Deep learning introduction

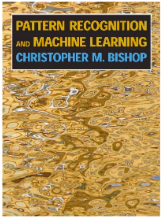
Master 2 Image Mining Course

Gianni Franchi

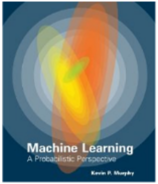
29/09/2021

Plan

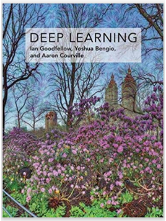
Some references



(a)



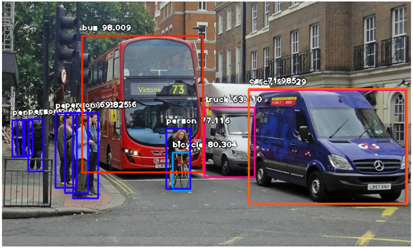
(b)



(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)

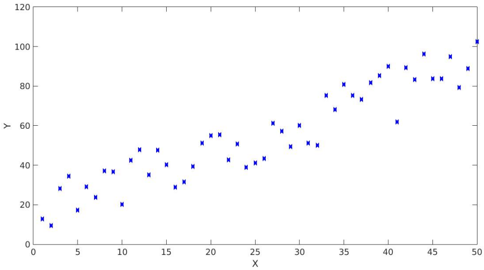
Example of applications



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

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:

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

Our goal is to learn the parameters $\omega = \{\omega_0, \ldots, \omega_D\}$ 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 ϵ of zero mean Gaussian and inverse variance β, such that:

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

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

The linear regression

Hence, we call τ_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, ω and β 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 ω and β:

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

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

Taking the logarithm of the likelihood function, we have:

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

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 ω 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

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

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

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

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

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

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

The linear regression

It is also possible to estimate β_{ML} as:

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

such that β_{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:

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

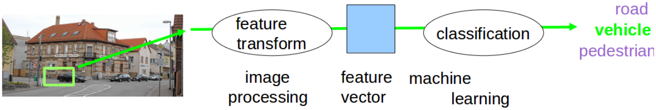
By following the same calculus as previously the solution is:

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

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

Typical recognition Algorithm



Standard procedure

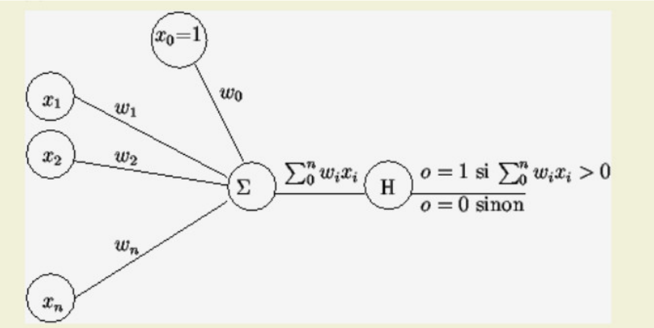
- Feature transform: problem-dependent, hand-crafted, transforms image into a form useful for classification
- Classification: generic, trained, takes feature vector and produces decision

History of Deep learning

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

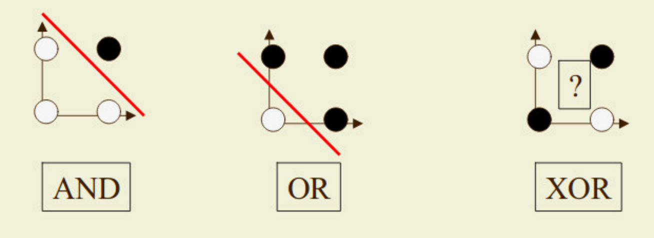
Perceptron algorithm

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



Perceptron algorithm

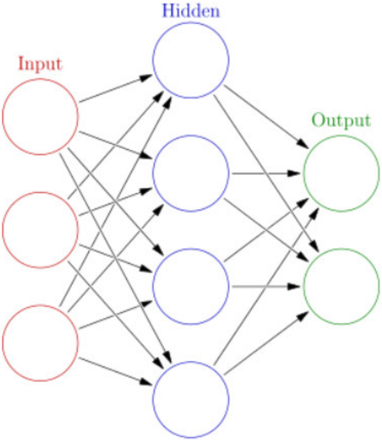
The issue is the XOR. How to solve it?



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:

$$f(\omega, x_i) = \omega_0 + \sum_{j=1}^{D} \omega_j x_{i,j}. \tag{5}$$

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

$$c_k = \omega_{0,k}^{(1)} + \sum_{j=1}^{D} \omega_{j,k}^{(1)} v_{i,j}. \tag{6}$$

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:

$$z_k = a^{(1)}(c_k). \tag{7}$$

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 $c_1, c_2, \ldots, c_{K_1}$ (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:

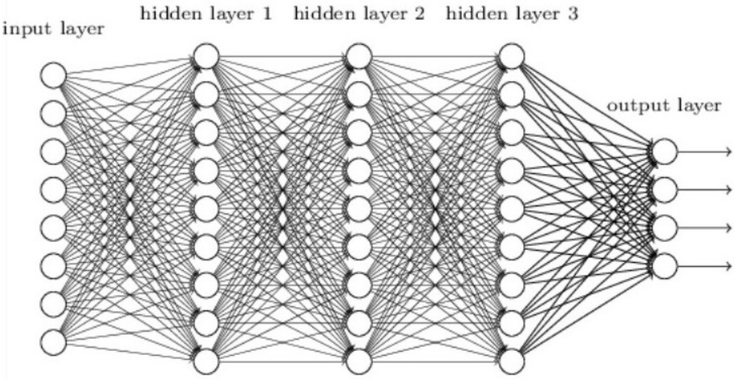
$$d_k = \omega_{0,k}^{(2)} + \sum_{k_1=1}^{K_1} \omega_{k_1,k}^{(2)} z_{k_1}. \tag{8}$$

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 :

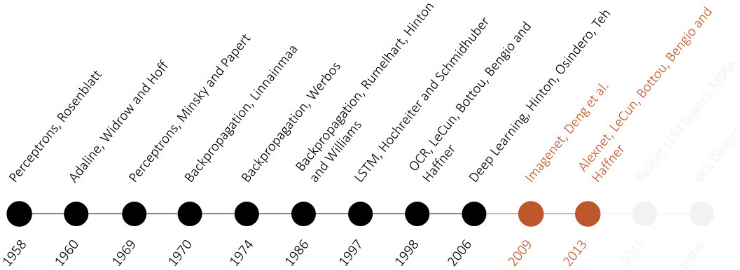
$$d_k = \omega_{0,k}^{(2)} + \sum_{k_1=1}^{K_1} \omega_{k_1,k}^{(2)} a^{(1)}(\omega_{0,k_1}^{(1)} + \sum_{j=1}^{D} \omega_{j,k_1}^{(1)} v_{i,j}). \tag{9}$$

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



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:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] \tag{10}$$

or equivalently (see commutativity) by:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[n-m]g[m]. \tag{11}$$

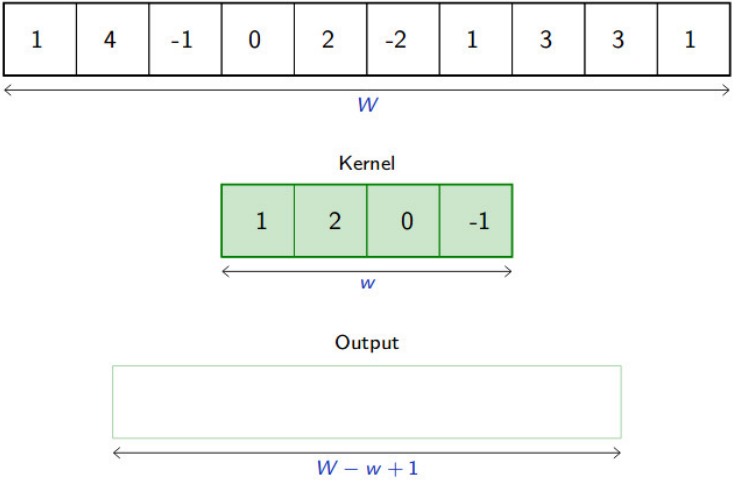
when g and f have finite supports; g in the set $\{-M, -M+1, \dots, M-1, M\}$ and f in $\{0, 1, \dots, N-1, N\}$ a finite summation is used:

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m]g[m] \; \forall n \in [M, N-M] \tag{12}$$

with $M \leq N$

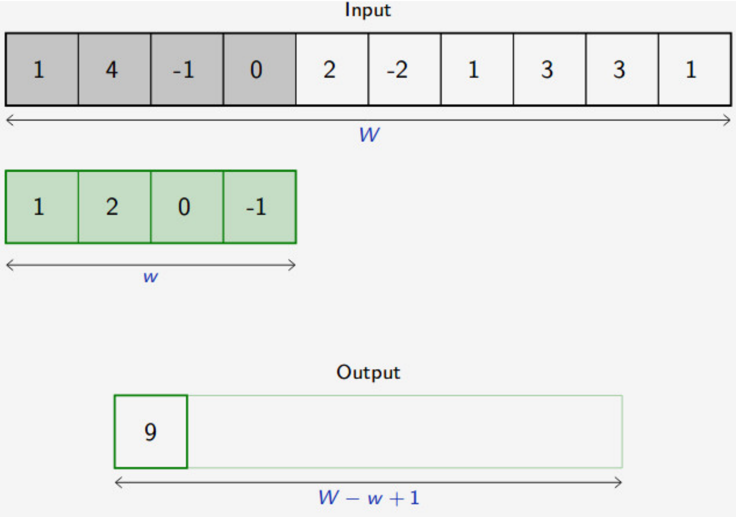
Example 1D convolution for deep learning[1]

Be careful, this is the cross-correlation.



