TP 1 – Introduction au traitement d'image

1- Introduction de NumPy

Numpy est le package scientifique le plus important de l'écosystème Python, car il fournit une structure de données commune sur laquelle de nombreux autres packages sont construits. L'objectif ici est d'introduire brièvement la structure de base des données, ndarray.

Qu'est-ce qu'un ndarray?

Le ndarray est la plus grande contribution de numpy. Un ndarray est une grille régulière de dimensions N, homogène par défaut (tous les éléments ont le même type), bloc de mémoire contigu avec les types correspondant aux types de machines (8 bits, 32 bits, 64 bits, ...).

A. Construction d'un tableau

Nous pouvons construire un tableau à partir de listes Python:

```
arr = np.array([
    [1.2, 2.3, 4.0],
    [1.2, 3.4, 5.2],
    [0.0, 1.0, 1.3],
    [0.0, 1.0, 2e-1]])
print(arr)
```

Inspection des propriétés du tableau:

```
print(arr.dtype)
print(arr.ndim)
print(arr.shape)
```

Ce tableau est de type float64. Il a 2 dimensions (4 lignes et 3 colonnes). Lors de la construction d'un tableau, nous pouvons spécifier explicitement le type:

```
iarr = np.array([1,2,3], np.uint8)
```

Les opérations arithmétiques sur la matrice respectent le type et peuvent inclure l'arrondi et le débordement

```
arr *= 2.5
iarr *= 2.5
print(arr)
print(iarr)
```

B. Opérations booléennes

Un important sous-ensemble d'opérations avec des tableaux numpy concerne l'utilisation d'opérateurs logiques pour construire des tableaux booléens. Par exemple:

```
is_greater_one = (arr >= 1.)
print(is_greater_one)
```

Nous pouvons utiliser l'opérateur [ ] de Python pour découper le tableau:

```
print(arr[0,0]) # First row, first column
print(arr[1]) # The whole second row
print(arr[:,2]) # The third column
```

Les tranches sont des vues, qui partagent la mémoire avec le tableau d'origine!

```
print("Before: {}".format(arr[1,0]))
view = arr[1]
view[0] += 100
print("After: {}".format(arr[1,0]))
```

Un petit résumé illustré du tranchage (indexation) et du découpage NumPy

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```

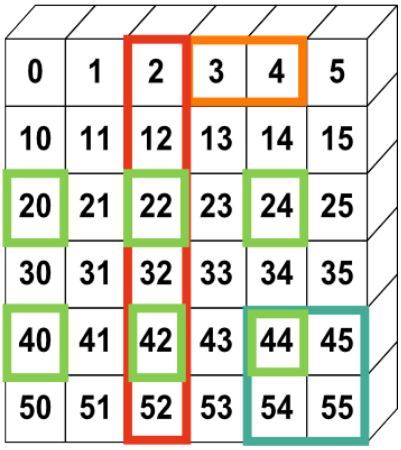


Figure 1 : l'indexation et du découpage NumPy (source : http://www.scipy-lectures.org/intro/numpy/array_object.html)

C. Fonctions de base sur les tableaux

La moyenne : `arr.mean()`

Également disponible : max, min, sum, ptp (point à point, c'est-à-dire différence entre les valeurs maximales et minimales).

Ces fonctions peuvent également fonctionner selon les axes:

```
arr.mean(axis=0)
```

Une astuce importante consiste à combiner des opérations logiques avec A

```
is_greater_one = (arr > 1)
print(is_greater_one.mean())
```

Plus de détail sur NumPy :

http://www.scipy-lectures.org/intro/numpy/array_object.html

2- Introduction à la vision par ordinateur avec OpenCV / Python

```
# library imports
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

A. Chargement des images avec OpenCV

Le chargement d'images avec OpenCV est simple, mais il y a certaines choses que vous devez spécifier. OpenCV importera toutes les images (niveaux de gris ou couleur) avec 3 canaux, donc pour lire une image en niveaux de gris comme il n'y a qu'un seul canal, vous devez passer l'arg 0 après l'emplacement de l'image. Les formats d'image suivants peuvent être lus par **cv2.imread()**:

- Windows bitmaps - *.bmp,* .dib (always supported)
- JPEG files - *.jpeg,* .jpg, *.jpe (see the Notes section)
- JPEG 2000 files - *.jp2 (see the Notes section)
- Portable Network Graphics - *.png (see the Notes section)
- WebP - *.webp (see the Notes section)
- Portable image format - *.pbm,* .pgm, *.ppm (always supported)
- Sun rasters - *.sr,* .ras (always supported)
- TIFF files - *.tiff,* .tif (see the Notes section)
-

```
# load an image
img = cv2.imread('image_test.png')

# load an image as a single channel grayscale
img_single_channel = cv2.imread('images/dolphin.png', 0)

# print some details about the images
print('The shape of img without second arg is: {}'.format(img.shape))
print('The shape of img_single_channel is:     {}'.format(img_single_ch
annel.shape))
```

B. Affichage d'images avec OpenCV

Commençons par utiliser la fonction OpenCV .imshow (), qui prend deux arguments requis : 1er argument -> le nom de la fenêtre où l'image sera affichée et le 2ème argument -> l'image à afficher.

Outre la fonction cv2.imshow (), il existe quelques autres éléments requis pour que cela fonctionne correctement.

D'abord, la fonction cv2.waitKey (). Son argument -> le temps en millisecondes. La fonction attend des millisecondes spécifiées pour tout événement de clavier. Si vous

appuyez sur une touche pendant ce temps, le programme continue. Si 0 est passé, il attend indéfiniment une touche de clavier. Il peut également être configuré pour détecter des touches spécifiques, par exemple si vous appuyez sur une touche, etc.

En plus, la fonction cv2.destroyAllWindows (), pour détruire toutes les fenêtres que nous avons créées. Si vous souhaitez détruire une fenêtre spécifique, utilisez plutôt la fonction cv2.destroyWindow () où vous passez le nom exact de la fenêtre en tant qu'argument.

