Exercise 7

High Performance Computing for Science and Engineering

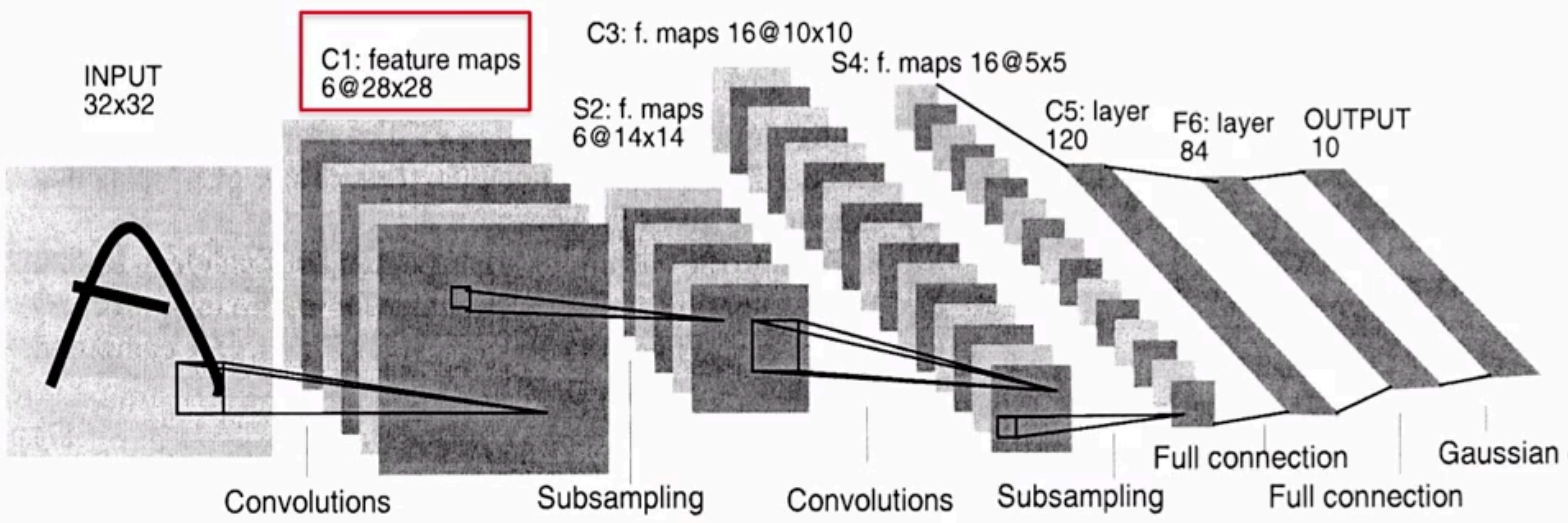
Guido Novati

November 10, 2017

CSElab
Computational Science & Engineering Laboratory
http://www.cse-lab.ethz.ch

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

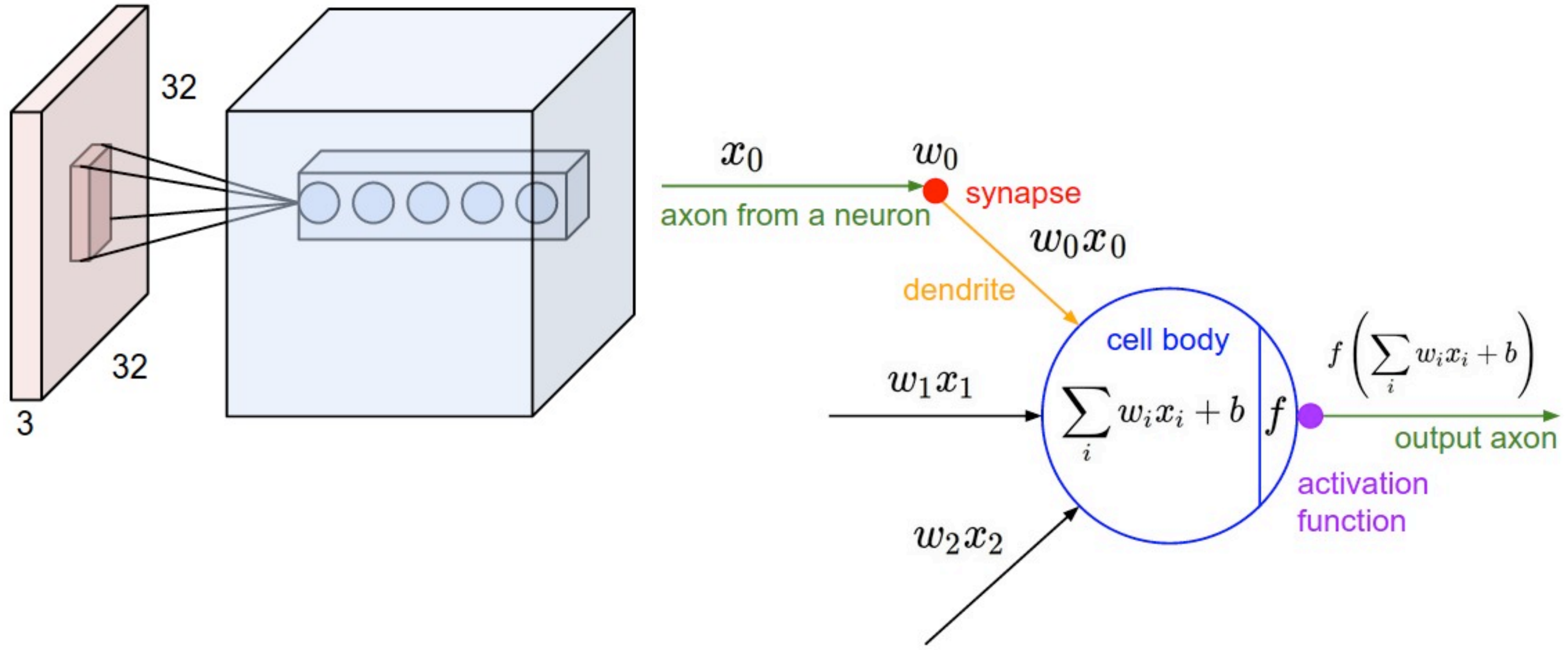
Convolutional Neural Networks



LeNet of Yann LeCun et al., 1998

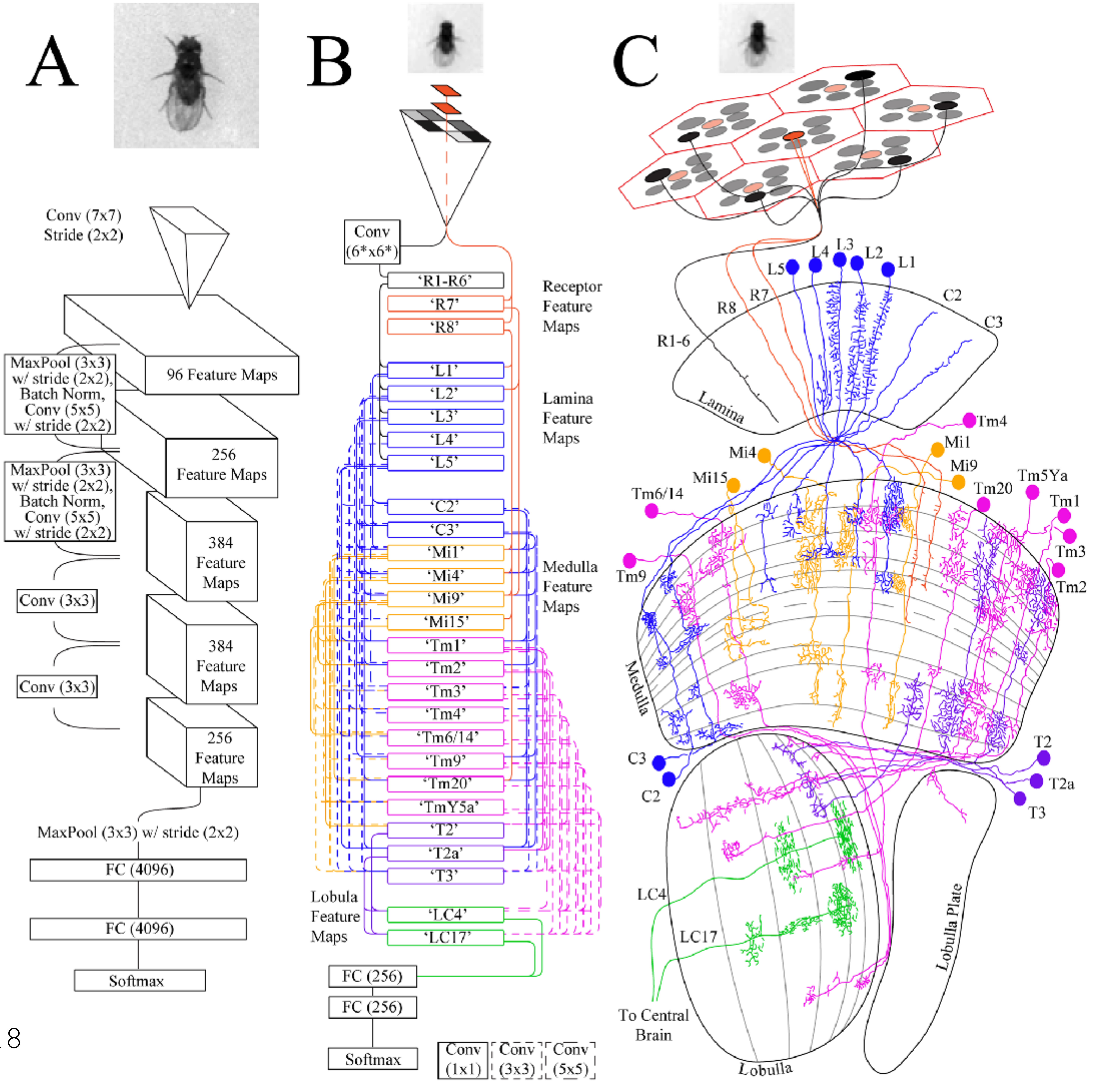
- Before deep learning, AI meant expertly designed "if" statements
- With deep learning, AI means GEMM.
- We (HPC people) are pretty good at doing GEMM.
- CNN are the backbone of recent hype in "deep learning"
- Parametric models that are well suited to classify / recognise image contents

Biological Intuition



- **Very** roughly speaking, biological brains have neurons that activate when they recognize a triggering pattern in their inputs.
- Each unit does "simple" pattern recognition. Complexity emerges from sheer numbers

