Introduction to (shallow) Neural Networks

Pr. Fabien MOUTARDE
Center for Robotics
MINES ParisTech
PSL Université Paris

Fabien.Moutarde@mines-paristech.fr

http://people.mines-paristech.fr/fabien.moutarde

Neural Networks: from biology to engineering

- Understanding and modelling of brain
- Imitation to reproduce high-level functions
- Mathematical tool for engineers

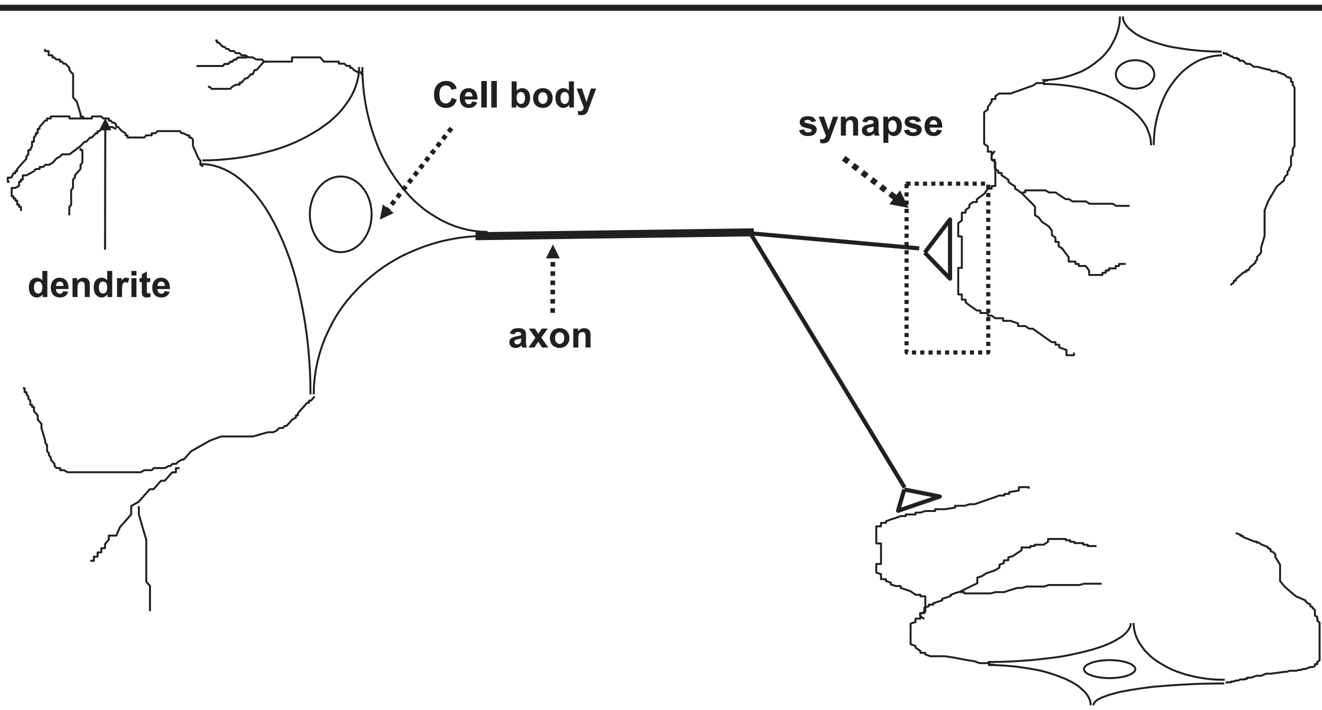
Application domains

Modelling any input-output function by "learning" from examples:

- Pattern recognition
- Voice recognition
- Classification, diagnosis

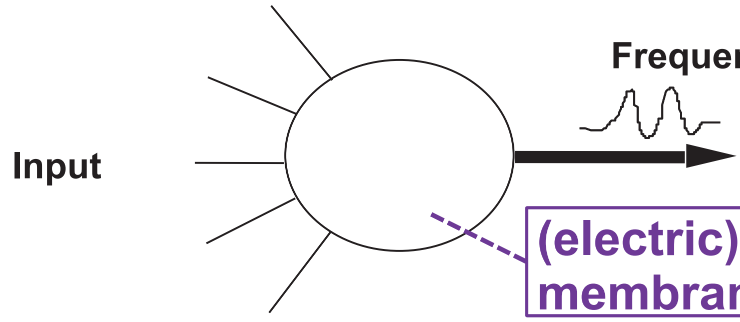
- Identification
- Forecasting
- Control, regulation

