

Convolutional Neural Networks

Giacomo Boracchi,
DEIB, Politecnico di Milano
October, 10th, 2022

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

Errata Corrige on Logistics

Task 1 (deadline Friday 27 October 23:59). Create the teams and fill in the form:

https://docs.google.com/forms/d/e/1FAIpQLScpcA5A5kfayZSbUZ3wqIw1lWS2nVRe0Ts_xu5HNcX03VlKHA/viewform?usp=sf_link

Task 2 Everyone start navigating the **CodaLab** platform to become familiar with it. Please have a look at the CodaLab documentation
<https://github.com/codalab/codalab-competitions/wiki#1-participants>.

Students with already complete teams: follow the Profile teams instructions (https://github.com/codalab/codalab-competitions/wiki/User_Teams#profile-teams) and register your team name on your CodaLab settings.

Students without a team and students with incomplete teams: wait for our communication for team assignments. Then register the team.

All the teams made by students without a team and students with incomplete teams will be finalized and communicated before.

Errata Corrige on Logistics

Deadlines:

Friday 27 October NO ANSWERS TO THE FORM WILL BE ACCEPTED AFTER THE DEADLINE

Wednesday 1 November. All the teams made by students without a team and students with incomplete teams will be finalized and communicated before **Wednesday 1 November.**

The challenge will start around 2 November

The Feature Extraction Perspective

The Feature Extraction Perspective

Images can not be directly fed to a classifier

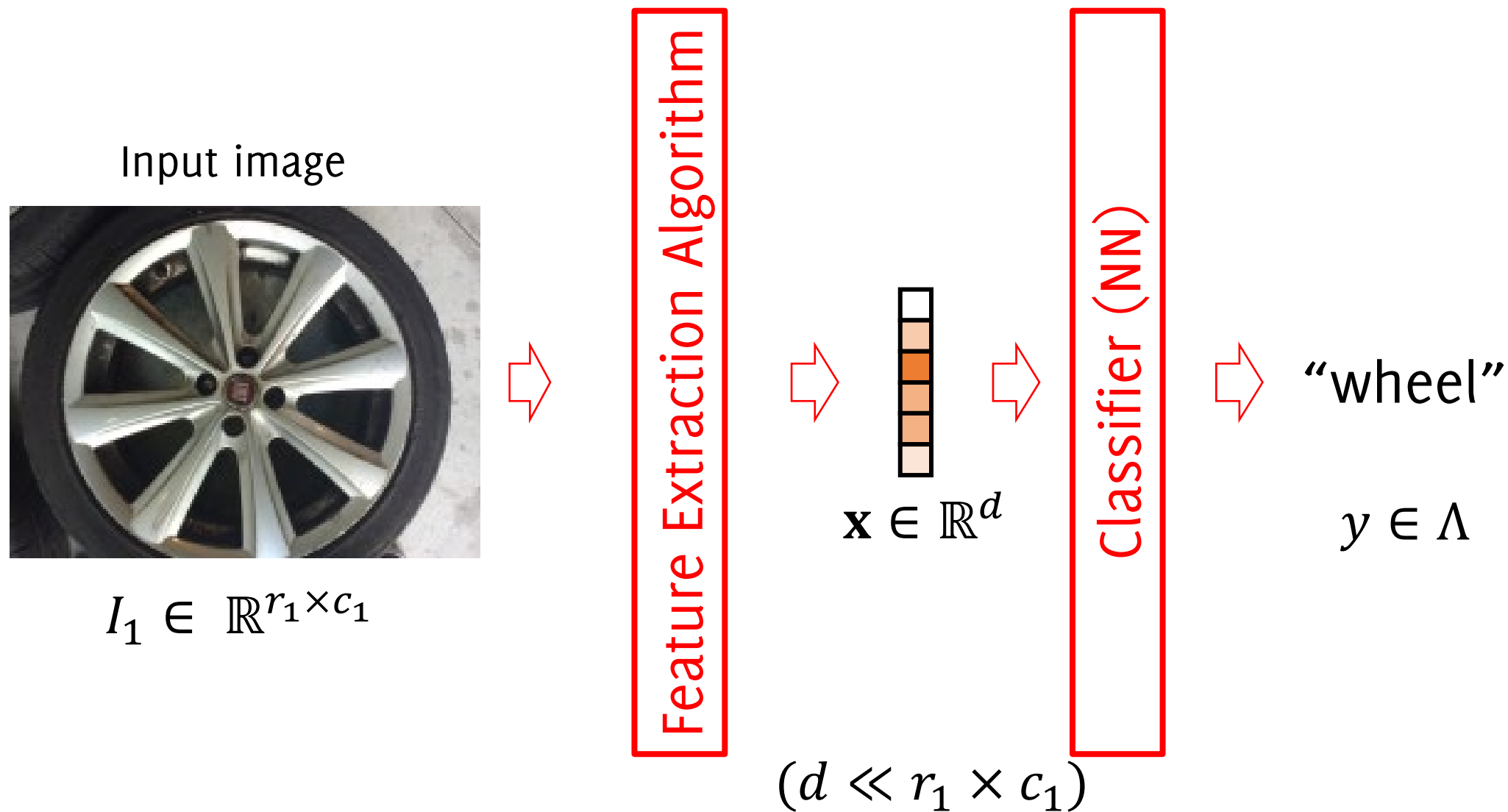
We need some intermediate step to:

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

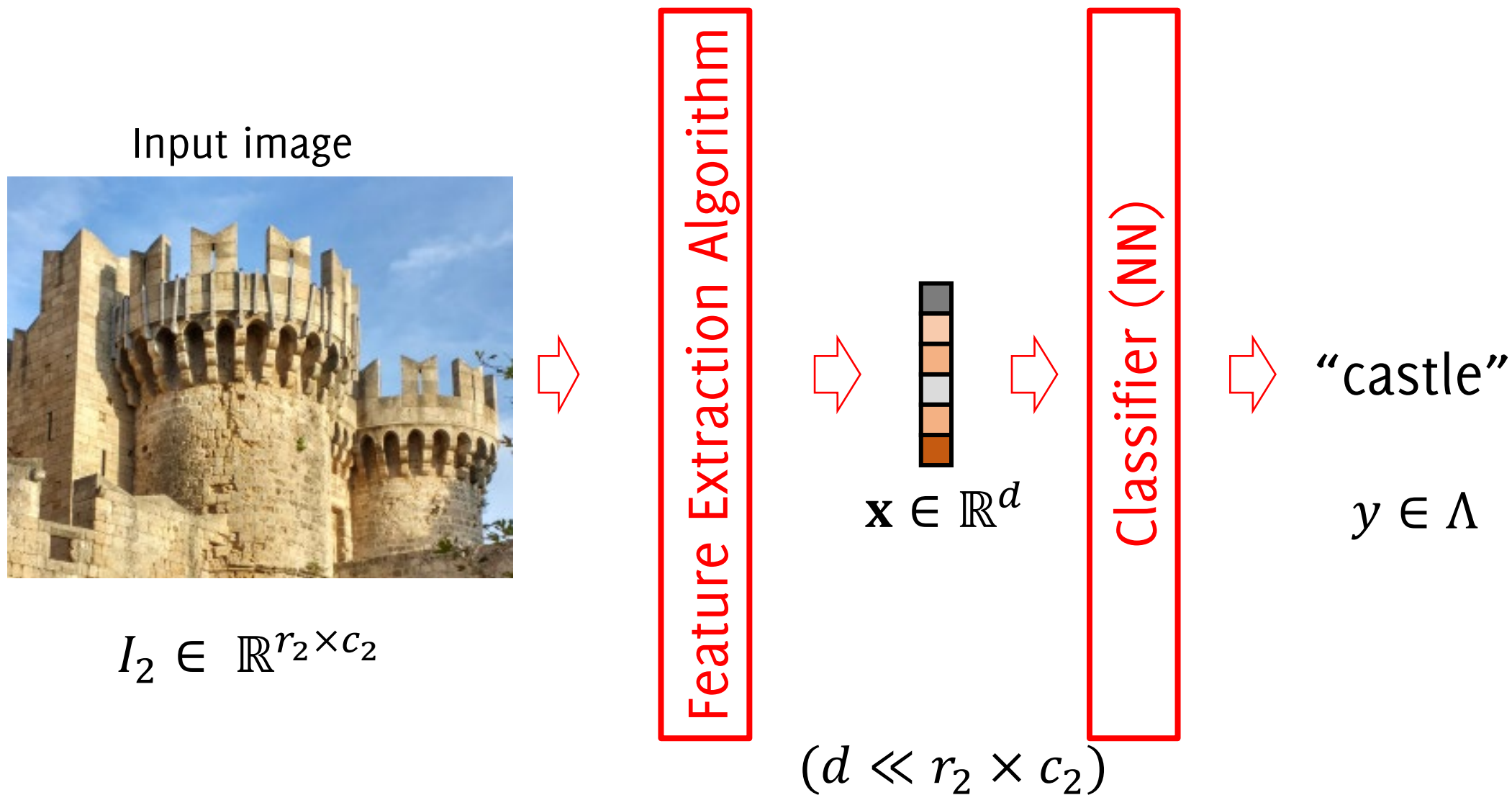
We need to extract features:

- The better our features, the better the classifier

The Feature Extraction Perspective



The Feature Extraction Perspective

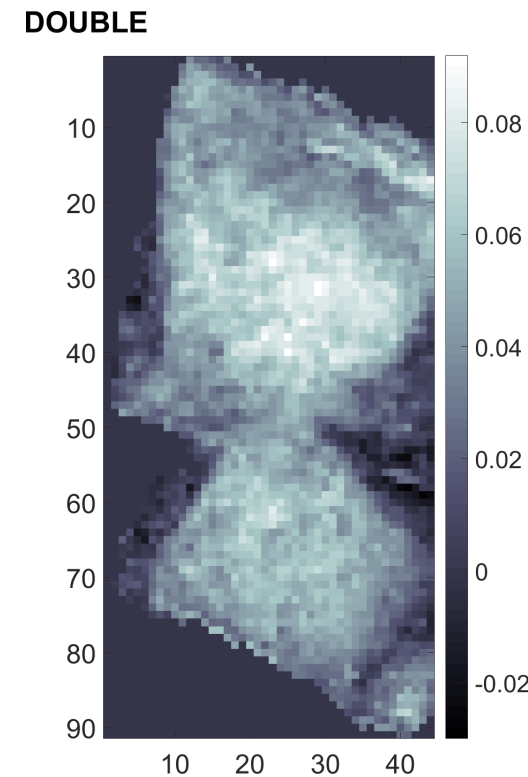
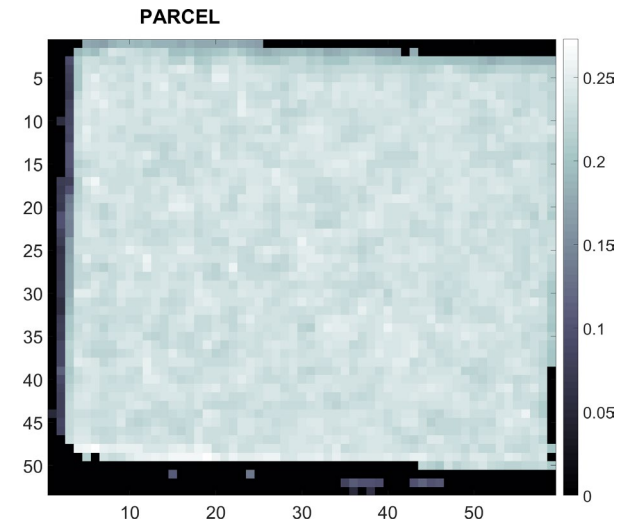
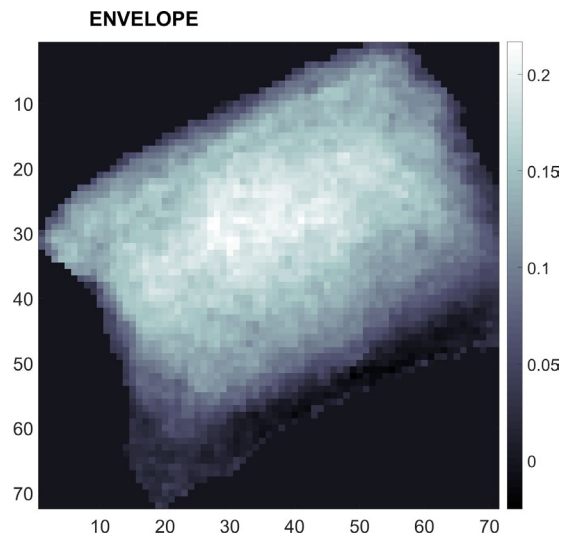


Hand-Crafted Features

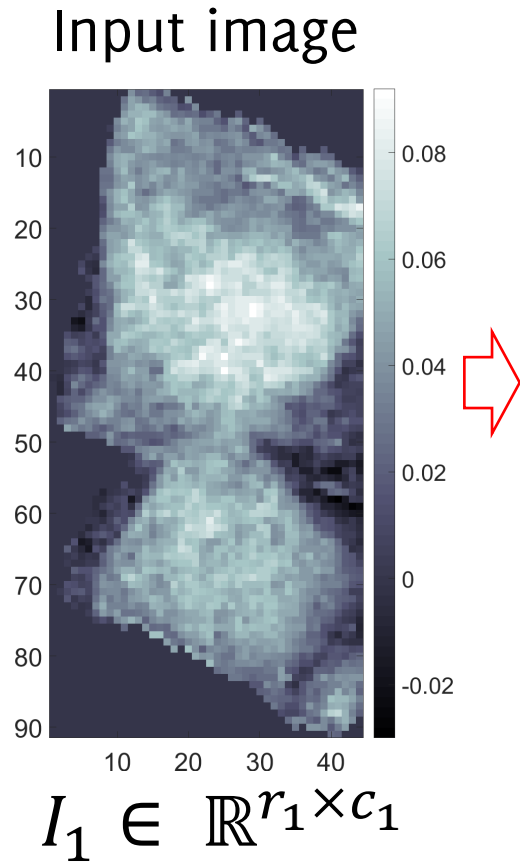
Example of Hand-Crafted Features

Example of features:

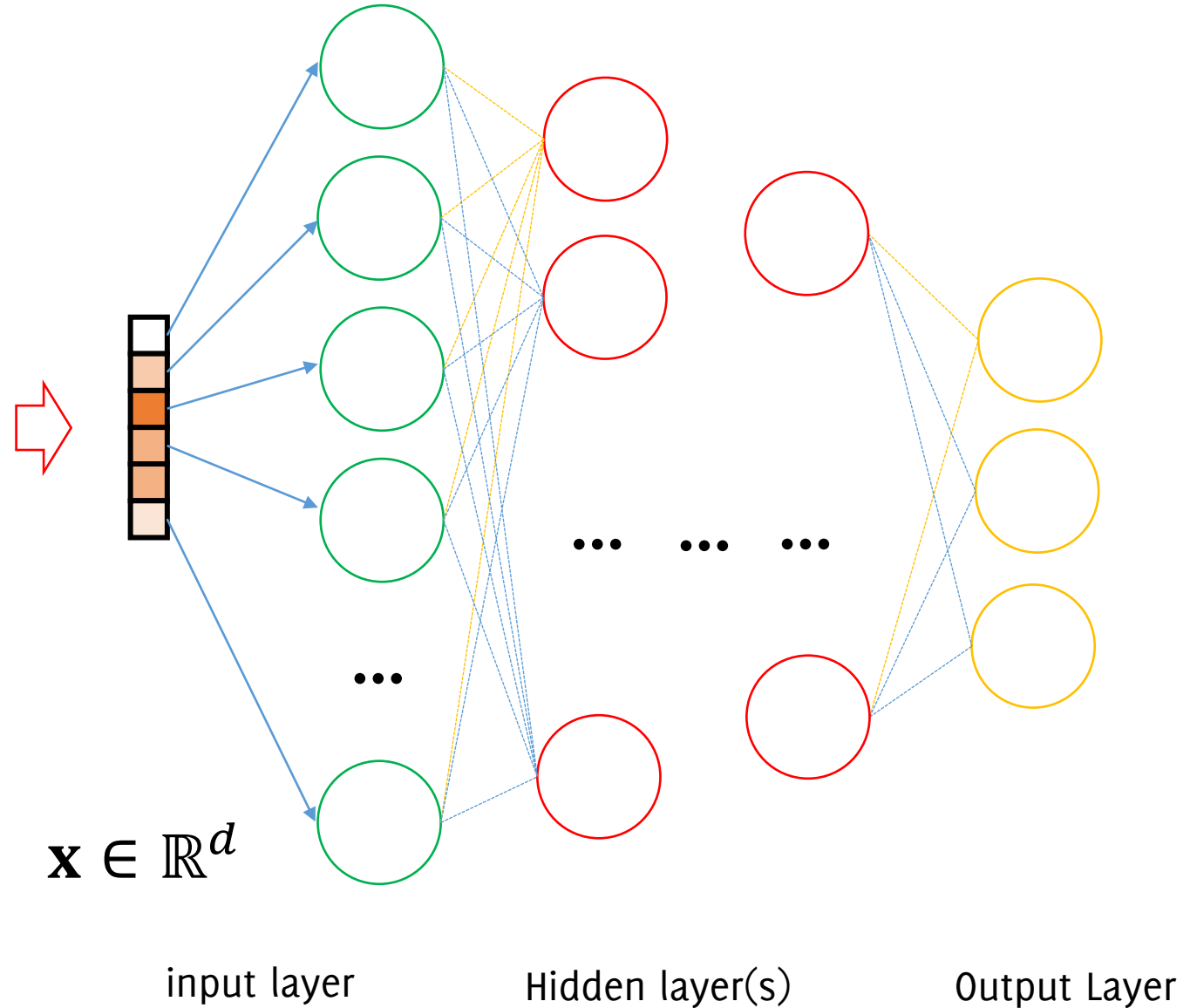
- Average height
- Area (coverage with nonzero measurements)
- Distribution of heights
- Perimeter
- Diagonals



Neural Networks



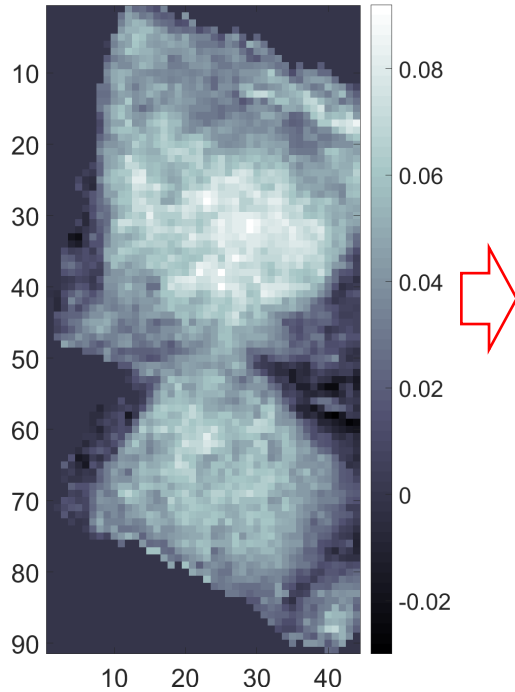
Feature Extraction Algorithm



Neural Networks

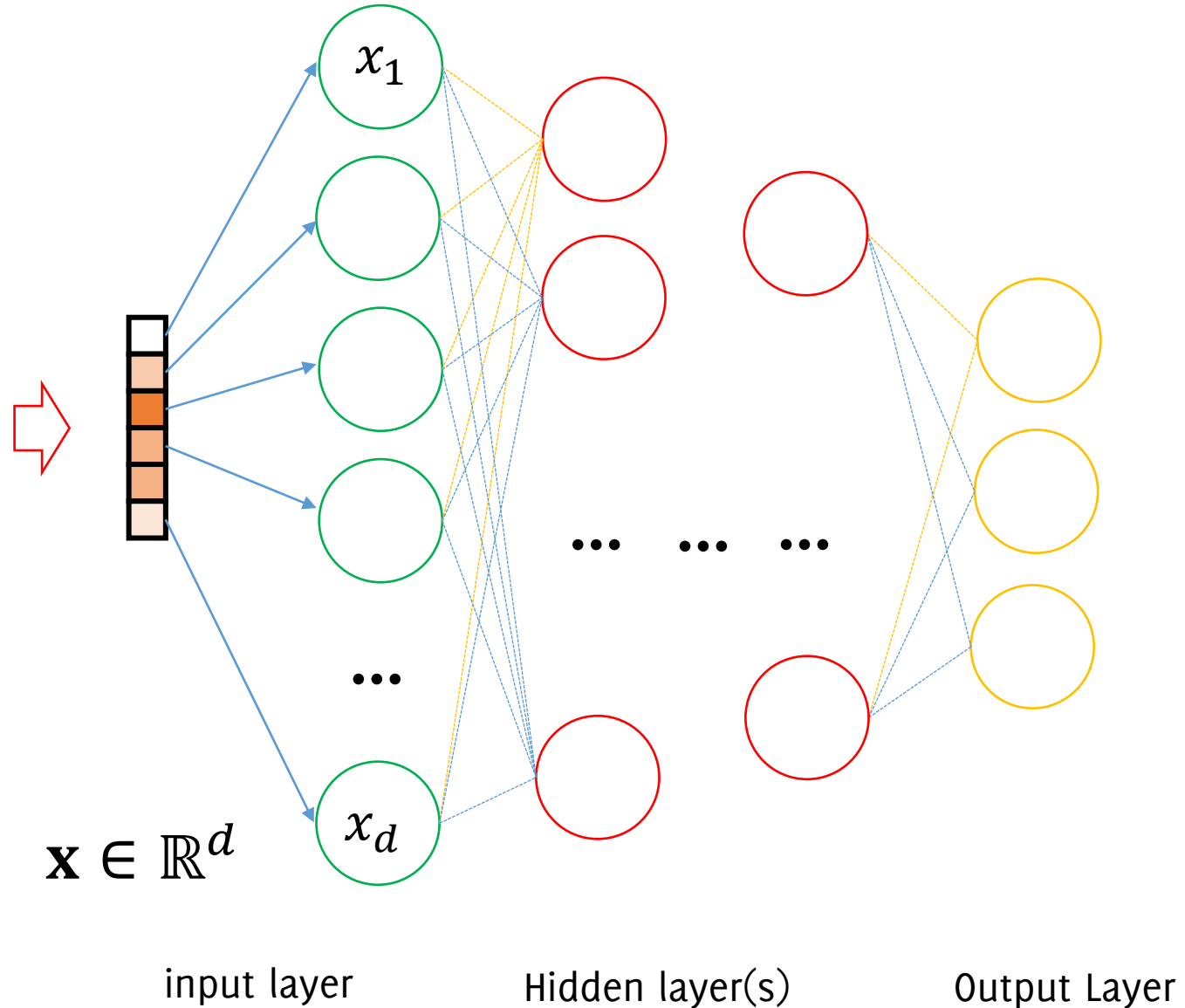
Input layer: Same size of the feature vector

