Projet n°1
Compression d'image à travers la factorisation SVD

Groupe n°1 - Équipe n°3

Responsable : oulo
Secrétaire : gdouezangrard
Codeurs : gcollioucloren, adoghmi, sbalafrej

Résumé : Ce projet consiste à mettre en place un mécanisme basé sur une décomposition SVD pour compresser des images. Les images peuvent êtres présentées sous forme d'une matrice, que nous allons transformer en matrice diagonale. Cette matrice sera privée de ses termes les plus faibles selon un indice de compression. Et nous récupèrerons alors l'image compressée.

1 Transformations de Householder

La matrice d'une symétrie hyperplane peut s'écrire sous la forme :

$$H = Id - 2 * U * {}^tU$$

Ici, U va correspondre au vecteur directeur de l'hyperplan de symétrie qui envoie X sur Y. Ainsi, il va être colinéaire (si X et Y ne sont eux-mêmes pas colinéaires, c'est à dire dans notre cas, si $X \neq Y$) à $X - Y$.

Afin de conserver la norme, il convient de prendre ce vecteur de façon à ce qu'il soit unitaire, c'est à dire, de choisir :

$$U = \frac{X - Y}{||X - Y||}$$

Une fois le vecteur U choisi, il ne reste plus qu'à appliquer la formule, en faisant attention au cas $X = Y$.

Une fois cette matrice obtenue, pour appliquer la méthode, il ne nous reste qu'à multiplier la matrice de Householder par le vecteur (ou la matrice) que l'on souhaite modifier.

Cependant, dans le cas où l'on souhaite multiplier une matrice, appliquer nos deux algorithmes de calcul de la matrice, puis de multiplication matricielle n'est pas optimisé. En effet, cette dernière a une complexité en $\Theta(n^3)$ (en considérant une matrice $n \times n$).

Afin d'obtenir un algorithme moins couteux, nous pouvons exécuter la multiplication de la matrice pendant le calcul de la matrice de Householder, c'est à dire réaliser l'opération : $M = Id - 2 * U({}^tU * A)$,où A est la matrice de départ, et M est la matrice résultat. (Dans le cas d'une multiplcation à gauche). Ici nous n'effectuons que des produits vectoriels, qui sont de complexités $\Theta(n^2)$, ce qui vas résulter en un algorithme en $\Theta(n^2)$.

Comme ici nous manipulons des images, représentés par de grandes matrices, nous obtenons un gain non négligeable.

2 Mise sous forme bidiagonale

Le but de cette partie est de transformer une matrice donnée A de taille $n \times m$ en une matrice bidiagonale (BD) en multipliant à droite et à gauche par deux matrices orthogonales de changement de base ($Qleft$ et $Qright$).
L'idée est de remplacer la colonne i de notre matrice de départ par un vecteur de même norme comportant que des valeurs nulles sauf à la *ieme* ligne, et de remplacer la *ieme* ligne par la transposée d'un vecteur de même norme, de valeurs nulles partout sauf à la $i$ème et $(i+1)$ème position. Ces deux vecteurs satisfont la condition pour génèrer deux matrices de Householder $H_{1,i}$ et $H_{2,i}$ que l'on multiplie à chaque tour de boucle à droite de la matrice $QLeft$ (respectivement à gauche de $Qright$).
Finalement on a :

$$Qleft = H_{1,n} * H_{1,n-1} * \ldots * I_n$$
$$Qright = H_{2,1} * H_{2,2} * \ldots * I_m$$
$$A = Qleft * BD * Qright$$

Au niveau de la complexité, on remarque que l'algorithme utilise une boucle de longueur n et effectue à chaque itération (i) deux produits matriciels et une génération de matrice de Householder de taille $n-i$. L'utilisation des fonctions de multiplication d'une matrice de Householder par un ensemble de vecteurs nous a permis d'optimiser le coût de la multiplication matricielle à $\Theta(n^2)$. Au final on se retrouve avec une complexité en $\Theta(n^3)$.

3 Transformation QR et décomposition SVD

Avant de se tourner vers la compression d'image il est nécessaire d'avoir une matrice diagonale, pour cela, on applique un certain nombre de fois (paramétrable) la décomposition QR sur la matrice bidiagonale.

3.1 Algorithme naif

La fonction `numpy.linalg.qr` permet d'effectuer une décomposition QR et d'obtenir les matrices Q et R. La complexité de cet algorithme est $\Theta(n^2)$ donc celui de la décomposition SVD est $\Theta(n^3)$

3.2 Algorithme optimisé

L'algorithme précédent n'est pas de complexité minimale. Nous essayerons donc de trouver un algorithme de complexité minimale :

Démontrons tout d'abord que S, R_1 et R_2 sont toujours bidiagonales

Preuve

Sachant que S R_1, R_2 sont bidiagonales, qu'elles résultent d'une série de transformations QR, donc démontrer que S, R_1 et R_2 sont toujours bidiagonales revient à démontrer que la décomposition QR d'une matrice bidiagonale est bidiagonale.

