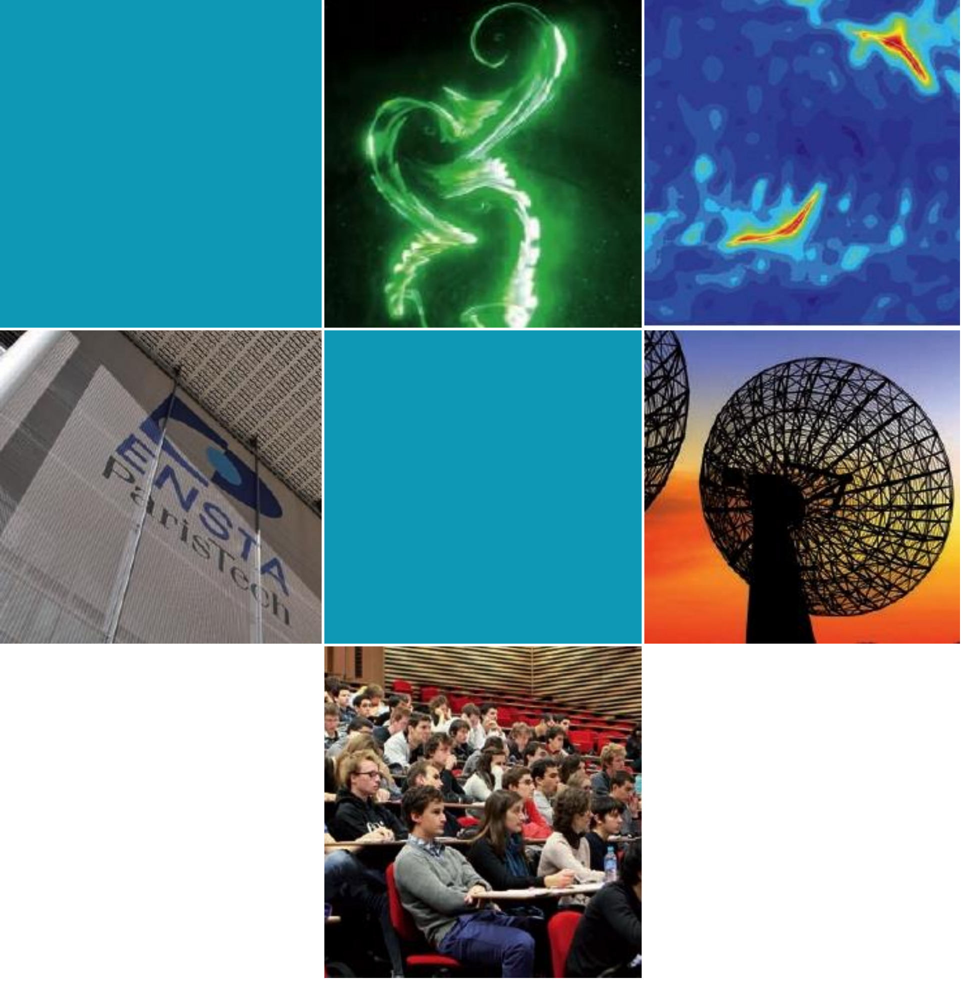


Deep learning for computer vision

Cours ENSTA Paris
5RO13 – 01 – 2024/2025

David Filliat
david.filliat@ensta-paris.fr

École Nationale Supérieure
de **Techniques Avancées**

ENSTA

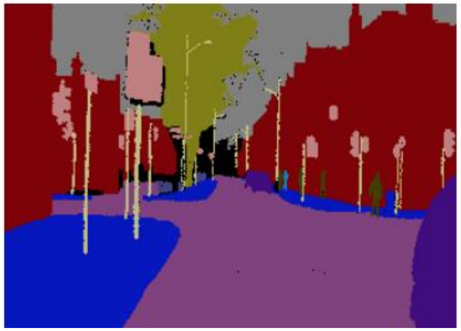
IP PARIS

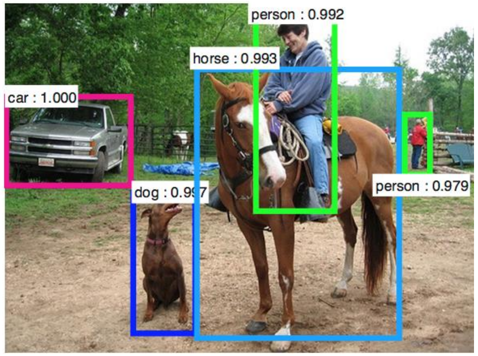
Course agenda

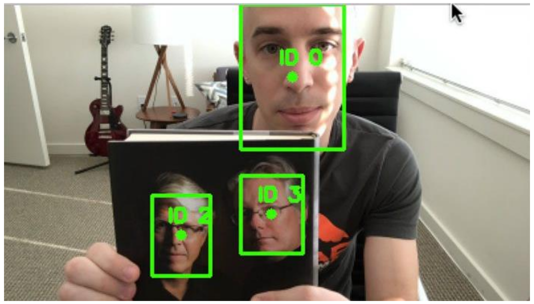
Deep Learning based Computer Vision for robotics

- Today : Deep Learning basics, classification
  - David Filliat
- Semantic segmentation
  - Antoine Manzanera
- Object detection
  - Philippe Xu
- Object tracking
  - Antoine Manzanera

CAT

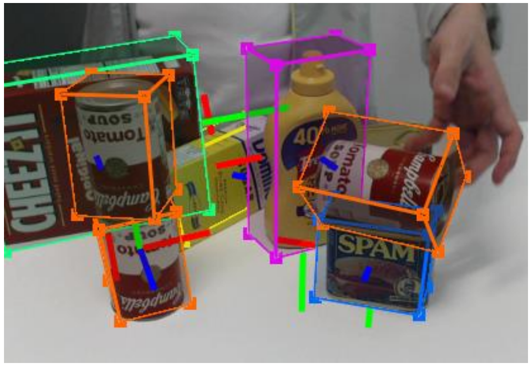




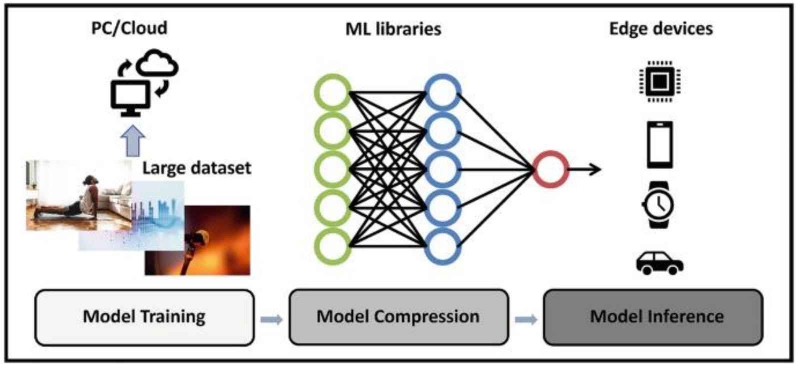


Course agenda

Deep Learning based Computer Vision for robotics



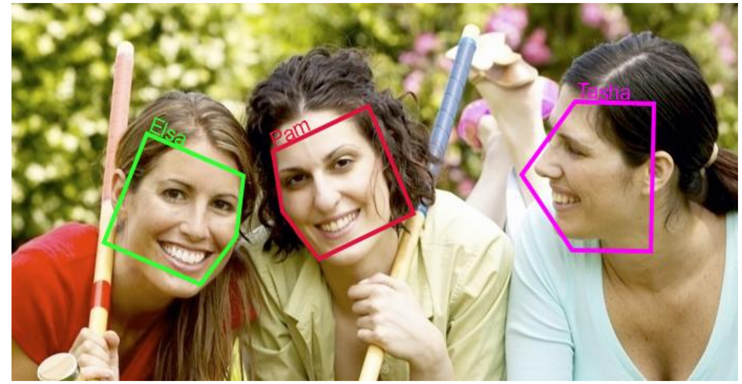
- Pose estimation
  - Thomas Rey
- Embedded deep learning
  - Zhi Yan



Grading

- Research paper oral presentation

Computer vision



Vision tasks such as …

- detection of objects
- recognition of places
- recognition of actions
- detection and recognition of persons

… are very difficult

- Using low level pixel information
- When environment condition change
- Given variability of targets

Many solutions rely on image processing and machine learning

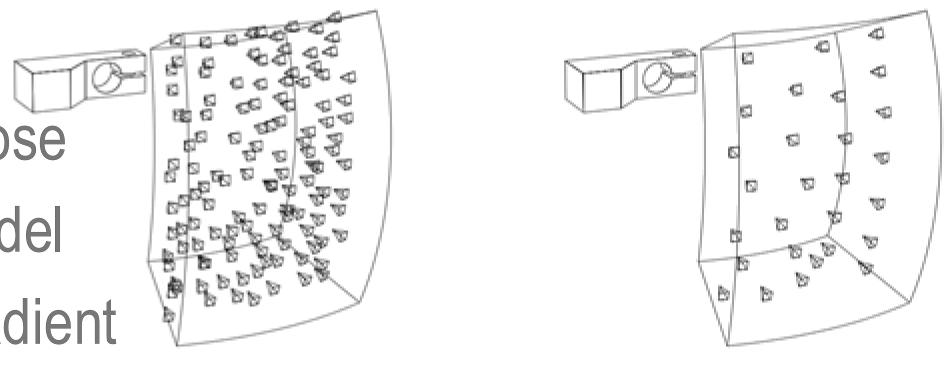
Note : without machine learning

Object recognition can be done without machine learning

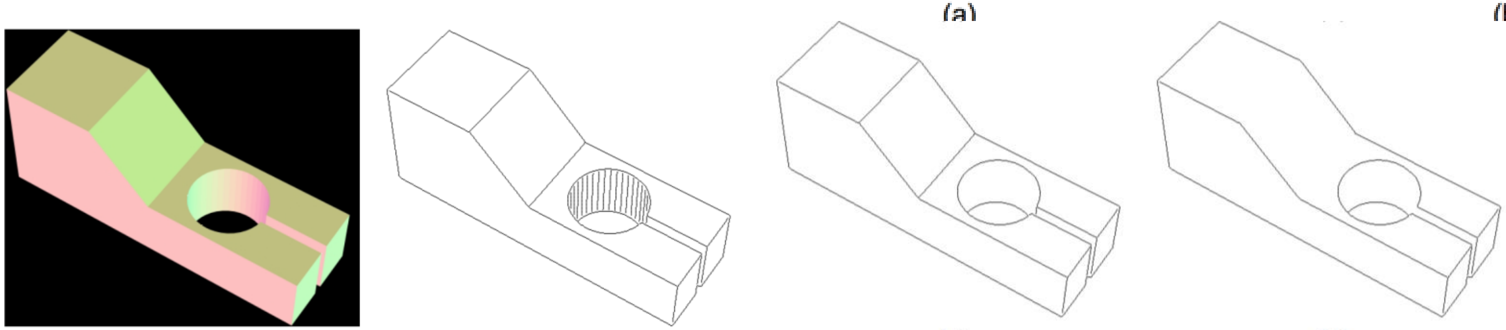
- Ex : Recognition from CAD models in factory environment
- Ex : **CAD**-Based **Recognition** of 3D **Objects** in Monocular Images. Ulrich, Wiedemann, and Steger, 2009

Approach

- Sample relative object/camera pose
- Generate contour views from model
- Measure distance with image gradient



(a) (b)



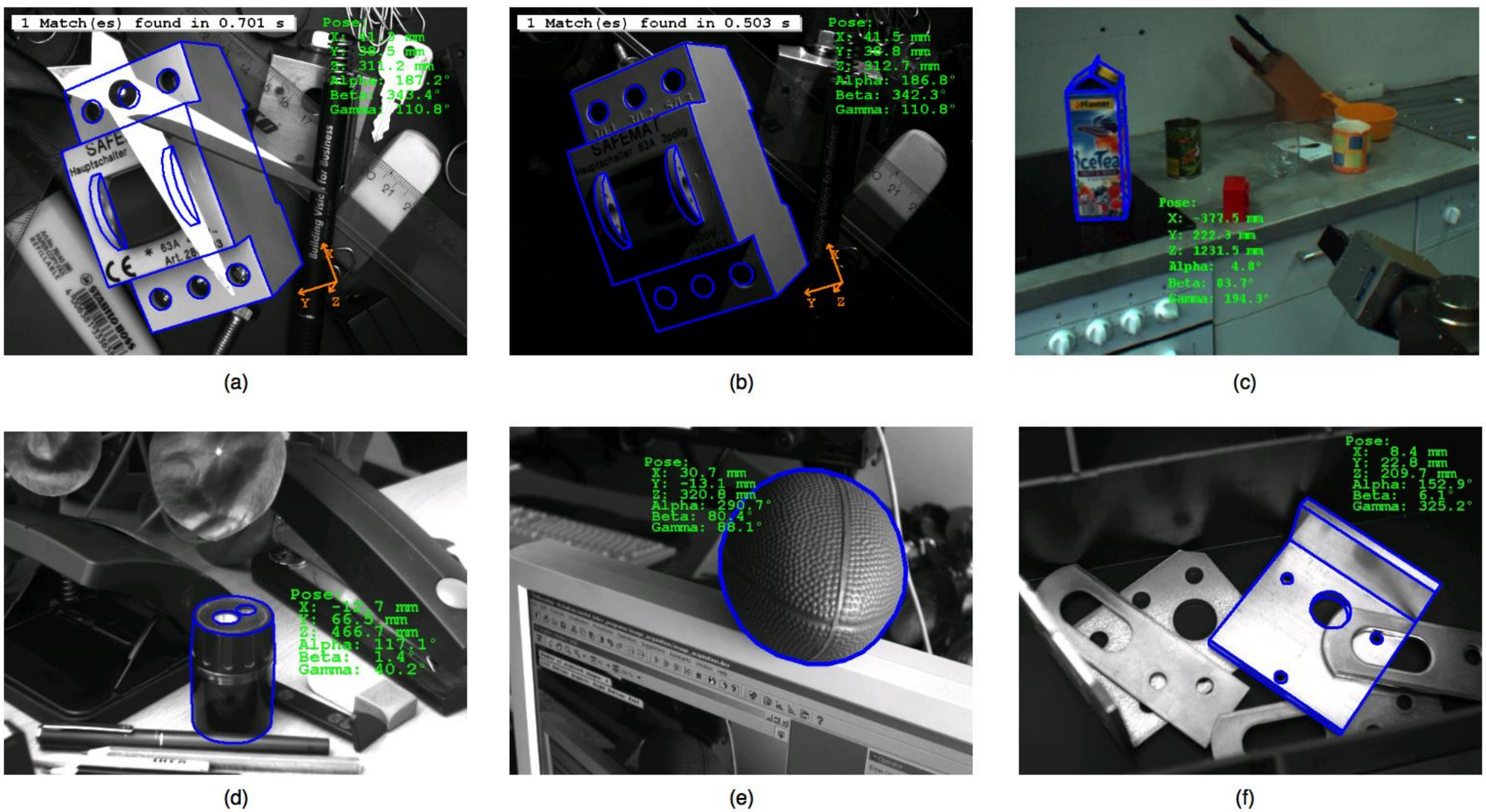
(a) (b) (c) (d)