Input image

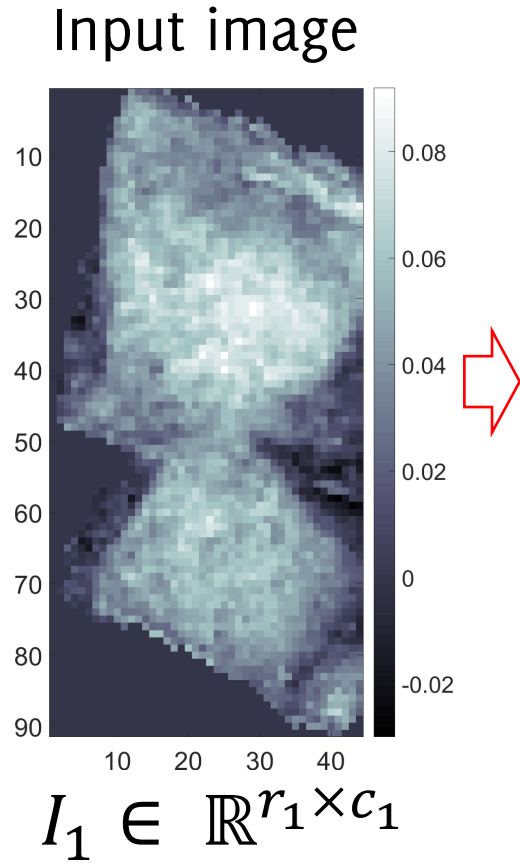


$$I_1 \in \mathbb{R}^{r_1 \times c_1}$$

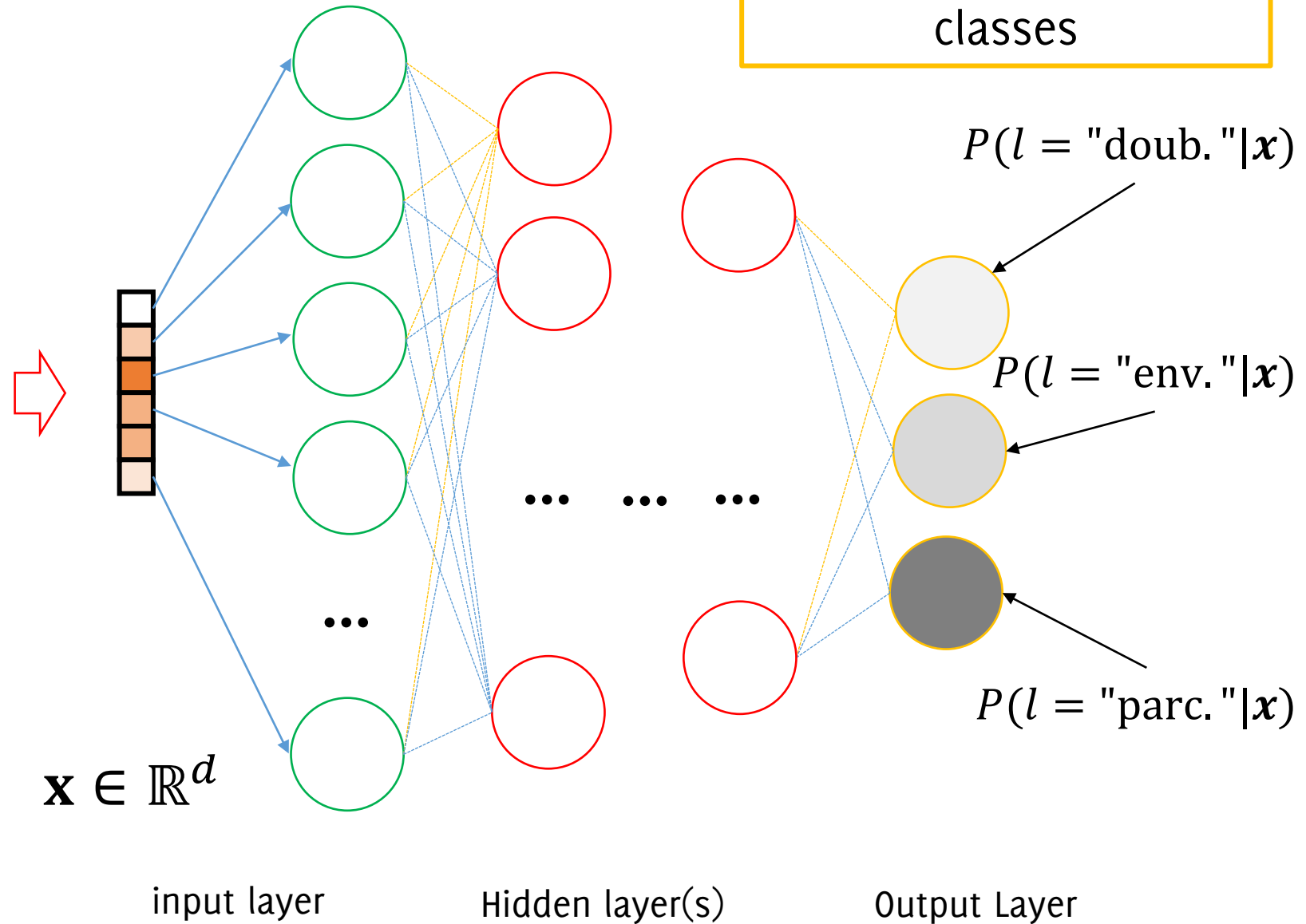
Feature Extraction Algorithm



Neural Networks



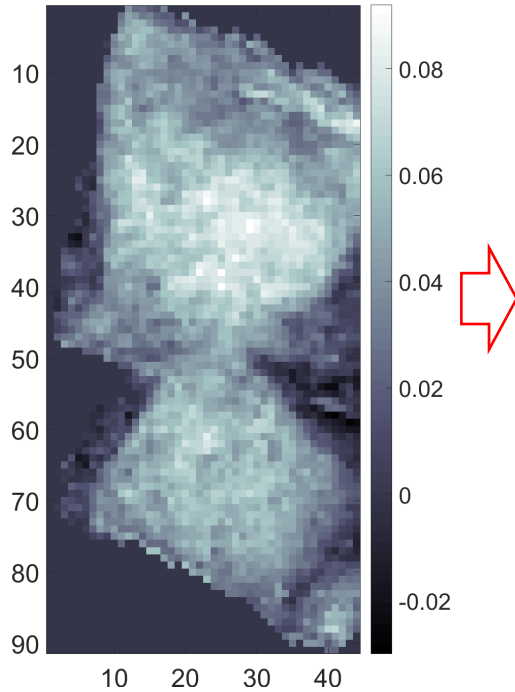
Feature Extraction Algorithm



Neural Networks

Hidden layers: arbitrary size

Input image



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

Feature Extraction Algorithm

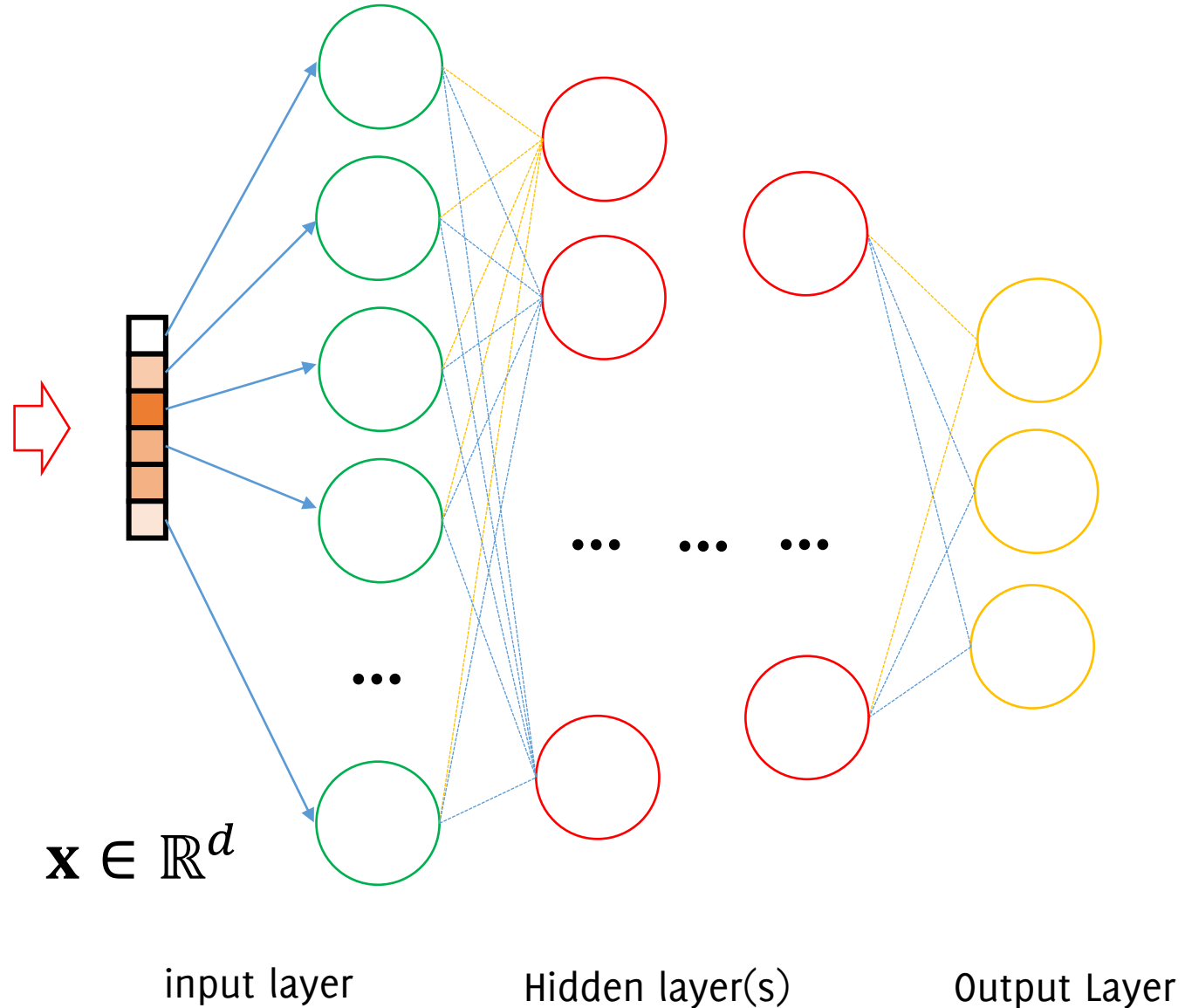
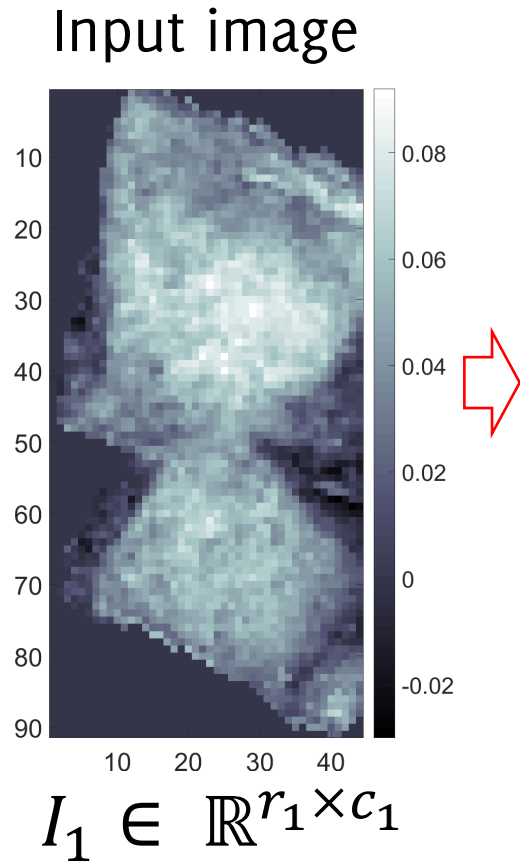
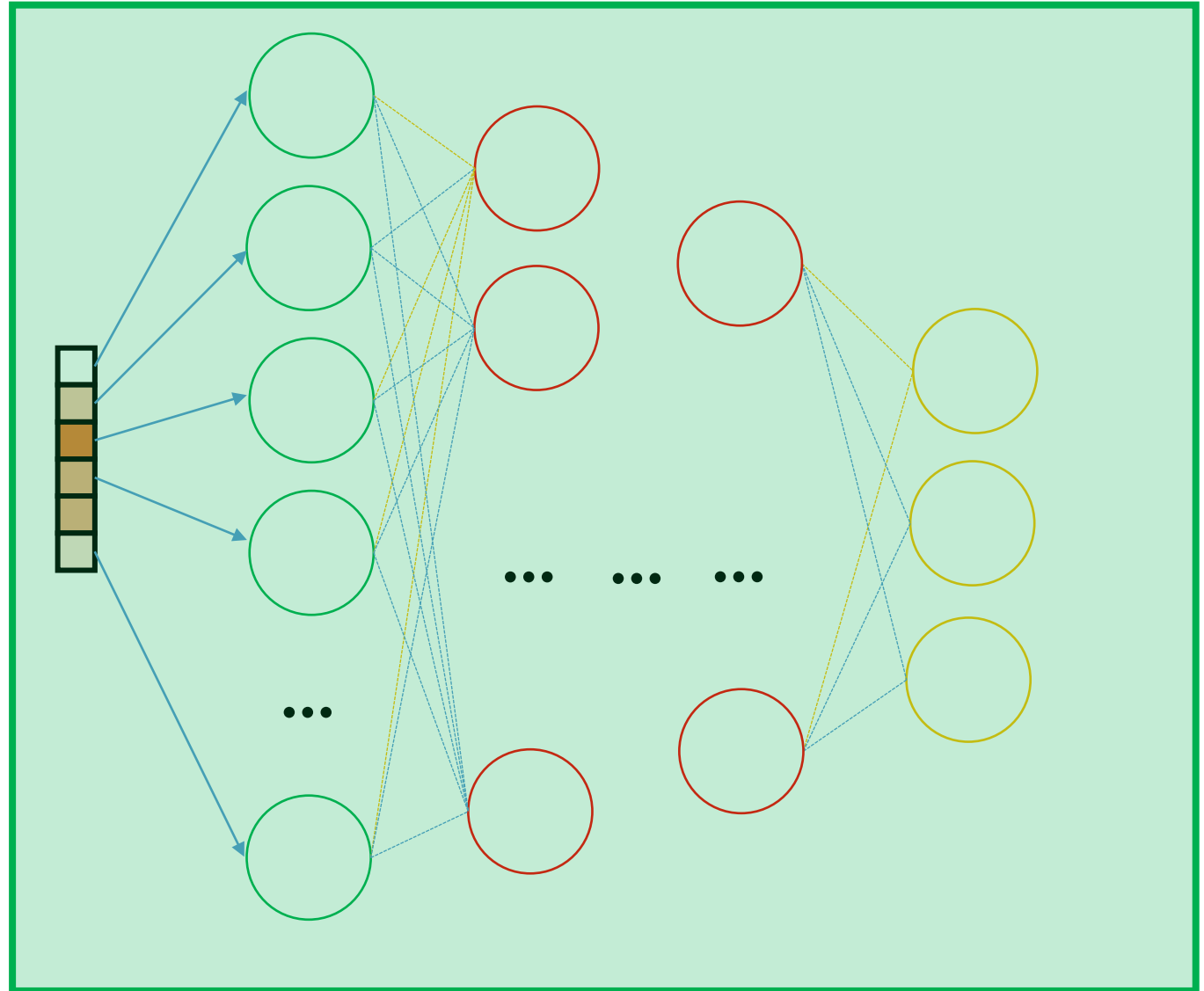


Image Classification by Hand Crafted Features



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

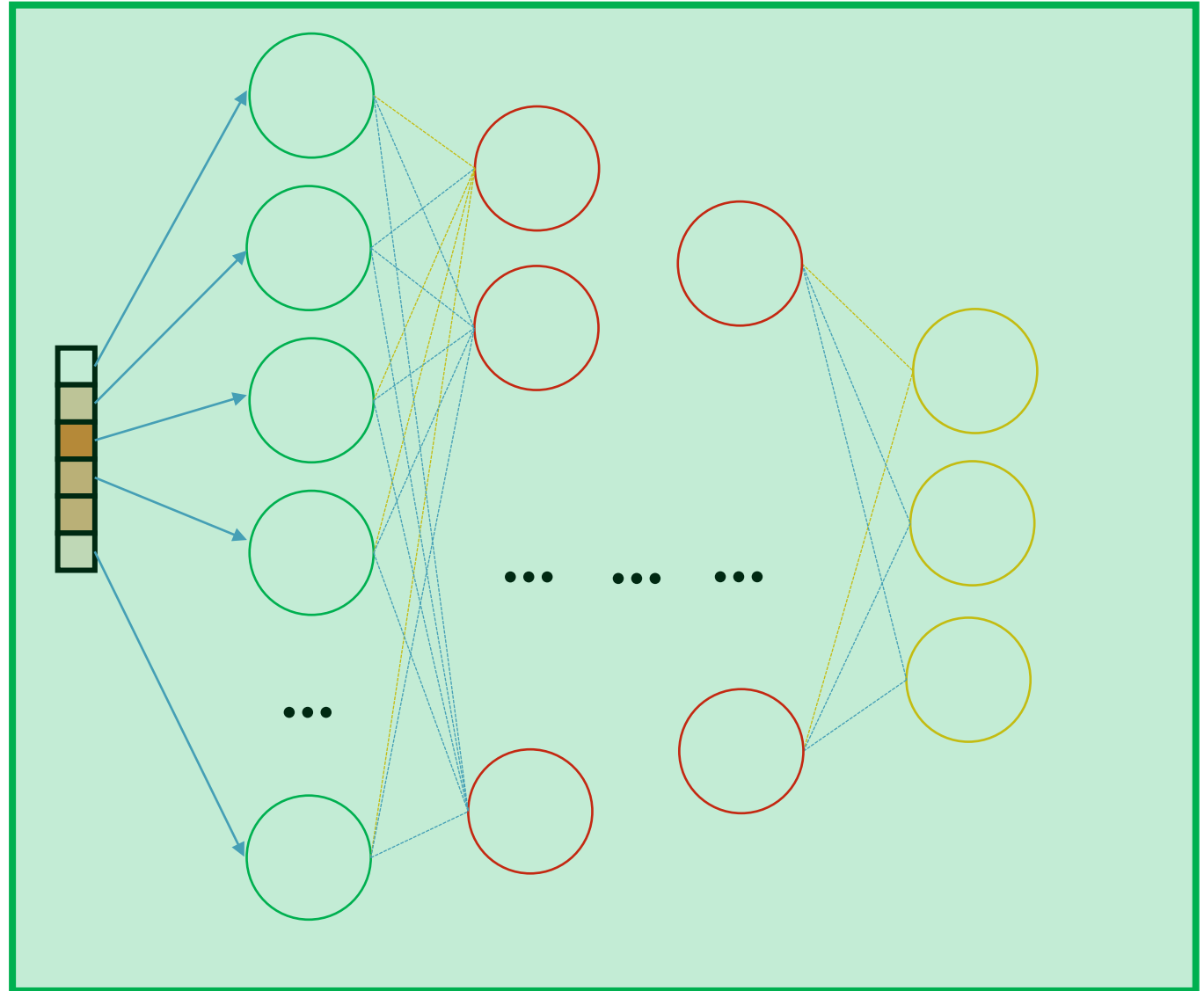
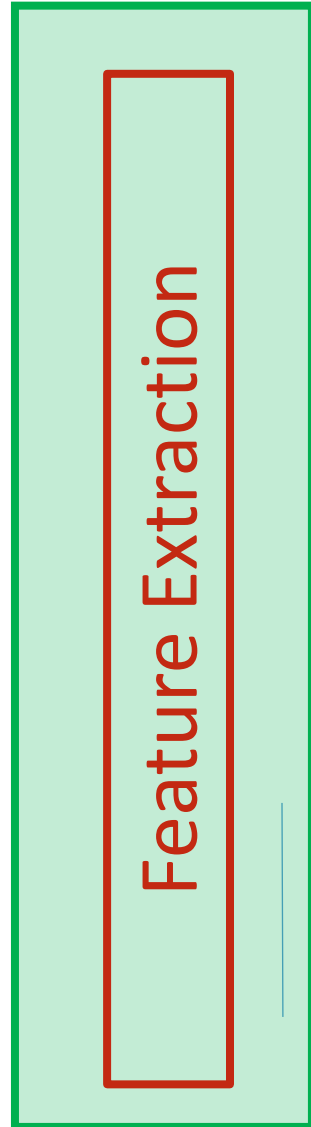
... the advent of deep learning

Data-Driven Features

Input image



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



Data Driven

Data Driven

Convolutional Neural Networks

Setting up the stage

Local Linear Filters

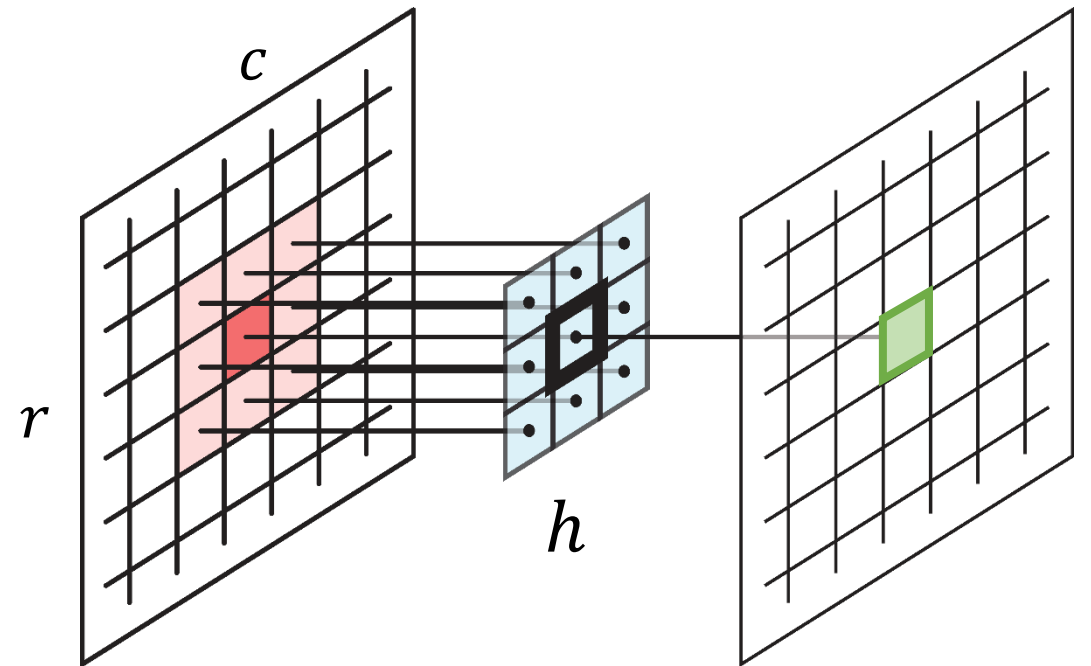
Linear Transformation: Linearity implies that the **output** $T[I](r, c)$ is a linear combination of the pixels in U :

$$T[I](r, c) = \sum_{(u, v) \in U} w_i(u, v) * I(r + u, c + v)$$

Considering *some weights* $\{w_i\}$

We can consider weights as an image, or a **filter** h

The filter h entirely defines this operation



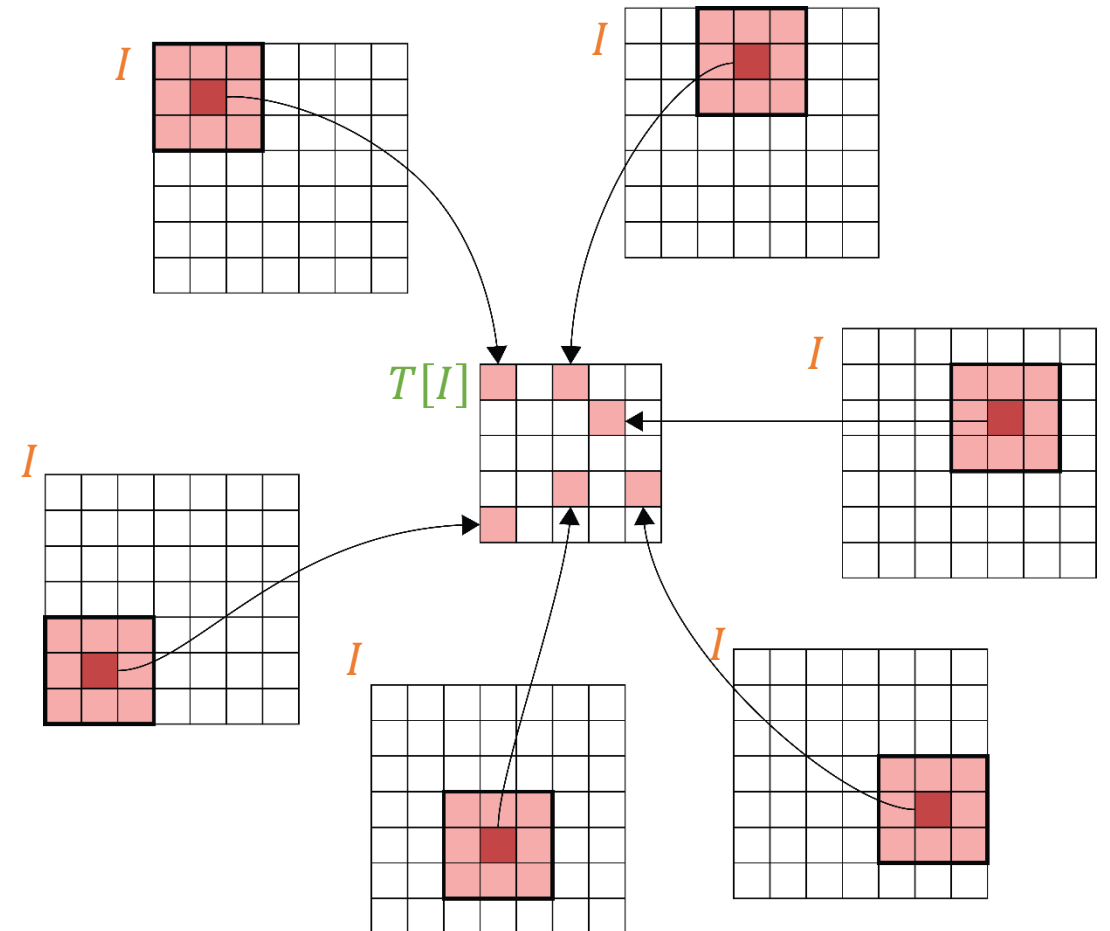
Local Linear Filters

Linear Transformation: the filter weights can be associated to a matrix \mathbf{w}

$$T[I](r, c) = \sum_{(u,v) \in U} w_i(u, v) * I(r + u, c + v)$$

\mathbf{w}		
$w(-1, -1)$	$w(-1, 0)$	$w(-1, 1)$
$w(0, -1)$	$w(0, 0)$	$w(0, 1)$
$w(1, -1)$	$w(1, 0)$	$w(1, 1)$

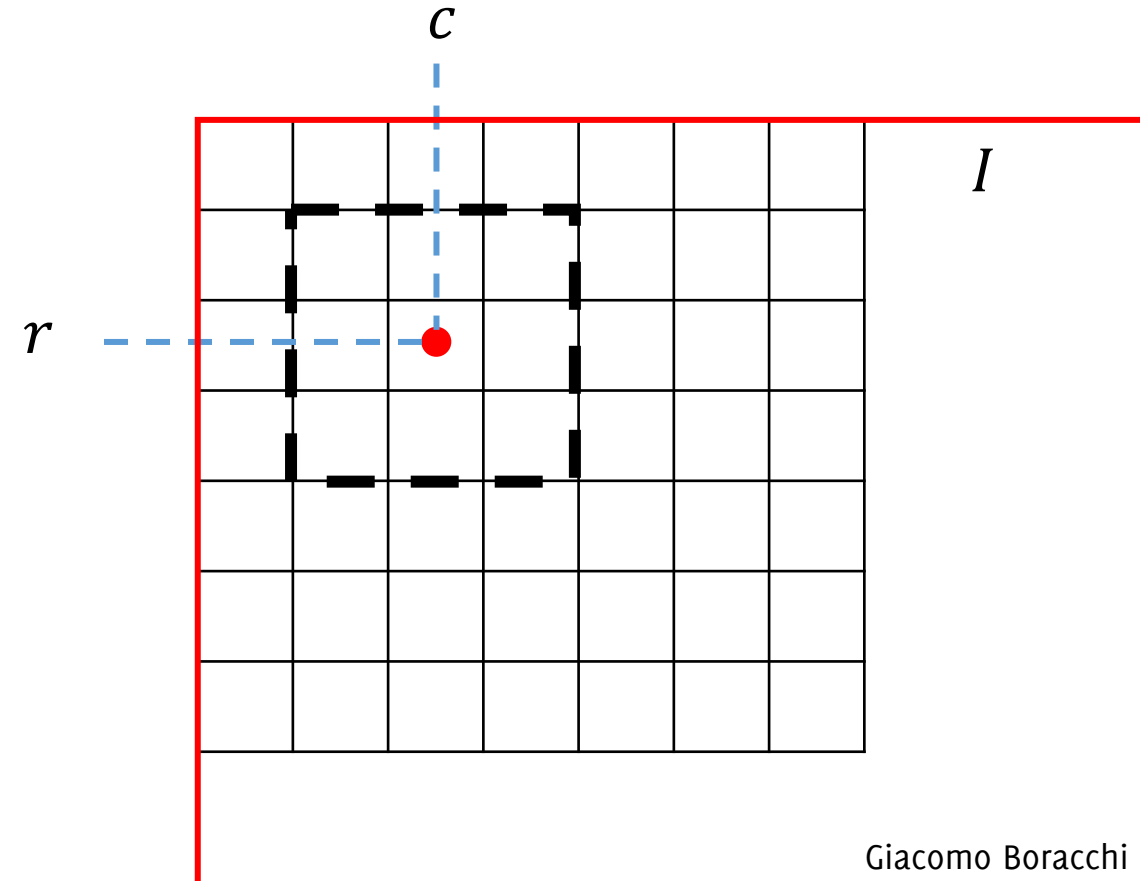
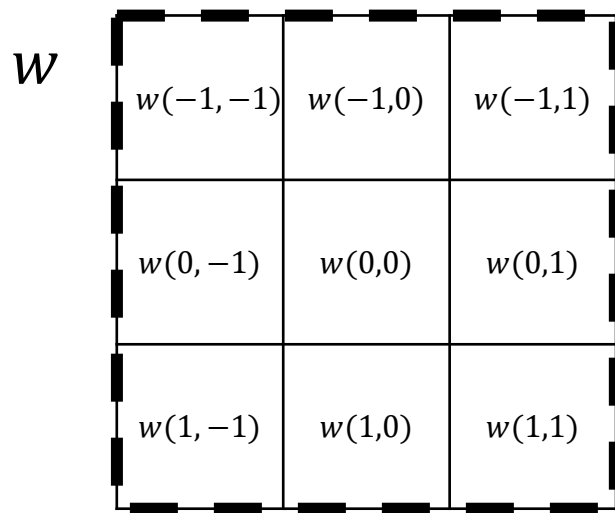
This operation is repeated for each pixel in the input image



2D Correlation

Convolution is a linear transformation. Linearity implies that

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

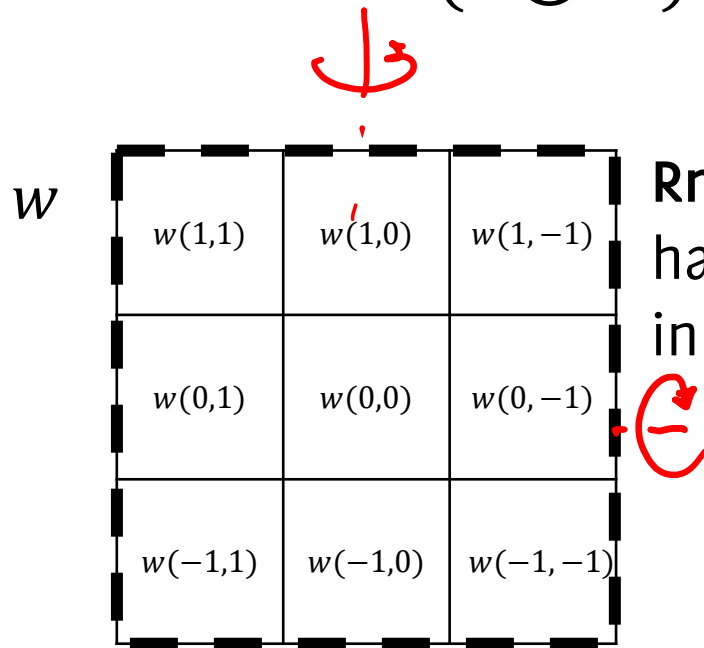


We can consider weights as a filter h
The filter h entirely defines convolution
Convolution operates the same in each pixel

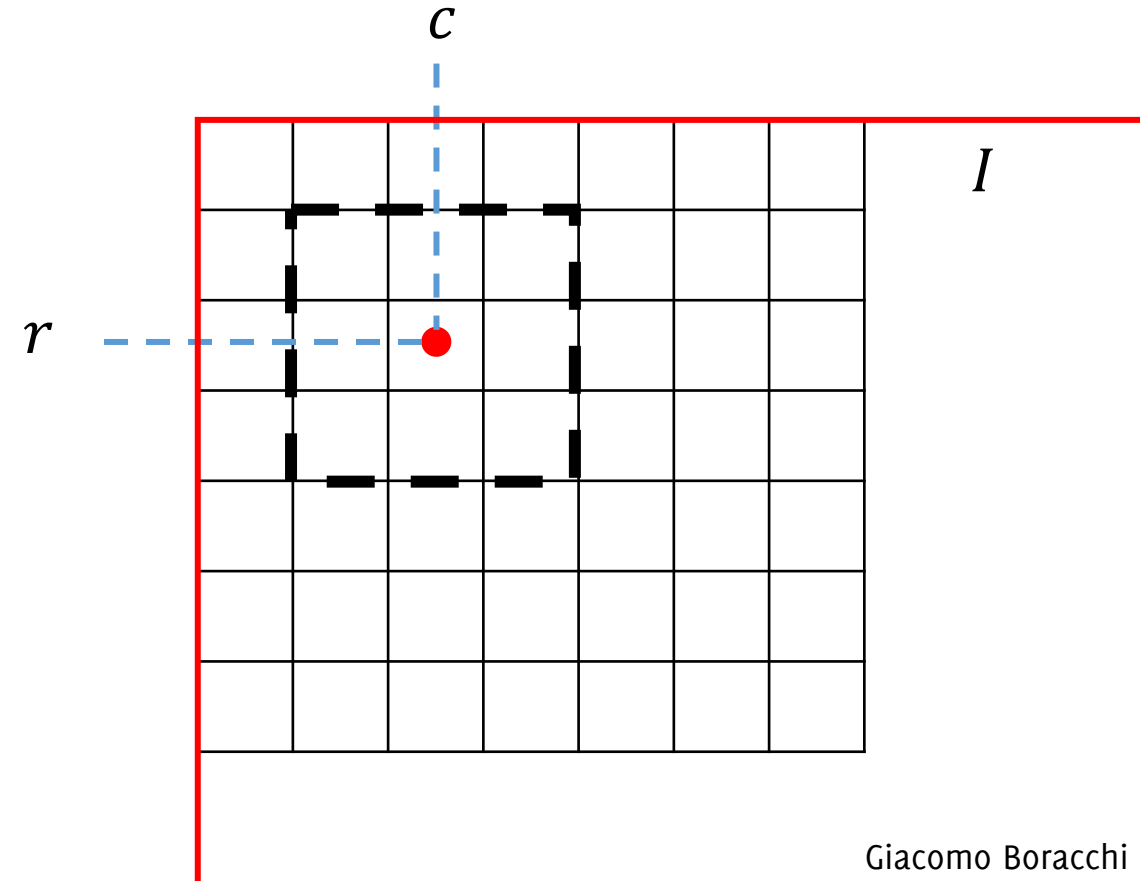
2D Convolution

Convolution is a linear transformation. Linearity implies that

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



Rmk: indexes have been shifted in the filter w

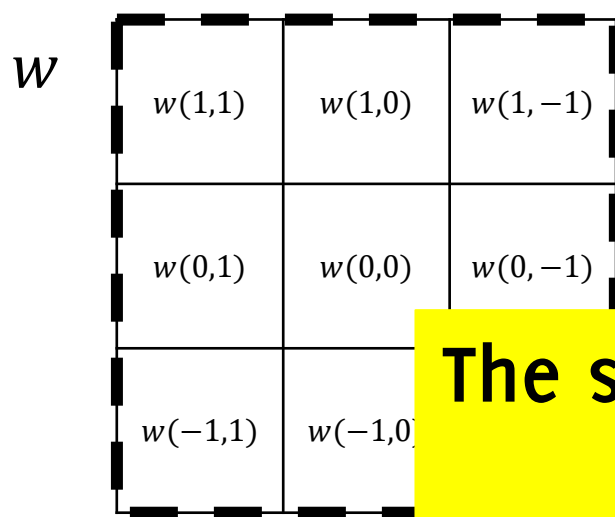


We can consider weights as a filter h
 The filter h entirely defines convolution
 Convolution operates the same in each pixel

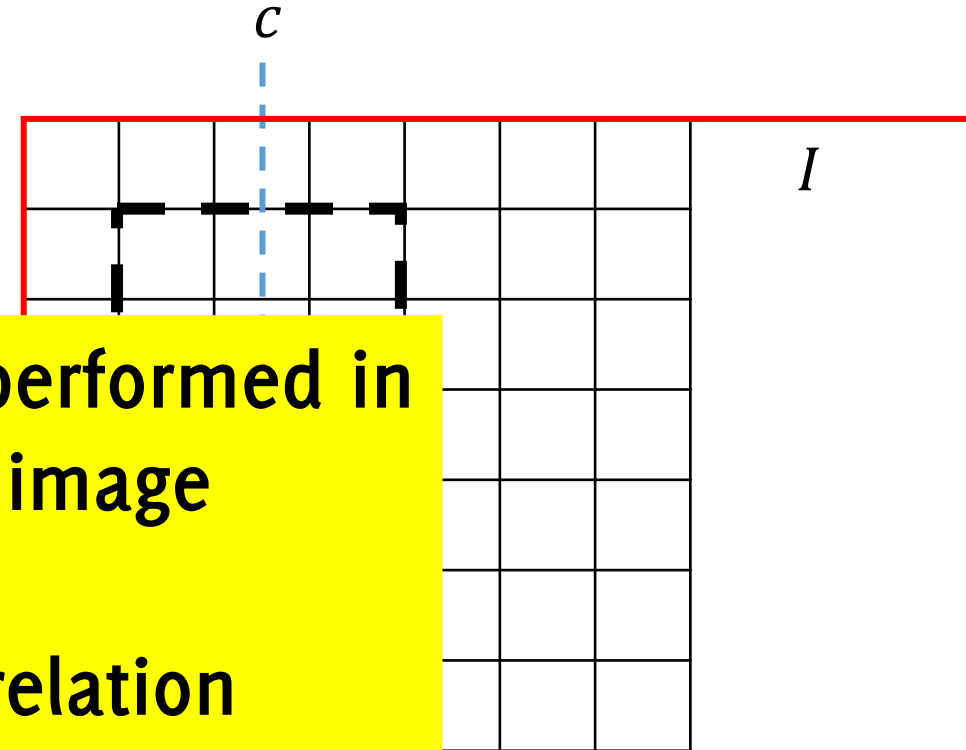
2D Convolution

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



**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**

We can con
The filter h e
Convolution op

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 must be considered 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!

