



UNIVERSIDAD AUTONOMA
CHAPINGO

PROYECTO DE SERVICIO SOCIAL



APLICACIÓN DE ALGORITMOS DE SEGMENTACIÓN Y CLÚSTER EN IMÁGENES AGRÍCOLAS, MEDIANTE EL USO DE HERRAMIENTAS DE COMPUTO

MODULO 1: OPERACIONES BASICAS

AUTOR: LOPEZ ESTEBAN MIGUEL

FECHA: 01 DE JUNIO 2024

El módulo actual se centra en el manejo básico de imágenes, proporcionando una base sólida para el entendimiento de las operaciones fundamentales. Dichas operaciones son importantes para una adecuada preparación y preprocesamiento de las imágenes. Estas operaciones son fundamentales antes de aplicar algoritmos más complejos, de manera que estos puedan ser más eficientes. Este módulo servirá como una revisión bibliográfica de las operaciones básicas, de manera que se proporcionan explicaciones teóricas y ejemplos del funcionamiento de cada operación.

INDICE

1. [librerias](#)
 - 1.1. [Opencv](#)
 - 1.2. [Matplotlib](#)
 - 1.3. [NumPy](#)
 - 1.4. [Importacion](#)
2. [Leer imagenes con opencv](#)
3. [Visualizacion de imagenes](#)
 - 3.1 [visualizacion con opencv](#)
 - 3.2. [Visualizacion con matplotlib](#)
4. [Imagen BGR a RGB](#)
5. [Propiedades de una imagen](#)
 - 5.1. [Dimensiones de una imagen](#)
 - 5.2. [Altura, ancho y numero de canales en una imagen](#)
 - 5.3. [Numero de pixeles en una imagen](#)
 - 5.4. [Tipos de datos en una imagen](#)
 - 5.4.1. [Datos uint8](#)

- 5.4.2. Datos uint16
- 5.4.3. Datos float32
- 5.4.4. Ejemplo
- 5.5. Tamaño de una imagen
- 6. Redimensionamiento de imágenes
- 7. Canales de una imagen RGB
 - 7.1. Separar los canales de una imagen RGB
 - 7.2. Creacion de imagenes
 - 7.3. Fusion de canales
 - 7.3.1. Apagar canales en una imagen
 - 7.3.2. Modificar canales
- 8. Estadisticas de pixeles
 - 8.1. Minimo y maximo en pixeles
 - 8.2. Media en pixeles
 - 8.3. Desviacion estandar en pixeles
- 9. Acceso y manipulacion de pixeles
 - 9.1. Valores RGB en los pixeles
 - 9.2. Manipulacion de pixeles en un rango (ROI)
 - 9.3. manipulacion de todos los pixeles
 - 9.3.1. Inversion de colores
 - 9.3.2. Imagenes en escala de grises
- 10. Referencias

librerías

En este modulo usaremos 3 librerías básicas para el procesamiento de imágenes en Python, estas librerías son:

- 1. Opencv
- 2. Matplotlib
- 3. Numpy

OPENCV

Nuestra primer librería opencv (open source computer visión library) es una biblioteca de código abierto que contiene diferentes algoritmos optimizados para el procesamiento de imágenes, esta librería proporciona diferentes herramientas para leer, manipular y procesar imágenes, estas herramientas son de gran utilidad para este modulo por lo cual estará presente casi siempre en cualquier proceso que trabaje con imágenes.

MATPLOTLIB

Matplotlib es una librería de visualización, la cual es útil para crear gráficos y figuras interactivas, en esta modulo será útil para visualizar las imágenes con las que trabajamos sin la necesidad de abrir una ventana emergente, esta característica la hace de gran utilidad para trabajar en notebooks para mostrar los resultados.

NUMPY

Numpy es una librería que sirve como soporte para trabajar con matrices, su importancia radica en que las imágenes son en esencia un arreglo de matrices, dado que estaremos trabajando con imágenes esta librería nos permite optimizar nuestras operaciones con las matrices de nuestras imágenes de manera que nuestro procesamiento sea más rápido.

IMPORTACION

Para usar estas librerías en nuestro código debemos de importarlas, esto se logra con el comando “import” seguido de el nombre de la librería a importar, para esto debemos tener en claro el nombre de cada librería.

Los nombres de cada librería son los siguientes:

1. Opencv = cv2
2. Matplotlib = matplotlib.pyplot
3. Numpy = numpy

Con estos nombres podremos hacer el llamado de las librerías siempre y cuando ya las tengamos descargadas.

Sin embargo, podemos observar que para la segunda y tercera librería se hace uso del comando “as”, este comando sirve para renombrar, dado que el nombre “matplotlib.pyplot” es muy largo y aunado a que es una librería que se mandara a llamar seguido vuelve tedioso el uso de esta librería, sin embargo con el comando “as” podemos cambiarle el nombre a “plt” el cual es más corto haciéndolo fácil de usar en el código.

Para el caso de numpy su nombre no es tan largo a comparación del caso anterior, sin embargo, es normal reducir su nombre a “np” cosa queharemos en este módulo con la intención de ahorrar espacio y tiempo al momento de hacer el llamado de la librería.

Para el caso de opencv dado que su nombre es “cv2” resulta un nombre muy corto por lo cual no hay necesidad de acortarlo.

```
In [ ]: import cv2  
import matplotlib.pyplot as plt  
import numpy as np
```

Leer imagenes con opencv

En el procesamiento de imágenes el primer paso es la lectura de las imágenes, en este apartado cargaremos la imagen con la que trabajaremos en el programa, en este apartado existen diferentes maneras de cargar imágenes al sistema, un ejemplo de ello es la librería pillow que es muy conocida en el procesamiento de imágenes, sin embargo, su funcionalidad es menor respecto a opencv, por lo cual solo trabajaremos con opencv.

Para cargar imágenes en opencv es muy simple, una vez que ya tengamos importadas las librerías lo que haremos será llamar a la librería “cv2” de la cual llamaremos la función “imread” que es la encargada de leer la imagen y cargarla, esta función recibe como parámetro una ruta, dependiendo donde tengamos almacenada la imagen colocaremos la ruta de la imagen su nombre y extensión, sin embargo si la imagen esta guardada en la misma carpeta que el programa “.py” lo único que tendremos que colocar será el nombre y extensión de nuestra imagen, la cual deberá ir siempre entre comillas, ya que se trata de un carácter.

en nuestro caso la imagen esta contenida en la misma carpeta que este archivo, por lo cual mandamos a llamar a la función y colocamos el nombre y extensión de nuestra imagen, como se muestra a continuación. El resultado de este llamado lo guardaremos en una variable, en nuestro caso sera en la variable imagen.

```
In [ ]: imagen = cv2.imread('fcolores.jpg')
```