Convolutional model of **a part of** the Fruit-fly's brain

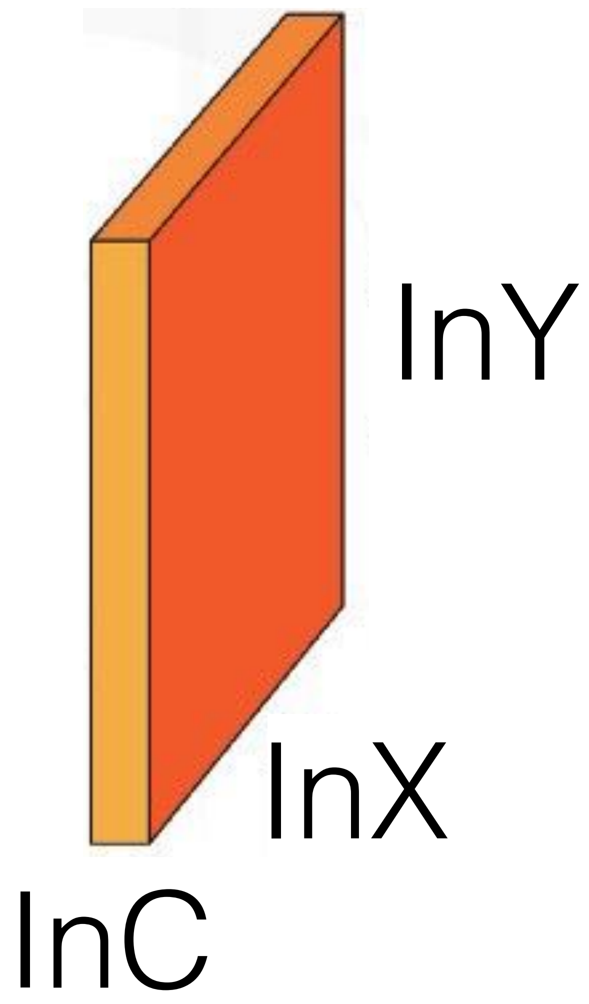


Jonathan Schneider et al., 2018

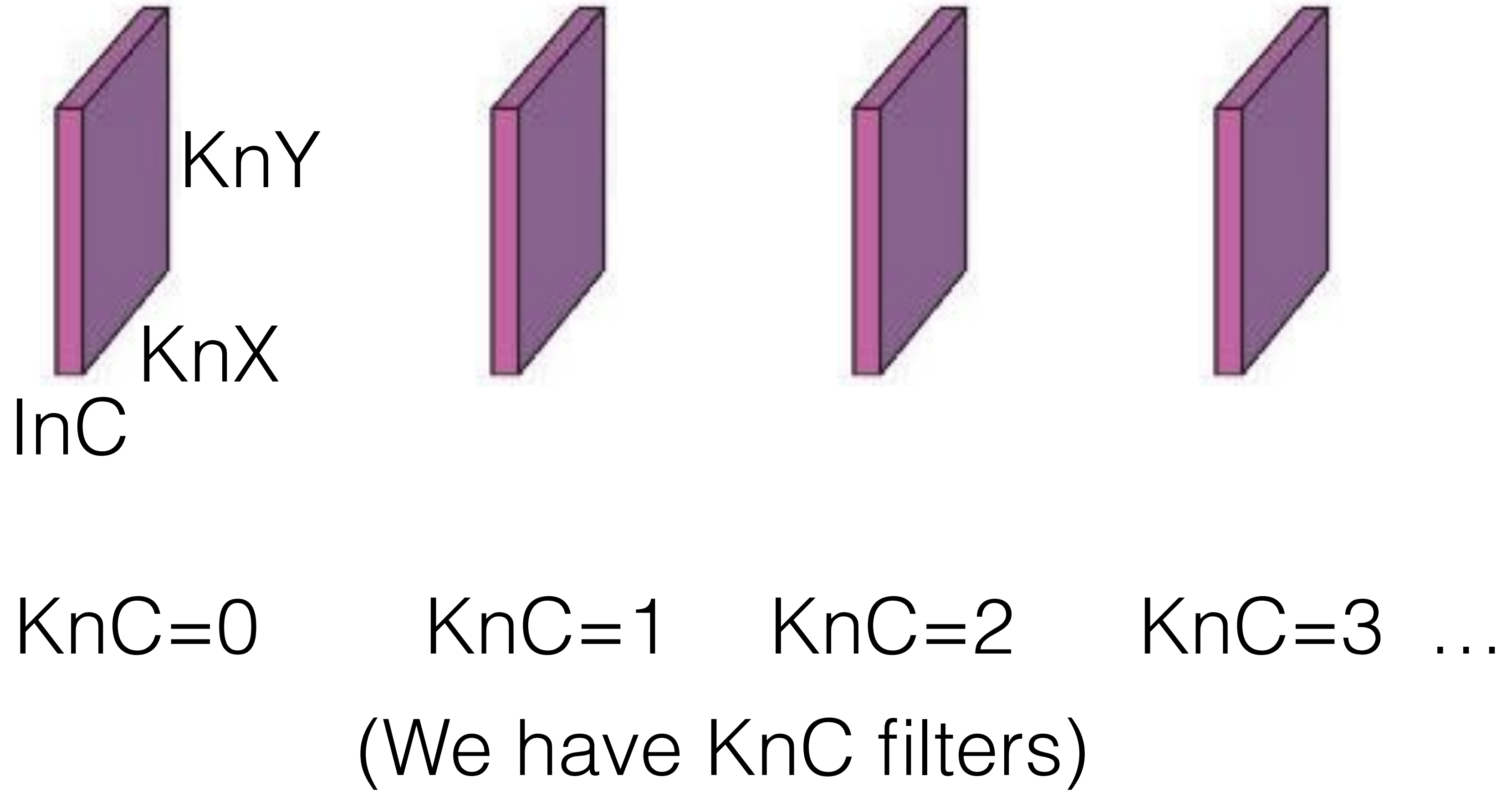
What is a Convolution (in machine learning) (1)

- A convolution is a parametric operation that maps an "image" to an other "image"

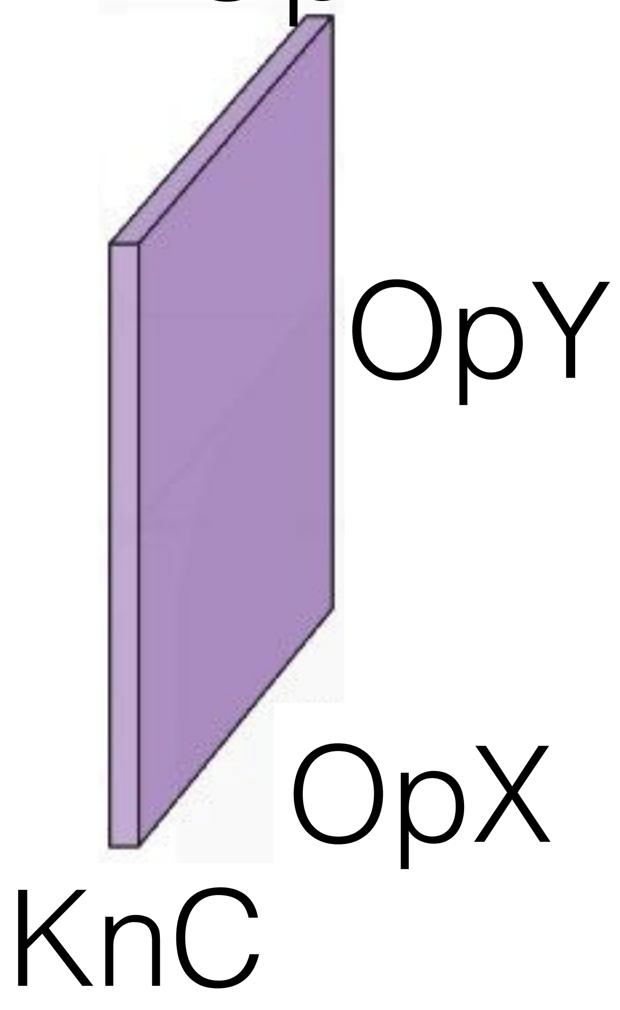
Input is a matrix :
InY * InX * InC



Parameters are a matrix:
KnY * KnX * InC * KnC



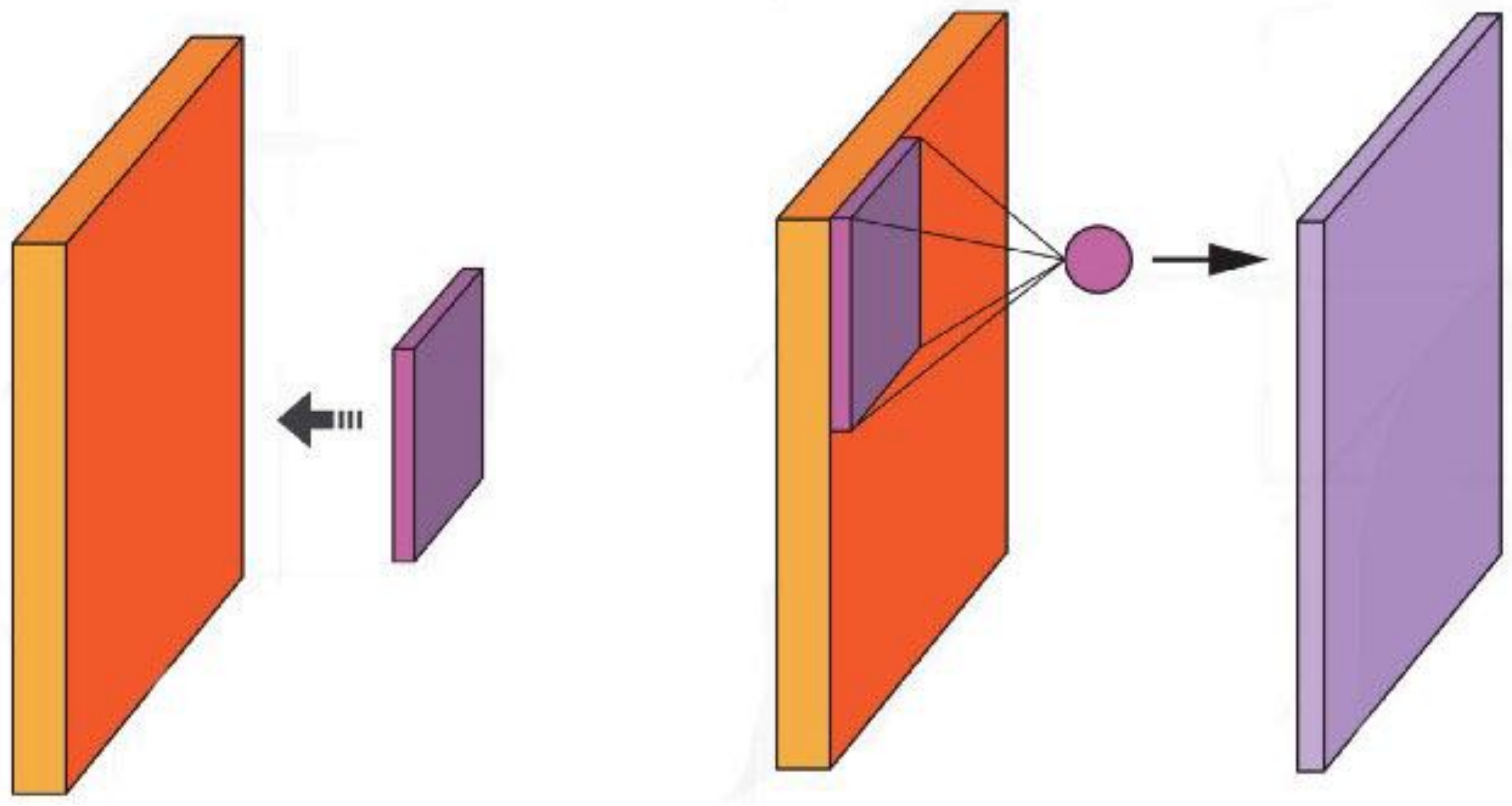
Output is a matrix :
OpY * OpX * KnC



- The third dimension are the "color" channels (or feature maps)
- InC and KnC can be any number
- Parameters are called "filters" or "kernels"

