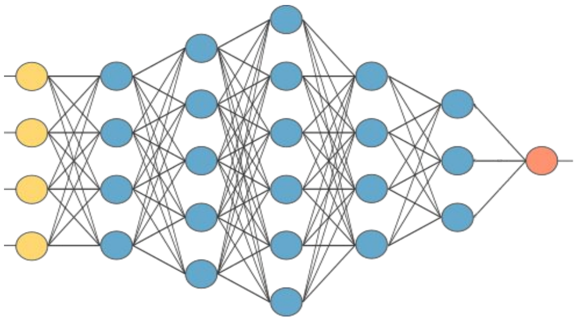
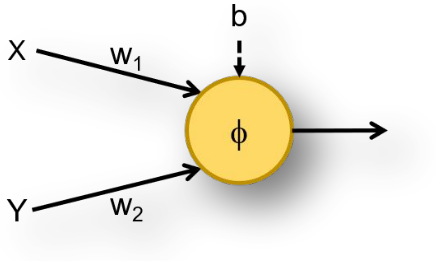
Deep Learning for image analysis
Part II - ConvNet

JB Fiche, CBS-Montpellier & Plateforme MARS-MRI
Francesco Pedaci, CBS-Montpellier
Volker Bäcker, CRBM & MRI
Cédric Hassen-Khodja, CRBM & MRI

Recap - vocabulary





activation Function
generalization
overfitting
gradient descent
ground truth
hyperparameter
matplotlib
neuron
one-hot encoding
learning rate
Backpropagation
Categorical Cross-Entropy
CNN, ConvNet
Dropout
Keras
Max-Pooling
MNIST
Momentum
Nonlinearity
ReLU
SGD
softmax
TensorFlow
Vanishing Gradient Problem
VGG16
data augmentation
transfer learning
epoch
weights
bias
hidden layer
batch size
classification
regression
convolution
loss function
testing set
validation set
training set
deep
optimizer
fully connected layer

Goal of the training :

- Understand what an **Artificial Neural Network (ANN)** is and what are the main parameters to characterize them

- What is a **Convolutional Neural Network (CNN)** and why is it used for image processing

- What are the **fundamentals for building and training a CNN using Keras**

- Understand the **most common applications** and **where to find the tools for your applications**

Outline :

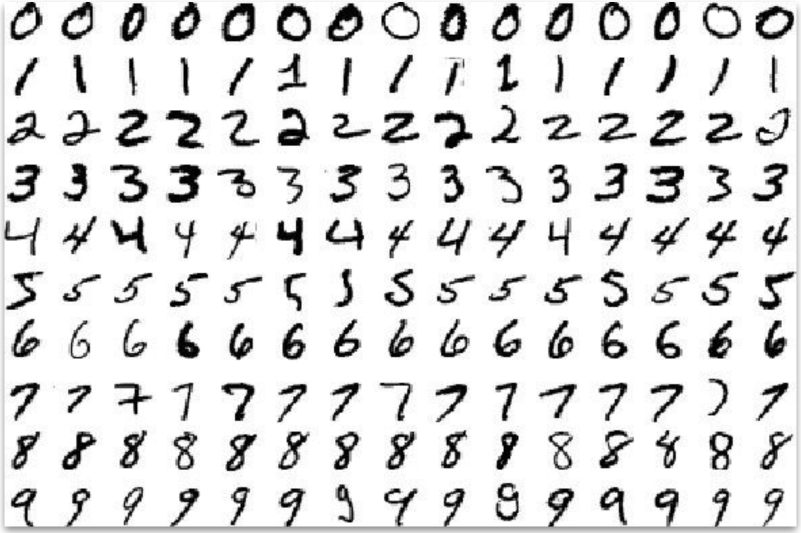
I. **Example #4 :**

    A. create an image classifier with a dense network
    B. Introduction to convolutional network

II. Example #5 : train and optimize your own convNet

III. Real-case examples

Working with the MNIST database :



Large database of **handwritten digits** that is commonly used for machine learning.

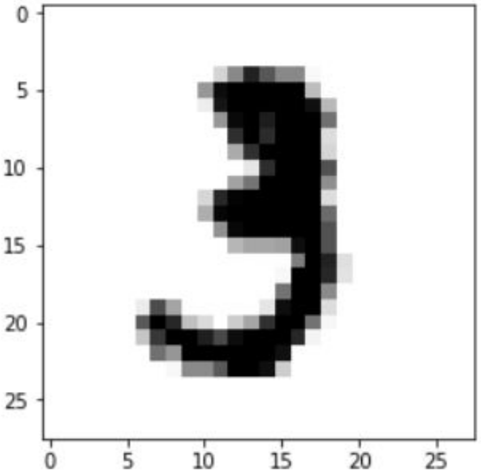
It contains :

- 60.000 images for training
- 10.000 images for testing/validation

Ex4_MNIST_dense_vs_convolutional_nn.ipynb

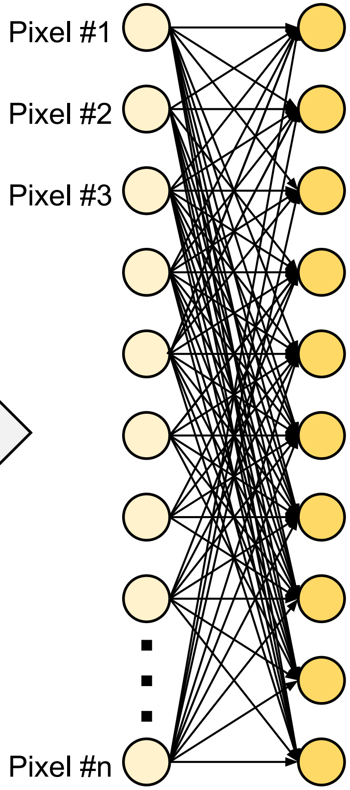
https://en.wikipedia.org/wiki/MNIST_database

Image classification with a dense network :



28x28 pixels image

784 inputs

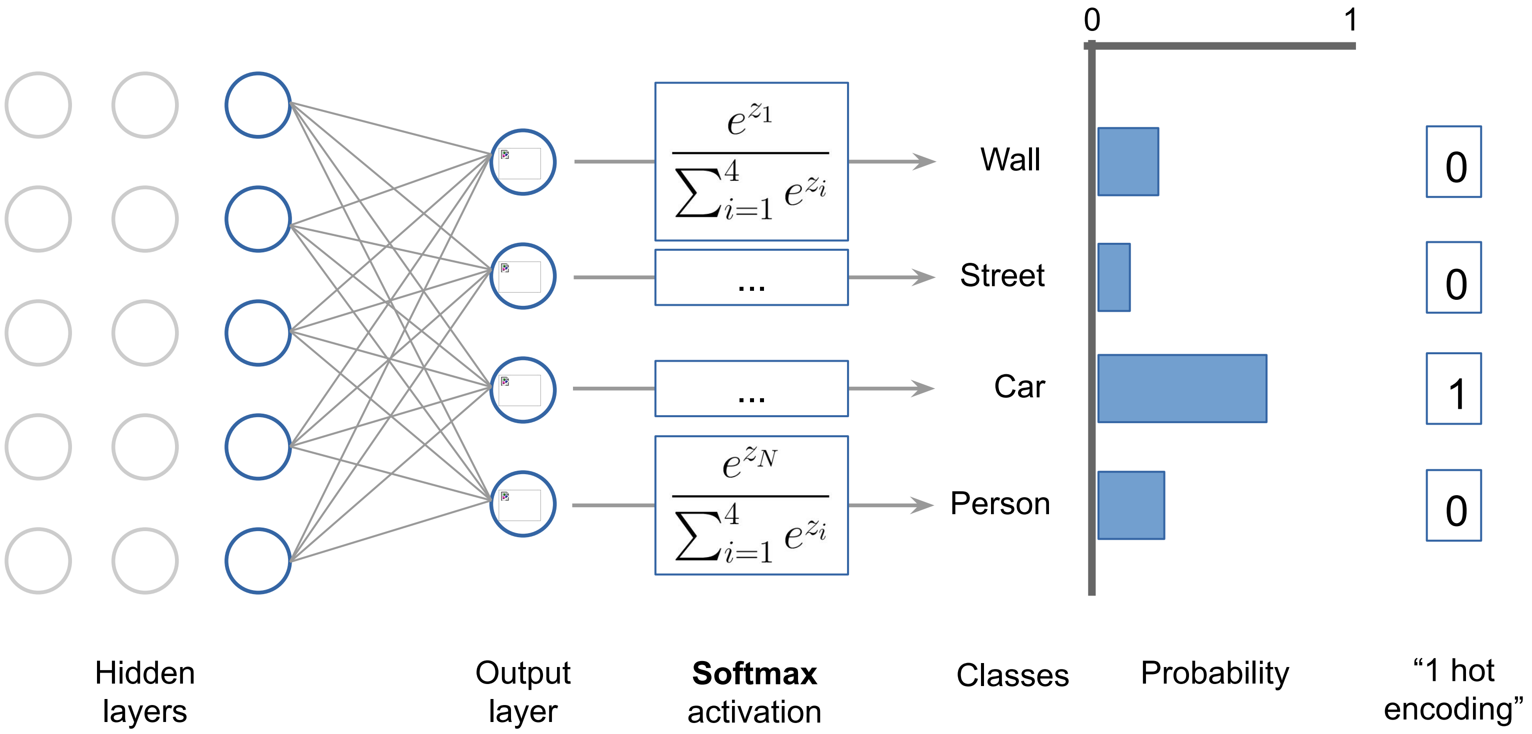


First layer 10 neurons

Number of parameters:

