Redimensionnement d'images

Mickaël Péchaud (mickaelpechaud@protonmail.com)

Février 2021

Image originale, redimensionnement naïf, et redimensionnement «intelligent»

Dans ce TP, nous allons implémenter et tester différentes méthodes permettant de **réduire une image** dans le sens de la largeur.

Vous travaillerez dans le fichier *redimensionnement.py* fourni.

- Documentez vos fonctions.
- Testez au fur et à mesure les fonctions que vous écrivez ! Pour ce faire, trois images en niveau de gris sont notamment fournies : *flamantsNB.jpg*, *guepiersNB.jpg* et *rueNB.jpg*.

Toutes les listes dont il sera question dans ce TP sont sauf mention contraire des listes non vides d'entiers.

Une image I est codée sous la forme d'une liste de listes d'entiers, qui peut être vue comme un tableau à deux dimensions. On notera h le nombre de lignes de l'image, et w son nombre de colonnes (respectivement pour **h**eight et **w**idth). Le pixel d'indice $(0, 0)$ correspond par convention au pixel situé en haut à gauche de l'image. Chaque pixel de coordonnées $(i, j) \in [\![0, h-1]\!] \times [\![0, w-1]\!]$ a une valeur entière $I_{i,j}$ comprise entre 0 et 255, correspondant à son niveau de gris (0 pour noir, 255 pour blanc).

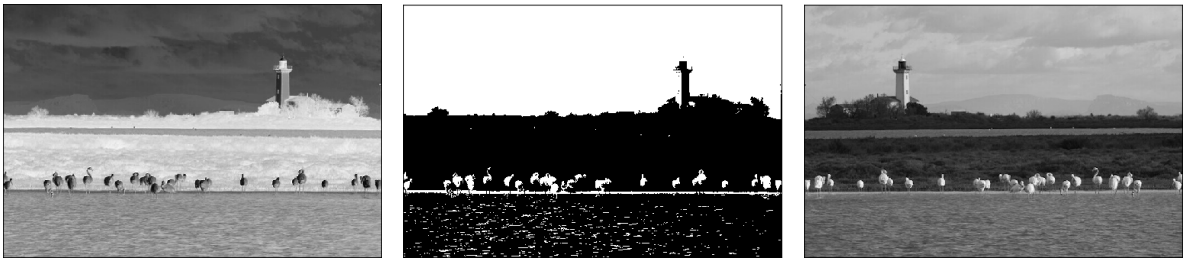
Des fonctions permettant de charger et d'afficher une image vous sont présentées dans le fichier.

Pour se chauffer...

1. Compléter la fonction `dim`, qui prend en argument une image et renvoie le couple d'entiers (w, h), w étant la largeur de l'image, et h sa hauteur (en nombre de pixels).
   On pourra ainsi dans la suite utiliser `(w, h) = dim(img)` pour récupérer les dimensions d'une image.
2. Pouvez-vous afficher les images suivantes, respectivement obtenues à partir de l'image *flamantsNB.jpg* :
   (a) en inversant le contraste ;
   (b) en seuillant les pixels (à la valeur 128) ;
   (c) en effectuant une symétrie horizontale ?

Réduction de largeur naïve

3. Complétez la fonction `reduction_moitie_ligne`, qui prend en argument une liste `l` de longueur paire $2n$ contenant des entiers $a_0, a_1, a_2, \dots a_{2n-1}$, et renvoie la liste de longueur n contenant $(a_0 + a_1)//2$, $(a_2 + a_3)//2$, etc.
4. Complétez la fonction `reduction_moitie_image`, qui prend en argument une image (ayant une largeur paire) et la modifie en appliquant à chaque ligne l'opération décrite dans la question précédente (votre fonction modifiera l'image par effet de bord, et ne renverra rien).
   Quelle est sa complexité en fonction du nombre de lignes h et du nombre de colonnes w de l'image ?
   Testez, et constatez la déformation obtenue.



(a) (b) (c)

> `reduction_moitie_ligne` est de compléxité linéaire en la longueur de la liste (`append` étant de complexité amortie constante).
> Donc `reduction_moitie_image` est de complexité $O(wh)$ – i.e. linéaire en le nombre de pixels de l'image.

5. Réflechissez à une méthode permettant de réduire la largeur d'une image d'un facteur autre que moitié. Il n'est pas demandé d'implémenter la méthode correspondante (mais n'hésitez pas à y revenir à l'issue du TP !).

Nous allons maintenant développer des algorithmes plus «intelligents», dans la mesure où ils prennent en compte le contenu de l'image. On commence par déterminer une mesure d'«importance» des pixels - que nous appellerons *énergie* dans la suite.

