# Machine/Statistical Learning

## Neural Networks, Perceptron, MLP, Back propagation principle

Florent Chatelain, Olivier.JJ.Michel

GIPSA-Lab, Grenoble-INP, Univ Grenoble Alpes.

2-5. février 2021, ENSTA

# overview

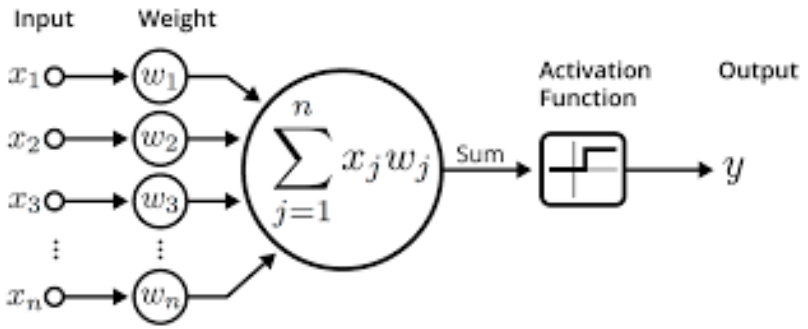
1. Motivation, Rosenblatt's perceptron
2. Mutlilayer perceptron : Gradient backpropagation method
3. Stochastic gradient method
4. Brief insights in CNNs
5. AutoEncoders

# Perceptron : Basic principle

## Motivation

- Build a Bio-inspired parametric model, with possibly high complexity.

## Rosenblatt's perceptron, 1957




An illustration of an artificial neuron. Source: Becoming Human.

- Neural like structure, with a single unit
- $n$ inputs

$w_j \overset{def}{=}$ *weight*, or *connecting weight*
Let $g(.)$ be the activation function, and $x = [x_1, \ldots, x_n]$ the input :

$$\begin{aligned} y &= g(a(x)) \\ a(x) &= \sum_{j=1}^{n} w_j x_x \end{aligned}$$

# Remarks

- -for a (linear) classification problem, $g(a)$ is a threshold (or sigmoidal) function
  -for a (linear) regression problem, $g(a) = a$
- for binary classification,

$$g(a) = \begin{cases} -1 & \text{if } a \leq 0 \\ 1 & \text{if } a > 1 \end{cases}$$

- It is often convenient to introduce a bias to account for possible affine separating hyperplane : $[x_1, \dots x_n] \leftarrow [1, x_1, \dots, x_n] = x$
  $[w_1, \dots, w_n] \leftarrow [w_0, w_1, \dots, w_n] = w$
- In order to predict the probability of $x$ to be in a given class :

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

See 8_NN_Perceptron_MLP/N1_Perceptron.ipynb

# Training the perceptron

As for other ML approaches, minimize the empirical risk, i.e. an averaged cost function.

## Online learning

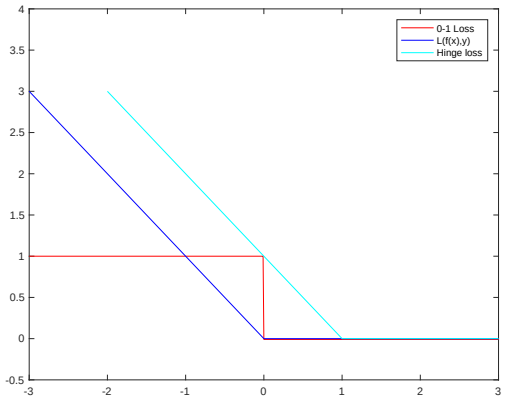
The weights (parameters) $w_0, \ldots, w_n$ are updated to minimize the risk $L(f(x^i), y^i)$ each time that a new pair $(x^i, y^i)$ is received, as opposed to batch learning. Gradient descent algorithm minimization :

$$w_j \leftarrow w_j - \nu \frac{\partial L(f(x^i), y^i)}{\partial w_j}$$

- $\nu$ is the learning rate. In practice, $\nu$ is often decreased when the risk is close to the minimum.
  - if $\nu$ is too large : possible instability
  - if $\nu$ is too small : slow convergence
- Many epochs may be performed on the whole training set.

# historical example : binary classifier

- $y^i \in \{-1, +1\} \forall i \in [1, N]$
- $L(f(x^i, y^i) = \max(0, -y^i a(x^i)) = \max(0, -y^i w^T x^i)$




$$\Rightarrow w_j \leftarrow \begin{cases} w_j & \text{if } y^i w^T x^i > 0 \\ w_j - \nu y^i w^T x^i & \text{if } y^i w^T x^i \leq 0 \end{cases}$$

Albert Novikov theorem, 1962

Let $\mathcal{T} = \{(x^i, y^i), i = 1 \ldots, N\}$ be the training set. Let $D, \gamma \in \mathbb{R}^{+*}$, then

**IF**

- $\forall x^i \in \mathcal{T}, ||x^i||^2 < D$ (← bounded support)
- $\exists u \in \mathbb{R}^{n+1} / \ ||u||^2 = 1 \text{and } \forall (x^i, y^i) \in \mathcal{T}, y^i u^T x^i \geq \gamma$(← margin condition)

