

Machine learning introduction

Part II – Deep Neural Networks

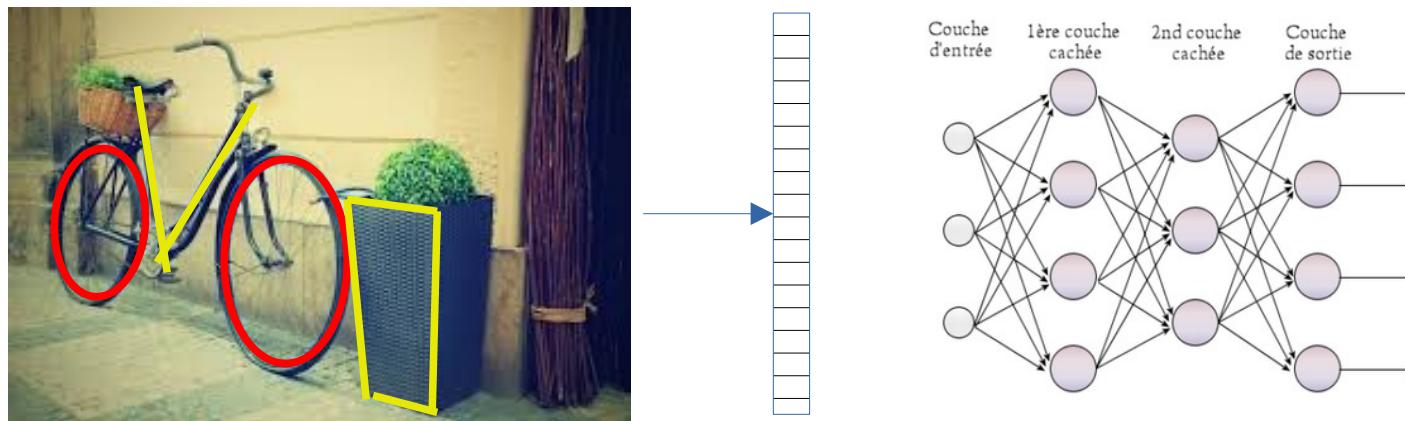
1 The deep networks

Simon Gay

Part II-1 : the deep networks

- The beginning: image recognition

- ‘Classic’ approach:
 - Detection of features (circles, lines, rectangles...)



- These features are used to fill a vector (number, position...)
 - A neural network is then trained on this vector
 - Not very reliable, error rate >25 % at ILSVRC 2011

Part II-1 : the deep networks

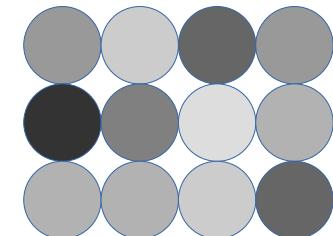
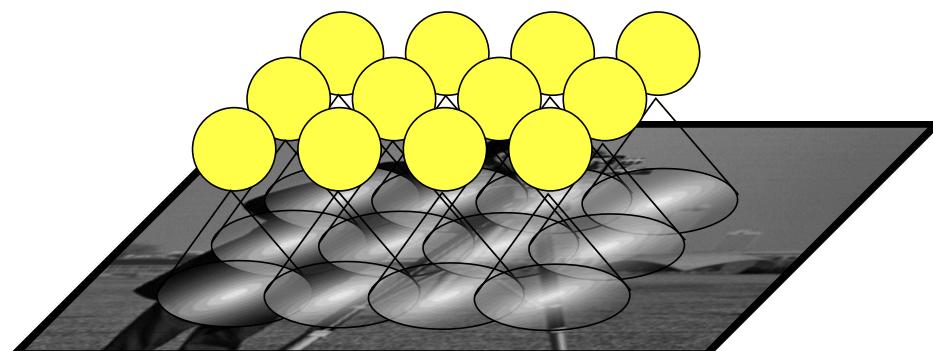
- **Deep learning**
 - Can a neural network generate such a feature vector from image ?
 - Problems:
 - More layers means more neurons and more connections to train
 - If the dataset is too small, the network would fail to generalize !
 - More neurons also means more computational resources
 - For many decades, neural networks were limited both by available resources and by available data sets.

Part II-1 : the deep networks

- **Deep learning**
 - Since middle of 2000s:
 - Computing power significantly increased
 - Parallelization on GPU (e.g. CUDA, 2007)
 - Dedicated architectures (Tensor Processing Unit, 2016)
 - But also collected data
 - GAFA and private life
 - Big data
 - Captcha
 - voluntarism
 - Open-source Dataset COCO (Common Objects in COnText) :
 >200k labeled images, 80+ classes

Part II-1 : the deep networks

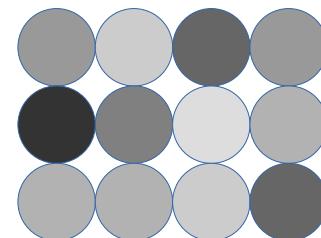
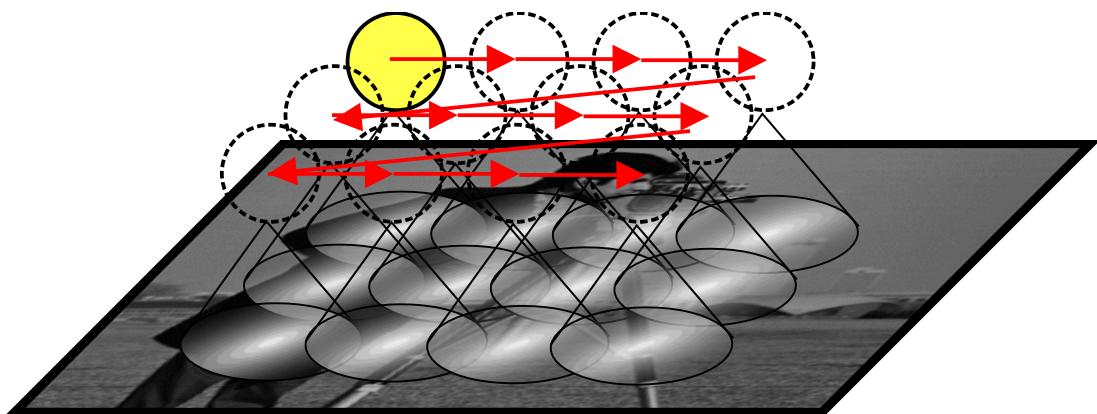
- Deep learning
 - But still not sufficient !
 - Example :
an image of 100x100 pixels requires 10 000 connections per input neuron !
 - Biological inspiration :
 - 1968: analysis on animals show that neurons of first processing layers are connected to small area of the retina
 - Several neurons detects the same feature



Detection of a feature
on the whole image

Part II-1 : the deep networks

- Deep learning
 - Yann Le Cunn (middle of 2000s) :
 - Idea: each neuron define a small feature and is applied on each position of the image



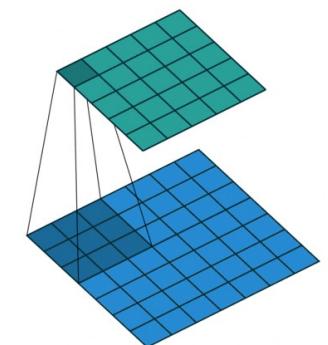
- Feature detection with few weights
- Works like a convolution → *convolutional neurons*

Part II-1 : the deep networks

- **Convolutions**

- A kernel

$$\begin{bmatrix} 0 & 0,25 & 0,5 & 0,25 & 0 \\ 0,25 & 0,5 & 0,75 & 0,5 & 0,25 \\ 0,5 & 0,75 & 1 & 0,75 & 0,5 \\ 0,25 & 0,5 & 0,75 & 0,5 & 0,25 \\ 0 & 0,25 & 0,5 & 0,25 & 0 \end{bmatrix}$$



<https://github.com/vdumoulin>

- Scalar product is applied on each position of the image
 - The result is a matrix giving the presence of kernel's feature on each position of the image



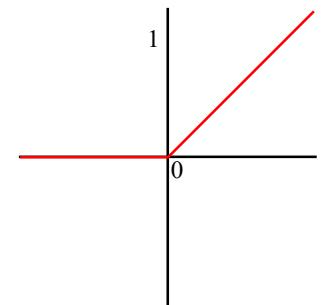
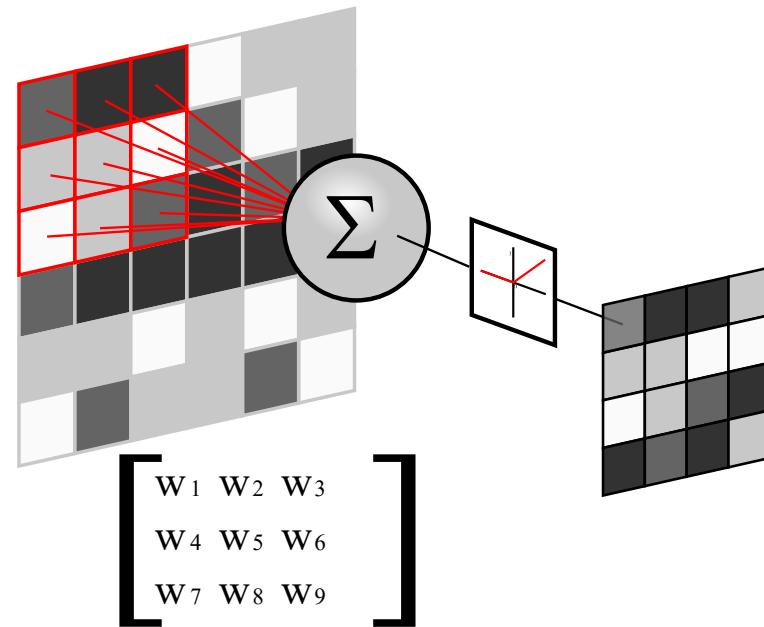
$$\otimes \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} =$$



Part II-1 : the deep networks

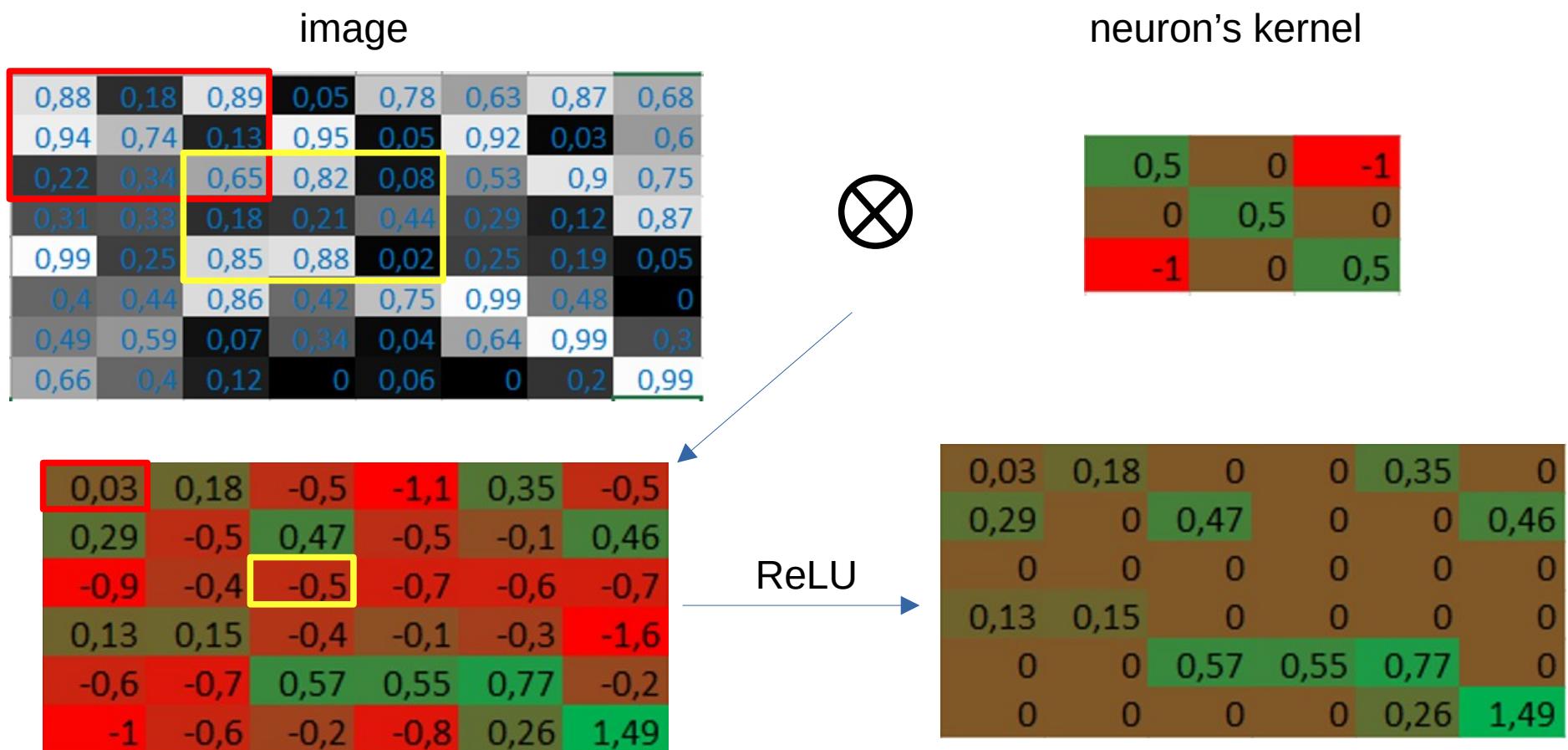
- **Convolutional neurons**

- Similar to a formal neuron
 - Input vector X
 - A set of weights W
→ kernel
 - An activation function
 - Reinforcement based on gradient descent
- Output is an “image”
 - Each pixel p_{xy} is the neuron output at position (x,y)
- Function RELU is often used : detect the presence of features