Multi-scale

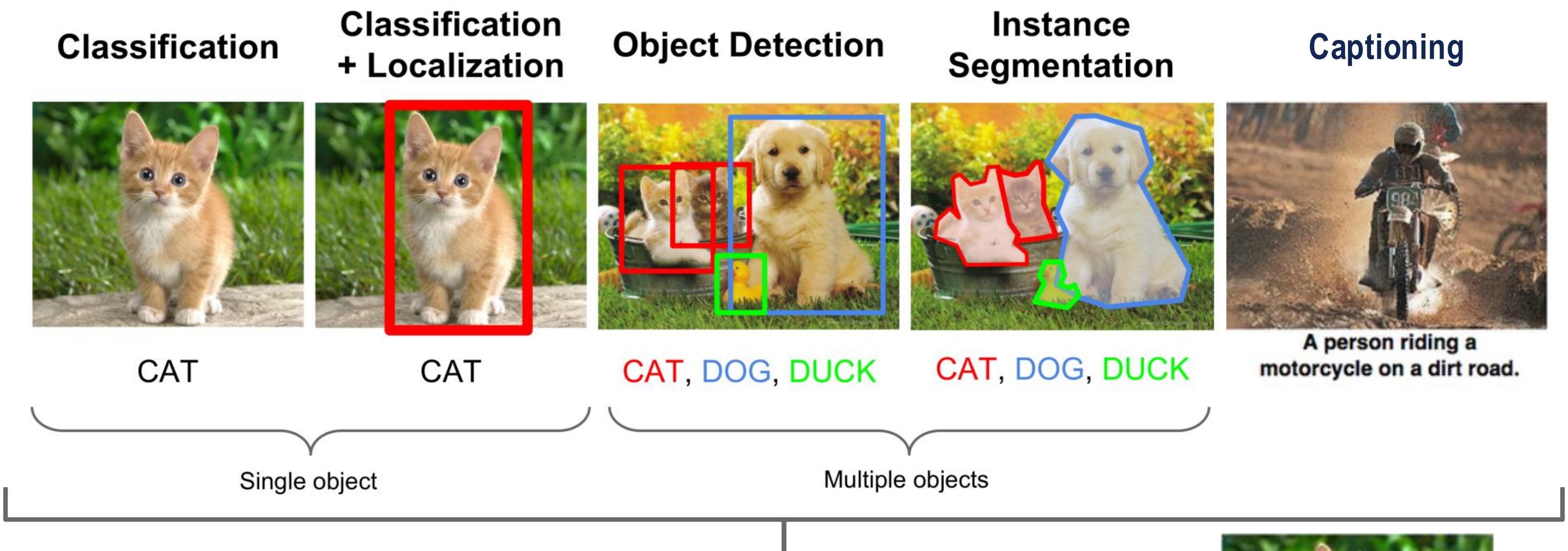
Note : without machine learning

Good performances in practice

- Limited to known/solid objects
- Very precise localization, robust to occlusions, light modifications



Computer Vision Tasks



Requires Classification

→ Program for today

CAT

Objectives

Todays program

- Machine learning / Neural networks basics
- Neural networks for computer vision
- Neural networks training
- Datasets

Practical work

- CIFAR10 image categorization in Pytorch with Google Colab

Machine Learning basics

Introduction to machine learning

Learning is a very weakly defined term

- Better definition needed for mathematical formalization
- Here : function approximation

Suppose there is

- an unknown function f: R^n --> R^m that may have a random component
- a set of **training examples**, consisting of:

  data vectors { x_i }, target values { y_i } obeing $y_i = f(x_i)$

Machine learning

- try to determine the unknown function f from training examples { x_i , y_i }
- **Problem:** how to determine a good approximation to f from data only?

Introduction to machine learning

Generic approach

- Use a parameterized family of functions $f_w(x)$ to approximate f
- Adapt parameter vector w by minimizing a loss function $L(\{f_w(x_i)\},\{y_i\})$ over training examples
- This is called **training** or **learning** !
- Example for L: L2 loss

$$\sum_i \left(f_w(\bar{x}_i) - y_i\right)^2$$

Getting data

- In general, human can “apply” the function (e.g., recognize an object)
- But computing it is hard -> learning

Choice of approximation function

Requirements

- Have “universal approximation capacity“ for a defined class of functions F
- Have an efficient way to update w for minimizing loss

Examples

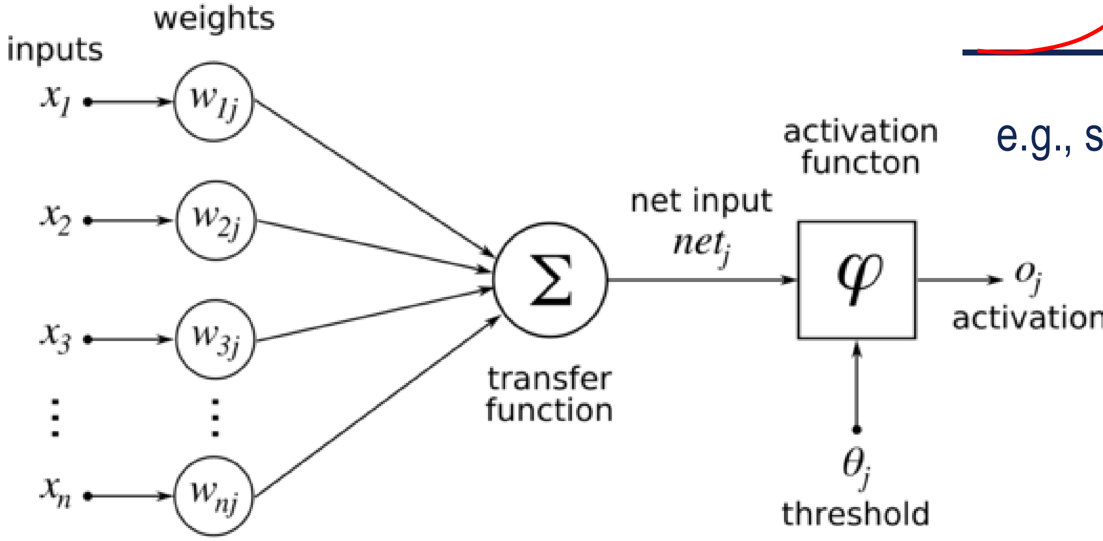
- Single-layer perceptron: linear functions
- Multilayer perceptron: continuous non-linear functions
- Random forest : continuous non-linear functions
- Support vector machine: binary functions
- Boosting: binary functions
- ….

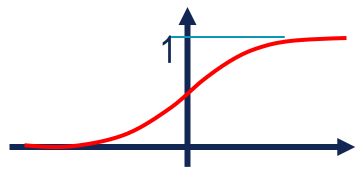
Choice depends on problem!

Neural Networks

Artificial neuron ~1950

- Element performing sum of weighted input + non linear fct



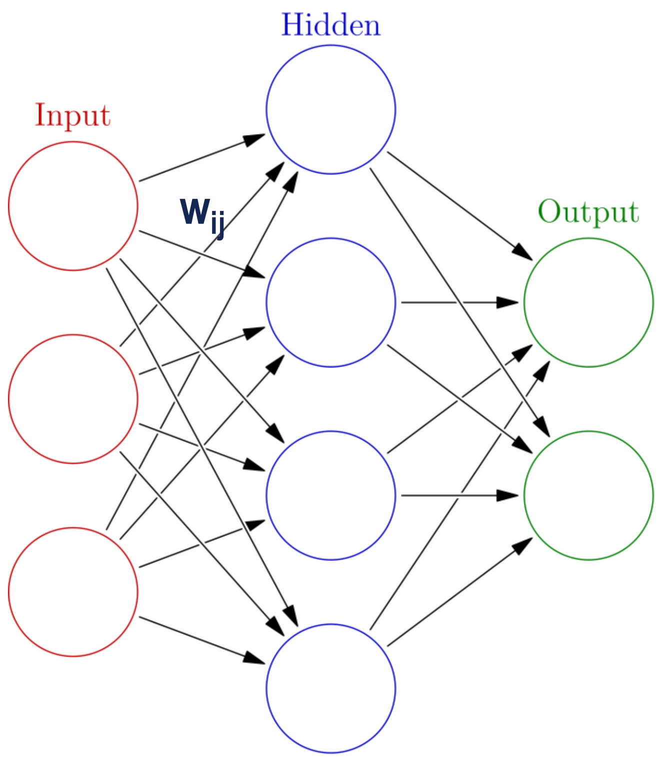


e.g., sigmoid

Neural Networks

Neural network (Perceptron, (Rosenblatt, 58))

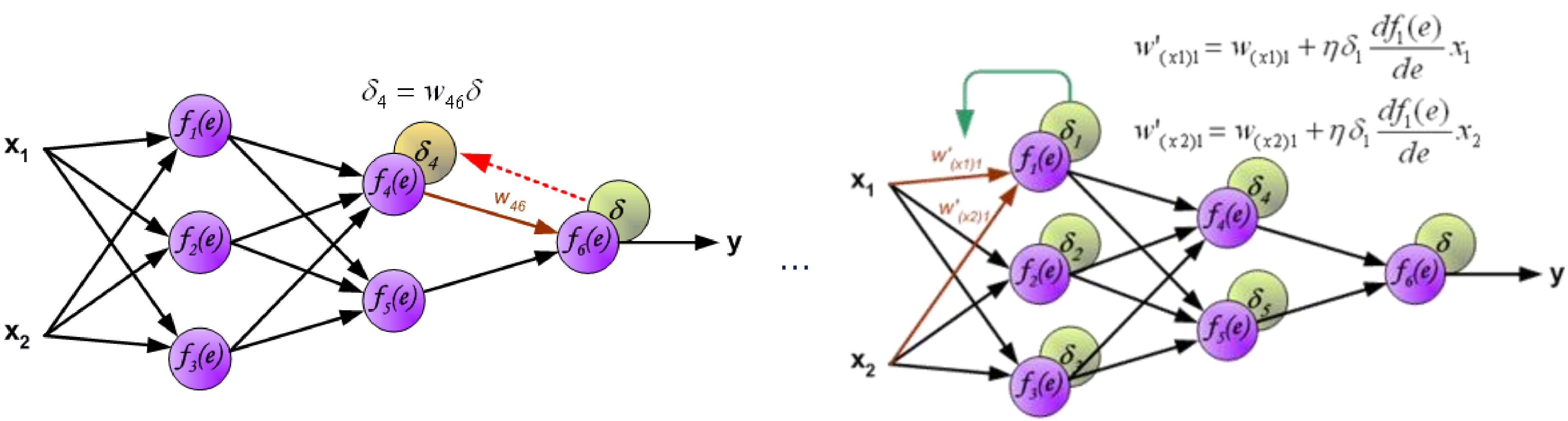
- Assembly of neurons, often organized in layers
- Parameterized by all connection weights $\mathbf{w}_{ij}$



Neural Networks

Learning in neural networks

- Find weights w_{ij} that minimize prediction error
- Backpropagation of error with gradient descent (Werbos, 75)
- Compute: error of output, gradient wrt. weights; update weight following gradient
- Do the same thing for previous layers using ‘chain rule’



Source : Mariusz Bernacki - http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Neural Networks

In practice, automatic gradient computation

- E.g. in pytorch : Define computation using 'Variables'

```
# Create a variable and tell PyTorch that we want to compute the gradient
w = Variable(input_tensor, requires_grad=True)
b = Variable(input_tensor, requires_grad=True)

# Input value
X = 2

# Define the transformation and store the result in new variables
y = w * X + b
loss = (y – 4)*(y – 4)
```

- Automatic gradient computation

```
loss.backward()
```

- Use gradient wrt. w to update weights

w = w - 10^{-3} * w.grad()

Deep Learning

Return of the neural networks

- Around 2010 ?
- Neural networks with “many” layers
- Theory similar to perceptron (for dense/cnn models)

Why “Deep” ?

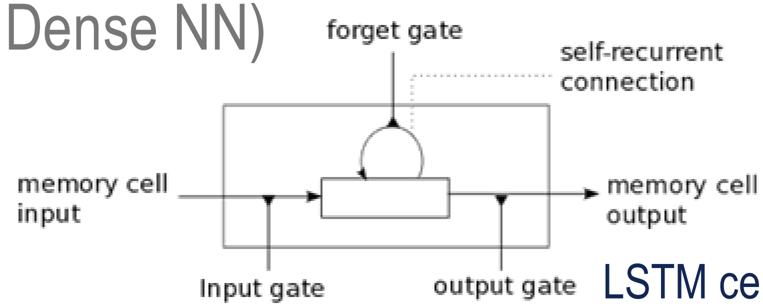
- Approximate more complex functions
- Works well in practice (on many problems)

Why now ?

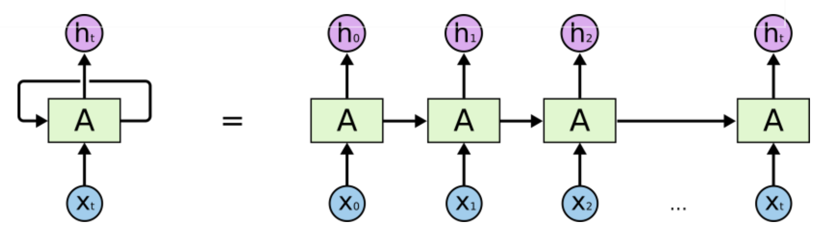
- More processing power
- Availability of large datasets (ImageNet)
- Found solutions to some learning problems

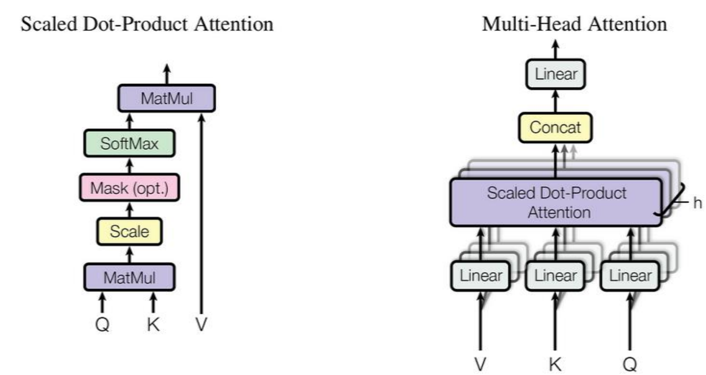
Many architectures

- Fullly connected Neural Networks (aka Dense NN)
- Convolutional Neural Networks
  - Specialized for image processing
- Recurrent architecture (e.g. LSTM)
  - Processing of temporal data
  - Trained by unfolding + supervised learning
- Transformer
  - Processing sequential data with attention
  - Can also be applied to images
- …



