Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 6: Convolutional neural networks

Lecturer: S. Scardapane

SAPIENZA
UNIVERSITÀ DI ROMA

Introduction

Why fully-connected layers are not enough

An **image** is a 3-dimensional tensor $X_{(h,w,c)}$, where:

- h is the **height** of the image (e.g., 512 pixels).
- w is the **width** of the image (e.g., 1024 pixels).
- c is the number of **channels** (e.g., 3 channels for a RGB image, 1 channel for a greyscale image).

The first two dimensions have a precise *grid* ordering, while the channels do not have a precise ordering (i.e., we can switch RGB to GBR or BRG with no information loss).

A simple way to process an image is to **vectorize** it by stacking all its values:

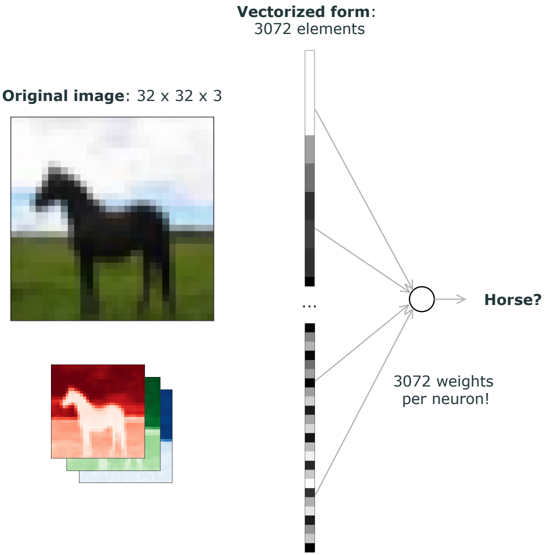
$$\underset{(hwc)}{\mathbf{x}} = \text{vect}(X)\,.$$

Once this is done, we can apply what we know, e.g., a fully-connected layer:

$$\mathbf{h} = \phi(\mathbf{W}\mathbf{x})\,. \tag{1}$$

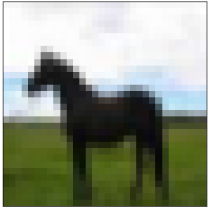
Can you see what is wrong with this approach?

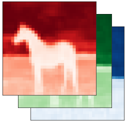
Vectorization of an image



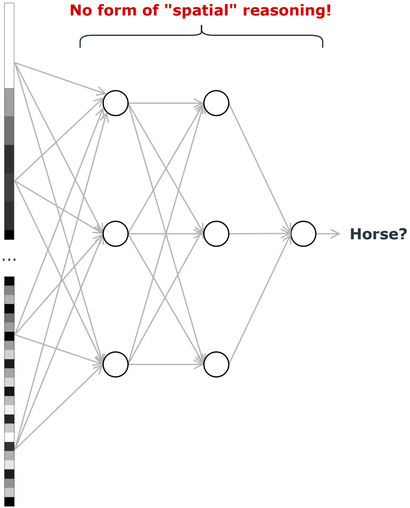
Where has the image gone?

Original image: 32 x 32 x 3





Vectorized form:
3072 elements



We have lost all the spatial information after the first operation, i.e., we cannot compose the previous block multiple times. A simple way to solve this would be:

$$H = \text{unvect}(\phi(\mathbf{W} \cdot \text{vect}(X))) \tag{2}$$

However, we still need a *huge* number of parameters: for example, for a 1024×1024 RGB image we need $\approx 3M$ parameters for a logistic regression!

Next, we show how we can properly incorporate this information, to define a layer targeted for image-like data.