Part II-1 : the deep networks

- **Convolutional neurons**



- We can notice that the output is smaller than the image

Part II-1 : the deep networks

- Deep Learning

- Padding :

- Pixels with values of 0 are added around the image

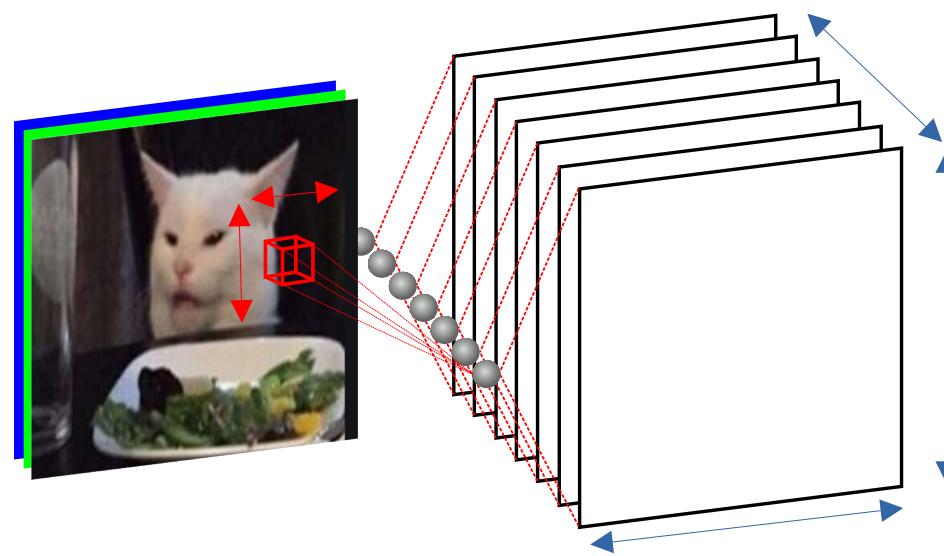
0,88	0,18	0,89	0,05	0,78	0,63	0,87	0,68						
0,94	0,74	0,13	0,95	0,05	0,92	0,03	0,6						
0,22	0,34	0,65	0,82	0,08	0,53	0,9	0,75						
0,31	0,33	0,18	0,21	0,44	0,29	0,12	0,87						
0,99	0,25	0,85	0,88	0,02	0,25	0,19	0,05						
0,4	0,44	0,86	0,42	0,75	0,99	0,48	0						
0,49	0,59	0,07	0,34	0,04	0,64	0,99	0,3						
0,66	0,4	0,12	0	0,06	0	0,2	0,99						

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0,88	0,18	0,89	0,05	0,78	0,63	0,87	0,68	0				
0	0,94	0,74	0,13	0,95	0,05	0,92	0,03	0,6	0				
0	0,22	0,34	0,65	0,82	0,08	0,53	0,9	0,75	0				
0	0,31	0,33	0,18	0,21	0,44	0,29	0,12	0,87	0				
0	0,99	0,25	0,85	0,88	0,02	0,25	0,19	0,05	0				
0	0,4	0,44	0,86	0,42	0,75	0,99	0,48	0	0				
0	0,49	0,59	0,07	0,34	0,04	0,64	0,99	0,3	0				
0	0,66	0,4	0,12	0	0,06	0	0,2	0,99	0	0			

Part II-1 : the deep networks

- **Deep learning**

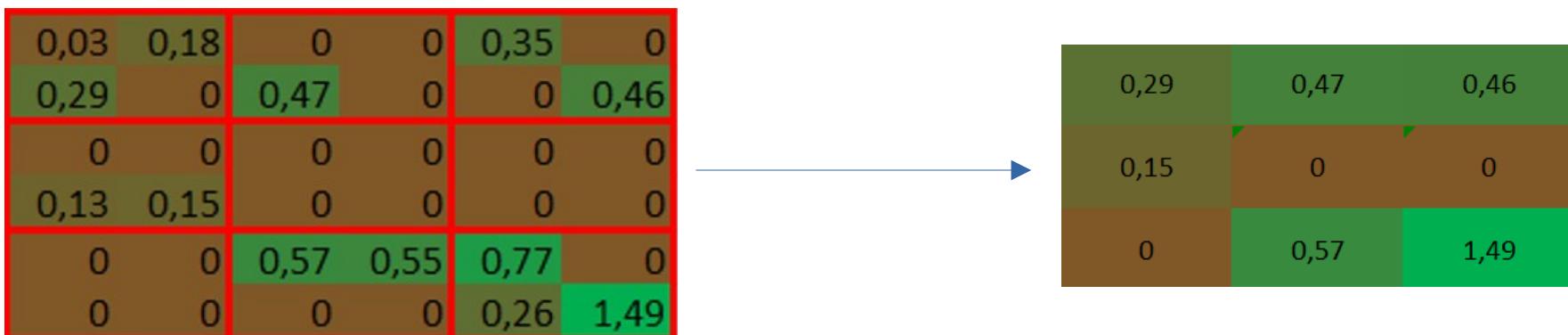
- Each neuron of the first layer generate its own convolution image
 - These images are gathered as a unique image with a depth
 - Depth of an image is the number of “layers”
 - A RGB image has a depth of 3
 - The kernel of a neuron has 3 dimensions (width, height and depth) and moves on 2 dimensions on the image



Part II-1 : the deep networks

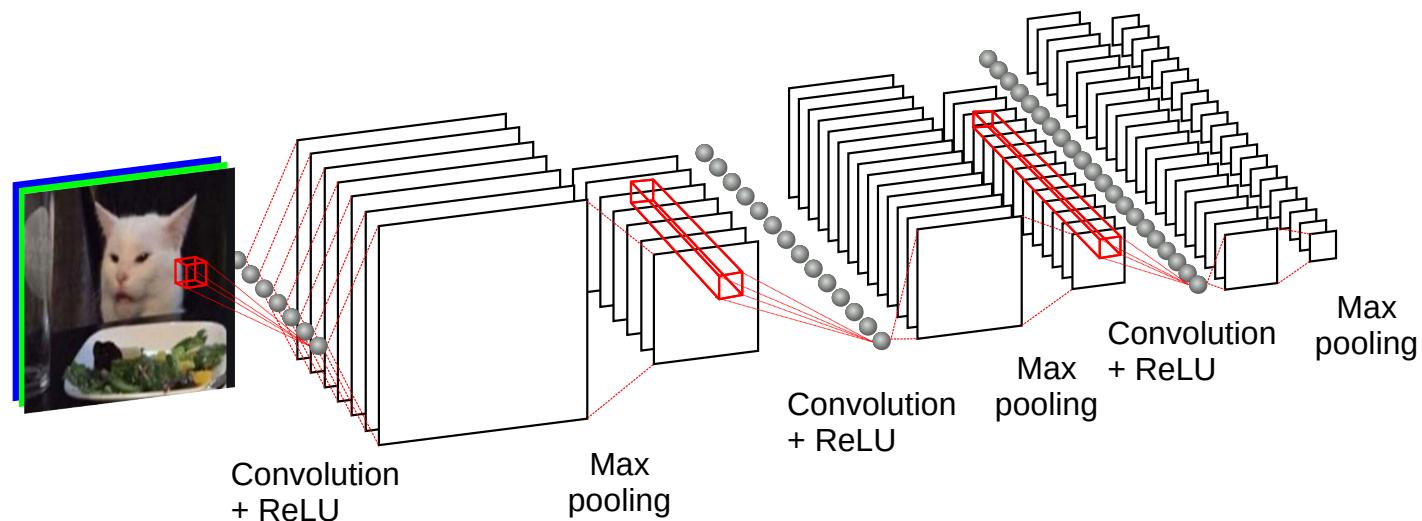
- **Pooling**

- As we looks for the presence of features, it is not necessary to keep adjacent pixels with high value
- Pooling layer :
 - Adjacent pixels are gathered
 - Average pooling → average of the group
 - Max pooling → keep maximum value (most used method)
 - Reduces the number of pixels for next layers



Part II-1 : the deep networks

- Deep learning
 - Convolution and pooling layers can be chained
 - We lose spatial information but get information about complex features

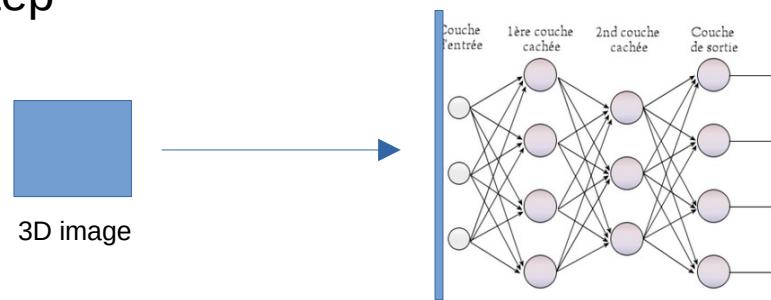


Part II-1 : the deep networks

- **Fully-connected layers**

- After several layers, the image looks more like a vector !
 - We thus can use it as input for a ‘normal’ neuronal network

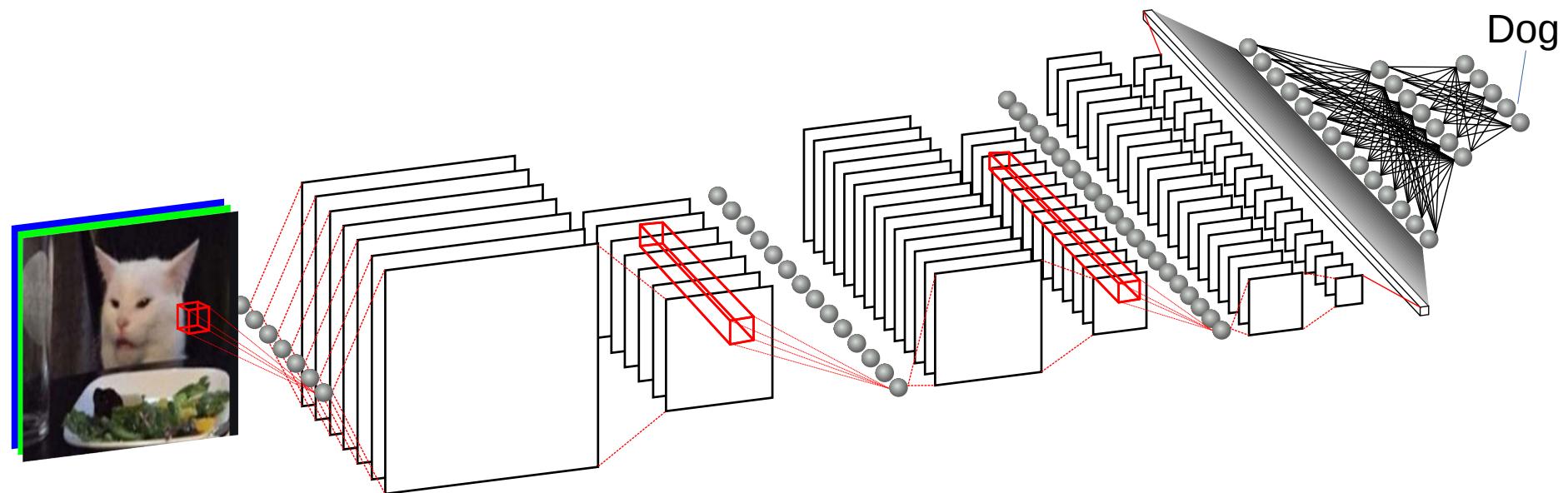
- Last image is converted to a 1-D vector
 - flattening step



