Convolutional Layers 2

Convolutional Layers vs Linear Layers

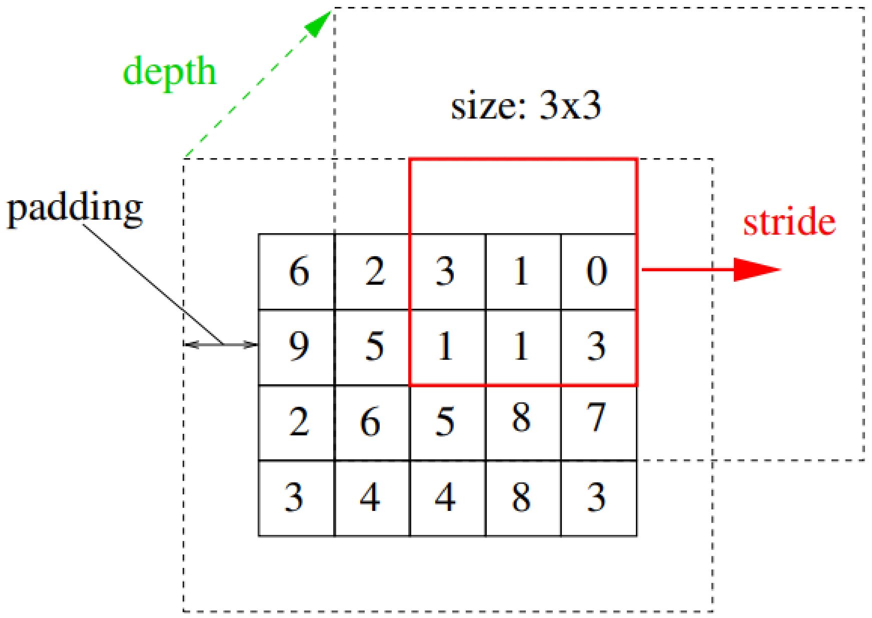
As we know, the kernel is a small window of weights, and applies these windows or weights to make a weighted combination of the input image. The computation performed by the convolutional layer is the same as the computation made by the linear layer, since we are still performing a *linear combinaiton.* - The real difference is that we are just considering a *small portion* of the inputs for the layer, and not all of them.

Structure of a CNN Layer

A convolutional layer is defined by the following parameters - **kernel size**: the *dimension* of the linear filter. - A **stride** is the *movement* we want for the kernel along the x or y axis. The minimum is 1, but you can also define a bigger stride. - With an *higher stride*, we have less overlap and the dimension of the output gets smaller (obv, we have more overlap with a bigger size). - The stride also influences the size of the output, since we are basically skipping cells. - **Padding**: Artificial *enlargement* of the input to allow *the application of filters on borders.* - i.e. zero-padding, the borders are filled with black pixels, or we can repeat the last values. - **depth**: number of different kernels that we wish to syntesize. Each *kernel* will produce a *different feature map* with a small spatial dimension. - Can be seen as the number of channel that we have in the layer in which we want to apply the convolution.

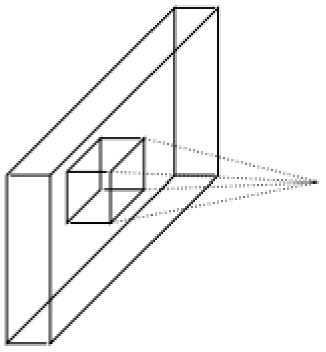
So, each CNN layer usually has 4 dimensions: - batch dimension - width - heght - depth

What do we do when our layer has more than 1 channel? In this case, ==the kernel operates on all the channels together==.
For this reason, a kernel of 1x1 operates just on the channels.



Reducing channels If in a layer we reduce the channels, i.e. 64 to 32, our CNN kinda behaves like PCA, since it compresses the 64 channels to 32, preserving the information that is important to us.

Unless stated differently (e.g. in separable convolutions), ==*a filter operates on all input channels in parallel*==. So, if the input layer has depth D, and the kernel size is $N \times M$, the actual dimension of the filter will be $N \times M \times D$. The convolution kernel is tasked with *simultaneously mapping cross-channel*



correlations and spatial correlations.

