Deep Learning: Introduction

MLP, backpropagation, CNN

Nicolas Keriven
CNRS, IRISA, Rennes

ENSTA 2024

Outline

Table of Contents

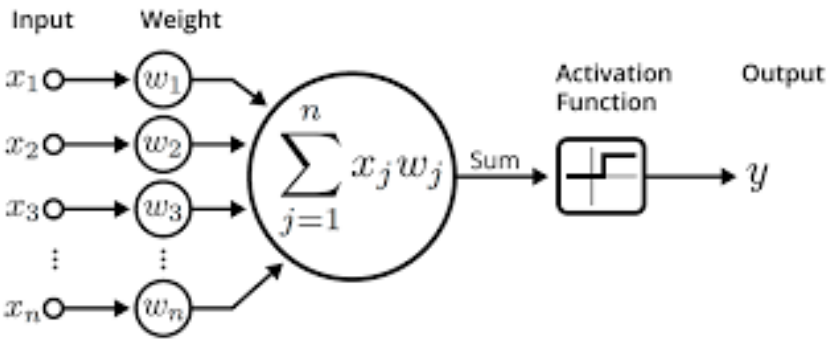
Perceptron : Basic principle

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

Perceptron : Basic principle

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

Rosenblatt's perceptron, 1957



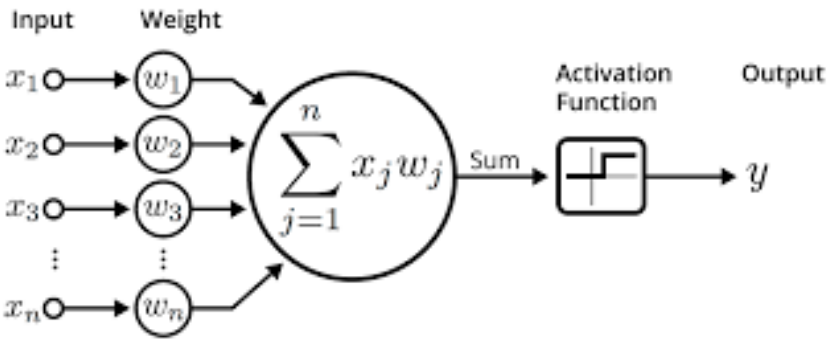
An illustration of an artificial neuron. Source: Becoming Human.

- Neural like structure, with a "single" unit. Frank Rosenblatt was a psychologist !
- Hugely influential, esp. since the coming back of neural networks.
- d "inputs" (dimension of the space), 1 output

Perceptron : Basic principle

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

Rosenblatt's perceptron, 1957



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

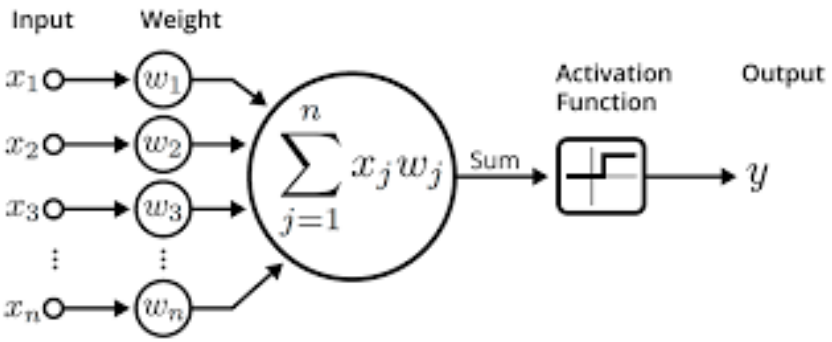
- Neural like structure, with a "single" unit. Frank Rosenblatt was a psychologist !
- Hugely influential, esp. since the coming back of neural networks.
- d "inputs" (dimension of the space), 1 output

- $x_i = [x_{i1}, \ldots, x_{id}]$ the input sample : one d-dimensional vector
- $w_1, \ldots, w_d$ the weights, or connecting weights

Perceptron : Basic principle

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

Rosenblatt's perceptron, 1957



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

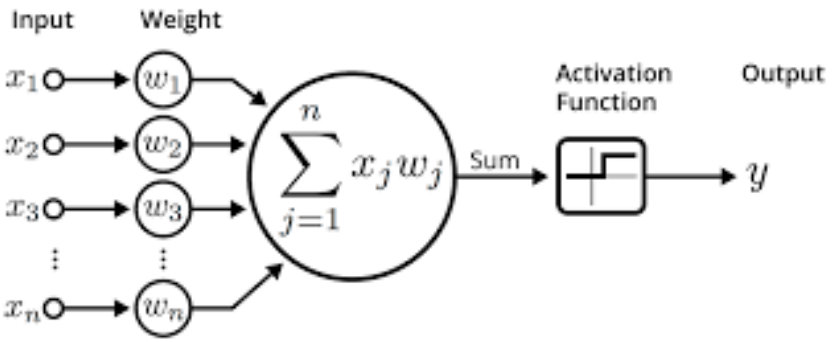
- Neural like structure, with a "single" unit. Frank Rosenblatt was a psychologist !
- Hugely influential, esp. since the coming back of neural networks.
- d "inputs" (dimension of the space), 1 output

- $x_i = [x_{i1}, \ldots, x_{id}]$ the input sample : one d-dimensional vector
- $w_1, \ldots, w_d$ the weights, or connecting weights
- $g(.)$ is the activation function

Perceptron : Basic principle

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

Rosenblatt's perceptron, 1957



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

- Neural like structure, with a "single" unit. Frank Rosenblatt was a psychologist !
- Hugely influential, esp. since the coming back of neural networks.
- d "inputs" (dimension of the space), 1 output

- $x_i = [x_{i1}, \ldots, x_{id}]$ the input sample : one d-dimensional vector
- $w_1, \ldots, w_d$ the weights, or connecting weights
- $g(.)$ is the activation function

$$a(x_i) = \sum_{j=1}^{d} w_j x_{ij} = w^\top x_i, \quad \hat{y}_i = f(x_i) = g(a(x_i))$$

This is just a linear model with an additional "activation" !

Remarks

- It is often convenient to introduce a bias to account for possible affine separating hyperplane : x is replaced by $[1, x_1, \ldots, x_p]$ and w by $[w_0, w_1, \ldots, w_d] \in \mathbb{R}^{d+1}$

Remarks

- It is often convenient to introduce a bias to account for possible affine separating hyperplane : x is replaced by $[1, x_1, \ldots, x_p]$ and w by $[w_0, w_1, \ldots, w_d] \in \mathbb{R}^{d+1}$
- Some simple activation functions :
  - for a (linear) regression problem $g(a) = a$

Remarks

- It is often convenient to introduce a bias to account for possible affine separating hyperplane : x is replaced by $[1, x_1, \ldots, x_p]$ and w by $[w_0, w_1, \ldots, w_d] \in \mathbb{R}^{d+1}$
- Some simple activation functions :
  - for a (linear) regression problem $g(a) = a$
  - for binary classification,

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

Remarks

- It is often convenient to introduce a bias to account for possible affine separating hyperplane : x is replaced by $[1, x_1, \ldots, x_p]$ and w by $[w_0, w_1, \ldots, w_d] \in \mathbb{R}^{d+1}$
- Some simple activation functions :
  - for a (linear) regression problem $g(a) = a$
  - for binary classification,

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

  - In order to predict the probability of x to be in a given class :

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

Sigmoidal, same expression than logistic function, but don't confuse loss function and activation function ! *We used max-likelihood for logistic regression. None of this here.*

Training the perceptron

As for other ML approaches, minimize the empirical risk, i.e. an averaged cost function : $\min_f \frac{1}{n} \sum_i L(f(x_i), y_i)$. This is done by using gradient descent.

Training the perceptron

As for other ML approaches, minimize the empirical risk, i.e. an averaged cost function : $\min_f \frac{1}{n} \sum_i L(f(x_i), y_i)$. This is done by using gradient descent.

Online learning

Originally, the perceptron is optimized by taking the examples one by one, which is similar to online learning (samples arrive in a stream) or Stochastic Gradient Descent (samples are taken randomly to decrease the computational cost). This is opposed to full-batch learning.

Training the perceptron

As for other ML approaches, minimize the empirical risk, i.e. an averaged cost function : $\min_f \frac{1}{n} \sum_i L(f(x_i), y_i)$. This is done by using gradient descent.

Online learning

Originally, the perceptron is optimized by taking the examples one by one, which is similar to online learning (samples arrive in a stream) or Stochastic Gradient Descent (samples are taken randomly to decrease the computational cost). This is opposed to full-batch learning. Each time a new pair (x_i, y_i) is received, all the weights are updated as :

$$w_j \leftarrow w_j - \nu \frac{\partial L(f(x_i), y_i)}{\partial w_j}$$

Training the perceptron

As for other ML approaches, minimize the empirical risk, i.e. an averaged cost function : $\min_f \frac{1}{n} \sum_i L(f(x_i), y_i)$. This is done by using gradient descent.

Online learning

Originally, the perceptron is optimized by taking the examples one by one, which is similar to online learning (samples arrive in a stream) or Stochastic Gradient Descent (samples are taken randomly to decrease the computational cost). This is opposed to full-batch learning. Each time a new pair (x_i, y_i) is received, all the weights are updated as :

$$w_j \leftarrow w_j - \nu \frac{\partial L(f(x_i), y_i)}{\partial w_j}$$

- ν is the learning rate. In practice, ν is often decreased when the risk is close to the minimum.
  - if ν is too large : possible instability
  - if ν is too small : slow convergence

Training the perceptron

As for other ML approaches, minimize the empirical risk, i.e. an averaged cost function : $\min_f \frac{1}{n} \sum_i L(f(x_i), y_i)$. This is done by using gradient descent.

Online learning

Originally, the perceptron is optimized by taking the examples one by one, which is similar to online learning (samples arrive in a stream) or Stochastic Gradient Descent (samples are taken randomly to decrease the computational cost). This is opposed to full-batch learning. Each time a new pair (x_i, y_i) is received, all the weights are updated as :

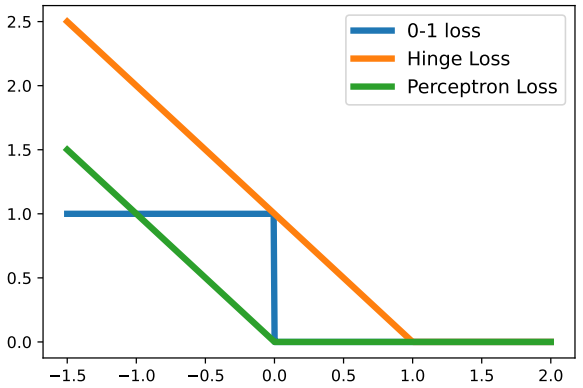
$$w_j \leftarrow w_j - \nu \frac{\partial L(f(x_i), y_i)}{\partial w_j}$$

- ν is the learning rate. In practice, ν is often decreased when the risk is close to the minimum.
  - if ν is too large : possible instability
  - if ν is too small : slow convergence
- A full cycle on the training set is an epoch. Many epochs may be performed on the whole training set.

Historical example : binary classifier

- $g(a) = a$ and (almost) hinge loss
  $L(y, \hat{y}) = \max(0, -y\hat{y})$

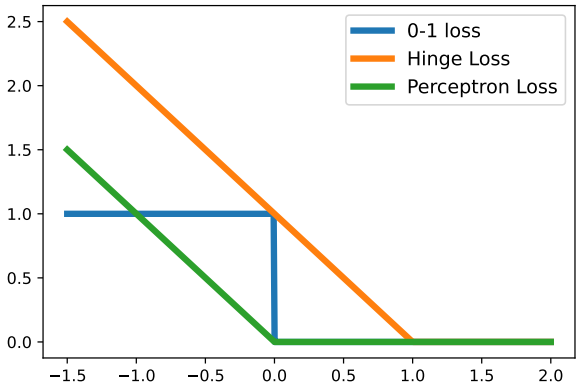
$$\Rightarrow w_j \leftarrow \begin{cases} w_j & \text{if } y_i w^T x_i > 0 \\ w_j + \nu y_i x_{ij} & \text{if } y_i w^T x_i \leq 0 \end{cases}$$



Historical example : binary classifier

- $g(a) = a$ and (almost) hinge loss
  $L(y, \hat{y}) = \max(0, -y\hat{y})$

$$\Rightarrow w_j \leftarrow \begin{cases} w_j & \text{if } y_i w^T x_i > 0 \\ w_j + \nu y_i x_{ij} & \text{if } y_i w^T x_i \leq 0 \end{cases}$$



Albert Novikov theorem, 1962

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

IF

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

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

Table of Contents

Multilayer perceptrons, Neural Nets

- Perceptrons are easy to train, but still (essentially) linear.

