Deep Learning

Marc Lelarge, Jill-Jenn Vie, and Kevin Scaman

Class notes by Antoine Groudiev

Last modified 21st June 2024

Contents

Abstract

This document is Antoine Groudiev's class notes while following the class *Deep Learning* at the Computer Science Department of ENS Ulm. It is freely inspired by the lectures of Marc Lelarge, Jill-Jênn Vie, and Kevin Scaman.

5 Convolutional Neural Networks

5.1 Introduction

Convolutional Neural Networks (CNNs) is a class of models widely used in computer vision. While Fully Connected Neural Networks are very powerful machine learning models, they do not respect the 2D spatial structure of the input images. For instance, training a Multilayer Perceptron on a dataset of 32×32 images required the model to start with a `Flatten` layer, that reshaped matrix images of size `(32, 32)` to flattened vectors of size `(1024, 1)`. Similarly, different color channels were handled separately, reshaping tensor images of dimensions `(32, 32, 3)` to `(3072, 1)`.

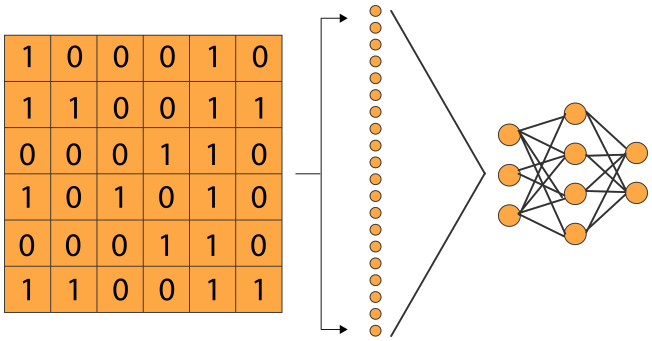

Figure 5.1: Flatten layer breaking the spatial strucutre of input data

CNNs introduce new operators taking advantage of the spatial structure of the input data, while remaining compatible with automatic differentiation. While MLPs build the basic blocks of Deep Neural Networks using Fully-Connected Layers and Activation Layers, this chapter will introduce three new types of layers: *Convolution Layers*, *Pooling Layers*, and *Normalization*.

5.2 Convolution Layers

Similarly to Fully-Connected Layers, *Convolution Layers* have learnable weights, but also have the particularity to respect the spatial information.

5.2.1 Input shape

A Fully-Connected layer receives some flattened vector and outputs another vector:

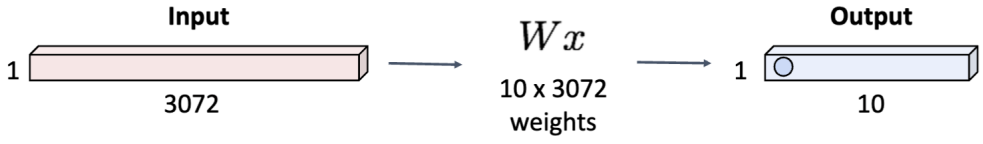

Figure 5.2: Fully-Connected Layer

Instead, a CNN takes as an input a 3D volume: for instance, an image can be represented as a tensor of shape $3 \times 32 \times 32$, the first dimension being the number of channels (red, green, blue), and the other two being the width and height of the image.

5.2.2 Kernels

The convolutional layer itself consists of small kernels (also called filters) used to *convolve* with the image, that is sliding over it spatially, and computing the dot products at each possible location.

Definition (Kernel)**.** A *kernel* (or *filter*) is a tensor of dimensions $D \times K \times K$, where D is the number of channels (or "depth") of the input, and K is a parameter called *kernel size*.

Definition (Convolution of two matrices)**.** Given two matrices $A = (a_{i,j})_{i,j}$ and $B = (b_{i,j})_{i,j}$ in $\mathscr{M}_{m,n}(\mathbb{R})$, the *convolution of A and B*, noted $A * B \in \mathbb{R}$, is the following:

$$A * B = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{(m-i+1),(n-j+1)} \cdot b_{i,j} \tag{5.2.1}$$

This corresponds to the dot product in the space $\mathscr{M}_{m,n}(\mathbb{R})$.

Definition (Kernel convolution)**.** An input of shape $C \times H \times W$ can be processed by a kernel of shape $C \times K \times K$ by computing at each possible spatial position the convolution between the kernel and the submatrix of the input.

