Computer Vision

4.- Image Processing II

1 Image Processing: Detection, Extraction and Segmentation

In this topic, we will learn how to detect regions of interest in images, features extraction and segmentation.

1.1 Detection

1.1.1 Edge Detection

Edge detection is a fundamental process used to identify areas in an image where there is a significant change in intensity or color. These changes often indicate the presence of boundaries or transitions between regions in the image. Edge detection is crucial in many image processing and computer vision problems as it highlights important structures and features in the image.

In technical terms, an edge in an image refers to the region where the intensity of pixels changes drastically. This can occur, for example, at boundaries between objects, at the edge of an object against a background, or in areas of the image where there are significant changes in texture or lighting.

There are a lot of methods and algorithms for edge detection, but the most common are:

1. Gradient Operators: calculate the magnitude of the image gradient (a measure of how quickly the intensity of the image changes in a particular direction). Common operators include the Sobel operator and the Canny operator.

2. Second Derivative Operators: are based on the concept of second-order derivatives to identify changes in the curvature of the image intensity. An example is the Laplacian operator.

3. Mask-Based Filters: or kernels, are used to highlight intensity changes in the image. The Prewitt filter and the Roberts filter are typical examples.

Edge detection is an important step as a proccessing vision algorithms and it is common to combine edge detection with other processes such as noise reduction and feature extraction to achieve more robust and accurate results.

As an example, OpenCV has a function that directly implements the Sobel gradient in both directions using a 3x3 kernel (this is the default value if ksize is not specified):

```
# X Gradient
sobelx = cv.Sobel(img, cv.CV_64F, 1, 0, ksize=3)

# Y Gradient
sobely = cv.Sobel(img, cv.CV_64F, 0, 1, ksize=3)
```

We can use different kernels to implement gradients using convolution, where the kernel is the matrix that will be convolved with the original image. The complete syntax of this method was covered in the transformations topic, but you can find a complete examples at this link `https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html`.

OpenCV can be used to extract gradients in both directions from a grayscale image using convolutions based on different formulas, and the following function:

```
    filtered = cv.filter2D(img, -1, kernel)
```

Noise Reduction

Noise reduction is the process of minimizing or eliminating unwanted disturbances or variations in an image that are not representative of the underlying scene. Image noise can arise from various sources, including sensor imperfections, environmental conditions, or transmission errors. The main goal is to enhance the quality of an image by preserving relevant information while suppressing or filtering out unwanted noise.

Nowadays, Convolutional Neural Networks (CNNs) can be trained to learn noise patterns and remove noise from images. In this text, we focusing on classic noise reduction in computer vision:

- Smoothing Filters: two main filters, *Gaussian* filter to the image for blurring it and reducing high-frequency noise, and *Median* filtering, to replaces each pixel's value with the median value in its neighborhood (effective for removing salt-and-pepper noise).
- Image Averaging: two main techniques, *Temporal*, for video sequences, averaging frames over time can reduce temporal noise, and *Spatial* to average pixel values within a local neighborhood to reduce spatial noise.
- Wavelet Denoising: decomposes an image into wavelet transform components and selectively removes noise in specific frequency bands.

- Non-Local Means Filtering (NLMeans): compares local image patches to denoise each pixel, taking into account similar structures in the image.
- Bilateral Filter: preserves edges while smoothing the image by considering both spatial and intensity differences.
- Total Variation Denoising (regularization): minimizes the total variation of pixel intensities to preserve edges while reducing noise.

The selection of an appropriate noise reduction method is often a crucial preprocessing step in computer vision applications, and depends on the characteristics of the noise in the image, the desired level of detail preservation, and the computational requirements. It's common to apply a combination of these techniques to achieve effective noise reduction while maintaining important image features.

In OpenCV, there are several functions to remove the noise, as Gaussian filters (GaussianBlur function), which requires us to specify the size of the filter and its standard deviation:

```
img = cv.GaussianBlur(src, (3,3), 0) # Apply a Gaussian filter with
                                      a 3x3 pixel kernel and standard
                                     deviation 0
```

This function accepts more parameters, such as the standard deviation in the Y-axis (if omitted, it is the same as in the X-axis) or the type of interpolation at the edges (default is `cv.BORDER_DEFAULT`).

Or we can also apply a bilateral filter as follows:

```
img = cv.bilateralFilter(src, 15, 80, 80) # Apply a bilateral
                                     filter with a diameter of 15
                                     neighbor pixels and a minimum
                                     intensity of 80.
```

As you can see, the last parameters are two thresholds instead of one (it is a bit complicated to explain, but they are used for color images). Usually, the same value is used for both thresholds. If it is small (¡ 10), the filter will not have much effect. If it is large (¿ 150), it will have a strong effect, giving the image a cartoon style. For more information, you can check the function reference.

Finally, in OpenCV, we can use a Canny filter as follows:

```
img = cv.Canny(src, 100, 200) # Canny filter with the provided
                                     minimum and maximum (hysteresis)
                                     thresholds
```

For more information about Canny in OpenCV, you can check this link: `https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html#ga9d7064d478c95d60003cf839430737ed`. Like the previous functions, Canny filters can also have more parameters.