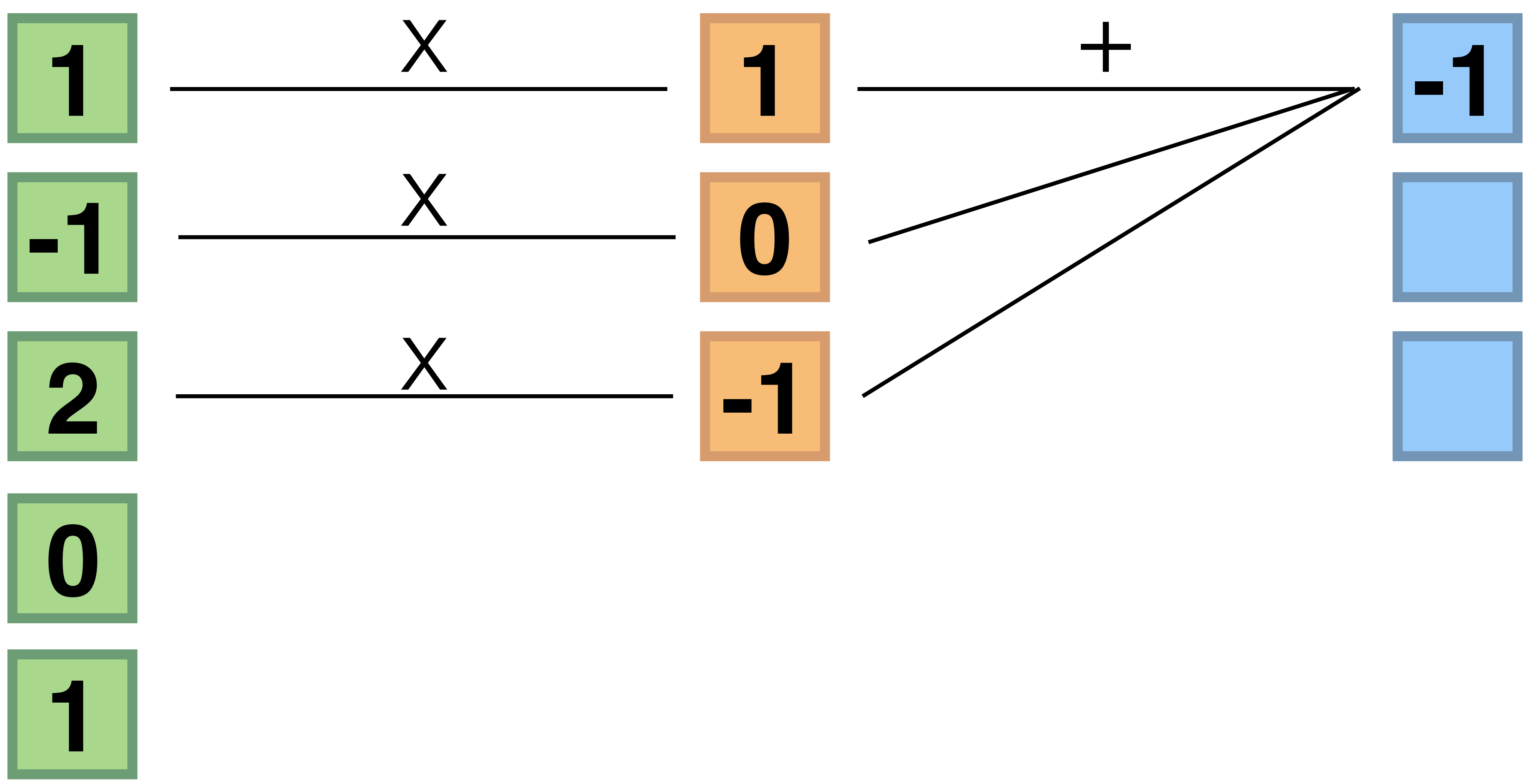
What is a Convolution (in machine learning) (2)

- Convolution is performed by iterating along X and Y
- At each position, and for each filter, we compute the scalar product between the filter and a patch of the image (KnY * KnX * InC)
- The output of the scalar prod is a number which is written onto one color pixel of the output image



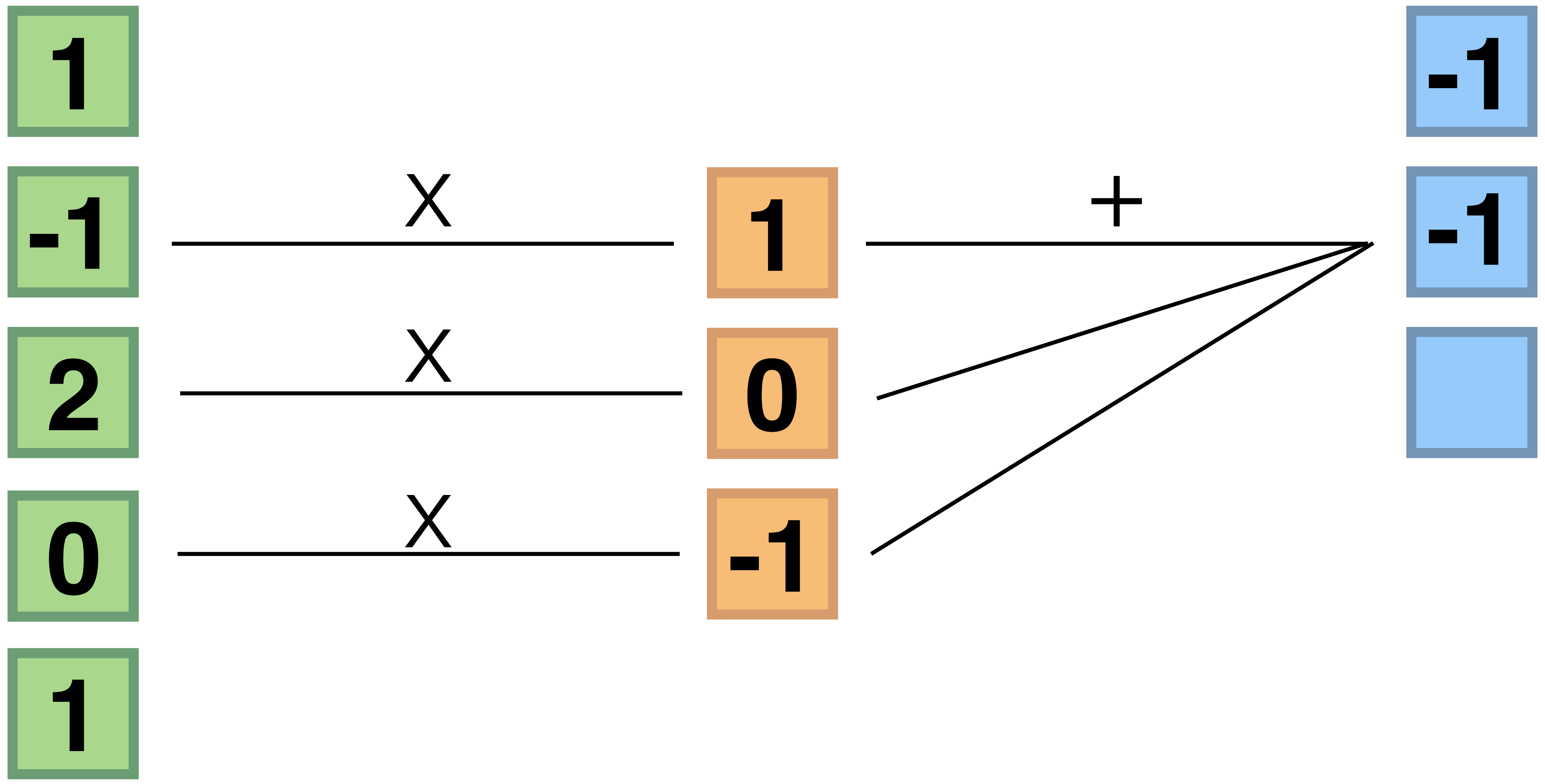
1D, 1 Filter

- Let's apply convolution in 1D:



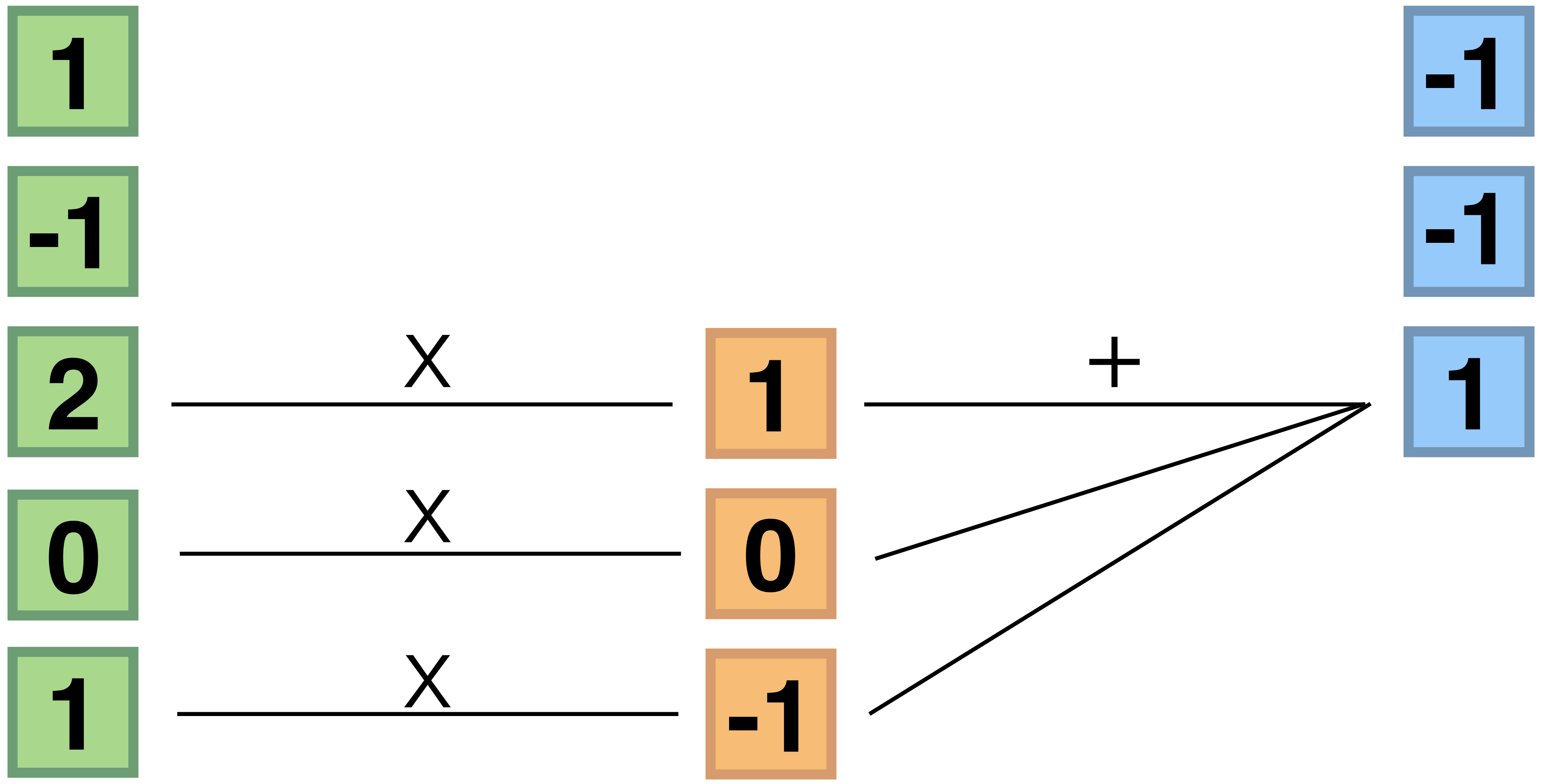
1D, 1 Filter

- Let's apply convolution in 1D:



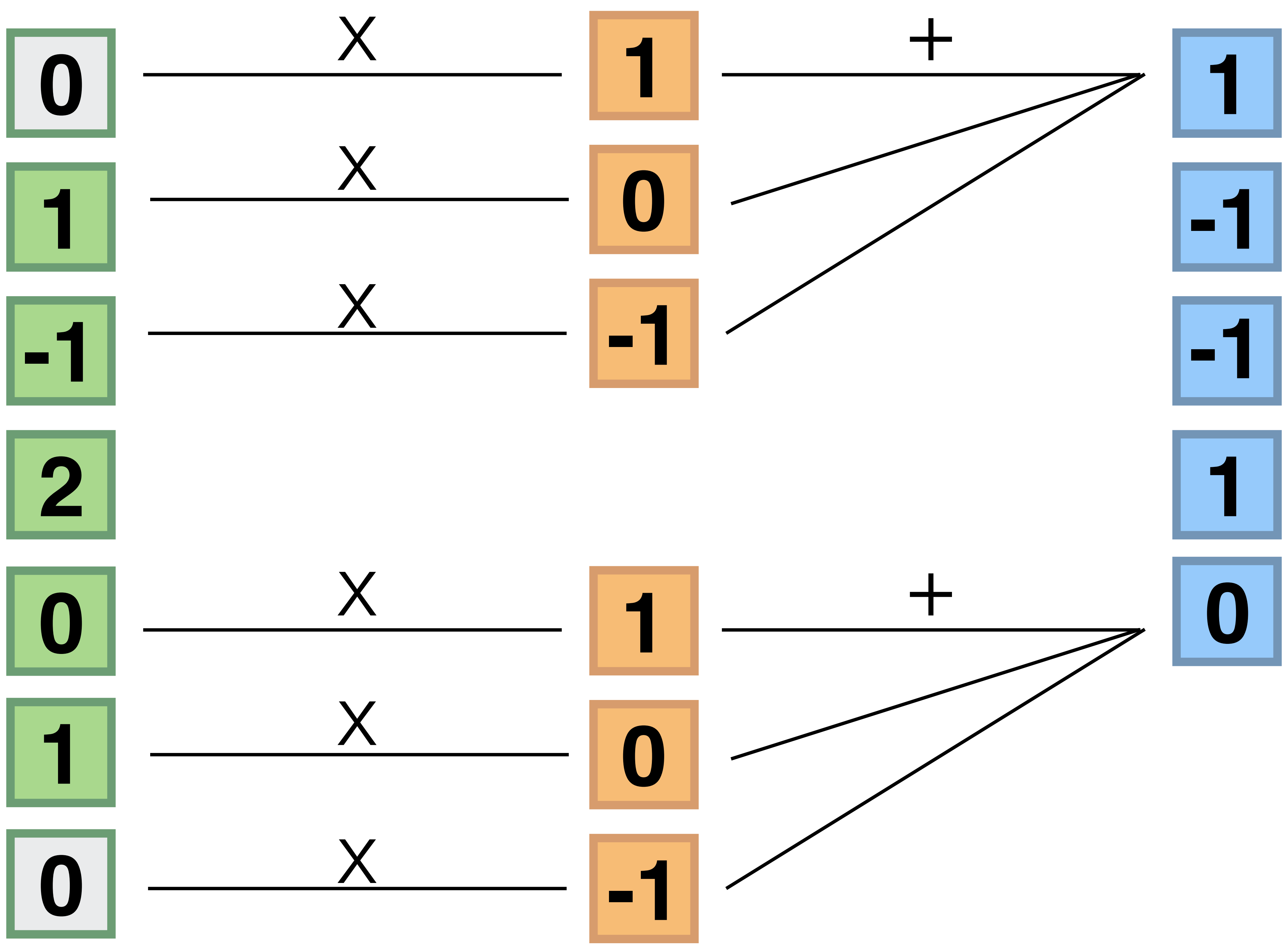
1D, 1 Filter

- Let's apply convolution in 1D:



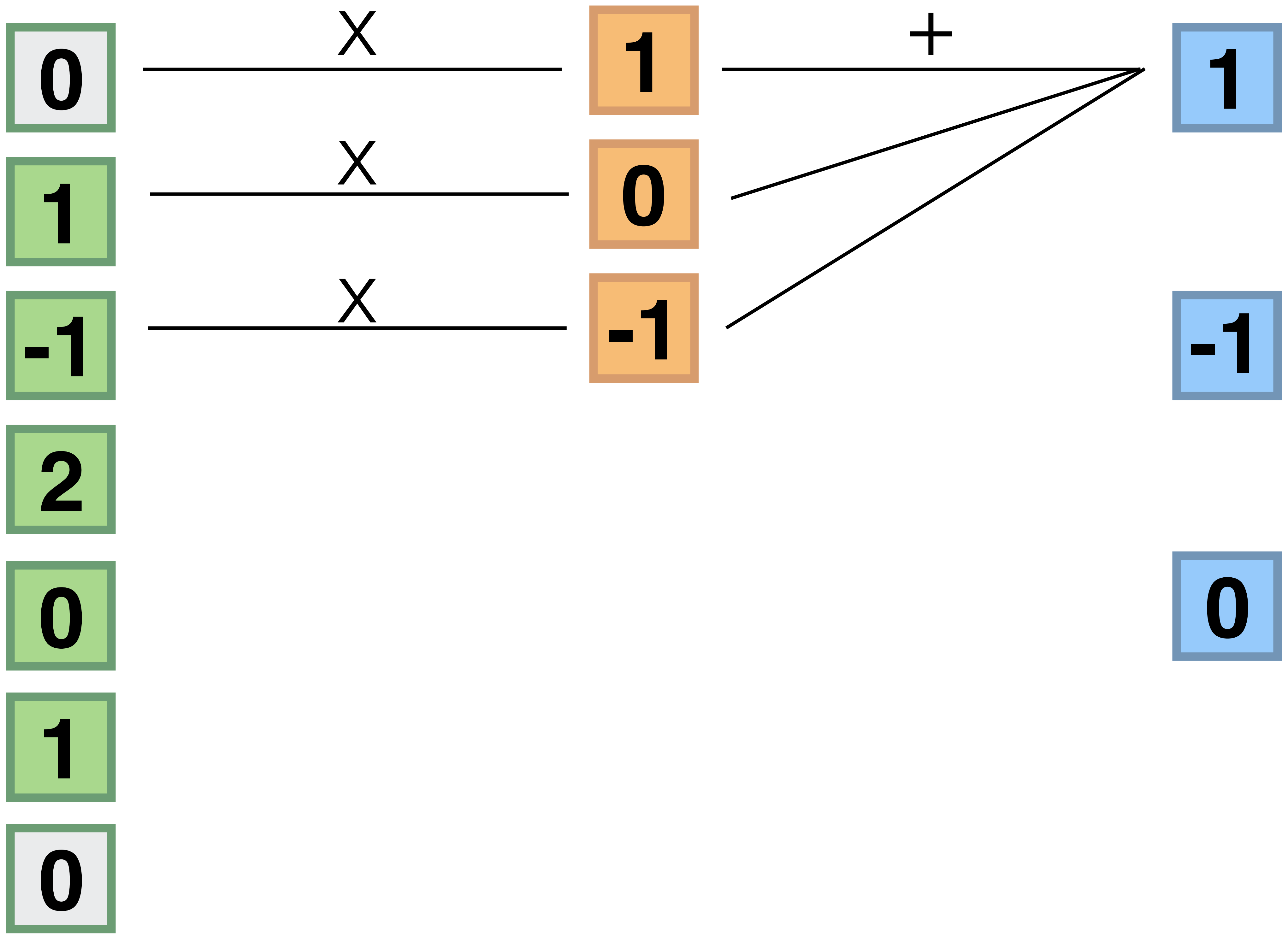
1D, 1 Filter

- Padding:
  - Size of the image is extended in both directions by Py and Px
  - Padding pixels usually carry value 0 (“zero padding”)



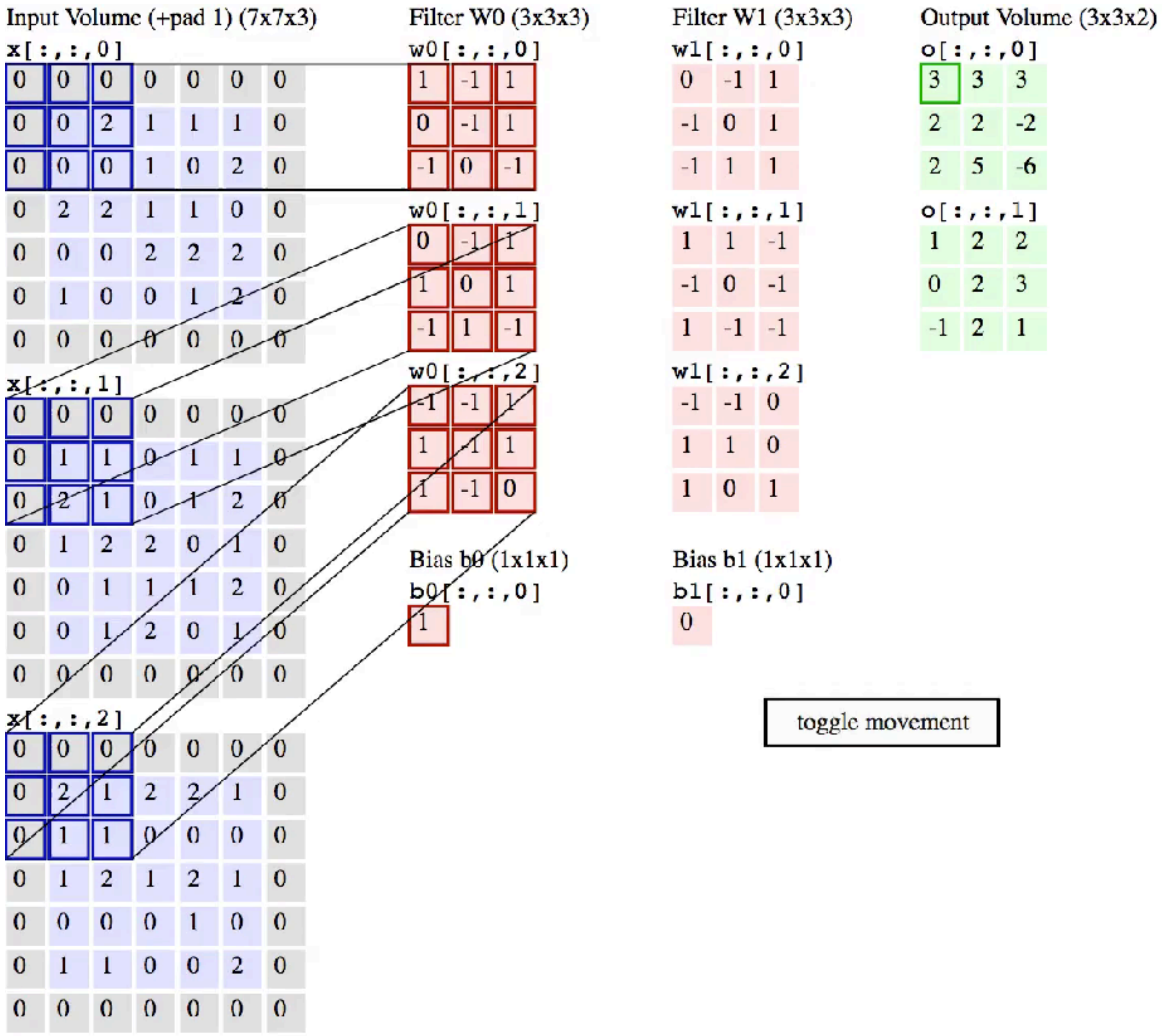
1D, 1 Filter

- Stride:
  - Convolution does not need to be computed by iterating with increments of 1 pixel
  - Here: padding 1, stride 2:



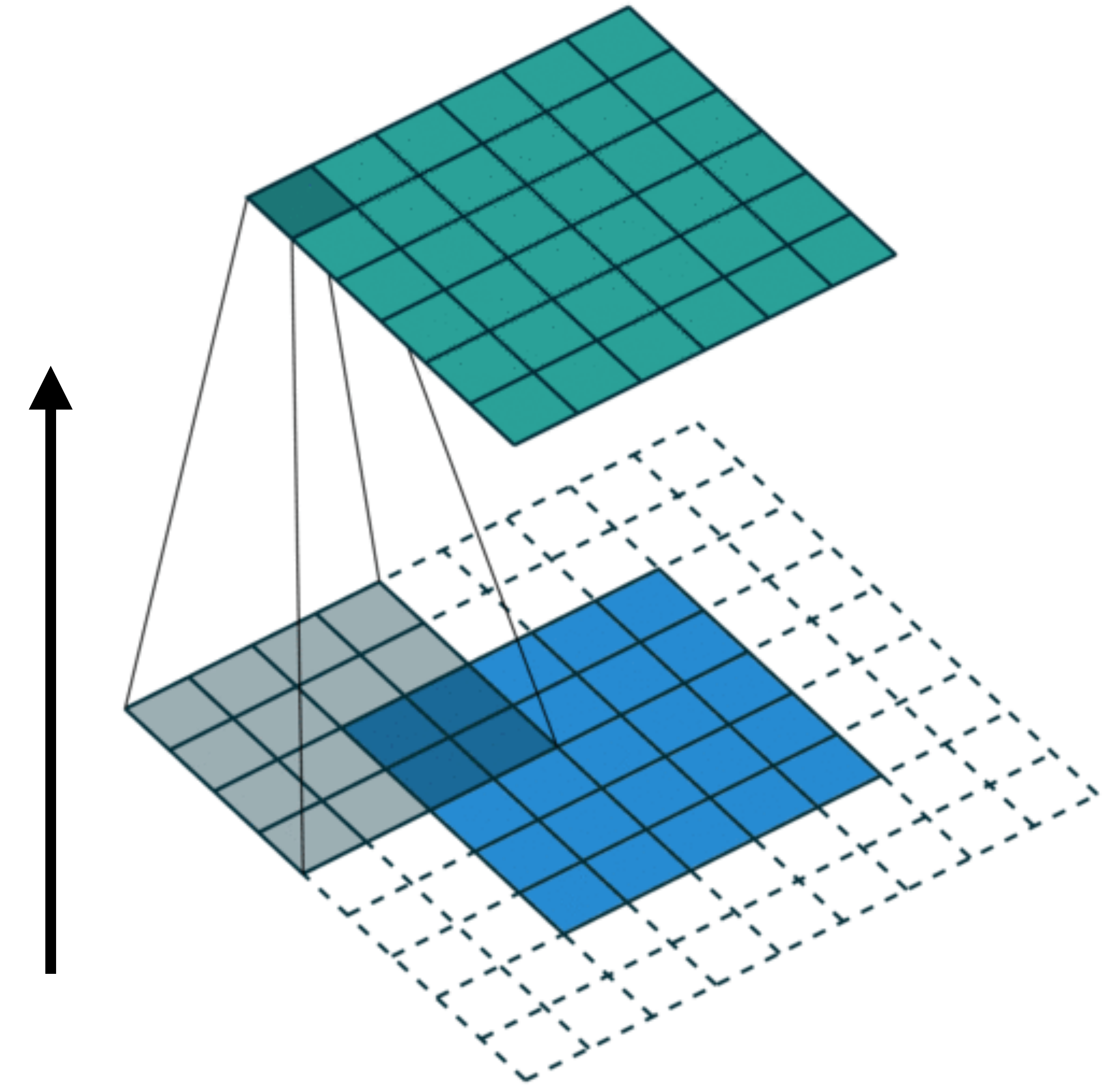
2D convolutions rules of thumb

- If the input image is InY * InX * InC
- The Filters are KnY * KnX * InC * KnC
- With strides Sy and Sx
- And padding Py and Px
- The output image has size:
    - OpY = (InY - KnY +2Py)/Sy + 1
    - OpX = (InX - KnX + 2Px)/Sx + 1
    - KnC
- These numbers must be integers!
  This requirement may force you to tweak the parameters of the convolution