- A neuronal network is then used to “interpret” the feature vector
 - these neuron layers are called ‘fully connected’

Part II-1 : the deep networks

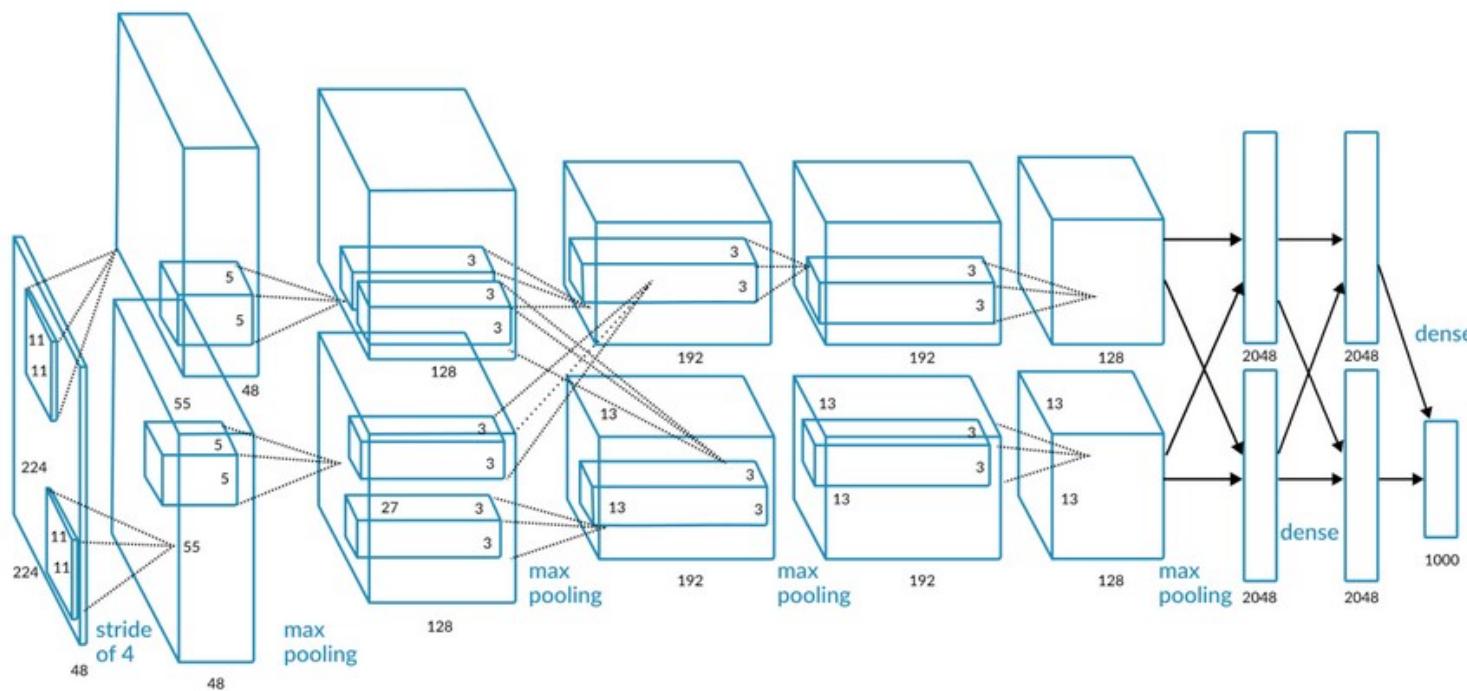
- “Classical” architecture of a convolutional deep network



- convolution + ReLU
- Pooling
- convolution + ReLU
- Pooling
- ...
- Flattening
- fully connected layer
- Output layer

Part II-1 : the deep networks

- The architecture can be more complex:
example : AlphaNet (2012)



Part II-1 : the deep networks

- **Libraries for deep learning model development:**

- TensorFlow : developed by Google, open-source since 2015, derived from *Disbelief* project (2011)
 - Since 2017, a Lite version was created for embedded systems
- MXNet : Developed by Apache, can use many languages
- Caffe
- Theanos : since 2008
- Microsoft Cognitive Toolkit : since 2016
- PyTorch : developed by Facebook, based on Torch
- Keras : library of high level function to easily develop a deep network. Uses Tensorflow or Theanos libraries

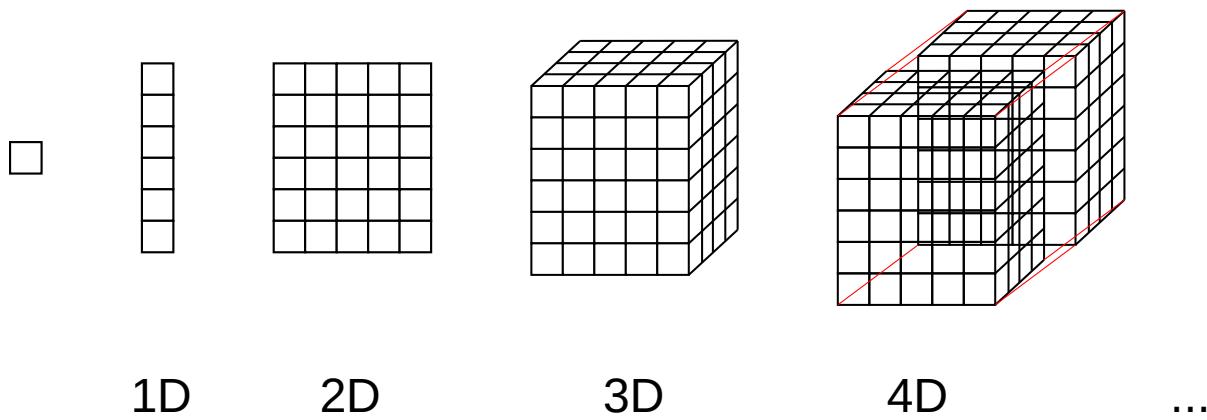


Part II-1 : the deep networks



- **TensorFlow**

- Library dedicated to the manipulation of *tensors*
 - A tensor is a matrix with a given number of dimensions



- Optimized functions
- Possibility to parallelize on GPU !

Part II-1 : the deep networks



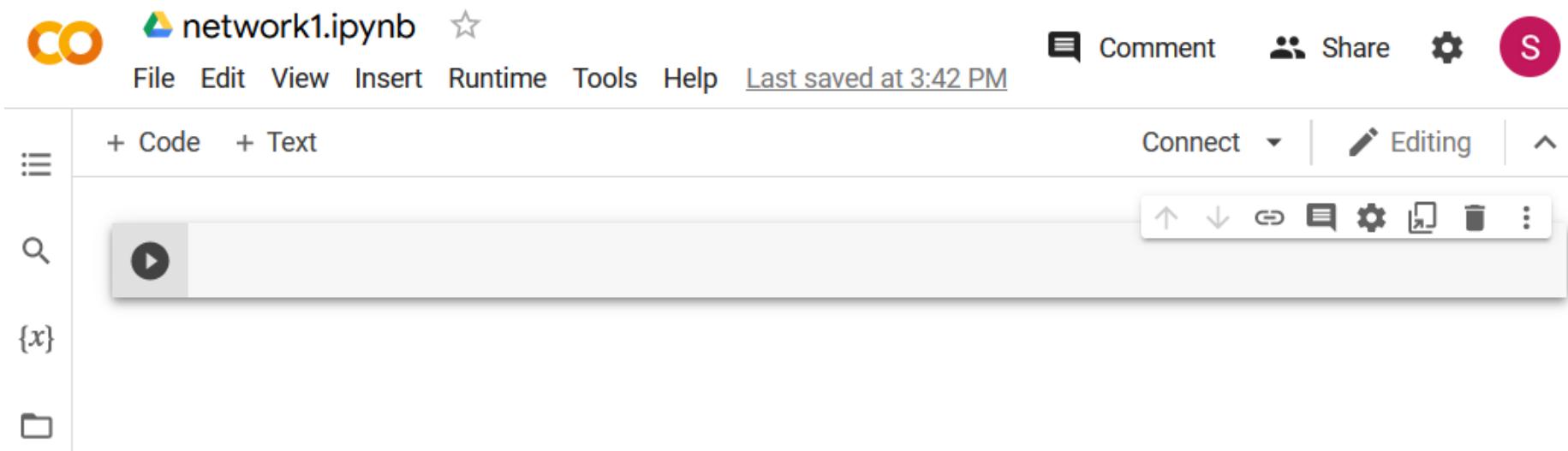
- **Keras**

- Library of high-level functions exploiting Tensorflow or Theanos
- Function to:
 - Load a ‘common’ dataset (ie used for benchmarking)
 - Define the architecture of a network
 - Define learning parameters and loss function
 - Train the network
 - Evaluate the network
 - Exploit the network on new data

Part II-1 : the deep networks



- **Let's start with Keras : implementing a convolutional network**
 - Open your session in Google Colab
 - Create a new notebook



Github : <https://github.com/gaysimon/ARTISAN2022>

Part II-1 : the deep networks



- **Let's start with Keras : implementing a convolutional network**
 - In the first cell, import the required libraries



```
import tensorflow as tf  
  
from tensorflow import keras  
  
import matplotlib.pyplot as plt  
import numpy as np
```

```
import tensorflow as tf  
  
from tensorflow import keras  
  
import matplotlib.pyplot as plt  
import numpy as np
```

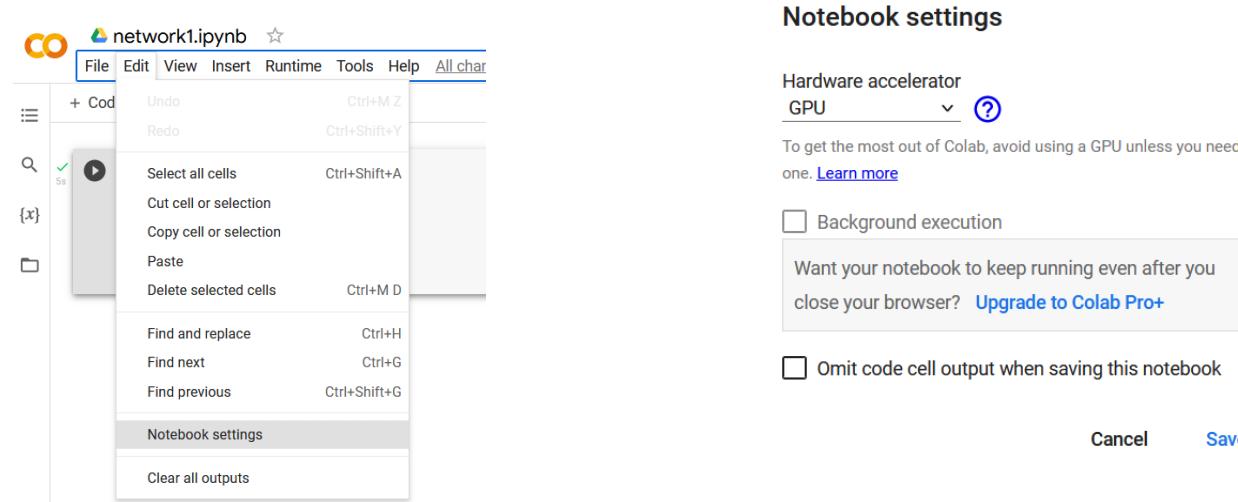
- Tensorflow and Keras for the neural network
- Pyplot and numpy for image display
- Run the cell to check the imports

Part II-1 : the deep networks



- **Using a GPU**

- Training a deep neural network requires a huge computational power, and can take several hours to train
- Hopefully, Tensorflow can parallelize calculations on a GPU
- Hopefully, Google Colab can provide GPUs
- Edit → notebook settings, then select ‘GPU’



Part II-1 : the deep networks



- **Using a GPU**

- DO NOT FORGET to turn off the GPU and stop the session after using it !
 - Runtime → manage sessions

Active sessions

Title	Last execution	RAM used	
 network1.ipynb Current session	GPU 0 minutes ago	0.89 GB	TERMINATE

- Otherwise, Google will reduce your access to GPUs for next sessions !

