

Convolutional Neural Networks

Giacomo Boracchi,
DEIB, Politecnico di Milano
October, 21th, 2021

giacomo.boracchi@polimi.it
<https://boracchi.faculty.polimi.it/>

The Feature Extraction Perspective

The Feature Extraction Perspective

Images can not be directly fed to a classifier

We need some intermediate step to:

- Extract meaningful information (to our understanding)
- Reduce data-dimension

We need to extract features:

- The better our features, the better the classifier

The Feature Extraction Perspective

Input image



$$I_1 \in \mathbb{R}^{r_1 \times c_1}$$

Feature Extraction Algorithm

$$(d \ll r_1 \times c_1)$$

$$\mathbf{x} \in \mathbb{R}^d$$
A vertical vector \mathbf{x} represented as a column of four colored squares: white, light orange, dark orange, and black.

Classifier (NN)

$$y \in \Lambda$$

“wheel”

The Feature Extraction Perspective

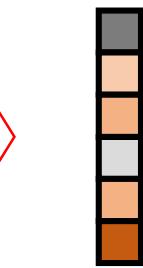
Input image



$$I_2 \in \mathbb{R}^{r_2 \times c_2}$$

Feature Extraction Algorithm

$$(d \ll r_2 \times c_2)$$



$$\mathbf{x} \in \mathbb{R}^d$$

Classifier (NN)

“castle”

$$y \in \Lambda$$

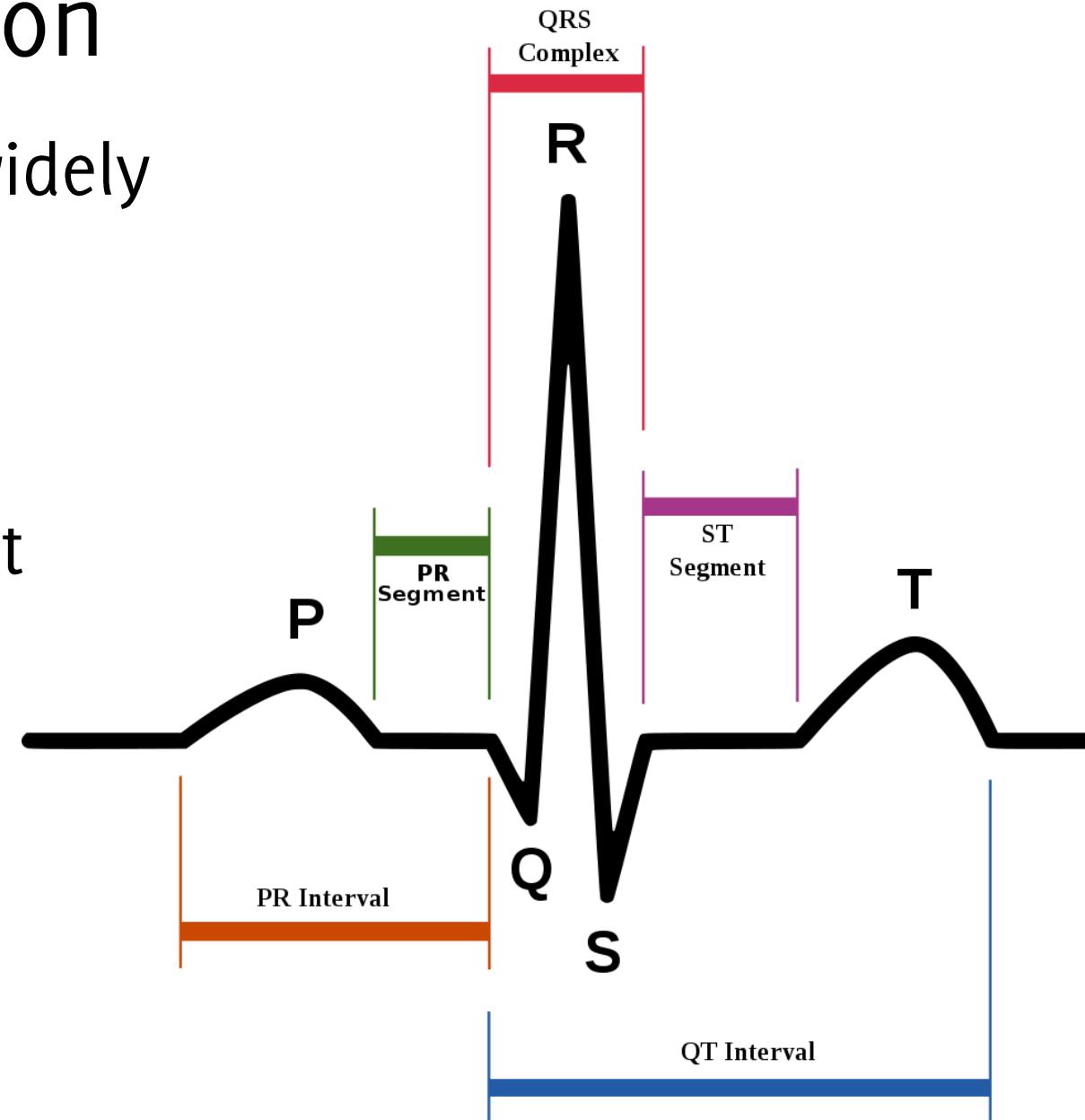
Hand-Crafted Features

Example: ECG classification

Heartbeats morphology has been widely investigated

Doctors know which patterns are meaningful for classifying each beat

Features are extracted from landmarks indicated by doctors:
e.g. QT distance, RR distance...

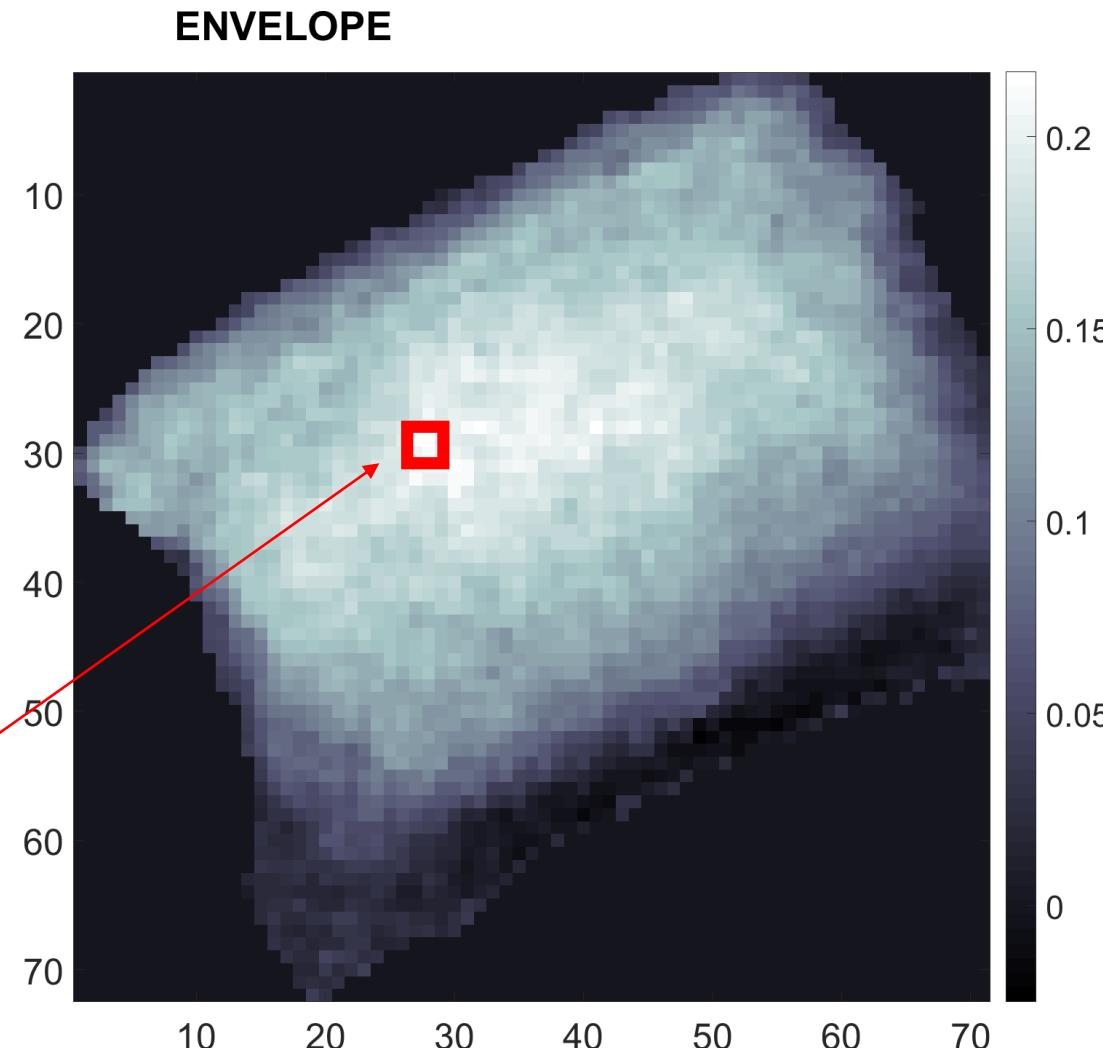


An Illustrative Example: Parcel Classification

Images acquired from a RGB-D sensor:

- No color information provided
- A few pixels report depth measurements
- Images of 3 classes
 - ENVELOPE
 - PARCEL
 - DOUBLE

Envelop height at that pixel

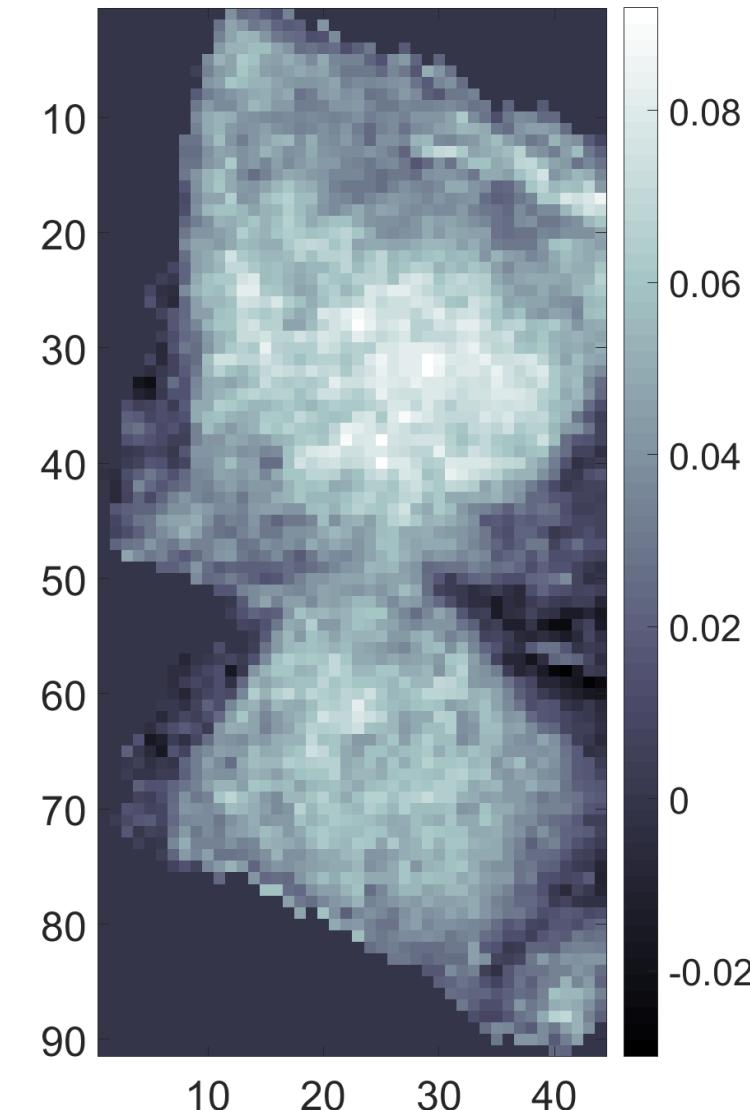


An Illustrative Example: Parcel Classification

Images acquired from a RGB-D sensor:

- No color information provided
- A few pixels report depth measurements
- Images of 3 classes
 - ENVELOPE
 - PARCEL
 - DOUBLE

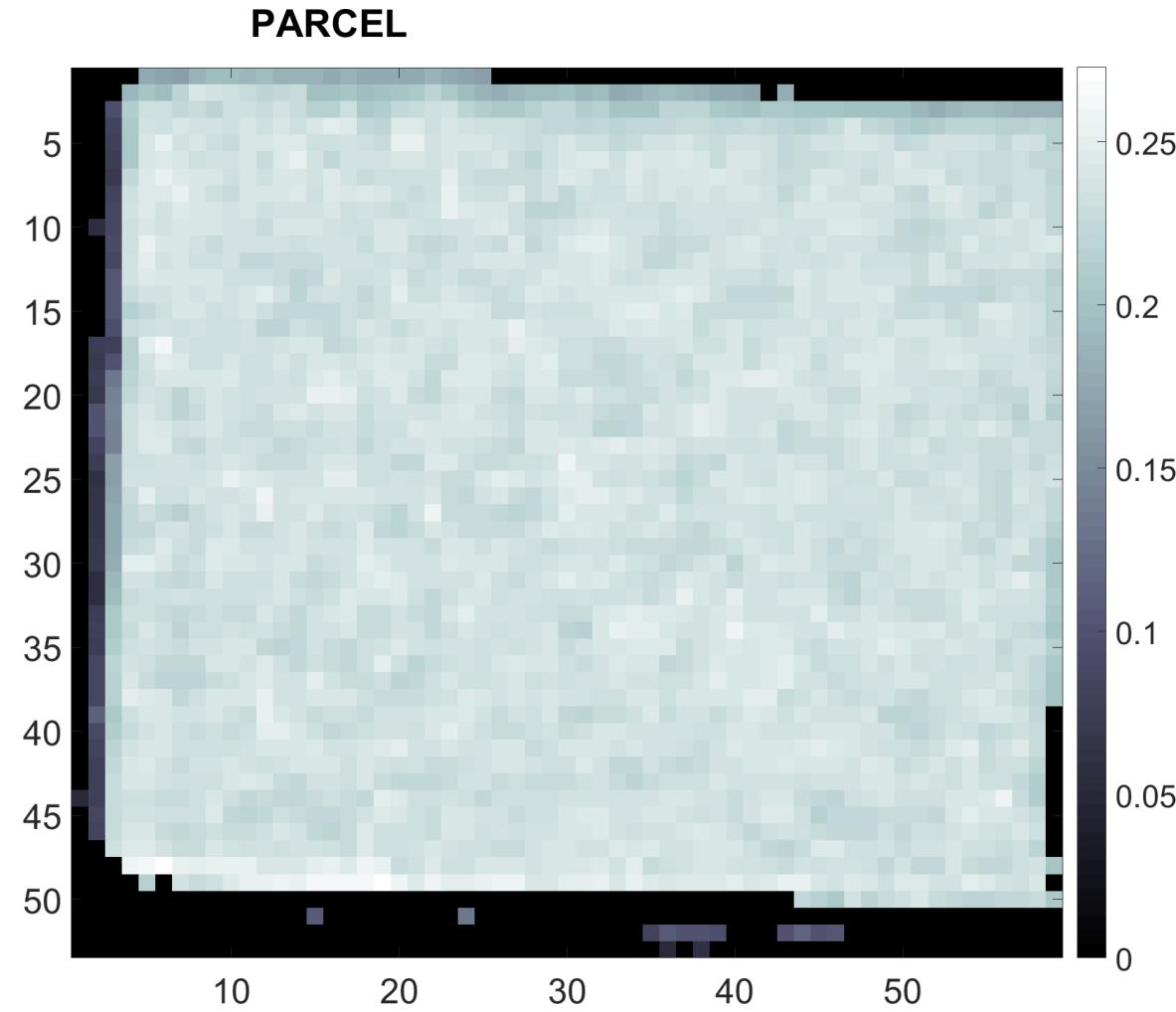
DOUBLE



An Illustrative Example: Parcel Classification

Images acquired from a RGB-D sensor:

- No color information provided
- A few pixels report depth measurements
- Images of 3 classes
 - ENVELOPE
 - PARCEL
 - DOUBLE



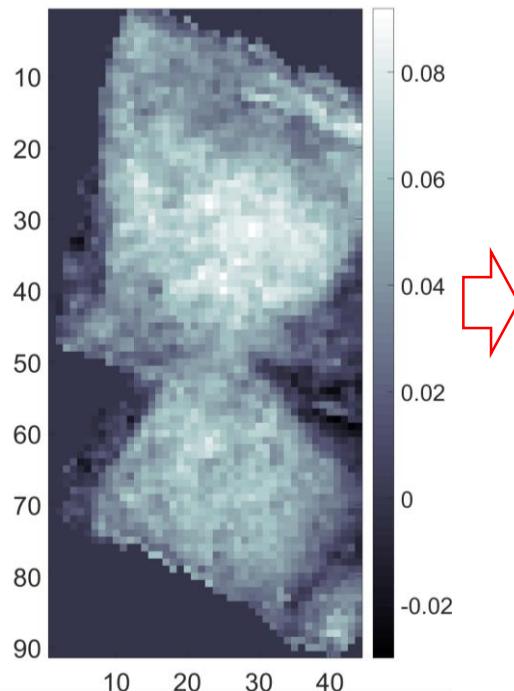
Example of Hand-Crafted Features

Example of features:

- Average height
- Area (coverage with nonzero measurements)
- Distribution of heights
- Perimeter
- Diagonals

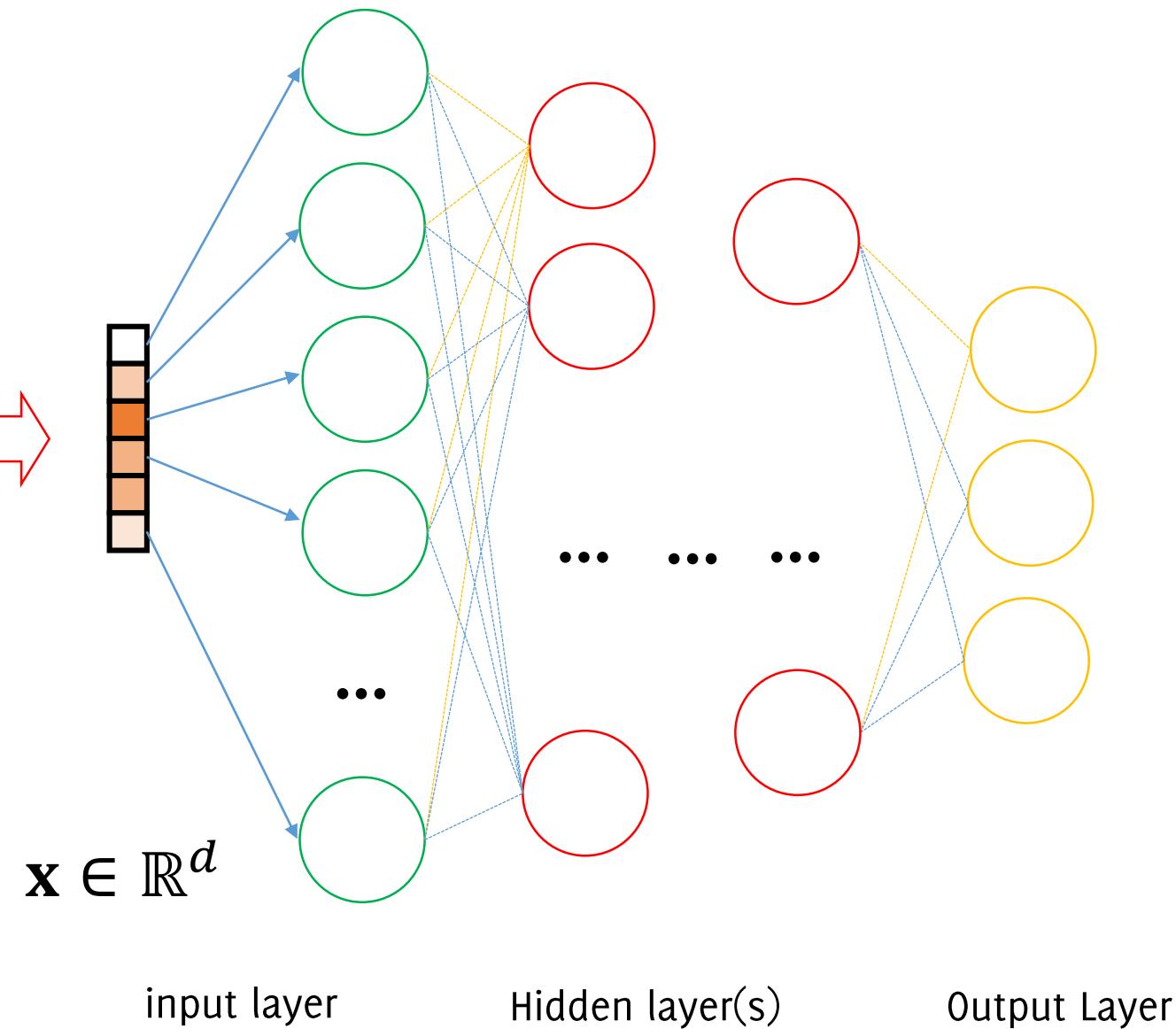
Neural Networks

Input image



$$I_1 \in \mathbb{R}^{r_1 \times c_1}$$

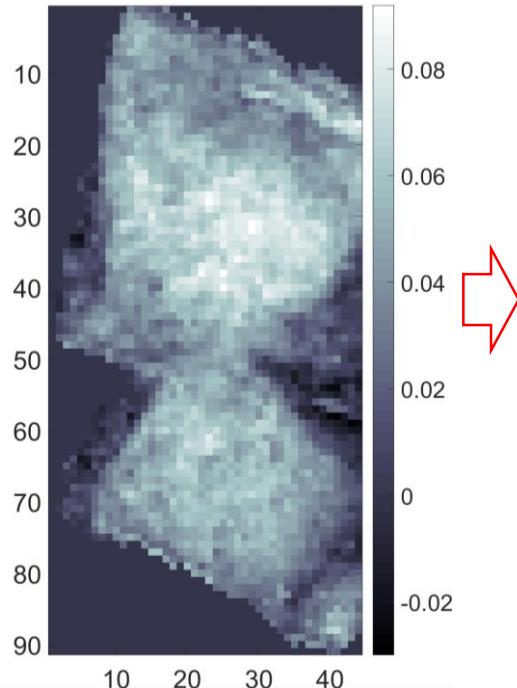
Feature Extraction Algorithm



Neural Networks

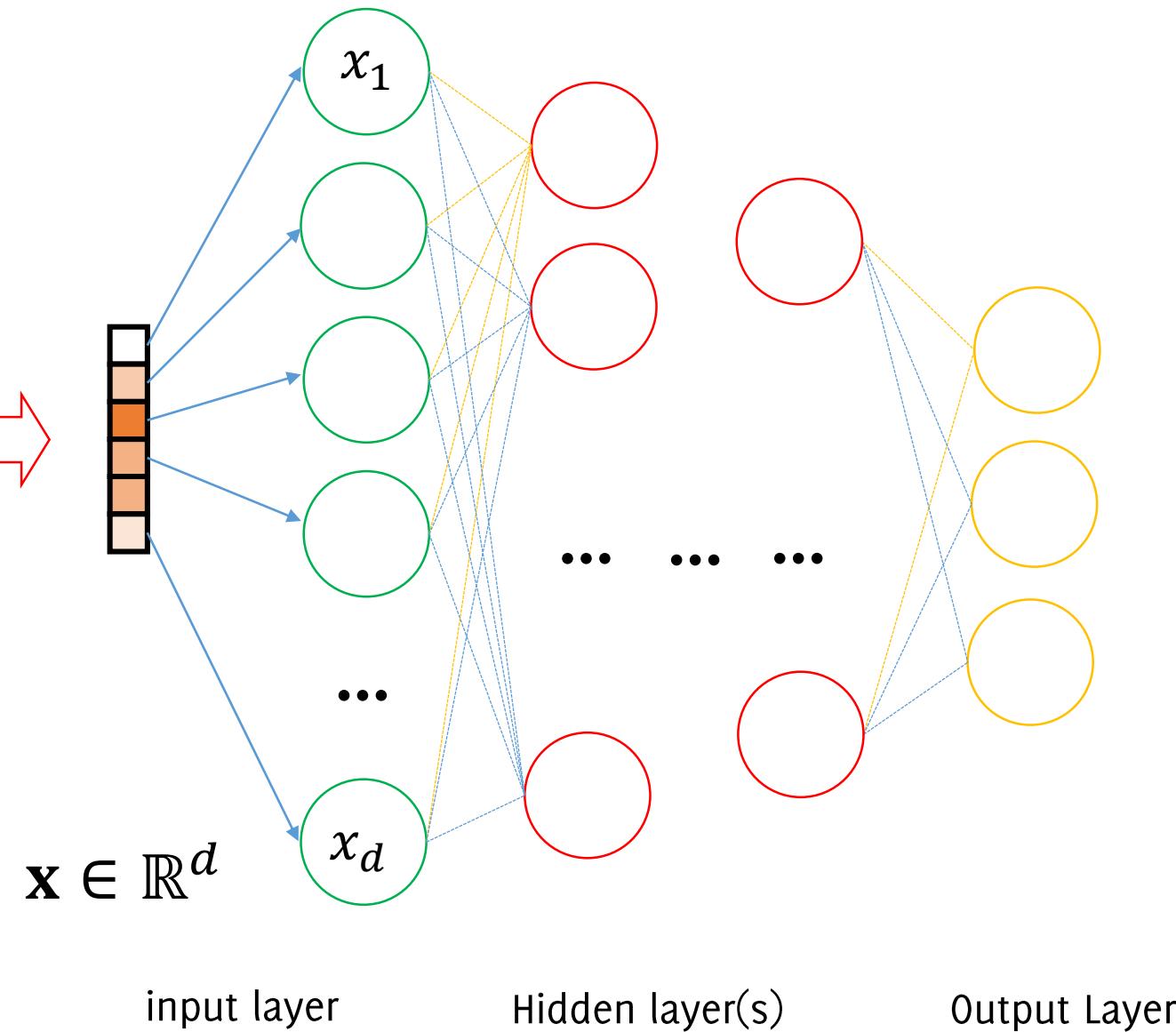
Input layer: Same size of the feature vector