LSTM cell



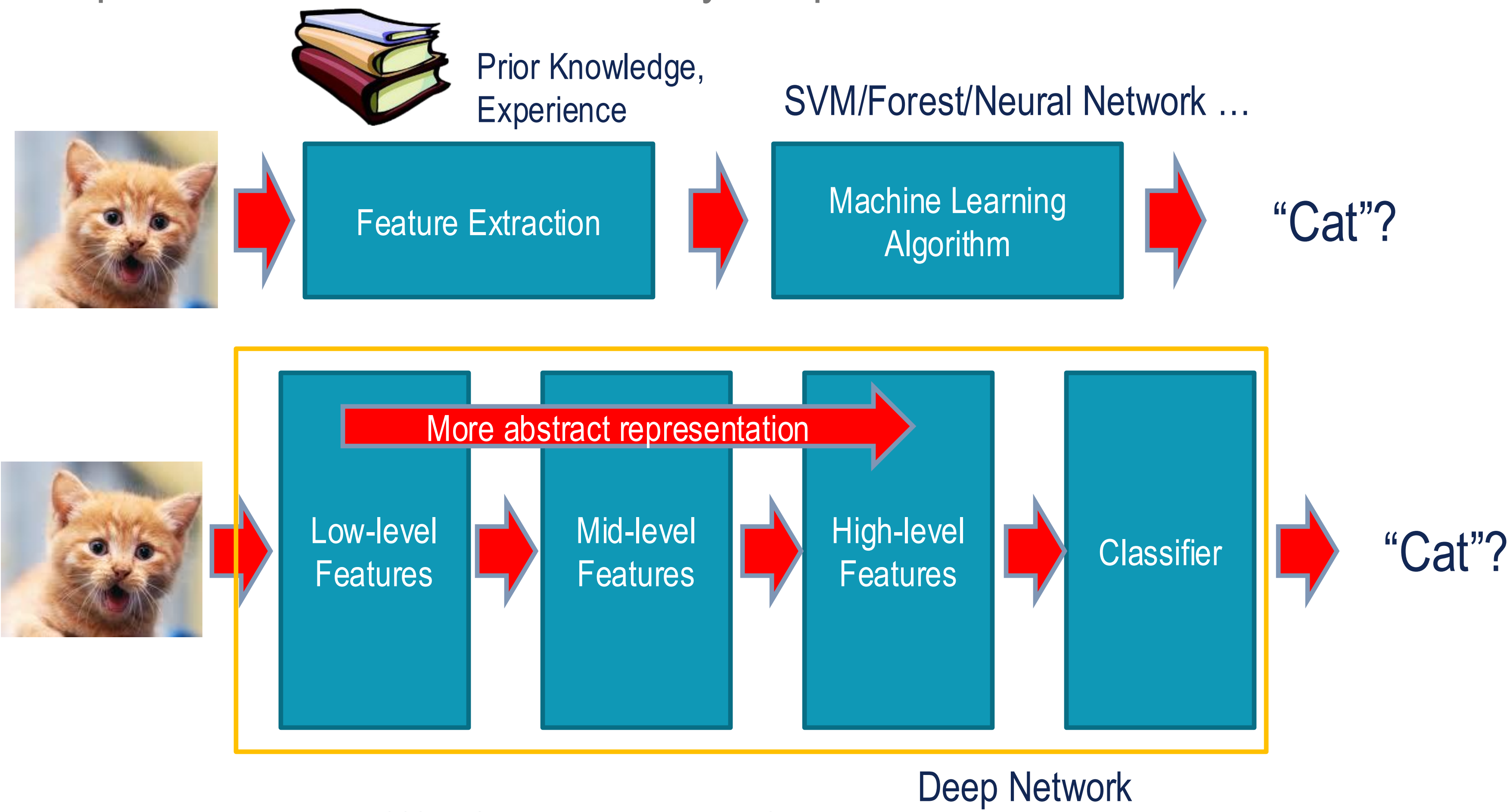


Neural Networks for computer vision

Deep learning for vision

Avoid manual feature construction

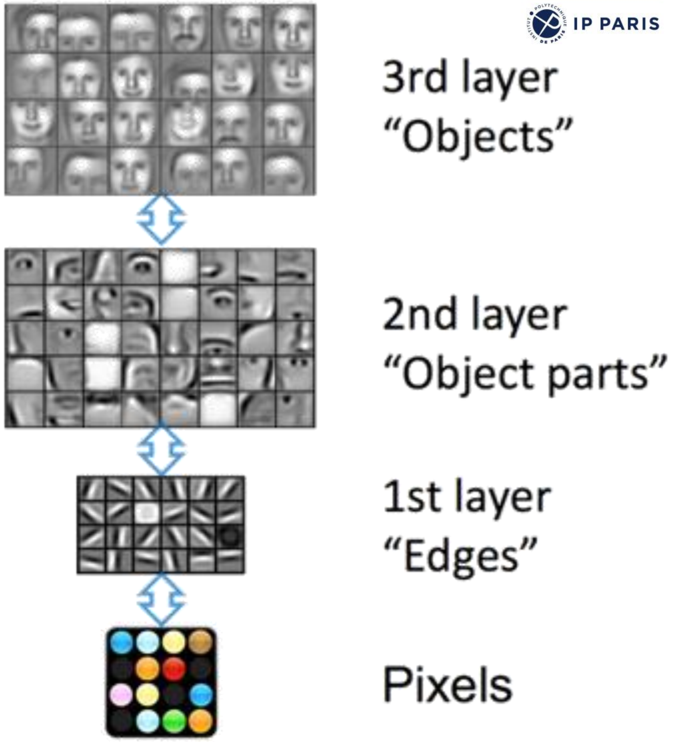
- Replace traditional architecture by deep network



Deep learning for vision

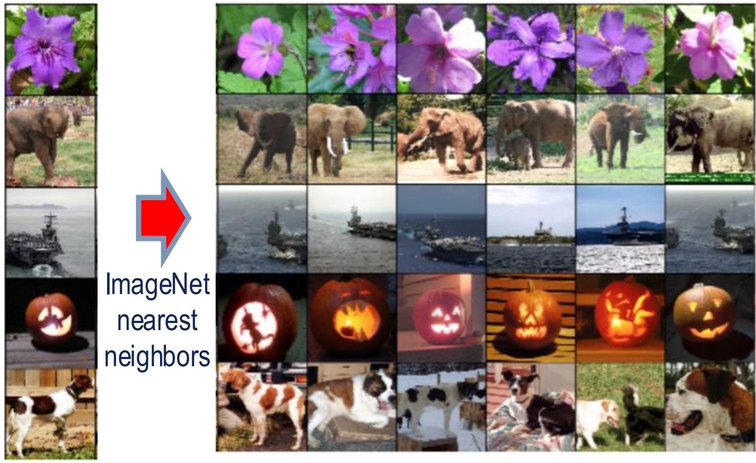
Avoid manual feature construction

- Process raw data directly
- Learn directly relevant feature from data
- Natural increase of feature abstraction
- ‘Semantic invariance’ of last layers
- Adapts to other modalities (depth, IR …)



Problems with ‘perceptron’

- Large image size -> large networks
- Need lots of training data
- $\longrightarrow$ reduce network parameters

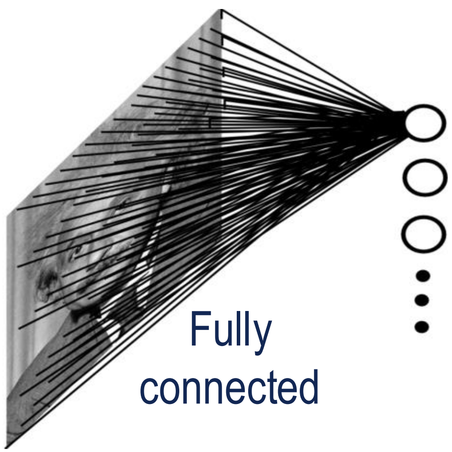


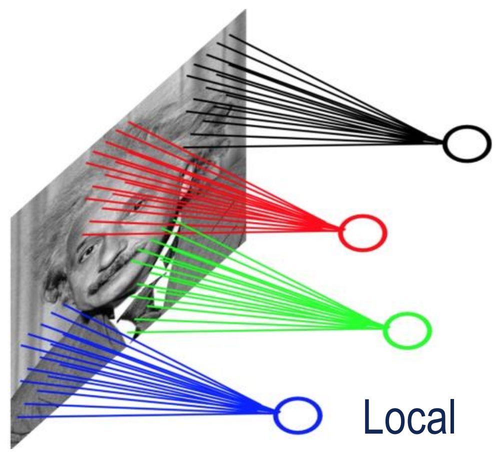
Convolutional Neural Networks

Reducing number of network parameters

- Use only limited local support
- Exploiting image invariance to translation:

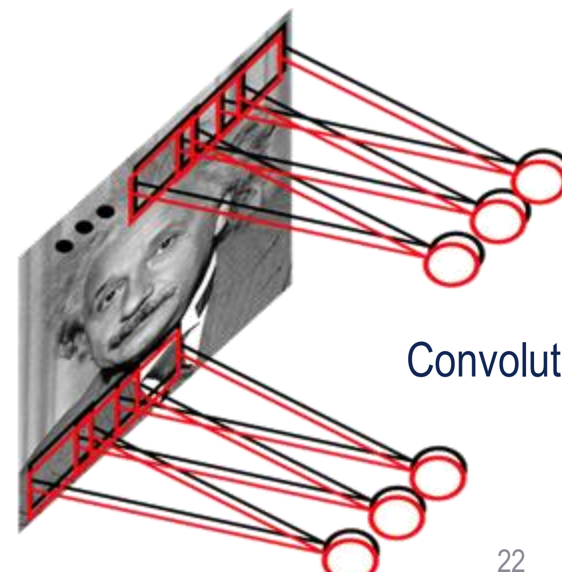
  Use same local weights for all positions -> convolution





- Use several convolutions

  at each position -> multiple features layers



Convolution

Convolution in 1D

- Mathematical definition:

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

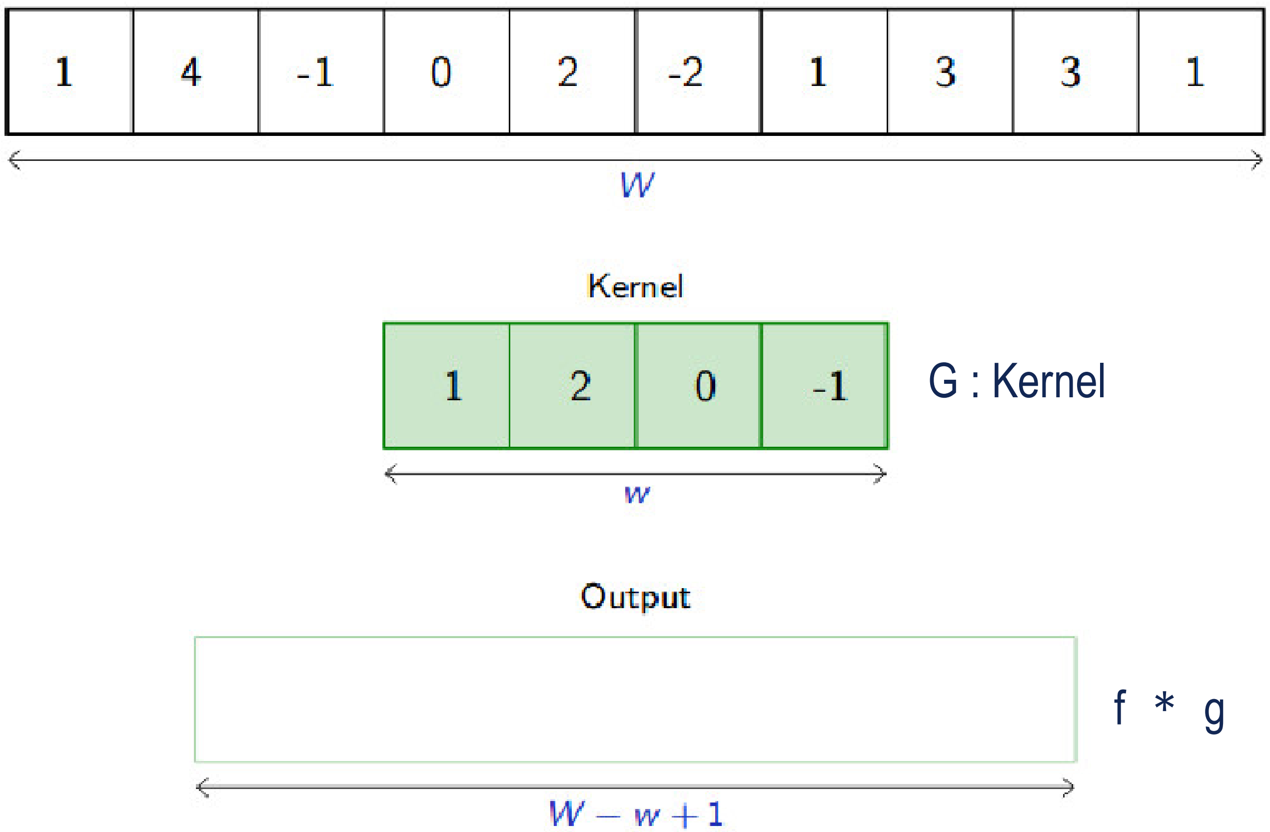
- In deep learning, we usually use cross-correlation which is very similar (but still use the name convolution…)