Part II-1 : the deep networks



- **Using a GPU**

- Then, copy this part of code and paste it just after importing Tensorflow

```
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')

print('Found GPU at: {}'.format(device_name))
```

!\\ Python takes indentation into account

- If the message ‘found GPU’ appears, the GPU is ready

A screenshot of a Jupyter Notebook cell. The cell contains the following Python code:

```
import tensorflow as tf

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))

from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np
```

The cell has a green checkmark icon and a play button icon. Below the code, the output is displayed in a light blue box:

```
Found GPU at: /device:GPU:0
```

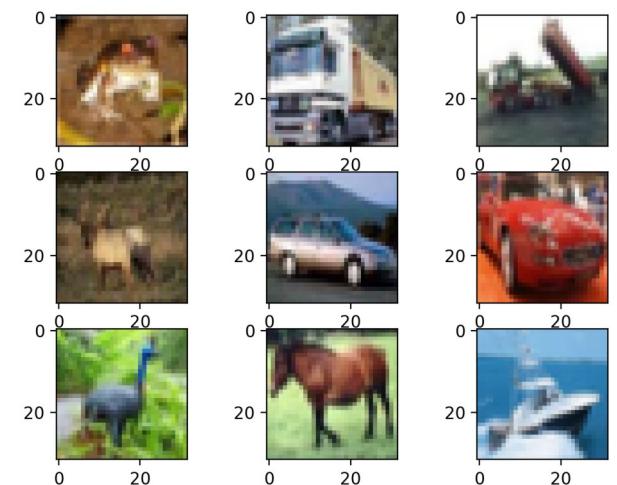
Part II-1 : the deep networks



- **Importing a dataset**

- Keras proposes different common datasets of different types :

- Boston housing prices (values)
→ *boston_housing*
 - CIFAR-10 (color images of size 32x32)
→ *cifar10*
 - CIFAR-100 (same but with 100 classes)
→ *cifar100*
 - Fashion MNIST (images B&W 28x28)
→ *fashion_mnist*
 - IMDB movie reviews (texts) → *imdb*
 - MNIST (images B&W 28x28) → *mnist*
 - Articles Reuters (texts) → *reuters*



CIFAR-10
(50 000 images)

Part II-1 : the deep networks



- **Importing a dataset**
 - We will use the MNIST dataset

- Add a new cell in your Colab ('+code')
- import dataset library
- Load the MNIST dataset into four matrices

```
▶ import tensorflow_datasets as tfds

dataset = keras.datasets.mnist
(img_train, label_train), (img_test, label_test) = dataset.load_data()
```

```
import tensorflow_datasets as tfds

dataset = keras.datasets.mnist
(img_train, label_train), (img_test, label_test) = dataset.load_data()
```

Part II-1 : the deep networks



- **Importing a dataset**

- The matrices must be converted
 - Images from integer [0,255] to float [0,1]
 - Labels must be converted from a number value to a vector of 10 results (e.g. 3 → [0, 0, 0, 1, 0, 0, 0, 0, 0, 0])
 - Keras proposes a function for that : *to_categorical*

```
import tensorflow_datasets as tfds
from tensorflow.keras.utils import to_categorical

dataset = keras.datasets.mnist
(img_train, label_train), (img_test, label_test) = dataset.load_data()

# convert images into float
img_train=img_train/255.0
img_test=img_test/255.0

output_train=keras.utils.to_categorical(label_train, num_classes=10)
output_test=keras.utils.to_categorical(label_test, num_classes=10)

print(img_train.shape)
print(img_test.shape)
```

(60000, 28, 28)
(10000, 28, 28)

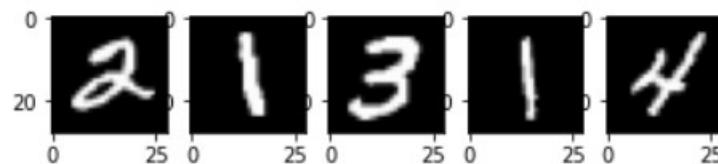
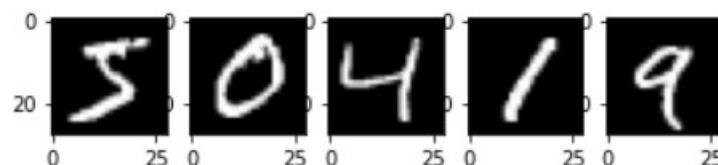
Part II-1 : the deep networks



- Importing a dataset
 - Let's display a sample of our dataset
 - Add a new colab cell
 - Copy the following code :

```
for i in range(10):
    plt.subplot(2,5,i+1)
    plt.imshow(img_train[i], cmap='gray')

plt.show()
```



Part II-1 : the deep networks



- **Importing a dataset**

- Problem : the images have 2 dimensions, but our network requires images with a depths !
- We add a dimension with *numpy's expand* function
 - Add a new colab cell
 - Add these lines :

```
img_train = np.expand_dims(img_train, axis=-1)
img_test = np.expand_dims(img_test, axis=-1)

print(img_train.shape)
print(img_test.shape)
```

(60000, 28, 28) → (60000, 28, 28, 1)
(10000, 28, 28) → (10000, 28, 28, 1)

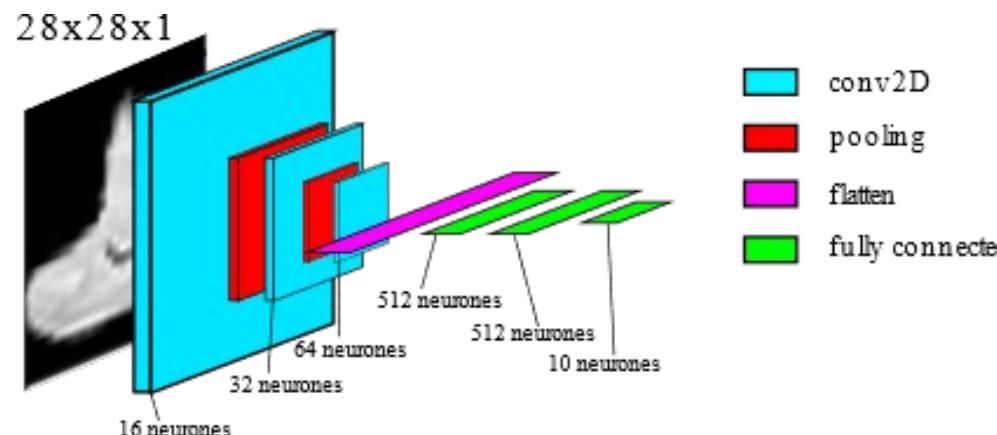
- Now, our images have a depth

Part II-1 : the deep networks



- **Implementing a convolutional network**

- We will implement this simple network :



- Create a new Colab cell
 - We start by declaring a Sequential network



```
model=keras.Sequential()
```

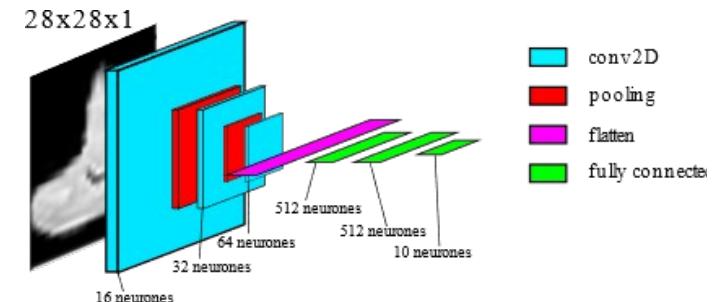
```
|
```

Part II-1 : the deep networks

• Implementing a convolutional network

- We then add layers :

- A first layer of convolutional neurons
 - Input images of size 28x28
 - 16 neurons
 - Convolution kernel of size 3x3
 - Padding
 - A ReLU activation function
- A max pooling layer with groups of size 2x2

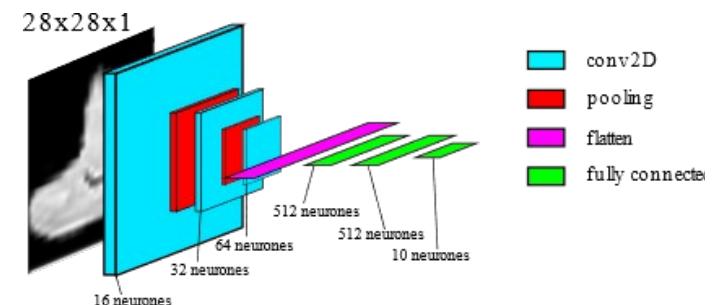


```
model=keras.Sequential()  
  
model.add(keras.layers.Conv2D(input_shape=(28,28,1), filters=16, kernel_size=(3,3), padding="same", activation="relu"))  
model.add(keras.layers.MaxPool2D(pool_size=(2,2), strides=(2,2)))  
|
```

```
model=keras.Sequential()  
  
model.add(keras.layers.Conv2D(input_shape=(28,28,1), filters=16, kernel_size=(3,3), padding="same", activation="relu"))  
model.add(keras.layers.MaxPool2D(pool_size=(2,2), strides=(2,2)))
```

Part II-1 : the deep networks

- **Implementing a convolutional network**
 - We then add layers :
 - A new convolutional layer (32 neurons)
 - A new max pooling
 - A last convolutional layer (64 neurons)

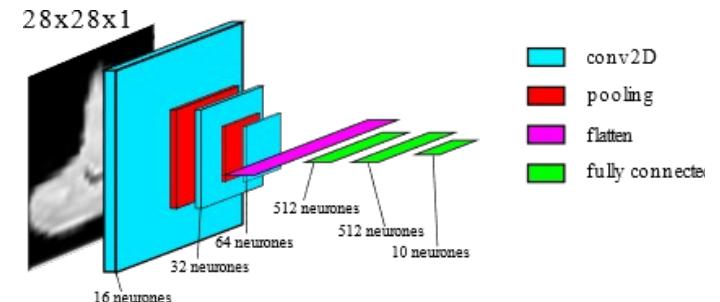


```
model=keras.Sequential()  
  
model.add(keras.layers.Conv2D(input_shape=(28,28,1), filters=16, kernel_size=(3,3), padding="same", activation="relu"))  
model.add(keras.layers.MaxPool2D(pool_size=(2,2), strides=(2,2)))  
  
model.add(keras.layers.Conv2D(filters=32, kernel_size=(3,3), padding="same", activation="relu"))  
model.add(keras.layers.MaxPool2D(pool_size=(2,2),strides=(2,2)))  
  
model.add(keras.layers.Conv2D(filters=64, kernel_size=(3,3), padding="same", activation="relu"))
```