1.1.2 Line Detection

Line detection is the process of identifying and extracting straight lines from an image or a set of data points. Detecting lines is often a crucial step in understanding the structure and content of an image.

Several algorithms and techniques are commonly used for line detection, but the most relevant is Hough Transform.

The Hough Transform. Represents lines in polar coordinates (angle and distance from the origin) and accumulates votes in a parameter space. Peaks in the accumulator space correspond to detected lines. Probabilistic Hough Transform is an extension of the Hough Transform, more efficient and suitable for detecting only a subset of lines. It randomly selects points and fits lines through them, improving computational efficiency.

In OpenCV, the simplest way to perform the Hough transform to detect lines is as follows:

```
lines = cv.HoughLinesP(src, lines, rho, theta, threshold)
```

Where:

- *src* is an image with a single channel in grayscale (usually binary, as Hough is applied after Canny).
- *rho* is the resolution of the accumulator distance (in pixels).
- *theta* is the resolution of the accumulator angle (in pixels).
- And *threshold* is the accumulator threshold. Only lines with more votes than this threshold are returned.

The result is saved in *lines*, which is a vector of lines. Each line is another vector of 4 elements (x1, y1, x2, y2), where (x1, y1) and (x2, y2) are the endpoints of the line.

In addition to these parameters, there are two optional ones:

- *minLineLength*, which indicates the minimum length of a line to discard shorter segments than this length.
- And *maxLineGap*, which is the maximum allowed gap between points on the same line to link them.

The Hough function should always be used after an edge detector. For example:

```
edges = cv.Canny(src, 50, 200, None, 3)
lines = cv.HoughLinesP(edges, 1, np.pi/180, 50, None, 50, 10)
```

The Hough transform can also be used for the detection of other geometric shapes, for example, circles. Here's an example of calling the *HoughCircles* function:

```
circles = cv.HoughCircles(img, cv.HOUGH_GRADIENT, 1, 20, param1=50,
                                    param2=30, minRadius=0,
                                   maxRadius=0)
```

You can check the *HoughCircles* documentation for more information about these parameters: `https://docs.opencv.org/3.4/da/d53/tutorial_py_houghcircles.html`

The *approxPolyDP* function approximates a curve or a polygon with another curve/polygon with fewer vertices, so that the distance between them is less than or equal to the specified precision. It is implemented using the Douglas-Peucker algorithm:

```
closed = True
epsilon = 0.1*cv.arcLength(contour, closed)
approx = cv.approxPolyDP(contour, epsilon, closed)
```

The *epsilon* parameter is the maximum distance from the contour to the approximate contour, and *closed* indicates whether the contour is closed or not.

This function is usually used after extracting contours from an image using the *findContours* function, which we'll cover in detail in the next section (Segmentation)).

1.1.3 Isolated Point Detection

The Laplacian is the derivative of the gradient and can be used to detect isolated points.

$$\text{Laplacian}(f) = \nabla^2 f = \nabla \cdot \nabla f$$

where:

- $\nabla^2 f$ is the Laplacian of f, i.e., the second spatial derivative of the function f.
- ∇ is the gradient operator.
- $\nabla \cdot \nabla f$ denotes the divergence of the gradient of f.

In simpler terms, the Laplacian of a function is the divergence of its gradient. This operation highlights areas where the gradient changes, which, in the context of an image, often corresponds to edges, corners, or points of interest where intensity changes rapidly.

It can be implemented through convolution with a Laplacian kernel, but OpenCV provides the Laplacian function directly, which internally calls Sobel to calculate gradients. Example of usage:

```
ddepth = cv.CV_16S
kernel_size = 3
img = cv.Laplacian(src, ddepth, ksize=kernel_size)
```

In this code snippet, *ddepth* represents the desired depth of the destination image, and `kernel_size` specifies the size of the Laplacian kernel. The Laplacian operator is applied to the source image, helping to identify points of interest or areas with rapid intensity changes.

1.1.4 Corner Detection

In OpenCV, we can detect corners using Harris through the *cornerHarris* function. It needs as input a grayscale image and the following parameters: the number of neighboring pixels to consider, the filter size (*apertureSize*) to calculate gradients with Sobel, and the detection threshold k, which is the only free parameter of the Harris algorithm:

```
blockSize = 2  # Size of the neighborhood considered for corner
                                detection
apertureSize = 3  # Size of the kernel for the Sobel filter
k = 0.04  # Harris threshold

img = cv.cornerHarris(src, blockSize, apertureSize, k)
```

1.2 Features Extraction

In this topic, we will learn how to detect and extract useful features that describe an image.

1.2.1 Contour Descriptors

We will start by looking at features used to describe contours, which are numerical representations that capture the key features of an object's shape in an image. These descriptors are used to describe and compare contours, which are the curves outlining the boundaries of objects in an image.