Dimension of the output

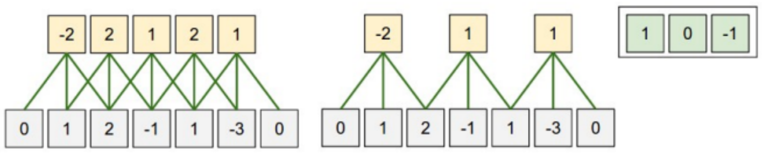
The spatial dimension of each output feature map depends form the spatial dimension of the input, the padding, and the stride. Along *each axes* the dimension of the output is given by the formula:

$$\frac{W+P-K}{S}+1$$

where: - W = dimension of the input - P = padding - K = Kernel size - S = Stride

Example

- The width of the input (gray) is W=7.
- The kernel has dimension K=3 with fixed weights $[1, 0, -1]$



- Padding is zero
- In the first case, the stride is $S = 1$. We get $(W-K)/S+1 = 5$ output values.
- In the second case, the stride is $S = 2$. We get $(W-K)/S+1 = 3$ output values.

Example 2D

- INPUT $[32 \times 32 \times 3]$ color image of 32×32 pixels. The three channels R G B define the input depth
- CONV layer. Suppose we wish to compute 12 filters with kernels 6×6, stride 2 in both directions, and zero padding. Since $(32-6)/2+1 = 14$ the output dimension will be $[14 \times 14 \times 12]$
- RELU layer. Adding an activation layer the output dimension does not change

Padding modes

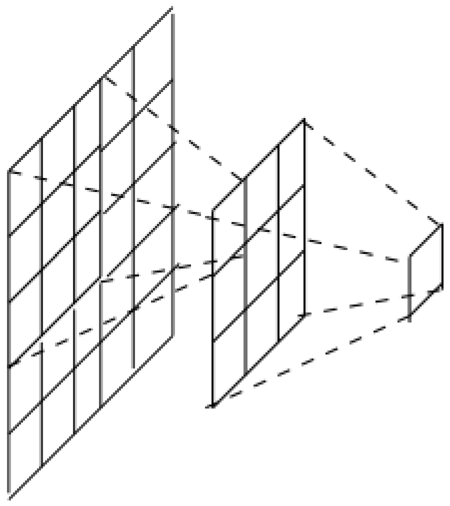
Usually, there are two main "modes" for padding: - *valid*: no padding is applied - *same*: you add a minimal padding enabling the kernel to be applied an integer number of times

This is also true in Keras.

Receptive field

- The **receptive field** of a (deep, hidden) neuron is the *dimension of the input* region *influencing* it.

- It is equal to the dimension of an input image producing (without padding) an output with dimension 1.
- ==A neuron cannot see anything outside its receptive field!==



This notion is only true in CNNs, since in normal, dense NNs we have that each neuron is connected to all the other neurons of the previous layer.

We can compute the dimension of the input (so the receptive field) from the dimensions of the output:

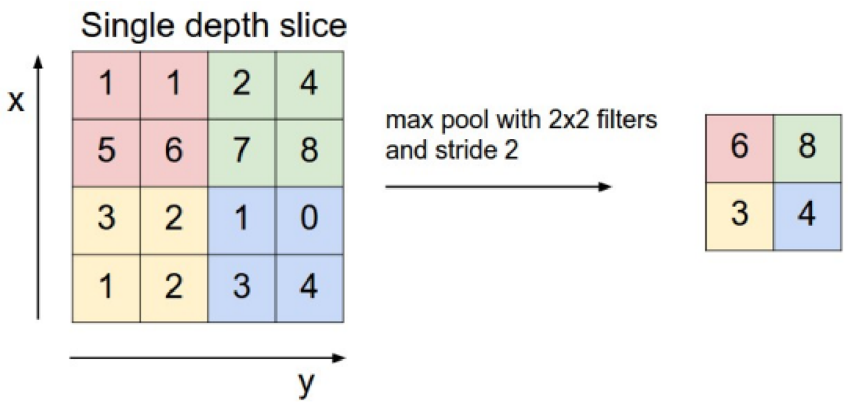
$$\frac{I-K}{S}+1=O \longrightarrow I=\frac{(O-1)\cdot S}{I-K}$$

where I is the dimesion of the input and O is the dimension of the output.