visualizacion de imagenes

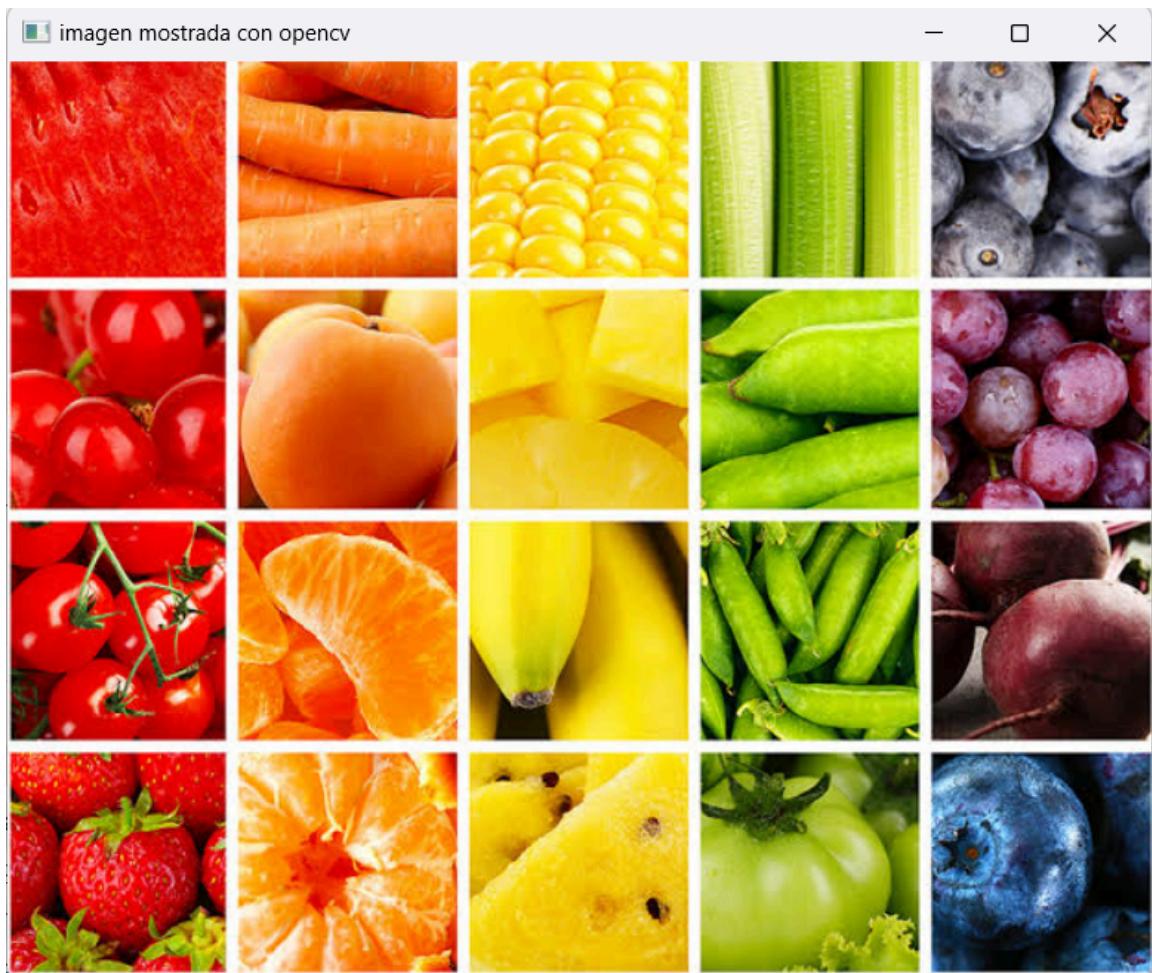
Una vez que la imagen a sido leída y cargada, el siguiente paso es visualizarla, para observar que se halla cargado de manera adecuada, este paso es crucial ya que podemos observar nuestra imagen y en caso de haberle hecho cambios podemos observar estos, en caso de estar llevando la imagen por un tratamiento, podemos ver los cambios que sufre nuestra imagen con nuestras operaciones, de esta manera podemos saber si nos estamos acercando a nuestro objetivo con la imagen.

Para visualizar una imagen existen diferentes métodos, a lo largo de este módulo solo se usara un método, sin embargo mostraremos 2 maneras diferentes para mostrar las imágenes.

VISUALIZACION CON OPENCV

La visualización de imágenes con opencv es demasiado sencilla, solo tenemos que usar la función “cv2.imshow()” y en los parametros colocar el título que le pondremos a la imagen, y la imagen que vamos a mostrar. Después colocamos “cv2.waitKey()” la cual sirve como una pausa para que podamos visualizar la imagen sin que se cierre repentinamente, cuando queramos cerrar la ventana solo la cerramos a pulsamos “0”, por ultimo “cv2.destroyAllWindows()” cierra todas las ventanas que se hallan abierto.

```
In [ ]: cv2.imshow('imagen mostrada con opencv',imagen)
cv2.waitKey()
cv2.destroyAllWindows()
```

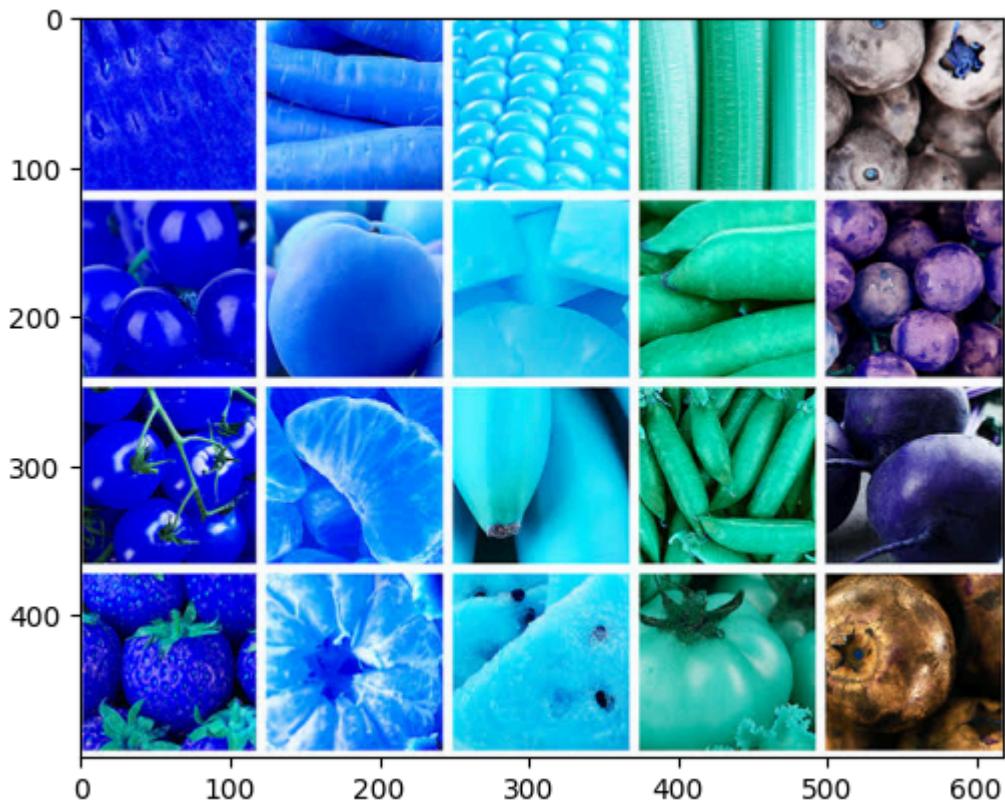


VISUALIZACION CON MATPLOTLIB

Para visualizar las imágenes con matplotlib haremos uso de la función “imshow()” la cual creamos y explicamos en el apartado de funciones, por lo tanto en este apartado simplemente la llamaremos e introduciremos los parámetros, en este caso la imagen que vamos a mostrar y el título.

```
In [ ]: plt.imshow(imagen)
```

```
Out[ ]: <matplotlib.image.AxesImage at 0x1e961507890>
```



Podemos observar que nuestra imagen se ve diferente a la imagen mostrada con opencv, esto se debe a la manera como cada biblioteca lee las imágenes.

La imagen con la que estamos trabajando tiene 3 canales (rojo, verde, azul), opencv lee las imágenes en el siguiente orden: azul, verde, rojo. Este orden se llama BGR, sin embargo matplotlib lee las imágenes en el siguiente orden: rojo , verde, azul. Este orden se llama RGB.

