Deep Learning

Leonardo Biason

October 21, 2024

Chapter 1

Convolutional Neural Networks

When dealing with **Multilayer Perceptrons** (**MLP**s), we mostly used data that didn't have articulate structures. Even in the example of the MNIST digits dataset, we still consider the input image as a stream of flattened vectors. This approach, even though helps for making small examples, would not hold well with actual images, which are more complex in terms of number of channels, possible values of each pixel, and so on...

So how can we deal with that? How can we use images with neural networks, so that to still keep track of relevant informations that would be otherwise lost? The solution is given by the architectural model of the **Convolutional Neural Network**, CNNs for short. In this first chapter, we'll see how CNNs are made, and what components they have.

1.1 Image Filters

Many times in the fotography field we hear the term **filter**, but what *is* a filter to begin with? Why do we use it? What kind of filters can we apply? Let's first give a proper definition:

DEFINITION

Filter

A **filter** is the **application** of a **specific function** to a **local image patch** of a given dimension

Consider the following example: we have a patch of an image (suppose that the patch's dimensions are smaller than the image ones) and we apply a function which returns the mean of all the pixels adjacent to a selected pixel:

8	7	4
5	9	1
2	3	6

$\xrightarrow{f(x)}$

	5	

Image filtering is a technique that is widely used for various reasons: to **reduce noise**, to **fill in missing values** and even to **extract image features**, such as edges and/or corners. The simplest type of filter that we can have is a filter that replaces each pixel with a linear combination of its neighbours. We call this a **linear filter**. One of the most known linear filters is the **2D convolution**.

DEFINITION

Convolution

A **convolution** is a **linear filter** which slides a given **filter kernel** through the image and performs the **matrix multiplication** between the filter and the overlapped image patch, returning a filtered image.

A filtered image f is expressed as follows:

$$f[m, n] = I \otimes g = \sum_{k,l} I[m-k, n-l] \cdot g[k, l]$$

where I is the image, g is the kernel and m, n, k and l are indexes.

In the case where in the formula there would've been + instead of − (so within $I[m-k, n-l]$), then we would've called that operation a **correlation**. Let's make a quick example to show how convolutions work:

EXAMPLE

1.1.0

Suppose that we have the following image I and kernel g:

8	5	2
7	5	3
9	4	1

$I[k, l]$

−1	0	1
−1	0	1
−1	0	1

$g[k, l]$

How can we perform the convolution of I with the kernel g? Suppose that we want to perform the convolution at the center of the image. When using k and l, it's important to note that the coordinates work in the following way:

- the center of the kernel has coordinates [0, 0];
- if from a coordinate $[k, l]$ we move to the right, then we arrive at $[k, l-1]$, and viceversa if we go to the left we arrive to $[k, l+1]$;
- if from a coordinate $[k, l]$ we move upwards, then we arrive at $[k+1, l]$, and viceversa if we go downwards we arrive to $[k-1, l]$.

The following schema sums up this coordinate system:

[1, 1]	[1, 0]	[0, −1]
[0, 1]	[0, 0]	[0, −1]
[−1, 1]	[−1, 0]	[−1, −1]

Now, for $k = -1$ and $l = -1$, we would have that:

$$I[m+1, n+1] \cdot g[-1, -1] = 1 \cdot -1 = -1$$

For $k = -1$ and $l = 0$ we would have instead:

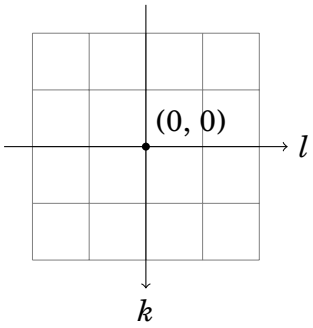
$$I[m+1, n+0] \cdot g[-1, 0] = 4 \cdot 0 = 0$$

Then, for $k = -1$ and $l = 1$ we would have:

$$I[m+1, n-1] \cdot g[-1, 1] = 9 \cdot 1 = 9$$

And so on and so forth for all the multiplications...

From this previous example we had a way to illustrate how the convolution works, but what if we had to code it? If we had to keep track of two different indexes, we would waste some computational memory. Let's try to find a quicker and more efficient method. We can start from the kernel: the multiplication $I \otimes g$ is made between items that are in mirrored positions with respect to some "invisible axes" l and k:



A simple way to make the computations easier is to flip the kernel along these axes, so that to align it to the image's axes. This way, we would just have to do the element-wise multiplication of the matrices and sum the resulting values.

