Deep Convolutional Neural Networks – Part 1

Machine Learning for Engineering Applications

Fall 2023

Purpose

- Convolutional neural networks (CNNs) are a family of models that were inspired by how the visual cortex of human brain works when recognizing objects

- CNNs break down features to learn “*shapes*” of the deconstructed input

- Weights in the ANN-structure of the CNN are adjusted to each feature and learns how to recognize many pieces in parallel

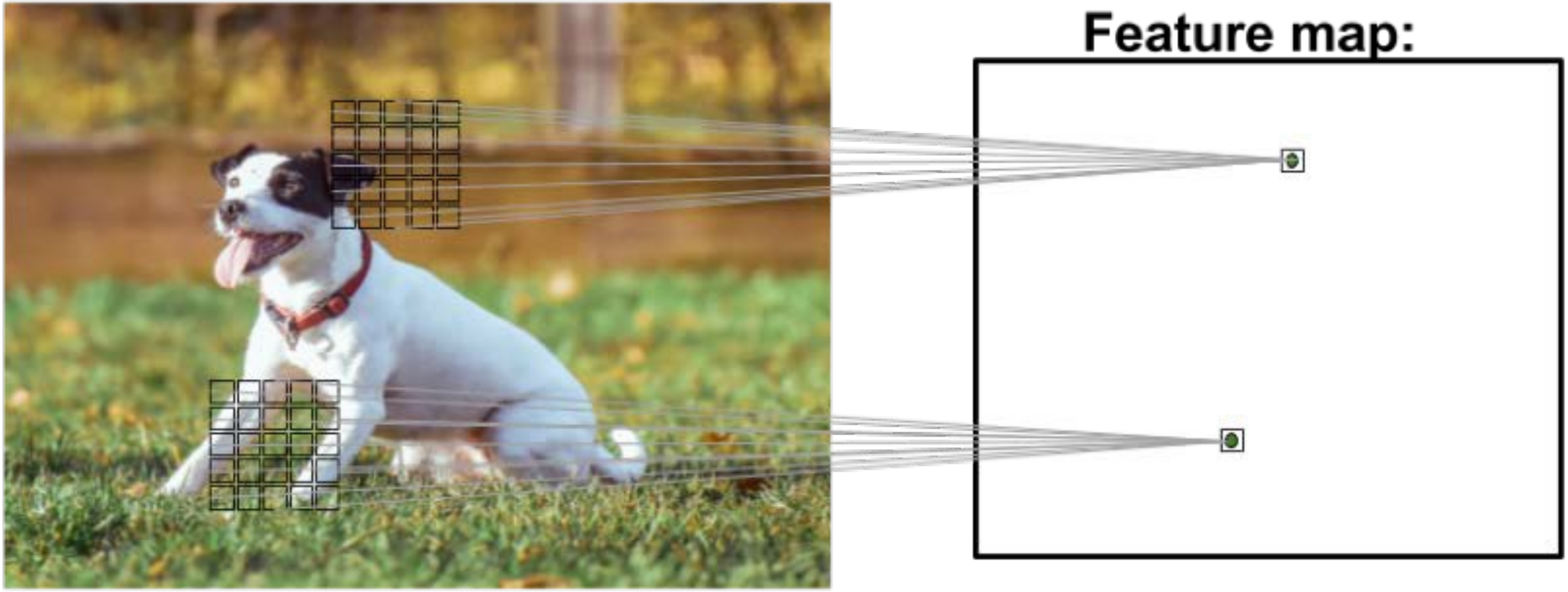
Learning Features

- Neural networks are able to automatically learn the features from raw data that are most useful for a particular task.

- It's common to consider a neural network as a feature extraction technique

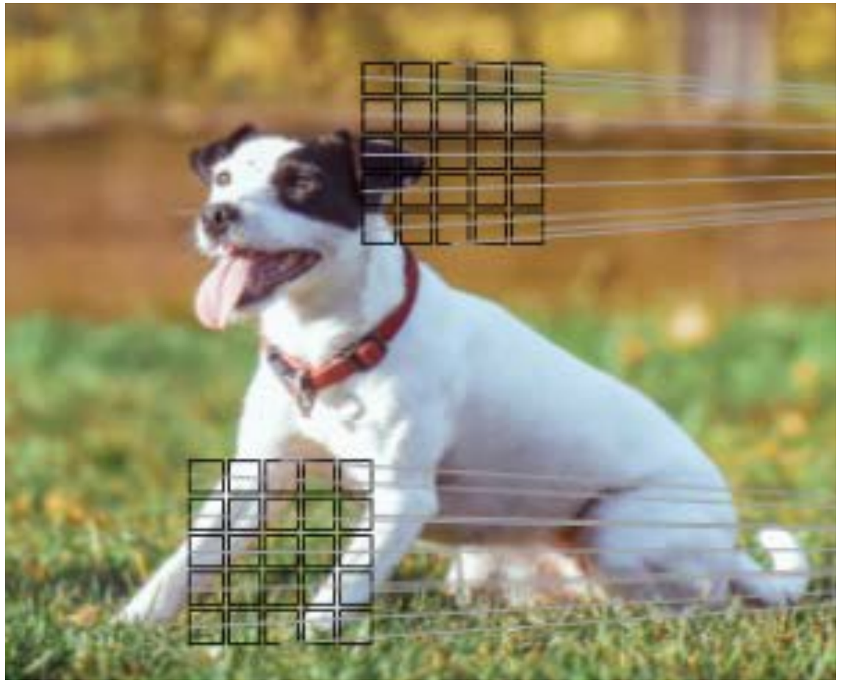
- CNNs construct a so-called *feature hierarchy*

- Combining the low-level features in a layer-wise fashion to form high-level features


Feature map:

- This local patch of pixels is referred to as the local receptive field

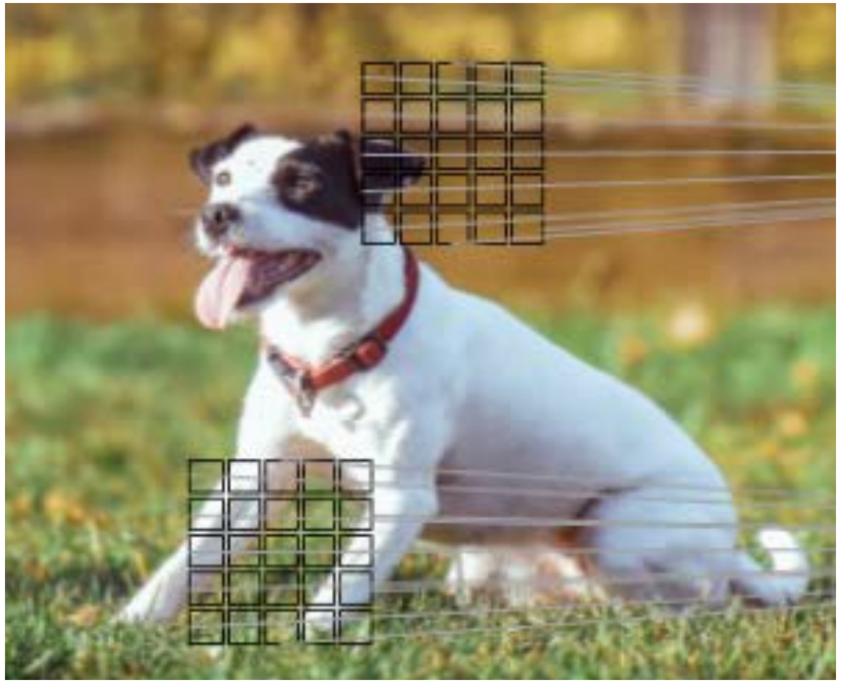
- **Sparse-connectivity**: A single element in the feature map is connected to only a small patch of pixels.



