Mathématiques & Informatique - F. Gaunard
http://frederic.gaunard.com
Lycée Voltaire, Paris 11e.

4 Travaux Pratiques : Algorithmes d'apprentissage

Dans une image RGB, chacun des canaux R, G et B a une valeur comprise entre 0 et 255. Il y a donc $256^3 \simeq 16 \times 10^6$ couleurs possibles pour un pixel, et il faut 24 bits pour coder l'ensemble de ces valeurs (car $2^{24} = 256^3$).

Afin de **compresser** une image, on souhaite par exemple n'utiliser que $k = 16$ couleurs différentes, ce qui permettrait de coder la couleur d'un pixel sur 4 bits seulement (car $2^4 = 16$). Cela réduirait théoriquement d'un facteur 6 la quantité de données nécessaire pour coder les pixels de l'image. Cette méthode est notamment utilisée dans certains formats d'images, comme le BMP en mode palette.

La question se pose alors, pour une image donnée, de choisir la palette de 16 couleurs la plus pertinente : c'est un problème de partitionnement de données, que permet de résoudre l'algorithme des k moyennes.

1 Importer et afficher des images

Pour manipuler des images sous `Python`, nous utiliserons les bibliothèques `PIL`, `urllib.request`, `numpy` et `matplotlib.pyplot`.
La bibliothèque `PIL` permet notamment d'ouvrir des fichiers images. La bibliothèque `urllib.request` permet quant à elle de récupérer une image disponible sur Internet à partir de son adresse `URL`.
Si l'on souhaite importer l'image `shiba500.png`, disponible à l'adresse `http://frederic.gaunard.com/2425/shiba500.png`, il suffit d'utiliser les instructions suivantes:

```
import urllib.request
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

# Pour importer l'image shiba500.png :
T = np.array(Image.open(
    urllib.request.urlopen(
        'http://frederic.gaunard.com/2425/shiba500.png'
    )
).convert('RGB'))

# Pour afficher l'image :
plt.imshow(T)
plt.axis('off')
plt.show()
```

La commande `urllib.request.urlopen()` permet d'accéder à l'image à partir de son adresse Internet. La fonction `Image.open()` permet d'ouvrir l'image, puis `np.array()` de la convertir en un tableau `NumPy`.
La fonction `imshow` permet d'afficher l'image. Pour ne pas afficher les axes, on utilise la commande `plt.axis('off')`.

La fonction `Image.open` de la bibliothèque `PIL` permet d'ouvrir une image. En utilisant ensuite la fonction `np.array` de la bibliothèque `numpy`, on obtient un tableau de type `ndarray` contenant les valeurs numériques des pixels.

Pour une image en niveaux de gris, ce tableau possède n lignes et p colonnes. Pour une image en couleur, il est généralement de dimension $n \times p \times c$, où n désigne le nombre de lignes, p le nombre de colonnes et c le nombre de canaux de couleur.

Dans le cas d'une image RGB, on a $c = 3$: les trois informations correspondent aux intensités L_R, L_G et L_B des canaux rouge, vert et bleu. Ces intensités sont généralement codées par des entiers compris entre 0 et 255. Dans le cas où la transparence est prise en compte (format RGBA), on a $c = 4$, la quatrième information étant celle du canal α associé à la transparence (voir la figure 1.1).

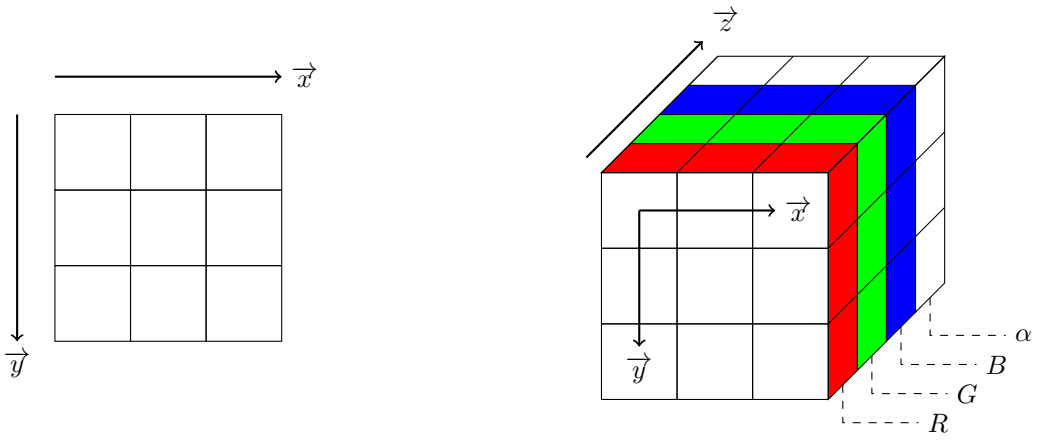

FIGURE 1.1. Représentation d'une image en niveaux de gris (à gauche) et d'une image en couleur avec ses canaux RGB et alpha (à droite).

