

# [Tutorial] Convolutional Neural Networks

## CNN Introduction

We will learn today about one of the most famous kind of Deep Learning method: the Convolutional Neural Networks (sometimes called ConvNets or CNN). But before, we need to talk a bit about image processing, in order to introduce necessary tools.

## I. Computer Vision introduction

Images are just numbers! So all we learnt about Machine Learning and Deep Learning has to work with images, both classification and regression!



Input Image



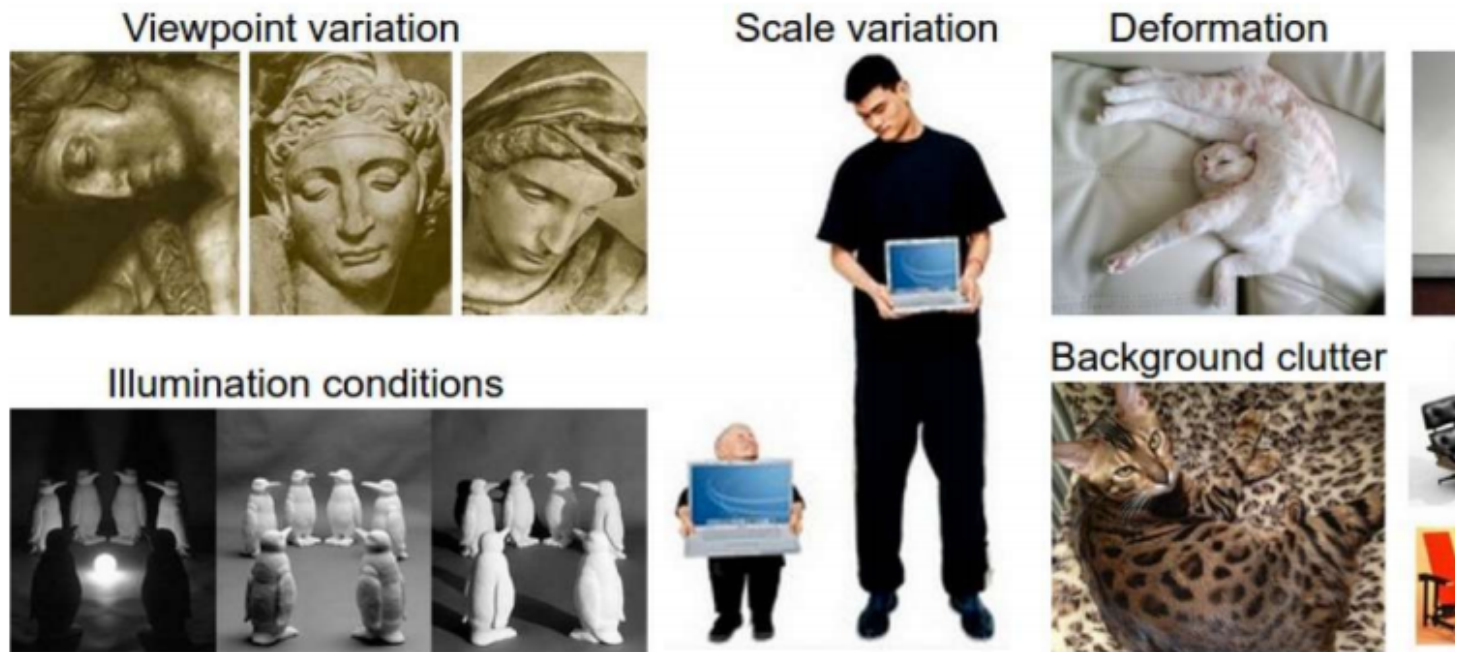
157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Pixel Representation

classification

- **Regression:** output variable takes continuous value
- **Classification:** output variable takes class label. Can produce probability

But unlike other kind of data, images have **spatial structure**! Indeed, the same object can be seen from various viewpoints, deformed, scaled, occluded, etc...



That's why in order to take full advantage of this spatial structure, we need to **add spatial information in our models**. This is where Convolutional Neural Networks (CNN) take place! So first, let's talk a bit about image processing.

---

## II. Image Processing

### II.1. Padding

Padding an image is adding borders to it, thus increasing the size of the image. Most of the time, images are padded with zeros.

Below is an example of a 4x4 image before padding, and after padding:

35	19	25	6
13	22	16	53
4	3	7	10
9	8	1	3

Original image



0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Padded image

Padding can also add more than one line of zeros.

## II.2. Kernel

A kernel in image processing is nothing more than a small image (or a small matrix), that we will use to make a convolution.

Thus, a kernel is necessary smaller than the image. Usually, kernels have odd dimensions, and are quite small. Typical sizes of kernels are:

- 3x3
- 5x5
- 7x7
- 9x9

Depending on the values of the kernel, the convolution will provide different results.