Multilayer perceptrons, Neural Nets

- Perceptrons are easy to train, but still (essentially) linear.
- To deal with non linear frontiers, we may apply several perceptrons sequentially, with non-linear activation function g.

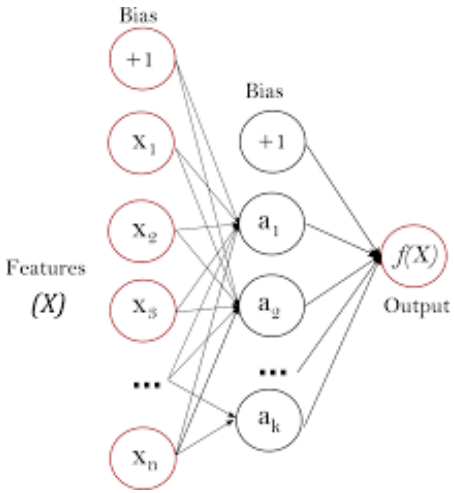
Multilayer perceptrons, Neural Nets

- Perceptrons are easy to train, but still (essentially) linear.
- To deal with non linear frontiers, we may apply several perceptrons sequentially, with non-linear activation function g.
- Each intermediate perceptron is a hidden layer
- The result is a feed forward network ("Neural Network", "deep neural network"...).

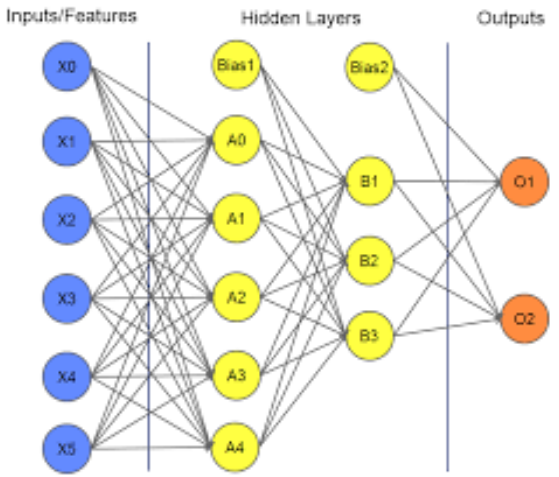
Multilayer perceptrons, Neural Nets

- Perceptrons are easy to train, but still (essentially) linear.
- To deal with non linear frontiers, we may apply several perceptrons sequentially, with non-linear activation function g.
- Each intermediate perceptron is a hidden layer
- The result is a feed forward network ("Neural Network", "deep neural network"...).

Single Hidden Layer Perceptron



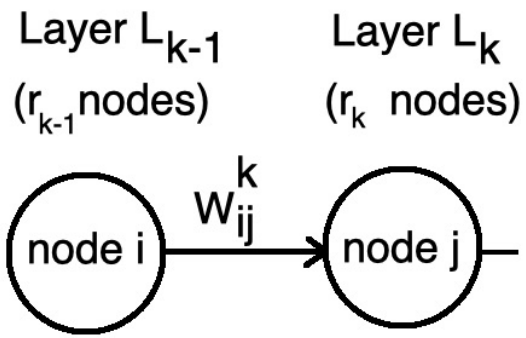
Multiple Hidden Layer Perceptron



It can be represented graphically : each edge is a scalar product associated to a coefficient that must be estimated. A "fully connected" layer is just a matrix-vector multiplication.

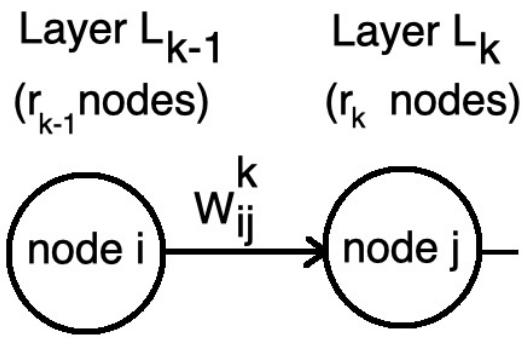
Notations

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



Notations

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

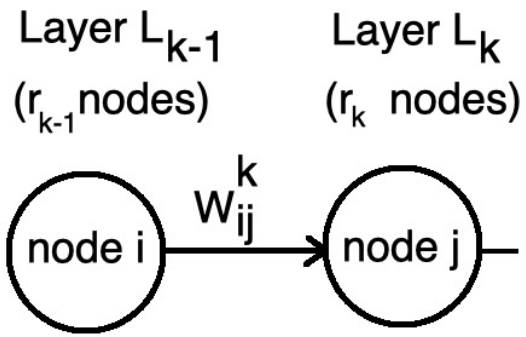


Then

- Input : $r_0 = p$ and $o_j^0 = x_{ij}$
- At layer $k = 1, \ldots, L$: $a_j^k = b_j^k + \sum_{i=1}^{r_{k-1}} w_{ij}^k o_i^{k-1}$ and $o_j^k = g(a_j^k)$
- Output : $f(x_i)_j = g_o(a_j^L)$

Notations

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



Then

- Input : $r_0 = p$ and $o_j^0 = x_{ij}$
- At layer $k = 1, \ldots, L$: $a_j^k = b_j^k + \sum_{i=1}^{r_{k-1}} w_{ij}^k o_i^{k-1}$ and $o_j^k = g(a_j^k)$
- Output : $f(x_i)_j = g_o(a_j^L)$

In vectorial form, this is just $o^k = g(W^k o^{k-1} + b^k)$ where g is applied pointwise and $o^k, b^k \in \mathbb{R}^{r_k}$, $W^k \in \mathbb{R}^{r_{k-1} \times r_k}$.

Table of Contents

Gradient backpropagation algorithm

Recall that our goal is to minimize the empirical risk (or loss)

$$F(\theta) = \frac{1}{n} \sum_i L(f_\theta(x_i), y_i)$$

with respect to the neural network f_θ weights (gathered in θ)

Gradient backpropagation algorithm

Recall that our goal is to minimize the empirical risk (or loss)

$$F(\theta) = \frac{1}{n} \sum_i L(f_\theta(x_i), y_i)$$

with respect to the neural network f_θ weights (gathered in θ)

(Stochastic) gradient descent is the workhorse of modern machine learning.
BACKPROPAGATION IS ITS *EXTREMELY EFFICIENT AND* ***PARALLELIZABLE***
INSTANCIATION (originally, for neural networks).

Gradient backpropagation algorithm

Recall that our goal is to minimize the empirical risk (or loss)

$$F(\theta) = \frac{1}{n} \sum_i L(f_\theta(x_i), y_i)$$

with respect to the neural network f_θ weights (gathered in θ)

(Stochastic) gradient descent is the workhorse of modern machine learning.
BACKPROPAGATION IS ITS *EXTREMELY EFFICIENT AND* ***PARALLELIZABLE***
INSTANCIATION (originally, for neural networks).

We have :

$$\frac{\partial F(\theta)}{\partial \theta} = \frac{1}{n} \sum_\ell \frac{\partial}{\partial \theta} L(f_\theta(x_\ell), y_\ell), \quad \text{and} \quad \frac{\partial}{\partial \theta} = \left[\frac{\partial}{\partial w_{ij}^k} \right]_{ijk}$$

For each weight/parameter w_{ij}^k and sample x_ℓ, we need to evaluate $\frac{\partial E}{\partial w_{ij}^k}$ for $E = E_\ell = L(f_\theta(x_\ell), y_\ell)$. Rk : w include the bias here

Backprop

There are many ways of "deriving" backpropagation equations, based on the CHAIN RULE : the differenciation of a composition of elementary operations.

Backprop

There are many ways of "deriving" backpropagation equations, based on the CHAIN RULE : the differenciation of a composition of elementary operations.

For $k \geq 1$ we write

$$\frac{\partial E}{\partial w_{ij}^k} = \underbrace{\frac{\partial E}{\partial o_j^k} \cdot \frac{\partial o_j^k}{\partial a_j^k}}_{\delta_j^k} \cdot \frac{\partial a_j^k}{\partial w_{ij}^k} \text{ with } a_j^k = b_j^k + \sum_{i=1}^{r_{k-1}} w_{ij}^k o_i^{k-1}, \quad o_j^k = g(a_j^k)$$

Backprop

There are many ways of "deriving" backpropagation equations, based on the CHAIN RULE : the differenciation of a composition of elementary operations.
For $k \geq 1$ we write

$$\frac{\partial E}{\partial w_{ij}^k} = \underbrace{\frac{\partial E}{\partial o_j^k} \cdot \frac{\partial o_j^k}{\partial a_j^k}}_{\delta_j^k} \cdot \frac{\partial a_j^k}{\partial w_{ij}^k} \text{ with } a_j^k = b_j^k + \sum_{i=1}^{r_{k-1}} w_{ij}^k o_i^{k-1}, \quad o_j^k = g(a_j^k)$$

We have easily $\frac{\partial o_j^k}{\partial a_j^k} = g'(a_j^k)$ (or g_o if $k = L$) and $\frac{\partial a_j^k}{\partial w_{ij}^k} = o_i^{k-1}$ ($= x_{\ell i}$ if $k = 1$), it remains the first term.

Backprop

There are many ways of "deriving" backpropagation equations, based on the CHAIN RULE : the differenciation of a composition of elementary operations.
For $k \geq 1$ we write

$$\frac{\partial E}{\partial w_{ij}^k} = \underbrace{\frac{\partial E}{\partial o_j^k} \cdot \frac{\partial o_j^k}{\partial a_j^k}}_{\delta_j^k} \cdot \frac{\partial a_j^k}{\partial w_{ij}^k} \text{ with } a_j^k = b_j^k + \sum_{i=1}^{r_{k-1}} w_{ij}^k o_i^{k-1}, \quad o_j^k = g(a_j^k)$$

We have easily $\frac{\partial o_j^k}{\partial a_j^k} = g'(a_j^k)$ (or g_o if $k = L$) and $\frac{\partial a_j^k}{\partial w_{ij}^k} = o_i^{k-1}$ ($= x_{\ell i}$ if $k = 1$), it remains the first term.

If $k = L$

Taking $r_L = 1$ for simplicity, $E = L(\hat{y}_\ell, y_\ell) = L(o_1^L, y_\ell)$, so

$$\frac{\partial E}{\partial o_1^L} = L'(\hat{y}_\ell, y_\ell) \Rightarrow \delta_1^L = L'(\hat{y}_\ell, y_\ell) g_o'(a_1^L)$$

Only requires to know the prediction $\hat{y}_\ell$: a forward pass is required before a backward pass (that is, just applying the network on x_ℓ).

