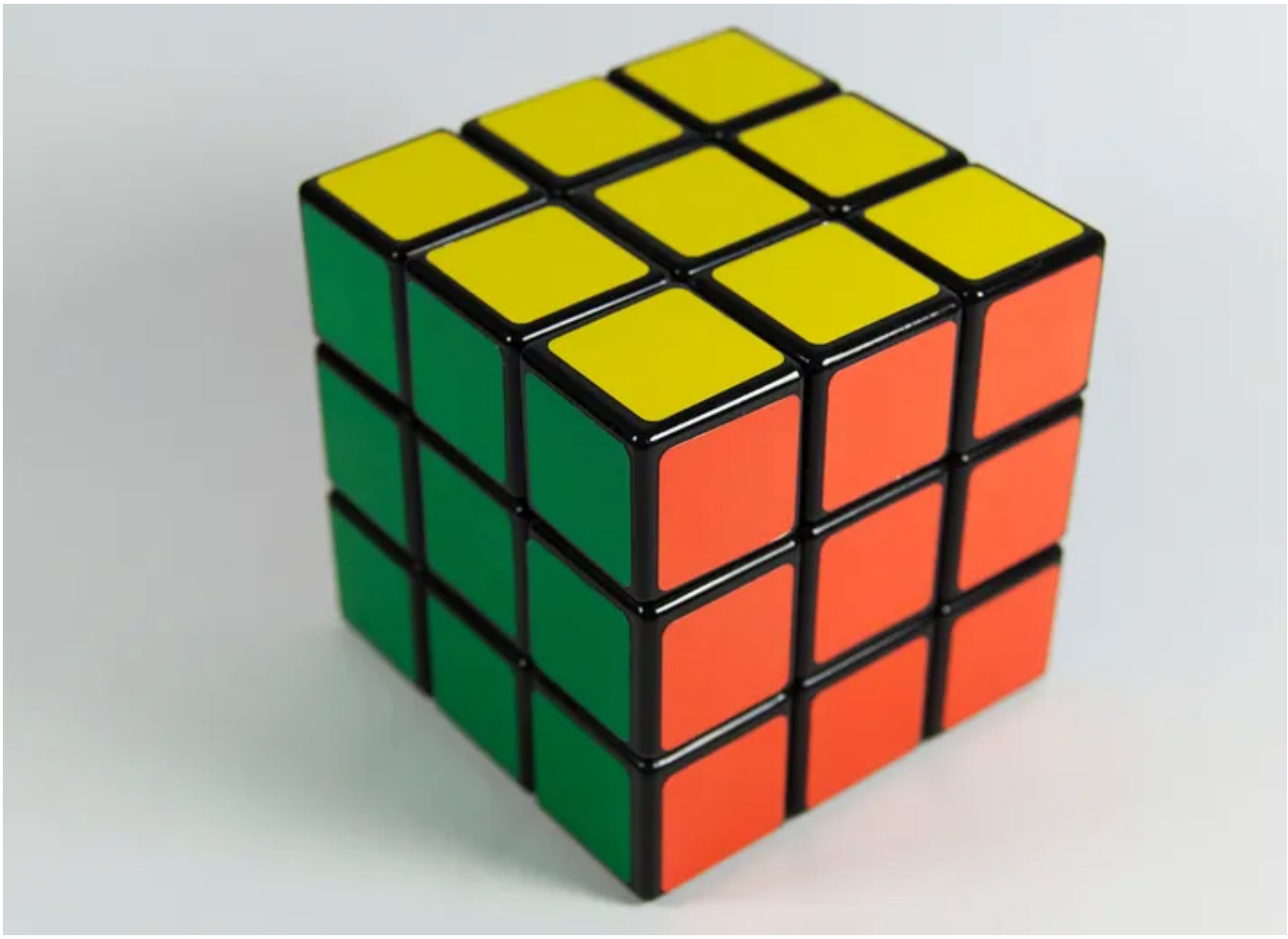
Published in Towards Data Science

Ismail Mebsout Follow

Feb 24, 2020 · 10 min read · ✦ · Listen



Picture of Miguel Á. Padriñán from Pexels

Convolutional Neural Networks' mathematics

Convolutional Neural Networks - Part 1: Deep dive into the fundamentals of CNNs.

Computer vision is a subfield of deep learning which deals with images on all scales. It allows the computer to process and understand the content of a large

number of pictures through an automatic process.
The main architecture behind Computer vision is the convolutional neural network which is a derivative of feedforward neural networks. Its applications are very various such as image classification, object detection, neural style transfer, face identification,... If you have no background on deep learning in general, I recommend you to first read my post about feedforward neural networks.

NB: Since Medium does not support LaTeX, the mathematical expressions are inserted as images. Hence, I advise you to turn the dark mode off for a better reading experience.

