

AI lab

ferdinandi.1958589

May 8, 2023

Contents

1	Introduction	3
2	Numpy.	3
2.1	np.random.seed(0)	4
2.2	Caratteristiche array numpy.	5
2.3	Indici array numpy.	5
2.4	Generazione array numpy.	7
2.5	Reshape.	7
2.6	Copy vs view.	8
2.7	Concatenate.	10
2.8	Vstack - Hstack - Dstack.	11
2.8.1	vstack.	11
2.8.2	hstack.	11
2.8.3	dstack.	12
2.9	Split.	12
2.9.1	vsplit.	13
2.9.2	hsplit.	14
2.10	Perchè utilizzare Numpy ?	15
2.11	Wrappers Numpy.	15
2.12	np.empty(x)	16
2.13	np.ones(x)	17
3	OpenCV.	18
3.1	Caricamento e visualizzazione di un immagine.	18
3.2	Modificare il formato dell'immagine o video.	19
3.3	Analisi immagine.	19
3.4	Canali dell'immagine.	20
3.5	Operazioni immagine.	21
3.5.1	Modifica pixel immagine.	21
3.5.2	Modifica porzione immagine.	22
3.5.3	Disegno nell'immagine.	23
3.5.4	Disegno LINEA nell'immagine.	24
3.5.5	Disegno RETTANGOLO nell'immagine.	25
3.5.6	Disegno CERCHIO nell'immagine.	27
3.6	Image Processing.	29
3.7	Conversione BGR - RGB.	30
3.8	Flag caricamento immagini.	30
3.9	Transparency: nuovo canale in RGB: RGBA	30
3.10	Trasformazione geometrica delle immagini.	32

3.10.1	Matrice di trasformazione.	32
3.10.2	Scaling.	34
3.10.3	Translation - traslazione immagine.	35
3.10.4	Rotation - rotazione.	36
3.10.5	Affine transformation.	38
3.10.6	Perspective transformation.	39
3.10.7	Perspective vs Affine.	39
3.10.8	Perspective transformation example.	40
3.11	Image Filtering.	43
3.11.1	Effetto BLUR.	45
3.11.2	Nitidezza immagine.	47
3.11.3	Effetto countour detection - estrazione bordi.	49
3.11.4	Blend Images.	51
3.12	Operazioni aritmetiche immagini.	51
3.13	BITWISE Operation.	53
3.14	Effetto Sketch	55
3.14.1	Effetto sketch real time con videocamera.	57
3.15	Histogram.	59
3.16	Feature extraction - Estrazione delle caratteristiche	67
3.16.1	Scale- Invariant feature Transform (SIFT)	71
3.16.2	Speeded-Up Robust Features - SURF	72
3.16.3	AKAZE (Accelerated-KAZE)	73
3.17	Binary feature extractors.	73
3.17.1	ORB - Oriented FAST and Rotated BRIEF	74
4	NLP.	75
4.1	Tokenizing.	77
4.1.1	stepwords	78
4.2	Stemming	79
4.3	POS: part of speech (discorso) tagging.	80
4.4	Lemmatizing.	81
4.5	Chunking.	82

1 Introduction

In questo corso verranno utilizzate le librerie:

- numpy;
- openCV;
- nltk;

2 Numpy.

Numpy è una libreria Python che fornisce strumenti per lavorare con array multidimensionali e matrici. La libreria è stata progettata per consentire agli utenti di eseguire operazioni matematiche e scientifiche complesse in modo semplice e veloce.

Numpy è una libreria fondamentale per l'elaborazione dei dati in Python, spesso utilizzata in congiunzione con altre librerie come pandas, matplotlib, scikit-learn e molti altri. Una delle caratteristiche distintive di Numpy è la sua capacità di gestire facilmente matrici di grandi dimensioni e di effettuare operazioni matematiche su di esse in modo efficiente.

Tra le funzionalità offerte dalla libreria ci sono:

- supporto per array multidimensionali, che possono avere qualsiasi numero di dimensioni;
- operazioni matematiche su array, tra cui addizione, sottrazione, moltiplicazione e divisione;
- funzioni matematiche avanzate, come trigonometria, logaritmi e esponenziali;
- generazione di numeri casuali;
- capacità di creare maschere booleane per selezionare parti di un array;
- capacità di ordinare e manipolare gli array in base a diversi criteri;
- integrabilità con altre librerie di analisi dati e machine learning.

In sintesi, Numpy è una libreria essenziale per qualsiasi attività che richieda la manipolazione di dati in forma di array multidimensionali e matrici, come l'elaborazione di immagini, l'analisi dati, la visualizzazione di dati e l'apprendimento automatico.

Numpy è spesso utilizzato in combinazione con OpenCV per il calcolo di operazioni matematiche e per l'elaborazione di immagini. In particolare, OpenCV utilizza gli array multidimensionali di NumPy come formato di immagine standard per le immagini acquisite e elaborate.

Ad esempio, quando si lavora con immagini in OpenCV, gli array multidimensionali di NumPy sono spesso utilizzati per rappresentare le immagini stesse e le loro trasformazioni. Gli array NumPy possono essere utilizzati anche per il calcolo di operazioni matematiche come la somma, la sottrazione, la moltiplicazione e la divisione tra immagini, la convoluzione e la trasformata di Fourier.

Vediamo alcuni esempi:

Per vedere alcuni esempi bisogna innanzitutto importare la libreria:

```
import numpy as np
```

2.1 np.random.seed(0)

Supponiamo di invocare:

```
np.random.seed(0)
```

Questa invocazione rende i numeri casuali prevedibili.

A questo punto scriviamo queste tre righe di codice:

```
x1 = np.random.randint(10, size=6) # one dimensional array
x2 = np.random.randint(10, size=(3,4)) # two dimensional array
x3 = np.random.randint(10, size=(3,4,5)) # three dimensional array
```

Ogni volta che facciamo partire il programma, verranno generati casualmente dei numeri.

Dato che abbiamo utilizzato:

```
np.random.seed(0)
```

i numeri saranno sempre uguali ad ogni esecuzione e quindi apparirà ogni volta lo stesso insieme di numeri.
Per:

```
print("x1:\n", x1, "\nx2:\n", x2, "\nx3:\n")
```

un possibile output potrebbe essere:

```
x1:
[5 0 3 3 7 9]

x2:
[[3 5 2 4]
 [7 6 8 8]
 [1 6 7 7]]

x3:
[[[8 1 5 9 8]
 [9 4 3 0 3]
 [5 0 2 3 8]
 [1 3 3 3 7]]

 [[0 1 9 9 0]
 [4 7 3 2 7]
 [2 0 0 4 5]
 [5 6 8 4 1]]

 [[4 9 8 1 1]
 [7 9 9 3 6]
 [7 2 0 3 5]
 [9 4 4 6 4]]]
```

e ad ogni esecuzione, verranno generate sempre le stesse matrici, grazie a .seed(0)

Se invece, np.random.seed(0) non viene impostato, verranno generate matrici diverse ad ogni invocazione.

2.2 Caratteristiche array numpy.

Consideriamo la matrice:

```
x2:  
[[3 7 5 5] [0 1 5 9] [3 0 5 0]]
```

e consideriamo il seguente frammento di codice:

```
print(f"x2 ndim: {x2.ndim}")  
print(f"x2 shape: {x2.shape}")  
print(f"x2 size: {x2.size}")  
print(f"x2 dtype: {x2.dtype}")  
print(f"x2 itemsize: {x2.itemsize} bytes")  
print(f"x2 bytes: {x2.nbytes} bytes")
```

L'output è:

```
x2 ndim: 2 # bidimensional array  
x2 shape: (3, 4) # how many elements in each dimension, for the first dimension 3 elements  
[[[]],  
# for the second dimension 4 elements [x,x,x,x]  
x2 size: 12 -> the total number  
x2 dtype: int64  
x2 itemsize: 8 bytes  
x2 bytes: 96 bytes (8 * 12 bytes)
```

Cosa indicano queste variabili:

- *ndim*: indica il numero di dimensioni dell'array (bidimensionale, tridimensionale, ...)
- *shape*: la forma dell'array, cioè il numero di elementi in ogni dimensione.
- *size*: il numero totale di elementi nell'array.
- *dtype*: il tipo di dati degli elementi nell'array.
- *itemsize*: a dimensione in byte di ogni elemento nell'array.
- *byte*: il numero totale di byte utilizzati dall'array

2.3 Indici array numpy.

Negli array numpy possono essere selezionati alcuni elementi grazie agli indici: Consideriamo:

```
x1: [4 3 4 4 8 4]
```

```
x2:  
[[3 7 5 5]  
[0 1 5 9]  
[3 0 5 0]]
```

```
x3:  
[[[1 2 4 2 0]  
[3 2 0 7 5]]]
```

```

[9 0 2 7 2]
[9 2 3 3 2]]
```

```

[[3 4 1 2 9]
 [1 4 6 8 2]
 [3 0 0 6 0]
 [6 3 3 8 8]]
```

```

[[8 2 3 2 0]
 [8 8 3 8 2]
 [8 4 3 0 4]
 [3 6 9 8 0]]]
```

```

print(x1[-3])
- 4
```

```

print(x2[1,0])
- 0 (seconda riga, prima colonna)
```

```

print(x3[1,0,0])
- 3 di questa riga [3 4 1 2 9]
```

```

print(x1[3:])
- [4 8 4]
```

```

print(x1[:3]) | print(x1[0:3])
- [4 3 4]
```

```

print(x1[::-1]) # al contrario
- [4 8 4 4 3 4]
```

```

print(x1[0:3:2])
- [4 4] (il :2 indica che salta di 2)
```

```

print(x1[0:3:-1])
- [] (il :-1 prende dall ultimo)
```

```

print(x2[:, 3])
- [[3 7 5]
 [0 1 5]]
```

```

print(x2[:, 2])
- [[3 5]
 [0 5]]
```

```

# Grazie a questo comando in opencv e' possibile ruotare un immagine
print(x2[::-1, ::-1])
- [[0 5 0 3]
 [9 5 1 0]
 [5 5 7 8]]
```

```

# Vogliamo prendere tutte le righe della colonna 3:
print(x2[:, 2])
- [5 5 5]
```

```

# Vogliamo tutte le colonne della prima riga:
print(x2[0, :])
- [3 7 5 5]
```

2.4 Generazione array numpy.

È possibile generare un array di interi da 0 a 9 in ordine crescente:

```
x4 = np.arange(10)
print(f"x4: {x4}") # x4: [0 1 2 3 4 5 6 7 8 9]
```

Possiamo modificare il valore di un elemento nell'array:

```
x2[0,0] = 99
print(x2)
"""
[[99 7 5 5]
 [ 0 1 5 9]
 [ 3 0 5 0]]
"""


```

2.5 Reshape.

Vediamo come ridimensioniamo un array:

```
np.random.seed(0)
x = np.arange(1,10)
print(x) # [1 2 3 4 5 6 7 8 9]

two_dim = x.reshape((3,3))
print(two_dim)
"""
[[1 2 3]
 [4 5 6]
 [7 8 9]]
"""


```

Questo codice Python utilizza il modulo NumPy per creare un array unidimensionale di numeri interi da 1 a 9 utilizzando la funzione arange().

In seguito, la funzione reshape() viene utilizzata per trasformare l'array unidimensionale in una matrice bidimensionale di 3 righe e 3 colonne.

Infine stampa entrambi.

Il modulo NumPy nella funzione reshape() accetta come parametro una tupla che indica le dimensioni dell'array bidimensionale risultante. In questo caso, la tupla (3,3) crea una matrice a 3 righe e 3 colonne.

Se noi avessimo provato a scrivere:

```
two_dim = x.reshape((2,-1))
```

Avremmo chiesto a NumPy di creare una matrice con due righe e con un numero di colonne determinato automaticamente.

Questo funziona solo se il numero di elementi nell'array iniziale è la moltiplicazione esatta delle dimensioni specificate nella tupla di reshape.

In questo caso, x ha nove elementi, e 2x4 farebbero 8 elementi totali, quindi NumPy non sarebbe stato in grado di creare una matrice bidimensionale con due righe in modo coerente con l'array iniziale, perciò

avremmo avuto in output un errore (perchè in questo caso l'array ha una lunghezza dispari e non riesce a gestire l'array bidimensionale con dimensione dispari).

Se invece scriviamo:

```
x = np.arange(1,11)
two_dim = x.reshape((2,-1))
print(two_dim)
"""
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
"""
```

Non abbiamo errori.

Altro esempio:

```
x = np.arange(16).reshape((4,4))
print (x)
"""
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
"""
```

2.6 Copy vs view.

Una view di un array in NumPy è un'istanza diversa dell'array originale, ma che condivide i dati dello spazio di memoria dello stesso. Ciò significa che se si modifica un elemento della view, l'array originale risentirà della modifica. Viceversa, se si modifica un elemento dell'array originale, la view sarà influenzata dalla modifica.

Al contrario, una copia di un array in NumPy è un'istanza completamente separata dell'array originale, che ha una propria copia dei dati. Ciò significa che una copia dell'array in NumPy è completamente indipendente dall'originale e le modifiche a una copia non influenzano l'originale.

Per gestire e controllare la vista o la copia di un array, è possibile usare il metodo ‘copy()’ per creare una copia esplicita di un array, come nell'esempio.

In numpy questa è una view, se modifichiamo un elemento della view anche l'array originale risentirà della modifica:

```
left, right = np.hsplit(x, [2])
left[0,0] = 333
print(x) # l'array originale risentira' della modifica della view.
print(left) # view

"""
array originale:
[[333  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

```
view:  
[[333  1]  
 [ 4  5]  
 [ 8  9]  
 [12 13]]  
***
```

Vediamo la copy():

```
left, right = np.hsplit(x10, [2])  
left = left.copy()  
left[0,0] = 333  
  
print(x) # l'array originale non risente della modifica fatta sulla copy()  
print(left) # view  
  
***  
x:  
[[ 0  1  2  3]  
 [ 4  5  6  7]  
 [ 8  9 10 11]  
 [12 13 14 15]]  
  
left:  
[[333  1]  
 [ 4  5]  
 [ 8  9]  
 [12 13]]  
***
```

Definiamo meglio la differenza tra vista e copia:

Creo una matrice 3x3 e la assegno alla variabile A

```
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
```

Poi estraggo dalla matrice una sottomatrice 2x2 composta dalle prime due righe e colonne e la assegno alla variabile B

```
B = A[0:2,0:2]
```

Il contenuto della matrice B è il seguente B: ([[1, 2],[4, 5]]).

La variabile B è una vista (view) dell'array A.

La variabile B non contiene i dati ma soltanto i riferimenti allo stesso spazio di memoria della variabile A.

Pertanto, se modifico un dato nella variabile A, la stessa modifica si vede anche nella variabile B. Ad esempio, modifico l'elemento (1,1) nell'array A

```
A[0,0] = -1
```

Ora il contenuto della variabile A è il seguente

```
A: ([[ -1,  2,  3], [ 4,  5,  6], [ 7,  8,  9]])
```

La stessa modifica si ripercuote anche sulla vista nella variabile B perché ha un riferimento che punta allo stesso spazio di memoria.

```
B : ([[ -1, 2], [4, 5]])
```

Per evitare questa interdipendenza devo copiare i dati al momento dell'estrazione del subarray.
Una copia è una copia fisica dei dati in uno spazio di memoria diverso rispetto alla variabile copiata.
In questo modo il contenuto della nuova variabile è del tutto indipendente dalla variabile copiata.
Ad esempio, estraggo le prime due righe e colonne della matrice indicando l'attributo copy().

```
C = A[0:2,0:2].copy() # C: ([[ -1, 2], [4, 5]])
```

In questo caso la variabile C contiene una copia dei dati della variabile C memorizzati in un altro spazio di memoria della RAM.

Pertanto, se modifico un elemento della variabile A, la modifica non si ripercuote anche sulla variabile C
Ad esempio, modifico l'elemento (0,0) della variabile array A

```
A[0,0] = -2 # A: ([[ -2, 2, 3], [4, 5, 6], [7, 8, 9]])
```

La modifica alla variabile A non modifica il contenuto della variabile C che è una copia della variabile A.

```
C: ([[ -1, 2], [4, 5]])
```

2.7 Concatenate.

La funzione ‘concatenate’ di NumPy unisce due o più array insieme lungo un asse specificato.

La sintassi generale della funzione è la seguente:

```
numpy.concatenate((array1, array2, ...), axis=0, out=None)
```

dove:

- ‘array1’, ‘array2’, ... sono gli array da concatenare.
- ‘axis’ è l’asse lungo cui eseguire la concatenazione; di default è ‘axis = 0’ (concatenazione lungo le righe); altrimenti si può specificare ‘axis = 1’ (concatenazione lungo le colonne)
- ‘out’ è l’array di output opzionale dove risultato della concatenazione viene scritto.

Ad esempio:

```
x = np.array([1,2,3])
y = np.array([4,5,6])
z = np.concatenate([x,y])
print(z) # [1 2 3 4 5 6]
```

```
x2 = np.array([1,2,3])
y2 = np.array([4,5,6])
t2 = np.array([99,99,99])
z2 = np.concatenate([x2,y2,t2])
print(z2) # [ 1 2 3 4 5 6 99 99 99 ]
```

```
x3 = np.array([[1,2,3], [4,5,6]])
```

```

y3 = np.array([[99,99,99],[99,99,99]])
z3 = np.concatenate([x3,y3])
print(z3)
"""
[[ 1  2  3]
 [ 4  5  6]
 [99 99 99]
 [99 99 99] ]
"""

a = np.array([[1,2], [3,4], [5,6]])
b = np.array([[7,8], [9,10], [11,12]])
c = np.concatenate((a, b), axis=0)
print(c)
"""
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12] ]
"""

```

2.8 Vstack - Hstack - Dstack.

NumPy hstack e NumPy vstack sono simili in quanto entrambi combinano insieme gli array NumPy.

La differenza principale è che np.hstack combina gli array NumPy orizzontalmente, np.vstack combina gli array verticalmente e dstack opera su un asse di profondità aggiuntivo.

2.8.1 vstack.

vstack è uguale a concatenate: unisce elementi di due o più array in un singolo array verticalmente (a riga).

```

x4 = np.array([[1,2,3], [4,5,6]])
y4 = np.array([[99,99,99],[99,99,99]])
z4 = np.vstack([x4,y4])
print(z4)
"""
[[ 1  2  3]
 [ 4  5  6]
 [99 99 99]
 [99 99 99] ]
"""

```

2.8.2 hstack.

hstack unisce elementi di due o più array in un singolo array orizzontalmente (a colonna).

```

x5 = np.array([[1,2,3], [4,5,6]])
y5 = np.array([[99,99,99],[99,99,99]])
z5 = np.hstack([x5,y5])

```

```
print(z5)
"""
[[ 1  2  3 99 99 99]
 [ 4  5  6 99 99 99]
 ...
```

2.8.3 dstack.

dstack viene utilizzato per disporre a pila gli array in termini di profondità di sequenza (lungo il terzo asse). Può creare un array tridimensionale. Ad esempio, se abbiamo due array [[1], [2], [3]] e [[4], [5], [6]] la concatenazione profonda (dstack) produrrebbe l'array [[[1,4]], [[2,5]], [[3,6]]].

Posso ricostruire l'immagine principali con i canali dell'immagine.

```
x6 = np.array([[1,2,3], [4,5,6]])
y6 = np.array([[99,99,99],[99,99,99]])
z6 = np.hstack([x6,y6])
print(z6)
"""
[[ 1  2  3 99 99 99]
 [ 4  5  6 99 99 99]
 ...
```

2.9 Split.

La funzione ‘split’ di NumPy divide un array in più sottoinsiemi lungo un asse specificato.

La sintassi generale della funzione ‘split’ è la seguente:

```
numpy.split(array, num_or_sections, axis=0)
```

dove:

- ‘array’ : l’array da suddividere;
- ‘num or sections’ : specifica il numero di sotto-arrays da creare o la posizione degli indici;
- ‘axis’ : l’asse lungo cui suddividere l’array.

L’argomento ‘num or sections’ può essere un intero o una sequenza di interi che specificano il numero di pezzi in cui bisogna dividere l’array.

Se si specifica un singolo intero ‘k’, l’array viene diviso in ‘k’ parti uguali lungo l’asse specificato.

Se ‘num or sections’ è una sequenza di interi, questi rappresentano i punti di intersezione dell’array lungo l’asse specificato (ad esempio, ‘[2, 4]’ suddividerà l’array in tre parti: dal primo all’indice 2, dall’indice 2 all’indice 4 e dall’indice 4 alla fine).

Ad esempio, supponiamo di avere l’array ‘a’ di dimensione ‘(6,)’, contenente i numeri da ‘0’ a ‘5’, e di volerlo dividere in ‘3’ parti uguali lungo l’asse 0. Possiamo farlo con il seguente codice, Il risultato sarà una lista di tre array, ciascuno contenente due elementi:

```
a = np.array([0, 1, 2, 3, 4, 5])
b = np.split(a, 3) # divisione di 'a' in 3 parti
# [0,1] [2,3] [4,5]
```

La funzione ‘split’ supporta anche la suddivisione lungo l’asse delle colonne (‘axis=1’). In questo caso, l’array viene diviso lungo l’asse delle colonne e ogni sottoarray avrà il numero di colonne specificato come argomento in ‘num or sections’.

Altri esempi:

```
x = np.array([1,2,3,4,5,6])
x7, x8, x9 = np.split(x, [1,4])
print(x7, x8, x9) # [1] [2 3 4] [5 6]

x7, x8, x9 = np.split(x, [2,5])
print(x7, x8, x9) # [1 2] [3 4 5] [6]

x7, x8, x9 = np.split(x, [2,4])
print(x7, x8, x9) # [1 2] [3 4] [5 6]

x = np.array([1,2,3,4,5,6])
x7, x8, x9 = np.split(x, [1,4])
print(x7, x8, x9) # [1] [2 3 4] [5 6]

x7, x8, x9 = np.split(x, [2,5])
print(x7, x8, x9) # [1 2] [3 4 5] [6]

x7, x8, x9 = np.split(x, [2,4])
print(x7, x8, x9) # [1 2] [3 4] [5 6]
```

Le funzioni ‘vsplit’ e ‘hsplit’ di NumPy sono delle varianti della funzione ‘split’ e permettono di suddividere un array lungo gli assi verticali e orizzontali rispettivamente.

2.9.1 vsplit.

La funzione ‘vsplit’ divide l’array in più parti verticali, ciascuna composta da un determinato numero di righe. La sintassi della funzione ‘vsplit’ è la seguente:

```
numpy.vsplit(array, num_sections)
```

dove :

- ‘array’ è l’array che si vuole dividere;
- ‘num sections’ indica il numero di parti in cui l’array deve essere diviso lungo l’asse verticale.

Ad esempio, supponiamo di avere l’array ‘a’ di dimensione ‘(4, 4)’ e di volerlo dividere in due parti lungo l’asse verticale. Possiamo farlo con la seguente istruzione e il risultato sarà una lista contenente due array, ciascuno formato da due righe e quattro colonne:

```
a = np.arange(16).reshape((4, 4))
#####
([[0, 1, 2, 3],
 [4, 5, 6, 7]])
#####

b = np.vsplit(a, 2)
#####
([[ 8,  9, 10, 11],
 [12, 13, 14, 15]])
#####
```

2.9.2 hsplit.

La funzione ‘hsplit‘, invece, divide l’array in più parti orizzontali, ciascuna composta da un determinato numero di colonne. La sintassi della funzione ‘hsplit‘ è la seguente:

```
numpy.hsplit(array, num_sections)
```

dove :

- ‘array‘ è l’array che si vuole dividere ;
- ‘num sections‘ indica il numero di parti in cui l’array deve essere diviso lungo l’asse orizzontale.

Ad esempio, supponiamo di avere l’array ‘a‘ di dimensione ‘(4, 4)‘ e di volerlo dividere in due parti lungo l’asse orizzontale. Possiamo farlo con la seguente istruzione, il risultato sarà una lista contenente due array, ciascuno formato da quattro righe e due colonne:

```
a = np.arange(16).reshape((4, 4)) #
"""
[[ 0,  1],
 [ 4,  5],
 [ 8,  9],
 [12, 13]]"""

b = np.hsplit(a, 2)
"""
[[ 2,  3],
 [ 6,  7],
 [10, 11],
 [14, 15]]"""


```

Ad esempio consideriamo:

```
x10 = np.arange(16).reshape((4,4))
print (x10)
"""
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]"""


```

Con:

```
upper , lower = np.vsplit(x10, [2])
print(upper) # [[0 1 2 3], [4 5 6 7]]
print(lower) # [[ 8 9 10 11] [12 13 14 15]]

left , right = np.hsplit(x10, [2])
print(left) # [[ 0 1], [ 4 5], [ 8 9], [12 13]]
print(right) # [[ 2 3], [ 6 7], [10 11], [14 15]]

left , center, right = np.hsplit(x10, [2,3])
print(left) # [[ 0 1], [ 4 5], [ 8 9], [12 13]]
print(center) # [[ 2], [ 6], [10], [14]]
print(right) # [[ 3], [ 7], [11], [15]]
```

2.10 Perchè utilizzare Numpy ?

