Deep Learning

Marc Lelarge, Jill-Jenn Vie, and Kevin Scaman

Class notes by Antoine Groudiev

Last modified 24th 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.

1 Introduction and general overview

1.1 What is Deep Learning?

1.1.1 Neural networks

1.1.2 Timeline of Deep Learning

1.1.3 Recent applications and breakthroughs

1.1.4 Usual setup

1.1.5 Required skills

1.1.6 Building blocks of deep learning

1.1.7 Why deep learning now?

1.2 Machine Learning pipeline

1.2.1 Cats vs. dogs

1.2.2 Typical Machine Learning setup

1.2.3 Training objective

1.3 Multi-Layer Perceptron

1.3.1 Definition

1.3.2 PyTorch implementation

2 Automatic Differentiation

In the following, we will consider a "set" of data points

$$X \in \mathbb{R}^{N \times d}$$

made of N inputs of size d, and targets

$$Y \in \mathcal{Y}^n$$

where $\mathcal{Y}$ is an arbitrary set. It can be for instance $\mathcal{Y} = \mathbb{R}$ is the case of regression, a finite set such as $[\![1, C]\!]$ in the case of classification, or $\mathcal{Y} = \mathbb{R}^{d'}$ in a more general setup.

2.1 Introduction

As stated previously, neural networks is a very expressive class of functions. However, the associated optimization problem is in general non-convex, giving very few theoretical guarantees and no closed-form expression. In practice, this is not an issue, since such optimization problem can be solved using *gradient descent*.

2.1.1 Loss function

Gradient descent is done by minimizing the average of a differentiable loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. For instance, for regression, we might choose the squared error:

$$\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$$

For classification, we might choose the logistic loss. Its expression for a two-classes model (that is $y \in \{0, 1\}$) is:

$$\mathcal{L}(\hat{y}, y) = y \log \hat{y} + (1 - y) \log(1 - \hat{y})$$

More generally, for a C-classes model (that is $y \in [\![1, C]\!]$), the cross-entropy loss is:

$$\mathcal{L}(\hat{y}, y) = \sum_{c=1}^{C} y_c \log \hat{y}_c$$

The average of the loss function is then given by:

$$J(f) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}\left(f(X_n), Y_n\right)$$

which we will try to minimize.

2.1.2 Gradient descent

The idea behind gradient descent is therefore to be able to compute the gradient of $\mathcal{L}$ with respect to the paramters θ for each point of the dataset:

```
for epoch in range(EPOCHS):
    for x, y in zip(X, Y):
        compute ∇_θ L
        θ = θ − γ ∇_θ L
```

Figure 2.1: Pseudo-code of gradient descent

The only remaining challenge is the computation of $\nabla_\theta \mathcal{L}$, preferably automatically; this is the problem which we will address in this chapter.

2.2 Optimization methods

2.2.1 Stochastic Gradient Descent

2.2.2 Batch gradient descent

2.2.3 Minibatch gradient descent

2.2.4 Newton's method

2.3 Computing gradients

2.3.1 By hand

2.3.2 Numerical differentiation

2.3.3 Symbolic differentiation

2.3.4 Automatic differentiation

3 Introduction to Reinforcement Learning

4 Optimization and loss functions

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)`.

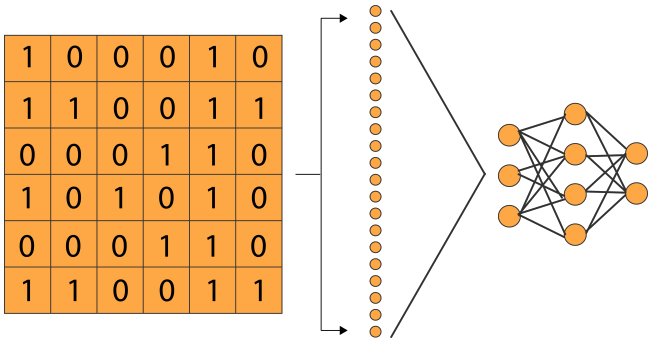

Figure 5.1: Flatten layer breaking the spatial strucutre of input data

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:

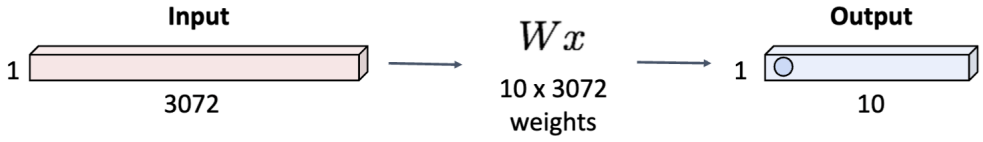

Figure 5.2: Fully-Connected Layer

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.