**THEN** the perceptron algorithm converges in less than $\left(\frac{D}{\gamma}\right)^2$ iterations.

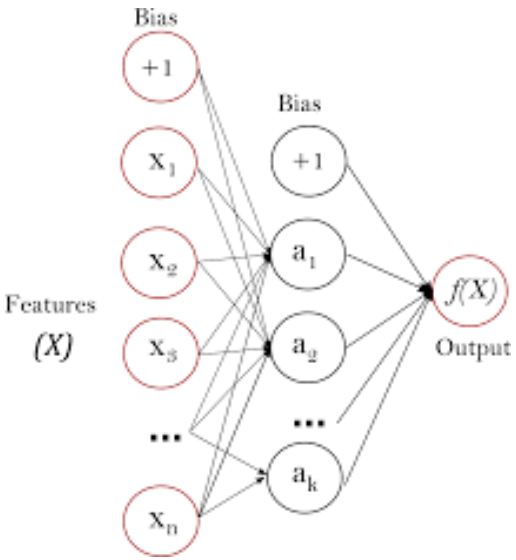
See Perceptron_sonar_example.ipynb(part I on logical example)

# Multilayer perceptrons, Neural Nets

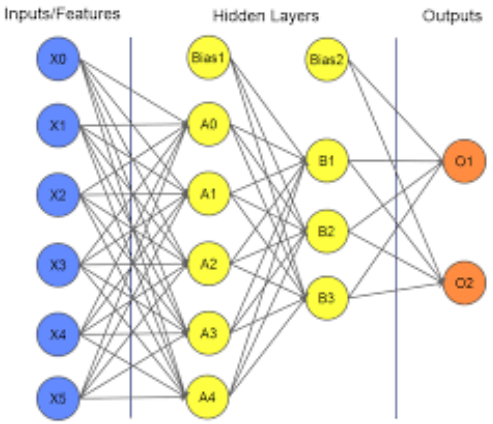
## Motivation

Allow to deal with non linear frontiers between classes by inserting hidden layers, within a FEED FORWARD network.

Single Layer Perceptron




Mulpiple Layer Perceptron



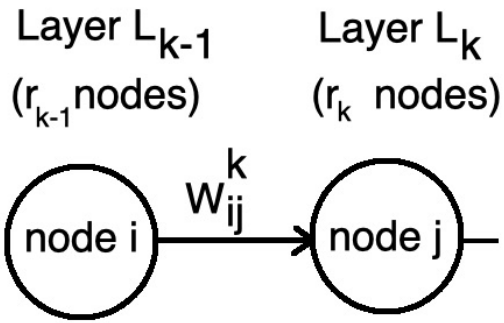

ALL edges are associated to a coefficient that must be estimated.

## Notations

$w_{ij}^k$ : weight for node $j$ in layer $l_k$ for incoming node $i$
$b_i^k$ : bias for node $i$ in layer $l_k$
$a_i^k$ : product sum plus bias (activation) for node $i$ in layer $l_k$
$o_i^k$ : output for node $i$ in layer $l_k$
$r_k$ : number of nodes in layer $l_k$
$g$ : activation function for the hidden layer nodes
$g_o$ : activation function for the output layer nodes

Then

$$a_j^k = b_j^k + \sum_{i=1}^{r_{k-1}} w_{ij}^k o_i^{k-1} = \sum_{i=0}^{r_{k-1}} w_{ij}^k o_i^{k-1}$$




$$o_j^k = g(a_j^k) = g(\sum_{i=0}^{r_{k-1}} w_{ij}^k o_i^{k-1})$$

# Gradient backpropagation algorithm

Backpropagation attempts to minimize the empirical risk (or loss)

$$L(X, \theta) = \frac{1}{N} \sum_i L(f(x^i), y^i)$$

with respect to the neural network's weights (gathered in $\theta$) :
$\rightarrow$ for each weight $w_{ij}^k$, evaluate $\frac{\partial L}{\partial w_{ij}^k}$. By decomposing into a sum over individual error terms for each individual input-output pair

$$\frac{\partial L(X, \theta)}{\partial w_{ij}^k} = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial L(f(x^d), y^d)}{\partial w_{ij}^k} = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial L_d}{\partial w_{ij}^k}.$$

# Loss function derivatives (index $d$ is omitted hereafter

Remind the expression of output at node $N_j$ :

$$o_j^k = g(a_j^k) = g(\sum_{i=0}^{r_{k-1}} w_{ij}^k o_i^{k-1})$$

$L$ depends on $w_{ij}^k$ trough $a_j^k$ : Apply the chain rule to the loss function partial derivative

$$\frac{\partial L}{\partial w_{ij}^k} = \frac{\partial L}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k}$$

$$\begin{cases} \delta_j^k & \stackrel{def}{=} \frac{\partial L}{\partial a_j^k} \\ \frac{\partial a_j^k}{\partial w_{ij}^k} & = \frac{\partial}{\partial w_{ij}^k} \left( \sum_{l=0}^{r_{k-1}} w_{lj}^k o_l^{k-1} \right) = o_i^{k-1} \end{cases} \Rightarrow \frac{\partial L}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}.$$