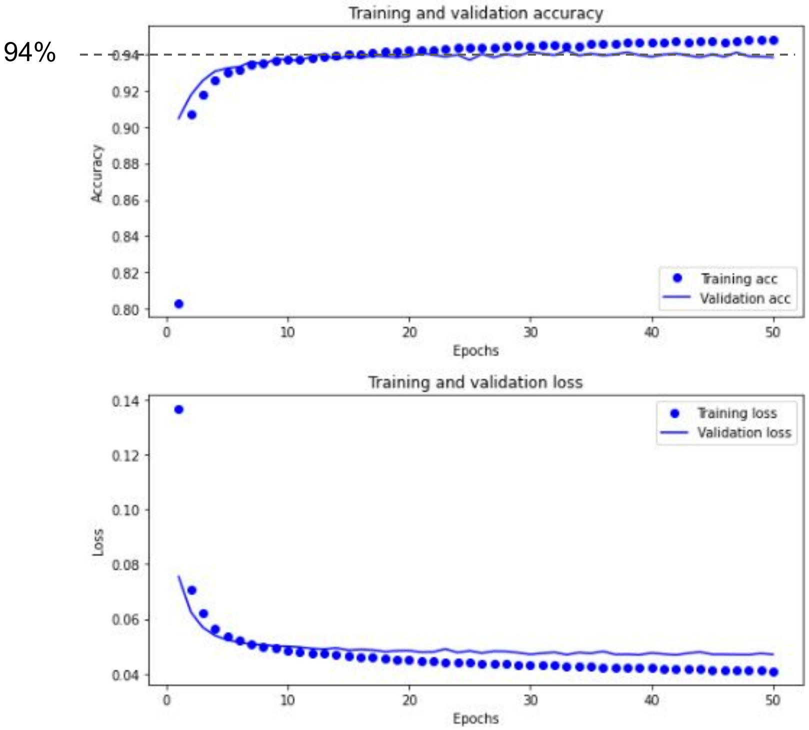
784*10+10 = 7850

Classifier with multiple classes : **softmax** activation

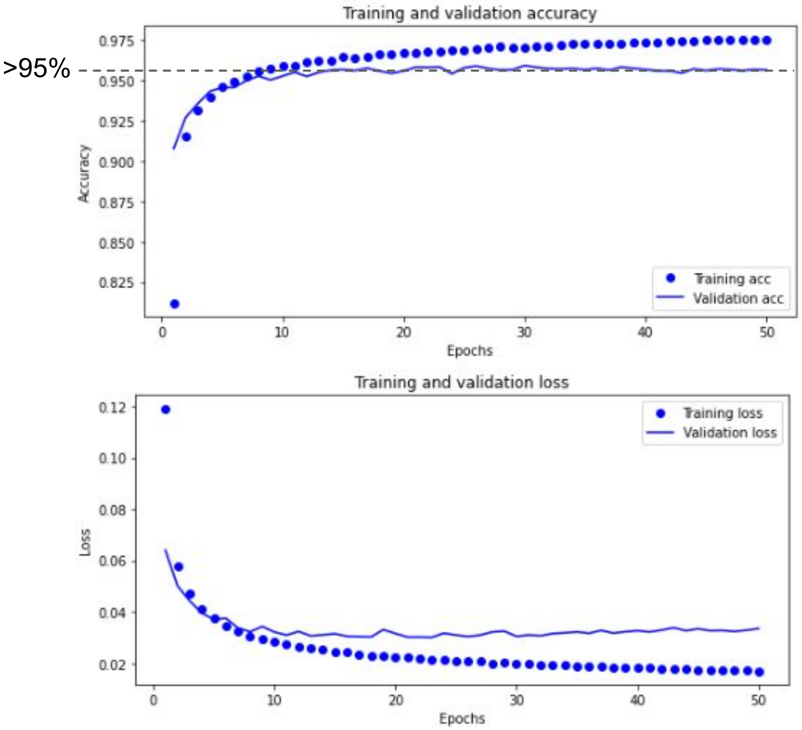


MNIST classification with a dense network :

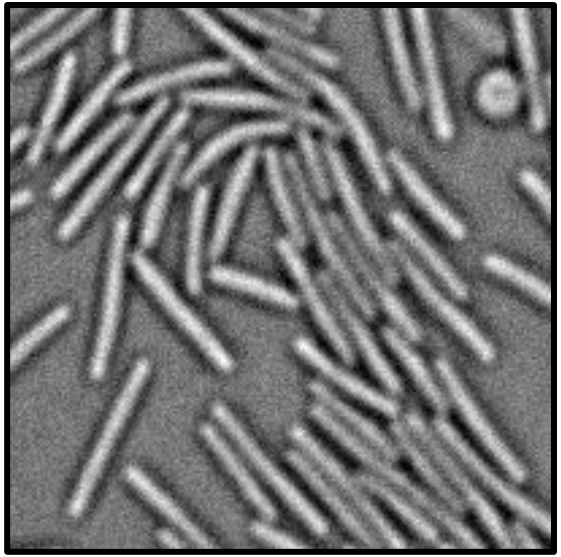
2 layers of 10 neurons : **7960 parameters**



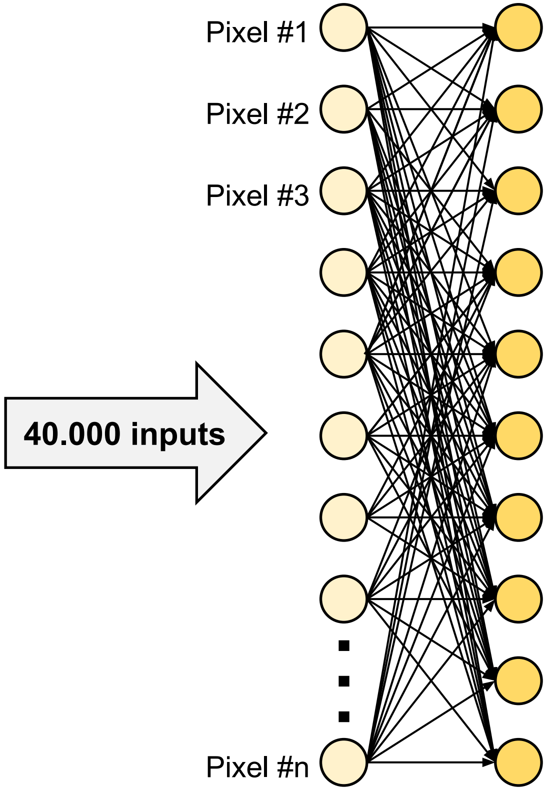
3 layers of 15 neurons : **12175 parameters**



Dense network for large images?



200x200 pixels image



First layer 10 neurons

Number of parameters:

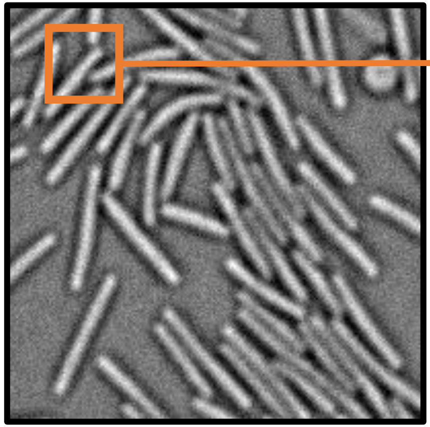
40.000*10+10 = 400.010 (!!!)

Image analysis with convolutional network :

Densely connected networks are **not suited** for image analysis:

- Too many parameters, even for small images
- Loses the local information around each pixel
- The network architecture depends on the image size

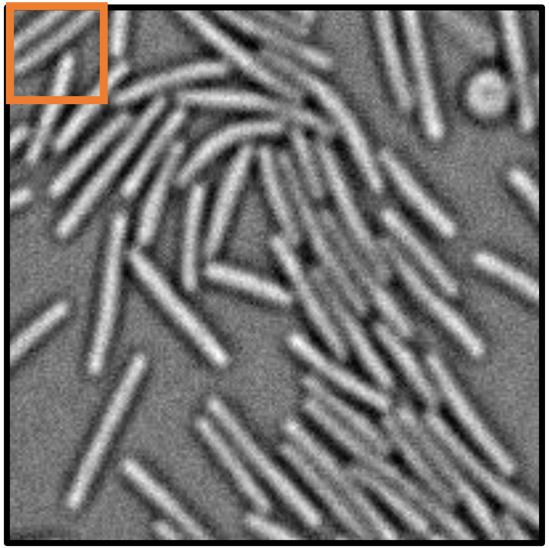
Go local



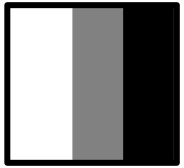
Learn local representations with **convolutional** network.

What is convolution?

Input image



Kernel /
convolution matrix
/ filter

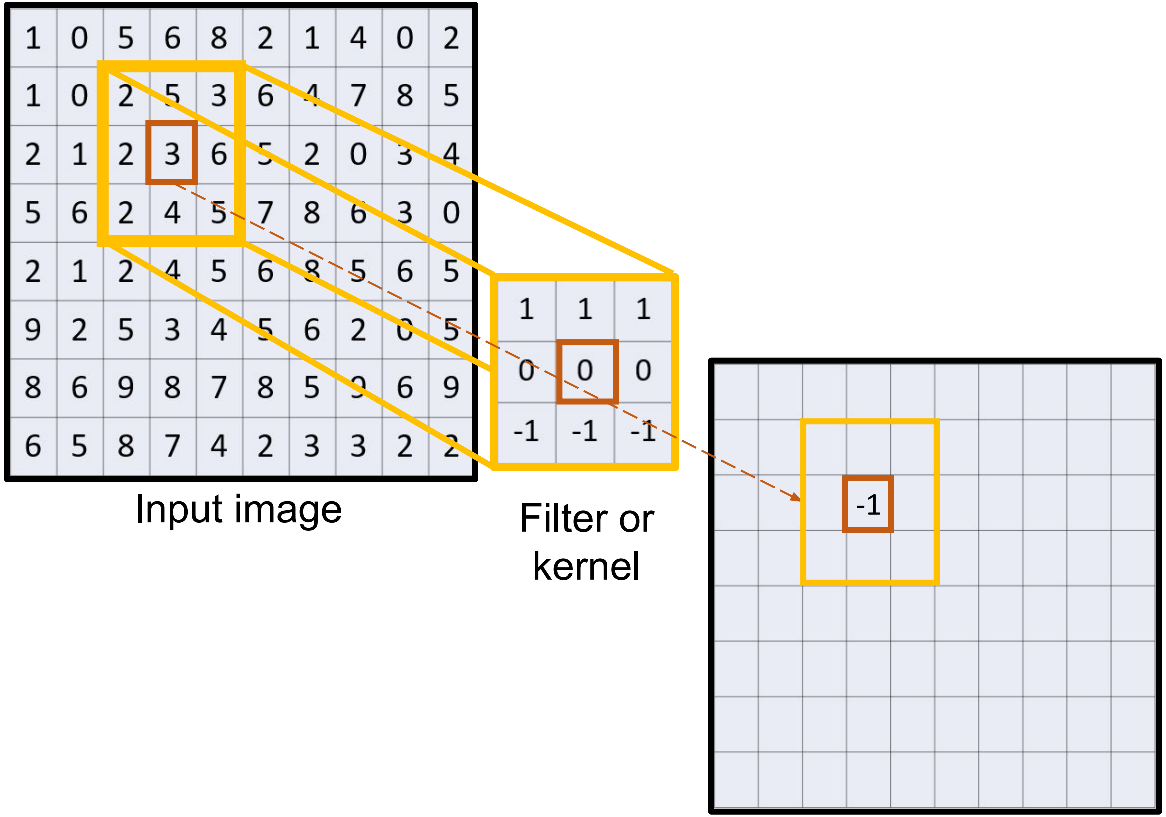


3x3

Convolution

Output image : highlight **vertically oriented objects**

Convolution operation in one scheme :



