Convolutional Neural Networks: A Comprehensive Guide

Exploring the power of CNNs in image analysis

Jorgecardete · Follow

Published in The Deep Hub

14 min read · Feb 7, 2024

Image created by the author with DALL-E 3

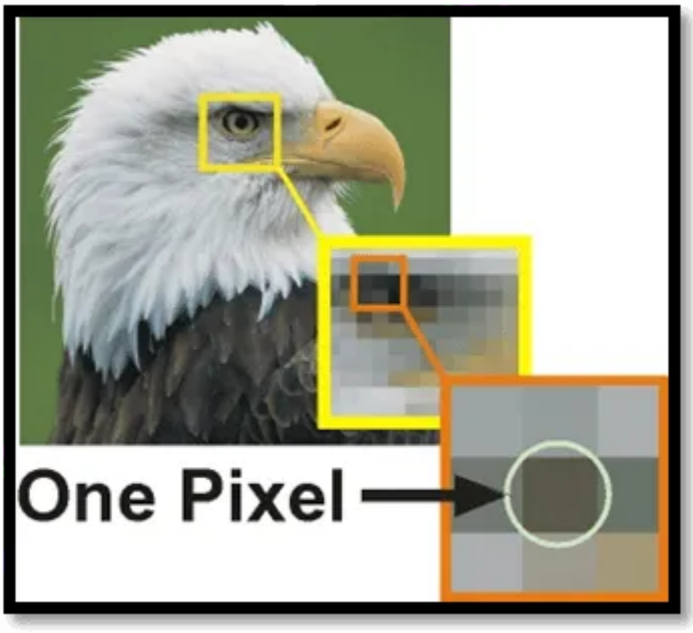
Table of contents

Convolutional Neural Networks, commonly referred to as **CNNs** are a specialized type of neural network designed to process and classify images.

If you are new to this field you might be thinking **how is it possible to classify an image?**

Well… **images are also numbers!**

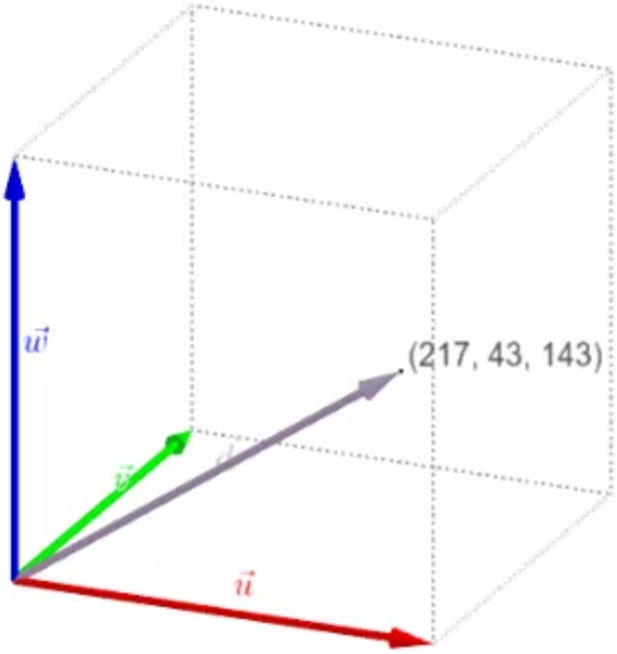
Digital images are essentially **grids of tiny units called pixels**. Each pixel represents the smallest unit of an image and holds information **about the color and intensity at that particular point**.



Pixel representation | Source

Typically, each pixel is composed of three values corresponding to the **red, green, and blue (RGB)** color channels. These values determine the **color and intensity** of that pixel.

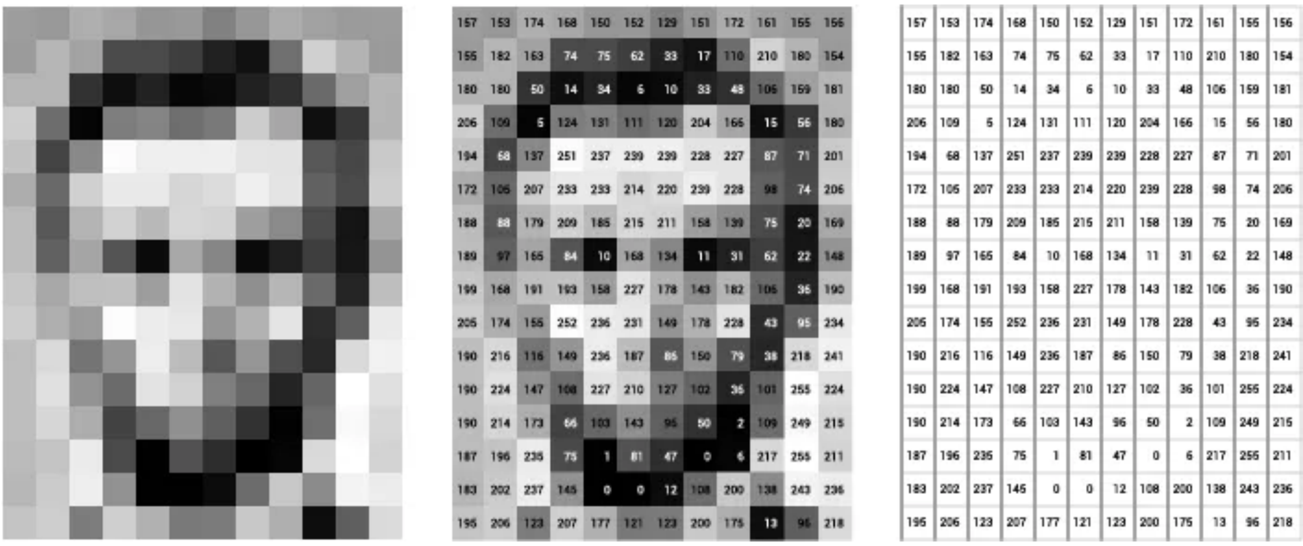
> *You can use the following tool to understand better* ***how the RGB vector is formed:***



Geogebra RGB tool | Source

In contrast, in a **grayscale image**, each pixel carries a single value that represents the intensity of light at that point.

> *Usually ranging* ***from black (0) to white (255).***



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

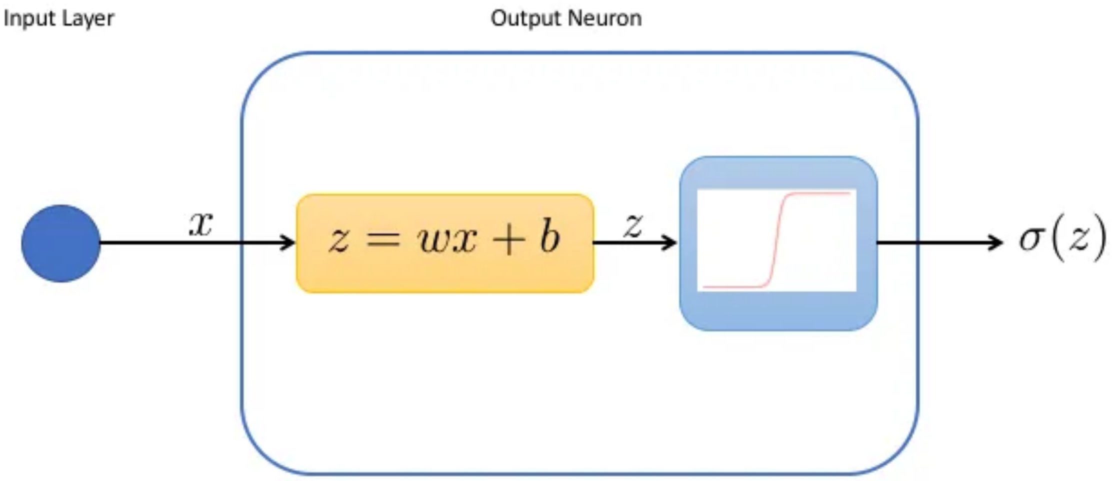
Grayscale image | Source

How do CNNs work?