As a running example to visualize what follows, consider a 1D sequence (think of this as "4 *pixels with a single channel*"):

$$\mathbf{x} = [x_1, x_2, x_3, x_4]$$

In this case, we do not need any reshaping operations, and the previous layer (with $c' = 1$) can be written as:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & W_{14} \\ W_{21} & W_{22} & W_{23} & W_{24} \\ W_{31} & W_{32} & W_{33} & W_{34} \\ W_{41} & W_{42} & W_{43} & W_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

Convolutional neural networks

Convolutional layers

We want a layer of the form:

$$\underset{(h,w,c')}{H} = f\underset{(h,w,c)}{(X)},$$

with the following properties:

- the output tensor must exploit the 'spatial information' contained in the image;
- It must be efficient (with a small number of parameters);
- It must be *composable* and *differentiable*, i.e., we want to do:

$$Y = (f_l \circ \ldots \circ f_2 \circ f_1)(X)$$

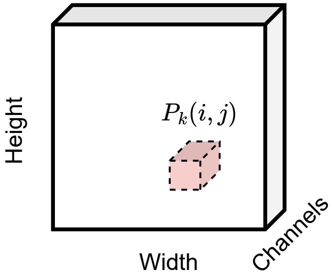
We can define many distances between two pixels i, j and i', j', e.g.:

$$d(i, j, i', j') = \max\left\{|i - i'|, |j - j'|\right\} \ .$$

Fix an odd number $s = 2k + 1$. A **patch** is a sub-image centered at (i, j), containing all pixels (i', j') under distance k:

$$\underset{(s,s,c)}{P_{i,j,k}} = [X]_{i-k:i+k,j-k:j+k,:} \ . \tag{3}$$

We can think of a patch as a small slice of the original tensor:



The size of the patch will be called the **filter size** or **kernel size**.

> An image layer is **local** if $[H]_{i,j}$ only depends on $P_{i,j,k}$ for some k.

We can achieve this by restricting the linear operation to the single patch:

$$H_{ij} = \phi\left(\underbrace{\mathbf{W}_{ij}}_{\text{Position-dependent weight matrix}} \cdot \overbrace{\mathrm{vect}(P_k(i,j))}^{\text{Flattened patch (of shape } s^2c'c)}\right)$$

where we have a separate weight matrix $\underset{(c',ssc)}{\mathbf{W}_{i,j}}$ for each location. These are called **locally-connected** layers.

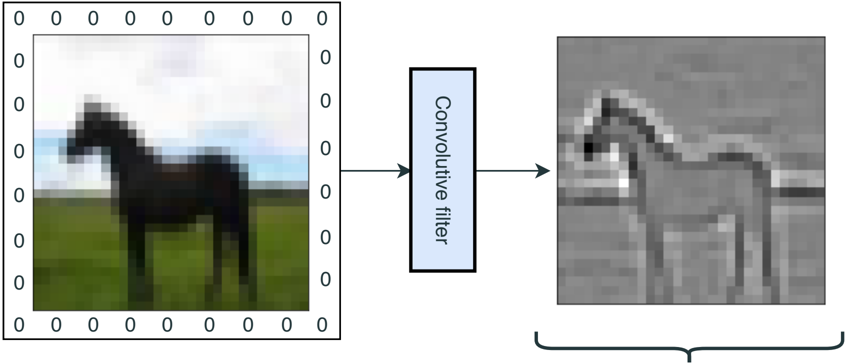
Considering our toy example, assuming for example $k = 1$ (hence $s = 3$) we can write the resulting operation as:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{12} & W_{13} & 0 & 0 \\ W_{21} & W_{22} & W_{23} & 0 \\ 0 & W_{31} & W_{32} & W_{33} \\ 0 & 0 & W_{41} & W_{42} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

The operation is not defined for x_1 and x_4. Instead of shortening the output, we can add 0 on the border whenever necessary:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & 0 & 0 & 0 \\ 0 & W_{21} & W_{22} & W_{23} & 0 & 0 \\ 0 & 0 & W_{31} & W_{32} & W_{33} & 0 \\ 0 & 0 & 0 & W_{41} & W_{42} & W_{43} \end{bmatrix} \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}$$

This is called **zero padding**.



Same size as the
original image

The previous layer keeps the spatial information, but it is definitely not efficient: in total, it requires $o \cdot ssc \cdot h \cdot w$ parameters.

Fortunately, there is another nice property we can exploit.

> An image layer is **translational equivariant** if $P_{i,j,k} = P_{i',j',k}$ implies $[H]_{i,j} = [H]_{i',j'}$.