Calcul de l'énergie d'un pixel

Si l'on note $I_{i,j}$ le niveau de gris du pixel de coordonnées $(i,j) \in [\![0, h-1]\!] \times [\![0, w-1]\!]$, l'*énergie* d'un pixel i,j intérieur à l'image est définie comme

$$e_{i,j} = \left| \frac{I_{i+1,j} - I_{i-1,j}}{2} \right| + \left| \frac{I_{i,j+1} - I_{i,j-1}}{2} \right|.$$

Cette quantité est d'autant plus petite que le pixel est dans une zone uniforme de l'image, et d'autant plus grande qu'il est dans une zone de forte variation du niveau de gris (par exemple au niveau d'un contour) [1].

6. Complétez la fonction `energie`, qui prend en argument une image, et renvoie une nouvelle image correspondant à son énergie (les pixels n'auront plus nécessairement une valeur entière). On réfléchira en particulier à comment adapter la formule ci-dessus à un pixel situé sur un bord de l'image.
Quelle est la complexité de votre fonction ?
Testez, et affichez l'image des énergies obtenue.

> La complexité est $O(wh)$.

Image originale, image des énergies

1. Il s'agit en fait d'une version discrétisée de $\|\nabla I\|_1 = \left|\frac{\partial I}{\partial x}\right| + \left|\frac{\partial I}{\partial y}\right|$, qui est une norme du gradient de l'image.

Réduction ligne par ligne

Maintenant que nous disposons d'une image et de son énergie, pour réduire la largeur d'une image d'un pixel, nous allons simplement enlever un pixel d'énergie minimale dans chaque ligne.
Nous avons pour cela besoin de quelques fonctions élémentaires de manipulations de listes.

7. Complétez la fonction `enlever`, qui prend en argument une liste `l` et un indice `i` compris entre 0 et la longueur `l` moins 1, et renvoie une nouvelle liste correspondant à `l` dont l'élément d'indice `i` a été supprimé. On pourra utiliser des extractions de tranches et concaténations de listes.
8. Complétez la fonction `indice_min`, qui prend en argument une liste `l` et renvoie un indice auquel apparaît la valeur minimale de la liste.
9. Complétez la fonction `reduction_par_ligne`, qui prend en arguments une image `img`, et la modifie en supprimant un pixel d'énergie minimale dans chacune de ses lignes.
10. Complétez la fonction `itere_reduction_par_ligne`, qui prend en arguments une image `img` et un entier $n > 0$, applique n fois la procédure décrite dans la question précédente à `img`.
    Quelle est sa complexité ?
    La tester. (On pourra par exemple réduire de 200 pixels la largeur de l'image *flamantsNB.jpg*.)

> `enlever` et `indice_min` sont de complexité linéaire (attention aux coûts cachés des extractions de tranches et concaténations dans `enlever` !).
> `reduction_par_ligne` est donc de complexité $O(wh)$, et `itere_reduction_par_ligne` de complexité $O(nwh)$.

Le résultat est visuellement peu satisfaisant.

11. Expliquez qualitativement les distortions observées.

> Les pixels enlevés peuvent être très éloignés entre une ligne et la suivante, créant des décalages de parties de lignes et donc des distortions importantes.

Réduction par colonne

Pour y remédier, on souhaite lors d'une itération enlever uniquement des pixels situés sur une même colonne.

12. Complétez la fonction `meilleure_colonne`, qui prend en argument une image d'énergies `e`, et renvoie un indice de colonne d'énergie minimale (l'énergie d'une colonne étant définie comme la somme des énergies de ses pixels).
13. Complétez la fonction `reduction_meilleure_colonne`, qui prend en argument une image, et en supprime une colonne d'énergie minimale.
14. En déduire une fonction `itere_reduction_meilleure_colonne`, qui applique la méthode décrite ci dessus pour réduire l'image `img` de `n` pixels dans sa largeur.
    Quelle est sa complexité ?

> `meilleure_colonne` est de complexité $O(wh)$. `energie` également, et `enlever` est de complexité linéaire en la longueur de la liste qui lui est passée en argument. `append` étant de complexité amortie constante en la taille de la liste, `reduction_meilleure_colonne` est de complexité $O(wh)$. `itere_reduction_meilleure_colonne` est donc de complexité $O(nwh)$.

15. Testez, et expliquez qualitativement pourquoi les résultats obtenus sont «bons» sur les images *guepiersNB.jpg* et *flamantsNB.jpg*, mais pas sur *rueNB.jpg*.

> Les deux premières images ont des colonnes «globalement intéressantes» (celles où il y a un cheval, un flamant, le phare...), et d'autres «globalement inintéressantes», car ne traversant que du paysage. Ça n'est pas le cas dans la troisième, où les colonnes croisent toutes une voiture ou un autre détail, rendant la supression d'une colonne génante visuellement.