2D convolution forward loop:

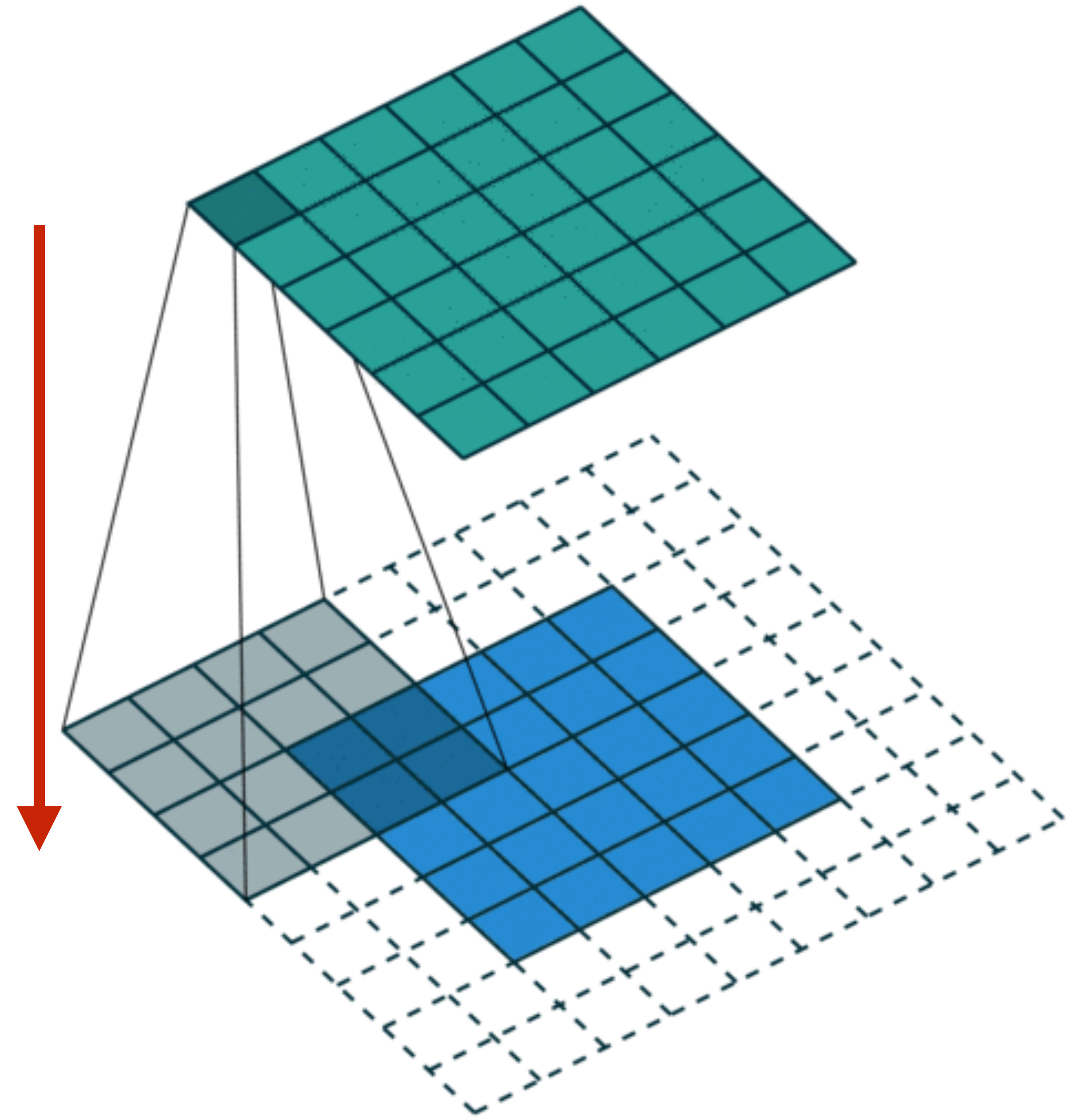
- forward operation requires iterating through input and computing the inner product



```
for (int oy = 0; oy < OpY; oy++) // loop over output image Y
for (int ox = 0; ox < OpX; ox++) // loop over output image X
{
  for (int fy = 0; fy < KnY; fy++) // loop over filter Y
  for (int fx = 0; fx < KnX; fx++) // loop over filter X
  {
    //index along input map of the convolution op:
    const int ix = ox * Sx - Px + fx;
    const int iy = oy * Sy - Py + fy;
    //padding: skip addition if outside input boundaries
    if (ix < 0 || ix >= InX || iy < 0 || iy >= InY) continue;

    for (int bc=0; bc<BS; bc++) // loop over batch
      for(int ic=0; ic<InC; ic++) // loop over inp feature maps
        for(int fc=0; fc<KnC; fc++) // loop over filters
          OUT[bc][oy][ox][fc] += INP[bc][iy][ix][ic] * K[fy][fx][ic][fc];
  }
}
```

2D convolution backward loop (1):



- As for linear layers, we want to compute two things:
  - Gradient of loss wrt. input
  - Gradient of loss wrt. parameters

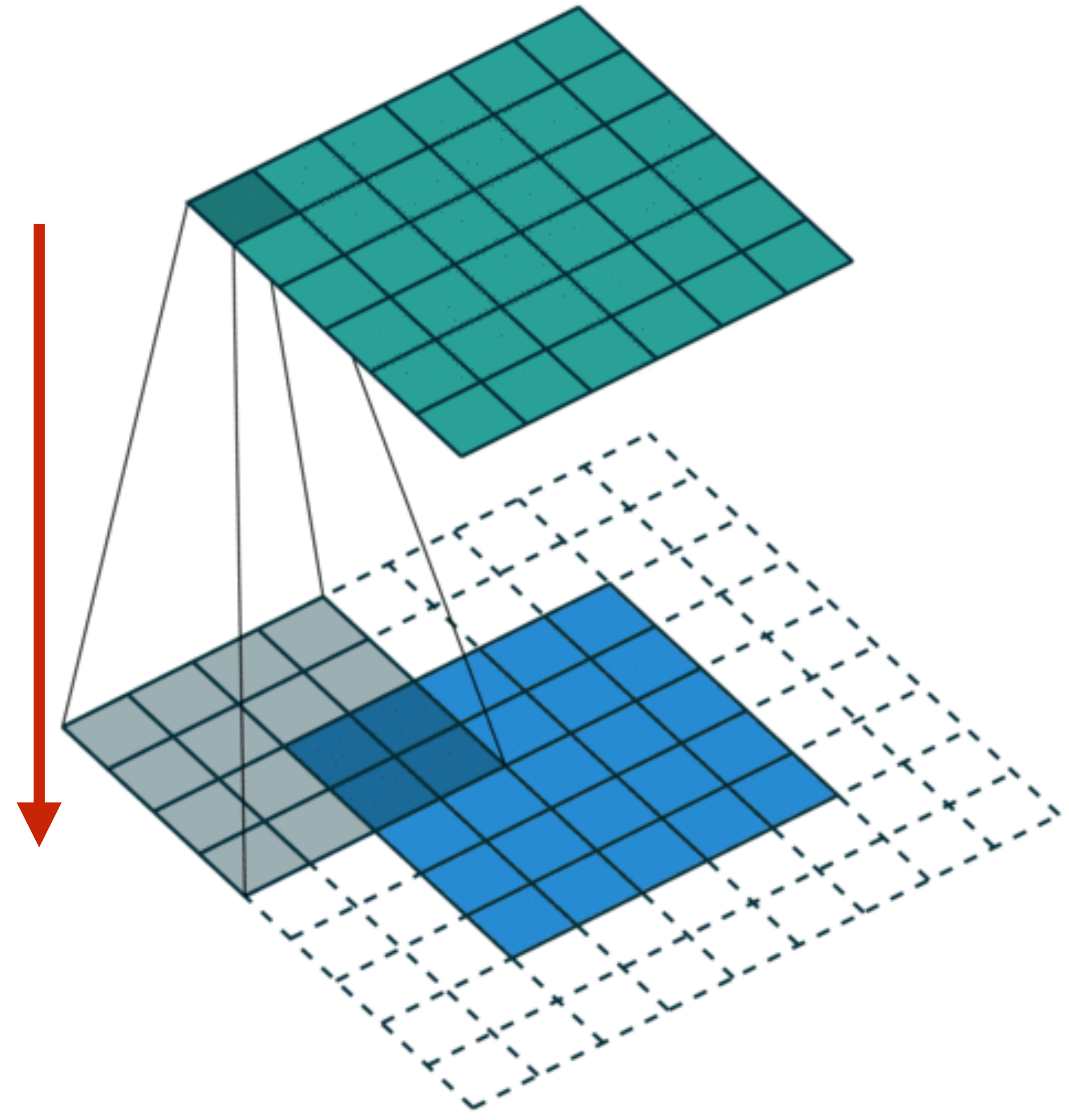
- Gradient of Loss wrt. parameters is…
  - a matrix of same shape as parameters
  - obtained by multiplying input and gradient of output:

```
for (int iy = 0; iy < InY; iy++) // loop over input Y
for (int ix = 0; ix < InX; ix++) // loop over input X
{
  for (int fy = 0; fy < KnY; fy++)
  for (int fx = 0; fx < KnX; fx++)
  {
    const int oy = ( iy + Py - fy ) / Sy;
    const int ox = ( ix + Px - fx ) / Sx;
    //padding: skip addition if outside input boundaries
    if (oy < 0 || oy >= OpX || ox < 0 || ox >= OpY) continue;

    for (int bc=0; bc<BS; bc++) // loop over batch
      for (int ic = 0; ic < InC; ic++) // loop over inp feature maps
        for (int fc = 0; fc < KnC; fc++) // loop over filters
          GK[fy][fx][ic][fc] += ERR[bc][oy][ox][fc] * INP[bc][iy][ix][ic];
  }
}
```

2D convolution backward loop (2):

- As for linear layers, we want to compute two things:
  - Gradient of loss wrt. input
  - Gradient of loss wrt. parameters