# Output Layer (MLP with $m+1$ layers)

Assume a one-output neural network, so there is only one output node $j = 1$).
Expressing $L$ in terms of the value $a_1^m$ (since $\delta_1^m$ is a partial derivative with respect to $a_1^m$) gives

$$L(f(x), y) = L\big(g_o(a_1^m), y\big)$$

where $g_o(x)$ is the activation function for the output layer.
thus,

$$\delta_1^m = L'\left(g_0(a_1^m), y\right) g_o'(a_1^m)$$

and finaly

$$\frac{\partial L}{\partial w_{i1}^m} = \delta_1^m o_i^{m-1} = L'\left(g_0(a_1^m), y\right) g_o'(a_1^m)\ o_i^{m-1}.$$

# Backpropagation for hidden layers -cont'd- :

One has for the error term $\delta_j^k$ in layer $1 \leq k < m$ :

$$\delta_j^k = \frac{\partial L}{\partial a_j^k} = \sum_{l=1}^{r^{k+1}} \frac{\partial L}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial a_j^k},$$

where $l$ ranges from 1 to $r^{k+1}$
(The bias input $o_0^k$ corresponds to $w_{0j}^{k+1}$ is fixed, does not depend on the outputs of previous layers, thus $l$ does not take on the value 0.)
Plugging in the error term :

$$\delta_j^k = \sum_{l=1}^{r^{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k}.$$

From the definition of $a_l^{k+1}$

$$a_l^{k+1} = \sum_{j=1}^{r^k} w_{jl}^{k+1} g\left(a_j^k\right),$$

where $g(x)$ is the activation function for the hidden layers,

$$\frac{\partial a_l^{k+1}}{\partial a_j^k} = w_{jl}^{k+1} g'\left(a_j^k\right).$$

# Backpropagation -cont'd- :

Plugging this into the latter equation yields a final equation for the error term $\delta_j^k$ in the hidden layers, called the *backpropagation* formula :

$$\delta_j^k = \sum_{l=1}^{r^{k+1}} \delta_l^{k+1} w_{jl}^{k+1} g'\left(a_j^k\right) = g'\left(a_j^k\right) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}.$$

And putting all equations together :

$$\frac{\partial L}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = g'\left(a_j^k\right) o_i^{k-1} \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}.$$

$\Rightarrow$ gradient values for updating weights at layer $k$ are computed from the gradient $\frac{\partial L_d}{\partial a_l^{k+1}}$ used for updating layer $k+1$

## Computation principle

For each pair $(x_d^i, y_d^i)$, compute the output of each neuron by going forward long the network. Then , during a of back propagation of the errors, update all the weights going from the last hidden layer toward the first one.

# BACKPROPAGATION ALGO, Main equations

For the partial derivatives,

$$\frac{\partial L_d}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}$$

where, for the final layer's error term,

$$\delta_1^m = g_o'(a_1^m) L'(f(x_d), y_d)$$

where, for the hidden layers' error terms,

$$\delta_j^k = g'\left(a_j^k\right) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}.$$

For combining the partial derivatives for each input-output pair,

$$\frac{\partial L(X,\theta)}{\partial w_{ij}^k} = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial L_d}{\partial w_{ij}^k}.$$

For updating the weights,

$$\Delta w_{ij}^k = -\nu \frac{\partial L(X,\theta)}{\partial w_{ij}^k}.$$

# Backpropagation algorithm summary

**1)** Calculate the **forward phase** for each $(x_d, y_d)$; store the results $\hat{y}_d$, $a_j^k$, and $o_j^k$ for each node $j$ in layer $k$ by proceeding from layer 0, to layer $m$, the output layer.

**2)** Calculate the **backward phase** for each $(x_d, y_d)$; store $\frac{\partial L_d}{\partial w_{ij}^k}$ for each weight $w_{ij}^k$.

Proceed from output layer $m$, to layer 1, the input layer.

a) Evaluate $\delta_1^m$

b) Backpropagate the error terms for the hidden layers $\delta_j^k$, working backwards

c) Evaluate the partial derivatives of the individual error $L_d$ with respect to $w_{ij}^k$

**3)** Combine the individual gradients for each input-output pair to get the total gradient for the entire set $X = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ (a simple average of the individual gradients).

**4)** Update the weights according to the learning rate $\alpha$ and total gradient $\frac{\partial L(X,\theta)}{\partial w_{ij}^k}$

# Remarks

In the classic formulation,
for hidden nodes $\big(g(x) = \sigma(x)\big)$ (sigmoidal function)
and the output activation function is $\big(g_o(x) = x\big)$

$$g'(x) = \frac{\partial \sigma(x)}{\partial x} = \sigma(x)\big(1 - \sigma(x)\big).$$

$$g'_o(x) = \frac{\partial g_o(x)}{\partial x} = \frac{\partial x}{\partial x} = 1.$$

$\rightarrow$ No need to remember the activation values $a_1^m$ and $a_j^k$ in addition to the output values $o_1^m$ and $o_j^k$, greatly reducing the memory footprint of the algorithm.