1. Télécharger le fichier TP4.py sur lequel vous travaillerez tout au long du TP.

2. En utilisant les fonctions `Image.open` de la bibliothèque `PIL` et `np.array` de la bibliothèque `numpy`, importer les images sous forme de tableaux. Vérifier que ce sont bien des images RGB en utilisant la commande `np.shape` de la bibliothèque `numpy`.

2 Nombre de couleurs effectif d'une image

On ne s'intéressera ici qu'aux images de type RGB. Lorsqu'une image RGB est importée avec les fonctions `Image.open` et `np.array`, on obtient un tableau 3D de taille $(n, p, 3)$, de sorte que chacun des $n \times p$ pixels de l'image est défini par des coordonnées $(i, j) \in [\![0, n[\![\times [\![0, p[\![$ et correspond à un triplet (r, g, b) où r, g et b sont des entiers compris entre 0 et 255, représentant respectivement les intensités de rouge, de vert et de bleu du pixel.

3. Écrire une fonction `Egalite(p:ndarray,q:ndarray)->bool` qui teste si deux pixels (c'est-à-dire des tableaux 1D de longueur 3) `p` et `q` sont égaux ou pas.

4. Écrire une fonction `Appartient(p:ndarray,L:[ndarray])->bool` qui détermine si le pixel `p` appartient à la liste `L` contenant des pixels.

5. Écrire une fonction `EgaliteListes(L1:list,L2:list)->bool` qui teste si deux listes contenant des pixels sont égales ou pas.

6. Écrire une fonction `NbCouleurs(img:ndarray)->int` qui calcule le nombre de couleurs différentes utilisées dans une image `img`. Tester la fonction avec l'image `shiba_min.png`. Combien de couleurs sont utilisés dans l'image `prisme_kmystere.png` ?
Quelle est la complexité de votre fonction ?

3 Recherche des barycentres avec l'algorithme des k moyennes

Notre objectif est de réduire ce nombre à seulement k couleurs. Pour ce faire, nous allons appliquer l'algorithme des k moyennes pour regrouper les différents pixels en k classes, calculer la couleur moyenne de chacune de ces classes, puis attribuer cette valeur moyenne à chacun des pixels de la classe correspondante.

7. Écrire une fonction `Distance(p:ndarray,q:ndarray)->float` qui prend pour arguments deux pixels `p` et `q` (représentés par des tableaux 1D de longueur 3) et renvoie la distance euclidienne entre eux.

8. Écrire une fonction `Initialise(img:ndarray,k:int)->[ndarray]` qui prend pour arguments une image `img`, un entier `k` et renvoie une liste de `k` pixels distincts choisis au hasard dans l'image.
On rappelle que `rd.randint(0,n)` renvoie un entier choisi au hasard (c'est-à-dire suivant la loi uniforme) dans l'intervalle $[\![0, n]\!]$.

9. Écrire une fonction `Barycentre(L:[ndarray])->ndarray` qui prend pour argument une liste `L` de pixels, puis renvoie un pixel égal au barycentre des pixels de `L`.

10. Écrire une fonction `PlusProchePixel(p:ndarray,L:[ndarray])->int` qui prend pour argument un pixel `p` et une liste `L` de pixels $[P_0, \dots, P_{k-1}]$, puis qui renvoie l'indice j qui minimise la distance $\|p - P_j\|$.

11. En déduire une fonction `KMoyennes(img:ndarray,k:int)->[ndarray]` qui prend pour arguments une image et un entier `k` puis qui renvoie une liste de k pixels obtenue par l'algorithme des k moyennes, selon le principe rappelé ci-dessous.