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

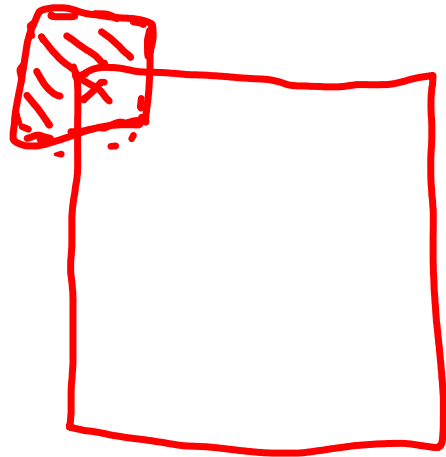
Input Volume (+pad 1) (7x7x3)							Filter W0 (3x3x3)		
$x[:, :, 0]$							$w0[:, :, 0]$		
0	0	0	0	0	0	0	0	1	-1
0	1	0	2	1	0	0	-1	-1	0
0	1	1	1	0	1	0	1	-1	-1
0	1	2	1	0	1	0			
0	1	2	0	2	2	0			
0	1	0	1	0	0	0			
0	0	0	0	0	0	0			

Original image is in violet, grey values are padded to zero to enable convolution at image boundaries

Convolution: Padding

How to define convolution output close to image boundaries?

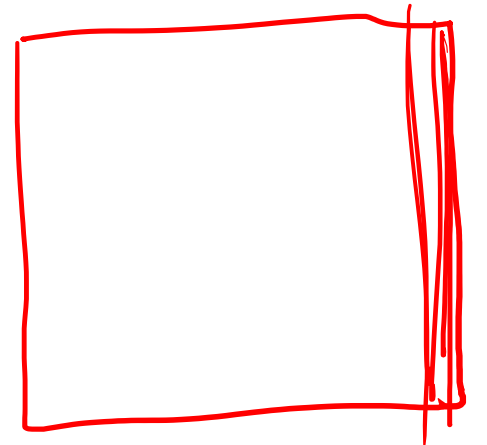
Padding with zero is the most frequent option, as this does not change the output size. However, no padding or symmetric padding are also viable options



Input Volume (+pad 1) (7x7x3)							Filter W0 (3x3x3)		
$x[:, :, 0]$							$w0[:, :, 0]$		
0	0	0	0	0	0	0	0	1	-1
0	1	0	2	1	0	0	-1	-1	0
0	1	1	1	0	1	0	1	-1	-1
0	1	2	1	0	1	0			
0	1	2	0	2	2	0			
0	1	0	1	0	0	0			
0	0	0	0	0	0	0			

Original image is in violet, grey values are padded to zero to enable convolution at image boundaries

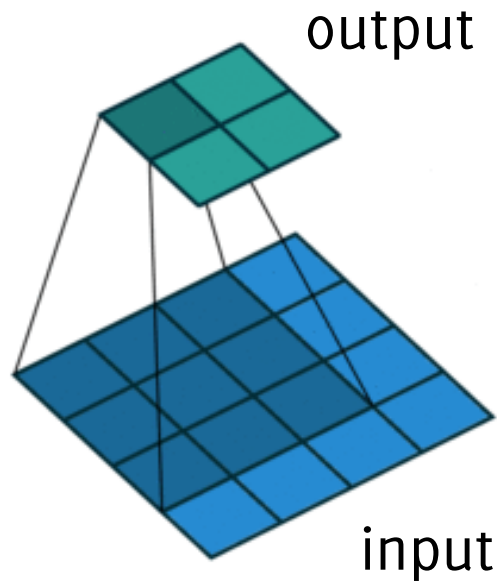
Conv2(I, f, 'same')



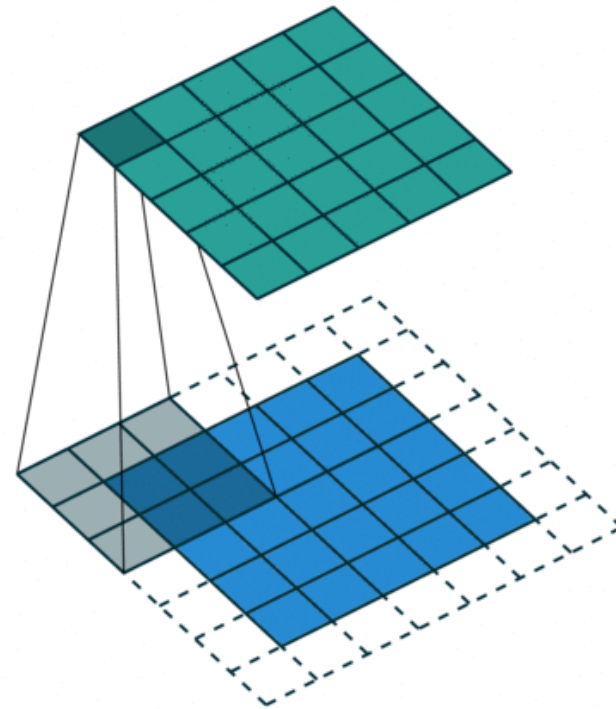
Padding Options in Convolution Animation

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

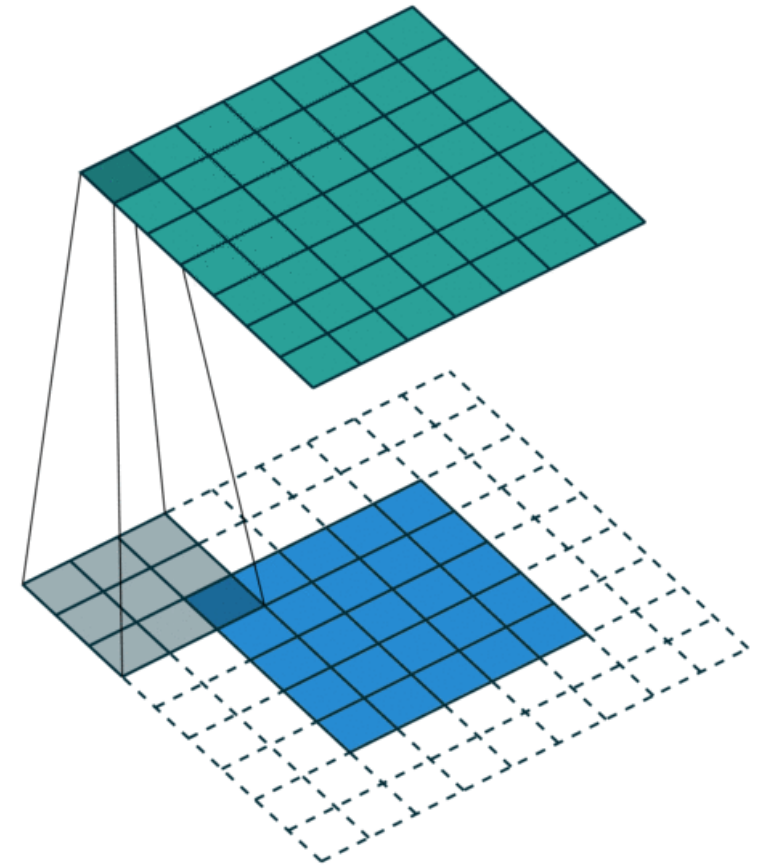
Rmk: the filter here is 3×3



No padding
«valid»



Half padding
«same»

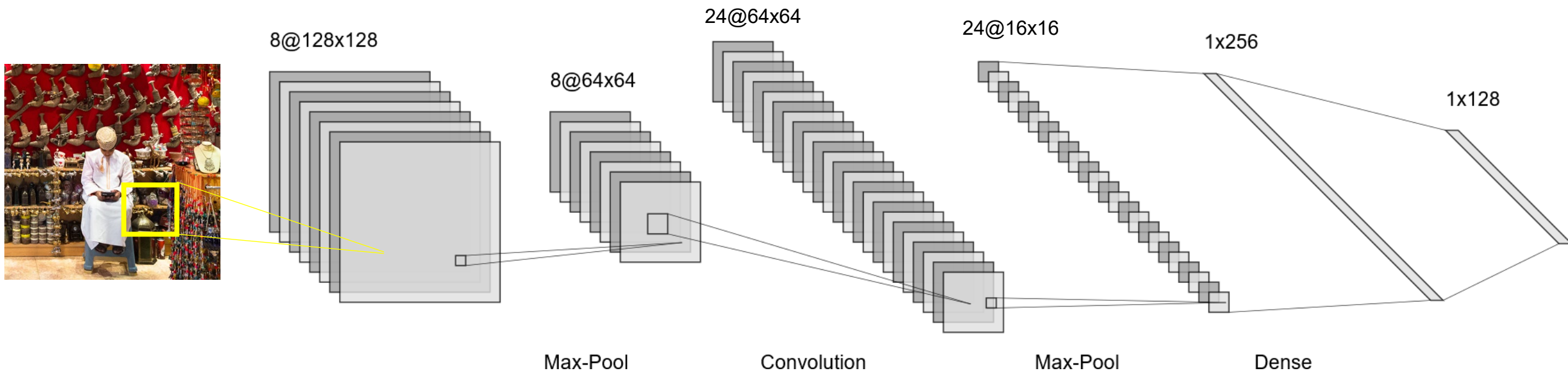


full padding
«full»

Convolutional Neural Networks

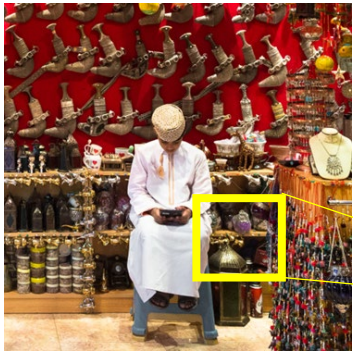
CNNs

The typical architecture of a CNN

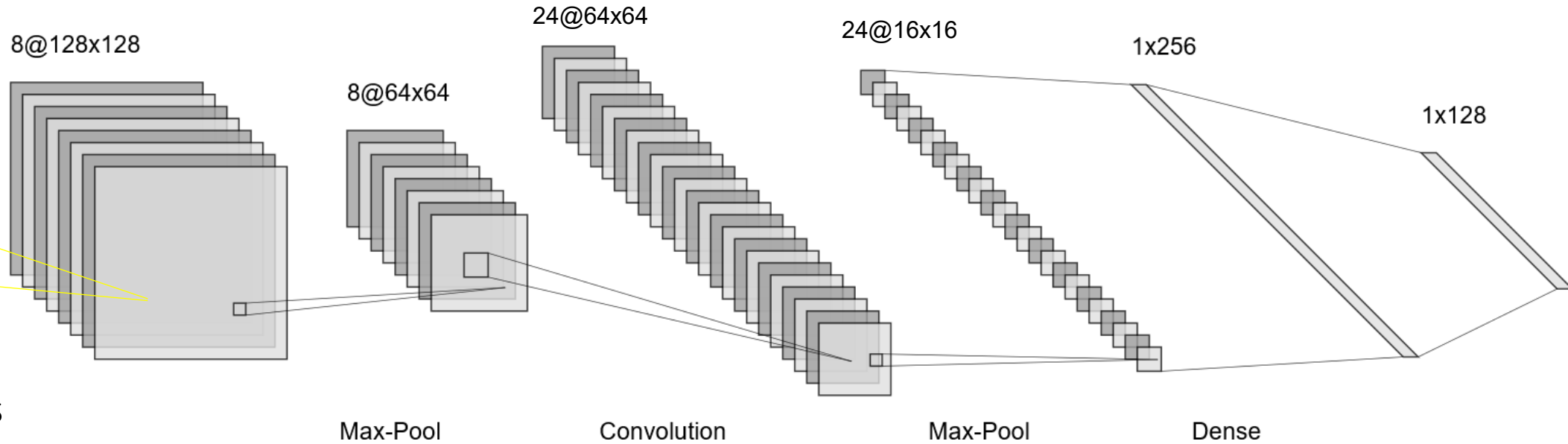


The typical architecture of a CNN

The input of a CNN
is an entire image

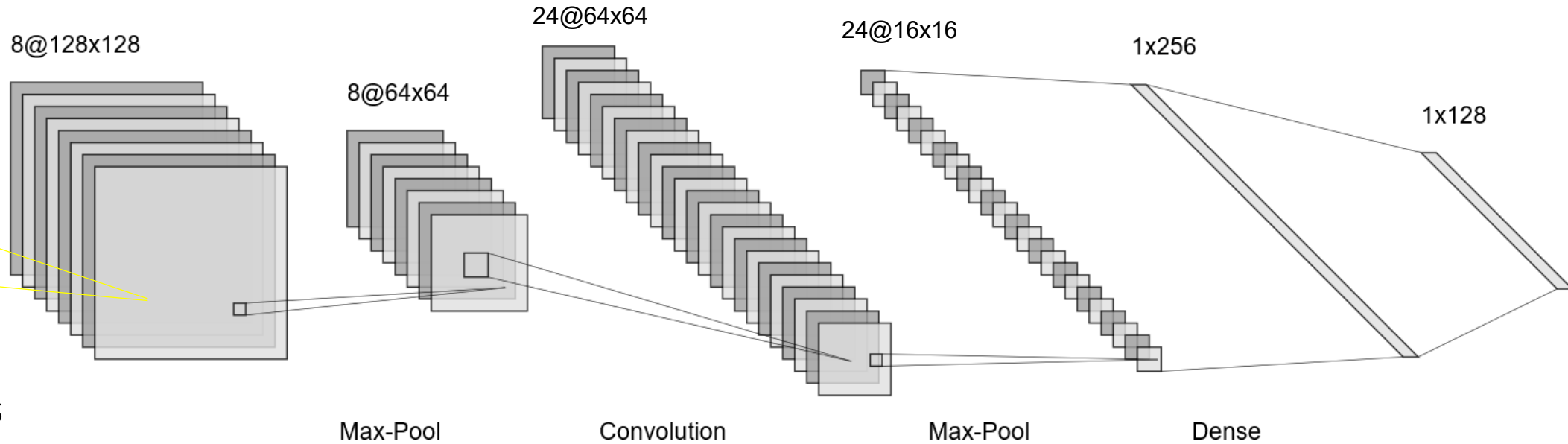
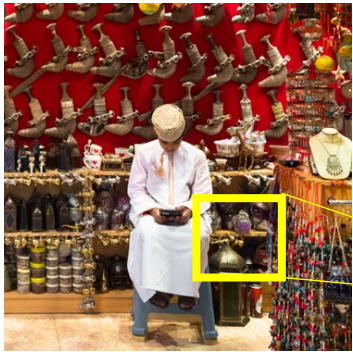


The image gets
convolved against
many filters



The typical architecture of a CNN

The input of a CNN
is an entire image

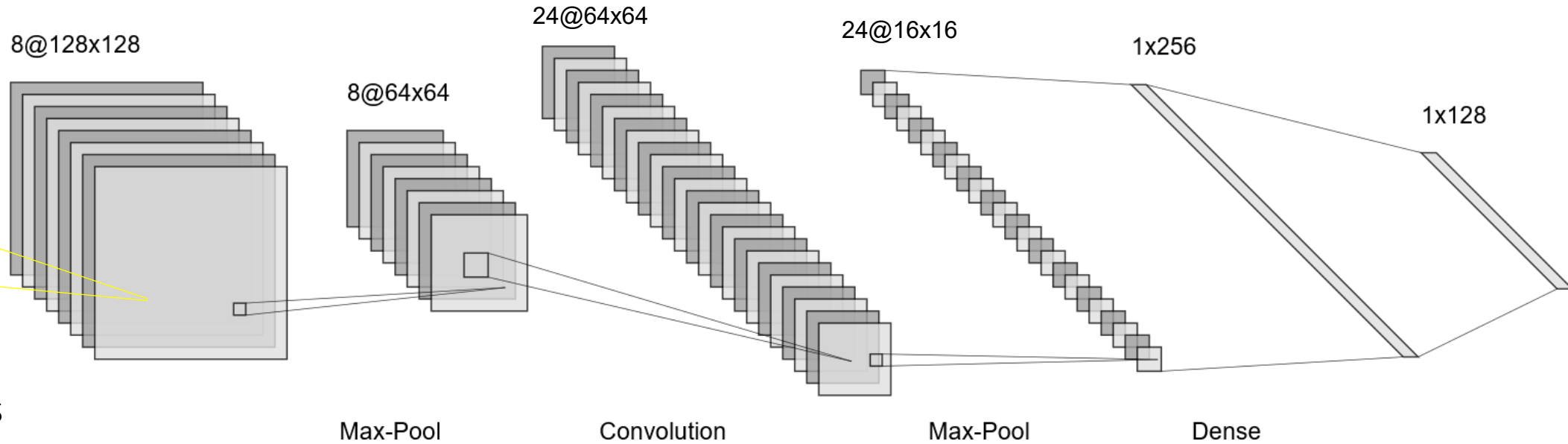
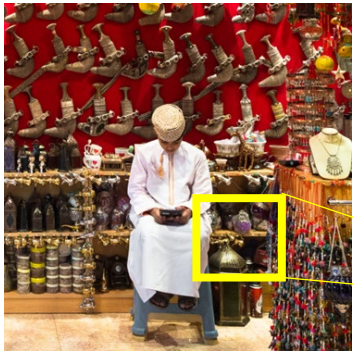


The image gets
convolved against
many filters

When progressing along the network, the
«number of images» or the «number of
channels in the images» increases, while
the image size decreases

The typical architecture of a CNN

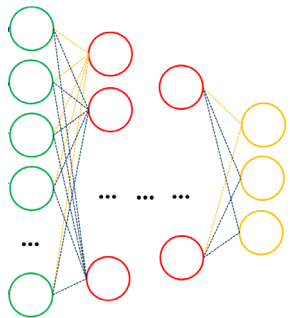
The input of a CNN is an entire image



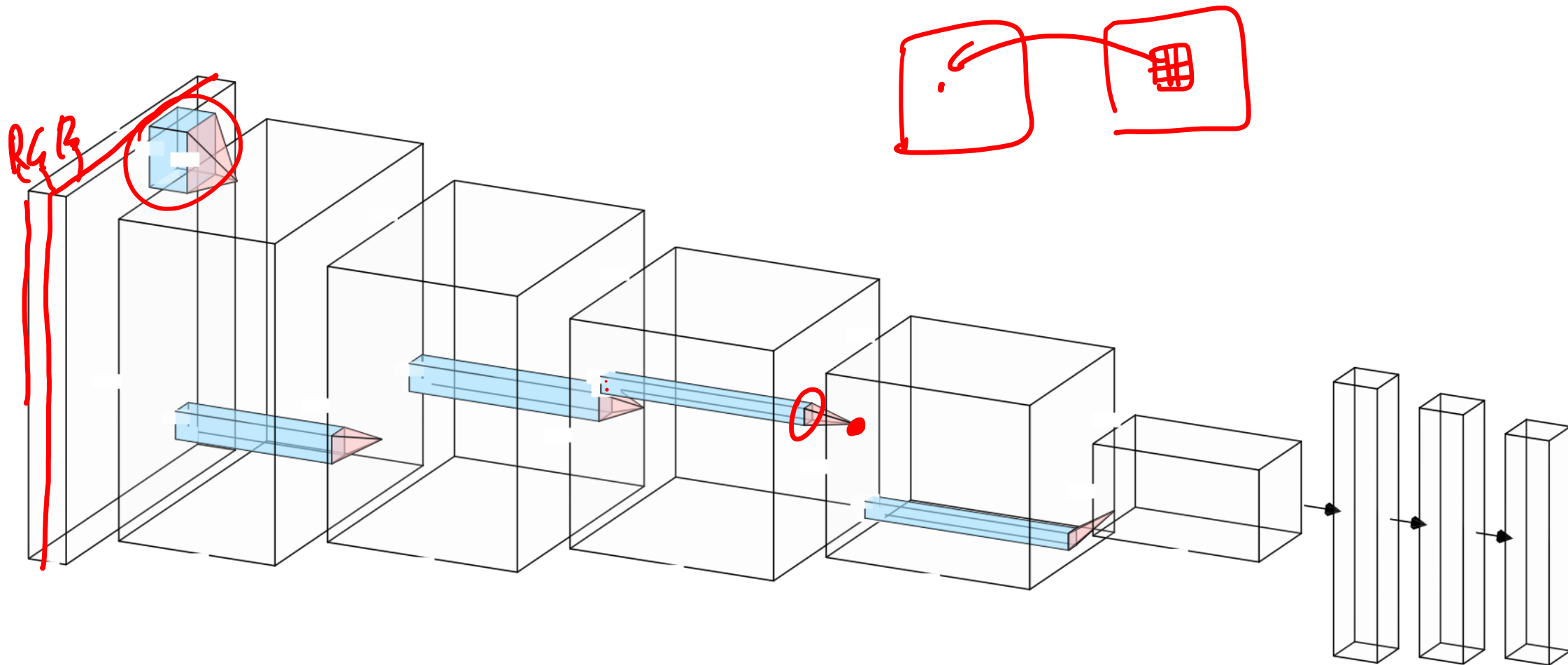
The image gets convolved against many filters

When progressing along the network, the «number of images» or the «number of channels in the images» increases, while the image size decreases

Once the image gets to a vector, this is fed to a traditional neural network



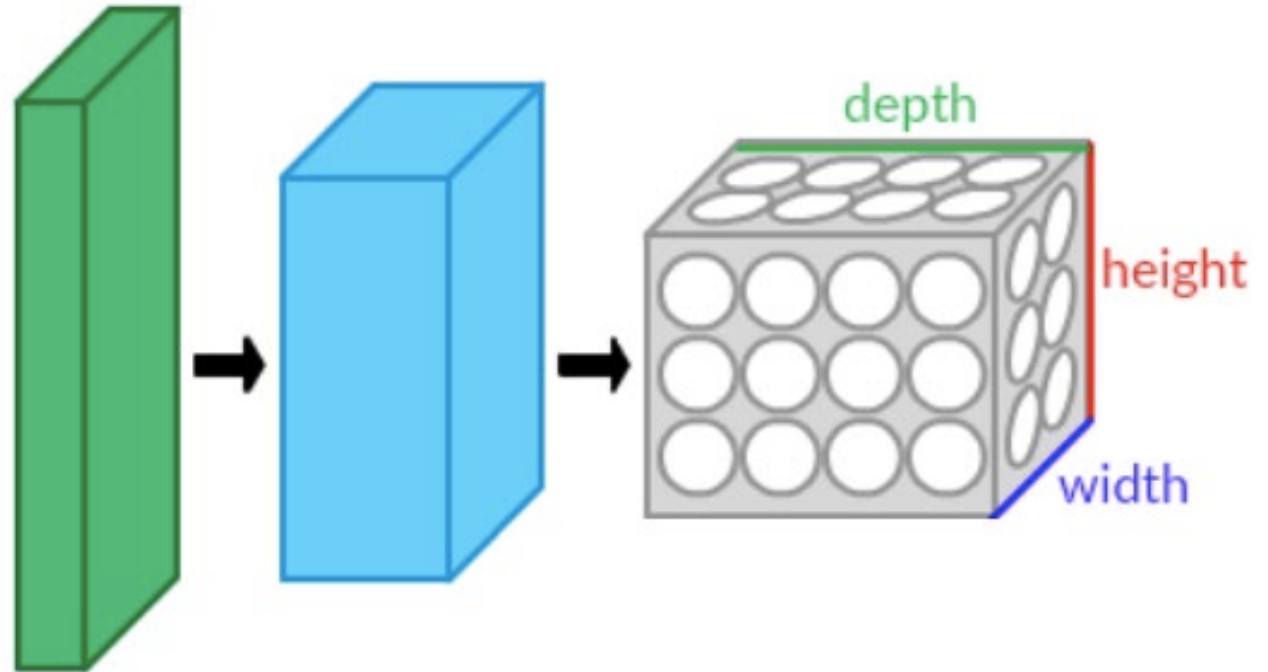
The typical architecture of a CNN



Convolutional Neural Networks (CNN)

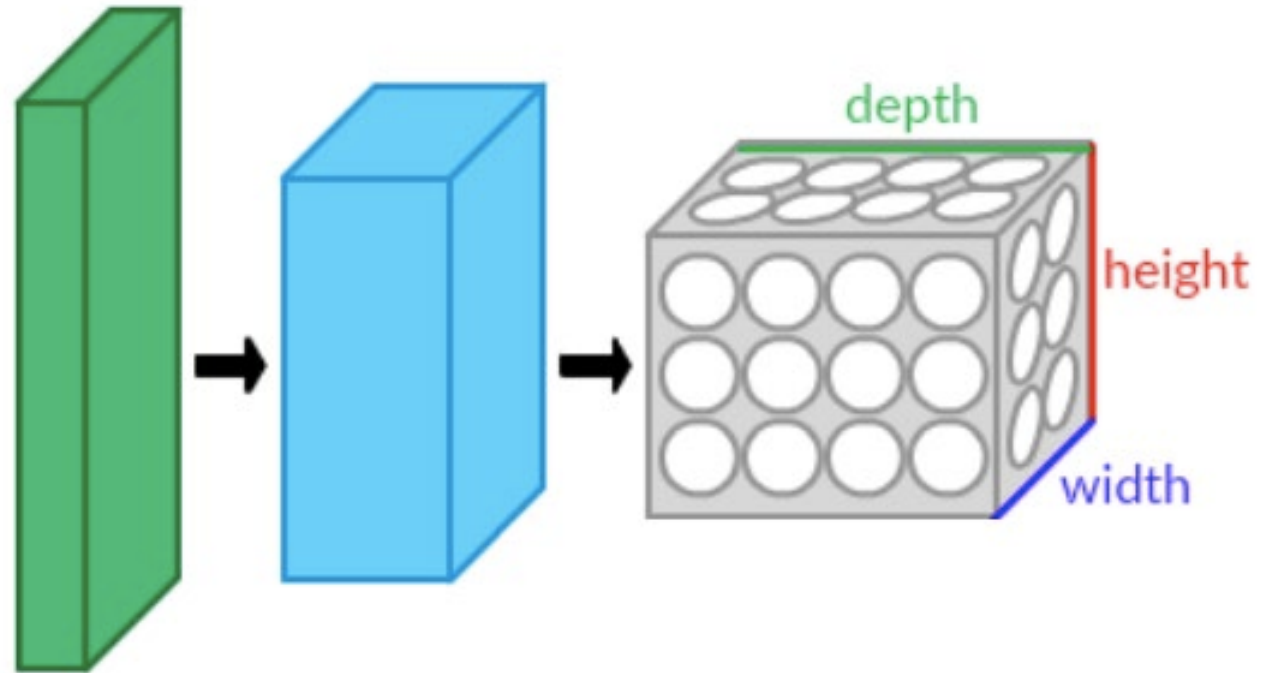
CNN are typically made of blocks that include:

- Convolutional layers
- Nonlinearities (activation functions)
- Pooling Layers (Subsampling / 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

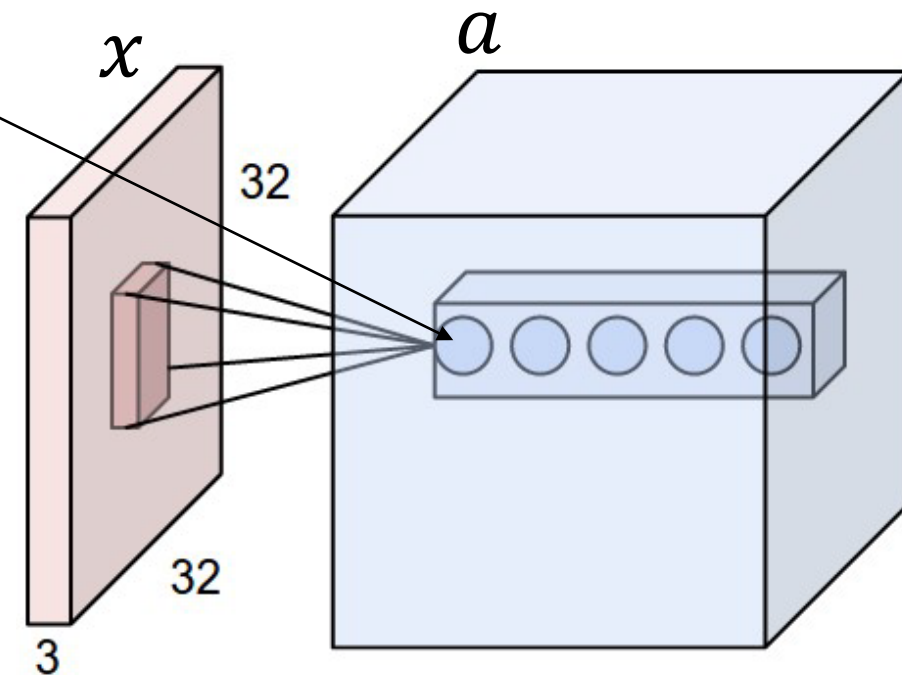
Convolutional Layers

Convolutional layers "mix" all the input components

The output is a linear combination of **all the values in a region of the input, considering all the channels**

$$a(r, c, 1) = \sum_{(u,v) \in U, k} w^1(u, v, k) x(r + u, c + v, k) + b^1$$

Filters need to have **the same number of channels** as the input, to process all the values from the input layer

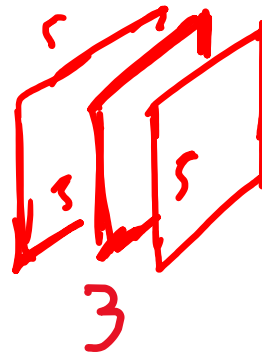


Convolutional Layers

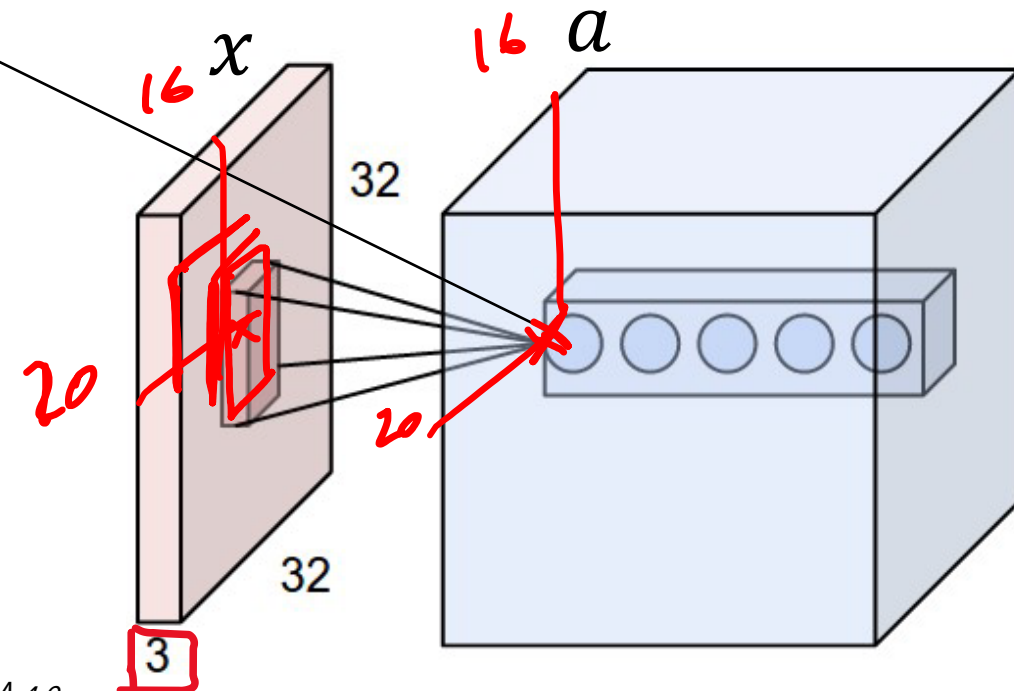
Convolutional layers "mix" all the input components

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

$$a(r, c, 1) = \sum_{(u,v) \in U, k} w^1(u, v, k) x(r + u, c + v, k) + b^1$$



Filters need to have the same number of channels as the input, to process all the values from the input layer



Convolutional Layers

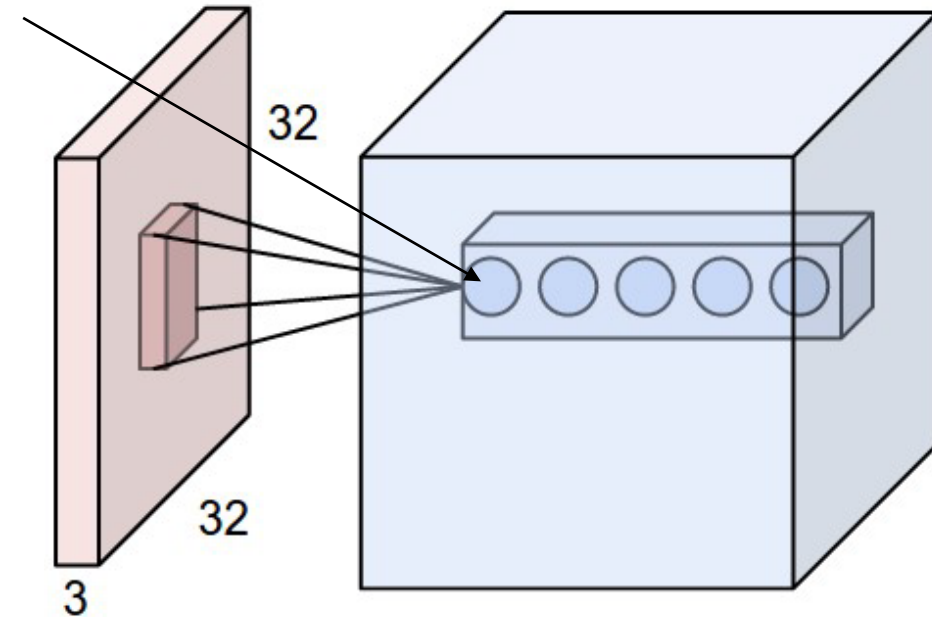
Convolutional layers "mix" all the input components

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

$$a(r, c, 1) = \sum_{(u,v) \in U, k} w^1(u, v, k) x(r + u, c + v, k) + b^1$$

The parameters of this layer are called **filters**.

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



Convolutional Layers

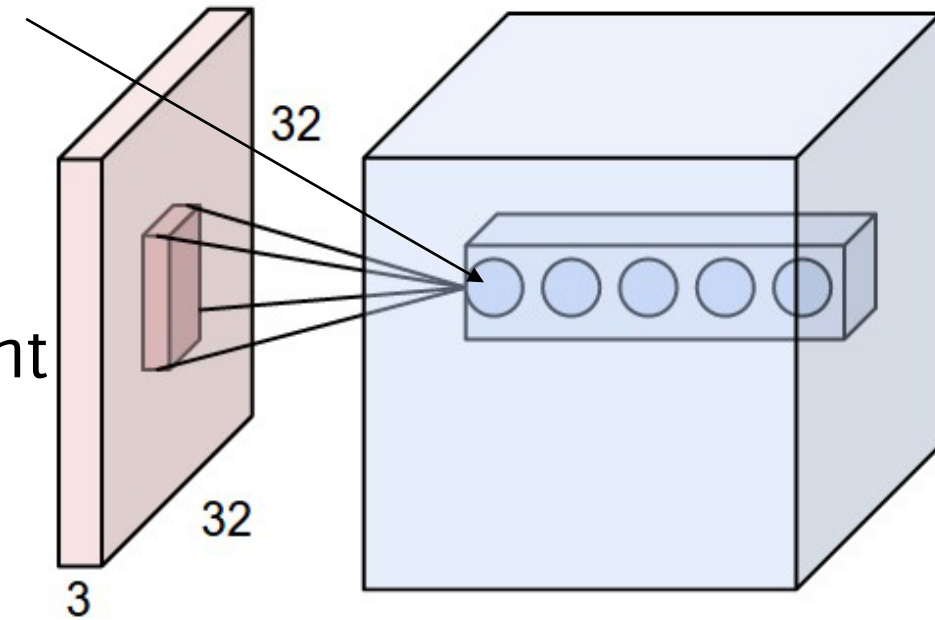
Convolutional layers "mix" all the input components

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

$$a(r, c, 1) = \sum_{(u,v) \in U, k} w^1(u, v, k) x(r + u, c + v, k) + b^1$$

The **spatial dimension**:

- spans a small neighborhood U (local processing, it's a convolution)
- U needs to be specified, it is a very important attribute of the filter



Convolutional Layers

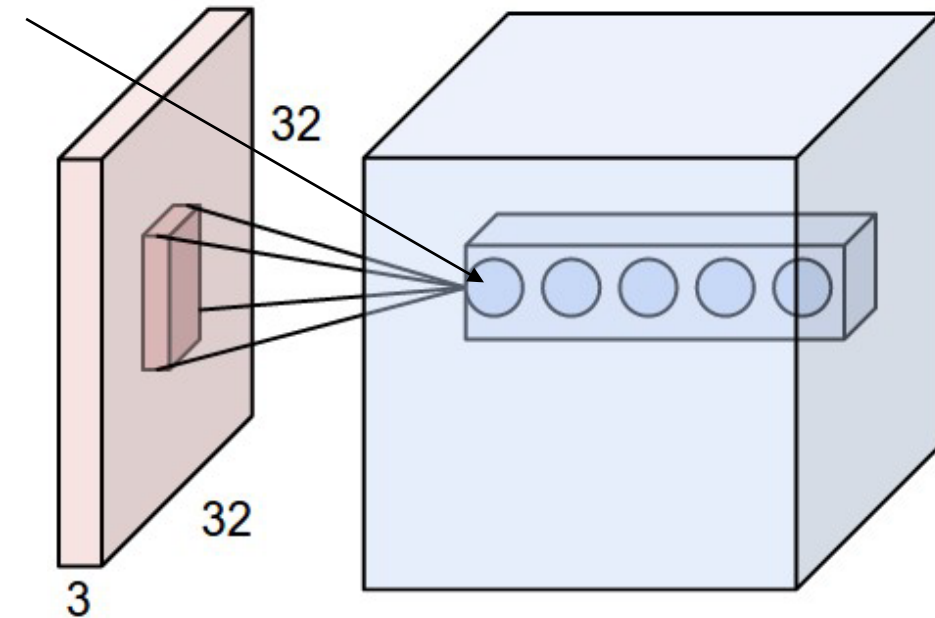
Convolutional layers "mix" all the input components

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

$$a(r, c, 1) = \sum_{(u,v) \in U} w^1(u, v, k) x(r + u, c + v, k) + b^1$$

The channel dimension:

- spans the entire input depth (no local processing, like spatial dimension)
- there is no need to specify that in the filter attributes

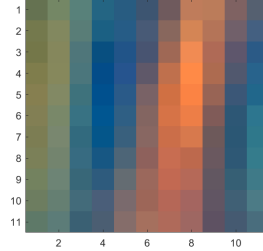


Convolutional Layers

I

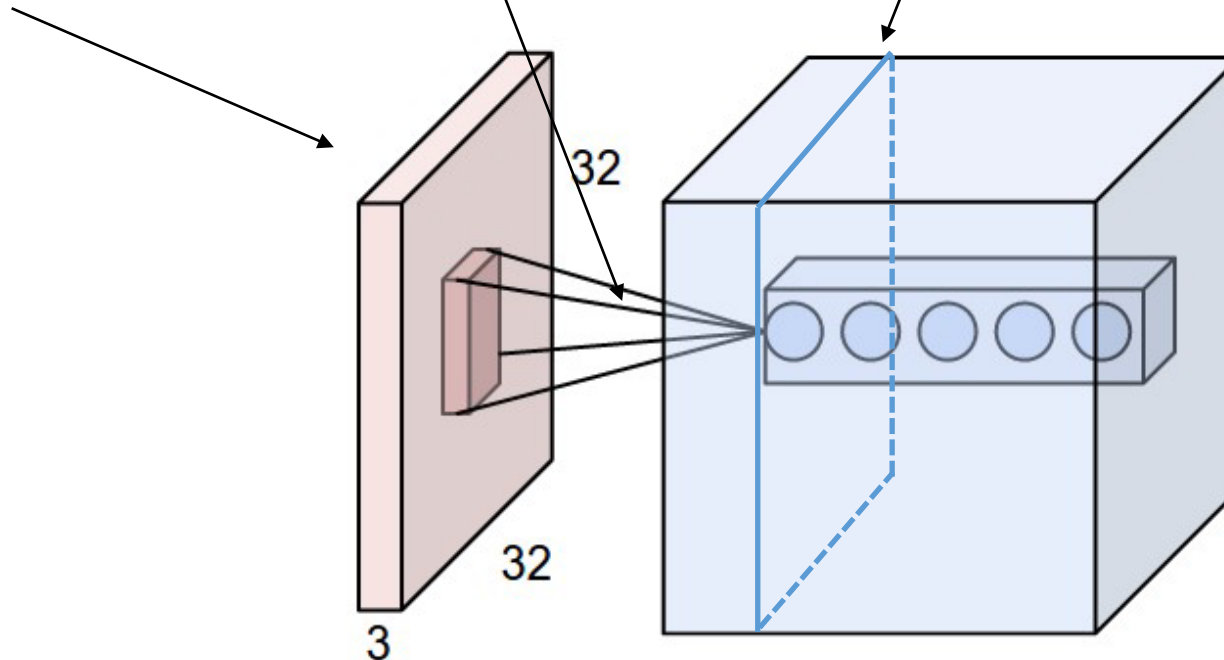
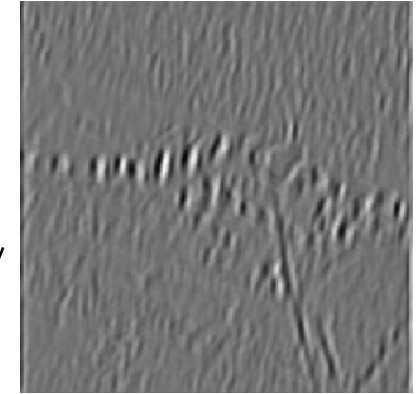


w^1



b^1

$a(:, :, 1)$



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

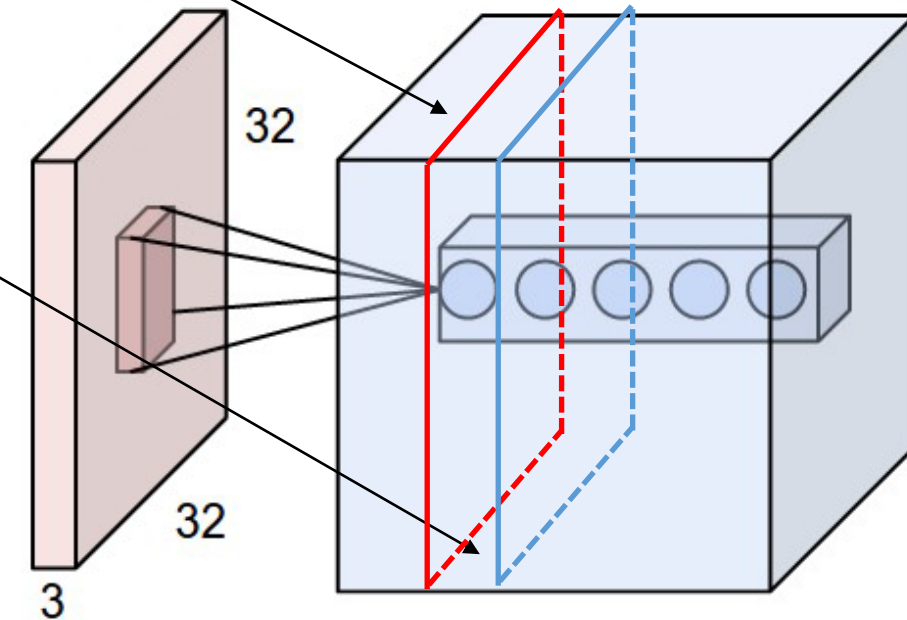
Convolutional Layers

Different filters yield different layers in the output

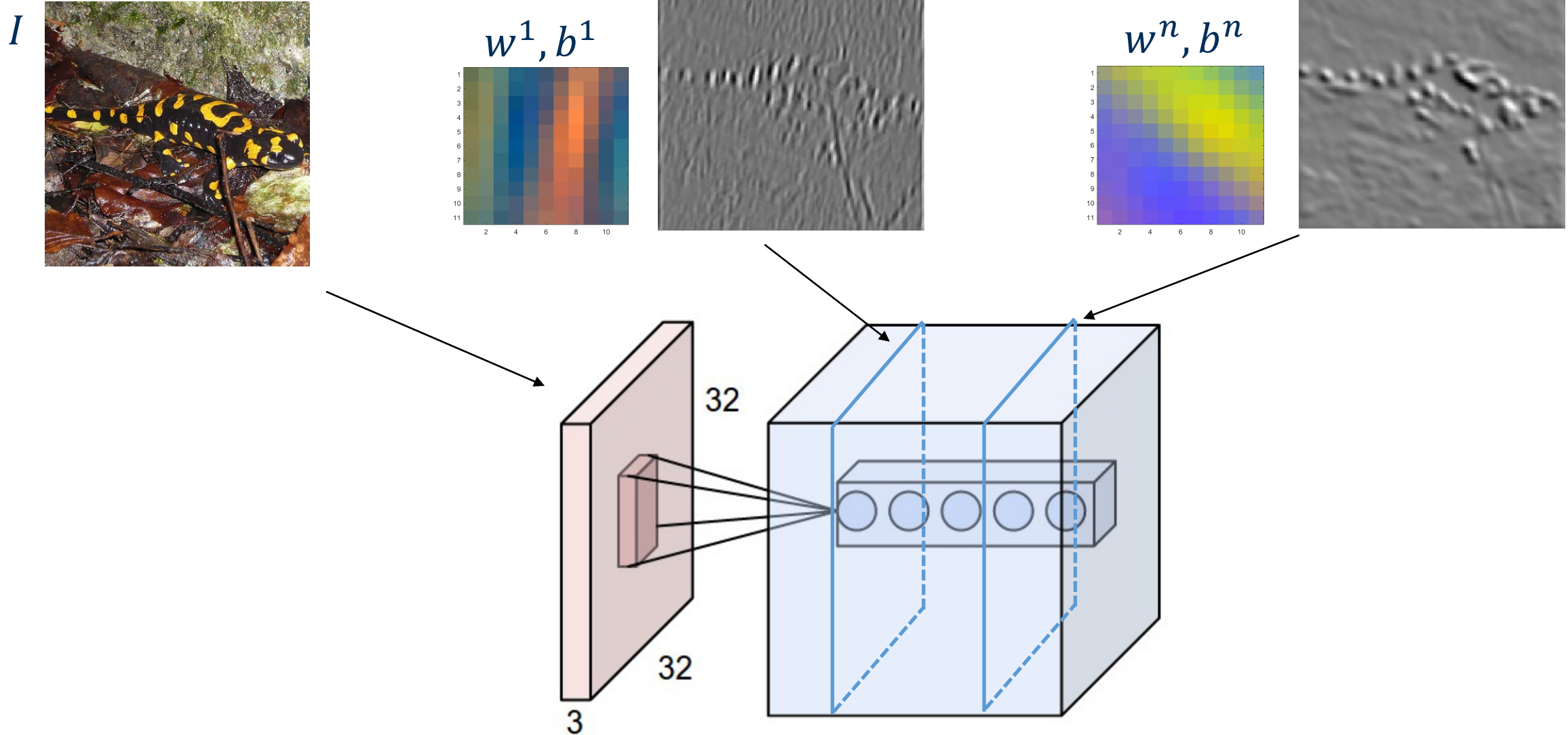
$$a(r, c, 1) = \sum_{(u,v) \in U,k} w^1(u, v, k) x(r + u, c + v, k) + b^1$$

$$a(r, c, 2) = \sum_{(u,v) \in U,k} w^2(u, v, k) x(r + u, c + v, k) + b^2$$

Different filters of the same layer have the same spatial extent



Convolutional Layers



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

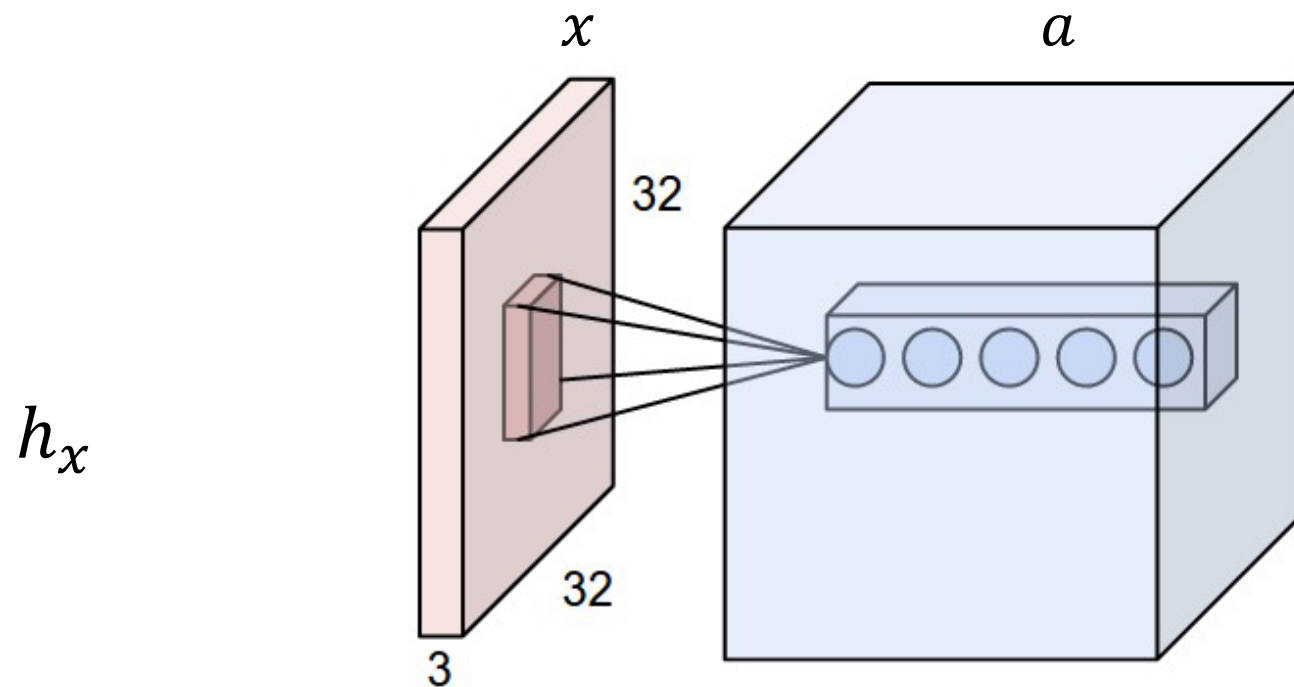
Recap: Convolutional Layers

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

Overall number of parameters

Conv2(



Convolutional Layers, remarks:

Given:

```
conv2 = tfkl.Conv2D(  
    filters = n_f,  
    kernel_size = (h_x, h_y),  
    activation = 'relu',  
    name = 'conv2'  
)
```

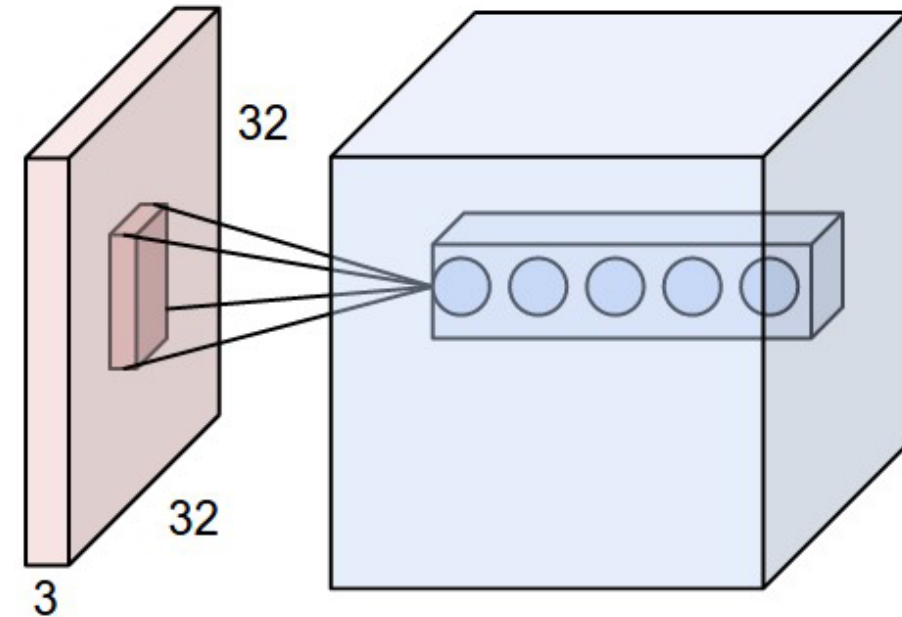
The parameters are the weights + one bias per filter

The overall number of parameters is

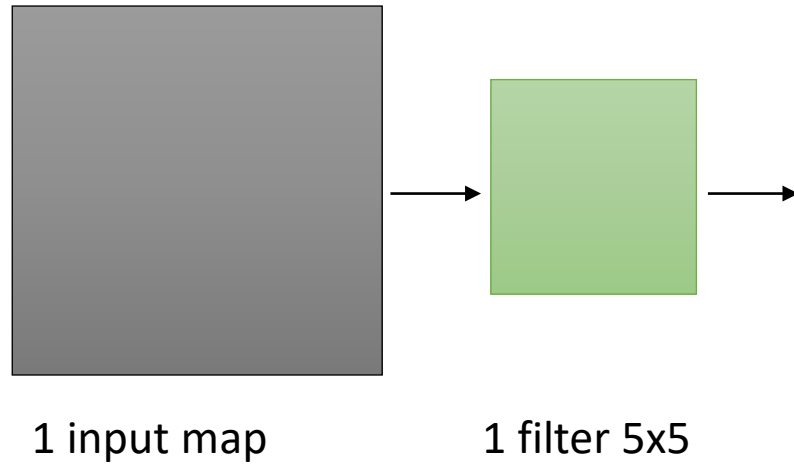
$$(h_x \cdot h_y \cdot d) \cdot n_f + n_f$$