```
# display the image with OpenCV imshow()
cv2.imshow('OpenCV imshow()', img)

# OpenCV waitKey() is a required keyboard binding function after
 imwshow()
cv2.waitKey(0)

# destroy all windows command
cv2.destroyAllWindows()
```

Vous pouvez créer une fenêtre et y charger une image ultérieurement. Dans ce cas, vous pouvez spécifier si la fenêtre est redimensionnable ou non. Cela se fait avec la fonction cv2.namedWindow (). Par défaut, l'indicateur est cv2.WINDOW_AUTOSIZE. Mais si vous spécifiez flag comme étant cv2.WINDOW_NORMAL, vous pouvez redimensionner la fenêtre. Cela sera utile lorsque l'image est trop grande en dimension et en ajoutant une barre de suivi aux fenêtres.

Sachez que les images couleur chargées par défaut dans OpenCV sont en mode BGR (bleu, vert, rouge).

C. Enregistrer des images avec OpenCV

L'enregistrement d'images avec OpenCV se fait avec la fonction cv2.imwrite (). Cette fonction prend un chemin relatif ou absolu où vous souhaitez enregistrer l'image et l'image que vous souhaitez enregistrer. Voici un code que vous pouvez essayer à partir d'un script Python.

```
# Saving an Image on a key press
img = cv2.imread('image_test.png')
cv2.imshow('Option to Save image', img)
print("press 's' to save the image as 'image_test_2.png\n")
key = cv2.waitKey(0) # NOTE: if you are using a 64-bit machine,
this needs to be: key = cv2.waitKey(0) & 0xFF
if key == 27: # wait for the ESC key to exit
    cv2.destroyAllWindows()
elif key == ord('s'): # wait for 's' key to save and exit
    cv2.imwrite('image_test_2.png', img)
    cv2.destroyAllWindows()
```

```
# write an image with imwrite
image_to_save = 'images/'image_test_3.png’
cv2.imwrite(image_to_save, img)

print('Image saved as {}'.format(image_to_save))
```

Comment OpenCV gère les plans de couleur ?

D’abord, il faut lire et afficher l'image comme référence

```
# imports
import numpy as np
import cv2
import matplotlib.pyplot as plt
from matplotlib import gridspec

# read and display the image as a reference
img = cv2.imread('images/fruit.png')
cv2.imshow('Fruit Image', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Puisque l'image chargée est en couleur, sa taille est de 3 chiffres : # height (# of rows), width (# de cols) et enfin des plans de couleur (BGR)

```
height, width, channels = img.shape[:3]
print 'Image height: {}, Width: {}, # of channels: {}'.format(height, w
idth, channels)
```

Rappelez-vous que openCV lit comme mode BGR, donc le canal 0 est bleu, le canal 1 est vert et le canal 2 est rouge

```
blues = img[:, :, 0]
greens = img[:, :, 1]
reds = img[:, :, 2]
```

Afficher les plans d’image bleu, vert et rouge pour l’image de fruit en utilisant imshow()

```
cv2.imshow('Fruit Blues', blues)
cv2.imshow('Fruit Greens', greens)
cv2.imshow('Fruit Reds', reds)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Tracer les valeurs pour chaque plan de couleur sur une ligne spécifique :

```
# plot values for each color plane on a specific row
fig = plt.figure(figsize=(10, 4))
gs = gridspec.GridSpec(1, 2, width_ratios=[1, 1])
```

```
# original image
ax0 = plt.subplot(gs[0])
ax0.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB)) # need to convert BGR
to RGB
ax0.axhline(50, color='black') # show the row being used
ax0.axvline(100, color='k'), ax0.axvline(225, color='k') # ref lines

# image slice
ax1 = plt.subplot(gs[1])
ax1.plot(blues[49, :], color='blue')
ax1.plot(greens[49, :], color='green')
ax1.plot(reds[49, :], color='red')
ax1.axvline(100, color='k', linewidth=2), ax1.axvline(225, color='k', l
inewidth=2)

plt.suptitle('Examen des valeurs du plan de couleur pour une seule lign
e')
plt.show()
```

Rogner (crop) l'image :

```
cropped = img[109:310, 9:160]
cv2.imshow('Cropped Image', cropped)
print "press 's' to save the image as cropped_bicycle.png\n"
key = cv2.waitKey(0) # if you are using a 64-bit machine see below
# the above line should be: key = cv2.waitKey(0) & 0xFF
if key == 27: # wait for the ESC key to exit
    cv2.destroyAllWindows()
elif key == ord('s'): # wait for 's' key to save and exit
    cv2.imwrite('images/cropped_bicycle.png', img)
    cv2.destroyAllWindows()

# get the size of the cropped image
height, width = cropped.shape[:2]
print 'Cropped Width: {}px, Cropped Height: {}px'.format(width, height)
```

Comment appliquer des opérateurs arithmétiques afin d'additionner, de soustraire et de fusionner des images dans OpenCV ?

Vous pouvez ajouter deux images par la fonction cv.add () d'OpenCV ou simplement par opératio res = img1 + img2 de numpy. Les deux images doivent avoir la même profondeur et le même type, ou la seconde image peut simplement être une valeur scalaire.

Attention : Il y a une différence entre l'addition d'OpenCV et l'addition de Numpy. L'addition d'OpenCV est une opération saturée alors que l'addition de Numpy est une opération modulo. Par exemple, considérons l'exemple ci-dessous:

