



ieeta instituto de engenharia electrónica e telemática de aveiro



universidade  
de aveiro

Departamento de Eletrónica, Telecomunicações e  
Informática

# Deep Learning

**LECTURE : CONVOLUTIONAL NEURAL NETWORKS (CONVNETS)**

**Petia Georgieva**  
**(petia@ua.pt)**

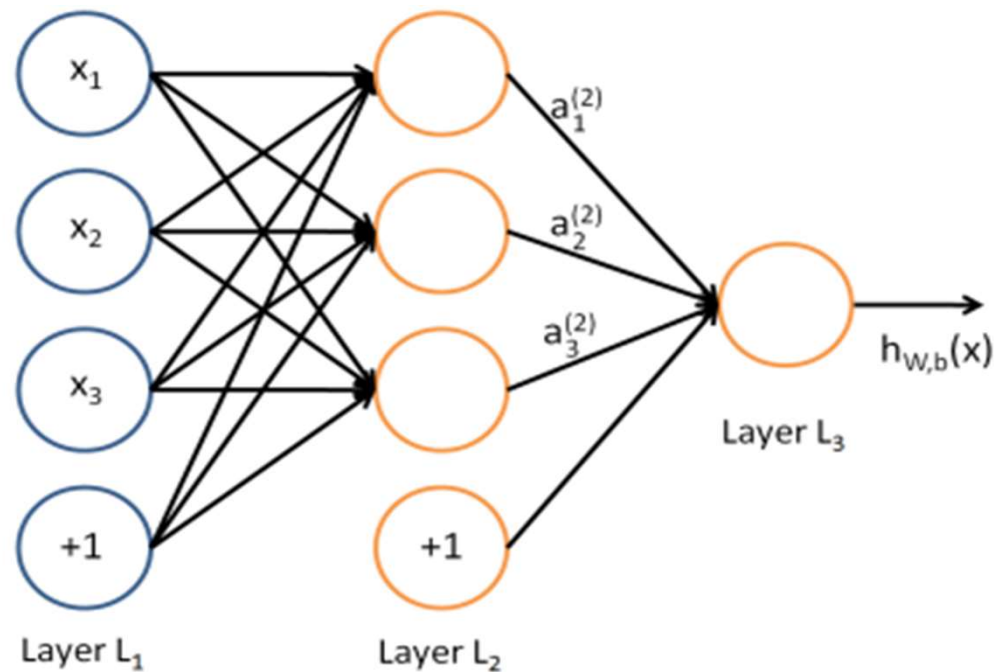
# **Outline**

**Deep Neural Networks**

**Convolutional Neural Networks (CNN) -  
Basic building blocks**

**Classic CNN – LeNet-5, AlexNet, VGG**

# Standard (shallow) Neural Network (recap)



$$(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$$

Standard NN parameter terminology:

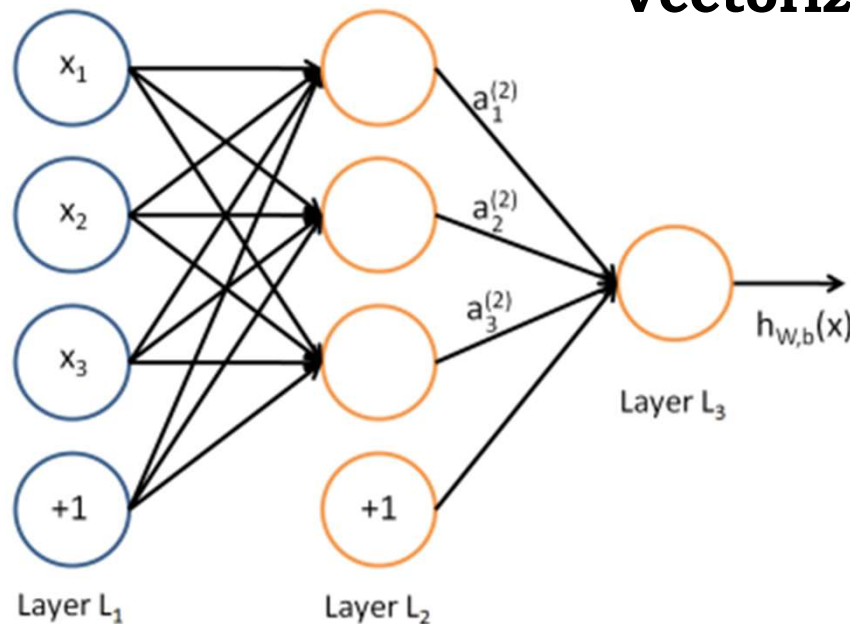
$W$  –matrix of network (model) weights

$b$ – bias/intercept (the weight of the input  $+1$ )

# NN forward computations (recap)

$$\begin{aligned}a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \\a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \\a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \\h_{W,b}(x) &= a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})\end{aligned}$$

## Vectorized computations:



$$\begin{aligned}z^{(2)} &= W^{(1)}x + b^{(1)} \\a^{(2)} &= f(z^{(2)}) \\z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\h_{W,b}(x) &= a^{(3)} = f(z^{(3)})\end{aligned}$$

$$\begin{aligned}z^{(l+1)} &= W^{(l)}a^{(l)} + b^{(l)} \\a^{(l+1)} &= f(z^{(l+1)})\end{aligned}$$

# NN backward computations (recap)

**Regularized Cost Function:**

$$\begin{aligned} J(W, b) &= \left[ \frac{1}{m} \sum_{i=1}^m J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left( W_{ji}^{(l)} \right)^2 \quad (1) \\ &= \left[ \frac{1}{m} \sum_{i=1}^m \left( \frac{1}{2} \|h_{W,b}(x^{(i)}) - y^{(i)}\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left( W_{ji}^{(l)} \right)^2 \end{aligned}$$

**Weight adaptation:**

$$\begin{aligned} W_{ij}^{(l)} &:= W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \\ b_i^{(l)} &:= b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b) \end{aligned}$$

**Weight Gradients  
(partial derivatives):**

$$\begin{aligned} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) &= \left[ \frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \right] + \lambda W_{ij}^{(l)} \\ \frac{\partial}{\partial b_i^{(l)}} J(W, b) &= \frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial b_i^{(l)}} J(W, b; x^{(i)}, y^{(i)}). \end{aligned}$$

# Why deep learning ?

Hardware get smaller.