To understand how a CNN functions let´s recap some of the basic concepts about Neural Networks.

> *(If you are reading this post I am assuming that you are familiar with **basic neural networks**. If that´s not the case I strongly recommend you to read this article).*

1.- **Neurons:** The most basic unit in a neural network. They are composed of a **sum of linear functions** and a **non-linear function** known as the **activation function** is applied to them.



Neuron representation | Source

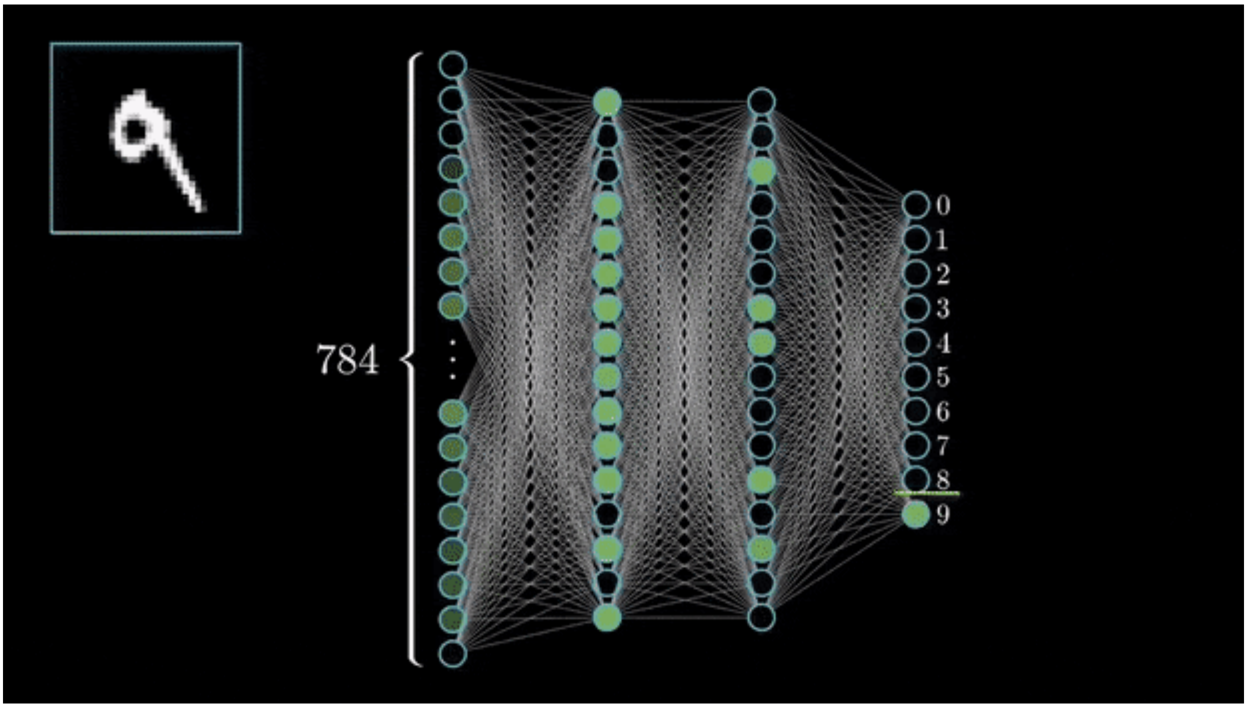
2.- **Input layer:** Each neuron in the input layer corresponds to one of the input features.

> *For instance, in an image classification task where the input is a **28 x 28-pixel image,** the input layer would have **784 neurons** (one for each pixel).*

3.- **Hidden Layer:** The layers between the input and the output layer. Each neuron in this layer is s**ummed** by the result of the neurons in the previous layers and multiplied by a **non-linear function.**

4.- **Output Layer:** The number of neurons in the output layer corresponds to the number of output classes (In case we are facing a **regression** problem the output layer will only have **one neuron**).

> *For example, in a classification task with digits* ***from 0 to 9****, the output layer would have* ***10 neurons****.*



Nerual Network process | Source: 3Blue1Brown

Once a prediction is made, a **loss** is calculated and the network enters a **self-improvement iterative process** through which the weights are adjusted with **backpropagation** to reduce this error.

Now we are ready **to understand convolutional neural networks!**

The first question we should ask ourselves:

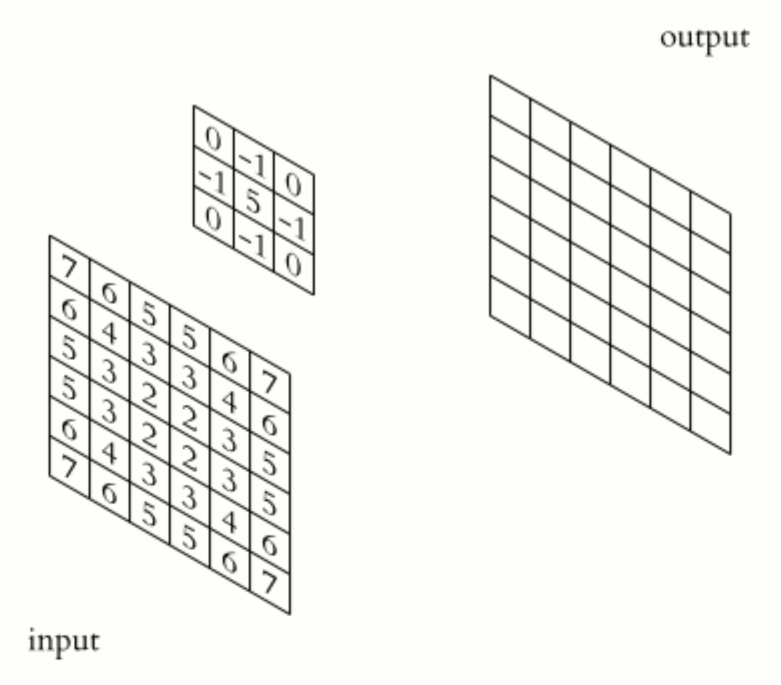
- What makes a CNN different from a basic neural network?

Convolutional layers

They are the fundamental building blocks of CNNs. These layers perform a critical mathematical operation known as **convolution.**

This process entails the application of **specialized filters known as kernels**, that traverse through the input image to learn complex visual patterns.

Kernels
They are essentially small matrices of numbers. These filters move across the image performing **element-wise multiplication** with the part of the image they cover, extracting features such as **edges, textures, and shapes**.



0	-1	0
-1	5	-1
0	-1	0

7	6	5	5	6	7
6	4	3	3	4	6
5	3	2	2	3	5
5	3	2	2	3	5
6	4	3	3	4	6
7	6	5	5	6	7

Kernel operation | Source

In the figure above, visualize the input as an image transformed into pixels.

We multiply each term of the image by a 3 × 3 matrix (this shape can vary) **and pass it into an output matrix**.

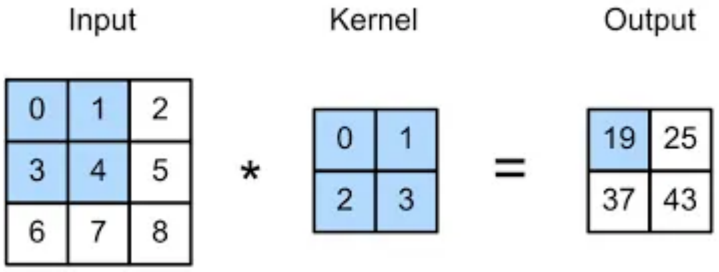
There are various methods to decide the digits inside the kernel. This will depend on the effect you want to achieve such as detecting edges, blurring, sharpening…

> ## But what are we doing exactly?