```
x = np.uint8([250])
y = np.uint8([10])
print 'Open CV Addition {}'.format(cv2.add(x, y)) # 250+10 = 260 => 255
```

```
print ''
print 'Numpy Addition {}\n'.format(x+y) # 250+10 = 260 % 256 = 4
```

Si nous voulons appliquer la fonction `cv2.add(), l`es images doivent avoir la même taille pour une addition par paire ou le second argument peut simplement être une valeur scalaire pour l'ajout d'éléments :

```
if bicycle.shape[:2] == dolphin.shape[:2]:
    sum_img = cv2.add(bicycle, dolphin) # add images together
    cv2.imshow('Summed Images', sum_img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    scaled_img = cv2.add(bicycle, 50)
    cv2.imshow('Scalar Addition on Bicycle Image', scaled_img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Il y a aussi la foction de soustraction absolue cv2.absdiff. Il faut l'utiliser d'en sorte que nous n'avons pas d'intensités négatives, l'ordre n'a pas d'importance avec absdiff :

```
if bicycle.shape[:2] == dolphin.shape[:2]:
    diff = cv2.absdiff(bicycle, dolphin)
    cv2.imshow('Subtracted Images', diff)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Nous pouvons aussi trouver l'image moyenne entre deux images. Ici, nous montrons deux façons de le faire et leurs images résultantes :

```
if bicycle.shape[:2] == dolphin.shape[:2]:
    average_img = bicycle / 2 + dolphin / 2
    alt_average_img = cv2.add(bicycle, dolphin) / 2
    cv2.imshow('Averaged Images', average_img)
    cv2.imshow('Alt. Averaged Images', alt_average_img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Image Blending

Nous pouvons aussi additionner deux images via la fonction de *Blending* cv2.addWeighted. Il s'agit d'une addition d'image, mais des poids différents sont attribués aux images pour donner une impression de mélange ou de transparence. Les images sont ajoutées selon l'équation ci-dessous :

$$g(x) = (1 - \alpha)f_0(x) + \alpha f_1(x)$$

Ici, j'ai pris deux images pour les mélanger. La première image se voit attribuer un poids de 0,7 et la deuxième image, 0,3. cv.addWeighted () s'applique à l'équation suivante sur l'image.

$$dst = \alpha \cdot img1 + \beta \cdot img2 + \gamma$$

Ici, γ est pris comme zéro.

```
dst = cv.addWeighted(img1,0.7,img2,0.3,0)
```

D. Seuillage d'image

Opencv offre une fonction cv2.threshold qui permet d'effectuer le seuillage. Le premier argument de cette fonction est l'image source, qui doit être une image en niveaux de gris. Le deuxième argument est la valeur de seuil utilisée pour classer les valeurs de pixels. Le troisième argument est le maxVal qui représente la valeur à donner si la valeur du pixel est supérieure à (parfois inférieure à) la valeur du seuil. OpenCV fournit différents styles de seuillage et est décidé par le quatrième paramètre de la fonction. Différents types sont:

- cv2.THRESH_BINARY
- cv2.THRESH_BINARY_INV
- cv2.THRESH_TRUNC
- cv2.THRESH_TOZERO
- cv2.THRESH_TOZERO_INV

Veuillez consulter la documentation en ligne d'Opencv pour comprendre clairement à quoi sert chaque type.

E. Conversion des couleurs

Pour la conversion des couleurs, nous utilisons la fonction cv2.cvtColor (image_entrée, flag) où l'indicateur détermine le type de conversion. Pour la conversion BGR → Gray, nous utilisons les indicateurs cv2.COLOR_BGR2GRAY. De même pour BGR→HSV, nous utilisons le drapeau cv2.COLOR_BGR2HSV.

Charger une image couleur, appliquer ces fonctions de conversion entre les différents espaces de couleur et afficher leur composant !

Afin de mieux comprendre l'utilité de conversion entre espaces, essayons d'implémenter un système de tracking d'objet coloré à partir d'un floux vidéo. D'abord, pour récupérer le flux vidéo, image par image, vous pouvez utiliser la fonction cv2.VideoCapture(0), qui exige l'installation du package *opencv-contrib-python*

Cet exemple applique un seuillage dans l'espace HSV. Testez-le et essayer de l'appliquer dans l'espace RGB.

```
import cv2
import numpy as np

cap = cv2.VideoCapture(0)

while(1):

    # Take each frame
    _, frame = cap.read()

    # Convert BGR to HSV
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # define range of blue color in HSV
    lower_blue = np.array([110,50,50])
    upper_blue = np.array([130,255,255])

    # Threshold the HSV image to get only blue colors
    mask = cv2.inRange(hsv, lower_blue, upper_blue)

    # Bitwise-AND mask and original image
    res = cv2.bitwise_and(frame,frame, mask= mask)

    cv2.imshow('frame',frame)
    cv2.imshow('mask',mask)
    cv2.imshow('res',res)
    k = cv2.waitKey(5) & 0xFF
    if k == 27:
        break

cv2.destroyAllWindows()
```

Exercice 1:

Proposer un système de détection de mouvement dans un contexte de surveillance d'une zone par une caméra vidéo. Le système doit envoyer une alarme lors de la détection d'une personne dans la zone surveillée. Vous pouvez enregistrer votre propre flux vidéo par votre webcam de votre pc à l'aide de ce code :

```
import cv2

def save_webcam(outPath, fps, mirror=False):
    # Capturing video from webcam:
    cap = cv2.VideoCapture(0)

    currentFrame = 0

    # Get current width of frame
    width = cap.get(cv2.CAP_PROP_FRAME_WIDTH)  # float
    # Get current height of frame
    height = cap.get(cv2.CAP_PROP_FRAME_HEIGHT)  # float

    # Define the codec and create VideoWriter object
    fourcc = cv2.VideoWriter_fourcc(*"XVID")
    out = cv2.VideoWriter(outPath, fourcc, fps, (int(width), int(height
)))

    while (cap.isOpened()):

        # Capture frame-by-frame
        ret, frame = cap.read()

        if ret == True:
            if mirror == True:
                # Mirror the output video frame
                frame = cv2.flip(frame, 1)
            # Saves for video
            out.write(frame)

            # Display the resulting frame
            cv2.imshow('frame', frame)
        else:
            break

        if cv2.waitKey(1) & 0xFF == ord('q'):  # if 'q' is pressed then
 quit
            break

        # To stop duplicate images
        currentFrame += 1

    # When everything done, release the capture
    cap.release()
    out.release()
    cv2.destroyAllWindows()


def main():
    save_webcam('output.avi', 30.0, mirror=True)


if __name__ == '__main__':
    main()
 image = cv2.imread(image_path)
```

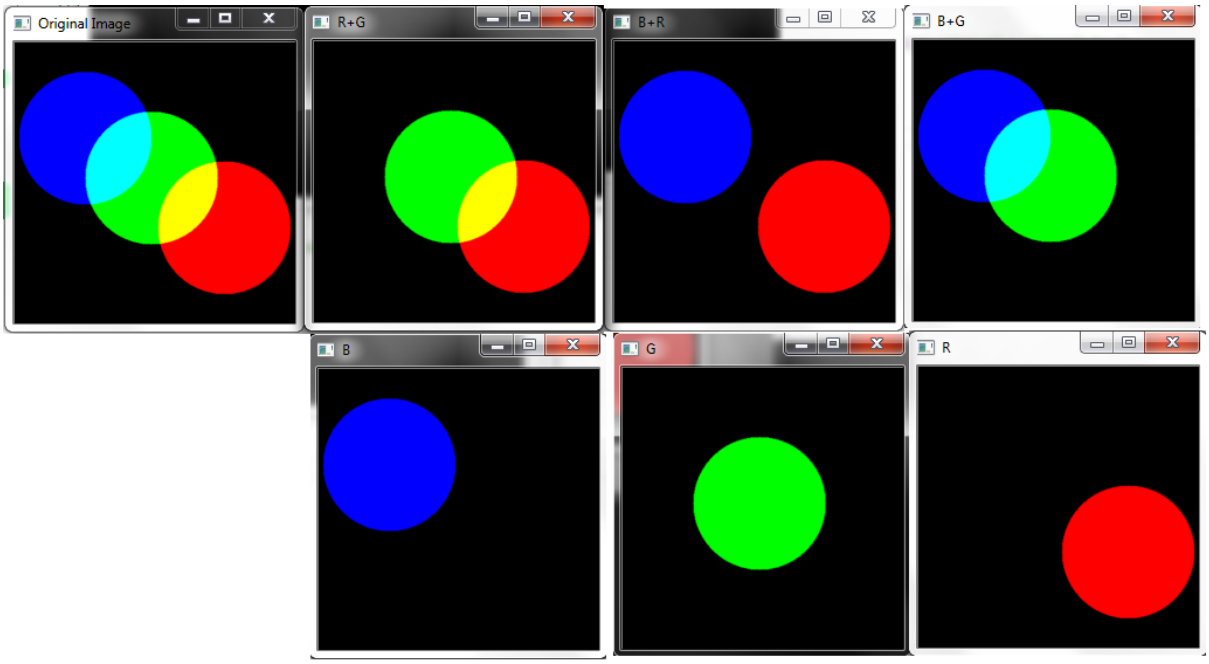
Exercice 2:

Dans un espace BGR, une image est traitée comme un résultat additif des trois couleurs de base (bleu, vert et rouge).

Ecrire un code permettant de lire une image dans l'espace de couleur RGB, faire appel à une procédure pour séparer ses trois canaux et les afficher (voir la figure suivante)

Rappelez-vous qu'une image couleur est constituée de trois canaux et quand on voit chacun d'eux séparément, la sortie correspond à une image en niveaux de gris.

Modifier votre image afin de permettre la séparation des trois ronds dans l'image d'origine (voir la figure suivante)

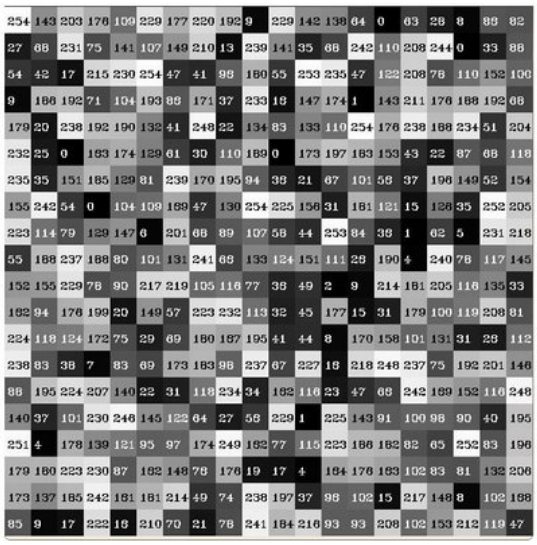


Penser d'abord à les séparer en trois canaux.
Pour qu'un canal soit visualisé en couleur, il faut le recombiner avec les deux autres canaux mais en gardant une valeur fixe pour ces derniers.

