Image Processing Techniques Using OpenCV

Introduction to Image Transformations

Image transformations are fundamental operations in image processing that allow us to manipulate images for various purposes such as enhancing visual quality, extracting features, or preparing data for machine learning models. These transformations can be categorized into affine and non-affine transformations.

- Affine Transformation: A linear mapping method that preserves points, straight lines, and planes. It includes operations like translation, rotation, scaling, and shearing.
- Non-Affine Transformation: Non-linear transformations that distort the image in ways that do not preserve straight lines, such as perspective warping.

-Syntax

OpenCV provides several functions for applying transformations:

```
import cv2
import numpy as np
# Affine Transformation
M = np.float32([[1, 0, tx], [0, 1, ty]])  # Translation matrix
translated_image = cv2.warpAffine(image, M, (width, height))
# Rotation
M = cv2.getRotationMatrix2D(center, angle, scale)
rotated_image = cv2.warpAffine(image, M, (width, height))
```

Explanation:

- `cv2.warpAffine`: Applies an affine transformation to an image.
- `cv2.getRotationMatrix2D`: Computes the rotation matrix for a given center, angle, and scale.

Image Translations

Image translation involves moving an image up, down, left, or right without altering its size or orientation.

Translation shifts every pixel in the image by a specified amount along the x-axis (horizontal) and y-axis (vertical).

-Syntax

```
import cv2
import numpy as np
# Define translation matrix
M = np.float32([[1, 0, tx], [0, 1, ty]])  # tx: horizontal shift, ty: vertical shift
translated_image = cv2.warpAffine(image, M, (width, height))
```

-Example

```
image = cv2.imread('example.jpg')
height, width = image.shape[:2]
tx, ty = 50, 30  # Move 50 pixels right and 30 pixels down
M = np.float32([[1, 0, tx], [0, 1, ty]])
translated_image = cv2.warpAffine(image, M, (width, height))
cv2.imshow('Translated Image', translated_image)
cv2.waitKey(0)
```

-Explanation

The translation matrix `M` specifies how much to shift the image. Positive values of `tx` and `ty` move the image to the right and down, respectively.

Rotations and Horizontal Flipping

Rotating an image involves spinning it around a point, while flipping mirrors the image horizontally or vertically.

- Rotation: Spins the image around a specified center point by a given angle.
- Flipping: Reverses the image along the horizontal or vertical axis.

-Syntax

```
# Rotation
M = cv2.getRotationMatrix2D(center, angle, scale)
rotated_image = cv2.warpAffine(image, M, (width, height))

# Flipping
flipped_image = cv2.flip(image, flip_code)
```

-Example

```
image = cv2.imread('example.jpg')
height, width = image.shape[:2]

# Rotate 90 degrees clockwise
center = (width // 2, height // 2)
M = cv2.getRotationMatrix2D(center, -90, 1)
rotated_image = cv2.warpAffine(image, M, (width, height))

# Horizontal Flip
flipped_image = cv2.flip(image, 1)
cv2.imshow('Rotated Image', rotated_image)
cv2.imshow('Flipped Image', flipped_image)
cv2.waitKey(0)
```

Scaling, Resizing, and Interpolation

Scaling changes the size of an image, while interpolation determines how missing pixel values are calculated during resizing.

- Scaling: Adjusts the dimensions of an image (e.g., enlarging or shrinking).

- Interpolation: Estimates new pixel values when an image is resized.

-Syntax

resized_image = cv2.resize(image, (new_width, new_height), interpolation=cv2.INTER_LINEAR)

-Example

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

 # Resize to half the original size

resized_image = cv2.resize(image, None, fx=0.5, fy=0.5, interpolation=cv2.INTER_AREA)

cv2.imshow('Resized Image', resized_image)

cv2.waitKey(0)

-Explanation

- `cv2.INTER_LINEAR`: Suitable for shrinking images.
- `cv2.INTER_CUBIC`: Better for enlarging but slower.
- `cv2.INTER_AREA`: Best for shrinking.

Image Pyramids

Image pyramids represent an image at multiple resolutions, useful for tasks like object detection.

Image pyramids reduce or expand an image's resolution iteratively.

-Syntax

 # Gaussian Pyramid

lower_res = cv2.pyrDown(image)

higher_res = cv2.pyrUp(lower_res)

-Example

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

 # Downscale using Gaussian Pyramid

lower_res = cv2.pyrDown(image)

 # Upscale back

higher_res = cv2.pyrUp(lower_res)

cv2.imshow('Lower Resolution', lower_res)

cv2.imshow('Higher Resolution', higher_res)

cv2.waitKey(0)

Cropping

Cropping extracts a specific region of interest (ROI) from an image.

Cropping selects a rectangular portion of an image by specifying pixel ranges.

-Syntax

cropped_image = image[y1:y2, x1:x2]

-Example

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

 # Crop a region

cropped_image = image[50:200, 100:300] # y1:y2, x1:x2

```
cv2.imshow('Cropped Image', cropped_image)
cv2.waitKey(0)
```

Arithmetic and Bitwise Operations

Arithmetic and bitwise operations manipulate pixel values to adjust brightness or create masks.

-Arithmetic Operations: Add/subtract scalar values to brighten/darken images.

- Bitwise Operations: Perform logical operations on binary masks.

