Image enhancement

**Negative image**

In image processing, a negative image (also known as an inverse image) is created by subtracting the pixel values of an image from the maximum pixel value that the image can have. This operation results in an image where dark pixels become light and light pixels become dark.

To create a negative image using OpenCV in Python, you can use the following code

import cv2

# Load the image

img = cv2.imread('image.jpg')

# Convert the image to grayscale

gray\_img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# Invert the grayscale image

neg\_img = cv2.bitwise\_not(gray\_img)

# Display the original and negative images side by side

cv2.imshow('Original Image', img)

cv2.imshow('Negative Image', neg\_img)

cv2.waitKey(0)

cv2.destroyAllWindows()

In this code, we first load the image using the **cv2.imread** function. We then convert the image to grayscale using the **cv2.cvtColor** function. Next, we use the **cv2.bitwise\_not** function to invert the grayscale image and create the negative image. Finally, we display both the original and negative images using the **cv2.imshow** function.

Note that the **cv2.bitwise\_not** function can also be used to create a negative image directly from a color image, without first converting it to grayscale.

Here's a basic example of creating a negative image using OpenCV in Python

import cv2

# Load the image

img = cv2.imread('image.jpg')

# Invert the image

neg\_img = 255 - img

# Display the original and negative images side by side

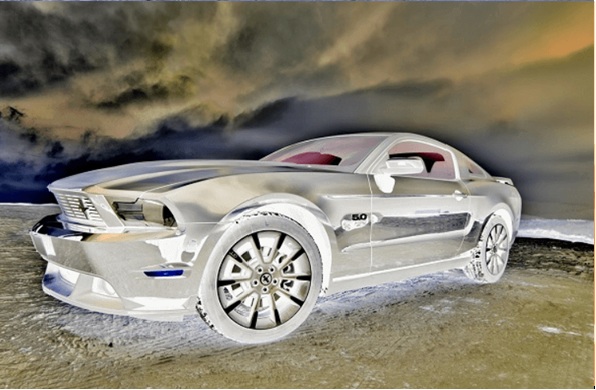
cv2.imshow('Original Image', img)

cv2.imshow('Negative Image', neg\_img)

cv2.waitKey(0)

cv2.destroyAllWindows()





In this example, we first load the image using the **cv2.imread** function. We then invert the image by subtracting each pixel value from 255 (which is the maximum value that a pixel can have). This creates the negative image, where dark pixels become light and light pixels become dark. Finally, we display both the original and negative images using the **cv2.imshow** function.

Note that this example assumes that the input image is a color image (i.e., has three color channels: blue, green, and red). If the input image is a grayscale image (i.e., has only one color channel), you can use the following code to create the negative image:

import cv2

# Load the image

img = cv2.imread('image.jpg', cv2.IMREAD\_GRAYSCALE)

# Invert the grayscale image

neg\_img = 255 - img

# Display the original and negative images side by side

cv2.imshow('Original Image', img)

cv2.imshow('Negative Image', neg\_img)

cv2.waitKey(0)

cv2.destroyAllWindows()

In this code, we load the image using the **cv2.imread** function and specify the **cv2.IMREAD\_GRAYSCALE** flag to read the image as a grayscale image. We then invert the grayscale image using the same method as before, and display both the original and negative images using the cv2.imshow function.





**Power-Law (Gamma) Transformation**

Power-Law (Gamma) Transformations are a type of image processing technique that involve applying a non-linear transformation to the pixel values of an image. The transformation is based on a power-law function that raises the pixel values to a specific power, which can be controlled by the user. This can be expressed mathematically as:

V\_out = V\_in^gamma

where V\_in and V\_out are the input and output pixel values, respectively, and gamma is the power value.

The gamma value determines the amount of contrast and brightness that is added to the image. A gamma value of 1.0 results in no change to the image, while gamma values greater than 1.0 will darken the image and increase the contrast, and gamma values less than 1.0 will lighten the image and decrease the contrast.

Here's an example of how to apply a gamma transformation to an image using OpenCV in Python

import cv2

import numpy as np

# Load the image

img = cv2.imread('image.jpg')

# Define the gamma value

gamma = 0.5

# Apply the gamma transformation

gamma\_img = np.power(img/255, gamma) \* 255

# Convert the image to uint8 data type

gamma\_img = np.uint8(gamma\_img)

# Display the original and gamma-transformed images side by side

cv2.imshow('Original Image', img)

cv2.imshow('Gamma-Transformed Image', gamma\_img)

cv2.waitKey(0)

cv2.destroyAllWindows()

In this code, we load the image using the **cv2.imread** function, define the gamma value as 0.5, and then apply the gamma transformation using the **np.power** function. We normalize the pixel values to the range of 0 to 1 by dividing by 255, and then multiply them by 255 again to bring them back to the range of 0 to 255. Finally, we convert the pixel values to the **uint8** data type and display both the original and gamma-transformed images using the **cv2.imshow** function.





**Logarithmic Transformations**

Logarithmic Transformations are a type of image processing technique that involve applying a logarithmic function to the pixel values of an image. This transformation can be used to enhance the contrast of images with low intensity values, such as images with mostly dark or shadowy areas.

