

# Deep learning introduction

## ENSTA 3A - Parcours Robotique & IA

Gianni Franchi

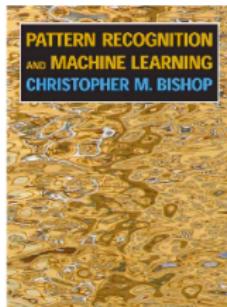
17/12/2021

# Plan

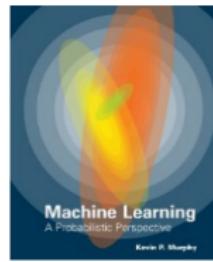
- 1 Linear Regression
- 2 Unlearned feature space-Kernel
- 3 Typical recognition Algorithm
- 4 Neural Network
  - Perceptron
  - Multilayer Perceptron (MLP)
- 5 Convolutional Neural Network
  - 1D convolution
  - 2D convolution
  - Different layers of convolutional neural network
- 6 Transformer architecture
  - Attention in NLP + the bases
  - Attention in Computer Vision (VIT)
- 7 Training a neural network
  - Gradient descent
  - Stochastic optimization
  - Initialization
- 8 Regularization
- 9 Examples of applications of classical CNN

- 1 Linear Regression
- 2 Unlearned feature space-Kernel
- 3 Typical recognition Algorithm
- 4 Neural Network
- 5 Convolutional Neural Network
- 6 Transformer architecture
- 7 Training a neural network
- 8 Regularization
- 9 Examples of applications of classical CNN

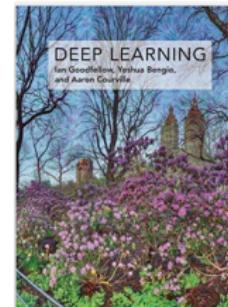
# Some references



(a)



(b)



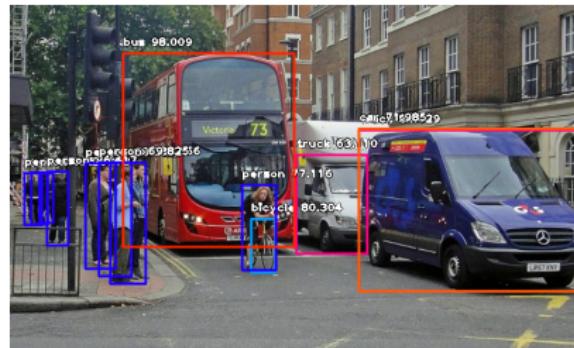
(c)

- (a) :Christopher M. Bishop " Pattern Recognition and Machine Learning "  
" Springer Verlag, 2006
- (b) : Kevin P. Murphy, " Machine Learning " MIT Press, 2013
- (c) : Ian Goodfellow , Yoshua Bengio, and Aaron Courville. " Deep  
Learning (Adaptive Computation and Machine Learning series) ", The  
MIT Press (November 18, 2016)

# Example of applications



7210414959  
0690159784  
9665407401  
3134727121  
1742351244



- classify data (images, music,...)
- denoise images
- find and localize objects in images
- segment objects in images
- translate text
- synthesize new images
- play video games

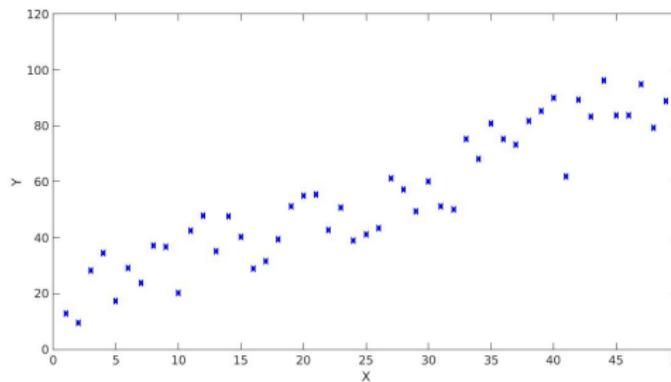
- 1 Linear Regression
- 2 Unlearned feature space-Kernel
- 3 Typical recognition Algorithm
- 4 Neural Network
- 5 Convolutional Neural Network
- 6 Transformer architecture
- 7 Training a neural network
- 8 Regularization
- 9 Examples of applications of classical CNN

## Notations and problem

First let us consider two kinds of data: the observation denoted  $x \in \mathbb{R}$  and the prediction denoted  $t \in \mathbb{R}$ .

We want to be able to predict  $t$  given the observation  $x$ . Example: we want to predict the salary given the age.

We consider that we have a set called the **training set** where we have  $N_1$  examples of pairs  $(x_i, t_i)$  with  $i \in N_1$  and we have a second set called the **testing set** composed just of the observations  $(x_i, ..)$   $i \in N_2$ .



# The linear regression

Let us consider that the observations belong to  $\mathbb{R}^D$ .

So for all  $i \in N_1$  and  $i \in N_2$  we have  $x_i \in \mathbb{R}^D$

So for simplicity and  $i \in N_1$  we have  $x_i \in \mathbb{R}^D$

A simple model often used in regression is to consider that the prediction function is given by:

$$f(\omega, x_i) = \omega_0 + \omega_1 x_{i,1} + \dots + \omega_D x_{i,D} = \omega_0 + \sum_{j=1}^D \omega_j x_{i,j}. \quad (1)$$

Our goal is to learn the parameters  $\omega = \{\omega_0, \dots, \omega_D\}$  thanks to the training set. **This model is called linear regression**, and may have some limitations.

Let us consider that the target data is given by the previous deterministic function, corrupted by Gaussian noise  $\epsilon$  of zero mean Gaussian and inverse variance  $\beta$ , such that:

$$t_i = f(\omega, x_i) + \epsilon,$$

with  $\epsilon \sim \mathcal{N}(0, 1/\beta)$ .

# The linear regression

Hence, we call  $\tau_i$  the random variable associated to the target value  $t_i$ , such that we have  $\tau \sim \mathcal{N}(f(\omega, x_i), \beta^{-1})$ , which depends on two parameters,  $\omega$  and  $\beta$  and the observation  $x_i$ .

We remind that  $X \sim \mathcal{N}(\mu, \sigma^2)$  then  $P(X = x) = \frac{1}{\sqrt{2\sigma^2}\pi} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$

Let us consider that the training set is drawn independently from the previous law. Then we can write the likelihood function of the parameters  $\omega$  and  $\beta$ :

$$\mathcal{L}(t_1, \dots, t_{N_1}/\omega, \beta) = \prod_{i=1}^{N_1} \mathcal{N}(f(\omega, x_i), \beta^{-1}).$$

$$\mathcal{L}(t_1, \dots, t_{N_1}/\omega, \beta) = \prod_{i=1}^{N_1} \frac{\sqrt{\beta}}{\sqrt{2\pi}} \exp\left(\frac{-\beta(t_i - f(\omega, x_i))^2}{2}\right).$$

Taking the logarithm of the likelihood function, we have:

$$\log \mathcal{L}(t_1, \dots, t_n/\omega, \beta) = \sum_{i=1}^n \left( \frac{1}{2} \log \beta - \frac{1}{2} \log 2\pi - \frac{\beta}{2} (t_i - f(\omega, x_i))^2 \right).$$

# The linear regression

If we want to find the set of parameters that maximize the likelihood, we have first to derive it according to each of the parameters of the log-likelihood, and set it to zero. On the previous expression the term that depends just on  $\omega$  is:

$$E_d(\omega) = \frac{\beta}{2} \sum_{i=1}^{N_1} (t_i - f(\omega, x_i))^2.$$

# The linear regression

We can rewrite it in a matrix form. First let us define the following matrices:  $t \in M_{N_1, 1}(\mathbb{R})$  is defined by:

$$t = \begin{pmatrix} t_1 \\ \vdots \\ t_{N_1} \end{pmatrix}$$

$x \in M_{N_1, D+1}(\mathbb{R})$  is defined by:

$$x = \begin{pmatrix} 1, x_{1,1} & \dots & x_{1,D} \\ \vdots & \ddots & \vdots \\ 1, x_{N_1,1} & \dots & x_{N_1,D} \end{pmatrix}$$

$\omega \in M_{D+1, 1}(\mathbb{R})$  is defined by:

$$\omega = \begin{pmatrix} \omega_0 \\ \vdots \\ \omega_D \end{pmatrix}$$

# The linear regression

We can rewrite  $E_D$  in a matrix form

$$E_d(\omega) = \frac{\beta}{2}(t - x\omega)^t(t - x\omega).$$

$$E_d(\omega) = \frac{\beta}{2}(t^t \cdot t + \omega^t x^t x \omega - t^t \cdot x \omega - \omega^t x^t \cdot t).$$

However we know that  $\frac{\partial \omega^t x^t x \omega}{\partial \omega} = 2 * (x^t x)\omega$  and  
 $\frac{\partial t^t \cdot x \omega}{\partial \omega} = \frac{\partial \omega^t x^t \cdot t}{\partial \omega} = 2 * x^t \cdot t$

$$\frac{\partial}{\partial \omega} E_d(\omega) = \beta((x^t x)\omega - x^t \cdot t).$$

We can set it to zero, to finally obtain that:

$$\omega_{ML} = (x^t x)^{-1} x^t t, \quad (2)$$

# The linear regression

It is also possible to estimate  $\beta_{ML}$  as:

$$\beta_{ML} = \frac{1}{N_1} \sum_{i=1}^{N_1} (t_i - \omega_{ML}^t x_i)^2, \quad (3)$$

such that  $\beta_{ML}$  provides us information on the precision of the regression.

# The linear regression

Instead of solving :

$$E_d(\omega) = \frac{\beta}{2} \sum_{i=1}^{N_1} (t_i - f(\omega, x_i))^2.$$

In order to control over-fitting, the total error function to be minimized takes the form:

$$E_d(\omega) = \frac{\beta}{2} \sum_{i=1}^{N_1} (t_i - f(\omega, x_i))^2 + \frac{\lambda}{2} \omega^t \omega.$$

By following the same calculus as previously the solution is:

$$\omega_{ML} = (\lambda I_{D+1} + x^t x)^{-1} x^t t, \quad (4)$$

# The linear regression

We are now able to learn a simple function  $f$  linking the target  $t$  and the observation  $x$ .

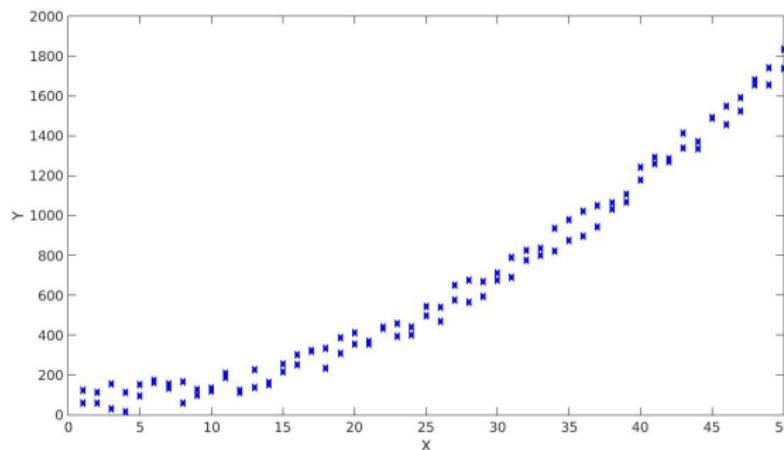
if  $t$  is continuous it is a regression

if  $t$  is discrete it is a classification

- 1 Linear Regression
- 2 Unlearned feature space-Kernel
- 3 Typical recognition Algorithm
- 4 Neural Network
- 5 Convolutional Neural Network
- 6 Transformer architecture
- 7 Training a neural network
- 8 Regularization
- 9 Examples of applications of classical CNN

## The non linear regression

In the case where the data do not follow a linear model, the linear regression might not be the best solution.



**A solution:** do not use  $x$  but another data representing  $x$ .

## The non linear regression

In the case where the data do not follow a linear model, the linear regression might not be the best solution.

We prefer to consider a more general model:

$$f(\omega, x_i) = \omega_0 + \sum_{j=1}^{D_2} \omega_j \phi_j(x_i), \quad (5)$$

where  $\phi_j(x_i) \in \mathbb{R}^{D_2}$  a function describing  $x_i$  with  $D_2$  the dimension of the descriptor. There are a lot of possible descriptors :

$$\phi_j(x_i) = x_{i,j}^M, \quad (6)$$

where the power  $M$  is a hyper-parameter.

We can also choose a feature space represented by a kernel<sup>1</sup>.

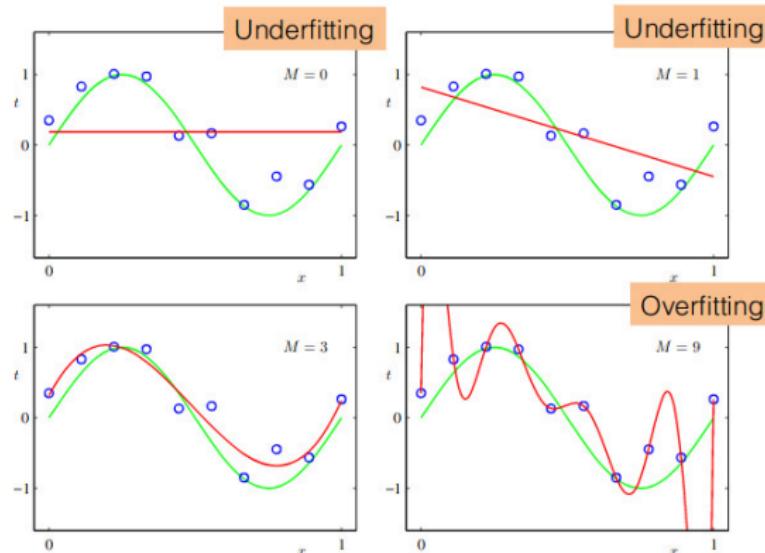
One can use the SIFT descriptor of any descriptors one want to use.

---

<sup>1</sup>Scholkopf, Bernhard, and Alexander J. Smola. Learning with kernels: support vector machines, regularization, optimization, and beyond. MIT press, 2001.

# The non linear regression

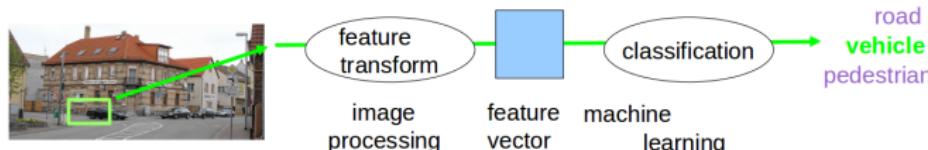
What value of  $M$  should we choose?



[C. Bishop, Pattern recognition and Machine learning, 2006]

**A solution** learn the feature space.

# Typical recognition Algorithm



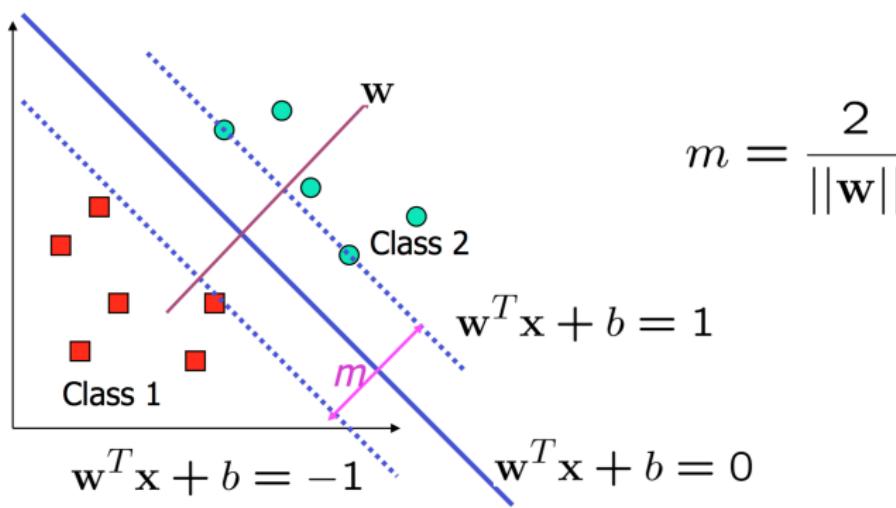
## Standard procedure

- Feature transform: problem-dependent, hand-crafted, transforms image into a form useful for classification
- Classification: generic, trained, takes feature vector and produces decision

# Support Vector Machines

## Large margin classifier

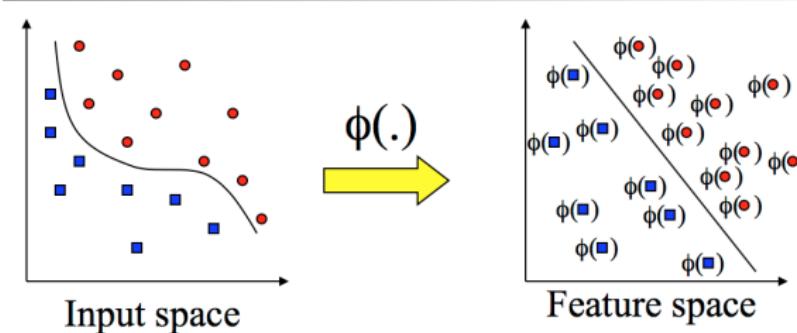
- Binary classification
- Finds the boundary that 'best' separates two classes
- Implemented as an optimization problem :
  - Find  $w$  for maximizing  $m$
  - With constraints that all points are well classified



# Support Vector Machines

## Application for non linear problems

Project input in a space where they are linearly separable



Note: feature space is of higher dimension than the input space in practice

We can use the 'Kernel trick' : only the dot product of two feature is needed, no need to create feature map  $\phi$ .

- 1 Linear Regression
- 2 Unlearned feature space-Kernel
- 3 Typical recognition Algorithm
- 4 Neural Network
- 5 Convolutional Neural Network
- 6 Transformer architecture
- 7 Training a neural network
- 8 Regularization
- 9 Examples of applications of classical CNN

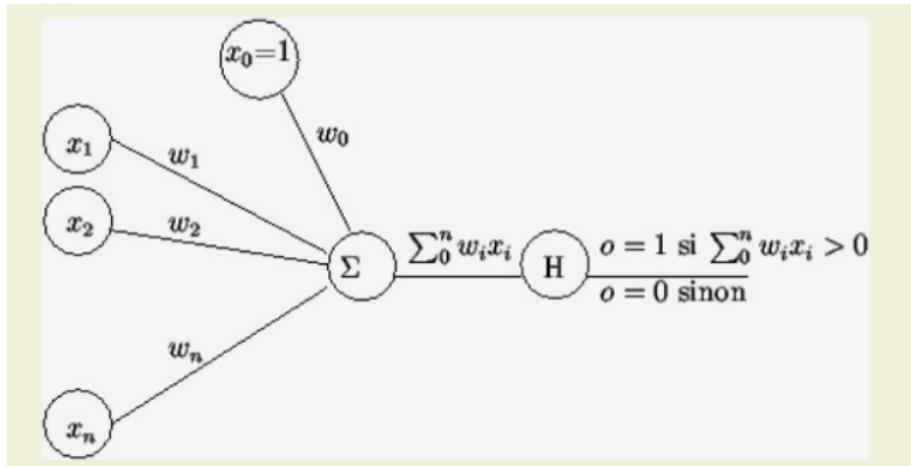
# History of Deep learning

Deep Learning is a long story. It all started with the Perceptron:



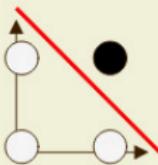
# Perceptron algorithm

Deep Learning is a long story. It all started with perceptron:

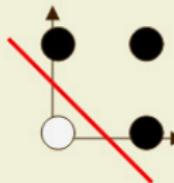


# Perceptron algorithm

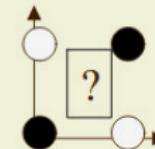
The issue is the XOR. How to solve it?



AND



OR

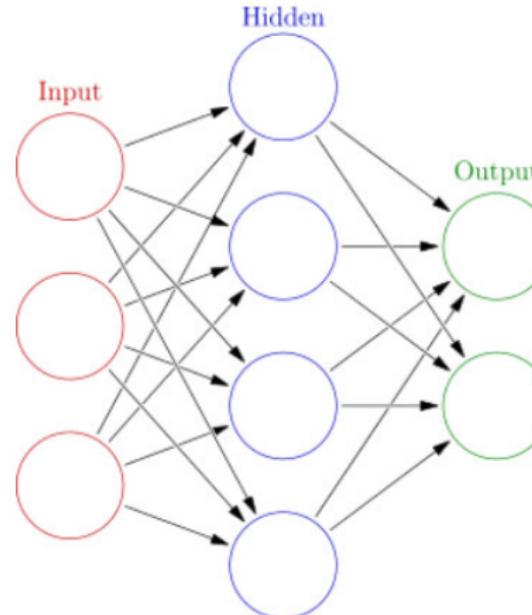


XOR

# neural network

(Artificial) neural networks are approaches which attempt to find a mathematical representation of how our biological system processes information.

Let us start with the following simple neural network:



# The Neural Network

In regression, the optimization problem was modeled by:

$$f(\omega, x_i) = \omega_0 + \sum_{j=1}^D \omega_j x_{i,j}. \quad (7)$$

Here we will build a first neuron denoted  $c_k$  with  $k \in [1, K_1]$  (in this example  $K_1 = 4$  and  $D = 3$ ) :

$$c_k = \omega_{0,k}^{(1)} + \sum_{j=1}^D \omega_{j,k}^{(1)} v_{i,j}. \quad (8)$$

each  $c_k$  is a neuron of the first layer. The superscript (1) indicates that these parameters are the parameters of the first hidden layer. Then, a nonlinear activation function  $a$  is applied on these quantities  $c_k$ :

$$z_k = a^{(1)}(c_k). \quad (9)$$

with  $k \in [1, K_1]$ .

# The Neural Network

We can choose different kinds of activation functions, typically:

- A sigmoid function  $a(x) = \frac{1}{1+e^{-x}}$ ;
- $a(x) = \tanh(x)$ ;
- Rectified Linear Unit (ReLU):  $a(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ .

We have now the  $K_1$  first neurons  $c_1, c_2, \dots, c_{K_1}$  (according to the example  $K_1 = 4$ ).

Thanks to activation functions the neural network acts like human neurons. Moreover, the activation functions allow the neural network to approximate any functions.

# The Neural Network

On the output of the first layer, a second linear combination is applied:

$$d_k = \omega_{0,k}^{(2)} + \sum_{k_1=1}^{K_1} \omega_{k_1,k}^{(2)} z_{k_1}. \quad (10)$$

with  $k \in [1, K_2]$  (on this example  $K_2 = 2$ ).

