Computer Vision

3.- Image Processing

1 Image Processing: Transformations

In this topic, we will start modifying images through various types of transformations.

1.1 Basic Transformations and Arithmetic operations

As we've seen before, in OpenCV, you can perform direct matrix operations using the NumPy library. For example, you can multiply all pixels in an image by 4 as follows:

```
dst = src * 4
```

As you can see, arithmetic operations can use either numbers or arrays/matrices interchangeably.

In addition to basic arithmetic operations (addition, subtraction, multiplication, and division), you can also use AND, OR, XOR, and NOT through the following NumPy functions:

```
# Perform arithmetic transformations
# Multiply all pixel values by 1.5
transformed_multiply = cv2.multiply(img, 1.5)

# Add a constant value (100) to all pixel values
transformed_add = cv2.add(img, 100)

# Subtract a constant value (50) from all pixel values
transformed_subtract = cv2.subtract(img, 50)

# Invert the pixel values
transformed_invert = cv2.bitwise_not(img)
```

For example, to invert an image (transform white to black and black to white), you can use the `bitwise_not instruction`, which is an alias of np.invert.

Histogram equalization in grayscale is straightforward with the equalizeHist function:

```
equ = cv.equalizeHist(img)
```

Thresholding is a simple and effective method for image segmentation. The basic idea is to convert an image into a binary image, where pixels are classified as either foreground or background based on their intensity values. The process involves setting a threshold value, and pixels with intensities above the threshold are set to one value (e.g., white), while those below the threshold are set to another value (e.g., black).

In OpenCV, the threshold function is commonly used for thresholding:

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

# Read the image in grayscale
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Set a threshold value (e.g., 128)
threshold_value = 128

# Apply binary thresholding
_, binary_image = cv2.threshold(image, threshold_value, 255, cv2.
                                  THRESH_BINARY)

# Display the original and binary images using Matplotlib
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(binary_image, cmap='gray')
plt.title('Binary Image')
plt.axis('off')

plt.show()
```

You can also threshold a grayscale image using the threshold function, resulting in a binary image (also called a mask) that can highlight relevant information for a specific task. Thresholding involves setting pixels with values below the specified threshold to 0 and those above it to 255, and it's the most basic form of segmentation (as we'll see in detail in Segmentation section):

```python
# Set pixels below 128 to 0 and pixels above 128 to 255
th, dst = cv.threshold(src, 128, 255, cv.THRESH_BINARY)
```

The last parameter is the thresholding type. In OpenCV, there are 5 types of thresholding:

- `cv.THRESH_BINARY`
- `cv.THRESH_BINARY_INV`
- `cv.THRESH_TRUNC`

- `cv.THRESH_TOZERO`
- `cv.THRESH_TOZERO_INV`

For more information, see Segmentation section.

This method only works with grayscale images. For thresholding color images, OpenCV provides the inRange function. Given a 3-channel image, this function returns another single-channel image with pixels in a certain range colored white and others black. Therefore, it can be used for basic color segmentation, as we'll see in detail in Segmentation Section.

```
# Set pixels to white if they are between (0,10,20) and (40,40,51)
dst = cv.inRange(src, (0, 10, 20), (40, 40, 51))
```

In OpenCV, there are alternative binarization techniques such as adaptive thresholding or the Otsu method, which we'll explore in the segmentation topic as they consider the intensity values of neighboring pixels and can't be considered basic transformations.

1.2 Global Transformations

One of the most commonly used global transformations in image processing is the Fourier transform.
The Fourier Transform is a mathematical technique used in signal processing and image analysis. In computer vision, the Fourier Transform is often applied to analyze the frequency content of images. The transformation represents an image in terms of its frequency components rather than its pixel values.

In OpenCV, you can use the 'dft' function to calculate this transform. However, some preprocessing is required to prepare the input for this function, and postprocessing is needed to calculate the magnitude and phase from its result. In Computer Vision, we won't go into details on how to use the Fourier transform in OpenCV, but if you want to learn more, you can check this link`https://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html#dft`.

1.3 Affine Transformations

In OpenCV, most geometric transformations are implemented by creating a transformation matrix and applying it to the original image using 'warpAffine'.

This function requires a 2x3 matrix as input because it implements affine transformations using augmented matrices. As we've seen in theory, the last row of the augmented matrix in an affine transformation is always (0,0,1), so there's no need to specify it (which is why a 2x3 matrix is indicated instead of 3x3).

The 'warpAffine' function also has parameters to indicate the interpolation type (flags) and behavior at the edges, as shown in its documentation `https://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.`

`html#void%20warpAffine(InputArray%20src,%20OutputArray%20dst,%20InputArray%20M,%20Size%20dsize,%20int%20flags,%20int%20borderMode,%20const%20Scalar&%20borderValue).`

In general, you can use 'warpAffine' to implement any affine transformation. For example, you can implement the translation, that refers to shifting an object or image from one location to another along a specified direction: [[1, 0, tx], [0, 1, ty], [0, 0, 1]]

with this code:

```
import cv2 as cv
import numpy as np

img = cv.imread('img.jpg', cv.IMREAD_GRAYSCALE)

# Translation values
tx = 100
ty = 50

# Define the matrix
M = np.float32([[1, 0, tx],
                [0, 1, ty]])

# The flags parameter can be omitted; by default, it is
                                  INTER_LINEAR
rows, cols = img.shape
dst = cv.warpAffine(img, M, (cols, rows), flags=cv.INTER_CUBIC)

#Show the translation
plt.imshow(dst, cmap='gray')
plt.title('Translation')
plt.axis('off')

plt.show()
```

Alternatively to using affine transformation matrices with warpAffine, there are specific functions to help manage rotation, reflection, and scaling, as we'll see next.

Rotation

Rotation involves turning an object or image around a specified point or axis.
Rotation by an angle is defined by the following transformation matrix:

[[cos θ, -sin θ, 0], [sin θ, cos θ, 0], [0, 0, 1]]

However, OpenCV also allows rotation by specifying an adjustable rotation center, enabling the use of any reference point as the axis. For this, the getRotationMatrix2D function is used, taking the rotation center as its first parameter:

```
rows, cols = img.shape
# Get the rotation matrix with 90 degrees using the center of the
                                  image as a reference
```

```
M = cv.getRotationMatrix2D((cols/2, rows/2), 90, 1)  # The last
                                    parameter (1) is the scale
dst = cv.warpAffine(img, M, (cols, rows))
```

Reflection

Reflection in computer vision refers to the transformation that flips an image or object horizontally, vertically, or both. It's a geometric operation that produces a mirror image of the original. Reflection is commonly used in computer vision for various purposes, such as data augmentation, image processing, and computer graphics.
In OpenCV, there is a specific function (flip) that implements reflection without the need for warpAffine.

```
flipVertical = cv.flip(img, 0)
```

The third parameter of flip can be 0 (reflection over the x-axis), positive (e.g., 1 is reflection over the y-axis), or negative (e.g., -1 is over both axes).

Scaling

Scaling in computer vision refers to the transformation that changes the size of an image or object. It involves resizing the image by either enlarging or reducing its dimensions. Scaling is a common operation in image processing, computer vision, and graphics.

OpenCV provides functions for scaling images, and there are various ways to perform scaling based on specific requirements. Scaling is also implemented using a specific function called resize, allowing you to specify specific dimensions or a ratio between the source and destination images.

```

# 1- Specifying a specific size (in this example, 20x30):
dim = (20, 30)
dst = cv.resize(src, dim, interpolation=cv.INTER_LINEAR)  # The
                                    last interpolation parameter is
                                    optional

# 2- Specifying a scale, for example, 75% of the original image:
dst = cv.resize(src, (0, 0), fx=0.75, fy=0.75, cv.INTER_LINEAR)  #
                                    The last interpolation parameter
                                    is optional
```

1.4 Projective Transformations

Projective transformations, also known as perspective transformations, are used in computer vision to represent the transformation of 3D scenes onto a 2D plane, considering the effects of perspective. These transformations are essential for tasks such as image stitching, object recognition, and augmented reality.

Projective transformation is not affine, so it does not preserve the parallelism of lines in the original image.

To perform a projective transformation, you need to specify a 3x3 matrix and use the warpPerspective function. For example:

```python
# Define the matrix
M = np.float32([[1, 0, 0],
                [0.5, 1, 0],
                [0.2, 0, 1]])

# Implement the projective transformation
rows, cols = img.shape
dst = cv.warpPerspective(img, M, (cols, rows))
```

The complete list of parameters for this function can be found here `https://docs.opencv.org/4.5.2/da/d54/group__imgproc__transform.html#gaf73673a7e8e18ec6963e3774e6a94b`

There's also a practical alternative for implementing such a transformation, as it's often challenging to estimate the matrix values beforehand for a specific transformation. This alternative involves providing two arrays of 4 points each (each point being a two-dimensional vector representing its coordinates in the XY plane): The first array is for the original image, and the second contains the projection of those points (where they will end up) in the destination image. With this data, you can use getPerspectiveTransform to calculate the transformation matrix values.