```
np.random.seed(0)

def reciprocal(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1.0 / values[i]

values = np.random.randint(1, 10, size=10000000)

start = time.time()

# reciprocal(values)
print(time.time() - start)

# 6.378808259963989

start = time.time()
result = 1.0 / values # this is the numpy version.
print(time.time() - start)

# 0.011694192886352539
```

In questo codice vengono effettuate due operazioni:

1. Viene definita una funzione Python "reciprocal()" che prende un elenco di valori in input, crea un array vuoto di output e quindi itera su ciascun valore nell'elenco di input e ne calcola il reciproco ($1/x$), quindi lo aggiunge all'array di output.
2. Viene generato un array NumPy "values" di 10 milioni di numeri interi casuali tra 1 e 10.

Successivamente, viene misurato il tempo necessario per eseguire la funzione "reciprocal()" e per calcolare il reciproco degli elementi di "values" con l'operazione di NumPy.

Il primo blocco di codice utilizza "reciprocal()" per calcolare i reciproci degli elementi della matrice "values". Il tempo necessario per eseguire questa operazione viene misurato utilizzando la funzione `time.time()` e viene stampato a schermo.

Il secondo blocco di codice utilizza invece l'operazione di NumPy, cioè " $1.0/values$ ", per calcolare i reciproci degli elementi della matrice "values". Il tempo necessario per eseguire questa operazione viene nuovamente misurato utilizzando `time.time()` e viene stampato a schermo.

Come si può notare dai tempi di esecuzione riportati, l'utilizzo dell'operazione di NumPy per l'elaborazione del reciproco risulta molto più veloce rispetto all'utilizzo della funzione "reciprocal()" scritta in Python.

2.11 Wrappers Numpy.

In NumPy, i "wrappers" sono funzioni che aiutano a integrare codice scritto in linguaggi diversi, come C o Fortran, all'interno di codice Python utilizzando la sintassi di NumPy.

In particolare, questi wrappers forniscono un'interfaccia di alto livello che semplifica l'uso di librerie e codice esterno all'interno di programmi Python che utilizzano NumPy, consentendo in pratica di utilizzare il codice esterno come se fosse una libreria di funzioni di NumPy.

Ci sono diversi tipi di wrappers in NumPy, come ad esempio:

- I wrapper per librerie BLAS e LAPACK: questi wrapper forniscono un’implementazione ad alte prestazioni delle funzioni algebriche di base, ad esempio la moltiplicazione di matrici, l’inversione di matrici, la risoluzione di sistemi di equazioni lineari, e altre operazioni matematiche.
- I wrapper per librerie libsvm, liblinear e libquadprog: questi wrapper forniscono un’interfaccia per utilizzare efficientemente le librerie per la classificazione e la regressione utilizzando le macchine a vettori di supporto (SVM) e altri algoritmi di apprendimento automatico.
- I wrapper per librerie di grafica OpenGL: questi wrapper forniscono un’interfaccia per utilizzare OpenGL per la visualizzazione di dati e grafici all’interno di programmi Python che utilizzano NumPy.

In generale, l’uso di wrappers rende possibile l’accesso a librerie scritte in altri linguaggi all’interno di Python, senza la necessità di dover riscrivere tutto il codice in Python nativo.

Esempi:

```

x = np.arange(9).reshape((3,3))
result = x ** 2
print(result) # [[ 0  1  4] [ 9 16 25] [36 49 64]]
# ** è il wrappers di np.power(base, esponente)
result = np.power(x, 2)
print(f'result: {result}') # [[ 0  1  4] [ 9 16 25] [36 49 64]]


x2 = np.arange(9)
result = x2 + 2
print(result) # [ 2 3 4 5 6 7 8 9 10] x
# + è il wrappers di np.add(x,2)
x3 = np.arange(9)
result = np.add(x3, 2)
print(result) # [ 2 3 4 5 6 7 8 9 10]

```

Come possiamo vedere:

- ** è il wrappers di `np.power(base, esponente)`
- $+$ è il wrappers di `np.add`.

L’utilizzo della funzione ‘`np.add`’, ‘`np.power`’ offre il vantaggio di poter gestire array di grandi dimensioni in modo più efficiente rispetto all’operatore ‘ $+$ ’, ‘ ** ’, poiché può sfruttare le proprietà di broadcasting di NumPy.

2.12 `np.empty(x)`

La funzione ‘`np.empty(x)`’ crea un array numpy vuoto di dimensione `x`, ovvero un array di `x` elementi in cui ogni elemento ha un valore di default che dipende dalla memoria che era già allocata per l’array in questione. Il tipo di dato degli elementi dell’array dipende dalla configurazione di numpy usata dal programma.

Se si esegue la funzione ‘`print(np.empty(x))`’, verrà stampato a schermo l’array vuoto creato, dove ogni elemento apparirà come una casella vuota o con un valore di default che dipende dal sistema operativo e dal tipo di dato degli elementi.

Ad esempio:

```

print(np.empty(9)) # [9.9e-324 1.5e-323 2.0e-323 2.5e-323 3.0e-323 3.5e-323 4.0e-323 4.4e-323
4.9e-323]

```

Le funzioni di np, come np.add permettono di specificare altri parametri opzionali quali l'utilizzo di mask, valori di ritorno in caso di overflow e underflow, etc.

```
x = np.arange(9)
result = np.empty(9)
np.add(x, 2, out=result) # mette x con i membri sommati di due in result

print(result) # [ 2. 3. 4. 5. 6. 7. 8. 9. 10.]
```

In questo codice, viene creato un array ‘x‘ di dimensione 9, utilizzando la funzione ‘np.arange()‘, che restituisce una sequenza di numeri da 0 a 8. Successivamente, viene creato un altro array ‘result‘ vuoto, sempre di dimensione 9.

La funzione ‘np.add()‘ viene utilizzata per sommare 2 all’array ‘x‘, ed i risultati vengono immagazzinati nell’array ‘result‘. In pratica, ogni elemento dell’array ‘x‘ viene aumentato di 2 e il risultato viene copiato nell’array ‘result‘.

Il risultato stampato sarà un array con i suoi elementi formati dalla somma di 2 con gli elementi dell’array ‘x‘. Questo significa che l’output sarà ‘[2, 3, 4, 5, 6, 7, 8, 9, 10]‘.

2.13 np.ones(x)

La funzione ‘np.ones(x)‘ di NumPy crea un array di dimensione x con valori pari a 1 in ogni elemento. In altre parole, crea un array di forma (x,) con valori tutti pari a 1. Questa funzione è utile quando si desidera creare un array di una certa dimensione e inizializzarlo con un valore costante (in questo caso, 1).

Se si esegue la funzione ‘print(np.ones(9))‘, il risultato stampato sarà un array con 9 elementi, dove ogni elemento ha un valore di 1.

Ad esempio:

```
x = np.arange(9)
result = np.ones(9)
print(result) # [1. 1. 1. 1. 1. 1. 1. 1. 1.]

np.add(x, 2, out=result) # mette x con i membri sommati di due in result

print(result) # [ 2. 3. 4. 5. 6. 7. 8. 9. 10.]
```

In questo codice, viene creato un array ‘x‘ di dimensione 9, utilizzando la funzione ‘np.arange()‘, che restituisce una sequenza di numeri da 0 a 8.

Viene poi creato un altro array ‘result‘ di dimensione 9, utilizzando la funzione ‘np.ones()‘, che restituisce un array di forma (9,) con ogni elemento pari a 1.

Il risultato stampato sarà un array con 9 elementi, dove ogni elemento ha un valore di 1.

Successivamente, la funzione ‘np.add()‘ viene utilizzata per sommare 2 all’array ‘x‘, ed i risultati vengono immagazzinati nell’array ‘result‘. In pratica, ogni elemento dell’array ‘x‘ viene aumentato di 2 e il risultato viene copiato nell’array ‘result‘.

Il risultato stampato sarà un array con i suoi elementi formati dalla somma di 2 con gli elementi dell’array ‘x‘. Questo significa che l’output sarà ‘[2, 3, 4, 5, 6, 7, 8, 9, 10]‘.

3 OpenCV.

OpenCV (Open Source Computer Vision) è una libreria open source utilizzata per l'elaborazione delle immagini e la visione artificiale.

Questa libreria fornisce una vasta gamma di funzionalità, in grado di soddisfare le esigenze di una vasta gamma di applicazioni, tra cui riconoscimento facciale, tracciamento dei movimenti degli occhi, identificazione degli oggetti, ecc. È stata sviluppata originariamente da Intel e ora è mantenuta da Willow Garage.

OpenCV è disponibile in diversi linguaggi di programmazione, tra cui Python, C / C ++, Java, ecc. Python è un linguaggio di programmazione versatile e facile da imparare, che lo rende una scelta popolare per sviluppatori che lavorano con OpenCV.

Python OpenCV fornisce numerosi strumenti per la visione artificiale, tra cui il rilevamento e il tracciamento degli oggetti, il riconoscimento facciale, la correzione dell'immagine, la segmentazione dell'immagine e la ricostruzione 3D. Inoltre, offre anche un ambiente di sviluppo facile da usare e una vasta documentazione, che lo rendono un'ottima scelta per gli sviluppatori.

3.1 Caricamento e visualizzazione di un'immagine.

Per caricare un'immagine in OpenCV con Python, è necessario assicurarsi di avere l'immagine corretta nella stessa cartella in cui si lavora con Python. Una volta fatto ciò, ecco come caricare l'immagine in OpenCV:

- Importare OpenCV:

```
import cv2
```

- Caricare l'immagine utilizzando la funzione cv2.imread():

```
img = cv2.imread('nome_file_immagine.jpg')
```

Assicurarsi di sostituire "nome_file_immagine.jpg" con il nome effettivo del file dell'immagine che si desidera caricare.

OpenCV lavora anche con immagini di formato 'png'.

- Verificare che l'immagine sia stata caricata correttamente utilizzando la funzione cv2.imshow():

```
cv2.imshow('Immagine', img)
cv2.waitKey(0)
```

La prima riga visualizza l'immagine e la seconda linea attende che si prema un tasto per chiudere la finestra. Nella funzione waitKey possiamo inserire un intero che indica il numero di millisecondi per cui l'immagine è aperta.

Con lo '0' indichiamo che l'immagine rimane aperta senza scadenza di tempo.

Ricorda che l'immagine deve essere nella stessa cartella di lavoro di Python. Se l'immagine è in una cartella diversa, assicurarsi di specificare il percorso corretto.

Per mostrare un'immagine in OpenCV, è possibile creare una funzione che semplifica le chiamate a due funzioni. Questa funzione richiede l'immagine da visualizzare, il titolo della finestra di visualizzazione e il tempo di visualizzazione. La funzione è la seguente:

```
def see(img, title, time):
    cv2.imshow(title, img)
    cv2.waitKey(time) # se il parametro = 0, allora l'immagine rimane aperta senza nessun timer.
```

La prima riga utilizza ‘cv2.imshow()‘ per visualizzare l’immagine specificata nella finestra di visualizzazione con il titolo specificato. La seconda riga utilizza ‘cv2.waitKey()‘ per impostare il tempo di visualizzazione. Se il tempo è impostato a 0, l’immagine rimane aperta senza alcun timer.

A questo punto per il caricamento e la visualizzazione dell’immagine possiamo:

```
img = cv2.imread('nome_file_immagine.jpg')
see(img, 'Immagine del 10...', 10000)
```

3.2 Modificare il formato dell’immagine o video.

Può succedere di avere bisogno di modificare il formato di un’immagine o video in OpenCV.

Per fare ciò, è possibile utilizzare la funzionalità di salvataggio delle immagini ‘cv2.imwrite()‘. Questo metodo richiede il nome del file e l’immagine da salvare, come segue:

```
cv2.imwrite('nome_file_immagine.png', img)
```

Il file viene salvato nella cartella di lavoro in formato PNG con il nome specificato. Se esiste già un file con lo stesso nome, il file precedente verrà sovrascritto.

L’immagine ‘img‘ viene salvata con il nome specificato come un file PNG.

3.3 Analisi immagine.

Possiamo fare un’analisi sulle caratteristiche dell’immagine, ad esempio:

```
print(f'The width of the image is: {img.shape[1]}')
print(f'The height of the image is: {img.shape[0]}')
print(f'The number of channels of the image is: {img.shape[2]}')

...
The width of the image is: 4256
The height of the image is: 2832
The number of channel of the image is: 3
..."
```

Queste variabili rappresentano le proprietà della immagine.

- ‘img.shape[0]‘ ci restituisce l’altezza dell’immagine,
- ‘img.shape[1]‘ ci restituisce l’ampiezza dell’immagine
- ‘img.shape[2]‘ ci restituisce il numero di canali dell’immagine (quindi 3 in questo caso indica che l’immagine è a colori, con i canali Red, Green e Blue).

Una determinata immagine può essere assunta come matrice 2D con ogni elemento come un pixel. Ci sono vari tipi di formati di immagine come Gray, RGB o RGBA, ecc. e ogni formato è diverso per quanti colori (pixel) può supportare.

3.4 Canali dell'immagine.

In OpenCV, le immagini sono rappresentate come matrici di pixel, dove ogni pixel contiene un valore di intensità per ogni canale di colore. Il numero di canali di colore dipende dal tipo di immagine.

Come abbiamo visto, con il comando ‘img.shape[2]’ viene restituito il numero di canali dell’immagine.

A seconda del numero di canali restituiti ci sono vari tipi di immagine:

- Un’immagine grayscale ha un unico canale di colore e ogni pixel rappresenta la luminosità dell’immagine in scala di grigi, che può variare da 0 (nero) a 255 (bianco). In OpenCV, un’immagine grayscale è rappresentata da una matrice bidimensionale di valori interi o in virgola mobile.
- Un’immagine RGB ha tre canali di colore (rosso, verde e blu), dove ogni pixel viene rappresentato da tre valori di intensità, uno per ogni canale di colore.

In OpenCV, l’ordine predefinito dei canali di colore per le immagini a colori è BGR invece di RGB, il che significa che il primo canale rappresenta il blu, il secondo il verde e il terzo il rosso. Anche in questo caso, ogni canale di colore può variare da 0 a 255.

- Un’immagine RGBA ha quattro canali di colore, come le immagini RGB, ma l’ultimo canale è utilizzato per memorizzare il valore alfa, che rappresenta la trasparenza del pixel. In OpenCV, l’ordine predefinito dei canali di colore per le immagini RGBA è BGRA invece di RGBA, il che significa che il primo canale rappresenta il blu, il secondo il verde, il terzo il rosso e il quarto il valore alfa. Anche in questo caso, ogni canale di colore può variare da 0 a 255.

In sintesi, in OpenCV abbiamo:

- Immagine grayscale: 1 canale di colore con valori di intensità che variano da 0 a 255.
- Immagine RGB: 3 canali di colore (BGR) con valori di intensità che variano da 0 a 255.
- Immagine RGBA: 4 canali di colore (BGRA) con i primi tre canali che rappresentano i colori e l’ultimo canale che rappresenta il valore alfa (trasparenza), con valori di intensità che variano da 0 a 255.

Il range di valori 0-255 rappresenta un byte di informazione per ogni canale di colore, il che significa che ogni pixel viene rappresentato da un numero intero a 8 bit. Questo range è il più comune perché è possibile rappresentare una vasta gamma di colori con 256 sfumature per ogni canale di colore.

La manipolazione dei canali dell’immagine è un’operazione comune nell’elaborazione delle immagini con OpenCV. Ad esempio, si può dividere un’immagine a colori in tre canali separati (rosso, verde e blu) e manipolare i canali separatamente per ottenere un effetto specifico.

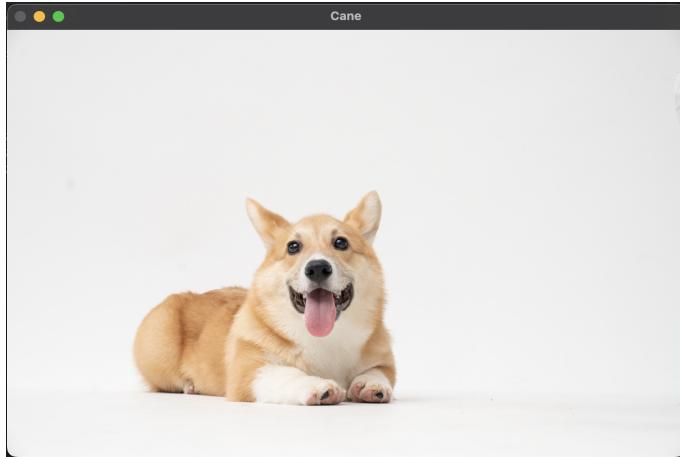
Oppure si può convertire un’immagine a colori in una immagine grayscale utilizzando un’opportuna funzione di conversione.

In ogni caso, la comprensione del valore dei pixel nei diversi canali è essenziale per manipolare correttamente le immagini in OpenCV.

3.5 Operazioni immagine.

Consideriamo la seguente immagine:

```
img = cv2.imread('dog.jpg')
see(img, 'Cane', 10000)
```



```
(b,g,r) = img[0,0]
```

L'istruzione scomponete il colore del pixel iniziale dell'immagine, memorizzato nella posizione ((0,0): nell'angolo in alto a sinistra), e lo memorizza in 3 variabili che stanno a rappresentare i tre canali dell'immagine: blu (b), verde (g) e rosso (r).

Da notare che la struttura del colore bgr è specifica di openCV, mentre il formato RGB è più comunemente utilizzato. Ciò significa che l'hardware trasmette immagini in formato bgr, ma per lavorare con le immagini è necessario convertirle in RGB, tramite lo scambio dei canali (red, green, blue) dell'array.

Il pixel memorizzato è una sezione dell'immagine, tuttavia la sua visualizzazione non sarebbe significativa.

Possiamo però stampare il valore:

```
print(f'({b}, {g}, {r})')
# (216, 216, 216) colore sabbia/grigio, si avvicina al bianco: (255,255,255)
```

3.5.1 Modifica pixel immagine.

Possiamo cambiare il valore del pixel con l'istruzione:

```
img[0,0] = (0, 0, 255) # lo modifichiamo in rosso.
```

Perciò se stampiamo il valore dopo il cambiamento:

```
print(f'Dopo il cambiamento: {img[0,0]}') # (0, 0, 255): rosso
```

3.5.2 Modifica porzione immagine.

Possiamo selezionare una porzione dell'immagine con lo slicing.

Ad esempio se vogliamo un quadrato di 100 pixel che parte dall'angolo sinistro alto possiamo salvare questo valore in una variabile corner:

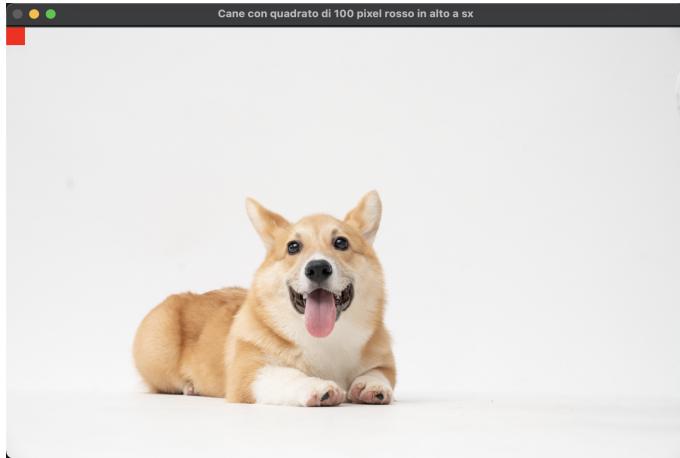
```
corner = img[0:100, 0:100]
```

Possiamo modificare questa parte, esattamente come nella stessa maniera del cambiamento di un pixel:

```
img[0:100, 0:100] = (0, 0, 255) # quadrato rosso
```

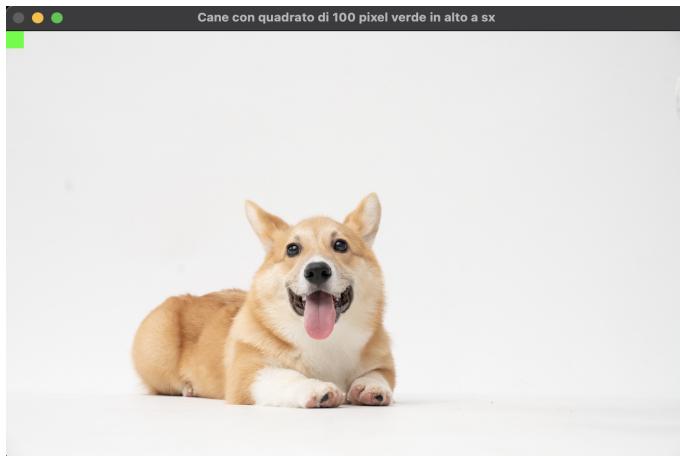
Proviamo a visualizzare adesso l'immagine:

```
see(img, 'Cane con quadrato di 100 pixel rosso in alto a sx', 0)
```



Se adesso modifichiamo la stessa parte di prima, la porzione dell'immagine viene sovra-scritta.

```
img[0:100, 0:100] = (0, 255, 0) # quadrato verde  
see(img, 'Cane con quadrato di 100 pixel verde in alto a sx', 0)
```



3.5.3 Disegno nell'immagine.

È possibile disegnare all'interno di un immagine.

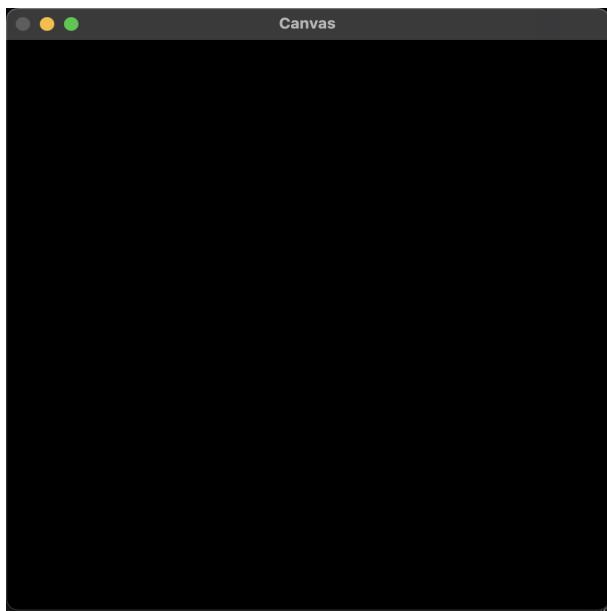
Inizialmente creiamo una tela vuota per l'elaborazione delle immagini o le attività di disegno, in cui a ogni pixel della tela può essere assegnato un valore di colore.