## II.3 Convolution

A convolution is a mathematical operation that you already encountered. For example, below is the convolution result of an image with a 3x3 kernel of ones:

1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0
0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved  
Feature

But as you know, the kernel can have different values. Below is an example of a convolution of an image I with a non uniform kernel K:

0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	1	1	0	0	0	0
1	1	0	0	0	0	0

**I**

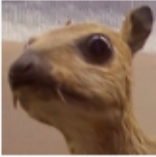



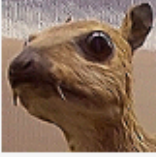
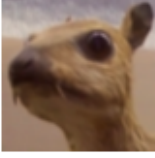
1	0	1
0	1	0
1	0	1

**K**

**I \* K**

1	4	3	4	1
1	2	4	3	3
1	2	3	4	1
1	3	3	1	1
3	3	1	1	0

A wisely chosen kernel can be really useful to detect patterns in images. Here is a list of commonly used kernels and what they do on a given image (from [wikipedia](https://en.wikipedia.org/wiki/Kernel_(image_processing)) ([https://en.wikipedia.org/wiki/Kernel\\_\(image\\_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing)))):

Kernel	Usage	Example
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	Identity	
$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	Edge detection	
$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	Identity	
$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	Edge detection	
$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	Sharpening	
$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	Gaussian blur	

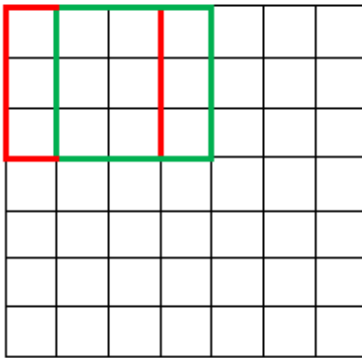
So to summarize, a convolution is a mathematical operation on a image with a given kernel, resulting in a new processed image.

## II.4. Strides

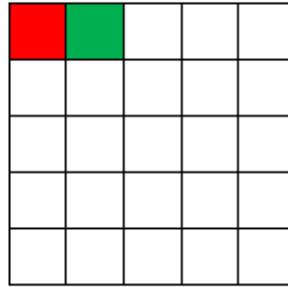
Up to now, we know how to compute a convolution, with a given kernel, and to pad an image. The last tool needed is what is called stride.

The stride value is the step between two convolutions with a kernel. For the moment, all we did was with a stride of one, as in the example below:

7 x 7 Input Volume



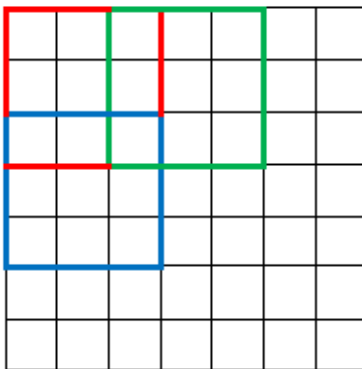
5 x 5 Output Volume



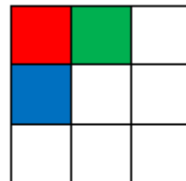
So as we see in this example, an image of dimension 7x7, convoluted with a kernel of 3x3 results in an image of 5x5.

What if we use a stride of 2, meaning our kernel with use a step of 2 on our image:

7 x 7 Input Volume



3 x 3 Output Volume



With a stride of 2, an input image of dimension 7x7 convoluted with a kernel of 3x3 results in an image of 3x3.

## III. Convolutional Neural Networks

### III.1. Introduction

Why would we use convolutions in neural networks? Isn't a MLP complicated enough?