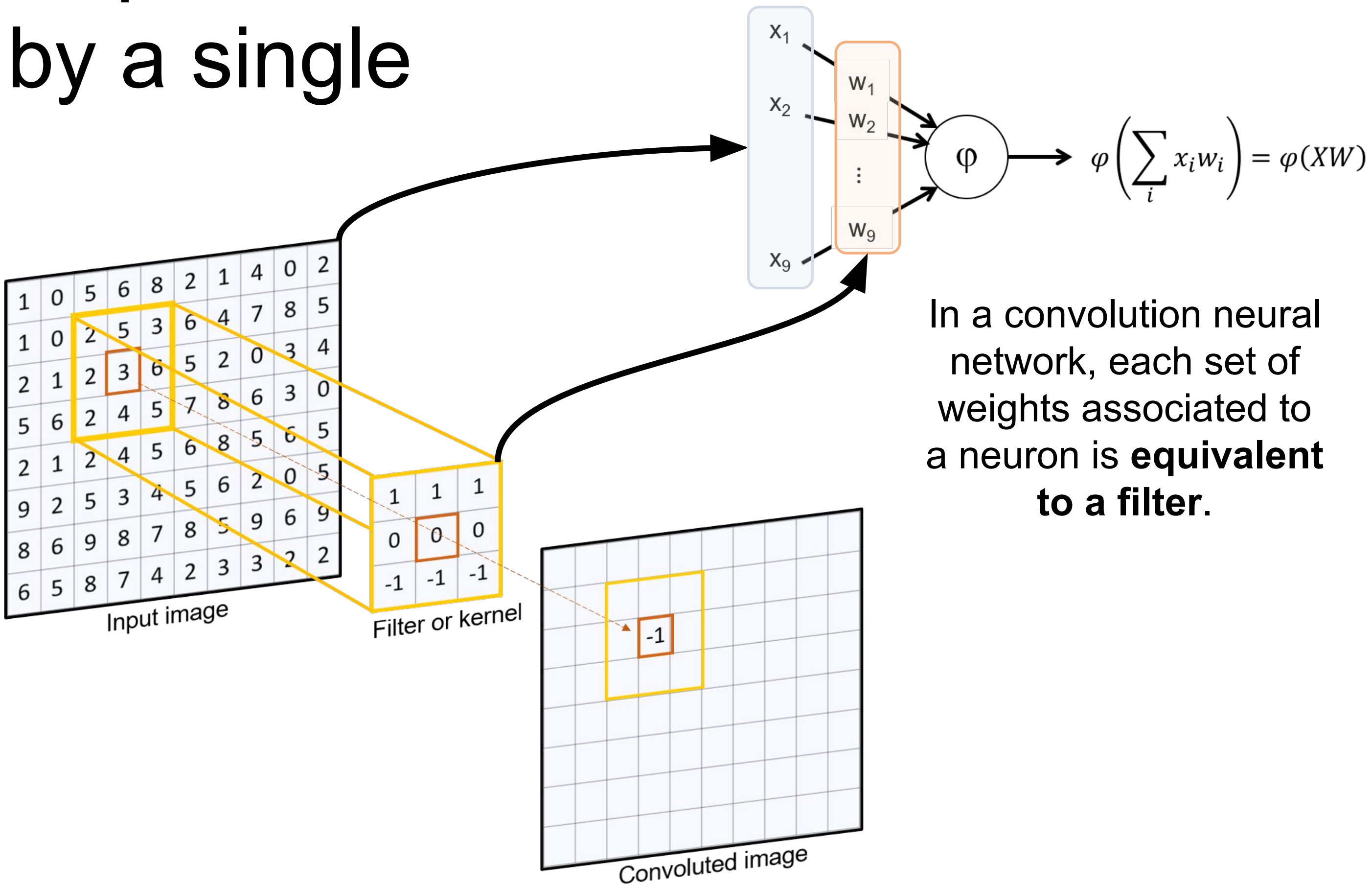
Input image

Filter or kernel

Convoluted image

$$
\begin{aligned}
& 2 \times 1 \\
+\ & 5 \times 1 \\
+\ & 3 \times 1 \\
+\ & 2 \times 0 \\
+\ & 3 \times 0 \\
+\ & 6 \times 0 \\
+\ & 2 \times -1 \\
+\ & 4 \times -1 \\
+\ & 5 \times -1 \\
\hline
=\ & -1
\end{aligned}
$$

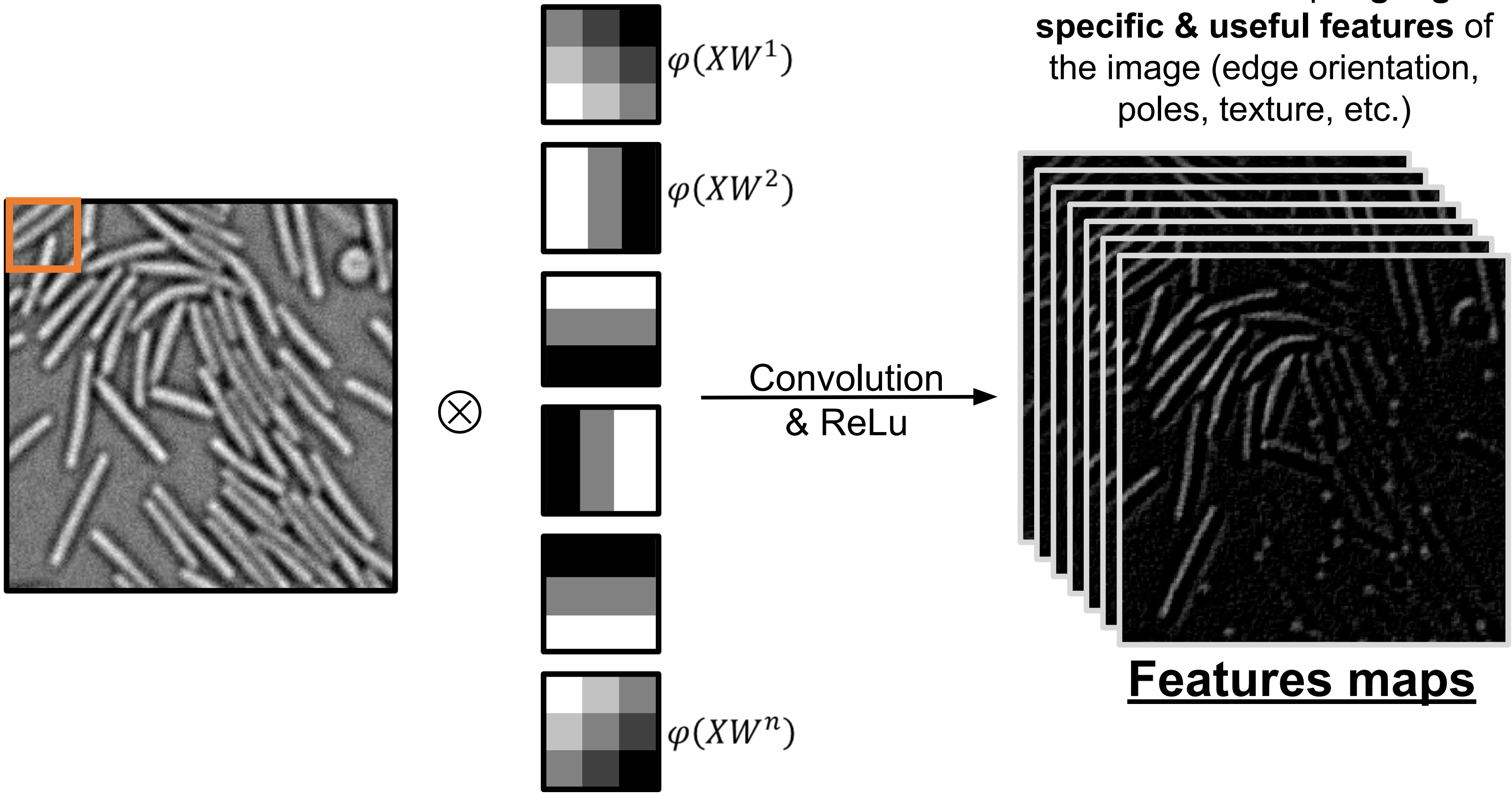
Convolution operation performed by a single neuron:



In a convolution neural network, each set of weights associated to a neuron is **equivalent to a filter**.

Features map :

Each filters will help **highlights specific & useful features** of the image (edge orientation, poles, texture, etc.)



Features maps

ConvNet syntax :