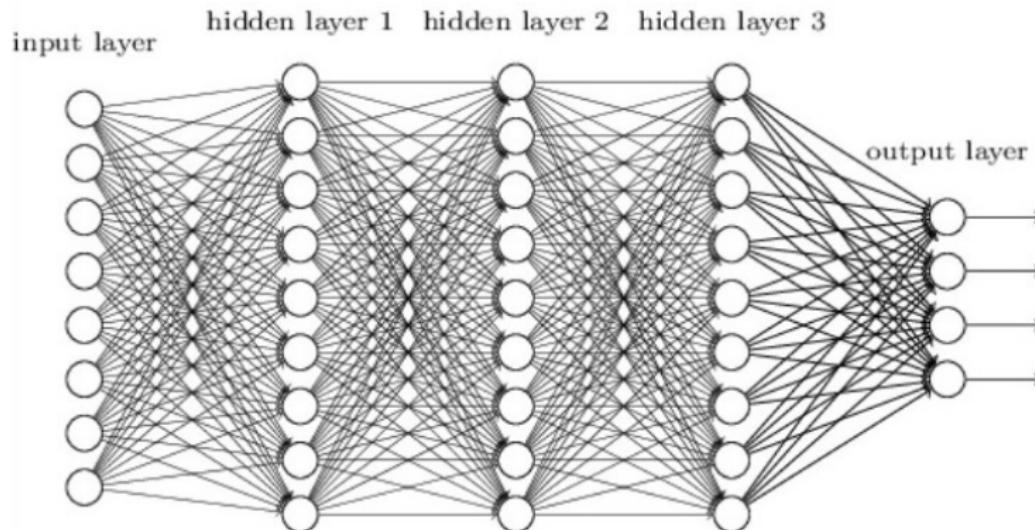
In this example,  $d_1$  and  $d_2$  are the outputs of the CNN.

To summarize, the output is equal to :

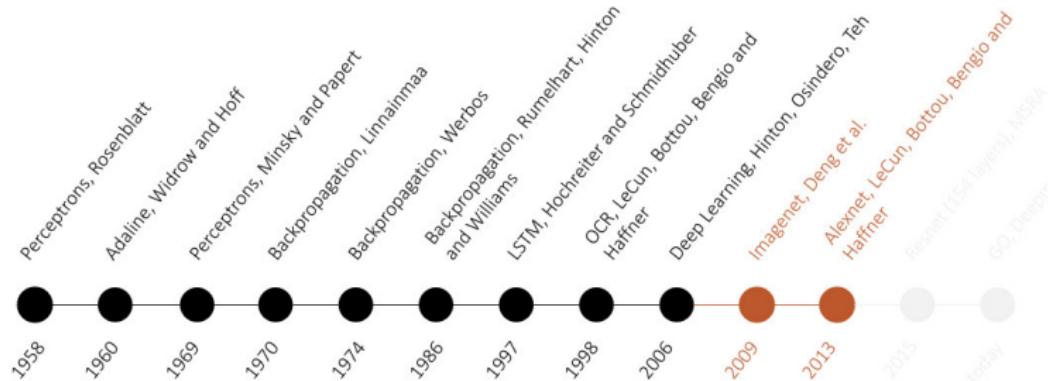
$$d_k = \omega_{0,k}^{(2)} + \sum_{k_1=1}^{K_1} \omega_{k_1,k}^{(2)} a^{(1)}(\omega_{0,k_1}^{(1)} + \sum_{j=1}^D \omega_{j,k_1}^{(1)} v_{i,j}). \quad (11)$$

In addition we can add multiple layers. So the function represented by the neural network can be really complicated.

# Neural network deeper



# Story of Neural network



- 1 Linear Regression
- 2 Unlearned feature space-Kernel
- 3 Typical recognition Algorithm
- 4 Neural Network
- 5 Convolutional Neural Network
- 6 Transformer architecture
- 7 Training a neural network
- 8 Regularization
- 9 Examples of applications of classical CNN

# 1D convolution

For real functions  $f, g$  defined on the set  $\mathbb{Z}$  of integers, the discrete convolution of  $f$  and  $g$  is given by:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] \quad (12)$$

or equivalently (see commutativity) by:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[n-m]g[m]. \quad (13)$$

when  $g$  and  $f$  have finite supports;  $g$  in the set

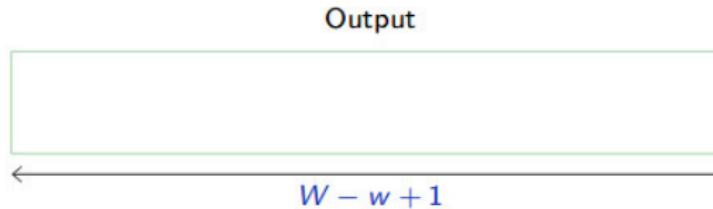
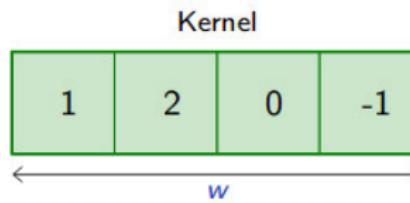
$\{-M, -M+1, \dots, M-1, M\}$  and  $f$  in  $\{0, 1, \dots, N-1, N\}$  a finite summation is used:

$$(f * g)[n] = \sum_{m=-M}^M f[n-m]g[m] \quad \forall n \in [M, N-M] \quad (14)$$

with  $M \leq N$

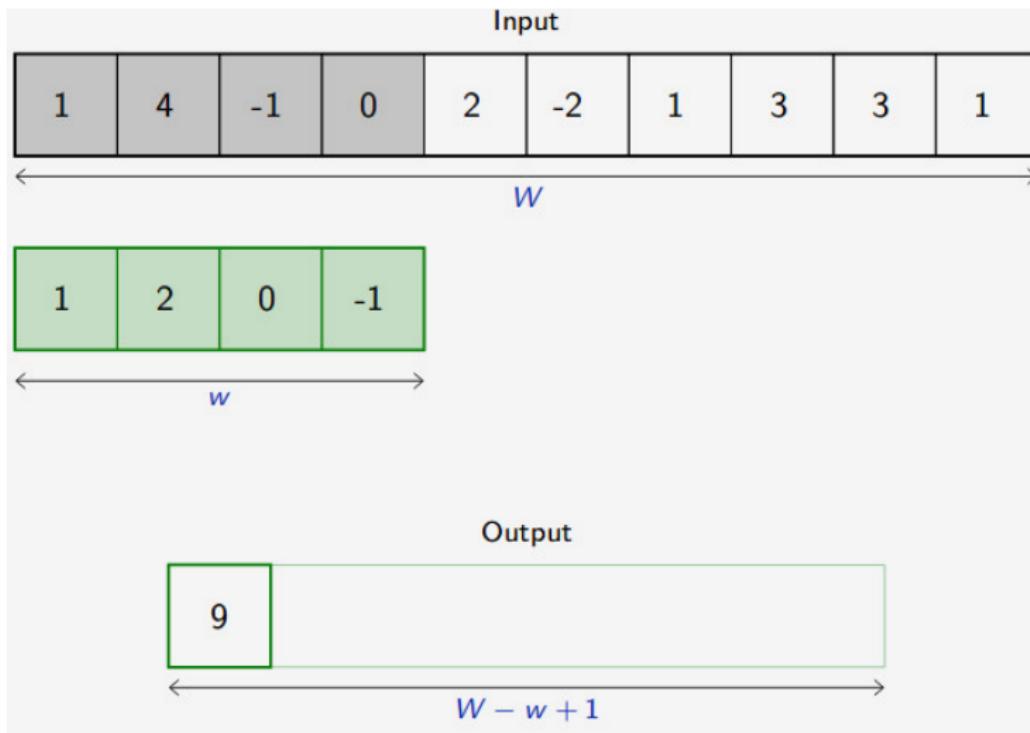
# Example 1D convolution for deep learning<sup>2</sup>

Be careful, this is the cross-correlation.



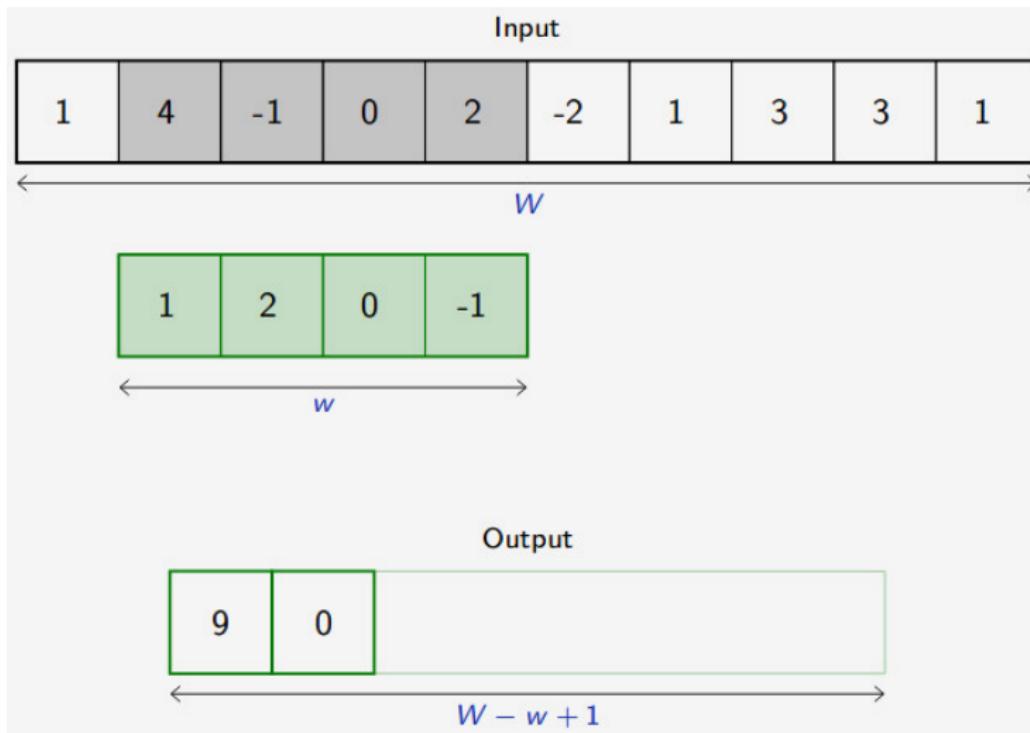
<sup>2</sup>Credits: Francois Fleuret

# Example 1D convolution for deep learning<sup>3</sup>



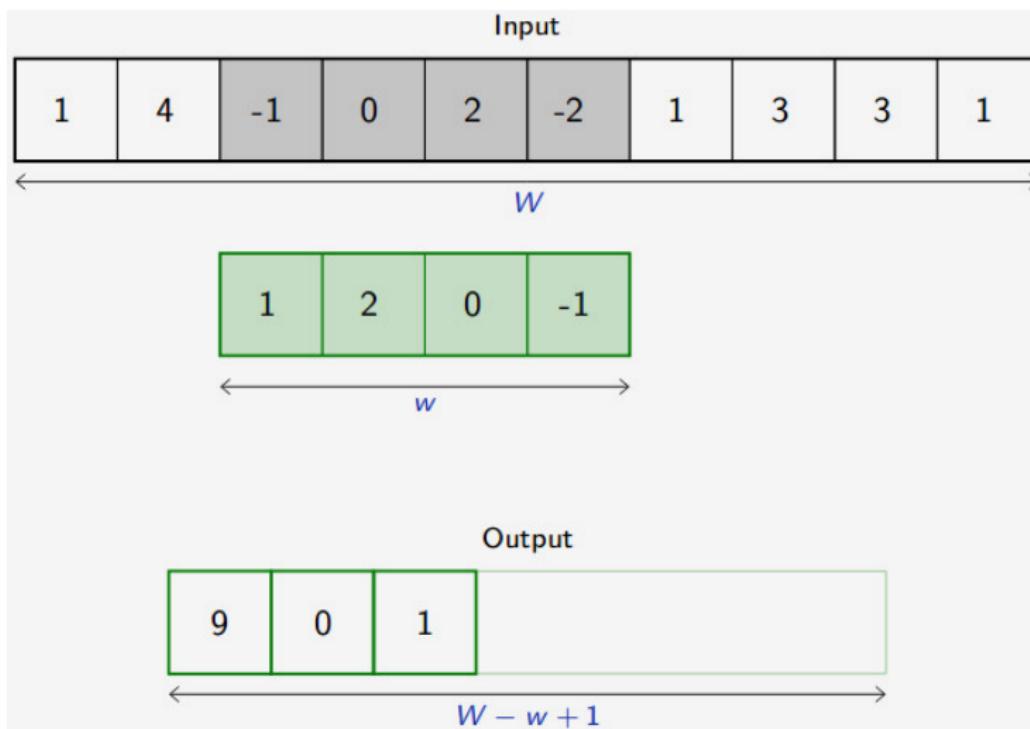
<sup>3</sup>Credits: Francois Fleuret

# Example 1D convolution for deep learning<sup>4</sup>



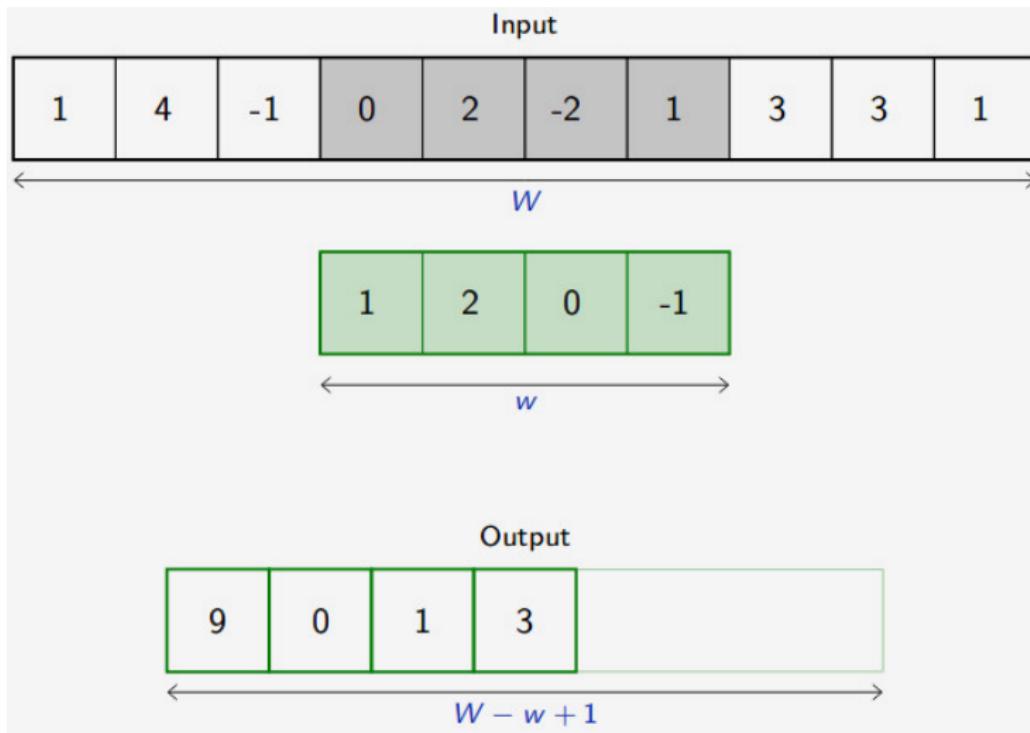
<sup>4</sup>Credits: Francois Fleuret

# Example 1D convolution for deep learning<sup>5</sup>



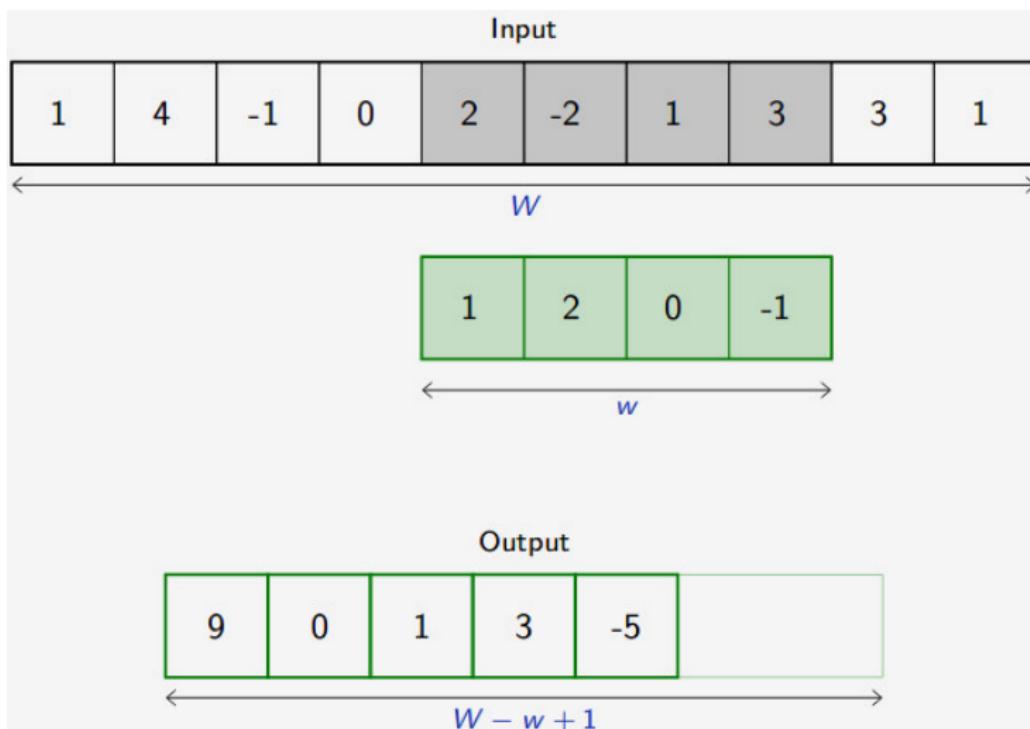
<sup>5</sup>Credits: Francois Fleuret

# Example 1D convolution for deep learning<sup>6</sup>



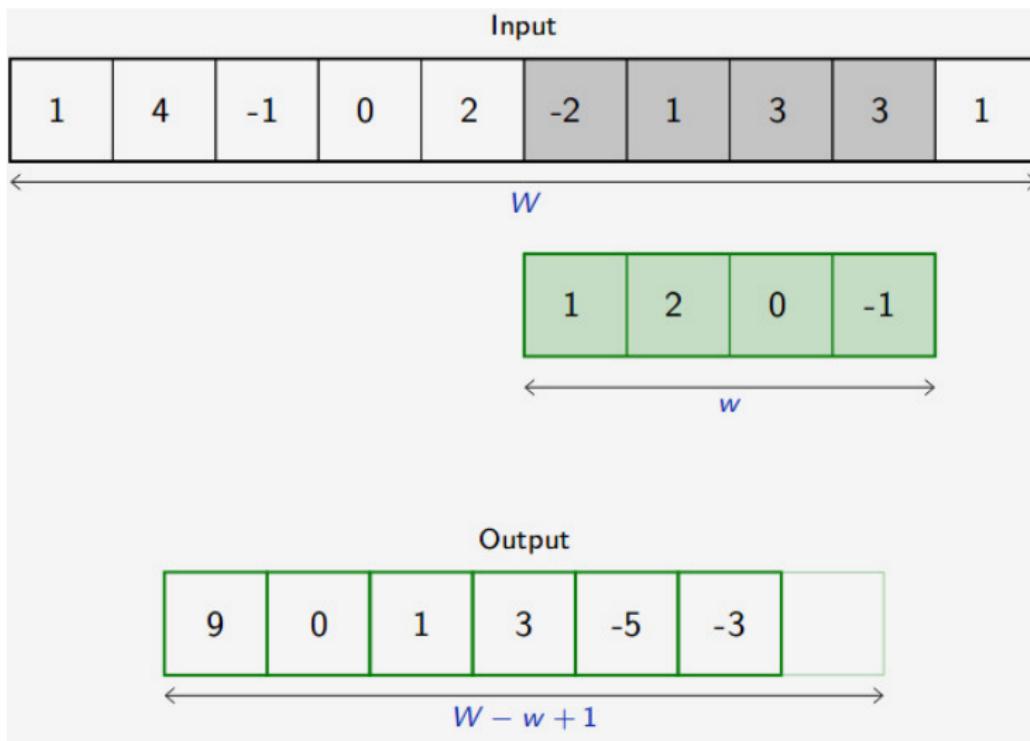
<sup>6</sup>Credits: Francois Fleuret

# Example 1D convolution for deep learning<sup>7</sup>



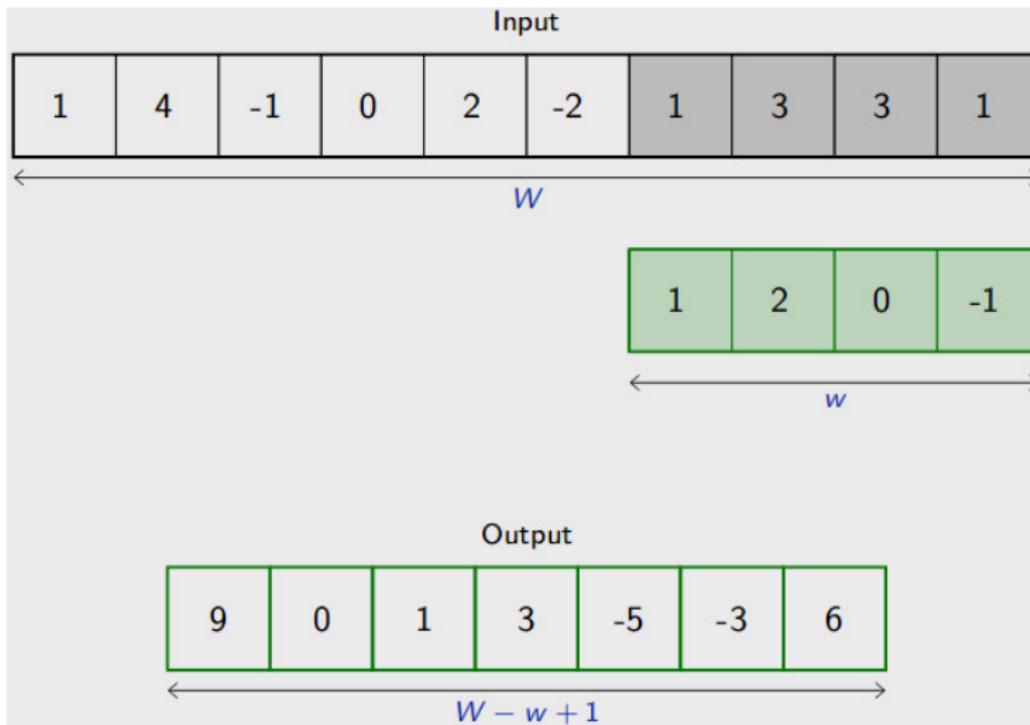
<sup>7</sup>Credits: Francois Fleuret

# Example 1D convolution for deep learning<sup>8</sup>



<sup>8</sup>Credits: Francois Fleuret

# Example 1D convolution for deep learning<sup>9</sup>



<sup>9</sup>Credits: Francois Fleuret

## 2D convolution

Similarly to the 1D case, let us define two functions  $f, g$ .  $g$  is a function of two variables defined in the set  $\{-M, -M + 1, \dots, M - 1, M\}^2$  and  $f$  in  $\{0, 1, \dots, N - 1, N\}^2$ . We can define the 2D convolution for all  $(n_1, n_2) \in [M, N - M]^2$

$$(f * g)[n_1, n_2] = \sum_{m_1=-M}^M \sum_{m_2=-M}^M f[n_1 - m_1, n_2 - m_2]g[m_1, m_2] \quad (15)$$

