Universidad Nacional de San Antonio Abad del Cusco
Departamento Académico de Ing. Informática
VISION COMPUTACIONAL
Práctica Nº 06

IMÁGENES BINARIAS Y OPERACIONES MORFOLOGICAS

Iván C. Medrano Valencia

1. OBJETIVO.

- Conocer la umbralización simple, la umbralización adaptativa y la umbralización de Otsu.
- Conocer las operaciones morfológicas de erosión, dilatación, apertura y cierre.

2. ACTIVIDAD I. UMBRALIZACIÓN SIMPLE

En umbralización simple, si el valor del pixel es mayor al valor del umbral, se le asigna un valor (puede ser blanco), de otro modo se le asigna otro valor (puede ser negro). La función utilizada es ***cv2.threshold***. El primer argumento es la imagen fuente, que debería encontrarse en escala de grises. El segundo argumento es el valor del umbral que se usa para calificar los valores de pixeles. El tercer argumento es el maxVal el cual representa el valor dado si el valor del pixel es mayor que (a veces menor que) el valor del umbral. *OpenCV* provee diferentes estilos de umbralización y se decide por medio del cuatro parámetro de la función. Los distintos tipos son:

- THRESH_BINARY
- THRESH_BINARY_INV
- THRESH_TRUNC
- THRESH_TOZERO
- THRESH_TOZERO_INV

Se obtienen dos salidas. La primera es un **retval** y la segunda es nuestra **imagen umbralizada**.

Ejercicio 6.1. En este ejercicio, leemos una imagen en escala de grises y luego la mostramos umbralizada en distintos tipos.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('images\lena.jpg',0)
ret,thresh1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
ret,thresh2 = cv2.threshold(img,127,255,cv2.THRESH_BINARY_INV)
ret,thresh3 = cv2.threshold(img,127,255,cv2.THRESH_TRUNC)
ret,thresh4 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO)
ret,thresh5 = cv2.threshold(img,127,255,cv2.THRESH_TOZERO_INV)
```

```
titles = ['Original Image','BINARY','BINARY_INV','TRUNC','TOZERO','TOZERO
_INV']
images = [img, thresh1, thresh2, thresh3, thresh4, thresh5]
miArray = np.arange(6)
for i in miArray:
  plt.subplot(2,3,i+1),plt.imshow(images[i],'gray')
  plt.title(titles[i])
  plt.xticks([]),plt.yticks([])

plt.show()
```

3. ACTIVIDAD II. UMBRALIZACIÓN ADAPTATIVA.

En la actividad anterior, usamos un valor global como valor umbral. Pero puede no ser bueno en todos los casos donde las imágenes difieren en cuanto a condiciones de luz en distintas áreas. En ese caso, utilizamos la umbralización adaptativa. En esta, el algoritmo calcula el umbral para una pequeña región de la imagen. Así que obtenemos diferentes umbrales para distintas regiones de la misma imagen. Y nos da mejores resultados para imágenes con iluminación variante.
Posee tres parámetros "especiales" de entrada y sólo un argumento de salida.

- **Metodo Adaptativo – Decide cómo el valor de umbralización es calculado.**

  - ADAPTIVE_THRESH_MEAN_C : el valor umbral es equivalente al valor del área vecina.
  - ADAPTIVE_THRESH_GAUSSIAN_C : en este caso el valor umbral es la suma de los pesos de los valores vecinos donde dichos valores correspondían a pesos de una ventana gaussiana.

Ejercicio 6.2.
El siguiente programa compara la umbralización global con la adaptativa para una imagen de iluminación variante:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('images\lena.jpg',0)
img = cv2.medianBlur(img,5)

ret,th1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
th2 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_MEAN_C,cv2.THRESH
_BINARY,11,2)
th3 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,cv2.TH
RESH_BINARY,11,2)
```

```
titles = ['Original Image', 'Global Thresholding (v = 127)','Adaptive Mea
n Thresholding', 'Adaptive Gaussian Thresholding']
images = [img, th1, th2, th3]
miArray = np.arange(4)
for i in miArray:
 plt.subplot(2,2,i+1),plt.imshow(images[i],'gray')
 plt.title(titles[i])
 plt.xticks([]),plt.yticks([])
plt.show()
```

