



Institut de Mathématiques de Toulouse, INSA Toulouse

# Neural Networks and Introduction to Deep Learning

Formation en Machine Learning  
EUR NanoX

Béatrice Laurent - Brendan Guilloet - Olivier Roustant

# Outline

---

Introduction

Neural Networks, Multilayer perceptrons

Estimation of the parameters

Convolutional Neural Networks (CNN)

Conclusion

# Introduction

- Deep learning is a set of learning methods attempting to model data with complex architectures combining different non-linear transformations.
- The elementary bricks of deep learning are the neural networks, that are combined to form the deep neural networks.

There exist several types of architectures for neural networks :

- The multilayer perceptrons, that are the oldest and simplest ones
- The Convolutional Neural Networks (CNN), particularly adapted for image processing
- The recurrent neural networks, used for sequential data such as text or times series.

# Introduction

- Deep learning architectures are based on **deep cascade of layers**.
- They need clever **stochastic optimization algorithms**, and **initialization**, and also a clever choice of the **structure**.
- They lead to very **impressive results**, although very **few theoretical foundations** are available till now.
- These techniques have enabled **significant progress** in the fields of **sound and image processing**, including facial recognition, speech recognition, computer vision, automated language processing, text classification (for example spam recognition).

# Outline

Introduction

Neural Networks, Multilayer perceptrons

Estimation of the parameters

Convolutional Neural Networks (CNN)

Conclusion

# Neural networks

- An **artificial neural network** is non linear with respect to its parameters  $\theta$  that associates to an entry  $x$  an output  $y = f(x, \theta)$ .
- The neural networks can be used for **regression or classification**.
- The parameters  $\theta$  are estimated from a learning sample.
- The function to minimize is not convex, leading to local minimizers.
- The success of the method came from a **universal approximation theorem** due to Cybenko (1989) and Hornik (1991).
- Le Cun (1986) proposed an efficient way to compute the gradient of a neural network, called **backpropagation of the gradient**, that allows to obtain a local minimizer of the quadratic criterion easily.

# Artificial Neuron

## Artificial neuron

- a function  $f_j$  of the input  $x = (x_1, \dots, x_d)$
- weighted by a vector of connection weights  $w_j = (w_{j,1}, \dots, w_{j,d})$ ,
- completed by a neuron bias  $b_j$ ,
- and associated to an activation function  $\phi$  :

$$y_j = f_j(x) = \phi(\langle w_j, x \rangle + b_j).$$

# Activation functions

Several activation functions can be considered.

## Activation functions

- The identity function  $\phi(x) = x$
- The sigmoid function (or logistic)  $\phi(x) = \frac{1}{1+e^{-x}}$
- The hyperbolic tangent function ("tanh")

$$\phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- The hard threshold function  $\phi_\beta(x) = \mathbb{1}_{x \geq \beta}$
- The Rectified Linear Unit (ReLU) activation function  
 $\phi(x) = \max(0, x)$

# Activation functions

The following Figure represents the activation function described above.

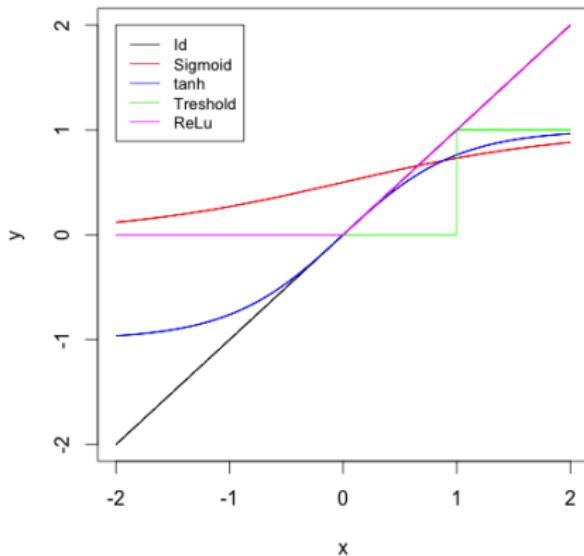
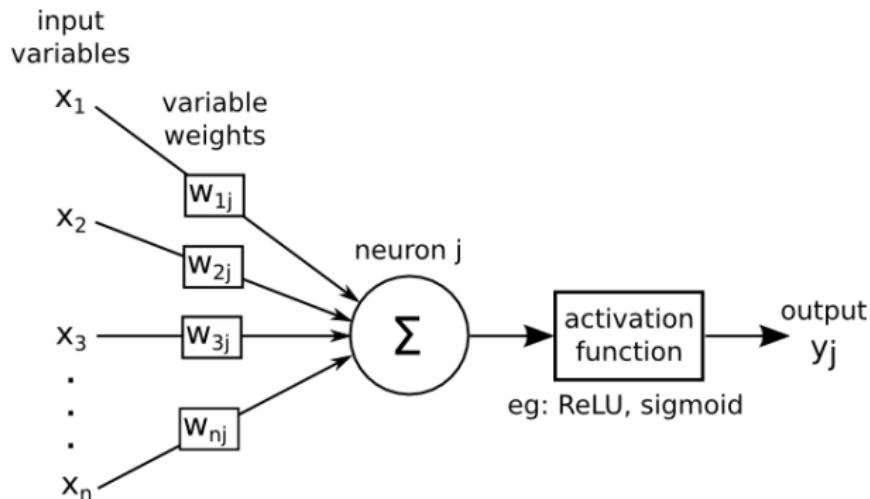


FIGURE – Activation functions

# Artificial Neuron

Schematic representation of an **artificial neuron** where  $\Sigma = \langle w_j, x \rangle + b_j$ .



- Historically, the sigmoid was the mostly used activation function since it is differentiable and allows to keep values in the interval  $[0, 1]$ .
- Nevertheless, it is problematic since its gradient is very close to 0 when  $|x|$  is not close to 0.

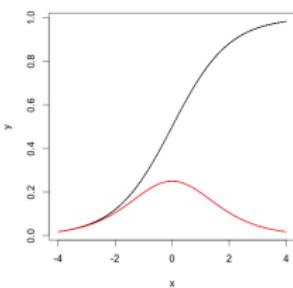


FIGURE – Sigmoid function (in black) and its derivatives (in red)

# Activation functions

- With neural networks with a high number of layers, this causes troubles for the **backpropagation algorithm** to estimate the parameters.
- This is why the sigmoid function was supplanted by the **rectified linear function (ReLU)**. This function is piecewise linear and has many of the properties that make linear model easy to optimize with gradient-based methods.

