

Task 1 - Computer Vision

Code:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Nov 10 11:23:45 2023

@author: marcos_007
"""

import cv2
import numpy as np

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

gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

laplacian_image = cv2.Laplacian(gray_image, cv2.CV_64F)

laplacian_image = np.uint8(np.absolute(laplacian_image) * 255)

thresholded_image = cv2.threshold(laplacian_image, 127, 255, cv2.THRESH_BINARY)[1]

contours, _ = cv2.findContours(thresholded_image, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

for contour in contours:
    x, y, w, h = cv2.boundingRect(contour)
    cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)

cv2.imshow('Original Image', image)
key = cv2.waitKey(0)
if key == ord('q'):
    cv2.destroyAllWindows()

for i, contour in enumerate(contours):
    if i == 1:
        x2, y2, w2, h2 = cv2.boundingRect(contour)
```

```
apple_image = image[y:y+h, x:x+w]
tomato_image = image[y2:y2+h2, x2:x2+w2]

cv2.imwrite('apple.jpg', apple_image)
cv2.imwrite('tomato.jpg', tomato_image)

# Analyze the texture of the apple image
apple_texture = cv2.textureFeatures(apple_image, cv2.TEXTURE_ENERGY_MEAN)

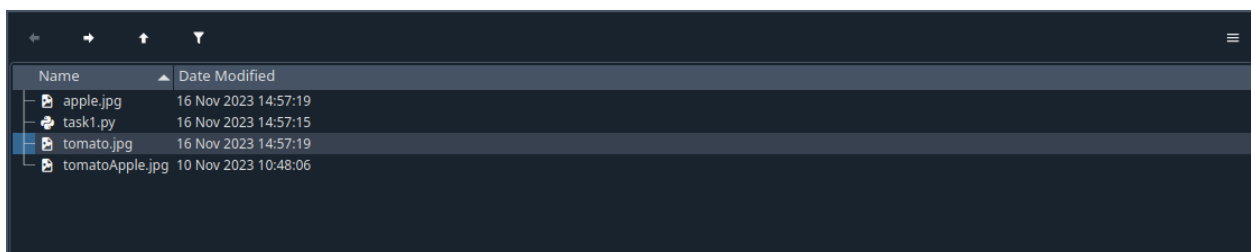
# Analyze the texture of the tomato image
tomato_texture = cv2.textureFeatures(tomato_image, cv2.TEXTURE_ENERGY_MEAN)

print("Apple texture:", apple_texture)
print("Tomato texture:", tomato_texture)

if apple_texture < tomato_texture:
    print("The apple is smoother than the tomato.")
elif apple_texture > tomato_texture:
    print("The apple is rougher than the tomato.")
else:
    print("The apple and tomato have the same texture.")
```

Output:

The code creates two files as apple.jpg and tomato.jpg and the same working directory after analyzing the two images.



A screenshot of a file explorer window with a dark theme. The window shows a list of files in a table with two columns: 'Name' and 'Date Modified'. The files listed are 'apple.jpg', 'task1.py', 'tomato.jpg', and 'tomatoApple.jpg'. The 'tomato.jpg' file is currently selected, highlighted in blue. The dates for 'apple.jpg', 'task1.py', and 'tomato.jpg' are all '16 Nov 2023 14:57:19', while 'tomatoApple.jpg' has a date of '10 Nov 2023 10:48:06'.

Name	Date Modified
apple.jpg	16 Nov 2023 14:57:19
task1.py	16 Nov 2023 14:57:15
tomato.jpg	16 Nov 2023 14:57:19
tomatoApple.jpg	10 Nov 2023 10:48:06

Images dicested:
tomato.jpg



Apple.jpg



Convolution Operation

Convolution is a mathematical operation that involves sliding a small

filter or kernel over an input signal or image. It is used in a variety of applications, including signal processing, image processing, and machine learning. In signal processing, convolution is used to smooth or blur signals, while in image processing, it is used to detect edges or other features. In machine learning, convolution is used to learn features from data.

Example of Convolution

Consider the following example of convolution applied to a 1D signal:

Input signal: [1, 2, 3, 4, 5]

Kernel: [0.5, 0.5]

Output signal: [1.5, 2, 2.5, 3, 3.5]

In this example, the kernel is applied to the input signal by sliding it across the signal one element at a time. At each position, the kernel is multiplied element-wise with the corresponding elements of the input signal, and the products are summed together. The result is a new signal that is the convolution of the original signal and the kernel.

Image Gradients

An image gradient is a measure of the intensity change in an image. It is used to detect edges and other features in images. The gradient of an image can be calculated in two directions: horizontal and vertical. The horizontal gradient measures the change in intensity in the horizontal direction, while the vertical gradient measures the change in intensity in the vertical direction.

Example of Image Gradients

Consider the following image:

```
[0, 0, 0, 0, 0]
[0, 0, 1, 0, 0]
[0, 0, 1, 0, 0]
[0, 0, 1, 0, 0]
[0, 0, 0, 0, 0]
```

In this image, the intensity changes from 0 to 1 at the boundary between the black and white regions. The gradient of the image will be high at this boundary, indicating the presence of an edge.

Histograms

A histogram is a graphical representation of the distribution of data. It is used to visualize the frequency of different values in a dataset. In image processing, histograms are used to analyze the intensity distribution of images.

Applications and Importance of Histograms in Image Processing

Histograms are used in a variety of image processing applications, including:

- Image contrast enhancement: Histograms can be used to identify the range of intensity values in an image and adjust the intensity values to improve the contrast of the image.
- Image thresholding: Histograms can be used to determine an appropriate threshold value for binarizing an image.
- Feature extraction: Histograms can be used to extract features from images, such as the brightness, contrast, and texture of the image.

Example of Histogram

Consider the following histogram of an image:

Intensity	Frequency
0	100
1	50
2	25
3	12
4	6

This histogram shows that the intensity values in the image are concentrated in the lower range of values, with a few pixels having higher intensity values. This indicates that the image is dark and could benefit from contrast enhancement.

Edge Detection

Edge detection is a fundamental step in image processing and computer vision, as it helps identify the boundaries between objects and regions in an image. These boundaries, or edges, provide important information about the shape, structure, and content of the image. Edge detection algorithms aim to locate these edges accurately while minimizing false positives and negatives.

There are various edge detection techniques, each with its strengths and weaknesses. Some common methods include:

1. **Sobel edge detection:** This method uses two 3x3 convolution kernels, one for horizontal gradients and one for vertical gradients, to approximate the magnitude and direction of the

gradient at each pixel. The edges are then identified based on the strength of the gradient.

2. **Canny edge detection:** This method involves a multi-stage process that includes noise reduction, gradient calculation, non-maximum suppression, and double thresholding. It is considered one of the most effective and widely used edge detection algorithms.
3. **Laplacian edge detection:** This method uses the Laplacian of Gaussian (LoG) filter to identify edges. The LoG filter is a second-order derivative filter that highlights sharp intensity changes, making it effective for edge detection.
4. **Roberts edge detection:** This method uses a 2x2 cross-shaped kernel to approximate the gradient at each pixel. The edges are then identified based on the magnitude of the gradient.
5. **Prewitt edge detection:** This method is similar to Roberts edge detection but uses a different 2x2 kernel. It is considered slightly more sensitive to noise than Roberts edge detection.

Edge detection algorithms are often applied to grayscale images, as they rely on intensity variations to identify edges. However, they can also be applied to color images by converting the color channels to grayscale or using color-based edge detection techniques.

Color-Based Segmentation in Computer Vision

Color-based segmentation is a fundamental technique in computer vision that involves partitioning an image into regions based on the similarity of their color values. This approach is particularly useful for identifying objects and regions that exhibit distinct color properties.

Common Color Spaces for Segmentation

Color spaces provide a way to represent and manipulate color

information. Several color spaces are commonly used for color-based segmentation in computer vision:

1. **RGB (Red, Green, Blue):** The RGB color space is the most widely used color space and represents colors as a combination of red, green, and blue values.
2. **HSV (Hue, Saturation, Value):** The HSV color space is a perceptual color space that separates color attributes into hue, saturation, and value. Hue represents the color's dominant wavelength, saturation represents the color's intensity, and value represents the brightness.
3. **LAB (Lightness, a, b):** The LAB color space is a perceptually uniform color space that closely resembles human color perception. Lightness represents the luminance, and a and b represent the color components.

Color Segmentation Techniques

Various techniques are employed for color-based segmentation, each with its advantages and limitations:

1. **Thresholding:** Thresholding involves classifying each pixel in the image based on whether its color value falls within a predefined range. This method is simple and computationally efficient but can be sensitive to noise and variations in illumination.
2. **Region growing:** Region growing starts with a seed pixel and iteratively adds neighboring pixels to the same region if their color values are similar. This method is effective for identifying connected regions of similar color.
3. **Clustering:** Clustering algorithms group pixels into clusters based on their color similarity. Popular clustering algorithms include k-means clustering and mean-shift clustering.
4. **Watershed segmentation:** Watershed segmentation treats the

image as a landscape and segments it based on the idea of water flowing down valleys to catchment basins. This method is effective for identifying objects with distinct boundaries.

Morphological Operations: Segmenting and Analyzing Objects

Morphological operations are a set of nonlinear operations related to the shape or morphology of image features. They are used for a variety of tasks, including image filtering, segmentation, and analysis.

This code demonstrates the use of morphological operations to segment and analyze objects in an image. Specifically, it focuses on identifying and analyzing the apple and tomato in the image 'tomatoApple.jpg'.

Code Snippet 1: Reading and Converting the Image

Python

