浙江大学爱丁堡大学联合学院
ZJU-UoE Institute

Image preprocessing

Nicola Romanò - nicola.romano@ed.ac.uk

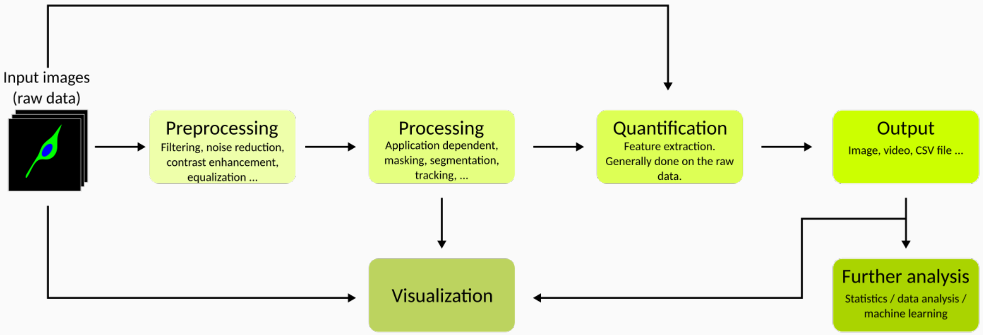
Learning objectives

At the end of this lecture you will be able to:

- Give examples of preprocessing in image analysis
- Explain basic preprocessing operations such as cropping, resizing, histogram manipulation and filtering
- Perform such operations using Python

A general quantitative image processing pipeline

When analysing a set of images you will need to process them to go from raw data to an output.



Input images
(raw data)
Preprocessing
Filtering, noise reduction, contrast enhancement, equalization ...
Processing
Application dependent, masking, segmentation, tracking, ...
Quantification
Feature extraction. Generally done on the raw data.
Output
Image, video, CSV file ...
Visualization
Further analysis
Statistics / data analysis / machine learning

What is preprocessing?

Preprocessing is a series of operations done on input images to improve their quality, suppress unwanted features such as noise, standardise or enhance them to improve further processing operations.

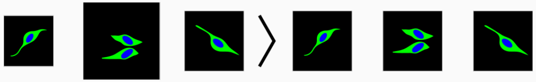
Preprocessing can happen at the pixel level (e.g. changing intensity) or at the whole image level (e.g. changing size).

It's important to always **keep your raw images** and to be aware of possible distortions in your data due to preprocessing (i.e. be sure to know what you are doing!)

Why preprocessing?

- You run an image-classification task to identify whether your image contains normal or cancer tissue. You decide to use a neural network to do so, but your images are all different sizes.

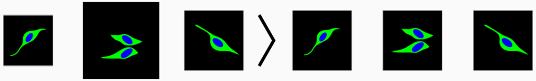
Preprocessing: you crop/rescale images so that they are all the same size.



Why preprocessing?

- You run an image-classification task to identify whether your image contains normal or cancer tissue. You decide to use a neural network to do so, but your images are all different sizes.

Preprocessing: you crop/rescale images so that they are all the same size.



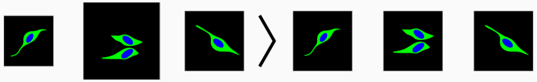
- You want to detect the outline of cells in a series of images (segmentation). Some of your images are undersaturated.

Preprocessing: you normalise the intensity in all of the images to be between 0 and max (e.g. 0 and 255 for an 8-bit image).

Why preprocessing?

- You run an image-classification task to identify whether your image contains normal or cancer tissue. You decide to use a neural network to do so, but your images are all different sizes.

Preprocessing: you crop/rescale images so that they are all the same size.

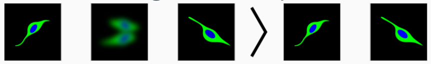


- You want to detect the outline of cells in a series of images (segmentation). Some of your images are undersaturated.

Preprocessing: you normalise the intensity in all of the images to be between 0 and max (e.g. 0 and 255 for an 8-bit image).

- You are analysing a set of automatically-acquired images from high-throughput screening. Some images are out of focus.

Preprocessing: you automatically remove out-of-focus images (not easy!).



Your toolkit

There are countless tools to process images, each with its strength and weaknesses.

