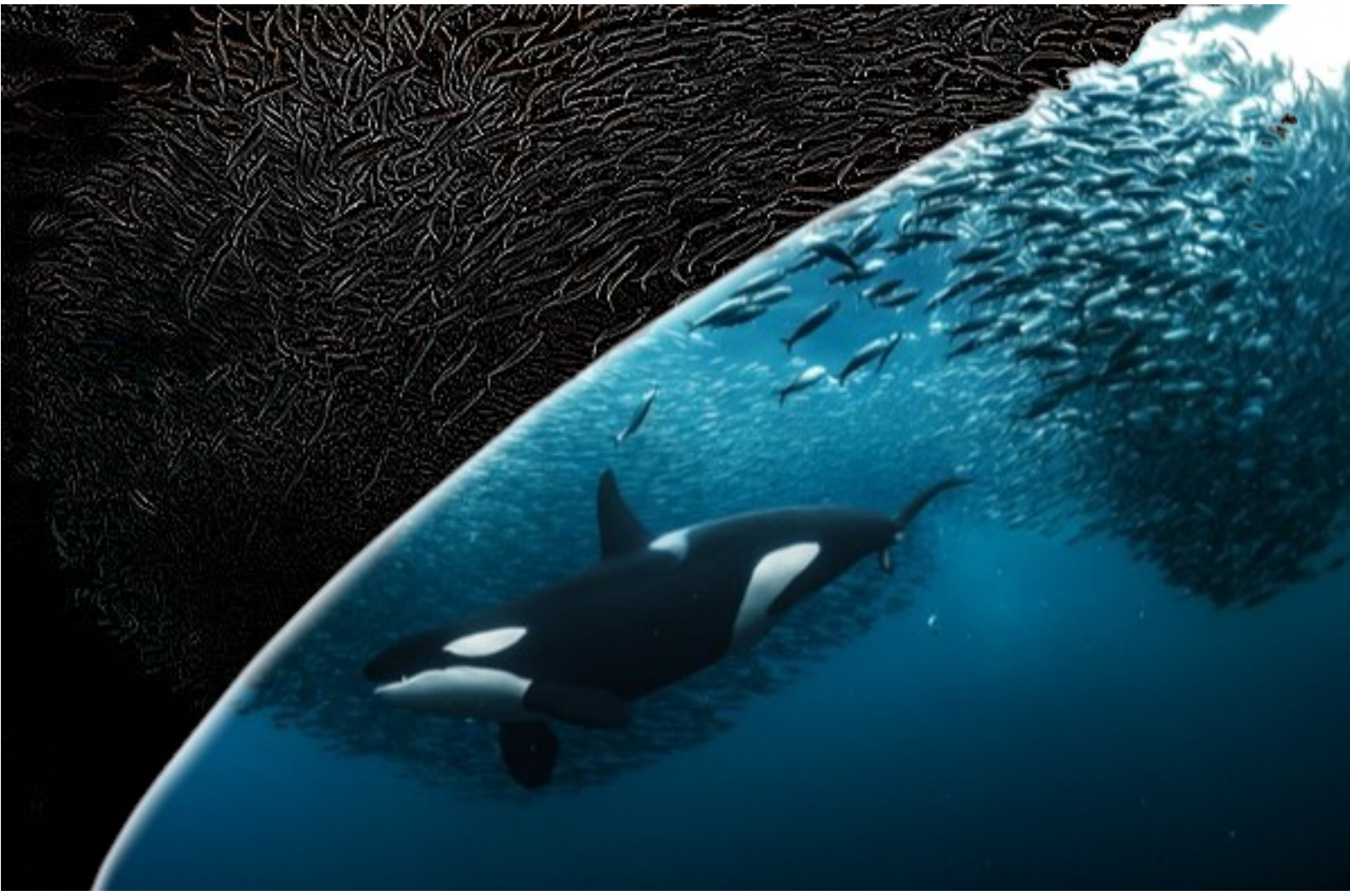
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Projet scientifique encadré
STPI/PSE/2024-007

Traitement d'images : application de la convolution de matrices



Étudiants :

Thomas BAVOUX
Léa DANIEL
Virgil DUCROCQ
Candice BROYER
Ryann DERA-SCHTYK
Paul LOZACH

Enseignant-responsable du projet :
Alexis LECOMTE

Date de remise du rapport : 15/06/2024

Référence du projet : STPI/PSE/2024 – 007

Intitulé du projet : Traitement d'images : applications de la convolution de matrices

Type de projet : *modélisation*

Objectifs du projet :

Notre projet portant sur le traitement d'images à l'aide de la convolution de matrices, notre objectif principal était d'en réaliser une implémentation informatique. Notre but était aussi de créer une interface graphique. Ensuite, nous souhaitions trouver la provenance de différents filtres de convolution déjà existants, à savoir : le flou, la détection et le renforcement des bords, l'augmentation du contraste ainsi que le repoussage. Enfin, un autre but était d'approfondir les recherches sur la convolution. Plus précisément, nous voulions trouver d'autres utilisations de la convolution pour le traitement d'images.

Mots-clefs du projet :

— Convolution
— Matrices
— Images
— Implémentation

Table des matières

Notations et Acronymes

Notation à définir :

Acronymes à définir :

CNN : Réseau de neuronne convolutif (Convolutional neural network)
FFT : Transformation de Fourier rapide (Fast Fourier Transformation)
GIMP : Programme GNU de Manipulation d'Images (GNU Image Manipulation Program)
IA : Intelligence Artificielle
POO : Programation Orientée Objet

Introduction

Dans le cadre de notre projet scientifique encadré, nous avons étudié la convolution de matrices appliquée au traitement d'images. Ce sujet est particulièrement intéressant car, avec l'essor actuel de l'intelligence artificielle, des IA comme Midjourney ou Sora repoussent les limites du possible dans le domaine du traitement d'images. La convolution de matrices, outil mathématique, est au cœur des résultats impressionnants obtenus par ces IA. Nos recherches sur le sujet s'inscrivent donc dans ce contexte de révolution technologique.

Au cours de ce projet, nous avons compris l'aspect théorique de la convolution de matrices appliquée au traitement d'images. Nous avons vu quelles étaient les matrices de convolution utilisées et comment elles étaient obtenues. De plus, nous nous sommes intéressés aux applications de la convolution. Enfin, nous avons implémenté des algorithmes de convolution pour réaliser nous-mêmes nos propres traitements d'images.

Ainsi, notre projet vise à répondre à la question suivante : "Comment la convolution de matrices permet-elle le traitement d'images ?"

Méthodologie, organisation du travail

1.1 Description de l'organisation adoptée

1.1.1 Répartition en sous-groupes

Lors de la première séance consacrée au projet, nous avons tout d'abord pris du temps pour apprendre à nous connaître et échanger sur nos connaissances concernant la convolution de matrices. Puis, nous nous sommes réunis avec M. Lecomte, notre encadrant de projet, pour aborder le sujet de l'organisation. Notre professeur référent nous a soumis l'idée de nous diviser en sous-groupes de deux ou trois personnes afin de se répartir plus facilement les tâches. Suite à un accord commun pour la divison en trois groupes de deux personnes, M. Lecomte nous a fait une rapide présentation des enjeux de notre projet. Après avoir établi les tâches à réaliser, nous avons décidé qu'un groupe s'occuperait de l'implémentation informatique du sujet, un autre groupe chercherait et expliquerait les différents noyaux de convolution, tandis que le dernier groupe effectuerait des recherches sur la convolution en général ainsi que ses applications.

1.1.2 Communication

Pour pouvoir partager à tout moment nos résultats et notre travail, souvent nécessaires à l'avancement des différents groupes, nous avons créé un dossier partagé dans Google Drive, ainsi qu'un serveur de discussion Discord.

À chaque début de séance consacrée au projet, nous nous rassemblions. Nous faisions alors un point sur le travail effectué par chacun d'entre nous en présence de M. Lecomte.

Un organigrame, présent en annexe, permet de visualiser les tâches réalisées par chaque membre du projet.

