CNN Introduction and basics

Why do we need CNNs?

So far, we have learned about different artificial neural network (ANN) models. These models are very powerful; however, when dealing with problems where the input is an **image**, they have certain limitations.

For example, consider building a model to classify photos of circles and squares. As humans, we instinctively know that the object we are trying to classify could be anywhere in the image, so we subconsciously scan through it to locate the object. Only after identifying the object do we begin to analyze whether it is a circle or a square. For tasks like these, a new type of network called Convolutional Neural Networks (CNNs) performs significantly better.

CNNs use convolution to extract features from an image at different stages. This approach allows these models to utilize significantly fewer parameters when analyzing a photo compared to simple artificial neural networks (ANNs), which helps reduce the model's vulnerability to overfitting.

Because CNNs employ convolution, they are known for their translation invariance. This means they can detect and analyze objects in multiple locations within the image, regardless of their position, orientation, or other variations. For these reasons, when we think of image classification or any problem where the input is an image, we immediately consider CNNs.

CNN Introduction and basics

What is a CNN?

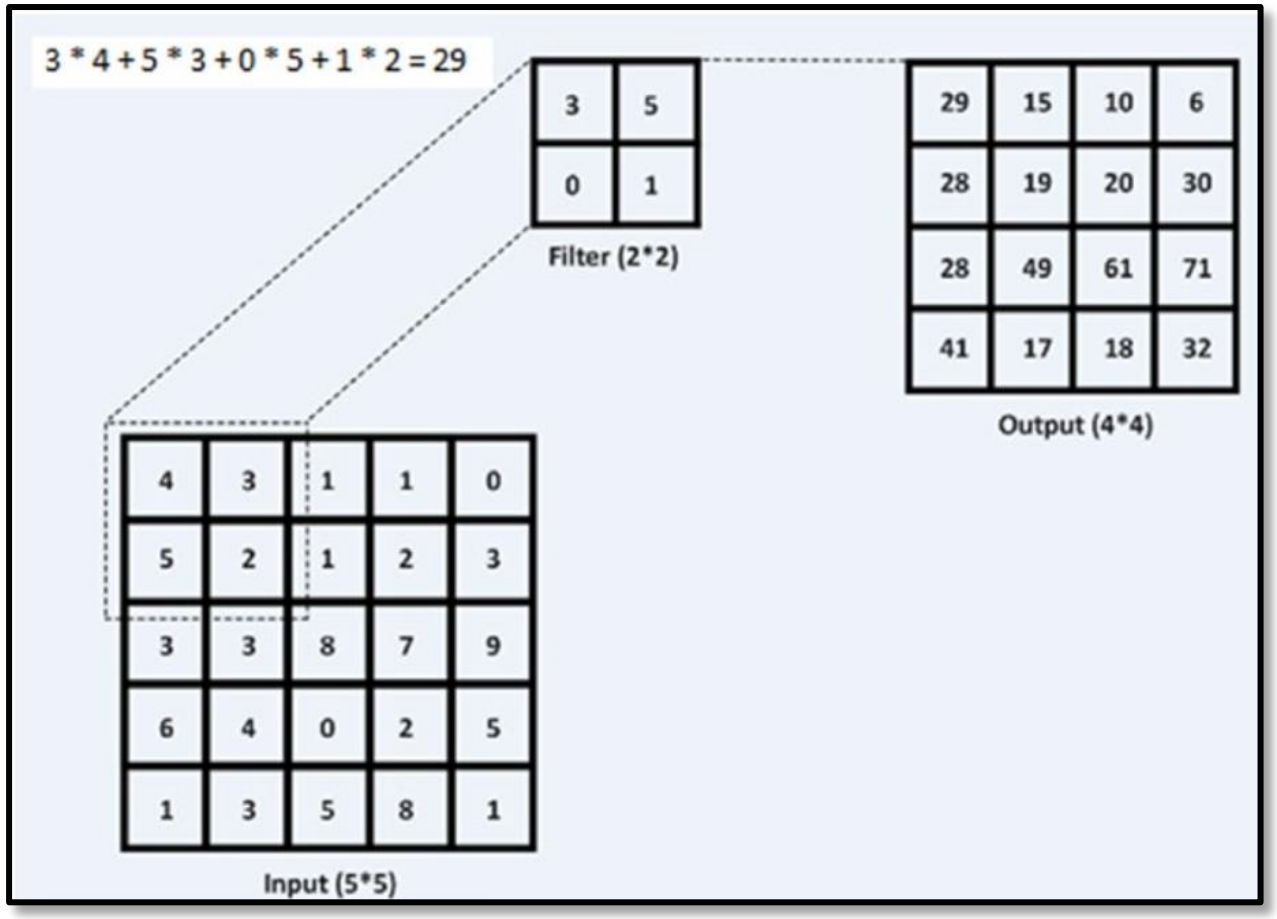
To understand what a Convolutional Neural Network (CNN) is, we first need to grasp the concept of convolution. We will start by taking the convolution of two matrices: a 5x5 matrix and a 2x2 matrix.

