Numpy et les tableaux

INFORMATIQUE COMMUNE - TP n° 1.7 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ importer la bibliothèque Numpy
- ☞ utiliser un tableau Numpy mono et multidimensionnel
- ☞ écrire des opérations élément par élément

A Tableaux numpy

Pour comprendre les différences entre les listes et les vecteurs, on propose de définir deux tableaux numpy à partir de deux listes comme suit :

```
import numpy as np

L1 = [1, 2, 3]
L2 = [4, 5, 6]
L3 = [L1, L2]

v1 = np.array(L1)
v2 = np.array(L2)
m = np.array([L1, L2])
```

A1. Exécuter puis commenter ce qui est renvoyé par les commandes suivantes :

```
print(L1 + L2)
print(v1 + v2)
print(2 * L1)
print(2 * v1)
print(1.5 * L1)
print(1.5 * v1)
print(L1 ** 2)
print(v1 ** 2)
L1.append(7)
v1.append(7)
print(np.sin(v1))
print(L3[1][2])
print(m[1, 2])
print(L3[1,2])
```

Solution :

Code 1 – Tableaux numpy

```
import numpy as np

L1 = [1, 2, 3]
L2 = [4, 5, 6]
L3 = [L1, L2]

v1 = np.array(L1)
v2 = np.array(L2)
m = np.array([L1, L2])

print(L1 + L2)  # List concatetation [1, 2, 3, 4, 5, 6]
print(v1 + v2)  # element by element addition # [5 7 9]
print(2 * L1)  # List demultiplication [1, 2, 3, 1, 2, 3]
print(2 * v1)  # element by element multiplication [2 4 6]
# print(1.5 * L1) # TypeError: can't multiply sequence by non-int of type 'float
    '
print(1.5 * v1)  # element by element multiplication [1.5 3.  4.5]
# print(L1 ** 2) # TypeError: unsupported operand type(s) for ** or pow(): 'list
    ' and 'int'
print(v1 ** 2)  # element by element exponentiation [1 4 9]
L1.append(7)  # append an element to the list
# v1.append(7) # AttributeError: 'numpy.ndarray' object has no attribute 'append
    '
print(np.sin(v1))  # [0.84147098 0.90929743 0.14112001]
print(L3[1][2])  # nested list 6
print(m[1, 2])  # matrice 6
# print(L3[1, 2]) # TypeError: list indices must be integers or slices, not
    tuple
```

B Image et tableaux de pixel

Les images sont représentées par des tableaux de pixels.

Vocabulary 1 — Picture element or Pixel ⇜⇝ élément d'une image (point)

Un pixel peut être encodé de manière différente selon que l'image est couleur ou en niveaux de gris.

a Image couleur

Lorsque l'image est couleur, chaque pixel décompose la lumière par synthèse additive en trois couleurs primaires : rouge, vert et bleu. C'est ce qu'on appelle le système RVB. On peut coder l'intensité de chaque couleur avec une profondeur plus ou moins grande allant de quelques centaines à quelques millions de couleurs. Couramment, on choisit 256 niveaux d'intensité pour une couleur et on la code sur 8 bits, de 0 (le plus sombre) à 255 (le plus clair). Dans ce cas, un pixel couleur est donc un vecteur à dimension 3 contenant 3 entiers non signés sur 8 bits généralement encodés un `uint8`.

Une image couleur `img` est donc un tableau de dimension `(largeur, hauteur, 3)`. L'accès à l'élément `img[i,j]` renvoie un vecteur de taille 3.

b Image en niveaux de gris

Lorsque l'image est en niveaux de gris, chaque pixel n'est cette fois représenté que par un seul entier, codant la nuance de gris entre le noir 0 et le blanc 255. Une image en niveaux de gris est un tableau à deux dimensions d'entiers non signés sur 8 bits, donc une matrice de dimension `(largeur, hauteur)`. L'accès à un élément `img[i,j]` renvoie un scalaire de type `uint8`.

C Manipuler les images

Pour manipuler des images avec Python, on procède comme indiqué sur le code 2.

Code 2 – Manipuler des images en Python

```
import matplotlib.pyplot as plt

img = plt.imread("plage.jpg")
plt.imshow(img)  # for color image
# plt.imshow(img, cmap = plt.cm.gray)}  # for gray level image
plt.show()

print(img)
print(img.shape)
```

C1. Téléchargez l'image plage.jpg et l'enregistrer dans le répertoire de travail (cf. premier TP et module `os`).