Travail réalisé et résultats

2.1 Traitement d'images par convolution de matrices

2.1.1 Différents types d'images

Tout d'abord, il est important de rappeler qu'il existe deux types d'images : les images vectorielles et les images matricielles.

Les images vectorielles sont des illustrations basées sur un ensemble de formes géométriques. Ces dernières sont créées à partir d'équations mathématiques suivant différents paramètres donnés à des vecteurs.

Les images matricielles, aussi appelées bitmap, sont des images constituées de carrés numériques, appelés pixels, représentant une matrice. Pour les images en noir et blanc, chaque valeur de pixels varie de 0 à 255 où le 0 représente du noir, le 255 du blanc, et les valeurs intermédiaires représentent différentes teintes de gris. En revanche, pour les images en couleur, le pixel est défini par trois valeurs allant chacune de 0 à 255 : une pour chaque couleur (rouge, vert et bleu). En assemblant les différentes nuances de ses trois couleurs, il est possible d'obtenir toutes les couleurs souhaitées dans différentes intensités.

L'avantage principal de l'image vectorielle est sa capacité à s'étendre à l'infini sans jamais perdre sa qualité : elle ne deviendra jamais pixélisée. C'est pourquoi elle est très utilisée dans le milieu de l'art, comme les dessins animés, pour garantir une grande qualité des détails. Les images vectorielles sont notamment plus économiques sur l'espace mémoire que les images matricielles. En effet, dans une image vectorielle, un cercle de couleur sera défini par une ligne de code définissant ses paramètres, c'est-à-dire sa position, sa couleur et son rayon. Par contre, dans une image matricielle, il sera défini par de nombreux pixels. Il est donc préférable d'utiliser l'application vectorielle pour les images comportant des formes géométriques ou destinées à l'impression en grand format. Au contraire, il est mieux d'utiliser des images matricielles pour les photographies et les illustrations comportant des dégradés. Le produit de convolution pour le traitement d'images est applicable uniquement pour celles qui sont matricielles. Il faut donc convertir les images vectorielles en matricielles pour ensuite effectuer le produit de convolution des matrices. Cette conversion s'appelle la rastérisation et le changement inverse est nommé vectorisation. De nombreux logiciels permettent ces conversions complexes.

2.1.2 Fonctionnement de la convolution de matrices

Une convolution de matrices est un outil mathématique permettant de traiter une matrice d'entrée par une autre matrice appelé noyau qui est de taille carrée. Le signe de l'opération de la convolution est noté $*$. Cela peut se résumer sous la forme ci-contre : "matrice d'entrée $*$ motif = matrice de sortie".
La matrice de sortie est de même taille que la matrice d'entrée.
Les valeurs des éléments de la matrice de sortie dépendent de la taille du noyau. Le noyau est très souvent de taille 3×3 dans le traitement d'images, mais il peut être de n'importe quelle taille $m \times q$ avec m et q impairs pour pouvoir centrer le noyau sur les éléments de la matrice d'entrée.

Matrice d'entrée à n lignes et p colonnes :

$$A = \begin{bmatrix} a_{1,1} & \dots & a_{1,p} \\ \dots & \dots & \dots \\ a_{n,1} & \dots & a_{n,p} \end{bmatrix}$$

Motif de taille 3×3 :

$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & k \end{bmatrix}$$

Dans la convolution de matrices, il est nécéssaire de retourner le noyau M, c'est-à-dire faire une rotation de 180° des coefficients autour de la valeur centrale. Le noyau retourné est noté $\tilde{M}$:

$$\tilde{M} = \begin{bmatrix} k & h & g \\ f & e & d \\ c & b & a \end{bmatrix}$$

Avec le produit de convolution de A par M, on obtient une matrice de sortie S de taille $n \times p$:

$$S = \begin{bmatrix} s_{1,1} & \dots & s_{1,p} \\ \dots & \dots & \dots \\ s_{n,1} & \dots & s_{n,p} \end{bmatrix}$$

Pour calculer les coefficients $s_{i,j}$, il suffit de considérer la matrice A' de taille 3×3 (même taille que le noyau M) issue de A et centrée en $a_{i,j}$:

$$A' = \begin{bmatrix} a_{i-1,j-1} & a_{i,j-1} & a_{i+1,j-1} \\ a_{i-1,j} & a_{i,j} & a_{i+1,j} \\ a_{i-1,j+1} & a_{i,j+1} & a_{i+1,j+1} \end{bmatrix}$$

On notera pour simplifier les calculs :

$$A' = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix}$$

Ainsi pour chaque emplacement :

$$s_{i,j} = a_1 \times k + a_2 \times h + a_3 \times g + a_4 \times f + a_5 \times e + a_6 \times d + a_7 \times c + a_8 \times b + a_9 \times a$$

Le calcul consiste à faire la somme des produits des éléments aux mêmes emplacements entre la matrice A' et $\tilde{M}$.

Pour les contours, si il n'y a aucun nombre affecté aux élements autour de $a_{i,j}$, alors ils sont considérés comme nuls.

2.1.3 Différents types de noyaux

Nous avons vu comment la convolution de matrices fonctionnait, nous allons maintenant voir les noyaux de convolution les plus communs pour le traitement d'images. Des images convoluées à partir de ces noyaux sont disponibles en annexe.

2.1.3.1 Flou

Ce premier noyau est le plus simple :

$$\frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

En effet, il s'agit tout simplement d'une matrice 3×3 remplie de 1 et divisée par 9. Après convolution, la nouvelle valeur d'un pixel est égale à la moyenne non pondérée du pixel modifié et des 8 pixels qui l'entourent. Une fois la convolution effectuée sur toute l'image, les différences de valeur d'un pixel à un autre vont être fortement réduites, sans pour autant changer la luminosité globale de l'image car la somme des valeurs du noyau est égale à 1. On comprend donc que c'est le "lissage" des valeurs des pixels, cette perte d'information, qui crée cet effet de flou.

2.1.3.2 Augmentation du contraste

L'augmentation du contraste peut être considérée comme l'inverse du flou. On dit que c'est un filtre passe-haut. En effet, on souhaite, pour ce type de filtre, garder ou augmenter les grandes différences entre les valeurs des pixels à proximité. On pourrait donc se dire que pour avoir un filtre passe-haut, il suffirait de retirer le filtre floutant au noyau identité (celui qui n'applique aucune transformation). On obtiendrait donc un résultat comme tel :

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} -1/9 & -1/9 & -1/9 \\ -1/9 & 8/9 & -1/9 \\ -1/9 & -1/9 & -1/9 \end{bmatrix}$$

Or cela n'est pas du tout ce que l'on veut. La moyenne des valeurs du noyau obtenu est de 0, ce qui veut dire qu'en théorie on obtiendrait une image qui a une moyenne de luminosité égale à 0. Augmenter le contraste ne veut pas dire perdre toute la lumière de notre image, on voudrait même garder la luminosité originale. Il nous faudrait donc, pour obtenir le filtre que l'on souhaite, avoir une moyenne de noyau égale à 1. On obtient ainsi la matrice :

$$\frac{1}{9} \times \begin{bmatrix} -1 & -1 & -1 \\ -1 & 17 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Notons qu'on peut régler le niveau de contraste souhaité en réalisant la combinaison linéaire suivante :

$$a \times \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - b \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Par exemple, le logiciel de traitement d'images GIMP utilise la matrice :

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 5 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Le contraste est ici pondéré de façon à l'augmenter davantage par rapport à ce que l'on a calculé.