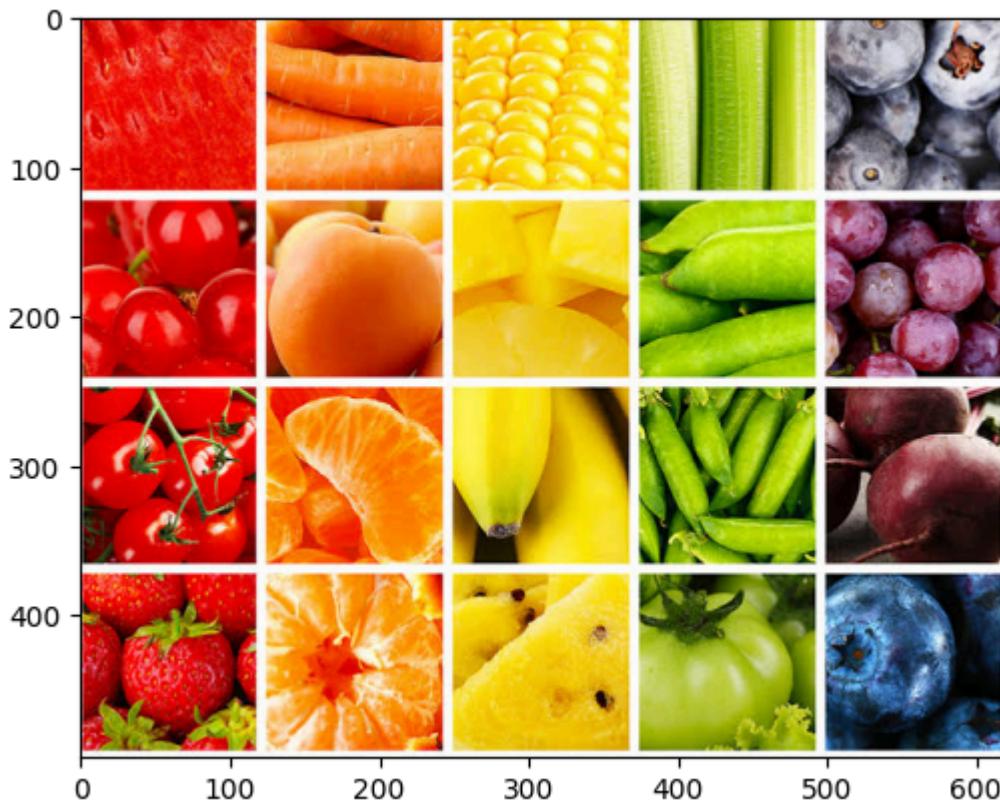
Como la imagen la hemos cargado con opencv esta tiene el orden BGR por lo que al leerla con la librería de matplotlib interpreta la imagen a su manera (RGB), lo cual nos muestra una imagen con colores diferentes a los normales, sin embargo este no es un gran problema ya que en el siguiente tema se corregirá la imagen.

Imagen BGR a RGB

Como notamos en el tema anterior, la visualización de la imagen con matplotlib se realiza de manera incorrecta debido a la errónea lectura de la imagen, para corregir este aspecto vamos a intercambiar los canales de la imagen con la función “cvtColor” de manera que podamos leerla de manera correcta. podemos observar como la imagen se muestra con sus colores correspondientes, lo cual nos indica que hemos intercambiado los canales de manera adecuada

```
In [ ]: imagen_rgb = cv2.cvtColor(imagen, cv2.COLOR_BGR2RGB)
plt.imshow(imagen_rgb)
```

```
Out[ ]: <matplotlib.image.AxesImage at 0x1e962719ac0>
```



Propiedades de una imagen

Una imagen tiene consigo diferentes propiedades, a continuación vamos a desglosar como podemos hallar mediante programación algunas de ellas, y saber que es lo que nos quiere decir cada una.

DIMENSIONES DE UNA IMAGEN

Para saber que significan el número de dimensiones en nuestra imagen debemos tener en claro que nuestra imagen es un arreglo de matrices, estas matrices tienen una altura y un ancho, lo cual indica que nuestra imagen tiene 2 dimensiones, pero esto solo sucede cuando nuestra imagen tiene un solo canal, en el caso de las imágenes RGB estas tienen 3 canales, por ende nuestra imagen tiene alto, ancho y profundidad.

La profundidad quiere decir que nuestra imagen consta de la superposición de varios arreglos de matrices, los cuales al juntarse forman la imagen que vemos, sin embargo, cuando la imagen esta compuesta por un solo canal esta no cuenta con esta dimensión lo cual nos dice que solo esta formada por un arreglo de matrices.

```
In [ ]: dimensiones = imagen_rgb.ndim
print(f'Nuestra imagen tiene {dimensiones} dimensiones')
```

Nuestra imagen tiene 3 dimensiones

En este caso observamos qe nuestra imagen tiene 3 dimensiones lo cual es correcto dado que introducimos una imagen con 3 canales, por lo cual nuestra imagen tiene un alto, ancho y una profundidad (canales).

ALTURA, ANCHO Y NUMERO DE CANALES EN UNA IMAGEN

Como se observó en el punto anterior, nuestra imagen consta de 3 dimensiones, y en este apartado vamos a descubrir estas 3 dimensiones, las cuales son:

1. Alto
2. Ancho
3. Numero de canales

Estas tres propiedades las podemos descubrir con la misma función “shape” la cual arrojara estos datos de nuestra imagen.

```
In [ ]: alto, ancho, ncanales = imagen_rgb.shape  
print(f'Nuestra imagen tiene {alto} pixeles de alto, {ancho} pixeles de ancho y con
```

Nuestra imagen tiene 496 pixeles de alto, 618 pixeles de ancho y consta de 3 canales

NUMERO DE PIXELES EN UNA IMAGEN

Teniendo en cuenta el alto y ancho de nuestra imagen podemos calcular el numero de pixeles que conforman a nuestra imagen, esto a través de una simple operación, multiplicaremos estas propiedades de nuestra imagen y obtendremos el numero de pixeles que conforman a nuestra imagen.

```
In [ ]: npixeles = alto * ancho  
print(f'Nuestra imagen esta conformada por {npixeles} pixeles')
```

Nuestra imagen esta conformada por 306528 pixeles

TIPOS DE DATOS EN UNA IMAGEN

Cada píxel en una imagen es definido en un rango de valor, estos rangos definen el tipo de datos que conforman a la imagen y son cruciales debido a que son los encargados de almacenar y manipular los datos en la memoria, generalmente encontramos 3 tipos de datos:

1. Uint8
2. Uint16
3. Float32

DATOS UINT8

El dato uint8 es generalmente el mas usado, ya que es el que contiene la mayoría de las imágenes, debido a que contiene valores de 0 a 256, es común usarlo en imágenes que se encuentran en escala de grises o en imágenes RGB.

Este tipo de dato ocupa muy poco espacio en la memoria, llegando a ocupar solamente 1 byte por pixel, lo cual nos permite un procesamiento rápido, en el caso de nuestra imagen al estar en RGB

tendrá 1 valor uint8 por cada canal de la imagen, por lo que cada pixel de nuestra imagen constará de 3 valores de este tipo.

DATOS UINT16

Los valores uint16 son datos un poco mas pesados que contienen un rango de 0 a 65,535, por lo cual estamos hablando de imágenes con una alta profundidad de color, este tipo de dato es útil en imágenes que requieren cierto grado de detalle.

DATOS FLOAT32

Los datos de tipo float32 son imágenes muy avanzadas que comprenden valores de -3.4e38 a 3.4e38, lo cual los hace útil en imágenes medicas o científicas, por lo cual este tipo de datos permite una precisión demasiada alta al momento de realizar operaciones con estas.

EJEMPLO

Para hallar el tipo de dato de nuestra imagen usamos la función “dtype” la cual nos arrojara el tipo de datos del que se compone nuestra imagen.