Backprop - cont'd

If $k < L$

We use the "total derivative rule" : given a multivariate function $F(f_1, \ldots, f_r)$ and r functions $f_1, \ldots, f_r$,

$$f(x) = F(f_1(x), \ldots, f_r(x)) \Rightarrow \frac{\partial f(x)}{\partial x} = \sum_i \frac{\partial F}{\partial f_i} \cdot f_i'(x)$$

Backprop - cont'd

If $k < L$

We use the "total derivative rule" : given a multivariate function $F(f_1, \ldots, f_r)$ and r functions $f_1, \ldots, f_r$,

$$f(x) = F(f_1(x), \ldots, f_r(x)) \Rightarrow \frac{\partial f(x)}{\partial x} = \sum_i \frac{\partial F}{\partial f_i} \cdot f_i'(x)$$

Here E depends on o_j^k through all the a_i^{k+1}, hence

$$\frac{\partial E}{\partial o_j^k} = \sum_{i=1}^{r_{k+1}} \frac{\partial E}{\partial a_i^{k+1}} \frac{\partial a_i^{k+1}}{\partial o_j^k} = \sum_{i=1}^{r_{k+1}} \underbrace{\frac{\partial E}{\partial o_i^{k+1}} \cdot \frac{\partial o_i^{k+1}}{\partial a_i^{k+1}}}_{\delta_i^{k+1}} \cdot \underbrace{\frac{\partial a_i^{k+1}}{\partial o_j^k}}_{w_{ji}^{k+1}}$$

Backprop - cont'd

If $k < L$

We use the "total derivative rule" : given a multivariate function $F(f_1, \ldots, f_r)$ and r functions $f_1, \ldots, f_r$,

$$f(x) = F(f_1(x), \ldots, f_r(x)) \Rightarrow \frac{\partial f(x)}{\partial x} = \sum_i \frac{\partial F}{\partial f_i} \cdot f_i'(x)$$

Here E depends on o_j^k through all the a_i^{k+1}, hence

$$\frac{\partial E}{\partial o_j^k} = \sum_{i=1}^{r_{k+1}} \frac{\partial E}{\partial a_i^{k+1}} \frac{\partial a_i^{k+1}}{\partial o_j^k} = \sum_{i=1}^{r_{k+1}} \underbrace{\frac{\partial E}{\partial o_i^{k+1}} \cdot \frac{\partial o_i^{k+1}}{\partial a_i^{k+1}}}_{\delta_i^{k+1}} \cdot \underbrace{\frac{\partial a_i^{k+1}}{\partial o_j^k}}_{w_{ji}^{k+1}}$$

- δ_j^k can be computed if all the δ_i^{k+1} are known.
- Backpropagation : evaluate the gradients by decreasing order of layers $k = L, L-1, \ldots$ ("backward" propagation)

BACKPROPAGATION ALGO, Main equations

The gradients are given by

$$\frac{\partial L(f_\theta(x_\ell), y_\ell)}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}$$

BACKPROPAGATION ALGO, Main equations

The gradients are given by

$$\frac{\partial L(f_\theta(x_\ell), y_\ell)}{\partial w_{ij}^k} = \delta_j^k \, o_i^{k-1}$$

where :

- the successive o_i^k are stored during the forward pass to compute $f_\theta(x_\ell) = \hat{y}_\ell$

BACKPROPAGATION ALGO, Main equations

The gradients are given by

$$\frac{\partial L(f_\theta(x_\ell), y_\ell)}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}$$

where :

- the successive o_i^k are stored during the forward pass to compute $f_\theta(x_\ell) = \hat{y}_\ell$
- The errors are propagated backward :
  - $\delta_1^L = g_o'(a_1^L) L'(\hat{y}_\ell, y_\ell)$
  - $\delta_j^k = g'(a_j^k) \sum_{i=1}^{r_{k+1}} w_{ji}^{k+1} \delta_i^{k+1}$

Note that if $L(\hat{y}_\ell, y_\ell)$ reach its minimum in $\hat{y}_\ell$ and $L'(\hat{y}_\ell, y_\ell) = 0$, then all the gradients are zero !

BACKPROPAGATION ALGO, Main equations

The gradients are given by

$$\frac{\partial L(f_\theta(x_\ell), y_\ell)}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}$$

where :

- the successive o_i^k are stored during the forward pass to compute $f_\theta(x_\ell) = \hat{y}_\ell$
- The errors are propagated backward :
  - $\delta_1^L = g_o'(a_1^L) L'(\hat{y}_\ell, y_\ell)$
  - $\delta_j^k = g'(a_j^k) \sum_{i=1}^{r_{k+1}} w_{ji}^{k+1} \delta_i^{k+1}$

Note that if $L(\hat{y}_\ell, y_\ell)$ reach its minimum in $\hat{y}_\ell$ and $L'(\hat{y}_\ell, y_\ell) = 0$, then all the gradients are zero !

Then the gradient corresponding to b samples x_ℓ, y_ℓ are averaged (full batch $b = n$, online $b = 1$, SGD...), and a gradient step is performed

$$w_{ij}^k \leftarrow w_{ij}^k - \nu \left(\frac{1}{b} \sum_{\ell=1}^{b} \frac{\partial L(f_\theta(x_\ell), y_\ell)}{\partial w_{ij}^k} \right)$$

Backprop - remark and modern implementation

"Remarks" (foundations of deep learning !)

- "Just" an efficient application of the chain rule to evaluate the gradients of functions made of simple functions applied sequentially

Backprop - remark and modern implementation

"Remarks" (foundations of deep learning !)

- "Just" an efficient application of the chain rule to evaluate the gradients of functions made of simple functions applied sequentially
- Csq : the gradient of such functions is essentially **not more expensive** to compute than the function itself (absolutely foundational)

Backprop - remark and modern implementation

"Remarks" (foundations of deep learning !)

- "Just" an efficient application of the chain rule to evaluate the gradients of functions made of simple functions applied sequentially
- Csq : the gradient of such functions is essentially **not more expensive** to compute than the function itself (absolutely foundational)
- Highly parallelizable on GPU (over samples, layer dimension, etc.), key for modern architecture

Backprop - remark and modern implementation

"Remarks" (foundations of deep learning !)

- "Just" an efficient application of the chain rule to evaluate the gradients of functions made of simple functions applied sequentially
- Csq : the gradient of such functions is essentially **not more expensive** to compute than the function itself (absolutely foundational)
- Highly parallelizable on GPU (over samples, layer dimension, etc.), key for modern architecture

Backprop today...

- is automatic in Pytorch and Tensorflow ! Just implement the forward pass using only Pytorch/Tensorflow operations and elements, then call `.backward()` (or equivalent)
- The use of GPU is completely transparent. The practical sessions will be done on CPU, the code almost does not change for GPU.

Backprop - remark and modern implementation

"Remarks" (foundations of deep learning !)

- "Just" an efficient application of the chain rule to evaluate the gradients of functions made of simple functions applied sequentially
- Csq : the gradient of such functions is essentially **not more expensive** to compute than the function itself (absolutely foundational)
- Highly parallelizable on GPU (over samples, layer dimension, etc.), key for modern architecture

Backprop today...

- is automatic in Pytorch and Tensorflow ! Just implement the forward pass using only Pytorch/Tensorflow operations and elements, then call `.backward()` (or equivalent)
- The use of GPU is completely transparent. The practical sessions will be done on CPU, the code almost does not change for GPU.
- Can be applied to any functions ! (... that are composition of elementary operations). Consequences far beyond deep learning...

Pros and cons

- Empirical risk minimization for multilayer perceptron (or any deep net) is an ill-posed and ill-conditioned NON CONVEX problem.
- Hyperparameters (weights initialization values, learning rate, choice of $g()$, L, r_k...) all influence the result !
- To avoid saturation effects of node outputs (either to 0 or to 1), l_2 regularization (or other regularization) of $L(f(x), y)$ may be applied on θ.

Vanishing/exploding gradient problem

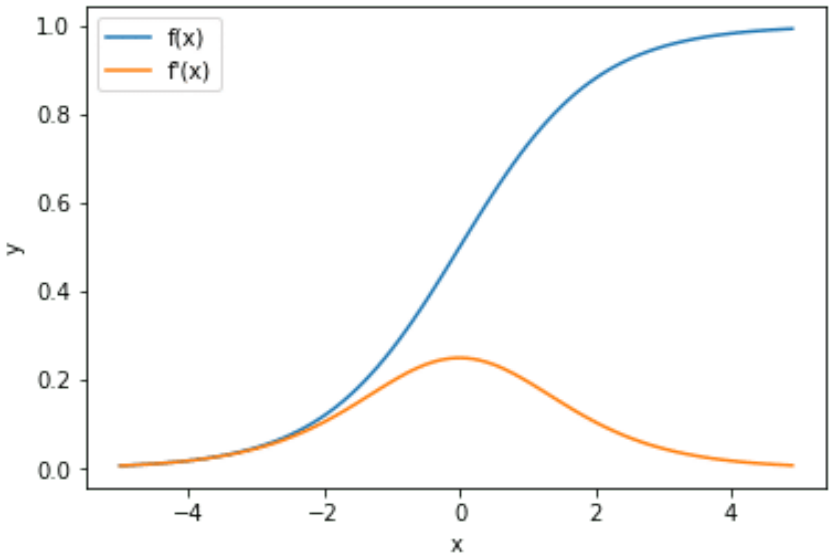
- Since they are **multiplied** L times, gradient values in the first hidden layers often takes either too large (gradient explosion) or too low values (vanishing gradient, leading to slow down learning convergence).

Vanishing/exploding gradient problem

- Since they are **multiplied** L times, gradient values in the first hidden layers often takes either too large (gradient explosion) or too low values (vanishing gradient, leading to slow down learning convergence).
- Consider a 3-layers DNN with sigmoid activation. The chain rule leads to equation that look (roughly) like

$$\frac{\partial L}{\partial U} = \frac{\partial L_3}{\partial out_3}\frac{\partial out_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial h_0}\frac{\partial h_0}{\partial U}$$

In this product of gradients, each involves to evaluate the gradient of a sigmoid function : may be very small (which occurs if the output is close to either 0 or 1), the product becomes exponentially small. Other activation function may lead to exploding gradients instead.



ResNets

- Important solution : **Residual Networks** (ResNet, He et al. 2015), which includes *skip connections* and fit the residual signal :

$$x^{\ell} = x^{\ell-1} + \mathcal{F}(x^{\ell-1})$$

such that $\frac{\partial x^{\ell}}{\partial x^{\ell-1}} = 1 + \nabla\mathcal{F}(x^{\ell-1})$

ResNets

- Important solution : **Residual Networks** (ResNet, He et al. 2015), which includes *skip connections* and fit the residual signal :

$$x^{\ell} = x^{\ell-1} + \mathcal{F}(x^{\ell-1})$$

such that $\frac{\partial x^{\ell}}{\partial x^{\ell-1}} = 1 + \nabla \mathcal{F}(x^{\ell-1})$

- All modern DNNs include skip connections. ResNets are still (!) state-of-the-art in several tasks

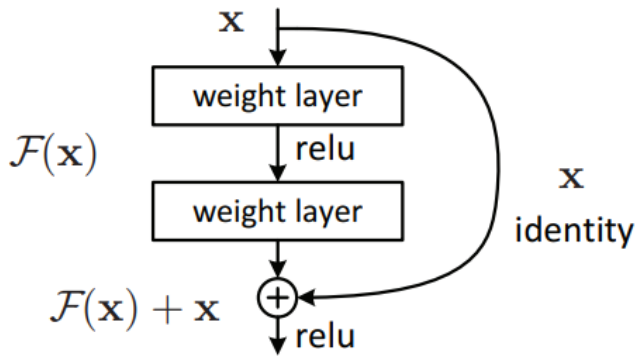


Table of Contents

Stochastic Gradient method

At each iteration, rather than computing

$$\nabla_\theta F(\theta) = \nabla_\theta \left(\frac{1}{n} \sum_{\ell=1}^{n} L(f_\theta(x_\ell), y_\ell) \right) = \frac{1}{n} \sum_{\ell=1}^{n} \nabla_\theta E_\ell$$

Stochastic Gradient Descent (SGD) randomly samples ℓ at uniform and computes $\nabla_\theta E_\ell$ instead :

Stochastic Gradient method

At each iteration, rather than computing

$$\nabla_\theta F(\theta) = \nabla_\theta \left(\frac{1}{n} \sum_{\ell=1}^{n} L(f_\theta(x_\ell), y_\ell) \right) = \frac{1}{n} \sum_{\ell=1}^{n} \nabla_\theta E_\ell$$

Stochastic Gradient Descent (SGD) randomly samples ℓ at uniform and computes $\nabla_\theta E_\ell$ instead : SGD uses ∇E_ℓ as an unbiased estimator of $\nabla F(\theta)$:

$$\mathbb{E}_\ell \left[\nabla_\theta E_\ell \right] = \frac{1}{n} \sum_{i=1}^{n} \nabla E_i = \nabla F(\theta)$$

Stochastic Gradient method

At each iteration, rather than computing

$$\nabla_\theta F(\theta) = \nabla_\theta \left(\frac{1}{n} \sum_{\ell=1}^{n} L(f_\theta(x_\ell), y_\ell) \right) = \frac{1}{n} \sum_{\ell=1}^{n} \nabla_\theta E_\ell$$

Stochastic Gradient Descent (SGD) randomly samples ℓ at uniform and computes $\nabla_\theta E_\ell$ instead : SGD uses ∇E_ℓ as an unbiased estimator of $\nabla F(\theta)$:

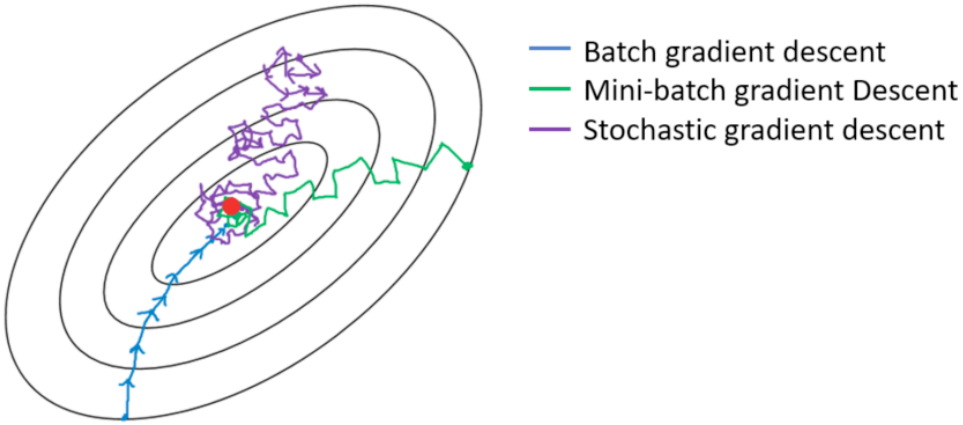
$$\mathbb{E}_\ell \left[\nabla_\theta E_\ell \right] = \frac{1}{n} \sum_{i=1}^{n} \nabla E_i = \nabla F(\theta)$$

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

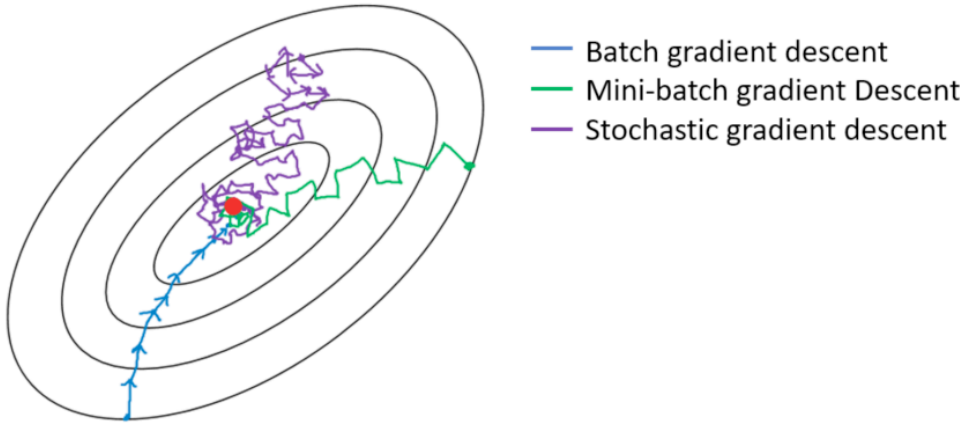
$$\nabla E_\mathcal{B} = \frac{1}{|\mathcal{B}|} \sum_{\ell \in \mathcal{B}} \nabla E_\ell$$

update θ as

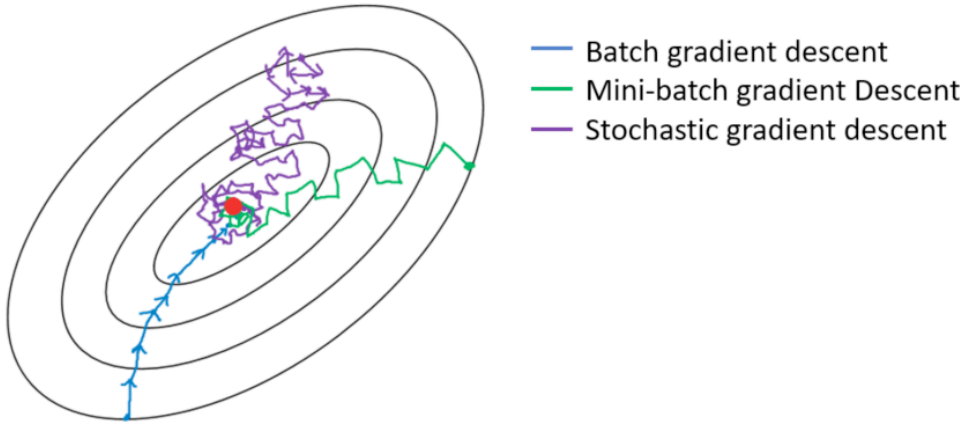
$$\theta := \theta - \nu \nabla E_\mathcal{B}$$



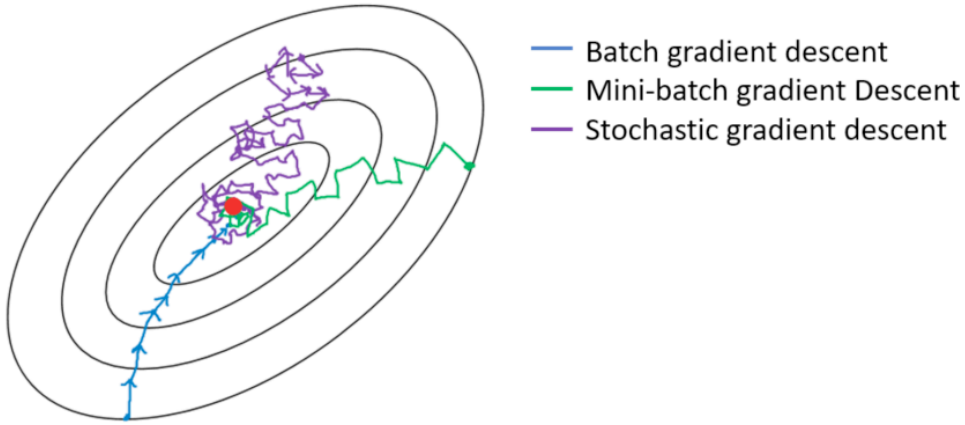
- SGD may be so close to the true gradient $\nabla_\theta F$ that a small number of iterations will find useful solutions.



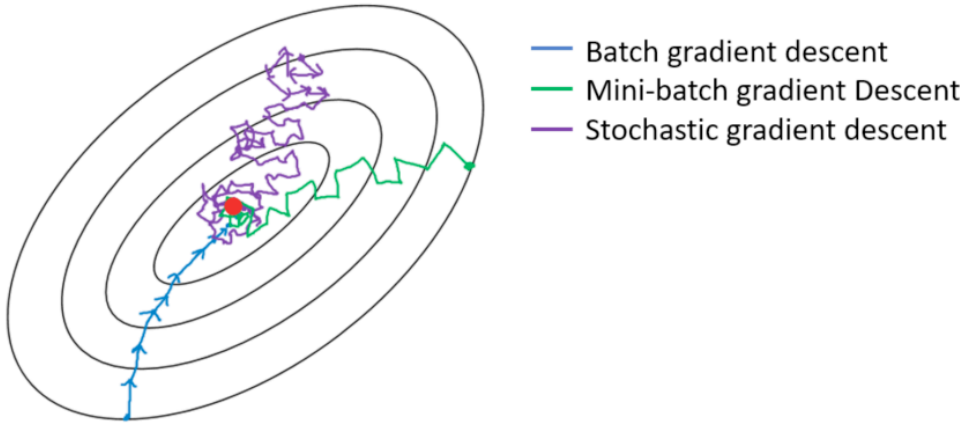
- SGD may be so close to the true gradient $\nabla_\theta F$ that a small number of iterations will find useful solutions.
- The per-iteration computational cost is $\mathcal{O}(|\mathcal{B}|)$. Thus, when the batch size is small, the computational cost at each iteration is light. However, computations are parallelized over samples in a batch. So, with a high-end GPU, a high batch size results in a *faster* epoch.



- SGD may be so close to the true gradient $\nabla_\theta F$ that a small number of iterations will find useful solutions.
- The per-iteration computational cost is $\mathcal{O}(|\mathcal{B}|)$. Thus, when the batch size is small, the computational cost at each iteration is light. However, computations are parallelized over samples in a batch. So, with a high-end GPU, a high batch size results in a *faster* epoch.
- But : SGD can be considered as offering a regularization effect especially when the mini-batch size is small due to the randomness and noise in the mini-batch sampling. Maybe be better than full-batch ! (remember we deal with NON-CONVEX optim !)



- SGD may be so close to the true gradient $\nabla_\theta F$ that a small number of iterations will find useful solutions.
- The per-iteration computational cost is $\mathcal{O}(|\mathcal{B}|)$. Thus, when the batch size is small, the computational cost at each iteration is light. However, computations are parallelized over samples in a batch. So, with a high-end GPU, a high batch size results in a *faster* epoch.
- But : SGD can be considered as offering a regularization effect especially when the mini-batch size is small due to the randomness and noise in the mini-batch sampling. Maybe be better than full-batch ! (remember we deal with NON-CONVEX optim !)
- Globally, the batch size in an important hyperparameter, whose effects are not often predictable, and still only partially understood in theory.



- SGD may be so close to the true gradient $\nabla_\theta F$ that a small number of iterations will find useful solutions.
- The per-iteration computational cost is $\mathcal{O}(|\mathcal{B}|)$. Thus, when the batch size is small, the computational cost at each iteration is light. However, computations are parallelized over samples in a batch. So, with a high-end GPU, a high batch size results in a *faster* epoch.
- But : SGD can be considered as offering a regularization effect especially when the mini-batch size is small due to the randomness and noise in the mini-batch sampling. Maybe be better than full-batch ! (remember we deal with NON-CONVEX optim !)
- Globally, the batch size in an important hyperparameter, whose effects are not often predictable, and still only partially understood in theory.
- Many variants of SGD : with momentum, etc. Popular ones include Adam ("just use Adam"), RMSProp,...

Backprop in Practice

Y LeCun

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

Table of Contents

Brief insights into Convolutional Neural networks (CNN)

"Universal approximation theorem" (Cybenko 89, Hornik 89, Pinkus 99)

MLP with at least one hidden layer and sufficiently many parameters can approximate any continuous function (and thus learn "any" model).

Brief insights into Convolutional Neural networks (CNN)

"Universal approximation theorem" (Cybenko 89, Hornik 89, Pinkus 99)

MLP with at least one hidden layer and sufficiently many parameters can approximate any continuous function (and thus learn "any" model).

- ☞ Why would we need anything else ? Literally hundreds of different deep learning models...
- ☞ Why do we need to be "deep" ? One hidden layer is sufficient...

Brief insights into Convolutional Neural networks (CNN)

"Universal approximation theorem" (Cybenko 89, Hornik 89, Pinkus 99)

MLP with at least one hidden layer and sufficiently many parameters can approximate any continuous function (and thus learn "any" model).

- ☞ Why would we need anything else ? Literally hundreds of different deep learning models...
- ☞ Why do we need to be "deep" ? One hidden layer is sufficient...

Motivations for other models

- MLP can require a lot of parameters. For example a 256x256 image over 3 channels and 1000 nodes → more than 200 million parameters ! This is not only a memory issue : this may lead to overfitting and degrade generalization.

Brief insights into Convolutional Neural networks (CNN)

"Universal approximation theorem" (Cybenko 89, Hornik 89, Pinkus 99)

MLP with at least one hidden layer and sufficiently many parameters can approximate any continuous function (and thus learn "any" model).

- ☞ Why would we need anything else ? Literally hundreds of different deep learning models...
- ☞ Why do we need to be "deep" ? One hidden layer is sufficient...

Motivations for other models

- MLP can require a lot of parameters. For example a 256x256 image over 3 channels and 1000 nodes → more than 200 million parameters ! This is not only a memory issue : this may lead to overfitting and degrade generalization.
- For reasons still unclear, being "deep" often (but not always) improve performance at every level (but training may become tricky).

Brief insights into Convolutional Neural networks (CNN)

"Universal approximation theorem" (Cybenko 89, Hornik 89, Pinkus 99)

MLP with at least one hidden layer and sufficiently many parameters can approximate any continuous function (and thus learn "any" model).

- ☞ Why would we need anything else ? Literally hundreds of different deep learning models...
- ☞ Why do we need to be "deep" ? One hidden layer is sufficient...