2.1.3.3 Détection des bords

Le noyau de détection de bords utilisé par le logiciel GIMP est le suivant :

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

La détection de contours consiste à chercher les fortes variations continues, c'est-à-dire les fortes variations proches entre elles, de l'image. La détection de ces variations correspond à l'opération de dérivation. Mais comme l'image est un objet en plusieurs dimensions, on utilise le gradient (vecteur composé des dérivées premières de f selon x et y) :

$$\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$$

Le logiciel GIMP calcule le gradient en réalisant les opérations suivantes :

$$\text{selon x : } \frac{\partial f}{\partial x} = f * \begin{bmatrix} 0 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & 0 & 0 \end{bmatrix} \text{ et selon y : } \frac{\partial f}{\partial x} = f * \begin{bmatrix} 0 & -1 & 0 \\ 0 & 2 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

Il faut noter que l'opération de dérivation est réalisée en faisant une simple différence entre 2 pixels. Cela est dû au fait que notre image n'est pas un objet continu au sens mathématique. L'opération de dérivation est définie comme tel : $f'(x_0) = \lim_{h \to 0} \frac{f(x_0+h)-f(x_0)}{h}$. Or la valeur minimale de h sur notre image est 1 (la distance entre 2 pixels) donc l'opération de dérivation pour notre image est une simple différence entre 2 pixels. L'avantage de cette méthode est que l'on peut regrouper ces 2 calculs en un seul avec la matrice indiquée précédemment.

$$\nabla f = f * \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Ce noyau est le regroupement de 4 noyaux calculant le gradient :

le nord : $\begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$, le sud : $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix}$, l'ouest $\begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ et l'est : $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{bmatrix}$.

Rassembler le calcul de gradient en 4 directions en un seul noyau a l'avantage de pouvoir détecter davantage les bords continus. Cependant, l'inconvénient de ce noyau est qu'il ne peut pas calculer les gradients diagonaux. Il faudrait donc appliquer un noyau plus adapté si nécessaire pour notre traitement d'images. Ce noyau n'est évidemment pas la seule méthode pour calculer le gradient : le filtre de Sobel, que l'on présentera plus tard, emploie par exemple un autre noyau.

Un autre exemple de noyau est décrit en annexe.

2.2 Applications de la convolution de matrices pour le traitement d'images

2.2.1 Détection de contours grâce au filtre de Canny

2.2.1.1 Flou Gaussien

Ce noyau, permettant de flouter, est différent de celui expliqué précédement. Sa spécificité et son utilité sont expliquées en annexe.

2.2.1.2 Filtre de Sobel et détection de contours

Présentation du filtre de Sobel

Le filtre de Sobel est un des opérateurs "simples" de détection de contours en traitement d'images. Il permet, par l'utilisation du gradient et de la convolution de matrices, de détecter les contours horizontaux et verticaux séparément afin d'obtenir une version de notre image où ne figurent que les contours des objets. Dans le cadre d'un filtre de Canny, il est utilisé en deuxième étape après un lissage de l'image pour éliminer le bruit.

Gradient

On assimile ici notre image à une fonction à deux variables, on peut alors grapher l'intensité lumineuse de chaque pixel en fonction des variables x et y, coordonnées, de notre fonction que l'on nommera Z.
On se trouve ici dans un cas discret, des approximations du gradient sont donc nécessaires. On utilisera, en considérant l'image comme un tableau T :

$$\frac{\partial}{\partial x}Z(x,y) \approx (T[x+1,y] - T[x-1,y])$$

$$\frac{\partial}{\partial y}Z(x,y) \approx (T[x,y+1] - T[x,y-1])$$

On discrétise ainsi la dérivée par une simple soustraction des valeurs d'intensité lumineuse entre le pixel suivant et le pixel précédent. Nous possédons donc maintenant notre outil qui nous permettra de repérer les brusques variations (les bordures d'objets généralement). Il faut maintenant l'appliquer à notre image, et ce, à l'aide de la convolution de matrices. Les pentes selon un axe peuvent être obtenues en calculant les produits de convolutions $T * C$ (vers les x positifs, en considérant l'origine du repère en haut à gauche de notre image, l'axe des x vers le bas et l'axe y horizontal vers la droite), avec :

$$C = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

On remarque ici que (x, y) désigne le pixel du milieu pour obtenir notre équation précédente. Pour ce qui est de l'autre axe (en direction des y positifs donc) on transpose simplement cette matrice pour obtenir :

$$D = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}$$

On peut verifier :

$$T*C = \begin{bmatrix} x+1,y+1 & x+1,y & x+1,y-1 \\ x,y+1 & x,y & x,y-1 \\ x-1,y+1 & x-1,y & x-1,y-1 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} = T[x+1,y]-T[x-1,y] = \frac{\partial}{\partial x}Z(x,y)$$

Ces noyaux sont fonctionnels, ils pourraient être utilisés dans un algorithme, cependant ils sont très sensibles au bruit (pixels aberrants, avec une intensité très faible ou très forte) car ils ne prennent en compte qu'un pixel à la fois. Pour palier à ce problème le noyau utilisé prendra en compte cette moyenne des écarts sur trois pixels, on a donc :

$$C_2 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \quad D_2 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

Finalement, pour obtenir les noyaux du filtre de Sobel, que l'on nommera G_x et G_y, on pondère en ajoutant un coefficient 2 au pixel central afin de lui donner plus d'importance. Ainsi :

$$\frac{\partial}{\partial x}Z(x,y) \approx (T[x+1,y+1]-T[x-1,y+1])+2\times(T[x+1,y]-T[x-1,y])+(T[x+1,y-1]-T[x-1,y-1])$$

$$\frac{\partial}{\partial y}Z(x,y) \approx (T[x+1,y+1]-T[x+1,y-1])+2\times(T[x,y+1]-T[x,y-1])+(T[x-1,y+1]-T[x-1,y-1])$$

Ce qui correspond aux noyaux :

$$G_x = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

2.2.1.3 Suppression des non-maximums

Pour revenir dans un contexte d'utilisation du filtre de Canny, après l'application de nos noyaux (donc du filtre de Sobel), une suppression des non-maximums permet de nettoyer l'image en suprimmant le bruit restant et en réduisant l'épaisseur des bordures. Pour cela, on compare l'amplitude du gradient de chaque pixel avec celle de ses voisins dans la même direction. Si la valeur n'est pas maximale par rapport aux autres alors le pixel est mis à zéro.

2.2.1.4 Seuillage par hystérésis

Cette dernière étape permet de contrer une détection trop ou trop peu selective, avec donc trop ou trop peu de contours. On définit pour cela un seuil haut et un seuil bas que l'on notera respectivement S_h et S_b on applique ensuite un algorithme simple à l'image obtenue par le filtre de Sobel : si la valeur du pixel est inferieur à Sb on juge qu'il ne s'agit pas d'un contour et la valeur est mise à 0, si elle est superieure à S_h il s'agit d'un contour. Il reste ensuite les pixels compris entre S_b et S_h pour lesquels on regardera si les pixels voisins sont des contours auquel cas ils seront gardés et sinon mis à 0.

Le seuillage offre ainsi une autre protection contre le bruit. De plus, on voit que la valeur de ces seuils joue un rôle très important dans le visuel de notre résultat, c'est pourquoi il est important de pouvoir les modifier aisément. Cependant, cela ne suffit pas toujours, les images fortement texturées avec des intensités lumineuses variant très rapidement localement fournissent des résultats insatisfaisants par ce seuillage (faux contours et contours manquants). Ainsi une possibilité est d'utiliser d'autres seuils.

2.2.2 Réseaux de neurones convolutifs

2.2.2.1 Les réseaux de neurones

Fonctionnement du cerveau humain

Un neurone est composé de dendrites qui reçoivent des signaux qui sont traités et produisent des informations. Ces informations traversent ensuite l'axone et arrivent aux synapses, lieu de connection du neurone avec d'autres neurones. Les informations sont ainsi transmises aux autres neurones. Le transfert d'informations d'un neurone à un autre est à la fois électrique et chimique. On peut donc voir ce que fait un neurone comme un module de calcul très simple. Pourtant, une fois le neurone mis en connection avec un large nombre d'autres neurones, la correspondance des entrées et des sorties effectuée est remarquable.