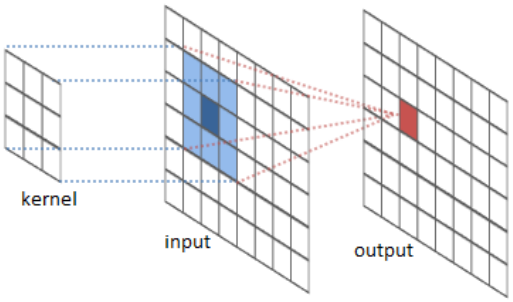

Figure 5.3: Kernel convolution

The output of this operation in an *activation map* of dimension $1 \times (H - K + 1) \times (W - K + 1)$ representing for each pixel the convolution between the kernel and the corresponding chunk of the image.

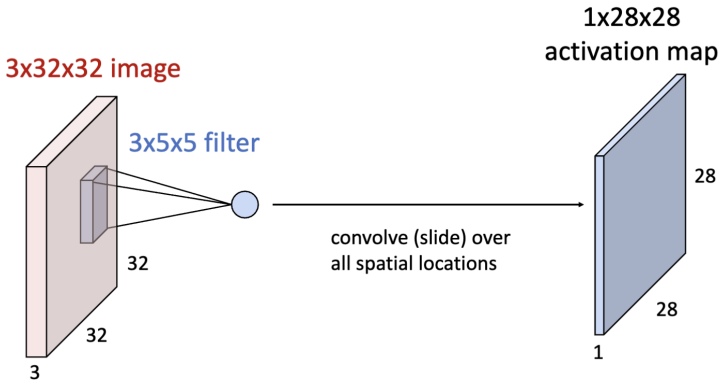

Figure 5.4: Input and output of the convolution operation

Intuitively, the result of the kernel convolution tells us for each pixel *how much the neighbourhood of the input pixel corresponds to the kernel.*

Example (Gaussian blur)**.** Let $G \in \mathscr{M}_3(\mathbb{R})$ be the following kernel:

$$G := \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 1 \\ 1 & 2 & 1 \end{bmatrix}$$

Each coefficient of this matrix is an approximation of the Gaussian distribution. Applying this kernel to an image produces a smoothed version of the input.

Example (Sobel operator)**.** Let S_x and $S_y \in \mathscr{M}_3(\mathbb{R})$ be the following kernels:

$$S_x := \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \quad \text{and} \quad S_y := S_x^\top = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

The convolution between these operators and an image produces horizontal and vertical derivatives approximations of the image pixels.

Input image | Sobel operator applied to the image

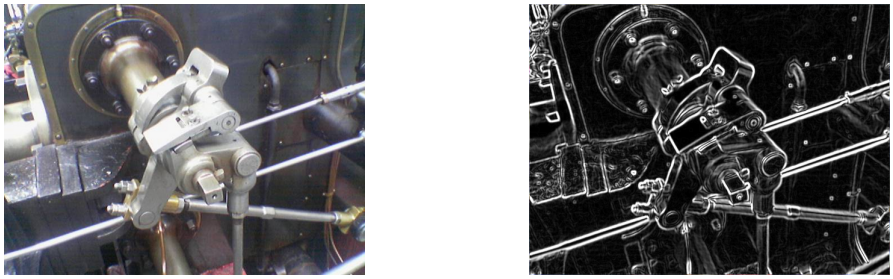

Figure 5.5: Effect of the Sobel operator on an image

These two examples show that kernels used in convolutional layers express meaningful transformations of the input, justifying their use in CNNs. For instance, one could hardcode different kernels (gaussian blur, Sobel operator, vertical/horizontal lines extraction) to extract interesting features from an image, and plug these features into an MLP to obtain an improved classifier compared to a basic, flattening MLP. We will see that instead, CNNs have learnable kernel weights, allowing the model to choose the kernels that it considers bests.

5.2.3 Multiple kernels

In Figure 5.4, we used simply one kernel to compute one activation map. In practice, we repeat this process multiple times: we consider a set (or *bank*) of filters having different weights values, and for each kernel of the set, we compute its activation map.