El cual en este caso es un tipo de dato uint8 debido a que estamos trabajando con una imagen RGB sencilla.

```
In [ ]: datatype = imagen_rgb.dtype  
print(f'La imagen esta formado por pixeles de tipo: {datatype}')
```

La imagen esta formado por pixeles de tipo: uint8

Tamaño de una imagen

Un dato fundamental de las imágenes es su tamaño, es decir la cantidad de memoria que ocupa la imagen en el sistema, este dato es fácil de calcular a partir de otros datos ya recopilados. Para poder calcular este aspecto tenemos que tener conocimiento de el numero de pixeles que tiene nuestra imagen, el numero de canales de esta, y el tipo de datos que maneja. Con estos datos simplemente hacemos la siguiente operación:

$$\text{Tamaño} = \text{npixeles} * \text{canales de la imagen} * \text{tipo de dato}$$

Hasta el momento sabemos que el numero de pixeles se obtiene al multiplicar el alto por el ancho de la imagen, lo cual lo multiplicamos por el numero de canales que contenga nuestra imagen, hasta este momento tenemos el total de elementos que conforman nuestra imagen, pero cada uno de estos elementos ocupan un espacio en memoria, cuando manejamos datos uint8 estos datos ocupan solamente 1 byte de memoria, por lo que multiplicamos lo que llevábamos por 1 y obtenemos el espacio de memoria que ocupa nuestra imagen.

Sin embargo podemos simplificar este proceso con la función “ nbytes” la cual nos mostrara el resultado de la operación explicada anteriormente, lo cual se muestra en el siguiente ejemplo.

```
In [ ]: tamaño_bytes = imagen_rgb.nbytes  
print(f'La imagen tiene un tamaño de {tamaño_bytes} bytes')
```

La imagen tiene un tamaño de 919584 bytes

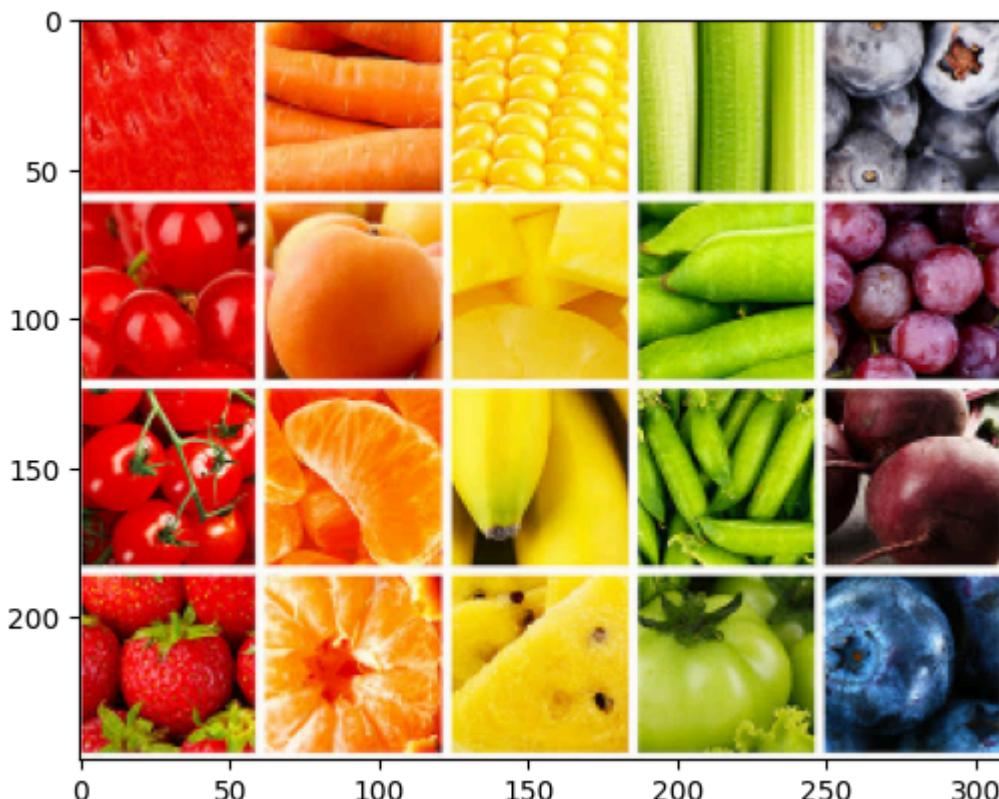
REDIMENSIONAMIENTO DE IMAGENES

El redimensionamiento es una parte crucial del tratamiento de imágenes, ya que este paso nos permite reducir o ampliar el tamaño de una imagen, de manera que hagamos más fácil el trabajar con estas, al disminuir las medidas de una imagen podemos aumentar la rapidez de procesamiento, lo cual nos permite ser mas productivos en nuestros programas.

Para redimensionar una imagen tenemos el siguiente ejemplo.

```
In [ ]: #creamos una variable donde almacenamos los nuevos tamaños de nuestra imagen  
new_tamaño = (ancho//2,alto//2)  
#usamos la función de cv2 para redimensionar nuestra imagen, recibe como parámetros  
imagen_redimensionada = cv2.resize(imagen_rgb,new_tamaño)  
#mostramos la imagen  
plt.imshow(imagen_redimensionada)
```

Out[]: <matplotlib.image.AxesImage at 0x1e9648a6870>



En este caso no notamos un cambio en la imagen mostrada, esto se debe a que matplotlib ajusta la imagen de manera automática a la ventana, por lo cual vemos la imagen del mismo tamaño, sin embargo observaremos sus propiedades para ver en que ha cambiado.

```
In [ ]: #dimensiones de la imagen
dimensiones_new = imagen_redimensionada.ndim
#tamaño alto y canales
alto_new, ancho_new,ncanales_new = imagen_redimensionada.shape
#tipo de datos
datatype_new = imagen_redimensionada.dtype
#numero de pixeles
npixeles_new = alto_new*ancho_new
#tamaño en bytes
tamaño_bytes_new = imagen_redimensionada nbytes

print(f'Dimensiones      normal: {dimensiones}      redimensionado:{dimensiones_new}')
print(f'Alto              normal: {alto}          redimensionado:{alto_new}')
print(f'Ancho             normal: {ancho}         redimensionado:{ancho_new}')
print(f'canales           normal: {ncanales}       redimensionado:{ncanales_new}')
print(f'tipo de dato      normal: {datatype}       redimensionado:{datatype_new}')
print(f'num pixeles       normal: {npixeles}        redimensionado:{npixeles_new}')
print(f'Tamaño            normal: {tamaño_bytes}    redimensionado:{tamaño_bytes_ne}

Dimensiones      normal: 3      redimensionado:3
Alto              normal: 496     redimensionado:248
Ancho             normal: 618     redimensionado:309
canales           normal: 3      redimensionado:3
tipo de dato      normal: uint8    redimensionado:uint8
num pixeles       normal: 306528   redimensionado:76632
Tamaño            normal: 919584   redimensionado:229896
```

Como observamos solo algunas propiedades de nuestra imagen han cambiado, vamos a desglosar cada propiedad.

1. Nuestras dimensiones siguen siendo las mismas, ya que nuestra imagen sigue teniendo alto, ancho y un numero de canales.
2. El alto de nuestra imagen a cambiado dado que directamente modificamos esta propiedad.
3. Al igual que el alto el ancho a cambiado debido a que fue el atributo que modificamos.
4. El numero de canales sigue igual dado que no hicimos cambios en este aspecto.
5. Nuestra imagen sigue manteniendo el mismo tipo de dato uint8.
6. Dado que redujimos la altura y ancho de nuestra imagen, el número de pixeles de nuestra imagen ha cambiado, y se ha reducido respecto a la imagen original, y esto se debe a que la imagen ahora es mas pequeña, por lo cual la imagen tiene menos pixeles.
7. Debido a que nuestra imagen tiene menos pixeles el tamaño de nuestra imagen a reducido su tamaño, por lo cual podemos comprobar que redimensionar una imagen nos ayuda a disminuir el espacio de memoria que ocupa una imagen, lo cual nos ayuda a poder hacer procesamientos más rápidos.

CANALES DE UNA IMAGEN RGB

Como hemos explicado anteriormente una imagen es un arreglo de matrices, en este caso un canal es una matriz bidimensional que contiene información sobre un color en cada pixel de una imagen, si la imagen esta constituida por un solo canal podemos observar una imagen gris, sin embargo cuando

sobreponemos las matrices de cada canal obtenemos una imagen con color, en este proyecto trabajaremos con imágenes RGB , es decir imagen con 3 canales, uno rojo, uno verde y uno azul, los cuales al sobreponerse entre si generan diferentes tonalidades que representan una imagen clásica que podemos observar en cualquier lugar.