BUT gradient descent algorithm may be infeasible when the training data size is huge.
Thus, a stochastic version of the algorithm is often used instead.

# Remarks -cont'd-

- Empirical risk minimization for multilayer perceptron is an ill-posed and ill-conditioned NON CONVEX problem.
- Gradient values in the first hidden layers often takes either too large (explosion) or too low values (vanishing gradient, leading to slow down learning convergence).
- Weights initialization values, learning rate, choice of $g()$, $m$, $r_m$ do all influence the result ! first results date back to Hinton, 2006
- To avoid saturation effects of node outputs (either to 0 or to 1), $l_2$ regularization of $L(f(x), y)$ may beapplied on $\theta$.

# Stochastic Gradient method

At each iteration, rather than computing

$$\nabla_\theta L(\mathrm{X}) = \nabla_\theta(\sum_{d=1}^{N} L(x_d)) = \sum_{d=1}^{N} \nabla_\theta L(x_d)$$

stochastic gradient descent randomly samples $d$ at uniform and computes $\nabla_\theta L(x_d)$ instead :
SGD uses $\nabla L(x_d)$ as an unbiased estimator of $\nabla L(\mathrm{X})$ ;

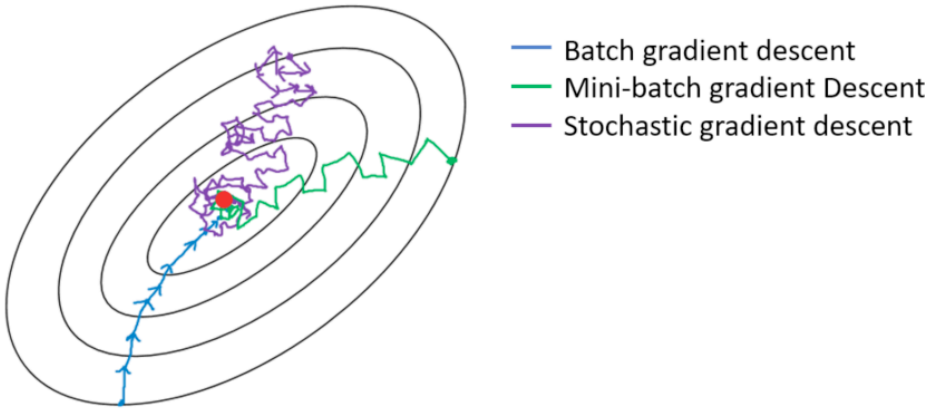
$$\mathbb{E}\left[\nabla_\theta L(x_d)\right] = \mathbb{E}\left[\frac{1}{k}\sum_{i=1}^{k} \nabla L(x_k)\right] = \nabla L(\mathrm{X}).$$

In a generalized case, at each iteration a mini-batch $\mathcal{B}$ that consists of indices for training data instances may be sampled at uniform with replacement.

$$\nabla L_{\mathcal{B}}(\mathrm{X}) = \frac{1}{|\mathcal{B}|}\sum_{d\in\mathcal{B}} \nabla L(x_d)$$

update $\theta$ as

$$\theta := \theta - \eta \nabla L_{\mathcal{B}}(\mathrm{X})$$




- The per-iteration computational cost is $\mathcal{O}(|\mathcal{B}|)$. Thus, when the mini-batch size is small, the computational cost at each iteration is light.
- If the training data set has many redundant data instances, stochastic gradients may be so close to the true gradient $\nabla_\theta L(\mathrm{X})$ that a small number of iterations will find useful solutions to the optimization problem.
- Stochastic gradient descent can be considered as offering a regularization effect especially when the mini-batch size is small due to the randomness and noise in the mini-batch sampling.
- Certain hardware processes mini-batches of specific sizes more efficiently.

See 8_NN_Perceptron_MLP/N1_Perceptron.ipynb (part II)
See 8_NN_Perceptron_MLP/N2_MLPClassifier.ipynb