To calculate the convolution of these two matrices, we slide the 2x2 matrix over the 5x5 matrix, calculate the result at each position, and record it in the output matrix. For example, in the first step, we calculate the following mathematical expression:

$$3 \times 4 + 5 \times 3 + 0 \times 5 + 1 \times 2 = 29$$

Next, we move the 2x2 matrix one step to the right and repeat the same process. When we reach the end of a row, we then move the 2x2 matrix one step downward to continue the convolution process.

Convolution process



Filter, Stride and Padding

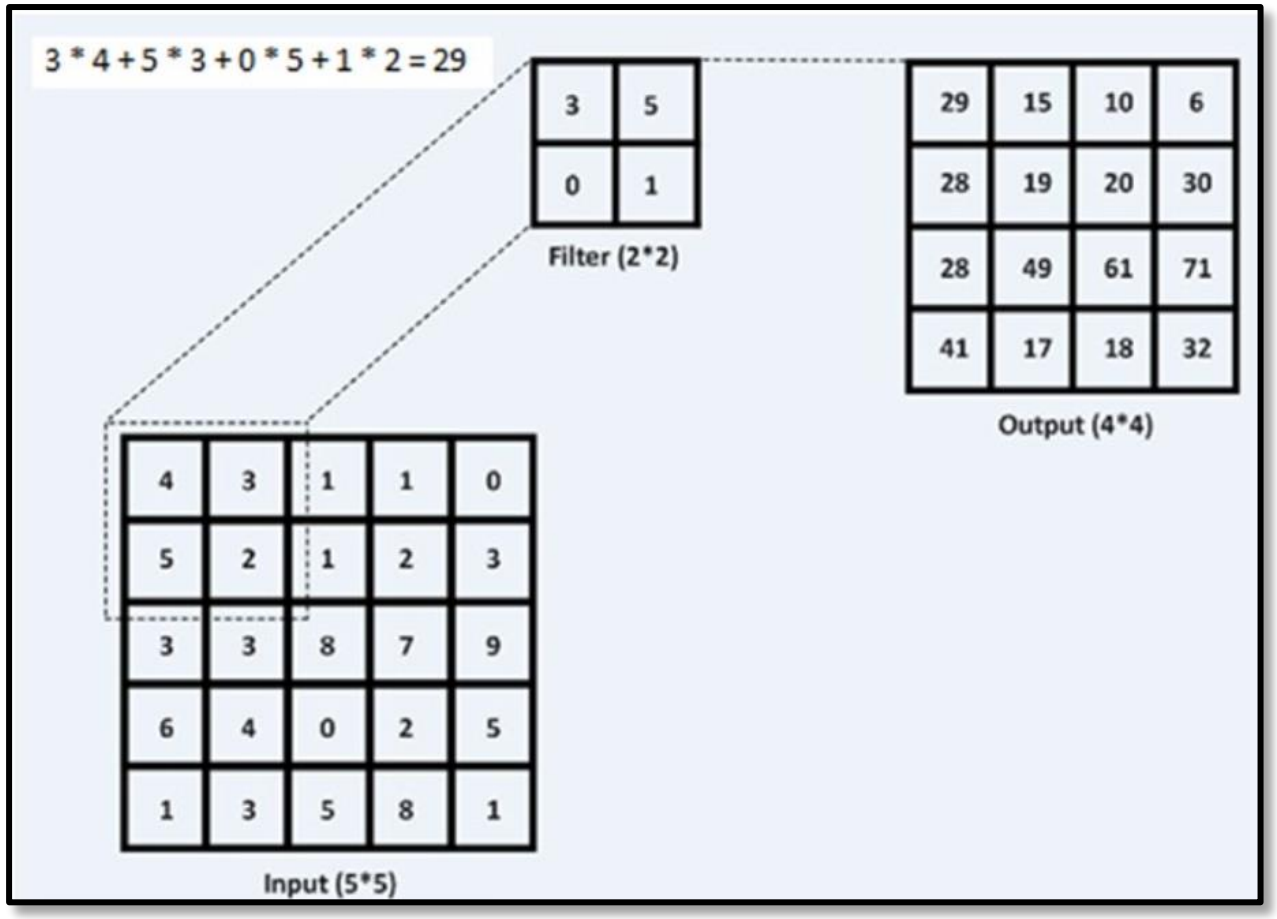
Now, let us become familiar with some key terminology in convolution.

The 2x2 matrix that we slid over the 5x5 matrix is called the **filter** or the **kernel**. It is important to note that different filters will produce different outputs.

In this example, we moved the filter one step to the right after each calculation. This value is referred to as the **stride**. For instance, if we set the stride to 2, it means that after each step, we will move the filter two steps to the right.