However, color images are discrete functions of two variables with values in  $\mathbb{R}^3$ .

$$(f * g)[n_1, n_2] = \sum_{k=0}^3 \sum_{m_1=-M}^M \sum_{m_2=-M}^M f[n_1 - m_1, n_2 - m_2, k]g[m_1, m_2, k] \quad (16)$$

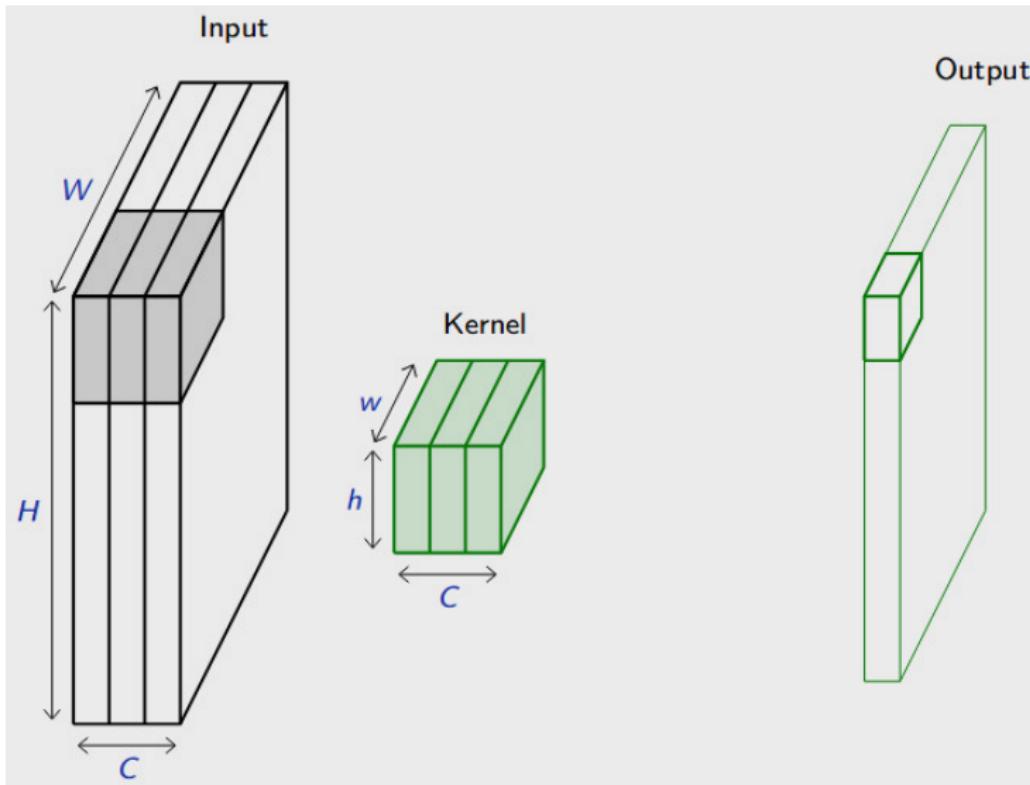
## 2D convolution

We note that in deep learning, we do not use the convolution but the cross-correlation, and we call it the convolution.

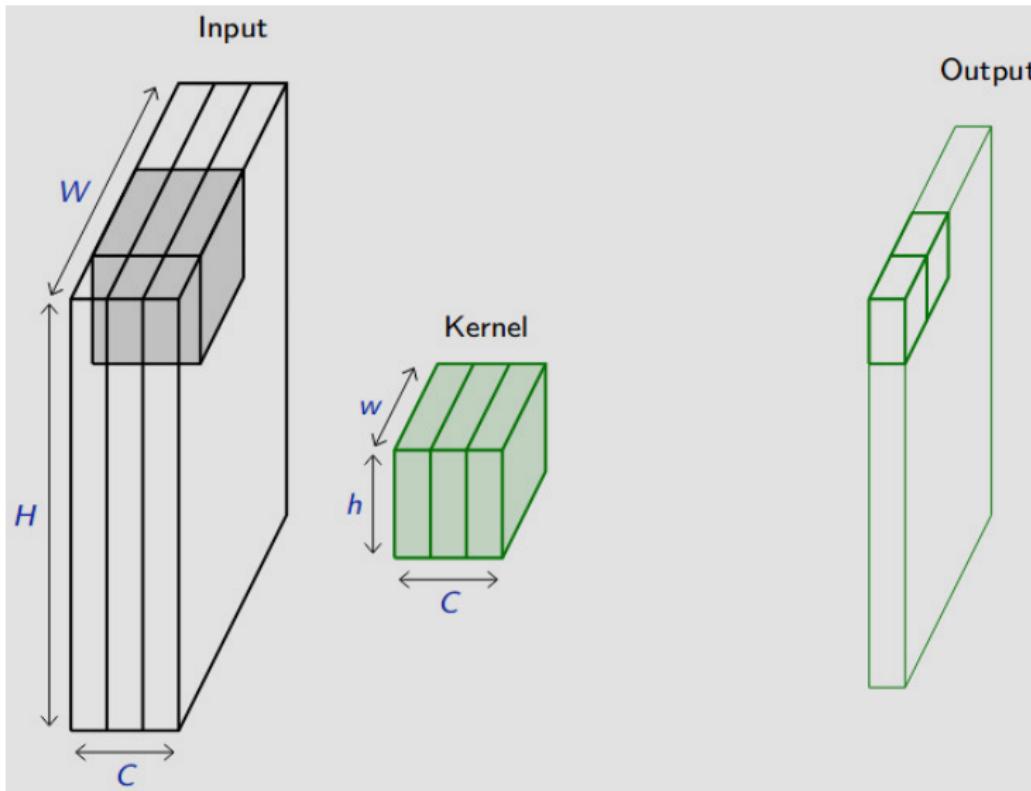
Here is the definition of the convolution used in most of the deep learning libraries:

$$(f * g)[n_1, n_2] = \sum_{k=0}^3 \sum_{m_1=-M}^M \sum_{m_2=-M}^M f[n_1 + m_1, n_2 + m_2, k]g[m_1, m_2, k]. \quad (17)$$

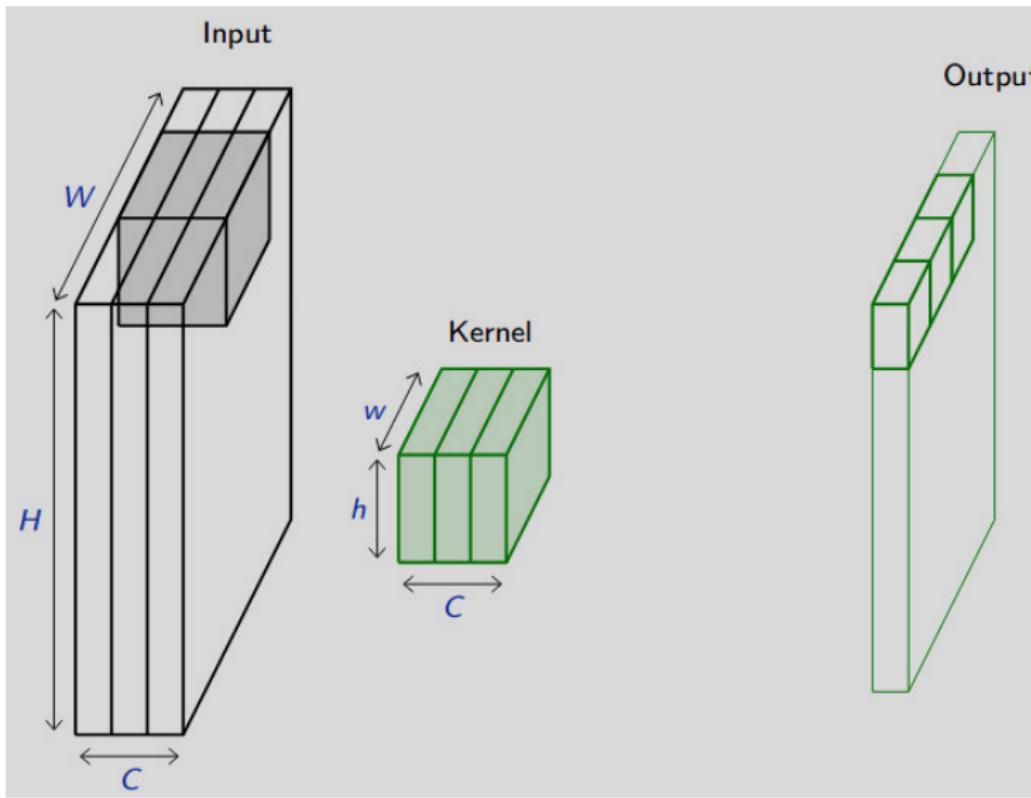
# Example 2D convolution<sup>10</sup>

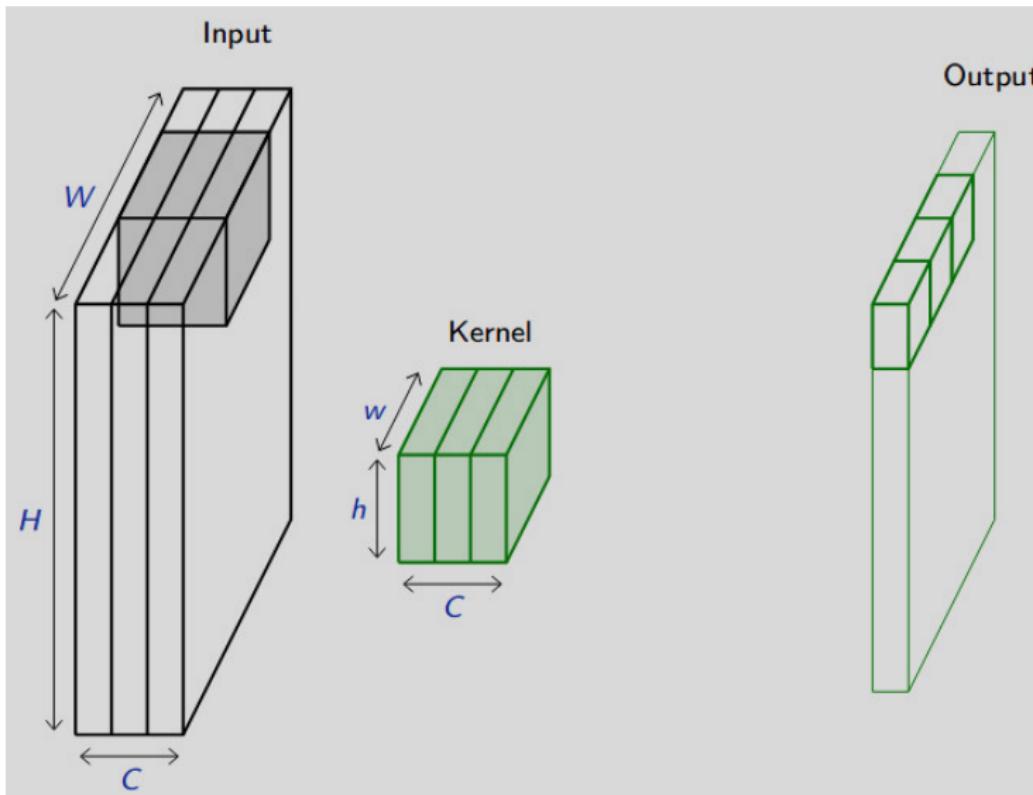


# Example 2D convolution<sup>11</sup>

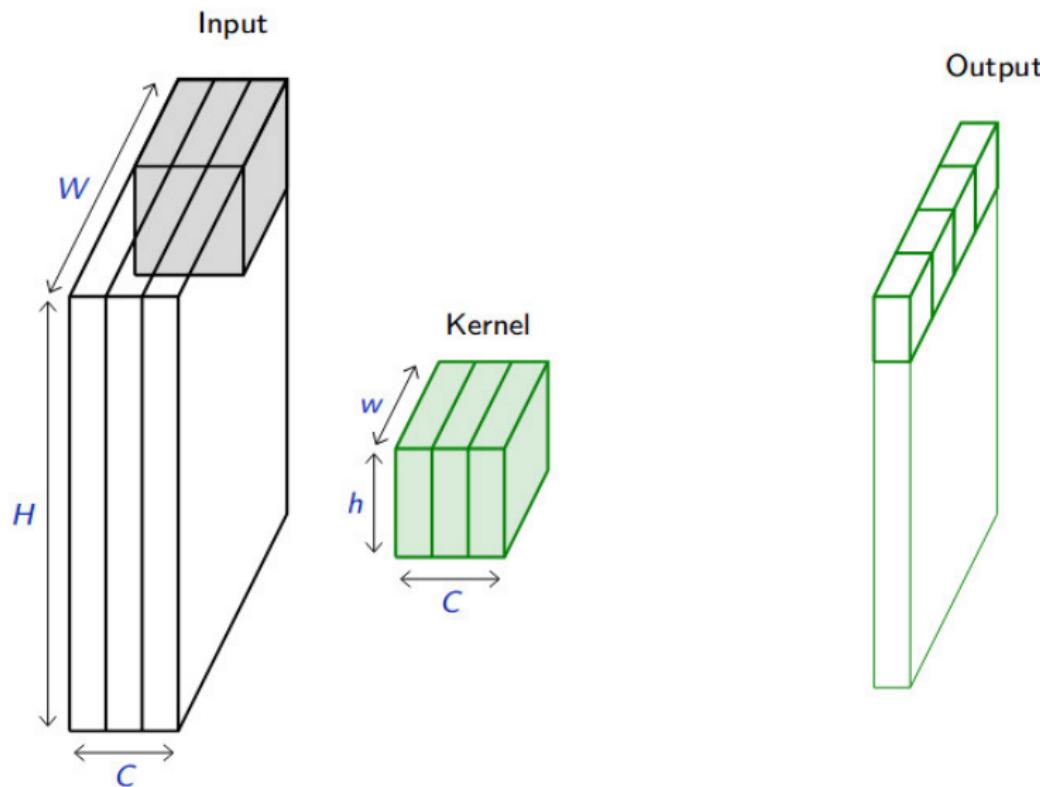


<sup>11</sup>Credits: Francois Fleuret

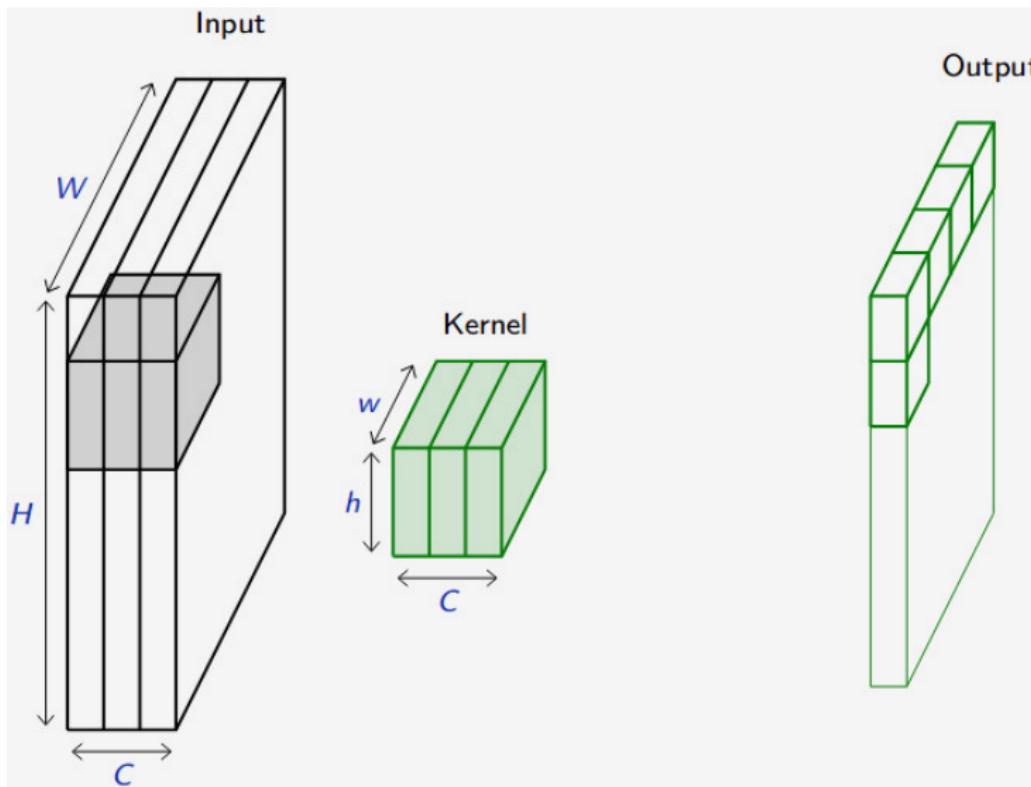
Example 2D convolution<sup>12</sup>

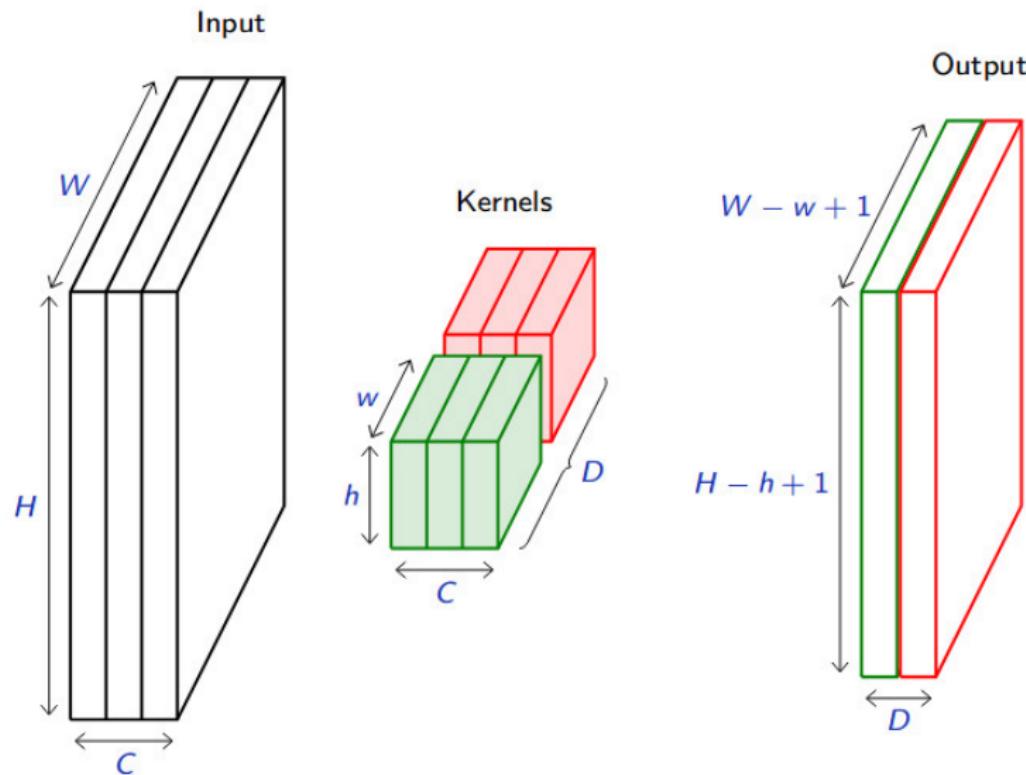
Example 2D convolution<sup>13</sup>

# Example 2D convolution<sup>14</sup>



# Example 2D convolution<sup>15</sup>



Example 2D convolution<sup>16</sup>

## 2D convolution

- Let  $f \in \mathbb{R}^{C_{\text{in}} \times H \times W}$  be an image. it is a **3D tensor** called the **input feature map**.
- Let  $u \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times h \times w}$  be a kernel across the input feature map, along its height and width. The size  $h \times w$  is the size of the receptive field.
- The final output  $o$  is a 3D tensor of size  $C_{\text{out}} \times (H_{\text{out}}) \times (W_{\text{out}})$  called the **output feature map**

$$o[C_{\text{out},j}] = \text{bias}[C_{\text{out},j}] + \sum_{k=0}^{C_{\text{in}}} \sum_{n=0}^{h-1} \sum_{m=0}^{w-1} f[k, n+j, m+i] u[C_{\text{out},j}, k, n, m] \quad (18)$$

$$C_{\text{out}} \times (H - h + 1) \times (W - w + 1)$$

## 2D convolution

The output **feature map** size  $C_{\text{out}} \times (H_{\text{out}}) \times (W_{\text{out}})$  depends on :

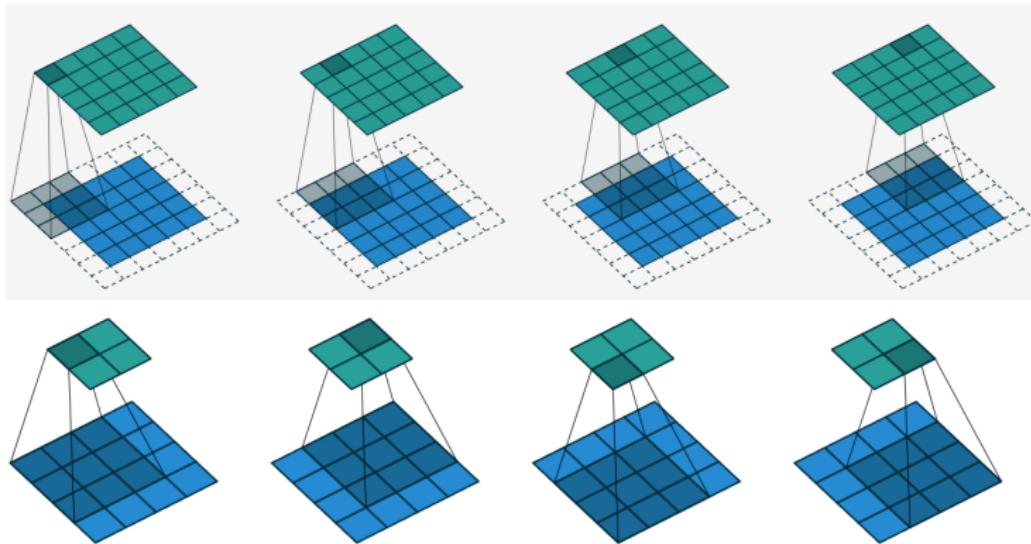
- The padding which specifies number of zeros concatenated at the beginning and at the end of an axis
- The stride which specifies a step size when moving the kernel across the signal.
- The dilation which modulates the expansion of the filter without adding weights.