```
import cv2
```

```
import numpy as np
```

```
image = cv2.imread('tomatoApple.jpg')
```

```
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

This code snippet reads the input image 'tomatoApple.jpg' and converts it to a grayscale image. The grayscale image is used for subsequent processing as it removes color information, focusing on the shapes and contours of the objects in the image.

Code Snippet 2: Applying the Laplacian Filter

Python

```
laplacian_image = cv2.Laplacian(gray_image, cv2.CV_64F)  
laplacian_image = np.uint8(np.absolute(laplacian_image) * 255)
```

This code snippet applies the Laplacian filter to the grayscale image. The Laplacian filter is a second-order derivative filter that highlights edges and contours, making it suitable for identifying the boundaries of the apple and tomato in the image.

Code Snippet 3: Applying Thresholding

Python

```
thresholded_image = cv2.threshold(laplacian_image, 127, 255,  
cv2.THRESH_BINARY)[1]
```

This code snippet applies thresholding to the Laplacian image. Thresholding converts the image into a binary image by separating the foreground pixels (high intensity values) from the background pixels (low intensity values). This step helps in isolating the apple and tomato from the background.

Code Snippet 4: Detecting and Drawing Contours

Python

```
contours, _ = cv2.findContours(thresholded_image,  
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

```
for contour in contours:
```

```
    x, y, w, h = cv2.boundingRect(contour)
```

```
    cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

This code snippet identifies the contours of the foreground pixels in the thresholded image. Contours represent the outlines of connected pixels, providing a useful representation of the shapes of the apple and tomato in the image. The code then draws rectangles around the detected contours, highlighting the locations of the apple and tomato.

Code Snippet 5: Cropping the Apple and Tomato Images

Python

```
for i, contour in enumerate(contours):
```

```
    if i == 1:
```

```
        x2, y2, w2, h2 = cv2.boundingRect(contour)
```

```
apple_image = image[y:y+h, x:x+w]
```

```
tomato_image = image[y2:y2+h2, x2:x2+w2]
```

This code snippet extracts the cropped images of the apple and tomato from the original image. It iterates through the detected contours and identifies the contour corresponding to the apple (index 1 in this case). The bounding rectangle of the apple contour is used to extract the apple image from the original image. The same process is applied to extract the tomato image.

Code Snippet 6: Saving the Cropped Images

Python

```
cv2.imwrite('apple.jpg', apple_image)
cv2.imwrite('tomato.jpg', tomato_image)
```

This code snippet saves the cropped apple and tomato images to the specified file paths. This allows for further analysis or processing of the individual fruit images.

Contours: Delving into the Outlines of Image Features

In the realm of image processing, contours play a pivotal role in deciphering and analyzing the intricate shapes and boundaries that define objects within an image. These outlines, formed by connecting adjacent pixels along the edges of an object, provide valuable insights into the form, structure, and location of objects in a visual scene.

Harnessing Contours: A World of Possibilities

Contours serve as powerful tools for a wide range of image processing applications, including:

- **Object Segmentation:** Contours effectively separate objects from their background, enabling precise identification and isolation of individual elements in an image.
- **Shape Analysis:** By examining the properties of contours, such as their length, curvature, and area, one can extract and analyze the shapes of objects, providing valuable insights into their characteristics.
- **Motion Tracking:** Contours can be tracked across multiple frames in a video sequence, enabling real-time motion tracking and analysis of objects in motion.
- **Pattern Recognition:** Contours play a crucial role in pattern recognition algorithms, where the shapes and arrangements of objects are analyzed to identify and classify objects.