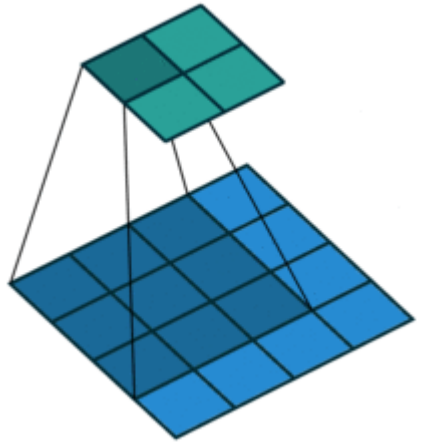
In our example, we used the initial values of the 5x5 matrix, which resulted in a change in the size of the output after the convolution. To preserve the original size, we can add a set of zeros around each edge of the matrix. This process is known as **padding**.

Convolution process

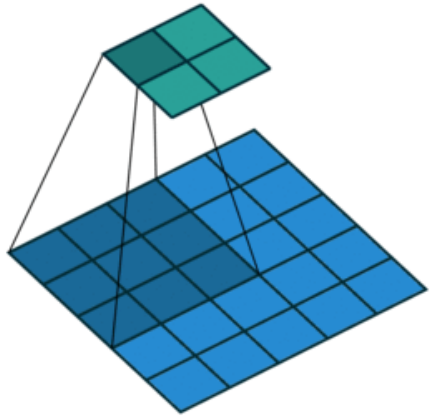


Filter, Stride and Padding

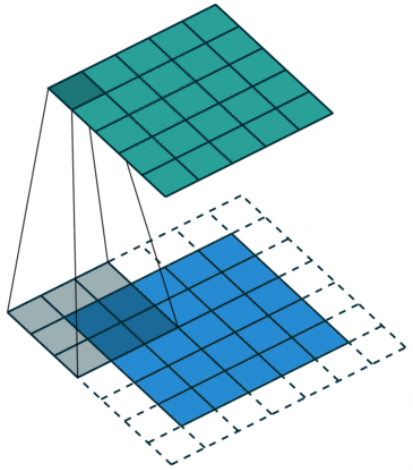
Now that we understand the meanings of filter, stride, and padding in the context of CNNs, let us examine how these elements affect the output of our convolution.



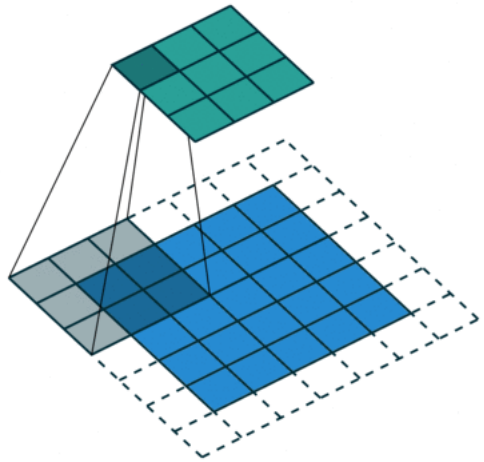
4*4 input, 3*3 filter
No padding
Stride=1



5*5 input, 3*3 filter
No padding
Stride = 2



5*5 input, 3*3 filter
Padding=1
Stride = 1



5*5 input, 3*3 filter
Padding=1
Stride = 2

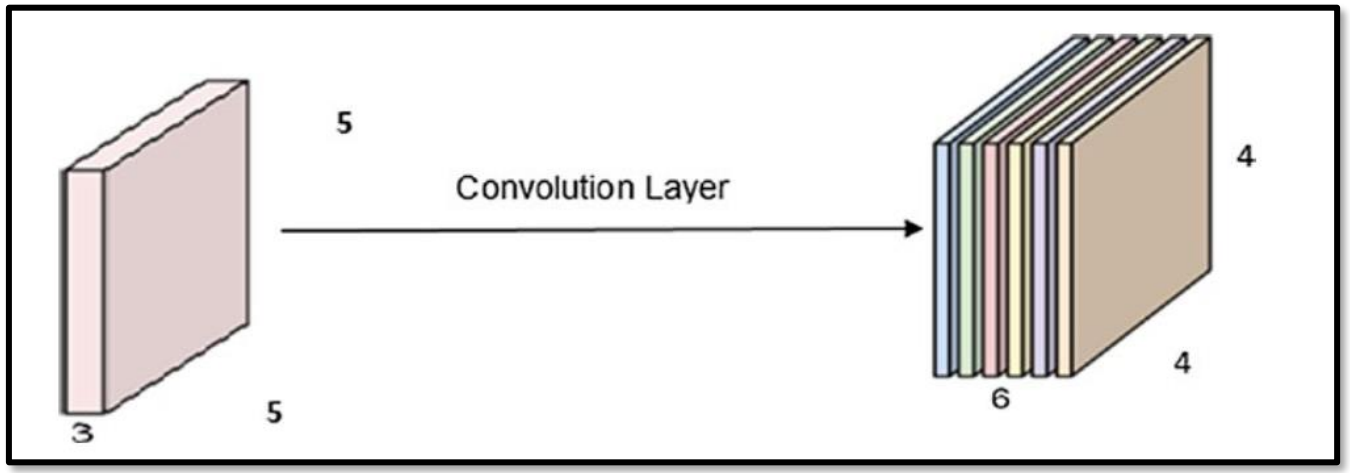
Output Size

In the example we discussed earlier, we had a 2-dimensional input matrix and a 2-dimensional filter. In practice, we could encounter problems where we have a 3-dimensional input matrix. The convolutional process remains the same in this case.

