Neuroengineering (I)
2. Feed-forward neural networks

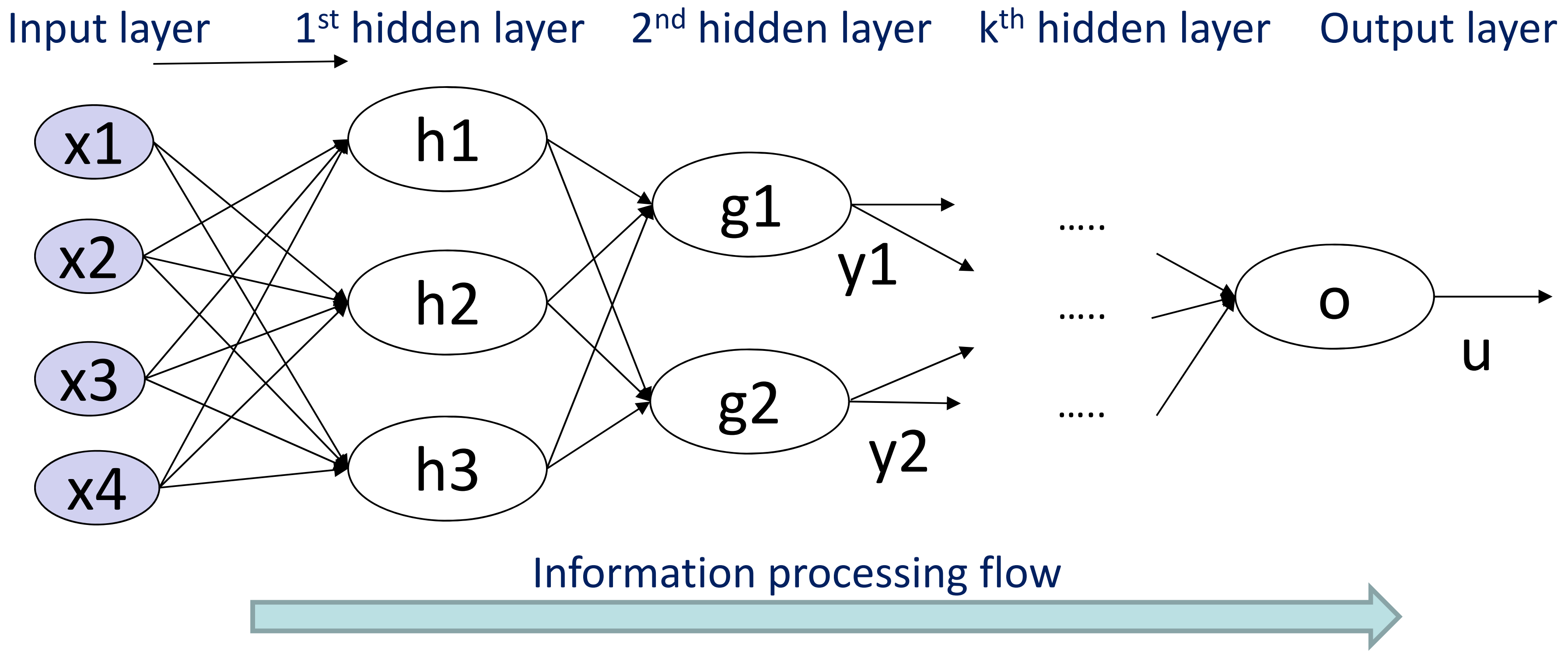
- **Scuola di Ingegneria Industriale e dell'Informazione** – Politecnico di Milano
- Prof. Pietro Cerveri

22-09

Two ways of learning:
1) perceptron learning rule
2) delta rule

FFNN: Sort of multilayer where information goes from one layer to the other without intralayer connections and (interlayer?). The input is not an neuron, but it can be considered a virtual neuron layer, indeed we start numebering the layer with the input layer. It's the first layer and so on. In FFNN it's easy to discriminate the input, the hidden layer "in the middle" (from the 2nd layer and so on) then output layer. We have seen that the specialization of the network is based on different strategies: conserving the connections (thus topology, aka the arrows that connects neurons), but also the activation functions: many different. Defining the activation functions you act on the topology. **In this case learnin means compute the weight and threshold**.

FFNN – Multi-layer network



Basic neuron model: MCP

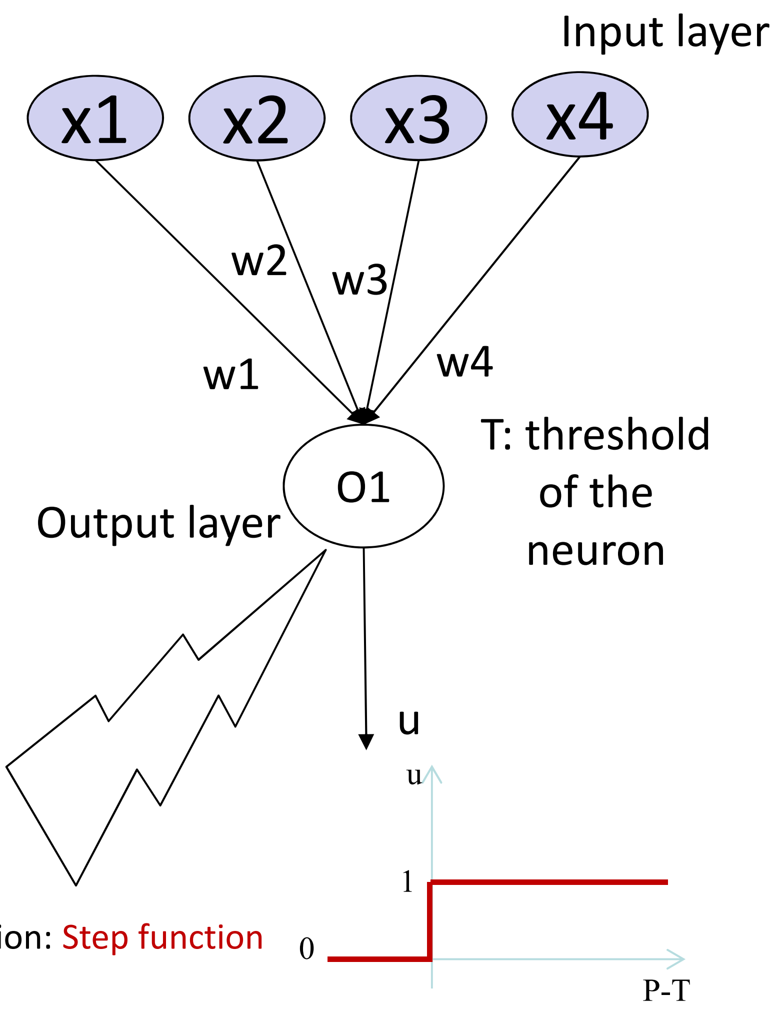
Basic two-layer network: PERCEPTRON

Basic NN with just one neuron with an activation function described as a step, just one output line, and one or many inputs. We can batch the input in a vector and create a pattern for the input. Pattern: multidimensional variable that has the size of the input layer. The activation function is a step, therefore you can just outputting two different values: 0 and 1. Clearly it's assimilable to a classifier: 1 it's a class, 0 it's another class. (binary classification). This can be seen as a decision model in general. This architecture can describe logic ports. Decision model is a system that is able to perform analysis starting from a set, or a combination of conditions (the input), and usually each condition is a binary thing (it can happen or not). If I'm enable to weight these conditions, and integrate these with AP (weighted sommation), I can realize the decision, acting by means of the activation function. I could build more complex decision models by allowing in the network more layers of integrations and analysis of the conditions, meaning that each layer is analyizing the result of the previous results. Layer by layer the conditions are synthezided.

Perceptron

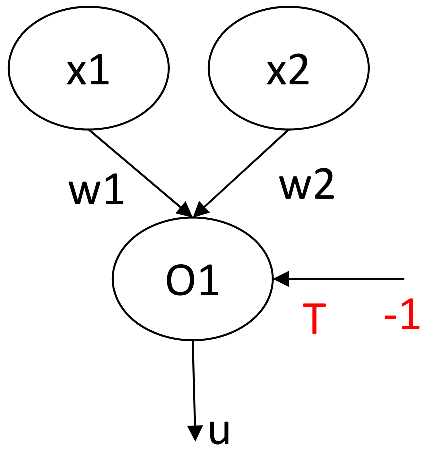
- 2-layer neural network
- Input layer (N>=1 signals) – Output layer (M>=1 signals)
- The set [x1 x2 x3 ,…] of binary values is called the input pattern (stimulus)
- It can be used to:
  - classify categories of patterns

- The simplest Perceptron has one single unit in the output layer
- With binary activation (step) function
  - 1 if the stimulus in input belongs to the desired category
  - 0 if the stimulus in input does not belong to the desired category
- ~~Decision model~~



The threshold can be considered a weight of the neuron, and to keep valid the computation of the AP, I consider a virtual input that is constant at -1. This way allows me to build an overall AP of the neuron. "Additional input" of -1. From now if I the AP and it overcomes 0 I get an activation function with a positive output. This corresponds to substitute P-T with P. So I include the effect on the threshold whenever Im thinking about the action potential. This will allows us to apply the same mathematical approach to learn the weight and the threshold.

Action potential and neural threshold



Notice that the threshold can be considered as an additional weight of the neuron with a virtual constant input equal to -1

$$P = \sum_{j=1}^{2} w_j x_j - T = w_1 x_1 + w_2 x_2 + T(-1)$$

$$P = \sum_{j=1}^{3} w_j x_j = w_1 x_1 + w_2 x_2 + w_3 x_3 \quad with \quad w_3 = \mathrm{T} \; and \; x_3 = -1$$

From now on we call *P-T* action potential and we name it with *P*

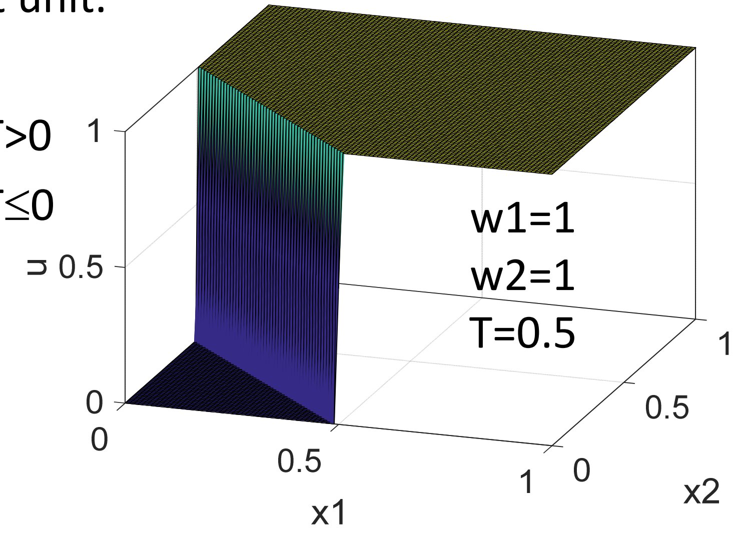
Appunti quaderno

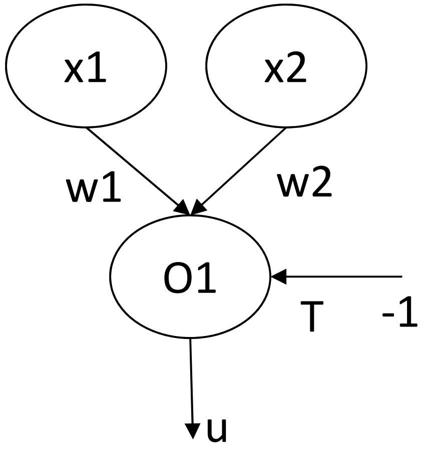
Perceptron with step function in action

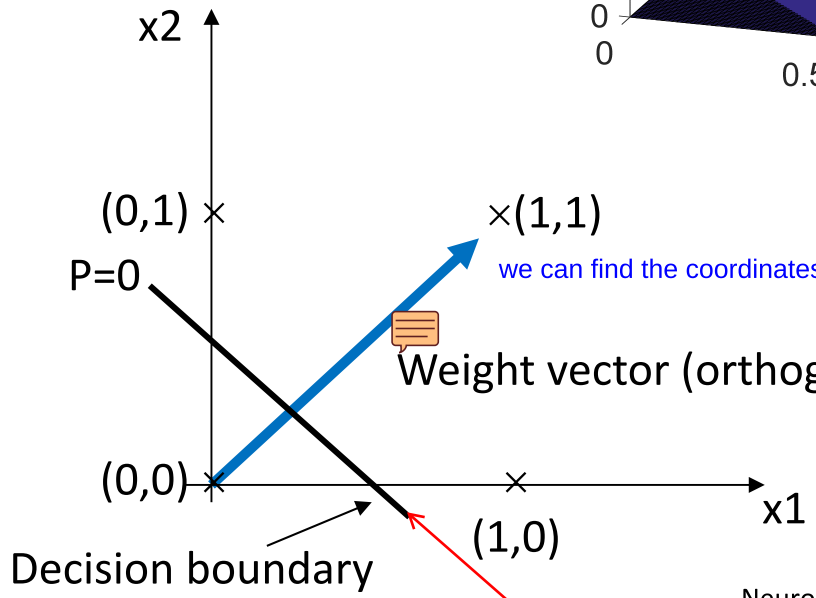
Let assume the perceptron be composed by 2 input units and 1 output unit.

u=1 if w1*x1+w2*x2 – T>0

u=0 if w1*x1+w2*x2 – T≤0







we can find the coordinates where the equations for u=1 holds (or the opposite)

Weight vector (orthogonal to the decision boundary)

output of the neuron equal to 0

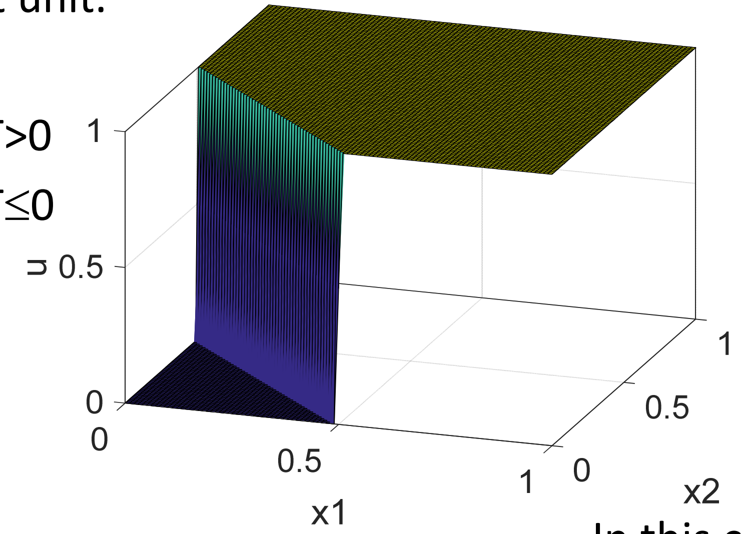
This let the perceptron behaviour like an OR port. If Im dealing with real number I can imagine it's still an or port but with "extended". I can obtain an AND if I move on the right the boundary, setting different values of w1w2.

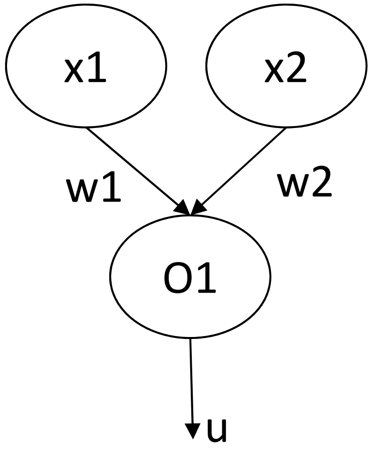
Perceptron with step function in action

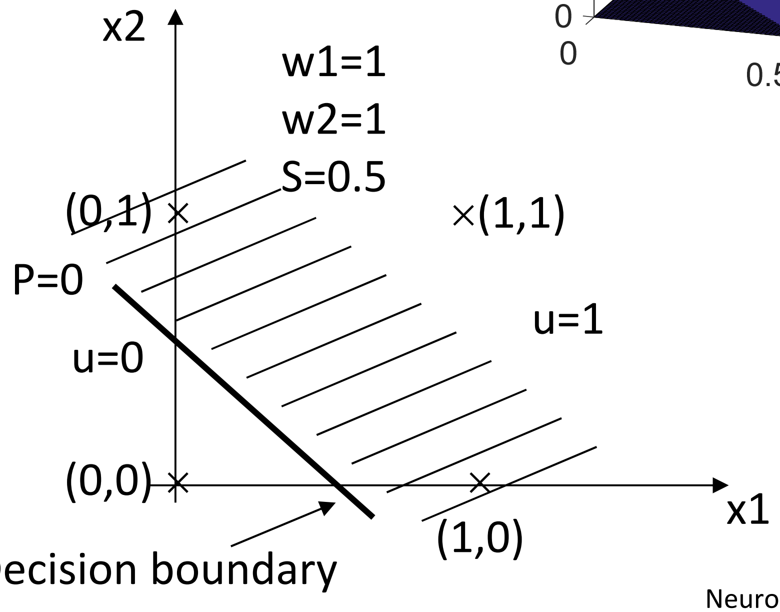
Let assume the perceptron be composed by 2 input units and 1 output unit.

u=1 if w1*x1+w2*x2 – T>0

u=0 if w1*x1+w2*x2 – T≤0







In this case the perceptron is classifying (0,1) , (1,1) and (1,0) as positive whereas the input pattern (0,0) will be classified as negative

OR port

If S=1.5 then it is an AND port

Actually w1 and w2 are the same but Im changing the threshold, I'm translating. I can have an "all pass" if I move the boundary completely to the left, the opposite if I move it to the right (over the points).

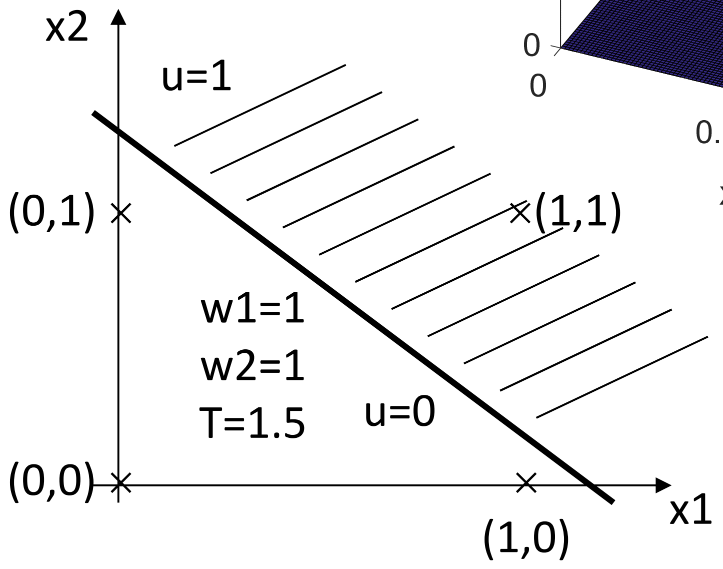
All this make sense if we are thinking about logic inputs.

