Computer vision practical assignment 1

Images Segmentation .

20321233 Zou Letian

20321308 Cao Xinyang

20321309 Chen Hanzheng

1.Objective

Study of the basic methods for images segmentation into semantic areas.

2.Procedure of Practical Assignment Performing

1. Binarization. Choose an arbitrary image. Perform the image binarization using the considered methods. Depending on the image, use upper or lower threshold binarization.
2. Segmentation 1. Select an arbitrary image containing the face(s). Perform the image segmentation according to the Weber principle (obligatory). Perform the image segmentation based on the skin color and try different formulas with different photo illumination conditions (optional).
3. Segmentation 2. Select an arbitrary image containing a limited number of colored objects. Perform image segmentation in the CIE Lab color space by the nearest neighbors method (obligatory). Perform image segmentation in the CIE Lab color space by the k-means method (optional).
4. Segmentation 3. Select an arbitrary image containing two heterogeneous textures. Perform texture segmentation of the image (obligatory). Evaluate at least three parameters of the selected textures, determine which class the textures belong to (optional).

Note. Please note that when doing the practical assignment you are not allowed to use the “Lenna” image or any other image that was used either in this book or during the presentation.

2.1 Part1　Binarization.

original images

Binary

```
import numpy as np
from PIL import Image

# Load the image as a grayscale image
I = np.array(Image.open("yjsp.jpg").convert('L'))

# Set the threshold value
t = 127

# Apply the thresholding operation
Inew = np.where(I > t, 255, 0).astype('uint8')

# Display the binary image
Image.fromarray(Inew).show()
```

resulting images

Comments： This code performs image binarization, which is a process of converting a grayscale image into a binary image. The binary image has only two possible pixel values, typically 0 and 255, which represent black and white, respectively.

The code first loads an image named "yjsp.jpg" as a grayscale image using the PIL library. It then sets a threshold value of 127, which is used to determine whether a pixel should be black or white in the binary image. Pixels with intensity values greater than the threshold value are set to 255 (white), while pixels with intensity values less than or equal to the threshold value are set to 0 (black). Finally, the binary image is displayed using the PIL library.

Overall, this code demonstrates a simple but effective technique for image binarization.

Double threshold Binary

```
import cv2

# Load the image
I = cv2.imread("yjsp.jpg",cv2.IMREAD_COLOR)

# Set the threshold values
t1 = 127
t2 = 200

# Convert the image to grayscale
Igray = cv2.cvtColor(I, cv2.COLOR_BGR2GRAY)

# Apply double threshold binary
ret, Inew = cv2.threshold(Igray, t1, 255,cv2.THRESH_BINARY)

# Display the image
cv2.imshow( "Double threshold Binary" , Inew)
cv2.waitKey(0)
```

resulting images

Comments: This code performs double threshold binary on an image. It first loads an image using OpenCV's imread function. Then, it sets two threshold values, t1 and t2. The image is converted to grayscale using cv2.cvtColor function. Finally, double threshold binary is applied using cv2.threshold function with the grayscale image and the two threshold values as inputs. The resulting binary image is displayed using cv2.imshow function. The program waits for a key press before closing the window.

Binarization by adaptive method

```
import cv2

# Read the image file "yjsp.jpg" in color mode using the imread()
function from OpenCV.
I = cv2.imread("yjsp.jpg", cv2.IMREAD_COLOR)

# Convert the color image "I" to grayscale using the cvtColor()
function from OpenCV.
Igray = cv2.cvtColor(I, cv2.COLOR_BGR2GRAY)

# Apply adaptive thresholding to the grayscale image "Igray" using the
adaptiveThreshold() function from OpenCV.
Inew = cv2.adaptiveThreshold(Igray, 255,
                             cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                             cv2.THRESH_BINARY, 11, 2)

# Display the binary image using the imshow() function from OpenCV.
cv2.imshow( "Binarization by adaptive method" , Inew)

# Wait for a key press before closing the window.
cv2.waitKey(0)
```

