Reto 3 - Segmentación watershed y detección de contornos

(Martes 13 Febrero)

3.1. Segmentación Watershed

Cualquier imagen en escala de grises se puede ver como una superficie topográfica donde una intensidad alta indica picos y colinas, mientras que intensidades bajas indican valles. En este algoritmo se empieza por llenar cada valle aislado (mínimos locales) con agua de diferentes colores (etiquetas). A medida que el agua sube, dependiendo de los picos (pendientes) cercanos, el agua de diferentes valles, obviamente con diferentes colores, comenzará a fusionarse. Para evitar esto, se construyen barreras en los lugares donde se une el agua. Luego, se continúa el trabajo de rellenar con agua y construir barreras hasta que todos los picos estén bajo el agua. Así, las barreras creadas en este proceso, no son más que la segmentación de la imagen. Esta es la "filosofía" detrás del algoritmo *Watershed*.

Sin embargo, este enfoque da un resultado sobre-segmentado debido al ruido o a cualquier otra irregularidad en la imagen. Así que *OpenCV* implementó un algoritmo de cuenca hidrográfica basado en marcadores en el que se especifican cuáles son todos los puntos del valle que se fusionarán y cuáles no. Es una segmentación de imagen interactiva. Lo que hacemos es dar diferentes etiquetas a nuestro objeto. De este modo, tendremos que etiquetar la región que estamos seguros de que es el primer plano o el objeto en sí con un color (o intensidad), y debemos etiquetar la región de la que estamos seguros que es el fondo y no el objeto, con otro color. Finalmente la región de la que no estamos seguros de nada la debemos etiquetar con 0. Ese es nuestro marcador. Sólo después de este etiquetado aplicamos el algoritmo *Watershed*. Entonces nuestro marcador se actualizará con las etiquetas que dimos, y los límites de los objetos tendrán un valor de -1.

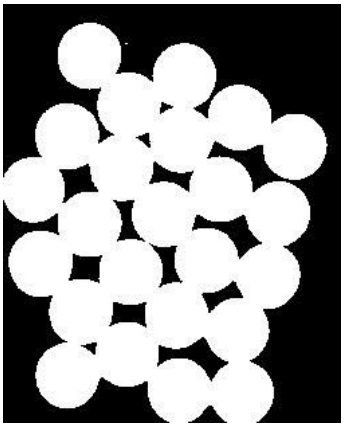
A continuación, veremos un ejemplo sobre cómo usar la Transformación de distancia junto con el *Watershed* para segmentar objetos que se tocan mutuamente.

Considere la imagen de las monedas a continuación, las monedas se tocan entre sí. Incluso si lo limitas, se tocarán entre sí.

Comenzamos por encontrar una estimación aproximada de las monedas. Para eso, podemos usar la binarización de *Otsu*.

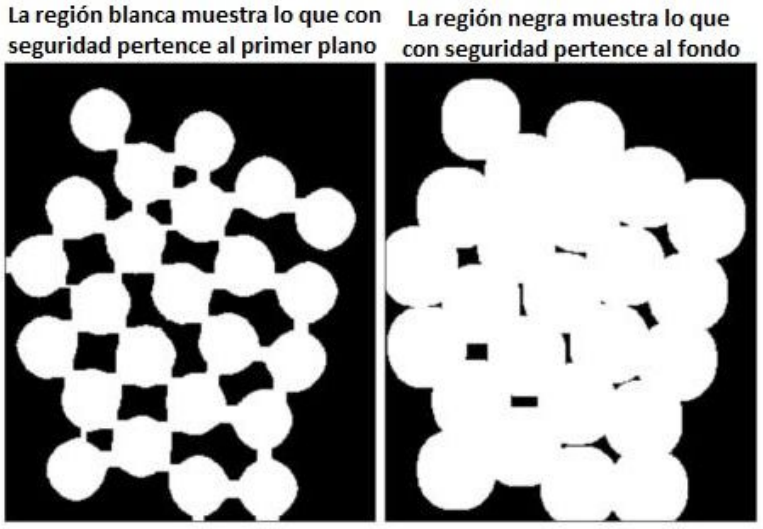
```
import numpy as np

import cv2

from matplotlib import pyplot as plt

img = cv2.imread('monedas.png')

gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

ret, thresh = cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
```

El resultado:



Ahora necesitamos eliminar cualquier pequeño ruido blanco en la imagen. Para eso podemos usar la apertura morfológica. Para eliminar cualquier agujero pequeño en el objeto, podemos usar el cierre morfológico. Por lo tanto, ahora sabemos con certeza que la región cercana al centro de los objetos está en primer plano y que la región más alejada del objeto es el fondo. Solo la región de la que no estamos seguros es la región límite de las monedas.