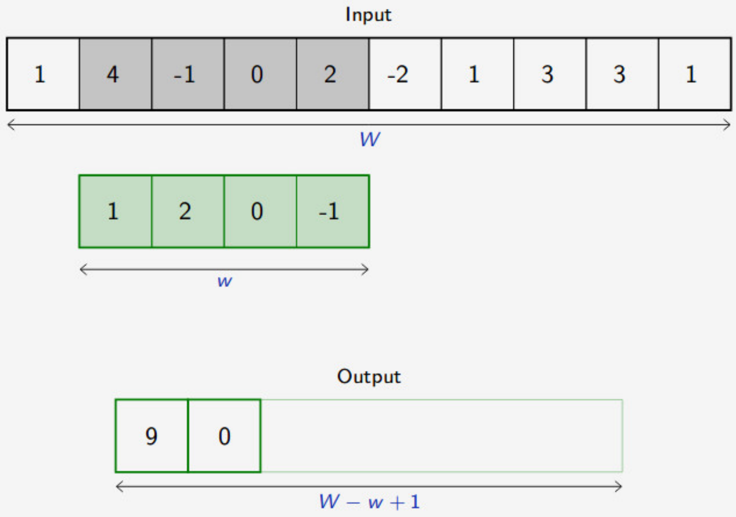
[1]Credits: Francois Fleuret

Example 1D convolution for deep learning[2]



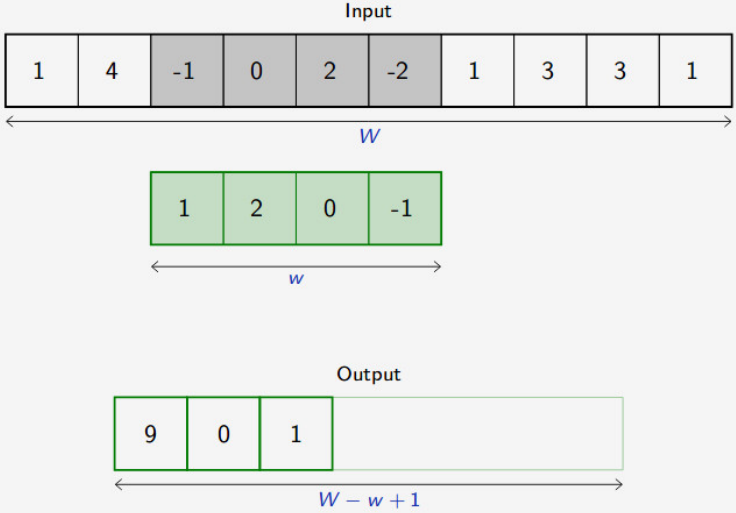
[2] Credits: Francois Fleuret

Example 1D convolution for deep learning[3]



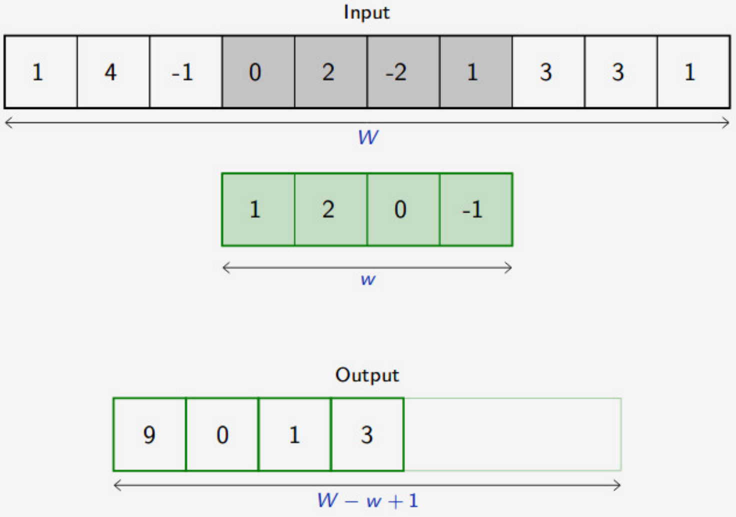
[3] Credits: Francois Fleuret

Example 1D convolution for deep learning[4]



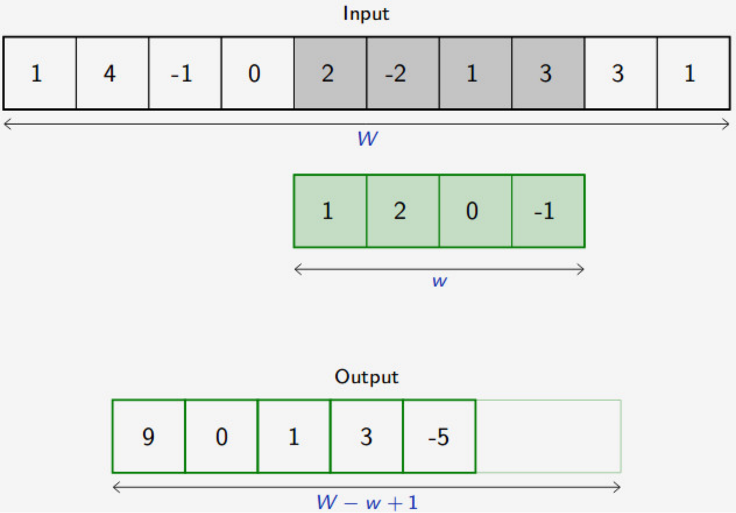
[4]Credits: Francois Fleuret

Example 1D convolution for deep learning[5]



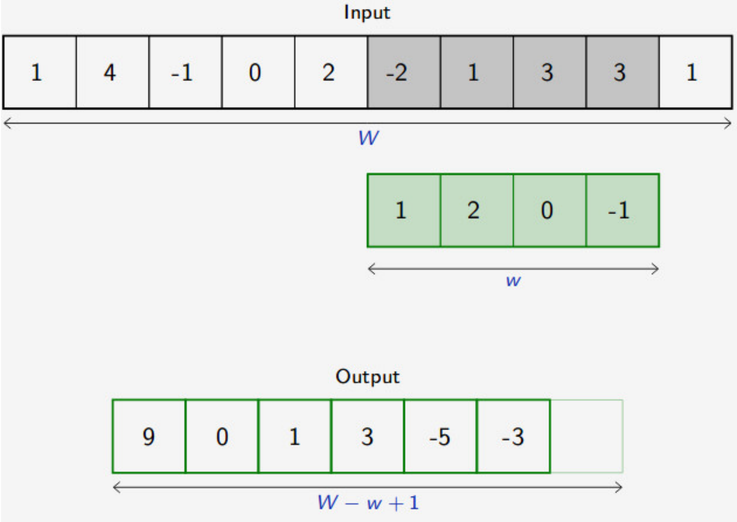
[5] Credits: Francois Fleuret

Example 1D convolution for deep learning[6]



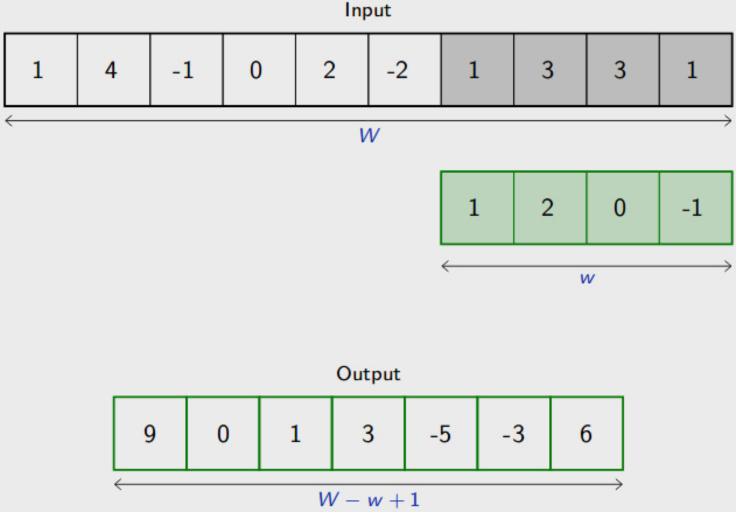
[6] Credits: Francois Fleuret

Example 1D convolution for deep learning[7]



[7] Credits: Francois Fleuret

Example 1D convolution for deep learning[8]



[8] 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 $\{-M, -M+1, \ldots, M-1, M\}^2$ and f in $\{0, 1, \ldots, N-1, N\}^2$ We can define the 2D convolution for all $(n_1, n_2) \in [M, N-M]^2$

$$(f * g)[n_1, n_2] = \sum_{m_1=-M}^{M} \sum_{m_2=-M}^{M} f[n_1 - m_1, n_2 - m_2] g[m_1, m_2] \quad (13)$$

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

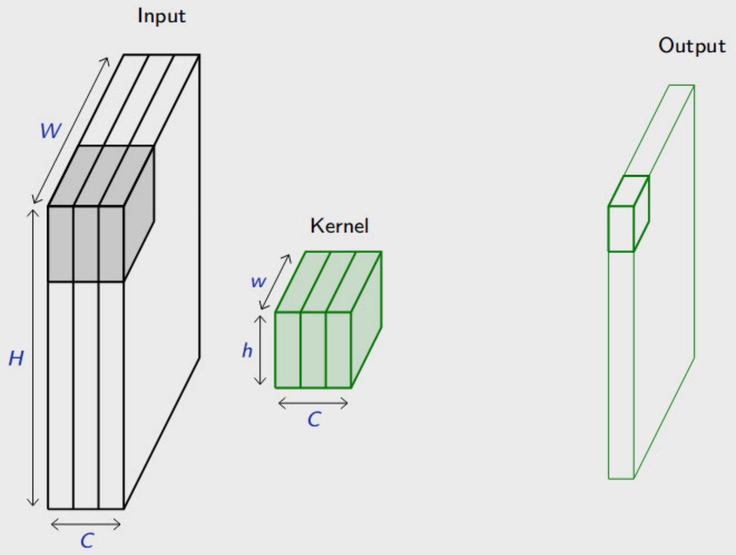
$$(f * g)[n_1, n_2] = \sum_{k=0}^{3} \sum_{m_1=-M}^{M} \sum_{m_2=-M}^{M} f[n_1 - m_1, n_2 - m_2, k] g[m_1, m_2, k] \quad (14)$$

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:

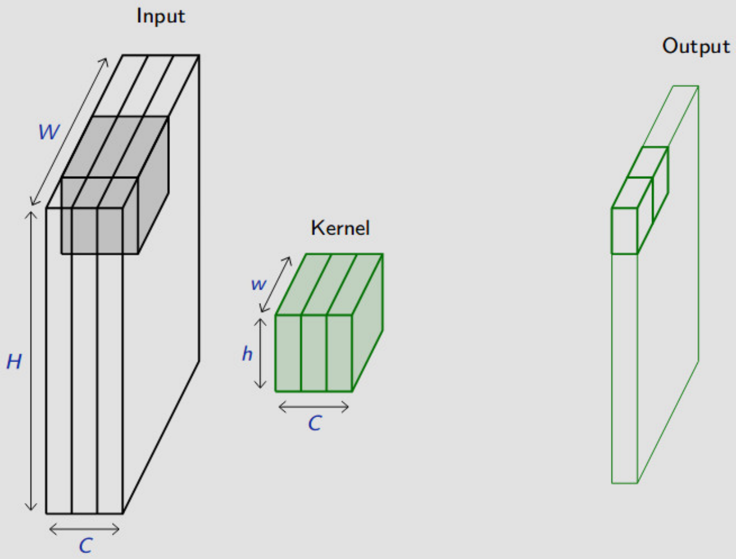
$$(f * g)[n_1, n_2] = \sum_{k=0}^{3} \sum_{m_1=-M}^{M} \sum_{m_2=-M}^{M} f[n_1 + m_1, n_2 + m_2, k] g[m_1, m_2, k]. \quad (15)$$

Example 2D convolution[9]



