1 Convolution Neural Network step-by-step

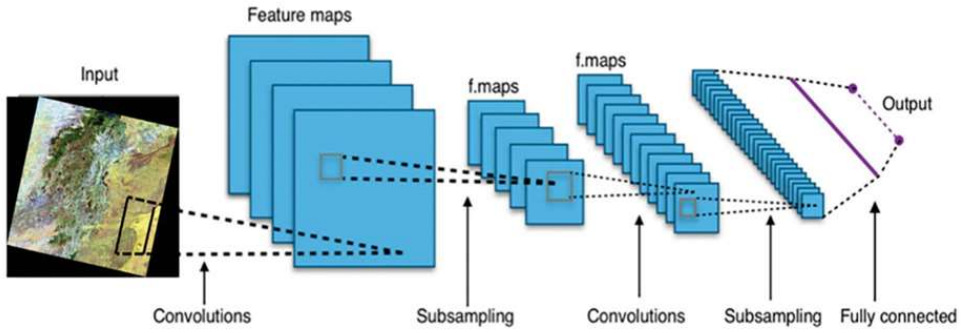
1.1 How CNN work for classification

Convolution Neural Network

is the most native approach for image processing due to:

- **No need for manual feature extraction**.
  - it is hard to formally describe any feature of image, even if you can visually distinguish it.
  - neural network able to extract features in correspondence with its receptive field by its self.
  - CNN use filters to generate invariant features which are passed on to the next layer.

    The features in next layer are convoluted with different filters to generate more invariant and abstract features and the process continues till one gets final feature / output (let say face of X) which is invariant to occlusions.



- The CNN implement sparse interaction which allow to

  reduce the number of connections (and parameters) in CNN in comparison with full-connected one.

  The sparse interaction is carried out using one small-size filter for full input feature map processing.

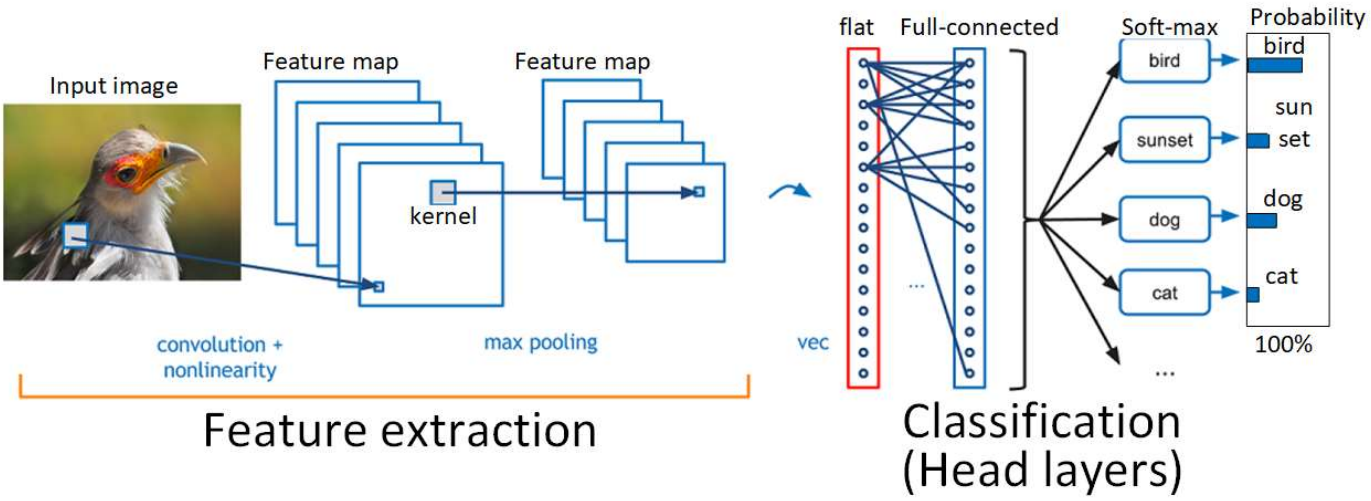
- The CNN imply equivariant representations of feature . The result of the

  CNN independent on the feature position (weakly dependent).

- The CNN explorer the fact (actually assumption) that in image with dimension high enough,

  any region (feature) are contiguous blocks of neighbor pixels.

  Each layer of CNN consist of one of several small-size kernels, which almost sliding through the input matrix.



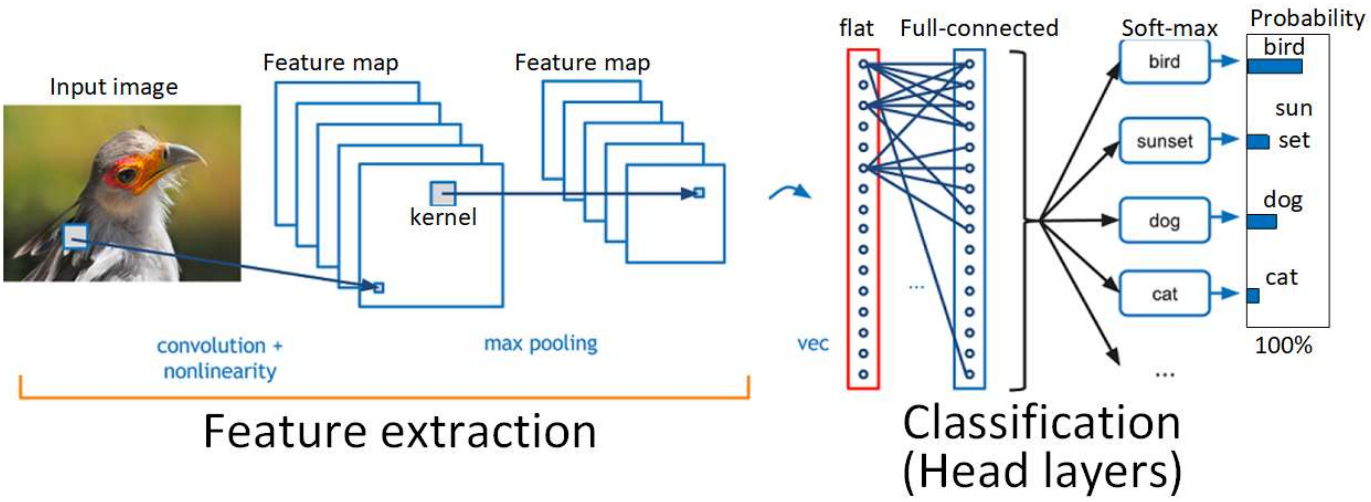
It can be assumed how the simplest CNN (for classification) look-like:

$$\hat{y} = f_2(w \cdot \vec{vec}[\text{pooling}(f_1(k * x))]),$$

where

- $\hat{y}$ is the classification output.
- w is the vector of classification weights.
- pooling is the operation for dimension reduction (subsampling).
- vec is the operation of matrix vectorization (reshape, revel).
- f_1 and f_2 are the **activation functions** .
  - $*$ is the operation of convolution.
  - For input layers f_1 is the frequently any kind of **Rectified Linear Unit (ReLU)** functions (simplest **relu**$(x) = \max(0, x)$).
  - For output classification layers f_2 is the frequently any kind of probability producing functions (often **softmax**$(x_i) = \exp(x_i) / \sum_i \exp(x_i)$).

> *Note* In the basic idea the sequence of operation in the layer is
>
> conv → activation function → pooling



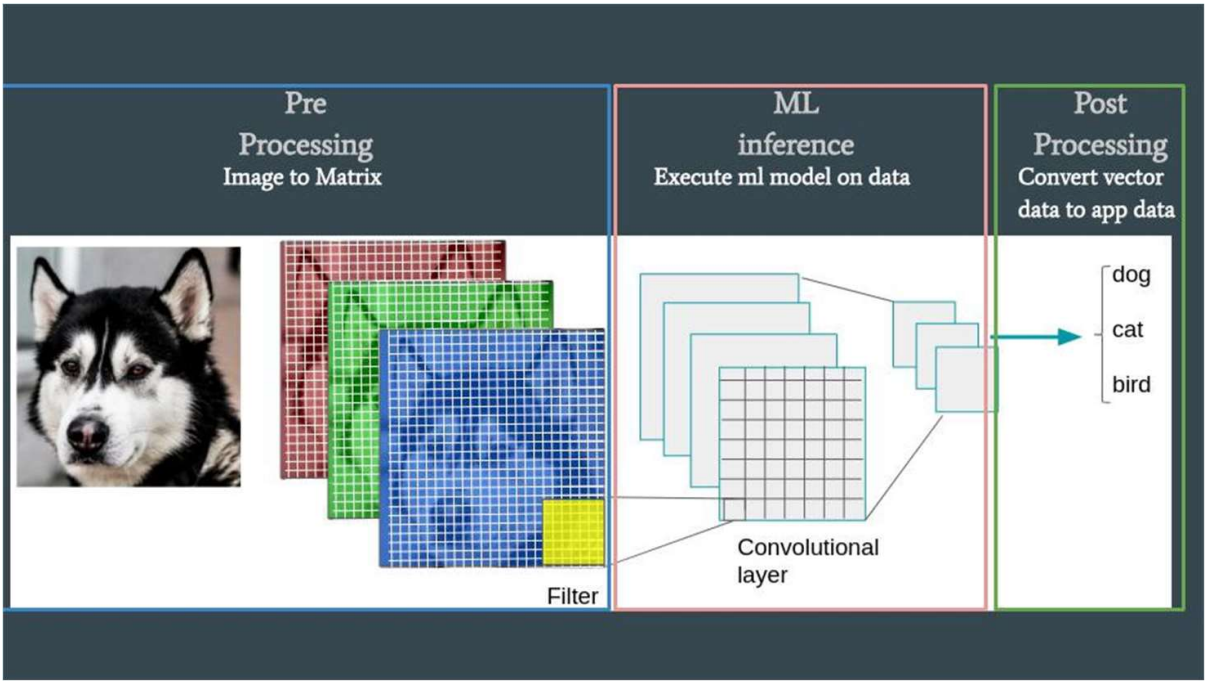
See how CNN work [here (https://www.cs.cmu.edu/~aharley/vis/conv/)](https://www.cs.cmu.edu/~aharley/vis/conv/) and [here (https://transcranial.github.io/keras-js/#/)](https://transcranial.github.io/keras-js/#/).

In the **basic variant** the convolution neural network (for classification task) consists of:

- **Feature extraction part**: several convolution layers combined with pooling (down sampling) and other sub-layers (e.g. normalization, drop-out and e.t.c.).

  The goal of this part is to extract a so-called **feature map** that represent a such set of input image features that suit for the task the best.

- **Head layers part** (here full-connected classifier): the goal of this part is to make a decision in correspondence with the task. For instance, in the case below head layers make decision that it is the bird in the input image due to extracted features.



The correctiveness of feature extraction and decision in the head layers is provided by the supervised learning routine.

As a rule back-propagation is applied for CNN model learning.

Then higher the layer is then more abstract (less formal) and more large features are extracted.

This explanation is connected with the concept **"receptive field"**.

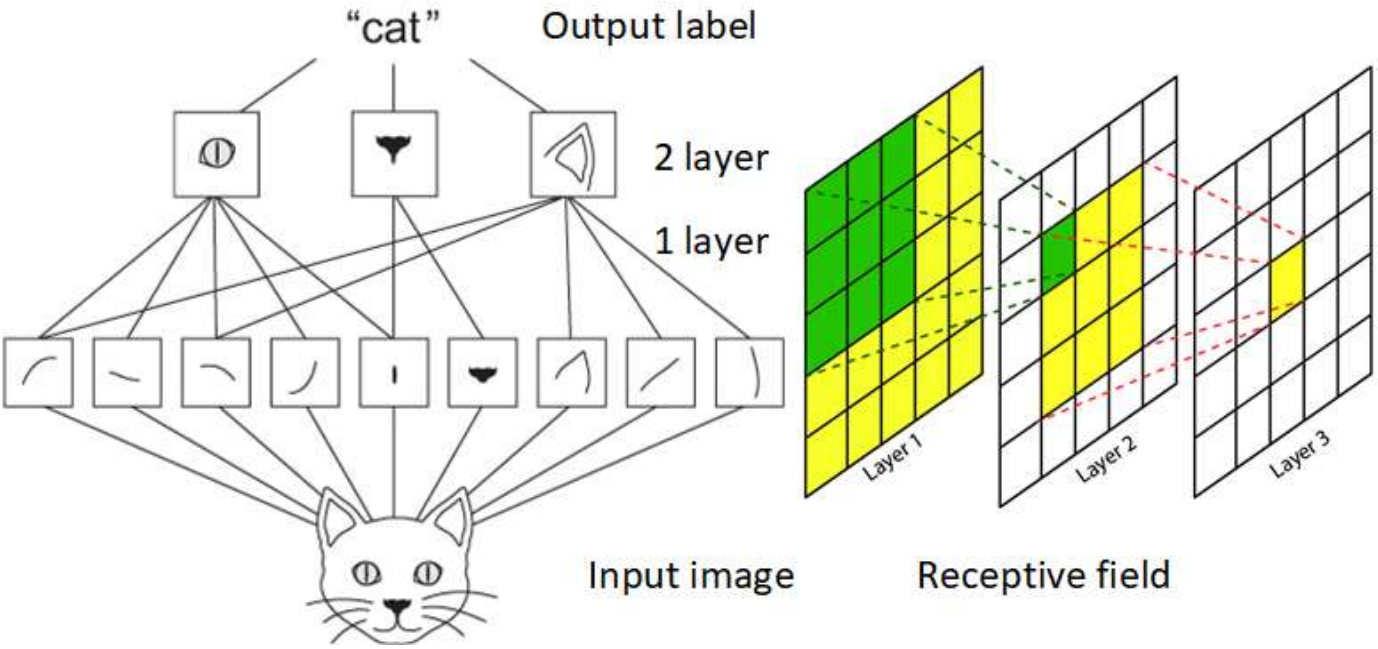
Typesetting math: 100%

The **receptive field** is the region in the input space that a particular CNN's feature is

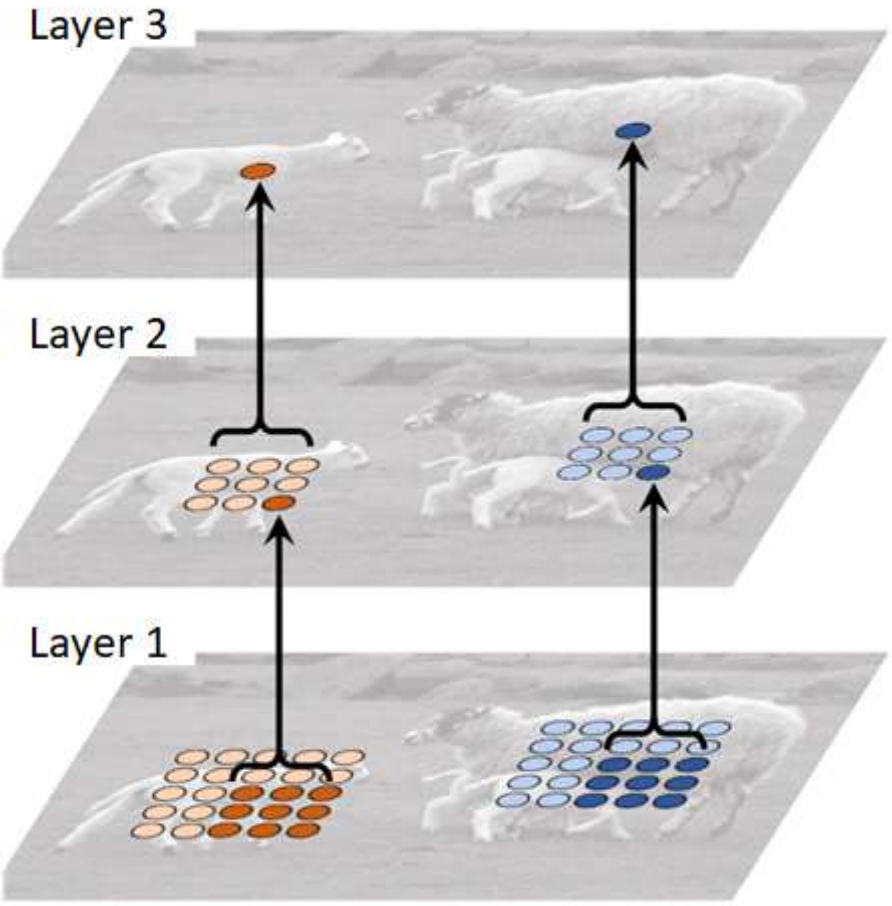
looking at (i.e. be affected by).

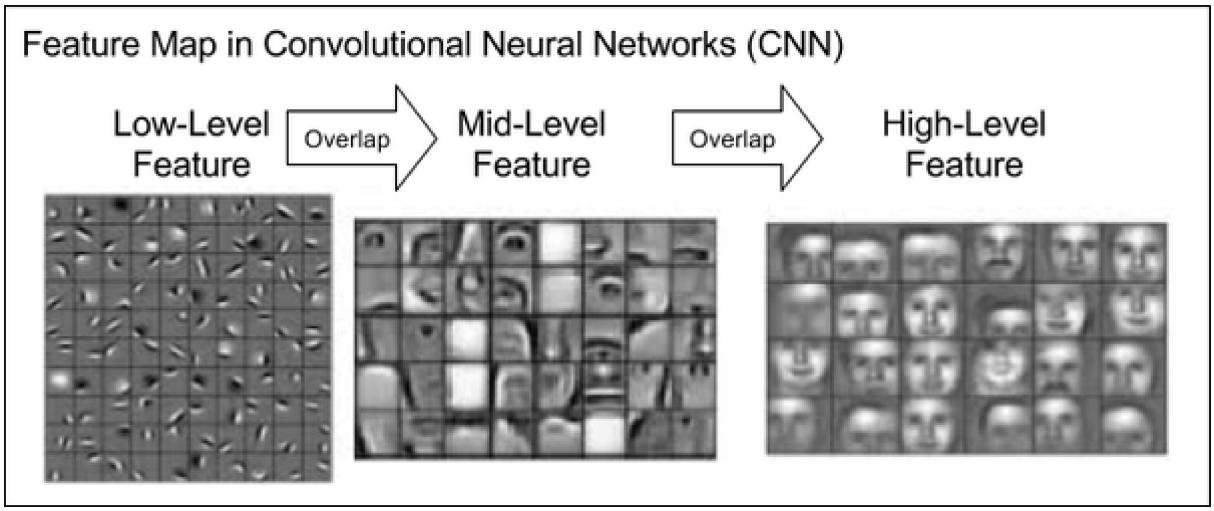
Then deep the network or then higher the kernels size - then more large the receptive field.

However, in practice we always try to deeper the network, due to the increasing of the kernel size can lead to the neglication of the small features, and increasing of the network size at all.



