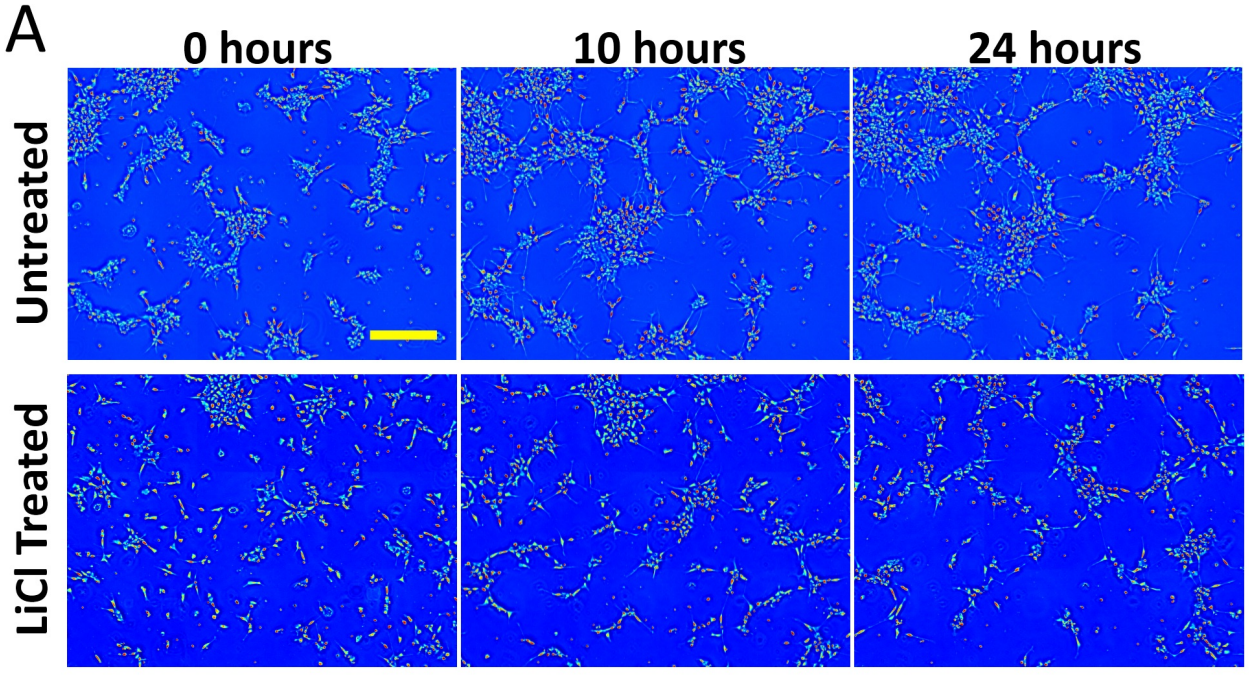
TSIA-SD 210 - Machine Learning

Lecture 6 - Introduction to neural networks and deep learning

Florence d'Alché–Buc
Télécom ParisTech, LTCI
florence.dalche@telecom-paristech.fr

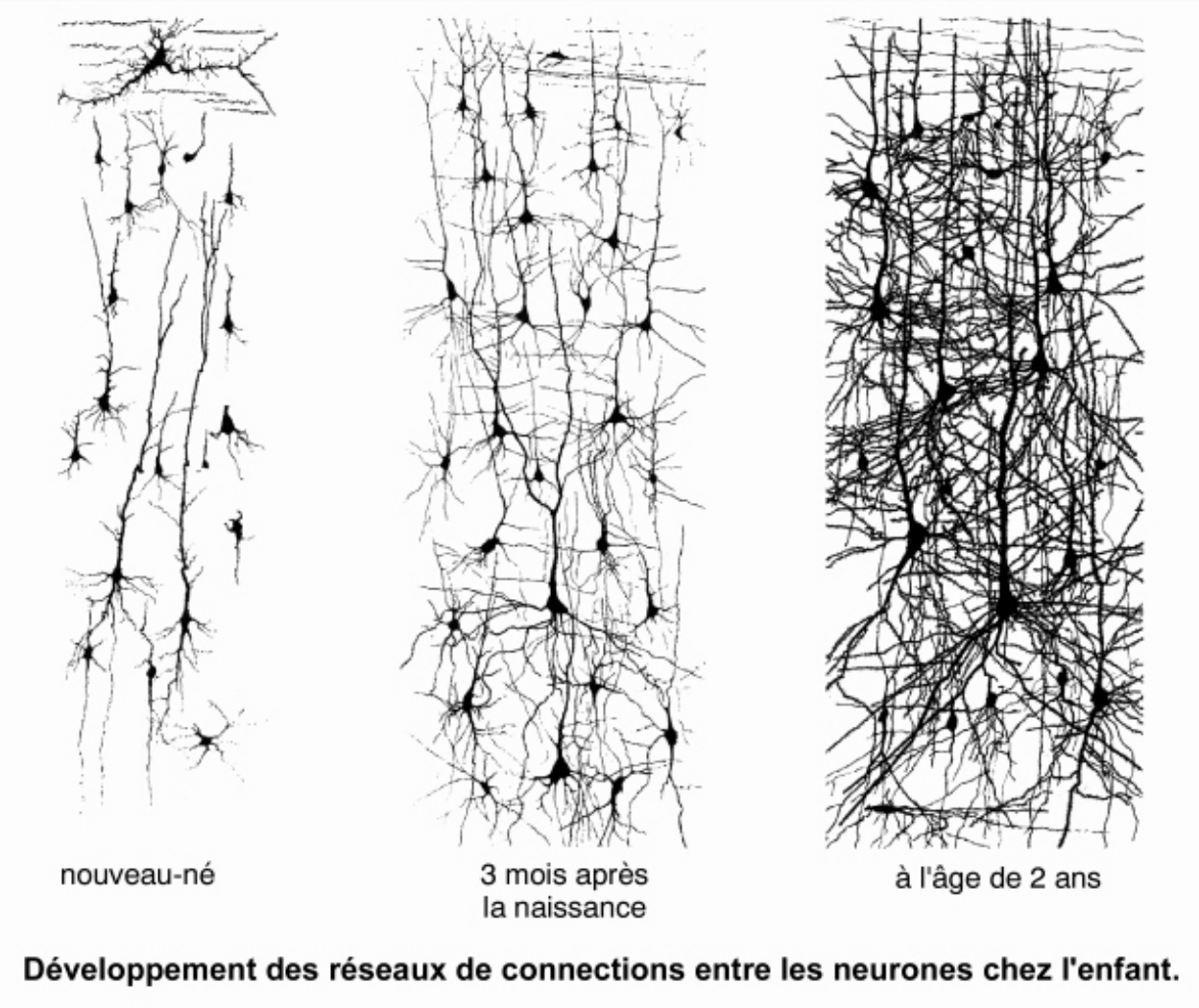
Outline

Neuron network growth over 24 hours



In 2014, the group of Gabriel Popescu at Illinois U. visualized a growing net of baby neurons using spatial light interference microscopy (SLIM). Ref: `http://light.ece.illinois.edu/wp-content/uploads/2014/03/Mir_SRep_2014.pdf`
Video: `https://youtu.be/KjKsU_4s0nE`

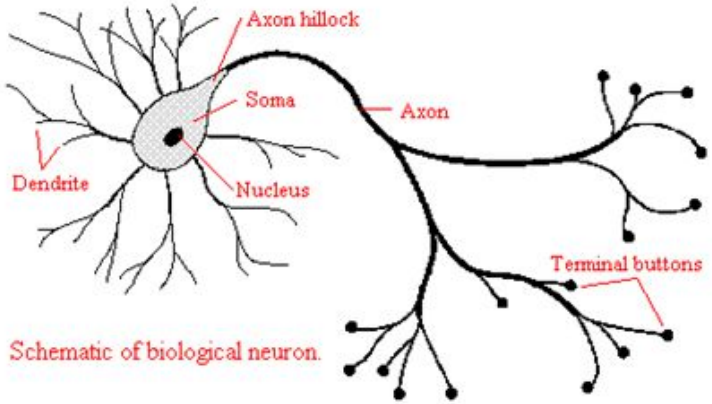
Development of neural networks in children

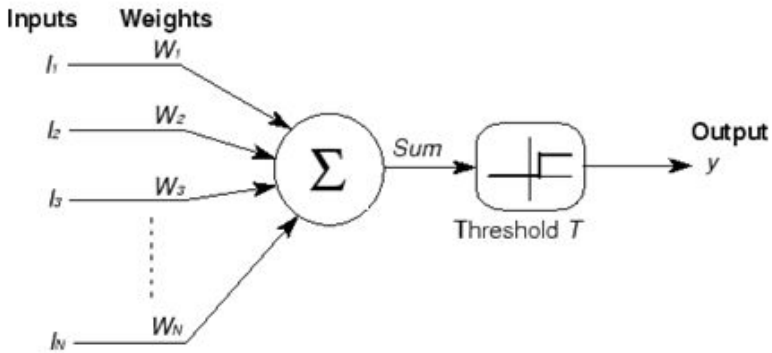


Développement des réseaux de connections entre les neurones chez l'enfant.