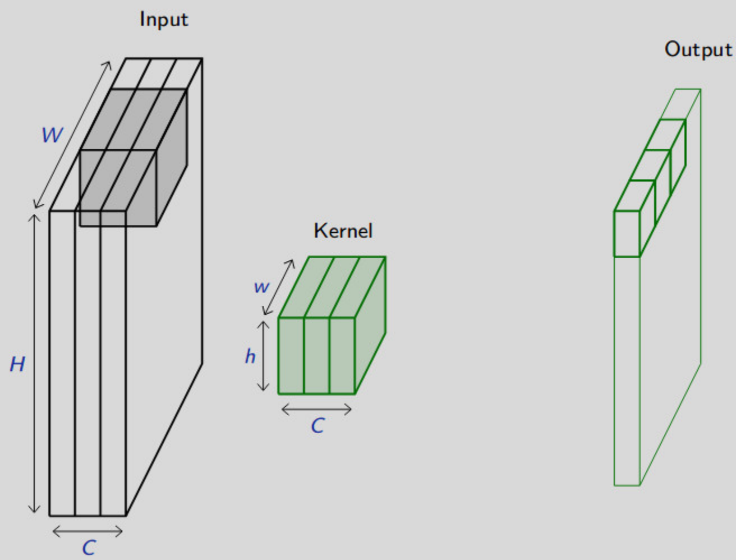
[9] Credits: Francois Fleuret

Example 2D convolution[10]



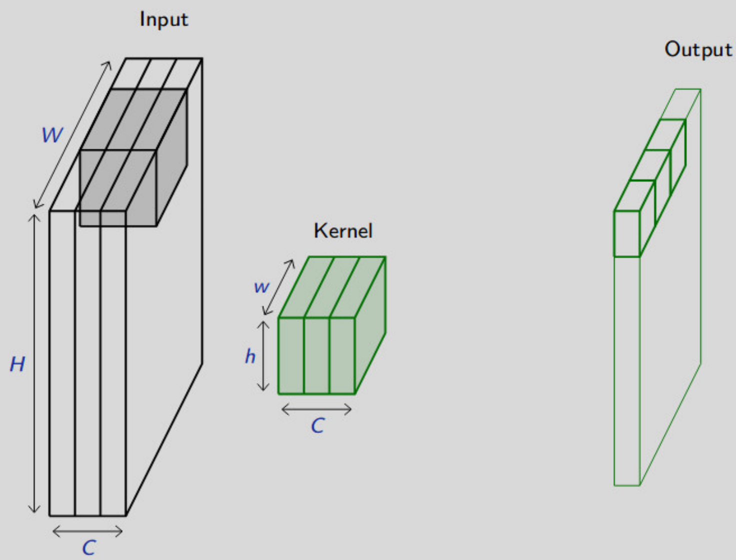
[10] Credits: Francois Fleuret

Example 2D convolution[11]



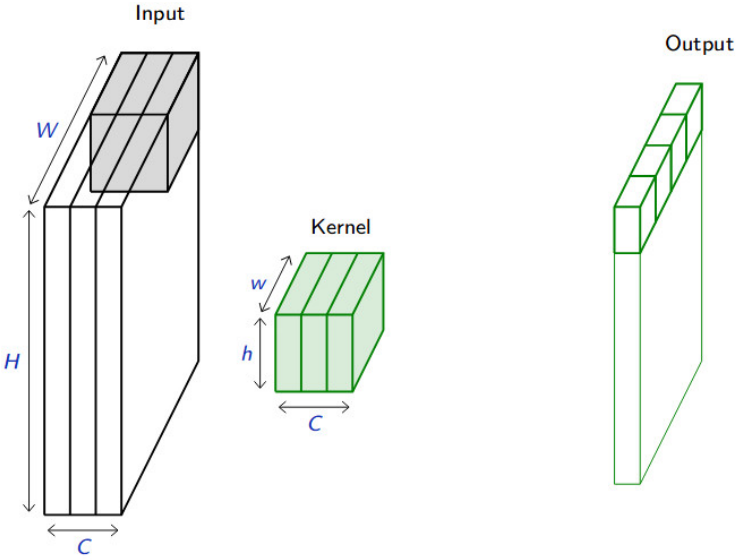
[11] Credits: Francois Fleuret

Example 2D convolution[12]



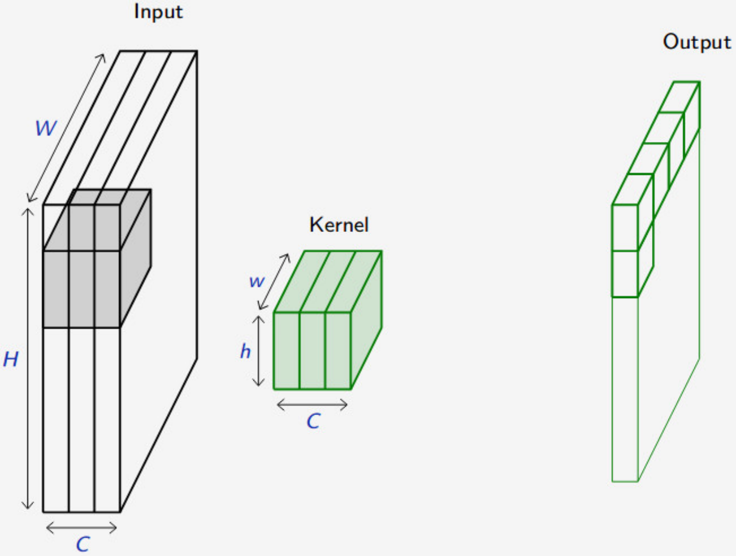
[12] Credits: Francois Fleuret

Example 2D convolution[13]



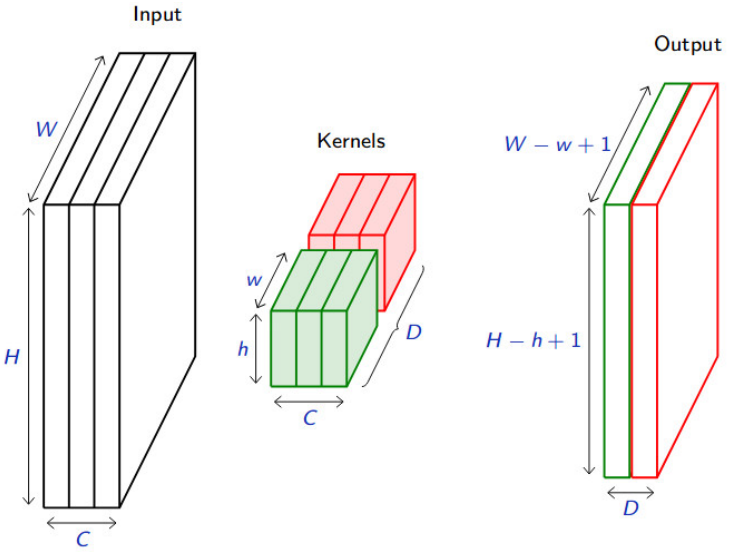
[13] Credits: Francois Fleuret

Example 2D convolution[14]



[14] Credits: Francois Fleuret

Example 2D convolution[15]



[15] Credits: Francois Fleuret

2D convolution

- Let $\mathrm{f} \in \mathbb{R}^{C_{\text{in}} \times H \times W}$ be an image. it is a **3D tensor** called the input **feature map**.
- Let $\mathrm{u} \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times h \times w}$ 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**

$$\mathrm{o}[C_{\text{out},j}] = \mathrm{bias}[C_{\text{out},j}] + \sum_{k=0}^{C_{\text{in}}} \sum_{n=0}^{h-1} \sum_{m=0}^{w-1} \mathrm{f}[k, n+j, m+i] \mathrm{u}[C_{\text{out},j}, k, n, m] \quad (16)$$

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

2D convolution

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

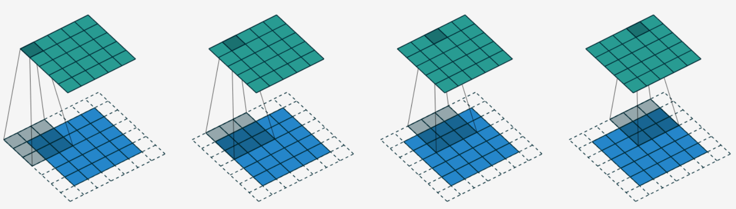
- The padding which specifies number of zeros concatenated at the beginning and at the end of an axis
- The stride which specifies a step size when moving the kernel across the signal.
- The dilation which modulates the expansion of the filter without adding weights.

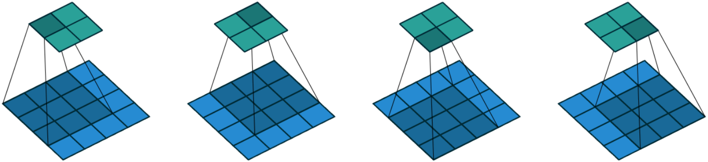
$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (h-1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

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

2D convolution[16]

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

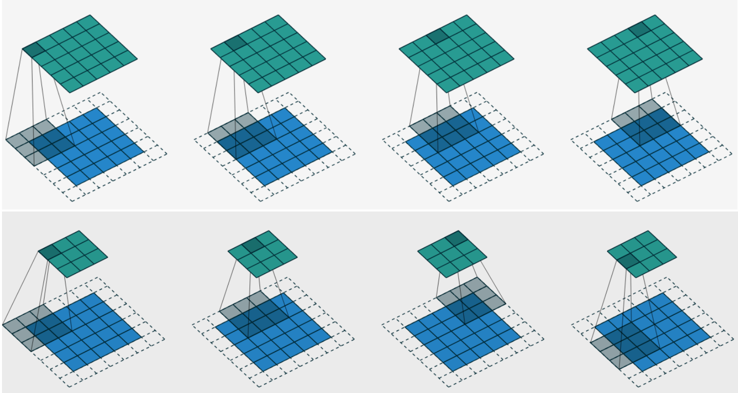




[16] Credits: https://arxiv.org/pdf/1603.07285.pdf

2D convolution[17]

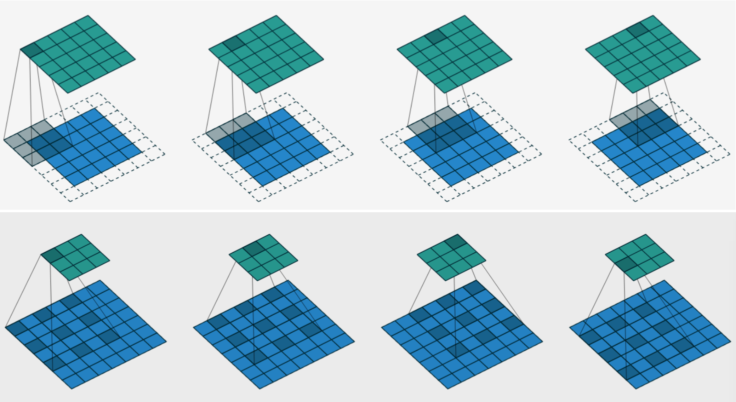
Stride is useful to reduce the spatial dimension of the feature map by a constant factor.