These features assume that the image has been binarized beforehand, and we have the contour of the objects we want to recognize. When working with binarized images (where the objects of interest are in white and the background is in black, or vice versa), various attributes and features of contours can be extracted for analysis. Some common contour descriptors include the perimeter length (measures the total length of the contour), the convex Hull Area (calculates the area of the smallest convex polygon that envelops the contour), the compactness (measure of the "compactness" of the contour in relation to a circle), or moments, which we explain in following sections.

Moments

Moments are numerical statistics used to describe various properties of an image or an object within an image. These moments are calculated based on the distribution of pixel intensities in an image and are useful for characterizing the shape, orientation, and other aspects of an object.

Once we have extracted the contours of an image, for example using the findContours function in OpenCV whose syntax we saw in the previous topic, we can calculate the moment of a contour using the moments function:

```
moments = cv.moments(contour)
```

This function calculates all the moments of the contour. To access a specific moment, we can indicate, for example: *moments['m00']*. These are all the moments that the function returns:

```
# Spatial moments
m00, m10, m01, m20, m11, m02, m30, m21, m12, m03
# Central moments
mu20, mu11, mu02, mu30, mu21, mu12, mu03
# Normalized central moments
nu20, nu11, nu02, nu30, nu21, nu12, nu03
```

As you can see, not all moments are calculated. For example, since the moment 'mu00' is equal to 'm00', OpenCV does not extract it.

If you want to know more details about the moments function, you can check this link: `https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=moments#moments`

Hu Moments

The Hu moments are a set of seven invariant moments used in image analysis and pattern recognition, and are derived from the statistical moments of an image, describing how pixel intensities are distributed in relation to the image's center of mass. Hu moments are invariant to transformations such as scaling, rotation, and reflection, meaning they are useful for characterizing the shape of an object regardless of its size, orientation, and position in the image.

The core idea behind Hu moments is to capture geometric and topological properties of an object in a compact numerical representation. This representation is robust to the mentioned variations, making it valuable in tasks such as shape recognition and object comparison in images.

The formula for calculating Hu moments involves using normalized central moments, which are computed from the statistical moments of the image. They are calculated with the HuMoments function in OpenCV from the previously extracted normalized central moments using moments. For example:

```
hu = cv.HuMoments(moments)  # The hu array contains the 7 Hu
                            moments
```

The matchShapes function of OpenCV internally uses these Hu moments to compare contours: https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html#double%20matchShapes(InputArray%20contour1,%20InputArray%20contour2,%20int%20method,%20double%20parameter)

Shape Context (SC) Descriptor

The Shape Context Descriptor is a method to quantitatively describe and compare shapes, that central idea is to capture the distribution of points on a shape in relation to a reference point. There are four main steps:

- Key Point Sampling: key points on the shape are selected (are interest points representing distinctive features of the shape).
- Distance Normalization: for each key point, distances to other key points are normalized with respect to a central reference point to make the description invariant to the scale and position of the shape.
- Orientation Histograms: construction of histograms of the normalized directions of points with respect to the reference point. This captures the angular distribution of points around the central point.
- Final Description: concatenating the orientation histograms of all key points forms the SC descriptor for that specific shape.

This method has the advantage of being invariant to the scale, rotation, and translation of shapes, being very useful and robust in situations when shapes vary in size and orientation.

In OpenCV, to extract and compare contours with Shape Context, we can use the following code:

```
# Create an instance of this descriptor
mySC = cv.createShapeContextDistanceExtractor()

# Apply it to two contours to obtain their distance.
distance = mySC.computeDistance(contour1, contour2)
```

1.2.2 Texture

The spatial arrangement of pixel intensities in an image is called Texture, which provides information about the visual patterns and structures present. Analyzing texture is an important aspect of computer vision, as it enables systems to understand and recognize different surfaces, materials, or objects based on their textural characteristics.

Several techniques are used to extract and analyze texture information from images. One common approach is the use of texture descriptors, which are

mathematical representations of the texture patterns present in an image. These descriptors can be computed using various methods, such as statistical measures, frequency domain analysis, or spatial domain analysis. For example (avoiding CNNs), we can classify in three different analysis: statistical, frequency domain and spatial domain. Exists some techniques such as:

- Gray-Level Co-occurrence Matrix (GLCM): is a Statistical texture analysis. Describes the spatial relationships between pairs of pixels with the same intensity in an image.
- Gray-Level Run-Length Matrix (GLRLM): is other Statistical texture analysis. Captures the lengths of consecutive pixels with the same intensity in different directions.
- Gabor Filters: is a Frequency Domain Analysis method. Used to analyze the frequency content and orientation of textures in an image.
- Local Binary Patterns (LBP): is a Spatial Domain Analysis method. Describes the local patterns of pixel intensities in an image.
- Histogram of Oriented Gradients (HOG): another spatial method. Captures the distribution of gradient orientations in local image regions.

By understanding the texture properties of an image, computer vision systems can make more informed decisions about the content and context of visual data.