Input image



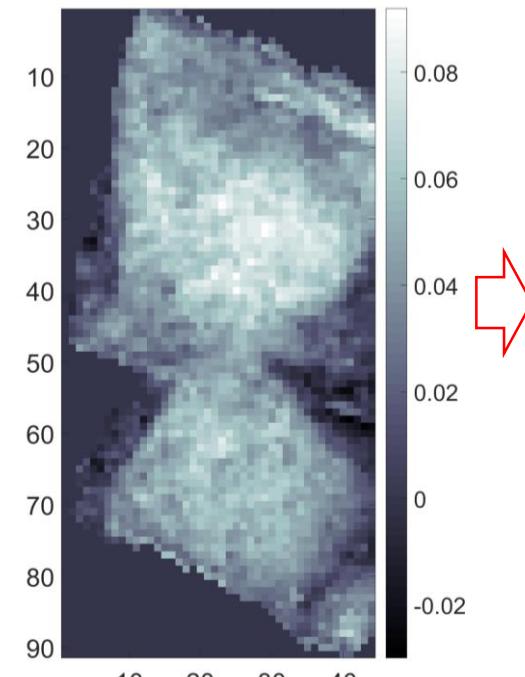
$$I_1 \in \mathbb{R}^{r_1 \times c_1}$$

Feature Extraction Algorithm

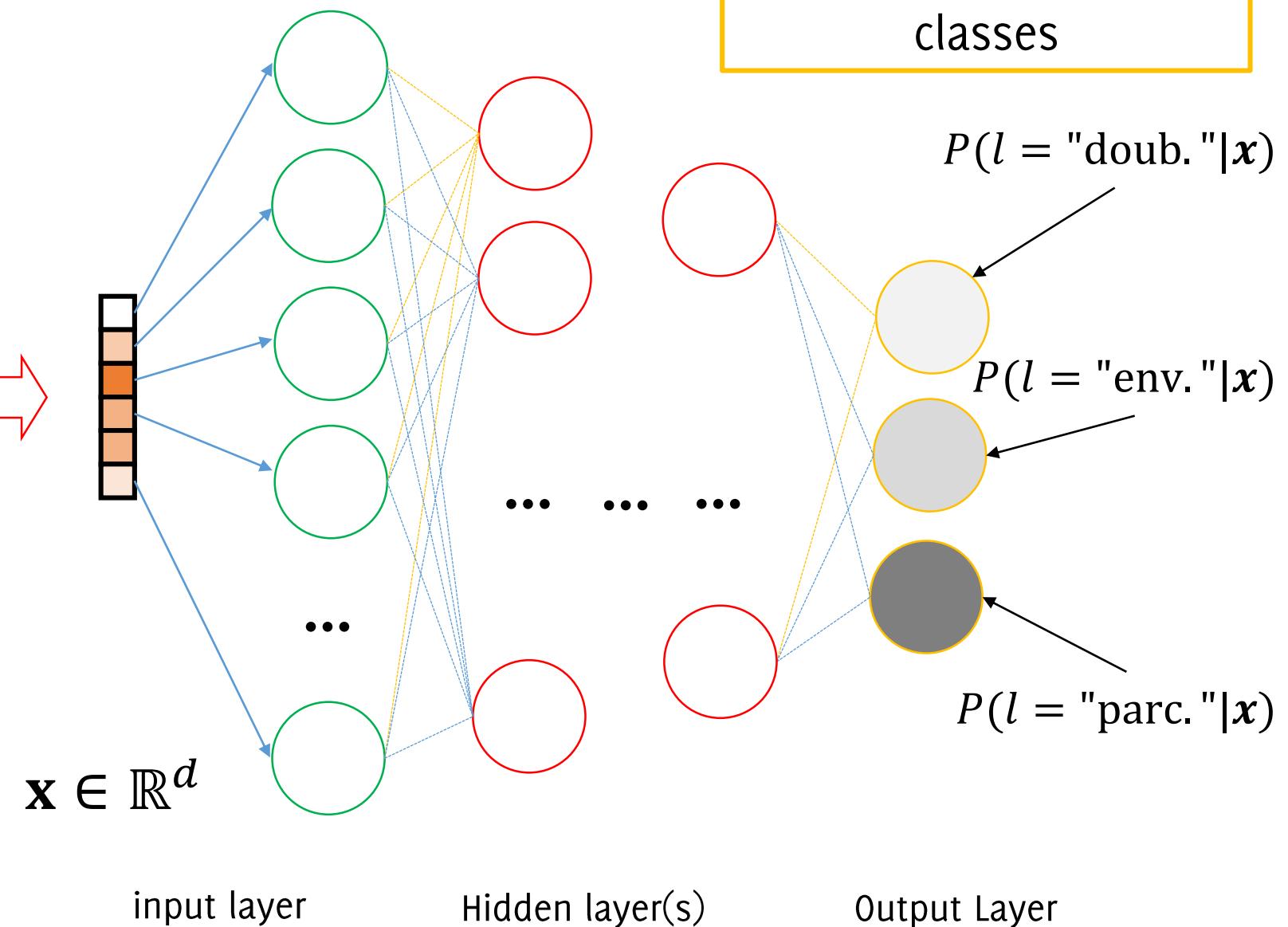


Neural Networks

Input image

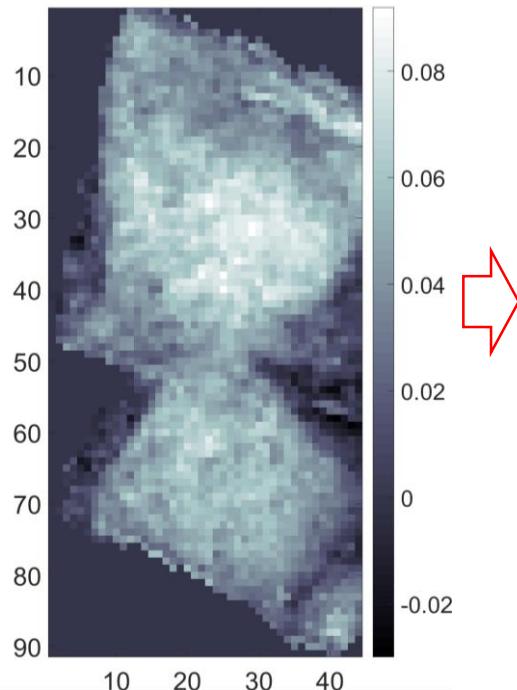


Feature Extraction Algorithm



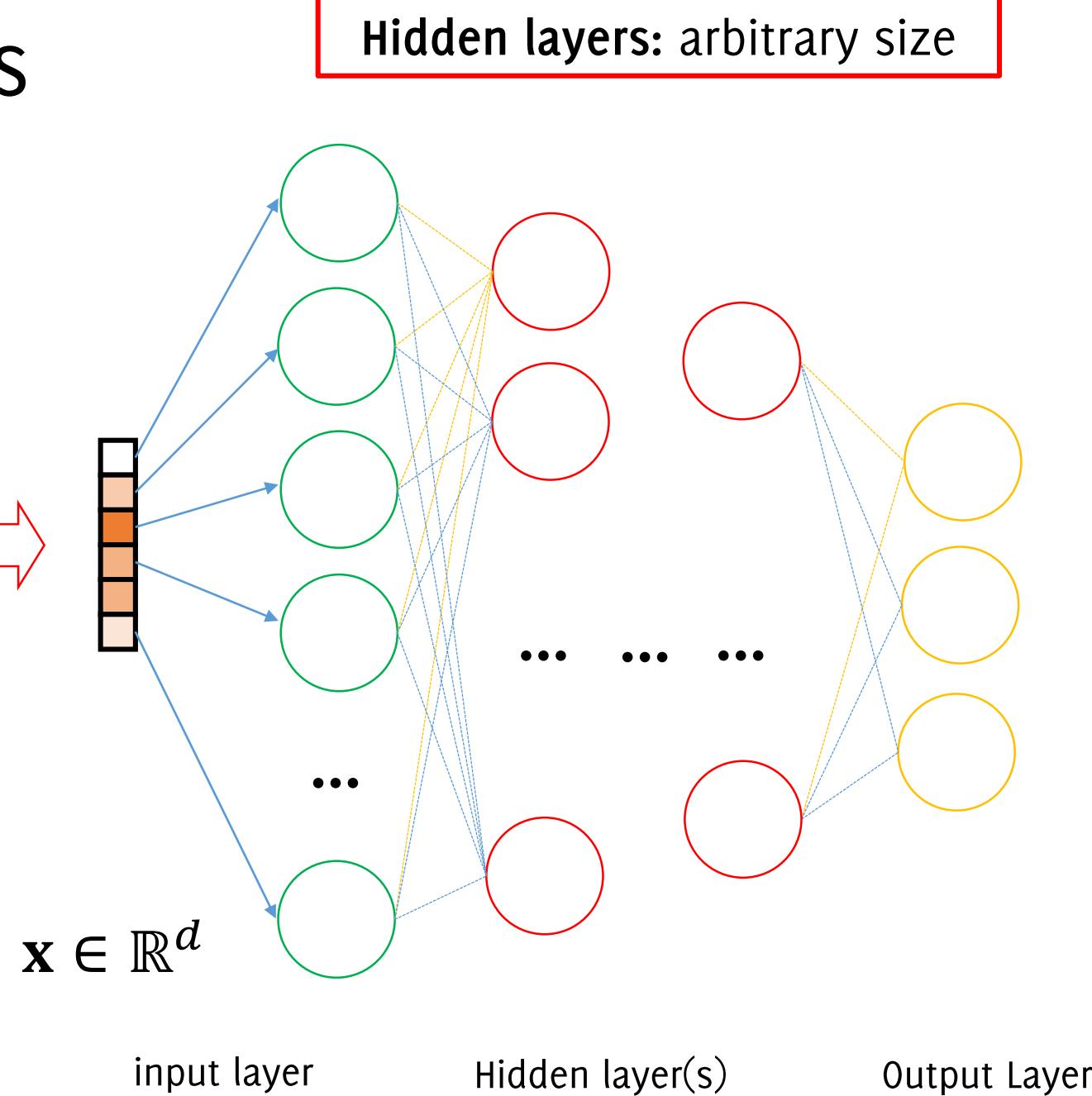
Neural Networks

Input image



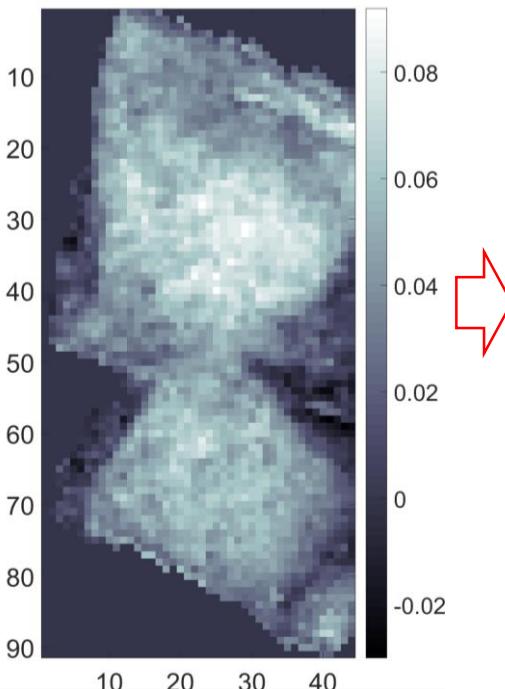
$$I_1 \in \mathbb{R}^{r_1 \times c_1}$$

Feature Extraction Algorithm



Neural Networks

Input image



$$I_1 \in \mathbb{R}^{r_1 \times c_1}$$

Feature Extraction Algorithm

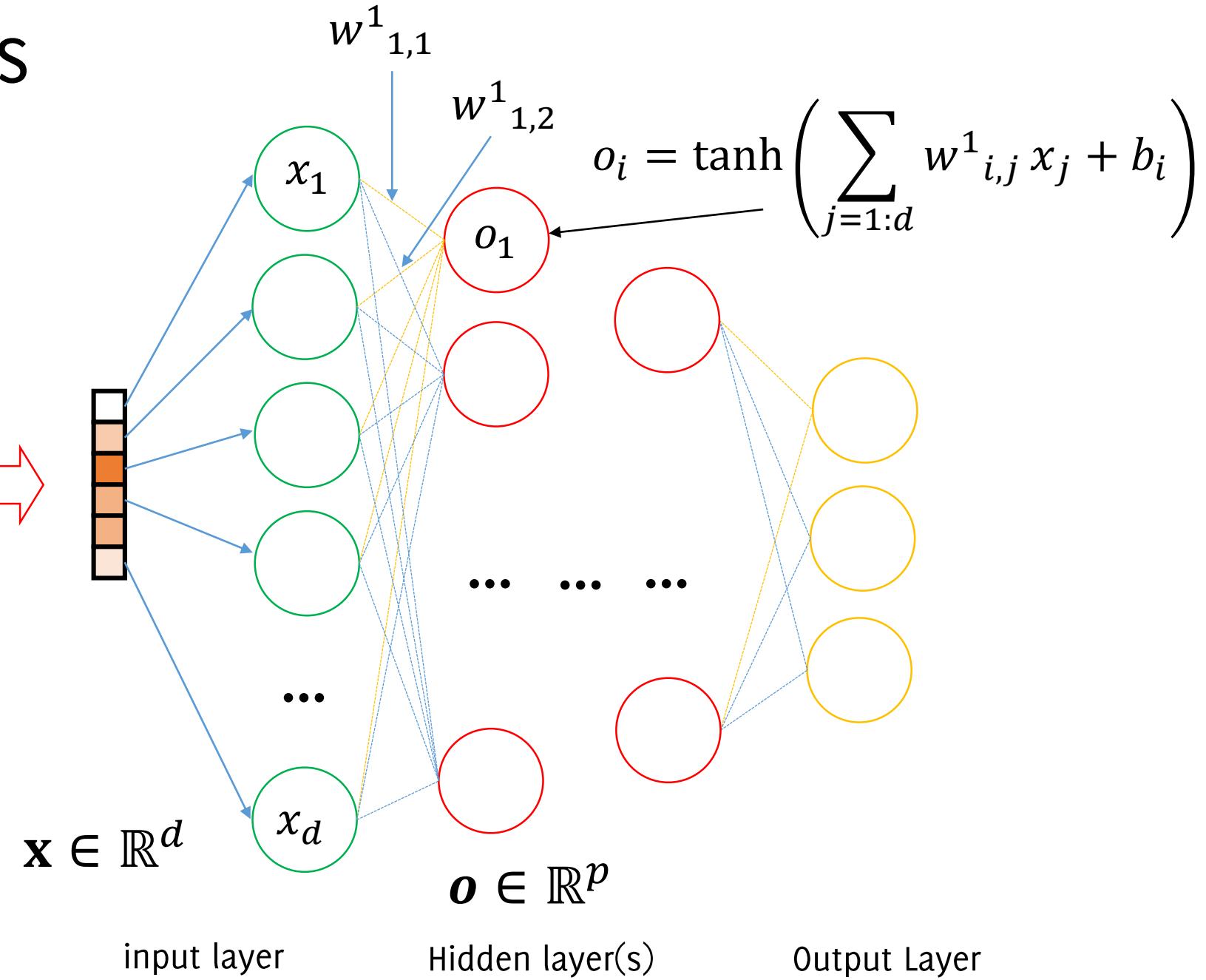
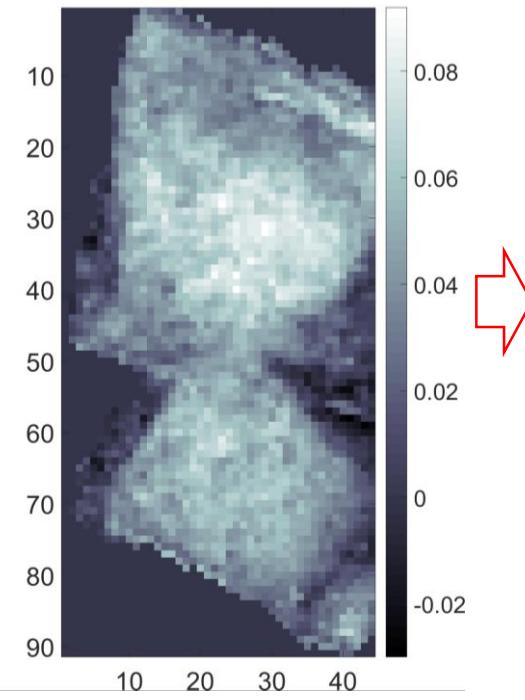
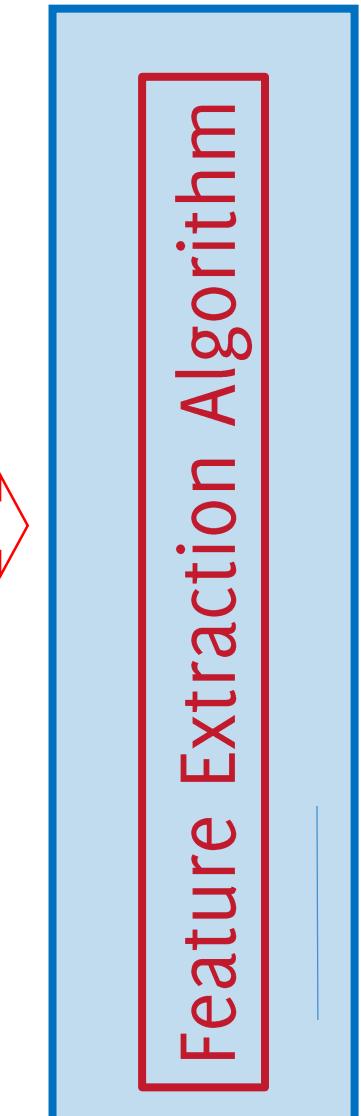


Image Classification by Hand Crafted Features

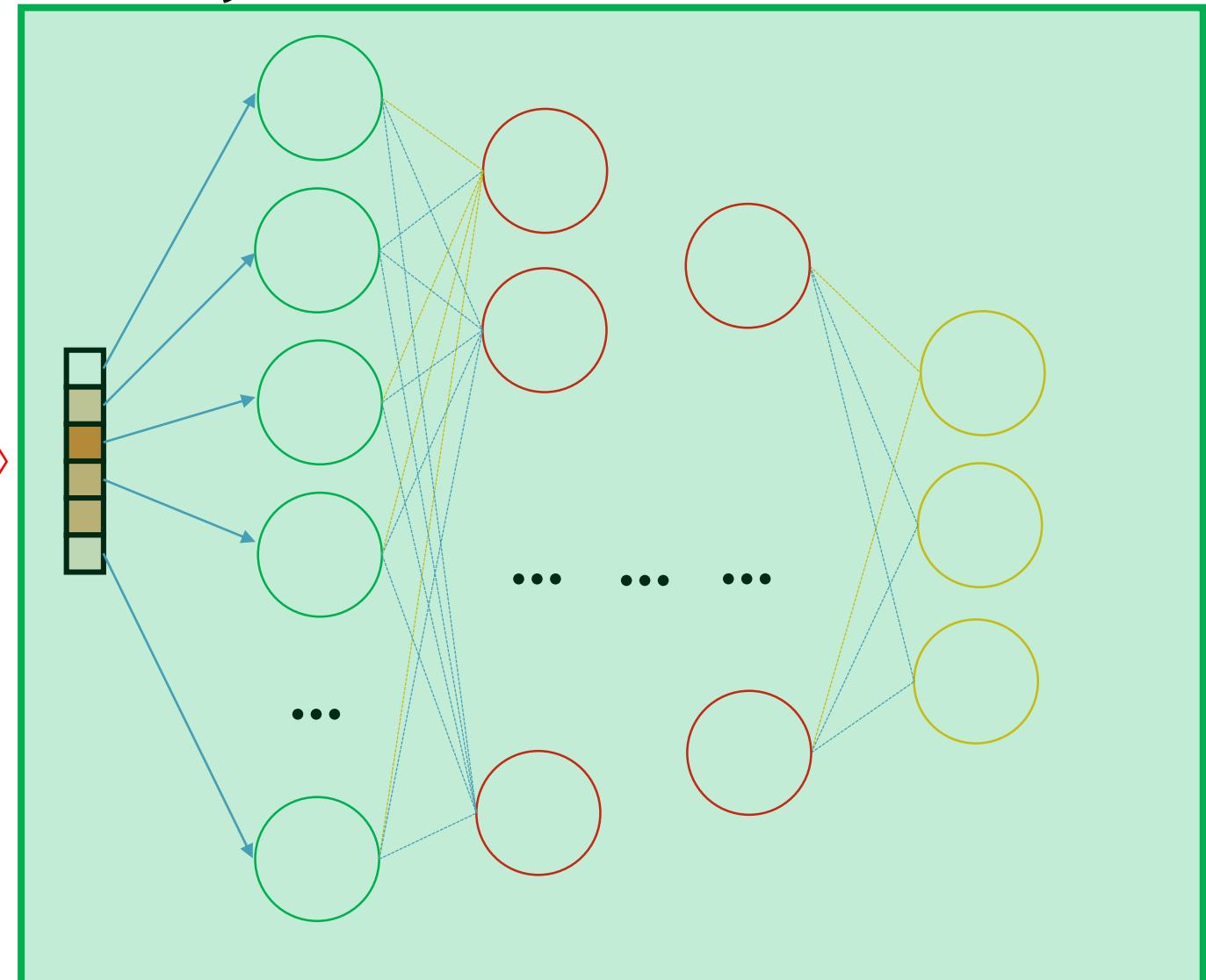
Input image



Feature Extraction Algorithm



Hand Crafted



Data Driven

Hand Crafted Features, pros:

- Exploit *a priori* / expert information
- Features are **interpretable** (you might understand why they are not working)
- You can **adjust features** to improve your performance
- **Limited amount of training data** needed
- You can give more relevance to some features

Hand Crafted Features, cons:

- Requires a lot of **design/programming** efforts
- **Not viable** in many **visual recognition** tasks (e.g. on natural images) which are easily performed by humans
- **Risk of overfitting** the training set used in the design
- **Not very general** and "portable"

Data-Driven Features

... the advent of deep learning

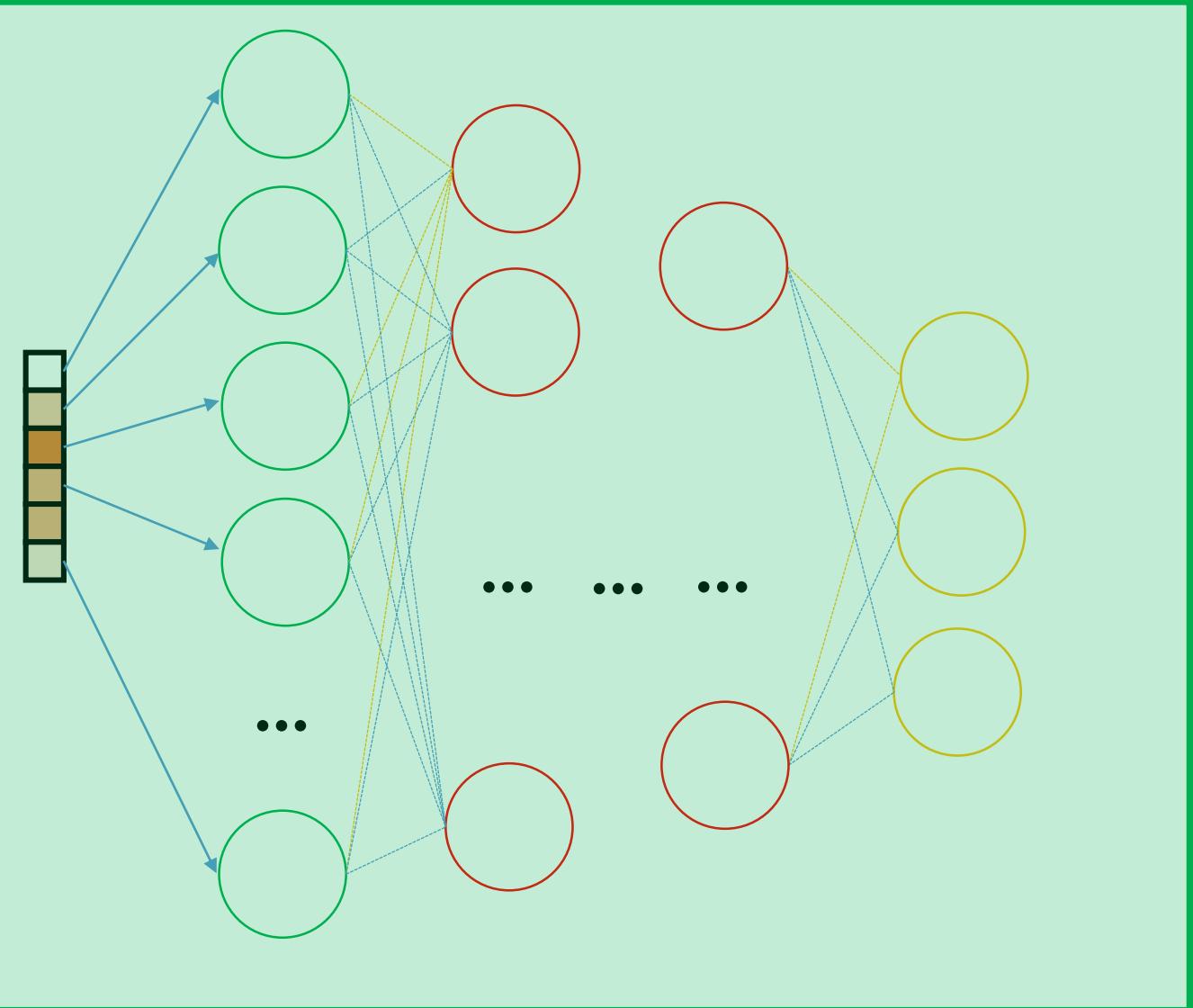
Data-Driven Features

Input image



$$I_1 \in \mathbb{R}^{r_1 \times c_1}$$

Feature Extraction



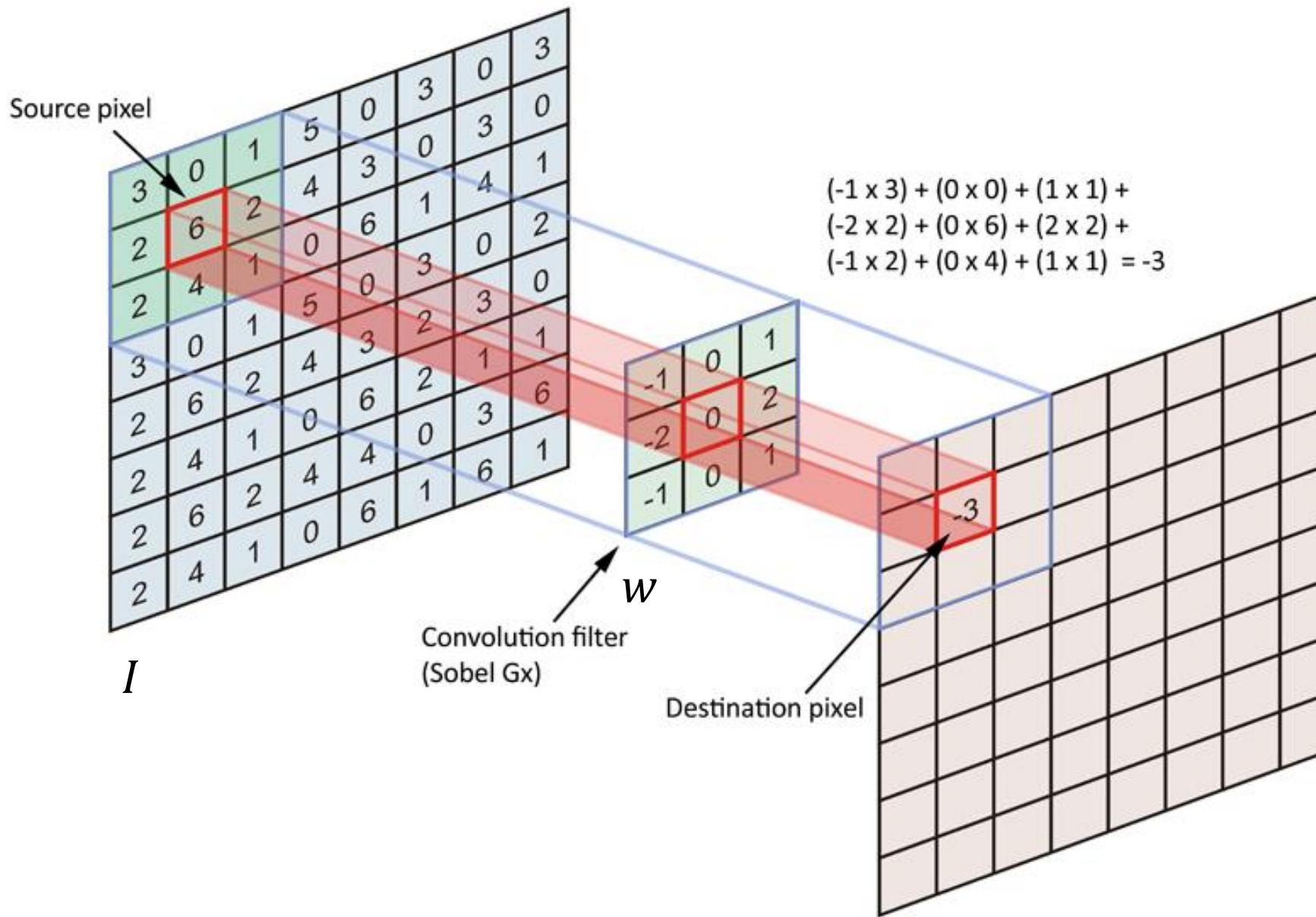
Data Driven

Data Driven

Convolutional Neural Networks

Setting up the stage

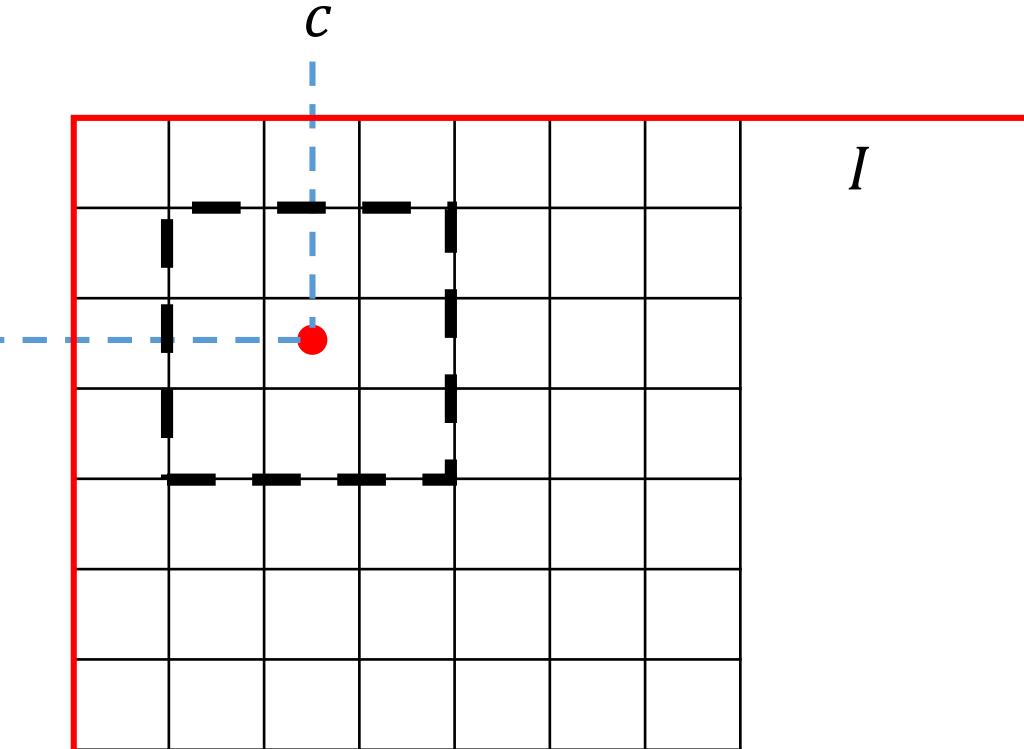
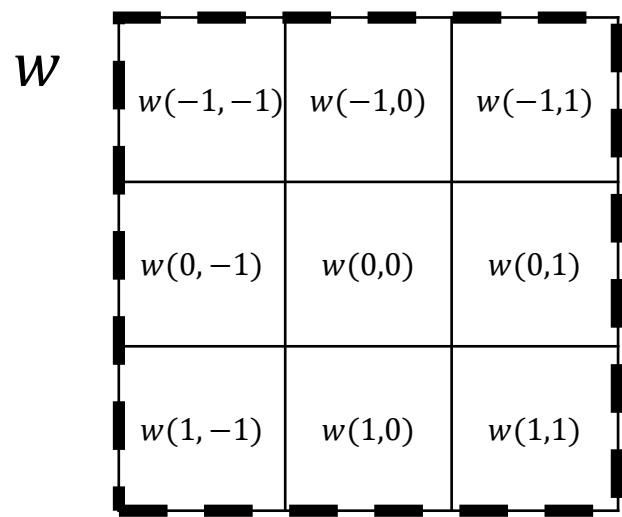
What is a convolution?



2D Correlation

Correlation is a linear transformation. Linearity implies that

$$(I \otimes w)(r, c) = \sum_{(u,v) \in U} w(u, v) * I(r + u, c + v)$$



We can consider weights as a filter h

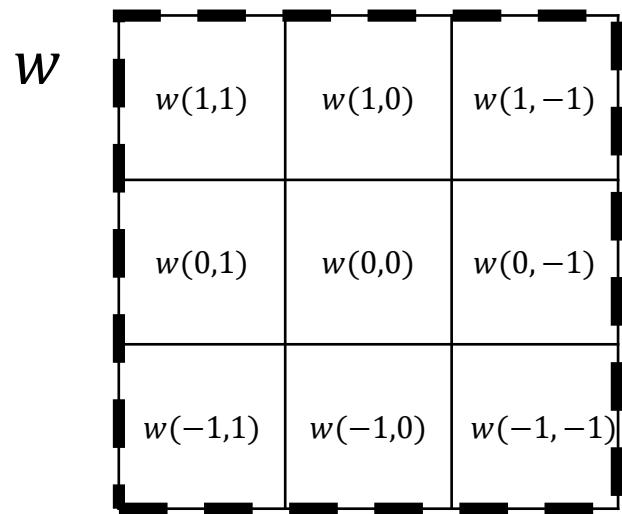
The filter h entirely defines convolution

Convolution operates the same in each pixel

2D Convolution

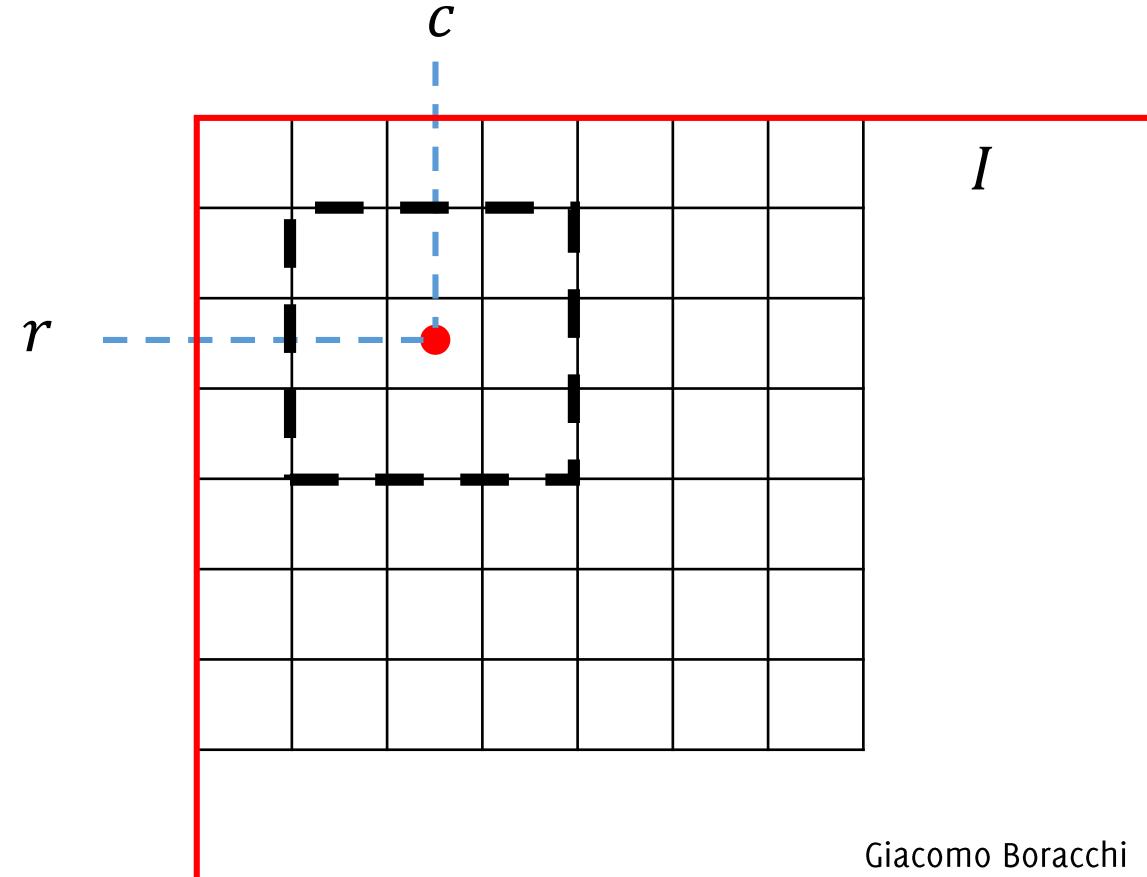
Convolution is a linear transformation. Linearity implies that

$$(I \circledast w)(r, c) = \sum_{(u,v) \in U} w(u, v) * I(r - u, c - v)$$



Rmk: indexes
have been shifted
in the filter w

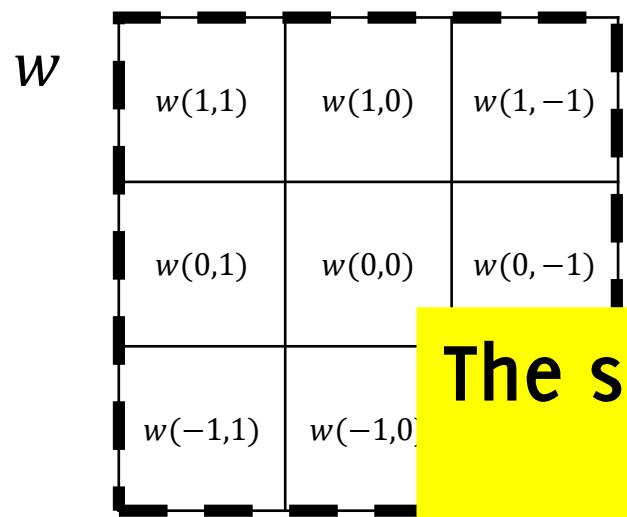
We can consider weights as a filter h
The filter h entirely defines convolution
Convolution operates the same in each pixel



2D Convolution

Convolution is a linear transformation. Linearity implies that

$$(I \circledast w)(r, c) = \sum_{(u,v) \in U} w(u, v) * I(r - u, c - v)$$

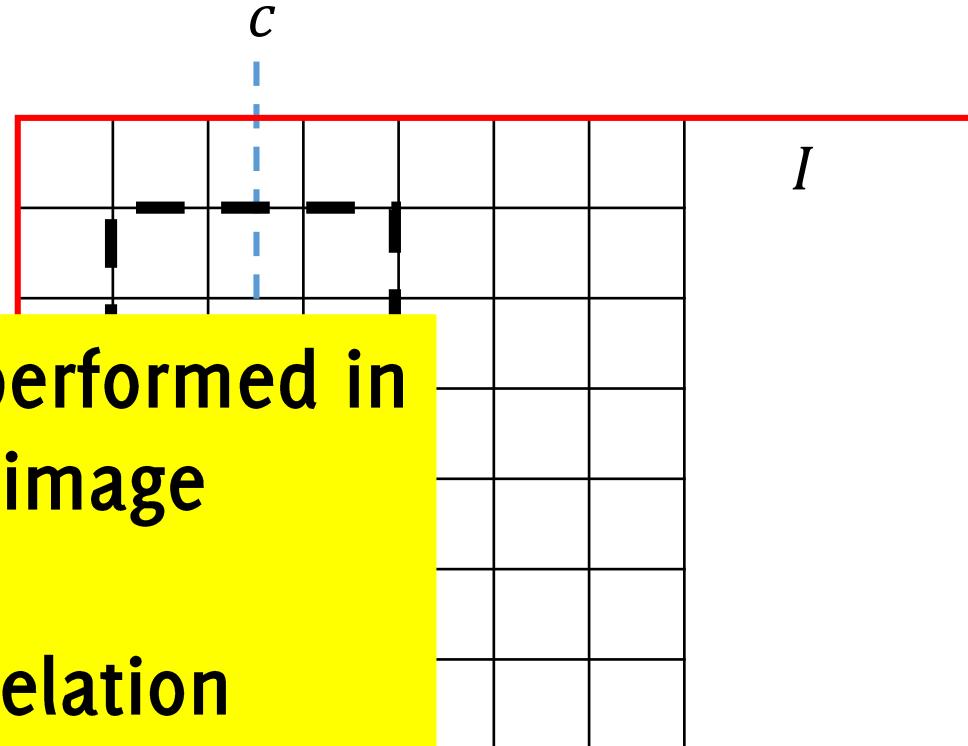


Rmk: indexes
have been shifted
in the filter w

We can con
The filter h e
Convolution op

The same operation is being performed in each pixel of the input image

It is equivalent to 2D Correlation
up to a «flip» in the filter w



2D Convolution

Convolution is a linear transformation. Linearity implies that

$$(I \circledast w)(r, c) = \sum_{(u,v) \in U} w(u, v) * I(r - u, c - v)$$

Convolution is defined up to the “filter flip” for the Fourier Theorem to apply. Filter flip has to be taken into account when computing convolution in Fourier domain and when designing filters.

