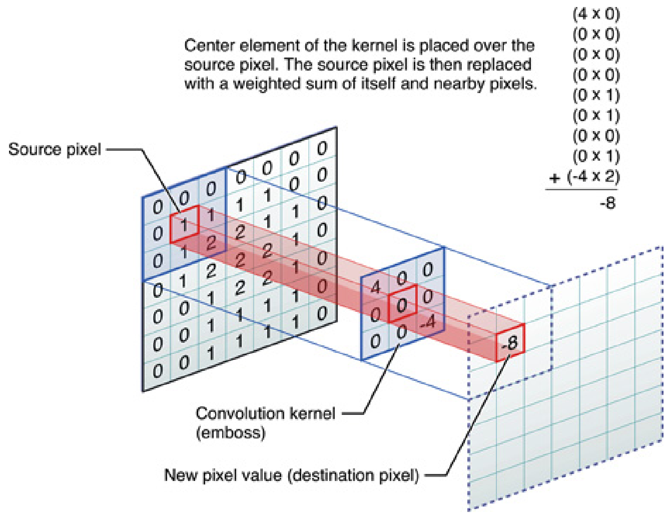
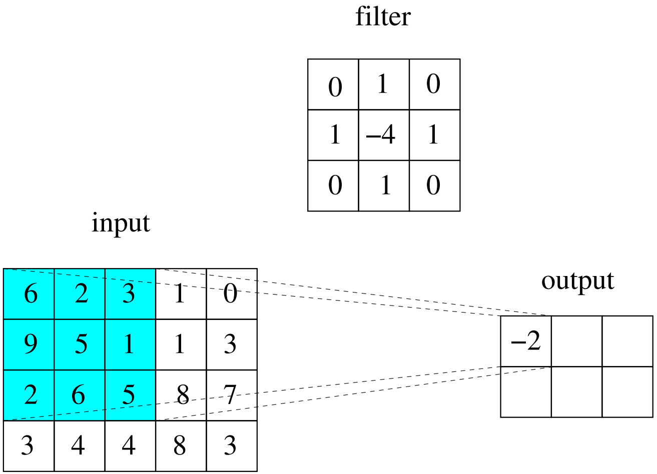
Lesson 4