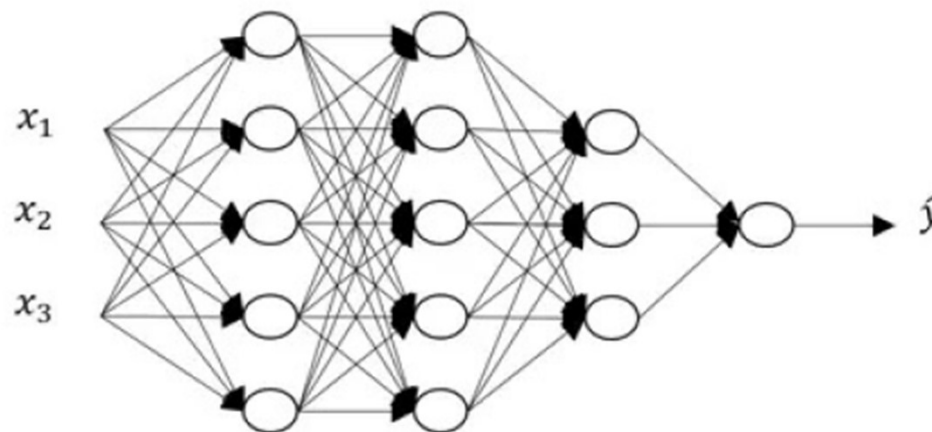
Sensors get cheaper, widely available IoT devices with high sample-rate.

Data sources: sound, vibration, image, electrical signals, accelerometer, temperature, pressure, LIDAR, etc.


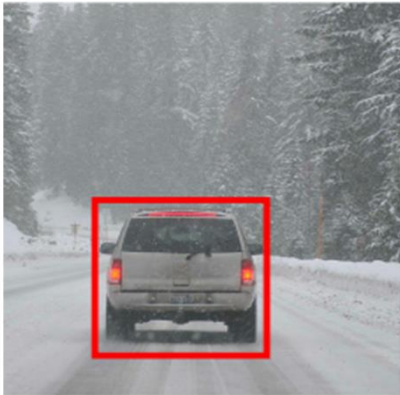

Big Data: Exponential growth of data, (IoT, medical records, biology, engineering, etc.)

How to deals with unstructured data (image, voice, text, EEG, ECG, etc.) => needs for feature extraction (data mining).

Deep Neural Networks: first extract the features, then solve ML tasks (classification, regression)



# CV: classification/localization/detection

Image classification	Classification & Localization	Detection
	 $b_x, b_y, b_h, b_w$	

**Image classification:** input a picture into the model and get the class label (e.g. person, bike, car, background, etc.)

**Classification & localization:** the model outputs not only the class label of the object but also draws a bounding box (the coordinates) of its position in the image.

**Detection:** the model detects and outputs the position of several objects.

# Deep Neural Networks (DNN)

DNNs have a large number of hidden layers. The first hidden layer finds simple functions like identifying the edges in the image. As we go deeper into the network these simple functions combine together to form more complex functions like identifying the face.

Common Examples:

## Face Recognition:

Image  $\Rightarrow$  Edges  $\Rightarrow$  Face parts  $\Rightarrow$  Faces  $\Rightarrow$  Desired face



## Audio recognition:

Audio  $\Rightarrow$  Low level audio wave features  $\Rightarrow$  Phonemes  $\Rightarrow$  Words  $\Rightarrow$  Sentences



# Why Convolution Learning ?

## Deep learning on large images

If the image has  $1000 \times 1000 \times 3$  (RGB) pixels = 3 million features (inputs)

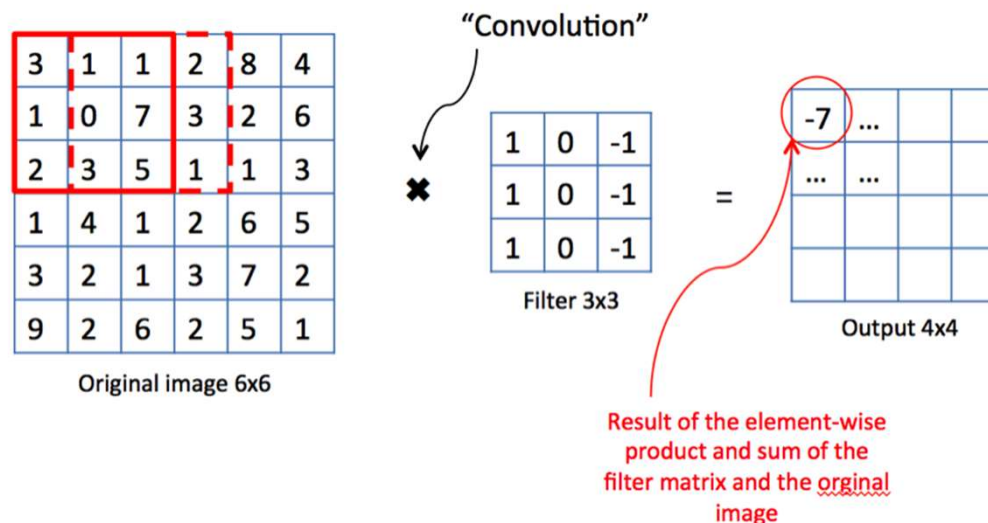
If the first hidden layer has 1000 nodes =>

The parameter matrix between the input and the hidden layer has  $(1000 \times 3 \text{million})$  3billion parameters.

**1<sup>st</sup> problem:** Difficult to get enough data to prevent model overfitting

**2<sup>nd</sup> problem:** Computational (memory) requirements to train such networks are not feasible.

## Solution: implement convolution operation



# OPERATION CONVOLUTION

1 <small>x1</small>	1 <small>x0</small>	1 <small>x1</small>	0	0
0 <small>x0</small>	1 <small>x1</small>	1 <small>x0</small>	1	0
0 <small>x1</small>	0 <small>x0</small>	1 <small>x1</small>	1	1
0	0	1	1	0
0	1	1	0	0

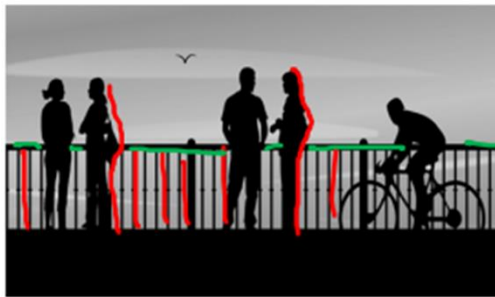
Image

4		

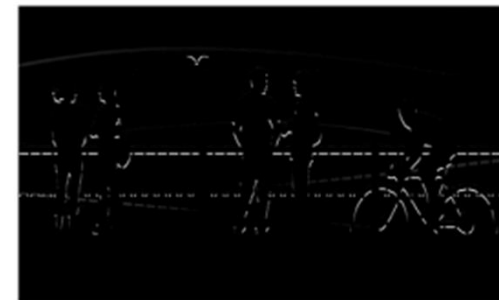
Convolved  
Feature

There is some inconsistency between convolution in math text books (signal processing view) and Deep learning literature. In classical convolution the filter has to be flipped (transposed) before the inner product and summing. In CNN, the flipping is skipped, mathematically this operation means cross-correlation. But by convention, in CNN this is called convolution.