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (h - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (w - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

# 2D convolution<sup>17</sup>

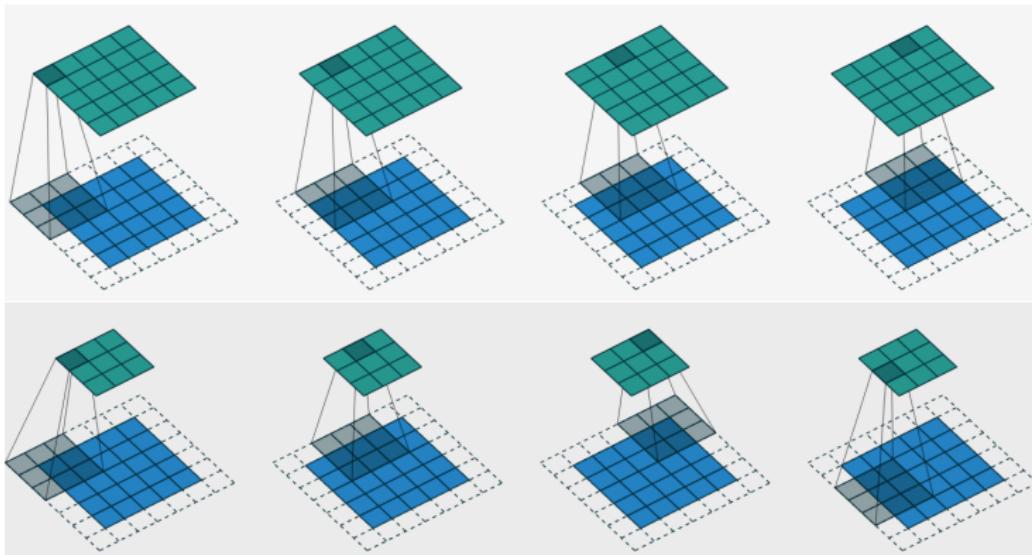
**Padding** is useful to control the spatial dimension of the feature map, for example to keep it constant across layers.



<sup>17</sup>Credits: <https://arxiv.org/pdf/1603.07285.pdf>

## 2D convolution<sup>18</sup>

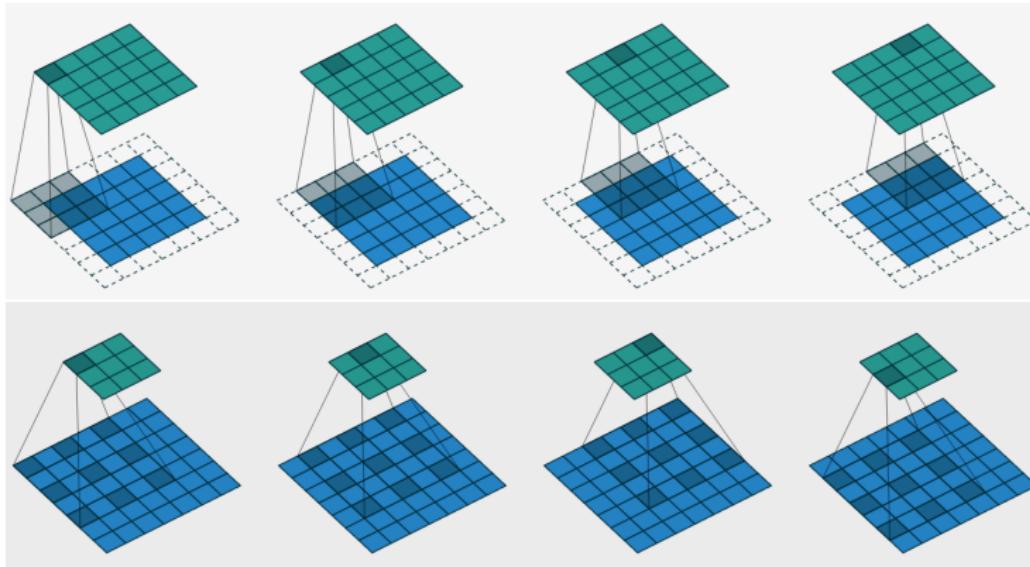
**Stride** is useful to reduce the spatial dimension of the feature map by a constant factor.



<sup>18</sup>Credits: <https://arxiv.org/pdf/1603.07285.pdf>

## 2D convolution<sup>19</sup>

The **dilation** modulates the expansion of the kernel. Having a dilation coefficient greater than one increases the units receptive field size without increasing the number of parameters.



<sup>19</sup>Credits: <https://arxiv.org/pdf/1603.07285.pdf>

# Convolutions as matrix multiplications

As a guiding example, let us consider the convolution of single-channel tensors  $x \in \mathbb{R}^{4 \times 4}$  and  $u \in \mathbb{R}^{3 \times 3}$ :

$$x \circledast u = \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} \circledast \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

# Convolutions as matrix multiplications

The convolution operation can be equivalently re-expressed as a single matrix multiplication:

the convolutional kernel  $u$  is rearranged as a sparse Toeplitz circulant matrix, called the convolution matrix:

$$U = \begin{pmatrix} 1 & 4 & 1 & 0 & 1 & 4 & 3 & 0 & 3 & 3 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 1 & 0 & 1 & 4 & 3 & 0 & 3 & 3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 4 & 1 & 0 & 1 & 4 & 3 & 0 & 3 & 3 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 4 & 1 & 0 & 1 & 4 & 3 & 0 & 3 & 3 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 4 & 1 & 0 & 1 & 4 & 3 & 0 & 3 \end{pmatrix}$$

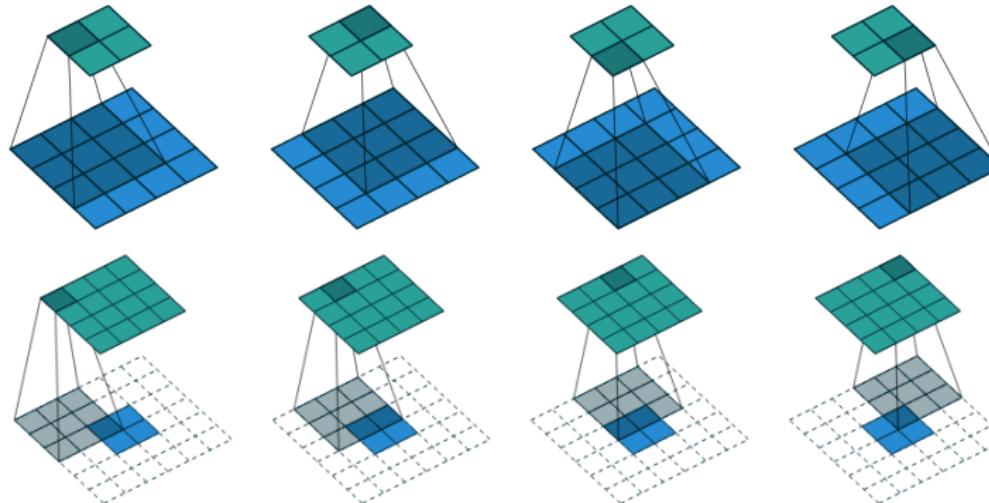
the input  $x$  is flattened row by row, from top to bottom:

$$x = (4 \ 5 \ 8 \ 7 \ 1 \ 8 \ 8 \ 8 \ 3 \ 6 \ 6 \ 4 \ 6 \ 5 \ 7 \ 8)^T$$

Then,  $v(x) = (122 \ 148 \ 126 \ 134)^T$  which we can reshape to a  $2 \times 2$  matrix to obtain  $x \circledast u$ .

# Transposed convolution<sup>20</sup>

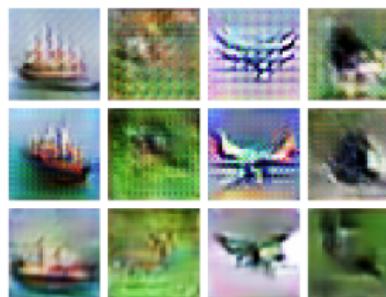
The need for **transposed convolutions** generally arises from the desire to use a transformation going in the opposite direction of a normal convolution. This operation is known as **deconvolution**.



<sup>20</sup>Credits: <https://arxiv.org/pdf/1603.07285.pdf>

# Transposed convolution 21

Input	Kernel					Output					
$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$	$=$	$\begin{bmatrix} 0 & 0 & \cdot \\ 0 & 0 & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$	$+$	$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$	$+$	$\begin{bmatrix} 0 & 2 \\ 4 & 6 \end{bmatrix}$	$+$	$\begin{bmatrix} 0 & 3 \\ 6 & 9 \end{bmatrix}$	$=$	$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 4 & 6 \\ 4 & 12 & 9 \end{bmatrix}$



Deconv in last two layers.  
Other layers use resize-convolution.  
*Artifacts of frequency 2 and 4.*

Deconv only in last layer.  
Other layers use resize-convolution.  
*Artifacts of frequency 2.*

All layers use resize-convolution.  
*No artifacts.*

---

<sup>21</sup>Credits: <http://d2l.ai/> and <https://distill.pub/2016/deconv-checkerboard/>

## initialization of the 2D convolution

A convolutional neural network (CNN) uses different types of layers:

- Convolution layer
- Activation layer
- Pooling layer
- Fully connected layer

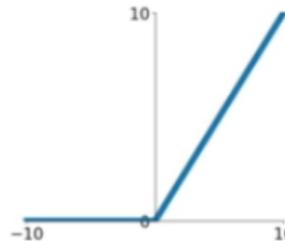
We already saw the Convolution and Fully connected layers.

## Activation function layer

Every activation function (or non-linearity) takes a single number and performs a certain fixed mathematical operation on it. There are several activation functions you may encounter. In practice, the most used is the RELU.

$$f(x) = \max(0, x) \quad (19)$$

## Activation Functions



**ReLU**  
(Rectified Linear Unit)

# Pooling layer

Consider a pooling area of size  $h \times w$  and a 3D input tensor  $x \in \mathbb{R}^{C \times (rh) \times (sw)}$ .

Max-pooling produces a tensor  $o \in \mathbb{R}^{C \times r \times s}$  such that

$$o_{c,j,i} = \max_{n < h, m < w} x[c, j + n, i + m]$$

Average pooling produces a tensor  $o \in \mathbb{R}^{C \times r \times s}$  such that

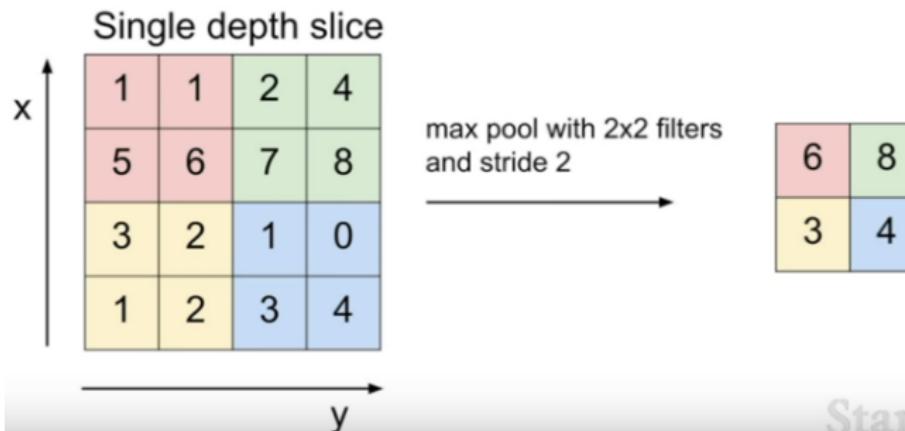
$$o_{c,j,i} = \frac{1}{hw} \sum_{n=0}^{h-1} \sum_{m=0}^{w-1} x[c, j + n, i + m]$$

Pooling is very similar in its formulation to convolution.

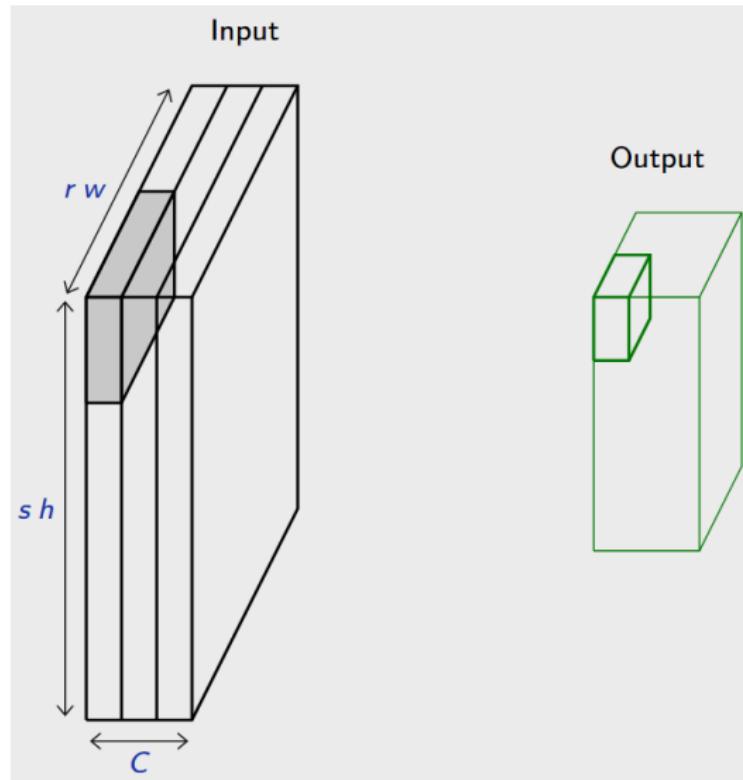
# Pooling layer

A common pooling layer : the max pooling (or the average pooling).  
Max pooling is a discretization process. The goal of the pooling is to concentrate the information in a down-sampled input representation.