Motivations for other models

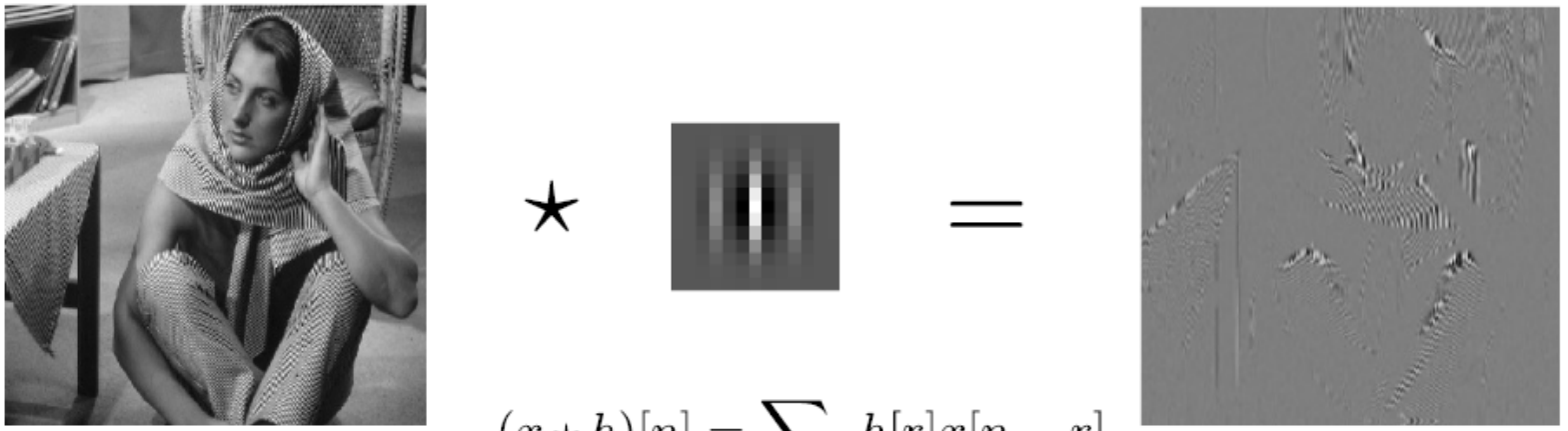
- MLP can require a lot of parameters. For example a 256x256 image over 3 channels and 1000 nodes → more than 200 million parameters ! This is not only a memory issue : this may lead to overfitting and degrade generalization.
- For reasons still unclear, being "deep" often (but not always) improve performance at every level (but training may become tricky).
- MLP totally ignore potentially existing structure in the data. This is essential : **deep learning is not automatic !**

CNN : historic origin

- Convolutional Neural Networks (CNN) come from image processing (mostly).

CNN : historic origin

- Convolutional Neural Networks (CNN) come from image processing (mostly).
- Since the 80s - 90s, the heart of image processing are convolutions, aka pattern matching.



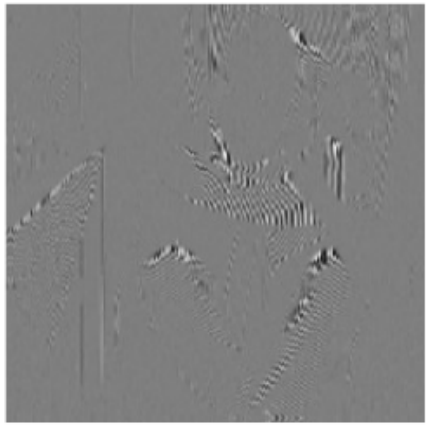
$$(x \star h)[n] = \sum_r h[r]x[n-r]$$

CNN : historic origin

- Convolutional Neural Networks (CNN) come from image processing (mostly).
- Since the 80s - 90s, the heart of image processing are convolutions, aka pattern matching.
  - how much the image locally correlate with a small kernel (pattern, patch...)
  - Ex : Fourier Transform, wavelets...

$\star$

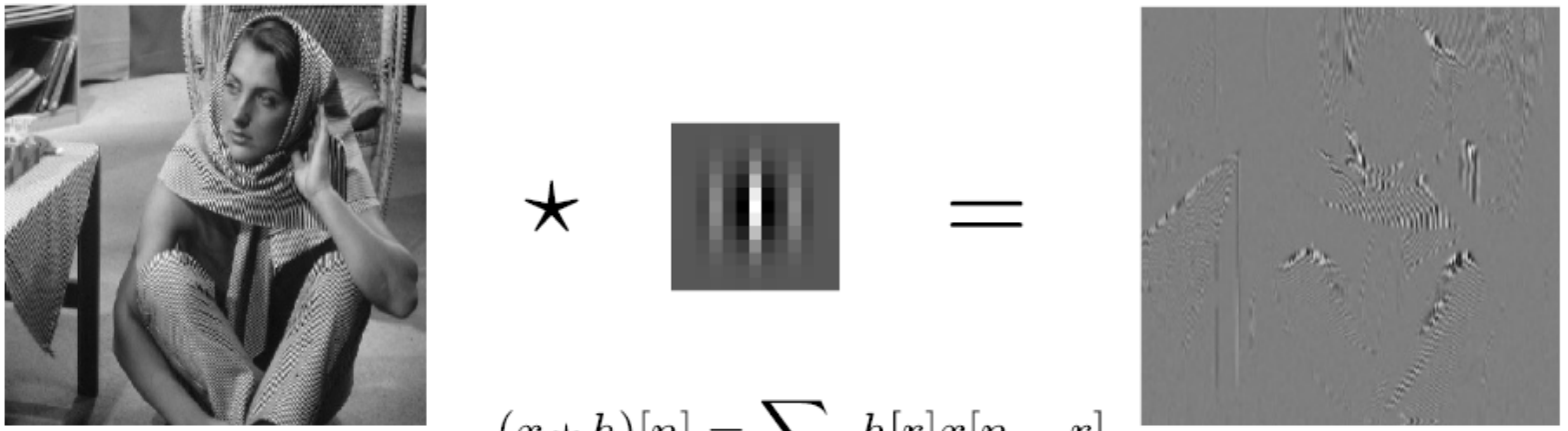
$=$



$$(x \star h)[n] = \sum_r h[r]x[n-r]$$

CNN : historic origin

- Convolutional Neural Networks (CNN) come from image processing (mostly).
- Since the 80s - 90s, the heart of image processing are convolutions, aka pattern matching.
  - how much the image locally correlate with a small kernel (pattern, patch...)
  - Ex : Fourier Transform, wavelets...



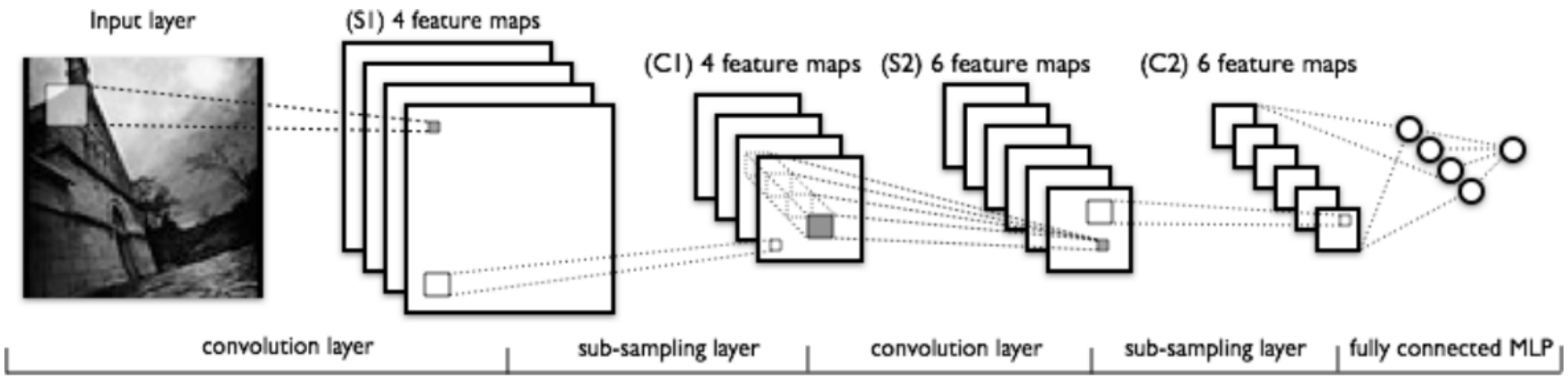
$$(x \star h)[n] = \sum_r h[r]x[n-r]$$

But :

- Impossible to try every pattern !
- Patterns have a hierarchical structure : small patterns are "combined" to make bigger objects, etc.

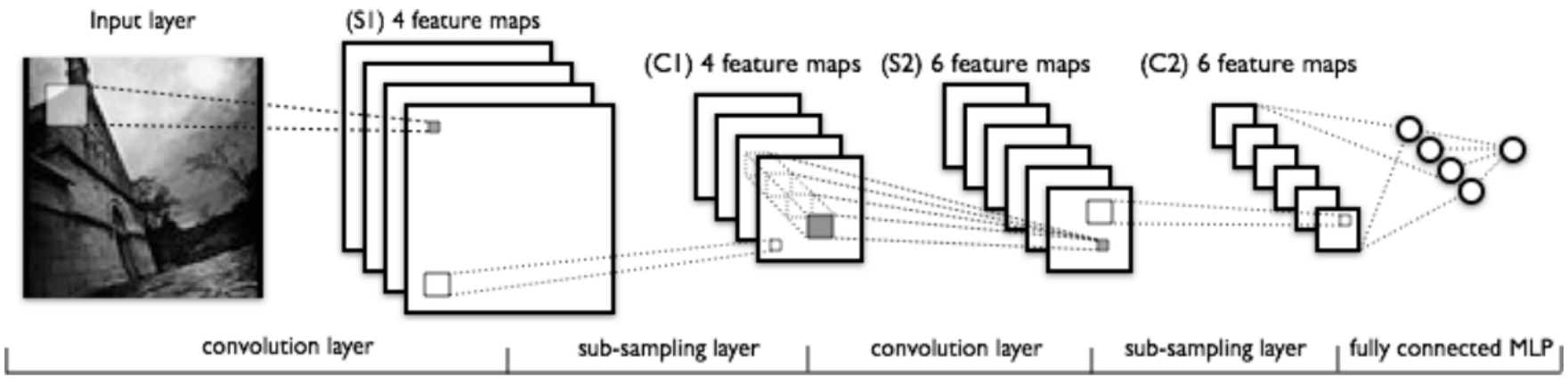
Ingredients of a CNN

- Convolution filters : the convolution kernels are learned (unlike wavelets)



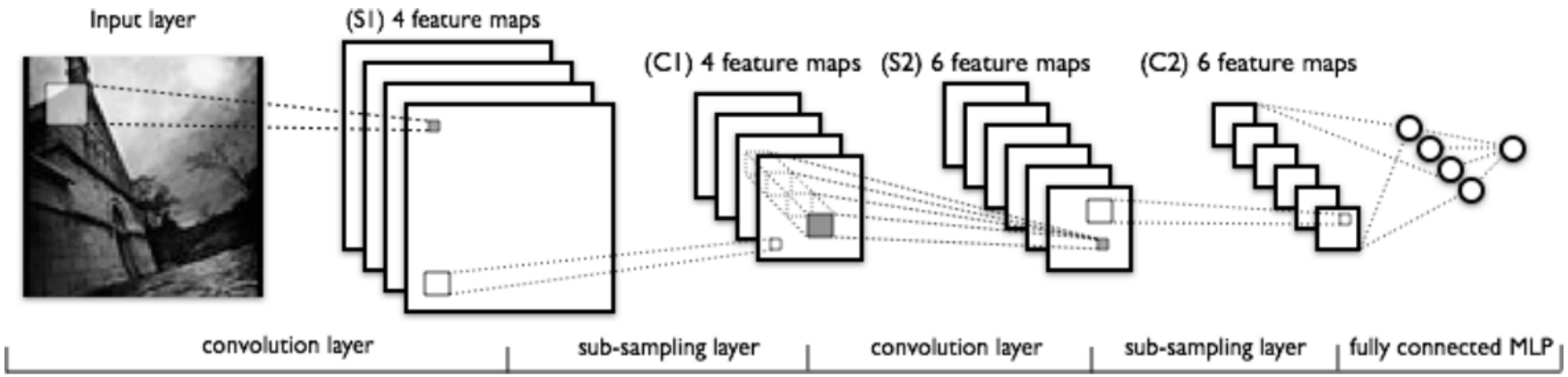
Ingredients of a CNN

- Convolution filters : the convolution kernels are learned (unlike wavelets)
- Non-linearity : without it, stacking convolution would still be linear !