# Multilayer perceptron

- A multilayer perceptron (or neural network) is a structure composed by several hidden layers of neurons where the output of a neuron of a layer becomes the input of a neuron of the next layer.
- On last layer, called output layer, we may apply a different activation function as for the hidden layers depending on the type of problems we have at hand : regression or classification.

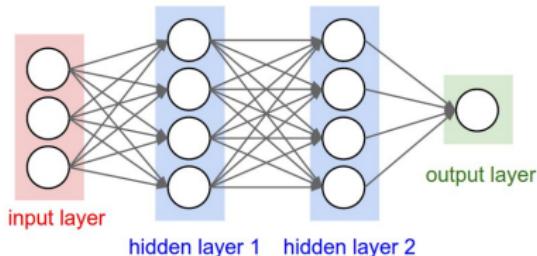


FIGURE – A basic neural network.

# Multilayer perceptron

- The output of a neuron can also be the input of a neuron of the same layer or of neuron of previous layers : this is the case for recurrent neural networks.

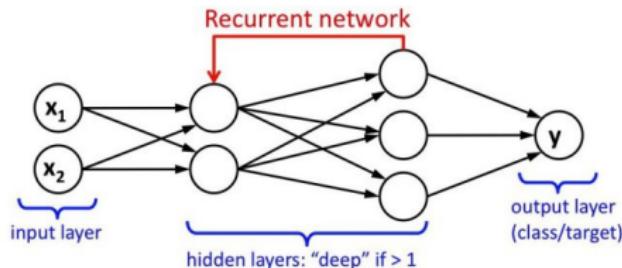


FIGURE – A recurrent neural network

# Multilayers perceptrons

- The parameters of the architecture are the **number of hidden layers and of neurons in each layer**.
- The **activation functions** are also to choose by the user. On the **output layer**, we apply **no activation function** (the identity function) in the case of regression.
- For **binary classification**, the output gives a prediction of  $\mathbb{P}(Y = 1/X)$  since this value is in  $[0, 1]$ , **the sigmoid activation function** is generally considered.
- For **multi-class classification**, the output layer contains one neuron per class  $i$ , giving a prediction of  $\mathbb{P}(Y = i/X)$ . The sum of all these values has to be equal to 1.
- The multidimensional function **softmax** is generally used

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}.$$

# Mathematical formulation

Multilayer perceptron with  $L$  hidden layers :

- We set  $h^{(0)}(x) = x$ .

**For**  $k = 1, \dots, L$  (hidden layers),

$$\begin{aligned} a^{(k)}(x) &= b^{(k)} + W^{(k)} h^{(k-1)}(x) \\ h^{(k)}(x) &= \phi(a^{(k)}(x)) \end{aligned}$$

**For**  $k = L + 1$  (output layer),

$$\begin{aligned} a^{(L+1)}(x) &= b^{(L+1)} + W^{(L+1)} h^{(L)}(x) \\ h^{(L+1)}(x) &= \psi(a^{(L+1)}(x)) := f(x, \theta). \end{aligned}$$

where  $\phi$  is the activation function and  $\psi$  is the output layer activation function

- At each step,  $W^{(k)}$  is a matrix with number of rows the number of neurons in the layer  $k$  and number of columns the number of neurons in the layer  $k - 1$ .

# Universal approximation theorem

## Theorem

Let  $\phi$  be a bounded, continuous and non decreasing (activation) function. Let  $K_d$  be some compact set in  $\mathbb{R}^d$  and  $\mathcal{C}(K_d)$  the set of continuous functions on  $K_d$ . Let  $f \in \mathcal{C}(K_d)$ . Then for all  $\varepsilon > 0$ , there exists  $N \in \mathbb{N}$ , real numbers  $v_i, b_i$  and  $\mathbb{R}^d$ -vectors  $w_i$  such that, if we define

$$F(x) = \sum_{i=1}^N v_i \phi(\langle w_i, x \rangle + b_i)$$

then we have

$$\forall x \in K_d, |F(x) - f(x)| \leq \varepsilon.$$

# Universal approximation theorem

- This theorem due to Hornik (1991) shows that **any bounded and regular function  $\mathbb{R}^d \rightarrow \mathbb{R}$  can be approximated at any given precision by a neural network with one hidden layer** containing a finite number of neurons, having the same activation function, and one linear output neuron.
- This theorem is interesting from a theoretical point of view. From a practical point of view, this is not really useful since **the number of neurons in the hidden layer may be very large**. The strength of deep learning lies in the deep (number of hidden layers) of the networks.

# Outline

Introduction

Neural Networks, Multilayer perceptrons

**Estimation of the parameters**

Convolutional Neural Networks (CNN)

Conclusion

# Estimation of the parameters

- Once the architecture of the network has been chosen, the parameters (the weights  $w_j$  and biases  $b_j$ ) have to be estimated from a learning sample  $(X_i, Y_i)_{1 \leq i \leq n}$ .
- As usual, the estimation is obtained by minimizing a loss function, generally with a gradient descent algorithm.
- We first have to choose the loss function between the output of the model  $f(x, \theta)$  for an entry  $x$  and the target  $y$ .

# Loss functions

- If the **regression model**, we generally consider the loss function associated to the  $\mathbb{L}_2$  norm :

$$\ell(f(x, \theta), y) = \|y - f(x, \theta)\|^2.$$

- For **binary classification**, with  $Y \in \{0, 1\}$ , maximizing the log likelihood corresponds to the **minimization of the cross-entropy**. Setting  $f(x, \theta) = P_\theta(Y = 1 | X = x)$ ,

$$\begin{aligned}\ell(f(x, \theta), y) &= -[y \log(f(x, \theta)) + (1 - y) \log(1 - f(x, \theta))] \\ &= -\log(P_\theta(Y = y | X = x))\end{aligned}$$

# Loss functions

- For a **multi-class classification** problem, we consider a generalization of the previous loss function to  $k$  classes

$$\ell(f(x, \theta), y) = - \left[ \sum_{j=1}^k \mathbb{1}_{y=j} \log P_\theta(Y = j | X = x) \right].$$

- Ideally we would like to minimize the **classification error**, but it is not smooth, this is why we consider the **cross-entropy**.