## Backprop in Practice

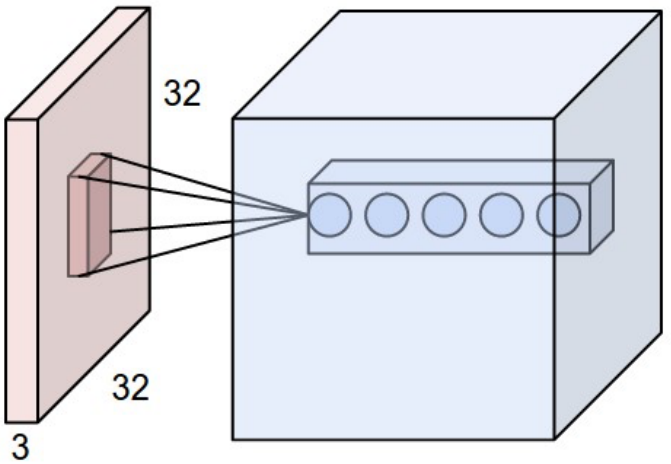
Y LeCun

- Use ReLU non-linearities
- Use cross-entropy loss for classification
- Use Stochastic Gradient Descent on minibatches
- Shuffle the training samples (← very important)
- Normalize the input variables (zero mean, unit variance)
- Schedule to decrease the learning rate
- Use a bit of L1 or L2 regularization on the weights (or a combination)
  - But it's best to turn it on after a couple of epochs
- Use "dropout" for regularization
- Lots more in [LeCun et al. "Efficient Backprop" 1998]
- Lots, lots more in "Neural Networks, Tricks of the Trade" (2012 edition) edited by G. Montavon, G. B. Orr, and K-R Müller (Springer)
- More recent: Deep Learning (MIT Press book in preparation)

# Brief insights into Convolutional Neural networks (CNN)

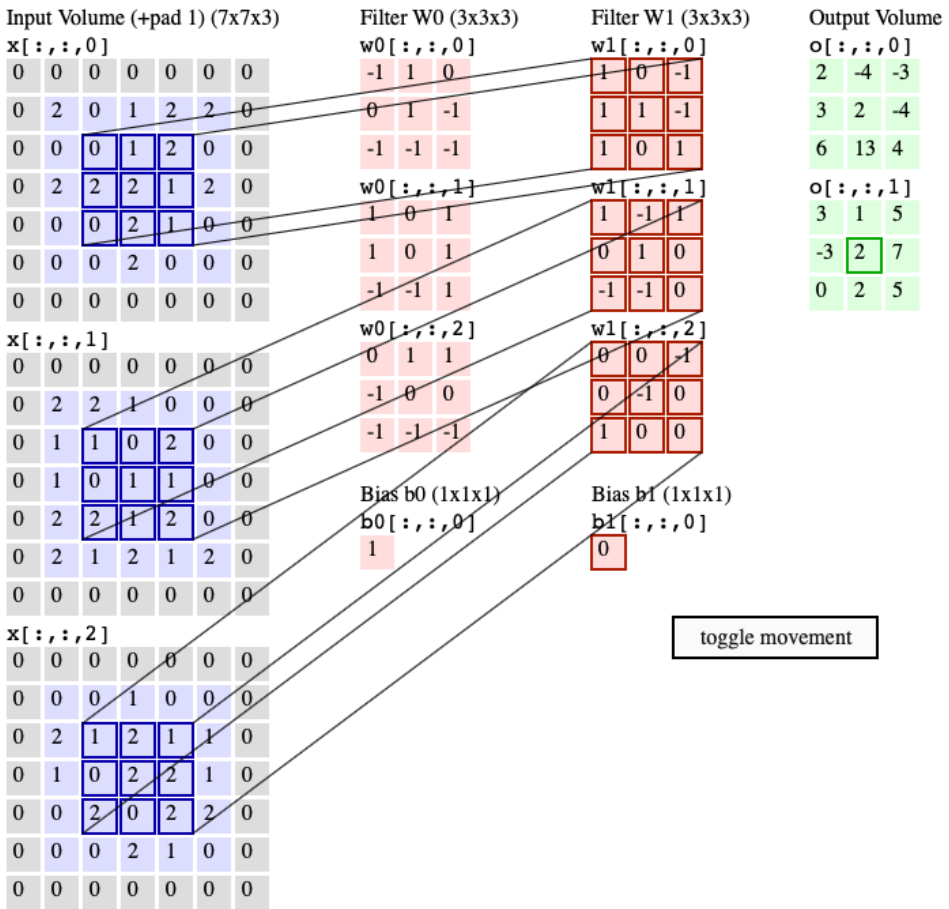
Motivations :

- MLP can require a lot of parameters. For example a 256x256 image over 3 channels and 1000 nodes in first MLP layer would require to identify more than 200 million parameters !
- The MLP hidden layers totally ignore potentially existing spatial structure !
- Convolutional layers associate each of their nodes with a weighted window (named "receptive field" or "filter kernel")



