50.039 Theory and Practice of Deep Learning

W4-S1 Introduction to Computer Vision and Convolutional Neural Networks

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Restricted

About this week (Week 4)

1. What are **images** and how is this datatype represented?
2. What is a **pixel** and how can **its information be interpreted**?
3. What is the **spatial dependence** property of pixels?
4. What is the **homophily property** of pixels?
5. Why is the **linear processing operation failing** on images?
6. What is the **convolution** operation?
7. How can we perform **image processing using convolutions**?
8. What is **padding** in convolutions?

About this week (Week 4)

9. What is **stride** in convolution?
10. What is **dilation** in convolution?
11. How does convolution apply to **higher dimensional images**?
12. How to write **our own convolutional processing layer** in PyTorch?
13. What is **Conv2d** in PyTorch?
14. What is a **Convolutional Neural Network** (CNN)?
15. What is the **intuition** behind using Convolutional layers in a Convolutional Neural Network?

About this week (Week 4)

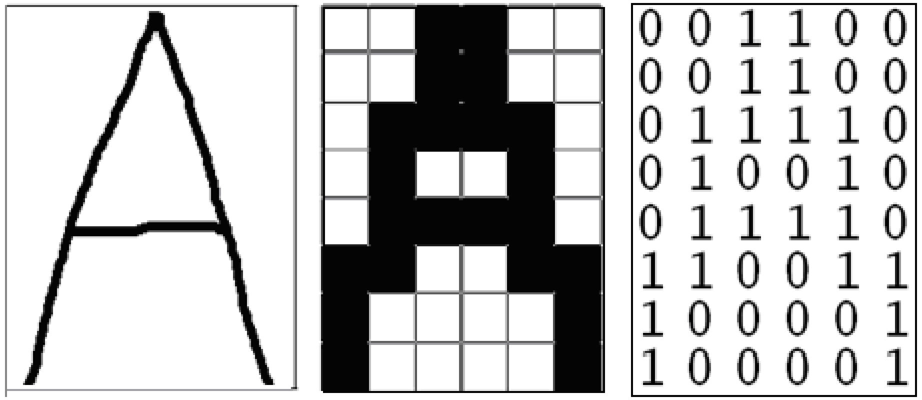
16. How to **train** our first Convolutional Neural Network on MNIST?
17. What are **other typical image processing layers** in Computer Vision and how are they implemented?
18. What is the **pooling layer**?
19. What is the **batchnorm layer**?
20. What is the **dropout layer**?
21. What is **data augmentation** in Computer Vision?
22. What are some **milestone Computer Vision models** and their **contributions** to the field of Deep Learning?

About this week (Week 4)

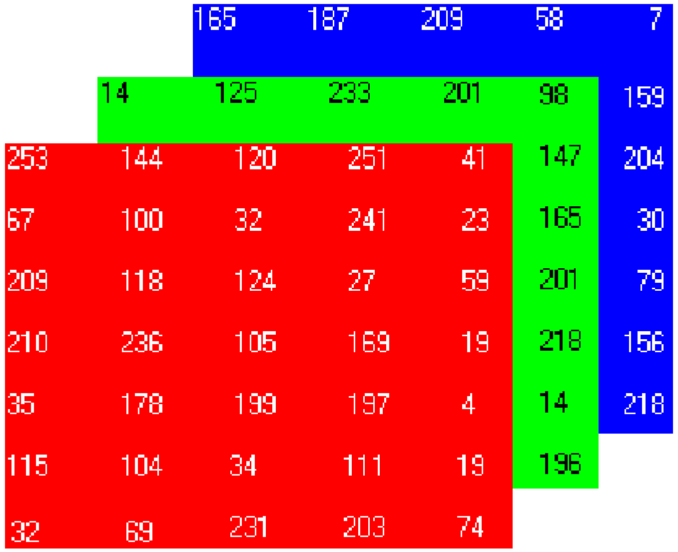
23. What are the **AlexNet** and **VGG models**?
24. What is a **skip connection/residual**? What is its effect on a Neural Network and **how does it help with vanishing gradient problems**?
25. What are **ResNet** and **DenseNet models**?
26. What is the **Inception model**?
27. What is the **EfficientNet model**?
28. What is **transfer learning** and its uses?
29. How to **freeze** and **fine-tune layers** in a Neural Network?

How to encode an image

- Both **grayscale** and **RGB** (Red Green Blue) **images** can be represented as **matrices** or **tensors**.



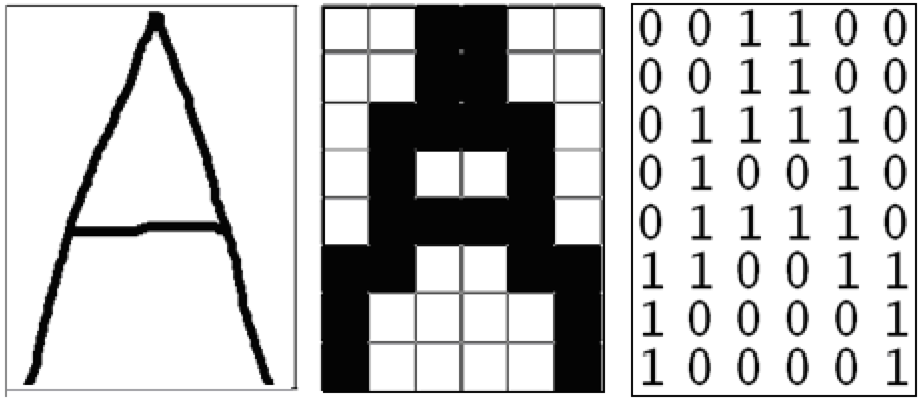
- Typically, our sample images in the MNIST dataset were 2D matrices of shape 28×28.

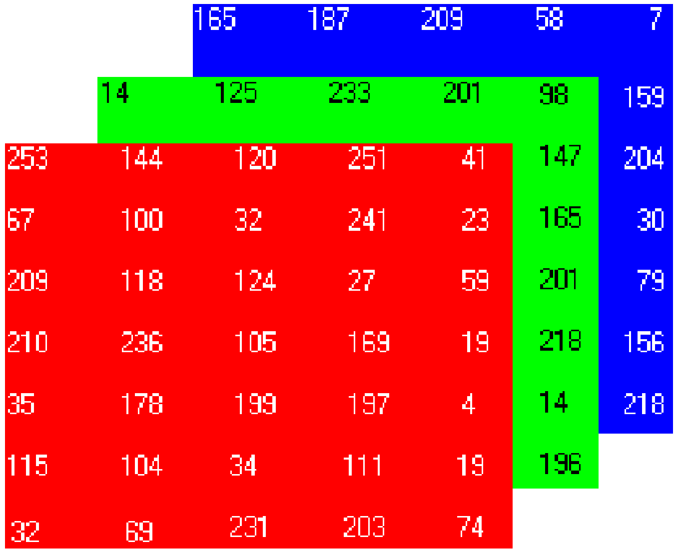


How to encode an image

Definition (encoding a grayscale image):

- **Grayscale images** can be represented as a **2D matrix** of shape $height \times\ width$, where each element is an 8-bit integer.
- Each **element** of the matrix represents the **intensity of light** at that pixel.
- The intensity of light can be represented as a single number between 0 and 255, where **0** represents no light (**black**) and **255** represents maximum intensity (**white**).

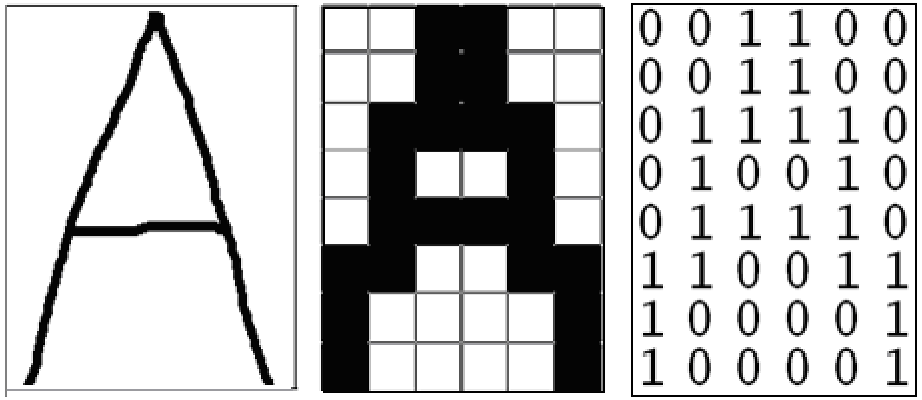


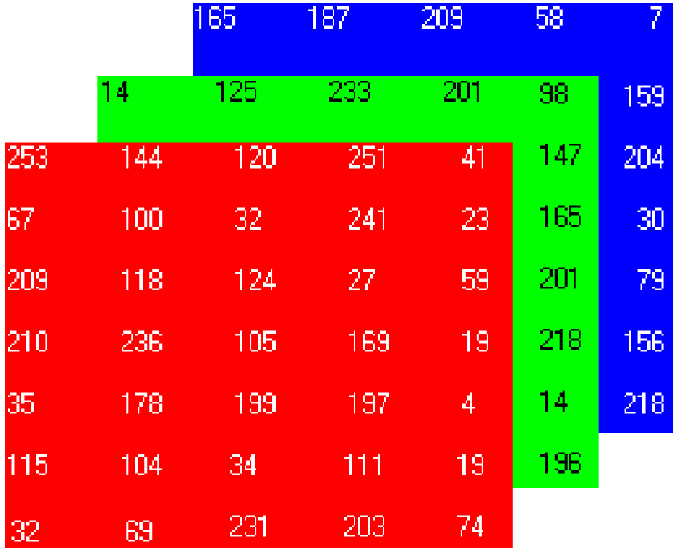


How to encode an image

Definition (encoding a RGB image):

- **RGB images** are more complex because each pixel has three values representing the intensity of light for each color **channel** (red, green, and blue).
- An RGB image is represented as a **3D tensor** of shape $height \times width \times 3$.
- Each element in the tensor would be an 8-bit integer representing the intensity of the corresponding color channel.