Where d is the **depth of the input activation**

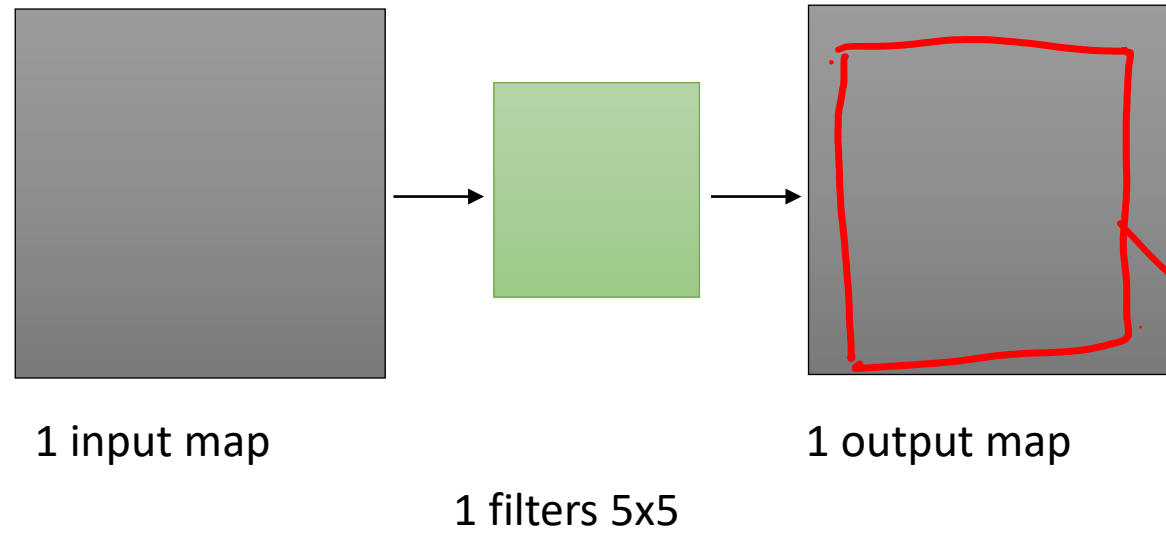
Layers with the same attribute can have different number of parameters depending on where these are located



CNN Arithmetic

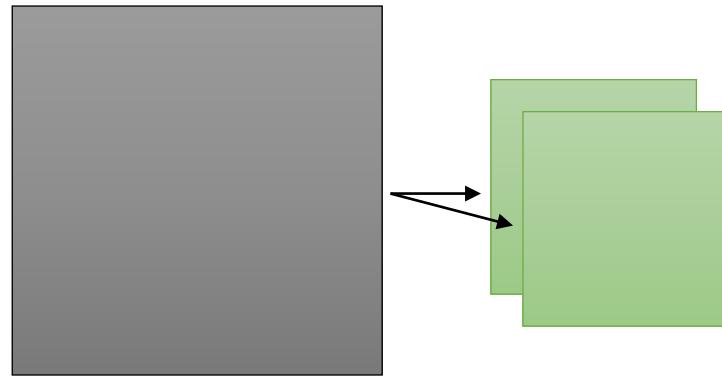


CNN Arithmetic



*in core
pixels
"what"*

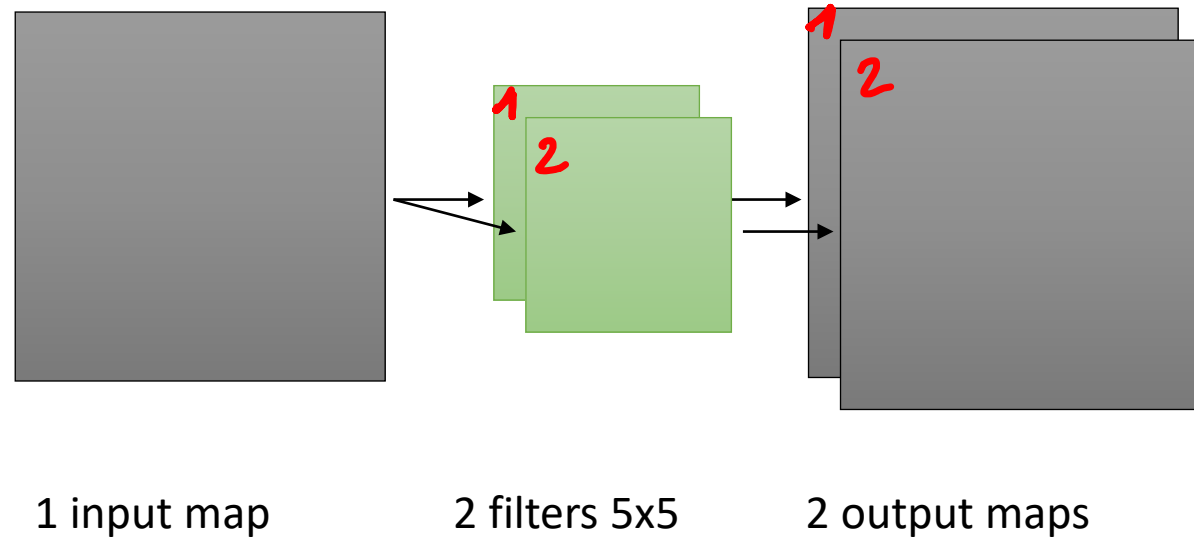
CNN Arithmetic



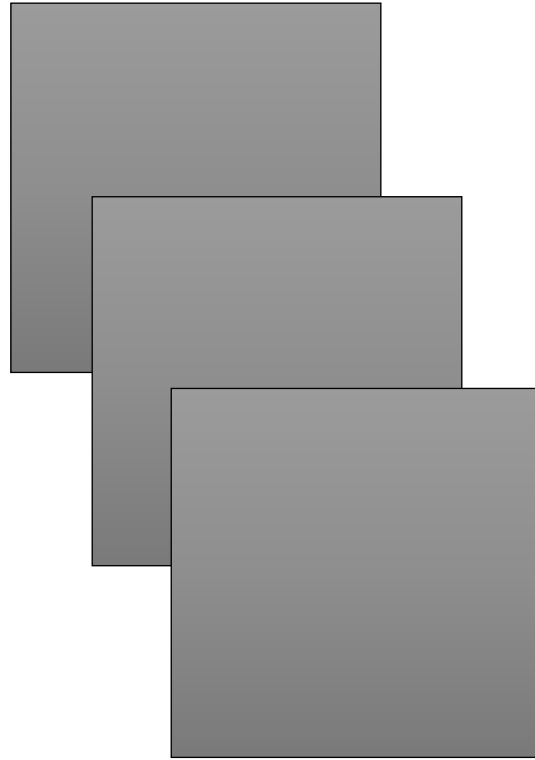
1 input map

2 filters 5x5

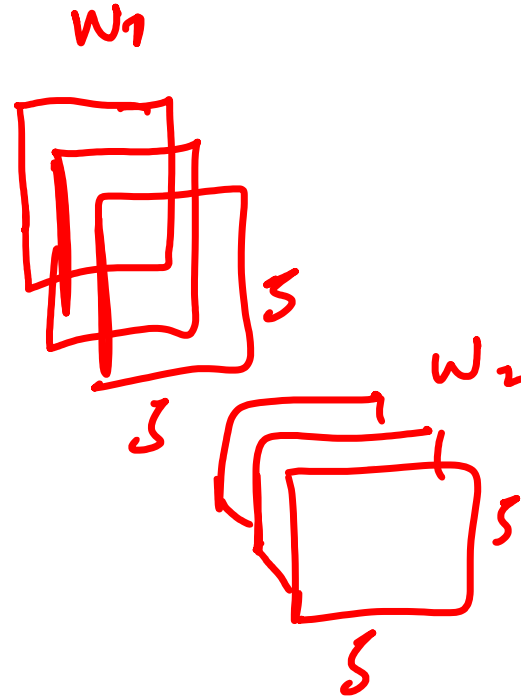
CNN Arithmetic



CNN Arithmetic

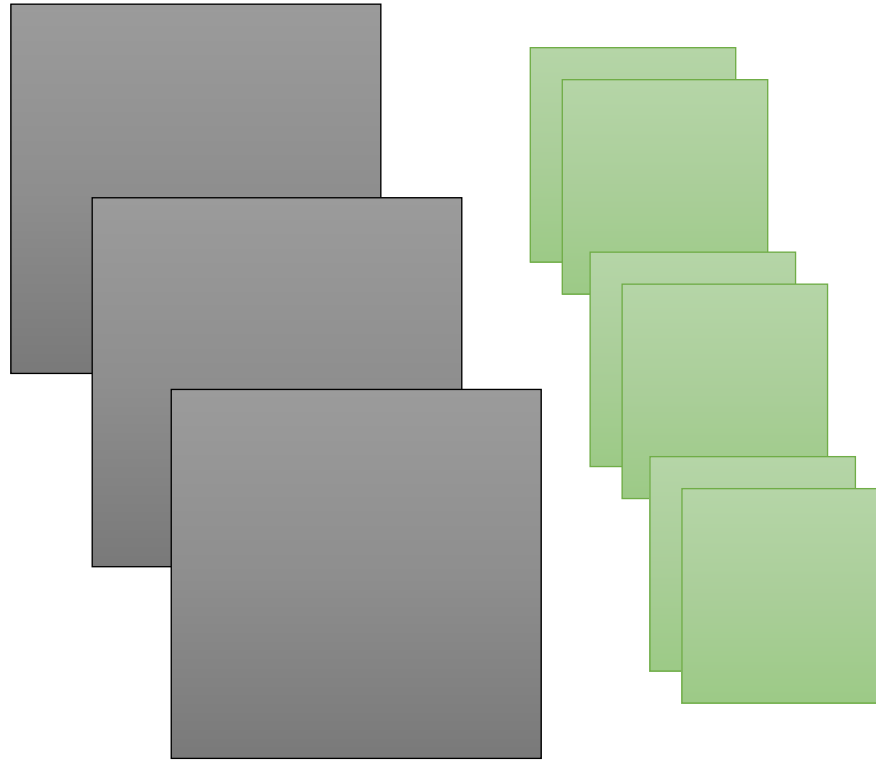


3 input maps



2 filters 5x5

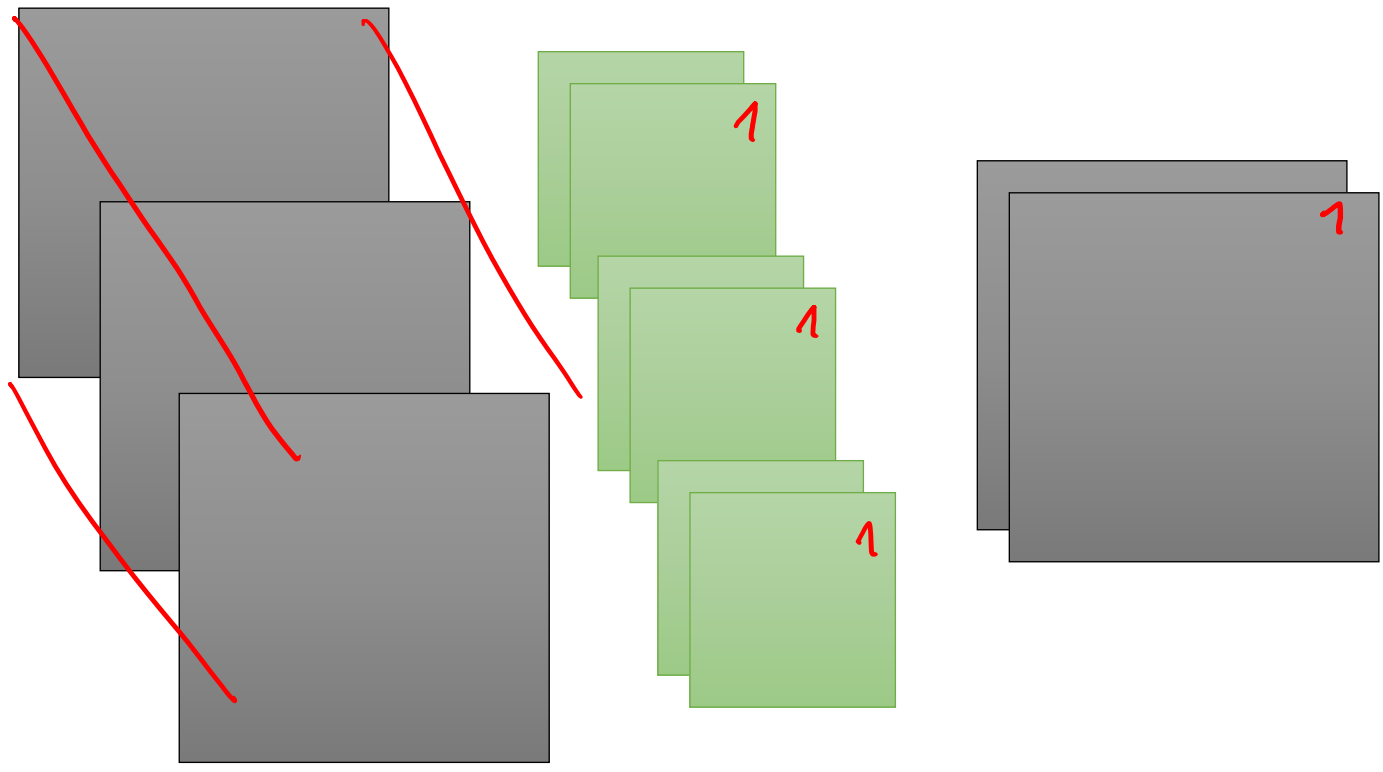
CNN Arithmetic



3 input maps

2 filters 5x5

CNN Arithmetic

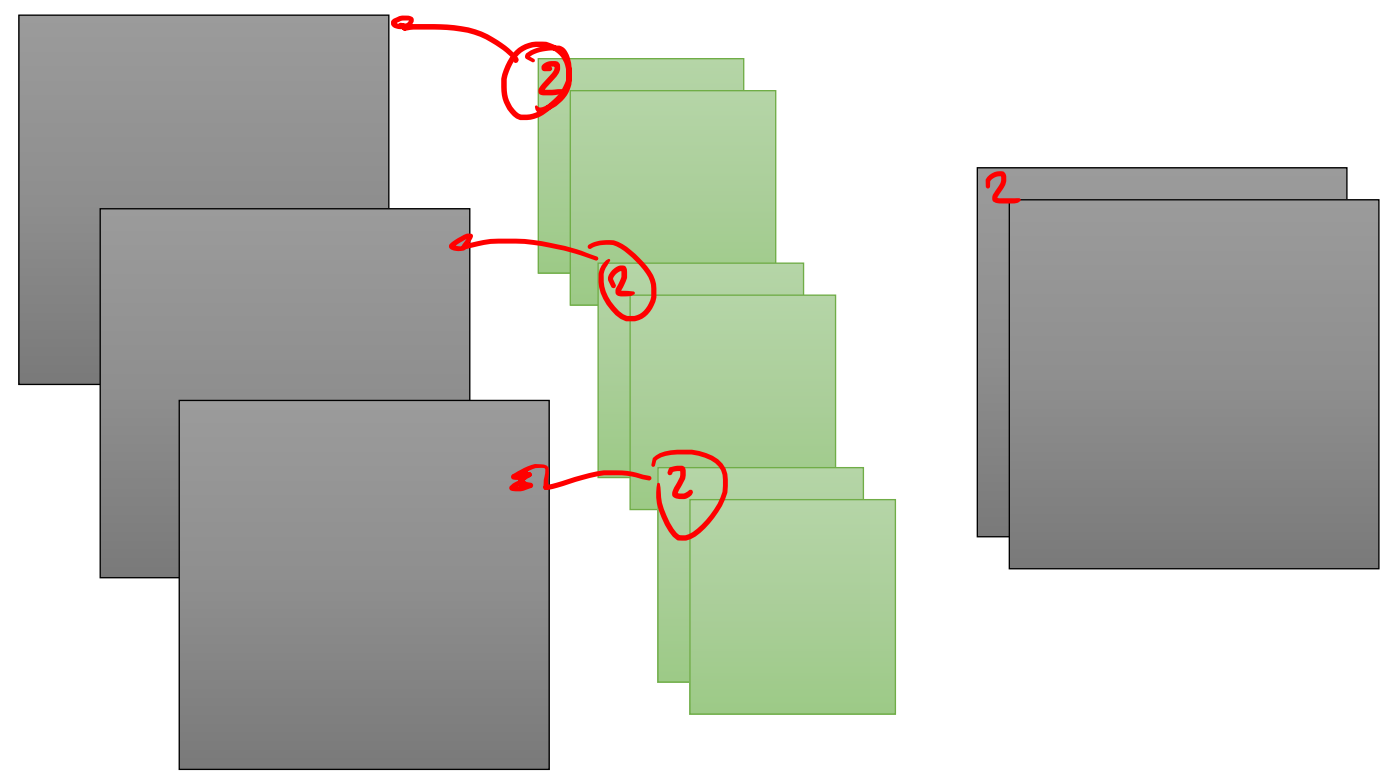


3 input maps

2 filters 5x5

2 output maps

CNN Arithmetic

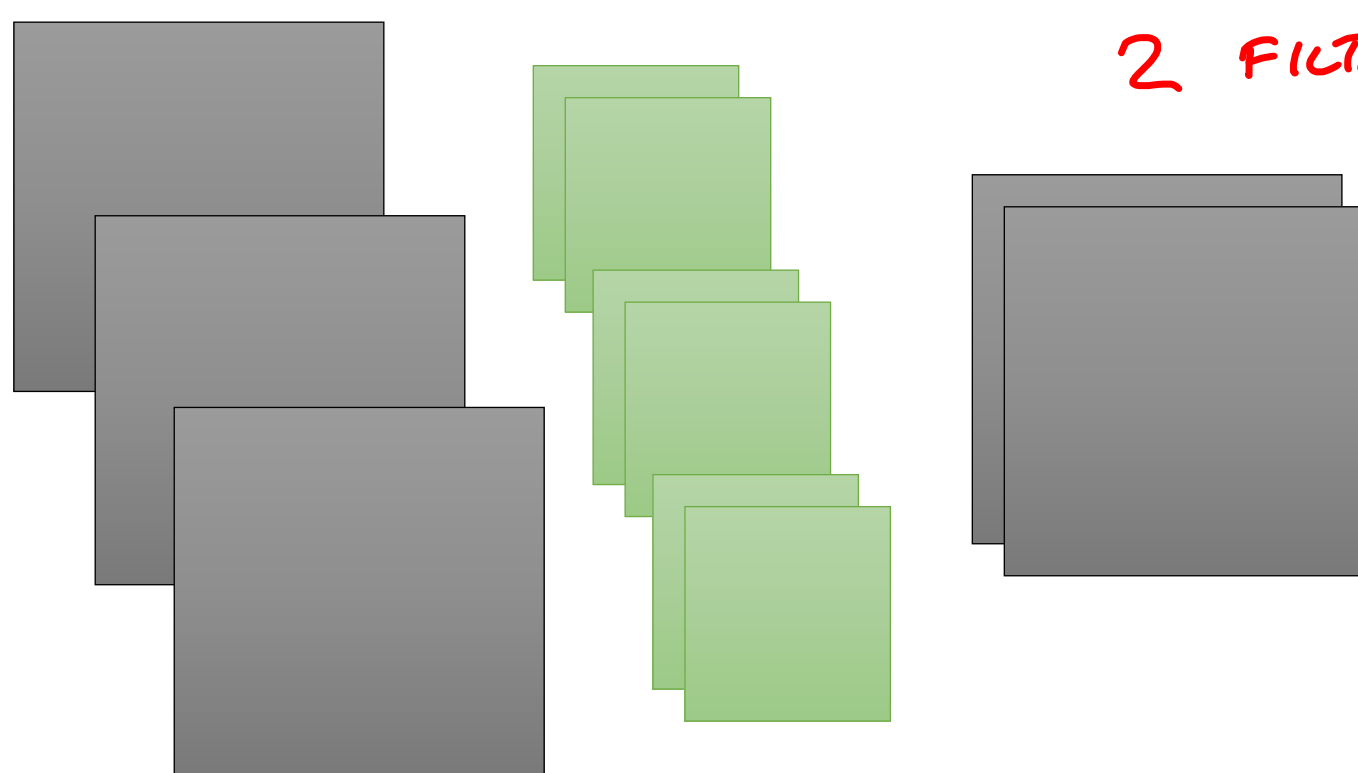


3 input maps

2 filters 5x5

2 output maps

CNN Arithmetic



2 FILTERS $3 \times 5 \times 3$
150 params
+ 2
152 params

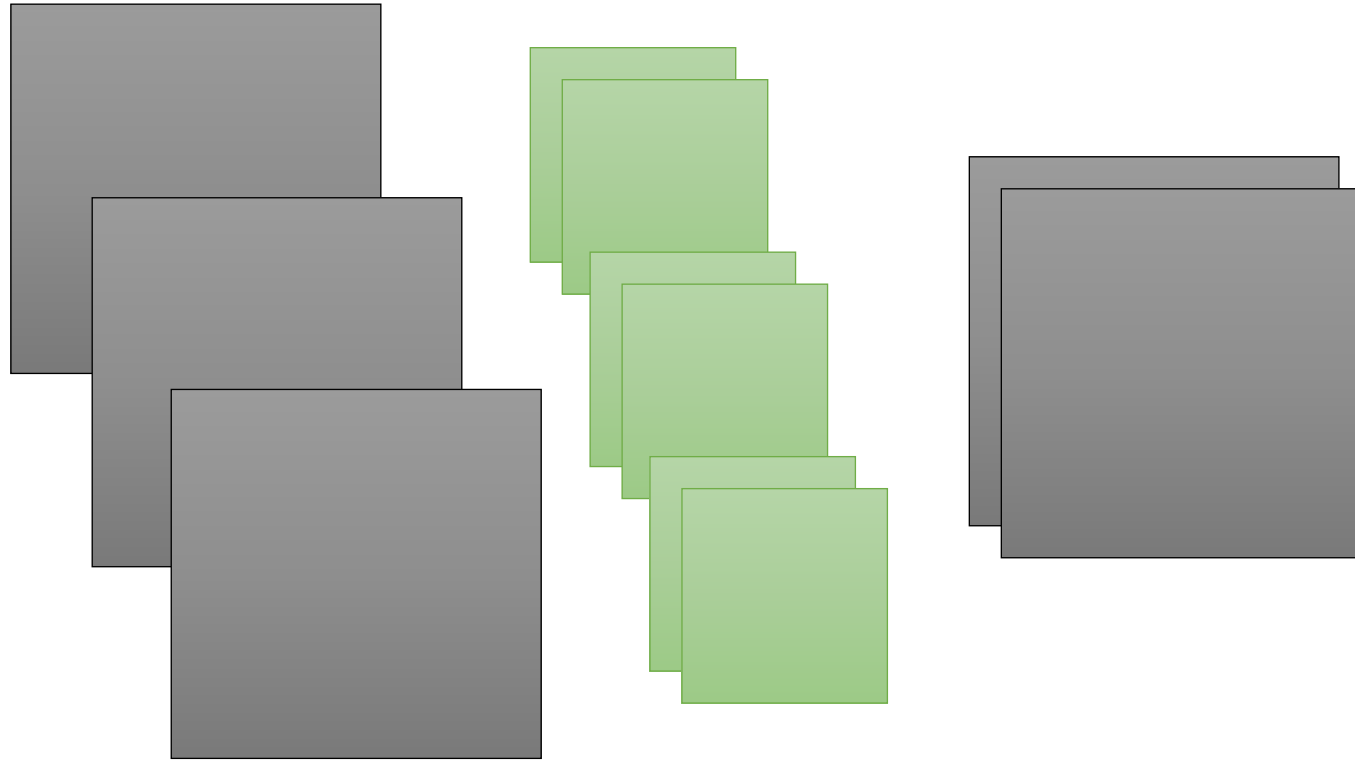
3 input maps

2 filters 5×5

2 output maps

Quiz: how many parameters
does this layer have?

CNN Arithmetic



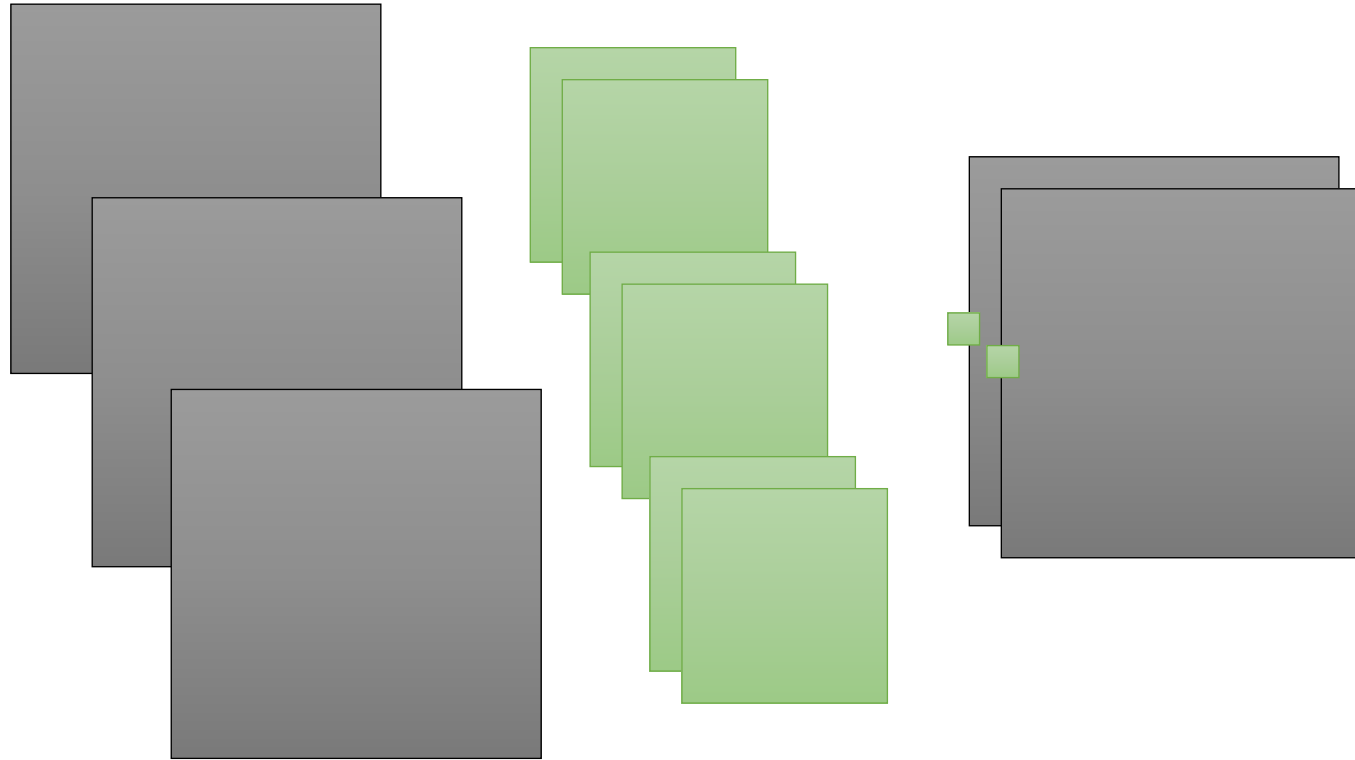
3 input maps

2 filters 5x5

2 output maps

= 150 parameters in the filters

CNN Arithmetic

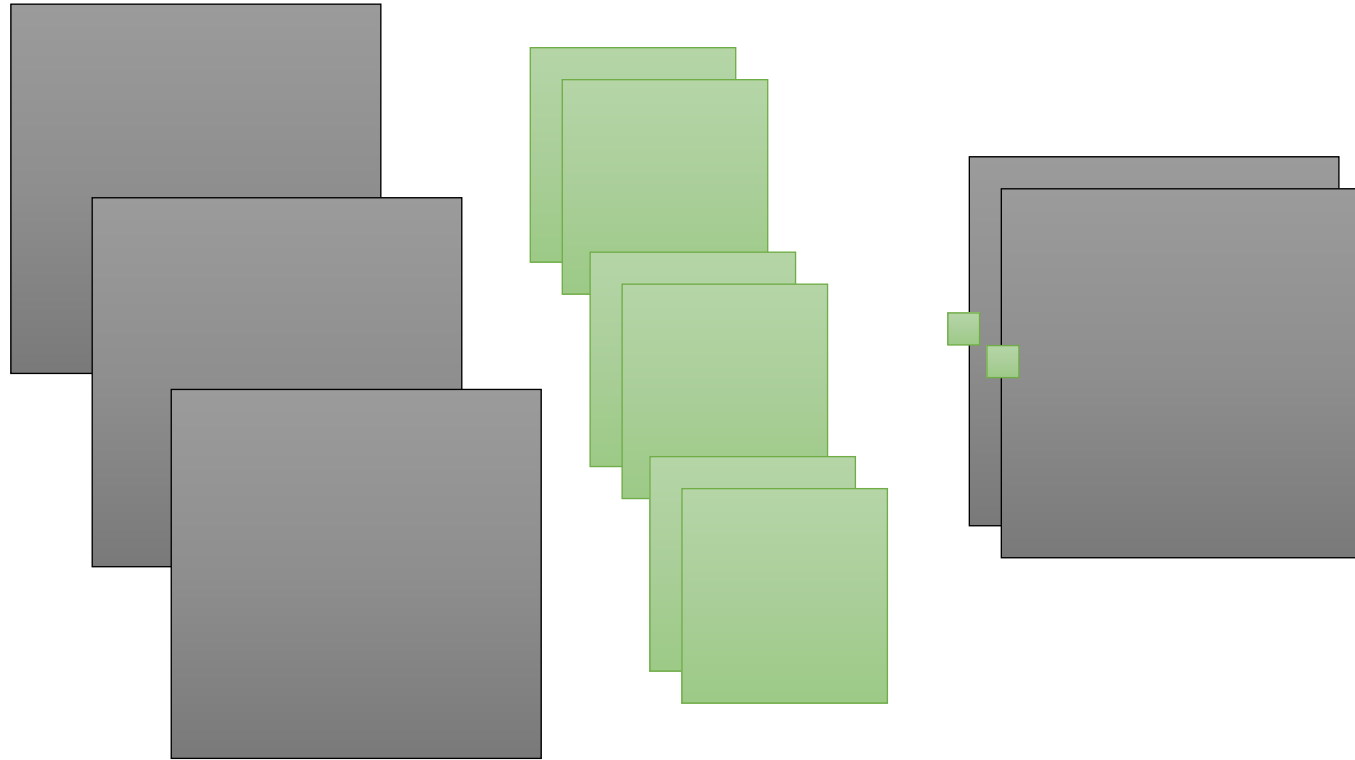


3 input maps

2 filters 5x5
= 150 ...