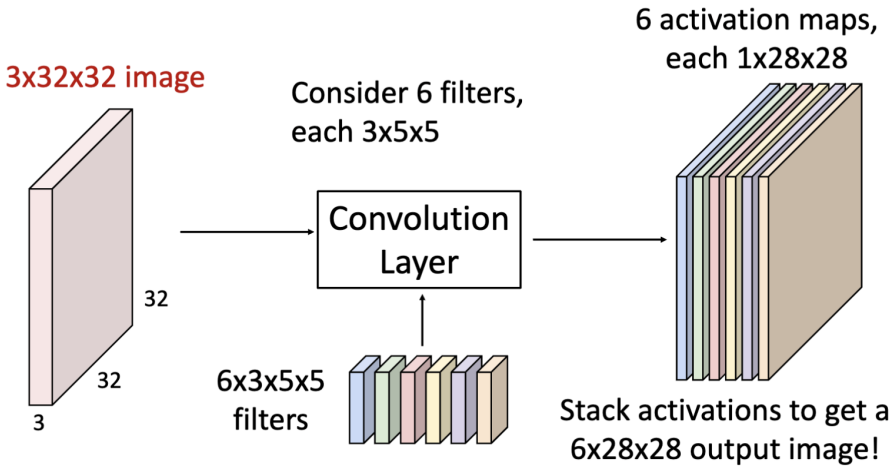

Figure 5.6: Convolutional Layer using 6 kernels

Using a bank containing C' filters, the output of the convolutional layer in an *activation map* of dimension $C' \times (H - K + 1) \times (W - K + 1)$ representing for each pixel the convolution between the given kernel and the corresponding chunk of the image.

Remark (Biases in Convolutional Layers)**.** *Similarly to fully-connected layers, we often add to the activation map of each kernel a bias of size* $1 \times (H - K + 1) \times (W - K + 1)$*. Those biases might me ommited in the rest of the chapter for the sake of simplicity.*

5.2.4 Stacking convolutions

Like previously introduced layers, convolutional layers can be stacked to form deep networks. The layer shapes need to match, in particular the output channels of a layer must match the input channels of the next layer, and the output height and width must match the next input height and width.

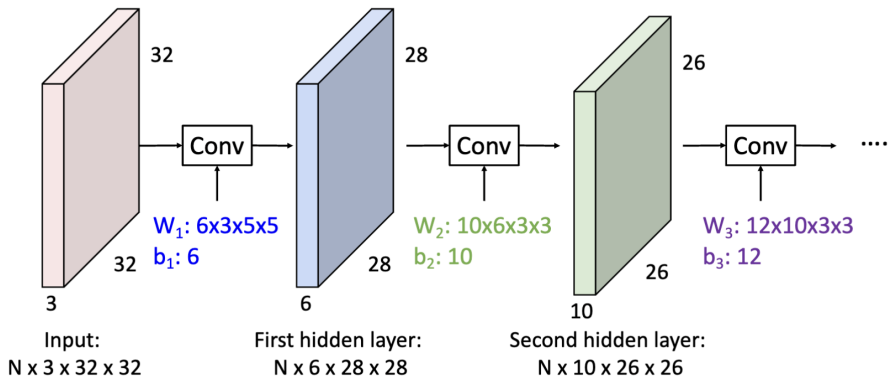

Figure 5.7: Stacking of 3 Convolutional Layers of correct shapes

However, stacking two convolution layers next to each other produces another convolutional layers, and do not add representation power. Therefore, we use the exact same solution as for linear classifiers: we introduce non-linear layers using activation functions in between convolutional layers.

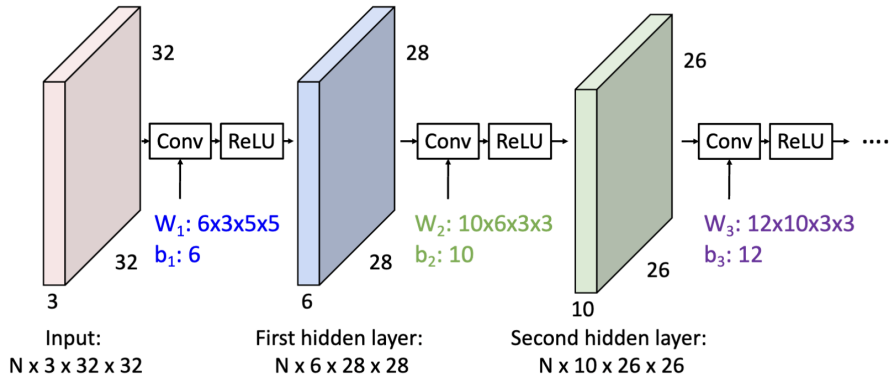

Figure 5.8: Adding ReLU layers in between Convolution Layers

5.2.5 Spatial dimensions and Padding

As stated previously, using an input of width W with a filter of kernel size K, the output width is $W - K + 1$. A problem with the approach is that features maps decrease in size with each layer. This creates an upper bound on the maximum number of layers that we can use for our model.