[17] Credits: https://arxiv.org/pdf/1603.07285.pdf

2D convolution[18]

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.



[18] 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

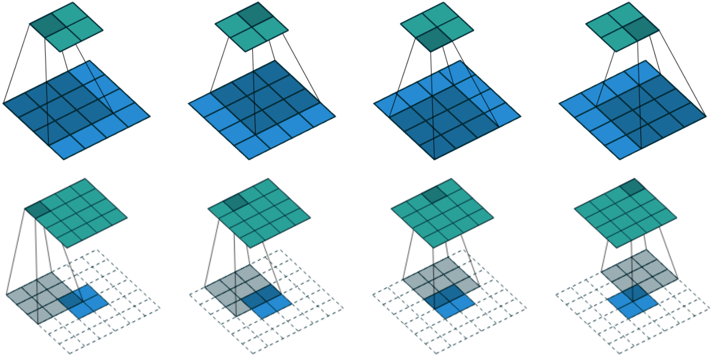
The convolution operation can be equivalently re-expressed as a single matrix multiplication:
the convolutional kernel u is rearranged as a sparse Toeplitz circulant matrix, called the convolution matrix:

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

the input x is flattened row by row, from top to bottom:
$x = \begin{pmatrix} 4 & 5 & 8 & 7 & 1 & 8 & 8 & 8 & 3 & 6 & 6 & 4 & 6 & 5 & 7 & 8 \end{pmatrix}^T$
Then, $v(x) = \begin{pmatrix} 122 & 148 & 126 & 134 \end{pmatrix}^T$ which we can reshape to a 2×2 matrix to obtain $x \circledast u$.

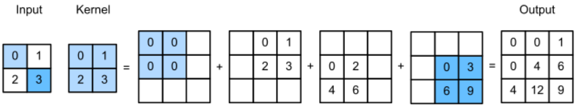
Transposed convolution [19]

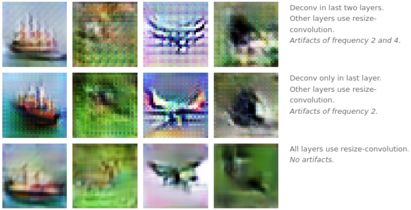
The need for **transposed convolutions** generally arises from the desire to use atransformation going in the opposite direction of a normal convolution, This operationis known as **deconvolution**.



[19] Credits: https://arxiv.org/pdf/1603.07285.pdf

Transposed convolution [20]





[20] 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:

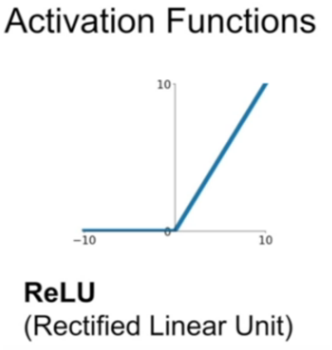
- Convolution layer
- Activation layer
- Pooling layer
- Fully connected layer

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.

$$f(x) = \max(0, x) \tag{17}$$



Pooling layer

Consider a pooling area of size $h \times w$ and a 3D input tensor $\mathrm{x} \in \mathbb{R}^{C \times (rh) \times (sw)}$.
Max-pooling produces a tensor $\mathrm{o} \in \mathbb{R}^{C \times r \times s}$ such that

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

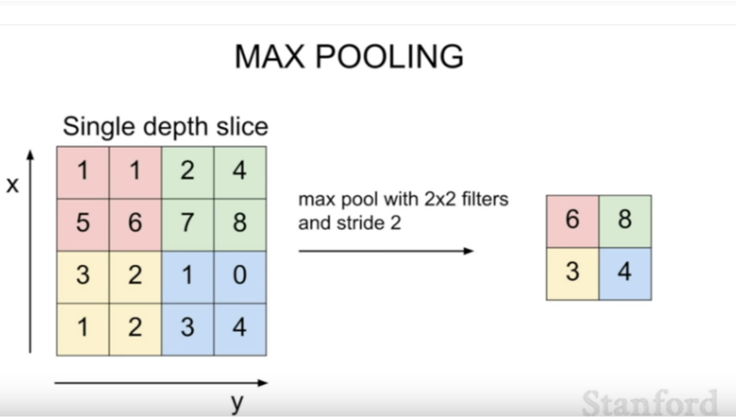
Average pooling produces a tensor $\mathrm{o} \in \mathbb{R}^{C \times r \times s}$such that

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

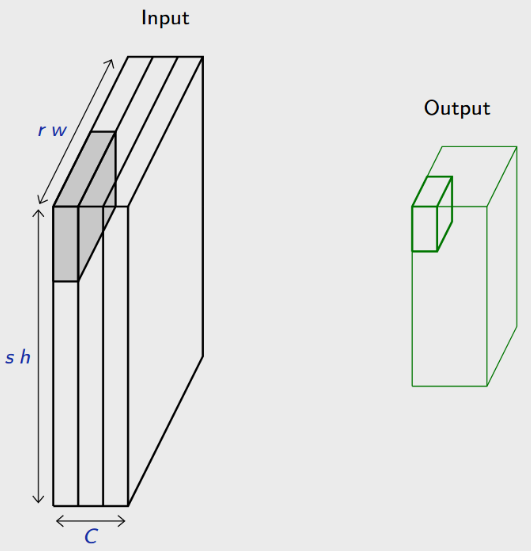
Pooling is very similar in its formulation to convolution.

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.

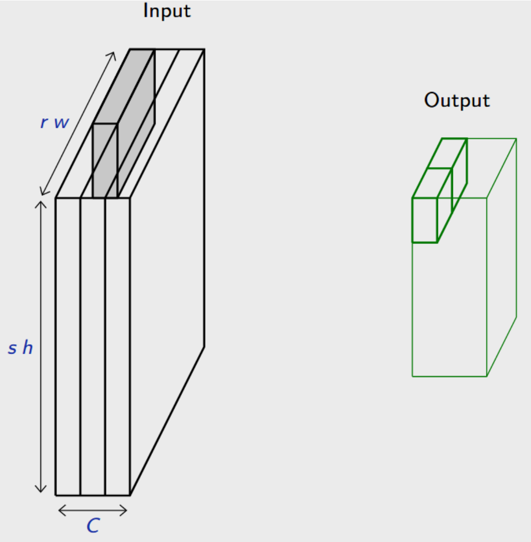


Example 2D pooling[21]



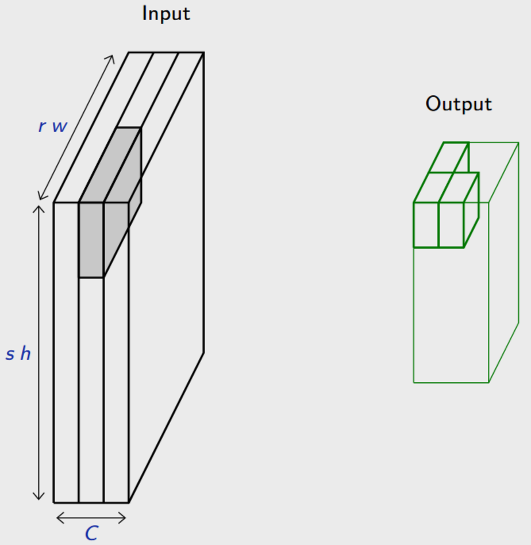
[21] Credits: Francois Fleuret

Example 2D pooling[22]



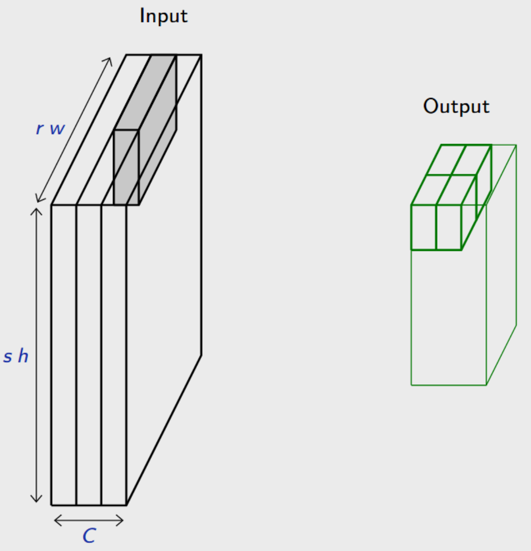
[22] Credits: Francois Fleuret

Example 2D pooling[23]



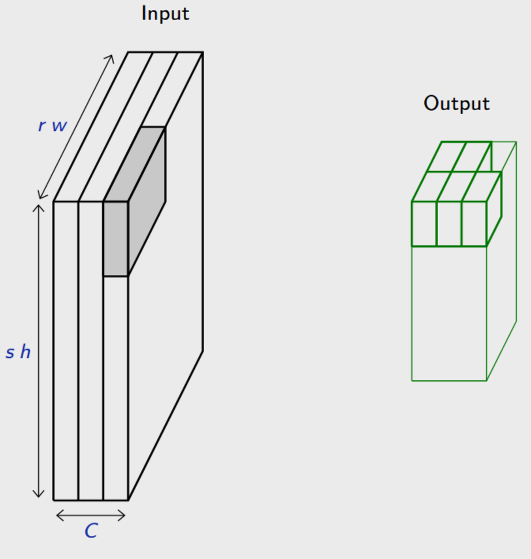
[23] Credits: Francois Fleuret

Example 2D pooling[24]



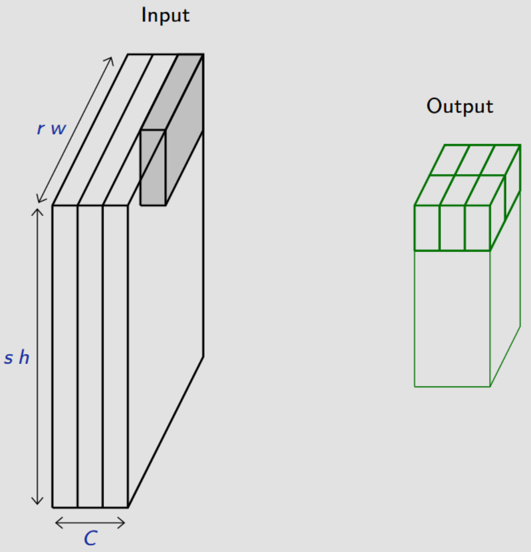
[24] Credits: Francois Fleuret

Example 2D pooling[25]



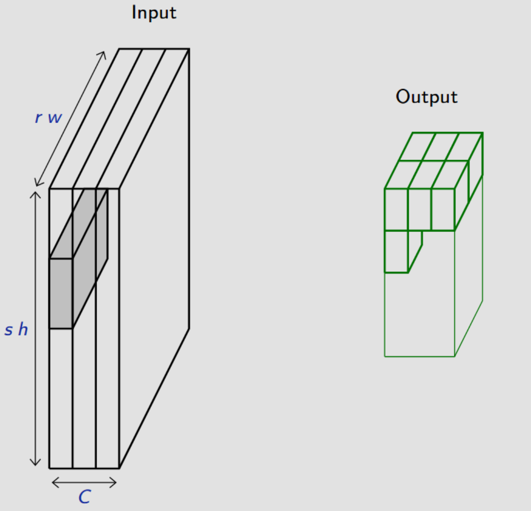
[25] Credits: Francois Fleuret

Example 2D pooling[26]