F. Calcul d'histogramme

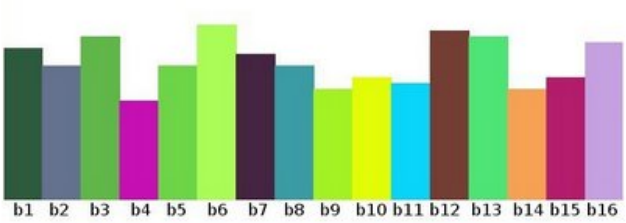
L'histogramme est un mode de représentation graphique de la distribution tonale d'une image. L'histogramme associe à chaque niveau – de 0 pour noir à 255 pour blanc – le nombre de pixels correspondant dans l'image considérée. Le niveau 128 représente le gris moyen, qui se situe donc à mi-chemin entre le noir et le blanc. On peut ainsi dire que l'histogramme représente la distribution des valeurs de pixels dans une image.

Imaginez qu'une matrice contient des informations d'une image (intensité est dans les rangs 0-255):



Puisque la plage de la valeur de l'information pour ce cas est de 256 valeurs, nous pouvons segmenter notre rang dans des sous-parties (bins) et calculer le nombre de pixels qui tombent dans le rang de chaque bin_ {i}

$$[0, 255] = [0, 15] \cup [16, 31] \cup .... \cup [240, 255]$$
$$\text{range} = \text{bin}_1 \cup \text{bin}_2 \cup .... \cup \text{bin}_{n=15}$$



OpenCV et Numpy ont tous deux une fonction intégrée pour calculer l'histogramme d'une image. Avant d'utiliser ces fonctions, nous devons comprendre certaines terminologies liées aux histogrammes.

BINS: L'histogramme ci-dessus indique le nombre de pixels pour chaque valeur de pixel, c'est-à-dire de 0 à 255. Par exemple, vous avez besoin de 256 valeurs pour représenter l'histogramme ci-dessus. Mais si vous devez trouver le nombre de pixels compris entre 0 et 15, puis 16 à 31, ..., 240 à 255, vus n'aurez besoin que de 16 valeurs pour représenter l'histogramme. Donc, ce que vous faites est simplement de diviser tout l'histogramme en 16 sous-parties et la valeur de chaque sous-partie est la somme de tous les nombres de pixels. Cette sous-partie est appelée «BIN». Dans le premier cas, nombre de cases où 256 (un pour chaque pixel) alors que dans le second cas, il est seulement de 16. BINS est représenté par le terme histSize dans les documents d'Opencv.

DIMS: C'est le nombre de paramètres pour lesquels nous collectons les données. Dans ce cas, nous collectons des données concernant la valeur d'intensité.

RANGE: C'est la plage de valeurs d'intensité que vous voulez mesurer. Normalement, c'est [0,256], soit toutes les valeurs d'intensité.

Opencv offre la fonction cv2.calcHist() pour calculer l'histogramme d'une image.

cv2.calcHist(images, channels, mask, histSize, ranges[, hist[, accumulate]])

images : l'image source de type uint8 ou float32.

channels : l'indice de canal pour lequel on calcule l'histogramme. Par exemple, si input est une image en niveaux de gris, sa valeur est [0]. Pour l'image couleur, vous pouvez passer [0], [1] ou [2] pour calculer l'histogramme du canal bleu, vert ou rouge respectivement.

mask : l'image du masque. image du masque. Pour trouver l'histogramme de l'image complète, il est donné comme "None". Mais si vous voulez trouver l'histogramme d'une région particulière de l'image, vous devez créer une image de masque pour cela et lui donner un masque.

histSize : cela représente notre nombre de BIN

Chargez une image en mode niveaux de gris et trouvez son histogramme complet.

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

hist est un tableau 256x1, chaque valeur correspond au nombre de pixels de cette image avec sa valeur de pixel correspondante.

Numpy vous fournit également une fonction, np.histogram( ) :

```
hist,bins = np.histogram(img.ravel(),256,[0,256])
```

La différence ici est que les bins auront 257 éléments, car Numpy calcule les bins comme 0-0.99, 1-1.99, 2-2.99 etc. La plage finale serait donc 255-255.99. Pour représenter cela, ils ajoutent également 256 à la fin des bacs. Mais nous n'en avons pas besoin 256. Jusqu'à 255 est suffisant.

Vous pouvez directement calculer l'histogramme et l'afficher :

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

img = cv2.imread('home.jpg',0)
plt.hist(img.ravel(),256,[0,256]); plt.show()
```

Ou bien avec matplotlib :

```
img = cv2.imread('home.jpg')
color = ('b','g','r')
for i,col in enumerate(color):
    histr = cv2.calcHist([img],[i],None,[256],[0,256])
    plt.plot(histr,color = col)
    plt.xlim([0,256])
plt.show()
```

Si vous voulez trouver des histogrammes de certaines régions d'une image, il suffit de créer une image de masque avec une couleur blanche sur la région que vous souhaitez trouver en histogramme et en noir sinon. Puis passez ceci comme masque.

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

# create a mask
mask = np.zeros(img.shape[:2], np.uint8)
mask[100:300, 100:400] = 255
masked_img = cv2.bitwise_and(img,img,mask = mask)

# Calculate histogram with mask and without mask
# Check third argument for mask
hist_full = cv2.calcHist([img],[0],None,[256],[0,256])
hist_mask = cv2.calcHist([img],[0],mask,[256],[0,256])

plt.subplot(221), plt.imshow(img, 'gray')
plt.subplot(222), plt.imshow(mask,'gray')
plt.subplot(223), plt.imshow(masked_img, 'gray')
plt.subplot(224), plt.plot(hist_full), plt.plot(hist_mask)
```

```
plt.xlim([0,256])

plt.show()
```

Pour comparer deux histogrammes, il faut d'abord choisir une métrique pour exprimer à quel point les deux histogrammes correspondance. La fonction compareHist permet d'effectuer cette comparaison, et dispose également de 4 mesures différentes pour calculer l'appariement.

```
cv.CompareHist(hist1, hist2, method)
où method : cv2.HISTCMP_CORREL, cv2.HISTCMP_CHISQR,
cv2.HISTCMP_INTERSECT ou cv2.HISTCMP_BHATTACHARYYA
```