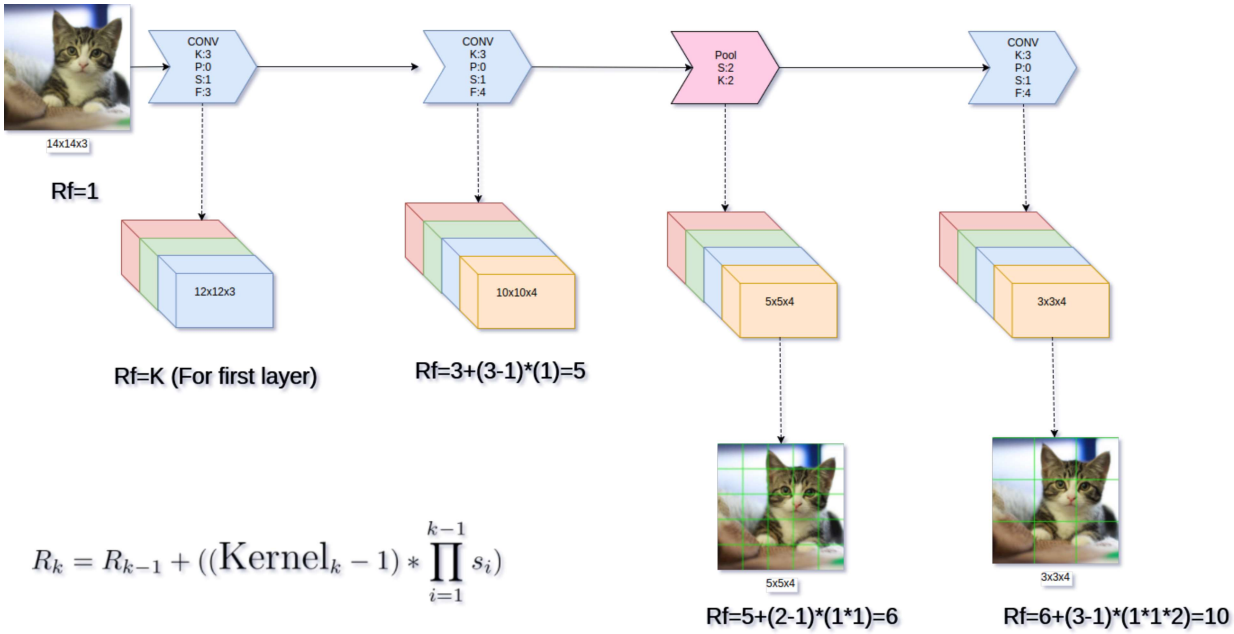
Another illustration of feature size for same-padding 2d convolution





Note with increasing the layer number (and reducing the feature size) it it recommended to proportionally increase the amount of feature maps.

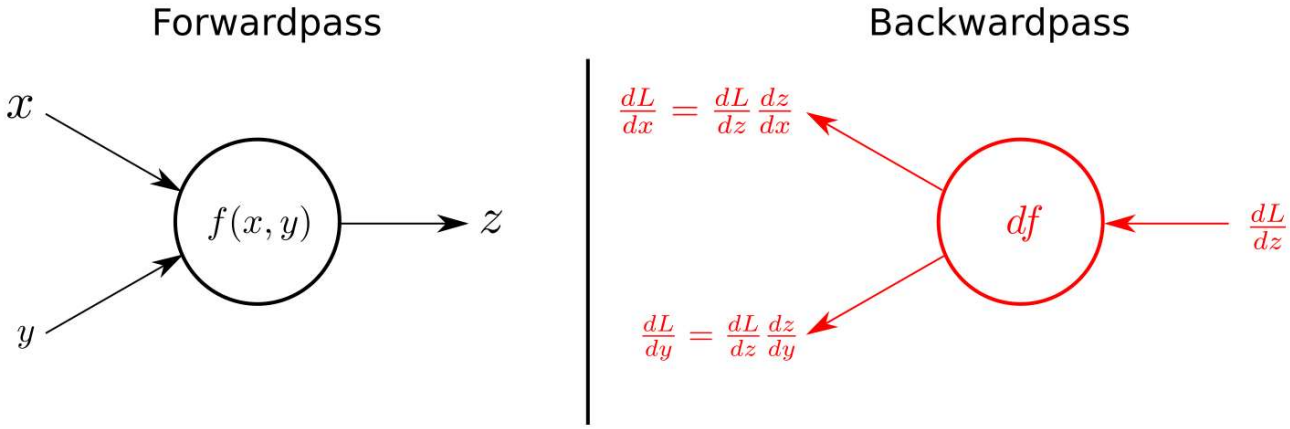
Another example how receptive filed is growing with increasing layer depth.

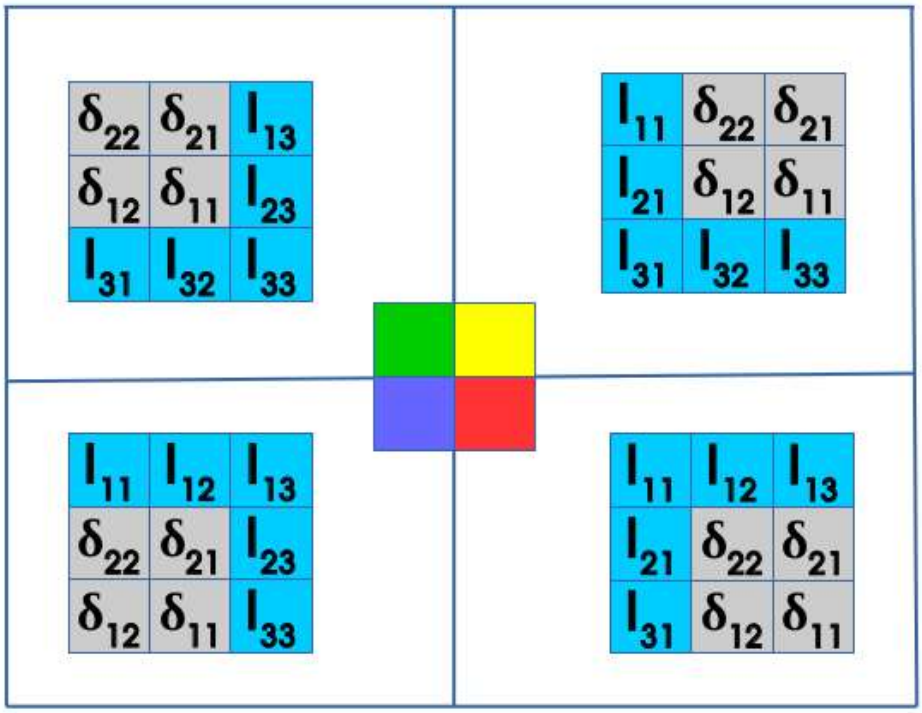


$$R_k = R_{k-1} + ((\text{Kernel}_k - 1) * \prod_{i=1}^{k-1} s_i)$$

It is need to remind that CNN training weights using back-propagation.

In essence **The Back-Propagation** is the gradient descent method of parameters update using modified chain rule.





1.2 Convolution Operation in Depth

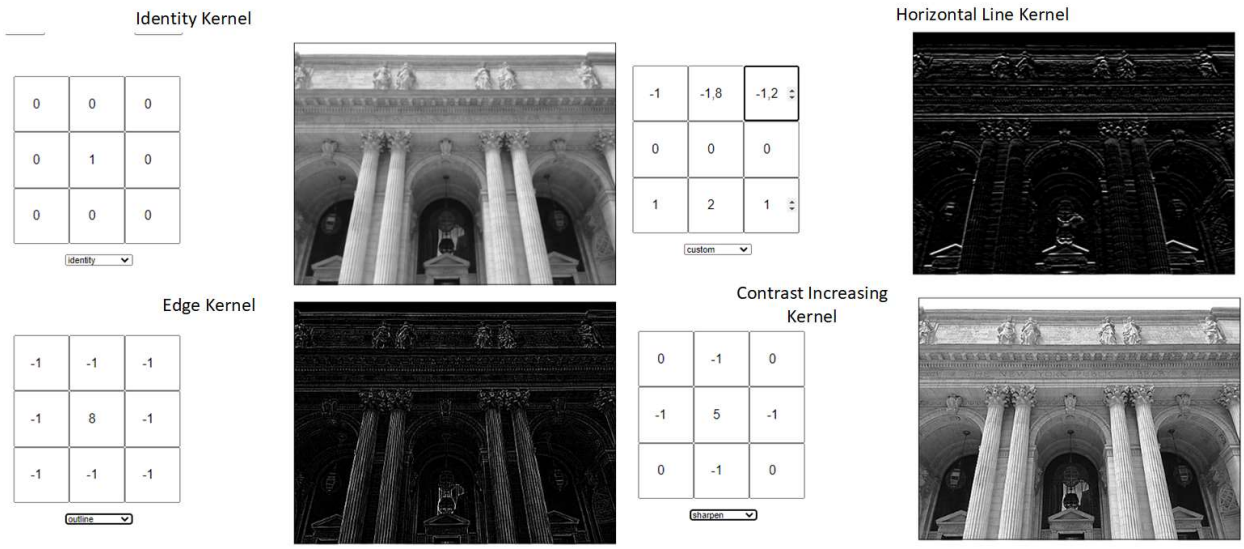
1.2.1 2D-Convolution

The principle of the convolution work for 1 channel is the following operation

$$r[i,j] = (k * x)[ij] = \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} k[a,b]x[i+a, j+b],$$

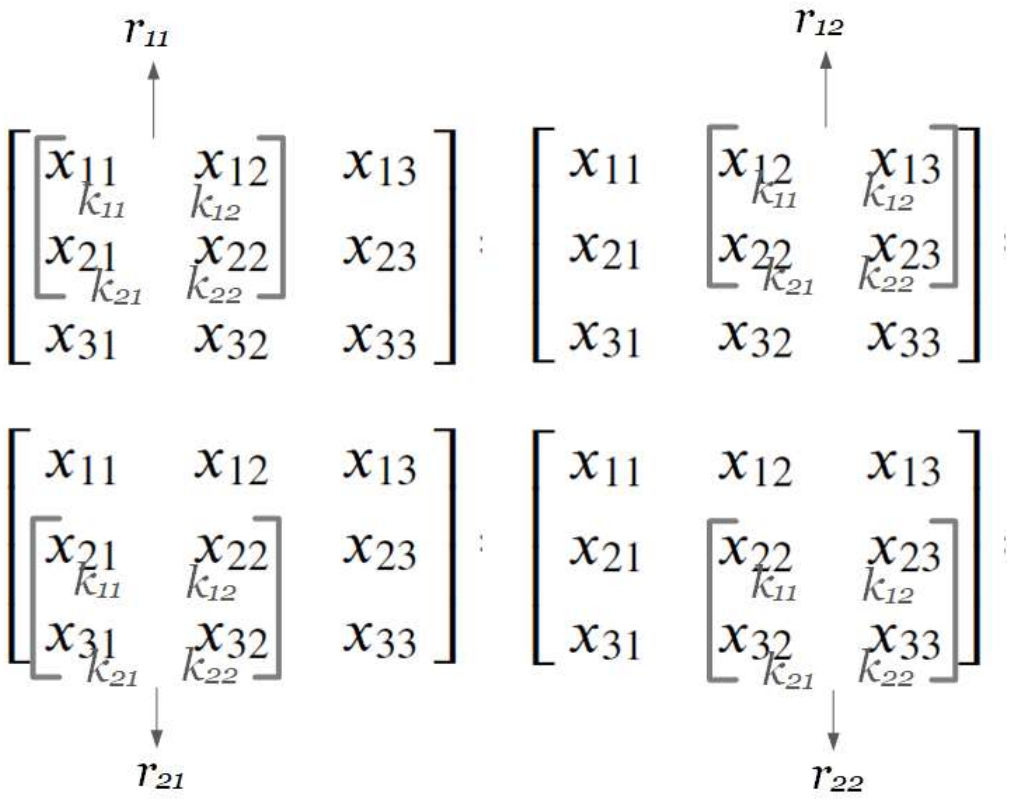
where r is the convolution output for convolution of input image x and kernel k with size $k_h \times k_w$.

The meaning of each kernel in CNN is to extract some feature, as it shown in the example below.



You may find a lot of how simple convolution filter work [here (https://setosa.io/ev/image-kernels/)](https://setosa.io/ev/image-kernels/).

The convolution operation is equal to the sliding by the filter through the image as it shown in the example:



So we can actually rewrite convolution using pointwise (or element-wise, Hadamard) product $\odot$ as

```
r[0,0] = np.sum(x[:3,:3]  * k)
r[1,0] = np.sum(x[1:4,:3] * k)
r[2,0] = np.sum(x[2:5,:3] * k)
r[0,1] = np.sum(x[:3,1:4] * k)
r[0,2] = np.sum(x[:3,2:5] * k)
```

$$ r[ij] = \sum_{ij}{x[i:i+k_h,j:j+k_w]\odot k}$$

Actually the convolution operation can be represented trough the convolution matrix.

In the example below you may see the case of valid convolution operation

representation in the matrix form

$$ \begin{aligned} &r = k\ast x = \boldsymbol{k}\cdot \vec{x} = \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \\ \end{bmatrix} * \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} = \\ &= \begin{bmatrix} k_{11} & k_{12} & 0 & k_{21} & k_{22} & 0 & 0 & 0 & 0 \\ 0 & k_{11} & k_{12} & 0 & k_{21} & k_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & k_{11} & k_{12} & 0 & k_{21} & k_{22} & 0 \\ 0 & 0 & 0 & k_{11} & k_{12} & 0 & k_{21} & k_{22} \\ \end{bmatrix} \cdot \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{21} \\ x_{22} \\ x_{23} \\ x_{31} \\ x_{32} \\ x_{33} \\ \end{bmatrix} = \\ &= \begin{bmatrix} k_{11}x_{11}+ k_{12}x_{12} + k_{21}x_{21} + k_{22}x_{22}\\ k_{11}x_{21}+ k_{12}x_{31} + k_{21}x_{22} + k_{22}x_{23}\\ k_{11}x_{31}+ k_{12}x_{21} + k_{21}x_{31} + k_{22}x_{32}\\ k_{11}x_{22}+ k_{12}x_{23} + k_{21}x_{32} + k_{22}x_{33}\\ \end{bmatrix} = \begin{bmatrix} r_{11} \\ r_{12} \\ r_{21} \\ r_{22} \\ \end{bmatrix} \to \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix} \end{aligned} $$

where:

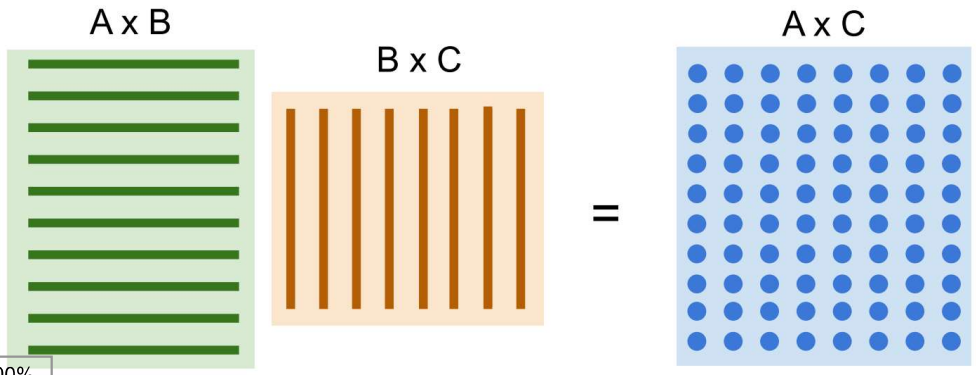
- r is convolution output;
- k is the kernel matrix;
- x is the input image;
- $\boldsymbol{k}$ is the convolution matrix, made from kernel;
- $\vec{x}$ is the vectorized (flatten) input image;

Note matrix multiplication is the operation made by the rule:

> Each output element ij is sum of product of i-th row of first matrix and j-th column of the second matrix.

For the previous example:

$$ \begin{aligned} &r_{11} = \begin{bmatrix} k_{11} & k_{12} & 0 & k_{21} & k_{22} & 0 & 0 & 0 & 0 \\ \end{bmatrix} \cdot \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{21} \\ x_{22} \\ x_{23} \\ x_{31} \\ x_{32} \\ x_{33} \\ \end{bmatrix} = \\ &= k_{11}x_{11}+ k_{12}x_{12} + k_{21}x_{21} + k_{22}x_{22} \end{aligned} $$

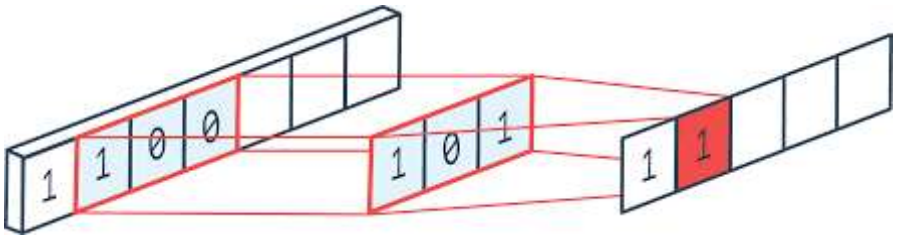


For better understanding the convolution in multi-dimension case it is useful to consider the 1-d case first.
The 1d convolution can be described as $$r = k\ast x = \boldsymbol{k}\cdot \vec{x},$$ where:

- k is the weights vector $k=(k_0, k_1, ..., k_{K-1})$, K is the kernel size.,
- x is the input vector, $x = (x_0, x_1, ..., x_{N-1})$, N is the input size, and $\vec{x} = x^T$,
- $\boldsymbol{k}$ is the convolution matrix. $$ \boldsymbol{k} = \begin{bmatrix} k_0 & k_1 & \dots & k_{K-1} & 0 & \dots & 0 & \dots & 0 \\ 0 & k_0 & \dots & k_{K-2} & k_{K-1} & \dots & 0 & \dots & 0 \\ \dots \\ 0 & 0 & \dots & 0 & 0 & \dots & k_0 & \dots & k_{K-1}\\ \end{bmatrix} $$

$$ \begin{aligned} &r = k\ast x = \boldsymbol{k}\cdot \vec{x} = \begin{bmatrix} k_0 & k_1 & \dots & k_{K-1} & 0 & \dots & 0 & \dots & 0 \\ 0 & k_0 & \dots & k_{K-2} & k_{K-1} & \dots & 0 & \dots & 0 \\ \dots \\ 0 & 0 & \dots & 0 & 0 & \dots & k_0 & \dots & k_{K-1}\\ \end{bmatrix} \cdot \begin{bmatrix} x_{0} \\ x_{1} \\ x_{2} \\ \vdots \\ x_{n-1} \\ \end{bmatrix} = \\ &= \begin{bmatrix} k_0\cdot x_0 + k_1 \cdot x_1 + \dots + k_{K-1} \cdot x_{K-1} \\ k_0\cdot x_1 + k_1 \cdot x_2+ \dots + k_{K-1} \cdot x_{K} \\ k_0\cdot x_2 + k_1 \cdot x_3 + \dots + k_{K-1} \cdot x_{K+1} \\ \vdots \\ k_0\cdot x_{n-1-K} + k_1 \cdot x_{n-K} + \dots + k_{K-1} \cdot x_{n-1} \\

\end{bmatrix} = \begin{bmatrix} r_{0} \\ r_{1} \\ r_{2} \\ \vdots \\ r_{n-1} \\ \end{bmatrix} \end{aligned} $$ Example of 1-d convolution

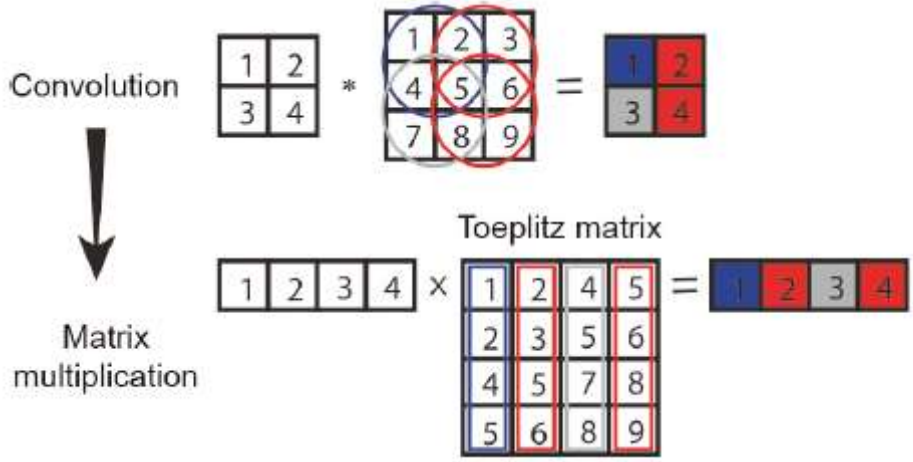


Note: The filter kernel size K corresponds to the receptive field size in classical CNN approach.

