PROJET 3: Compression d'image à travers la factorisation SVD

Coordinateur: Ilyes Bechoual
Secrétaire: Alexandre Tabouret
Programmeurs: Ethan Mocellin, Ramy Jeribi

24 octobre 2023

Département Informatique
IS104 - Algorithme numérique
S6-Année 2022-2023

Introduction

La décomposition en valeurs singulières (ou singular value decomposition en anglais, abrégée SVD) est une technique de compression d'image bien utile utilisant de nombreux procédés issus de l'algèbre linaire. Ce projet montre sa réalisation sous python.

1 Transformations de Householder

Matrices de Householder

La transformation de Householder est une symétrie hyperplane $H_{U,V}$ permettant d'envoyer un vecteur U sur un vecteur V (ces deux vecteurs étant de même norme). L'obtention de H passe par celle du vecteur N définie par $H_{U,V} = I_n - 2 * N * N^t$. Or H envoie U sur V. Il vient alors :

$$H_{U,V} * U = V \iff U - 2 * N * (U * N)^t = V \iff U - V = 2 * N * (U * N)^t$$

Si $U = V$ ou $(U*N)^t = 0$ alors $H = Id$ fonctionne. Sinon N est colinéaire à $U-V$ et donc $N = \pm\frac{U-V}{||U-V||}$.

Optimisation du produit matriciel

Cette optimisation repose sur le fait que multiplier un vecteur par un autre vecteur est moins coûteux que de multiplier un vecteur par une matrice. Ainsi, pour calculer le produit $H * X$, il est donc possible de réaliser l'opération :

$$X - 2 * N * (N^t * X)$$

Le produit $(N^t * X)$ est de complexité $O(n)$ (produit de deux vecteurs) au lieu de $O(n^2)$ pour un produit matrice-vecteur usuel.
Pour calculer le produit d'une matrice de Householder une matrice M les colonnes de la matrice sont considérées comme étant des vecteurs auxquels l'opération précédente est itérée. Finalement la fonction *Produit_Householder_Matrice()* est obtenue et elle a une complexité temporelle de $O(n^2)$ au lieu de $O(n^3)$ pour un produit matrice-matrice usuel.

2 Mise sous forme bidiagonale

La première étape de la SVD consiste à bidiagonaliser l'image à compresser.
La bidiagonalisation est un procédé visant à transformer une matrice A de taille $n * m$ est matrice bidiagonale BD de même taille. Soit : $A = Q_{left} * BD * Q_{right}$ où Q_{left} de taille $n * n$ et Q_{right} de taille $m * m$ sont des matrices de changement de base.
Le sujet fournit un algorithme de bidiagonalisation utilisant des matrices de Householder, dont l'obtention a été réalisé précédemment. Il consiste à placer le bon nombre de zéros sur la première colonne et la première ligne en déduisant les bonnes matrices de changement de base à l'aide des matrices de Householder, puis à itérer sur les colonnes et lignes suivantes.
Ainsi, par exemple, la bidiagonalisation de la matrice A ci-dessous est :

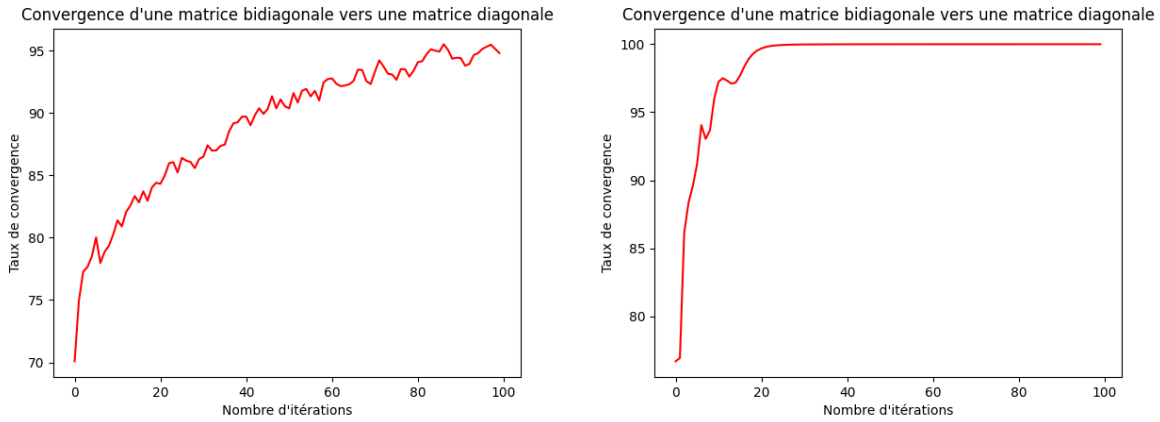
$$A = \begin{pmatrix} 89 & 84 & 6 \\ 11 & 97 & 85 \\ 62 & 96 & 34 \\ 96 & 80 & 38 \\ 96 & 5 & 63 \end{pmatrix} = \begin{pmatrix} 0.5111 & -0.0117 & 0.5235 & 0.5788 & 0.3599 \\ 0.0632 & 0.8706 & -0.3824 & 0.3028 & 0.0078 \\ 0.3561 & 0.3443 & 0.2528 & -0.7538 & 0.35 \\ 0.5513 & 0.0379 & 0.1055 & -0.0694 & -0.8238 \\ 0.5513 & -0.3492 & -0.7103 & -0.0151 & 0.2632 \end{pmatrix} * \begin{pmatrix} 174.1206 & 150.7968 & 0 \\ 0 & 134.5438 & 3.4002 \\ 0 & 0 & 71.2933 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0.8628 & 0.5055 \\ 0 & 0.5055 & -0.8628 \end{pmatrix}$$