D Traitement d'images

D1. Écrire une fonction `gris(p)` prenant en entrée un pixel de couleur (donc un vecteur de taille 3) et renvoyant le niveau de gris correspondant à la moyenne de ses trois composantes RVB, arrondie à l'entier le plus proche. On utilisera les commandes :

- `np.mean(v)` renvoie la moyenne des éléments du vecteur `v` sous forme d'un flottant, compatible avec le type `np.uint8`,
- `round` pour arrondir un flottant en entier,
- `np.uint8()` pour convertir le résultat en non signé entier codé sur 8 bits.

D2. Écrire une fonction `conversion(img)` prenant en entrée une image en couleurs (un tableau à 3 dimensions) et renvoyant une image en niveaux de gris (un tableau à 2 dimensions) construite à partir de celle-ci en prenant pour chaque pixel la moyenne (arrondie) de ses 3 composantes RVB.

Dans toute la suite, on ne manipulera que des images en niveaux de gris, donc des tableaux à 2 dimensions.

D3. Écrire une fonction `inverser(A)` renvoyant l'image inverse de `A`, c'est-à-dire que chaque pixel de niveau de gris `p` est remplacé par celui de niveau `255-p`.

D4. Écrire une fonction `flip(A)` renvoyant l'image obtenue à partir de `A` en appliquant une symétrie d'axe vertical passant par le milieu de l'image.

D5. Écrire une fonction `tourner(A)` renvoyant l'image obtenue à partir de `A` en appliquant une rotation de 180 degrés.

On souhaite maintenant redimensionner une image. On dipose d'une image A de taille (N, M) que l'on veut redimensionner en une image a de taille (n, m). La première méthode est celle d'interpolation au plus proche voisin, définie par la formule :

$$a(i,j) = A\left(\left\lfloor \frac{iN}{n} \right\rfloor, \left\lfloor \frac{jM}{m} \right\rfloor\right),$$

où $\lfloor x \rfloor$ désigne la partie entière de x.

D6. Écrire une fonction `ppvoisin(A,n,m)` prenant en entrée une image A en niveaux de gris et deux entiers n et m et qui renvoie une nouvelle image de taille (n, m) obtenue par redimensionnement de A à l'aide de l'interpolation au plus proche voisin.

La deuxième méthode de redimensionnement est celle de réduction par moyenne locale : on suppose ici que $p_n = N/n$ et $p_m = M/m$ sont des entiers. Chaque pixel de la nouvelle image $a(i, j)$ est égal à la moyenne (arrondie) des $A(k, l)$ pour (k, l) dans un rectangle de côtés (p_n, p_m) et de coin gauche $(i * p_n, j * p_m)$.

D7. Écrire une fonction `moyenne_locale(A,n,m)` réalisant ce redimensionnement. On pourra utiliser une commande de tronçonnage comme `A[i1:i2, j1:j2]` qui extrait la sous-matrice de lignes comprises entre i_1 et $i_2 - 1$ et de colonnes comprises entre j_1 et $j_2 - 1$.

On souhaite maintenant détecter les contours d'une image. Pour cela on remplace chaque pixel $A(i, j)$ par la norme de son gradient discret :

$$\sqrt{(A(i,j) - A(i,j-1))^2 + (A(i,j) - A(i-1,j))^2}$$

sauf pour la première ligne et la colonne où on mettra des 0.

D8. Écrire une fonction `contour(A)` qui réalise cette opération sur l'image A. Pour plus de lisibilité, on pourra renvoyer une image inversée. Pour réaliser la soustraction de deux entiers `x` et `y` codés sur 8 bits : on pourra écrire `x + (-1)*y`, le résultat sera codé sur 32 bits, et on pourra utiliser `np.sqrt` pour la racine carrée.

Solution : Tout l'intérêt de numpy réside dans l'utilisation des opérations éléments par éléments pour éviter les boucles. Comme vous débutez dans la manipulation, on propose deux corrections :

- la première (cf. code 3) explicite la manipulation des tableaux avec des boucles for.
- La seconde (cf. code 4) utilise les opérations éléments par éléments et le tronçonnage et l'inversion de séquences pour accélérer le traitement.