The matrix representation of the convolution is also known as **kn2raw** or **kernel to raw**.

The the order of operations kn2raw can be transformed to the case known as **Img2col** or **Image to column**.

The Img2col is also could be shown as below



Beside the simple convolution (**direct loop convolution**) representation as it shown above, it can be proposed several types of acceleration the convolution operation:

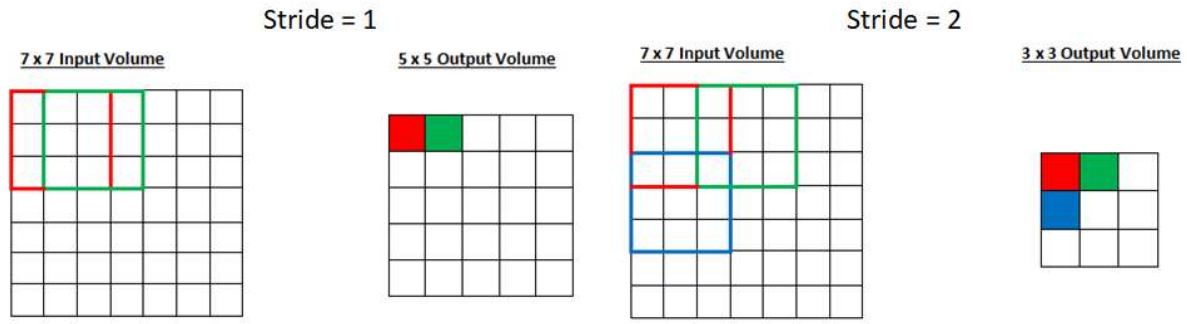
1. **direct loop convolution**.
2. **Image to column convolution (im2col)**.
3. **kernel to column convolution (kn2col)**.
4. **Convolution though the Fast Fourier Transform (FFT)**:

   $$r = \mathcal{F}^{-1}(\mathcal{F}(k)\cdot\mathcal{F}(x)),$$ where $\mathcal{F}$ and $\mathcal{F}^{-1}$ are the direct and inverse Fast Fourier Transforms; and in the case of the k the kernel is zero-padded to the shape of image.
5. **Fast Winograd algorithm (Winograd)**: method is based on the dividing the input image to a set of small blacks and simplification the complexity for convolution for each of the blocks.

Algorithm	Time	Memory	Stridded	Bad cases	Quant. ready
Direct loop	−−	++	++	Non-strided	✓
im2	+	−−	++	Large inputs	✓
kn2	+	+	−−	Few Channels	✓
Winograd	++	−	−	Unpredictable	✗
FFT	++	−	+	Small kernel	✗

So the convolution can be represented as sliding window which is moving through the image

with any predefined stride (stride=1 in the example).



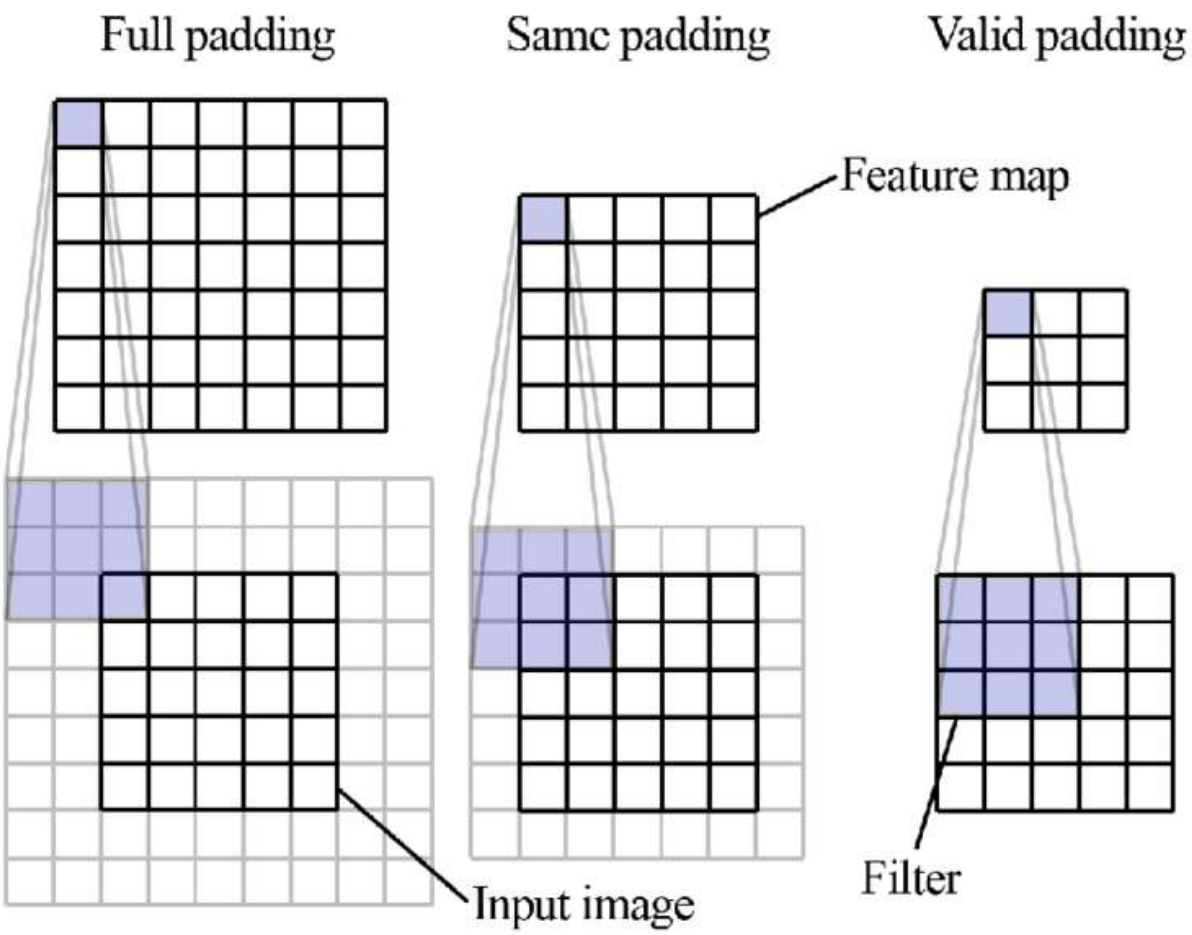
The convolution can be carried out with or without padding.

The padding is the operation of virtually add values beside the image boards.

As a rule zero-values padding is using. It can be distinguished

- **same** padding (with add such zeros that the output size will be equal to the input one).
- **valid** padding (without add zeros).

- **full** padding (with add such number zeros that increase the output image size on the kernel size - 1).



The output size of the convolution will be

$$dim(r) =r_h\times r_w = (\frac{x_h-k_h+2p_h}{s_h}+1) \times (\frac{x_w-k_w+2p_w}{s_w}+1),$$

where:

- The output image size $r_h\times r_w$;

- The input image size $x_h\times x_w$;
- The kernel image size $k_h\times k_w$;
- The padding size
  - p_h is the number of addition columns and
  - p_w, is the number of addition raws;
- The stride size
  - s_h is the stride in the horizontal direction
  - p_w, is the stride in the vertical direction.

Here is another example for kernel with size 3

$$ \begin{bmatrix} r_{11} & r_{12} \\\\ r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\\\ k_{21} & k_{22} & k_{23} \\\\ k_{31} & k_{32} & k_{33} \end{bmatrix} * \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14}\\\\ x_{21} & x_{22} & x_{23} & x_{24}\\\\ x_{31} & x_{32} & x_{33} & x_{34}\\\\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix} \to \begin{bmatrix} r_{11} \\\\ r_{12} \\\\ r_{21} \\\\ r_{22} \end{bmatrix} = \boldsymbol{k}\cdot \begin{bmatrix} x_{11}\\\\ x_{12}\\\\ \vdots \\\\ x_{44} \end{bmatrix} $$ where

$$ \boldsymbol{k}= \left[\begin{smallmatrix} k_{11} & k_{12} & k_{13} & 0 & k_{21} & k_{22} & k_{23} & 0 & k_{31} & k_{32} & k_{33} & 0 & 0 & 0 & 0 & 0\\\\ 0 & k_{11} & k_{12} & k_{13} & 0 & k_{21} & k_{22} & k_{23} & 0 & k_{31} & k_{32} & k_{33} & 0 & 0 & 0 & 0\\\\ 0 & 0 & 0 & 0 & k_{11} & k_{12} & k_{13} & 0 & k_{21} & k_{22} & k_{23} & 0 & k_{31} & k_{32} & k_{33} & 0\\\\ 0& 0 & 0 & 0 & 0 & k_{11} & k_{12} & k_{13} & 0 & k_{21} & k_{22} & k_{23} & 0 & k_{31} & k_{32} & k_{33}\\\\ \end{smallmatrix} \right] $$

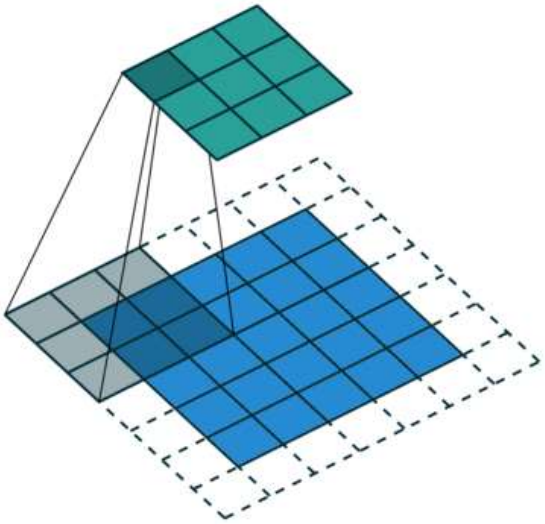
As extension of the simple convolution in may cases the dialed convolution can be applied.

$$ r[i,j] = (k\ast_l x)[ij] = \sum_{a=0}^{k_h-1}\sum_{b=0}^{k_w-1}k[a,b]x[i\cdot l+a,j \cdot l+b], $$

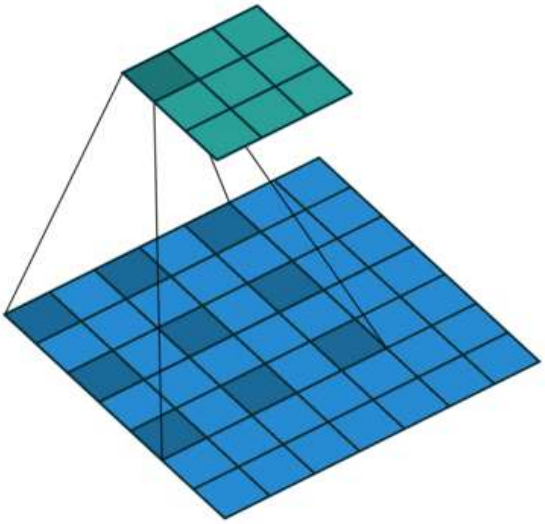
where $l $ is the dialed step.

Types of convolutions

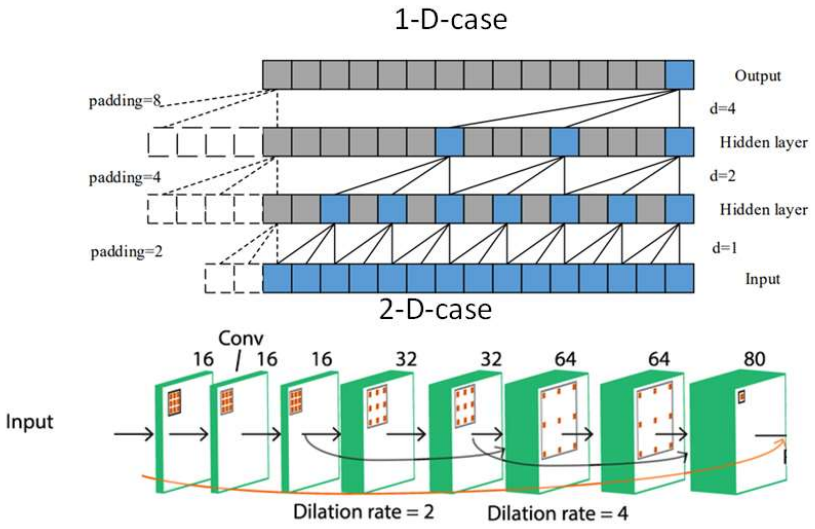
Traditional convolution



Dialed convolution



Notes: Example of multi-scaled dialed convolution for 1d-case and 2d-case receptive field increasing.



As it can be seen the dialed convolution allowing increase the receptive field without increasing the kernel size.

1.2.2 Multiple Channel Multiple Kernel (MCMK) Convolution 2D

As a rule a set of convolution kernels simultaneously applied for a input tensor (set of feature maps)

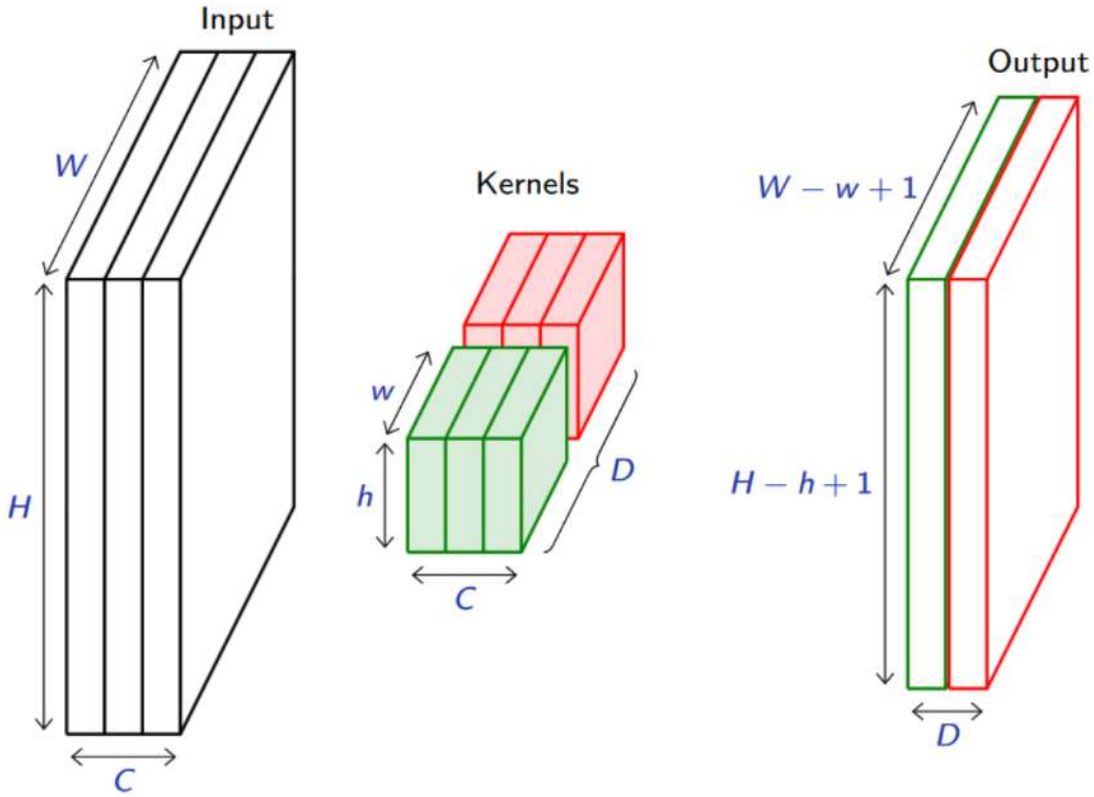
for obtain one output feature map.

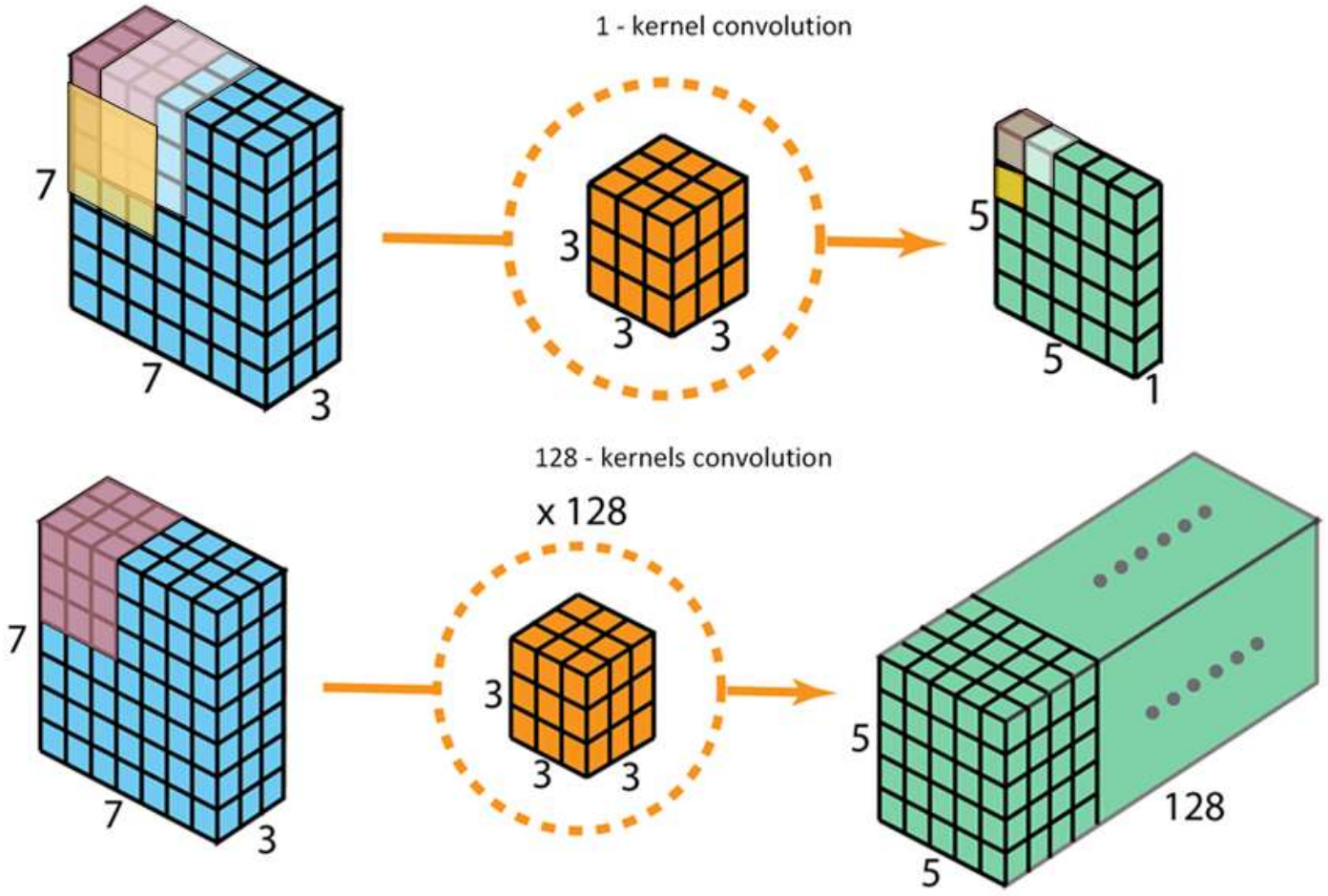
Thus, in fact you obtain 3d-convolution which is a sum of 2d-convolution with a corresponding set of kernels.