$$(f * g)[n] = \sum_{m=-M}^{M} f[n+m]g[m]$$

Convolution

Convolution (cross correlation) in 1D

$$(f * g)[n] = \sum_{m=-M}^{M} f[n + m]g[m]$$



F : Image

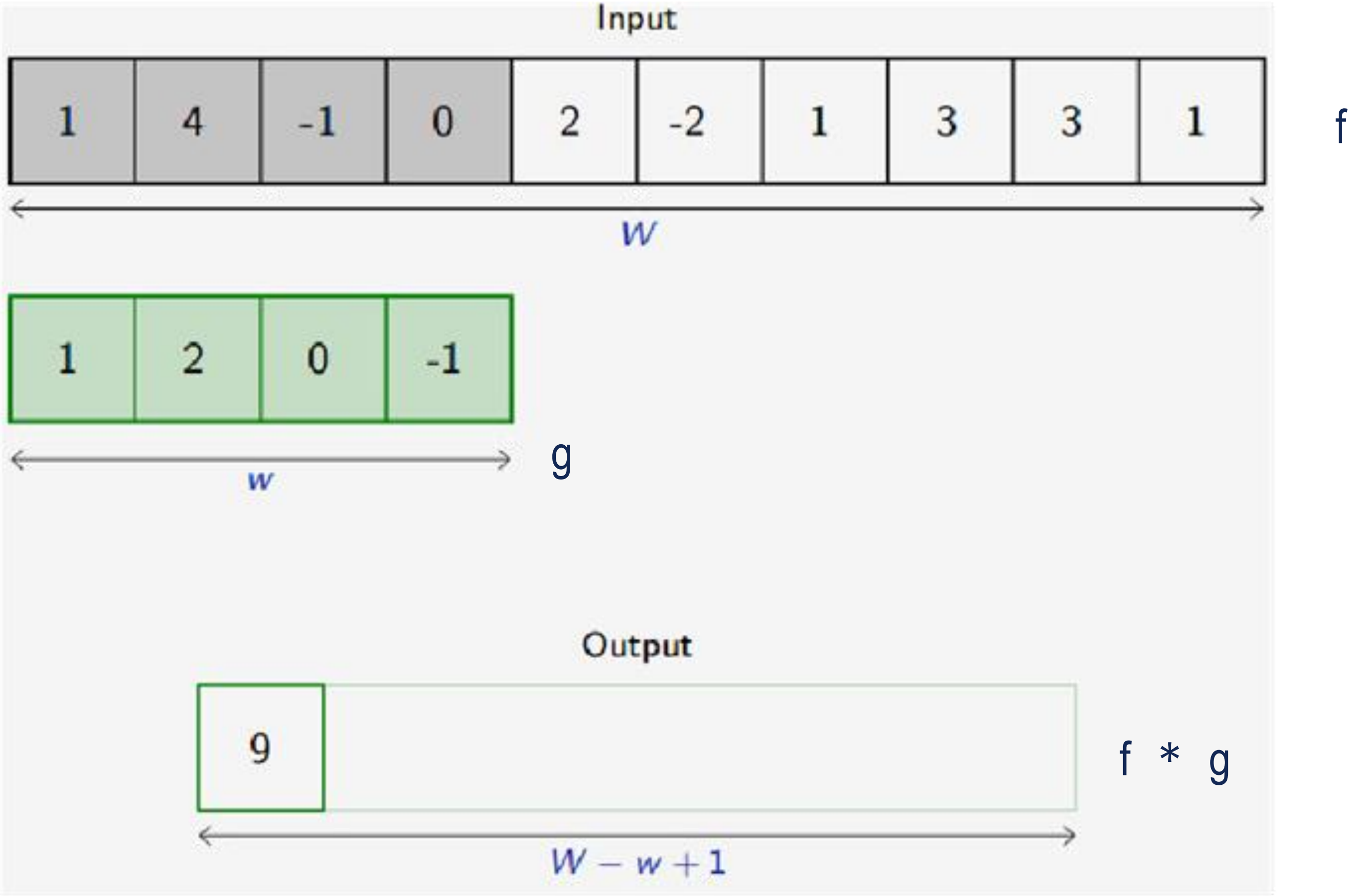
G : Kernel

f * g

[2]Credits: Francois Fleuret

Convolution

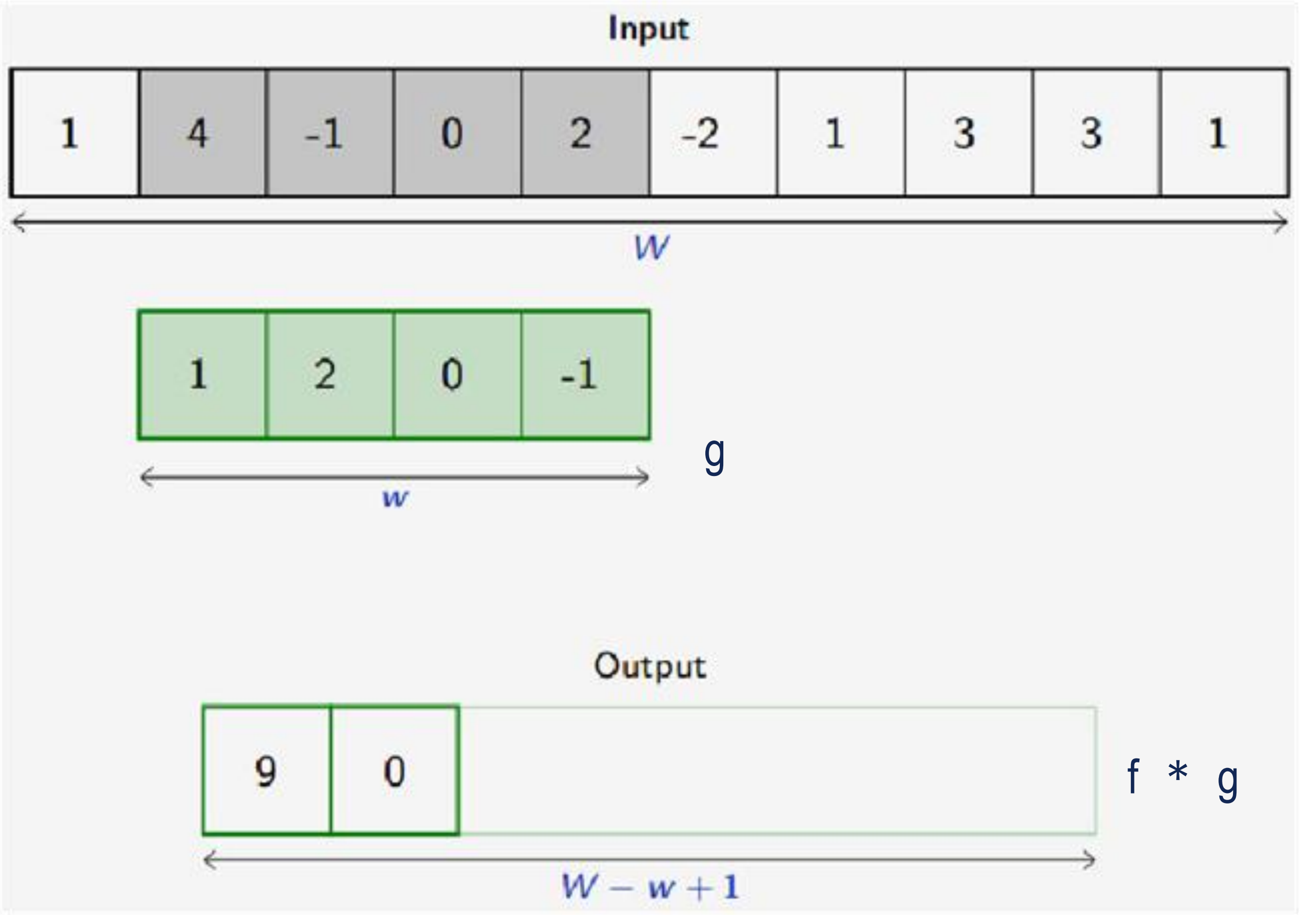
Convolution in 1D



[3]Credits: Francois Fleuret

Convolution

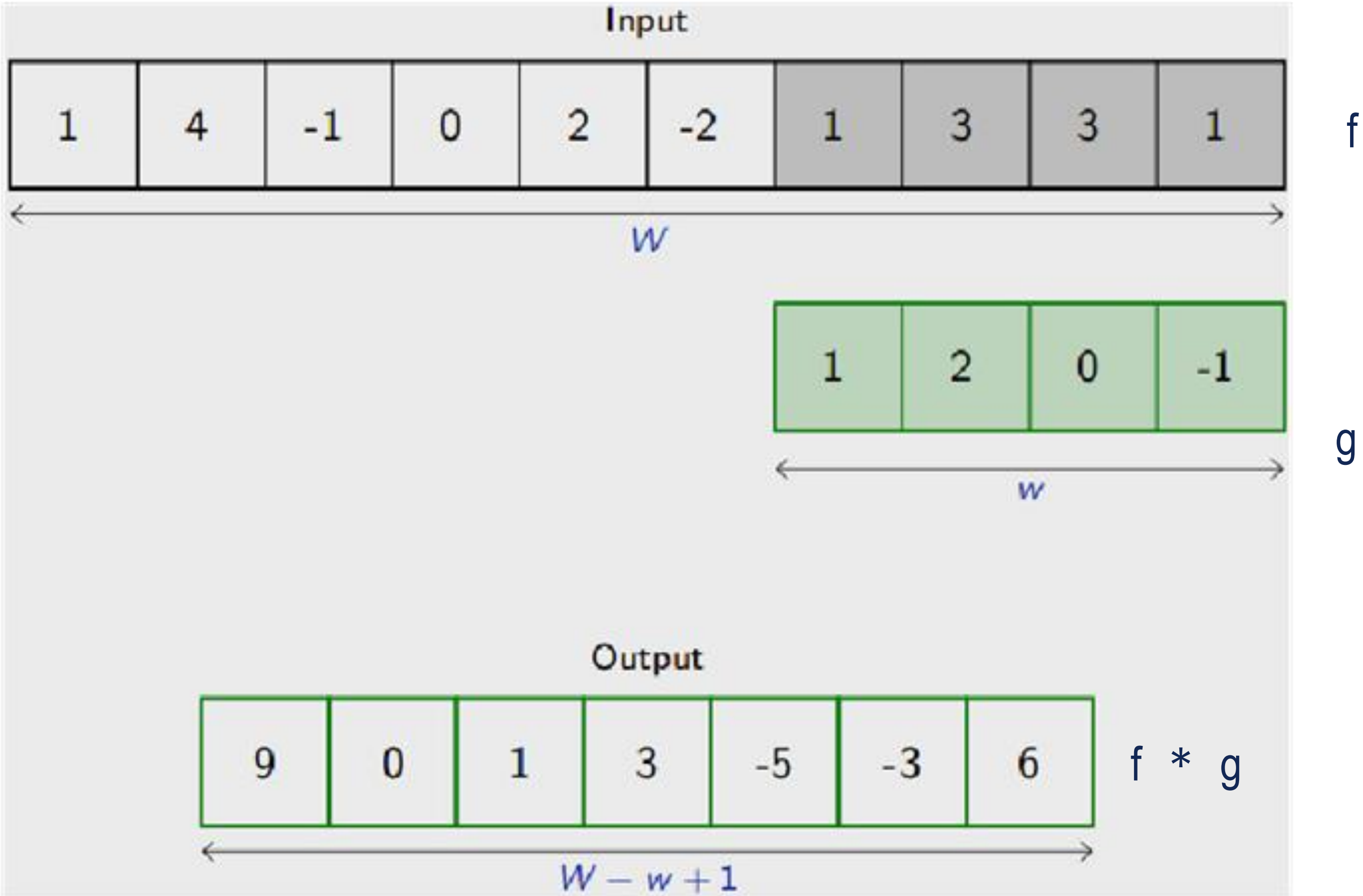
Convolution in 1D



[4]Credits: Francois Fleuret

Convolution

Convolution in 1D



[9]Credits: Francois Fleuret

Convolution

Convolution in 2D

- Gray scale images

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

- Color images (c = 3)

3D matrix = tensor

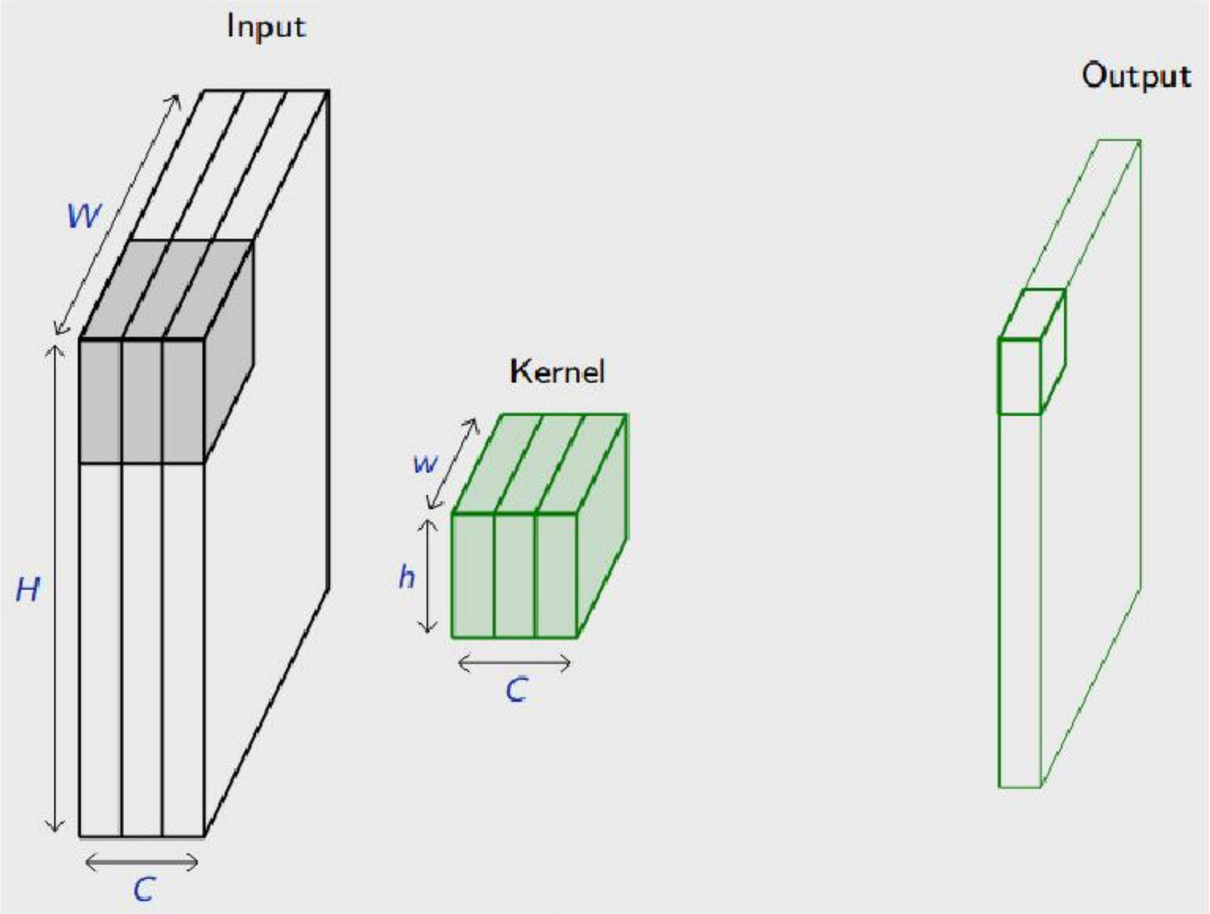
$$(f * g)[n_1, n_2] = \sum_{k=0}^{3} \sum_{m_1=-M}^{M} \sum_{m_2=-M}^{M} f[n_1 - m_1, n_2 - m_2, k] g[m_1, m_2, k]$$

- (in fact cross-correlation)