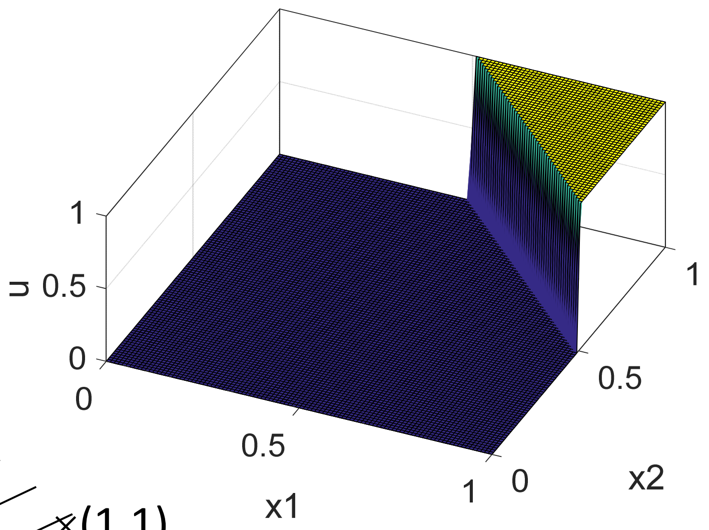
Perceptron with step function in action

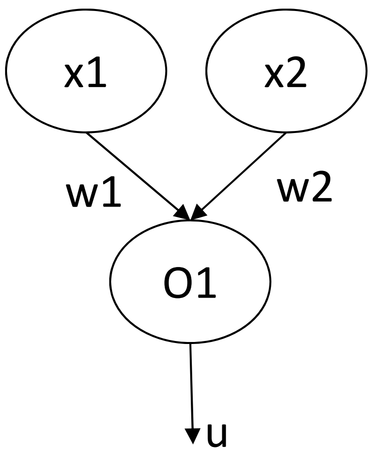
Let assume the perceptron be composed by 2 input units and 1 output unit.

u=1 if 1*x1+1*x2 -1.5 > 0

u=0 if 1*x1+1*x2 -1.5 $\leq$ 0







AND port

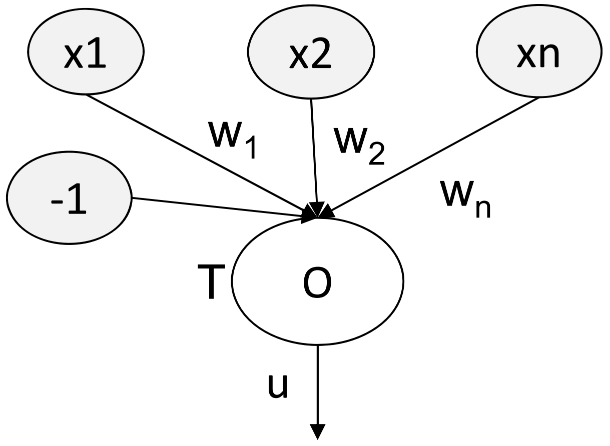
t sposta a destra ed a sinistra, vedi geogebra

The best way to learn is to exploit observation (brain). It's the same here, you had for the same target many different experience. So the idea is that I'm looking at the output of the network to see if it happens what I'm expecting.For example for an OR port, if I input a 0,0 I expect an output of 0. The more the example, in principle more robust should be the learning. There are actually a lot of position of decision boundaries that allow that. From a mathematical point of view we have 3 degrees of freedom, so I can expect 3 equation. I could input the same value.. like 0,0 then 0,0 and so on, but it's clealry uneffective.

Training the perceptron

[appunti quaderno]

- ANN topology is given
  - Structure
  - Activation functions
- Training process: estimation of the neural weights
- **Supervision**: expected outputs are known for each input pattern
- Training dataset
  - Set of input examples (training patterns)
  - Set of corresponding reference output patterns t



$$\mathbf{w} = \left[w_1 \; w_2 \; \ldots.. \; w_n \; T \right]$$

is the set of weights to be estimated

Principle for learning: exploit the error signal e = (t-u) wrt a specified expected output t (supervision)

We should have this update of weights ONLY if a classification error occurs.

Perceptron learning rule Rosenblatt (1962)

- Activation: a step function
- *R* training patterns
- Incremental procedure: $\mathbf{w}_{\text{start}}$ (initial weight vector)
- Any pattern contributes to a direct update of the weights
- On-line updating implies that each pattern error contributes sequentially to the weight updating (selection can be random)

$$\mathbf{w}^{(new)} = \mathbf{w}^{(old)} + \Delta\mathbf{w} \quad \text{with} \quad \Delta\mathbf{w} = \eta\left(t^{(k)} - u^{(k)}\right)\underset{\uparrow}{\mathbf{x}^{(k)}}$$

η is the positive scalar learning rate ≤ 1

input vector (pattern)-k

The learning rule corrects the weight vector if and only if a misclassification occurs

Perceptron learning rule

If e = +1 then $\mathbf{w}^{(new)} = \mathbf{w}^{(old)} + \mathbf{x}^{(k)}$

If e = -1 then $\mathbf{w}^{(new)} = \mathbf{w}^{(old)} - \mathbf{x}^{(k)}$

If e = 0 then $\mathbf{w}^{(new)} = \mathbf{w}^{(old)}$

$$\mathbf{w}^{(new)} = \mathbf{w}^{(old)} + \left(t^{(k)} - u^{(k)}\right)\mathbf{x}^{(k)}$$

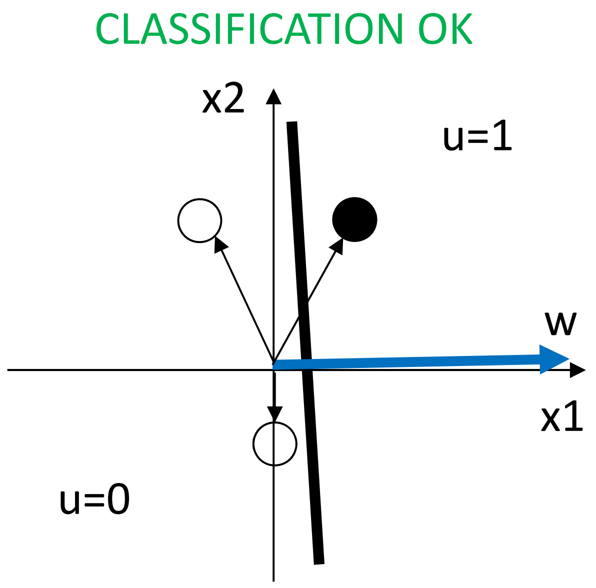
The learning rule correct the weight vector if and only if a misclassification occurs

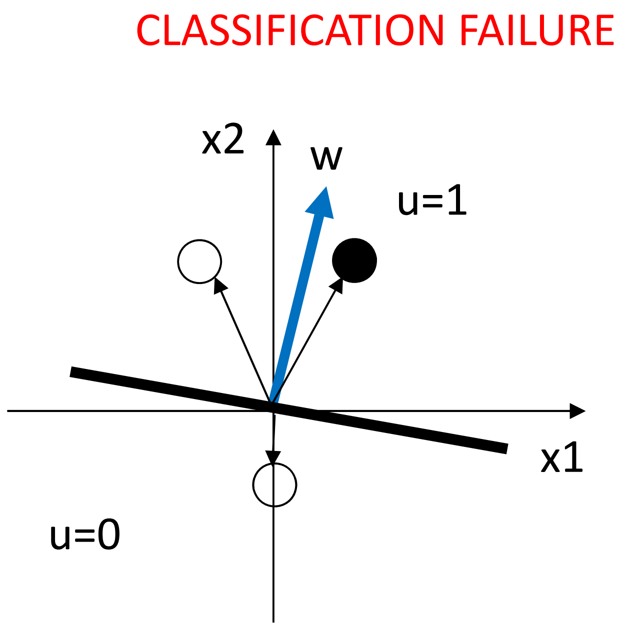
When you have a positive error, like +1 in the example in the quaderno, we correct with that formulation. Xk is the KEITH pattern in the traning dataset. In the case of perceptron with binary output the error can just be 0, -1 or +1.

$$\mathbf{z}^{T(k)}\mathbf{w}^{(k)} \le 0$$

$$\mathbf{z}^{(k)} = \begin{cases} +\mathbf{x}^{(k)} & \text{if } t^{(k)} = +1 \\ -\mathbf{x}^{(k)} & \text{if } t^{(k)} = -1 \end{cases}$$

Geometric validation

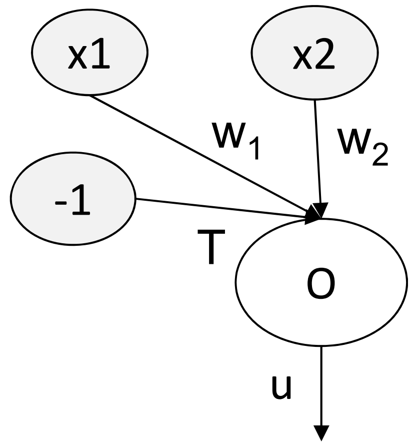




Perceptron rule can be thought of as a way to orient the decision boundary in such a way that the scalar product of weight vector with all the patterns into the first class is positive whereas it is negative with all the patterns into the second class.

Affinchè questo sia vero, il vettore non dovrebbe partire dal piano? altrimenti ci sono casi in cui il prodotto potrebbe essere >0.. forse?

Perceptron learning rule: example



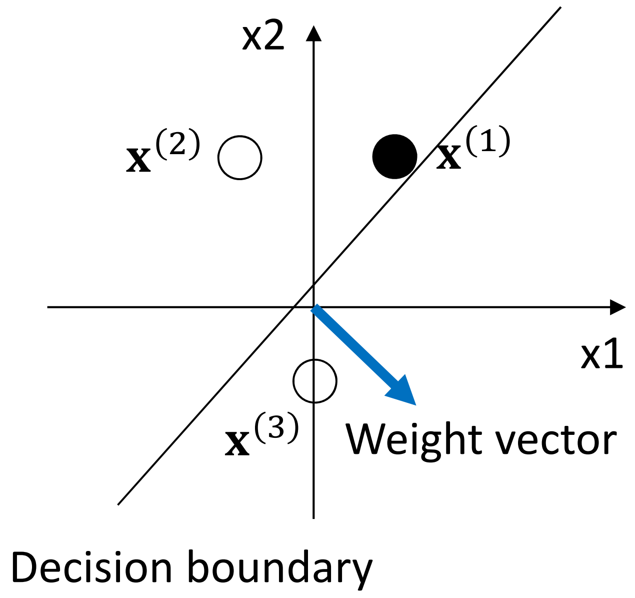
$$\left\{ \mathbf{x}^{(1)} = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}, t_1 = 1 \right\}$$

$$\left\{ \mathbf{x}^{(2)} = \begin{bmatrix} -1 \\ 2 \\ -1 \end{bmatrix}, t_2 = 0 \right\}$$

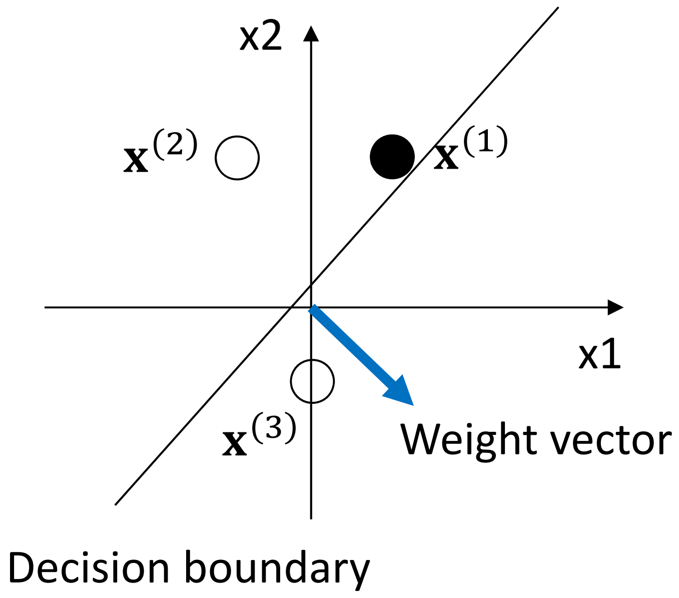
$$\left\{ \mathbf{x}^{(3)} = \begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix}, t_3 = 0 \right\}$$

thershold

$\mathbf{w}_{start}$=[1.0 -0.8 0.5] = $\mathbf{w}_0$



$\mathbf{w}_0$=[1.0 -0.8 0.5]



Step1

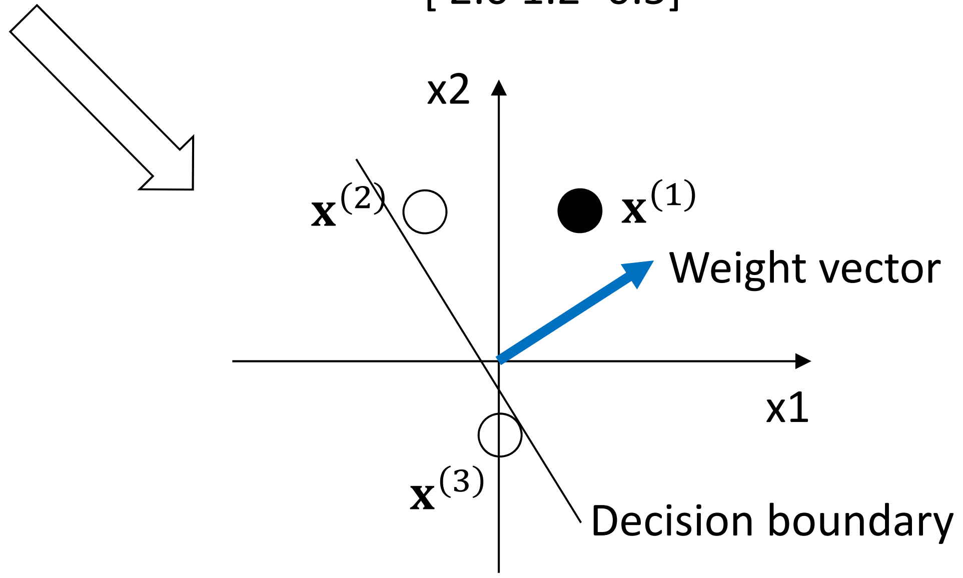
a. Compute u_1

u_1 = heaviside($\mathbf{w}_0$* $\mathbf{x}^{(1)}$)

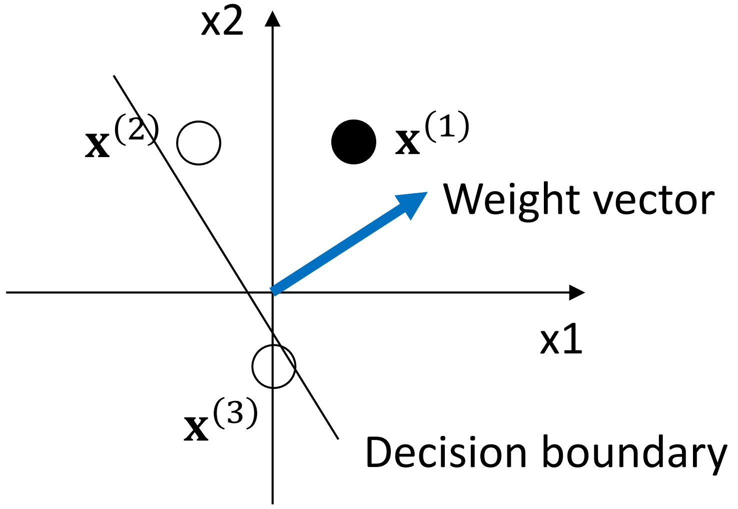
b. Compute $\mathbf{w}_1$

$\mathbf{w}_1 = \mathbf{w}_0 + (t_1 - u_1) * \mathbf{x}^{(1)}$

= [2.0 1.2 -0.5]



$\mathbf{w}_1$=[2.0 1.2 -0.5]



Step2

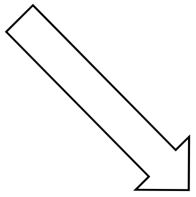
a. Compute u_2

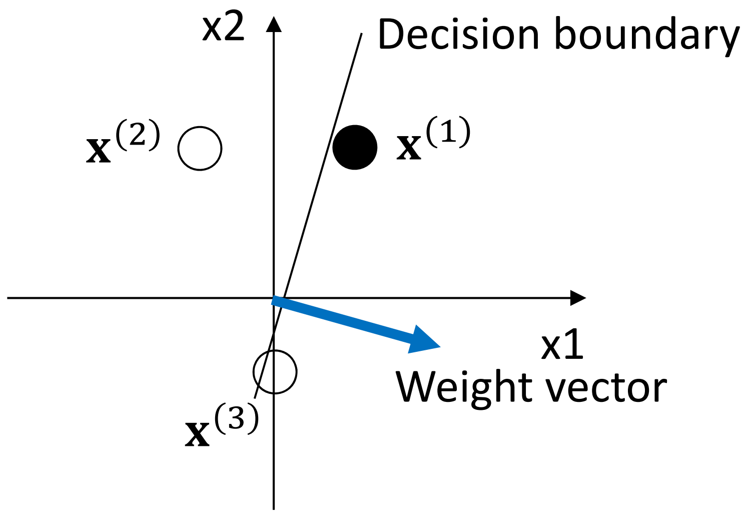
u_2 = heaviside($\mathbf{w}_1$* $\mathbf{x}^{(2)}$)

b. Compute $\mathbf{w}_2$

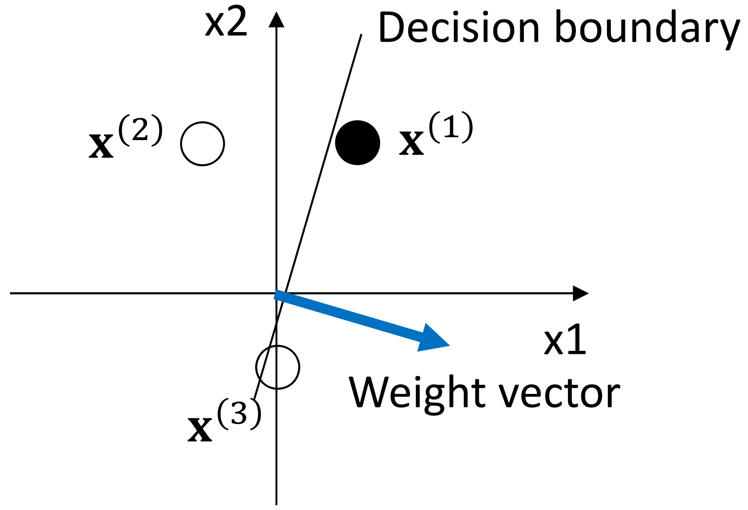
$\mathbf{w}_2 = \mathbf{w}_1 + (t_2 - u_2) * \mathbf{x}^{(2)}$

= [3.0 -0.8 0.5]





$\mathbf{W}_2$=[3.0 -0.8 0.5]



Counterpart: degrees of freedom: there are MANY DIFFERENT weight set satisfy the classification!

Step3

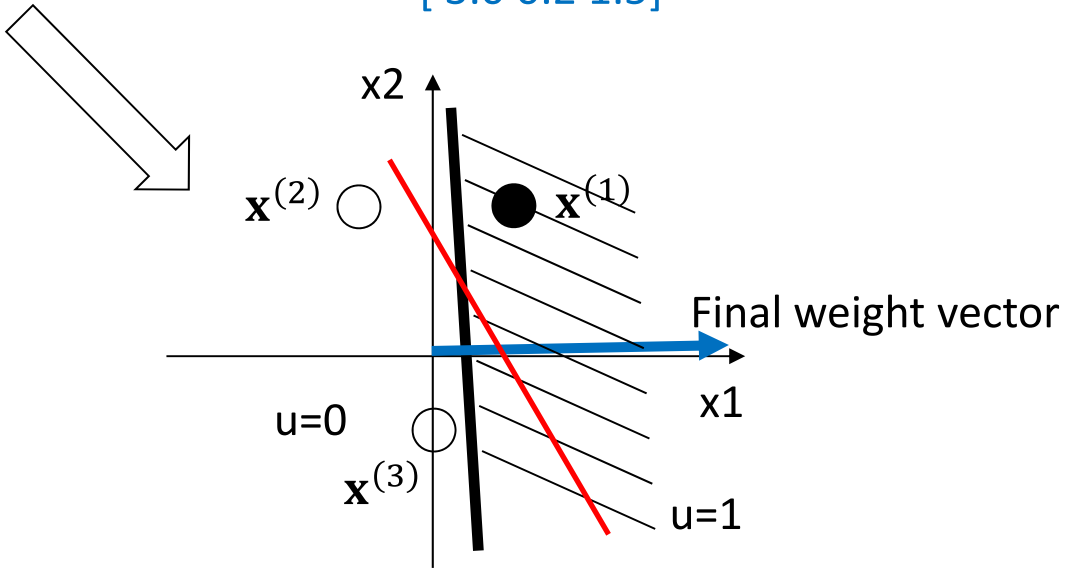
a. Compute u_3

u_3 = heaviside($\mathbf{w}_2$* $\mathbf{x}^{(3)}$)

b. Compute $\mathbf{w}_1$

$\mathbf{w}_3 = \mathbf{w}_2 + (t_3 - u_3) * \mathbf{x}^{(3)}$

= [3.0 0.2 1.5]



Final decision boundary

You can achieve a linear separation, so if you have complicated, or weird distrubution of pattern elements you can't assume that you achieve a separation with a linear boundary. What you can argue is that you could add an additional neuron in the output... but it's actually not right. You need to add an hidden layer.

