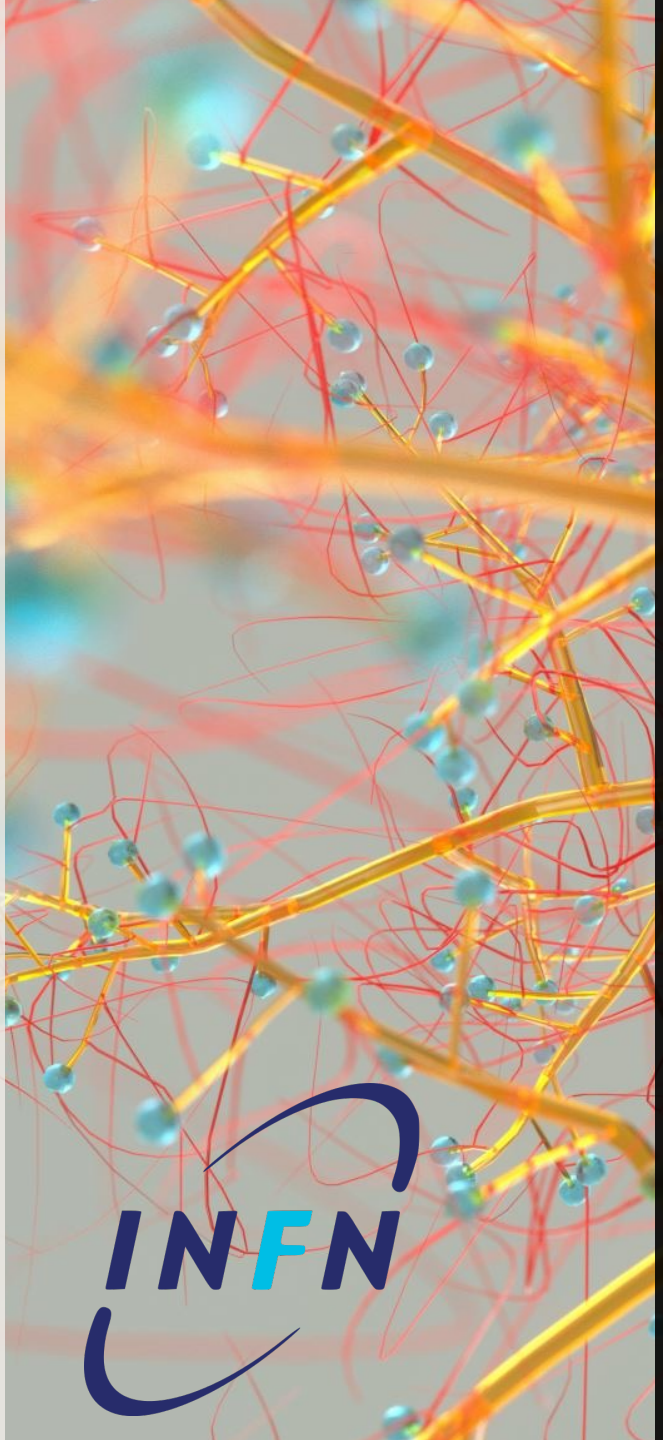


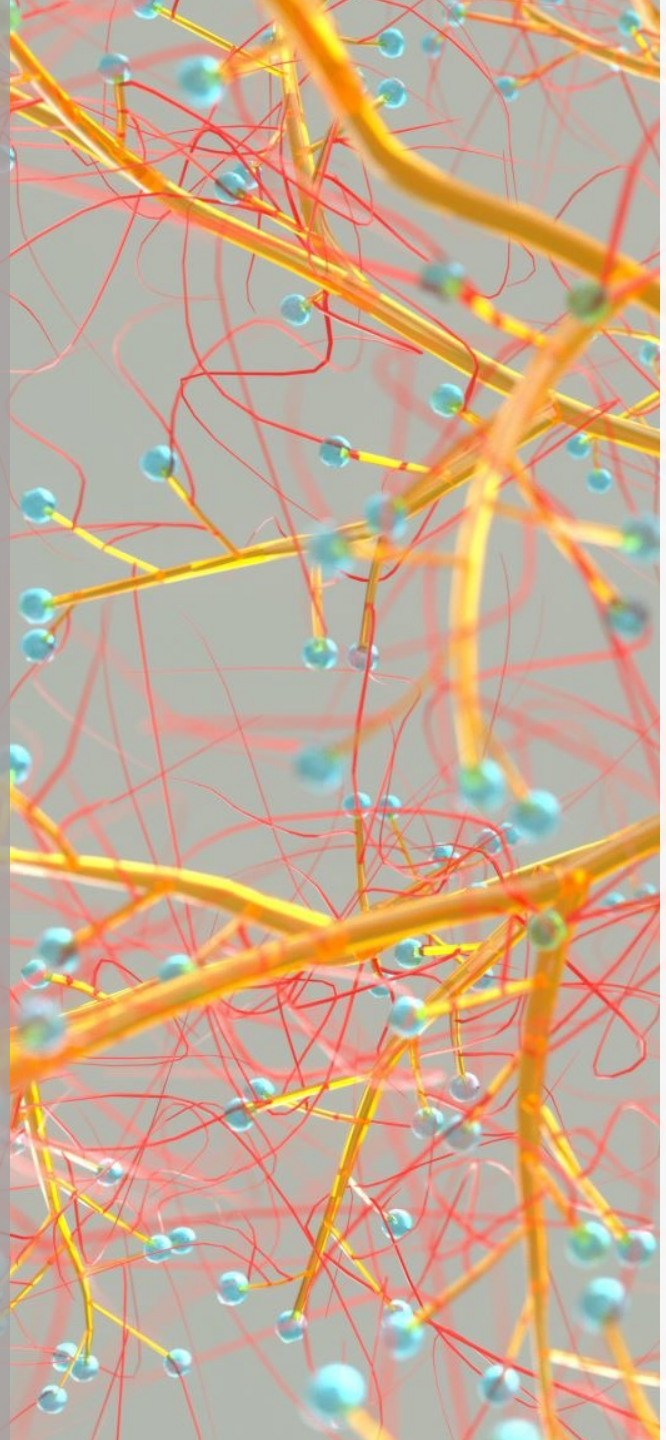
Convolutional Neural Networks

Corso di Formazione Nazionale INFN

"Introduzione alle reti neurali e applicazioni sui dispositivi elettronici"

Napoli, 17/11/2022

Silvia Auricchio & Francesco Cirotto
Università degli studi di Napoli "Federico II" - INFN sezione di Napoli



Introduction to the course

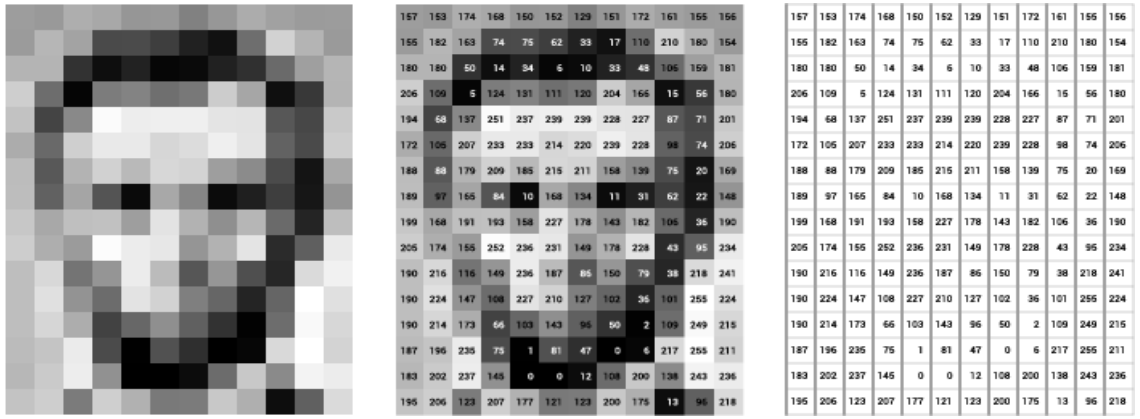
- This course is intended to give a fast overview on the basic aspects of Convolutional Neural Network
- The covered topics are:
  1. Digital image classification problem
  2. An introduction to convolution in 1 and 2 dimensions
  3. An overview on the different layers of a CNN
  4. A brief explanation of the Data Augmentation
  5. An example on how to build a CNN with Keras and TensorFlow

Digital images

- Convolutional Neural Networks are a powerful family of neural networks that are specifically designed for the Images Processing Task

WHAT ARE IMAGES IN THE DIGITAL WORLD?

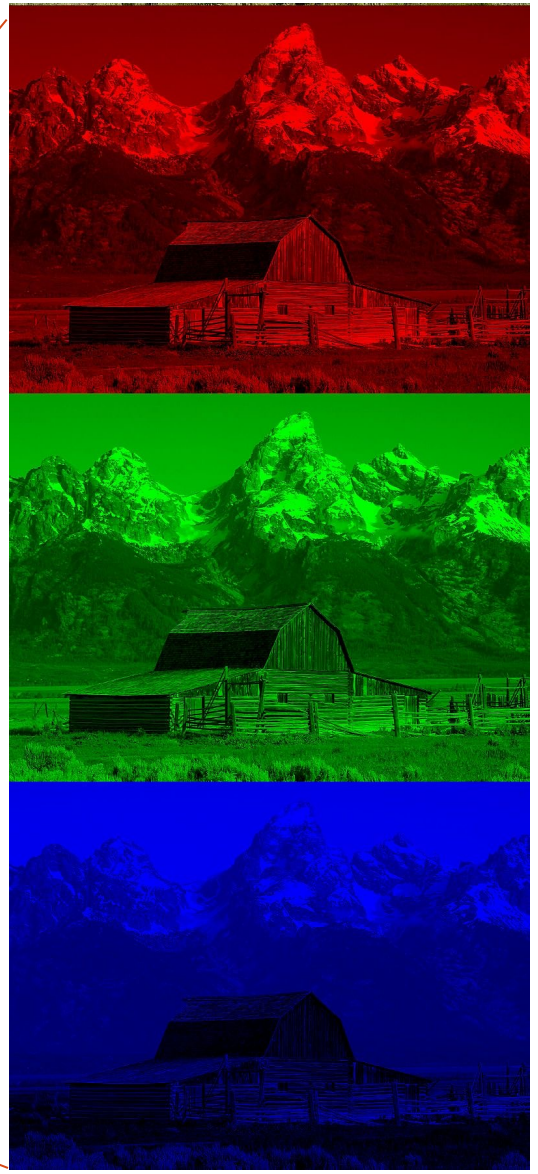
- From Wikipedia: "***A digital image is an image composed of picture elements, also known as pixels, each with finite, discrete quantities of numeric representation for its intensity***."



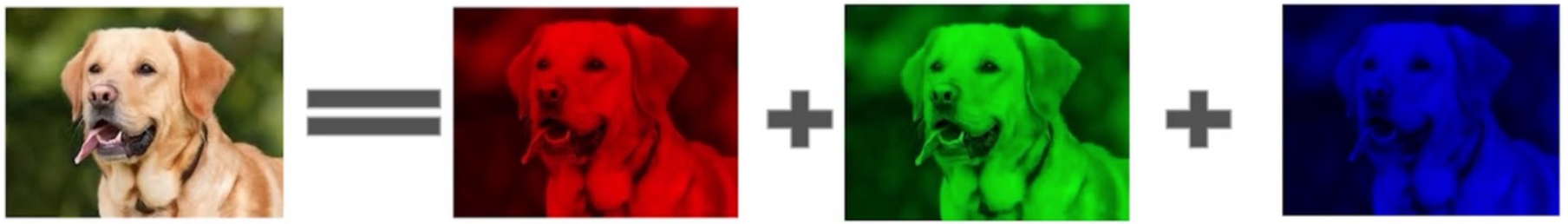
- Images can be **greyscale**, where **each pixel value is the grey intensity**, or colored

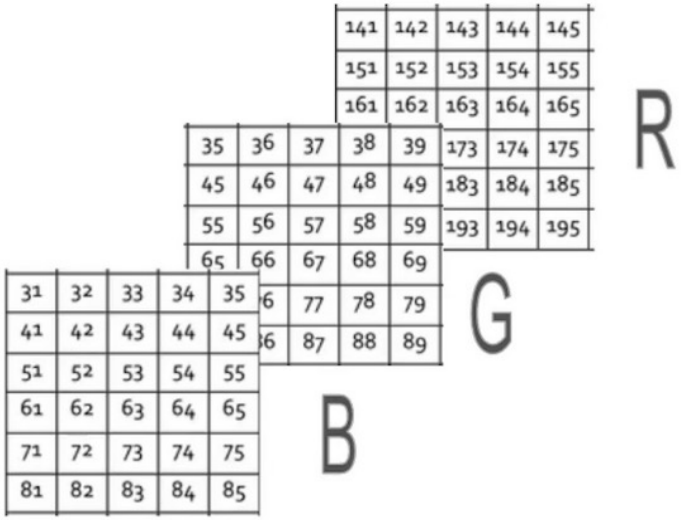
Digital images

- Colored images are usually coded with the **RGB color model: each pixel is associated to three numbers**, corresponding to the **Red, Green and Blue intensity**
- The image is obtained as the sum of the three components