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

Convolution

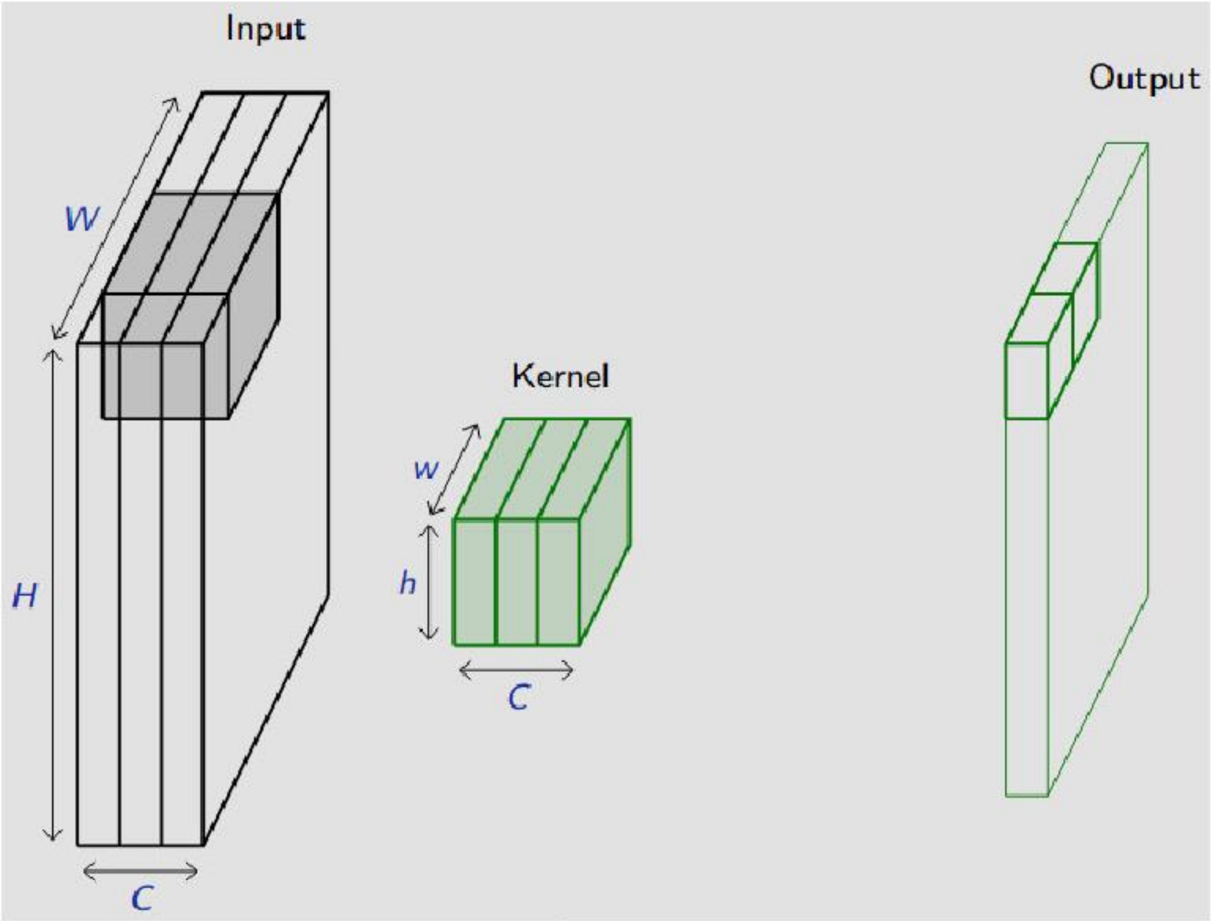
Convolution in 2D



[10] Credits: Francois Fleuret

Convolution

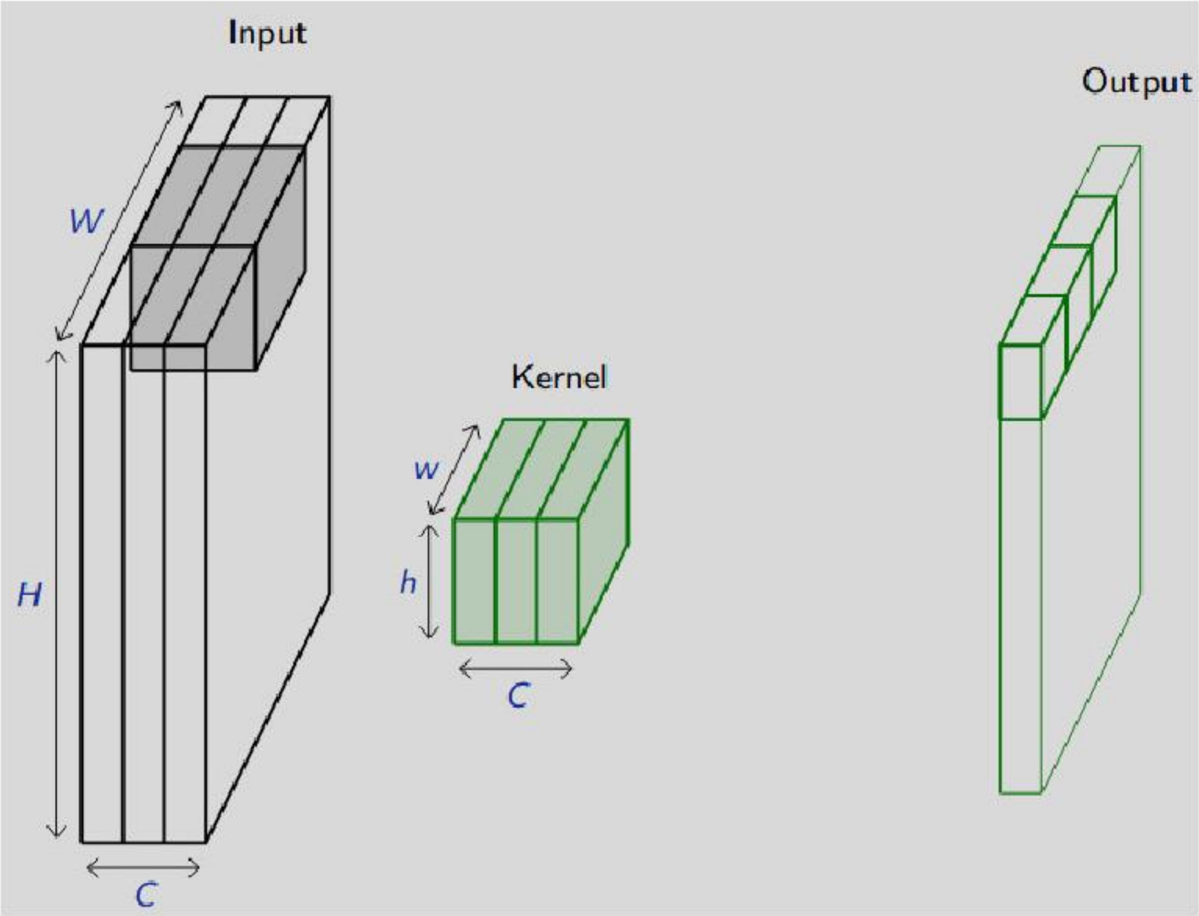
Convolution in 2D



[11] Credits: Francois Fleuret

Convolution

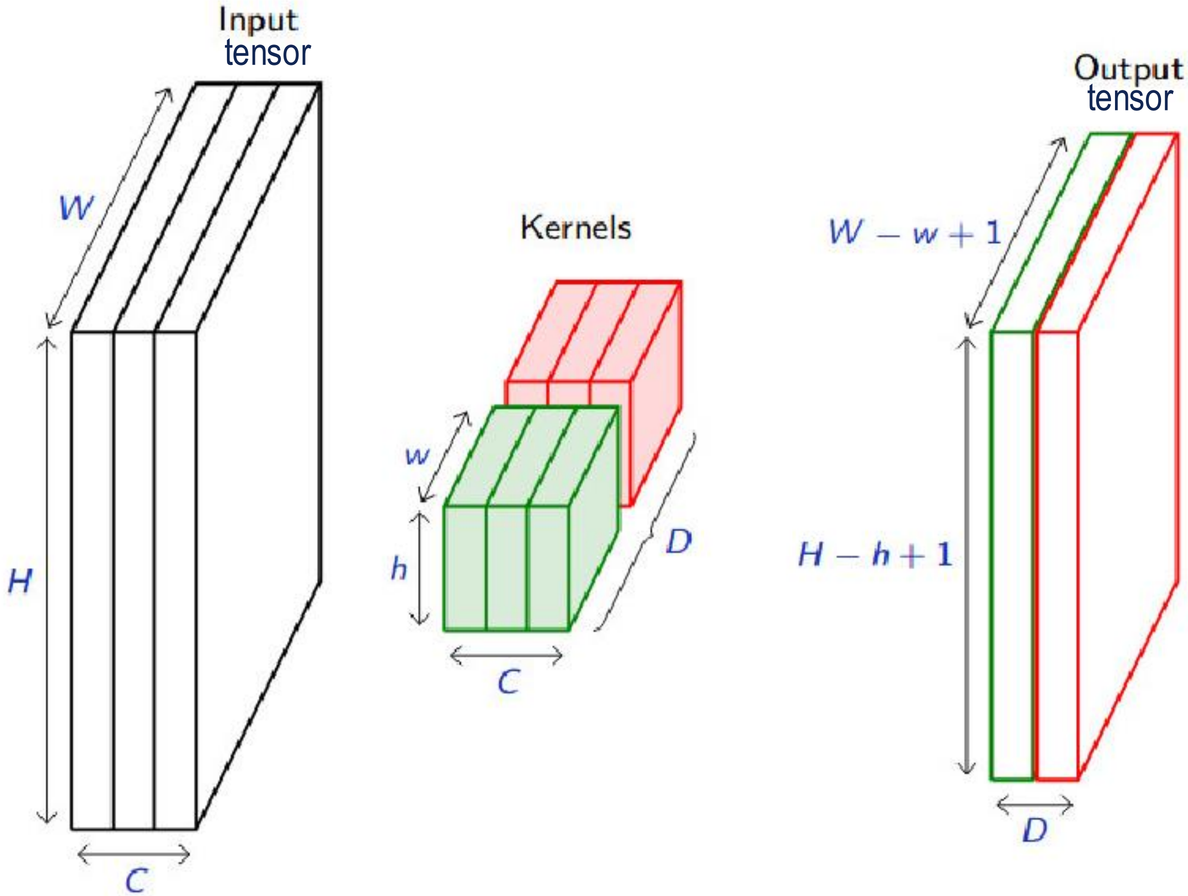
Convolution in 2D



[12] Credits: Francois Fleuret

Convolution

Multiple convolutions in 2D

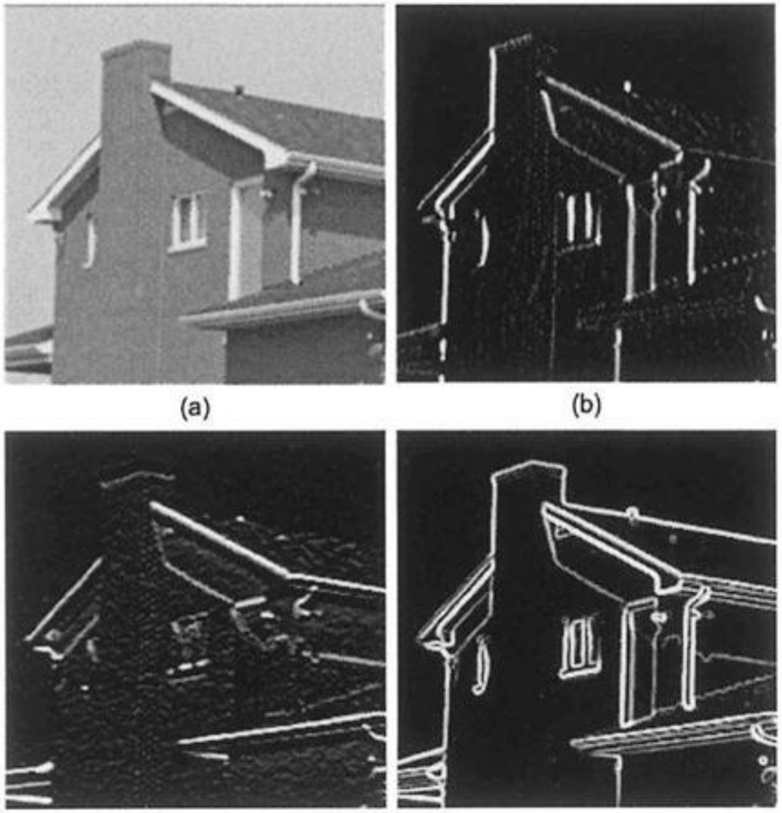


[16]Credits: Francois Fleuret

NB: Convolution in image processing

Ex : Sobel edge detector (1968)

- Convolution with 'Hand made' filters



(a) (b)

(c) (d)

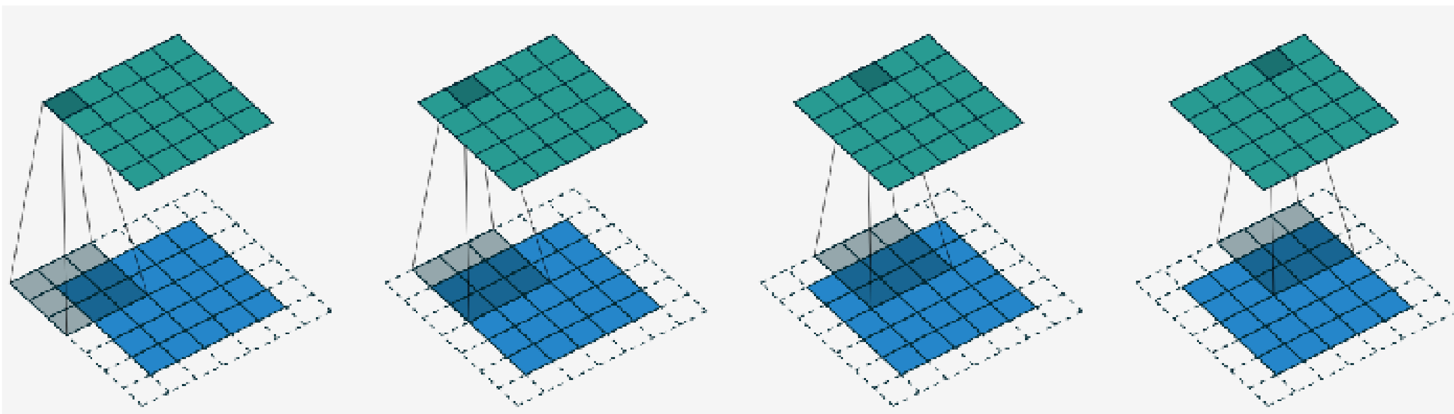
$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

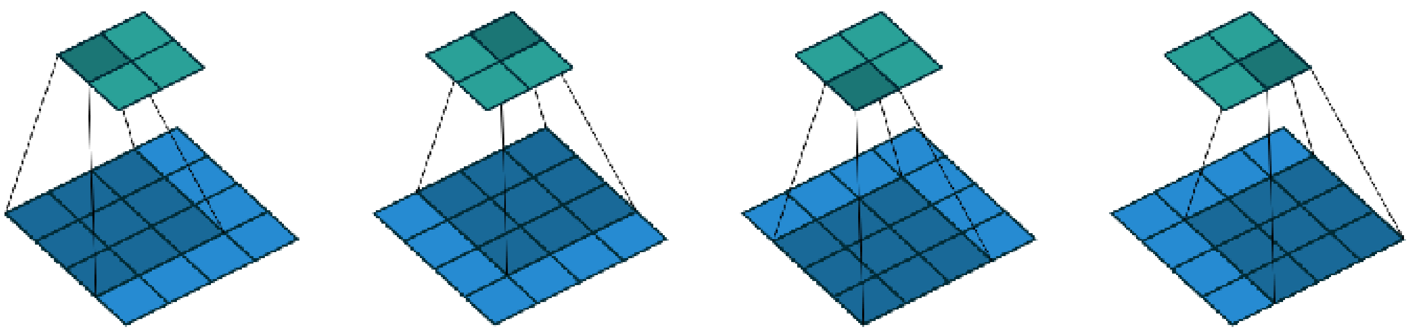
$$\sqrt{G_x^2 + G_y^2}$$

Padding

Keeping constant image/feature map size



Padding 1

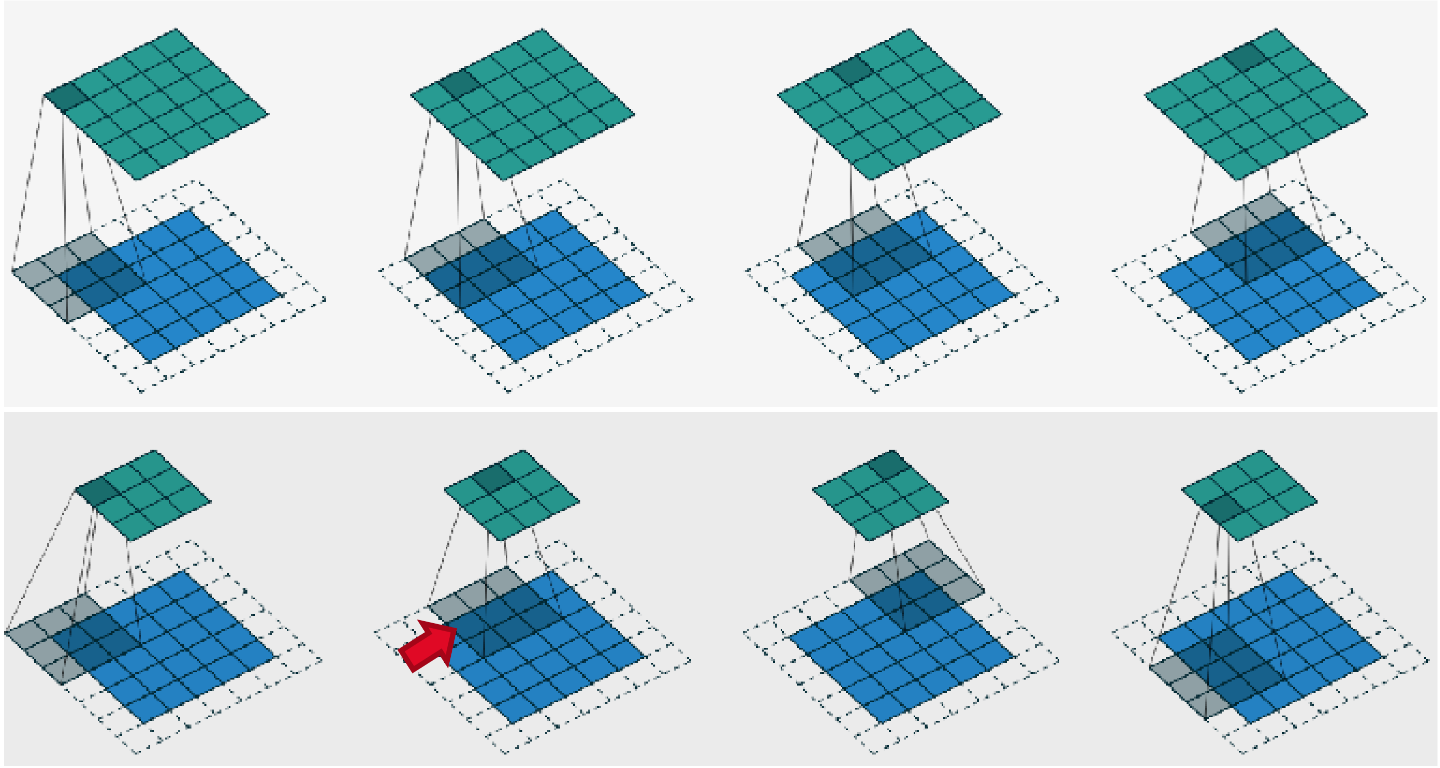


Padding 0

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

Stride

Reducing image / feature map size



Stride 1

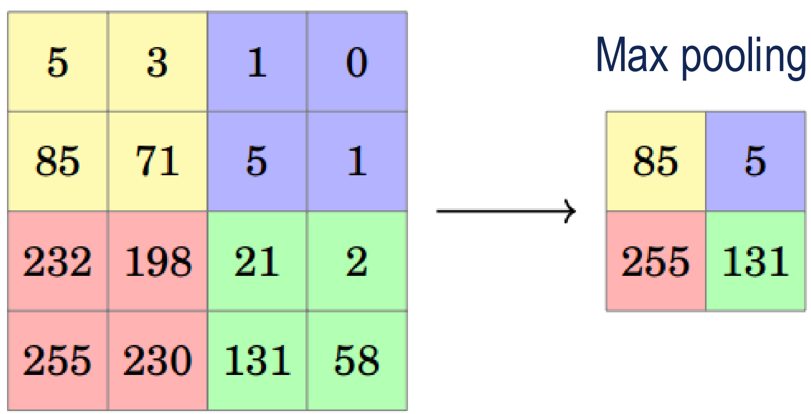
Stride 2

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

Pooling

Reducing feature map size

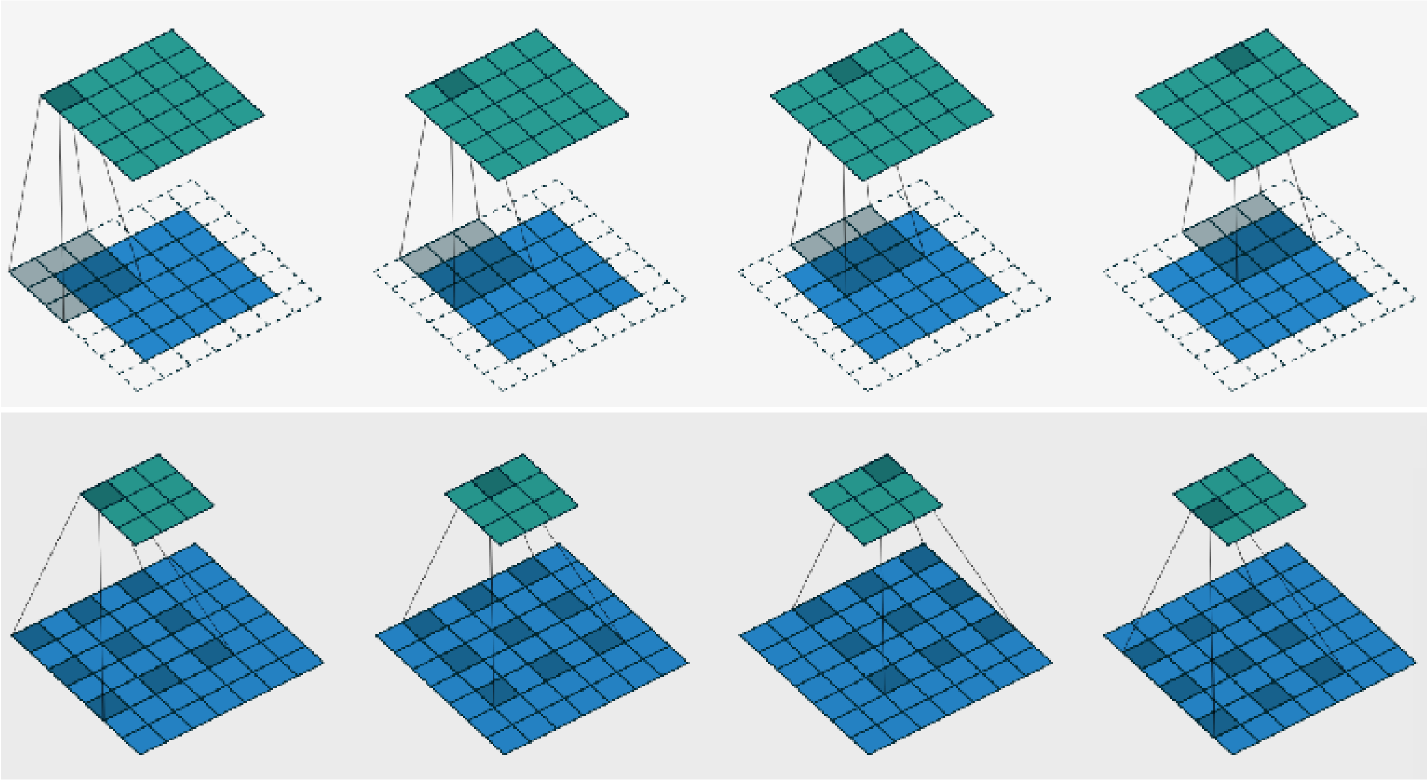
- Because higher level are more ‘semantic’ and less fine grained
- Replace values by max / average of local neighborhood
- Computation similar to convolution