Sometimes, we use more than one filter. In such scenarios, we apply each filter separately and then stack the results together, forming a 3-dimensional output.

For example, if we have a 5x5x3 input matrix and 6 filters with a size of 2x2, using 0 padding and a stride of 1, we will end up with a 4x4x6 output matrix.

Input and output size in convolution



Output Size

Now, we know that the input size, the number of filters, padding, stride and the filter size will affect the output shape.
We can calculate the output dimensions using this formula.
Let us say the input volume is $W_1 \times H_1 \times D_1$ and we have N, ($F \times F$) filters with stride = S and padding = P. This will produce an output with the size $W_2 \times H_2 \times D_2$ where:

$$W_2 = \frac{W_1 - F + 2P}{S} + 1$$
$$H_2 = \frac{H_1 - F + 2P}{S} + 1$$
$$D_2 = N$$

Input and output size in convolution

If we plug in the numbers we had for the previous example, we will have:

$$W_1 = 5, H_1 = 5, D_1 = 1$$

$$N = 6, F = 2, S = 1, P = 0$$

$$W_2 = \frac{5 - 2 + 2 \times 0}{1} + 1 = 4$$

$$H_2 = \frac{5 - 2 + 2 \times 0}{1} + 1 = 4$$

$$D_2 = 6$$

Output size = $(4 \times 4 \times 6)$

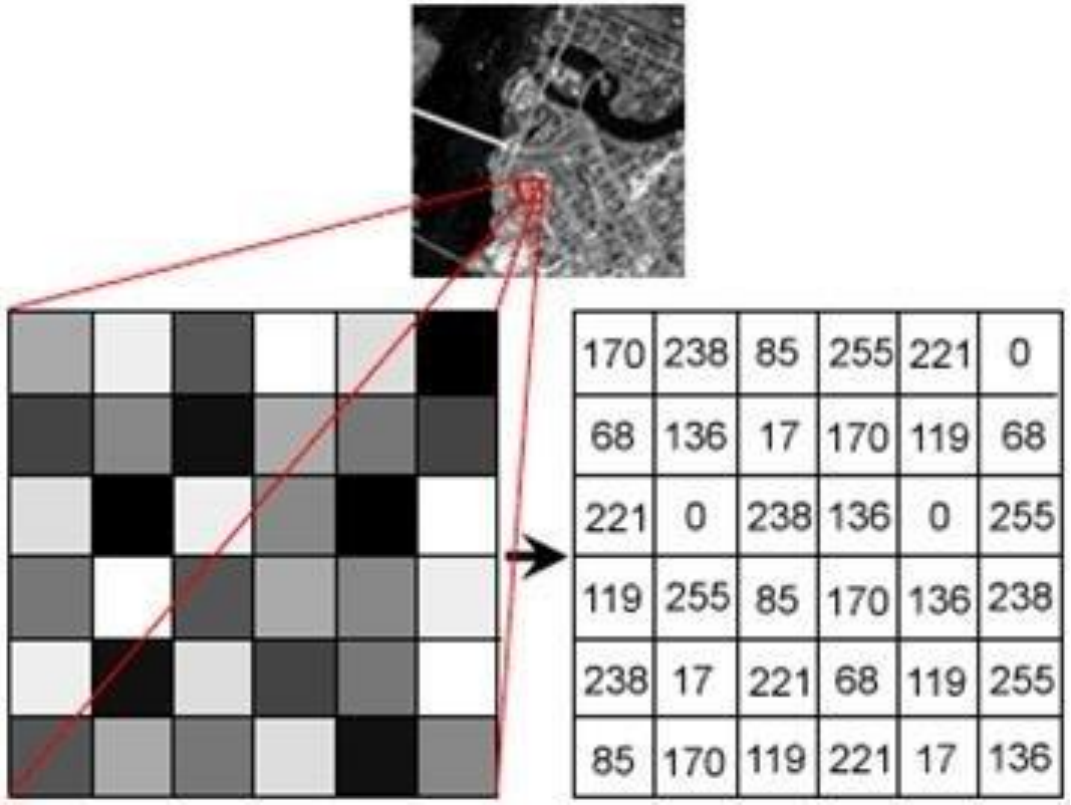
Grayscale images as matrices

Now that we know what convolution is and how it changes matrices, it is essential to understand how images can be converted into matrices before we delve into convolutional neural networks.

We will start with a grayscale image. In these images, each pixel can represent any color between white and black. If we assign 0 to black and 255 to white, then 127 would correspond to gray.

Let us consider a 512 by 512 grayscale photo. This image can be easily converted into a 512 by 512 matrix, where each element is a number between 0 and 255.

