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

Definition of an image

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).

Vectorizing an image

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

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

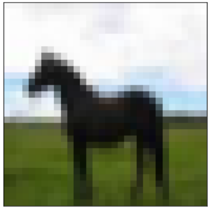
Once this is done, we can apply what we know, e.g., a fully-connected layer:

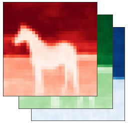
$$\mathbf{h} = \phi(\mathbf{W}\mathbf{x})\,. \tag{1}$$

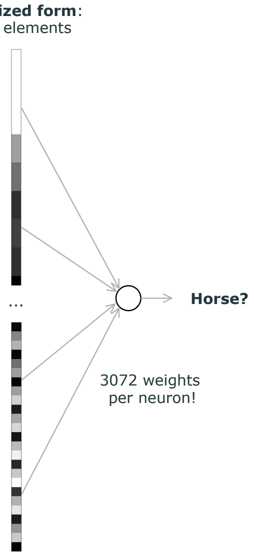
Can you see what is wrong with this approach?

Vectorization of an image

Original image: 32 x 32 x 3

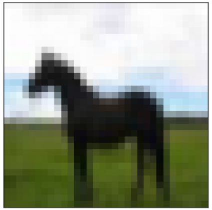


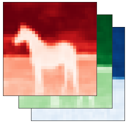




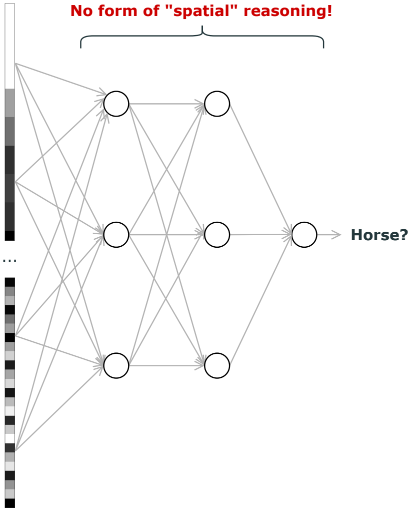
Where has the image gone?

Original image: 32 x 32 x 3





Vectorized form:
3072 elements



Drawbacks

A major drawback of this approach is that it requires a *huge* number of parameters: for example, for a 1024×1024 RGB image we need $\approx 3M$ parameters for a logistic regression!

Worse, we have completely lost all the spatial information after the first operation, i.e., we cannot compose the previous block multiple times.

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

Convolutional neural networks

Convolutional layers

What we want

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:

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

Distance between pixels

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

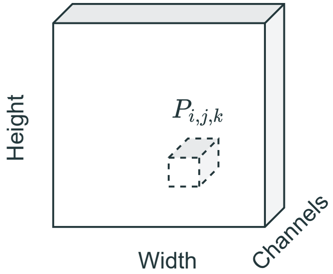
$$d(i,j,i',j') = \max\left\{|i-i'|, |j-j'|\right\} \ .$$

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:

$$\underset{(s,s,c)}{P_{i,j,k}} = [X]_{i-k:i+k,j-k:j+k,:} \ . \tag{2}$$

Visualizing a patch

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**.

Locality

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:

$$\underbrace{[H]_{i,j}}_{\text{Output for pixel (i,j)}} = \phi\left(\overbrace{\mathbf{W}_{i,j} \cdot \text{vect}(P_{i,j,k})}^{\text{Linear combination of patch elements}}\right) ,$$

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

Translational equivariance

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'.

Convolutional layer

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

$$[H]_{i,j} = \phi(\mathbf{W} \cdot \text{vect}(P_{i,j,k}))\,, \tag{3}$$

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:

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

Equivalent formulation

Another way to define the previous operation is to define a 4-dimensional weight tensor $\underset{(s,s,c,c')}{W}$. Then we have:

$$[H]_{i,j,d} = \phi\left(\sum_{i'=-k}^{+k}\sum_{j'=-k}^{+k}\sum_{z=1}^{c}[W]_{i'+k+1,j'+k+1,z,d}[X]_{i+i',j+j',z}\right)$$

If you know some signal processing, this should clarify the meaning of convolution for this operation. We use this formulation in the following to define several variations of the basic convolutional layer.

Terminology time!

- $s = 2k + 1$ 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.