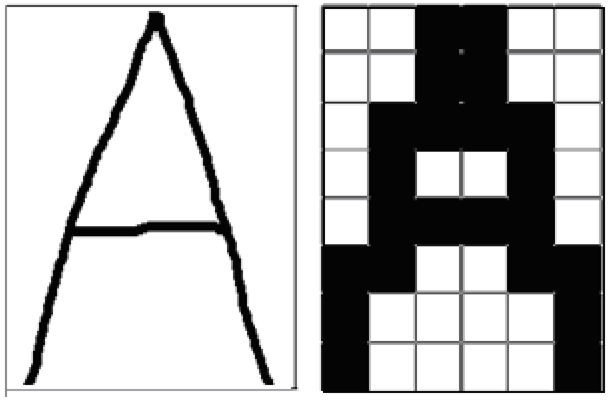




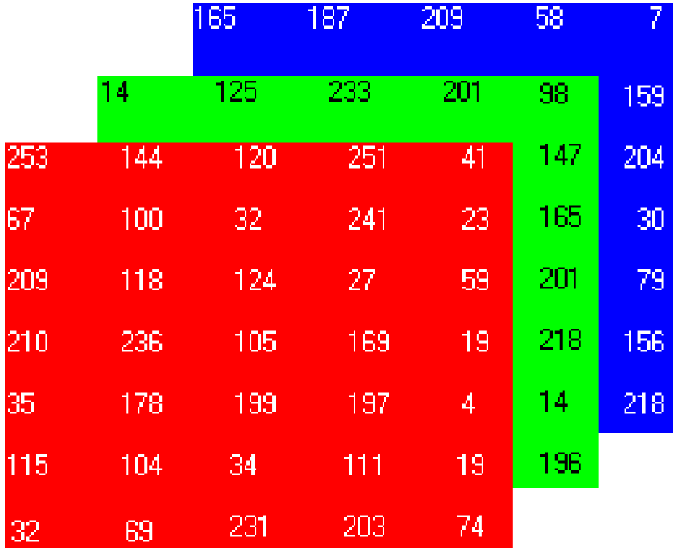
How to encode an image

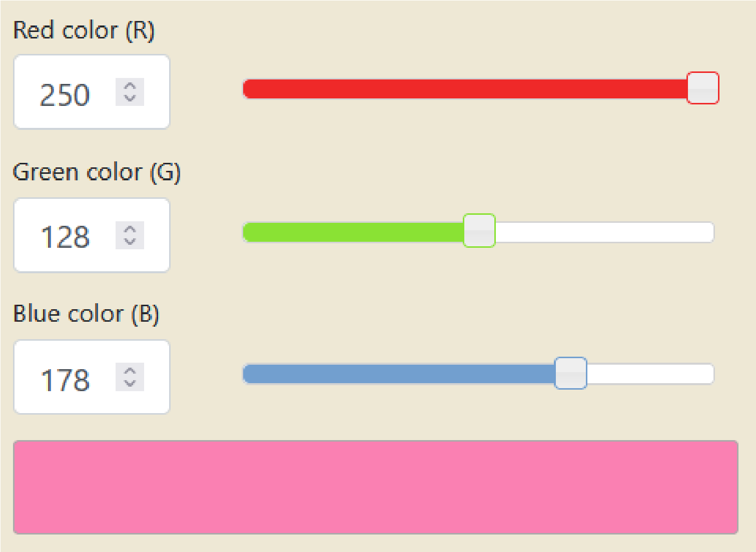
- Combining the red, green and blue values allows to generate any color. Try it!

https://www.rapidtables.com/convert/color/rgb-to-hex.html



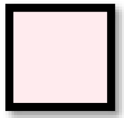
0	0	1	1	0	0
0	0	1	1	0	0
0	1	1	1	1	0
0	1	0	0	1	0
0	1	1	1	1	0
1	1	0	0	1	1
1	0	0	0	0	1
1	0	0	0	0	1





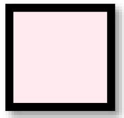
Pixels information

- Each element in a Greyscale matrix or RGB matrix corresponds to a pixel, representing something in the image.
- The problem, however, is that **it is often impossible to guess the meaning of a given pixel, by looking at a pixel alone**.



Pixels information

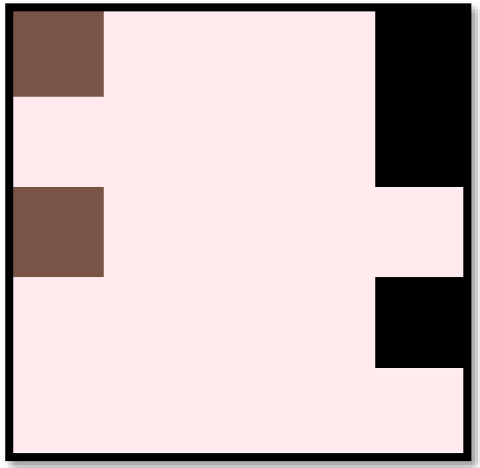
→ **Can you guess what this pixel represents and what object it belongs to?**



1. Flower petal
2. Piece of clothing
3. Skin of a human person
4. Feather of a bird

Pixels information

→ **What if I show you <u>some</u> neighbouring pixels now?**

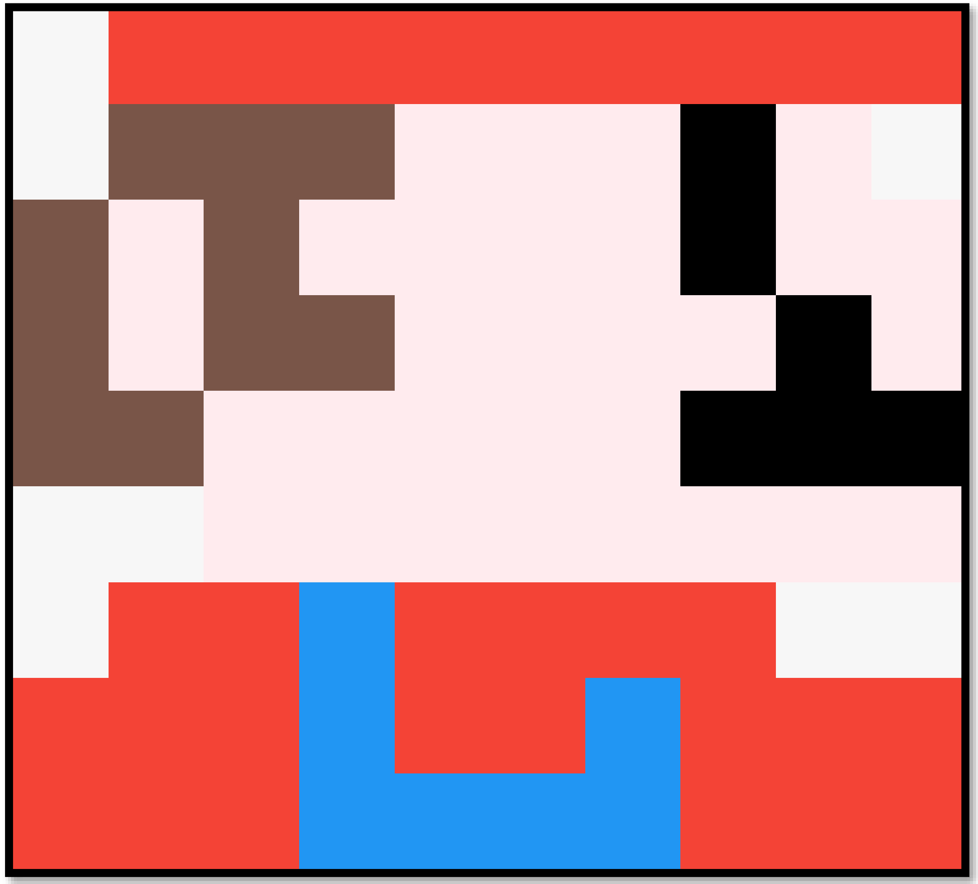


1. Flower petal
2. Piece of clothing
3. Skin of a human person
4. Feather of a bird

Pixels information

→ **What if I show you MORE neighbouring pixels now?**

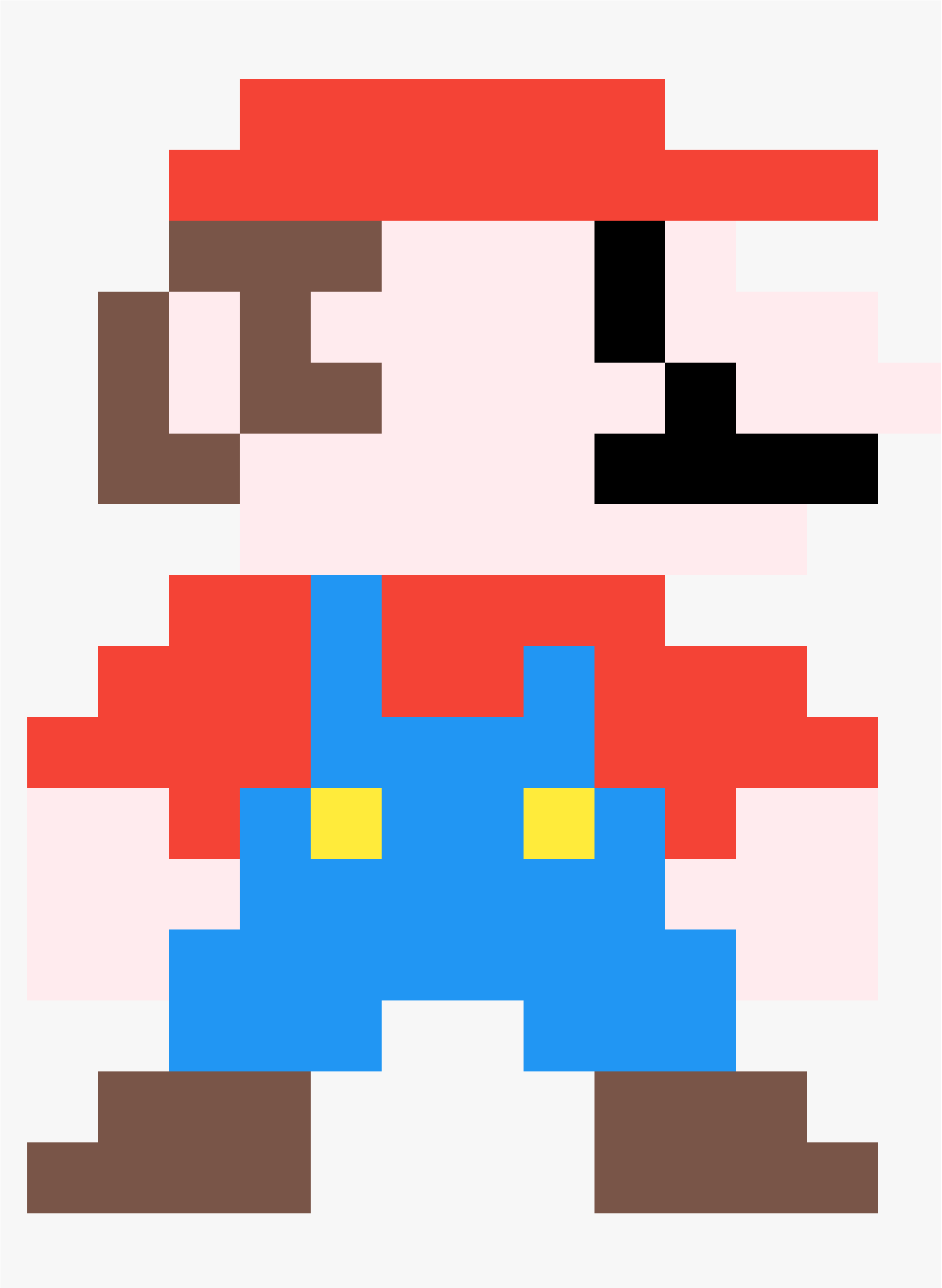
1. Flower petal
2. Piece of clothing
3. Skin of a human person
4. Feather of a bird



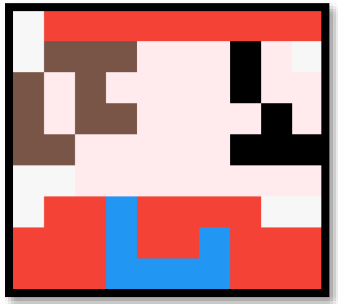
Pixels information

→ **What if I show you <u>MORE</u> neighbouring pixels now?**

1. Flower petal
2. Piece of clothing
3. **Skin of a human person**
4. Feather of a bird



The spatial dependence property



Definition (the spatial dependence property of pixels):

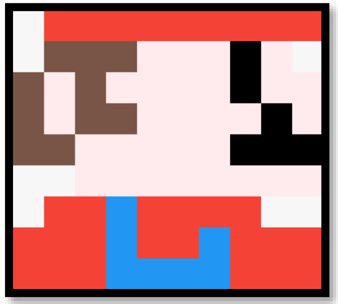
The **spatial dependence property of pixels** states that **the meaning or interpretation of a single pixel in an image can often only be understood in the context of the pixels around it**.