Digital images



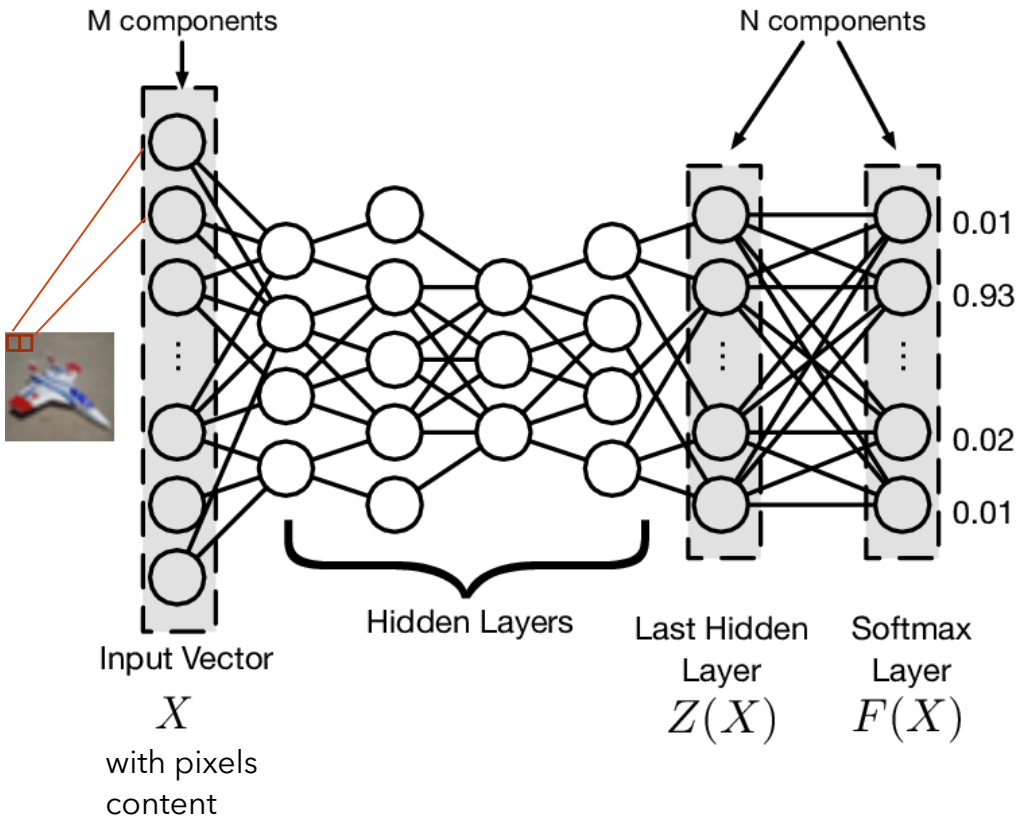


- An RGB image is therefore represented by a matrix (weight)x(height)x3
- A greyscale image is (weight)x(height)x1

HOW DO WE CLASSIFY IMAGES?

Images classification with DNN

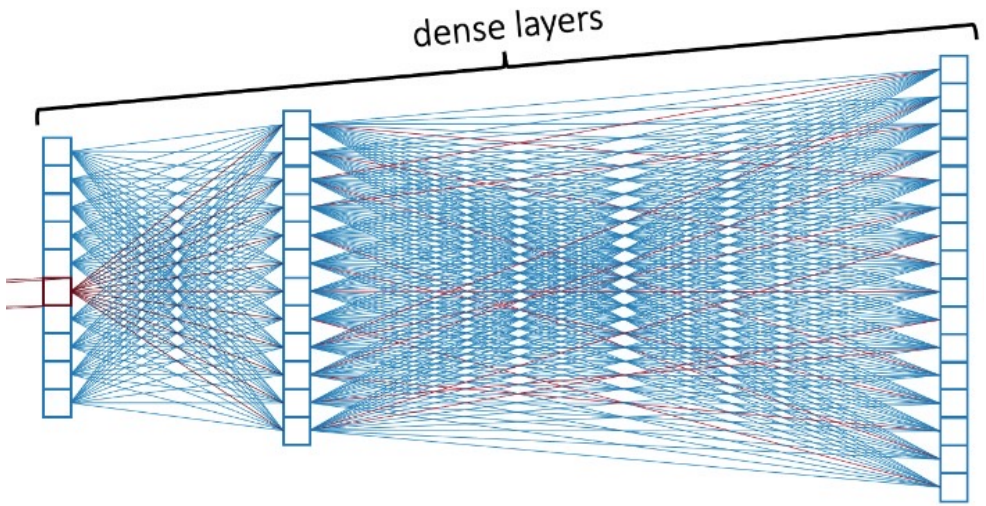
- As we saw in the previous tutorials, a classification problem can be easily addressed from a neural network.
  - The more the problem is complex and not linear the better Deep Networks perform with respect to single layer networks.
- Images can be given in input to DNN by flattening pixels to form a 1D array.
  - DNNs are also invariant to input features order, therefore they could also be shuffled before being fed the network



Images classification with DNN

Example:

- the input is a **64x64 grayscale image**. Such an image can be represented by 64x64x1 = 4096 values
- the **input layer** of a DNN processing such an image has **4096 nodes**
- if the (fully connected) inner layer has **500 nodes**, we will have **4096x500 = 2048000 weights between the input and the hidden layer**
- If the image were an RGB image the input layer would have 64x64x3 = 12288 nodes
- For complex problems, we usually need multiple hidden layers...

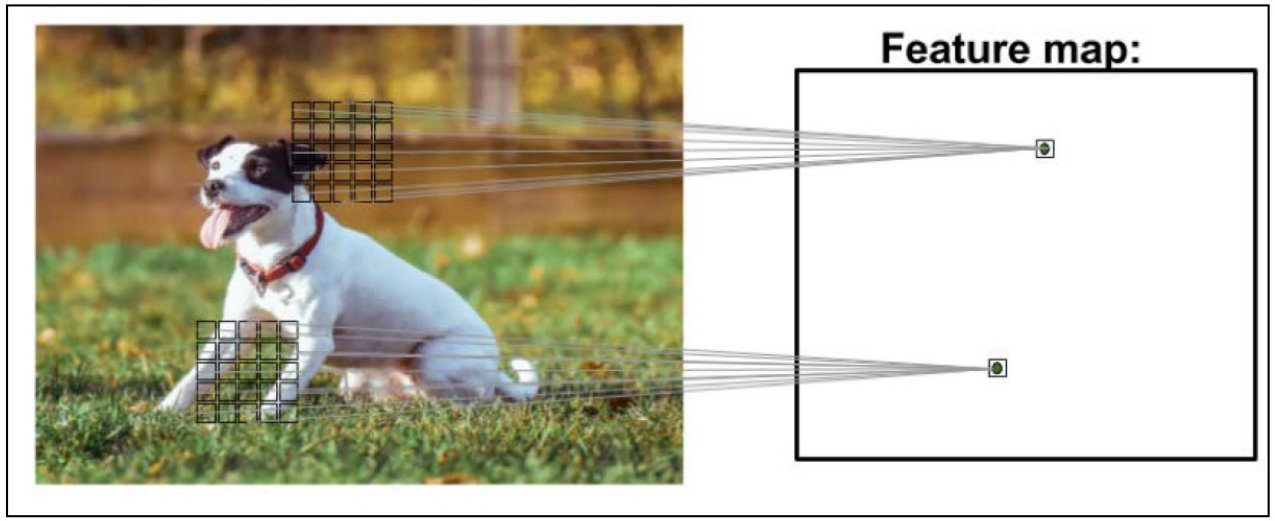


A DNN CANNOT SCALE TO HANDLE LARGE IMAGES. WE NEED A MORE SCALABLE ARCHITECTURE

Convolutional Neural Networks

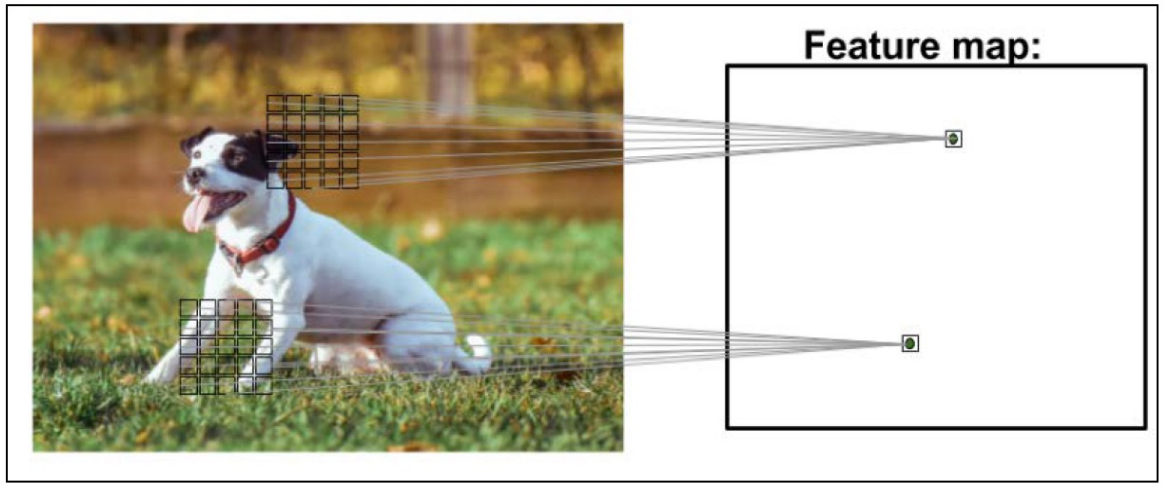
- To flat an image into a 1D array is not the best way to model images
  - any spatial relationship in the data is ignored
- A Convolutional Neural Network (CNN) maintains the spatial structure of the data, and is better suited for finding spatial relationships in the image data

- **The idea behind:** to use filters that automatically learns the most discriminants features in an image, such as edges, filled patterns, specific geometric forms and so on



(Photo by Alexander Dummer on Unsplash)

Convolutional Neural Networks



BASIC CONCEPTS:

- **Sparse connectivity**: A single element in the feature map is connected to only a small patch of pixels.
- **Parameter-sharing**: the same weights are used for different patches of the input image

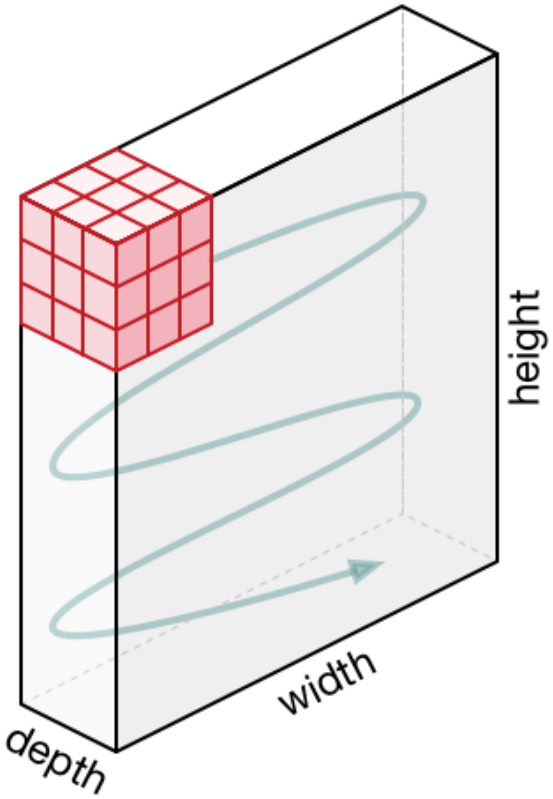
from **Spatial Invariance Principle**: *whatever method is used to recognize objects it should not be concerned with the precise location of the object in the image*

The Convolutional Layer

- Let's start from **the Convolutional Layer**:
  - It is the core building block of a Convolutional Network that does most of the computational heavy lifting

INTUITION WITHOUT BRAIN STUFF

- The CONV layer's parameters consist of a set of learnable filters.
- Every filter is small spatially (along width and height) and extends through the full depth of the input volume.
- The output is a single layer with a certain spatial size (width x height x1)



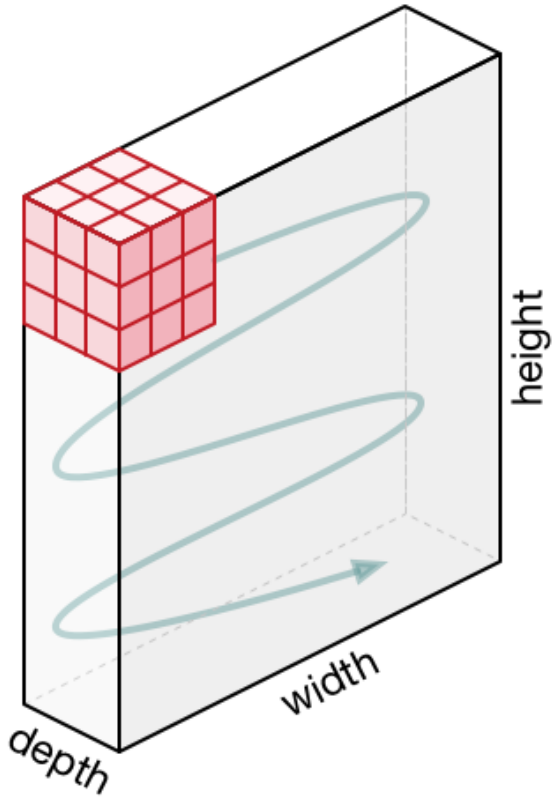
The Convolutional Layer

- Let's start from **the Convolutional Layer**:
  - It is the core building block of a Convolutional Network that does most of the computational heavy lifting

INTUITION WITHOUT BRAIN STUFF

Example:

- a typical filter on a first layer of a CNN has size 5x5x3
- Let's suppose we have a 28x28 pixel RGB image
- Convolution is the process of placing the filter 5x5x3 on the top left corner of the image, multiplying filter values by the pixel values and adding the results, moving the filter to the right one pixel at a time and repeating this process



The Convolutional Layer

THE MATHEMATICAL VIEW

- Now let's explain the convolution in more mathematical details

Discrete convolution in one dimension

- A discrete convolution between two vectors with finite size, x and w, is mathematically defined as:

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]\, w[k]$$

- w is typically called the *filter* or *kernel*
- The index i runs through each element of the output vector y
- In machine learning applications we always deal with finite feature vectors

- Let's assume that x and w have n and m elements respectively, where $m \leq n$.