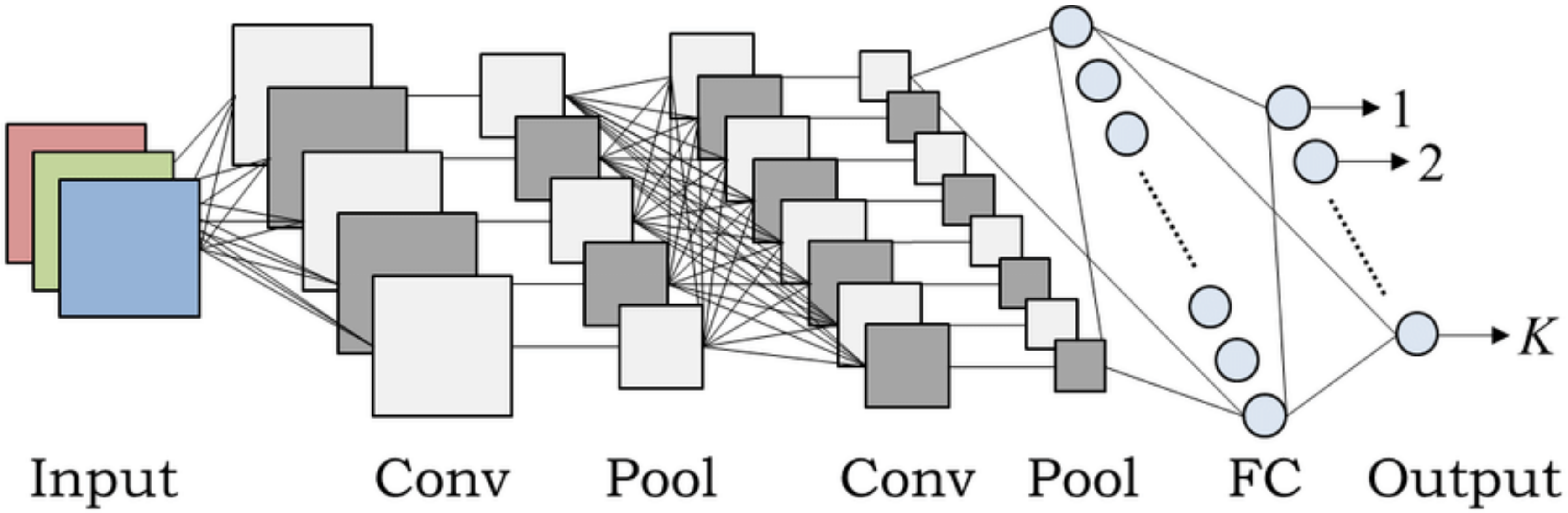
- **Parameter-sharing**: The same weights are used for different patches of the input image.

- This helps to reduce the computation

- **Assumption**: neighboring pixels look the same

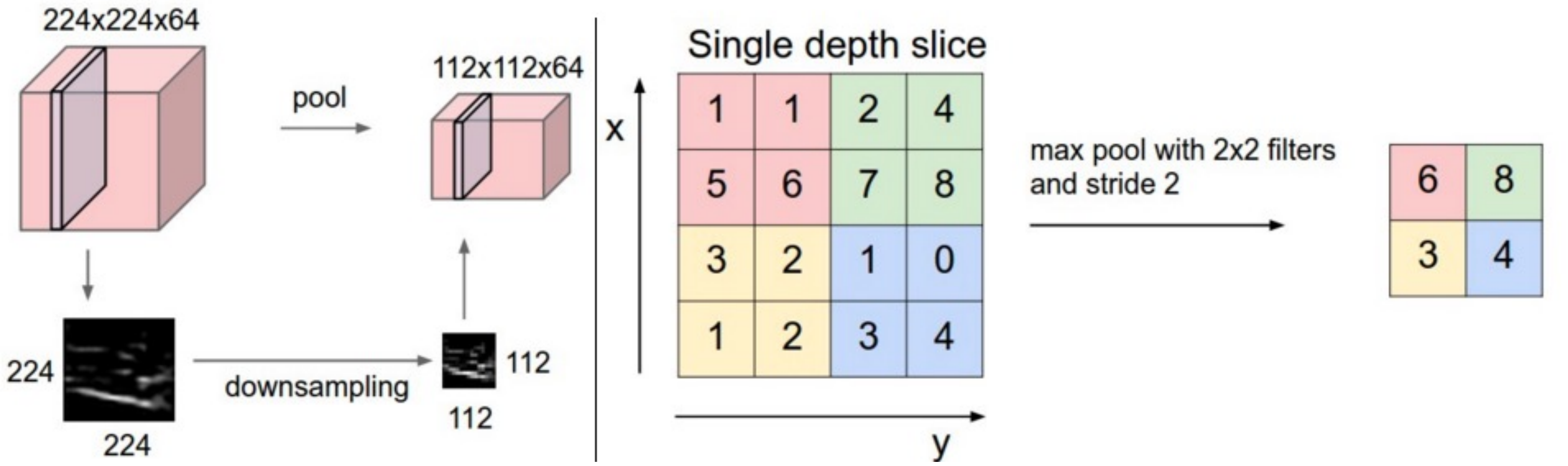


- CNNs are composed of several Convolutional (***conv***) layers
- Subsampling (aka - Pooling (***Pool***))
- Dropout layers (*overfitting fix*)
- Fully Connected (***FC***) layers at the end



- **Pooling**: It does not have a learning components

- Technique to "*pool*" the pixel values and determine a representation of it



1-Dimentional Convolution

- A discrete convolution for two one-dimensional vectors x and w is denoted by :

$$\boldsymbol{y = x * w}$$

- “*” – Not the multiplication operation, it is the convolution operations
- Vector *x* is the input
- *w* is called the filter or kernel

- A discrete convolution for two one-dimensional vectors *x* and *w* is denoted by :

$$\boldsymbol{y = x * w}$$

- The mathematical representation:

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \rightarrow \boldsymbol{y}[i] = \sum_{k=-\infty}^{+\infty} \boldsymbol{x}[i-k]\boldsymbol{w}[k]$$