This is because the information in a single pixel is often not enough to describe the complete picture, or even just the pixel itself.

The relationship between the given pixel and its neighbouring pixels is necessary to provide a full understanding of this particular pixel, or even the image as a whole.

This is something we observed in the Mario pixel art example.

The homophily property



Definition (the homophily property of pixels):

Homophily in images refers to the phenomenon where **pixels that are close to each other in space tend to have similar properties**, such as color, texture, or intensity. In other words, is the **tendency for similar pixels to be grouped together**.

For example, in an image of a sky, pixels that are close to each other and have a similar blue color can often be grouped together and be considered part of the sky region.

This property is a **key factor in image processing** and **computer vision** algorithms, as it provides a way to segment images into meaningful regions and perform tasks like object recognition and image classification.

The problem with Linear processing

So far we have only used Linear operations in our neural networks, even when playing with images, as in the MNIST dataset.

- We would flatten the image, assembling all pixels in a 1D vector, essentially **losing the spatial information of the image**.
- Then, Linear processing layers would simply **process pixels independently**, applying individual coefficients to each pixel separately.

$$Z = \sum_i \sum_j w_{i,j} x_{i,j} + b$$

With $x_{i,j}$ the pixel value at location (i, j), $w_{i,j}$ the weight coefficient for the Linear layer assigned to this pixel, and b the bias.

The problem with Linear processing

Linear layers are **not great at processing images** because they do not take into account the **spatial dependence of pixels.**

- Pixels in images are typically related to each other in a spatial manner, i.e. the meaning of a pixel depends on the neighbour pixels.
- Unfortunately, linear models would assume each pixel is independent of the others and process them independently, which is not correct.
- As a result, linear layers can miss important spatial relationships between pixels. This can, in turn, result in poor image representation and interpretation.

→ Need for a more specific processing operation for images, to use in place of the Linear one!

Introducing convolution!

Definition (convolution):

Convolution is a **mathematical operation** that is widely used in image processing, computer vision, and deep learning.

It involves applying a small matrix called a **convolution kernel** or **filter** K, over a given image X, element-wise multiplying each overlapping set of values in the image with the kernel, and then summing the results.

The result of this operation is a new image Y, where each output pixel represents the sum of the product of the corresponding input pixel and the filter.

$$Y = f(X, K)$$

Introducing convolution!

Let us consider

- A grayscale image, defined as a 2D matrix X of size $h \times w$, with pixel values $\left(X_{i,j}\right) \in [0, 255]$.
- A convolution kernel K of size $k \times k$, with values $\left(K_{i,j}\right) \in \mathbb{R}$.

First, let us discuss sizes…

The convolution operation defined as $Y = f(X, K)$, produces an image Y of size $h' \times w'$,

- With $h' = h - k + 1$,
- And $w' = w - k + 1$.

Note: we often use a kernel defined as a square matrix, with odd sizes (i.e. 3×3, 5×5, etc.). The operation, however, remains valid for other kernel sizes (but some notations showed in next slide would have to be adjusted).

Introducing convolution!

Second, the pixel values $y_{i,j}$ for image Y are calculated using the convolution operation, defining $Y_{i,j}$, $\forall i \in [1, h'], j \in [1, w']$ as:

$$Y_{i,j} = \frac{1}{k^2} \sum_{m=1}^{k} \sum_{n=1}^{k} X_{i+m-1,\, j+n-1} \, K_{m,n}$$

This operation might seem complicated, but is easily visualised!

X

1	2	4	2	1
4	7	3	2	1
4	5	2	3	1
2	1	7	8	4
3	2	4	7	8

X is 5 by 5

K is 3 by 3

Y is therefore 3 by 3.

(5-3+1 = 3)

K

1	0	1
0	2	1
1	0	1

Y

...	...	...
...	...	...
...	...	...

Introducing convolution!

Second, the pixel values $y_{i,j}$ for image Y are calculated using the convolution operation, defining $Y_{i,j}$, $\forall i \in [1, h'], j \in [1, w']$ as:

$$Y_{i,j} = \frac{1}{k^2} \sum_{m=1}^{k} \sum_{n=1}^{k} X_{i+m-1,\, j+n-1}\, K_{m,n}$$

This operation might seem complicated, but is easily visualised!

$$X = \begin{bmatrix} 1 & 2 & 4 & 2 & 1 \\ 4 & 7 & 3 & 2 & 1 \\ 4 & 5 & 2 & 3 & 1 \\ 2 & 1 & 7 & 8 & 4 \\ 3 & 2 & 4 & 7 & 8 \end{bmatrix}$$

$$\begin{aligned} & 1 \times 1 \\ & +2 \times 0 \\ & +4 \times 1 \\ & +4 \times 0 \\ & +7 \times 2 \\ & +3 \times 1 \\ & +4 \times 1 \\ & +5 \times 0 \\ & +2 \times 1 \\ & = 28 \end{aligned}$$

$$K = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

$$Y = \begin{bmatrix} \frac{28}{9} & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix}$$

Introducing convolution!

Second, the pixel values $y_{i,j}$ for image Y are calculated using the convolution operation, defining $Y_{i,j}$, $\forall i \in [1, h'], j \in [1, w']$ as:

$$Y_{i,j} = \frac{1}{k^2} \sum_{m=1}^{k} \sum_{n=1}^{k} X_{i+m-1,\ j+n-1}\ K_{m,n}$$

This operation might seem complicated, but is easily visualised!

X

1	2	4	2	1
4	7	3	2	1
4	5	2	3	1
2	1	7	8	4
3	2	4	7	8

K

1	0	1
0	2	1
1	0	1

Y

$\frac{28}{9}$	…	…
…	…	…
…	$\frac{39}{9}$	…

$$\begin{aligned} & 5 \times 1 \\ & +2 \times 0 \\ & +3 \times 1 \\ & +1 \times 0 \\ & +7 \times 2 \\ & +8 \times 1 \\ & +2 \times 1 \\ & +4 \times 0 \\ & +7 \times 1 \\ & = 39 \end{aligned}$$

Introducing convolution!

Second, the pixel values $y_{i,j}$ for image Y are calculated using the convolution operation, defining $Y_{i,j}$, $\forall i \in [1, h'], j \in [1, w']$ as:

$$Y_{i,j} = \cancel{\frac{1}{k^2}} \sum_{m=1}^{k} \sum_{n=1}^{k} X_{i+m-1,\, j+n-1}\, K_{m,n}$$

Note: Sometimes, the **normalization term** is removed and we may instead **normalize the kernel** (ensuring elements in kernel matrix sum up to 1), or not.

$$X = \begin{bmatrix} 1 & 2 & 4 & 2 & 1 \\ 4 & 7 & 3 & 2 & 1 \\ 4 & 5 & 2 & 3 & 1 \\ 2 & 1 & 7 & 8 & 4 \\ 3 & 2 & 4 & 7 & 8 \end{bmatrix}$$

$$K = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

$$Y = \begin{bmatrix} \frac{28}{9} & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ \cdots & \frac{39}{9} & \cdots \end{bmatrix}$$

$$\begin{aligned} & 5 \times 1 \\ & +2 \times 0 \\ & +3 \times 1 \\ & +1 \times 0 \\ & +7 \times 2 \\ & +8 \times 1 \\ & +2 \times 1 \\ & +4 \times 0 \\ & +7 \times 1 \\ & = 39 \end{aligned}$$

Introducing convolution!

Second, the pixel values $y_{i,j}$ for image Y are calculated using the convolution operation, defining $Y_{i,j}$, $\forall i \in [1, h'], j \in [1, w']$ as:

$$Y_{i,j} = \cancel{\frac{1}{k^2}} \sum_{m=1}^{k} \sum_{n=1}^{k} X_{i+m-1,\, j+n-1} K_{m,n}$$

→ **At the end of the day, convolution is just another type of matrix multiplication!**

$$X = \begin{bmatrix} 1 & 2 & 4 & 2 & 1 \\ 4 & 7 & 3 & 2 & 1 \\ 4 & 5 & 2 & 3 & 1 \\ 2 & 1 & 7 & 8 & 4 \\ 3 & 2 & 4 & 7 & 8 \end{bmatrix}$$

$$5 \times 1 + 2 \times 0 + 3 \times 1 + 1 \times 0 + 7 \times 2 + 8 \times 1 + 2 \times 1 + 4 \times 0 + 7 \times 1 = 39$$

$$K = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

$$Y = \begin{bmatrix} \frac{28}{9} & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \frac{39}{9} & \dots \end{bmatrix}$$

Convolution and Image Processing courses

- **Convolution** is a **key mathematical operation** that is widely used in computer vision and image processing to modify the apply a wide range of **transformations on an image** (enhancing contrast, blurring, edge detection, Instagram filters, etc).
- If **Computer Vision** is a direction that interests you, I would strongly advise to take a course on Image Processing (with implementations in OpenCV, Photoshop, etc.) during your continued learning.
- For instance: https://www.udemy.com/course/digital-image-processing-from-ground-up-in-python/

Testing convolution on images

Let us try the convolution operation on images to see their effect.

- First, we need a **greyscale image**.

```python
# Open the image and convert it to grayscale
im = Image.open('flower.jpg').convert('L')

# Convert the image to a NumPy array
im_array = np.array(im)

# Print the shape of the array
print(im_array.shape)
```

```
(459, 612)
```

```python
# Display image in matplotlib
plt.imshow(im_array, cmap = 'gray')
plt.show()
```

Testing convolution on images

Let us try the convolution operation on images to see their effect.

- First, we need a greyscale image.
- Second, we need a **convolution function**.

```
# Our convolution function
def convolution(image, kernel):
    # Flip the kernel (optional)
    kernel = np.flipud(np.fliplr(kernel))

    # Get the dimensions of the image and kernel
    image_rows, image_cols = image.shape
    kernel_rows, kernel_cols = kernel.shape

    # Convolve using Numpy
    output = correlate(image, kernel, mode = 'valid')

    # Note that this is equivalent to this
    '''
    # Loop through the image, applying the convolution
    output = np.zeros_like(image)
    for x in range(image_rows - kernel_rows + 1):
        for y in range(image_cols - kernel_cols + 1):
            output[x, y] = (kernel * image[x:x+kernel_rows, y:y+kernel_cols]).sum()
    '''
    return output
```

Testing convolution on images

What is the purpose of flipping the kernel?

Optional, but in short, flipping the kernel has mathematical and practical advantages that make it a common practice in deep learning

(It has to do with signal processing, Fast Fourier Transform and the convolution theorem, out-of-scope).

```python
# Our convolution function
def convolution(image, kernel):
    # Flip the kernel (optional)
    kernel = np.flipud(np.fliplr(kernel))

    # Get the dimensions of the image and kernel
    image_rows, image_cols = image.shape
    kernel_rows, kernel_cols = kernel.shape

    # Convolve using Numpy
    output = correlate(image, kernel, mode = 'valid')

    # Note that this is equivalent to this
    '''
    # Loop through the image, applying the convolution
    output = np.zeros_like(image)
    for x in range(image_rows - kernel_rows + 1):
        for y in range(image_cols - kernel_cols + 1):
            output[x, y] = (kernel * image[x:x+kernel_rows, y:y+kernel_cols]).sum()
    '''
    return output
```

https://stackoverflow.com/questions/45152473/why-is-the-convolutional-filter-flipped-in-convolutional-neural-networks

Testing convolution on images

Let us try the convolution operation on images to see their effect.