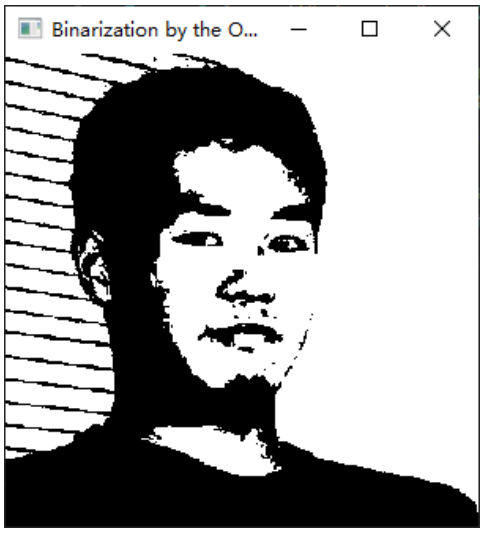
resulting images

Comments: This code reads an image file named "yjsp.jpg" in color mode using the imread() function from OpenCV. Then, it converts the color image to grayscale using the cvtColor() function from OpenCV. Finally, it applies adaptive thresholding to the grayscale image using the adaptiveThreshold() function from OpenCV. The resulting binary image is displayed using the imshow() function from OpenCV, and the program waits for a key press before closing the window.

Binarization by the Otsu method

```
# This code reads an image and applies the Otsu method for binarization
import cv2

# Reading the image in grayscale
I = cv2.imread("yjsp.jpg", cv2.IMREAD_GRAYSCALE)

# Applying the Otsu method for binarization
ret, Inew = cv2.threshold(I, 0, 255,
                          cv2.THRESH_OTSU)

# Displaying the binarized image
cv2.imshow( "Binarization by the Otsu method" , Inew)
cv2.waitKey(0)
```

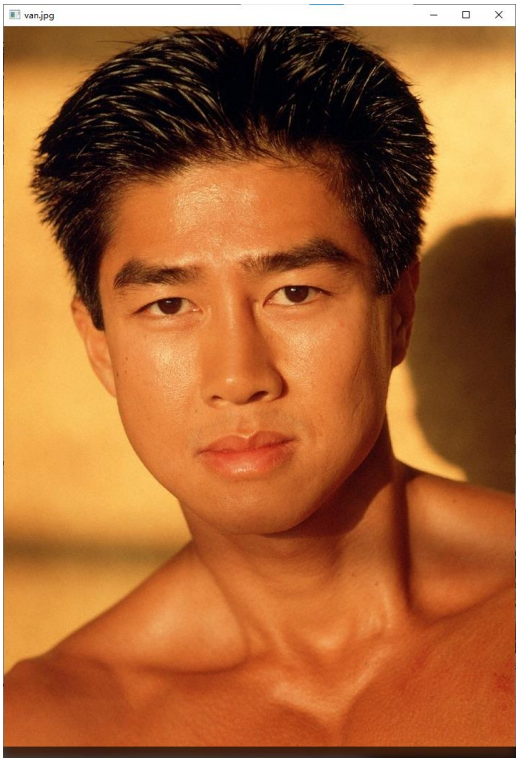
resulting images



comments： This code reads an image in grayscale using OpenCV's cv2.imread() function. Then, it applies the Otsu method for binarization using cv2.threshold() function. The Otsu method is a thresholding technique that automatically calculates a threshold value from image histogram for a bimodal image. Finally, it displays the binarized image using cv2.imshow() function and waits for a key event using cv2.waitKey() function.

2.2 Part2　Segmentation 1

original images



SkinColorSegmentation

```
# Importing necessary libraries
import numpy as np
import cv2

# Defining function for skin color segmentation
def SkinColorSegmentation(imname):
  # Reading image
  img = cv2.imread(imname, cv2.IMREAD_COLOR)
  # Converting image to HSV color space
  hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
  (_h, _s, _v) = cv2.split(hsv)
  # Creating an array of zeros with the same shape as _h
  skin3 = np.zeros(_h.shape, dtype=np.uint8)
  (x, y) = _h.shape
  # Looping through each pixel of the image
  for i in range(0, x):
    for j in range(0, y):
      # Checking if the pixel is within the skin color range
      if (_h[i][j] > 7) and (_h[i][j] < 20) and (_s[i][j] > 28) and
(_s[i][j] < 255) and (_v[i][j] > 50) and (
        _v[i][j] < 255):
        # Setting the pixel value to 255 if it is within the skin color
range
        skin3[i][j] = 255
      else:
        # Setting the pixel value to 0 if it is not within the skin
color range
        skin3[i][j] = 0

  # Displaying the original image and the skin color segmented image
  cv2.imshow(imname, img)
  cv2.imshow(imname + " Skin3 HSV", skin3)

# Running the function on the given image
if __name__=="__main__":
  imname="van.jpg"
  SkinColorSegmentation(imname)
  cv2.waitKey(0)
```