- The index *i* runs through each element of the output vector *y*

- **Problem**: The input is not infinite

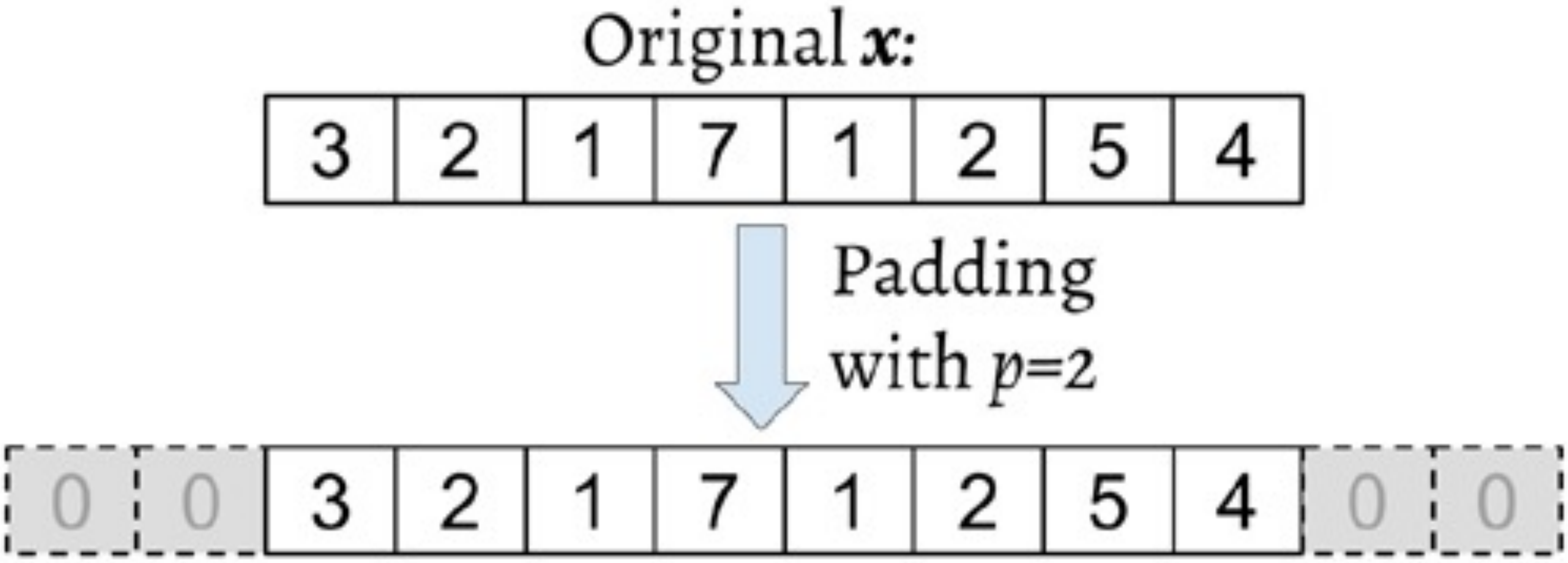
$$y = x * w \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]w[k]$$

- To correctly compute the summation → it is assumed that *x* and *w* are filled with zeros.

- This will result in an output vector *y* that also has infinite size with lots of zeros as well

- *This is what is known as **zero-padding** (or just **padding**)*

- *This is what is known as* ***zero-padding*** *(or just* ***padding****)*



- Assume (as an example):

- The original input *x* and filter *w* have *n* and *m*-elements, respectively, where $m \le n$

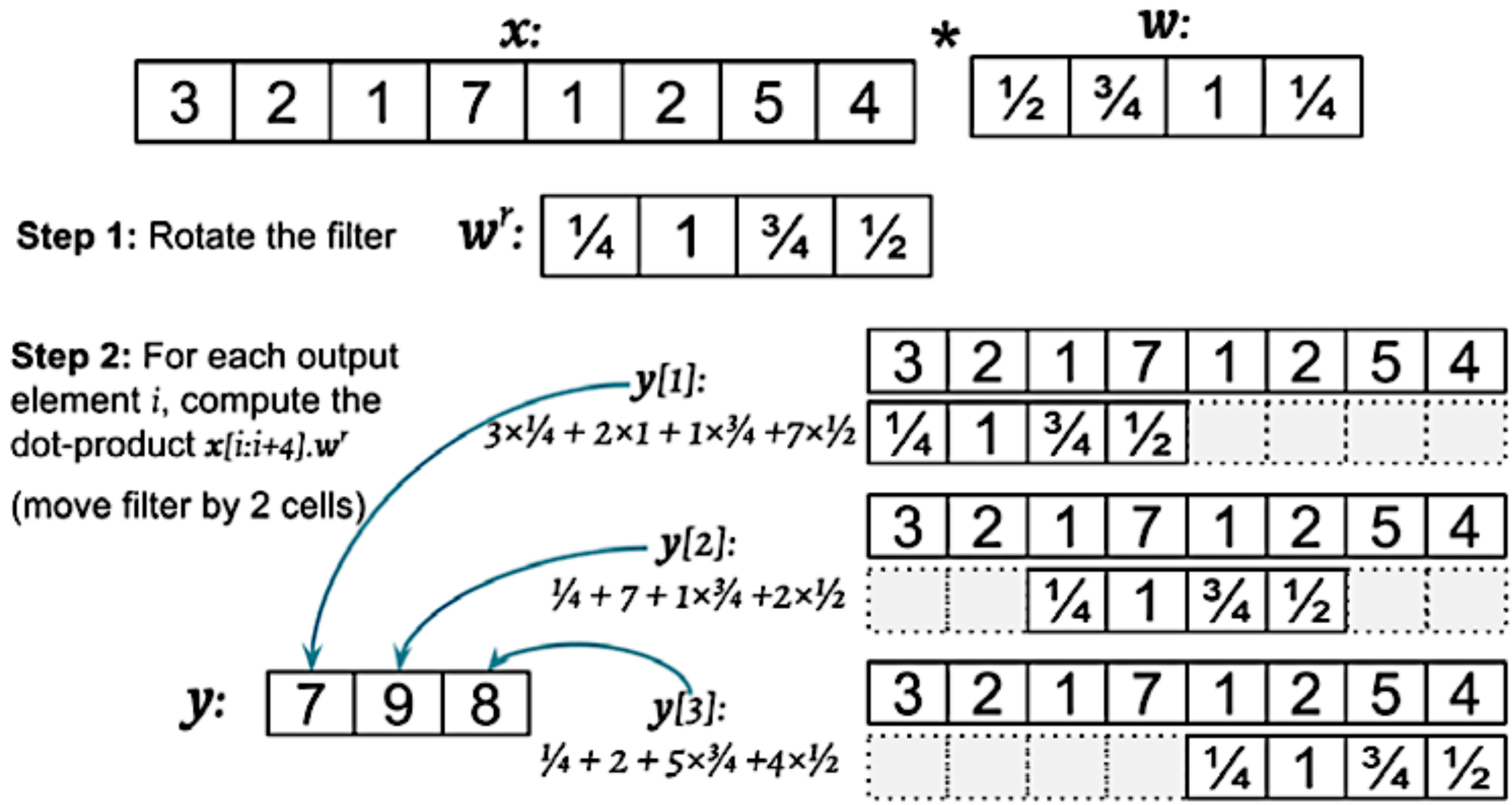
- *The padded vector* x^p *has size* $n + 2p$*, therefore:*

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \rightarrow \mathbf{y}[i] = \sum_{k=0}^{k=m-1} \mathbf{x}^p[i+m-k]\,\mathbf{w}[k]$$

- The problem of indexing is still a problem with *x* and *w* indexed in two different direction.