- First, we need a greyscale image.
- Second, we need a **convolution function**.
- The magic for the transformation then happens when deciding on **which kernel to use**.

```python
# Our convolution function
def convolution(image, kernel):
    # Flip the kernel (optional)
    kernel = np.flipud(np.fliplr(kernel))

    # Get the dimensions of the image and kernel
    image_rows, image_cols = image.shape
    kernel_rows, kernel_cols = kernel.shape

    # Convolve using Numpy
    output = correlate(image, kernel, mode = 'valid')

    # Note that this is equivalent to this
    '''
    # Loop through the image, applying the convolution
    output = np.zeros_like(image)
    for x in range(image_rows - kernel_rows + 1):
        for y in range(image_cols - kernel_cols + 1):
            output[x, y] = (kernel * image[x:x+kernel_rows, y:y+kernel_cols]).sum()
    '''
    return output
```

Testing convolution on images

Definition (the blur kernel):

The **blur kernel** is a matrix with constant values, summing up to 1.

For instance, a 5×5 **blur kernel** is defined as:

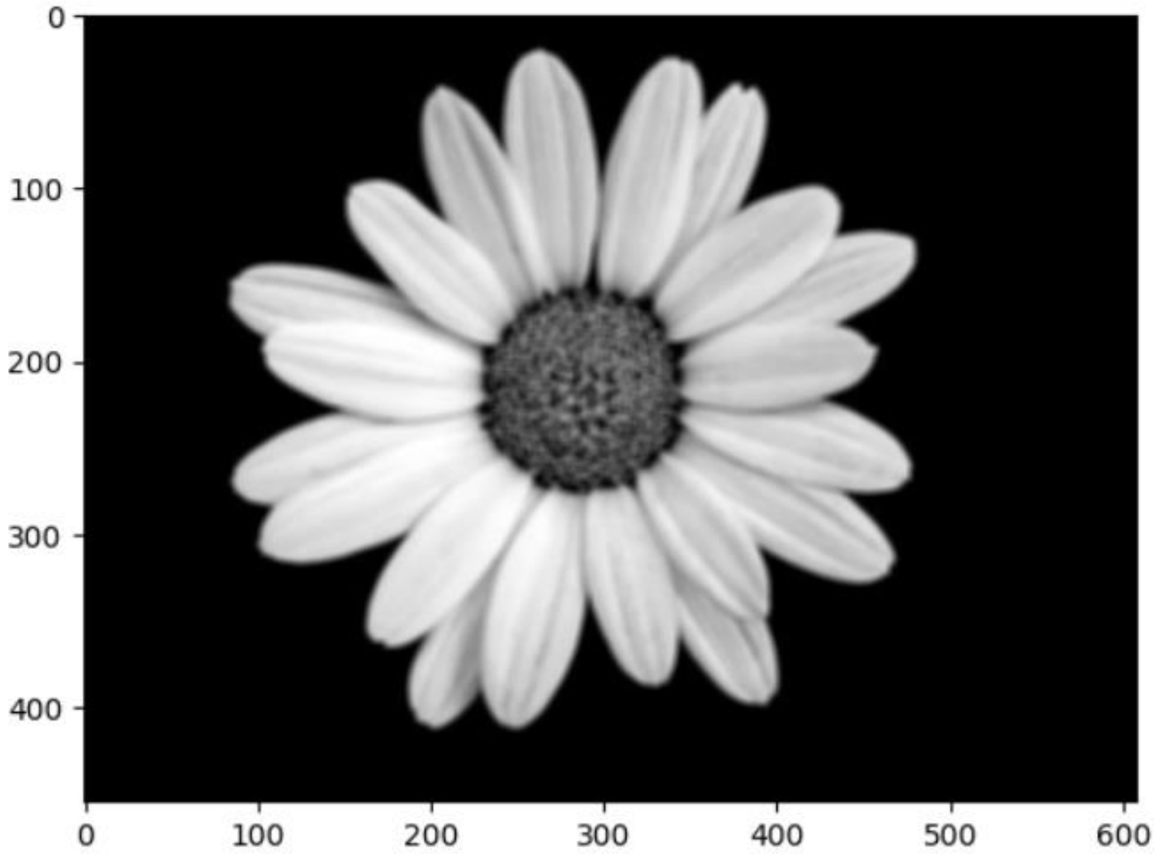
$$K = \frac{1}{25}\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Its effect is to **transform the image in a more blurred version of it.**

```
# Blur kernel
kernel = np.array([[1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1]])/25
image_conv = convolution(im_array, kernel)
print(image_conv.shape)
```

```
(455, 608)
```

```
# Display image in matplotlib
plt.imshow(image_conv, cmap = 'gray')
plt.show()
```



```python
# Open the image and convert it to grayscale
im = Image.open('flower.jpg').convert('L')

# Convert the image to a NumPy array
im_array = np.array(im)

# Print the shape of the array
print(im_array.shape)
```

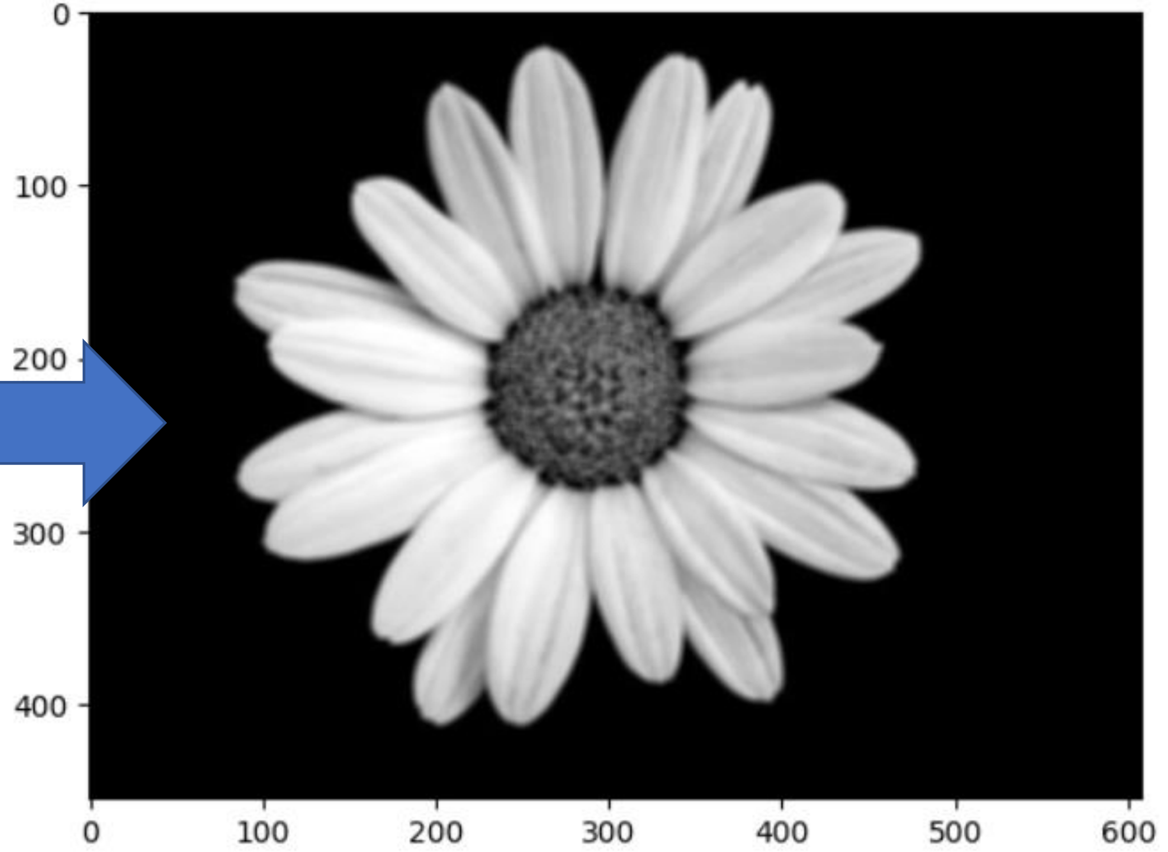
```
(459, 612)
```

```python
# Display image in matplotlib
plt.imshow(im_array, cmap = 'gray')
plt.show()
```


```python
# Blur kernel
kernel = np.array([[1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1]])/25
image_conv = convolution(im_array, kernel)
print(image_conv.shape)
```

```
(455, 608)
```

```python
# Display image in matplotlib
plt.imshow(image_conv, cmap = 'gray')
plt.show()
```



Testing convolution on images

Definition (the edge detection kernel):

We can also perform **edge detection**, using, for instance the 3×3 **Prewitt horizontal kernel**, defined as:

$$K = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

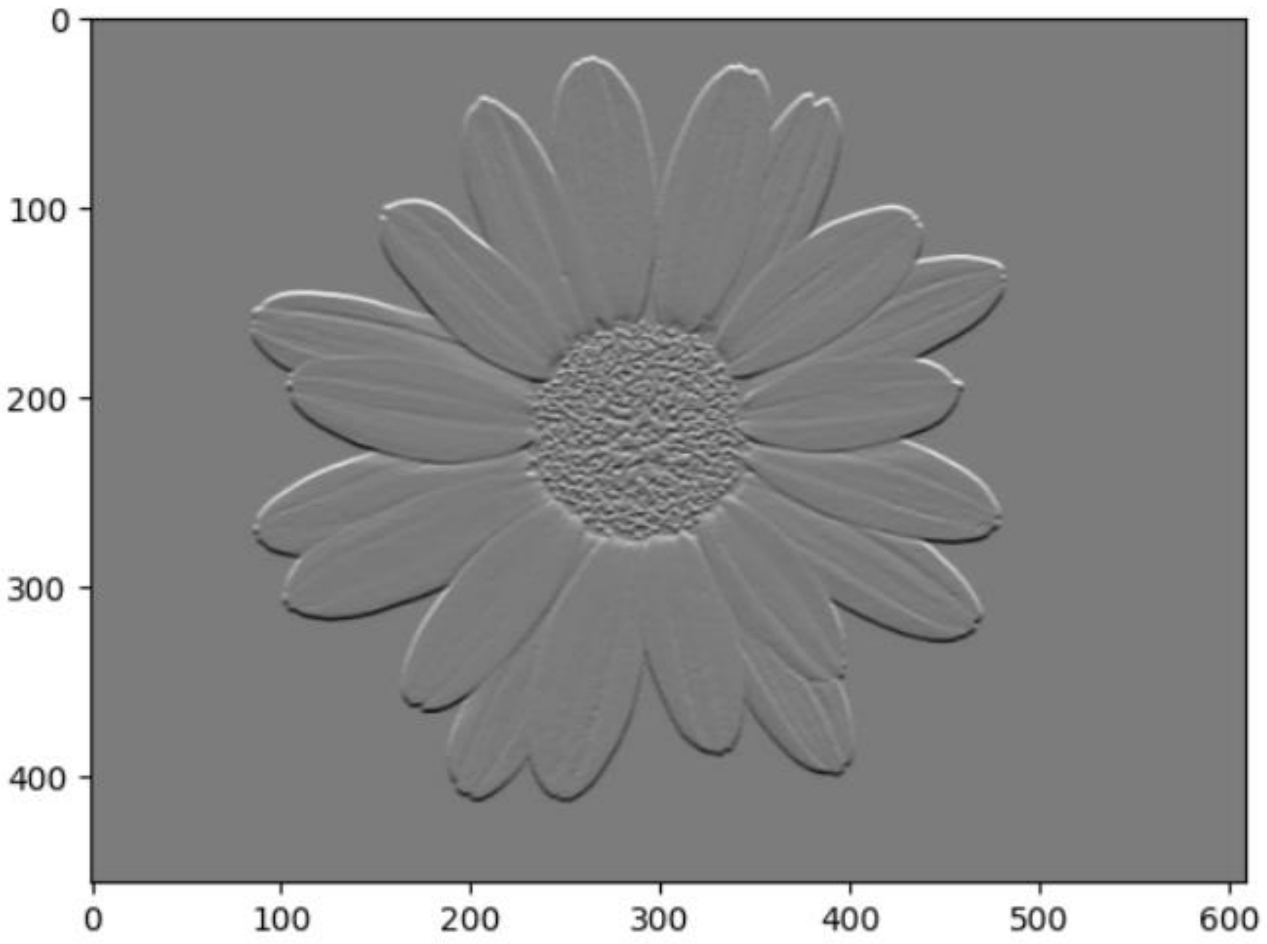
This kernel highlights the horizontal edges in the image.