# Penalized empirical risk

- Denoting  $\theta$  the vector of parameters to estimate, we consider the expected loss function

$$L(\theta) = \mathbb{E}_{(X, Y) \sim P} [\ell(f(X, \theta), Y)],$$

associated to the loss function  $\ell$ .

- In order to estimate the parameters  $\theta$ , we use a training sample  $(X_i, Y_i)_{1 \leq i \leq n}$  and we minimize the empirical loss

$$\tilde{L}_n(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(f(X_i, \theta), Y_i)$$

eventually we add a regularization term.

- This leads to minimize the penalized empirical risk

$$L_n(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(f(X_i, \theta), Y_i) + \lambda \Omega(\theta).$$

# Penalized empirical risk

- We can consider  $\mathbb{L}^2$  regularization.

$$\begin{aligned}\Omega(\theta) &= \sum_k \sum_i \sum_j (W_{i,j}^{(k)})^2 \\ &= \sum_k \|W^{(k)}\|_F^2\end{aligned}$$

where  $\|W\|_F$  denotes the Frobenius norm of the matrix  $W$ .

- Note that only the weights are penalized, the biases are not penalized.
- It is easy to compute the gradient of  $\Omega(\theta)$  :

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

- One can also consider  $\mathbb{L}^1$  regularization (LASSO penalty), leading to parsimonious solutions :

$$\Omega(\theta) = \sum_k \sum_i \sum_j |W_{i,j}^{(k)}|.$$

# Penalized empirical risk

- In order to minimize the criterion  $L_n(\theta)$ , a **stochastic gradient descent algorithm** is often used.
- In order to compute the gradient, a clever method, called **Backpropagation algorithm** is considered.

# The stochastic gradient descent algorithm

- Initialization of  $\theta = (W^{(1)}, b^{(1)}, \dots, W^{(L+1)}, b^{(L+1)})$ .
- For  $j = 1, \dots, N$  iterations :
  - At step  $j$  :

$$\theta = \theta - \varepsilon \frac{1}{m} \sum_{i \in B} [\nabla_\theta \ell(f(X_i, \theta), Y_i) + \lambda \nabla_\theta \Omega(\theta)],$$

where  $B$  is a subset of  $\{1, \dots, n\}$  with cardinality  $m$ .

# The stochastic gradient descent algorithm

- In the previous algorithm, we do not compute the gradient for the empirical loss function at each step of the algorithm but only on a subset  $B$  of cardinality  $m$  (called a **batch**).
- This is what is classically done for **big data sets** (and for deep learning) or for sequential data.
- $B$  is taken **at random without replacement**.
- An iteration over all the training examples is called an **epoch**.
- The **numbers of epochs** to consider is a parameter of the deep learning algorithms.
- The total number of iterations equals the number of epochs times the sample size  $n$  divided by  $m$ , the size of a batch.
- This procedure is called **batch learning**, sometimes, one also takes batches of size 1, reduced to a single training example  $(X_i, Y_i)$ .

To apply the SGD algorithm, we need to compute the gradients  $\nabla_{\theta} \ell(f(X_i, \theta), Y_i)$ . For this, we use the **Backpropagation algorithms**.

# Backpropagation algorithm for regression

- We consider the **regression case** and explain how to compute the gradient of the empirical **quadratic loss** by the **Backpropagation algorithm**.
- To simplify, we do not consider here the penalization term, that can easily be added.
- Assuming that the output of the multilayer perceptron is of size  $K$ , and using the previous notations, the **empirical quadratic loss** is proportional to

$$\sum_{i=1}^n R_i(\theta) = \sum_{i=1}^n \|Y_i - f(X_i, \theta)\|^2$$

with

$$R_i(\theta) = \sum_{k=1}^K (Y_{i,k} - f_k(X_i, \theta))^2.$$

# Mathematical formulation

Multilayer perceptron with  $L$  hidden layers :

- We set  $h^{(0)}(x) = x$ .

**For**  $k = 1, \dots, L$  (hidden layers),

$$\begin{aligned} a^{(k)}(x) &= b^{(k)} + W^{(k)} h^{(k-1)}(x) \\ h^{(k)}(x) &= \phi(a^{(k)}(x)) \end{aligned}$$

**For**  $k = L + 1$  (output layer),

$$\begin{aligned} a^{(L+1)}(x) &= b^{(L+1)} + W^{(L+1)} h^{(L)}(x) \\ h^{(L+1)}(x) &= \psi(a^{(L+1)}(x)) := f(x, \theta). \end{aligned}$$

where  $\phi$  is the activation function and  $\psi$  is the output layer activation function

- At each step,  $W^{(k)}$  is a matrix with number of rows the number of neurons in the layer  $k$  and number of columns the number of neurons in the layer  $k - 1$ .

# Backpropagation algorithm for regression

- Let us introduce the notations

$$\begin{aligned}\delta_{k,i} &= -2(Y_{i,k} - f_k(X_i, \theta))\psi'(a_k^{(L+1)}(X_i)) \\ s_{m,i} &= \phi' \left( a_m^{(L)}(X_i) \right) \sum_{k=1}^K W_{k,m}^{(L+1)} \delta_{k,i}.\end{aligned}$$

- Then we have

$$\frac{\partial R_i}{\partial W_{k,m}^{(L+1)}} = \delta_{k,i} h_m^{(L)}(X_i) \quad (1)$$

$$\frac{\partial R_i}{\partial W_{m,l}^{(L)}} = s_{m,i} h_l^{(L-1)}(X_i), \quad (2)$$

known as the **backpropagation equations**.

# Backpropagation algorithm for regression

- We use the **Backpropagation equations** to compute the gradient by a two pass algorithm.
- In the *forward pass*, we fix the value of the current weights  $\theta^{(r)} = (W^{(1,r)}, b^{(1,r)}, \dots, W^{(L+1,r)}, b^{(L+1,r)})$ , and we compute the predicted values  $f(X_i, \theta^{(r)})$  and all the intermediate values  $(a^{(k)}(X_i), h^{(k)}(X_i) = \phi(a^{(k)}(X_i)))_{1 \leq k \leq L+1}$  that are stored.
- Using these values, we compute during the *backward pass* the quantities  $\delta_{k,i}$  and  $s_{m,i}$  and the partial derivatives given in Equations 1 and 2.
- We have computed the partial derivatives of  $R_i$  only with respect to the weights of the output layer and the previous ones, but we can go on to compute the partial derivatives of  $R_i$  with respect to the weights of the previous hidden layers.
- In the back propagation algorithm, each hidden layer gives and receives informations from the neurons it is connected with.
- Hence, the algorithm is adapted for parallel computations.