The Convolutional Layer

THE MATHEMATICAL VIEW

- The convolution becomes:

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \rightarrow \quad y[i] = \sum_{k=0}^{k=m-1} \boldsymbol{x}\ [i + m - k]w[k]$$

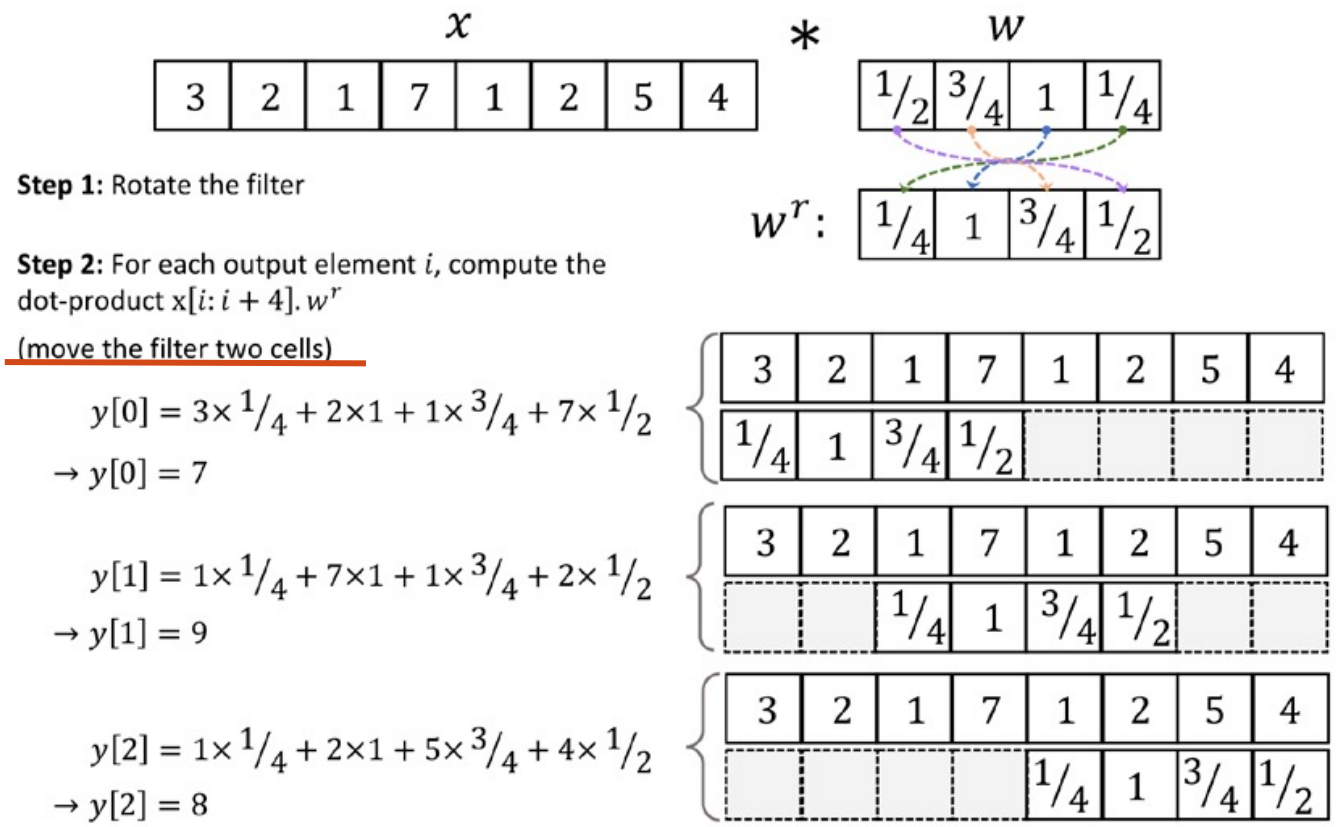
- It's important to notice that x and w are indexed in different directions in this summation.
- Computing the sum with one index going in the reverse direction is equivalent to computing the sum with both indices in the forward direction after flipping one of those vectors
- This operation is repeated like in a sliding window approach to get all the output elements.

The Convolutional Layer

THE MATHEMATICAL VIEW

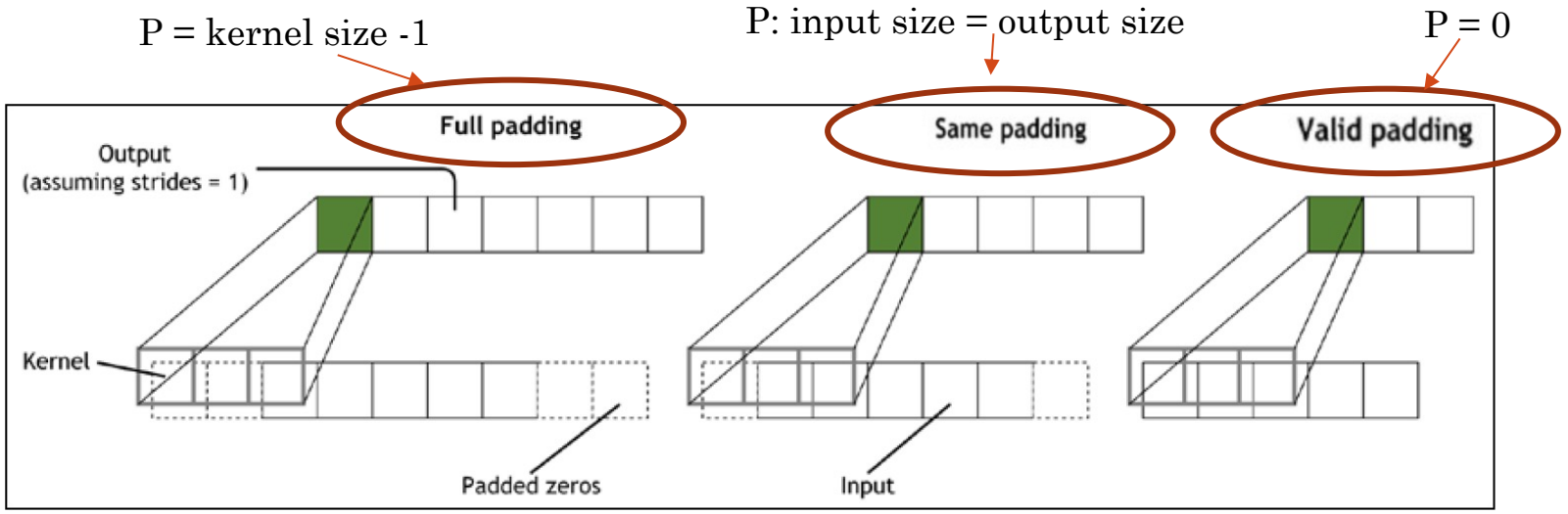
Example:

- x = [3 2 1 7 1 2 5 4], w = [½, ¾, 1, ¼]



Padding Layer

- The result of this convolution is a tensor with a smaller shape than the input one
- To preserve/increase shape a ***padding*** procedure can be applied
  - It consists in padding zero pixels to input tensor
  - Usually, a ***same padding*** procedure is used, meaning that the output vector has the same size as the input one.



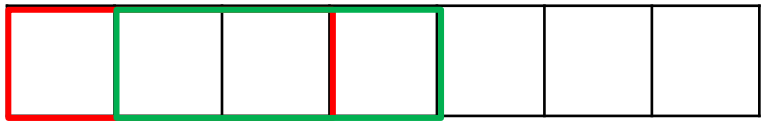
Strides

MOVING ALONG INPUTS

- One concept introduced in the previous example is **the number of cells the filter is moved when shifted** across the vector x (to pass from a y index to another)
- It is called **strides**

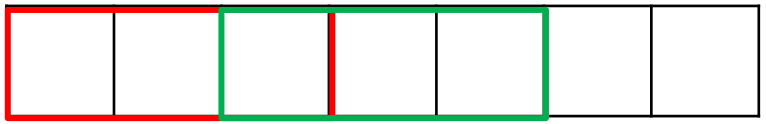
Example:

- N=7, filter = 3

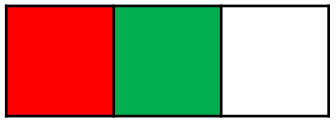


Stride = 1

Output = 5



Stride = 2



Output = 3

The output size

SIZE OF OUTPUT

- The size of the vector obtained by a convolution can be calculated as follows:

$$o = \left[\frac{n + 2p - m}{s}\right] + 1$$

- o = output dimension
- n = input dimension
- p = padding
- m = kernel size
- s = stride

Convolution in 2D

THE MATHEMATICAL VIEW

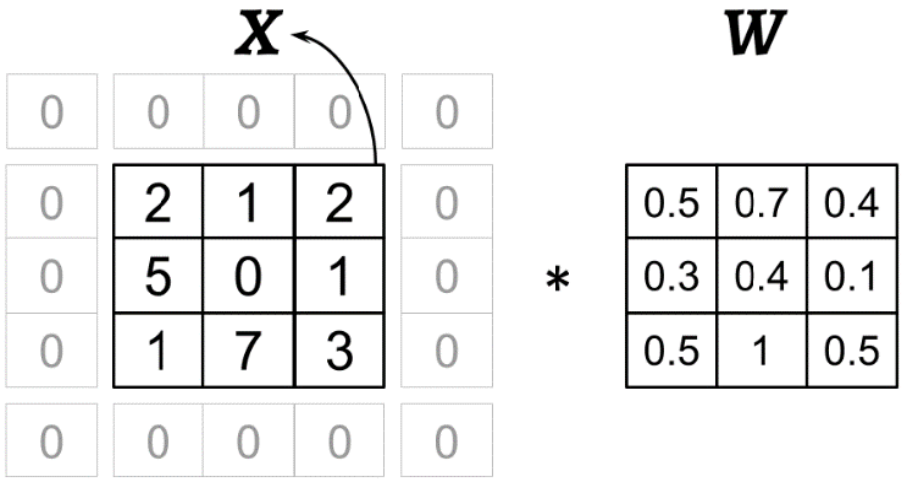
- The concepts we have now discussed are easily extendible to 2D case:

$$\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W} \rightarrow \quad Y[i,j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i-k_1, j-k_2]\, W[k_1, k_2]$$

  - $\boldsymbol{X}_{n1\times n2}$ and $\boldsymbol{W}_{m1\times m2}$ are now two matrices -> Y is a 2D matrix as well

Example:

- input matrix $\boldsymbol{X}_{3\times3}$
- kernel matrix $\boldsymbol{W}_{3\times3}$
- p=(1, 1)
- stride s=(2, 2)