Biological neurons

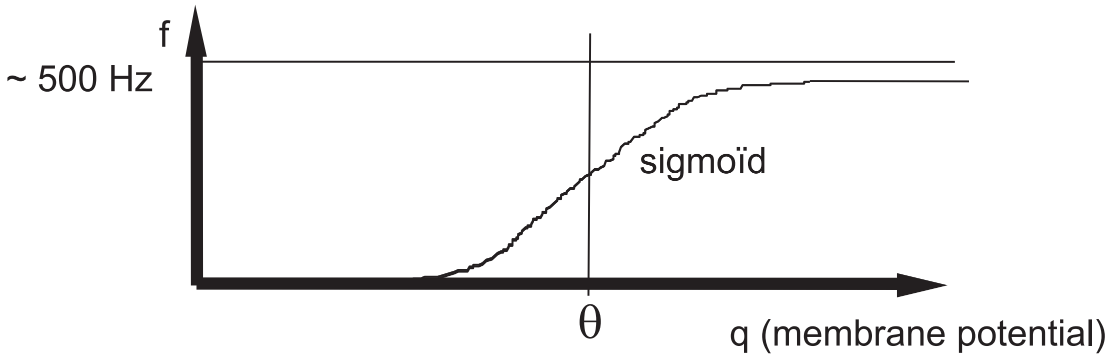


- Electric signal: dendrites → cell body →> axon →synapses

Empirical model of *neurons*



(electric) potential of membrane $q \approx \sum_i C_i f_i$

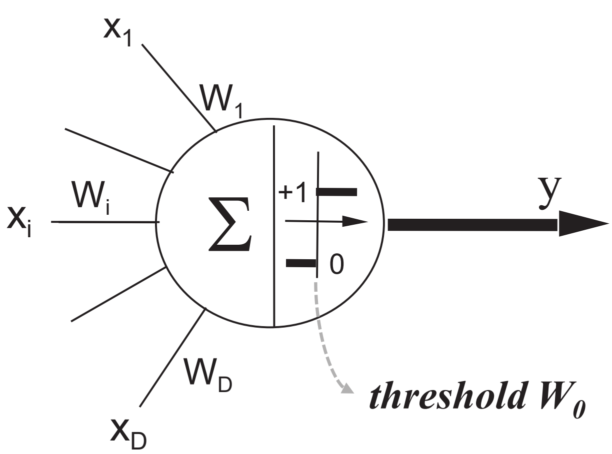


➔ **Neuron output = periodic signal with frequency f ≈ sigmoid(q) = sigmoid($\sum_i C_i f_i$)**

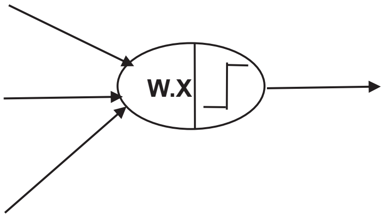
"Birth" of *formal* Neuron

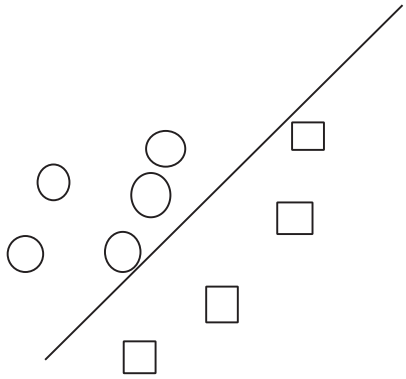
- **Mc Culloch & Pitts (1943)**

  - Simple model of neuron
  - goal: model the brain



Linear separation by a single neuron

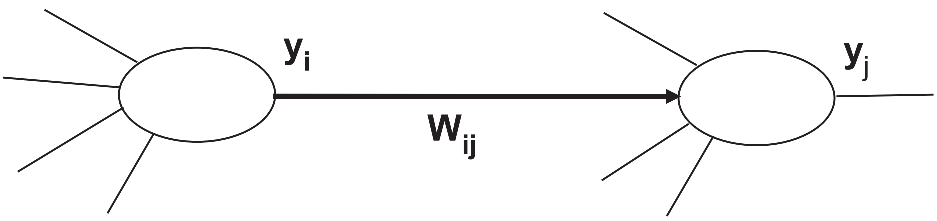




Theoretical model for *learning*

- **Hebb rule (1949)**

"*Cells that fire together wire together*", ie synaptic weight increases between neurons that activate *simultaneously*



$$W_{ij}(t+dt) = W_{ij}(t) + \lambda y_i(t) y_j(t)$$

First formal Neural *Networks*

(en français : Réseaux de Neurones)

- **PERCEPTRON (Rosenblatt, 1957)**
- **ADALINE (Widrow, 1962)**

Formal neuron of Mac Culloch & Pitts

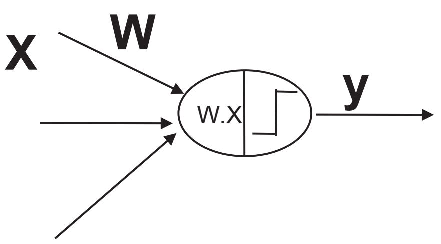
+

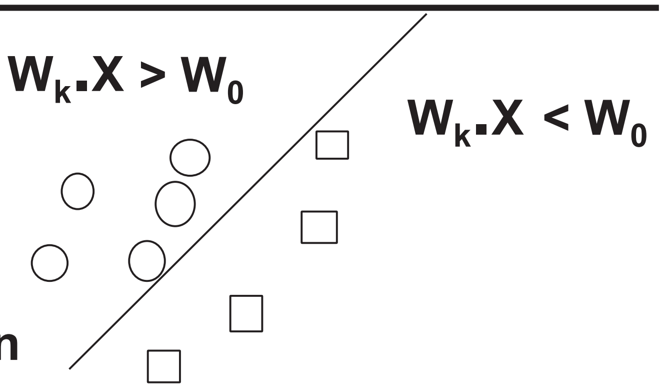
Hebb rule for learning

↓

Possible to "learn" Boolean functions by training from examples

Training of Perceptron





➔ **<u>Linear</u> separation**

Training algorithm:

$W_{k+1} = W_k + vX$ if X incorrectly classified (v: target value)

$W_{k+1} = W_k$ if X correctly classified

- Convergence if linearly-separable problem
- ?? if NOT linearly-separable

Limits of first models, necessity of "hidden" neurons

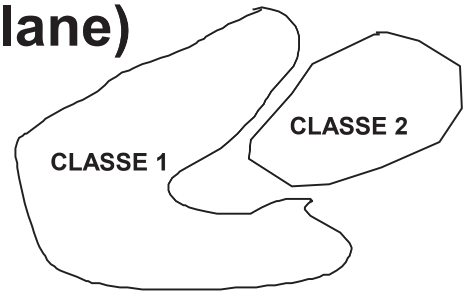
- **PERCEPTRONS, book by Minsky & Papert (1969)**

  Detailed study on Perceptrons and their intrinsic limits:

  - can NOT learn some types of Boolean functions (even simple one like XOR)

  - can do ONLY LINEAR separations

But many classes cannot be linearly-separated (by a single hyper-plane)