Filter operation 1/3

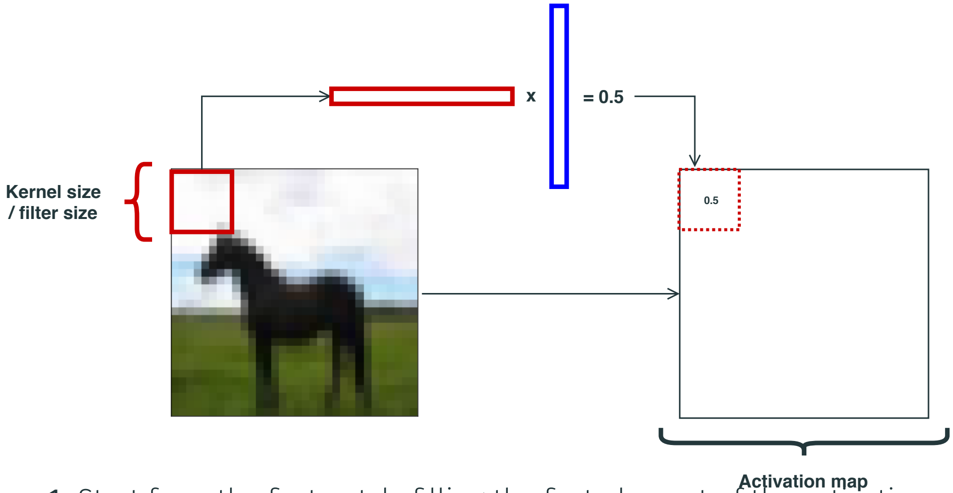


Figure 1: Start from the first patch, filling the first element of the activation map.

Filter operation 2/3

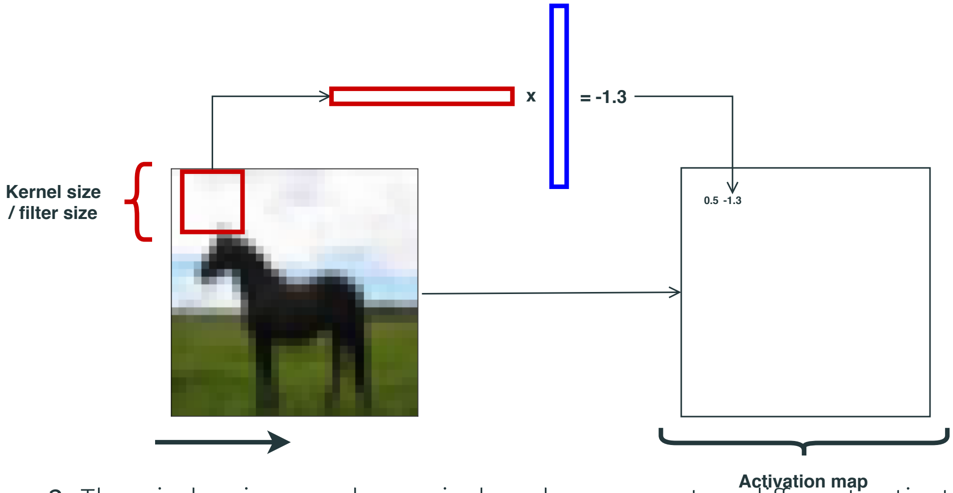


Figure 2: The window is moved one pixel, and we compute a different activation.

