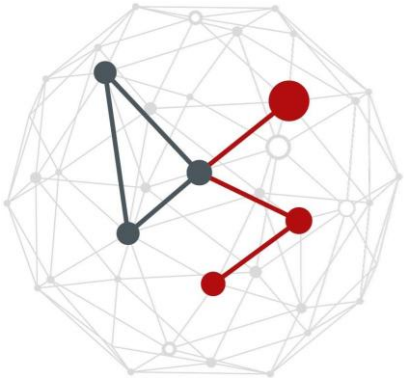
MIME
Master's degree ICT Internet Multimedia Engineering

LAB 2: CNN for supervised and self-supervised learning

Eleonora Cicciarella, Cesare Bidini
eleonora.cicciarella@phd.unipd.it
cesare.bidini@phd.unipd.it
University of Padova, IT



DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO MATEMATICA

UNIVERSITÀ DEGLI STUDI DI PADOVA

You will learn to

- Part I
  - use the Sequential class to implement a deep CNN
  - implement your own training pipeline
- Part II
  - use the tf.data.Dataset module to build an advanced input pipeline
  - use the Model class to implement a CNN-based autoencoder
  - use the training methods by TensorFlow
  - change the optimizer's and loss's parameters in the built-in training method compile()

Convolutional Neural Networks Recap

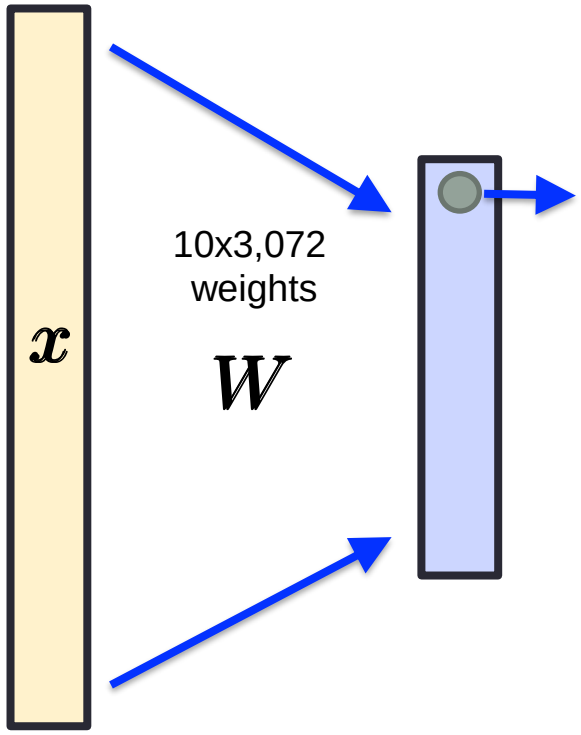
Convolutional neural networks in brief

- CNNs are a specialized kind of ANN to
  - process data that has a known *grid-like topology*
  - 1D time series (grid over time)
  - 2D images (grid over space)

- CNN
  - Employ a mathematical operation called convolution
  - **Convolution**
    - is a specialized kind of linear operation
    - differs slightly from standard convolution
    - is used in place of matrix multiplication in at least one layer
  - Have been hugely successful in many applications

Fully connected (dense) layer

- Example 32x32x3 image (3 color ch.) → flatten to 3,072x1



each entry j corresponds to one number obtained taking the dot product between:

- Row j of matrix **W**
- The input vector **x**

$$Wx$$

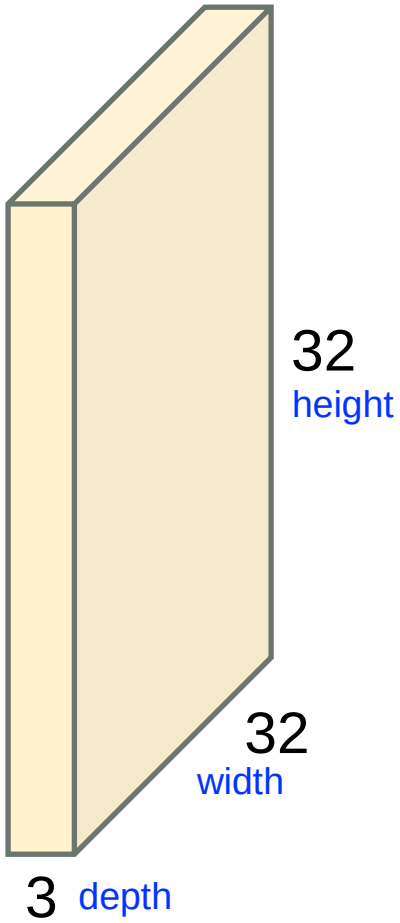
- And then applying a non-linear activation function

input vector
3,072x1

first layer's neurons
10x1

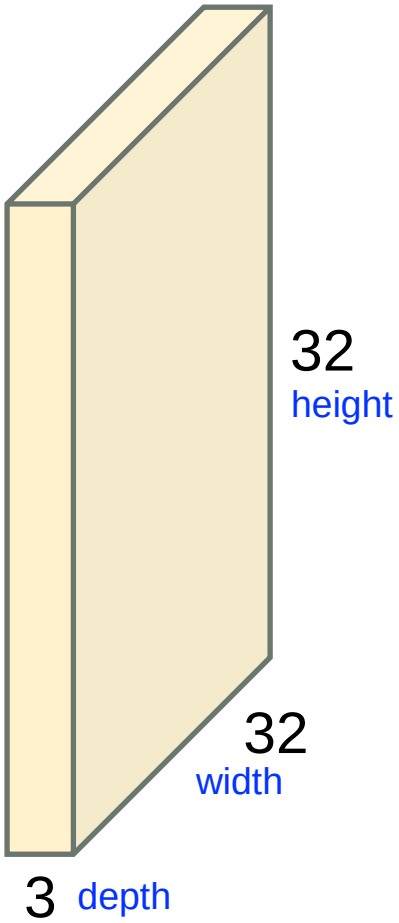
Convolutional layer

- Example 32x32x3 image → preserve spatial structure

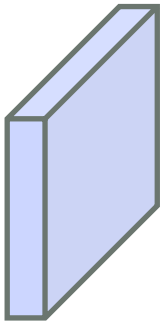


Convolutional layer

32x32x3 image



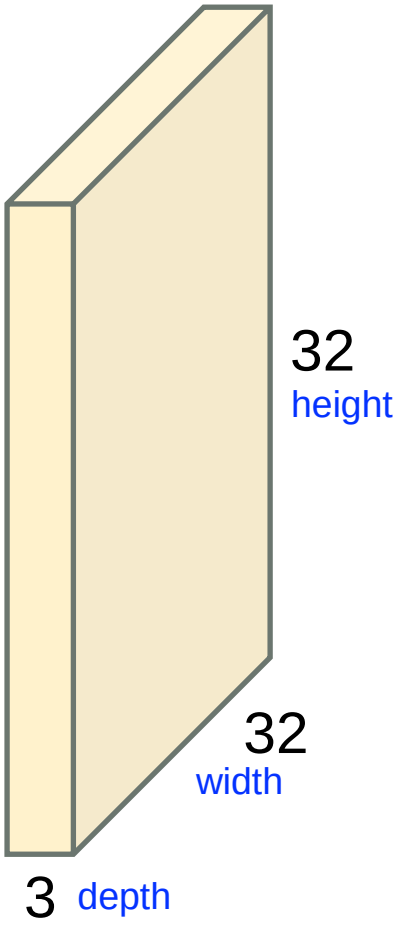
5x5x3 filter (or kernel)



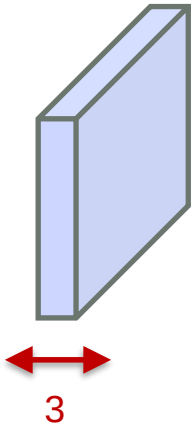
convolve the filter with the input image, i.e., *"slide over the image spatially convolving the kernel with each image patch"*

Convolutional layer

32x32x3 image



5x5x3 filter (or kernel)



Filters always extend the full depth of the input volume

2D convolution example

- Example 3x3 kernel
  - Called “Sobel filter”
  - Used to detect edges

-1	-2	-1
0	0	0
1	2	1

$K(m,n)$

- Input 3x3 image

1	2	3
4	5	6
7	8	9

$I(i,j)$

Output:

$$S(m,n) = (I * K)(m,n) = \sum_{i=-\infty}^{+\infty} \sum_{j=-\infty}^{+\infty} I(i,j)K(m-i,n-j)$$

Cross-correlation

$$(I \otimes K)(m,n) = \sum_{i=-\infty}^{+\infty} \sum_{j=-\infty}^{+\infty} I(i+m, j+n) K(i,j)$$

- It is often preferred for CNN implementations
- Reduces the variability of the indices
  - Leads to faster implementations

If we define:

$$K'(m,n) = \mathrm{rot}_{180^\circ}(K(m,n))$$

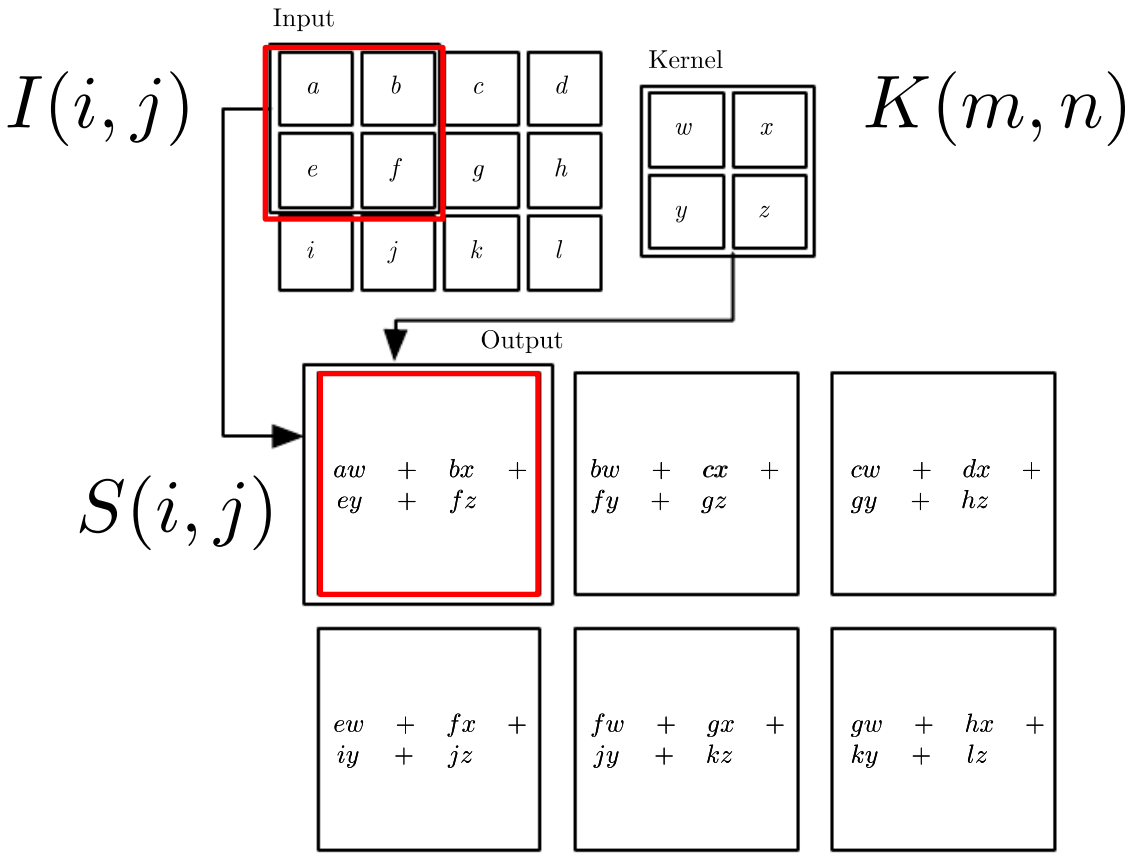
We have that:

$$(I * K)(m,n) = (I \otimes K')(m,n)$$

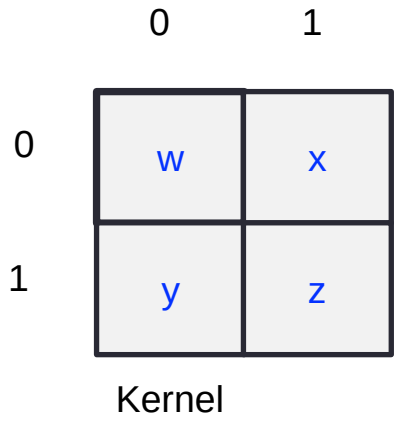
Convolution: equivalent to cross-correlation with a flipped Kernel

Cross-correlation

$$S(i,j) = (I \otimes K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i+m, j+n) K(m,n)$$



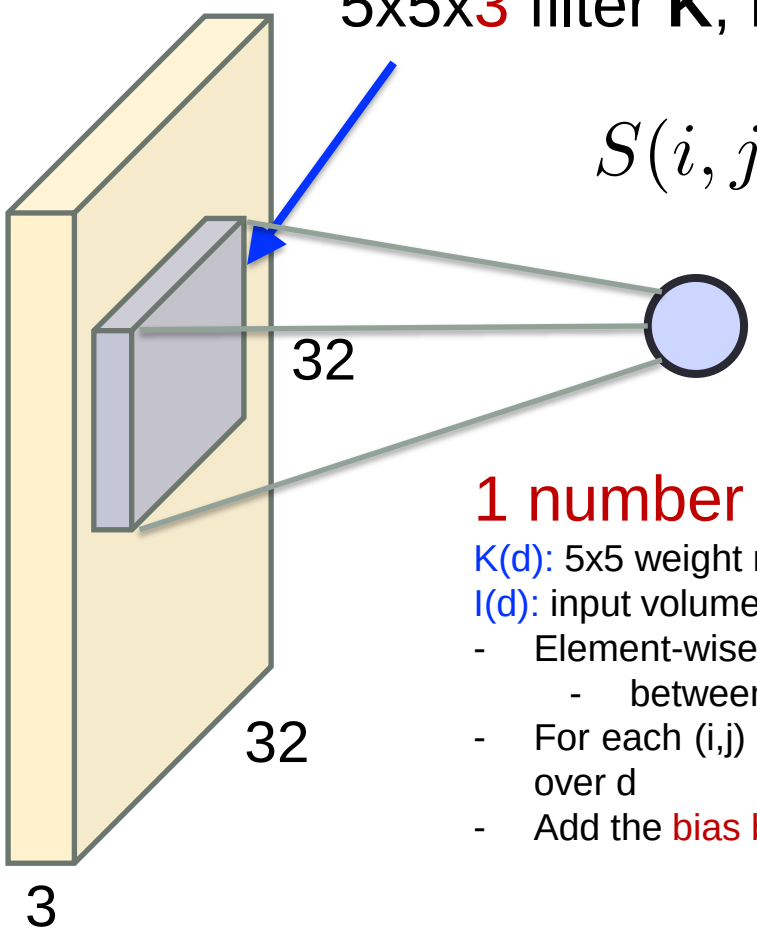
Kernel indexing



Convolution - input volume depth d=3

32x32x3 image **I**

5x5x3 filter **K**, filter depth d=3



$$S(i,j) = \sum_d (I(d) \otimes K(d))_{i,j} + b$$

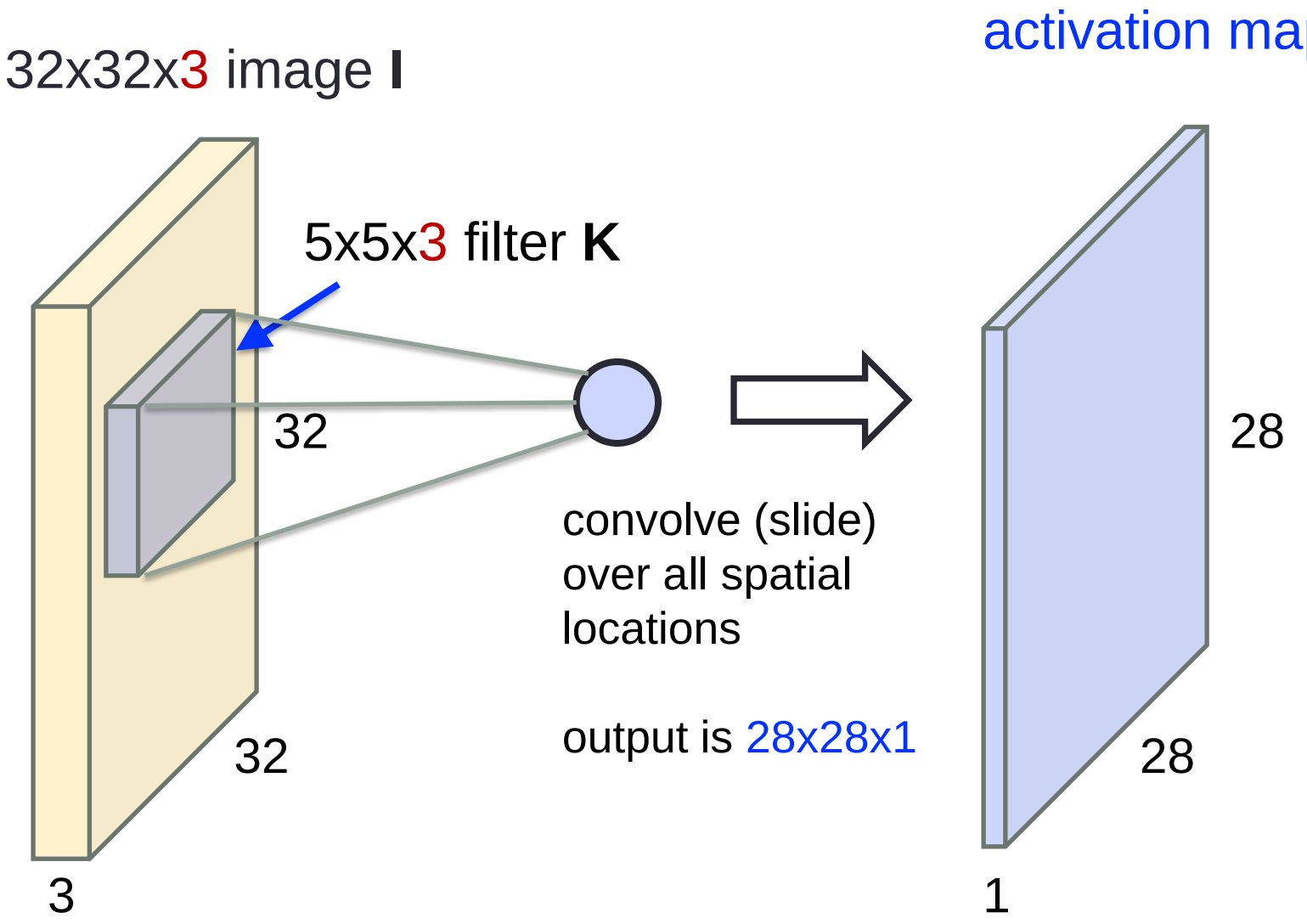
1 number - also called linear activation

K(d): 5x5 weight matrix along direction d

I(d): input volume along direction d

- Element-wise product (cross-correlation)
  - between image I(d) and K(d)
- For each (i,j) Results into a single number for each dimension d, sum over d
- Add the bias b