Remarque. *L'algorithme de bidiagonalisation est relativement coûteux. En effet, il nécessite de nombreux produits matriciels, que ce soit pour calculer les matrices de Householder ou Q_{left} et Q_{right}. Cependant, cela reste bien moins coûteux que de déterminer les valeurs propres de la matrice à transformer, d'où l'utilisation de cette transformation.*

Remarque. *La bidiagonalisation peut être en théorie optimisée en intégrant à l'algorithme le produit matriciel optimisé de la partie précédente. Cependant, le module* Numpy *utilisé optimise déjà les calculs matriciels et notamment bien mieux que l'optimisation proposée (le module est des dizaines de fois plus rapide). Ainsi, le produit matriciel optimisé écrit précédemment n'a pas été retenu.*

3 Transformations QR

Dans le cadre de la SVD, il est nécessaire de transformer la matrice bidiagonale BD précédemment obtenue en matrice diagonale. Pour ce faire, plusieurs décompositions QR sont appliquées sur la matrice BD, le but étant de faire converger BD vers un matrice diagonale S. Une boucle *Pour* est donc utilisée et le nombre d'itérations est fixé.
Une première implémentation de ce procédé est réalisé à l'aide la fonction *numpy.linalg.qr*. La figure 1a montre la convergence d'une matrice bidiagonale de taille $100 * 100$ vers une matrice diagonale sur 100 itérations. La matrice bidiagonale tend effectivement vers une matrice diagonale. La vitesse de convergence dépend grandement de la taille de la matrice. En effet, cette matrice peine à atteindre une matrice diagonale en 100 itérations tandis qu'une matrice d'ordre 10 devient diagonale en environ 25 itérations comme observé sur la figure 1b. Cette transformation permet ainsi l'obtention de la décomposition $BD = U * S * V$ (où U et V sont des matrices de changement de base).



(a) Matrice d'ordre 100

(b) Matrice d'ordre 10

FIGURE 1 – Convergence de matrices bidiagonales vers des matrices diagonales

Remarque. *Le taux de convergence correspond au rapport de la norme de la matrice bidiagonale BD transformée (et donc presque diagonale) et une matrice diagonale de même diagonale que BD (le tout multiplié par 100).*

Remarque. *Lorsque la matrice bidiagonale a été transformé en une matrice considérée comme suffisamment proche d'une matrice diagonale, on peut supprimer les termes extradiagonaux pour empêcher des calculs* négligeables.

Optimisation pour une matrice bidiagonale

L'optimisation repose sur le fait de remplacer *np.linalg.qr()* par une autre fonction. La méthode de Gram-Schmidt est un candidat idéal.
L'algorithme de Gram-Schmidt est une méthode permettant l'obtention d'une base orthogonale à partir d'une base quelconque d'un espace vectoriel. Ici, les colonnes de la matrice à décomposer sont considérés comme le dit espace vectoriel. Cela signifie que pour une matrice donnée, l'algorithme permet de construire une nouvelle matrice où chaque colonne est orthogonal à tous les autres vecteurs de la base, et chaque vecteur est également normalisé (c'est-à-dire qu'il a une norme de 1).
Ainsi, à chaque colonne est appliqué le processus suivant :

$$u_n = (a_n - \sum_{i=0}^{n-1} \langle a_n, u_i \rangle \, u_i)/(\|u_n\|)$$

avec a_i les colonnes de la matrice M et u_i les colonnes de la matrice orthogonale U.
À ce stade, une vérification permet de s'assurer que les colonnes de la matrice U sont bien orthogonales

les unes avec les autres et qu'elles ont toutes une norme de 1.
L'équation précédente peut être réécrite sous forme matricielle comme ci-dessous :

$$\begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix} = \begin{pmatrix} u_1 & u_2 & \dots & u_n \end{pmatrix} \begin{pmatrix} \|a_1\| & \langle a_2,u_1\rangle & \langle a_3,u_1\rangle & \dots & \langle a_n,u_1\rangle \\ 0 & \|a_2\| & \langle a_3,u_2\rangle & \dots & \langle a_n,u_2\rangle \\ 0 & 0 & \|a_3\| & \dots & \langle a_n,u_3\rangle \\ & & & \ddots & \vdots \\ 0 & 0 & 0 & \|a_{n-1}\| & \langle a_{n-1},u_{n-1}\rangle \\ 0 & 0 & 0 & 0 & \|a_n\| \end{pmatrix}$$