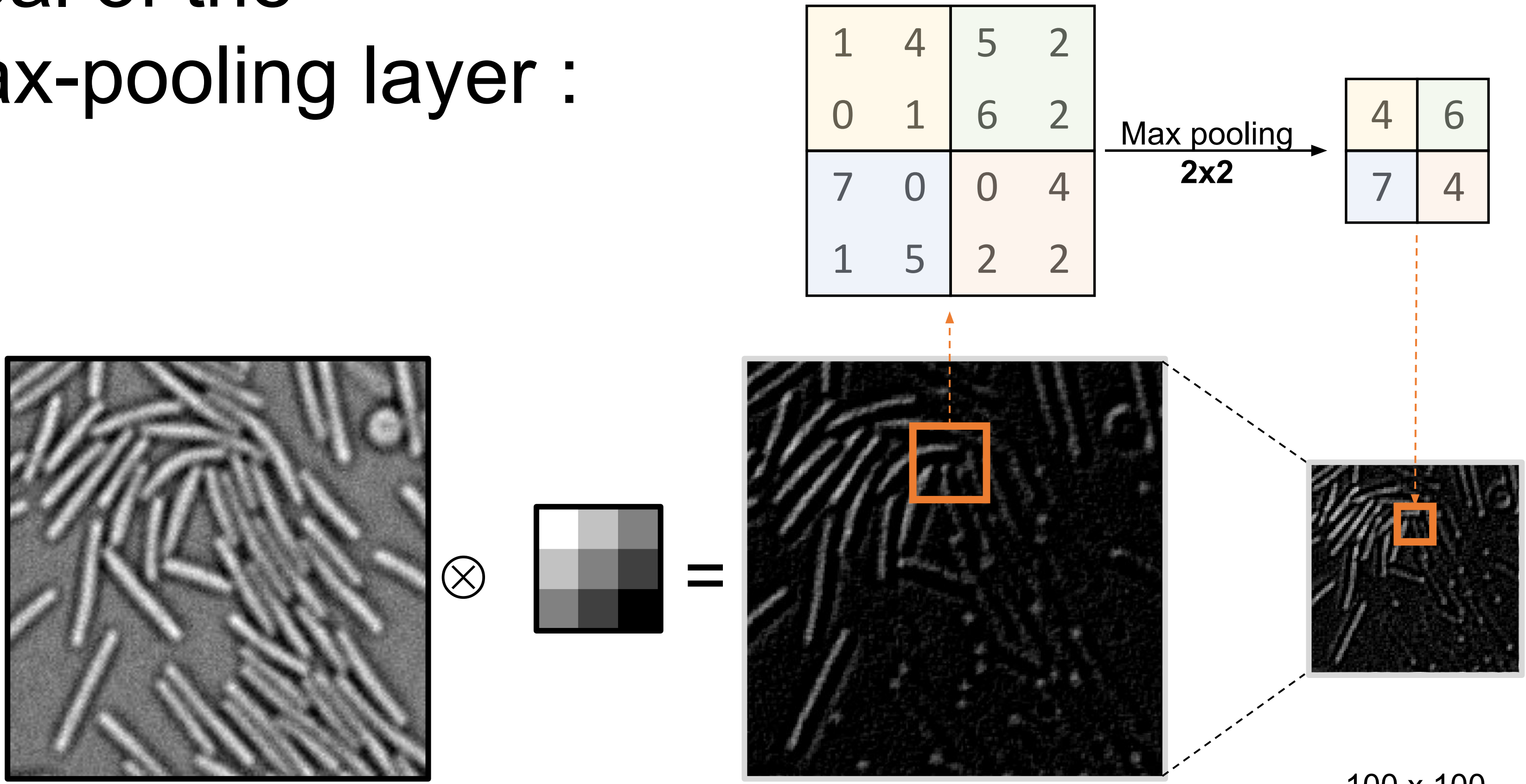
Convolution part

Dense part

```
modelCNN = Sequential([

    # Convolution Layer 1
    Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1)), # 16 different 3x3 kernels -- so 32 feature maps
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    # Convolution Layer 2
    Conv2D(16, (3, 3), activation='relu'), # 16 different 3x3 kernels
    MaxPooling2D(pool_size=(2, 2)),

    # Convolution Layer 3
    Conv2D(16, (3, 3), activation='relu'), # 16 different 3x3 kernels

    Flatten(), # Flatten final 3x3x16 output matrix into a 144-length vector

    # Fully Connected Layer 4
    Dense(10), # 10 FCN nodes
    Activation('relu'),
    Dense(10), # Necessary for the last layer since we have 10 classes
    Activation('softmax'),
])
```

Convolution layer   MaxPooling layer   Dense layer

Goal of the max-pooling layer :



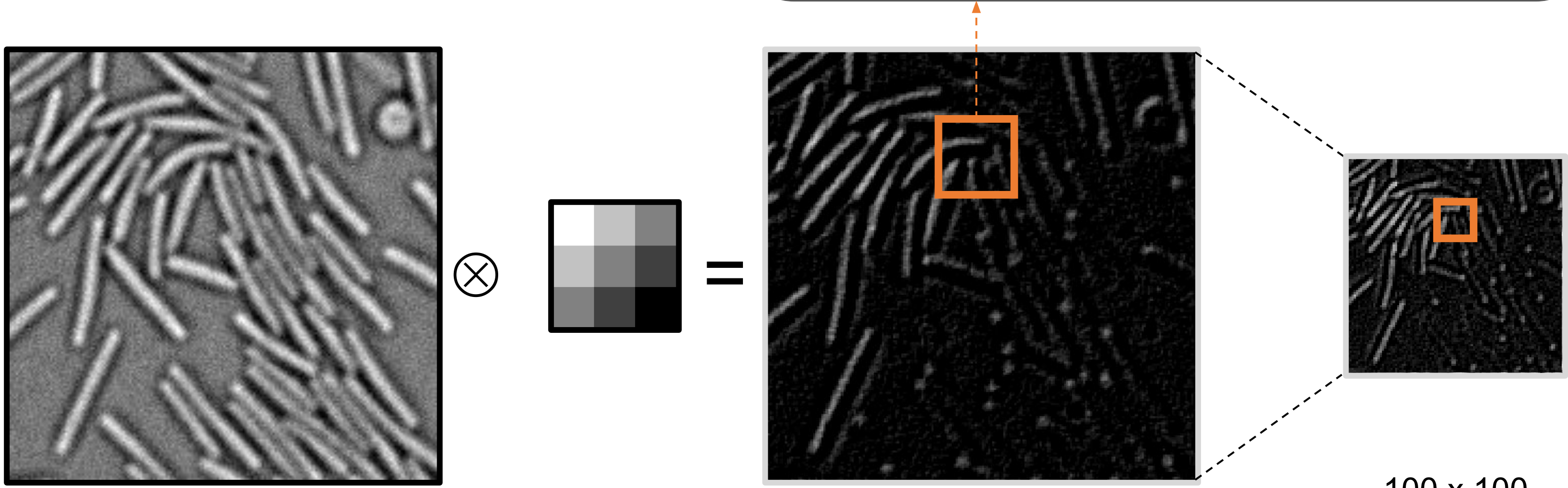
200 x 200 pixels

200 x 200 pixels

100 x 100 pixels

Goal of the max-pooling layer :

1. **Reduce the spatial resolution** of the feature maps while keeping only the most relevant information
2. **Lowering memory and computing requirements**
3. Create **translation invariance**



200 x 200 pixels

200 x 200 pixels

100 x 100 pixels

MNIST classification with a ConvNet :

```
modelCNN = Sequential([

    # Convolution Layer 1
    Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=(2, 2)),

    # Convolution Layer 2
    Conv2D(16, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),

    # Convolution Layer 3
    Conv2D(16, (3, 3), activation='relu'),

    Flatten(),

    # Fully Connected Layer 4
    Dense(15),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

Model: "sequential_2"

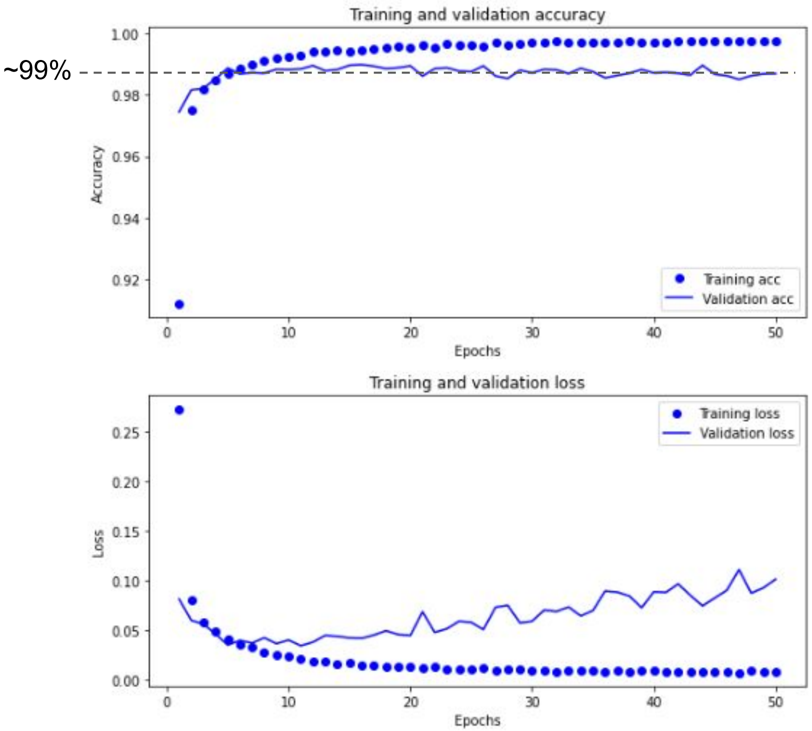
Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 26, 26, 16)	160
max_pooling2d_4 (MaxPooling2	(None, 13, 13, 16)	0
conv2d_7 (Conv2D)	(None, 11, 11, 16)	2320
max_pooling2d_5 (MaxPooling2	(None, 5, 5, 16)	0
conv2d_8 (Conv2D)	(None, 3, 3, 16)	2320
flatten_2 (Flatten)	(None, 144)	0
dense_4 (Dense)	(None, 15)	2175
activation_4 (Activation)	(None, 15)	0
dense_5 (Dense)	(None, 10)	160
activation_5 (Activation)	(None, 10)	0

Total params: 7,135
Trainable params: 7,135
Non-trainable params: 0

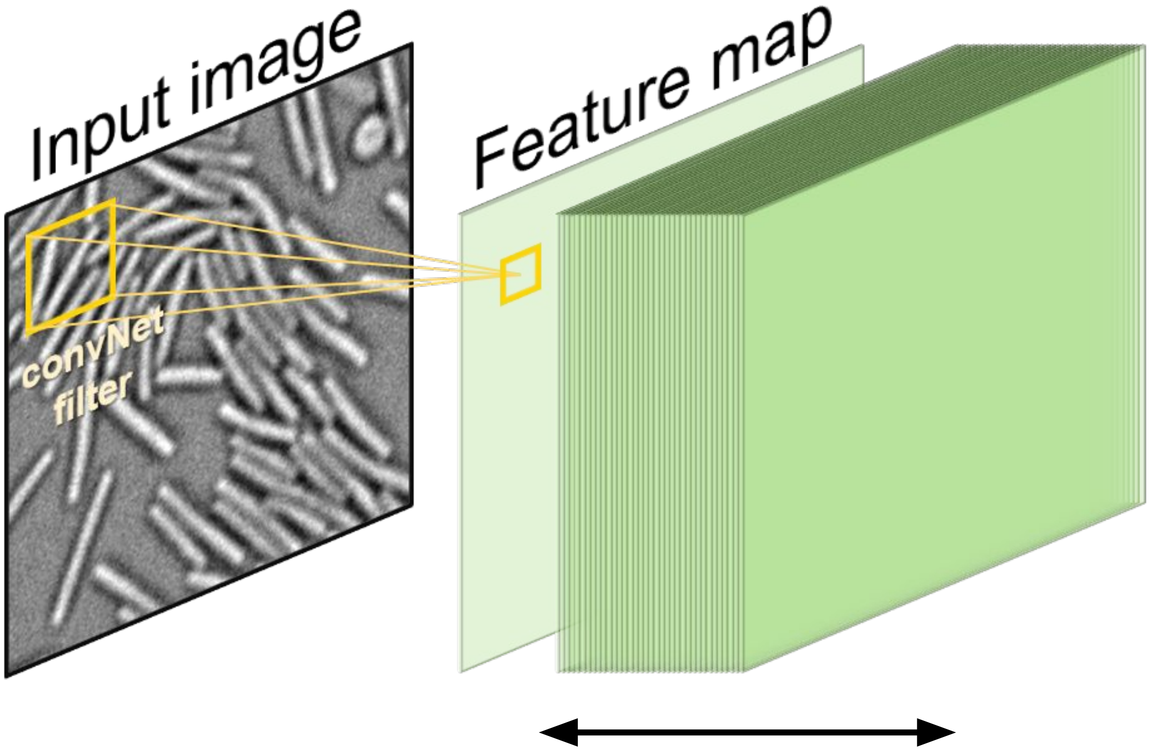
MNIST classification with a ConvNet :

3 convolution layers of 16 kernel and 2 dense layers of 10 neurons : **7135 parameters**