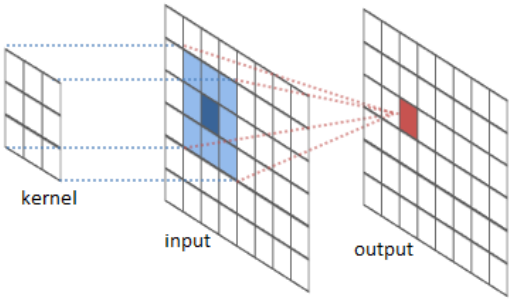

Figure 5.3: Kernel convolution

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.

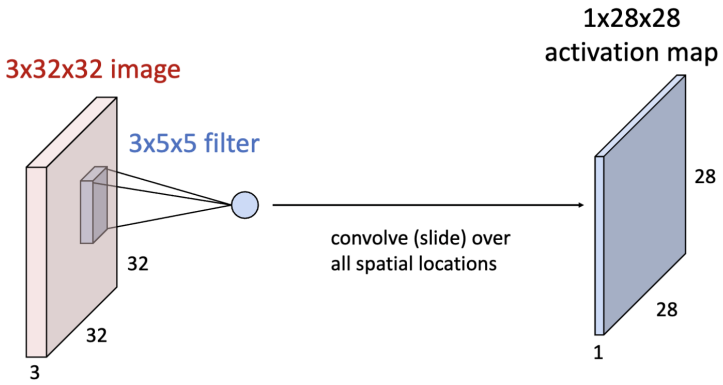

Figure 5.4: Input and output of the convolution operation

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

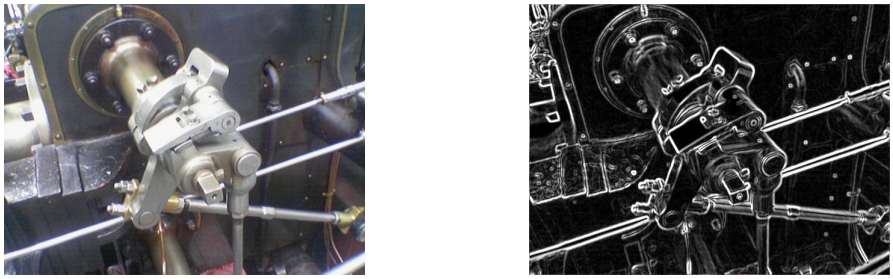

Figure 5.5: Effect of the Sobel operator on an image

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.

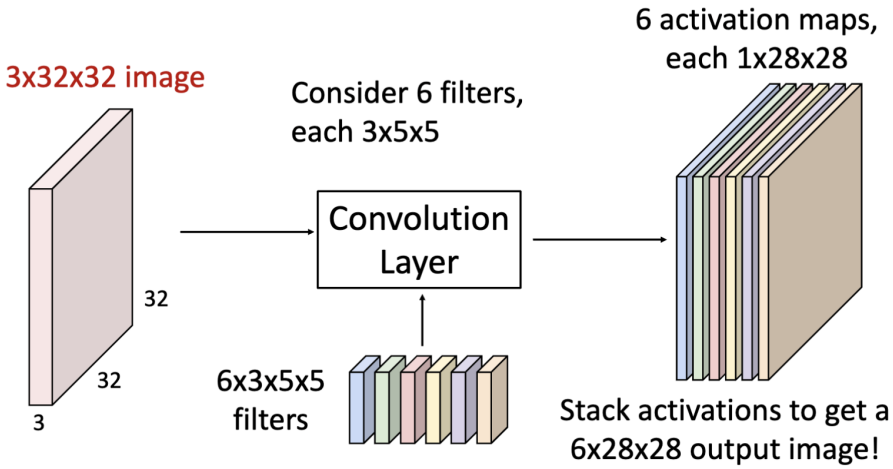

Figure 5.6: Convolutional Layer using 6 kernels

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.

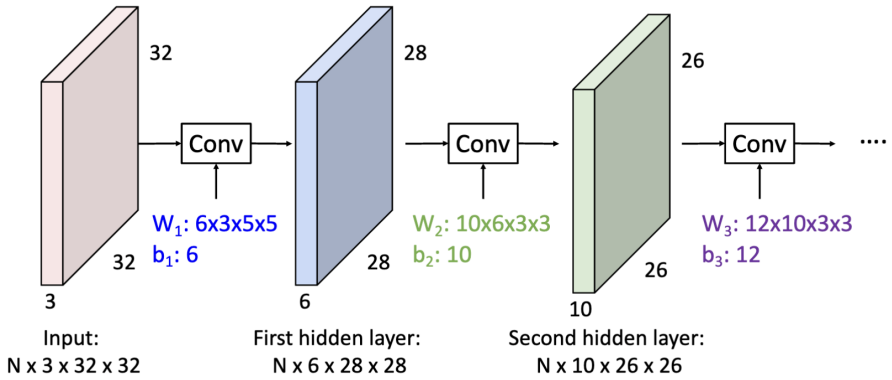

Figure 5.7: Stacking of 3 Convolutional Layers of correct shapes

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.