```
canvas = np.zeros((500,500, 3), dtype='uint8')
see(canvas, "Canvas", 0)
```

Questa riga di codice crea una matrice NumPy con dimensioni di 500x500 e tre canali di colore (rosso, verde, blu).

Il parametro ‘dtype=’uint8’ imposta il tipo di dati dell’array su interi senza segno (non negativi) a 8 bit, con valori compresi tra 0 e 255 (range dei pixel sui canali).

Infine viene mostrata la tela:



Definiamo un insieme di colori che risulteranno utili per l’attività di disegno:

```
blue = (255, 0, 0)
green = (0, 255, 0)
red = (0, 0, 255)
```

A questo punto è possibile disegnare delle figure geometriche su un’immagine utilizzando le funzioni di disegno fornite dalla libreria. (cv2.line, cv2.rectangle, cv2.circle)

Vediamoli tutti:

3.5.4 Disegno LINEA nell'immagine.

La funzione "cv2.line" di OpenCV viene utilizzata per disegnare una linea su un'immagine.

Di seguito è riportata la sua sintassi:

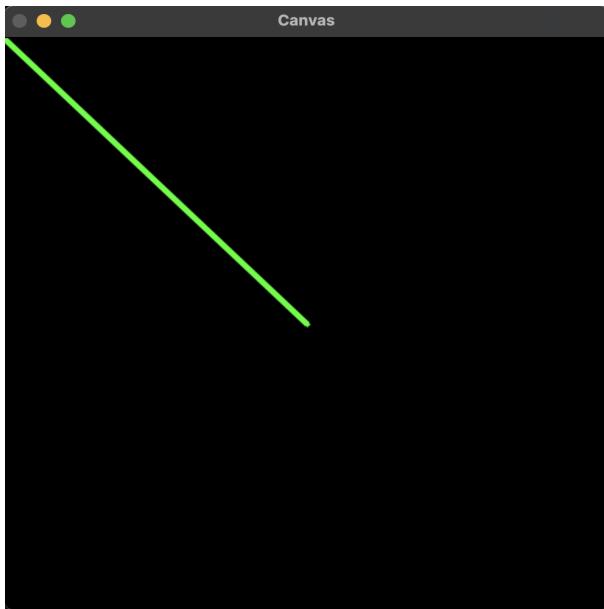
```
cv::line(image, start_point, end_point, color, thickness)
```

dove:

- 'image' è l'immagine su cui disegnare la linea
- 'start_point' e 'end_point' sono le coordinate del punto di origine e di destinazione della linea, rispettivamente.
- 'color' è il colore della linea
- 'thickness' è lo spessore della linea.

Definiamo una linea diagonale verde che passa dal punto (0,0) al punto (250, 250): centro di spessore 3 (spessore dato dal 4° parametro). Questa linea parte dall'angolo sinistro alto e arrivo al centro:

```
cv2.line(canvas, (0,0), (250, 250), green, 3)
see(canvas, "Canvas", 0)
```



3.5.5 Disegno RETTANGOLO nell'immagine.

Il metodo ‘cv2.rectangle()‘ disegna un rettangolo sull’immagine fornita.

La sintassi del metodo ‘cv2.rectangle‘ è:

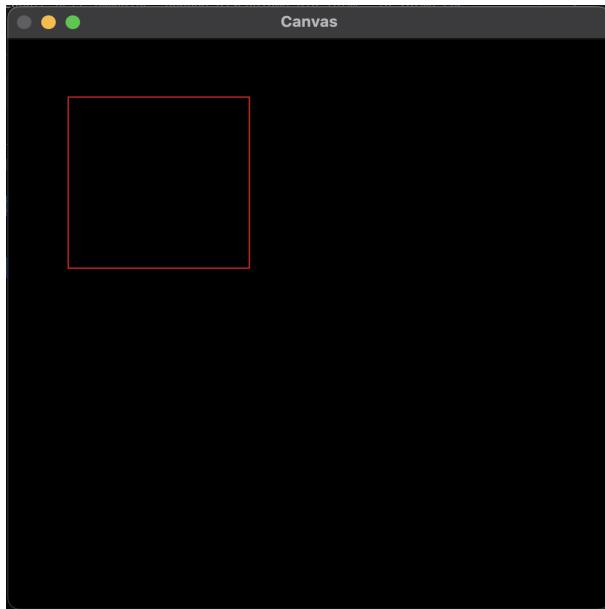
```
cv2.rectangle(img, pt1, pt2, color[, thickness[, lineType[, shift]]])
```

dove:

- ‘img’: L’immagine su cui viene disegnato il rettangolo.
- ‘pt1’: Vertice del rettangolo. L’angolo in alto a sinistra del rettangolo è (x1, y1).
- ‘pt2’: Vertice del rettangolo opposto a pt1. L’angolo in basso a destra del rettangolo è (x2, y2).
- ‘color’: Il colore del rettangolo. Viene rappresentato da una tupla con ordinamento ‘(b, g, r)‘.
- ‘thickness’: La larghezza dei bordi del rettangolo. Se viene passato ‘-1‘ per la larghezza, il rettangolo viene riempito invece di disegnarne i bordi.
- ‘lineType’: Tipo di linea da utilizzare (‘8‘, ‘4‘). Di default, viene utilizzata una linea ‘8‘.
- ‘shift’: Numero di bit frazionari nelle coordinate dei punti, utilizzato per la precisione sub-pixel. Di default, è ‘0‘.

Disegniamo un rettangolo rosso che parte dal pixel ((50, 50):angolo in alto a sinistra) e arriva a ((200, 200):angolo in basso a destra) con uno spessore non definito, quindi di default 1:

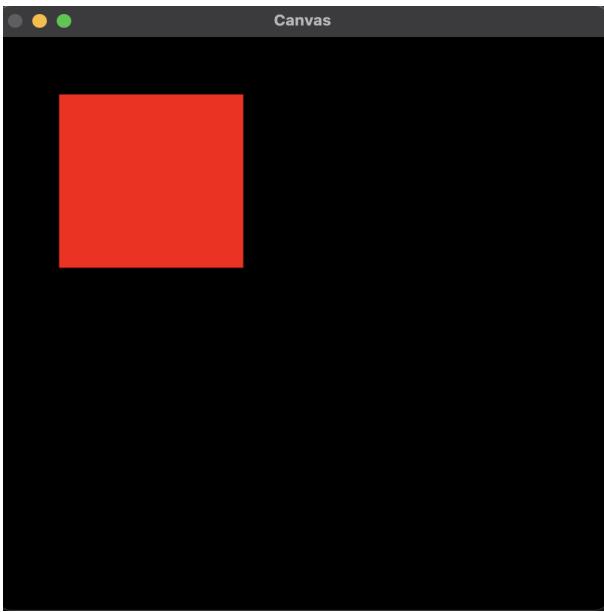
```
cv2.rectangle(canvas, (50, 50), (200, 200), red)
see(canvas, "Canvas", 0)
```



Disegniamo ora un rettangolo rosso pieno nelle stesse coordinate di prima:

```
cv2.rectangle(canvas, (50, 50), (200, 200), red, -1)
see(canvas, "Canvas", 0)
```

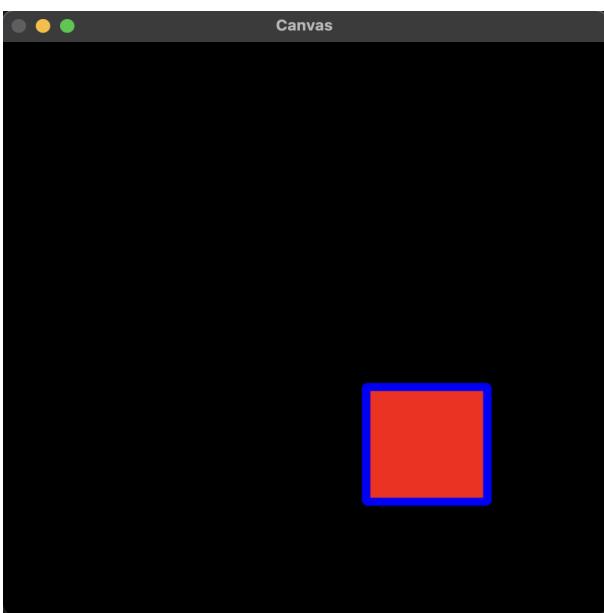
Per la creazione di un rettangolo di colore pieno si deve impostare il parametro ‘thickness’ con un valore ‘-1’.



Se vogliamo rappresentare un rettangolo con bordo blu e interno rosso come possiamo fare?

L'algoritmo che utilizziamo è quello di creare prima un rettangolo pieno rosso, e dopo di che un rettangolo non pieno con il bordo blu.

```
cv2.rectangle(canvas, (300, 300), (400, 400), red, -1)
cv2.rectangle(canvas, (300, 300), (400, 400), blue, 5)
see(canvas, "Canvas", 0)
```



3.5.6 Disegno CERCHIO nell'immagine.

Il metodo ‘cv2.circle()‘ disegna un rettangolo sull’immagine fornita. La sintassi del metodo ‘cv2.circle‘ è:

```
cv2.circle(img, center, radius, color[, thickness[, lineType[, shift]]]) -> img
```

dove:

- ‘img‘: l’immagine su cui disegnare il cerchio.
- ‘center‘: le coordinate del centro del cerchio. Di solito vengono fornite come una tupla (x, y).
- ‘radius‘: il raggio del cerchio in pixel.
- ‘color‘: il colore del cerchio, rappresentato come una tupla (B, G, R).
- ‘thickness‘: lo spessore del bordo del cerchio. Se viene impostato su -1, il cerchio verrà riempito anziché disegnato.
- ‘lineType‘: il tipo di linea del bordo del cerchio (opzionale, il valore predefinito è 8).
- ‘shift‘: il numero di bit frazionari nelle coordinate del centro e del raggio (opzionale, il valore predefinito è 0).

Disegniamo un cerchio: sicuramente serve il centro del cerchio e il raggio.

Se voglio il centro del cerchio al centro dell’immagine:

```
center_x , center_y = canvas.shape[1]/2 , canvas.shape[0]/2
```

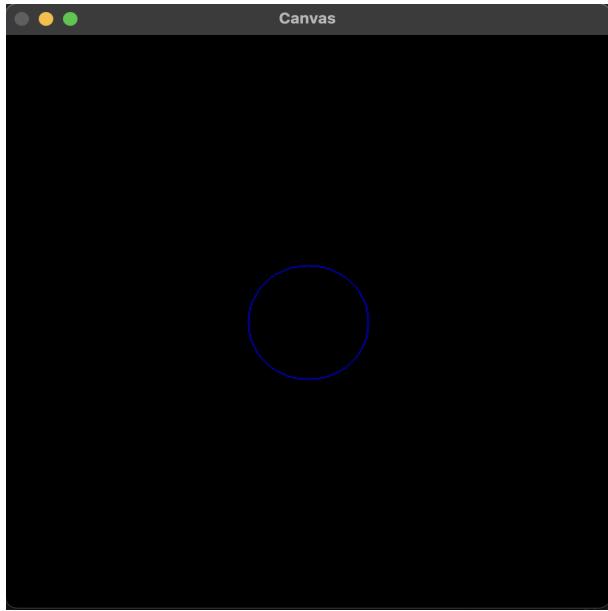
Questa istruzione di Python assegna alle variabili ‘center_x‘ e ‘center_y‘ il valore della metà della larghezza e dell’altezza dell’array ‘canvas‘.

Dividendo la dimensione delle colonne per 2 e la dimensione delle righe per 2 si ottengono le coordinate del punto centrale dell’immagine. Questi valori vengono assegnati alle variabili ”center_x“ e ”center_y“.

A questo punto:

```
cv2.circle(canvas, (center_x, center_y), 50, blue)
see(canvas, "Canvas", 0)
```

Come abbiamo detto il secondo argomento (center_x, center.y) indica le coordinate del centro del cerchio, mentre il terzo argomento (50) indica il raggio del cerchio in pixel.



Cosa accade con un canvas dove è presente un singolo canale (imagine grayscale)?

```
canvas_no_channels = np.zeros((500,500), dtype='uint8')
```

Di tutte le cose che abbiamo disegnato nel canvas normale possiamo vedere solo il cerchio.

Questo perchè ogni volta che passiamo il parametro del colore dentro le varie forme geometriche in un canvas senza canali viene preso solo il primo valore tra i 3: (255,0,0) quindi in questo caso viene preso solo il valore 255.

Per questo si vede solo il colore blue.

Se mettessimo 125 avremo il grigio, 50 avremo il grigio più scuro.

Quindi non ci sono errori nel codice, ma tutte le forme geometriche disgnate non vengono mostrate perchè prendono come pixel il nero (0) (il primo numero tra i 3 dei colori verdi e rossi) in un canvas che è già nero.

3.6 Image Processing.

Il "Processing delle immagini" (Image Processing) è un campo dell'Informatica e dell'Ingegneria che si occupa dell'analisi, del processing e della manipolazione di immagini. Questo campo di studio si concentra sulla creazione di algoritmi, tecniche e metodologie per modificare le immagini digitali in modo da migliorarne la qualità, estrarne informazioni o effettuarne la compressione e la codifica.

Abbiamo già caricato l'immagine del cane, utilizzando il metodo 'cv2.split(img)' di OpenCV, è possibile estrarre separatamente i tre canali (blu, verde e rosso) dall'immagine 'img'.

Questi canali potrebbero essere salvati separatamente o utilizzati per altre operazioni di elaborazione delle immagini.

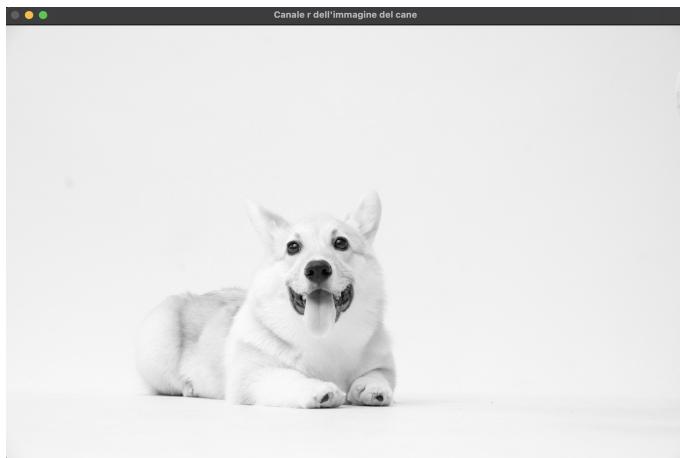
```
(b, g, r) = cv2.split(img)
```

In alternativa, è possibile utilizzare NumPy per estrarre uno specifico canale, ad esempio il canale blu, utilizzando l'indicizzazione a 3 dimensioni:

```
blue = img[:, :, 0]
```

Se a questo punto decidessimo di mostrare r:

```
see(r, "Canale r dell'immagine del cane", 2000)
```



Possiamo vedere che si tratta di una graysclale image. Ma questo non avviene solo con il canale 'r', ma anche con i canali 'g' e 'b'.

Questo perchè ogni canale d'immagine è in effetti una immagine in scala di grigi (unico canale), mentre il colore può essere creato solo quando si combinano i tre canali insieme per ricreare l'immagine a colori complete.

Combiniamo di nuovo i tre canali con l'istruzione:

```
img = cv2.merge((b,g,r))
```

L'immagine torna nel suo colore originale. Potremmo ricostruire i colori con la notazione RGB, ma i colori cambiano da quelli dell'immagine originale.

```
img2 = cv2.merge((r,g,b))
```

3.7 Conversione BGR - RGB.

BGR è il formato di immagine predefinito in OpenCV. Tuttavia, in alcune librerie esterne, come ad esempio matplotlib, l'immagine è rappresentata con un ordine di canali RGB. Quindi, se vogliamo visualizzare correttamente un'immagine OpenCV in una libreria esterna come matplotlib, dobbiamo convertirla in formato RGB.

```
rgb_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

Questo codice utilizza la funzione ‘cv2.cvtColor()‘ di OpenCV per convertire l'immagine ‘img‘ dal formato BGR (Blu-Verde-Rosso) a RGB (Rosso-Verde-Blu).

La funzione ‘cv2.cvtColor()‘ prende due argomenti:

- l'immagine da convertire;
- il codice di conversione che specifica il tipo di conversione da eseguire.

In questo caso, ‘cv2.COLOR_BGR2RGB‘ indica che l'immagine deve essere convertita dal formato BGR al formato RGB.

Dopo la conversione, l'immagine convertita viene assegnata a ‘rgb_img‘, che può essere utilizzata per ulteriori elaborazioni o visualizzazioni dell'immagine in formato RGB.

3.8 Flag caricamento immagini.

Si possono inserire alcuni flag durante il caricamento dell'immagine, se inseriamo:

- ’0‘ : l'immagine viene convertita durante il caricamento in un unico canale (grayscale image):

```
img = cv2.imread('nome_file_immagine.jpg', 0)
```

- ’1‘ : l'immagine viene convertita durante il caricamento in bgr standard:

```
img = cv2.imread('nome_file_immagine.jpg', 1)
```

- ’-1‘ : l'immagine viene letta ”as is”, cioè viene caricata in modo ”grezzo” , senza alcuna modifica, mantenendo tutti i canali di colore e la trasparenza (se presente). Quindi anche ulteriori canali, oltre ai 3 standard.

```
img = cv2.imread('nome_file_immagine.jpg', -1)
```

3.9 Transparency: nuovo canale in RBG: RGBA

La trasparenza in OpenCV Python si riferisce alla capacità di rendere trasparenti le parti di un'immagine che non sono visibili o non sono desiderate. Questo può essere utile in molte applicazioni, come la sovrapposizione di immagini, la creazione di maschere di immagine o la rimozione dello sfondo di un'immagine.

```
transp_img = cv2.cvtColor(img, cv2.COLOR_BGR2BGRA)
```

Questo codice è utilizzato per convertire un'immagine in formato BGR (Blu-Verde-Rosso) in un formato BGRA (Blu-Verde-Rosso-Trasparenza).

In altre parole, aggiunge un canale di trasparenza all'immagine, consentendo di sovrapporla su altre immagini o sfondi senza coprire ciò che si trova dietro di essa.

In computer vision viene utilizzato poco.

Soltimamente 'imshow' prima di mostrare l'immagine perde la trasparenza.

```
alpha = np.full((img.shape[0], img.shape[1]), fill_value=255, dtype='uint8')
transp_img2 = np.dstack([img, alpha])
```

Il codice crea una nuova immagine (chiamata "transp_img2") a partire da un'altra immagine (indicata come "img") con l'aggiunta di un nuovo canale di trasparenza (chiamato "alpha").

In particolare, viene utilizzata la funzione NumPy "full" per creare un'array (chiamato "alpha") della stessa dimensione dell'immagine originale, inizializzata con il valore 255 in ogni posizione.

Il valore 255 in ogni posizione dell'array alpha rappresenta la massima opacità, mentre un valore di 0 rappresenta la massima trasparenza.

Questo nuovo array rappresenta la maschera di trasparenza dell'immagine.

Successivamente, la funzione NumPy "dstack" viene utilizzata per creare un nuovo array con tre canali utilizzando l'immagine originale e l'array di trasparenza creato in precedenza.

In pratica la funzione "dstack" impila gli array passati come input in modo che il terzo asse (depth) dell'array risultante sia formato dalle immagini impilate.

L'array risultante (chiamato "transp_img2") diventa quindi un'immagine con una nuova dimensione (4 canali) rispetto all'immagine originale (3 canali).

L'aggiunta del canale di trasparenza rende l'immagine trasparente in alcune regioni.

Ad esempio, se il valore di un pixel nel canale alfa è zero, quel pixel sarà completamente trasparente e il contenuto dell'immagine sottostante sarà visibile attraverso di esso.

3.10 Trasformazione geometrica delle immagini.

Le trasformazioni geometriche delle immagini sono operazioni che modificano l'aspetto di un'immagine, ad esempio rendendola più grande o più piccola, ruotandola o spostandola.

Ci sono diverse tipologie di trasformazioni geometriche, tra cui:

- **Scaling:** questa trasformazione rende l'immagine più grande o più piccola rispetto alle sue dimensioni originali. La scala dell'immagine può essere specificata manualmente (ad esempio, raddoppiando la dimensione dell'immagine) o calcolata automaticamente (ad esempio, per farla adattare a un'area di visualizzazione).
- **Translation:** questo tipo di trasformazione sposta l'immagine lungo l'asse x o l'asse y. Le coordinate di spostamento possono essere specificate manualmente o calcolate automaticamente.
- **Rotation:** questa trasformazione ruota l'immagine intorno ad un punto di riferimento (ad esempio, il centro dell'immagine) di un certo angolo.
- **Perspective transformation:** questa trasformazione consente di deformare l'immagine in modo da simularne la prospettiva. È utile quando si vuole creare un'immagine composta da diverse porzioni provenienti da diverse angolazioni. Per questa trasformazione sono necessari 4 punti di riferimento.
- **Affine transformation:** questa trasformazione consente di combinare diverse immagini in un'unica immagine in base a punti comuni. I punti comuni possono essere selezionati manualmente o calcolati automaticamente. Per questa trasformazione sono necessari 3 punti di riferimento.

Una delle applicazioni comuni della prospettiva affine transformation è la creazione di foto panoramiche a partire da multiple foto scattate dallo stesso punto di vista, utilizzando i punti di riferimento comuni per unire le immagini. Ci sono sia metodi manuali che automatici per selezionare i punti di riferimento.

3.10.1 Matrice di trasformazione.

Le trasformazioni geometriche delle immagini, come la rotazione, la traslazione, lo scaling e la deformazione, possono essere espresse matematicamente utilizzando le matrici di trasformazione. Una matrice di trasformazione è una matrice quadrata che rappresenta una trasformazione geometrica.

In OpenCV, le trasformazioni geometriche vengono eseguite utilizzando la funzione cv2.warpAffine o cv2.warpPerspective, che richiedono come input una matrice di trasformazione.

La matrice di trasformazione dipende dal tipo di trasformazione geometrica che si vuole eseguire.

Ad esempio per la **traslazione**, la matrice di trasformazione è una matrice 2x3 che ha la seguente forma:

```
[[1, 0, tx],  
 [0, 1, ty]]
```

dove tx e ty sono le traslazioni orizzontale e verticale, rispettivamente. Questa matrice viene moltiplicata per le coordinate di ogni pixel nell'immagine per eseguire la traslazione.

Per la **rotazione**, la matrice di trasformazione è una matrice 2x2 che ha la seguente forma:

```
[[cos(theta), -sin(theta)],  
 [sin(theta), cos(theta)]]
```

dove theta è l'angolo di rotazione in radianti. Questa matrice viene moltiplicata per le coordinate di ogni pixel nell'immagine per eseguire la rotazione.

Per lo **scaling**, la matrice di trasformazione è una matrice 2x2 che ha la seguente forma:

```
[[sx, 0],  
 [0, sy]]
```

dove sx e sy sono i fattori di scala orizzontale e verticale, rispettivamente. Questa matrice viene moltiplicata per le coordinate di ogni pixel nell'immagine per eseguire lo scaling.

Le trasformazioni geometriche affini e prospettiche sono più complesse rispetto alla semplice traslazione, rotazione e scalatura. Per eseguire queste trasformazioni, è necessario utilizzare matrici di trasformazione 2x3 e 3x3, rispettivamente.

Una trasformazione **affine** può essere rappresentata dalla seguente matrice di trasformazione:

```
[[a11, a12, b1],  
 [a21, a22, b2]]
```

dove a11 e a22 sono i fattori di scala per i due assi, a12 e a21 sono i coefficienti di inclinazione e b1 e b2 sono le traslazioni sui due assi. In questo caso, la trasformazione può essere vista come una combinazione di traslazione, rotazione, scalatura e inclinazione.

Per le trasformazioni prospettiche, invece, è necessaria una matrice di trasformazione 3x3:

```
[[h11, h12, h13],  
 [h21, h22, h23],  
 [h31, h32, h33]]
```

dove h11, h12, h13, h21, h22, h23, h31, h32 e h33 sono i parametri di trasformazione. In questo caso, la trasformazione può essere vista come una proiezione da uno spazio 3D a uno spazio 2D. La trasformazione prospettica è in grado di produrre effetti di deformazione più complessi rispetto alla trasformazione affine, come ad esempio la simulazione di una vista prospettica di un'immagine.

L'ultima riga della matrice è sempre 0 0 1 (spesso omessa), che indica che non ci sono modifiche dell'immagine lungo l'asse z (profondità).

Inoltre, la matrice viene utilizzata anche per la prospettiva transformation, dove sono necessari quattro punti di riferimento, invece di tre. In questo caso, la matrice 3x3 viene estesa a una matrice 3x4, con la quarta colonna contenente i valori necessari per la prospettiva.

Ci sono diverse funzioni e metodi in Python che permettono di specificare e applicare queste matrici per le trasformazioni geometriche delle immagini.

3.10.2 Scaling.

Come abbiamo già detto, questa trasformazione rende l'immagine più grande o più piccola rispetto alle sue dimensioni originali.