```python
modelCNN = Sequential([

    # Convolution Layer 1
    Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=(2, 2)),

    # Convolution Layer 2
    Conv2D(16, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),

    # Convolution Layer 3
    Conv2D(16, (3, 3), activation='relu'),

    Flatten(),

    # Fully Connected Layer 4
    Dense(15),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

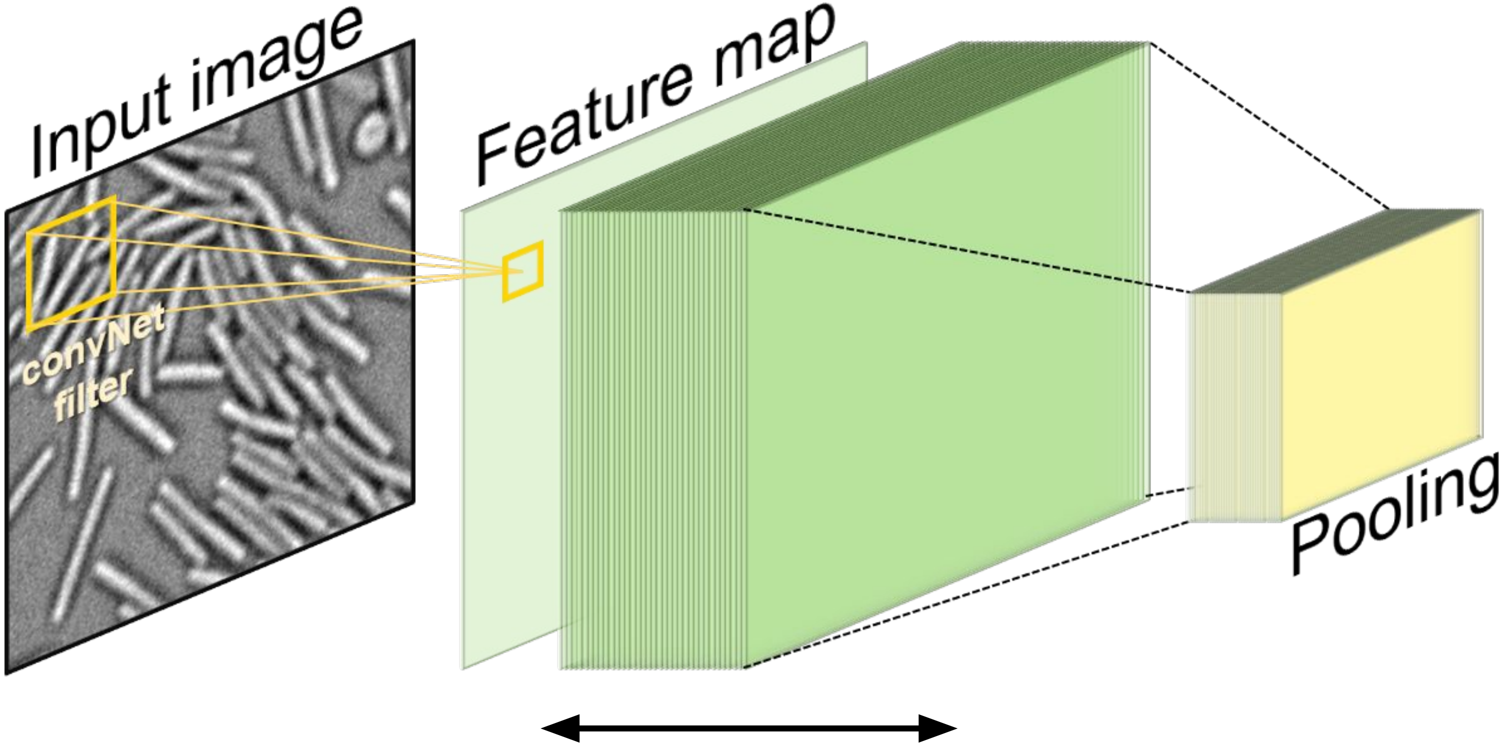


ConvNet summary :



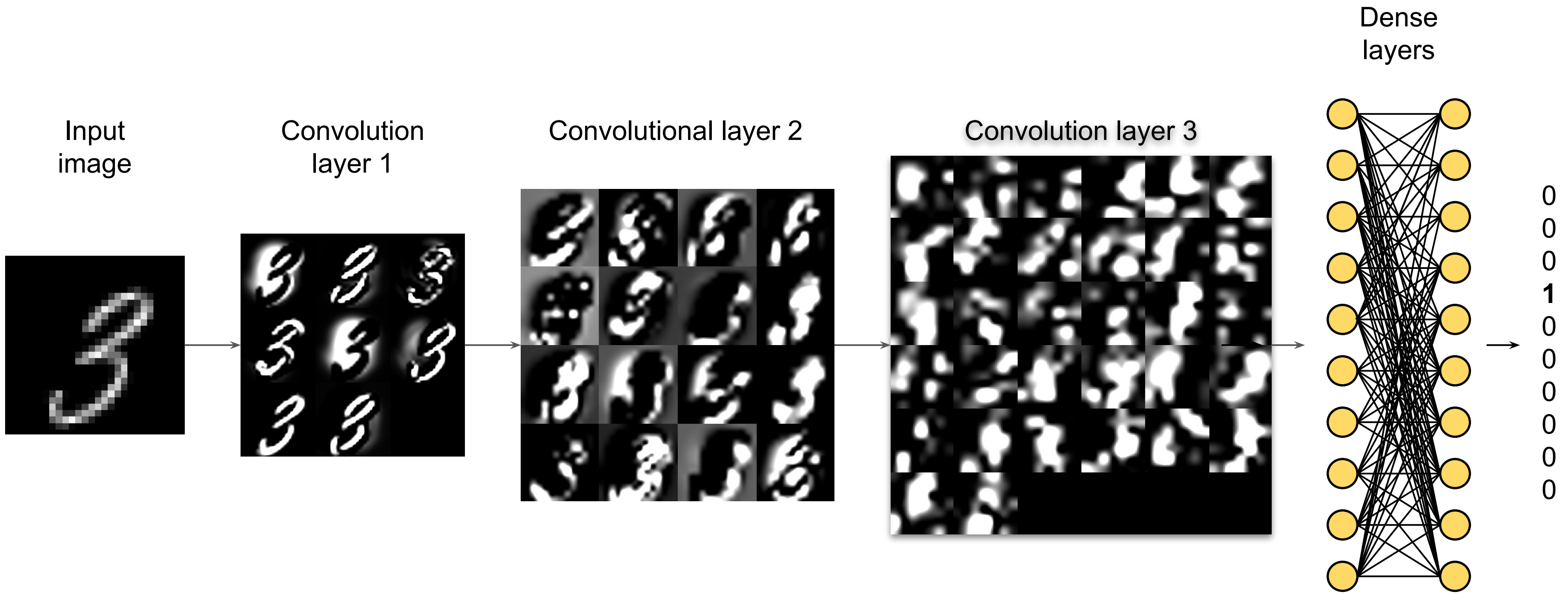
Feature maps : there is as **many maps as neurons in the convolution layer**

ConvNet summary :



Feature maps : there is as **many maps as neurons in the convolution layer**

ConvNet summary :



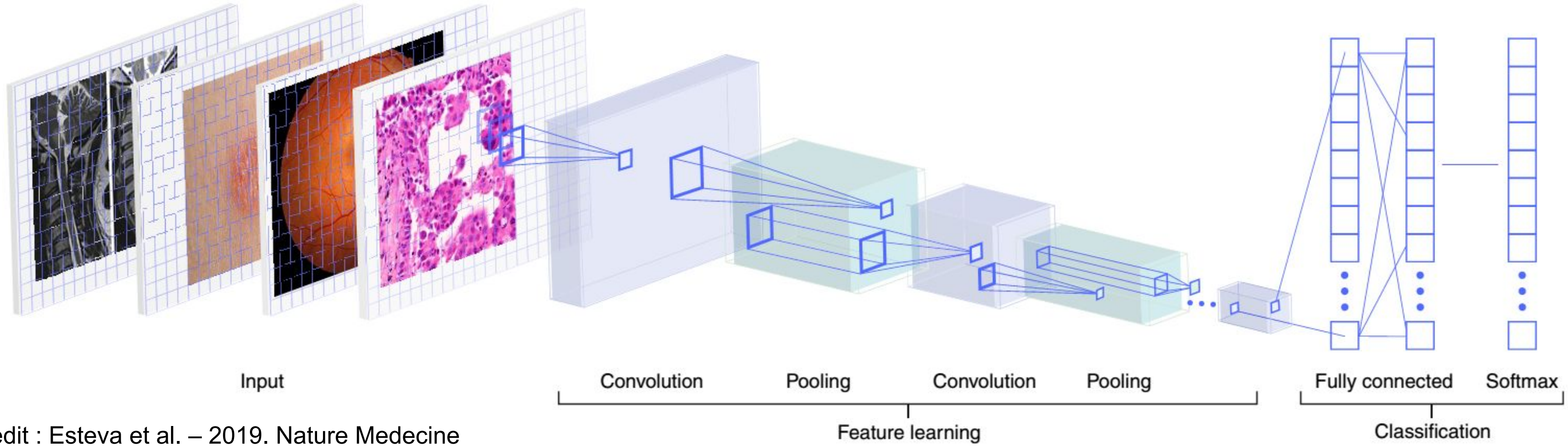
ConvNet architecture for image classification :

The first part of the network is **using convolution layers to learn the features**

- **Convolution layers → features extraction**
- **Pooling → downsampling**

The second part is classifying those features using **densely connected layers** in order to predict the right output.