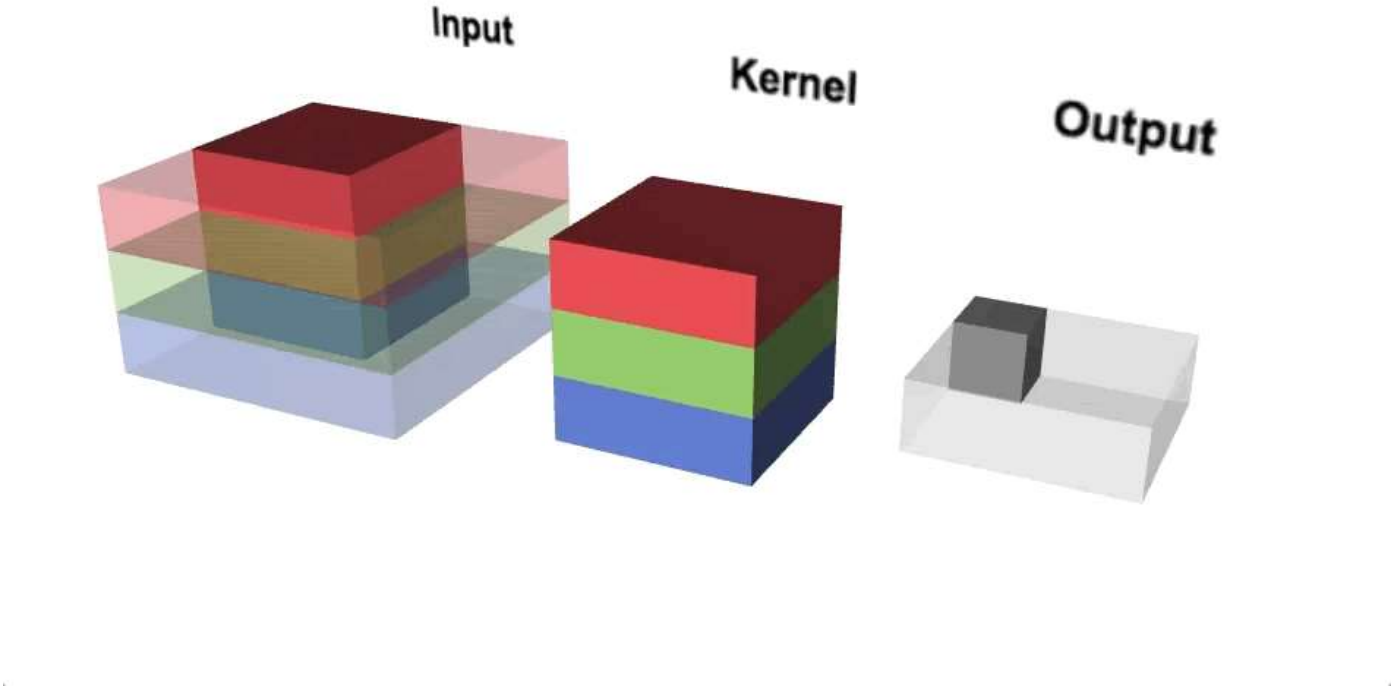
$$ r[i,j] = (k\ast x)[ij] = \sum_{c=0}^{C-1} \big[\sum_{a=0}^{k_h-1}\sum_{b=0}^{k_w-1}k^c[a,b]x^c[i+a,j +b]\big], $$

where C is the number of input feature maps and k^c is the kernel for c-th feature map.

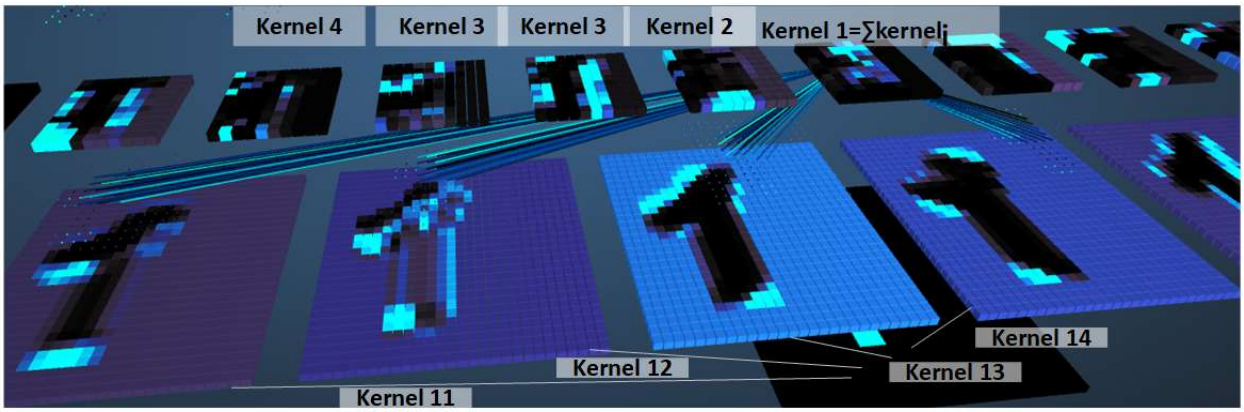
For obtaining several output feature maps we need several 3-d sets of kernels.





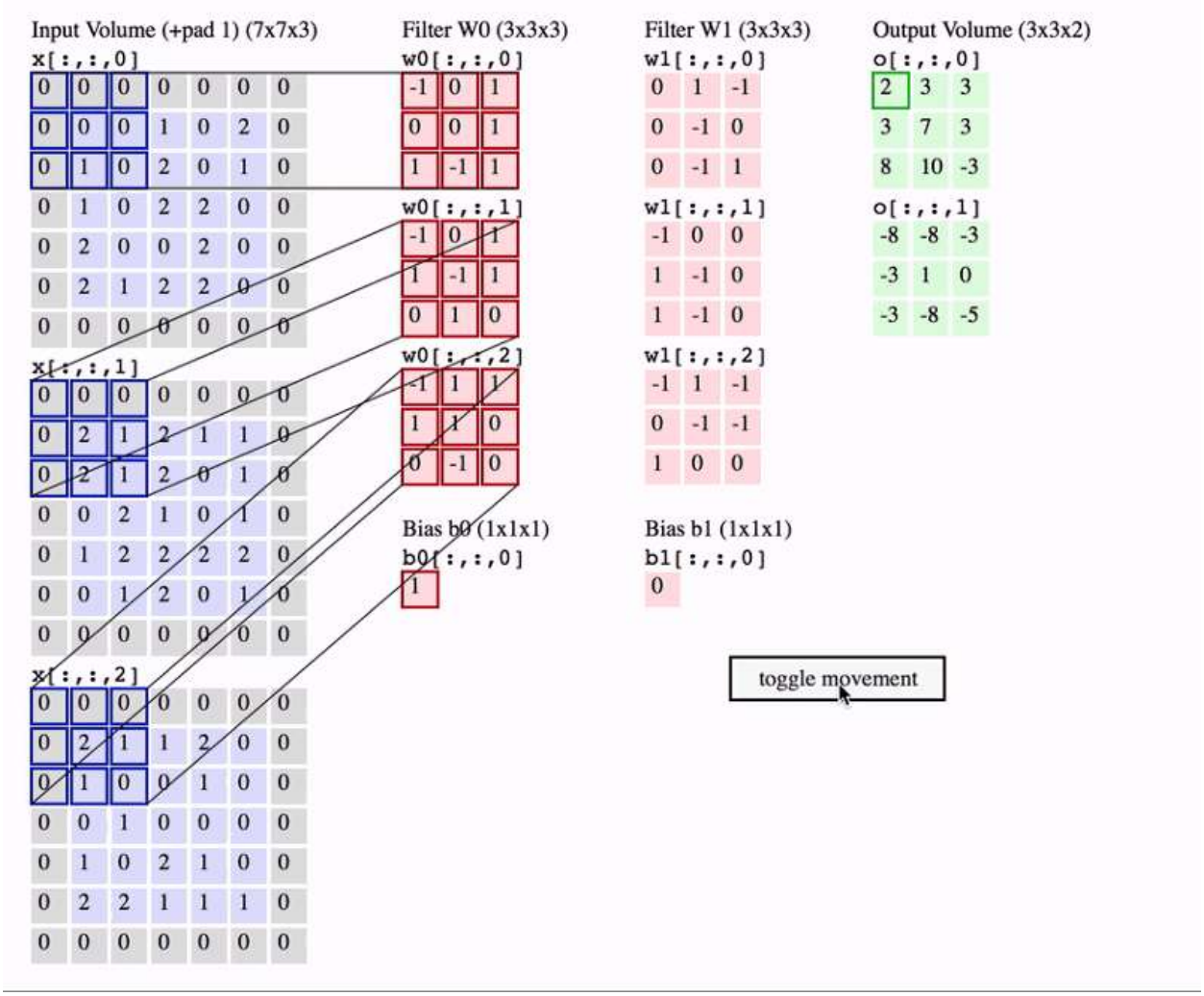


Another illustration of how to obtain $ r = (k\ast x) = \sum_{c=0}^{C-1} (k^c\ast x^c)$



In Addition to convolution as its self the bias is always add for each feature map.

$$ \begin{align} &r[i,j] = (k\ast x)[ij] = \sum_{c=0}^{C-1}(k^c\ast x^c)[ij] =\\ &= \sum_{c=0}^{C-1} \big[\sum_{a=0}^{k_h-1}\sum_{b=0}^{k_w-1}k^c[a,b]x^c[i+a,j +b,c]\big]+bias, \end{align} $$



So we can actually rewrite convolution using pointwise (or element-wise, Hadamard) product $\odot$ as

```
r[0,0,0] = np.sum(X[:3,:3,:3]  * k0[:3,:3,:3]) + b0
r[1,0,0] = np.sum(X[1:4,:3,:3] * k0[:3,:3,:3]) + b0
r[2,0,0] = np.sum(X[2:5,:3,:3] * k0[:3,:3,:3]) + b0
r[0,1,0] = np.sum(X[:3,1:4,:3] * k0[:3,:3,:3]) + b0
r[0,0,1] = np.sum(X[:3,:3,:3]  * k1[:3,:3,:3]) + b1
```

$$ r_{c_{out}}[ij] = \sum_{ijc}{x[i:i+k_h,j:j+k_w,0:C_{in}-1]\odot k_{c_{out}}}+b_{c_{out}}$$

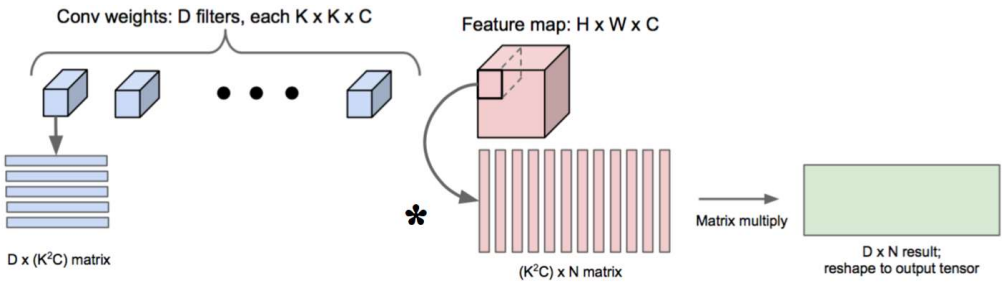
Typesetting math: 100%

where c_{out} is the output channel; C_{in} is the number of input channels; $b_{c_{out}}$ is the kernel bias.

In more mathematical view we could have as input an image (X) with height H, width W and C_{in} channels such that $x\in R^{H\times W\times C_{in}}$. Subsequently for a bank of C_{out} filters we have $K\in R^{k_w \times k_h\times C_{in} \times C_{out}}$ and biases $b\in R^{C_{out}}$, i.e. one for each filter.

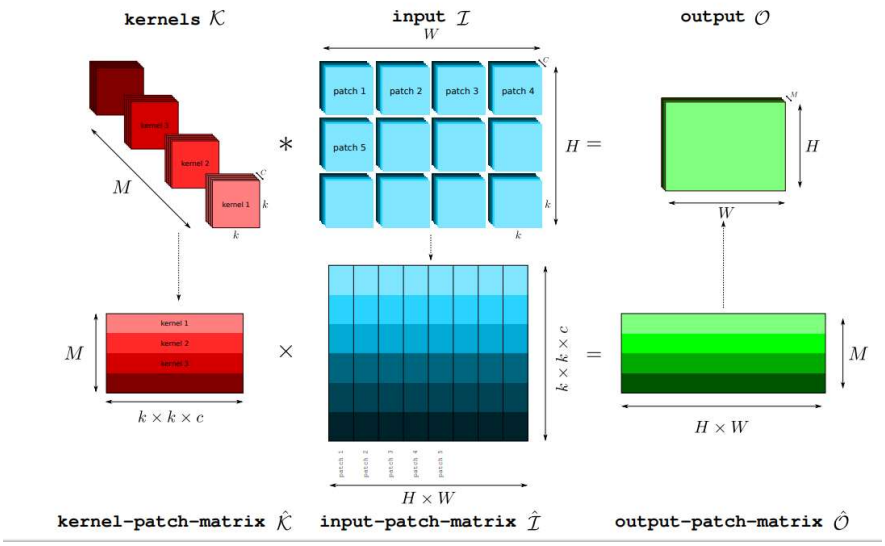
$$ \begin{align} r_{c_{out}}[ij] = (x \ast k_{c_{out}})_{ij} &= \sum_{m = 0}^{k_w - 1} \sum_{n = 0}^{k_h - 1} \sum_{c = 1}^{C_{in}} k_{c_{out}} (m,n,c) \cdot x(i+m, j+n, c) + b_{c_{out}} \end{align} $$

*Note: For the set of convolutions the **img2col** convolution can be represented to the matrix product form, such as:*

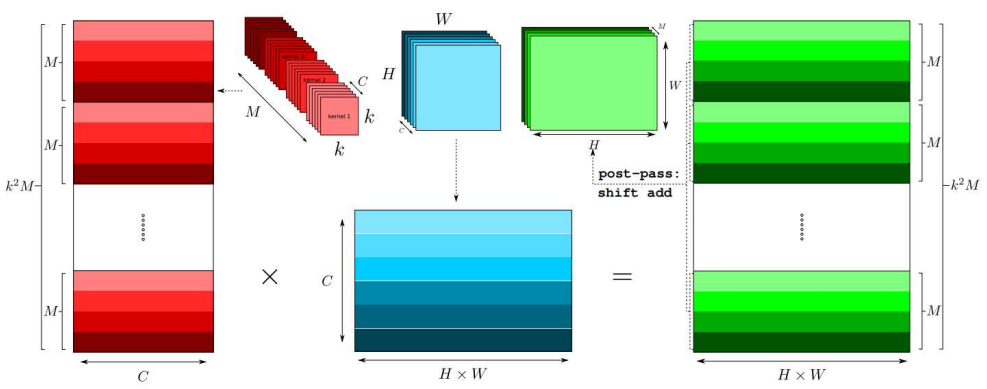


where N is virtual number of patches through the input feature map.

Other representations of img2col and kn2raw convolutions: Img2col

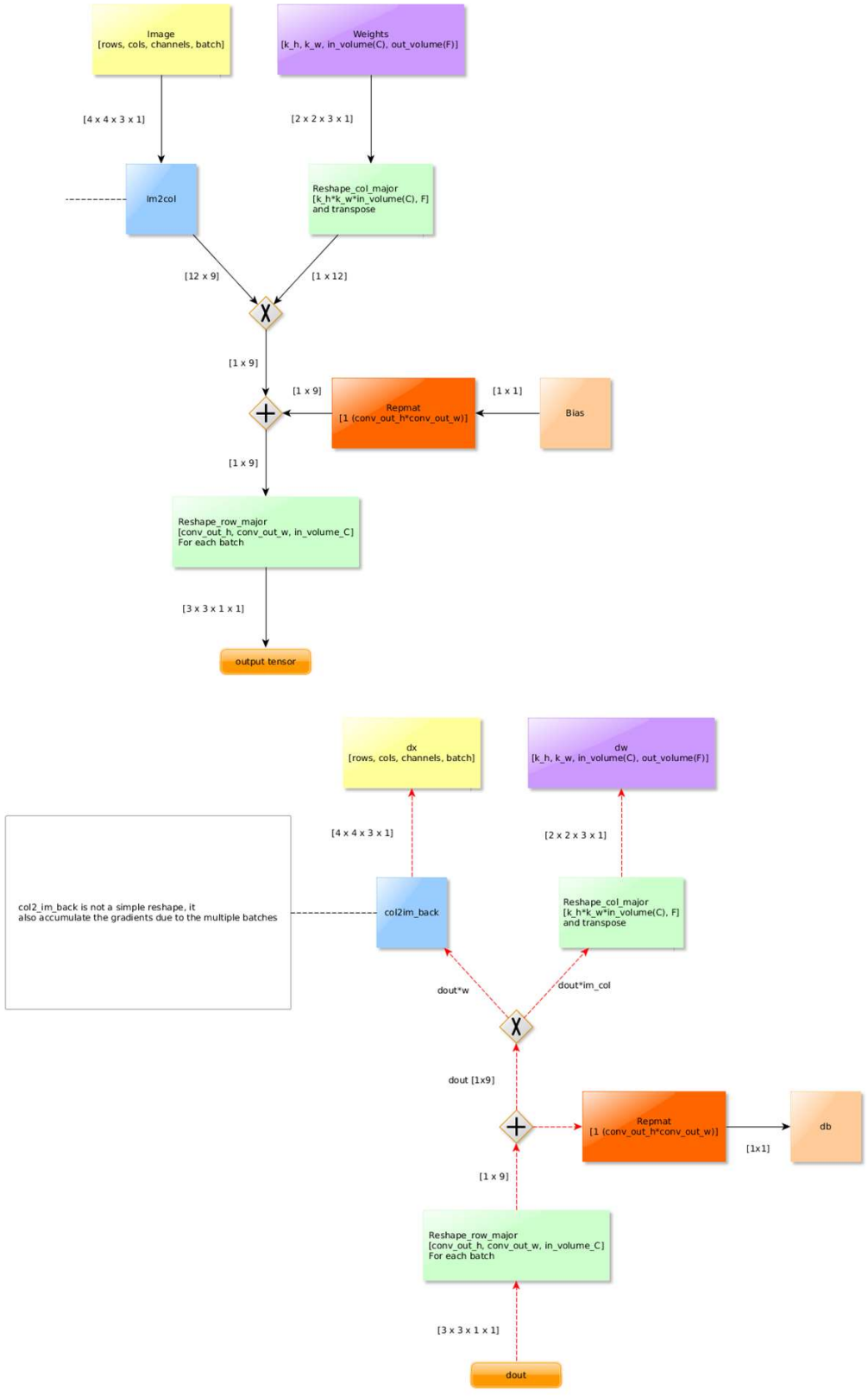


kn2raw



Typesetting math: 100%

The **img2col** Implementation CNN layer is the most popular.



1.2.3 Types of Convolution

Beside from the conventional (Square-kernel convolution) it can be distinguished a several its modifications.