Perceptron learning rule

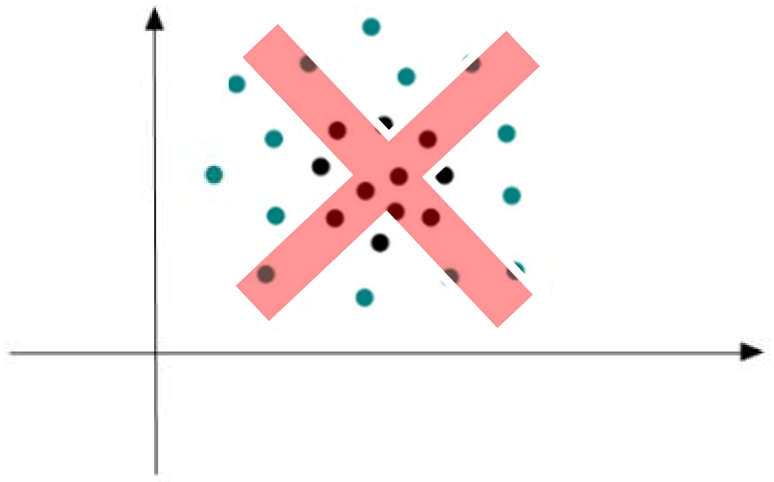
The decision boundary is always orthogonal to the weight vector.

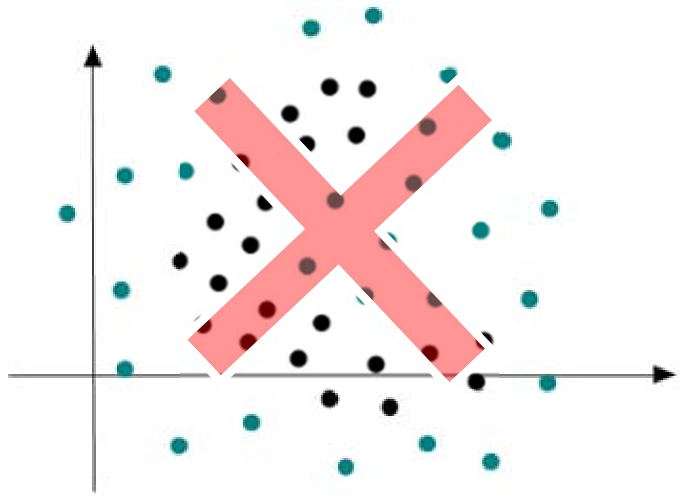
The perceptron learning rule is guaranteed to converge to a solution in a finite number of steps, so long as a solution exists.

but …

linearly separable problem only

SPOILER: IL THRESHOLD DI UN PERCEPTRONE TI FA SCORRERE LA DECISION BOUNDARY LUNGO IL VETTORE DEI WEIGHTS. QUINDI SI', IL WEIGHTS SENZA COSNIDEREARE IL THRESHOLD E' QUELLO PERPENDICOLARE AL DECISION BOUNDARY

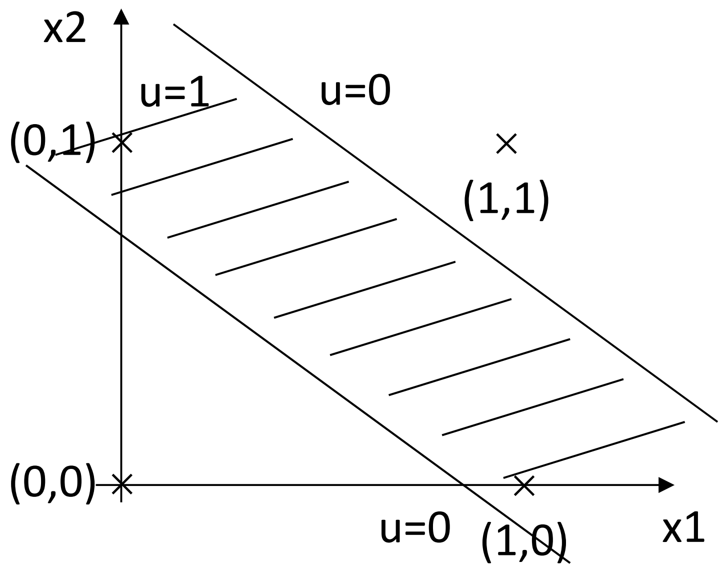




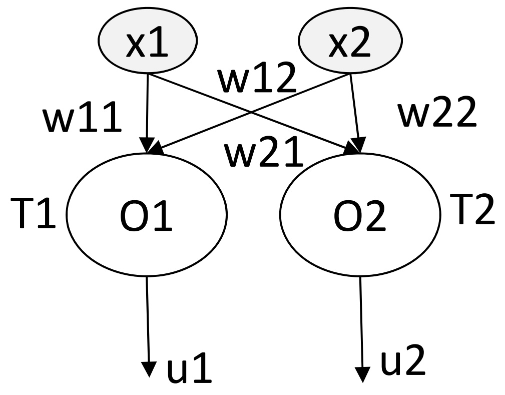
Perceptron as a linear binary classifier

- Perceptron can only solve problems that are linearly separable
  - In N-dimensional space it must exist a hyper-plane w1x1+w2x2+w3x3+…=0 that separates completely the patterns in between
- The perceptron with one single neuron cannot solve the XOR classification

  - What if we assume a perceptron with 2 output units?

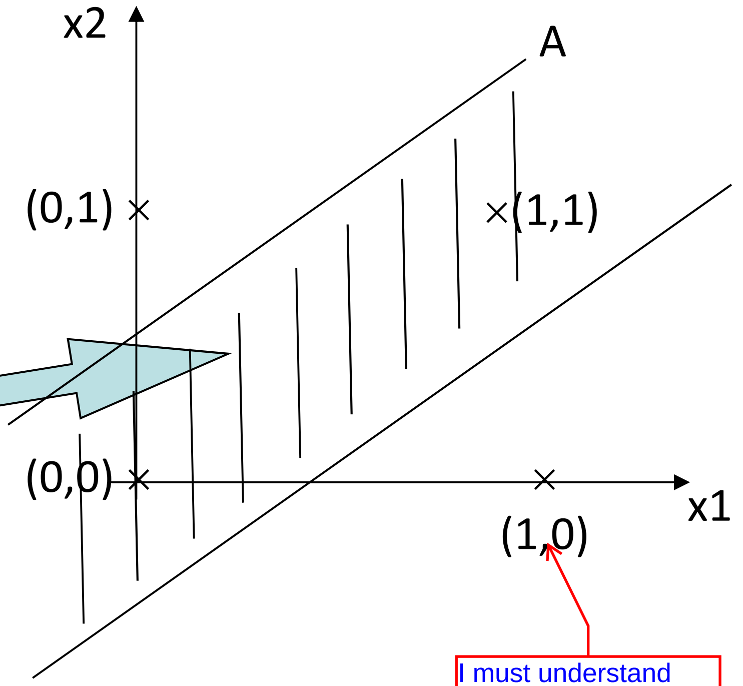


Multiple output perceptron



- With this architecture the network allows up to 4 categories to be classified
- (No, No) (Yes, Yes) (Yes, No) (No, Yes)

w11=1, w12=-1, T1=-0.5
w21=-1, w22=1, T2=-0.5
A: x1-x2+0.5=0
B: -x1+x2+0.5=0



x1=1 x2=1-> u1=1 u2=1
x1=0 x2=0-> u1=1 u2=1

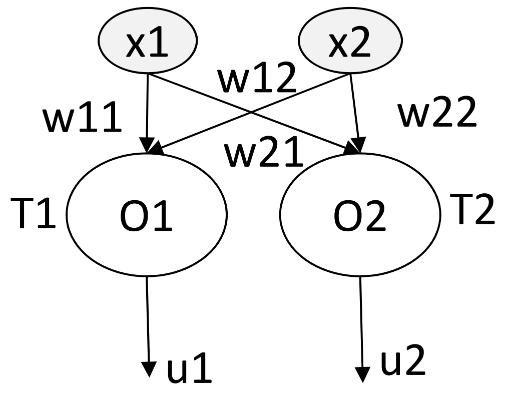
that could be nice

x1=1 x2=0-> u1=1 u2=0
x1=0 x2=1-> u1=0 u2=1

different combinations for the same class!

I must understand that this one and the other must be of the same class, but they have different combinations

Multiple output perceptron

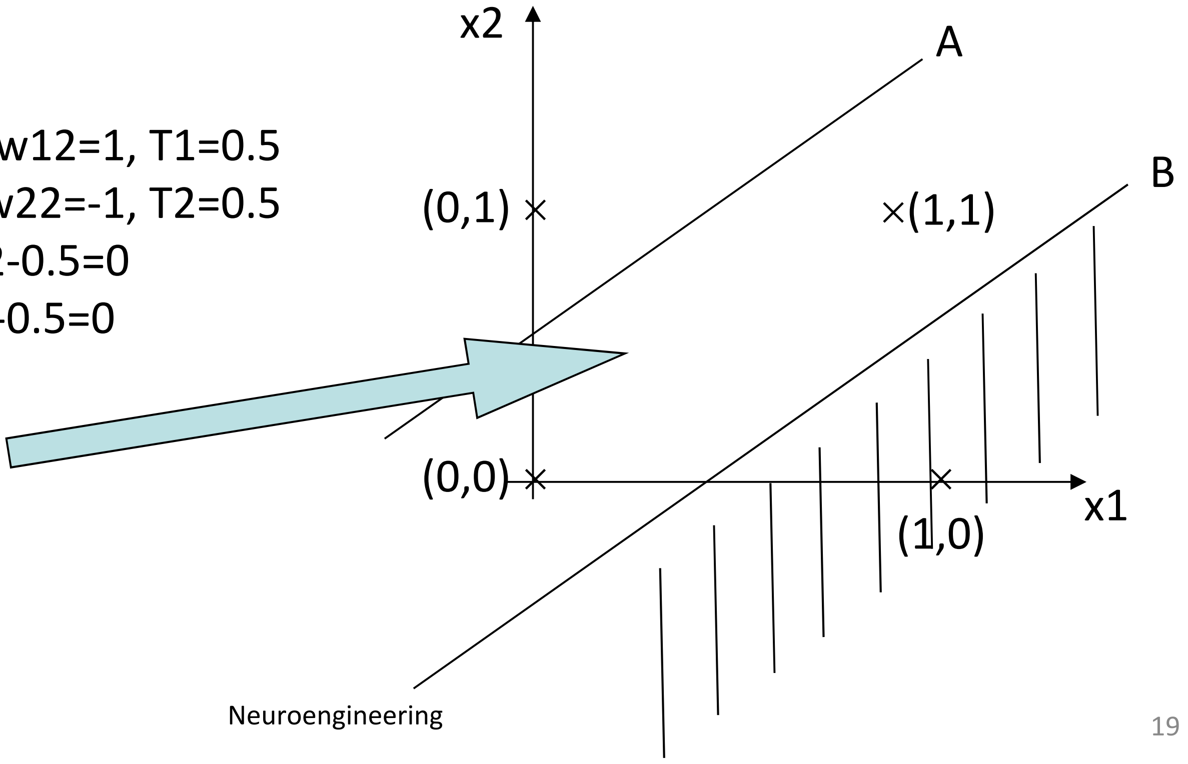


- With this architecture the network allows up to 4 categories to be classified
- (No, No) (Yes, Yes) (Yes, No) (No, Yes)

w11=-1, w12=1, T1=0.5
w21=1, w22=-1, T2=0.5
A: -x1+x2-0.5=0
B: x1-x2-0.5=0

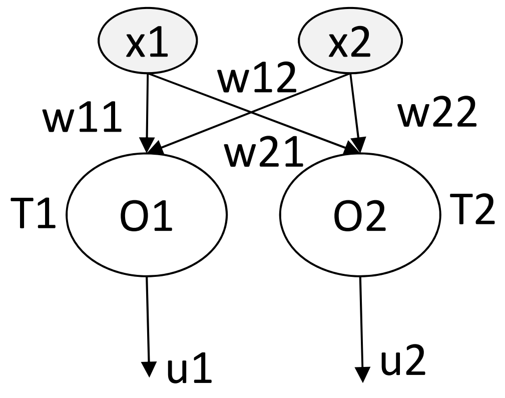
x1=1 x2=1-> u1=0 u2=0
x1=0 x2=0-> u1=0 u2=0

x1=1 x2=0-> u1=0 u2=1
x1=0 x2=1-> u1=1 u2=0



There's no way! :(You won't have convergence, you keep skipping and changing from one configuration to another configuration, there's no way to having the correct classification.

Multiple output perceptron

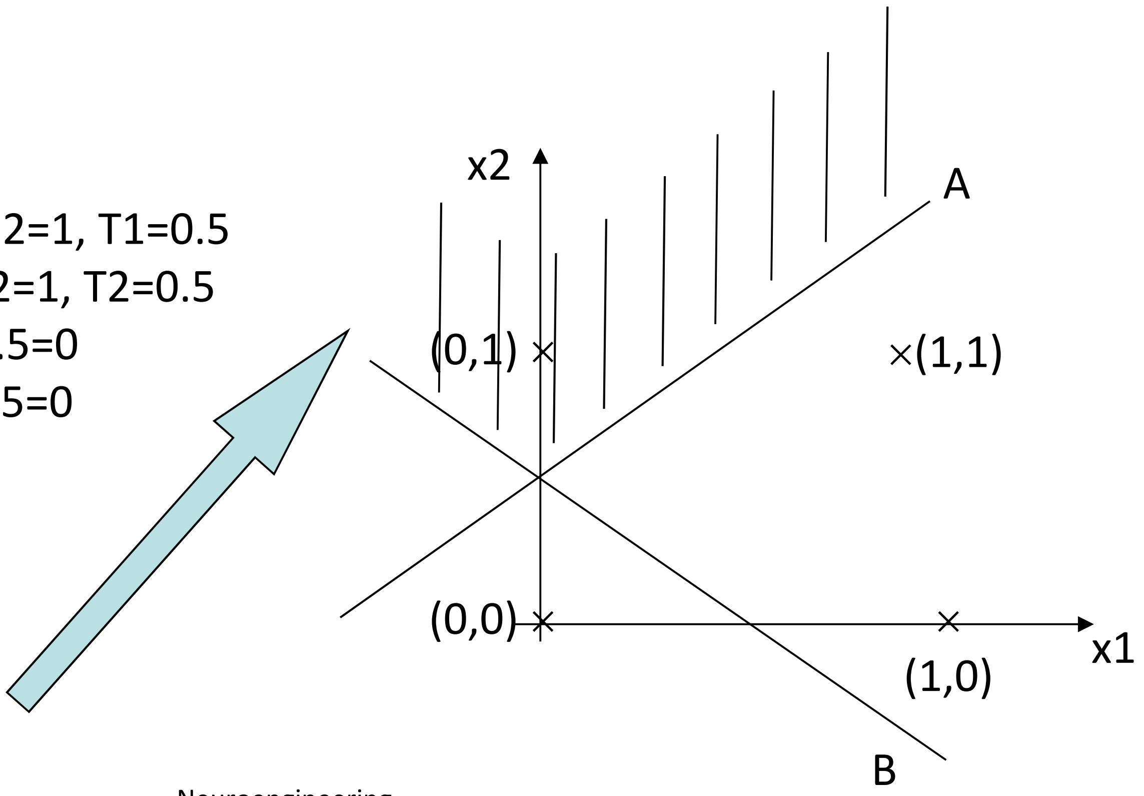


- With this architecture the network allows up to 4 categories to be classified
- (No, No,) (Yes, Yes) (Yes, No) (No, Yes)

w11=-1, w12=1, T1=0.5
w21=1, w22=1, T2=0.5
A: -x1+x2-0.5=0
B: x1+x2-0.5=0

x1=1 x2=1-> u1=0 u2=1
x1=0 x2=0-> u1=0 u2=0

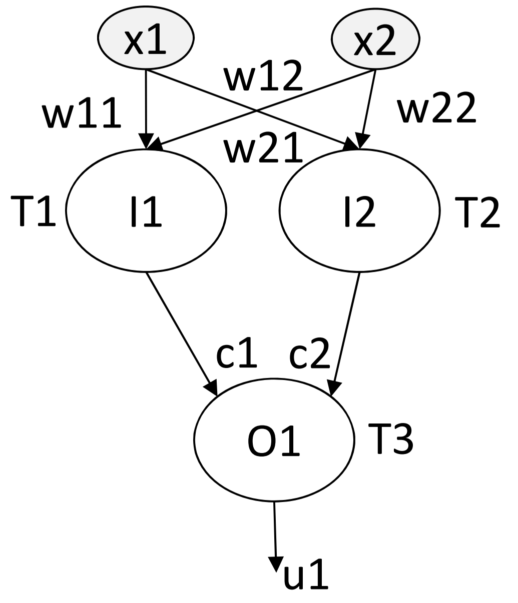
x1=1 x2=0-> u1=0 u2=1
x1=0 x2=1-> u1=1 u2=1



This converges to a solution :).

When I talk about perceptron I'm assuming that the activation function is an heavyside function

XOR operator (need an additional layer)



BOOLEAN NETWORKS

- Perceptron with multiple units in the output layer can perform classification more complex than Perceptron with a single unit in the output layer
- But…other classifications cannot be performed at all
- Three layer Perceptron can solve the problem of XOR operator

See example

25-09

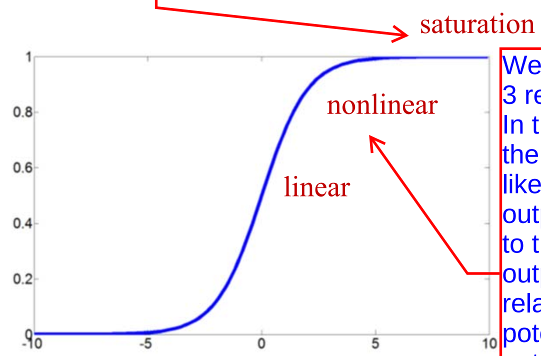
output is fixed! No matter the input

Role of saturation: smooth down high inputs, in both directions

Extending the perceptron

$$P_i = \sum_{j=1}^{M} w_{ij} x_j(t) = w_{i1}x_1 + w_{i2}x_2 + w_{i3}x_3 + .. + w_{iM}x_M + T_i(-1)$$

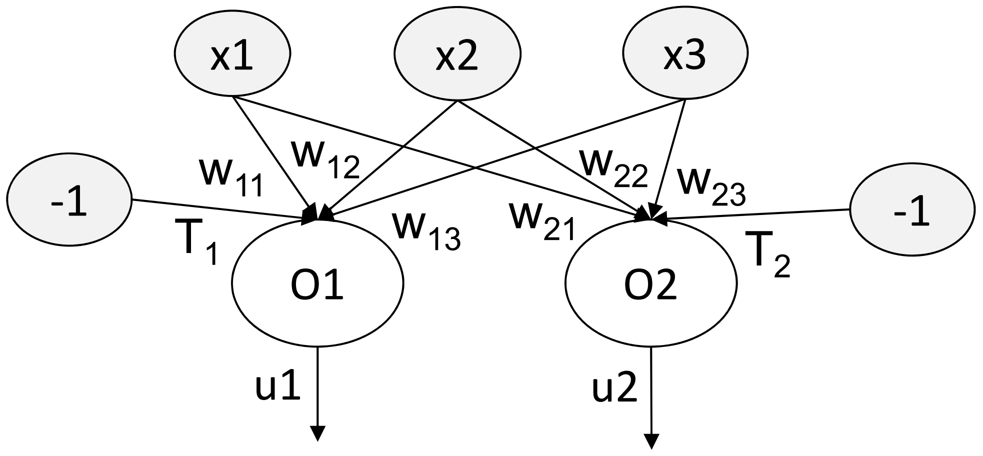
- Continuous input/output
- **Non-linear activation function**
- x_s: signals of the input (1:M)
- u_s : signal of the output neurons (1:N)
- w_{ij} connection weight between the jth input and ith output neuron



We can discriminate 3 regions: sat, nl, l. In the linear region the unit is behaving like a linear unit: the output is proportional to the input.The output is linearly related to the action potential. Then we get the NL, that tries to smoothly reduce the effect of the AP on the activation functions, then sat.