Re

```
# Trying one kernel (Prewitt kernel for horizontal edges)
kernel1 = np.array([[1, 1, 1],[0, 0, 0],[-1, -1, -1]])
image_conv1 = convolution(im_array, kernel1)
print(image_conv1.shape)
```

```
(457, 610)
```

```
# Display image in matplotlib
plt.imshow(image_conv1, cmap = 'gray')
plt.show()
```



Re

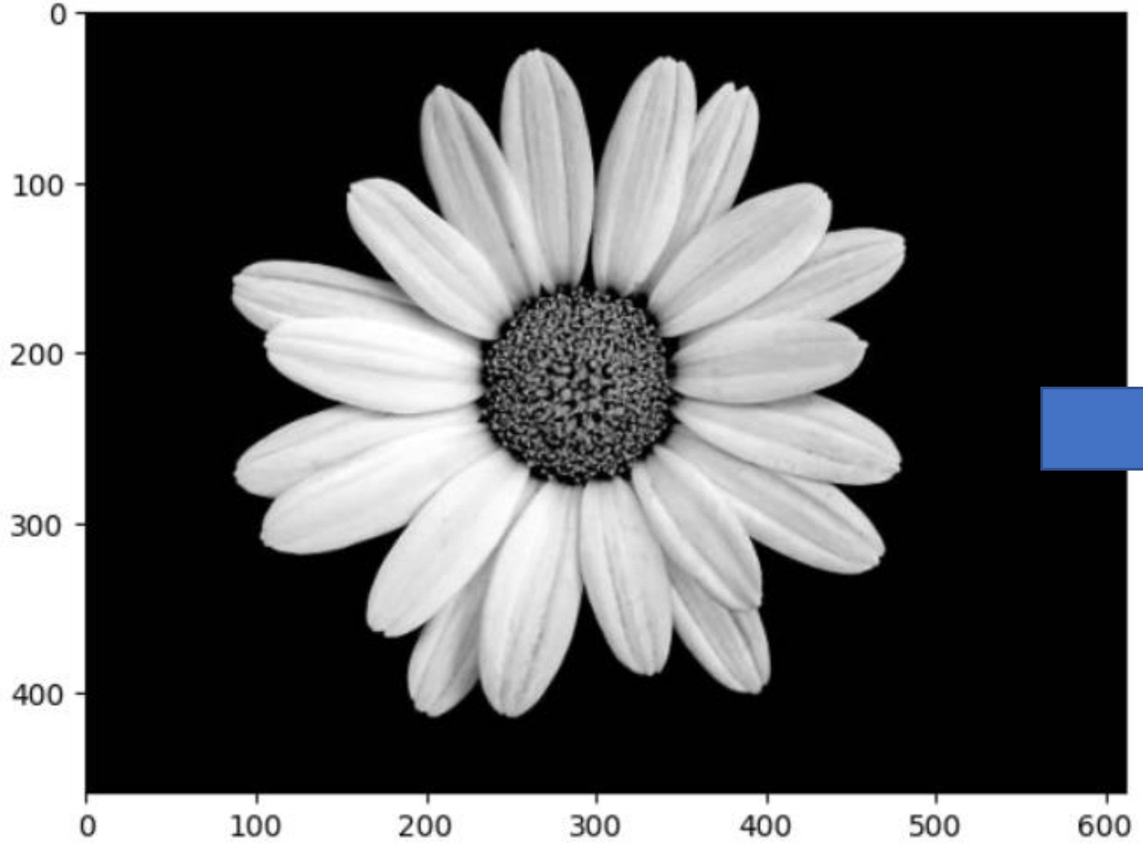
```python
# Open the image and convert it to grayscale
im = Image.open('flower.jpg').convert('L')

# Convert the image to a NumPy array
im_array = np.array(im)

# Print the shape of the array
print(im_array.shape)
```

(459, 612)

```python
# Display image in matplotlib
plt.imshow(im_array, cmap = 'gray')
plt.show()
```

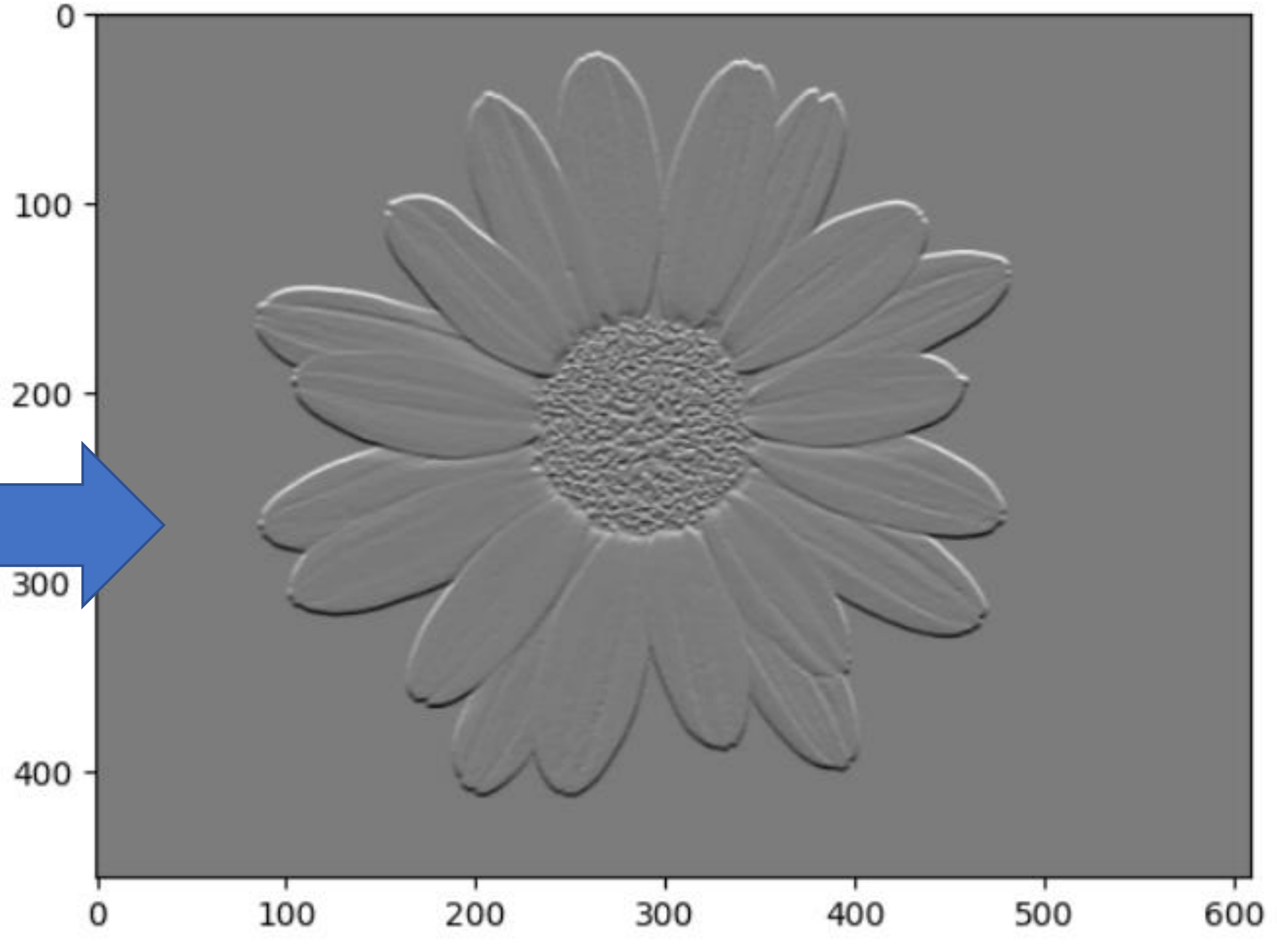


Re

```python
# Trying one kernel (Prewitt kernel for horizontal edges)
kernel1 = np.array([[1, 1, 1],[0, 0, 0],[-1, -1, -1]])
image_conv1 = convolution(im_array, kernel1)
print(image_conv1.shape)
```

(457, 610)

```python
# Display image in matplotlib
plt.imshow(image_conv1, cmap = 'gray')
plt.show()
```



Re

Testing convolution on images

Definition (the sharpen, or contrast enhancer kernel):

We can also perform **contrast enhancement** (or **sharpening**), using, the following 3×3 **sharpen kernel**, defined as:

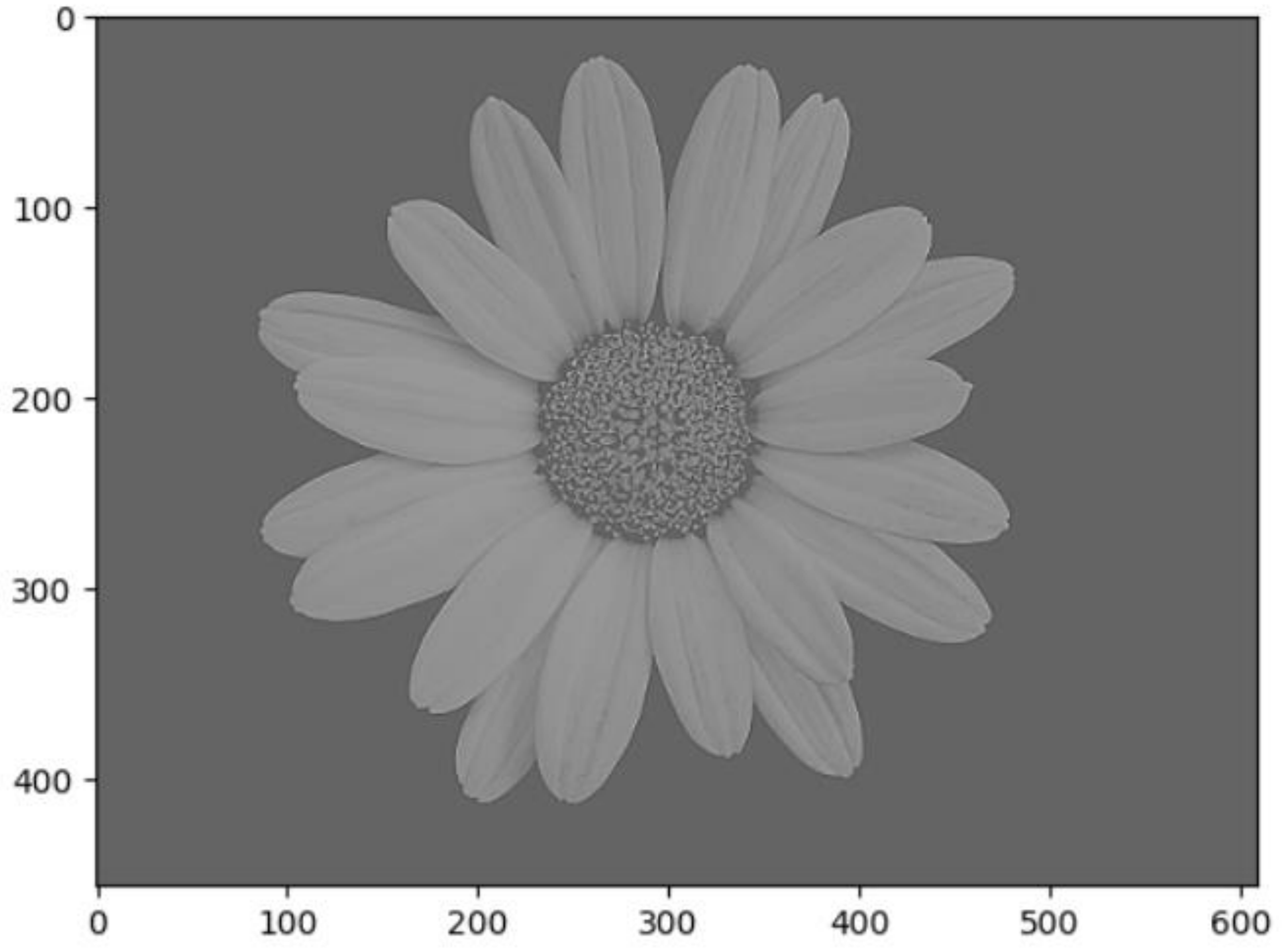
$$K = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