Informally, we want to recognize something *irrespective* of where it appears in the image, i.e., if something moves (the patch) we want the output feature to move 'with it'.

We can achieve translational equivariance easily by *sharing* the same weights across all locations, i.e., $\mathbf{W}_{i,j} = \mathbf{W}$:

$$[H]_{i,j} = \phi(\mathbf{W} \cdot \text{vect}(P_{i,j,k})) \,, \tag{4}$$

The resulting layer is called a **convolutional layer**. It has all the properties we were looking for, including efficiency (we have only $c' \cdot ssc$ parameters).

Remember that in general we always consider a version with bias:

$$[H]_{i,j} = \phi(\mathbf{W} \cdot \text{vect}(P_{i,j,k}) + \underset{(c')}{\mathbf{b}}) \,. \tag{5}$$

The final convolutional layer in our toy example is:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_2 & W_3 & 0 & 0 \\ W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \\ 0 & 0 & W_1 & W_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{6}$$

where we now have only three weights $\mathbf{W} = [W_1, W_2, W_3]^\top$. Note the special (**Toeplitz**) structure of the matrix – the convolutional layer remains a linear transformation.

An equivalent way to define convolution is to consider a 4-dimensional weight tensor $\underset{(s,s,c,c')}{W}$, with a scalar function to convert between offsets:

$$t(i) = i - k - 1 \tag{7}$$

We now rewrite the output of the layer with explicit summations across the axes:

$$H_{ijz} = \sum_{i'=1}^{2k+1} \sum_{j'=1}^{2k+1} \sum_{d=1}^{c} [W]_{i',j',z,d} [X]_{i'+t(i),j'+t(j),d} \tag{8}$$

In signal processing terminology, this is a filtering operation exploiting a **finite impulse response** filter.

- $s = 2k + 1$ is called the **kernel size** or **filter size**. It is a hyper-parameter of the layer, together with the number c' of output channels.
- In accordance with signal processing, the elements of the matrix $\mathbf{W}$ (or the equivalent tensor W) are called **filters**.
- A single slice $[H]_{:,:,a}$ is called an **activation map**. Sometimes, we distinguish between pre-activation (before ϕ) and post-activation.

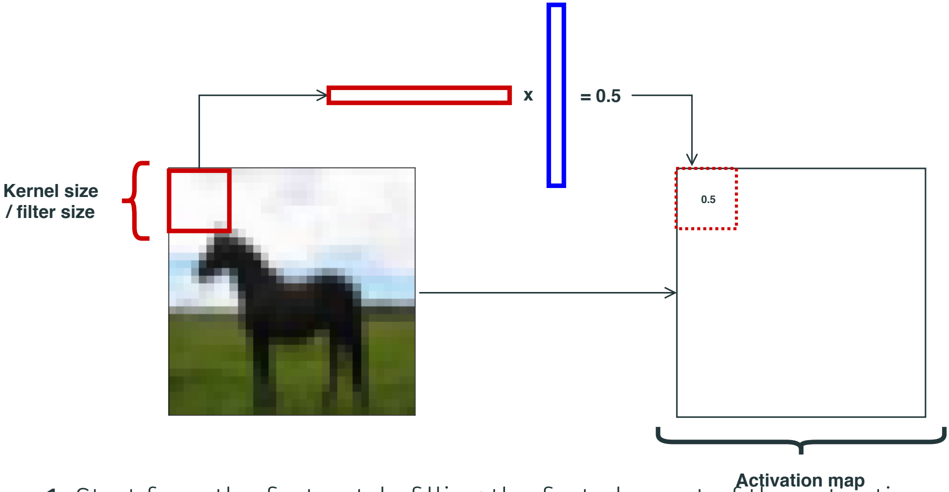


Figure 1: Start from the first patch, filling the first element of the activation map.