For this task, your *best friends* in Python are (this is not a complete list)

- **Numpy** (see previous lecture) to deal with matrices
- **Scikit Image** and **OpenCV** for image processing
- **Matplotlib** for visualization
- **Pandas** for dealing with data frames
- **Scikit Learn** for machine learning (ML)
- **Keras** for advanced ML (convolutional neural network)

Scikit Image

Scikit Image is an open-source collection of algorithms for image processing.

Note that the official name of the library (e.g. for installing it using pip) is *scikit-image*. When you import it in a script you must use `import skimage`)

Scikit Image

Scikit Image is an open-source collection of algorithms for image processing.

Note that the official name of the library (e.g. for installing it using pip) is *scikit-image*. When you import it in a script you must use `import skimage`)

There are many submodules in Scikit Image that allow performing from simple to very complex operations on images. We will use some of the modules today, the full list can be found in the scikit-image documentation.

Image cropping

Image cropping is very easy using Numpy, just select the part of the image you want to take.

```
from skimage.io import imread
img = imread("cells.tif")
print(img.shape) # (256, 256)
```

Image cropping

Image cropping is very easy using Numpy, just select the part of the image you want to take.

```
from skimage.io import imread
img = imread("cells.tif")
print(img.shape) # (256, 256)
img_cropped = img[50:150, 60:160]
print(img_cropped.shape) # (100, 100)
```

Image cropping

Image cropping is very easy using Numpy, just select the part of the image you want to take.

```
from skimage.io import imread
img = imread("cells.tif")
print(img.shape) # (256, 256)
img_cropped = img[50:150, 60:160]
print(img_cropped.shape) # (100, 100)
# Also works in multiple dimensions
movie = imread("cellsmovie.tif") # shape (1000, 256, 256)
movie_start = img [1:100] # First 100 frames only
```

Image cropping

Image cropping is very easy using Numpy, just select the part of the image you want to take.

```
from skimage.io import imread
img = imread("cells.tif")
print(img.shape) # (256, 256)
img_cropped = img[50:150, 60:160]
print(img_cropped.shape) # (100, 100)
# Also works in multiple dimensions
movie = imread("cellsmovie.tif") # shape (1000, 256, 256)
movie_start = img [1:100] # First 100 frames only
# Note that time could have been on axis 2!
# In that case shape would have been (256, 256, 1000)
# You would have used img[:, :, 1:100]
```

Image cropping

Image cropping is very easy using Numpy, just select the part of the image you want to take.

```
from skimage.io import imread
img = imread("cells.tif")
print(img.shape) # (256, 256)
img_cropped = img[50:150, 60:160]
print(img_cropped.shape) # (100, 100)
# Also works in multiple dimensions
movie = imread("cellsmovie.tif") # shape (1000, 256, 256)
movie_start = img [1:100] # First 100 frames only
# Note that time could have been on axis 2!
# In that case shape would have been (256, 256, 1000)
# You would have used img[:, :, 1:100]
```

Cropping is a lossy operation. You **will lose information** from your original image, so proceed with caution.

Image resizing

When resizing an image the number of pixel in the output will differ from the input.

How to account for *lost* pixels when down-scaling or for *missing* pixels when up-scaling?
Interpolation!

Image resizing

When resizing an image the number of pixel in the output will differ from the input.

How to account for *lost* pixels when down-scaling or for *missing* pixels when up-scaling?
Interpolation!

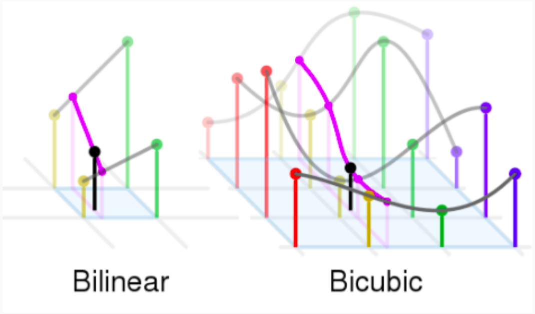


Image resizing - skimage

We can use *skimage.transform* to rescale an image.