# Backpropagation algorithm

- The values of the gradient are used to update the parameters in the gradient descent algorithm.
  - At step  $r + 1$ , we have :

$$\tilde{\nabla}_{\theta} = \frac{1}{m} \sum_{i \in B} \nabla_{\theta} \ell(f(X_i, \theta), Y_i),$$

where  $B$  is a batch with cardinality  $m$ .

- Update the parameters

$$\theta^{(r+1)} = \theta^{(r)} - \varepsilon_r \tilde{\nabla}_{\theta},$$

where  $\varepsilon_r > 0$  is the learning rate. that satisfies  $\varepsilon_r \rightarrow 0$ ,  $\sum_r \varepsilon_r = \infty$ ,  $\sum_r \varepsilon_r^2 < \infty$ , for example  $\varepsilon_r = 1/r$ .

# Regularization

- We have already mentioned  $L^2$  or  $L^1$  penalization.
- For deep learning, the mostly used method is the **dropout** introduced by Hinton et al. (2012).
- With a certain **probability  $p$** , and independently of the others, each unit of the network is **set to 0**.
- It is classical to set  $p$  to **0.5** for units in the **hidden layers**, and to **0.2** for the **entry layer**.
- The **computational cost is weak** since we just have to set to 0 some weights with probability  $p$ .

# Dropout

- This method **improves significantly** the generalization properties of deep neural networks and is now the **most popular regularization method** in this context.
- The disadvantage is that **training is much slower** (it needs to increase the number of epochs).

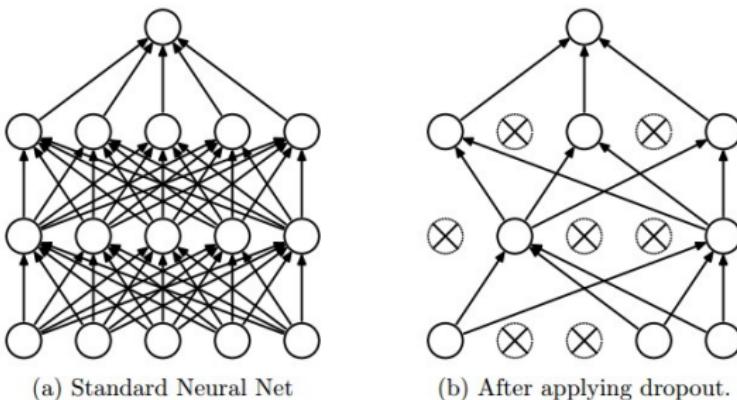


FIGURE – Dropout

# Outline

Introduction

Neural Networks, Multilayer perceptrons

Estimation of the parameters

Convolutional Neural Networks (CNN)

Conclusion

# Convolutional neural networks

- For some types of data, especially for **images**, multilayer perceptrons are **not well adapted**.
- They are defined for **vectors**. By transforming the images into vectors, we loose **spatial informations**, such as forms.
- The **convolutional neural networks (CNN)** introduced by LeCun (1998) have revolutionized image processing.
- CNN act directly on **matrices**, or even on **tensors** for images with three RGB color chanels.

# Convolutional neural networks

- CNN are now widely used for **image classification, image segmentation, object recognition, face recognition ..**

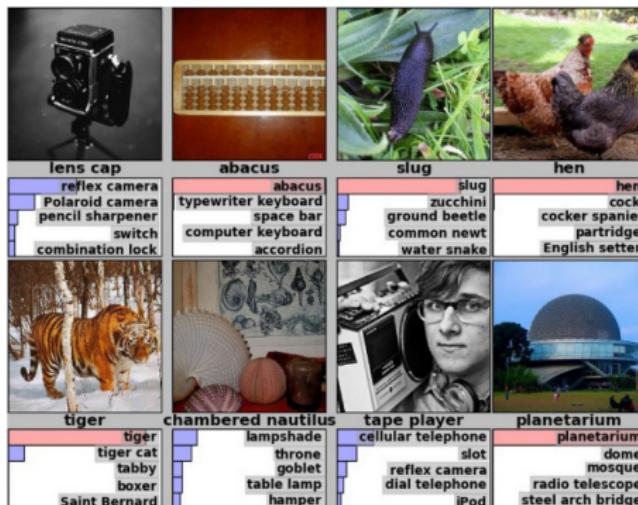


FIGURE – Image annotation

# Convolutional neural networks

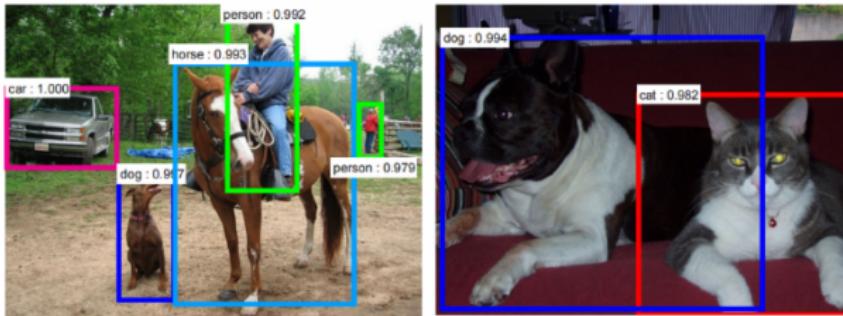


FIGURE – Image Segmentation.

# Image Classification

## Your specialized huggable recommendation

Go for it! Hug it out. With a score of **0.695779**, we're somewhat sure.



## Your specialized huggable recommendation

Don't hug that. Please. With a score of **0.999875**, we're pretty sure.



## Your specialized huggable recommendation

Don't hug that. Please. With a score of **0.778686**, we're pretty sure.



## Your specialized huggable recommendation

Go for it! Hug it out. With a score of **0.958104**, we're pretty sure.