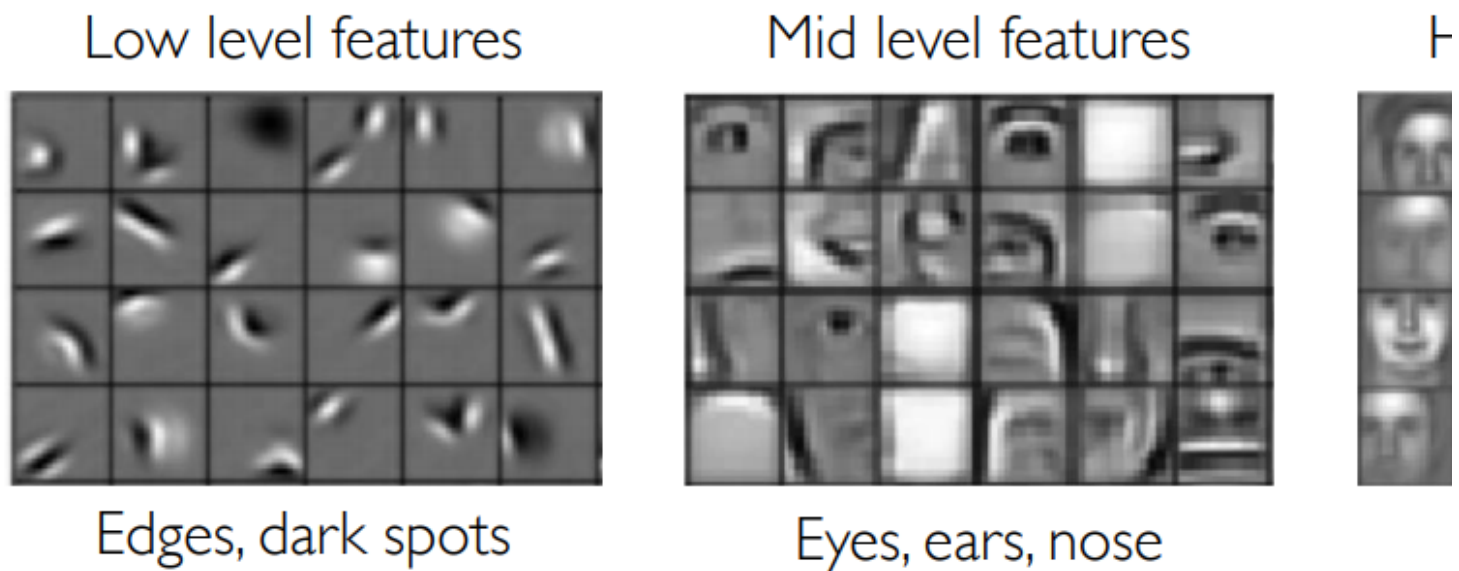
Well, indeed MLP are complicated enough, but they do not take into account *spatial* correlations. Indeed in an image, there are spatial correlations everywhere:

- In a face the eye is always close to the nose, and the nose is always in the middle of the face.
- A bike always has two wheels.
- All numbers and letters have a particular shape.

For all of those examples, a classical MLP will not see the spatial correlations, and might have to learn them by itself. But if you add convolutions, this is far more easier to understand the structure of the data for the neural network.

Historically, convolutional neural networks were not always famous. Yann Le Cun, commonly admitted as one of the creators of this technique, is now a rock star in AI, but had rough years in academic research in the past.

CNNs exploded with [AlexNet](https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf) [\\_ \(https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf\)](https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf) in 2012, a CNN that made a breakthrough in the Computer Vision field. Since then, CNNs are quite popular and widely used in the Computer Vision are.



## III.2. Convolutional Layer

To make a CNN, we need first to make a Convolutional Layer. A convolutional layer is simply a convolution on our input data, with several options to choose: the size of the kernel, the padding and the stride.

Here is the signature in TensorFlow:

```
tensorflow.keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid',  
data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True,  
kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None,  
bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

With especially the following parameters to use:

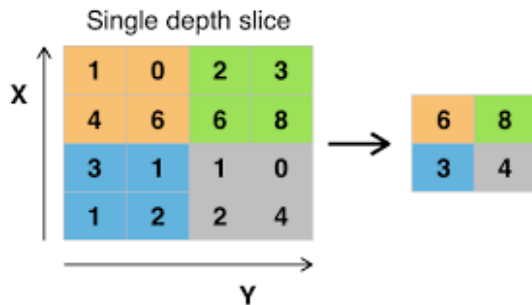
- `filters`: the number of kernels to use
- `kernel_size`: the size of the kernels (e.g. `(3,3)` for a 3x3 kernel)
- `padding`: `valid` for no padding, `same` to keep the same dimensions for the image
- `kernel_regularizer`: to add regularization

We will see how to use it in an example in the last section.

## III.3. Pooling Layer

A pooling layer is a layer that will reduce the size of the images by taking the average or max value of a given number of pixels.

Below is an example of pooling layer, with a kernel of 2x2 and a stride of 2 applied on an image of 4x4:



A pooling layer is usually used right after a convolutional layer.

Pooling layers are defined in Keras as well, with the following signature:

```
tensorflow.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid',  
data_format=None)
```

Where:

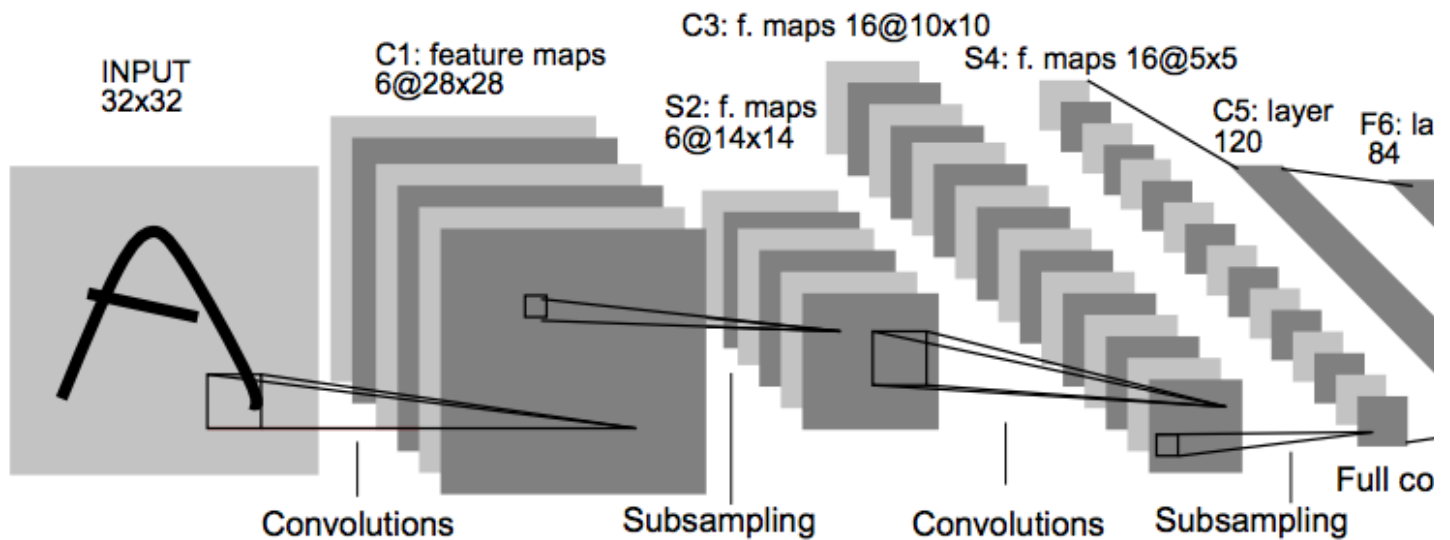
- `pool_size` is the dimensions of the kernel
- `strides` is the stride value, if `None` the value will be the same as `pool_size`
- `padding` can be `valid` for no padding or `same` for keeping image dimension

## IV. LeNet-5 on MNIST digits

### IV.1. Architecture diagram

We will first see the architecture of the LeNet-5: this is the algorithm that was proposed by [LeCun et al.](http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf) (<http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf>) for digit classification using convolutional networks.





We will now decompose and explain the layers step by step:

- Input (32x32): this is the input image of a digit
- C1 (6@28x28): this is a convolutional layer of 3x3 with 6 filters
- S2 (6@14x14): this is a max pooling layer of 2x2
- C3 (16@10x10): this is a convolution layer of 3x3 with 10 filters
- S4 (16@5x5): this is a max pooling layer of 2x2
- C5 (120): this is a fully connected layer (MLP) of 120 units
- F6 (84): this is a fully connected layer (MLP) of 84 units
- Output (10): this is a softmax layer of 10 units (1 unit per digit)

## IV.2. Implementation

The implementation of the LeNet-5 algorithm would be the following in Keras:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import MaxPooling2D, Conv2D, Flatten, Dense

def lenet5():

    model = Sequential()

    # Layer C1
    model.add(Conv2D(filters=6, kernel_size=(3, 3), activation='relu', input_shape=(28,28,1)))
    # Layer S2
    model.add(MaxPooling2D(pool_size=(2, 2)))
    # Layer C3
    model.add(Conv2D(filters=16, kernel_size=(3, 3), activation='relu'))
    # Layer S4
    model.add(MaxPooling2D(pool_size=(2, 2)))
    # Before going into layer C5, we flatten our units
    model.add(Flatten())
    # Layer C5
```

```

model.add(Dense(units=120, activation='relu'))
# Layer F6
model.add(Dense(units=84, activation='relu'))
# Output layer
model.add(Dense(units=10, activation = 'softmax'))

return model

```

## IV.3. Application to MNIST Digits

We will now apply this model to the MNIST digits classification problem.

```

from tensorflow.keras.datasets import mnist
# Import the dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

```

```

# Rescale the data
X_train = X_train/255.
X_test = X_test/255.