SEPARAR LOS CANALES DE UNA IMAGEN RGB

En Python podemos separar los canales de una imagen para poder analizar cada canal por separado, esto se muestra a continuación.

```
In [ ]: #separamos los canales de nuestra imagen en cada variable almacenaremos un canal
r,g,b = cv2.split(imagen_redimensionada)

#mostramos los canales y la imagen original
fig, axs = plt.subplots(1,4,figsize=(20,5))

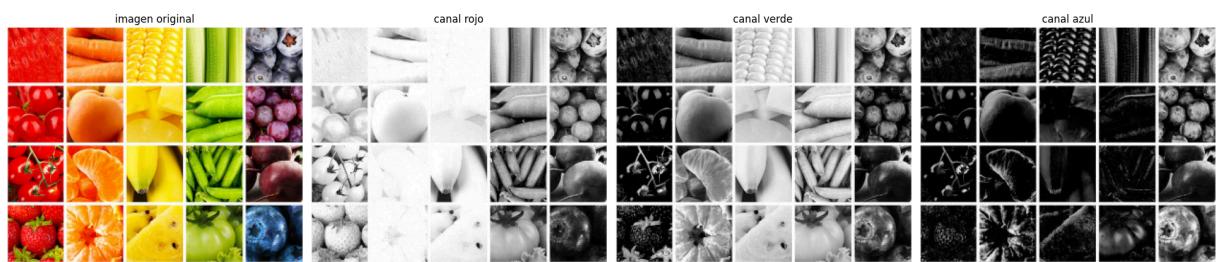
axs[0].imshow(imagen_redimensionada)
axs[0].set_title('imagen original')
axs[0].axis('off')

axs[1].imshow(r,cmap ='gray')
axs[1].set_title('canal rojo')
axs[1].axis('off')

axs[2].imshow(g, cmap ='gray')
axs[2].set_title('canal verde')
axs[2].axis('off')

axs[3].imshow(b, cmap ='gray')
axs[3].set_title('canal azul')
axs[3].axis('off')

plt.tight_layout()
plt.show()
```



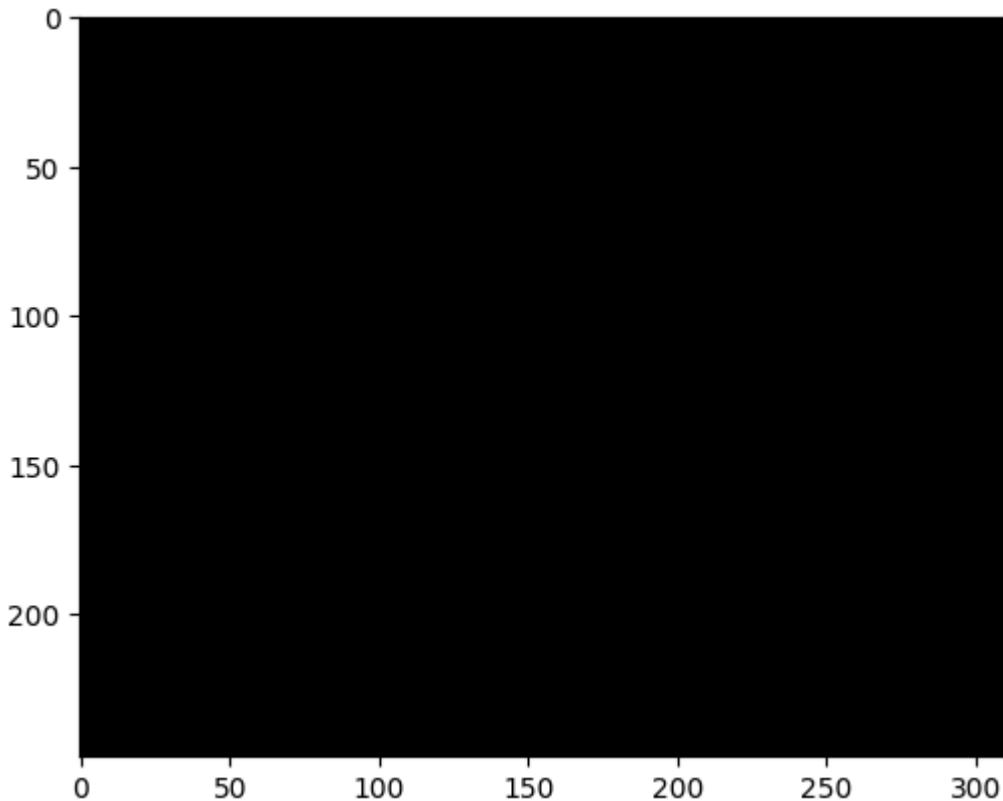
Ahora podemos observar los canales separados, los cuales se notan en un tono gris, sin embargo esto representa la intensidad de cada color en cada capa, por ejemplo en el canal rojo observamos como la parte de la izquierda está marcada en un tono muy claro, esto significa que en esta zona el color rojo tiene una amplia presencia, sin embargo del lado derecho observamos que la imagen se hace mas oscura, esto nos dice que en esta zona el color rojo tiene poca participacion, así podemos analizar cada canal de la imagen, para saber en que zonas tiene mayor presencia cada color.

CREACION DE IMAGENES

Hasta este punto tenemos claro que una imagen es un arreglo de matrices, sin embargo ¿si creamos una matriz, puedo verlo en forma de imagen?, la respuesta es sí, podemos hacer una matriz y leerla de manera que la podemos ver como una imagen. A continuación lo que vamos a hacer es crear una matriz que tendrá el mismo tamaño que la imagen que estamos trabajando, después con numpy llenaremos nuestra matriz, podemos llenarla con cualquier valor que queramos, sin embargo en este caso vamos a hacer una matriz llena de ceros lo cual nos mostraría una imagen completamente en negro, y mencionar que debemos asignarle a nuestra matriz su tipo de dato, como hemos venido trabajando vamos a usar uint8, es decir que podemos darle valores a nuestra matriz desde 0 hasta el 255.

```
In [ ]: black = np.zeros(imagen_redimensionada.shape[:2], dtype=np.uint8)
plt.imshow(black,cmap='gray')
```

```
Out[ ]: <matplotlib.image.AxesImage at 0x1e95f1c54c0>
```



FUSION DE CANALES

Una vez visto el tema de separación de canales veremos la manera de unir las matrices de nuevo, sin embargo observaremos como al modificar una capa, podemos observar cambios en nuestra imagen

