MSE | MASTER OF SCIENCE IN ENGINEERING

Machine Learning

T-MachLe

9. Artificial Neural Networks

Andres Perez Uribe
Jean Hennebert

Plan

9.1 Motivation for neural networks

9.2 Perceptron & Delta-rule

9.3 Multi-Layer Perceptron (MLP)

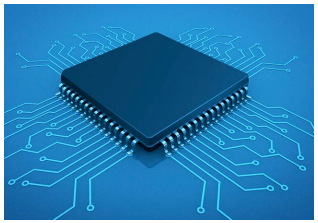
9.4 Backpropagation

9.5 Example: digit recognition

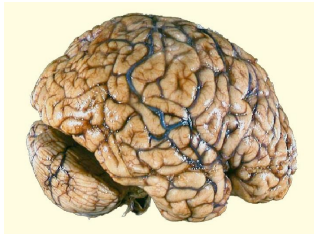
9.6 Practical considerations

9.7 Model selection

Practical Work 9



VS

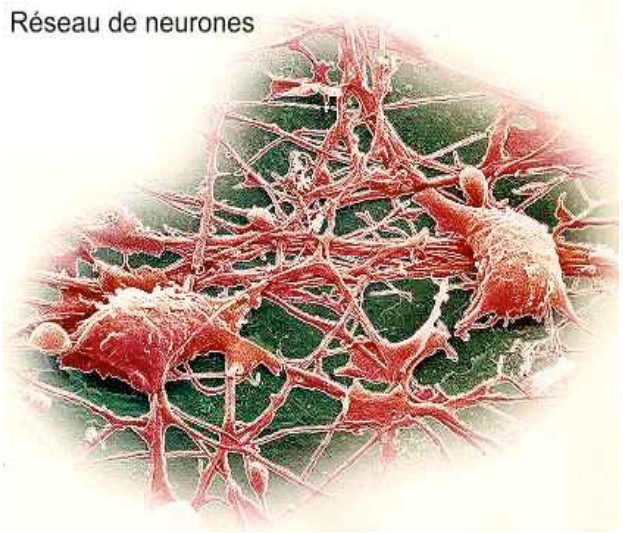


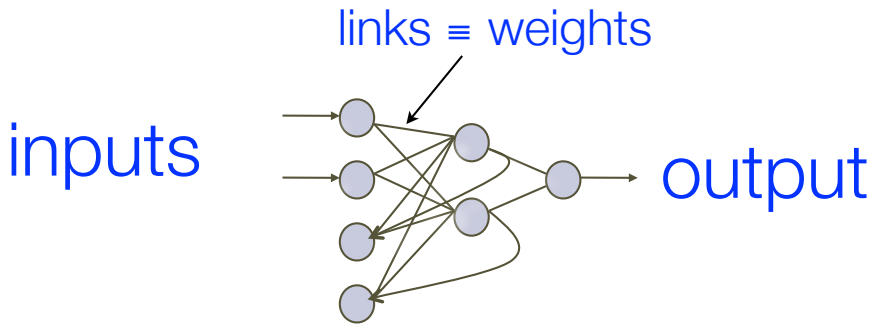
	von Neumann computer	Brain
processor	Complex High frequency (GHz) one or several (cores)	Simple (neuron) low frequency (Hz) Many!
memory	Separated from the processor Address-based access	Integrated to computing (distributed) Content-addressable
computing	centralized sequential Programmable	Distributed Parallel Learning
reliability	Highly vulnerable	Very robust

Connectionist systems

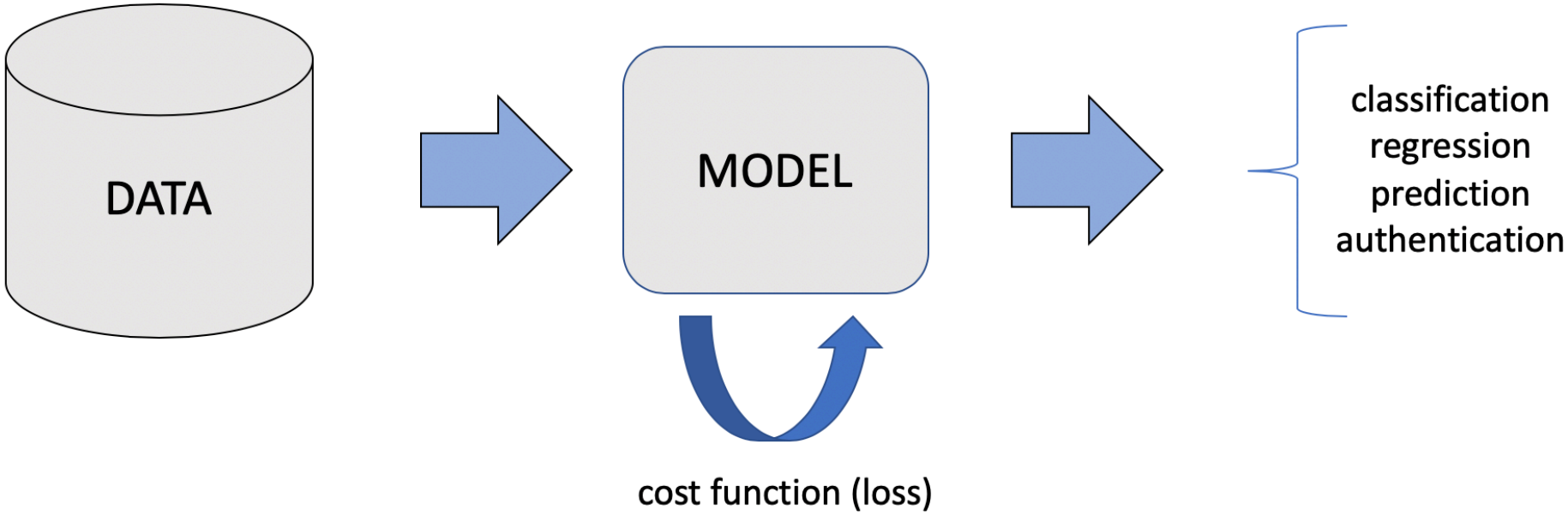
We ideally use a massively parallel architecture where each computational element (neuron) performs a sort of a correlation between inputs and stored values called synaptic weights.

- Paradigm shift: we replace programming by learning





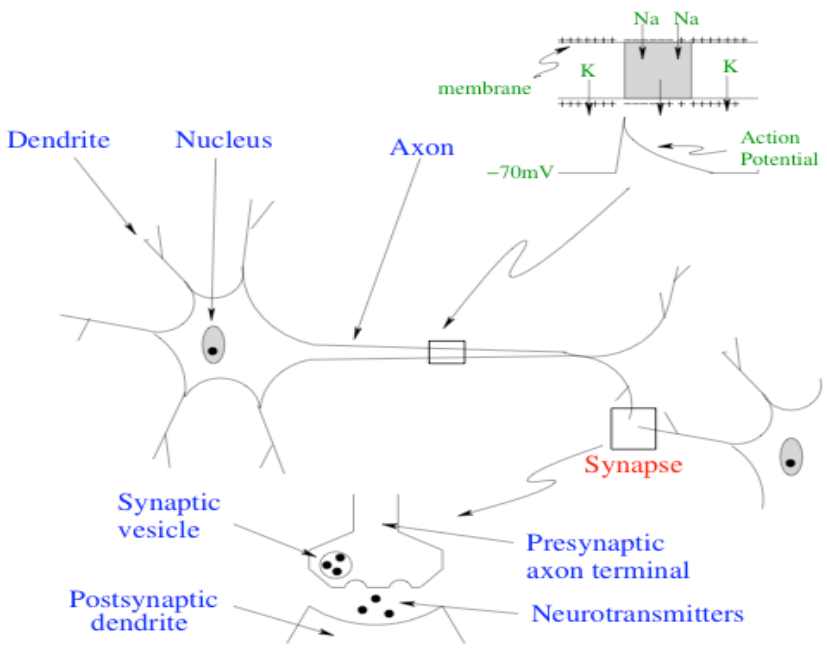
Neural-net state-of-the-art solutions



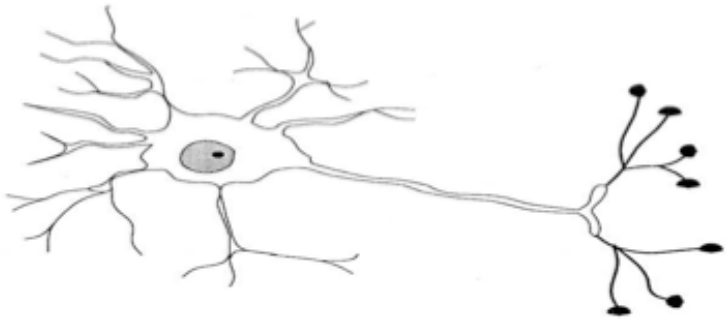
- A neural network is just an alternative algorithm for Machine Learning (e.g., for learning a model using the data-driven approach).
- Nevertheless, nowadays neural networks are at the core of many state-of-the-art solutions

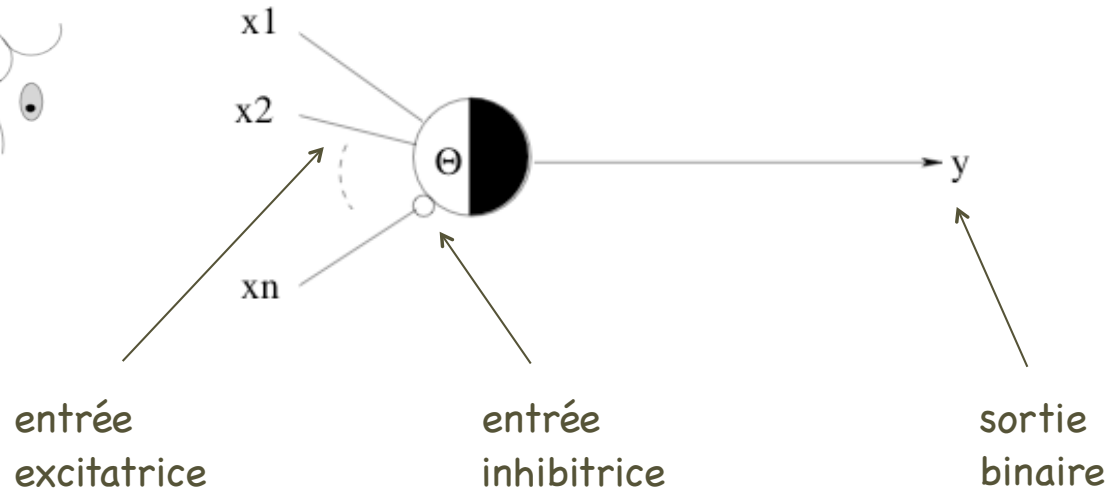
One neuron models

The artificial neuron



McCulloch-Pitts (1948)

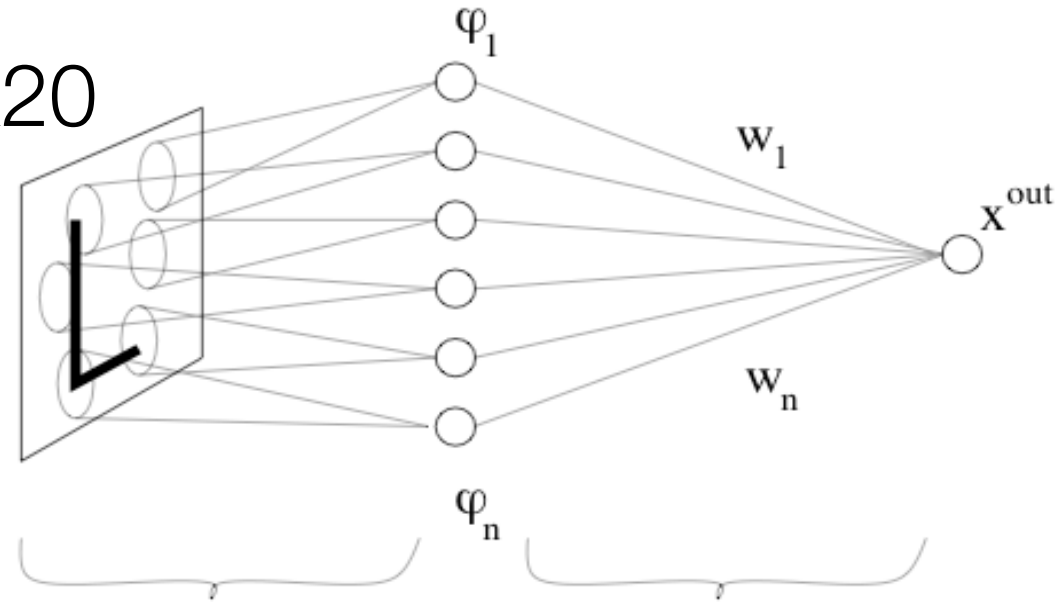




Si $(x_1 + x_2 + \ldots - x_n) \geq \Theta$ alors $y = 1$, sinon $y = 0$

Perceptron

20x20



NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

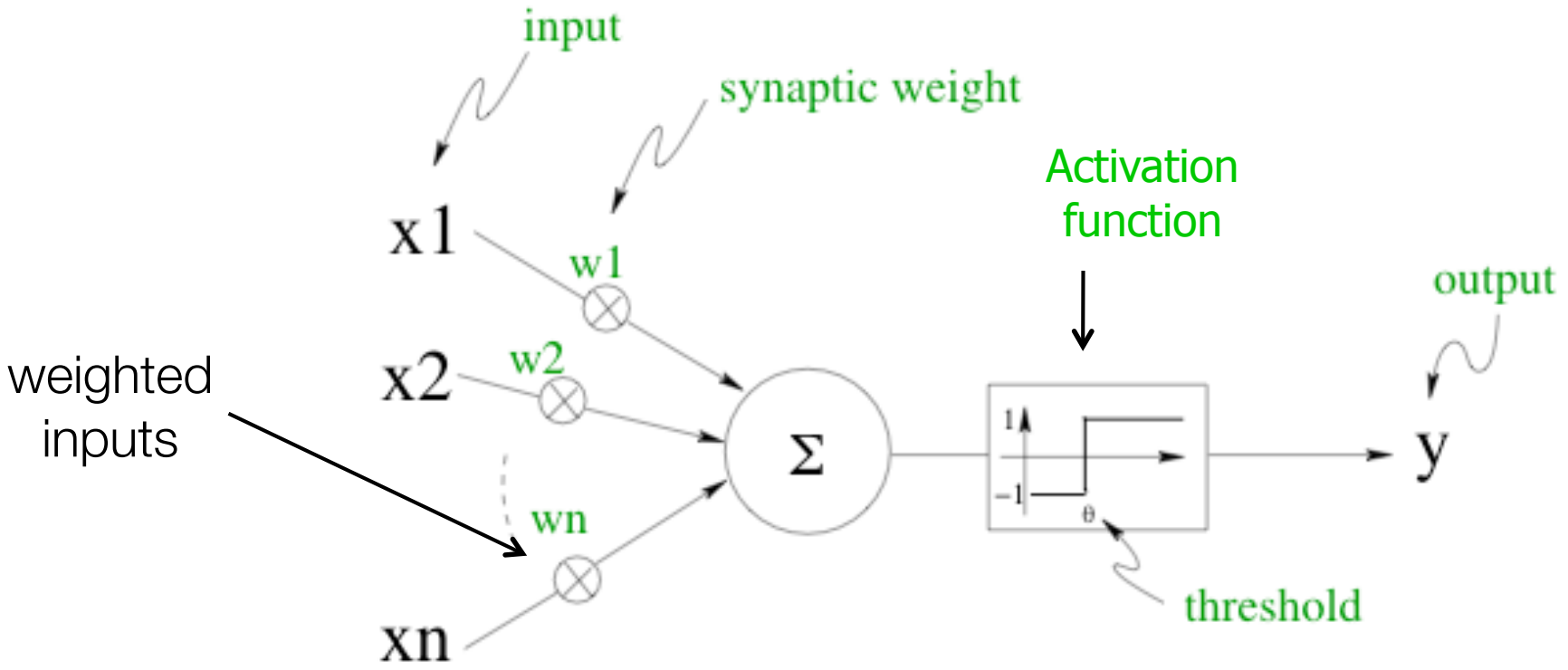
WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's $2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

The New York Times, July 8 1958

Rosenblatt's Perceptron (1958)

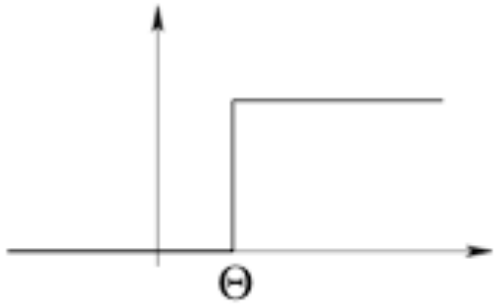


If $(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n) \geq \Theta$ then y = 1, otherwise y = -1