# Detect horizontal/vertical edges



vertical edges



horizontal edges

# VERTICAL EDGES DETECTOR

**Illustrative example:**

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0


\*

1	0	-1
1	0	-1
1	0	-1


=

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

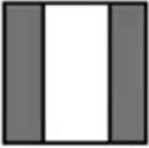
  



\*



=



**Detection of bright to dark transition (+30)**

# Hand-picked convolutional filters(kernels)

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

=> **Horizontal edge detector**

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

=> **Sobel filter**

$$\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$$

=> **Sharr filter**

# CONVOLUTIONAL FILTERS (KERNELS)

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

$w_1$	$w_2$	$w_3$
$w_4$	$w_5$	$w_6$
$w_7$	$w_8$	$w_9$


Hand-picking the filter values is difficult.

Why not let the computer to learn them?

Treat the filter numbers as parameters ( $w$ ), and let the computer learn them automatically.

Other than vertical and horizontal edges, such computer-generated filter can learn information from different angles (e.g.  $45^\circ$ ,  $70^\circ$ ,  $73^\circ$ ) and is more robust than hand-picking values.

By convention the conv filter is a square matrix with odd size (typically  $3 \times 3$ ;  $5 \times 5$ ;  $7 \times 7$ , also  $1 \times 1$ ).

It is nice to have a central pixel and it facilitates the padding.

# PADDING

$$\begin{bmatrix} 3 & 0 & 1 & 2 & 7 & 4 \\ 1 & 5 & 8 & 9 & 3 & 1 \\ 2 & 7 & 2 & 5 & 1 & 3 \\ 0 & 1 & 3 & 1 & 7 & 8 \\ 4 & 2 & 1 & 6 & 2 & 8 \\ 2 & 4 & 5 & 2 & 3 & 9 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -5 & -4 & 0 & 8 \\ -10 & -2 & 2 & 3 \\ 0 & -2 & -4 & -7 \\ -3 & -2 & -3 & -16 \end{bmatrix}$$

**Ex.** Take  $6 \times 6$  image, apply  $3 \times 3$  conv filter, get  $4 \times 4$  output matrix, because we shift the filter one row down or one column right and therefore we have  $4 \times 4$  possible positions for the  $3 \times 3$  filter to appear in the  $6 \times 6$  input matrix.

**In general:** given  $n \times n$  input matrix and  $f \times f$  filter matrix, the convolution operation will compute  $(n-f+1) \times (n-f+1)$  output matrix by applying one pixel up/down left/right rule.

**1<sup>st</sup> problem:** Shrink the matrix size as we continue to further apply convolution. The image will get very small if we have many convolution layers.

**2<sup>nd</sup> problem:** Pixels on the corner of the image are used only once while the pixels in the centre of the image are used many times. This is uneven, loose of inf.

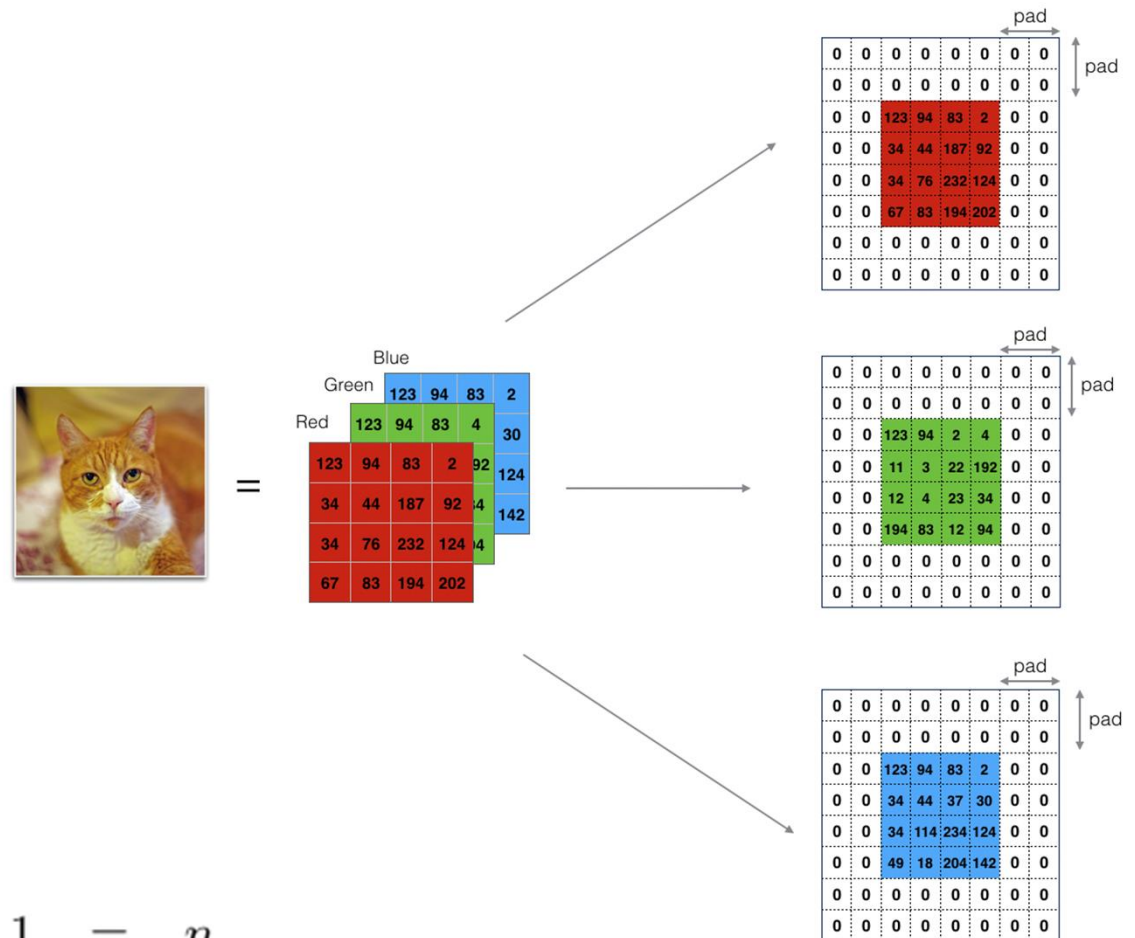


## SIMETRIC PADDING

Add  $p$  extra columns and rows at the image borders with 0 values (zero padding). Output matrix size:  $(n+2p-f+1) \times (n+2p-f+1)$