L'algorithme des k moyennes consiste à déterminer k pixels $P_1, \dots, P_k$ de la façon suivante :

i. on choisit aléatoirement k pixels $P_1, \dots, P_k$;
ii. chacun des pixels de l'image est associé au pixel P_j dont la couleur est la plus proche, ce qui crée k classes $\mathrm{C}_1, \dots, \mathrm{C}_k$;
iii. on calcule les barycentres $P_1, \dots, P_k$ de ces classes, qui remplacent les valeurs précédentes;
iv. on reprend le calcul à partir de l'étape *ii.*

4 Compression de l'image par réduction de la palette graphique

Une fois la partition obtenue, il reste à calculer la couleur moyenne de chacune des classes et à attribuer cette couleur à chacun des pixels de la classe correspondante. On obtient ainsi une image qui ne comporte plus que k couleurs (ou au plus k couleurs si certaines classes sont vides ou si plusieurs barycentres coïncident).

12. Écrire une fonction `Compression(img,k)->ndarray` qui prend pour argument une image et renvoie une nouvelle image dans laquelle seules k couleurs au maximum sont utilisées. Attention, cette fonction peut être lente...

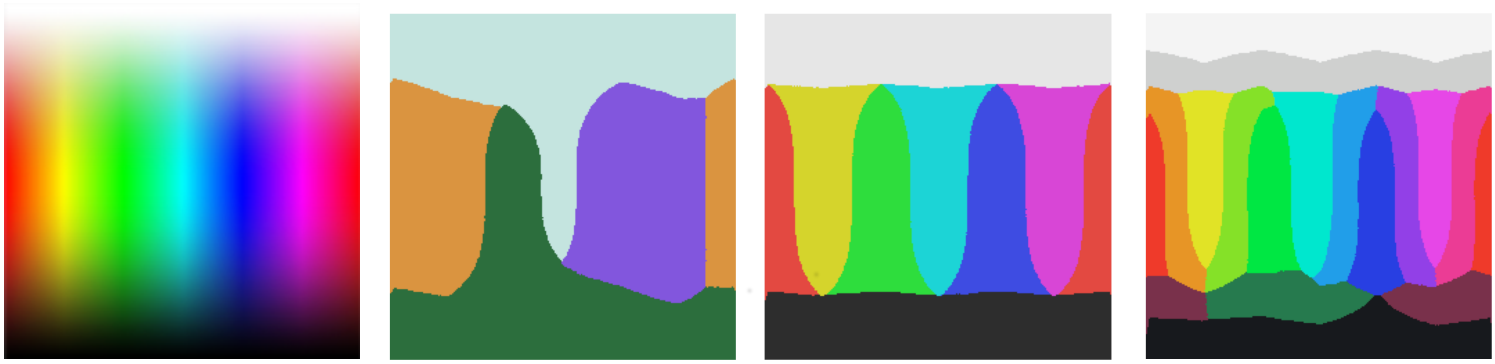

FIGURE 1. APPLICATION DE LA COMPRESSION AVEC $k = 4$, $k = 8$ ET $k = 16$ SUR L'IMAGE prisme.png

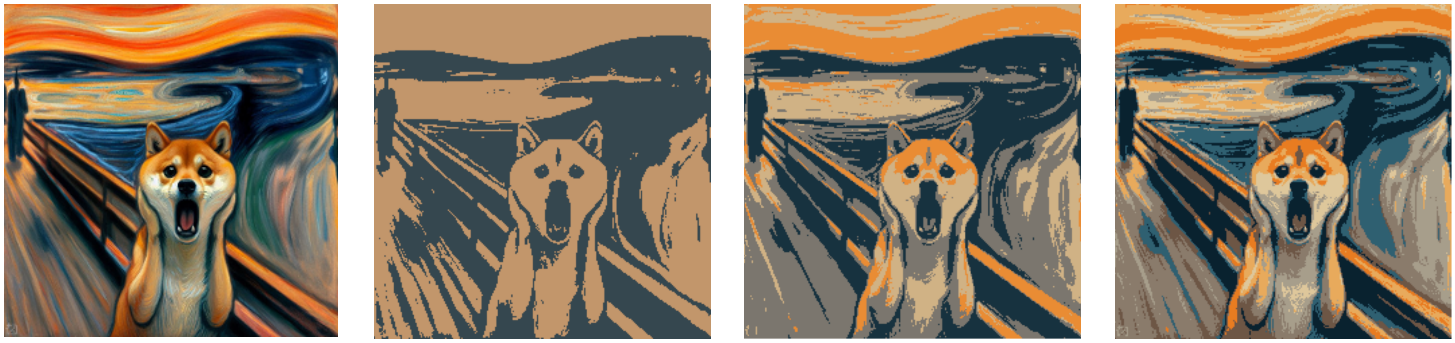

FIGURE 2. APPLICATION DE LA COMPRESSION AVEC $k = 2$, $k = 4$ ET $k = 16$ SUR L'IMAGE shiba1.png