Dilation

Reducing image / feature map size

- Expand receptive field with same number of params



Dilation 1
Kernel 3x3

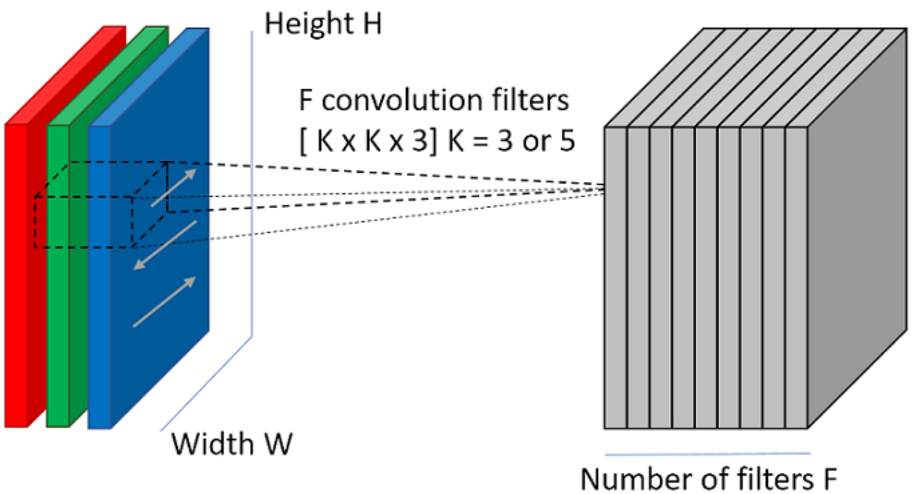
Dilation 2
Kernel 3x3
-> 5x5

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

Convolutional Layers

Convolution layer parameters

- Kernel size / padding / stride
- Number of Input/output feature maps



Input Layer (RGB pixels)
[H x W x 3]

Convolution Layer Output
[H x W x F]
assuming stride=1 and zero padding

Output feature map size

$$H_2 = \lfloor \frac{H_1 - kernel\_size + 2 \times padding}{stride} \rfloor + 1$$

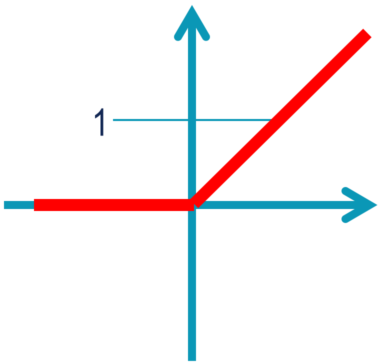
$$W_2 = \lfloor \frac{W_1 - kernel\_size + 2 \times padding}{stride} \rfloor + 1$$

Other Layers

Activation layers

- Simply apply function to each tensor element
- In practice, often use ReLU (see later)
- Many variants (GeLU, leaky ReLU, …)

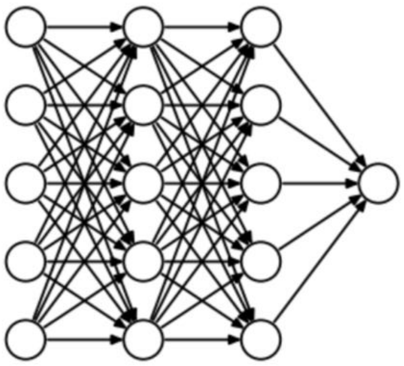
ReLU(z) = max{0, z}



"Rectified Linear Unit"

Dense layers

- 'linear' layers with full connections
- Cf. Perceptrons
- Can be seen as a convolution with kernel size equal to the input feature map size



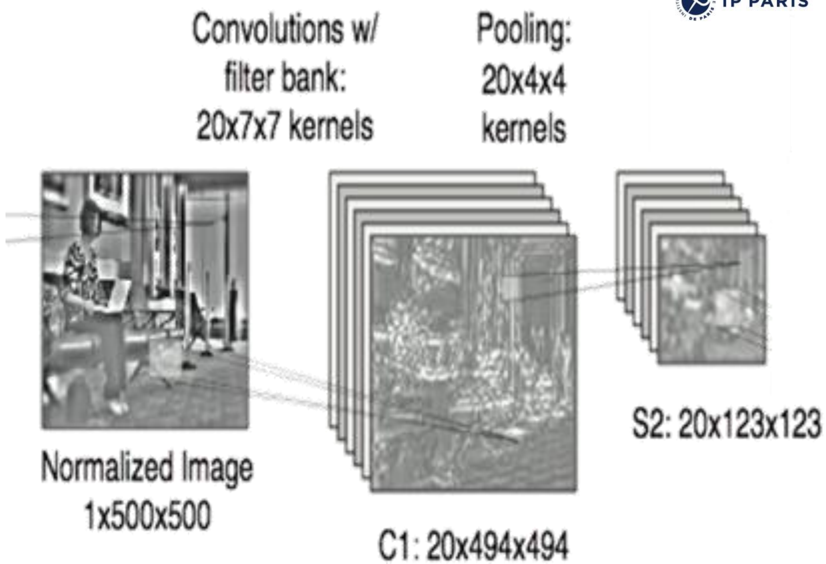
Batch Normalization

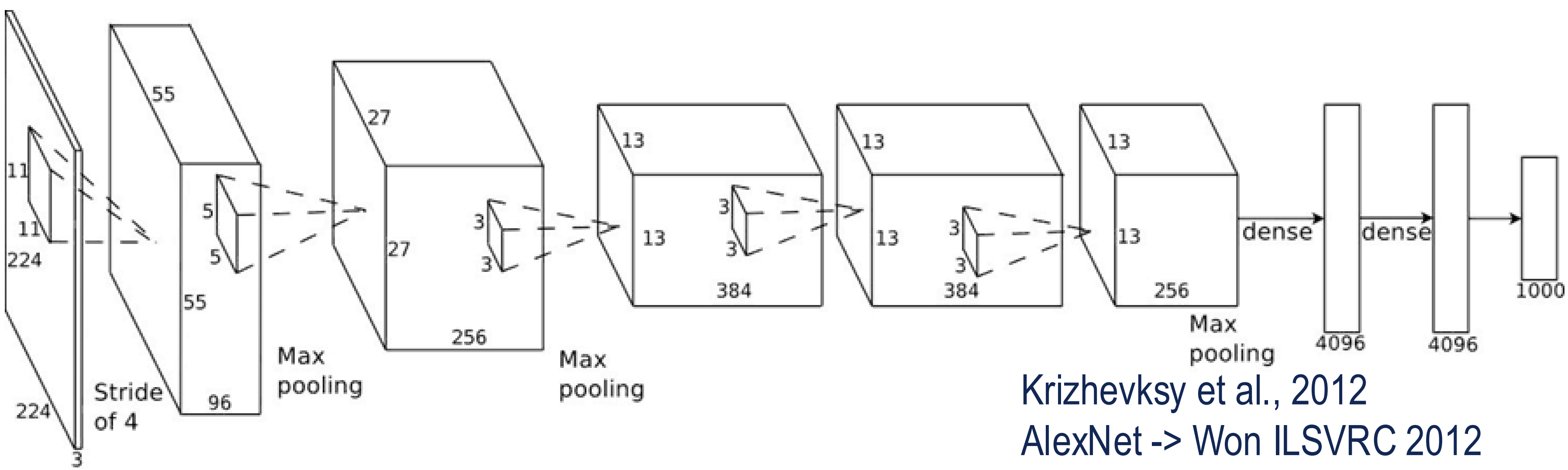
- Normalize activations (see later)

Convolutional Neural Networks

Stack of basic layers

- Convolutions with a given step (stride)
- Non linearity (ReLu)
- Pooling (Reduce resolution)
- Batch Normalization (optional)
- Finish with fully connected layers



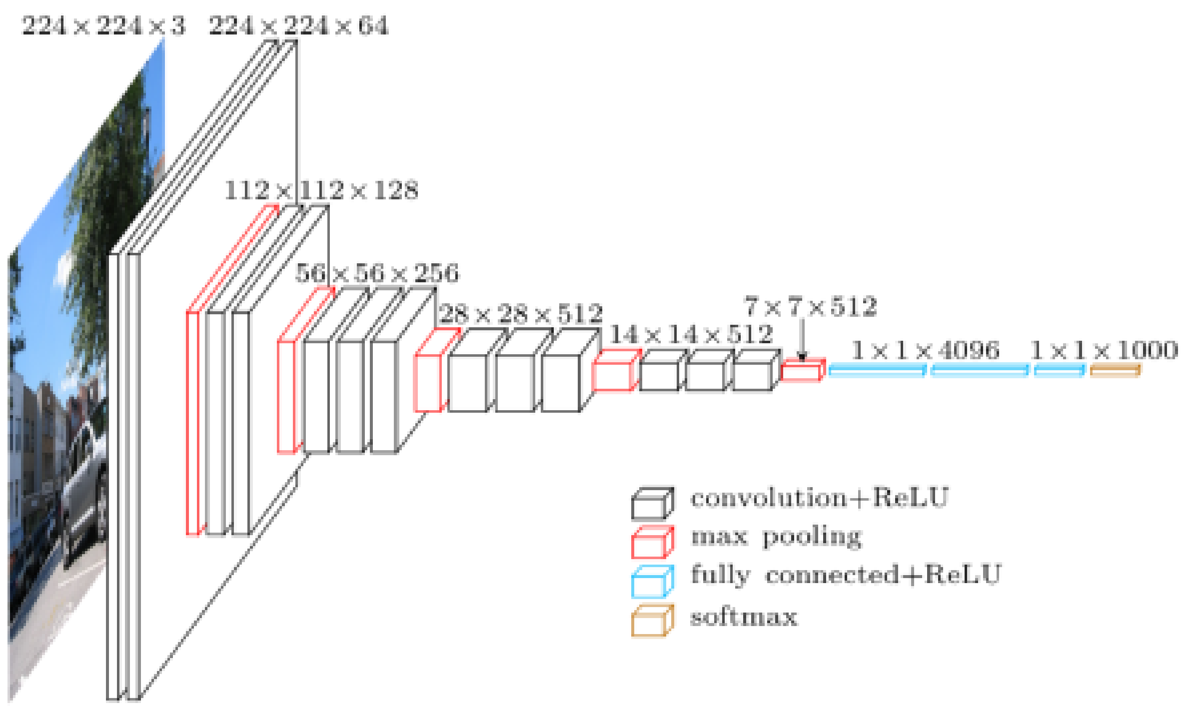


Krizhevksy et al., 2012
AlexNet -> Won ILSVRC 2012

Some common architectures

VGG Net

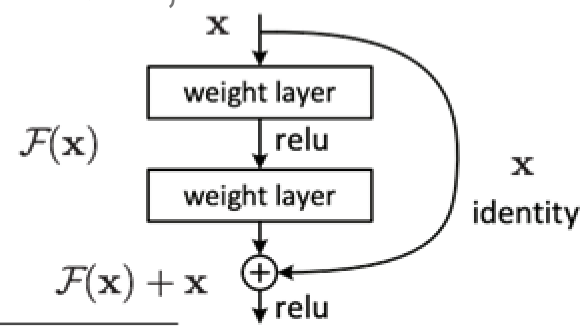
- Very Deep Convolutional Networks for Large-Scale Image Recognition, Simonyan & Zisserman, 2015
- Often used as pre-trained network

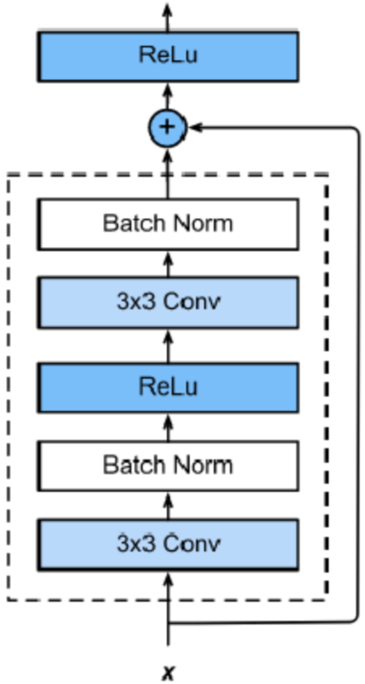


Some common architectures

Resnet

- Deep Residual Learning for Image Recognition, He & al., 2015
- Added connections to encode 'residuals' (i.e. Δx)
- Much easier to train for deeper nets (-> 1000)
- Won several challenges in 2015





model	top-1 err.	top-5 err.
VGG-16 [41]	28.07	9.33
GoogLeNet [44]	-	9.15
PReLU-net [13]	24.27	7.38
plain-34	28.54	10.02
ResNet-34 A	25.03	7.76
ResNet-34 B	24.52	7.46
ResNet-34 C	24.19	7.40
ResNet-50	22.85	6.71
ResNet-101	21.75	6.05
ResNet-152	**21.43**	**5.71**

ImageNet Validation

Some common architectures

DenseNet architecture

- Generalize resnet by connecting to several forward layers
- Concatenate information instead of summation
- Overall smaller networks because number of layers can le reduced