# Layers in a CNN

- A Convolutional Neural Network is composed by several kinds of layers, that are described in this section :
  - ▶ convolutional layers
  - ▶ pooling layers
  - ▶ fully connected layers

# Convolution layer

- For 2-dimensional signals such as images, we consider the **2D-convolutions** :  $(K * I)(i, j) = \sum_{m,n} K(m, n)I(i + n, j + m)$ .
- $K$  is a **convolution kernel** applied to a 2D signal (or image)  $I$ .

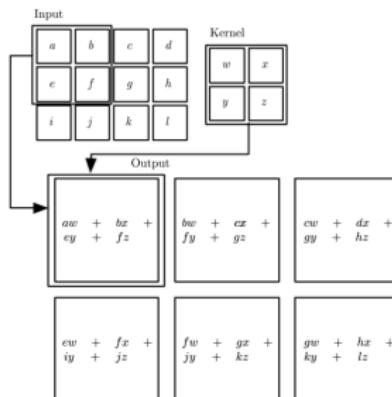


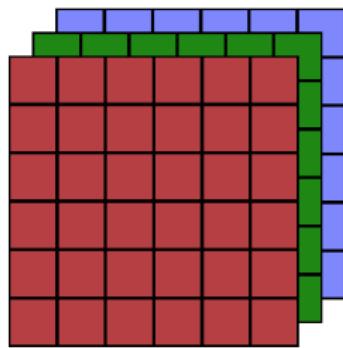
FIGURE – 2D convolution

# Convolution layer

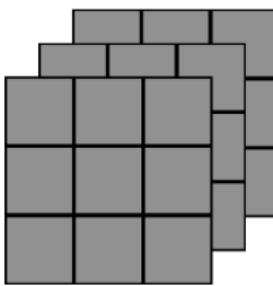
- The principle of 2D convolution is to drag a convolution kernel on the image.
- At each position, we get the convolution between the kernel and the part of the image that is currently treated.
- Then, the kernel moves by a number  $s$  of pixels,  $s$  is called the *stride*.
- When the stride is small, we get redundant information.
- Sometimes, we also add a *zero padding*, which is a margin of size  $p$  containing zero values around the image in order to control the size of the output.

# Convolution - channels

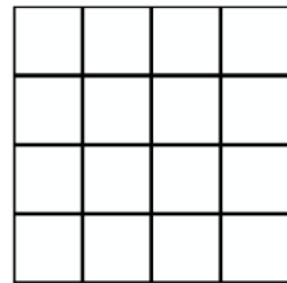
Colored image in 3 dimensions : (height, width, channels (RGB))



$6 \times 6 \times 3$



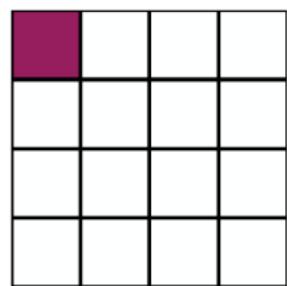
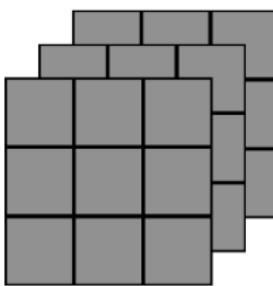
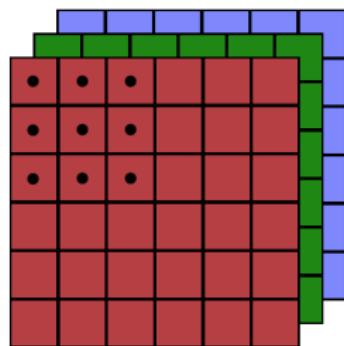
$3 \times 3 \times 3$



$4 \times 4 \times 1$

# Convolution - channels

Colored image in 3 dimensions : (height, width, channels (RGB))

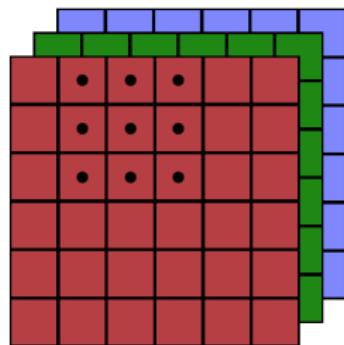


$6 \times 6 \times 3$        $3 \times 3 \times 3$        $4 \times 4 \times 1$

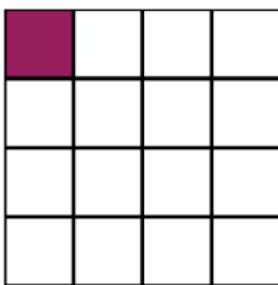
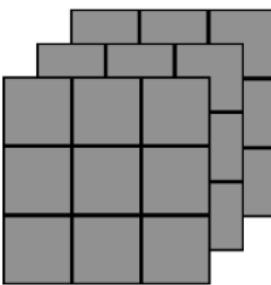
$$(F * img)(x, y) = \sum_c \sum_m \sum_n F^c(n, m) \cdot Img^c(x + m, y + n)$$

# Convolution - channels

Colored image in 3 dimensions : (height, width, channels (RGB))



**6x6x3**

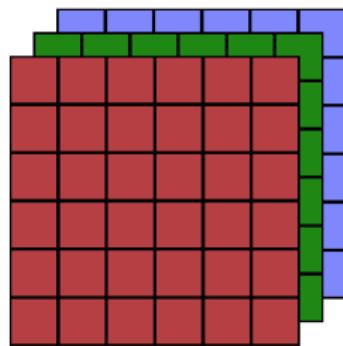


**4x4x1**

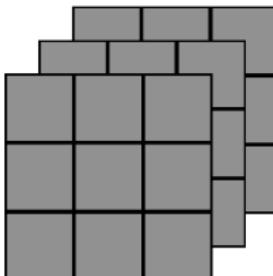
$$(F \star img)(x, y) = \sum_c \sum_m \sum_n F^c(n, m) \cdot Img^c(x + m, y + n)$$

# Convolution - channels

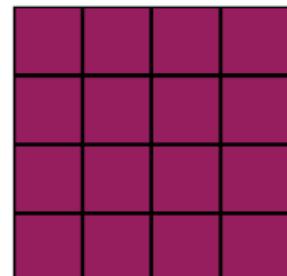
Colored image in 3 dimensions : (height, width, channels (RGB))