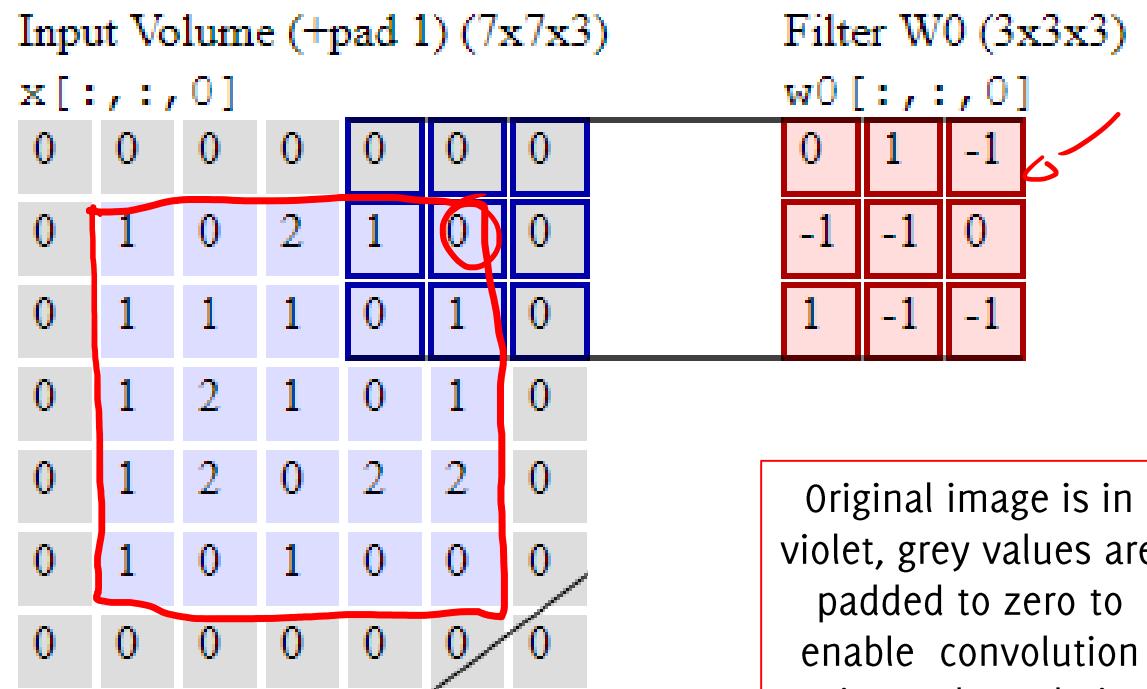
However, in CNN, convolutional filters are being learned from data, thus it is only important to use these in a consistent way.

In practice, in CNN arithmetic, there is no flip during training/inference.

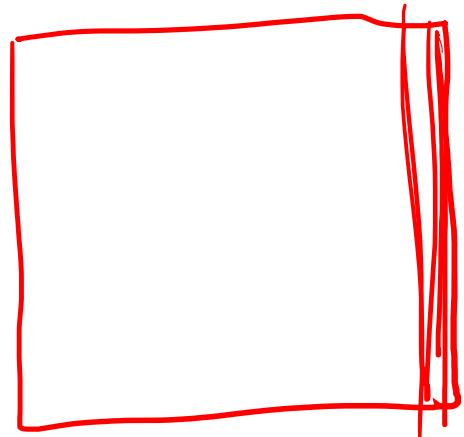
Convolution: Padding

How to define convolution output close to image boundaries?

Padding with zero is the most frequent option, as this does not change the output size. However, no padding or symmetric padding are also viable options



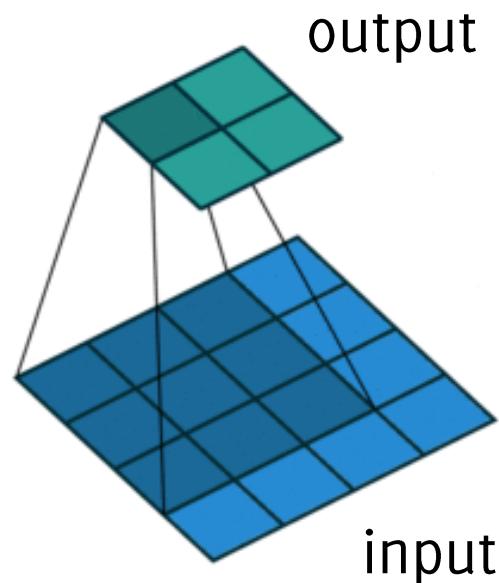
conv2(I, f, 'same')



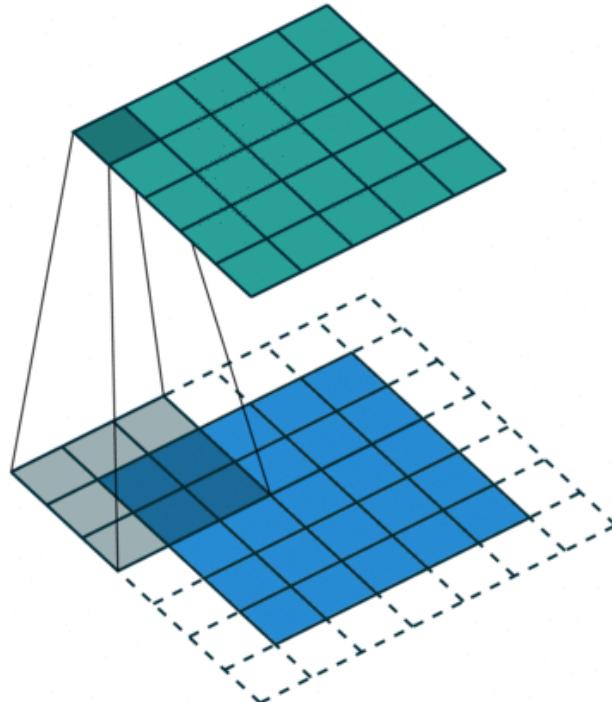
Padding Options in Convolution Animation

Rmk: Blue maps are inputs, and cyan maps the outputs.

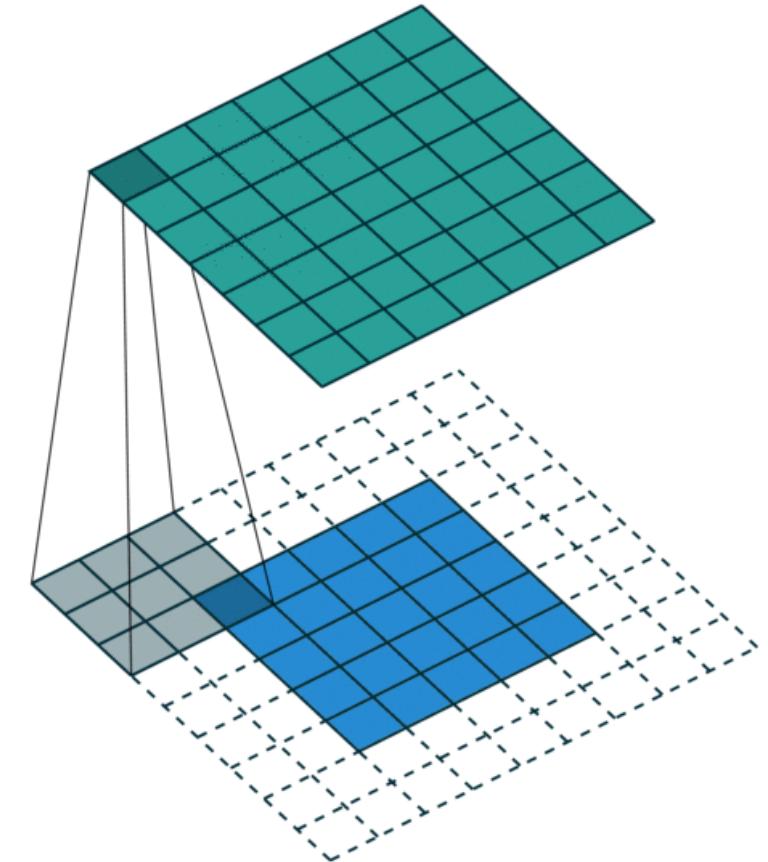
Rmk: the filter here is 3×3



No padding
«valid»



Half padding
«same»

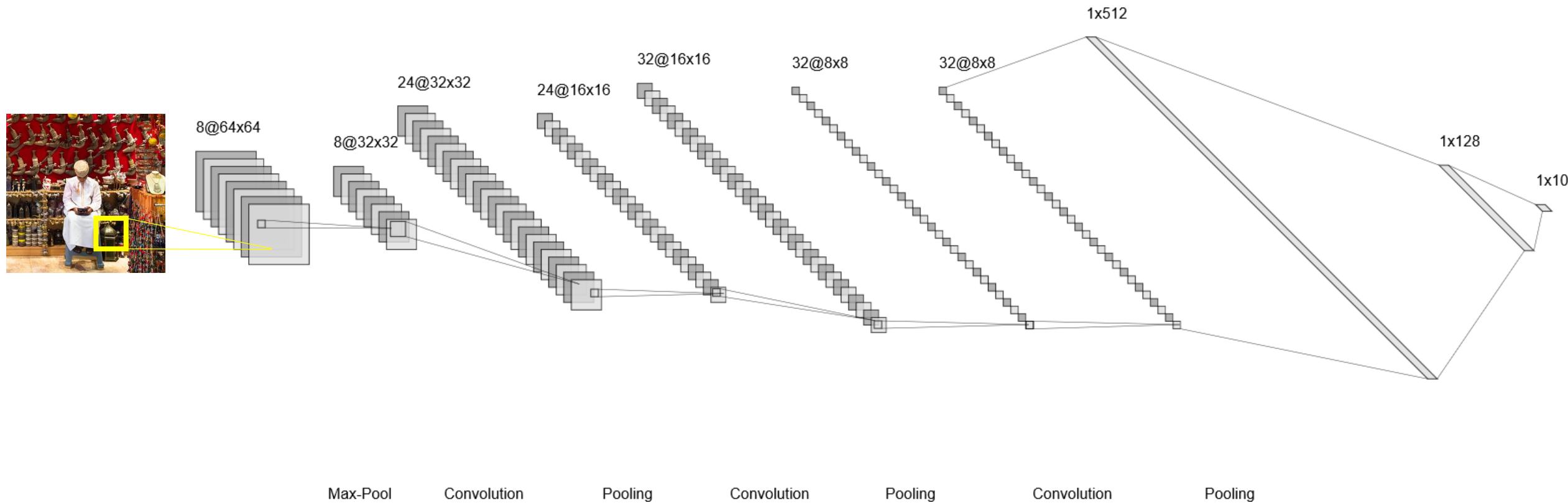


full padding
«full»

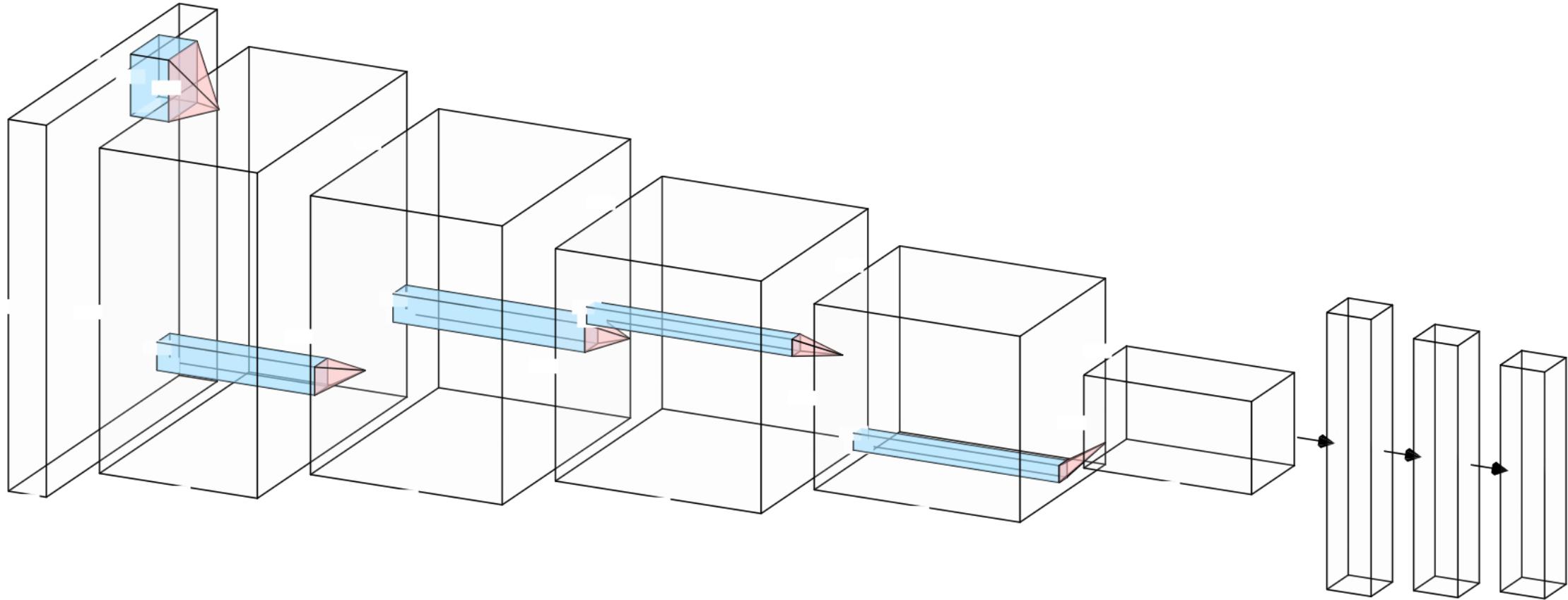
Convolutional Neural Networks

CNNs

The typical architecture of a CNN



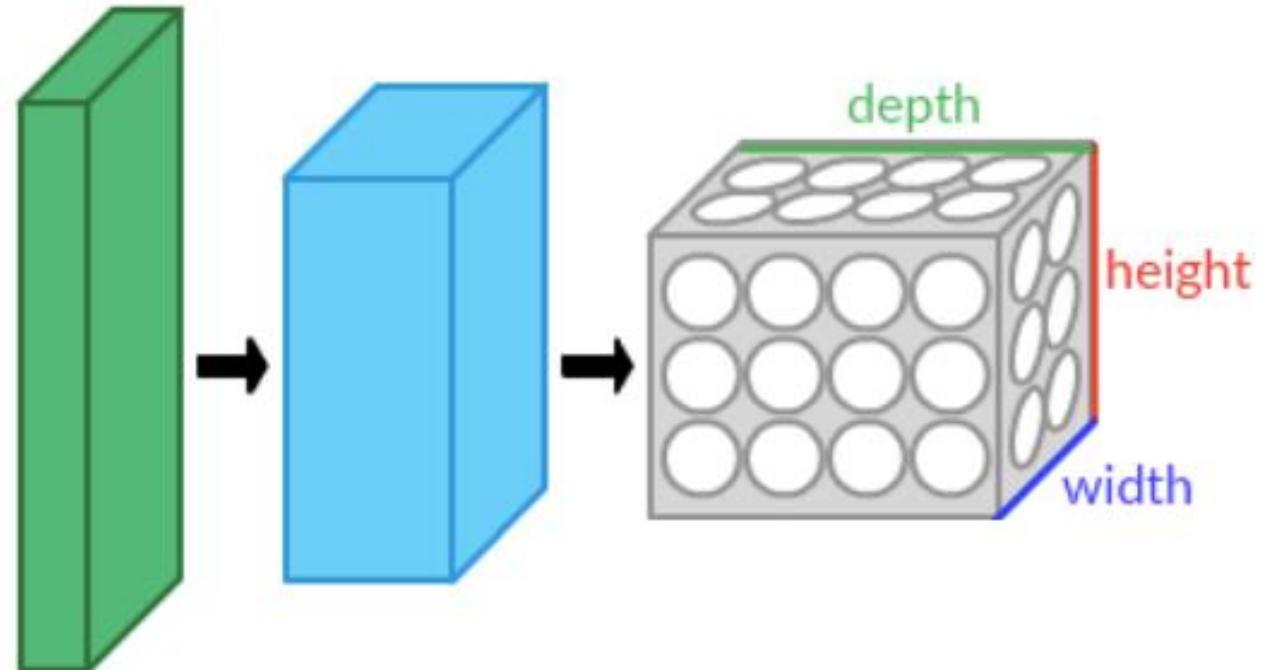
The typical architecture of a CNN



Convolutional Neural Networks (CNN)

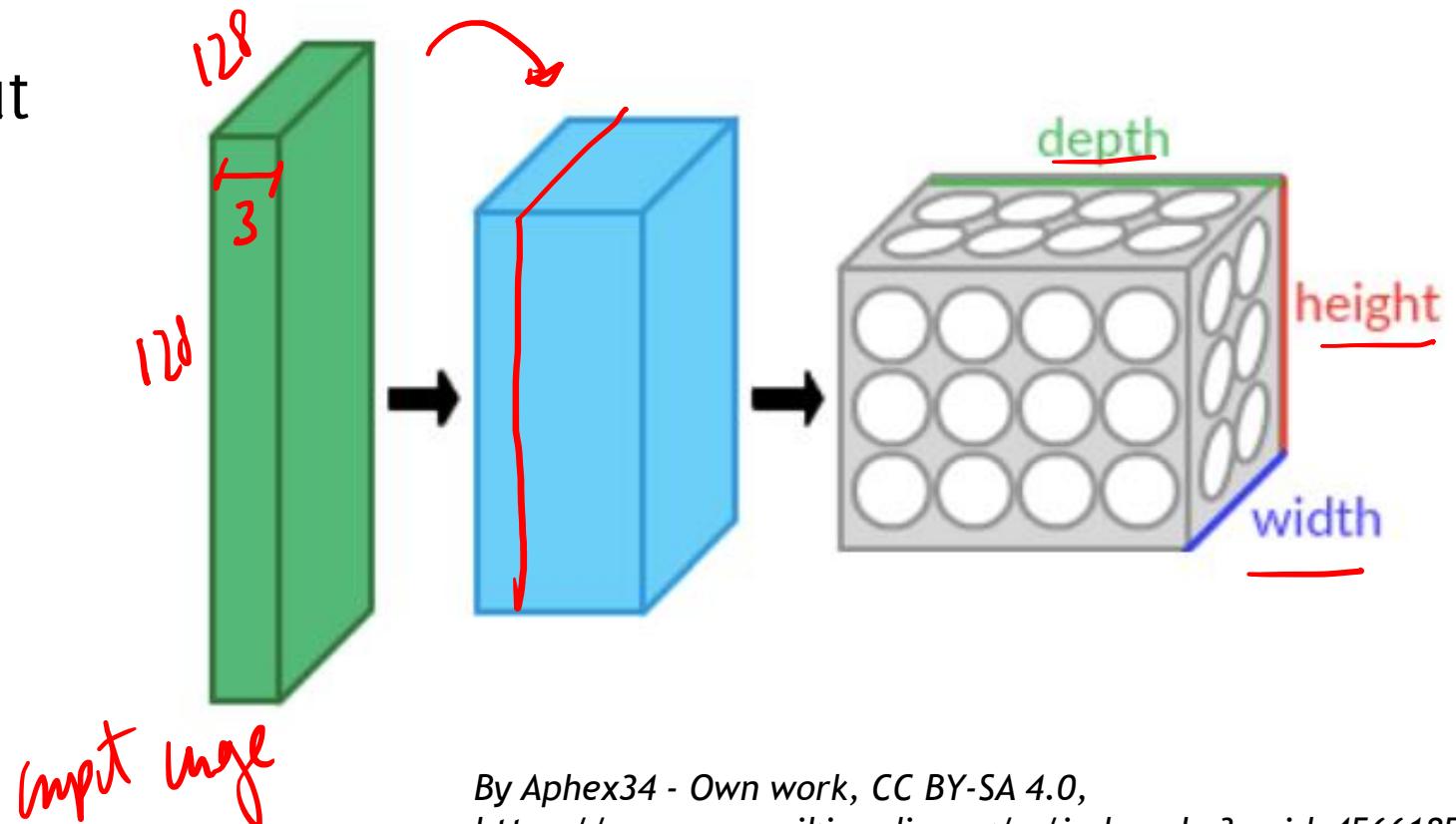
CNN are typically made of blocks that include:

- Convolutional layers
- Nonlinearities (activation functions)
- Maxpooling



Convolutional Neural Networks (CNN)

- An image passing through a CNN is transformed in a sequence of volumes.
- As the depth increases, the height and width of the volume decreases
- Each layer takes as input and returns a volume



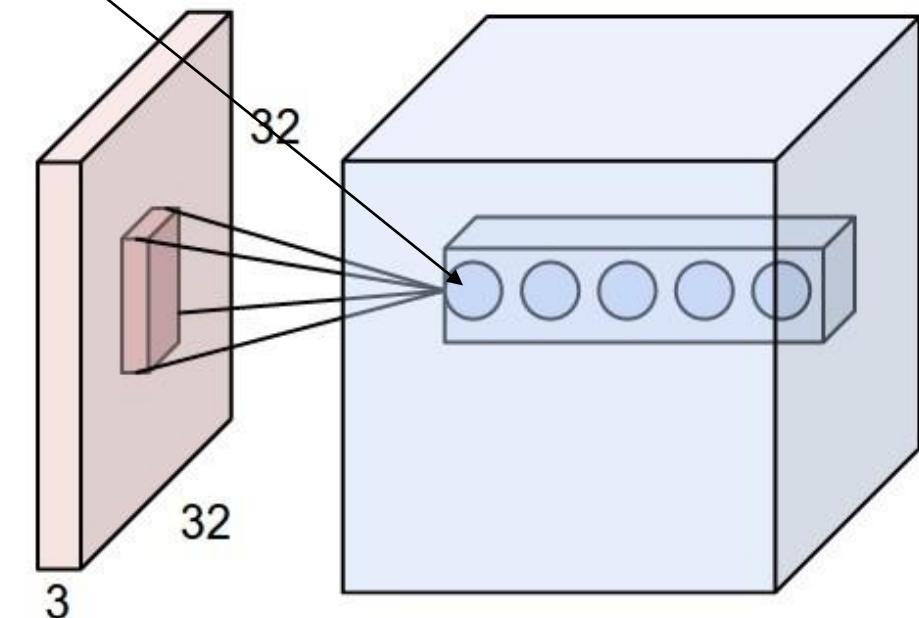
Convolutional Layers

Convolutional Layers

Convolutional layers "mix" all the input components

The output is a linear combination of all the values in a region of the input

$$y = \sum_i w_{i,j}^1 x_i + b^1$$



By Aphex34 - Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=45659236>

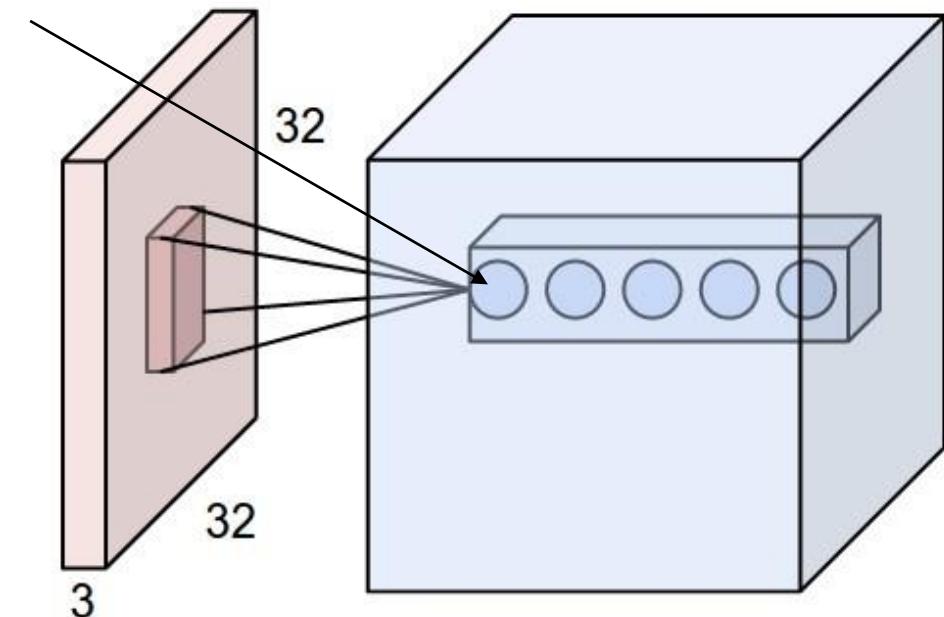
Convolutional Layers

Convolutional layers "mix" all the input components

The output is a linear combination of all the values in a region of the input

$$y = \sum_i w_{i,j}^1 x_i + b^1$$

The parameters of this layer are called filters.



By Aphex34 - Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=45659236>

Convolutional Layers

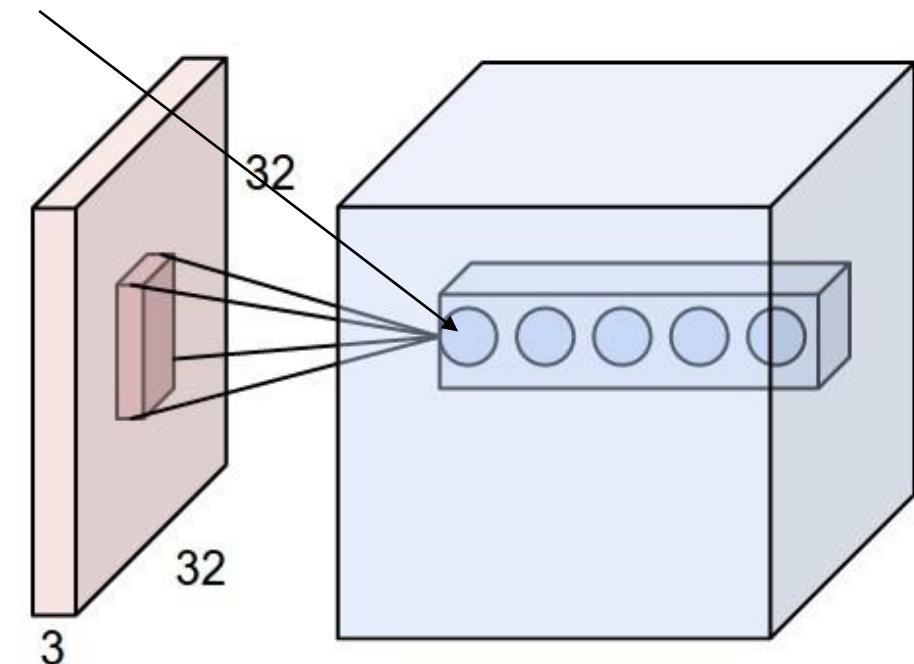
Convolutional layers "mix" all the input components

The output is a linear combination of all the values in a region of the input

$$y = \sum_i w_{i,j}^1 x_i + b^1$$

The parameters of this layer are called filters.