Let´s take a deeper look at it.

Convolution Operation
The convolution operation involves multiplying **the kernel value**s by the **original pixel values** of the image and then **summing up the results.**

This is a basic example with a 2 × 2 kernel:



We start in the left corner of the input:

- *(0 × 0) + (1 × 1) + (3 × 2) + (4 × 3) =* ***19***

Then we slice one pixel to the right and perform the same operation:

- *(1 × 0) + (2 × 1) + (4 × 2) + (5 × 3) =* ***25***

After we completed the first row we move one pixel down and start again from the left:

- *(3 × 0) + (4 × 1) + (6 × 2) + (7 × 3) =* ***37***

Finally, we again slice one pixel to the right:

- *(4 × 0) + (5 × 1) + (7 × 2) + (8 × 3) =* ***43***

The output matrix of this process is known as the **Feature map.**

Perfect, now we understand how **this operation works!** But…

> Why is it so useful? We are just multiplying and adding pixels, how can we extract image features doing this?

For now, I won´t be diving deeper into the convolution operation because I don´t consider it to be pivotal for understanding Conv. nets in the beginning.

However, if you are very curious I will leave you what I believe to be **the best public answer** to that question:

That´s it, you´ve understood the most fundamental concept behind CNNs, **Convolutional Layers**!

At this point, you may be having a bunch of doubts (at least I had them).

I mean, we understand **how a convolution works**, but:

- Kernels always traverse through the image matrix **one pixel at a time**?
- What happens with the **pixels in the corners**, we are only passing over them one time, what if they have an important feature?
- And what about **RGB images**? We stated that they are represented in **3 dimensions**, how does the kernel traverse over them?

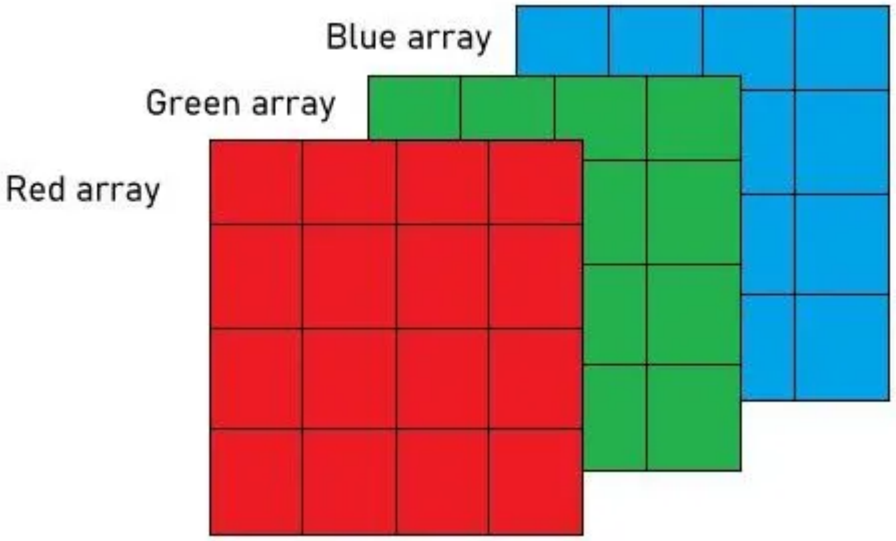
These are a lot of questions but don´t worry, all of them have an easy answer.

We'll start by understanding **three essential components** inside convolutional layers:

1. *Channels*
2. *Stride*
3. *Padding*

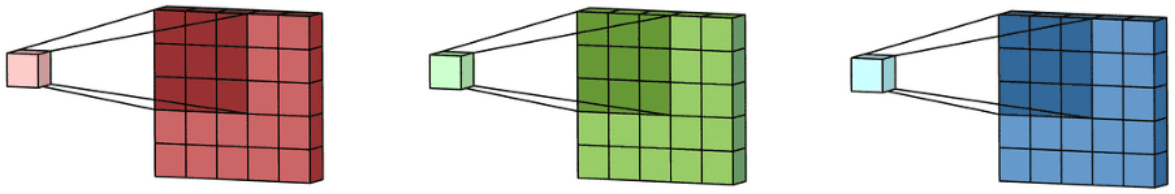
1.- Channels

As I explained before, digital images are often composed of **three channels (RGB)** which are represented in three different matrices.



RGB decomposed image | Source

For an RGB image, there are typically **separate kernels for each color channel** because different features might be more visible or relevant in one channel compared to the others.



Convolution operation in Red, Green, and Blue channels | Source

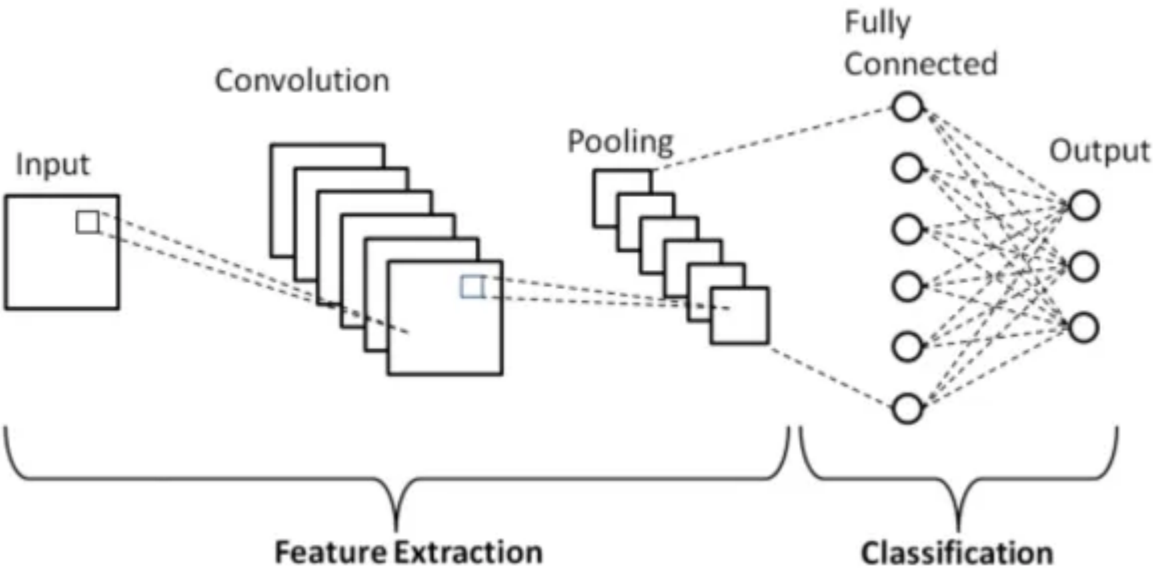
- **Depth of the layer**

The **'depth'** of a layer refers to the number of kernels it contains. Each filter produces a separate **feature map**, and the collection of these feature maps forms the ***complete output of the layer***.

> The output normally has multiple channels, where each channel is a feature map corresponding to a particular kernel.

In the case of RGB, we typically use **one channel** for each of the 3 matrices, but we can add as many as we want.

> ***For example***, *let´s say that you have a gray-scale image of a cat, you could create a channel specialized in detecting the ears and another in the mouth.*



CNN representation | Source

This image illustrates the concept quite well, think of each layer in the convolution as a feature map with a different kernel (don´t worry about the pooling part for now, we`ll break it down in a minute).