This kernel typically enhances the edges and high-frequency details

```
# Sharpen (= improve contrast)
kernel5 = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])
image_conv5 = convolution(im_array, kernel5)
print(image_conv5.shape)
```

```
(457, 610)
```

```
# Display image in matplotlib
plt.imshow(image_conv5, cmap = 'gray')
plt.show()
```



```python
# Open the image and convert it to grayscale
im = Image.open('flower.jpg').convert('L')

# Convert the image to a NumPy array
im_array = np.array(im)

# Print the shape of the array
print(im_array.shape)
```

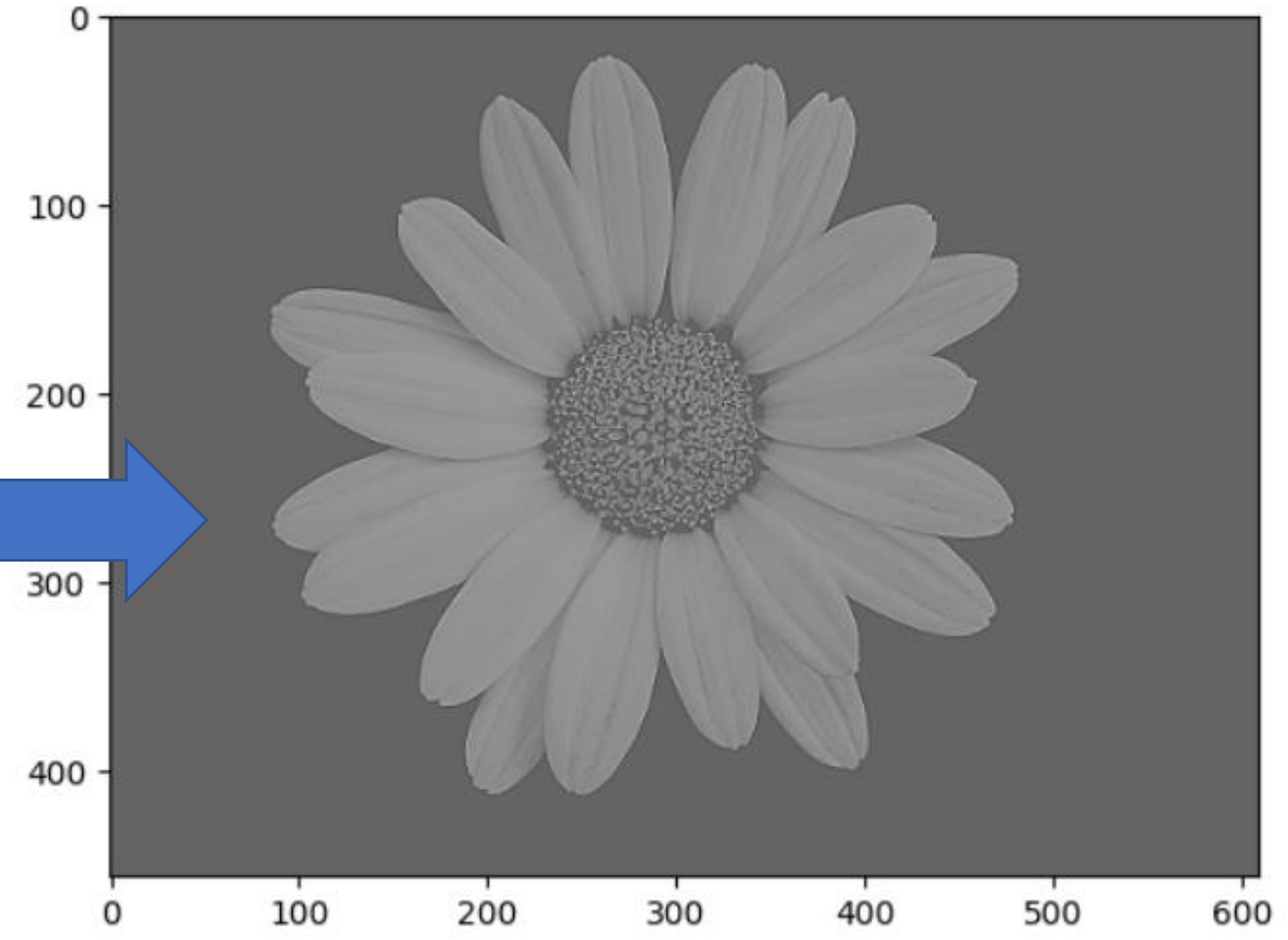
```
(459, 612)
```

```python
# Display image in matplotlib
plt.imshow(im_array, cmap = 'gray')
plt.show()
```

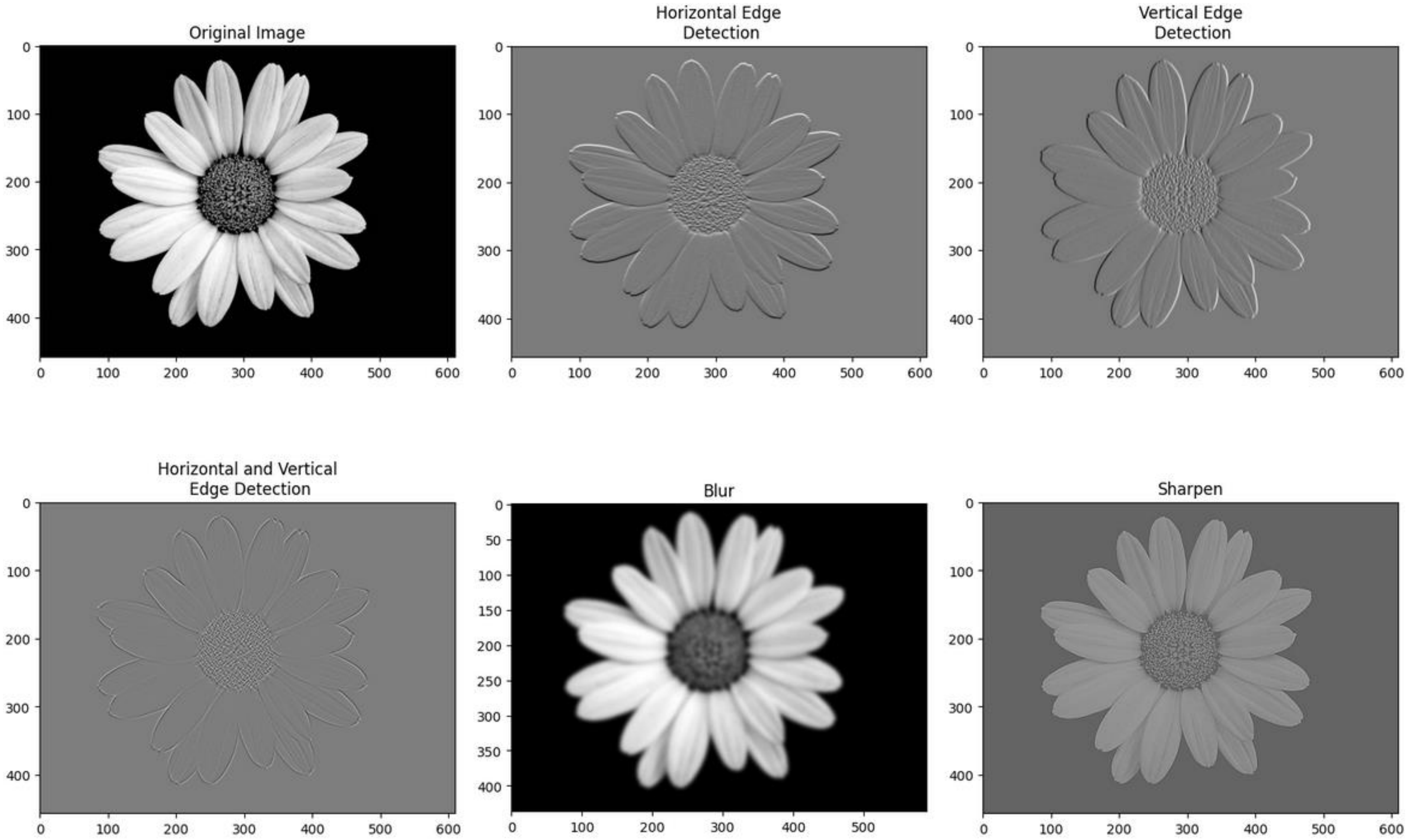

```python
# Sharpen (= improve contrast)
kernel5 = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])
image_conv5 = convolution(im_array, kernel5)
print(image_conv5.shape)
```

```
(457, 610)
```

```python
# Display image in matplotlib
plt.imshow(image_conv5, cmap = 'gray')
plt.show()
```



Many transformation kernels exist



Downsizing

Remember: The convolution operation defined as $Y = f(X, K)$, produces an image Y of size $h' \times w'$,

- With $h' = h - k + 1$,
- And $w' = w - k + 1$.

A typical problem with the convolution operation we defined earlier is that **it produces a new image Y whose size/resolution has been reduced**.

This has to do with the kernel being used, which sums several pixels together, but only produces one pixel as a result.

This effect is even increased with the size of the kernel being used.

Adding some padding

Definition (padding in convolution):

A typical way to address this issue consists of **padding** the image.

Padding consists of **adding some extra pixels on the outer part of the original image X.**

Padding artificially increases the size of the original image X, so that the convolution produces an image Y matching the size of X.