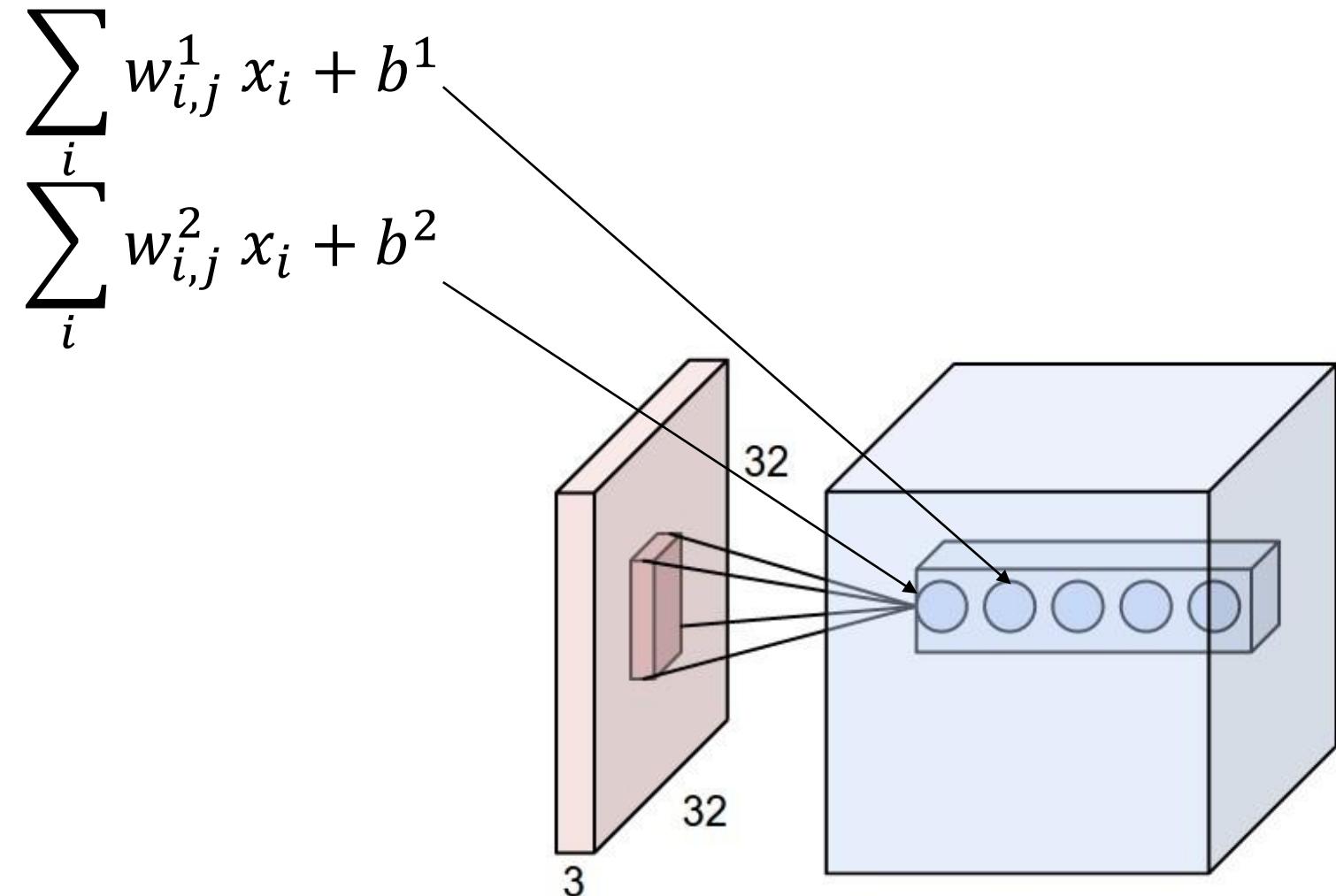
The same filter is used through the whole spatial extent of the input



By Aphex34 - Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=45659236>

Convolutional Layers

Different filters yield different layers in the output

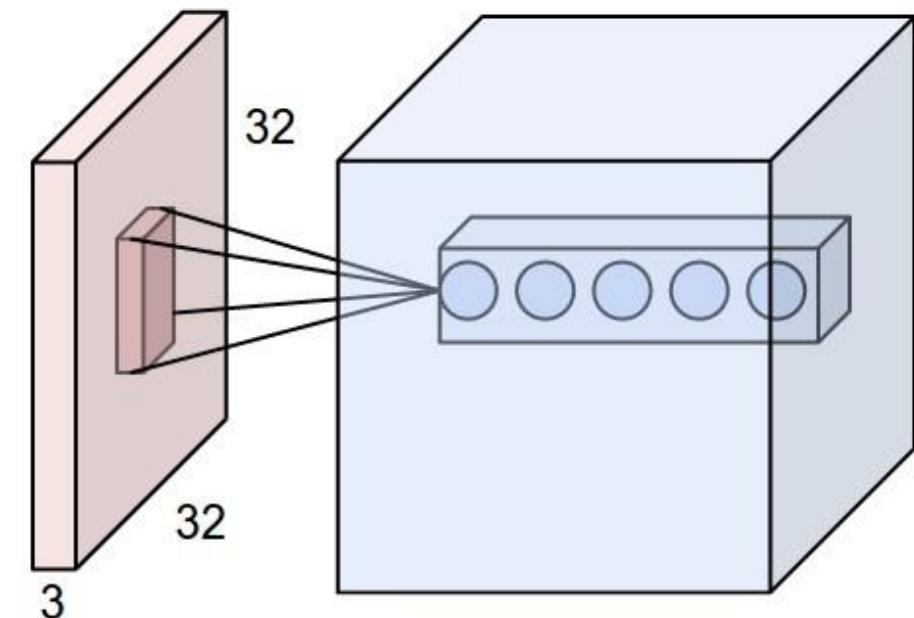


By Aphex34 - Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=45659236>

Recap: Convolutional Layers

Convolutional layers "mix" all the input components

- The **output** is also called **volume** or **activation maps**
- Each filter yields a different slice of the output volume
- Each filter has depth equal to the depth of the input volume

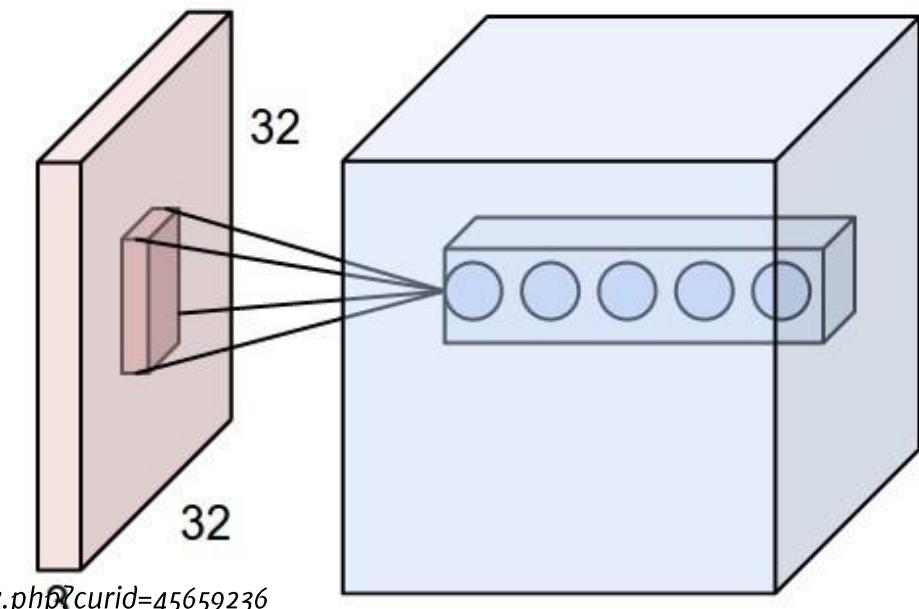


By Aphex34 - Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=45659236>

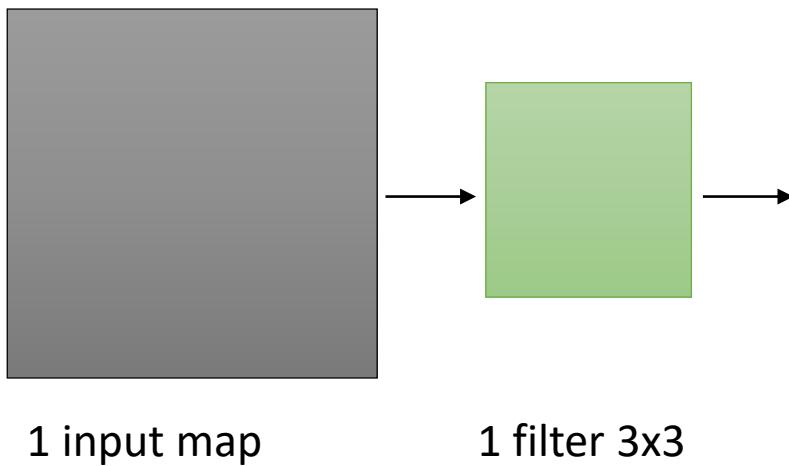
Convolutional Layers, remarks:

- Convolutional Layers are described by a set of filters
- Filters represent the weights of these linear combination.
- Filters have very small spatial extent and large depth extent,
- The filter depth is typically not specified, as it corresponds to the number of layers of the input volume

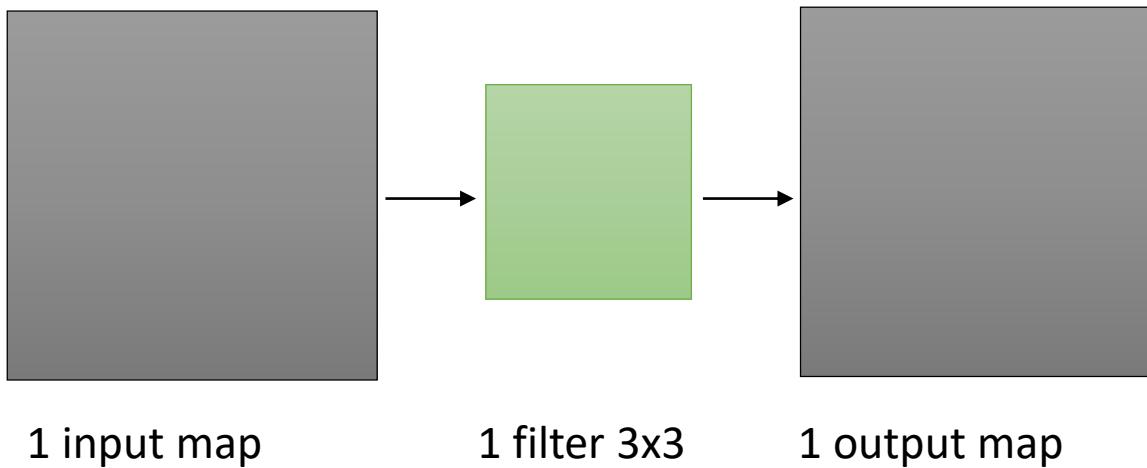
The output of the convolution against a filter becomes a slice in the volume feed to the next layer



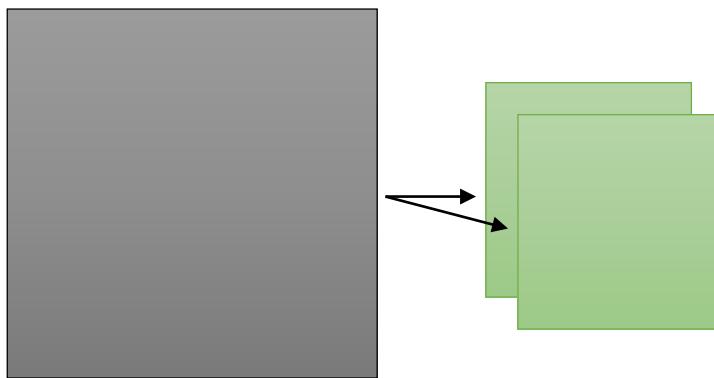
CNN Arithmetic



CNN Arithmetic



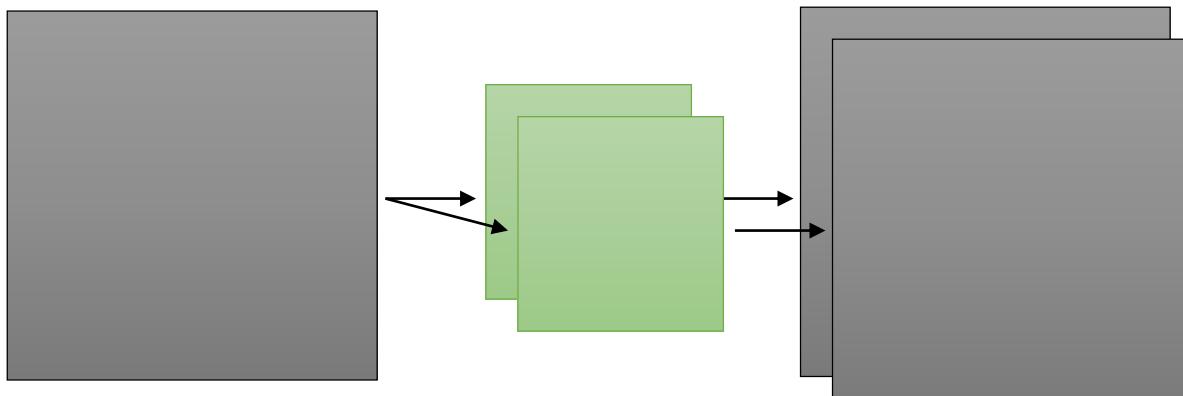
CNN Arithmetic



1 input map

2 filters 3x3

CNN Arithmetic

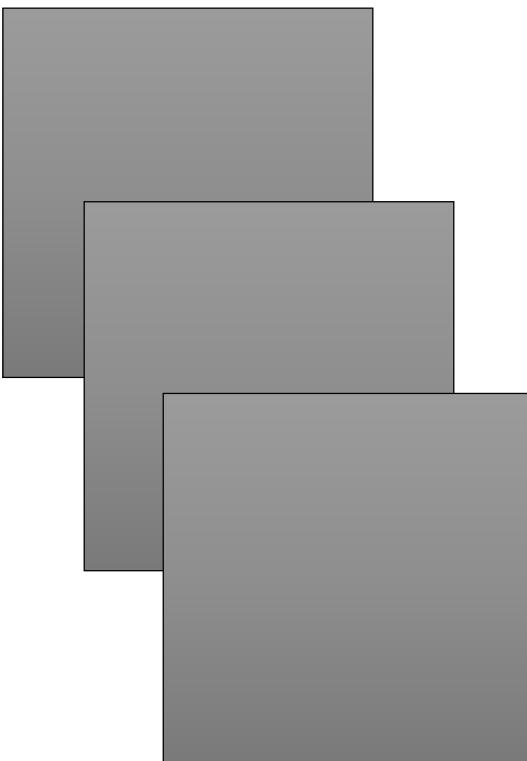


1 input map

2 filters 3x3

2 output maps

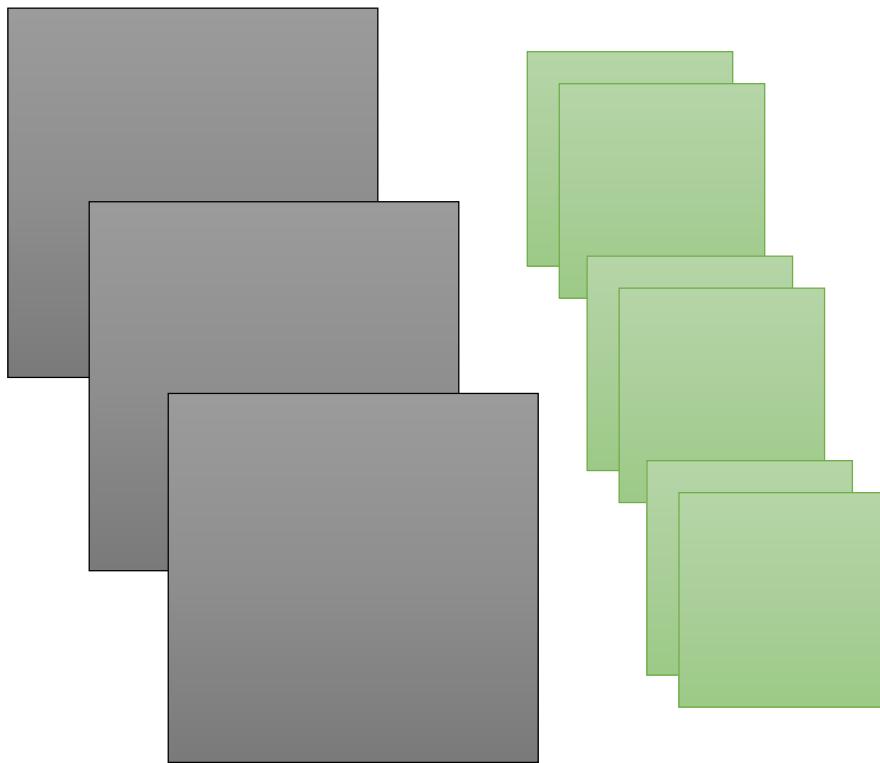
CNN Arithmetic



3 input maps

2 filters 3x3

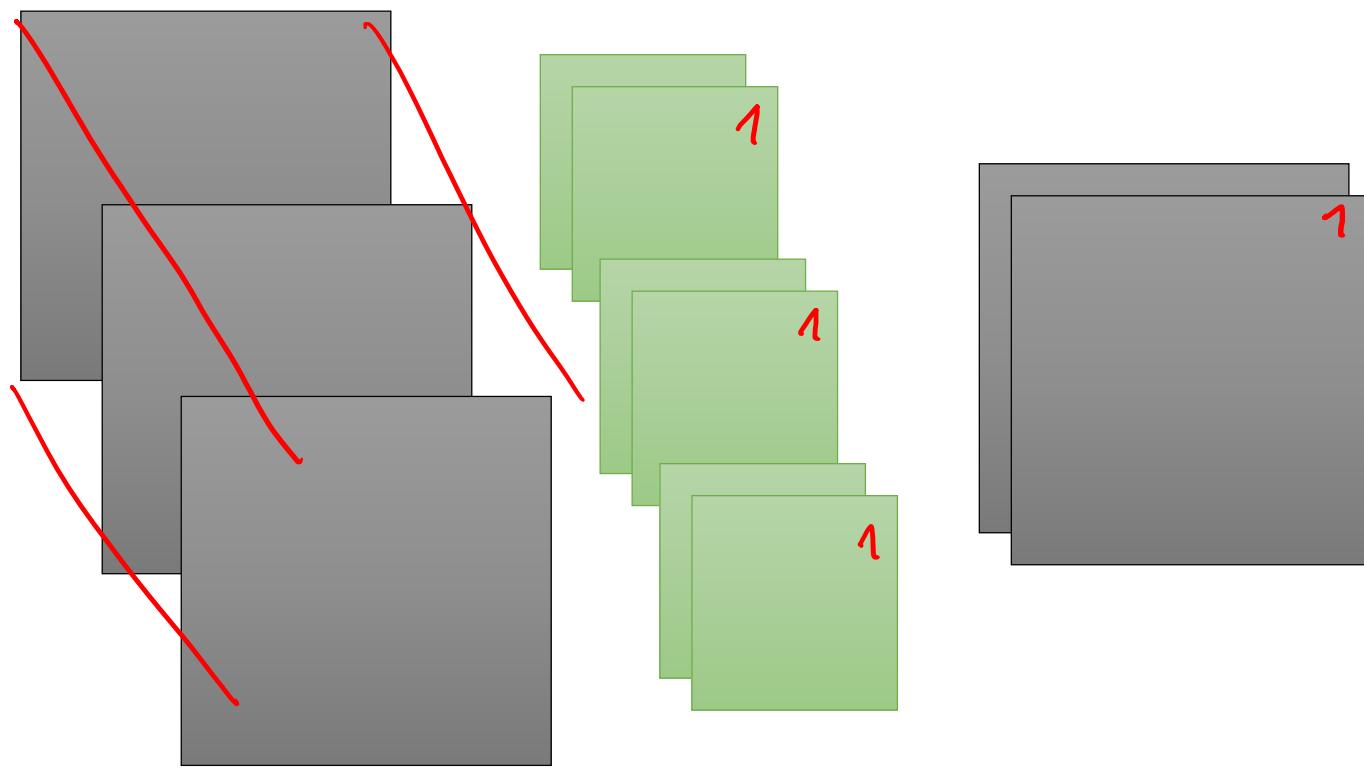
CNN Arithmetic



3 input maps

2 filters 3x3

CNN Arithmetic

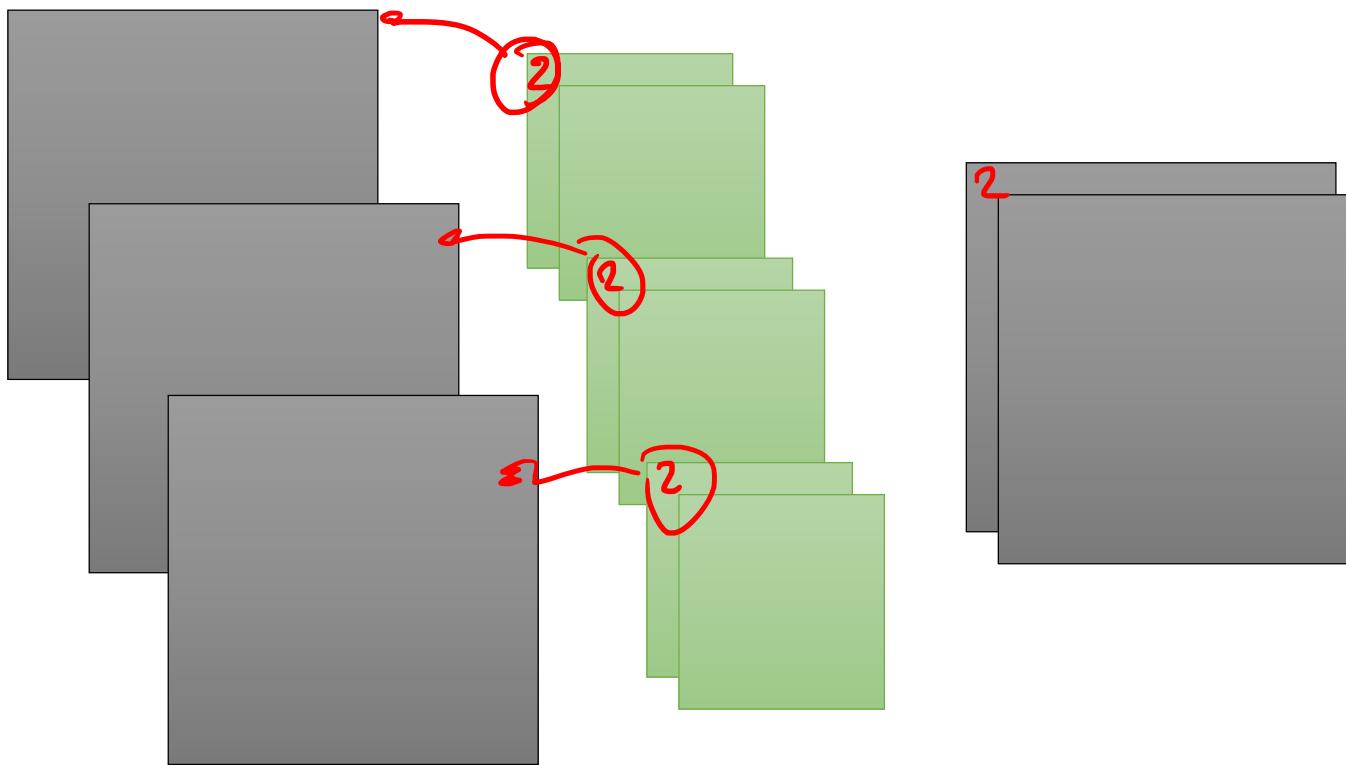


3 input maps

2 filters 3x3

2 output maps

CNN Arithmetic

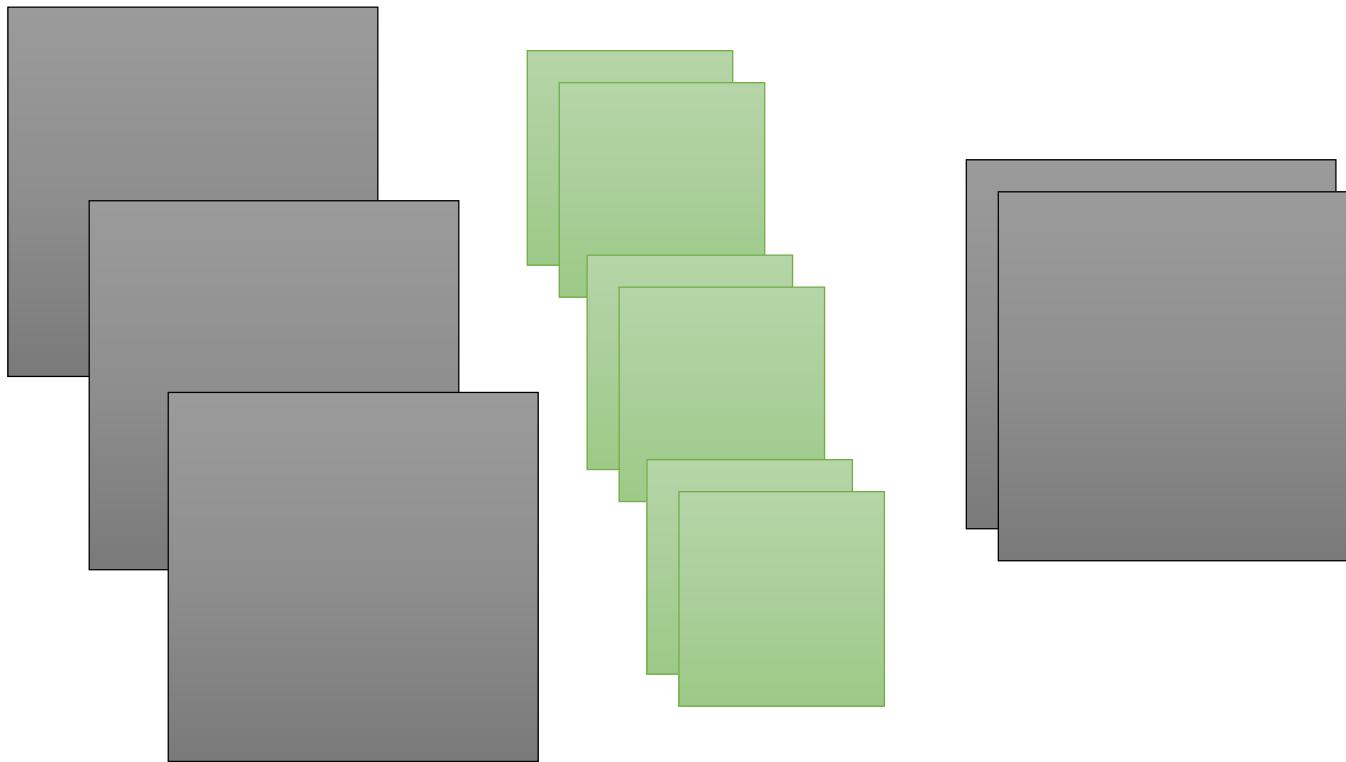


3 input maps

2 filters 3x3

2 output maps

CNN Arithmetic



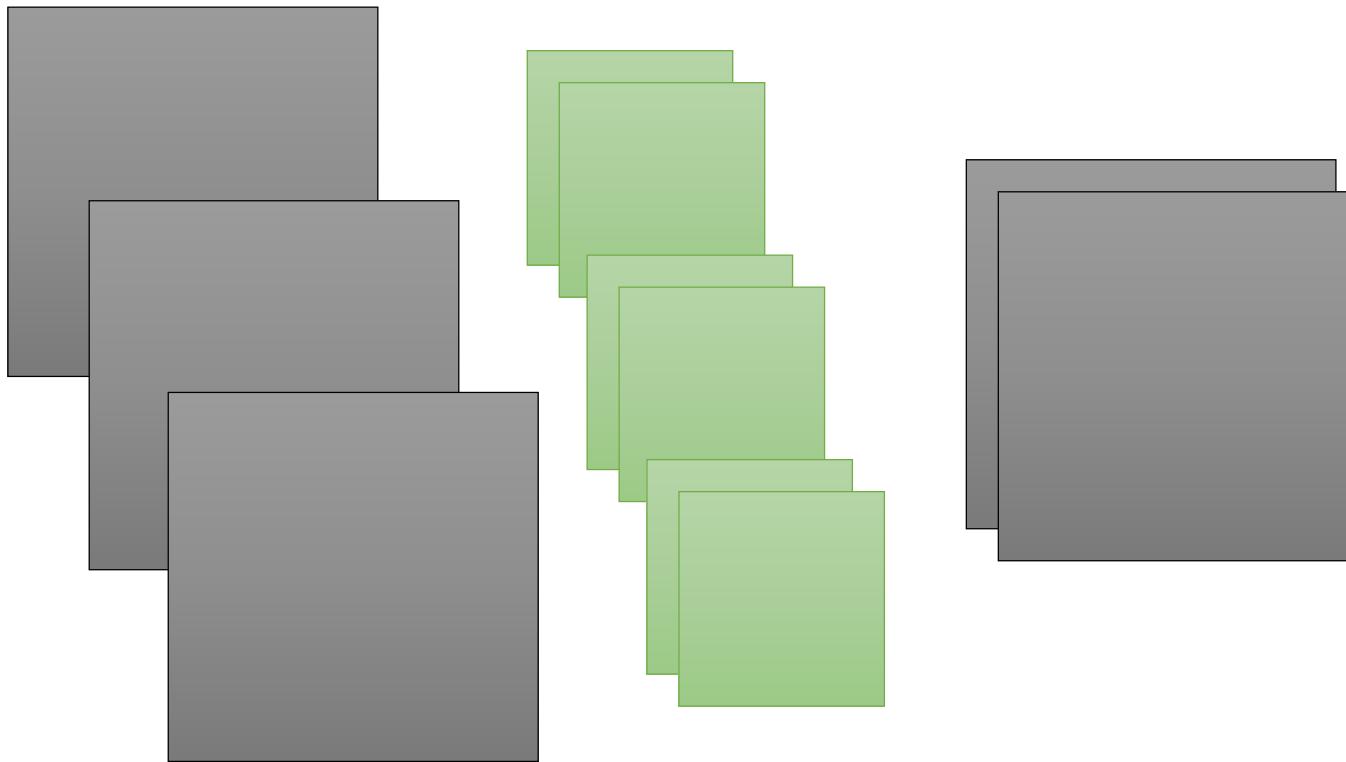
3 input maps

2 filters 3x3

2 output maps

Quiz: how many parameters
does this layer have?