resulting images

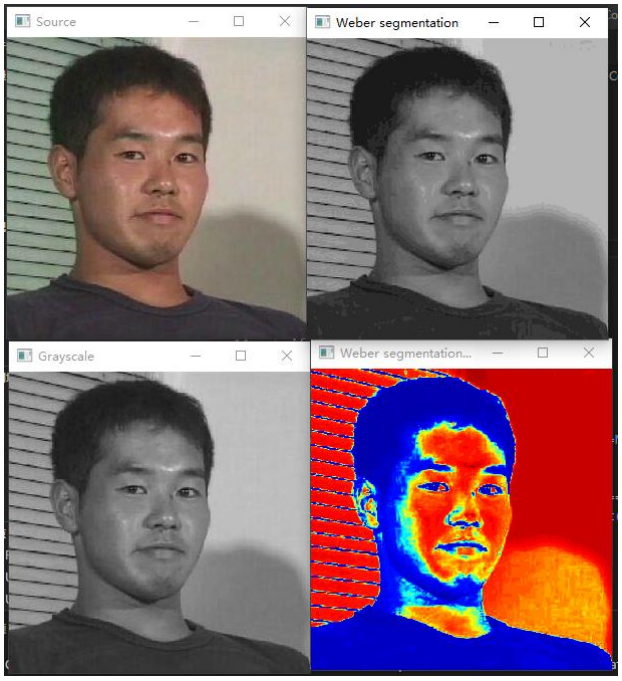
Comments: This code is for skin color segmentation of an image. It reads an image and converts it to the HSV color space. Then it loops through each pixel of the image and checks if the pixel is within the skin color range. If the pixel is within the skin color range, it sets the pixel value to 255, otherwise it sets the pixel value to 0. Finally, it displays the original image and the skin color segmented image.

WeberSegmentation

```
import cv2
import numpy as np

# Define the Weber function
def weber(i):
  if i<0:
    return 0
  if i>255:
    return 255
  if i<=88:
    return int(20-12*i/88)
  if i<=138:
    return int(0.002*(i-88)*(i-88))
  return int(7*(i-138)/117+13)
```

```
# Define the Weber Segmentation function
def WeberSegmentation(fn= "yjsp.jpg",fn_out=None):
  # Read the image
  I=cv2.imread(fn,cv2.IMREAD_COLOR)
  if not isinstance(I,np.ndarray) or I.data==None:
    print("Error reading file \"{}\"".format(fn))
    exit()

  # Show the source image
  cv2.imshow("Source",I)

  # Convert the image to grayscale
  Igray=cv2.cvtColor(I,cv2.COLOR_BGR2GRAY)
  cv2.imshow("Grayscale",Igray)

  # Initialize the Weber image and the Weber2 image
  Iweber =np.zeros_like(Igray)
  Iweber2=np.zeros_like(Igray)
  n=1

  # Perform Weber segmentation
  while(Iweber==0).any():
    Imin=Igray[Iweber==0].min()
    Iw=weber(Imin)
    n=n+1
    mask=np.logical_and(Igray >=Imin,Igray<=Imin+Iw)
    Iweber[mask]=n
    Iweber2[mask]=Imin
  n=n-1
  Iweber=Iweber-1

  # Show the Weber segmentation image in JET colormap
  cv2.imshow("Weber segmentation
JET",cv2.applyColorMap((Iweber.astype(np.float32)*255/(n+1)).astype(np.
uint8),cv2.COLORMAP_JET))

  # Show the Weber segmentation image
  cv2.imshow("Weber segmentation",Iweber2)
  cv2.waitKey(0)

# Call the Weber Segmentation function
WeberSegmentation("yjsp.jpg")
```

Resulting images:



Comments: This code is an implementation of Weber Segmentation algorithm. It reads an image, converts it to grayscale, and then performs Weber segmentation on it. The Weber function is defined to calculate the Weber value of a pixel intensity. The Weber value is used to segment the image. The output of the segmentation is shown in two windows: one with the Weber segmentation image in JET colormap and the other with the Weber segmentation image.