Grayscale images as matrices



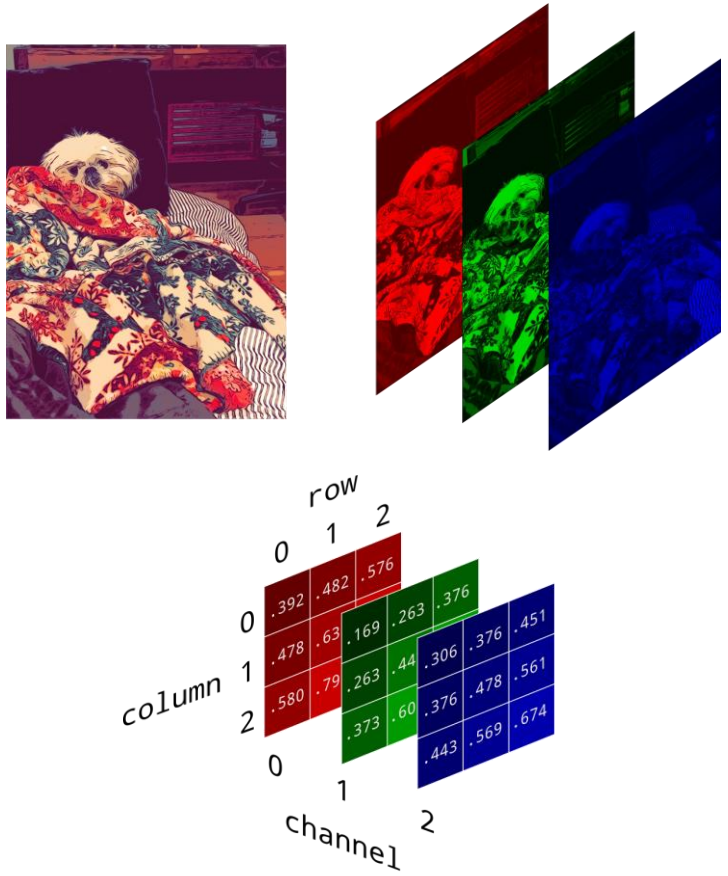
Source: www.researchgate.net

RGB images as matrices

Unlike grayscale images, which have only one channel, RGB images consist of three different channels: one for red, one for green, and one for blue. For each pixel, we will have three numbers, each representing the intensity of its corresponding color. For example, a blue pixel is represented as (0, 0, 255), while a red pixel is represented as (255, 0, 0).

Now, you might wonder how RGB images can be converted into matrices. We need a 3-dimensional matrix, where each channel will have its own 2-dimensional matrix.

RGB images as matrices



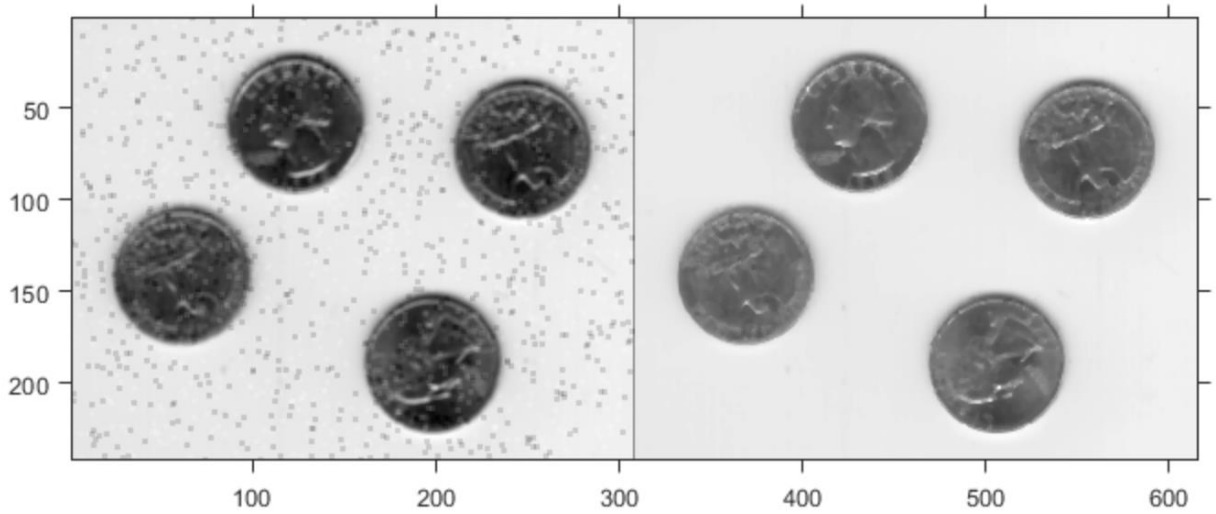
Source: www.e2eml.school

What could convolution do?

n this section, we will explore what convolution can do to a photo. One of the applications of convolution, aside from its use in CNNs, is noise reduction.

In photography, when the lighting of a photo is not ideal, the image may appear noisy. By using specialized filters and convolving the original photo with these filters, we can effectively remove noise from the picture and, consequently, drastically improve its quality.