CNN Arithmetic



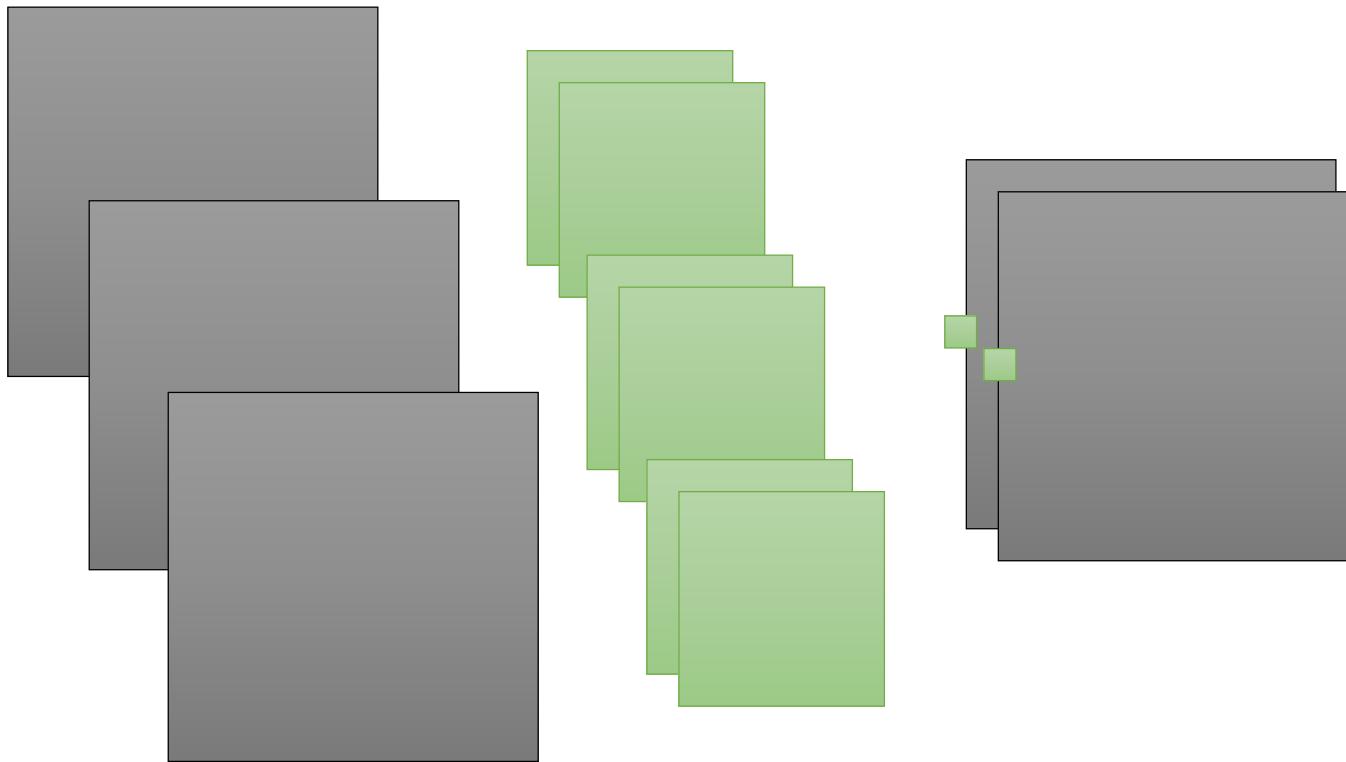
3 input maps

2 filters 3x3

= 54 parameters in the filters

2 output maps

CNN Arithmetic

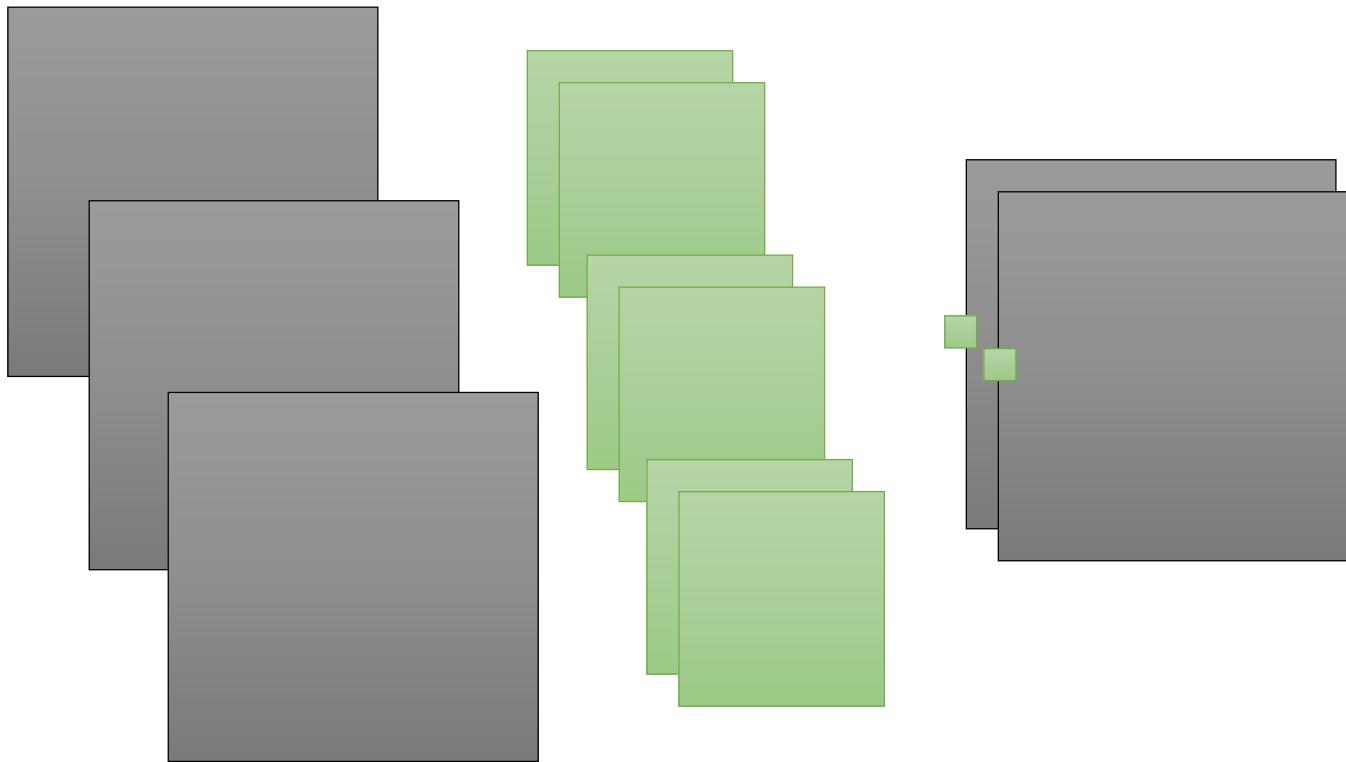


3 input maps

2 filters 3x3
= 54 ...

2 output maps
+ 2 biases

CNN Arithmetic



3 input maps

2 filters 3x3

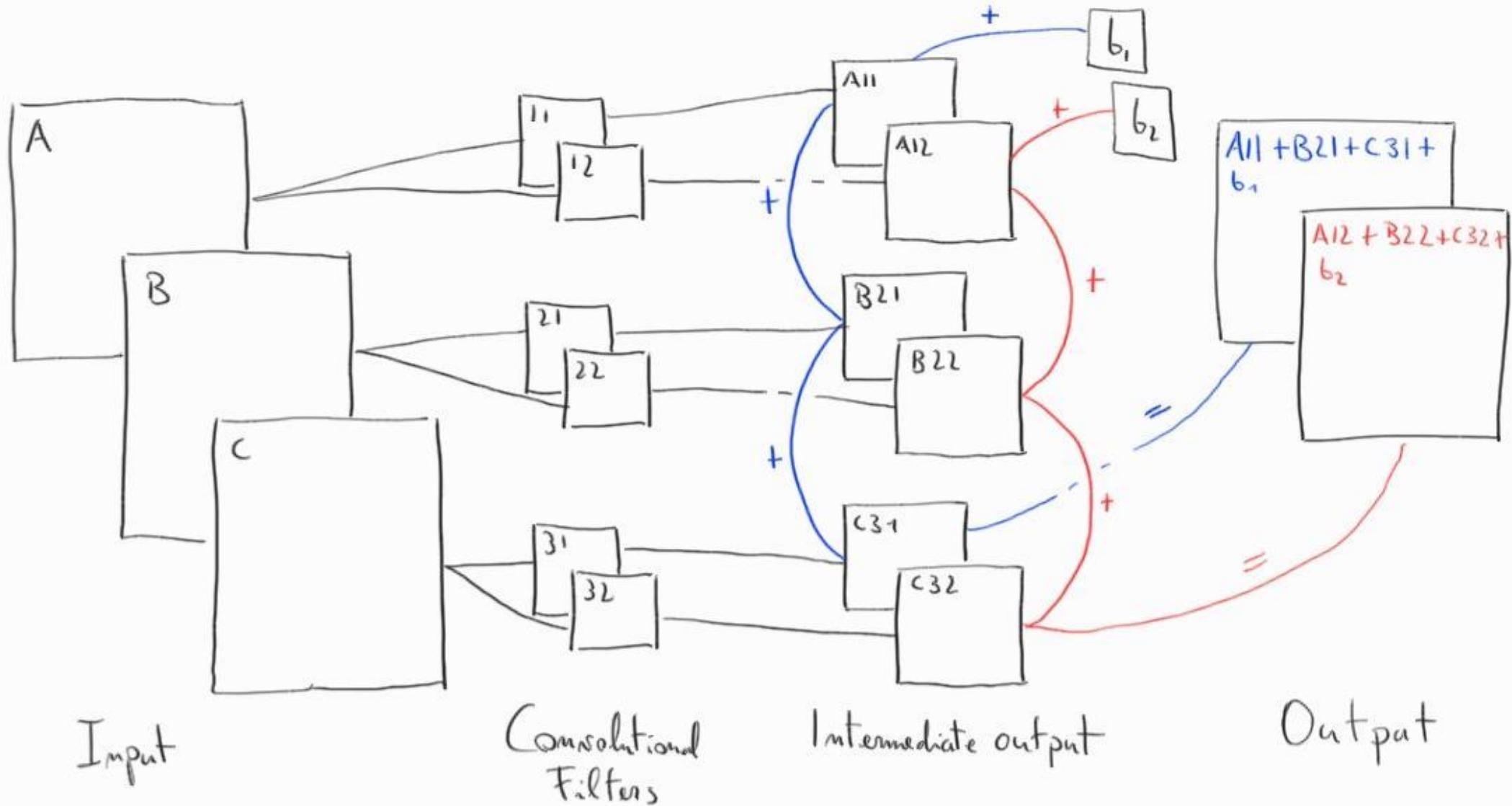
2 output maps

= 54 ...

+ 2 biases

= 56 trainable parameters (weights)

To Recap...



Other Layers

Activation and Pooling

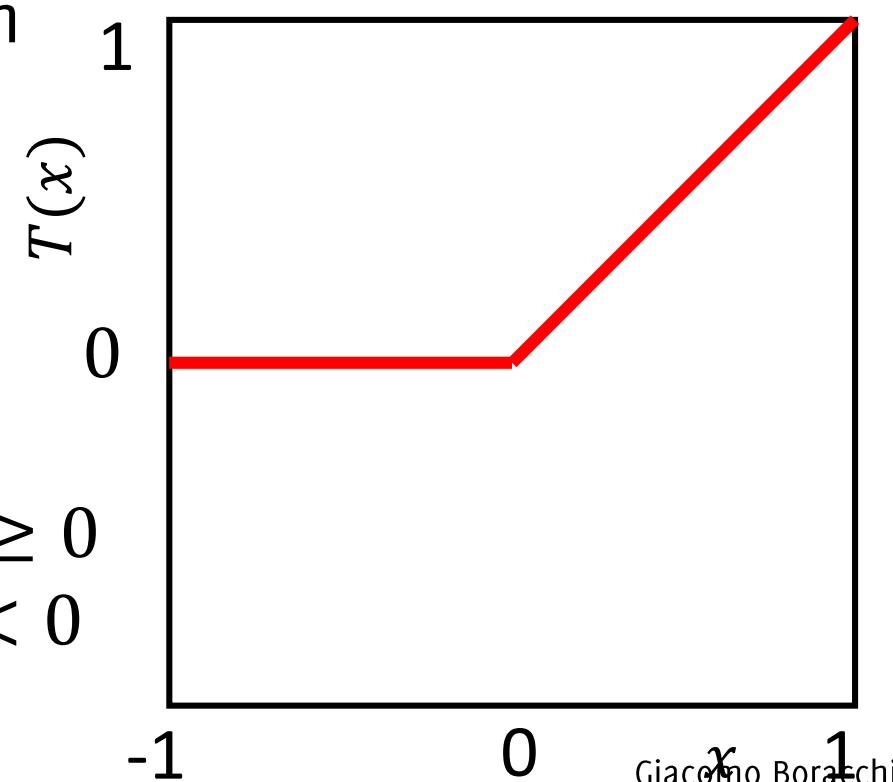
Activation Layers

Introduce nonlinearities in the network, otherwise the CNN might be equivalent to a linear classifier...

RELU (Rectifier Linear Units): it's a thresholding on the feature maps, i.e., a $\max(0, \cdot)$ operator.

- By far the most popular activation function in deep NN (since when it has been used in AlexNet)
- Dying neuron problem: a few neurons become insensitive to the input (vanishing gradient problem)

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

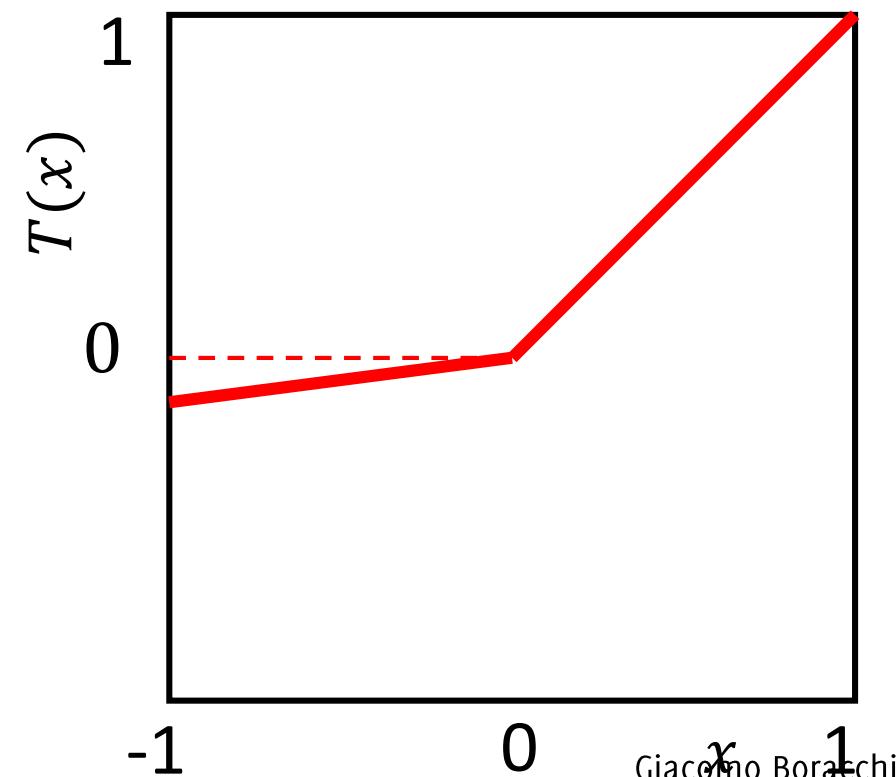


Activation Layers

Introduce nonlinearities in the network, otherwise the CNN might be equivalent to a linear classifier...

LEAKY RELU: like the relu but include a small slope for negative values

$$T(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0.01 * x & \text{if } x < 0 \end{cases}$$



Activation Layers

Introduce nonlinearities in the network, otherwise the CNN might be equivalent to a linear classifier...

TANH (hyperbolic Tangent): has a range (-1,1), continuous and differentiable

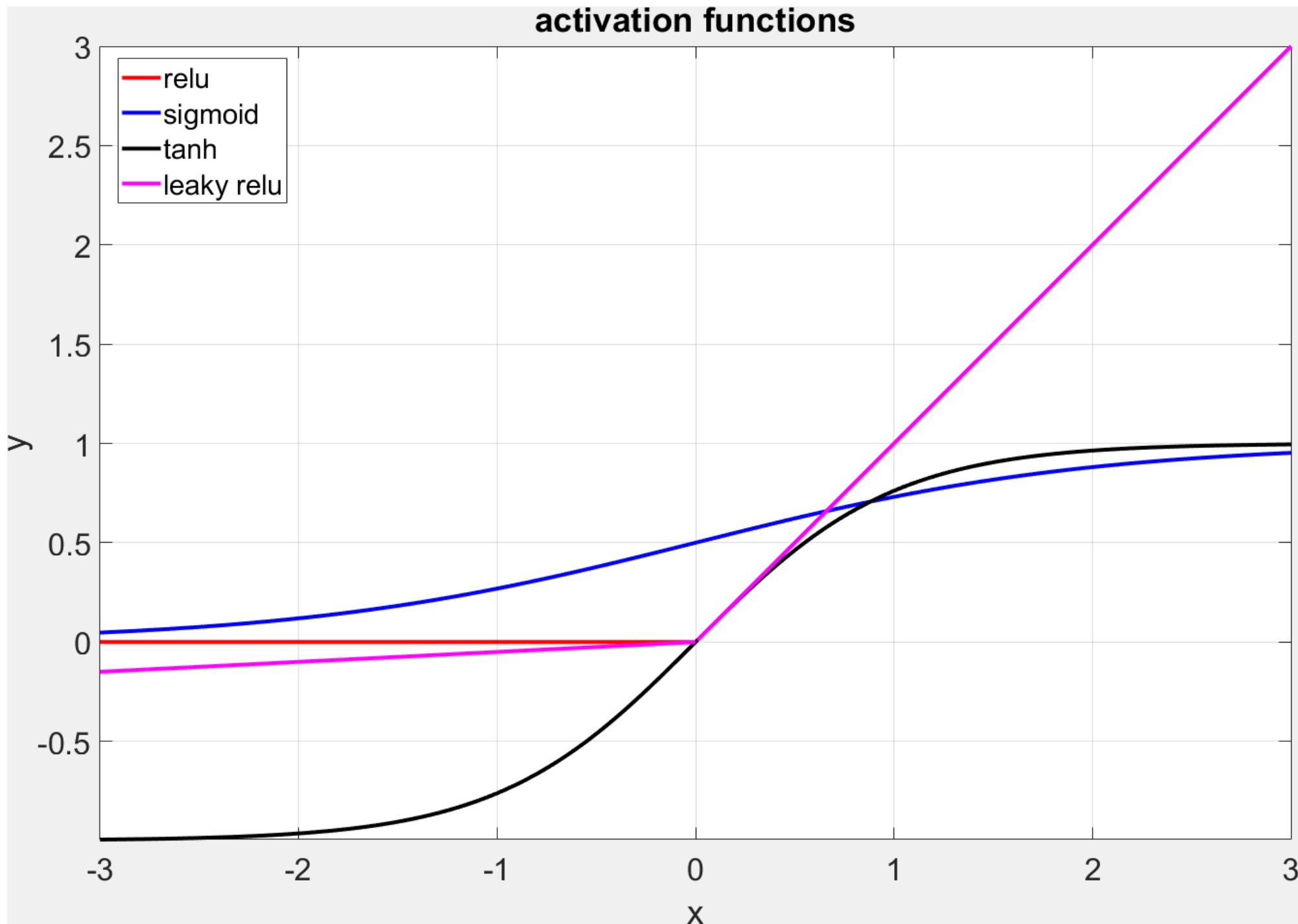
$$T(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} - 1$$

SIGMOID: has a range (0,1), continuous and differentiable

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

These activation functions are mostly popular in **MLP architectures**

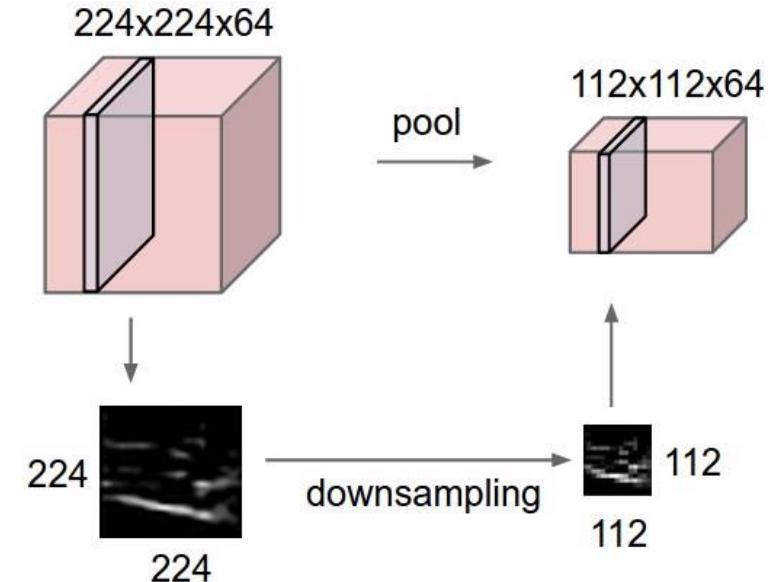
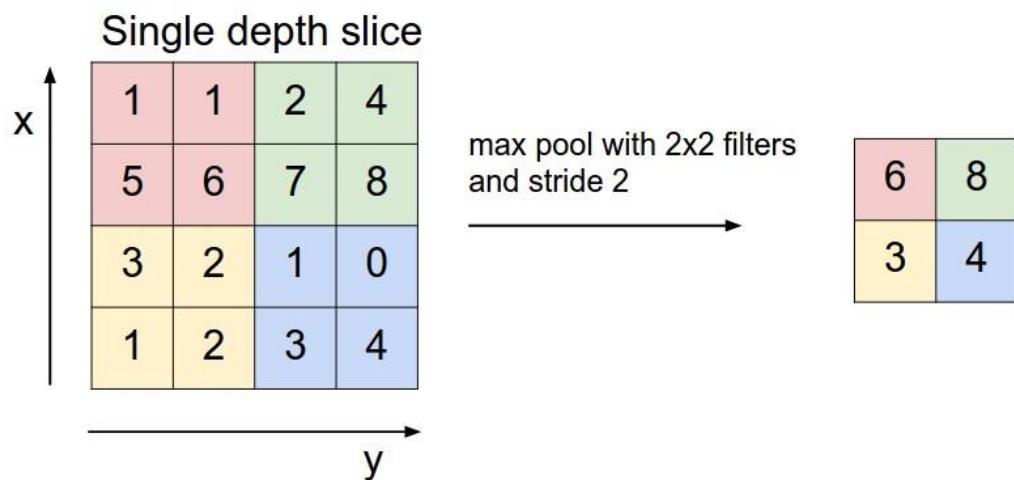
activation functions



Pooling Layers

Pooling Layers reduce the spatial size of the volume.

The Pooling Layer operates **independently** on every depth slice of the input and resizes it spatially, often using the **MAX operation**.



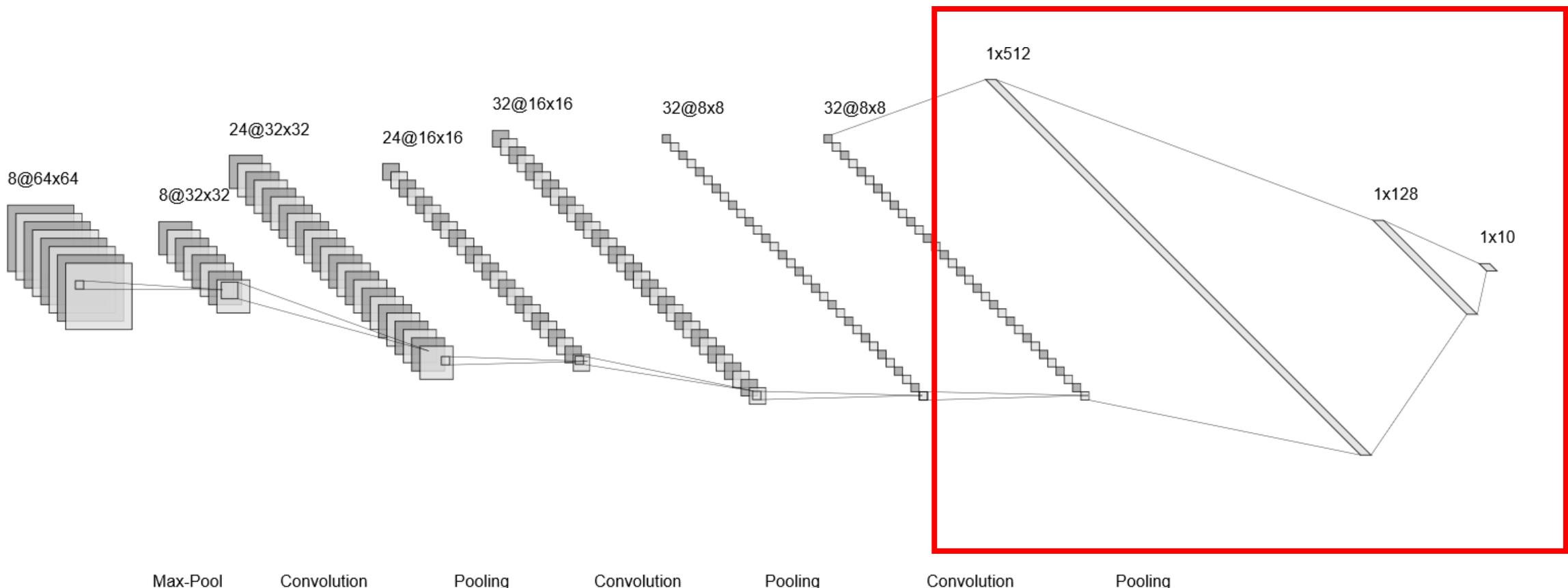
In a 2x2 support it discards 75% of samples in a volume

Dense Layers

As in feed-forward NN

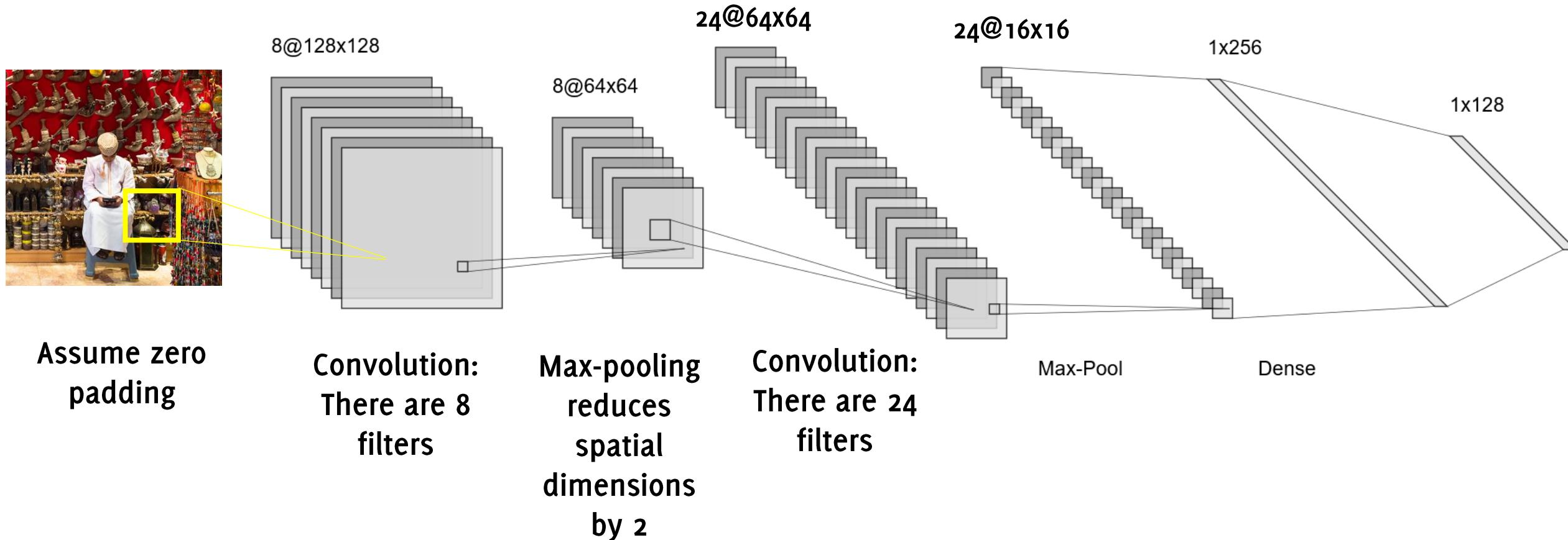
The dense layers

Here the spatial dimension is lost, the CNN stacks hidden layers from a MLP NN.
It is called Dense as each output neuron is connected to each input neuron



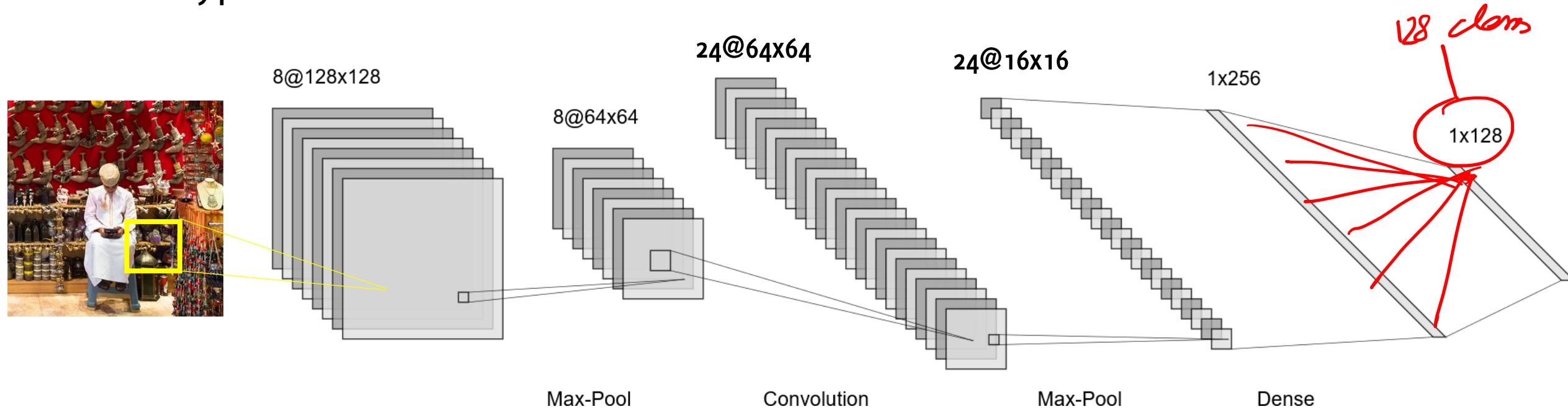
Convolutional Neural Networks (CNN)

The typical architecture of a convolutional neural network



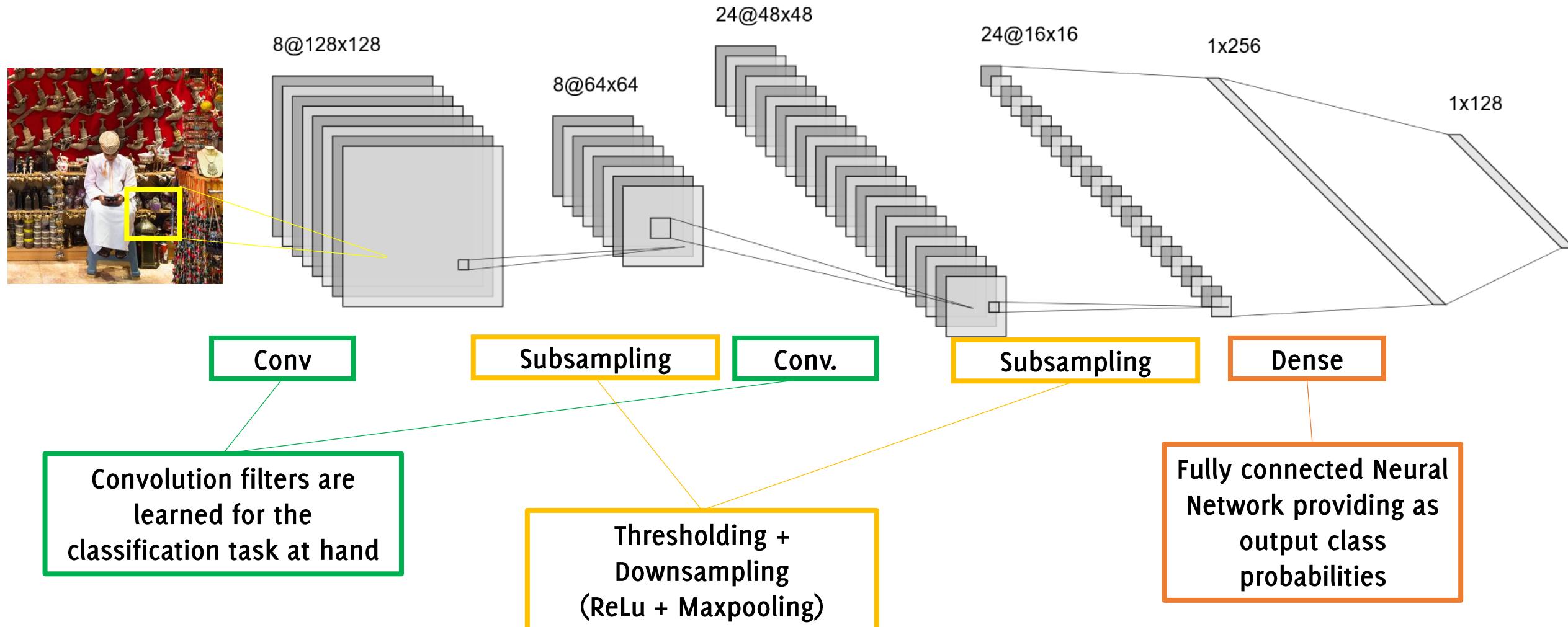
Convolutional Neural Networks (CNN)