➔ Necessity of *__several__* layers in the Neural Network

➔ Requires new training algorithm

1^{st} revival of Neural Nets

USE OF DERIVABLE NEURONS

+

APPLY GRADIENT DESCENT METHOD

- **__GRADIENT BACK-PROPAGATION__ (Rumelhart 1986, Le Cun 85)**
  (en français : *Rétro-propagation du gradient*)

→ Overcome Credit Assignment Problem by training Neural Networks with HIDDEN layers

- **Empirical solutions for MANY real-world applications**

- **Some strong theoretical results:**
  Multi-Layer Perceptrons are UNIVERSAL (and parsimonious) approximators

- **around years 2000's: still used, but much less popular than SVMs and boosting**

2^{nd} recent « revival »: Deep-Learning

- Since ~2006, rising interest for, and excellent results with *<u>"deep"</u> neural networks*, consisting in MANY layers:
  - Unsupervised "intelligent" initialization of weights
  - Standard gradient descent, and/or fine-tuning from initial values of weights
  - Hidden layers ➔ learnt hierarchy of features
- In particular, since ~2013 dramatic progresses in visual recognition (and voice recognition), with deep <u>*Convolutional* Neural Networks</u>

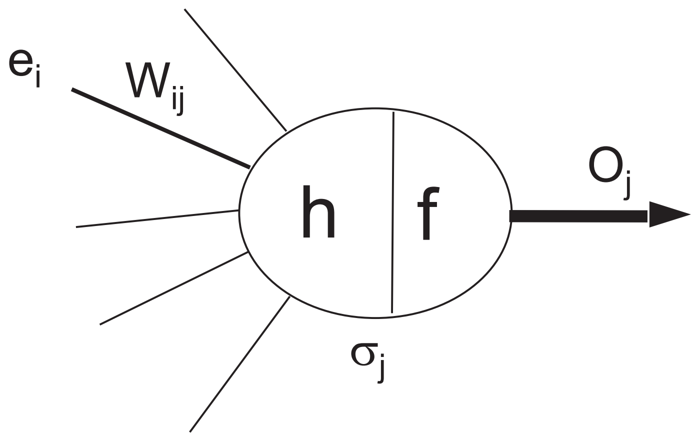
What is a FORMAL neuron?

DEFINITIONS OF FORMAL NEURONS

<u>In general:</u> a processing "unit" applying a simple operation to its inputs, and which can be "connected" to others to build a networks able to realize any input-output function

<u>"Usual" definition:</u> a "unit" computing a weighted sum of its inputs, and then applying some non-linearity (sigmoïd, ReLU, Gaussian, …)

General formal neuron



e_i: inputs of neuron
σ_j: potential of neuron
O_j: output of neuron

W_{ij}: (synaptic) weights
h: input function (computation of
potential = Σ, dist, kernel, …)
f: activation (or transfer) function

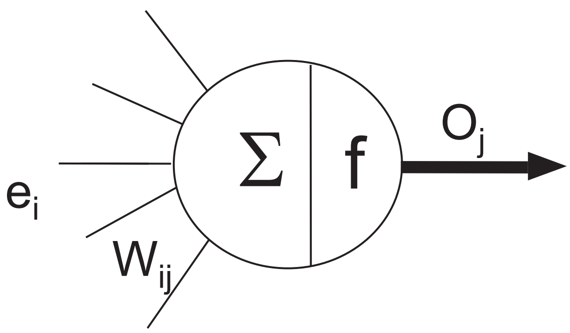
$$\sigma_j = h\left(e_i, \{W_{ij},\ i=0 \text{ à } k_j\}\right)$$

$$O_j = f(\sigma_j)$$

The combination of particular h and f functions defines the *type* of formal neuron

Summating artificial "neurons"

PRINCIPLE

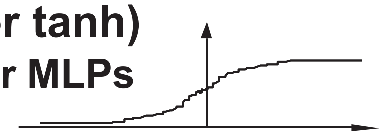


$$O_j = f\left(W_{0j} + \sum_{i=1}^{n_j} W_{ij} e_i\right)$$

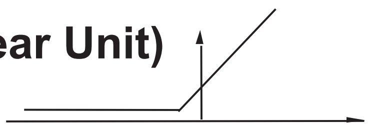
W_{0j} = "bias"

<u>ACTIVATION FUNCTIONS</u>

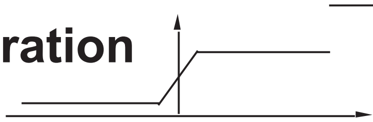
- **Threshold (Heaviside or sign)**
  → ***<u>binary</u>*** **neurons**
- **Sigmoïd (logistic or tanh)**
  → **most common for MLPs**



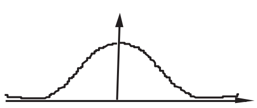
- **Identity** → ***<u>linear</u>*** **neurons**
- **ReLU (Rectified Linear Unit)**