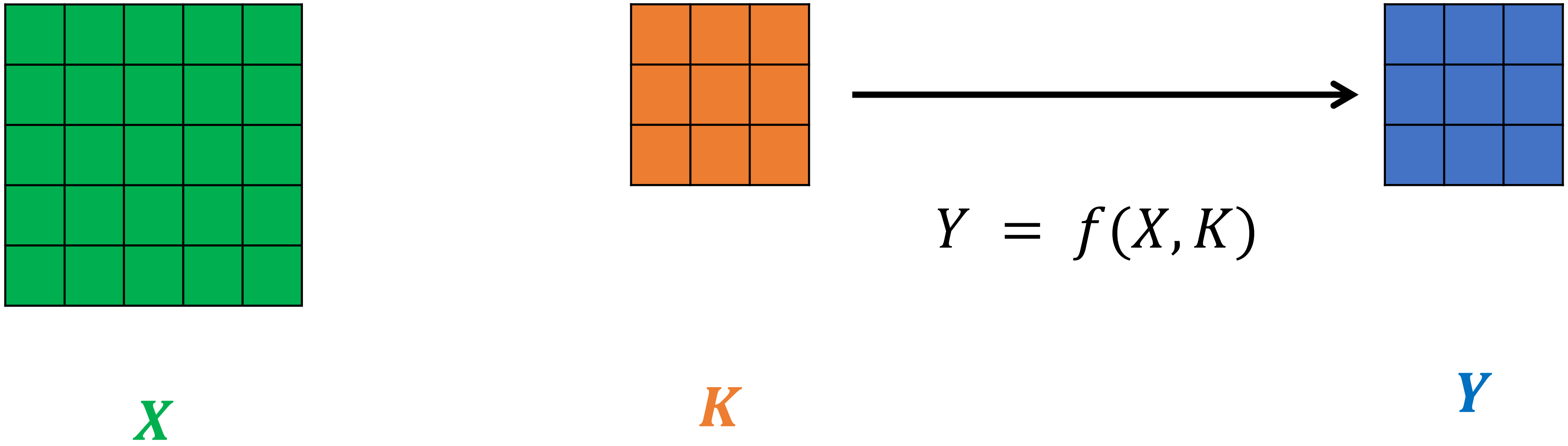
Three possible patterns for padding exist:

- Valid/Same padding,
- Zero padding,
- Other types of padding…

This is typically be used to maintain the spatial dimensions of the input image and prevent the spatial dimensions of the output from getting too small.

Adding some padding

Valid padding: default configuration, no padding is applied to the input data. The convolution operation is only performed only on the valid parts of the input and the output size is then smaller than the input size.

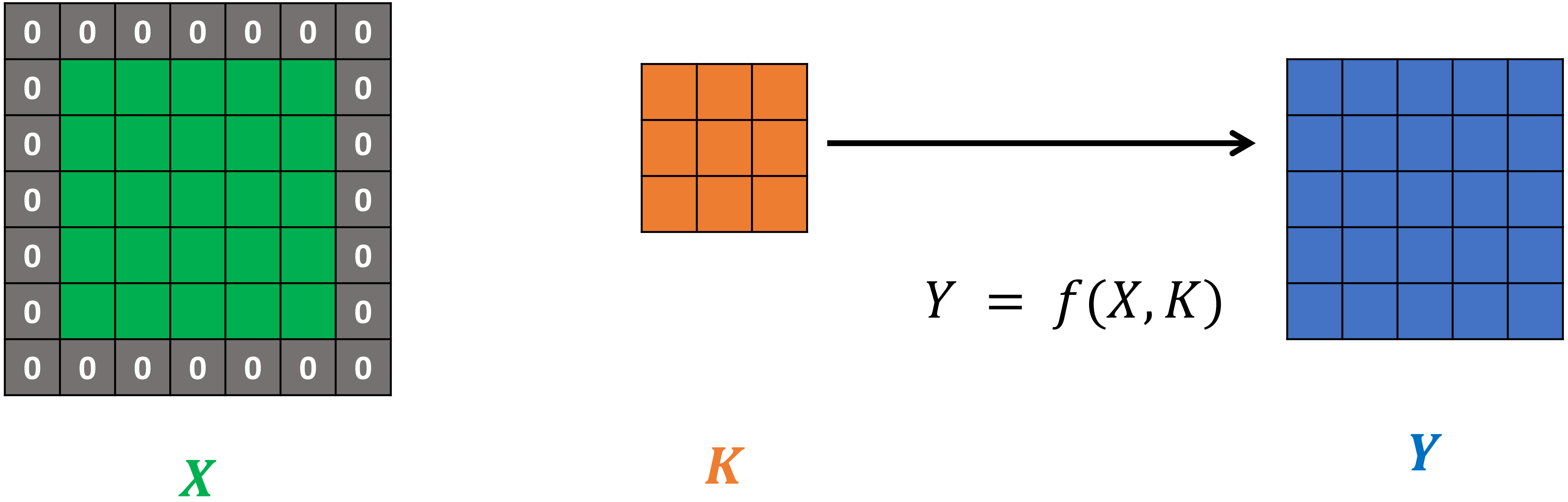


$$Y = f(X, K)$$

Adding some padding

Same padding: The number of zeros added (i.e. **padding size $\boldsymbol{p}$**) is defined so that the image Y has the same size as image X.

Zero padding: added pixels can simply be set to have the value 0.



Restricted

Adding some padding

When using a padding with size $\boldsymbol{p}$, the convolution operation defined as $Y = f(X, K)$, produces an image Y of size $h' \times w'$,

- With $h' = h + \mathbf{2p} - k + 1$,
- And $w' = w + \mathbf{2p} - k + 1$.

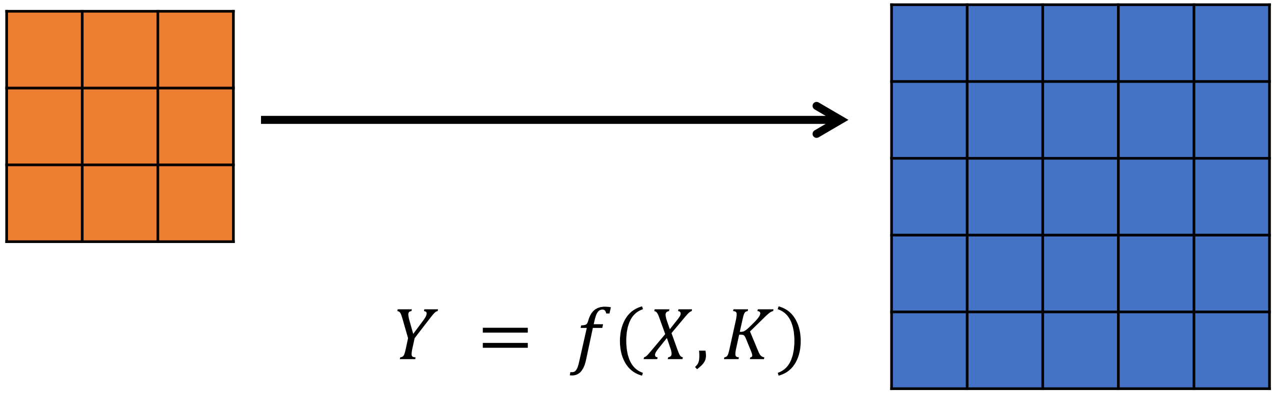
Same padding: to ensure the image Y has the same dimension as X, we need to use a padding size $p = \frac{k-1}{2}$. It is therefore adjusted to the kernel size k.

If we are not interested in Y having the same size as X, we are free to use any padding size.

Adding some padding

Interpolation padding: instead of zeros, added pixels have the value of the closest pixel. Can also “zoom in” the original image and resize it to the expected number of pixels we want for image X after padding.

1	1	5	5	2	4	4
1	1	5	5	2	4	4
2	2	4	5	2	2	2
3	3	1	1	1	1	1
4	4	4	7	2	3	3
5	5	0	0	3	1	1
5	5	0	0	0	1	1



$$Y = f(X, K)$$

$\boldsymbol{X}$ $\boldsymbol{K}$ $\boldsymbol{Y}$

Adding some padding

Padding is simply implemented using the pad() functions,

It adds a layer of zeroes to the matrix image, of size padding on top, bottom, right and left side of the image matrix.

```python
def convolution_with_padding(image, kernel, padding = 0):
    # Flip the kernel (optional)
    kernel = np.flipud(np.fliplr(kernel))

    # Get the dimensions of the image and kernel
    image_rows, image_cols = image.shape
    kernel_rows, kernel_cols = kernel.shape

    # Add padding to the image
    image = np.pad(image, ((padding, padding), (padding, padding)), 'constant')

    # Set the output image to the correct size
    output_rows = image_rows - kernel_rows + 1
    output_cols = image_cols - kernel_cols + 1
    output = np.zeros((output_rows, output_cols))

    # Convolve using Numpy
    output = correlate(image, kernel, mode = 'valid')

    return output
```

Adding some padding

- When using no padding, the resulting image is downsized.
- When using same padding, i.e.

$$p = \frac{k-1}{2} = \frac{5-1}{2} = 2,$$

the resulting image has the same size as the original one.

```
# Blur (no padding)
kernel = np.array([[1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1]])/25
image_conv = convolution_basic(im_array, kernel)
# Print the shape of the array
print(im_array.shape)
print(image_conv.shape)
```

```
(459, 612)
(455, 608)
```

```
# Blur (same padding)
kernel = np.array([[1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1]])/25
image_conv_pad = convolution_with_padding(im_array, kernel, padding = 2)
# Print the shape of the array
print(im_array.shape)
print(image_conv_pad.shape)
```

```
(459, 612)
(459, 612)
```

Adding a stride

Definition (stride in convolution):

The purpose of **stride** in convolution is to control the movement or step size of the convolution filter as it slides over the input image.

A larger stride results in a smaller output feature map, while a smaller stride results in a larger output feature map.

Stride can also be used to reduce the spatial dimensions of the feature map, which can help to reduce the number of parameters and computation in the network.

Note: Using a stride with a size s that is not 1, will drastically reduce the size of the output image, so beware!

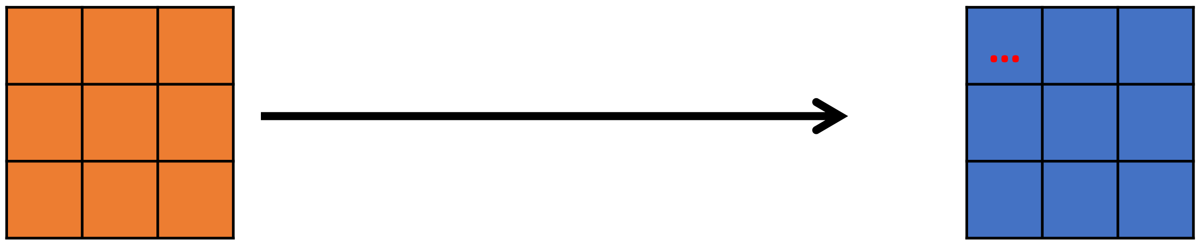
Adding a stride

Below, we demonstrate what a stride of size $s = 2$ does (no padding).

First, use the convolution as before, next elements to be used are “two steps” away from the first position, and so on.

1	1	5	5	2	4	4
1	1	5	5	2	4	4
2	2	4	5	2	2	2
3	3	1	1	1	1	1
4	4	4	7	2	3	3
5	5	0	0	3	1	1
5	5	0	0	0	1	1

$\boldsymbol{X}$



$$Y = f(X, K)$$

$\boldsymbol{K}$

$\boldsymbol{Y}$

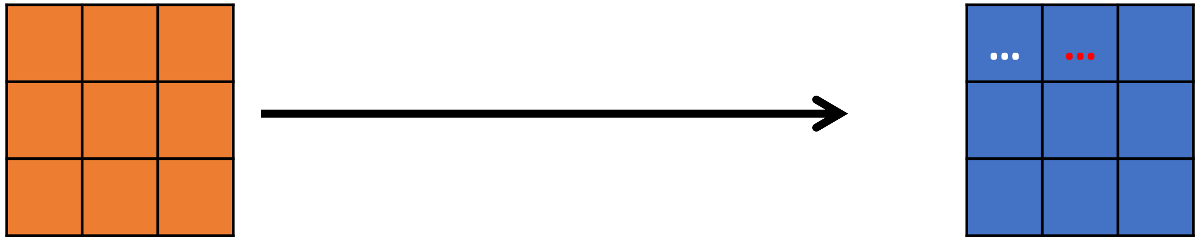
Restricted

Adding a stride

Below, we demonstrate what a stride of size $s = 2$ does (no padding).