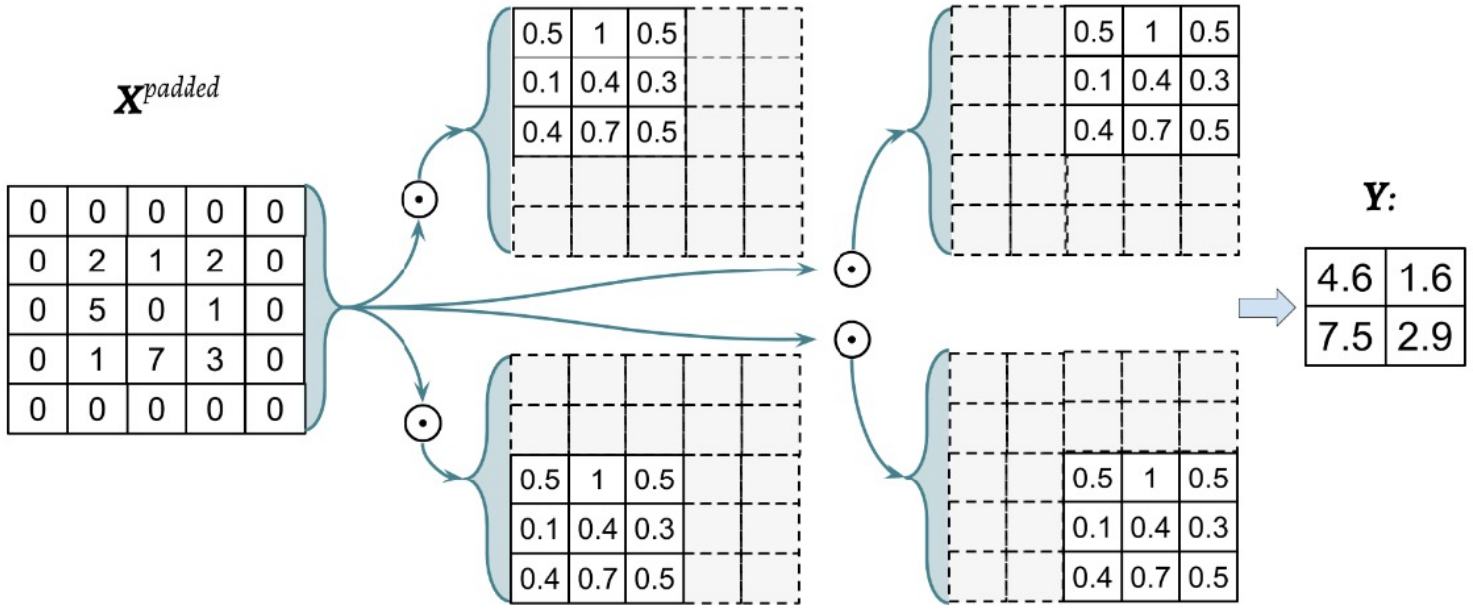


Convolution in 2D

THE MATHEMATICAL VIEW

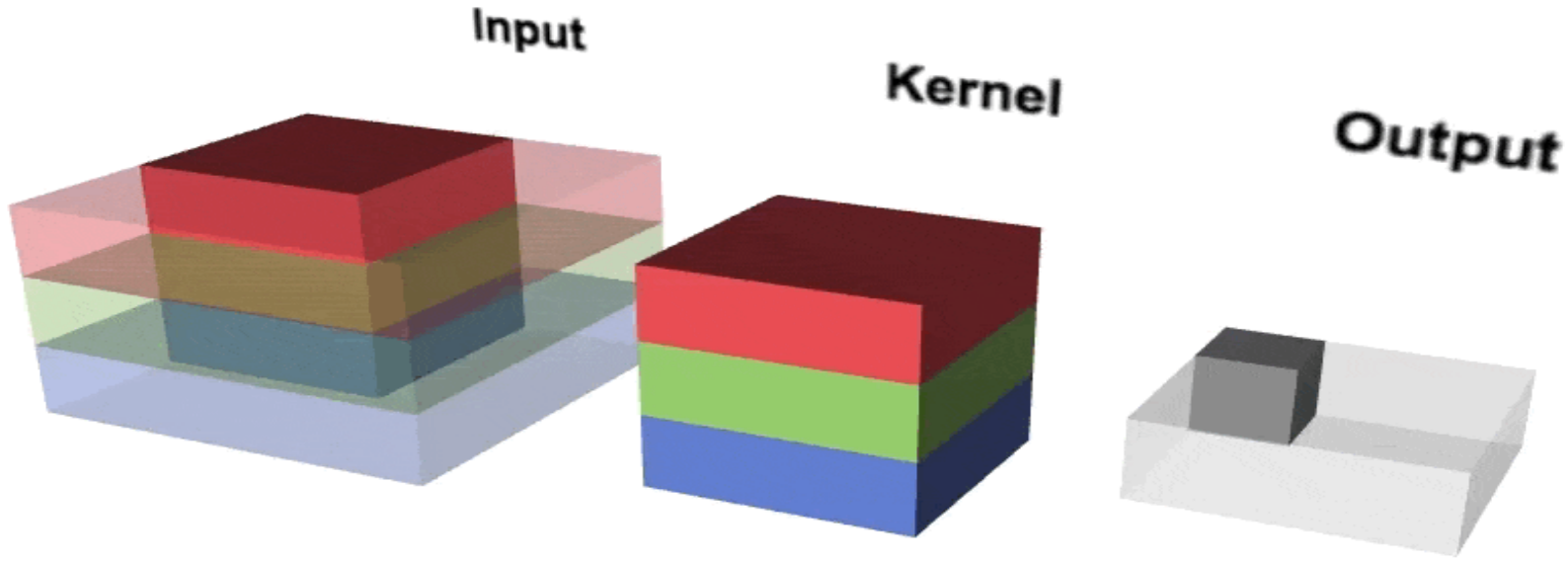
- We can rotate the filter to perform the sum on indices running in the same directions

$$\boldsymbol{W}^r = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

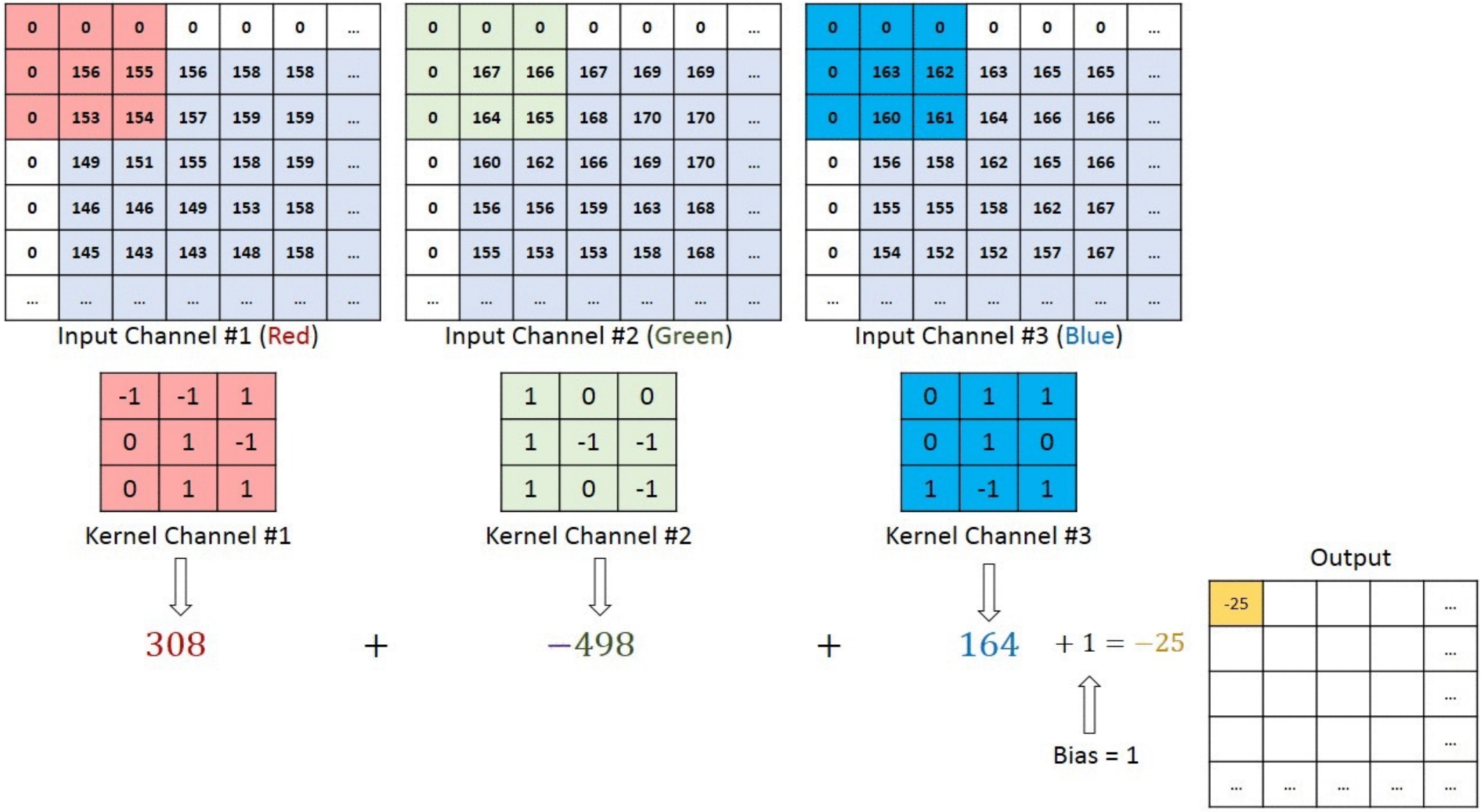


Convolution in 2D

- How does a convolutional layer work on a RGB image?
  1. For each channel color there is a different filter
  2. The three outputs are added together
  3. **The output of a convolutional layer with a multi-layer input is a single layer**



Convolution in 2D



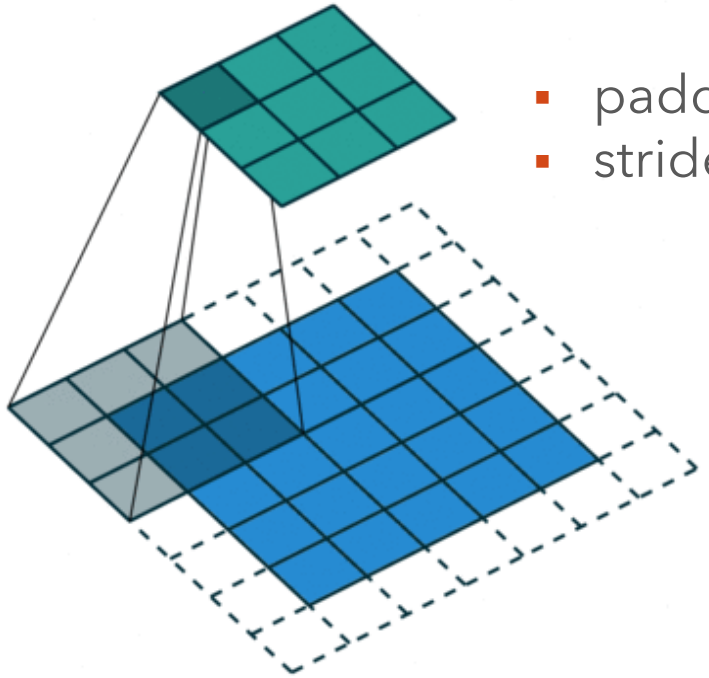
Strides and padding in 2D

- Zero-padding and strides concepts are the same of 1D case
- The output size of a 2D filter is still calculable with the formula seen before, applied on weight and height separately

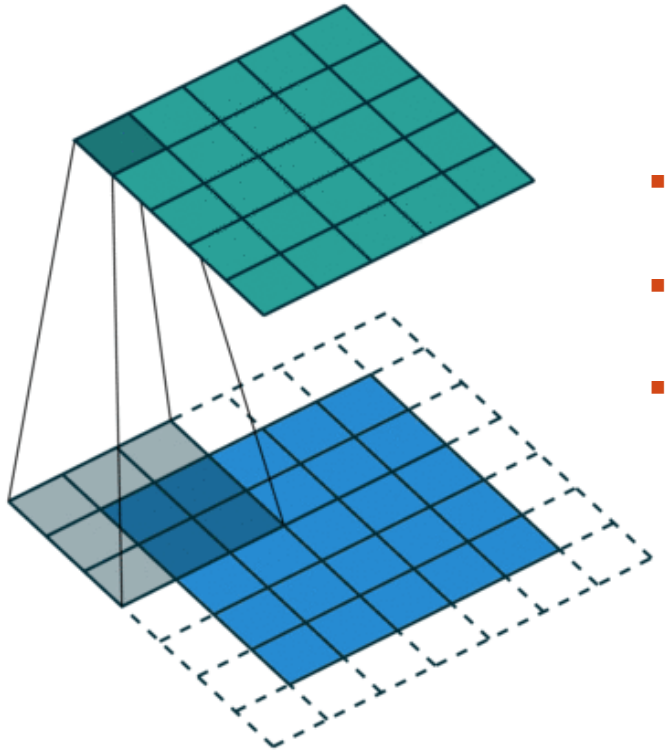
Input dimension Zero-padding Kernel dimension

$$o = \left[\frac{n + 2p - m}{s}\right] + 1$$

stride



- padding (1,1)
- strides (2,2)



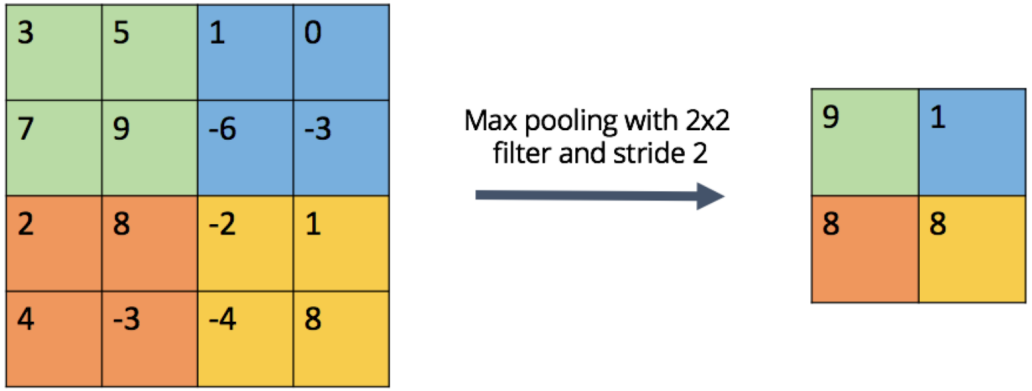
- padding (1,1)
- strides 1
- The input shape is preserved (padding 'same')

The Pooling Layer

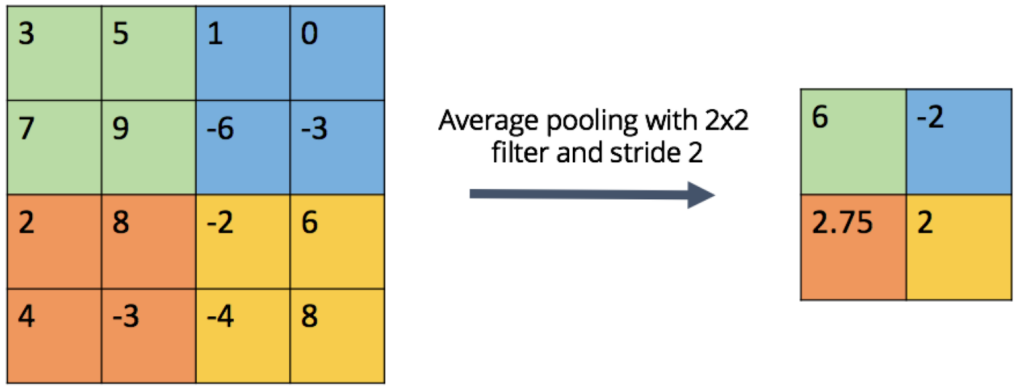
- According to zero-padding and strides it is possible to change (usually reduce) the input dimension
- This task can be also performed with a "Pooling" layer

TWO KINDS OF POOLING:

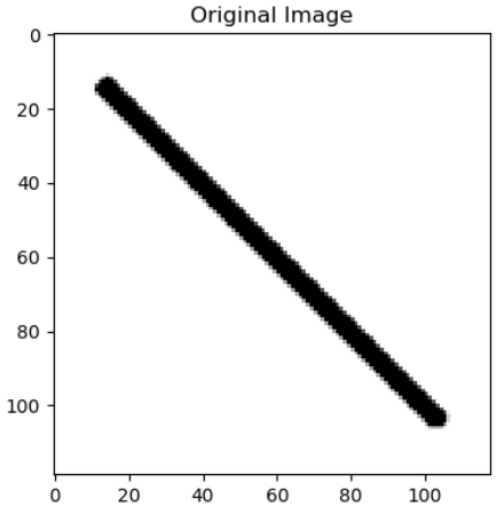
- **Maximum Pooling (or Max Pooling)**: Calculate the maximum value for each patch of the feature map.