```

```

# Reshape
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], X_train.shape[2], 1)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], X_test.shape[2], 1)

```

```

from tensorflow.keras.utils import to_categorical
# Transform the targets to categorical vectors
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

```

```

from tensorflow.keras.callbacks import EarlyStopping, TensorBoard

# Instantiate the model
model = lenet5()

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

# Define the callbacks
callbacks = [EarlyStopping(monitor='val_loss', patience=5),
             TensorBoard(log_dir='./Graph', histogram_freq=0, write_graph=True, write_images=True)]

# Finally fit the model
model.fit(x=X_train, y=y_train, validation_data=(X_test, y_test), epochs=10, batch_size=64, callback
s=callbacks)

```

```

Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [=====] - 14s 226us/sample - loss: 0.2663 - accuracy: 0.9193 -
val_loss: 0.0904 - val_accuracy: 0.9721

```

```
Epoch 2/10
60000/60000 [=====] - 13s 223us/sample - loss: 0.0825 - accuracy: 0.9745 -
val_loss: 0.0602 - val_accuracy: 0.9794
Epoch 3/10
60000/60000 [=====] - 14s 235us/sample - loss: 0.0571 - accuracy: 0.9822 -
val_loss: 0.0505 - val_accuracy: 0.9841
Epoch 4/10
60000/60000 [=====] - 14s 237us/sample - loss: 0.0458 - accuracy: 0.9854 -
val_loss: 0.0472 - val_accuracy: 0.9845
Epoch 5/10
60000/60000 [=====] - 14s 230us/sample - loss: 0.0380 - accuracy: 0.9883 -
val_loss: 0.0385 - val_accuracy: 0.9879
Epoch 6/10
60000/60000 [=====] - 13s 215us/sample - loss: 0.0312 - accuracy: 0.9905 -
val_loss: 0.0369 - val_accuracy: 0.9871
Epoch 7/10
60000/60000 [=====] - 13s 222us/sample - loss: 0.0273 - accuracy: 0.9914 -
val_loss: 0.0390 - val_accuracy: 0.9877
Epoch 8/10
60000/60000 [=====] - 13s 218us/sample - loss: 0.0218 - accuracy: 0.9927 -
val_loss: 0.0367 - val_accuracy: 0.9888
Epoch 9/10
60000/60000 [=====] - 13s 217us/sample - loss: 0.0199 - accuracy: 0.9935 -
val_loss: 0.0424 - val_accuracy: 0.9874
Epoch 10/10
60000/60000 [=====] - 13s 210us/sample - loss: 0.0163 - accuracy: 0.9947 -
val_loss: 0.0406 - val_accuracy: 0.9872
```

```
# Compute the accuracy
print('accuracy on train with NN:', model.evaluate(X_train, y_train, verbose=0)[1])
print('accuracy on test with NN:', model.evaluate(X_test, y_test, verbose=0)[1])
```

```
accuracy on train with NN: 0.9953667
accuracy on test with NN: 0.9872
```

Our algorithm has about 99% accuracy. Amazing, right?!

## V. Transfer Learning

### V.1 Transfer Learning Principle

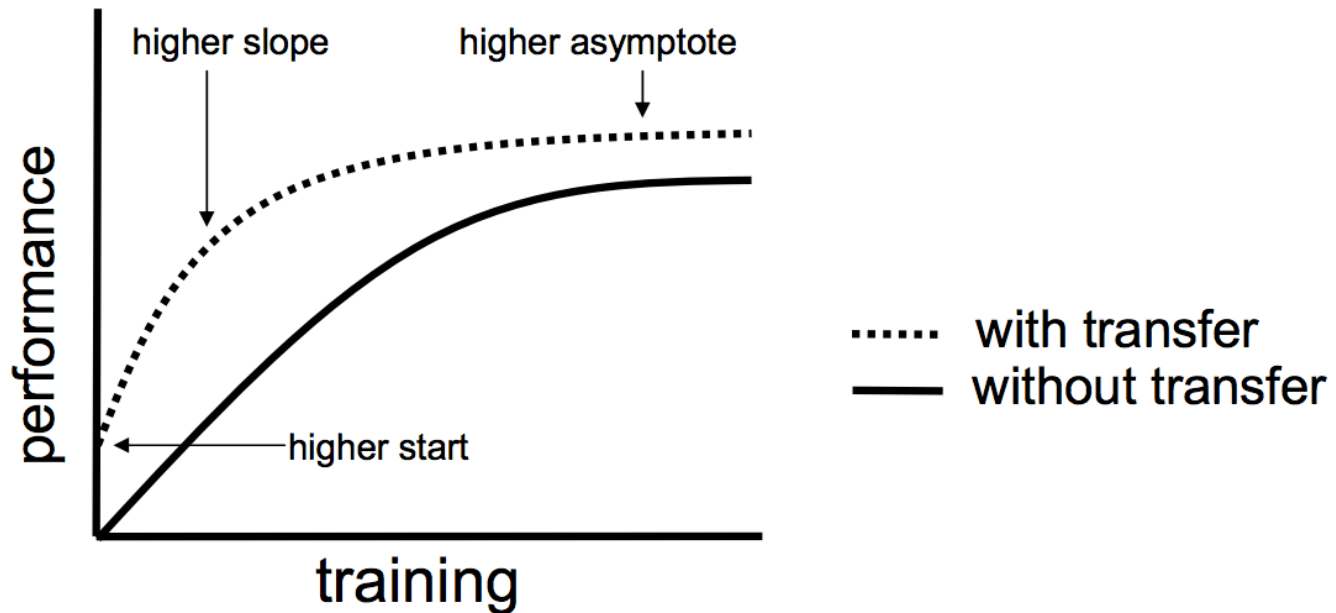
**Transfer Learning** is the ability of a model to reuse knowledge from a given problem and apply it to another problem.