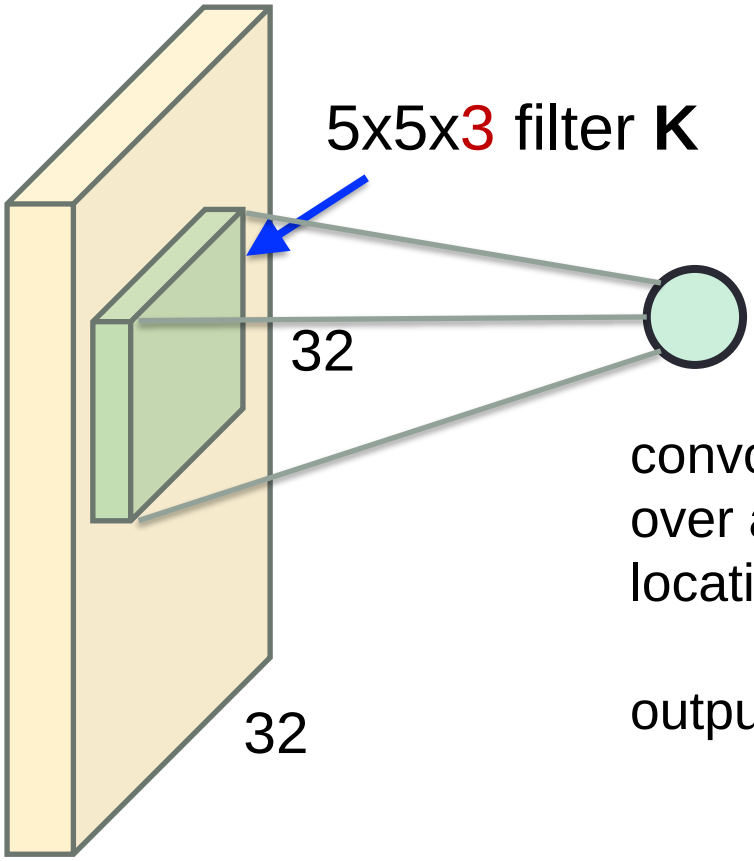
Convolution



Convolution

32x32x3 image **I**

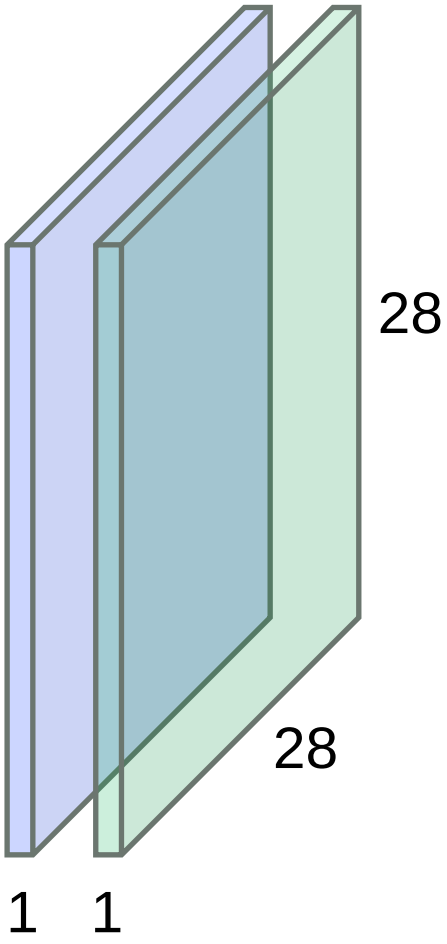
5x5x3 filter **K**



convolve (slide) over all spatial locations

output is 28x28x1

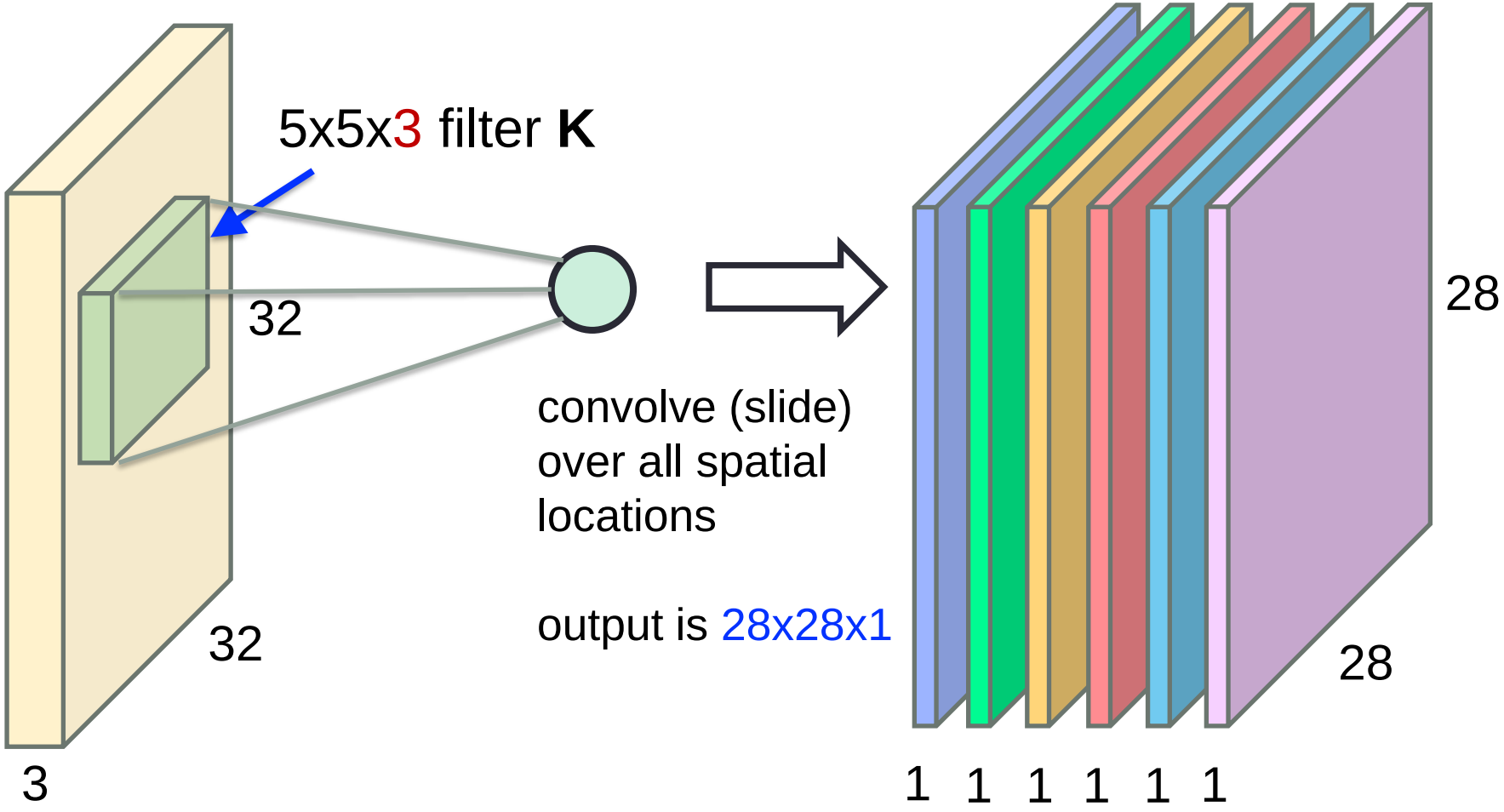
consider a second **green** filter
2 activation maps



Convolution

If we have 6, 5x5x3 filters, we get
6 activation maps of size 28x28x1

32x32x3 image **I**

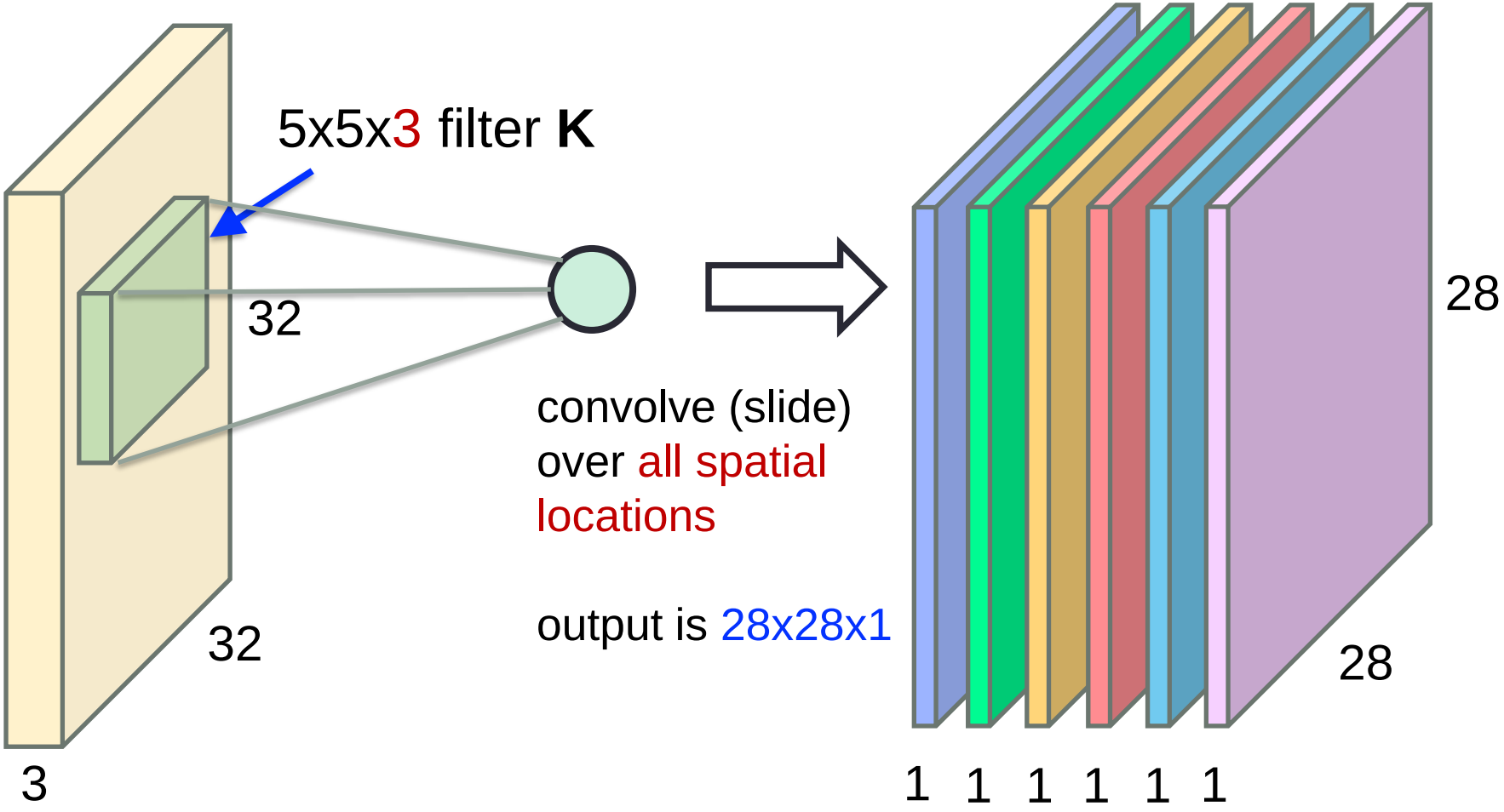


Convolution

depth of output volume is 6 (6 filters)