Possiamo utilizzare ‘cv2.resize()’, una funzione di OpenCV che consente di ridimensionare un’immagine a una nuova dimensione specificata. La funzione prende come argomenti l’immagine di input e le nuove dimensioni (in pixel), e restituisce l’immagine ridimensionata.

Ecco le caratteristiche principali della funzione ‘cv2.resize()’:

- È possibile scegliere il tipo di algoritmo di ridimensionamento da utilizzare, come ad esempio ‘INTER_LINEAR’ (lineare), ‘INTER_CUBIC’ (cubico) o ‘INTER_NEAREST’ (il più vicino).
- È possibile scegliere se mantenere o meno l’aspetto originale dell’immagine durante la ridimensione, utilizzando l’opzione ‘cv2.INTER_AREA’.
- È possibile ridimensionare l’immagine utilizzando il fattore di scala invece delle dimensioni, specificando ‘fx’ e ‘fy’ come valori di scala orizzontale e verticale.
- È possibile specificare se l’immagine di output deve essere ridimensionata in modo che il valore massimo o il valore minimo della dimensione sia uguale alla dimensione richiesta, utilizzando l’opzione ‘cv2.WARP_FILL_OUTLIERS’.
- È possibile specificare il tipo di metodo di interpolazione bilineare utilizzato per la ridimensione dell’immagine, utilizzando l’opzione ‘cv2.INTER_LINEAR_EXACT’.

In generale, la funzione ‘cv2.resize()’ è molto utile nelle applicazioni di elaborazione delle immagini, ad esempio per ridimensionare le immagini per la visualizzazione o per la riduzione del rumore nelle immagini ad alta risoluzione.

Consideriamo un esempio, innanzitutto ci salviamo dei valori fondamentali (altezza e base).

```
height = img.shape[0]
width = img.shape[1]
```

Ridimensioniamo l’immagine al doppio della sua grandezza originale, utilizzando l’interpolazione bilineare (‘cv2.INTER_LINEAR’) per calcolare i pixel della nuova immagine. La nuova dimensione viene specificata come una tupla di valori ‘(width * 2, height * 2)’.

```
img_resizing = cv2.resize(img, (width * 2, height * 2), interpolation=cv2.INTER_LINEAR)
```

Possiamo anche ridimensionare l’immagine al doppio della sua grandezza originale, ma questa volta utilizzando l’interpolazione bicubica (‘cv2.INTER_CUBIC’), che è più lenta ma produce immagini con maggiore qualità.

```
img_resizing = cv2.resize(img, (width * 2, height * 2), interpolation=cv2.INTER_CUBIC)
```

Se vogliamo invece l’immagine di metà più piccola:

```
height_met = int(img.shape[0] * 0.5)
width_met = int(img.shape[1] * 0.5)
img_resizing2 = cv2.resize(img, (width_met, height_met))
```

In questo caso il riscalamento viene eseguito in modo uniforme su tutte le dimensioni (in questo caso, sulla larghezza e l’altezza, rispettivamente).

Possiamo anche passare anche altri parametri per la scaling ($fx = 2$, $fy = 2$, raddoppiare l'immagine, $fx,fy=0.5$ diminuire l'immagine in 2):

```
img_resizing2 = cv2.resize(img, None, fx=0.5, fy=0.5)
```

Viene di nuovo ridimensionata l'immagine passando come argomenti il valore di scala orizzontale e verticale, utilizzati per dimezzare la grandezza dell'immagine. La funzione viene chiamata usando l'opzione 'None' come primo argomento, poiché non è necessario specificare l'immagine di output di destinazione.

3.10.3 Translation - traslazione immagine.

La traslazione di un'immagine consiste nello spostamento dell'immagine stessa in una determinata direzione.

Vediamo la traslazione di un'immagine rispetto all'asse x e y.

Definiamo gli assi x e y, che sono i valori di traslazione desiderati:

```
x = 200  
y = 30
```

Creiamo la matrice di transizione con i valori di traslazione x e y:

```
M = np.float32([[1,0,x], [0,1,y]])
```

La prima riga contiene i valori $[1, 0, x]$, dove x rappresenta il valore della traslazione lungo l'asse x (orizzontale).

La seconda riga contiene i valori $[0, 1, y]$, dove y rappresenta il valore della traslazione lungo l'asse y (verticale).

Questi valori determinano la quantità di pixel di spostamento dell'immagine.

In particolare, la matrice M rappresenta una traslazione di $x=200$ pixel verso destra e $y=30$ pixel verso il basso. Questo significa che l'immagine verrà spostata di 200 pixel verso destra e 30 pixel verso il basso rispetto alla sua posizione originale.

Se ci vogliamo muovere da dx verso sx x deve essere negativo e se vogliamo muoverci dal basso verso l'alto un numero negativo per y.

Successivamente, la funzione cv2.warpAffine viene utilizzata per applicare questa matrice di transizione all'immagine (img), generando l'immagine di destinazione traslata (destination_img)

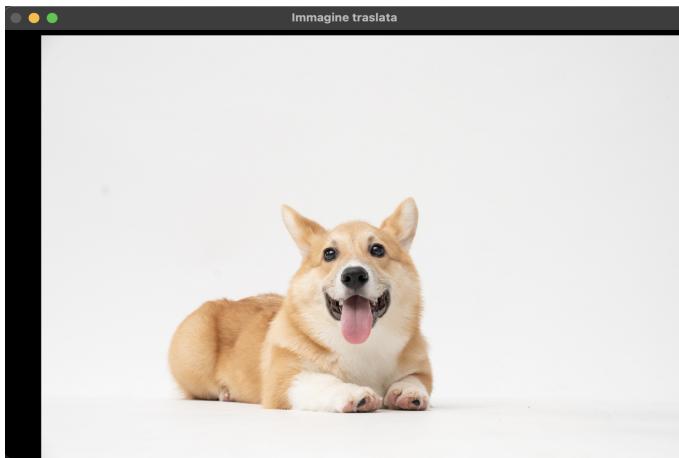
```
destination_img = cv2.warpAffine(img, M, (img.shape[1], img.shape[0]))
```

In particolare, la funzione cv2.warpAffine richiede tre parametri:

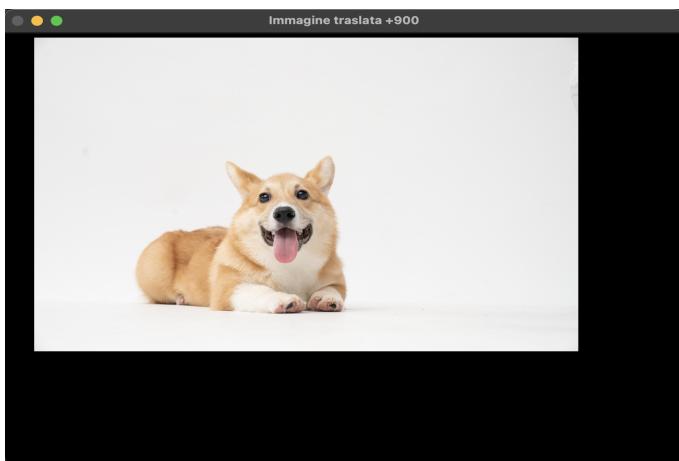
- l'immagine di origine (img);
- la matrice di trasformazione (M);
- le dimensioni dell'immagine di destinazione (img.shape[1], img.shape[0]);

Puoi modificare i valori di traslazione x e y per ottenere diverse traslazioni dell'immagine e vedere come cambia l'immagine di destinazione.

In questo caso l'immagine è stata traslata a dx, se utilizziamo x = -200, l'immagine viene traslata a sx.



Inoltre, è possibile modificare le dimensioni dell'immagine di destinazione (utilizzando ad esempio `img.shape[1]+900`, `img.shape[0]+900`), per rendere l'immagine risultante più grande o più piccola. Grazie a questo incremento riusciamo a vedere maggiormente la traslazione.



3.10.4 Rotation - rotazione.

La rotazione di un'immagine consiste nella trasformazione dell'immagine stessa tramite una rotazione attorno a un punto di riferimento.

In OpenCV, questo può essere fatto utilizzando la funzione `cv2.getRotationMatrix2D()` per ottenere la matrice di rotazione, e `cv2.warpAffine()` per applicarla all'immagine.

Definiamo innanzitutto 2 variabili che indicano l'altezza e la larghezza dell'immagine:

```
height, width = img.shape[:2]
```

A questo punto possiamo creare la matrice di rotazione:

```
M = cv2.getRotationMatrix2D((width//2, height//2), 180, 1)
```

La funzione `cv2.getRotationMatrix2D()` è utilizzata per ottenere una matrice di trasformazione per eseguire una rotazione 2D di un’immagine. I suoi parametri sono i seguenti:

- center: è una tupla che rappresenta le coordinate del punto intorno al quale deve avvenire la rotazione. Di solito, questo punto è il centro dell’immagine e viene specificato come `(width/2, height/2)`. In questo caso `(width//2, height//2)` è il punto di rotazione dell’immagine, ovvero il centro dell’immagine.
- angle: è l’angolo di rotazione in gradi. Se l’angolo è positivo, l’immagine ruota in senso antiorario, mentre se è negativo ruota in senso orario. Nel nostro caso viene indicato un angolo di 180° .
- scale: è il fattore di scala che indica se l’immagine deve essere ridimensionata durante la rotazione. Se `scale=1`, l’immagine è ruotata senza ridimensionamento. (dimensione dell’immagine di destinazione)

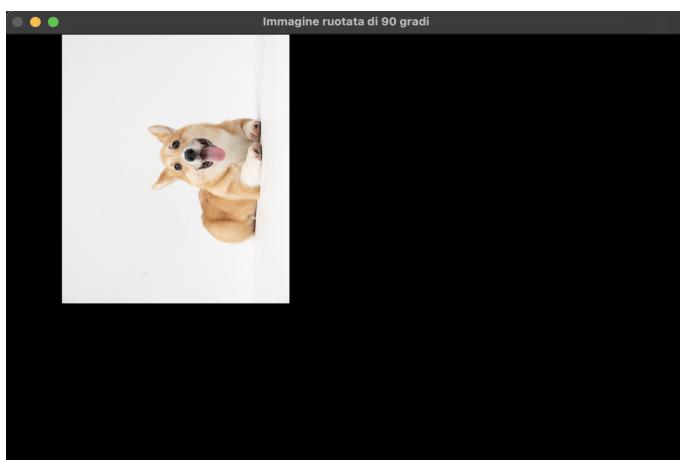
```
rotate_img = cv2.warpAffine(img, M, (width, height))
```

Questa matrice M viene quindi utilizzata con la funzione `cv2.warpAffine` per applicare la rotazione all’immagine originale. L’immagine ruotata viene quindi assegnata alla variabile `rotate_img`.



Quando ruotiamo un’immagine di 90 gradi, potrebbe accadere che l’immagine esca dai bordi perché la forma diventa rettangolare orizzontalmente. In questo caso, possiamo ridimensionare l’immagine passando al metodo `cv2.warpAffine()` una dimensione più grande rispetto a quella dell’immagine originale, utilizzando l’operatore `*`. Al contrario, se volessimo una dimensione più piccola, possiamo utilizzare l’operatore `.`

```
M = cv2.getRotationMatrix2D((width//2, height//2), 90, 1)
rotate_img = cv2.warpAffine(img, M, (width * 2, height * 2))
```



3.10.5 Affine transformation.

L’Affine transformation è una trasformazione geometrica 2D che preserva la forma degli oggetti e le linee rette, ma può modificare la scala, la rotazione, la traslazione e l’inclinazione degli oggetti nell’immagine.

In OpenCV, la trasformazione affine viene eseguita utilizzando una matrice di trasformazione affine 2x3, che viene definita come una combinazione di trasformazioni di traslazione, rotazione, scalatura e inclinazione. La matrice di trasformazione affine viene moltiplicata per le coordinate di ogni pixel dell’immagine originale per ottenere le coordinate dei pixel corrispondenti nell’immagine di output trasformata. L’Affine transformation può essere utilizzata per correggere le distorsioni geometriche dell’immagine, allineare le immagini, ruotare e ridimensionare le immagini.

Per creare questa matrice di trasformazione, abbiamo bisogno di due set di punti: i punti di partenza (`pts1`) e i punti di destinazione (`pts2`). Questi punti verranno abbinati tra loro per creare la matrice di trasformazione.

```
pts1 = np.float32([[135, 45], [200, 80], [155, 200]])
pts2 = np.float32([[200, 70], [250, 100], [175, 210]])
```

In questo caso, abbiamo creato due set di punti (`pts1` e `pts2`) contenenti tre coppie di coordinate x e y. Questi punti verranno abbinati insieme per creare la matrice di trasformazione M.

La funzione `cv2.getAffineTransform` prende in input i due set di punti e restituisce la matrice di trasformazione M.

Questa matrice è una matrice 2x3, che contiene i parametri di trasformazione per l’Affine Transformation.

In particolare, la matrice ha la seguente struttura:

```
[[a11, a12, b1],
 [a21, a22, b2]]
```

dove:

- a_{11} e a_{12} rappresentano la scala in x e la rotazione,
- a_{21} e a_{22} rappresentano la scala in y e la rotazione,
- b_1 e b_2 rappresentano la traslazione in x e y.

Nell’esempio specifico, i valori molto piccoli della matrice di trasformazione sono dovuti alla presenza di punti molto vicini tra loro nei due set di punti.

Questi valori indicano la rotazione e la traslazione dell’immagine, e possono essere utilizzati per applicare la stessa trasformazione a un’altra immagine che mostra lo stesso oggetto in una posizione e in un’orientazione differente.

Ad esempio, se abbiamo un’immagine del telecomando posizionato orizzontalmente e un’altra immagine dello stesso telecomando posizionato verticalmente, possiamo utilizzare la matrice di trasformazione calcolata per applicare la stessa rotazione e traslazione all’immagine del telecomando verticale, in modo che le due immagini possano essere sovrapposte correttamente senza problemi.

3.10.6 Perspective transformation.

La perspective transformation è una trasformazione geometrica 2D che può modificare la prospettiva dell'immagine, ovvero l'angolo di visualizzazione dell'oggetto in modo che sembri essere visto da un punto di vista diverso. Questa trasformazione viene solitamente utilizzata per correggere le distorsioni prospettiche causate dalla fotocamera durante la ripresa di oggetti in posizioni non frontali.

In OpenCV, la perspective transformation viene eseguita utilizzando una matrice di trasformazione prospettica 3×3 (in questo caso 4×3 perché la riga 0 0 1 è omessa), che viene definita da quattro punti di controllo nell'immagine originale e le corrispondenti posizioni nell'immagine di output desiderata. Questi quattro punti di controllo devono essere specificati manualmente o automaticamente utilizzando tecniche di rilevamento dei punti chiave.

```
a1 a2 tx
a3 a4 ty
0 0 1
w1 w2 1
```

Per la perspective transformation serve un'altra riga, dove w1, w2 servono a cambiare la prospettiva dell'immagine.

La perspective transformation può essere utilizzata per creare effetti di zoom, inclinare o ruotare l'immagine in modo che sembri vista da un angolo diverso. È particolarmente utile per l'elaborazione di immagini per la visione artificiale, la realtà aumentata, la mappatura di texture su oggetti tridimensionali e la correzione di distorsioni prospettiche dell'immagine.

3.10.7 Perspective vs Affine.

La perspective transformation è utilizzata quando vogliamo proiettare un'immagine in una forma non affine, cioè quando i punti nell'immagine originale non sono necessariamente parallelamente allineati nella forma finale desiderata. Ad esempio, possiamo utilizzarla per correggere la prospettiva di un'immagine di un documento scattato da un angolo, grazie ai 4 punti della matrice. La matrice di trasformazione è chiamata homography.

L'affine transformation, invece, è utile per trasformare un'immagine in un'altra forma affine, cioè quando i punti dell'immagine originale sono parallelamente allineati nella forma finale desiderata. Ad esempio, possiamo utilizzarla per ruotare, ridimensionare o traslare un'immagine.

La perspective transformation è una tecnica molto utilizzata per creare immagini panoramiche e per effettuare lo scanning di documenti. In entrambi i casi, l'idea è quella di prendere più immagini e trasformarle in modo che abbiano la stessa prospettiva e quindi possano essere combinate insieme. Ad esempio, per creare una immagine panoramica, si possono scattare diverse foto dello stesso panorama, poi si utilizza la perspective transformation per trasformare tutte le immagini in modo che abbiano la stessa prospettiva e quindi possano essere unite insieme per creare una singola immagine panoramica. Invece, per lo scanning di documenti, si possono scattare diverse foto del documento e poi utilizzare la perspective transformation per trasformare tutte le immagini in modo che sembrino riprese da una singola prospettiva, in questo modo si può ottenere una singola immagine del documento che contiene tutte le informazioni necessarie.

3.10.8 Perspective transformation example.

Vogliamo creare uno scanner di documenti.

Date le due immagini vogliamo far sì che venga preso il foglio bianco da sopra il tavolo e creare così uno scanner di documenti.



(a) Immagine originale.



(b) Immagine dopo l'esecuzione dell'esercizio (documento di scanner).

In questo dobbiamo utilizzare la perspective transformation proprio perchè i bordi del foglio sono punti non parallelamente allineati nella forma finale desiderata.

Per svolgere l'esercizio dobbiamo innanzitutto trovare i punti di partenza e di destinazione per mapparli.

Per i punti di partenza possiamo semplicemente controllare a mano i bordi del foglio sul tavolo oppure utilizzare una funzione che permette di selezionare i punti con un click sul mouse oppure trovare questi punti automaticamente.

Consideriamo il primo caso, cioè che abbiamo trovato a mano i pixel che rappresentano i bordi del foglio e memorizziamoli in un array chiamato 'src_point'.

```
src_points = np.array([
    [28, 227], # top left corner
    [131, 987], # bottom left
    [730, 860], # bottom right
    [572, 149], # top right
], dtype=np.float32)
```

In questo caso la cosa è semplice perchè questi punti sono le coordinate del foglio bianco.

NOTA BENE: la funzione che prende questi punti come input pretende che questi punti siano del tipo plot 32. Se passiamo integer la funzione non funziona e ritorna un errore, perciò specifichiamo questo nel dtype.

A questo punto definiamo i punti di destinazione, dato che noi vogliamo l'immagine come un documento scansionato perfetto, dobbiamo far sì che:

- l'angolo in alto a sinistra sarà: (0,0);
- l'angolo in basso a sx deve essere : (0, e il numero di righe che vogliamo) ;
- l' angolo in basso a dx deve essere (il numero di colonne e il numero di righe che vogliamo);
- l'angolo in alto a dx sarà : (0, il numero di colonne che vogliamo per l'immagine);

Adesso selezioniamo i punti di destinazione dst_point, che determinano la dimensione dell'immagine di output.

Quando applichiamo la transformation, non possiamo essere sicuri che la nuova immagine calzerà perfettamente nelle dimensioni originali, quindi dobbiamo specificare la dimensione desiderata dell'immagine di output, in questo caso 600x800.

```
dst_point = np.array([
    [0, 0],
    [0, 800],
    [600, 800],
    [600, 0] # 0: prima riga
], dtype=np.float32)
```

Successivamente, utilizziamo la funzione cv2.getPerspectiveTransform per calcolare la matrice di trasformazione H utilizzando i punti di partenza src_points e di destinazione dst_point.

La matrice di trasformazione dovrà mappare il punto [28, 227] in [0,0], [131,987] in [0, 800], ...

```
H = cv2.getPerspectiveTransform(src_points, dst_point)
```

Infine, utilizziamo la funzione cv2.warpPerspective per applicare la matrice di trasformazione H all'immagine originale img e generare l'immagine di output output_img della dimensione specificata (600, 800).

```
output_img = cv2.warpPerspective(img, H, (600, 800))
```

Se mostriamo l'immagine di output avremo il foglio contentente la foto di gerry.

Adesso possiamo considerare il caso in cui i punti di partenza vengono selezionati con il mouse e non vengono trovati a mano. È possibile fare questo in 2 modi:

- rilevare automaticamente i punti rilevanti nell'immagine; (successivamente nel corso)
- cliccare i bordi del foglio e usare questi punti come source_point(punti di partenza)

Vediamo il secondo modo: quando si utilizza OpenCV per eseguire una trasformazione prospettica su un'immagine, spesso è necessario selezionare manualmente i quattro angoli dell'immagine.

OpenCV fornisce una funzione chiamata setMouseCallback che può essere utilizzata per impostare una funzione di callback che viene chiamata quando si verificano determinati eventi del mouse:

```
# callback definition: quando clicchiamo con il mouse.
def onClick(event, x, y, flags, param): # event: click mouse; x,y coordinate del mouse
    if event == cv2.EVENT_LBUTTONDOWN:
        if len(src_points2) < 4: # dobbiamo essere sicuri di avere 4 punti
            src_points2.append([x, y])
            cv2.circle(img_copy, (x,y), 10, (0,0,255), -1 )
            cv2.imshow("Img", img_copy)
```

La callback controlla se è stato fatto un clic sinistro del mouse e, se sì, aggiunge le coordinate del punto alla lista src_points2 e disegna un cerchio rosso nel punto cliccato sull'immagine img_copy (una copia dell'immagine originale).

Quindi creiamo una copia dell'immagine e definiamo l'array che conterrà i 4 punti di partenza:

```
img_copy = img.copy()  
src_points2 = []  
dst_point2 = dst_point.copy()
```

In questo caso i punti di destinazione sono i stessi del primo caso.

```
cv2.namedWindow("Img", cv2.WINDOW_NORMAL) # dobbiamo associare onclick alla window:  
cv2.setMouseCallback("Img", onClick)  
  
cv2.imshow("Img", img_copy) # mostriamo l'immagine di copia per prendere i punti di partenza:  
cv2.waitKey(0)
```

A questo punto dobbiamo garantire che src_points2 sia un numpy piuttosto che un array normale di python:

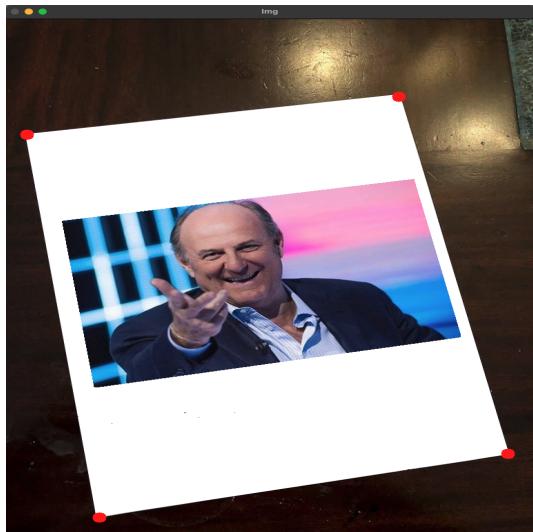
```
src_float2 = np.array(src_points2, dtype=np.float32)
```

Anche in questo caso calcoliamo la matrice di trasformazione H utilizzando i punti di partenza src_points2 e di destinazione dst_points2.

Applichiamo la matrice all'immagine originale e mostriamo l'immagine:

```
output_img2 = cv2.warpPerspective(img, H, (600, 800))  
cv2.imshow(output_img2, 3000)
```

Se facciamo partire il programma dovremmo innanzitutto selezionare i 4 punti in un ordine preciso: altosx, bassosx, bassodx, altodx.



Abbiamo creato un documento di scanner.

Potremmo migliorare il programma facendo sì che l'utente può cliccare in qualsiasi ordine.

3.11 Image Filtering.

I filtri in computer vision sono filtri che solitamente permettono di migliorare l'immagine o vengono utilizzati per estrarre informazioni rilevanti dalle immagini.

Ad esempio, il filtro di nitidezza può migliorare la definizione e la chiarezza dell'immagine, mentre il filtro di sfocatura può essere utilizzato per creare un effetto di sfocatura o bouquet. Inoltre, i filtri possono essere combinati per ottenere effetti specifici, come il filtro per creare un effetto cartoon.

Inoltre, i filtri possono anche essere utilizzati per estrarre i bordi dei soggetti, che è un'operazione molto comune in computer vision.

Per applicare i filtri in computer vision e image processing noi usiamo operazioni matematiche che sono chiamate **convoluzione**.

La convoluzione è un'operazione matematica utilizzata nell'elaborazione delle immagini per applicare un filtro (o kernel) all'immagine.