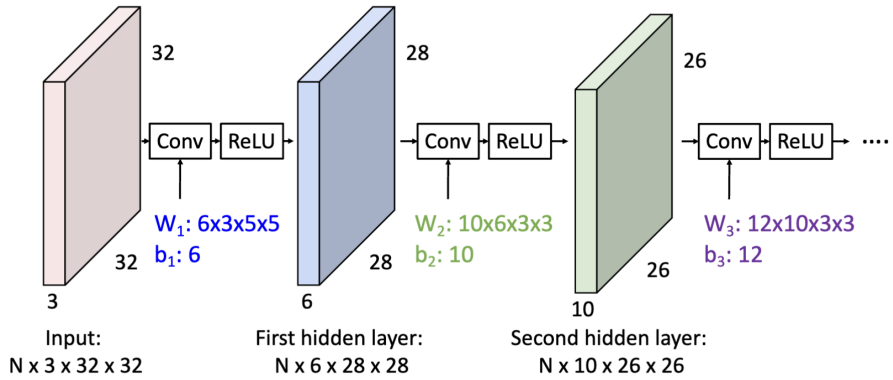

Figure 5.8: Adding ReLU layers in between Convolution Layers

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.

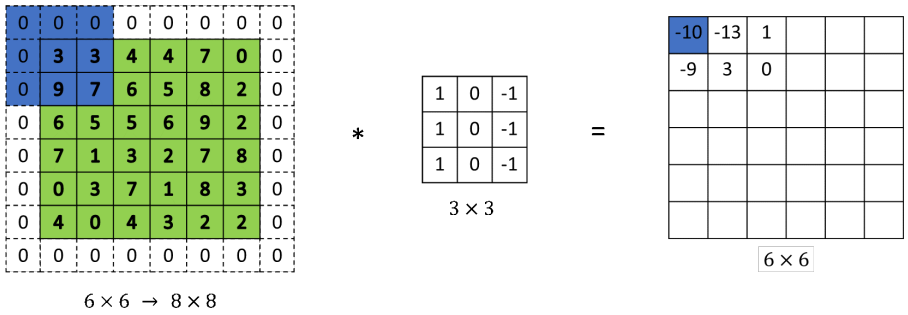

Figure 5.9: Adding padding around the input

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 $1 + L \times (K-1)$.

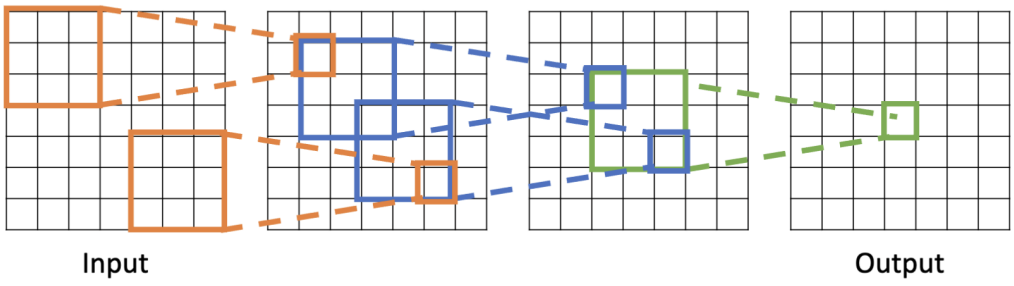

Figure 5.10: Receptive field of an output neuron

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.

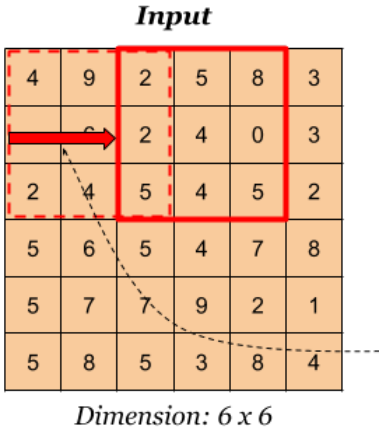

Figure 5.11: Effect of stride

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 $W - K + 2P$.

5.3 Pooling Layers

5.3.1 Introduction

Computer Vision, one of the most frequent use for CNNs, often deals with images of high quality, making downsampling an important task to drastically reduce the number of layers and the quantity of VRAM used by the model. We saw a first approach to downsampling embedded in Convolutional Layers, that is strided convolution. Pooling Layers are layers dedicated to downsampling, without learnable parameters.

Pooling layers work similarly to convolutional layers, using a mechanism of kernels. Nevertheless, instead of applying a convolution between some kernel and the image, the layer will apply a pooling function to the area of the input. This will produce an activation map with dimensions depending on the hyperparameters of the layer – kernel size, padding and stride, the same as the parameters of a convolutional layer.