In this example, we can see the noisy photo on the left, followed by the result after applying convolution.

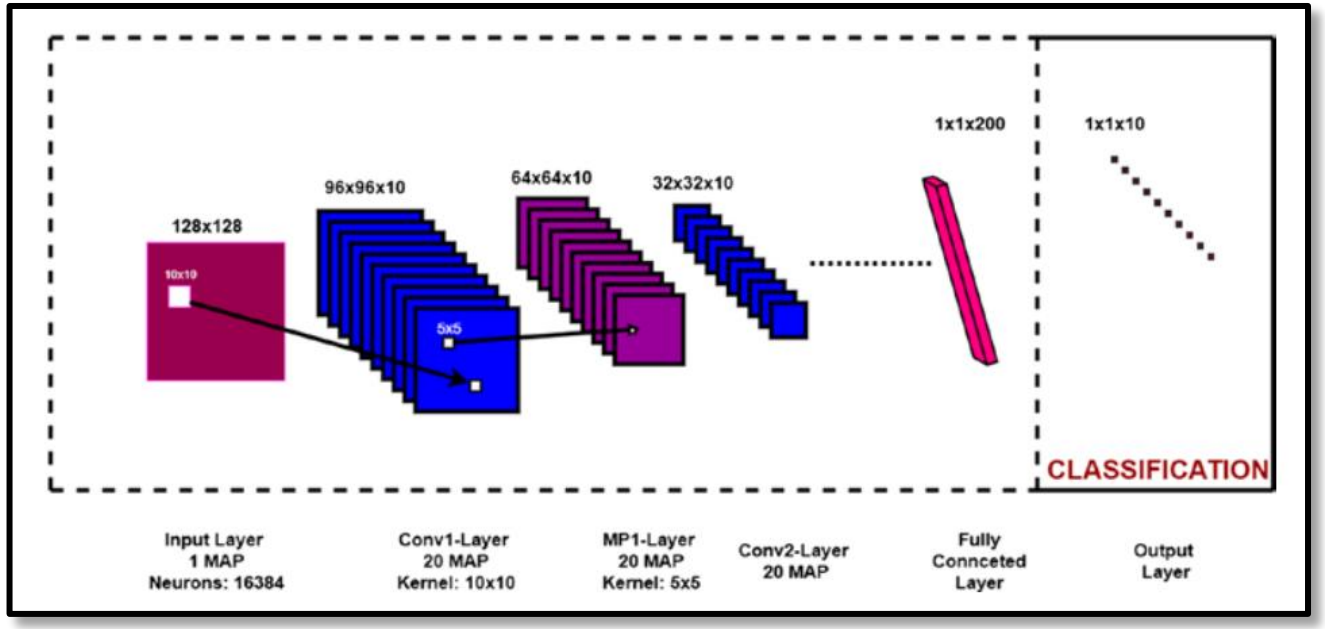


Source: ch.mathworks.com

Convolutional Neural Networks

Now that we have covered the necessary concepts, we can understand what Convolutional Neural Networks (CNNs) are and how they work. CNNs consist of different layers, similar to Artificial Neural Networks (ANNs). In each layer of a CNN, we apply various filters. The number and size of these filters are considered hyperparameters, while the weights within these filters are the parameters. The goal in CNNs is to optimize these weights and biases.

The image will be convolved with different filters, allowing the network to extract various features from the image. After all the convolutional layers, we will flatten the output and add a fully connected layer on top. Ultimately, we will arrive at our output layer.