For instance, Gabor filters are implemented in OpenCV by creating a kernel using the getGaborKernel function, which can then be convolved with an image using filter2D, similar to any other filter.

```
ksize = 32
sigma = 1
theta = 0
lambd = 1.0
gamma = 0.02
psi = 0

kernel = cv.getGaborKernel((ksize, ksize), sigma, theta, lambd,
                                    gamma, psi)

final = cv.filter2D(img, -1, kernel)
```

As you can see, a Gabor filter can be constructed with various parameters, but the main ones are:

- ksize: Size of the filter.
- sigma: Standard deviation of the Gaussian envelope.
- theta: Orientation of the parallel bands in the Gabor function.

- lambd: Wavelength of the sinusoidal signal.

With respect to HOG method, in OpenCV we can extract it using *HOGDescriptor*:

```
winSize = (32, 16)
blockSize = (8, 8)
blockStride = (4, 4)
cellSize = (4, 4)
nbins = 9

hog = cv.HOGDescriptor(winSize, blockSize, blockStride, cellSize,
                                    nbins)
```

To simplify, the main parameters of the constructor (although there are more) are as follows:

- winSize: Size of the window.
- blockSize: Size of the block.
- blockStride: Block stride.
- cellSize: Size of the cell.
- nbins: Number of bins used to compute the histogram of gradients.

We can also create an HOG descriptor with default values:

```
hog = cv.HOGDescriptor()
# Equivalent to: cv.HOGDescriptor((64, 128), (16, 16), (8, 8), (8,
                                    8), 9)
```

Once the descriptor is created, it can be applied to an image as follows:

```
winStride = (0,0)
padding = (0,0)

descriptors = hog.compute(img, winStride, padding, locations)
```

The *compute* function stores the points where people are found in the image in the locations vector and the descriptor values for each point in the descriptors vector.

If instead of extracting the descriptor, you want to directly detect people in an image (which is more common), you can use the following code, but it is based on a machine learning, which is not seen in this section:

```
# The following instruction initializes a person detector.
hog.setSVMDetector(cv.HOGDescriptor_getDefaultPeopleDetector())
# Apply the detector to the image
hog.detectMultiScale(img)
```

For more information on the options of *detectMultiScale*, you can check this link: `https://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html#hogdescriptor-defaultpeopledetector)`.

Unfortunately, in OpenCV, there is no straightforward way to visualize the gradients of the HOG descriptor. However, the scikit-image library has convenient functions for calculating and visualizing HOG.

1.2.3 Local features

Local feature extraction is a preprocess that involves identifying and describing key points or regions within an image. These local features are usedin tasks such as image matching, object recognition, and scene understanding. First, we have to differenciate the concepts of keypoints and local descriptors:

Key Points (Keypoints). Are distinctive locations in an image that are stable under transformations such as rotation, scaling, and illumination changes. They are often located at corners, edges, or other salient structures in the image.

Local Descriptor. Are representations of the image content around a key point. These descriptors capture the visual information within a local neighborhood of the key point and are used to distinguish one key point from another.

To detect features and descriptions in an image, there are known algorithms such as:

- Harris Corner Detector: identifies corners in an image based on variations in intensity.
- FAST (Features from Accelerated Segment Test): a corner detection algorithm that is computationally efficient.
- SIFT (Scale-Invariant Feature Transform): detects and describes keypoints, and is invariant to scale and rotation changes.
- SURF (Speeded-Up Robust Features): similar to SIFT but computationally faster.
- MSER (Maximally Stable Extremal Regions): detection of stable regions or blobs in images. Robust to variations in scale, illumination, and viewpoint.
- ORB (Oriented FAST and Rotated BRIEF): combines the efficiency of FAST and the descriptiveness of BRIEF.

Moreover, we can apply two postprocessing methods for complementary actions as local feature matching, after extracting local features and their descriptors from two images, where the task is to match corresponding features between

the images. This is often done using distance metrics (e.g., Euclidean distance) to find the closest matches. Or RANSAC (Random Sample Consensus), used to robustly estimate geometric transformations (e.g., affine or homography) between images based on the matched key points.

The common uses are object recognition, image stitching and augmented reality. In OpenCV, there are implemented these methods. For instance, we can see MSER and SIFT:

```
#MSER
import cv2 as cv
import argparse
import numpy as np

# Read the image in grayscale
img = cv.imread('lena.jpg', cv.IMREAD_GRAYSCALE)

# Create the detector
detector = cv.MSER_create()

# Apply the detector to obtain keypoints
keypoints = detector.detect(gray, None)

# Draw the keypoints on the image. The last option is to show
                                    circles with their corresponding
                                    size.
output = cv.drawKeypoints(img, keypoints, None, flags=cv.
                                    DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
                                    )

# Visualize the result
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1), plt.imshow(img, cmap="gray"), plt.title("
                                    Original Image")
plt.subplot(1, 2, 2), plt.imshow(output, cmap="gray"), plt.title("
                                    Keypoints")

plt.show()
```

SIFT example. We replace *MSERcreate* with *SIFTcreate* to use a SIFT detector.

In general, OpenCV provides many combinations of detectors and descriptors, as can be seen in this link: `https://docs.opencv.org/master/da/df5/tutorial_py_sift_intro.html`).