Filter operation 3/3

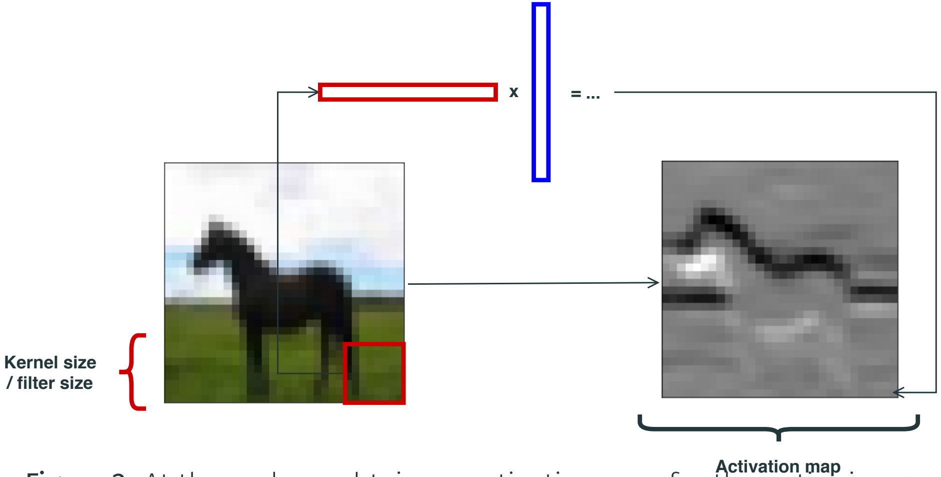
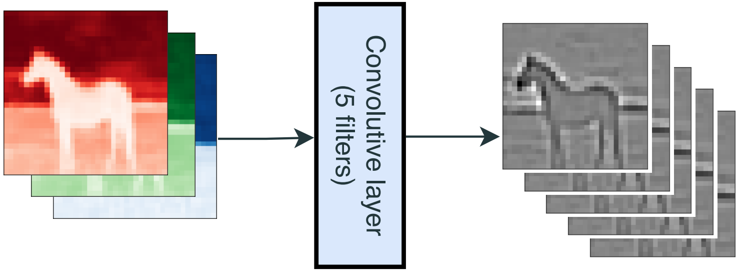


Figure 3: At the end, we obtain an activation map for the entire image.

Convolutive layer

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



Multiple channels

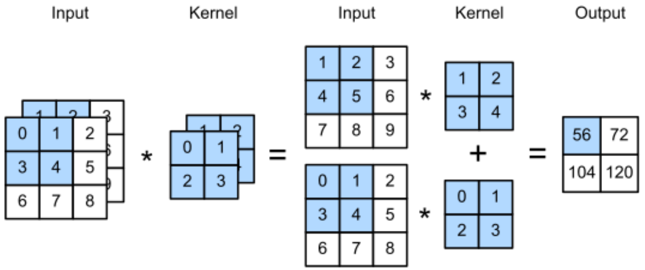


Figure 4: Dive into Deep Learning, Chapter 6.

Receptive field

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.

Convolutional layers are linear

A convolutive layer remains a linear operation, in the sense that convolution can be represented as a matrix $\mathbf{W}$ with a special Toeplitz-like structure. For a simplified example, consider a 1D sequence $[x_1, x_2, x_3, x_3]^\top$. Convolution with a filter $[w_1, w_2]$ of size 2 is equivalent to multiplication by the matrix:

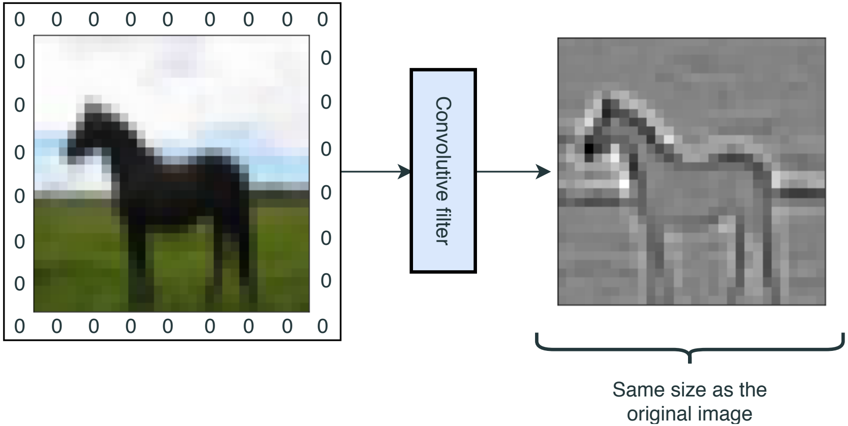
$$\begin{bmatrix} w_1 & w_2 & 0 & 0 \\ 0 & w_1 & w_2 & 0 \\ 0 & 0 & w_1 & w_2 \end{bmatrix} . \tag{5}$$

Concatenating two convolutional layers is equivalent to a single convolutional layer with different kernel size. Like for fully-connected layers, it is important to interleave them with activation functions, typically ReLUs.

Convolutional neural networks

Padding

Convolution as described before cannot be applied to the borders of the image. **Zero padding** can be added to preserve the original width / height.