$6 \times 6 \times 3$



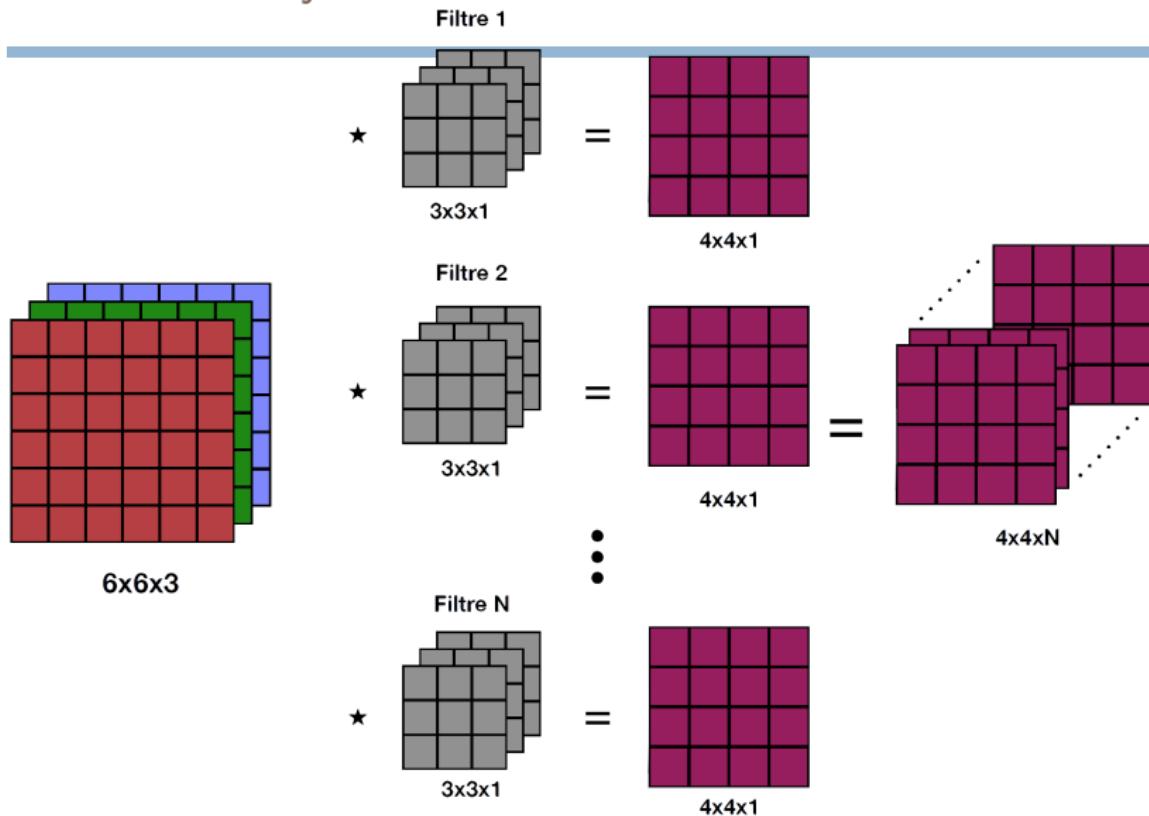
$3 \times 3 \times 3$



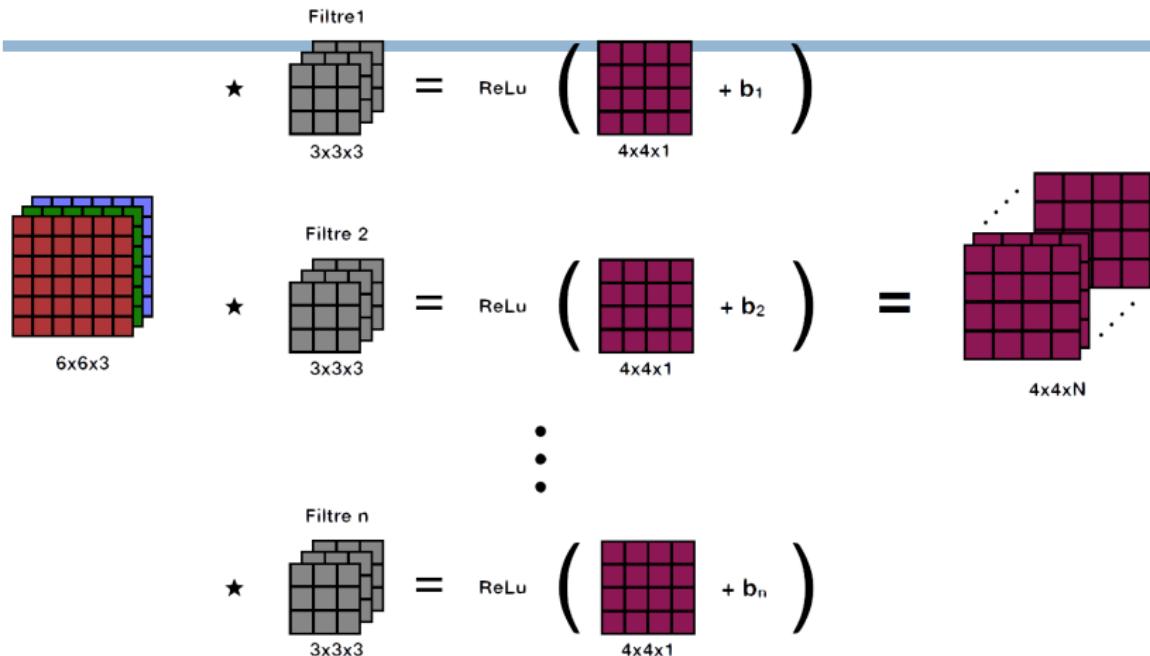
$4 \times 4 \times 1$

$$(F * img)(x, y) = \sum_c \sum_m \sum_n F^c(n, m) \cdot Img^c(x + m, y + n)$$

# Convolution - Layer



# Convolution - Layer



- Input Dimension :  $(h, w, c_i)$
- Filter Dimension :  $(f, f)$ , Nb filter :  $N$
- Output Dimension :  $(h - f + 1, w - f + 1, N)$

# Dimensions formula - Convolutional Layer

**Input Dimensions** :  $(h \times w \times c)$

- h : height of the image,
- w : width of the image,
- c : channel of the image.