Il kernel è solitamente una matrice di dimensioni ridotte che viene spostata sull'immagine pixel per pixel, producendo un'immagine di output modificata in base al kernel applicato.

La convoluzione permette di combinare l'immagine di partenza e il kernel per ottenere una terza funzione, che è l'immagine di output. Il valore di ogni pixel nell'immagine di output viene calcolato come la somma dei prodotti dei valori dei pixel adiacenti nell'immagine di input moltiplicati con i valori corrispondenti del kernel.

La convoluzione è una tecnica importante nell'elaborazione delle immagini, utilizzata per la riduzione del rumore, l'accentuazione dei bordi e molte altre applicazioni.

Per applicare la convoluzione ad un'immagine, il kernel viene posizionato sul primo pixel dell'immagine e moltiplicato con i valori dei pixel adiacenti. Questi prodotti vengono sommati per ottenere il valore del nuovo pixel, che viene posizionato nella stessa posizione nella nuova immagine.

Ad esempio, se abbiamo l'immagine seguente:

```
1 2 3  
4 5 6  
7 8 9
```

E un kernel 3x3:

```
1 0 -1  
2 0 -2  
1 0 -1
```

Allora, per calcolare il valore del pixel nell'angolo in alto a sinistra della nuova immagine, il kernel viene posizionato sul pixel 1 dell'immagine di partenza:

Il valore del nuovo pixel è quindi calcolato come: $(1 * 1) + (2 * 0) + (3 * -1) + (4 * 2) + (5 * 0) + (6 * -2) + (7 * 1) + (8 * 0) + (9 * -1) = -8$.

Questo processo viene poi ripetuto per ogni pixel dell'immagine di partenza per ottenere la nuova immagine filtrata.

In sostanza, durante l'operazione di convoluzione, il kernel viene moltiplicato per ogni pixel dell'immagine e i risultati vengono sommati per ottenere un nuovo valore per quel pixel nell'immagine di output. Questo processo viene ripetuto per tutti i pixel dell'immagine, producendo l'immagine filtrata come risultato.

La convoluzione viene anche utilizzata come parte fondamentale dei modelli di deep learning per l'elaborazione delle immagini, in cui gli strati di convoluzione sono utilizzati per estrarre automaticamente le caratteristiche dell'immagine.

Facciamo alcuni esempi, innanzitutto carichiamo l'immagine

```
img = cv2.imread('data/beach.png')
```



A questo punto definiamo 2 kernel (matrici 3x3) che andiamo ad applicare sull'immagine:

```
my_kernel = np.array([[1,0,1],[1,0,1],[1,0,1]])
my_kernel2 = np.array([[-1,0,1], [-1,0,1], [-1,0,1]])
```

Applichiamo le matrici: my_kernel e my_kernel2 all'immagine:

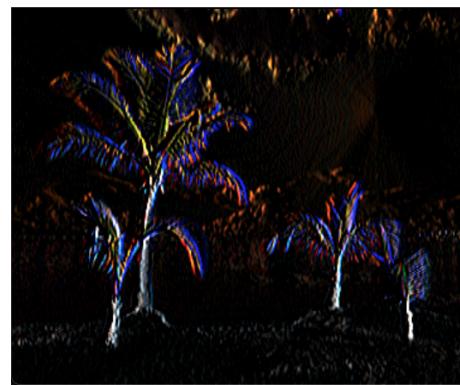
```
filtered_img = cv2.filter2D(img, -1, my_kernel)
filtered_img = cv2.filter2D(img, -1, my_kernel2)
```

La funzione cv2.filter2D esegue un'operazione di convoluzione tra l'immagine di input e il kernel, e restituisce l'immagine filtrata.

L'argomento -1 specifica che i canali dell'immagine in uscita deve essere la stessa dell'immagine di input, ovviamente il canale di trasparenza non viene usato nella convoluzione perchè non ha senso.



(a) Immagine con 'my_kernel'.



(b) Immagine con 'my_kernel2'.

Nell'immagine a sx il 'my_kernel' (filtro) applicato permette di aumentare il contrasto all'immagine.

Mentre nell'immagine a dx con il 'my_kernel2' vengono evidenziati i bordi verticali nell'immagine. In questo caso, il kernel "cerca" i cambiamenti di intensità tra i pixel orizzontali nell'immagine, e produce una risposta maggiore quando viene trovata una transizione tra i pixel orizzontali, come ad esempio un bordo verticale.

3.11.1 Effetto BLUR.

L'effetto blur in computer vision è spesso ottenuto applicando un filtro di smoothing o di sfocatura all'immagine. Questo tipo di filtro lavora ottenendo quindi una perdita di dettaglio e un effetto di "sfocatura" che può essere più o meno accentuato a seconda del tipo di filtro e dei suoi parametri.

In openCV, l'applicazione del filtro di blur si può ottenere in vari modi.

Uno dei modi è quello di definire una matrice (o kernel) di convoluzione, che viene poi applicata all'immagine con la funzione cv2.filter2D().

Il kernel di convoluzione rappresenta essenzialmente il filtro che viene applicato all'immagine, e la convoluzione stessa rappresenta l'operazione matematica di convoluzione tra il kernel e l'immagine.

Ad esempio, per applicare un filtro di blur a un'immagine si può utilizzare il kernel della media, che rappresenta un filtro che esegue una media dei pixel vicini ad ogni pixel dell'immagine. Questo kernel è definito come una matrice quadrata (ad esempio 3x3 o 5x5) con tutti i valori uguali, normalizzati in modo che la somma dei valori sia 1.

```
my_kernel_blur = np.array([ [1,1,1], [1,1,1], [1,1,1] ])
```

Visto che dobbiamo fare la media, la media è data dalla somma dei valori diviso numero di elementi, dobbiamo dividere tutti i valori per 9:

```
my_kernel_blur = my_kernel_blur/9
```

oppure costituito da tutti valori uguali a 5 e diviso per 45 (9*5) (stesso l'effetto blur).

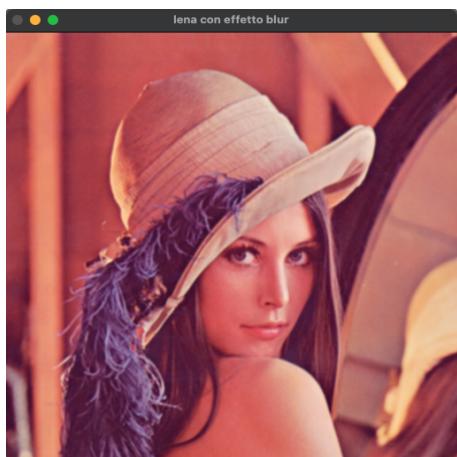
Applichiamo la matrice o kernel all'immagine:

```
filtered_img_blur_matrix = cv2.filter2D(img, -1, my_kernel_blur)
see(filtered_img_blur_matrix, 'lena con effetto blur', 20000)
```

In openCV, un altro modo per l'effetto blu è utilizzare la funzione cv2.blur() per applicare questo tipo di filtro di blur, specificando la dimensione del kernel di convoluzione. Ad esempio, per applicare un filtro di blur con un kernel 3x3, si può scrivere:

```
blurred_img = cv2.blur(img, (3, 3))
```

In questo caso per una maggior comprensione dell'effetto utilizziamo l'immagine della modella lena:



Supponiamo di avere l'immagine di lena con molto rumore (noise) anche chiamato 'sale e pepe' poichè assomiglia a quest'ultimo.



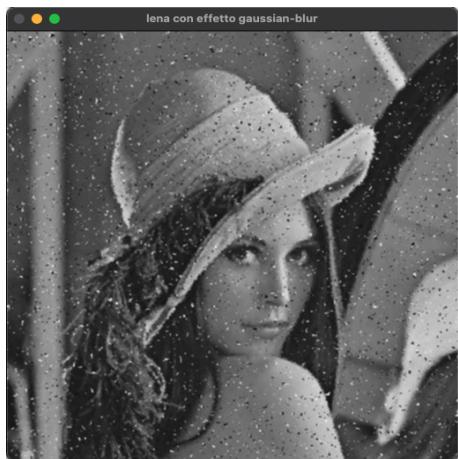
Con alcuni filtri blur possiamo ridurre il rumore:

Il filtro gaussiano è un tipo di filtro di smoothing, che consente di sfocare un'immagine per rimuovere il rumore o le imperfezioni. Il processo è basato sulla funzione di distribuzione gaussiana, che assegna un peso maggiore ai pixel vicini al centro e un peso minore ai pixel più lontani.

Per applicare il filtro gaussiano in OpenCV, si utilizza la funzione cv2.GaussianBlur, che prende in input l'immagine da filtrare, la dimensione del kernel e la deviazione standard del kernel.

```
filtered_img_gaussian = cv2.GaussianBlur(img, (3,3), 1)
```

Nel codice sopra, viene applicato il filtro gaussiano all'immagine img con un kernel di dimensione (3,3) e una deviazione standard di 1, il che significa che i pixel più vicini al centro avranno un peso maggiore rispetto ai pixel più lontani. Questo creerà un effetto di sfocatura più forte al centro e più leggero ai lati.

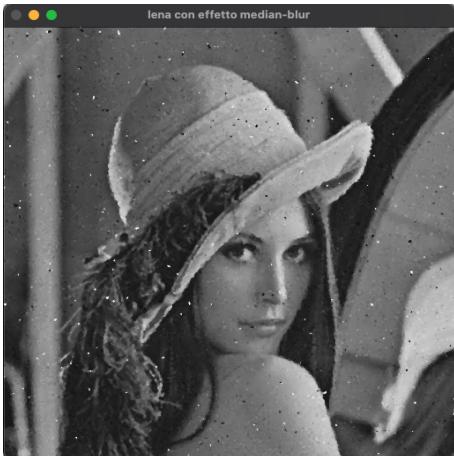


Median blur è una tecnica di elaborazione delle immagini utilizzata per l'eliminazione del rumore. Il filtro mediano sostituisce il valore del pixel con il valore mediano dei pixel del suo intorno definito da una finestra quadrata di dimensione predefinita. Questo tipo di filtro è molto efficace per rimuovere il rumore impulsivo, come il rumore "sale e pepe".

L'esempio che hai dato utilizza il metodo cv2.medianBlur() di OpenCV per applicare il filtro mediano con

una finestra di dimensione 3x3.

```
filtered_img_gaussian = cv2.medianBlur(img, 3)
```



3.11.2 Nitidezza immagine.

Ci sono dei filtri che permettono di incrementare la nitidezza, incrementare il dettaglio dell'immagine.

L'unsharp mask è una tecnica di miglioramento dell'immagine che prevede l'applicazione di un filtro di smoothing, come ad esempio il filtro Gaussiano o il filtro mediano, per rimuovere il rumore dall'immagine originale. Successivamente, viene sottratta dall'immagine originale l'immagine ottenuta con il filtro di smoothing, generando un'immagine che contiene solo gli alti dettagli dell'immagine originale, ovvero gli edge. Infine, questa immagine viene sommata all'immagine originale, in modo da migliorare la nitidezza e il contrasto dell'immagine finale.

L'unsharp mask può essere implementata attraverso la funzione "GaussianBlur" per applicare il filtro di smoothing, la funzione "subtract" per sottrarre le due immagini e la funzione "addWeighted" per sommare l'immagine risultante con l'immagine originale. Ad esempio:

```
img_gaussian = cv2.GaussianBlur(img, (9,9), 10) # (9,9) posso usare qualsiasi valore,  
l'importante e' che le dimensioni siano dispari
```

Abbiamo l'immagine originale e la stessa immagine con il blur (in questo caso utilizziamo il GaussianBlur che è un effetto non troppo aggressivo)

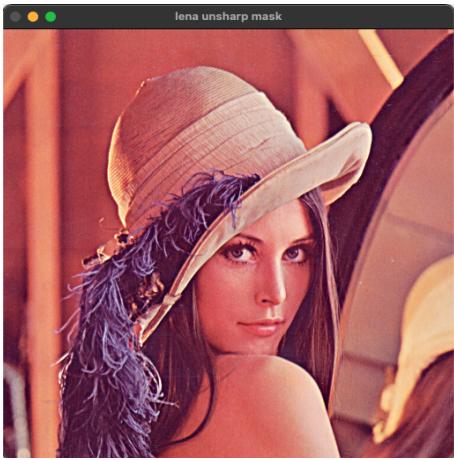
```
unsharped = cv2.addWeighted(img, 1.5, img_gaussian, -0.5, 0)
```

Mostriamo l'immagine risultante:

```
see(unsharped, 'lena unsharp mask', 20000)
```

'addWeighted' ci permette di sommare o sottrarre insieme due immagini dando più peso ad una immagine o all'altra.

In questo caso diamo più peso all'immagine originale con il parametro utilizzando un peso del 150 per cento sull'immagine risultante e un peso del -50 per cento sull'immagine di blur, per migliorare la nitidezza e il contrasto dell'immagine finale.



C'è un altro modo per aumentare la nitidezza dell'immagine, tramite lo sharpening kernel.

Questo filtro funziona aumentando la differenza di intensità tra i pixel adiacenti dell'immagine.

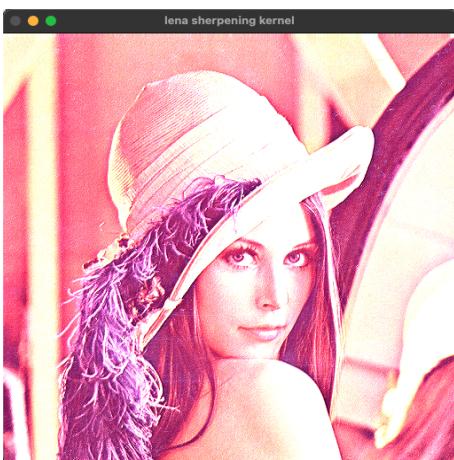
Il kernel di sharpening è composto da un valore negativo nel centro e valori positivi attorno ad esso. Questo kernel viene applicato all'immagine originale attraverso la convoluzione, generando una nuova immagine con maggiori dettagli e nitidezza.

In pratica, l'applicazione di uno sharpening kernel porta ad un aumento del contrasto locale, evidenziando i bordi e gli elementi dell'immagine. È importante notare che l'utilizzo di questo tipo di filtro deve essere fatto con cautela, in quanto può anche evidenziare eventuali rumori presenti nell'immagine.

```
my_sharpening_kernel = np.array([ [0,-1,1], [-1,5,-1], [0,-1,0] ])  
  
filtered_img_sharpening_kernel = cv2.filter2D(img, -1, my_sharpening_kernel)
```

Mostriamo l'immagine risultante:

```
see(filtered_img_sharpening_kernel, 'lena sharpening kernel', 20000)
```



3.11.3 Effetto countour detection - estrazione bordi.

Vediamo alcuni filtri che permettono di mostrare i contorni, i bordi dell'immagine. (countour detection).

Ci sono 2 modi per estrarre i contorni, entrambi utilizzano kernel:

- Sobel operator: calcola la prima derivata dell'immagine perchè se pensiamo all'immagine come una funzione matematica, in questo punto della funzione matematica noi abbiamo un cambio della funzione perchè passiamo da un'area chiara a un'area scura, quindi con la derivata vediamo il cambio di funzione.

Quando il pixel cambia, passa da un area chiara a un'area scura , se calcoliamo la derivata di questo punto c'è un picco e abbiamo un cambio della funzione.

L'operatore sobel richiede come obbligatorio che l'immagine di input sia un'immagine in scala di grigi. Perché per trovare i bordi non abbiamo bisogno di tre canali e nell'immagine in scala di grigi gli angoli sono semplici da trovare.

Innanzitutto carichiamo l'immagine in scala di grigi:

```
img_gray_scale = cv2.imread('data/beach.png', 0)
```

Dobbiamo calcolare le derivate in entrambe le direzioni:

```
# computes the derivate
der_x = cv2.Sobel(img_gray_scale, -1, 1, 0) # x to 1, y to 0.
# dobbiamo fare lo stesso con l'asse y:
der_y = cv2.Sobel(img_gray_scale, -1, 0, 1) # x to 0, y to 1.
```

-1 specifica il numero di canali che vogliamo in output: con -1 è dato da quello in input.

Se voglio calcolare le derivate nella direzione x, devo impostare x=1 e y=0 e viceversa.

Sto facendo la derivata in 2 direzioni differenti: se mostro la der_x mi mostra solo la parte sopra verso dx, se mostro der_y mi mostra la parte verso il basso.

Dobbiamo assumere che le derivate sono positive, ma non ne siamo sicuri.

Se abbiamo derivate negative c'è un problema dato che i pixel nelle immagini hanno solo valori positivi. (0 to 255 in RGB).

Possiamo quindi utilizzare il valore assoluto:

```
abs_x = cv2.convertScaleAbs(der_x)
# questa funzione non e' una funzione che calcola semplicemente il valore assoluto, ma lo
# calcola per le immagini.
abs_y = cv2.convertScaleAbs(der_y)
```

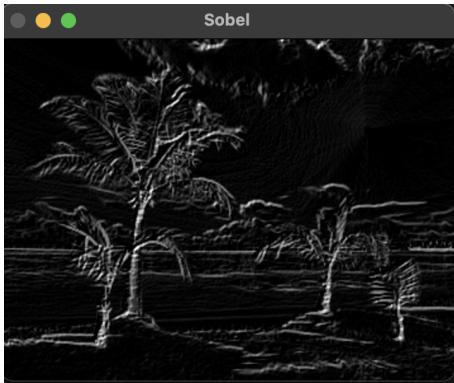
queste due righe di codice non sono necessarie, ma per assicurarsi che non ci siano errori le scriviamo.

Possiamo utilizzare la stessa funzione di prima: addWeighted, ma in questo caso diamo importanza alle immagini alla stessa maniera.

```
der_total = cv2.addWeighted(abs_x, 0.5, abs_y, 0.5, 0)
```

Mostriamo l'immagine risultante:

```
see(der_total, 'Sobel', 20000)
```



- Laplacian operator: questo è un po' complesso perché dobbiamo fare di più rispetto a Sobel poichè dobbiamo calcolare due derivate per entrambi gli assi e poi sommarle insieme.

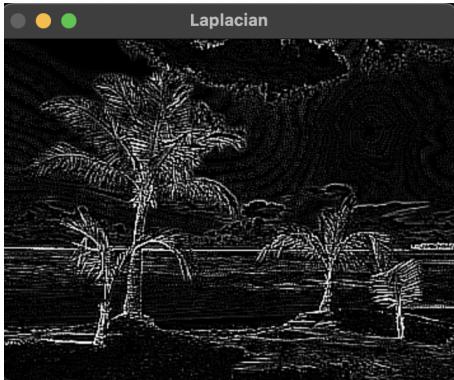
Quindi vengono sommate le derivate seconde calcolate su entrambi gli assi (x,y).

Grazie a openCV abbiamo una funzione per calcolare le derivate per entrambi gli assi:

```
der = cv2.Laplacian(img, -1, ksize=3) # ksize= 3 means 3x3
# converte i pixel in un numero tra 0 e 255.
abs_der = cv2.convertScaleAbs(der)
```

Mostriamo l'immagine risultante:

```
see(abs_der, 'Laplacian', 20000)
```



3.11.4 Blend Images.

Possiamo sovrapporre un'immagine sopra l'altra e vedere quale vogliamo vedere di più.

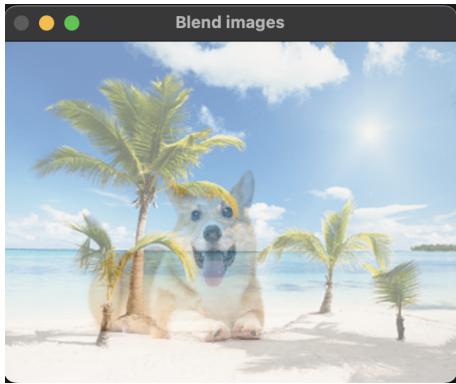
Dobbiamo avere due immagini con le stesse dimensioni quindi possiamo ridimensioniamo la seconda immagine che abbiamo in input:

```
img = cv2.imread('data/beach.png')
img2 = cv2.imread('data/dog.jpg') # serve anche cambiare formato file

resized = cv2.resize(img2, (img.shape[1], img.shape[0]))
added = cv2.addWeighted(img, 0.5, resized, 0.5, 0)
```

Mostriamo l'immagine risultante:

```
see(abs_der, 'Laplacian', 20000)
```



Le immagini sono state sovrapposte con il metodo addWeighted e abbiamo dato importanza ad entrambe le immagini.

3.12 Operazioni aritmetiche immagini.

Vediamo un po' di aritmetica applicata immagini: vediamo cosa può succedere se si sommano, sottraggono pixel nelle immagini, se vengono effettuate operazioni binarie, ...

Se abbiamo un array con numpy e facciamo un'operazione come '0 - 2' il programma andrà in overflow. Invece questo non è vero per le immagini opencv.

OpenCV è costruito per lavorare con le immagini, quindi se eseguiamo un'operazione in openCV il risultato di tale operazione sarà sempre nell'intervallo 0-255.

Ciò significa che se abbiamo un pixel bianco 255 e sommiamo 10 rimane 255, se abbiamo un pixel nero 0 e togliamo da quel pixel rimane 0.

In numpy abbiamo un problema perché c'è overflow.

```
x = np.uint8([250]) # intero a 8 bit tra 0 e 255.
y = np.uint8([50])
```

```
# se sommiamo questi numeri con openCV:  
result_openCV = cv2.add(x,y) # [[255]]  
result_numpy = x + y # [44] overflow: quando raggiunge il massimo valore riparte da 0
```

In result_numpy il risultato viene calcolato considerando solo i primi 8 bit dei numeri, quindi si ha:

11111010	(250 in binario)
+ 00110010	(50 in binario)

00101100	(44 in decimale)

Il risultato viene troncato ai primi 8 bit, che corrispondono al valore 44. Questo è un esempio di overflow, in cui l'aggiunta di due numeri porta a un valore superiore al valore massimo rappresentabile dal tipo di dati utilizzato.

Vediamo cosa accade se, presa un immagine, aggiungiamo dei pixel ad ognuno di quelli dell'immagine. Per fare questo possiamo utilizzare 2 metodi:

- creare una nuova immagine con stesso numero di pixel (righe e colonne) e canali di quella precedente e sommare i valori di un altro array (M):

```
M = np.full(img.shape, 50, dtype=np.uint8) # img.shape ha le stesse dimensioni  
dell'immagine img  
added = cv2.add(img, M)  
  
# se aggiungiamo pixel all'immagine, quest'ultima diventa piu' chiara;  
# se sottraiamo pixel all'immagine, quest'ultima diventa piu' scura.
```

La matrice viene creata con le stesse dimensioni dell'immagine originale, tutti gli elementi sono costituiti dal valore '50' intero a 8 bit.

Grazie al metodo 'add' viene aggiunto ad ogni pixel dell'immagine il valore '50' della matrice gestendo l'overflow.

Questo tipo di operazione può essere utilizzato per aumentare la luminosità di un'immagine o per applicare un filtro di tipo "brightening".

- usare una scala rotation, stesso effetto di prima:

```
M2 = np.full((1,3), 50, dtype=np.float64)  
added = cv2.add(img, M2)
```

Grazie alla funzione di addizione viene eseguita la somma tra l'immagine 'img' e un array NumPy 'M2' di dimensione (1,3) di tipo float.64. In questo array tutti gli elementi sono inizializzati al valore '50.0'. Grazie al metodo 'cv2.add' vengono gestiti i casi in cui ci può essere overflow.

In tutti e due i casi la somma viene eseguita separatamente sui tre canali (BGR) dell'immagine.

3.13 BITWISE Operation.

Le operazioni bitwise su immagini in OpenCV sono operazioni logiche che vengono applicate bit per bit ai pixel dell'immagine. Queste operazioni sono utili per eseguire operazioni come l'AND, l'OR e lo XOR tra due immagini, o per eseguire operazioni di mascheramento sull'immagine.

In OpenCV, le operazioni bitwise possono essere eseguite utilizzando le funzioni cv2.bitwise_and, cv2.bitwise_or, cv2.bitwise_xor e cv2.bitwise_not.

Queste funzioni accettano due immagini come input e restituiscono un'immagine risultante che rappresenta l'operazione bitwise applicata ai pixel corrispondenti delle due immagini di input.