If $\sum w_i x_i \geq \Theta$ then y = 1, otherwise y = -1

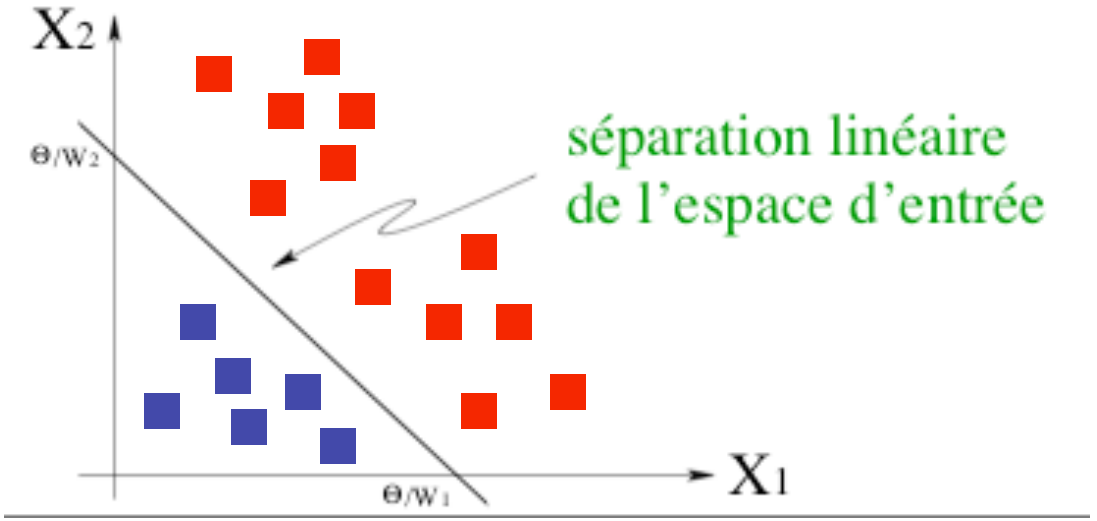
Geometric interpretation

Discontinuity:

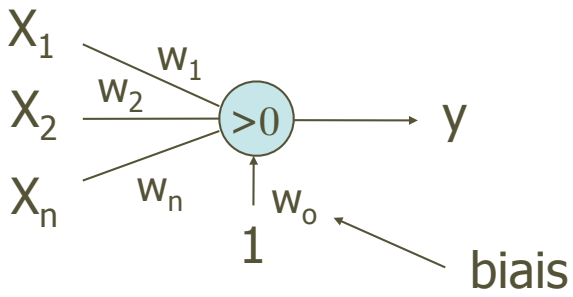


$$\sum_i WiXi = \Theta$$

2D case: $w_1 x_1 + w_2 x_2 = \Theta$, therefore $x_2 = -(w_1/w_2)\, x_1 + \Theta/w_2$



Bias: a learnable threshold of activation



Si $\sum w_i x_i \geq \Theta$ alors y = 1, sinon y = 0

Si $\sum w_i x_i - \Theta > 0$ alors y = 1, sinon y = 0

Si $w_0 = -\Theta$ alors,

Si $\sum w_i x_i + w_0 > 0$ alors y = 1, sinon y = 0

Perceptron learning algorithm

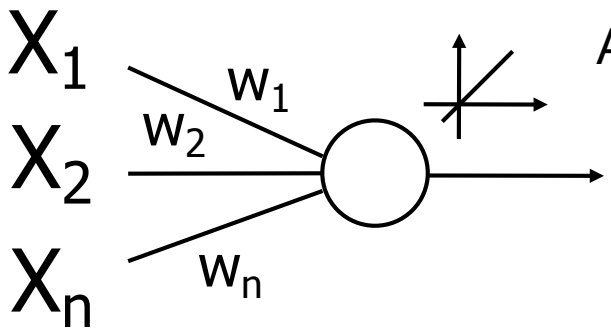
1. Randomly initialize weights
2. Compute the neuron's output **y** for a given input vector **x**

   $$\mathbf{y = \sum_j w_j x_j}$$
3. Weight update: $\mathbf{W_j(t+1) = W_j(t) + \eta(d-y)x_j}$

   d is the desired output and η is the learning rate, $\mathbf{0.0 < \eta < 1.0}$
4. Repeat 2 and 3 for a given number of steps or until the error is smaller than a given threshold, i.e., $\mathbf{error < 1/2 \sum_p (y_p - d_p)^2}$

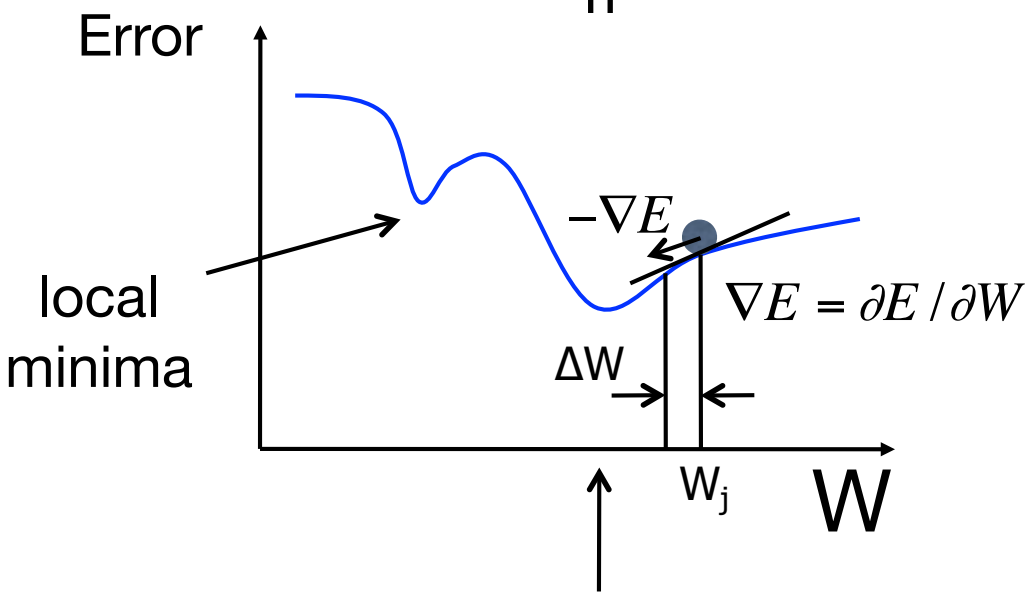
Gradient descent

Widrow-Hoff algorithm / Least Mean Squared or Delta rule



ADALINE:ADAptive LINear Element

$y=\Sigma w_i x_i$



Set of weights minimizing the error

$$E = 1/2\sum_{i=1}^{p}\|y_i - d_i\|^2, y = \sum_j w_j x_j$$

$$\nabla E = \partial E / \partial w = \partial E / \partial y \cdot \partial y / \partial w$$

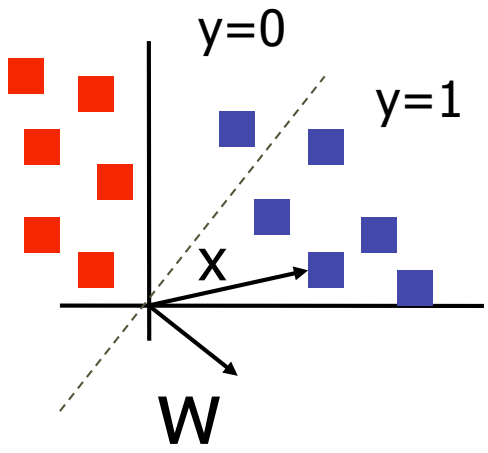
$$\partial E / \partial y = y - d$$

$$\partial y / \partial w_j = \frac{\partial}{\partial w_j}\sum_j w_j x_j = x_j$$

$$\Delta w_j = -\nabla E = \eta(d - y)x_j$$

Learning process:

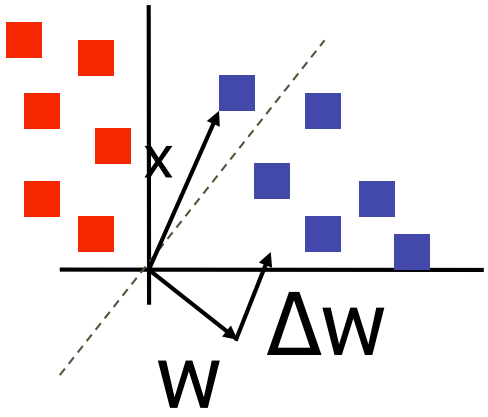
No classification error

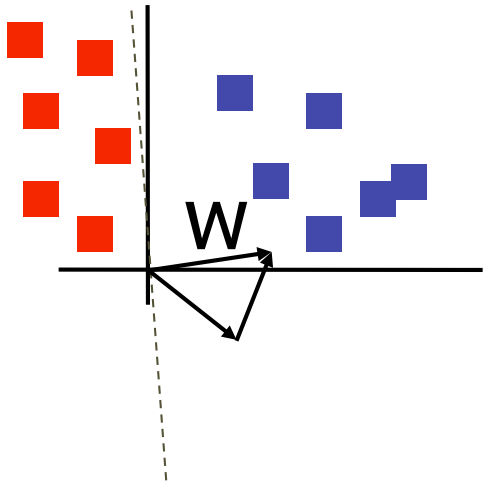


$W_j(t+1) = W_j(t) + \eta(d-y)x_j$

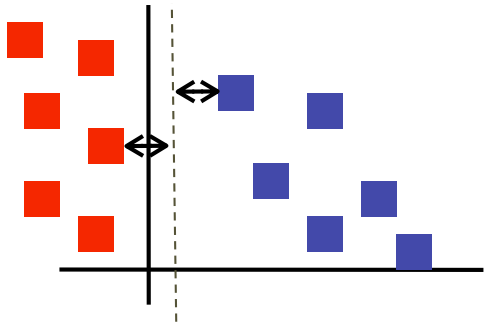
$d=y$, so $W_j(t+1) = W_j(t)$

Error correction after a weight update



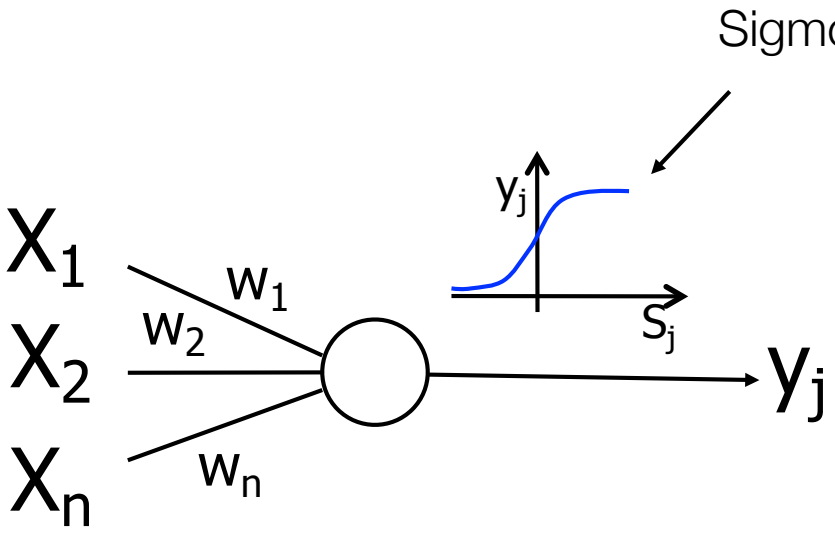


$W_j(t+1) = W_j(t) + \eta(d-y)x_j$



optimal
separation

Generalized Delta rule



$$y_j = \frac{1}{1+e^{-S_j}}, \quad S_j = \sum w_i x_i$$

$$\Delta w_j = \eta(d-y) y_j' x_j$$

$$y_j' = y_j(1-y_j)$$

The tanh and ReLu are other widely used activation functions

Learning code

```
import numpy as np

# sigmoid function and derivative
def sigmoid(neta):
    output = 1 / (1 + np.exp(-neta))
    d_output = output * (1 - output)
    return (output, d_output)

# input dataset
X = np.array([  [0.2,0.2],
                [0.2,0.5],
                  …….
                [0.8,0.7],
                [0.8,0.9] ])

# output classes
y = np.array([[0,0,0,0, …… 1,1,1,1]]).T

# seed random numbers to make calculation
# deterministic (just a good practice)
np.random.seed(1)

# initialize weights randomly with mean 0
weights = np.random.normal(size=2)
bias = np.random.normal(size=1)
```

```
inputs = X
target = y

for iter in xrange(100):

    # forward propagation
    neta = np.dot(inputs, weights) + bias
    output, d_output = sigmoid(neta)

    # how much did we miss?
    error = target - output

    # learning rate
    alpha = 0.1

    d_w_x = alpha * error * d_output * inputs[:,0]
    d_w_y = alpha * error * d_output * inputs[:,1]
    d_b = alpha * error * d_output

    # update weights
    weights += np.array([np.sum(d_w_x), np.sum(d_w_y)])
    bias += np.sum(d_b)

print "Network After Training:"
print weights, bias
```

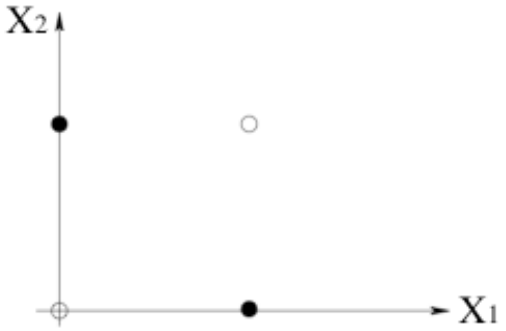
Neural network models

The XOR problem

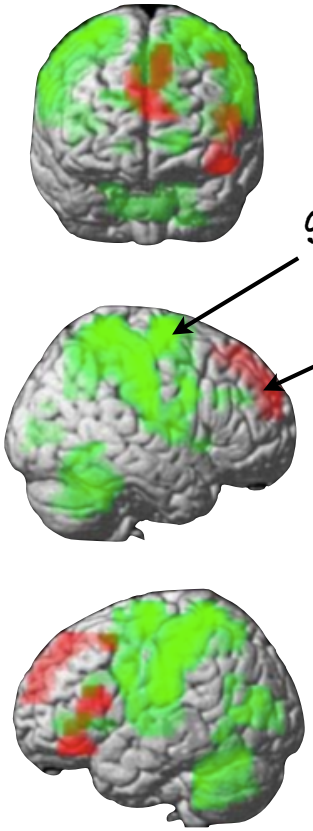
Can you draw a line to separate the two classes ?

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

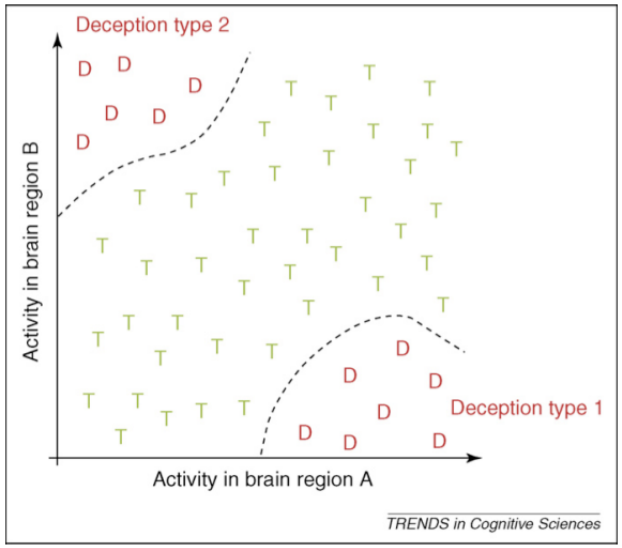
y { ○ : 0
  ● : 1



The XOR problem
(Minsky & Pappert, 1969)



NeuroImage 28 (2005) 663 – 668

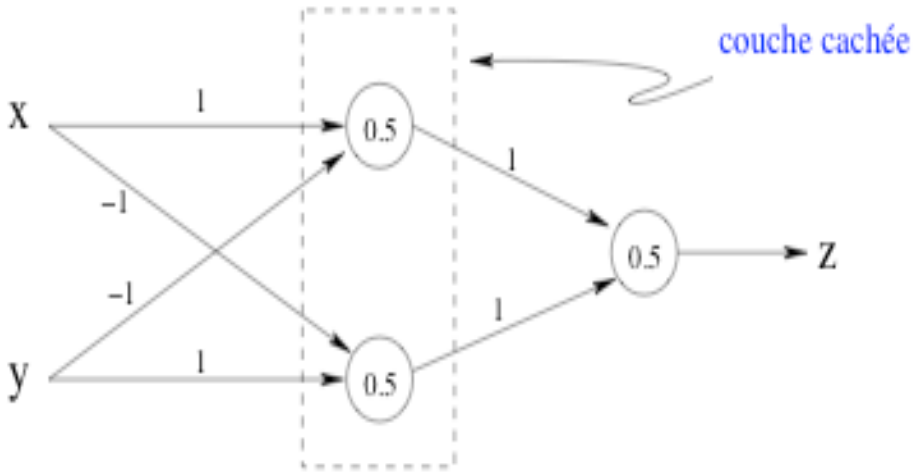


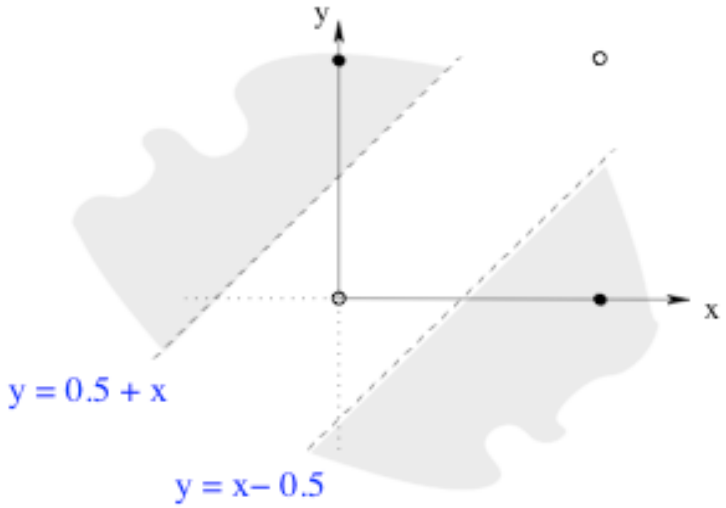
Trends in Cog. Sci, Vol 12, No 4, March 2008

Towards multi-layer networks

two lines can do the work!