> ☣ ***BE CAREFUL*** *with misunderstanding the channels in the convolution layer with the color channels in the image. That was a representative example to understand the concept but* ***you can add as many channels as you want****.*
>
> *Each channel will detect a* ***different feature*** *in the image based on the values you assign to its kernel.*

2.- Stride

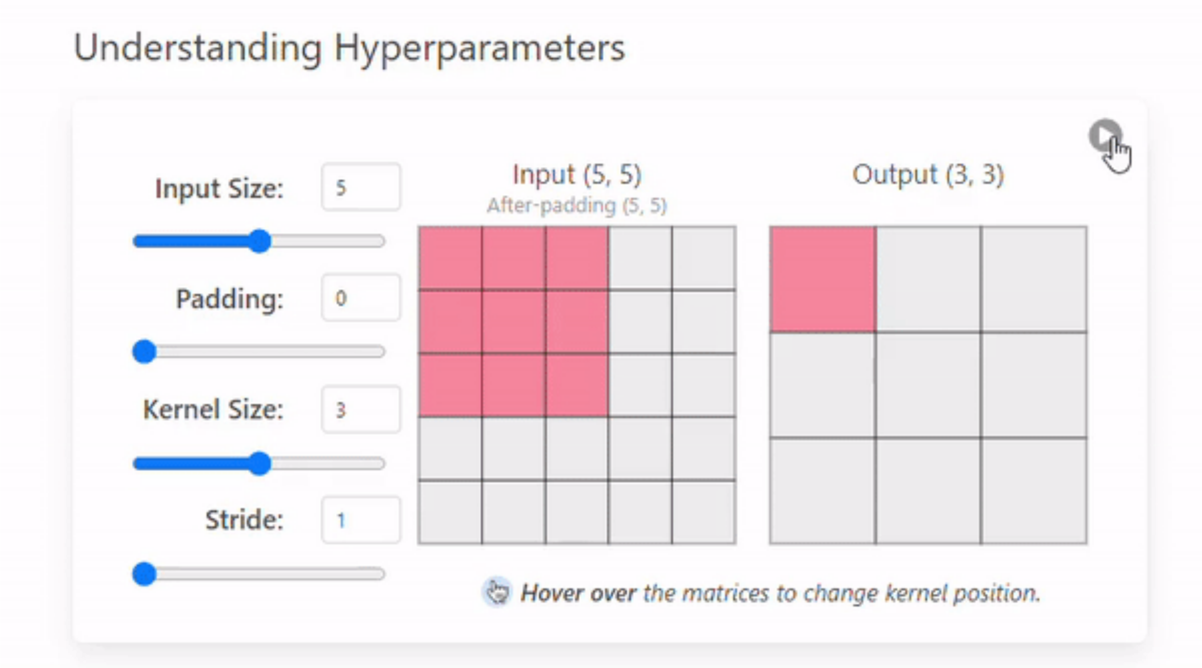
We have discussed that in a convolution a kernel moves through the pixels of an image, but we haven´t talked about the different ways in which it can do it.

Stride refers to **the number of pixels by which a kernel moves across the input image.**

The example we saw before had a stride of 1, but this can change.

Let´s see a visual representation:

- Stride = 1



Understanding Hyperparameters

Input Size: 5

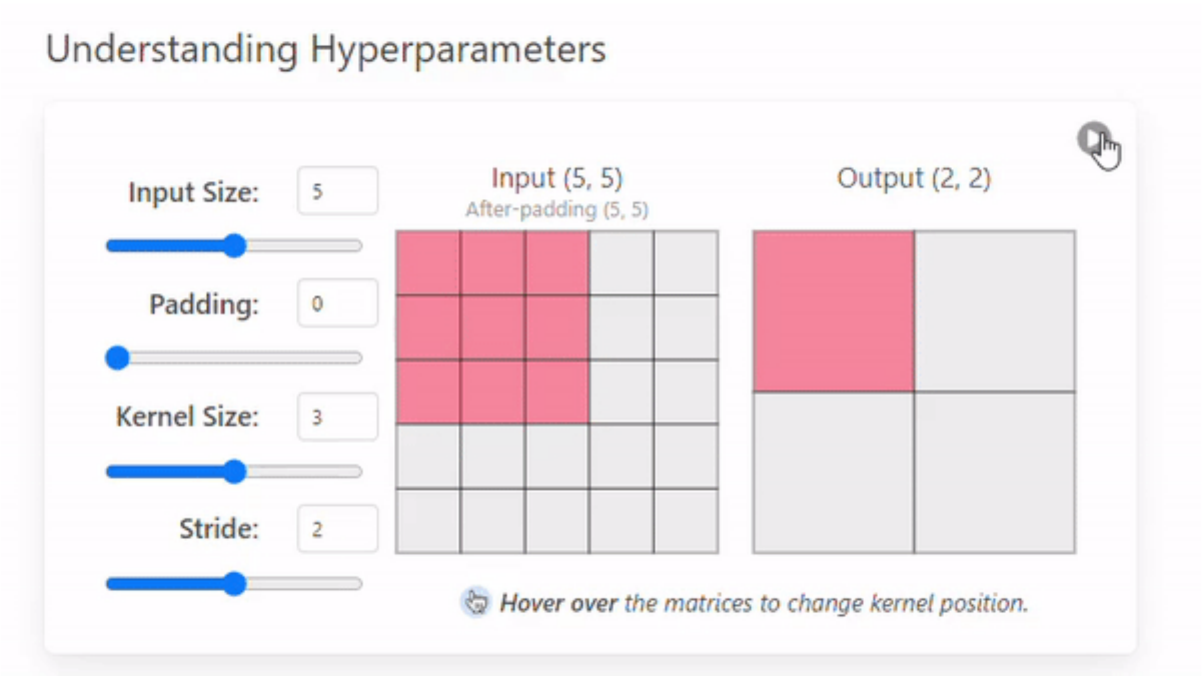
Padding: 0

Kernel Size: 3

Stride: 1

Hyperparameters of a Convolutional Layer | Source

- Stride = 2



Understanding Hyperparameters

Input Size: 5

Padding: 0

Kernel Size: 3

Stride: 2

Hyperparameters of a Convolutional Layer | Source

A stride of 2 not only changes the way the convolution iterates over the input size but also the output by making it smaller (2 × 2).

Taking this into account we can conclude that:

> *A **larger stride** will produce smaller output dimensions (as it covers the input image faster), whereas a **smaller stride** results in a larger output dimension.*

But why would we want to change the stride?

Increasing the stride will allow the filter to cover a **larger area of the input image**, which can be useful for capturing **more global features**.

In contrast, **lowering** the stride will capture **finer and more local details.**

In addition, increasing the stride will control **overfitting** and **reduce computational efficiency** as it will reduce the spatial dimensions of the feature map.

3.- Padding

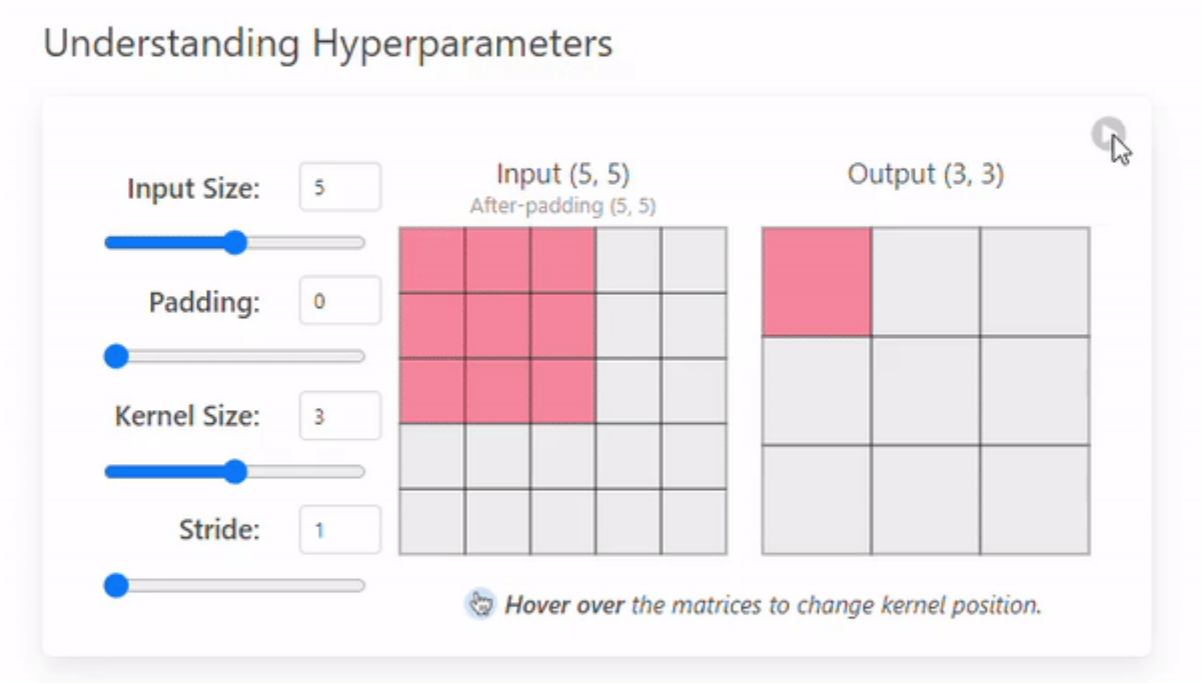
Padding refers to the **addition of extra pixels around the edge** of the input image.