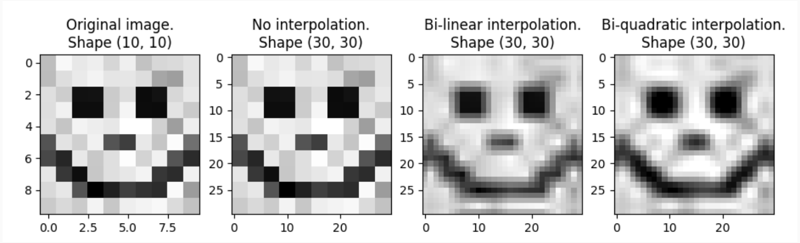
Three functions are available:

- **rescale** -> to scale up/down by a certain factor
- **resize** -> to scale up/down to a certain size
- **downscale_local_mean** -> downscale with local averaging

Image resizing - skimage

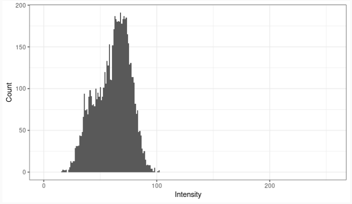
We can use *skimage.transform* to rescale an image.

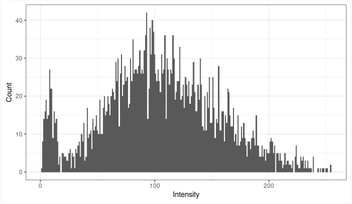
```
from skimage.transform import rescale
smiley = imread("smiley.png")
fig, ax = plt.subplots(1, 4)
ax[0].imshow(smiley, cmap="gray")
ax[1].imshow(rescale(smiley, 3, order=0), cmap="gray")
ax[2].imshow(rescale(smiley, 3, order=1), cmap="gray")
ax[3].imshow(rescale(smiley, 3, order=2), cmap="gray")
plt.show()
```

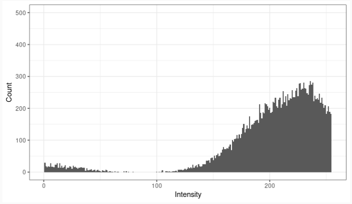


Over- and under-exposure

Histograms are a great tool to check the quality of your image. When adjusting the acquisition parameters on a microscope, care should be taken to use as much of the dynamic range as possible to avoid losing information.







Histogram manipulation - stretching

We can increase the contrast of our image by increasing the amount of dynamic range used by its pixels.

So, if the pixels in an 8-bit image are between 0 and 50, we are not using the possible information between 50 and 255.

Histogram manipulation - stretching

We can increase the contrast of our image by increasing the amount of dynamic range used by its pixels.

So, if the pixels in an 8-bit image are between 0 and 50, we are not using the possible information between 50 and 255.

The simplest way of solving this problem without retaking the image is to stretch the histogram.

If we have an image with intensity limits m_{im} and M_{im}, and want to stretch its histogram to m_{str} and M_{str} we can apply the following pixel-wise

$$I_{out} = (I_{in} - m_{im})(\frac{M_{str}-m_{str}}{M_{im}-m_{im}} + m_{str})$$

Histogram stretching in skimage

Luckily for us, *skimage.exposure* has a function for that!

```
import numpy as np
from skimage import exposure
img = imread("nuclei.tif")
img_rescale = exposure.rescale_intensity(img, in_range=(0, 100))
```

Histogram stretching in skimage

Luckily for us, *skimage.exposure* has a function for that!

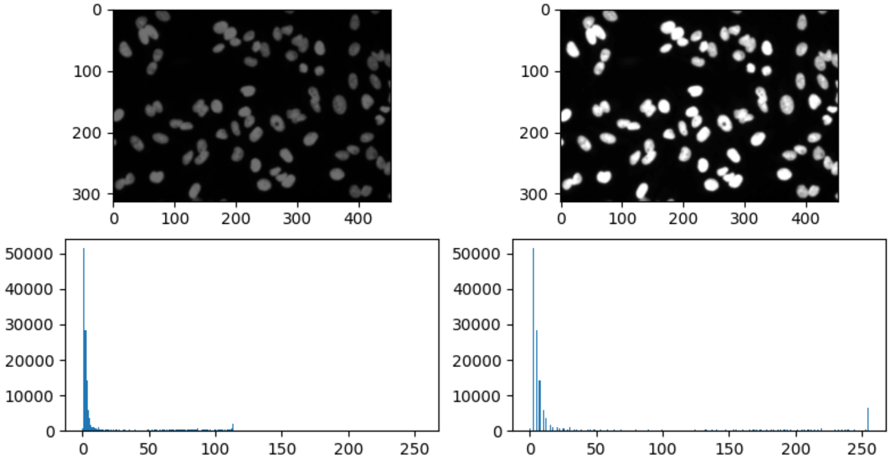
```
import numpy as np
from skimage import exposure
img = imread("nuclei.tif")
img_rescale = exposure.rescale_intensity(img, in_range=(0, 100))
```

Often we specify the input range dynamically, for instance by using intensity percentiles

```
p2, p98 = np.percentile(img, (2, 98))
img_rescale = exposure.rescale_intensity(img, in_range=(p2, p98))
```

Histogram stretching in skimage

Luckily for us, *skimage.exposure* has a function for that!