Modèle du perceptron

Un perceptron est un type de neurone artificiel simplifié. Il prend plusieurs entrées et produit une sortie binaire ou une décision. Pour faire cela, il prend les entrées notées x_i qu'il multiplie par des poids notés w_i et compare la somme pondérée des entrées $\sum_i w_i x_i$ avec un seuil, aussi appelé biais et noté b. En fonction de la comparaison, on obtient une sortie, appelée valeur d'activation, qui est égale à 0 ou à 1 suivant la fonction suivante f :

$$f(x_i, w_i) = \begin{cases} 0 \; si \; \sum_i w_i x_i \leq -b \\ 1 \; si \; \sum_i w_i x_i > -b \end{cases}$$

On note $z = wx + b$ avec w le vecteur correspondant aux w_i et x le vecteur correspondant aux x_i. Ce que fait le perceptron revient donc à appliquer la fonction f à z :

$$f(z) = \begin{cases} 0 \; si \; z \leq 0 \\ 1 \; si \; z > 0 \end{cases}$$

Cette fonction, qu'on appelle la fonction d'activation, est une fonction en escalier. Cependant, la fonction d'activation présente certaines limites. En effet, les poids et les biais sont des paramètres d'un réseau de perceptron. Ainsi, il est important de comprendre l'impact qu'ils ont sur la sortie de la fonction d'activation. On voudrait donc qu'un léger changement du poids ou du biais entraîne également un léger changement dans le résultat de la fonction d'activation. Or ce n'est pas le cas : il est par exemple possible qu'en ne modifiant que très peu la valeur du poids, la valeur d'activation change radicalement en passant de 0 à 1. Et, au contraire, changer drastiquement la valeur du poids peut ne pas avoir d'impact sur la valeur d'activation. Dans ce contexte, il est très compliqué de savoir comment ajuster les poids et biais afin d'obtenir la sortie désirée. Ainsi, il serait plus intéressant d'utiliser une fonction continue, où le changement de valeur d'activation serait moins brusque. C'est justement le cas d'un neurone sigmoïde qui fonctionne comme le perceptron mais la fonction d'activation est différente. Cette fonction, appelée sigmoïde, est donnée par la formule suivante :

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Ainsi, si on change un poids ou un biais, la valeur de z va être modifiée. Si z diminue, la valeur d'activation diminue et, au contraire, si z augmente, la valeur d'activation augmente également.

Choix de la fonction d'activation

La fonction d'activation est donc la transformation appliquée à la donnée d'entrée d'un neurone, c'est-à-dire qu'elle permet de savoir si la donnée de sortie sera activée pour être en entrée du prochain neurone. Le choix de la fonction d'activation est donc très important pour avoir le résultat le plus précis en sortie du réseau de neurones. Ces fonctions d'activation appliquent une non-linéarité au modèle. L'avantage de la non linéarité est qu'elle peut modifier la représentation spatiale des données. Il existe de nombreuses fonctions d'activation dont les plus utilisées sont les suivantes en raison de leur efficacité et leur simplicité des calculs.

Premièrement, la fonction reLu (Rectified Linear Unit) : $reLu(x) = max(0, x)$. Elle ne renvoie que les valeurs positives et permet ainsi de garder uniquement certaines caractéristiques, c'est pourquoi elle est beaucoup utilisée pour la détection d'objets.

Deuxièmement, la fonction sigmoid : $sigmoid(x) = \frac{1}{1+e^{-x}}$. Elle retourne uniquement des valeurs comprises entre 0 et 1. Elle est généralement appliquée dans la classification binaire. La valeur retournée est considérée comme la probabilité que l'image appartienne à une certaine classe.

Ensuite, la fonction tanh : $tanh(x) = \frac{2}{1+e^{-2x}} - 1$. Elle renvoie des valeurs comprises entre -1 et 1. Elle permet de normaliser les données d'entrée. Elle est beaucoup employée dans la reconnaissance d'expressions faciales ou de gestes.

Enfin, la fonction softmax : $softmax(x) = \frac{e^x}{\sum_{i=0}^{n} e^{x_i}}$. Elle renvoie des valeurs entre 0 et 1 et permet de retourner une probabilité dans une classification multiclasse.

2.2.2.2 Architecture des réseaux de neurones convolutifs

Différentes couches d'un réseau de neurones convolutif

Les réseaux de neurones convolutifs, aussi appelés CNN, sont constitués de trois types de couches : les couches de convolution, les couches de pooling et les couches entièrement connectées. Entrons en détails sur le rôle de chacune de ses couches.

Premièrement, la couche de convolution est la base du CNN, c'est ici où est réalisée la majorité des calculs. Au sein de la couche de convolution est effectué un produit de convolution d'une donnée d'entrée, une matrice, par un filtre. La couche de convolution renvoie ainsi une carte de caractéristiques. Une carte de caractéristiques est une matrice qui contient les différentes features (éléments) de la donnée d'entrée. Par exemple, sur l'image d'une maison, les différentes features sont les fenêtres, les portes et le toit. Le filtre de convolution a pour but de détecter des caractéristiques comme les contours ou le bruit. La donnée d'entrée étant une matrice, elle peut être une image ou une carte de caractéristiques. Après chaque produit de convolution, une fonction d'activation, très souvent la fonction reLu, est appliquée sur la carte des caractéristiques afin de conserver uniquement les valeurs positives, c'est-à-dire les caractéristiques activées, et affecter 0 aux valeurs négatives. Afin de détecter plusieurs features différentes, il est possible d'appliquer plusieurs couches de convolution à la suite, chacune détectant une feature différente. Le CNN est ainsi hiérarchique.

Ensuite, la couche de Pooling permet de réduire la dimension, c'est-à-dire le nombre de paramètres d'entrée, tout en conservant le maximum d'information sur les caractéristiques de l'image. La réduction de paramètres permet de réduire la complexité des calculs dans les passages des différentes couches et ainsi qu'ils soient plus rapides. La couche de Pooling applique un filtre sur la carte de caractéristiques, dont les deux principaux sont le max Pooling et l'average Pooling. Le max Pooling consiste à renvoyer les valeurs maximales du champ réceptif pour la transmettre en sortie du produit de convolution. Quant à l'average pooling, il renvoie la valeur moyenne des valeurs du champ réceptif. La différence de ces deux fonctions est illustrée sur l'image ci-dessous.

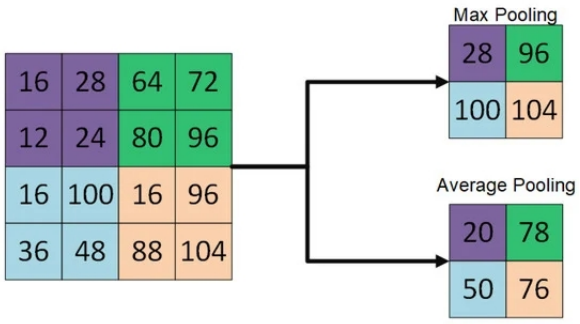


FIGURE 2.1 – source : www.mdpi.com/2313-433X/6/11/121

Enfin, la couche entièrement connectée repose sur le principe de base des réseaux de neurones expliqué précédemment. Cette couche présente cependant une particularité : chaque neurone reçoit en entrée les sorties de tous les neurones de la couche précédente. Cette

couche permet de classer l'image. La classification est faite à partir de la carte des caractéristiques obtenues à travers la couche de convolution et de Pooling.

Fonctionnement du processus de convolution

Le processus de convolution est un empilement des différentes couches vues ci-dessus, qui permet de classer une image. L'architecture débute toujours par une couche de convolution, qui elle, peut être suivie soit par une même couche soit par une couche de Pooling. Enfin, l'architecture du CNN se termine obligatoirement par une couche entièrement connectée. Un exemple d'architecture de CNN est illustré par le schéma ci-dessous.

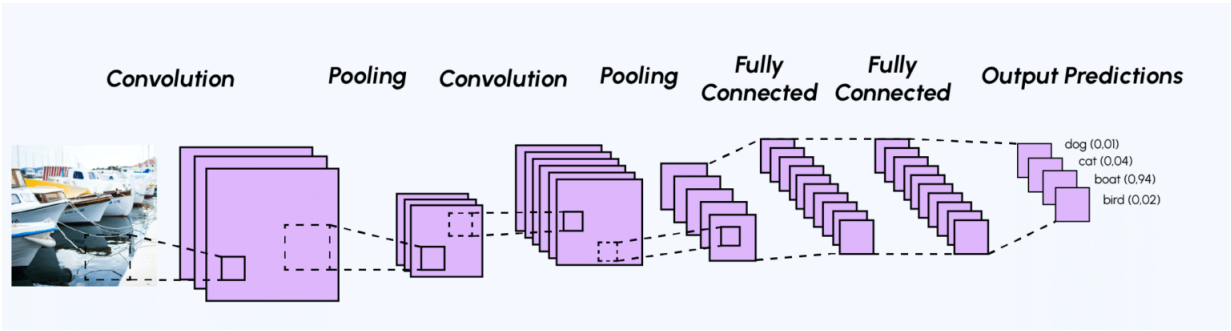


FIGURE 2.2 – source : datascientest.com/convolutional-neural-network

L'image en entrée est filtrée et réduite plusieurs fois. Elle passe ensuite à travers les couches entièrement connectées qui renvoient une probabilité que l'image soit d'une certaine classe.