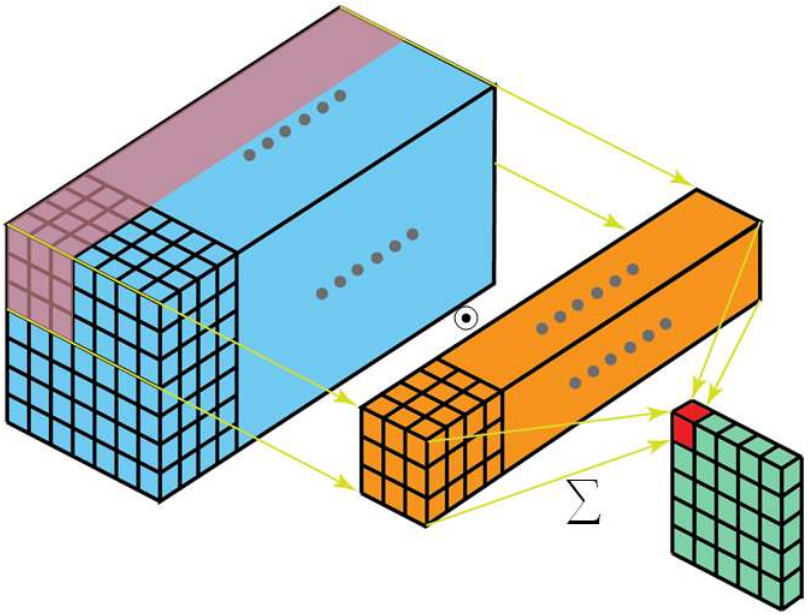
The main motivations for modify traditional convolution are:

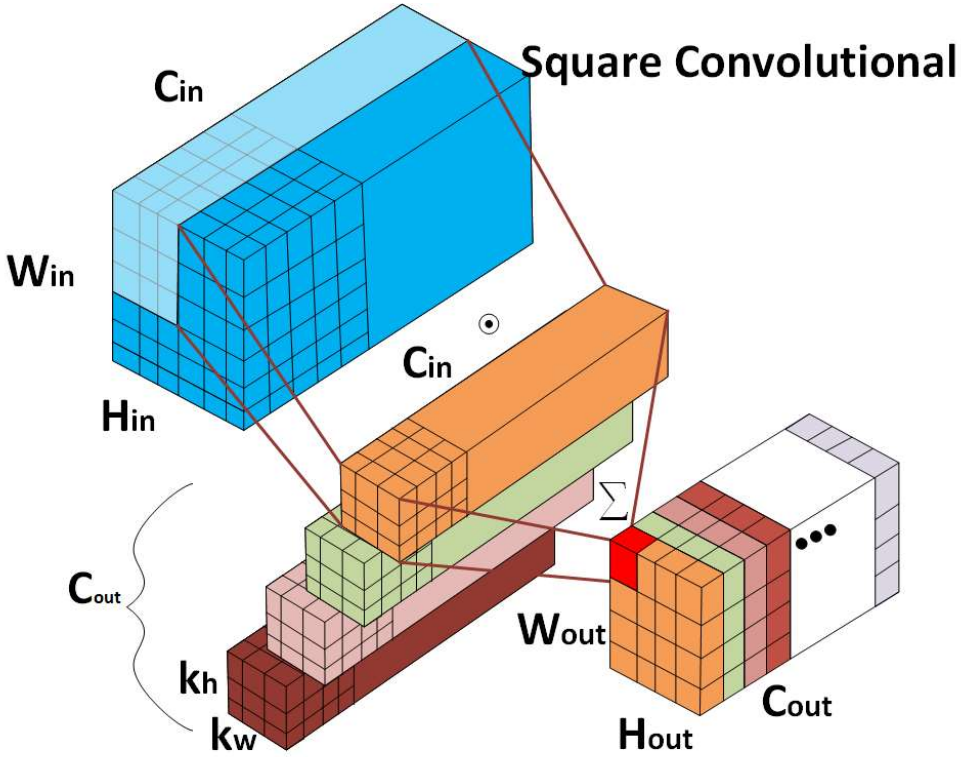
- Reducing number of parameters without loose of accuracy.
- Increase the receptive field (i.e. sensitive to feature while automatically extraction during model training).
- Increase sensitivity to feature by layer complication with minimal increase of computation cost.
- make convolution more effective for our computational resources.

Depends on the architecture of convolution neural network the following **types of convolutions** are proposed.

- **Square-kernel convolution** (for instance, $3\times 3\times C$).

  The one wide convolution can be replaced with several convolution smaller (like 5×5 kernel can be replaced with 2 sequential 3×3 kernels).



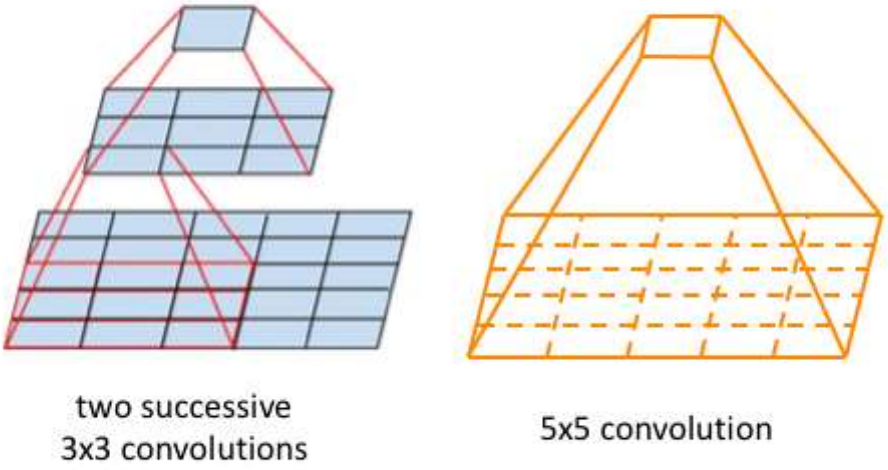


Cascade Convolution

Beside the simple implementation of the convolution it is important to note that one convolution $5x5$ with stride 1 can be replaced on two

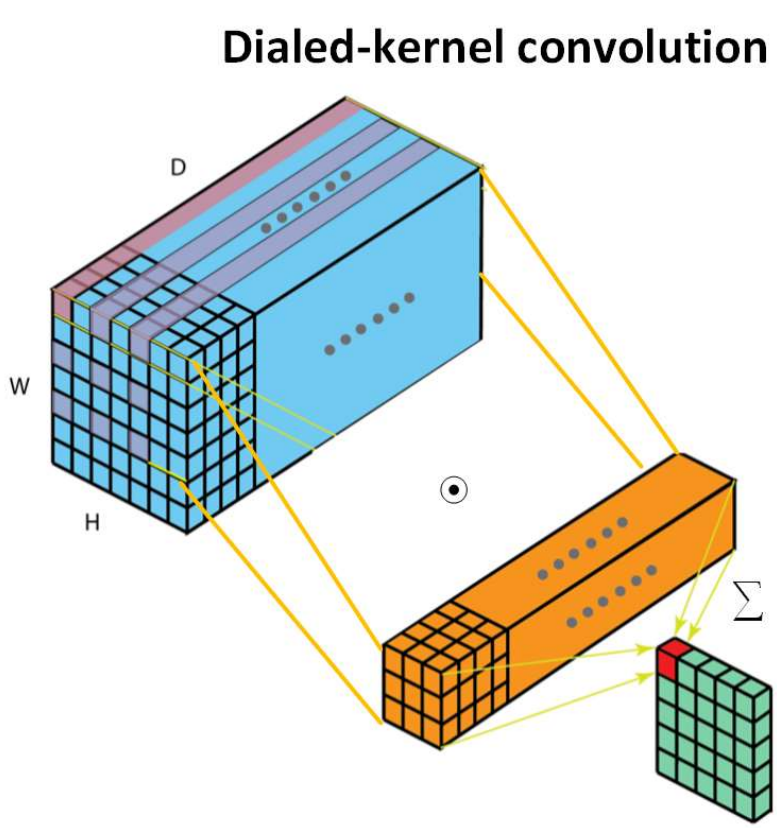
convolution $3x3$ without loose of receptive filed.

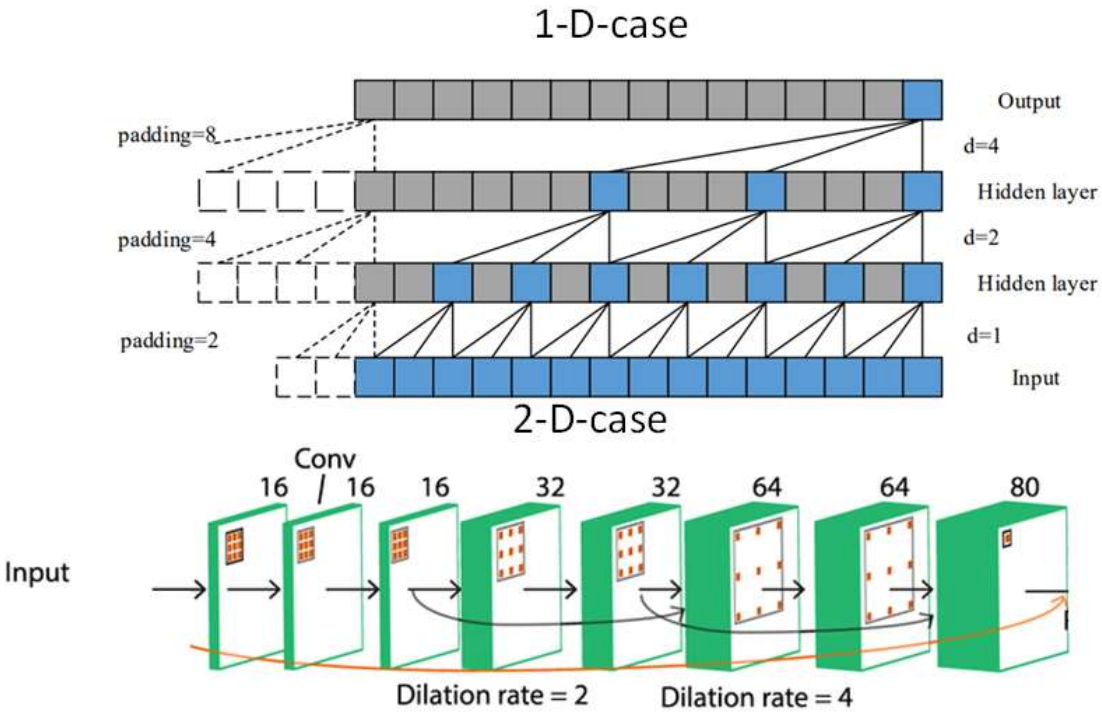
However, in this case the full number of parameters reduced to $$ 2 \times 3 \times 3 = 18 \text{ versus } 1 \times 5 \times 5 = 25 $$ And the number of non-linearities can also be increased.



- **Dialed-kernel Convolution (Atrous Convolution)** (for instance, $3\times 3\times C$ with dialed rate 2).

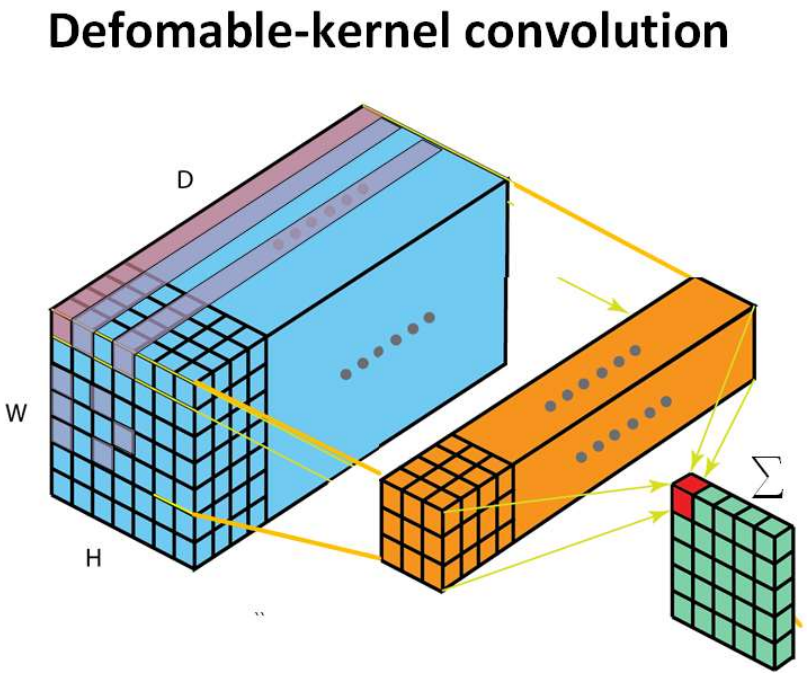
  Reduce the number of parameters with keep wider receptive field (without change 5×5 kernel with 3×3 kernels we can take one dialed 3×3 kernel with rate 2).



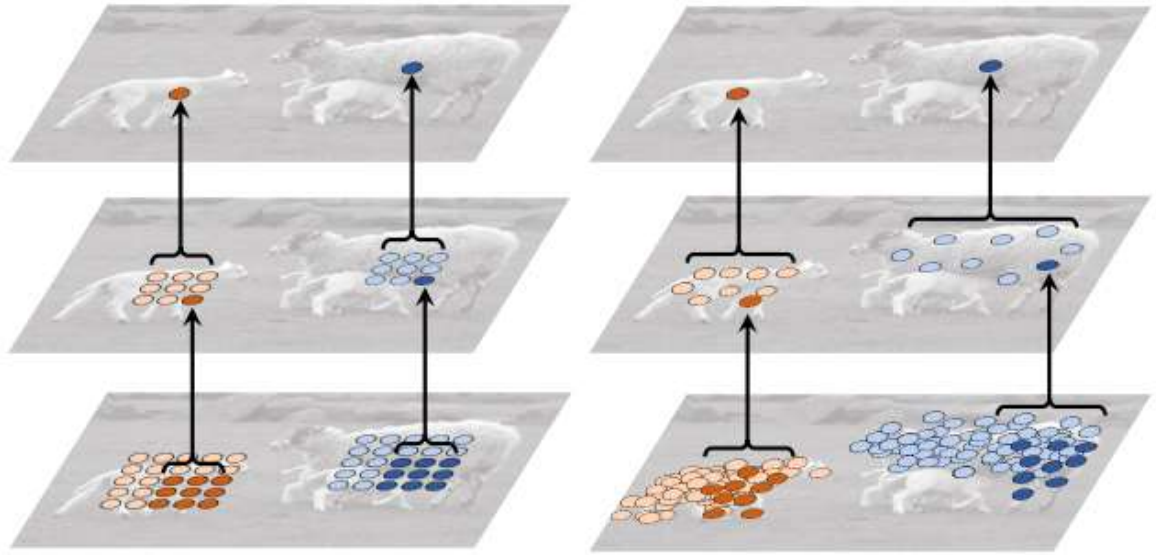


- **Deformable-Kernel Convolution** (for instance, $3\times 3\times C$ with arbitrary dialed rate).

The expansion of dialed convolution for the case of arbitrary rate in each dimension. The main idea here is to increase the virtual receptive filed using virtually wide kernel with virtual pruning.



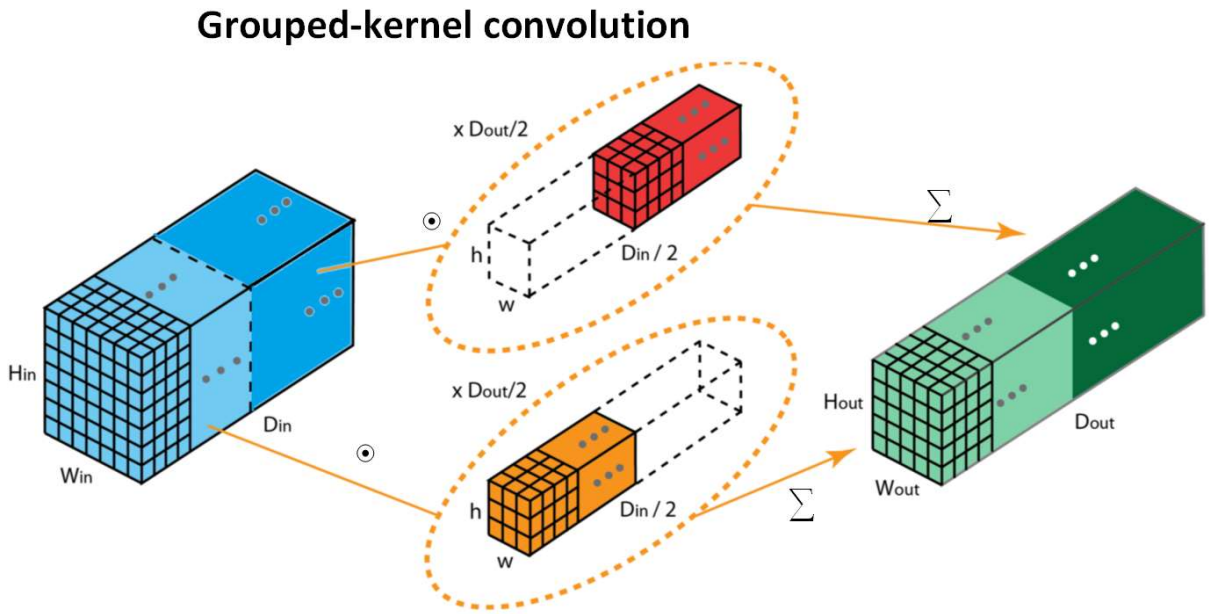
Some explanation of deformable convolution



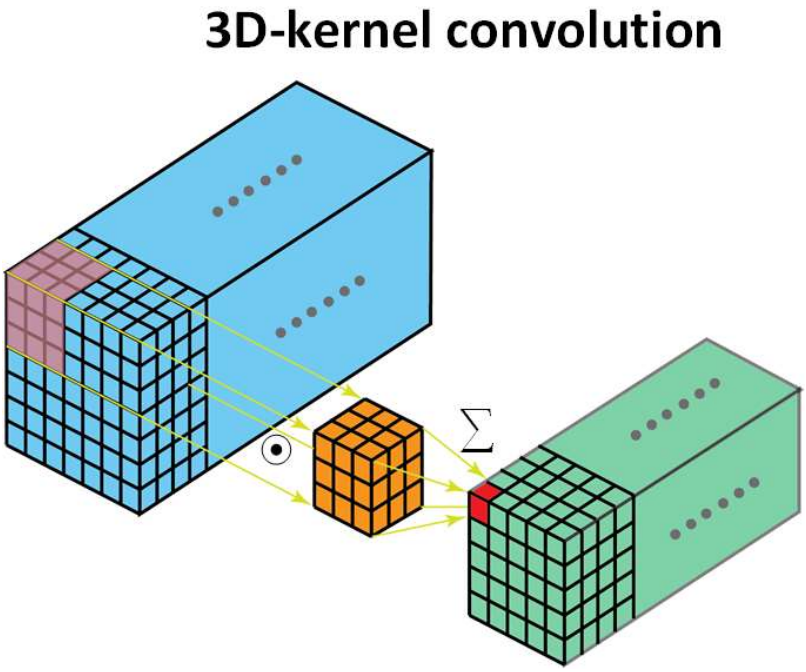
(a) standard convolution (b) deformable convolution

- **Grouped-kernel Convolution** (for instance, $3\times 3\times C/2$ and $3\times 3\times C/2$).

  Reduce the number of parameters in each convolution (for simultaneously computations optimization).