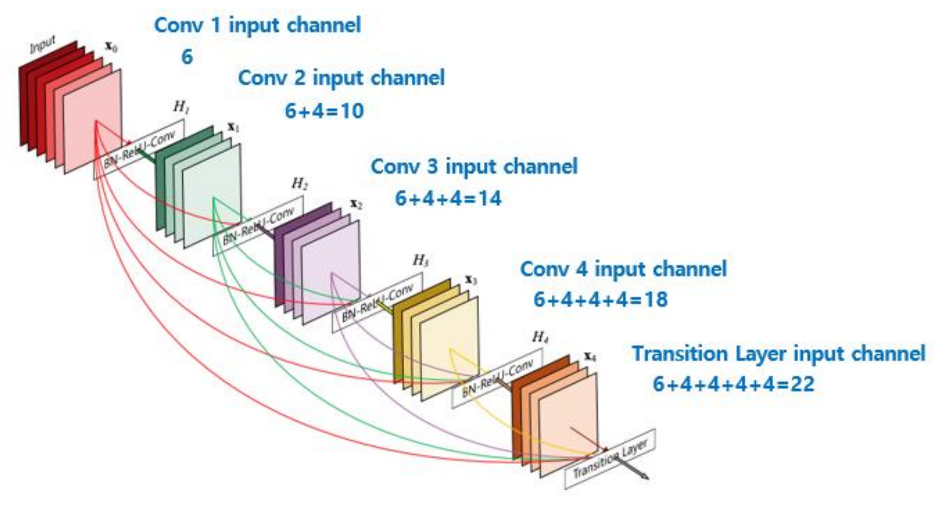
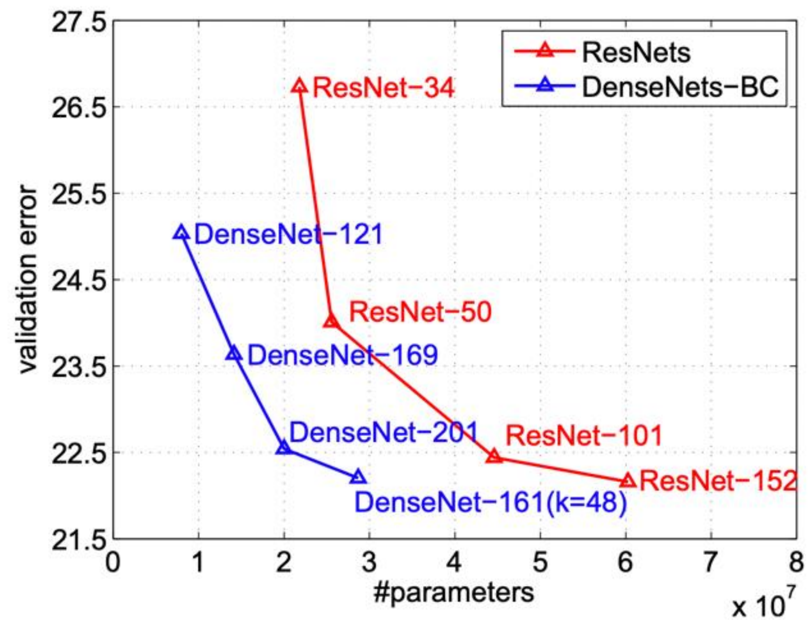


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

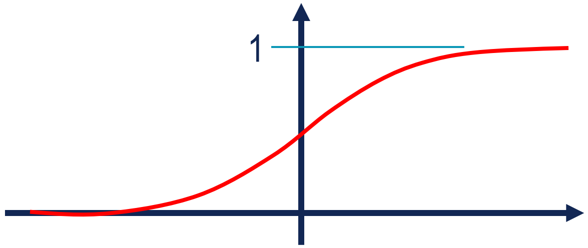


Densely Connected Convolutional Networks, Gao Huang, Zhuang Liu, Laurens van der Maaten, Kilian Q. Weinberger, 2017

Neural Networks training

Training Deep Networks

$$\frac{1}{1+\exp(-\theta^\top x)}$$



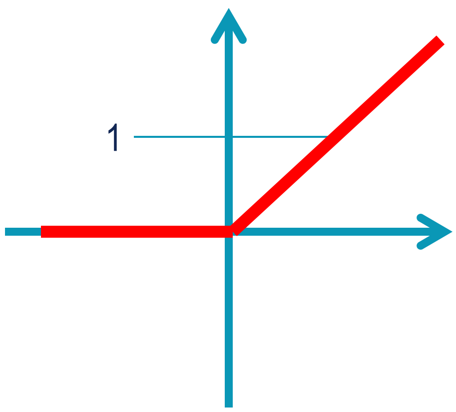
Training with back-propagation

- Dates back to Werbos (75)
- But did not work on “deep” networks
  - Many local minima in cost function
  - Vanishing/exploding gradient in the deep layers
  - Hard to debug/understand

What’s new ?

- Choice on activation function (instead of sigmoid)
  - Tanh, **ReLU** -> reduces gradient vanishing
- More effective gradient descent
  - SGD, momentum, …

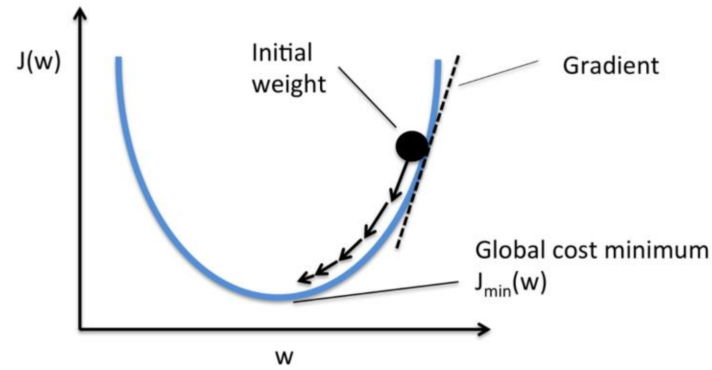
ReLU(z) = max{0, z}



“Rectified Linear Unit”
[Nair & Hinton, 2010]

Stochastic Gradient descent

Gradient descent



- Assumes computation with all data

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

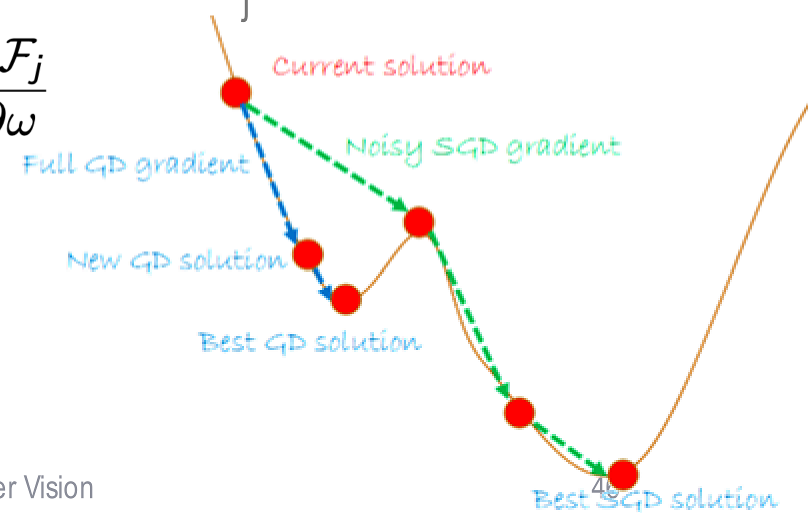
$$\omega_{t+1} = \omega_t - \lambda \frac{\partial \mathcal{F}}{\partial \omega}$$

- Converges to local minima if function is not convex

Stochastic Gradient descent

- Computation with random sample of data : batch B_j

$$\frac{\partial \mathcal{F}_j}{\partial \omega} = \frac{\partial}{\partial \omega} \sum_{i \in B_j} \|f(\omega, x_i) - y_i\|^2 \quad \omega_{t+1} = \omega_t - \lambda \frac{\partial \mathcal{F}_j}{\partial \omega}$$



- May help avoiding local minima
- But no convergence guarantee

Stochastic Gradient descent

SGD parameters

- Learning rate λ : see later
- Batch size B : increasing B reduces the variance of the gradient estimates and enables the speed-up of batch processing, but converges to 'standard' gradient descent

SGD with momentum

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

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

- Add a 'history' of gradient
- Can go through local barriers
- Accelerates if the gradient does not change much
- Reduces oscillations in narrow valleys
- 3rd parameter γ

SGD variants

Adaptive Learning Rate

- SGD rely a lot on learning rate
- Various strategies exist to adapt learning rate automatically
- AdaGrad, RMSProp, ADAM, …

Ex: AdaGrad

- Extension of SGD with momentum
- Accumulates gradient magnitudes
- Use it to decay learning rate
- Used with fix λ, usually 0.01

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

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

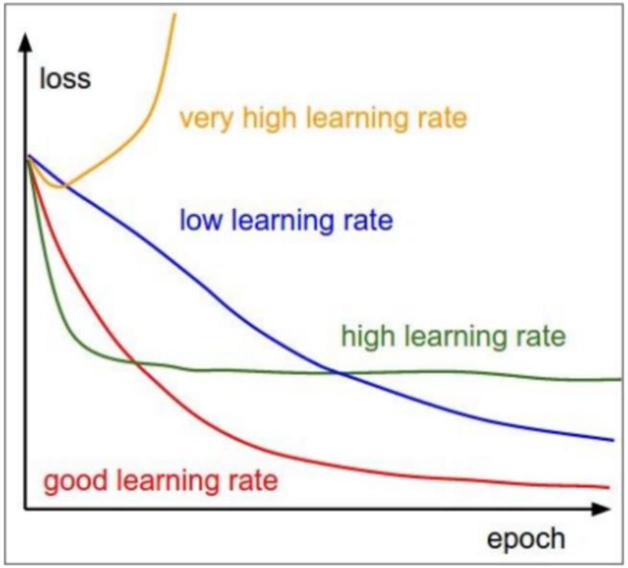
$$r_t = r_{t-1} + \left(\frac{\partial \mathcal{F}_j}{\partial \omega}\right)^2$$

$$\omega_{t+1} = \omega_t - \frac{\lambda}{\sqrt{\hat{r}_t} + \epsilon}\hat{m_{t+1}}$$

Learning rate

Learning Rate influences learning a lot

- High learning rate good at the beginning
- Low learning rate better at the end



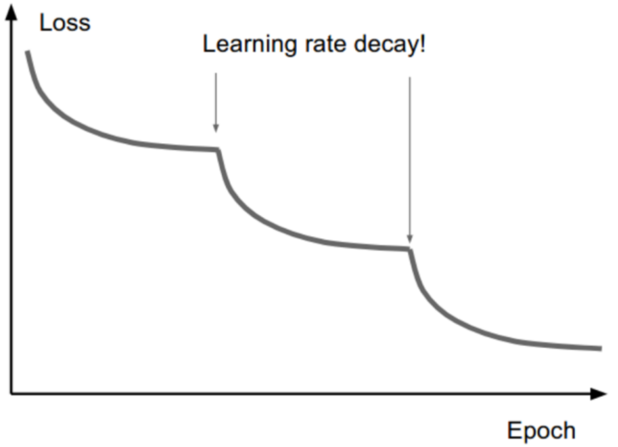
Scheduling learning rates

- Various approaches exist
- E.g. : Exponential

$$\lambda(t) = \lambda_0 \times e^{-kt}$$

- E.g. : 1/x

$$\lambda(t) = \lambda_0 / (1 + kt)$$



NB : Epoch = 1 pass of full dataset

Initialization

Initialize weights

- Initialization should put weights in area where gradients are large
  - Initialize to fixed value will lead to symmetries
  - Random initialization is better (usually Gaussian $(0,\sigma)$)
  - Weight should not be too big nor too small
- Various existing schemes
  - Xavier/Glorot
  - He/Kaiming for ReLu

fan = number of neurons
$fan_{avg} = (fan_{in} + fan_{out})/2$

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{avg}$
He	ReLU and variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

Xavier Glorot and Yoshua Bengio (2010): Understanding the difficulty of training deep feedforward neural networks. International conference on artificial intelligence and statistics.
Kaiming He, etal (2015): Delving Deep into Rectifiers:Surpassing Human-Level Performance on ImageNet Classification

Loss function

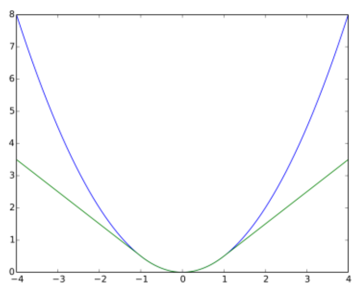
Compute error of the prediction

- L1 loss: for regression, ~ constant gradient, robust to outliers

$$L1 = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- L2 loss: for regression, gradient proportional to errors, sensitive to outliers

$$L2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$



- Huber Loss: Mix L1 (>1) and L2 (<1)

- Cross entropy : for categorization, transform network output to probabilities

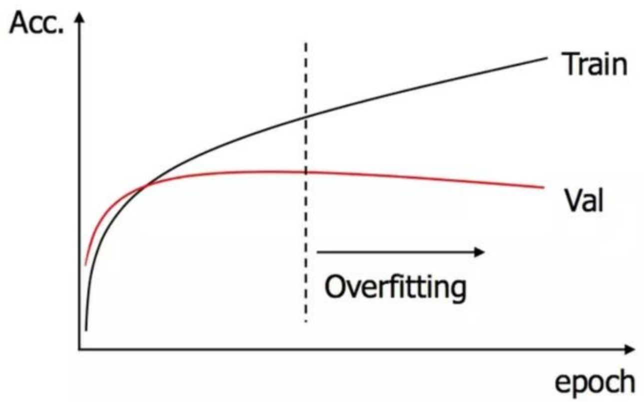
Softmax: $\hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$ $\quad L(\hat{y}, y) = -\sum_j y_j \log \hat{y}_j$

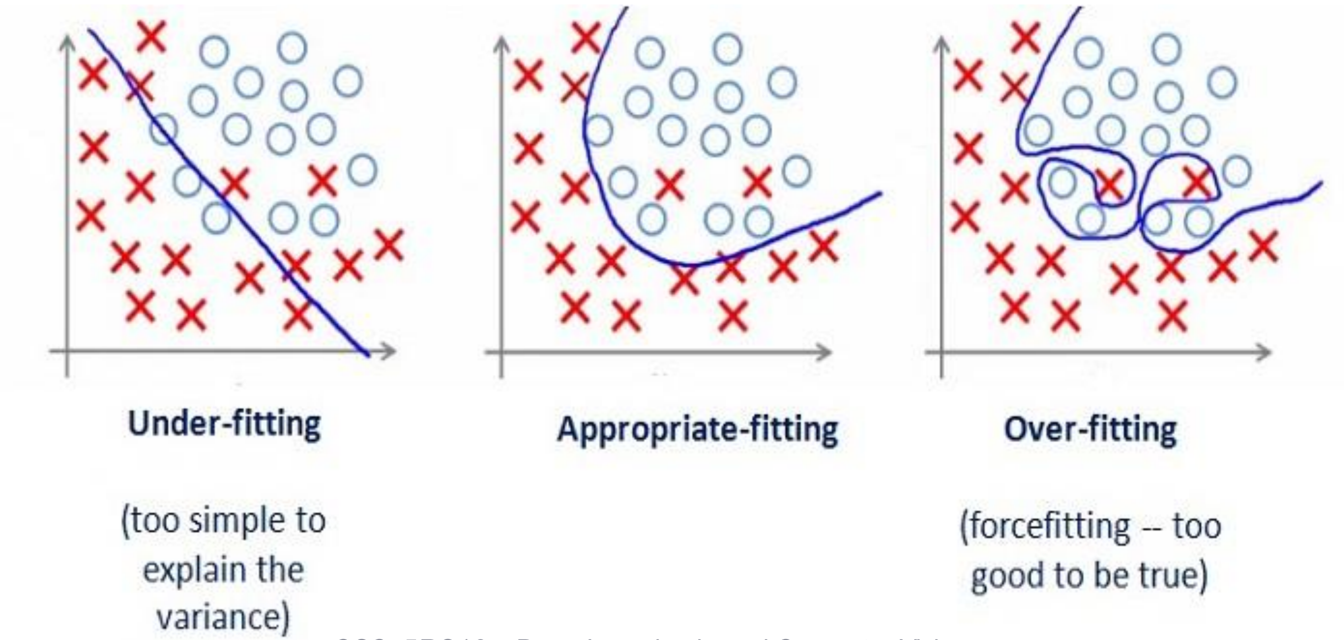
z_i: network output; $\hat{y}_i$: estimated prob of class i; y_i: true prob of class i;

Training Deep Networks

Avoid overfitting

- Training too much limits generalization
- Important to keep an eye on validation error
- Stop learning if validation error increase
- Using regularization also helps





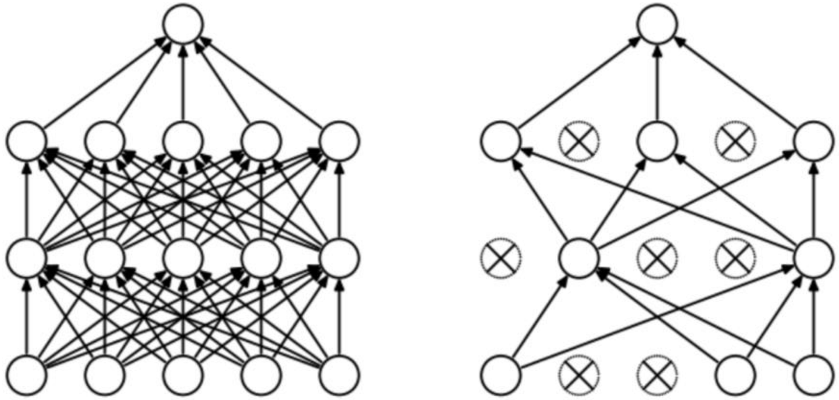
Training Deep Networks

Regularization

- Various ways to stabilize training and avoid overfitting
  - Weight decay
  - Dropout
  - Batch normalization
- Weight decay
  - Avoid overfitting / weigh explosion

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

- Dropout
  - Train while removing random connections
  - Force robustness to noise / redundancy



(a) Standard Neural Net

(b) After applying dropout.