APAGAR CANALES EN UNA IMAGEN

En el tema anterior creamos una imagen completamente en negro, esta imagen la vamos a usar para mostrar cada canal de nuestra imagen, nuestra imagen en negro funcionara como un apagador, es decir que nuestra imagen tendrá 3 canales, pero apagaremos 2 canales y solo mostraremos uno, se podría pensar que es lo que vimos al separar los canales de la imagen, sin embargo, podremos notar algunos cambios.

```
In [ ]: """
para unir canale usamos la funcion merge, en la cual introduciremos las imagenes a
en este caso colocaremos el canal en su lugar correspondiente, y en los 2 lugares s
la imagen que creamos anteriormente (black)
"""

rojo = cv2.merge([r,black,black])
verde = cv2.merge([black,g,black])
azul = cv2.merge([black,black,b])

#mostramos la imagen original y las imagenes generadas al apagar Los canales
fig, axs = plt.subplots(3,3,figsize=(20,10))

#imagen origina
axs[0,0].imshow(imagen_redimensionada)
axs[0,0].set_title('imagen original')
axs[0,0].axis('off')

#canal en negro (imagen creada para apagar canales)
axs[0,1].imshow(black,cmap='gray')
axs[0,1].set_title('capa en negro creada')
axs[0,1].axis('off')

axs[0,2].axis('off')

#canales separados
axs[1,0].imshow(r,cmap='gray')
axs[1,0].set_title('canal rojo')
axs[1,0].axis('off')

axs[1,1].imshow(g,cmap='gray')
axs[1,1].set_title('canal verde')
axs[1,1].axis('off')

axs[1,2].imshow(b,cmap='gray')
axs[1,2].set_title('canal azul')
axs[1,2].axis('off')

#imagenes con un canal activo y 2 desactivados
axs[2,0].imshow(rojo)
axs[2,0].set_title('canal rojo activo')
axs[2,0].axis('off')

axs[2,1].imshow(verde)
axs[2,1].set_title('canal verde activo')
axs[2,1].axis('off')

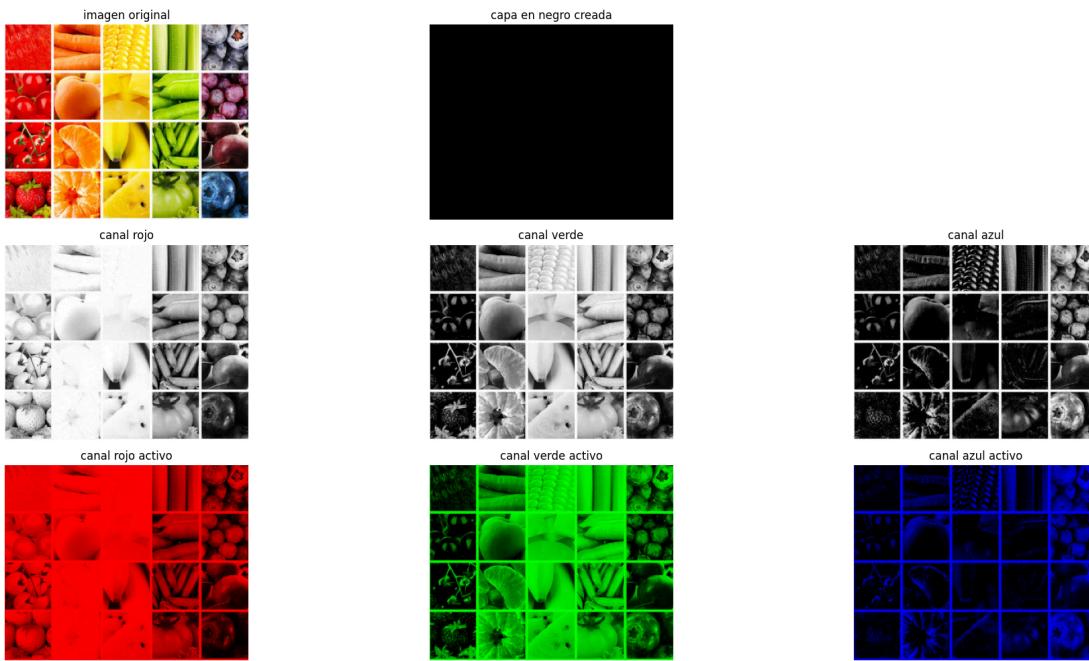
axs[2,2].imshow(azul)
axs[2,2].set_title('canal azul activo')
```

```

axs[2,2].axis('off')

plt.tight_layout()
plt.show()

```



Como podemos observar al desactivar algunos canales en una imagen nos permite observar de manera clara la presencia de un solo canal en la imagen, al separar los canales solo observamos tonalidades grises, pero cuando desactivamos los canales podemos observar la presencia del canal en la imagen.

MODIFICAR CANALES

Antes de unir los canales en una imagen podemos hacer modificaciones a cada canal con la intención de resaltar algunas características en la imagen final , en este caso solo invertiremos la capa roja, y observaremos el cambio que se produce en nuestra imagen.

```

In [ ]: #modificamos un canal para notar las diferencias
r_invert = cv2.bitwise_not(r)
#unimos las capas con la capa modificada
fusion = cv2.merge([r_invert,g,b])

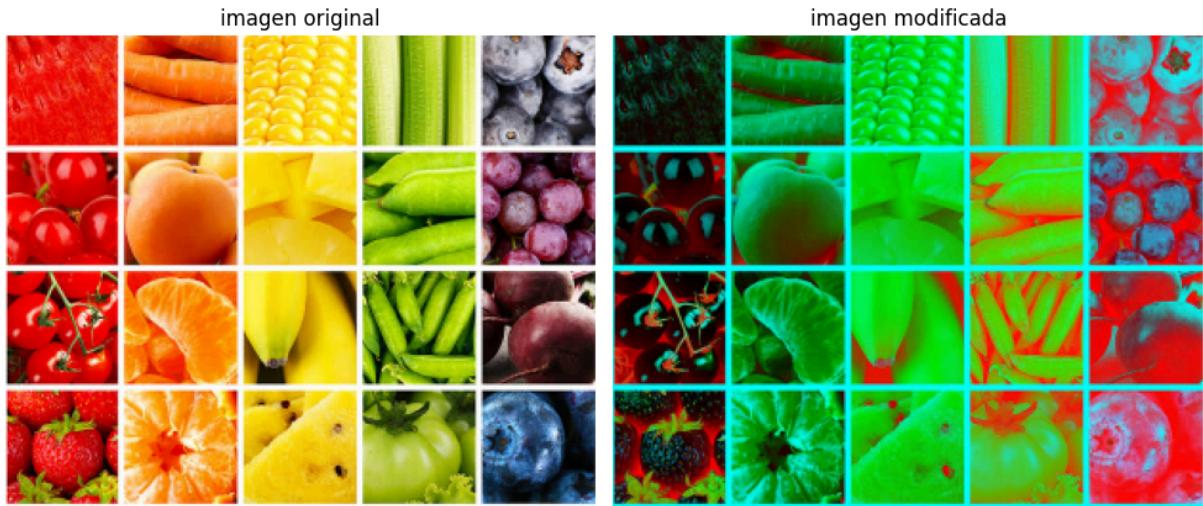
#mostramos la imagen original y la imagen modificada
fig, axs = plt.subplots(1,2,figsize=(10,10))

#imagen original
axs[0].imshow(imagen_redimensionada)
axs[0].set_title('imagen original')
axs[0].axis('off')

axs[1].imshow(fusion)
axs[1].set_title('imagen modificada')
axs[1].axis('off')

```

```
plt.tight_layout()  
plt.show()
```



Podemos observar como al invertir el color rojo hicimos que los elementos que naturalmente son rojos cambien su color, mientras que aquellos elementos donde la presencia de este color no era significativa cambiaron su color a un tono mas rojizo.

ESTADISTICAS DE PIXELES

Este apartado se centra en hacer operaciones estadísticas con los valores de los pixeles, con la intención de hacer algunas interpretaciones de nuestra imagen, cada operación tiene una finalidad que se desglosara a continuación.

MINIMO Y MAXIMO EN PIXELES

Este punto es sencillo consiste en hallar aquellos pixeles de la imagen que tengan el valor más pequeño y el mas alto, de igual manera hallaremos su ubicación en la imagen, esto se hace para conocer en que punto tenemos colores con mayor intensidad o en su caso con menor intensidad.