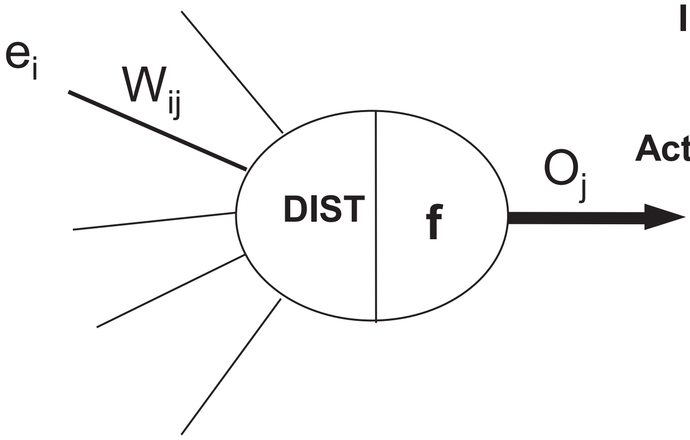
- **Saturation**



- **Gaussian**



"Distance" formal neurons

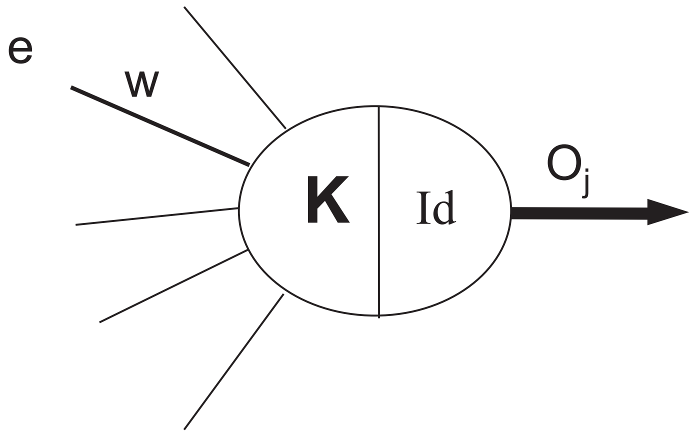


Input function: $h\left(\begin{pmatrix} e_1 \\ \dots \\ e_k \end{pmatrix}, \begin{pmatrix} W_{1j} \\ \dots \\ W_{kj} \end{pmatrix}\right) = \sum_i (e_i - W_{ij})^2$

Activation function = Identity of Gaussian

The potential of these neurons is the Euclidian DISTANCE between input vector $(e_i)_i$ and weight vector $(W_{ij})_i$

Kernel-type formal neurons



Input function: $h(e,w) = K(e,w)$

with K symmetric and "positive" in Mercer sense: $\forall \psi$ **tq** $\int \psi^2(x)dx < \infty$, $\int K(u,v)\psi(u)\psi(v)dudv \geq 0$

Activation function = Identity

Examples of possible kernels:

- **Polynomial:** `K(u,v)=[u.v + 1]`^{p}
- **Radial Basis Function:** `K(u,v)=exp(-||u-v||`^{2} `/ 2`σ^2`)`
  - ➔ ***equivalent to distance-neuron+gaussian-activation***
- **Sigmoïd:** `K(u,v)=tanh(u.v+`θ`)`
  - ➔ ***equivalent to summating-neurons+sigmoïd-activation***

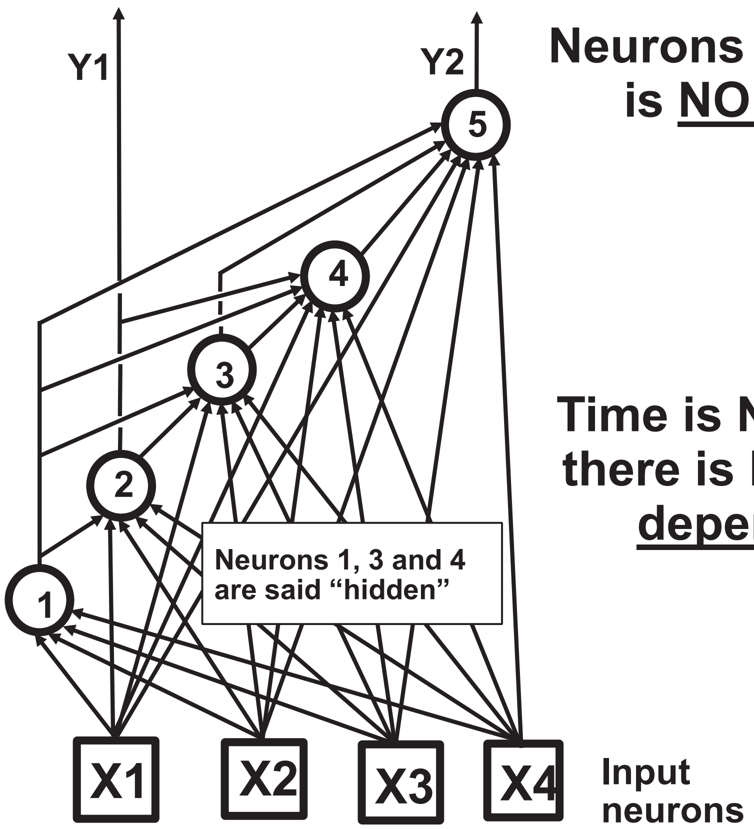
TWO FAMILIES OF NETWORKS

- **FEED-FORWARD NETWORKS**
  (en français, "*réseaux non bouclés*"):
  NO feedback connection,
  The output depends only on current input (NO memory)

- **FEEDBACK OR RECURRENT NETWORKS**
  (en français, "réseaux *bouclés*"):
  Some internal feedback/backwards connection
  ➔ output depends on current input
  AND ON ALL PREVIOUS INPUTS (some memory inside!)

Feed-forward networks

(en français : réseau "NON-bouclé")