Part II-1 : the deep networks

• Implementing a convolutional network

- We then add the fully connected part :
 - A flatten layer
 - 2 fully connected layers of 512 neurons
 - An output fully connected layer of 10 neurons (softmax function)



```
[ ]  
model=keras.Sequential()  
  
model.add(keras.layers.Conv2D(input_shape=(28,28,1), filters=16, kernel_size=(3,3), padding="same", activation="relu"))  
model.add(keras.layers.MaxPool2D(pool_size=(2,2), strides=(2,2)))  
model.add(keras.layers.Conv2D(filters=32, kernel_size=(3,3), padding="same", activation="relu"))  
model.add(keras.layers.MaxPool2D(pool_size=(2,2), strides=(2,2)))  
model.add(keras.layers.Conv2D(filters=64, kernel_size=(3,3), padding="same", activation="relu"))  
  
model.add(keras.layers.Flatten( ))  
  
model.add(keras.layers.Dense(units=512,activation="relu"))  
model.add(keras.layers.Dense(units=512,activation="relu"))  
  
model.add(keras.layers.Dense(units=10, activation="softmax"))
```

```
model.add(keras.layers.Flatten( ))  
  
model.add(keras.layers.Dense(units=512,activation="relu"))  
model.add(keras.layers.Dense(units=512,activation="relu"))  
  
model.add(keras.layers.Dense(units=10, activation="softmax"))
```

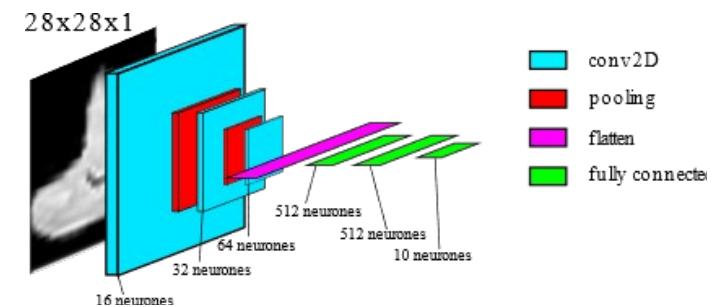
Part II-1 : the deep networks

- **Implementing a convolutional network**
 - The model architecture can be displayed with
model.summary()

```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 28, 28, 16)	160
max_pooling2d (MaxPooling2D)	(None, 14, 14, 16)	0
)		
conv2d_1 (Conv2D)	(None, 14, 14, 32)	4640
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 32)	0
conv2d_2 (Conv2D)	(None, 7, 7, 64)	18496
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 512)	1606144
dense_1 (Dense)	(None, 512)	262656
dense_2 (Dense)	(None, 10)	5130
<hr/>		
Total params: 1,897,226		
Trainable params: 1,897,226		
Non-trainable params: 0		



Part II-1 : the deep networks



- **Keras** : some important types of layers (with some important parameters) :

- **tf.keras.layers.Dense**

```
    units,  
    activation=None,  
    use_bias=True,  
)
```

→ number of neurons in the layer
→ activation function
→ use a bias or not

- **tf.keras.layers.Conv2D**

```
    filters,  
    kernel_size,  
    strides=(1, 1),  
    padding="valid",  
    activation=None,  
    use_bias=True,
```

```
)
```

note : versions 1D and 3D also exist

→ number of neurons
→ size of kernel
→ displacement (steps) of kernel
→ padding when "same"
→ activation function
→ use a bias or not

- **tf.keras.layers.MaxPooling2D**

```
    pool_size=(2, 2),  
    strides=None,  
    padding="valid"
```

```
)
```

→ pooling reduction
→ 'movements'

- Some activation functions: relu, sigmoid, softmax, tanh, ...
- Complete list of 100+ types of layers on: <https://keras.io/api/layers/>

Part II-1 : the deep networks



- **Keras** : construction of a network model
 - A sequential model can also be defined as a vector of layers:

```
model = Sequential([
    Conv2D(input_shape=(224,224,3), filters=64, kernel_size=(3,3), padding="same", activation="relu"),
    Conv2D(filters=64,kernel_size=(3,3), padding="same", activation="relu"),
    MaxPool2D(pool_size=(2,2), strides=(2,2)), ...
])
```

- Another method: functionnal model
 - Layers are created separately and connected

```
input_layer = Input(shape=(28,28,1))
conv1 = Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(32,32,1))(input_layer)
conv2 = Conv2D(64, (3, 3), activation='relu', input_shape=(32,32,1))(input_layer)
merge = concatenate([conv1, conv2])
flatten = Flatten()(merge)
output_layer = Dense(num_classes, activation='softmax')(flatten)
model = Model(inputs=input_layer, outputs=output_layer)
```
 - Allows defining non sequential models (fusion, split...)

Part II-1 : the deep networks



- **Learning parameters**

- After constructing the network, we have to define
 - The loss function (error measure to optimize)
 - The optimization function (weight update algorithm)
- With a softmax activation function, we will use the CategoricalCrossentropy loss function !

Other loss functions :

- *MeanSquaredError* : $E = (r - y)^2$
- ... and dozens of other available functions : <https://keras.io/api/losses>
- In a new Colab cell, add :

A screenshot of a Jupyter Notebook cell. The cell contains a single line of Python code: "loss=keras.losses.CategoricalCrossentropy()". A play button icon is visible at the top left of the cell.

```
loss=keras.losses.CategoricalCrossentropy();
```

Part II-1 : the deep networks



- **Learning parameters**

- The optimization function is the function that reinforce neurons' weights
 - The SGD is the gradient descent presented previously
 - Other more optimized function adapt the learning rate to reduce learning time
 - Adam, Adadelta, Nadam, Ftrl... (see <https://keras.io/api/optimizers/>)
- We select Adam optimizer, with a learning rate of 0.001 :



```
loss=keras.losses.CategoricalCrossentropy()  
optim=keras.optimizers.Adam(learning_rate=0.001)  
|
```

```
optim=keras.optimizers.Adam(learning_rate=0.001)
```

Part II-1 : the deep networks



- **Finalizing the network**

- Finally, the network is compiled : the neuron reinforcement functions are defined according to the network architecture and selected loss and optimization functions :



```
loss=keras.losses.CategoricalCrossentropy()  
optim=keras.optimizers.Adam(learning_rate=0.001)  
  
model.compile(loss=loss, optimizer=optim, metrics=["accuracy"])
```

```
model.compile(loss=loss, optimizer=optim, metrics=["accuracy"])
```

- Run the cell : if no error message appears, the network is ready !

Part II-1 : the deep networks



- **Training the network**

- Keras takes care of everything with function *fit*
 - Parameters:
 - The training data and label
 - The size of batches (number of data learned simultaneously)
 - The number of epoches
 - The level of details to display
 - In a new cell, write :



```
model.fit(img_train, output_train, batch_size=16, epochs=10, verbose=2)
```

```
Epoch 1/10  
3750/3750 - 13s - loss: 0.1175 - accuracy: 0.9638 - 13s/epoch - 3ms/step  
Epoch 2/10  
3750/3750 - 10s - loss: 0.0502 - accuracy: 0.9848 - 10s/epoch - 3ms/step  
-
```

- The learning process can takes several minutes...

Part II-1 : the deep networks



- **Evaluating/Exploiting the network**

- Keras takes care of everything (again) :
 - Function evaluate (works in a same way than fit)

```
▶ model.evaluate(img_test, output_test, batch_size=16, verbose=2)

625/625 - 1s - loss: 0.0703 - accuracy: 0.9884 - 1s/epoch - 2ms/step
[0.07027573138475418, 0.9883999824523926]
```

◀ < 0.12 %

- An image can be used as parameter to get a prediction :

```
▶ print(label_test[0:1])

model(img_test[0:1,:,:])
```

◀ [7] ←

```
<tf.Tensor: shape=(1, 10), dtype=float32, numpy=
array([[1.1164599e-26, 2.6863731e-18, 2.3253600e-17, 1.2962786e-17,
       2.5873384e-16, 5.7846084e-20, 1.6555734e-36, 1.0000000e+00,
       1.5967833e-22, 9.6893570e-12]], dtype=float32)>
```

Part II-1 : the deep networks

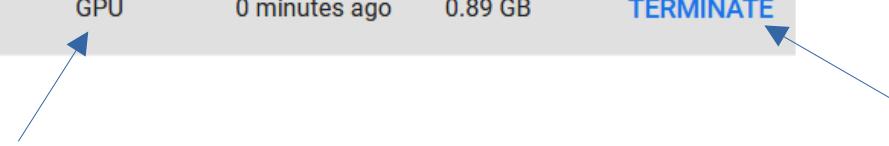


- Save your colab
- **Do not forget to disable the GPU and stop the session !**

Active sessions

Title	Last execution	RAM used
network1.ipynb Current session	0 minutes ago	0.89 GB

Edit → notebook settings, then select 'none'

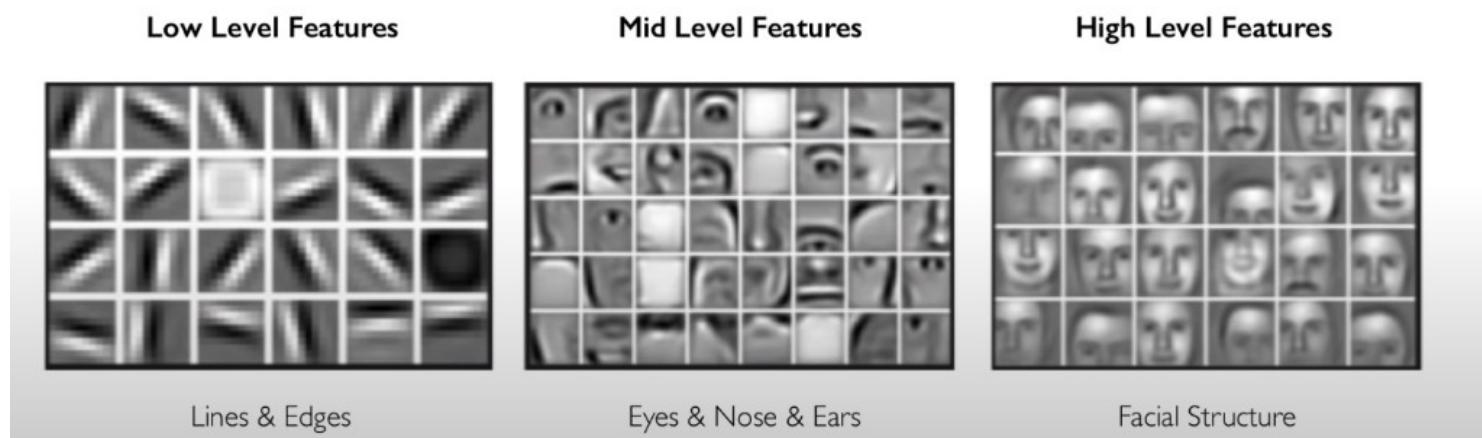


- **Note to go further :** there is a dataset similar to MNIST :
 - dataset = keras.datasets.**fashion_mnist**
 - You can try to improve your model to get the best accuracy

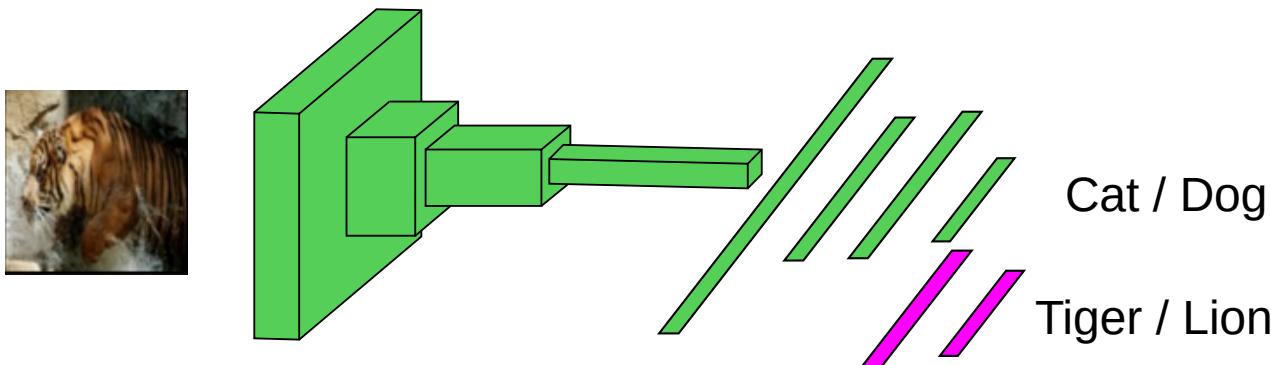
Training a deep network:
the question of the dataset

Part II-1 : the deep networks

- **Training a deep network**
 - The more layers of neurons there are, the larger the learning dataset must be
 - A deep network may require a dataset of tens, even hundreds thousand images (or other kind of data)
 - Do we need a huge dataset for each new problem ?
 - For two similar problems, features defined by lowest layers are similar !



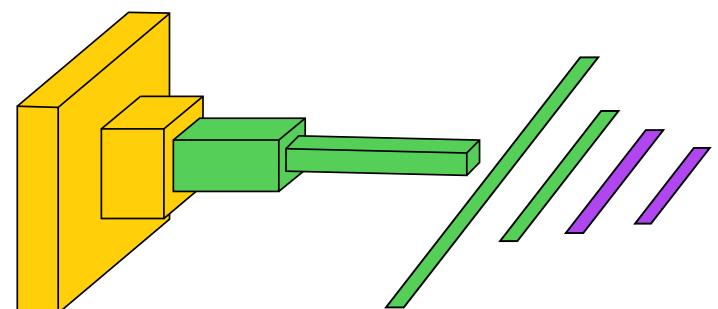
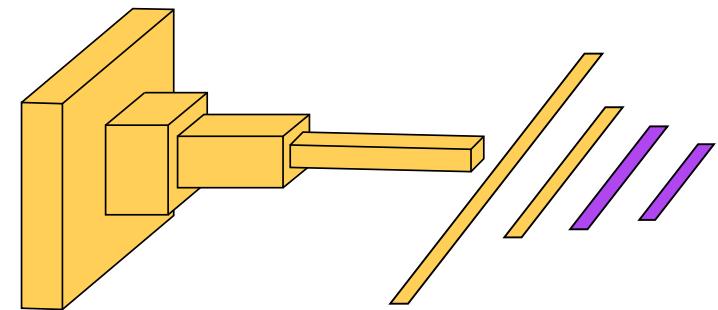
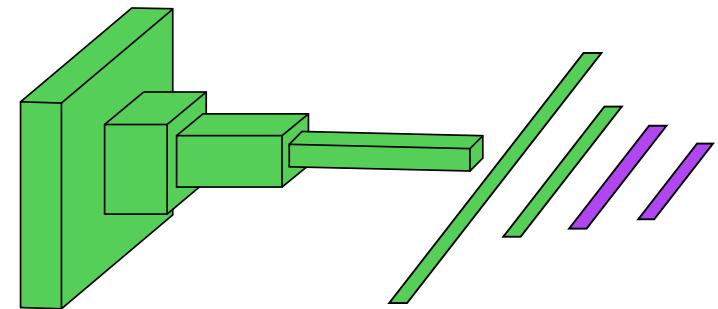
Part II-1 : the deep networks

- **Training a deep network**
 - For similar problem, it is possible to re-use low layers of an existing trained network
 - Principle of **transfer learning**
 - We can use a network trained on a similar problem
 - We then replace last layers of the network with new layers
- 
- The number of layers to replace depends on problems similarities and available dataset's size

Part II-1 : the deep networks

- **Transfer Learning**

- Fine tuning: last layers (often fully connected part) are replaced with new initialized layers
 - Faster learning with less examples
 - Low level features can adapt to the new problem
 - Still requires a large dataset
- Feature extraction: last layers are replaced and remaining layers are frozen
 - Fast learning with few examples
 - Resulting network is not always efficient
- Partial Fine tuning: olast layers are replaces, but only lowest layers are frozen
 - Good compromise between above methods
 - Low level features are more ‘universal’



Part II-1 : the deep networks

- **Transfer Learning**
 - **Several well-trained open-source networks are proposed for transfer learning :**
 - AlexNet : architecture used for ILSVRC 2012
 - VGGnet : one of most versatile. Uses 16 convolution layers
 - GoogleNet : winner of ILSVRC 2014, uses 22 convolution layers
 - ResNet : winner of ILSVRC 2015 (with an error ratio of 3,57 %, it is more efficient than human participants!), uses 152 convolution layers

Part II-1 : the deep networks



- **Let's implement a true deep neural network !**

- New way to build a dataset
- Transfer learning from an existing network
- Goal : recognize a cat from a dog
- Create a new Colab Notebook, enable GPU, and copy the imports

CO deep1.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text

6s