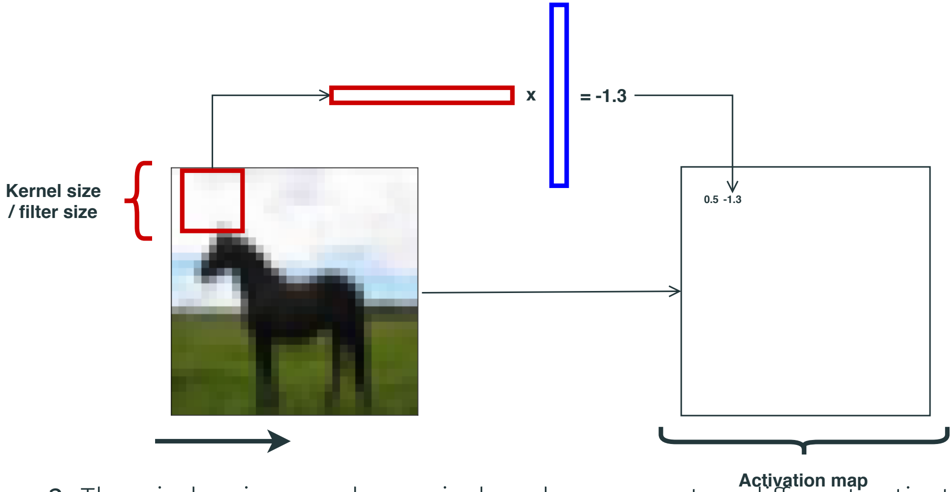


Figure 2: The window is moved one pixel, and we compute a different activation.

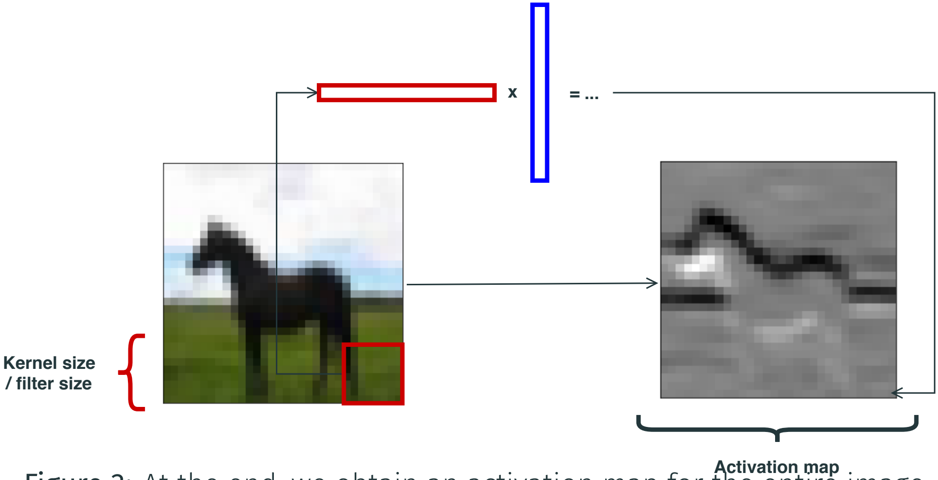
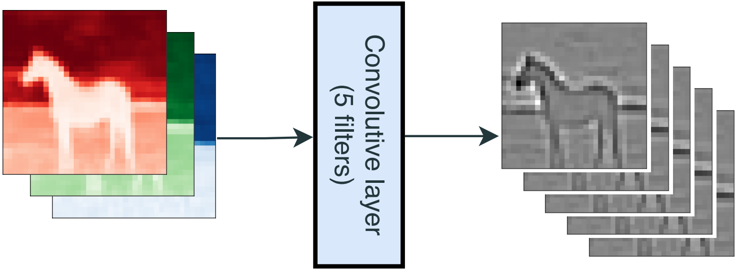


Figure 3: At the end, we obtain an activation map for the entire image.

The previous operation is shown for a single filter. Stacking many filters together gives us the complete **convolutional layer**.



Suppose we stack several convolutional layers:

$$H = (f_3 \circ f_2 \circ f_1)(X)$$

The **receptive field** of $[H]_{i,j}$ is the subset of X that contributed to its computation.