2.2.2.3 Entrainement des réseaux de neurones convolutifs

Descente du gradient

Nous avons vu qu'un réseau de neurones a des paramètres : les poids et les biais. Ainsi, après avoir défini l'architecture d'un réseau de neurones, il faut l'entraîner afin d'avoir des paramètres optimaux. Nous allons donc maintenant voir comment un réseau de neurones est entraîné. Tout d'abord les valeurs des poids et des biais sont initialisés de manière aléatoire. Ensuite, on calcule la valeur de la fonction d'activation avec des entrées dont on connaît la sortie. En fonction du résultat obtenu, on modifie les valeurs des poids et des biais : plus la performance du réseau de neurone est faible, plus on augmente le coût. Pour calculer le coût, on utilise la formule suivante :

$$C_x(w,b) = \| \hat{a}(x_{w,b}) - a(x_{w,b}) \|$$

avec $\hat{a}(x_{w,b})$ l'activation désirée et $a(x_{w,b})$ l'activation obtenue avec les paramètres w et b fixés. Ainsi, pour l'entièreté des données d'entraînement, on fait la moyenne de chaque coût :

$$C(w,b) = \frac{1}{n} \sum_x C_x(w,b)$$

avec n le nombre de données d'entraînement.

Entraîner un réseau de neurones consiste donc à ajuster les poids et les biais afin de minimiser ces coûts. Pour cela, on utilise une méthode d'optimisation qui est la descente du gradient. Les explications mathématiques de cette méthode sont présentées en annexe.

2.3 Traitement fréquentiel d'une image

Une image est un signal. Il est donc possible d'analyser ses fréquences spatiales, c'est ce qu'on appelle une analyse spectrale. Ainsi, le traitement d'images peut non seulement être effectué dans le domaine spatial mais aussi dans le domaine fréquentiel.

2.3.1 Transformée de Fourier

2.3.1.1 Explications théoriques

Pour une image, la transformée de Fourier d'une fonction est un outil mathématique permettant de passer du domaine spatial au domaine fréquentiel.

Il existe deux formules pour la transformée de Fourier : une pour le domaine continu et une pour le domaine discret. Dans le cadre de notre étude où les images numériques ont un nombre de pixels défini et sont associées à des matrices dont chaque coefficient correspond à l'intensité lumineuse du pixel, nous allons utiliser la transformée de Fourier discrète. Soit f la fonction qui, à chaque coordonnée spatiale (m,n) d'une image de taille $M \times N$ associe l'intensité lumineuse du pixel correspondant. On note u et v les variables décrivant les coordonnées fréquentielles. La transformée de Fourier discrète de cette image est la fonction F définie par :

$$F(u,v) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) e^{-i2\pi(m\frac{u}{M}+n\frac{v}{N})}$$

En appliquant la transformée de Fourier discrète à une image, on obtient un spectre de Fourier des différentes fréquences et amplitudes de l'image. Au contraire, si l'on souhaite passer d'une représentation fréquentielle à une représentation spatiale, il faut utiliser l'inverse de la transformée de Fourier discrète F qui est définie par :

$$f(m,n) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) e^{i2\pi(m\frac{u}{M}+n\frac{v}{N})}$$

La transformée de Fourier est à valeurs complexes donc elle peut s'écrire sous la forme : $F(u,v) = \rho(u,v)e^{i\phi(u,v)}$ avec $\rho(u,v)$ le module de F et $\phi(u,v)$ sa phase. Généralement, l'application d'un filtre sous forme de fonction est réalisée par le calcul du produit de convolution de la fonction de l'image par celle du noyau. La transformée de Fourier est un moyen plus simple d'appliquer un filtre à une image. En effet, une propriété très importante de la transformée de Fourier est que la convolution de deux fonctions correspond simplement à la multiplication des transformées de Fourier de ces dernières. Ainsi, afin de traiter une image dans le domaine fréquentiel, il suffit de calculer les transformées de Fourier des fonctions de l'image et du noyau, de les multiplier et de déterminer l'inverse de la transformée de Fourier discrète pour obtenir l'image filtrée.

2.3.1.2 Explications sur la représentation visuelle

On appelle plan de Fourier la représentation de la transformée de Fourier. Une image étant un signal, on lui associe un module et une phase. La représentation du module de la transformée de Fourier discrète permet de voir les basses et hautes fréquences de l'image. Les basses fréquences correspondent aux changements d'intensité graduels et aux zones homogènes, floues. Elles se trouvent au centre du plan de Fourier. Les hautes fréquences correspondent aux changements d'intensité brusques, aux contours, aux zones texturées et au bruit. Elles se trouvent éloignées du centre du plan de Fourier. Même si le module est plus

facile à interpréter, il n'en reste pas moins que la phase doit être considérée. Très souvent, la phase est plus importante que le module pour assurer la conservation des informations de l'image.

2.3.1.3 Différents filtres

Dans le traitement d'images, il existe de nombreux filtres dont nous pouvons donner quelques exemples.

Les filtres passe-bas et passe-haut sont très connus dans le domaine fréquentiel. Un filtre passe-bas permet d'atténuer les détails, de flouter une image. En effet, ce filtre ne conserve que les basses fréquences, c'est-à-dire les pixels qui ont des intensités lumineuses proches. Les hautes fréquences prennent alors une valeur moyenne des basses fréquences. A l'inverse, un filtre passe-haut permet de mettre en avant les contours d'une image. En effet, ce filtre ne conserve que les hautes fréquences et donc les changements brusques d'intensité lumineuse.

Il existe de nombreux autres filtres parmi lesquels on peut citer le filtre Gaussien. Il suffit de calculer la transformée de Fourier de l'image, celle de la fonction gaussienne, de multiplier ces deux fonctions et, enfin, d'appliquer l'inverse de la transformée de Fourier. L'image obtenue est la même que celle que l'on aurait en utilisant la convolution dans le domaine spatial.

2.3.2 Comparaison entre le domaine spatial et le domaine fréquentiel

2.3.2.1 Applications du filtrage dans le domaine fréquentiel

Le domaine fréquentiel peut être très utile et est donc utilisé dans un certain nombre de domaines. Nous allons en présenter quelques uns.

En IRM, le signal obtenu correspond à la transformée de Fourier. En utilisant la transformée de Fourier inverse, on obtient l'image qui nous intéresse.

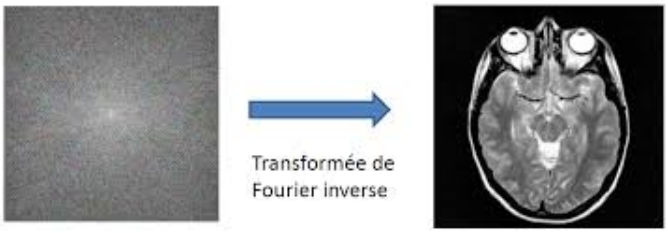


FIGURE 2.3 – source : perso.univ-rennes1.fr/pierre.maurel/IMA/CM/ima05_Fourier_imprimable.pdf.

La transformée de Fourier est également utile pour compresser des images. En effet, l'œil est moins sensible aux hautes fréquences qu'aux basses fréquences : on remarque moins une perte en hautes fréquences qu'en basses fréquences. Ainsi, en ne conservant que les basses fréquences d'une image, on gagne de la place sans perdre trop d'informations.