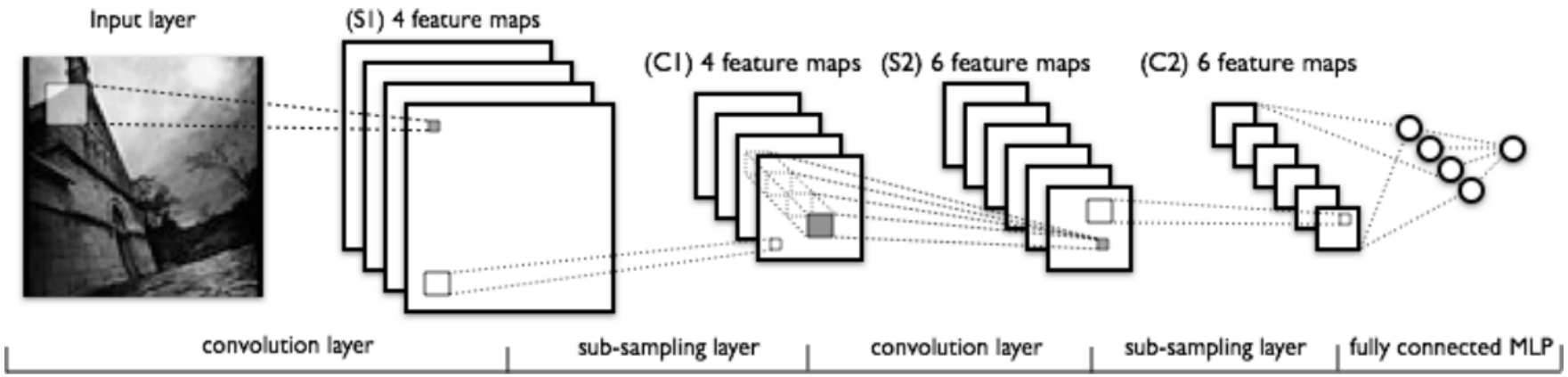
Ingredients of a CNN

- Convolution filters : the convolution kernels are learned (unlike wavelets)
- Non-linearity : without it, stacking convolution would still be linear !
- Pooling (downsampling, subsampling...) = local aggregation : reduce dimension, make the model hierarchical



Ingredients of a CNN

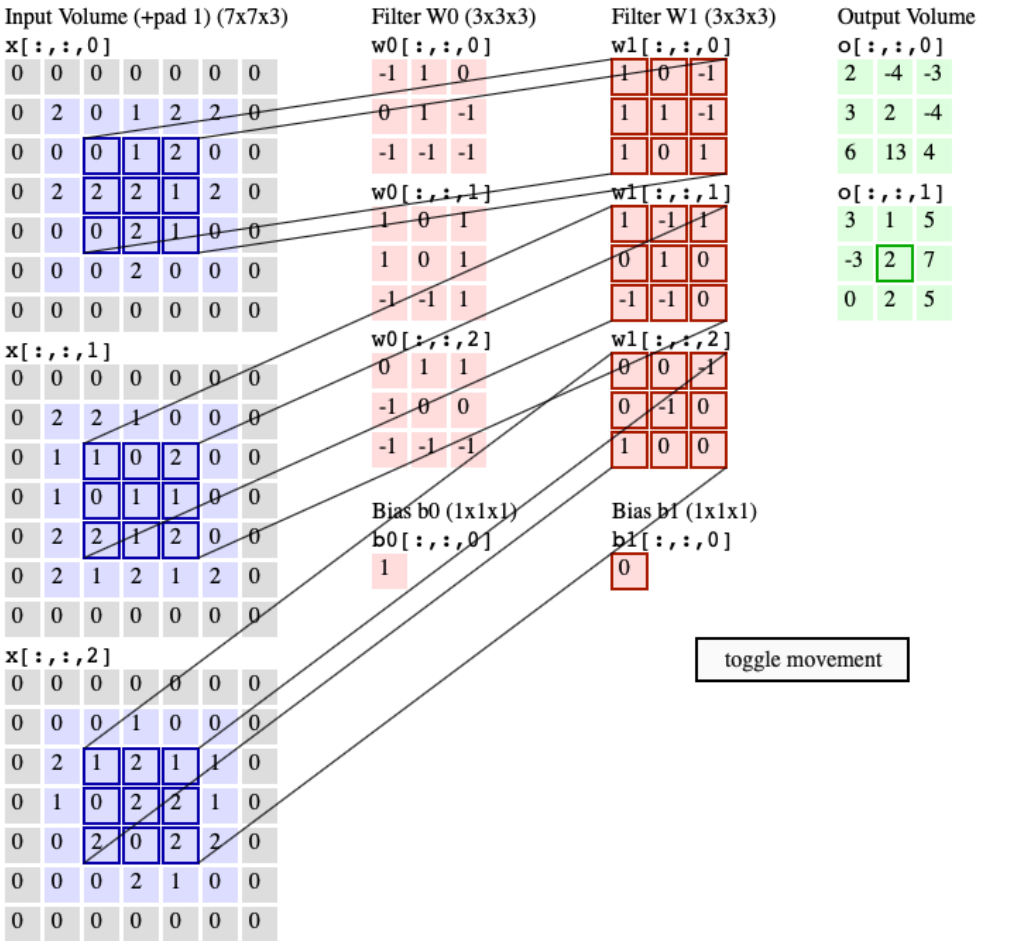
- Convolution filters : the convolution kernels are learned (unlike wavelets)
- Non-linearity : without it, stacking convolution would still be linear !
- Pooling (downsampling, subsampling...) = local aggregation : reduce dimension, make the model hierarchical



- Similar to an MLP, but with significant zeroed weights and weight-sharing (instead of dense parameter matrices, use block matrices with same, structured blocks)

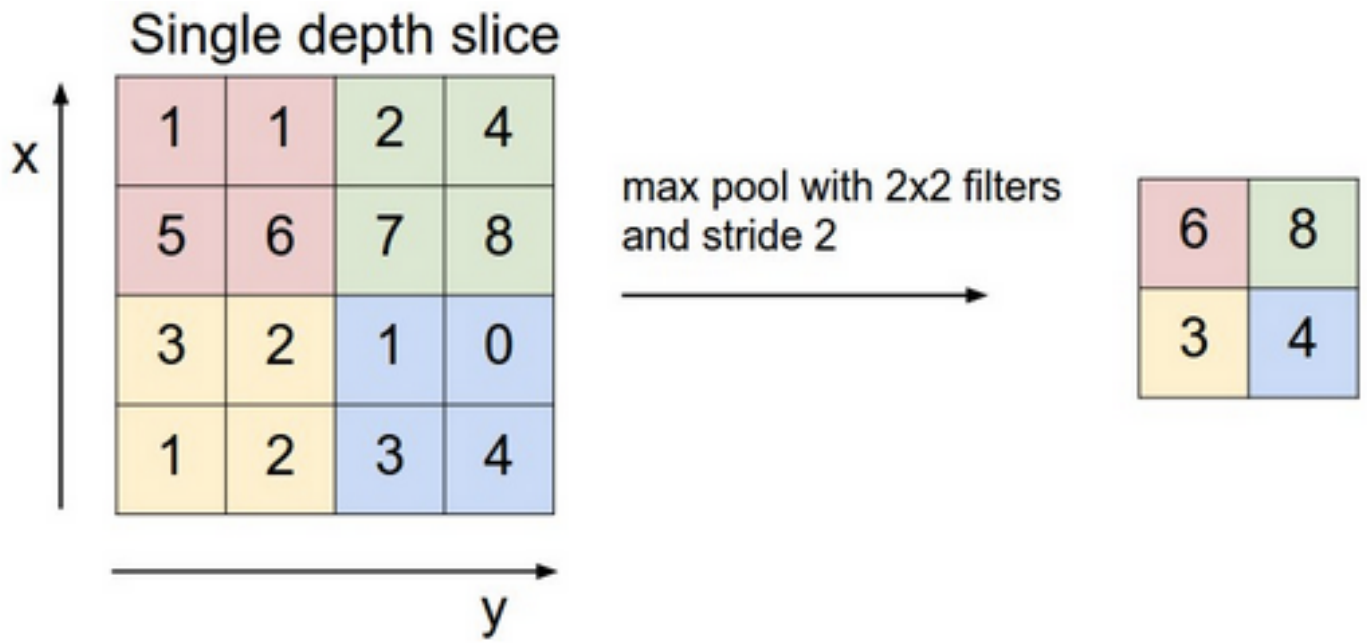
CNN Convolutive Layer example

RGB Channels Filter 1 Filter 2 Output Stack



Pooling / Downsampling within CNNs

Example : Max Pooling



- Only the locations on the image that shows the strongest correlation to each feature (the max value) are preserved, and those max values combine to form a lower-dimensional space
- Other examples : mean pooling, softmax...

CNNs : remark

- Depth is important to make the model hierarchical : it is impossible to enumerate all objects, but small patterns are combined to learn bigger objects

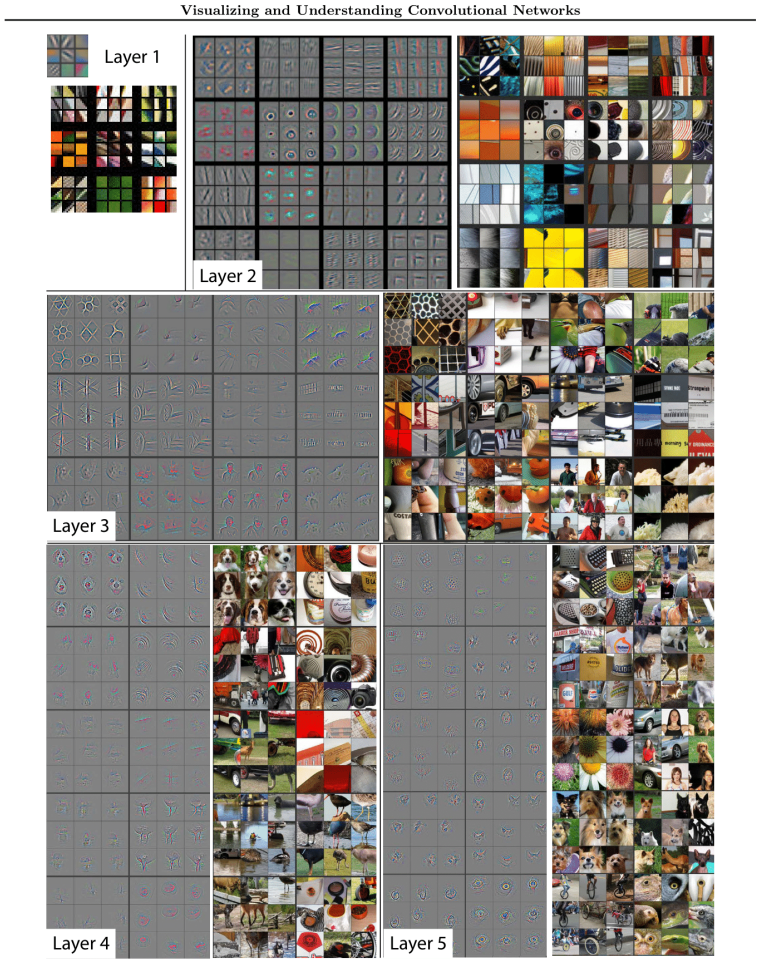
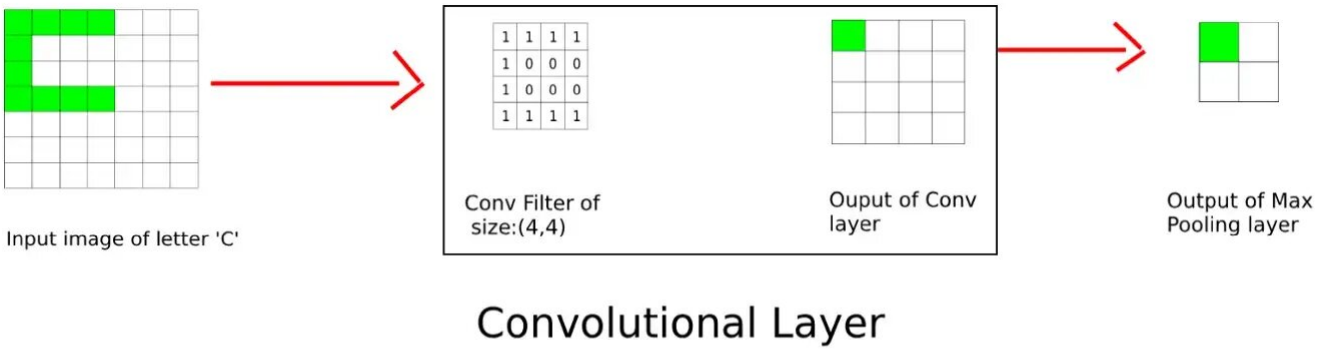


Figure 2. Visualization of features in a fully trained model. For layers 2-5 we show the top 9 activations in a random subset of feature maps across the validation data, projected down to pixel space using our deconvolutional network approach. Our reconstructions are *not* samples from the model: they are reconstructed patterns from the validation set that cause high activations in a given feature map. For each feature map we also show the corresponding image patches. Note: (i) the the strong grouping within each feature map, (ii) greater invariance at higher layers and (iii) exaggeration of discriminative parts of the image, e.g. eyes and noses of dogs (layer 4, row 1, cols 1). Best viewed in electronic form.