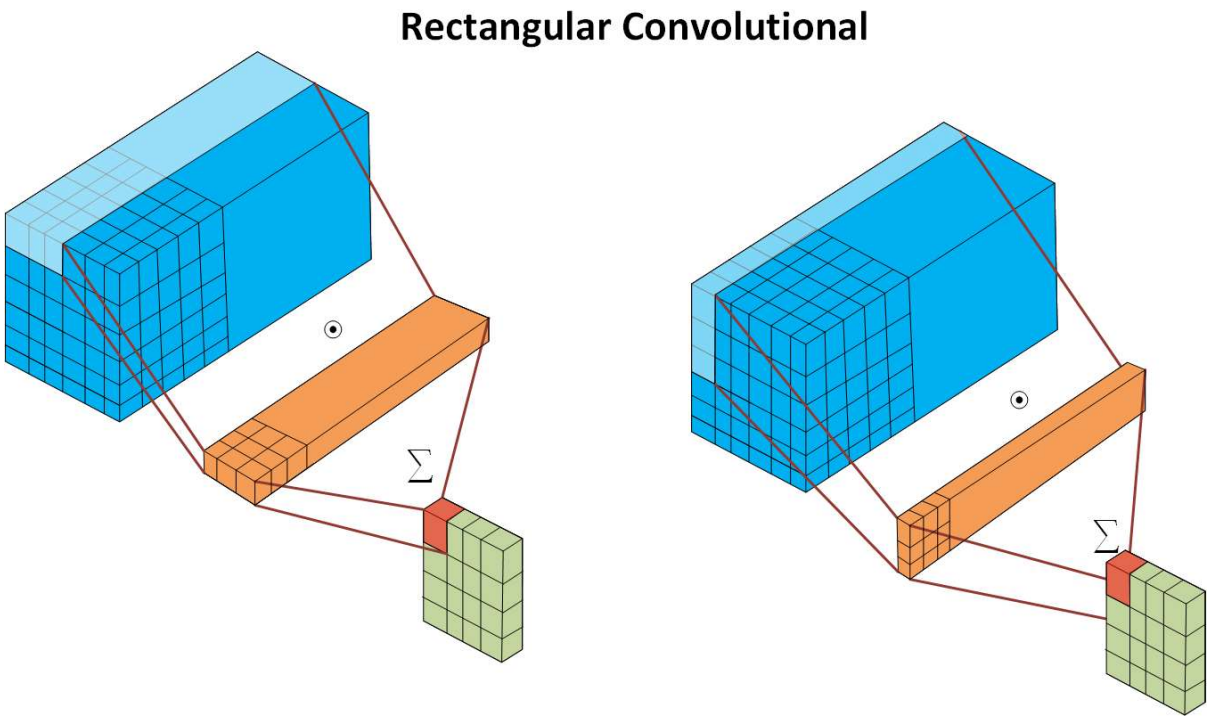


- **3D-kernel Convolution** (for instance, $3\times 3\times 3$ with sliding through 3 dimensions).

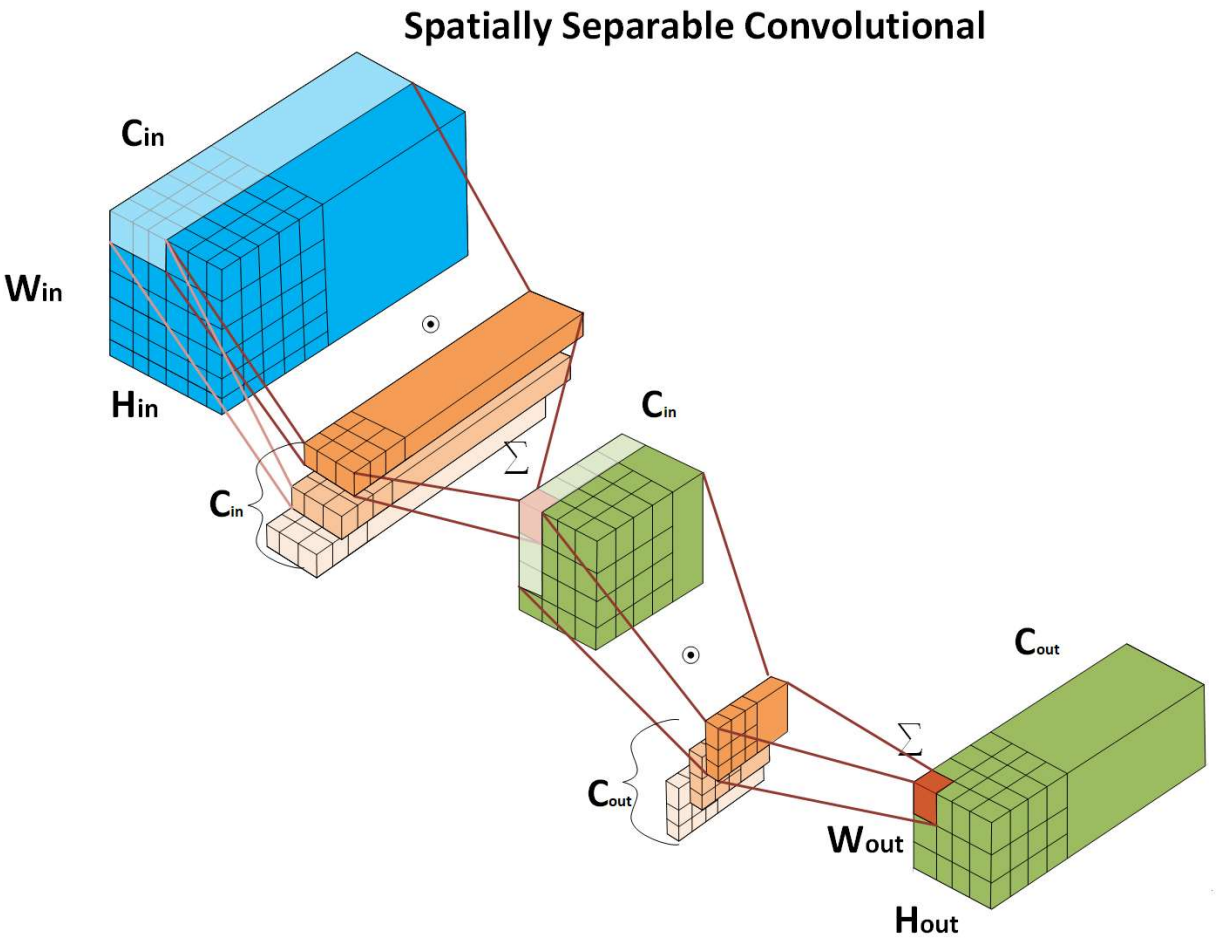


- **Rectangular-kernel convolution** (**spatial convolution**) (for instance, $1\times 3\times C$ or $3\times 1\times C$).

    Reduce the number of parameters.



Spatially Separable Convolution (for instance, 1-stage $C \cdot 3\times 1\times 1$; 2-stage $C \cdot 1\times 3\times 1$).
Combination of two sequential rectangular convolutions for replacing conventional rectangular convolution (with summation)



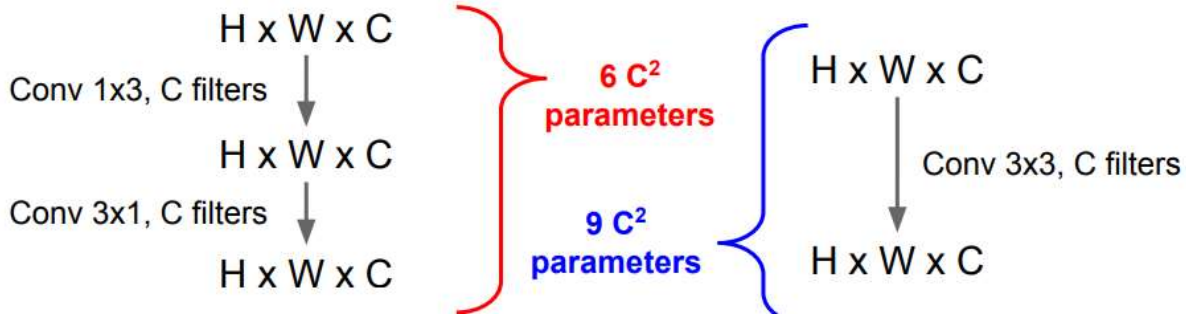
The Separable convolution is based on the following identity

$$\begin{bmatrix} A\cdot a & A\cdot b & A\cdot c \\ B\cdot a & B\cdot b & B\cdot c \\ C\cdot a & C\cdot b & C\cdot c \end{bmatrix} = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \cdot \begin{bmatrix} a & b & c \end{bmatrix}$$

Thus we can divide 3×3 kernel into 3×1 and 1×3 parts.

For image with size $H\times W$ and C channels lets considering 3×3 convolution and

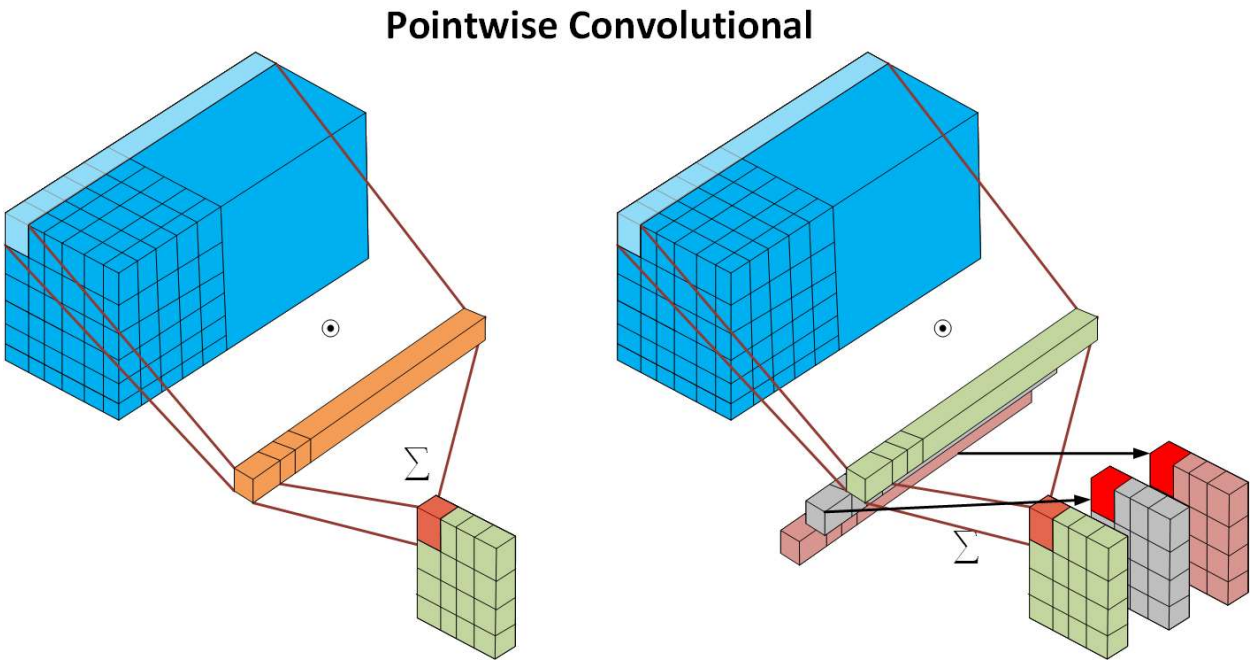
two layers 3×1 and 1×3



- **Pointwise Convolution** (for instance, $1\times 1\times C$).

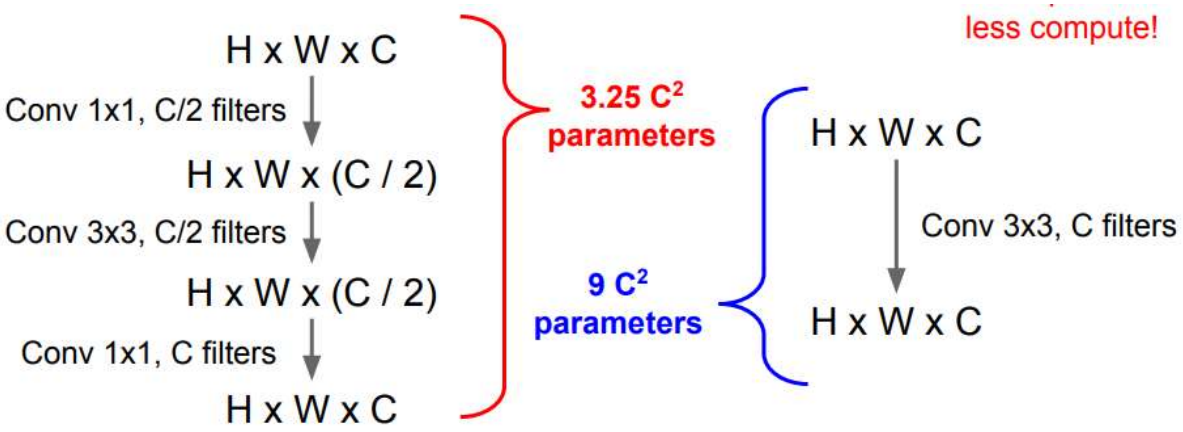
  Reduce the number of channels(feature maps) or increase if it needed.

  The Convolution is also known as **"Bottleneck Convolution"**.



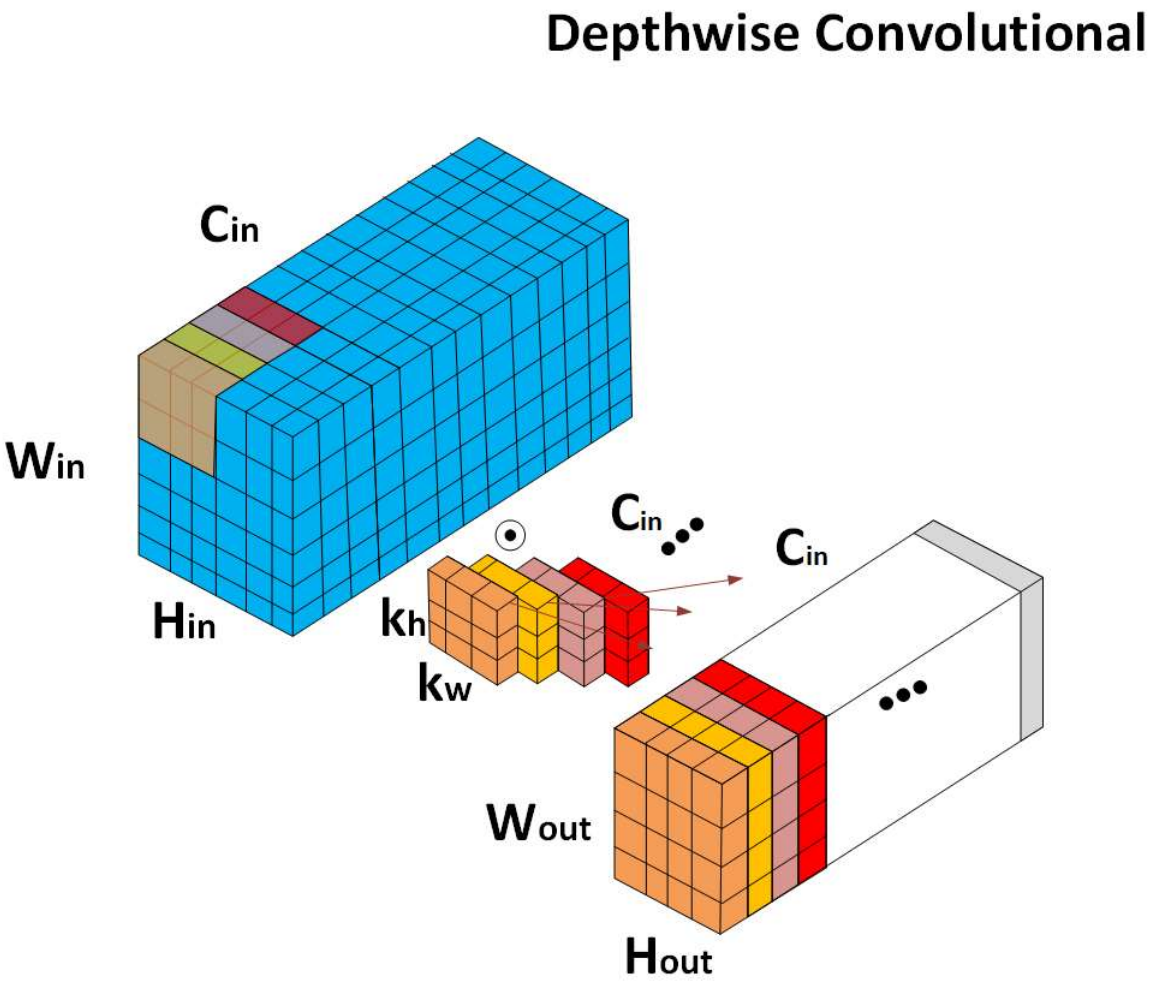
  For image with size $H\times W$ and C channels lets considering 3×3 convolution and

  three-layers bottle-neck $1 \times 1 \times C/2$; $3 \times 3 \times C/2$; $1 \times 1 \times C$ pipeline.



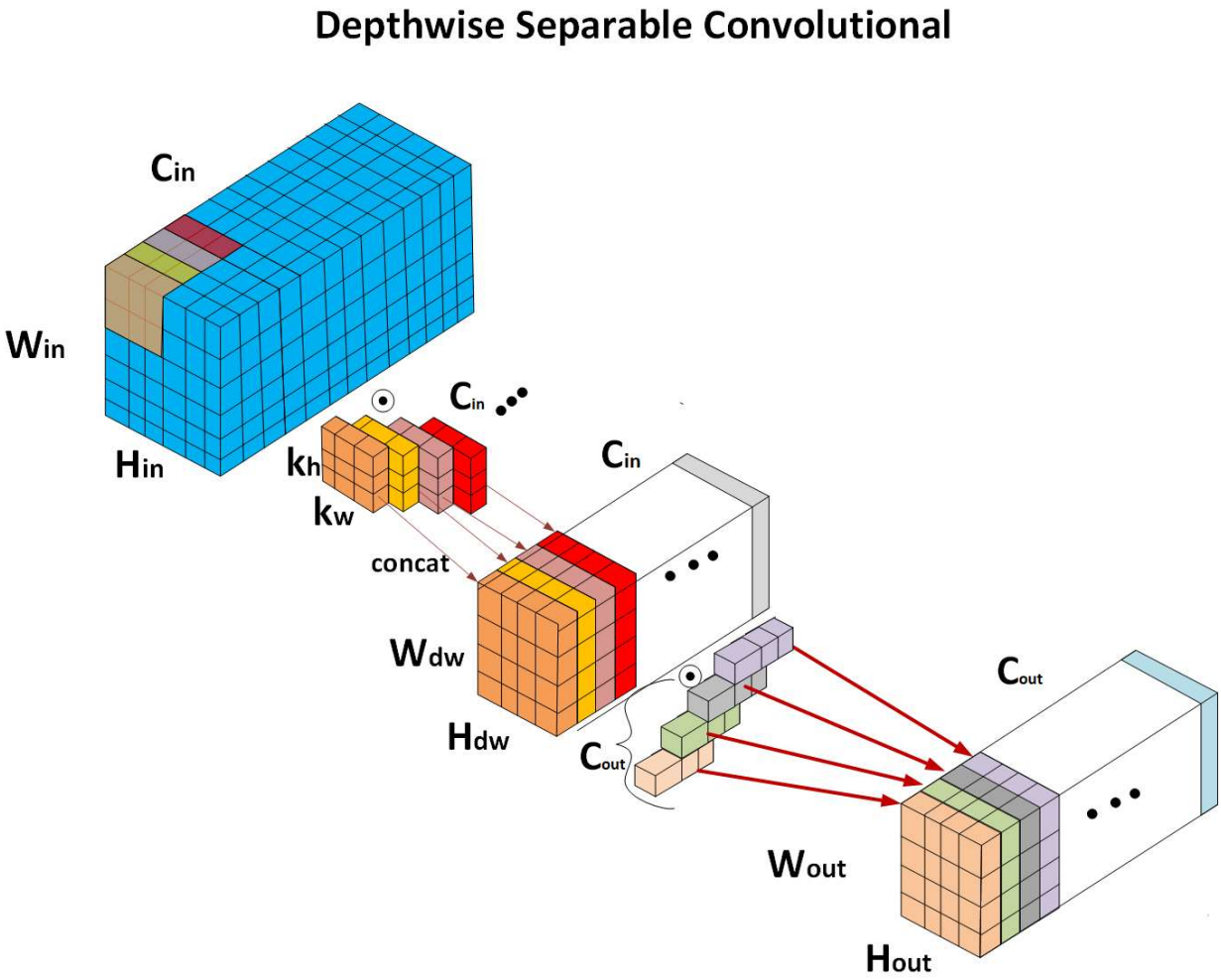
- **Deepwise-Convolution** (for instance, $C \cdot 3\times 3\times 1$).

