Universidad Nacional de San Antonio Abad del Cusco
Departamento Académico de Ing. Informática
VISION COMPUTACIONAL
Práctica Nº 01

INTRODUCCIÓN AL PROCESAMIENTO DE IMÁGENES CON OPENCV EN PYTHON

Iván C. Medrano Valencia

1. OBJETIVO.

- Conocer la estructura de datos de la librería **NumPy** para representar imágenes
- Conocer instrucciones básicas de **OpenCV** para el procesamiento de imágenes

2. ACTIVIDAD 1.

2.1. Arreglos NumPy.

Para utilizar la librería **NumPy** necesitamos importar el módulo **NumPy** usando el comando **import numpy** *as* **np**. Aquí, **np** se usa como alias. Esto significa que en lugar de escribir **numpy function_name**, podemos usar **np.function_name**.

Veremos cuatro formas de crear una matriz **NumPy**:

- Matriz llena de ceros: el comando **np.zeros**
- Matriz llena de unos: el comando **np.ones**
- Matriz llena de números aleatorios: el comando **np.random.rand**
- Matriz llena de valores especificados: el comando **np.array**.

Comencemos con los comandos **np.zeros** y **np.ones**. Hay dos argumentos importantes para estas funciones:

La forma de la matriz. Para una matriz 2D, esto es (número de filas, número de columnas).

• El tipo de datos de los elementos. De forma predeterminada, NumPy usa números de punto flotante para sus tipos de datos. Para las imágenes, usaremos enteros de 8 bits sin firmar: **np.uint8**.

La razón detrás de esto es que los enteros sin signo de 8 bits tienen un rango de 0 a 255, que es el mismo rango que se sigue para los valores de píxeles.

Por ejemplo, si queremos crear una matriz llena de ceros de tamaño 4x3. Podemos hacer esto usando **np.zeros (4,3)**. De manera similar, si queremos crear una matriz de 4x3 llena de unos, podemos usar **np.ones (4,3)**.

La función **np.random.rand**, por otro lado, solo necesita la forma de la matriz. Para una matriz 2D, se proporcionará como **np.random.rand (number_of_rows, number_of_columns)**.

Finalmente, para la función **np.array**, proporcionamos los datos como primer argumento y el tipo de datos como segundo argumento.

Una vez que tenga una matriz **NumPy**, puede usar **npArray.shape** para averiguar la forma de la matriz, donde **npArray** es el nombre de la matriz **NumPy**. También podemos usar **npArray.dtype** para mostrar el tipo de datos de los elementos en la matriz.

2.2. Creación de un arreglo NumPy.

1. Crear un archivo de código fuente de Python llamado **Ejercicio1.py**
   *
2. Primero, importe el módulo NumPy:

   ```
   import numpy as np
   ```

3. A continuación, creemos una matriz 2D NumPy con 5 filas y 6 columnas, llena de ceros:

   ```
   npArray = np.zeros((5,6))
   ```

4. Imprimamos la matriz que acabamos de crear:

   ```
   print(npArray)
   ```

   El resultado es el siguiente:

   ```
   [[0. 0. 0. 0. 0. 0.]
    [0. 0. 0. 0. 0. 0.]
    [0. 0. 0. 0. 0. 0.]
    [0. 0. 0. 0. 0. 0.]
    [0. 0. 0. 0. 0. 0.]]
   ```

5. A continuación, imprimamos el tipo de datos de los elementos de la matriz:

   ```
   print(npArray.dtype)
   ```

   La salida es: `float64`.

6. Finalmente, imprimamos la forma de la matriz:

   ```
   print(npArray.shape)
   ```

   La salida es: `(5, 6)`.

7. Imprima el número de filas y columnas en la matriz:

   ```
   print("Number of rows in array = {}"\
   .format(npArray.shape[0]))

   print("Number of columns in array = {}"\
   .format(npArray.shape[1]))
   ```

   La salida es como sigue:

```
Number of rows in array = 5
Number of columns in array = 6
```

8. Observe que la matriz que acabamos de crear usó números de punto flotante como tipo de datos. Creemos una nueva matriz con otro tipo de datos, un entero de 8 bits sin signo, y averigüemos su forma y tipo de datos:

```
npArray = np.zeros((5,6), dtype=np.uint8)
```

9. Utilice la función print () para mostrar el contenido de la matriz:

```
print(npArray)
```

La salida es como sigue:

```
[[0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]]
```

10. Imprima el tipo de datos de np.Array, de la siguiente manera:

```
print(npArray.dtype)
```

La salida es: `uint8`.