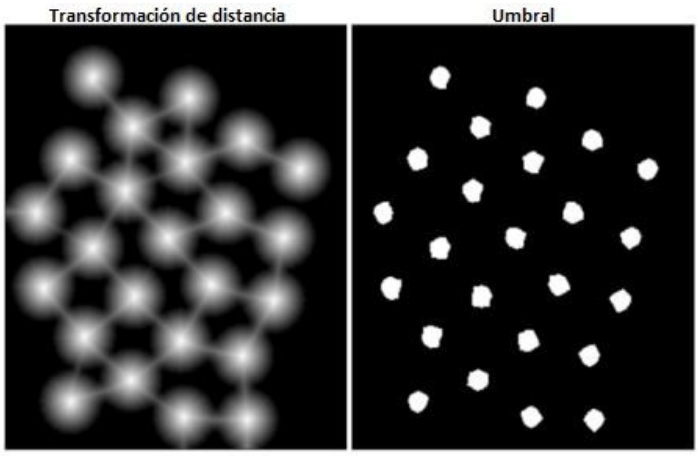
Entonces necesitamos extraer el área de la cual estamos seguros que son monedas. La erosión elimina los píxeles del límite. Entonces, lo que quede, podemos estar seguros de que es una moneda. Eso funcionaría si los objetos no se tocaran entre sí. Pero como se están tocando entre sí, otra buena opción sería encontrar la distancia de transformación y aplicar un umbral adecuado. Luego tenemos que encontrar el área que estamos seguros de que no son monedas. Para eso, dilatamos el resultado. La dilatación aumenta el límite del objeto al fondo. De esta forma, podemos asegurarnos de que cualquier región en el fondo en el resultado sea realmente un fondo, ya que la región límite se elimina. Vea la imagen a continuación:



Las regiones restantes son aquellas de las que no tenemos idea de si son monedas o fondo. El algoritmo de Watershed debería ser capaz de discriminar entre monedas y fondo en estas regiones conflictivas. Estas regiones corresponden al área alrededor de los límites de las monedas donde se cruzan el primer plano y el fondo (o incluso se encuentran dos monedas diferentes). Tales regiones delimitantes son las fronteras. Se puede obtener restando el área de la figura de la izquierda del área de la figura de la derecha.

```
# Usaremos el operador morfológico de Apertura como vimos en el reto anterior, para la eliminación del ruido de la imagen.

kernel = np.ones((3,3),np.uint8)

opening = cv2.morphologyEx(thresh,cv2.MORPH_OPEN,kernel, iterations = 2)


# Encuentra el área del fondo

sure_bg = cv2.dilate(opening,kernel,iterations=3)


# Encuentra el área del primer plano

dist_transform = cv2.distanceTransform(opening,cv2.DIST_L2,5)

ret, sure_fg = cv2.threshold(dist_transform,0.7*dist_transform.max(),255,0)


# Encuentra la región desconocida (bordes)

sure_fg = np.uint8(sure_fg)

unknown = cv2.subtract(sure_bg,sure_fg)
```

Vea el resultado en la imagen a la que se ha aplicado un umbral; se obtienen algunas regiones de monedas de las cuales estamos seguros de las monedas y que ahora además están separadas.
En algunos casos, puede interesarle solo la segmentación del primer plano, y no en separar los objetos que se tocan mutuamente. En ese caso, no necesita usar la transformación de distancia, basta con la erosión. La erosión es solo otro método para extraer el área de primer plano.

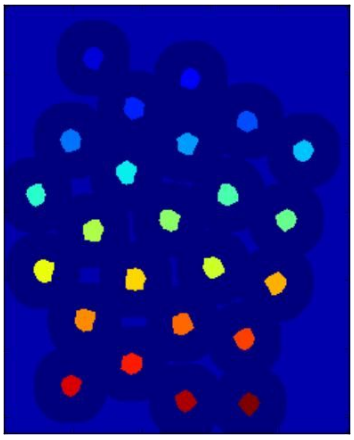


Ahora sabemos con certeza cuáles son las regiones de las monedas, que es parte del fondo y el resto. Ahora creamos marcador (es un arreglo del mismo tamaño que el de la imagen original, pero con el tipo de datos int32) y etiquetamos las regiones dentro de él. Las regiones que sabemos con certeza (ya sea en primer plano o en segundo plano) están etiquetadas con números enteros positivos, pero enteros diferentes, y el área que no sabemos con certeza simplemente queda en cero. Para esto usamos ***cv2.connectedComponents()***. Con esta función etiquetamos el fondo de la imagen con 0, y el resto de los objetos quedan etiquetados con números enteros a partir de 1.