For example, you want to train a model to classify cars, bus, trucks... on pictures, but you don't have many pictures of cars: how would you do that?

➡ You can reuse a model that was trained for autonomous driving, that is used to 'see' this kind of objects, and retrain it.

It works just like us humans, who can recognize almost any kind of cat easily, because we are already used to 'see' cats.

This is a really powerful method, that can lead to improved performance and/or fast training.



## V.2. Computer Vision: Famous Architectures

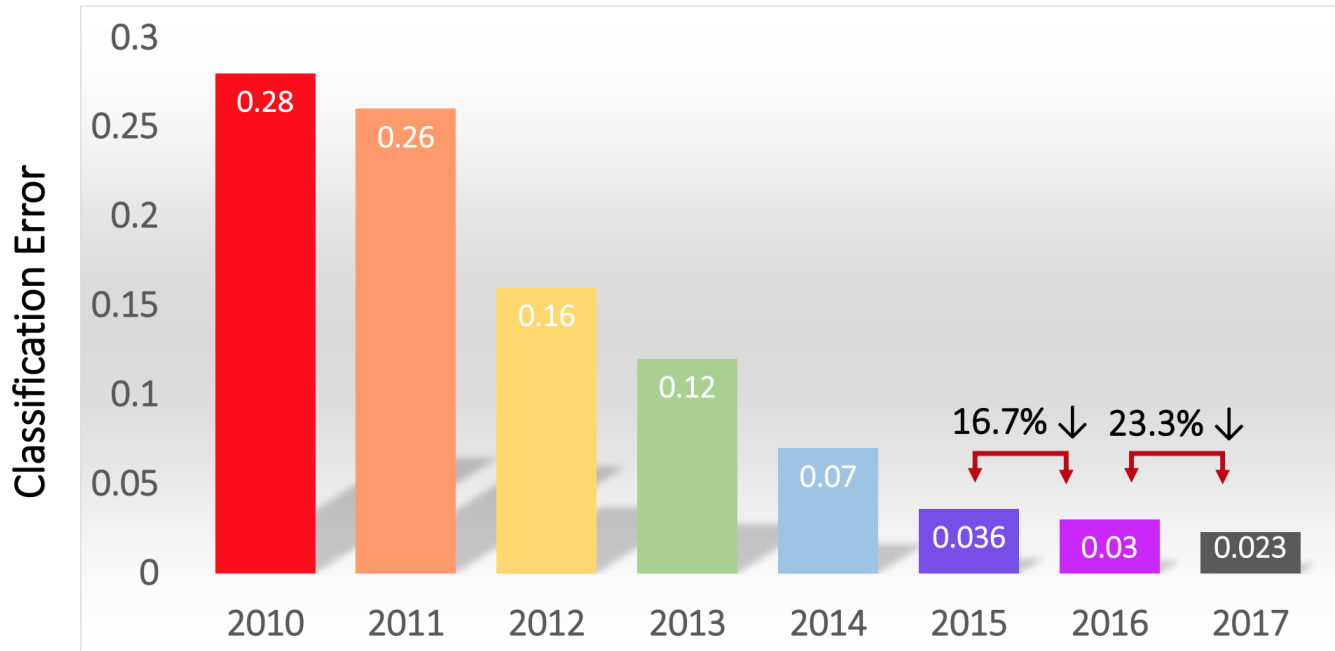
Historically, convolutional neural networks were not always famous. Yann Le Cun, commonly recognized as one of the creators of this technique, is now a rock star in AI, but had rough years in academic research in the past.

Now there are several architectures that have made history in the field of computer vision, let's see some of them.