2.3 Part3 Segmentation 2

original images

```python
import cv2 as cv
import numpy as np


##########################################################################
###
## Mouse event handler function
# @param[in] event Mouse event id
# @param[in] x X coordinate of the mouse event
# @param[in] y Y coordinate of the mouse event
# @param[in] flags Event flags
# @param[in] param Additional parameters passed to event. In current
#            implementation it is a tuple containing source image in
#            RGB color space, same image in Lab color space and an array
#            to store selected pixel coordinates.
## Mouse event handler function

def MouseHandler(event, x, y, flags, param):
    # Only double click event is processed
    if event != cv.EVENT_LBUTTONDBLCLK:
        return

    # Extract data from parameters
    if param is None:
        return
    I, Ilab, sampleAreas, radius, colorMarksBGR = param

    # Add current double-clicked point to the list
    sampleAreas.append((x, y))

    # Create new image with marked pixel
    I2 = I.copy()
    for pix in sampleAreas:
        cv.circle(I2, pix, radius, (0, 0, 255), 1)
    cv.imshow("Source", I2)

    # Calculate color means
    colorMarks = []
    for pix in sampleAreas:
        # Create a mask for the current pixel location
        mask = np.zeros_like(Ilab[0])
        cv.circle(mask, pix, radius, 255, -1)
```

```
        # Calculate the mean values of the a and b channels of the Lab
color space within the mask
        a = Ilab[1].mean(where=mask > 0)
        b = Ilab[2].mean(where=mask > 0)
        # Add the mean values to the list of color marks
        colorMarks.append((a, b))
        # Calculate the mean BGR color within the mask and add it to the
list
        colorMarksBGR.append(I[mask > 0, :].mean(axis=(0)))

    # Calculate distance and create segmented areas
    labels = np.zeros_like(Ilab[0], dtype=np.uint8)
    for i, pix in enumerate(colorMarks):
        # Calculate the Euclidean distance between each pixel in the Lab
color space and the color mark
        d = np.sqrt((Ilab[1] - pix[0])**2 + (Ilab[2] - pix[1])**2)
        # Create a mask for pixels that are within radius distance of
the color mark
        labels[d < np.sqrt(2)*radius] = i + 1

    # Show segmented image
    cv.imshow("Segmented", cv.cvtColor(labels*50, cv.COLOR_GRAY2BGR))


## Perform the segmentation in the CIE Lab color space using the
nearest neighbor method
# @param[in] fn Image file name
def CIELabSegmentation(fn='yjsp.jpg'):
    # Read an image from file
    I = cv.imread(fn, cv.IMREAD_COLOR)
    if not isinstance(I, np.ndarray) or I.data is None:
        print(f"Error reading file \"{fn}\"")
        return
    cv.imshow("Source", I)

    # Convert to CIE Lab color space
    Ilab = cv.cvtColor(I, cv.COLOR_BGR2LAB)
    Ilab = cv.split(Ilab)

    # Define mouse callback function
    sampleAreas = []
    colorMarksBGR = []
    cv.setMouseCallback("Source", MouseHandler, (I, Ilab, sampleAreas,
10, colorMarksBGR))
```

```
    # Start an infinite event processing loop
    while True:
        key = cv.waitKey(20) & 0xFF
        if key == 27:
            break
        elif key == 114:
            cv.destroyAllWindows()
            sampleAreas = []
            colorMarksBGR = []
            cv.imshow("Source", I)
            cv.setMouseCallback("Source", MouseHandler, (I, Ilab,
sampleAreas, 10, colorMarksBGR))

    cv.destroyAllWindows()


if __name__ == "__main__":
    CIELabSegmentation("yjsp.jpg")
```