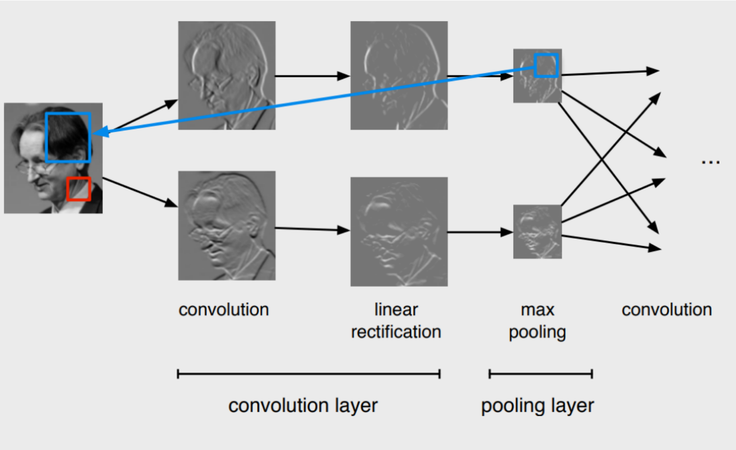
[26] Credits: Francois Fleuret

Example 2D pooling[27]

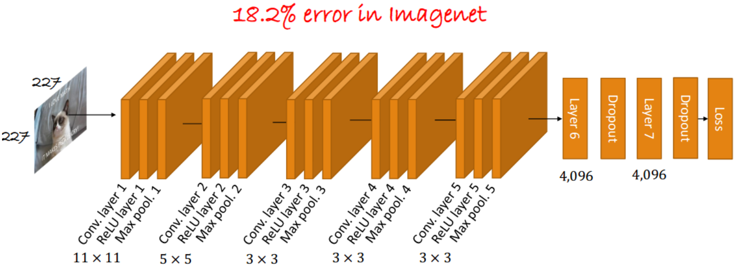


[27] Credits: Francois Fleuret

CNN : architecture



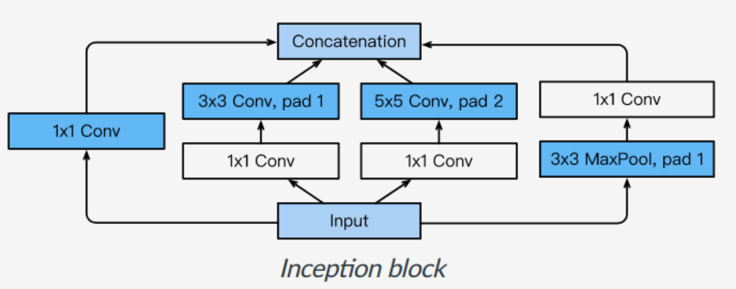
Example of CNN : AlexNet



Example of CNN : VGG

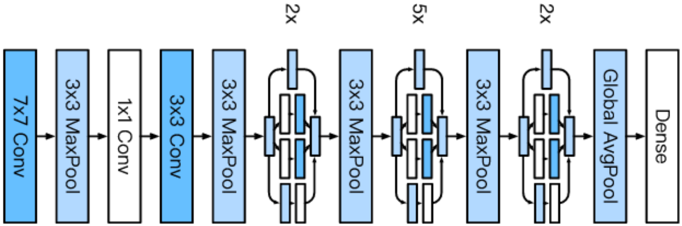
Example of CNN : GoogLeNet [28]

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



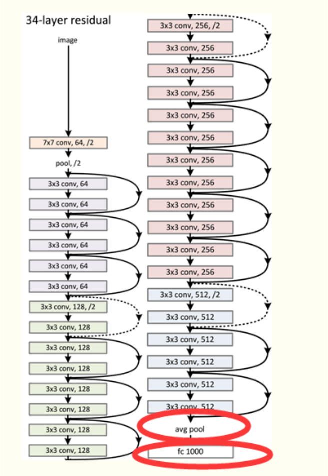
[28] Credits: Dive Into Deep Learning, 2020.

Example of CNN : GoogLeNet [29]



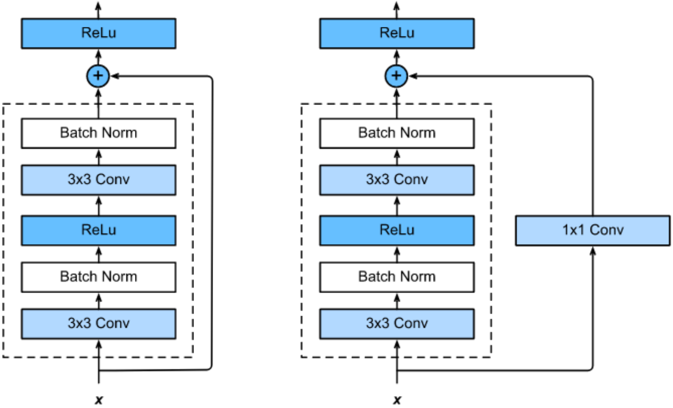
[29] Credits: Dive Into Deep Learning, 2020.

Example of CNN : resnet 34



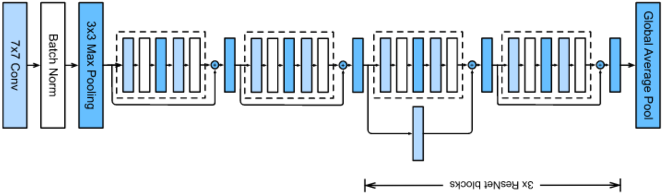
Example of CNN : resnet [30]

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.



[30] Credits: Dive Into Deep Learning, 2020.

Example of CNN : resnet [31]

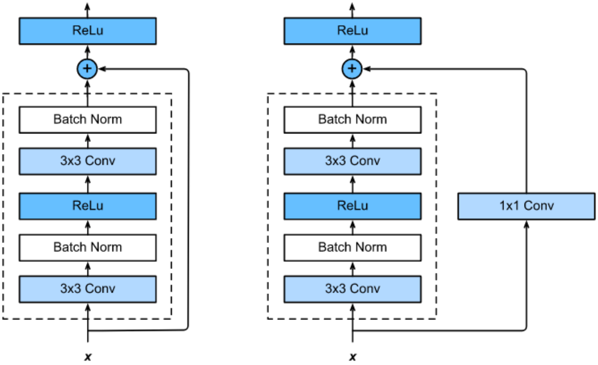


[31] Credits: Dive Into Deep Learning, 2020.

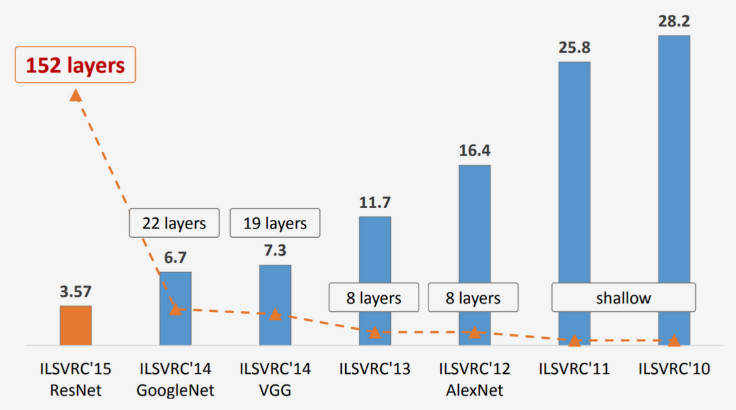
CNN

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 [32]



[32] Credits: Gilles Louppe

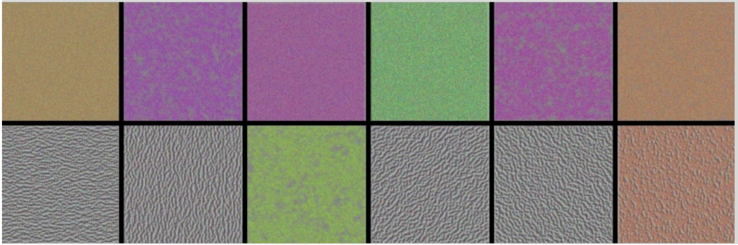
Inside a CNN [33]

AlexNet's first convolutional layer, first 20 filters.

[33] Credits: Gilles Louppe

Inside a CNN [34]

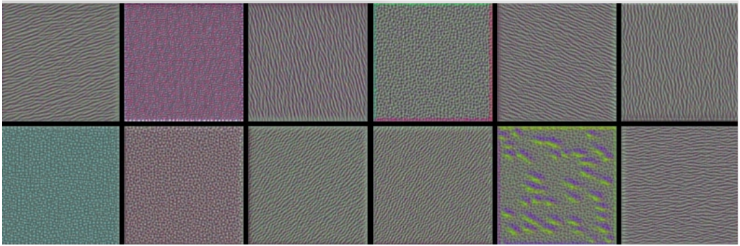
VGG-16, convolutional layer 1-1, a few of the 64 filters



[34] Credits: Gilles Louppe

Inside a CNN [35]

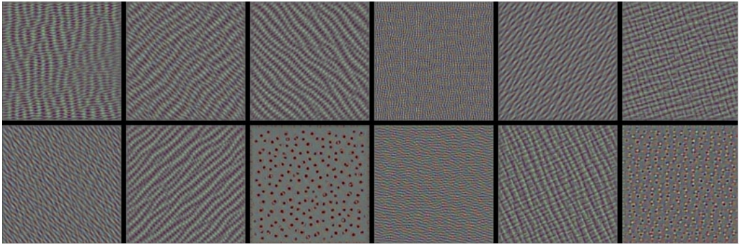
VGG-16, convolutional layer 2-1, a few of the 128 filters



[35] Credits: Gilles Louppe

Inside a CNN [36]

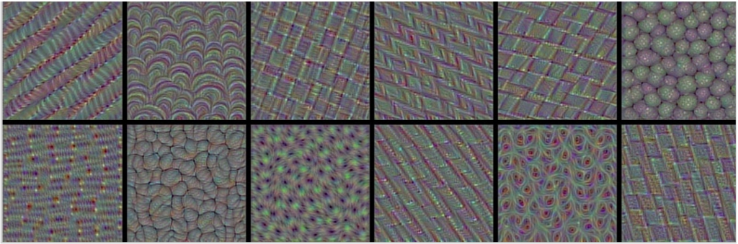
VGG-16, convolutional layer 3-1, a few of the 256 filters



[36] Credits: Gilles Louppe

Inside a CNN [37]