Ad esempio, se vogliamo eseguire un'operazione di mascheramento sull'immagine originale utilizzando un'immagine di maschera (immagine in bianco e nero dove i pixel neri rappresentano le aree della maschera in cui l'immagine originale deve essere mascherata), possiamo utilizzare la funzione cv2.bitwise come segue:

```
img2 = np.zeros(img.shape, dtype=np.uint8)
# creo un immagine con stesso numero di colonne, di righe dell'immagine della spiaggia e la
utilizzo come maschera per l'immagine originale

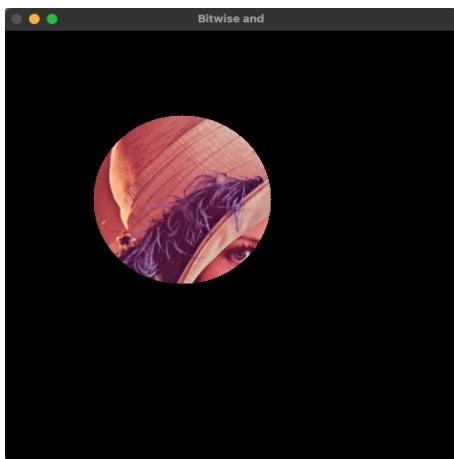
# inserisco un cerchio nell'immagine nera
masked = cv2.circle(img2, (200, 200), 100, (255,255,255), -1) # -1 : cerchio pieno, se
incrementiamo 200 andiamo verso dx, se incrementiamo 100 andiamo giu'
```

L'operazione di mascheramento in computer vision e image processing consiste nell'applicare una maschera (che può essere un'immagine binaria o a scala di grigi) su un'altra immagine per selezionare solo una specifica regione di interesse e applicare un'operazione solo a quella regione.

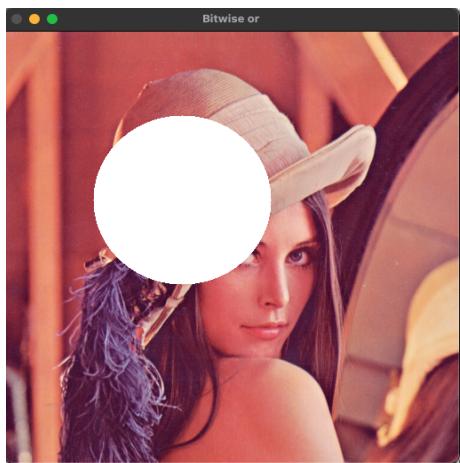
Il processo di mascheramento è utile per isolare parti di un'immagine che sono rilevanti per una determinata analisi o operazione, ad esempio per rimuovere il rumore, migliorare la nitidezza dell'immagine, rilevare oggetti o contorni, o estrarre caratteristiche specifiche.

Adesso utilizziamo gli operatori logici:

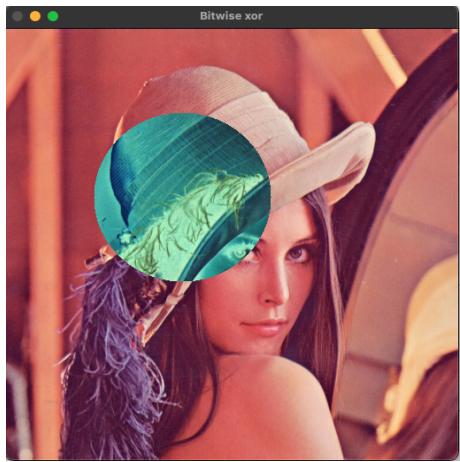
```
and_img = cv2.bitwise_and(img, masked)
# viene mostrata l'immagine nera con un cerchio in cui si vede lena.
```



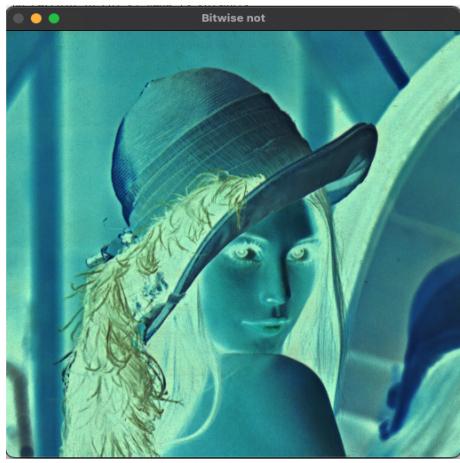
```
or_img = cv2.bitwise_or(img, masked)
# viene mostrata l'immagine di lena con un cerchio bianco all'interno.
```



```
xor_img = cv2.bitwise_xor(img, masked)
# viene mostrata l'immagine di lena con un cerchio all'interno con colori all'infrarosso.
```



```
not_img = cv2.bitwise_not(img, masked)
# inverte i pixel
```



3.14 Effetto Sketch

L'effetto sketch o 'disegno a matita' di un'immagine può essere ottenuto tramite l'elaborazione dell'immagine con una combinazione di filtri e tecniche di elaborazione dell'immagine.

Cosa si fa:

- Carichiamo l'immagine da cartoonizzare:

```
img = cv2.imread('data/lena.jpg')
```

- Converto l'immagine in grayscale perché più tardi avrò bisogno del colore altrimenti avremo un'immagine cartoonizzata ma solo ridimensionata:

```
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

- Possiamo applicare il filtro median blur per eliminare il rumore se presente nell'immagine:

```
img_gray = cv2.medianBlur(img_gray, 5) # piu' il parametro dopo l'img_gray e' piccolo,  
piu' il filtro dell'immagine e' aggressivo.
```

Per ottenere l'effetto cartoonize dobbiamo calcolare due cose: il colore e per ottenere un effetto di colore cartone animato possiamo usare un filtro blur molto aggressivo perché un filtro sfocato fa la media, quindi se facciamo la media perderemo tutti i bordi e avremo tutti i colori misti, che è quello che vogliamo.

- Come prima cosa estriamo i bordi con Laplacian filter:

```
edges = cv2.Laplacian(img_gray, cv2.CV_8U, ksize=5) # CV_8U:indica 8 bit senza segno
```

Abbiamo bisogno che i bordi siano forti perché vogliamo il fumetto come effetto, quindi vogliamo copiare ad esempio lo stile di una matita o di una penna,...

Se abbiamo un CV_8U, i bordi contengono valori compresi tra 0 e 255.

Possiamo eseguire la thresholding per ottenere dall'immagine della scala dell'indirizzo un'immagine binaria.

Il Thresholding è una tecnica di elaborazione delle immagini utilizzata per separare gli oggetti di interesse dallo sfondo, rendendo più facile l'analisi dell'immagine.

In pratica, il thresholding consiste nel convertire un'immagine in bianco e nero, in cui i pixel vengono classificati in base al loro valore di intensità. Si stabilisce una soglia di intensità, chiamata "threshold", e tutti i pixel con un valore di intensità superiore a questa soglia vengono classificati come oggetti di interesse (generalmente bianchi), mentre i pixel con un valore inferiore vengono classificati come sfondo (generalmente neri).

Ad esempio, quindi impostiamo un valore di 100, ogni pixel che ha un valore superiore a 100 verrà impostato su bianco e ogni pixel che ha un valore inferiore a 100 verrà impostato su nero.

```
_, thresholded = cv2.threshold(edges, 70, 255, cv2.THRESH_BINARY_INV) # la funzione ritorna  
due valori, ma a noi interessa solo il secondo valore
```

Parametri della funzione thresholded:

1. l'immagine alla quale vogliamo applicare la funzione, in questo caso i bordi estratti.

2. il valore di thresholded (maxval): il valore massimo
3. 255 perchè voglio bordi marcati molto.
4. come vuoi eseguire il thresholded: Il metodo threshold restituisce una nuova immagine come una maschera binaria: tutto nero con i bordi in bianco ma siccome vogliamo emulare lo schizzo di una matita o di una penna, abbiamo bisogno di una maschera invertita: quindi invece di avere uno sfondo nero e bordi bianchi noi abbiamo uno sfondo bianco e bordi neri.

Ogni pixel che ha un valore da 70 in su verrà impostato su 255.

Ora abbiamo lo sketch, vediamo come fare per i colori: Il filtro bilaterale esegue una sfocatura gaussiana sull'immagine, ma ha una proprietà interessante, invece di sfocare l'intera immagine, il filtro bilaterale è in grado di preservare i bordi all'interno dell'immagine.

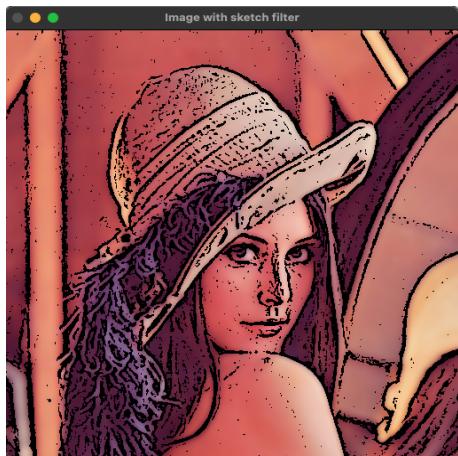
```
# get the colors with the bilateral filter:
color_img = cv2.bilateralFilter(img, 10, 250, 250) # 10: size kernels, 250: sigma x and y for
# the gaussian distribution.
```

Ora che abbiamo angoli e colori:

```
# merge color and edges:

# thresholded is a one channel image
skt = cv2.cvtColor(thresholded, cv2.COLOR_GRAY2BGR) # convertiamo in 3 canali
sketch_img = cv2.bitwise_and(color_img, skt)

cv2.imshow('Image with sketch filter', sketch_img)
```



3.14.1 Effetto sketch real time con videocamera.

Creazione dell'effetto sketch in diretta grazie alla videocamera:

```
import cv2
import numpy as np

# Usiamo il filtro in tempo reale:

cap = cv2.VideoCapture(0) # il numero la videocamera a cui vogliamo accedere

while True:

    # we read a frame from the video stream
    img = cap.read()[1]

    # convert image to grayscale
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # smooth the image to clean it up a bit
    img_gray = cv2.medianBlur(img_gray,5)

    # use the Laplacian Operator to extract edges
    edges = cv2.Laplacian(img_gray, cv2.CV_8U, ksize=5)

    # threshold the edges
    ret, thresholded = cv2.threshold(edges, 70, 255, cv2.THRESH_BINARY_INV)

    # use bilateralFilter with big numbers to get the colors
    color_img = cv2.bilateralFilter(img, 10, 250, 250)

    # add together color and sketch
    skt = cv2.cvtColor(thresholded, cv2.COLOR_GRAY2BGR)

    sketch_img = cv2.bitwise_and(color_img, skt)
    cv2.imshow("Edges", sketch_img)
    k = cv2.waitKey(4) # 4 perche' intendiamo 24-25 frame per secondo

    if k == ord('q'):
        break
```

Vediamo come possiamo estrarre funzionalità di basso livello, vedremo che gli istogrammi vengono usati per fare molte cose e vedremo come estrarre automaticamente delle caratteristiche come dei punti di un'immagine o video.

Vedremo un esercizio in cui mettiamo insieme due immagini con delle parti in comune e creiamo un'immagine panorama.

Vedremo anche un programma di rilevamento degli oggetti: se mettiamo una bottiglia davanti alla telecamera ci dirà che è una bottiglia. Ovviamente questo non è preciso come la machine e deep learning. Questo sarà solo un algoritmo di classificazione degli oggetti, non di rilevamento degli oggetti.

La differenza tra questi 2 è data dal fatto che :

- classificazione significa assegnare una classe alla cosa che sto dando in input al mio programma: quindi dico questa è una bottiglia, questo è un libro, ...
- il rilevamento degli oggetti ha unito la classificazione e la localizzazione delle immagini.

Se metto la bottiglia davanti alla telecamera il mio algoritmo dirà che è una bottiglia e scatterà un riquadro

di delimitazione attorno all'oggetto, quindi mi dirà l'oggetto che sta vedendo e mi dirà anche dove si trova l'oggetto è dentro la scena. C'è una cosa molto simile al rilevamento degli oggetti che è image segmentation dove invece di avere un riquadro di delimitazione come un rettangolo c'è una sagoma dell'oggetto.

3.15 Histogram.

Gli istogrammi sono utilizzati per rappresentare la distribuzione dei dati all'interno di un'immagine, di un testo o di qualsiasi altra fonte di dati.

In particolare, negli applicativi di visione artificiale, gli istogrammi sono spesso utilizzati per analizzare l'immagine e comprendere meglio le sue caratteristiche.

Gli istogrammi sono costituiti da un asse x che rappresenta il valore che vogliamo rappresentare (per esempio, il valore dei pixel in un'immagine) e un asse y che rappresenta il numero di elementi che appartengono a ciascuno di questi valori sull'asse x. In questo modo, gli istogrammi ci permettono di analizzare la distribuzione dei dati e di estrarre informazioni utili per elaborare l'immagine o il testo in modo più efficace.

In questo caso il valore x è dato dal valore del pixel del singolo canale che va da 0 a 255 e l'asse y abbiamo il numero totale di pixel.

Le cose che sono poste sull'asse x sono chiamate bins: quante frequenze vogliamo controllare.

Per esempio in un'immagine in grayscale vogliamo contare le occorrenze per ogni livello di grigio, dobbiamo avere un bins per ogni livello di grigio.

Possiamo dividere in 3 macro bins: (0-60, 61-120, 120-255): vari livello di grigio.

Il range è l'intervallo che vogliamo considerare, nell'immagine di scala di grigi il range è 0-255.

Facciamo un esempio:

Carichiamo un'immagine in grayscale (1 canale) e creiamo l'istogramma:

```
img = cv2.imread('dog.jpg', 0)
hist = cv2.calcHist([img], [0], None, [256], [0,255])
```

La funzione calcHist è una funzione in OpenCV utilizzata per calcolare un istogramma di una o più immagini. I parametri sono:

- images: un elenco di immagini di input per cui si desidera calcolare l'istogramma.
- channels: l'indice del canale dell'immagine per cui si desidera calcolare l'istogramma. Se l'immagine è in scala di grigi, il valore di channels dovrebbe essere [0]. Se l'immagine è un'immagine a colori, i canali di colore sono [0] per il canale blu, [1] per il canale verde e [2] per il canale rosso.
- mask: una maschera opzionale per limitare la porzione dell'immagine di cui vogliamo completare l'istogramma, perché si potrebbe avere l'istogramma di una certa parte dell'immagine.

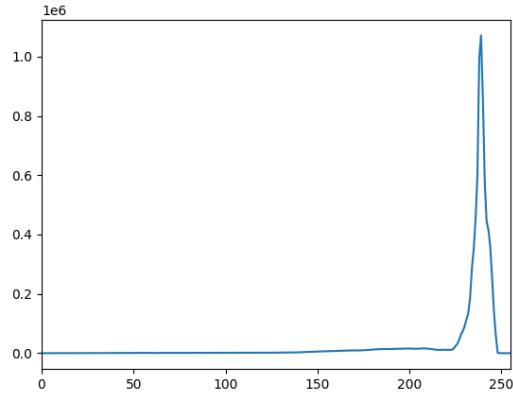
Se vogliamo l'istogramma dell'immagine completa utilizziamo None.

- histSize: il numero di bin dell'istogramma.
Supponiamo di voler calcolare la frequenza di ogni colore in grayscale image (cioè quanti pixel hanno il valore 0, quanti hanno il valore 1, ...)
Quindi in questo caso abbiamo 256 valori per l'immagine grayscale.
- ranges: il range dei valori di pixel. In genere, per un'immagine a colori, il valore di ranges è [0,256,0,256,0,256] perché ci sono tre canali di colore e ogni canale ha valori che vanno da 0 a 255.
In questo caso c'è [0,255] perché abbiamo solo un canale (grayscale image)

Per tracciare l'istogramma e mostrarlo possiamo utilizzare openCV oppure matplotlib:

```
plt.plot(hist)
plt.xlim([0,255]) # limiti asse x
plt.show()
```

Quindi avremo la distribuzione dei pixel della mia immagine: Nella parte finale dell'istogramma (220-250) ho dei picchi alti perchè ci sono molti pixel bianchi.



Ora vediamo l'istogramma di un'immagine a colori, con 3 canali (BGR):

- Innanzitutto carichiamo l'immagine e separiamo i 3 canali:

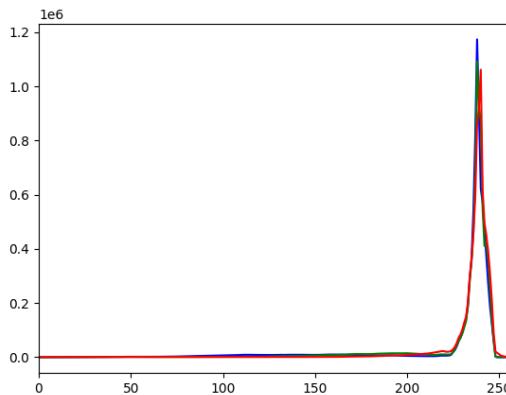
```
img_color = cv2.imread('dog.jpg')
color = ('b', 'g', 'r')
```

- Per ogni canale traccio l'istogramma associato:

```
for i,col in enumerate(color):
    hist2 = cv2.calcHist([img_color], [i], None, [256], [0,255])
    plt.plot(hist2, color = col) # setto il colore della linea come il colore del canale
```

- Mostro i vari istogrammi:

```
plt.xlim([0,255]) # comando in matplotlib utilizzato per impostare i limiti dell'asse x
                   di un grafico.
plt.show() # Viene mostrato l'istogramma contenente tutti i colori dei canali
```



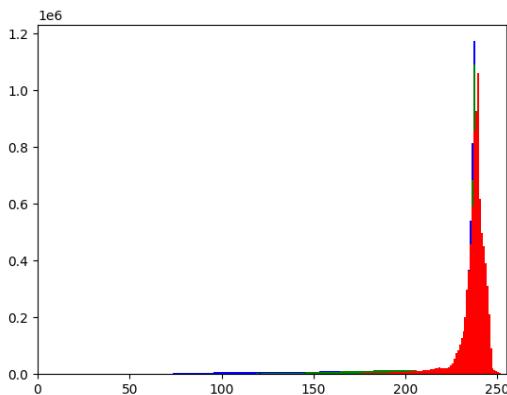
`plt.plot()` viene utilizzata per tracciare una linea che collega i punti in un grafico. Viene spesso utilizzata per visualizzare le tendenze dei dati, come ad esempio l'andamento di una serie temporale. La funzione accetta due elenchi di valori come input, rappresentanti i valori sull'asse x e quelli sull'asse y, e può anche accettare altri parametri opzionali come il colore, lo stile di linea e le etichette degli assi.

`plt.hist()`, d'altra parte, viene utilizzata per visualizzare l'istogramma di una distribuzione di dati. L'input della funzione è un elenco di valori, rappresentanti i dati che si desidera visualizzare. La funzione suddivide questi dati in bin (gruppi di valori), conta il numero di valori in ogni bin e crea un grafico a barre in cui l'asse x rappresenta i bin e l'asse y rappresenta il conteggio dei valori in ogni bin.

Vediamo l'utilizzo della funzione `plt.hist()`:

```
for i,col in enumerate(color):
    channel = img_color[:, :, i]
    plt.hist(channel.ravel(), 256, [0,255], color=col)
    # .ravel() converte il canale in un mono dimensional array, gli altri parametri
    # sono uguali a quelli dell'altra funzione
plt.xlim([0,255]) # limiti asse x
plt.show()
```

Con la funzione `hist` viene colorata l'area della curva.



Esistono vari metodi per trovare oggetti all'interno delle immagini utilizzando gli istogrammi. Uno dei metodi più comuni è l'istogramma di gradiente orientato (HOG, Histogram of Oriented Gradients), che estrae le caratteristiche locali dell'immagine in base alla direzione dei gradienti. Una volta estratte le caratteristiche, è possibile addestrare un classificatore per riconoscere gli oggetti di interesse.

Per quanto riguarda la modifica dell'istogramma, esistono diverse tecniche di equalizzazione dell'istogramma, come la equalizzazione dell'istogramma globale o locale, che possono essere utilizzate per migliorare il contrasto dell'immagine. Tuttavia, la modifica dell'istogramma può avere effetti imprevisti sulle informazioni dell'immagine e può influire negativamente sulla rilevazione degli oggetti. Pertanto, è importante essere cauti quando si applicano queste tecniche e valutare attentamente l'impatto sulla rilevazione degli oggetti.

In generale, è importante selezionare il metodo di rilevamento degli oggetti e la tecnica di modifica dell'istogramma in base alle specifiche esigenze dell'applicazione e valutare attentamente gli effetti di tali modifiche sull'immagine e sulla rilevazione degli oggetti.

Vediamo come equalizzare l'istogramma:

```

img_ = cv2.imread('data/dog.jpg')
gray_img = cv2.cvtColor(img_, cv2.COLOR_BGR2GRAY)
equalized = cv2.equalizeHist(gray_img)

see(gray_img, 'Gray_img', 10000)
see(equalized, 'Equalized', 10000)

```

L'immagine viene convertita in scala di grigi utilizzando la funzione ‘cv2.cvtColor()‘ con il flag.

In seguito, viene applicata la tecnica di equalizzazione dell'istogramma utilizzando la funzione ‘cv2.equalizeHist()‘. Questa tecnica viene applicata all'immagine in scala di grigi ‘gray_img‘ per migliorare il contrasto dell'immagine e distribuire in modo uniforme i livelli di grigio presenti nell'immagine. Il risultato dell'equalizzazione viene assegnato alla variabile ‘equalized‘.

L'immagine equalizzata sembra avere una gamma di colori leggermente più ampia rispetto all'immagine originale in scala di grigi.

Tuttavia, è importante notare che la modifica dell'istogramma tramite l'equalizzazione può alterare le informazioni presenti nell'immagine, quindi è necessario valutare attentamente se tale modifica sia appropriata per le specifiche esigenze dell'applicazione.



(a) Iстограмма dell'immagine originale.



(b) Immagine equalizzata.

L'immagine equalized è definita da questi colori perchè l'istogramma cerca di distribuire i pixel nella maniera migliore.

Con equalized otteniamo qualche informazione in più.

In questo caso l'immagine del cane equalizzata non mostra un miglioramento dell'immagine.

Prendiamo invece un'immagine che ci permette di vedere che con equalized abbiamo un miglioramento dell'immagine, ad esempio come l'immagine della modella lena).