(cont. example):

- Flip the filter w to get the rotated filter $\boldsymbol{w}^r$
- The dot product x[i : i + m] . $\boldsymbol{w}^r$ is computed to get one element y[i]
  - x[i : i + m] is a patch of x with size m
- $x = (3,2,1,7,1,2,5,4)$
- $$\boldsymbol{w} = \left(\frac{1}{2}, \frac{3}{4}, 1, \frac{1}{4}\right)$$



x:
*
w:
3 2 1 7 1 2 5 4
½ ¾ 1 ¼
Step 1: Rotate the filter
w^r:
¼ 1 ¾ ½
Step 2: For each output element i, compute the dot-product x[i:i+4].w^r
(move filter by 2 cells)
y[1]:
3×¼ + 2×1 + 1×¾ +7×½
y[2]:
¼ + 7 + 1×¾ +2×½
y[3]:
¼ + 2 + 5×¾ +4×½
y:
7 9 8

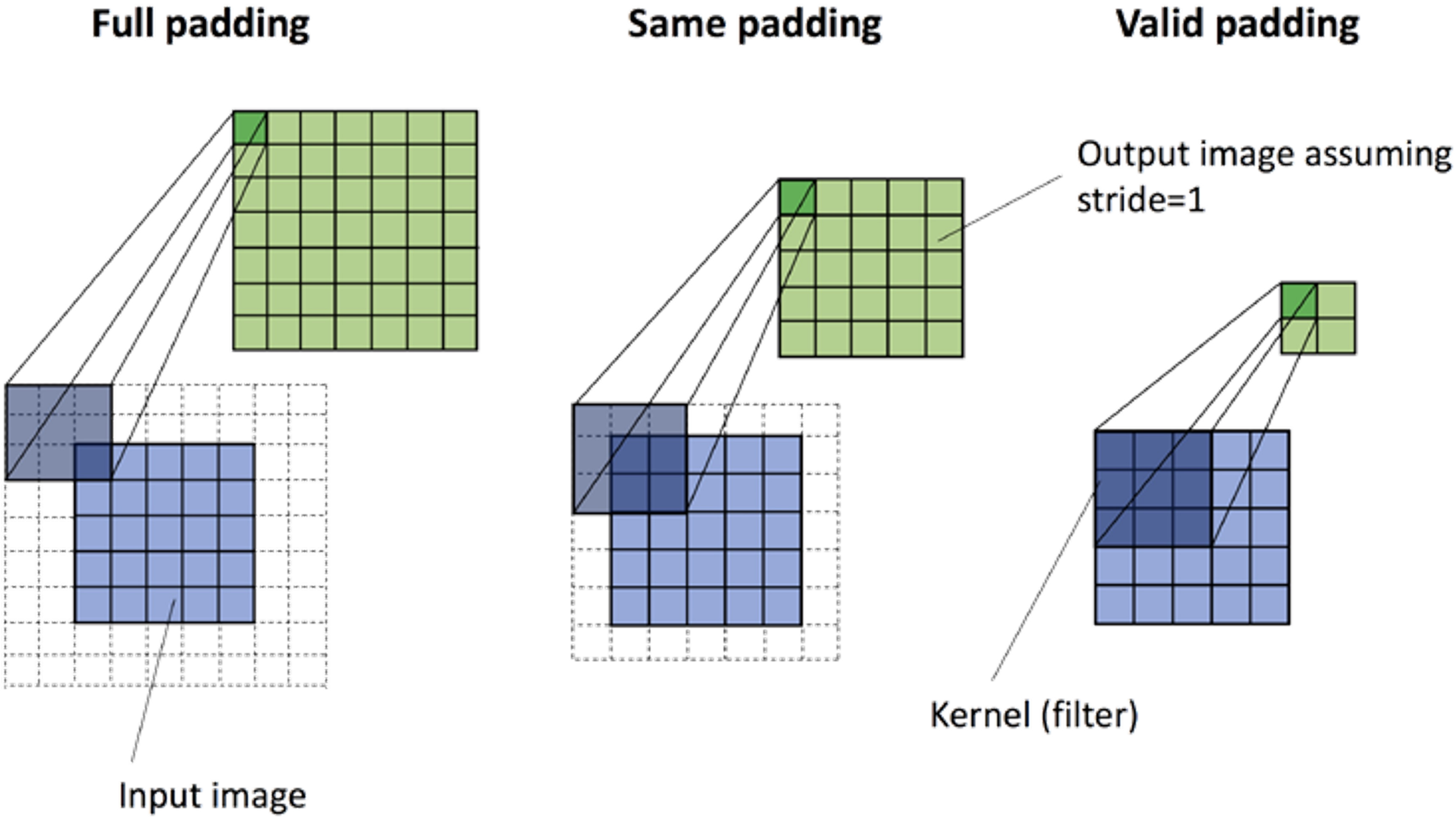
- In the last slide: the padding size is zero (***p = 0***).
- The shift is another hyperparameter of a convolution, the ***stride (s)***
- In the last slide: the stride is two, **s = 2**.
- The stride has to be a positive number smaller than the size of the input vector.

- There are 3 types of padding:

- **Full mode**: the padding parameter *p* is set to $p = m\text{-}1$. Full padding increases the dimensions of the output

- **Same** padding: the size of the output the same as the input vector *x*. In this case, the padding parameter *p* is computed according:
    - filter size, the input size, & output size are the same

- **Valid** mode: the case where $p = 0$ (no padding).