When you focus on the pixels in the image's edges, you'll notice that **we traverse them fewer times** compared to those **positioned in the center**.

The purpose of padding is to **adjust the spatial size** of the output of a convolutional operation and to **preserve spatial information at the borders**.

Let´s see another example with the CNN explainer

- Padding = 0 (focus on the edges and count how many times the kernel is passing through them)



Understanding Hyperparameters

Input Size: 5

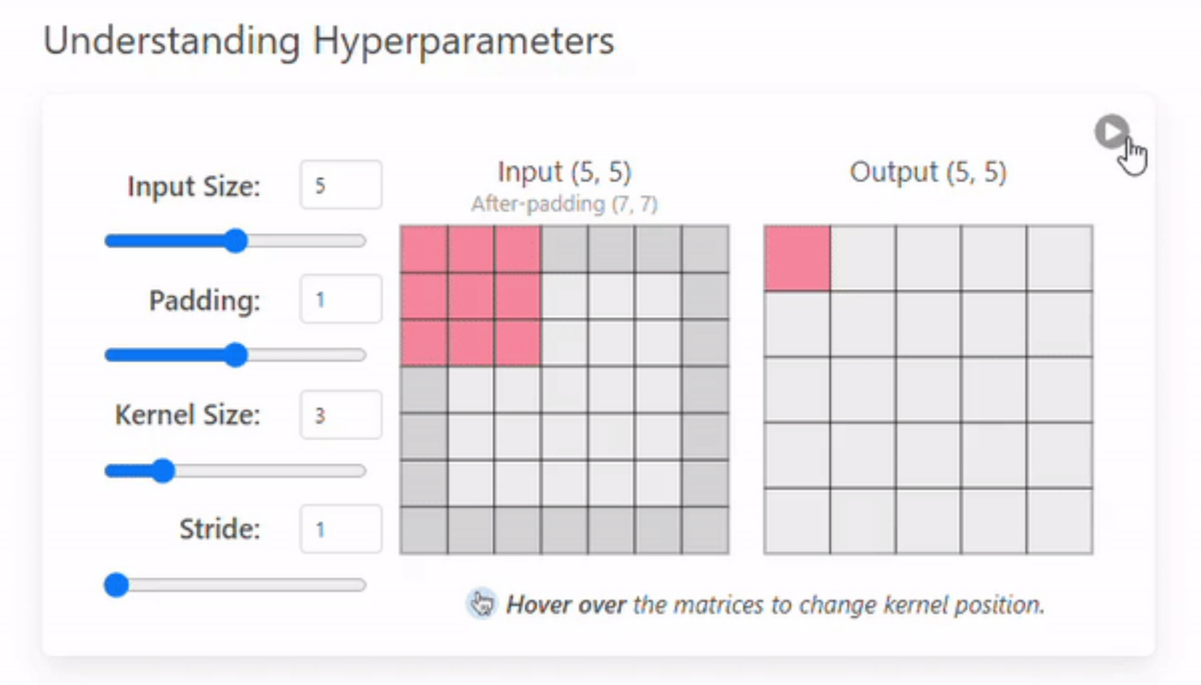
Padding: 0

Kernel Size: 3

Stride: 1

Hyperparameters of a Convolutional Layer | Source

- Padding = 1



Understanding Hyperparameters

Input Size: 5

Padding: 1

Kernel Size: 3

Stride: 1

Hyperparameters of a Convolutional Layer | Source

Now we are passing more times through the pixels in the edges and getting more information about them.

In which cases do you want to apply padding?

Mainly when the edges of the image **contain useful information** that you want to capture.

> *You can increase the padding up to the kernel size you are using.*

And how does it affect the output field?

Padding **increases the size of the output feature map.** If you increase the padding while keeping the kernel size and stride constant, the convolution operation has more "room" to take place, **resulting in a larger output.**

The output size of a convolutional layer can be calculated using the following formula:

$$\text{Output size} = \frac{\text{Input size} + 2 \times \text{Padding} - \text{Kernel size}}{\text{Stride}} + 1$$

Where

- **"2 × Padding"** accounts for padding applied to both the left and right sides (or top and bottom sides) of the input.
- **"+ 1"** accounts for the initial position of the filter, which starts at the beginning of the padded input.

> ☣ *This is a visual explanation of Padding but at a practical level, it doesn´t have to be **always the same on all sides of the image.***
>
> *The padding dimensions can be **asymmetric** or even have a **custom padding** design.*

If you have reached this point now you can officially say that you know how Convolutional Layers work!

Nevertheless, **this is not the end of the journey...**

There is a common misconception among beginners that Conv. layers are Convolutional Neural Networks.

Well, convolutional layers are an essential component, but as its name indicates, they are a **LAYER** inside CNNs.

We have comprehended the most important part of CNNs, but there are still **two other special types of layers** that we have to understand:

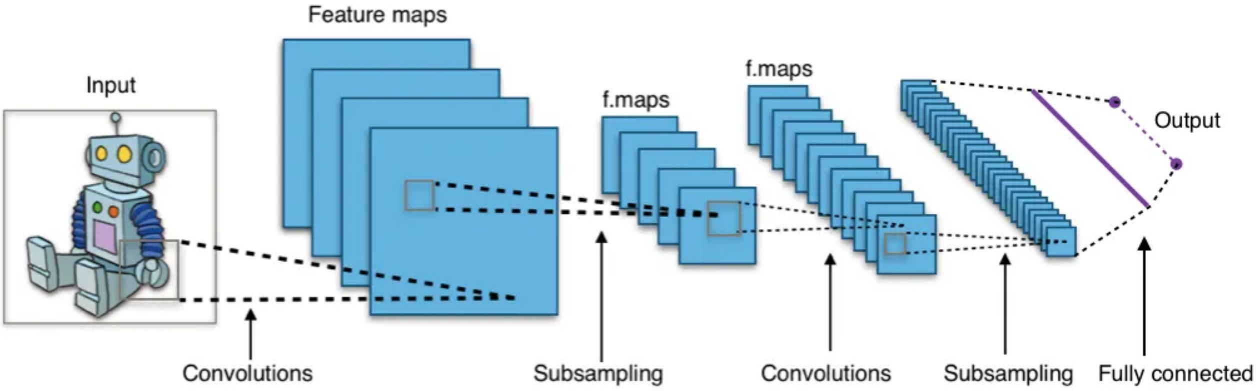
- Pooling Layers
- Flattening Layers

Pooling Layers

Before explaining how these layers work **it´s crucial to have this clear:**

> *Although Convolutional Layers can decrease the output size, their principal objective is not* ***DIMENSIONALITY REDUCTION****.*
>
> *The main objective of Convolutional Layers is* ***FEATURE EXTRACTION****.*

In fact, in most cases we are **not reducing the dimensions** of our data because we are creating **new channels** that weren´t there before, so even if our feature map dimensions are smaller, **we have more of them.**



Convolutional neural network representation | Source

Take a look at this example, here we might be reducing a bit our feature map in each Convolutional Layer but we are creating much more channels.