11. Imprima la forma de la matriz, de la siguiente manera

```
print(npArray.shape)
```

La salida es: **`(5, 6)`**.

12. Imprima el número de filas y columnas en la matriz, de la siguiente manera:

```
print("Number of rows in array = {}"\
.format(npArray.shape[0]))

print("Number of columns in array = {}"\
.format(npArray.shape[1]))
```

La salida es como sigue:

```
Number of rows in array = 5
Number of columns in array = 6
```

13. Ahora, crearemos matrices del mismo tamaño, es decir, (5,6), y el mismo tipo de datos (un entero de 8 bits sin signo) usando los otros comandos que vimos anteriormente. Creemos una matriz llena de unos:

```
npArray = np.ones((5,6), dtype=np.uint8)
```

14. Imprimamos la matriz que hemos creado:

```
print(npArray)
```

La salida es como sigue:

```
[[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 1 1 1 1 1]]
```

15. Imprimamos el tipo de datos de la matriz y su forma. Podemos verificar que el tipo de datos de la matriz es en realidad un uint8:

```
print(npArray.dtype)
print(npArray.shape)
```

La salida es como sigue:

```
uint8
(5, 6)
```

16. A continuación, imprimamos el número de filas y columnas en la matriz:

```
print("Number of rows in array = {}"\
.format(npArray.shape[0]))

print("Number of columns in array = {}"\
.format(npArray.shape[1]))
```

La salida es como sigue:

```
Number of rows in array = 5
Number of columns in array = 6
```

17. A continuación, crearemos una matriz llena de números aleatorios. Tenga en cuenta que no podemos especificar el tipo de datos mientras construimos una matriz llena de números aleatorios:

```
npArray = np.random.rand(5,6)
```

18. Como hicimos anteriormente, imprimamos la matriz para encontrar los elementos de la matriz:

```
print(npArray)
```

El resultado es el siguiente:

```
[[0.19959385 0.36014215 0.8687727 0.03460717 0.66908867 0.65924373]
 [0.18098379 0.75098049 0.85627628 0.09379154 0.86269739 0.91590054]
 [0,79175856 0,24177746 0,95465331 0,34505896 0,49370488 0,06673543]
 [0.54195549 0.59927631 0.30597663 0.1569594 0.09029267 0.24362439]
 [0.01368384 0.84902871 0.02571856 0.97014665 0.38342116 0.70014051]]
```

19. A continuación, imprimamos el tipo de datos y la forma de la matriz aleatoria:

```
print(npArray.dtype)
print(npArray.shape)
```

La salida es como sigue:

```
float64
(5, 6)
```

20. Finalmente, imprimamos el número de filas y columnas en la matriz:

```
print("Number of rows in array = {}"\
.format(npArray.shape[0]))

print("Number of columns in array = {}"\
.format(npArray.shape[1]))
```

La salida es como sigue:

```
Number of rows in array = 5
Number of columns in array = 6
```

21. Finalmente, creemos una matriz que se parezca a la que se muestra en la siguiente figura:

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

Fig. 1. Matriz Numpy

El código para crear e imprimir la matriz es el siguiente:

```
npArray = np.array([[1,2,3,4,5,6],
[7,8,9,10,11,12],
[13,14,15,16,17,18],
[19,20,21,22,23,24],
[25,26,27,28,29,30]],
dtype=np.uint8)
print(npArray)
```

La salida es la siguiente:

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]
[25 26 27 28 29 30]]
```

22. Ahora, al igual que en los casos anteriores, imprimiremos el tipo de datos y la forma del arreglo:

```
print(npArray.dtype)
print(npArray.shape)
print("Number of rows in array = {}"\
.format(npArray.shape[0]))
print("Number of columns in array = {}"\
.format(npArray.shape[1]))
```

La salida es como sigue:

```
uint8
(5, 6)
Number of rows in array = 5
Number of columns in array = 6
```

En este ejercicio, vimos cómo crear matrices NumPy usando diferentes funciones, cómo especificar sus tipos de datos y cómo mostrar su forma.

Más información sobre la librería NumPy puede encontrar en:

https://numpy.org/devdocs/user/quickstart.html

3. ACTIVIDAD 2

3.1. Introducción a OpenCV.

OpenCV, es la biblioteca de visión por computadora de código abierto más utilizada. Escrito principalmente en C++, también se usa comúnmente en Python y otros lenguajes de programación. A lo largo de los años, OpenCV ha pasado por múltiples revisiones y su versión actual es 4.5.2. (https://opencv.org/releases/)

Lo que la diferencia de otras bibliotecas de visión por computadora es el hecho de que es rápida y fácil de usar, brinda soporte para bibliotecas como QT y OpenGL, y lo más importante, proporciona aceleración de hardware para procesadores Intel.

Estas poderosas características hacen de OpenCV la elección perfecta para comprender los diversos conceptos de visión por computadora e implementarlos. Además de OpenCV, también usaremos NumPy para algunos cálculos básicos y Matplotlib para visualización cuando sea necesario.

3.2. Imágenes en OpenCV

OpenCV tiene su propia clase para representar imágenes: cv :: Mat. La parte "Mat" proviene del término “matriz” ya que las imágenes no son más que matrices.

Sabemos que cada imagen tiene tres atributos específicos de sus dimensiones: ancho, alto y número de canales. También sabemos que cada canal de una imagen es una colección de valores de píxeles que se encuentran entre 0 y 255.

Observe cómo el canal de una imagen comienza a parecerse a una matriz 2D. Entonces, una imagen se convierte en una colección de matrices 2D apiladas una encima de la otra.

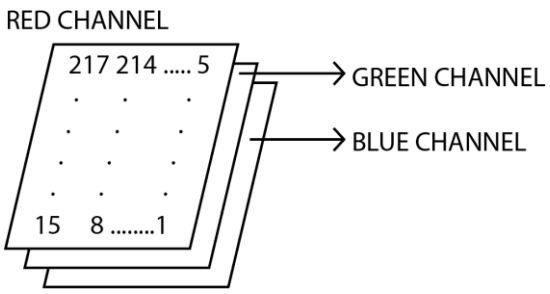

Fig. 2. Imagen como matrices 2D apiladas una encima de la otra

Las matrices 2D son suficientemente buenas para representar imágenes en scala de grises, pero sabemos que nuestras imágenes RGB no son como matrices 2D. No solo tienen una altura y una anchura; también tienen una dimensión adicional: el número de canales en la imagen. Es por eso que podemos referirnos a las imágenes RGB como matrices 3D.

Entonces para representar una imagen RGB a través de una matriz NumPy tenemos que agregar una dimensión adicional: el número de canales.

Como sabemos que las imágenes RGB tienen solo tres canales, la forma de las matrices NumPy se convierte en (número de filas, número de columnas, 3).

Además, debemos tener en cuenta que el orden de los elementos en forma de matrices NumPy sigue este formato: (número de filas, número de columnas, 3). Aquí, el número de filas equivale a la altura de la imagen, mientras que el número de columnas equivale al ancho de la imagen. Es por eso que la forma de la matriz NumPy puede también se representará como (alto, ancho, 3).

3.3. Funciones de OpenCV importantes

Podemos dividir las funciones de OpenCV que vamos a utilizar en las siguientes categorías:

- Leer una imagen
- Modificar una imagen
- Visualización de una imagen
- Guardar una imagen

La única función que usaremos para leer una imagen es **cv2.imread**. Esta función toma los siguientes argumentos:

- Nombre de archivo de la imagen que queremos leer / cargar
- Banderas para especificar en qué modo queremos leer la imagen

Si intentamos cargar una imagen que no existe, la función devuelve **None**. Esto se puede utilizar para comprobar si la imagen se leyó correctamente o no.

Actualmente, OpenCV admite formatos como .bmp, .jpg, .jpeg, .png, .tiff y tif. Para obtener la lista completa de formatos, puede consultar la documentación:

https://docs.opencv.org/4.2.0/d4/da8/group__imgcodecs.html#ga288b8b3da0892bd651fce07b3bbd3a56

Lo último en lo que debemos enfocarnos con respecto a la función cv2.imread es la bandera. Solo hay tres banderas que se usan comúnmente para leer una imagen en un modo específico:

- cv2.IMREAD_UNCHANGED: Lee la imagen tal cual es.
- cv2.IMREAD_GRAYSCALE: Lee la imagen en formato de escala de grises. Esto convierte cualquier imagen en color en escala de grises.

- cv2.IMREAD_COLOR: Este es el modo predeterminado y lee cualquier imagen como una imagen en color (modo RGB).

Hay tres funciones principales que usaremos con fines de visualización:

- Para mostrar una imagen, usaremos la función **cv2.imshow**. Se necesitan dos argumentos. El primer argumento es una cadena, que es el nombre de la ventana en la que vamos a mostrar la imagen. El segundo argumento es la imagen que queremos mostrar.
- Después de llamar a la función **cv2.imshow**, usamos la función **cv2.waitKey**. Esta función determina cuánto tiempo debe permanecer el control en la ventana. Si desea pasar al siguiente fragmento de código después de que el usuario presiona una tecla, puede proporcionar 0. De lo contrario, puede proporcionar un número que especifique el número de milisegundos que el programa esperará antes de pasar al siguiente fragmento de código. Por ejemplo, si desea esperar 10 milisegundos antes de pasar a la siguiente pieza de código, puede utilizar **cv2.waitKey (10).**
- Sin llamar a la función **cv2.waitKey**, la ventana de visualización no se verá correctamente. Pero después de pasar al siguiente código, la ventana seguirá abierta (pero parecerá que no responde). Para cerrar todas las ventanas de visualización, podemos usar la función **cv2.destroyAllWindows()**. No necesita argumentos. Se recomienda cerrar las ventanas de visualización una vez que ya no sean necesarias.

Finalmente, para guardar una imagen, usaremos la función **cv2.imwrite** de OpenCV. Se necesitan dos argumentos:

- Una cadena que especifica el nombre de archivo con el que queremos guardar la imagen.
- La imagen que queremos guardar

3.4. EJERCICIOS PRÁCTICOS

Ejercicio 1.1. Leer y mostrar una imagen.