## MAX POOLING

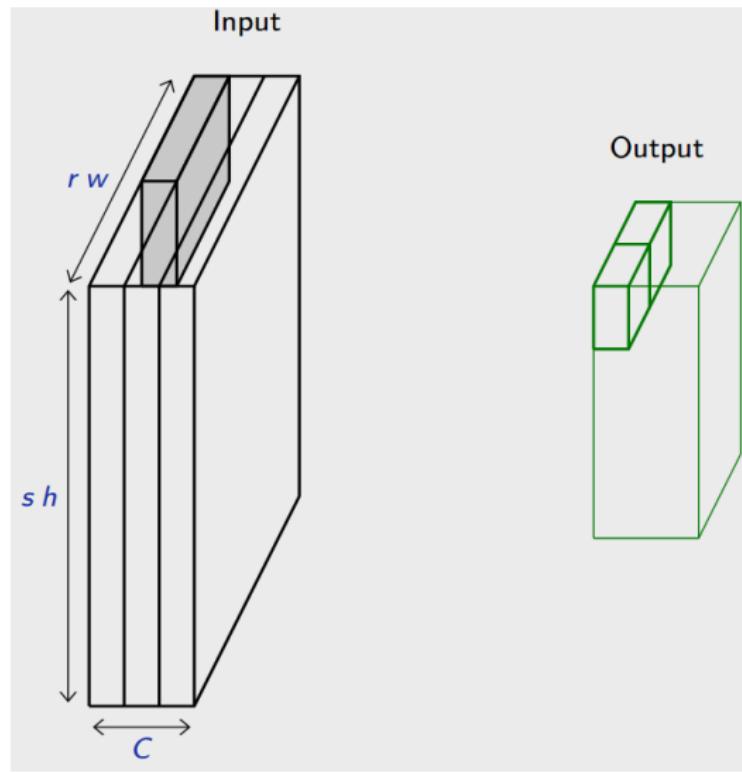


# Example 2D pooling<sup>22</sup>



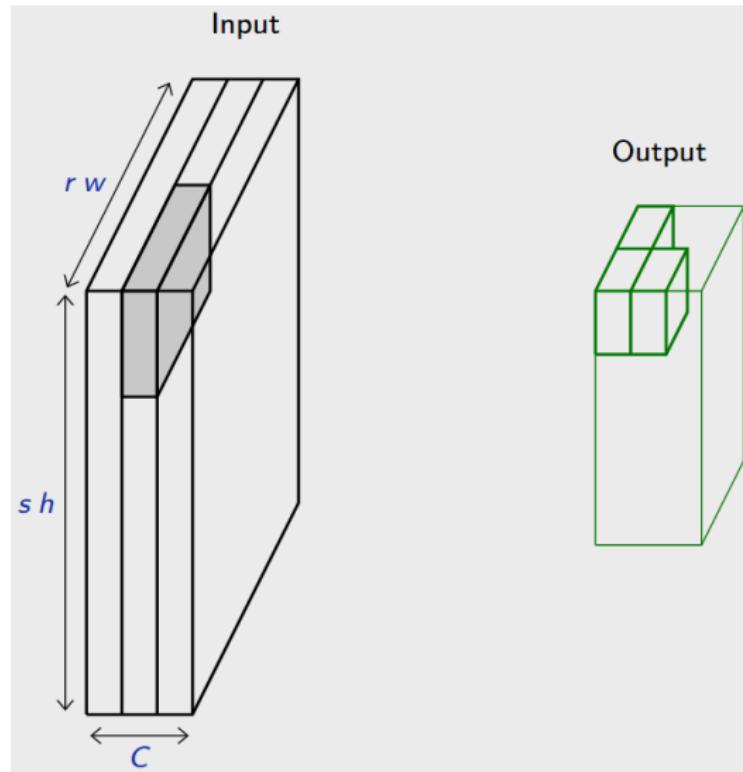
<sup>22</sup>Credits: Francois Fleuret

# Example 2D pooling<sup>23</sup>



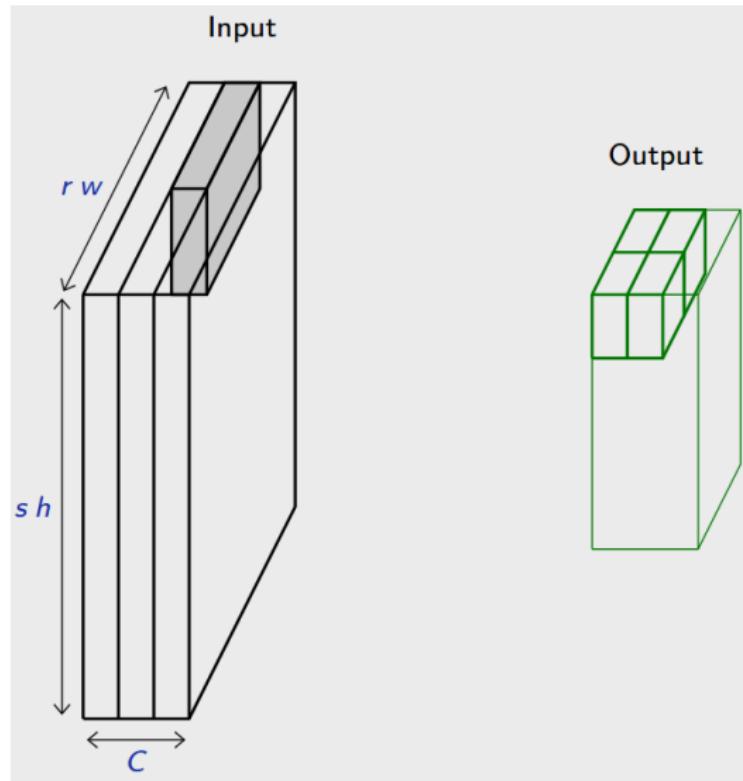
<sup>23</sup>Credits: Francois Fleuret

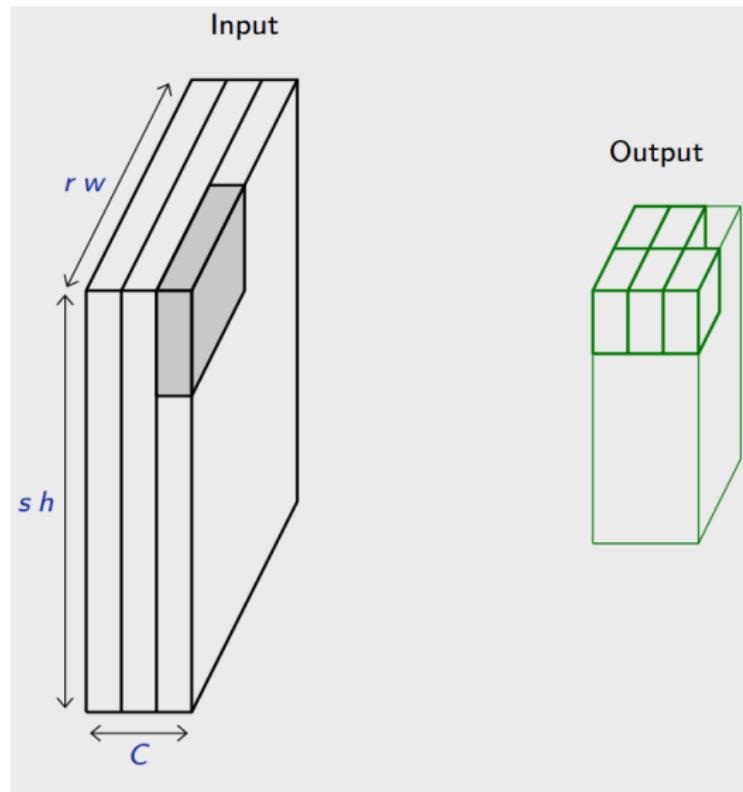
# Example 2D pooling<sup>24</sup>



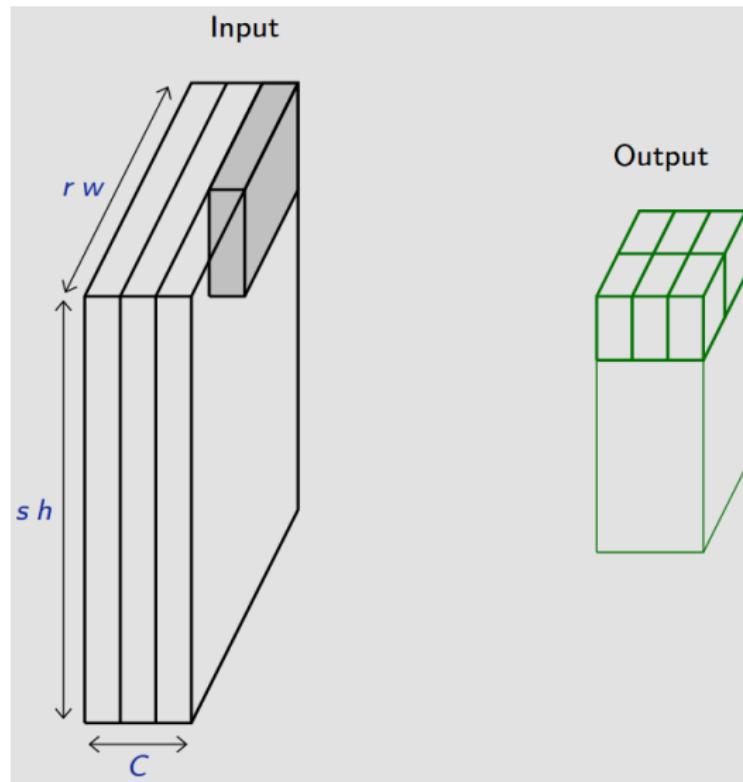
<sup>24</sup>Credits: Francois Fleuret

# Example 2D pooling<sup>25</sup>

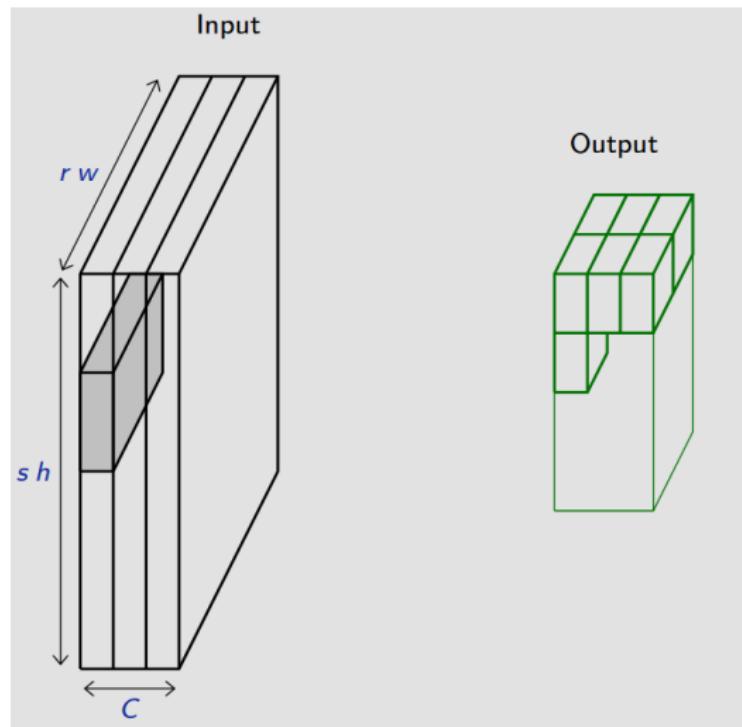


Example 2D pooling<sup>26</sup>

# Example 2D pooling<sup>27</sup>

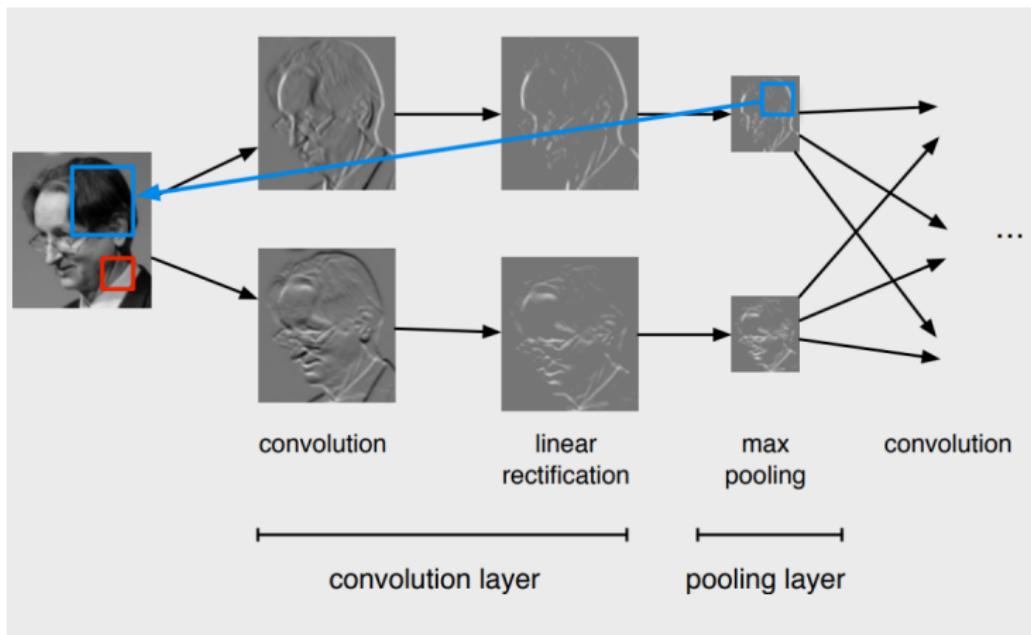


# Example 2D pooling<sup>28</sup>

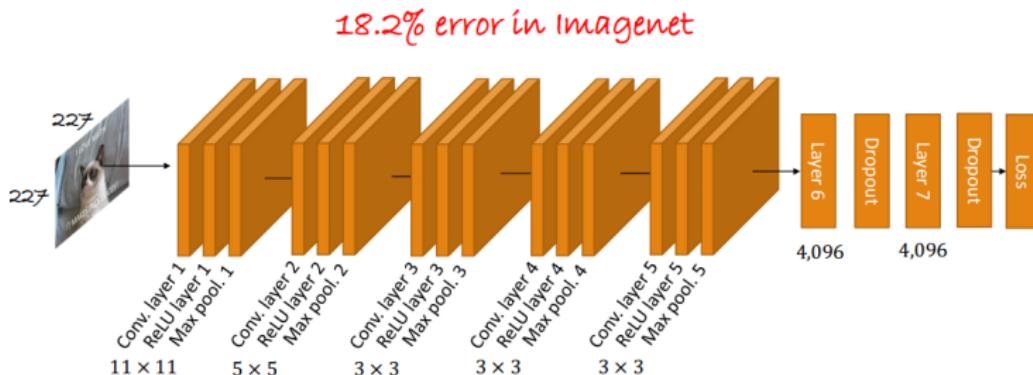


<sup>28</sup>Credits: Francois Fleuret

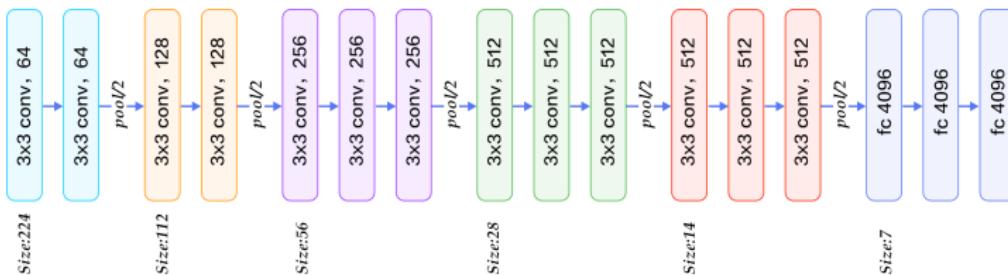
# CNN : architecture



# Example of CNN : AlexNet

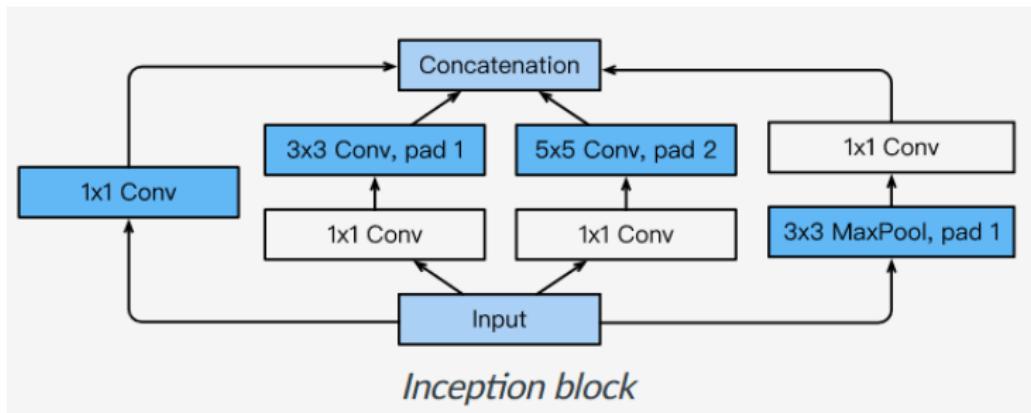


# Example of CNN : VGG

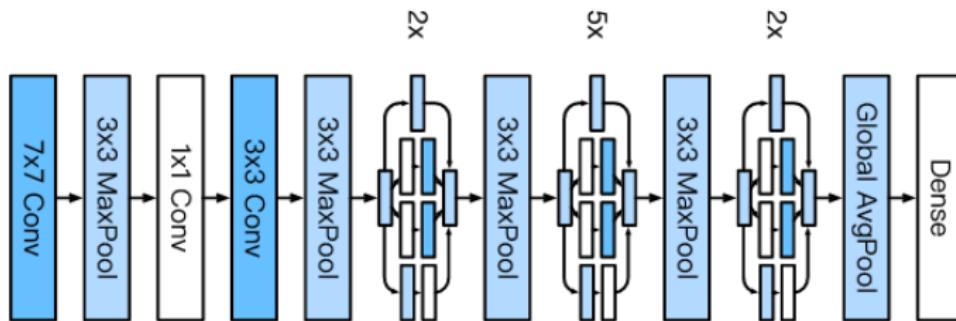


# Example of CNN : GoogLeNet<sup>29</sup>

Each inception block is itself defined as a convolutional network with 4 parallel paths.

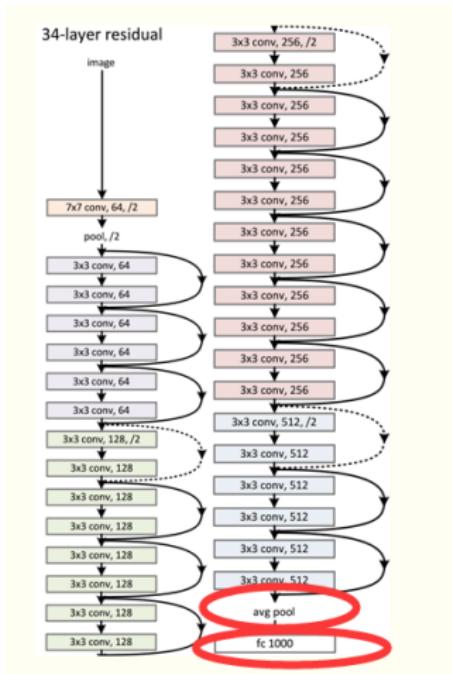


<sup>29</sup>Credits: Dive Into Deep Learning, 2020.

Example of CNN : GoogLeNet<sup>30</sup>

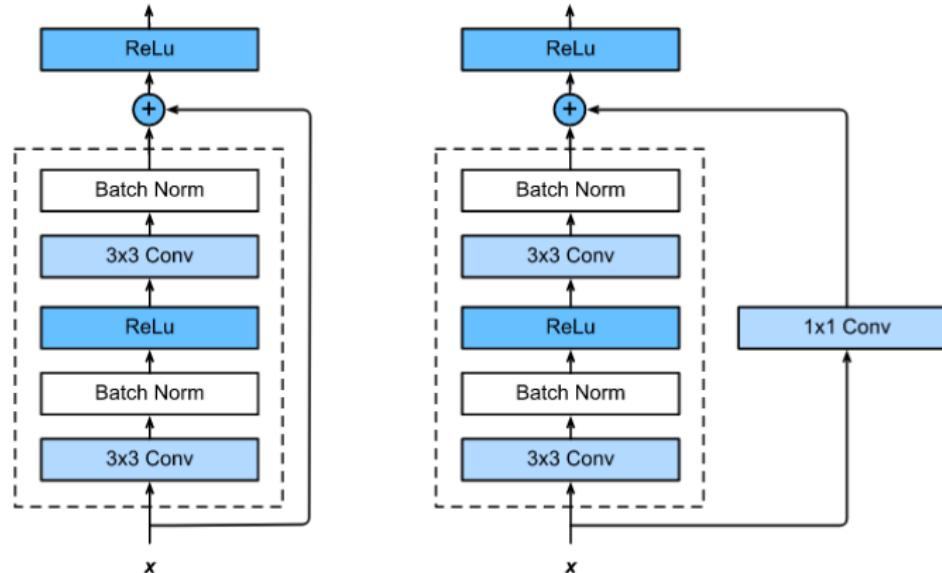
<sup>30</sup>Credits: Dive Into Deep Learning, 2020.

# Example of CNN : resnet 34



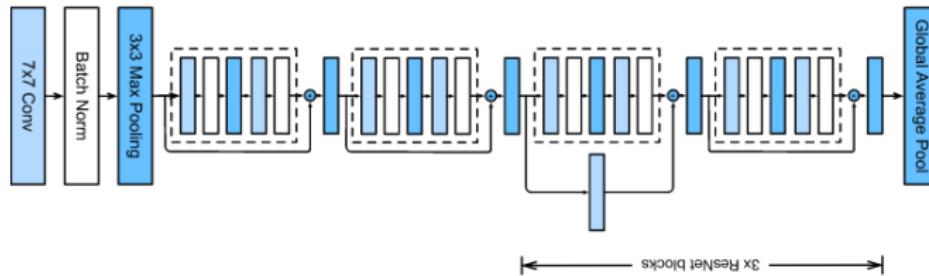
# Example of CNN : resnet<sup>31</sup>

Training networks of this depth is made possible because of the skip connections in the residual blocks. They allow the gradients to shortcut the layers and pass through without vanishing.



<sup>31</sup>Credits: Dive Into Deep Learning, 2020.

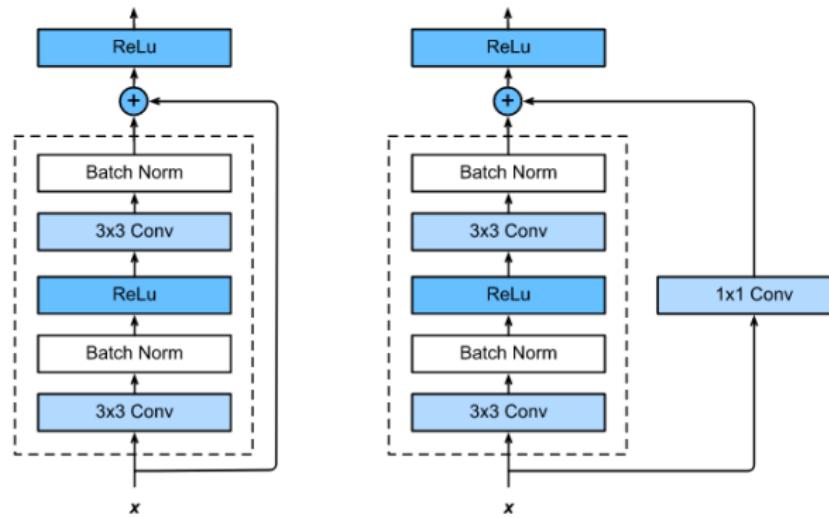
# Example of CNN : resnet<sup>32</sup>



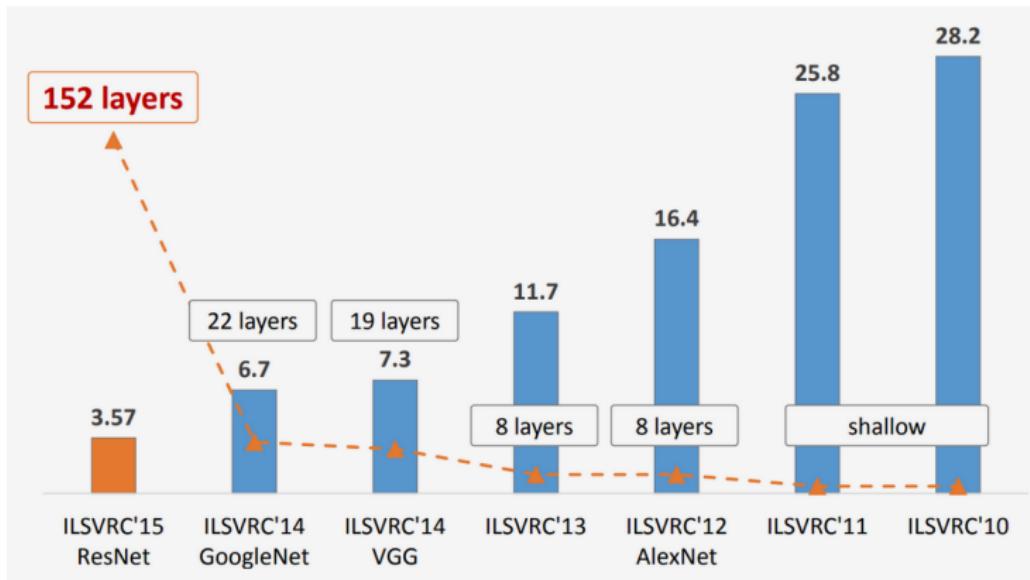
# CNN

Some observations:

- The first layers appear to encode direction and color.
- The direction and color filters get combined into grid and spot textures.
- These textures gradually get combined into increasingly complex patterns.



# Evolution of CNN 33



<sup>33</sup>Credits: Gilles Louppe

## Inside a CNN<sup>34</sup>

AlexNet's first convolutional layer, first 20 filters.