Convolutional Neural Networks

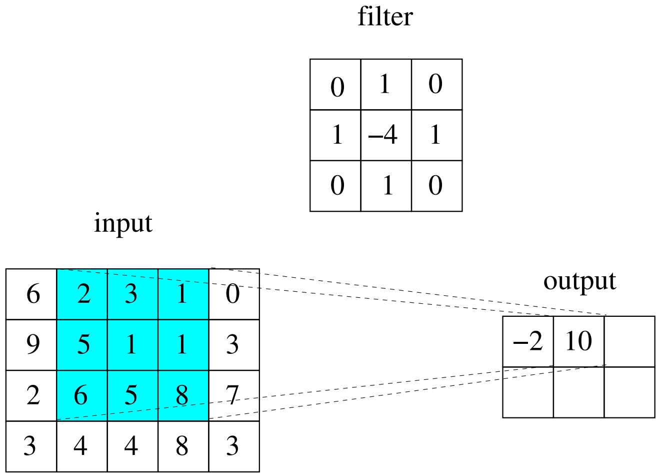
Filters and convolutions


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

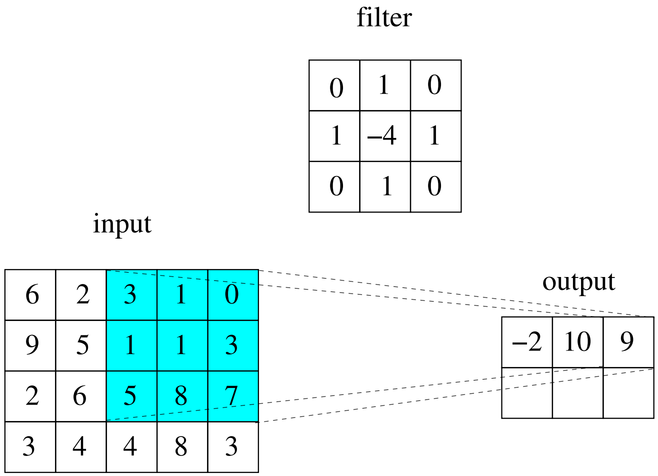
Filters and convolutions



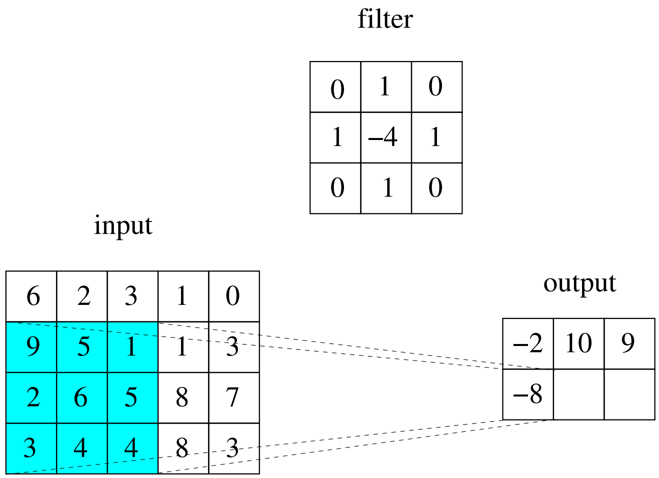
Filters and convolutions



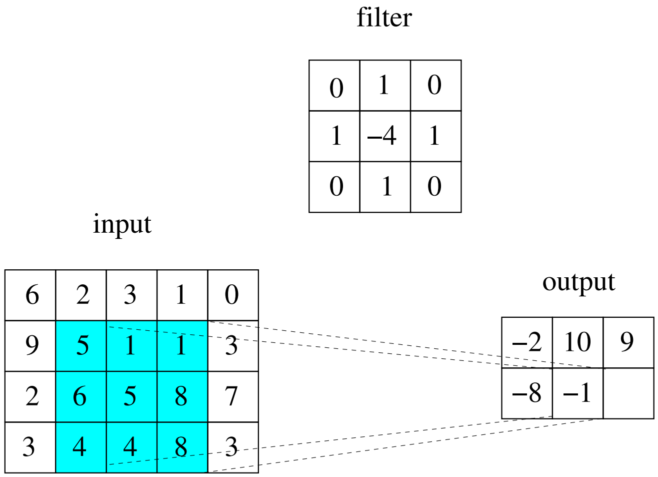
Filters and convolutions



Filters and convolutions



Filters and convolutions



filter

0	1	0
1	–4	1
0	1	0

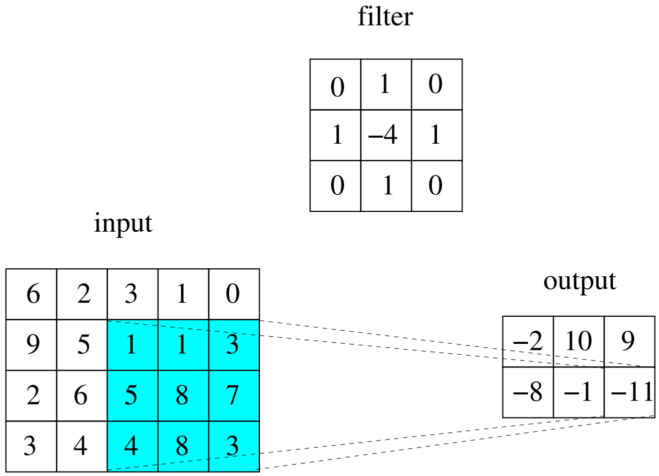
input

6	2	3	1	0
9	5	1	1	3
2	6	5	8	7
3	4	4	8	3

output

–2	10	9
–8	–1	

Filters and convolutions



Loose connectivity and shared weights

- 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 perform 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.