Exercice 3:

Proposer une méthode qui permet de comparer une image requête à un ensemble d'image de référence afin de trouver la plus proche. Prenez comme image requête : "data/waves.jpg", et comme images de référence l'ensemble d'images : beach.jpg, dog.jpg, polar.jpg, bear.jpg, lake.jpg et moose.jpg

Généralement, la méthode de Template Matching est utilisée pour la détection d'objet dans une image (recherche dans une image img du template en faisant glisser le template sur toute l'image et en calculant une proximité à chaque position). Regarder cet exemple :

```
# template matching

import cv2
import numpy as np

# reading in the image
img_rgb = cv2.imread('image.jpg')
# converting to grayscale
img_gray = cv2.cvtColor(img_rgb, cv2.COLOR_BGR2GRAY)

# reading the template to be matched.
template = cv2.imread('template.jpg',0)
w, h = template.shape[::-1]

# threshold option, where if something is maybe an 80% match, then we s
ay it's a match.
res = cv2.matchTemplate(img_gray,template,cv2.TM_CCOEFF_NORMED)
threshold = 0.8
loc = np.where( res >= threshold)

# marking all the matches on the original image, using the cordinates f
ound in gray image.
for pt in zip(*loc[::-1]):
    cv2.rectangle(img_rgb, pt, (pt[0] + w, pt[1] + h), (0,255,255), 2)
```

```
cv2.imshow('Detected',img_rgb)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Comment l'histogramme peut contribuer à la recherche d'objet dans une image?

Exercice 4:

Appliquer le calcul d'histogramme pour des régions d'intérêt (ou entre une région d'intérêt requête et des régions d'intérêt dans l'image), comparer leurs histogrammes. N'oublier pas de normaliser vos histogrammes avec cv2.normalize().

G. ***Clustering - kmeans***

Kmeans est un algorithme de clustering, dont l'objectif est de partitionner n points de données en k grappes. Chacun des n points de données sera assigné à un cluster avec la moyenne la plus proche. La moyenne de chaque groupe s'appelle «centroïde» ou «centre». Globalement, l'application de k-means donne k grappes distinctes des n points de données d'origine. Les points de données à l'intérieur d'un cluster particulier sont considérés comme «plus similaires» les uns aux autres que les points de données appartenant à d'autres groupes. Cet algorithme peut être appliquer sur des points d'origine géométrique, colorimétriques et autres.

Voyons comment appliquer Kmeans pour trouver les couleurs dominantes dans une image RGB en regroupant ses intensités de pixels. Étant donné une image de taille MxN, nous avons donc MxN pixels, chacun composé de trois composants: r, g et b respectivement, que nous les traitons comme points de données. Les pixels appartenant à un cluster donné seront plus similaires en couleur que les pixels appartenant à un cluster distinct.

D'abord, importer les packages necessaires :

```
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import cv2
import numpy as np
```

Ensuite charger une image et la convertir de BGR à RGB si nécessaire et l'afficher :

```
image = cv2.imread('lena.jpg')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

plt.figure()
plt.axis("off")
plt.imshow(image)
```

Afin de traiter l'image en tant que point de données, il faut la convertir d'une forme matricielle à une forme vectorielle (liste de couleur rgb) avant de la clusterer en n_clusters en fonction de l'intensité de pixels:

```
image = image.reshape((image.shape[0] * image.shape[1], 3))
clt = KMeans(n_clusters = n_clusters )
clt.fit(image)
```

Pour afficher les couleurs les plus dominantes dans l'image, il faut définir deux fonctions : centroid_histogram() pour récupérer le nombre de clusters différents et créer un histogramme basé sur le nombre de pixels affectés à chaque cluster ; et

plot_colors() pour initialiser le graphique à barres représentant la fréquence relative de chacune des couleurs

```
def centroid_histogram(clt):
    numLabels = np.arange(0, len(np.unique(clt.labels_)) + 1)
    (hist, _) = np.histogram(clt.labels_, bins=numLabels)

    # normalize the histogram, such that it sums to one
    hist = hist.astype("float")
    hist /= hist.sum()

    return hist


def plot_colors(hist, centroids):
    bar = np.zeros((50, 300, 3), dtype="uint8")
    startX = 0

    # loop over the percentage of each cluster and the color of
    # each cluster
    for (percent, color) in zip(hist, centroids):
        # plot the relative percentage of each cluster
        endX = startX + (percent * 300)
        cv2.rectangle(bar, (int(startX), 0), (int(endX), 50),
                      color.astype("uint8").tolist(), -1)
        startX = endX

    return bar
```

Il suffit maintenant de construire un histogramme de clusters puis créer une figure représentant le nombre de pixels étiquetés pour chaque couleur.