Re: Museum de Toulouse `http://www.museum.toulouse.fr/-/connecte-a-vie-notre-cerveau-le-meilleur-des-reseaux-2-`

Le neurone

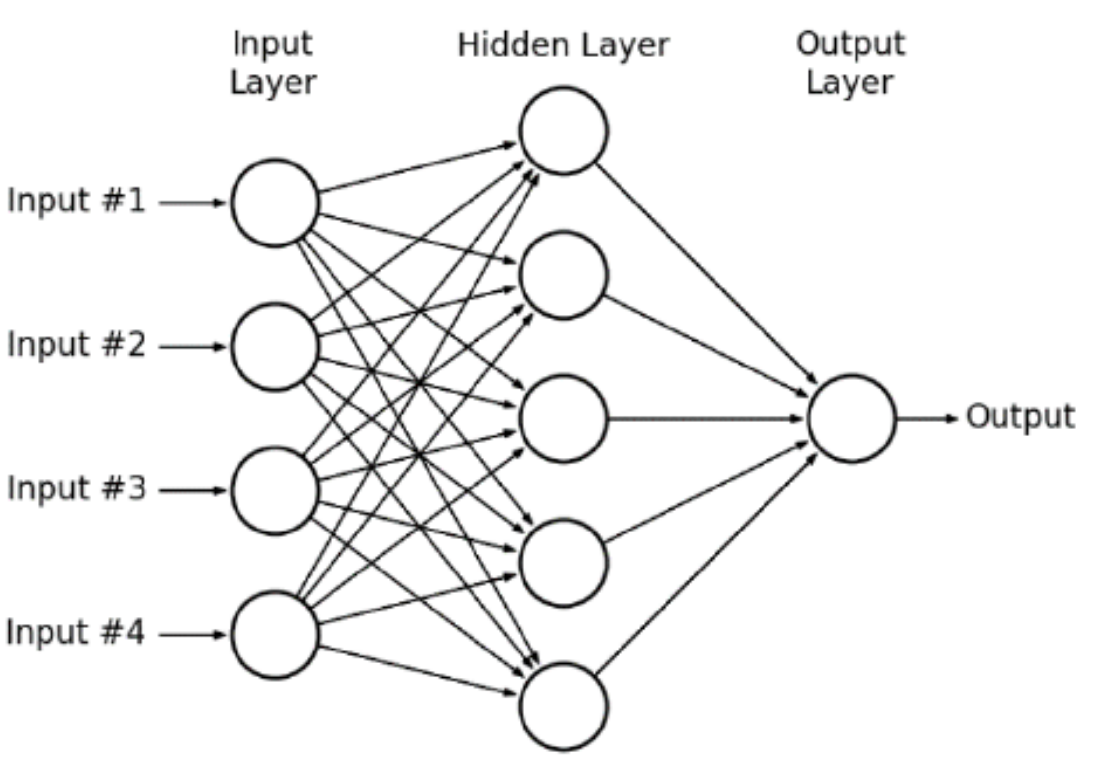




Neurone biologique

Neurone artificiel

Formal neural network: the example of multi-layered perceptron



From formal neuron to formal neural networks 1/2

- Formal neuron: Mc Cullogh et Pitts (Physiologists), 1943
- Learning rule, Rosenblatt, 1957
- Minsky et Papert: limited capacity of a perceptron, 1959
- Learning in MLP by gradient backpropagation, Y. Le Cun, 1985, Hinton et Sejnowski, 1986.
- A one hidden layered perceptron is a universal approximator= Hornik et al. 1991
- Convolutional networks, 1995, Y. Le Cun et Y. Bengio
- From 1995 and 2008, limited expansion of the domain (learning is low, optimization pb are not convex, no theory)

From formal neuron to formal neural networks 2/2

- 2004: Spreading of Graphical Processor Units developed for video games 2005
- Huge dataset of images : Imagenet, Fei-Fei et al. 2008 (now 11 millions of images)
- Deeper neural networks can be trained using these huge datasets
- Initialization of deep networks using unsupervised auto-encoders.
- Learning to represent words: Word2vec (Mikolov et al. 2013)
- Regularization by Dropout (Srivastava et al. 2014)
- 2016: combination of Monte-Carlo Tree search and deep learning

Introduction to neural networks

- Formal neuron and perceptron
- Multi-layered Pereceptron
- Autoencoders
- Convolutional networks (in brief)

Outline

Outline

Outline

A linear classifier: the formal neuron and perceptron

- First model proposed by McCullogh and Pitts (physiologists) in 1943 to model the activity of a neuron
- Input signals represented by a vector $\mathbf{x}$ is processed by a neuron whose weighted synapses are linked to the input
- The neuron computes a weighted sum of the components of the signal
- Rosenblatt proposed a learning rule in 1959

Formal neuron and perceptron

- $h_{perc}(\mathbf{x}) = sign(\mathbf{w}^T\mathbf{x})$
- $sign(a) = 1$ if $a \geq 0$ and -1 otherwise

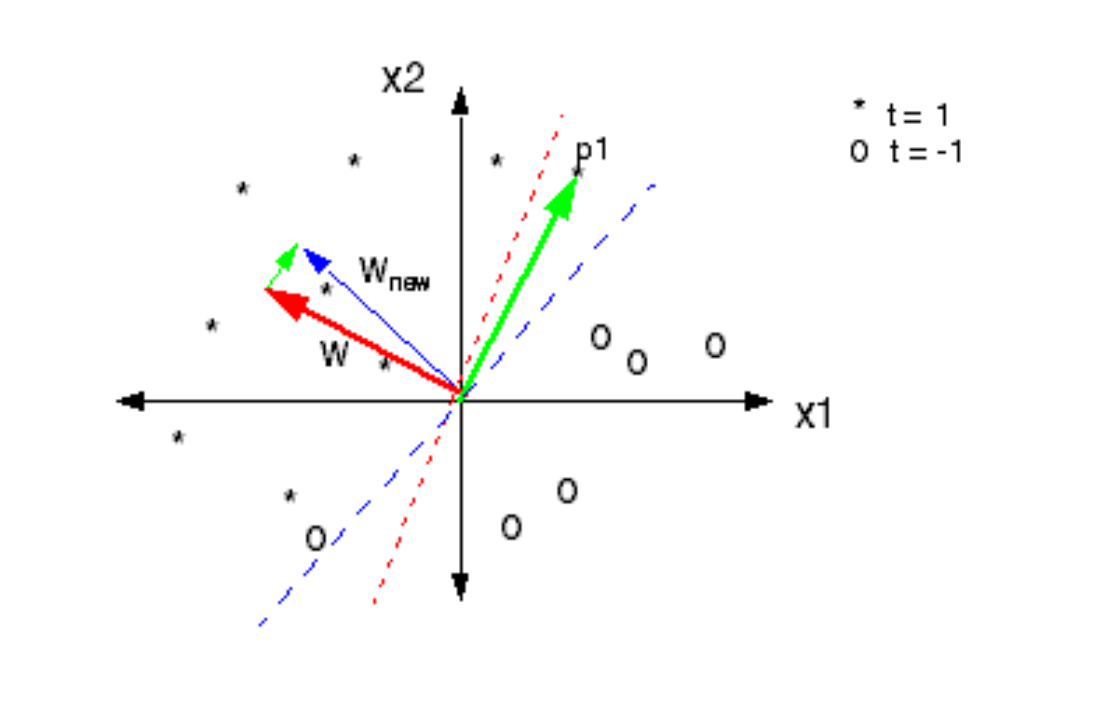
Training data:

- $\mathcal{S} = \{(\mathbf{x_1}, \mathbf{y_1}), ..., (\mathbf{x_n}, \mathbf{y_n})\}$
- $x_i \in \mathbb{R}^{p+1}$: the 0^{th} component is fixed to 1.
- $y_i \in \{-1, +1\}$

Perceptron rule

1. $t = 1$ and Start with the all-zeroes weight vector $\mathbf{w}_1 = 0$, scale the examples to norm 1.
2. Given example x, predict positive class iff $\mathbf{w}_t^T \mathbf{x} > 0$
3. On mistake do:
   - Mistake on positive class: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \mathbf{x}$
   - Mistake on negative class: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \mathbf{x}$
4. $t \leftarrow t + 1$

Correction by a perceptron



Convergence theorem

The algorithm converges if the data are exactly non linearly separable. Here is a slightly more general theorem:

Convergence theorem
Assume there exist a parameter $\mathbf{w}^*$ such that $\|\mathbf{w}\| = 1$, and denote $\gamma > 0$ such that for all $i = 1, \dots n$:

$$y_i(\mathbf{x}_i^T \mathbf{w}^*) \geq \gamma$$

and there exists $R > 0$: $\|\mathbf{x}_i\| \leq R$,
Then the perceptron algorithm converges in at most $\frac{R^2}{\gamma^2}$ iterations.