**Convolution parameter** : (f, p, s)

- f : filter size
- p : padding size
- s : stride size
- n : number of output channels

**Output Dimensions** :

$$\left( \left[ \frac{h + 2p - f}{s} + 1 \right] \times \left[ \frac{w + 2p - f}{s} + 1 \right] \times n \right)$$

**Number of parameters** :  $N_p = ((h \cdot w \cdot c) + 1) \cdot n$

# Convolution layer

- The convolution operations are combined with an activation function  $\phi$  (ReLU in general) : if we consider a kernel  $K$  of size  $k \times k$ , if  $x$  is a  $k \times k$  patch of the image, the activation is obtained by sliding the  $k \times k$  window and computing  $z(x) = \phi(K * x + b)$ , where  $b$  is a bias.
- CNN learn the filters (or kernels) that are the most useful for the task that we have to do (such as classification).
- Several convolution layers are considered : the output of a convolution becomes the input of the next one.

# Pooling layer

- CNN also have *pooling layers*, which allow to reduce the dimension, also referred as *subsampling*, by taking the **mean or the maximum** on patches of the image (**mean-pooling or max-pooling**).
- Like the convolutional layers, pooling layers act on **small patches of the image**.
- If we consider  $2 \times 2$  patches, over which we take the maximum value to define the output layer, with a stride 2, we **divide by 4** the size of the image.

# Pooling layer

- Another advantage of the pooling is that it makes the network **less sensitive to small translations** of the input images.

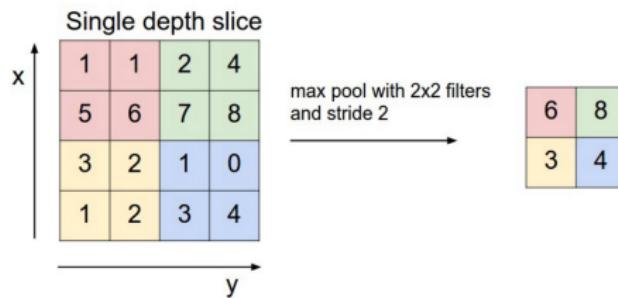


FIGURE – Maxpooling and effect on the dimension

# Fully connected layers

- After several convolution and pooling layers, the CNN generally ends with several *fully connected* layers.
- The tensor that we have at the output of these layers is *transformed into a vector* and then we add several *perceptron layers*.

# Architectures

- We have described the **different types of layers** composing a CNN.
- We now present how these layers are combined to form the **architecture of the network**.
- Choosing an architecture is very complex and this is more engineering than an exact science.
- It is therefore important to study the architectures that have **proved to be effective** and to draw inspiration from these **famous examples**.
- In the most classical CNN, we chain several times a convolution layer followed by a pooling layer and we add at the end fully connected layers.

# LeNet-5 and Mnist Classification

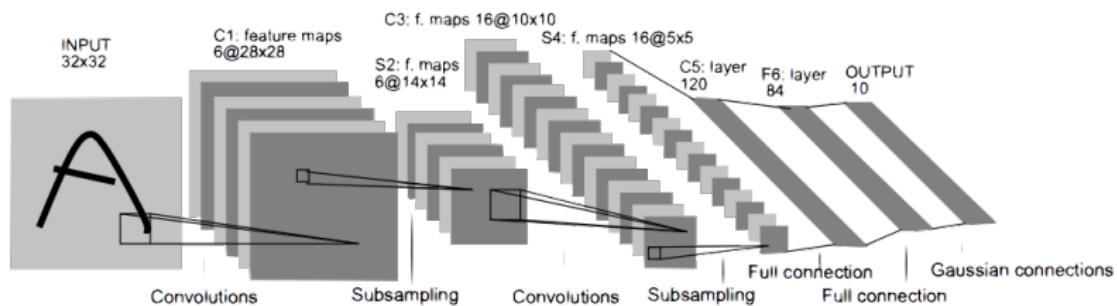
Objectif :



- Manuscript number from 0 to 9 hand-written
- Dimension (32,32,1)
- Image for training :  $N_{train} = 60.000$
- Image for testing :  $N_{test} = 10.000$

# LeNet-5 and Mnist Classification

The **LeNet network**, proposed by the inventor of the CNN, **Yann LeCun (1998)** was devoted to **digit recognition**. It is composed only on few layers and few filters, due to the computer limitations at that time.



**FIGURE –** Architecture of the network Le Net. LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998)

Size and height decrease while channel increase.

# Architectures

- With the appearance of GPU (Graphical Processor Unit) cards, much more complex architectures for CNN have been proposed, like the network **AlexNet** (Krizhevsky (2012)).
- AlexNet** won the **ImageNet competition** devoted to the classification of one million of color images ( $224 \times 224$ ) onto 1000 classes.
- AlexNet is composed of 5 convolution layers, 3 max-pooling  $2 \times 2$  layers and fully connected layers.

# Architectures

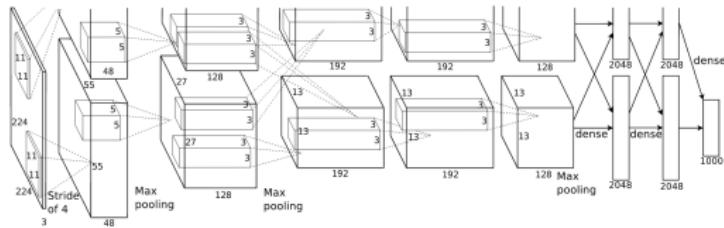


FIGURE – Architecture of the network AlexNet. Krizhevsky, A. et al (2012)

# Architectures

## Alexnet architecture

Input	227 * 227 * 3			
Conv 1	55*55*96	96	11 *11	filters at stride 4, pad 0
Max Pool 1	27*27*96		3 *3	filters at stride 2
Conv 2	27*27*256	256	5*5	filters at stride 1, pad 2
Max Pool 2	13*13*256		3 *3	filters at stride 2
Conv 3	13*13*384	384	3*3	filters at stride 1, pad 1
Conv 4	13*13*384	384	3*3	filters at stride 1, pad 1
Conv 5	13*13*256	256	3*3	filters at stride 1, pad 1
Max Pool 3	6*6*256		3 *3	filters at stride 2
FC1	4096	4096	neurons	
FC2	4096	4096	neurons	
FC3	1000	1000	neurons	(softmax logits)