2 output maps
+ 2 biases

CNN Arithmetic



3 input maps

2 filters 5x5

2 output maps

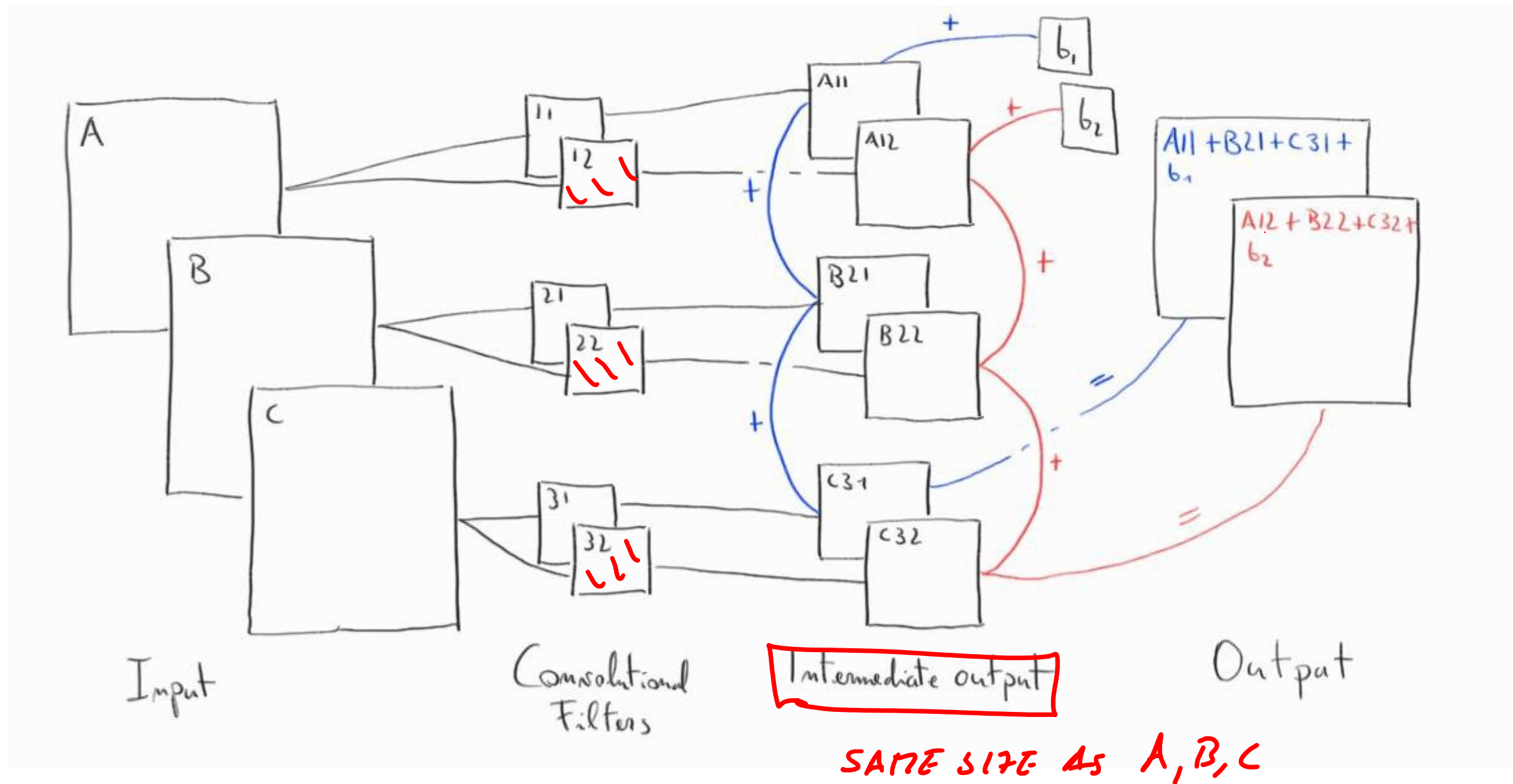
= 150 ...

+ 2 biases

= 152 trainable parameters (weights)

output maps
=
of filters
in 1st layer

To Recap...



Other Layers

Activation and Pooling

Activation Layers

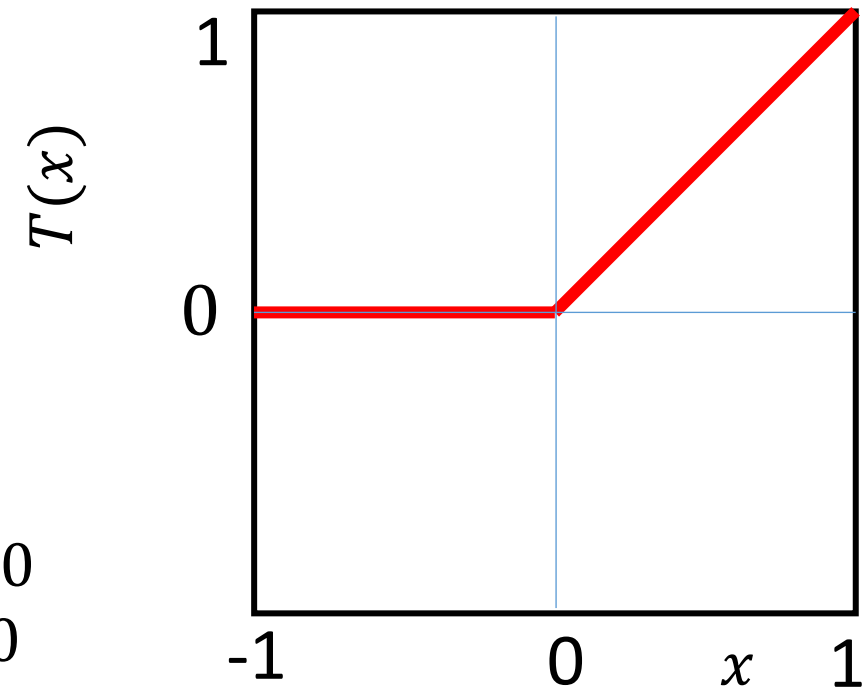
Introduce nonlinearities in the network, otherwise the CNN might be equivalent to a linear classifier...

Activation functions are scalar functions, namely they operate on each single value of the volume. **Activations don't change volume size**

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)

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

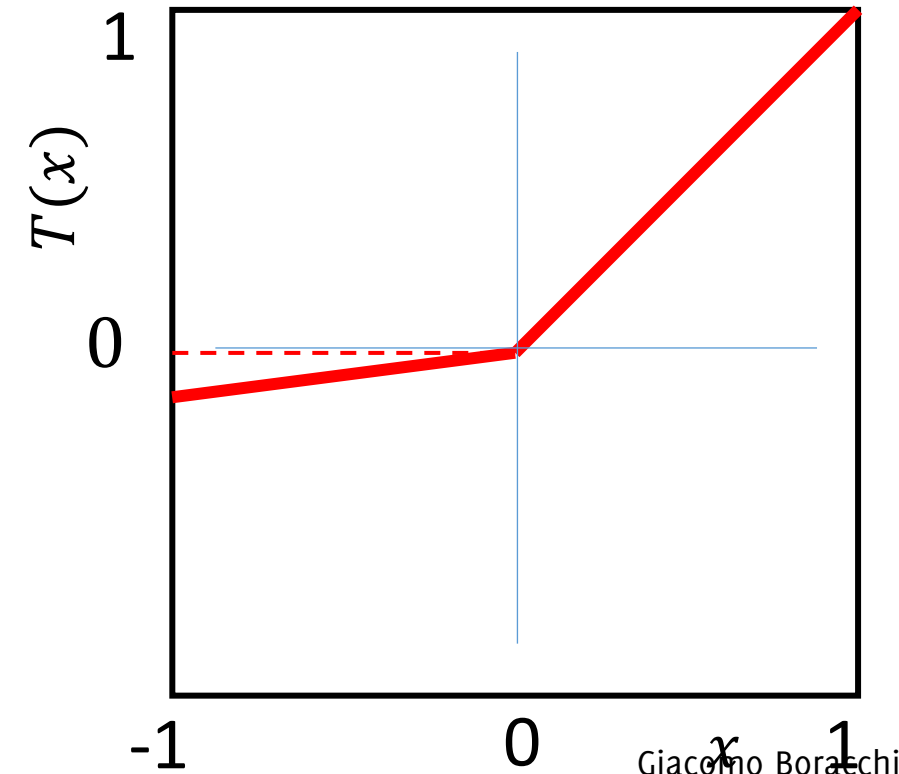


Activation Layers

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

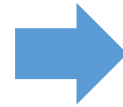
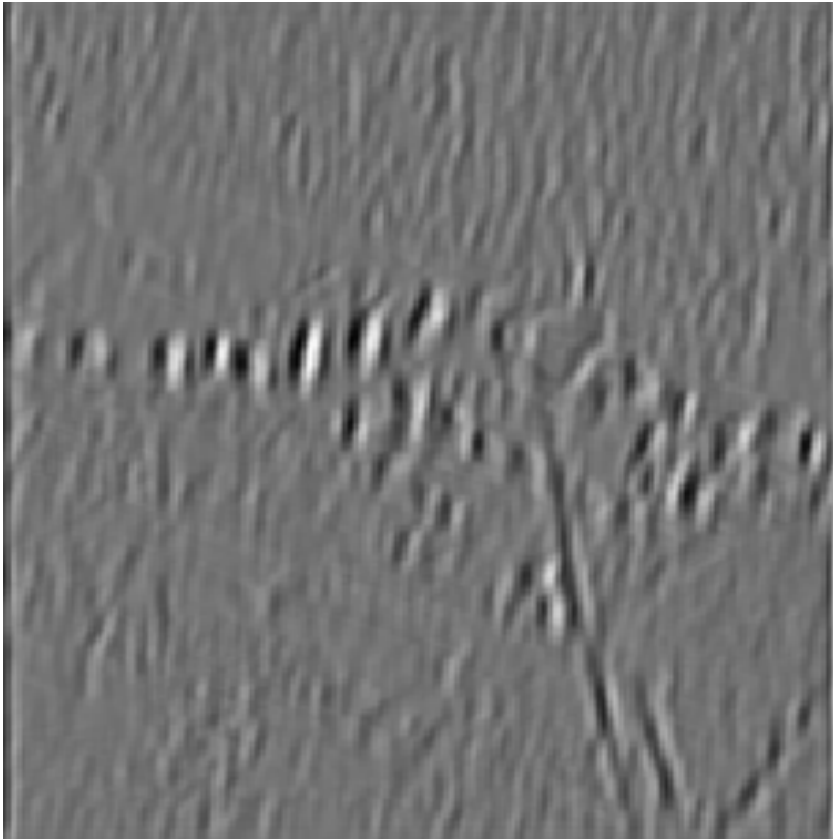
$$T(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0.01 * x & \text{if } x < 0 \end{cases}$$



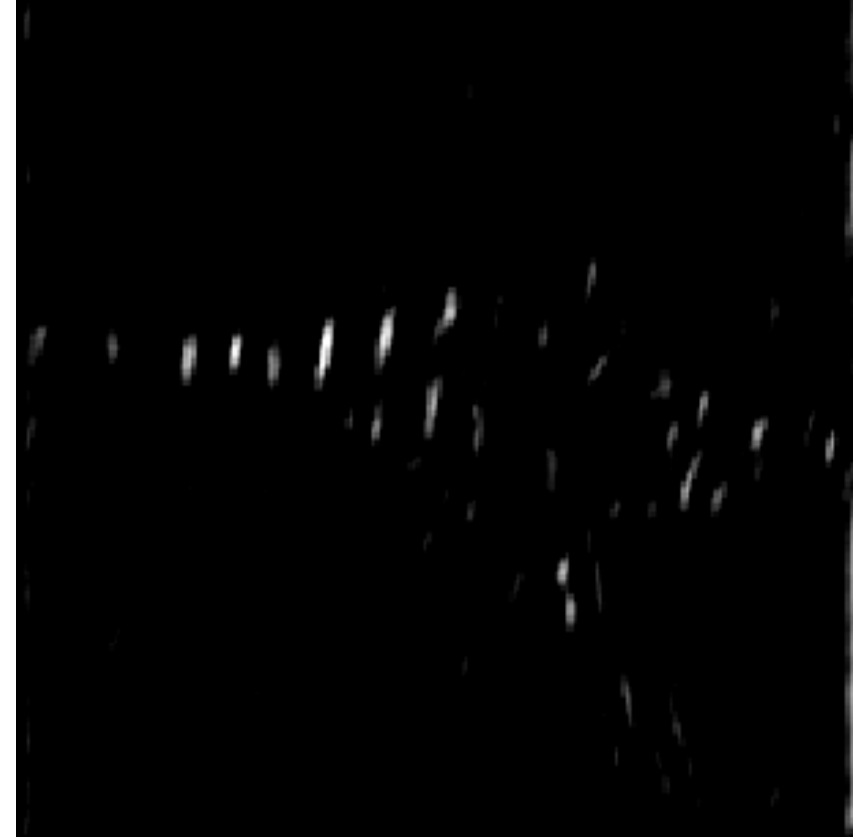
ReLu

Acts separately on each layer

a^1



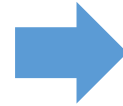
$ReLU(a^1)$



ReLu

Acts separately on each layer

a^2



$ReLU(a^2)$



Activation Layers

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

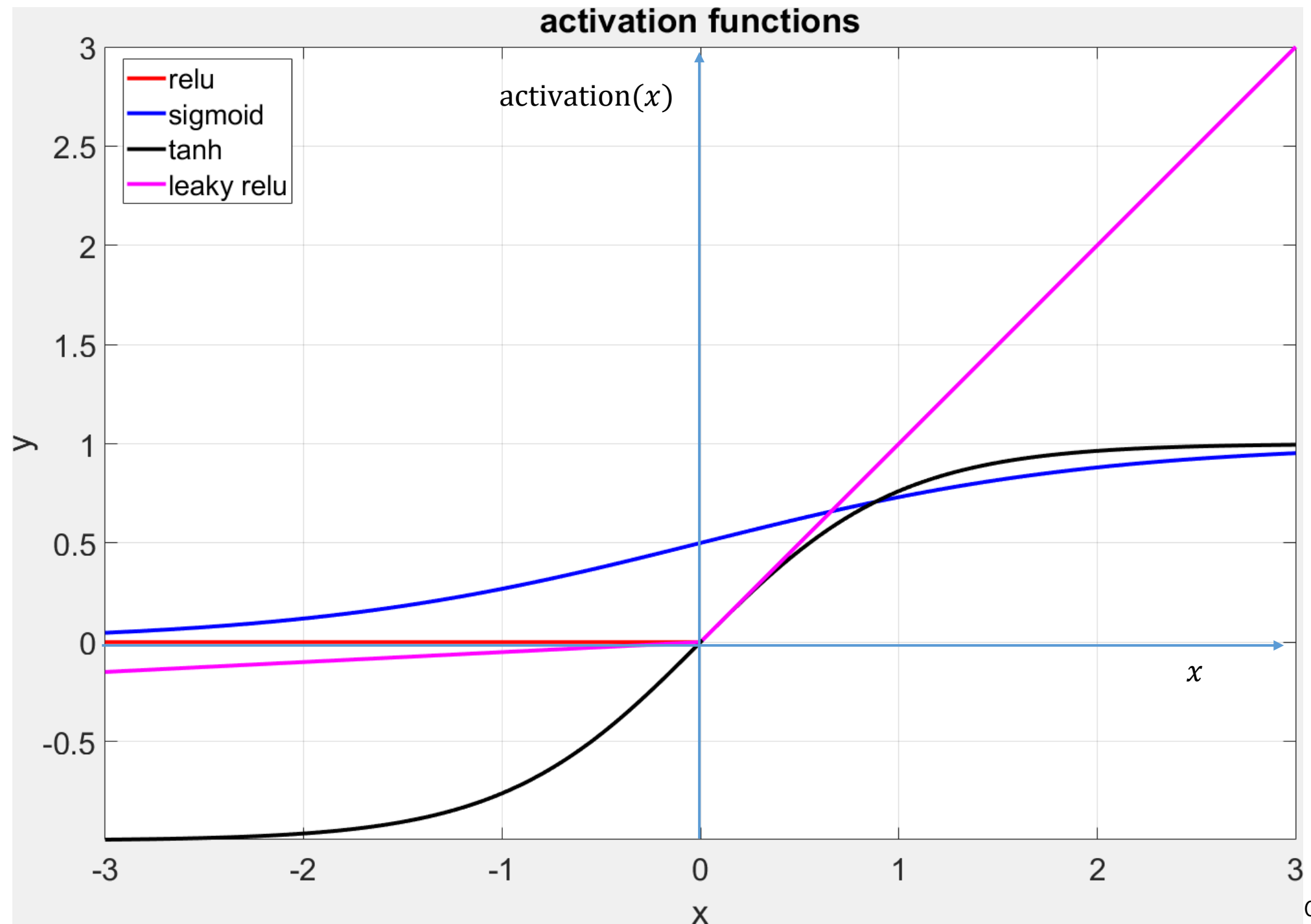
TANH (hyperbolic Tangent): has a range $(-1,1)$, continuous and differentiable

$$T(x) = \frac{2}{1 + e^{-2x}} - 1$$

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

$$S(x) = \frac{1}{1 + e^{-2x}}$$

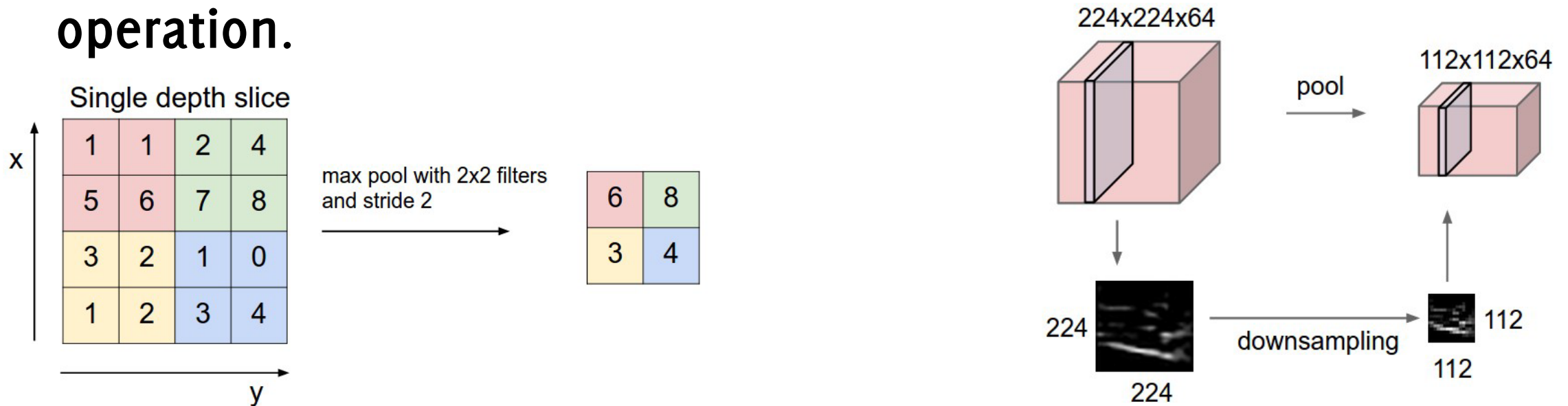
These activation functions are mostly popular in **MLP architectures**



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**.

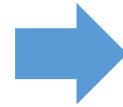
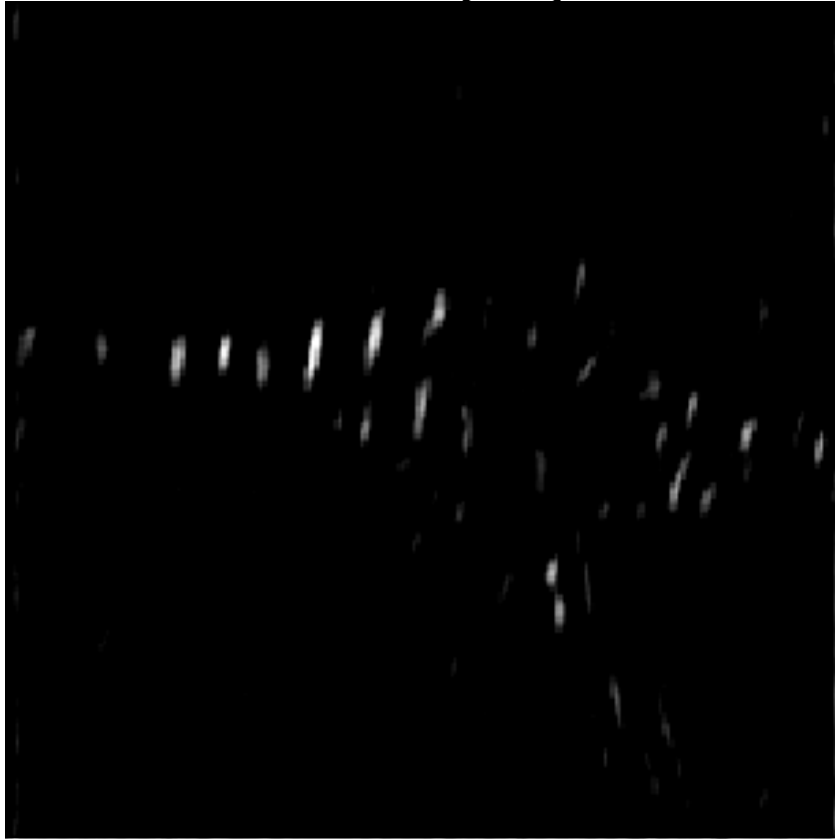


In a 2x2 support it discards 75% of samples in a volume

Max-Pooling (MP)

Acts separately on each layer

$ReLU(a^1)$



$MP(ReLU(a^1))$



Strides in Pooling Layers

Typically, the **stride** is assumed equal to the **pooling size**

- Where note specified, maxpooling has stride 2×2 and reduces image size to 25%

It is also possible to use a different stride. In particular, it is possible to adopt stride = 1, which does not reduce the spatial size, but just perform pooling on each pixel

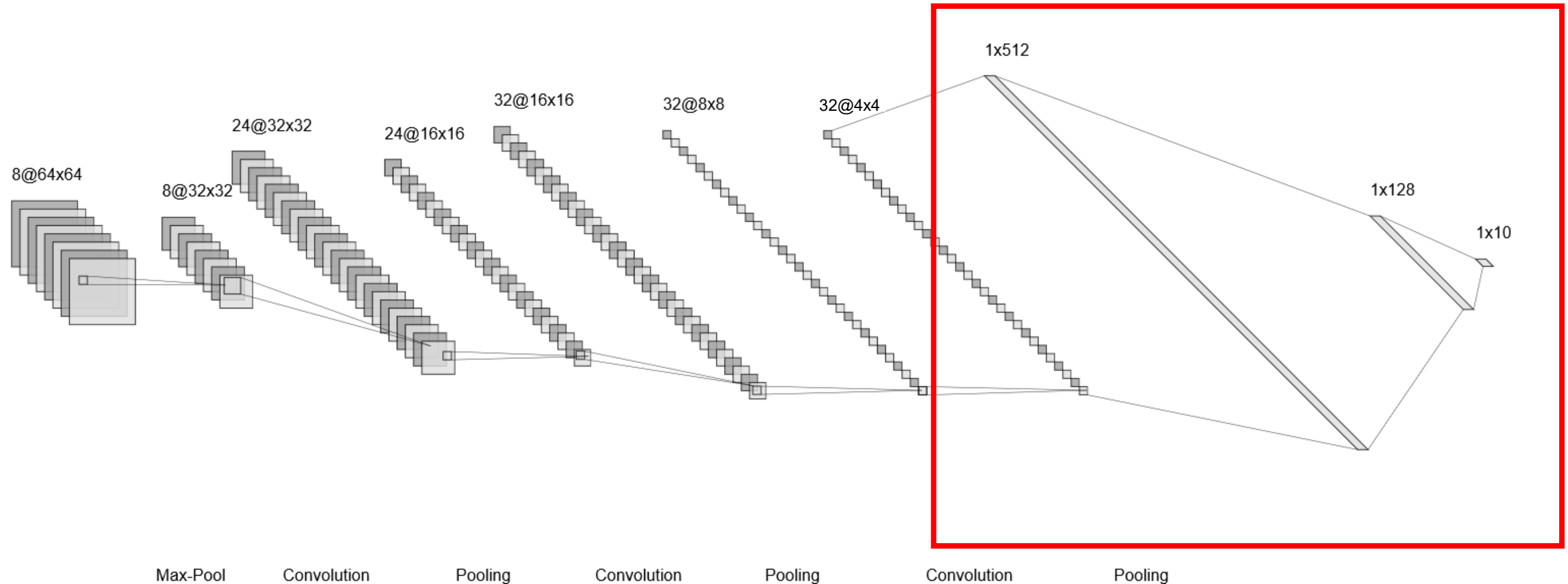
- this operation makes sense with nonlinear pooling (max-pooling)

Dense Layers

As in feed-forward NN

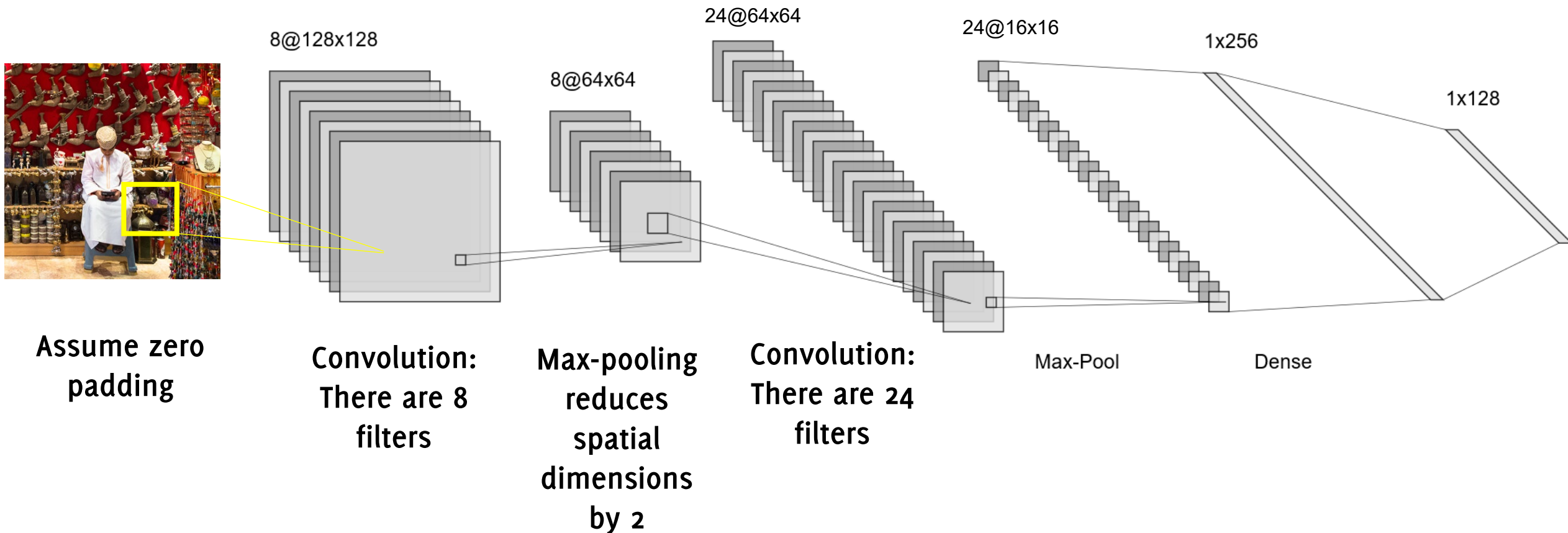
The Dense Layers

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



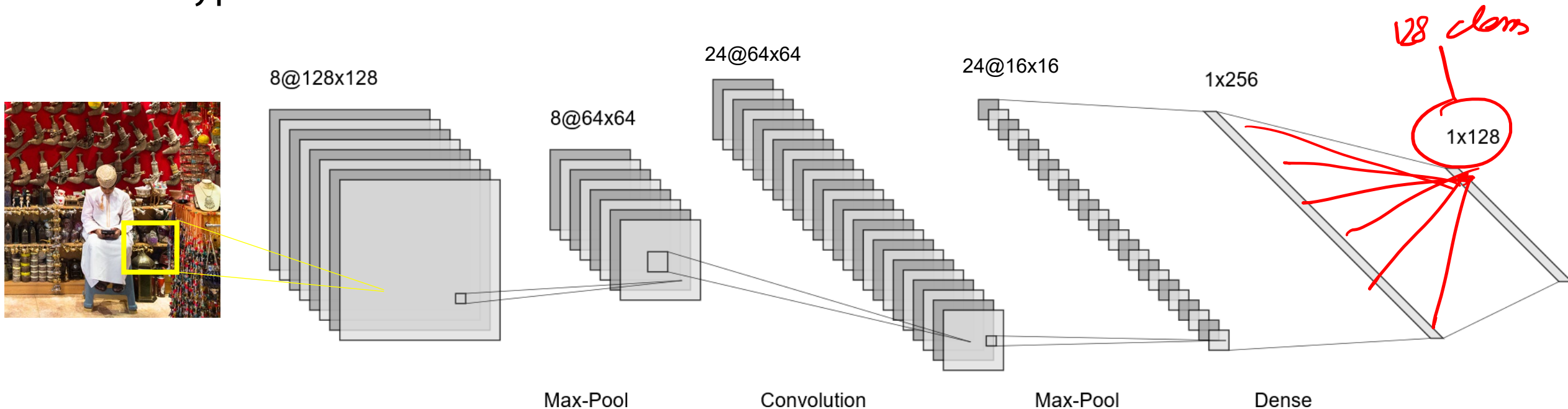
Convolutional Neural Networks (CNN)