**“valid” convolution** => no padding

**“same” convolution** =>  
Pad so that output size is the same as the input size. Formula for choosing p  
(f is usually odd number!):



$$\begin{array}{rcl} n + 2p - f + 1 & = & n \\ 2p - f + 1 & = & 0 \\ 2p & = & f - 1 \\ p & = & (f - 1)/2 \end{array}$$



# STRIDED CONVOLUTION

$$\begin{bmatrix} 2 & 3 & 7 & 4 & 6 & 2 & 9 \\ 6 & 6 & 9 & 8 & 7 & 4 & 3 \\ 3 & 4 & 8 & 3 & 8 & 9 & 7 \\ 7 & 8 & 3 & 6 & 6 & 3 & 4 \\ 4 & 2 & 1 & 8 & 3 & 4 & 6 \\ 3 & 2 & 4 & 1 & 9 & 8 & 3 \\ 0 & 1 & 3 & 9 & 2 & 1 & 4 \end{bmatrix} * \begin{bmatrix} 3 & 4 & 4 \\ 1 & 0 & 2 \\ -1 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 91 & 100 & 83 \\ 69 & 91 & 127 \\ 44 & 72 & 74 \end{bmatrix}$$

**Stride:** how many pixels (steps) we shift to the right or down after each convolution.

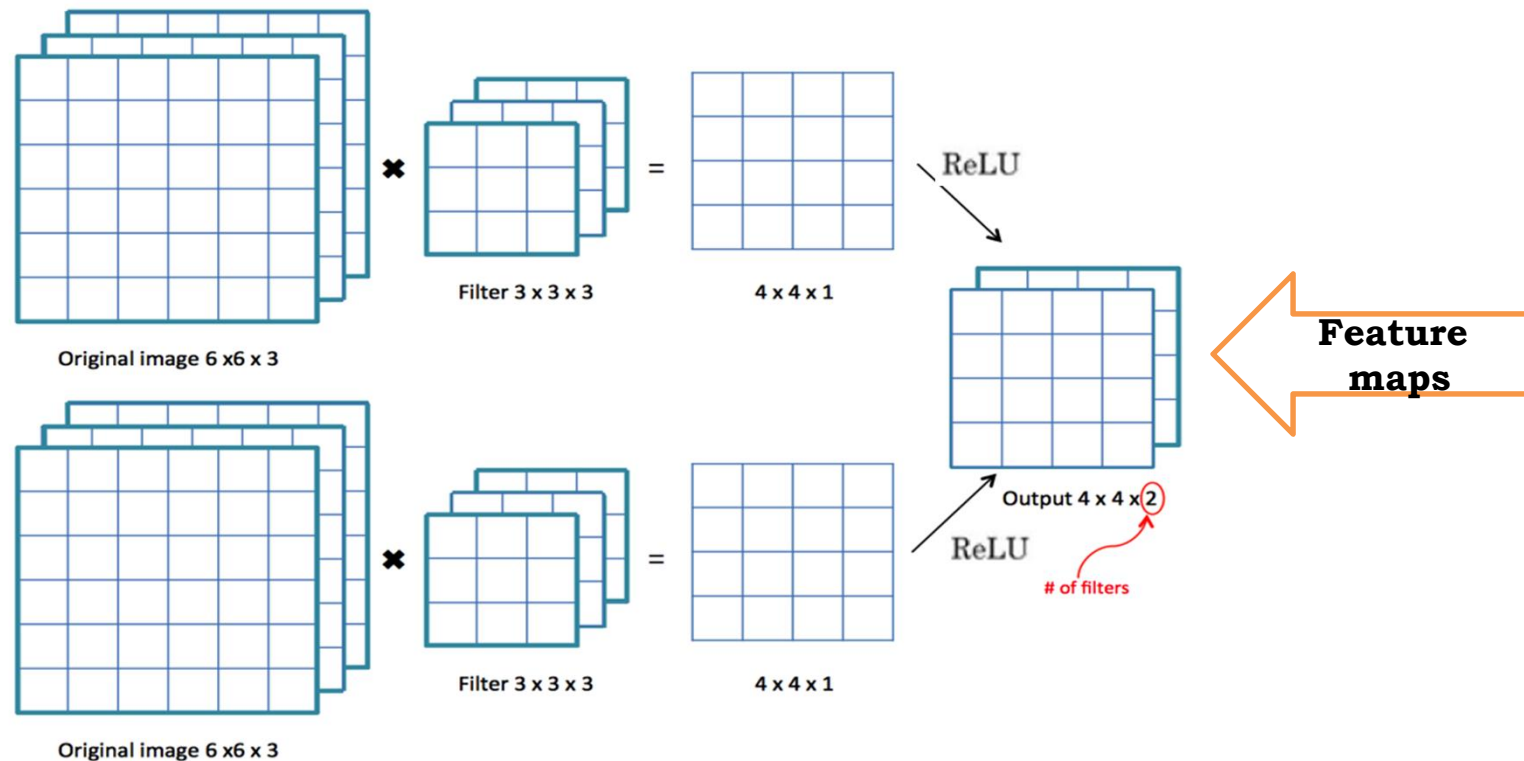
**In general:** given  $n \times n$  input matrix and  $f \times f$  filter with padding  $p$  and stride  $s$  the convolution operation will compute output matrix with size:

$$\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \text{ by } \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$$

If the formula computes a non-integer value => choose the closest lower integer.