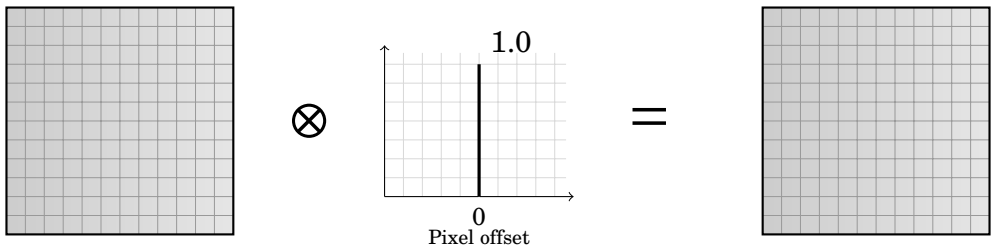
A special case of convolution is when we use a 1D filter on a 2D image, but we'll see more about this later on.

1.1.1 Linear Filters

There are different types of filters, depending on the values stored within g and on its shape. For instance, assume that we have a filter whose shape is 1×9, and that it's equal to the following:

$$g = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

What would it happen if we multiplied $I \otimes g$? For each pixel, the filter would return only the central pixel, so the one in the coordinates (0, 0) of the filter when it passes through the image. So, at the end, nothing would change

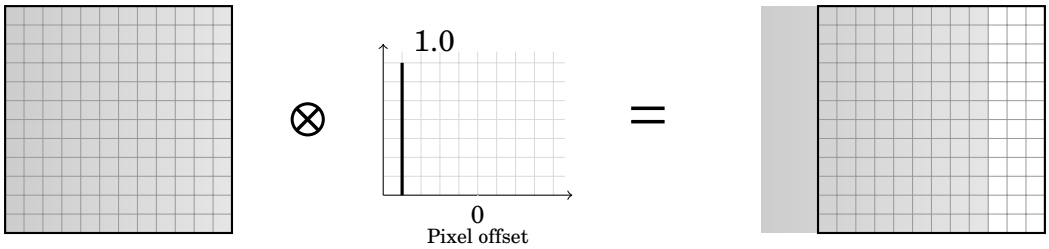


I g f

What if we used instead the following filter g?

$$g = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Now, for each pixel on which the filter slides, we would take a pixel that is 3 pixels at the right and place it at the local coordinates (0, 0). If we repeat that for the entire image, we will then obtain an image shifted by 3 pixels:

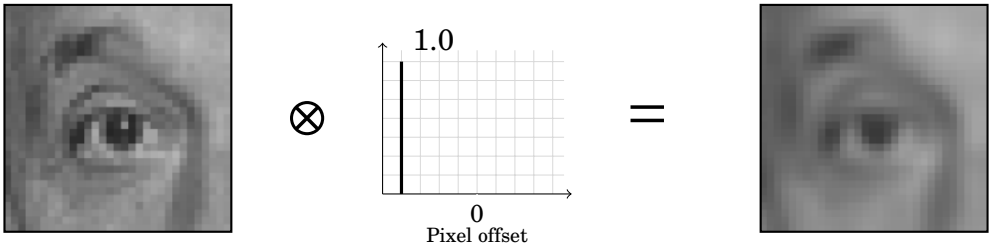


I g f

A very interesting filter is the following:

$$g = \begin{bmatrix} 0 & 0 & 0 & 0{,}33 & 0{,}33 & 0{,}33 & 0 & 0 & 0 \end{bmatrix}$$

This filter combines a pixel with its neighbours, taking a third of each pixel's values and then summing them up together. This kind of filter is called **blur**.



I g f

When dealing with images, it's important to also recognize some details, such as an edge or some variations in the color. But how do we define them?

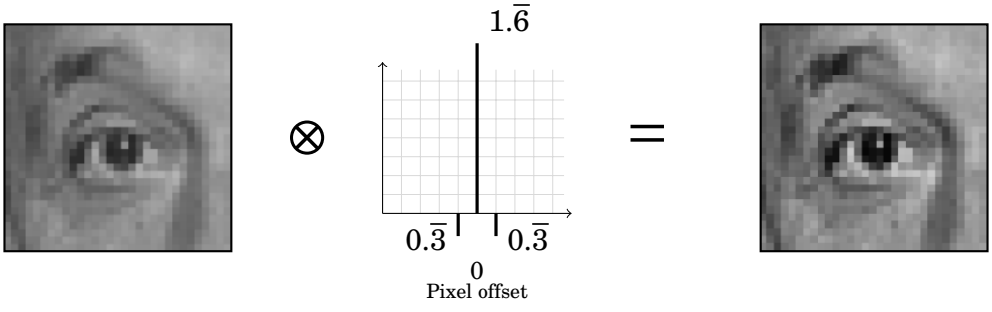
DEFINITION

Edge

An **edge** of an image is when there is an **abrupt change** of the values from one pixel to an adjacent one

Also, it's possible that we can sometimes deal with **impulses**, so images where only one pixel stores non-zero values.