-Syntax

```
# Brightening/Darkening
brightened_image = cv2.add(image, scalar_value)
darkened_image = cv2.subtract(image, scalar_value)

# Bitwise AND
result = cv2.bitwise_and(image1, image2, mask=mask)
```

-Example

```
image = cv2.imread('example.jpg')
# Brighten by adding 50 to each pixel
brightened_image = cv2.add(image, 50)
# Create a mask and apply bitwise AND
mask = np.zeros(image.shape[:2], dtype=np.uint8)
mask[50:200, 100:300] = 255
masked_image = cv2.bitwise_and(image, image, mask=mask)
cv2.imshow('Brightened Image', brightened_image)
cv2.imshow('Masked Image', masked_image)
cv2.waitKey(0)
```

Blurring - The Many Ways We Can Blur Images & Why It's Important

Blurring reduces image noise and detail by averaging pixel values in a neighborhood. It is crucial for preprocessing tasks like edge detection, noise removal, and reducing high-frequency details.

-Types of Blurring Techniques

1. Averaging Blur: Uses a normalized box filter to average pixel values.
2. Gaussian Blur: Applies a weighted average with a Gaussian kernel.
3. Median Blur: Replaces each pixel with the median value of its neighbors.
4. Bilateral Filtering: Preserves edges while blurring.

-Syntax

```
# Averaging Blur
blurred_image = cv2.blur(image, (kernel_size, kernel_size))

# Gaussian Blur
blurred_image = cv2.GaussianBlur(image, (kernel_size, kernel_size), sigmaX)

# Median Blur
blurred_image = cv2.medianBlur(image, kernel_size)

# Bilateral Filtering
blurred_image = cv2.bilateralFilter(image, d, sigmaColor, sigmaSpace)
```

-Example

```
import cv2
image = cv2.imread('example.jpg')
```

```
# Averaging Blur
averaged_blur = cv2.blur(image, (5, 5))

# Gaussian Blur
gaussian_blur = cv2.GaussianBlur(image, (5, 5), 0)

# Median Blur
median_blur = cv2.medianBlur(image, 5)

# Bilateral Filtering
bilateral_blur = cv2.bilateralFilter(image, 9, 75, 75)

cv2.imshow('Averaged Blur', averaged_blur)
cv2.imshow('Gaussian Blur', gaussian_blur)
cv2.imshow('Median Blur', median_blur)
cv2.imshow('Bilateral Blur', bilateral_blur)
cv2.waitKey(0)
```

-Explanation

-`cv2.blur`: Simple averaging of pixel values.

-`cv2.GaussianBlur`: Weighted average with a Gaussian kernel, ideal for smoothing.

-`cv2.medianBlur`: Removes salt-and-pepper noise effectively.

-`cv2.bilateralFilter`: Smooths while preserving edges, useful for beautification.

Sharpening - Reverse Your Image Blurs

Sharpening enhances edges and fine details in an image, often reversing the effects of blurring. It increases contrast between adjacent pixels.

-Technique

Sharpening is achieved using a **kernel-based convolution** operation. A common sharpening kernel is:

[0, -1, 0]

[-1, 5, -1]

[0, -1, 0]

-Syntax

```
sharpened_image = cv2.filter2D(image, -1, kernel)
```

-Example

```
import cv2
import numpy as np

image = cv2.imread('example.jpg')
# Define a sharpening kernel
kernel = np.array([[0, -1, 0],
                   [-1, 5, -1],
                   [0, -1, 0]])

# Apply sharpening
sharpened_image = cv2.filter2D(image, -1, kernel)
cv2.imshow('Original Image', image)
cv2.imshow('Sharpened Image', sharpened_image)
cv2.waitKey(0)
```

-Explanation

- `cv2.filter2D`: Applies a custom convolution kernel to the image.
- The kernel emphasizes edges by increasing pixel intensity differences.

Thresholding (Binarization) - Making Certain Image Areas Black or White

Thresholding converts a grayscale image into a binary image by setting pixel values above or below a threshold to either black (0) or white (255). It is widely used in segmentation and object detection.

-Types of Thresholding

1. Simple Thresholding: Fixed threshold value.
2. Adaptive Thresholding: Dynamically calculates thresholds based on local regions.
3. Otsu's Thresholding: Automatically determines the optimal threshold for bimodal images.

-Syntax

```
# Simple Thresholding
_, binary_image = cv2.threshold(gray_image, threshold_value, max_value, threshold_type)

# Adaptive Thresholding
binary_image = cv2.adaptiveThreshold(gray_image, max_value, adaptive_method, threshold_type, block_size, C)

# Otsu's Thresholding
_, binary_image = cv2.threshold(gray_image, 0, max_value, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

-Example

```
import cv2
image = cv2.imread('example.jpg', cv2.IMREAD_GRAYSCALE)

# Simple Thresholding
_, simple_thresh = cv2.threshold(image, 127, 255, cv2.THRESH_BINARY)
```

 # Adaptive Thresholding

adaptive_thresh = cv2.adaptiveThreshold(image, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 11, 2)

cv2.imshow('Simple Thresholding', simple_thresh)

cv2.imshow('Adaptive Thresholding', adaptive_thresh)

cv2.waitKey(0)

-Explanation

-Simple Thresholding: Converts pixels above a fixed threshold to white and others to black.

- Adaptive Thresholding: Useful for uneven lighting conditions.

Github Repo with all image processing code examples:

https://github.com/rajm-weboccult/Basics-of-OpenCV/tree/main/image_processing_code