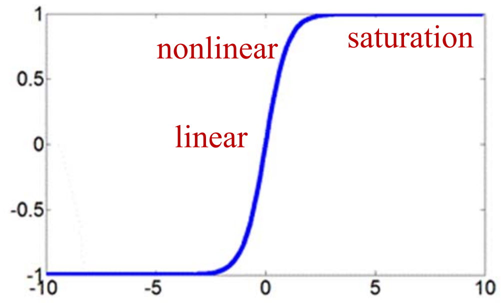
Sigmoidal

$$u = \text{logsig}(P_i) = \frac{1}{1+e^{-P_i}}$$



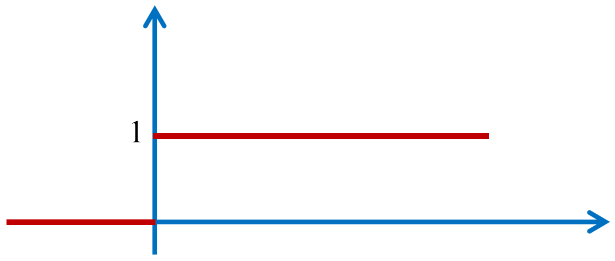
Hyperbolic

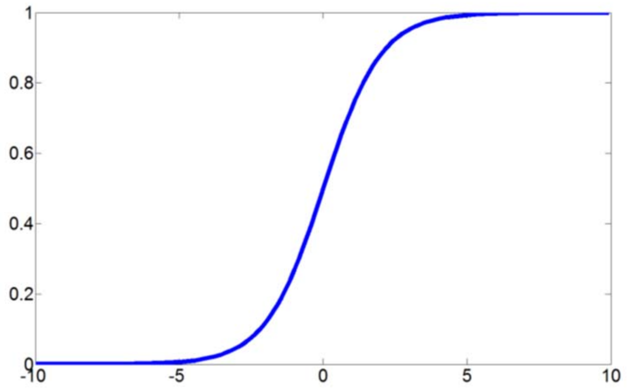
$$u = f(P_i) = \tanh(P_i) = \frac{e^{P_i} - e^{-P_i}}{e^{P_i} + e^{-P_i}}$$



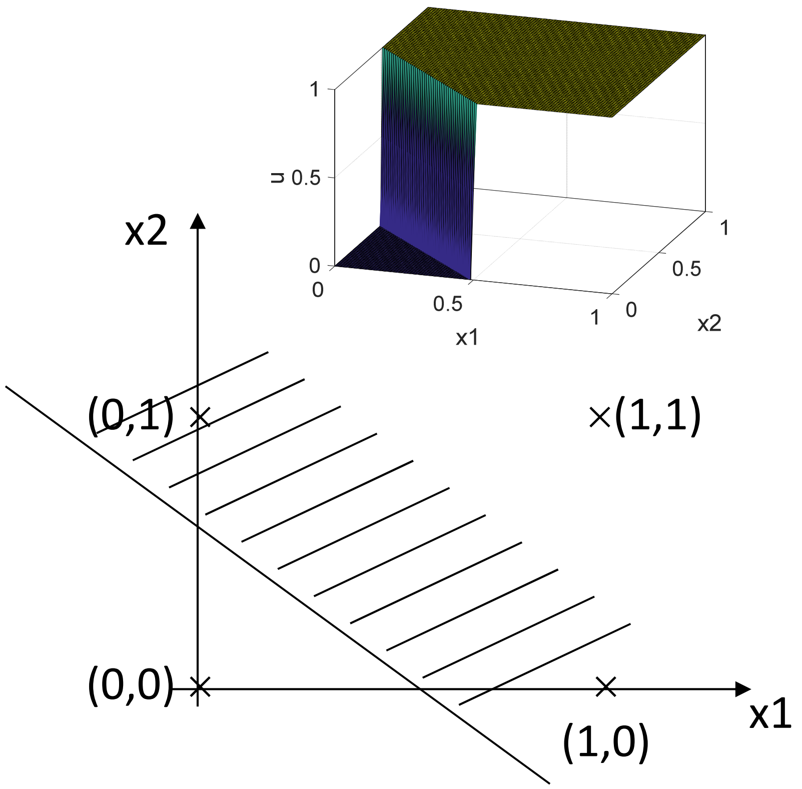
Here it changes the dynamics of the output, here the output goes from -1 to +1! (with sigmoidal from 0 to 1). This allows to provide a negative output.

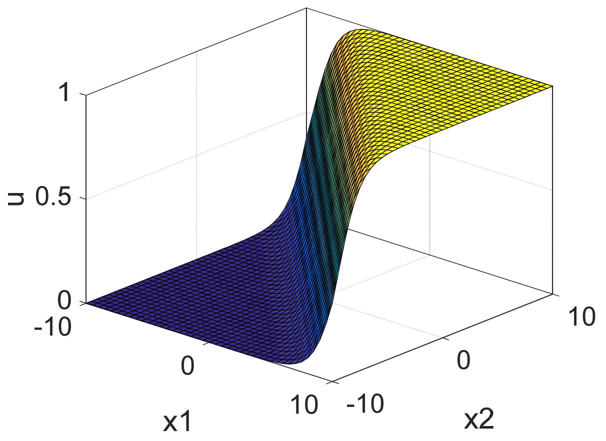
Sign -> Sigmoidal





$$u = \text{logsig}(P) = \frac{1}{1+e^{-P}}$$

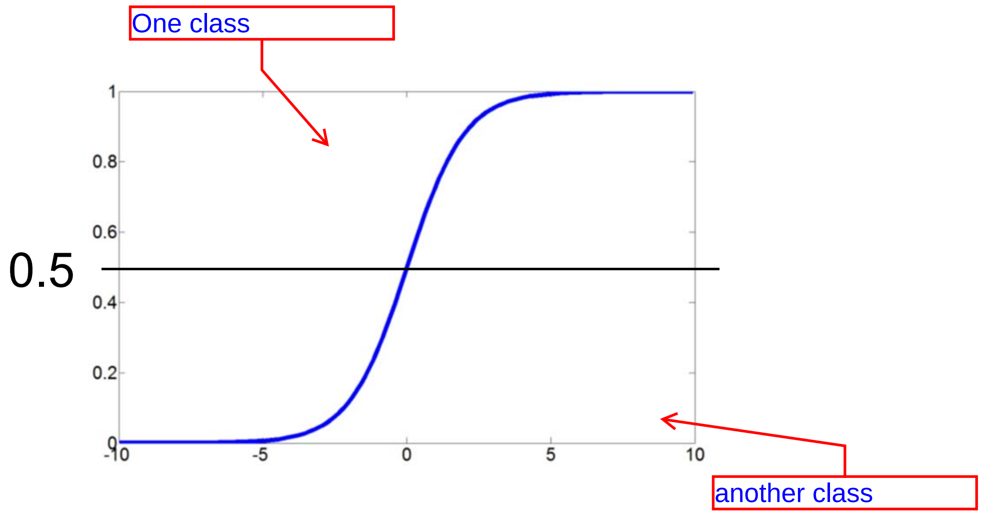


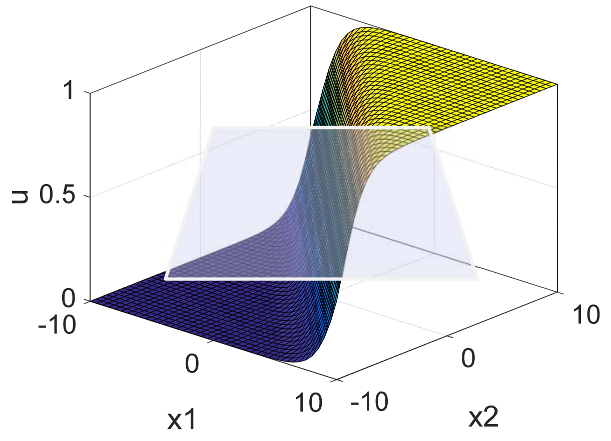


I can also use sigmoidal for classification. Now I don't have a binary output but I can assign a membership to the output. The classical way is to set a threshold to the activation function at like 0.5 for the sigmoid. But there are more efficient solutions (softmax).

Binary classification with Sigmoidal

Need a manual threshold on the output





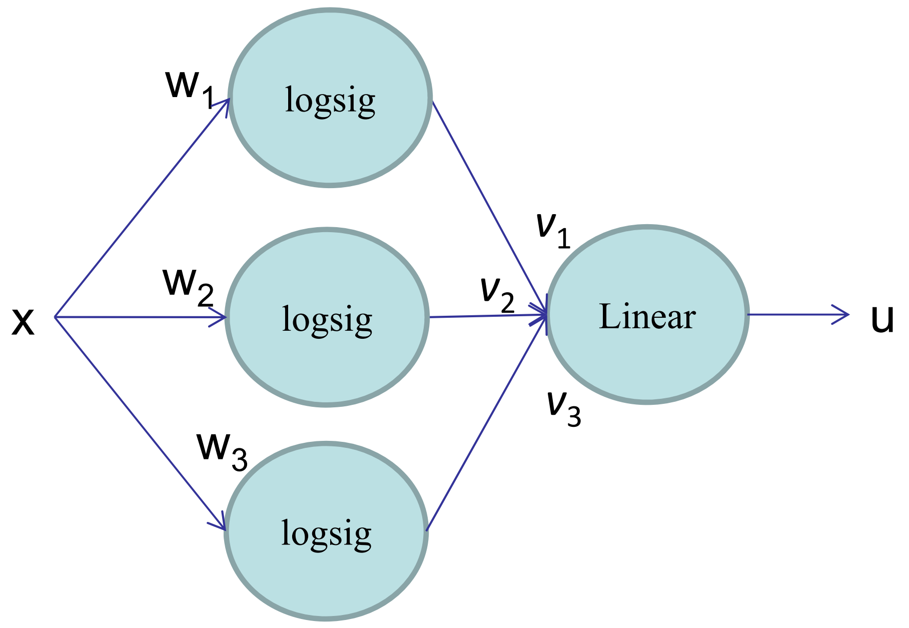
Class 0: u >=0.5
Class 1: u <0.5

better solution: using Softmax network (see next)

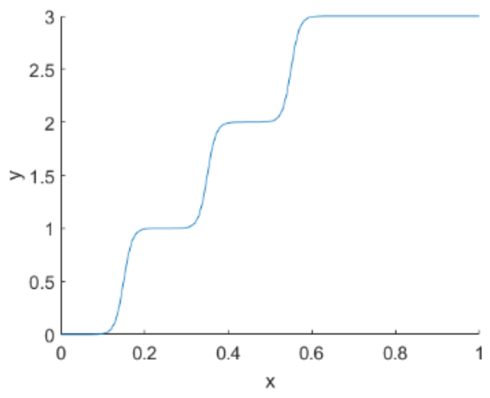
Continuous input/Continuous output

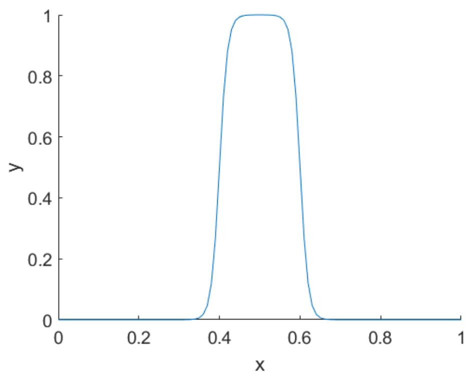
Function modeling

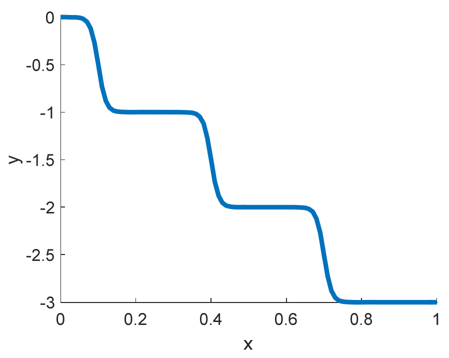
e.g. u is a linear combination of logistic functions



$$u = v_1 * \log\text{sig}(w_1 x - T_1) + v_2 * \log\text{sig}(w_2 x - T_2) + v_3 * \log\text{sig}(w_3 x - T_3) - T_4$$







We have a set of pattterns with corrisponding references (the starting condition is the same like with the perceptron rule). With least square optimization, based on building the power of the error and trying to minize a function of this.

Learning with C^1 activation functions

Pattern: input dataset
Supervised learning: the expected(reference) is the output dataset $\{t\}$

- **Error signal E**
  - Training set (R patterns)
  - $u^{(k)}_i$ is the ith output neuron for the kth pattern

error for each element of the set. K is the sample I'm considering

$$E^{(k)} = \sum_{i=1}^{N} \frac{1}{2}\left(t_i^{(k)} - u_i^{(k)}\right)^2$$

is the error of the network for kth input pattern

Then I sumup the error for all the patterns:

$$E = \sum_{k=1}^{R} E^{(k)} = \sum_{k=1}^{R}\sum_{i=1}^{N} \frac{1}{2}\left(t_i^{(k)} - u_i^{(k)}\right)^2$$

over all the R patterns

Network		Pattern
output	$u^{(1)}_1$, $u^{(1)}_2$	1: ($x^{(1)}_1$, $x^{(1)}_2$, $x^{(1)}_3$)
reference	$t^{(1)}_1$, $t^{(1)}_2$	
output	$u^{(2)}_1$, $u^{(2)}_2$	2: ($x^{(2)}_1$, $x^{(2)}_2$, $x^{(2)}_3$)
reference	$t^{(2)}_1$, $t^{(2)}_2$	
		…
output	$u^{(k)}_1$, $u^{(k)}_2$	*k*: ($x^{(k)}_1$, $x^{(k)}_2$, $x^{(k)}_3$)
reference	$t^{(k)}_1$, $t^{(k)}_2$	
		…
output	$u^{(R)}_1$, $u^{(R)}_2$	*R*: ($x^{(R)}_1$, $x^{(R)}_2$, $x^{(R)}_3$)
reference	$T^{(R)}_1$, $t^{(R)}_2$	

R: number of patterns
M=3: input *lines*
N=2: output lines

I expect that the minimizations of the function let me obtain weights and threshold that satisfy the input/output relationship I'm searching for. [[I cannot exploit the linear solution, I don't obtain a closed solution, like the increment of weights is a linear funtion of the error: dW=a*E]] Potential + activation function is highly non linear, so no linear relationshipt between weights and error. So I must search the minimum of the function, moving step by step. Typical way to address optimizations. Like moving along the gradient, or exploiting curvature, and so on.

Problem: how do I define the initial weight set? In case of a very weird error function this could be an issue. "issue of local minimum"-> suboptimal solutions can occur.

Delta rule: Widrow and Hoff (1960)

- Minimization of the error signal E
  - Square makes error positive and penalizes large errors more
  - Use E to update the weights: Δw_{ij} function of E

- Main criteria
  - If E increases with the increase of w_{ij} then the variation Δw_{ij} is to be negative
  - If E increases with the decrease of w_{ij} then the variation Δw_{ij} is to be positive

- Gradient descent method
  - Equation for Δw_{ij} ?
  - Iterative until E is below a predefined threshold (or weight convergence)
  - w_{ij} at the initial step??
  - Stop criterion

Max number of step, thresholds, things like that. In matlab toolbox there are several criteria, and there's a combination

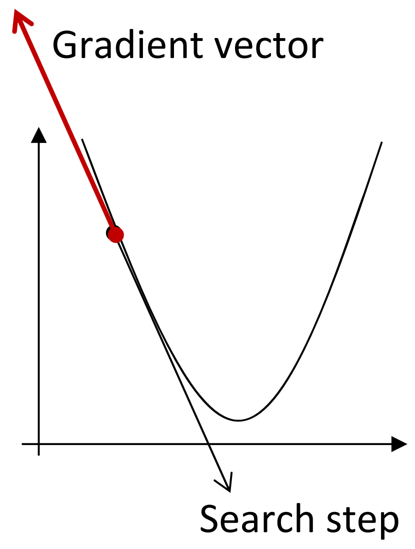
Minimize error on the training set of examples

$$E = \sum_{k=1}^{R} E^{(k)} = \sum_{k=1}^{R}\sum_{i=1}^{N}\frac{1}{2}\left(t_i^{(k)} - u_i^{(k)}\right)^2$$

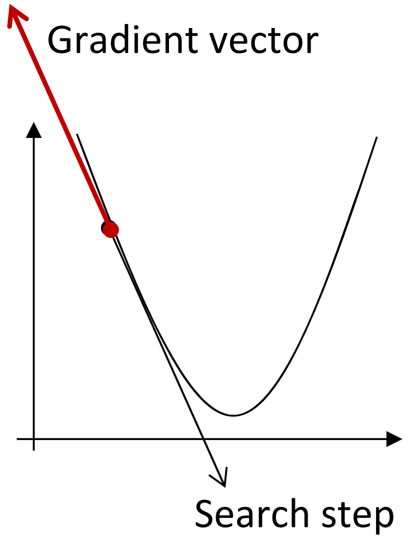
I compute the Jacobian of the function, then I have a vector that is tangent in the point where I compute it.

Gradient descent

- Gradient descent is an optimization algorithm that approaches a local minimum of a function by taking steps proportional to the negative of the gradient of the function as the current point



Gradient descent



- Gradient descent is an optimization algorithm that approaches a local minimum of a function by taking steps proportional to the negative of the gradient of the function as the current point

- So, calculate the derivative (gradient) of the Cost Function with respect to the weights, and then change each weight by a small increment in the negative (opposite) direction to the gradient

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial w} = -(\bar{y} - y)x = -\delta\, x \quad \text{with} \quad \delta = (desired - measured)$$

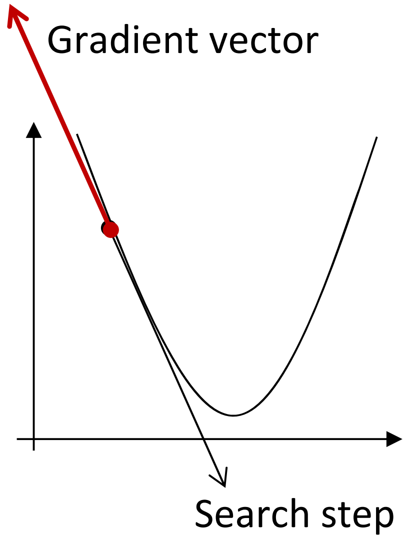
I apply chain rule

When I'm computing the modulus of the error I have to compute the reference minus the output!

Direi fosse scelta E tale che ti viene questo comodo derivando. Poi y = wx, quindi se derivi rispetto a w ti rimane solo x. Direi che ci sia il meno davanti perché la derivata di E è rispetto y calcolato, non il target (che ha - davanti). Questo vale nei casi """lineari""" dopo generalizza

I use an increment of the weights, this is a function of the error, weighted by the learning rate.