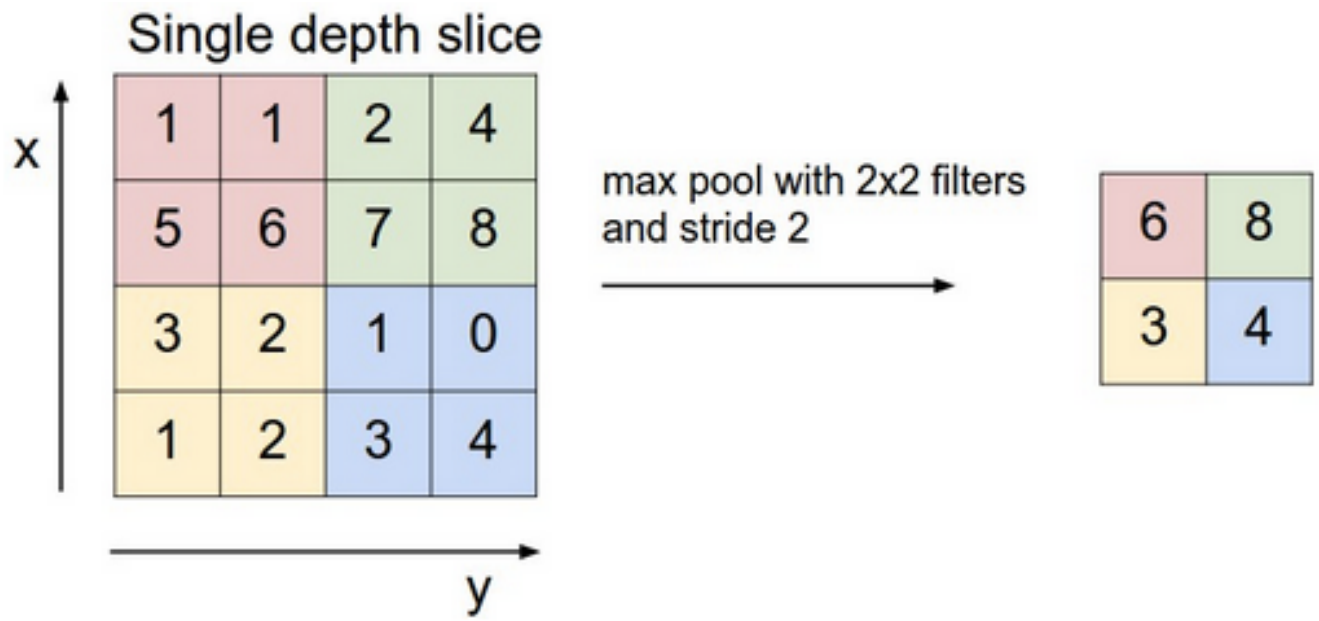
# CNN Convolutive Layer example

RGB Channels  Filter 1  Filter 2  Output Stack



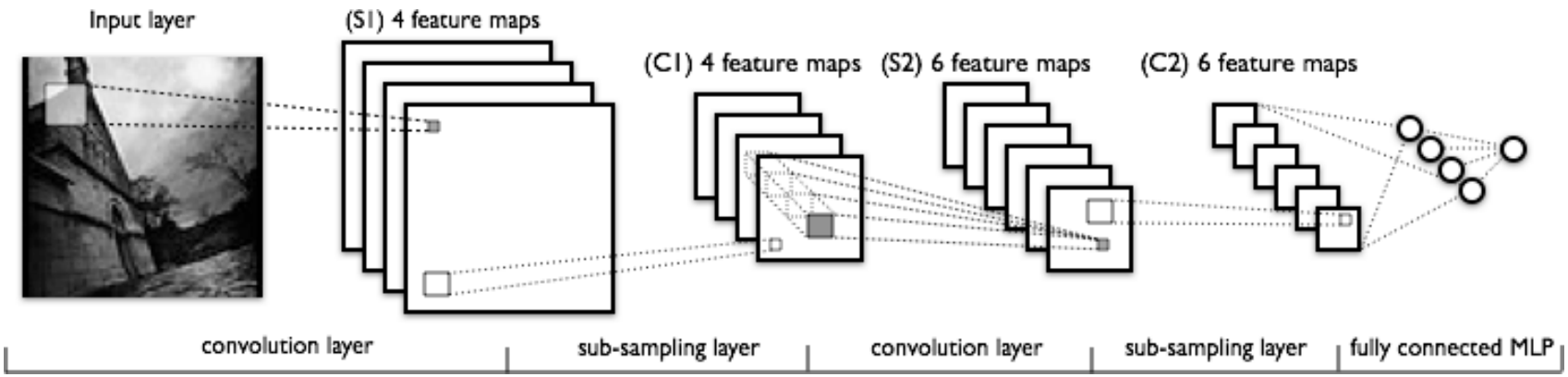
# Pooling / Downsampling within CNNs

Example : Max Pooling :




- Only the locations on the image that shows the strongest correlation to each feature (the max value) are preserved, and those max values combine to form a lower-dimensional space
- Decreases the amount if storage and processing requirements but at the expend of the loss of information about lesser values

# Example of alternating sequences of transformations involved in CNNs

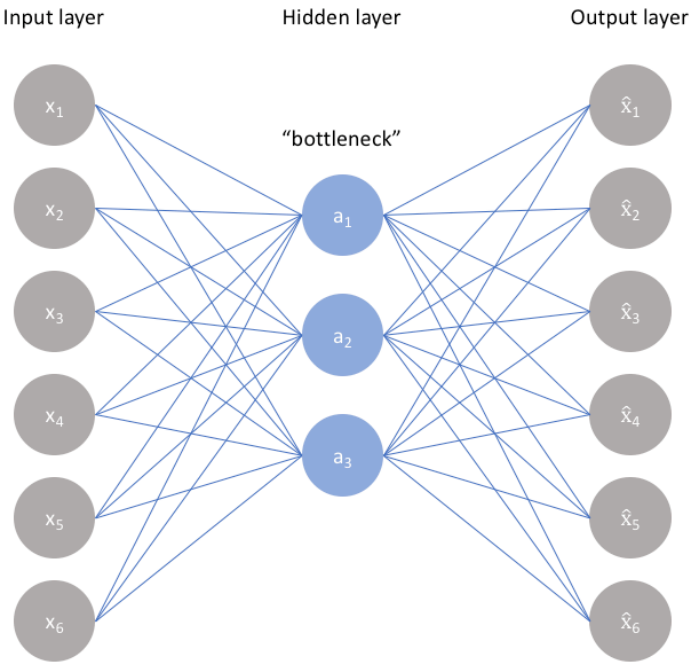


# AutoEncoder

Images for this section were adapted from https ://www.jeremyjordan.me/autoencoders/