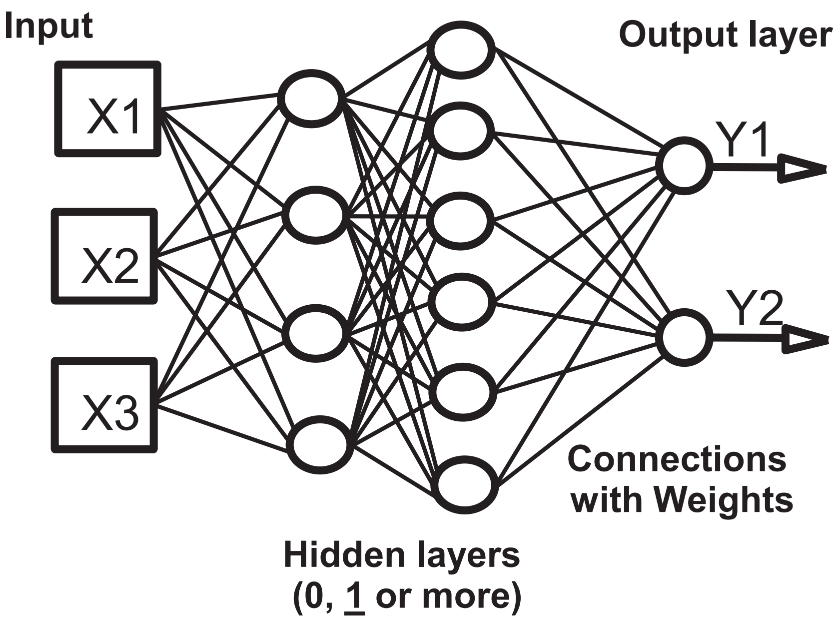


Neurons can be ordered so that there is <u>NO "backwards" connection</u>

↓

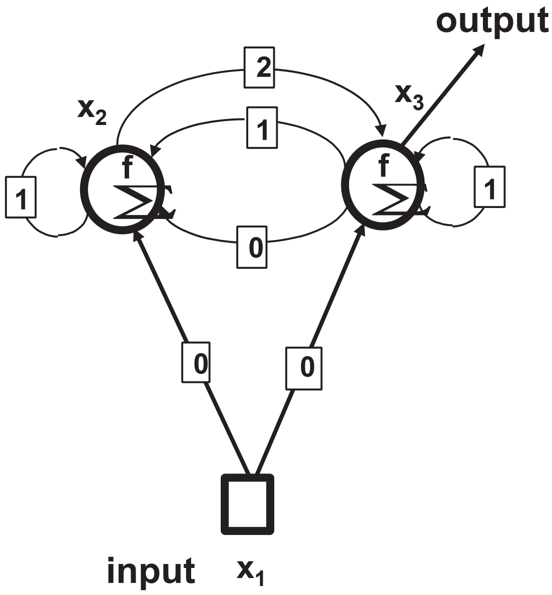
Time is NOT a functional variable, i.e. there is NO MEMORY, and <u>the output depends only on current input</u>

Feed-forward Multi-layer Neural Networks

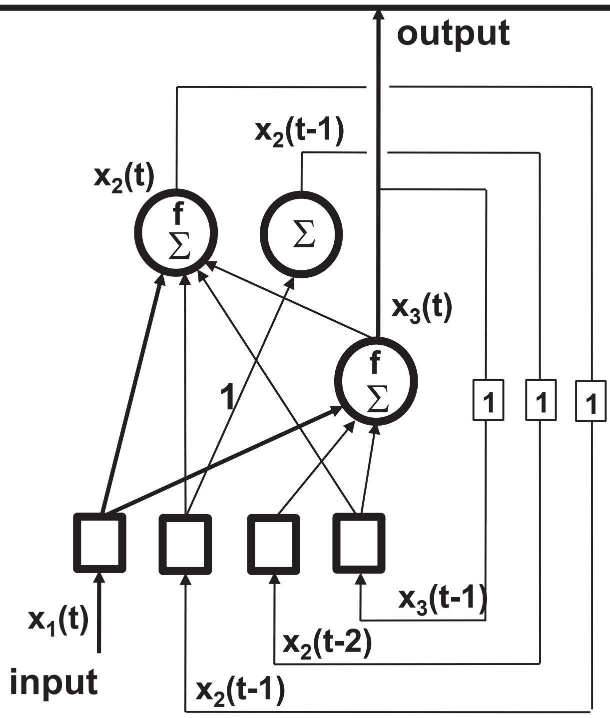


For "<u>Multi-Layer Perceptron" (MLP)</u>,
neurons type generally "summating with sigmoid activation"
[terme français pour MLP : "<u>Réseau Neuronal à couches</u>"]

Recurrent Neural Networks

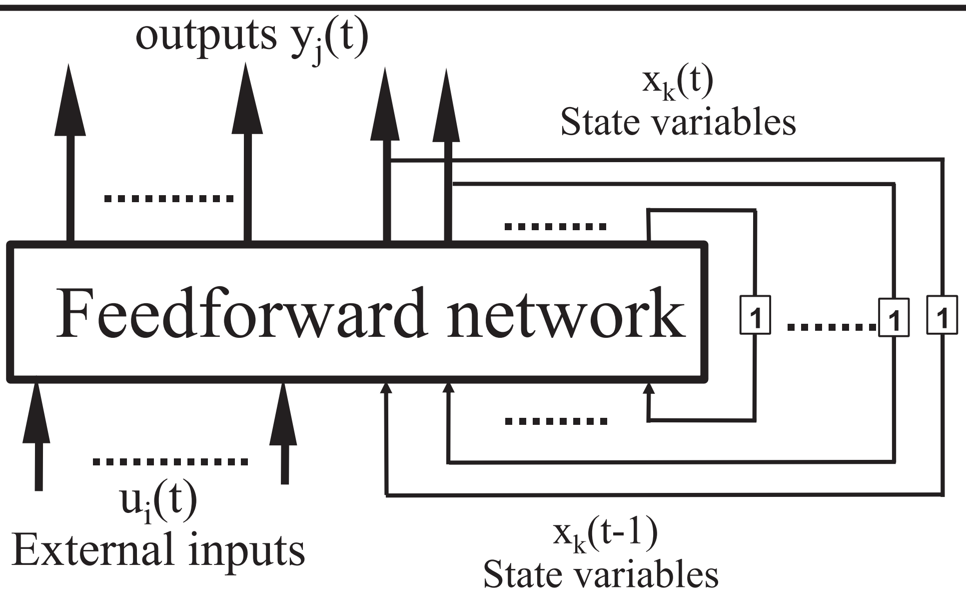


A time-delay is associated to each connection



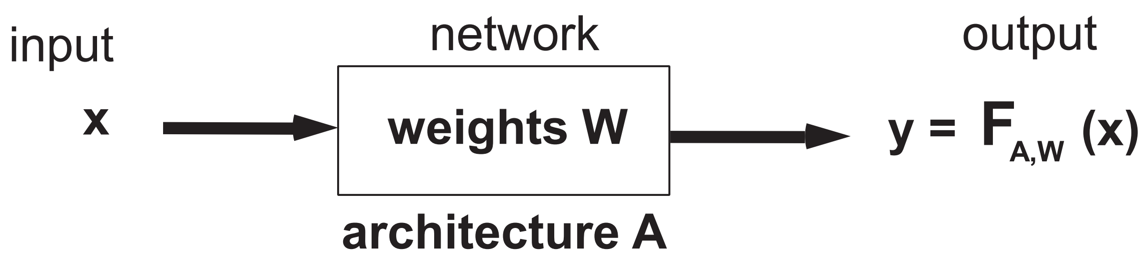
Equivalent form

Canonical form of Recurrent Neural Networks



The output at time t depend not only on external inputs U(t), but also (via internal "state variables") on the whole sequence of previous inputs (and on initialization of state variables)