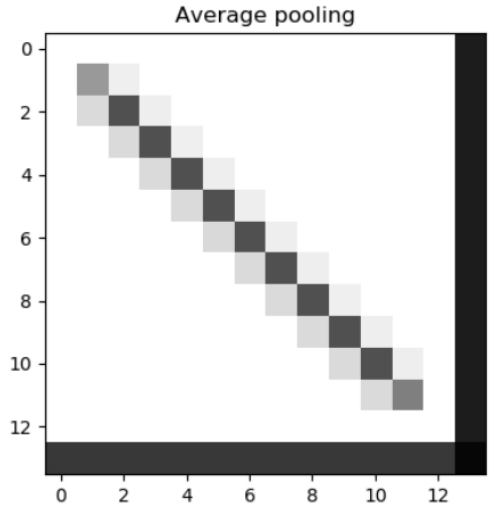


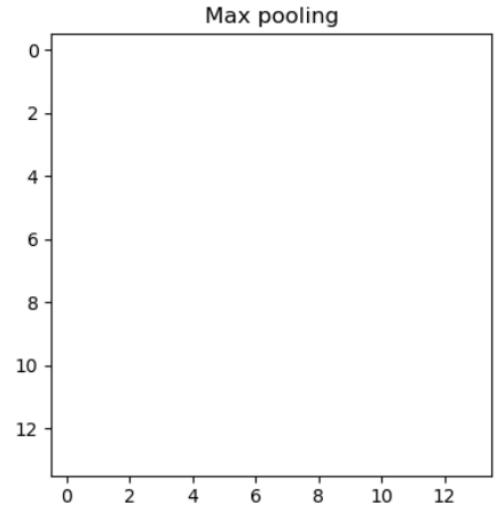
- **Average Pooling**: Calculate the average value for each patch on the feature map.

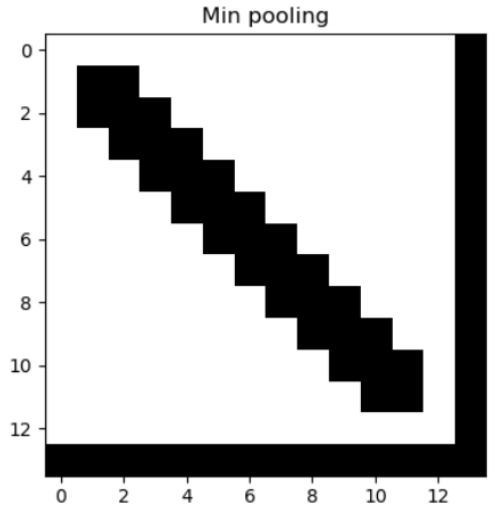


Different Pooling Layers

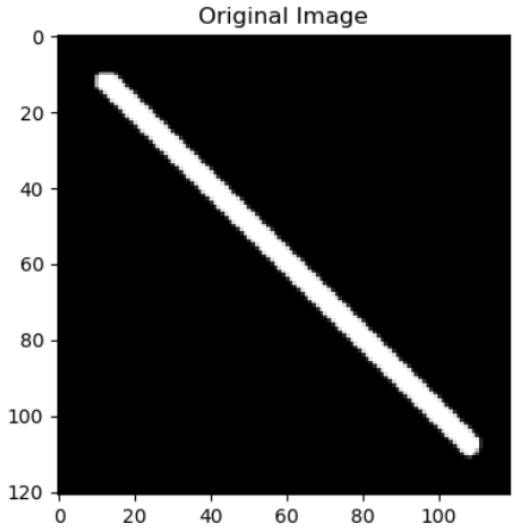


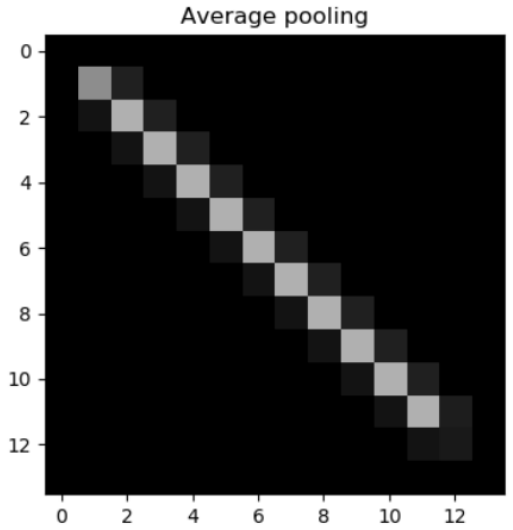


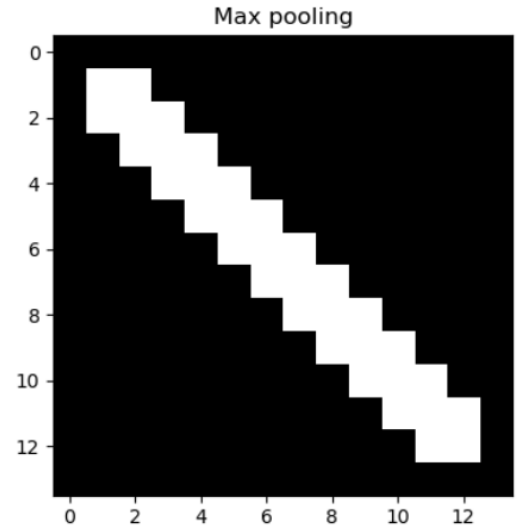


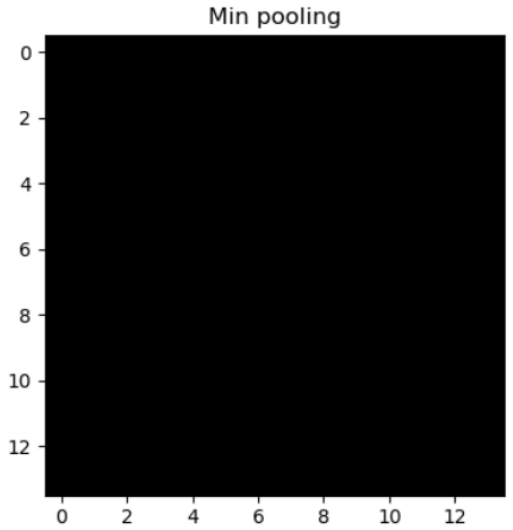


Different Pooling Layers









The Max Pooling Layer

MAIN ASPECTS:

- **Pooling** (max-pooling) **introduces a local invariance**. This means that small changes in a local neighbourhood do not change the result of max-pooling

$$X_1 = \begin{bmatrix} 10 & 255 & 125 & 0 & 170 & 100 \\ 70 & 255 & 105 & 25 & 25 & 70 \\ 255 & 0 & 150 & 0 & 10 & 10 \\ 0 & 255 & 10 & 10 & 150 & 20 \\ 70 & 15 & 200 & 100 & 95 & 0 \\ 35 & 25 & 100 & 20 & 0 & 60 \end{bmatrix}$$

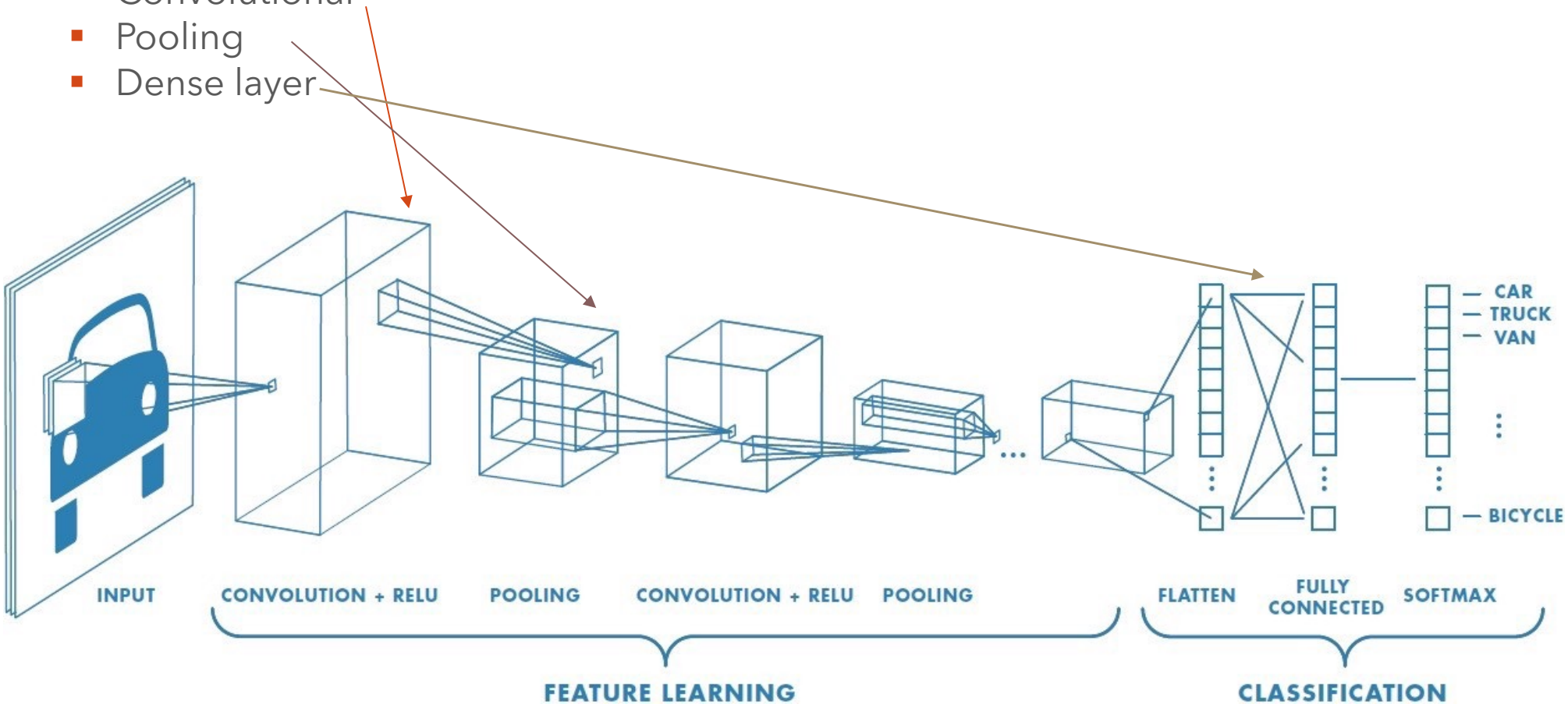
$$X_2 = \begin{bmatrix} 100 & 100 & 100 & 50 & 100 & 50 \\ 95 & 255 & 100 & 125 & 125 & 170 \\ 80 & 40 & 10 & 10 & 125 & 150 \\ 255 & 30 & 150 & 20 & 120 & 125 \\ 30 & 30 & 150 & 100 & 70 & 70 \\ 70 & 30 & 100 & 200 & 70 & 95 \end{bmatrix}$$

$$\xrightarrow{\text{max pooling } P_{2\times 2}} \begin{bmatrix} 255 & 125 & 170 \\ 255 & 150 & 150 \\ 70 & 200 & 95 \end{bmatrix}$$

- Pooling **decreases the size of features**, which results in higher computational efficiency. Furthermore, reducing the number of features may reduce the degree of overfitting as well.
- Traditionally, pooling is assumed to be non-overlapping (**pooling size = stride**)

Putting everything together in a CNN

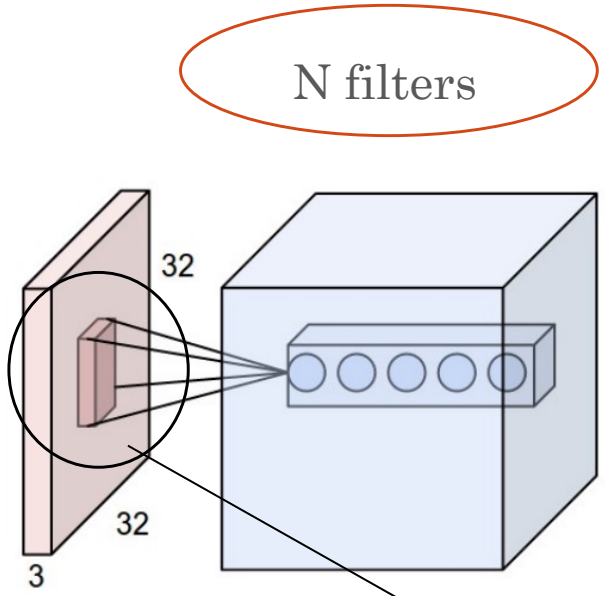
- A convolutional neural network is a sequence of the following layers ordered in different ways:
  - Convolutional
  - Pooling
  - Dense layer