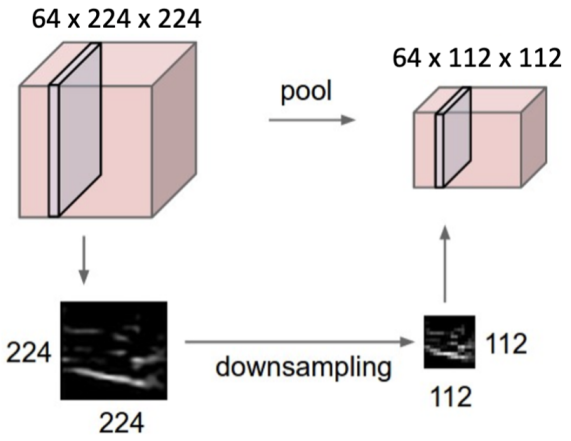

Figure 5.12: Behavior of a pooling layer

Another difference is that the operation is applied for each slice of the input volume. Choosing appropriate values for the hyperparameters (for instance $S \geqslant 2, \dots$) allow to downsample the input without the need for learnable parameters.

5.3.2 Max Pooling

Definition (Max Pooling function)**.** For $K \geqslant 1$, the *Max Pooling function* of kernel size K is:

$$\begin{aligned} \max_P : \mathscr{M}_K(\mathbb{R}) &\longrightarrow \mathbb{R} \\ (m_{i,j}) &\longmapsto \max_{i,j} m_{i,j} \end{aligned}$$

A *Max Pooling* layer applies the Max Pooling function to each location of size $K \times K$ in each slice of the input volume. We often choose the same kernel size as the slide (that is $K = S$) to avoid recovering twice the same pixel value. In this setting, it is equivalent to spliting each input channel into non-overlapping regions of size $K \times K$, from which are extracted the maximum value of the section and stored in the output channel.

Max Pooling has some advantages over convolutional layers with stride: it does not involve learnable parameters, reducing the computational cost, but also introduces translational invariance to small spatial shifts. Indeed, if the position of a specific maximum pixel is moved slightly, we might intuitively think that it will stay the maximum of its region, making the model robust to small translations.

5.3.3 Average Pooling

Definition (Average Pooling function)**.** For $K \geqslant 1$, the *Average Pooling function* of kernel size K is:

$$\begin{aligned} \operatorname{avg}_P : \mathscr{M}_K(\mathbb{R}) &\longrightarrow \mathbb{R} \\ (m_{i,j}) &\longmapsto \frac{1}{K^2} \sum_{i=1}^{K} \sum_{j=1}^{K} m_{i,j} \end{aligned}$$

It simply returns the average of the matrix coefficients.

Average Pooling works exactly the same as Max Pooling, but applying avg_P as a pooling function instead of $\max_P$.

5.4 A full CNN example: LeNet-5

Now that we have these different types of layers, we can stack them together to create a full CNN architecture. A classic model often fits the following architecture:

$$(\texttt{Conv, ReLU, Pooling})^{N_1} \rightarrow \texttt{Flatten} \rightarrow (\texttt{Linear, ReLU})^{N_2} \rightarrow \texttt{Linear}$$

As an example, we will take the 1998 model *LeNet-5*:

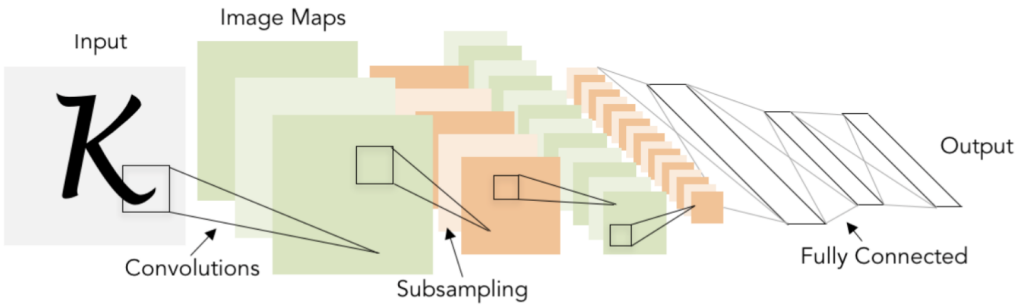

Figure 5.13: LeNet-5 architecture

The first blocks of Convolutional and Pooling layers progressively decrease the spatial size of the input, while increasing the number of channels: the total volume of the input is preserved.

6 Recursive Neural Networks

7 Attention and Transformers

8 Robustness and regularity

9 Q-Deep Learning for Breakout

10 Autoencoders

11 Generative Adversarial Networks

12 Normalizing Flows