Histogram manipulation - equalization

Histogram equalization is a histogram manipulation technique that allows increasing image contrast by spreading the most common intensity values in the image in the possible intensity range.

Histogram manipulation - equalization

Histogram equalization is a histogram manipulation technique that allows increasing image contrast by spreading the most common intensity values in the image in the possible intensity range. It does so by flattening the cumulative density function (CDF) of the histogram.

Given the probability of a pixel to have intensity i (with i between 0 and the maximum pixel value M)

$$p_x(i) = p(x = i) = \frac{n_i}{n}, \quad 0 \leq i < M$$

Histogram manipulation - equalization

Histogram equalization is a histogram manipulation technique that allows increasing image contrast by spreading the most common intensity values in the image in the possible intensity range. It does so by flattening the cumulative density function (CDF) of the histogram.

Given the probability of a pixel to have intensity i (with i between 0 and the maximum pixel value M)

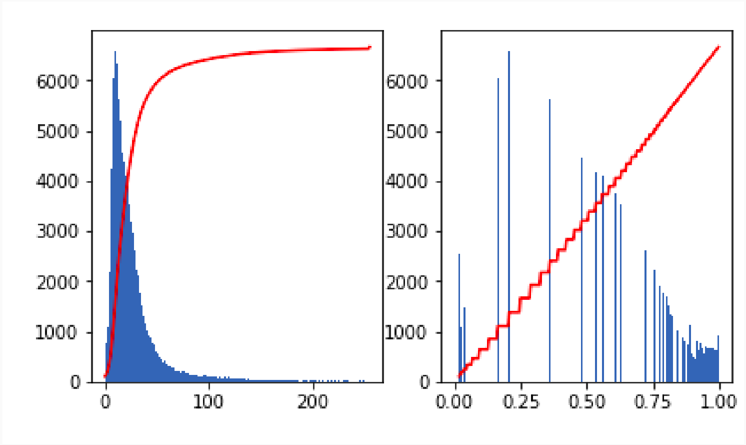
$$p_x(i) = p(x = i) = \frac{n_i}{n}, \quad 0 \leq i < M$$

The CDF is defined as

$$\text{CDF} = \sum_{j=0}^{i} p_x(x = j)$$

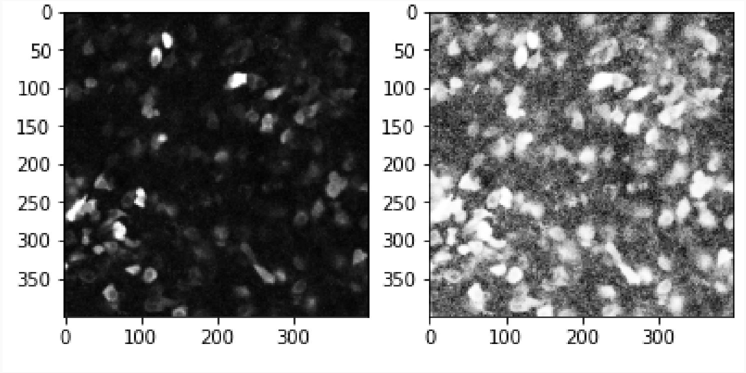
Histogram equalization in Python

```
from skimage.exposure import equalize_hist
img = imread("cells.tif")
img_eq = equalize_hist(img)
```



Histogram equalization in Python

```
from skimage.exposure import equalize_hist
img = imread("cells.tif")
img_eq = equalize_hist(img)
```



Equalization strengthens signals... but can also increase noise!

Better equalization

Other more sophisticated equalization strategies exist, such as **adaptive histogram equalization** (AHE) which computes several histograms of different regions of the image to improve local contrast. This tends to increase noise in areas with low contrast.

Better equalization

Other more sophisticated equalization strategies exist, such as **adaptive histogram equalization** (AHE) which computes several histograms of different regions of the image to improve local contrast. This tends to increase noise in areas with low contrast.

Contrast Limited Adaptive Equalization (CLAHE) solves this by reducing the amount of contrast enhancement.
This is implemented in the `skimage.exposure.equalize_adapthist` function.