4. ACTIVIDAD III. LA BINARIZACION DE OTSU

En la primera actividad, mencionamos que había un segundo parámetro denominado **retVal**. Su uso llega cuando usamos la Binarización de Otsu. En la umbralización global, utilizamos un valor arbitrario como umbral, y ¿cómo podemos saber si el valor que hemos escogido es bueno o no? La respuesta es, mediante el método de ensayo y error. Pero si consideramos una imagen bimodal (una imagen bimodal es una imagen cuyo histograma posee dos picos). Para esa imagen, podemos tomar un valor aproximado entre esos dos picos como el valor umbral. Eso es lo que hace la binarización de Otsu. En pocas palabras, se calcula de forma automática un valor de umbral desde el histograma de la imagen bimodal. Para imágenes que no son bimodales, la binarización no será precisa.

Para esto, usamos la función **cv2.threshold()**, pero con un indicador adicional, **cv2.THRESH_OTSU**. Para el valor umbral, sólo usamos cero. Luego el algoritmo encuentra el valor umbral óptimo y lo regresa como la segunda salida, **retVal**. Si la umbralización de Otsu no se usa, **retVal** es igual al valor de umbral que se use.

Ejercicio 6.3.

En el ejercicio siguiente, la imagen de entrada posee mucho ruido. En el primer caso, aplicamos la umbralización global para un valor de 127. En el segundo caso, aplicamos la umbralización de Otsu de forma directa. En el tercer caso, filtramos la imagen con kernel gaussiano 5×5 para remover el ruido, luego aplicamos la umbralización de Otsu. Observa como el filtro que remueve el ruido mejora los resultados.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread('images\lena_noisy.jpg',0)

# umbral global
ret1,th1 = cv2.threshold(img,127,255,cv2.THRESH_BINARY)

# umbral de Otsu
ret2,th2 = cv2.threshold(img,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

# umbral de Otsu después del filtro Gaussiano
blur = cv2.GaussianBlur(img,(5,5),0)
ret3,th3 = cv2.threshold(blur,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
```

```
# plotear las imagenes y sus histogramas
images = [img, 0, th1, img, 0, th2, blur, 0, th3]
titles = ['Original Noisy Image','Histogram','Global Thresholding (v=127)
',
'Original Noisy Image','Histogram',"Otsu's Thresholding",
'Gaussian filtered Image','Histogram',"Otsu's Thresholding"]
miArray = np.arange(3)
for i in miArray:
  plt.subplot(3,3,i*3+1),plt.imshow(images[i*3],'gray')
  plt.title(titles[i*3]), plt.xticks([]), plt.yticks([])
  plt.subplot(3,3,i*3+2),plt.hist(images[i*3].ravel(),256)
  plt.title(titles[i*3+1]), plt.xticks([]), plt.yticks([])
  plt.subplot(3,3,i*3+3),plt.imshow(images[i*3+2],'gray')
  plt.title(titles[i*3+2]), plt.xticks([]), plt.yticks([])
plt.show()
```

5. ACTIVIDAD IV. OPERACIONES MORFOLÓGICAS

Las transformaciones morfológicas son algunas operaciones simples basadas en la forma de la imagen, que normalmente se aplican a imágenes binarias. Necesita dos entradas, una es nuestra imagen original, la segunda se llama elemento estructurante o núcleo (kernel) que decide la naturaleza de la operación. Dos operadores morfológicos básicos son Erosión y Dilatación. Luego, sus formas variantes como Apertura, Cierre, Gradiente, etc.

Los kernel usados son:

Kernel Rectangular

cv2.getStructuringElement(cv2.MORPH_RECT,(5,5))

```
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]], dtype=uint8)
```

Kernel Eliptico

cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))

```
array([[0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0]], dtype=uint8)
```

Kernel Cruz

cv2.getStructuringElement(cv2.MORPH_CROSS,(5,5))

