# Convolutional Neural Networks

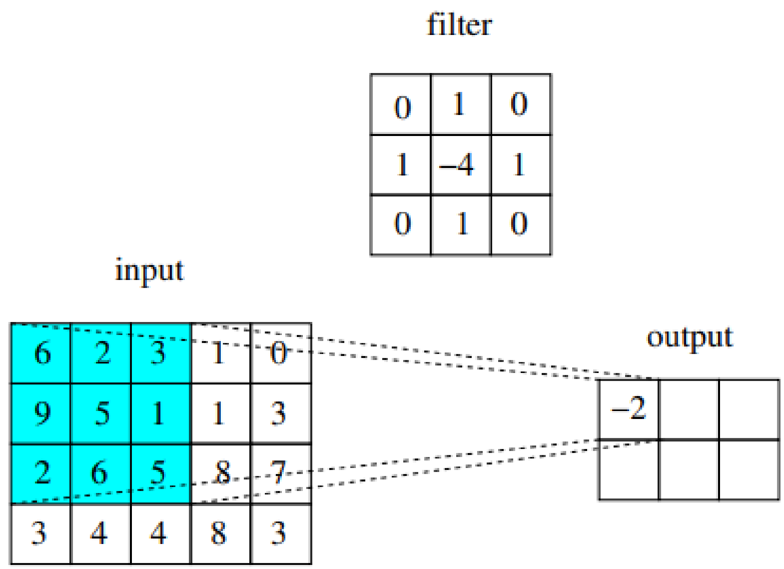
As we know, in a NN we have dense layers (in which each node is connected to the previous layer). In CNN, there are some differences.

In a CNN, each neuron is computed as a function of only some specific neurons of the previous layer (in particular, a neighbourhood of the previous layer).

To determine the neighbourhood of each pixel, we usually just select a kernel of weights, which is then computed through a dot product with the rest of the neurons.

## Filters and convolutions

We have a grid of weights (a kernel or filter), which we then slide.




As we've said: - the activation of a neuron is not influenced from all neurons of the previous layer, but only from a *small subset of adjacent neurons*: his **receptive field**. - every neuron works as a convolutional filter. Weights are shared: *every neuron performs the same trasformation on different areas of its input*. - with a *cascade* of convolutional filters intermixed with activation functions we get complex non-linear filters *assembing local features* of the image into a global structure.

### CNNs and Images

Convolutions are very useful expecially for extracting features from images. An image is coded as a numerical matrix (array) which can be either grascale or rgb.

Some interesting features that we can extract from images are: - Edges, angles, ...: points where there is a discontinuity, i.e. a *fast variation of the intensity*.

We measure variations of intensitites by means of *derivatives* and we can compute discrete approximations of derivatives convolving *simple linear filters*.

## Computing approximations of derivatives

If we think of this variation as a surface, we may notice that probebly in that point this repentine change can be translated in a high derivative. We can approxiamte the derivateive by meanse of the finite central difference:

#### Finite central difference

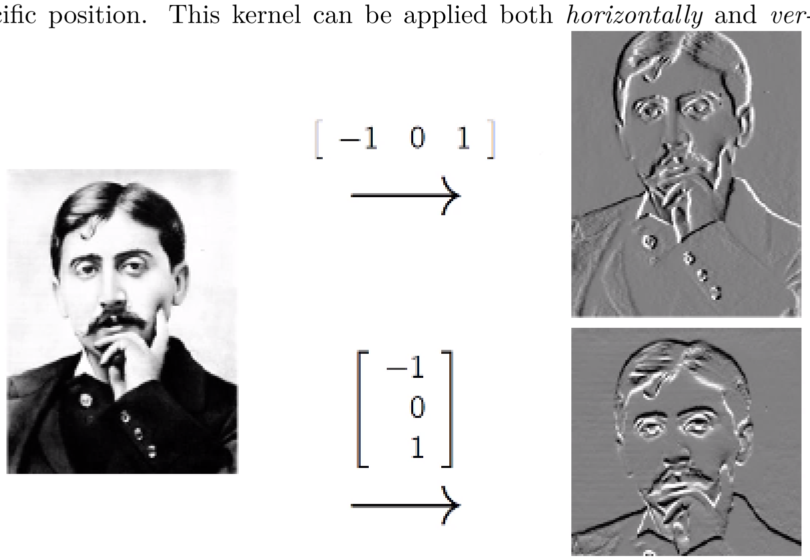
$$\frac{f(x+h)-f(x-h)}{2h} = f'(x) + O(h^2)$$

We essentially are computing $\frac{y_1 - y_0}{x_1 - x_0}$.

Usually, $h = 1$ (since we can't take 0) pixel, and negleting the costant 1/2 we compute with the following filter

$$[-1 \ 0 \ 1]$$

This kernel is quite interesting in image processing and allows us to approximate a derivative of the input image (w.r.t. the difference of the intensity of the pixel) in a specific position. This kernel can be applied both *horizontally* and *vertically*.




From the input image we extract the visible contours, using different orientations of the kernel.

In general, the kernel is a *pattern* of the image *that we are interested in*. We can have many, complex patterns, we look for this pattern over the input. The weak point is that the pattern is linear, and so only part of the pattern is recognixed. It's better to combine pattern in successive elaborations.

## Code Demo

```
kernel = np.zeros((3,3))
kernel[:,0] = -1
kernel[:,2] = 1

img = cv2.filter2D(image, -1, kernel) # allows to apply our kernel
fig, ax = plt.subplots(1, figsize=(12,8))
plt.imshow(img)
```

The kernel that we obtain is:

```
array([[-1.,  0.,  1.],
       [-1.,  0.,  1.],
       [-1.,  0.,  1.))
```

And the result is: 

A kernel that would shift the image looks something like this:

```
array([[0.,  0.,  0.],
       [0.,  0.,  1.],
       [0.,  0.,  0.))
```

You can find the full code for this demo here: link