**Ex.:** no padding ( $p=0$ ) , stride  $s=2$  =>  $(7 + 0 - 3)/2 + 1 = 4/2 + 1 = 3$  =>  $3 \times 3$  matrix

# Multiple Conv Filters over Volumes (3D filters)

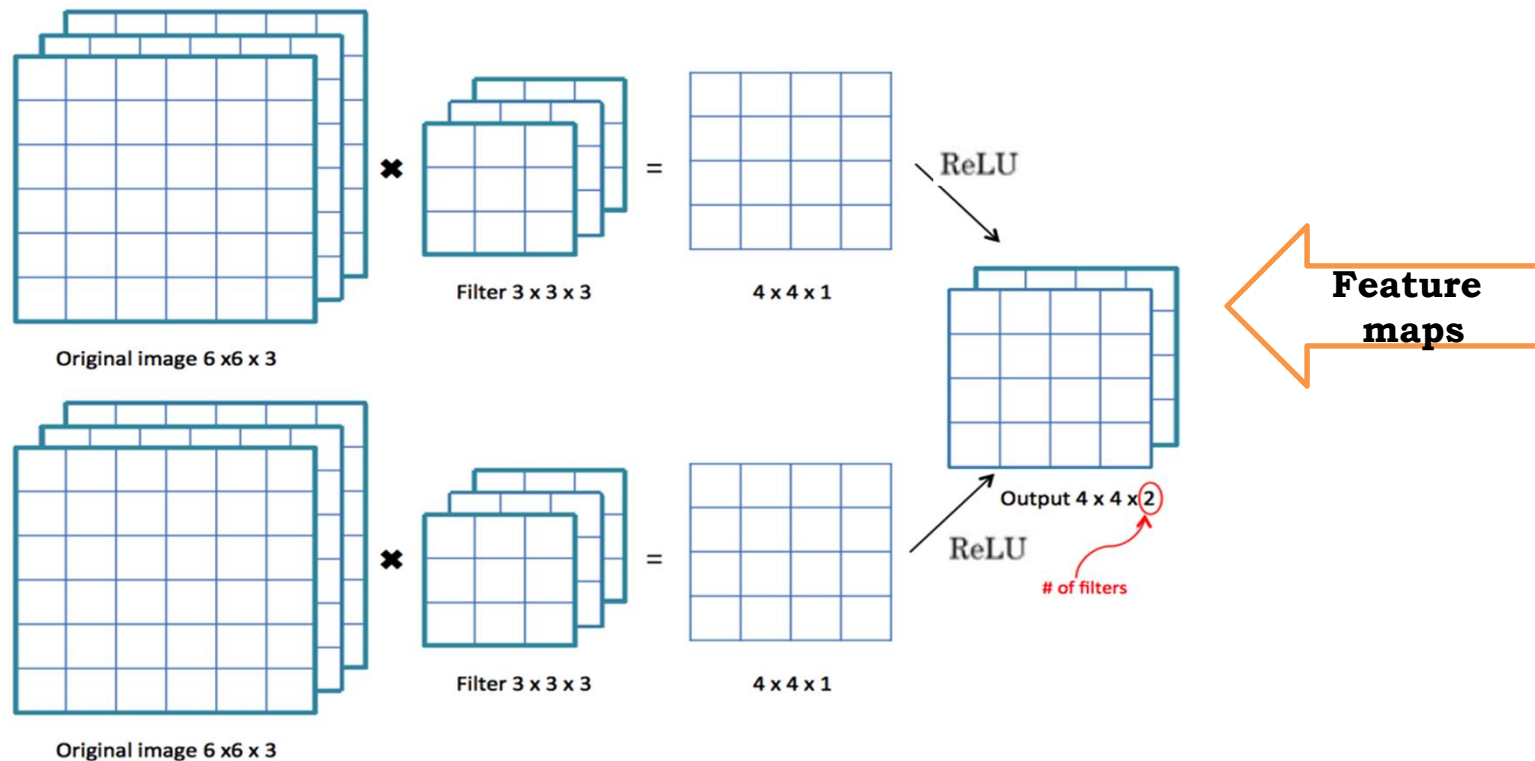


RGB images have 3 dimensions: height, width, and number of channels (3D volume).

The conv filter will be also a 3D volume :  $f \times f \times 3$  (number of channels has to be equal in the image and the filter)

$(n \times n \times n_c) \text{ image} * (f \times f \times n_c) \text{ filter} \Rightarrow (n-f+1) \times (n-f+1) \times n_{\text{filters}}$  (no padding)

# ONE CONV LAYER OF CNN



Different 3D filters (kernels) are applied to the 3D input image and the result matrices are stacked to form a 3D output volume.

After the convolution operation the result is passed through an activation function (e.g. ReLU, or sigmoid, linear, etc.).

The outputs of the conv layers are known as feature maps.

# SUMMARY OF NOTATION

As a summary, we write the following notation. If layer  $l$  is a convolution layer, we have  $f^{[l]}$  to be filter size,  $p^{[l]}$  is padding, and  $s^{[l]}$  is stride. We have input matrix  $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$ , where  $H$ ,  $W$ , and  $C$  denotes, “height”, “width”, and “convolution”, respectively. The output matrix would be  $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$  where the size is given

$$n^{[l]} = \lfloor \frac{n^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \rfloor \text{ for } n_H, n_W, \text{ respectively}$$

# Number of parameters in one layer

**Ex.** If you have 10 filters that are  $3 \times 3 \times 3$ , in one layer of a CNN, how many parameters does that layer have ?

Answer:  $3 \times 3 \times 3 = 27 + 1$  (bias)  $= 28 \times 10$  (filters)  $\Rightarrow$  in total 280 parameters

## Major property of CNN:

The number of parameters does not depend on the image size (or on the input from the previous layer).

Even in a very large image, we end up with a small number of parameters.

This make them less prone to overfitting.

The filters detect different features (horizontal, vertical edges, etc.)

# POOLING (POOL)

Average Pool

2	3	1	9
4	7	3	5
8	2	2	2
1	3	4	5

→

4	4.5
3.25	3.25

Average Pool with a 2 by 2 filter and stride 2.

Max Pool

2	3	1	9
4	7	3	5
8	2	2	2
1	3	4	5

→

7	9
8	5

Max-Pool with a 2 by 2 filter and stride 2.

Pooling operation reduce the size of the representation to speed up the computation and make the features more robust.

**Ex.** Divide the input in regions (e.g.  $2 \times 2$  filter), choose stride (e.g.  $s=2$ ), each output will be the max (**max pooling**) or the average (**average pooling**) from the corresponding regions.