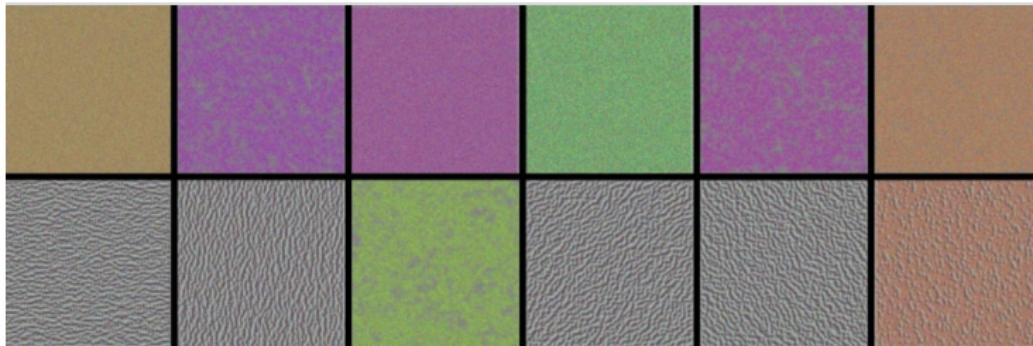


---

<sup>34</sup>Credits: Gilles Louppe

# Inside a CNN <sup>35</sup>

VGG-16, convolutional layer 1-1, a few of the 64 filters

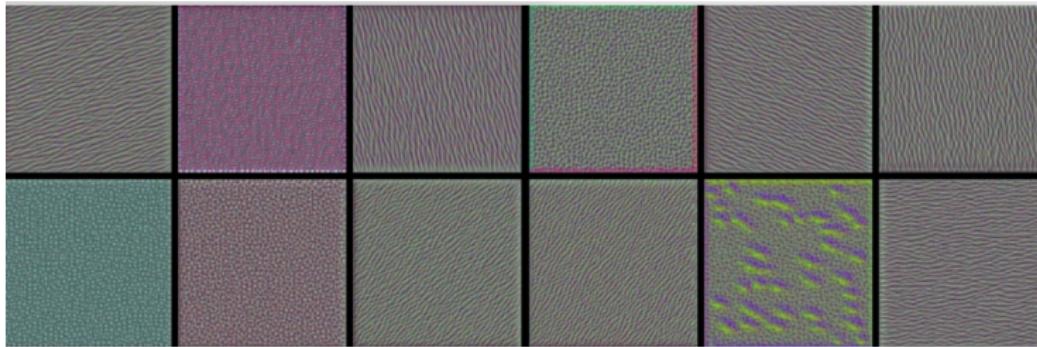


---

<sup>35</sup>Credits: Gilles Louppe

# Inside a CNN<sup>36</sup>

VGG-16, convolutional layer 2-1, a few of the 128 filters

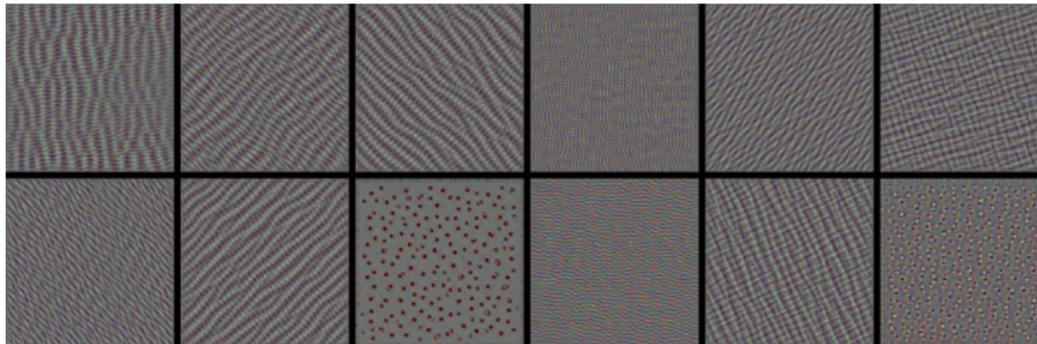


---

<sup>36</sup>Credits: Gilles Louppe

## Inside a CNN <sup>37</sup>

VGG-16, convolutional layer 3-1, a few of the 256 filters

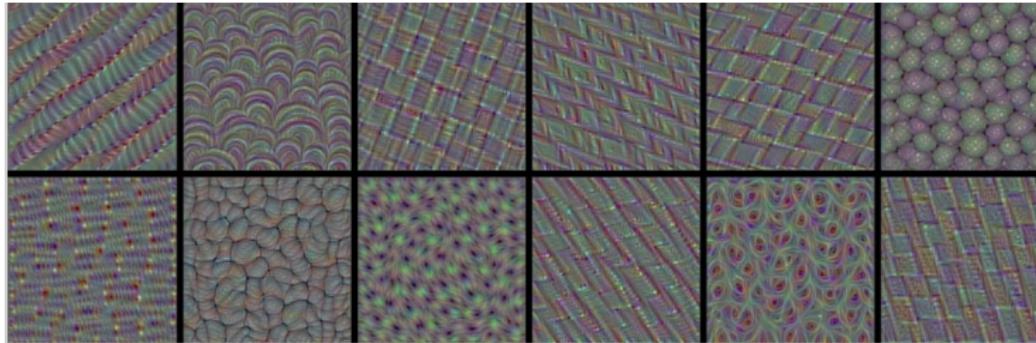


---

<sup>37</sup>Credits: Gilles Louppe

# Inside a CNN<sup>38</sup>

VGG-16, convolutional layer 4-1, a few of the 512 filters

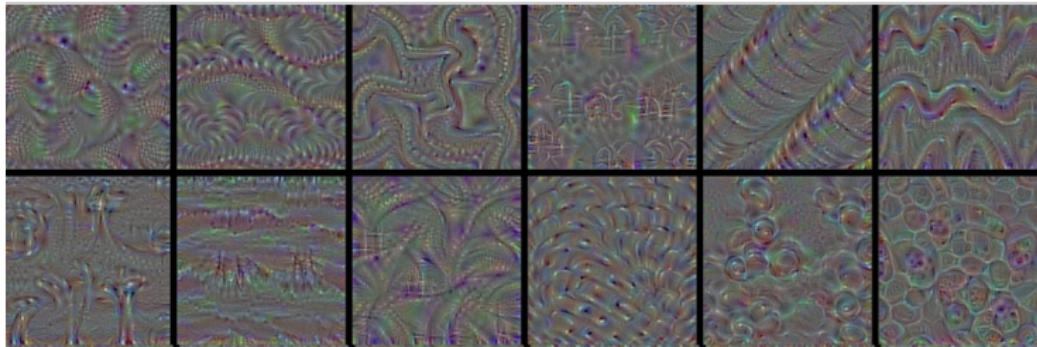


---

<sup>38</sup>Credits: Gilles Louppe

# Inside a CNN <sup>39</sup>

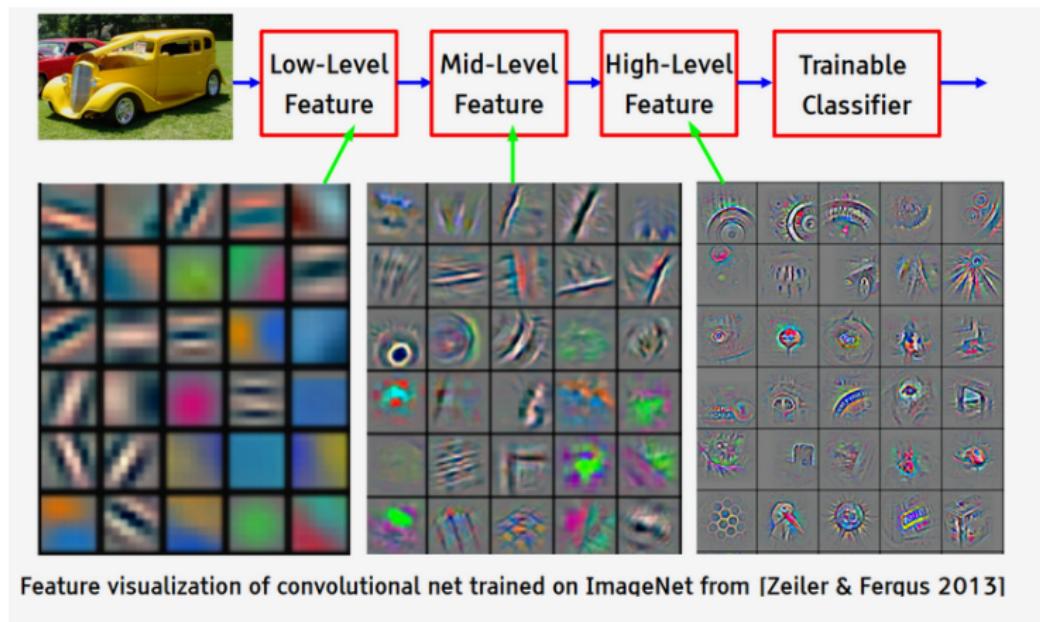
VGG-16, convolutional layer 5-1, a few of the 512 filters



---

<sup>39</sup>Credits: Gilles Louppe

# Inside a CNN<sup>40</sup>



<sup>40</sup>Credits: Gilles Louppe

# Attention layer <sup>41</sup>

Transformer layers were invented for Natural Language Processing. Yet, it is more and more use in computer vision.



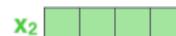
<sup>41</sup>Credits: Jay Alammar

# Attention layer 42

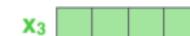
First, you need to represent each word by a representation. There are nice tools to do that. You can use the word2vec embedding.



Je



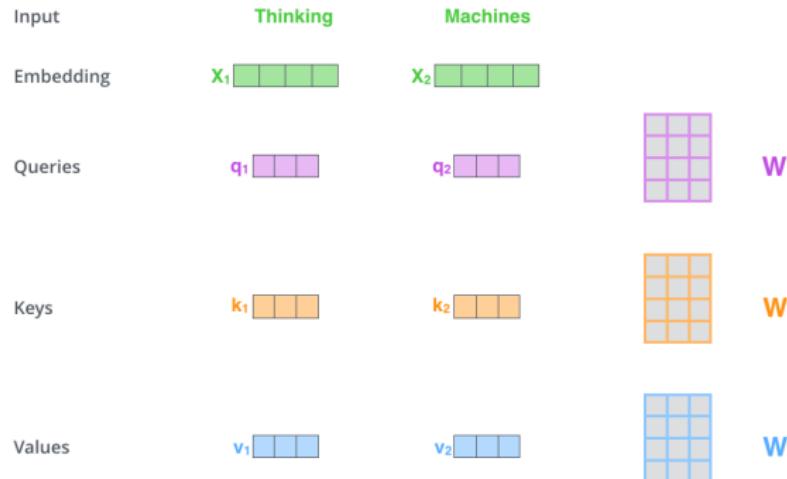
suis



étudiant

# Attention layer 43

The core component in the transformer architecture is the attention layer, or called attention for simplicity. An input of the attention layer is called a **query**. For a query, the attention layer returns the output based on its memory, which is a set of **key-value** pairs.



## Attention layer<sup>44</sup>

Let us consider that we have a **querry**  $q$ , a set of **keys**  $\{k_i\}_i$ , and a set of **values**  $\{v_i\}_i$ . To compute the output, we first assume there is a score function  $\alpha$  which measure the similarity between the query and a key. Then we compute all  $n$  scores  $a_1, \dots, a_n$  defined by

$$a_i = \alpha(q, k_i).$$

Next we use softmax to obtain the attention weights

$$b_1, \dots, b_n = \text{softmax}(a_1, \dots, a_n).$$

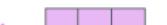
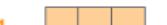
The final output is a weighted sum of the values

$$o = \sum_i b_i v_i.$$

---

<sup>44</sup>Credits: d2l.ai

## Transformer layer<sup>45</sup>

Input	Thinking	Machines
Embedding	$x_1$ 	$x_2$ 
Queries	$q_1$ 	$q_2$ 
Keys	$k_1$ 	$k_2$ 
Values	$v_1$ 	$v_2$ 
Score	$q_1 \cdot k_1 = 112$	$q_1 \cdot k_2 = 96$
Divide by 8 ( $\sqrt{d_k}$ )	14	12
Softmax	0.88	0.12

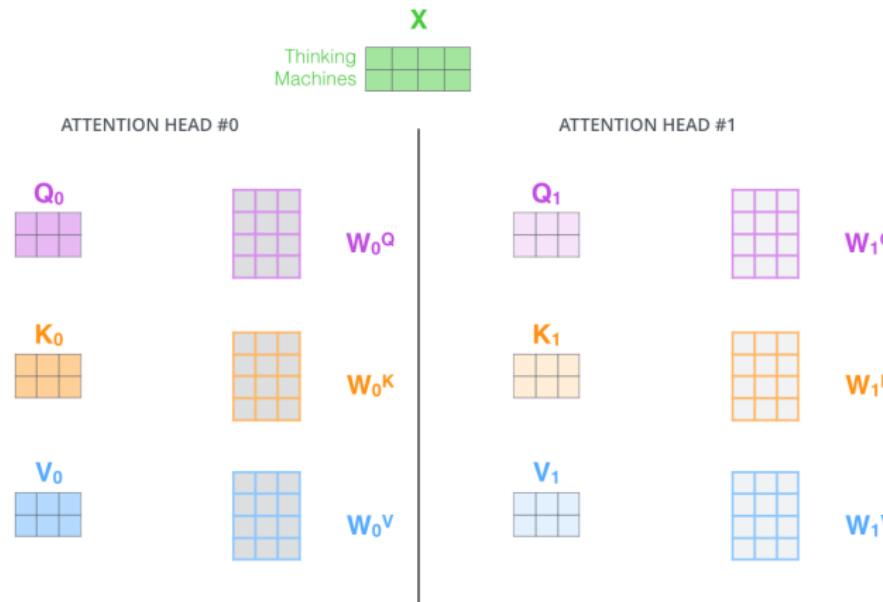
<sup>45</sup>Credits: Jay Alammar

# Attention layer 46

$$\begin{array}{c} \text{X} \quad \quad \quad \text{W}^Q \quad \quad \quad \text{Q} \\ \text{X} \times \text{W}^Q = \text{Q} \\ \text{X} \quad \quad \quad \text{W}^K \quad \quad \quad \text{K} \\ \text{X} \times \text{W}^K = \text{K} \\ \text{X} \quad \quad \quad \text{W}^V \quad \quad \quad \text{V} \\ \text{X} \times \text{W}^V = \text{V} \\ \text{Q} \quad \quad \quad \text{K}^T \quad \quad \quad \text{V} \\ \text{softmax} \left( \frac{\text{Q} \times \text{K}^T}{\sqrt{d_k}} \right) \text{V} \\ \text{Z} \\ = \quad \text{Z} \end{array}$$

# Attention layer 47

In NLP we do not apply just one attention layer, but multiple ones.

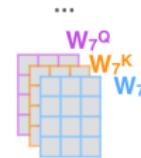
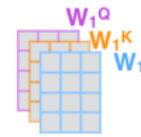
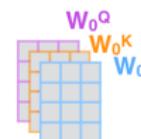


# multi-headed Self-Attention layer <sup>48</sup>

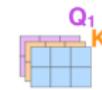
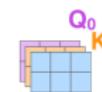
1) This is our input sentence\*  
2) We embed each word\*



3) Split into 8 heads.  
We multiply  $X$  or  $R$  with weight matrices



4) Calculate attention using the resulting  $Q/K/V$  matrices



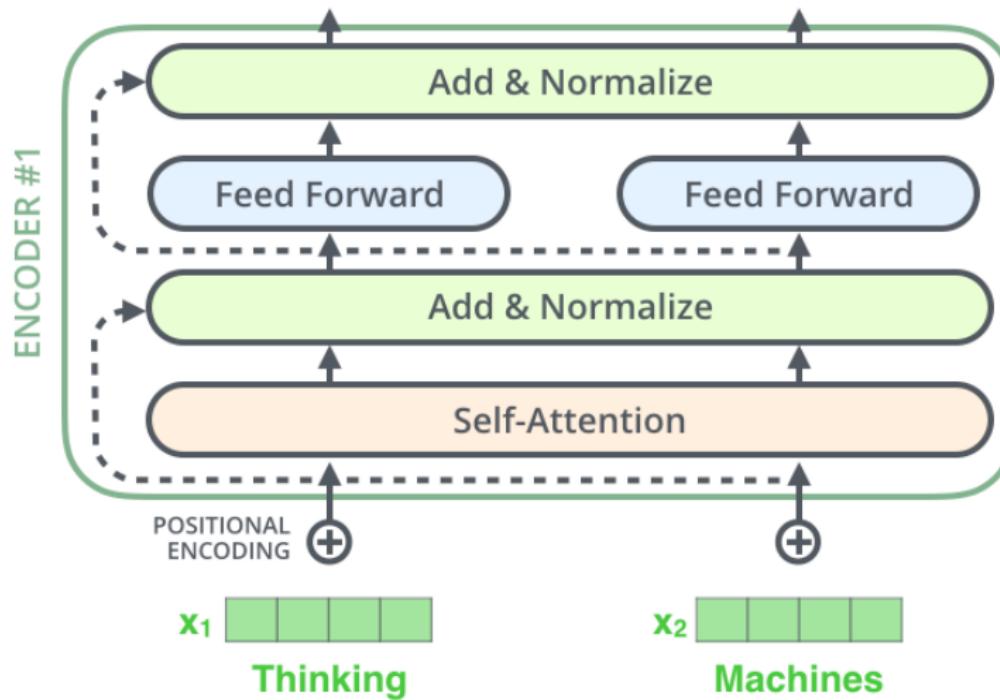
5) Concatenate the resulting  $Z$  matrices, then multiply with weight matrix  $W^O$  to produce the output of the layer



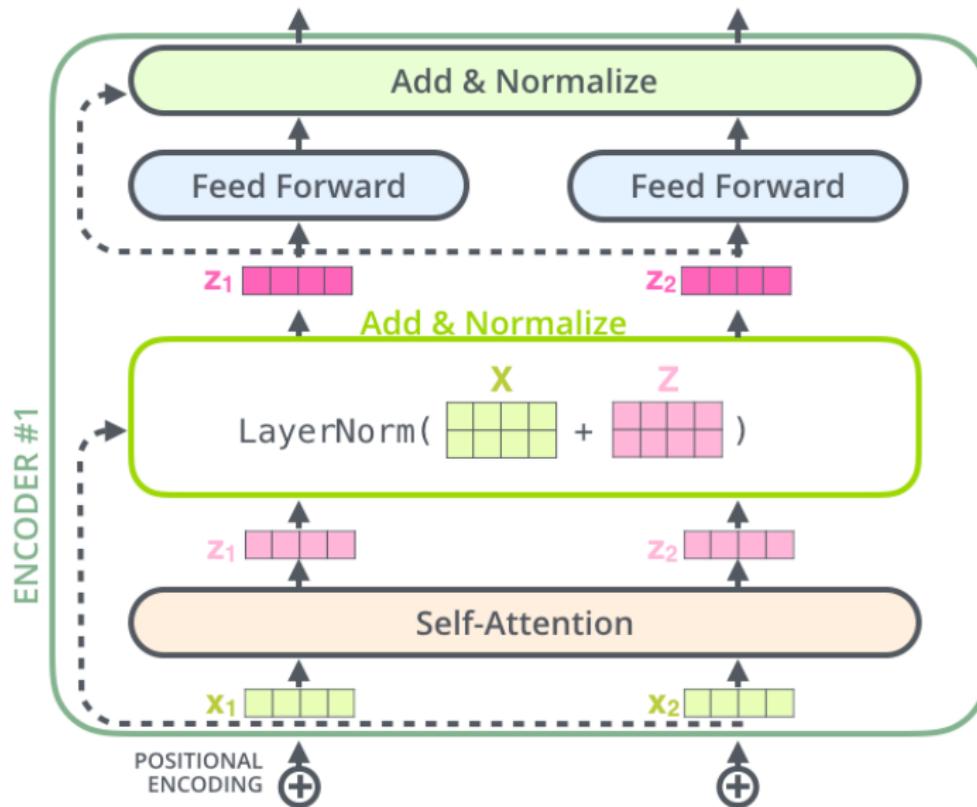
\* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



# multi-headed Self-Attention layer<sup>49</sup>



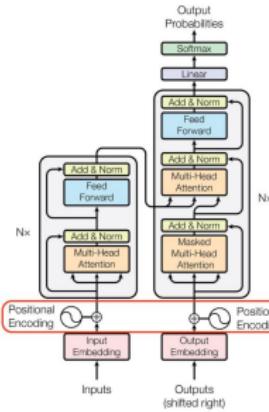
<sup>49</sup>Credits: Jay Alammar

multi-headed Self-Attention layer<sup>50</sup>

# multi-headed Self-Attention layer

One problem is that the position and the order of words is essential for many languages. It defines the grammar and thus the actual semantics of a sentence.

A solution: use a **positional encoding** which is a piece of information to each word about its position in the sentence.



# Positional encoding

The **first bad idea** is to assign a number to each time-step within the [0, 1] range. 0 = 'the first word' and 1 = 'the last word'.

**Problem:** it will introduce is that you can't figure out how many words are present within a specific range.

A **second bad idea** is to the valued 1 to the first word, 2 to the second, and so on.

**Problems:** the values can get quite large, bringing training issues. Also, our model can face test sentences longer than the ones in training.

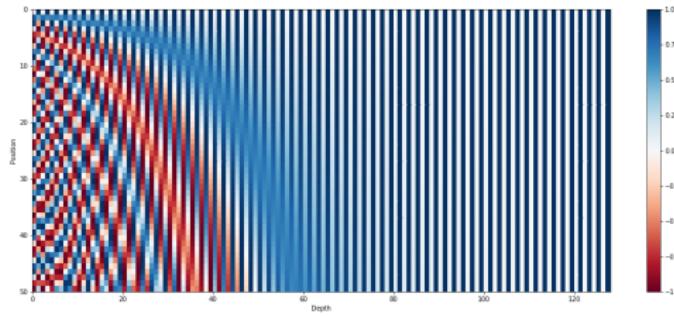
# Positional encoding

Let  $t$  be the desired position in an input sentence. We denote  $\vec{p}_t \in \mathbb{R}^d$  its corresponding encoding, and  $d$  be the encoding dimension (where  $d = 20$ ) Then we have:

$$\vec{p}_t = \begin{bmatrix} \sin(\omega_1 t) \\ \cos(\omega_1 t) \\ \vdots \\ \sin(\omega_{d/2} t) \\ \cos(\omega_{d/2} t) \end{bmatrix}_{d \times 1} \quad (20)$$

where  $\omega_k = \frac{1}{10000^{2k/d}}$

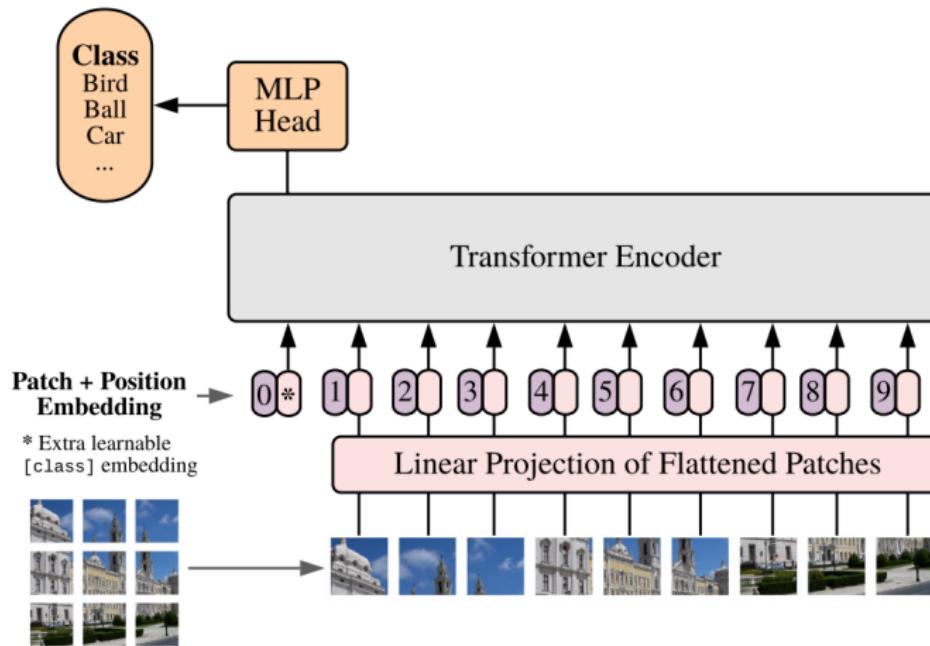
# Positional encoding<sup>51</sup>



Most of the time, the position encoding is summed with the word's embedding to build a new word's representation.

<sup>51</sup>Credits: Amirhossein Kazemnejad

## ViT 52



<sup>52</sup><https://arxiv.org/pdf/2010.11929.pdf>

ViT 53

Please try the tutorial about ViT: <https://colab.research.google.com/drive/1Rc1Y-mEwsPQn04lDGLC1ovdR9DvUs-3K?usp=sharing>

---

<sup>53</sup><https://arxiv.org/pdf/2010.11929.pdf>

- 1 Linear Regression
- 2 Unlearned feature space-Kernel
- 3 Typical recognition Algorithm
- 4 Neural Network
- 5 Convolutional Neural Network
- 6 Transformer architecture
- 7 Training a neural network
- 8 Regularization
- 9 Examples of applications of classical CNN

# Optimization

We have a set of data  $\{x_i, t_i\}_{i=1}^{N_1}$ :

$$\mathcal{F}(\omega) = \frac{\beta}{2} \sum_{i=1}^{N_1} \|f(\omega, x_i) - t_i\|^2. \quad (21)$$

Now  $\omega$  stands for all the weights and biases of the CNN and  $f(\omega, x_i)$  is the result of the CNN with the weights and biases  $\omega$  applied on  $x_i$ . Finding the optimal  $\omega$  that minimizes  $\mathcal{F}$  is complicated. There are different techniques:

- genetic optimization (Neuro evolution, markov chain,...)
- stochastic gradient descent

## Basic of deep learning optimization

Let us start with the previous problem:

$$\min_{\omega} \mathcal{F}(\omega) \text{ , with } \mathcal{F}(\omega) = \sum_{i=1}^{N_1} \|f(\omega, x_i) - t_i\|^2 \quad (22)$$

How can we proceed? A simple algorithm called gradient descent consists in the following, after having checked that  $\mathcal{F}$  is convex ( $\mathcal{F}''(\omega) > 0$ ) and is of class C1.

First we initialize  $\omega_0$ .

Then, at each iteration we calculate:

$$\omega_{t+1} = \omega_t - \lambda \frac{\partial \mathcal{F}}{\partial \omega} \quad (23)$$

$\lambda > 0$  is a parameter that modulates the correction (when  $\lambda$  is too low, slow convergence, when  $\lambda$  is too high, there are oscillations)

# Basic of deep learning optimization

Why does it work?

We remind the derivative of a function:

$$\frac{\partial g}{\partial x} = \lim_{h \rightarrow 0} \frac{g(x + h) - g(x)}{h} \quad (24)$$

For simplicity, we consider for  $h$  really small :

$$\frac{\partial g}{\partial x} \simeq \frac{g(x + h) - g(x)}{h} \quad (25)$$

Now let us consider that  $h = -\lambda \frac{\partial g}{\partial x}$ .