En nuestro ejemplo hallaremos el valor de los pixeles con mayor y menor intensidad en cada canal de una imagen, y hallaremos su ubicación con la intención de saber en que punto nuestro canal esta más concentrado.

```
In [ ]: #hallamos el valor minimo, maximo y su ubicacion, respecto a cada canal  
minvalr,maxvalr,minlocr,maxlocr = cv2.minMaxLoc(r)  
minvalg,maxvalg,minlocg,maxlocg = cv2.minMaxLoc(g)  
minvalb,maxvalb,minlocb,maxlocb = cv2.minMaxLoc(b)  
  
#mostramos en consola los valores hallados  
print(f'En la capa roja el valor minimo es:{minvalr} ubicado en:{minlocr}, el valor  
print(f'En la capa verde el valor minimo es:{minvalg} ubicado en:{minlocg}, el valo  
print(f'En la capa azul el valor minimo es:{minvalb} ubicado en:{minlocb}, el valor
```

```

En la capa roja el valor minimo es:0.0 ubicado en:(283, 67), el valor maximo es:255.
0 ubicado en(61, 0)
En la capa verde el valor minimo es:0.0 ubicado en:(2, 2), el valor maximo es:255.0
ubicado en(122, 0)
En la capa azul el valor minimo es:0.0 ubicado en:(23, 0), el valor maximo es:255.0
ubicado en(248, 0)

```

```

In [ ]: #mostramos los canales
fig, axs = plt.subplots(1,3,figsize=(10,5))

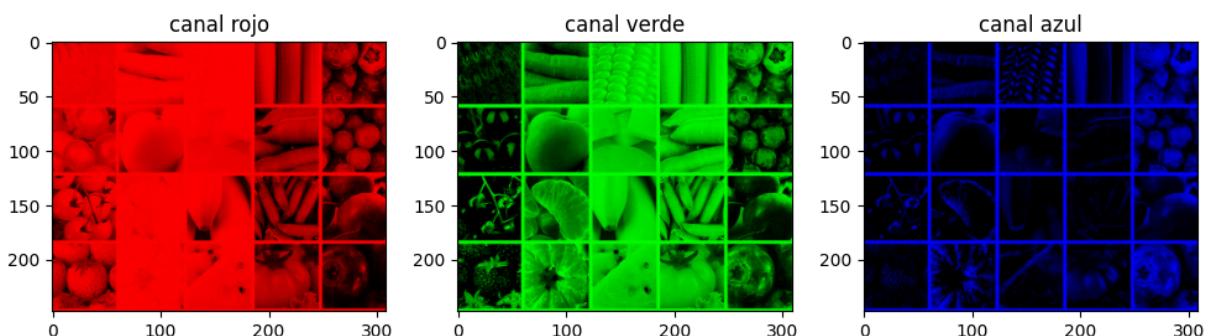
#imagen original
axs[0].imshow(rojo)
axs[0].set_title('canal rojo')

axs[1].imshow(verde)
axs[1].set_title('canal verde')

axs[2].imshow(azul)
axs[2].set_title('canal azul')

plt.tight_layout()
plt.show()

```



Ahora que mostramos los canales con sus ejes, podemos buscar los valores hallados y comprobar si estos valores corresponden a píxeles mínimos o máximos.

MEDIA EN PIXELES

Hallar la media de los píxeles en una imagen nos permite saber que tan buena es la iluminación en nuestra imagen y tener una noción de en qué valore rondan los píxeles de nuestra imagen.

```

In [ ]: #hallamos la media de los píxeles en cada capa
mediar = np.mean(r)
mediag = np.mean(g)
mediab = np.mean(b)

#hallamos la media de los píxeles en la imagen RGB
media = np.mean(imagen_redimensionada)

#imorimimos los valores obtenidos
print(f'La media de los píxeles en la capa roja es: {mediar}')

```

```
print(f'La media de los pixeles en la capa verde es: {mediag}')
```

```
print(f'La media de los pixeles en la capa azul es: {mediab}')
```

```
print(f'La media de los pixeles en la imagen RGB: {media}')
```

```
La media de los pixeles en la capa roja es: 188.3379006159307
```

```
La media de los pixeles en la capa verde es: 134.59475153982672
```

```
La media de los pixeles en la capa azul es: 62.295725023488885
```

```
La media de los pixeles en la imagen RGB: 128.40945905974877
```

DESVIACION ESTANDAR EN PIXELES

La desviación estándar nos permite saber que tanta es la variación entre los pixeles de nuestra imagen, esto nos permite conocer si nuestra imagen tiene una definición alta, mientras mayor sea la variación tendremos bordes mas definidos, lo cual nos permitirá identificar objetos en la misma.

```
In [ ]: stdr = np.std(r)
```

```
stdg = np.std(g)
```

```
stdb = np.std(b)
```

```
std = np.std(imagen_redimensionada)
```



```
print(f'La desviacion estandar en la capa roja es: {stdr}')
```

```
print(f'La desviacion estandar en la capa verde es: {stdg}')
```

```
print(f'La desviacion estandar en la capa azul es: {stdb}')
```

```
print(f'La desviacion estandar en la imagen RGB es: {std}')
```

```
La desviacion estandar en la capa roja es: 74.11144545370038
```

```
La desviacion estandar en la capa verde es: 83.7586355565115
```

```
La desviacion estandar en la capa azul es: 76.83191512276828
```

```
La desviacion estandar en la imagen RGB es: 93.8293847675871
```

ACCESO Y MANIPULACION DE PIXELES

podemos obtener los valores de cada pixel en cada canal de la siguiente manera, en donde el primer valor sera el canal rojo el segundo el canal verde y el tecrero el canal azul

VALORES RGB EN LOS PIXELES

Podemos hallar los valores RGB de un píxel de manera sencilla, solo debemos saber las coordenadas del pixel al que queremos acceder.

```
In [ ]: pixel_value = imagen[0,0]
```

```
print(F'El valor del pixel [0,0] es: {pixel_value}')
```

```
El valor del pixel [0,0] es: [238 255 254]
```

En este caso accedimos a un solo pixel, sin embargo podemos acceder tambiean a rango de pixeles.

```
In [ ]: #definimos un rango en este caso sera toda la columna 0
```

```
range_value = imagen[:,0]
```

```
print(F'Los valores de los pixeles en la columna 0 son: {range_value}')
```

```
Los valores de los pixeles en la columna 0 son: [[238 255 254]
[244 255 255]
[250 255 254]
...
[252 243 255]
[255 241 255]
[255 255 243]]
```

MANIPULACION DE PIXELES EN UN RANGO (ROI)

Ya vimos como acceder a los valores de los pixeles, sin embargo, podemos modificar sus valores de la siguiente manera.

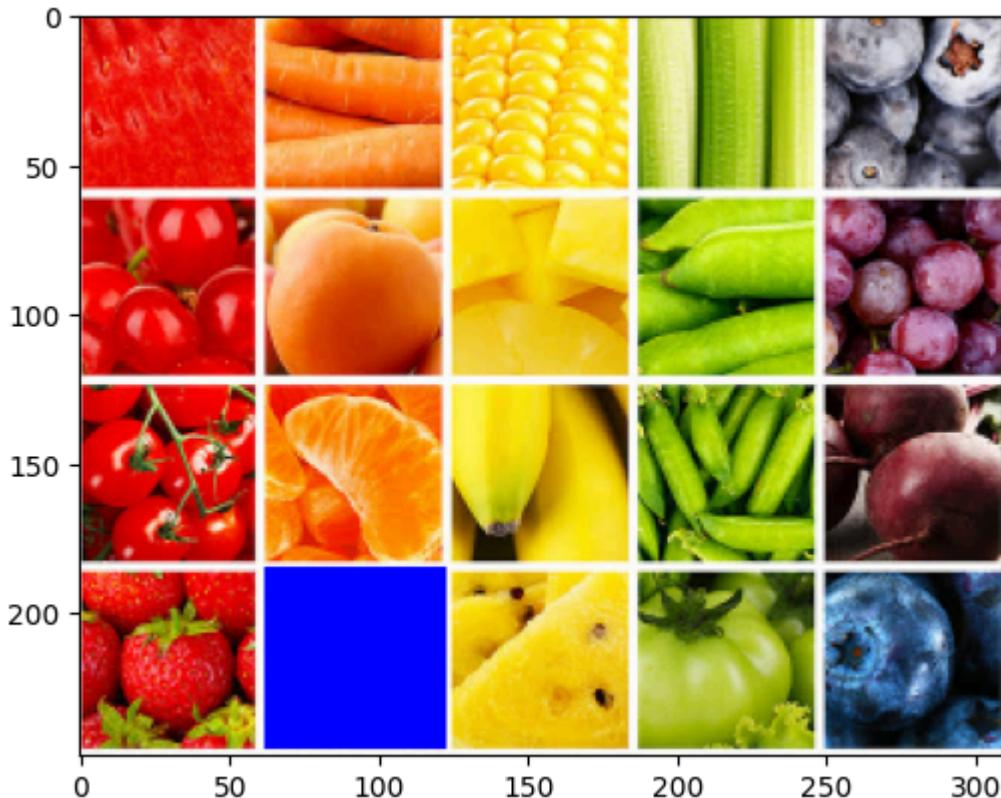
```
In [ ]: #creamo una copia de la imagen en la que efectuaremos cambios
img_manipulada = imagen_redimensionada.copy()

#definimos un rango a manipular
rangol = img_manipulada[185:246,62:123]

#modificamos el valor de los pixeles en el rango, en este caso les asignaremos sol
rangol[:] = [0,0,255]

#mostramos la imagen
plt.imshow(img_manipulada)
```

Out[]: <matplotlib.image.AxesImage at 0x1e961e0cf80>



Podemos observar como modificamos un cuadro de la imagen, el cual le asignamos un color azul completamente.