Other preprocessing operations

Depending on the image and type of analysis you are working on, there are several other operations that can be performed, such as removing noise, blurring or sharpening the image, detecting edges etc.

These can be obtained by **convolution** of the image with a **kernel**.

Convolution takes each pixel of the image, together with its neighbours, and adds them together weighting the sum by the value of a **kernel** of the same size of the neighbourhood.

Other preprocessing operations

Depending on the image and type of analysis you are working on, there are several other operations that can be performed, such as removing noise, blurring or sharpening the image, detecting edges etc.

These can be obtained by **convolution** of the image with a **kernel**.

Convolution takes each pixel of the image, together with its neighbours, and adds them together weighting the sum by the value of a **kernel** of the same size of the neighbourhood.

The kernel can be of arbitrary size, and its values should generally sum to 1.

Convolution example

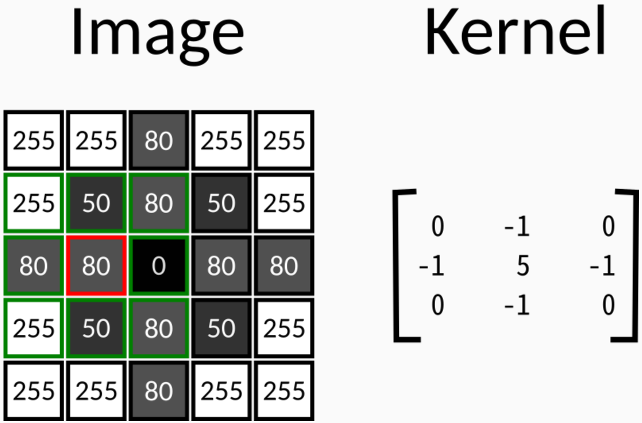
Image

255	255	80	255	255
255	50	80	50	255
80	80	0	80	80
255	50	80	50	255
255	255	80	255	255

Kernel

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Convolution example



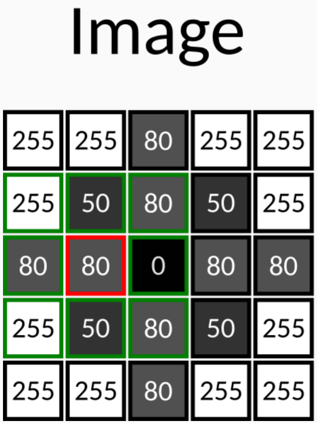
255	255	80	255	255
255	50	80	50	255
80	80	0	80	80
255	50	80	50	255
255	255	80	255	255

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The convolved pixel value will be

$$255*0+50*(-1)+80*0+80*(-1)+80*5+0*(-1)+255*0+50*(-1)+80*0 = \mathbf{220}$$

Convolution example



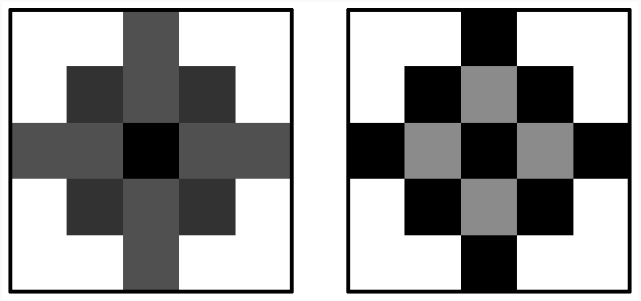
Kernel

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The convolved pixel value will be

$$255*0+50*(-1)+80*0+80*(-1)+80*5+0*(-1)+255*0+50*(-1)+80*0 = \mathbf{220}$$

Convolution example



Our image after applying the kernel to every pixel.

Pixel on the edges can be treated in different manner, for example they could be skipped, or the image could be "wrapped around", the kernel cropped... many other solutions exist!

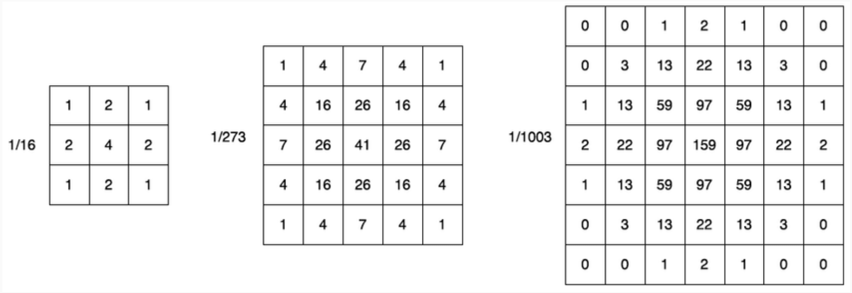
You can play around with filters on the "Explained Visually" website.