Resulting images:

Comments: This is a script that performs image segmentation in the CIE Lab color space using the nearest neighbor method. The script reads an image from file, converts it to the CIE Lab color space, and then displays the image. The user can then double-click on the image to select a region of interest, which is used to calculate the mean values of the a and b channels of the Lab color space within the region. These mean values are then used as color marks to segment the image. The script creates a new image with marked pixels, calculates the mean values of the a and b channels of the Lab color space within the selected region, and then calculates the Euclidean distance between each pixel in the Lab color space and the color mark. Pixels that are within a certain radius distance of the color mark are assigned to the same segment. The segmented image is then displayed.

2.4 Part4 Segmentation 3

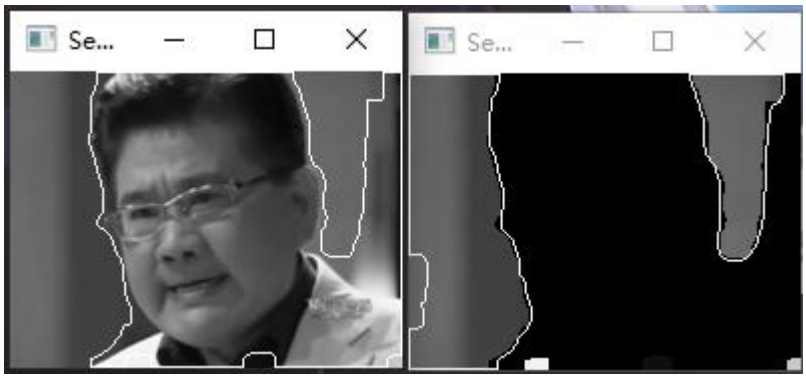
original images

```python
import cv2
import numpy as np
from skimage import filters, morphology
from PA1_Appendix import bwareaopen, imfillholes

# Read the image in grayscale
I = cv2.imread("jzdc.jpg",cv2.IMREAD_GRAYSCALE)

# Calculate the entropy of the image
E = filters.rank.entropy(I,morphology.square(9)).astype(np.float32)

# Normalize the entropy image
Eim = (E - E.min()) / (E.max() - E.min())

# Display the entropy image
cv2.imshow("E", Eim)

# Threshold the entropy image using Otsu's method
ret, BW1 = cv2.threshold(np.uint8(Eim * 255), 0, 255,cv2.THRESH_OTSU)

# Display the thresholded image
cv2.imshow("BW1", BW1)

# Remove small objects from the thresholded image
BWao = bwareaopen(BW1, 2000)

# Define a structuring element for morphological operations
nhood = cv2.getStructuringElement(cv2.MORPH_RECT, (9, 9))

# Close the thresholded image
closeBWao = cv2.morphologyEx(BWao, cv2.MORPH_CLOSE, nhood)

# Fill holes in the closed image
Mask1 = imfillholes(closeBWao)

# Display the processed images
```

```
cv2.imshow("After bwareaopen 1", BWao)
cv2.imshow("After close 1", closeBWao)
cv2.imshow("After fill holes 1", Mask1)

# Find contours in the filled image
contours, h = cv2.findContours(Mask1, cv2.RETR_TREE,
cv2.CHAIN_APPROX_NONE)

# Create a binary image of the contour boundary
boundary = np.zeros_like(Mask1)
cv2.drawContours(boundary, contours, -1, 255, 1)

# Create a segmented image using the boundary
segmentResults = I.copy()
segmentResults[boundary != 0] = 255

# Display the segmented image
cv2.imshow("Segment result 1", segmentResults)

# Create a masked image of the original image
I2 = I . copy ()
I2 [ Mask1 != 0] = 0

# Calculate the entropy of the masked image
E2 = filters . rank . entropy ( I2 , morphology . square (9)). astype
( np . float32 )

# Normalize the entropy image
Eim2 = ( E2 - E2 . min ()) / ( E2 . max () - E2 . min ())

# Display the entropy image
cv2.imshow("E2", Eim2)

# Threshold the entropy image using Otsu's method
ret , BW2 = cv2 . threshold ( np . uint8 ( Eim2 * 255) , 0 , 255 ,
cv2 . THRESH_OTSU )

# Display the thresholded image
cv2.imshow("BW2", BW2)

# Remove small objects from the thresholded image
BW2ao = bwareaopen ( BW2 , 2000)

# Close the thresholded image
```

```
closeBW2ao = cv2 . morphologyEx ( BW2ao , cv2 . MORPH_CLOSE , nhood )

# Fill holes in the closed image
Mask2 = imfillholes ( closeBW2ao )

# Display the processed images
cv2.imshow("After bwareaopen 2", BWao)
cv2.imshow("After close 2", closeBWao)
cv2.imshow("After fill holes 2", Mask1)

# Find contours in the filled image
contours2 , h = cv2 . findContours ( Mask2 , cv2 . RETR_TREE , cv2 .
CHAIN_APPROX_NONE )

# Create a binary image of the contour boundary
boundary2 = np . zeros_like ( Mask2 )
cv2 . drawContours ( boundary2 , contours2 , -1 , 255 , 1)

# Create a segmented image using the boundary
segmentResults2 = I2 . copy ()
segmentResults2 [ boundary2 != 0] = 255

# Display the segmented image
cv2.imshow("Segment result 2", segmentResults2)

# Create a masked image of the original image
texture1 = I . copy ()
texture1 [ Mask2 == 0] = 0

# Create a masked image of the original image
texture2 = I . copy ()
texture2 [ Mask2 != 0] = 0

# Display the texture images
cv2.imshow("Texture 2", texture2)

# Wait for user input
cv2.waitKey(0)
```