Gradient descent



- Gradient descent is an optimization algorithm that approaches a local minimum of a function by taking steps proportional to the negative of the gradient of the function as the current point
- So, calculate the derivative (gradient) of the Cost Function with respect to the weights, and then change each weight by a small increment in the negative (opposite) direction to the gradient

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial w} = -(\bar{y} - y)x = -\delta\, x \quad \text{with} \quad \delta = (desired - measured)$$

- In order to reduce E by gradient descent, move/increment weights in the negative direction of the gradient vector
- η is the learning rate to modulate the amplitude of the gradient vector (to be chosen a priori)
  - $0<\eta<1$

$$\Delta w = f\left(\frac{\partial E}{\partial w}\right)$$

My target: analytical relationship of the weight increment. I can start from computing the gradient (the derivative)

$$w_{new} = w_{old} + \Delta w$$

$$\Delta w = \eta\, \delta\, x$$

$$E=\sum_{k=1}^{R}E^{(k)}=\sum_{k=1}^{R}\sum_{i=1}^{N}\frac{1}{2}\left(t_i^{(k)}-u_i^{(k)}\right)^2$$

sì

$$\frac{\partial E}{\partial w_{ij}}=\frac{\partial\left(\frac{1}{2}\left(t_i-u_i\right)^2\right)}{\partial w_{ij}} \Rightarrow \frac{\partial\left(\frac{1}{2}\left(t_i-u_i\right)^2\right)}{\partial u_i}\frac{\partial u_i}{\partial w_{ij}}$$

I'll have and update for each weight of the network

Apply Chain rule

$$-\left(t_i-u_i\right)\frac{\partial u_i}{\partial w_{ij}}$$

with

$$u_i=f\left(P_i\right)=f\left(\sum_{j}^{M}w_{ij}x_j\right)$$

output: means of action potential AND function f. If we are using sigmoidal function f is sigmoidal function.

$$\Rightarrow -\left(t_i-u_i\right)\frac{\partial u_i}{\partial P_i}\frac{\partial P_i}{\partial w_{ij}} \Rightarrow -\left(t_i-u_i\right)f'\left(P_i\right)\frac{\partial P_i}{\partial w_{ij}}$$

derivative of the activation function

Differential between a linear sum and wij

$$\Rightarrow -\left(t_i-u_i\right)f'\left(P_i\right)\frac{\partial\left(\sum_{j=1}^{M}w_{ij}x_j\right)}{\partial w_{ij}} \Rightarrow -\left(t_i-u_i\right)f'\left(P_i\right)x_j$$

This is the delta rule! This is the increment I must add! (Without the - sign)

[49.00] I cannot use delta rule for step function. Because I have to do the derivative

Delta rule

$$\Delta w_{ij} \approx \left(\frac{\partial E}{\partial w_{ij}} \right)$$

$$\Delta w_{ij} = \eta \sum_{\boxed{k=1}}^{R} \left(t_i^{(k)} - u_i^{(k)} \right) f'\left(P_i^{(k)} \right) x_j^{(k)}$$

Linear

$$f(P_i) = aP_i$$

$$f'(P_i) = a$$

Hyperbolic

$$f(P_i) = \tanh(P_i) = \frac{e^{P_i} - e^{-P_i}}{e^{P_i} + e^{-P_i}}$$

$$f'(P_i) = 1 - (\tanh(P_i))^2$$

Logistic

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

$$f'(P_i) = f(P_i)\left(1 - f(P_i)\right)$$

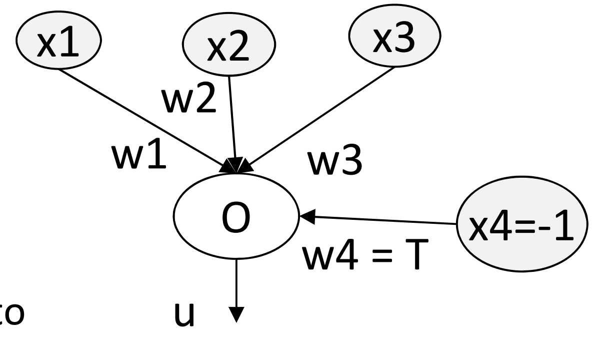
Heaviside

$$f(P_i) = \begin{cases} 0, & P_i < 0 \\ 1, & P_i \geq 1 \end{cases}$$

$$f'(P_i) = \delta(P_i) = \begin{cases} 1, & P_i = 0 \\ 0, & elsewhere \end{cases}$$

Example

Learning AND with logsig activation using the Delta Rule



Output function is sigmoid (logistics)

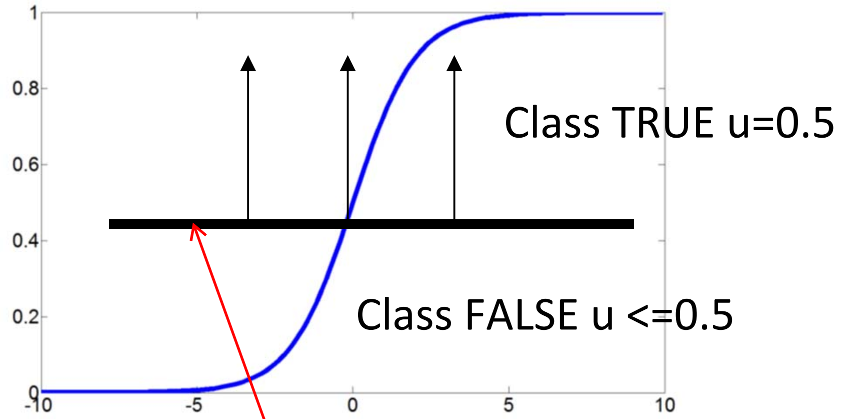
	x1	x2	x3	x4
p1	0	0	0	-1
p2	0	0	1	-1
p3	0	1	0	-1
p4	1	0	0	-1
p5	1	1	0	-1
p6	0	1	1	-1
p7	1	0	1	-1
p8	1	1	1	-1

t
0
0
0
0
0
0
0
1

only condition expected to have 1 in this example. Ofc it can be seen as an AND port, or a classifier with two classes, 0 and 1

- Three input
- 8 patterns
- Weights are initialized to random (Gaussian)
- **On-line updating**
- Learning rate η =0.95

$\mathbf{w}_{\text{start}}$ = [-0.35 0.63 1.45 0.95]



I have to set a threshold! Intuitly I set that to 0.5

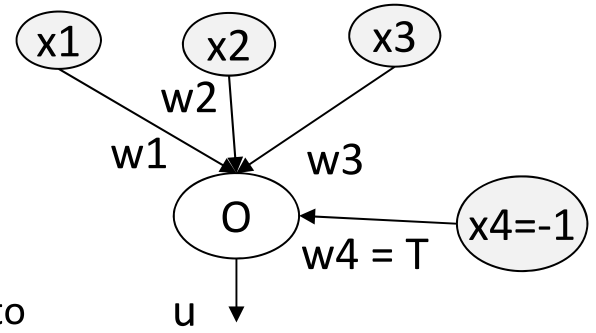
Example

Learning AND with logsig activation using the Delta Rule

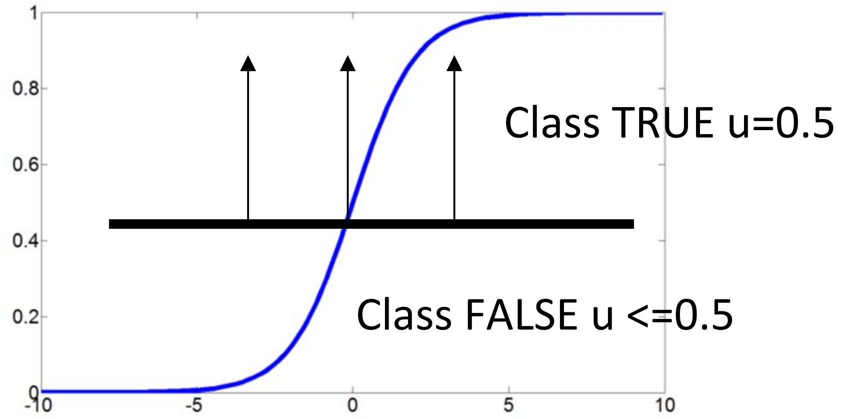
	x1	x2	x3	x4
p1	0	0	0	-1
p2	0	0	1	-1
p3	0	1	0	-1
p4	1	0	0	-1
p5	1	1	0	-1
p6	0	1	1	-1
p7	1	0	1	-1
p8	1	1	1	-1

t
0
0
0
0
0
0
0
1

- Three input
- 8 patterns
- Weights are initialized to random (Gaussian)
- **On-line updating**
- Learning rate η =0.95



Output function is sigmoid (logistics)



$\mathbf{w}_{\text{start}}$ = [-0.35 0.63 1.45 0.95]

Neuron output for each pattern using $\mathbf{w}_{\text{start}}$:

$u^{(1)}$=0.27 $u^{(2)}$=0.62 $u^{(3)}$=0.42 $u^{(4)}$=0.21 $u^{(5)}$=0.33 $u^{(6)}$=0.75 $u^{(7)}$=0.53 $u^{(8)}$=0.68

wrong, I expect <0.5 wrong wrong

Applying Delta rule

$$\left(\Delta w_{ij}\right)^{(k)} = \eta\left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)}$$

$$P = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4(-1)$$

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

$$f'(P_i) = f(P_i)\left(1 - f(P_i)\right)$$

Increment to the threshold

- **Step 1**: I pattern (**p1**)

w1	w2	w3	w4	P	u	t	e	Δw1	Δw2	Δw3	Δw4	nw1	nw2	nw3	nw4
-0.35	0.63	1.45	0.95	-0.95	0.27	0	-0.27	0	0	0	0.05	-0.35	0.63	1.45	1.00

P =x1*w1+x2*w2+x3*w3+w4(-1)

logsig(P)

u-t

η * e* (logsig(P)*(1-logsig(p))) * **p1**(x1)

η * e* (logsig(P)*(1-logsig(p))) * **p1**(x2)

η * e* (logsig(P)*(1-logsig(p))) * **p1**(x3)

η * e* (logsig(P)*(1-logsig(p))) * **p1**(x4)

$$w_{new} = w_{old} + \Delta w$$

Now FOR ALL patterns I compute the new output with these weights

Applying Delta rule

$$\left(\Delta w_{ij}\right)^{(k)} = \eta\left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)}$$

$$P = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4(-1)$$

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

$$f'(P_i) = f(P_i)\left(1 - f(P_i)\right)$$

- **Step 2**: II pattern (**p2**)

Now FOR ALL patterns I compute the new output with these weights. (I'm doing with P2 though now)

w1	w2	w3	w4	P	u	t	e	Δw1	Δw2	Δw3	Δw4	nw1	nw2	nw3	nw4
-0.35	0.63	1.45	0.95	-0.95	0.27	0	-0.27	0	0	0	0.05	-0.35	0.63	1.45	1.00

w1	w2	w3	w4	P	u	t	e	Δw1	Δw2	Δw3	Δw4	nw1	nw2	nw3	nw4
-0.35	0.63	1.45	1.00	-0.44	0.61	0	-0.61	0	0	-0.13	0.13	-0.35	0.63	1.31	1.14

Applying Delta rule

$$\left(\Delta w_{ij}\right)^{(k)} = \eta\left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)}$$

$$P = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4(-1)$$

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

$$f'(P_i) = f(P_i)\left(1 - f(P_i)\right)$$

- **Step 3**: III pattern **(p3)** With P3 now

w1	w2	w3	w4	P	u	t	e	Δw1	Δw2	Δw3	Δw4	nw1	nw2	nw3	nw4
-0.35	0.63	1.45	0.95	-0.95	0.27	0	-0.27	0	0	0	0.05	-0.35	0.63	1.45	1.00

w1	w2	w3	w4	P	u	t	e	Δw1	Δw2	Δw3	Δw4	nw1	nw2	nw3	nw4
-0.35	0.63	1.45	1.00	-0.44	0.61	0	-0.61	0	0	-0.13	0.13	-0.35	1.31	1.45	1.14

w1	w2	w3	w4	P	u	t	e	Δw1	Δw2	Δw3	Δw4	nw1	nw2	nw3	nw4
-0.35	1.31	1.45	1.14	-0.51	0.37	0	-0.37	0	-0.08	0	0.08	-0.35	0.54	1.31	1.22

....

Applying Delta rule

$$\left(\Delta w_{ij}\right)^{(k)} = \eta\left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)}$$

$$P = w_1x_1 + w_2x_2 + w_3x_3 + w_4(-1)$$

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

$$f'(P_i) = f(P_i)\left(1 - f(P_i)\right)$$

- **Step 8**: VIII pattern (**p8**)

w1	w2	w3	w4	P	u	t	e	Δw1	Δw2	Δw3	Δw4	nw1	nw2	nw3	nw4
-0.35	0.63	1.45	0.95	-0.67	0.33	0	-0.33	-0.07	0	-0.07	0.07	-0.49	0.36	1.10	1.50

w1	w2	w3	w4	P	u	t	e	Δw1	Δw2	Δw3	Δw4	nw1	nw2	nw3	nw4
-0.49	0.36	1.10	1.50	-0.52	0.37	**1**	0.62	0.13	0.13	0.13	-0.13	-0.35	0.50	1.24	1.36

Applying Delta rule

$$\left(\Delta w_{ij}\right)^{(k)} = \eta\left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)}$$

$$P = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4(-1)$$

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

$$f'(P_i) = f(P_i)\left(1 - f(P_i)\right)$$

!!!! NO CONVERGENCE (So far)

So I must feed again the network again

- **Step 8**: VIII pattern (**p8**)

w1	w2	w3	w4	P	u	t	e	Δw1	Δw2	Δw3	Δw4	nw1	nw2	nw3	nw4
-0.35	0.63	1.45	0.95	-0.67	0.33	0	-0.33	-0.07	0	-0.07	0.07	-0.49	0.36	1.10	1.50

w1	w2	w3	w4	P	u	t	e	Δw1	Δw2	Δw3	Δw4	nw1	nw2	nw3	nw4
-0.49	0.36	1.10	1.50	-0.52	0.37	**1**	0.62	0.13	0.13	0.13	-0.13	-0.35	0.50	1.24	1.36

Neuron output for each pattern using $\mathbf{w}_{\mathbf{end}}$:

$u^{(1)}=0.20$ $u^{(2)}=0.46$ $u^{(3)}=0.29$ $u^{(4)}=0.15$ $u^{(5)}=0.22$ $u^{(6)}=0.62$ $u^{(7)}=0.33$ $u^{(8)}=0.37$

Still not right with what I expect!

Applying Delta rule

Let us try to reinforce the learning

Feeding the network several times with the same training set of the 8 patterns

Ex: **10** iterations

$\mathbf{w}_{\mathbf{end}}$ = [0.28 0.61 0.97 2.28]

Neuron output for each pattern using $\mathbf{w}_{\mathbf{end}}$:

$u^{(1)}=0.00$ $u^{(2)}=0.21$ $u^{(3)}=0.15$ $u^{(4)}=0.11$ $u^{(5)}=0.22$ $u^{(6)}=0.29$ $u^{(7)}=0.20$ $u^{(8)}=0.27$

Applying Delta rule

Let us try to reinforce the learning

Feeding the network several times with the same training set of the 8 patterns

Ex: **25** iterations

$\mathbf{w}_{\mathbf{end}}$ = [1.00 1.04 1.23 3.06]

Neuron output for each pattern using $\mathbf{w}_{\mathbf{end}}$:

$u^{(1)}=0.00$ $u^{(2)}=0.13$ $u^{(3)}=0.11$ $u^{(4)}=0.11$ $u^{(5)}=0.26$ $u^{(6)}=0.28$ $u^{(7)}=0.25$ $u^{(8)}=0.42$

Applying Delta rule

Let us try to reinforce the learning

Feeding the network several times with the same training set of the 8 patterns

Ex: **25** iterations

$\mathbf{w}_{\mathbf{end}}$ = [1.00 1.04 1.23 3.06]

Neuron output for each pattern using $\mathbf{w}_{\mathbf{end}}$:

$u^{(1)}$=0.00 $u^{(2)}$=0.13 $u^{(3)}$=0.11 $u^{(4)}$=0.11 $u^{(5)}$=0.26 $u^{(6)}$=0.28 $u^{(7)}$=0.25 $u^{(8)}$=0.42

Applying Delta rule

Let us try to reinforce the learning

Feeding the network several times with the same training set of the 8 patterns

Ex: **50** iterations

$\mathbf{w}_{\mathbf{end}}$ = [1.57 1.50 1.66 4.15]

Neuron output for each pattern using $\mathbf{w}_{\mathbf{end}}$:

$u^{(1)}=0.01$ $u^{(2)}=0.07$ $u^{(3)}=0.06$ $u^{(4)}=0.06$ $u^{(5)}=0.25$ $u^{(6)}=0.25$ $u^{(7)}=0.24$ $u^{(8)}=0.54$

!!!! FINALLY CONVERGENCE (what is the role of η)

$\Downarrow$

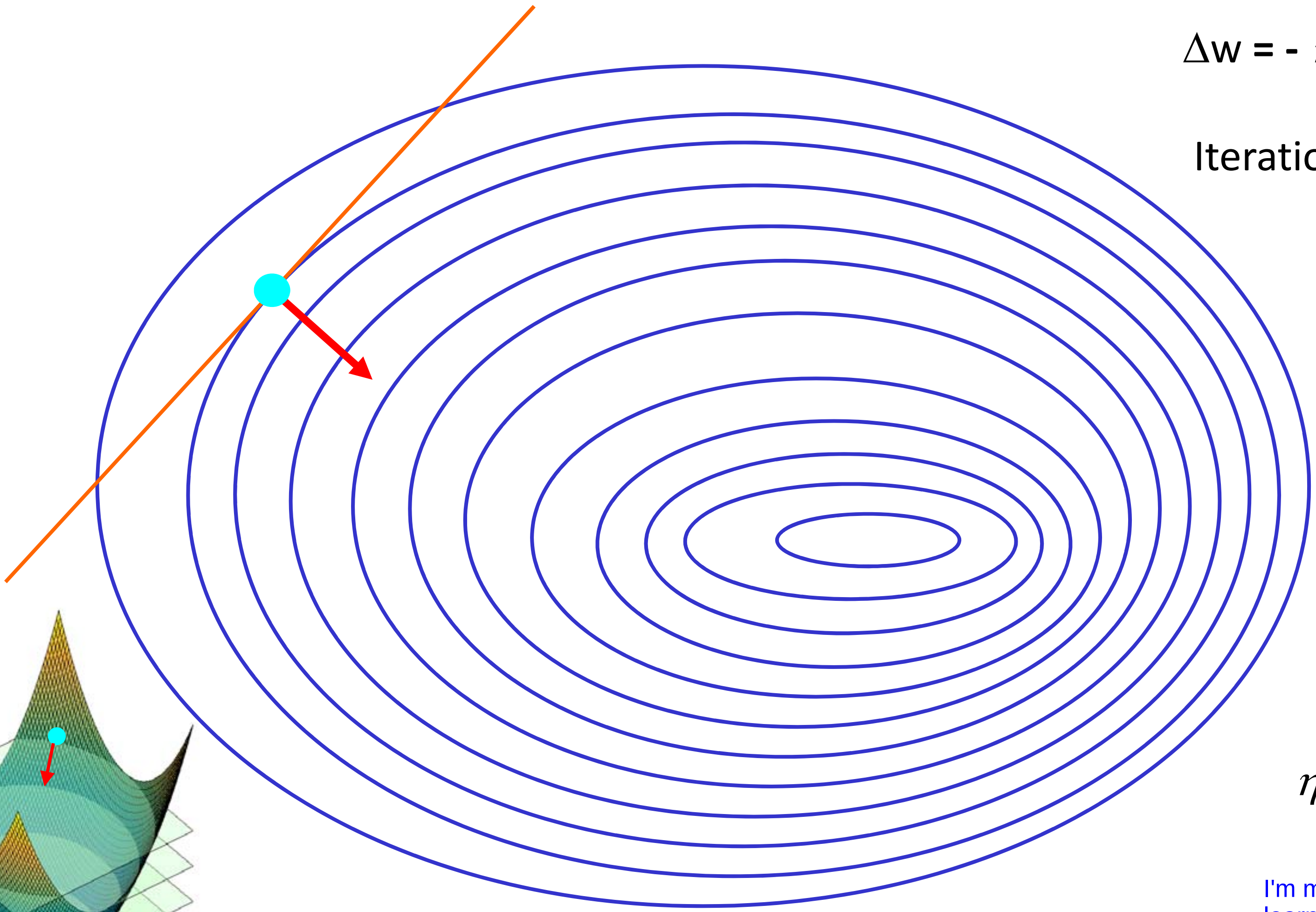
Number of iterations is critical if we do not use any other stop criterion

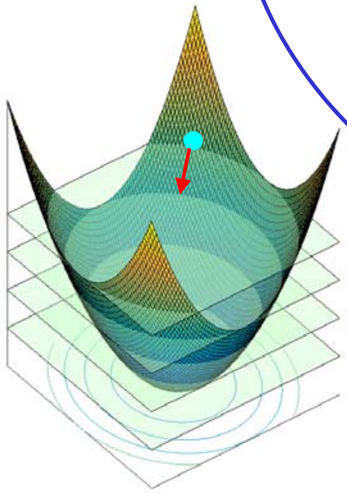
Example, instead of having an hyperbole in the space, here we are in 4 dimensional space

Derivative based Direct Search :
the Gradient Method

$\Delta w = - \eta \cdot \nabla f(x)$

Iteration # 1





Contour plot of the cost function

Δw = Step

η = Learning Rate

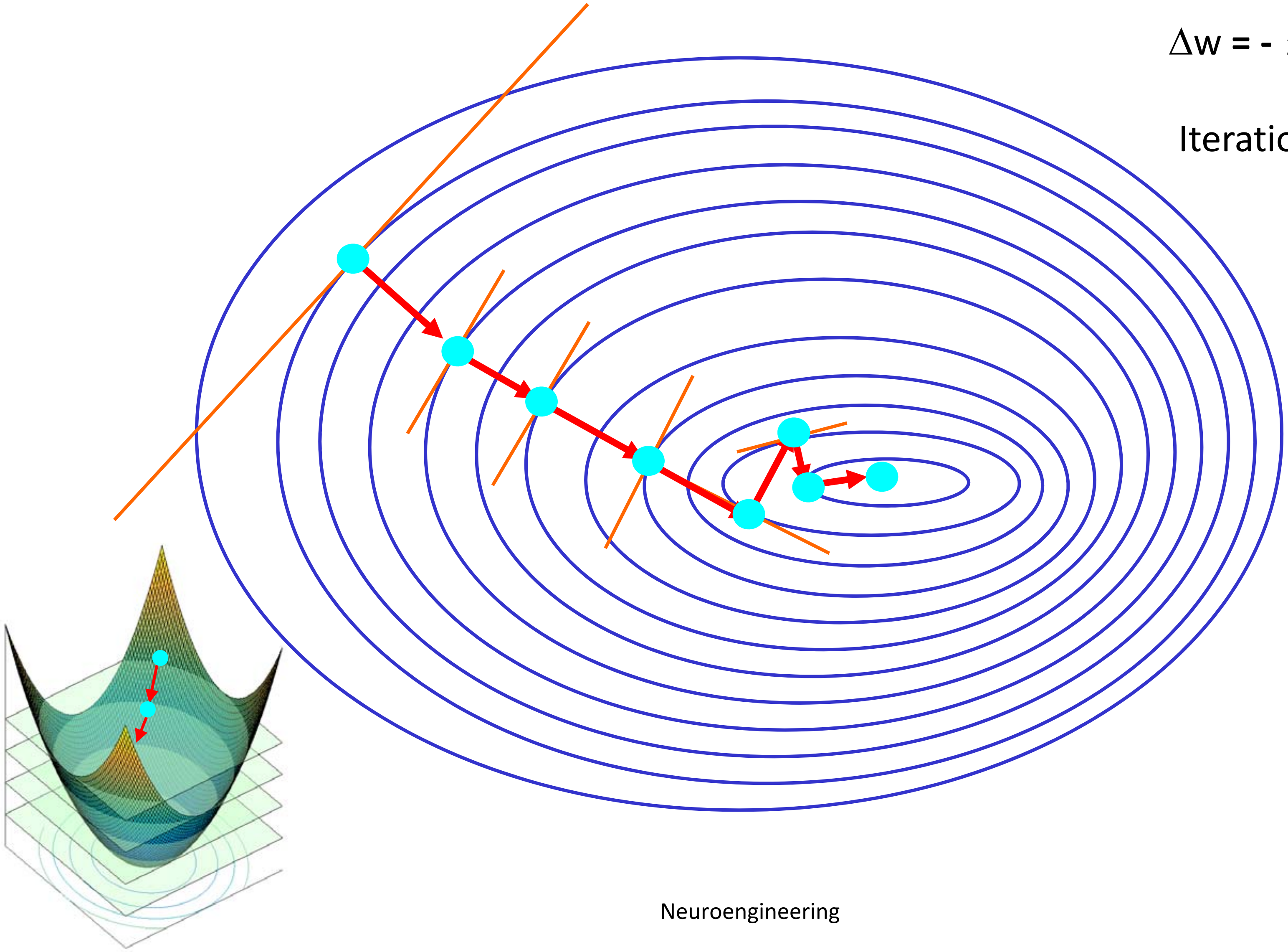
I'm modulating the movement with the learning rate.