Use of a Neural Network



- **Two modes:**
  - **<u>training:</u> based on examples of (input,output) couples, the network modifies**
    - **Its parameters W (synaptic weights of connections)**
    - **And also potentially its architecture A (by creating/eliminating neurons or connections)**
  - **<u>recognition:</u>**
    computation of output associated to a given input (architecture and weights remaining frozen)

Training principle for Neural Networks

- Supervised training = adaptation of synaptic weights of the network so that its output is close to target value for each example

- Given n examples (X_p; D_p), and the network outputs Yp=NN(Xp), the average quadratic error is

$$E(W) = \sum_p \left(Y_p - D_p\right)^2$$

Training ~ finding W* =ArgMin(E), ie minimize the cost function E(W)

- Generally this is done by using gradient descent (total, partial or stochastic):

$$W(t+1) = W(t) - \lambda \, grad_W(E)$$ [+ $\mu(t)(W(t)-W(t-1))$]

Usual training algo for Multi Layer Perceptrons (MLP)

- Training by <u>Stochastic Gradient Descent</u> (SGD), using *back-propagation*:
  - Input 1 (or a few) random training sample(s)
  - Propagate
  - Calculate error (loss)
  - Back-propagate through all layers from end to input, to compute gradient and update weights

Back-propagation principle

Smart method for efficient computing of gradient (w.r.t. weights) of a Neural Network cost function, based on chain rule for derivation.

Cost function is $Q(t) = \sum_m loss(Y_m, D_m)$, where m runs over training set examples

Usually, $loss(Y_m, D_m) = ||Y_m - D_m||^2$ [quadratic error]

Total gradient:

$$W(t+1) = W(t) - \lambda(t)\, grad_W(Q(t)) + \mu(t)(W(t)-W(t-1))$$

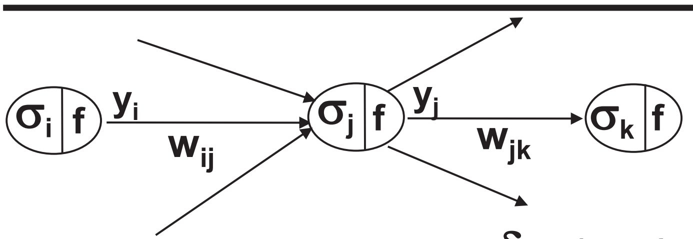
Stochastic gradient:

$$W(t+1) = W(t) - \lambda(t)\, grad_W(Q_m(t)) + \mu(t)(W(t)-W(t-1))$$

where $Q_m = loss(Y_m, D_m)$, is error computed on only ONE example randomly drawn from training set at every iteration and $\lambda(t)$ = *learning rate* (fixed, decreasing or adaptive), $\mu(t)$ = momentum

Now, how to compute dQ_m/dW_{ij}?

Backprop through fully-connected layers: use of chain rule derivative computation



$dE_m/dW_{ij} = (dE_m/d\sigma_j)(d\sigma_j/dW_{ij}) = (dE_m/d\sigma_j)\, y_i$

Let $\delta_j = (dE_m/d\sigma_j)$. Then $\boxed{W_{ij}(t+1) = W_{ij}(t) - \lambda(t)\, y_i\, \delta_j}$

(and $W_{0j}(t+1) = W_{0j}(t) - \lambda(t)\delta_j$)

If neuron j is output, $\delta_j = (dE_m/d\sigma_j) = (dE_m/dy_j)(dy_j/d\sigma_j)$ with $E_m = ||Y_m - D_m||^2$

so $\boxed{\delta_j = 2(y_j - D_j) f'(\sigma_j) \text{ if neuron j is an output}}$

Otherwise, $\delta_j = (dE_m/d\sigma_j) = \sum_k (dE_m/d\sigma_k)(d\sigma_k/d\sigma_j) = \sum_k \delta_k (d\sigma_k/d\sigma_j) = \sum_k \delta_k W_{jk}(dy_j/d\sigma_j)$

so $\boxed{\delta_j = (\sum_k W_{jk}\delta_k) f'(\sigma_j) \text{ if neuron j is "hidden"}}$

➔ all the δ_j can be computed successively from last layer to upstream layers by "error backpropagation" from output

Universal approximation theorem

Cybenko 1989

- For any continuous function F defined and bounded on a bounded set, and for any ε, there exists a layered Neural Network with ONLY ONE HIDDEN LAYER (of *sigmoïd* neurons) which approximates F with error < ε

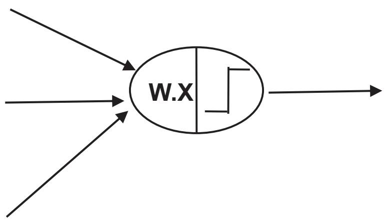
…But the theorem does not provide any clue about how to find this one_hidden-layer NN, nor about its size!
And the size of hidden layer might be huge…