Some intuition:

Large number means there is some strong feature (edge, eye ) detected in this part of the image, which is not present in another part.

Max Pool: whenever this feature is detected it remains preserved in the output.

# SUMMARY OF POOLING

The size of the regions ( $f$ ) and the stride ( $s$ ) are hyper-parameters.  
Common choice  $f=2$ ,  $s=2$  has the effect of shrinking the height and width of the representation by a factor of 2.

There are no parameters to learn (by the optimization method).

The pooling is done independently for each of the input channels.

Max pooling is much often used than average pooling.

There is no theoretical proofs why pooling works well.

It is just a fact in practice that this approach work well with some data sets.

# Softmax Regression Layer

Softmax Regression (SR) is a supervised learning algorithm, a generalization of logistic regression, suitable for multiclass classification ( $j=1,2,\dots,k$ )

SR estimates the probability that an example belongs to each of the  $k$  classes:

$$p(y^{(i)} = j | x^{(i)}; \theta) = \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^k e^{\theta_l^T x^{(i)}}}$$

SR outputs  $k$  dimensional vector with estimated probability for each class  $k$ :

$$h_{\theta}(x^{(i)}) = \begin{bmatrix} p(y^{(i)} = 1 | x^{(i)}; \theta) \\ p(y^{(i)} = 2 | x^{(i)}; \theta) \\ \vdots \\ p(y^{(i)} = k | x^{(i)}; \theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^k e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix}$$



# Softmax Regression Cost Function

$$J(\theta) = -\frac{1}{m} \left[ \sum_{i=1}^m \sum_{j=1}^k 1\{y^{(i)} = j\} \log \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^k e^{\theta_l^T x^{(i)}}} \right]$$

$1\{\}$  – indicator function;

$1\{\text{true statement}\}=1$

$1\{\text{false statement}\}=0$

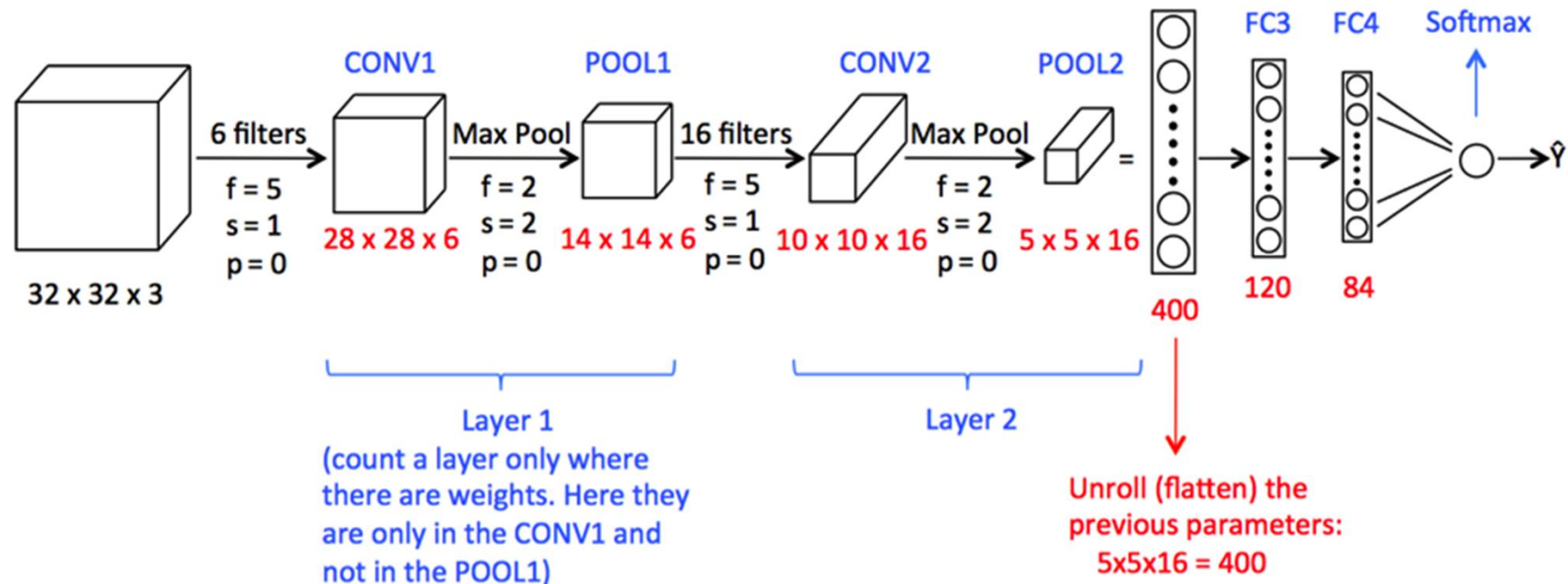
**Softmax cost function with regularization term:**

$$J(\theta) = -\frac{1}{m} \left[ \sum_{i=1}^m \sum_{j=1}^k 1\{y^{(i)} = j\} \log \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^k e^{\theta_l^T x^{(i)}}} \right] + \frac{\lambda}{2} \sum_{i=1}^k \sum_{j=0}^n \theta_{ij}^2$$

**Parameter (weight) update:**

$$\theta := \theta - \alpha \frac{d}{d\theta} J(\theta)$$

# Classical CNN Example: LeNet5\*



**\*LeCun et al., 1998, “Gradient-based learning applied to document recognition”.**

Original LeNet5 applied to handwritten digit recognition (grey scale images). Avg pooling, no padding, not softmax classifier; ReLU and sigmoid/tanh neurons in the Fully Connected (FC) layers.

The activation function is always present after the convolution, even if it is not drawn on the CNN diagram.

**General trend:** CNNs start with large image, then height and width gradually decrease as it goes deeper in the network, where as the number of channels increase.

# Convolution Benefits

Major advantages of conv layers over fully connected (FC) layers:

**(1) parameter sharing**

**(2) sparsity of connections**

**Ex.** Take  $32 \times 32 \times 3$  RGB image (3,072 inputs), using 6 filters ( $5 \times 5 \times 3$ ), we get  $28 \times 28 \times 6$  dimensional output (4,704 neurons). If we connect every neuron to the inputs (as in a classical DNN), the number of parameters would be about 14 million. This is a lot of parameters to train and this is just a small image.