- Autoencoders are Unsupervised Neural Networks, designed for Representation learning and/or dimension reduction.
- main idea : impose a bottleneck in the network to compressed knowledge representation of the input.

*rk* : This assumes that the data are structured (input features are correlated) as for e.g. iid data, such compression will be very difficult if not impossible without loosing much information.

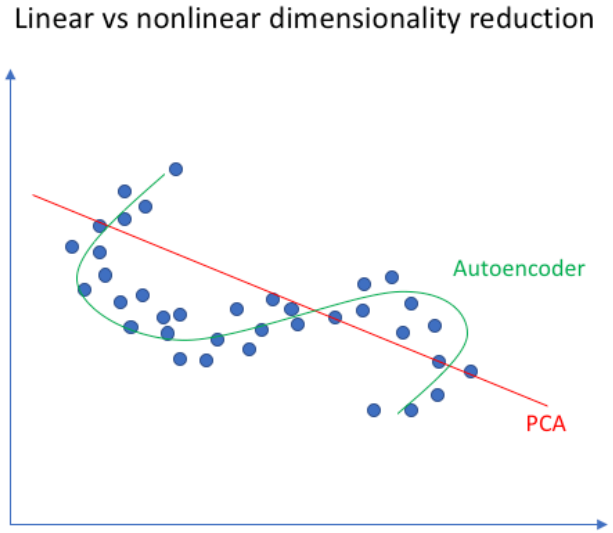


# AutoEncoder principle

$\Rightarrow$ Formulate the problem as a supervised learning problem whose output is $\{x^i\}$

$\Rightarrow$ The empirical risk to minimize is thus $L(f(x), x)$ : the bottleneck plays a key role (otherwise the network simply passes the values to the output.

$\Rightarrow$ if linear activation function were used, that would perform PCA like dimension reduction




## The AutoEncoder must be :

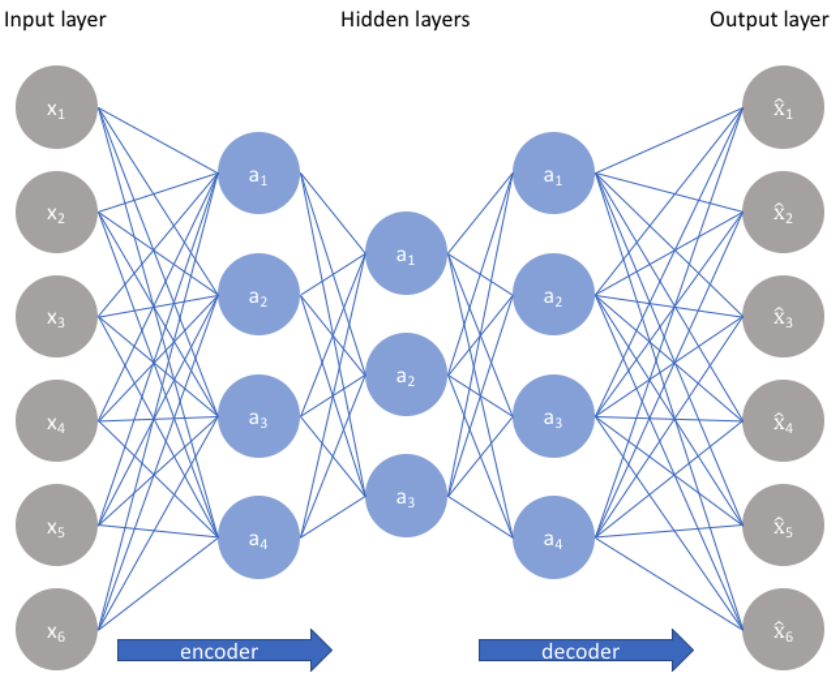
- sensitive enough to the inputs, to built an accurate reconstruction
- insensitive enough to the inputs to avoid overfitting

This requires to regularize the loss function of the form

$$L(f(x), x) + \textit{regularization}$$

# UnderComplete AE

Limit of the flow of information going through the NN by limiting the nb of nodes in the hidden layers :




No explicit regularization term is required here.
If too many nodes, i.e. high capacity -in the sense of Vapnik-, the AE may be capable of learning a way to simply memorize the data. The primary aim to discover latent variable cannot be attained !