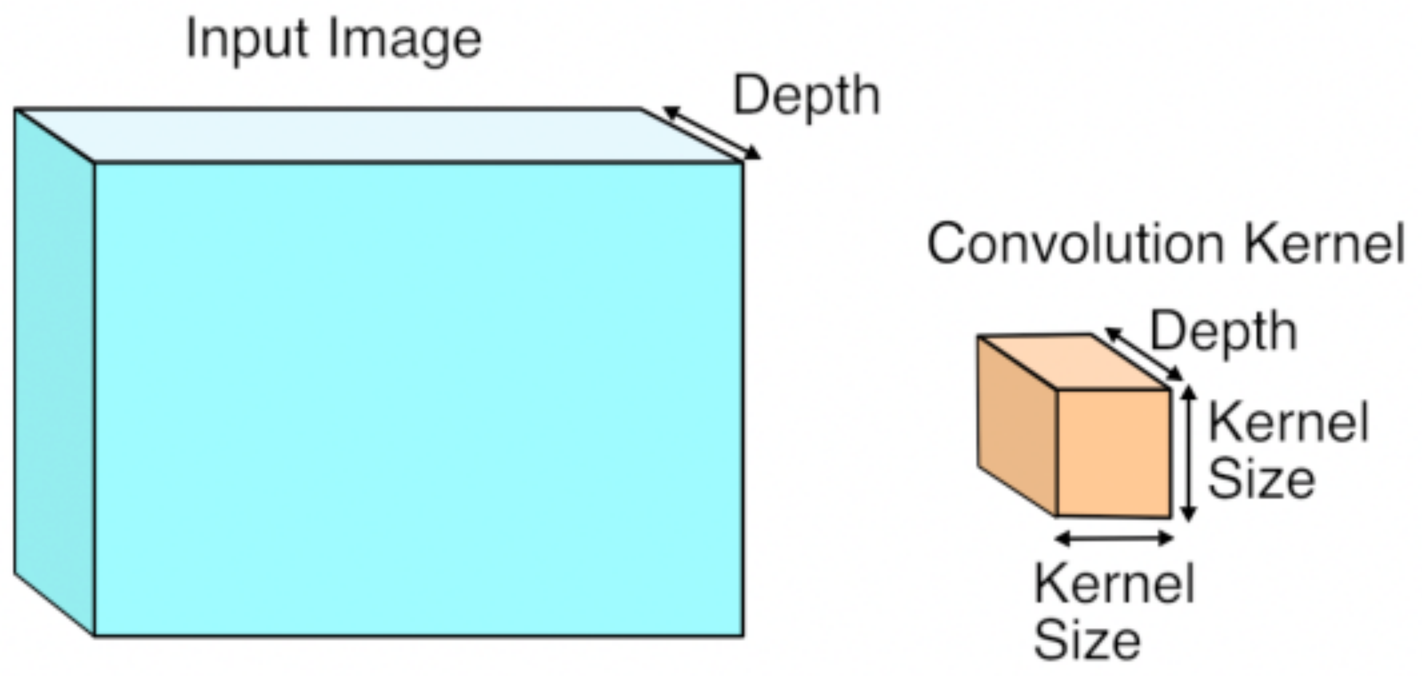


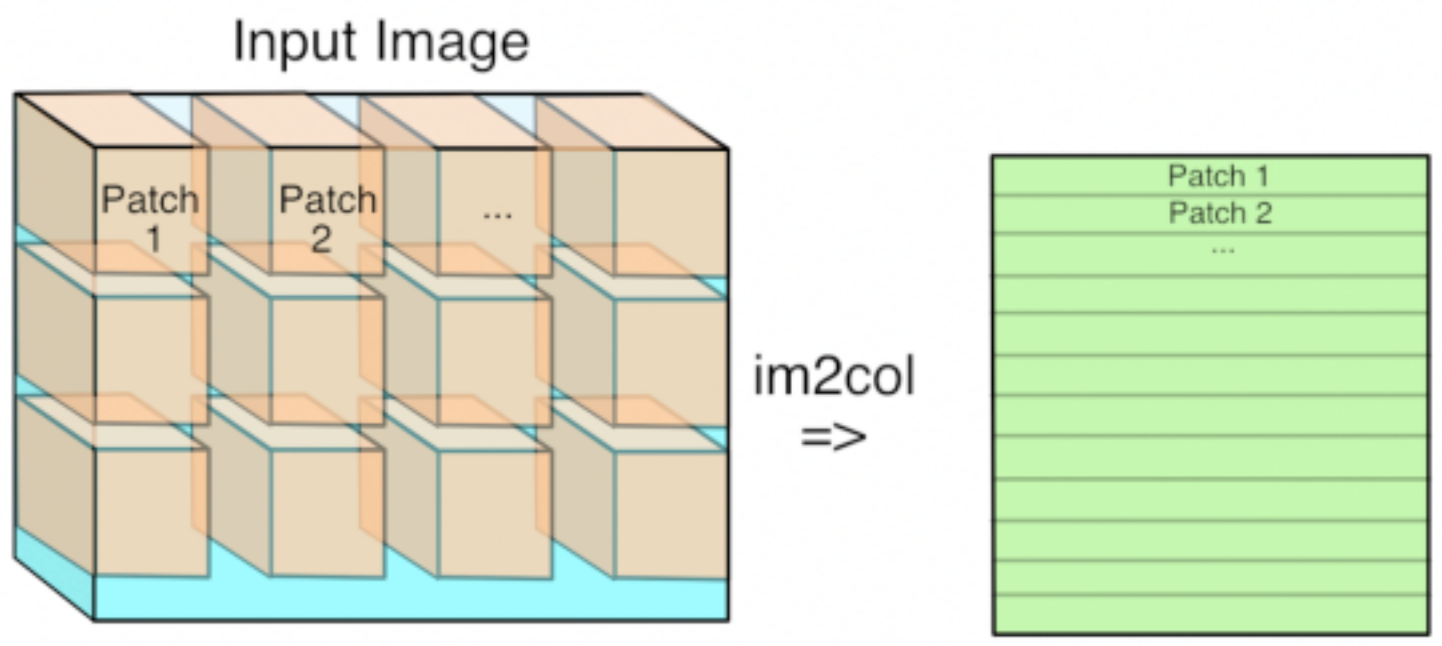
- Gradient of Loss wrt. input is…
  - a matrix of same shape as input
  - obtained by multiplying filter parameters and gradient of output:

```
for (int iy = 0; iy < InY; iy++) // loop over input Y
for (int ix = 0; ix < InX; ix++) // loop over input X
{
  for (int fy = 0; fy < KnY; fy++)
  for (int fx = 0; fx < KnX; fx++)
  {
    const int oy = ( iy + Py - fy ) / Sy;
    const int ox = ( ix + Px - fx ) / Sx;
    //padding: skip addition if outside input boundaries
    if (oy < 0 || oy >= OpX || ox < 0 || ox >= OpY) continue;

    for (int bc=0; bc<BS; bc++) // loop over batch
      for (int ic = 0; ic < InC; ic++) // loop over inp feature maps
        for (int fc = 0; fc < KnC; fc++) // loop over filters
          GINP[bc][iy][ix][ic] += ERR[bc][oy][ox][fc] * K[fy][fx][ic][fc];
  }
}
```

Exercise: how do we GEMM this?



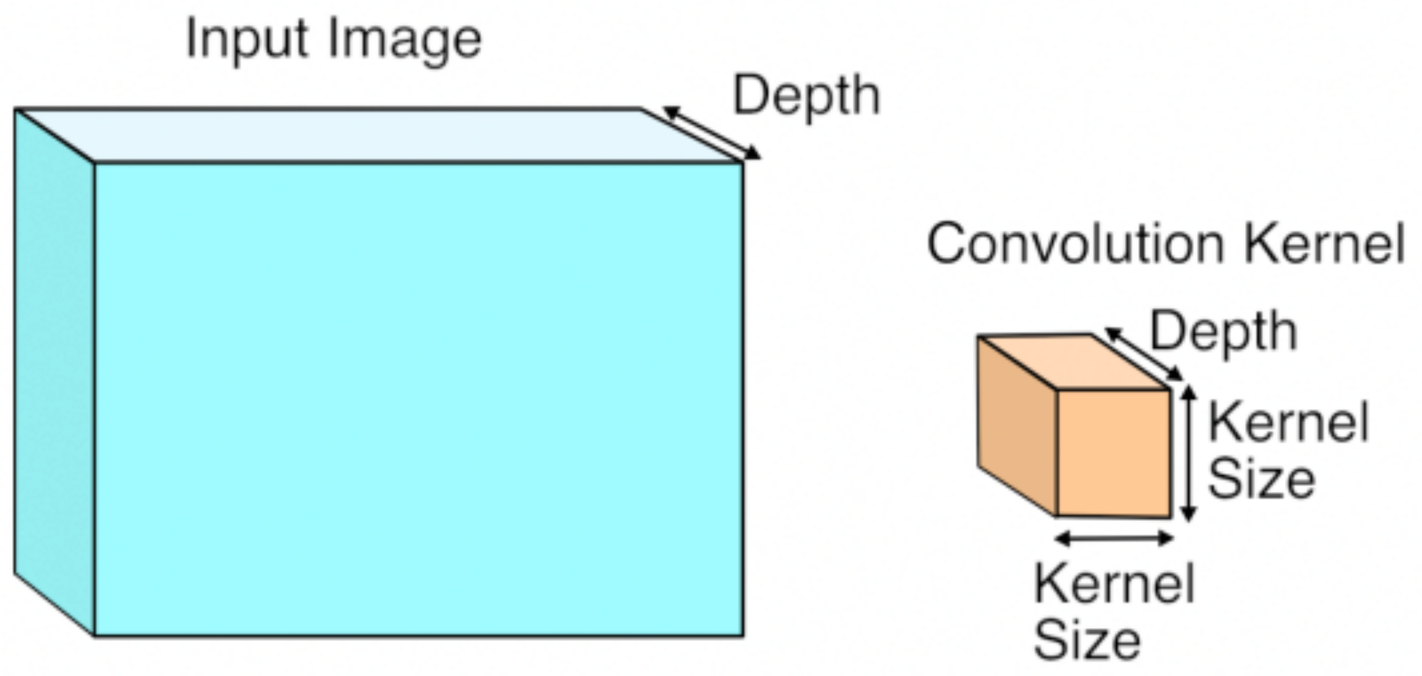


im2col

- ML libraries generally split convolution in two steps: "im2col" and GEMM
- Each convolution is performed by "dotting" a 3D patch of the image with the filter
- im2col copies these patches and organises them into a matrix of size

  (BS * OpY * OpX) x (KnY * KnX * InC)
- One row for each x and y pixel of the batch
- Column of size of the inner prod with one filter
- im2col generally produces an output which occupies more memory than the input img

Exercise: how do we GEMM this?



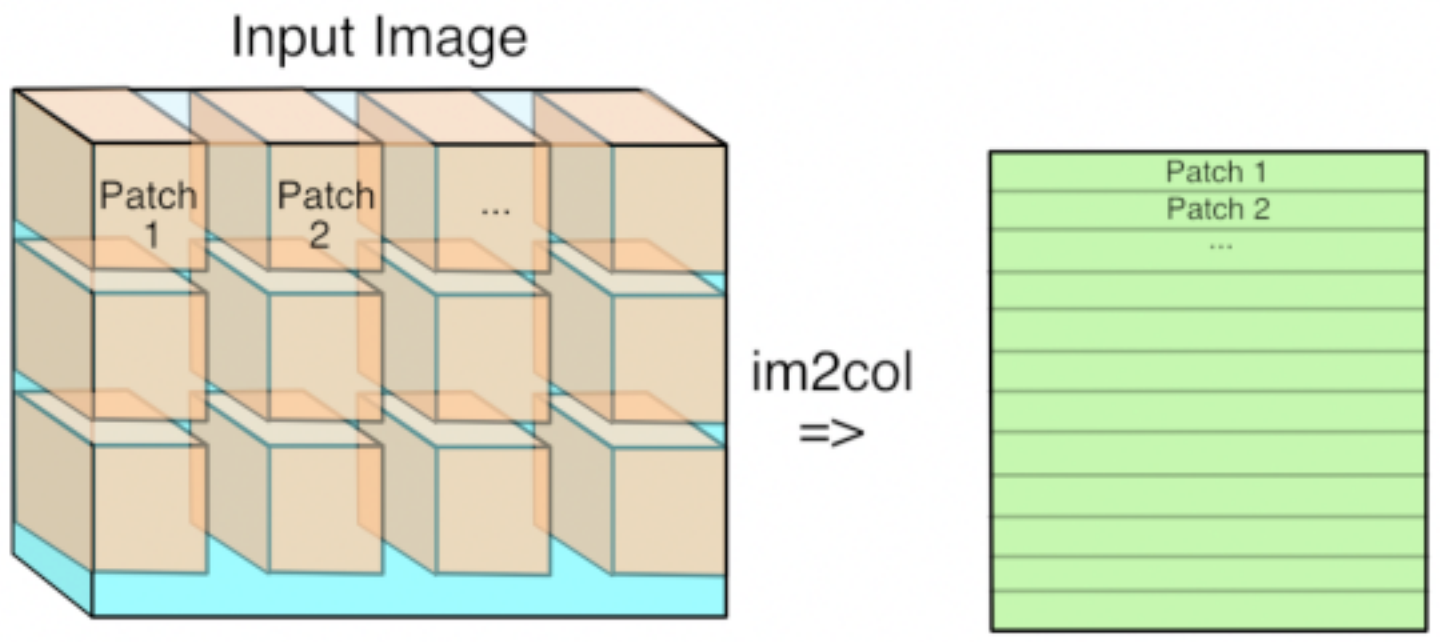
- Now that memory is aligned, compute output with GEMM:

(mini-batch of output images)

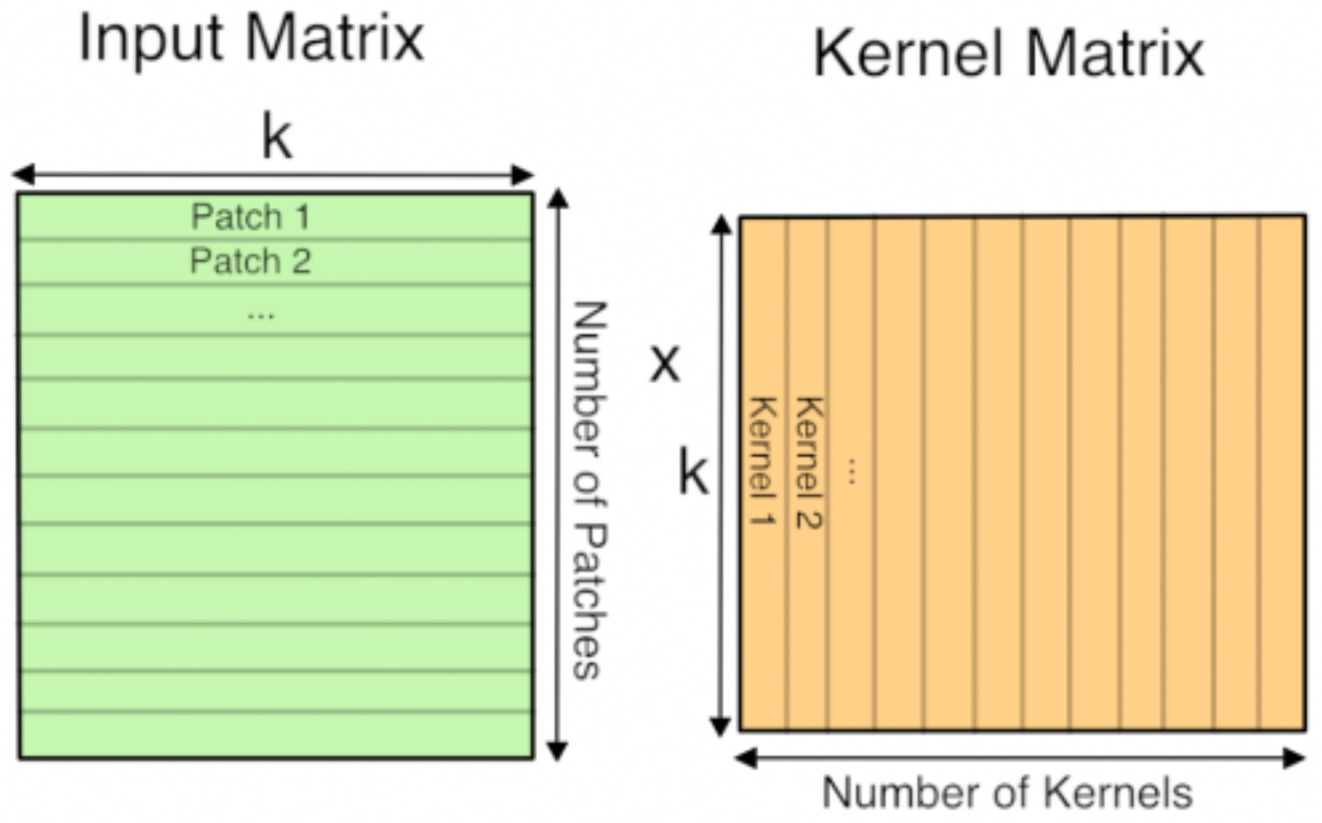
[(BS * OpY * OpX) (KnC)] =

[(BS * OpY * OpX) (KnY * KnX * InC)] * [(KnY * KnX * InC) (KnC)]

(output of im2col) (filter parameters)



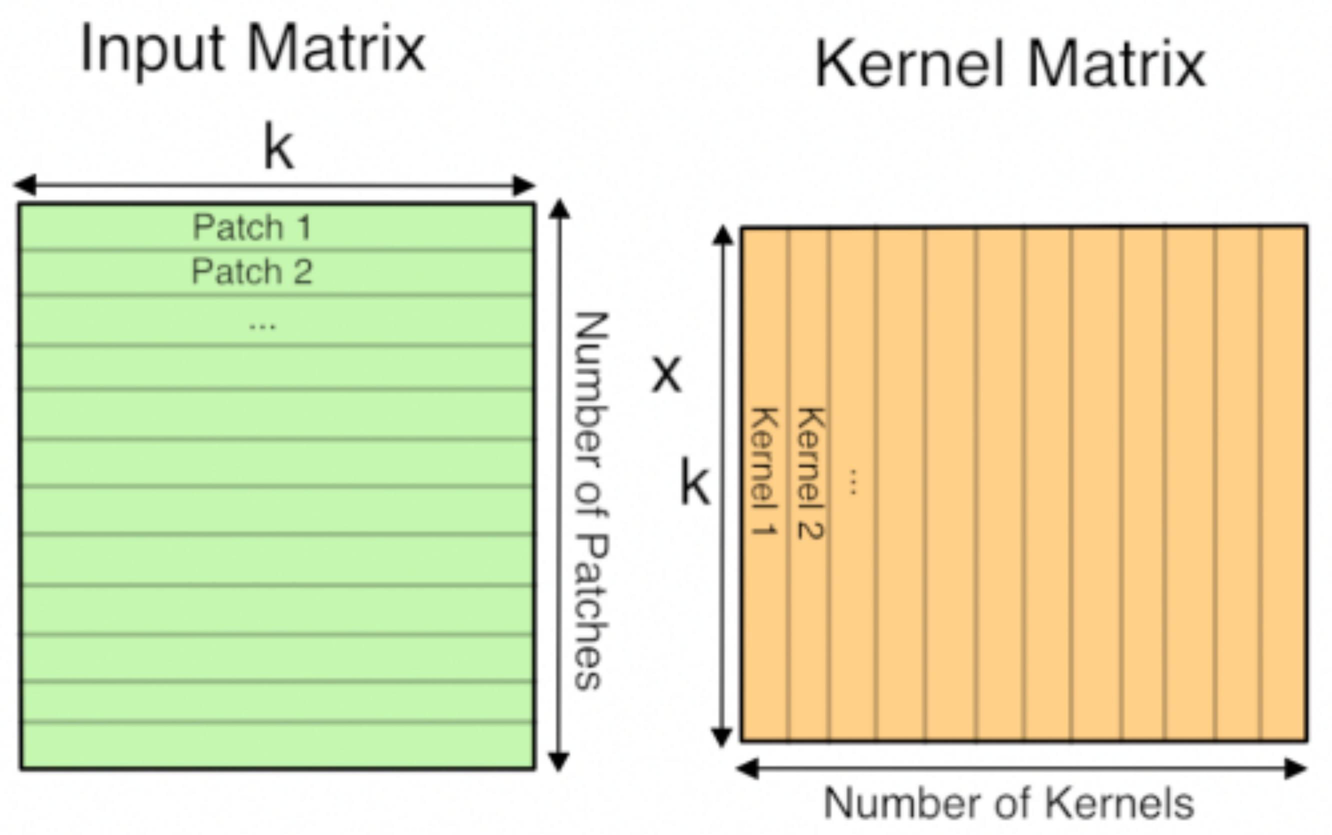
im2col

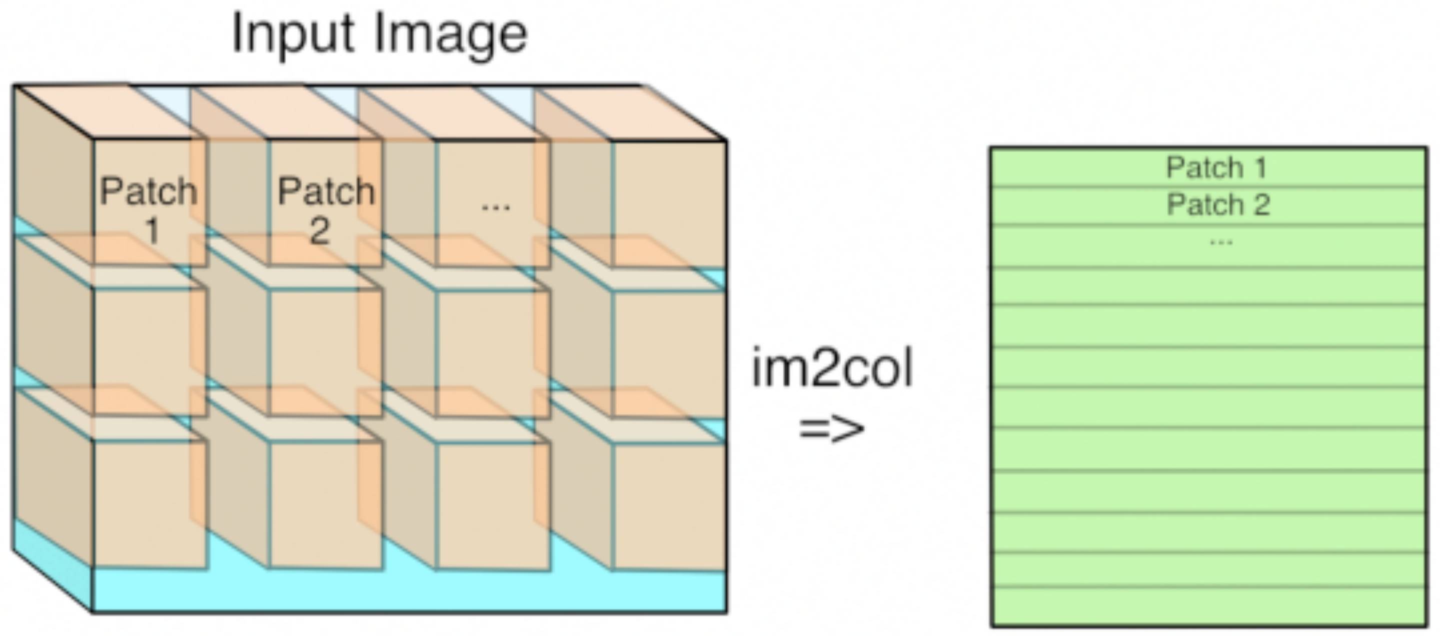


GEMM

- Likewise, we split conv layer into two layers:
  - im2col layer
  - conv2d (gemm) layer