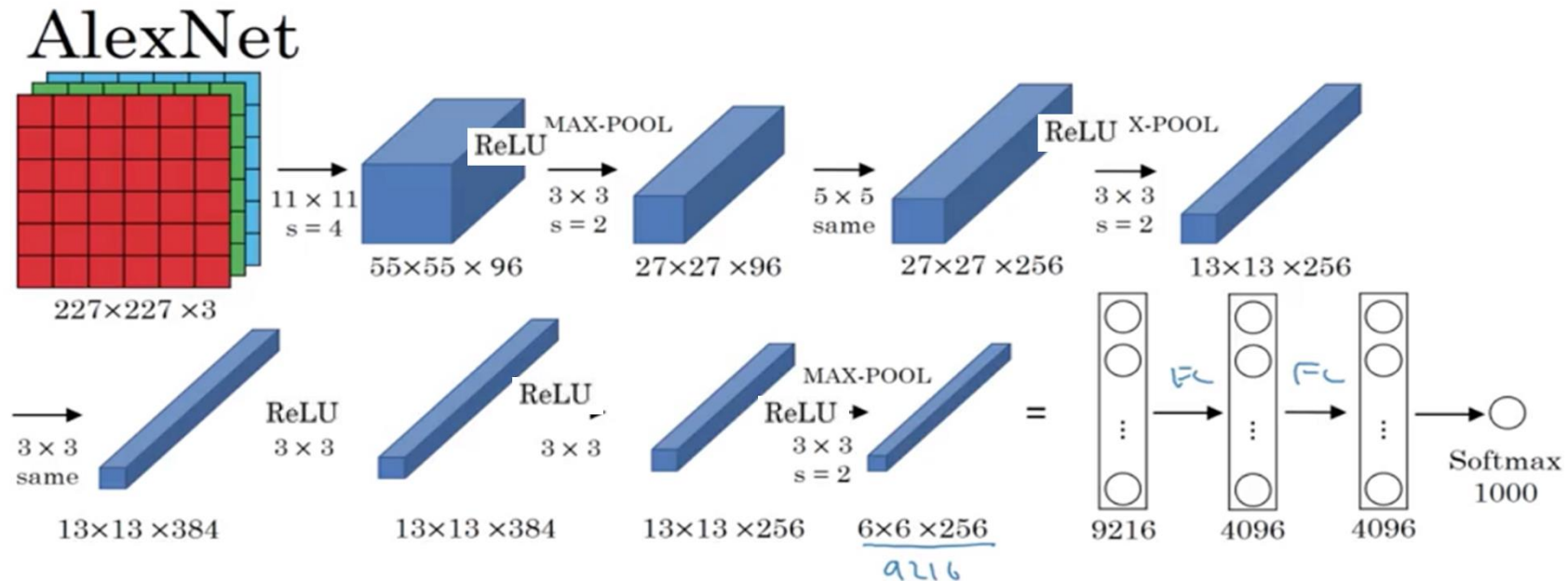
**Parameter sharing** is a feature detector (such as vertical edge detector) that is useful in one part of the image and is probably useful in another part of the image.

**Sparsity of connections** means that, in each layer, each output value depends only on a small number of inputs.

LeNet5 has only 60,000 parameters.

The conv layers have much less parameters than FC layers.

# AlexNet\*



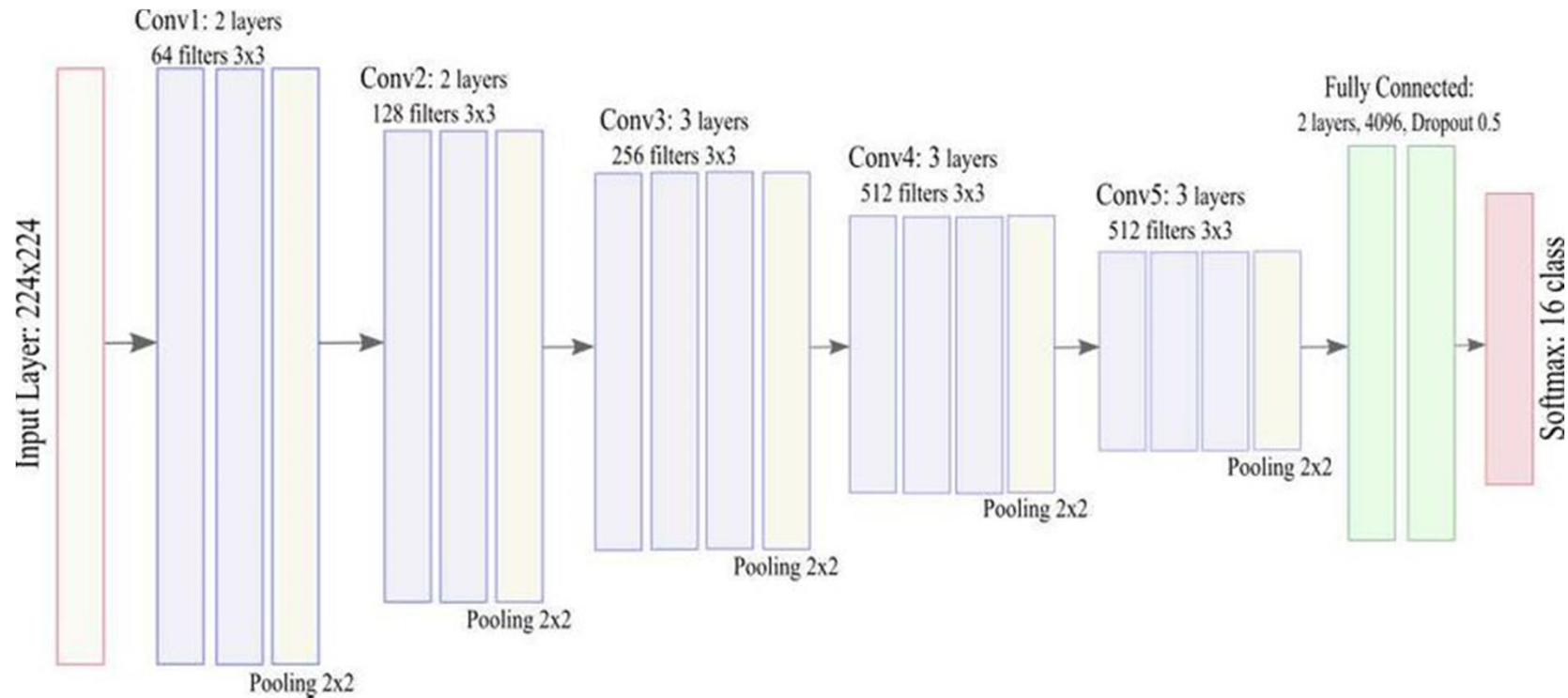
**\* Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, 2012, ImageNet classification with deep convolutional neural networks.**

5 conv layers, 3 FC layers with softmax output, 60 million parameters in total.