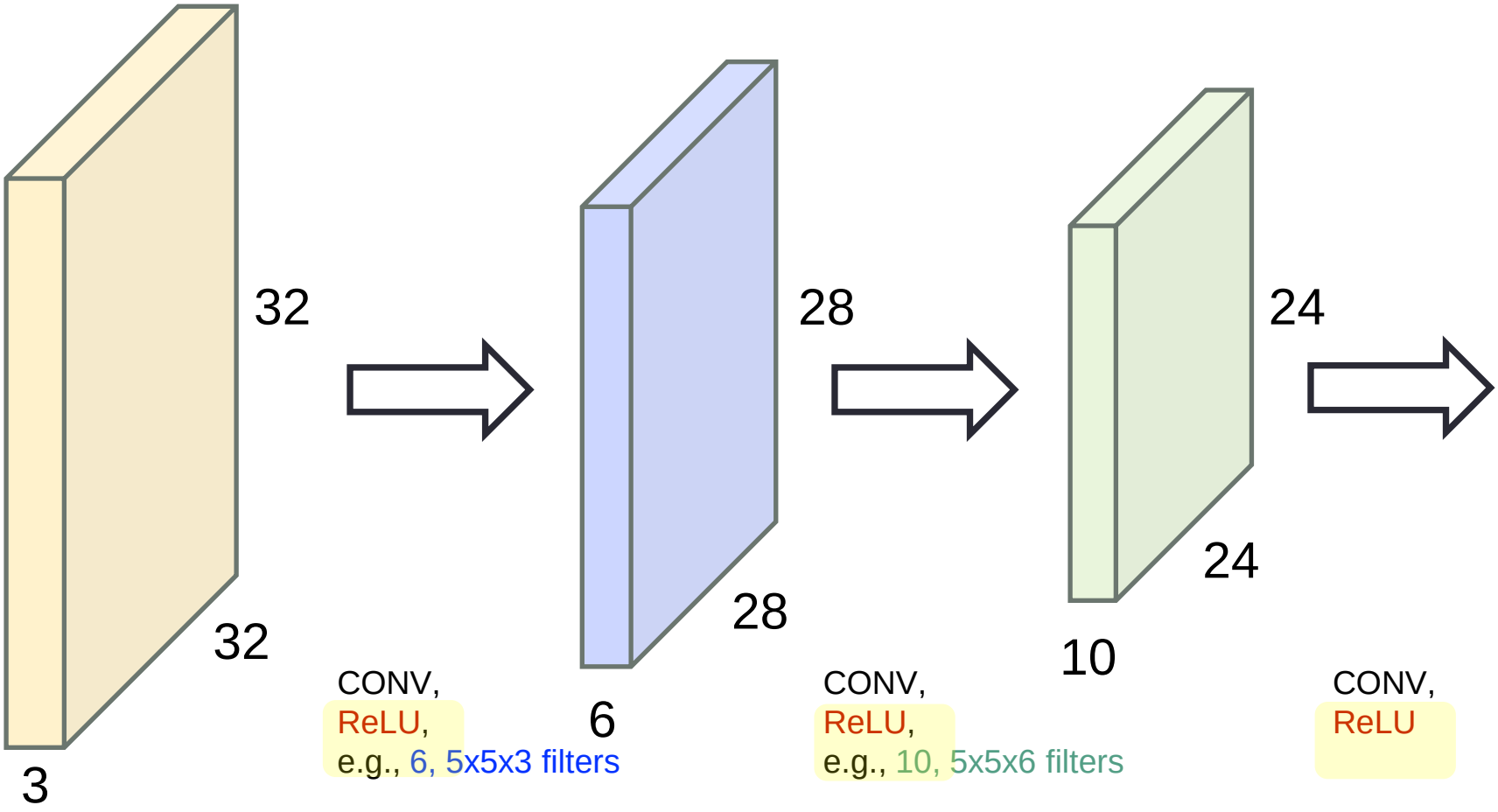
we obtain 6 2D feature maps

32x32x3 image **I**

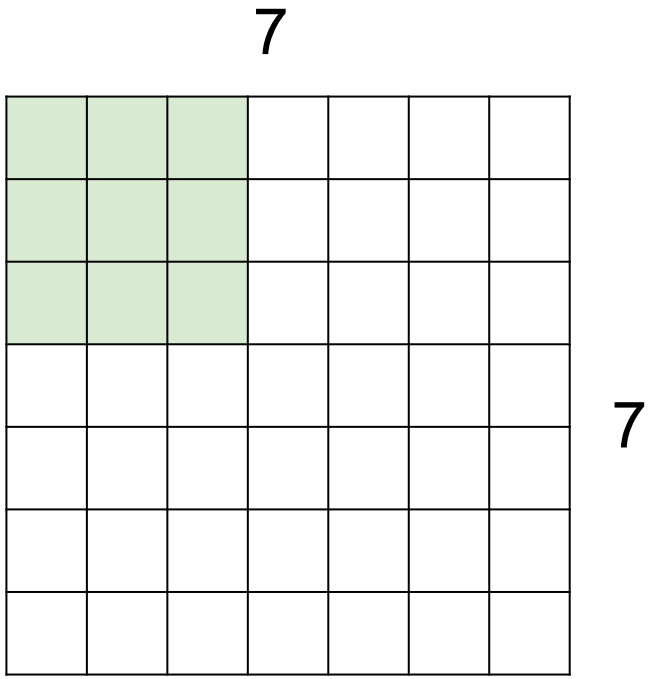


Convolutional network

It is a sequence of convolutions *interspersed* with activation functions



A closer look at spatial dimensions



STEP 1

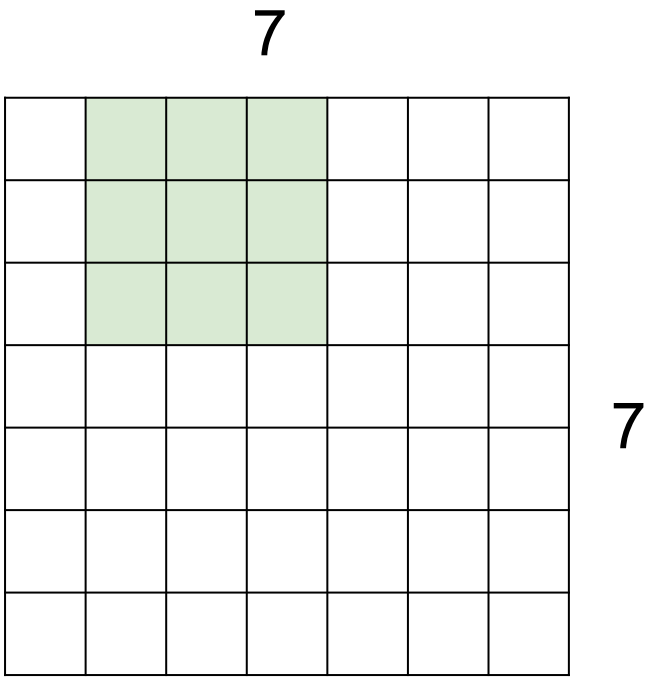
7x7(x1) input

assume a 3x3(x1) green filter

sliding step = 1

sliding step is referred to as stride

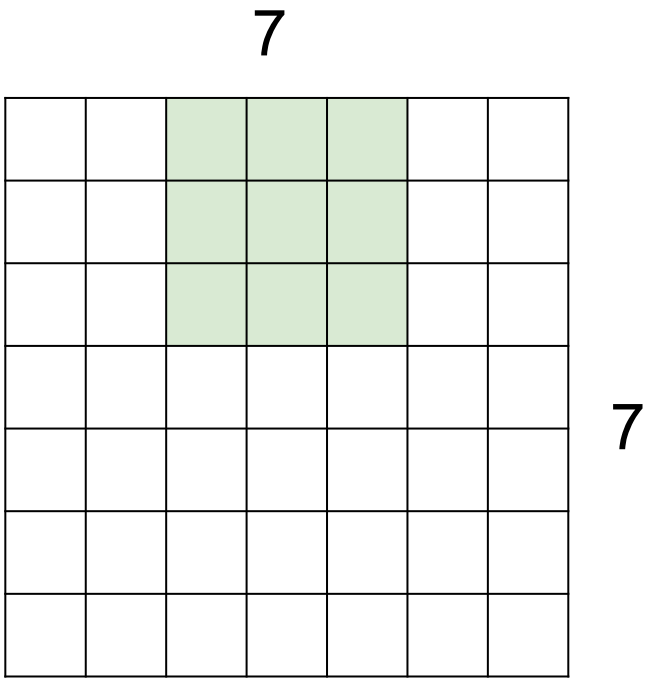
A closer look at spatial dimensions



STEP 2

move filter one step to the right

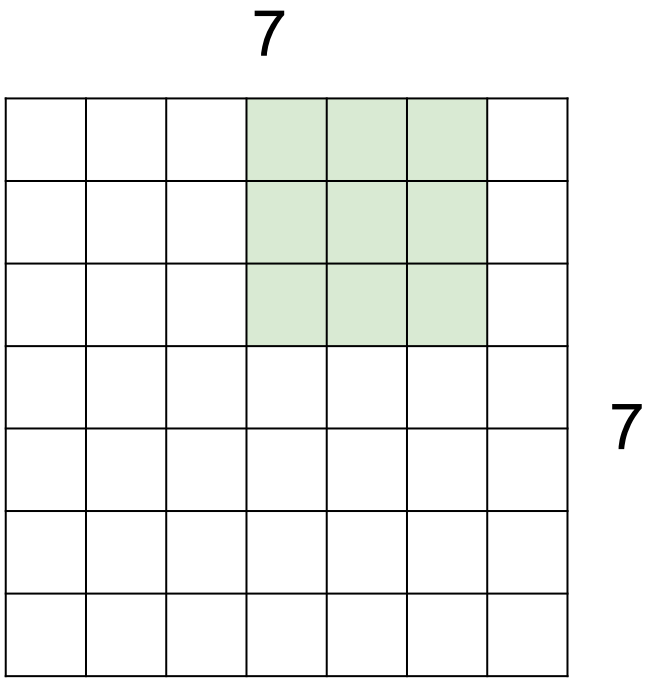
A closer look at spatial dimensions



STEP 3

move one more step to the right

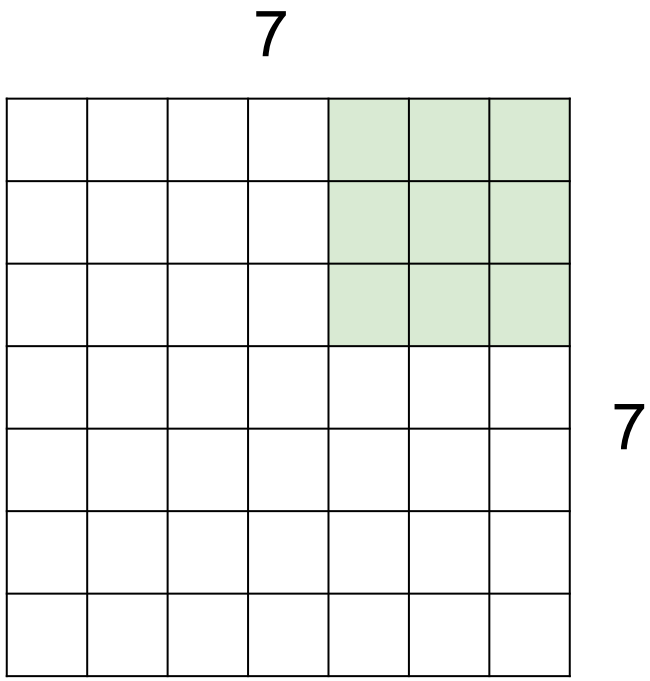
A closer look at spatial dimensions



STEP 4

move one more step to the right

A closer look at spatial dimensions

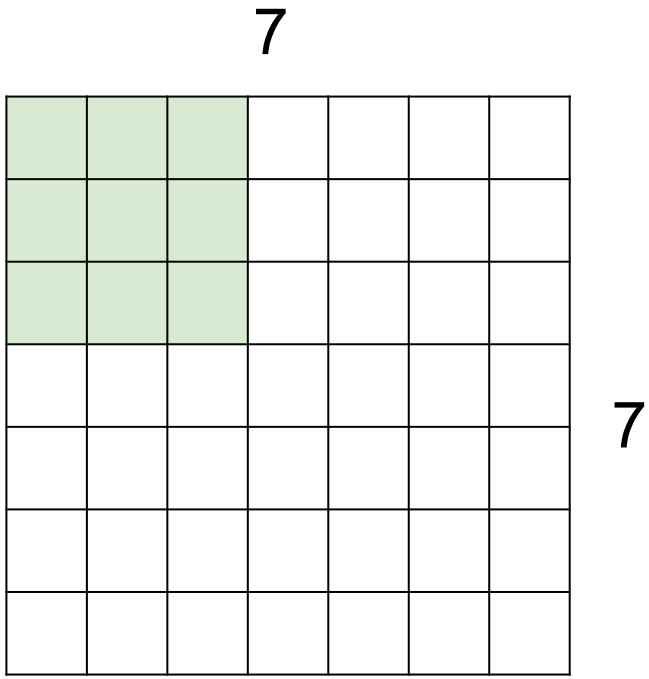


STEP 5

move one more step to the right

5x5 output

A closer look at spatial dimensions

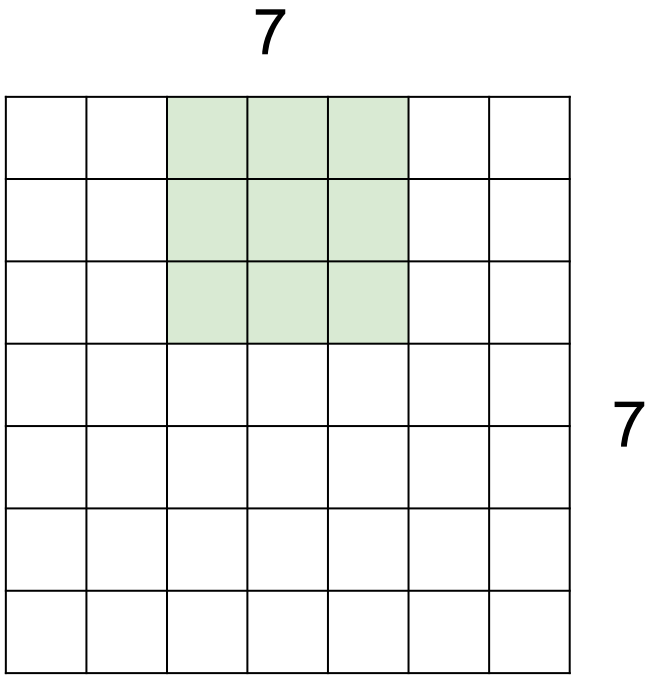


STEP 1

7x7 spatial input
assume a 3x3 green filter

apply a stride of 2

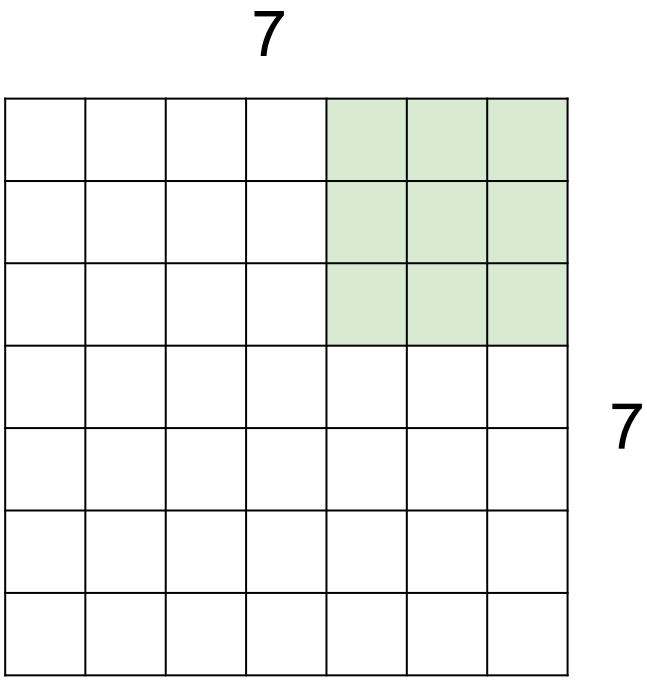
A closer look at spatial dimensions



STEP 2

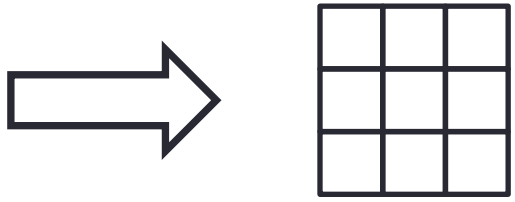
move 2 steps to the right

A closer look at spatial dimensions



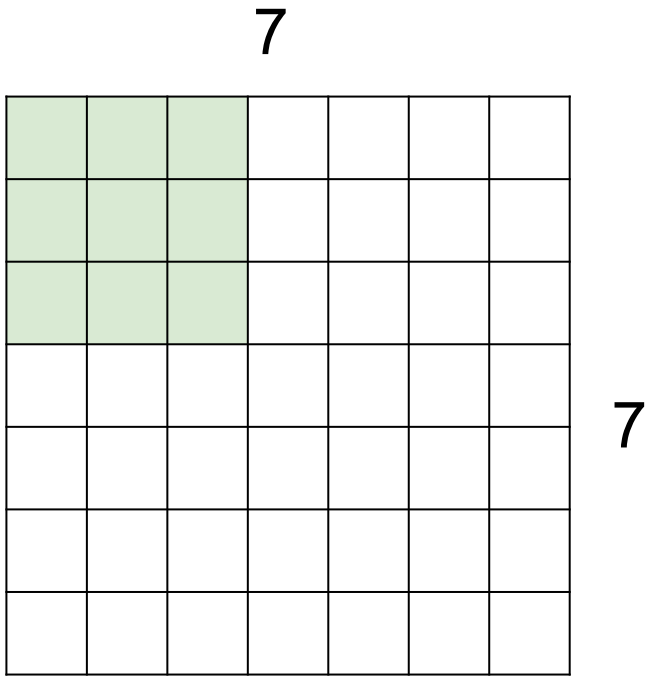
STEP 3

move 2 more steps to the right



3x3 output

Spatial dimensions: stride = 3

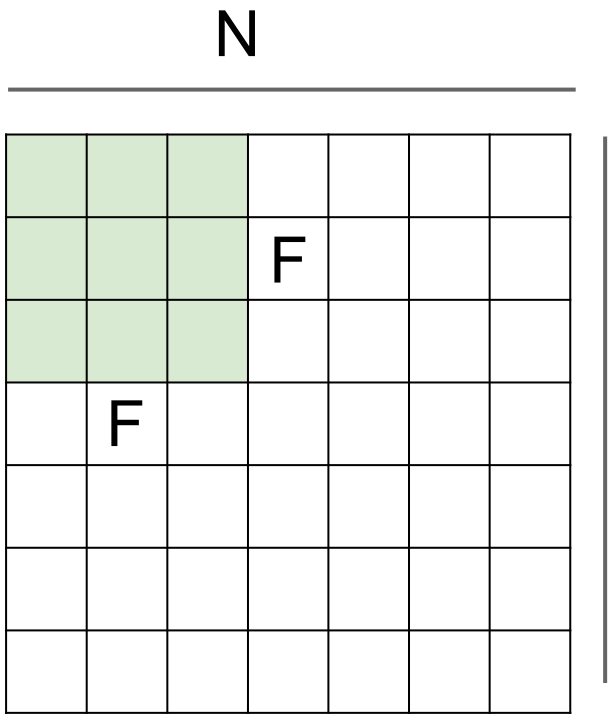


We now try with a stride of 3

Does not fit!

Cannot apply a 3x3 filter with a 7x7 input and stride = 3

Output size *vs* filter size & stride

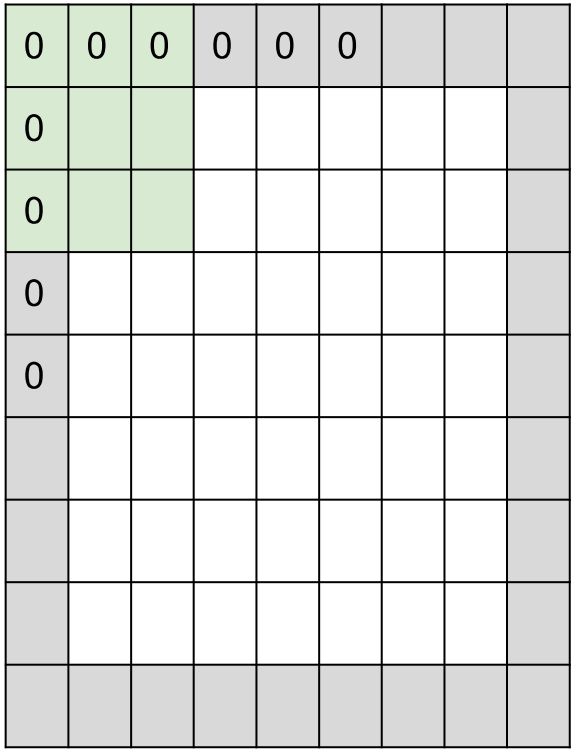


Output volume (OxO):
O = (N-F) / stride + 1

e.g., N=7, F=3
stride=1, O = (7-3)/1 + 1 = 5
stride=2, O = (7-3)/2 + 1 = 3
stride=3, O = (7-3)/3 + 1 = 2.33 :-(

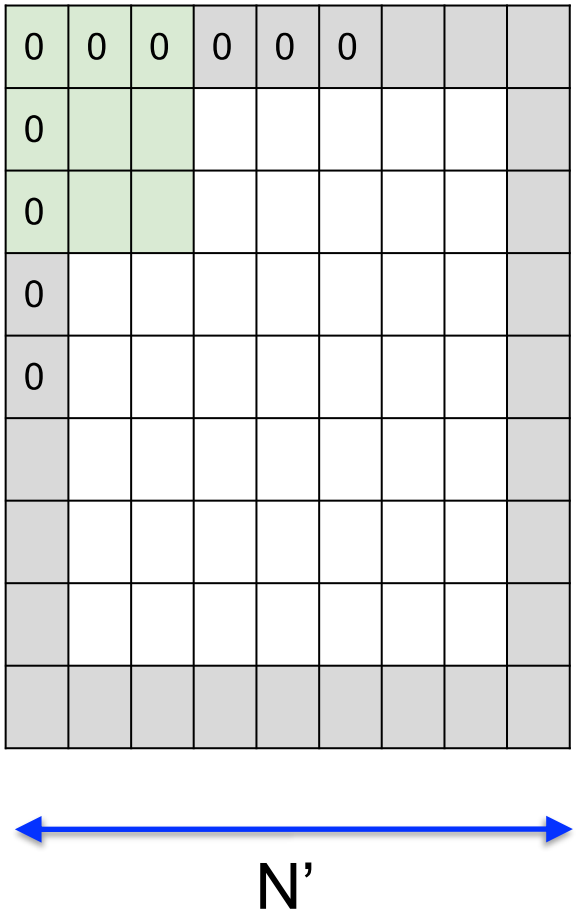
NOTE: output volume O is generally smaller than input volume and it also decreases with an increasing stride

Solution



It is common to zero-pad the border

Solution



If stride = 1, FxF filters and zero padding width (F-1)/2

We have that:

$N' = N + 2((F - 1)/2) = N + F - 1$

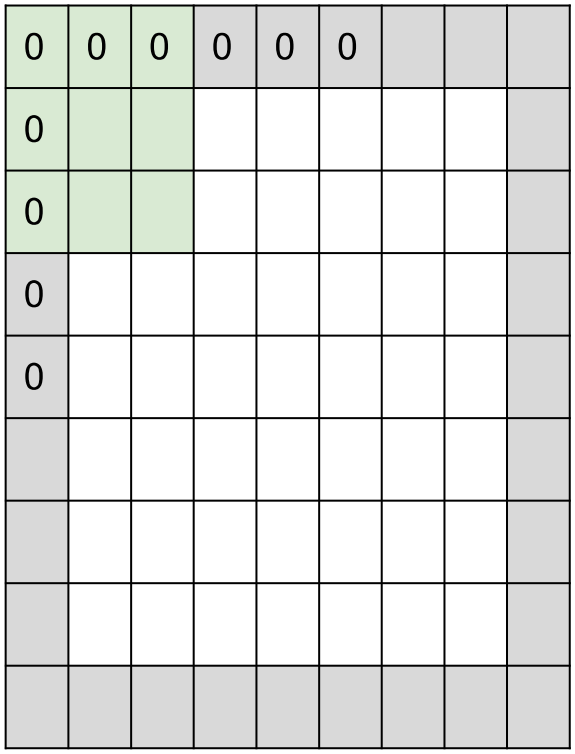
$O = (N' - F) / \text{stride} + 1$

$= (N + F - 1 - F) / 1 + 1 = N$

This means that the output size O is equal to the input size N

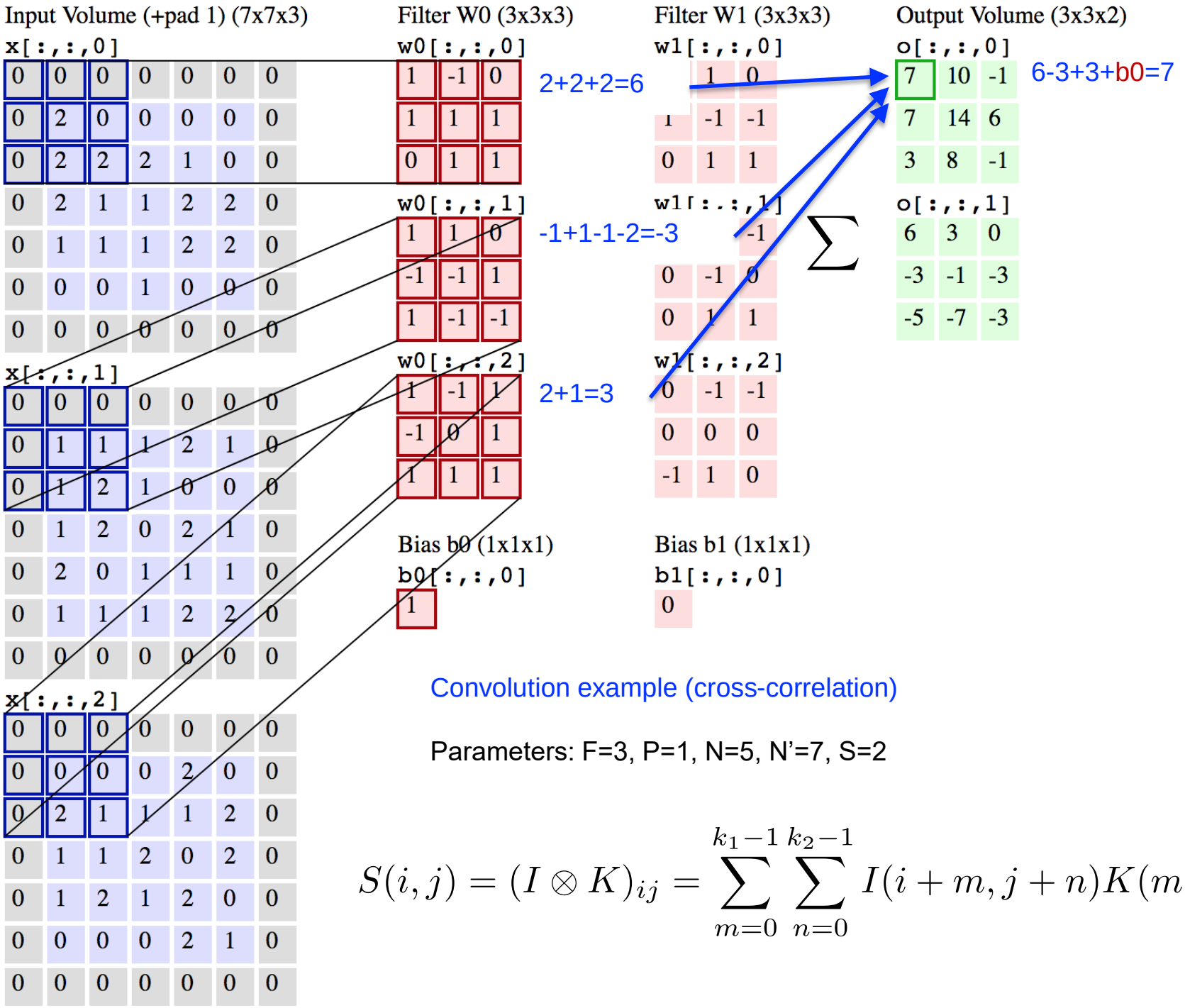
From input to output volume

input volume



N = input width/height
F = filter size
P = padding size
S = stride
O = output width/height