For one layer, the receptive field is just $P_{i,j,k}$. For two layers, however (with the same kernel size), it becomes $P_{i,j,2k}$. This justifies our choice of locality: even if a *single* layer is highly localized, *many* layers can still process the entire image at once, since the receptive field increases linearly.

Most frameworks, including TensorFlow, provide a primitive with an efficient low-level implementation:

```
# Image (with mini-batch dimension)
X = tf.random.normal((1, 64, 64, 3))

# Filters (filter size = 5, output filters = 100)
W = tf.random.normal((5, 5, 3, 100))

# Convolution
H = tf.nn.conv2d(X, W, 1, 'SAME')
print(H.shape) # (1, 64, 64, 100)
```

https://www.tensorflow.org/api_docs/python/tf/nn/conv2d.

Convolutional neural networks

Defining the network

We can define a convolutional block by interleaving convolutional layers with activation functions:

$$H = (\phi \circ \text{Conv} \circ \ldots \circ \phi \circ \text{Conv})(X)$$

Convolutional blocks can modify the number of channels, but they keep the spatial resolution (h, w) constant. We might want to reduce the resolution in-between blocks, to make the networks faster and more efficient.

This is also justified from a signal processing perspective, where **multi-resolution** filter banks are common.

In a convolution with **stride**, we compute only 1 every s elements of the output tensor H, where s is the stride parameter.

For example, for $s = 2$, we have:

$$\underbrace{[H]_{i,j}}_{(h/2,w/2,c')} = \phi(\mathbf{W} \cdot \text{vect}(P_{2i-1,2j-1,k}))\,,$$

The `tf.nn.conv2d` function we saw before requires, in fact, a stride parameter.

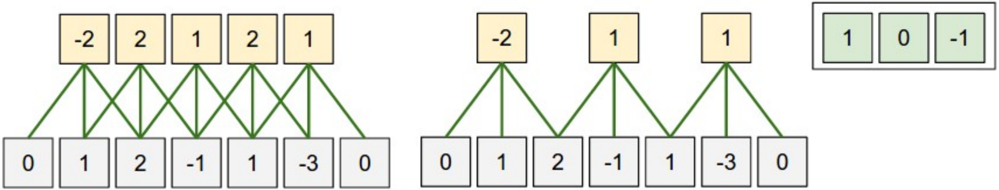


Figure 4: Left figure has stride = 1, right figure has stride = 2. Image source is `http://cs231n.github.io/convolutional-networks/`.

Alternatively, a **max-pooling** (or an **average-pooling**) layer can be used. It computes the maximum (or the average) from small blocks of the input tensor.

Differently from convolutional layers, it is common to consider even-dimensional blocks (2x2, 4x4, ...). It acts on each channel separately.

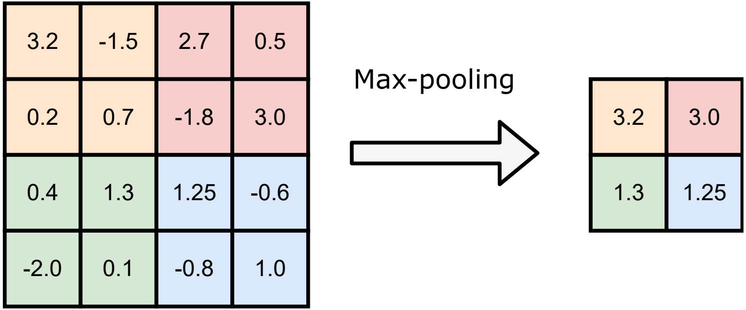


Figure 5: Visualization of max-pooling on a 4×4 image with windows of size 2×2. Note that the maximum operation can be replaced with any differentiable aggregation (e.g., average).

A standard CNN for classification is then composed by:

- Interleaving convolutional and pooling layers;
- Flattening (or **global pooling**);
- A classification block.

Note: with global pooling, the final layer is roughly **invariant** to a translation, despite each convolutional layer being equivariant.