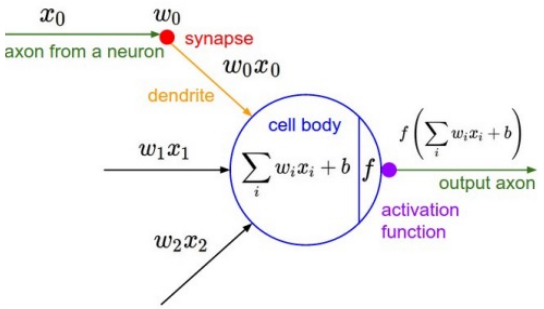
- FILTERS WEIGHTS ARE LEARNT FROM DATA DURING TRAINING
- THE NETWORK LEARNS WHICH ARE THE MOST DISCRIMINANT PATTERNS
- A CNN PERFORMS THE CLASSIFICATION BY READING THESE EXTRACTED FEATURES
- A DNN READS ONLY PIXELS VALUES

Putting everything together in a CNN

- Typically, a convolutional layer is composed of:



Conv operation

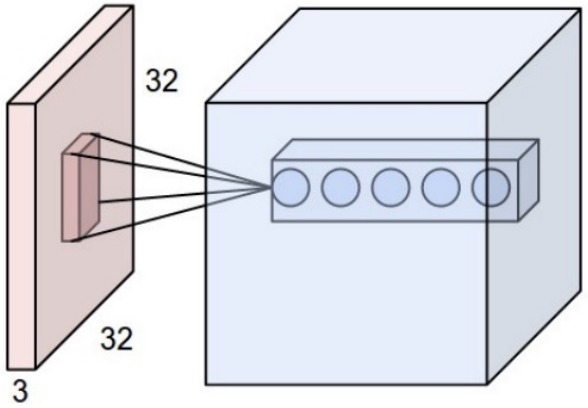


- In a convolution layer, usually, not only one but several filters are stacked together
- Each filter learns some different information from the same image

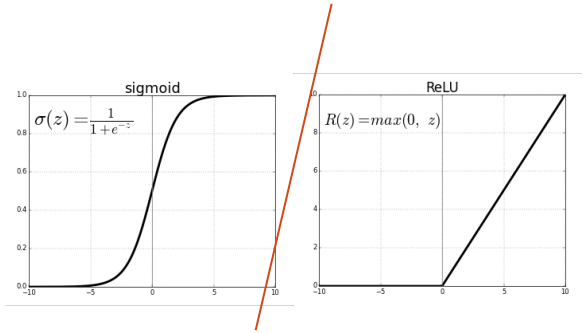
Putting everything together in a CNN

- Typically, a convolutional layer is composed of:

N filters + activation function



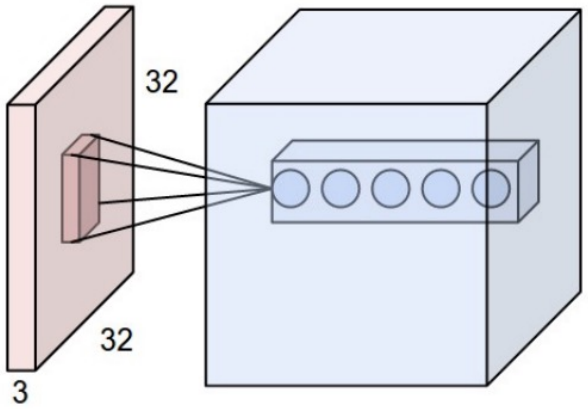
+



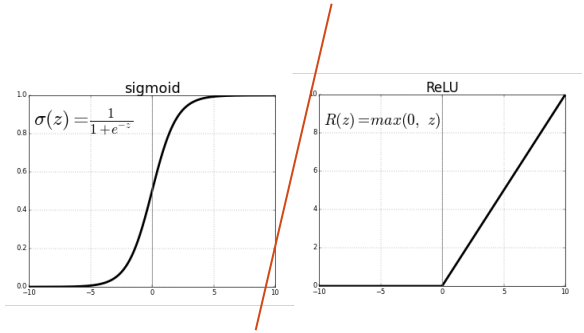
Putting everything together in a CNN

- Typically, a convolutional layer is composed of:

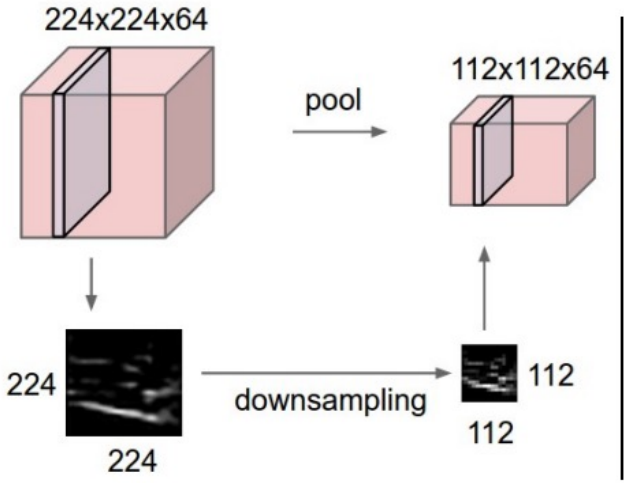
N filters + activation function + pooling layer



+

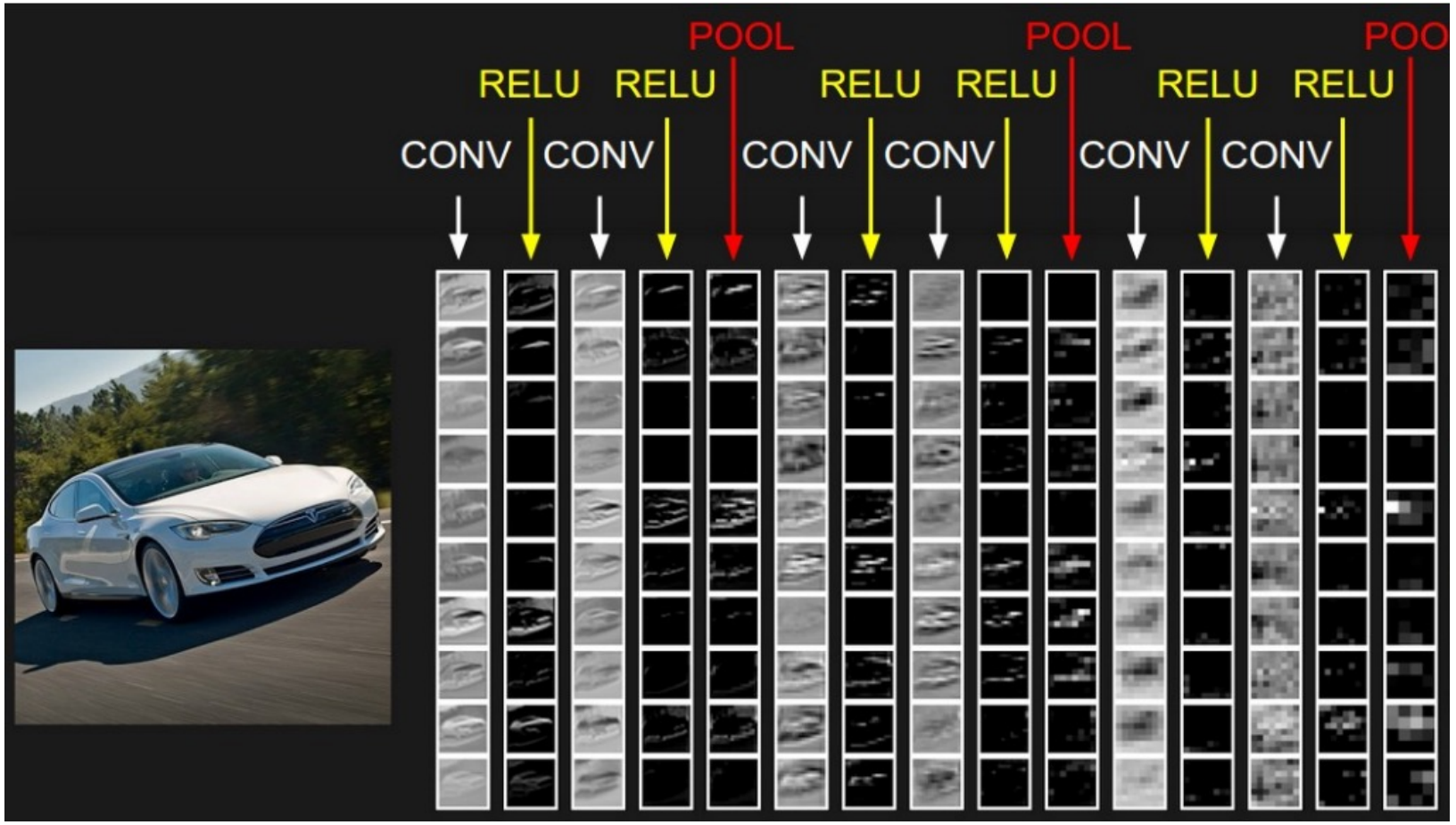


+



Putting everything together in a CNN

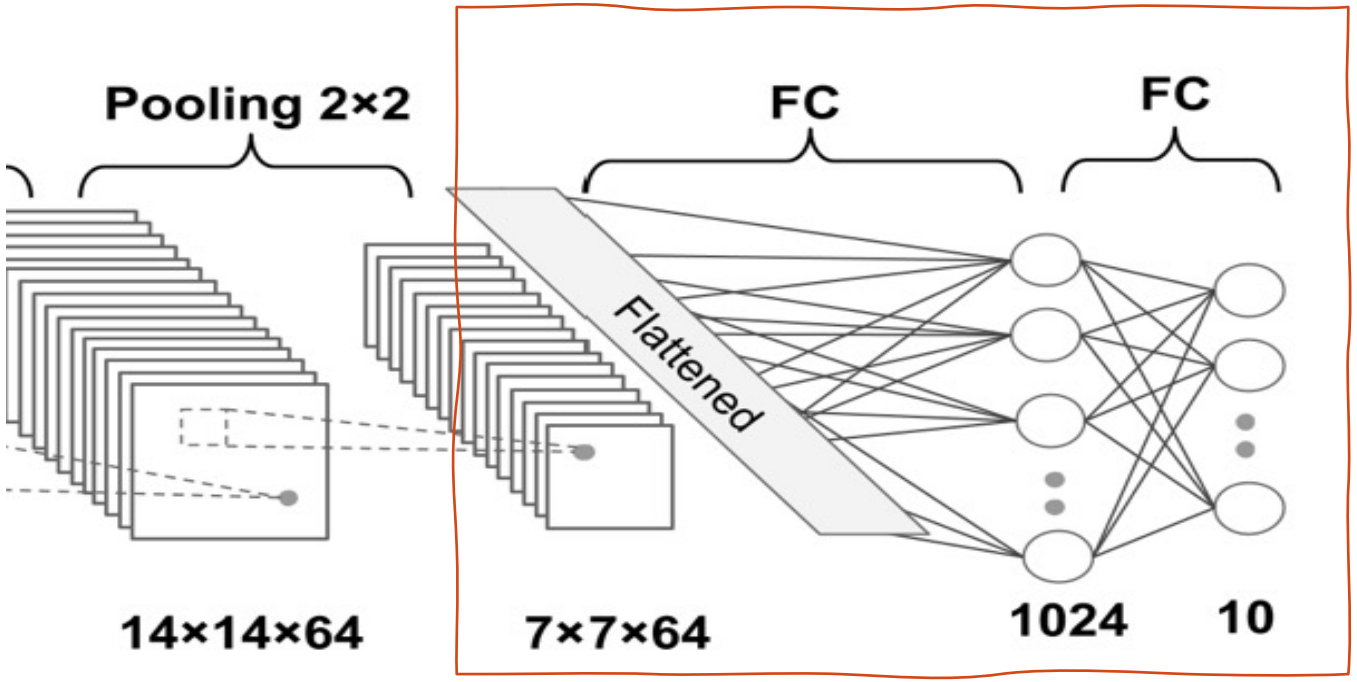
- The sequence convolutional + pooling is not the only one possible choice
  - Modern networks don't use pooling but adjust the output size by tuning the padding and strides of convolutional layers



Dense layers for classification

- Now we must **flatten the final output and feed it to a regular Neural Network for classification purposes**
- Adding a Fully-Connected layer is a way of learning non-linear combinations of the high-level features (from filters)

1. The image is **flattened** into a column vector
2. then **fed to a fully-connected neural network**



Activations and Loss functions for classification

- Now what our model misses is the final probabilistic interpretation of the output **z** to perform classification:

$$z = \boldsymbol{w}^T \boldsymbol{x} = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m$$

- An **activation function** should be applied to the output of the last fully-connected layer:
  - 'Sigmoid' for binary classification $\phi(z) = \frac{1}{1 + e^{-z}}$
  - 'Softmax' for multi-classification $p(z) = \phi(z) = \frac{e^{z_i}}{\sum_{j=1}^{M} e^{z_j}}$

Recall: Sigmoid and Softmax are both probabilities for an event to belong to a given class, therefore they are used only in the outer layer. Other activation functions, like ReLU and tanh are mainly used in the intermediate (hidden) layers to add non-linearities to our model

Activations and Loss functions for classification

- Focusing on classification problems, depending on the type of problem and the type of output (logits versus probabilities), we should choose the **appropriate loss function** to train our model

