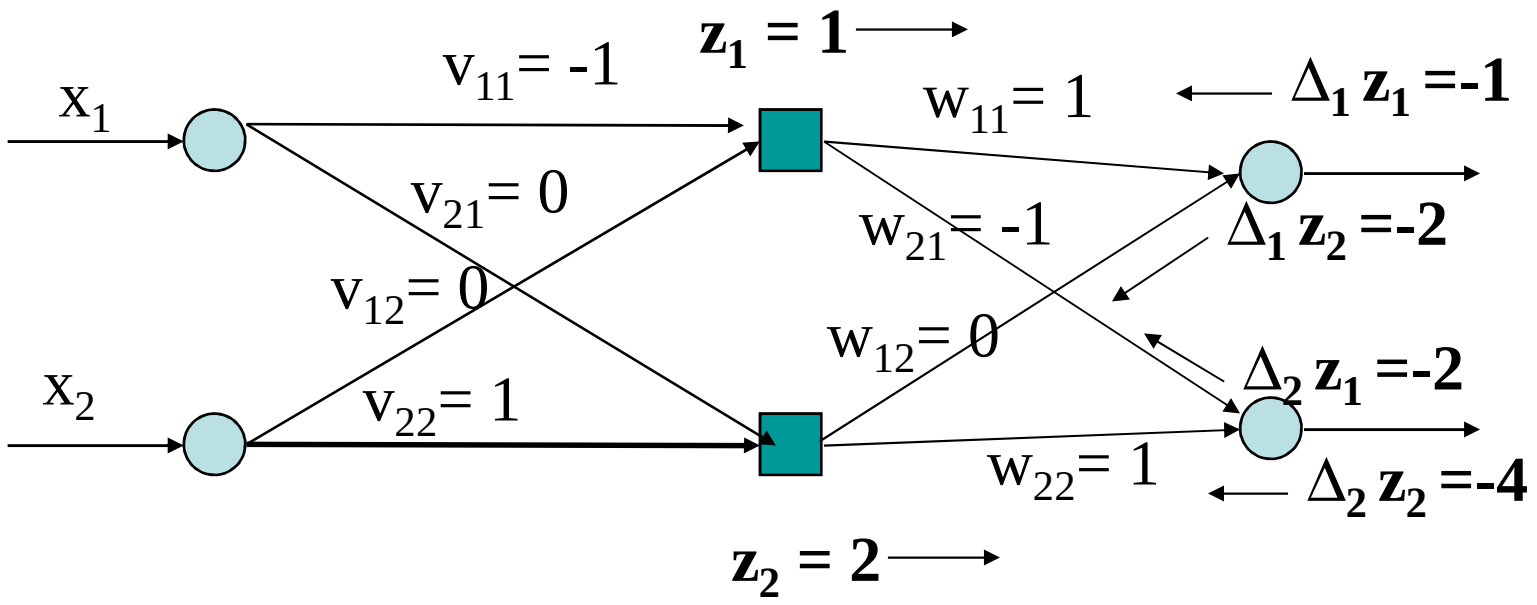
Target =[1, 0] so $d_1 = 1$ and $d_2 = 0$

So:

$\Delta_1 = (d_1 - y_1) = 1 - 2 = -1$

$\Delta_2 \ (d_2 - y_2) = 0 - 2 = -2$

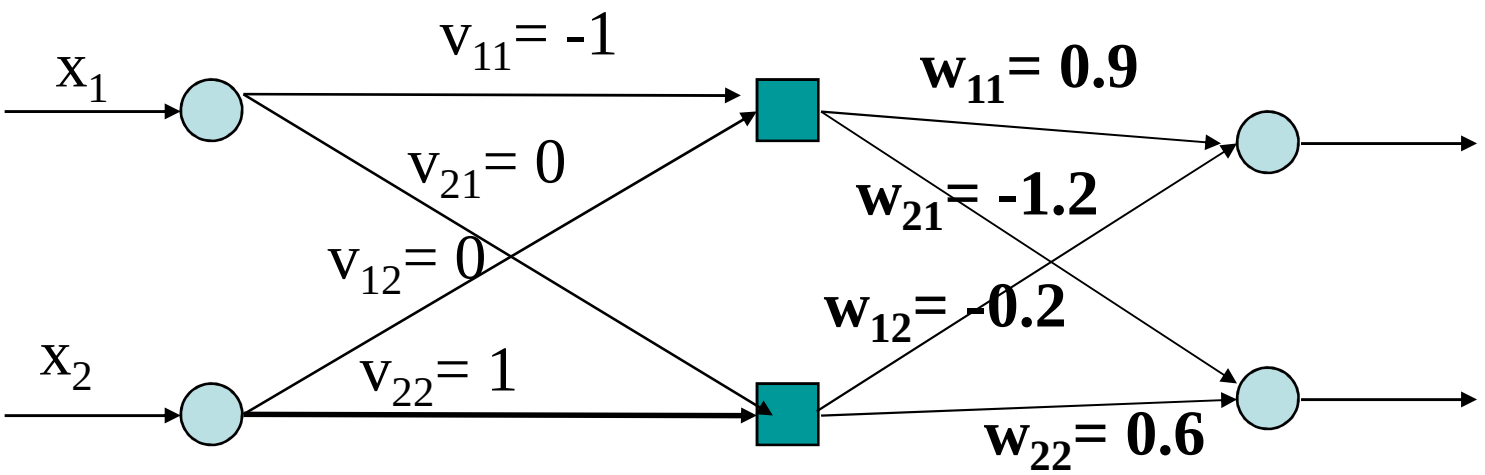
Calculate weight changes for 1st layer (cf perceptron learning):



$$w_{ij}(t+1) - w_{ij}(t) = \eta \Delta_i(t) z_j(t)$$

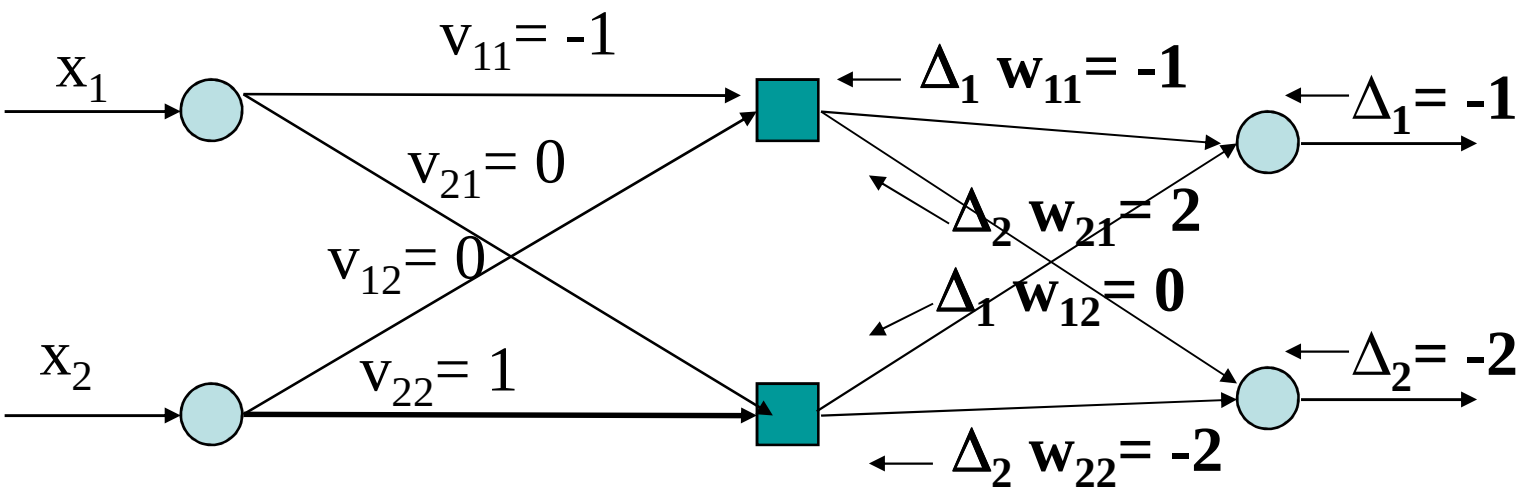
Weight changes will be:

$$w_{ij}(t+1) - w_{ij}(t) = \eta \Delta_i(t) z_j(t)$$



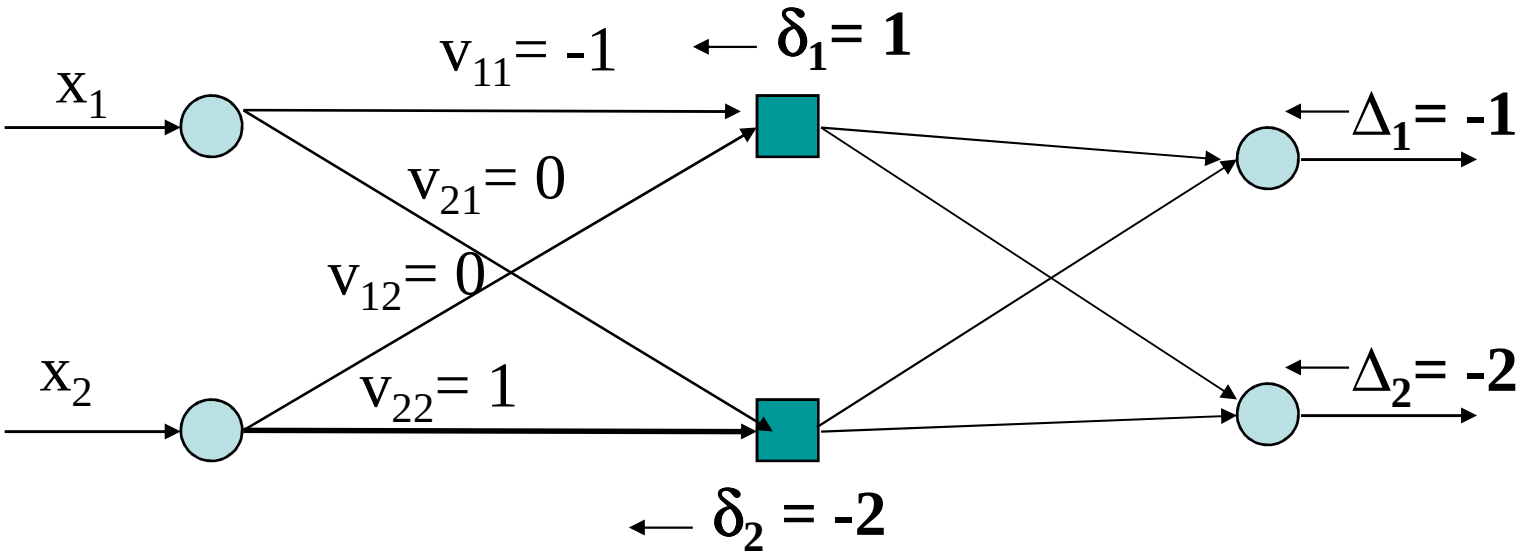
But first must calculate δ's:

$$\delta_i(t) = g'(u_i(t)) \sum_k \Delta_k(t) w_{ki}$$



Δ's propagate back:

$$\delta_i(t) = g'(u_i(t)) \sum_k \Delta_k(t) w_{ki}$$

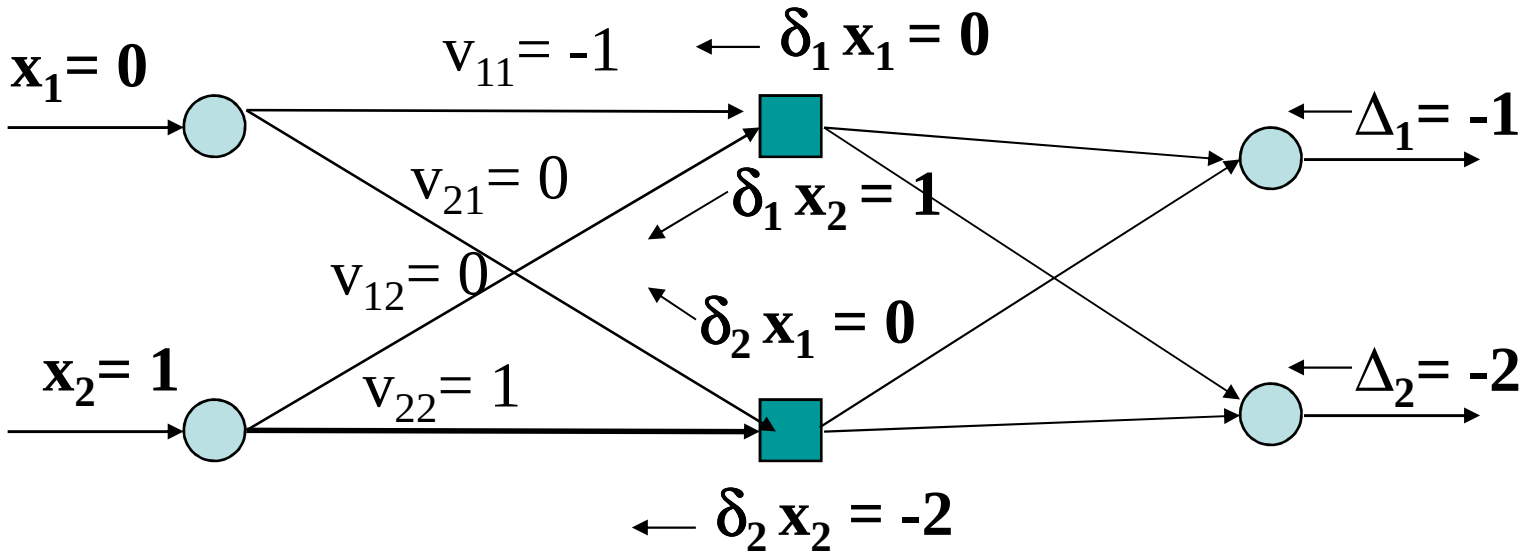


$\delta_1 = -1 + 2 = 1$

$\delta_2 = 0 - 2 = -2$

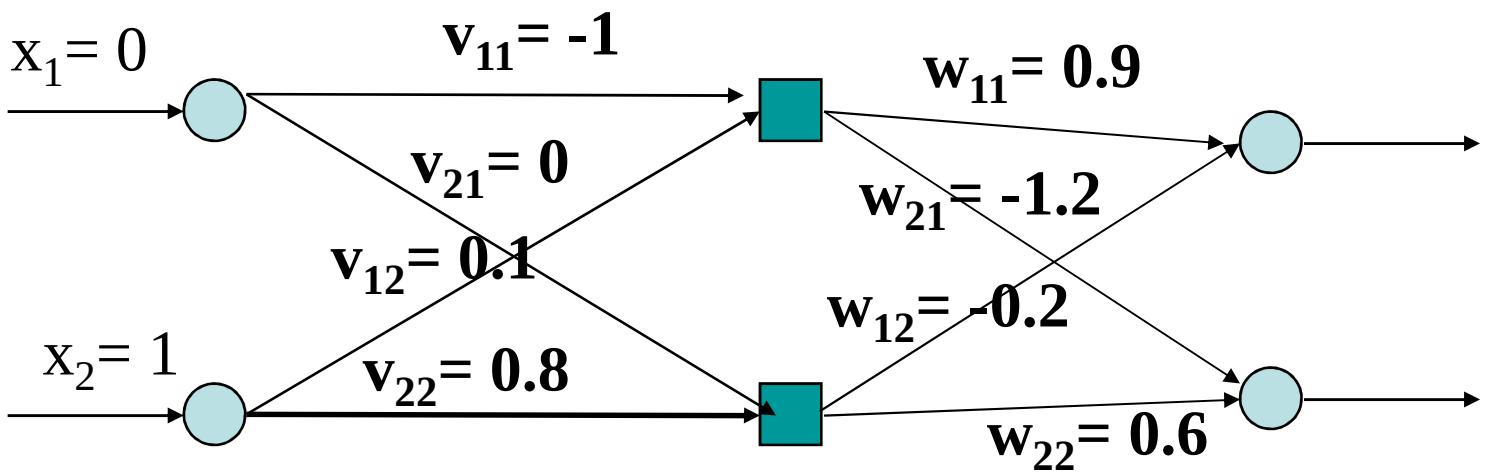
And are multiplied by inputs:

$$v_{ij}(t+1) - v_{ij}(t) = \eta \delta_i(t) x_j(t)$$



Finally change weights:

$$v_{ij}(t+1) - v_{ij}(t) = \eta \delta_i(t) x_j(t)$$

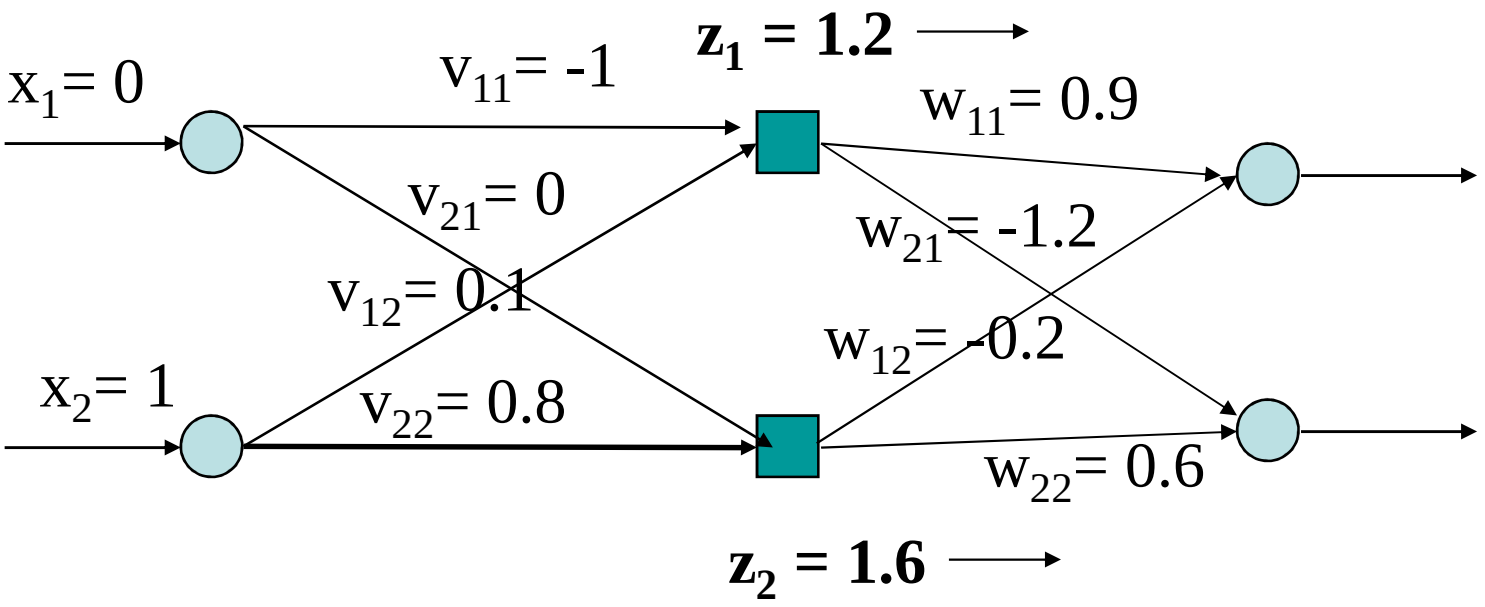


Note that the weights multiplied by the zero input are unchanged as they do not contribute to the error

We have also changed biases (not shown)

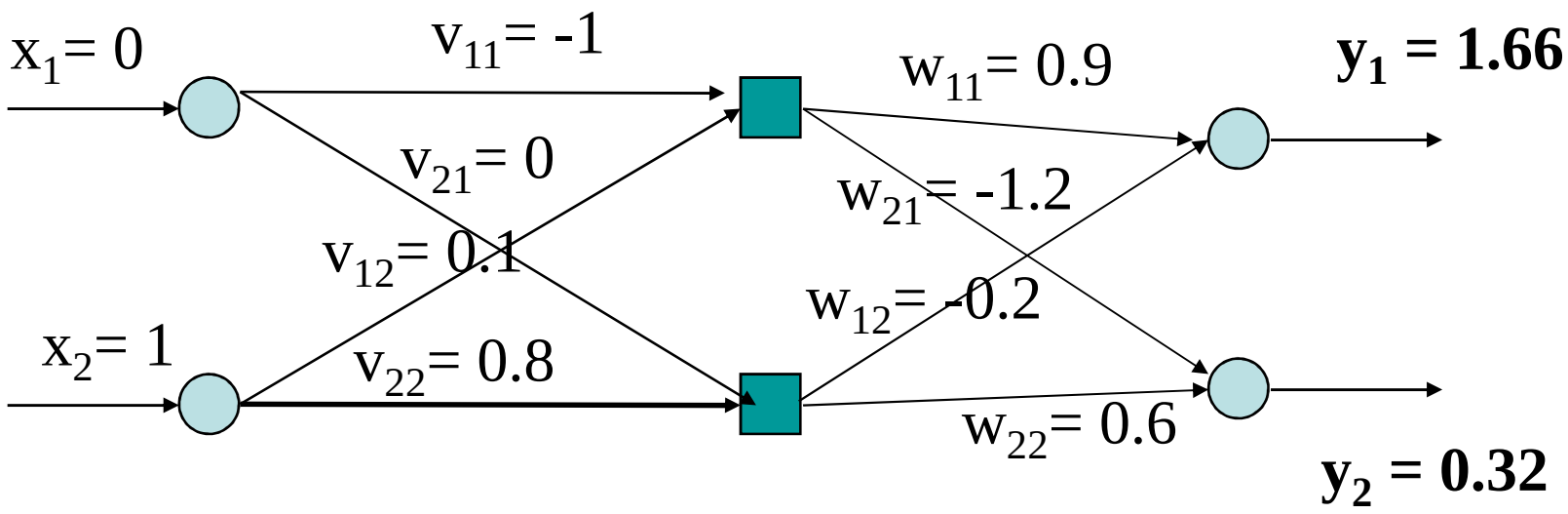
Now go forward again (would normally use a new input vector):