A parenthesis

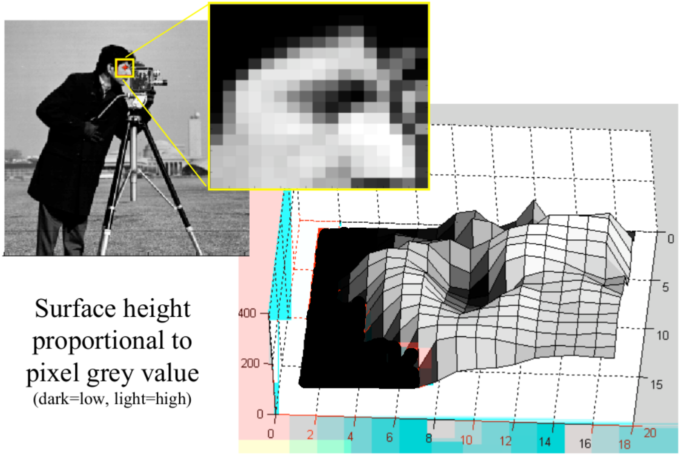
About the relevance of convolutions for image processing

Images are numerical arrays

An image is coded as a numerical matrix (array)
grayscale (0-255) or rgb (triple 0-255)

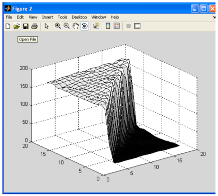
$$\begin{bmatrix} 207 & 190 & 176 & 204 & 204 & 208 \\ 110 & 108 & 114 & 112 & 123 & 142 \\ 94 & 100 & 96 & 121 & 125 & 108 \\ 95 & 86 & 81 & 84 & 88 & 88 \\ 69 & 51 & 36 & 72 & 78 & 81 \\ 74 & 97 & 107 & 116 & 128 & 133 \end{bmatrix}$$

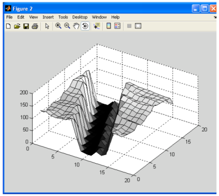
Images as surfaces



Interesting points

Edges, angles, ...: points where there is a discontinuity, i.e. a fast variation of the intensity





More generally, are interested to identify patterns inside the image.

The key idea is that the kernel of the convolution expresses the pattern we are looking for.

Example: finite derivative

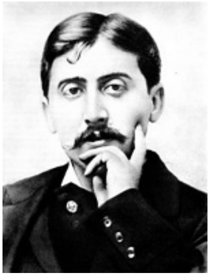
Suppose we want to find the positions inside the image where there is a sudden horizontal passage from a dark region to a bright one. The pattern we are looking for is

$$\begin{bmatrix} -1 & 1 \end{bmatrix}$$

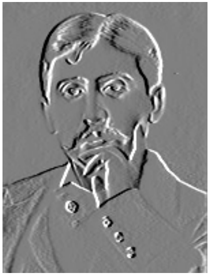
or, varying the distance between pixels:

$$\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

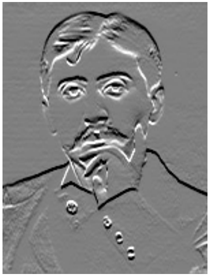
The finite derivative at work



$$\begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \longrightarrow$$



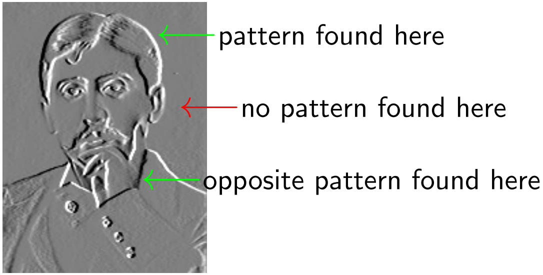
$$\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \longrightarrow$$



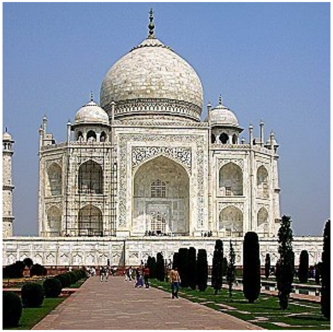
Recognizing Patterns

Each neuron in a convolutional layer gets activated by specific patterns in the input image.