The typical architecture of a convolutional neural network



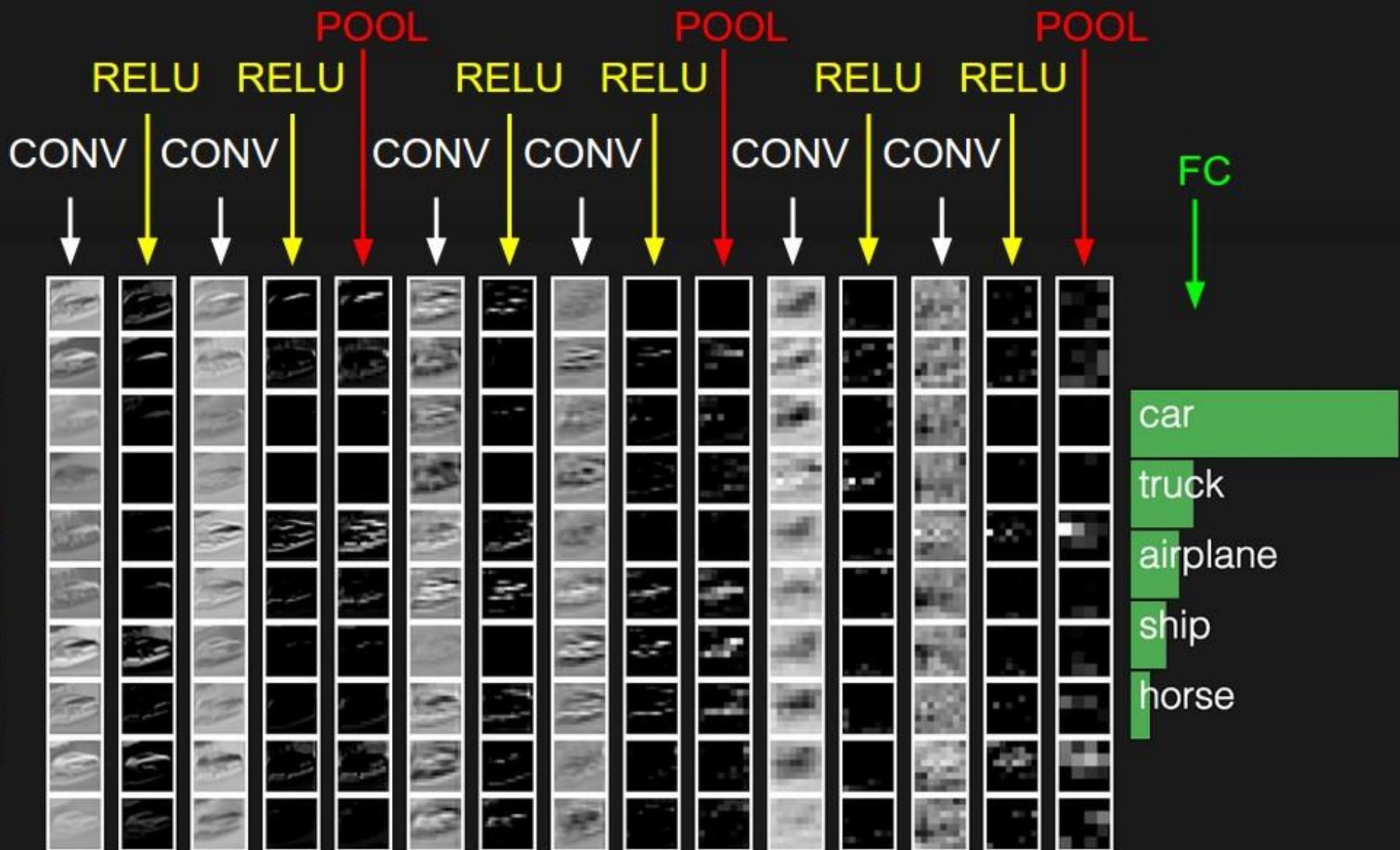
The output of the **fully connected (FC) layer** has the same size as the number of classes, and provides a score for the input image to belong to each class

Convolutional Neural Networks (CNN)

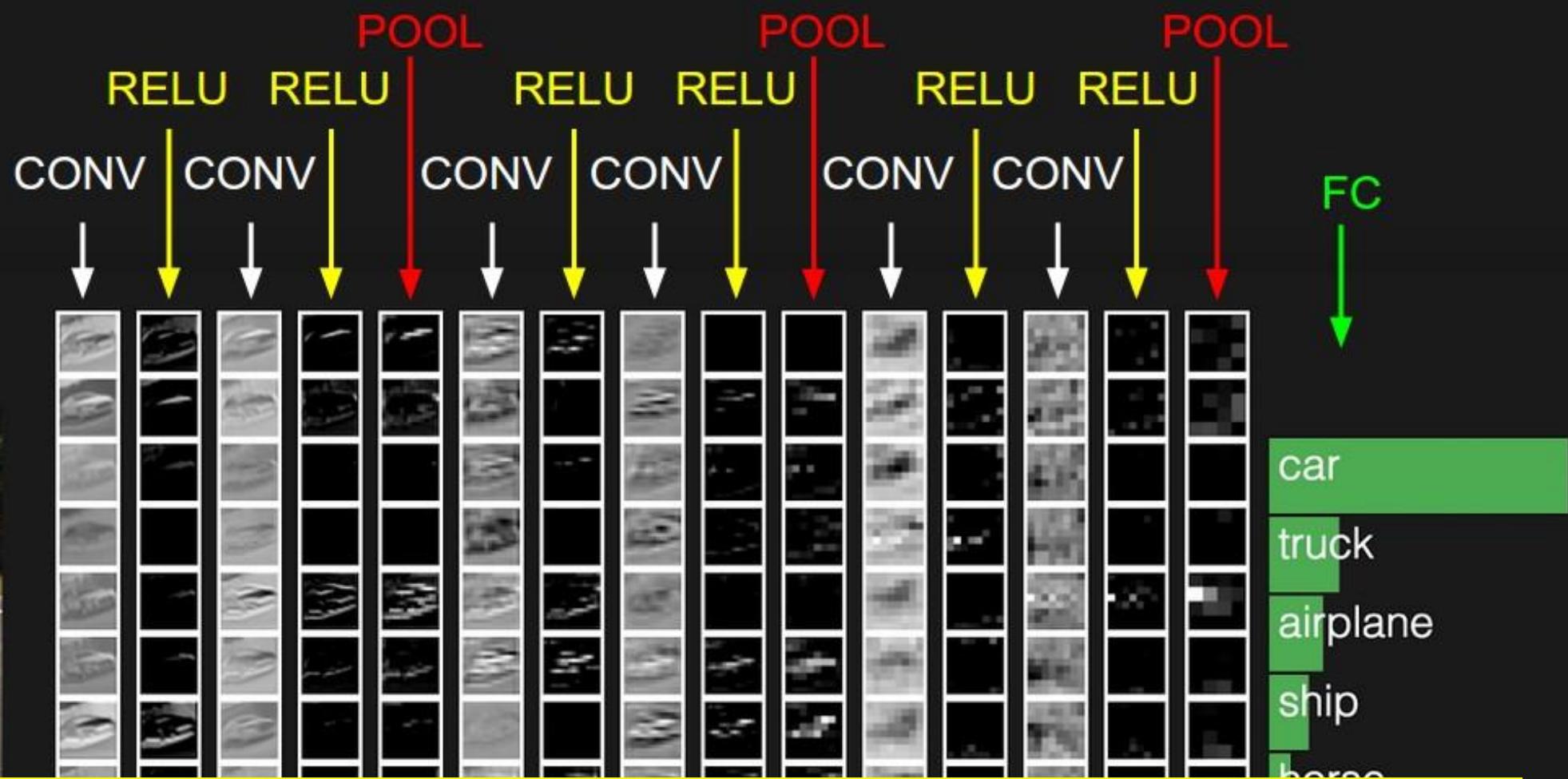


CNN «in action»

Activations in a convolutional network

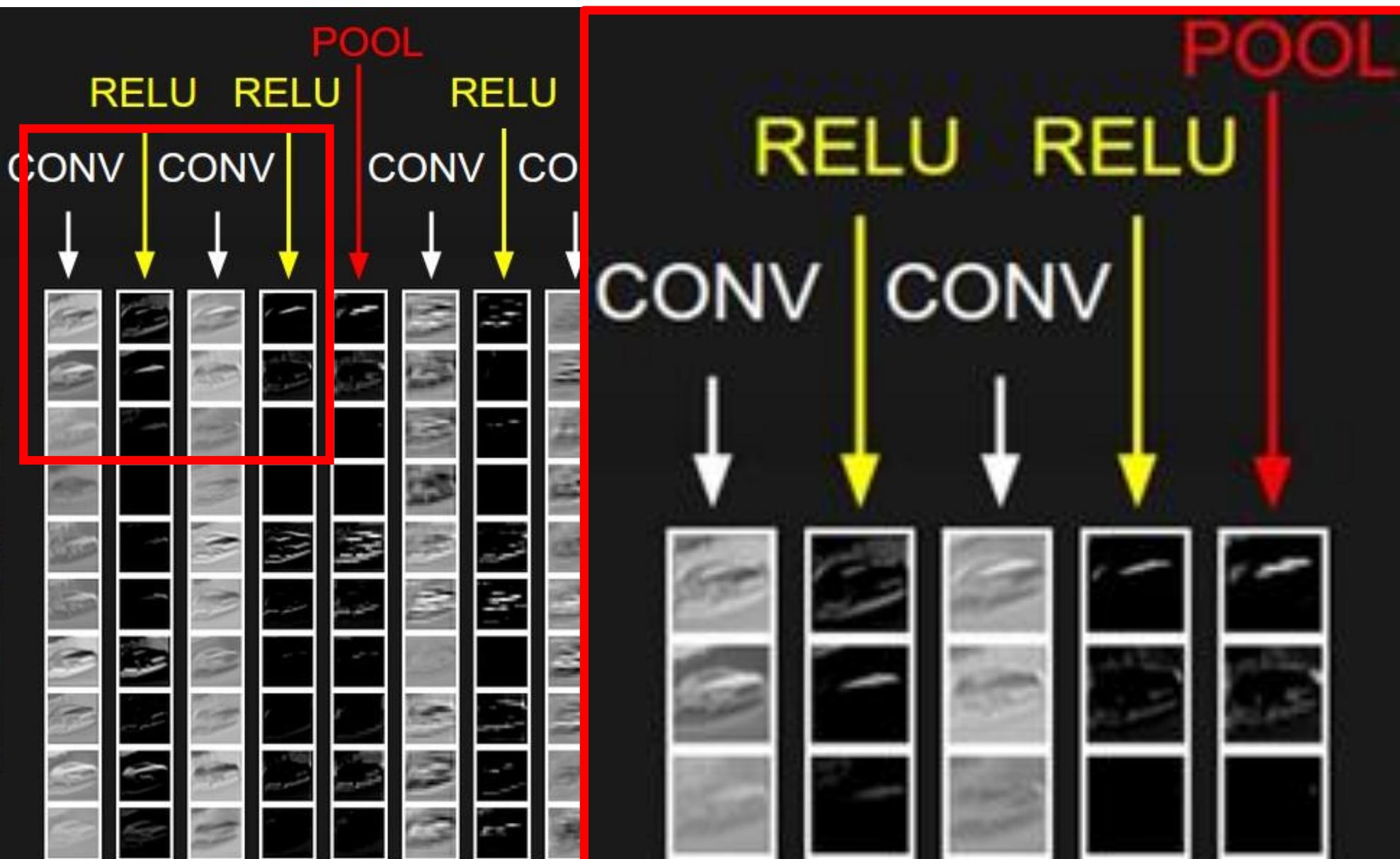


Activations in a convolutional network

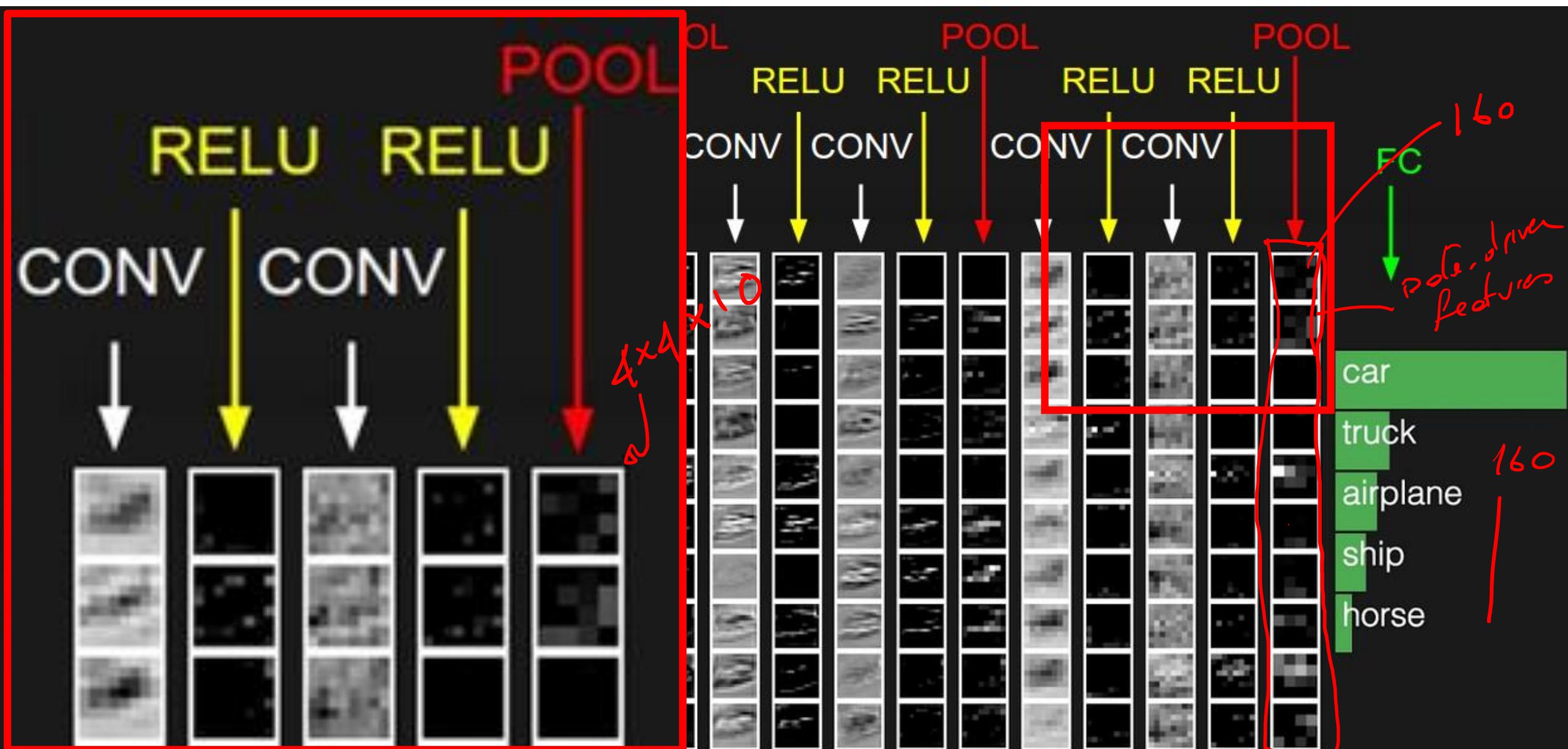


Each layer in the volume is represented as an image here
(using the same size but different resolution for visualization sake)

Activations in a convolutional network

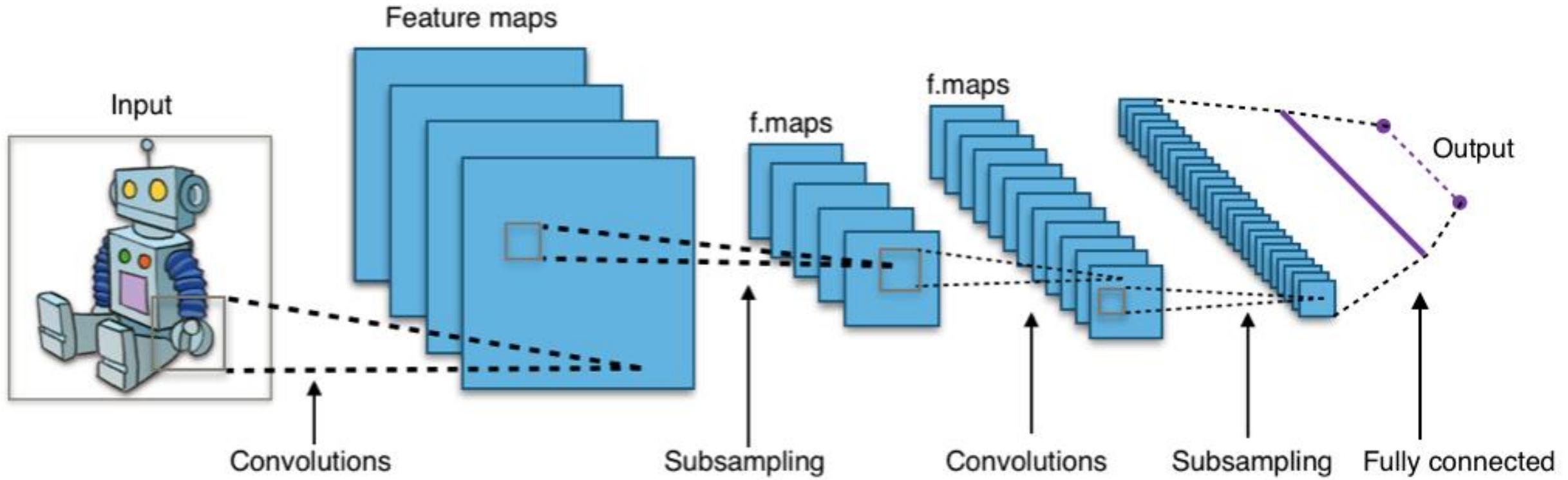


Activations in a convolutional network



Convolutional Neural Networks (CNN)

Btw, this figure contains an error.
If you are CNN-Pro, you should spot it!

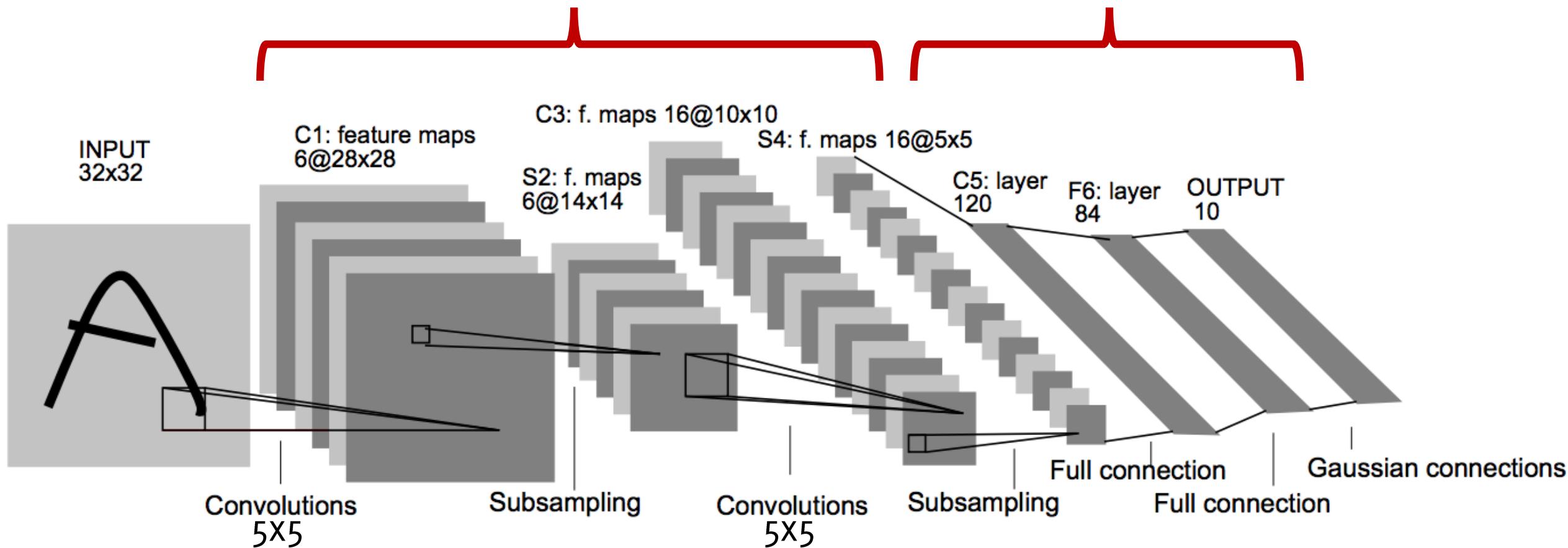


The First CNN

LeNet-5 (1998)

Stack of Conv2D + TANH + AVG-
POOLING with "valid" padding

A TRADITIONAL MLP



The first CNN

Do not use each pixel as a separate input of a large MLP, because:

- images are highly spatially correlated,
- using individual pixel of the image as separate input features would not take advantage of these correlations.

The first convolutional layer: 6 filters 5x5

The second convolutional layer: 16 filters 5x5

LeNet-5 in Keras

```
from keras.models import Sequential  
from keras.layers import Dense, Flatten, Conv2D, AveragePooling2D  
  
num_classes = 10;  
input_shape=(32, 32, 1);  
  
model = Sequential()  
  
model.add(Conv2D(filters = 6, kernel_size = (5, 5), activation='tanh', input_shape=input_shape, padding = 'valid'))  
model.add(AveragePooling2D(pool_size=(2, 2)))  
model.add(Conv2D(filters = 16, kernel_size = (5, 5), activation='tanh', padding = 'valid'))  
model.add(AveragePooling2D(pool_size=(2, 2)))  
model.add(Flatten())  
model.add(Dense(120, activation='relu'))  
model.add(Dense(84, activation='relu'))  
model.add(Dense(num_classes, activation='softmax'))
```

model.summary()

Layer (type)	Output Shape	Param #
=====		
conv2d_1 (Conv2D)	(None, 28, 28, 6)	...
average_pooling2d_1 (Average)	(None, 14, 14, 6)	...
conv2d_2 (Conv2D)	(None, 10, 10, 16)	...
average_pooling2d_2 (Average)	(None, 5, 5, 16)	...
flatten_1 (Flatten)	(None, 400)	...
dense_1 (Dense)	(None, 120)	...
dense_2 (Dense)	(None, 84)	...
dense_3 (Dense)	(None, 10)	...
=====		

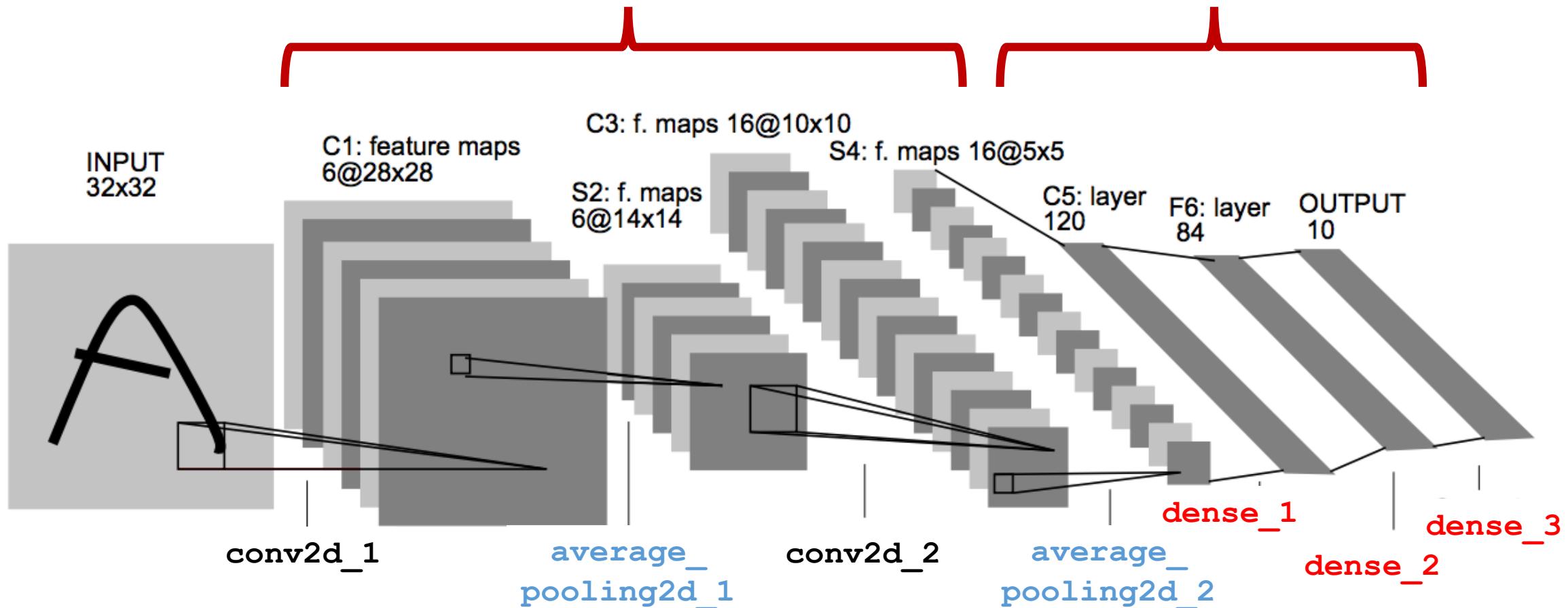
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0

model.summary()

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 6)	156 (6 x 5 x 5 + 6)
average_pooling2d_1 (Average)	(None, 14, 14, 6)	0
conv2d_2 (Conv2D)	(None, 10, 10, 16)	2416 (16 x 5 x 5 x 6 + 16)
average_pooling2d_2 (Average)	(None, 5, 5, 16)	0
flatten_1 (Flatten)	(None, 400)	0
dense_1 (Dense)	(None, 120)	48120
dense_2 (Dense)	(None, 84)	10164
dense_3 (Dense)	(None, 10)	850
Total params:	61,706	
Trainable params:	61,706	
Non-trainable params:	0	

LeNet-5 (1998)

Stack of Conv2D + RELU + AVG-POOLING A TRADITIONAL MLP



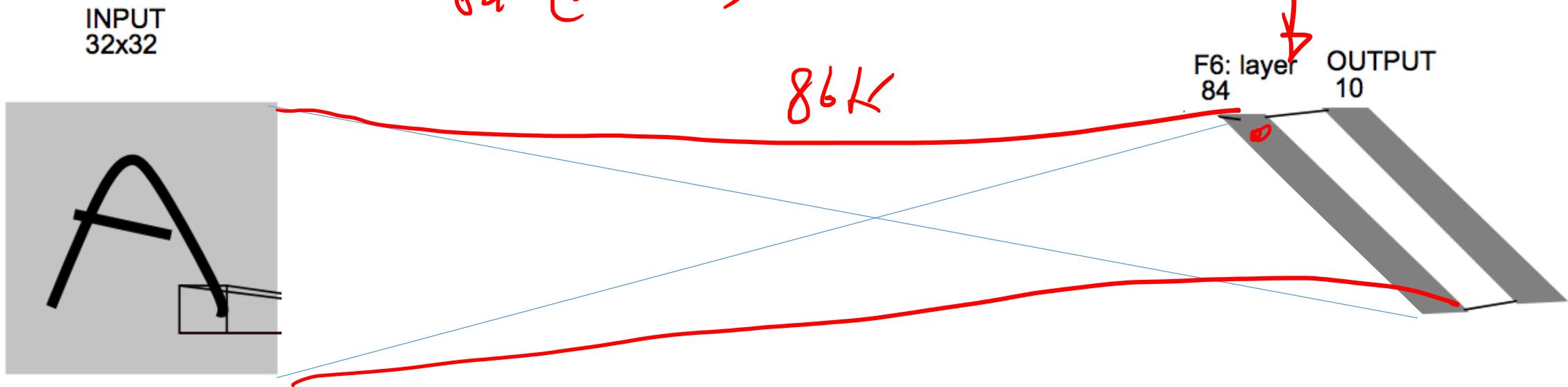
Most of parameters are in MLP

What about a MLP taking as input the whole image?

Input $32 \times 32 = 1024$ pixels, fed to a 84 neurons (the last FC layers of the network) -> 86950 parameters: $1024 * 84 + 84 + 84 * 10 + 10$

$$84 \times (32 \times 32) + 84 +$$

86K



Most of parameters are in MLP

What about a MLP taking as input the whole image?

Input $32 \times 32 = 1024$ pixels, fed to a 84 neurons (the last FC layers of the network) $\rightarrow 86950$ parameters

But.. If you take an RGB input: $32 \times 32 \times 3$,

CNN: only the nr. of parameters in the filters at the first layer increases

$$\begin{aligned} 156 + 61550 &\rightarrow 456 + 61550 \\ (6 \times 5 \times 5) &\rightarrow (6 \times 5 \times 5 \times 3) \end{aligned}$$

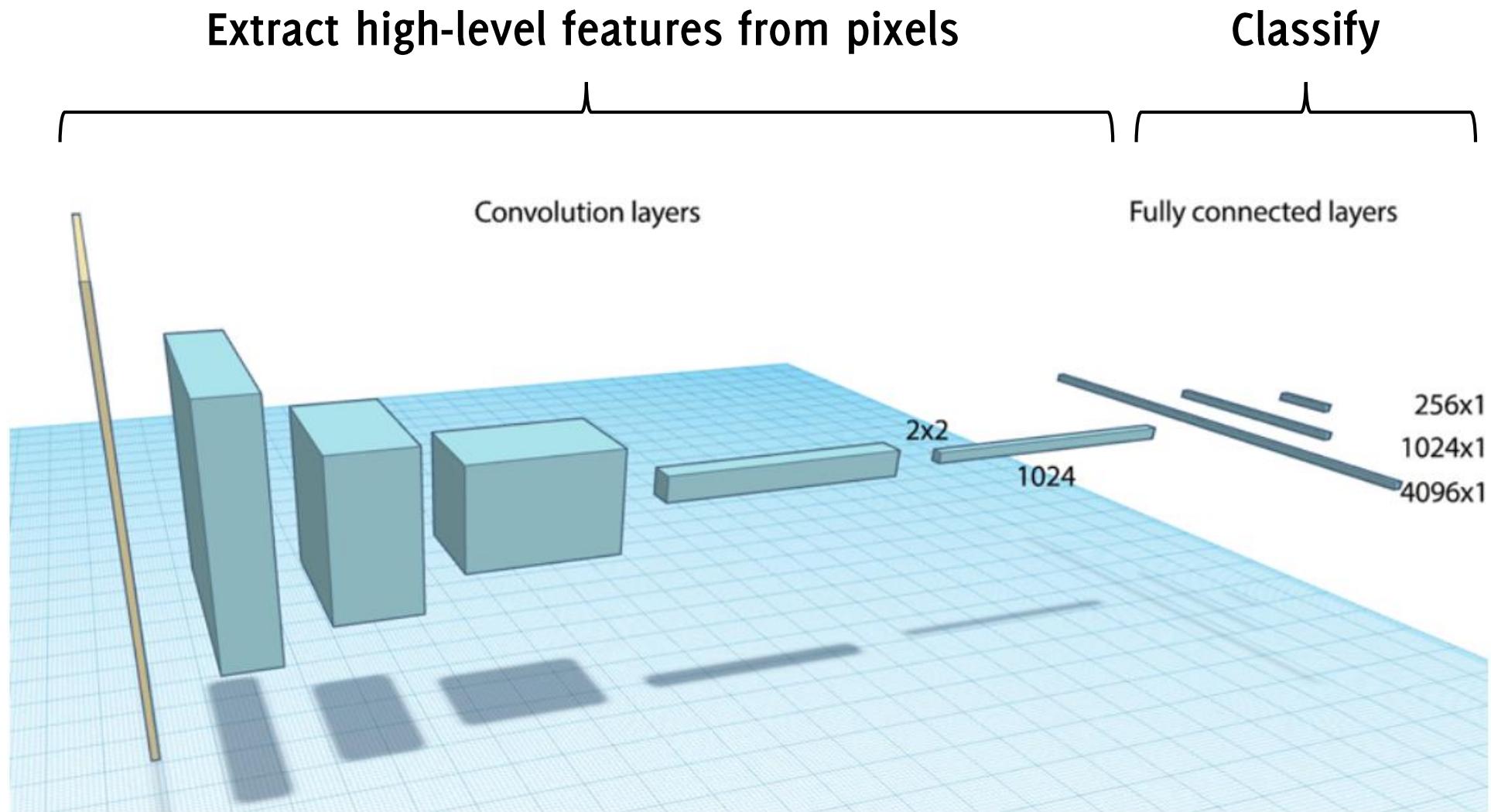
MLP: everything increases by a factor 3

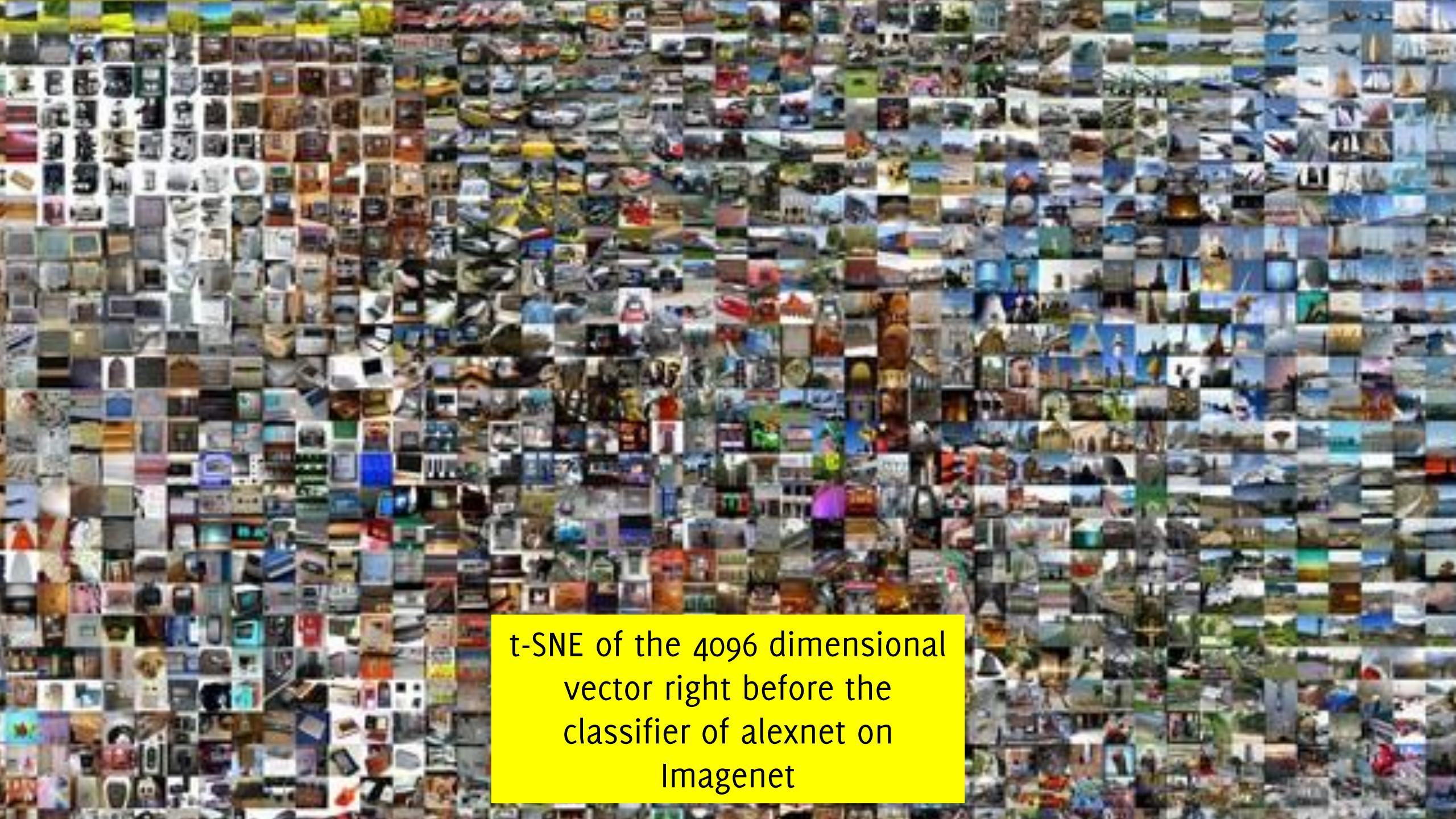
$$86950 \rightarrow 86950 \times 3$$

Latent representation in CNNs

Repeat the «t-SNE experiment» on the CIFAR dataset,
using the last layer of the CNN as vectors