Proof1/2

- $\mathbf{w}(0) = 0$
- Supposons que la k-ieme erreur est faite sur l'exemple d'indice t, nous avons:

$$\begin{aligned} \mathbf{w}(k+1)^T\mathbf{w}^* &= (\mathbf{w}(k) + y_t\mathbf{x}_t)^T\mathbf{w}^* \\ &= \mathbf{w}(k)^T\mathbf{w}^* + y_t\mathbf{x}_t^T\mathbf{w}^* \\ &\geq \mathbf{w}(k)^T\mathbf{w}^* + \gamma \end{aligned}$$

Par récurrence sur k : $\mathbf{w}(k+1)^T\mathbf{w}^* \geq k\gamma$
(Par Cauchy-Schwartz:
$\|\mathbf{w}(k+1)^T\mathbf{w}^*\| \leq \|\mathbf{w}(k+1)\|\|\mathbf{w}^*\| \leq \|\mathbf{w}(k+1)\|$)
donc : $\|\mathbf{w}(k+1)\| \geq \|\mathbf{w}(k+1)^T\mathbf{w}^*\| \geq k\gamma$

Preuve 2/2

On dérive ensuite une majoration pour $\|\mathbf{w}(k+1)\|$:

$$\begin{aligned} \|\mathbf{w}(k+1)\|^2 &= \|\mathbf{w}(k) + y_t \mathbf{x}_t\|^2 \\ &= \|\mathbf{w}(k)\|^2 + y_t^2 \|\mathbf{x}_t\|^2 + 2 y_t \mathbf{x}_t^T \mathbf{w}(k) \end{aligned}$$

le terme de correction $2 y_t \mathbf{x}_t^T \mathbf{w}(k)$ est par définition négatif donc:

$$\|\mathbf{w}(k+1)\|^2 \leq \|\mathbf{w}(k)\|^2 + R^2$$

Par récurrence sur k, on a :

$$\|\mathbf{w}(k+1)\|^2 \leq kR^2$$

Au final, en prenant les deux inégalités:
on a : $k^2\gamma^2 \leq \|w(k+1)\|^2 \leq kR^2$ donc : $k \leq \frac{R^2}{\gamma^2}$
Si k est borné par $\frac{R^2}{\gamma^2}$, cela veut dire qu'en au plus $\frac{R^2}{\gamma^2}$ itérations, on n'a plus de corrections à faire.

Convergence of perceptron rule

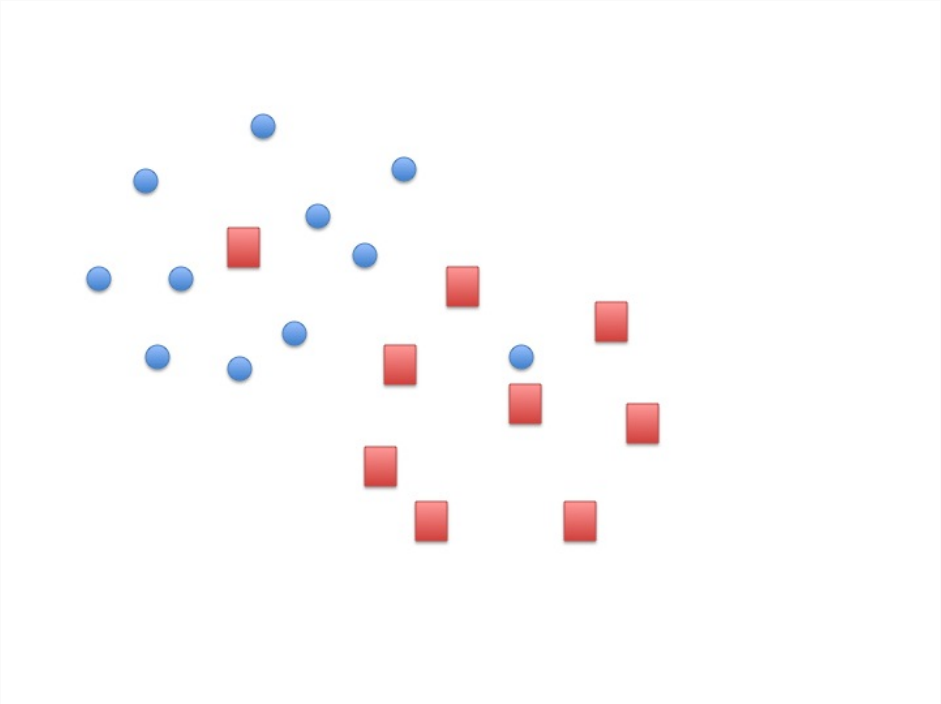
Convergence
The algorithm converges if the data are linearly separable

NonLinear separability:

1. Data almost linearly separable (not separable but Bayes classifier is an hyperplane)
2. Data not linearly separable (Bayes classifier is nonlinear)

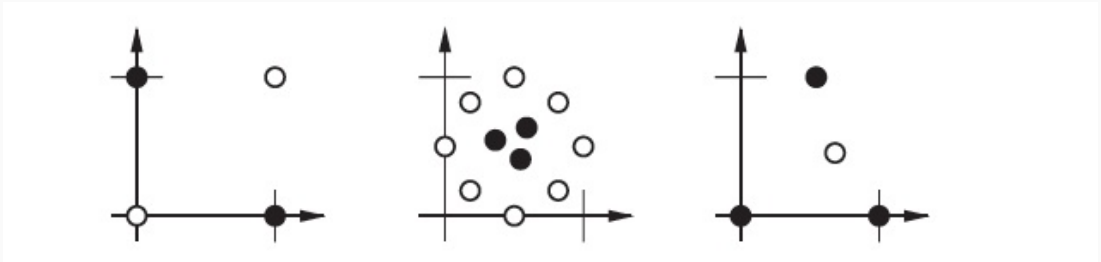
Limitation of a perceptron 1

Example 1 : noisy data :



The algorithm will fail on such data.

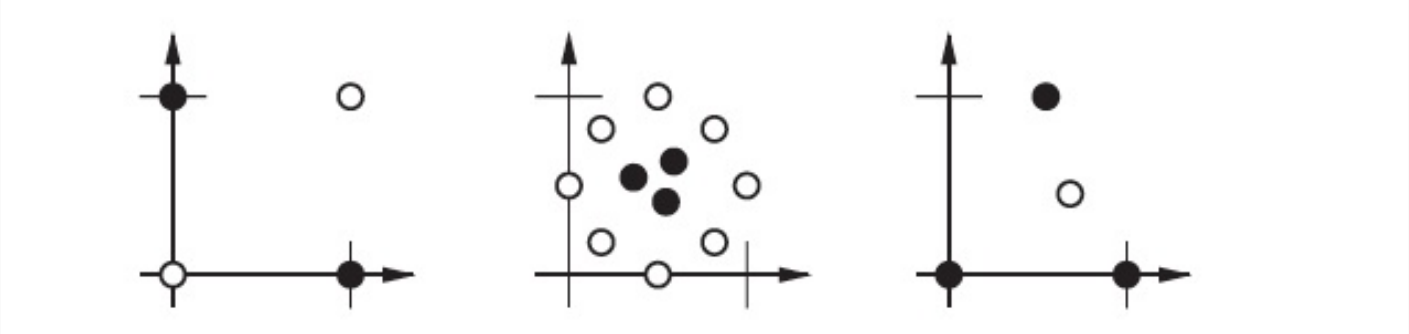
Limitation of a perceptron 2



- XOR problem: a single perceptron cannot approximate a XOR
- Solution :
  - Add additional layers $\rightarrow$ Multi-layered perceptron (with backprop algorithm), Werbos 1974, Le Cun 1985, Rumelhart et al. 1986.
  - Or, use a feature map and transform the data into a space where they become linearly separable

Limitation of the formal neuron

Limited on linearly separable data



Idea: add a layer

$$\begin{aligned} \Phi(x)_1 &= \text{AND}(\bar{x_1}, x_2) \\ \Phi(x)_2 &= \text{AND}(x_1, \bar{x_2}) \end{aligned}$$

Now let us compute :

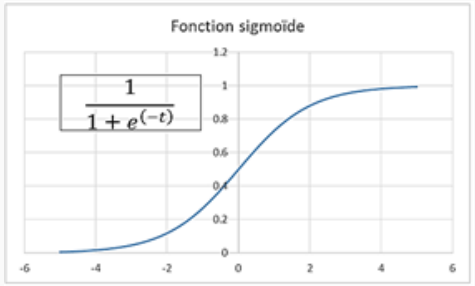
$$f(x) = g(\Phi(x)^T w + b)$$

We talk about feature map or internal representation

To come: Big Advantage of Neural Networks with more than one layer: backpropagation algorithm enables to learn Φ.