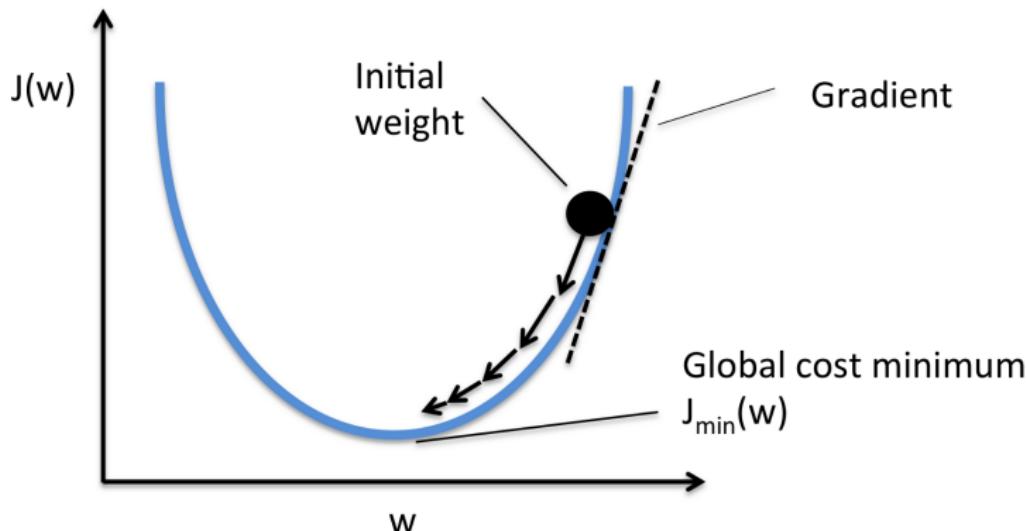
Then have

$$g(x + h) - g(x) \simeq -\lambda \times \left( \frac{\partial g}{\partial x} \right)^2 \quad (26)$$

Since  $\lambda > 0$ , then

$$g(x + h) < g(x) \quad (27)$$

# Basic of deep learning optimization



## Basic of deep learning optimization

Now let us focus on  $\frac{\partial \mathcal{F}}{\partial \omega}$ . This term is

$$\frac{\partial \mathcal{F}}{\partial \omega} = \frac{\partial}{\partial \omega} \sum_{i=1}^{N_1} (f(\omega, x_i) - y_i)^t (f(\omega, x_i) - y_i) \quad (28)$$

$$\frac{\partial \mathcal{F}}{\partial \omega} = \frac{\partial}{\partial \omega} \sum_{i=1}^{N_1} (f(\omega, x_i)^t f(\omega, x_i) - 2y_i^t f(\omega, x_i) + y_i^t y_i) \quad (29)$$

$$\frac{\partial \mathcal{F}}{\partial \omega} = \sum_{i=1}^{N_1} \left( \frac{\partial}{\partial \omega} f(\omega, x_i)^t f(\omega, x_i) - \frac{\partial}{\partial \omega} 2y_i^t f(\omega, x_i) \right) \quad (30)$$

Now let us consider that  $N_1$  is really big (about a billion), this might take ages to sum all the gradients over  $N_1$  and over all the parameters  $w$  and to iterate it one million times.

## Stochastic gradient descent

Now let us focus on  $\frac{\partial \mathcal{F}}{\partial \omega}$ . This term is

$$\frac{\partial \mathcal{F}}{\partial \omega} \simeq \frac{\partial}{\partial \omega} \sum_{i \in B_j} \|f(\omega, x_i) - y_i\|^2 \quad (31)$$

With  $B_j$  a sample of the dataset.

One dataset  $B_j$  might not be representative of the full dataset so we take all the possible  $B_j$

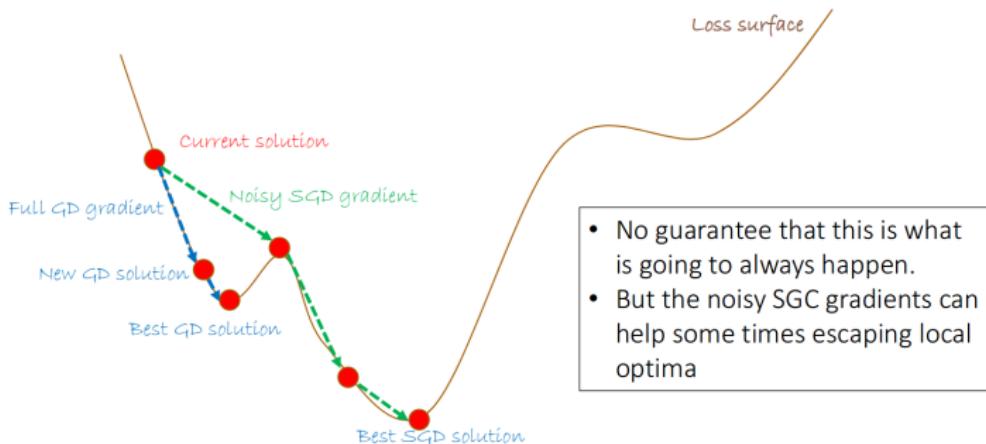
Hence at each iteration we calculate

$$\omega_{t+1} = \omega_t - \lambda \frac{\partial \mathcal{F}_j}{\partial \omega} \quad (32)$$

with

$$\frac{\partial \mathcal{F}_j}{\partial \omega} = \frac{\partial}{\partial \omega} \sum_{i \in B_j} \|f(\omega, x_i) - y_i\|^2 \quad (33)$$

# Stochastic gradient descent



# Stochastic gradient descent algorithm

## The stochastic gradient descent

First, we initialized the parameters  $\omega_0$ .

Then, at each iteration we calculate

$$\omega_{t+1} = \omega_t - \lambda \frac{\partial \mathcal{F}_j}{\partial \omega} \quad (34)$$

## The stochastic gradient descent with momentum

First, we initialized the parameters  $\omega_0$ .

Then, at each iteration we calculate

$$u_{t+1} = \gamma u_t + \lambda \frac{\partial \mathcal{F}_j}{\partial \omega} \quad (35)$$

$$\omega_{t+1} = \omega_t - u_{t+1} \quad (36)$$

the term  $u_{t+1}$  allow us to stabilize the gradient descent.  $\gamma \geq 0$  is the momentum parameter. This parameter add inertia in the choice of the step direction.

## Stochastic gradient descent

Stochastic Gradient Descent (**SGD**) consists in visiting the samples in mini-batches and updating the parameters each time like a classical Gradient Descent. There are two parameters : the size of the batch  $B$  and the learning rate  $\lambda$ .

Increasing the batch size  $B$  reduces the variance of the gradient estimates and enables the speed-up of batch processing. The interplay between  $B$  and  $\lambda$  is still a bit unclear.

# The tradeoffs of large-scale learning

A fundamental result due to Bottou and Bousquet (2011)<sup>54</sup> states that stochastic optimization algorithms (e.g., SGD) yield the best **generalization** performance (in terms of excess error) when compared to GD and 2GD despite having the worst optimization performance on the **empirical cost**.

---

<sup>54</sup> Bottou, LÃ©on, and Olivier Bousquet. "13 the tradeoffs of large-scale learning." Optimization for machine learning (2011): 351.

## Stochastic gradient descent with momentum

An improvement to Stochastic gradient descent is to use momentum to add inertia in the choice of the step direction.

### The stochastic gradient descent with momentum

First, we initialized the parameters  $\omega_0$ .

Then, at each iteration we calculate

$$u_{t+1} = \gamma u_t + \lambda \frac{\partial \mathcal{F}_j}{\partial \omega} \quad (37)$$

$$\omega_{t+1} = \omega_t - u_{t+1} \quad (38)$$

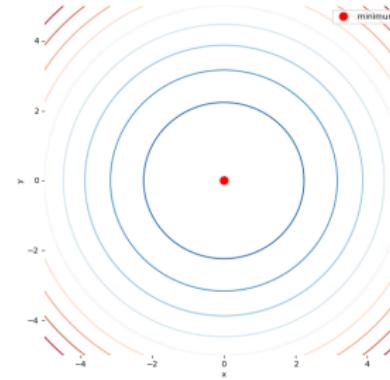
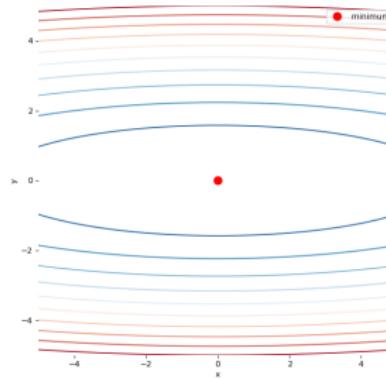
the term  $u_{t+1}$  allow us to stabilize the gradient descent.  $\gamma \geq 0$  is the momentum parameter. This parameter add inertia in the choice of the step direction.

Gradient descent with momentum has three nice properties:

- it can go through local barriers,
- it accelerates if the gradient does not change much,
- it reduces oscillations in narrow valleys.

# Adaptive learning rate<sup>55</sup>

One problem is that SGD and SGD with momentum relieve too much on the learning rate and assume the isotropy of the loss of the DNN.



A solution use second order gradient descent. **But it is too heavy.**

<sup>55</sup>Credits: Gilles Louppe

# AdaGrad

## The AdaGrad algorithm

$$r_t = r_{t-1} + \left( \frac{\partial \mathcal{F}_j}{\partial \omega} \right)^2 \quad (39)$$

$$\omega_{t+1} = \omega_t - \frac{\lambda}{\sqrt{\hat{r}_t} + \epsilon} \hat{m}_{t+1} \quad (40)$$

AdaGrad eliminates the need to manually tune the learning rate. Most implementation use  $\lambda = 0.01$  as default. Attention  $r_t$  is an increasing sequence.

# RMSProp

Same as AdaGrad but accumulate an exponentially decaying average of the gradient.

## The AdaGrad algorithm

$$r_t = \rho r_{t-1} + (1 - \rho) \left( \frac{\partial \mathcal{F}_j}{\partial \omega} \right)^2 \quad (41)$$

$$\omega_{t+1} = \omega_t - \frac{\lambda}{\sqrt{\hat{r}_t} + \epsilon} \hat{m}_{t+1} \quad (42)$$

## Adam algorithm

The Adam algorithm uses moving averages of each coordinate. The update rule is:

### The Adam algorithm

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) \frac{\partial \mathcal{F}_j}{\partial \omega} \quad (43)$$

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \beta_1} \quad (44)$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) \left( \frac{\partial \mathcal{F}_j}{\partial \omega} \right)^2 \quad (45)$$

$$\hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta_2} \quad (46)$$

$$\omega_{t+1} = \omega_t - \frac{\lambda}{\sqrt{\hat{v}_{t+1}} + \epsilon} \hat{m}_{t+1} \quad (47)$$

This is a mix with momentum and having a special learning rate for each parameter  $w$ . There are 3 parameters:  $\lambda, \beta_1, \beta_2$ .

## Chain rule

The chain rule states that  $(f \circ g)' = (f' \circ g)g'$ . Let us have a look at functions of two variables.

- let  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  be a differentiable function,
- let  $g : \mathbb{R}^p \rightarrow \mathbb{R}^n$  be a differentiable function,
- let  $h = (f \circ g)$  be a differentiable function,

$h$  is differentiable and  $h' = (f' \circ g)g'$

$$h' = \begin{pmatrix} \frac{\partial h}{\partial x_1} & \frac{\partial h}{\partial x_2} & \cdots & \frac{\partial h}{\partial x_p} \end{pmatrix}$$

# Chain rule

$h$  is differentiable and  $h' = (f' \circ g)g'$

$$h' = \begin{pmatrix} \frac{\partial h}{\partial x_1} & \frac{\partial h}{\partial x_2} & \cdots & \frac{\partial h}{\partial x_p} \end{pmatrix}$$

$$g' = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_p} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_p} \\ \vdots & \cdots & \cdots & \vdots \\ \frac{\partial g_n}{\partial x_1} & \frac{\partial g_n}{\partial x_2} & \cdots & \frac{\partial g_n}{\partial x_p} \end{pmatrix}$$

$$f'(g) = \begin{pmatrix} \frac{\partial f}{\partial g_1} & \frac{\partial h}{\partial g_2} & \cdots & \frac{\partial f}{\partial g_n} \end{pmatrix}$$

## Chain rule

$h$  is differentiable and  $h' = (f' \circ g)g'$

$$h' = \begin{pmatrix} \frac{\partial h}{\partial x_1} & \frac{\partial h}{\partial x_2} & \cdots & \frac{\partial h}{\partial x_p} \end{pmatrix}$$

$$h' = \begin{pmatrix} \frac{\partial f}{\partial g_1} & \frac{\partial h}{\partial g_2} & \cdots & \frac{\partial f}{\partial g_n} \end{pmatrix} \times \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \cdots & \frac{\partial g_1}{\partial x_p} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \cdots & \frac{\partial g_2}{\partial x_p} \\ \vdots & \ddots & \cdots & \vdots \\ \frac{\partial g_n}{\partial x_1} & \frac{\partial g_n}{\partial x_2} & \cdots & \frac{\partial g_n}{\partial x_p} \end{pmatrix}$$

Hence, the chain rule results is:

$$\frac{\partial h}{\partial x_i} = \sum_{k=1}^n \underbrace{\frac{\partial f}{\partial g_k}}_{\text{recursive case}} \frac{\partial g_k}{\partial x_i}$$

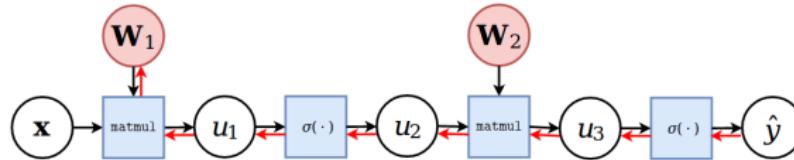
## Chain rule

Let us consider a simplified 2-layer MLP and the following loss function:

$$f(x; W_1, W_2) = \sigma(W_2^T \sigma(W_1^T x))$$

$$\ell(y, \hat{y}; W_1, W_2) = \text{cross\_ent}(y, \hat{y})$$

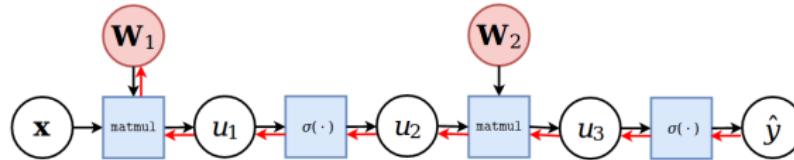
# Chain rule<sup>56</sup>



Let us zoom in on the computation of the network output  $\hat{y}$  and of its derivative with respect to  $W_1$ .

<sup>56</sup>Credits: Gilles Louppe

# Chain rule<sup>57</sup>

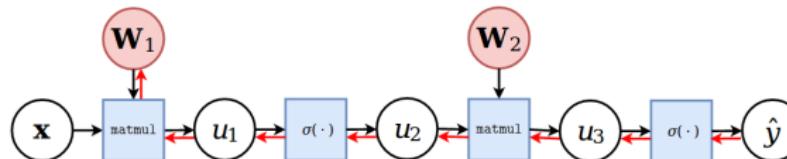


Forward pass: values  $u_1$ ,  $u_2$ ,  $u_3$  and  $\hat{y}$  are computed by traversing the graph from inputs to outputs given  $x$ ,  $W_1$  and  $W_2$ .

<sup>57</sup>Credits: Gilles Louppe

## Chain rule<sup>58</sup>

For simplicity let us consider that  $W_1$ ,  $W_2$ ,  $x$  and  $\hat{y}$  are scalar.  
We replace  $W_1$ ,  $W_2$  by  $w_1$  and  $w_2$ .



Backward pass: by the chain rule we have

$$\begin{aligned}\frac{\partial \hat{y}}{\partial w_1} &= \frac{\partial \hat{y}}{\partial u_3} \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial w_1} \\ &= \frac{\partial \sigma(u_3)}{\partial u_3} \frac{\partial w_2 \cdot u_2}{\partial u_2} \frac{\partial \sigma(u_1)}{\partial u_1} \frac{\partial w_1 \cdot x}{\partial w_1}\end{aligned}$$

<sup>58</sup>Credits: Gilles Louppe

# Chain rule<sup>59</sup>

Let us develop the chain rule of  $f(x; w_1, w_2, w_3) = \sigma(w_3\sigma(w_2\sigma(w_1x)))$ .

Let us rewrite the intermediate functions

$$u_1 = w_1x$$

$$u_2 = \sigma(u_1)$$

$$u_3 = w_2u_2$$

$$u_4 = \sigma(u_3)$$

$$u_5 = w_3u_4$$

$$\hat{y} = \sigma(u_5)$$

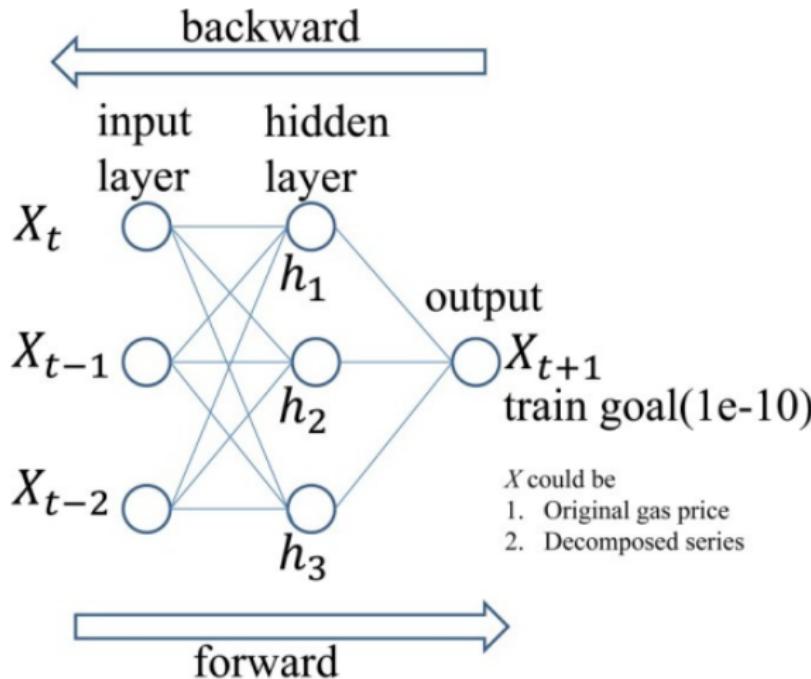
Now, we can write  $\frac{\partial \hat{y}}{\partial w_1}$  as :

$$\begin{aligned}\frac{\partial \hat{y}}{\partial w_1} &= \frac{\partial \hat{y}}{\partial u_5} \frac{\partial u_5}{\partial u_4} \frac{\partial u_4}{\partial u_3} \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial w_1} \\ &= \frac{\partial \sigma(u_5)}{\partial u_5} w_3 \frac{\partial \sigma(u_3)}{\partial u_3} w_2 \frac{\partial \sigma(u_1)}{\partial u_1} x\end{aligned}$$

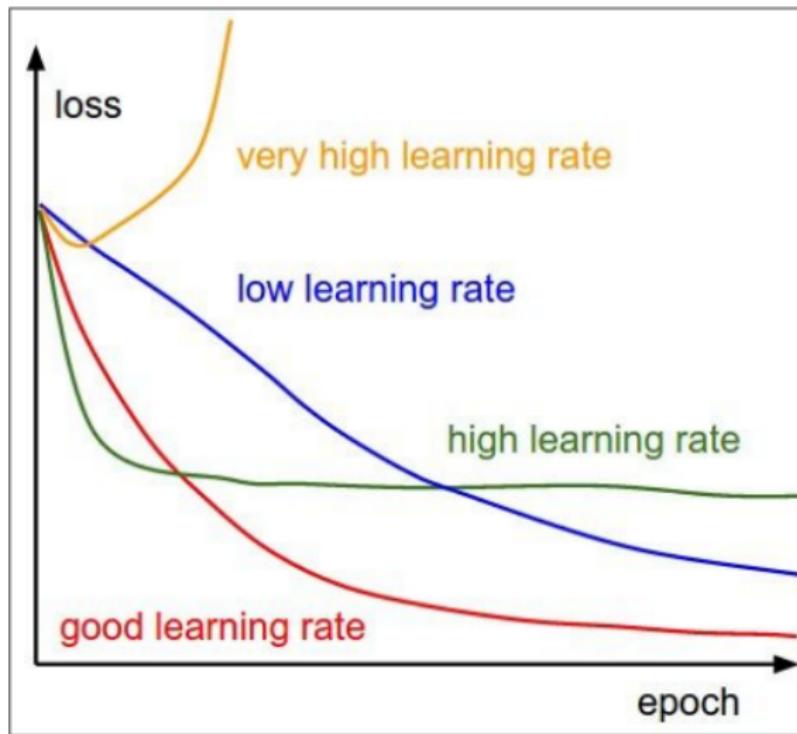
---

<sup>59</sup>Credits: Gilles Louppe

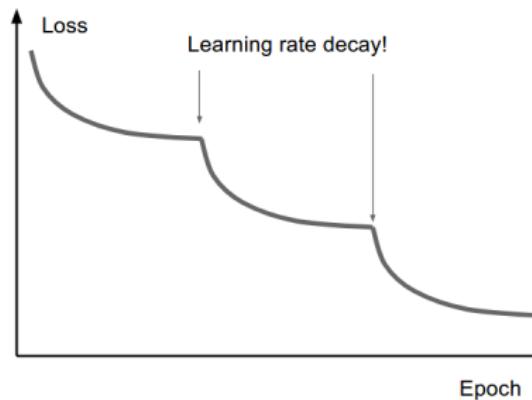
# Forward/backward



Which one of these learning rates is best to use?



Which one of these learning rates is best to use?



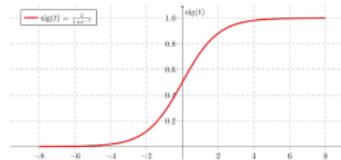
**Solution :** Learning rate decay over time.

- step decay: a decay learning rate by half every few epochs.
- exponential decay:  $\lambda(t) = \lambda_0 \times e^{-kt}$
- $1/t$  decay:  $\lambda(t) = \lambda_0 / (1 + kt)$

# Vanishing gradients

Now let us have a look at the sigmoid function :

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}.$$

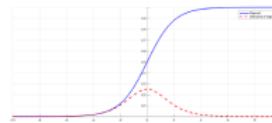


Can you evaluate the derivative?

# Vanishing gradients

Now let us have a look at the sigmoid function :

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}.$$



Can you evaluate the derivative?

$$\sigma(x)' = \sigma(x)(1 - \sigma(x)).$$

# Vanishing gradients

Now let assume that the weights are initialized randomly from a Gaussian with zero-mean and small variance, such that  $w_i \in [-1, 1]$  for  $i \in 1, 2, 3$ . Then we have:

$$\frac{d\hat{y}}{dw_1} = \underbrace{\frac{\partial \sigma(u_5)}{\partial u_5}}_{\leq 1/4} \underbrace{w_3}_{\leq 1} \underbrace{\frac{\partial \sigma(u_3)}{\partial u_3}}_{\leq 1/4} \underbrace{w_2}_{\leq 1} \underbrace{\frac{\partial \sigma(u_1)}{\partial u_1}}_{\leq 1/4} x$$

This implies that the gradient  $\frac{d\hat{y}}{dw_1}$  shrinks . A solution use Relu, then fore,

$$\frac{d\hat{y}}{dw_1} = \underbrace{\frac{\partial \sigma(u_5)}{\partial u_5}}_{=1} w_3 \underbrace{\frac{\partial \sigma(u_3)}{\partial u_3}}_{=1} w_2 \underbrace{\frac{\partial \sigma(u_1)}{\partial u_1}}_{=1} x$$

# initialization of neural networks

In convex problems, provided a good learning rate  $\gamma$ , convergence is guaranteed regardless of the initial parameter values. In the non-convex regime, initialization is more important!