Padding in practice

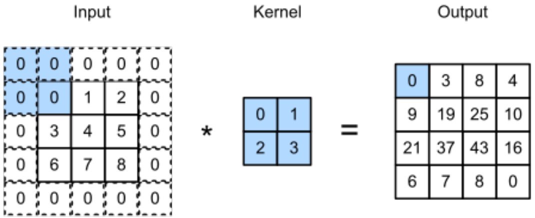


Figure 5: Dive into Deep Learning, Chapter 6.

Implementation

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

```python
# 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

Max pooling and classification

Reducing dimensionality

Convolutional layers (as described up to now) can modify the number of channels, but they keep the spatial resolution (h, w) constant.

In practice, 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.

Stride

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.

Convolution with a larger stride

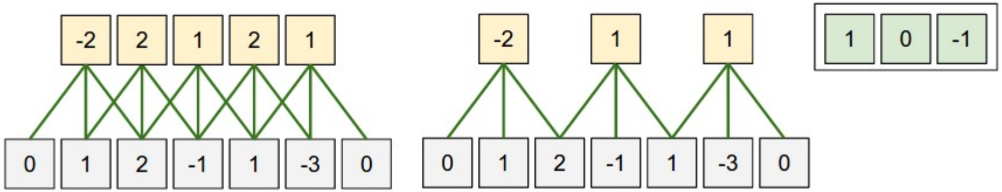


Figure 6: Left figure has stride = 1, right figure has stride = 2. Image source is `http://cs231n.github.io/convolutional-networks/`.

Max-pooling

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.

Visualization of max-pooling

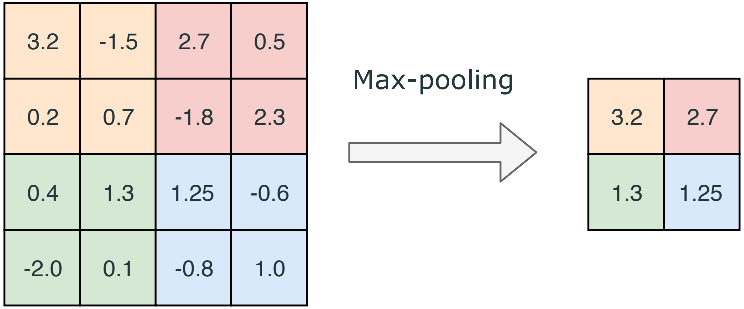


Figure 7: Visualization of max-pooling on a 4×4 image with windows of size 2×2. Note that the maximum operation can be replaced with any differentiable aggregation (e.g., average).

(1)

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.

Complete architecture of a CNN (2)

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

Example of a simple CNN specification

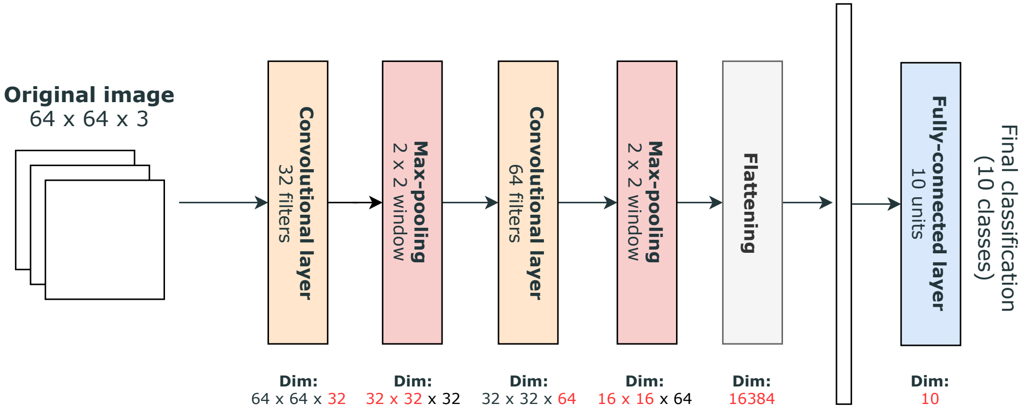


Figure 8: Note how multiple down-sampling layers are required to make the final classification dimensionality manageable.

Designing in blocks

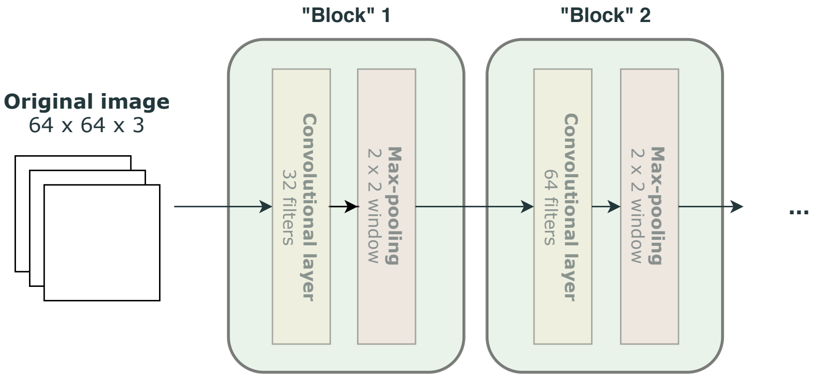


Figure 9: When CNNs tends to become deep, it is simpler to reason in repeating *blocks* made of multiple components. This is easy using the layering abstraction.

Convolutional neural networks

Other notable types of convolutions

1x1 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.

Depth-wise convolutional layer

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:

$$\underset{(h,w,c)}{[H]_{i,j,d}} = \phi\left(\sum_{i'=-k}^{+k}\sum_{j'=-k}^{+k}\underset{(s,s,c)}{[W]_{i'+k+1,j'+k+1,d}}[X]_{i+i',j+j',d}\right)$$

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.

The inception block

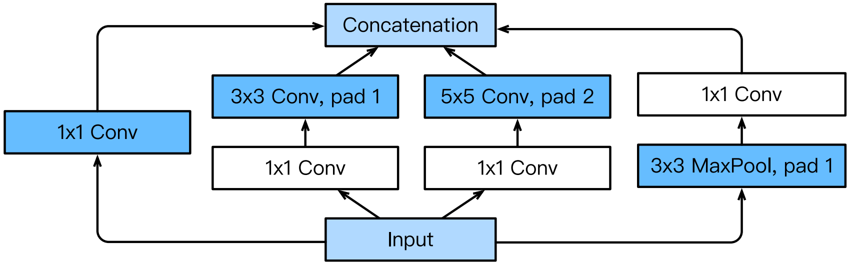


Figure 10: The inception block combines convolutions with different kernel sizes. Source: Dive into Deep Learning, Chapter 7.4.

Szegedy, C. et al., 2015. **Going deeper with convolutions**. In *Proceedings of IEEE CVPR* (pp. 1-9).

Additional material

Convolutions and the Fourier transform

About this material