```
hist = centroid_histogram(clt)
bar = plot_colors(hist, clt.cluster_centers_)

plt.figure()
plt.axis("off")
plt.imshow(bar)
plt.show()
```

Voici le résultat obtenu pour cette image (n_clusters = 5) :

Il existe une fonction d'OpenCV cv2.kmeans() ; regardez les documents sur l'utilisation de la fonction avec les exemple donnés: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_ml/py_kmeans/py_kmeans_opencv/py_kmeans_opencv.html

H. Classification - KNN

Le principe derrière la méthode des plus proches voisins (ou KNN) est de trouver un nombre prédéfini d'échantillons d'apprentissage les plus proches en distance du nouveau point et de prédire l'étiquette à partir de ces derniers.

Afin de réaliser un classifier capable de reconnaitre le label d'une image test, suivez les points ci-dessous :

Télécharger la base d'image CIFAR10 à partir du lien http://www.cs.toronto.edu/~kriz/cifar.html et l'extrait dans un répertoire « data »

```
basedir_data = "./data/"
rel_path = basedir_data + "cifar-10-batches-py/"

Désérialiser les fichiers image afin de permettre l'accès aux données e
t aux labels:

def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo)
    return dict

X = unpickle(rel_path + 'data_batch_1')
img_data = X[b'data']
img_label_orig = img_label = X[b'labels']
img_label = np.array(img_label).reshape(-1, 1)
```

Vérifier:

```
print(img_data)
print('shape', img_data.shape)
```

Vous devriez trouver un tableau numpy de 10000x3072 d'uint8s (le 3072 vient du 3 x 1024). Chaque ligne du tableau stocke une image couleur 32x32 en RGB. L'image est stockée dans l'ordre des lignes principales, de sorte que les 32 premières entrées du tableau correspondent aux valeurs des canaux rouges de la première ligne de l'image. Vérifier les labels :

```
print(img_label)
print('shape', img_label.shape)
```

Nous avons les étiquettes comme matrice 10000 x 1

Charger les données de test. Vous devriez trouver la forme des données de test est identique à la forme des données d'apprentissage.

```
test_X = unpickle(rel_path + 'test_batch');
test_data = test_X[b'data']
test_label = test_X[b'labels']
test_label = np.array(test_label).reshape(-1, 1)
```

Prenez les 5 premières images de data pour voir ce que contient le fichier image

```
sample_img_data = img_data[0:10, :]
print(sample_img_data)
print('shape', sample_img_data.shape)
```

Pour afficher le nom de l'image :

```
batch = unpickle(rel_path + 'batches.meta');
meta = batch[b'label_names']
print(meta)
```

et pour affiche rune image, voici une procédure:

```
from PIL import Image
import numpy as np
from IPython.display import display


def default_label_fn(i, original):
    return original


def show_img(img_arr, label_arr, meta, index, label_fn=default_label_fn
):
    """
        Given a numpy array of image from CIFAR-10 labels this method t
ransform the data so that PIL can read and show
        the image.
        Check here how CIFAR encodes the image http://www.cs.toronto.ed
u/~kriz/cifar.html
    """
    one_img = img_arr[index, :]
    # Assume image size is 32 x 32. First 1024 px is r, next 1024 px is
 g, last 1024 px is b from the (r,g b) channel
    r = one_img[:1024].reshape(32, 32)
    g = one_img[1024:2048].reshape(32, 32)
    b = one_img[2048:].reshape(32, 32)
    rgb = np.dstack([r, g, b])
    img = Image.fromarray(np.array(rgb), 'RGB')
    display(img)
    print(label_fn(index, meta[label_arr[index][0]].decode('utf-8')))
```

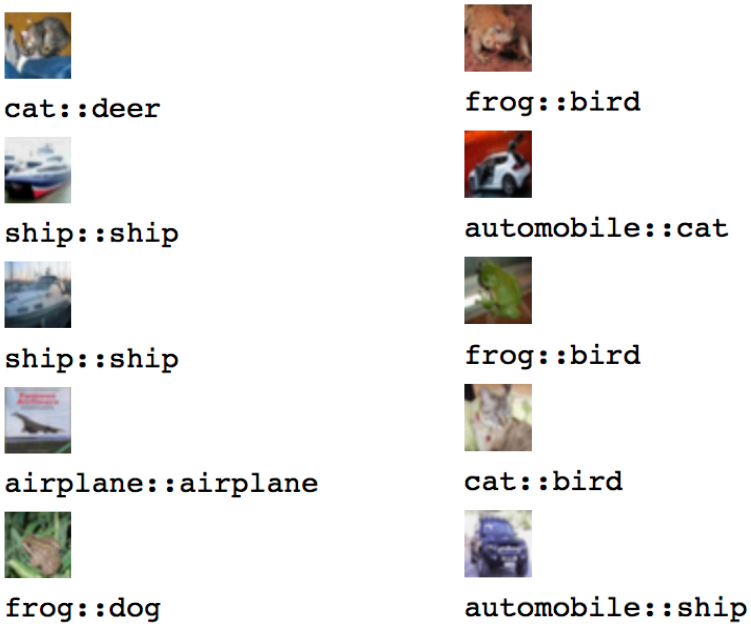
Tester la procedure:

```
for i in range(0, 10):
    show_img(sample_img_data, img_label, meta, i)
```

Pour tester l'algorithme KNN, sélectionnez le nombre d'images de test sur lesquelles vous souhaitez exécuter le modèle.

```
from sklearn.neighbors import KNeighborsClassifier

def pred_label_fn(i, original):
    return original + '::' + meta[YPred_soa[i]].decode('utf-8')


data_point_no = 10
sample_test_data = test_data[:data_point_no, :]

nbrs = KNeighborsClassifier(n_neighbors=3, algorithm='brute').fit(img_d
ata, img_label_orig)
YPred = nbrs.predict(sample_test_data)