More recent CNNs add many variations on this basic architecture. How to choose the sequence of layers and their hyper-parameters is still an open *model selection* research issue.

$$
\begin{aligned}
\text{Block 1:} \quad & \underset{(h',w',c')}{H} = (f_l \circ \ldots \circ f_2 \circ f_1)(X) && \text{(convolutional or pooling layers)} \\
\text{Block 2a:} \quad & \underset{(h'w'c')}{\mathbf{h}} = \text{vect}(H) && \text{(flattening)} \\
\text{Block 2b:} \quad & \underset{(c')}{\mathbf{h}} = \frac{1}{h'w'} \sum\nolimits_{i,j} [H]_{i,j} && \text{(global pooling)} \\
\text{Block 3:} \quad & \mathbf{y} = \text{softmax}(g(\mathbf{h})) && \text{(e.g., logistic regression)}
\end{aligned}
$$

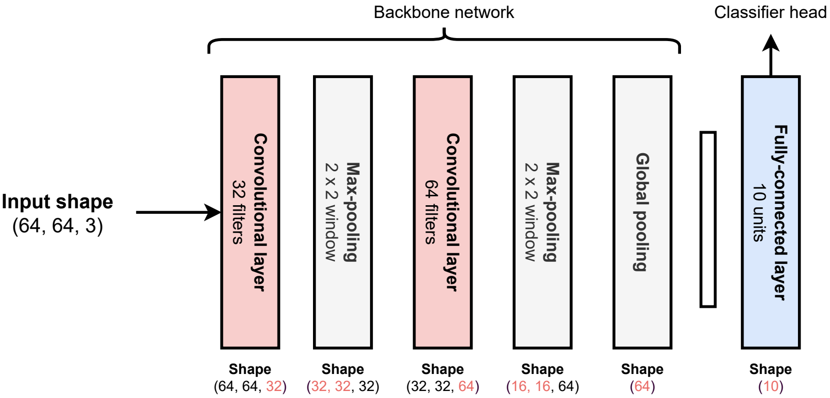


Figure 6: Note how multiple down-sampling layers are required to make the final classification dimensionality manageable.

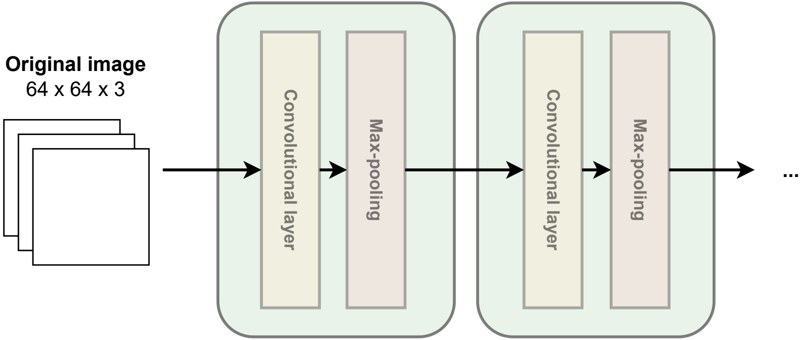


Figure 7: When CNNs tends to become deep, it is simpler to reason in repeating *blocks* made of multiple components. This is easy using the layering abstraction.

Convolutional neural networks

Other notable types of convolutions

One important type of convolutional layer is a 1x1 layer, i.e., a layer with a kernel size of 1 ($k = 0$), also called a **pointwise convolution**.

This can be understood as a pixel-wise operation, which is applied independently at every pixel, with no contribution from the neighbours.

It is especially important when we desire to simply modify the number of channels.

An orthogonal idea is to apply a convolution to each channel *independently*, by combining only information across the spatial dimensions.

The result is a **depth-wise** (separable) convolution:

$$H_{ijc} = \sum_{i'=1}^{2k+1} \sum_{j'=1}^{2k+1} W_{i',j',c} X_{i'+t(i),j'+t(j),c}$$

This idea can also be extended to **group** convolution. A depthwise convolution followed by a pointwise convolution is called a **depthwise-separable convolution** and it is extremely common for modeling efficient architectures.

- **Chapter 7** of the book.