En traitant les images dans le domaine fréquentiel, il est possible d'obtenir des images hybrides. Par exemple, on a deux images initiales numérotées 1 et 2 et on leur applique la transformée de Fourier en ne gardant que les hautes fréquences pour la 1 et les basses fréquences pour la 2. En faisant la moyenne de ces deux images, on obtient une image hybride. En étant proche de cette image on voit les détails, donc les hautes fréquences, c'est-à-dire l'image 1. En revanche, en s'éloignant de l'image, notre œil agit comme un filtre passe-bas et on voit l'image 2.

Une autre chose qu'il est possible de faire beaucoup plus simplement dans le domaine fréquentiel que dans le domaine spatial est la déconvolution. Elle est très utile dans le cadre de la restauration d'images. En effet, lorsque l'on prend une photographie mais que l'on tremble, l'image est floue, c'est comme si on avait fait la convolution de l'image idéale, non floutée, avec un filtre qui floute. On cherche donc à enlever cette convolution, c'est-à-dire à déconvoluer cette image. Si on note f la fonction représentant l'image idéale, f' l'image floue et h le filtre floutant on a la relation suivante : $f(m,n)\ h(m,n) = f'(m,n)$. Or avec les transformées de Fourier discrètes, on a : $F(u,v)\ H(u,v) = F'(u,v)$. On a donc simplement : $F(u,v) = F'(u,v)/H(u,v)$. Et, en appliquant l'inverse de la transformée de Fourier discrète, on trouve $f(m,n)$. En réalité, en plus du flou causé par le mouvement lors de la prise de la photographie, il y a également du bruit. La relation est donc complexifiée, on n'utilise donc pas un simple filtre floutant mais le filtre de Weiner, ce qui permet de prendre en compte le bruit.

2.3.2.2 Compléxité du code

On peut comparer la complexité du code suivant l'utilisation du domaine spatial ou fréquentiel. On considère une image de taille $N \times N$ et un noyau de taille $K \times K$. Dans le cas du domaine spatial, la complexité de l'implémentation de la convolution directe est en $O(K^2N^2)$. Dans le cas du domaine fréquentiel, la complexité de l'implantation de la convolution par multiplication est celle de deux calculs de transformée de Fourier (un direct et un inverse) et d'une multiplication. Le coût de la multiplication est en $O(N^2)$. Et le coût de la transformée de Fourier est en $O(N^2 log_2(N))$ si on utilise la FFT. La FFT est un algorithme qui utilise la récursivité et les propriétés des racines n-ièmes de l'unité, il est présenté en annexe. Dans ce cas, la complexité est indépendante de la taille $K \times K$ du noyau. Par conséquent, pour de grands noyaux $(K^2 >> log_2(N))$ il est plus intéressant d'utiliser le domaine fréquentiel. Et, au contraire, pour de plus petits noyaux, il est préférable d'utiliser le domaine spatial.

2.4 Implémentation informatique

2.4.1 Réalisation d'un algorithme de convolution naïf

Après avoir vu la partie théorique de la convolution de matrices il est maintenant temps de l'implémenter pour pouvoir faire nos propres tests de convolutions d'images. Mais avant toute chose il faut se mettre d'accord sur le langage de programmation à utiliser. Nous avons choisi le langage **Python** car il est simple à utiliser et intuitif. De plus il est aussi enseigné en I4, cours auquel tous les membres du groupe sont inscrits.

Pour ce qui est de la répartition du travail nous avons séparé la logique métier de l'interface homme-machine, et nous aborderons principalement dans cette partie la logique métier. La première version du code a été faite de la manière la plus simple et directe possible car l'algorithme de convolution repose sur des concepts faciles à implémenter en **Python** : la multiplication d'entiers et la gestion d'indices de tableaux. La seule difficulté a été celle de la gestion des bords où il faut remplacer par des 0 les accès à des indices hors du tableau représentant l'image afin d'éviter des erreurs de dépassement de tableau (code en annexe). Ce premier algorithme utilisait également la librairie `Pillow` pour la gestion des images : ouvrir l'image, récupérer la couleur d'un pixel, changer la couleur d'un pixel, etc.

2.4.2 Complexité du code et problèmes d'optimisation

2.4.2.1 Complexité

Après avoir implémenté cet algorithme, les premiers tests nous ont permis de réaliser que les calculs étaient longs dès que les images étaient grandes. Par exemple, pour traiter une image d'une taille 1920 $\times$ 1080 pixels, le temps d'exécution était d'environ 30 secondes selon l'ordinateur sur lequel le code était exécuté.

Cela s'explique par la complexité du code précédemment calculée : $O(H \times L \times P \times K^2)$ avec H la hauteur de l'image, L la largeur, K le côté du noyau et ici on rajoute P pour la profondeur de l'image qui vaut presque tout le temps 3 pour les couleurs de chaque pixel : rouge, vert et bleu. Pour tester ensuite l'efficacité de ce premier algorithme nous avons développé une version du code en utilisant plusieurs librairies externes (code en annexe) dont `scipy` qui inclut un algorithme optimisé de convolution de matrices. Les résultats ont été clairs : notre code était entre 35 et 45 fois plus lent que celui avec `scipy`.

2.4.2.2 Multiprocessing

La première idée pour régler ce problème de temps de calcul a été d'utiliser le multiprocessing. Dans un ordinateur, la partie permettant de gérer tous les calculs est le processeur. Celui-ci est souvent divisé en plusieurs cœurs, généralement de 6 à 12, qui peuvent chacun exécuter une tâche en parallèle, donc exécuter plusieurs tâches en même temps. Le multiprocessing consiste à diviser un algorithme en plusieurs parties et à donner chacune de ces parties à un cœur du processeur. Cette méthode a été implémentée (code en annexe) en divisant l'image en 3 : la partie rouge, la verte et la bleue. Les temps de calcul ont donc été divisé par 3 mais le résultat n'était toujours pas satisfaisant.

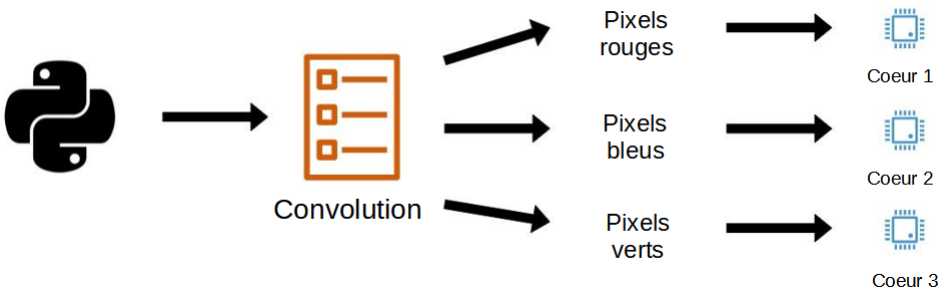


FIGURE 2.4 – source : medium.com/@mehta.kavisha/different-methods-of-multiprocessing-in-python

2.4.2.3 Langages de haut et bas niveau

Après avoir cherché dans la documentation de `scipy`, nous avons découvert que cette librairie était codée en **C**. Le problème du temps de calcul vient donc du langage **Python** lui-même. En effet **Python** est un langage de haut niveau, qui s'oppose aux langages de bas niveau. Ces derniers sont des langages où la syntaxe est plus rigoureuse mais permettent des échanges d'instructions plus rapides avec le processeur. Donc même si **Python** est intuitif et facile à comprendre il n'est adapté pour un algorithme de convolution. Cet algorithme, reposant sur la réalisation d'une énorme quantité de petits calculs, est parfaitement adapté pour un langage de bas niveau tel que le **C**.

Une nouvelle version du code a donc été réalisée en **C** (code en annexe) et cette fois-ci les résultats étaient bien meilleurs avec une amélioration du temps de calcul d'un facteur entre 20 et 30.

2.4.2.4 Résultats

Voici les résultats pour une image de taille 4000×2250 :
— Algorithme naïf : 145,6 s
— `scipy` : 4,2 s
— Multiprocessing : 72,2 s
— **C** : 6,8 s

En réalité la version qui utilise le multiprocessing n'est pas réellement 3 fois plus rapide car la séparation de l'image en 3 prend du temps et le temps d'exécution dépend également de la disponibilité des cœurs. Des exemples de convolutions d'images sont présentes en annexe.