First, use the convolution as before, next elements to be used are "two steps" away from the first position, and so on.

1	1	5	5	2	4	4
1	1	5	5	2	4	4
2	2	4	5	2	2	2
3	3	1	1	1	1	1
4	4	4	7	2	3	3
5	5	0	0	3	1	1
5	5	0	0	0	1	1



$$Y = f(X, K)$$

$\boldsymbol{X}$ $\boldsymbol{K}$ $\boldsymbol{Y}$

Restricted

Adding a stride

- Stride is easily implemented by using a slicing in the result of the correlation.
- This is not the most optimized way to do it, however!
- *(We do not care so much, as we will use the PyTorch version of convolution after this!)*

```python
def convolution_with_stride_and_padding(image, kernel, stride = 1, padding = 0):
    # Flip the kernel (optional)
    kernel = np.flipud(np.fliplr(kernel))

    # Get the dimensions of the image and kernel
    image_rows, image_cols = image.shape
    kernel_rows, kernel_cols = kernel.shape

    # Add padding to the image
    image = np.pad(image, ((padding, padding), (padding, padding)), 'constant')

    # Set the output image to the correct size
    output_rows = (image_rows - kernel_rows) // stride + 1
    output_cols = (image_cols - kernel_cols) // stride + 1
    output = np.zeros((output_rows, output_cols))

    # Convolve using Numpy
    # (not the most optimal way to implement it but good enough!)
    output = correlate(image, kernel, mode = 'valid')[::stride, ::stride]

    return output
```

The (final) magic formula for convolution

Magic formula for convolution: let us consider

- An input image, defined as a 3D tensor X of size $h \times w \times c$, with h the height, w the weight and c the number of channels.
- A convolution kernel K of size $k \times k$.
- A padding of size p, a stride of size s.

The resulting image Y will have a size $h' \times w' \times c$, with:

$$h' = \left\lfloor \frac{h + 2p - k}{s} + 1 \right\rfloor$$

And

$$w' = \left\lfloor \frac{w + 2p - k}{s} + 1 \right\rfloor$$

Note: the floor function $\lfloor \dots \rfloor$ is used in case the division result is an integer.

Adding a stride

```
# Blur (with stride and padding)
kernel = np.array([[1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1]])/25
image_conv_pad_stride = convolution_with_stride_and_padding(im_array, \
                                                            kernel, \
                                                            stride = 2, \
                                                            padding = 2)
# Print the shape of the array
print(im_array.shape)
print(image_conv_pad_stride.shape)
```

```
(459, 612)
(230, 306)
```

$$h' = \left\lfloor \frac{h + 2p - k}{s} + 1 \right\rfloor$$

$$h' = \left\lfloor \frac{459 + 2 \times 2 - 5}{2} + 1 \right\rfloor = 230$$

Adding a dilation

Definition (dilation in convolution):

Dilation in image convolution determines the spacing between the values of the original image multiplying the kernel.

By default, the value for dilation is $d = 1$.

With a dilation greater than 1, the pixel values of the original image multiplying the kernel are spread apart by a specified number of pixels.

This is very niche, and easier visualised than explained.

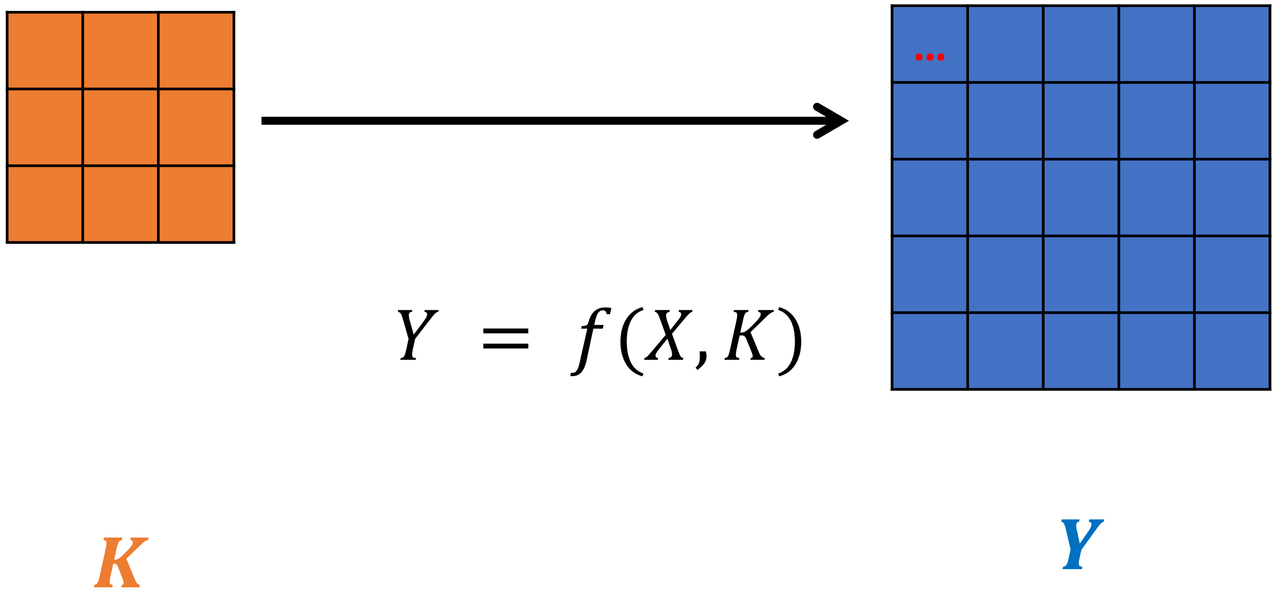
(We leave its implementation as practice to the reader.)

Adding a dilation

Below, we demonstrate what a dilation of size $d = 2$ does.
Stride is set to 1, padding is 0.

1	1	5	5	2	4	4
1	1	5	5	2	4	4
2	2	4	5	2	2	2
3	3	1	1	1	1	1
4	4	4	7	2	3	3
5	5	0	0	3	1	1
5	5	0	0	0	1	1

X



The (final) magic formula for convolution

Final magic formula for convolution: let us consider

- An input image, defined as a 3D tensor X of size $h \times w \times c$, with h the height, w the weight and c the number of channels.
- A convolution kernel K of size $k \times k \times c'$.
- A padding of size p, a stride of size s, and a dilation of size d.

The resulting image Y will have a size $h' \times w' \times c'$, with:

$$h' = \left\lfloor \frac{h + 2p - d(k-1) - 1}{s} + 1 \right\rfloor$$

And

$$w' = \left\lfloor \frac{w + 2p - d(k-1) - 1}{s} + 1 \right\rfloor$$

Note: the floor function $\lfloor \ldots \rfloor$ is used in case the division result is an integer.

Implementing a custom Conv in PyTorch

```
def convolution_with_stride_and_padding_torch(image, kernel, stride = 1, padding = 0):
    # Convert image and kernel to PyTorch tensors
    image = torch.from_numpy(image)
    kernel = torch.from_numpy(kernel)

    # Flip the kernel (optional)
    kernel = torch.flip(torch.flip(kernel, [0]), [1])

    # Add padding to the image
    image = torch.nn.functional.pad(image, (padding, padding, padding, padding))

    # Set the output image to the correct size
    output_rows = (image.shape[0] - kernel.shape[0]) // stride + 1
    output_cols = (image.shape[1] - kernel.shape[1]) // stride + 1
    output = torch.zeros((output_rows, output_cols))

    # Convolve using PyTorch
    for i in range(0, output_rows, stride):
        for j in range(0, output_cols, stride):
            output[i, j] = (kernel * image[i:i + kernel.shape[0], j:j + kernel.shape[1]]).sum()

    return output
```

Implementing a custom Conv in PyTorch

The PyTorch and the Numpy implementations are indeed equivalent.

```
# Blur
kernel = np.array([[1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1]])/25

# Numpy and torch implementations comparison
image_conv_pad_stride = convolution_with_stride_and_padding(im_array, kernel, stride = 2, padding = 2)
image_conv_pad_stride_torch = convolution_with_stride_and_padding_torch(im_array, kernel, stride = 2, padding = 2)
# Print the shape of the array
print(im_array.shape)
print(image_conv_pad_stride.shape)
print(image_conv_pad_stride_torch.shape)
```

```
(459, 612)
(230, 306)
torch.Size([230, 306])
```

The Conv2d in PyTorch

PyTorch has a Conv2d function implementing said convolution.

We will rely on it from now on.

```python
def convolution_batch_torch_conv2d(images, kernel, stride = 1, padding = 0):

    # Convert kernel to PyTorch tensor, if needed
    kernel = torch.from_numpy(kernel)
    kernel = kernel.view(1, 1, kernel.shape[0], kernel.shape[1])
    kernel = kernel.float()

    # Flip the kernel (optional)
    kernel = torch.flip(torch.flip(kernel, [2]), [3])

    # Create a convolutional layer
    conv = torch.nn.Conv2d(in_channels = images.shape[1], \
                           out_channels = 1, \
                           kernel_size = kernel.shape[2:], \
                           stride = stride, \
                           padding = padding)

    # Assign the kernel to the layer
    conv.weight = torch.nn.Parameter(kernel)
    conv.bias = torch.nn.Parameter(torch.tensor([0.0]))

    # Perform convolution
    output = conv(images)

    return output
```

The Conv2d in PyTorch

Note, however that this layer has **two trainable parameters**:

- **Weight** (which is our kernel, whose values will be decided later during training).
- **Bias** (which was not there before?)

```python
def convolution_batch_torch_conv2d(images, kernel, stride = 1, padding = 0):

    # Convert kernel to PyTorch tensor, if needed
    kernel = torch.from_numpy(kernel)
    kernel = kernel.view(1, 1, kernel.shape[0], kernel.shape[1])
    kernel = kernel.float()

    # Flip the kernel (optional)
    kernel = torch.flip(torch.flip(kernel, [2]), [3])

    # Create a convolutional layer
    conv = torch.nn.Conv2d(in_channels = images.shape[1], \
                           out_channels = 1, \
                           kernel_size = kernel.shape[2:], \
                           stride = stride, \
                           padding = padding)

    # Assign the kernel to the layer
    conv.weight = torch.nn.Parameter(kernel)
    conv.bias = torch.nn.Parameter(torch.tensor([0.0]))

    # Perform convolution
    output = conv(images)

    return output
```

The Conv2d in PyTorch

Note, however that this layer has **two trainable parameters**:

- **Weight** (which is our kernel, whose values will be decided later during training).
- **Bias** (which was not there before?)

Bias is simply added to our convolution operation, following the intuition of the $WX + b$ operation from earlier.

$$Y_{i,j} = \cancel{\frac{1}{k^2}} \sum_{m=1}^{k} \sum_{n=1}^{k} X_{i+m-1,\, j+n-1} K_{m,n} + b$$