Deep Learning

Leonardo Biason

November 28, 2024

Contents

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

DEFINITION

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:

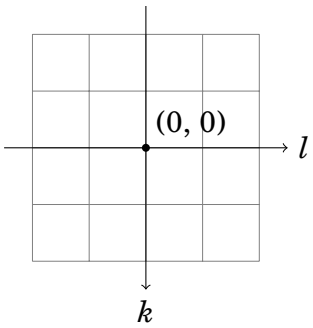
$$I[m+1, n+0] \cdot g[-1, 0] = 4 \cdot 0 = 0$$

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.

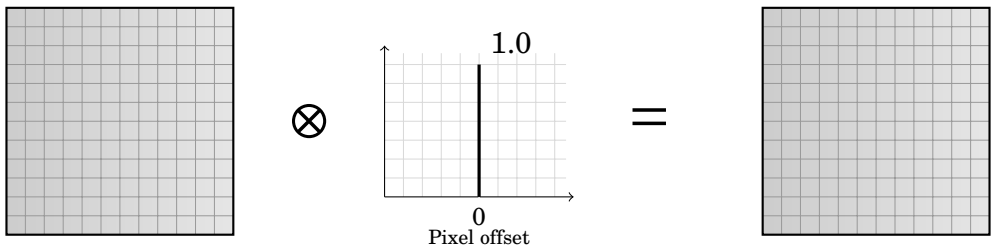
A special case of convolution is when we use a 1D filter on a 2D image, but we'll see more about this later on.

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

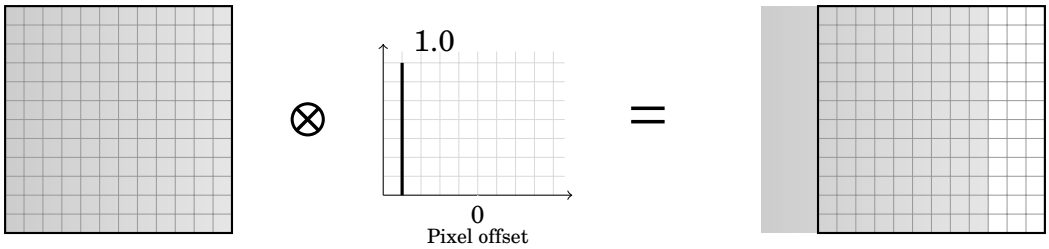


I g f

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:

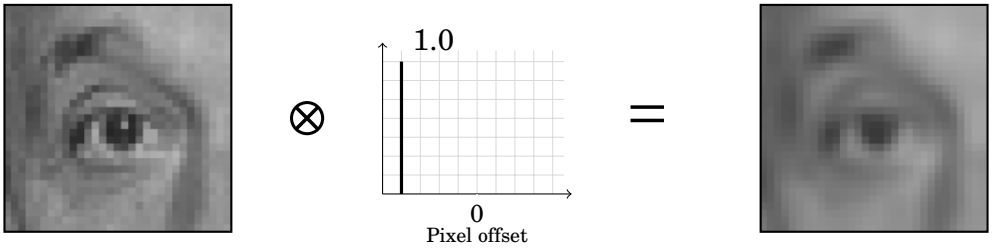


I g f

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?

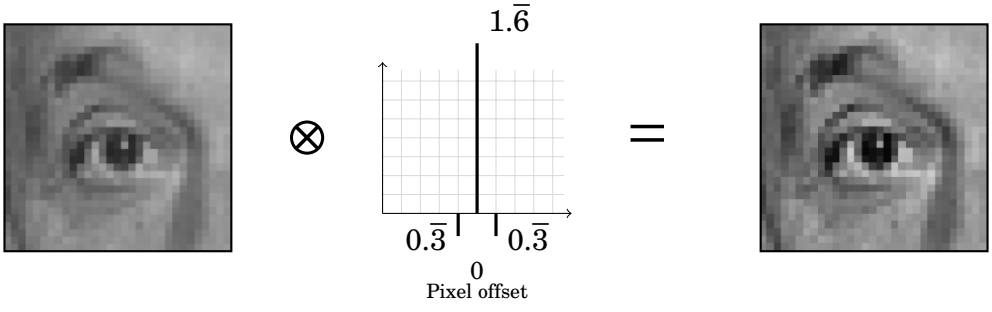
DEFINITION

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.

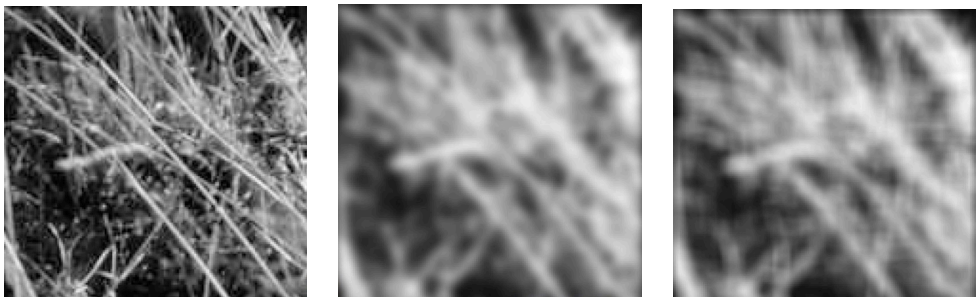
1.1.2 Average and Gaussian Filters

There are other interesting types of filters that we can use with convolutions, such as the **average filter**. The task of this filter is to replace each pixel with an average of its neighbours (so for instance, each neighbour of a pixel will contribute to 1/9 of the total value). The average can also have different **weights**, so that to give more importance to some pixels than to others. If all the weights are equal, then such filter is also called **box filter**.