$$\text{Donc}, M = UR$$

la fonction *gram_shmidt_SVD()* utilise la fonction *QR_Decomposition()* qui est de complexité $O(n^2)$. Donc la compléxité totale de la fonction est $O(n^2 * N_{max})$ où N_{max} est le nombre de colonnes sur les quelles est réalisée la SVD.
Usuellement, la décomposition SVD demande à ce que les éléments de la matrice S soient positifs et ordonnés de manière décroissante. Pour cela la fonction *organisation()* est implémentée. Elle parcourt les valeurs de la diagonale $S[i][i]$ et les multiplie par -1 celles négatives. De plus, elle ordonne les valeurs de la diagonale et échange parallèlement les colonnes de la matrice U. (si S[j][j]>S[i][i] , S[i][i] prend la place de S[j][j] et U[i] prend la place de U[j]). Cette fonction a une complexité de $O(n^2)$

4 Application à la compression d'image

L'application de la SVD se fait en pratique sur la compression d'image. Pour ce faire, une photo d'un arbre face à un coucher de soleil a été prise, et pour limiter le temps d'attente lors du traitement, une taille de 200×200 a été sélectionnée. Le point négatif (ou positif selon ce qu'on désire) est que de ce fait, l'image reste reconnaissable même avec un rang 5 (cf 3c).
Le cheminement effectué pour réaliser cette compression SVD s'est fait en cinq étapes principales décrites ci-dessous :

1. Séparation des canaux RGB
2. Bidiagonalisation de la matrice-image sur les trois canaux
3. Application de l'algorithme de Gram-Schmidt
4. Recomposition de la matrice-image avec les matrices de changement de base et Q_{left}/Q_{right}
5. Fusion des canaux en un seul canal RGB

Il est important de faire remarquer que cette compression SVD pèse en complexité temporelle. En effet, d'après nos tests, lors de la compression au rang 50, le procédé de Gram-Schmidt a pris la quasi-totalité du temps nécessaire à la compression. Cette technique est en effet lourde, mais elle permet des fichiers au final plus légers en espace disque.
De ce fait, le gain en espace a été évalué à environ 87%. Ce gain est conséquent, surtout quand on sait qu'à partir d'un certain rang, le gain en espace devient négatif, et donc l'image "compressée" avec la SVD devient plus lourde que l'originale. Pour avoir une idée de comment la qualité de l'image se détériore au fil du rang, le graphique 2 nous en donne un aperçu quantitatif. Il est possible d'y apercevoir qu'à partir du rang 5, la distance diminue moins vite que précédemment. Cependant, pour ce qui est qualitatif, il dépend de chaque personne et de son usage de prendre le rang qui correspond au mieux à ses attentes.

Conclusion

La SVD est une technique de compression particulièrement efficace. En effet, elle permet de réduire drastiquement la taille d'images tout en conservant suffisamment d'informations. Cependant, la version implémentée lors de ce projet souffre d'une complexité temporelle importante, ce qui a rendu certaines observations difficiles.

Commentaire

Bien que les parties aient été connectées, certaines étaient plus difficiles que d'autres. La partie 2 a été particulièrement problématique en raison d'un manque de précision, ce qui a retardé l'avancement du projet. La partie 3 était assez simple à réaliser, bien qu'elle comportait quelques subtilités, tandis que

la partie 4 a été la plus longue et nous a demandé beaucoup de temps. Dans l'ensemble, la répartition des tâches était assez équitable et le projet était cohérent avec le cours, ce qui nous a donné une base solide pour réfléchir et comprendre.

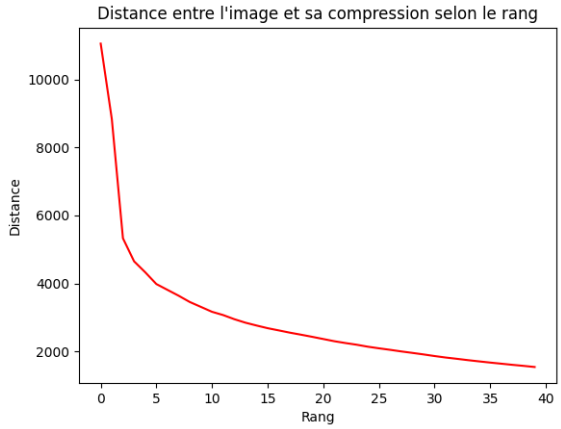


FIGURE 2 – Distance entre l'image et sa version compressée en fonction du rang

(a) Image originale

(b) Image compressée au rang 50

(c) Image compressée au rang 5

FIGURE 3 – Compressions de rang 5 et 50 d'une image d'arbre de résolution $200 * 200$

$$\int_{10}^{13} 2 * x \, dx$$