A solution to this is to introduce *padding* by adding zeros around the border of the input. When the kernel will slide around the edges of the input, a part of the coefficients that it will consider in its convolution will be zeros.

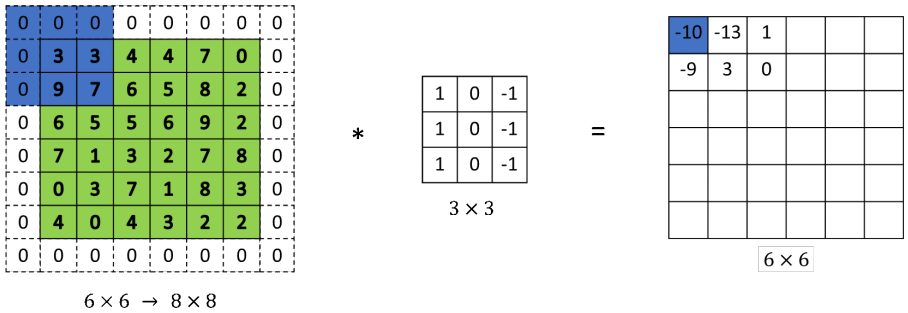

Figure 5.9: Adding padding around the input

Remark (Padding strategies)**.** *Even though we might imagine different padding strategies instead of always padding with zeros (for instance, nearest-neightbour padding, circular padding, random padding...), zero-padding seems to be both simple and effective in practice, and is the most commonly used strategy.*

Padding introduces an additional hyperparameter to the layer, P. Using padding, the width of the output of the layer becomes:

$$W' = W - K + 2P + 1 \tag{5.2.2}$$

A common way to set the value of P is to choose it such as the output have the same size as the input. This is achieved by taking $P = (K - 1)/2$, called *same-padding*.

5.2.6 Receptive Fields

Definition (Receptive Field)**.** The *receptive field* of an output neuron is the set of neurons of the input of which the output neuron depends on.

By essence, Fully-connected layers have a trivial notion of receptive field: an output neuron is connected to each input neuron, its receptive field is therefore the entire input.

Convolution layers are build in such a way that each element in the output simply depends on a receptive field of size K (that is a square of area $K \times K$) in the input. As we stack convolutional layers after the others, each successive convolution adds $K - 1$ to the receptive field size. After L layers, the receptive field size is $1 + L \times (K - 1)$. This linear growth shows that by stacking enough layers, each output neuron will eventually have the entire input image in its receptive field. Nevertheless, this can be a problem in practice as we might need many layers for each output to depend on the whole image.

A solution to this problem is to downsample the image size inside the network. This can be done by adding another hyperparameter, *stride*.

5.2.7 Strided Convolution

Definition (Stride)**.** The hyperparameter *stride* defines the number of pixels between two applications of the kernel.

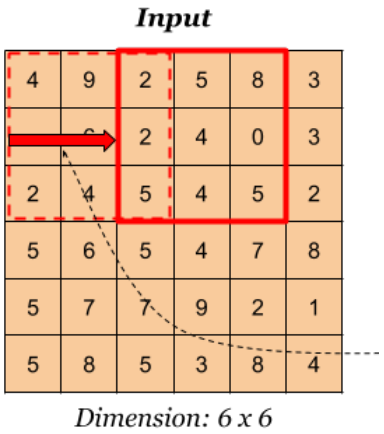

Figure 5.10: Effect of stride

Stride effectively downsamples the size of the image. Applying a convolution between an image of width W and padding P with a kernel of size K and stride S produces the following output dimension:

$$W' = \frac{W - K + 2P}{S} + 1 \tag{5.2.3}$$

Note that choosing $S = 1$ in (5.2.3) gives the same result as (5.2.2). Depending on the implementation, the result can be rounded up or down in the case where it is not an integer. Usually, all the parameters are chosen such that S divides $W - K + 2P$.