SIFT was patented, but its rights have expired in 2020, so it can now be used without any problem in OpenCV. However, SURF is still under patent, and since OpenCV 4.2, it has been excluded from the library because its philosophy is to include only open-source content.

You can check many code examples using OpenCV detectors and descriptors in `https://docs.opencv.org/master/df/d65/tutorial_table_of_content_`

`feature2d.html`.

1.3 Segmentation

In this topic, we will see how to segment images by detecting pixels belonging to the objects of interest.

1.3.1 Thresholding

Thresholding is method for separate objects or regions of interest from the background in an image. The goal is to convert a grayscale or color image into a binary image, where pixels are classified as either part of the object (foreground) or the background based on a specified threshold.

The process of thresholding involves comparing the intensity or color values of pixels in an image with a predefined threshold value. Pixels with values below the threshold are assigned to one class (e.g., background), while pixels with values above the threshold are assigned to another class (e.g., foreground).

There are several thresholding methods such as:

- Binary Thresholding: is the simplest form of thresholding, that Pixels with intensity values below the threshold are set to 0 (black), and those above the threshold are set to 255 (white).
- Adaptive Thresholding: adjusts the threshold dynamically based on the local properties of the image. Useful when illumination conditions vary across the image.
- Otsu's Thresholding: automatically calculates an optimal threshold by maximizing the variance between two classes of pixels.
- Multi-level Thresholding: involves setting multiple threshold values to segment an image into more than two classes. Useful when dealing with images with multiple regions of interest.

Using OpenCV in Python, we can implement a binary thresholding as follows:

```
import cv2 as cv
from matplotlib import pyplot as plt

# Read the image in grayscale
img = cv.imread('lena.jpg', cv.IMREAD_GRAYSCALE)

# Apply binary thresholding
_, binary_image = cv.threshold(img, 128, 255, cv.THRESH_BINARY)

# Display the original and thresholded images
plt.subplot(121), plt.imshow(img, cmap='gray'), plt.title('Original
                                    Image')
```

```
plt.subplot(122), plt.imshow(binary_image, cmap='gray'), plt.title(
                                    'Binary Thresholding')
plt.show()
```

In this example, the threshold value is set to 128, and pixels with intensity values below 128 are set to 0, while those above 128 are set to 255. Adjusting the threshold value can significantly impact the results, and it often requires experimentation depending on the characteristics of the image.

As seen in this link (`https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html`), OpenCV provides various methods for basic thresholding using the *threshold* function. This function also implements Otsu's thresholding by specifying `cv.THRESH_OTSU` as a parameter.

Adaptive thresholding is implemented using the *adaptiveThreshold* function.

1.3.2 Contours

We can use an edge detection algorithm to subsequently estimate the contours of objects (and thus segment them). In OpenCV, there is a function for performing this task called findContours, which can only be used to extract contours from edges detected with another algorithm (that is, it works with a binary image as input). In this link (`https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=moments#findcontours`), you can see an example of a program that uses findContours and subsequently the drawContours function to draw the result using random colors.

Below, we can see an example of the syntax for *findContours*:

```
contours, hierarchy = cv.findContours(image, cv.RETR_LIST, cv.
                                    CHAIN_APPROX_NONE)
```

This function returns a list of contours detected in the image along with their hierarchy. Hierarchy (`https://docs.opencv.org/4.x/d9/d8b/tutorial_py_contours_hierarchy.html`) refers to the relationship between contours, as sometimes we have contours inside others (in such cases, the outer contours are "parent" and the inner contours are "children").

The second parameter of this function is the type of algorithm used to return the contours. The simplest method is `cv.RETR_LIST`, which returns a simple list and ignores the hierarchy. Alternatively, you can use, for example, `cv.RETR_TREE`, which contains the complete hierarchy.

The third parameter is the approximation method. All points of the contour are returned using `cv.CHAIN_APPROX_NONE`,. However, as this can be quite inefficient for some algorithms (as we will see in the next topic), point reduction techniques (`https://docs.opencv.org/master/d4/d73/tutorial_py_contours_`

`begin.html`) are sometimes used to simplify contours (i.e. using the option `cv.CHAIN_APPROX_SIMPLE`).

1.3.3 Region Growing and Splitting

Region growing is a pixel-based image segmentation technique that groups pixels based on certain criteria such as intensity, color, or texture. The basic idea behind region growing is to start with a seed pixel (or a set of seed pixels) and iteratively add neighboring pixels to the region if they satisfy a specified similarity criterion. This process continues until no more pixels can be added to the region.

A key term in this method is the *connectivity*, that refers to how neighboring pixels are defined in the context of the region-growing algorithm. The choice of connectivity depends on how you want neighboring pixels to be considered in the region-growing process. Some common options are:

- 4-connectivity: pixels that share a side are considered neighbors. The neighborhood includes pixels on the top, bottom, left, and right of the current pixel.
- 8-connectivity: pixels that share a side or a corner are considered neighbors. The neighborhood includes pixels on the top, bottom, left, right, and diagonals of the current pixel.
- 6-connectivity in 3D: in three-dimensional applications, connectivity in 6 directions can be considered, where neighboring pixels share a face.