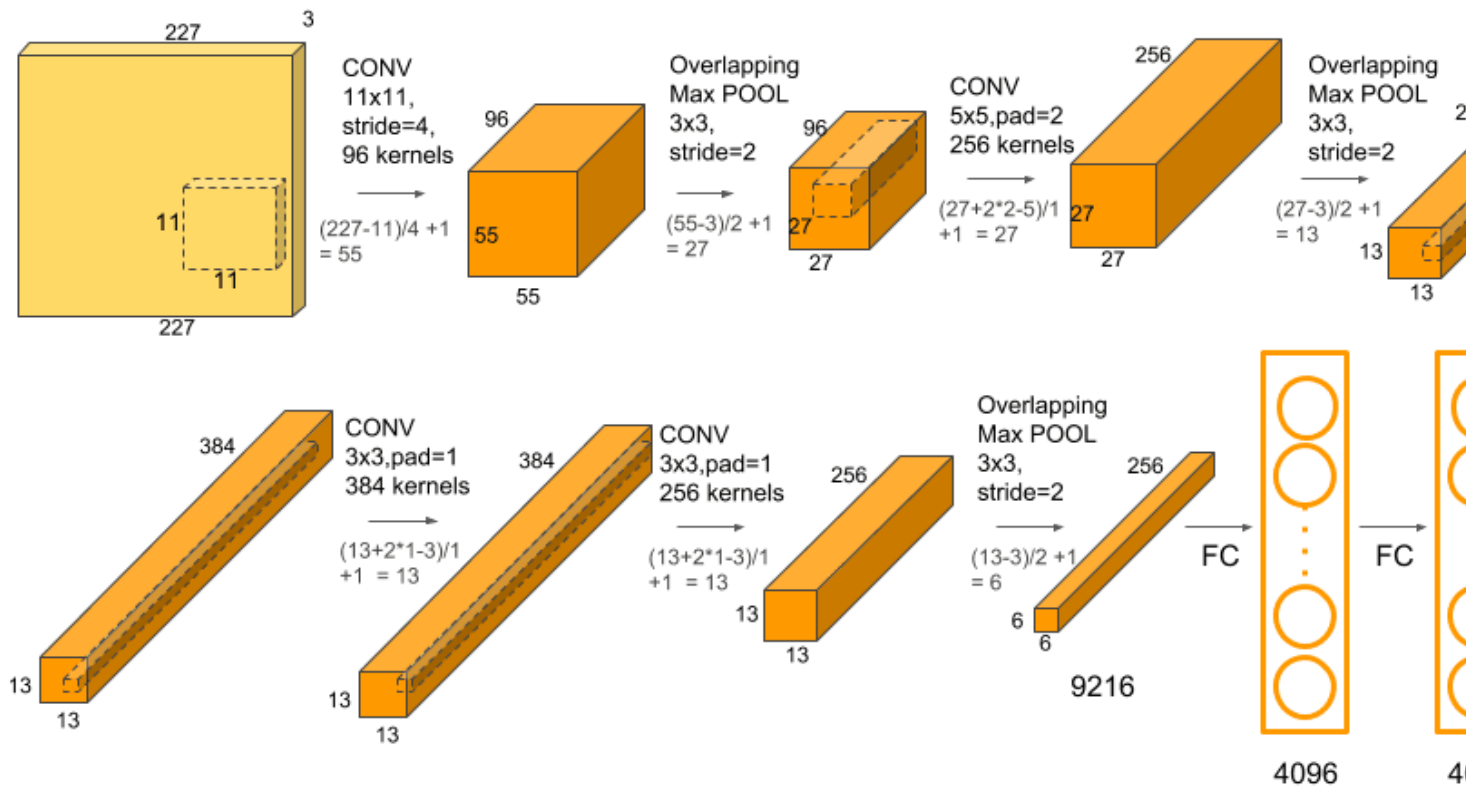
### AlexNet: the game changer

CNNs really exploded with [AlexNet](https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf) [\\_ \(https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf\)](https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf) in 2012, a CNN that made a breakthrough in the Computer Vision field.

# Classification Results (CLS)



The architecture of AlexNet is the following:



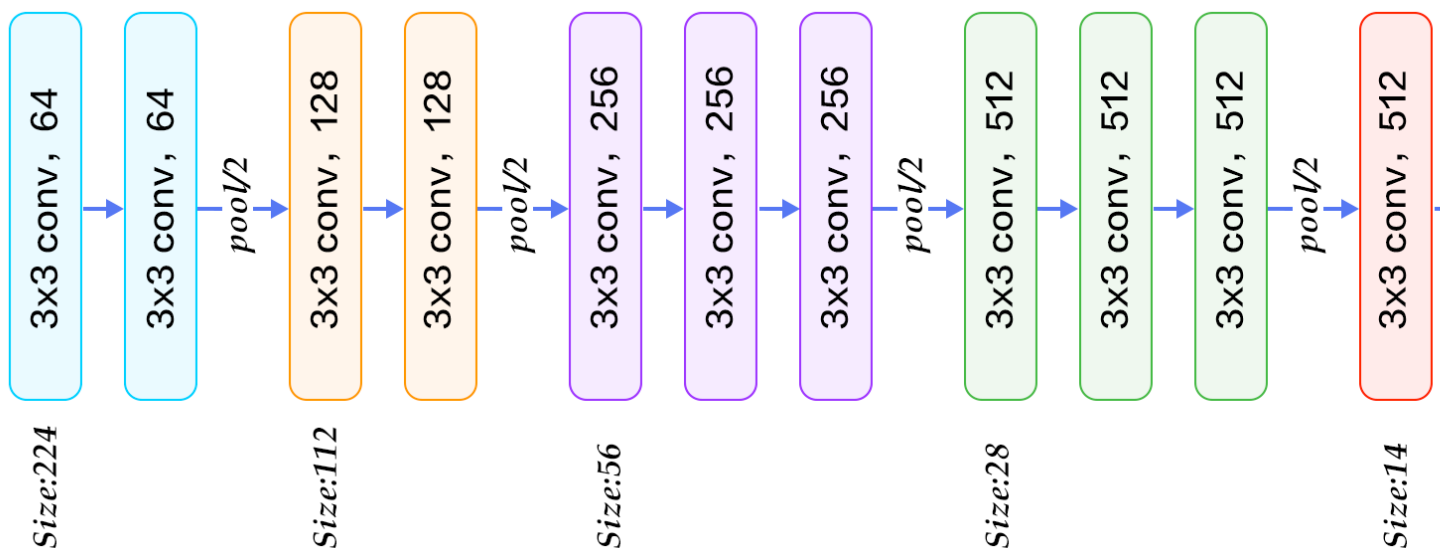
The architecture is the following:

- Convolutional layer of 11x11 with 96 filters and stride of 4
- Max pooling layer of 3x3 with stride of 2
- Convolutional layer of 5x5 with 256 filters
- Max pooling layer of 3x3 with stride of 2
- Convolutional layer of 3x3 with 384 filters
- Convolutional layer of 3x3 with 384 filters
- Convolutional layer of 3x3 with 256 filters
- Max pooling layer of 3x3 with stride of 2
- Fully connected layer of 4096 units
- Fully connected layer of 4096 units
- Output: fully connected softmax layer of 1000 units (1 units per class)

Since then, CNNs have become quite popular and are widely used in Computer Vision: many new powerful architectures have been developed. Let's see some of them.

## VGG

**VGG** is one the best architectures of the 2014 image net challenge. It has 16 layers, with the following architecture:



**If you want to reuse the VGG architecture**, it has been made very user friendly within Keras.

The object VGG16 exists with the following signature:

```
keras.applications.vgg16.VGG16(include_top=True, weights='imagenet', input_tensor=None, input_shape=
None, pooling=None, classes=1000)
```

So that one can define a VGG16 architecture with just one line of code.

Even better, **you can reuse the weights of the network trained on the image net challenge**, with the parameter `weights='imagenet'` !

# ResNet

**ResNet** (for Residual Network) is the winner of the 2015 image net challenge.

It was a mini revolution, introducing for the first time the **residual blocks**:

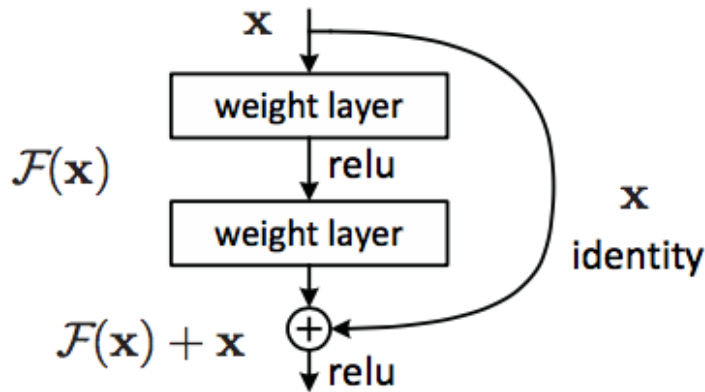


Figure 2. Residual learning: a building block.

**Residual blocks are just skip connections:** the network can choose to skip one or more layers, if it's better for the learning to do so.

The main advantage is that it helps learning over very deep networks: at worst, nothing is learnt, but the information keeps going forward!

There have been several versions of the ResNet, with several depth: 50, 101 and 152 layers. All are available in Keras for easy implementation:

```
keras.applications.resnet.ResNet50(include_top=True, weights='imagenet', input_tensor=None, input_shape=None, pooling=None, classes=1000)
keras.applications.resnet.ResNet101(include_top=True, weights='imagenet', input_tensor=None, input_shape=None, pooling=None, classes=1000)
keras.applications.resnet.ResNet152(include_top=True, weights='imagenet', input_tensor=None, input_shape=None, pooling=None, classes=1000)
```

Again, they are **available with weights trained on the image net challenge**, which is really convenient for training!

## Other architectures

There are plenty of other architectures that are quite famous, and that can be used. Many of them are available in Tensorflow / Keras [here](https://keras.io/applications/) [\(https://keras.io/applications/\)](https://keras.io/applications/):

- Inception

- MobileNet
- DenseNet
- ...