Seam carving

L'idée de l'algorithme de *Seam carving*, introduit en 2007 par S.Avidan et A.Shamir est d'assouplir un peu la contrainte de réduction colonne par colonne.
On définit un *chemin* de pixels comme une suite de pixels connectés (verticalement ou en diagonale) dont le premier appartient au bord haut de l'image, le dernier au bord bas, et contenant exactement un pixel par ligne de l'image.
L'énergie d'un chemin est la somme des énergies des pixels le constituant.

Voici par exemple un chemin d'énergie 6 dans une image des énergies jouet e de 4×4 pixels.

1	**1**	0	3
4	1	**2**	4
1	2	**2**	1
4	**1**	1	0

Pour réduire la largeur d'une image d'un pixel, on souhaite trouver puis enlever un chemin d'énergie minimale.
On définit un *chemin partiel* comme un chemin, sans la contrainte que son dernier pixel atteigne le bord bas de l'image. Afin de calculer un chemin minimal, on va commencer par calculer, pour chaque pixel, l'énergie minimale E d'un chemin partiel terminant sur ce pixel.
Par exemple, pour notre image des énergies e, on obtient le tableau E suivant :

1	1	0	3
5	1	2	4
2	3	3	3
6	3	4	3

16. Calculer à la main E pour l'image des énergies e suivante, puis trouver un chemin d'énergie minimale.

2	1	1	0
3	3	2	2
2	0	1	2

> On obtient le tableau et le chemin suivants :
>
> | | | | |
> |---|---|---|---|
> | 2 | 1 | 1 | **0** |
> | 4 | 4 | **2** | 2 |
> | 6 | **2** | 3 | 4 |

17. Expliquez comment construire E à partir de e, en procédant ligne par ligne.

> La première ligne de E est celle de e. Ensuite, $E(i,j)$ est la somme de $e(i,j)$ et du minimum des $E(i-1,j')$ aux points $(i-1,j')$ voisins de (i,j).

18. Une fois E construit, comment en déduire un chemin d'énergie minimale ?

> On cherche un point de valeur minimale sur la dernière ligne de E. Puis un point de valeur minimale parmi ses voisins sur la ligne précédente. On poursuit en remontant ligne par ligne, et l'on obtient ainsi un chemin d'énergie minimale.

19. Implémentez tout cela. Il sera judicieux d'écrire plusieurs fonctions, et vous pourrez utiliser la fonction `min` permettant de calculer le minimum des éléments d'une liste (de complexité linéaire en la longueur de la liste). Enfin, complétez la fonction `seam_carving`, qui prend en arguments une image `img` et un entier `n`, et modifie l'image en enlevant successivement n chemins minimaux.
Quelle est sa complexité ?
Testez !

Annexe

Références

- L'algorithme de *Seam carving* est introduit dans l'article *Seam Carving for Content-Aware Image Resizing* (Shai Avidan, Ariel Shamir, 2007).
- L'algorithme ainsi que quelques extensions sont démontrés sur la page suivante : http://mpechaud.fr/scripts/traitementimages/seamcarving.html.

Fonctions et méthodes sur les listes

Voici les fonctions et méthodes sur les listes dont l'usage est autorisé – ainsi que les complexités que vous utiliserez pour les calculs de complexité (en fonction de la longueur de la liste n). Tout autre fonction dont vous auriez besoin doit être implémentée !

Opération	Exemple	Complexité
Accès direct	`l[0]`	$O(1)$
Longueur	`len(l)`	$O(1)$
Concaténation	`l1+l2`	$O(n1+n2)$
Ajout en fin de liste	`l.append(1)`	$O(1)$
Suppression en fin de liste	`l.pop()`	$O(1)$
Extraction de tranche	`l[1 :10]`	$O(n)$, où n est la longueur de la tranche.
Répétition	`[0]*k`	$O(n)$, où n est la longueur de la liste créée.
Création par compréhension	`[k**2 for k in range(n)]`	$O(n)$ si l'expression est évaluée en temps constant

Chargement et affichage d'image

Ce TP utilise les modules `matplotlib.image` et `matplotlib.pyplot` pour le chargement et l'affichage d'images.

L'instruction suivante permet de charger une image sous forme de liste de listes – comme indiqué en introduction. Elle nécessite que `monimage.jpg` soit dans le répertoire de travail.

Code Python :

```
image = (mpimg.imread('monimage.jpg')).tolist()
```

Les instructions suivantes permettent l'affichage d'une image en niveaux de gris.

Code Python :

```
plt.imshow(image, cmap='gray', clim=(0,255))
plt.show()
```

En cas de besoins plus importants, il existe des modules dédiés très puissants pour le traitement d'images. Notamment :

- *pillow* (https://python-pillow.org/)
- *opencv* (https://pypi.org/project/opencv-python/)