- Lots or parameters → **heavy on the memory**
- Image input size is fixed → **not flexible**



Pic credit : Esteva et al. – 2019. Nature Medecine

Outline :

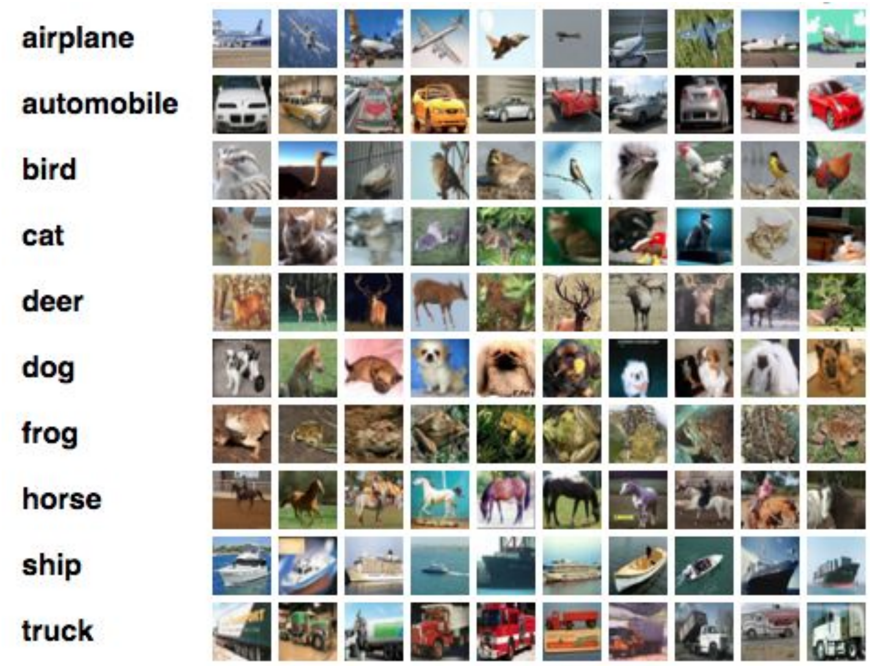
I. Example #4 :

   A. create an image classifier with a dense network
   B. Introduction to convolutional network

II. Example #5 : train and optimize your own convNet

III. Real-case examples

Optimize your own classifier :



The CIFAR database is a collection of **RGB images** classified into **10 classes** .

It contains :

- 60.000 images for training. For each class, there are 6000 images.
- 10.000 images for testing/validation

Ex5_CIFAR_convolutional_nn.ipynb

Start with a good baseline model :

A good practice is to **start working with a network architecture that is know to be efficient for your problem**. For example, the VGG16 architecture is easy to implement and well documented for image classification.

Convolution block #1

Convolution block #2

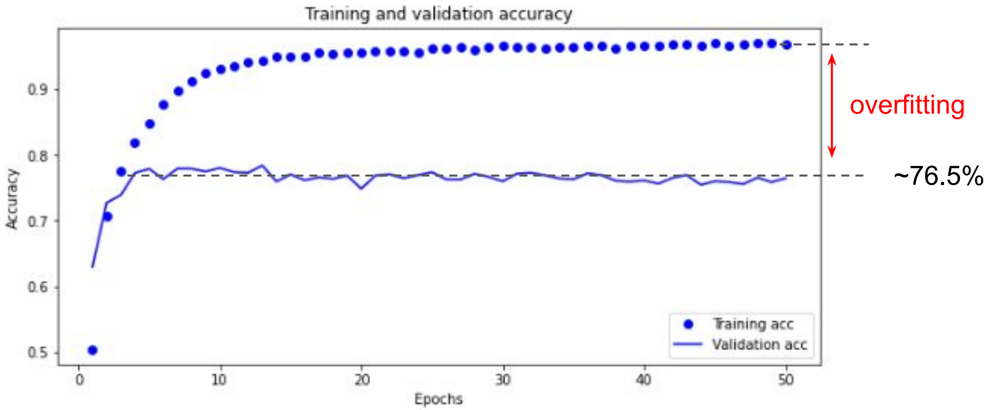
Convolution block #3

Fully connected network

```
modelCNN = Sequential([

    # Convolution Layer 1 &2
    Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)), # 64 different 3x3 kernels
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    # Convolution Layer 3 &4
    Conv2D(128, (3, 3), activation='relu', padding='same'), # 128 different 3x3 kernels
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    # Convolution Layer 5 &6
    Conv2D(256, (3, 3), activation='relu', padding='same'), # 256 different 3x3 kernels
    Conv2D(256, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    Flatten(), # Flatten final 3x3x128 output matrix into a 1152-length vector

    # Fully Connected Layer 4
    Dense(128), # 128 FCN nodes
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

VGG1 : Number of trainable parameters : <u>1,441,738</u>

Start with a good baseline model :



We observe that the training and validation loss & accuracy are diverging after epoch #5. **The network is no longer learning new useful features**.

- Overfitting
- Validation loss & accuracy are noisy
- The global accuracy of the network is ~76.5%

Reduce the network size :

A good practice is to **start working with a network architecture that is know to be efficient for your problem**. For example, the VGG16 architecture is easy to implement and well documented for image classification.

Convolution block #1

Convolution block #2

Convolution block #3

Fully connected network

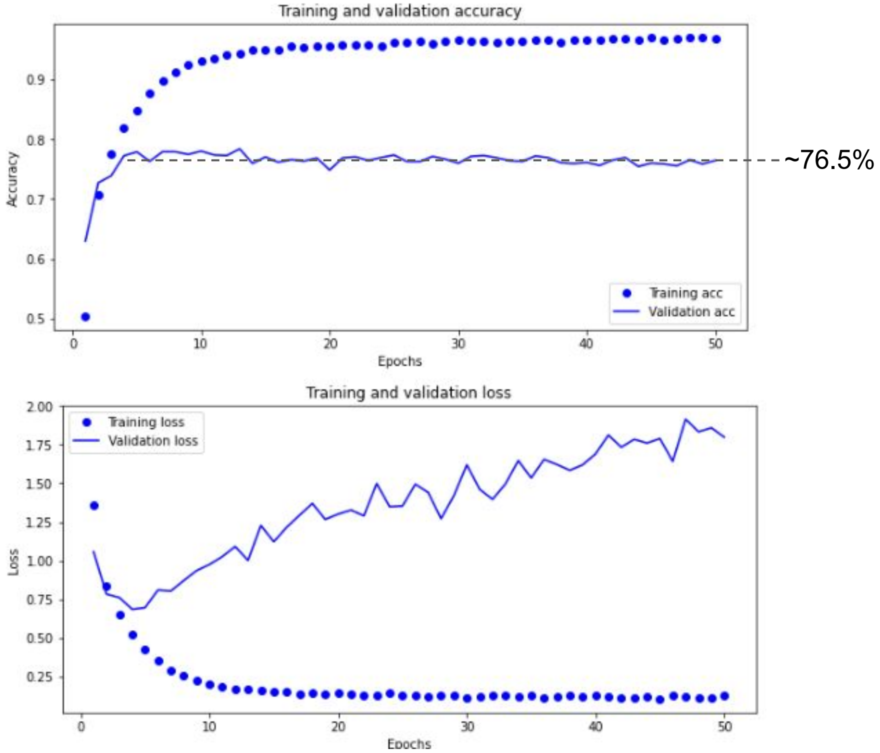
```
modelCNN = Sequential([

    # Convolution Layer 1 &2
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)), # 32 different 3x3 kernels
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    # Convolution Layer 3 &4
    Conv2D(64, (3, 3), activation='relu', padding='same'), # 64 different 3x3 kernels
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    # Convolution Layer 5 &6
    Conv2D(128, (3, 3), activation='relu', padding='same'), # 128 different 3x3 kernels
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    Flatten(), # Flatten final 3x3x128 output matrix into a 1152-length vector

    # Fully Connected Layer 4
    Dense(128), # 128 FCN nodes
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

VGG2 : Number of trainable parameters : 435,882

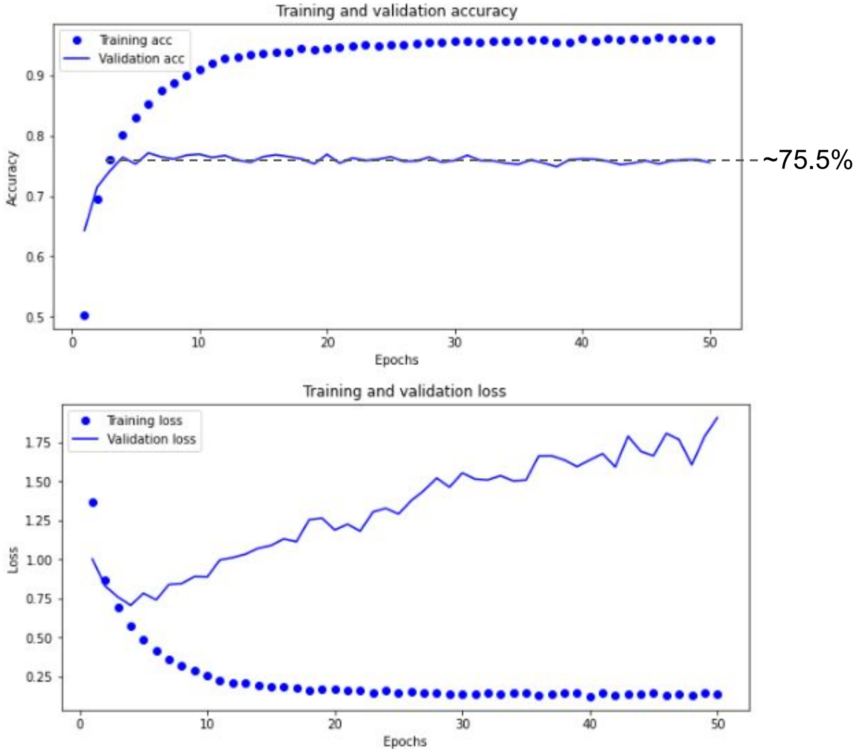
BIAS & WEIGHTS

Comparison VGG 1 & 2 :

VGG1

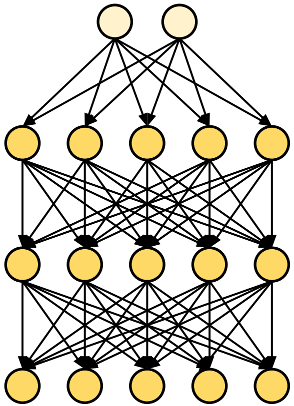


VGG2

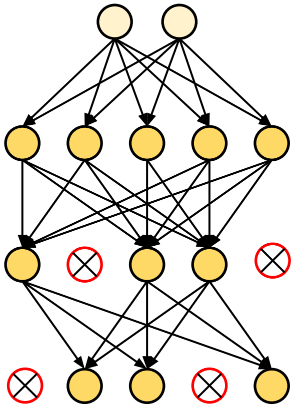


How to reduce overfitting?