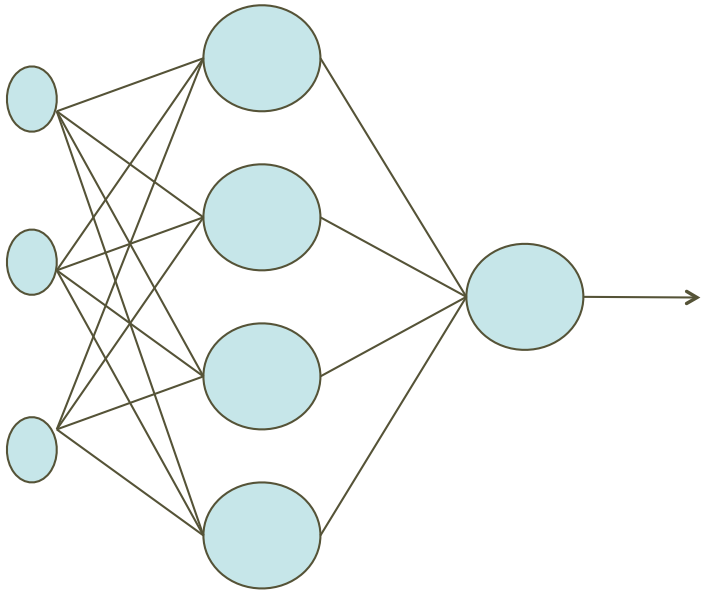
gray = one class
white = other class

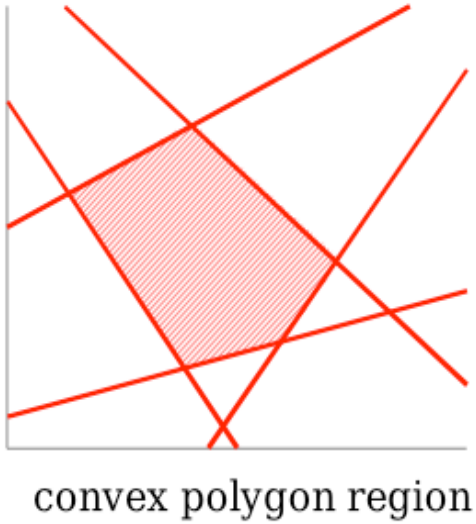




if (x - y ≥ 0.5) OR (y - x ≥ 0.5) then z = 1, else z = 0

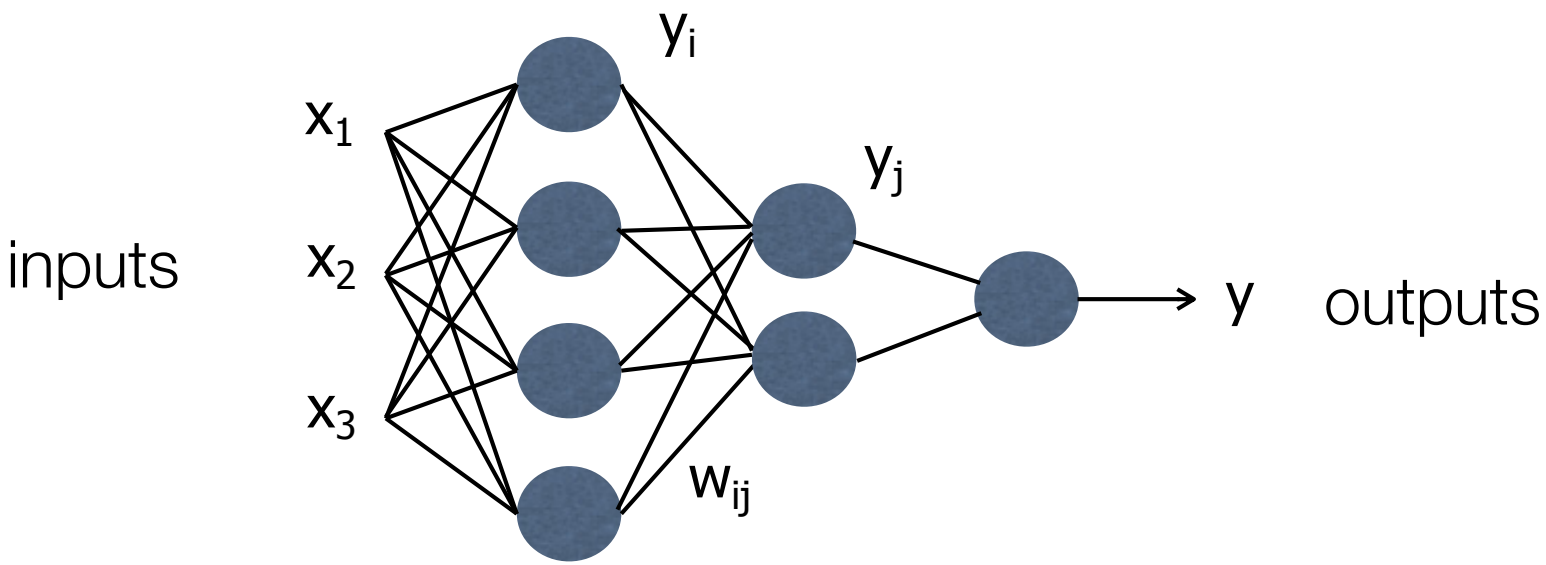
A Multi-Layer Perceptron (MLP) with one hidden layer / linear activation functions

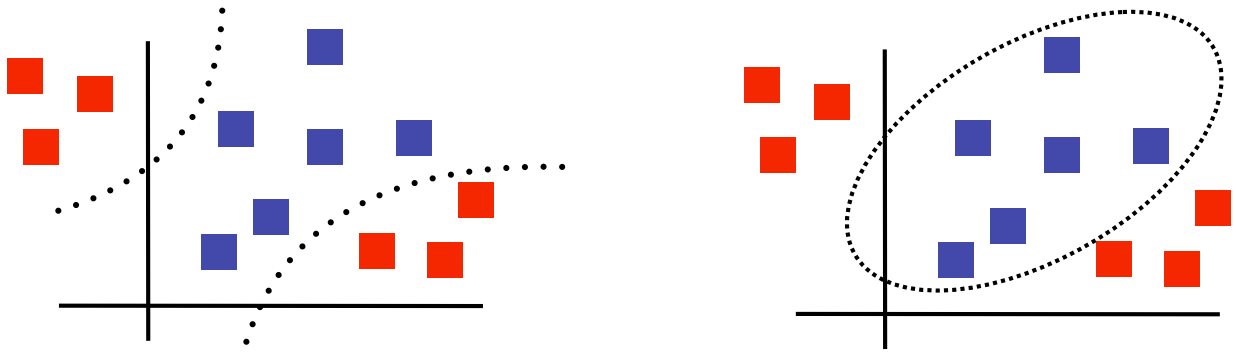




two-layer neural network

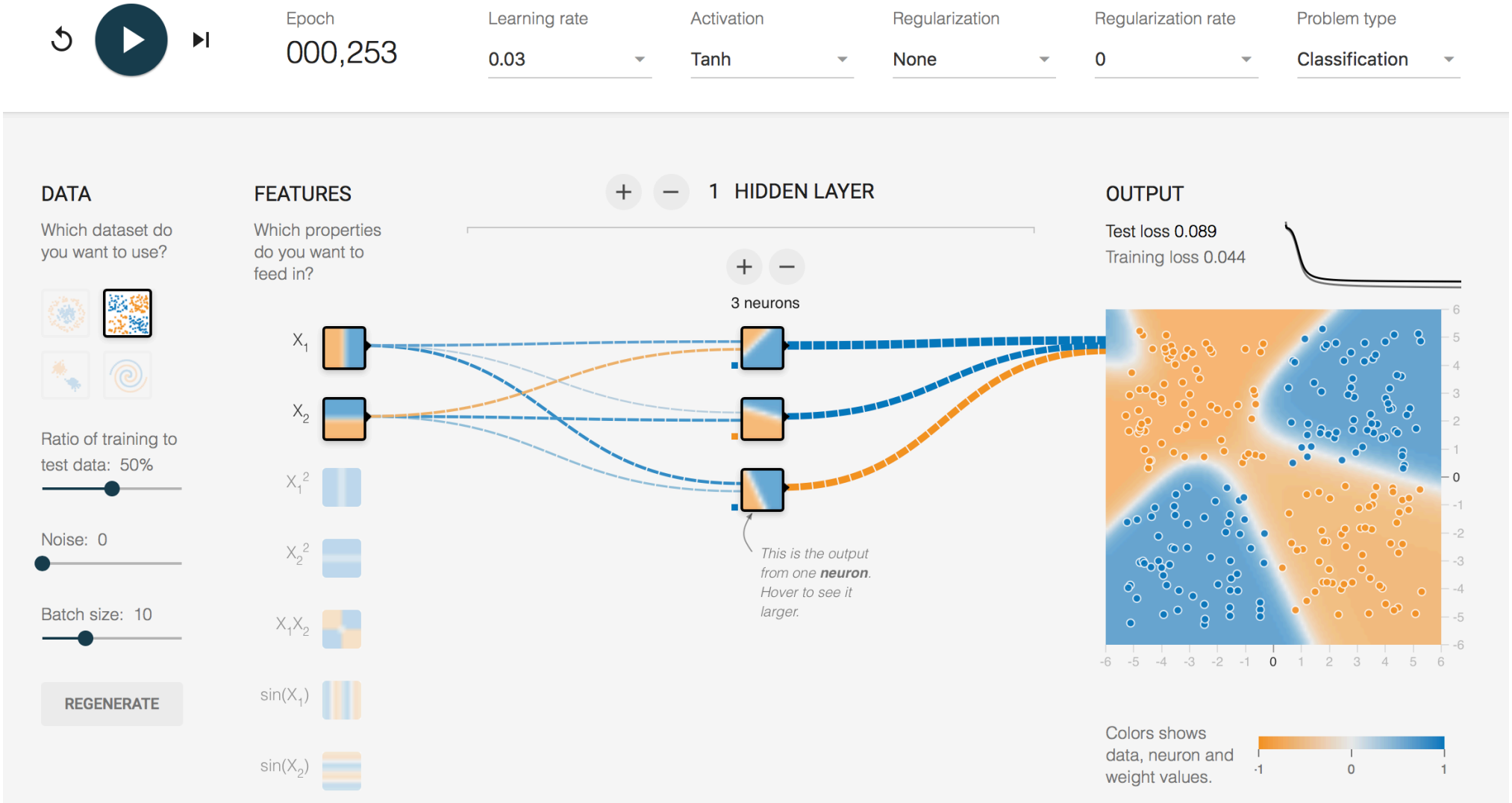
Multi-Layer Perceptron (MLP) with non-linear activation functions





Nonlinear separation

http://playground.tensorflow.org



Backpropagation

Backpropagation algorithm

(Paul Werbos, 1974; Rumelhart & McClelland, 1986)

Error function (or loss) to be minimized:

$$E(w^{(1)}, w^{(2)}) = 1/2 \sum_{\mu} \sum_{i} \left\| t_i^{out}(\mu) - x_i^{out}(\mu) \right\|^2$$

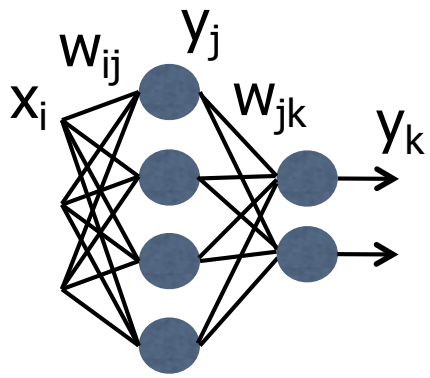
training pattern output neuron

Stochastic gradient descent:

(k) indicates the layer

$$\Delta w_{ij}^{(k)} = -\eta \partial E / \partial w_{ij}^{(k)}$$

Backpropagation



1. Randomly initialize weights
2. Compute the outputs y_k for a given input vector X:

   $y_k=f(\sum_j w_{jk}y_j)$, where $y_j=f(\sum_i w_{ij}x_i)$ and $f(s)=1/(1+e^{-s})$

3. For each output neuron, compute:

   $\delta_k=(y_k-t_k)f'(y_j)$, where $f'(y_j)=y_k(1-y_k)$ is the derivative of the sigmoid function, and t_k is the desired output of output neuron k

4. For each neuron of the hidden layer, compute:

   $\delta_j=\sum_j w_{ij}f'(x_i)\delta_k$, where $f'(x_i) = y_j(1-y_j)$

5. Update the networks' weights as follows:

   $w_{jk}(t+1) = w_{jk}(t) - \eta\delta_k y_j$; $w_{ij}(t+1) = w_{ij}(t) - \eta\delta_j x_i$,

   where η is the learning rate, $0<\eta<1$

6. Repeat 2 to 5 for a given number of steps or until the error is smaller than a given threshold

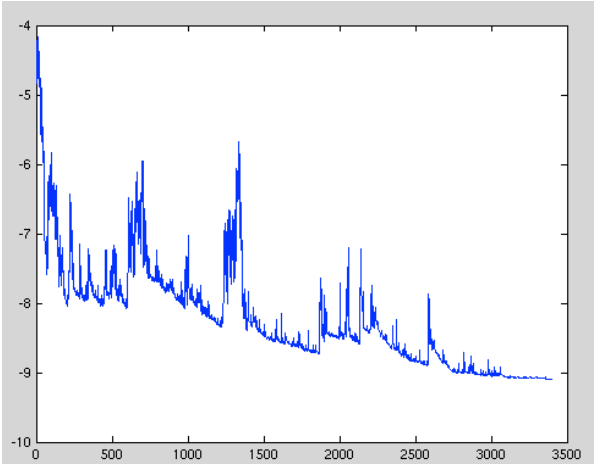
Backpropagation code

```
01. import numpy as np
02. X = np.array([ [0,0,1],[0,1,1],[1,0,1],[1,1,1] ])
03. y = np.array([[0,1,1,0]]).T
04. alpha,hidden_dim = (0.5,4)
05. synapse_0 = 2*np.random.random((3,hidden_dim)) - 1
06. synapse_1 = 2*np.random.random((hidden_dim,1)) - 1
07. for j in xrange(60000):
08.     layer_1 = 1/(1+np.exp(-(np.dot(X,synapse_0))))
09.     layer_2 = 1/(1+np.exp(-(np.dot(layer_1,synapse_1))))
10.     layer_2_delta = (layer_2 - y)*(layer_2*(1-layer_2))
11.     layer_1_delta = layer_2_delta.dot(synapse_1.T) * (layer_1 * (1-
            layer_1))
12.     synapse_1 -= (alpha * layer_1.T.dot(layer_2_delta))
13.     synapse_0 -= (alpha * X.T.dot(layer_1_delta))
```


from: http://iamtrask.github.io/2015/07/27/python-network-part2/

Online vs batch learning

- The stochastic gradient descent method is an “on-line” method where the true gradient is approximated by means of a single example.
- The error fluctuates
- Several passes over the « shuffled » training set are required for convergence



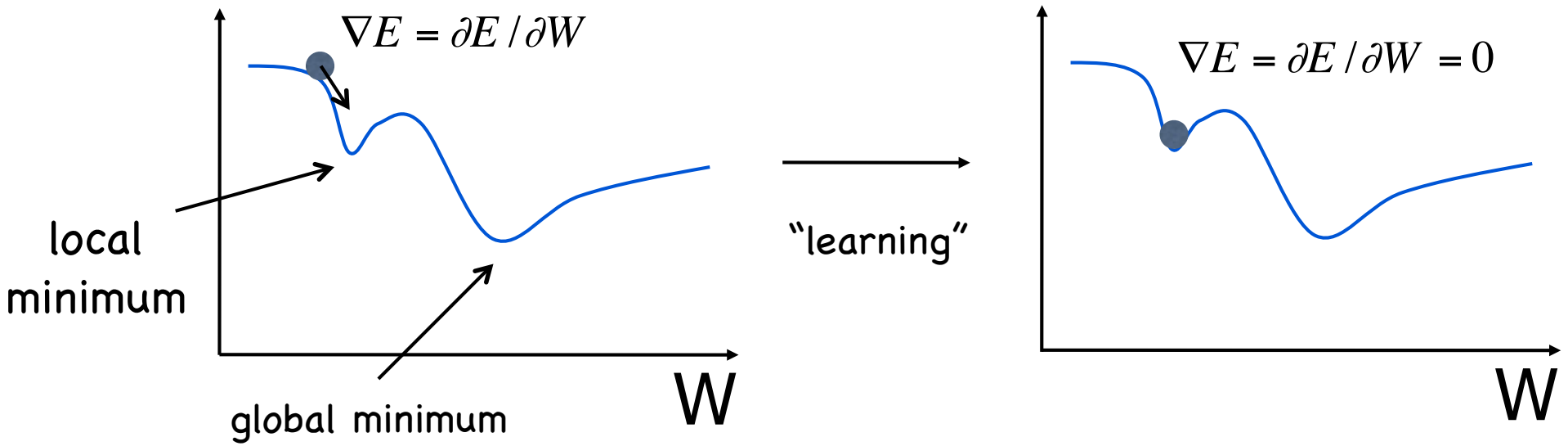
- A compromise between computing the true gradient and the gradient at a single example, is to compute the gradient against more than one training example (called a "mini-batch") at each step.