Moreover, we have the split-and-merge method, that is a recursive image segmentation algorithm that divides an image into regions and merges them based on a certain criterion. The basic idea is to start with the entire image and recursively split it into smaller regions until some homogeneity criterion is met. Then, adjacent regions that satisfy a merging criterion are merged.

This technique has 4 steps: splitting, merging, homogeneity and merging criterion. In detail:

1. Splitting: start with the entire image as one region. checking a homogeneity criterion for the current region. If the criterion is not satisfied, split the region into four quadrants (or more, depending on the implementation). Repeat the splitting process for each sub-region.
2. Merging: after splitting, check for adjacent regions that satisfy a merging criterion. If the criterion is met, merge the adjacent regions. Repeat the merging process until no more merges can be performed.
3. Homogeneity Criterion: determines when to stop splitting a region. It is often based on the difference in intensity, color, or other image properties within the region.

4. Merging Criterion: determines when two adjacent regions should be merged. It is often based on the similarity of properties between the regions.

In OpenCv does not exists a specific funtion of this method, but its implementation is easy. We can see an example in this link: `https://vgg.fiit.stuba.sk/2016-06/split-and-merge/`.

1.3.4 Watershed

The watershed algorithm is a technique of image segmentation involves dividing an image into meaningful and semantically homogeneous regions, being particularly useful for segmenting images with irregular and complex structures, such as medical images or images with multiple objects touching each other.

The watershed algorithm is inspired by the concept of a watershed in hydrology, where watersheds are the dividing lines between different drainage basins. In image processing, the intensity values of an image are treated as a topographic surface, and the algorithm simulates a flooding process on this surface.

The algorithm works as follows:

1. At first, we calculate the image gradient by the intensity gradient or gradient magnitude of the image. The gradient represents the rate of change of intensity at each pixel.

2. Later, identify markers or seeds that will be used as starting points for the segmentation process. These markers can be manually defined or generated automatically based on certain criteria.

3. Then, we simulate a flooding process starting from the markers. The intensity values are treated as elevations, and the flooding process simulates raising the water level. Pixels are labeled with the marker index from which the flooding originated.

4. As the flooding process progresses, watershed lines (also called watershed boundaries) start to emerge between different regions. These lines represent the boundaries between objects.

5. Finally, the final segmented regions are obtained by analyzing the watershed lines. Each region corresponds to a catchment basin, and the watershed lines delineate the boundaries between different catchment basins

In OpenCV, we have the function *cv2.watershed*. There are two approaches: interactive (see an example in `https://github.com/opencv/opencv/blob/master/samples/python/watershed.py`) and non-interactive, that is, automatically deducing the initial markers.

For example, we have a non-interactive watershed as follows:

```python
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load an image
image = cv2.imread("image.jpg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert to RGB
                                    for displaying with matplotlib

# Convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# Apply thresholding to create a binary image
_, binary = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2
                                    .THRESH_OTSU)

# Perform morphological operations to clean up the image
kernel = np.ones((3,3), np.uint8)
sure_bg = cv2.dilate(binary, kernel, iterations=3)

# Distance Transform
dist_transform = cv2.distanceTransform(binary, cv2.DIST_L2, 5)
_, sure_fg = cv2.threshold(dist_transform, 0.7 * dist_transform.max
                                    (), 255, 0)

# Finding unknown region
sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(sure_bg, sure_fg)

# Marker labelling
_, markers = cv2.connectedComponents(sure_fg)

# Add 1 to all labels so that sure background is not 0, but 1
markers = markers + 1

# Mark the region of unknown with 0
markers[unknown == 255] = 0

# Apply Watershed algorithm
cv2.watershed(image, markers)
image[markers == -1] = [255, 0, 0]  # Mark watershed boundaries in
                                    red

# Display the result
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.imshow(image)
plt.title('Original Image')

plt.subplot(1, 2, 2)
plt.imshow(markers, cmap='viridis')
plt.title('Segmentation Result (Markers)')
plt.colorbar()

plt.show()
```

1.3.5 Clustering

Clustering is used to group similar or related data into sets called "clusters." There are several known methods for tasks such as image segmentation, object recognition, and pattern analysis, being the most common:

K-means. Groups data into k clusters based on the distance between points and centroids, and it used to group pixels in an image into regions.

Summurising, this method groups a dataset into k clusters, where each data point belongs to the cluster whose centroid is closest to it.There are a set of basic working steps:

1. Initialization: choose k initial centroids. They can be selected randomly from the dataset or by some other method, such as k-means++, which intelligently chooses initial centroids to improve convergence.
2. Point assignment to clusters: assign each data point to the cluster whose centroid is closest. This is done by calculating the distance (usually Euclidean) between each point and each centroid.
3. Centroid update: recalculate the centroids of the clusters as the average of all points assigned to that cluster. The centroids are the "center of gravity" of the points in the cluster.
4. Iteration: repeat steps 2 and 3 until the centroids no longer change significantly or until a predetermined number of iterations is reached.
5. Convergence: the algorithm converges when the centroids no longer change significantly between iterations. Convergence implies that the clusters have stabilized, and the algorithm has found a solution.