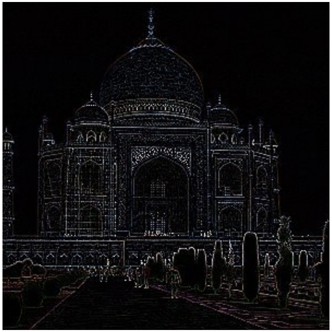
$$\text{pattern} = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$



Another example: the finite laplacian



$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \longrightarrow$$



Discovering patterns

But how to find good patterns?

Usual idea:

instead of using human designed pre-defined patterns, let the net learn them.

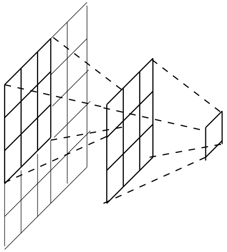
Particularly important in deep architectures, because:

- stacking kernels we **enlarge their receptive fields** (see next slide)
- adding non-linear activations we synthesize complex, **non-linear kernels**

Receptive field

The receptive field of a (deep, hidden) neuron is the dimension of the input region influencing it.

It is equal to the dimension of an input image producing (without padding) an output with dimension 1.



A neuron cannot see anything outside its receptive field!

We may also rapidly enlarge the receptive fields by means of **downsampling** layers, e.g. pooling layers or convolutional layers with non-unitarian stride

An example

[DEMO]

Next argument

Tensors for 2D processing

Tensors

Convolutional Networks process Tensors. A Tensor is just an multidimensional array of floating numbers.

The typical tensor for 2D images has **four** dimensions:

$$batchsize \times width \times height \times channels$$

Features maps are **stacked** along the channel dimension.

At start, for a color image, we just have 3 channels: r,g,b.

How do kernel operate along the channel dimension?

Tensors

Convolutional Networks process Tensors. A Tensor is just an multidimensional array of floating numbers.

The typical tensor for 2D images has **four** dimensions:

$$\textit{batchsize} \times \textit{width} \times \textit{height} \times \textit{channels}$$

Features maps are **stacked** along the channel dimension.

At start, for a color image, we just have 3 channels: r,g,b.

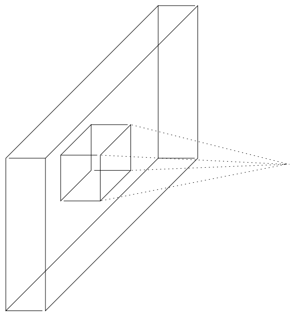
How do kernel operate along the channel dimension?

Dense processing along channel axis

Unless stated differently (e.g. in separable convolutions), a filter operates on all input channels in parallel.

So, if the input layer has depth D, and the kernel spatial size is NxM, the actual dimension of the kernel will be

NxMxD



The convolution kernel is tasked with simultaneously mapping cross-channel correlations and spatial correlations

Spatial dimension of the resulting feature map

Each kernel produces a single feature map.
Feature maps produced by different kernels are stacked along the channel dimension: the number of kernels is equal to the channel-**depth** of the next layer.