A step towards MLP networks)

- Replace the sign function by a (differentiable) sigmoid function
  - $sigm(x) = \frac{1}{1+\exp(-\frac{1}{2}x)}$



- Differentiable loss
  - $\ell_i(\mathbf{w}) = (y_i - \ sigm(\mathbf{w}^T x))^2$
  - $L(\mathbf{w}) = \sum_i \ell_i(\mathbf{w})$
  - Gradient descent

Gradient descent for perceptron

Perceptron algorithm (gradient-like version)

- STOP = false
- ε; nbIter; $j = 0$; $t = 0$
- Initialiser $\mathbf{w}_0$
- Until STOP be TRUE
  - For i from 1 to n:
    - $\mathbf{w^{t+1}} = \mathbf{w^t} - \eta \nabla_{\mathbf{w}} \ell_{\mathbf{i}}(\mathbf{w})$
    - $t \rightarrow t + 1$
- $j \rightarrow j + 1$
- STOP = ($L(||\mathbf{w}(nouveau) - \mathbf{w}(ancien)|| < \varepsilon$) eand ($nbIter \leq nbMax$)

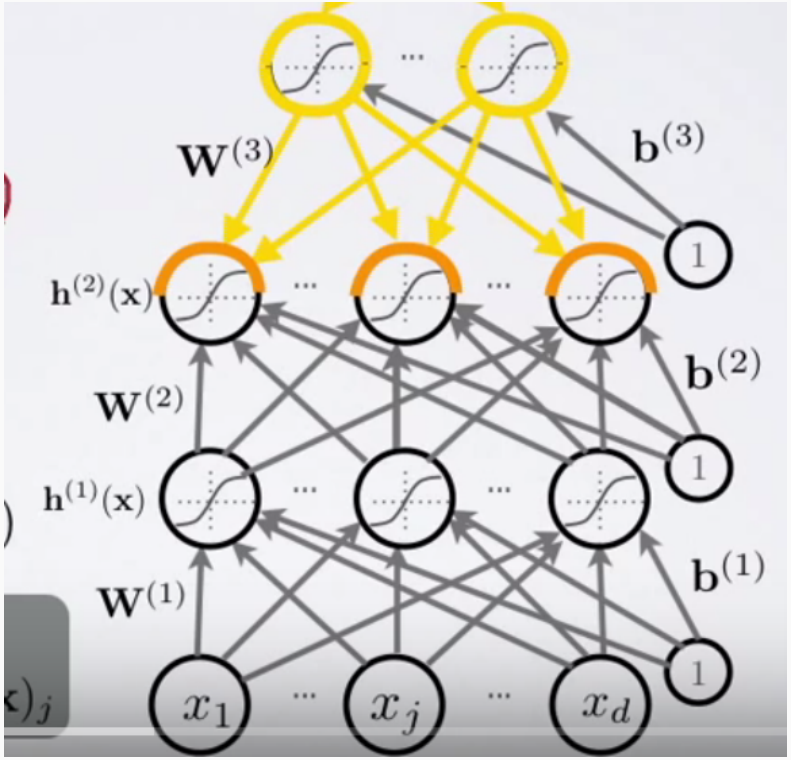
Perceptron

- Early stopping: stop iterations before overfitting
- Avoid overfitting : classic ℓ_2 regularization
  - The loss function becomes $L(\mathbf{w}) = \sum_i \ell_i(\mathbf{w}) + \lambda ||\mathbf{w}||^2$
- Prefer Stochastic Gradient Descent to Gradient Descent

Outline

Outline

Multi-layered perceptron



The function to be learned:

$$\begin{aligned} \mathbf{f}_{MLP}(x) &= \mathbf{h}^{(3)}(x) = \mathbf{o}(\mathbf{a}^{(3)}(x)) \\ \mathbf{a}^{(k)}(x) &= W^{(k)}\mathbf{h}^{(k-1)}(x) + \mathbf{b}^{(k)} \\ \mathbf{h}^{(k-1)}(x) &= g(\mathbf{a}^{(k-1)}(x)) \end{aligned}$$

Activation function

In early times, mainly use of sigmoïdal or hyperbolic tangent functions
Hyperbolic tangent

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

$$g'(a) = 1 - h(a)^2 \quad (2)$$

Sigmoïd function

$$g(a) = \frac{1}{1 + \exp(-a)} \quad (3)$$

$$g'(a) = g(a)(1 - g(a)) \quad (4)$$

When we write: $g(\mathbf{a})$, t means that we apply g to each component of vector $\mathbf{a}$.

NB: the derivations computed from the activation are thus cheap to compute.

Output Activation function for classification

Binary classification

- $o(a) = \frac{1}{1+\exp(-1/2a)}$

Multi-class classification (K classes) : *softmax* activation function

- $\mathbf{o}(\mathbf{a}) = \text{softmax}(\mathbf{a}) = [\frac{\exp(a_1)}{\sum_{c=1}^{C}\exp(a_c)} \cdots \frac{\exp(a_C)}{\sum_{c=1}^{C}\exp(a_c)}]^T$
- Similar to logistic regression for C classes, strictly positive, sum to 1.

Outline

Regularized Empirical risk minimization

As usual define $\mathcal{S} = \{(\mathbf{x}_i, y_i), i = 1, \ldots n\} \subset (\mathbb{R}^p \times \{1, \ldots, C\})$, n-length i.i.d. sample from a fixed but unknown probability distribution $\mathbb{P}_{X,Y}$. W denotes the set of parameters of the feedforward network.

$$\mathcal{L}(W; \mathcal{S}) = \sum_{n=1}^{N} \ell(\mathbf{f}(\mathbf{x}_n), y_n)) + \lambda \Omega(W)$$

Loss function for classification

If we use *softmax* output activation function, we can make the output estimate the posterior probability of each class. In order to get that: $f_c(\mathbf{x}) = \hat{p}(y = c|\mathbf{x})$, we choose to minimize the negative log-likelihood (sometimes referred as cross-entropy):

$$\ell(\mathbf{f}(x), y) = -\sum_{c=1}^{C} 1_{y=c} \log f(y)_c = \log f(\mathbf{x})_y,$$

with y an index of class between 1 and C. Important $\mathcal{L}$ is non-convex and possesses local minima

- The best we can do: find a good local minimum
- This is why for a long period of time (1995-2005) SVMs have been preferred to NNs

Loss function for multiple output regression

In this case we usually we use a *linear* output activation function.

We then minimize the square loss:

$$\ell((\mathbf{x}), \mathbf{y}) = \|\mathbf{y} - (\mathbf{x})\|^2$$

Sitll due to the nature of computations in the network, $\mathcal{L}$ is non-convex and possesses local minima

- The best we can do: find a good local minimum
- This is why for a long period of time (1995-2005) SVMs have been preferred to NNs

Regularization term

Typically, a ℓ_2 regularization is chosen:

$$\Omega(W) = \sum_{k,i,j} (W_{i,j}^{(k)})^2 \tag{5}$$

Outline

Stochastic Gradient descent

Parameter to learn: $\theta = (W^{(1)}, b^{(1)}, \ldots, W^{(M)}, b^{(M)})$ for a M-layered network

Basic idea of the algorithm:

- Initialize parameter θ^0, stopping criterion, epoch=0
- Repeat
  - N times : uniformly pick a training example $(\mathbf{x}^t, y^t)$ from $\mathcal{S}$
    - Compute $\Delta = -\nabla_\theta \ell(\mathbf{f}_\theta(\mathbf{x}^t), y^t))) - \lambda \nabla_\theta \Omega(\theta)$
    - Make a correction: $\theta \leftarrow \theta + \alpha\Delta$
  - epoch $\leftarrow$ epoch + 1
  - Update stopping criterion
  - Until stopping criterion is met of epoch reaches a max.

Backpropagation of the gradient

Use the chain rule to propagate errors on hidden parameters:
Références:

- Y. LeCun: Une procédure d'apprentissage pour réseau à seuil asymmétrique (a Learning Scheme for Asymmetric Threshold Networks), Proceedings of Cognitiva 85, 599-604, Paris, France, 1985.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986) Learning representations by back-propagating errors. Nature, 323, 533–536.