Code 3 – Traitement des images

```
import matplotlib.pyplot as plt
import numpy as np


def gris(p):
    m = np.mean(p)
    return np.uint8(round(m))


def conversion(img):
```

```
    (n, m, p) = img.shape
    A = np.zeros((n, m), dtype=np.uint8)
    for i in range(n):
        for j in range(m):
            A[i, j] = gris(img[i, j])
    return A


def inverser(A):
    (n, m) = A.shape
    B = np.zeros((n, m), dtype=np.uint8)
    for i in range(n):
        for j in range(m):
            B[i, j] = 255 - A[ i, j]
    return B


def flip(A):
    (n, m) = A.shape
    C = np.zeros((n, m), dtype=np.uint8)
    for i in range(n):
        for j in range(m):
            C[i, j] = A[i, m - 1 - j]
    return C


def tourner(A):
    (n, m) = A.shape
    D = np.zeros((n, m), dtype=np.uint8)
    for i in range(n):
        for j in range(m):
            D[i, j] = A[n - 1 - i, m - 1 - j]
    return D


def ppvoisin(A, n, m):
    (N, M) = A.shape
    a = np.zeros((n, m), dtype=np.uint8)
    for i in range(n):
        for j in range(m):
            a[i, j] = A[(i * N) // n, (j * M) // m]
    return a


def moyenneLocale(A, n, m):
    (N, M) = A.shape
    a = np.zeros((n, m), dtype=np.uint8)
    pn, pm = N // n, M // m
    for i in range(n):
        for j in range(m):
            A_ex = A[(i * pn):((i + 1) * pn),
                     (j * pm):((j + 1) * pm)]
            a[i, j] = round(np.mean(A_ex))
    return a
```

```
def contour(A):
    (n, m) = A.shape
    E = np.zeros((n, m), dtype=np.uint8)
    for i in range(1, n):
        for j in range(1, m):
            deriv = np.sqrt(abs(A[i, j] + (-1) * A[i, j - 1]) ** 2 + abs(A[i, j]
                + (-1) * A[i - 1, j]) ** 2)
            E[i, j] = np.uint8(round(deriv))
    return inverser(E)


if __name__ == "__main__":
    img = plt.imread("plage.jpg")
    # plt.imshow(img)  # for color image
    # plt.imshow(img, cmap = plt.cm.gray)}  # for gray level image
    # plt.show()

    print(img)
    print(img.shape)

    A = conversion(img)
    print(A)
    plt.imshow(A, cmap=plt.cm.gray)
    plt.show()

    B = inverser(A)
    plt.imshow(B, cmap=plt.cm.gray)
    plt.show()

    C = flip(A)
    plt.imshow(C, cmap=plt.cm.gray)
    plt.show()

    D = tourner(A)
    plt.imshow(D, cmap=plt.cm.gray)
    plt.show()

    a = ppvoisin(A, 102, 204)
    plt.imshow(a, cmap=plt.cm.gray)
    plt.show()

    b = moyenneLocale(A, 102, 204)
    plt.imshow(b, cmap=plt.cm.gray)
    plt.show()

    E = contour(A)
    plt.imshow(E, cmap=plt.cm.gray)
    plt.show()
```

Code 4 – Traitement accéléré des images avec numpy

```
import matplotlib.pyplot as plt
import numpy as np
```

```python
def gray(img):
    return np.dot(img[..., :3], [0.299, 0.587, 0.114])


def conversion(img):
    return gray(img)


def invert(A):  # gray scale
    return 255 - A


def flip(A):  # gray scale
    return A[:, ::-1]


def rotate(A):  # gray scale
    return A[::-1, :]


def ppvoisin(A, n, m):
    (N, M) = A.shape
    step_n = N // (n - 1)
    step_m = M // (m - 1)
    return A[0:-1:step_n, 0:-1:step_m]



if __name__ == "__main__":
    img = plt.imread("plage.jpg")

    print(img)
    print(img.shape)

    A = conversion(img)
    print(A, A.shape)
    plt.imshow(A, cmap=plt.cm.gray)
    plt.show()

    B = invert(A)
    plt.imshow(B, cmap=plt.cm.gray)
    plt.show()

    C = flip(A)
    plt.imshow(C, cmap=plt.cm.gray)
    plt.show()

    D = rotate(A)
    plt.imshow(D, cmap=plt.cm.gray)
    plt.show()

    a = ppvoisin(A, 322, 429)
    plt.imshow(a, cmap=plt.cm.gray)
```

```
        plt.show()
```