Another type of blur filter is the **Gaussian Average filter**, which weights the neighbours depending on their distance from the central pixel (the nearer, the greater the weight). The smoothing of the kernel is proportional to the Gaussian function:

$$\underbrace{g}_{\text{kernel}} \propto \underbrace{\frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}}_{\text{Gaussian function}}$$

We can here compare the results of the average and Gaussian filters:



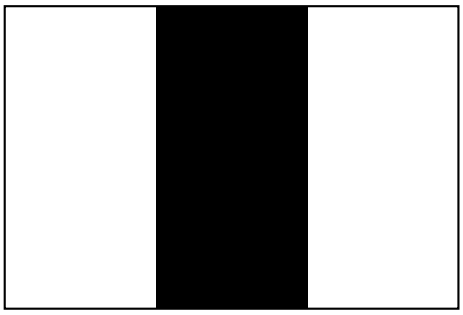
Starting image I **Average Filter** **Gaussian Filter**

The utility of the box filter and the Gaussian filters is that they are **separable**, which means that we can first convolve the rows with a one dimensional filter and then convolve the columns with another one dimensional filter. This is very **efficient** when it comes to computing the filters, because with the full box filter for instance we would have to do $k^2 \cdot n$ operations, but with the separated box filters we would only have to do $2k \cdot n$ operations. This is possible because the **convolution is linear**, so we can use the associative and commutative operations.

$$(f_x \otimes f_y) \otimes I = f_x \otimes (f_y \otimes I)$$

1.1.3 Edges and Derivatives

When we have an image with abrupt changes of values, we say that we have an **edge**. But how can we detect such edges mathematically? Suppose that we have the following image:



$$\begin{bmatrix} 255 & ... & 255 & 0 & ... & 0 & 255 & ... & 255 \\ 255 & ... & 255 & 0 & ... & 0 & 255 & ... & 255 \\ 255 & ... & 255 & 0 & ... & 0 & 255 & ... & 255 \\ 255 & ... & 255 & 0 & ... & 0 & 255 & ... & 255 \\ 255 & ... & 255 & 0 & ... & 0 & 255 & ... & 255 \\ 255 & ... & 255 & 0 & ... & 0 & 255 & ... & 255 \end{bmatrix}$$

We see how we have an abrupt change of values from 255 to 0. We can exploit derivatives to see the behaviour of the function. Indeed, if we compute the first derivative, we obtain the following matrix:

$$\begin{bmatrix} 0 & ... & 0 & 255 & 0 & ... & 0 & -255 & 0 & ... & 0 \\ 0 & ... & 0 & 255 & 0 & ... & 0 & -255 & 0 & ... & 0 \\ 0 & ... & 0 & 255 & 0 & ... & 0 & -255 & 0 & ... & 0 \\ 0 & ... & 0 & 255 & 0 & ... & 0 & -255 & 0 & ... & 0 \\ 0 & ... & 0 & 255 & 0 & ... & 0 & -255 & 0 & ... & 0 \\ 0 & ... & 0 & 255 & 0 & ... & 0 & -255 & 0 & ... & 0 \end{bmatrix}$$

As we can see, the values 255 and -255 denote where we have a change of values in the image. This allows us to detect eventual changes in the image. We can also compute the second derivative, which will allow us to gain further intel about which color is changing to which.

1.2 Classifying Images

Classifying data is a task that in the recent years has been an active field of development. The reason why this is the case is because, unlike with a more standard algorithm such as sorting, there exists **no obvious way** for a computer to do it. While for us humans it's pretty easy to recognize the contents of an image, for a computer it changes drastically.

Machine Learning presents us a more **data-driven approach**, where classification happens through a **trained model**. Usually, for this step we must have a dataset of images and labels, then a model must be trained and finally the model must be evaluated on never-seen-before images.

The idea is then to have what we call a **linear classifier**: starting from an image of size $h \times w \times c$ (where h and w are height and width respectively and c is the number of channels), a function such as

$$f : x, W \longmapsto n$$

where x is the **input image**, W is the **set of parameters** (or **weights**) and n is the set of **class scores**. The size of n is the number of classes that the model can recognize.

But we know that models can suffer from overfitting, which would make the whole model unusable (since it wouldn't be able to generalize anymore), so what can we do? We can add a **bias** b, which would create some noise. Our final function of the linear classifier would then have the following structure:

$$f(x, W) = W \cdot x + b$$

At the end of the classifier, we will obtain a set of scores, one for each classes. Usually, the higher the score, the higher the probability that the image belongs to the respective class.