Loss function	Usage	Examples	
		Using probabilities	Using logits
		from_logits=False	*from_logits=True*
`BinaryCrossentropy`	Binary classification	y_true: 1 y_pred: 0.69	y_true: 1 y_pred: 0.8
`CategoricalCrossentropy`	Multiclass classification	y_true: 0 0 1 y_pred: 0.30 0.15 0.55	y_true: 0 0 1 y_pred: 1.5 0.8 2.1
`Sparse CategoricalCrossentropy`	Multiclass classification	y_true: 2 y_pred: 0.30 0.15 0.55	y_true: 2 y_pred: 1.5 0.8 2.1

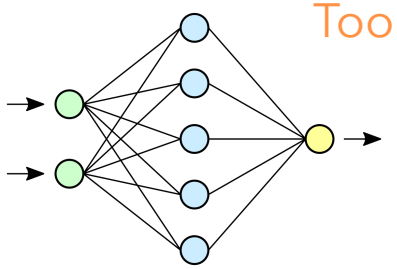
- With 'from_logits=True' logits are provided as inputs to the loss function (not the activation output), the inverse of the sigmoid function:

$$\text{logit}(p) = \ln\left(\frac{p}{1-p}\right)$$

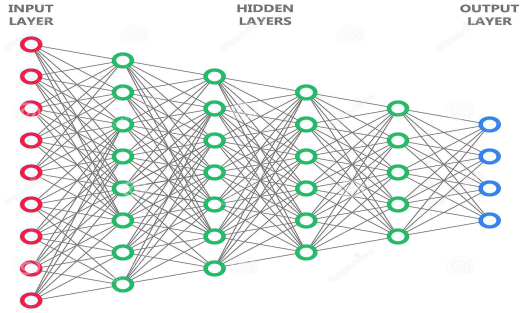
  - It is preferred due to numerical stability reasons

CNN regularization: Dropout

- Choosing the size of a network has always been a challenging problem
  - Small networks, or networks with a relatively small number of parameters, are likely to underfit, resulting in poor performance
  - very large networks may result in overfitting, where the network will do extremely well on the training dataset while achieving a poor performance on the test dataset



Too few parameters!



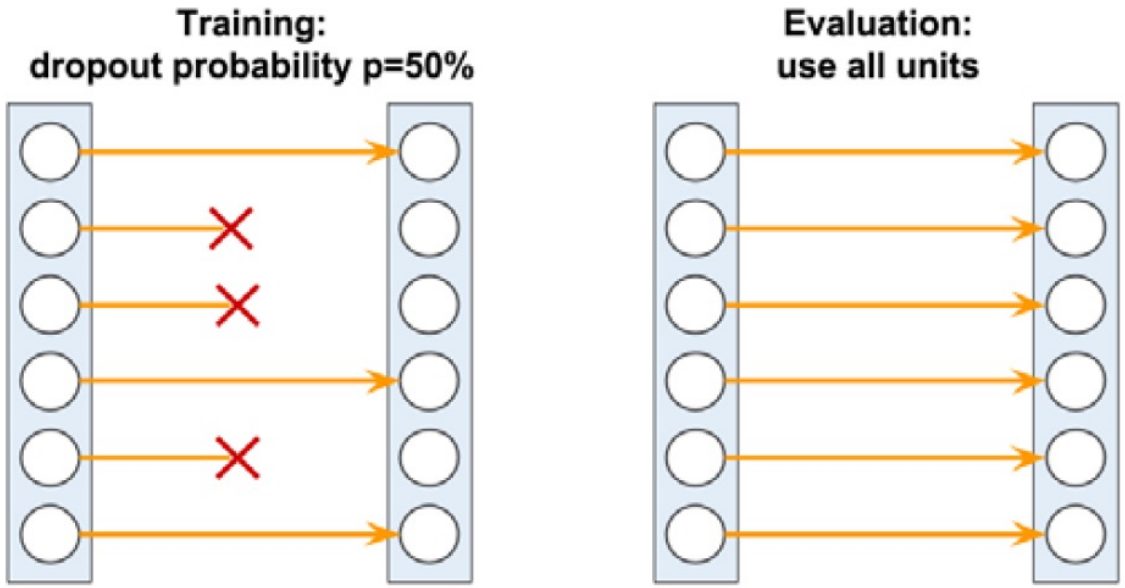
Too many parameters!

- One way to address this problem is **to build a network with a relatively large capacity to do well on the training dataset, then to prevent overfitting we can apply one or multiple regularization schemes** to achieve a good performance on new data

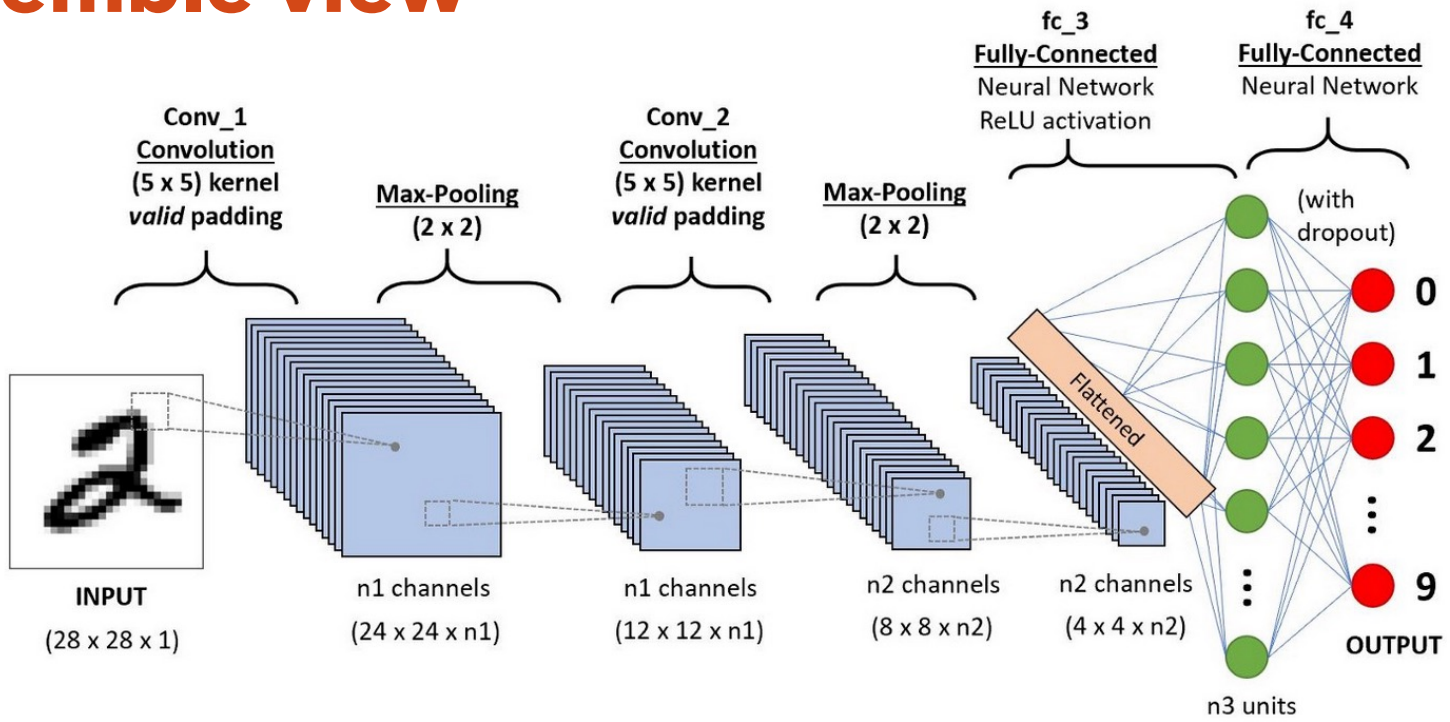
CNN regularization: Dropout

- **Dropout** has emerged as a popular technique for regularizing: during the training phase **a fraction of the hidden units is randomly dropped** at every iteration with a certain probability (***rate***)

- During prediction, all neurons will contribute to computing the pre-activations of the next layer



An ensemble view



- Since **the first convolutional** filters learn high level features in the image in input and the input size is larger than in inner layers, the **number of filters is relatively small to not insert too many weights**
- A good practice is **to increment this number in the subsequent convolutional steps**
- **Dropout can be inserted not only between dense layers but also between a Conv layer and its input**

Data Augmentation

- When the size of training dataset is small, a good practice to increase the fit performances is the **Data Augmentation**:
  - It consists in **replicating existent images by applying small changes to it** (**rotation, translation, resizing**..)
- In this way not only the number of training samples increases but each picture is fed to the network with different prospections

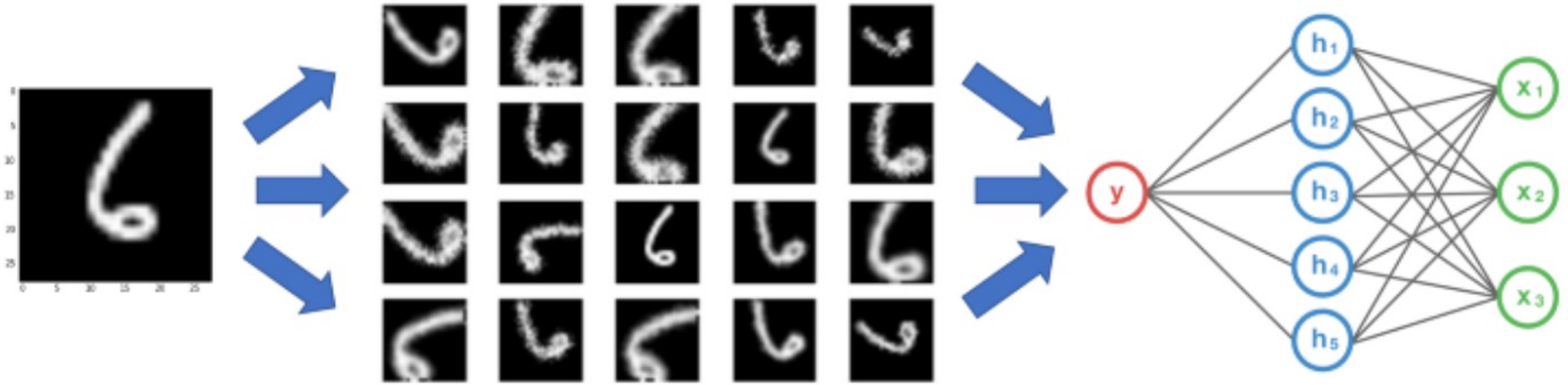
- A poorly trained neural network would think that these three tennis balls shown below are distinct images, instead they are not

Data Augmentation

- In the real-world scenario, we may have a **dataset** of images taken in a **limited set of conditions, but** our **target application** may exist in a **variety of conditions**, such as different orientation, location, scale, brightness etc.

- A convolutional neural network that can robustly classify objects even if it is placed in different orientations is said to have the property called **invariance** to **translation, viewpoint, size** or **illumination**

- We account for these situations by training our neural network with additional **synthetically modified data**



Data Augmentation

- Data augmentation can help to increase the amount of **relevant data** in the dataset

Example:

- Let's suppose you have to train a CNN for learning to distinguish between two car brands:

Brand A (Ford)

Brand B (Chevrolet)

- In the dataset all Brand A cars are facing left and all Brand B cars are facing right
- Now, you feed this dataset to your "state-of-the-art" neural network, and you hope to get impressive results once it's trained

Data Augmentation

- From training you get a 95% accuracy on your dataset
- If you feed a Brand A car to the CNN ….
- …. your neural network output is a Brand B car!

Why does this happen?

- A CNN finds the most obvious features that distinguishes one class from another, here all cars of Brand A were facing left and all cars of Brand B were facing right

Solution

Implementation of a CNN with TensorFlow

- We are now going to see the main steps **to implement a CNN for gender classification**

- **Dataset**: CelebFaces Attributes Dataset (CelebA) is a large-scale face attributes dataset with more than 200K celebrity images (from TensorFlow)

1. LOADING THE DATASET

```
>>> import tensorflow as tf
>>> import tensorflow_datasets as tfds
>>> celeba_bldr = tfds.builder('celeb_a')
>>> celeba_bldr.download_and_prepare()
>>> celeba = celeba_bldr.as_dataset(shuffle_files=False)

>>> celeba_train = celeba['train']
>>> celeba_valid = celeba['validation']
>>> celeba_test = celeba['test']
```

Implementation of a CNN with TensorFlow

2. DATA AUGMENTATION

- Let's see some of the possible transformations that are available via the *tf.image* module

Cropping to a bounding box

```
>>> img_cropped = tf.image.crop_to_bounding_box(examples[0], 50, 20, 128, 128)
```

Flipping horizontally

```
>>> img_flipped = tf.image.flip_left_right(examples[1])
```

Adjust contrast

```
>>> img_adj_contrast = tf.image.adjust_contrast( examples[2], contrast_factor=2)
```