Gradient descent methods

- Popular gradient descent methods are:
  - conjugate gradient method (the searching direction is based on the previous searching direction)
  - Rprop: Resilient Backprop uses the sign of the derivative only and modifies a weight by multiplying it by a constant
  - BFGS: Broyden–Fletcher–Goldfarb–Shanno algorithm
  - RMSprop: Root Mean Square Propagation
  - Adam: Adaptive Moment Estimation:

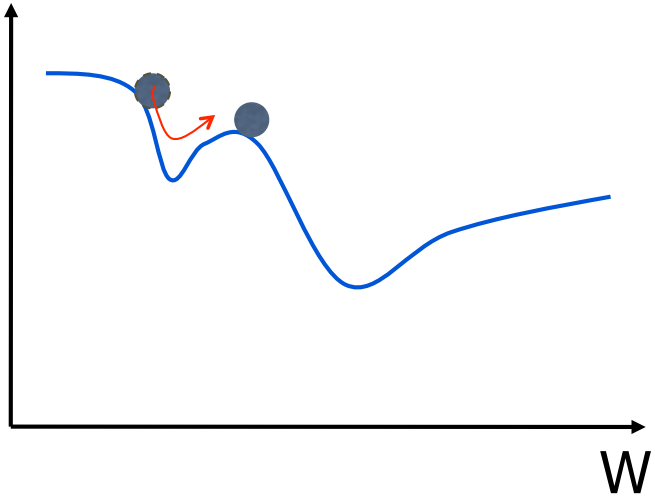
$$
\begin{aligned}
m(w,t) &:= \gamma_1 * m(w,t-1) + (1-\gamma_1) * \nabla Q_i(w) \\
v(w,t) &:= \gamma_2 * v(w,t-1) + (1-\gamma_2) * (\nabla Q_i(w))^2 \\
\hat{m}(w,t) &:= \frac{m(w,t)}{(1-\gamma_1^t)} \\
\hat{v}(w,t) &:= \frac{v(w,t)}{(1-\gamma_2^t)} \\
w &:= w - \frac{\eta}{\sqrt{\hat{v}(w,t)} + \epsilon} * \hat{m}(w,t)
\end{aligned}
$$

Backpropagation with momentun



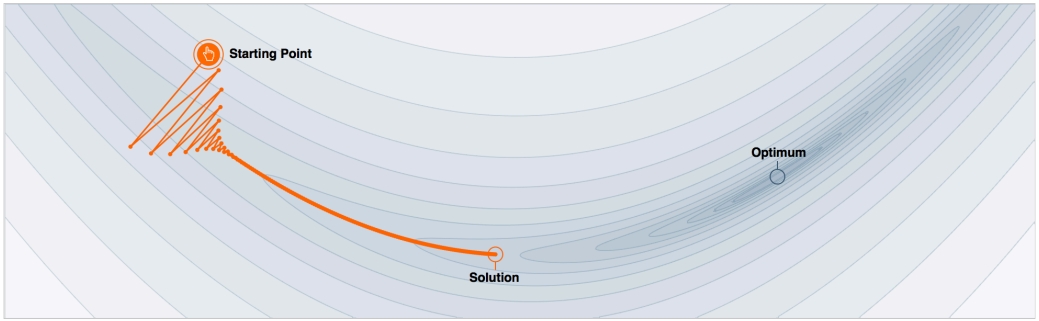
$$\Delta w_{ij}^{(k)}(t) = -\eta \partial E / \partial w_{ij}^{(k)}(t) + \mu \Delta w_{ij}^{(k)}(t-1)$$

momentum term
(e.g., μ=0.9)

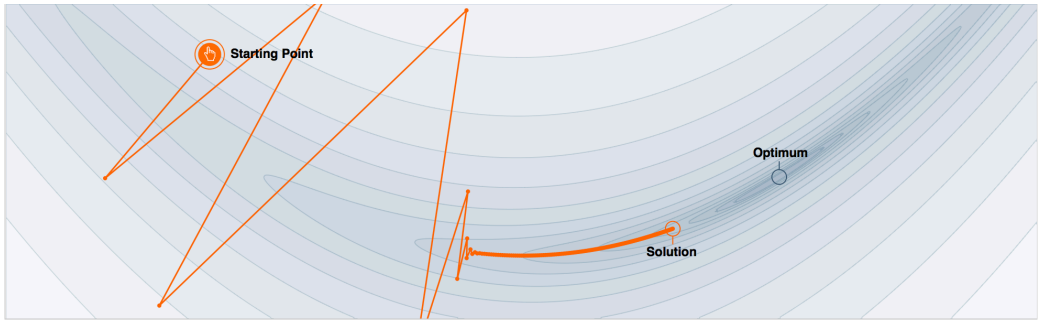


The momentum in action

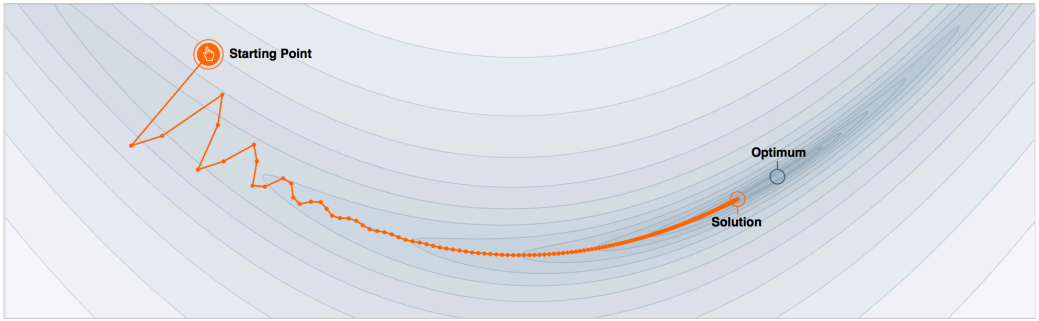
https://distill.pub/2017/momentum/



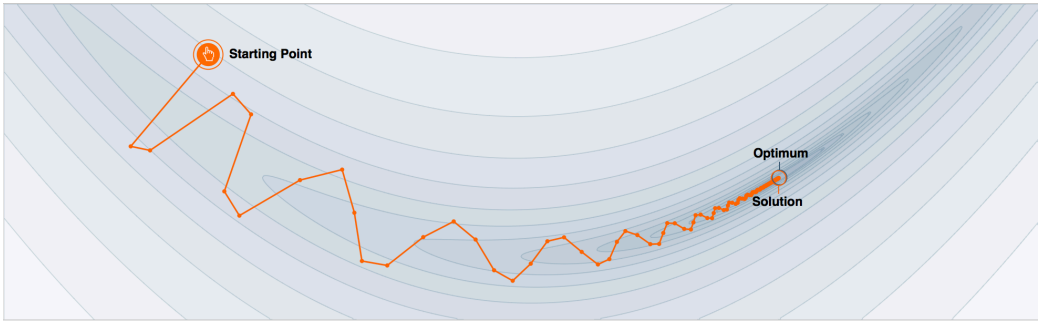
Learning rate = 0.003, Momentum = 0.0



Learning rate = 0.004, Momentum = 0.0



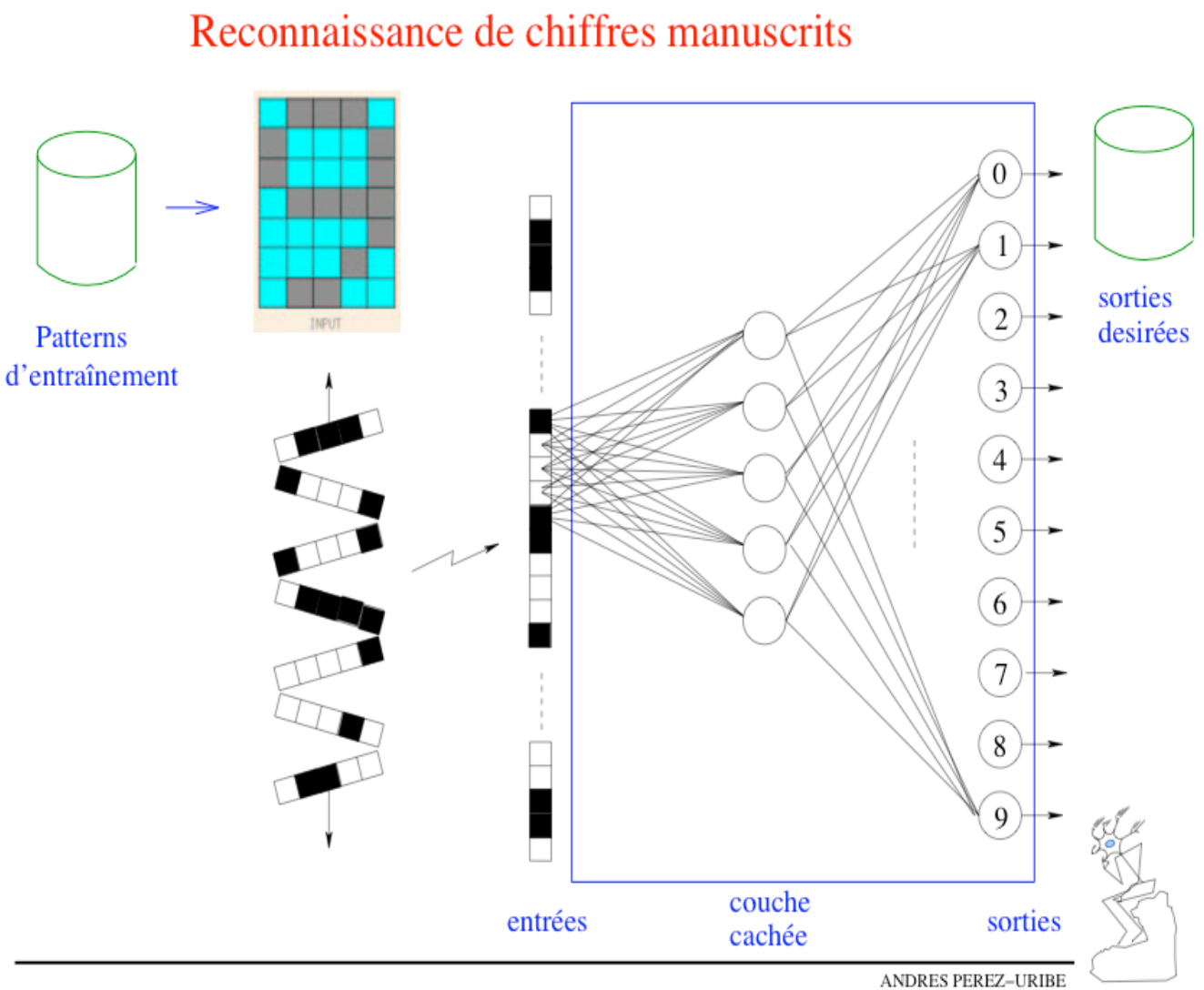
Learning rate = 0.003, Momentum = 0.7



Learning rate = 0.003, Momentum = 0.85

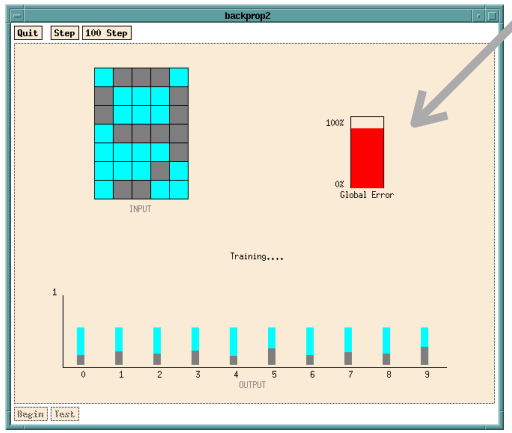
Example application

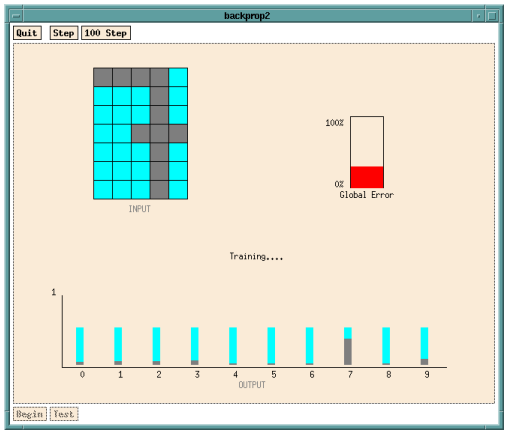
Example


Reconnaissance de chiffres manuscrits
Patterns
d'entraînement
INPUT
0
1
2
3
4
5
6
7
8
9
sorties
desirées
entrées
couche
cachée
sorties
ANDRES PEREZ-URIBE