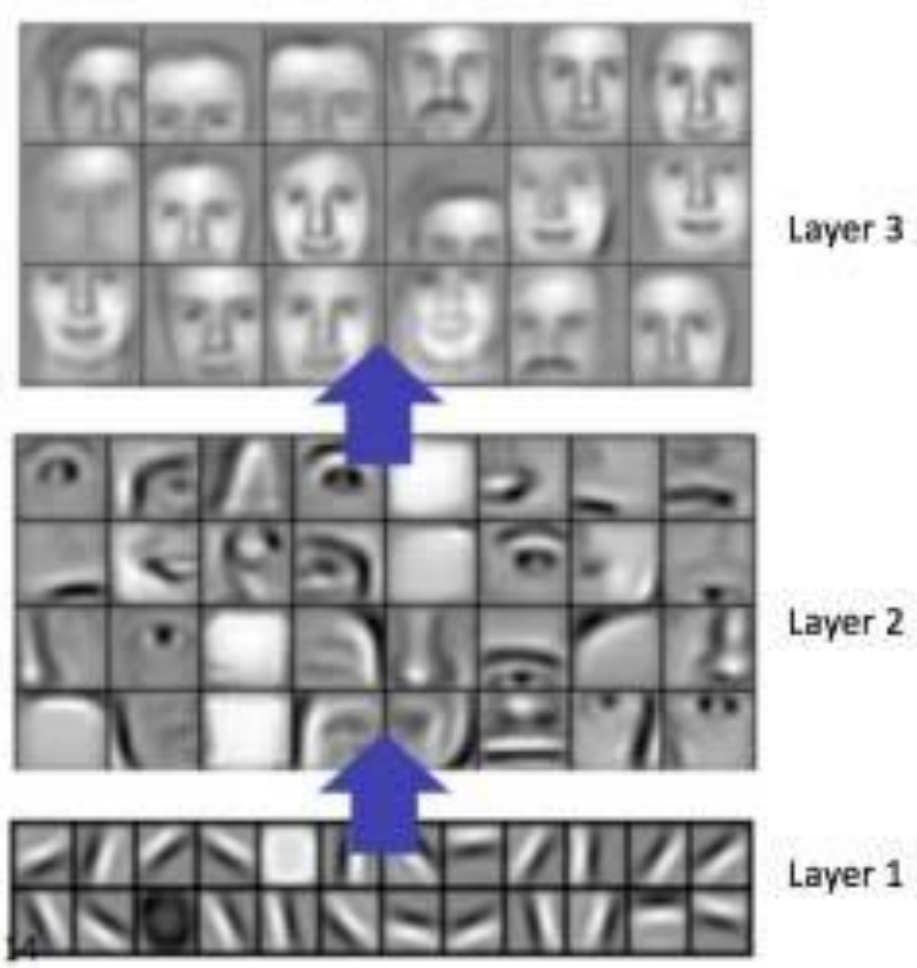


Filters as feature extractors

It is helpful to think of each filter in a CNN as a feature extractor. The filters in the early layers of the CNN will extract basic features such as edges. In contrast, the filters in the later layers of the CNN will extract more sophisticated and complicated features.

In this example, we can see the result of the filters of a CNN after being trained on a dataset of human face portraits.

We observe that the filters in the first layer have extracted simple edges. In the middle layer, the filters have extracted different parts of a human face, like the nose, ears, and eyes. Finally, in the last layer, the model has extracted complete faces with great detail.

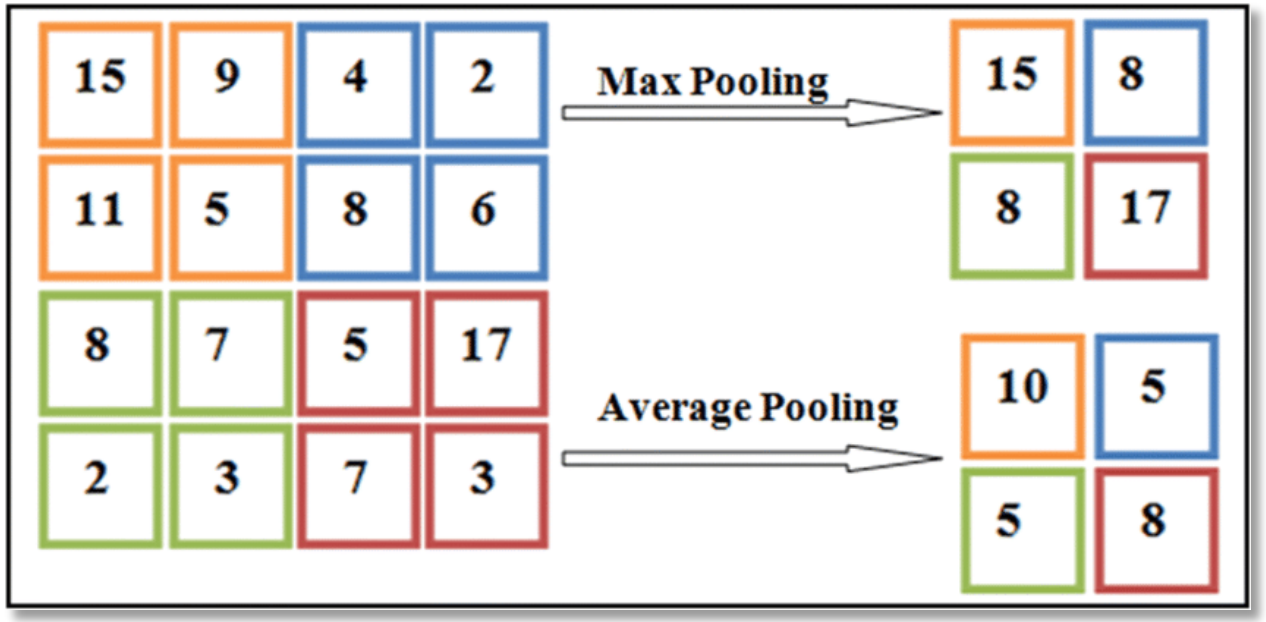


Source: www.researchgate.net

Max pooling and Average pooling

In this section, we will examine another component of CNNs. Earlier, it was mentioned that CNNs use fewer parameters compared to ANNs. One of the components that reduces the dimensions and, consequently, the parameters in CNNs is pooling layers.
After applying several convolutional layers, we insert a pooling layer to capture the important features and reduce the dimensions. The two most popular pooling methods are max pooling and average pooling. In both methods, we use a kernel that we slide through the matrix. In max pooling, we select the maximum value within the kernel as the output, while in average pooling, we take the average of all values in the kernel.