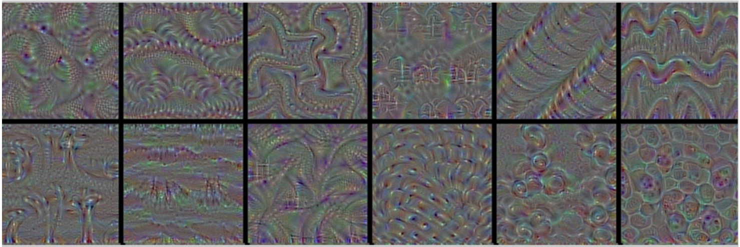
VGG-16, convolutional layer 4-1, a few of the 512 filters



[37] Credits: Gilles Louppe

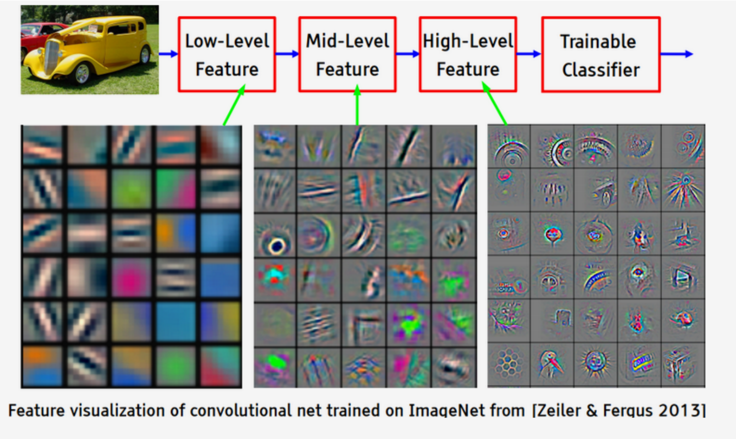
Inside a CNN [38]

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



[38] Credits: Gilles Louppe

Inside a CNN [39]



[39] Credits: Gilles Louppe

Attention layer [40]

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

[40] Credits: Jay Alammar

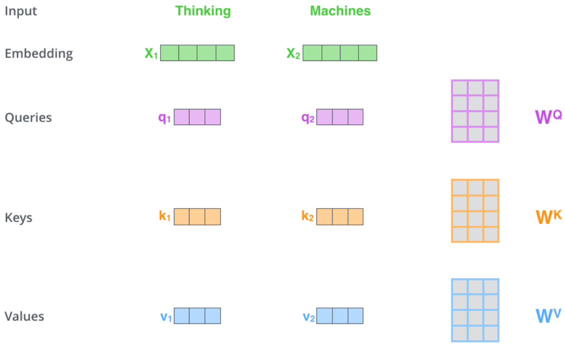
Attention layer [41]

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

[41] Credits: Jay Alammar

Attention layer [42]

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.



[42] Credits: Jay Alammar

Attention layer [43]

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 α which measure the similarity between the query and a key. Then we compute all n scores $a_1, \ldots, a_n$ defined by

$$ai = \alpha(q, ki).$$

Next we use softmax to obtain the attention weights

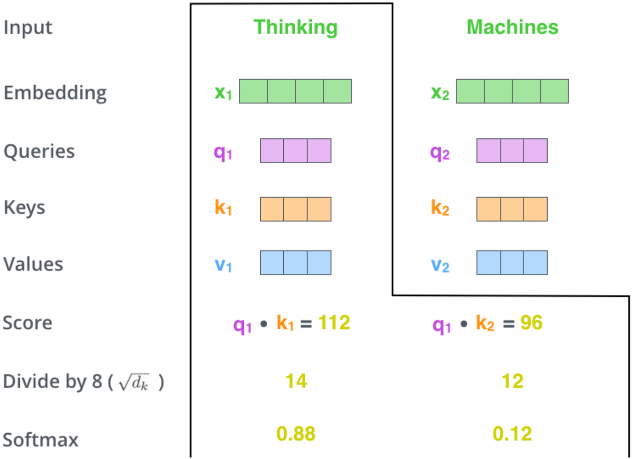
$$b_1, \ldots, b_n = softmax(a_1, \ldots, a_n).$$

The final output is a weighted sum of the values

$$o = \sum_i b_i v_i.$$

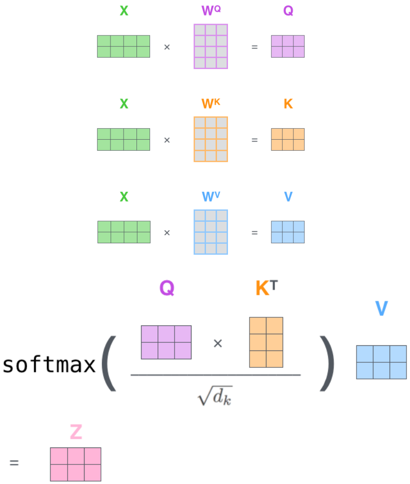
[43] Credits: d2l.ai

Transformer layer [44]



[44] Credits: Jay Alammar

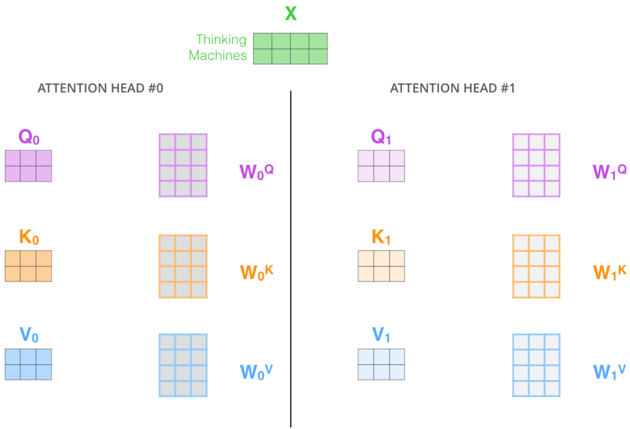
Attention layer [45]



[45] Credits: Jay Alammar

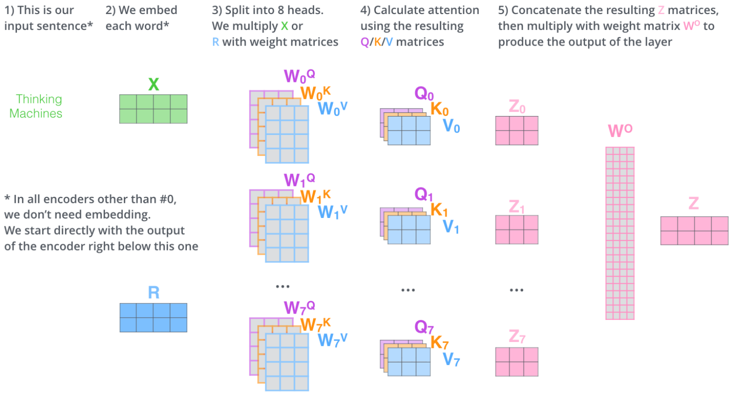
Attention layer [46]

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



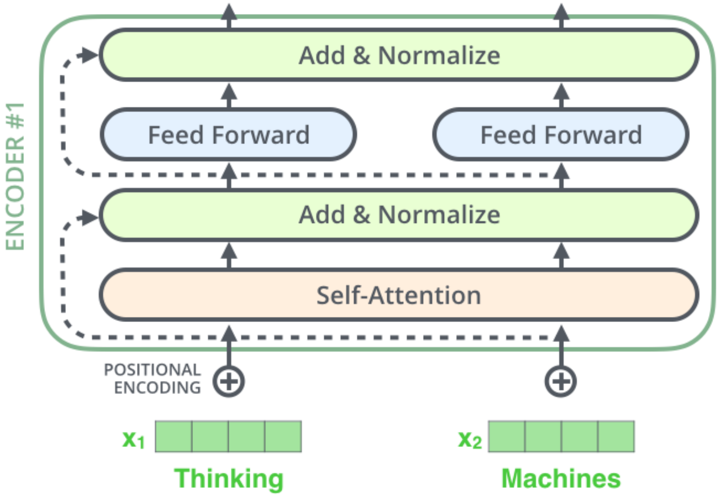
[46] Credits: Jay Alammar

multi-headed Self-Attention layer [47]



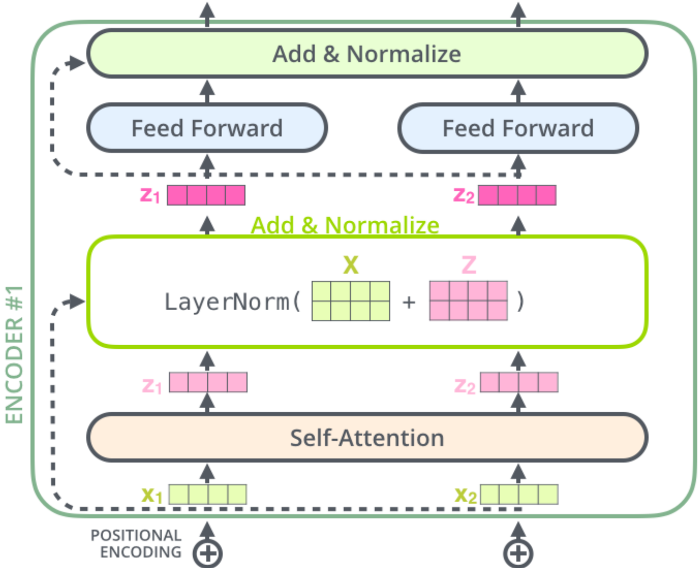
[47] Credits: Jay Alammar

multi-headed Self-Attention layer [48]

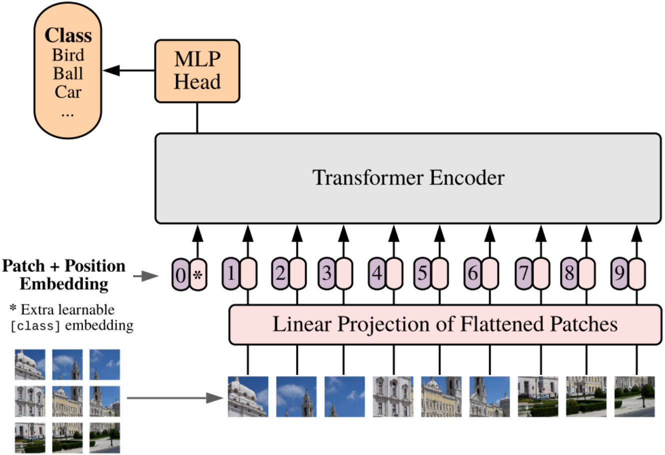


[48] Credits: Jay Alammar

multi-headed Self-Attention layer [49]



VIT [50]



[50] https://arxiv.org/pdf/2010.11929.pdf

Optimization

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

$$\mathcal{F}(\omega) = \frac{\beta}{2} \sum_{i=1}^{N_1} \| f(\omega, x_i) - t_i \|^2. \tag{18}$$