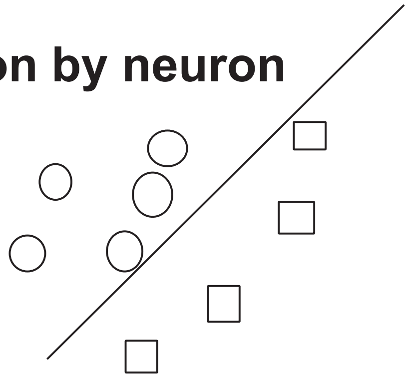
Sussman 92

- The set of MLPs with ONE hidden layer of sigmoid neurons is a family of PARCIMONIOUS approximators: for equal number of parameters, more functions can be correctly approximated than with polynoms

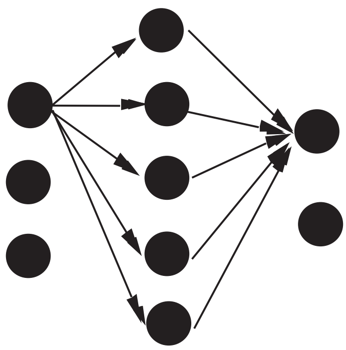
Multi-layer (MLP) v.s. single-layer (perceptron)

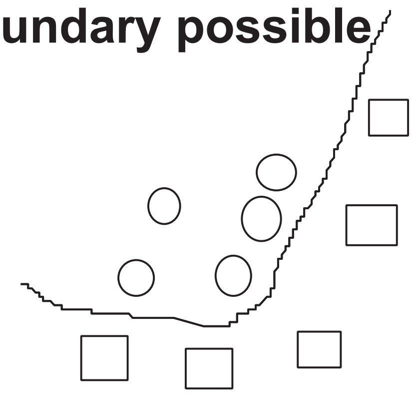
Single-layer → one linear separation by neuron





Multi-layer: any shape of boundary possible





Pros and cons of MLPs

ADVANTAGES

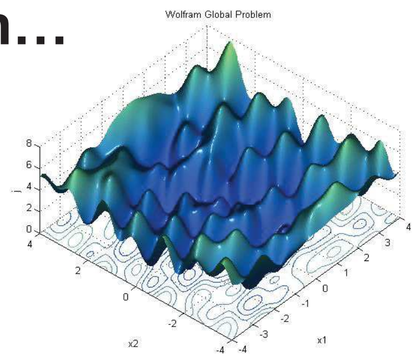
- **Universal and parsimonious approximators (& classifiers)**
- **Fast to compute**
- **Robustness to data noise**
- **Rather easy to train and program**

DRAWBACKS

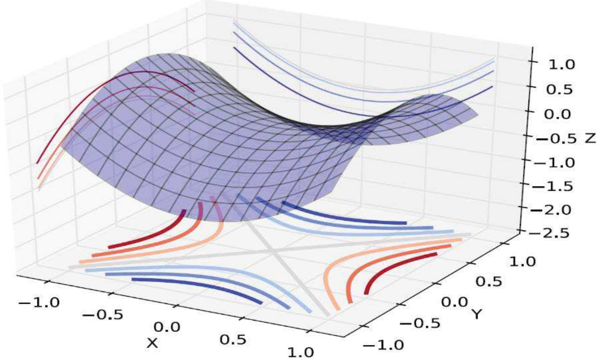
- ***Choice of ARCHITECTURE (# of neurons in hidden layer) is CRITICAL, and empiric!***
- **Many other critical hyper-parameters (learning rate, # of iterations, initialization of weights, etc…)**
- ***Many local minima in cost function***
- **Blackbox: difficult to interpret the model**

Why gradient descent works *despites non-convexity*?

- **Local minima dominate in low-Dim…**

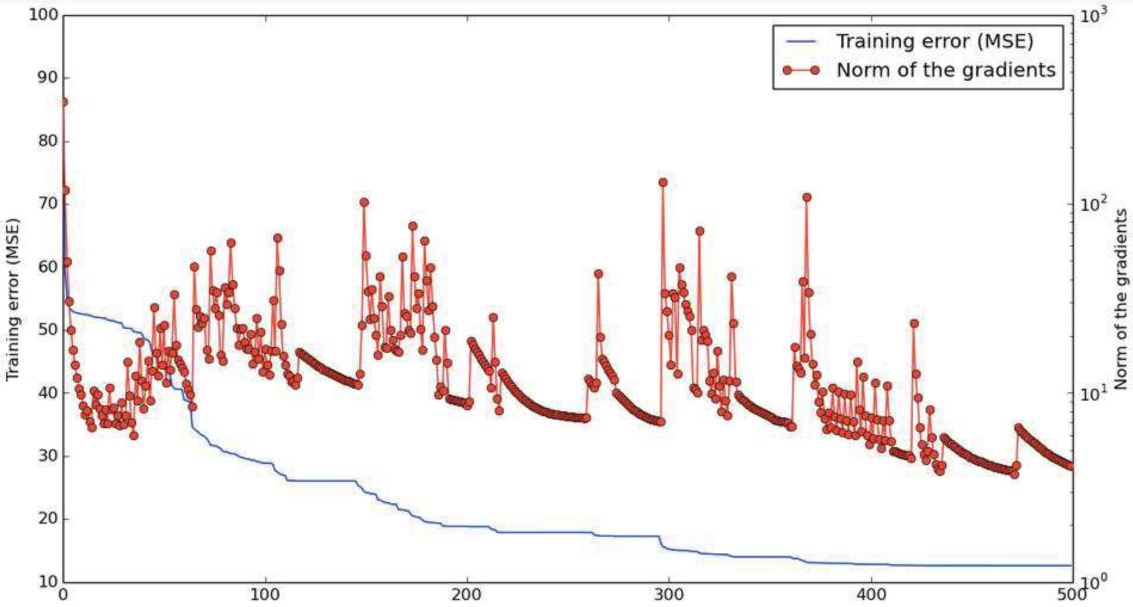


- **…but recent work has shown *saddle points* dominate in high-Dim**



- **Furthermore, most local minima are close to the global minimum**

Saddle points in training curves



- **Oscillating between two behaviors:**
  - **Slowly approaching a saddle point**
  - **Escaping it**