# Alex Net (Krizhevsky, A. et al (2012))

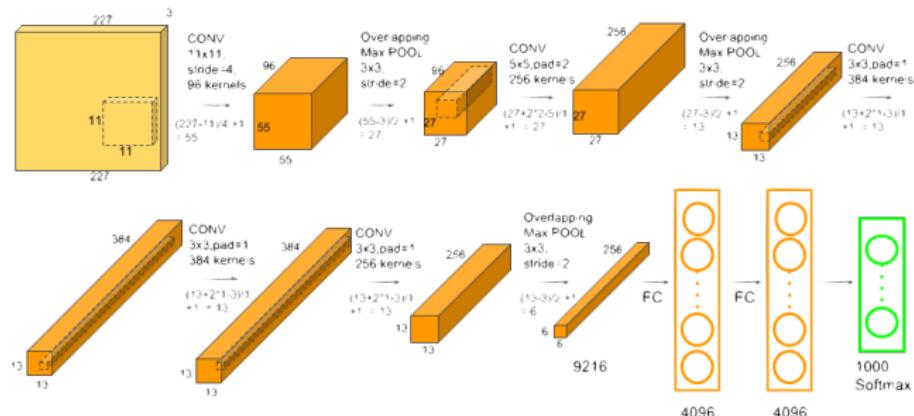


Image source : <https://www.learnopencv.com/understanding-alexnet/>

- $\simeq 60M$  parameters
- Similar to LeNet-5 but Much bigger
- Only ReLu is different.
- Multiple GPUs
- Learned on huge amount of data : **ImageNet** .

# Image net

<http://image-net.org/>

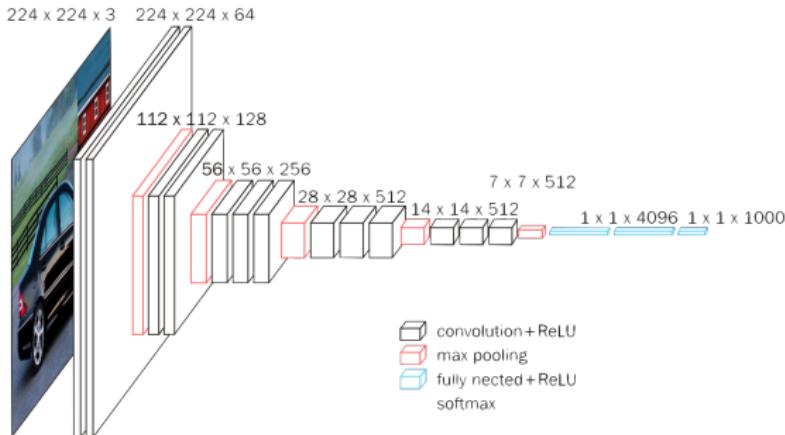


- 1000 classes
  - ▶ 1,2 M training images,
  - ▶ 100k test images.

# VGG16

Use always same layer :

- Convolution :  $3 \times 3$  filter, stride = 1, padding= SAME
- Max Pooling :  $2 \times 2$  filter, stride = 2



**FIGURE –** Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition (2014). 138M parameters.

Image source : <https://blog.datawow.io/cnn-models-ef356bc11032>, realized with Tensorflow.

# Architectures

The Figure shows a comparison of the depth and of the performances of the different networks, on the ImageNet challenge.

## Revolution of Depth

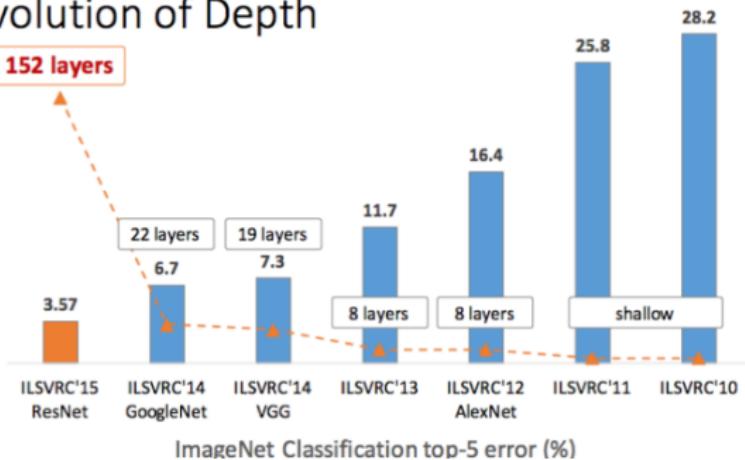


FIGURE – Evolution of the depth of the CNN and the test error

# In practice

## Pre-trained model

- Some models using previous architectures have already been trained on massive amount of data (ImageNet).
- These models are available on easily usable deep learning Framework such as Keras (Python library).
- Two ways to use them :
  - ▶ Transfer Learning
  - ▶ Fine Tuning

## Data Augmentation

# Outline

Introduction

Neural Networks, Multilayer perceptrons

Estimation of the parameters

Convolutional Neural Networks (CNN)

Conclusion

# Conclusion

- We have presented in this course the feedforward neural networks and explained how the parameters of these models can be estimated.
- The choice of the **architecture** of the network is also a crucial point. Several models can be compared by using a **validation** methods on a test sample to select the "best" model.
- The perceptrons are defined for vectors. They are not well adapted for some types of data such as images. By transforming an image into a vector, we loose spatial information, such as forms.

# Conclusion

- The **convolutional neural networks (CNN)** introduced by LeCun (1998) have revolutionized image processing.
- CNN act directly on **matrices**, or even on **tensors** for images with three RGB color channels.
- They involve complex architectures, which are in constant evolution.
- Various libraries and frameworks have been developed : we will use Keras for simplicity reasons.
- GPU is required for training the networks.
- Few theoretical foundations available.

# References

The **main references** for the course on Neural networks are :

- Ian Goodfellow, Yoshua Bengio and Aaron Courville *Deep learning*
- Bishop (1995) *Neural networks for pattern recognition*, Oxford University Press.
- Charles Ollion et Olivier Grisel *Deep learning course*  
<https://github.com/m2dsupsdlclass/lectures-labs>
- Wikistat 2.0 Github *High Dimensional and Deep Learning*,  
<https://github.com/wikistat/>, Brendan Guillouet et Béatrice Laurent