Table of content

1. Filter processing
2. Definition
3. Foundations
4. Training the CNN
5. Common architectures

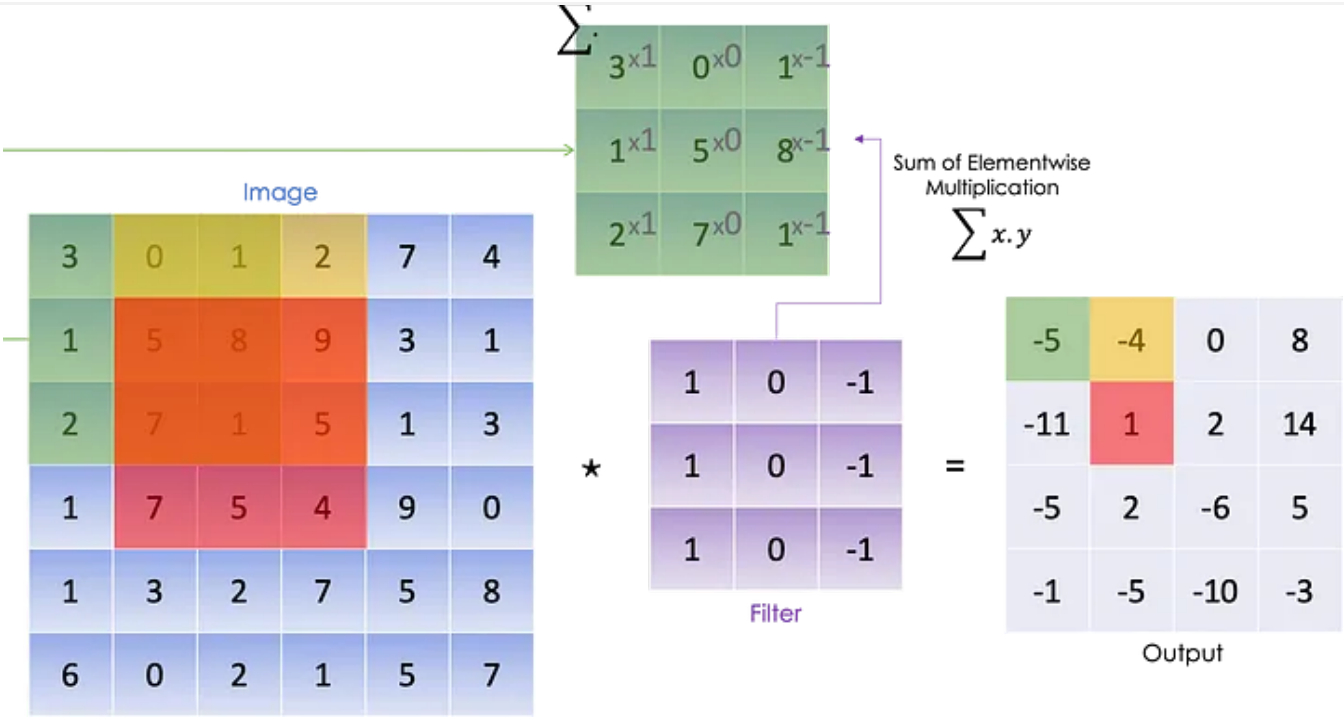
1- Filter processing

The first processing of images was based on filters that allowed, for instance, to get the edges of an object in an image using the combination of vertical-edge and horizontal-edge filters.
Mathematically speaking, the vertical edge filter, VEF, if defined as follows:

$$VEF = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = {}^{T}HEF$$

Where HEF stands for the horizontal edge filter.

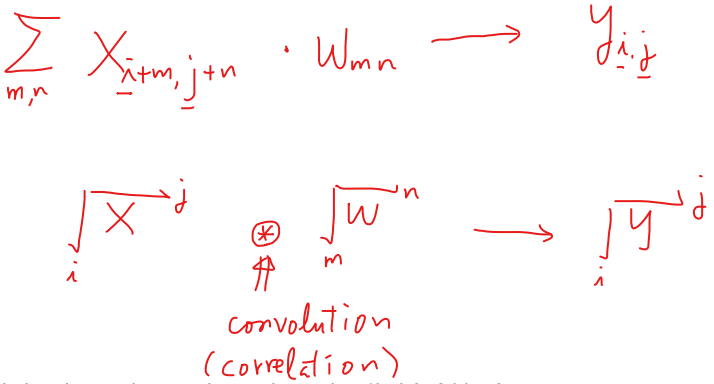
For the sake of simplicity, we consider grayscale 6x6 image A, a 2D matrix where the value of each element represents the amount of light in the corresponding pixel.



We carry out the elementwise multiplication on the first 3x3 block of the image then we consider the following block on the right and do the same thing until we have covered all the potential blocks.

We can sum up the following process in:

Given this example, we can think of using the same process for `any objective` where the `filter is learned` by `neural network` as follows:



The main intuition is to set a neural network that takes the image as an input and outputs a defined target. The parameters are learned using backpropagation.

2- Definition

A convolutional neural network is a serie of convolutional and pooling layers which allow extracting the main features from the images responding the best to the final objective.

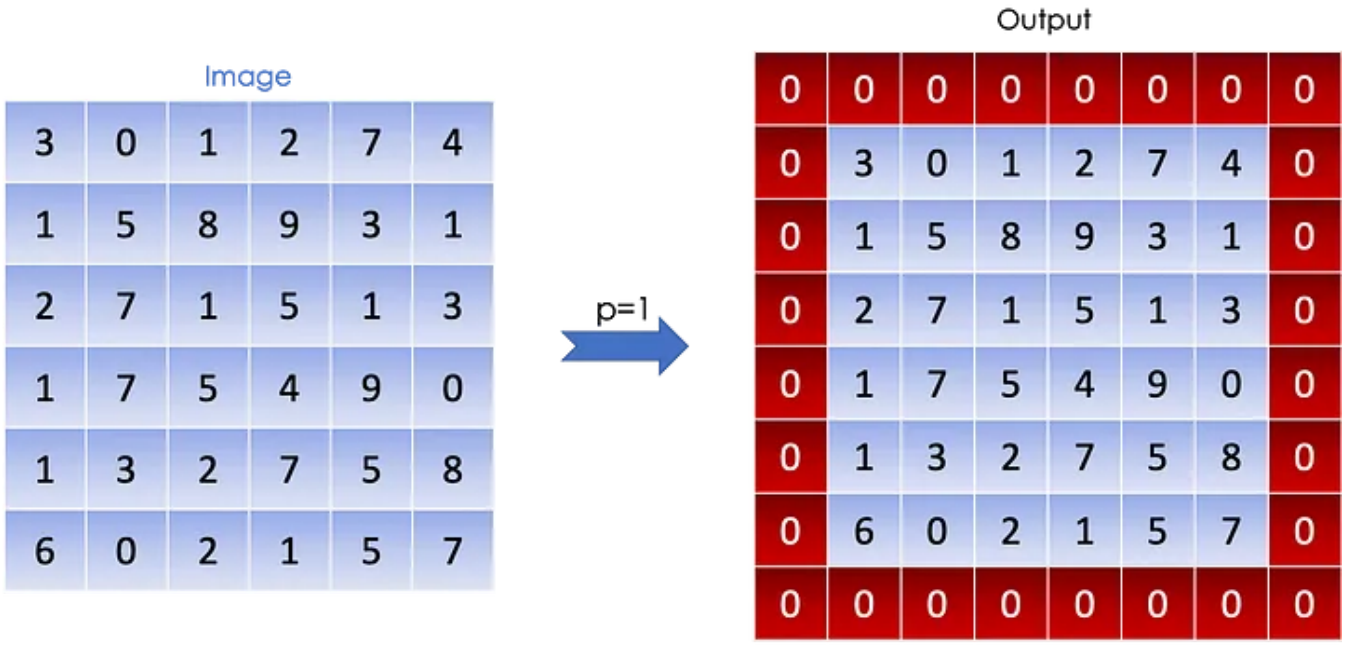
In the following section, we will detail each brick along with its mathematical equations.

Convolution product

Before we explicitly define the convolution product, we will first start by defining some basic operations such as the padding and the stride.