```

img_2 = cv2.imread('lena.png')
channels = cv2.split(img_2)
eq_channels = []

for ch in channels:
    eq_channels.append(cv2.equalizeHist(ch))

eq_img = cv2.merge(eq_channels)
see(eq_img, 'Eq_img', 2000)

```



Possiamo fare delle modifiche all'immagine attraverso l'istogramma:

Se volessi mantenere un po' di rosso sulla mia immagine perché era l'effetto che volevo sull'immagine o se volessi normalizzare per equalizzare il mio istogramma ma senza ottenere risultati troppo aggressivi, ci sono diversi modi per gestirlo questi casi:

Possiamo drasticamente modificare i colori dell'immagine mentre a volte abbiamo solo bisogno di cambiare ad esempio il contrasto o la luminosità.

Ad esempio se decidiamo di prendere i dati in questo formato e vogliamo rimodellare i dati è molto semplice da fare utilizzando l'istogramma: prendiamo i dati e rimodelliamo questi con la distribuzione gaussiana. È molto più semplice da fare con l'istogramma rispetto alla modifica dell'immagine.

E se volessimo cambiare solo la saturazione del colore o il cambio di luminosità e così via fino ad ora abbiamo visto un solo formato colore dell'immagine: RGB (3 canali: uno per il colore), ma ci sono alcune rappresentazioni di immagini che ci permettono di gestire meglio queste situazioni (toccare solo il contrasto, la saturazione dell'immagine, ...)

Un esempio di questa rappresentazione si chiama HSV.

Queste rappresentazioni sono chiamate 'color spaces': un modo per rappresentare l'immagine RGB, HSV sono color spaces.

HSV è differente da RGB perchè in HSV abbiamo sempre 3 canali che ci permettono di lavorare con dati diversi. Qual è lo scopo di questi canali?

- Il canale V permette di rendere il colore più luminoso o più scuro.
- Il canale V permette di rendere il colore più luminoso o più scuro.
- Il canale H permette di cambiare il colore.

La saturazione è quanto è saturato il colore. (l'intensità del colore)

Mentre il valore può essere inteso come luminosità del colore.

Vediamo come utilizzare HSV:

```
img_hsv = cv2.imread('data/lena.png')
# trasformiamo il 'color spaces' dell'immagine: la trasformiamo da RGB in HSV.
hsv_img = cv2.cvtColor(img_hsv, cv2.COLOR_BGR2HSV)
```

```

# splittiamo l'immagine in 3 canali:
h,s,v = cv2.split(hsv_img)

# Modifichiamo il valore v: luminosità
eq_v = cv2.equalizeHist(v)

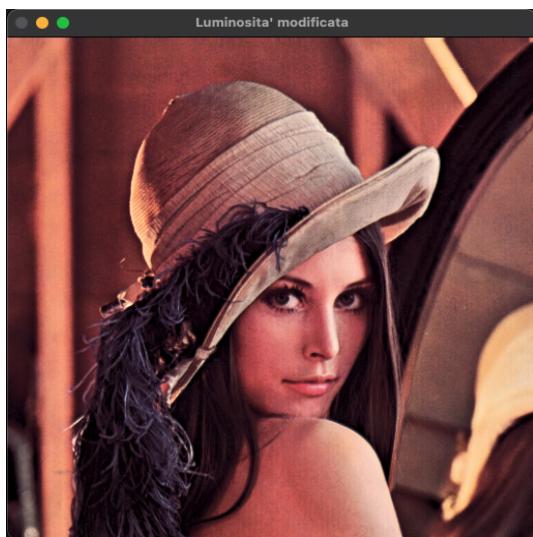
# In questo modo abbiamo unito i canali dell'immagine con quello modificato.
eq_img = cv2.merge([h, s, eq_v])

# Ora per mostrare l'immagine dobbiamo riconvertirla in RGB o BGR:
eq_img = cv2.cvtColor(eq_img, cv2.COLOR_HSV2BGR)
cv2.imshow(eq_img, "Luminosità modificata", 8000)

```

In questo caso l'immagine è migliore rispetto alla precedente in cui è stato rimosso il rosso.

In questo caso abbiamo mantenuto l'idea del fotografo del colore rosso chiaro, ma nell'immagine è stato aumentato il contrasto e appaiono più dettagli.



A volte quando sistemiamo l'immagine non abbiamo il risultato che ci aspettavamo.

Fa l'esempio dell'immagine in bianco e nero con la statua e la libreria dietro: se noi eseguiamo cv2.equalizeHist(img) non ci sarà un miglioramento dell'immagine.



Questo perchè magari nell'immagine presa in considerazione, sono stati guadagnati molti dettagli dallo sfondo della libreria, che è migliorata e resa più dettagliata.

Ma la statua ha perso tutte le informazioni perchè 'histogram equalization techniques' considera l'immagine totale, vengono presi in considerazione tutti i pixel e dopo di che vengono ridistribuiti.

Quindi c'è un miglioramento per la tecnica di equalizzazione dell'istogramma chiamata equalizzazione adattiva: invece di considerare l'immagine totale divido l'immagine in piccole regioni come blocchi.

Dopo vado ad applicare l'equalization per ogni regione.

Ad esempio nell'immagine presa in considerazione nella parte della statua ci sono pixel chiari, mentre sullo sfondo ci sono più pixel scuri.

C'è un metodo in openCV per fare questo: CLAHE: contrast limited adaptive histogram equalization.

Il metodo CLAHE, ovvero contrast limited adaptive histogram equalization, è una tecnica di equalizzazione dell'istogramma adattiva che è stata sviluppata per risolvere il problema dell'equalizzazione dell'istogramma su immagini con contrasto locale molto variabile. Invece di applicare l'equalizzazione dell'istogramma all'intera immagine, CLAHE divide l'immagine in piccoli blocchi e applica l'equalizzazione dell'istogramma a ciascun blocco separatamente.

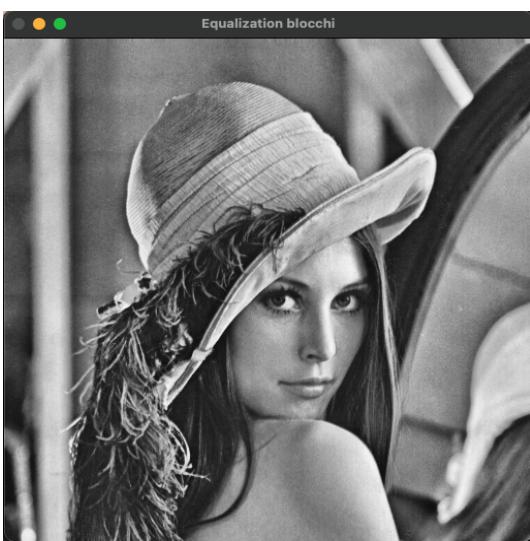
Tuttavia, per evitare di esagerare il contrasto locale, CLAHE limita la quantità di equalizzazione che può essere applicata a ciascun blocco. Ciò viene fatto riducendo il massimo valore di contrasto per ogni blocco. In questo modo, i dettagli nelle regioni a basso contrasto vengono migliorati senza esagerare il contrasto nelle regioni ad alto contrasto.

In OpenCV, CLAHE è implementato nella classe cv2.createCLAHE(). Per utilizzare CLAHE, è necessario creare un'istanza di questa classe e specificare i parametri di configurazione, come il clip limit (la quantità di equalizzazione che può essere applicata a ciascun blocco) e le dimensioni dei blocchi.

```
img = cv2.imread('data/lena.png', 0)

# creiamo un'istanza del metodo perchè abbiamo bisogno di alcuni parametri per l'equalization:
clahe = cv2.createCLAHE(2, tileGridSize=(8,8))
# tileGridSize indica la grandezza delle zone (blocchi), se aumentiamo la totalGridSize
# l'immagine
# sarà più contrastata (come HDR effect). Se inseriamo (1,1) prendiamo ogni pixel e la
# ridistribuzione
# non avviene.
# 2 indica il modo in cui distribuisce i valori tra i blocchi

eq_img = clahe.apply(img)
see(eq_img, 'Equalization blocchi', 3000)
```



In questo caso le immagini sono migliorata con l'equalization nelle immagini in bianco e nero (grayscale).

3.16 Feature extraction - Estrazione delle caratteristiche

La feature extraction o estrazione delle caratteristiche è un processo fondamentale nella computer vision e nell'elaborazione delle immagini. Si tratta di identificare e estrarre le informazioni più rilevanti dall'immagine al fine di rappresentarla in modo efficace e utilizzarla per compiti come il riconoscimento di pattern, la classificazione, la segmentazione e la localizzazione degli oggetti.

Ad esempio all'interno di un'immagine, le caratteristiche possono essere l'angolo che si trova all'interno dell'immagine.

Invece in machine learning features sono le cose di cui abbiamo bisogno per addestrare il modello.

Se dobbiamo addestrare un modello per distinguere diversi tipi di fiori una caratteristica può essere la lunghezza dei fiori, la lunghezza del filo, ...

A seconda della struttura di estrazione delle caratteristiche noi distinguiamo 2 cose: Il vettore che descrive i key point è il descrittore.

1. the feature position, i key point (punti da estrarre) che rappresentano la posizione di una caratteristica all'interno di un'immagine. (perchè in alcune coordinate all'interno dell'immagine c'è qualcosa di importante).
2. il descrittore è un vettore numerico che descrive le proprietà di quella caratteristica (key point), come la forma, il colore o l'orientamento.

In sostanza, il descrittore è una rappresentazione compatta delle informazioni più rilevanti di una caratteristica, che può essere utilizzata per confrontare e riconoscere diverse immagini. Esistono diverse tecniche per l'estrazione dei key point e la generazione dei descrittori, come ad esempio SIFT, SURF, ORB e molti altri.

Quando si estraggono i punti questi possono essere utilizzati per qualsiasi cosa:

- machine learning algorithm.
- panorama images.
- work and unwork images.
- trasformazione omografica (come esercizio in classe in cui si clicca sui punti e cerchio rosso e la seconda immagine viene messa nei punti selezionati)
- capire quanto un'immagine è ruotata, ...
- metriche omografiche.
- the representation of the feature.

È possibile utilizzare le caratteristiche estratte dalle immagini presenti nel database per cercare se un'immagine specifica è già presente. Questo è possibile confrontando i descrittori o le caratteristiche estratte dall'immagine cercata con quelle presenti nel database. Se le caratteristiche dell'immagine cercata corrispondono a quelle di una o più immagini presenti nel database, allora si può concludere che l'immagine cercata è presente nel database. Questa tecnica di confronto di caratteristiche viene utilizzata in molte applicazioni di ricerca di immagini, come ad esempio la ricerca di immagini simili o la ricerca di corrispondenze in un database di impronte digitali.

Il feature extractors lavora con immagini in scala di grigio perché non è importante colore dell'immagine per estrarre delle caratteristiche, stessa caratteristica dell'estrattore dei bordi.

Quindi cairichiamo l'immagine e convertiamola in scala di grigi:

```
img_lena = cv2.imread('data/lena.png')
gray_img_lena = cv2.cvtColor(img_lena, cv2.COLOR_BGR2GRAY)
```

Uno dei primi estrattori è Harris Corner detector, il suo scopo è quello di estrarre angoli dalle immagini.

```
dst = cv2.cornerHarris(gray_img_lena, 2, 23, 0.04)
```

cv2.cornerHarris è una funzione di OpenCV utilizzata per individuare i corner (angoli) all'interno di un'immagine. Questa funzione utilizza il metodo di Harris corner detection, che si basa sul calcolo di una metrica di cornerness in ogni pixel dell'immagine.

La funzione cv2.cornerHarris prende come input

- un'immagine in scala di grigi;
- il parametro blockSize, che indica la dimensione del vicinato (quadrato) considerato per il calcolo della metrica di cornerness (quanti pixel vogliamo considerare attorno ad un certo pixel per rilevare un angolo. In questo caso 2 pixel su ciascun asse.)
- il parametro k, che indica una costante empirica utilizzata nella formula di calcolo della metrica di cornerness.
- un parametro che rappresenta il valore della costante nel calcolo della funzione di cornerness di Harris, impostato a 0.04 in questo caso.

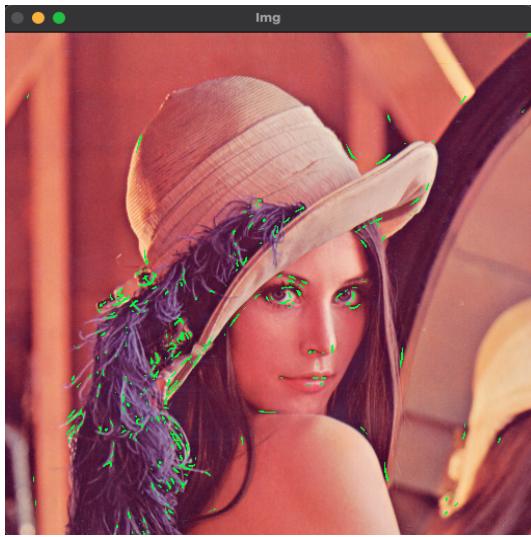
La funzione restituisce un'immagine con i valori della metrica di cornerness calcolati per ogni pixel dell'immagine originale. Questi valori possono essere utilizzati per individuare i corner dell'immagine utilizzando tecniche di thresholding o clustering.

cv2.cornerHarris può essere utilizzata in diversi contesti, ad esempio nella localizzazione di oggetti in immagini, nella navigazione di robot e droni, nella stabilizzazione di immagini e nella realtà aumentata.

Ora tracciamo l'angolo sopra l'immagine utilizzando numpy masking (uso di un array di booleani (noto come maschera o "mask" in inglese) per selezionare o modificare i valori di un array NumPy in base a certe condizioni. In pratica, la maschera è un array di booleani con le stesse dimensioni dell'array originale, dove ogni elemento della maschera è True o False a seconda che si desideri includere o escludere l'elemento corrispondente nell'operazione che si sta eseguendo):

In questo caso dst contiene le coordinate degli angoli della mia immagine.

```
# thresholding the corners for getting the better ones.  
# abbiamo alcuni angoli che sono migliori degli altri.  
img_lena[dst > 0.01 * dst.max()] = [0,255,0] # Voglio evidenziarli con il rosso  
# ottenere solo gli angoli vicini al valore massimo ottenuto dalla funzione  
see(img_lena, 'Img', 3000)
```



Harris corner detection può rilevare non solo gli angoli, ma anche altri punti rilevanti dell’immagine. Ciò è dovuto al fatto che la funzione Harris calcola la differenza di intensità tra i pixel circostanti il punto in esame, utilizzando un kernel di dimensioni scelte dall’utente. Pertanto, i punti in cui la differenza di intensità è significativa rispetto alla media dei pixel circostanti sono considerati punti rilevanti.

Tuttavia, Harris corner detection è ancora utilizzato in alcune applicazioni specifiche, come ad esempio il rilevamento degli angoli in immagini mediche o di microscopia, dove gli angoli rappresentano punti di interesse critici.

In generale, ci sono molti altri rilevatori di punti rilevanti, come SIFT, SURF e ORB, che sono più precisi e robusti di Harris, soprattutto in presenza di rumore o di immagini con basso contrasto.

La funzione di cornerness di Harris è un algoritmo che permette di individuare gli angoli presenti in un’immagine. Gli angoli, in questo caso, si riferiscono a quei punti dell’immagine in cui c’è una netta variazione del gradiente, ovvero una brusca variazione di luminosità in almeno due direzioni diverse.

La funzione di cornerness di Harris calcola un valore per ogni pixel dell’immagine, che rappresenta la sua “cornerness” ovvero la sua somiglianza con un angolo.

Questo valore viene calcolato attraverso la valutazione del gradiente dell’immagine in diverse direzioni e l’analisi della risposta del gradiente.

Il gradiente di un’immagine rappresenta l’intensità della variazione della luminosità in diverse direzioni. La funzione di cornerness di Harris utilizza la risposta del gradiente in diverse direzioni per determinare se in quel punto dell’immagine ci sia un angolo.

Il calcolo della funzione di cornerness di Harris prevede l’utilizzo del tensori del gradiente dell’immagine, che è una matrice simmetrica che rappresenta la variazione del gradiente in diverse direzioni. La matrice dei tensori viene analizzata per determinare i punti dell’immagine in cui si verificano le variazioni più significative del gradiente, ovvero dove sono presenti gli angoli.

Se c’è una variazione significativa del gradiente in una sola direzione, significa che c’è un forte cambiamento di luminosità in quella direzione e quindi che potrebbe essere presente un bordo o un’area di transizione tra due regioni dell’immagine.

Se c’è una variazione significativa del gradiente in due o più direzioni, questo indica la presenza di un angolo nell’immagine.

Vediamo con un esempio come Harris non è invarianta alla scala:

Carichiamo di nuovo l'immagine di lena e creiamo una nuova immagine rimpicciolita di lena, facciamo in modo che questa immagine esca dalle dimensioni dell'immagine originale: possiamo utilizzare la matrice di transizione oppure il metodo predefinito:

```
img_lena = cv2.imread('data/lena.png')
resized = cv2.resize(img_lena, None, fx=0.5, fy=0.5)
```

- i parametri fx e fy rappresentano i fattori di scala rispettivamente per l'asse x (larghezza) e y (altezza) dell'immagine.
- il parametro None potrebbe essere sostituito dalla nuova dimensione in pixels.

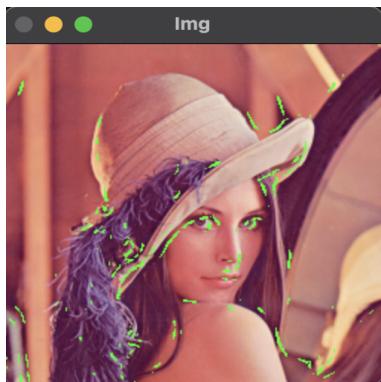
Meglio utilizzare fx,fy perchè con il parametro None non è garantito che le nuove dimensioni siano corrette.

Convertiamo le due immagini in scala di grigi:

```
gray_img_lena = cv2.cvtColor(img_lena, cv2.COLOR_BGR2GRAY)
resized_gray = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)
```

Estriamo gli angoli dall'immagine rimpicciolita ed evidenziiamoli con il rosso:

```
dst = cv2.cornerHarris(resized_gray, 2, 23, 0.04)
resized[dst > 0.01 * dst.max()] = [0,255,0]
cv2.imshow('Img', resized)
```



Come possiamo vedere ci sono diversi angoli:

Il rilevamento dell'angolo Harris non è invarianta di scala, il che significa che se cambiamo la risoluzione dell'immagine, non è garantito che avremo gli stessi angoli rilevati dopo la modifica della risoluzione.

Questo è un problema con delle applicazioni: Ad esempio con un immagine Panorama, se modifichiamo le immagini, le ruotiamo ,...

con Harris non troverebbe i punti per matchare insieme le immagini.

Quindi nel matching di immagini in un panorama è importante avere una robustezza della feature rispetto alle modifiche di scala e di rotazione e quindi Harris non va bene

Vediamo altri estrattori di caratteristiche che sono invarianti alla scala:

3.16.1 Scale- Invariant feature Transform (SIFT)

La Scale-Invariant Feature Transform (SIFT) è un algoritmo di computer vision ampiamente utilizzato per l'estrazione di feature invarianti di scala dalle immagini.

L'algoritmo SIFT si basa sulla costruzione di una "piramide di scale" dell'immagine, dove l'immagine originale viene ridotta gradualmente di dimensione utilizzando un filtro gaussiano.

Il primo passaggio della tecnica Pyramid Feature Transform (PFT) consiste nella costruzione della piramide di scale dell'immagine. Questa piramide di scale è costituita da una serie di immagini a diverse risoluzioni, ottenute applicando un filtro gaussiano e riducendo le dimensioni dell'immagine originale.

Ogni livello della piramide corrisponde a un'immagine a una determinata scala.

Quante volte viene applicato questo procedimento di filtro gaussiano e ridimensionamento? Di solito lo standard è 8 volte (8 livelli nella piramide)

Perchè utilizziamo questo approccio? Supponiamo di trovare un punto nel primo livello della piramide, al prossimo livello perdiamo informazioni per la scala e il filtro blur.

Quindi se nei diversi livelli abbiamo lo stesso punto significa che è invariante alla scala.

Successivamente, vengono estratti i punti di interesse invarianti di scala dall'immagine utilizzando l'algoritmo SIFT. L'algoritmo SIFT utilizza una serie di filtri per rilevare le caratteristiche invarianti di scala dell'immagine. In particolare, l'algoritmo SIFT utilizza un filtro di Laplace per individuare i punti chiave nell'immagine a diverse scale, che corrispondono ai punti di interesse.

Infine, vengono utilizzate le feature invarianti di scala e le informazioni sulla scala per mappare i punti di interesse dell'immagine originale alle corrispondenti posizioni dell'immagine ridotta.

In questo modo, le feature invarianti di scala vengono adattate alla scala dell'immagine ridotta, consentendo di identificare gli stessi punti di interesse in diverse scale dell'immagine.

Questa tecnica consente di identificare i punti di interesse in diverse scale dell'immagine, rendendo l'algoritmo SIFT invarianti alla scala e adatto per l'analisi di immagini in diverse condizioni di illuminazione e scala.

```
sift = cv2.SIFT_create()
keypoints , descriptors = sift.detectAndCompute(img_lena, None) # trova prima i keypoints e
                     crea il descriptors
cv2.drawKeypoints(img_lena, keypoints,img_lena, (255,0,0),
                  cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
see(img_lena, 'SIFT', 2000)
```

"sift" è l'oggetto che rappresenta l'algoritmo SIFT creato utilizzando il metodo "cv2.SIFT_create()".

Il metodo "detectAndCompute" viene utilizzato per trovare i keypoints e creare i descriptors a partire dall'immagine "img_lena". Questo metodo prende in input l'immagine di cui si vogliono estrarre le caratteristiche (in questo caso "img_lena") e un parametro opzionale che indica la presenza di una maschera. In questo caso, la maschera viene impostata a "None".

Il metodo "detectAndCompute" restituisce due valori: "keypoints" e "descriptors". I keypoints rappresentano i punti salienti dell'immagine in cui si concentrano le caratteristiche più rilevanti. I descriptors sono i vettori che descrivono le caratteristiche dei keypoints.

cv2.drawKeypoints è una funzione di OpenCV che permette di disegnare i keypoints su un'immagine. I keypoints sono rappresentati da cerchi che indicano la posizione e l'orientamento di punti di interesse nell'immagine. La funzione prende come input l'immagine di destinazione, i keypoints, l'immagine di origine, il colore e la flag di drawing. Il flag cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS indica

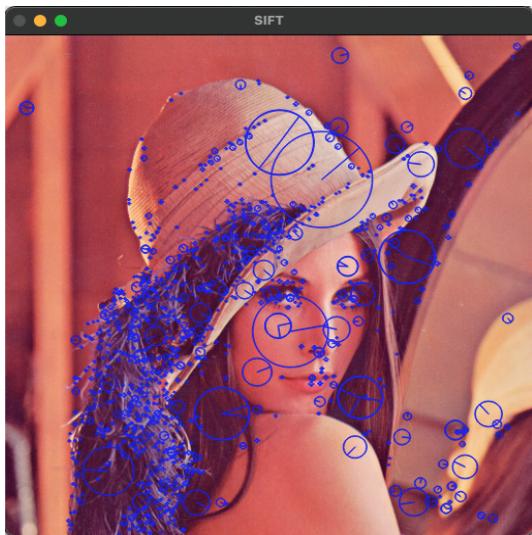
che i keypoints devono essere disegnati con informazioni aggiuntive come la scala e l'orientamento.

L'estrattore di caratteristiche SIFT è un algoritmo che analizza un'immagine e individua i punti chiave significativi (key point) e i loro descrittori associati.

Questa regione viene divisa in sottoregioni e per ognuna di esse viene calcolato un istogramma orientato dei gradienti.

Il descrittore finale è costituito dalla concatenazione di tutti gli istogrammi.

In sintesi, l'estrattore di caratteristiche SIFT analizza un'immagine e individua i punti chiave e i loro descrittori associati. Questi possono essere utilizzati per identificare e confrontare immagini, riconoscere oggetti, individuare punti di interesse, e così via.



Come possiamo vedere abbiamo molti più punti chiave rispetto ad Harris corner, per ogni punto chiave abbiamo un cerchio (più grande è il cerchio più robusto è il punto chiave) e anche la direzione del punto chiave.

Ovviamente questi punti chiave sono quelli che sopravvivono dopo il filtro blur e dopo il ridimensionamento effettuato più di una volta.

È possibile avere cerchi con più direzioni perché ci sono due punti chiavi che agiscono in due direzioni diverse.

Se si guarda la variabile del punto chiave vedremo che non contiene solo le coordinate x e y, ma anche altre informazioni come il diametro del cerchio, l'angolo del gradiente, il livello della piramide, id (perchè deve essere usato dopo), ...

3.16.2 Speeded-Up Robust Features - SURF

SURF (Speeded-Up Robust Features) è un altro algoritmo di estrazione di caratteristiche che è stato sviluppato per superare alcune limitazioni di SIFT, in particolare la sua lentezza di calcolo. In sostanza, SURF è una versione ottimizzata di SIFT, che utilizza una rappresentazione differenziale dell'immagine e una versione approssimata dell'operatore Laplaciano del gaussiano per individuare i punti di interesse.

In OpenCV, puoi creare un oggetto SURF utilizzando la funzione `cv2.xfeatures2d.SURF_create()`. Come per SIFT, puoi utilizzarne

3.16.3 AKAZE (Accelerated-KAZE)

AKAZE è un algoritmo di estrazione delle caratteristiche utilizzato per l'elaborazione delle immagini. Si basa su una modifica dell'algoritmo KAZE, ma con una maggiore velocità di elaborazione.

AKAZE utilizza una combinazione di detettori di bordi multiscale, filtri di differenza di Gaussiana (DoG) e filtri di tipo Hessian per estrarre le caratteristiche dall'immagine. In particolare, AKAZE utilizza una scala adattiva per adattarsi alla struttura dell'immagine in modo da estrarre caratteristiche con diverse scale e orientamenti.

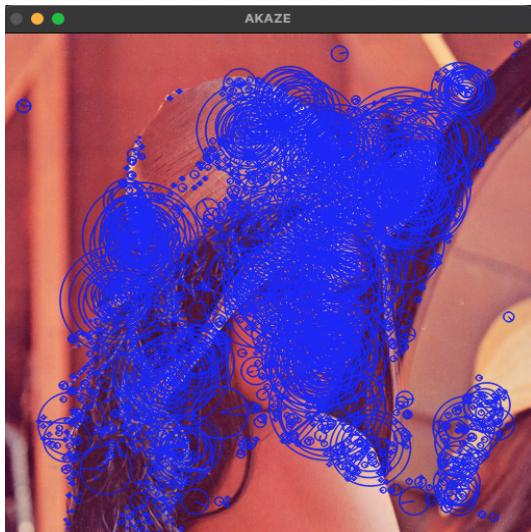
AKAZE è stato progettato per essere robusto rispetto alle rotazioni, alle traslazioni e alle distorsioni dell'immagine. Inoltre, AKAZE utilizza una tecnica di non massima soppressione per eliminare le caratteristiche ridondanti e una tecnica di orientamento per migliorare la descrizione delle caratteristiche.

In OpenCV, AKAZE può essere utilizzato come estrattore di caratteristiche per l'abbinamento delle immagini, la rilevazione degli oggetti e la ricostruzione 3D.