Now ω 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 ω applied on x_i. Finding the optimal ω 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:

$$\min_{\omega}\mathcal{F}(\omega) \text{ , with } \mathcal{F}(\omega) = \sum_{i=1}^{N_1} \|f(\omega, x_i) - t_i\|^2 \tag{19}$$

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 ω_0.
Then, at each iteration we calculate:

$$\omega_{t+1} = \omega_t - \lambda\frac{\partial \mathcal{F}}{\partial \omega} \tag{20}$$

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

Basic of deep learning optimization

Why does it work?
We remind the derivative of a function:

$$\frac{\partial g}{\partial x} = \lim_{h \to 0} \frac{g(x+h) - g(x)}{h} \tag{21}$$

For simplicity, we consider for h really small :

$$\frac{\partial g}{\partial x} \simeq \frac{g(x+h) - g(x)}{h} \tag{22}$$

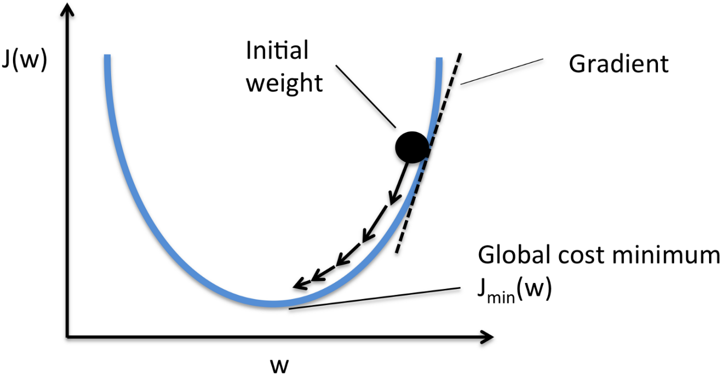
Now let us consider that $h = -\lambda \frac{\partial g}{\partial x}$.
Then have

$$g(x+h) - g(x) \simeq -\lambda \times (\frac{\partial g}{\partial x})^2 \tag{23}$$

Since $\lambda > 0$, then

$$g(x+h) < g(x) \tag{24}$$

Basic of deep learning optimization



Basic of deep learning optimization

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

$$\frac{\partial \mathcal{F}}{\partial \omega} = \frac{\partial}{\partial \omega} \sum_{i=1}^{N_1} (f(\omega, x_i) - y_i)^t (f(\omega, x_i) - y_i) \tag{25}$$

$$\frac{\partial \mathcal{F}}{\partial \omega} = \frac{\partial}{\partial \omega} \sum_{i=1}^{N_1} \left(f(\omega, x_i)^t f(\omega, x_i) - 2y_i^t f(\omega, x_i) + y_i^t y_i \right) \tag{26}$$

$$\frac{\partial \mathcal{F}}{\partial \omega} = \sum_{i=1}^{N_1} \left(\frac{\partial}{\partial \omega} f(\omega, x_i)^t f(\omega, x_i) - \frac{\partial}{\partial \omega} 2y_i^t f(\omega, x_i) \right) \tag{27}$$

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

$$\frac{\partial \mathcal{F}}{\partial \omega} \simeq \frac{\partial}{\partial \omega} \sum_{i \in B_j} \|f(\omega, x_i) - y_i\|^2 \tag{28}$$

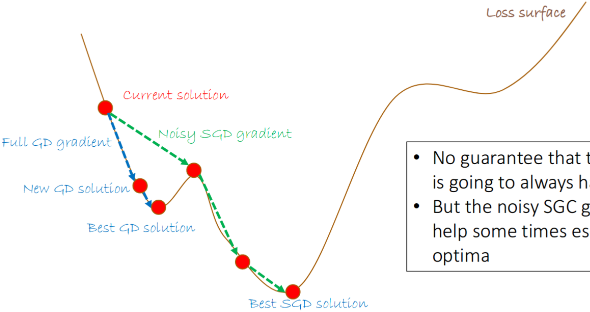
With B_j a sample of the dataset.
One dataset B_j might not be representative of the full dataset so we take all the possible B_j
Hence at each iteration we calculate

$$\omega_{t+1} = \omega_t - \lambda \frac{\partial \mathcal{F}_j}{\partial \omega} \tag{29}$$

with

$$\frac{\partial \mathcal{F}_j}{\partial \omega} = \frac{\partial}{\partial \omega} \sum_{i \in B_j} \|f(\omega, x_i) - y_i\|^2 \tag{30}$$

Stochastic gradient descent



- No guarantee that this is what is going to always happen.
- But the noisy SGC gradients can help some times escaping local optima

stochastic gradient descent with momentum

The stochastic gradient descent

First, we initialized the parameters ω_0.
Then, at each iteration we calculate

$$\omega_{t+1} = \omega_t - \lambda \frac{\partial \mathcal{F}_j}{\partial w} \tag{31}$$

The stochastic gradient descent with momentum

First, we initialized the parameters ω_0.
Then, at each iteration we calculate

$$u_{t+1} = \gamma u_t + \lambda \frac{\partial \mathcal{F}_j}{\partial \omega} \tag{32}$$

$$\omega_{t+1} = \omega_t - u_{t+1} \tag{33}$$

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.

Adam algorithm

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

The Adam algorithm

$$m_{t+1} = \beta_1 m_t + (1-\beta_1)\frac{\partial \mathcal{F}_j}{\partial \omega} \tag{34}$$

$$\hat{m_{t+1}} = \frac{m_{t+1}}{1-\beta_1} \tag{35}$$

$$v_{t+1} = \beta_2 v_t + (1-\beta_2)(\frac{\partial \mathcal{F}_j}{\partial \omega})^2 \tag{36}$$

$$\hat{v_{t+1}} = \frac{v_{t+1}}{1-\beta_2} \tag{37}$$

$$\omega_{t+1} = \omega_t - \frac{\lambda}{\sqrt{\hat{v_{t+1}}} + \epsilon}\hat{m_{t+1}} \tag{38}$$

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

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

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

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

Chain rule

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

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

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

Hence, the chain rule results is:

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

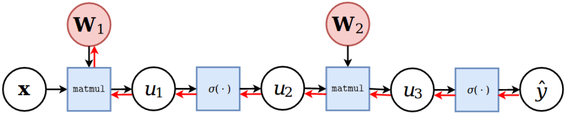
Chain rule

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

$$f(\mathrm{x}; \mathrm{W}_1, \mathrm{W}_2) = \sigma\left(\mathrm{W}_2^T \sigma\left(\mathrm{W}_1^T \mathrm{x}\right)\right)$$

$$\ell(y, \hat{y}; \mathrm{W}_1, \mathrm{W}_2) = \mathrm{cross\_ent}(y, \hat{y})$$

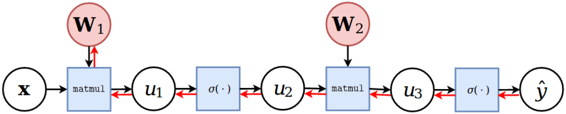
Chain rule[51]



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

[51] Credits: Gilles Louppe

Chain rule[52]

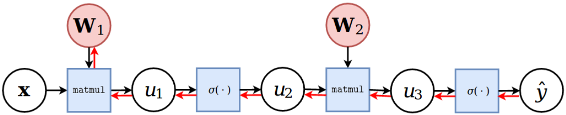


Forward pass: values u_1, u_2, u_3 and $\hat{y}$ are computed by traversing the graph from inputs to outputs given x, W_1 and W_2.

[52] Credits: Gilles Louppe

Chain rule[53]

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

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

[53] Credits: Gilles Louppe

Chain rule[54]

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

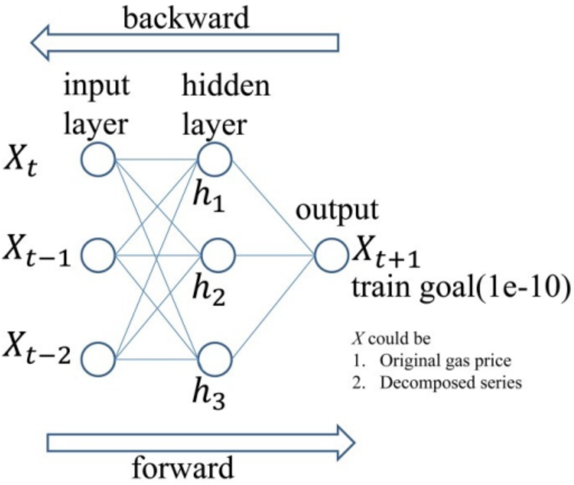
$$\begin{aligned}
u_1 &= w_1x \\
u_2 &= \sigma(u_1) \\
u_3 &= w_2u_2 \\
u_4 &= \sigma(u_3) \\
u_5 &= w_3u_4 \\
\hat{y} &= \sigma(u_5)
\end{aligned}$$

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

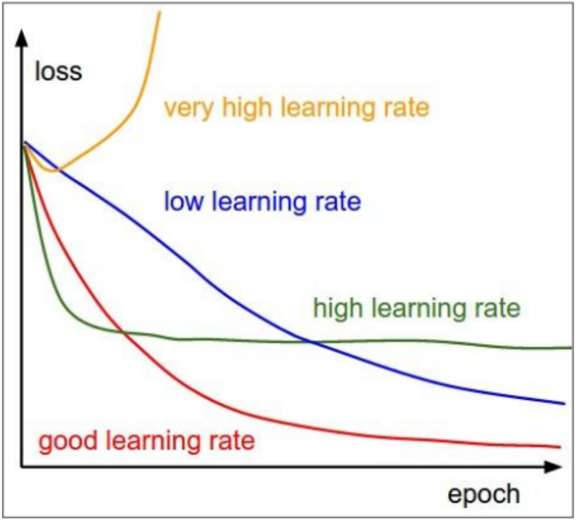
$$\begin{aligned}
\frac{\partial \hat{y}}{\partial w_1} &= \frac{\partial \hat{y}}{\partial u_5}\frac{\partial u_5}{\partial u_4}\frac{\partial u_4}{\partial u_3}\frac{\partial u_3}{\partial u_2}\frac{\partial u_2}{\partial u_1}\frac{\partial u_1}{\partial w_1} \\
&= \frac{\partial \sigma(u_5)}{\partial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

[54] Credits: Gilles Louppe

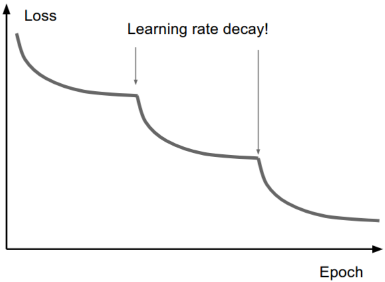
Forward/backward



Which one of these learning rates is best to use?



Which one of these learning rates is best to use?



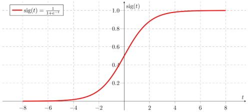
Solution : Learning rate decay over time.