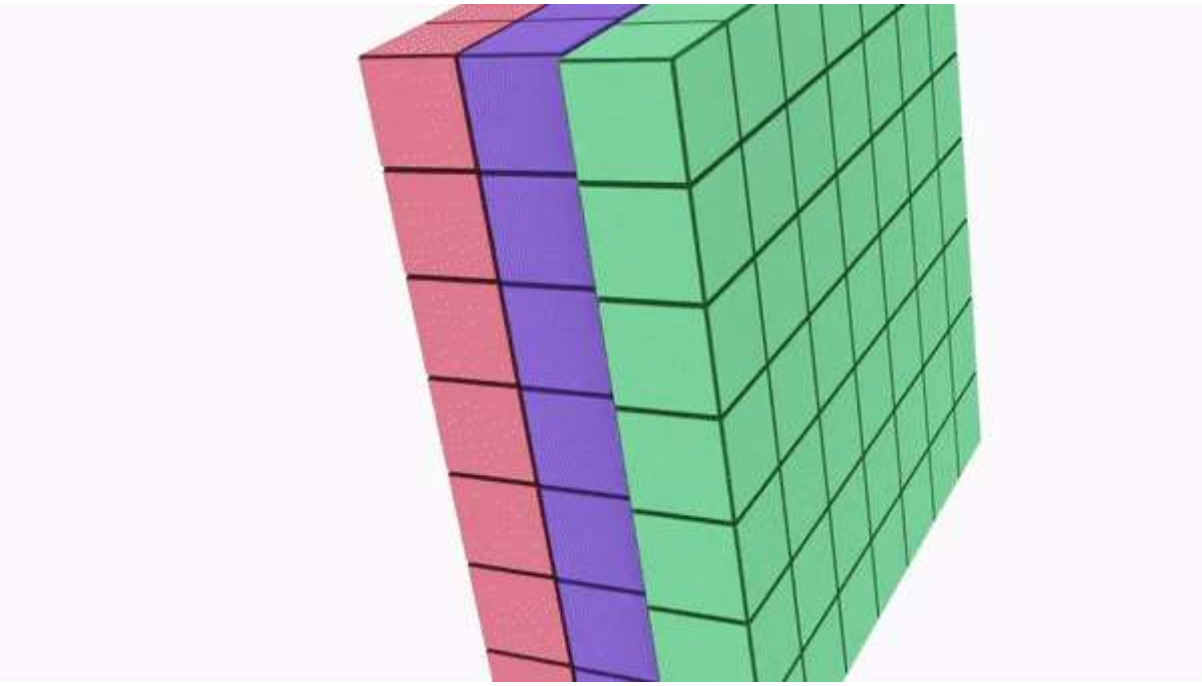
  This is auxiliary (almost always intermediate operation which differs from conventional one in replacing $\sum$ operation with **concatenation**.)



- **Deepwise-Separable Convolution** (for instance, 1-stage $C \cdot 3\times 3\times 1$; 2-stage $C \cdot 1\times 1\times 1$).

  Combination of sequential rectangular convolution and point-wise convolution for reducing number of parameters in comparison with Spatially Separable Convolution





Some explanation of DeepWise Separable Convolution.
It is the one of the most interesting ways to reduce number of parameters in comparison with traditional ones (square, rectangular end e.t.c.).

For traditional convolution the number of parameters is

$$ P_{layer} =C_{out} \times C_{in} \times k_w \times k_h + C_{out} = C_{out}\times P_{kernel}, $$

where:

- P_{layer} is the number of parameters in the layer.
- $ P_{kernel} = C_{in} \times k_w \times k_h + 1$ is the number of parameters for kernel.
- C_{out} is the number output channels.
- C_{in} is the number input channels.
- k_w and k_h is the kernel dimension.

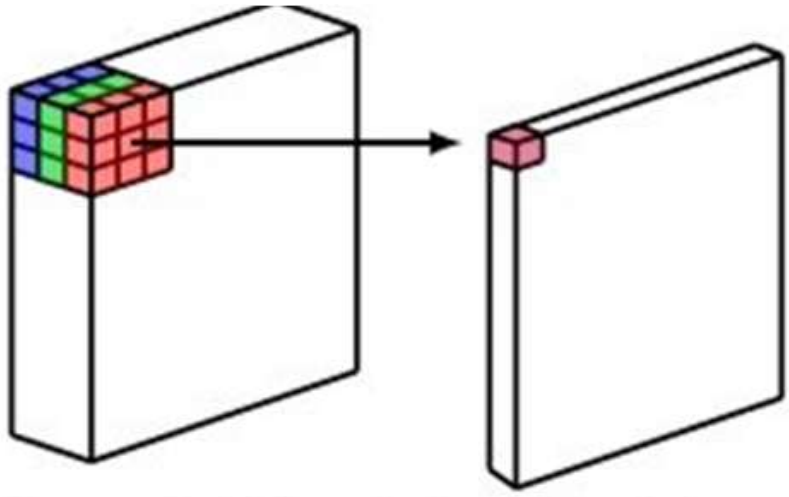
For Deep-wise separable convolution the number of parameters is $$

$$P_{layer} = (C_{in} \times k_w \times k_h + C_{in}) + C_{out} \times 1 \times 1 \times C_{in} + C_{out}= \\= C_{in}\times P_{kernel_{DW}} + C_{out}\times P_{kernel_{point}} , $$ where:

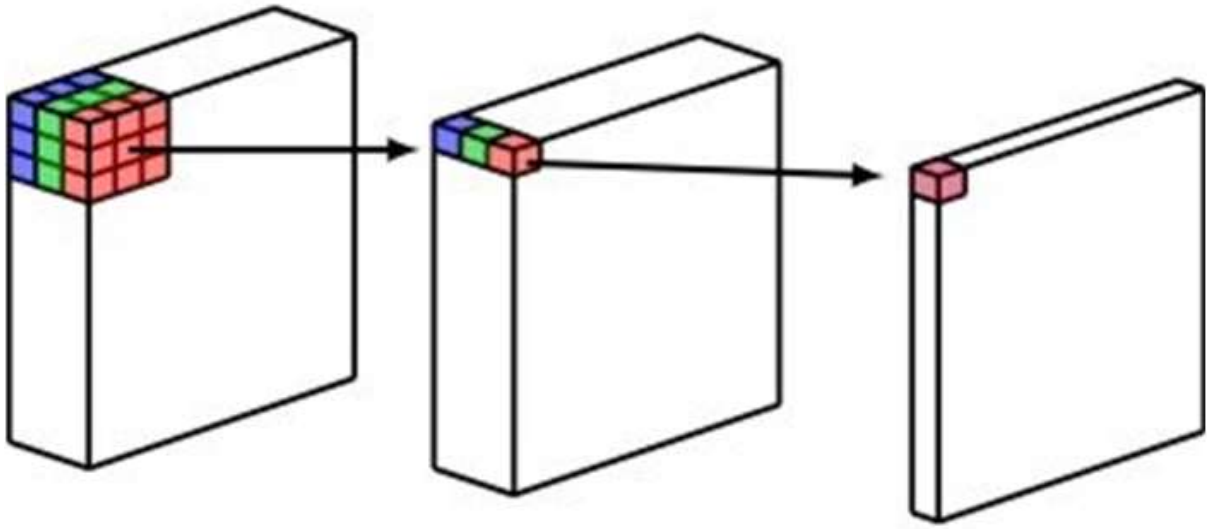
- P_{layer} is the number of parameters in the layer.
- $P_{kernel_{DW}} = \times k_w \times K_h + 1$ is the number of parameters for each Deepwise kernel (input kernel).
- $P_{kernel_{point}} = 1 \times 1 \times C_{in} + 1$ is the number of parameters for the pointwise kernel.
- C_{out} is the number output channels.
- C_{in} is the number input channels.
- k_w and k_h is the kernel dimension.

Thus for common case - kernel 3×3 with 16 input channels and 32 output channels we have:

$$ \frac{ P_{layer_{TR}} } { P_{layer_{DS}} }= $$ $$ = \frac{ C_{out} \times P_{kernel} } {C_{in}\times P_{kernel_{DW}} + C_{out}\times P_{kernel_{point}}}\sim \frac{32\times(16\times 3\times 3+1) } {16\times(3\times 3+1)+32\times 17}=\frac{4640}{704}=6.5 $$



(a) Conventional Convolutional Neural Network

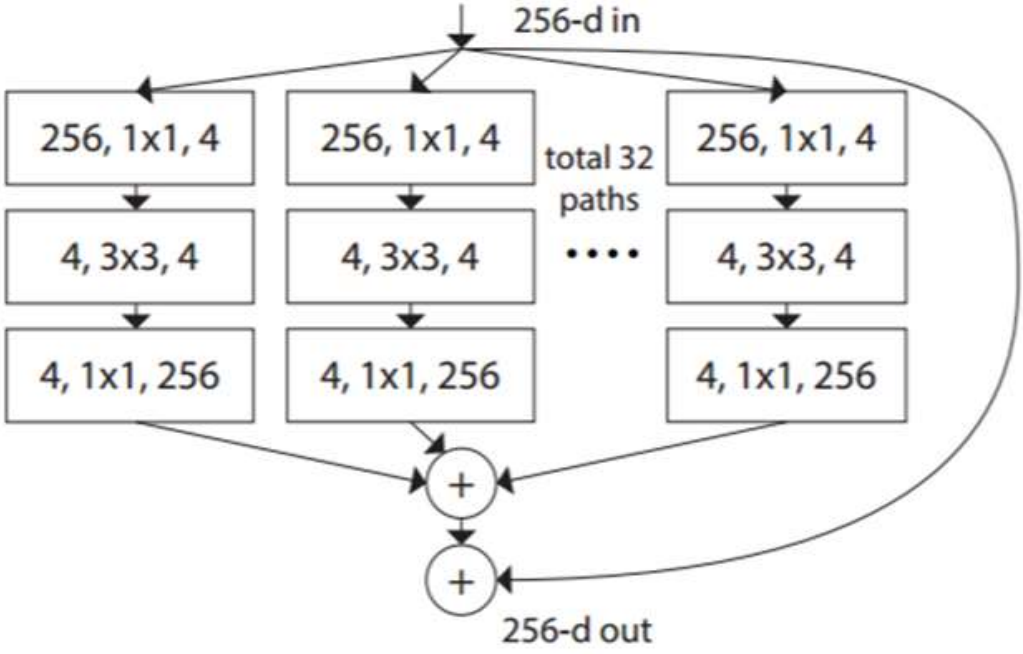


(b) Depthwise Separable Convolutional Neural Network

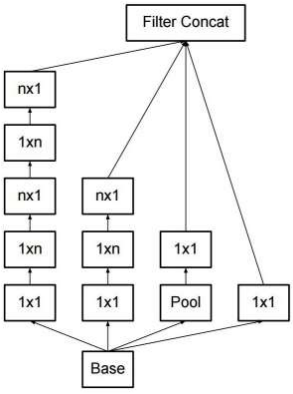
- **Network in Network Convolution** (for instance, $ \text{concat}[3\times 1\times C, 1\times 3\times C, 1\times 1\times C, \text{skip}]$).

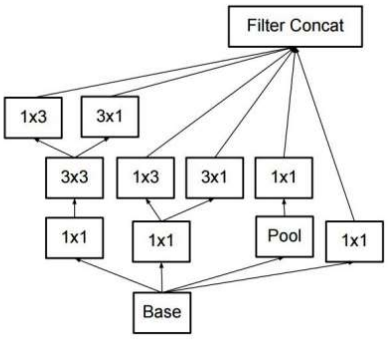
    An modification of layer idea as built in a combination of grouped, dialed and other variants of convolutions (including skip-connection one) to increase the feature extraction.

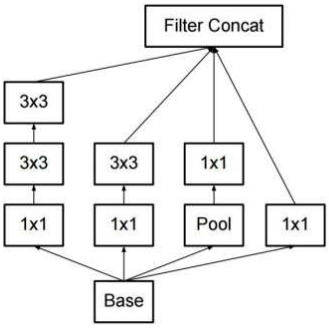
Network in network convolution



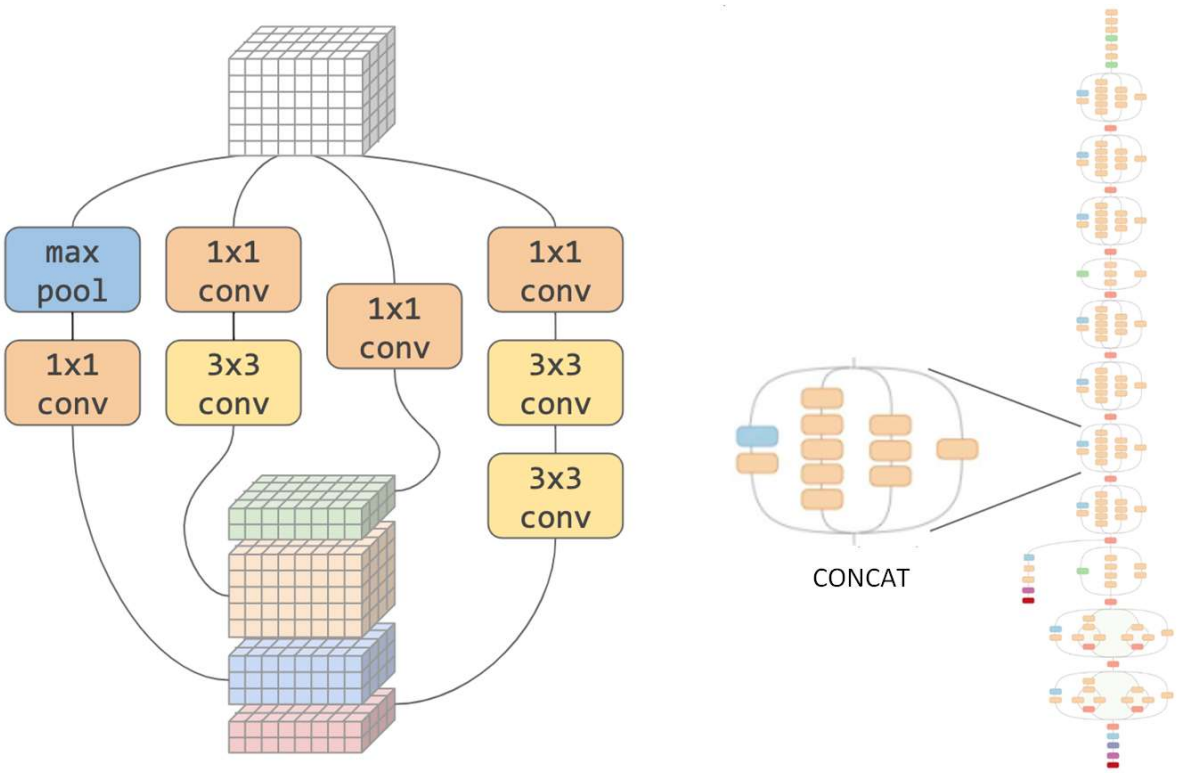
other implementations





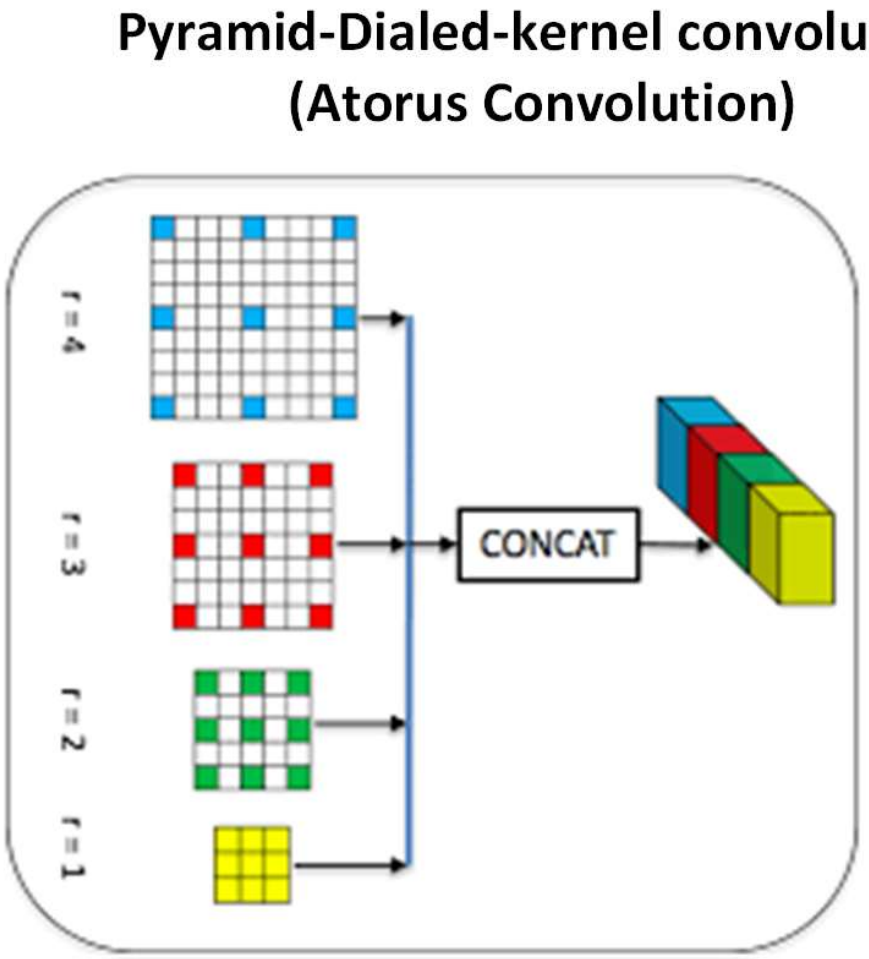


It also can be concatenation inplace of summation



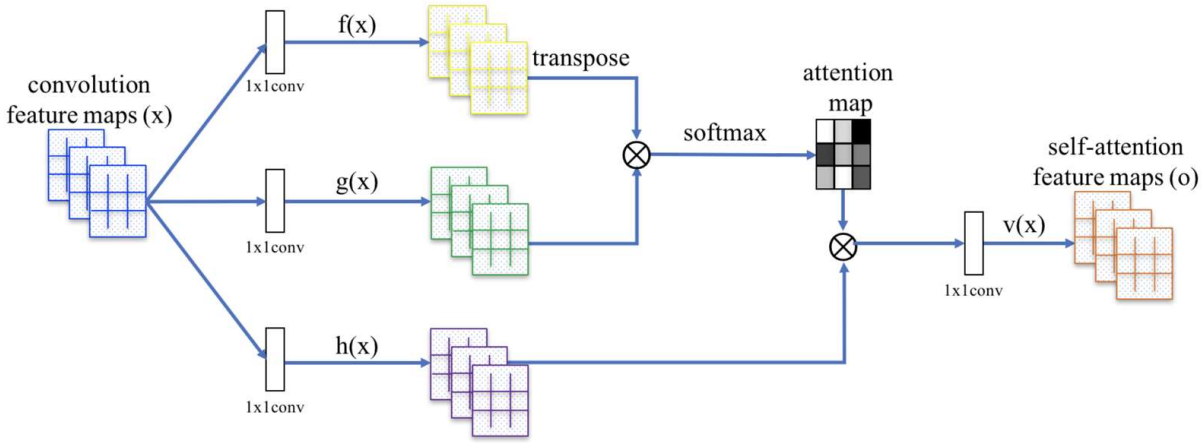
- **Pyramid-Dialed-kernel Convolution** (for instance, $\text{concat}[3\times 3\times C \text{, rate=1 }; 3\times 3\times C \text{, rate=2 }; 3\times 3\times C \text{, rate=4 }]$).