Full padding
Same padding
Valid padding
Output image assuming stride=1
Kernel (filter)
Input image

- The output size of a convolution is determined by <u>the total # of times what we shift the filter *w* along the input vector</u>

- The size of the output resulting from $x * w$ with padding p and stride s is determined:

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1$$

- The <u>*floor operation*</u> returns the largest integer that is equal or smaller to the input

- The <u>*floor operation*</u> returns the largest integer that is equal or smaller to the input

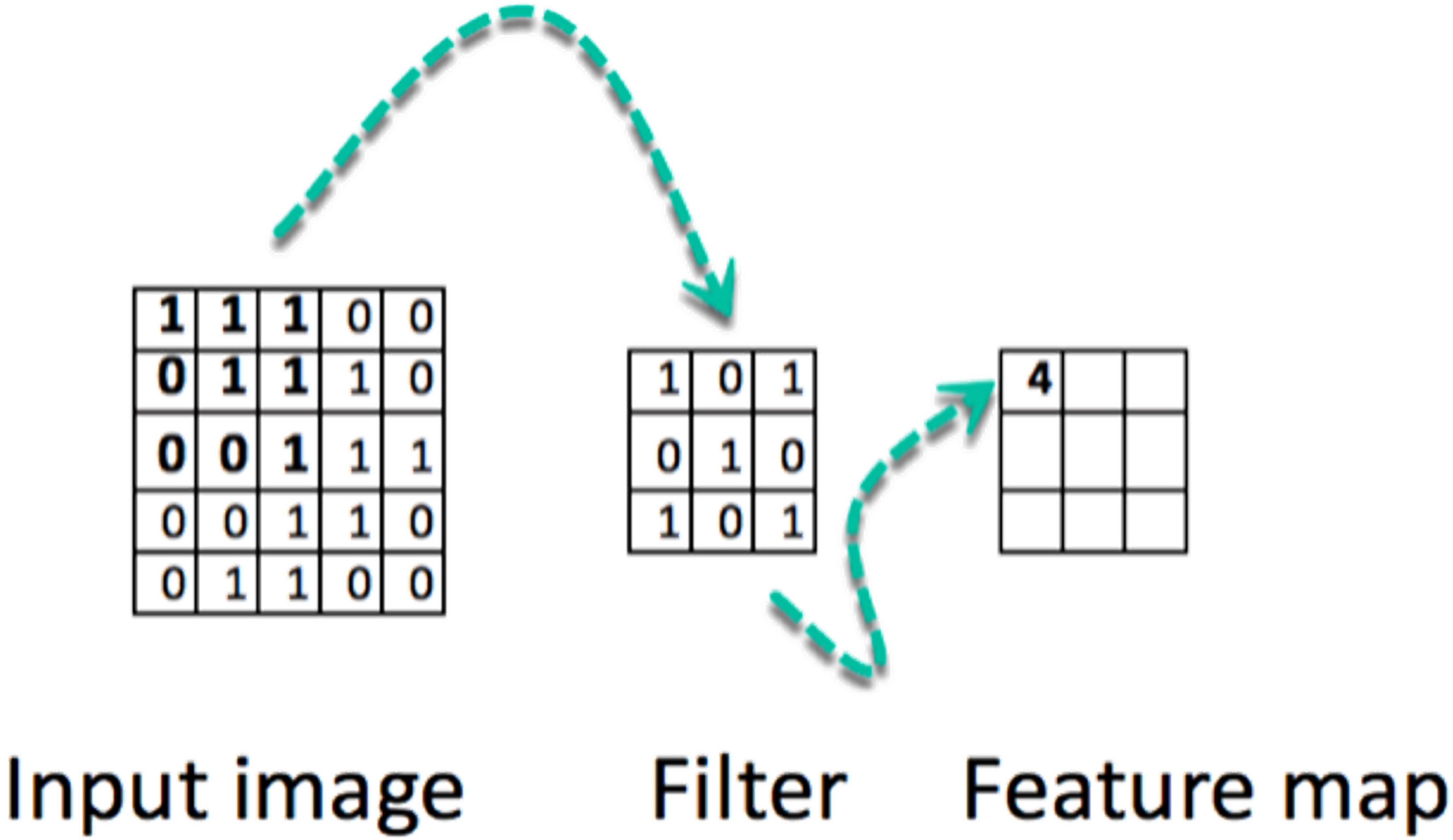
$$n = 10, m = 5, p = 2, s = 1 \rightarrow o = \left\lfloor \frac{10 + 2 \times 2 - 5}{1} \right\rfloor + 1 = 10$$

$$n = 10, m = 3, p = 2, s = 2 \rightarrow o = \left\lfloor \frac{10 + 2 \times 2 - 3}{2} \right\rfloor + 1 = 6$$

2-Dimentional Convolution

- **X** and the filter matrix $\mathbf{W}_{m1\times m2}$
  - where $\mathrm{m}_1 \leq n_1$ and $\mathrm{m}_2 \leq n_2$
  - then the matrix ***Y = X*W*** is the result of 2D convolution of ***X*** with **W**.

$$Y = X * W \rightarrow Y[i,j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i-k_1, j-k_2] W[k_1, k_2]$$


1 1 1 0 0
0 1 1 1 0
0 0 1 1 1
0 0 1 1 0
0 1 1 0 0
1 0 1
0 1 0
1 0 1
4
Input image
Filter
Feature map