CNNs : remark

- Due to convolution+pooling, CNNs are (approximately) translation-invariant : moving the object around does not change the output $h(\tau(x)) = h(x)$.



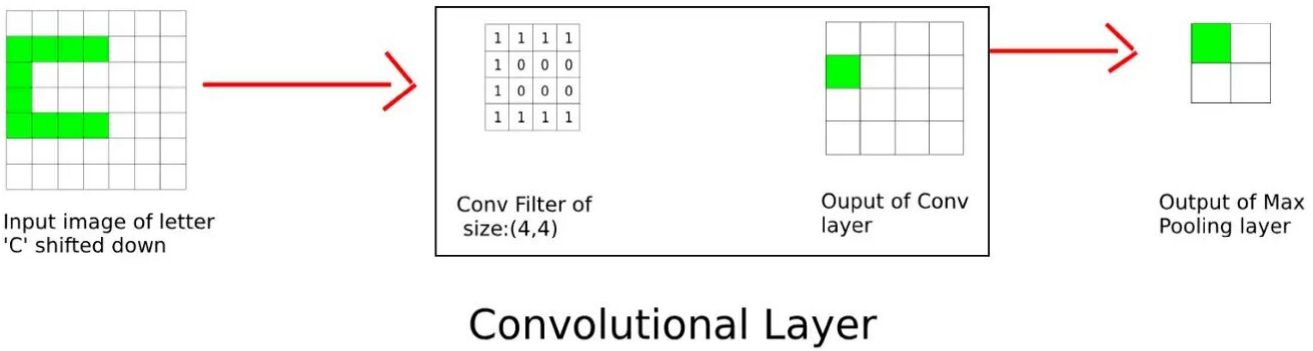


Table of Contents

Table of Contents

AutoEncoder

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

- Autoencoders are Unsupervised Neural Networks, designed for representation learning and/or dimension reduction

AutoEncoder

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

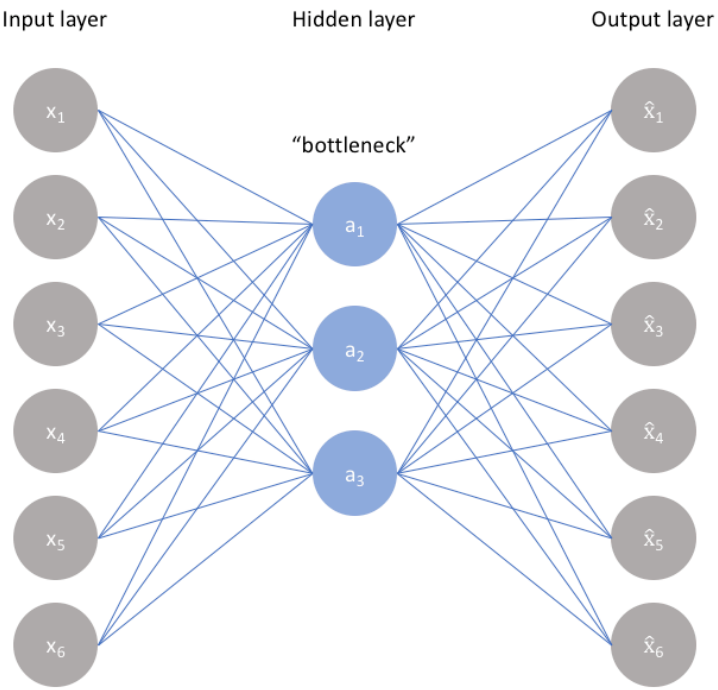
- Autoencoders are Unsupervised Neural Networks, designed for representation learning and/or dimension reduction
- Two parts : an encoder $f_\theta : x \to z$, and a decoder $g_\pi : z \to \tilde{x} \approx x$

AutoEncoder

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

- Autoencoders are Unsupervised Neural Networks, designed for representation learning and/or dimension reduction
- Two parts : an encoder $f_\theta : x \to z$, and a decoder $g_\pi : z \to \tilde{x} \approx x$
- main idea : impose a bottleneck in the network to compressed knowledge representation of the input.

Rk : This assumes that the data are structured (like images !), otherwise such compression will be very difficult if not impossible without loosing much information.

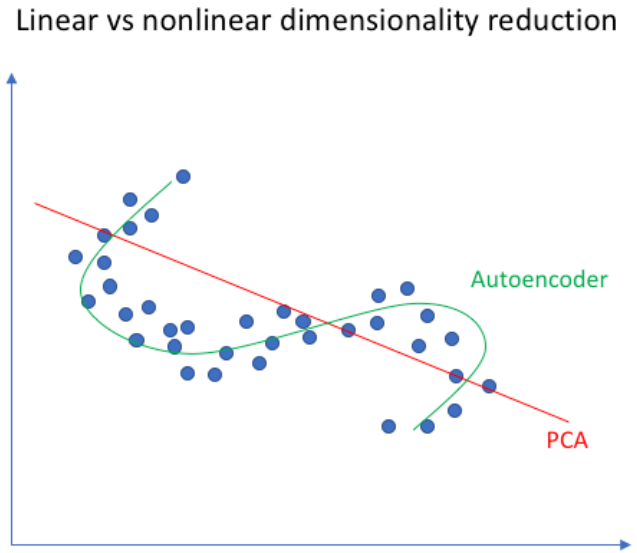


AutoEncoder principle

$\Rightarrow$ Formulate the problem as a supervised learning problem whose output is $\{x_\ell\}$

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

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

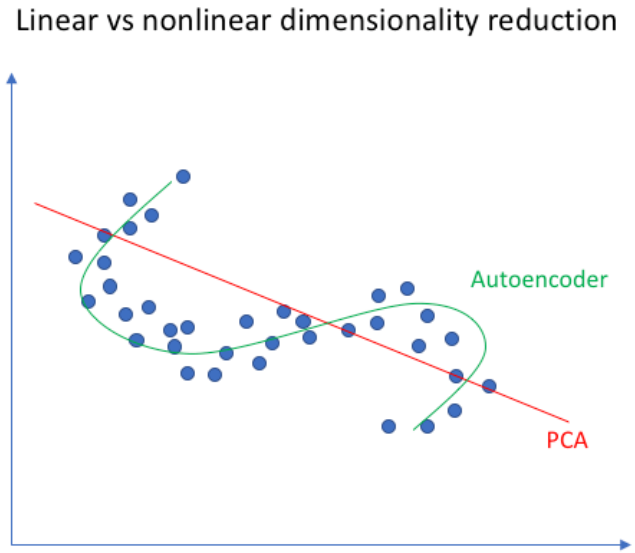


AutoEncoder principle

$\Rightarrow$ Formulate the problem as a supervised learning problem whose output is $\{x_\ell\}$

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

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

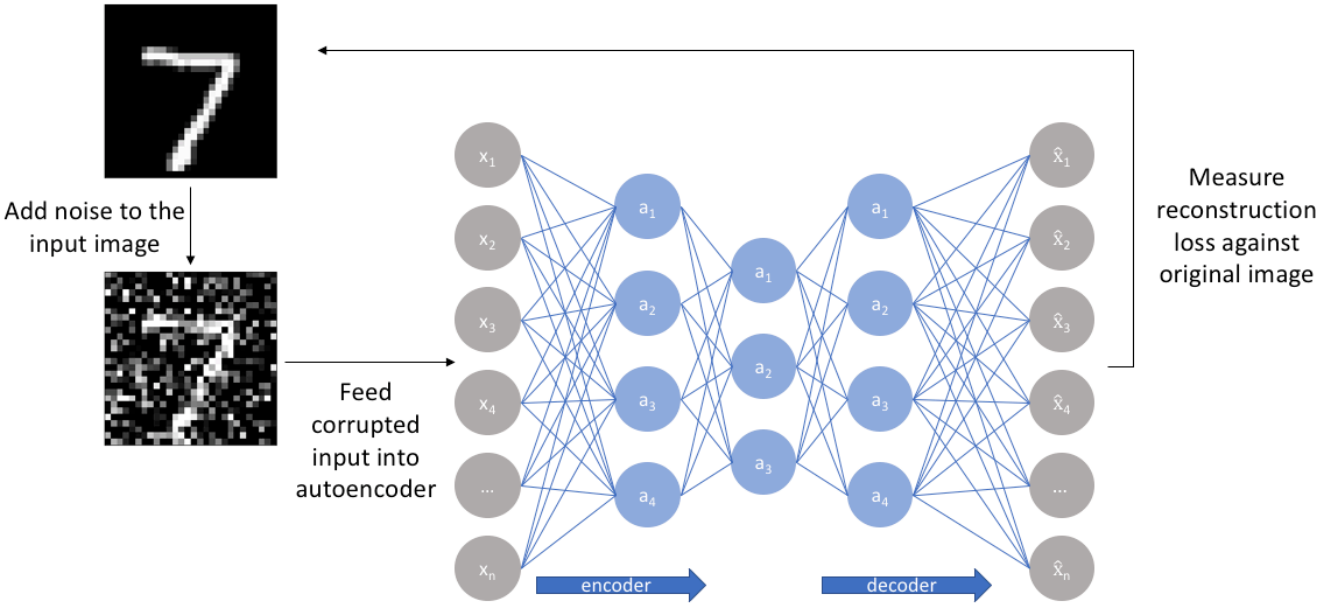


The AutoEncoder must be :

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

This requires to regularize the loss function of the form $L(f(x), x) + \textit{regularization}$

Example of application : AE for denoising



The low-dimensional representation has a regularizing effect (like wavelet thresholding).

Sparse AE

How many nodes ?

- If too many nodes, the AE may be capable of learning a way to simply memorize the data.

Sparse AE

How many nodes ?

- If too many nodes, the AE may be capable of learning a way to simply memorize the data.

- Taking inspiration from ℓ_1 sparse regularization (like LASSO) : keep the number of nodes in hidden layers quite large, but regularize the loss function by penalizing activations within a layer. For layer k :

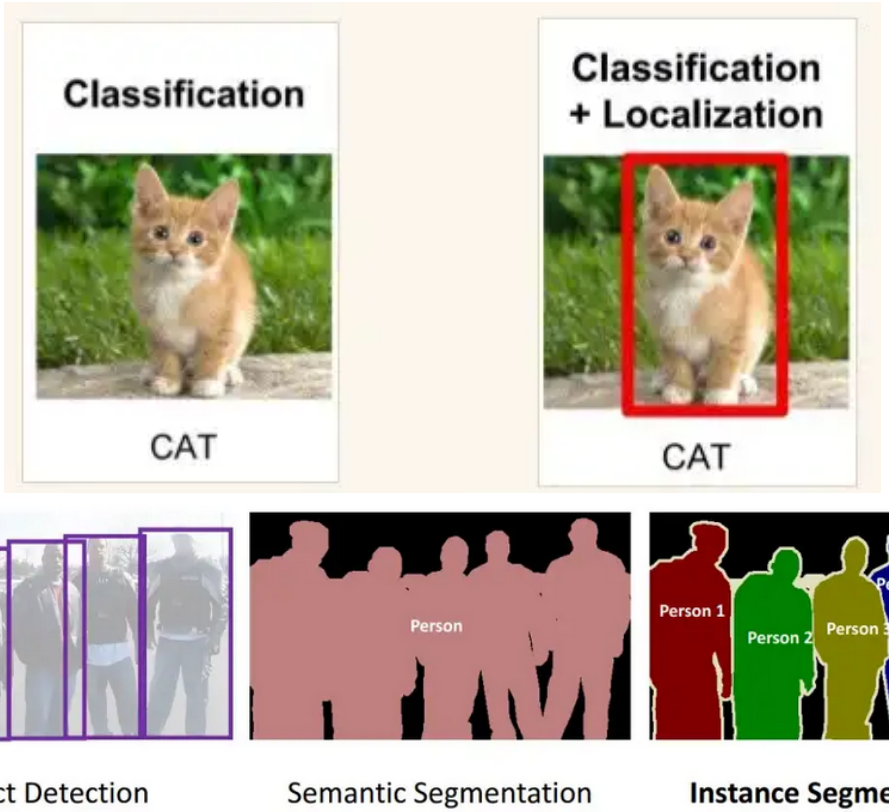
$$L(f(x), x) + \lambda \sum_{j=1}^{r_k} |o_j^k|$$

Table of Contents

U-Net

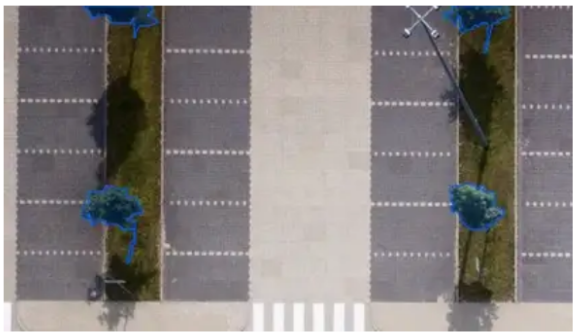
https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47

- U-Net is a popular architecture for image-to-image tasks such as localization, segmentation...