Reminder: One-hidden-layer feedforward architecture

$$\mathbf{f}_{MLP}(\mathbf{x}) = \mathbf{h}^{(2)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(2)}(\mathbf{x})) \tag{6}$$

$$\mathbf{a}^{(2)}(x) = W^{(2)}\mathbf{h}^{(1)}(\mathbf{x}) + \mathbf{b}^{(2)} \tag{7}$$

$$\mathbf{h}^{(1)}(x) = g(\mathbf{a}^{(1)}(\mathbf{x})) \tag{8}$$

$$\mathbf{a}^{(1)}(x) = W_j^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \tag{9}$$

Which gradient to compute ? Local loss

Output layer

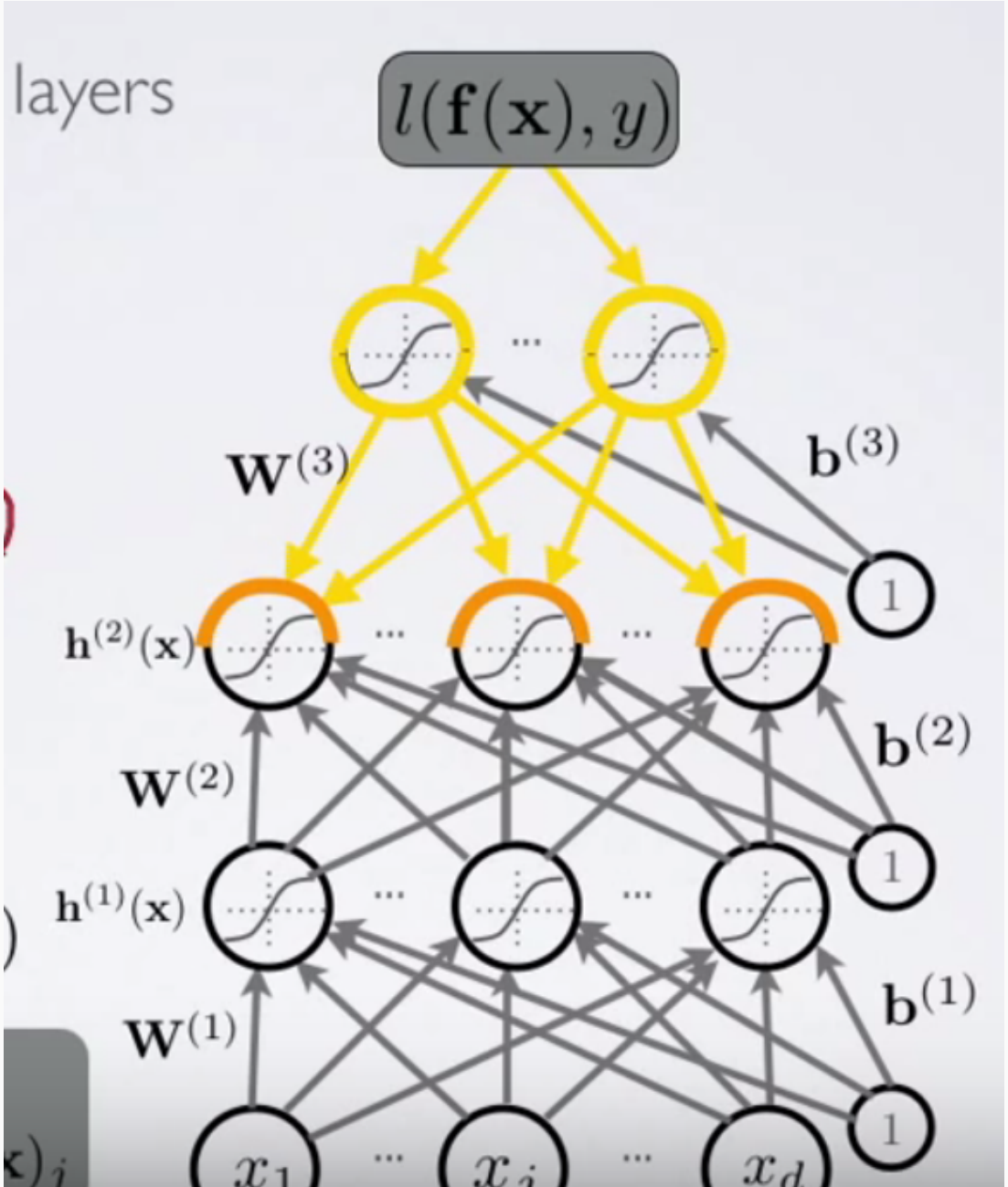
- $\nabla_{W^{(2)}} \ell(y, \mathbf{f}(x))$
- $\nabla_{\mathbf{b}^{(2)}} \ell(y, \mathbf{f}(x))$

Hidden layer: for each $j = 1, \ldots, M$

- $\nabla_{W_j^{(1)}} \ell(y, \mathbf{f}(x))$
- $\nabla_{\mathbf{b}_j^{(1)}} \ell(y, \mathbf{f}(x))$

However , we will introduce general corrections

How to backpropagate corrections in the network ?



The output layer

$$\frac{\partial \ell(y, \mathbf{f}(x))}{\partial f(x)_c} = \frac{\partial(-\log f(x)_y)}{\partial f(x)_c} = \frac{-1_{(y=c)}}{f(x)_y}$$

Gradient is now:

$$\nabla_{\mathbf{f}(x)}(-\log f(x)_y) = -\frac{\mathbf{e}(y)}{f(x)_y}$$

with $\mathbf{e}(y)^T = [1_{y=1} \dots 1_{y=C}]$.

The output layer: pre-activation

$$\begin{aligned}\frac{\partial f(x)_y}{\partial a^{(2)}(x)_c} &= \frac{-1}{f(x)_y}\frac{\partial \mathbf{o}(\mathbf{a}^{(2)}(x))_y}{\partial a^{(2)}(x)_c} \\ &= -(1_{y=c} - f(x)_c)\end{aligned}$$

Chain rule in the network

Use $\frac{\partial p(a)}{\partial a} = \sum_i \frac{\partial p(a)}{\partial q_i(a)} \frac{\partial q_i(a)}{\partial a}$
The chain rule can be invoked in the network:

- set a to a unit in a given layer
- $q_i(a)$ to a pre-activation in the layer above
- $p(a)$, being the loss function

Gradient computation: loss gradient at hidden layer

Partial derivatives

$$
\begin{aligned}
-\frac{\partial \log f(x)_y}{\partial h^{(k)}(x)_j} &= \sum_i -\frac{\partial \log f(x)_y}{\partial a^{(k+1)}(x)_i} \frac{\partial a^{(k+1)}(x)_i}{\partial h^{(k)}(x)_j} \\
&= \sum_i -\frac{\partial \log f(x)_y}{\partial a^{(k+1)}(x)_i} W^{(k+1)}_{i,j} \\
&= W_j^T (\nabla_{a^{(k+1)}(x)} - \log f(x)_y)
\end{aligned}
$$

Now, gradient:

$$
\nabla_{h^{(k)}(x)}(-\log f(x)_y) = (W^{(k+1)})^T (\nabla_{a^{(k+1)}(x)} - \log f(x)_y)
$$

NB: $a^{(k)}(x)_i = b_i^{(k)} + \sum_j W_j^{(k)} h^{(k-1)}(x)_j$

Loss gradient at hidden layers: pre-activation

Partial derivatives:

$$
\begin{aligned}
-\frac{\partial \log f(x)_y}{\partial a^{(k)}(x)_j} &= -\frac{\partial \log f(x)_y}{\partial h^{(k)}(x)_j} \frac{\partial h^{(k)}(x)_j}{\partial a^{(k)}(x)_j} \\
&= -\frac{\partial \log f(x)_y}{\partial h^{(k)}(x)_j} g'(a^{(k)}(x)_j)
\end{aligned}
$$

NB: $h^{(k)}(x)_j = g(a^{(k)}(x)_j)$

Loss gradient at hidden layers: pre-activation

Gradient:

$$\begin{aligned}\nabla_{\mathbf{a}^{(k)}(x)} - \log f(x)_y &= (\nabla_{\mathbf{h}^{(k)}(x)} - \log f(x)_y)^T \nabla_{\mathbf{a}^{(k)}(x)} \mathbf{h}^{(k)}(x) \\ &= (\nabla_{\mathbf{h}^{(k)}(x)} - \log f(x)_y) \odot [\ldots, g'(a^{(k-1)}(x)_j), \ldots]\end{aligned}$$

Note that this term $\nabla_{\mathbf{a}^{(k)}(x)} \mathbf{h}^{(k)}(x)$ is a Jacobian matrix, but here it is a diagonal matrix.