AlexNet applied to ImageNet LSVRC-2010 dataset to recognize 1000 different classes.

This paper convinced the CV community that DL really works and will have a huge impact not only in CV but also in speech/language processing.

# VGG-16 \*



**\* Karen Simonyan, Andrew Zisserman, 2015, Very Deep Convolutional Networks for Large-Scale Image Recognition**

VGG-16 has 16 layers (with weights !), 138 million parameters.

Unified architecture: All conv layers: (3x3) filters, s=1, same; All MaxPool =2x2, s=2.

Each convolution the height and width go down by a factor of 2, the channels go up by a factor of 2.

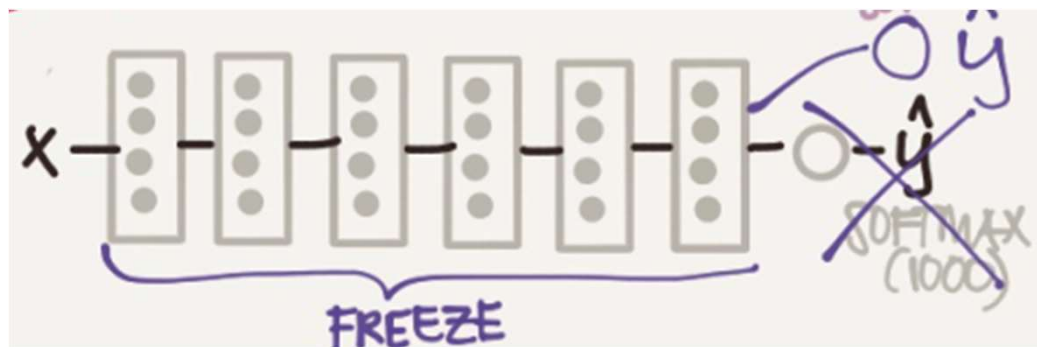
# Transfer Learning

- Starting from an open-source architecture from web (e.g. cloned from github) is faster than implementing code from scratch.
- Training may takes weeks/months, many GPUs, better use a pre-trained model (pre-trained parameters) => that is Transfer Learning (TL)

**Ex.:** your problem has 3 classes (car, pedestrian, neither).

You have a small training set.

- Take a DNN trained for 1000 classes.
- Substitute the last classification layer (softmax with 1000 outputs) ) with a softmax with 3 outputs.
- Freeze the parameters in all other layers, train only the softmax layer.
- Comp. trick: pre-compute and save on disc the features before softmax layer.



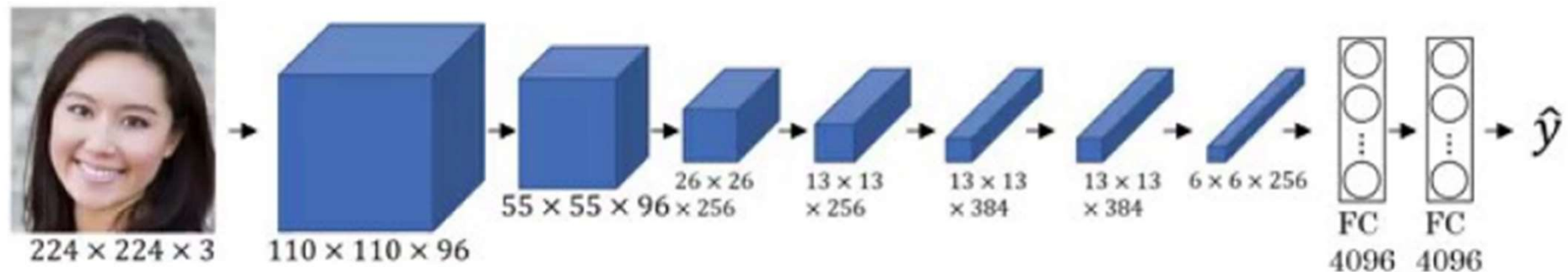
# Transfer Learning (cont)

**Ex. (cont):** If you have larger training set: freeze fewer layers, train latter layers.

The more data you have, the more layers you may train.

If you have big training set, you may use the trained DNN only at the initialization stage, starting not from random parameters but from the parameters of the trained DNN. Then update all weights during the optimization.

**Intuition:** Hidden layers earlier in the network extract much more general features not specific to the particular task.






# Data Augmentation



- Rotation,
- Random Noise
- Mirroring
- Random Cropping
- Color shifting: add distortions to the R (+20) G (-20) B (+20) channels
- Etc.

**Off-line:** generate all distortions and save the augmented data set.

**On-line:** common way of implementing data augmentation during training: CPU is constantly loading a stream of images coming from the hard disc and generate distortions to form mini-batches that are constantly passed to a Training algorithm (implemented on a different CPU or GPUs). The two processes (data augmentation and training run in parallel).

 Data augmentation also has hyper-parameters (what kind of distortion, how much distortion).



# Data vs. Hand-Engineering

Most ML applications lay somewhere in this spectrum:

**Little data** <-----> **Lots of data**

Lots of data: speech recognition

Reasonable large data: image recognition (cats or dogs)

Less data: object detection (bounding boxes)

**If Lots of data:** the best way to get good performance is building better learning system, playing with network architectures, big models (several layers) but simple algorithms, less hand-engineering.

**If Little data:** the best way to get good performance is hand-engineering – very difficult and skilful task that requires a lot of inside.

**ML applications have two sources of data:**

- Labelled data
- Hand-engineering features

CV tries to learn a really complex function (needs a lot of data), therefore historically the hand-engineered data was the main source of data.

# Tips for doing well on benchmarks/winning competitions

## **Ensembling:**

Train several (3, 5, 7) networks independently and average their outputs => 1-2 % better

**Multi-crop (e.g. 10-crop) at test time** (data augmentation of test images):  
Run classifier on multiple versions of test images and average results.

**NOTE:** Not recommended for production system deployed to serve costumers, it is more costly and memory expensive.