A Typical Architecture





t-SNE of the 4096 dimensional
vector right before the
classifier of alexnet on
Imagenet





Distances in the latent representation of a CNN are much more meaningful than data itself

Parameters in a CNN

Convolutions as MLP

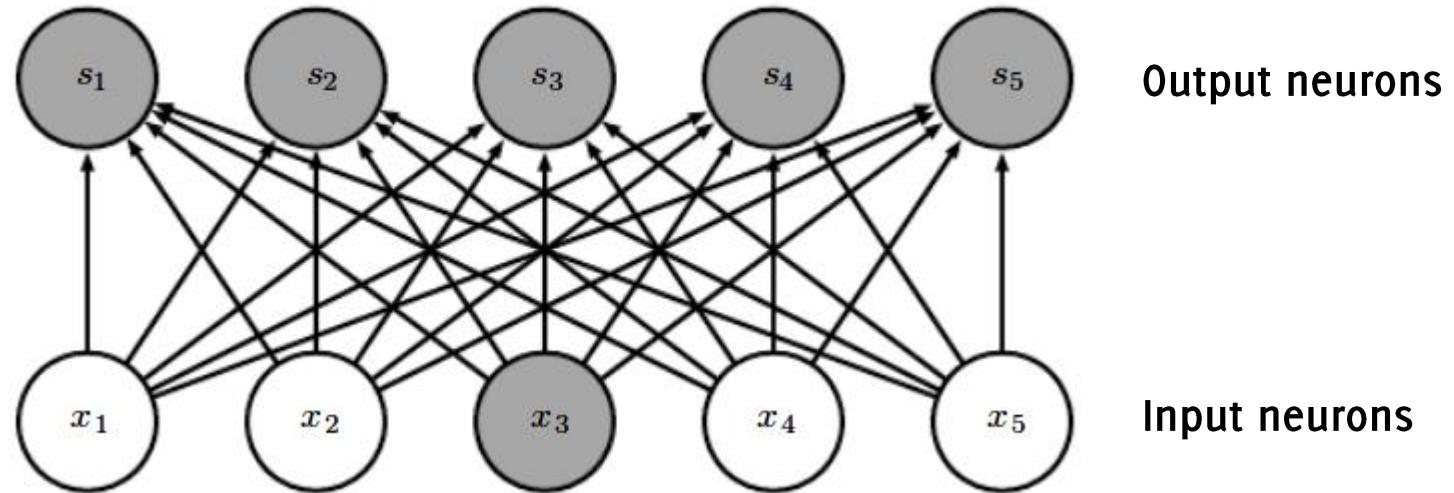
Convolution is a linear operation!

Therefore, if you unroll the input image to a vector, **you can consider convolution weights as the weights of a Multilayer Perceptron Network**

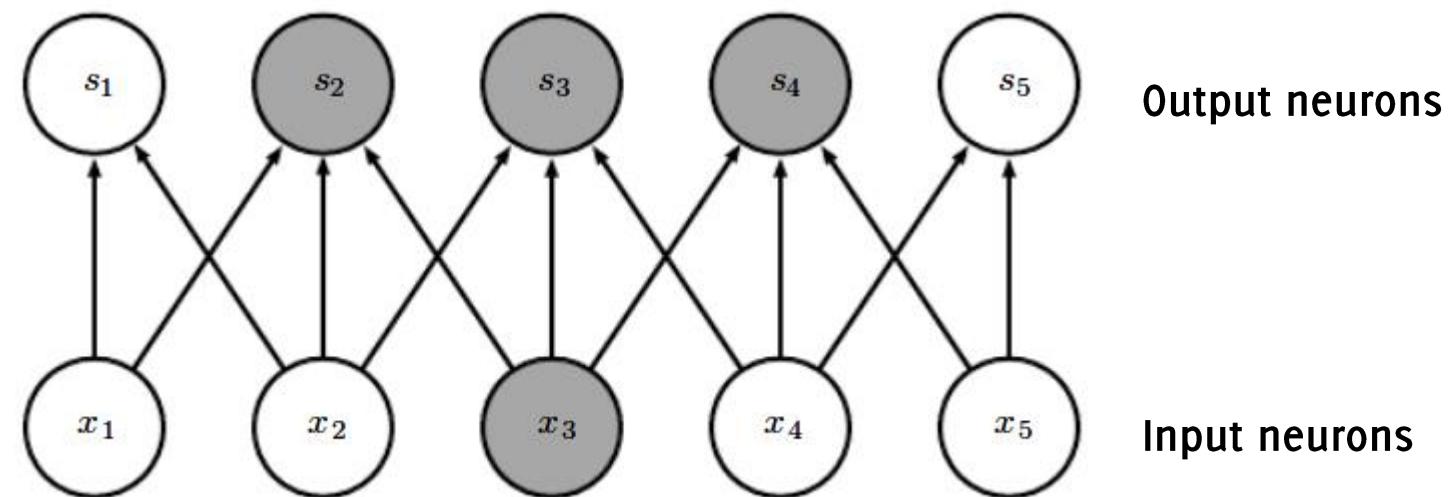
What are the differences between MLP and CNNs then?

CNNs has Sparse Connectivity

MLP, Fully connected



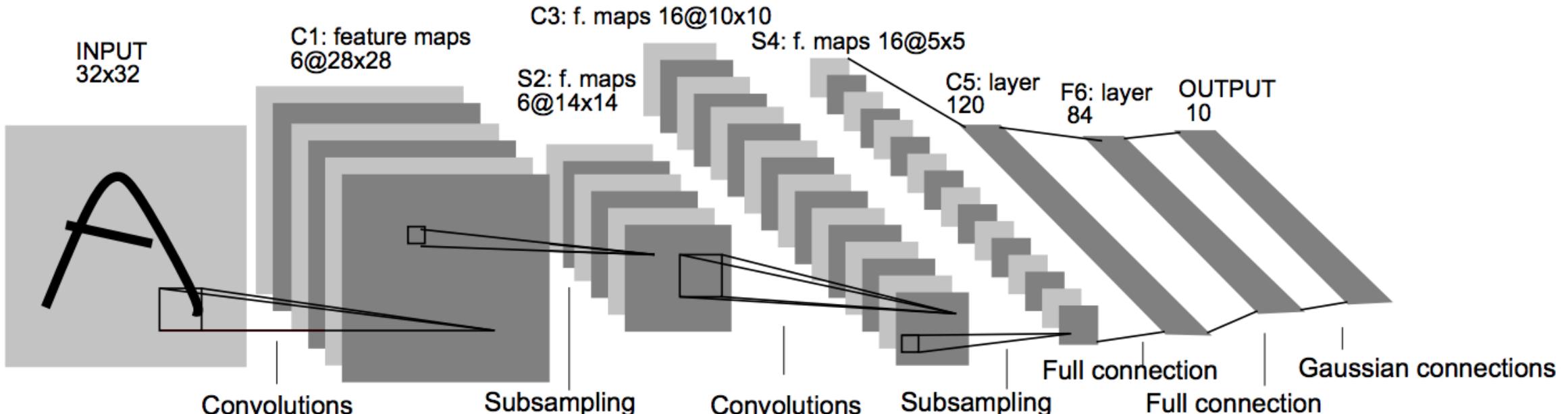
3x1 convolutional



Weight Sharing / Spatial Invariance

In a CNN, all the neurons in the same slice of a feature map use the same weights and bias: this reduces the nr. of parameters in the CNN.

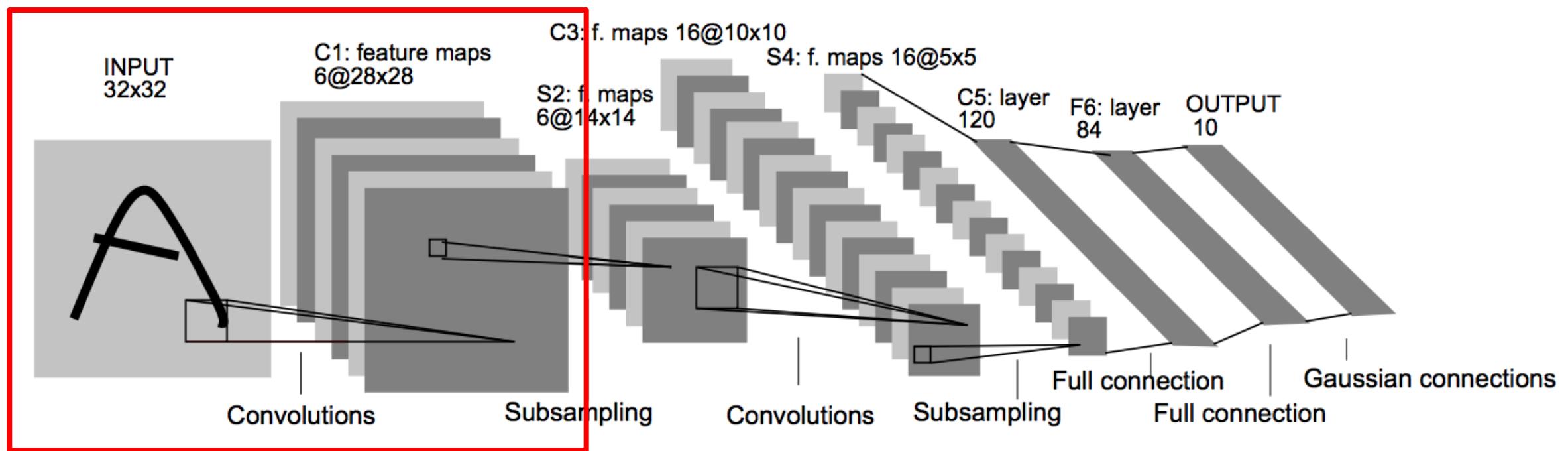
Underlying assumption: **if one feature is useful to compute at some spatial position (x,y), then it should also be useful to compute at a different position (x₂,y₂)**



Weight Sharing / Spatial Invariance

If the first layer were a MLP:

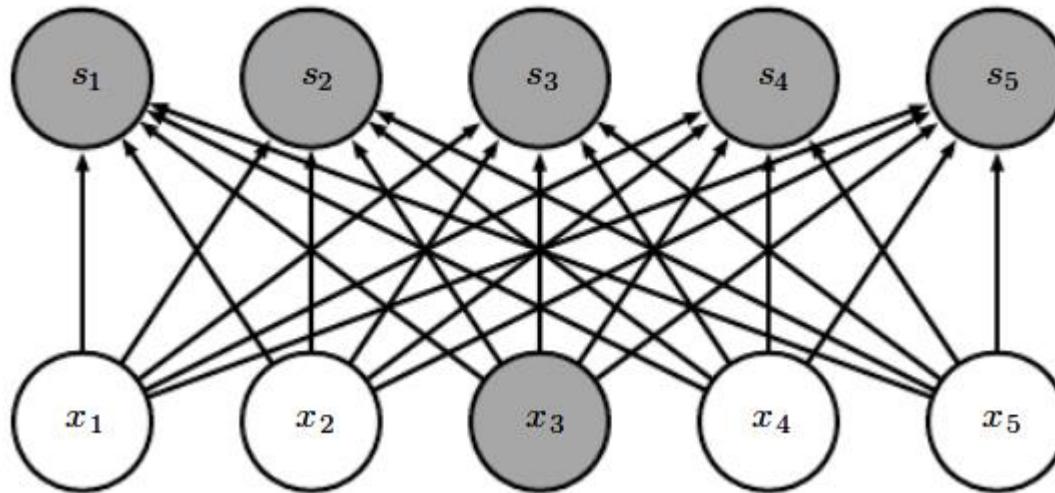
- it should have had $28 \times 28 \times 6$ neurons in the output
 - Consider sparse connectivity: 5×5 nonzero weights each neuron
 - It would be a $28 \times 28 \times 6 \times 25$ weights + $28 \times 28 \times 6$ biases (122 304)
- CNN share the same weights among a few neurons of the same layer**



Parameter sharing

Fully connected

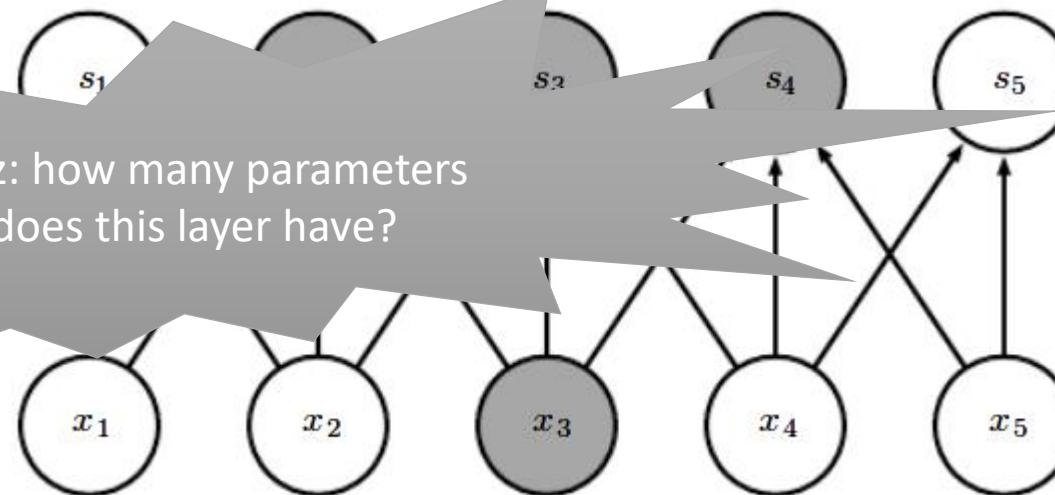
$5 \times 5 = 25$ weights
(+ 5 bias)



3x1 convolve

3 weights!
(+ 1 bias)

Quiz: how many parameters does this layer have?



The Receptive Field

A very important aspect in CNNs

The Receptive Field

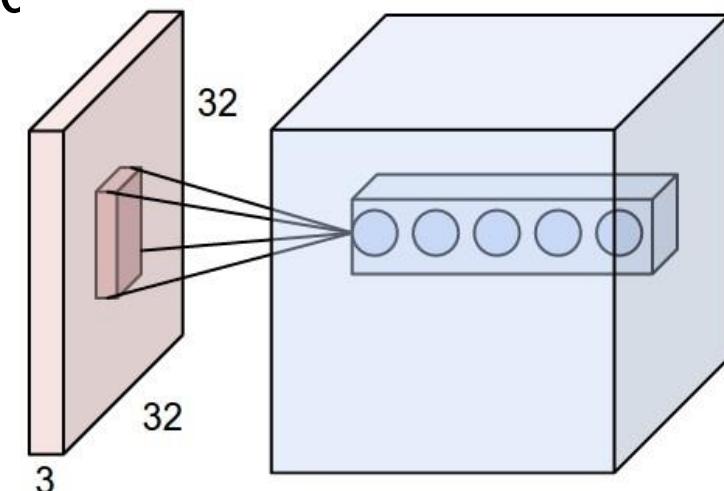
One of the basic concepts in deep CNNs.

Due to sparse connectivity, unlike in FC networks where the value of each output depends on the entire input, **in CNN an output only depends on a region of the input**.

This region in the input is the receptive field for that output

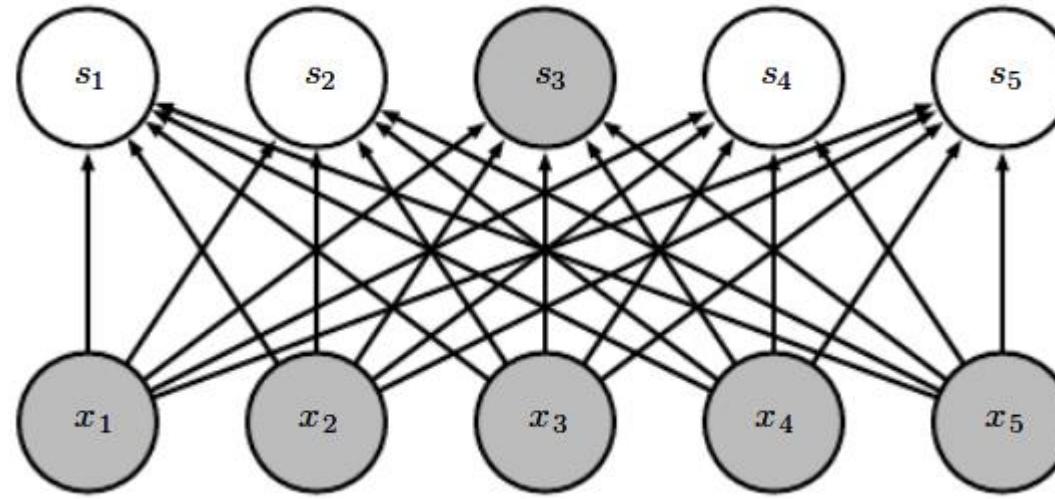
The deeper you go, the wider the receptive field is: maxpooling, convolutions and stride > 1 increase the receptive field

Usually, the receptive field refers to the final output unit of the network in relation to the network input, but the same definition holds for intermediate volumes

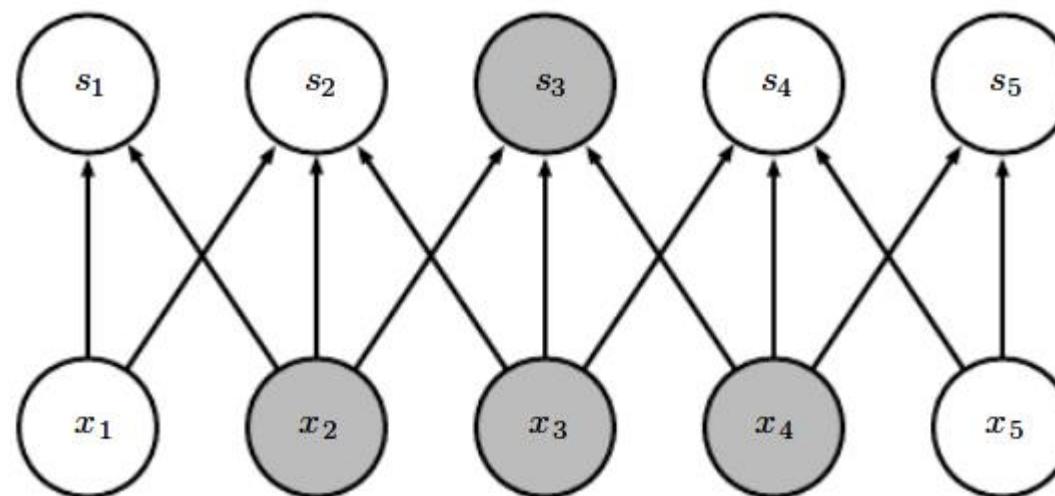


Receptive fields

MPL, Fully connected



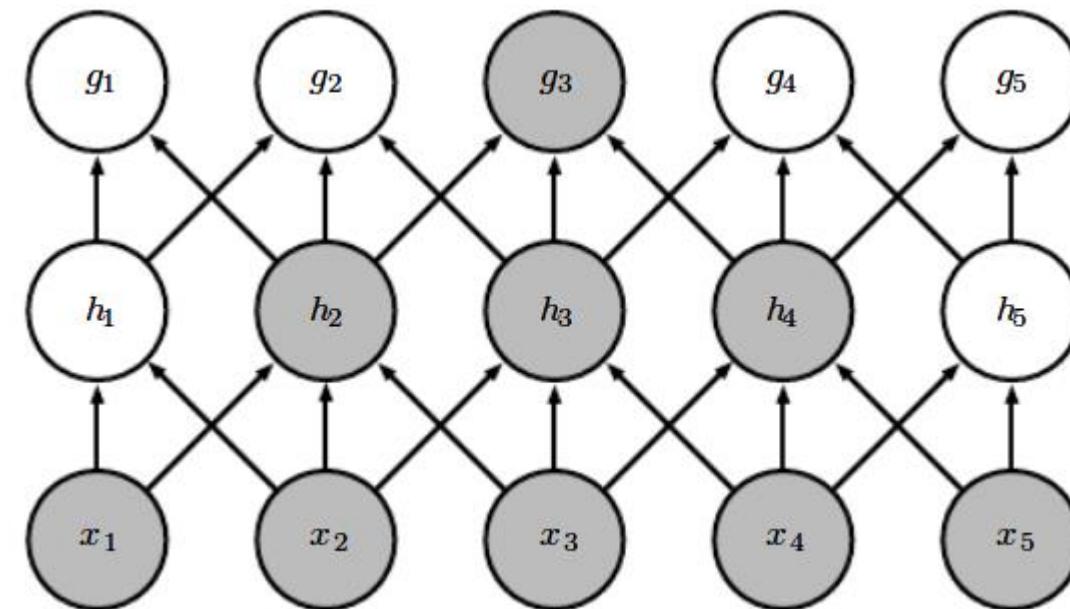
3x1 convolutional



Receptive fields

Deeper neurons depend on wider patches of the input (convolution is enough to increase receptive field, no need of maxpooling)

3x1 convolutional
3x1 convolutional



Exercise

Input:



Conv 3x3

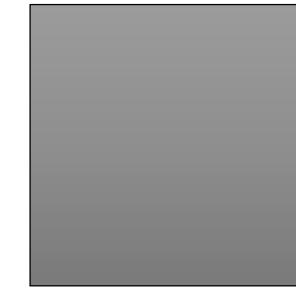
Conv 3x3

Conv 3x3

Conv 3x3

Conv 3x3

Conv 3x3



Conv 3x3

MP 2x2

Conv 3x3

MP 2x2

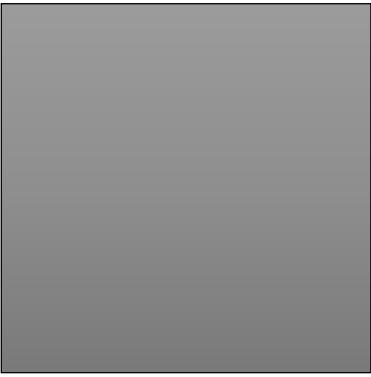
Conv 3x3

MP 2x2



Receptive fields

Input



Conv 3x3

Conv 3x3

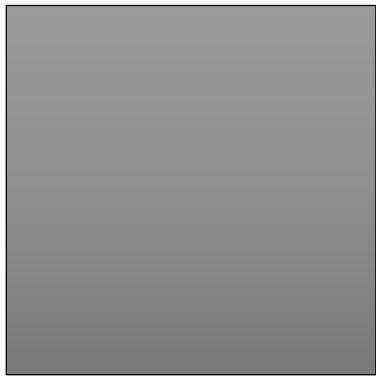
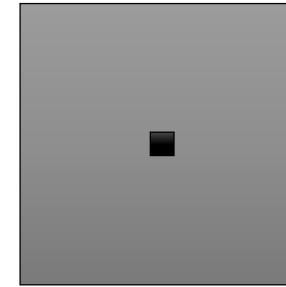
Conv 3x3

Conv 3x3

Conv 3x3

Conv 3x3

map



Conv 3x3

MP 2x2

Conv 3x3

MP 2x2

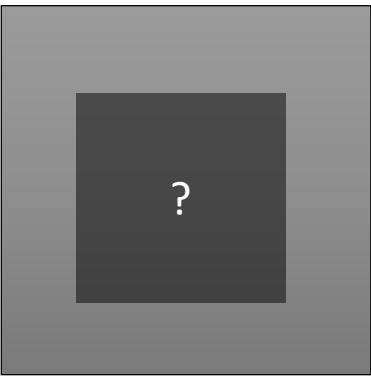
Conv 3x3

MP 2x2



Receptive fields

Input



Conv 3x3

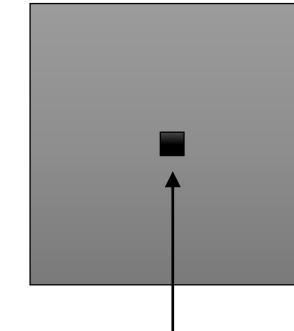
Conv 3x3

Conv 3x3

Conv 3x3

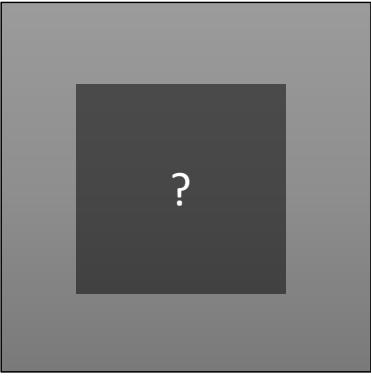
Conv 3x3

Conv 3x3



map

How large is the receptive field of the black neuron?



Conv 3x3

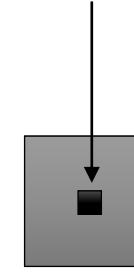
MP 2x2

Conv 3x3

MP 2x2

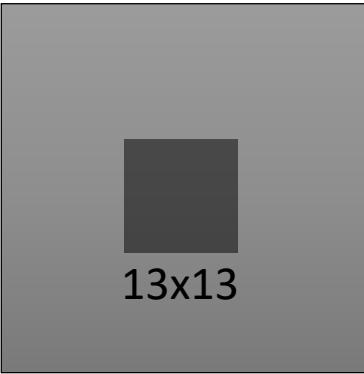
Conv 3x3

MP 2x2

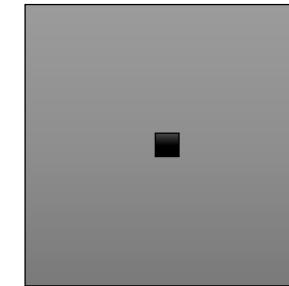


Receptive fields

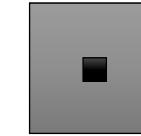
Input



map



How large is the receptive field of the black neuron?



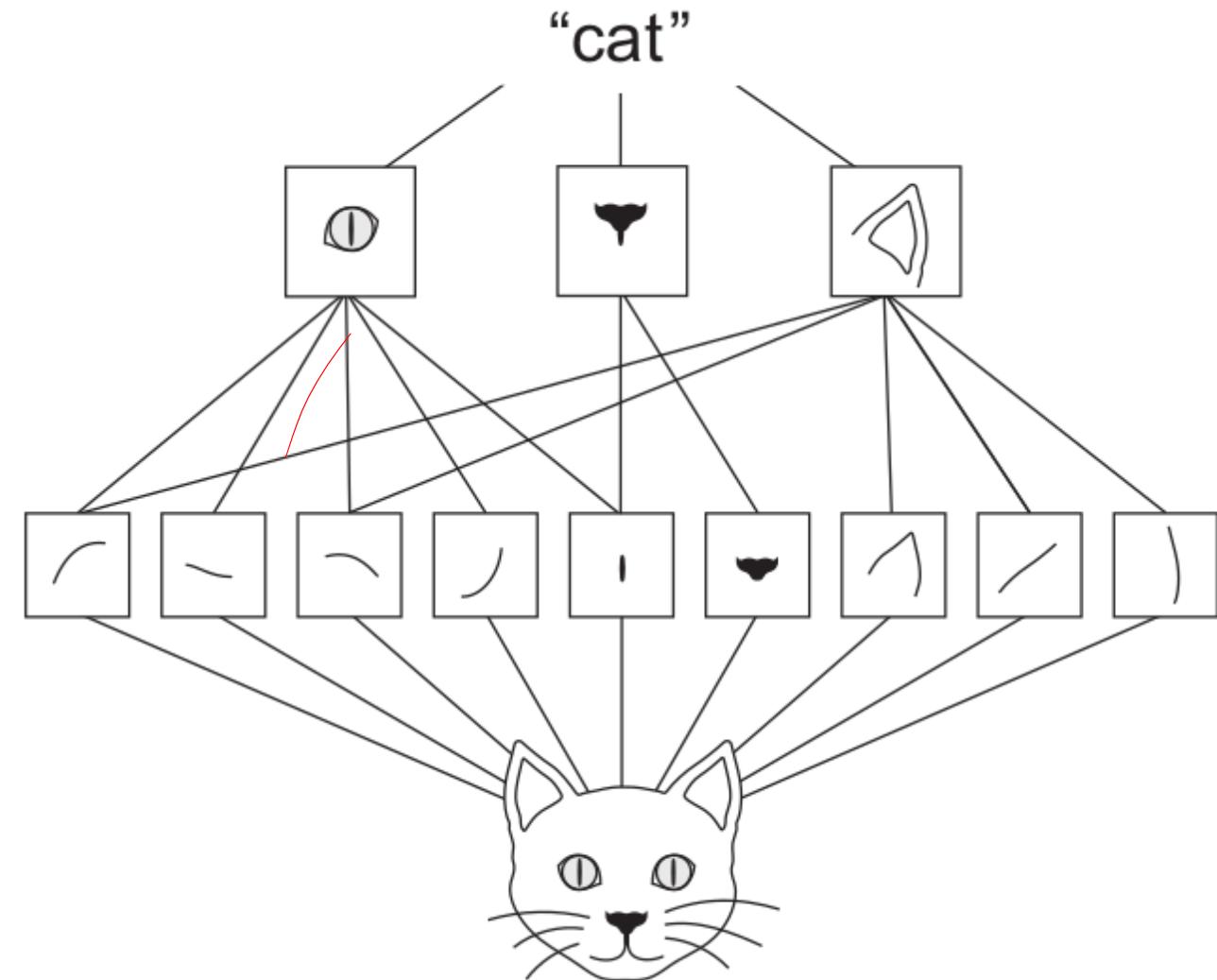
As we move deeper...