Derivative based Direct Search :

the Gradient Method

$\Delta w = - \eta \cdot \nabla f(x)$

Iteration #	1
	2
	3
	…



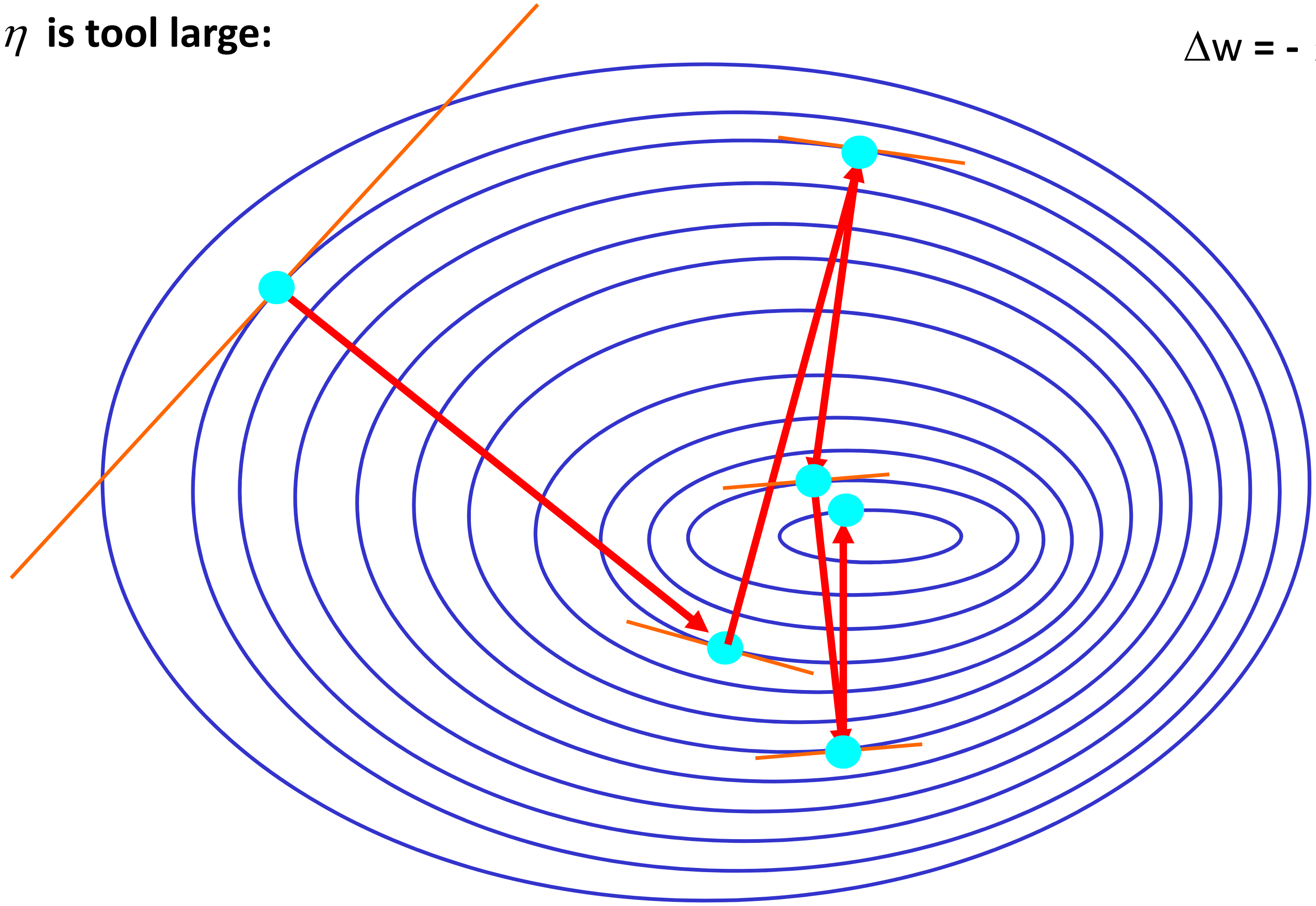
I cannot reach that region! The grading I compute is too high

Derivative based Direct Search :

the Gradient Method

If η is tool large:

$\Delta w = - \eta \cdot \nabla f(x)$



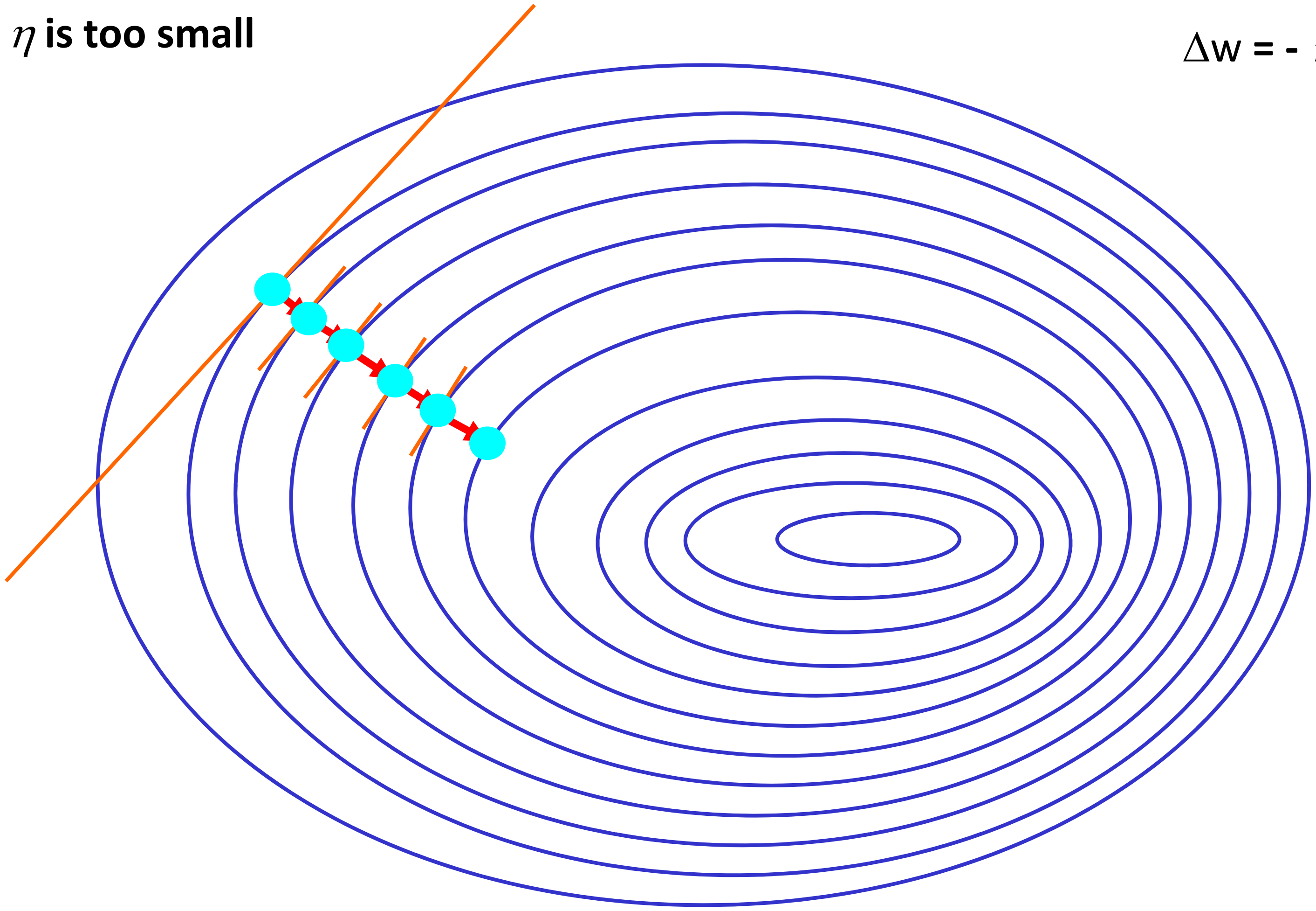
In this case I need more iterations

Derivative based Direct Search :

the Gradient Method

If η is too small

$\Delta w = - \eta \cdot \nabla f(x)$

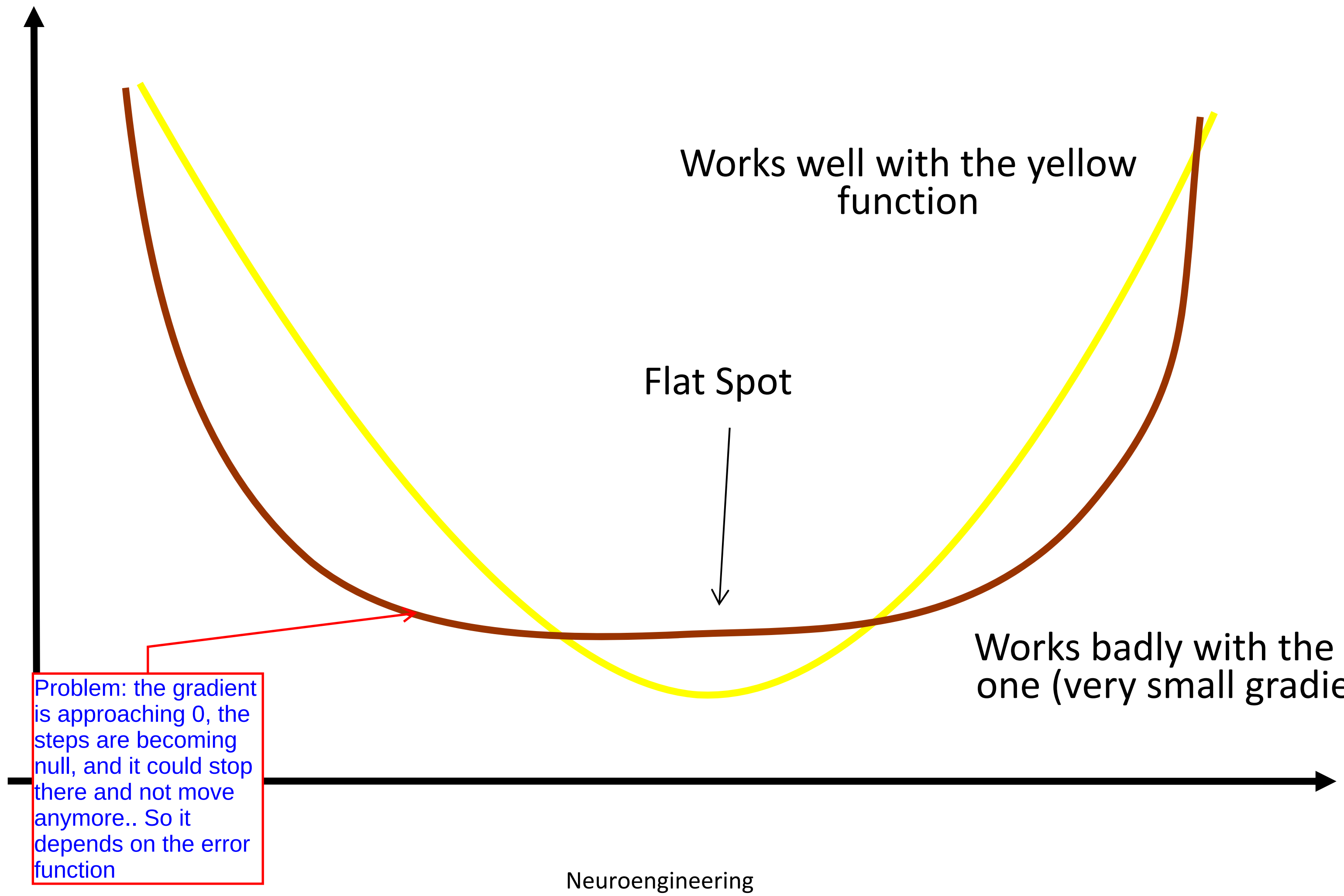


There's actually no rule, it's a parameter you have to set by trying, like trial and error activity

Derivative based Direct Search :

the Gradient Method

$\Delta w = - \eta \cdot \nabla f(x)$



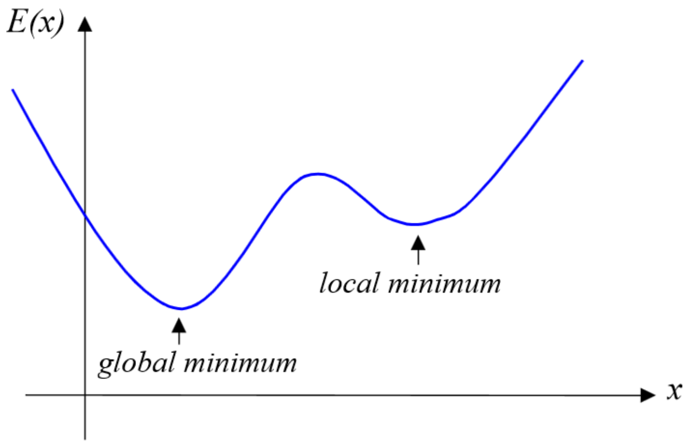
Problem: the gradient is approaching 0, the steps are becoming null, and it could stop there and not move anymore.. So it depends on the error function

Another problem: local minima. When you have many neurons the error functions becomes complicated and easily a lot different from a paraboloid. So the starting point is fondamental. So if the learning rate is very small it could be hard to "skip" a local minimum. But again, if the learning rate is too high you could not reach the minimum at all.

Notes on delta rule

- Local minima
  - Dependence on the starting guess (initial values of the weights)

If we start nearby a the local minimum, we may end up at the that point rather than the global minimum. Starting with a range of different initial weight sets increases our chances of finding the global minimum. Any variation from true gradient descent will also increase our chances of stepping into the deeper valley



You could also train with linear functions (that have no boundaries) But in this case, if you have samples that are misclassified this can trigger instability in the network. Instead if the output is saturated this effect is minimized. The output could increaase a lot, and instabiliy on the weights. Also generally the output range is important, usually it's better to use hyperbolic tangent, because you have a sort of "double range". This means that your output has an output range double than the logistic function. The span of moving of the activation function is more. It increases the degree of freedom, and learning is better.

Notes on delta rule

- Local minima
  - Dependence on the starting guess (initial values of the weights)
- Dependence on the activation function
  - Linear/Non-Linear

Differentiable activation function is important for the gradient descent algorithm to work.

Linear functions are suitable for continuous output but may lead to parameter unbound

The sigmoid(logistic) function ranges from 0 to 1

Better is having a range between -1 and 1 increasing the learning ability

Notes on delta rule

- Local minima
  - Dependence on the starting guess (initial values of the weights)
- Dependence on the activation function
  - Linear/Non-Linear
- On-line and batch updating
  - On-line updating implies that each pattern error contributes sequentially to the weight updating (selection can be random)

$$\Delta w_{ij} = \eta \left(t_i - u_i\right) f'\left(P_i\right) x_j$$

Notes on delta rule

- Local minima
  - Dependence on the starting guess (initial values of the weights)
- Dependence on the activation function
  - Linear/Non-Linear
- On-line and batch updating
  - On-line updating implies that each pattern error contributes sequentially to the weight updating (selection can be random)

each pattern is doing its contribute. With the learning rate I'm smooting time by time

$$\Delta w_{ij} = \eta \left(t_i - u_i\right) f'\left(P_i\right) x_j$$

I could also use the batch update, computing the output for all samples and accumulate the increments in boundle.

  - Batch updating implies that all the pattern errors are cumulated before updating the other weights

$$\Delta w_{ij} = \eta \sum_{k=1}^{R} \left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)}$$

you can't understand the contribute of each pattern, you are losing the selectivity, you are "reasoning" with a training set. With the learning rate I'm smoothing the summation.

On-line learning does not perform true gradient descent, and the individual weight changes can be rather erratic. Normally a much lower learning rate η will be necessary than for batch learning. However, overall the learning is often much quicker. This is particularly true if there is a lot of redundancy in the training data, i.e. many training patterns containing similar information

Notes on delta rule

- Local minima
  - Dependence on the starting guess (initial values of the weights)
- Dependence on the activation function
  - Linear/Non-Linear
- On-line and batch updating
  - On-line updating implies that each pattern error contributes sequentially to the weight updating (selection can be random)

$$\Delta w_{ij} = \eta (t_i - u_i)\, f'(P_i)\, x_j$$

  - Batch updating implies that all the pattern errors are cumulated before updating the other weights

$$\Delta w_{ij} = \eta \sum_{k=1}^{R} \left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)}$$

- <u>Learning rate</u>

When I'm approaching saturation from one point on there's no change in discrimating the output. The output is the same but the action potential could be any, we are losing the resolution of the action potential (From a certain point on). If one neuron during learning is producing AP in the sat. region, its effect can become erratic, or producing instability. The contribution on the delta rule its zero. There's the derivative! But that in the "sat zone" it's zero. So it loses its ability to contribute to the weights. One consideration could be to remove the neuron, maybe it's useless to learning. You know, oversampling, overdetermination etc. I'm using a spline, I'm over parametizing my functions, I'm using too many neurons to rapresent what I actually need. Maybe here I'm also overfitting. We could have a solution: check if we are approaching the sat line, and putting a threshold to the potential, so avoiding to triggering the sat region. Or we could act on the derivative of the function: you let the neuron overrun the sat point, but with a limit on the derivative, like != 0 (with an offset).

Flat Spots in the Error Function

- The gradient descent weight changes depending on the **gradient of the error function**. Consequently, if the error function has flat spots, the learning algorithm can take a long time to pass through them.
- A particular problem with the sigmoidal transfer functions is that the derivative tends to zero as it saturates (i.e. gets towards 0 or 1). This means that the weight updates are very small and the learning algorithm cannot easily correct itself. There are two simple solutions:

- Target Off-sets
  - Use targets of 0.1 and 0.9 (say) instead of 0 and 1. The sigmoids will no longer saturate and the learning will no longer get stuck.
- Sigmoid Prime Off-set
  - Add a small off-set (of 0.1 say) to the sigmoid derivative so that it is no longer zero when the sigmoids saturate

- We should keep the initial network weights small enough so that the sigmoids are not saturated before training. Off-setting the targets also has the effect of stopping the network weights growing too large.

Notes on supervised learning

- Local minima
  - Dependence on the starting guess (initial values of the weights)
- Dependence on the activation function
  - Linear/Non-Linear
- On-line and batch updating
  - On-line updating implies that each pattern error contributes sequentially to the weight updating (selection can be random)

$$\Delta w_{ij} = \eta (t_i - u_i) f'(P_i) x_j$$

  - Batch updating implies that all the pattern errors are cumulated before updating the other weights

$$\Delta w_{ij} = \eta \sum_{k=1}^{R} \left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) x_j^{(k)}$$

- Learning rate
- <u>Stop criteria</u>

Stop criteria

Maximum number of iterations $t < T_{\max}$

We don't care what's the final error, this assures me to control the computation time.