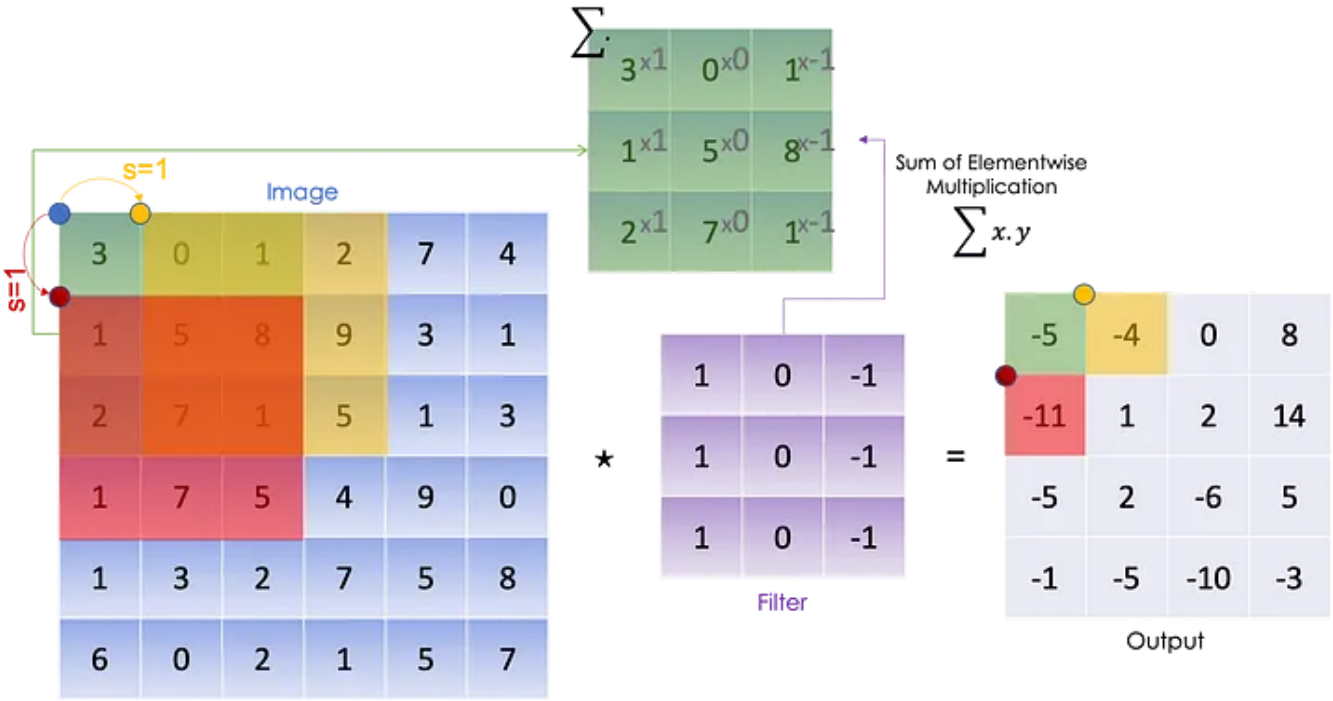
Padding

As we have seen in the convolutional product using the vertical-edge filter, the pixels on the corner of the image (2D matrix) are less used than the pixels in the middle of the picture which means that the information from the edges is thrown away.
To solve this problem, we often add padding around the image in order to take the pixels on the edges into account. In convention, we `padde` with `zeros` and denote with `p` the padding parameter which represents the number of elements added on each of the four sides of the image.
The following picture illustrates the padding of a grayscale image (2D matrix) where `p=1`:



Stride

The stride is the step taken in the convolutional product. A large stride allows to shrink the size of the output and vice-versa. We denote `s` the stride parameter. The following image illustrates a convolutional product (sum of element-wise element per block) with `s=1` :



Convolution

Once we have defined the stride and the padding we can define the convolution product between a tensor and a filter.

After previously defining the convolution product on a 2D matrix which is the sum of the `element-wise product`, we can now formally define the convolution product on a volume.
An image, in general, can be mathematically represented as a tensor with the following dimensions:

$$dim(image) = (n_H, n_W, n_C)$$

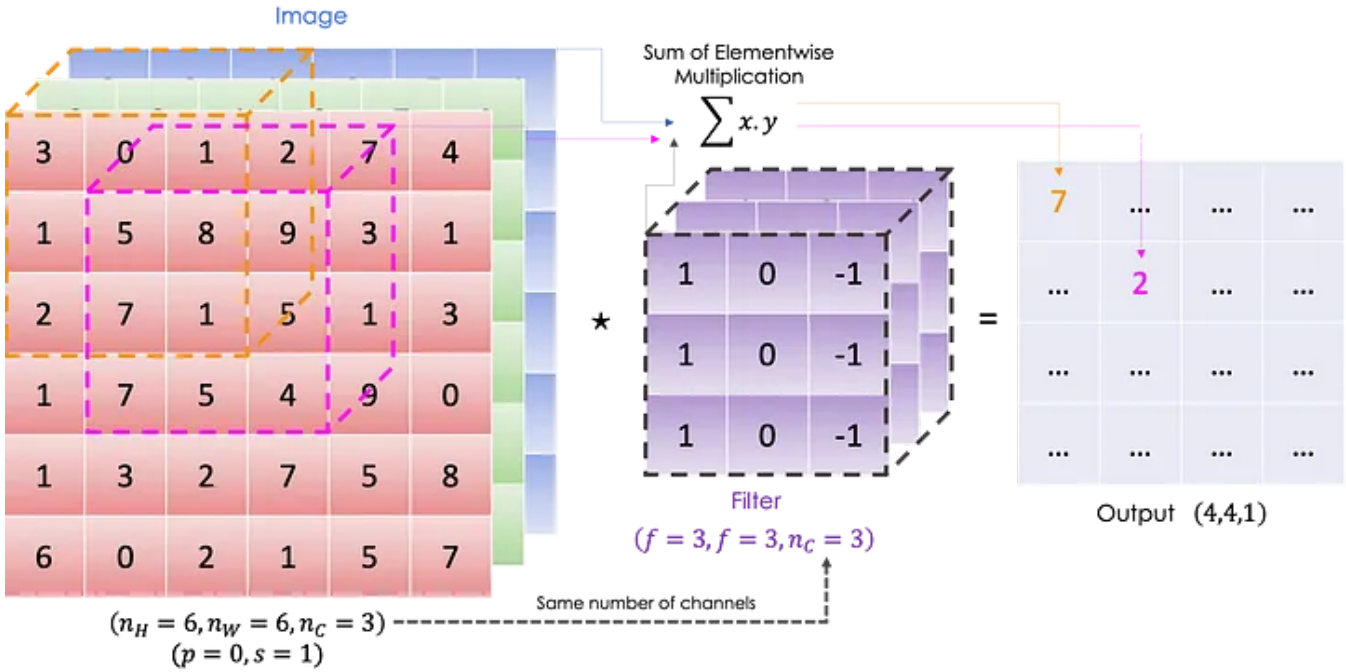
where:

- n_H: the size of the Height
- n_W: the size of the Width
- n_C: the number of Channels

In the case of an RGB image, for instance, n_C=3, we have, Red, Green and Blue. In convention, we consider the filter *K* to be `squared` and to have an `odd dimension` denoted *f,* which allows each pixel to be centered in the filter and thus consider all the elements around it.
When operating the convolutional product, the filter/kernel *K* must have the `same number of channels` as the image, this way we apply a different filter to each channel.
Thus the dimension of the filter is as follows:

$$dim(filter) = (f, f, n_C)$$

The `convolutional product` between the image and the filter is a `2D matrix` where each element is the **sum of the elementwise multiplication** of the cube (filter) and the subcube of the given image as illustrated below:



Mathematically speaking, for a given image and filter we have:

$$conv(I,K)_{x,y} = \sum_{i=1}^{n_H} \sum_{j=1}^{n_W} \sum_{k=1}^{n_C} K_{i,j,k} I_{x+i-1,y+j-1,k}$$

Keeping the same notations as before, we have:

$$\begin{aligned} dim(conv(I,K)) &= (\left\lfloor \frac{n_H+2p-f}{s} + 1 \right\rfloor, \left\lfloor \frac{n_W+2p-f}{s} + 1 \right\rfloor); s > 0 \\ &= (n_H + 2p - f, n_W + 2p - f); s = 0 \end{aligned}$$

where $\lfloor x \rfloor$ is the floor function of x.
There are some special types of convolution:

- `Valid convolution` : $p = 0$
- `Same convolution` : output size=input size $\rightarrow p = \frac{f-1}{2}$
- `1x1 convolution` : $f = 1$, it might be useful in same cases to shrink the number of channels n_C without changing the other dimensions (n_H, n_W).
  In the example bellow we filled the filter with numbers for the sake of illustration, in a convolutional neural network, the $f * f * n_C$ filter's parameters are `learned` through backpropagation.

Pooling

It is the step of downsampling the image's features through summing up the information. The operation is carried out through each channel and thus it only affects the dimensions (n_H, n_W)and keeps n_C intact.

Given an image, we slide a filter, with `no parameters` to learn, following a certain stride, and we apply a function on the selected elements. We have:

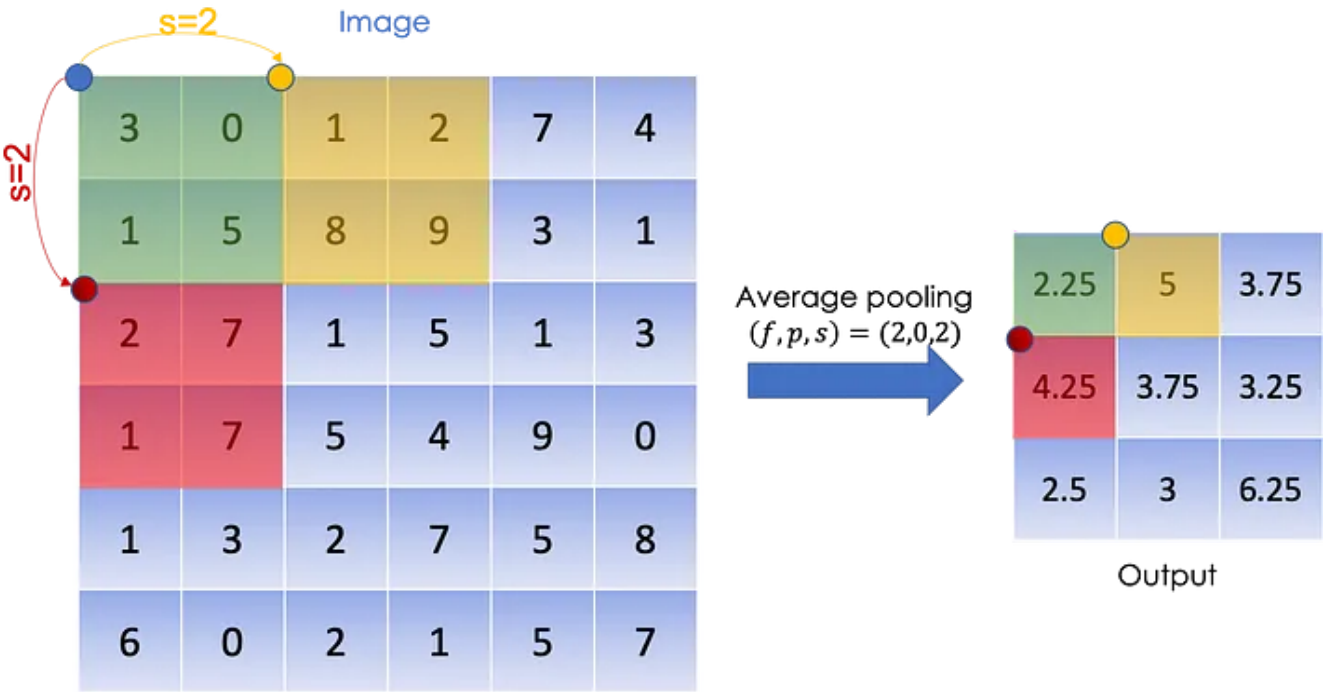
$$dim(pooling(image)) = (\left\lfloor \frac{n_H+2p-f}{s} + 1 \right\rfloor, \left\lfloor \frac{n_W+2p-f}{s} + 1 \right\rfloor, \mathbf{n_C}); s > 0$$
$$= (n_H + 2p - f, n_W + 2p - f, \mathbf{n_C}); s = 0$$

In convention, we consider a squared filter with size f and we usually set f=2 and consider s=2.

We often apply:

- `Average pooling` : we average on the elements present on the filter
- `Max pooling` : given all the elements in the filter, we return the maximum

Bellow, an illustration of an average pooling:



3- Foundations

In this section, we will combine all the operations defined above to construct a convolutional neural network, layer per layer.

One layer of a CNN

Each layer of the convolutional neural network can either be:

- `Convolutional layer -CONV-` followed with an `activation function`
- `Pooling layer -POOL-` as detailed above
- `Fully connected layer -FC-` a layer which is basically similar to one from a feedforward neural network,

You can have more details on the activations functions and the fully connected layer in my previous post.

• Convolutional layer

As we have seen before, at the convolutional layer, we apply convolutional products, using many filters this time, on the input followed by an activation function ψ.