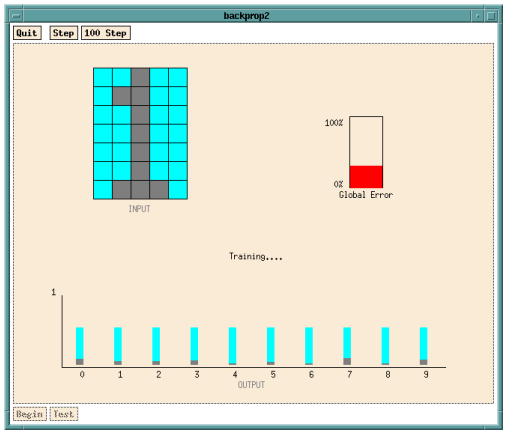
Learning process

loss

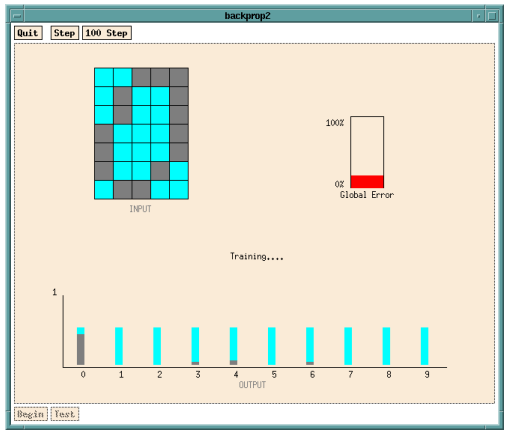




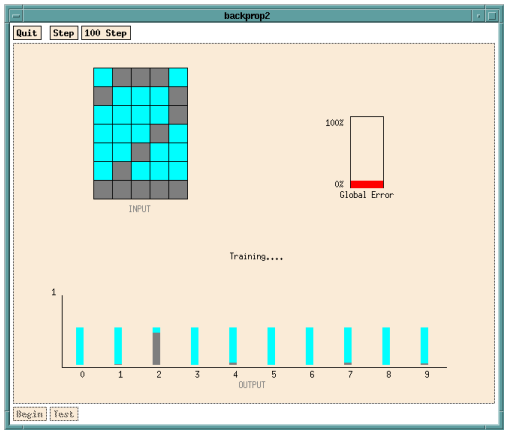
après 1000 pas



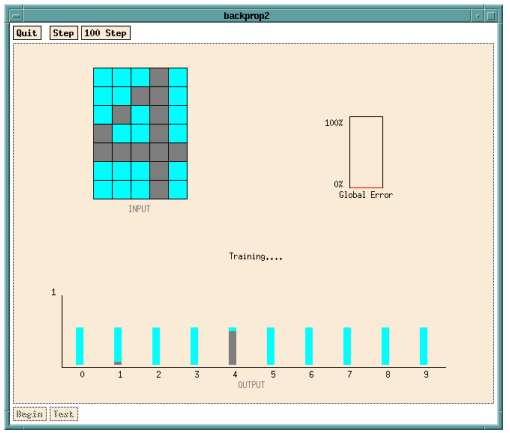
après 1021 pas



après 4000 pas



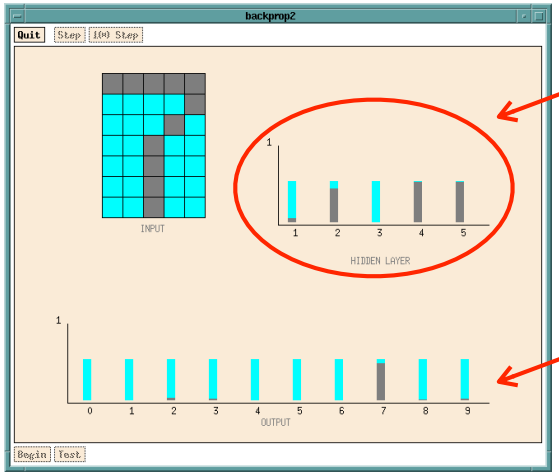
après 8000 pas



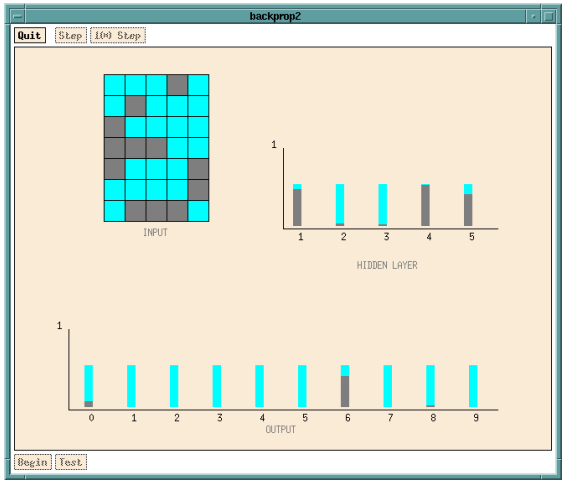
après 12000 pas

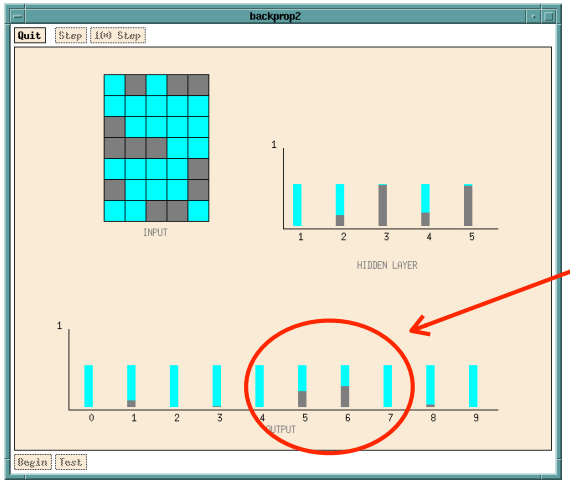
Generalization capability

hidden unit activation ≈ feature detection



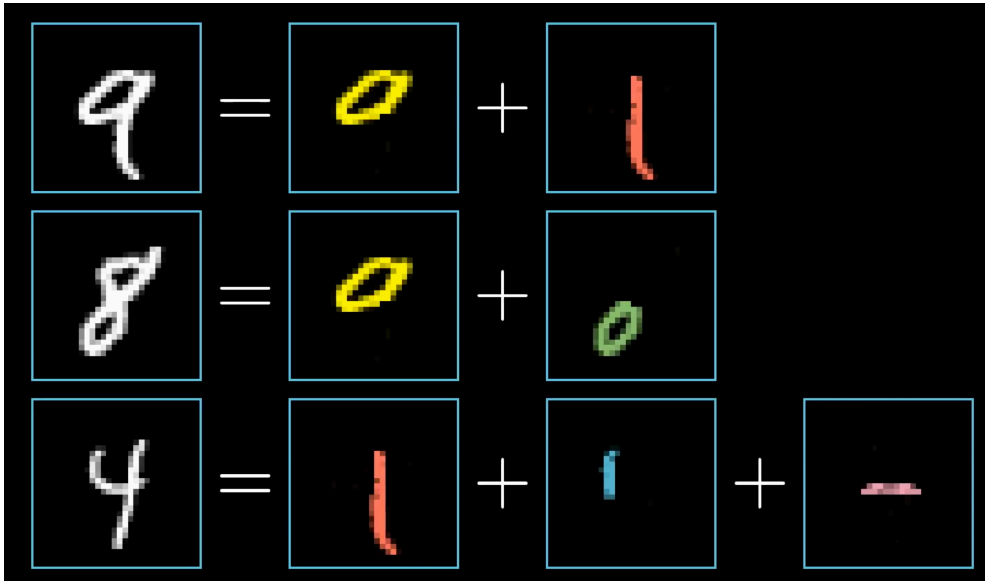
outputs' activation ≈ digit identification



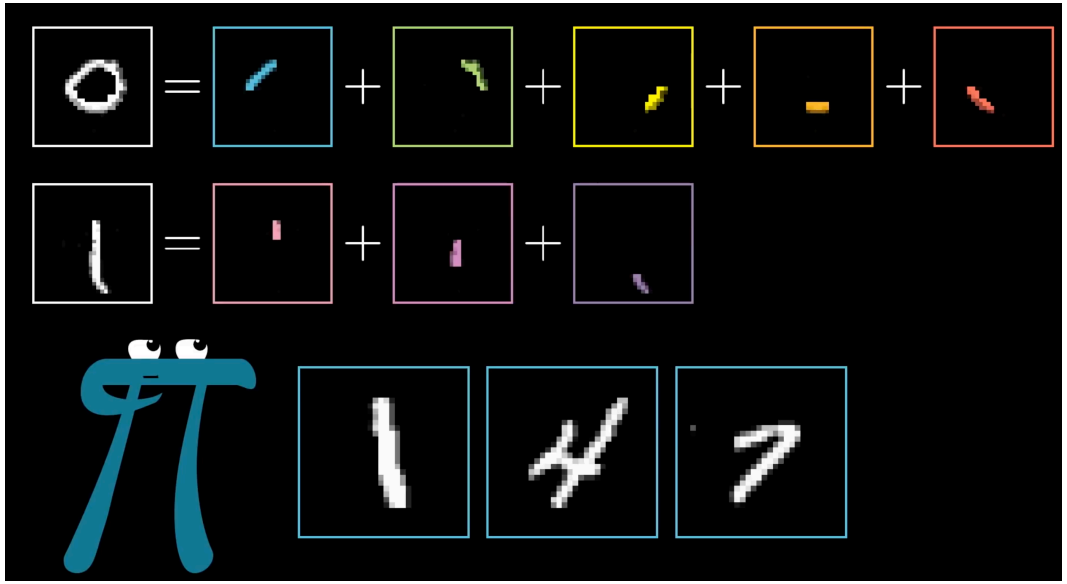


5 or 6 ?

Detected features in the hidden layers



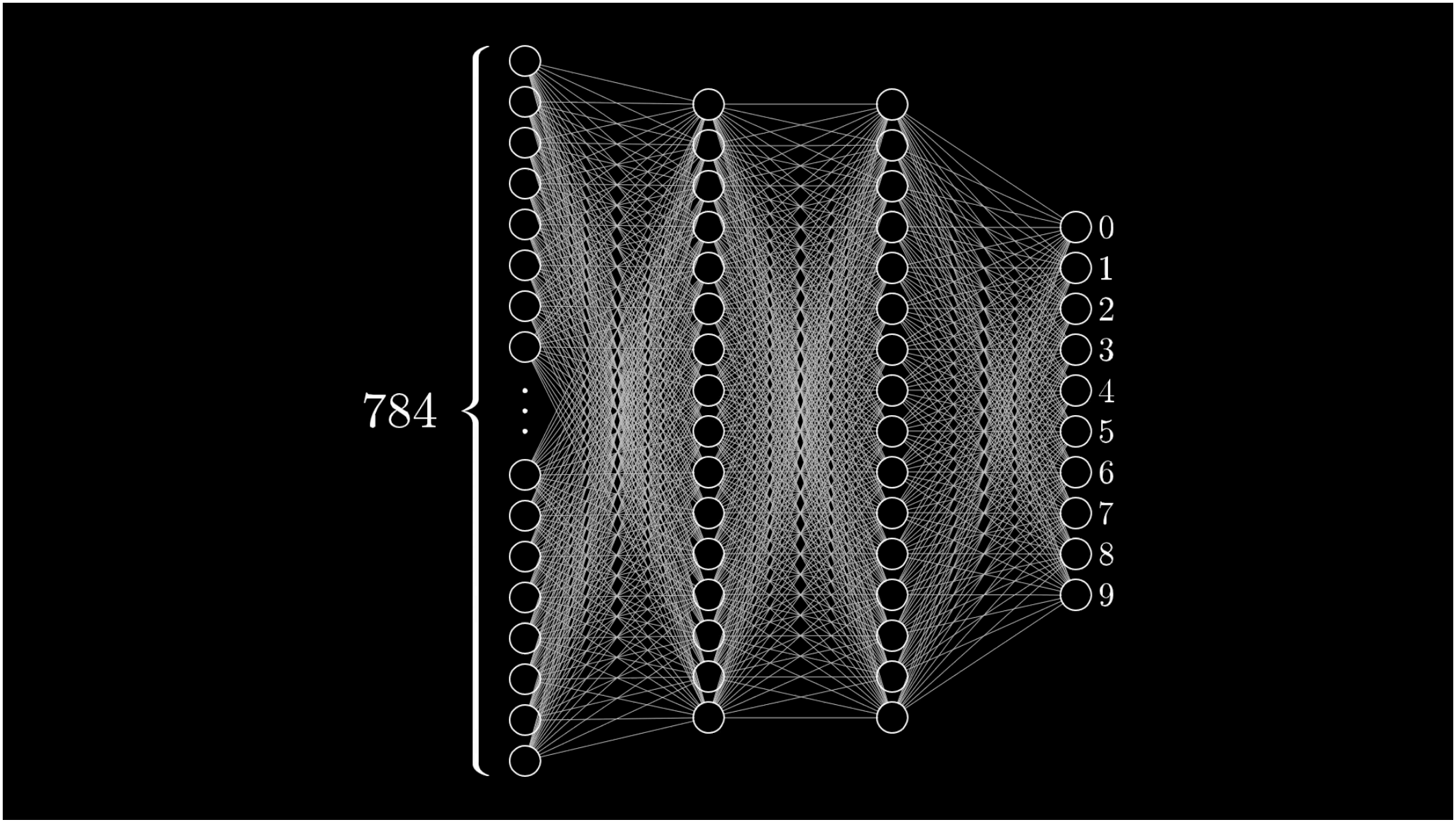
ideally, but too difficult



a loop might be the sum of several edges

- Unfortunately, we cannot be sure the Neural network will use meaningful features to solve the problem at hand (e.g., digit recognition).

from “What is a Neural Network”, video by 3blue1brown

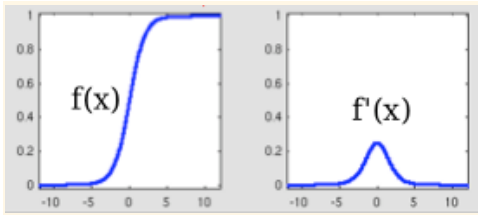


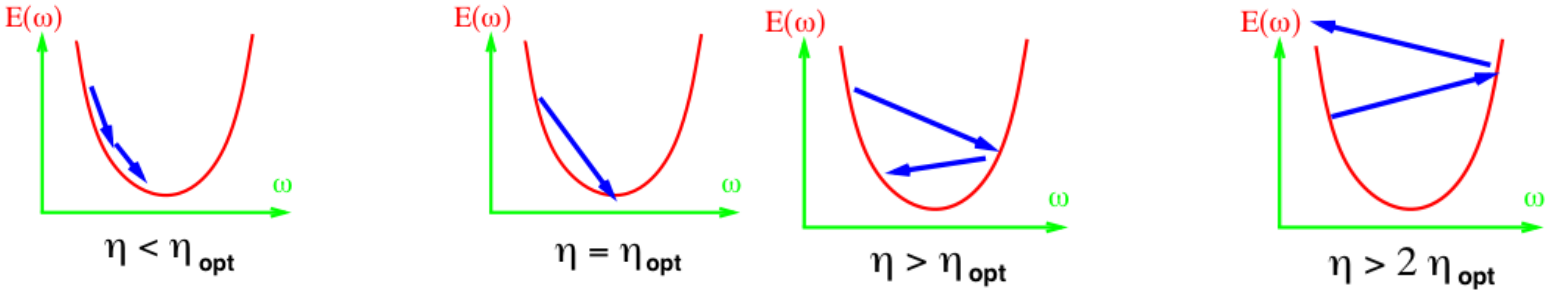
https://www.youtube.com/results?search_query=3+brown+1+blue

Practical considerations

Practical considerations (1)

- Topology: number of layers, number of hidden neurons in each hidden layer (complexity of the model)
- Activity function: sigmoid, tanh, ReLu, softmax
- Weight Initialisation: if too small, all units do the same, if too big, the sigmoid/tanh saturates
- Learning rate:





Y. LeCun, 2006

Practical considerations (2)

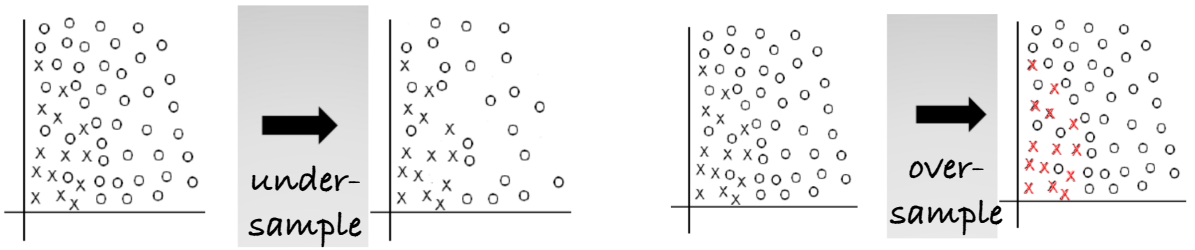
- Input normalization

  Neural networks might behave badly if the individual features do not more or less look like standard normally distributed data: Gaussian with zero mean and unit variance.