There is a bunch of problems with this approach though. Let's try to run our model on some images, and see what its generalization looks like it brought to an image:

Plane

Car

Bird

What we can see in the previous images are the generalizations that the model learned for each of the indicated classes. But, as we can see, they do not really resemble anything in particular: indeed, maybe the only recognizable item is the car. But from this we can recover an important point: **generalization happens** only from a **specific point of view**. We can clearly see that the car would be recognized only if the picture was taken at the front, but what if we had a side picture? We can't know for certain, but it most likely won't be labelled as a car (or at least, not with the same precision of a front picture).

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.

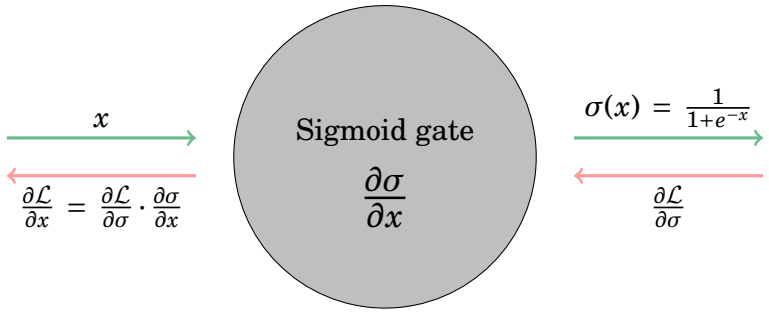
Definition — Sigmoid Activation Function

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

3.3 Weight Initialization

3.4 Batch Normalization

Sometimes, the data, if scattered randomly in the space, can have some patterns that make learning hard for a model. This will make the model not be able to fix a parameter, and would mostly go back and forth between multiple values. If we were able to normalize the data, then we would have a smoother learning.

3.5 Training and Testing Error

Whenever we train a model, we usually keep track of two important values: the value of the **loss function** and the value of the **accuracy**. We already know how to reduce the value of the loss function, so to use better optimization algorithms. For the accuracy instead, that's a different story: we want to maximize it, but we must avoid **overfitting**. Generally, we see that there is overfitting when the gap between training accuracy and validation accuracy starts to grow.

So how do we deal with this? An idea is to use an **early stopping** technique: whenever we see that the gap has been augmenting for too much and it also takes greater and greater values, then we want to stop training. This will return a model that could not be overfitting, and it's a technique that should be **always employed**.

If we had the possibility of training multiple models, we can use the **model ensemble** technique. By training multiple models, we would have for each model different parameters, and at test time we would average the parameters of each model. This would grant, usually a $\sim 2\%$ performance improvement, which may not seem a lot, but it's actually helpful depending on the cases.

3.5.1 Regularization and Dropout

The model ensemble technique works only if we have multiple trainable models, but what can we do to improve the performance of a single model? We can use the **regularization**. Regularization can be achieved in multiple ways:

- with an **extra term** in the **loss function**;
- with the **dropout**;

The second method, called **dropout**, is more advanced than the previous method, so adding an extra term to the loss function. For the dropout method, for each forward pass, the output of a neuron is randomly set to 0. This randomness is dictated by a hyperparameter, which is usually $0,5$. This means that around 50% of the neurons will return nothing.

But why is this a good idea? Because this makes the network develop a redundant representation, and this way it wouldn't rely on groups of neurons to determine the output. For instance, we define a cat as an animal which has a tail, ears and furr. Say that for each of these features we have a neuron assigned. Now, if we deactivate at random the neuron which detects the ears, if the model learned that a cat must have all of these three features together, then in might not recognize the cat. If we deactivate some neurons instead, then the model will try to recognize the cat also without the feature of the ears.

The behaviour of dropout is something that we want to have only during training, not during testing. We want to average out the randomness during test time. So what we can do is the following: if we define with $d(x, z)$ the function which determines whether a neuron is deactivated or not, then by taking it's expectation value, we can remove it from

the testing of the model.

$$y = \mathbb{E}(d(x, z)) = \int p(z)d(x, z)\mathrm{d}x$$

This integral is a bit hard to compute though, so how can we compute this without losing time and computational power? We can approximate it. Consider a single neuron a:

- at test time, the value of $\mathbb{E}[a]$ is equal to

$$\mathbb{E}[a] = w_1x + w_2y$$

- at training time, the value of $\mathbb{E}[a]$ is instead equal to

$$\mathbb{E}[a] = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) + \frac{1}{4}(0x + w_2y) + \frac{1}{4}(0x + 0y) = \frac{1}{2}(w_1x + w_2y)$$

Notice how the expectation value at training time is just multiplied by the value of the dropout: this makes it much easier for us to compute $\mathbb{E}$.

3.5.2 Data Augmentation

One last regularization method is to perform what is called **data augmentation**. For this method,

CHAPTER 4

RNNs & Transformers

tbd