On procède donc par récurrence sur n, la taille de la matrice.

Vérification : pour $n = 2$

Soit A une matrice de taille $2 * 2$ bidiagonale inférieure. La décomposition QR de la matrice donne une matrice R triangulaire supérieure de taille $2 * 2$ donc bidiagonale supérieure.

Hérédité

Soit $n \in \mathbb{N}$

On suppose que pour toute matrice de taille n bidiagonale inférieure, R est bidiagonale supérieure.

Soit A une matrice de taille $n + 1$.

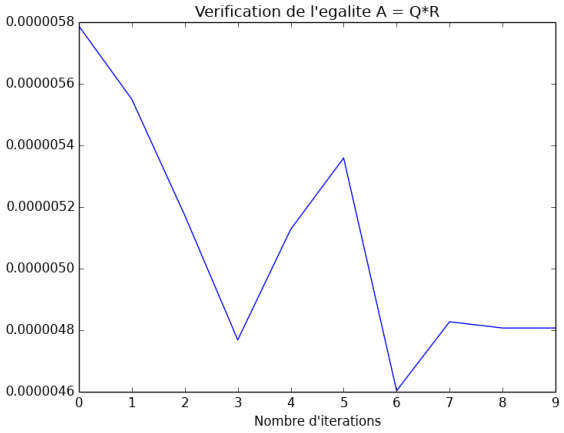
Soit B la matrice de taille n extraite de A en retirant la première ligne et colonne. Donc la décomposition QR de B donne $B = Q1 * R1$ avec R_1 bidiagonale supérieure (d'après l'hypothèse de récurrence).

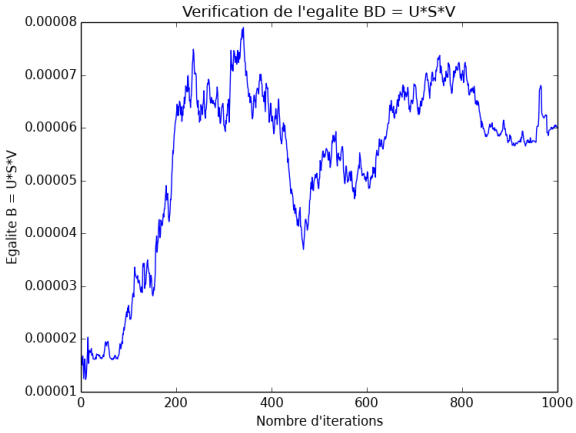
Soit maintenant une matrice Q, avec $Q_{11} = 1$ et $Q_{1j} = Q_{i1} = 0,\ \forall i, j \in [2, n]$ et $Q_{i+1,j+1} = Q_{1ij},\ \forall i, j \in [1, n]$.

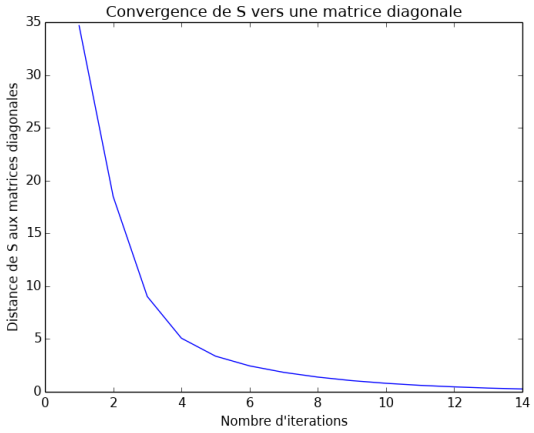
Finalement $Q * A$ est bidiagonale et donc R est bidiagonale supérieure.

Cet invariant va nous permettre de ré-écrire la factorisation QR pour obtenir une décomposition SVD plus efficace, en ne considérant plus des colonnes et des lignes entières de la matrice mais seulement un sous-matrice de taille 2×2.