In this example, we are going to use a 2*2 kernel and we set the stride to 2. This will divide the height and the width of the matrix by 2.



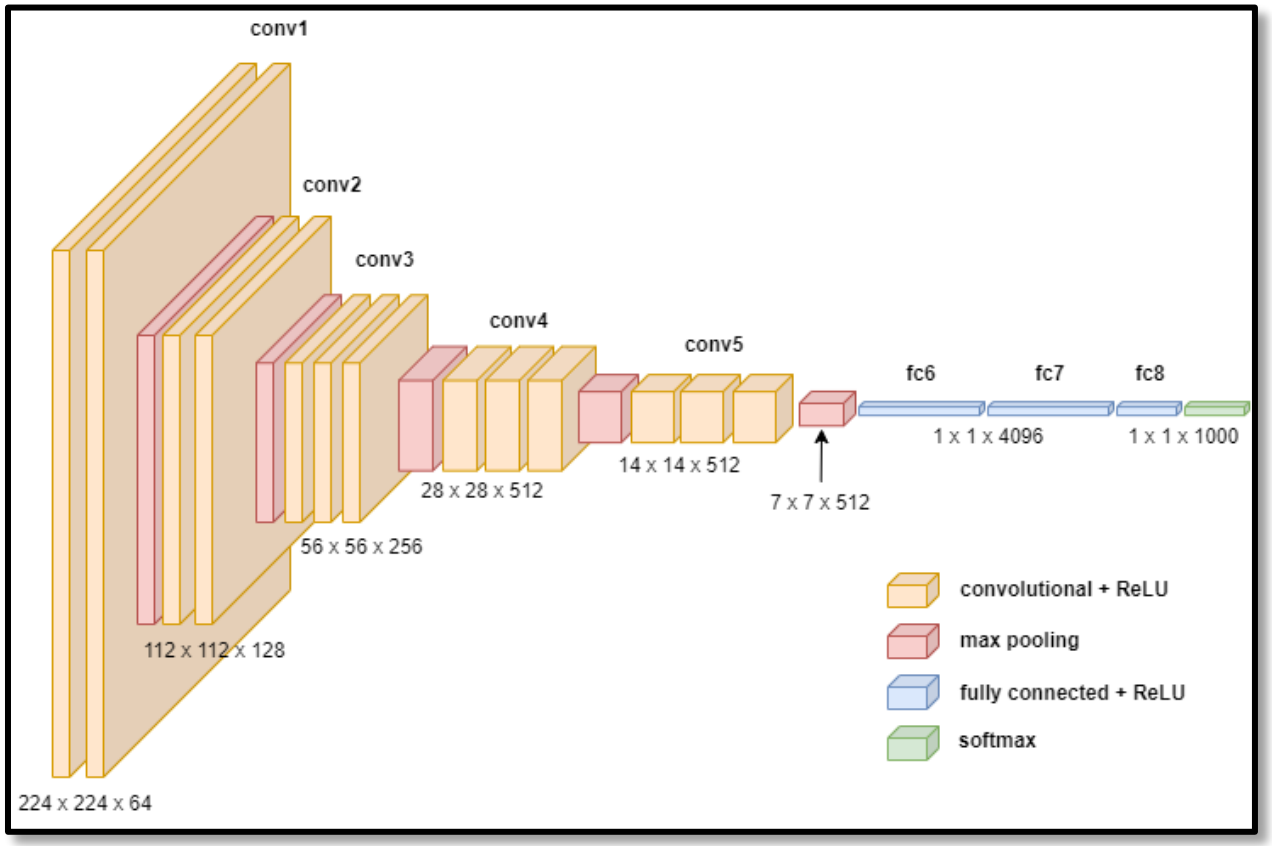
Source: www.researchgate.net

Complete CNN architecture example

Now, it is time to examine a complete CNN architecture and analyze its components. In this example, we start with a 224x224 picture. We then add a convolutional layer, followed by max pooling. Next, we add two more convolutional layers, followed by another max pooling operation.

We continue by adding four convolutional layers, followed by max pooling. Then, we add three more convolutional layers, followed by another max pooling step. Finally, we flatten the output.

Next, we add our fully connected layers. At the end, we include a layer with the SoftMax activation function, which will generate our final output.



Complete CNN architecture example

We have referred to the reduction of the number of parameters several times. In this section, we will explore how to calculate the number of parameters in a convolutional layer.
If the input size of the conv layer is $(W_1 \times H_1 \times D_1)$, and the filter size is (F_W, F_H). For each filter, we have $D_1 \times F_W \times F_H$ weights. We will also have a bias for each filter. So in total, if we have N filters, the total number of parameters will be:

$$total\ parameters = (D_1 \times F_W \times F_H + 1) \times N$$

It is important to note that max pooling layers do not have any parameters.