Pero sabemos que si el fondo está marcado con 0, el algoritmo de *Watershed* lo considerará como un área desconocida. Por tanto, queremos marcarlo con un número entero diferente. En cambio, marcaremos la región desconocida, definida como *unknown*, con 0.

```
# Etiquetado

ret, markers = cv2.connectedComponents(sure_fg)

# Adiciona 1 a todas las etiquetas para asegurra que el fondo sea 1 en lugar de
cero

markers = markers+1

# Ahora se marca la región desconocida con ceros

markers[unknown==255] = 0
```

Vea el resultado que se muestra en el mapa de colores JET. La región azul oscura muestra la región desconocida. Las monedas están coloreadas con diferentes valores. El área restante, que es de fondo seguro, se muestra en azul claro en comparación con la región desconocida.



Ahora nuestro marcador está listo. Es hora de dar el paso final, aplicar el algoritmo de Watershed. Al hacer esto la imagen del marcador será modificada y la región límite será etiquetada con -1.

```
markers = cv2.watershed(img,markers)

img[markers == -1] = [255,0,0]
```

Vea el resultado a continuación. Aunque en general, el algoritmo encuentra muy bien las fronteras de las monedas, en algunos casos, en las regiones de contacto entre dos monedas, el algoritmo falla.

https://github.com/avara1986/iot-opencv/blob/master/tip_01_05.py

Ejercicios adicionales

- Contar número de monedas.
- Con una imagen del video que aparezcan dos trabajadores, usar esta técnica para que el algoritmo detecte el número de personas. Es interesante primero realizar una resta entre la imagen de fondo sin trabajador y la imagen con trabajador.

Bibliografía

Para este reto hemos utilizado el articulo https://www.aprenderpython.net/segmentacion-imagenes-algoritmo-watershed/, por su facil explicacion de este algortimo.

3.2 Detección de contornos

Los contornos se pueden explicar simplemente como una curva que une todos los puntos contiguos (a lo largo de un límite), teniendo el mismo color o intensidad. Los contornos son una herramienta útil para el análisis de formas y la detección y el reconocimiento de objetos.
- Para una mayor precisión, use imágenes binarias. Antes de iniciar la detección de contornos es conveniente la aplicación de umbralización.
- La función findContours modifica la imagen de origen. Entonces, si quiere imagen de origen incluso después de encontrar contornos, ya la almacena en algunas otras variables.
- En OpenCV, encontrar contornos es como encontrar un objeto blanco desde un fondo negro. Así que recuerde, el objeto que se debe encontrar debe ser blanco y el fondo debe ser negro.

Como encontramos los contornos en una imagen binaria

```
import numpy as np
import cv2

im = cv2.imread('test.jpg')
imgray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
ret,thresh = cv2.threshold(imgray,127,255,0)
image, contours, hierarchy =
cv2.findContours(thresh,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
```

Para ello usaremos la función cv2.findContours (). Esta función recibe 3 argumentos, el primero es la imagen de origen, el segundo es el modo de recuperación de contorno, el tercero es el método de aproximación de contorno. Y devuelve la imagen, los contornos y la jerarquía de los mismos. Los contornos es una lista de Python de todos los contornos de la imagen. Cada contorno individual es una matriz Numpy de coordenadas (x, y) de puntos límite del objeto.

3.2.1 Dibujar contornos

Una vez que hemos identificado las coordenadas de los contornos, es interesante dibujarlos sobre la imagen, principalmente para saber si se han detectado de manera correcta.

Para dibujar los contornos, se usa la función cv2.drawContours. También se puede usar para dibujar cualquier forma siempre que tenga sus puntos de límite. Su primer argumento es la imagen fuente, el segundo argumento son los contornos que deben pasarse como una lista de Python, el tercer argumento es el índice de contornos (útil al dibujar un contorno en concreto, para dibujar todos los contornos, pasar -1) y los argumentos restantes son el color y grosor de la línea de dibujo.

Código para dibujar todos los contornos de una imagen:

```
img = cv2.drawContours(img, contours, -1, (0,255,0), 3)
```

Código para dibujar un contorno en concreto. En este caso el contorno situado en la posición 4 del array de contornos:

```
img = cv2.drawContours(img, contours, 3, (0,255,0), 3)
```

Ejercicios adicionales

En el ejemplo de a continuacion, podeis ver el desarrollo para la detección de contornos, para el caso de objetos de color rojo. (Ver en el notebook).

Ejercicio: detecta los contornos a imagen de tu elección y analiza que los resultados obtenidos son similares al ejemplo anterior. Pudiendo identificarse claramente los contornos de los objetos cuyo color está dentro del rango de color establecido.

Bibliografía

http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_contours/py_contours_begin/py_contours_begin.html