La version optimisée de la décomposition SVD se fera en $\Theta(n^2)$ et non en $\Theta(n^3)$

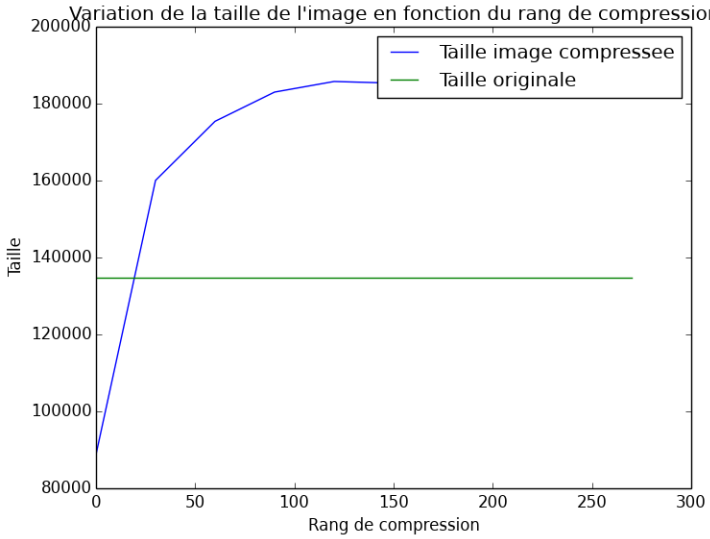




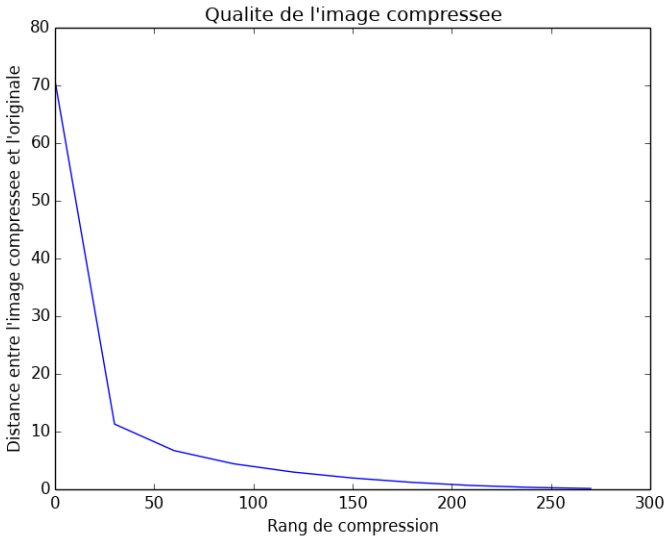


4 Application à la compression d'image

Il est donc temps de lier toutes les opérations ci-dessus et de les appliquer à la compression d'image. D'une part, il nous faut récuperer les images sous un format que l'on puisse exploiter, et dans notre cas, ce sera sous forme de matrice. Chaque case est un triplet représentant la couleur RGB (nous l'extrayons donc), et nous devons pour chacune de ces matrices extraites appliquer l'algorithme de bidiagonisation d'une part, puis l'algorithme de décomposition SVD. Nous obtenons donc une matrice $M = Qleft \times U \times S \times V \times Qright$. Nous assurons une bonne propriété sur la matrice diagonale S : les valeurs sur sa diagonales sont positives et décroissantes. Ceci nous permet de déterminer la contribution de chacune de ces valeurs dans l'image finale (puiqu'il s'agit d'un simple changement de base).



Pour effectuer l'opération de compression à un rang k donné, il nous suffit de commencer par supprimer les contributions les plus faibles, ainsi l'image sera dégradée mais sera toujours visionable. Bien évidemment, plus l'on augmente la compression, plus la distance entre l'image compressée et l'image originale augmente comme le montre le graphe suivant.



La compression n'a d'intérêt que si l'on gagne de la place, mais on se rend compte que pour des valeurs supérieures à environ 1/10 fois la taille de l'image pour le rang de compression, l'image générée devient plus lourde que l'image originale ! La conséquence sur la matrice de l'image est l'annulation de certaines composantes sur des lignes ou des colonnes, ce qui donne d'ailleurs cet aspect un peu hachuré, ou au contraire, l'introduction par les erreurs de calcul de composantes qui n'y étaient pas.

Enfin on peut obtenir une idée de la distance entre l'image d'origine et l'image compressée (de façon mathématique) en fonction du rang de compression (ceci a été réalisé avec la fonction `np.svd` pour des résultats plus intéressants).

Figure 1 – Image originale puis compressée à un rang 50

Conclusion

Nous avons donc vu un processus de compression, qui consiste à transformer une matrice correspondant à une image par changement de base en une autre sur laquelle on peut accéder à un certain ordre de termes par leur importance dans l'image finale, et donc supprimer leur contribution. Ceci permet de compresser l'image. Mais nous avons aussi pu constater certains travers de cette compression, avec par exemple des algorithmes assez coûteux qu'il faut optimiser, et puis un travail avec des matrice de tailles importantes puisque nous avons à faire à des images. Enfin nos algorithmes ont tendance à propager les erreurs, et donc le résultat où l'on espère une image compressée est en réalité une image plus lourde.