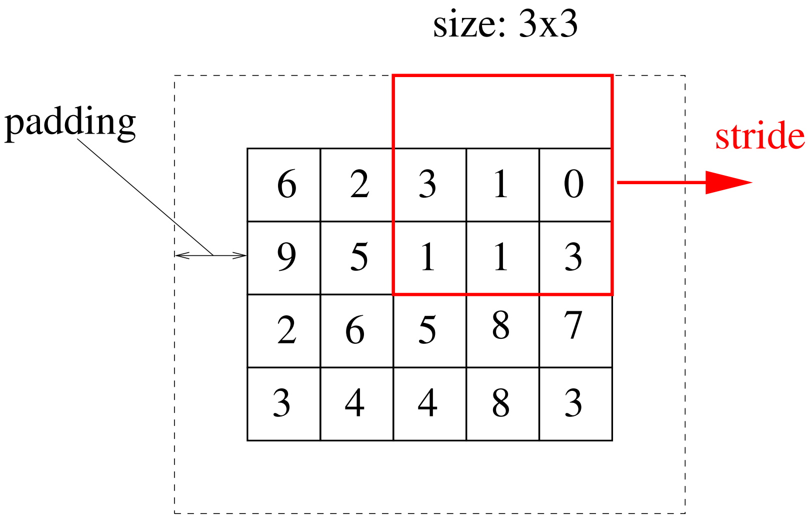
The **spatial** dimension of the feature map depends from two configurable factors:

- **paddding**: extra space added around the input
- **stride**: kernel deplacement over the input during convolution

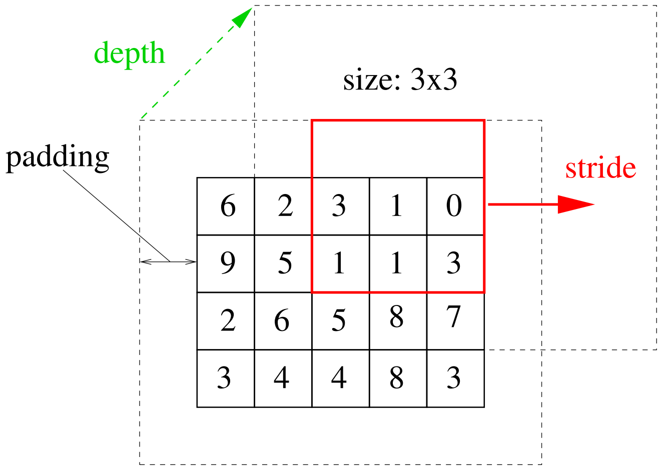
Relevant parameters for convolutional layers

- **kernel size**: the dimension of the linear filter.
- **stride**: movement of the linear filter. With a low stride (e.g. unitary) receptive fields largely overlap. With a higher stride, we have less overlap and the dimension of the output get smaller (lower sampling rate).
- **padding** Artificial enlargement of the input to allow the application of filters on borders.
- **depth**: number of features maps (stacked along the so called channel axis) that are processed in parallel.
  The depth of the output layer depends from the number of different kernels that we want to synthesize (each producing a different feature map).

Configuration params for conv2D layers



Configuration params for conv2D layers



Input-output spatial relation

Along each axes the dimension of the output is given by the following formula

$$\frac{W_{in} + P - K}{S} + 1 = W_{out}$$

where:

W = spatial dimension of the input
P = padding
K = Kernel size
S = Stride

Receptive field

We may compute the receptive field reverting the previous formula (with no padding):

$$W_{in} = K + (W_{out} - 1) * S$$

Suppose that we pass two layers with K=3 and S=2.
Initially $W_0 = W_{out} = 1$ (we consider a single neuron).
After the first layer we have

$$W_1 = 3 + (1 - 1) * 2 = 3$$

After two layers:

$$W_2 = 3 + (3 - 1) * 2 = 7$$

Pooling

An alternative way for reducing the spatial dimension of the input/ increase the receptive fields of neurons, is to use pooling layers.

In a pooling layer each neuron simply takes the mean or maximal value in its "kernel" region. The default stride for pooling layers is the spatial dimension of the kernel.

Pooling layers have a double advantage:

- they reduce the spatial dimension
- they provide some tolerance to translations