```
import tensorflow as tf

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))

from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np
```

Found GPU at: /device:GPU:0

```
import tensorflow as tf

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))

from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np
```

Part II-1 : the deep networks



- **The dataset**

- We will download a set of images and unzip them in different folders
- Copy and paste these code lines in two different colab cells, and execute them :

```
!wget --no-check-certificate \
    https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip \
    -O /tmp/cats_and_dogs_filtered.zip
```

```
import os
import zipfile

local_zip = '/tmp/cats_and_dogs_filtered.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/tmp')
zip_ref.close()

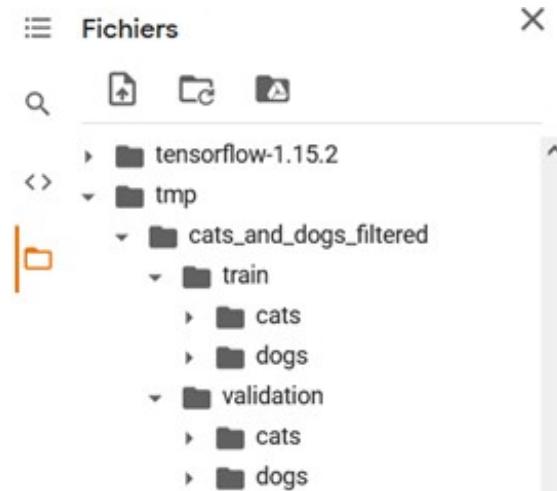
train_dir = '/tmp/cats_and_dogs_filtered/train'
test_dir = '/tmp/cats_and_dogs_filtered/validation'
```

Part II-1 : the deep networks



- **The dataset**

- The images are stored in a specific folder tree



- Keras proposes an object that can browse this kind of folder tree to feed a neural network for training or evaluating it :
 - The *ImageDataGenerator*

Part II-1 : the deep networks



- **The dataset**

- In a new Colab cell, we write the image generator :

```
trainDataGenerator = keras.preprocessing.image.ImageDataGenerator(rescale=1./255)
trainData = trainDataGenerator.flow_from_directory(directory=train_dir, target_size=(224,224))

testDataGenerator = keras.preprocessing.image.ImageDataGenerator(rescale=1./255)
 testData = testDataGenerator.flow_from_directory(directory=test_dir, target_size=(224,224))
```

Found 2000 images belonging to 2 classes.

Found 1000 images belonging to 2 classes.

- Then, we load the VGG16 network

```
VGG16= keras.applications.VGG16(weights="imagenet")

VGG16.summary()
```

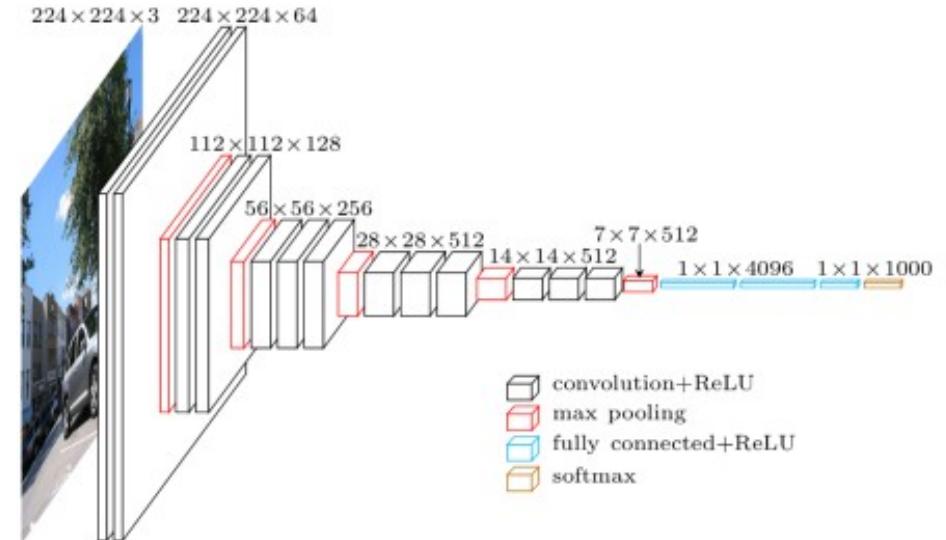
Part II-1 : the deep networks



- **The VGG16 network**

- We cannot train a network with so many parameters

Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0



- Especially with such a small dataset !
 - Hopefully, the VGG16 network was trained with multiple animals (including cats and dogs) : the lowest layers may recognize pertinent features
- Note : VGG16 was trained during multiple days on high-end GPUs with *imageNet* dataset (>1M labeled images)

Part II-1 : the deep networks



- **The VGG16 network**

- The VGG16 network, as proposed by Keras, can be loaded without the fully connected part
 - Parameter `include_top` can be set to `False`
 - Parameter `weights='imagenet'` allows loading weight obtained with imagenet dataset (10M+ images)
- Add a new Colab cell and paste this :

```
base_VGG16= keras.applications.VGG16(weights="imagenet", include_top=False, input_shape=(224,224,3))
base_VGG16.summary()
```



- Observe where the network is cut :
 - As the flatten layer is not included, other convolutional layers can be added

Part II-1 : the deep networks



- **The VGG16 network**

- We freeze the weights of the network (in a new Colab cell) :

```
base_VGG16.trainable=False  
  
base_VGG16.summary()
```

```
=====  
Total params: 14,714,688  
Trainable params: 0 ←  
Non-trainable params: 14,714,688
```

Part II-1 : the deep networks



- **The VGG16 network**

- The VGG16 part is then used as a layer for our network
- We then add the fully connected part using a flatten layer, a fully connected layer with 50 neurons and an output flatten layer with 2 neurons

```
model=keras.models.Sequential()
model.add(base_VGG16)

model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(50, activation="relu"))
model.add(keras.layers.Dense(2, activation="softmax"))

model.summary()
```

=====

Total params: 15,969,240
Trainable params: 1,254,552 ←
Non-trainable params: 14,714,688

Part II-1 : the deep networks



- **The VGG16 network**

- The network is compiled with Categorical_Crossentropy and Adam functions

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

- And trained using the data generator that provides both data and labels

```
model.fit(trainData, epochs=5, batch_size=32)
```



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

model.fit(trainData, epochs=5, batch_size=32)
```

Epoch 1/5

19/63 [=====>.....] - ETA: 6s - loss: 0.8847 - accuracy: 0.7105

Part II-1 : the deep networks



- **The VGG16 network**

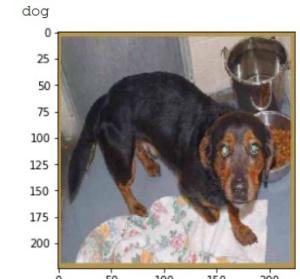
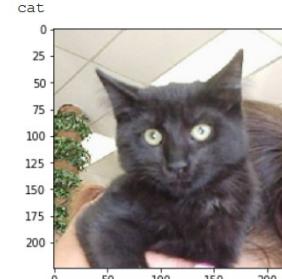
- The network can then be evaluated :

```
model.evaluate(testData, batch_size=32, verbose=2)
```

```
32/32 - 6s - loss: 0.2513 - accuracy: 0.9080 - 6s/epoch - 186ms/step  
[0.2513466775417328, 0.9079999923706055]
```

- The following code allows testing an image :

```
test_cats_files = os.listdir(test_dir+"/cats")  
test_dogs_files = os.listdir(test_dir+"/dogs")  
  
img = keras.preprocessing.image.load_img(test_dir+"/cats/"+test_cats_files[8],target_size=(224,224))  
#img = keras.preprocessing.image.load_img(test_dir+"/dogs/"+test_dogs_files[8],target_size=(224,224))  
img = np.asarray(img)  
plt.imshow(img)  
img = np.expand_dims(img, axis=0)  
  
prediction=model.predict(img)  
  
if prediction.argmax()==0: print("cat")  
else : print("dog")
```



Part II-1 : the deep networks

- **Applications**
 - Question : is it possible to recreate the image from the feature vector ?
 - Successive convolution generate an information loss
 - The feature vector only contains a fraction of image information
 - New layers to reverse convolutional layer's processing
 - De-pooling
 - transposed convolution