Now ready to compute loss gradient of parameters

Partial derivatives

$$\frac{\partial - \log f(x)_y}{\partial W_{ij}^{(k)}} = \frac{\partial - \log f(x)_y}{\partial a^{(k)}(x)_i} \frac{\partial a^{(k)}(x)_i}{\partial W_{ij}^{(k)}}$$

$$\frac{\partial a^{(2)}(x)_i}{\partial W_{ij}^{(2)}} = h^{(k-1)}(x)_j$$

Gradient (weight matrix)

$$\nabla_{W^{(k)}} - \log f(x)_y = (\nabla_{\mathbf{a}^{(k)}(x)} - \log f(x)_y)\mathbf{h}^{(k-1)}(x)^T$$

Now ready to compute loss gradient of biases

Partial derivatives

$$\begin{aligned}\frac{\partial(-\log f(x)_y)}{\partial b_i^{(k)}} &= -\frac{\partial \log f(x)_y}{\partial a^{(k)}(x)_i}\frac{\partial a^{(k)}(x)_i}{\partial b_i^{(k)}} \\ &= -\frac{\partial \log f(x)_y}{\partial a^{(k)}(x)_i}\end{aligned}$$

Gradient (bias vector):

$$\nabla_{b^{(k)}}(-\log f(x)_y) = \nabla_{a^{(k)}(x)} - \log f(x)_y$$

Now Backpropagation Algorithm

- compute output gradient (before activation)

$$\nabla_{\mathbf{a}^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y \Longleftarrow -(\mathbf{e}(y) - \mathbf{f}(\mathbf{x}))$$

- for k from $L+1$ to 1
  - compute gradients of hidden layer parameter

$$\nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y \Longleftarrow \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \mathbf{h}^{(k-1)}(\mathbf{x})^\top$$

$$\nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y \Longleftarrow \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

  - compute gradient of hidden layer below

$$\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \Longleftarrow \mathbf{W}^{(k)^\top} \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right)$$

  - compute gradient of hidden layer below (before activation)

$$\nabla_{\mathbf{a}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \Longleftarrow \left(\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \odot [\ldots, g'(a^{(k-1)}(\mathbf{x})_j), \ldots]$$

credits/ Hugo Larochelle

- $\nabla_{W^{(k)}} \Omega(W) = 2W^{(k)}$

Model selection

- Nb of hidden layers
- Size of hidden layers
- parameter λ
- nb_{epoch}, ε
- learning gain

Most of them are found using cross-validation

Feedforward networks: pros and cons

Pro

- Flexibility in terms of outputs
- Universal approximator (one-hidden layer)
- Training algorithm since 1985
- Stochastic Gradient algorithm fits Big Data
- Benefit from GPU
- PLUG and PLAY: compose various schemes, extend to time-series

Feedforward networks: pros and cons

Cons

- Non convex loss
- Gradient descent requires much tuning
- Theoretical framework: recent studies on asymptotic behaviour of deep networks
- Many developments *ad hoc*

Outline

Universal approximator

In1990, Hornik et al. show that the family of perceptron with one hidden layer with $p+1$ inputs is dense in the space of continuous functions from a compact subset of $\mathbb{R}^p$ to $\mathbb{R}$.
ref:

- Hornik et al. in Neural networks, 1990
- ▸ Readmore

  Blog: `http://mcneela.github.io/machine_learning/2017/03/21/Universal-Approximation-Theorem.html`

Universal approximator theorem

Theorem (Hornik et al. 1990)
Let $\varphi(\cdot)$ be a nonconstant, bounded, and monotonically-increasing continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of continuous functions on I_m is denoted by $\mathcal{C}(I_m)$.

Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ where$i = 1, \cdots, N$ such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right)$$

as an approximate realization of the function f where f is independent of φ that is,

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$. In other words, functions of the form F are dense in $\mathcal{C}(I_m)$.

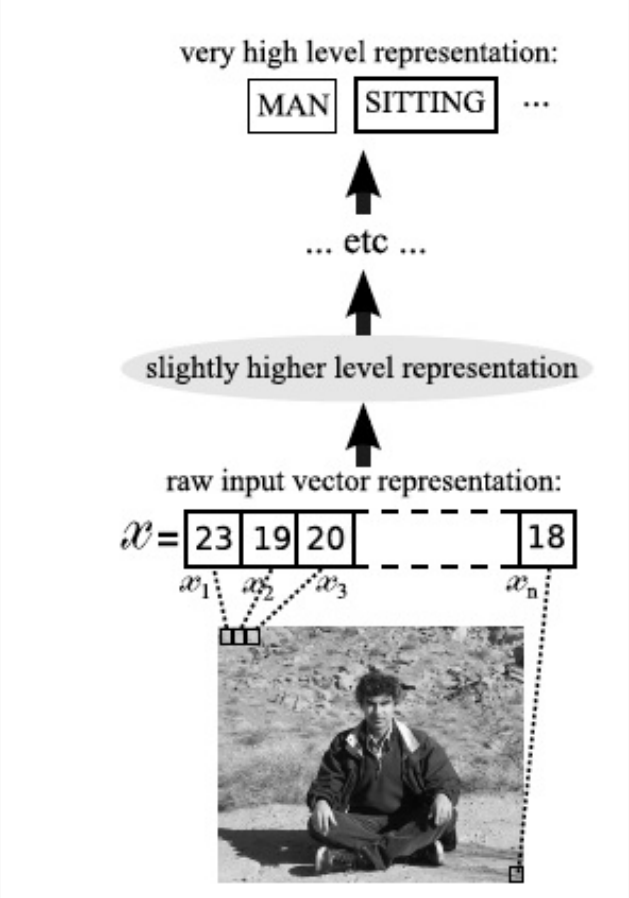
NB : This still holds when replacing I_m with any compact subset of $\mathbb{R}^m$.

Outline

Outline

From neural networks to deep learning

Image Y. Bengio



Deep learning

If several hidden layers, we talk about "deep learning". This type of network is usually relevant for complex data such as images or documents.

Why using more than one hidden layer ?

Even if a one-layered network is a universal approximator, it does not mean that a one-layered network has the best performance on a given problem. Deeper networks can provide a better data representation.

How to learn deep networks ? Not too long ?

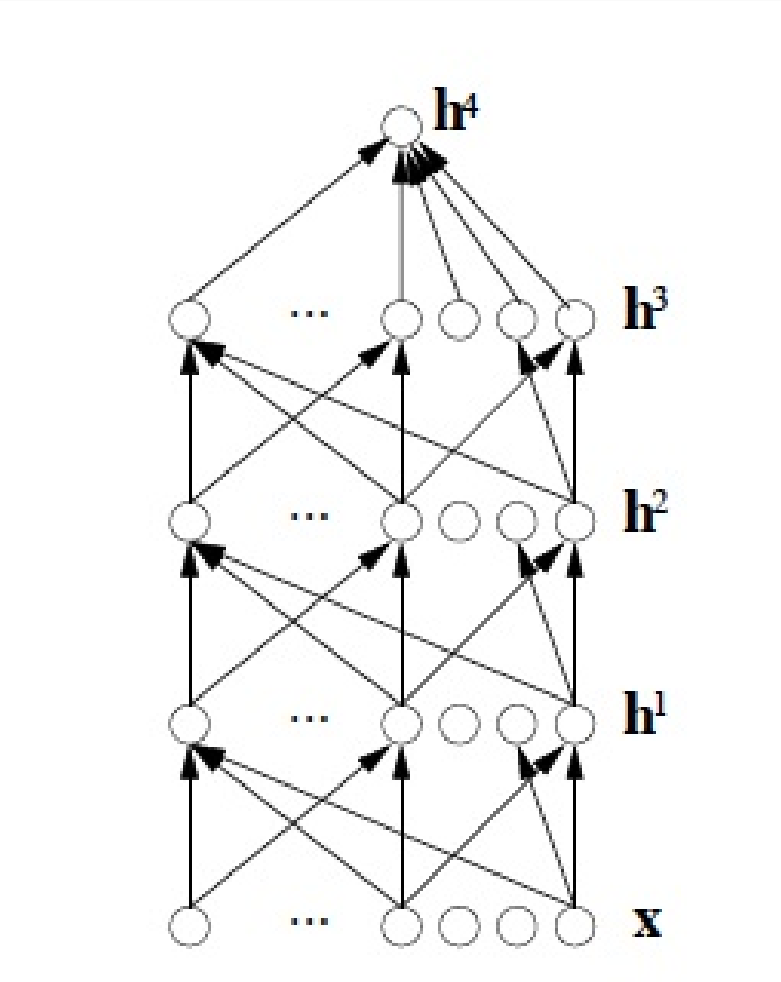
Even if there is an overfitting risk

two good reasons that make possible deep learning

- tremendous improvement of computational capacity (GPU)
- availability of huge datasets (Imagenet, Fei-Fei, 2008)

However learning a deep network is not that easy: the network falls into local minima very easily (Bengio et al. 2007; Erhan et al. 2009).