More preciously, at the $\mathbf{l^{th}layer}$, we denote:

- `Input` : $a^{[l-1]}$ with size $(n_H^{[l-1]}, n_W^{[l-1]}, n_C^{[l-1]})$, $a^{[0]}$ being the image in the input
- `Padding` : $p^{[l]}$, `stride` : $s^{[l]}$
- `Number of filters` : $n_C^{[l]}$ where each $K^{(n)}$ has the dimension: $(f^{[l]}, f^{[l]}, n_C^{[l-1]})$
- `Bias` of the n^{th} convolution: $b_n^{[l]}$
- `Activation function` : $\psi^{[l]}$
- `Output` : $a^{[l]}$ with size $(n_H^{[l]}, n_W^{[l]}, n_C^{[l]})$

And we have:

$\forall n \in [1, 2, ..., n_C^{[l]}]$:

$$conv(a^{[l-1]}, K^{(n)})_{x,y} = \psi^{[l]}(\sum_{i=1}^{n_H^{[l-1]}} \sum_{j=1}^{n_W^{[l-1]}} \sum_{k=1}^{n_C^{[l-1]}} K_{i,j,k}^{(n)} a_{x+i-1,y+j-1,k}^{[l-1]} + b_n^{[l]})$$
$$dim(conv(a^{[l-1]}, K^{(n)})) = (n_H^{[l]}, n_W^{[l]})$$

Thus:

$$a^{[l]} =$$
$$[\psi^{[l]}(conv(a^{[l-1]}, K^{(1)})), \psi^{[l]}(conv(a^{[l-1]}, K^{(2)})), ..., \psi^{[l]}(conv(a^{[l-1]}, K^{(n_C^{[l]})}))]$$
$$dim(a^{[l]}) = (n_H^{[l]}, n_W^{[l]}, n_C^{[l]})$$

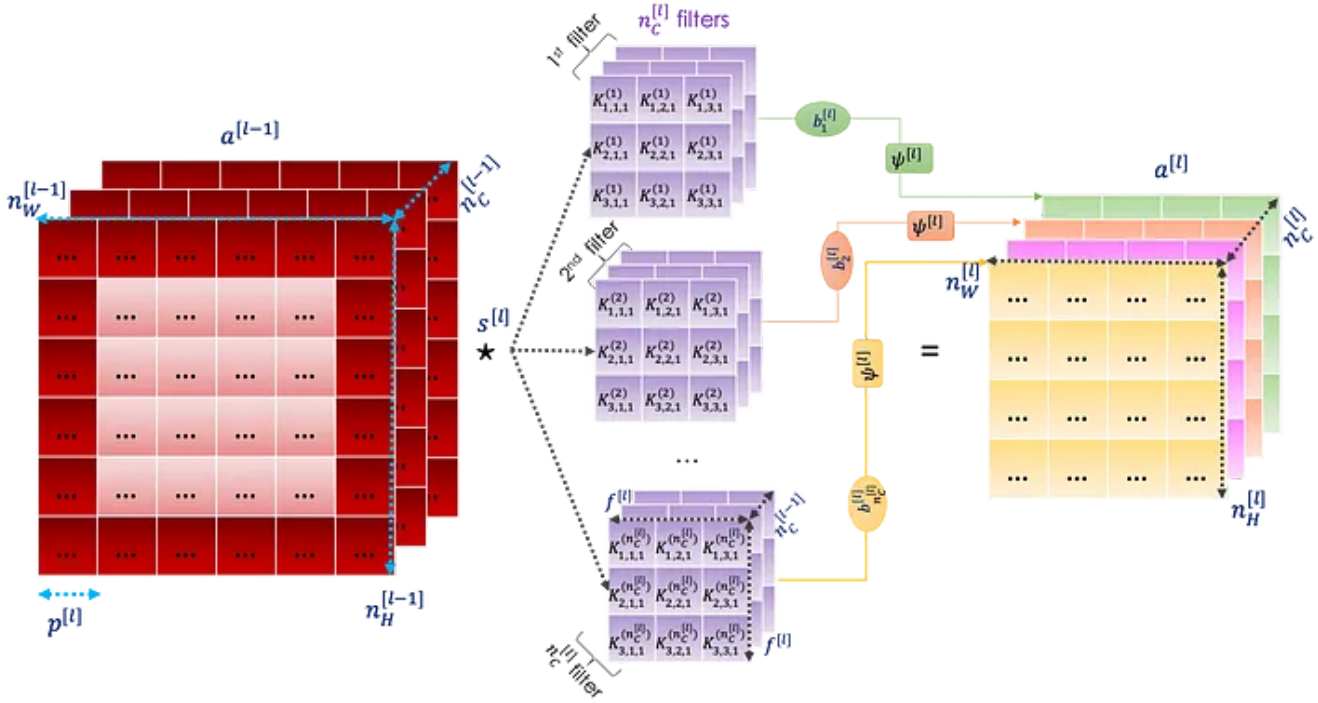
With:

$$n_{H/W}^{[l]} = \left\lfloor \frac{n_{H/W}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor ; s > 0$$
$$= n_{H/W}^{[l-1]} + 2p^{[l]} - f^{[l]}; s = 0$$
$$n_C^{[l]} = number\ of\ filters$$

The `learned parameters` at the l^{th} layer are:

- **Filters** with $(f^{[l]} \times f^{[l]} \times n_C^{[l-1]}) \times n_C^{[l]}$ parameters
- **Bias** with $(1 \times 1 \times 1) \times n_C^{[l]}$ parameters (broadcasting)

We can sum up the convolutional layer in the following graph:



• Pooling layer

As mentioned before, the pooling layer aims at downsampling the features of the input without impacting the number of the channels.
We consider the following notation:

- `Input` : $a^{[l-1]}$ with size $(n_H^{[l-1]}, n_W^{[l-1]}, n_C^{[l-1]})$, $a^{[0]}$ being the image in the input
- `Padding` : $p^{[l]}$(rarely used), `stride` : $s^{[l]}$
- `Size` of the pooling filter: $f^{[l]}$
- `pooling function` : $\phi^{[l]}$
- `Output` : $a^{[l]}$ with size $(n_H^{[l]}, n_W^{[l]}, n_C^{[l]} = n_C^{[l-1]})$

We can assert that:

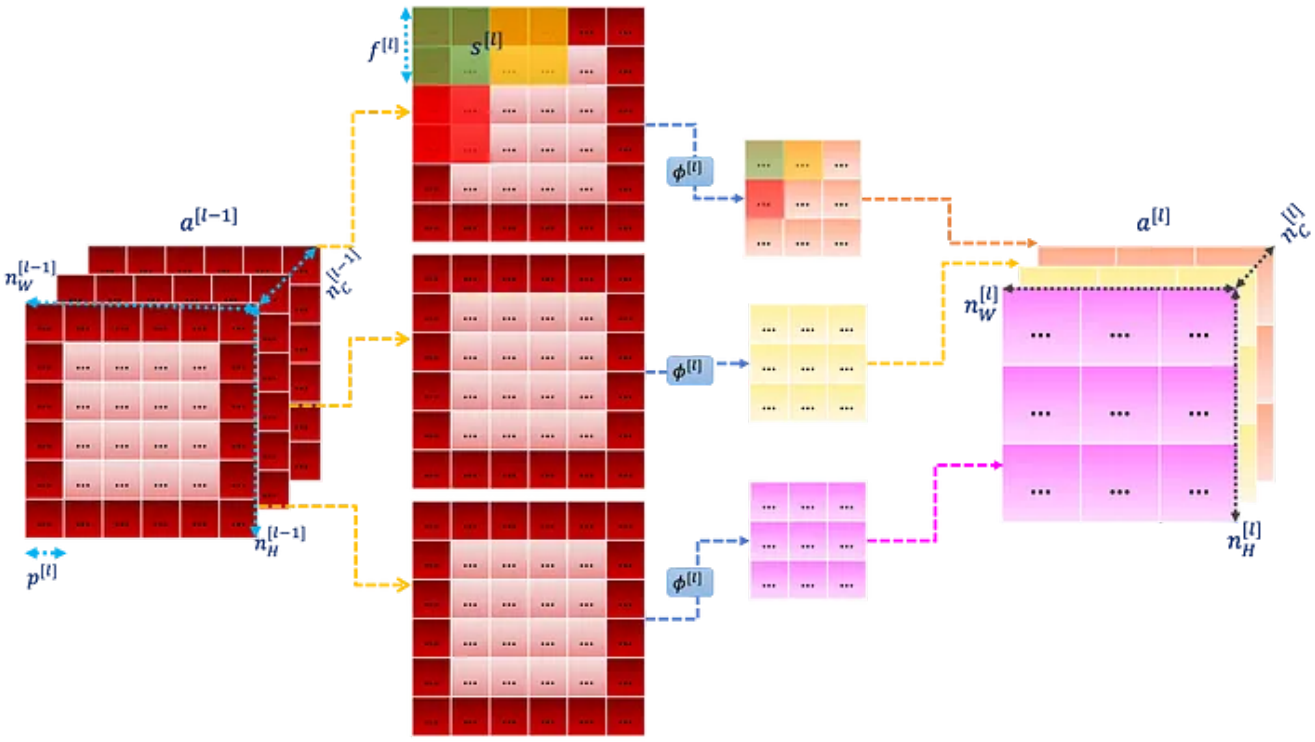
$$a_{x,y,z}^{[l]} = pool(a^{[l-1]})_{x,y,z} = \phi^{[l]}((a_{x+i-1,y+j-1,z}^{[l-1]})_{(i,j)\in[1,2,\dots,f^{[l]}]^2})$$
$$dim(a^{[l]}) = (n_H^{[l]}, n_W^{[l]}, n_C^{[l]})$$

With

$$n_{H/W}^{[l]} = \left\lfloor \frac{n_{H/W}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor ; s > 0$$
$$= n_{H/W}^{[l-1]} + 2p^{[l]} - f^{[l]}; s = 0$$
$$n_C^{[l]} = n_C^{[l-1]}$$

The pooling layer has `no parameters` to learn.

We sum up the previous operations in the following illustration:



• Fully connected layer

A fully connected layer is a finite number of neurons which takes in input a vector and returns another one.

In general, considering the j^{th} node of the i^{th} layer we have the following equations:

$$z_j^{[i]} = \sum_{l=1}^{n_{i-1}} w_{j,l}^{[i]} a_l^{[i-1]} + b_j^{[i]}$$
$$\rightarrow a_j^{[i]} = \psi^{[i]}(z_j^{[i]})$$

The input $a^{[i-1]}$ might be the result of a convolution or a pooling layer with the dimensions $(n_H^{[i-1]}, n_W^{[i-1]}, n_C^{[i-1]})$.
In order to be able to plug it into the fully connected layer we `flatten` the tensor to a `1D vector` having the dimension: $(n_H^{[i-1]} \times n_W^{[i-1]} \times n_C^{[i-1]}, 1)$, thus:

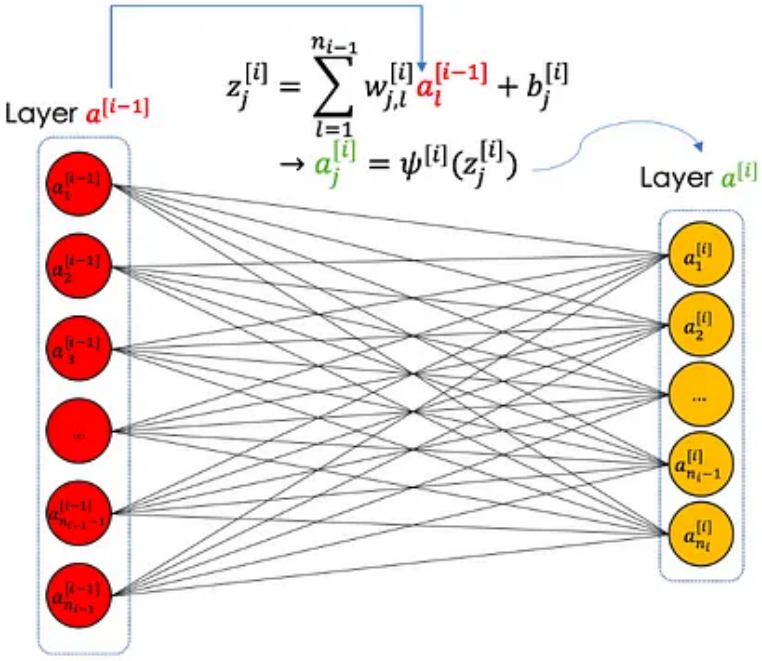
$$n_{i-1} = n_H^{[i-1]} \times n_W^{[i-1]} \times n_C^{[i-1]}$$

The `learned parameters` at the l^{th} layer are:

- **Weights** $w_{j,l}$ with $n_{l-1} \times n_l$ parameters
- **Bias** with n_l parameters