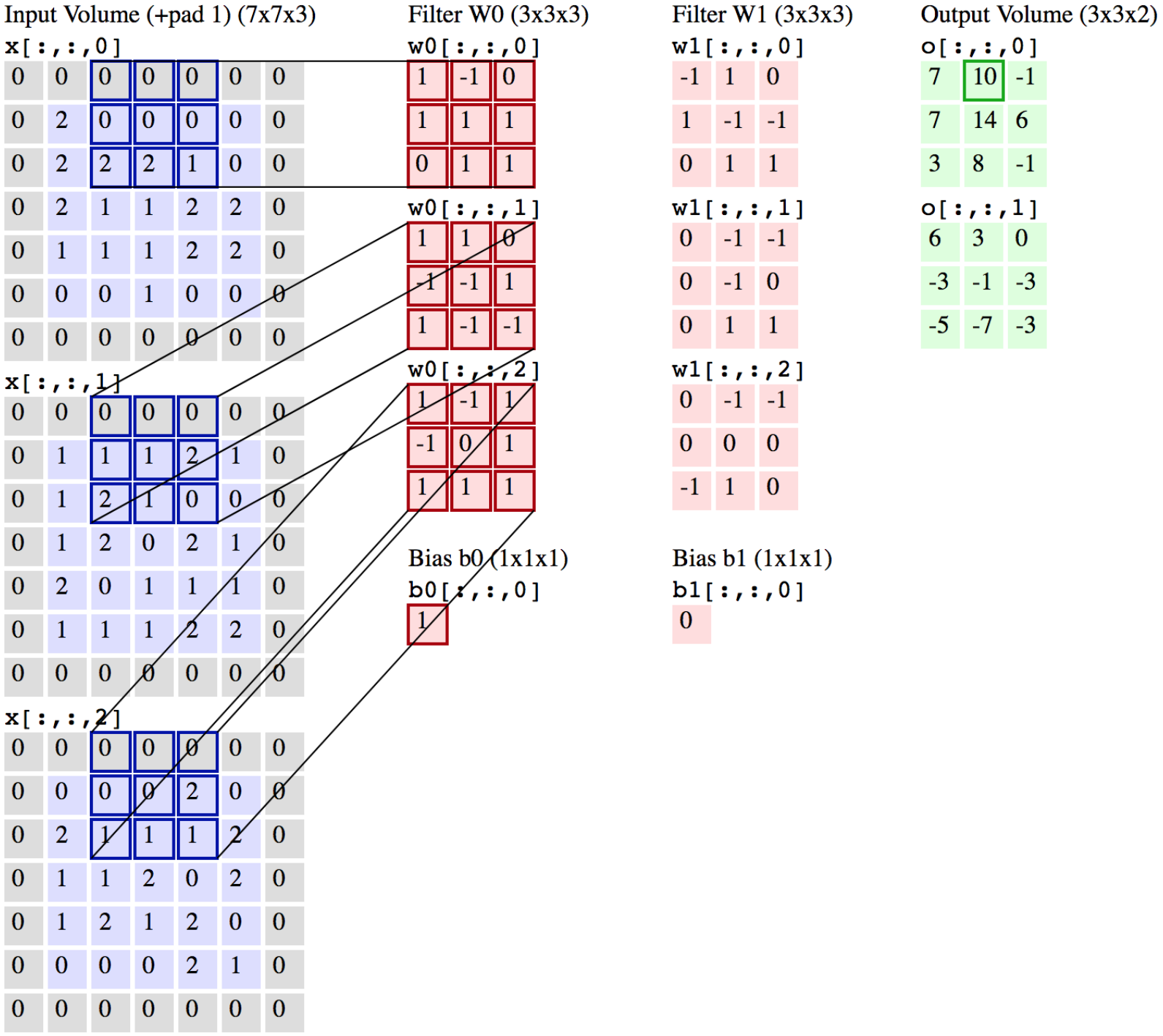
$$O = \frac{N - F + 2P}{S} + 1$$

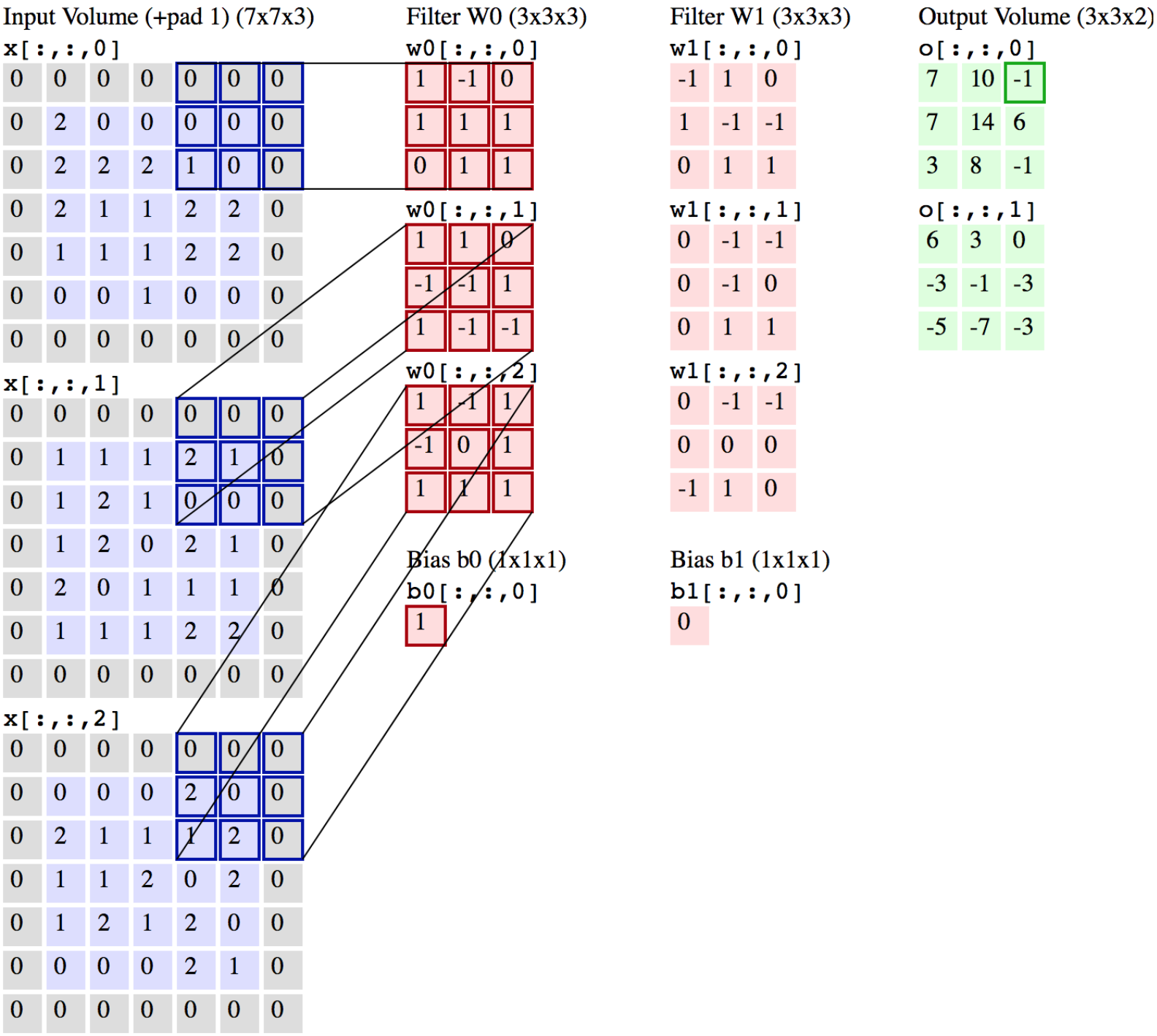


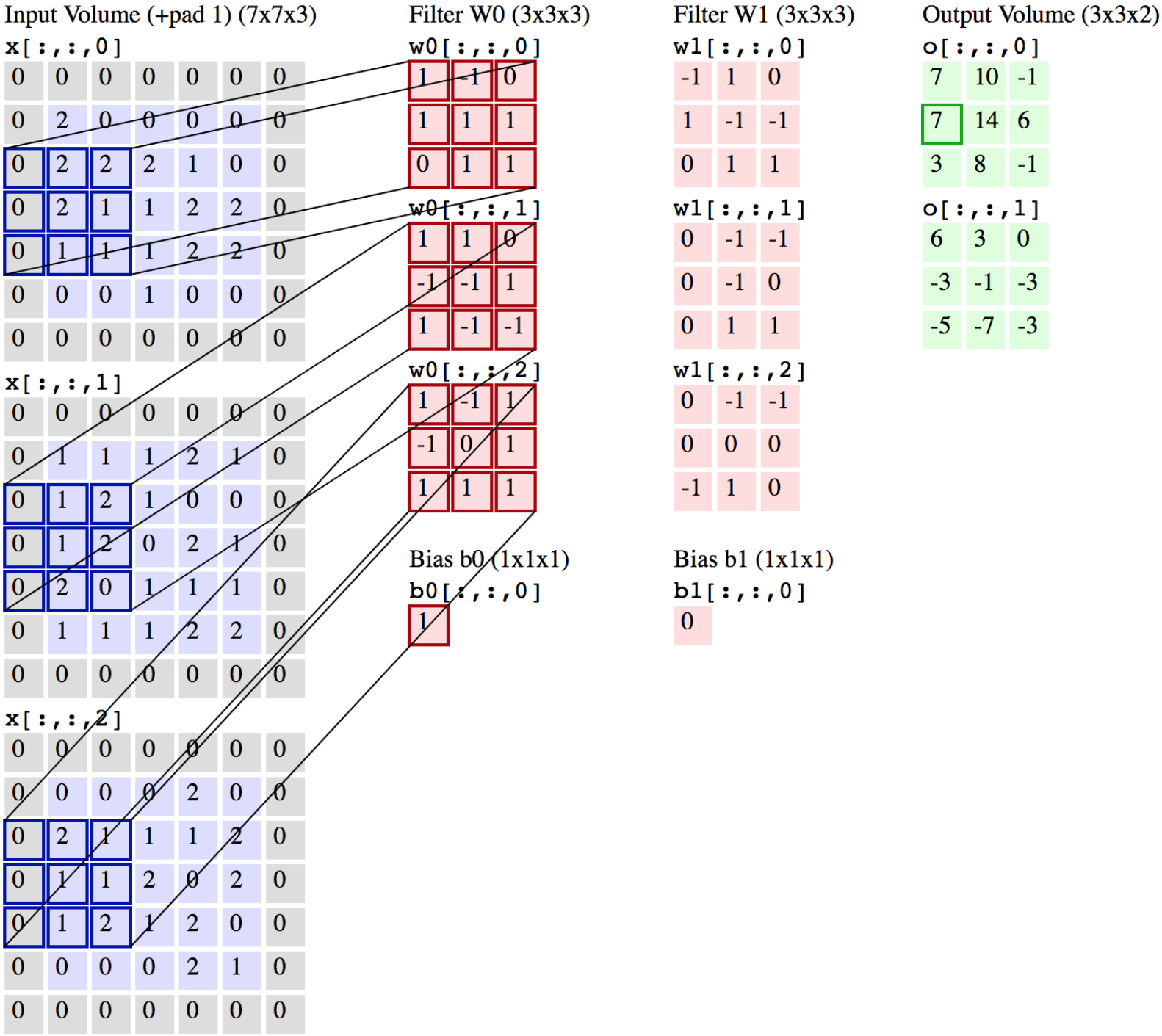
Convolution example (cross-correlation)

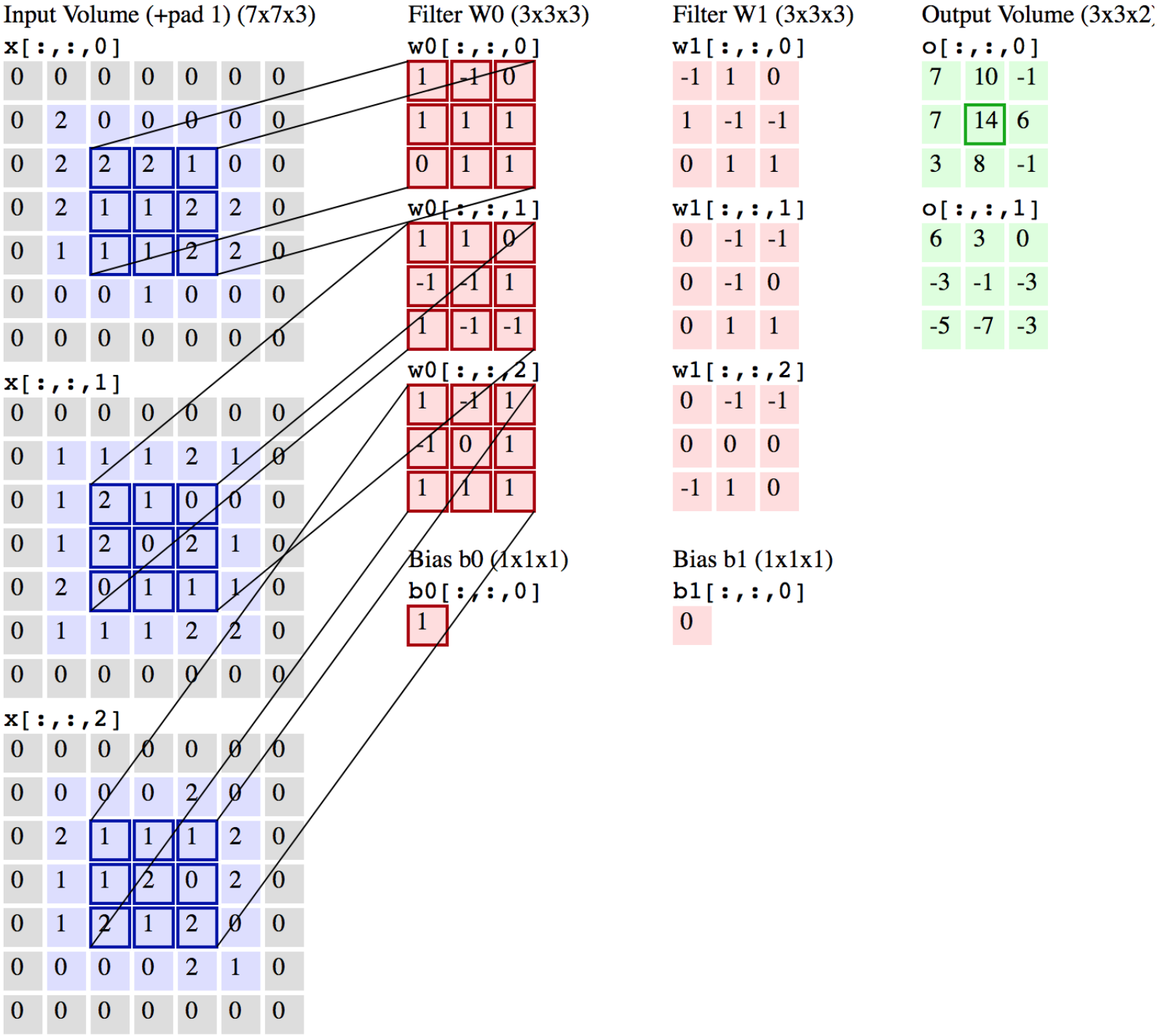
Parameters: F=3, P=1, N=5, N'=7, S=2

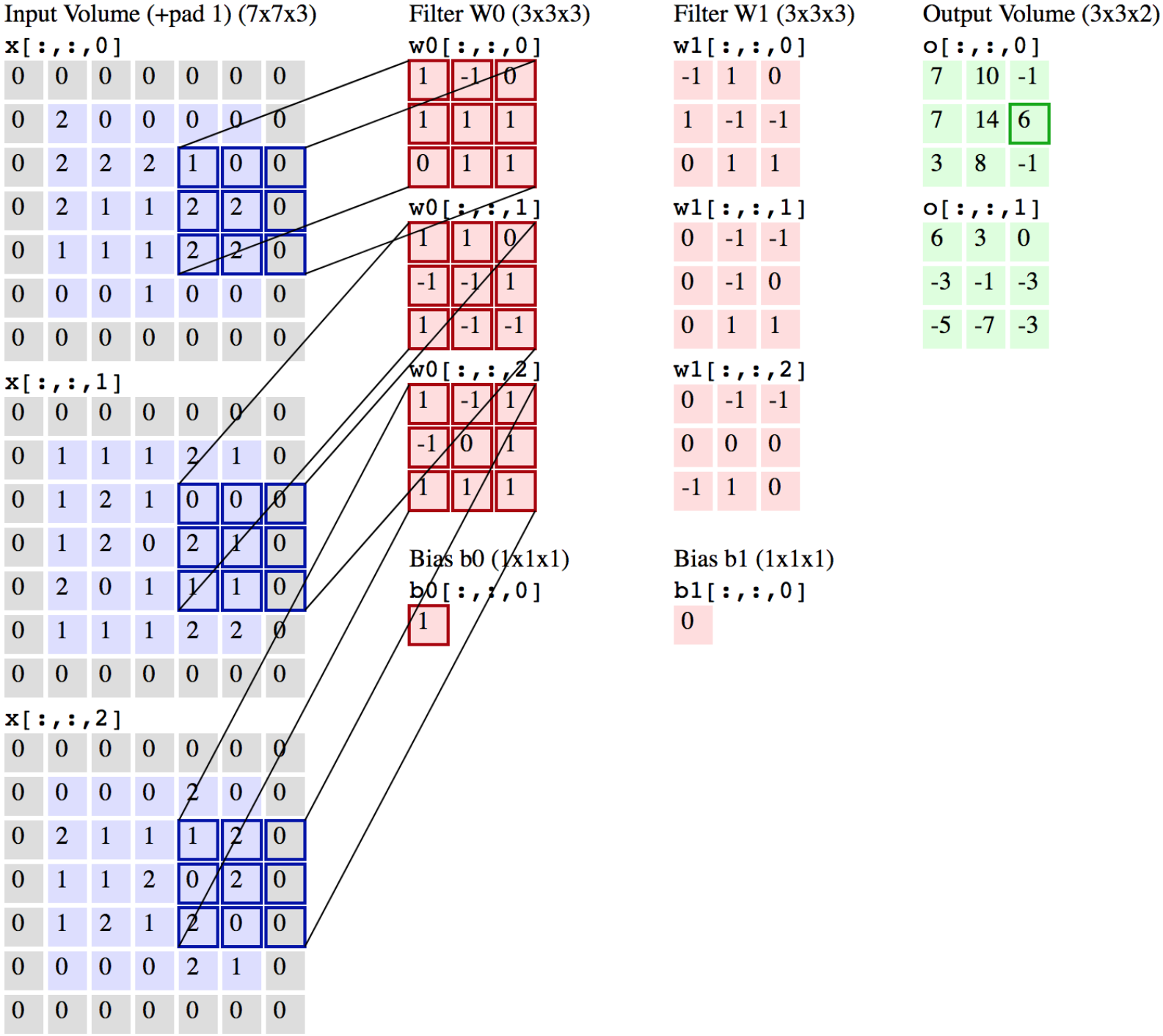
$$S(i,j) = (I \otimes K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i+m, j+n) K(m,n) + b$$

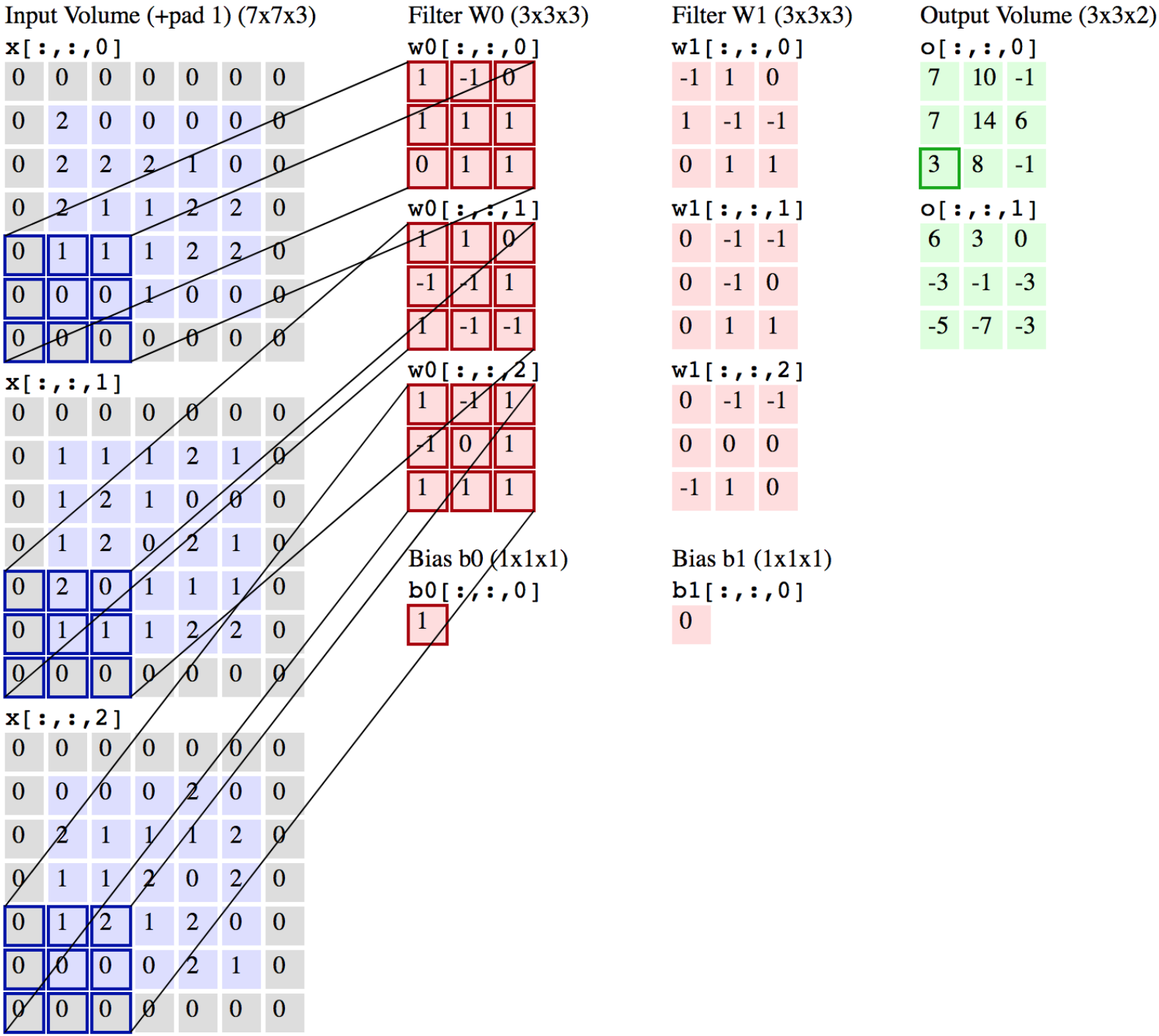

Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 0 0 0 0 0
0 2 2 2 1 0 0
0 2 1 1 2 2 0
0 1 1 1 2 2 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 1 2 1 0 0 0
0 1 2 0 2 1 0
0 2 0 1 1 1 0
0 1 1 1 2 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 0 0 2 0 0
0 2 1 1 1 2 0
0 1 1 2 0 2 0
0 1 2 1 2 0 0
0 0 0 0 2 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 -1 0
1 1 1
0 1 1
w0[:,:,1]
1 1 0
-1 -1 1
1 -1 -1
w0[:,:,2]
1 -1 1
-1 0 1
1 1 1
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
-1 1 0
1 -1 -1
0 1 1
w1[:,:,1]
0 -1 -1
0 -1 0
0 1 1
w1[:,:,2]
0 -1 -1
0 0 0
-1 1 0
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
7 10 -1
7 14 6
3 8 -1
o[:,:,1]
6 3 0
-3 -1 -3
-5 -7 -3