```
akaze = cv2.AKAZE_create()
keypoints , descriptors = akaze.detectAndCompute(img_lena, None)

cv2.drawKeypoints(img_lena, keypoints,img_lena, (255,0,0),
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

see(img_lena, 'AKAZE', 5000)
```



3.17 Binary feature extractors.

I binary feature extractors sono degli algoritmi utilizzati per estrarre le caratteristiche dalle immagini, in modo da poterle utilizzare per il riconoscimento e la comparazione di immagini. A differenza dei descrittori di caratteristiche convenzionali, che producono descrittori continui, i binary feature extractors producono descrittori binari, ovvero vettori di 0 e 1.

ORB (Oriented FAST and Rotated BRIEF) e BRISK (Binary Robust Invariant Scalable Keypoints). Questi

algoritmi utilizzano tecniche come l'analisi di Harris e la tecnica di rilevamento dei punti di interesse.

La logica dietro l'utilizzo di descrittori binari è quella di rendere il processo di confronto delle immagini più veloce ed efficiente, poiché i descrittori binari possono essere confrontati utilizzando operazioni di bit a bit, a differenza dei descrittori continui che richiedono operazioni più complesse.

In sostanza, i binary feature extractors sono algoritmi utilizzati per estrarre le caratteristiche dalle immagini in modo efficiente e veloce, utilizzando descrittori binari che semplificano il processo di confronto delle immagini.

3.17.1 ORB - Oriented FAST and Rotated BRIEF

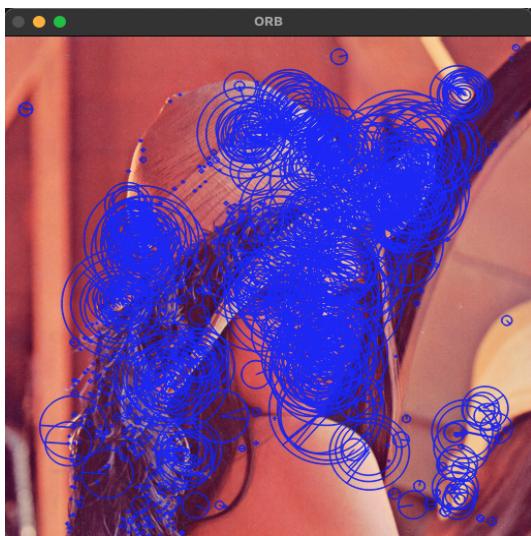
In pratica, ORB calcola i punti chiave dell'immagine utilizzando il metodo FAST. Per ogni punto chiave, viene calcolata l'orientazione e una descrizione binaria utilizzando la tecnica di BRIEF. La descrizione binaria viene quindi utilizzata per la corrispondenza delle feature tra le immagini.

Il vantaggio dell'approccio binario è la sua efficienza computazionale e la sua robustezza contro il rumore e le distorsioni.

```
orb = cv2.ORB_create()
keypoints , descriptors = orb.detectAndCompute(img_lena, None)

cv2.drawKeypoints(img_lena, keypoints,img_lena, (255,0,0),
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

see(img_lena, 'ORB', 5000)
# La differenza con SIFT e' che trova piu' punti chiave ma si concentra su meno zone (sulle
# zone di focus)
# Dobbiamo limitare il numero di features (punti chiave) perche' la stessa funzione viene
# calcolata
# due volte/tre volte. Per limitare questo:
orb = cv2.ORB_create(nfeatures=500) # si puo' fare cos
```



4 NLP.

Il Natural Language Processing (NLP) è un campo dell'informatica e dell'intelligenza artificiale che si occupa di elaborare il linguaggio naturale umano. L'obiettivo principale del NLP è quello di far comprendere ai computer il linguaggio naturale umano (rendere la macchina capace di capire il nostro linguaggio naturale) e di sviluppare algoritmi e applicazioni che possano elaborare e analizzare il testo scritto o parlato. Ciò è particolarmente importante perché il computer "capisce" solo i numeri e i linguaggi di programmazione.

Il NLP si basa su tecniche di apprendimento automatico, elaborazione del segnale, linguistica computazionale e altre tecnologie dell'informatica per analizzare e manipolare il linguaggio naturale. Le applicazioni del NLP sono molteplici e includono la traduzione automatica, il riconoscimento del parlato, l'analisi del sentimento, la generazione di testo automatica, l'elaborazione di grandi quantità di testo (text mining), l'etichettatura del linguaggio (part-of-speech tagging) e l'analisi semantica.

Il NLP è diventato sempre più importante negli ultimi anni, grazie all'aumento delle grandi quantità di dati testuali disponibili grazie all'avvento di internet e dei social media.

Le applicazioni di NLP sono utilizzate in molti settori, come ad esempio il marketing, la finanza, la medicina, l'assistenza sanitaria e la sicurezza informatica.

Una delle tecniche più comuni è l'utilizzo di algoritmi di machine learning, che consentono alla macchina di apprendere dalle esperienze passate e migliorare la sua capacità di elaborare il linguaggio naturale. Gli algoritmi di machine learning possono essere addestrati utilizzando grandi quantità di dati testuali, in modo da far sì che la macchina impari a riconoscere i modelli e le strutture del linguaggio.

Inoltre, per far sì che la macchina comprenda il linguaggio naturale, è necessario sviluppare un sistema di rappresentazione del linguaggio naturale, in modo che le parole e le frasi possano essere convertite in una forma che la macchina possa elaborare. Una delle tecniche più comuni per fare ciò è l'utilizzo di alberi sintattici, che rappresentano la struttura grammaticale delle frasi in modo gerarchico.

Infine, per far sì che la macchina possa comprendere il linguaggio naturale, è necessario creare un dizionario, o insieme di regole, che definisca le corrispondenze tra le parole e le frasi del linguaggio naturale e i concetti che la macchina può comprendere e utilizzare.

Questo dizionario può essere creato manualmente dagli esperti del dominio, oppure può essere creato automaticamente utilizzando tecniche di apprendimento automatico e di elaborazione del linguaggio naturale.

Uno scopo è quello di predire la parola prossima in un testo, utilizzato molto in machine translation.

NLP viene utilizzato da alexa, google home, ...

Quando si parla con Alexa, il sistema NLP dietro l'assistente vocale elabora il linguaggio naturale dell'utente e cerca di capire ciò che si sta chiedendo.

Il sistema di NLP di Alexa utilizza tecniche di riconoscimento del parlato per convertire il suono delle parole in testo, che viene quindi elaborato per comprendere il significato della frase.

Questo processo richiede la comprensione del contesto in cui viene pronunciata la frase, la comprensione delle parole utilizzate e la sintassi della frase.

Si può dare anche un'immagine in input e cercare di dare una caption per l'immagine.

NLP è più difficile rispetto alla computer vision, perché quest'ultima lavora con immagini che sono costituiti da pixel (numeri) e la macchina lavora bene con i numeri. Invece il linguaggio dei testi umano è creato per noi ed è difficile che le macchine capiscano questo linguaggio.

NLP è difficile per alcuni aspetti principali:

- Ambiguity (ambiguità): c'è ambiguità nei linguaggi umani, nelle singole parole , in una parte del discorso, nelle strutture sintattiche
- Scale: nell'asse x abbiamo i linguaggi, nell'asse y abbiam i task che dobbiamo eseguire con i linguaggi. Più compiti vogliamo fare, più difficile sarà, anche per una sola lingua. Invece se si utilizzano più linguaggi sarà difficile anche per un singolo compito. Immaginiamo che vogliamo eseguire molti task con molti linguaggi. Un esempio di un singolo task per molte lingue è Google translate, che ancora non funziona bene con alcune lingue.
- Sparsity: la distribuzione delle parole. Se una parola compare molto all'interno di un documento non significa che quella parola sia importante.
- Variation: stesso grafico della scale, ma con un asse in più, l'asse z che contiene il dominio: se stiamo lavorando con il testo inglese, ma stiamo facendo una conversazione telefonica, se stiamo scrivendo un articolo per il giornale, se stiamo scrivendo una mail formale, .. cambia il registro della conversazione.
- Expressivity: è un'estensione dell'ambiguità: ambiguità significa che la propria forma può avere significati diversi; espressività significa che lo stesso significato può avere forme diverse; Ci sono vari esempi: Ho dato da fare le flessioni all'allievo vs ho dato da fare l'allievo alle flessioni.

Inoltre NLP non considera il tono, l'ironia, l'intenzione, il contesto, intimità, l'aspetto culturale (ad esempio in Giappone non si utilizza la parola 'tu', ma si utilizza 'nome cognome sun' in un discorso. Mentre in Europa la parola 'tu' - 'you' è molto importante),...

Quindi anche per l'aspetto culturale è molto difficile creare un traduttore preciso, con le tecniche di machine learning, i traduttori sono migliori rispetto ai vecchi perché all'inizio NLP seguiva delle regole per tradurre le frasi.

Quindi senza l'utilizzo dell'AI c'è bisogno di seguire molte regole, mentre ai giorni d'oggi con l'AI c'è necessità di alimentare il modello durante la fase di addestramento con molti dati.

NLP nell'ultimo periodo è cresciuto moltissimo grazie alla potenza di calcolo, grazie a chat-gpt, ... Più potenza di calcolo significa modelli più complessi, modelli più complessi significa gestire più casi, con più casi il modello funziona meglio.

Grazie ai dati che abbiamo oggi (aumentati molto), i modelli funzionano meglio e quindi il machine learning funziona meglio.

Chat-gpt è l'attuale stato dell'arte di NLP e lavora bene perché gli sono stati dati in input tutte le pagine di Wikipedia, molti siti web, l'intervento umano durante la formazione. Chat utilizza un loop umano di rinforzo, si può aiutare.

Quando usi chat gpt stai ancora addestrando il modello, utilizzo il reinforcement con i pollici all'insù e all'ingiù.

Le espressioni regolari sono utilizzate in NLP e sono espressioni che permettono di lavorare con il testo, permettono di combinare testi e possiamo estrarre le informazioni dalle stringhe (caratteri speciali), ...

Faremo solo alcune elaborazioni sul testo come abbiamo fatto per openCV. Ci sono varie azioni che possono essere effettuate:

Prima di vedere queste ultime dobbiamo scaricare alcuni modelli pre-addestrati per il processing del linguaggio naturale (NLP).

```

nltk.download('punkt') # questo è un modello pre-addestrato che sa come dividere il testo in
frasi
nltk.download("stopwords") # fornisce una lista di parole comuni che vengono spesso omesse
durante l'analisi del testo, come "il", "la", "di", "a", ecc.

```

```
nltk.download("averaged_perceptron_tagger") # rete neurale pre-addestrata, si deve solo scaricare: averaged_perceptron_tagger
nltk.download('tagsets') # per il tagging (assegnare a ogni parola un'etichetta)
nltk.download('wordnet') # per lemmatizing (cio la conversione di una parola al suo lemma, ad esempio "gatti" diventa "gatto") e sinonimi per le parole in un testo.)
```

4.1 Tokenizing.

La tokenizzazione consiste nel processo di suddividere un testo in unità più piccole, chiamate token. Ci sono principalmente due tipi di tokenizzazione:

- Tokenizzazione delle parole: il testo viene suddiviso in parole separate. Ad esempio, la frase "Questo è un esempio di tokenizzazione" verrebbe divisa in "Questo", "è", "un", "esempio", "di", "tokenizzazione".
- Tokenizzazione delle frasi: il testo viene suddiviso in frasi separate. Ad esempio, il paragrafo "Questo è un esempio di tokenizzazione. La tokenizzazione è un passaggio importante nell'elaborazione del linguaggio naturale." verrebbe diviso in "Questo è un esempio di tokenizzazione." e "La tokenizzazione è un passaggio importante nell'elaborazione del linguaggio naturale."

Vediamo il funzionamento, innanzitutto importiamo la libreria:

```
from nltk.tokenize import sent_tokenize, word_tokenize
```

Dopo di che separiamo per frasi:

```
example_sentences = "Hello. I'm Federico."
sent_t = sent_tokenize(example_sentences) # splitta in frasi: quindi in questo caso divide le frasi con i punti .
print(sent_t)
```

Avremo come risultato: ['Hello.', "I'm Federico."]

Ora separiamo per parole:

```
sent_w1 = word_tokenize(example_sentences)
print(sent_w1)
```

Avremo come risultato: ['Hello', '.', 'I', "'m", 'Federico', '.']

In questo caso separa per parole e per frasi. È anche in grado di capire che I'm sono due parole diver-

Altro esempi:

```
example_string = "all the speed he took, all the turn he'd takend and corners in Night City..."
sent_w = word_tokenize(example_string) # splitta parole nella frase
print(sent_w)
# ['all', 'the', 'speed', 'he', 'took', ',', 'all', 'the', 'turn', 'he', "'d", 'takend',
# 'and', 'corners', 'in', 'Night', 'City', '...']
```

```
quote = "It's leviosa, not leviosaaaa"
word_in_quote = word_tokenize(quote)
print(word_in_quote)
# ['It', "'", 's", 'leviosa', ',', 'not', 'leviosaaaa']
```

4.1.1 stepwords

In elaborazione del linguaggio naturale (NLP), il termine "stepwords" si riferisce alle parole che vengono comunemente utilizzate in una lingua ma che non aggiungono molto significato ad una frase. Spesso queste parole vengono filtrate durante l'elaborazione del testo per concentrarsi sulle parole importanti che trasmettono il messaggio principale del testo.

Esempi di stepwords in inglese includono gli articoli ("il", "un", "una"), le preposizioni ("in", "su", "a"), le congiunzioni ("e", "ma", "o"), e alcuni verbi comuni ("essere", "avere", "fare").

Filtrare le stepwords può aiutare a migliorare l'accuratezza di alcune attività di NLP come la classificazione del testo e l'analisi del sentimento, poiché consente al modello di concentrarsi sulle parole più significative in una frase.

Importiamo la libreria:

```
from nltk.corpus import stopwords
```

Prima dobbiamo definire i stopwords e dobbiamo indicare a nltk quali stop words dobbiamo utilizzare, in questo caso quello inglese.

```
stop_words = set(stopwords.words('english'))
```

Possiamo filtrare la nostra frase utilizzando questi stop_words:

```
filtered_word = []
```

Nel filtered_word inseriamo tutte le parole, tranne lo stopwords:

```
for word in word_in_quote:  
    if word.casfold() not in stop_words:  
        filtered_word.append(word)  
  
print(filtered_word)  
# ['s', 'leviosa', ',', 'leviosaaaa']
```

Questo è strano perché non c'è 'I', 't'. ”.

Questo dipende dall'analisi che dobbiamo fare e in generale, abbiamo 2 tipi di lavori:

- content words: parole che danno informazioni sull'argomento all'interno del testo,
- context words: parole che danno informazioni sullo stile di scrittura: troviamo dei patterns su come l'autore scrive, ...

4.2 Stemming

Lo Stemming è il processo di ridurre una parola alla sua radice o "stem", ovvero la parte della parola che rimane dopo che sono stati rimossi i suffissi e i prefissi. Ad esempio, le parole helper , helping (sono parole diverse, ma con la stessa radice) -> help.

Lo stemming viene utilizzato comunemente nel natural language processing (NLP) per ridurre il numero di parole diverse presenti in un testo, aiutando a semplificare l'analisi del testo e migliorare la precisione di alcune tecniche di elaborazione del linguaggio naturale. Tuttavia, a causa della sua natura semplice, lo stemming può anche produrre risultati imprecisi o non corretti in alcuni casi. Per esempio, "intelligenza" e "intelligente" hanno la stessa radice "intelligent" se sottoposte ad uno stemming, ma possono avere significati molto diversi all'interno di un testo.

Vediamone il funzionamento, inanzitutto importiamo la libreria:

```
from nltk.stem import PorterStemmer
```

Creiamo un istanza di stem:

```
stemmer = PorterStemmer()
```

Definiamo la stringa da stemmare:

```
string_to_stem = 'The crew of USS discovery, discovered many discoveries. Discovering is what
explorers do.'
```

Tokenizziamo la stringa:

```
words = word_tokenize(string_to_stem)
```

A questo punto possiamo applicare lo stemmer alle parole estratte:

```
stemmed_words = [stemmer.stem(word) for word in words]

print(stemmed_words)
# ['the', 'crew', 'of', 'uss', 'discoveri', ',', 'discov', 'mani', 'discoveri', '.', 'discov',
'is', 'what', 'explor', 'do', '.']
```

Tutte le parole che contengono 'discov...' sono messe però in radici diverse perchè di solito ci sono 2 problemi quando si applica questa tecnica:

- under stemming: avviene quando 2 parole che devono avere la stessa radice non sono ridotte alla stessa radice (false negative)
- over stemming: avviene quando 2 parole che non devono avere la stessa radice vengono messe nella stessa radice (falsi positivi)

Questo avviene perchè c'è un algoritmo vecchio del 1979, e magari la lingua inglese è cambiata ,...

4.3 POS: part of speech (discorso) tagging.

Chat-GPT funziona perchè capisce nella frasi quali parole sono nome, verbi, aggettivi, ...

La tecnica del POS è proprio questa: assegnare un'etichetta a ciascuna parte della frase.

Ci sono già delle etichette standard: noun (sostantivo), pronouns (you, she, he, ...) Importiamo la libreria:

```
import nltk
```

Definiamo la string da etichettare:

```
dijkstra = 'Computer science is no more about computers than astronomy is about telescopies.'
```

Prima del tagging dobbiamo spartire la frase in parole: (tokenizing the quote)

```
words = word_tokenize(words = word_tokenize(dijkstra))
```

A questo punto possiamo applicare il pos alle parole estratte:

```
result = nltk.pos_tag(words)

print(result)
# [('Computer', 'NNP'), ('science', 'NN'), ('is', 'VBZ'), ('no', 'DT'), ('more', 'RBR'),
# ('about', 'IN'), ('computers', 'NNS'), ('than', 'IN'), ('astronomy', 'NN'), ('is', 'VBZ'),
# ('about', 'IN'), ('telescopies', 'NNS'), ('.', '.')
```

Per ogni parola abbiamo un'etichetta:

- tutto quello che inizia con NN è un noun-sostantivo.
- VB: verb
- ...

Con la funzione:

```
nltk.help.upenn_tagset()
```

C'è una guida su tutte le etichette, ...

4.4 Lemmatizing.

Lemmatizing è una tecnica di elaborazione del linguaggio naturale che consiste nel ridurre le parole di una frase alla loro forma base, chiamata lemma, utilizzando la conoscenza della grammatica della lingua. Ad esempio, lemmatizing ridurrebbe le parole "corro", "correndo" e "corse" al lemma "correre".

La differenza principale tra stemming e lemmatizing è che stemming utilizza una tecnica di taglio delle parole per rimuovere i suffissi, mentre lemmatizing utilizza la conoscenza della grammatica della lingua per determinare il lemma di una parola. Questo rende lemmatizing più preciso di stemming, ma anche più lento e computazionalmente intensivo.

Lemmatizing è utile in molti casi, ad esempio nella creazione di un indice di parole per una ricerca testuale, nella normalizzazione del testo per l'analisi del sentimento e nell'elaborazione di query in un motore di ricerca.

sciarpe (sciarpe) -> (lemmatizing) scarv-scarf (sciarpa)

Importiamo la libreria:

```
from nltk.stem import WordNetLemmatizer
```

creiamo un'istanza e facciamo il lemmatizing:

```
lemmatizer = WordNetLemmatizer()
result = lemmatizer.lemmatize('scarves')
print(result)
# scarf
```

Cosa succede se vogliamo fare il lemmatizing di una parola molto differente dal core-meaning (significato fondamentale)

```
result = lemmatizer.lemmatize('worst')
print(result)
# worst
```

Rimane così perché in questo caso il lemmatizer assume che worst sia un sostantivo , invece è un aggettivo. Per risolverlo possiamo fare così:

```
result = lemmatizer.lemmatize('worst', pos='a')
print(result)
# bad
```

Abbiamo aiutato il lemmatizing dicendo che 'worst' non è un sostantivo, ma un aggettivo.

È importante etichettare correttamente le parole perché in NLP c'è una cosa chiamata omografo, diverso nell'omografia in computer vision.

L'omografo in PNL è quando due parole sono scritte nello stesso modo, ma il significato è diverso. (orsa significa orso e sostenere)

4.5 Chunking.

Qui che entrano in gioco le espressioni regolari.

Il Chunking è una tecnica di elaborazione del linguaggio naturale che permette di raggruppare le parole in unità più grandi, chiamate "chunk", sulla base della loro struttura grammaticale. Ad esempio, un chunk può essere formato da un sostantivo seguito da uno o più aggettivi che lo descrivono. Questa tecnica può aiutare a comprendere il significato di una frase o di un testo in modo più preciso e a estrarre informazioni rilevanti per determinate applicazioni.

Chunking è differente da tokenizing perché in quest'ultimo è principalmente per trovare parole. Invece chunking ci permette di identificare le frasi. Quest'ultimo fa uso del post tagging a differenza del tokenizing che usa per esempio un carattere speciale come lo spazio, ...
Con il chunking diciamo esplicitamente come vogliamo la frase grazie alla definizione della struttura della frase che voglio.

```
lotr= "It's a dangerous business, Frodo, going out your door."
```

Ora dobbiamo trovare i tagging (nome, aggettivo, verbo, ...) perchè chunking fa uso di questi.
Prima tokenize poi tagging.

```
words_in_lotr = word_tokenize(lotr)
```

ora installiamo una rete neurale pre-addestrata, si deve solo scaricare: averaged_perceptron_tagger.
Grazie a questa diamo in input delle parole e automaticamente etichetta tutte le parole con i tag.
Tag words:

```
lotr_pos_tagging = nltk.pos_tag(words_in_lotr) # fa la stessa cosa di POS
print(lotr_pos_tagging)
# [('It', 'PRP'), ("'s", 'VBZ'), ('a', 'DT'), ('dangerous', 'JJ'), ('business', 'NN'), (',', ','), ('', ','), ('Frodo', 'NNP'), ('', ','), ('', ','), ('going', 'VBG'), ('out', 'RP'), ('your', 'PRP$'), ('door', 'NN'), ('', '')]
```

Anche le virgole e i punti vengono etichettati.

Ora definiamo la nostra stringa che verrà utilizzata per estrarre la parola che vogliamo: cioè il template della frase che dovrò estrarre da un testo, ad esempio.

```
my_string = "NP: {<DT>?<JJ>*<NN>}"
```

NP: frase nominale: una frase che si basa su un sostantivo: ad esempio: the bottle is on the table:

DT: determiner (optional), ?: quantity: appartiene all'alfabeto delle espressioni regolari,

JJ: adjective , *: quanti aggettivi

NN: noun; stiamo dicendo che la frase che stiamo estraendo finisce con un sostantivo (NN)
sono i tagging del post tagging;

Ora possiamo utilizzare questa stringa come template per estrarre le frasi che seguono questo template:

Ora creiamo un parser(analizzatore) che sfrutta questo tipo di espressione regolare:

```
chunk_parser = nltk.RegexpParser(my_string)
```

```
# let's see how the chunker works:
tree = chunk_parser.parse(lotr_pos_tagging)
tree.draw()
```

quindi capisce la struttura della frase più grande.