Resulting images:



Comments: This code is an implementation of texture segmentation using entropy filtering. The code reads an image in grayscale, calculates the entropy of the image, normalizes the entropy image, and displays it. Then, it thresholds the entropy image using Otsu's method and displays the thresholded image. It removes small objects from the thresholded image, closes the thresholded image, fills holes in the closed image, and displays the processed images. It finds contours in the filled image, creates a binary image of the contour boundary, and creates a segmented image using the boundary. It displays the segmented image. It creates a masked image of the original image, calculates the entropy of the masked image, normalizes the entropy image, and displays it. Then, it thresholds the entropy image using Otsu's method and displays the thresholded image. It removes small objects from the thresholded image, closes the thresholded image, fills holes in the closed image, and displays the processed images. It finds contours in the filled image, creates a binary image of the contour boundary, and creates a segmented image using the boundary. It displays the segmented image. It creates a masked image of the original image and displays the texture images. Finally, it waits for user input.

3. Conclusion

Traditional machine vision usually consists of two steps: preprocessing and object detection. The bridge between the two is image segmentation.

The machine needs to distinguish the subject and background in the image after pretreatment and optimization, and make analysis, in order to make an effective and accurate judgment. But the machine can not distinguish the subject and background of the picture directly like human, so it is necessary to use the binarization of the image to process the picture.

Each pixel of a binary image has only two color values: black and white.

Binarization method can be divided into global threshold method and local threshold method, among which the global threshold method includes Otsu method. It is the best method to select the threshold value in image segmentation. Besides the calculation is simple, it will not be affected by image brightness and contrast.

In a word, the important significance of image binarization processing is to simplify the later processing and improve the processing speed

4. Answers to questions

1. When is it appropriate to use Weber segmentation?

Weber segmentation is most suitable for images with significant differences in brightness between different regions or objects. This technique is particularly useful for segmenting images of industrial or biological specimens, with an emphasis on identifying areas of interest based on their brightness or contrast.

2. What are the a and b color coordinates of the CIE Lab color space in a grayscale image?

In the CIE Lab color space, the a and b color coordinates represent the red, green and yellow and blue components of a color. Axis a represents the green/red component, which ranges from -128 (green) to +127 (red); The b axis represents the blue/yellow component, which ranges from -128 (blue) to +127 (yellow).

3. What is the reason for performing an image segmentation in the CIE Lab color space and not in the original RGB one?

CIE Lab color space separates color information from brightness information, making it more suitable for image segmentation. Meanwhile, CIE Lab color space is based on the perception color theory, which can better reflect human's perception of color.