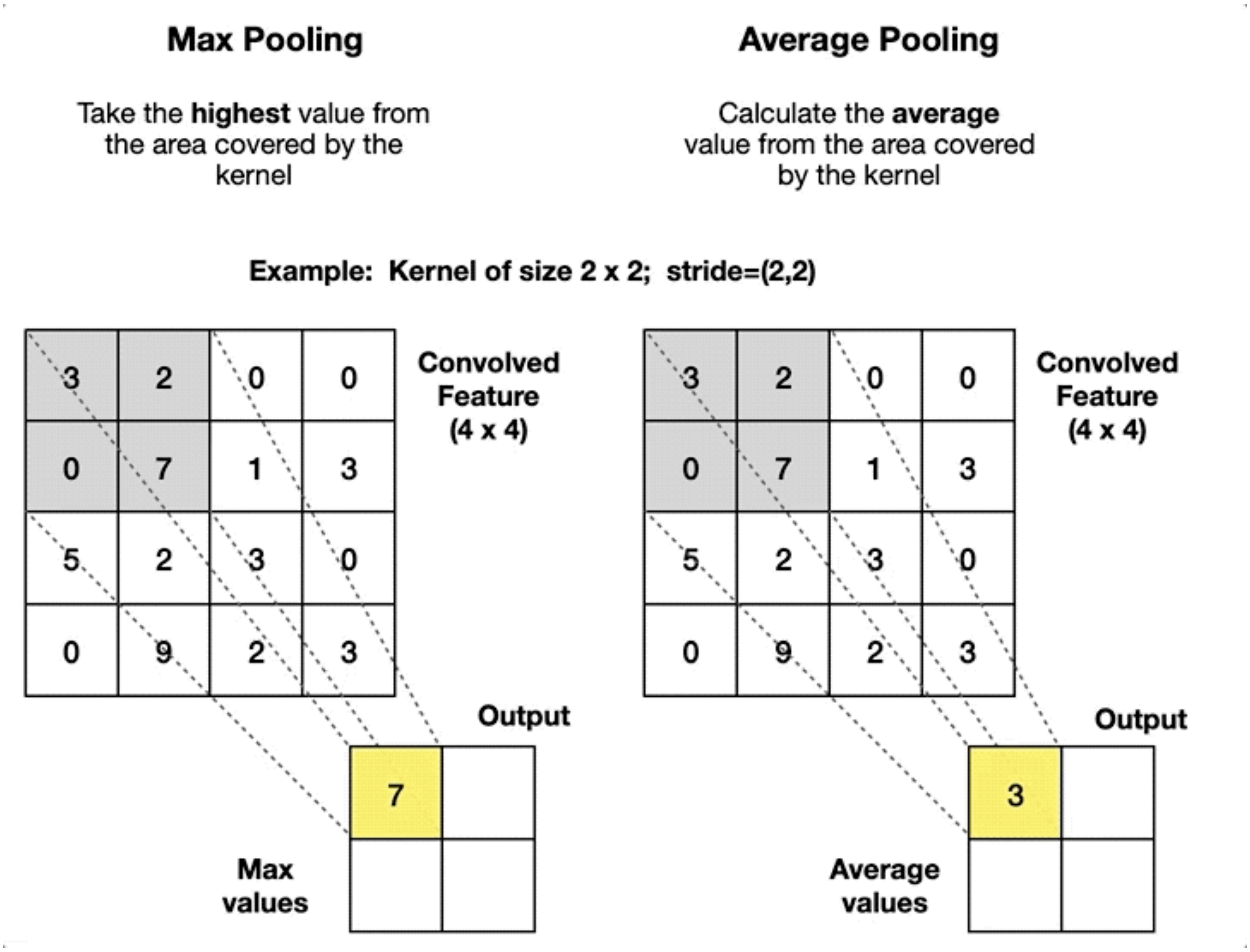
What about the subsampling layers?

Those are pooling layers and its main objective is indeed **dimensionality reduction!**

How Pooling Layers Work

Imagine you have a large image and want to make it smaller but keep a**ll the important features** like edges and colors.

The pooling layer operates independently on every depth slice of the input. It resizes it spatially, using the **Max** or **Average** of the values in a window slid over the input data.


Max Pooling
Take the **highest** value from the area covered by the kernel
Average Pooling
Calculate the **average** value from the area covered by the kernel
Example: Kernel of size 2 x 2; stride=(2,2)
Convolved Feature (4 x 4)
Output
7
Max values
Convolved Feature (4 x 4)
Output
3
Average values

3	2	0	0
0	7	1	3
5	2	3	0
0	9	2	3

3	2	0	0
0	7	1	3
5	2	3	0
0	9	2	3

Max and **Avg** Pooling Layers | Source

In this example, we have reduced the feature map from (4 × 4) to (2 × 2).

What is the difference between pooling and the

convolution operation?

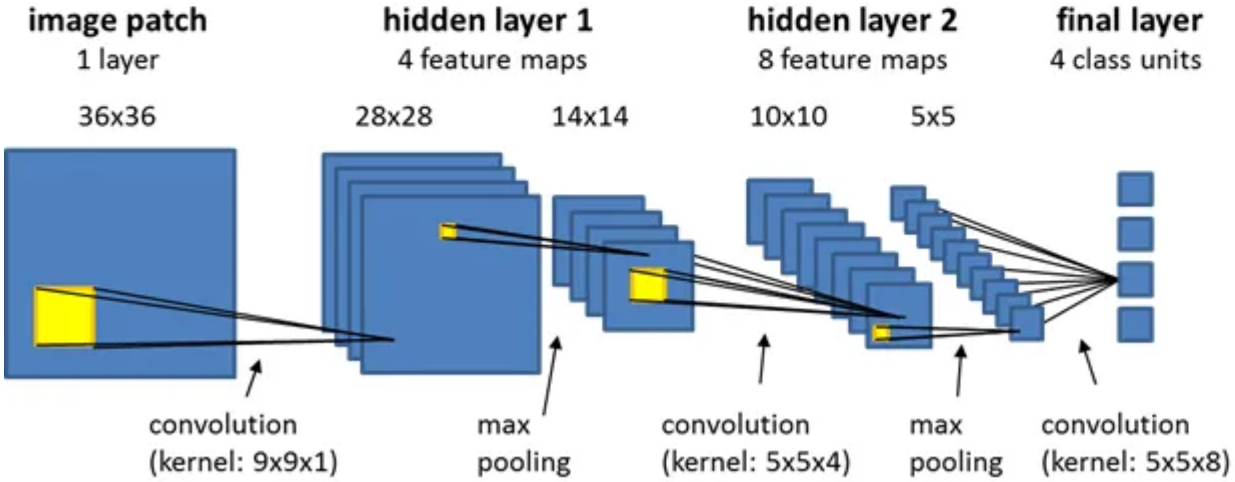
In **pooling**, we are not applying any kernel to the input data, we are just **simplifying the information** with a math operation (Max or Avg).

What about the channels, pooling also reduces the number of channels?

You must understand this:

> *Pooling layers* ***DO NOT REDUCE THE NUMBER OF CHANNELS****.*
>
> *Each pooling operation* ***IS APPLIED INDEPENDENTLY TO EACH CHANNEL*** *of the input data.*

Let´s see another example, channels can be a bit complex to visualize at first and I want to ensure that you understand correctly how they work.