We now describe briefly the relation between the convolutional layer we defined and the convolution operator as defined in signal processing.

While this does not provide particular insights in this case, it is the key to understand many recent developments in deep learning, most notably geometric deep learning.

Bronstein, M.M., Bruna, J., Cohen, T. and Veličković, P., 2021. **Geometric deep learning: Grids, groups, graphs, geodesics, and gauges**. *arXiv preprint arXiv:2104.13478*.

Convolution operator

Consider two scalar functions $x(t)$ and $w(t)$. Their **convolution** is defined as:

$$(x \star w)(t) = \int_{-\infty}^{+\infty} x(v)w(t-v)dv\,. \tag{6}$$

Note the minus sign in the definition: an equivalent definition with the plus sign is called **cross-correlation**. Since in our case $w(t)$ is eventually learned this distinction is minor.

Images as signals

Consider the set of integers $\mathcal{I} = \{0, 1, \ldots, n-1\}$. A function $f : \mathcal{I} \to \mathbb{R}$ can be identified with a vector $\underset{(n)}{\mathbf{f}}$, where:

$$[\mathbf{f}]_i = f(i)\,.$$

The reasoning can be extended to an image, which can be interpreted as a function defined over a grid domain.

Circular signals

For the sake of simplicity, we suppose the signal is *circular*, meaning that we can interpret it as a periodic signal in the sense:

$$[\mathbf{x}]_n = [\mathbf{x}]_0 \,, \tag{7}$$

or, more in general:

$$[\mathbf{x}]_k = [\mathbf{x}]_{k \bmod n} \,. \tag{8}$$

This is useful because it removes issues at the boundary, although the reasoning can be extended to the non-circular case.

Take two signals $\mathbf{x}$ and $\mathbf{w}$. Their circular **discrete convolution** is obtained by discretizing the integral:

$$[\mathbf{x} \star \mathbf{w}]_i = \sum_{j=0}^{n-1} [\mathbf{x}]_{j \bmod n} [\mathbf{w}]_{i-j \bmod n} \,, \tag{9}$$

This is similar to the convolutional layer, except for how we handle boundary conditions due to the previous cyclic assumption.

Convolution theorem

Convolution is closely linked with the **Fourier transform** $\mathcal{F}\left\{\mathbf{x}\right\}$, which transforms a signal into a corresponding *frequency* representation.

The convolution theorem, in particular, states that circular convolution becomes multiplication in the frequency domain:

$$\mathbf{w} \star \mathbf{x} = \mathcal{F}^{-1}\left(\mathcal{F}\left\{\mathbf{w}\right\} \odot \mathcal{F}\left\{\mathbf{x}\right\}\right) \,, \tag{10}$$

where $\mathcal{F}^{-1}$ is the inverse Fourier transform.

Frequency-domain filters are extremely important in signal processing, less so in CNNs, where (10) must be computed for each layer.