- **Reduce the size of the network** … but no magical formula to determine how many layers we need
- **Dropout**, randomly “turning-off” neurons of the network



No dropout



Dropout with 40% probability

Dropout is used to avoid co-adaptation of neurons → enforce the fact that neurons should **learn and work independently**.

VGG baseline and Dropout regularization :

```
modelCNN = Sequential([

    # Convolution Layer 1 &2
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)), # 32 different 3x3 kernels
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel
    Dropout(0.2),

    # Convolution Layer 3 &4
    Conv2D(64, (3, 3), activation='relu', padding='same'), # 64 different 3x3 kernels
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel
    Dropout(0.3),

    # Convolution Layer 5 &6
    Conv2D(128, (3, 3), activation='relu', padding='same'), # 128 different 3x3 kernels
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel
    Dropout(0.4),

    Flatten(), # Flatten final 3x3x128 output matrix into a 1152-length vector

    # Fully Connected Layer 4
    Dense(128), # 128 FCN nodes
    Activation('relu'),
    Dropout(0.4),
    Dense(10),
    Activation('softmax'),
])
```

Dropout 20% → Dropout(0.2)

Dropout 30% → Dropout(0.3)

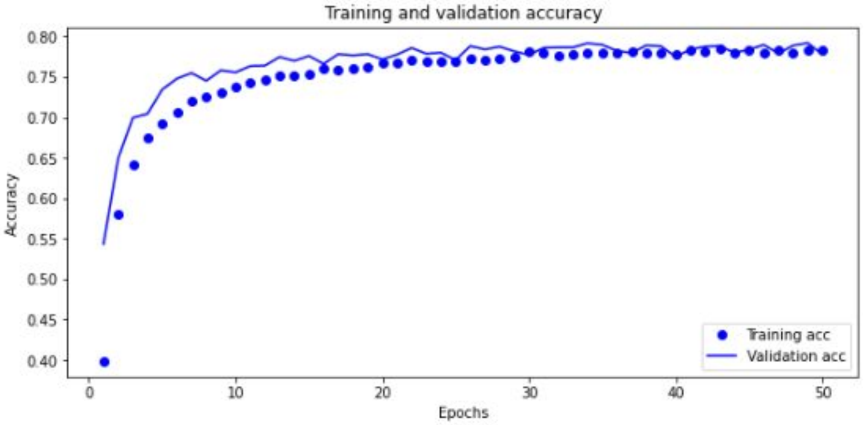
Dropout 40% → Dropout(0.4)

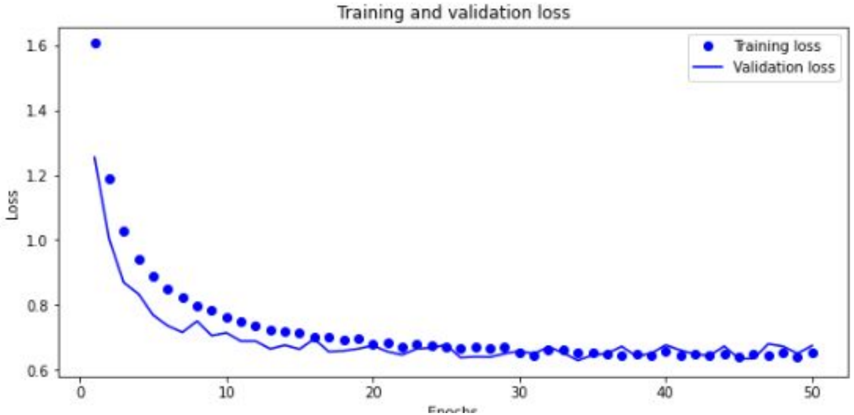
Dropout 40% → Dropout(0.4)

Number of trainable parameters : 435,882

BIAS & WEIGHTS

VGG baseline and Dropout regularization :





Adding Dropout regularization helps **reducing the overfitting** and slightly **improve the performance of the network (77.7% instead of 75.5%)**

How to reduce overfitting?

- **Reduce the size of the network** … but no magical formula to determine how many layers we need
- **Dropout**, randomly “turning-off” neurons of the network
- **Weight regularization L_1 & L_2**, a strategy to force the weights to take only small values during the training

L1/L2 regularization

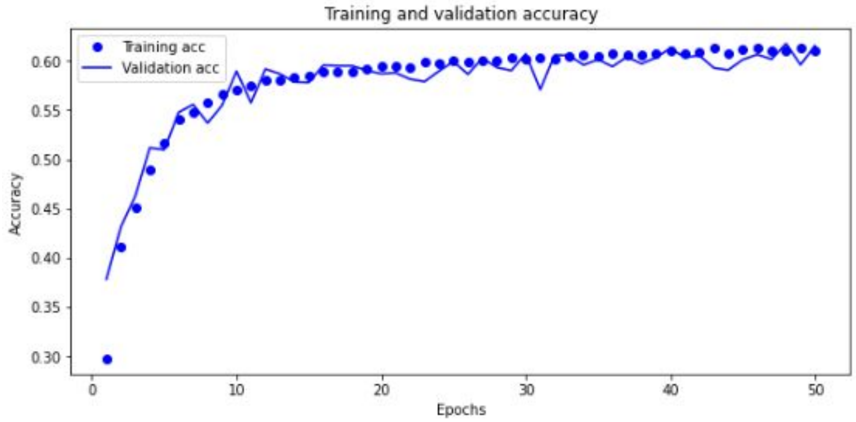
Pic credit : Moen et al. 2019. Nat. Meth.

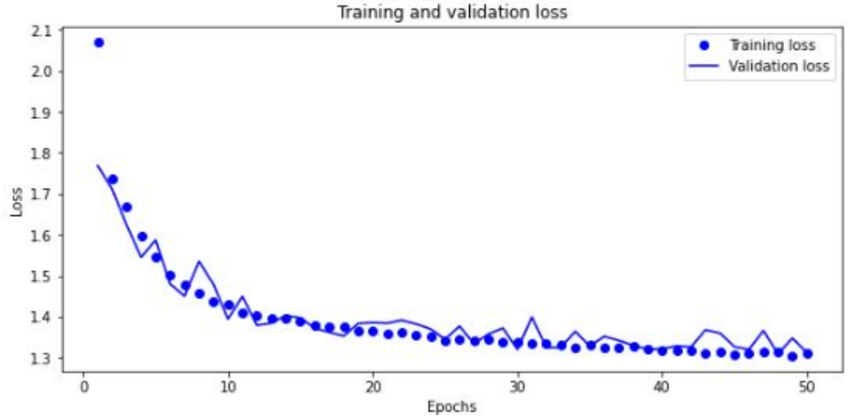
VGG baseline and L2 regularization :

```
regularizers.l2(l2=1e-4)
modelCNN = Sequential([

    # Convolution Layer 1 &2
    Conv2D(32, (3, 3), activation='relu', kernel_regularizer='l2', input_shape=(32, 32, 3)), # 32 different 3x3 kernel
    Conv2D(32, (3, 3), activation='relu', kernel_regularizer='l2'),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    # Convolution Layer 3 &4
    Conv2D(64, (3, 3), activation='relu', kernel_regularizer='l2', padding='same'), # 64 different 3x3 kernels
    Conv2D(64, (3, 3), activation='relu', kernel_regularizer='l2', padding='same'),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    # Convolution Layer 5 &6
    Conv2D(128, (3, 3), activation='relu', kernel_regularizer='l2', padding='same'), # 128 different 3x3 kernels
    Conv2D(128, (3, 3), activation='relu', kernel_regularizer='l2', padding='same'),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    Flatten(), # Flatten final 3x3x128 output matrix into a 1152-length vector

    # Fully Connected Layer 4
    Dense(128, kernel_regularizer='l2'), # 128 FCN nodes
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

Number of trainable parameters : 435,882

VGG baseline and L2 regularization :



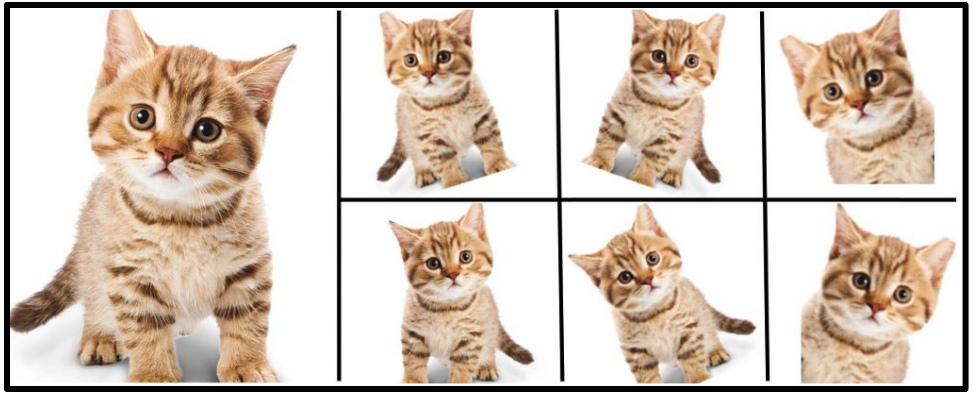


Adding L2 regularization helps **reducing the overfitting** but the overall performance of the network is getting worse (61.5% while the baseline was 75.5%).

L2 regularization is not a good method for our classification problem.

How to reduce overfitting?

- **Reduce the size of the network** … but no magical formula to determine how many layers we need
- **Dropout**, randomly “turning-off” neurons of the network
- **Weight regularization L_1 & L_2**, a strategy to force the weights to take only small values during the training
- Increase the size of the training set:
  - Add new images to the training set
  - Use **data augmentation**

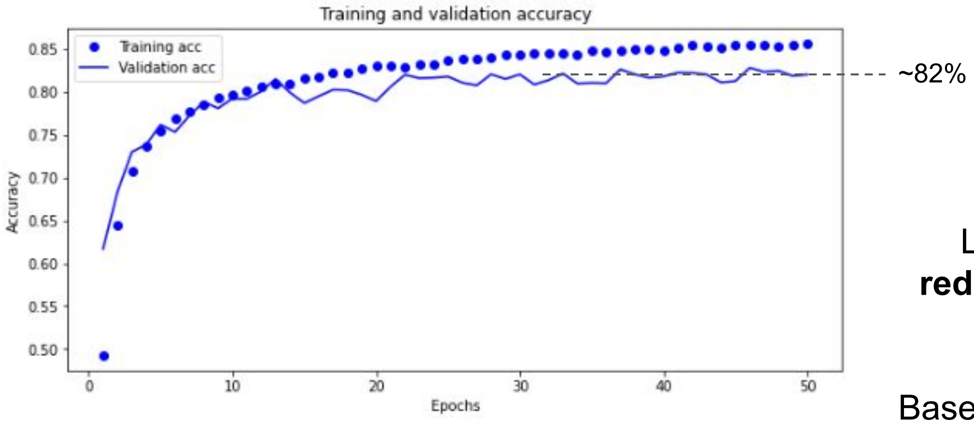


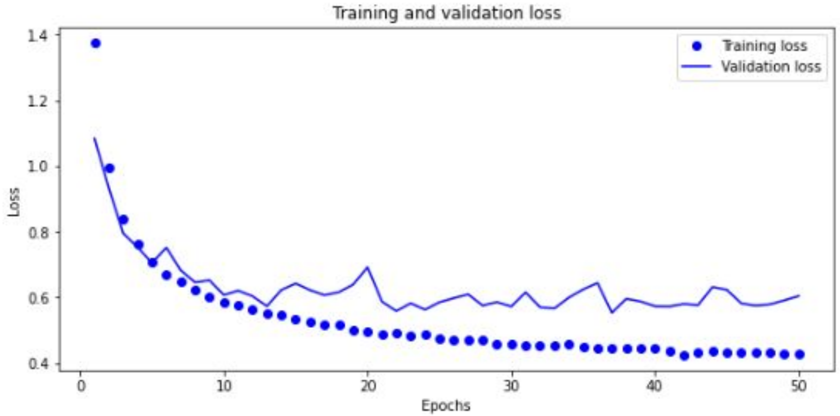
VGG baseline and image augmentation :

Create an image generator. Important to **select carefully the transformations**. Make sure they are not destroying important information (e.g: object size, shape)