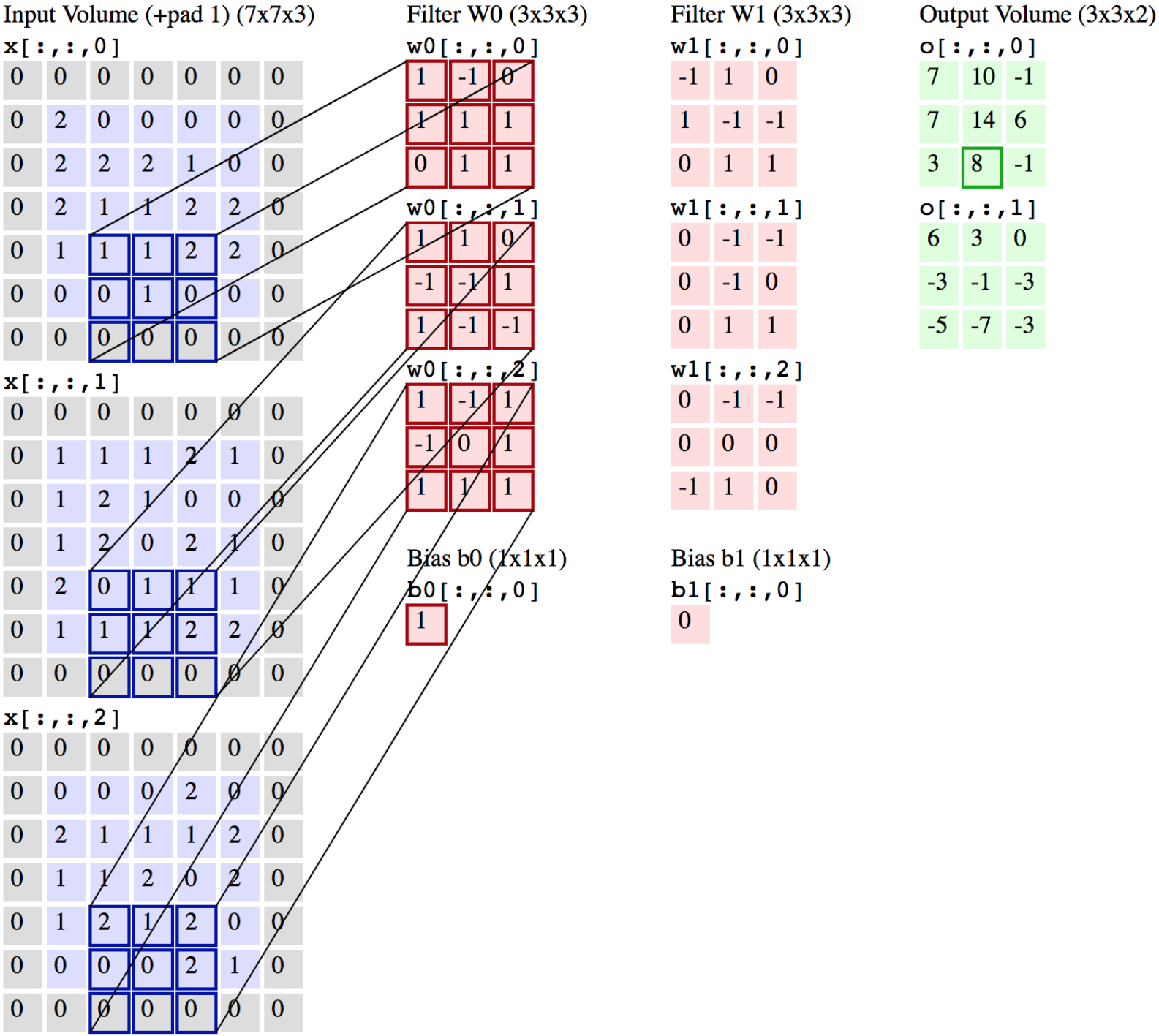

Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 0 0 0 0 0
0 2 2 2 1 0 0
0 2 1 1 2 2 0
0 1 1 1 2 2 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 1 2 1 0 0 0
0 1 2 0 2 1 0
0 2 0 1 1 1 0
0 1 1 1 2 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 0 0 2 0 0
0 2 1 1 1 2 0
0 1 1 2 0 2 0
0 1 2 1 2 0 0
0 0 0 0 2 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 -1 0
1 1 1
0 1 1
w0[:,:,1]
1 1 0
-1 -1 1
1 -1 -1
w0[:,:,2]
1 -1 1
-1 0 1
1 1 1
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
-1 1 0
1 -1 -1
0 1 1
w1[:,:,1]
0 -1 -1
0 -1 0
0 1 1
w1[:,:,2]
0 -1 -1
0 0 0
-1 1 0
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
7 10 -1
7 14 6
3 8 -1
o[:,:,1]
6 3 0
-3 -1 -3
-5 -7 -3

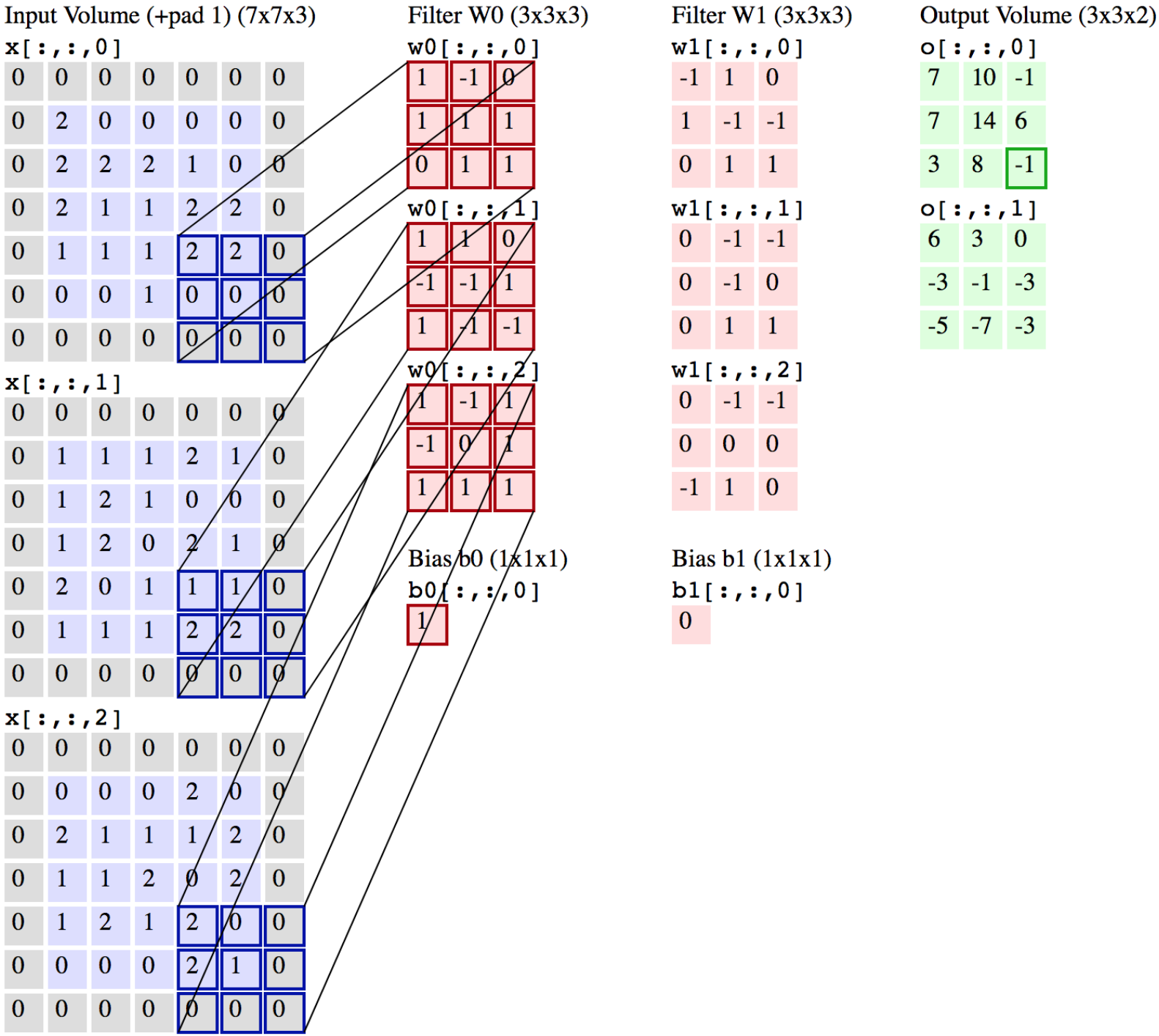

Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 0 0 0 0 0
0 2 2 2 1 0 0
0 2 1 1 2 2 0
0 1 1 1 2 2 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 1 2 1 0 0 0
0 1 2 0 2 1 0
0 2 0 1 1 1 0
0 1 1 1 2 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 0 0 2 0 0
0 2 1 1 1 2 0
0 1 1 2 0 2 0
0 1 2 1 2 0 0
0 0 0 0 2 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 -1 0
1 1 1
0 1 1
w0[:,:,1]
1 1 0
-1 -1 1
1 -1 -1
w0[:,:,2]
1 -1 1
-1 0 1
1 1 1
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
-1 1 0
1 -1 -1
0 1 1
w1[:,:,1]
0 -1 -1
0 -1 0
0 1 1
w1[:,:,2]
0 -1 -1
0 0 0
-1 1 0
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
7 10 -1
7 14 6
3 8 -1
o[:,:,1]
6 3 0
-3 -1 -3
-5 -7 -3

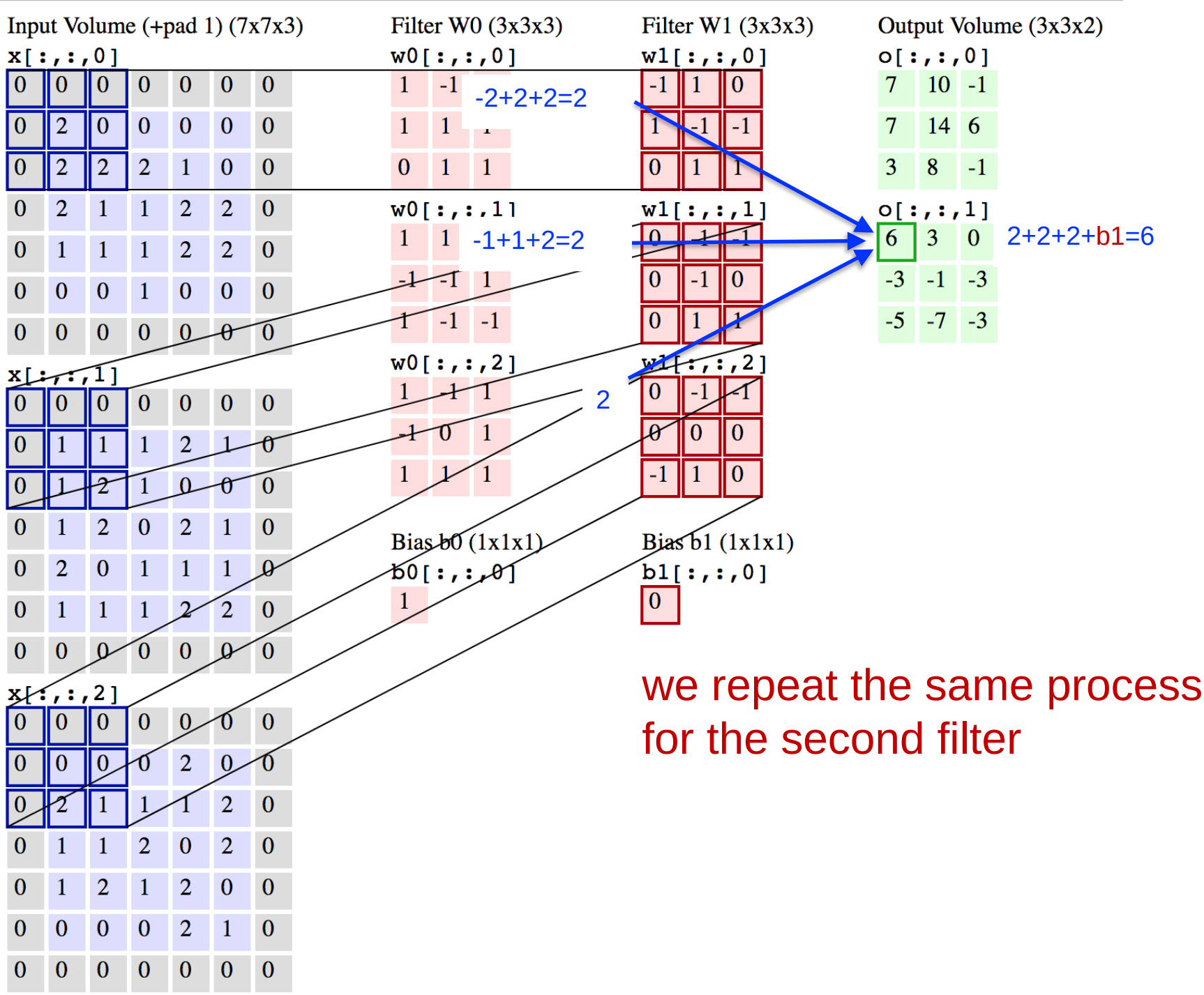

Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 0 0 0 0 0
0 2 2 2 1 0 0
0 2 1 1 2 2 0
0 1 1 1 2 2 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 1 2 1 0 0 0
0 1 2 0 2 1 0
0 2 0 1 1 1 0
0 1 1 1 2 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 0 0 2 0 0
0 2 1 1 1 2 0
0 1 1 2 0 2 0
0 1 2 1 2 0 0
0 0 0 0 2 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 -1 0
1 1 1
0 1 1
w0[:,:,1]
1 1 0
-1 -1 1
1 -1 -1
w0[:,:,2]
1 -1 1
-1 0 1
1 1 1
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
-1 1 0
1 -1 -1
0 1 1
w1[:,:,1]
0 -1 -1
0 -1 0
0 1 1
w1[:,:,2]
0 -1 -1
0 0 0
-1 1 0
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
7 10 -1
7 14 6
3 8 -1
o[:,:,1]
6 3 0
-3 -1 -3
-5 -7 -3

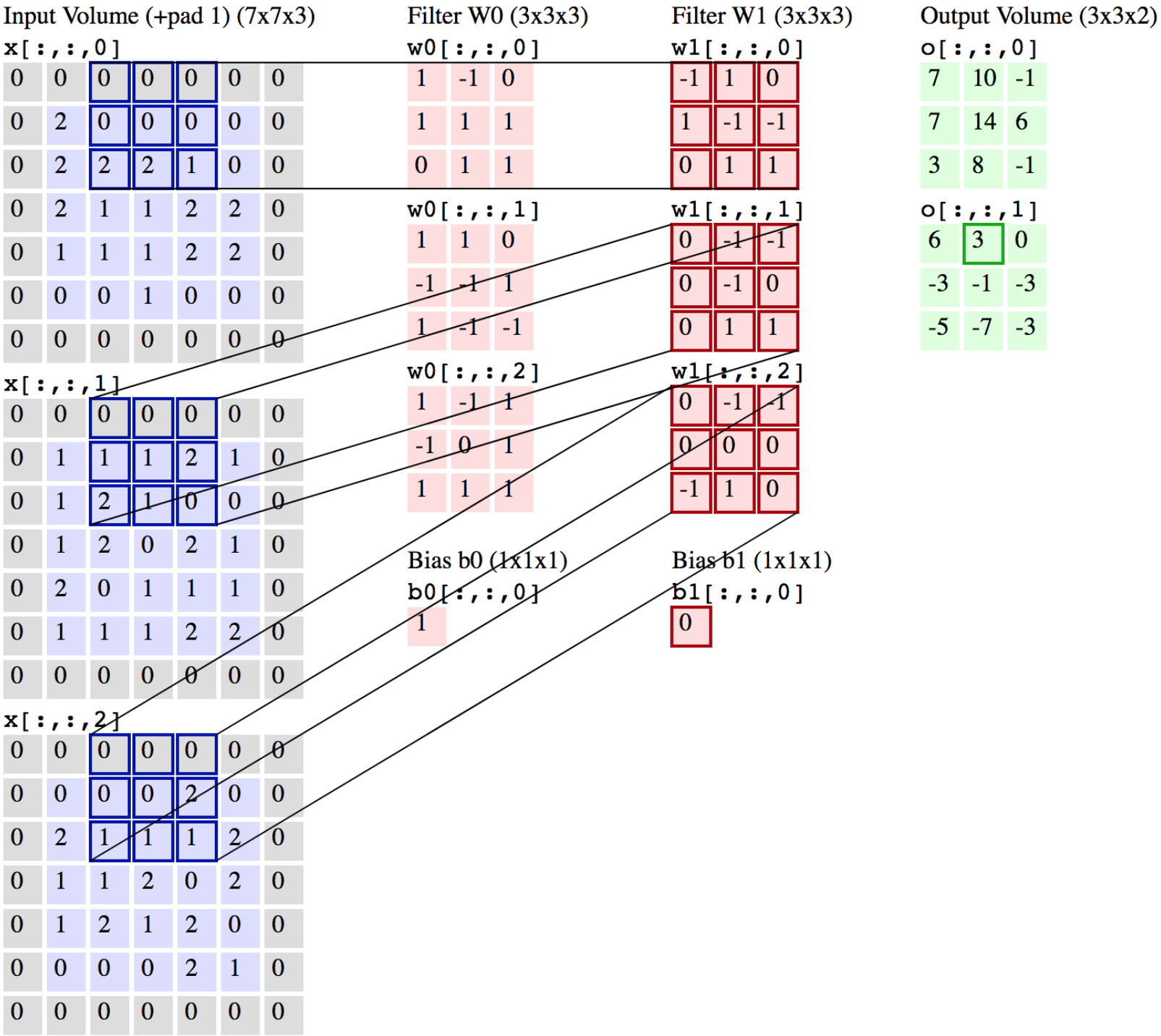

Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 0 0 0 0 0
0 2 2 2 1 0 0
0 2 1 1 2 2 0
0 1 1 1 2 2 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 1 2 1 0 0 0
0 1 2 0 2 1 0
0 2 0 1 1 1 0
0 1 1 1 2 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 0 0 2 0 0
0 2 1 1 1 2 0
0 1 1 2 0 2 0
0 1 2 1 2 0 0
0 0 0 0 2 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 -1 0
1 1 1
0 1 1
w0[:,:,1]
1 1 0
-1 -1 1
1 -1 -1
w0[:,:,2]
1 -1 1
-1 0 1
1 1 1
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
-1 1 0
1 -1 -1
0 1 1
w1[:,:,1]
0 -1 -1
0 -1 0
0 1 1
w1[:,:,2]
0 -1 -1
0 0 0
-1 1 0
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
7 10 -1
7 14 6
3 8 -1
o[:,:,1]
6 3 0
-3 -1 -3
-5 -7 -3