Vasilache, N. et al., 2014. **Fast convolutional nets with fbfft: A GPU performance evaluation**. *arXiv preprint arXiv:1412.7580*.

A **circulant matrix** of shape (n, n) for a vector $\mathbf{w}$ is defined as:

$$[\mathbf{C}_{\mathbf{w}}]_{i,j} = w_{i-j \bmod n} \quad (k)$$

For example, for $k = 3$ and $n = 5$:

$$\mathbf{C}_{\mathbf{w}} = \begin{bmatrix} w_0 & w_1 & w_2 & 0 & 0 \\ 0 & w_0 & w_1 & w_2 & 0 \\ 0 & 0 & w_0 & w_1 & w_2 \\ w_2 & 0 & 0 & w_0 & w_1 \\ w_1 & w_2 & 0 & 0 & w_0 \end{bmatrix}$$

Shift matrices

Consider a vector $\mathbf{w}$ with all 0 except a 1 in position k. Then, $\mathbf{C}_{\mathbf{w}} = \mathbf{S}_k$ is called a **shift matrix**, and $\mathbf{S}_k\mathbf{x}$ translates the signal by k positions:

$$[\mathbf{S}_k\mathbf{x}]_i = [\mathbf{x}]_{i+k \bmod n} \,. \tag{11}$$

As a special case, translation by 0 positions returns the original vector:

$$\mathbf{S}_0\mathbf{x} = \mathbf{I}\mathbf{x} = \mathbf{x} \,. \tag{12}$$

Circulant matrices and shifts

Convolution can be expressed as a matrix product with a circulant matrix:

$$\mathrm{x} \star \mathrm{w} = \mathrm{C}_{\mathrm{w}}\mathrm{x}\,. \tag{13}$$

We can also show that any two circulant matrices commute:

$$\mathrm{C}_{\mathrm{w}_1}\mathrm{C}_{\mathrm{w}_2} = \mathrm{C}_{\mathrm{w}_2}\mathrm{C}_{\mathrm{w}_1}\,. \tag{14}$$

Putting the two results together:

$$(\mathrm{S}_k\mathrm{x}) \star \mathrm{w} = \mathrm{C}_{\mathrm{w}}\mathrm{S}_k\mathrm{x} = \mathrm{S}_k\mathrm{C}_{\mathrm{w}}\mathrm{x} = \mathrm{S}_k(\mathrm{x} \star \mathrm{w})\,.$$

Tying everything together

If we translate the signal $\mathbf{x}$, then convolve by $\mathbf{w}$, the result is equivalent to shifting the convoluted signal $\mathbf{x} \star \mathbf{w}$. Formally, we say that convolution is **equivariant** to translations.

In fact, convolution is *the most general* equivariant linear operation of this type, which provides some justification for its use.

In geometric deep learning, we obtain convolutional operators for other domains (e.g., manifolds) by *defining* them to be equivariant to certain properties of interest (e.g., permutations over graphs).

Further readings

- **D2L**: Chapter 7; **UDL**: Chapter 10.
- There are many tools to help you visualize convolutional operations, e.g., `https://ezyang.github.io/convolution-visualizer/index.html`.
- For the geometric deep learning part, this series of blog posts and the accompanying text book: `https://towardsdatascience.com/deriving-convolution-from-first-principles-4ff124888028`.