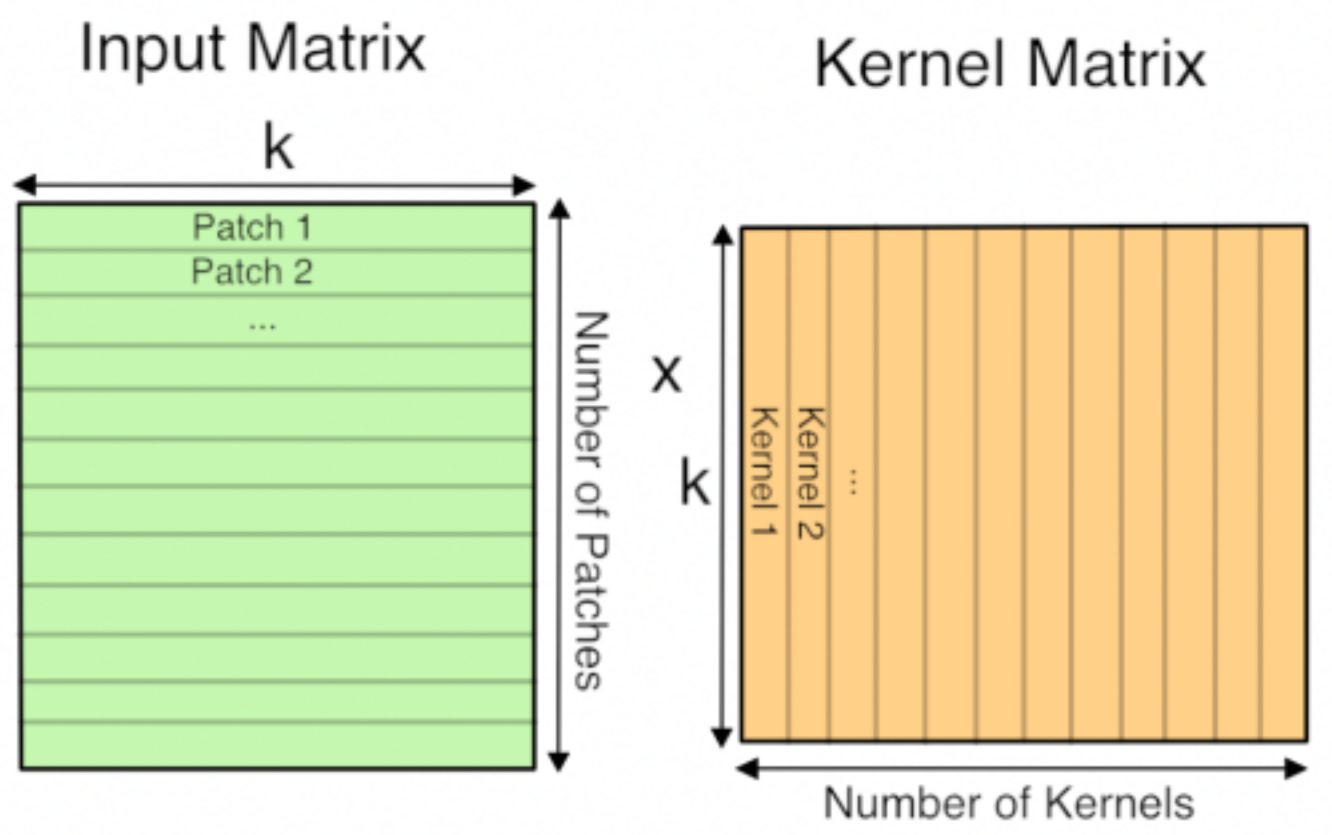
Exercise: how do we backprop this?

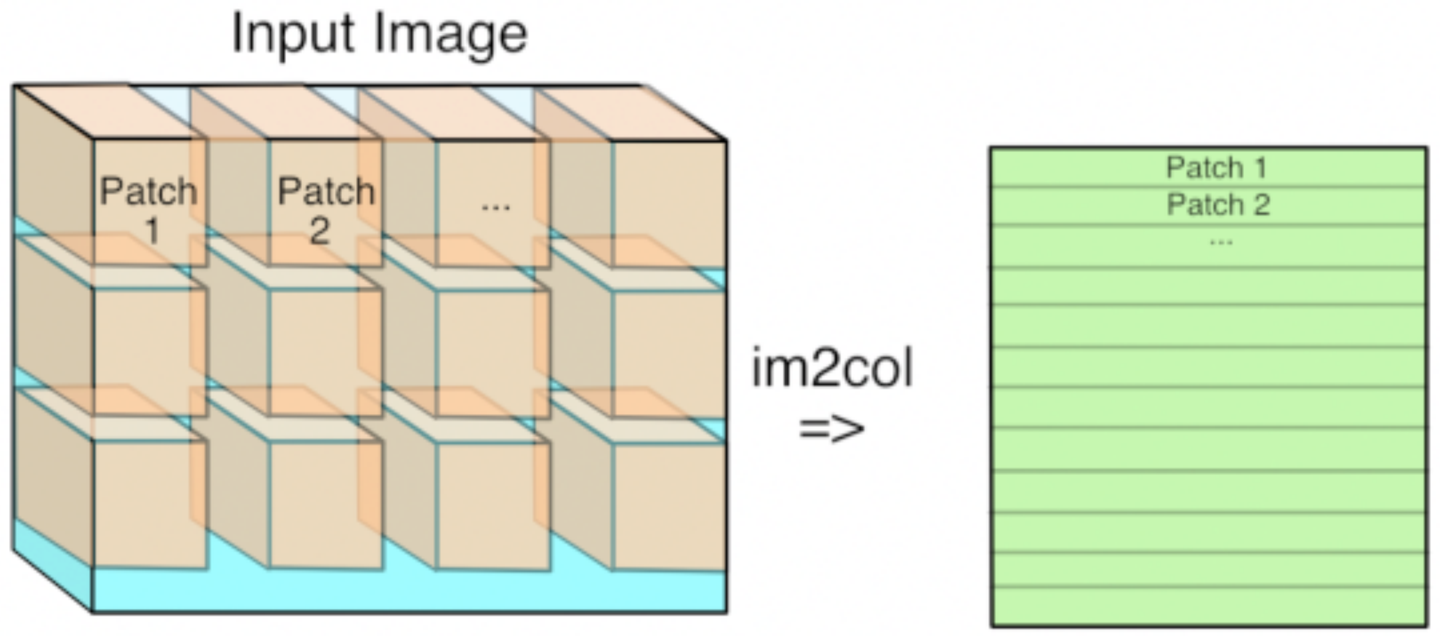




- Backprop will begin at output layer and proceed to input
- Therefore, for convolution, first we backprop the conv2d layer and then we backprop the im2col layer
- conv2d layer now behaves like linear layer, implement the operation like in exercise 6:
  - dL/dW, dL/dB, dL/dInp are performed by GEMM and loops
- backprop of im2col layer is col2im

Exercise: how do we backprop im2col?





- Backprop of conv2d computes D = dL / **dOutput** of Im2Col
- matrix D : [(BS * OpY * OpX) (KnY * KnX * InC)]
- As in the im2col step, multiple entries of this matrix are associated with each input pixel
- Backprop of im2col computes E = dL / **dInput** of Im2Col
- matrix E : [(BS) (InY * InX * InC)]
- Loop through E. For each pixel in (InY, InX) add up the errors contained in D for the corresponding (OpY, OpX, KnY, KnX)
- Mapping between (InY, InX) and the various (OpY, OpX, KnY, KnX) is the same as slide "*2D convolution backward loop*"

Exercise: a major help

- This is of course not the only way to iterate through arrays
- It sure is convenient
- Only works if array sizes are known at compile time

```
void Im2Mat(const int BS, const Real*const lin_inp, Real*const lin_out) const
{
```

Input is some linear memory allocation

```
  // Convert pointers to a reference to multi dim arrays for easy access:
  // 1) INP is a reference: i'm not creating new data
  // 2) The type of INP is an array of sizes [???][InY][InX][InC]
  // 3) The first dimension is the batchsize and is not known at compile time
  // 4) Because it's the slowest index the compiler does not complain
  // 5) The conversion should be read from right to left: (A) convert lin_inp
  // to pointer to a static multi-array of size [???][InY][InX][InC]
  // (B) Return the reference of the memory space pointed at by a.
  const Real (& INP )[][InY][InX][InC] =
          *  (Real(*)[][InY][InX][InC]) lin_inp;
  //      (B) (           A             )
```

Convert to C array

(works only if sizes are known at compile time, except for the slowest index which can be unknown)

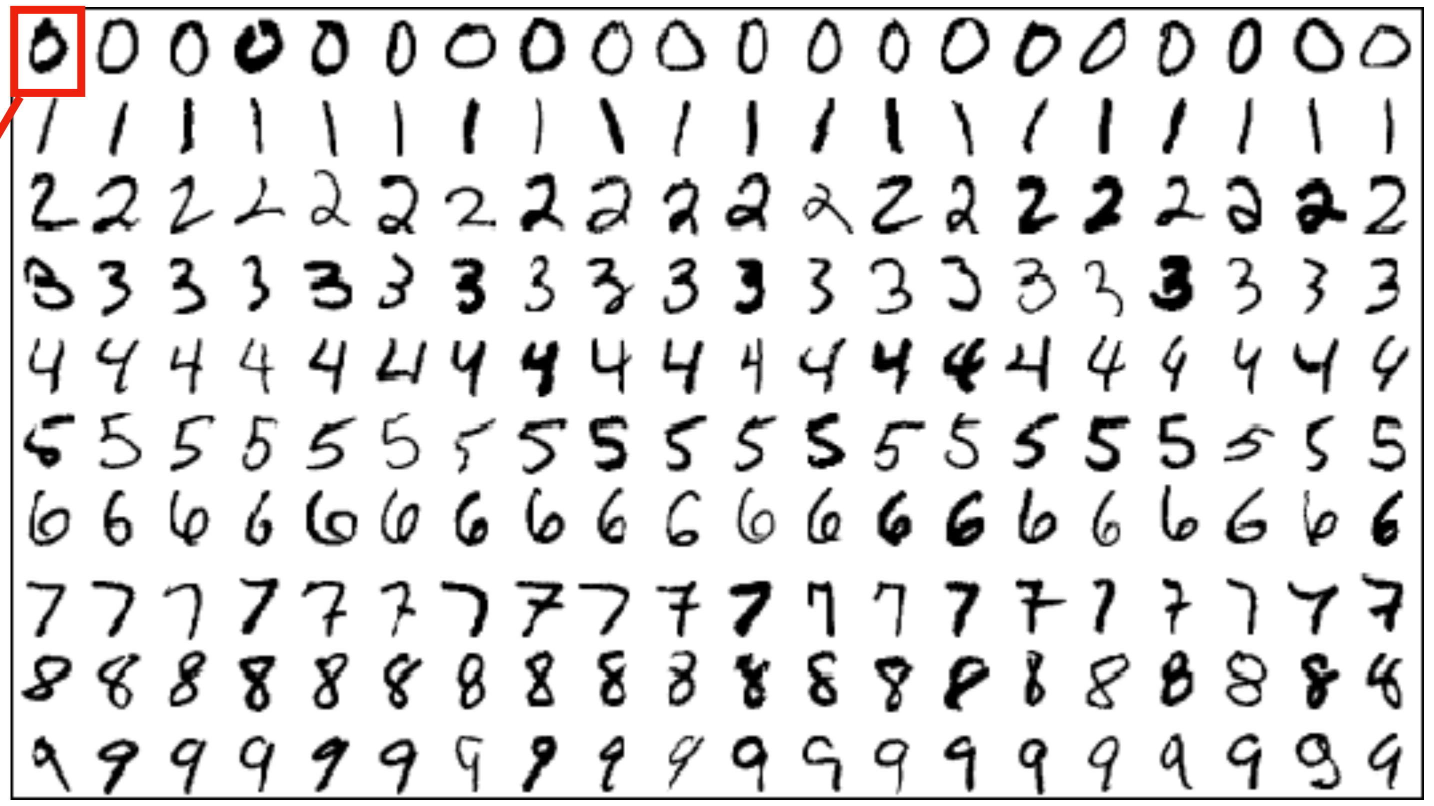
```
??? = INP[bc][iy][ix][ic];
```

Easy and clear access

Overall objective of the exercise

- Learn to classify MNIST digits

Learn to label each sample:



Output of the network is the probability of each image of being a certain digit

→ **CNN** →

	Probability that image is
{	
0.02	0
0.03	1
0.01	2
0.01	3
0.70	4
0.02	5
0.02	6
0.01	7
0.06	8
0.12	9
}	

Changes from Ex06

- For this exercise we use the Adam optimizer (Kingma & Ba, 2014)
- The non-linearity after each layer is a "Leaky" rectifier linear unit (LReLu):

```
f(x) = x > 0 ? x : 0.1 x
```

- Output layer is a SoftMax: $f(x_i) = \frac{\exp x_i}{\sum_{j=1}^{10} \exp x_j}$
- Sum of outputs is one: they behave like probabilities for each digit
- Loss function is the Cross Entropy: $$H(\tilde{f}, f) = -\sum_{i=1}^{10} \tilde{f}(x_i)\ \log f(x_i)$$
    - Measure of dissimilarity between probability distributions
    - Minimized if the distributions are identical
    - Target distribution $\tilde{f}(x)$ are the labels. What is probability that an image is each digit? Well, this is a 4, so:

$$\tilde{f}(x) = \{0, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0\}$$