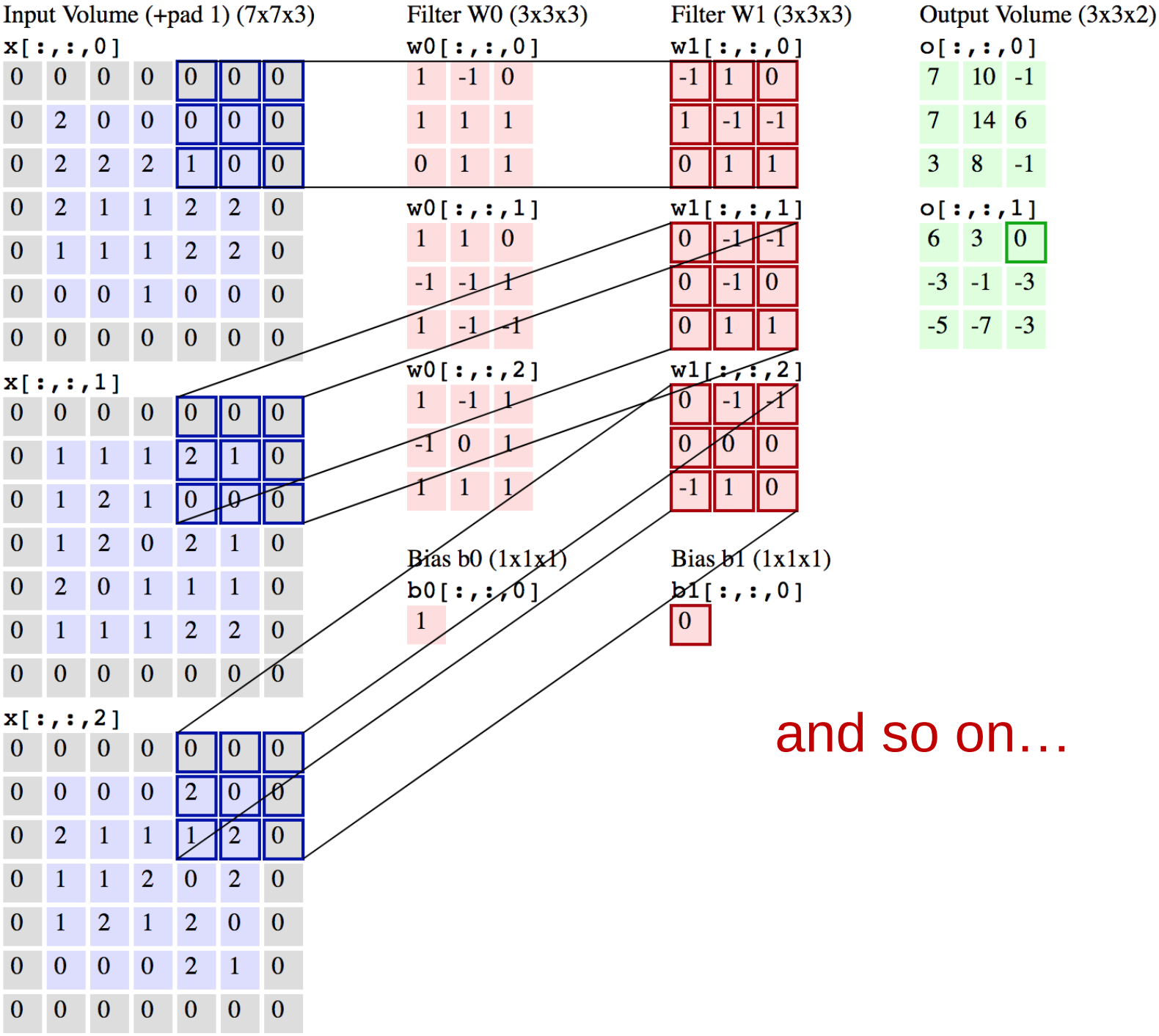

Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 0 0 0 0 0
0 2 2 2 1 0 0
0 2 1 1 2 2 0
0 1 1 1 2 2 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 1 2 1 0 0 0
0 1 2 0 2 1 0
0 2 0 1 1 1 0
0 1 1 1 2 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 0 0 2 0 0
0 2 1 1 1 2 0
0 1 1 2 0 2 0
0 1 2 1 2 0 0
0 0 0 0 2 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 -1 0
1 1 1
0 1 1
w0[:,:,1]
1 1 0
-1 -1 1
1 -1 -1
w0[:,:,2]
1 -1 1
-1 0 1
1 1 1
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
-1 1 0
1 -1 -1
0 1 1
w1[:,:,1]
0 -1 -1
0 -1 0
0 1 1
w1[:,:,2]
0 -1 -1
0 0 0
-1 1 0
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
7 10 -1
7 14 6
3 8 -1
o[:,:,1]
6 3 0
-3 -1 -3
-5 -7 -3


Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 0 0 0 0 0
0 2 2 2 1 0 0
0 2 1 1 2 2 0
0 1 1 1 2 2 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 1 2 1 0 0 0
0 1 2 0 2 1 0
0 2 0 1 1 1 0
0 1 1 1 2 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 0 0 2 0 0
0 2 1 1 1 2 0
0 1 1 2 0 2 0
0 1 2 1 2 0 0
0 0 0 0 2 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 -1 0
1 1 1
0 1 1
w0[:,:,1]
1 1 0
-1 -1 1
1 -1 -1
w0[:,:,2]
1 -1 1
-1 0 1
1 1 1
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
-1 1 0
1 -1 -1
0 1 1
w1[:,:,1]
0 -1 -1
0 -1 0
0 1 1
w1[:,:,2]
0 -1 -1
0 0 0
-1 1 0
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
7 10 -1
7 14 6
3 8 -1
o[:,:,1]
6 3 0
-3 -1 -3
-5 -7 -3


Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 0 0 0 0 0
0 2 2 2 1 0 0
0 2 1 1 2 2 0
0 1 1 1 2 2 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 1 2 1 0 0 0
0 1 2 0 2 1 0
0 2 0 1 1 1 0
0 1 1 1 2 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 0 0 2 0 0
0 2 1 1 1 2 0
0 1 1 2 0 2 0
0 1 2 1 2 0 0
0 0 0 0 2 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 -1 0
1 1 1
0 1 1
w0[:,:,1]
1 1 0
-1 -1 1
1 -1 -1
w0[:,:,2]
1 -1 1
-1 0 1
1 1 1
Bias b0 (1x1x1)
b0[:,:,0]
1
Filter W1 (3x3x3)
w1[:,:,0]
-1 1 0
1 -1 -1
0 1 1
w1[:,:,1]
0 -1 -1
0 -1 0
0 1 1
w1[:,:,2]
0 -1 -1
0 0 0
-1 1 0
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
7 10 -1
7 14 6
3 8 -1
o[:,:,1]
6 3 0
-3 -1 -3
-5 -7 -3


Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 0 0 0 0 0
0 2 2 2 1 0 0
0 2 1 1 2 2 0
0 1 1 1 2 2 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 1 2 1 0 0 0
0 1 2 0 2 1 0
0 2 0 1 1 1 0
0 1 1 1 2 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 0 0 2 0 0
0 2 1 1 1 2 0
0 1 1 2 0 2 0
0 1 2 1 2 0 0
0 0 0 0 2 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 -1
1 1
0 1 1
w0[:,:,1]
1 1
-1 -1 1
1 -1 -1
w0[:,:,2]
1 -1 1
-1 0 1
1 1 1
Filter W1 (3x3x3)
w1[:,:,0]
-1 1 0
1 -1 -1
0 1 1
w1[:,:,1]
0 -1 -1
0 -1 0
0 1 1
w1[:,:,2]
0 -1 -1
0 0 0
-1 1 0
Output Volume (3x3x2)
o[:,:,0]
7 10 -1
7 14 6
3 8 -1
o[:,:,1]
6 3 0
-3 -1 -3
-5 -7 -3
-2+2+2=2
-1+1+2=2
2
2+2+2+b1=6
Bias b0 (1x1x1)
b0[:,:,0]
1
Bias b1 (1x1x1)
b1[:,:,0]
0
we repeat the same process
for the second filter

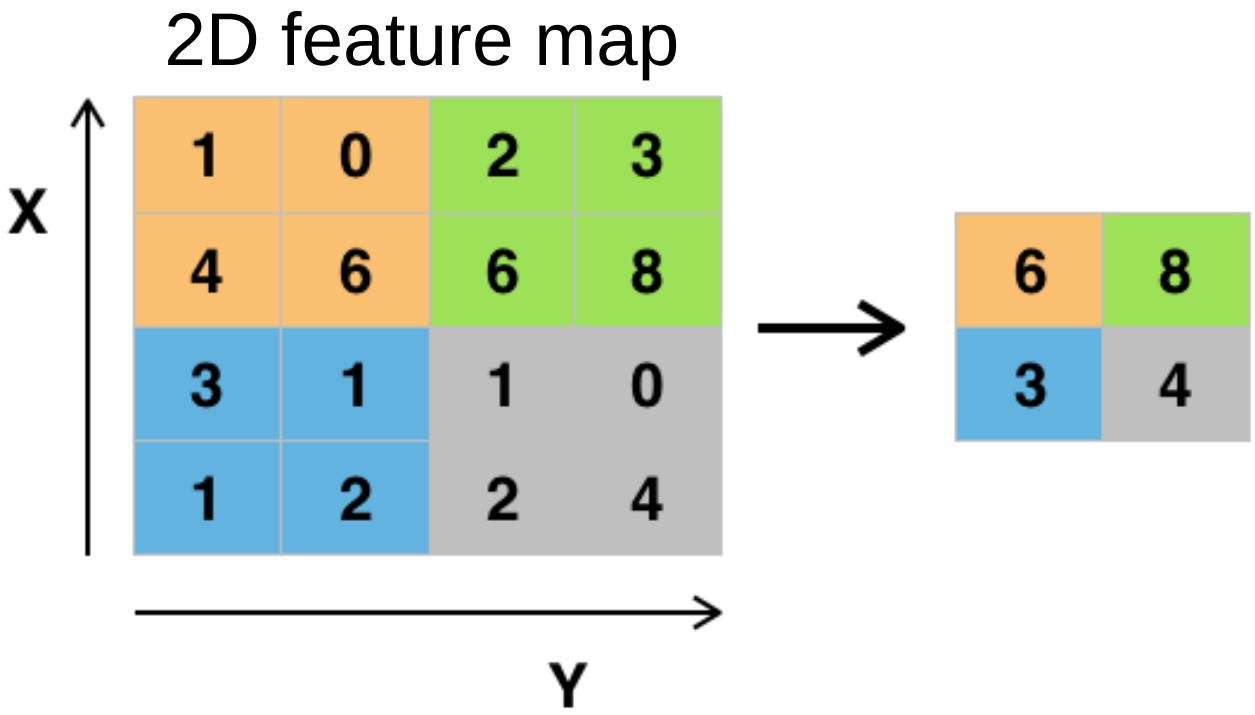

Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 0 0 0 0 0
0 2 2 2 1 0 0
0 2 1 1 2 2 0
0 1 1 1 2 2 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 1 2 1 0 0 0
0 1 2 0 2 1 0
0 2 0 1 1 1 0
0 1 1 1 2 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 0 0 2 0 0
0 2 1 1 1 2 0
0 1 1 2 0 2 0
0 1 2 1 2 0 0
0 0 0 0 2 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 -1 0
1 1 1
0 1 1
w0[:,:,1]
1 1 0
-1 -1 1
1 -1 -1
w0[:,:,2]
1 -1 1
-1 0 1
1 1 1
Filter W1 (3x3x3)
w1[:,:,0]
-1 1 0
1 -1 -1
0 1 1
w1[:,:,1]
0 -1 -1
0 -1 0
0 1 1
w1[:,:,2]
0 -1 -1
0 0 0
-1 1 0
Bias b0 (1x1x1)
b0[:,:,0]
1
Bias b1 (1x1x1)
b1[:,:,0]
0
Output Volume (3x3x2)
o[:,:,0]
7 10 -1
7 14 6
3 8 -1
o[:,:,1]
6 3 0
-3 -1 -3
-5 -7 -3


Input Volume (+pad 1) (7x7x3)
x[:,:,0]
0 0 0 0 0 0 0
0 2 0 0 0 0 0
0 2 2 2 1 0 0
0 2 1 1 2 2 0
0 1 1 1 2 2 0
0 0 0 1 0 0 0
0 0 0 0 0 0 0
x[:,:,1]
0 0 0 0 0 0 0
0 1 1 1 2 1 0
0 1 2 1 0 0 0
0 1 2 0 2 1 0
0 2 0 1 1 1 0
0 1 1 1 2 2 0
0 0 0 0 0 0 0
x[:,:,2]
0 0 0 0 0 0 0
0 0 0 0 2 0 0
0 2 1 1 1 2 0
0 1 1 2 0 2 0
0 1 2 1 2 0 0
0 0 0 0 2 1 0
0 0 0 0 0 0 0
Filter W0 (3x3x3)
w0[:,:,0]
1 -1 0
1 1 1
0 1 1
w0[:,:,1]
1 1 0
-1 -1 1
1 -1 -1
w0[:,:,2]
1 -1 1
-1 0 1
1 1 1
Filter W1 (3x3x3)
w1[:,:,0]
-1 1 0
1 -1 -1
0 1 1
w1[:,:,1]
0 -1 -1
0 -1 0
0 1 1
w1[:,:,2]
0 -1 -1
0 0 0
-1 1 0
Output Volume (3x3x2)
o[:,:,0]
7 10 -1
7 14 6
3 8 -1
o[:,:,1]
6 3 0
-3 -1 -3
-5 -7 -3
Bias b0 (1x1x1)
b0[:,:,0]
1
Bias b1 (1x1x1)
b1[:,:,0]
0
and so on…

CNN hyper-parameters

- Depth
  - It corresponds to the number of filters we would like to use
  - Each filter looks for something different in the input
- Stride
  - Controls the way in which the filter convolves around the input volume
  - Normally larger strides if
    - We want receptive fields to overlap less
    - We want to decrease the spatial dimension
- Zero-padding
  - Main reason for using it is to preserve volume size from input to output
  - Without, the volume size would reduce too fast → not good to build deep networks (with many layers)

Pooling example: max pooling



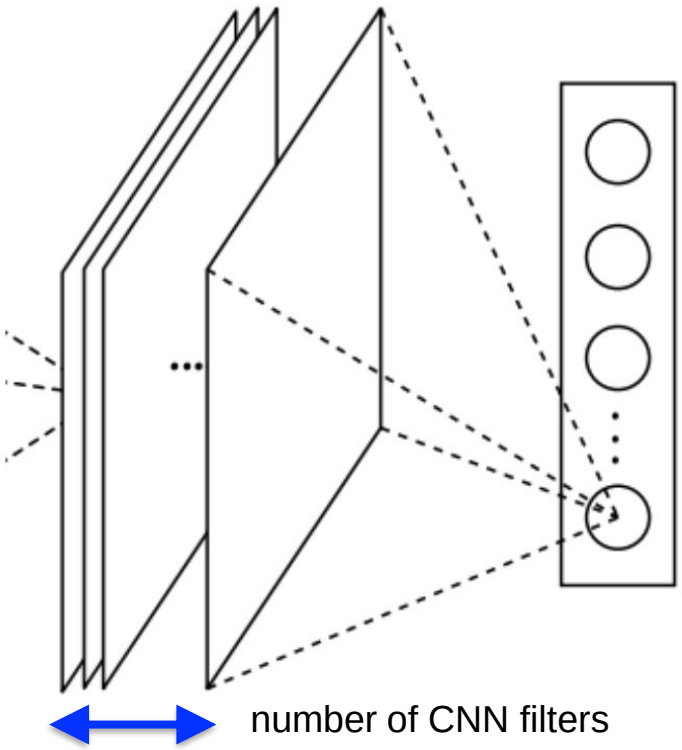
Example of maxpool with a 2x2 filter and a stride of 2

Pooling example: global average pooling

- Given a feature vector (1D) or a feature map (2D)
- Averages the values in it and returns a single value
- 2D CNN: returns a single value for each 2D feature map
  - In a 2D CNN → 1 map per filter

Main advantages

- No weights are to be trained
- Does not depend on ordering
- Output: order independent feature vector



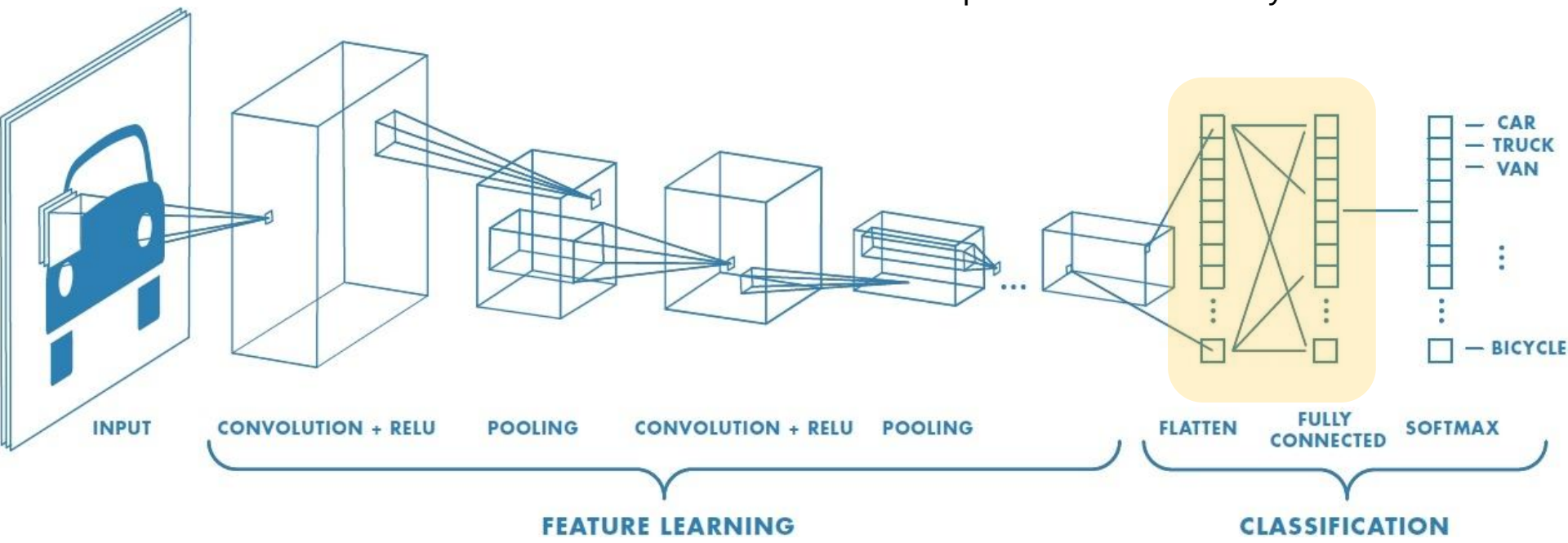
Three stages of a typical CONV layer

- A typical layer in a CNN consists of three stages
  - **First stage (convolution):** a layer performs several convolutions in parallel to produce a set of (linear) activations (i.e., the output of the convolution operation that we have seen previously)
  - **Second stage (detection):** each linear activation is run through a nonlinear activation function, such as the rectified linear activation function (ReLU)
  - **Third stage (pooling):** to modify the output of the second stage further. A pooling function replaces the output of a net at a certain location with a summary statistics of the nearby outputs. For example the max pooling function reports the maximum output within a rectangular neighborhood
    - other popular pooling functions: average over a rectangular neighborhood, the L_2 norm of a rectangular neighborhood or a weighted average based on the distance from the central point (pixel)

CNN - the whole picture

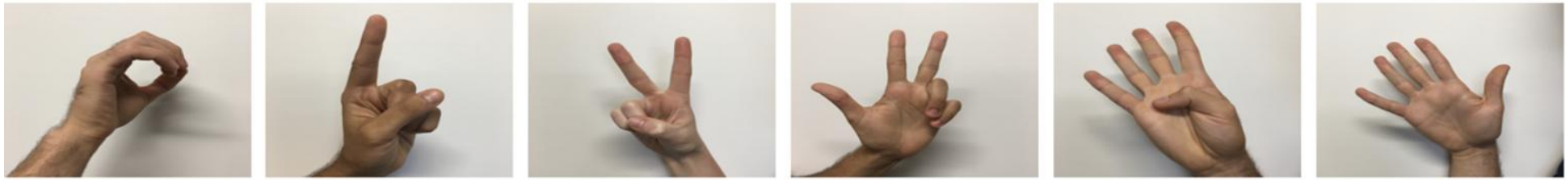
Flattening

- from 3D volume to vector
- which is the input of the last dense layer



Lab 2 (Part I) – Nov. 12

- Implement a CNN-based classifier
  - TensorFlow: low-level Python programming framework that gives you flexibility in defining the neural network structures
  - The challenge:
    - hand gesture recognition using the SIGNS dataset (six-classes classification)
  - You will learn to:
    - implement a CNN-based classifier using the Sequential class
    - implement your own training pipeline



TensorFlow

- https://www.tensorflow.org
- TensorFlow is an open-source machine learning platform written in Python 3
- The TensorFlow API consists of different modules to implement a complete learning pipeline, from data handling to model design and training
- https://www.tensorflow.org/api_docs/python/tf

Modules

`audio` module: Public API for tf.audio namespace.

`autodiff` module: Public API for tf.autodiff namespace.

`autograph` module: Conversion of plain Python into TensorFlow graph code.

`bitwise` module: Operations for manipulating the binary representations of integers.

`compat` module: Compatibility functions.

`config` module: Public API for tf.config namespace.

`data` module: `tf.data.Dataset` API for input pipelines.

`debugging` module: Public API for tf.debugging namespace.

`distribute` module: Library for running a computation across multiple devices.

`dtypes` module: Public API for tf.dtypes namespace.

`errors` module: Exception types for TensorFlow errors.

`estimator` module: Estimator: High level tools for working with models.

`experimental` module: Public API for tf.experimental namespace.

`feature_column` module: Public API for tf.feature_column namespace.

`graph_util` module: Helpers to manipulate a tensor graph in python.

`image` module: Image ops.