# Sparse AE

**Idea** is to keep the number of nodes in hidden layers quite large, but regularize the loss function by penalizing activations within a layer ($\neq$ weights regularisation) $\Rightarrow$ only a small nb of neurons are activated.

$l_1$ regularization

For layer $k$ :

$$(f(x), x) + \lambda \sum_{j=1}^{r_k} |o_j^k|$$

Remind that in the notation of the previous section, activation of a neuron was noted $o_j^k = g(a_j^k)$. Activations are in $[0, 1]$ or $[-1, 1]$, depending on the choice of $g()$. Below, we assume $o_j^k \in [0, 1]$.

# Sparse AE - cont'd-

## Kullback-Leibler regularization

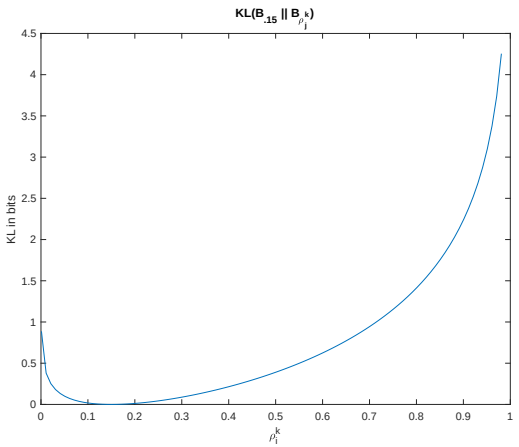
Let $\hat{\rho}_j^k \overset{def}{=} \frac{1}{n}\sum_{i=1}^{n} o_j^k(x)$ be the average activation of neuron $i$ in layer $k$, estimated over a collection of $n$ samples $\{x^i, i = 1 \dots n\}$.

$$L(f(x), x) + \lambda \sum_{k=1}^{m} \sum_{j=1}^{r_k} \mathrm{KL}(\mathcal{B}_\rho || \mathcal{B}_{\hat{\rho}_j^k})$$

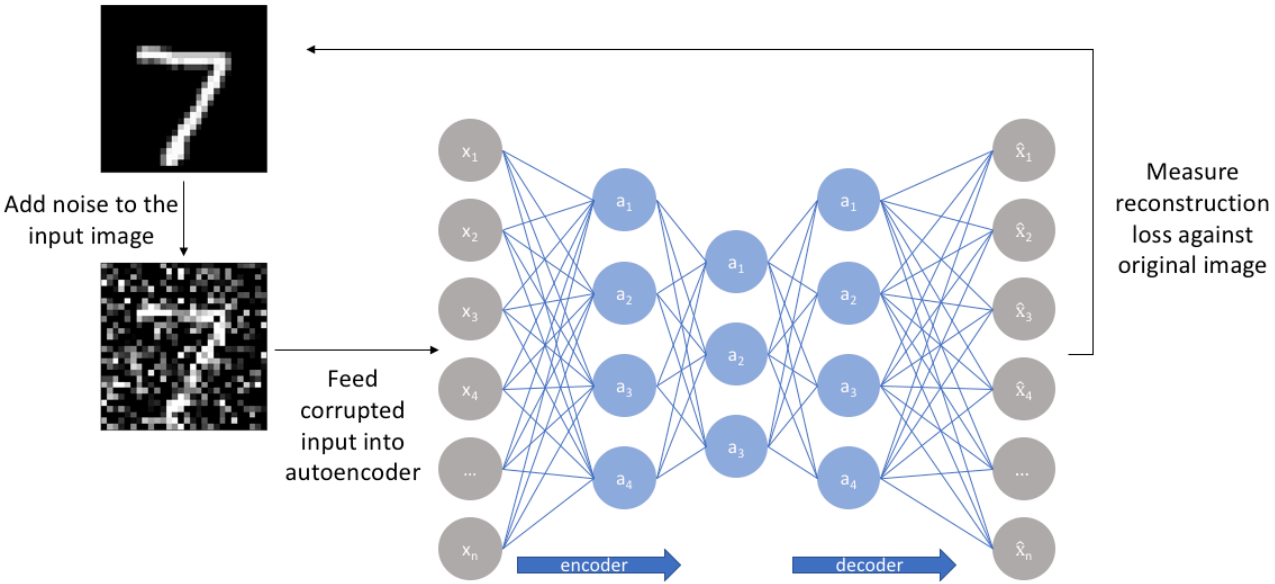
where $\mathcal{B}_\rho$ is the Bernoulli process of parameter $\rho$, thus

$$\mathrm{KL}(\mathcal{B}_\rho || \mathcal{B}_{\hat{\rho}_j^k}) = \rho \log \frac{\rho}{\hat{\rho}_{jk}} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_{jk}}$$

$\Rightarrow$ $\rho$ acts as a "sparsity parameter" ; small values of $\rho$ correspond to low probability for the neuron to fire.



# AE for denoising : principle




## Alternate approach for denoising

Force the activation of the hidden layer to be weakly sensitive to small deviations of the inputs

$$L(f(x), x) + \lambda \sum_{k=1}^{m} \sum_{j=1}^{r_k} ||\nabla_x o_j^k(x)||^2$$