2.4.3 Réalisation d'une première interface graphique avec `Tkinter`

En parallèle à la réalisation de notre algorithme de convolution nous avons décidé de développer une interface graphique utilisateur (GUI) intuitive et interactive. En utilisant des outils comme `Tkinter` et `Kivy`, nous avons pu créer des interfaces graphiques qui facilitent la manipulation des images et l'application des filtres de convolution, rendant notre projet plus accessible au public.

L'objectif de cette première interface graphique était d'afficher une fenêtre possédant principalement :
— un titre "Projet scientifique encadré : Convolution de matrices",
— un bouton "Choisir une image" qui déclenche une fonction ouvrant l'explorateur de fichiers et permettant d'obtenir le chemin d'une image,
— des cases à cocher nommées avec les noms de nos filtres, permettant de garder en mémoire si le filtre sera ou non appliqué,
— un bouton "Appliquer le/les filtre(s)" qui vérifie la valeur de chaque case pour savoir si elle est cochée ou non, et applique les filtres cochés à l'image choisie, puis l'exporte avec le suffixe "convolué".

Bien que `Tkinter` soit la bibliothèque principale utilisée, d'autres bibliothèques ont été employées, telles que `Pillow`, une bibliothèque **Python** permettant notamment d'ouvrir, manipuler et sauvegarder des images, ainsi que `os` pour manipuler les chemins de fichiers.

La première étape a été de créer une fenêtre vierge avec `Tkinter` et de définir quelques paramètres basiques : ses dimensions, son titre, la couleur de son fond, etc. Ce qui a été possible grâce à la création d'un objet `Tk`, représentant notre fenêtre principale, et en configurant ses attributs comme la taille, le titre, et la couleur de fond.

Ensuite, nous avons ajouté des composants interactifs, notamment des boutons et des cases à cocher. Le bouton "Choisir une image" permet à l'utilisateur de sélectionner une image via l'explorateur de fichiers, en utilisant la fonction `askopenfilename` de `Tkinter`. Une fois l'image sélectionnée, elle est affichée dans la fenêtre grâce à la bibliothèque `Pillow`, qui la redimensionne et la convertit en un format compatible avec `Tkinter` (`PhotoImage`).

Les cases à cocher permettent de sélectionner les filtres à appliquer à l'image. Chaque case à cocher est liée à une variable qui conserve l'état (cochée ou non) du filtre correspondant. Lorsque l'utilisateur clique sur le bouton "Appliquer les filtres", la fonction associée vérifie l'état des cases à cocher et applique les filtres sélectionnés à l'image en utilisant `Pillow`. Les filtres disponibles incluent le flou, l'amélioration de la netteté, et la détection de contours.

2.4.3.1 Intérêt de l'utilisation de `Tkinter`

`Tkinter` a été choisi pour sa simplicité et son intégration native avec **Python**, ce qui en fait une option idéale pour créer des interfaces graphiques rapidement sans nécessiter de configurations complexes. De plus, `Tkinter` est bien documenté et largement utilisé, offrant ainsi de nombreux exemples et ressources pour résoudre d'éventuels problèmes rencontrés lors du développement.

2.4.3.2 Apports de l'interface graphique au projet

Cette interface graphique a apporté plusieurs avantages significatifs au projet :
— **Interactivité** : Elle permet aux utilisateurs de manipuler facilement les images et d'appliquer divers filtres en quelques clics, rendant le processus de convolution plus accessible.
— **Visualisation** : En affichant immédiatement les résultats des filtres appliqués, elle facilite la compréhension des effets de chaque filtre sur l'image.
— **Flexibilité** : Grâce aux cases à cocher et au bouton d'application des filtres, l'interface offre une grande flexibilité dans le choix et la combinaison des filtres, permettant ainsi d'expérimenter différentes configurations de manière intuitive.
— **Praticité** : L'utilisation de bibliothèques standardisées comme `Tkinter` et `Pillow` assure une compatibilité et une simplicité de mise en œuvre, évitant des dépendances complexes.

Ainsi, cette interface graphique joue un rôle crucial dans la démonstration pratique du traitement d'images par convolution, en rendant le processus plus interactif et compréhensible pour les utilisateurs.

2.4.4 Réalisation d'une deuxième interface avec `Kivy`

Suite à cette première interface graphique, M.Lecomte nous a suggéré l'idée de créer une interface avec une bibliothèque différente (`Kivy`) et un autre paradigme de programmation qu'est la programmation orientée objet.

Cette notion n'étant pas abordée en STPI2 nous avons premièrement effectué un travail de recherche sur ce paradigme ainsi que sur la bibliothèque. La POO repose sur l'utilisation de classes, possédants certains attributs ainsi que certaines méthodes que nous pouvons utiliser sur ceux-ci. La bibliothèque `Kivy` utilise elle-même la librairie SDL que nous avions déjà pu rencontrer lors de nos projets informatiques au semestre 3.

Le code **Python** (disponible en annexe) se décompose en plusieurs parties :
— La création et l'affichage des boutons et des listes déroulantes à partir d'un fichier contenant les références des images et des noyaux dans des dictionnaires. La gestion des données disponibles pour la convolution se fait ainsi en dehors du programme de l'interface et permet d'être modifiée plus facilement. (Nous n'avons ici pas eu le besoin de le faire car nous ne testions que sur peu d'images mais une piste d'amelioration pourrait être la réecriture de cet algorithme afin d'intégrer toutes les images d'un fichier au dictionnaire de façon automatique.)
— L'appel de fonctions par un clic sur les boutons. Notamment la fonction convoluer de l'unité fournie par Thomas, qui sera appelé avec les attributs self.nom_image et self.nom_noyau, qui indiqueront les chemin d'accès aux images et noyaux choisis.

Séparer ainsi la partie interface humain machine de la partie logique métier nous a per-

mis de faire fonctionner des interfaces différentes avec un même programme, et inversement, avec une même interface , de tester différentes fonctions de convolution dont une codée en **C** par Thomas.

Conclusion et perspectives

Conclusions sur le travail réalisé Au cours de ce semestre nous avons pu découvrir l'outil de convolution de matrices, cela a mené à un travail de recherche ainsi que de mise en application. De nombreux aspects mathématiques ont été abordés tels que la création et l'utilisation de noyaux, les CNN, le lien entre la convolution et les séries de Fourier etc. Puis un algorithme de convolution a été codé, utilisable à l'aide de deux interfaces graphiques différentes qui nous ont permis de voir des cas concrets d'applications avec des photos et des noyaux choisis. L'organisation par groupe et un travail régulier nous ont mené à explorer de nombreux domaines d'applications et ce, sans rencontrer de problème majeur durant ce projet.

Conclusions sur l'apport personnel de cet E.C. projet Lors de ce projet qui mélange informatique et mathématiques nous avons pu découvrir un nouvel outil et y appliquer de nombreuses notions vues en cours cette année : séries de fourrier, CNN, algorithmie etc. De plus, nous avons pu nous familiariser avec le langage LaTeX. L'organisation en groupes nous a permis d'explorer des aspects selon nos préférences respectives, tout en apprenant du travail des autres grâce à une bonne communication au sein de l'équipe. Nous nous sommes rendus compte que les difficultés de ce projet se trouvaient tout autant dans les notions que dans l'organisation à mettre en place, notamment au niveau de l'écriture du rapport. De plus, ce projet nous a montré qu'il est simple de se perdre dans différentes pistes et de s'éloigner du sujet sans l'aide d'un référent.

Nous sommes unanimes quant à l'intérêt de ce projet, différent des précédents de par ses domaines de recherche plus divers. En effet, il est ce qui se rapproche le plus d'une expérience en milieu professionnel. De ce fait, nous avons tous trouvé un réel interêt à mener à bien ce projet, au-delà des connaissances acquises.