Adjust brightness

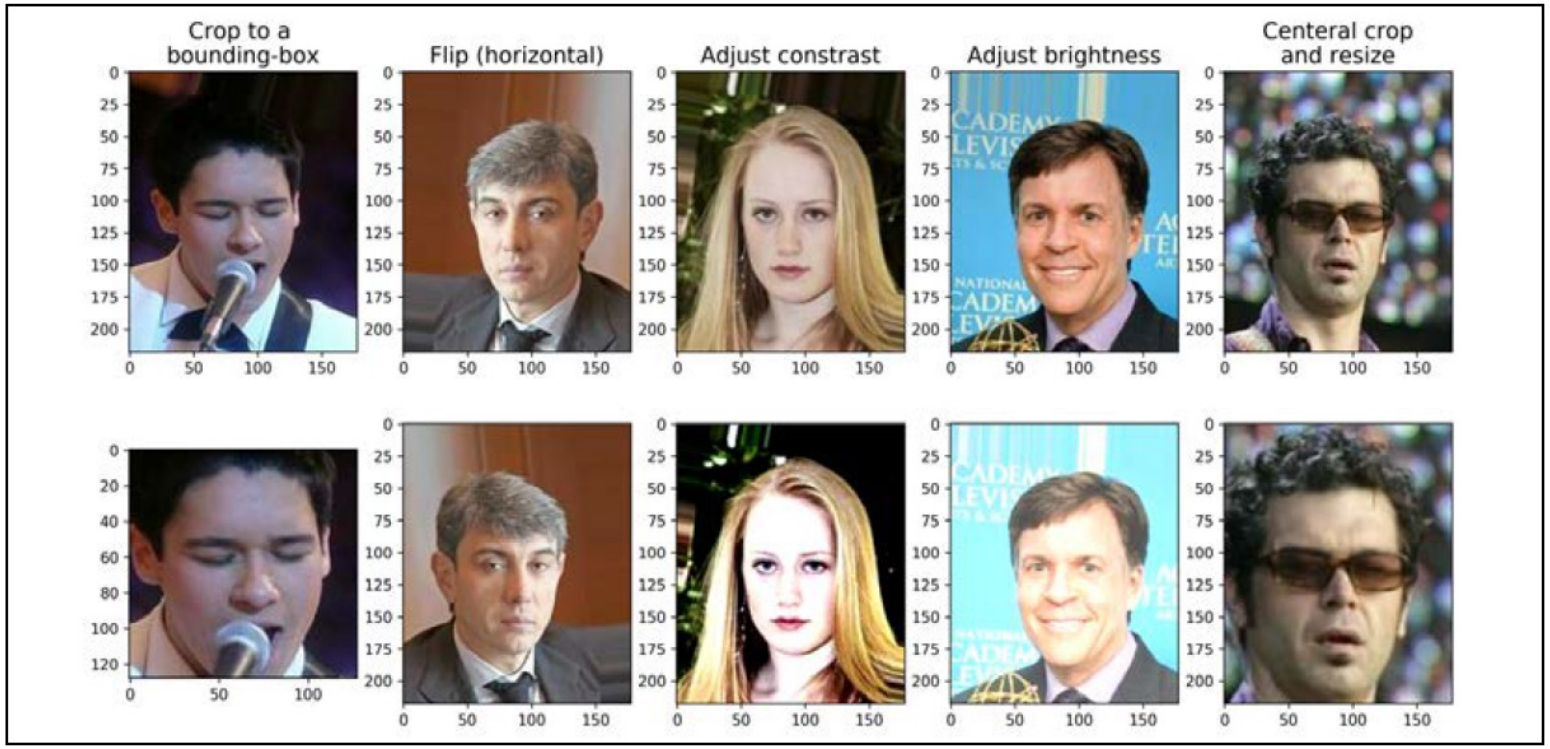
```
>>> img_adj_brightness = tf.image.adjust_brightness( examples[3], delta=0.3)
```

Cropping from image center

```
>>> img_center_crop = tf.image.central_crop( examples[4], 0.7)
```

Implementation of a CNN with TensorFlow

2. DATA AUGMENTATION



- TensorFlow has a method that automatically performs data augmentation 'online' per batch, instead of saving more pictures in the dataset (https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator)
- This is good for large datasets, but increases the computational time

Implementation of a CNN with TensorFlow

3. BUILDING THE MODEL

```
>>> model = tf.keras.Sequential([
...     tf.keras.layers.Conv2D(
...         32, (3, 3), padding='same', activation='relu'),
...     tf.keras.layers.MaxPooling2D((2, 2)),
...     tf.keras.layers.Dropout(rate=0.5),
...
...     tf.keras.layers.Conv2D(
...         64, (3, 3), padding='same', activation='relu'),
...     tf.keras.layers.MaxPooling2D((2, 2)),
...     tf.keras.layers.Dropout(rate=0.5),
...
...     tf.keras.layers.Conv2D(
...         128, (3, 3), padding='same', activation='relu'),
...     tf.keras.layers.MaxPooling2D((2, 2)),
...
...     tf.keras.layers.Conv2D(
...         256, (3, 3), padding='same', activation='relu')
>>>     ])
```

Implementation of a CNN with TensorFlow

3. BUILDING THE MODEL

- Input images of size 64 × 64 × 3 (the images have three color channels: RGB)
- TensorFlow has a method for computing the shape of the output feature maps after applying all the CNN layers

```
>>> model.compute_output_shape(input_shape=(None, 64, 64, 3))
TensorShape([None, 8, 8, 256])
```

- There are 256 feature maps (or channels) of size 8 × 8
- Now we can add a fully connected layer to get to the output layer with a single unit
  - If we flatten the feature maps, the number of input units to this fully connected layer will be 8 × 8 × 256 = 16384

```
>>> model.add(tf.keras.layers.Flatten())
>>> model.add(tf.keras.layers.Dense(16384, activation='relu'))
```

Implementation of a CNN with TensorFlow

3. BUILDING THE MODEL

- Alternatively, we can insert a global average pooling:
  - a special case of average-pooling when the pooling size is equal to the size of the input feature maps
  - It reduces the hidden units from 16384 to 256

```
>>> model.add(tf.keras.layers.GlobalAveragePooling2D())
>>> model.compute_output_shape(input_shape=(None, 64, 64, 3))
TensorShape([None, 256])
```

- Finally, we can add a fully connected (dense) layer to get a single output unit

```
>>> model.add(Dense(1, activation='sigmoid'))
>>> model.compile(optimizer=Adam(), loss=BinaryCrossentropy(), metrics=['accuracy'])
```

Implementation of a CNN with TensorFlow

3. BUILDING THE MODEL

- By using model.summary() a table with the network layers and their hyperparameters is printed out:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              multiple                  896
_________________________________________________________________
max_pooling2d (MaxPooling2D) multiple                  0
_________________________________________________________________
dropout (Dropout)            multiple                  0
_________________________________________________________________
conv2d_1 (Conv2D)            multiple                  18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 multiple                  0
_________________________________________________________________
dropout_1 (Dropout)          multiple                  0
_________________________________________________________________
conv2d_2 (Conv2D)            multiple                  73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 multiple                  0
_________________________________________________________________
conv2d_3 (Conv2D)            multiple                  295168
_________________________________________________________________
global_average_pooling2d (Gl multiple                  0
_________________________________________________________________
dense (Dense)                multiple                  257
=================================================================
Total params: 388,673
Trainable params: 388,673
Non-trainable params: 0
```

Implementation of a CNN with TensorFlow

4. TRAINING THE MODEL

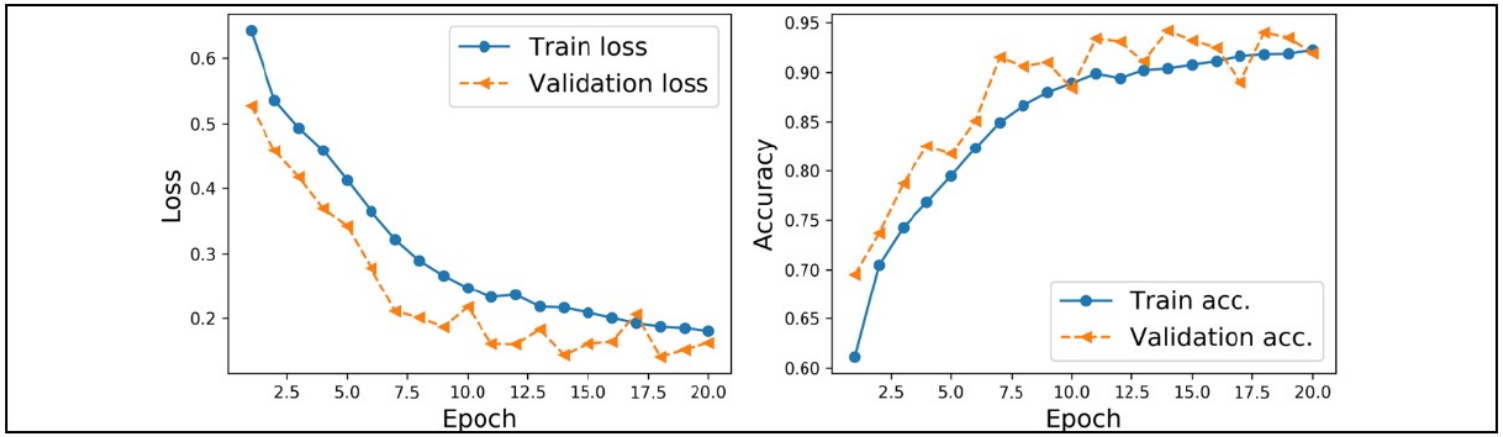
- Let's now train the model, with the *fit* method

```
>>> history = model.fit(ds_train, validation_data=ds_valid,
...                     epochs=20,
...                     steps_per_epoch=steps_per_epoch)
```

- And let's visualize the learning curves:

```
>>> hist = history.history
```

- hist['loss'], hist['val_loss'], hist['accuracy'] and hist['val_accuracy'] are the arrays with loss and accuracy for train and validation dataset, as a function of the training epochs



Implementation of a CNN with TensorFlow

4. TRAINING THE MODEL

- The losses for the training and validation have not converged to a plateau region.
- Using the fit() method, we can continue training for an additional 10 epochs

```
>>> history = model.fit(ds_train, validation_data=ds_valid,
...                     epochs=30, initial_epoch=20,
...                     steps_per_epoch=steps_per_epoch)
```

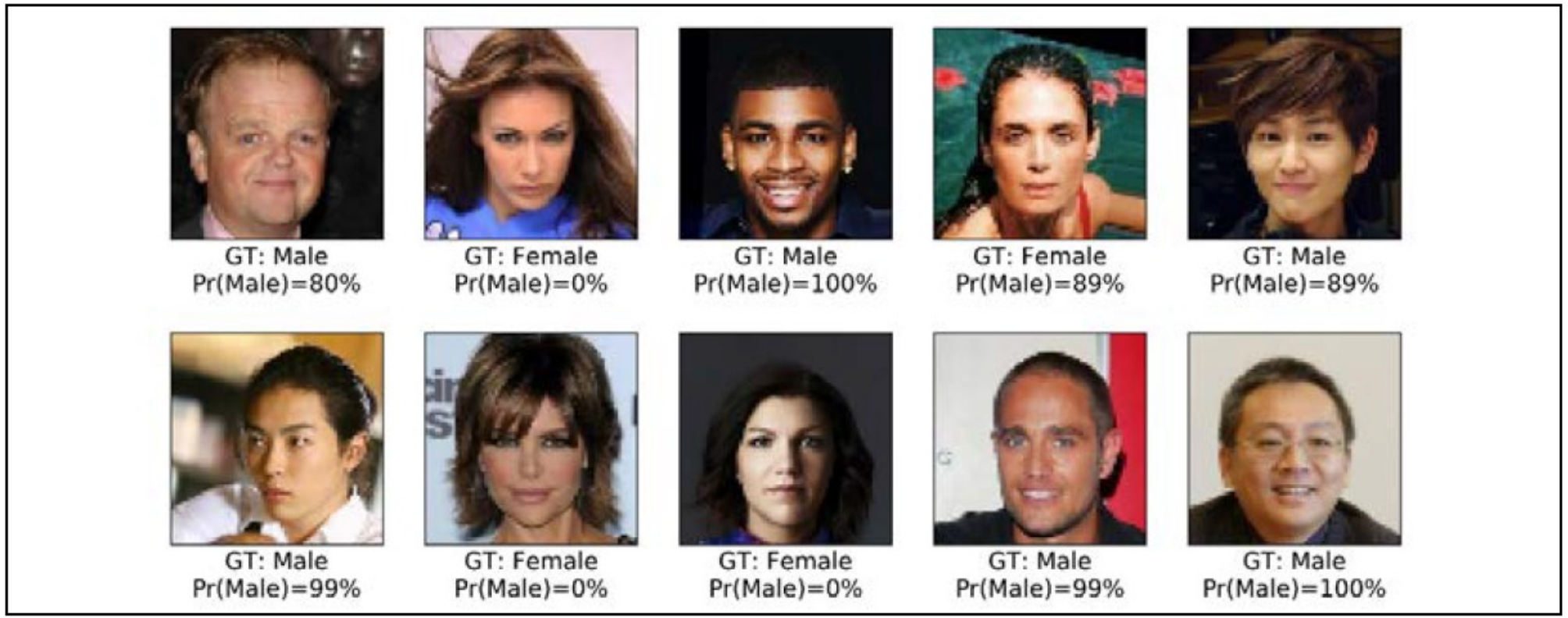
- Once we are happy with the learning curves, we can evaluate the model on the holdout test dataset

```
>>> test_results = model.evaluate(ds_test)
>>> print('Test Acc: {:.2f}%'.format(test_results[1]*100))
Test Acc: 94.75%
```

Implementation of a CNN with TensorFlow

5. APPLYING THE MODEL

- Finally, we can get the prediction results on some test examples using *model.predict()*



Hands on CNN

Today you will create a CNN model for classifying dogs and cats

- We will provide a dataset containing photos of dogs and cats
- Pre-processing steps to images
- Creating a model
- Training and testing
- Evaluating performances
- Improving the model!

References

- Raschka, Sebastian. Python machine learning. Packt publishing ltd, 2015
- Dive Into Deep Learning http://d2l.ai/chapter_preface/index.html