Pooling

Downsampling is a common practice in CNN in order to reduce both computational and spatial cost of the operation. Normally, in CNN, a way to downsample the input is to apply a kernel with stride > 1.

Another way is applying **pooling operation**. In deep convolutional networks, it is common practice to alternate convolutional layers with *pooling layers*, where each neuron simply takes the *mean* or *maximal value* in its *receptive field.* This has a double advantage: - it reduces the dimension of the output - it gives some tolerance to translations:



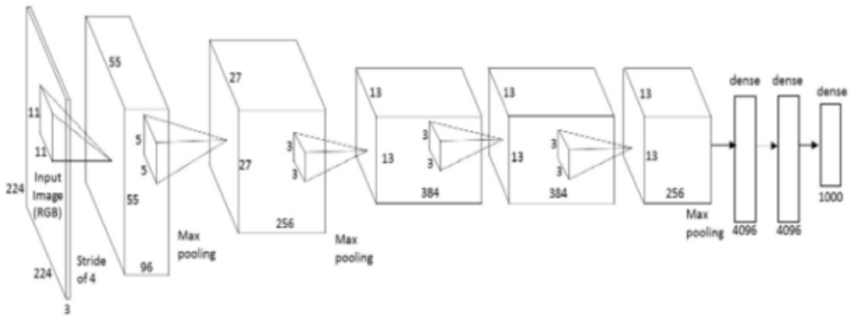
While the mean-pooling operation can be applied through a convolution (same kernel as the blurring kernel), the *max-pooling* can't be expressed through a convolution.

Usually, when downsampling, we are also *doubling* the channel dimention, otherwise the reduction would be too drastic.

Expressiveness of different CNNs kernels Now, a question: we have 2 methods we can use with the same receptive field. One uses a kernel of dimension 5x5, while the other uses 2 kernels of dimension 3x3.

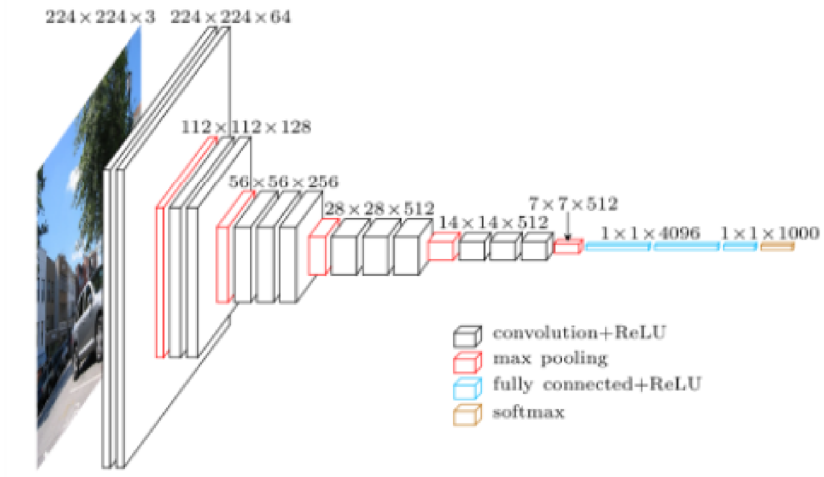
In the case of the kernel 5x5, we have 25+1 parameters. In the case of the 3x3 kernel, we have (9+1)x2 parameters. So, we have less parameters in the second case rather than in the first. We should expect that some transformations which were computable by means of a kernel of dimension 5x5 are not expressible anymore through these 2 kernels of dimensions 3x3. Thus, the first is more expressive than the second. However, we can introduce *non-linear* operations using the 3x3 kernel by introducing non-linearities between the two kernels, which we cannot compute with the single 5x5 kernel. So, which is the more expressive method? We cannot say, the two methods are simply not comparable.

Famous CNNs



AlexNet
This one is interesting since it proves again that we dont a long stack of deep neural layers: at the end of this network, there are just 2 dense layers, and are enough to extract complex features.