- Encoding categorical data

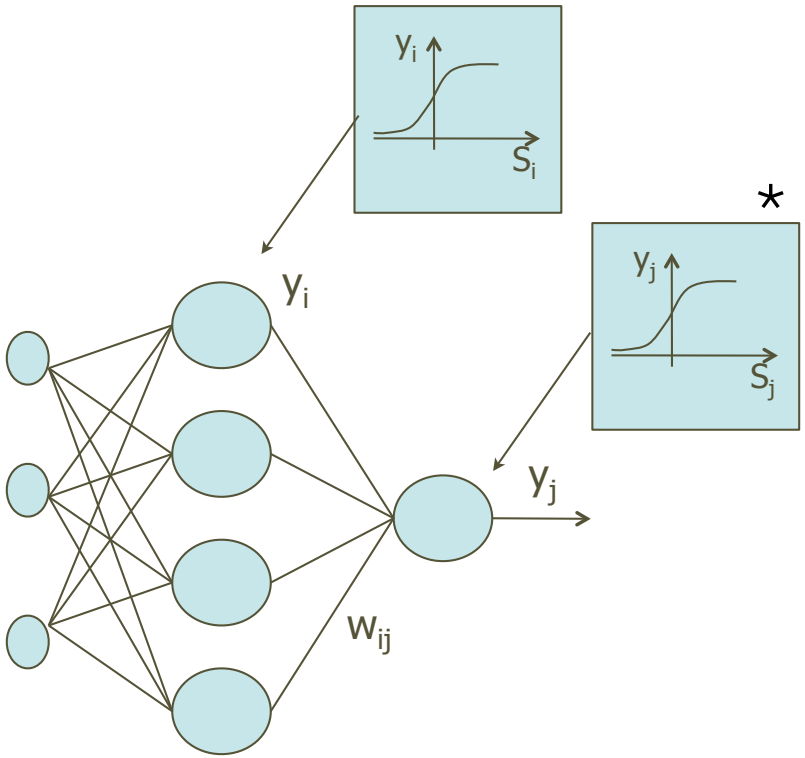
  e.g., given an input with two possible values high and low productivity -> use i-of-C coding: (1,0) and (0,1)

- Class imbalance problem: under-sample or over-sample to balance the set sizes for all classes.



from Longadge et al, 2013

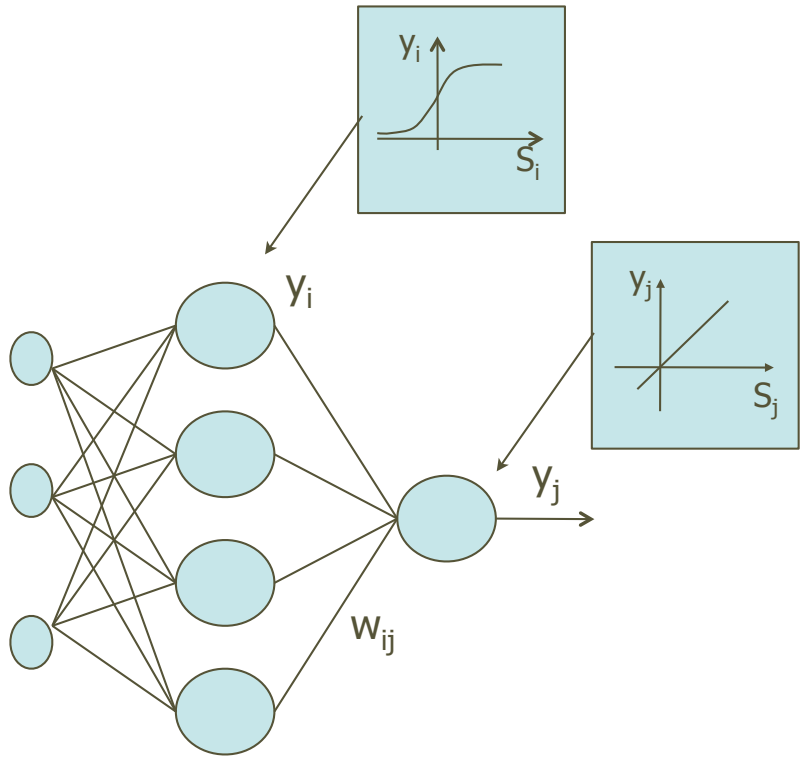
Classification and regression (3)



classification network

Loss function: Cross-entropy

$CE = - \sum_p d_p \log(y_p)$



regression network

Loss function: Mean Squared Error

$MSE = 1/2 \sum_p (y_p - d_p)^2$

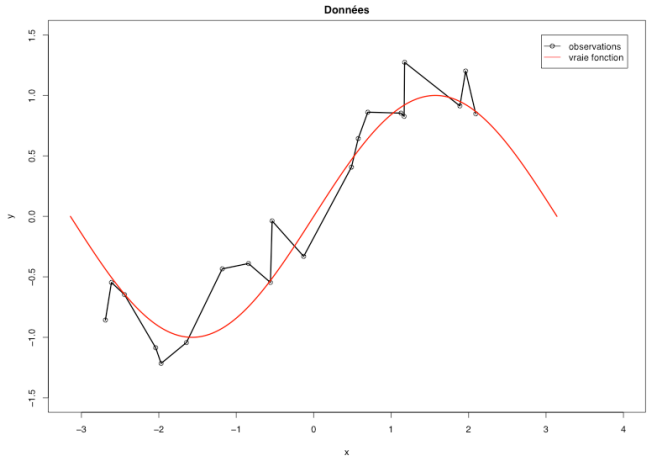
* softmax if more than 2 classes

Dealing with the number of hidden neurons (4)

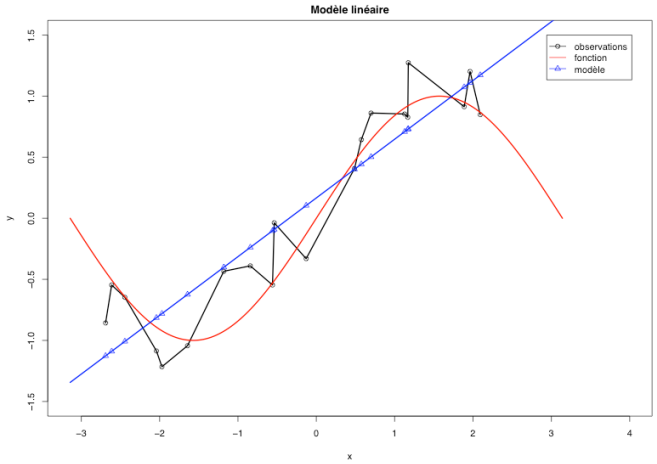
- More neurons, more capacity ?
- We might have the insight that increasing the number of neurons can always be beneficial. However, a model with more neurons has more weights, and those weights have to be learned to suitably solve our problem at hand.
- The problem is that a model with more adjustable parameters (weights) is more plastic or malleable and risks of overfitting the data in unsuitable ways (overfitting).

A regression problem

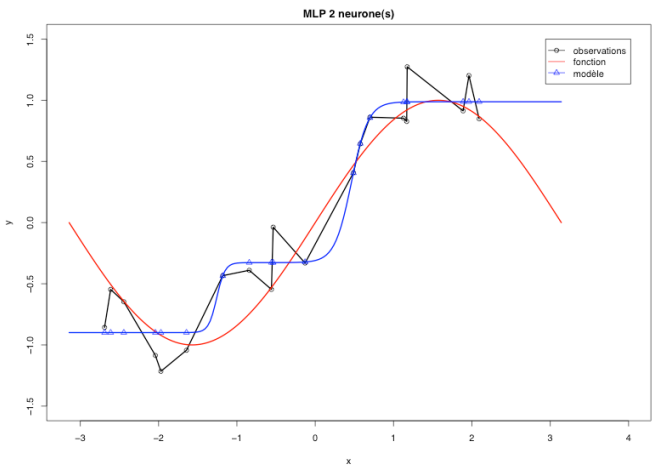
the risk of overfitting



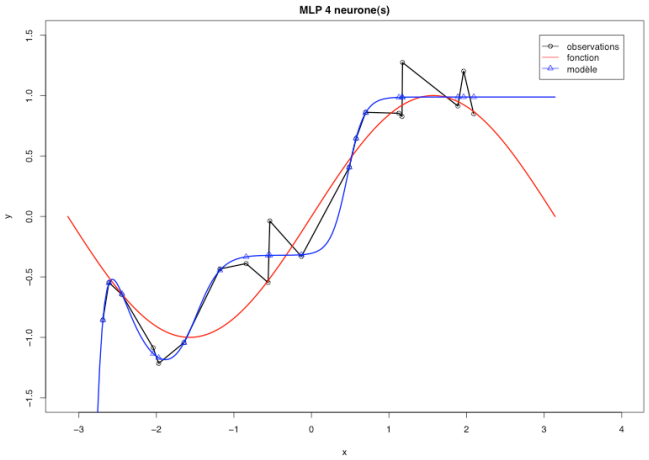
Data: training points extracted from a sin(x) function