Learning Deep networks

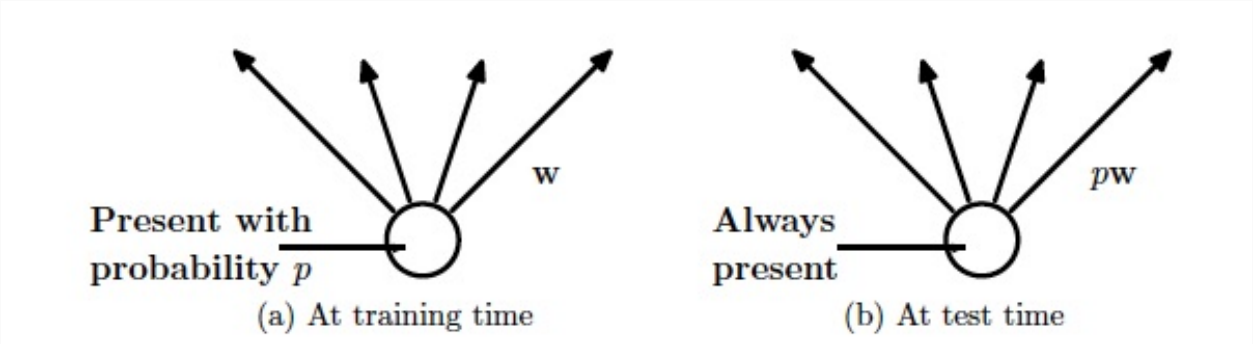


Learning Deep networks

- Dropout
- Auto-encoders

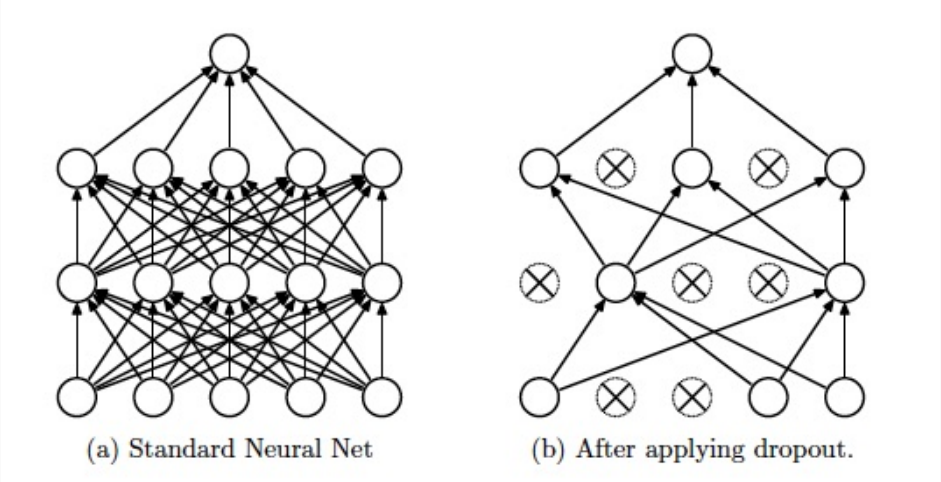
Avoid overfitting by using *dropout 1/3*

For deep networks (> >2 layers):



- During learning, at each gradient computation step: each unit is considered present with a given probability p which means that some units (neurons) are not present during this correction step and thus weights are not systematically corrected.
- During prediction pahse, each unit is present but a factor p is applied to its weights.

Avoid overfitting by using *dropout 2/3*



One interpretation:

If we have *m* neurons , this is equivalent to learn with many sparse networks and at prediction phase a unique one which agregagtes all the others.

THe neurons cannot adapt themselves to the other ones (better generalization properties)

Avoid overfitting by using *dropout 3/3*

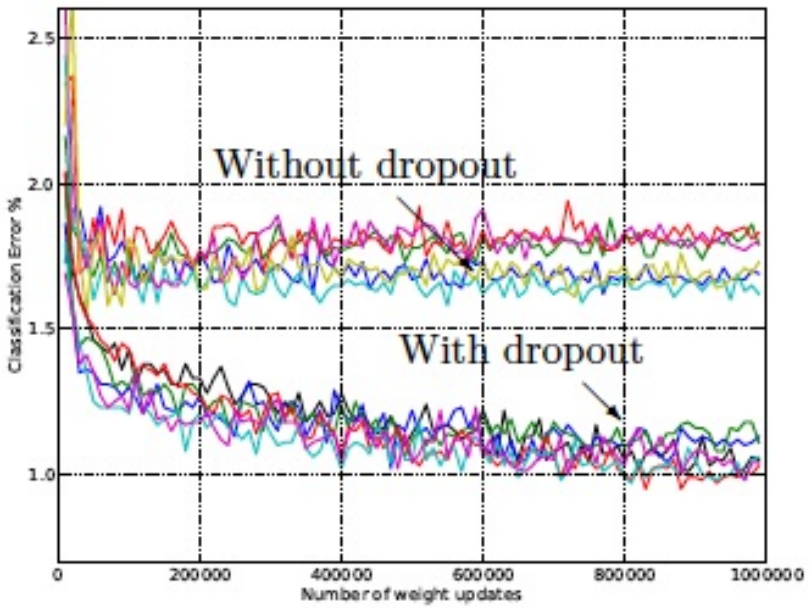


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Apprentissage des réseaux dits profonds

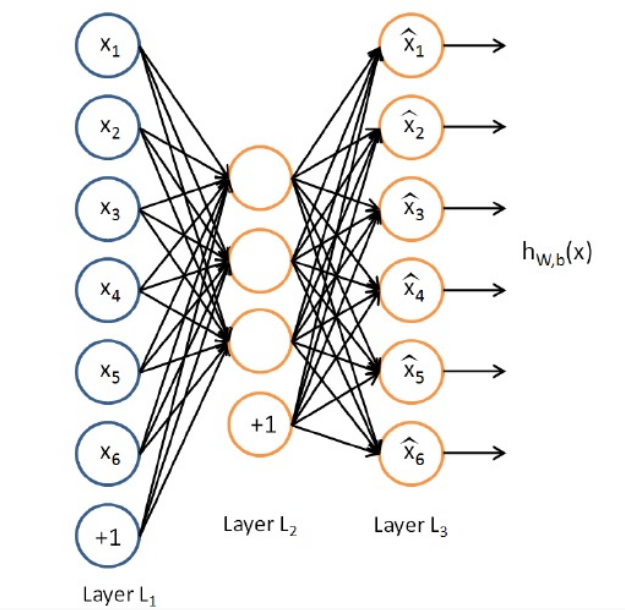
Les réseaux à plusieurs couches sont aujourd'hui appris en étant initialisés par un apprentissage non supervisé souvent à l'aide d'autoencoders ou de Machines de Boltzman restreintes (RBM).
Nous allons voir comment...

autoencoders

An autoencoder is a network devoted to the reconstruction of the data. It aims at reconstructing x by a composition of two or more functions:

$$f_{autoencoder}(x) = f_M \circ f_{M-1} \ldots \circ f_1(x)$$

The simplest architecture is : $f_{autoencoder}(x) = f_2 \circ f_1(x)$



Learning autoencoders

- One typical loss (reconstruction error) $\ell(x, f(x)) = \|x - f(x)\|^2$
- Training by backpropagation
- The auto encoder has two main interests: learning hidden representation (embedding of input data) and denoising
- The learned representations can be used for initialization of a deep network

Autoencoder and feedforward network learning (Erhan et al.)