The final result is that each data point is assigned to a cluster, and each cluster is represented by its centroid. The K-Means algorithm aims to minimize the sum of squared distances between data points and their assigned centroids.

It's key the choice of the number of clusters k (is a critical parameter and can significantly affect the results). In practice, various k values can be tested, and the quality of the resulting clusters can be evaluated using metrics such as inertia (the sum of squared distances) or external indices if true labels are available for the data.

In OpenCV, we use *cv2.kmeans*:

```
# Specify criteria and apply K-Means clustering
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100
                                    , 0.2) #Defines the stopping
                                    criteria for the algorithm
k = 3  # Number of clusters
_, labels, centers = cv2.kmeans(data, k, None, criteria, 10, cv2.
                                    KMEANS_RANDOM_CENTERS)
```

Mean Shift. Seeks modes (peaks) in the data distribution. It is useful to find dense regions in the feature space.

Below are the basic steps of how the Mean Shift method works:

1. Initialization: select a set of seed points in the feature space of the dataset. These points represent the initial locations of modes or clusters.
2. Computation of Mean Shifts: for each seed point, calculate the mean shift vector, indicating the direction towards the region of higher data density. This vector is computed as the weighted mean of vectors from all neighboring points, weighted by the distance from the seed point.
3. Shift of Points: shift each point towards the region of higher density, following the calculated mean shift vector.
4. Convergence: repeat steps 2 and 3 until the points converge to local modes. Convergence occurs when the points no longer shift significantly.
5. Grouping of Converged Points: points converging to the same position are grouped into the same cluster. The final convergence position represents a mode in the feature space, and clusters correspond to high-density regions in the dataset.

The Mean Shift method does not require specifying the number of clusters a priori as k-means method, as clusters are formed adaptively based on data density. However, the efficiency of the algorithm may depend on the choice of parameters, such as the search distance or the weighting kernel. In resume, is useful when clusters have non-convex shapes and cannot be efficiently modeled with regular geometric shapes.

In the practice, it is implemented in the sklearn library for general data clustering purposes, although it can also be found in the OpenCV library.

The sklearn library (actually scikit) is the most commonly used library in Python for traditional machine learning algorithms and is used by many programs that also use OpenCV.

In the latter case (using OpenCV), we have two options: the meanshift method, which is often used for tracking (as we will see in the video topic), or pyrMeanShiftFiltering, which is used directly for color image segmentation:

```
final = cv.pyrMeanShiftFiltering(img, 25, 60)
```

In this case, the second parameter of the function (25) is the spatial window radius, and the third parameter (60) is the color window radius. The segmentation from this function is pyramidal, meaning it is performed at different resolutions, and the results are combined.

Other methods are briefly explained below:

- DBSCAN (Density-Based Spatial Clustering of Applications with Noise): groups of points into clusters based on the density of points in their neighborhood. It is effective for detecting clusters of arbitrary shapes and is robust to noise.
- Hierarchical Clustering: builds a hierarchy of clusters. It can be agglomerative, starting with individual clusters and merging them, or divisive, starting with a single cluster and splitting it. It can be useful for analyzing structures at different scales.
- Gaussian Mixture Models (GMM): models the data distribution as a combination of Gaussian distributions. It is useful when the data does not form clearly separated clusters and may be generated by multiple distributions.
- Spectral Clustering: uses the spectral information of the affinity graph between points to perform clustering. It is effective for detecting non-spherical clusters and can capture complex structures.
- Agglomerative Clustering: starts by considering each point as a cluster and iteratively merges the closest clusters. It can be effective for large datasets.
- Affinity Propagation: finds exemplars among the data and groups points around these exemplars. It is useful when looking for representative exemplars.

Additionally, preprocessing techniques are often applied to extract meaningful features before clustering, and the choice of clustering algorithm depends on the type of data and the specific characteristics of the task in computer vision. In conclusion, only the experimentation let us determine which method is most suitable for a particular case.

1.3.6 Graph Cut

Graph Cut Segmentation representing an image as a graph, where nodes correspond to pixels or regions, and edges represent relationships between them. The segmentation is achieved by finding a cut in the graph that minimizes an energy function. It is very effective when dealing with images containing objects with well-defined boundaries.

To achieve this segmentation, we have to follow some key steps:

1. Graph construction: represent the image as a graph, where each pixel or region is a node in the graph. Edges between nodes represent relationships such as spatial proximity or color similarity. The weights of the edges reflect the dissimilarity between nodes.

2. Energy function: define an energy function that incorporates both data terms and smoothness terms. The data term measures how well the segmentation fits the input data, and the smoothness term encourages spatially coherent segmentations.

3. Min-cut/Max-flow optimization: formulate the segmentation problem as a min-cut or max-flow optimization problem. The goal is to find a cut (set of edges to remove) that minimizes the energy function. Efficient algorithms like the Ford-Fulkerson algorithm or the Push-Relabel algorithm are commonly used for this optimization.