`initializers` module: Keras initializer serialization / deserialization.

`io` module: Public API for tf.io namespace.

`keras` module: Implementation of the Keras API meant to be a high-level API for TensorFlow.

`linalg` module: Operations for linear algebra.

`lite` module: Public API for tf.lite namespace.

`lookup` module: Public API for tf.lookup namespace.

`losses` module: Built-in loss functions.

`math` module: Math Operations.

`metrics` module: Built-in metrics.

TF Keras

- Different modules for neural network design and implementation

- https://www.tensorflow.org/api_docs/python/tf/keras

Modules

`activations` module: Built-in activation functions.

`applications` module: Keras Applications are canned architectures with pre-trained weights

`backend` module: Keras backend API.

`callbacks` module: Callbacks: utilities called at certain points during model training.

`constraints` module: Constraints: functions that impose constraints on weight values.

`datasets` module: Public API for tf.keras.datasets namespace.

`estimator` module: Keras estimator API.

`experimental` module: Public API for tf.keras.experimental namespace.

`initializers` module: Keras initializer serialization / deserialization.

`layers` module: Keras layers API.

`losses` module: Built-in loss functions.

`metrics` module: Built-in metrics.

`mixed_precision` module: Public API for tf.keras.mixed_precision namespace.

`models` module: Code for model cloning, plus model-related API entries.

`optimizers` module: Built-in optimizer classes.

`preprocessing` module: Keras data preprocessing utils.

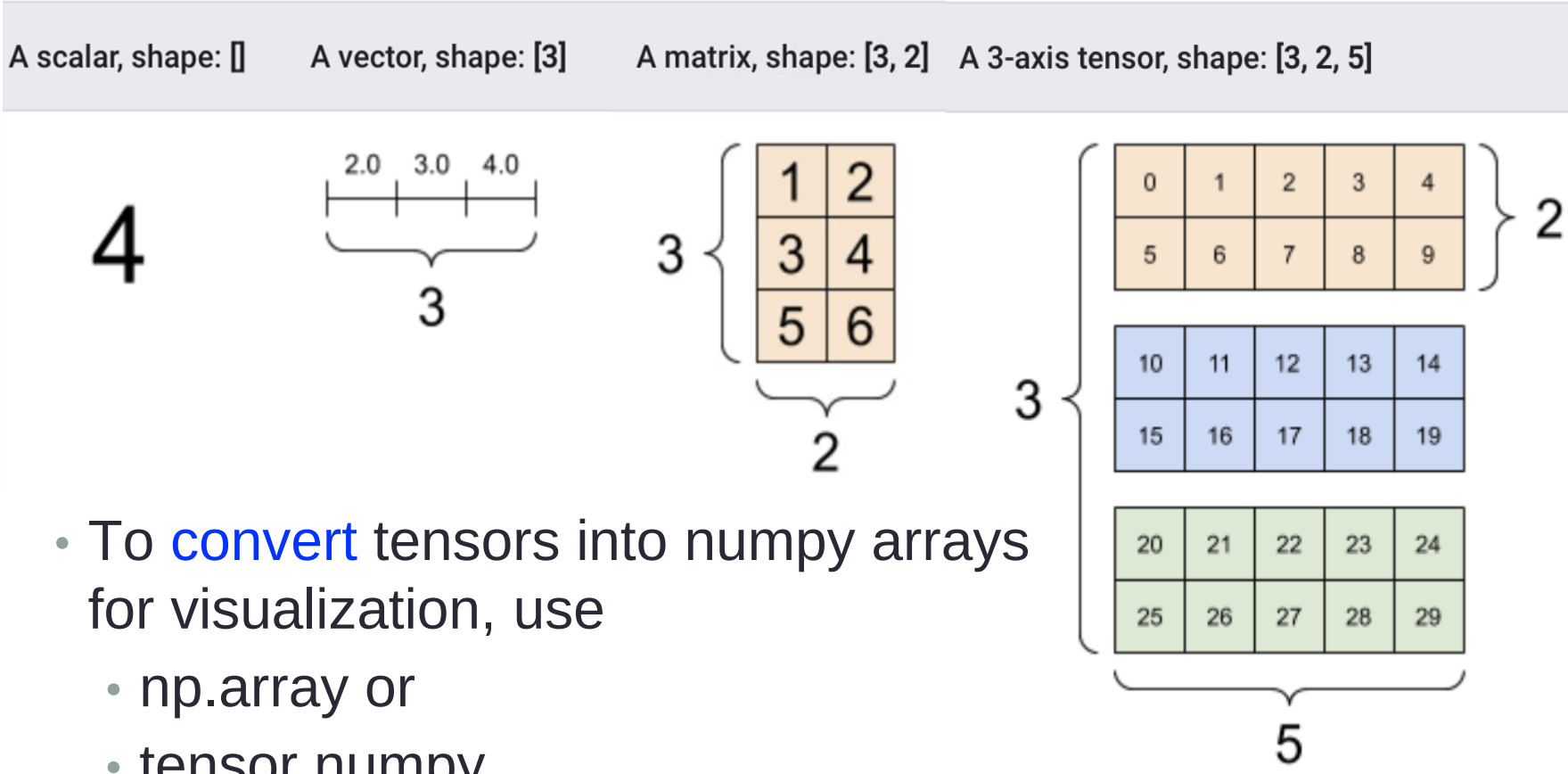
`regularizers` module: Built-in regularizers.

`utils` module: Public API for tf.keras.utils namespace.

`wrappers` module: Public API for tf.keras.wrappers namespace.

Tensor

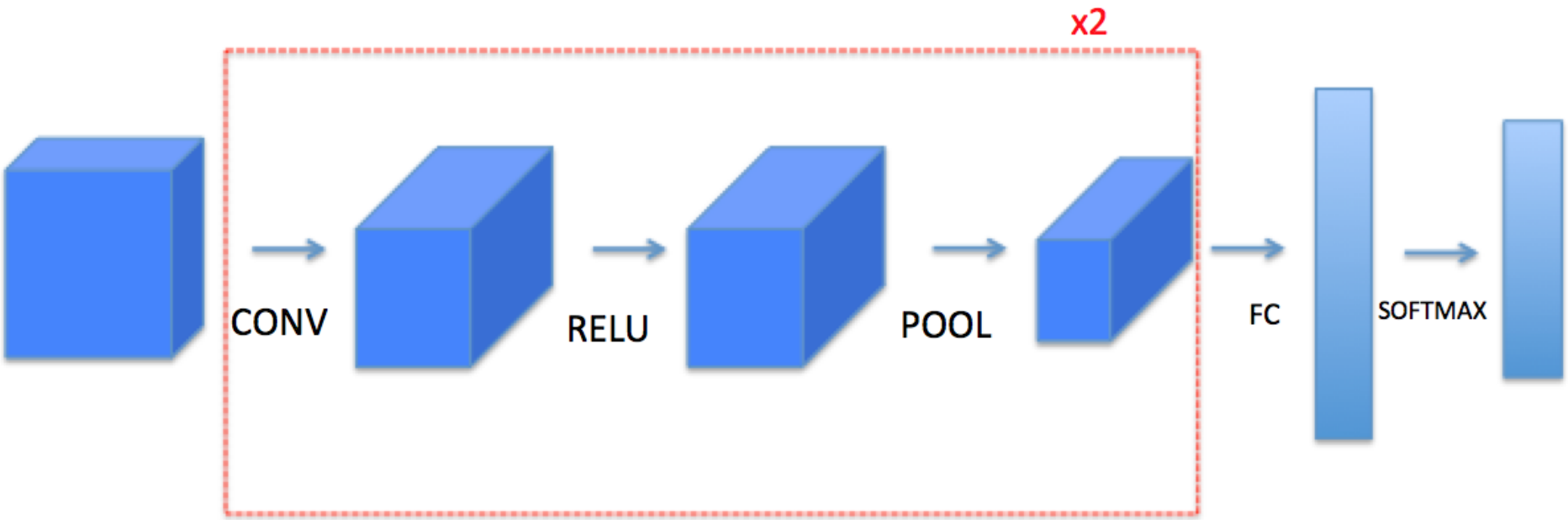
- Tensors are multi-dimensional arrays with a uniform type https://www.tensorflow.org/guide/tensor



- To convert tensors into numpy arrays for visualization, use
  - np.array or
  - tensor.numpy

CNN in TensorFlow

- Model that you will implement to solve this problem



Build a neural network model

- There are two options to build a neural network model in TensorFlow with the Keras API

Classes

`class Model`: `Model` groups layers into an object with training and inference features.

`class Sequential`: `Sequential` groups a linear stack of layers into a `tf.keras.Model`.

- During this lab you will use
  - the Sequential class in Part I
  - the Model class in Part II

Sequential class (Part I)

- tf.keras.Sequential https://www.tensorflow.org/api_docs/python/tf/keras/Sequential

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, 3, activation='relu',
                           kernel_regularizer=tf.keras.regularizers.l2(0.02),
                           input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(10)
])
```

Model class (Part II)

- Keras functional API
  https://www.tensorflow.org/guide/keras/functional
- can handle models with non-linear topology, shared layers, and multiple inputs or outputs

```
from tensorflow import keras
from tensorflow.keras import layers
encoder_input = keras.Input(shape=(28, 28, 1), name='img')
x = layers.Conv2D(16, 3, activation='relu')(encoder_input)
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.MaxPooling2D(3)(x)
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.Conv2D(16, 3, activation='relu')(x)
encoder_output = layers.GlobalMaxPooling2D()(x)

encoder = keras.Model(encoder_input, encoder_output, name='encoder')
```

Train a neural network model

- There are two options to train a neural network model in TensorFlow with the Keras API

- During this lab you will
  - implement your own training pipeline (Part I)
  - use built-in methods using the default parameters or changing them (Part II)

Implement your training pipeline

- https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch
  - forward the input through the network
  - compute the loss
  - use tf.GradientTape to record operations during the forward step https://www.tensorflow.org/api_docs/python/tf/GradientTape
  - compute the gradients
  - update the parameters using an optimizer

```
for x, y in dataset:
  with tf.GradientTape() as tape:
    # training=True is only needed if there are layers with different
    # behavior during training versus inference (e.g. Dropout).
    prediction = model(x, training=True)
    loss = loss_fn(prediction, y)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))
```

Implement your training pipeline

- Note: if there are layers with different behavior during training versus inference (e.g. Dropout), you should set the training parameter to different values in the two phases

```
train_out = model(train_data, training=True)
test_out = model(test_data, training=False)
```

Use built in methods

- https://www.tensorflow.org/guide/keras/train_and_evaluate
  - model.compile(…)
  - model.fit(…)
  - model.evaluate(…)
  - model.predict(…)

```
model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
model.fit(data, labels, epochs=10, batch_size=32,
          validation_data=(val_data, val_labels))
```

Use built in methods

- For the optimizer and the loss function you can
  - use the default parameters

```
### START CODE HERE ### (1 line)
happyModel.compile(optimizer = "adam", loss = "binary_crossentropy", metrics = ["accuracy"])
### END CODE HERE ###
```

  - change the optimizer's and loss's parameters:

```
### START CODE HERE ### (2 lines of code)
adam_optimizer = tf.keras.optimizers.Adam(learning_rate=0.0005)
loss_funct = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
### END CODE HERE ###