  An modification of dialed convolution idea as Network in Network Convolution combination of several rate (including skip-connection) to to increase the feature extraction.



- **Attention convolution** (for instance, $[3\cdot 1\times 1\times C+ 1\times 1\times C]$).

  The main idea is to produce dynamically change weights that trained for some feature highlighting.



Key: $K=f(x)=W_fx$

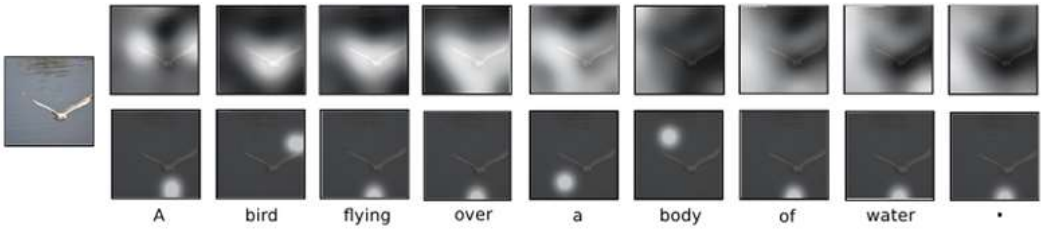
Query: $Q=g(x)=W_gx$

Value: $V=h(x)=W_hx$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{size}})\mathbf{V}$$

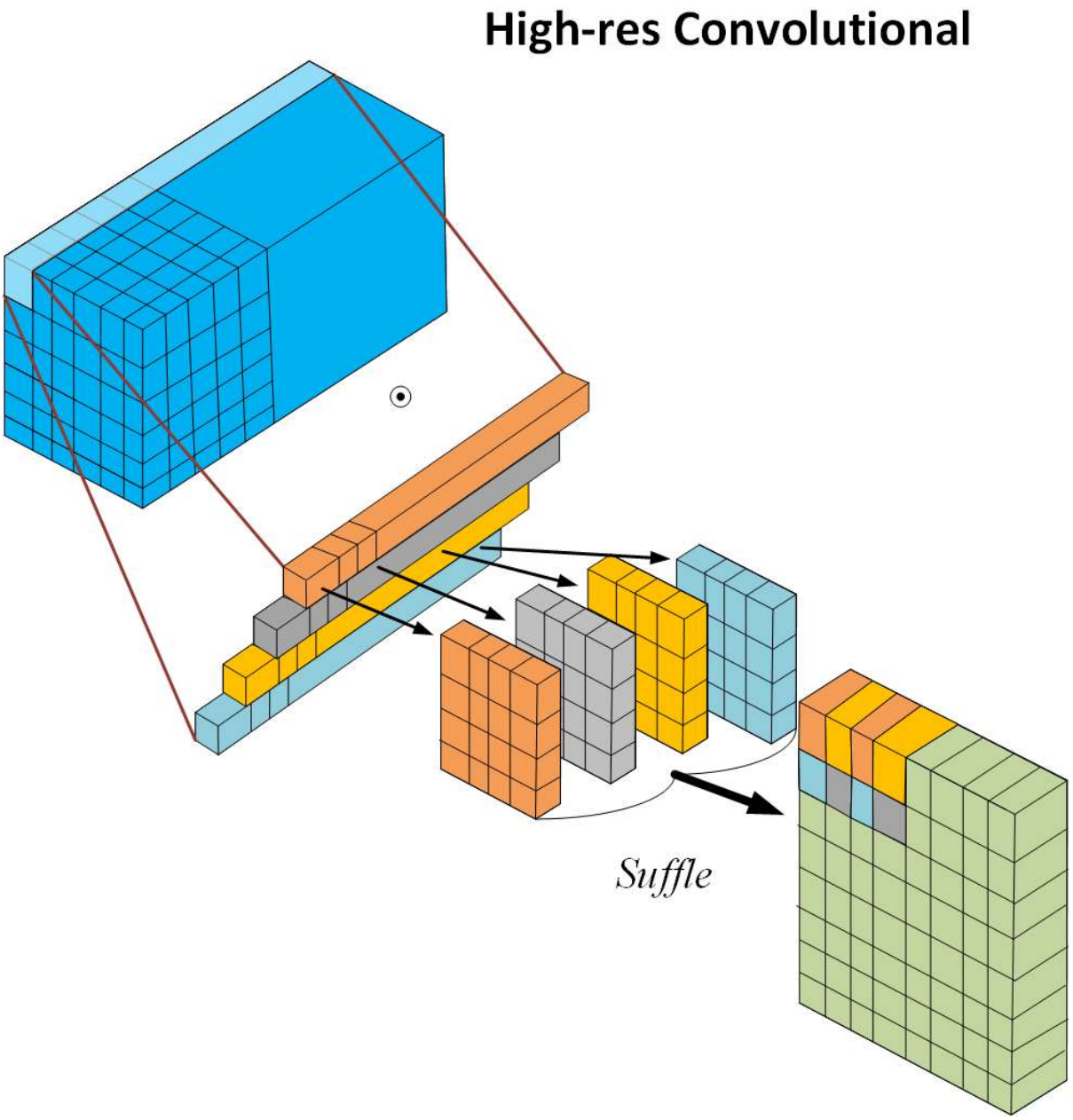
Example of how attention work for different classes

Example of how attention work for image description



- **Pixel-shuffle (high resolution) convolution** (for instance, $\text{shuffle}[9\cdot 3\times 1\times C]$).

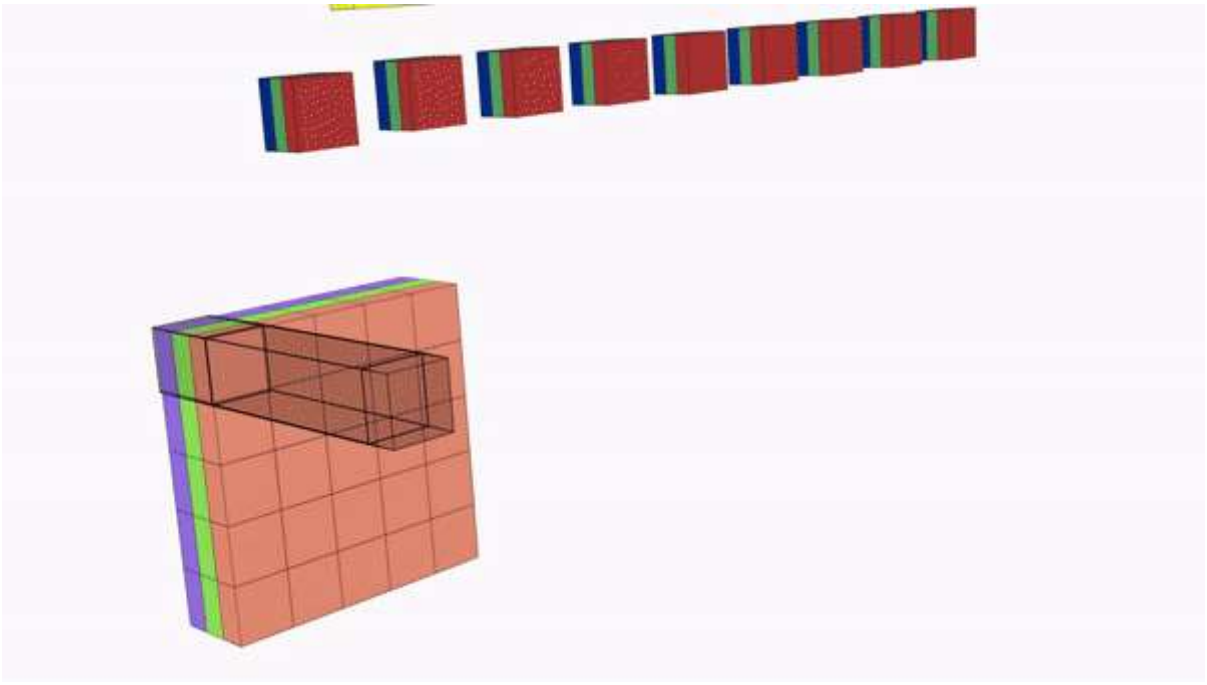
  The main idea is to increase output dimension by shuffling feature extracted by several 2d-kernels.



Some explanation of pixel shuffle.

In fact your need to take point-wise convolution first for extract $N=C\times C$ number of channels (for image or feature map with $W\times H\times C$ dimension).

From these $1,2,3,..., N$ outputs of point-conv, where N is CxC, take one pixel at a time from each consecutive layer from $1,2 \text{ to } N$ and then shuffle them to get the output.



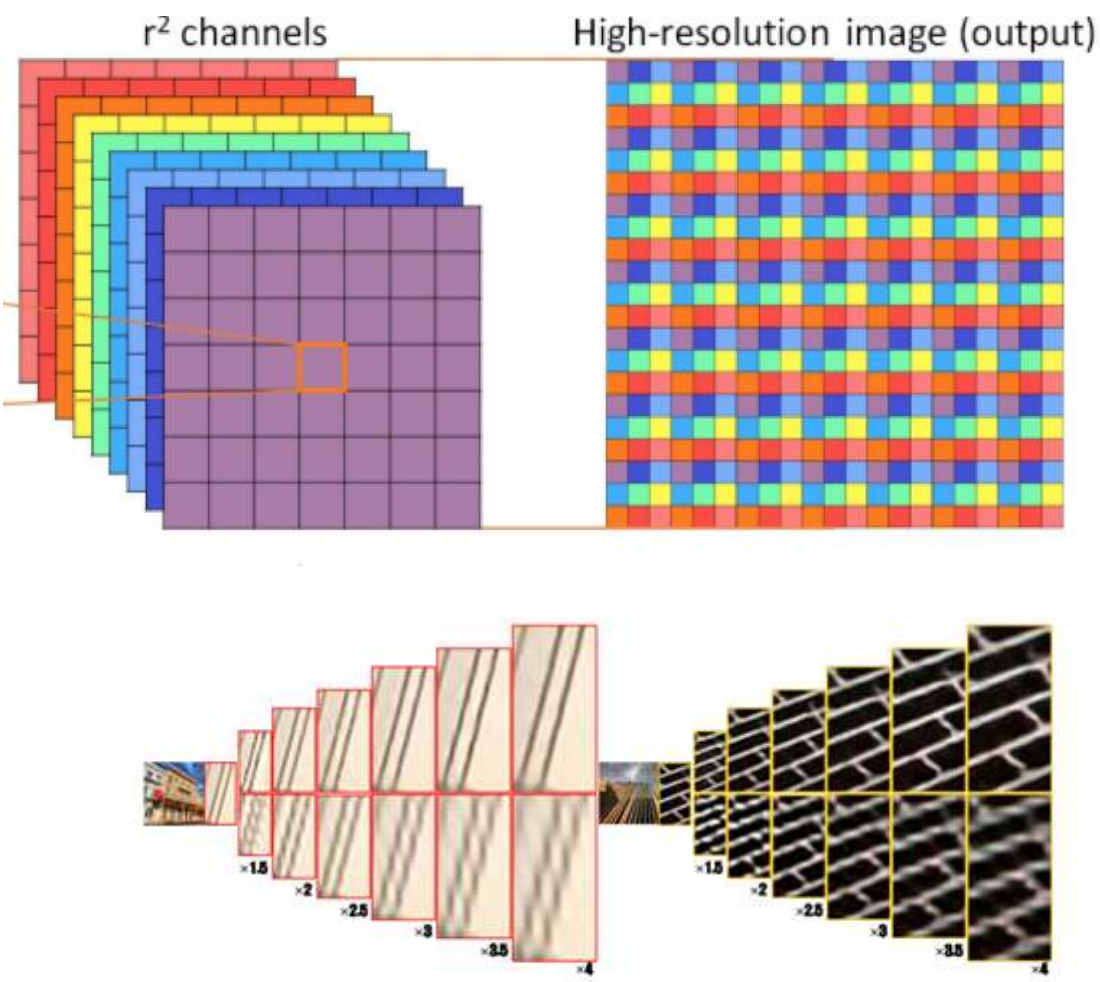


Figure 5: (Top) Our results using a single network for all scale factors. Super-resolved images over all scales are clean and sharp. (Bottom) Results of Dong et al. [5] (×3 model used for all scales). Result images are not visually pleasing. To handle multiple scales, existing methods require multiple networks.

- The reverse to the convolution operation is the **Transposed Convolution**.

  The transposed convolution is also called **de-convolution** or **Fractionally Strided Convolutions**.

  The transposed convolution is widely using in many application, starting from:

  - backward propagation in CNN model training.
  - reverse convolution (de-convolution) in segmentation learning (and auto-encoders).
  - up-sampling technique.

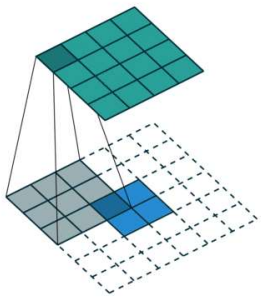
  For the first let's re-write **straight convolution** in matrix form as did it before

  $$\text{straight convolution: }r = k\ast x = \boldsymbol{k}\cdot \vec{x},$$ Then we can reconstruct the input by **pseudo-inverse matrix operation as:** $$\text{pseudo-inverse convolution: } \vec{x} =

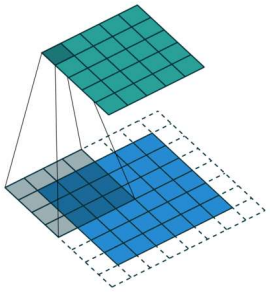
\frac{\boldsymbol{k^T}\cdot r}{\boldsymbol{k}\cdot \boldsymbol{k^T}} $$ However, in fact we do not need to take denominator due to our reason is to increase the dimension.

so the **transposed convolution** can be given as: $$\text{transposed convolution: } \vec{x} = \boldsymbol{k^T}\cdot r $$

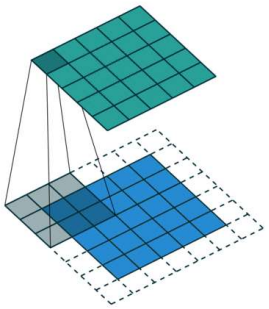
Transposed convolutions



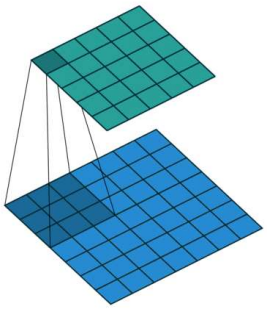
No padding, no strides



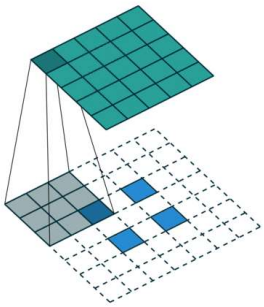
Arbitrary padding, no strides



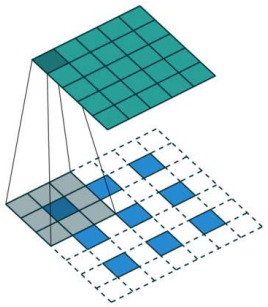
Half padding, no strides



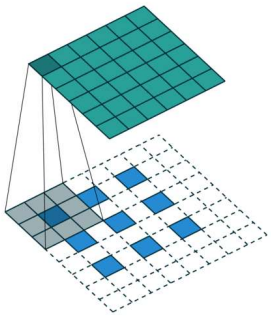
Full padding, no strides



No padding, strides



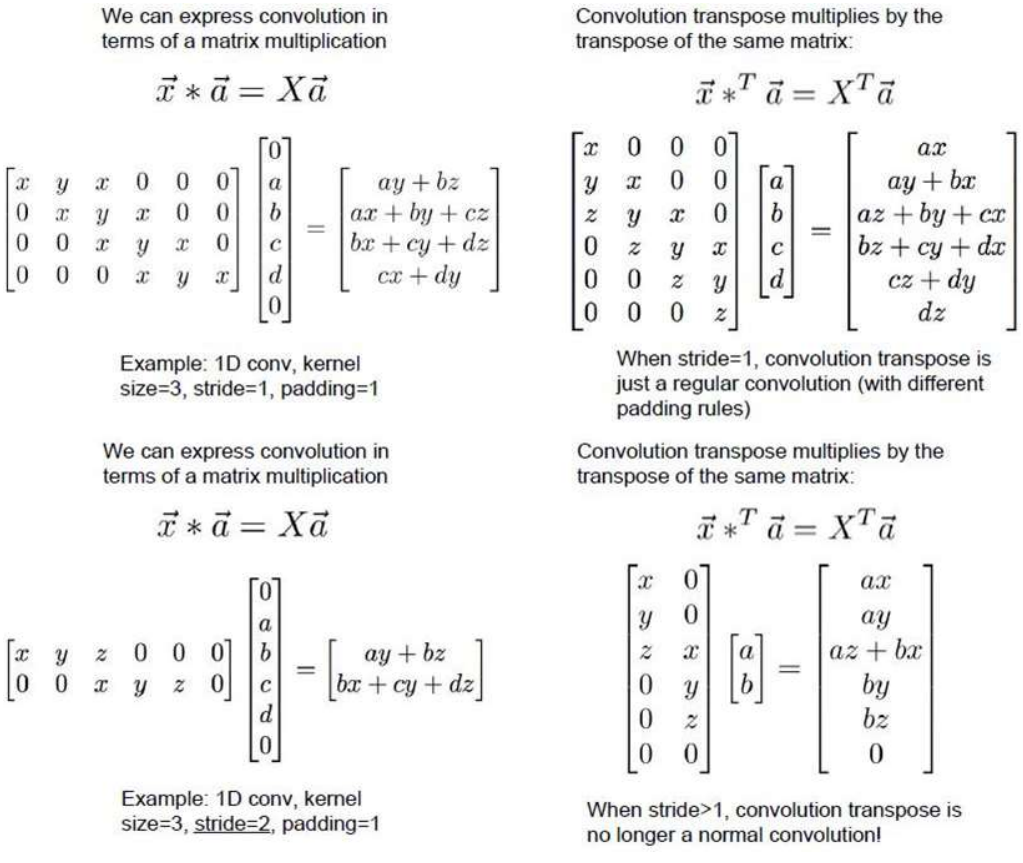
Padding, strides



Padding, strides (odd)

Note In the case of padding or striding the input image should be transformed to the expected one by adding zeros element (white in the figures above).

Note Examples of straight and transposed convolution for 1d matrix with stride 1 and 2



> *Note* the full list of other not-popular convolutions types can be found here (https://paperswithcode.com/methods/category/convolutions)

In [ ]:

```
```