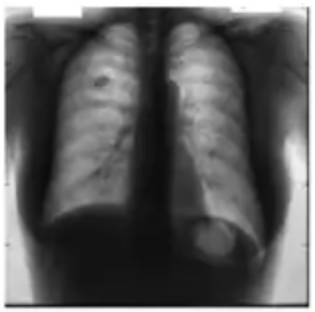


Why U-net ?

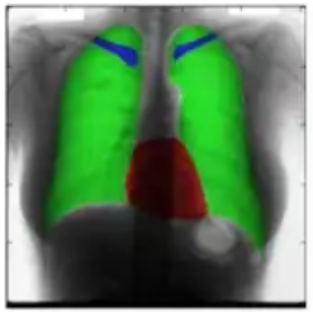
- Why not CNNs ?



Source: https://blog.playment.io/semantic-segmentation/



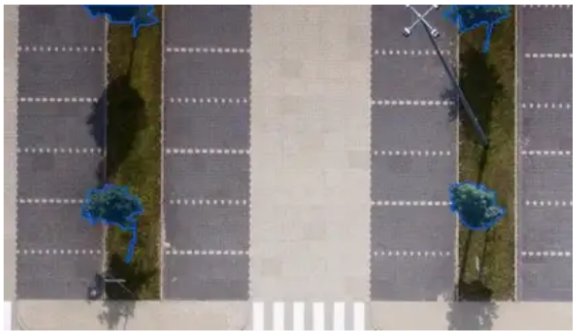
Input Image



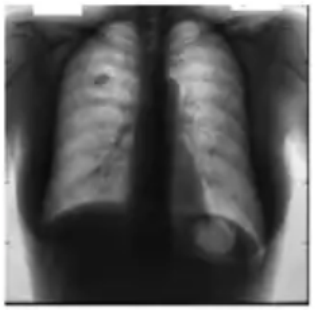
Segmented Image

Why U-net ?

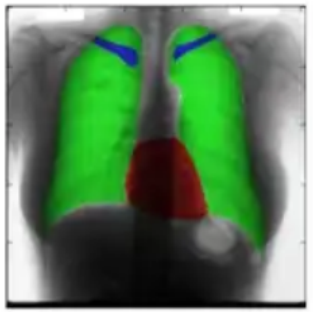
- Why not CNNs ? They include pooling



Source: https://blog.playment.io/semantic-segmentation/



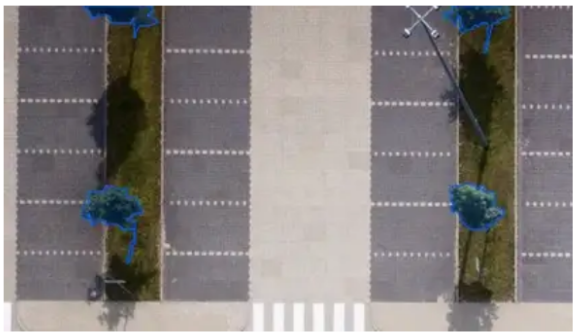
Input Image



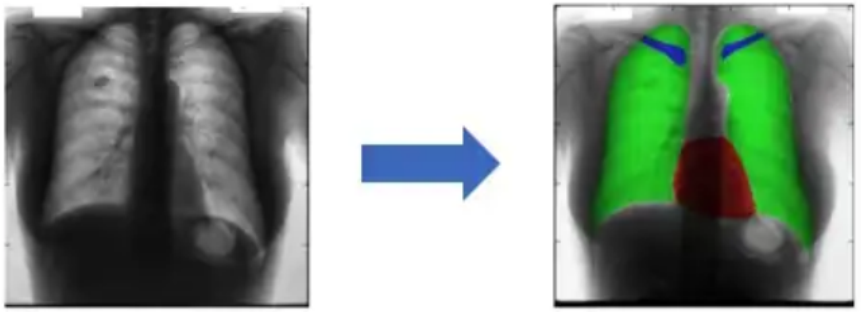
Segmented Image

Why U-net ?

- Why not CNNs ? They include pooling
- Why not regular AE ?

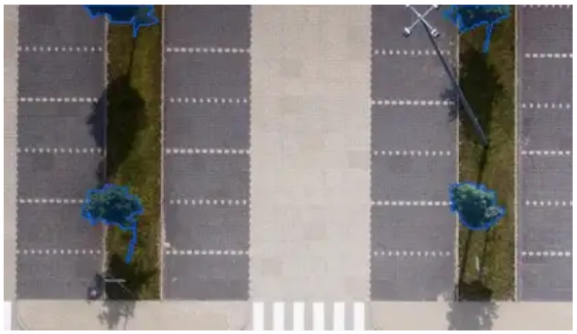


Source: https://blog.playment.io/semantic-segmentation/

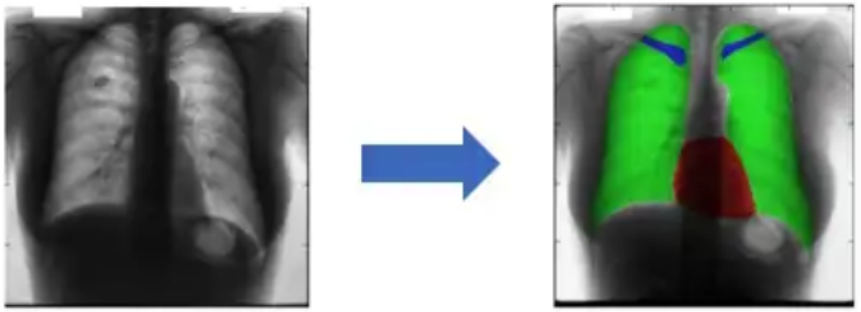


Why U-net ?

- Why not CNNs ? They include pooling
- Why not regular AE ? They are made to reconstruct the same image, and do not detect and localize high-level structure

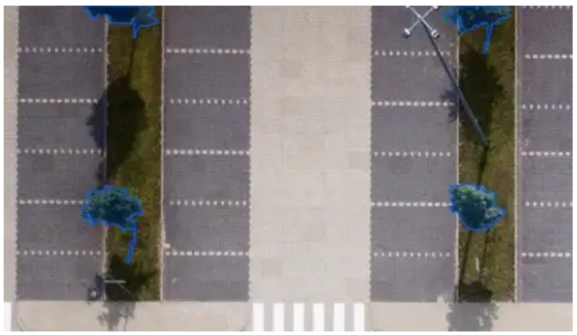


Source: https://blog.playment.io/semantic-segmentation/

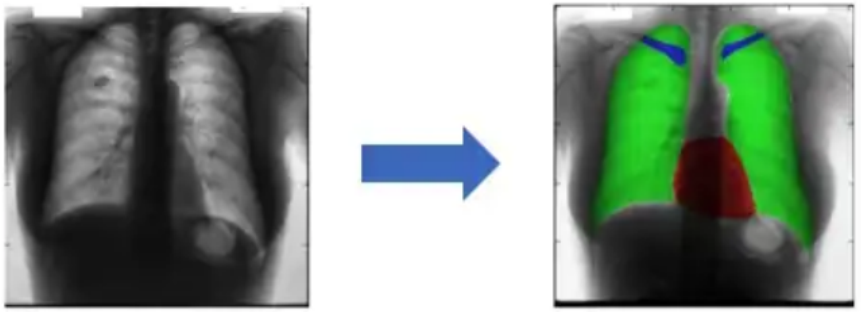


Why U-net ?

- Why not CNNs ? They include pooling
- Why not regular AE ? They are made to reconstruct the same image, and do not detect and localize high-level structure
- "Combine" the two ? U-Net



Source: https://blog.playment.io/semantic-segmentation/



- The pooling is important to detect WHAT is in the image, but loses WHERE it is since the resolution is crushed
- There are ways to upscale and restore the resolution (similar to AE), but the WHERE information is still lost

- The pooling is important to detect WHAT is in the image, but loses WHERE it is since the resolution is crushed
- There are ways to upscale and restore the resolution (similar to AE), but the WHERE information is still lost
- Idea of U-Net : include additional connections at each resolution scales between the encoder and decoder. The final architecture ressembles a “U”.

- The pooling is important to detect WHAT is in the image, but loses WHERE it is since the resolution is crushed
- There are ways to upscale and restore the resolution (similar to AE), but the WHERE information is still lost
- Idea of U-Net : include additional connections at each resolution scales between the encoder and decoder. The final architecture ressembles a "U".

At each scale, the output of the corresponding encoder layer is simply concatenated to the output of the corresponding decoder layer, before being passed to the next decoder layer. *Disarmingly simple !*

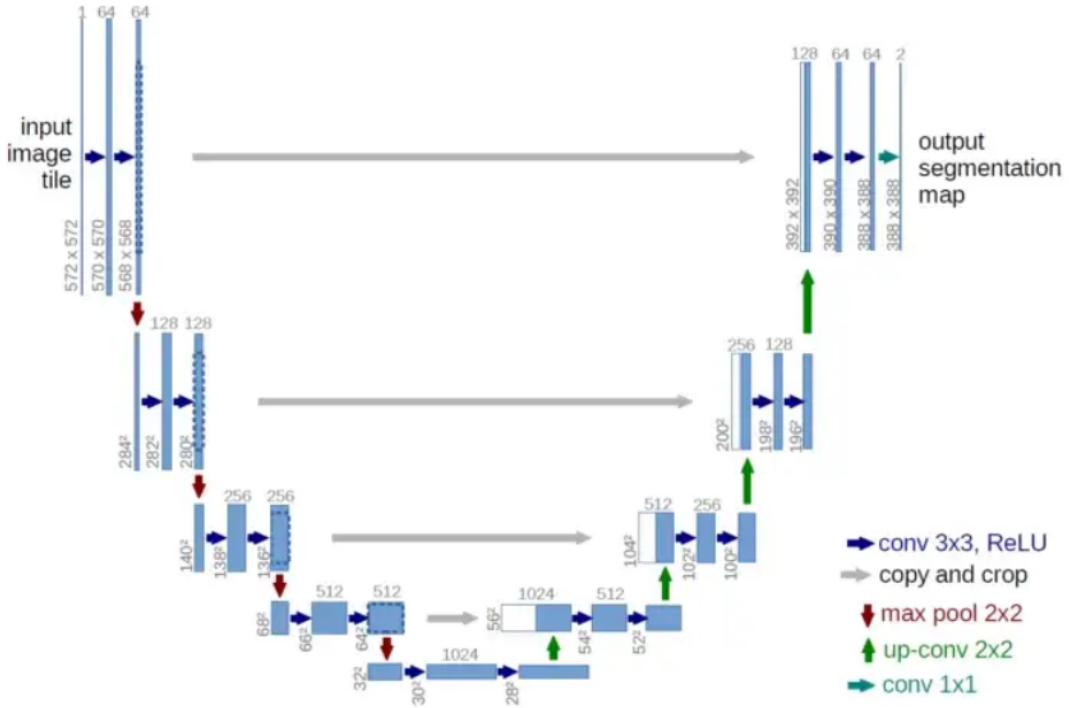


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.