4. Segmentation result: the optimized cut divides the graph into two disjoint sets, corresponding to the segmented regions. Pixels on one side of the cut belong to one segment, and pixels on the other side belong to the other segment.

In OpenCV, we have *GrabCut*, as we can see in the following example: `https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html`

The *cv2.grabCut* function is used to perform Graph Cut segmentation. The user is expected to provide an initial rectangle around the object of interest. The function then iteratively updates the foreground and background models to refine the segmentation.

1.3.7 Saliency

Saliency methods are techniques that aim to identify and highlight the most visually significant or salient regions in an image or a video. These methods are inspired by the human visual system, which tends to focus on the most important and interesting parts of a scene. There are a lot of methods, but we highlight the following:

- Graph-based visual saliency (GBVS): computes saliency using low-level features such as color, intensity, and orientation. It employs a graph-based representation to model the relationships between image regions and computes a saliency map.

- Itti-Koch-Niebur saliency model: is based on the principles of human visual attention. It combines features such as color, intensity, and orientation to compute a saliency map. It includes a center-surround mechanism to highlight regions with high contrast

- Frequency-tuned saliency: focuses on the frequency domain and identifies salient regions based on frequency content. It considers that salient objects often have different frequency characteristics compared to the background.

- Region-based saliency: defines regions in an image and assigns saliency scores based on features such as color, texture, and shape. Regions with distinctive features are considered more salient.

- Motion saliency: in video processing, motion saliency methods identify regions that exhibit significant motion. These regions are considered more salient because they are likely to contain important objects or events.
- Objectness: aim to identify regions in an image that are likely to contain objects. These regions are considered salient because they have a high likelihood of being relevant to object recognition.
- Spectral Residual model: based on the observation that salient objects often lead to spectral residual in the frequency domain, this model calculates the spectral residual of an image to determine salient regions.
- Deep Learning-based saliency models: have been employed to learn saliency maps directly from data. Models like DeepGaze and SALICON use deep learning to predict where people look in images.

The choice of a saliency method depends on the specific requirements of the application.
We show the Spectral Residual mode in detail as an example of saliency approach.

This model focuses on identifying visually salient regions in an image based on the spectral residual in the frequency domain. The key idea behind this model is that salient objects or regions often lead to an apparent spectral residual. We can summarise the model in five basic steps:

1. Frequency domain transformation: convert the input image from the spatial domain to the frequency domain using Fourier transform.
2. Spectral residual computation: compute the spectral residual of the transformed image. The spectral residual is essentially the difference between the magnitude spectrum and a low-pass filtered version of the magnitude spectrum.
3. Inverse Fourier transform: apply the inverse Fourier transform to obtain an image in the spatial domain.
4. Saliency map generation: calculate the saliency map by combining the spectral residual information with color information. This could involve additional processing to enhance the saliency map.
5. Normalization and thresholding: normalize the saliency map to bring the values within a specific range and apply a threshold to obtain a binary saliency map. This map highlights the salient regions in the original image.

The model assumes that salient objects in an image will generate a noticeable spectral residual due to their unique frequency characteristics. Regions with distinctive visual content are expected to stand out in the spectral residual domain.

In OpenCV, we can implement the model as below:

```
import cv2
import numpy as np

def spectral_residual(img):
    # Convert image to the frequency domain using Fourier transform
    fft_img = np.fft.fft2(img)

    # Compute the magnitude spectrum
    magnitude_spectrum = np.abs(fft_img)

    # Compute the phase spectrum
    phase_spectrum = np.angle(fft_img)

    # Calculate the spectral residual
    spectral_residual = np.exp(np.log(magnitude_spectrum) - cv2.
                                    boxFilter(np.log(
                                    magnitude_spectrum), -1, (3,
                                    3)))

    # Reconstruct the image in the spatial domain
    reconstructed_img = np.fft.ifft2(spectral_residual * np.exp(1j
                                    * phase_spectrum))

    # Calculate the saliency map by combining spectral residual
                                    with color information
    saliency_map = np.abs(reconstructed_img)

    return saliency_map

# Read an image
image = cv2.imread('image.jpg')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Convert the image to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# Apply the Spectral Residual Model
saliency_map = spectral_residual(gray_image)

# Normalize and threshold the saliency map
saliency_map = cv2.normalize(saliency_map, None, alpha=0, beta=255,
                                 norm_type=cv2.NORM_MINMAX)
_, saliency_binary = cv2.threshold(saliency_map.astype(np.uint8),
                                100, 255, cv2.THRESH_BINARY)

# Display the original image and the saliency map
plt.figure(figsize=(12, 6))

plt.subplot(1, 3, 1)
plt.imshow(image)
plt.title('Original Image')

plt.subplot(1, 3, 2)
plt.imshow(gray_image, cmap='gray')
plt.title('Grayscale Image')

plt.subplot(1, 3, 3)
```

```
plt.imshow(saliency_binary, cmap='gray')
plt.title('Saliency Map')

plt.show()
```