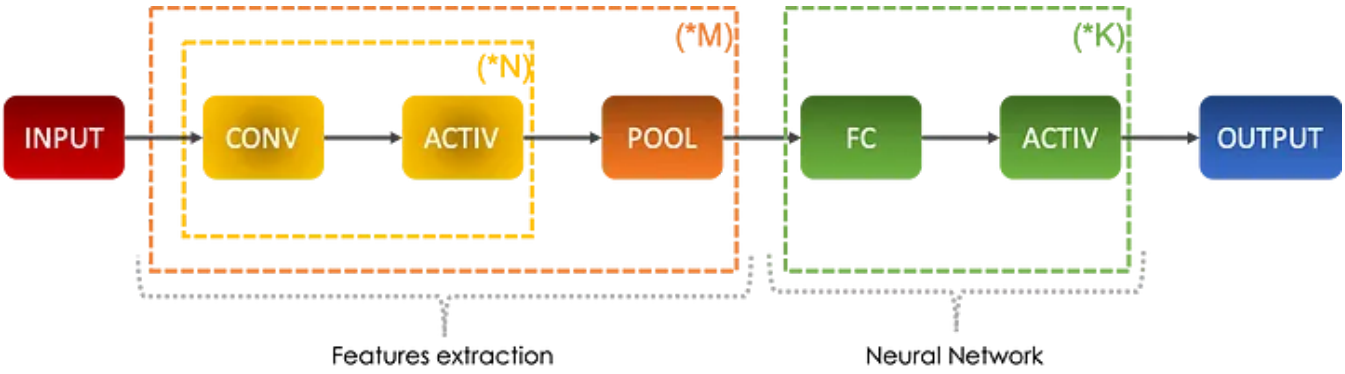
We sum up the fully connected layer in the following illustration:

For more details, you can visit my previous article on feedforward neural networks.



CNN in Overall

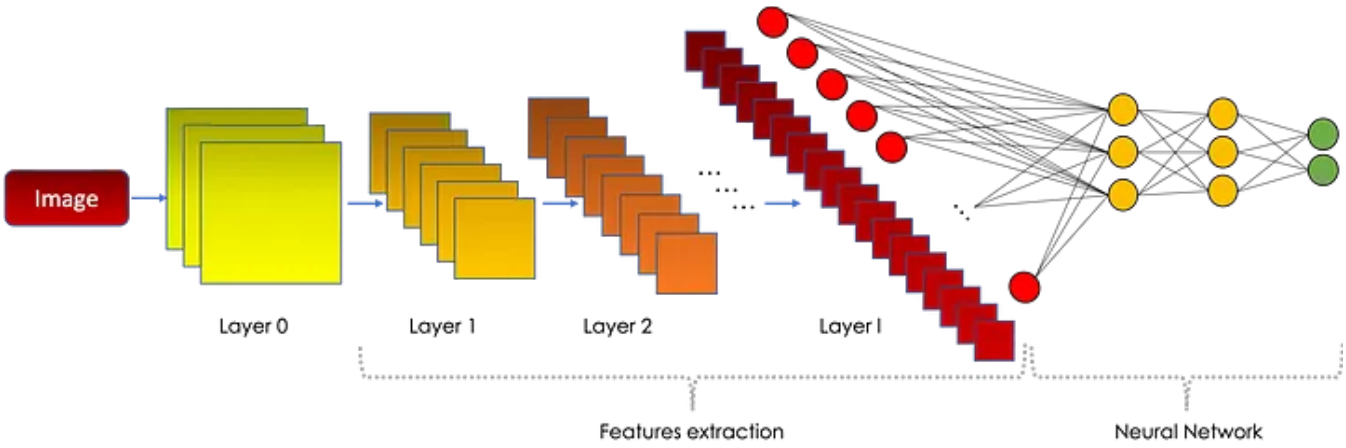
In general, a convolutional neural network is a serie of all the operations described above as follows:



After repeating a serie of convolutions followed by activation functions, we apply a pooling and repeat this process a certain number of time. These operations allow to `extract features` from the image that will be `fed` to a `neural network` described by the fully connected layers which are regularly followed by activation functions as well.

The main idea is to `decrease` n_H & n_W and `increase` n_C when going deeper through the network.

In 3D, a convolutional neural network has the following shape:



Why do CNN work efficiently?

Convolutional neural networks enable the state of the art results in image processing for two main reasons:

- **Parameter sharing:** a feature detector in the convolutional layer which is useful in one part of the image, might be useful in other ones
- **Sparsity of connections:** in each layer, each output value depends only on a small number of inputs

4- Training the CNN

Convolutional neural networks are trained on a set of labeled images. Starting from a given image, we propagate it through the different layers of the CNN and return the sought output.
In this chapter, we will go through the learning algorithm along with the different techniques used in the data augmentation.

Data preprocessing

Data augmentation is the step of increasing the number of images in a given dataset. There are many techniques used in data augmentation such as:

- `Crooping`
- `Rotation`
- `Flipping`

- `Noise injection`
- `Color space transformation`

It enables `better learning` due to the bigger size of the training set and allows the algorithm to learn from different `conditions` of the object in question.
Once the dataset is ready, we **split** it into three parts like any machine learning project:

- **Train set**: used to train the algorithm and construct batches
- **Dev set**: used to finetune the algorithm and evaluate bias and variance
- **Test set**: used to generalize the error/precision of the final algorithm

Learning algorithm

Convolutional neural networks are a special kind of neural networks specialized in images. Learning in neural networks, in general, is the step of calculating the weights of the parameters defined above in several layers.
In other words, we aim to find the best parameters that give the best prediction/approximation, starting from the input image, of the real value.
For this, we define an objective function called the `loss function` and denoted `J` which quantifies the distance between the real and the predicted values on the overall training set.
We minimize J following two major steps:

- **`Forward Propagation`** : we propagate the data through the network either in entirely or in batches, and we calculate the loss function on this batch which is nothing but the sum of the errors committed at the predicted output for the different rows.
- **`Backpropagation`** : consists of calculating the gradients of the cost function with respect to the different parameters, then apply a descent algorithm to update them.

We iter the same process a number of times called `epoch number` . After defining the architecture, the learning algorithm is written as follows:

- Initialization of the model parameters, a step equivalent to injecting noise into the model.
- `For` i=1,2...N: (N is the number of epochs)
  - Perform `forward propagation` :
    - $\forall i$, Compute the predicted value of x_i through the neural network: $\hat{y}_i^\theta$
    - Evaluate the function : $J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}_i^\theta, y_i)$
      where m is the size of the training set, θ the model parameters and $\mathcal{L}$ the cost$^{(*)}$ function
  - Perform `backpropagation` :
    - Apply a descent method to update the parameters :

$$\theta =: G(\theta)$$

(*) The cost function evaluates the distances between the real and predicted value on a single point.

For more details, you can visit my previous article on feedforward neural networks.