MANIPULACION DE TODOS LOS PIXELES

Hemos modificado pixeles de manera individual o en rangos, sin embargo, podemos aplicar modificaciones a todos los pixeles, a continuación se muestran algunas operaciones donde se modifican los valores de todos los pixeles.

INVERSION DE COLORES

Podemos invertir los valores de nuestra imagen de manera sencilla, como se muestra a continuación.

```
In [ ]: #INVERTIMOS LOS COLORES DE NUESTRA IMAGEN
         imagen_invertida = cv2.bitwise_not(imagen_redimensionada)

#mostramos la imagen
plt.imshow(imagen_invertida)
```

Out[]: <matplotlib.image.AxesImage at 0x1e961bf4c20>

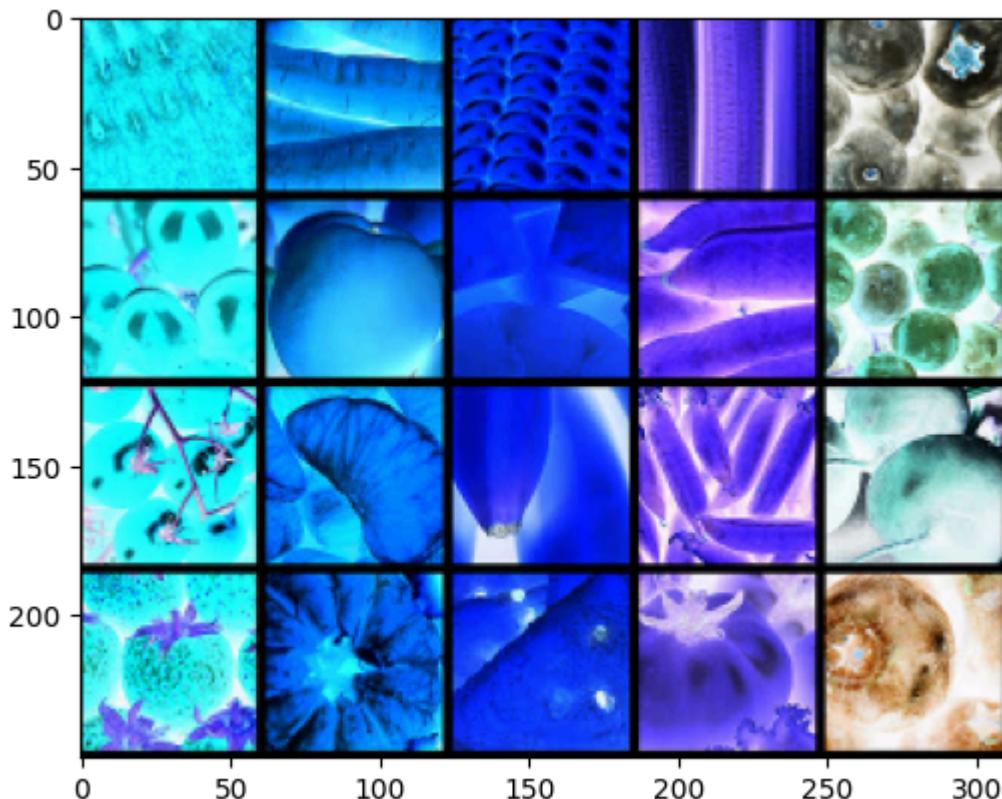


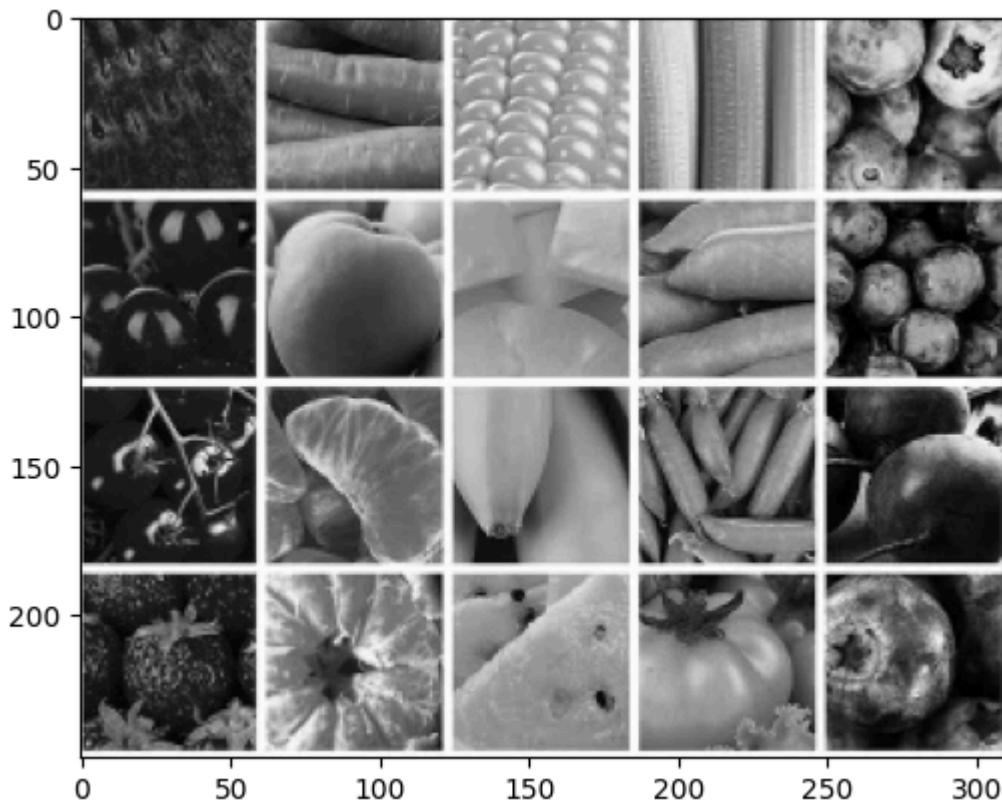
IMAGEN EN ESCALA DE GRISES

Podemos cambiar los valores de nuestros pixeles haciendo que dejen de tener 3 valores(RGB) haciendo que tengan un solo valor de intensidad, dejando como resultado una imagen en escala de grises.

```
In [ ]: #CONVERTIMOS NUESTRA IMAGEN EN ESCALA DE GRISES
         imagen_gray = cv2.cvtColor(imagen_redimensionada, cv2.COLOR_BGR2GRAY)
```

```
#MOSTRAMOS NUESTRA IMAGEN  
plt.imshow(imagen_gray,cmap='gray')
```

Out[]: <matplotlib.image.AxesImage at 0x1e961e38650>



En este caso hemos eliminado los canales de nuestra imagen por lo que ahora solo consta de uno, que asigna una intensidad a nuestra imagen con valores entre 0 a 256.

REFERENCIAS