Linear model



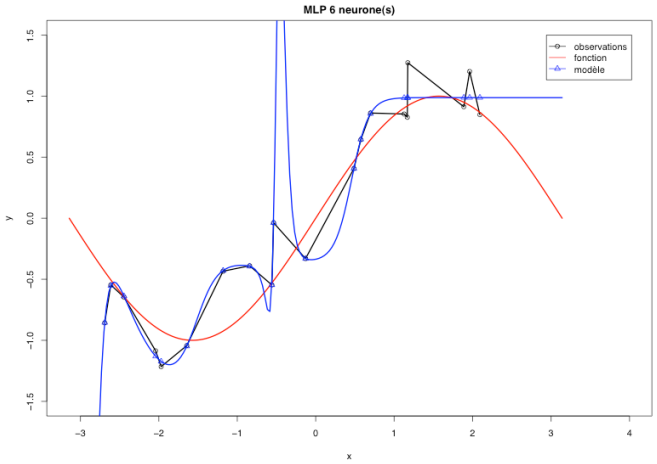
MLP 1-2-1



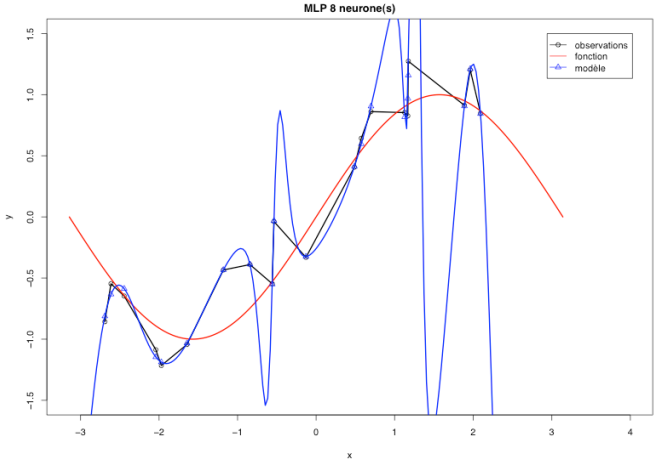
MLP 1-4-1

A regression problem

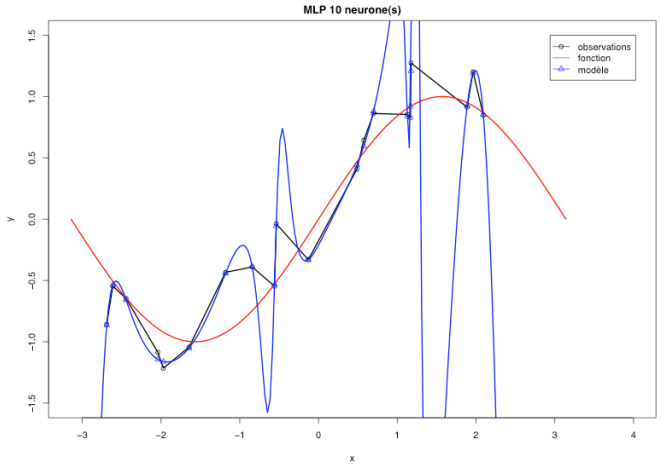
the risk of overfitting



MLP 1-6-1

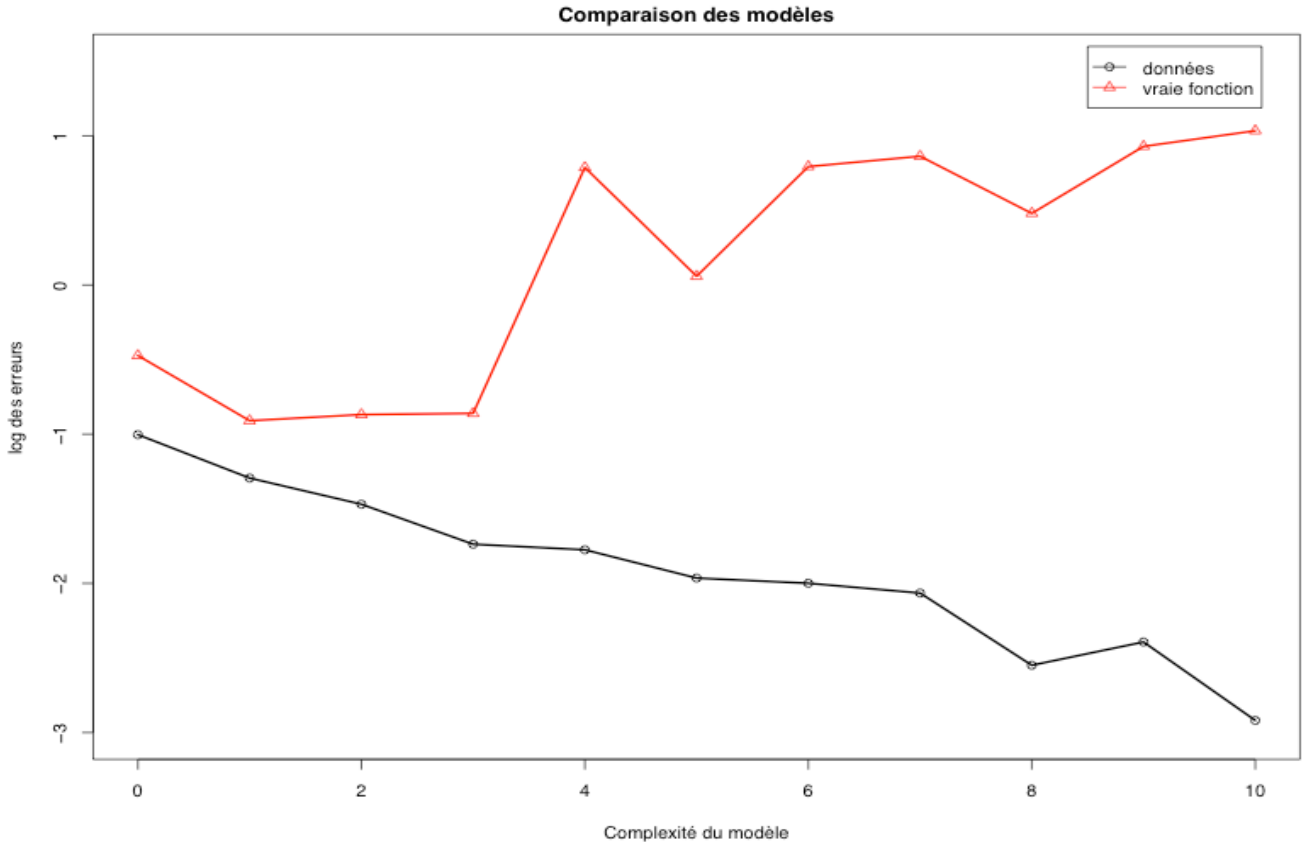


MLP 1-8-1



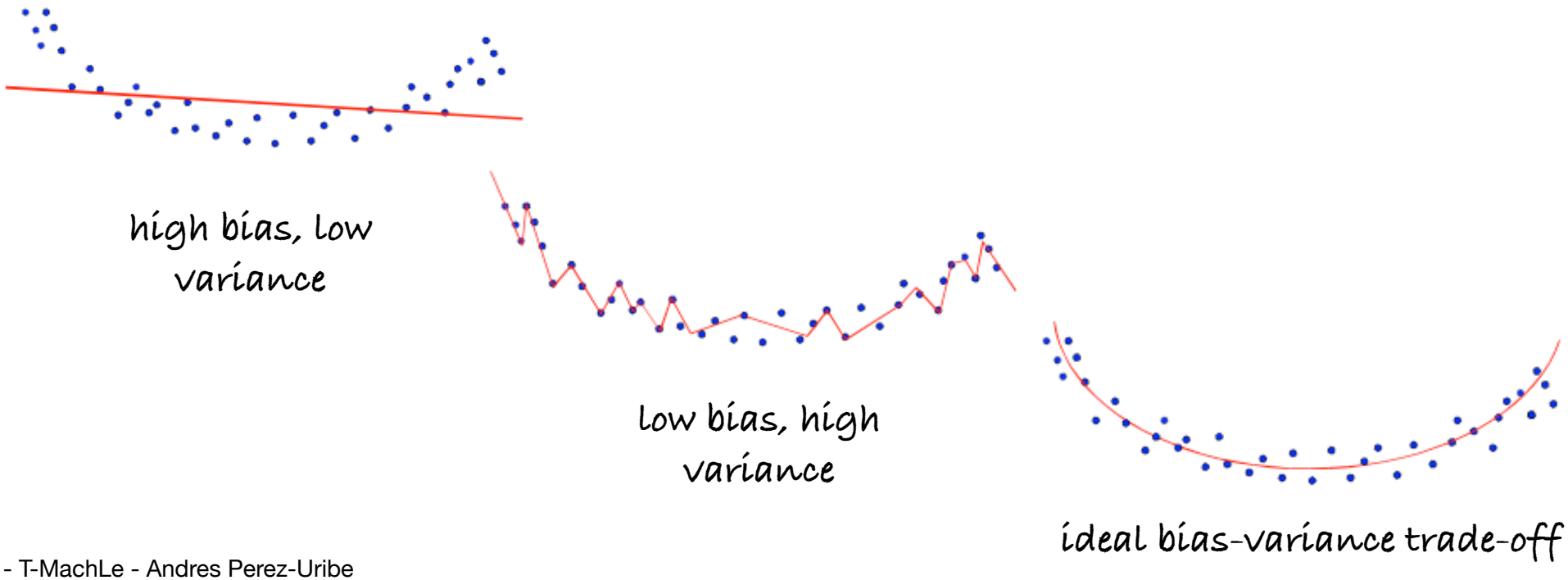
MLP 1-10-1

Error vs model capacity

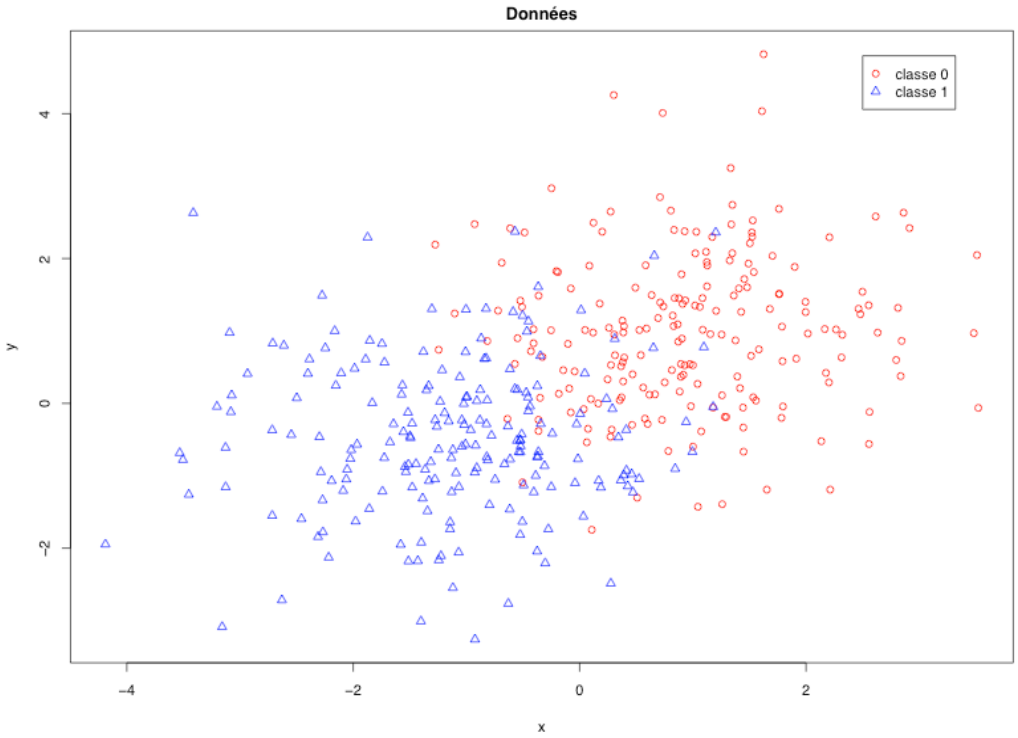


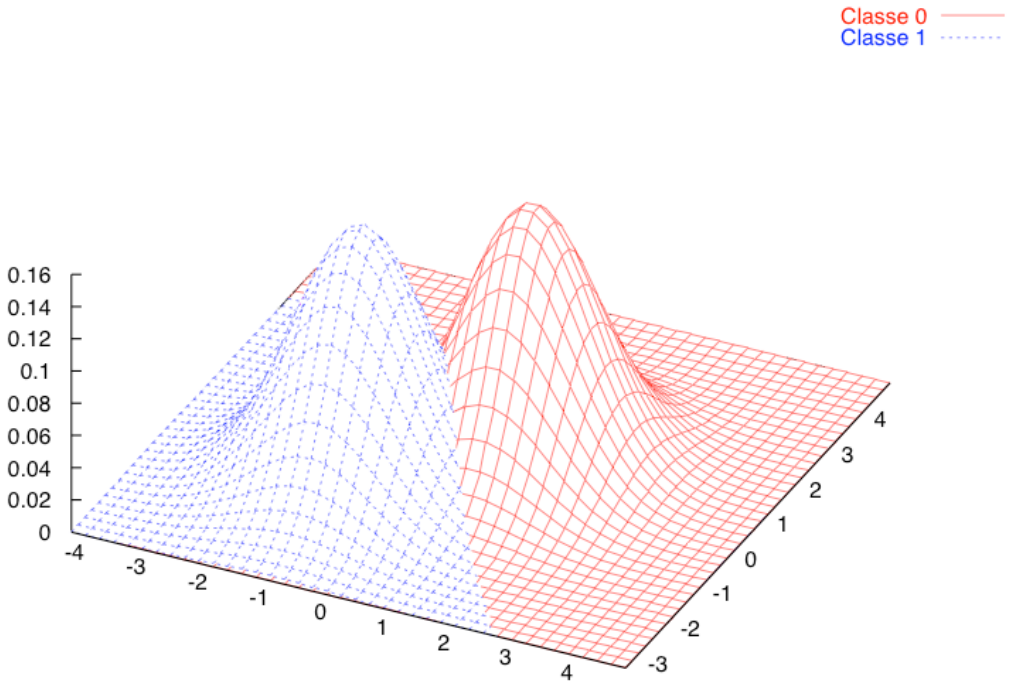
Bias-variance dilemma

- bias: systematic error of the model
- variance: sensibility of the model



The risk of overfitting in a classification problem

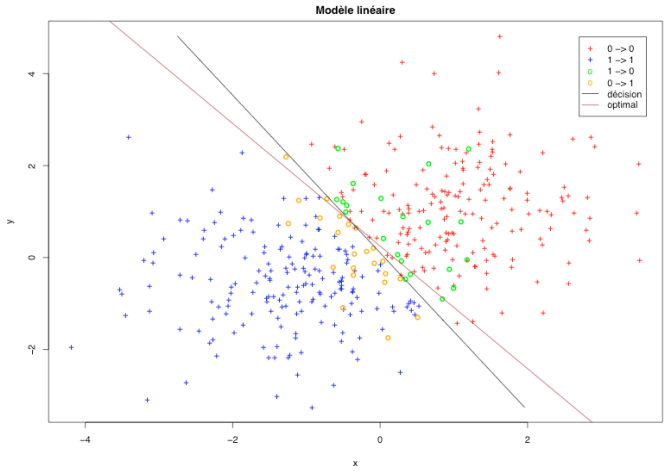




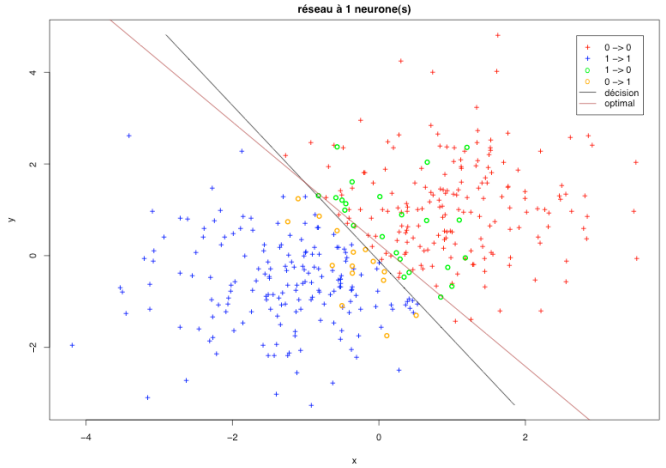
Two-class data points

©F. Rossi, INRIA

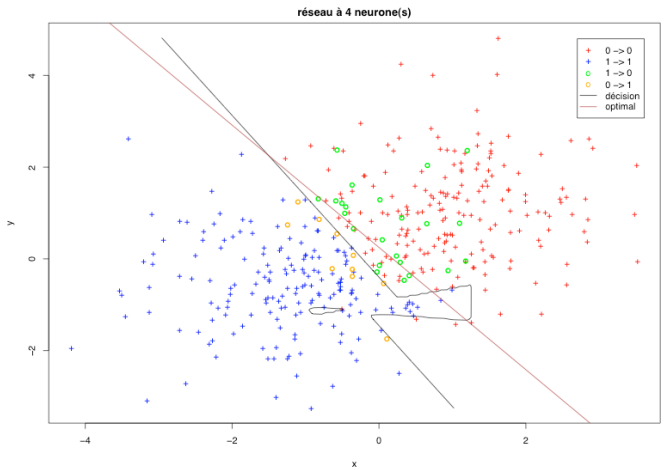
The risk of overfitting (1)



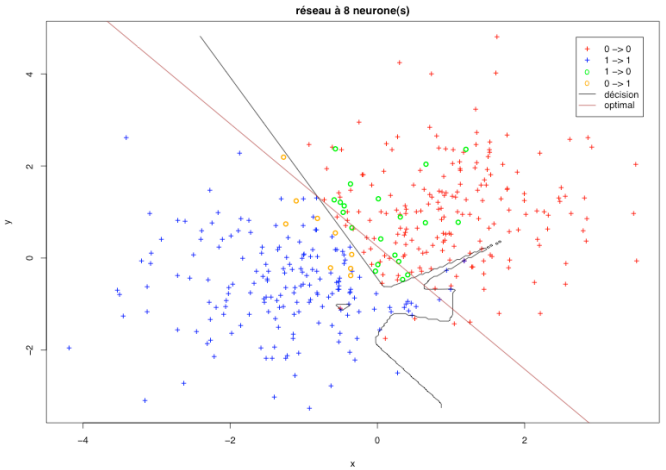
Linear classification



MLP 2-1-1

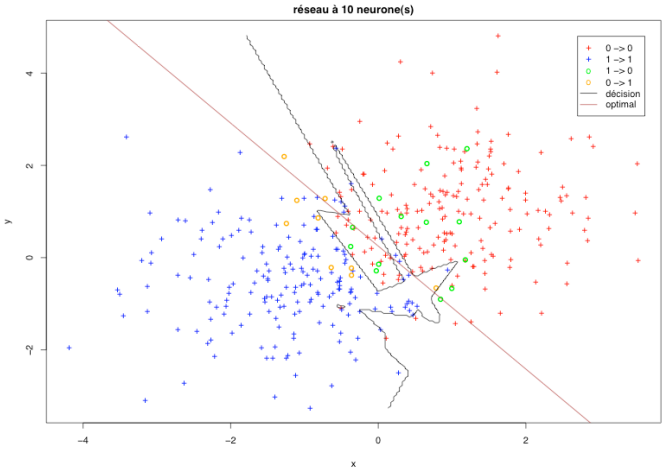


MLP 2-4-1

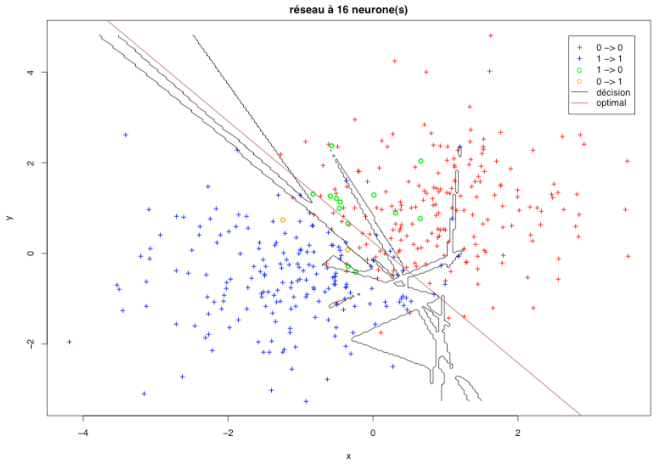


MLP 2-8-1

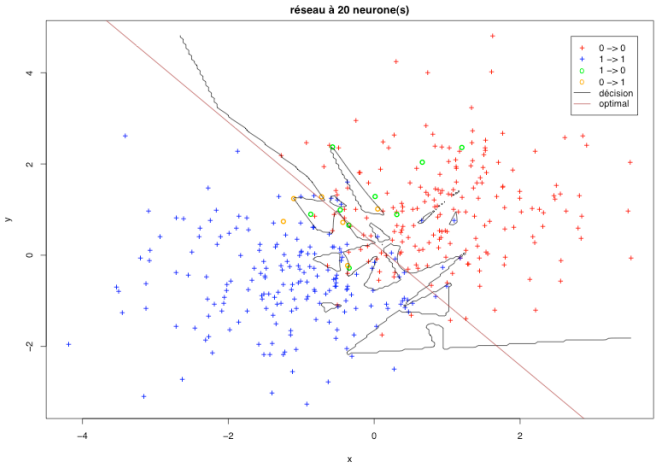
The risk of overfitting (2)



MLP 2-10-1

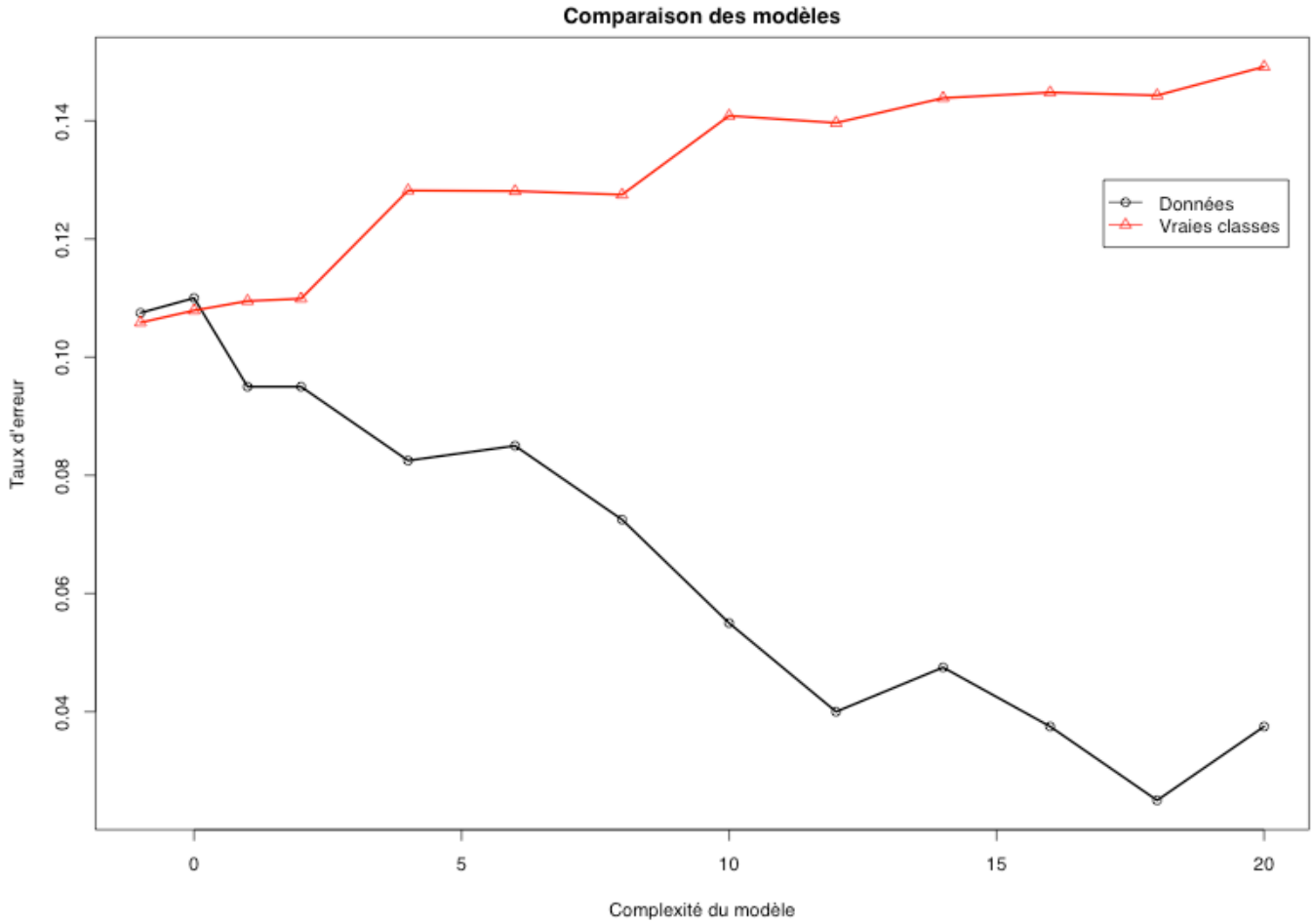


MLP 2-16-1



MLP 2-20-1

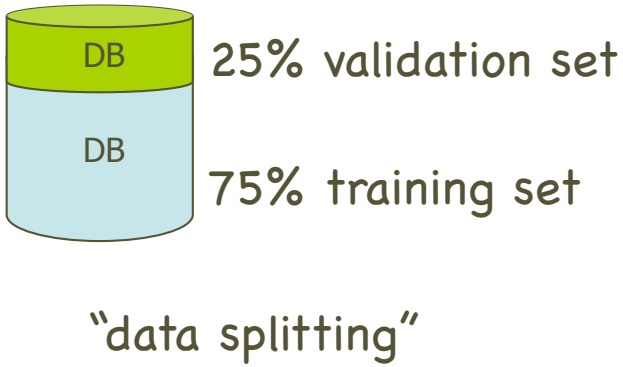
Error vs model capacity

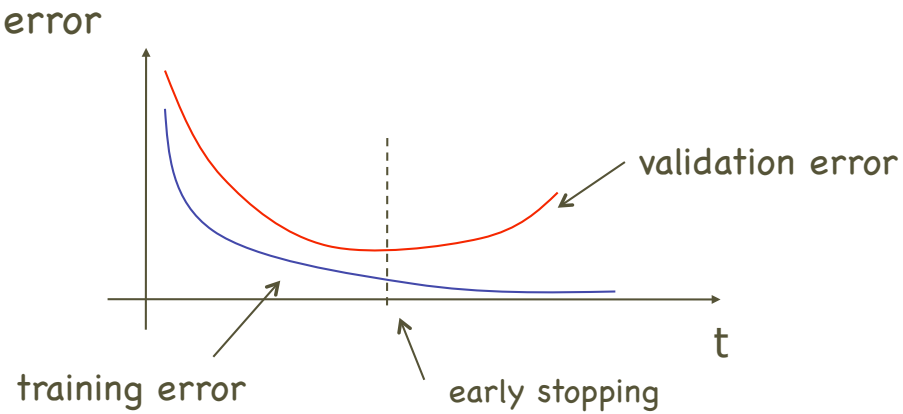


Avoiding overfitting

Avoiding overfitting

- Besides taking care of the number of neurons to avoid overfitting, there are other tricks that can help :)