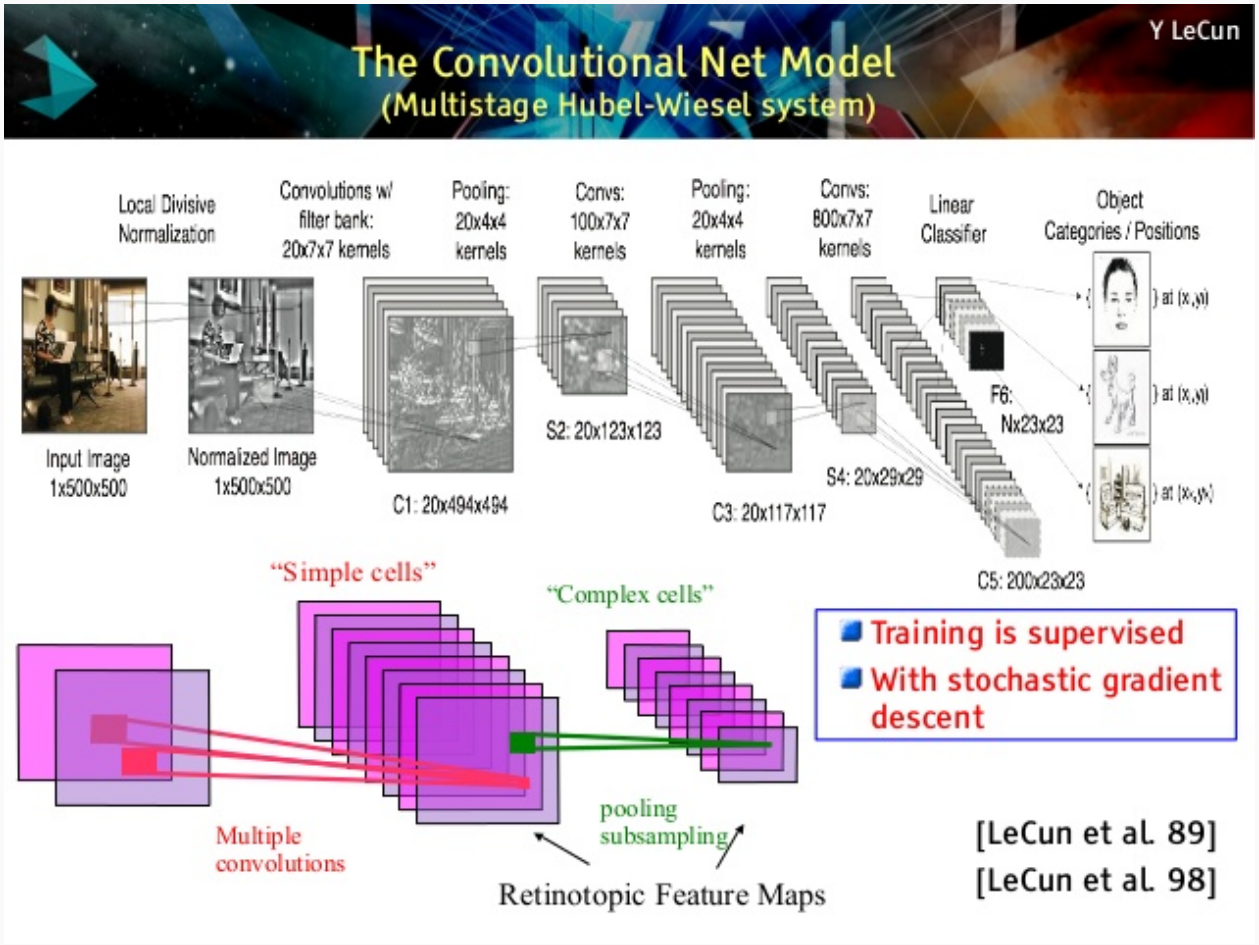
For each hidden layer starting from the closest to the input layer, weights are defined by extracting the first layer of an autoencoder learned on

- Weights of layer 2: learn the autoencoder $x \approx h(x)$. Get the central hidden layer
- Weights of layer 3: learn the autoencoder $f_1(x) \approx h(f_1(x))$. Get the central hidden layer
- etc...
- EThen, supervised learning by backpropagation

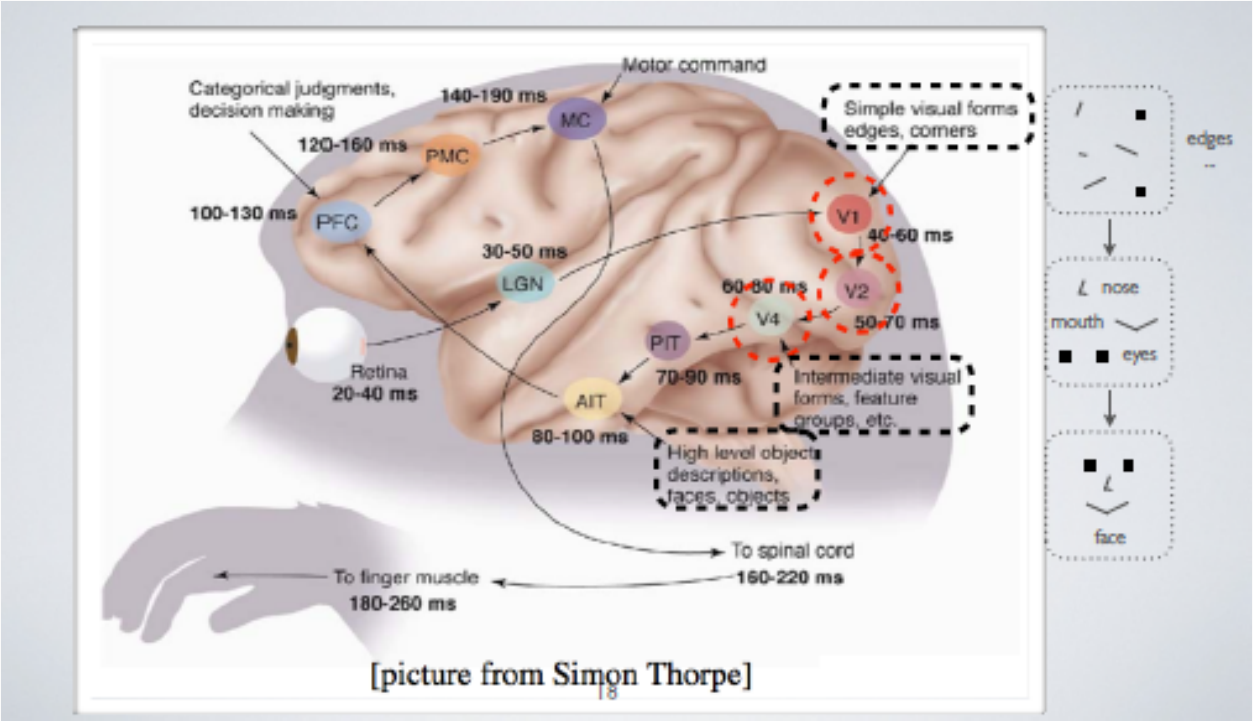
Outline

ConvNet for images

Y. Le Cun.

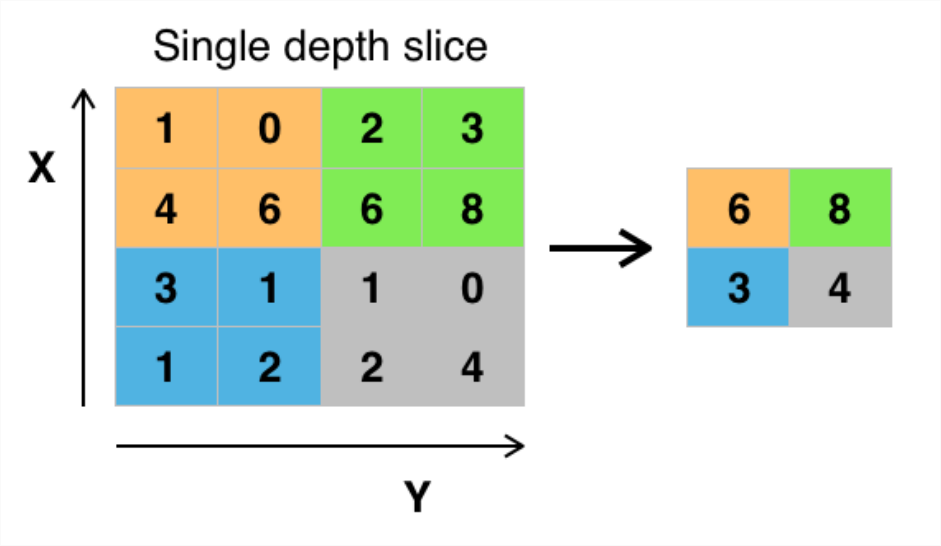


Cortex visuel



Different layers

Receptive fields (convolution layer): radial basis functions (gaussian kernels)
Max pooling:



Références

- Le super cours de Hugo Larochelle (youtube)
- Notes de cours IT6266, Université de Montréal, Equipe de Yoshua Bengio.
- Learning Deep Architectures for AI, Yoshua Bengio, Foundations and Trends in Machine Learning, 2009
- Dropout: A simple way to prevent overfitting, Srivastava et al. JMLR 2014
- Pattern Recognition and Machine Learning, C. Bishop, Springer, 2006.
- `http://deeplearning.net/tutorial/`: pour tout document y compris implémentations...