```
array([[0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0],
       [1, 1, 1, 1, 1],
       [0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0]], dtype=uint8)
```

- **Erosión.**

  Similar a la convolución 2D, en el proceso de erosionado un kernel (elemento estructurante) se desliza a través de la imagen. Un píxel de la imagen original (1 ó 0) sólo se considerará 1 si todos los píxeles que caen dentro de la ventana del kernel son 1, de lo contrario se erosiona (se hace a cero). Por tanto, todos los píxeles cerca de los bordes de los objetos en la imagen serán descartados dependiendo del tamaño del kernel. Como consecuencia, el grosor o el tamaño de los objetos en primer plano disminuye o, en otras palabras, la región blanca disminuye en la imagen. Este procedimiento **es útil para eliminar pequeños ruidos blancos**, separar dos objetos conectados, etc. Para aplicar esta operación con OpenCV usaremos la función **cv2.erode(src, dst, kernel).**

  Para utilizar el elemento estructurante utilizaremos **cv2.getStructuringElement()** que devuelve la forma y el tamaño especificados de los elementos estructurante, los cuales pueden ser:

  - Rectangular: MORPH_RECT;
  - Forma de cruz: MORPH_CROSS;
  - Elíptica: MORPH_ELLIPSE;

  Ejercicio 6.4.
  En el ejercicio siguiente se aplica la operación morfológica de erosión con diferentes elementos estructurantes.

```python
import cv2
import numpy as np
#--leer la imagen
img = cv2.imread('images\A.png',0)
#--elemento estructurante
kernel = np.ones((7,7),np.uint8)
#kernel = cv2.getStructuringElement(cv2.MORPH_CROSS, (7,7))
#kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
#kernel=cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(3,3))
#--realizar la erosion
erosion = cv2.erode(img,kernel,iterations = 1)
#--mostrar resultados
cv2.imshow("original", img)
cv2.imshow("erosion", erosion)

cv2.waitKey()
```

- **Dilatación.**

El proceso de dilatación es justo lo opuesto a la erosión. Aquí, un elemento de píxel es ‘1’ si al menos un píxel de la imagen de los que caen dentro de la ventana del kernel es ‘1’. Por lo tanto, la dilatación aumenta el tamaño de los objetos de primer plano, es decir, la región blanca. Normalmente, en casos como la eliminación del ruido, la erosión es seguida de dilatación. La razón para esto es que, aunque la erosión elimina los ruidos blancos también encoge los objetos. Por tanto, para recuperar el tamaño inicial, este se dilata. La transformación de dilatación también es útil para unir partes rotas de un objeto.

Ejercicio 6.5.

A continuación, un ejemplo de cómo la dilatación funciona:

```
import cv2
import numpy as np
#--leer la imagen
img = cv2.imread('images\A.png',0)
#--elemento estructurante
kernel = np.ones((7,7),np.uint8)
#kernel = cv2.getStructuringElement(cv2.MORPH_CROSS, (7,7))
#kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
#kernel=cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(3,3))
#--realizar la erosion
#erosion = cv2.erode(img,kernel,iterations = 1)
dilatacion = cv2.dilate(img,kernel,iterations = 1)
#--mostrar resultados
cv2.imshow("original", img)
cv2.imshow("erosion", dilatacion)


cv2.waitKey()
```

- **Apertura.**
  La apertura es simplemente otro nombre para erosión seguida de dilatación. Como se explicó anteriormente, es útil para eliminar el ruido. En este caso se utiliza la función, **cv2.morphologyEx().**

  Ejercicio 6.6.
  Ejemplo de apertura.

```
import cv2
import numpy as np
#--leer la imagen
img = cv2.imread('images\A_noise1.png',0)
#--elemento estructurante
kernel = np.ones((7,7),np.uint8)
#kernel = cv2.getStructuringElement(cv2.MORPH_CROSS, (7,7))
```

```
#kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
#kernel=cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(3,3))
#--realizar la erosion
apertura = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)

#--mostrar resultados
cv2.imshow("original", img)
cv2.imshow("erosion", apertura)

cv2.waitKey()
```

- **Cierre**

  El cierre es el opuesto de apertura, es decir, dilatación seguida de erosión. Es útil para cerrar pequeños agujeros dentro de los objetos de primer plano, o pequeños puntos negros en el objeto. Vea un ejemplo a continuación:

  Ejercicio 6.7.