Early stopping : stop learning when overfitting starts to appear

Regularization

- Regularization means providing constraints to limit the values of the parameters we are learning (e.g., the weights). Example:

Weight decay (or L2 regularization): avoid large weight values

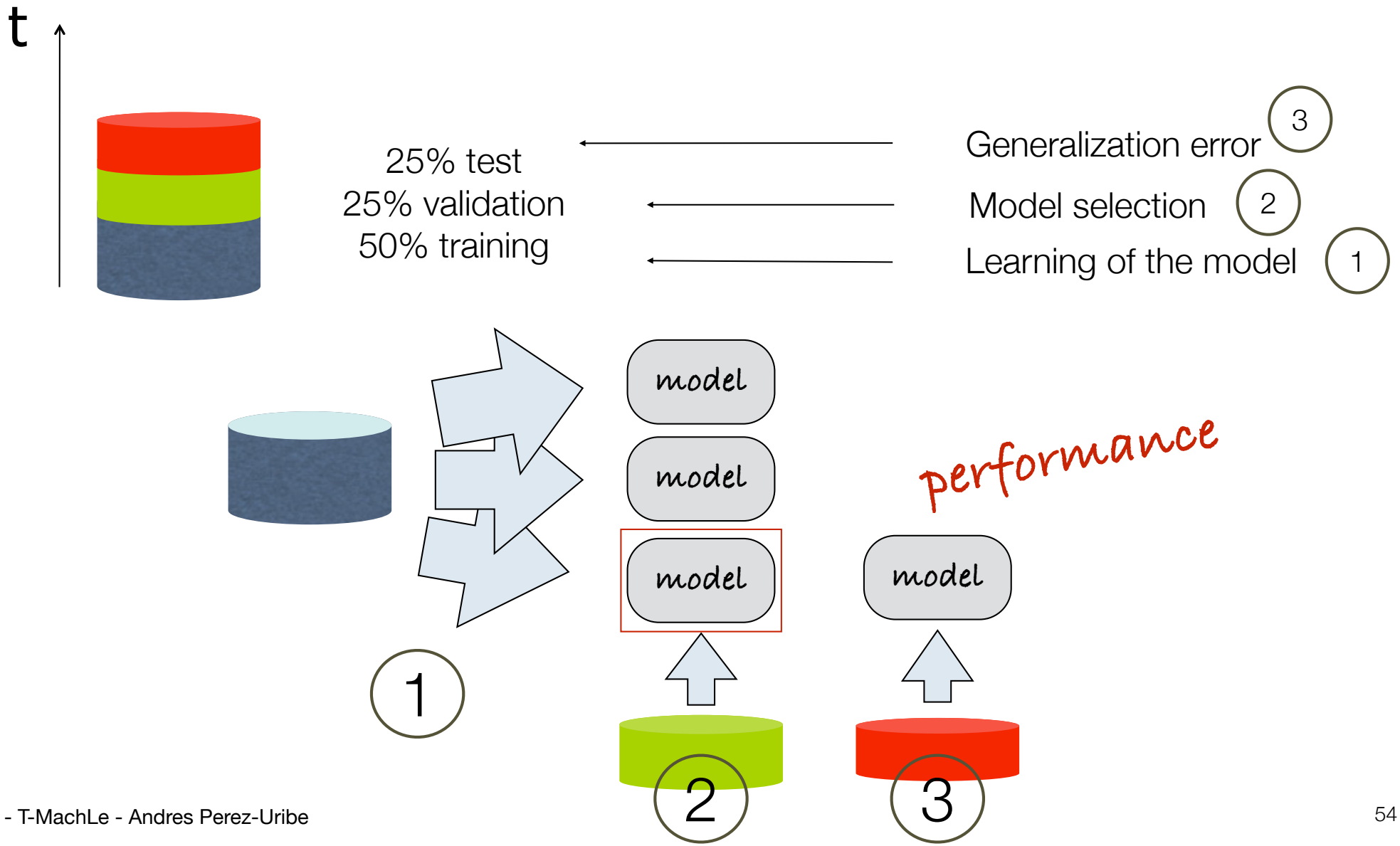
$$pénalité = 1/2 \sum_{ijk} \left[w_{ij}^{(k)} \right]^2$$

Modified objective function: E = E + λpénalité

$$\Delta w_{ij}^{(k)}(t) = \{terme\ standard\} - \eta \lambda w_{ij}^{(k)}$$

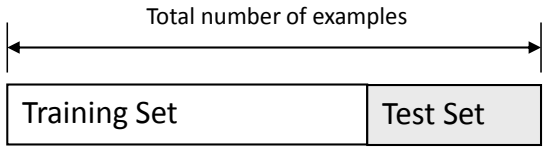
Model selection

Model selection and evaluation



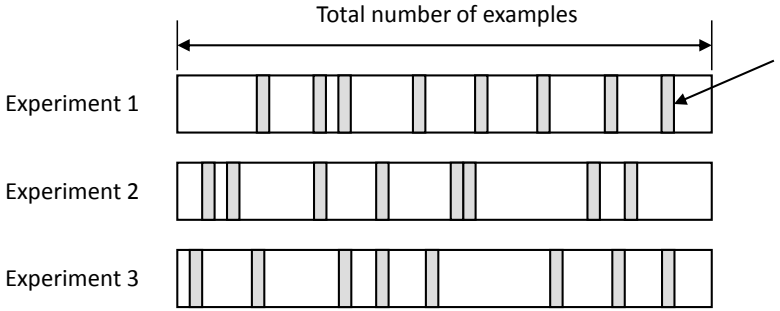
Training & Validation

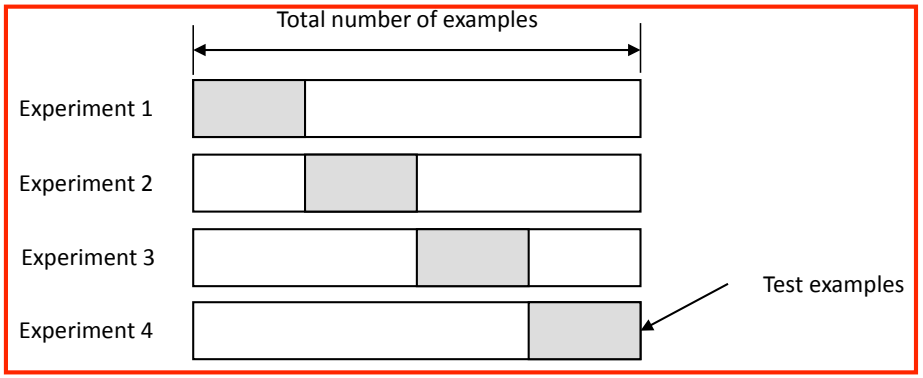
- holdout method



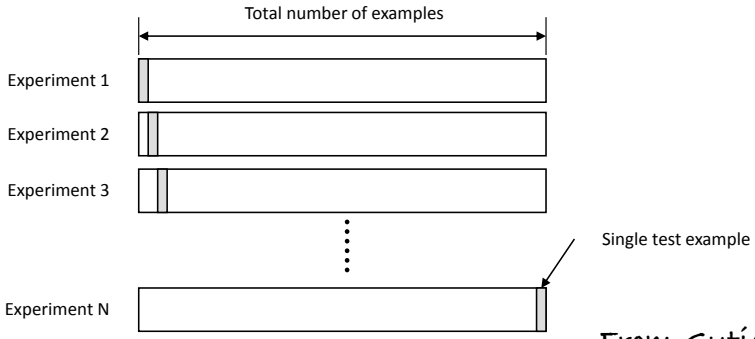
- in general, we do not want to not use available data
- misleading evaluation by unfortunate splitting

- random subsampling & k-fold crossvalidation





- leave-one-out crossvalidation



From Gutierrez-Osuna, TAMU

Hyper-parameter optimization

- Rule of thumb: try several log-spaced values (0.1, 0.01, 0.001) and narrow the grid search to the region where you obtain the lowest validation error. For some parameters use the *3 strategy: e.g., number of hidden neurons: 1, 3, ~10, 30 …
- Scikit-learn implements GridSearchCV and RandomizedSearchCV

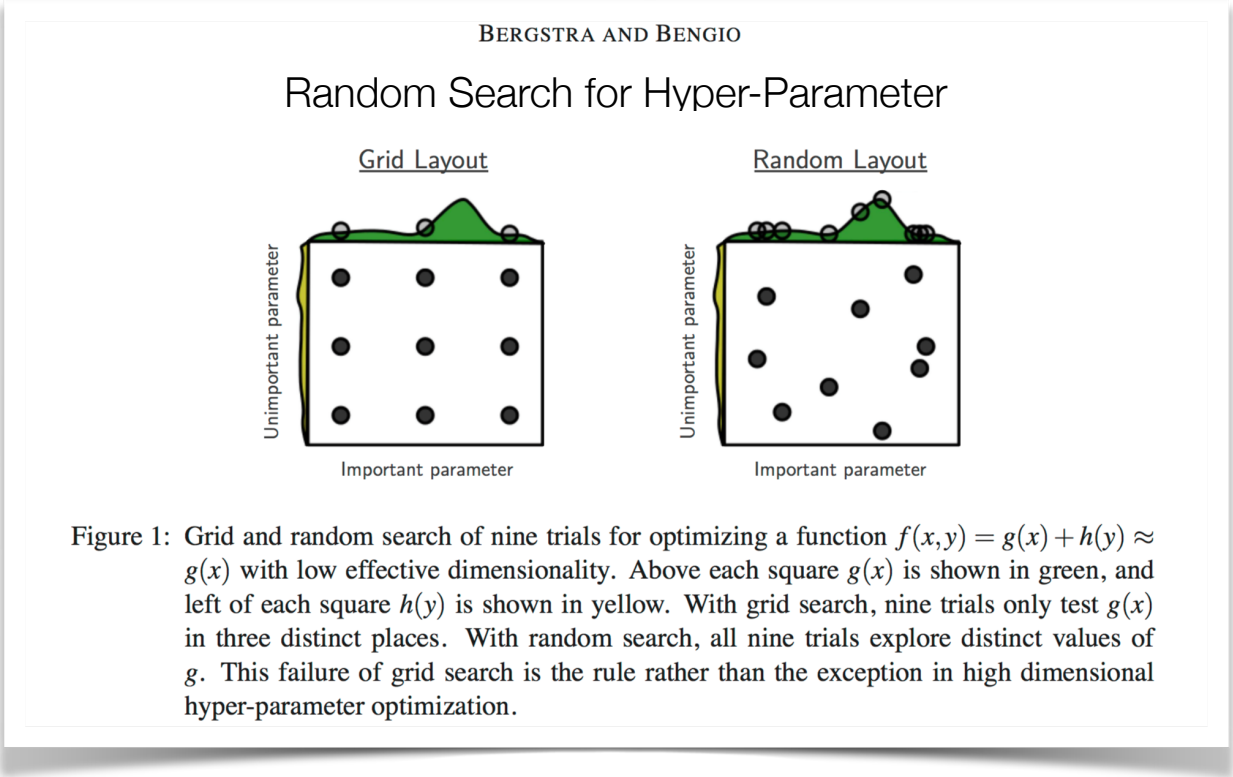
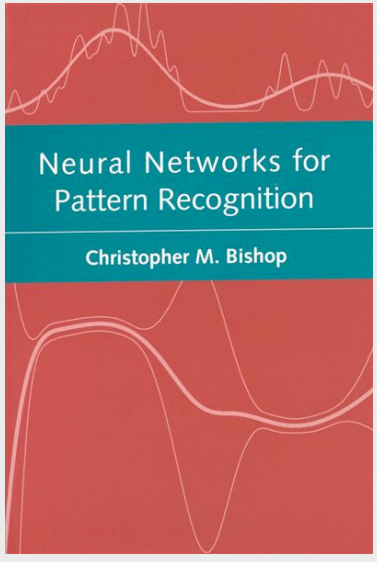
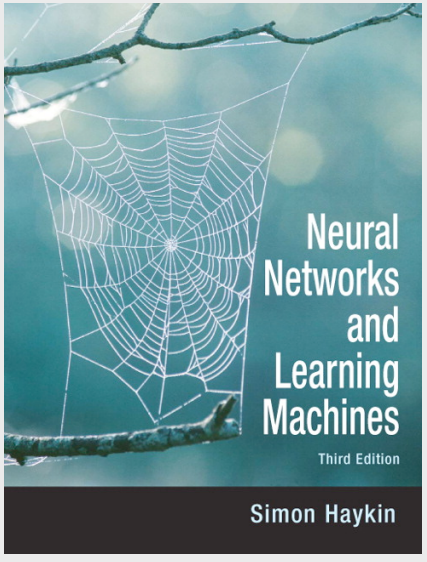
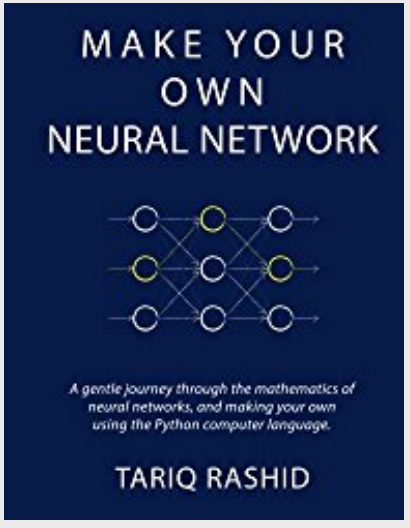
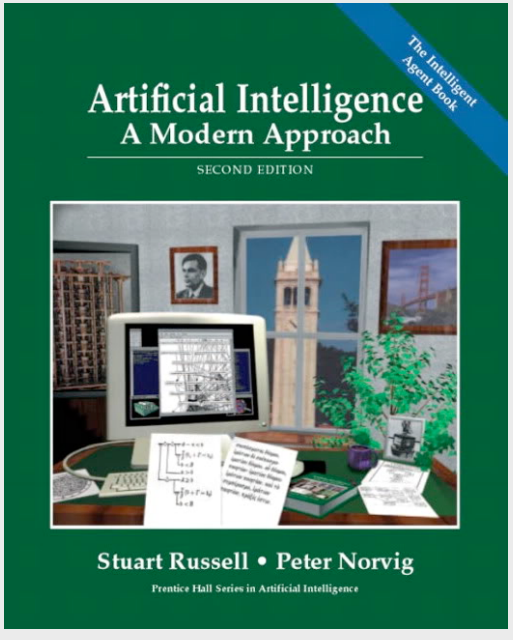


Figure 1: Grid and random search of nine trials for optimizing a function $f(x,y) = g(x) + h(y) \approx g(x)$ with low effective dimensionality. Above each square $g(x)$ is shown in green, and left of each square $h(y)$ is shown in yellow. With grid search, nine trials only test $g(x)$ in three distinct places. With random search, all nine trials explore distinct values of g. This failure of grid search is the rule rather than the exception in high dimensional hyper-parameter optimization.









- Robert Nielsen tutorial: neuralnetworksanddeeplearning.com
- Andrew Ng or Geoffrey Hinton course videos on Coursera
- 3blue1brown videos:
  - What is a NN (18min)
  - How NNs learn (20min)
  - What is Backprop (12min)
  - Backprop calculus (10min)