```python
# The two parameters that getPerspectiveTransform receives must be
                                  arrays of points, and each point
                                  is a two-element float array.
M = cv.getPerspectiveTransform(input_pts, output_pts)

# Apply the transformation using linear interpolation. The values
                                  widthDst and heightDst indicate
                                  the size of the destination image
                                  .
dst = cv.warpPerspective(src, M, (widthDst, heightDst), flags=cv.
                                  INTER_LINEAR)
```

Here's an example of an `input_pts` vector:

```python
input_pts = np.float32([[120, 13], [610, 24], [2, 491], [622, 500]]
                                  )
```

1.5 Neighborhood-based Transformations

In this section, we'll see how to implement neighborhood-based transformations using OpenCV, particularly convolution and median filters.

Convolution Filters

Convolution filters are a fundamental concept in image processing and computer vision. They are widely used for tasks such as blurring, sharpening, edge detection, and more. A convolution filter, also known as a kernel or mask, is a

small matrix applied to an image to perform operations like convolution.
Convolutions are implemented with the filter2D function.
This function takes the following parameters:

- src: Input image
- ddepth: Radiometric resolution (depth) of the dst matrix. A negative value indicates that the resolution is the same as that of the input image.
- kernel: The kernel to convolve with the image.
- anchor (optional): The anchor position of the kernel (as shown in the figure) relative to its origin. The point (-1,-1) indicates the center of the kernel (it's the default value).
- delta (optional): A value to add to each pixel during convolution. By default, it's 0.
- borderType (optional): The method to follow at the edges of the image for interpolation, as the filter goes out of the image at these points. It can be `cv.BORDER_REPLICATE`, `cv.BORDER_REFLECT`, `cv.BORDER_REFLECT_101`, `cv.BORDER_WRAP`, `cv.BORDER_CONSTANT`, or `cv.BORDER_DEFAULT` (which is the default value).

Examples of calls to this function:

```
# This form is the most common
dst = cv.filter2D(src, -1, kernel)
# Indicating what to do at the edges
dst = cv.filter2D(src, -1, kernel, borderType=cv.BORDER_CONSTANT)
```

You need to create a kernel before convolving it with the image. For example, it could be the following:

```
kernel = np.array([[-1, -1, -1, -1, -1],
                   [-1, -1, -1, -1, -1],
                   [-1, -1, 24, -1, -1],
                   [-1, -1, -1, -1, -1],
                   [-1, -1, -1, -1, -1]])
```

Question: What type of filter have we just created?

Median Filter

A median filter is a non-linear digital signal processing filter used to remove noise from an image or signal. Unlike linear filters, such as the mean filter, which replaces each pixel's value with the average value of its neighborhood, the median filter replaces each pixel's value with the median value of its neighborhood.

The median filter is implemented very simply in OpenCV:

```
dst = cv.medianBlur(src, 5)
```

The last parameter indicates the size of the kernel, which is always square (in this example, 5x5 pixels).

1.6 Morphological Transformations

Morphological transformations are operations based on the shape or morphology of an image. These operations are typically used for processing binary or grayscale images and involve the modification or analysis of structures within the image.

In OpenCV, morphological transformations are commonly applied using functions like cv2.erode, cv2.dilate, cv2.morphologyEx, and others.

Erosion and Dilation

Erosion is a basic morphological operation that involves the shrinking of the object in the binary image.

Dilation is a morphological operation that expands the boundaries of the object in the image.

The syntax for these basic morphological operations is simple:

```
dst = cv.erode(src, element)
dst = cv.dilate(src, element)
```

Both functions need a structuring element, called element in the code. As with filter2D, optional parameters anchor, delta, and borderType can also be added.

To create the structuring element, the getStructuringElement function is used:

```
# Shape of the filter
erosion_type = cv.MORPH_ELLIPSE

# The last parameter is the size of the filter, in this case, 5x5
element = cv.getStructuringElement(erosion_type, (5,5))
```

The structuring element can be shaped as a box (`MORPH_RECT`), cross (`MORPH_CROSS`), or ellipse (`MORPH_ELLIPSE`).

Opening and Closing

Opening is an erosion followed by dilation. It is useful for removing noise. And Closing is a dilation followed by erosion. It is useful for closing small holes.

The rest of morphological transformation functions are implemented using the morphologyEx function:

```
dst = cv.morphologyEx(src, cv.MORPH_OPEN, element)
```

This function is called with the same parameters as erode or dilate, adding a parameter indicating the type of operation:

- Opening: `MORPH_OPEN`
- Closing: `MORPH_CLOSE`
- Gradient: `MORPH_GRADIENT`
- White Top Hat: `MORPH_TOPHAT`
- Black Top Hat: `MORPH_BLACKHAT`

We can find example code for implementing an interface to test these operations by modifying their parameters in this link `https://docs.opencv.org/4.5.2/d3/dbe/tutorial_opening_closing_hats.html`.

Top-Hat

The top-hat transform is another morphological operation that is particularly useful for enhancing bright structures on a dark background or vice versa.

In OpenCV, the function is:

```
tophat_result = cv2.morphologyEx(image,cv2.MORPH_TOPHAT, kernel)
```