for i in range(0, len(YPred)):
    show_img(sample_test_data, test_label, meta, i, label_fn=pred_label
_fn)
```

Vous devriez trouver comme résultat :



Exercice 5: implémentez votre propre algorithme KNN et tester le à la place de `KNeighborsClassifier()` et `predict()`. Utiliser la distance euclidienne pour mesurer la distance entre deux images

Recherche d'image par le contenu visuel (Bag of Visual Words)

Bag of visual words ou Bag of features (BoF) est l'une des approches les plus utilisées pour la classification et la recherche par le contenu des données visuelles. Elle est inspirée d'un concept appelé Sac de mots utilisé dans la classification des documents. Un sac de mots est un simple vecteur de comptage de mots; c'est-à-dire un histogramme éparse sur le vocabulaire. En vision par ordinateur, un paquet de mots visuels d'entités est un simple vecteur de comptage d'occurrences d'un vocabulaire d'entités locales.

L'objectif du TP est de développer un système de recherche d'images par contenu visuel qui soit efficace pour les grandes collections d'images. Classiquement, un système de recherche d'images par contenu visuel comporte une phase hors ligne d'indexation de la base d'images et une phase en ligne de recherche à proprement dit.

Nous commençons par un processus de classification et nous verrons comment le transformer en système de recherche par le contenu (moteur de recherch). Le processus de classification se compose de deux phases principales :

1- Phase d'apprentissage : cette phase consiste à calculer des descripteurs SIFT (Scale-Invariant Feature Transform) à partir d'un ensemble d'images de référence et les stocker sous forme d'une structure de données convenable à réutiliser facilement.

Importer les packages nécessaires :

```
import cv2
import numpy as np
import os
from sklearn.svm import LinearSVC
from sklearn.externals import joblib
from scipy.cluster.vq import *
from sklearn.preprocessing import StandardScaler
from sklearn import preprocessing
```

Télécharger la base d'images sur whippet (dataset-classif.zip).
Fournir le chemin vers la base d'apprentissage dans votre code :

```
train_path = './dataset/train/'
```

Obtenir les labels des classes :

```
training_names = os.listdir(train_path)
```

Parser les dossiers de la base afin d'organiser les labels des classes

```
image_paths = []
image_classes = []
```

```
class_id = 0
for training_name in training_names:
    dir = os.path.join(train_path, training_name)
    class_path = imutils.imlist(dir)
    image_paths += class_path
    image_classes += [class_id] * len(class_path)
    class_id += 1
```

Commencer par l'extraction des points d'intérêts SIFT dans toutes les images. Ensuite, stocker les descripteurs autours de ces points dans une liste.

```
des_list = []
```

Initialise le détecteur SIFT :

```
sift = cv2.xfeatures2d.SIFT_create()

for image_path in image_paths:
    im = cv2.imread(image_path)
    gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
    kp, des = sift.detectAndCompute(gray, None)
    des_list.append((image_path, des))
```

Vous pouvez vérifier les points détectés en les affichant

```
cv2.drawKeypoints(gray,kp,img)
```

Il est possible aussi de voir les points en correspondance entre deux images :

```
# BFMatcher avec les parametres par defauts
bf = cv2.BFMatcher()
match = bf.knnMatch(dess[0], dess[i], k=2)
matches.append(match)
```

Empiler tous les descripteurs dans un tableau numpy

```
descriptors = des_list[0][1]
for image_path, descriptor in des_list[1:]:
    descriptors = np.vstack((descriptors, descriptor))
```

Maintenant, on applique le clustering afin de créer un vocabulaire de (ici 100) mots visuels

```
k = 100
voc, variance = kmeans(descriptors, k, 1)
```

Ensuite, on calcule les histogramme des caractéristiques, en utilisant la quantification vectorielle qui nous permet de créer notre vocabulaire de mots visuels.

```
im_features = np.zeros((len(image_paths), k), "float32")
for i in xrange(len(image_paths)):
    words, distance = vq(des_list[i][1], voc)
    for w in words:
        im_features[i][w] += 1

#transform vector
nbr_occurences = np.sum((im_features>0)*1,axis=0)
idf =
```

```
np.array(np.log((1.0*len(image_paths)+1)/(1.0*nbr_occurences+1)),'float
32')
stdSlr = StandardScaler().fit(im_features)
im_features = stdSlr.transform(im_features)
```

Avec les histogrammes obtenus pour chaque classe, on peut entrainer un classifier (knn, svm, …)

```
kn = 5
#clf = neighbors.KNeighborsClassifier()
clf =
neighbors.KNeighborsClassifier(kn,weights='uniform',p=2,metric='minkows
ki')
# Linear SVM
#clf = LinearSVC()
clf.fit(im_features, np.array(image_classes))
```

On peut enregistrer le modèle issu de l'entrainement, qui contient le classifier, les noms de classe, le vecteur de transformation, le nombre de clusters et le vocabulaire. Cela permet de séparer les deux phases, apprentissage et test.

```
joblib.dump((clf, training_names, stdSlr, k, voc), "bof.pkl",
compress=3)
```

2- Dans la phase de test, il suffit de charger le modèle, calculer les descripteurs SIFT dans les images de test et enfin appliquer la fonction de prédiction pour prédire la classe de chaque image de test.

```
clf, classes_names, stdSlr, k, voc = joblib.load("bof2.pkl")


predictions =  [classes_names[i] for i in clf.predict(test_features)]
```

Maintenant, on vous demande de transformer votre classifier en moteur de recherche qui permet de calculer un score de similarité au lieu de prédire un label. Autrement dit, c'est la recherche par similarité visuelle à partir d'un exemple. Dans sa forme la plus simple, le système reçoit en entrée une image exemple (requête) et retourne l'ensemble des images les plus similaires à cette image, au sens de la mesure de similarité associée à la signature (ici l'histogramme calculé à partir du vocabulaire visuel). La recherche implique de trouver les plus proches voisins de la signature associée à l'image requête.

Dans ce cas, au lieu d'apprendre un modèle d'apprentissage, il faut plutôt utiliser une distance :

```
im_features_r = preprocessing.normalize(im_features_r, norm='l2')
```

Dans la phase de recherche en ligne, on cherche un score (on utilise par exemple la fonction np.dot ne numpy).

```
score = np.dot(test_features, im_features.T)
rank_ID = np.argsort(-score)
```

Utiliser pour cette partie du TP la base dataset-retr.zip