The typical architecture of a convolutional neural network



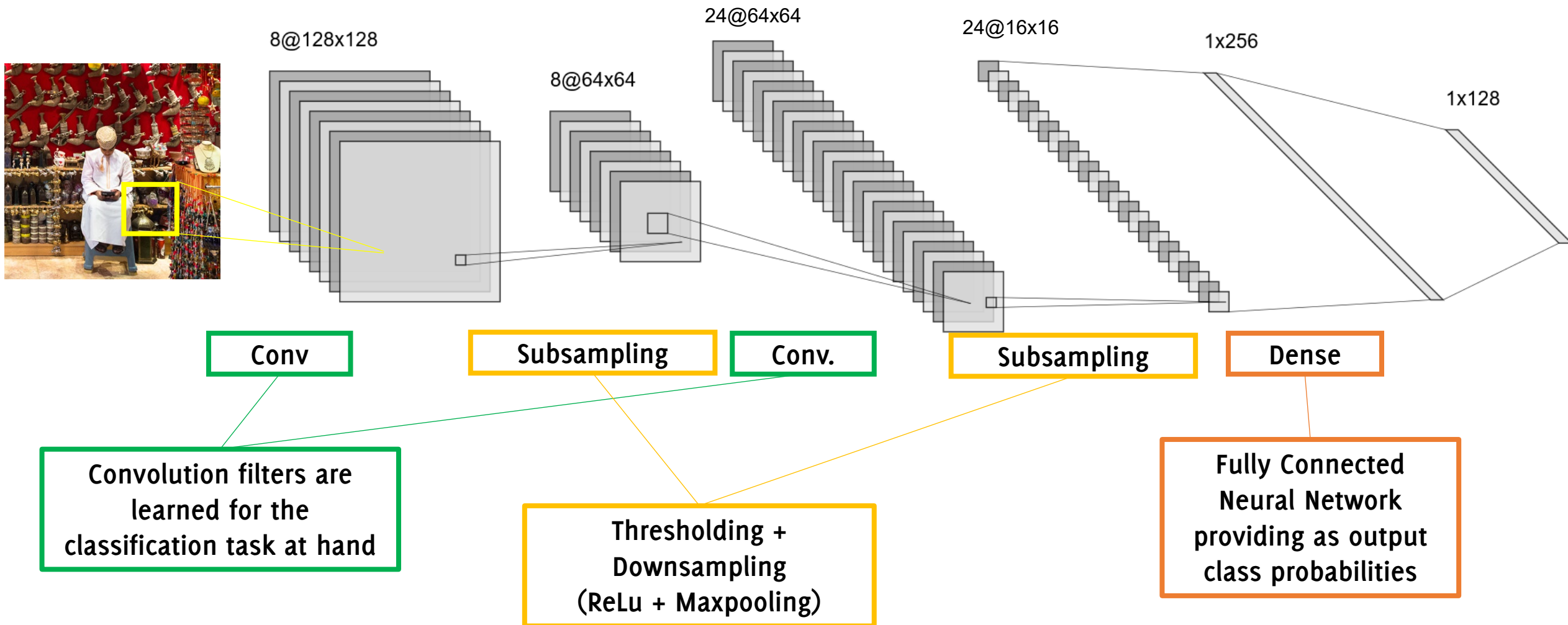
Convolutional Neural Networks (CNN)

The typical architecture of a convolutional neural network

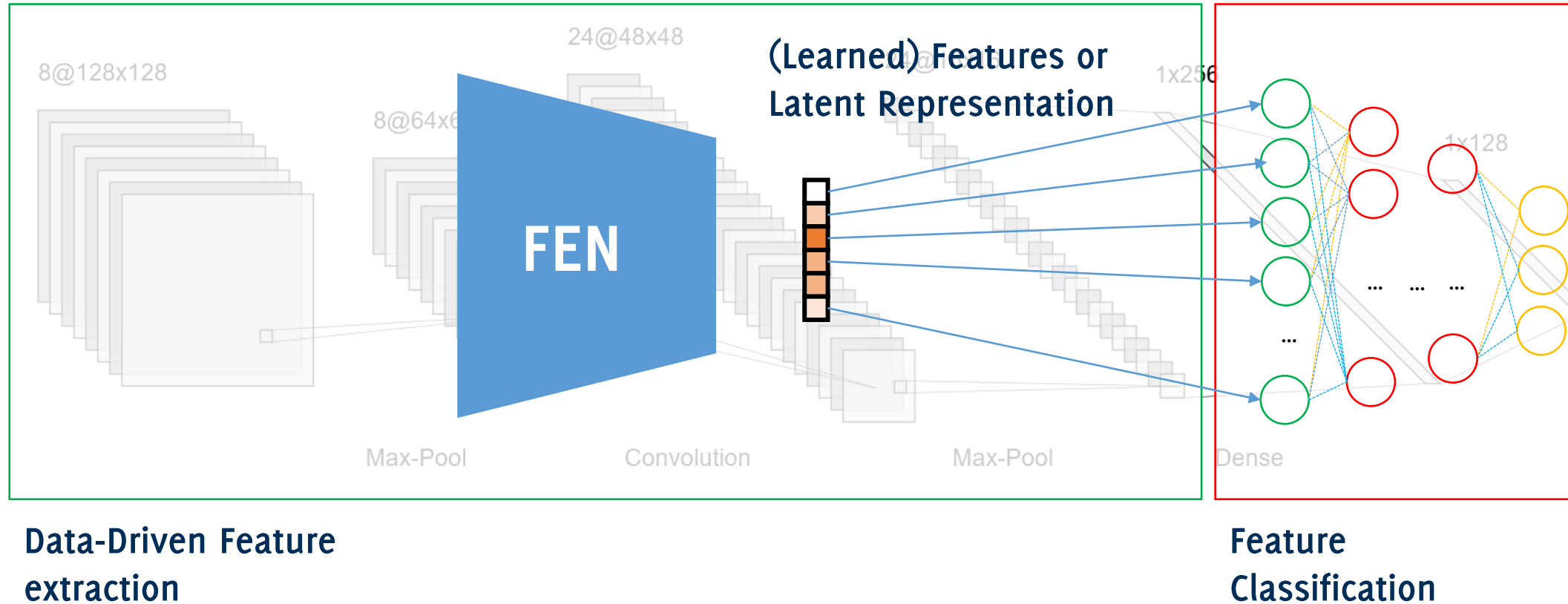


The output of the **fully connected (FC) layer** has the same size as the **number of classes**, and provides a **score** for the input image to belong to each class

Convolutional Neural Networks (CNN)

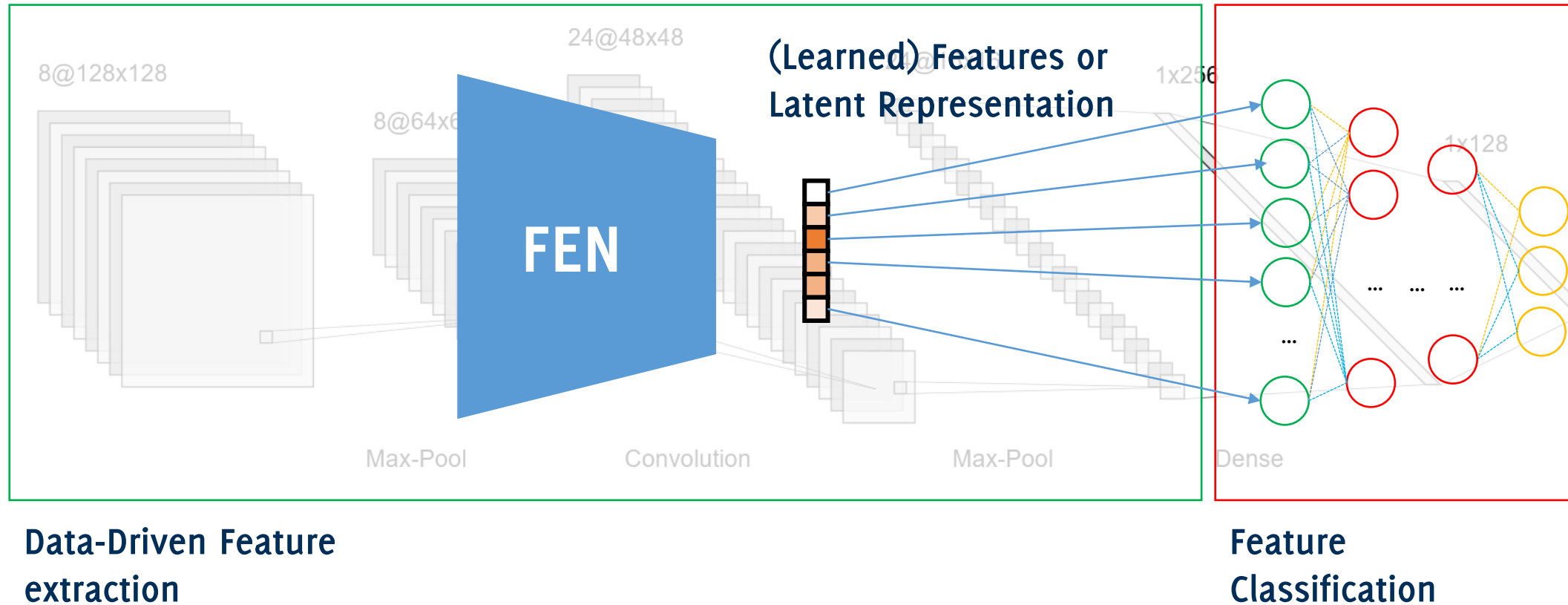


The typical architecture of a CNN



FEN: FEATURE EXTRACTION NETWORK, the convolutional block of CNN

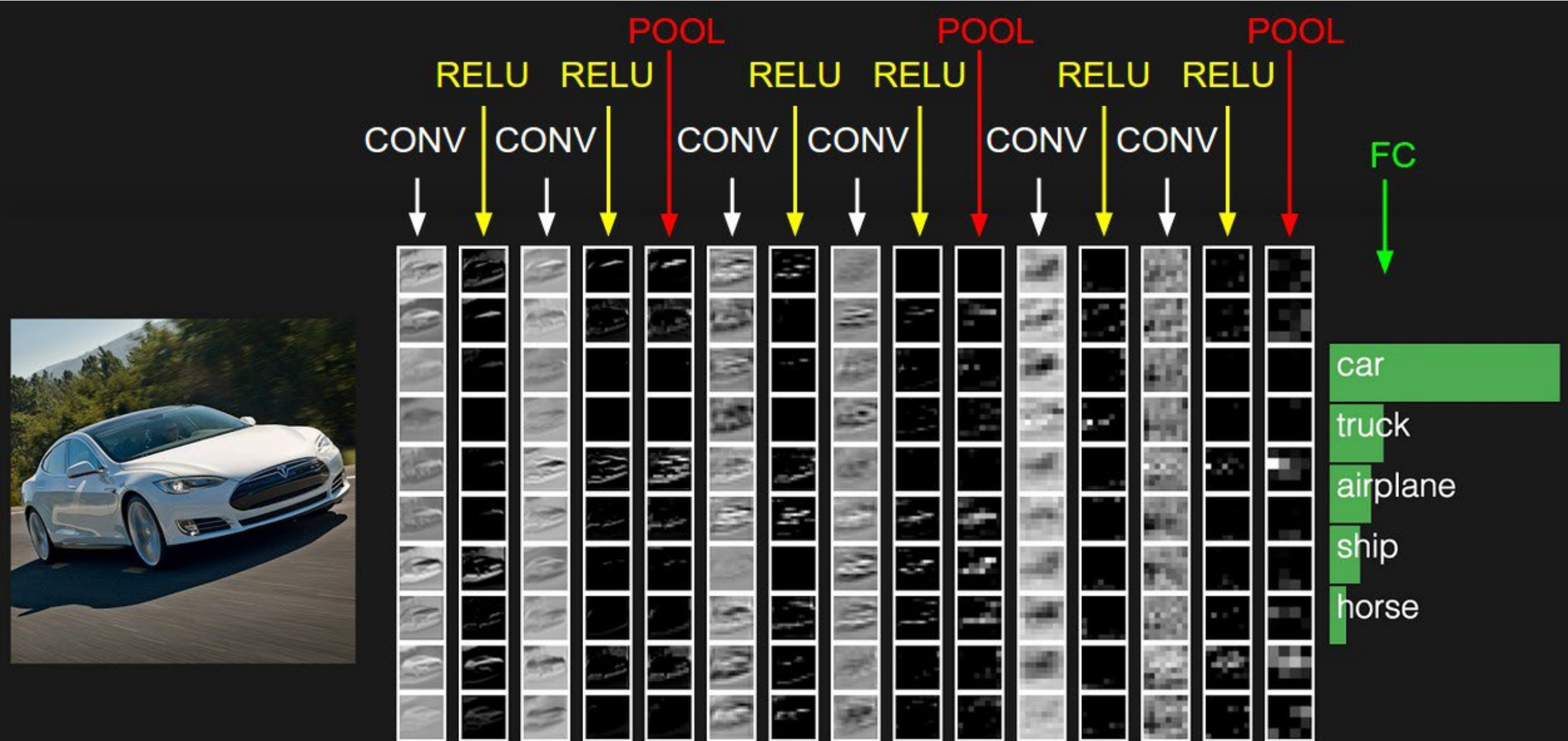
The typical architecture of a CNN



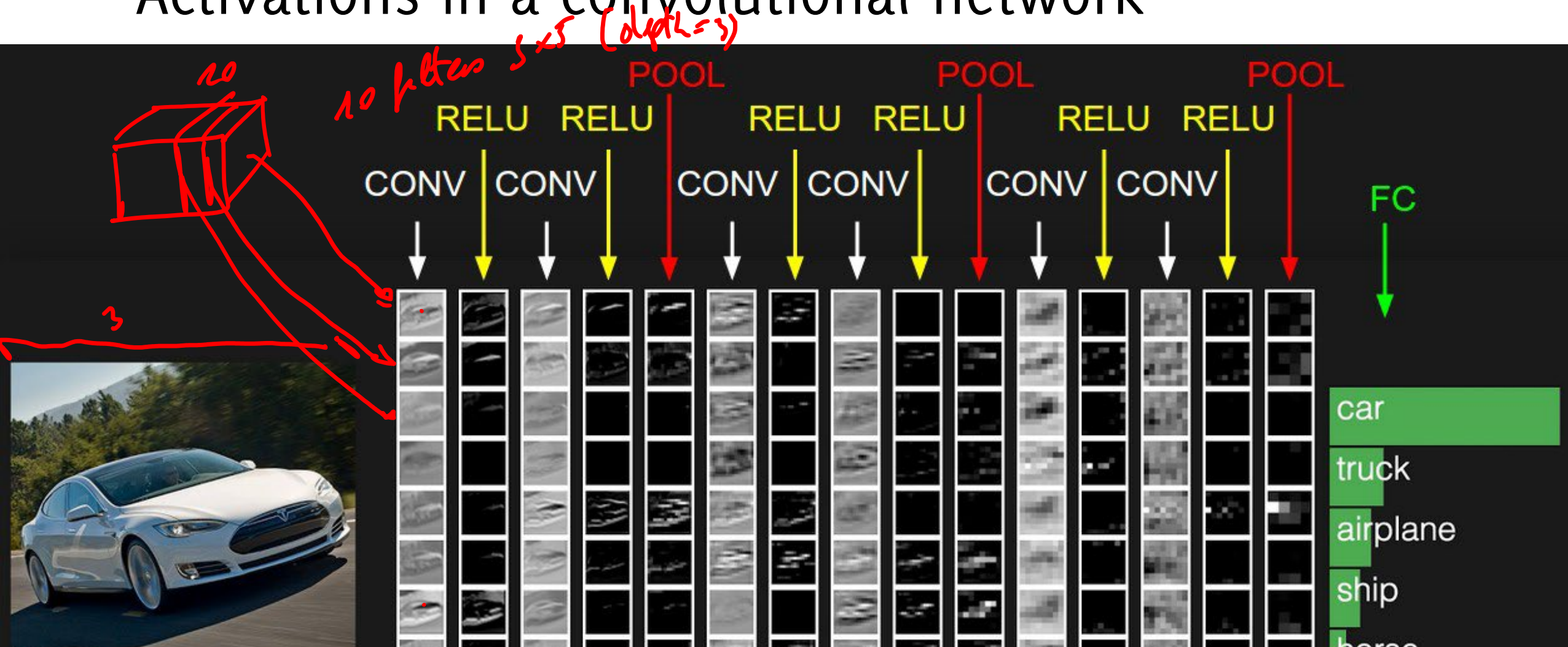
Typically, to learn meaningful representations, many layers are required
The network becomes deep

CNN «in action»

Activations in a convolutional network

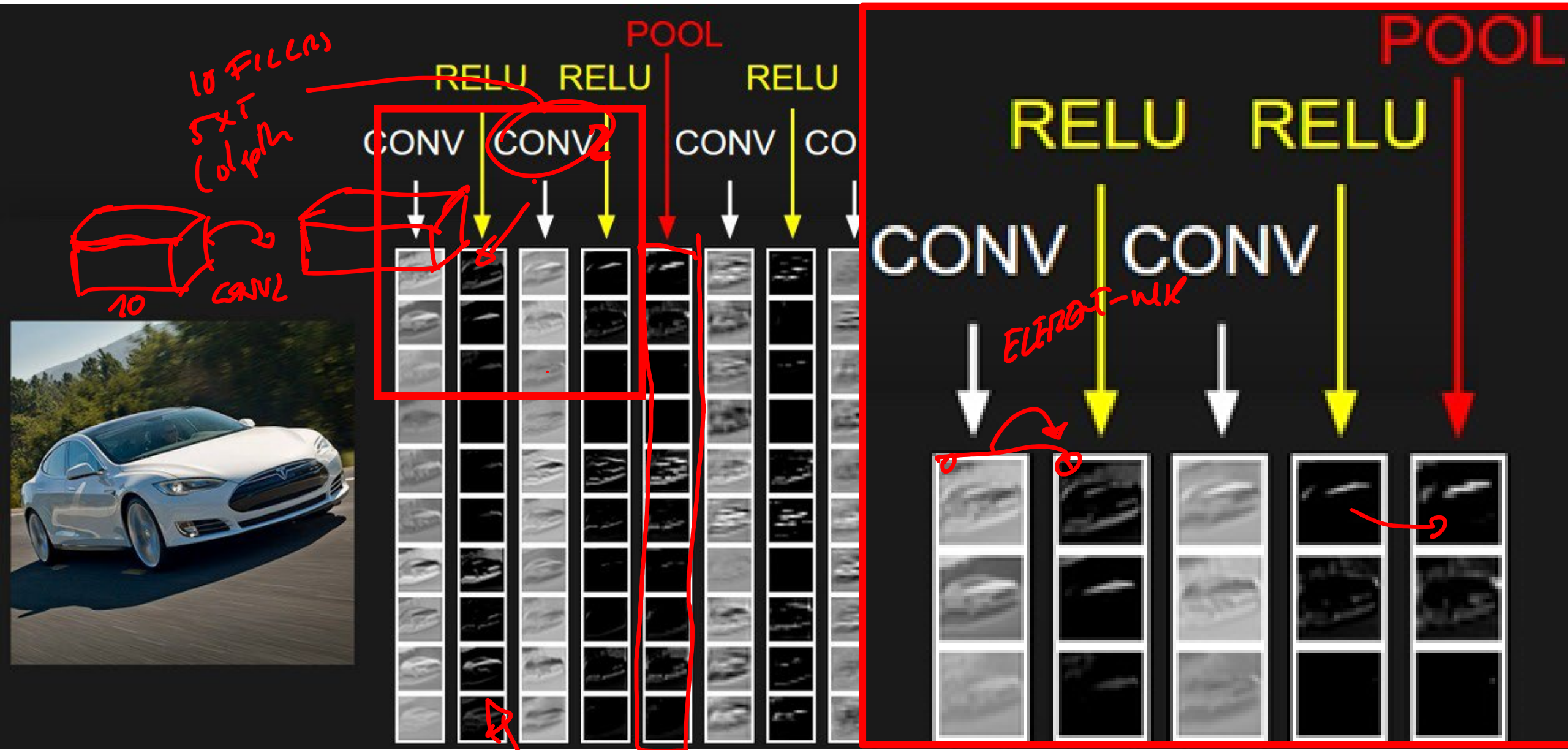


Activations in a convolutional network

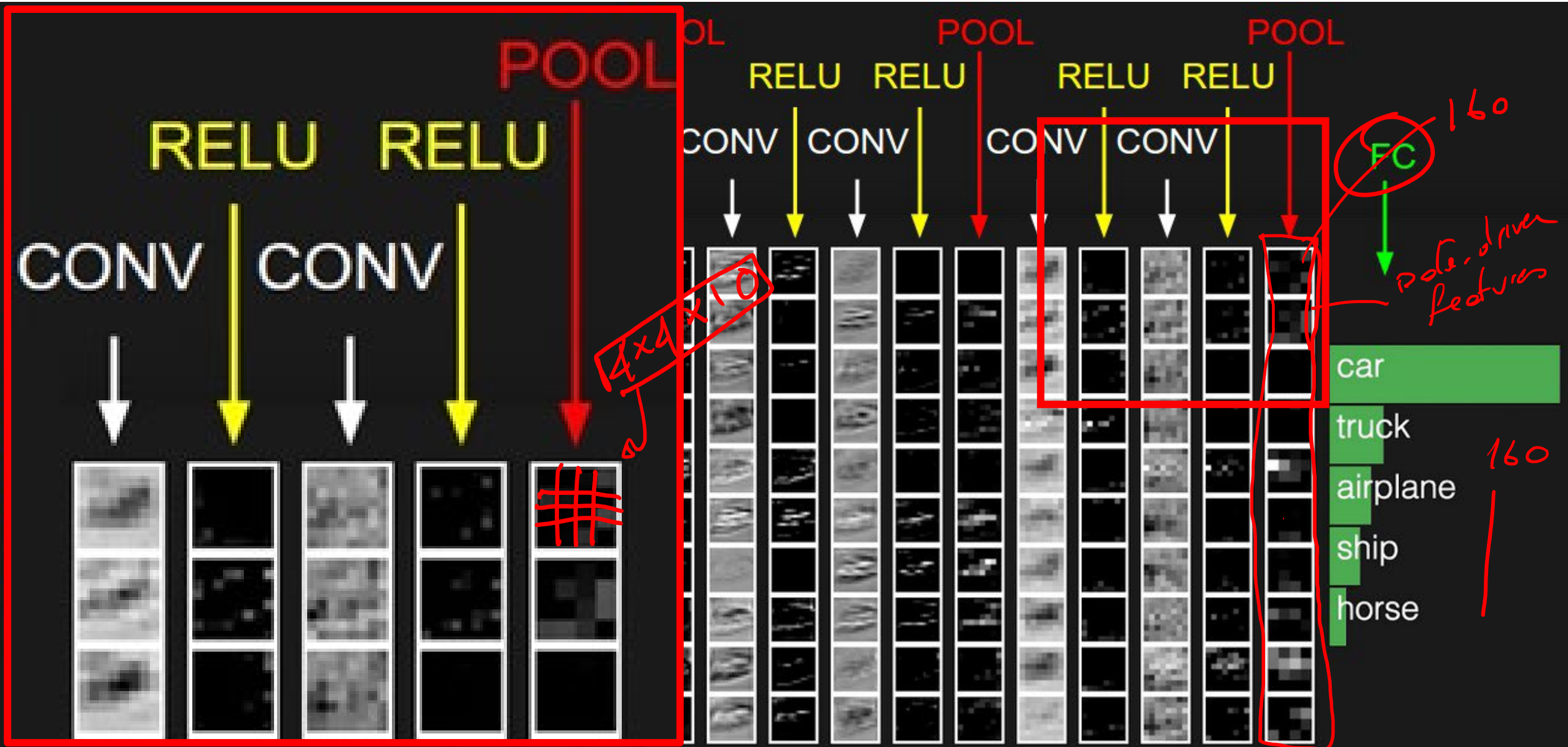


Each layer in the volume is represented as an image here (using the same size but different resolution for visualization sake)

Activations in a convolutional network

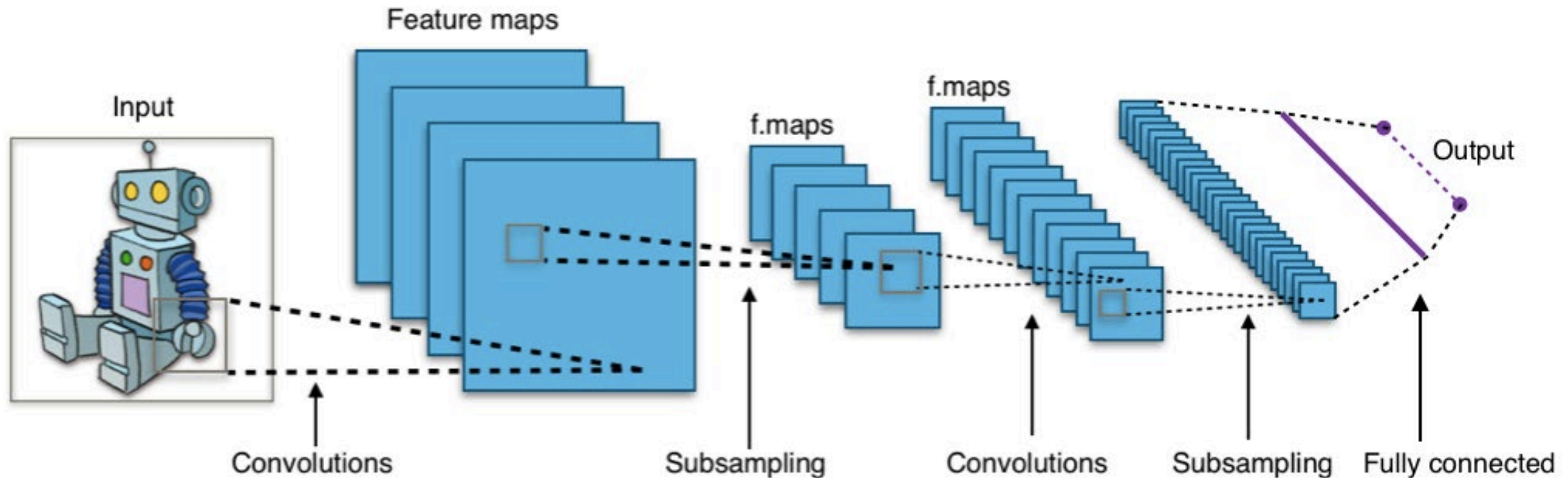


Activations in a convolutional network



Convolutional Neural Networks (CNN)

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



The First CNN

Gradient-Based Learning Applied to Document Recognition

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

Abstract—

Multilayer Neural Networks trained with the backpropagation algorithm constitute the best example of a successful Gradient-Based Learning technique. Given an appropriate network architecture, Gradient-Based Learning algorithms can be used to synthesize a complex decision surface that can classify high-dimensional patterns such as handwritten characters, with minimal preprocessing. This paper reviews various methods applied to handwritten character recognition and compares them on a standard handwritten digit recognition task. Convolutional Neural Networks, that are specifically designed to deal with the variability of 2D shapes, are shown to outperform all other techniques.

I. INTRODUCTION

Over the last several years, machine learning techniques, particularly when applied to neural networks, have played an increasingly important role in the design of pattern recognition systems. In fact, it could be argued that the availability of learning techniques has been a crucial factor in the recent success of pattern recognition applications such as continuous speech recognition and handwriting recognition.