Euclidean norm of the gradient vector less than a predefined threshold $\left\| \frac{\partial E}{\partial w} \right\| < \delta$

When I get close to a minimum it gets lower and lower

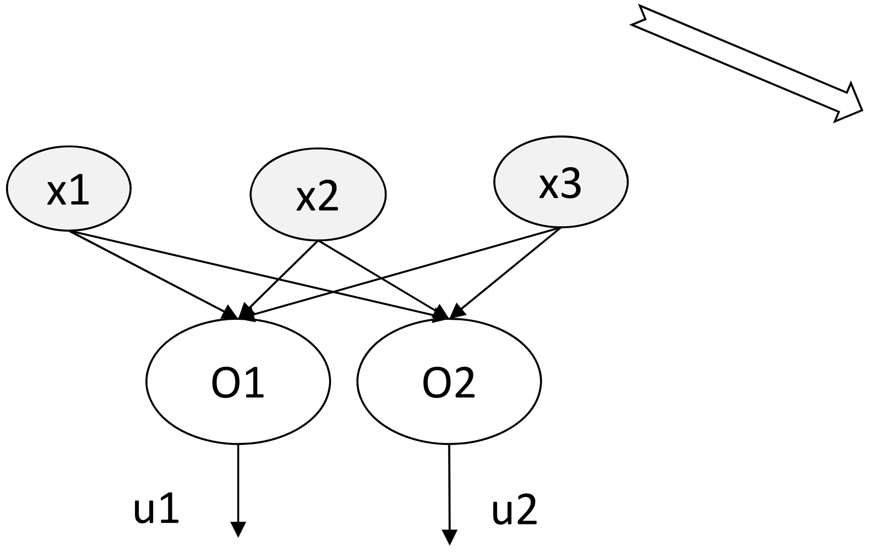
Error function less a predefined threshold $E < \varepsilon$

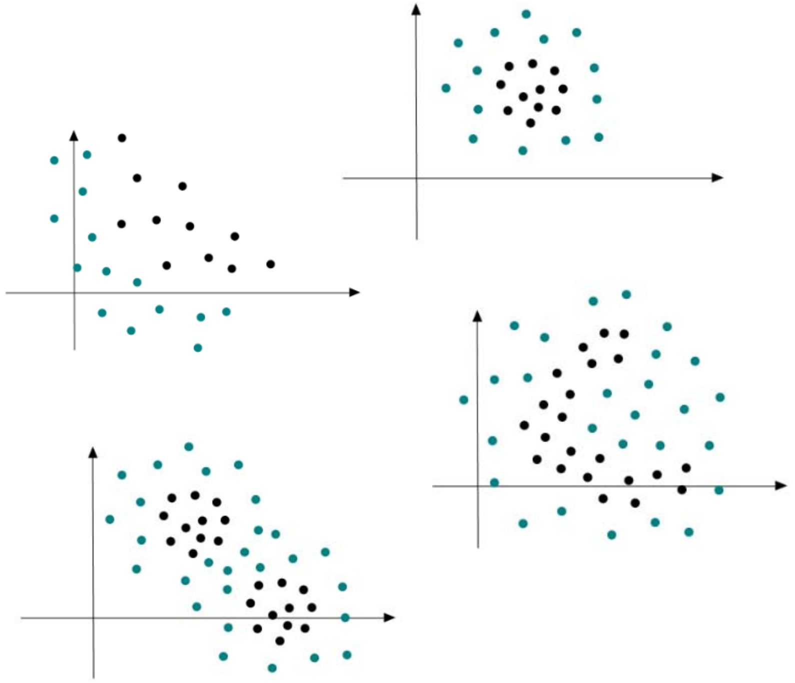
Criterion on the final error, a threshold that is sufficient to me

Hybrid criterion $\alpha \left\| \frac{\partial E}{\partial w} \right\| + \beta E < \gamma$

Issues for single layer networks

- Only linearly separable problems
  - Delta rule does not converge
  - Delta rule does not minimize the number of mistakes (error in reproducing the output)

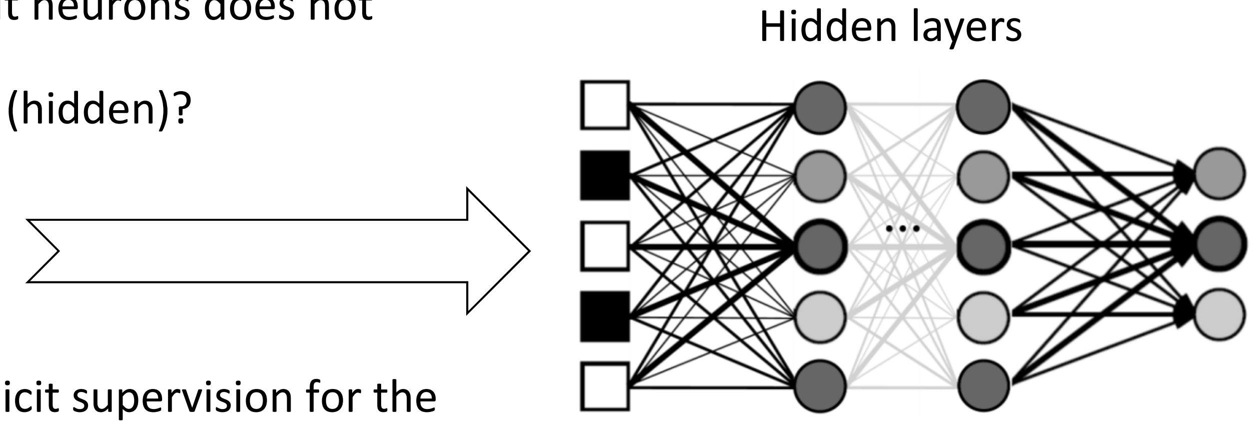




In this case this kind of network is no more valid. If I add a layer it's sufficient to deal with NL separable problems! How do I use the delta rule for the hidden layer!?! In that case I don't know the reference output, I can't directly use the delta rule, but I can generalize that and use back propagation.

Issues for single layer networks

- Only linearly separable problems with perceptron (sign activation)
- Logsig/Tanh activation
  - Only one predefined non-linearity
  - For small P the transform is almost linear
- Adding more output neurons does not help…
- Adding inter-layers (hidden)?

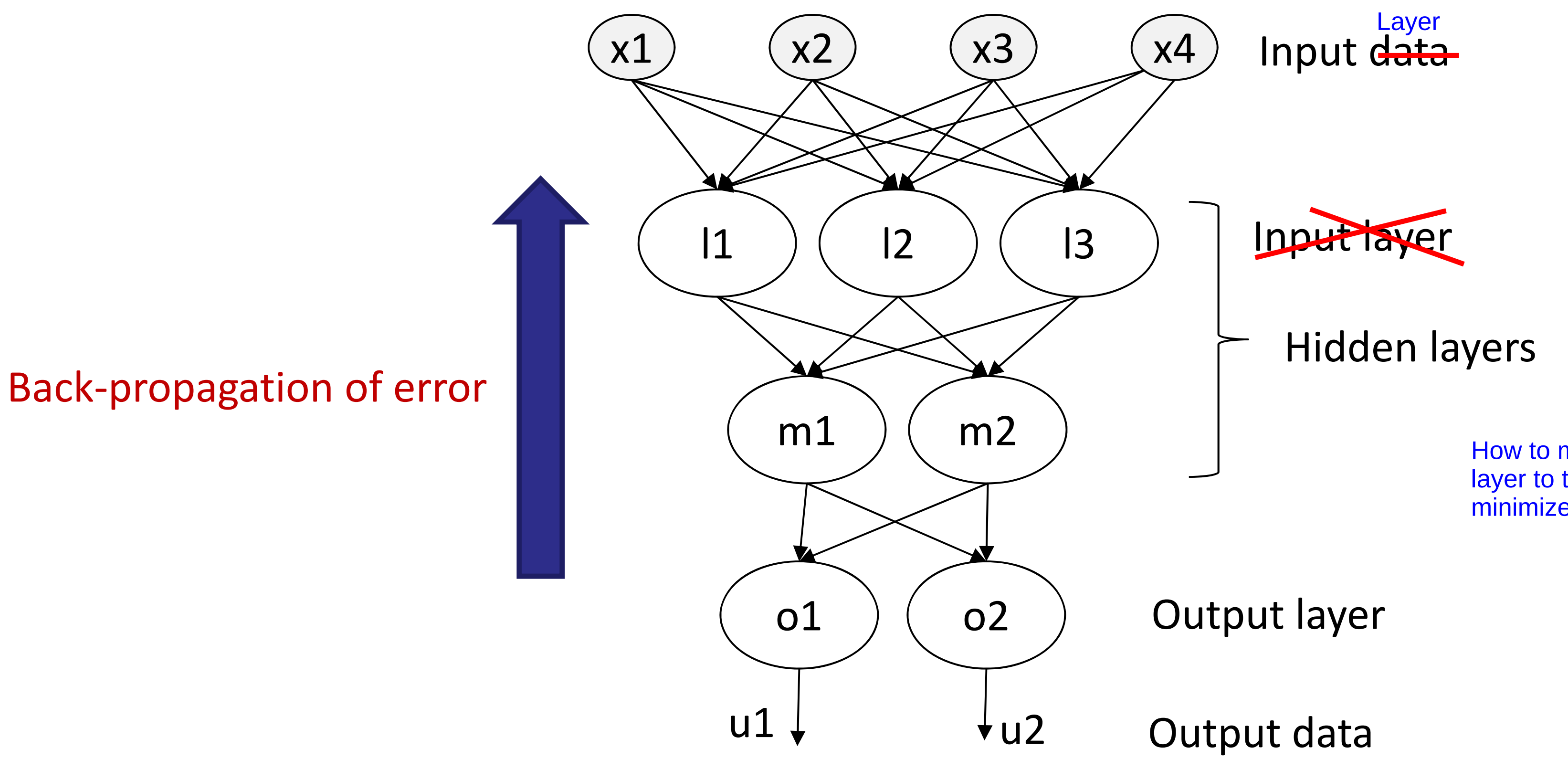


- But there is no explicit supervision for the output of hidden layer neurons
  - How to compute the weights of the hidden layer neurons?
- Solution: 1986 Supervised training technique based on the back-propagation of the error

Multi-layer network training (feed-forward networks)

Rumelhart, Hinton, Williams 1986 (Nature)

- Signal error is propagated from output layers back to hidden layers so that all the synaptic weights can be updated

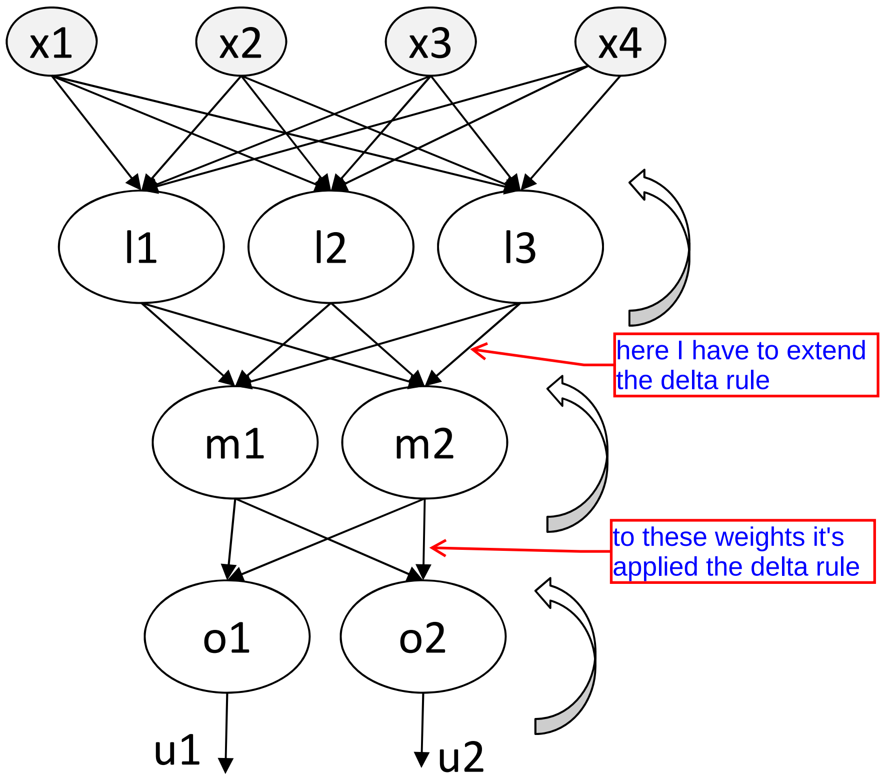


How to move the error from the output layer to the hidden layer? I still have to minimize the error

If I'm considering a very number of layers technically works, but the first layers the error is becoming small. So it can happen that the error is similar to noise! This is solved by learning with subset of layers.

Learning through Back-propagation

1. A set of examples for training the network is assembled. Each pattern consists represents the input into the network and the corresponding (desired) solution as the output from the network.
2. The input data is entered into the network via the input layer.
3. Each neuron in the network processes the input data with the resulting values steadily "running" through the network, layer by layer, until a result is generated by the output layer.
4. The actual output of the network is compared to expected output for that particular input. This results in the *error value.* The connection weights in the network are gradually adjusted, working backwards from the output layer, through the hidden layer, and to the input layer, until the correct output is produced. Fine tuning the weights in this way has the effect of teaching the network to produce the correct output for a particular input, i.e. the network *learns*.

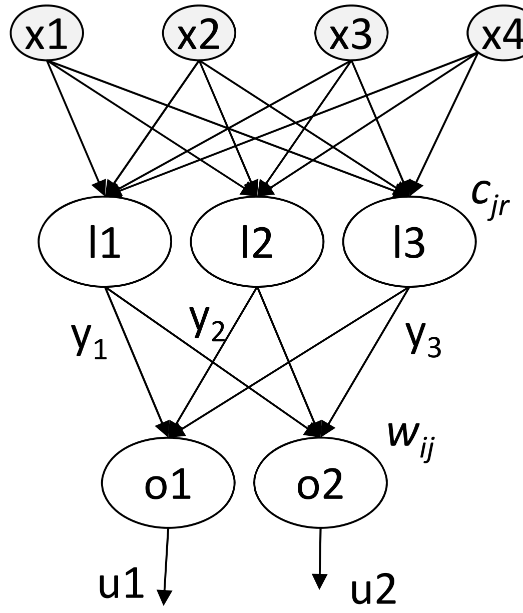


x1
x2
x3
x4
l1
l2
l3
m1
m2
o1
o2
u1
u2
here I have to extend the delta rule
to these weights it's applied the delta rule

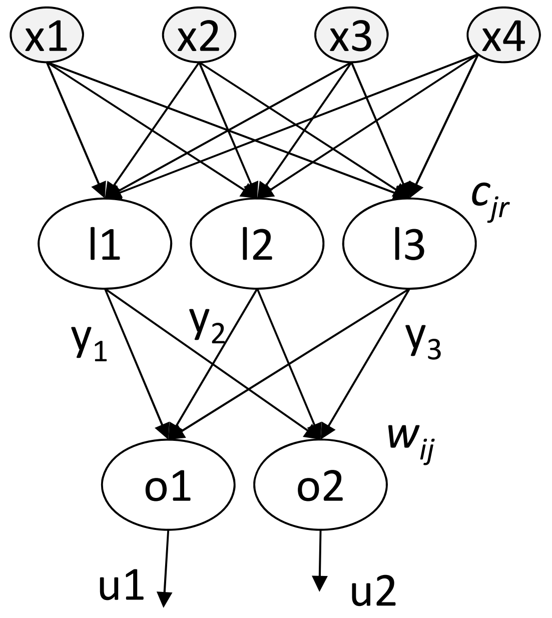
Example

- Assuming a two-layer network

- R: number of patterns
- x_r: input signals (1:M)
- l_j: neurons of the hidden layer with output y_j (1:L)
- c_{jr}: weights of the hidden layer
- o_i: neurons of the output layer with output u_i (1:N)
- w_{ij}: weights of the output layer

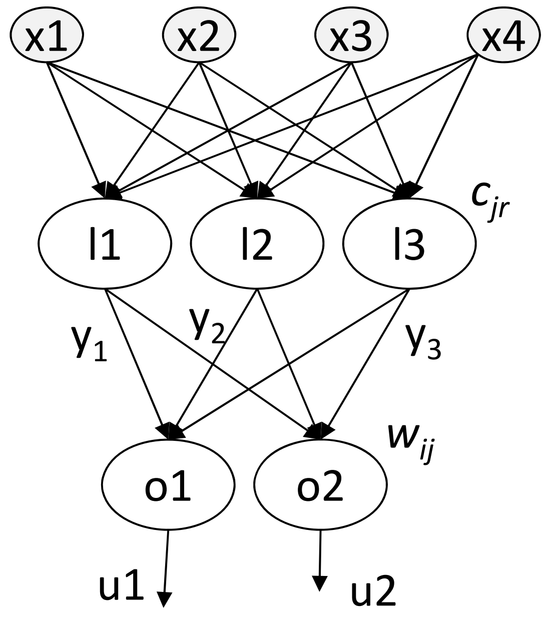


Back-propagation



- Each iteration involves two steps:
  1. Forward step
     - Presentation to the input of the k-th pattern
     - Compute the output signals for output and hidden units (weights are fixed to initial value)

Back-propagation



- Each iteration involves two steps:
  1. Forward step
     - Presentation to the input of the *k*-th pattern
     - Compute the output signals for output and hidden units (weights are fixed to initial value)
  2. Backward step
     - Update the weights w_{ij} using delta rule

$$\Delta w_{ij} = \eta \sum_{k=1}^{R} \left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) y_j^{(k)}$$

$$\downarrow$$

$$P_i^{(k)} = \sum_j w_{ij} y_j^{(k)}$$

Back-propagation

- Each iteration involves two steps:
  1. Forward step
     - Presentation to the input of the *k*-th pattern
     - Compute the output signals for output and hidden units (weights are fixed to initial value)
  2. Backward step
     - Update the weights w_{ij} using delta rule

$$\Delta w_{ij} = \eta \sum_{k=1}^{R} \left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) y_j^{(k)}$$

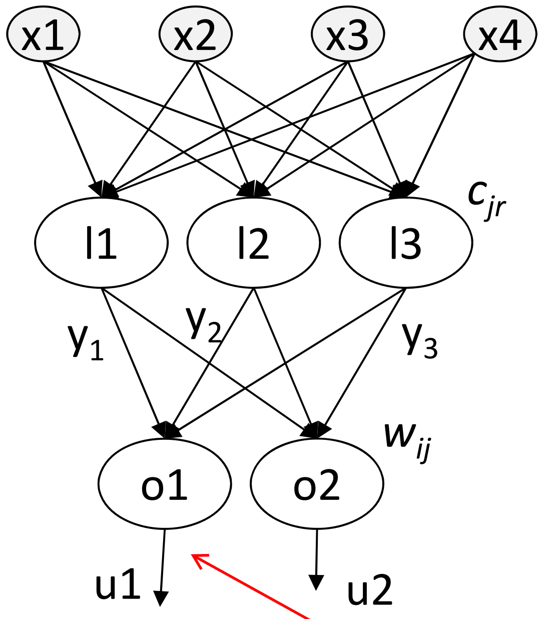
neuron 1 has 3 weights plus the threshold

$$\Delta w_{11} = \eta \sum_{k=1}^{R} \left(t_1^{(k)} - u_1^{(k)}\right) f'\left(P_1^{(k)}\right) y_1^{(k)} \qquad \Delta w_{21} = \eta \sum_{k=1}^{R} \left(t_2^{(k)} - u_2^{(k)}\right) f'\left(P_2^{(k)}\right) y_1^{(k)}$$

$$\Delta w_{12} = \eta \sum_{k=1}^{R} \left(t_1^{(k)} - u_1^{(k)}\right) f'\left(P_1^{(k)}\right) y_2^{(k)} \qquad \Delta w_{22} = \eta \sum_{k=1}^{R} \left(t_2^{(k)} - u_2^{(k)}\right) f'\left(P_2^{(k)}\right) y_2^{(k)}$$

$$\Delta w_{13} = \eta \sum_{k=1}^{R} \left(t_1^{(k)} - u_1^{(k)}\right) f'\left(P_1^{(k)}\right) y_3^{(k)} \qquad \Delta w_{23} = \eta \sum_{k=1}^{R} \left(t_2^{(k)} - u_2^{(k)}\right) f'\left(P_2^{(k)}\right) y_3^{(k)}$$