- 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 :

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

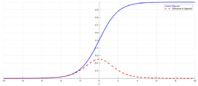


Can you evaluate the derivative?

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:

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} = \underbrace{\frac{\partial \sigma(u_5)}{\partial u_5}}_{\leq 1/4} \underbrace{w_3}_{\leq 1} \underbrace{\frac{\partial \sigma(u_3)}{\partial u_3}}_{\leq 1/4} \underbrace{w_2}_{\leq 1} \underbrace{\frac{\partial \sigma(u_1)}{\partial u_1}}_{\leq 1/4} x$$

This implies that the gradient $\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1}$ shrinks . A solution use Relu, then fore,

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} = \underbrace{\frac{\partial \sigma(u_5)}{\partial u_5}}_{=1} w_3 \underbrace{\frac{\partial \sigma(u_3)}{\partial u_3}}_{=1} w_2 \underbrace{\frac{\partial \sigma(u_1)}{\partial u_1}}_{=1} x$$

initialization of neural networks

In convex problems, provided a good learning rate γ, 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[55] initialisation and He [56] are the most used in practice since the weights depend on the size of the output/input. They have good properties.

[55] Xavier Glorot and Yoshua Bengio (2010): Understanding the difficulty of training deep feedforward neural networks. International conference on artificial intelligence and statistics.

[56] Kaiming He, etal (2015): Delving Deep into Rectifiers:Surpassing Human-Level Performance on ImageNet Classification

He initialization

Let us consider a deep neural network modelled by:

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

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

$a()$ 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:

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

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

He initialization

$$g(x_i,\omega)_{k2} = b^{(3)}_{k2} + \sum_{k1=1}^{M_3} \omega^{(3)}_{k2,k1}.a^{(2)}_{k1} \ \forall k2 \in [1, D_{\text{out}}]$$

These equations are can be synthesize:

$$g(x_i,\omega)_{k2} = b^{(3)}_{k2} + \sum_{k1=1}^{M_3} \omega^{(3)}_{k2,k1}.a^{(2)} \left(b^{(2)}_{k1} + \sum_{k=1}^{M_2} \omega^{(2)}_{k1,k}.a^{(1)} \left(b^{(1)}_{k} + \sum_{j=1}^{D_{\text{in}}} \omega^{(1)}_{k,j} x_{i,j} \right) \right)$$

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 k_2 of this vector.
The variance of the deep neural network is :

$$\text{var}_W(g(x,W)_{k2}) = \mathbb{E}_W\left(g^2(x,W)_{k2}\right) - \left(\mathbb{E}_W g(x,W)_{k2}\right)^2 \qquad (39)$$

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:

$$\mathrm{var}\left(g(x,W)^{(l)}\right) = M_l \mathrm{var}\left(\omega^{(l)} a^{(l-1)}\right) \tag{40}$$

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

Then:

$$\mathrm{var}\left(g(x,W)^{(l)}\right) = M_l \mathrm{var}\left(\omega^{(l)}\right) \mathbb{E}\left((a^{(l-1)})^2\right) \tag{41}$$

He initialization

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

$$\mathbb{E}\left(\left(a^{(l-1)}\right)^2\right) = 1/2\mathrm{var}\left(g(x, W)^{(l-1)}\right) \tag{42}$$

Then we have:

$$\mathrm{var}\left(g(x, W)^{(l)}\right) = M_l/2\mathrm{var}\left(\omega^{(l)}\right)\mathrm{var}\left(g(x, W)^{(l-1)}\right) \tag{43}$$

With L layers put together, we have

$$\mathrm{var}\left(g(x, W)^{(L)}\right) = \mathrm{var}\left(x\right)\prod_{l=2}^{L}\left(M_l/2\mathrm{var}\left(\omega^{(l)}\right)\right) \tag{44}$$

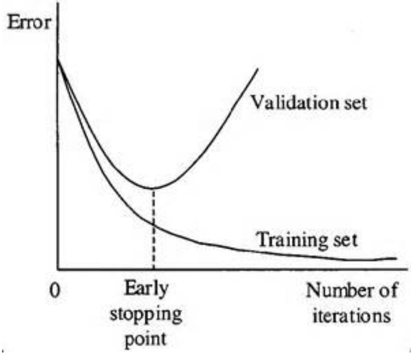
He initialization

A good **initialization** method should avoid **reducing** or **magnifying** the magnitudes of input signals exponentially.
So we want : $\forall l \in [1, L]\ M_l/2\mathrm{var}\left(\omega^{(l)}\right) = 1$

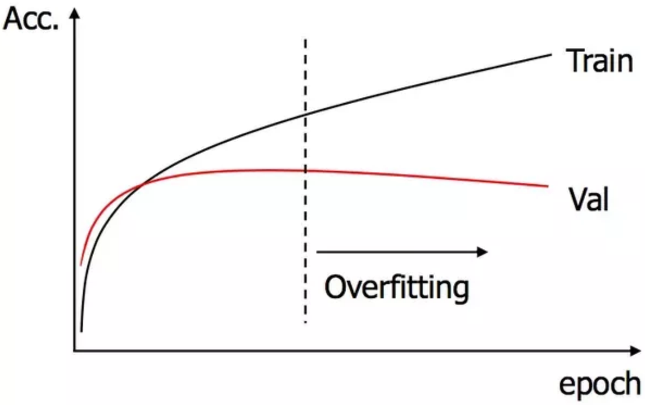
$$\forall l \in [1, L] \ \mathrm{var}\left(\omega^{(l)}\right) = \frac{2}{M_l} \text{ and } \mathbb{E}\left(\omega^{(l)}\right) = 0 \tag{45}$$

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 curbs?



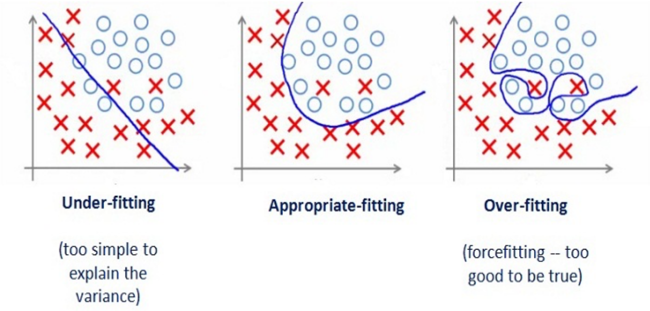
Regularization



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

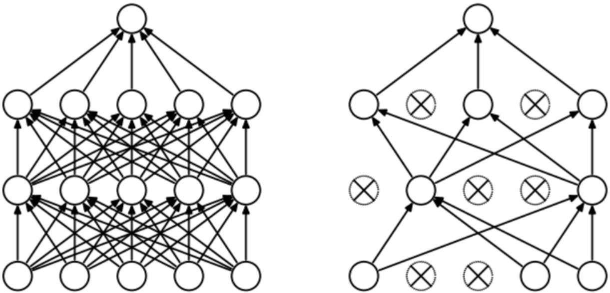
You can use weight decay :

$$\mathcal{L}(\omega) = \mathcal{F}_{\text{data}}(\omega) + \frac{\lambda_2}{2} \|\omega\|^2 \tag{46}$$

Then during the gradient descent we have

$$\frac{\partial \mathcal{F}}{\partial w}(\omega) = \frac{\partial \mathcal{F}_{\text{data}}}{\partial w}(\omega) + \lambda_2 w \tag{47}$$

Solution: regularization with dropout



(a) Standard Neural Net (b) After applying dropout.

Solution: regularization batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

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

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

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

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

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

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

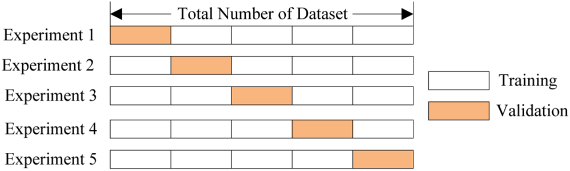
Solution: Cross validation

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 remaning one
- Repeat k times
- Report mean performances



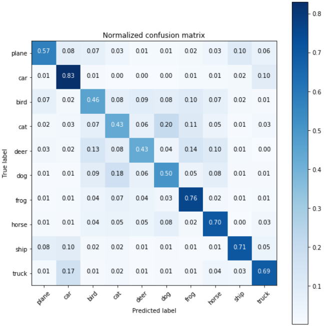
Solution: Reporting performances

Detection performance

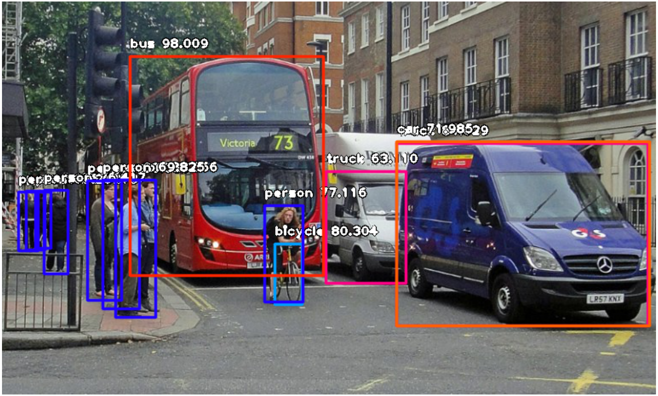
- precision,recall
- F1 score : harmonic mean of precision/recall
- mAP

Classification performance

- Accuracy
- Confusion matrix

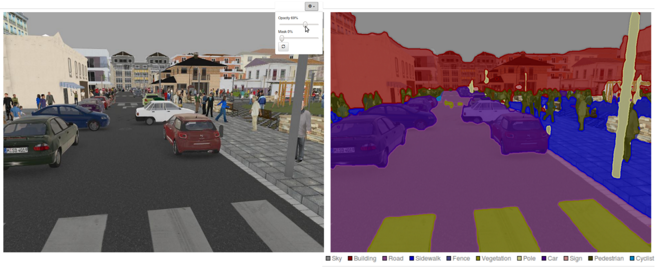


object detection



Style transfer

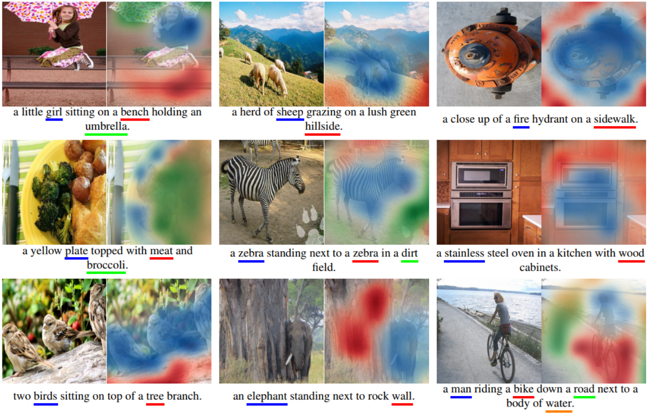
Segmentation



Deep dream

Style transfer

Image captioning



Ganimation