```
# Create the image augmentation generator
# ---------------------------------------

datagen = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)
it_train = datagen.flow(X_train, Y_train, batch_size=32)
steps = int(X_train.shape[0] / 32)

# Compile the model defining the optimizer and the loss function
# -------------------------------------------------------------

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

# Launch the training
# ------------------

history = modelCNN.fit_generator(it_train,
                        steps_per_epoch=steps,
                        validation_data=(X_val, Y_val),
                        epochs = 50,
                        verbose = 1)
```

VGG baseline and image augmentation :





Like Dropout regularization, image augmentation helps **reducing the overfitting** and **improve the performance of the network.**

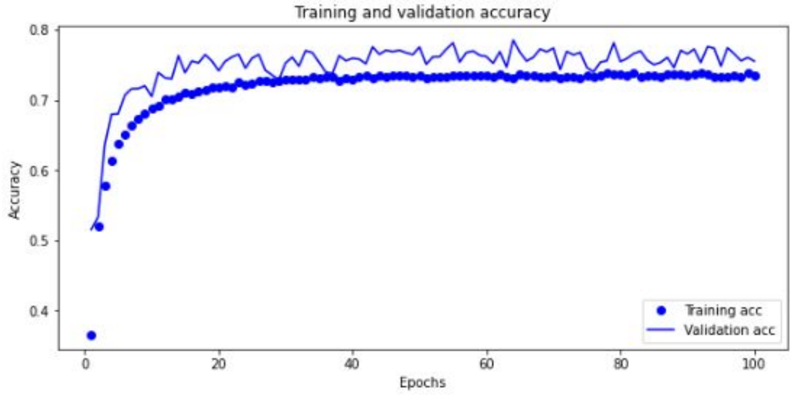
Baseline accuracy : 75.5%

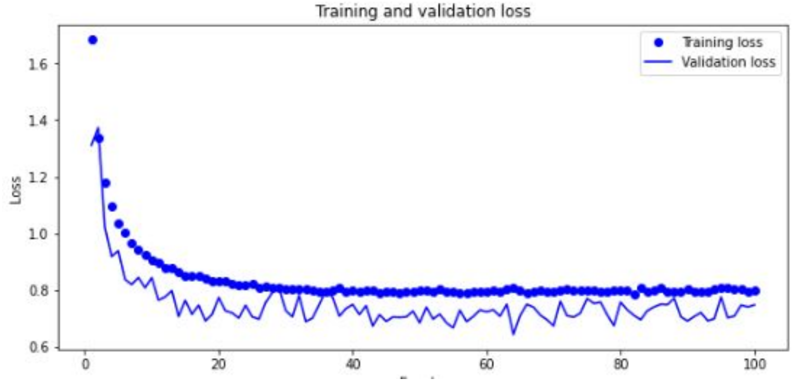
<u>Baseline + Dropout</u> : **77.7% and no overfitting**

Baseline + L2 : 61.5% and no overfitting

<u>Baseline + image augmentation</u> : **82% and less overfitting**

VGG baseline, dropout & image augmentation :





Like Dropout regularization, image augmentation helps **reducing the overfitting** and **improve the performance of the network.**

Baseline accuracy : 75.5%

Baseline + Dropout : **77.7% and no overfitting**

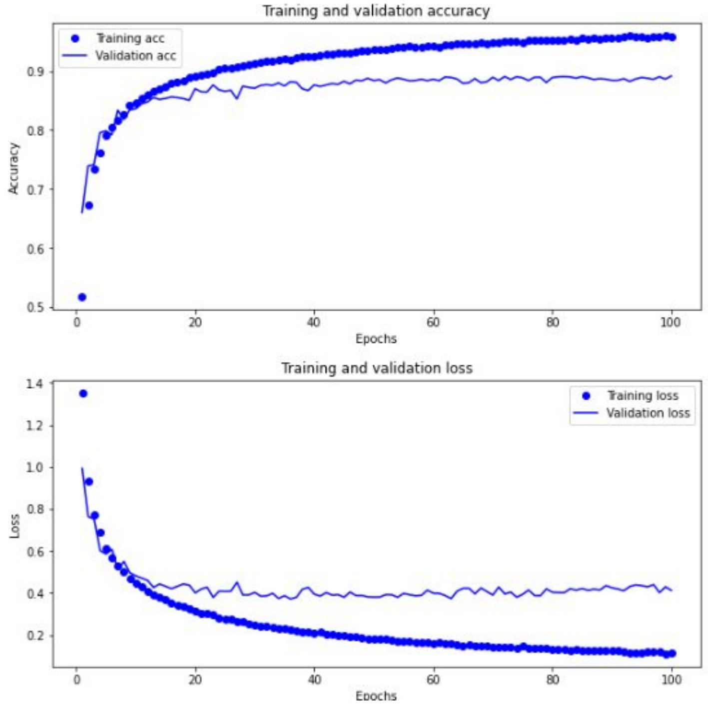
Baseline + L2 : 61.5% and no overfitting

Baseline + image augmentation : **82% and less overfitting**

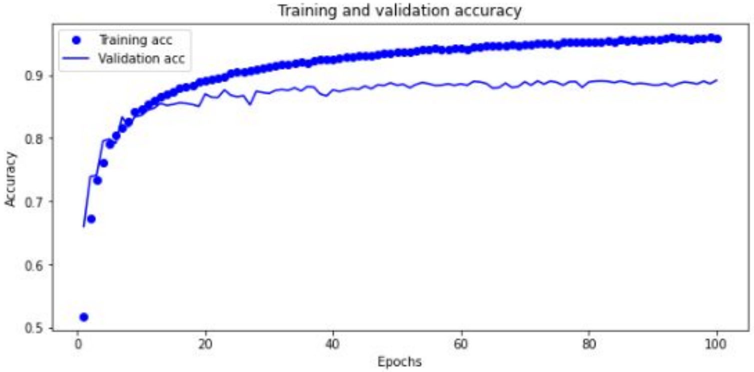
Baseline + Dropout + image augmentation : 75.5%

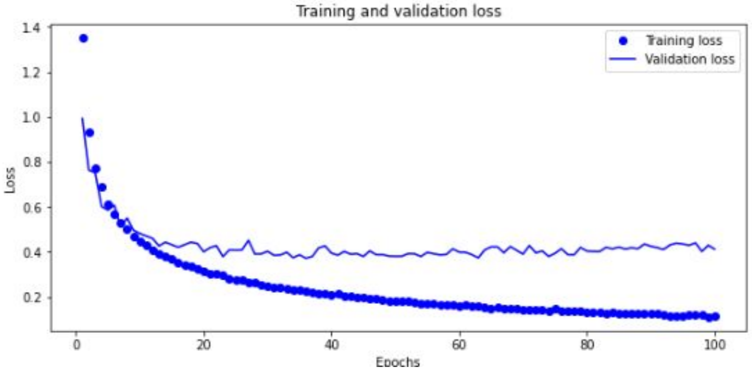
Adding batch normalization :

```
modelCNN = Sequential([

    # Convolution Layer 1 &2
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)), # 32 different 3x3 kernels
    BatchNormalization(),
    Conv2D(32, (3, 3), activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    # Convolution Layer 3 &4
    Conv2D(64, (3, 3), activation='relu', padding='same'), # 64 different 3x3 kernels
    BatchNormalization(),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    # Convolution Layer 5 &6
    Conv2D(128, (3, 3), activation='relu', padding='same'), # 128 different 3x3 kernels
    BatchNormalization(),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)), # Pool the max values over a 2x2 kernel

    Flatten(), # Flatten final 3x3x128 output matrix into a 1152-length vector

    # Fully Connected Layer 4
    Dense(128), # 128 FCN nodes
    Activation('relu'),
    BatchNormalization(),
    Dense(10),
    Activation('softmax'),
])
```



VGG baseline, image augmentation & batch normalization:





Like Dropout regularization, image augmentation helps **reducing the overfitting** and **improve the performance of the network.**

Baseline accuracy : 75.5%

<u>Baseline + Dropout</u> : **77.7% and no overfitting**

Baseline + L2 : 61.5% and no overfitting

<u>Baseline + image augmentation</u> : **82% and less overfitting**

Baseline + Dropout + image augmentation : 75.5%

<u>Baseline + image augmentation + Batch Norm :</u> 89.2%

What did we learn?

- How to create and use a **convolutional neural network for image classification** :

    - convolution layer
    - max-pool layer

- Recognize **overfitting** and methods to reduce it while **optimizing the performances of the network** :

    - dropout
    - regularization
    - **image augmentation**
    - batch normalization
    - transfert learning
    - ...

⇨ <u>There is no "unique solution".</u> It strongly depends on your dataset. Need to use a "try & error" strategy.