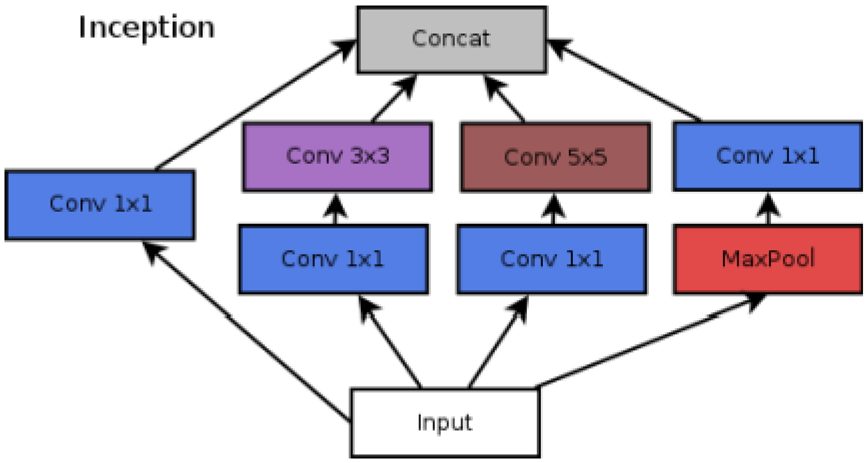
This net also uses very a big kernel (11x11), while now the standard is basically using a 3x3 kernel.



VGG

Inception modules

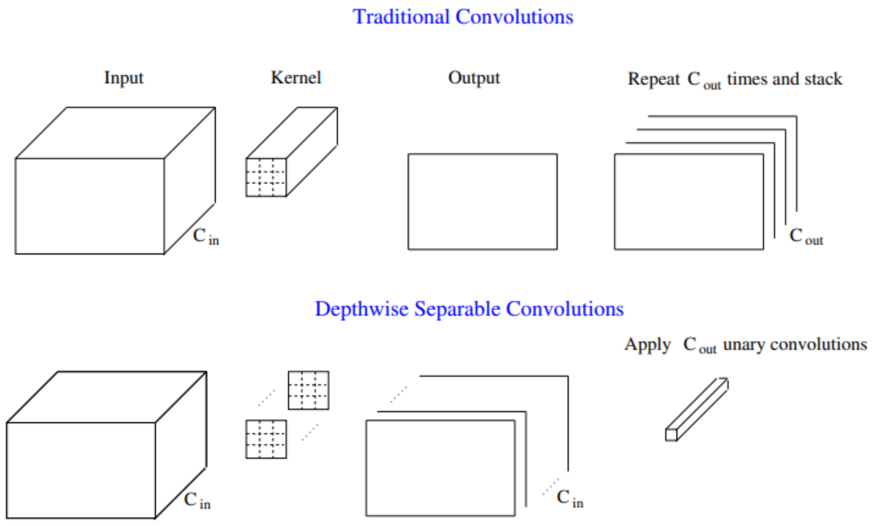
Inception modules re-combine together results and try to resythesize together



new features.
As we can see from this image, you are basically applying some different operations, which are then stacked together along the channel dimension (through concatenation).

Inception hypothesis and Depthwise Separable Convolution

Remember that normal convolutional kernels are 3D, *simultaneously* mapping *cross-channel correlations* and *spatial correlations*. It can be better to decouple them, independently looking for cross-channel correlations (via 1 × 1 convolutions), and spatial 2D convolutions. This operation is illustrated in the following


Traditional Convolutions
Input
Kernel
Output
Repeat C out times and stack
C in
C out
Depthwise Separable Convolutions
Apply C out unary convolutions
C in
C in

picture:

In Deptwise Separable Convolutions, we have a kernel for each channel, which is applied separately. Each kernel produces a single output, so the number of feature maps in input is the same as the number of features map in output. We then reduce the number of feature maps (called C_{out}) through a single unary convolution.

Inception modules can be understood as an *intermediate step* between a regular convolution and a depthwise separable convolution (a depthwise convolution followed by a pointwise convolution).

Traditional vs. Depthwise Separable Convolution Suppose we have a convolutional layer with a 3×3 kernel, 16 input channels and 32 output channels. The input is convolved 32 times with different kernels of dimension $3 \times 3 \times 16 = 144$: we have a total of $32 \times 144 = 4608$ parameters.

In a *depthwise separable convolution* on the same example, we first traverse the 16 channels with a different 3x3 kernel, and then we apply 32 different kernels with dimension $1 \times 1 \times 16$. The total number of parameters is $16 \times 3 \times 3 + 32 \times 1 \times 1 \times 16 = 656$.