$$v_{ij}(t+1) - v_{ij}(t) = \eta \delta_i(t) x_j(t)$$



Now go forward again (would normally use a new input vector):

$$v_{ij}(t+1) - v_{ij}(t) = \eta \delta_i(t) x_j(t)$$



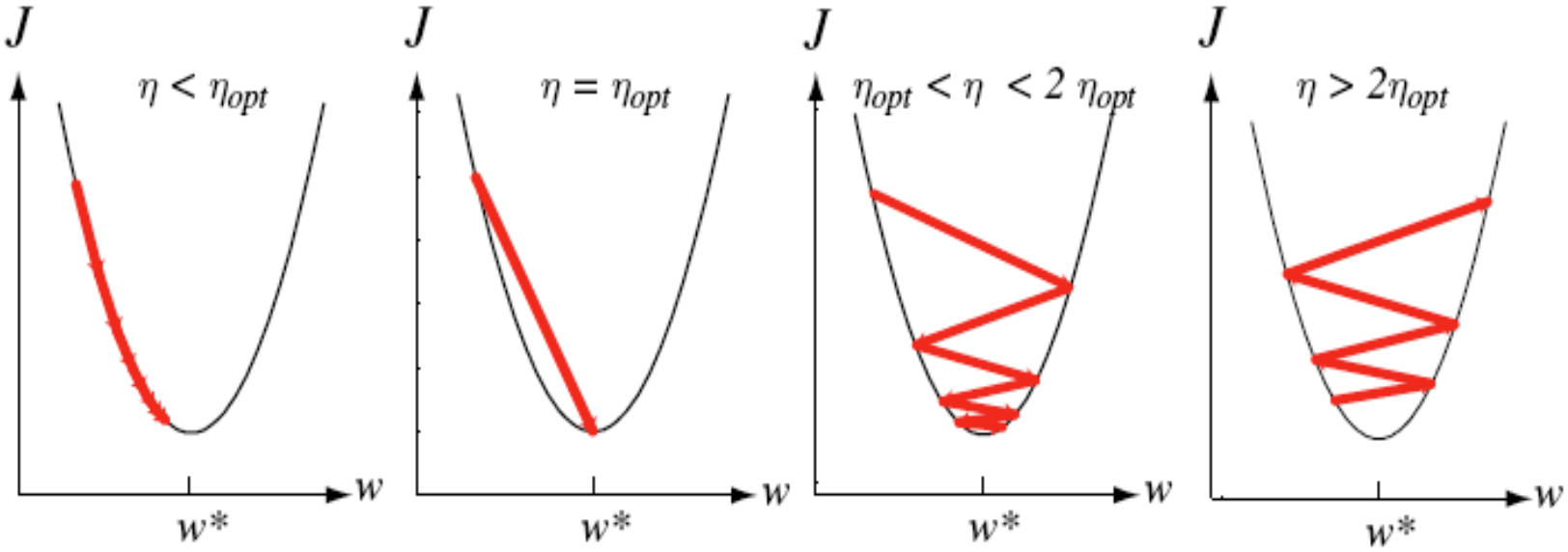
Outputs now closer to target value [1, 0]

Learning: Gradient descent

- Learning performance
  - Complexity: O(*w*) for each iteration, *w* = nb of connexions
  - Several hundreds of iteration are needed
  - This procedure shoud be iterated several times (tens) with different weight initialization