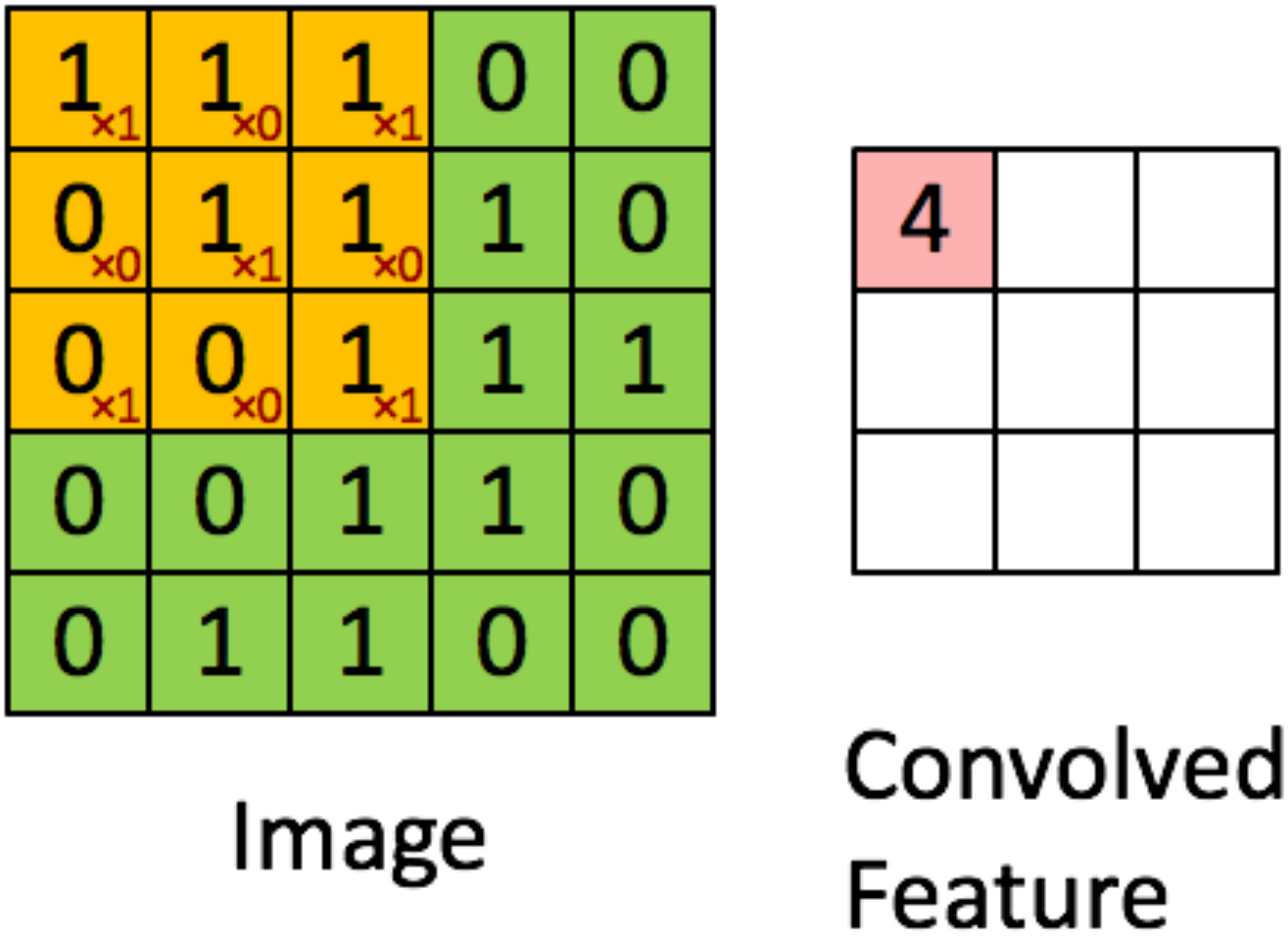
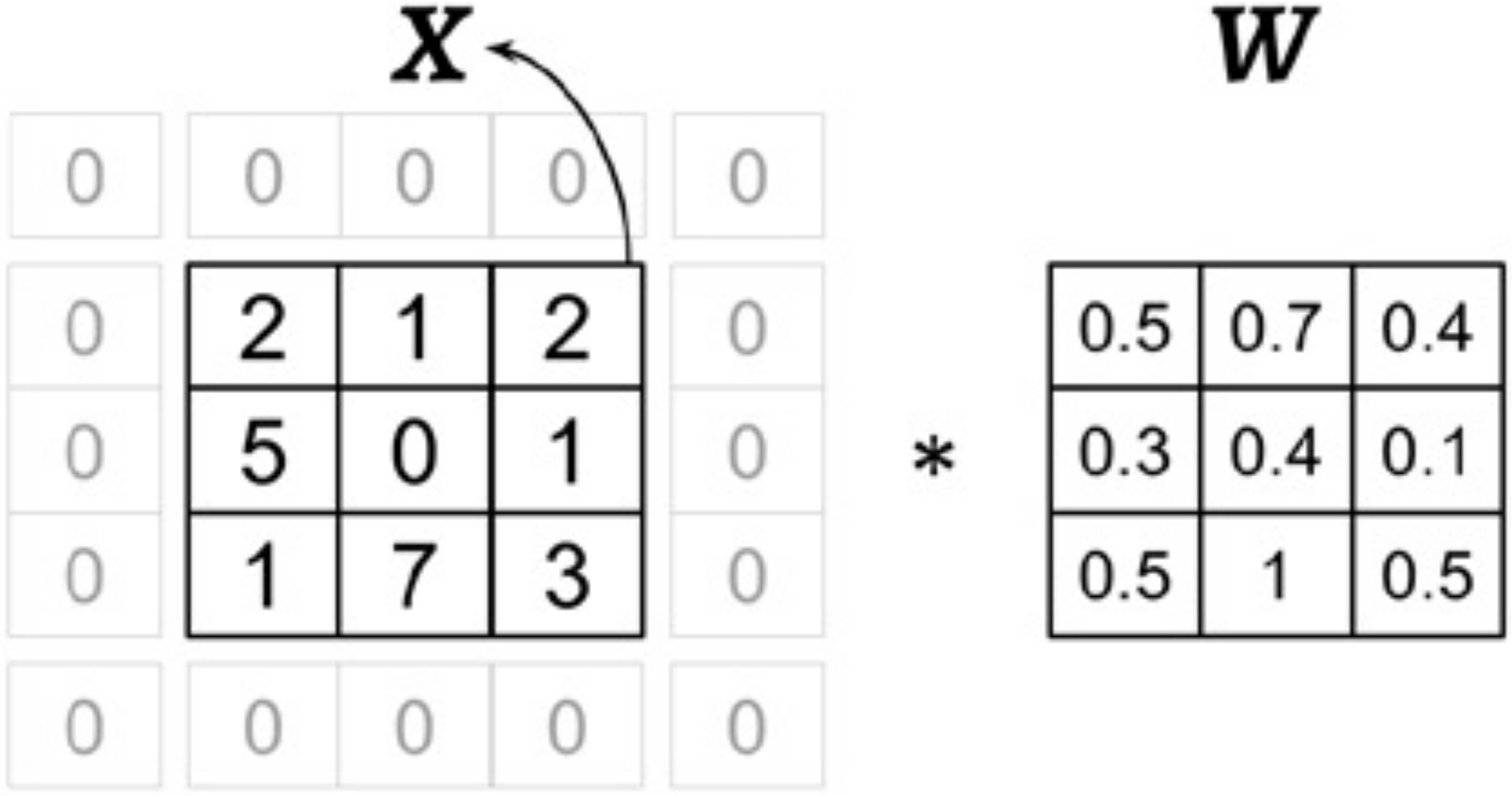