```
import cv2
import numpy as np
#--leer la imagen
img = cv2.imread('images\A_noise2.png',0)
#--elemento estructurante
kernel = np.ones((7,7),np.uint8)
#kernel = cv2.getStructuringElement(cv2.MORPH_CROSS, (7,7))
#kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
#kernel=cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(3,3))
#--realizar el cierre

cierre = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
#--mostrar resultados
cv2.imshow("original", img)
cv2.imshow("erosion", cierre)

cv2.waitKey()
```

- **Gradiente Morfológico**

  Es la diferencia entre la dilatación y la erosión de una imagen. El resultado se verá como el contorno del objeto.

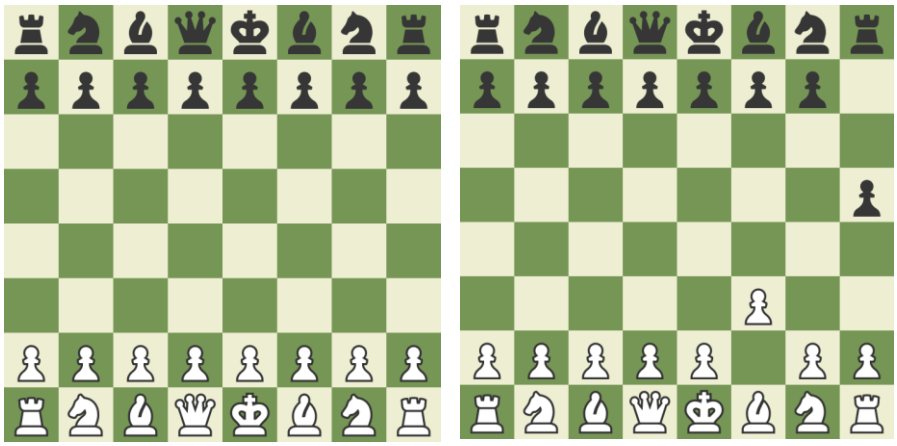
  Ejercicio 6.8.

```
import cv2
import numpy as np
#--leer la imagen
```

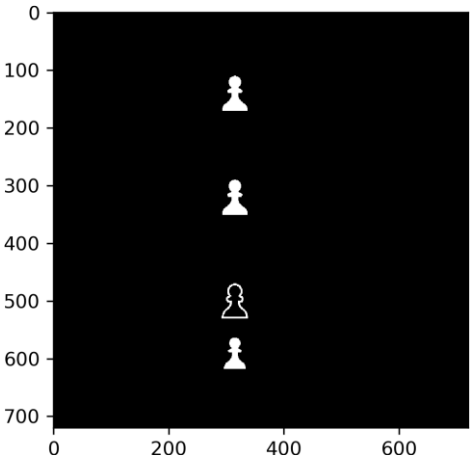
```
img = cv2.imread('images\A.png',0)
#--elemento estructurante
kernel = np.ones((7,7),np.uint8)
#kernel = cv2.getStructuringElement(cv2.MORPH_CROSS, (7,7))
#kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
#kernel=cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(3,3))
#--realizar el grqdiente
gradiente = cv2.morphologyEx(img, cv2.MORPH_GRADIENT, kernel)
#--mostrar resultados
cv2.imshow("original", img)
cv2.imshow("erosion", gradiente)

cv2.waitKey()
```

6. TAREA

Encontrar las piezas de ajedrez que se han movido usando dos imágenes tomadas de la misma partida.



La salida debe ser:



Nótese que, en la imagen anterior, las cuatro piezas que están presentes muestran las posiciones inicial y final de las dos únicas piezas que habían cambiado de posición en las dos imágenes.

7. REFERENCIAS BIBLIOGRÁFICAS

- Dawson-Howe, K. (2014). A Practical Introduction to Computer Vision With OpenCV, Wiley & Sons Ltd.
- Hafsa Asad, W. R., Nikhil Singh (2020). The Computer Vision Workshop. Birmingham U.K., Pack Publishing.
- Szeliski, R. (2011). Computer Vision Algorithms and Applications. London, Springer.