Convergence

- Setting the learning step : η

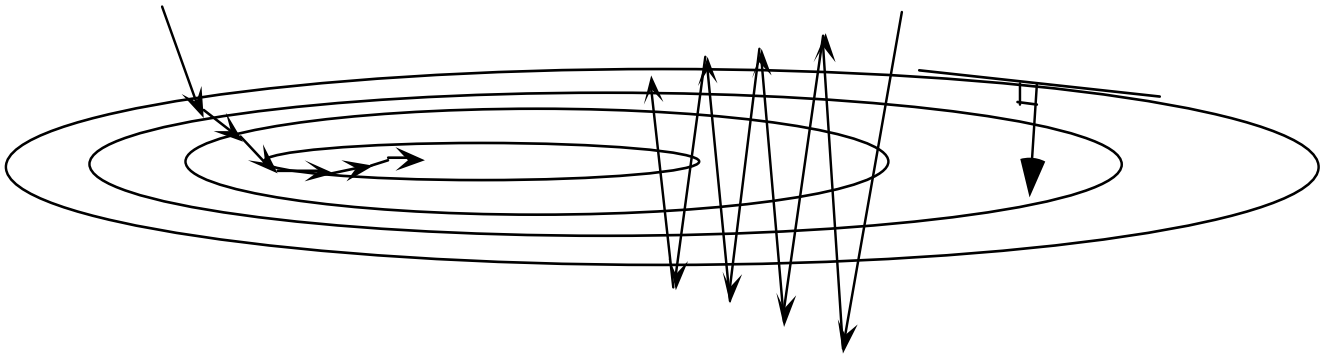


- Problem of the local minimums

Problems of convergence

Local minimum.

- Add an inertia term
- Identify the variable depency
- Add noise to the trainning examples
- Use the non constant learning step) .
- Use the second derivate (Hessien).



Problematic of feature selection

Intuition: The more features, the more information, the better classifier.

It is flase !!

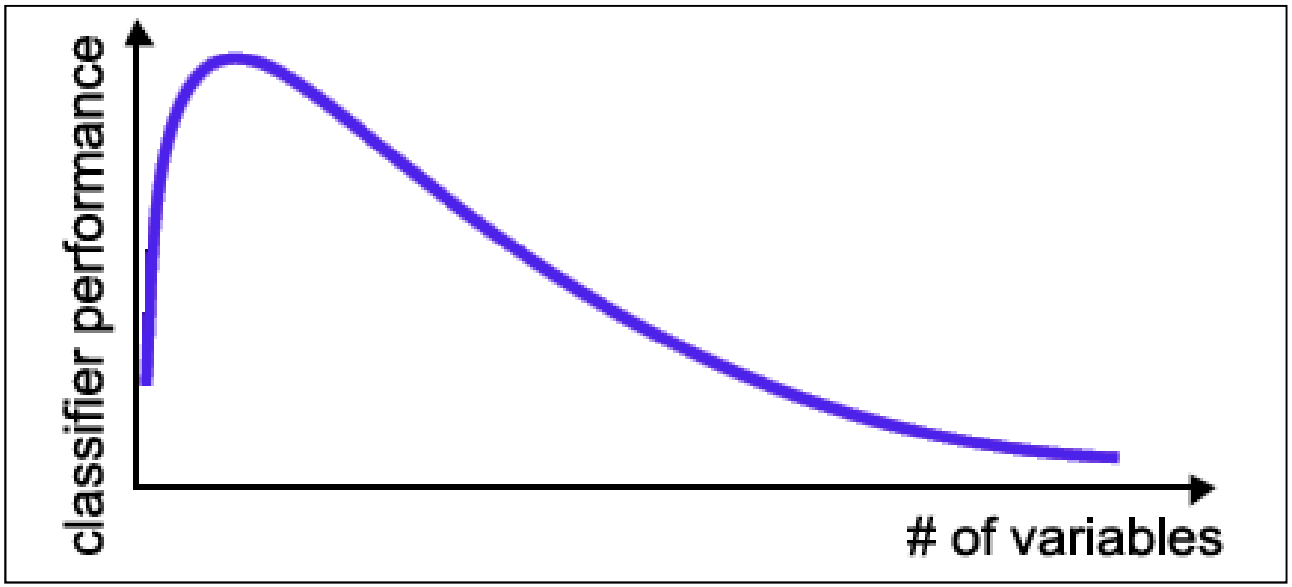
Why ?

- A lot of features may be useless (noise)
- Some features are redundant
- A lot of features implies a large computing time
- The results and classifiers are not easily interpretable if they imply a large number of features.