```
#importar el módulo OpenCV
import cv2
# cargar imagen
img = cv2.imread("images/lion.jpg")
if img is None:
    print("Imagen no encontrada")
    # Display the image
cv2.imshow("Lion",img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Ejercicio 1.2. Mostrar propiedades de la imagen.

```
import cv2
# ruta de la imagen
image_path = "images/marsrover.png"
# Leer la imagen de esa dirección
image = cv2.imread(image_path)
# La imagen es un array NumPy
print("Dimensiones de la imagen: ", image.ndim)
print("Altura de la imagen: ", format(image.shape[0]))
```

```
print("Ancho de la imagen: ", format(image.shape[1]))
print("Canales de la imagen: ", format(image.shape[2]))
print("Tamaño en bytes del arreglo imagen: ", image.size)
# Mostrar la imagen y esperar hasta presionar una tecla
cv2.imshow("Mi imagen", image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Ejercicio 1.3. Manipular píxeles de una imagen.

```
import cv2
# dirección de la imagen
image_path = "images/marsrover.png"
# Leer o cargar la imagen de esa dirección
image = cv2.imread(image_path)

# Acceder al pixel en la ubicación (0,0)
(b, g, r) = image[0, 0]
print("Valores Red, Green y Blue del pixel (0,0): ", format((b, g, r)))

# Manipuar pixels y mostrar la imagen modificada
image[0:100, 0:100] = (255, 255, 0)
cv2.imshow("Imagen modificada", image)
cv2.waitKey(0)
```

Ejercicio 1.4. Modificar y guardar una imagen.

```
import cv2
# Dirección de la imagen
image_path = "images/marsrover.png"
# Leer la imagen
image = cv2.imread(image_path)
# set the start and end coordinates
# Establecer coordenadas iniciales y finales
# de la esquina sup. izq. y de inf. der. del rectángulo
start = (100,70)
end = (350,380)
# Establecer el color y el grosor de los bordes
color = (0,255,0)
thickness = 5
# Dibujar el rectángulo
cv2.rectangle(image, start, end, color, thickness)
# Grabar la imagen modificada son el rectángulo dibujado
cv2.imwrite("rectangle.jpg", image)
# Mostrar la imagen modificada
cv2.imshow("Rectangle", image)
cv2.waitKey(0)
```

4. 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.