Layers inside a CNN | Source

This is a good representation, here you can see how each pooling layer is **reducing the dimensions of the spatial space** but it's not **reducing the number of channels.**

The number of channels is not reduced until the end of the architecture.

> *With Convolutional and Pooling layers* ***we CAN´T reduce the number of channels,*** *just add more to the existing ones.*

So why and how do we combine all these channels?

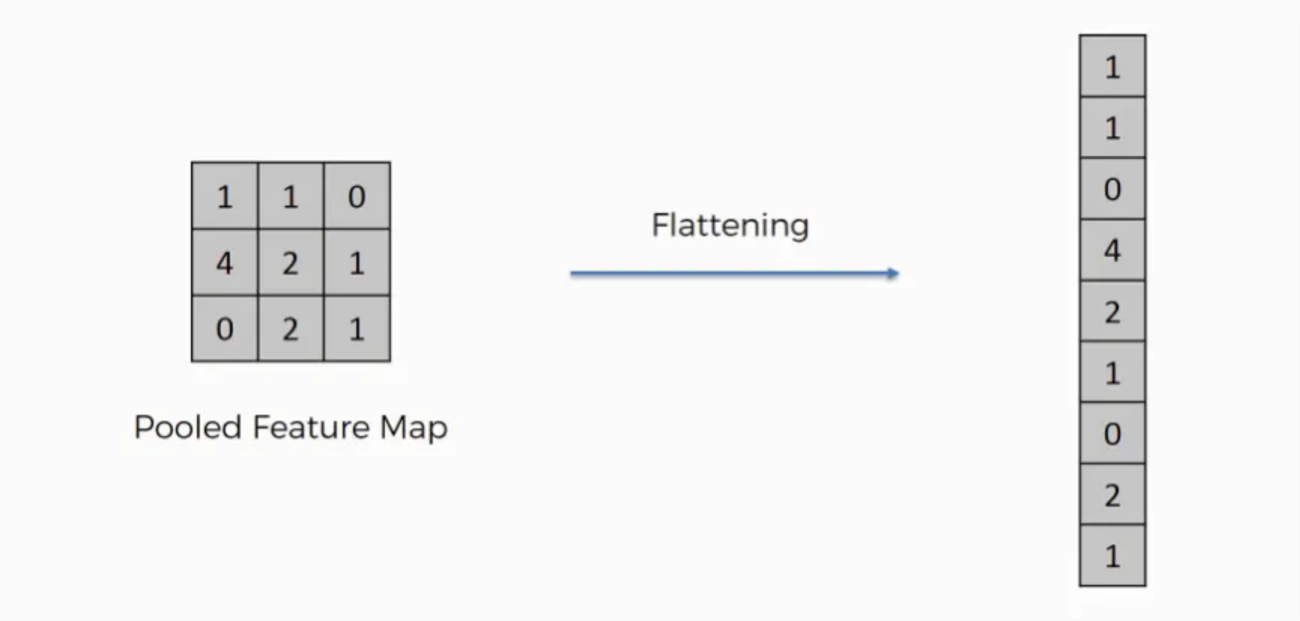
After **convolutional** and **pooling** layers have **extracted relevant features** from the input image we have to turn this high-dimensional feature map into a format suitable for feeding into fully connected layers.

Here´s where **flattening layers come into action!**

Flattening layers

Imagine you have a grid of data (like pixels in a feature map), and you want to line up all of these grid points in a single, long line.

That's what flattening does. It takes the entire feature map and reorganizes it into a **single, long vector**.



Flattening concept | Source

> ☣️ *Although flattening changes the shape of the data, it does not make any changes to the actual information.*

- Why do we need flattening layers?

Integration of features
By flattening the feature maps into a vector, the network can integrate the spatially distributed features extracted for tasks such as classification.

Compatibility with Dense Layers
Fully connected layers (dense layers) are designed to operate on **1-dimensional data**, hence, flattening is a necessary step to transition from the multidimensional tensors produced by convolutional layers to the format required for dense layers.

This leads us to our next question:

Why do we need Dense Layers in CNNs?

While convolutional layers are good at **detecting features** in input data, dense layers are essential for **integrating these features into predictions**.

For example, if we design a convolutional neural network for **facial recognition**, early layers might detect **edges and textures**, while dense layers might interpret these to **recognize specific facial features**.

> *Without dense layers, CNNs would not be able to perform the* ***high-level tasks*** *that are often required, such as* ***classifying images, detecting objects****, or* ***making predictions*** *based on visual inputs.*

CNN recap

Up to this point, we have revised the whole CNN structure:

- Convolutional Layers
- Pooling layers
- Flattening layers

- Dense layers

With the fundamental concepts of **channels, stride**, and **pooling**.

We could say that we have joined all the pieces of the puzzle!

Or maybe not… **what about activation functions and backpropagation?**

Backpropagation functions similarly in feed-forward neural networks but with some special adjustments. I won´t focus much on its technical details.

> *You can check out this very interesting article to know more about it!*

Convolutions and Backpropagations

Ever since AlexNet won the ImageNet competition in 2012, Convolutional Neural Networks (CNNs) have become ubiquitous...

pavisj.medium.com

> *If you know nothing about Backpropagation you can start by taking a look at my publication:*

Backpropagation

From Mystery to Mastery: Decoding the engine behind Neural Networks.

medium.com

However, I will certainly take a look at **activation functions**.

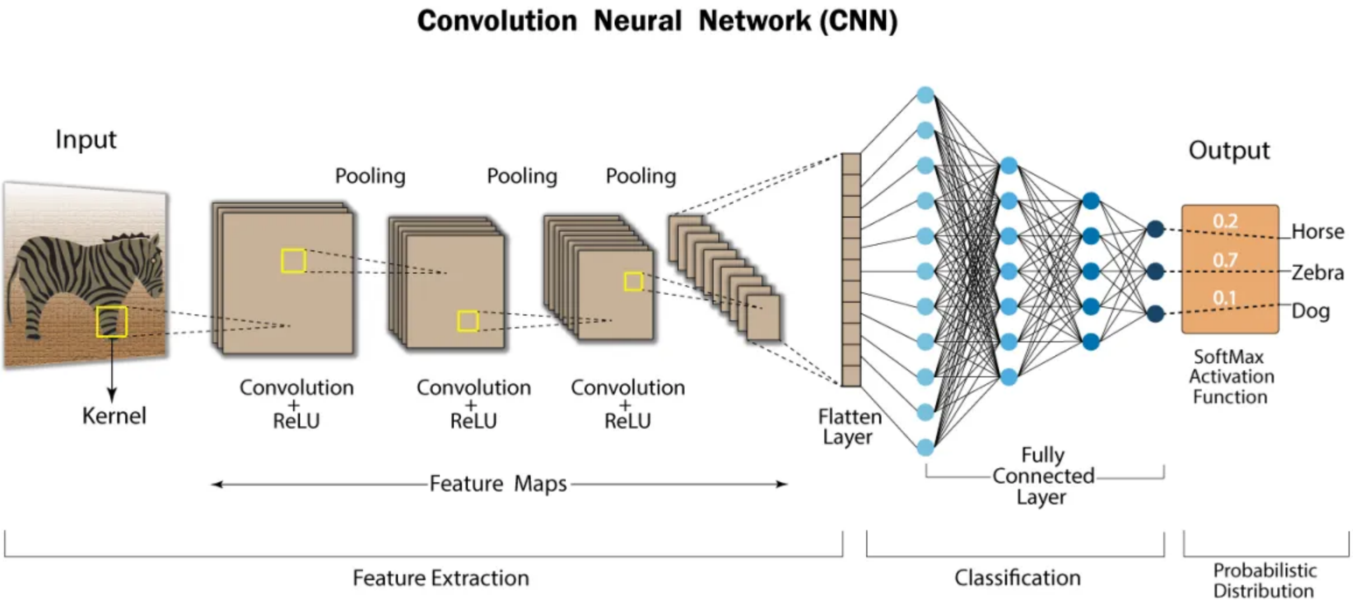
Activation functions in Convolutional Neural Networks

As you may know activation functions are indispensable, otherwise, we would be creating a very large linear model.