Image
4
Convolved
Feature

- Using padding @ p=(1,1)
- Stride @ s=(2,2)
- Kernel: 3x3



- Rotate the filter

- Rotation != Transpose

$\boldsymbol{W}$

0.5	0.7	0.4
0.3	0.4	0.1
0.5	1	0.5

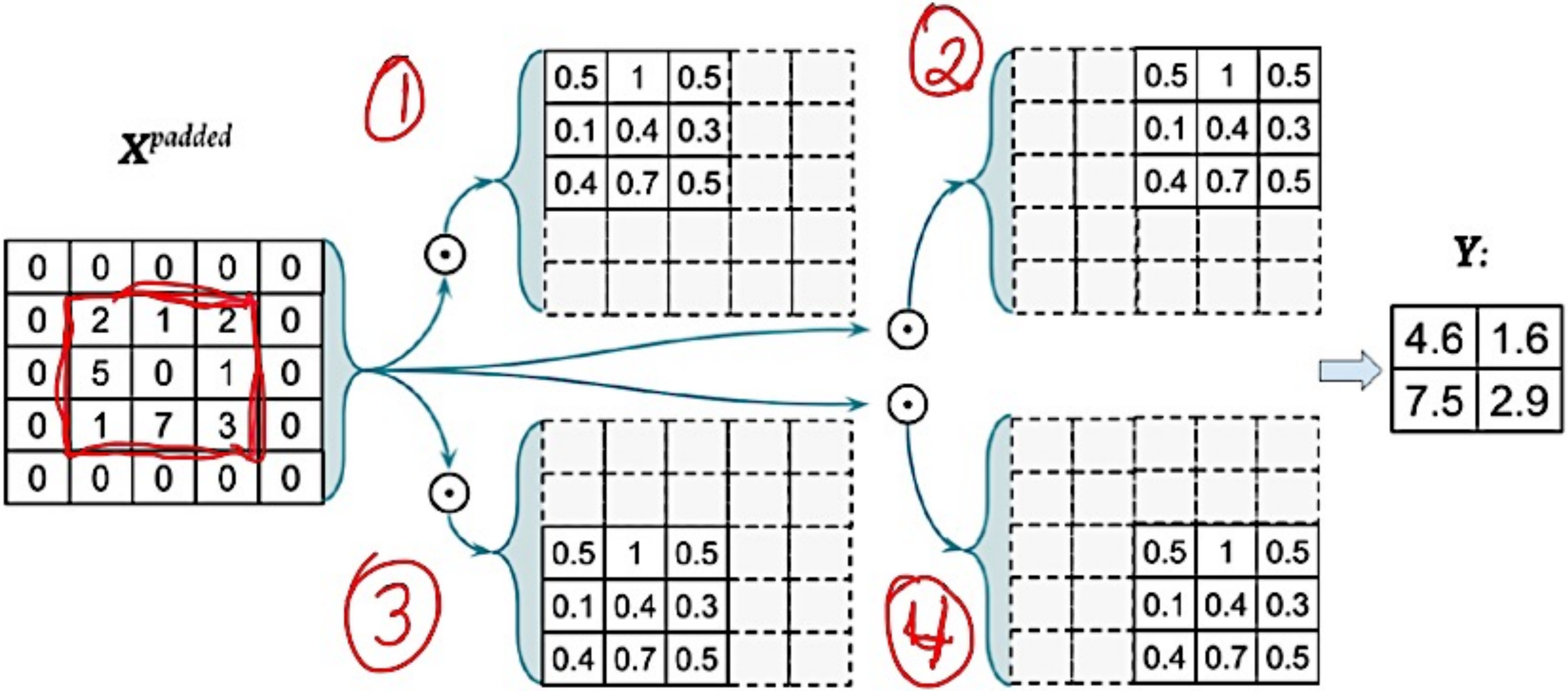
$$\boldsymbol{W}^r = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

- Shift the rotated filter matrix along the padded input matrix X_{padded} like a sliding window

- Compute the sum of the element-wise product, which is denoted by the $\odot$ operator

$$\mathbf{X}^{padded}$$

0	0	0	0	0
0	2	1	2	0
0	5	0	1	0
0	1	7	3	0
0	0	0	0	0



X^{padded}
0 0 0 0 0
0 2 1 2 0
0 5 0 1 0
0 1 7 3 0
0 0 0 0 0
1
2
3
4
0.5 1 0.5
0.1 0.4 0.3
0.4 0.7 0.5
Y:
4.6 1.6
7.5 2.9

0	0	0	0	0	0
0	105	102	100	97	96
0	103	99	103	101	102
0	101	98	104	102	100
0	99	101	106	104	99
0	104	104	104	100	98

Image Matrix

Kernel Matrix

0	-1	0
-1	5	-1
0	-1	0

320				

Output Matrix

$$\begin{aligned} &0*0+0*-1+0*0 \\ &+0*-1+105*5+102*-1 \\ &+0*0+103*-1+99*0 \quad = 320 \end{aligned}$$

Convolution with horizontal and vertical strides = 1

Subsampling

- ***Subsampling*** is typically applied in two forms of pooling operations in CNNs:
  - max-pooling
  - mean-pooling

- ***Pooling*** (max-pooling) introduces local invariance.
  - Small changes in a local neighborhood do not change the
  - Result of max-pooling.
  - It helps generate features that are more robust to noise in the input data

- Pooling decreases the size of features, which results in higher computational efficiency.

- Reducing the number of features may reduce the degree of overfitting

$$\left.\begin{array}{l} X_1 = \begin{bmatrix} 10 & 255 & 125 & 0 & 170 & 100 \\ 70 & 255 & 105 & 25 & 25 & 70 \\ 255 & 0 & 150 & 0 & 10 & 10 \\ 0 & 255 & 10 & 10 & 150 & 20 \\ 70 & 15 & 200 & 100 & 95 & 0 \\ 35 & 25 & 100 & 20 & 0 & 60 \end{bmatrix} \\ X_2 = \begin{bmatrix} 100 & 100 & 100 & 50 & 100 & 50 \\ 95 & 255 & 100 & 125 & 125 & 170 \\ 80 & 40 & 10 & 10 & 125 & 150 \\ 255 & 30 & 150 & 20 & 120 & 125 \\ 30 & 30 & 150 & 100 & 70 & 70 \\ 70 & 30 & 100 & 200 & 70 & 95 \end{bmatrix} \end{array}\right\} \xrightarrow{\text{max-pooling } P_{2\times 2}} \begin{bmatrix} 255 & 125 & 170 \\ 255 & 150 & 150 \\ 70 & 200 & 95 \end{bmatrix}$$

DCNN

- Convolutional layer may contain one or more 2D arrays or matrices with dimensions $N_1 \times N_2$
- These $N_1 \times N_2$ matrices are called channels.
- Using multiple channels as input to a convolutional layer requires us to use a 3D-array:

$$\mathbf{X}_{\mathbf{N1 \times N2 \times C}_{\boldsymbol{in}}}$$

, where C_{in} is the number of input channels

- The overall process:

Given a sample $\mathbf{X}_{n_1 \times n_2 \times c_{in}}$,

a kernel matrix $\mathbf{W}_{m_1 \times m_2 \times c_{in}}$,

and bias value b

$$\Rightarrow \begin{cases} Y^{Conv} = \sum_{c=1}^{C_{in}} W[:,:,c] * X[:,:,c] \\ \text{pre-activation:} \quad \mathbf{A} = \mathbf{Y}^{\text{Conv}} + \text{b} \\ \text{Feature map:} \quad H = \phi(A) \end{cases}$$

- For the RGB images: **3 channels**

- Therefore, $C_{in} = 3$. However, each channels produces a different feature map

- The overall process:

Given a sample $\boldsymbol{X}_{n_1 \times n_2 \times C_{in}}$

kernel matrix $\boldsymbol{W}_{m_1 \times m_2 \times C_{in} \times C_{out}}$

and bias vector $\boldsymbol{b}_{C_{out}}$

$$\Rightarrow \begin{cases} \boldsymbol{Y}^{Conv}[:,:,k] = \sum_{c=1}^{C_{in}} \boldsymbol{W}[:,:,c,k] * \boldsymbol{X}[:,:,c] \\ \boldsymbol{A}[:,:,k] = \boldsymbol{Y}^{Conv}[:,:,k] + \boldsymbol{b}[k] \\ \boldsymbol{H}[:,:,k] = \phi\left(\boldsymbol{A}[:,:,k]\right) \end{cases}$$

Dropout Layers

- Choosing the size of a network, whether we are dealing with a dense NN or a CNN, has always been a challenging problem.