Part II-1 : the deep networks

- **Image reconstruction**

- De-Pooling :

- Nearest neighbor

0,29	0,47	0,46
0,15	0	0
0	0,57	1,49

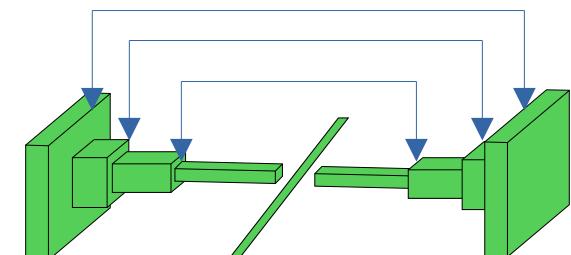
0,29	0,29	0,47	0,47	0,46	0,46
0,29	0,29	0,47	0,47	0,46	0,46
0,15	0,15	0	0	0	0
0,15	0,15	0	0	0	0
0	0	0,57	0,57	1,49	1,49
0	0	0,57	0,57	1,49	1,49

- Bed of nails

0,29	0	0,47	0	0,46	0
0	0	0	0	0	0
0,15	0	0	0	0	0
0	0	0	0	0	0
0	0	0,57	0	1,49	0
0	0	0	0	0	0

- Max UnPooling

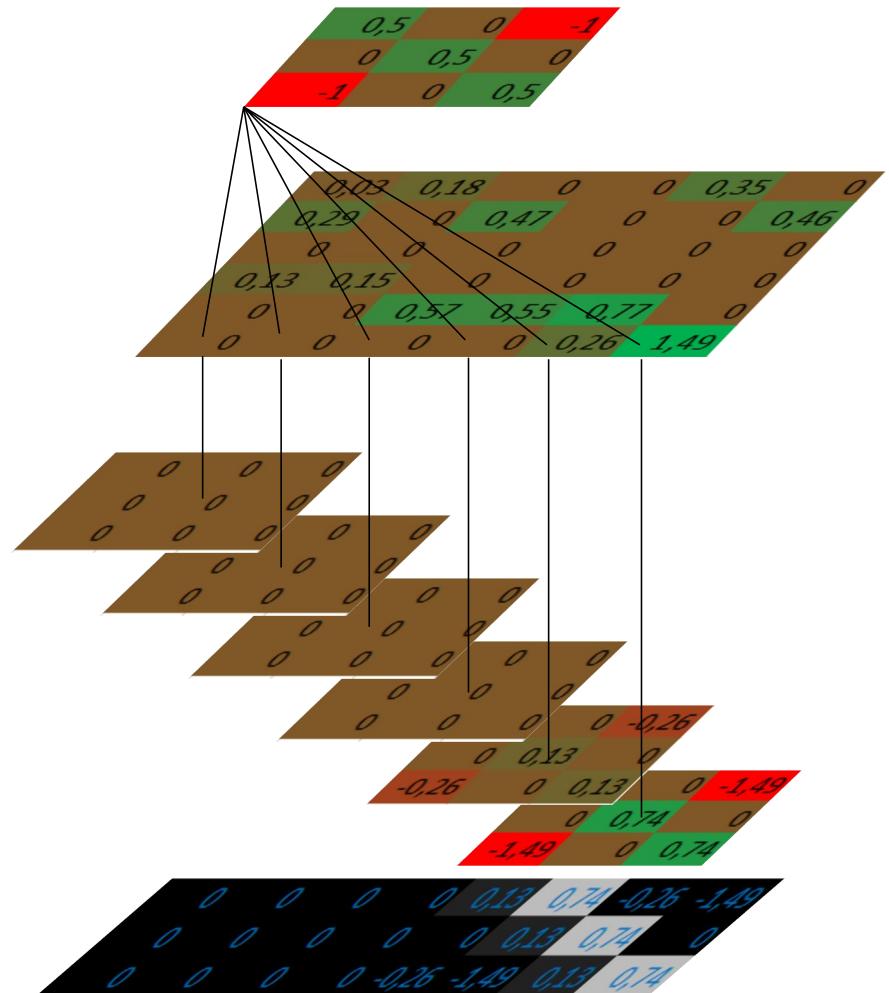
- use the mirror pooling layer to define which pixel get the value



Part II-1 : the deep networks

- **Image Reconstruction**

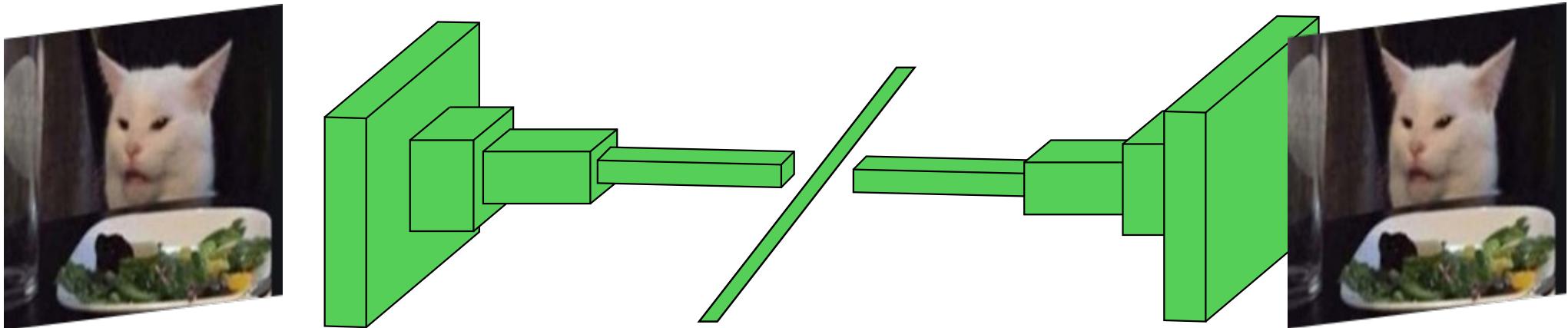
- Transposed Convolution:
 - Principle of convolution is ‘reversed’
 - Output image is the sum of kernel weighted by convolution image values



Part II-1 : the deep networks

- **Image reconstruction**

- The fully-connected part is replaced by a ‘mirror’ version of the network

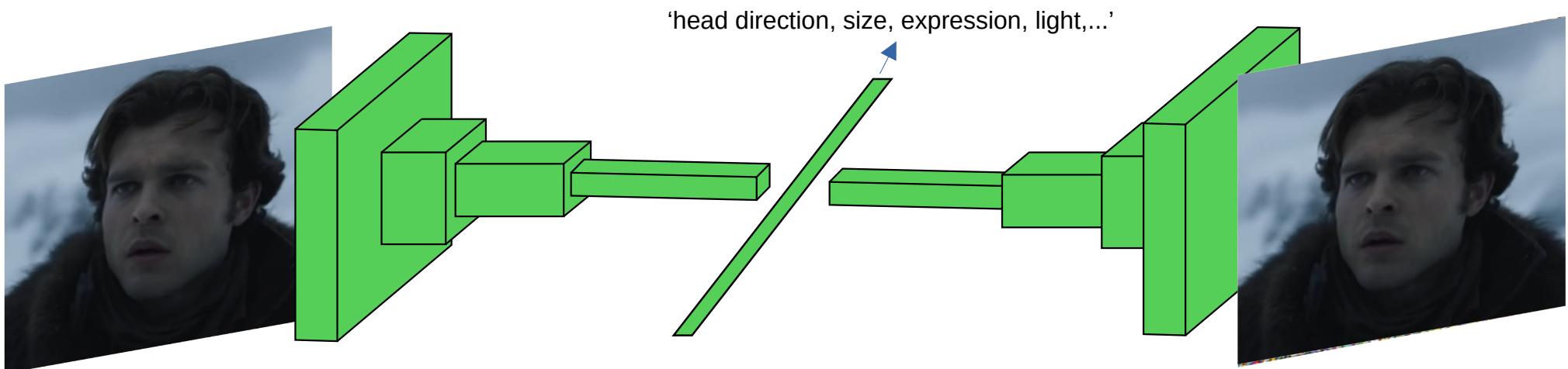


- Input image is used as expected result ($E = | \text{Img}_{\text{ref}} - \text{Img}_{\text{gen}} |^2$)
 - First part encodes the information: ‘encoder’ network
 - Second part ‘decode’ the feature vector to reconstruct the image: ‘decoder’ network
 - Decoder learns to generate textures, edges... from feature description
 - Very efficient way to de-noise an image

Part II-1 : the deep networks

- **Image reconstruction**

- The network ‘learns’ to define minimal information
- Let’s train the network on images of a specific face

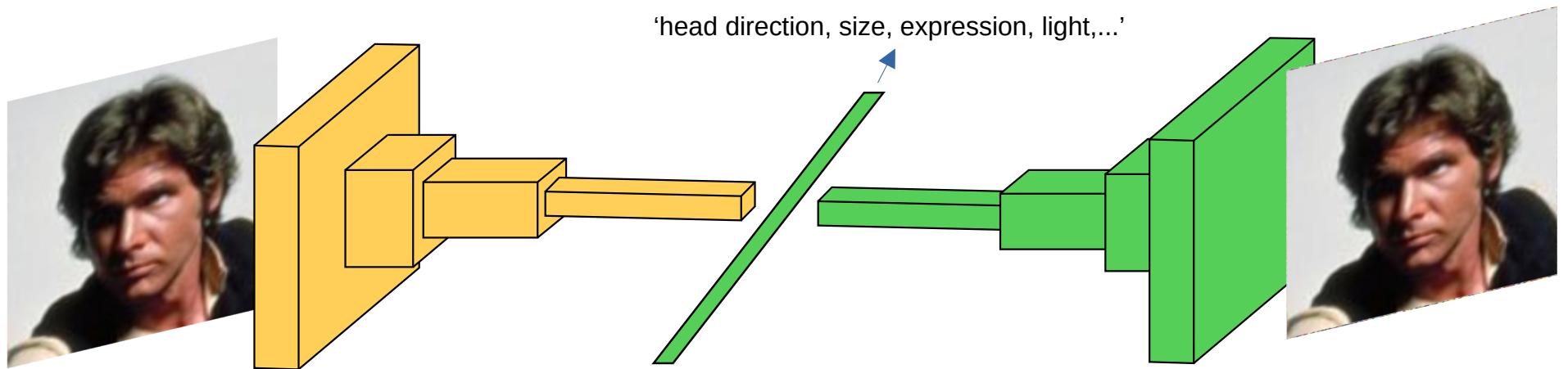


- The network will learn to encode minimum data to reconstruct the face
 - Head orientation and size, facial expression... (implicitly)
 - The decoder learns to reconstruct the image with these features

Part II-1 : the deep networks

- **Image reconstruction**

- The encoder's weights are frozen
- Decoder is reinitialized and trained with another face

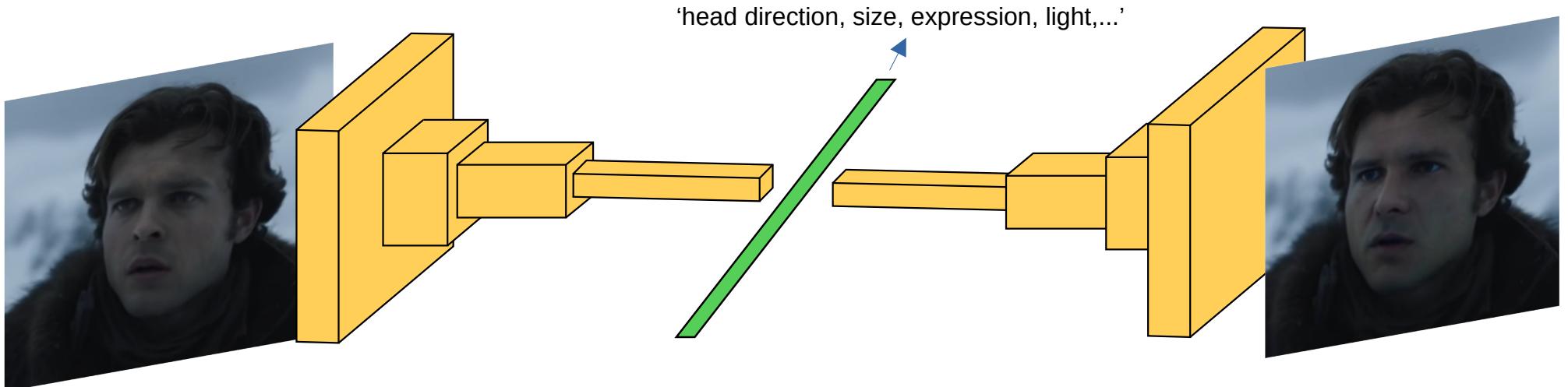


- The encoder will provide feature vectors in a similar way
- But the decoder must learn to generate this face from features