(Here it's not included the threshold b/c it was not fitting in the slide)

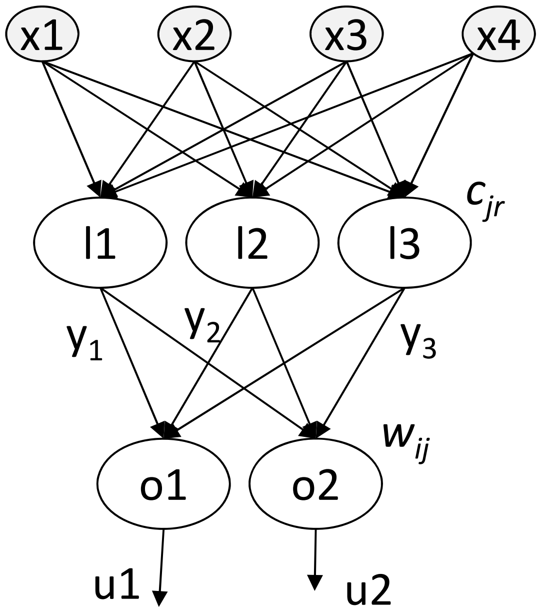


neuron one

3 input
2 output

Back-propagation

- Each iteration involves two steps:
  1. Forward step
     - Presentation to the input of the *k*-th pattern
     - Compute the output signals for output and hidden units (weights are fixed to initial value)
  2. Backward step
     - Update the weights w_{ij} using delta rule
     - Update the weights c_{jr} using delta rule



$$\Delta w_{ij} = \eta \sum_{k=1}^{R} \left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) y_j^{(k)}$$

$$\Delta c_{jr} = \eta \sum_{k=1}^{R} \left(\overline{y}_j^{(k)} - y_j^{(k)}\right) f'\left(P_j^{(k)}\right) x_r^{(k)}$$

→ desired output of the j-th hidden neuron

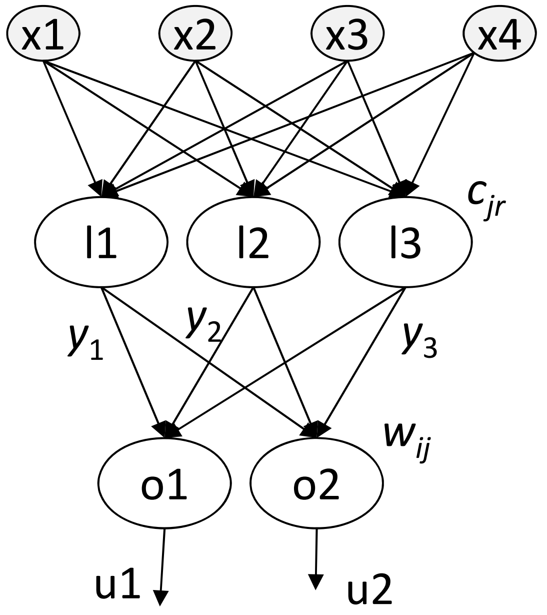
This is missing, we don't have a reference

Hidden (unknown)

$$\Delta c_{jr} = \eta \sum_{k=1}^{R} \left(\overline{y}_j^{(k)} - y_j^{(k)} \right) f'\left(P_j^{(k)}\right) x_r^{(k)}$$

this is known

$$P_j^{(k)} = \sum_r c_{jr} x_r^{(k)}$$



4 input
3 output

each neuron has 3 weights + 1 threshold

$$\Delta c_{11} = \eta \sum_{k=1}^{R} \left(\overline{y}_1^{(k)} - y_1^{(k)} \right) f'\left(P_1^{(k)}\right) x_1^{(k)} \qquad \Delta c_{21} = \eta \sum_{k=1}^{R} \left(\overline{y}_2^{(k)} - y_2^{(k)} \right) f'\left(P_2^{(k)}\right) x_1^{(k)} \qquad \Delta c_{31} = \eta \sum_{k=1}^{R} \left(\overline{y}_3^{(k)} - y_3^{(k)} \right) f'\left(P_3^{(k)}\right) x_1^{(k)}$$

$$\Delta c_{12} = \eta \sum_{k=1}^{R} \left(\overline{y}_1^{(k)} - y_1^{(k)} \right) f'\left(P_1^{(k)}\right) x_2^{(k)} \qquad \Delta c_{22} = \eta \sum_{k=1}^{R} \left(\overline{y}_2^{(k)} - y_2^{(k)} \right) f'\left(P_2^{(k)}\right) x_2^{(k)} \qquad \Delta c_{32} = \eta \sum_{k=1}^{R} \left(\overline{y}_3^{(k)} - y_3^{(k)} \right) f'\left(P_3^{(k)}\right) x_2^{(k)}$$

$$\Delta c_{13} = \eta \sum_{k=1}^{R} \left(\overline{y}_1^{(k)} - y_1^{(k)} \right) f'\left(P_1^{(k)}\right) x_3^{(k)} \qquad \Delta c_{23} = \eta \sum_{k=1}^{R} \left(\overline{y}_2^{(k)} - y_2^{(k)} \right) f'\left(P_2^{(k)}\right) x_3^{(k)} \qquad \Delta c_{33} = \eta \sum_{k=1}^{R} \left(\overline{y}_3^{(k)} - y_3^{(k)} \right) f'\left(P_3^{(k)}\right) x_3^{(k)}$$

$$\Delta c_{14} = \eta \sum_{k=1}^{R} \left(\overline{y}_1^{(k)} - y_1^{(k)} \right) f'\left(P_1^{(k)}\right) x_4^{(k)} \qquad \Delta c_{24} = \eta \sum_{k=1}^{R} \left(\overline{y}_2^{(k)} - y_2^{(k)} \right) f'\left(P_2^{(k)}\right) x_4^{(k)} \qquad \Delta c_{34} = \eta \sum_{k=1}^{R} \left(\overline{y}_3^{(k)} - y_3^{(k)} \right) f'\left(P_3^{(k)}\right) x_4^{(k)}$$

We did not solve the issue of the delta rule, this is just a trick to learn, all the issues: accuracy, learning rate etc. all remains

Single pattern

Action potential of the hidden neurons

Action potential of the output neurons

$$\Delta c_{jr} = f\left(\frac{\partial E}{\partial c_{jr}}\right) =$$

$$\frac{\partial E}{\partial c_{jr}} = \frac{\partial E}{\partial P_j^H}\frac{\partial P_j^H}{\partial c_{jr}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial P_j^H}\frac{\partial P_j^H}{\partial c_{jr}} = \sum_i^N\left(\frac{\partial E}{\partial P_i^O}\frac{\partial P_i^O}{\partial y_j}\right)\frac{\partial y_j}{\partial P_j^H}\frac{\partial P_j^H}{\partial c_{jr}} =$$

r is accounting the input

output of the hidden layer

because the output of *the* hidden layer is entering all the output neurons. The effect of this one output is acting on both neuron 1 and 2 of the output, so thus I need a sum

$$\underline{\sum_{i=1}^N\left(\frac{\partial E}{\partial u_i}\frac{\partial u_i}{\partial P_i^O}\frac{\partial P_i^O}{\partial y_j}\right)}\frac{\partial y_j}{\partial P_j^H}\frac{\partial P_j^H}{\partial c_{jr}} = \sum_{i=1}^N\left(\left(t_i - u_i\right)f'\left(P_i^O\right)w_{ij}\right)\frac{\partial y_j}{\partial P_j^H}\frac{\partial P_j^H}{\partial c_{jr}} =$$

output of the i-th neuron of the output layer

This is known from the beginning, wstart. If I'm using online updating I have already update the weights. In this case here I'll use the NEW values of weights

e * f' * x

$$\sum_{i=1}^N\left(\underline{\left(t_i - u_i\right)f'\left(P_i^O\right)w_{ij}}\right)f'\left(P_j^H\right)x_r$$

This contribution is the correction done on the output neurons.. the sort of error that you expect in the delta rule: (t-u) *f'*x. So somehow this is what we get.

it's calculated on Pjh

You still have the input

For batch updating with R patterns

$$\Delta c_{jr} = \eta \sum_{k=1}^{R} \left(\overline{y}_j^{(k)} - y_j^{(k)} \right) f'\left(P_j^{H(k)}\right) x_r^{(k)} =$$

$$\eta \sum_{k=1}^{R} \left(\sum_{i=1}^{N} \left(\left(t_i^{(k)} - u_i^{(k)} \right) f'\left(P_i^{O(k)}\right) w_{ij} \right) f'\left(P_j^{H(k)}\right) x_r^{(k)} \right)$$

$$errO = \left(t_i - u_i\right) * f'\left(P_i^O\right)$$

$$errH_{ij} = f'\left(P_j^H\right) * w_{ij} * errO_i$$

here you have to consider all the contributions

I can feed one pattern, compute the delta and keep it. This delta is for both layer. Then I feed another pattern, but the weights are not yet updated, and I have another contribution to the deltaW.

$$P_i^{O(k)} = \sum_j w_{ij} y_j^{(k)} \qquad P_j^{H(k)} = \sum_r c_{jr} x_r^{(k)}$$

I'm assuming more than 1 hidden layer, btw the first is a virtual one

Generalized back-propagation

"don't worry about maths"

here 1 is the first hidden layer, I don't consider the input layer

$w_{ij}^{(l)}$ with *l*:1,....*L*

l is the index of the layer , j the neuron of that layer, i the neuron of the layer before

$$\Delta w_{ij}^{(l)} = \eta \sum_{k=1}^{R} \delta_i^{(l),(k)} e_j^{(l-1),(k)}$$

$e_j^{(l-1),(k)}$ is the output signal of the j-th neuron of the (l-1)-th layer in correspondence of the k-th pattern of the training set

$$\delta_i^{(l),(k)} = \left(t_i^{(k)} - u_i^{(k)}\right) f'\left(P_i^{(k)}\right) \quad if \;\; l = L$$

output layer: you have exactly the delta rule

$$\delta_i^{(l),(k)} = f'\left(P_i^{(k)}\right) \sum_{r=1}^{M_{l+1}} \left(\delta_r^{(l+1),(k)} w_{ri}^{(l+1)}\right) \qquad if \;\; l < L$$

M_{l+1} is the number of neurons in the (*l*+1)-th layer

Procedure

Start from l=L (output layer) and compute $\delta_i^{(L),(k)}$

Move to l=L-1 and compute $\delta_i^{(L-1),(k)}$ in function of $\delta_i^{(L),(k)}$

Continue until l=2 $\delta_i^{(2),(k)}$

Update all the weights $\Delta w_{ij}^{(l)} = \eta \sum_{k=1}^{R} \delta_i^{(l),(k)} e_j^{(l-1),(k)}$

Remarks and issues on Multi-layer ANN (1)

- Structural issue
  - In theory hidden layers allow solving any classification task but…
    - Linear activation prevents general classification properties
      - The combination of hyper-planes generated by the internal neurons divides the input space in closed and partially-closed regions characterized by irregular linear bounds
    - Non-Linear activation functions to model non-linear decision boundaries
    - Feedforward NN with a single hidden layer of sigmoidal units are capable of approximating uniformly any continuous multivariate function, to any degree of accuracy (Hornik et al., 1989, " Multilayer feedforward networks are universal approximators" Neural Networks 2(5), 359-366).

no?

Remarks and issues on Multi-layer ANN (2)

- Information representation
  - Local
    - One single unit in the hidden layers activates in correspondence of a specific pattern
  - Super-local
    - One single unit in the hidden layer is connected to a subset of input neurons (the unit is coding a subset of the input space)
  - Distributed
    - One single input activates a subset of hidden layer
  - Super-distributed
    - A subset of input neurons are connected to a subset of hidden neurons

Remarks and issues on Multi-layer ANN (3)

- Training
  - The shape of E for multi-layer networks is complex as it is determined by the sequence of non-linear operations during the computation of the activation functions
    - The surface error is not smooth as interlayer weights are multiplied one with each other (Local minima)
  - It is not easy to establish the optimal value of the **learning rate** (it is dependent on the error function topology)
    - Small values of η imply good approximation but slow convergence
    - Large values of η imply fast convergence but possible either local minima or oscillations (see Perceptron lecture)

I have to do a sort of tuning, this means I have to define first the topology of the network, then I can train the network, but it can happen I'm not satisfied with the result no?

Remarks and issues on Multi-layer ANN (4)

- Weight updating in back-propagation
  - The updating of the weights can be made less sensitive to rapid variations of the error signal

$$\Delta w_{ij}^{(l)} = \eta \sum_{k=1}^{R} \delta_i^{(l),(k)} e_j^{(l-1),(k)} + \alpha \Delta' w_{ij}^{(l)}$$

$\alpha \Delta' w_{ij}^{(l)}$ The factor α ($0<\alpha<1$) multiplies the updating of the synaptic weights obtained at the previous steps (momentum)

Sort of memory effect

Practical considerations

Training

1. How training should be efficiently carried out?
2. How many pattern in the training set?
3. How to select initial weights?
4. How to select the value of the learning rate?
5. The update of the weights must be performed after each training (on-line mode) or after the presentation of the whole training set (batch mode)?
6. When we should stop the training process?
7. How to avoid local minima and smooth zones of the error function?

Structure

1. How many hidden layers? Issue of the vanishing gradient
2. How many neurons for each layer?
3. . Which are the best activation functions?

There is no standard methodology to determinate these values. Even there is some heuristic points, final values are determinate by a trial and error procedure.

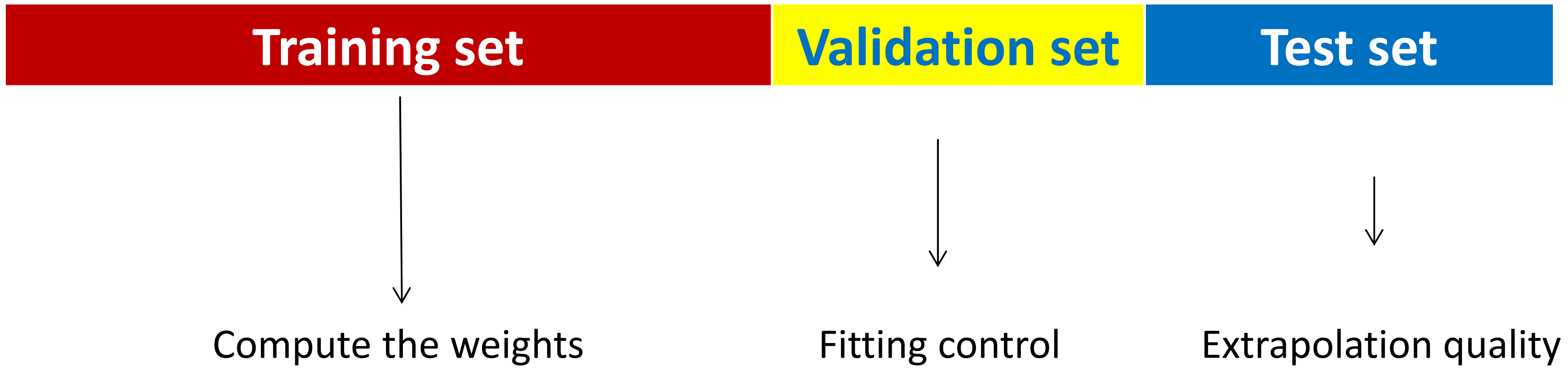
[10.37]

Training set

- Training data should be representative
  - it should not contain too many examples of one type at the expense of another
  - if one class of pattern is easy to learn, having large numbers of patterns from that class in the training set will only slow down the over-all learning process
  - In case of continuous input data
    - rescale the input values (zero mean and std normalization)
- Weights initialized randomly in a small range about zero
  - Sigmoidal function can easily saturate for great values of the weights
- Batch mode All training set in bundle, this allows to smooth out noise in the dataset
  - Small pattern errors can be smoothed out
- On-line mode The increment is more sensible to each pattern!
  - More sensitive to pattern errors
  - But … random pattern presentation (shuffle the order of the training data each epoch) makes the search in the weight space more stochastic
    - This reduces the probability to be trapped in local minima

3 classes in training dataset made from the overall set. These are percentage of the overall dataset. When the training has finished I can measure a sort of extrapolation error on the test set. The error is training to minimize the error on the validation, but it's using the other set to compute the weights, then I have the overall extrapolation quality.

I could have 3 error (1 for each subset)



Usually the implementation selects randomly the subsets, this is a way to try minimize the bias. I could also a priory define the subsets.

Learning and generalization

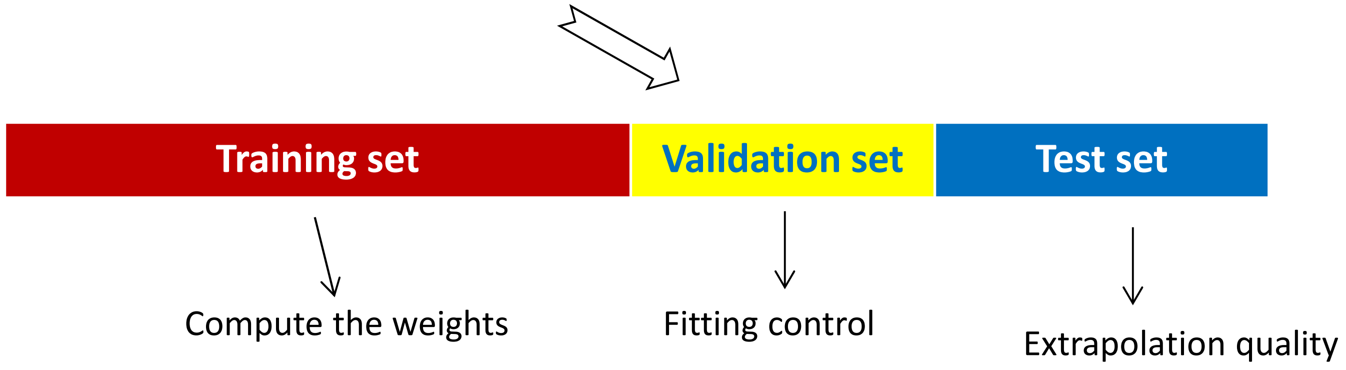
Empower generalization properties

- Use of the network with patterns (Test set) not being included in the training set
- The network has good generalization capabilities if its performance on the Test set is similar to that one obtained on the training set
  - Small residual error on the training set does not guarantee good generalization properties
  - Validation during training on a set different from that one used for compute weights
  - The available pattern set is partitioned in **training** and **validation (usually 10-20%)** sets
  - The training is performed only on training set and evaluated both on the training set and validation set
  - The training is stopped when the error on the validation set overcomes a predefined **threshold**

Also neural networks suffers from fitting issues: underfitting or overfitting. I want a tradeoff between the two conditions, If I'm using 100% of the trainingset for the computation of the weights I'm probably very fitting on these patterns, I could suffer for overfitting, evaluating the network for different patterns the result would be bad.
Using a very small number of patterns I may be an underfitting. There's no analitic rule on how to separate the sets, in NN toolbox by matlab there's a default which is like 70 20 10 %, but ofcours you can set them.

Fitting/Overfitting it's also dependant on the topology: many neurons could lead to overfitting, small number of neurons underfitting. Each neuron you have can be seen as an additional NL you have, an additional order of your system.

• Fitting against extrapolation error

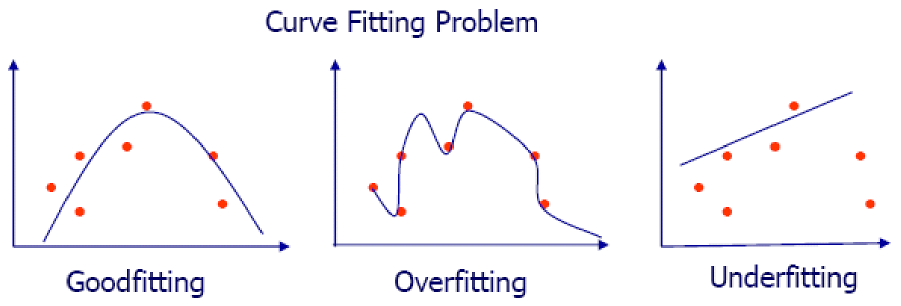


To prevent under-fitting

- The network must have a sufficient number of hidden units
- Convergence threshold

To prevent over-fitting

- Avoid too much layers and units
- Additional noise superimposed to the training patterns
- The training can be stopped before convergence



Synthesis

Statistics	FFNN
model	network
estimation	learning
regression	supervised learning
interpolation	generalization
observation	training set
parameters	synaptic weights
independent variables	input
dependent variables	output