## initialization of neural networks

A lot of weights have to be initialized. What value can we put? The same value for all the convolution layer is a bad idea because of the weight sharing.

The solution is to use a random initialization, not too small and not too big.

Xavier<sup>60</sup> initialisation and He<sup>61</sup> are the most used in practice since the weights depend on the size of the output/input. They have good properties.

---

<sup>60</sup>Xavier Glorot and Yoshua Bengio (2010): Understanding the difficulty of training deep feedforward neural networks. International conference on artificial intelligence and statistics.

<sup>61</sup>Kaiming He, et al (2015): Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

# He initialization

Let us consider a deep neural network modelled by:

$$g_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D_{\text{in}}} \omega_{k,j}^{(1)} x_{i,j} \quad \forall k \in [1, M_2]$$

$$a_k^{(1)} = a(g_k^{(1)}) \quad \forall k \in [1, M_2]$$

$a()$  is a Rectified Linear Unit (ReLU) function:

$$a(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

Then we have:

$$g_{k1}^{(2)} = b_{k1}^{(2)} + \sum_{k=1}^{M_2} \omega_{k1,k}^{(2)} \cdot a_k^{(1)} \quad \forall k1 \in [1, M_3]$$

$$a_{k1}^{(2)} = a(g_{k1}^{(2)}) \quad \forall k1 \in [1, M_3]$$

# He initialization

$$g(x_i, \omega)_{k2} = b_{k2}^{(3)} + \sum_{k1=1}^{M_3} \omega_{k2,k1}^{(3)} \cdot a_{k1}^{(2)} \quad \forall k2 \in [1, D_{\text{out}}]$$

These equations are can be synthesize:

$$g(x_i, \omega)_{k2} = b_{k2}^{(3)} + \sum_{k1=1}^{M_3} \omega_{k2,k1}^{(3)} \cdot a_{k1}^{(2)} \left( b_{k1}^{(2)} + \sum_{k=1}^{M_2} \omega_{k1,k}^{(2)} \cdot a_k^{(1)} \left( b_k^{(1)} + \sum_{j=1}^{D_{\text{in}}} \omega_{k,j}^{(1)} x_{i,j} \right) \right)$$

with  $k2 \in [1, D_{\text{out}}]$ .

$g(x_i, \omega)$  is a vector that belongs to  $\mathbb{R}^{D_{\text{out}}}$ , for now we will just focus on the element  $k2$  of this vector.

The variance of the deep neural network is :

$$\text{var}_W(g(x, W)_{k2}) = \mathbb{E}_W (g^2(x, W)_{k2}) - (\mathbb{E}_W g(x, W)_{k2})^2 \quad (48)$$

# He initialization

By assuming that the elements  $i$  in  $a_i^{(l-1)}$  are also mutually independent and share the same distribution, and that  $a_i^{(l-1)}$  and  $\omega_{i1,i}^{(l)}$ , we have:

$$\text{var}(g(x, W)^{(l)}) = M_l \text{var}(\omega^{(l)} a^{(l-1)}) \quad (49)$$

Using :

- the variance of the product of independent variables
- $\omega^{(l)}$  have zero mean

Then:

$$\text{var}(g(x, W)^{(l)}) = M_l \text{var}(\omega^{(l)}) \mathbb{E}((a^{(l-1)})^2) \quad (50)$$

# He initialization

we use the fact that  $\omega^{(l-1)}$  has a symmetric distribution around zero

So

$$\mathbb{E}((a^{(l-1)})^2) = 1/2\text{var}(g(x, W)^{(l-1)}) \quad (51)$$

Then we have:

$$\text{var}(g(x, W)^{(l)}) = M_l/2\text{var}(\omega^{(l)}) \text{var}(g(x, W)^{(l-1)}) \quad (52)$$

With  $L$  layers put together, we have

$$\text{var}(g(x, W)^{(L)}) = \text{var}(x) \prod_{l=2}^L (M_l/2\text{var}(\omega^{(l)})) \quad (53)$$

# He initialization

A good **initialization** method should avoid **reducing** or **magnifying** the magnitudes of input signals exponentially.

So we want :  $\forall l \in [1, L] \quad M_l / 2\text{var}(\omega^{(l)}) = 1$

$$\forall l \in [1, L] \quad \text{var}(\omega^{(l)}) = \frac{2}{M_l} \text{ and } \mathbb{E}(\omega^{(l)}) = 0 \quad (54)$$

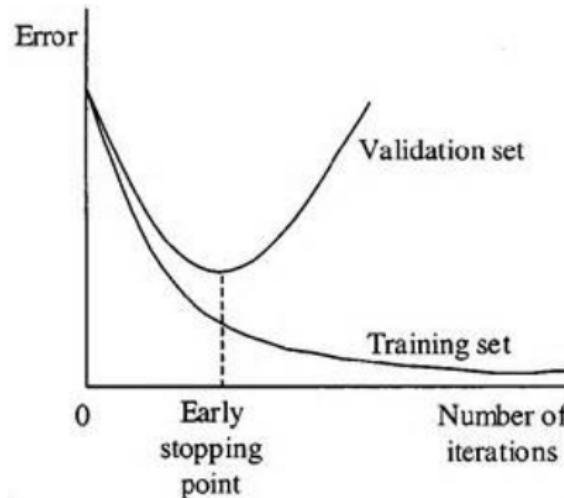
- 1 Linear Regression
- 2 Unlearned feature space-Kernel
- 3 Typical recognition Algorithm
- 4 Neural Network
- 5 Convolutional Neural Network
- 6 Transformer architecture
- 7 Training a neural network
- 8 Regularization
- 9 Examples of applications of classical CNN

# Regularization

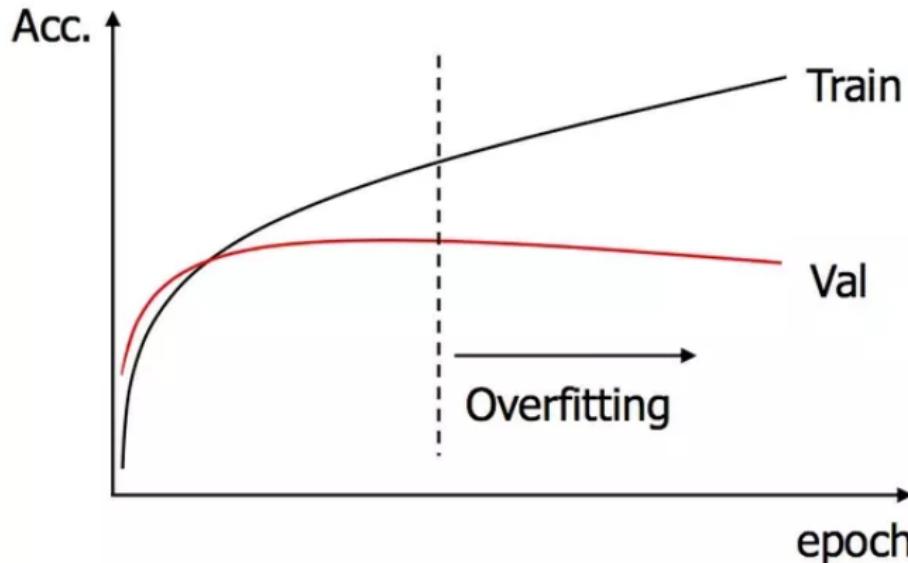
We remind you that you have two sets: a training set  $\{(x_i, t_i)\}_{i=1}^{N_1}$  and the validation set  $\{(x_i, t_i)\}_{i=1}^{N_2}$ .

What is the utility of these two sets?

What can we deduce from these curves?



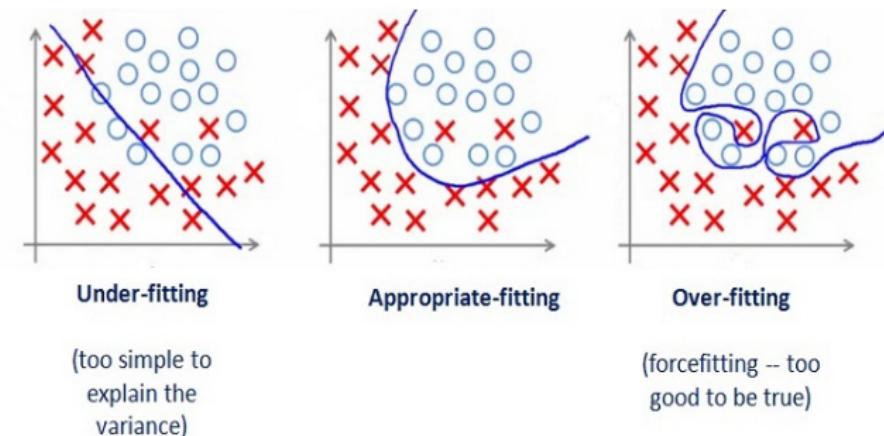
# Regularization



# Regularization

## Overfitting

- Training too much on training set limits generalization
- Important to keep an eye on validation error
- Stop learning if validation error increase



## Solution : regularization

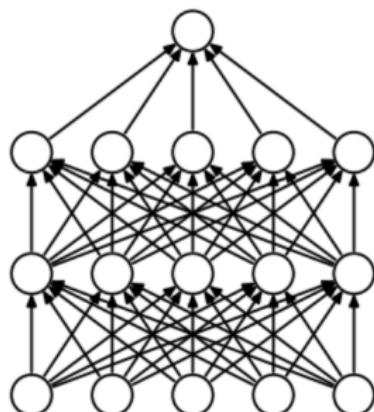
You can use weight decay :

$$\mathcal{L}(\omega) = \mathcal{F}_{\text{data}}(\omega) + \frac{\lambda_2}{2} \|\omega\|^2 \quad (55)$$

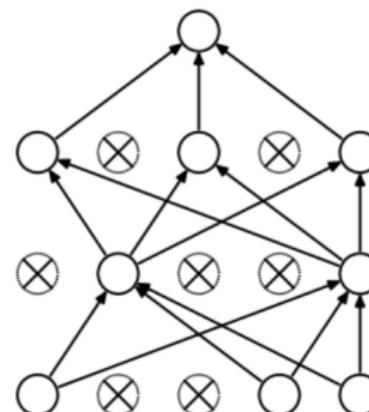
Then during the gradient descent we have

$$\frac{\partial \mathcal{F}}{\partial w}(\omega) = \frac{\partial \mathcal{F}_{\text{data}}}{\partial w}(\omega) + \lambda_2 w \quad (56)$$

# Solution: regularization with dropout



(a) Standard Neural Net



(b) After applying dropout.

# Solution: regularization with Batch Normalization

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;

Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

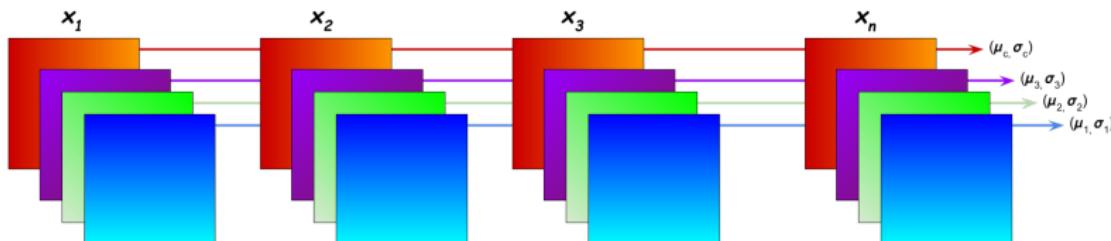
**Algorithm 1:** Batch Normalizing Transform, applied to activation  $x$  over a mini-batch.

# Solution: regularization with Batch normalization

For every channel  $c$  we estimate

$$\mu_c = \frac{1}{NHW} \sum_{i=1}^N \sum_{j=1}^H \sum_{K=W}^N x_{icjk} \text{ and } \sigma_c = \frac{1}{NHW} \sum_{i=1}^N \sum_{j=1}^H \sum_{K=W}^N (x_{icjk} - \mu_c)^2 \quad (57)$$

$$\hat{x} = \frac{x - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}} \quad (58)$$

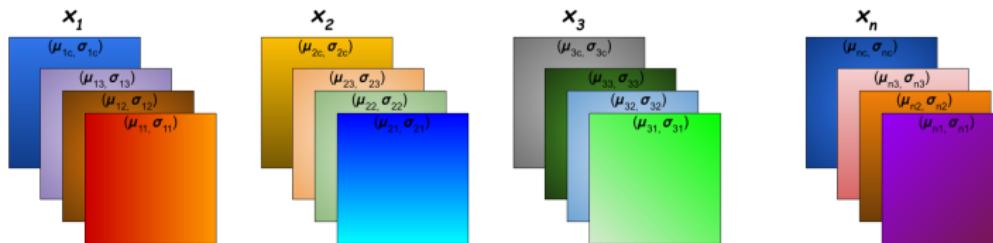


# Solution: regularization Instance Normalization

For every channel  $c$  we estimate

$$\mu_{nc} = \frac{1}{HW} \sum_{j=1}^H \sum_{K=W}^N x_{ncjk} \text{ and } \sigma_{nc} = \frac{1}{HW} \sum_{j=1}^H \sum_{K=W}^N (x_{ncjk} - \mu_{nc})^2 \quad (59)$$

$$\hat{x} = \frac{x - \mu_{nc}}{\sqrt{\sigma_{nc}^2 + \epsilon}} \quad (60)$$

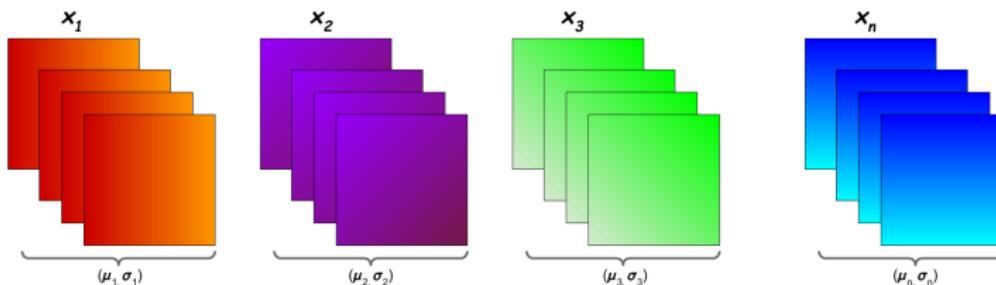


# Solution: regularization Layer Normalization

For every channel  $c$  we estimate

$$\mu_n = \frac{1}{CHW} \sum_{i=1}^C \sum_{j=1}^H \sum_{K=W}^N x_{nijk} \text{ and } \sigma_n = \frac{1}{CHW} \sum_{i=1}^C \sum_{j=1}^H \sum_{K=W}^N (x_{nijk} - \mu_n)^2 \quad (61)$$

$$\hat{x} = \frac{x - \mu_n}{\sqrt{\sigma_n^2 + \epsilon}} \quad (62)$$



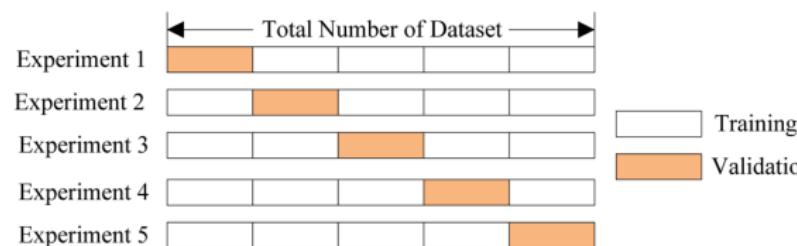
# Solution: Cross validation

## Data sets

- If possible, make 3 sets : training, validation, test
- Use Training for training ...
- Use Validation to check training quality, tune algorithm params
- Use test only to report final performance (hidden in ML competitions)

## K-fold Cross validation

- When little data : split dataset in k sets
- Train on  $k-1$ , validate on remaining one
- Repeat  $k$  times
- Report mean performances



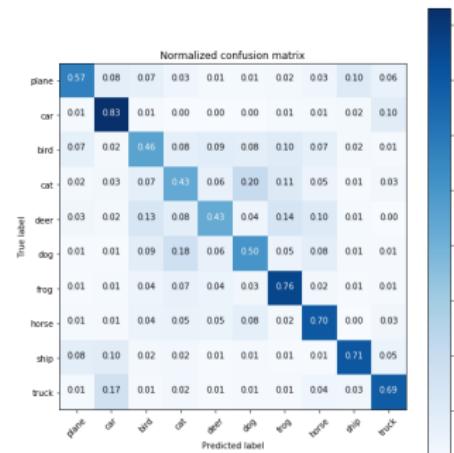
# Solution: Reporting performances

## Detection performance

- precision, recall
- F1 score : harmonic mean of precision/recall
- mAP

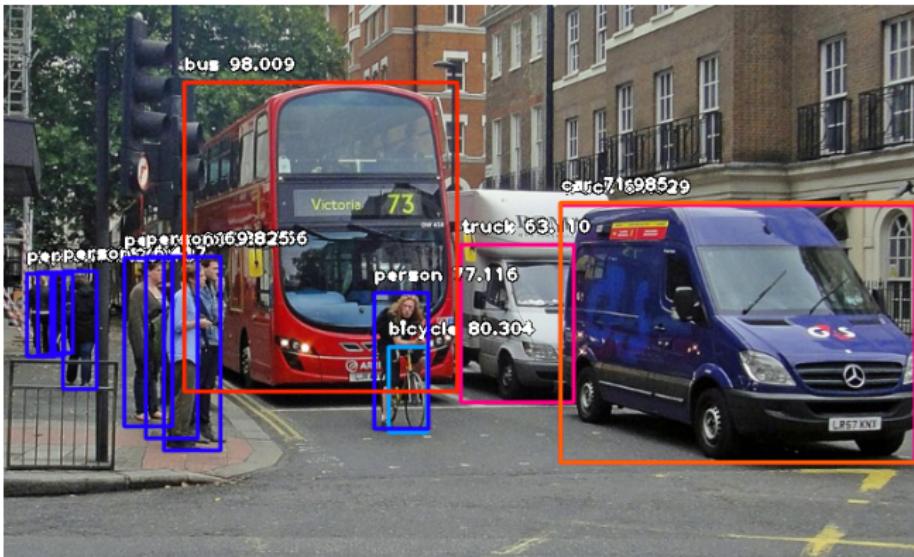
## Classification performance

- Accuracy
- Confusion matrix



- 1 Linear Regression
- 2 Unlearned feature space-Kernel
- 3 Typical recognition Algorithm
- 4 Neural Network
- 5 Convolutional Neural Network
- 6 Transformer architecture
- 7 Training a neural network
- 8 Regularization
- 9 Examples of applications of classical CNN

# object detection



# Style transfer



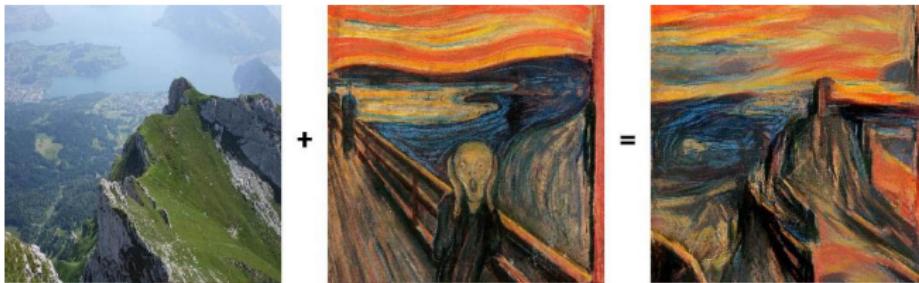
# Segmentation



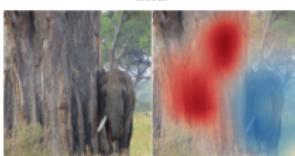
# Deep dream



# Style transfer



# Image captioning

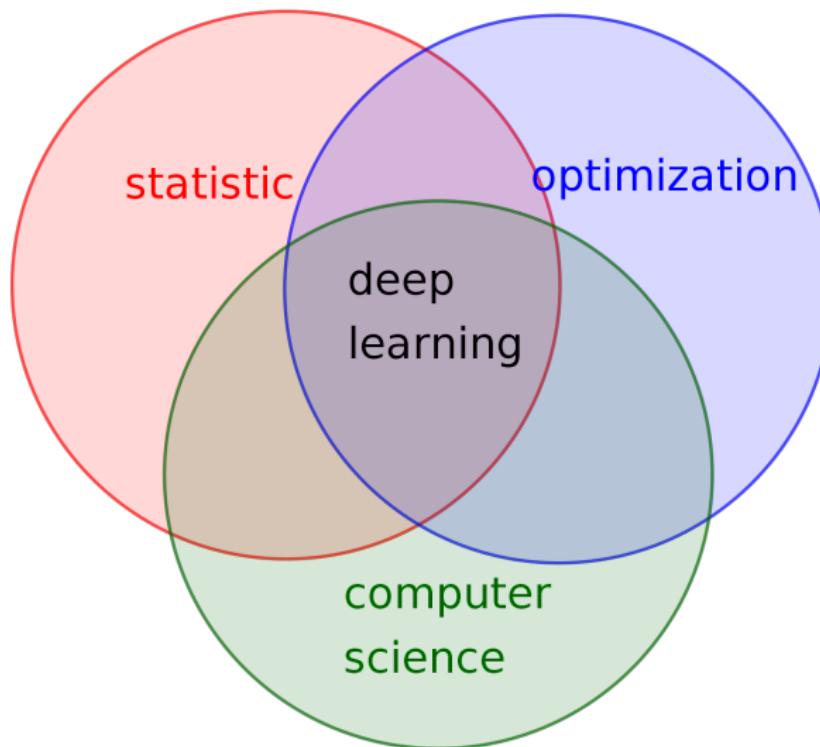


# Ganimation



- 1 Linear Regression
- 2 Unlearned feature space-Kernel
- 3 Typical recognition Algorithm
- 4 Neural Network
- 5 Convolutional Neural Network
- 6 Transformer architecture
- 7 Training a neural network
- 8 Regularization
- 9 Examples of applications of classical CNN

# What is deep learning?



# Conclusions on what we saw

We presented:

- the linear regression and examples of more evolved regressions,
- perceptron algorithm,
- deep learning principle,
- how to train it (optimize it and generalize it),
- A bit about GAN,

# The important points we did not see

More advanced topics :

- deep learning code samples,
- recurrent neural networks,
- object detection
- loss functions
- more evolved optimization