Part II-1 : the deep networks

- **Image reconstruction**

- We now froze the whole network
- An image of first face is presented to the network
 - The encoder extracts features from this face
 - The decoder will generate an image of the second face from these features



- Principle of deep fake

Part II-1 : the deep networks

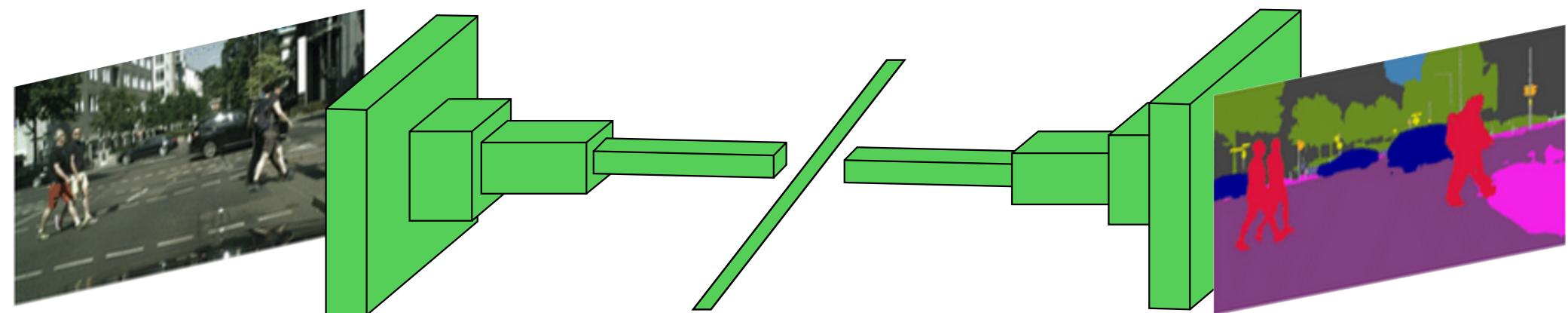
- Image reconstruction



Harrison Ford in Solo: A Star Wars Story (Shamook)

Part II-1 : the deep networks

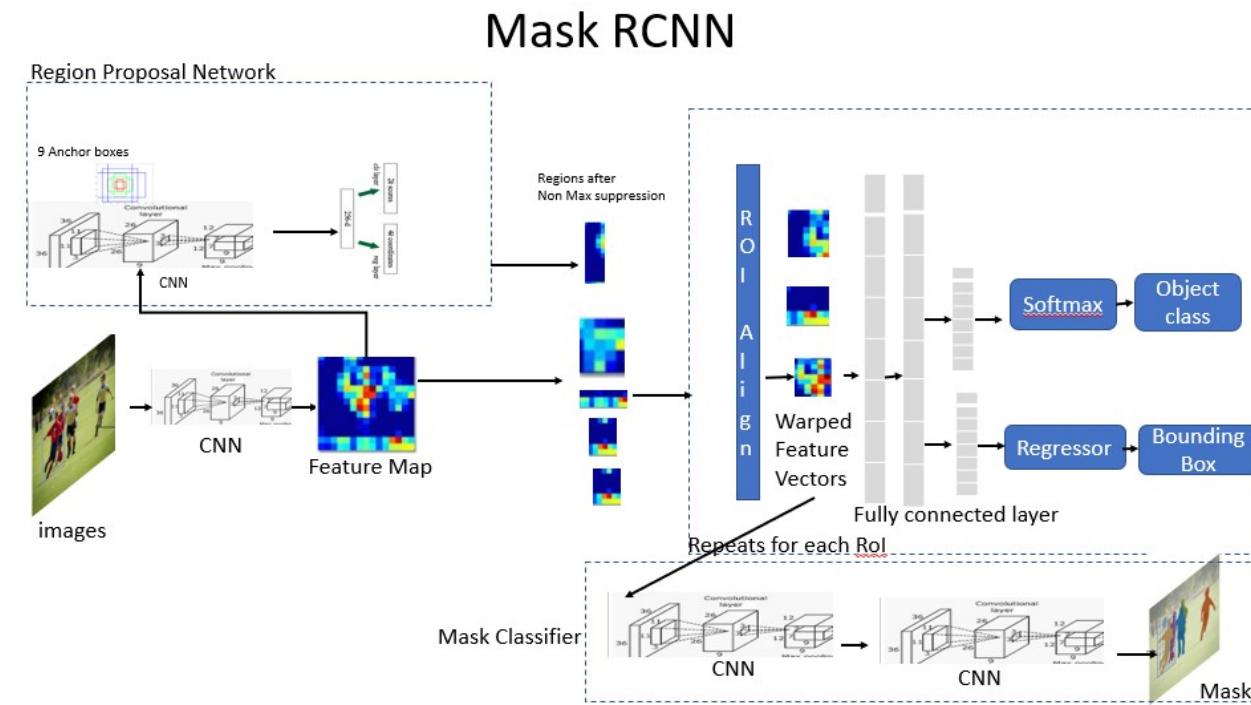
- **Fully-convolutional networks**
 - We can also use other type of images as output
 - Example : Mask-RCNN
 - The network is trained on images, with hand-labeled mask images



Part II-1 : the deep networks

- **Fully-convolutional network**

- Mask-RCNN : in practice, a little more complex
 - Detection of objects in the image
 - extraction of region of interest containing objects
 - Detection pixel by pixel to generate a mask



Part II-1 : the deep networks

- **Fully-convolutional network**
 - Mask-RCNN, trained on dataset COCO (Common Objects in COnext)



Training a deep network: bias and security

Part II-1 : the deep networks

- **The dataset bias**
 - Training a deep network requires a huge number of examples
 - Gathering a large number of example can be difficult, leading to statistical bias in the collected samples
 - Neural networks are very sensitive to statistical bias in datasets and will use ANY regularity they can found to minimize errors
 - Example : a network defining the age of a person uses the size of ears
 - These bias may generate serious issues
 - In 2014, Amazon used a deep network to analyze applicants' resume
 - This network massively rejected women's resume
 - The network was trained on resumes collected during 10 past years, most of them were from men...

Part II-1 : the deep networks

- **The dataset bias**

- Example de biais :
 - Network trained to recognize wolves from husky
 - High success ratio
 - But some (obvious) errors !
- After analyzing features, it appeared that the network only detect snow on the background !
 - In the dataset, a large majority of wolf pictures have snow in the background



Predicted: **wolf**
True: **wolf**



Predicted: **husky**
True: **husky**



Predicted: **wolf**
True: **wolf**



Predicted: **husky**
True: **husky**



Predicted: **wolf**
True: **wolf**

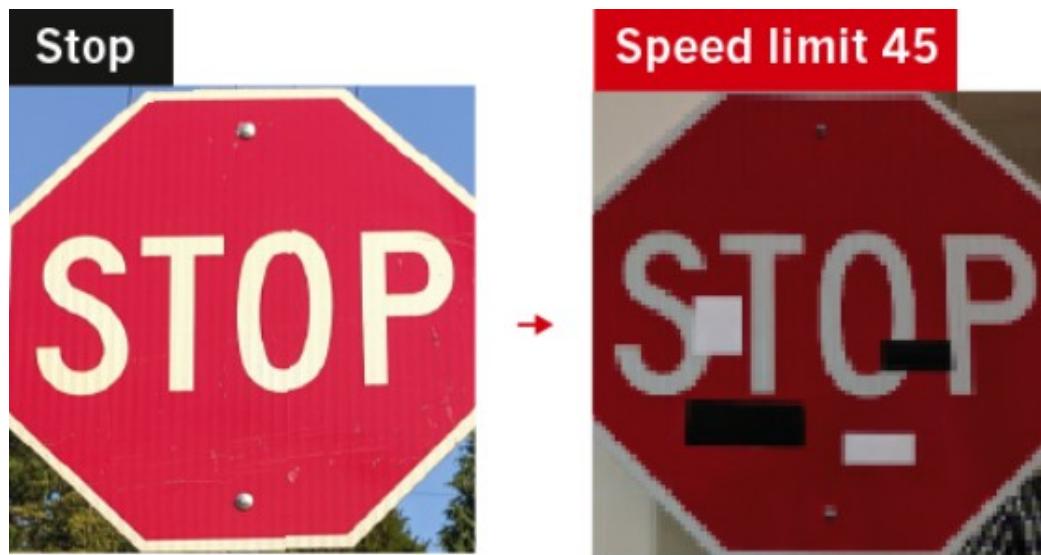


Predicted: **wolf**
True: **husky**

Part II-1 : the deep networks

- **Security**

- Training a network with a biased dataset not only reduce its reliability, but also makes it more vulnerable to attacks



Part II-1 : the deep networks

- **Security**

- It is also possible to ‘manipulate’ the network to modify its output

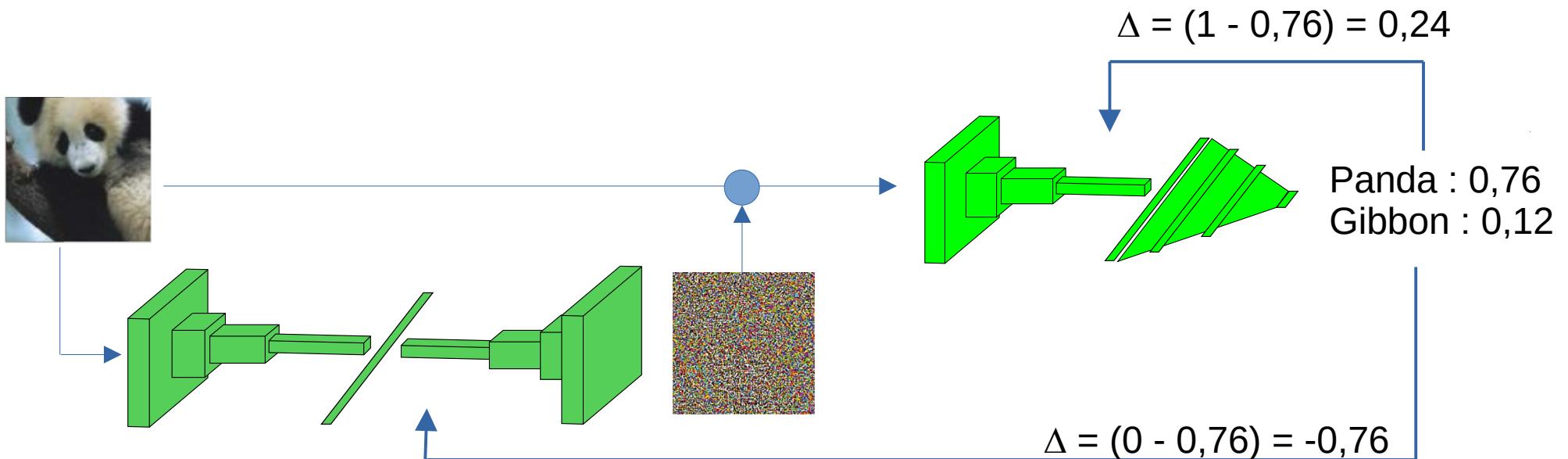


- The added noise disrupted low level features, which also interfere with higher level features, and thus, changed the result.
 - How this variation was created ?

Part II-1 : the deep networks

- **Manipulating a network**

- A ‘normal’ deep network recognize animals on images
- An encoder-decoder network is added to generate a ‘noise image’ that is added to the initial image
- The success of second network is related to the failure of first network



- If the two networks learn simultaneously, they have antagonist objectives !