network_model_2.compile(optimizer = adam_optimizer, loss = loss_funct, metrics = ["accuracy"])
```

Lab 2 (Part II)

- In the second part of the lab you will use the Malaria dataset (https://www.tensorflow.org/datasets/catalog/malaria)

- You will create a TensorFlow Dataset to process the data
- You can go through the full documentation at:
  - https://www.tensorflow.org/guide/data
  - https://www.tensorflow.org/api_docs/python/tf/data/Dataset

(see also next slide)

tf.data.Dataset

- Cache the dataset

This is one of the most interesting features of the TensorFlow Dataset API. With a single line of code, you can implement an advanced caching system.

Let's consider a practical use case. In this example, the normalization step can be performed a single time on the entire dataset, since the result is always the same (*deterministic preprocessing*). Repeating these steps every time you load data is a waste of computational time and is highly inefficient. To avoid multiple executions of the same function we can leverage the *cache* method. In this way, the output of the dataset is only evaluated the first time, and all the subsequent calls will load data from the stored cache. The cache can store partial results both in a file or in memory (RAM), see the documentation for more details.

IMPORTANT NOTE: The order of the dataset transformations is very important, especially when using a caching system. We will see below a practical example.

- Shuffle

This allows the data to be sampled in a different order on every epoch. Documentation here.

NOTE: The `shuffle()` method requires a buffer to be filled **before** it can start to sample data (the buffer size is set by the *buffer_size* parameter). In this case, we are setting the shuffle buffer size equal to the dataset size. So, before starting to sample, the dataset must load all the data, and it can require some time in the beginning, especially if you are still creating the cache.

- Repeat the dataset indefinitely

The `repeat()` method simply tells the dataset to start again when it runs out of samples. Documentation here. Without this step, each file would be loaded only one time during training, resulting in an error in the `fit()` method (as the last batch will be smaller than the others).

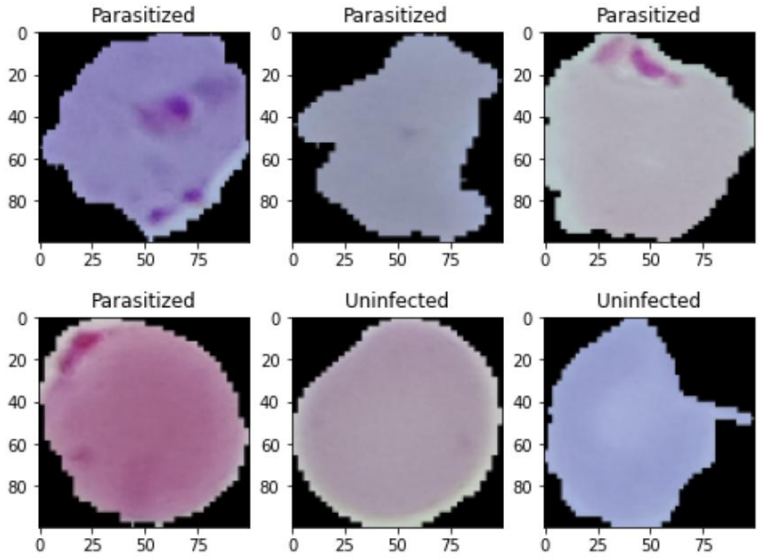
- Batch

Put together *batch_size* samples into a single batch of data, which will be the input of the network. The shape of the data changes from (100, 100, 3) to (batch_size, 100, 100, 3).

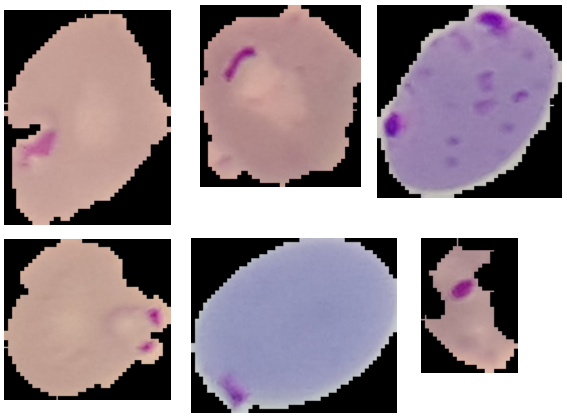
- Prefetch

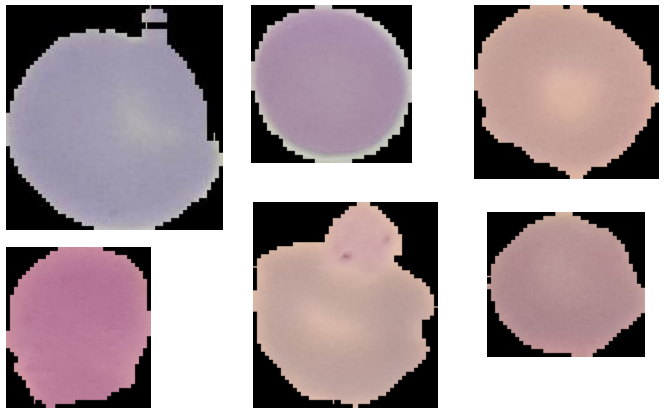
While a process is training your network, this operation allows loading and processing the subsequent chunk of data at the same time, greatly improving the idle time.

Lab 2 (Part II)



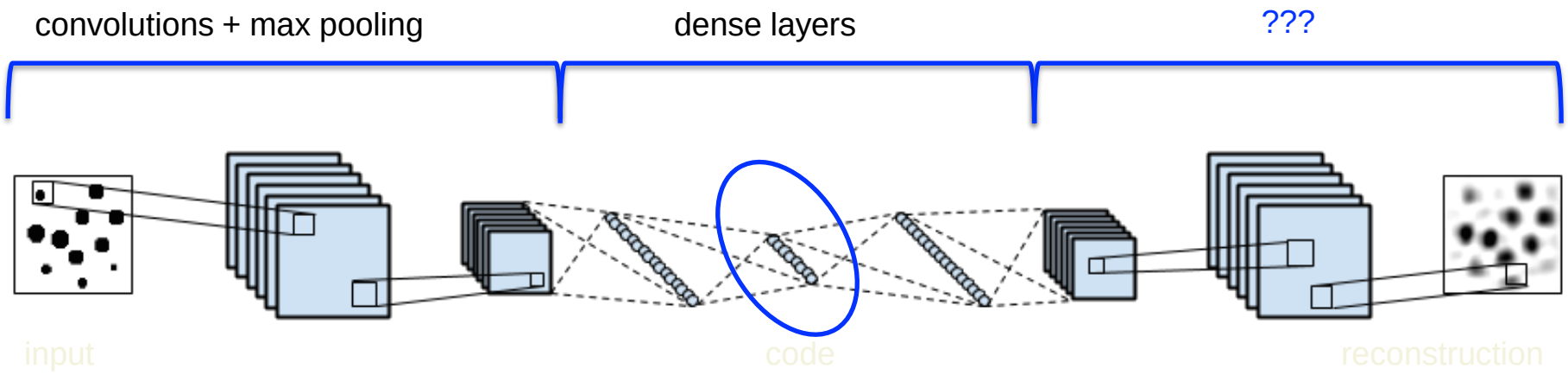
- CNN based autoencoder
  - TensorFlow
  - The challenge:
    - cell image dimensionality reduction
  - You will learn to:
    - implement a CNN based autoencoder using TensorFlow
    - use the autoencoder to denoise noisy images





CNN-based autoencoders

- autoencoder = encoder + decoder networks
- in a CNN autoencoder, the encoder and the decoder are composed of convolutional layers
- useful when dealing with images
- applications: image denoising, image coloring, semantic segmentation, similarity assessment and many others!
- problem:

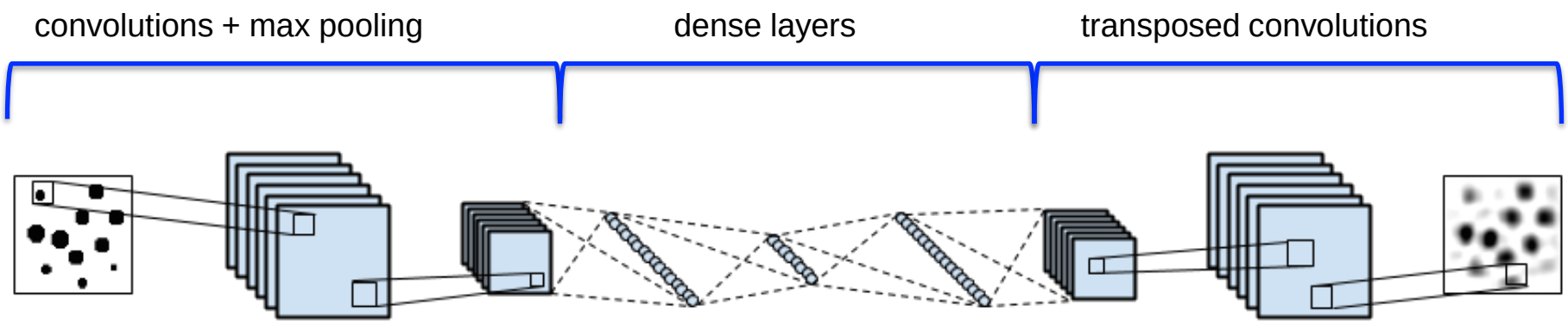


Convolutions

- convolution + max pooling layers -> reduce the dimensionality of the input
- how to reconstruct the input image at the output?

Transposed convolutions 1/4

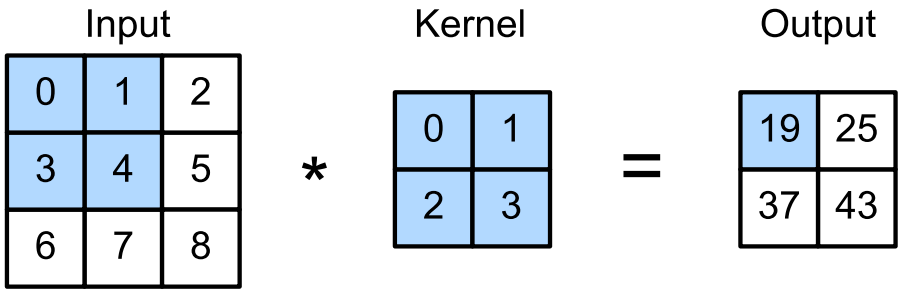
- We can use transposed convolutions



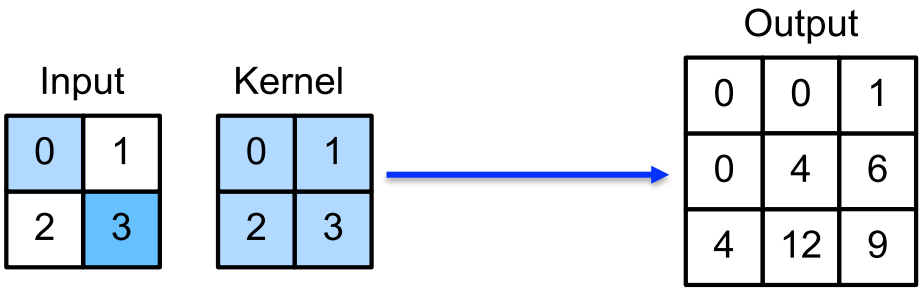
- TensorFlow Keras layer https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2DTranspose

Transposed convolutions 2/4

- convolution:



- transposed convolution: allows recovering the shape of the initial feature map



[Dumoulin2016] A guide to convolution arithmetic for deep learning
https://arxiv.org/abs/1603.07285

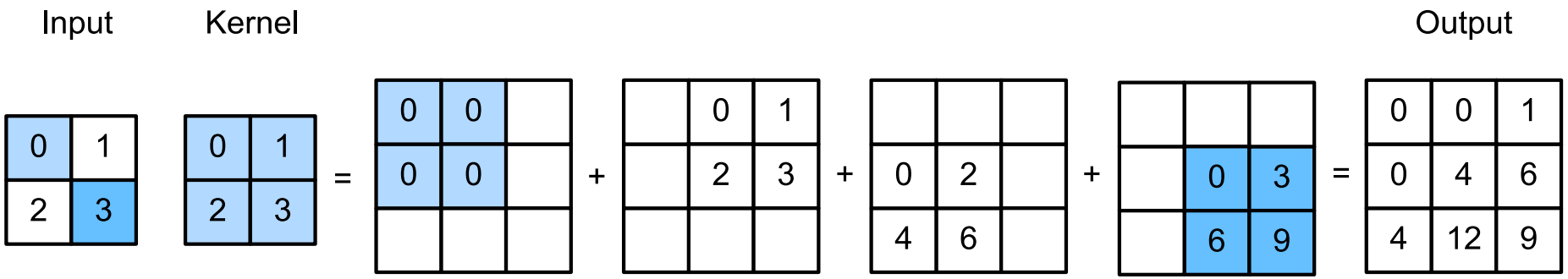
Transposed convolutions 3/4

1. Take a kernel -> learnable parameters

2. Multiply the kernel by each element in the input and place the results in the output matrix using strides

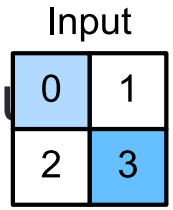
Transposed convolutions 4/4

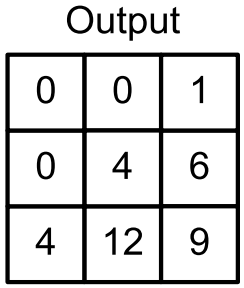
3. Sum all the outputs in the same position to obtain the result of the transposed convolution



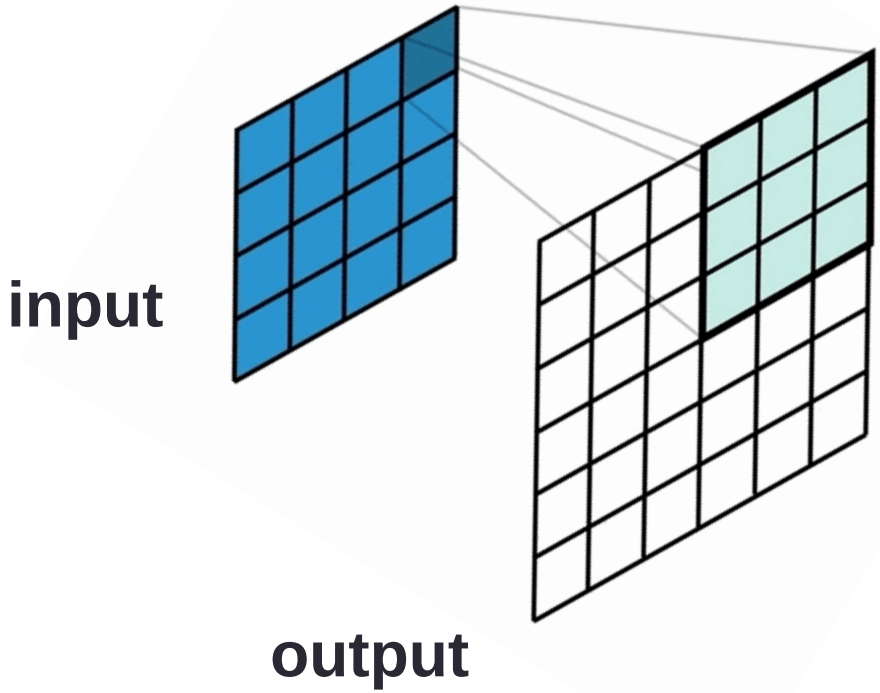
The kernel shape and the stride parameter are selected by considering

- the shape of the input
- the desired shape of the outpu



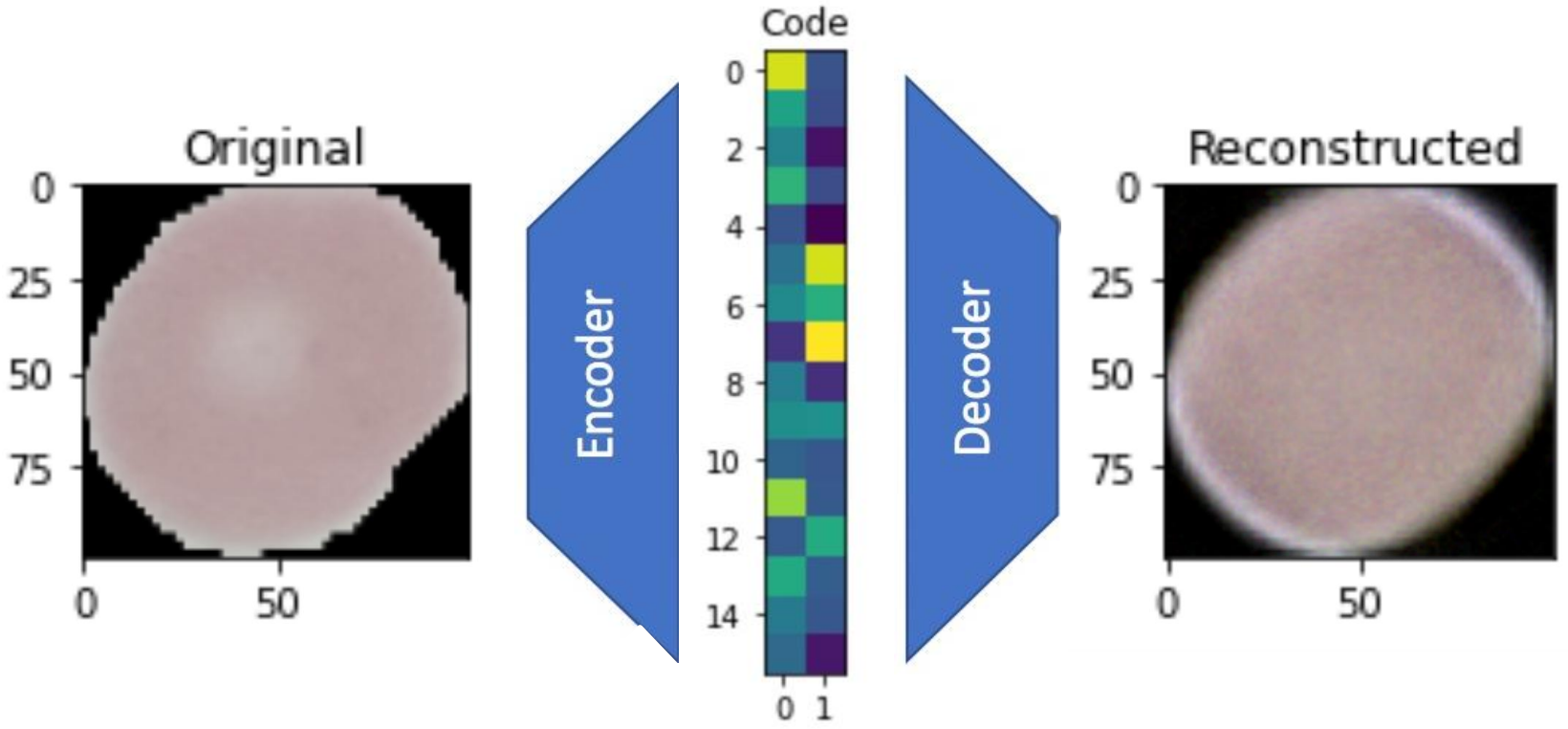


Transposed convolutions



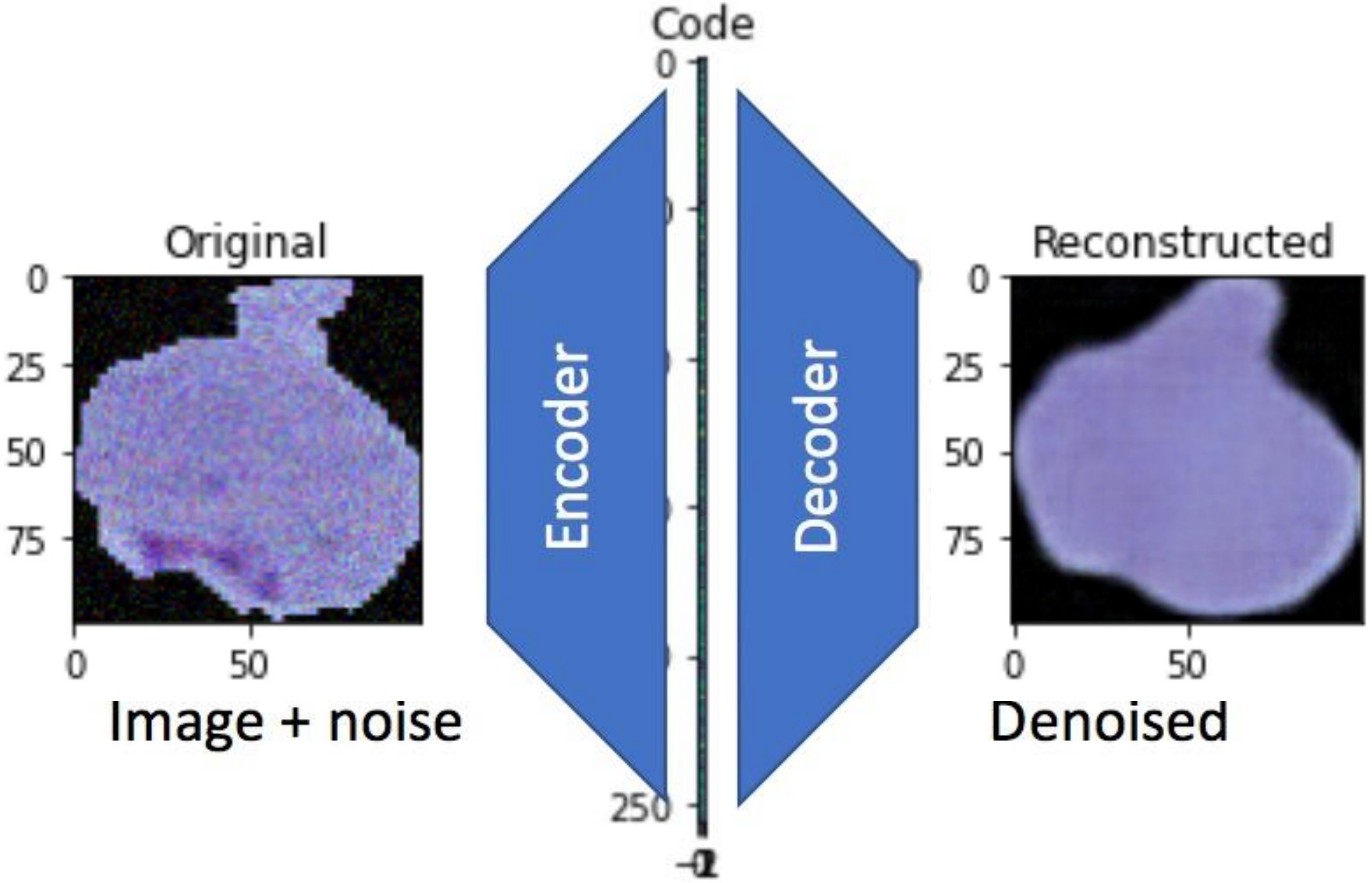
1. CNN-autoencoder design

- Design a CNN-based autoencoder (encoder + decoder)
- Train the network to reconstruct cell images



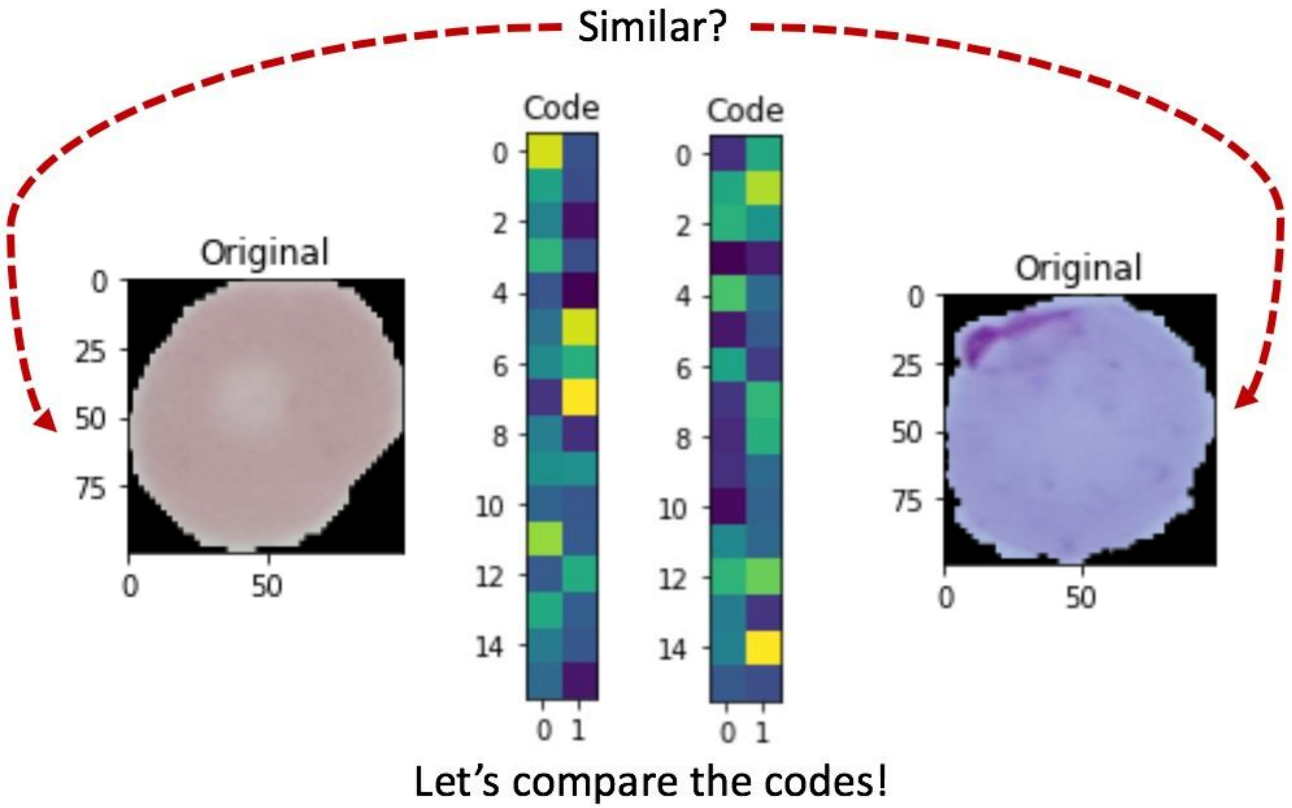
2. Application: denoising

- use the CNN-autoencoder as a denoising autoencoder
- obtain as output a denoised version of the input



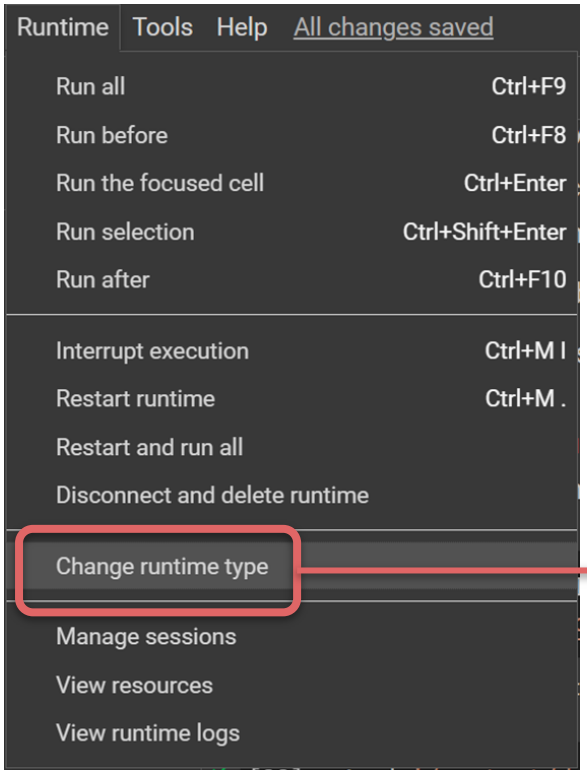
3. Application: image retrieval

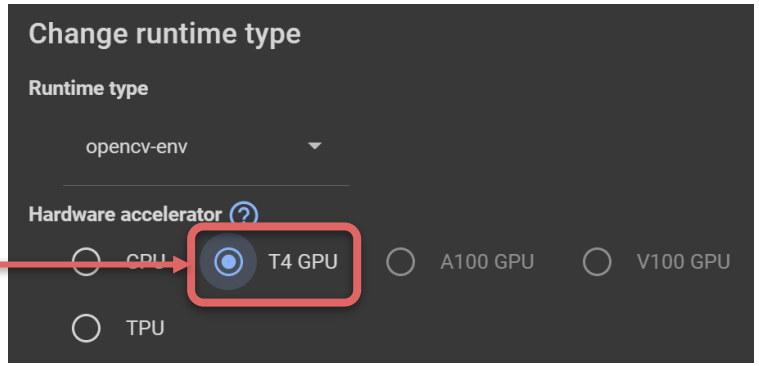
- Use the encoder to extract internal codes
- Use the codes to compare the images and search for similar images in the dataset



Google Colab Runtime

- Remember to switch the runtime type in colab from CPU to GPU, in order to access better computational resources





- In the version of Colab that is free of charge, access to expensive resources like GPUs is heavily restricted and dynamically allocated.

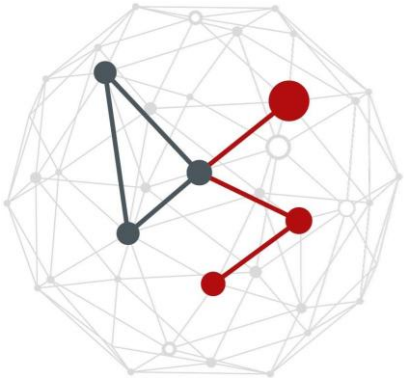
To get the most out of Colab:

- Consider closing your Colab tabs when you are done with your work
- Avoid opting for GPUs or extra memory when it is not needed for your work. This will make it less likely that you will run into usage limits within Colab

MIME
Master's degree ICT Internet Multimedia Engineering

LAB 2: CNN for supervised and self-supervised learning

Eleonora Cicciarella, Cesare Bidini
eleonora.cicciarella@phd.unipd.it
cesare.bidini@phd.unipd.it
University of Padova, IT



DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO MATEMATICA

UNIVERSITÀ DEGLI STUDI DI PADOVA