As in simple neural networks, we also need these **non-linear terms** in ConvNets. However, **not all the layers we have seen have an activation function.**

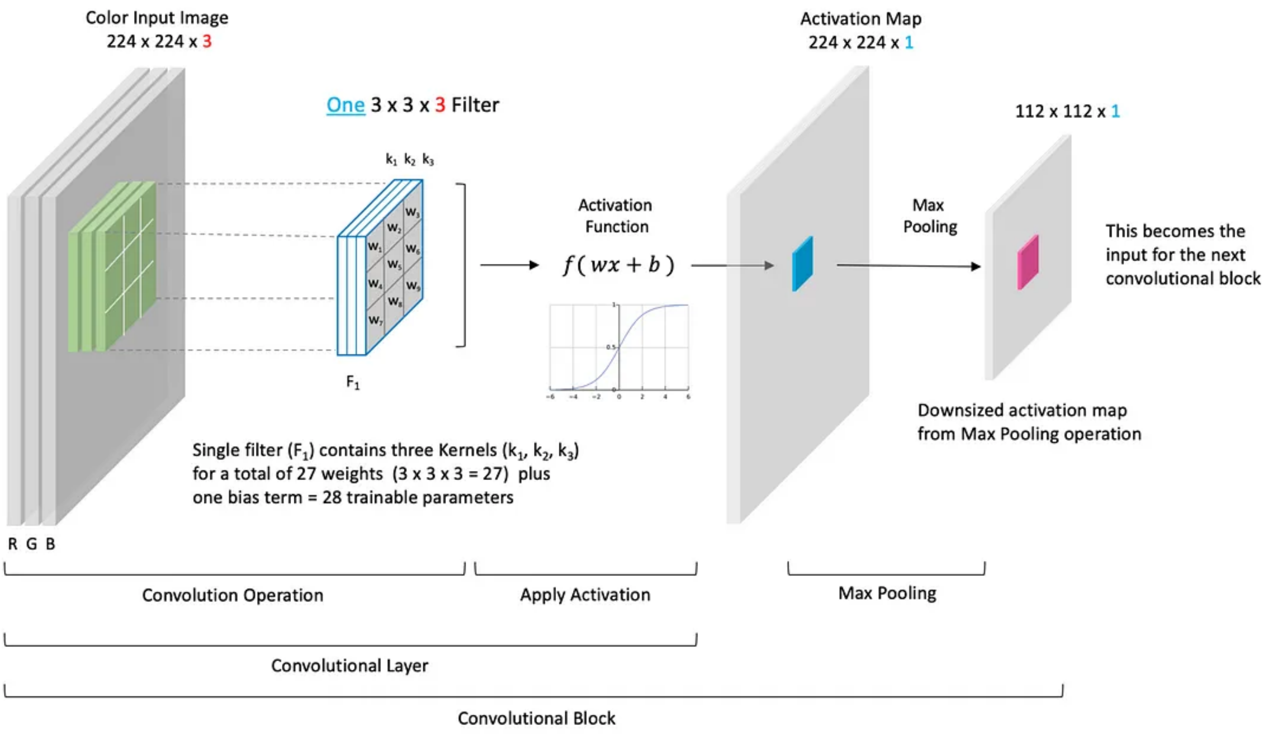
Let's use an image as a reference to visualize this. Now you should understand the representation without any problem! Just one little thing…

> *The first two **pooling layers** are not shown in this diagram, this is another way of visualizing CNNs, it doesn´t mean that they are not there, just imagine a **filter between each layer that makes them smaller**.*



Complete CNN representation | Source

In the **feature extraction** part, the activations will be in the **convolutional layers.** The process is quite straightforward, after each convolution operation you multiply the result by an activation function.



This becomes the input for the next convolutional block

Downsized activation map from Max Pooling operation

Single filter (F_1) contains three Kernels (k_1, k_2, k_3) for a total of 27 weights (3 x 3 x 3 = 27) plus one bias term = 28 trainable parameters

Convolutional layer structure | Source

The **pooling** and **flattening** layers **DON´T have an activation function.**

As we explained before the main function of pooling layers is **dimensionality reduction** and the main purpose of flattening layers is **restructuring the data into a 1D vector.**

We **don´t need to include non-linearities** for doing that. Nevertheless, we do need activation functions for extracting **complex features** (we won´t be able to capture relevant characteristics of an image with only a linear function).

In the **classification part**, all the fully connected layers and the output layer will have an activation function, as in simple neural nets.

Here we also need an activation function because we are using the features extracted to make a classification or a prediction, and the algorithm has to **learn complex interactions** (as a simple neural network would do).

Activations — Convolutional and dense layers

ReLU: is the most common activation function. It outputs the input directly **if it is positive**, otherwise, it **outputs zero**. It has the benefit of **reducing training time** and mitigating the **vanishing gradient problem**.

Leaky ReLU: A variation of ReLU that allows a **small, non-zero gradient** when the unit is inactive, which can help **prevent dead neurons** during training.

Activation functions: ReLU vs. Leaky ReLU

It's never too late to board the 'Learning and discussing the insights' train, and here are my two cents on my recent...

medium.com

Activations — Output layer

Sigmoid: Produces an output in the **range (0, 1)**. It's **not commonly used in hidden layers anymore** due to the vanishing gradient problem, but it's still used for **binary classification** in the output layer.

Tanh (Hyperbolic Tangent): Output values in the **range (-1, 1)**. It is similar to the sigmoid but can provide better training performance for some problems due to its output range.

Activation Functions in Neural Networks

Sigmoid, tanh, Softmax, ReLU, Leaky ReLU EXPLAINED !!!

towardsdatascience.com

Bibliography

1. ***Polo Club of Data Science**. (2020). CNN Explainer.*

2. ***IBM**. (2020). Convolutional Neural Networks.*

3. ***Saha, S.** (2018). A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. Towards Data Science.*

4. ***Cireșan, D. C.** (2016). Convolutional Neural Networks for Visual Recognition.*

Springer International Publishing.

5. ***DeepLearning.TV.** (2019). Convolutional Neural Networks (CNNs) explained. [Video]. YouTube.*

Thanks for reading! If you like the article make sure to clap (up to 50!) and follow me on Medium to stay updated with my new articles.

Also, make sure to follow **my new publication!**

The Deep Hub

Your data science hub. A Medium publication dedicated to exchanging ideas and empowering your knowledge.

medium.com

AI Machine Learning Data Science Deep Learning Cnn

Follow

Published in The Deep Hub

1.1K Followers · Last published 4 days ago

Your data science hub. A Medium publication dedicated to exchanging ideas and empowering your knowledge.

Follow

Written by Jorgecardete

2.2K Followers · 4K Following

AI enthusiast - I write as I learn 🚀 https://medium.com/thedeephub

Responses (38)

What are your thoughts? Respond

Michael Ehrig
10 months ago

> sum of linear functions multiplied by a non-linear function

They are not multiplied by the non-linear function. The non-linear function is **applied** to the sum of linear functions

-- 1 reply Reply

Mindscrafter powered by Medium.com
10 months ago

> Thanks for reading! If you like the article make sure to clap (up to 50!) and follow me on Medium to stay updated with my new articles.

very detailed and comprehensive article. Thanks for sharing.

-- Reply

Pablo (Apes Ascendance)
10 months ago

Thanks for sharing amazing work on CNN's literacy! We must understand essential concepts, rehearse, and practice theory while coding :)

👏 --

Reply

See all responses