[Home](#) > [Latest Awards News](#) > [2018 Turing Award](#)

Fathers of the Deep Learning Revolution Receive ACM A.M. Turing Award

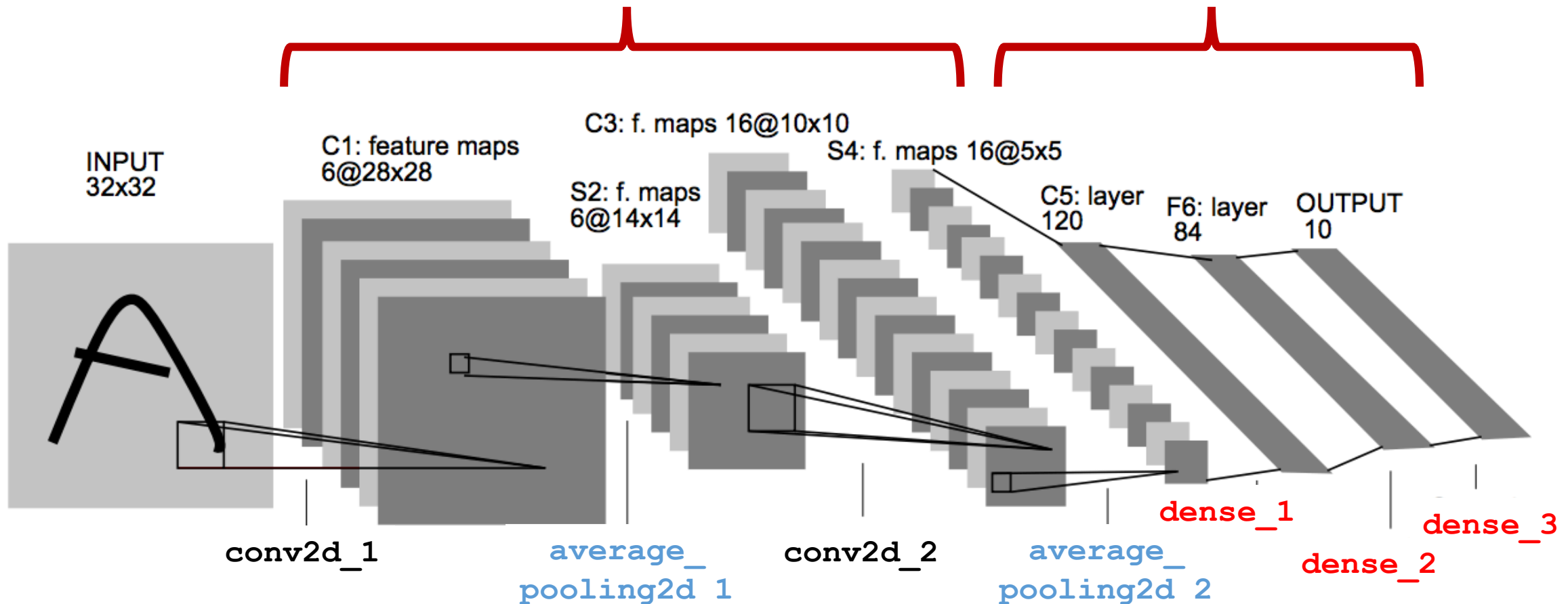
Bengio, Hinton and LeCun Ushered in Major Breakthroughs in Artificial Intelligence

<https://awards.acm.org/about/2018-turing>

LeNet-5 (1998)

Stack of Conv2D + RELU + AVG-POOLING

A TRADITIONAL MLP



LeCun, Yann, et al. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86.11 (1998)

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)	$6 \times (5 \times 5 \times 1) + 6 = 156$
average_pooling2d_1 (Average)	(None, 14, 14, 6)	0
conv2d_2 (Conv2D)	(None, 10, 10, 16)	$16 \times (3 \times 3 \times 6) + 16 = 304$
average_pooling2d_2 (Average)	(None, 5, 5, 16)	0
flatten_1 (Flatten)	(None, 400)	0
dense_1 (Dense)	(None, 120)	$400 \times 120 + 120 = 48120$
dense_2 (Dense)	(None, 84)	$120 \times 84 + 84 = 10164$
dense_3 (Dense)	(None, 10)	$84 \times 10 + 10 = 841$
Total params: 61,706		
Trainable params: 61,706		
Non-trainable params: 0		

INPUT $32 \times 32 \times 1$

FILTER SIZE 5×5



$$6 \times (5 \times 5 \times 1) + 6 = 156$$

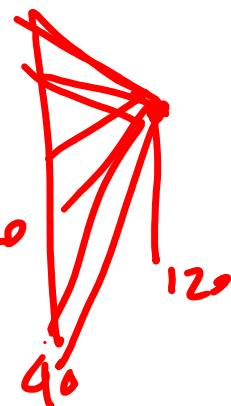
$$16 \times (3 \times 3 \times 6) + 16 = 304$$

$$5 \times 5 \times 16 = 400$$

$$400 \times 120 + 120 = 48120$$

$$120 \times 84 + 84 = 10164$$

$$84 \times 10 + 10 = 841$$



model.summary()

Layer (type)	Output Shape	Param #	
conv2d_1 (Conv2D)	(None, 28, 28, 6)	156 (6 x 5 x 5 + 6)	Input is a grayscale image
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)	The input is a volume having depth = 6
average_pooling2d_2 (Average)	(None, 5, 5, 16)	0	
flatten_1 (Flatten)	(None, 400)	0	Most parameters are still in the MLP
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			

model.summary()

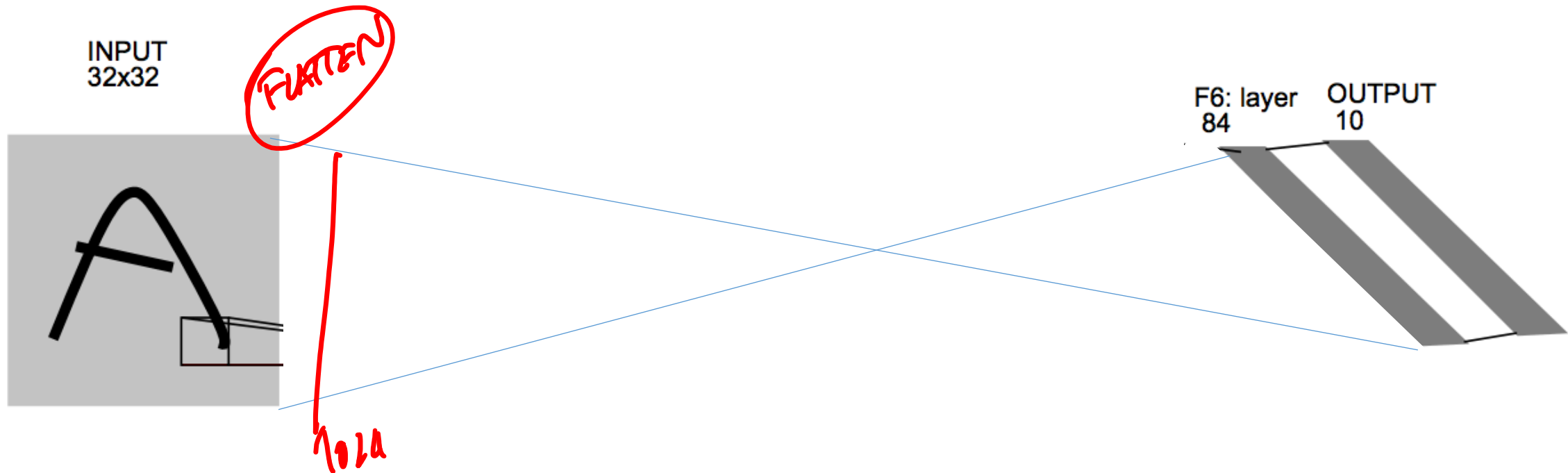
Layer (type)	Output Shape	Param #	
conv2d_1 (Conv2D)	(None, 28, 28, 6)	156 (6 x 5 x 5 + 6)	Input is a grayscale image
average_pooling2d_1 (Average)	(None, 14, 14, 6)	0	
conv2d_2 (Conv2D)	(None, 10, 10, 16)	2416 (16 x 5 x 5 x 6 + 16)	The input is a volume having depth = 6
average_pooling2d_2 (Average)	(None, 5, 5, 16)	0	
flatten_1 (Flatten)	(None, 400)	0	Most parameters are still in the MLP
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			

Here, no-padding at the first layer is necessary to reduce the size of the latent representation... and has no loss of information since images are black there!

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: $1024 * 84 + 84 + 84 * 10 + 10$



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: **only** the first layer increases the # of parameters by a factor 3

$$1024 \times 84 \rightarrow 1024 \times 84 \times 3$$

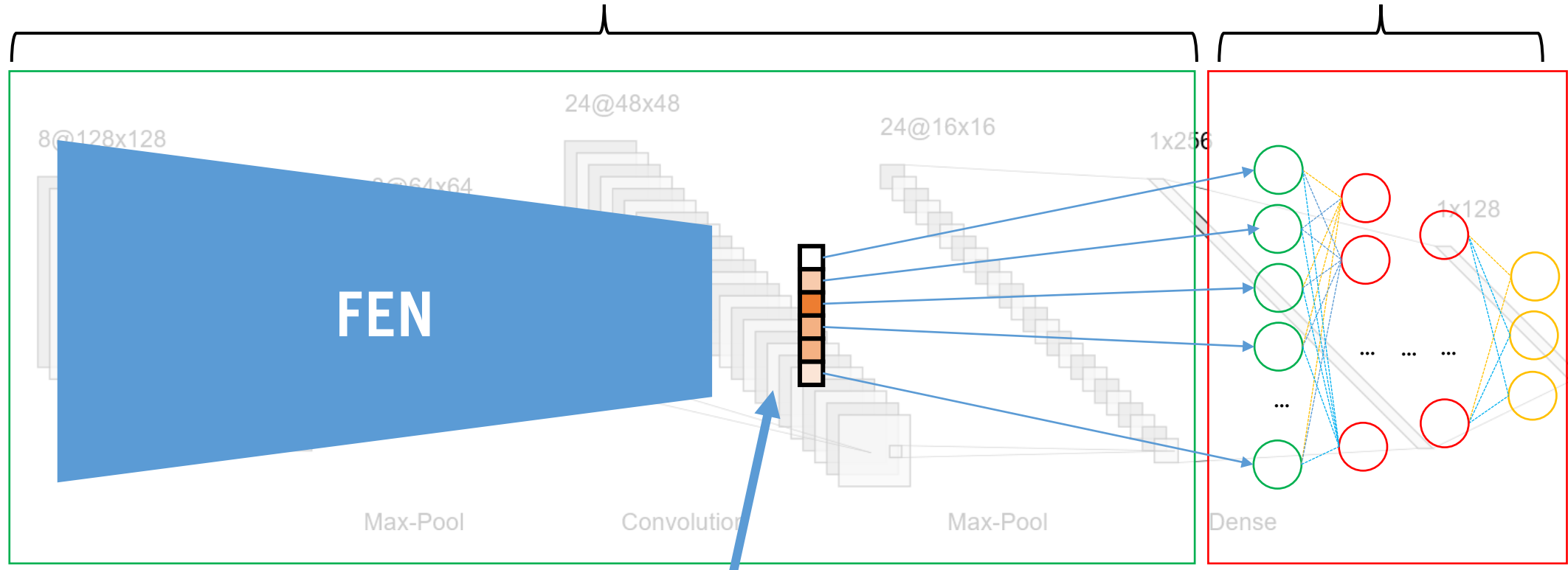
Latent representation in CNNs

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

The typical architecture of a CNN

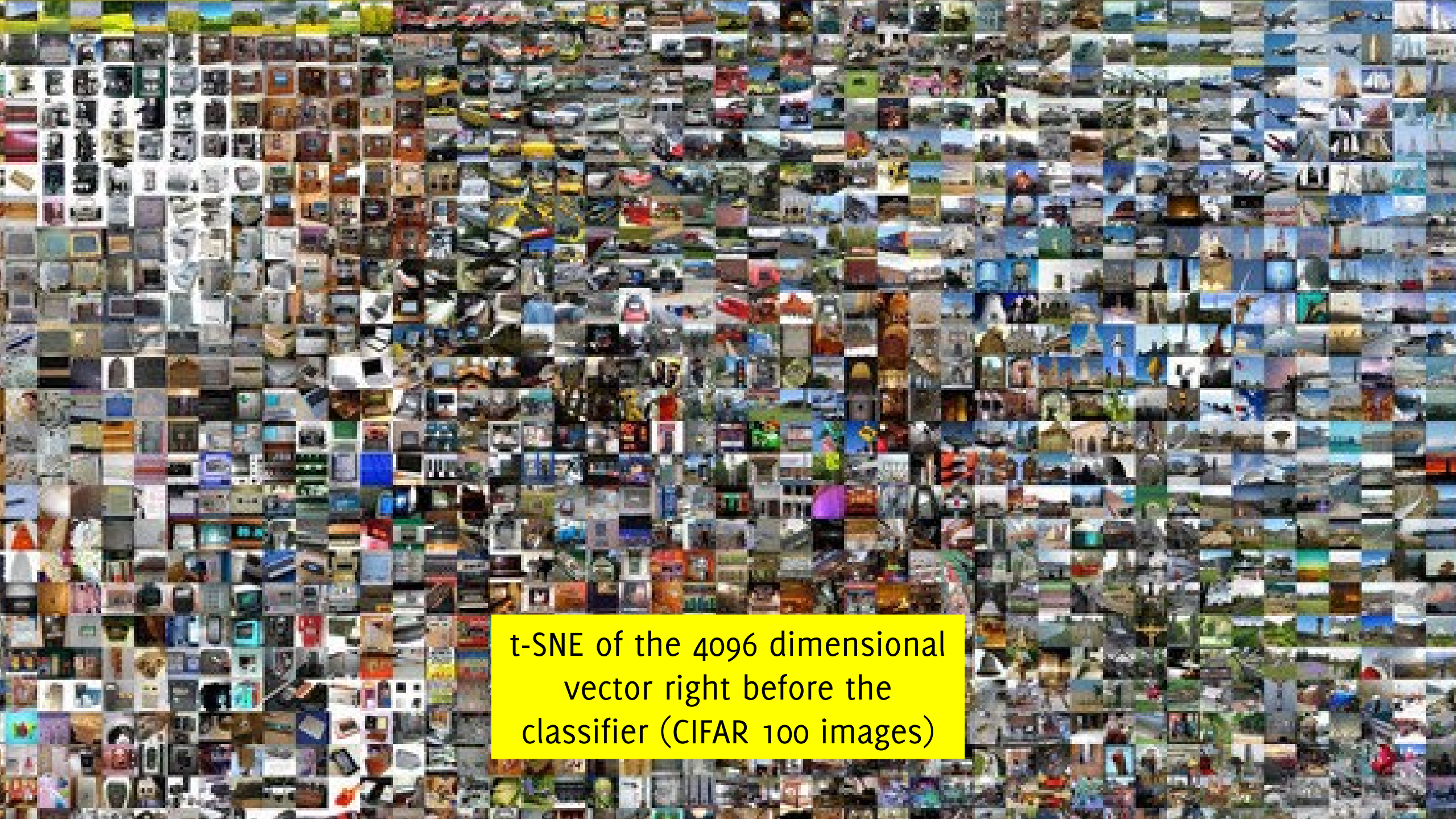
Convolutional Layers
Extract high-level features from pixels

Classify

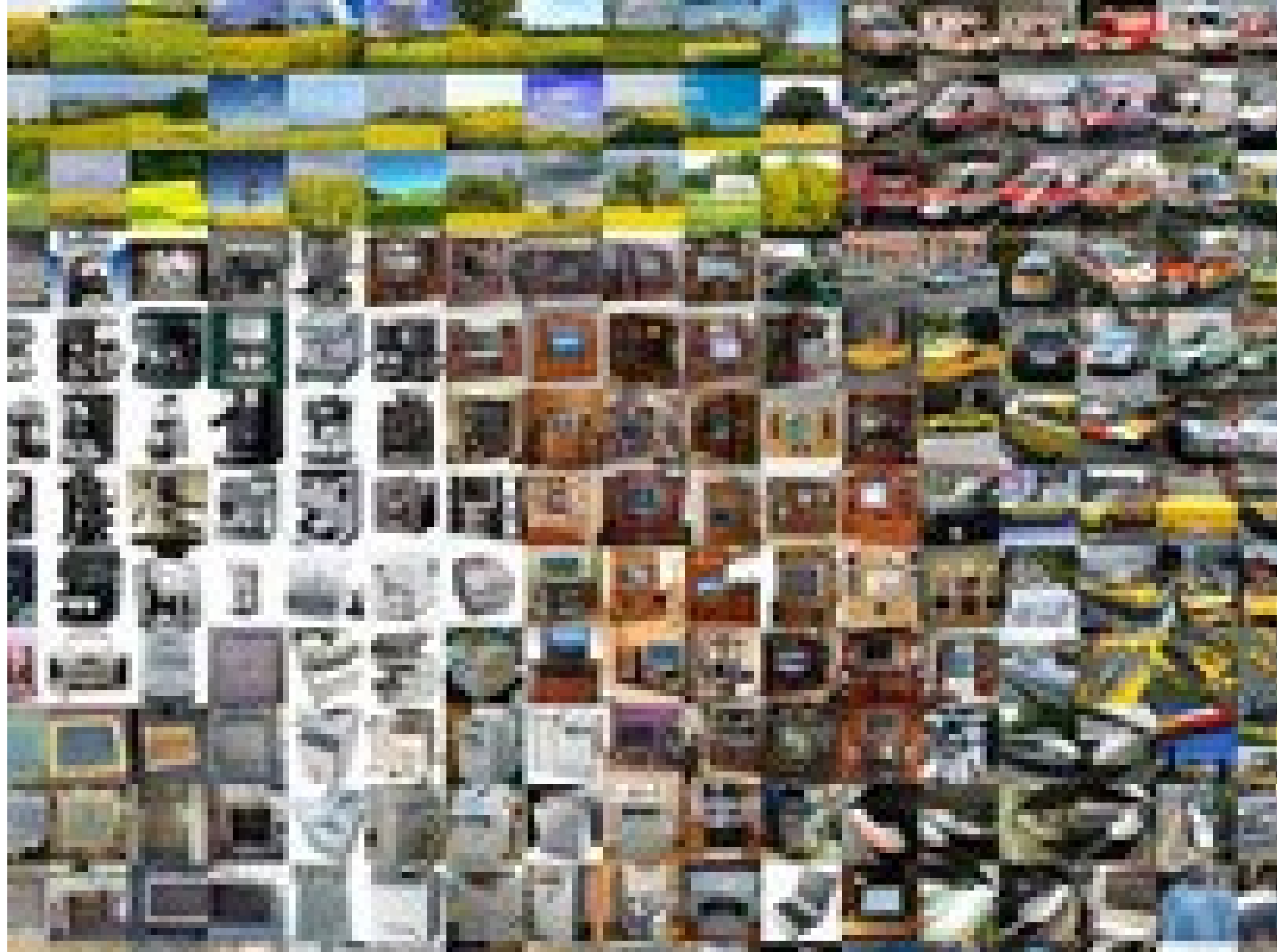


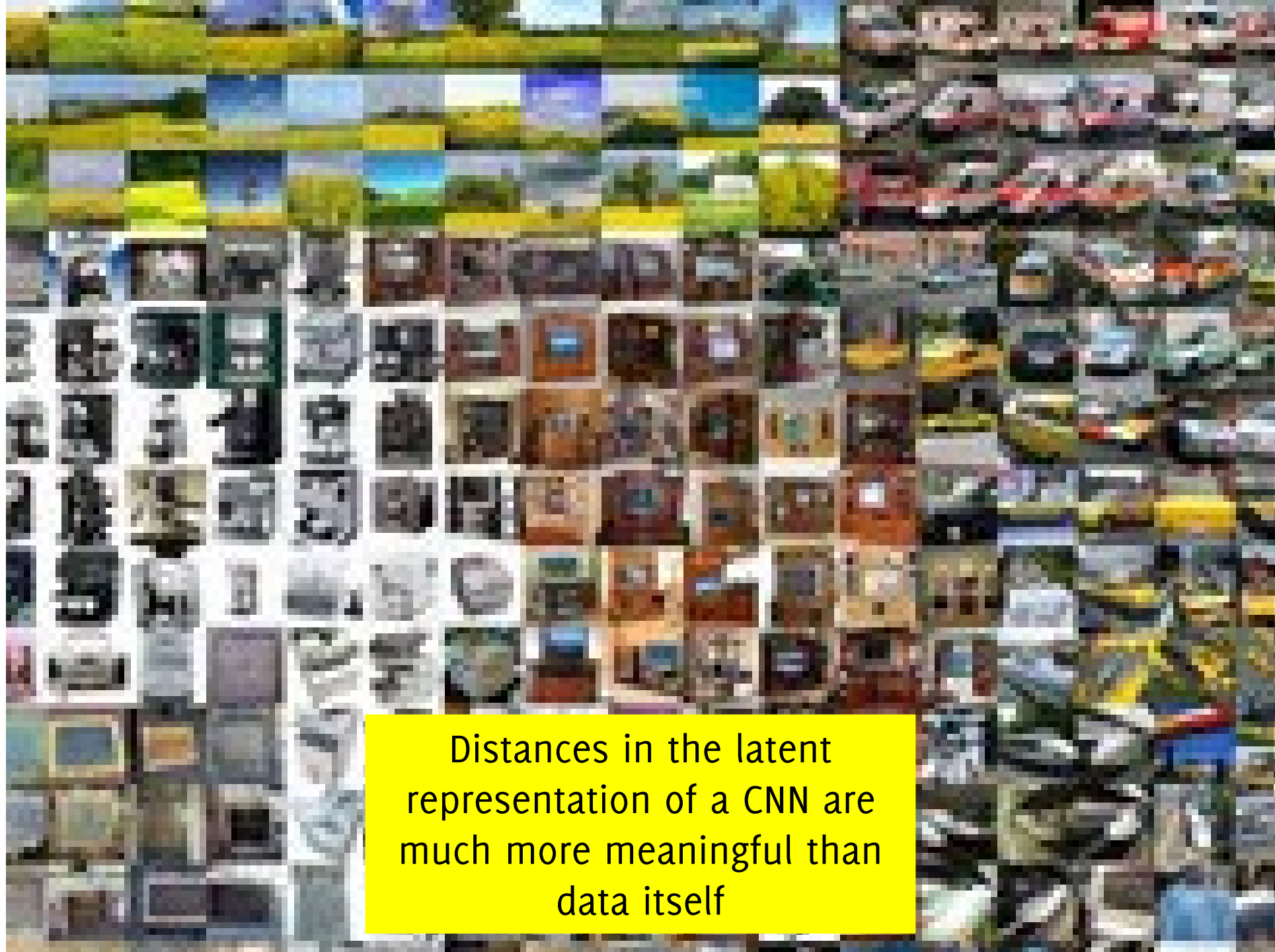
Latent Representation:
Data-Driven Feature Vector

MLP for feature
classification



t-SNE of the 4096 dimensional
vector right before the
classifier (CIFAR 100 images)





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

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**

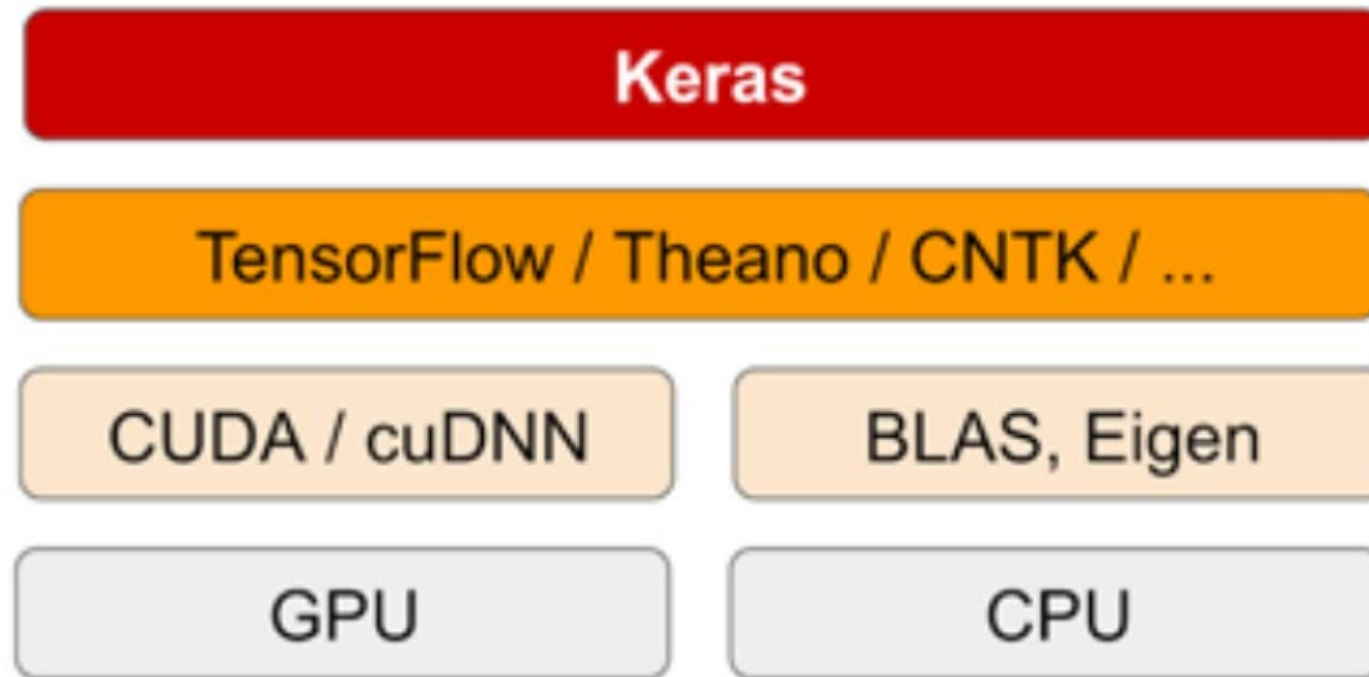
Doesn't handle low-level operations such as tensor manipulation and differentiation.

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:

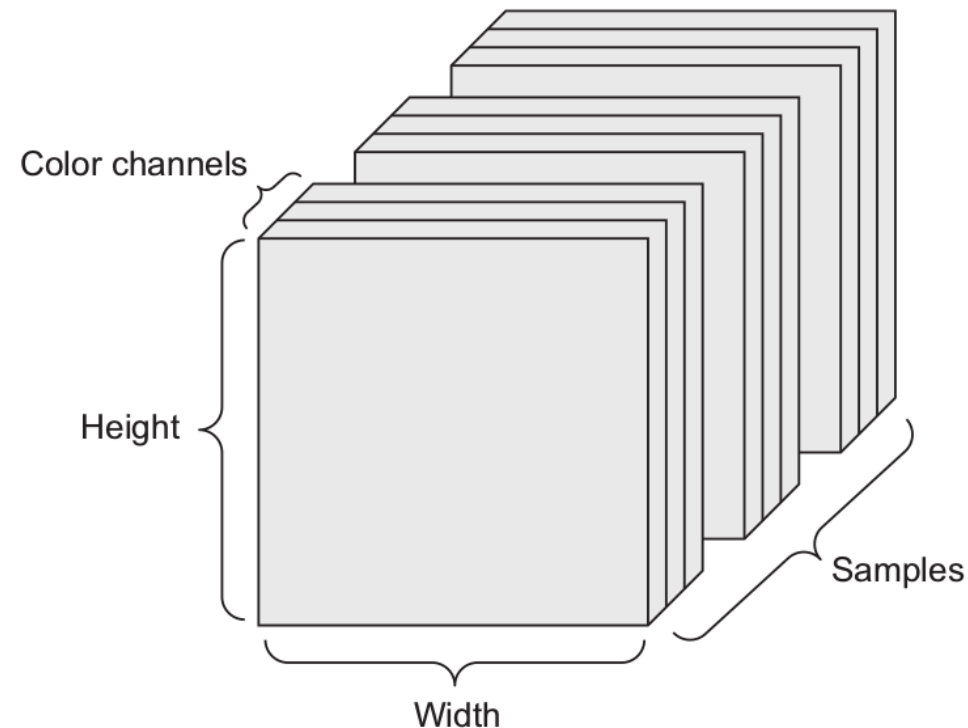
Not as flexible

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 $\neq 1$ is incompatible with specifying any `dilation_rate` value $\neq 1$.

padding: one of "valid" or "same" (case-insensitive). Note that "same" is slightly inconsistent across backends with strides $\neq 1$, 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 $\neq 1$ is incompatible with specifying any stride value $\neq 1$.

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).

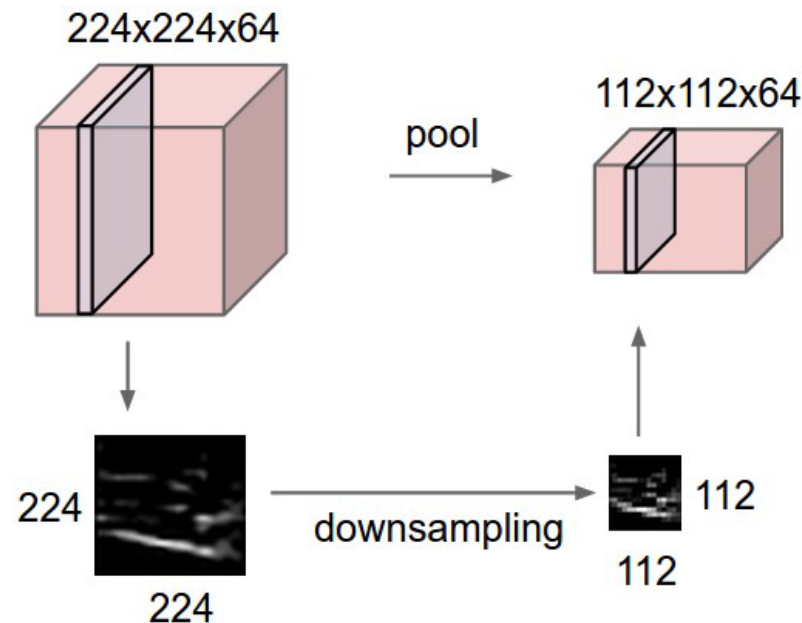
bias_constraint: Constraint function applied to the bias vector (see constraints).

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
<hr/>		
flatten_3 (Flatten)	(None, 43264)	0
<hr/>		
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