Batch Normalization

Batch normalization for CNN

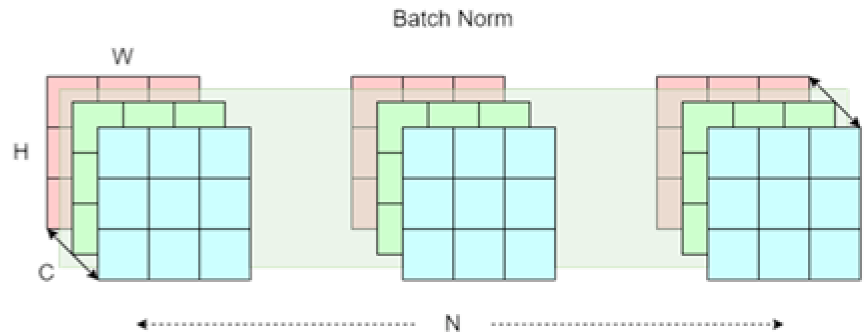
- Normalize data of a layer, for each batch, and output an affine transform with learned parameters γ,β

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

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

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

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



- Good empirical performances (no need for pretraining, dropout, …), reasons not completely clear
- Other normalization (layer, instance), for small batch size, transformers or RNN

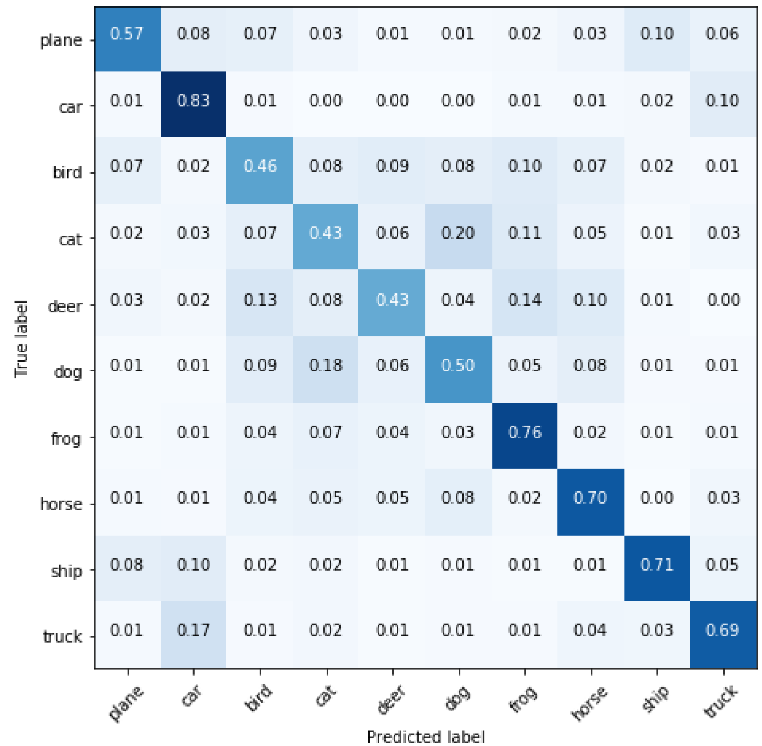
Reporting performances

Classification performance

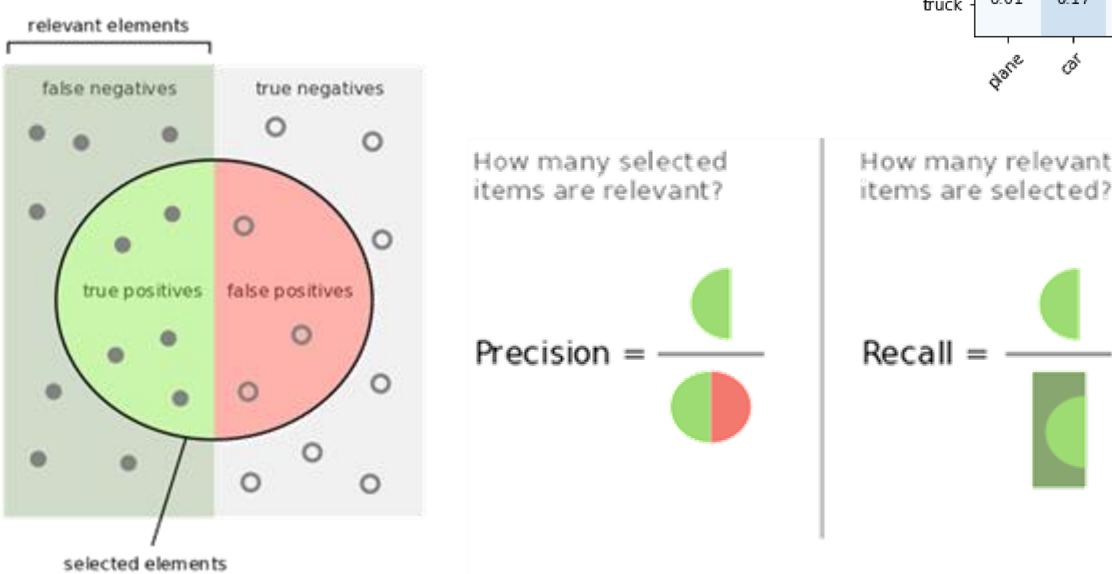
- Accuracy :

$$acc = \frac{correct\ predictions}{number\ of\ predictions}$$

- Confusion matrix
- For a class:
  - Precision/Recall



True label \ Predicted label	plane	car	bird	cat	deer	dog	frog	horse	ship	truck
plane	0.57	0.08	0.07	0.03	0.01	0.01	0.02	0.03	0.10	0.06
car	0.01	0.83	0.01	0.00	0.00	0.00	0.01	0.01	0.02	0.10
bird	0.07	0.02	0.46	0.08	0.09	0.08	0.10	0.07	0.02	0.01
cat	0.02	0.03	0.07	0.43	0.06	0.20	0.11	0.05	0.01	0.03
deer	0.03	0.02	0.13	0.08	0.43	0.04	0.14	0.10	0.01	0.00
dog	0.01	0.01	0.09	0.18	0.06	0.50	0.05	0.08	0.01	0.01
frog	0.01	0.01	0.04	0.07	0.04	0.03	0.76	0.02	0.01	0.01
horse	0.01	0.01	0.04	0.05	0.05	0.08	0.02	0.70	0.00	0.03
ship	0.08	0.10	0.02	0.02	0.01	0.01	0.01	0.01	0.71	0.05
truck	0.01	0.17	0.01	0.02	0.01	0.01	0.01	0.04	0.03	0.69



F1 Score = Harmonic mean of Precision and Recall

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Data for Deep Learning

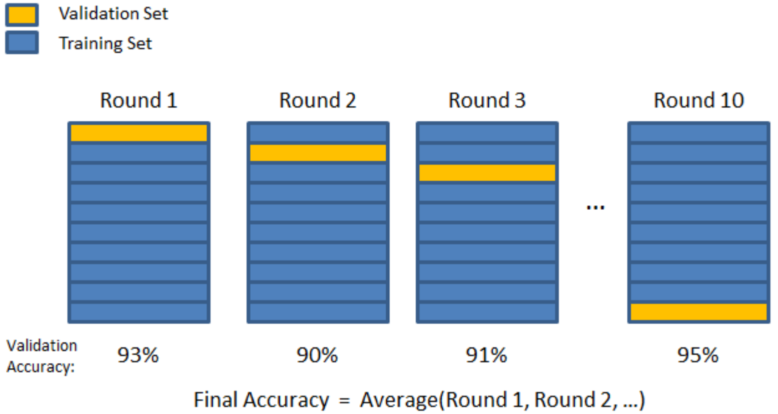
Datasets

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

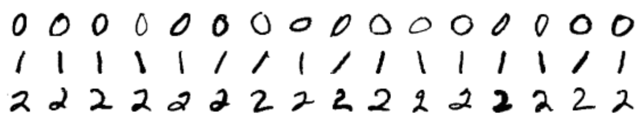
- When little data : split dataset in k sets
- Train on k-1, validate on remaning one
- Repeat k times
- Report mean performances



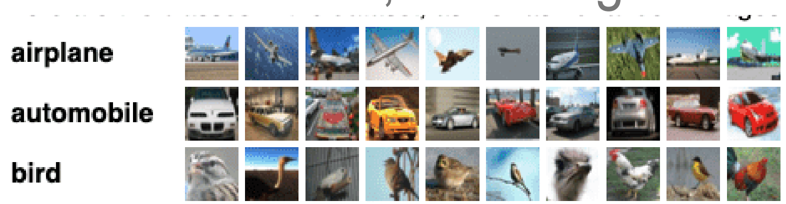
Datasets

Popular image classification datasets

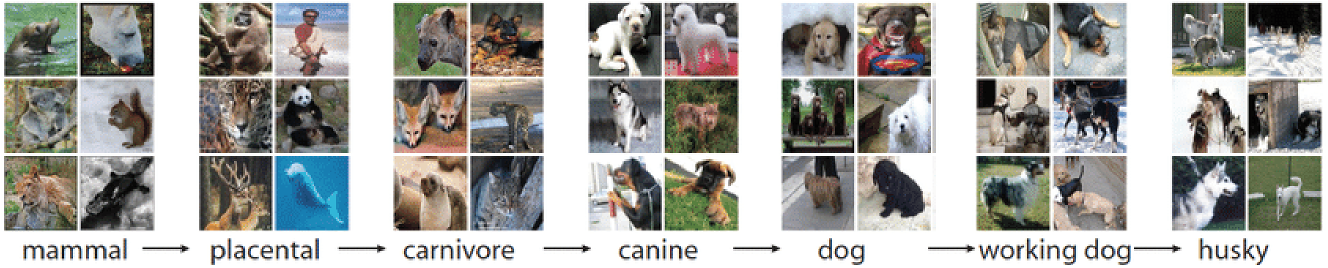
- MNIST : 28x28 gray level numbers, 60k images, variants : Fashion MNIST…



- CIFAR 10/100 : 32x32 color, 60k images



- ImageNet 21k : 21k categories, hierarchy, 14M images, very unbalanced



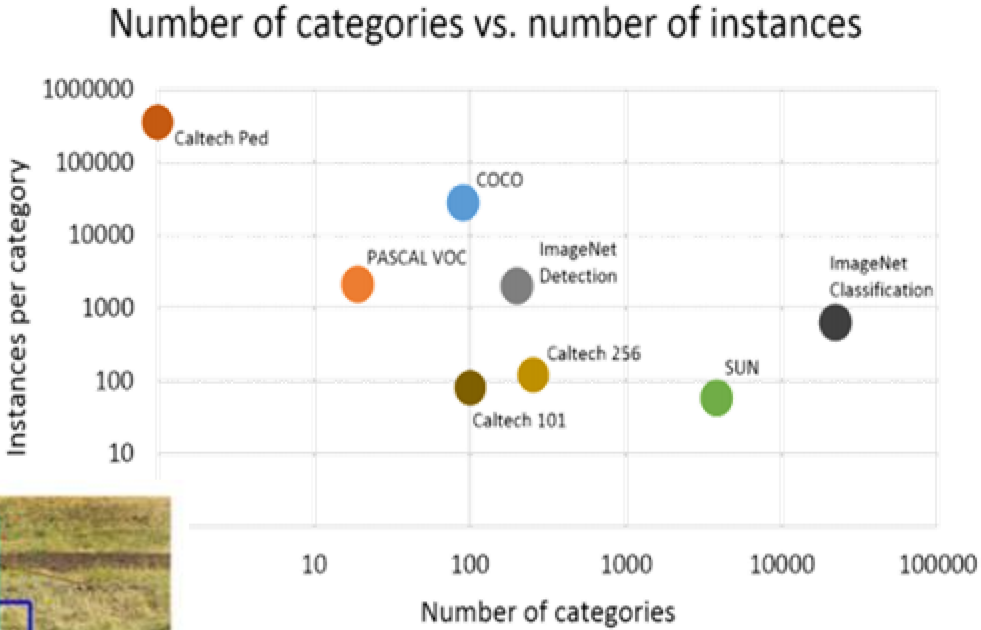
- ImageNet 1K : 1k categories, no hierarchy, 1.2M images

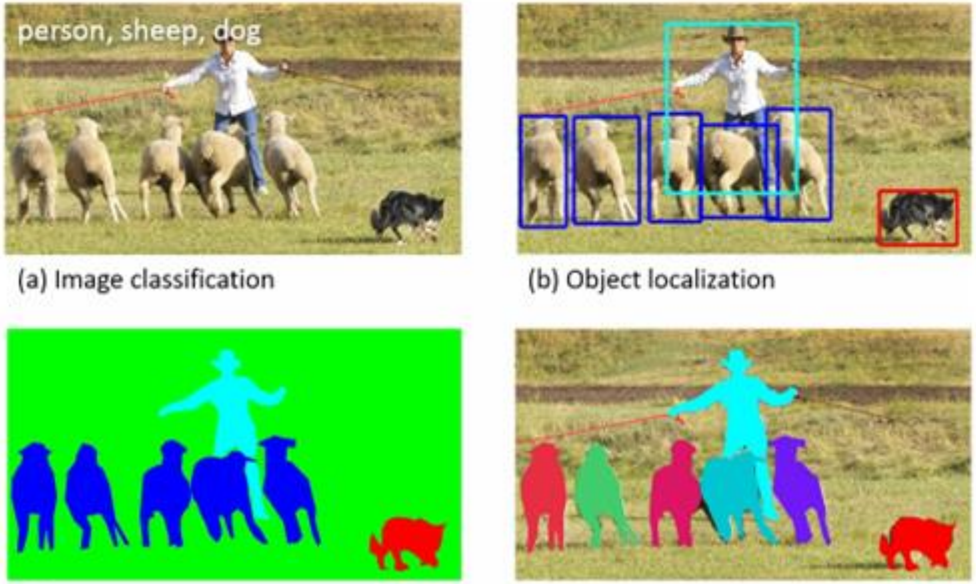
Datasets

Several large scale databases

- For various tasks
- Ex : Microsoft COCO

  Common Objects in Context





(a) Image classification (b) Object localization

(c) Semantic segmentation (d) This work

Summary

Training procedure (1/3)

- Create training / validation / test sets, or use existing dataset
- Normalize data
  - Substract mean (computed on training set)
  - Divide by std. dev. (computed on training set)
- Create your neural network structure
  - Manually by stacking layers (convolution, activation, pooling, Batch Norm, dense,…)
  - Or download existing structures (VGG, ResNet50, …)
- Initialize weights or download pretrained weights
  - E.g., Glorot initialization for personal NN
  - Or download ImageNet pretrained weights for existing structures

Training procedure (2/3)

- Choose a Loss function
  - For example for classification, use softmax + cross entropy.
- Select one variant of gradient descent (with momentum, ADAM, …)
  - Will use gradient to reduce the loss
- Define learning rate schedule
  - E.g. exponential decrease
- Define mini batch size
  - Bigger will smooth gradient noise -> allow larger steps -> learn faster
  - But too large mini-batches lead to problems (stuck in local min…)
  - Linked to memory size of GPUs

Training procedure (3/3)

- Overfit on few images
  - To check everything works: loss should go to 0 when trained on a few images
- Train
  - Refine hyperparameters, idealy use automatic parameter tuning (e.g. optuna)
- Deploy
  - Optimize network to fit on embedded platform

Deep Learning: summary

Deep learning works well

- Can be applied to lots of different tasks
- Very versatile approach
- Best performances in many vision tasks

But be aware of

- Very computationally intensive (can be optimized though)
- Need a lots of training data
- Quite sensitive parameters and open architectural possibilities

Fin