- The size of a weight matrix and the number of layers need to be tuned to achieve a reasonably good performance.

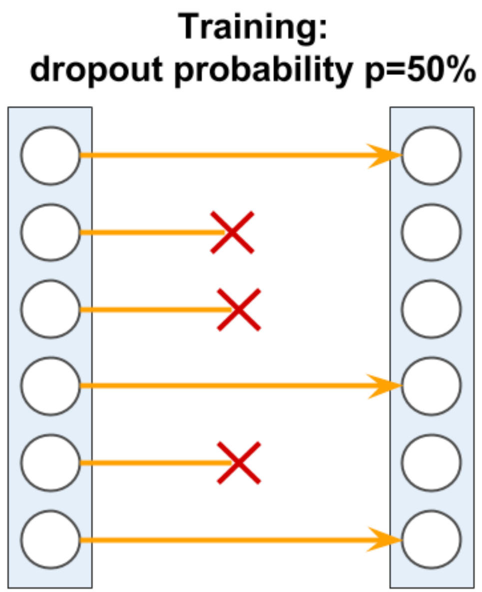
  - **Small networks** with a relatively small number of parameters → have a low capacity and are therefore likely to be under fit, resulting in poor performance.

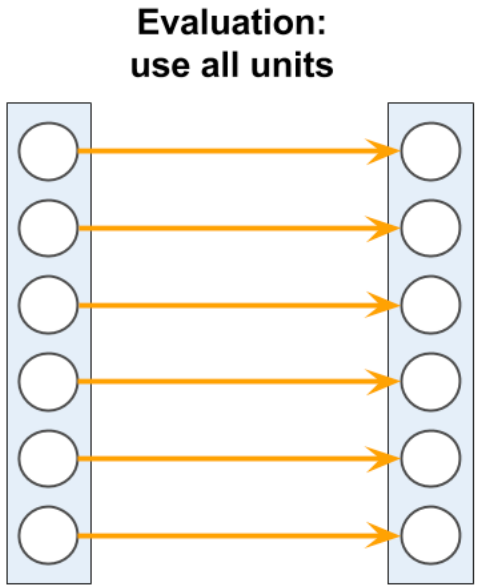
  - **Large networks** may more easily result in overfitting → where the network will memorize the training data and do extremely well on the training set

- Since they cannot learn the underlying structure of complex datasets.
- Yet, very large networks may more easily result in overfitting
  - Where the network will memorize the training data and
  - Do extremely well on the training set while achieving poor performance on the held-out test set

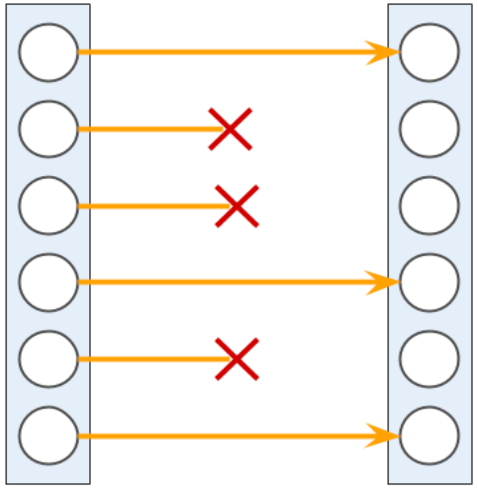
- Dropout has emerged that works amazingly well <u>for regularizing</u> (deep) neural networks

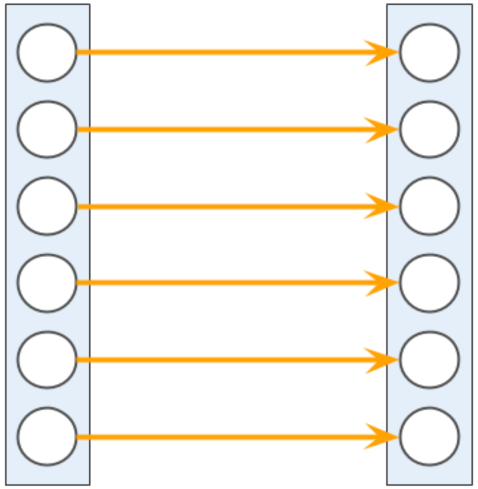
- Dropout offers a workaround with an efficient way to train many models at once and compute their average predictions at test or prediction time.





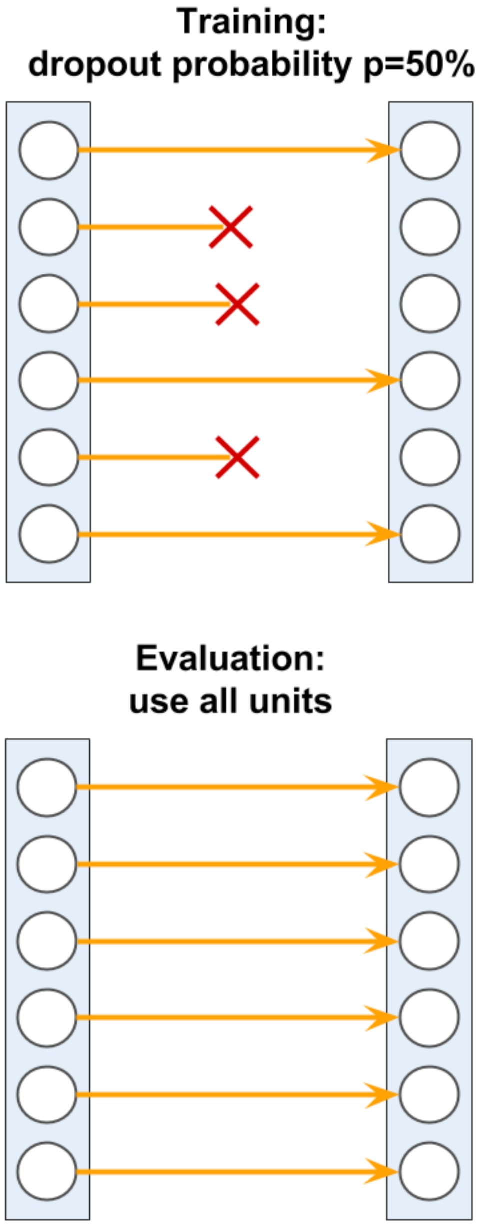
- Dropout is usually applied to the hidden units of higher layers.
- **During training** → a fraction of the hidden units is randomly dropped at every iteration with probability p_{drop}
- The probability is determined by the user and the common choice is $p_{drop}=0.5$





- The randomness forces the network to learn a redundant representation of the data.

- The network cannot rely on an activation of any set of hidden units since they may be turned off at any time during training
  - Forced to learn more general and robust patterns from the data



- During prediction, all neurons will contribute to computing the pre-activations of the next layer