Generally, it's important to recognize edges in the images because it helps on detecting items. But not in all images the edges are clear to recognize, and sometimes we may need to accentuate, to **sharp** them. This can be done through a **sharpening filter**, which usually has the following form:



I g f

We can notice how the differences of colors are accentuated, and how the costant areas are instead left similar to the original image.

Since these filters are all linear, they also can use the basic properties of the linear systems, which are the following three:

- **Homogeneity**: $T[a \cdot X] = a \cdot T[X]$;
- **Additivity**: $T[X_1 + X_2] = T[X_1] + T[X_2]$;
- **Superposition**: $T[a \cdot X_1 + b \cdot X_2] = a \cdot T[X_1] + b \cdot T[X_2]$.

In general, a system is said to be **linear** if and only if the **superposition** property **holds**. All the filters presented until now are linear filters (convolutions too), so they can take advantage of these properties.

1.1.2 Average and Gaussian Filters

1.1.3 Edges and Derivatives

Chapter 2

CNNs Architectures

2.1 AlexNet

do alexnet

2.2 VGG

do vgg

Chapter 3

Training Neural Networks

Until now, we saw various CNN architectures and the different layers that we can have in a model. But these models must also be trained, in order to obtain a working model. Here, we'll see various methods for training models, and the different tools that get used during training.

3.1 Activation Functions

Nowadays, there is a set of activation functions that are considered to be the avant-guarde of deep learning, but they might change in a few years. Although different, new versions and types of activation functions might come out, the reasoning between them remains mostly the same. But let's define first what an activation function is:

DEFINITION

Activation Function

An **activation function** is a function which **determines** whether a given neuron should **activate** or not by computing the weighted sum w and adding the bias b.

Until now there are a lot of known and used activation functions: some of them are the **Sigmoid** function, the **ReLU**, etc... For now, we'll concentrate on the **sigmoid** activation function.

DEFINITION

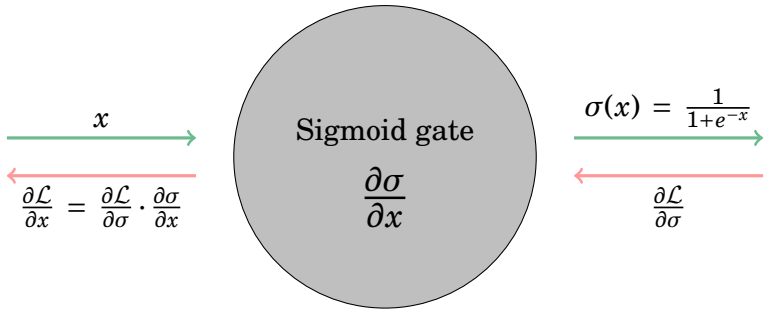
Sigmoid Activation Function

The **sigmoid activation function** is equal to

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

The output of the sigmoid function is in the range [0, 1].

The sigmoid function was very popular back in the days, mainly because it represented very well when a neuron should "fire" or not. However, it has 3 main problems, which led to the abandonment of this function. The first problem is that **saturated neurons kill** the **gradient**. For instance, suppose that we have the following neuron:



More specifically, we define the gradient of the sigmoid functions as follows:

$$\frac{\partial \sigma}{\partial x} = \sigma(x) \cdot (1 - \sigma(x))$$

Let us try to compute the value of the sigmoid gradient for 3 different values: for 10, 0 and -10. Let's begin with $x = 10$:

$$\sigma(10) = \sim 0 \qquad \frac{\partial \sigma}{\partial x} = \sigma(10) \cdot (1 - \sigma(10)) = 0 \cdot (1 - 0) = 0$$

Now, let's try with $x = 0$:

$$\sigma(0) = 0{,}5 \qquad \frac{\partial \sigma}{\partial x} = \sigma(0) \cdot (1 - \sigma(0)) = 0{,}5 \cdot (1 - 0{,}5) = 0.25$$

Finally, let's try with $x = -10$:

$$\sigma(10) = \sim 1 \qquad \frac{\partial \sigma}{\partial x} = \sigma(10) \cdot (1 - \sigma(10)) = 1 \cdot (1 - 1) = 0$$

We can see how the result of the gradient is mostly 0, and this clearly results in a problem because when the value of the gradient fill flow down into the network, then the parameters will never update.

The second problem with the sigmoid function is that the outputs of the sigmoid are **not centered in zero**.

3.2 Data Preprocessing

Generally the data can be