METHODOLOGY FOR SUPERVISED TRAINING OF MULTI-LAYER NEURAL NETWORKS

Training set vs. TEST set

- Space of possible input values usually infinite, and training set is only a FINITE subset

- Zero error on all training examples ≠ good results on whole space of possible inputs (cf generalization error ≠ empirical error…)

↓

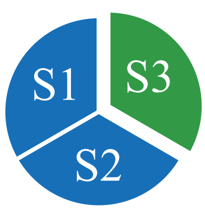
- Need to collect enough and representative examples
- Essential to keep aside a subset of examples that shall be used only as TEST SET for estimating final generalization (when training finished)

- Need also to use some "validation set" independant from training set, in order to tune all hyper-parameters (layer sizes, number of iterations, etc…)

Optimize hyper-parameters by "VALIDATION"

To avoid over-fitting and maximize generalization, absolutely ***essential to use some VALIDATION estimation,*** for optimizing training hyper-parameters (and stopping criterion):

- either use a *separate validation dataset* (random split of data into Training-set + Validation-set)
- or use ***CROSS-VALIDATION***:
  - Repeat k times: train on (k-1)/k proportion of data + estimate error on remaining 1/k portion
  - Average the k error estimations



3-fold cross-validation:

- Train on S1∪S2 then estimate errS3 error on S3
- Train on S1∪S3 then estimate errS2 error on S2
- Train on S2∪S3 then estimate errS1 error on S1
- Average validation error: (errS1+errS2+errS3)/3

Some Neural Networks training "tricks"

- Importance of input normalization

  (zero mean, unit variance)
- Importance of weights initialization

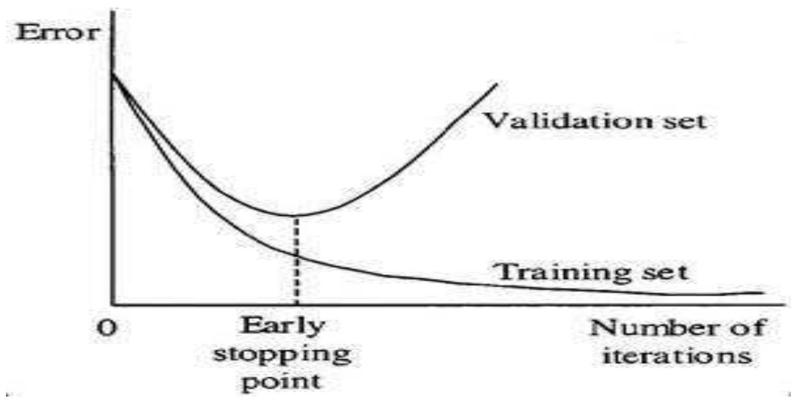
  random but SMALL and prop. to 1/sqrt(nbInputs)
- Decreasing (or adaptive) learning rate
- Importance of training set size

  If a Neural Net has a LARGE number of free parameters,

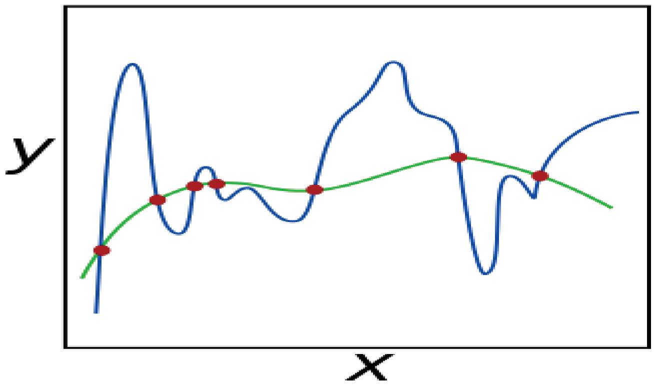
  ➔ train it with a sufficiently large training-set!
- Avoid overfitting by Early Stopping of training iterations
- Avoid overfitting by use of L1 or L2 regularization

Avoid overfitting by EARLY STOPPING

- For Neural Networks, a first method to avoid overfitting is to STOP LEARNING iterations as soon as the *validation*_error stops decreasing
- Generally, not a good idea to decide the number of iterations beforehand. Better to ALWAYS USE EARLY STOPPING



Avoid overfitting using regularization penalty (weight decay)



Trying to fit too many free parameters with not enough information can lead to overfitting

Regularization = penalizing too complex models
Often done by adding a special term to cost function

For neural network, the regularization term is just norm L2 or L1 of vector of all weights:

$$K = \Sigma_m(\text{loss}(Y_m,D_m)) + \beta\, \Sigma_{ij} |W_{ij}|^p \quad \text{with } p=2 \text{ (L2) or } p=1 \text{ (L1)}$$

→ name "Weight decay"

MLP hyper-parameters

- Number and sizes of hidden layers!!
- Activation functions
- Learning rate (& momentum) [optimizer]
- Number of gradient iterations!! (& early_stopping)
- Regularization factor
- Weight initialization

MLP tuning in practice

- Use 'adam' optimizer
- Test/compare *WIDELY VARIED* HIDDEN LAYER SIZES
  `(typically 30;100;300;1000;30-30;100-100)`
- Test/compare *SEVERAL INITIAL* LEARNING RATES
  `(typically 0.1;0.03;0.01;0.003;0.001)`
- Make sure ENOUGH ITERATIONS for convergence `(typically >200 epochs)`, but EARLY STOPPING on validation_error to avoid overfitting
  `(→ check by plotting learning curves!!)`

Some (old) references on (*shallow, i.e. non deep*) Neural Networks

- *Réseaux de neurones : méthodologie et applications*, G. Dreyfus et al., Eyrolles, 2002.
- *Réseaux de neurones formels pour la modélisation, la commande, et la classification*, L. Personnaz et I. Rivals, CNRS éditions, collection Sciences et Techniques de l'Ingénieur, 2003.
- *Réseaux de neurones : de la physique à la psychologie*,
  J.-P. Nadal, Armand Colin, 1993.