5- Common architectures

Resnet

A Resnet, short cut or a skip connection is a convolutional layer which takes into account the layer *n-2* at the layer n . The intuition comes from the fact that when neural networks get very deep, the accuracy at the output becomes very stable and does not increase. Injecting residuals from the previous layer help solve this problem.
Let's consider a residual block, when the `skip connection` is `off` , we have the following equations:

$$z_j^{[i]} = \sum_{l=1}^{n_{i-1}} w_{j,l}^{[i]} a_l^{[i-1]} + b_j^{[i]}$$
$$\rightarrow a_j^{[i]} = \psi^{[i]}(z_j^{[i]})$$

In the case where the skip connection is `on`, we have:

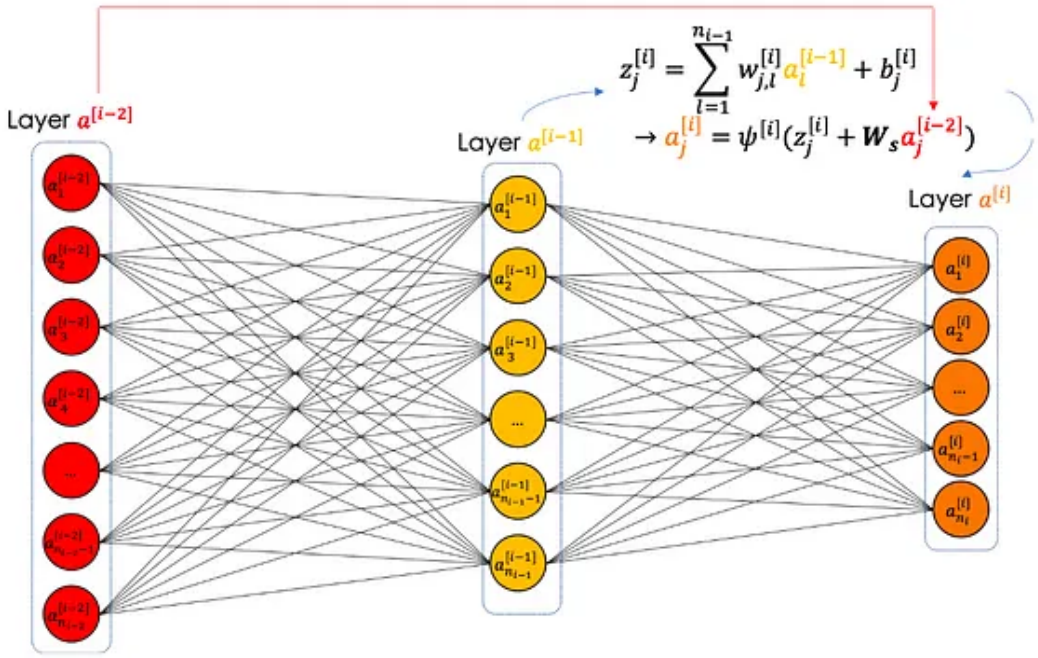
$$z_j^{[i]} = \sum_{l=1}^{n_{i-1}} w_{j,l}^{[i]} a_l^{[i-1]} + b_j^{[i]}$$
$$\rightarrow a_j^{[i]} = \psi^{[i]}(z_j^{[i]} + \mathbf{a_j^{[i-2]}})$$

With the right choice of the activation function $\psi^{[i]}$ and $w^{[i]} = 0, b^{[i]} = 0$, we can have: $a^{[i]} = a^{[i-2]}$, the residual block is capable of learning the identity function and thus it does not harm the neural network.
We usually need $a^{[i]}$ and $a^{[i-2]}$ to have the same shape so we often use `same convolution`. If not, we set:

$$\rightarrow a^{[i]} = \psi^{[i]}(z^{[i]} + \mathbf{W_s} a^{[i-2]})$$
$$dim(W_s) = [n^{[i]}, n^{[i-2]}]$$

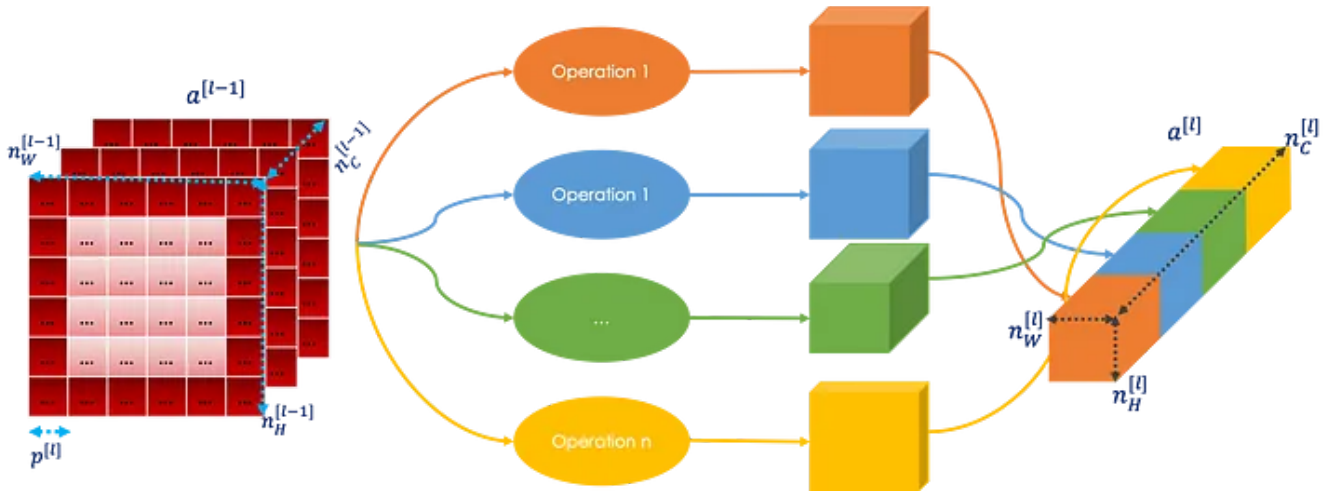
Where W_s might be a fixed tensor or a learned one.

We can sum up the residual block in the following illustration:



Inception Networks

When designing a convolutional neural network, we often have to choose the type of the layer: `CONV`, `POOL` or `FC`. The inception layer does them all. The result of all the operations is then `concatenated` in a single block which will be the input of the next layer as follows:



It is important to note that the inception layer raises the problem of `computational cost`. For information, the name `inception` comes from the movie!

Conclusion

In the first part of this article, we have seen the fundamentals of CNN from convolutional products, pooling/fully connected layers to the training algorithm.

In the second part, we will discuss some of the most famous architectures used in image processing.

Do not hesitate to check my previous article dealing with:

- Deep Learning's mathematics
- Object detection & Face recognition algorithms
- Recurrent Neural Networks

References

- Deep Learning Specialization, Coursera, Andrew Ng
- Machine Learning, Loria, Christophe Cerisara

Originally published at https://www.ismailmebsout.com on February 24, 2020.

Convolutional Network Mathematics Data Science Image Processing

Deep Learning