Filters in skimage

Scikit Image provides many filters out-of-the-box in the *filters* submodule. We will have a look at the most commonly used ones

Gaussian blur filter

This convolves the image with a Gaussian kernel, which results in stronger contribution of each pixel summed with a weaker contribution from its neighbours, thus resulting in the blurring of the image.



1/16

1	2	1
2	4	2
1	2	1

1/273

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

1/1003

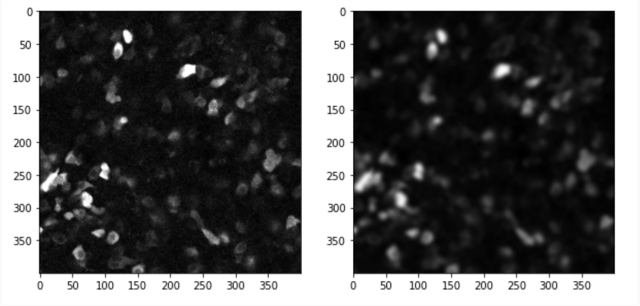
0	0	1	2	1	0	0
0	3	13	22	13	3	0
1	13	59	97	59	13	1
2	22	97	159	97	22	2
1	13	59	97	59	13	1
0	3	13	22	13	3	0
0	0	1	2	1	0	0

3x3, 5x5 and 7x7 Gaussian Kernels

Gaussian blur filter

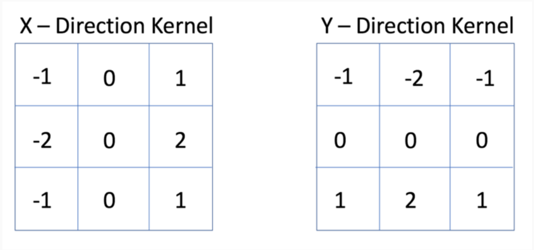
This convolves the image with a Gaussian kernel, which results in stronger contribution of each pixel summed with a weaker contribution from its neighbours, thus resulting in the blurring of the image.

```
from skimage.filters import gaussian
img = imread("cells.tif")
# Gaussian filter with kernel size of 3
img_blur = gaussian(img, 3)
```



Sobel filter

The Sobel filter is used for edge detection. It uses two kernels, one for determining horizontal edges and one for vertical edges



X – Direction Kernel

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

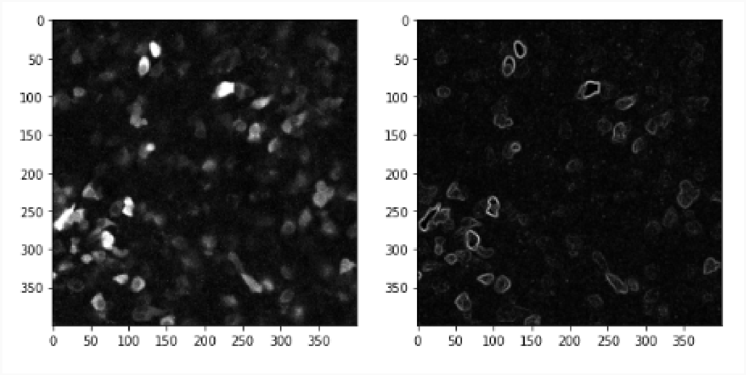
Y – Direction Kernel

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

Sobel filter

The Sobel filter is used for edge detection. It uses two kernels, one for determining horizontal edges and one for vertical edges

```
from skimage.filters import sobel
img = imread("cells.tif")
img_edges = sobel(img)
```



Median filter

The median filter is a commonly used denoising filter. It does not use a convolution kernel, but returns the median value in a $n \times n$ window around each pixel. Other denoising filters are available in the `skimage.restoration` submodule.

Median filter

The median filter is a commonly used denoising filter. It does not use a convolution kernel, but returns the median value in a $n \times n$ window around each pixel. Other denoising filters are available in the `skimage.restoration` submodule.

```
from skimage.filters import median
img = imread("cell.tif")
# Find the median in the 3x3 neighbourhood of # each pixel.
img_med = median(img, np.ones((3, 3))
```