As we move to deeper layers:

- spatial resolution is reduced
- the number of maps increases

We search for **higher-level patterns**, and don't care too much about their exact location.

There are more high-level patterns than low-level details!



CNN Training

Training a CNN

- The CNN can be seen as a MLP (with sparse and shared connectivities)
- CNN can in principle be trained by gradient descent to minimize a loss function over a batch (e.g. binary cross-entropy, RMSE, Hinge loss..)
- Gradient can be computed by backpropagation (chain rule) as long as we can derive each layer of the CNN
- Weight sharing needs to be taken into account (fewer parameters to be used in the derivatives) while computing derivatives
- There are just a few details missing

Detail: backprop with max pooling

The gradient is only routed through the input pixel that contributes to the output value; e.g.:

Gradient of \bullet with respect to $\bullet = 0$



max pool with 2x2 filters
and stride 2

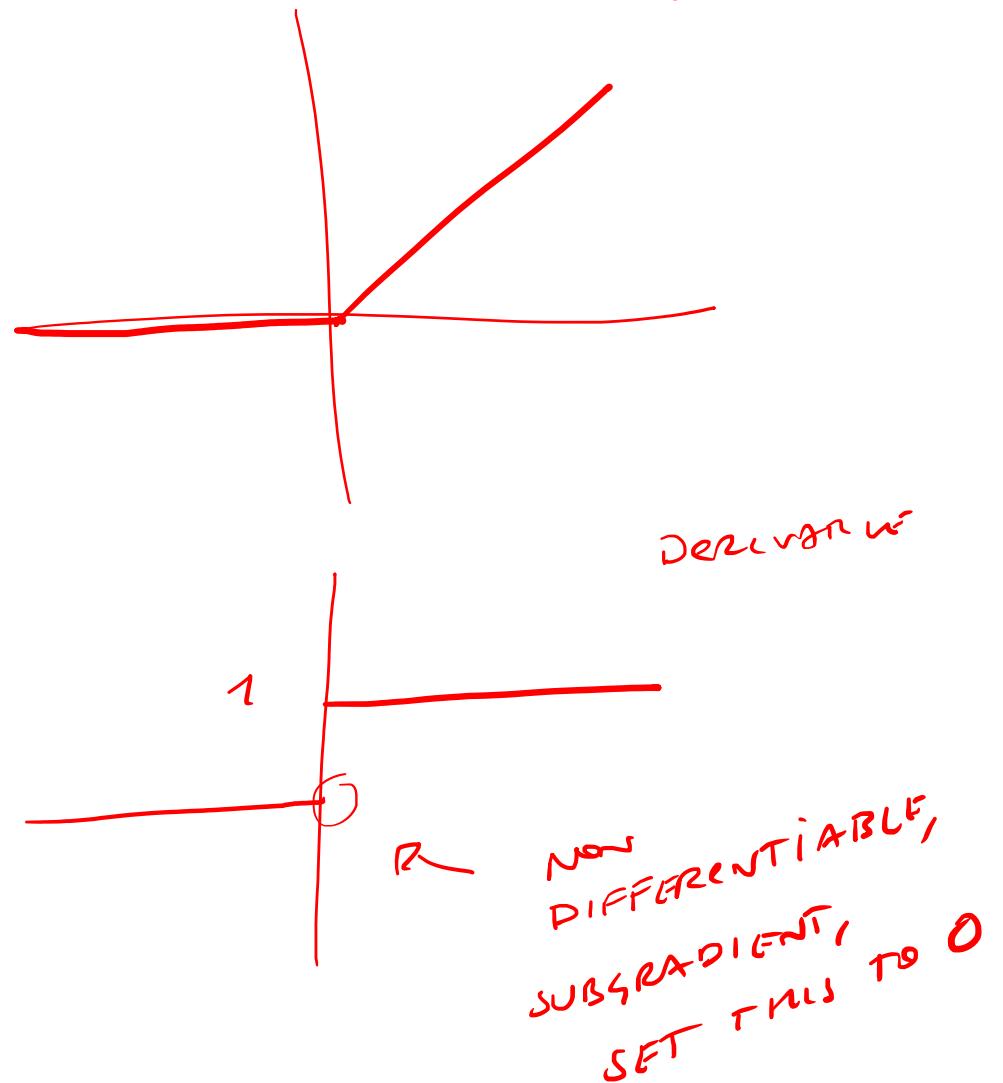


The derivative is:

- 1 at the location corresponding the maximum
- 0 otherwise

Detail: derivative of ReLU

The ReLU derivative is straightforward



Visual Recognition Problems

Setting up the stage..

Before moving to deep learning

Let's overview the visual recognition problems
we will address

Image Classification



I

$$\Lambda = \{"wheel", "cars", \dots, "castle", "baboon"\}$$

→ “wheel”



I

→ “castle”

Image Classification



I

$$\Lambda = \{"wheel", "cars", \dots, "castle", "baboon"\}$$

→ “wheel” 65%, “tyre” 30%..



I

→ “castle” 55%, “tower” 43%..

Image Classification, the problem

Assign to an input image $I \in \mathbb{R}^{R \times C \times 3}$:

- a label l from a fixed set of categories
 $\Lambda = \{"wheel", "cars", ..., "castle", "baboon"\}$

$$I \rightarrow l \in \Lambda$$

Image Classification Example

Sat, Apr 6



Thu, Apr 4



Sat, Apr 9, 2016



Inbox (39) - giacomo79@gmail.c x rabbit - Google Photos x +
← → C photos.google.com/search/rabbit

←

Search bar: rabbit

Localization



$(x, y, w, h, \text{"Hawk"})$

Localization, the problem

Assign to an input image $I \in \mathbb{R}^{R \times C \times 3}$:

- a label l from a fixed set of categories
 $\Lambda = \{"wheel", "cars", \dots, "castle", "baboon"\}$
- the coordinates (x, y, h, w) of the bounding box enclosing that object

$$I \rightarrow (x, y, h, w, l)$$

Object Detection



Object Detection, the problem

Assign to an input image $I \in \mathbb{R}^{R \times C \times 3}$:

- **multiple labels $\{l_i\}$ from a fixed set of categories $\Lambda = \{"wheel", "cars", \dots, "castle", "baboon"\}$, each corresponding to an instance of that object**
- the coordinates $\{(x, y, h, w)_i\}$ of the bounding box enclosing each object

$$I \rightarrow \{(x, y, h, w, l)_1, \dots, (x, y, h, w, l)_N\}$$

Segmentation

Objects appearing in the image:

Boat

Dining table

Person



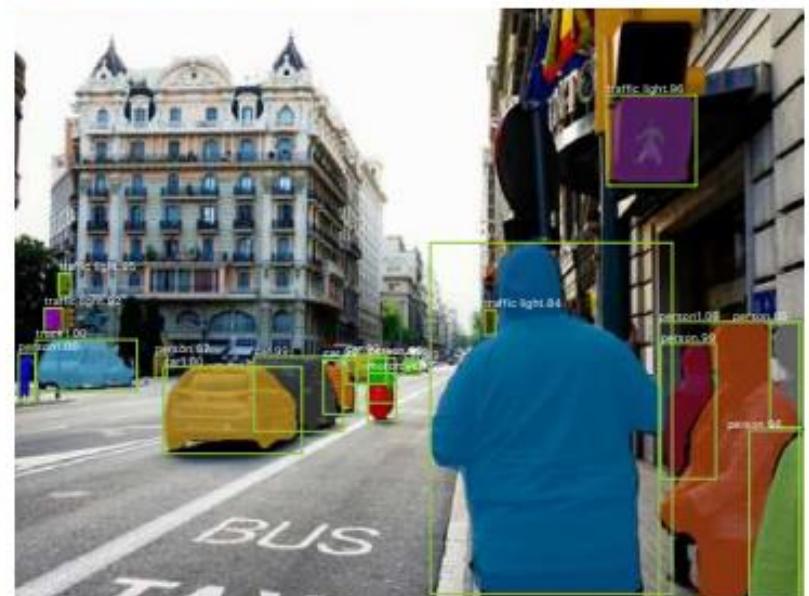
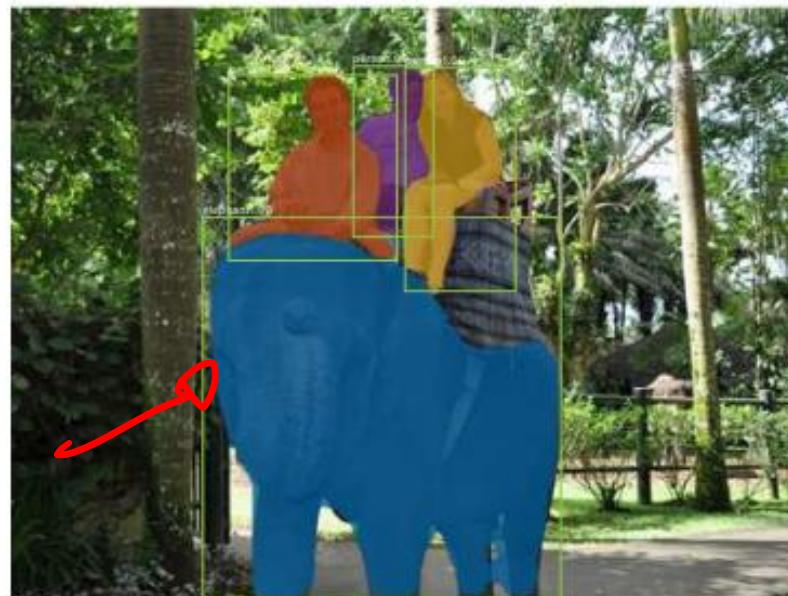
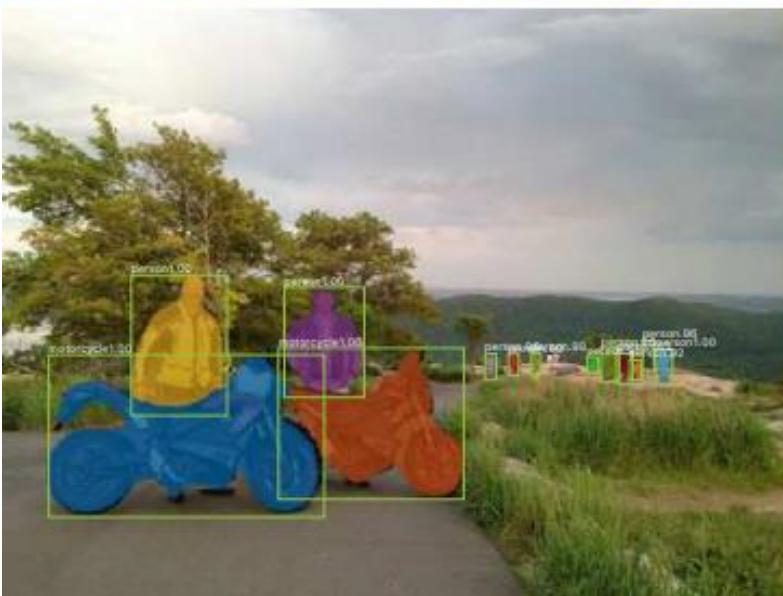
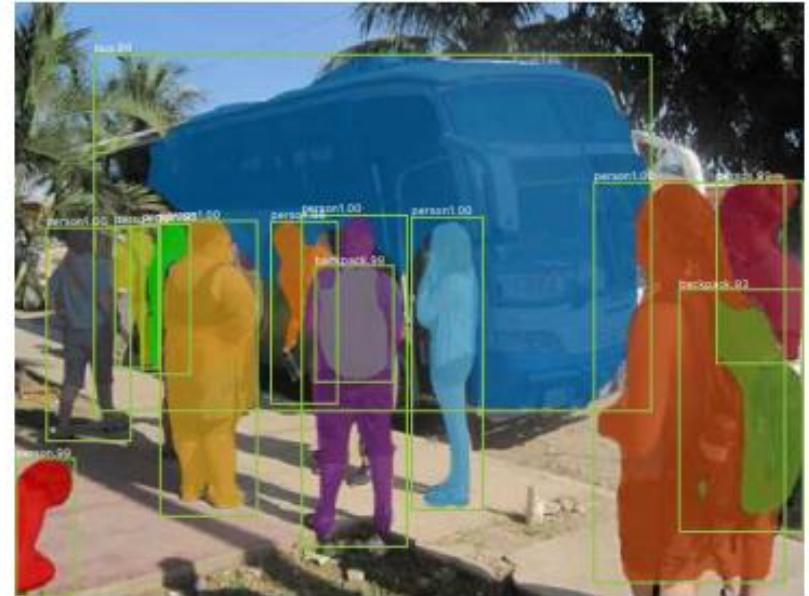
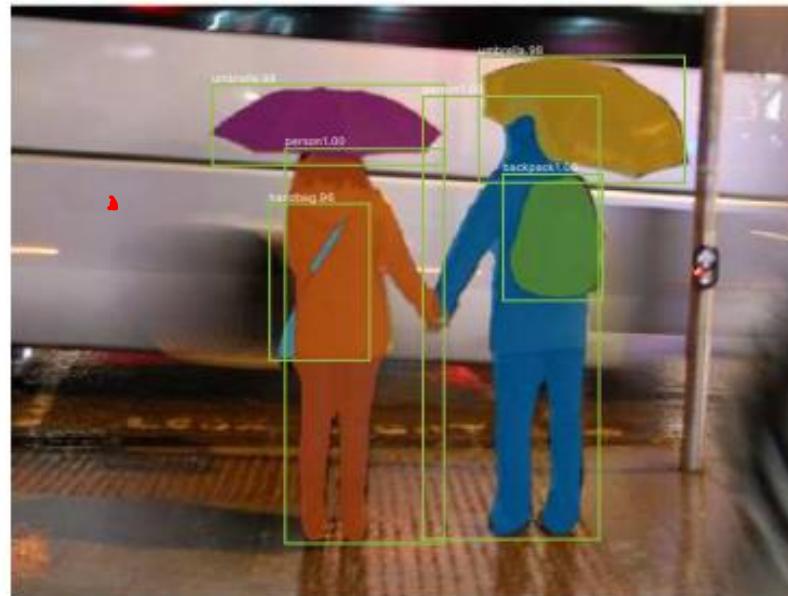
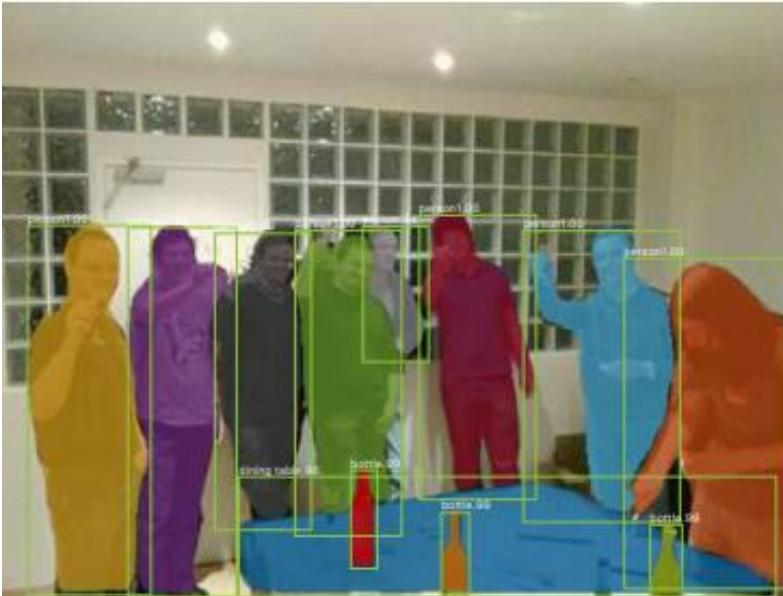
Image Segmentation, the problem

Assign to each pixel of an image $I \in \mathbb{R}^{R \times C \times 3}$:

- a label $\{l_i\}$ from a fixed set of categories
 $\Lambda = \{"wheel", "cars", \dots, "castle", "baboon"\},$
 $I \rightarrow S \in \Lambda^{R \times C}$

where $S(x, y) \in \Lambda$ denotes the class associated to the pixel (x, y)

Instance Segmentation: Mask R-CNN



Instance Segmentation, the problem

Assign to an input image I :

- **multiple labels** $\{l_i\}$ from a fixed set of categories $\Lambda = \{"wheel", "cars", \dots, "castle", "baboon"\}$, each corresponding to **an instance of that object**
- the coordinates $\{(x, y, h, w)_i\}$ of the **bounding box** enclosing each **object**
- the **set of pixels** S in each bounding box corresponding to that **label**

$$I \rightarrow \{(x, y, h, w, l, S)_1, \dots, (x, y, h, w, l, S)_N\}$$

Instance Segmentation



CNNs in Keras

What is Keras?

An open-source library providing **high-level building blocks** for developing deep-learning models in Python

Designed to enable **fast experimentation with deep neural networks**, it focuses on being **user-friendly, modular, and extensible**

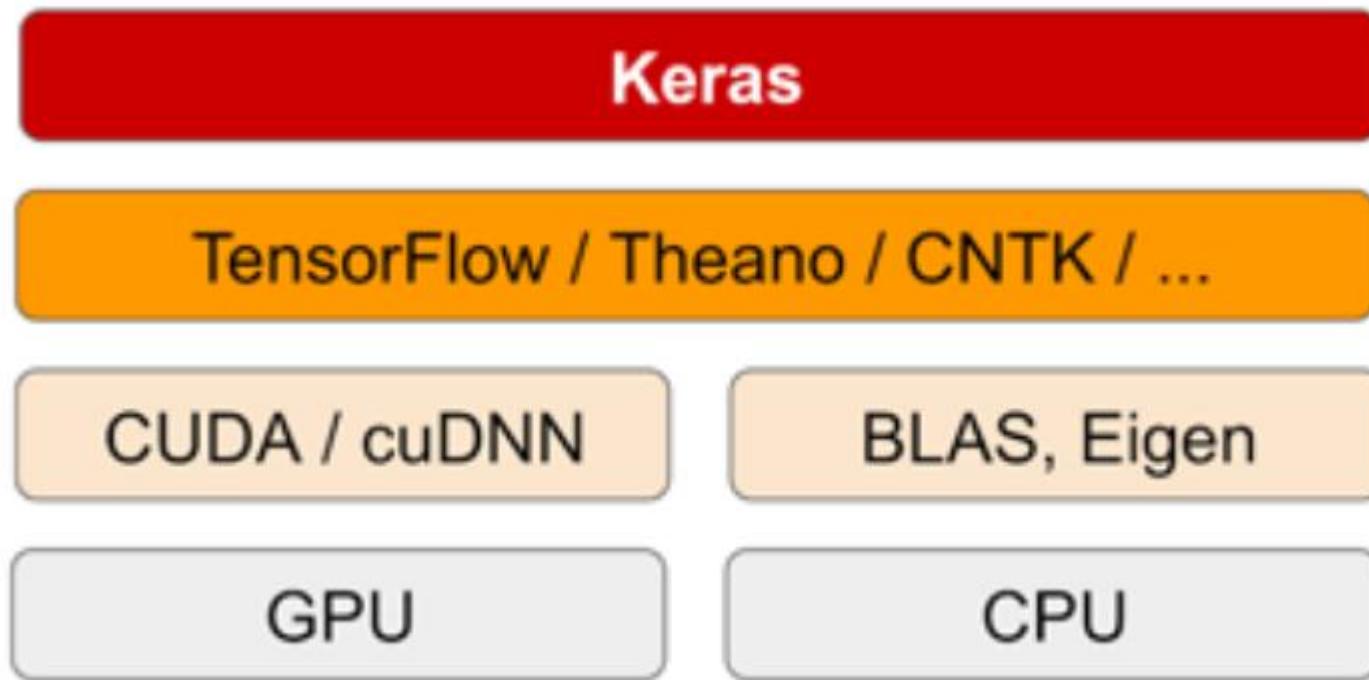
Doesn't handle low-level operations such as tensor manipulation and differentiation.

Relies on **backends** (TensorFlow, Microsoft Cognitive Toolkit, Theano, or PlaidML)

Enables full access to the backend



The software stack



Why Keras?

Pros:

Higher level → fewer lines of code

Modular backend → not tied to tensorflow

Way to go if you focus on applications

Cons:

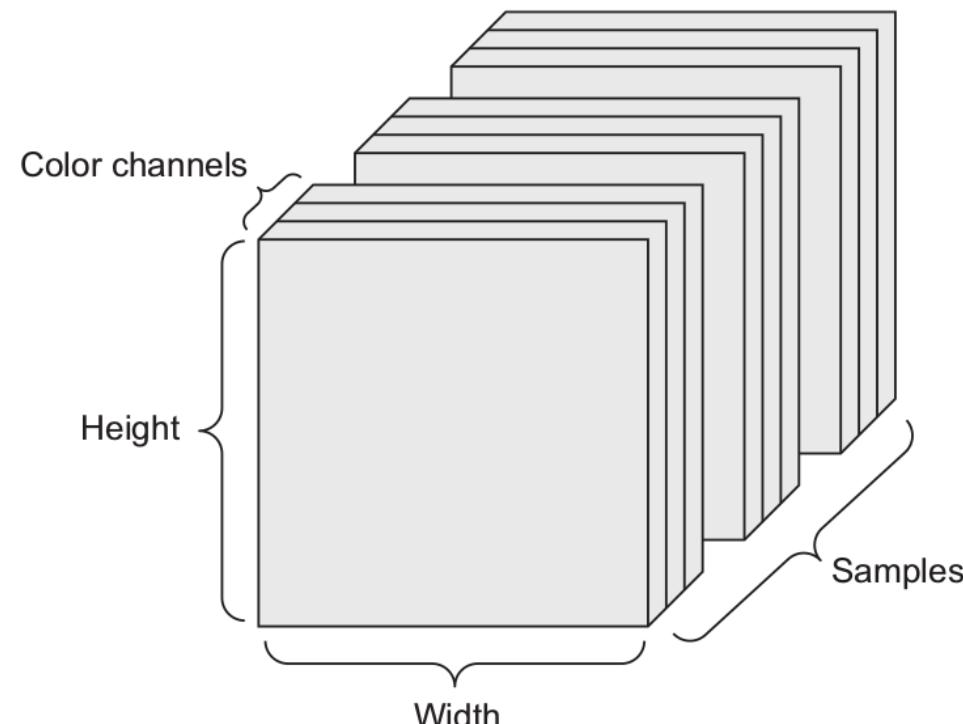
Not as flexible

Need more flexibility? Access the backend directly!

We will manipulate 4D tensors

Images are represented in 4D tensors:

Tensorflow convention: (samples, height, width, channels)



Building the Network

Convolutional Networks in Keras

```
# it is necessary to import some package
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.layers import Conv2D, MaxPooling2D

# and initialize an object from Sequential()
model = Sequential()
```

A very simple CNN

```
# Network Layers are stacked by means of the  
.add() method  
  
model.add(Conv2D(filters = 64, kernel_size=3,  
activation='relu', input_shape=(28,28,1)))  
model.add(MaxPooling2D(pool_size=(2, 2)))  
model.add(Flatten())  
model.add(Dense(10, activation='softmax'))
```

Convolutional Layers

```
# Convolutional Layer  
model.add(Conv2D(filters = 64, kernel_size=3,  
activation='relu', input_shape=(28,28,1)))  
  
# the input are meant to define:  
# - The number of filters,  
# - The spatial size of the filter (assumed squared), while the depth depends on the network structure  
# - the activation layer (always include a nonlinearity after the convolution)  
# - the input size: (rows, cols, n_channels)
```

Convolutional Layers

```
# Convolutional Layer  
  
model.add(Conv2D(filters = 64, kernel_size=3,  
activation='relu', input_shape=(28,28,1)))  
  
# This layer creates a convolution kernel that  
is convolved with the layer input to produce a  
tensor of outputs.  
  
# When using this layer as the first layer in a  
model, provide the keyword argument input_shape  
(tuple of integers, does not include the batch  
axis), e.g. input_shape=(128, 128, 3) for  
128x128 RGB pictures in  
data_format="channels_last".
```

Conv2D help

Arguments

filters: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).

kernel_size: An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.

strides: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.

padding: one of "valid" or "same" (case-insensitive). Note that "same" is slightly inconsistent across backends with strides != 1, as described here

data_format: A string, one of "channels_last" or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, height, width, channels) while "channels_first" corresponds to inputs with shape (batch, channels, height, width). It defaults to the image_data_format value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

Conv2D help

Arguments

dilation_rate: an integer or tuple/list of 2 integers, specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any stride value != 1.

activation: Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: $a(x) = x$).

use_bias: Boolean, whether the layer uses a bias vector.

kernel_initializer: Initializer for the kernel weights matrix (see initializers).

bias_initializer: Initializer for the bias vector (see initializers).

kernel_regularizer: Regularizer function applied to the kernel weights matrix (see regularizer).

bias_regularizer: Regularizer function applied to the bias vector (see regularizer).

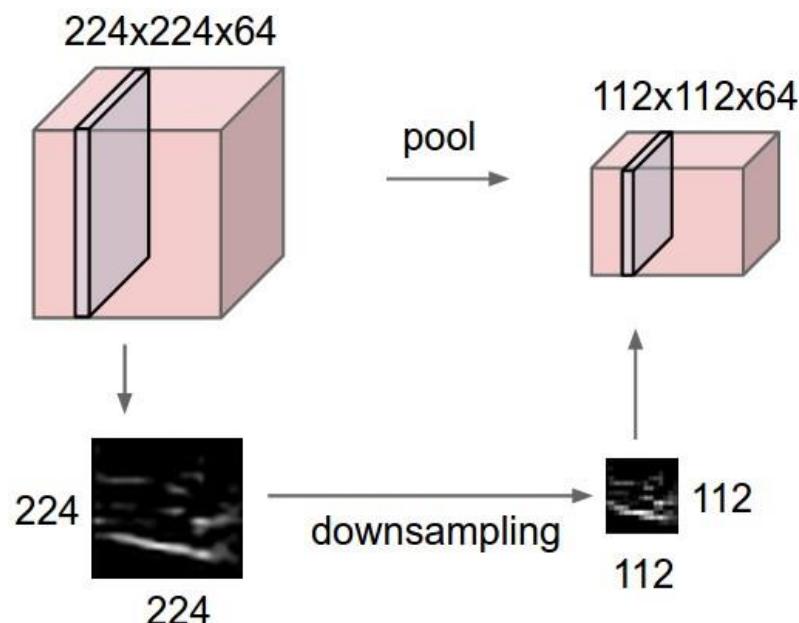
activity_regularizer: Regularizer function applied to the output of the layer (its "activation"). (see regularizer).

kernel_constraint: Constraint function applied to the kernel matrix (see constraints).

bias_constraint: Constraint function applied to the bias vector (see constraints).

MaxPooling Layers

```
# Maxpooling layer  
model.add(MaxPooling2D(pool_size=(2, 2)))  
# the only parameter here is the (spatial) size  
to be reduced by the maximum operator
```



MaxPooling2D help

Arguments:

pool_size: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.

strides: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to pool_size.

padding: One of "valid" or "same" (case-insensitive).

data_format: A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, height, width, channels) while channels_first corresponds to inputs with shape (batch, channels, height, width). It defaults to the image_data_format value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be "channels_last".

MaxPooling2D help

Input shape:

If `data_format='channels_last'`: 4D tensor with shape: (`batch_size`, `rows`, `cols`, `channels`)

If `data_format='channels_first'`: 4D tensor with shape: (`batch_size`, `channels`, `rows`, `cols`)

Output shape:

If `data_format='channels_last'`: 4D tensor with shape: (`batch_size`, `pooled_rows`, `pooled_cols`, `channels`)

If `data_format='channels_first'`: 4D tensor with shape: (`batch_size`, `channels`, `pooled_rows`, `pooled_cols`)

Fully Connected Layers

```
# at the end the activation maps are "flattened" i.e.  
# they move from an image to a vector (just unrolling)  
model.add(Flatten())  
  
# Dense is a Fully Connected layer in a traditional  
# Neural Network.  
  
model.add(Dense(units=10, activation='softmax'))  
  
# Implements:  
# output = activation(dot(input, kernel) + bias)  
• activation is the element-wise activation function  
  passed as the activation argument,  
• kernel is a weights matrix created by the layer,  
• bias is a bias vector created by the layer  
# "Units" defines the number of neurons
```

Visualizing the model

```
# a nice output describing the model  
architecture  
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
conv2d_7 (Conv2D)	(None, 26, 26, 64)	640
=====		
flatten_3 (Flatten)	(None, 43264)	0
=====		
dense_4 (Dense)	(None, 10)	432650
=====		

Total params: 433,290

Trainable params: 433,290

Non-trainable params: 0

Training the Model

Compiling the model

Then we need to compile the model using the `compile` method and specifying:

- **optimizer** which controls the learning rate. Adam is generally a good option as it adjusts the learning rate throughout training.
- **loss function** the most common choice for classification is ‘categorical_crossentropy’ for our loss function. The lower the better.
- **Metric** to assess model performance, ‘accuracy’ is more interpretable

```
model.compile(optimizer='adam',  
              loss='categorical_crossentropy',  
              metrics=['accuracy'])
```

Training the model using

The **fit()** method of the model is used to train the model.

Specify at least the following inputs:

- training data (input images),
- target data (corresponding labels in categorical format),
- validation data (a pair of data, labels to be used only for computing validation performance)
- number of epochs (number of times the whole dataset is scanned during training)

```
model.fit(x_train, y_train,  
validation_data=(x_test, y_test), epochs=3)
```

Training output

Epoch 22/100

```
18000/18000 [=====] - 136s 8ms/step - loss: 0.7567  
- acc: 0.6966 - val_loss: 1.9446 - val_acc: 0.4325
```

Epoch 23/100

```
18000/18000 [=====] - 137s 8ms/step - loss: 0.7520  
- acc: 0.6959 - val_loss: 1.9646 - val_acc: 0.4275
```

Epoch 24/100

```
18000/18000 [=====] - 137s 8ms/step - loss: 0.7442  
- acc: 0.7024 - val_loss: 1.9067 - val_acc: 0.4129
```

Advanced Training Options

Callbacks in Keras

A callback is a **set of functions** to be applied at given stages of the training procedure.

Callbacks give a **view on internal states** and statistics of the model **during training**.

You can pass a **list of callbacks** (as the keyword argument `callbacks`) to the `.fit()` method of the Sequential or Model classes.

The relevant methods of the callbacks will then be called at each stage of the training.

```
callback_list = [cb1,...,cbN]  
  
model.fit(X_train, y_train,  
validation_data=(X_test, y_test), epochs=3,  
callbacks = callback_list)
```

Model Checkpoint

Training a network might take up to several hours

Checkpoints are snapshots of the state of the system to be saved in case of system failure.

When training a deep learning model, the checkpoint is the weights of the model. These weights can be used to make predictions as is, or used as the basis for ongoing training.

```
from keras.callbacks import ModelCheckpoint  
[...]  
  
cp = ModelCheckpoint(filepath,  
monitor='val_loss', verbose=0,  
save_best_only=False, save_weights_only=False,  
mode='auto', period=1)
```

Early Stopping

The only stopping criteria when training a Deep Learning model is “reaching the required number of epochs.”

However, it might be enough to train a model further, as sometimes the training error decreases but the validation error does not (overfitting)

Checkpoints are used to stop training when a monitored quantity has stopped improving.

```
from keras.callbacks import EarlyStopping  
[...]  
  
es = EarlyStopping(monitor='val_loss',  
min_delta=0, patience=0, verbose=0, mode='auto',  
baseline=None, restore_best_weights=False)
```

Testing the model

Predict() method

```
#returns the class probabilities for the input  
image x_test  
score = model.predict(x_test)  
# select the class with the largest score  
prediction_test = np.argmax(score, axis=1)
```

Tensorboard

When training a model it is important to monitor its progresses

Google has developed tensorboard a very useful tool for visualizing reports.

```
from keras.callbacks import TensorBoard  
[...]  
tb = TensorBoard(log_dir="dirname")
```

... and add tb to the checkpoint list as well