The logarithmic transformation is given by the following equation:

V\_out = c \* log(1 + V\_in)

where V\_in and V\_out are the input and output pixel values, respectively, and c is a constant that controls the scaling of the output values.

Here's an example of how to apply a logarithmic transformation to an image using OpenCV in Python:

import cv2

import numpy as np

# Load the image

img = cv2.imread('image.jpg')

# Define the constant value

c = 255 / np.log(1 + np.max(img))

# Apply the logarithmic transformation

log\_img = c \* np.log(1 + img)

# Convert the image to uint8 data type

log\_img = np.uint8(log\_img)

# Display the original and logarithmically transformed images side by side

cv2.imshow('Original Image', img)

cv2.imshow('Logarithmic-Transformed Image', log\_img)

cv2.waitKey(0)

cv2.destroyAllWindows()



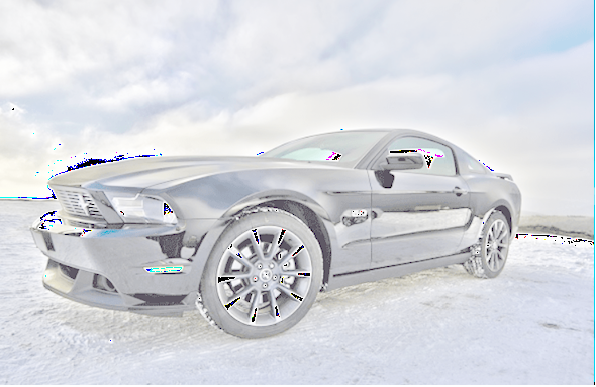


Image channel access

In image processing, an image channel refers to a single color channel of an image. For example, a color image can be decomposed into its red, green, and blue (RGB) channels, where each channel represents the contribution of that color to the overall image. In OpenCV, the color channels of an image can be accessed and manipulated separately using indexing.

Here's an example of how to access the color channels of an image using OpenCV in Python:

**import cv2**

**import numpy as np**

**# Load the color image**

**img = cv2.imread('image.jpg')**

**# Split the image into its color channels**

**b, g, r = cv2.split(img)**

**# Display the individual color channels**

**cv2.imshow('Blue Channel', b)**

**cv2.imshow('Green Channel', g)**

**cv2.imshow('Red Channel', r)**

**# Merge the color channels back into an image**

**merged = cv2.merge([b, g, r])**

**cv2.imshow('Merged Image', merged)**

**cv2.waitKey(0)**

**cv2.destroyAllWindows()**









In this code, we load a color image using the **cv2.imread** function, and then use the **cv2.split** function to split the image into its blue, green, and red color channels. We display each individual color channel using the **cv2.imshow** function, and then use the **cv2.merge** function to merge the color channels back into a single image. Finally, we display the merged image using the **cv2.imshow** function.

Note that each color channel is a grayscale image, and can be processed or manipulated separately from the other channels. This can be useful for tasks such as color correction or object detection, where certain colors may need to be emphasized or suppressed.

there are other ways to access and manipulate the color channels of an image in OpenCV. One common technique is to use NumPy array indexing to directly access and modify the pixel values of each color channel.

Here's an example of how to access the color channels of an image using NumPy array indexing in OpenCV:

**import cv2**

**import numpy as np**

**# Load the color image**

**img = cv2.imread('image.jpg')**

**# Access the blue color channel using array indexing**

**b = img[:, :, 0]**

**# Access the green color channel using array indexing**

**g = img[:, :, 1]**

**# Access the red color channel using array indexing**

**r = img[:, :, 2]**

**# Display the individual color channels**

**cv2.imshow('Blue Channel', b)**

**cv2.imshow('Green Channel', g)**

**cv2.imshow('Red Channel', r)**

**# Modify the pixel values of the blue channel**

**b[:, :] = 0**

**# Merge the modified color channels back into an image**

**merged = cv2.merge([b, g, r])**

**cv2.imshow('Merged Image', merged)**

**cv2.waitKey(0)**

cv2.destroyAllWindows()

In this code, we use NumPy array indexing to access the blue, green, and red color channels of the image. We then display each individual color channel using the **cv2.imshow** function. We can also modify the pixel values of each color channel directly using NumPy array indexing. In this example, we set all the pixel values of the blue channel to zero, effectively removing the blue color from the image. Finally, we merge the modified color channels back into a single image using the **cv2.merge** function, and display the merged image using the **cv2.imshow** function.

**Spatial Filtering (Neighborhood Processing)**

Spatial filtering, also known as neighborhood processing, is a technique in image processing where the value of each pixel in an image is replaced by a weighted average of its neighboring pixels. This can be useful for tasks such as noise reduction, edge detection, and image sharpening.

In OpenCV, spatial filtering can be implemented using the **cv2.filter2D** function, which performs a convolution operation between the image and a kernel. The kernel is a small matrix that defines the weights for the weighted average operation. By adjusting the values of the kernel, we can control the behavior of the spatial filter.