The curse of dimension

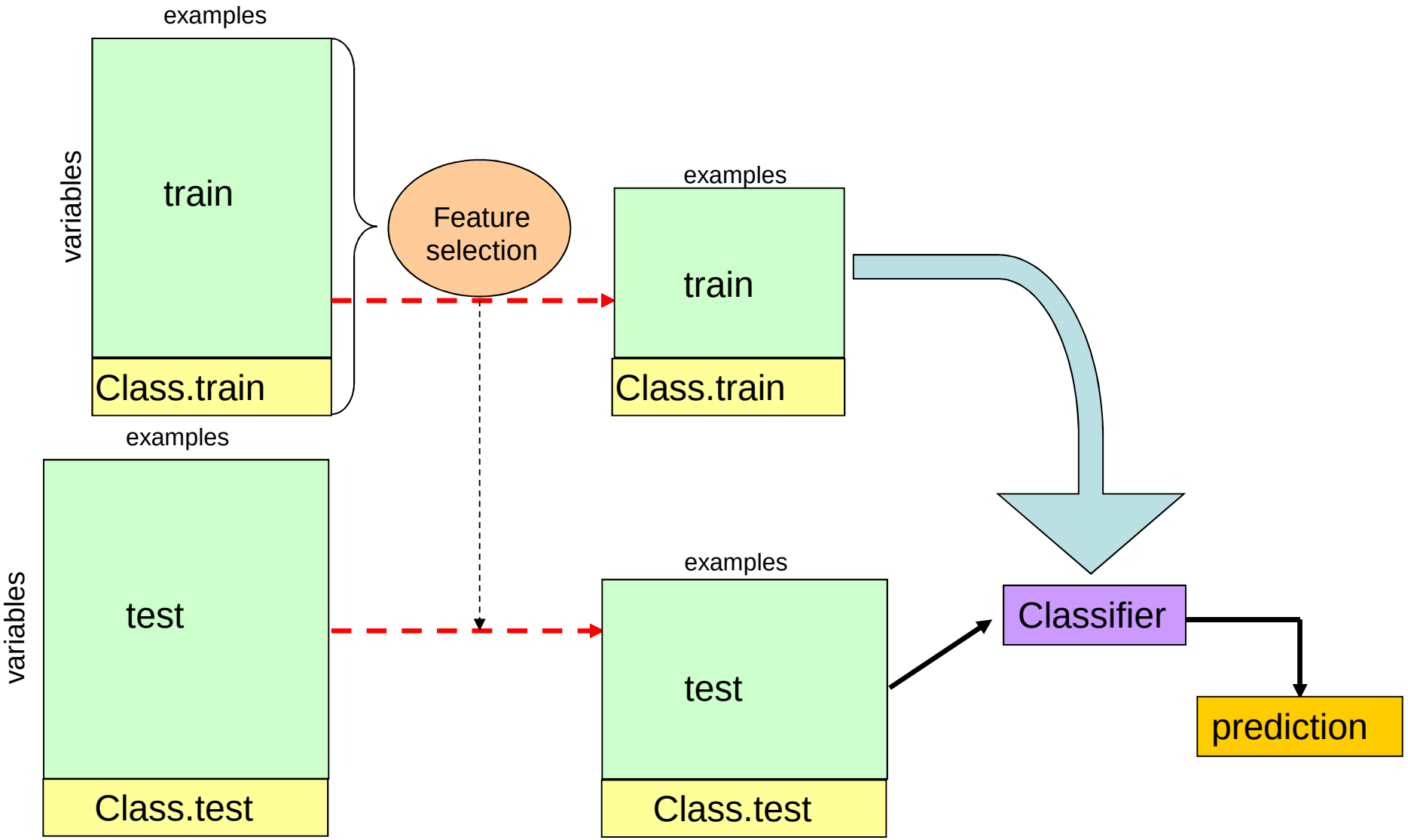
The needed number of examples increases exponentially with the number of features

- There is an optimal number of features d*
- For d>d* the performance of the classifier decreases with the number of features



Generally a feature selection step is necessary !

Where is the feature selection step ?



Feature selection

Objective:

Selection of the most relevant feature for the classification task

The ones that will improve the classifier performance

Problem:

We cannot test all possible feature subsets

For D features there are 2^D possible feature subsets

($<10^6$ subsets for 20 features)

We have to rely on heuristics