Here's an example of how to apply a spatial filter to an image using OpenCV in Python:

**import cv2**

**import numpy as np**

**# Load the grayscale image**

**img = cv2.imread('image.jpg', 0)**

**# Define the kernel for the spatial filter**

**kernel = np.array([**

**[-1, -1, -1],**

**[-1, 8, -1],**

**[-1, -1, -1]])**

**# Apply the spatial filter using cv2.filter2D**

**filtered\_img = cv2.filter2D(img, -1, kernel)**

**# Display the original and filtered images side by side**

**cv2.imshow('Original Image', img)**

**cv2.imshow('Filtered Image', filtered\_img)**

**cv2.waitKey(0)**

**cv2.destroyAllWindows()**

In this code, we first load a grayscale image using the **cv2.imread** function. We then define a 3x3 kernel that performs edge detection by subtracting the average of the neighboring pixels from the center pixel. We apply this kernel to the image using the **cv2.filter2D** function, and display the original and filtered images side by side using the **cv2.imshow** function.

Note that the **cv2.filter2D** function applies the kernel to each pixel in the image, including the pixels on the edge of the image. This can result in artifacts or invalid values at the edges of the image. To avoid this, we can use other filtering functions in OpenCV such as **cv2.GaussianBlur** or **cv2.medianBlur**, which apply a filter that takes into account only the valid pixels in the image.





**Smoothing Spatial Filters**

Smoothing spatial filters, also known as blurring filters, are a type of neighborhood processing technique in image processing. The goal of these filters is to remove high-frequency noise from an image by averaging the pixel values in the local neighborhood. This results in a smoother image that can be easier to process for tasks such as edge detection or segmentation.

In OpenCV, there are several functions that can be used to apply smoothing spatial filters to an image, including **cv2.blur**, **cv2.GaussianBlur**, and **cv2.medianBlur**.

Here's an example of how to apply a smoothing filter to an image using OpenCV in Python

import cv2

# Load the image

img = cv2.imread('image.jpg')

# Apply a Gaussian smoothing filter with a 5x5 kernel

filtered\_img = cv2.GaussianBlur(img, (5, 5), 0)

# Display the original and filtered images side by side

cv2.imshow('Original Image', img)

cv2.imshow('Filtered Image', filtered\_img)

cv2.waitKey(0)

cv2.destroyAllWindows()

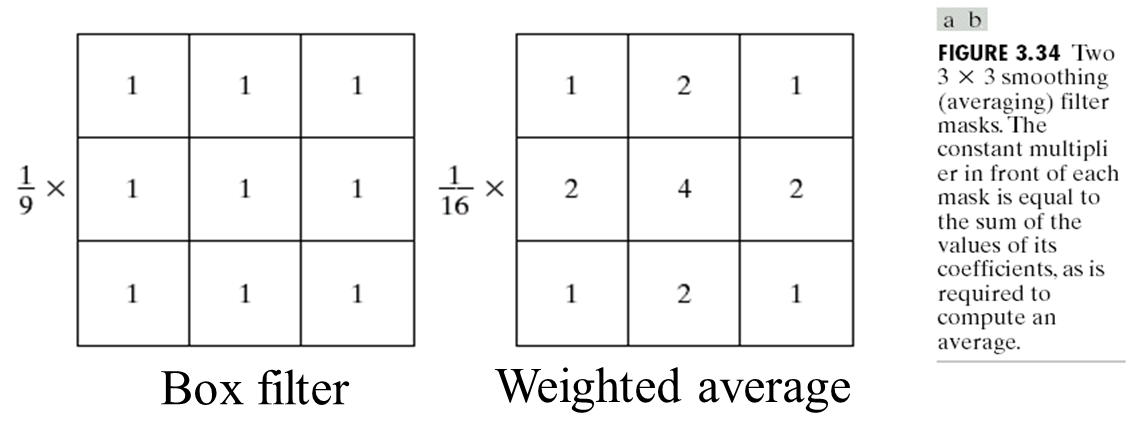
In this code, we first load an image using the **cv2.imread** function. We then apply a Gaussian smoothing filter with a 5x5 kernel using the **cv2.GaussianBlur** function. The **0** argument specifies that the standard deviation of the Gaussian kernel should be calculated based on the kernel size. We display the original and filtered images side by side using the **cv2.imshow** function.

Other smoothing filters can be applied in a similar way using the appropriate OpenCV function. The **cv2.blur** function applies a simple averaging filter, while the **cv2.medianBlur**

function applies a median filter, which can be useful for removing salt-and-pepper noise. The choice of filter depends on the specific requirements of the image processing task.







import cv2

# Load the image

img = cv2.imread('image.jpg')

# Apply a Gaussian smoothing filter with a 5x5 kernel

filtered\_img = cv2.GaussianBlur(img, (5, 5), 1/9)

# Display the original and filtered images side by side

cv2.imshow('Original Image', img)

cv2.imshow('Filtered Image', filtered\_img)

cv2.waitKey(0)

cv2.destroyAllWindows()