Perspectives pour la poursuite de ce projet Pour la partie implémentation, il pourrait être intéressant de coder un filtre complet (comme celui de Canny) afin d'obtenir un résultat plus convainquant. De plus, il faudrait ainsi penser le programme différemment pour ces filtres qui utilisent deux noyaux (horizontal + vertical). D'autres exemples d'applications, notamment avec les CNN ou les séries de Fourier, pourraient être envisagés car ils créeraient un lien et renforceraient nos acquis dans les matières étudiées ce semestre.

Bibliographie

[1] AUDIBERT, Thierry et OUSSALAH, Amar *Informatique tronc commun - CPGE scientifiques 1re et 2e années - Nouveaux programmes* , Ellipses, 2021.

[2] NOM DES AUTEURS, "*Titre de l'article*", Titre du journal, Volume, pages, année.

[3] AIX MARSEILLE UNIVERSITÉ, "*Traitement du signal et des images*", https://www.google.com/url?q=https://www.fresnel.fr/perso/marot/Documents/Enseignements/ATI/CoursImNum4_TF2D.pdf&sa=D&source=docs&ust=1716281268964384&usg=AOvVaw0DeLXZ7a3isf5jJ91s0V_s (Valide à la date du 21/05/2024)

[4] BERGOUNIOUX MAÏTINE, "*Méthodes de filtrage en Traitement d'Image*", https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://hal.science/hal-00512280v2/file/CoursFiltrage.pdf&ved=2ahUKEwjVusDRop6GAxWRVqQEHeX4ACIQFnoECA8QAQ&usg=AOvVaw3fmjgEFEm6AXwnwUdsjQFw (Valide à la date du 21/05/2024)

[5] FIRST PRINCIPLES OF COMPUTER VISION, "*Image Filtering in Frequency Domain*", https://www.youtube.com/watch?v=OOu5KP3Gvx0&list=PL2zRqk16wsdorCSZ5GWZQr1EMWXs2TDeu&index=12 (Valide à la date du 21/05/2024)

[6] HUMBERT FLORENT, PUBLIÉ LE 01/04/2007, "*La transformée de Fourier en traitement d'images*", https://humbert-florent.developpez.com/algorithmique/traitement/fourier/ (Valide à la date du 21/05/2024)

[7] LOSSON OLIVIER, "*Transformée de Fourier d'une image*", https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://www.fil.univ-lille.fr/~losson/aiv1/03-compression/aiv1-semaine-03-compression.pdf&ved=2ahUKEwim6-PxoZ6GAxXeUKQEHcRTAc4QFnoECBYQAQ&usg=AOvVaw1bFX5TOJPGBVpxrYJlTPc0 (Valide à la date du 21/05/2024)

[8] MALGOUYRES FRANÇOIS, "*Signal, Fourier, Image*", https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://www.math.univ-toulouse.fr/~fmalgouy/enseignement/downloadSfi/poly_sfi.pdf&ved=2ahUKEwjejqXzop6GAxV9UaQEHTcVBnIQFnoECA8QAQ&usg=AOvVaw1oxlepLTK7CMbaS5s8Uboo (Valide à la date du 21/05/2024)

[9] MAUREL PIERRE, "*Transformée de Fourier pour l'imagerie numérique*", https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://perso.univ-rennes1.fr/pierre.maurel/IMA/CM/ima05_Fourier_imprimable.pdf&ved=2ahUKEwjoxqubo56GAxVgU6QEHTzCBgcQFnoECBEQAQ&usg=AOvVaw17HB9BoCcGPm8Ie-mtqiGm (Valide à la date du 21/05/2024)

[10] "*Convolution de matrice en deux dimensions*", https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://exo7math.github.io/deepmath-exo7/convolution2d/convolution2d.pdf&ved=2ahUKEwiDqJKzop6GAxWIVaQEHTFAB8sQFnoECBAQAQ&usg=AOvVaw0nNo-LNx2HugvBsskX2tJP (Valide à la date du 21/05/2024)

[11] "*Convolutional Neural Networks : A Comprehensive Guide*", https://medium.com/thedeephub/convolutional-neural-networks-a-comprehensive-guide-5cc0b5eae175 (Valide à la date du 27/03/2024)

[12] "*Filtrage et Restauration*", https://www.google.com/url?q=https://perso.ensta-paris.fr/~manzaner/Cours/Poly/Poly_Chap2_Filtrage.pdf&sa=D&

source=docs&ust=1716281328785861&usg=AOvVaw2XfDGZX21y2mVlsFaOzPwA (Valide à la date du 21/03/2024)

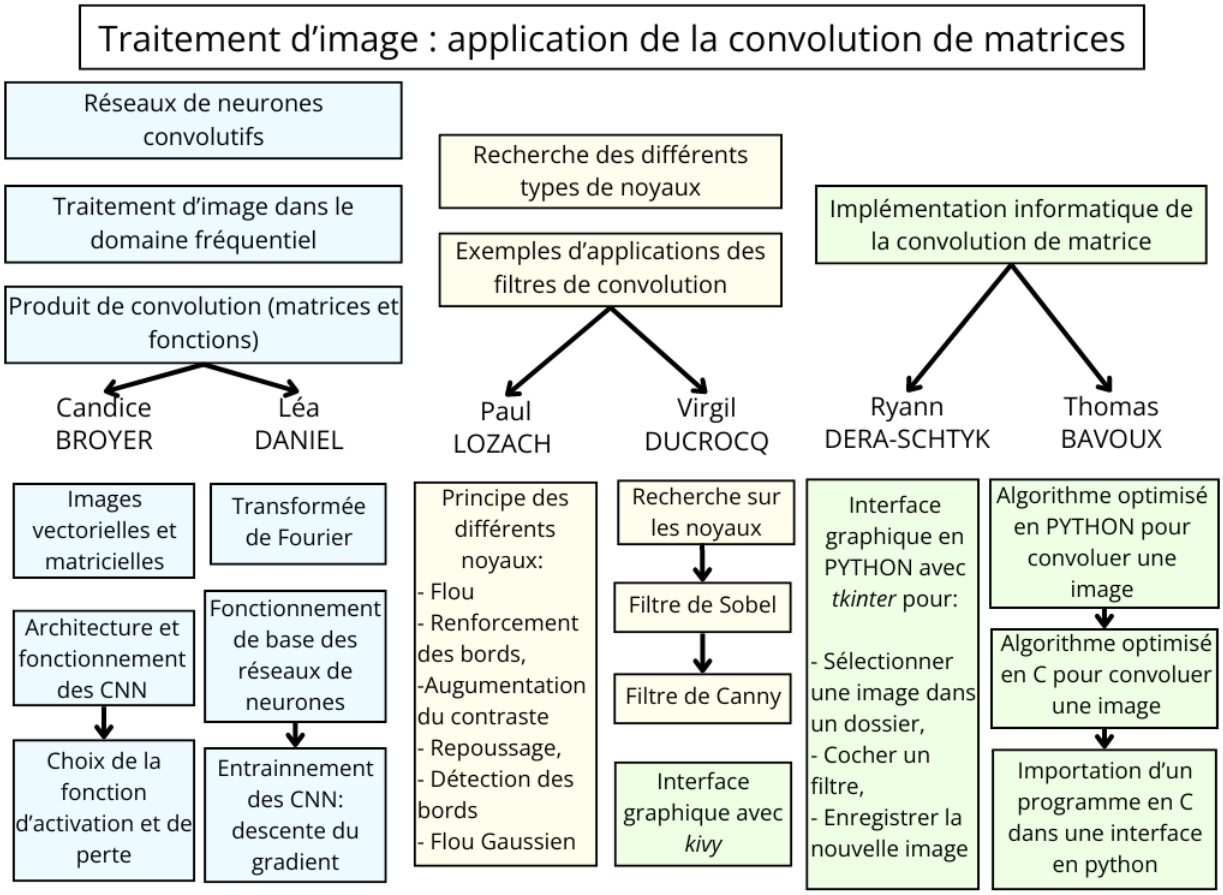
[13] "*Que sont les réseaux neuronaux ?*", https://www.ibm.com/fr-fr/topics/neural-networks (Valide à la date du 21/05/2024)

[14] "*Transformée de Fourier d'une image*", https://www.f-legrand.fr/scidoc/docmml/numerique/tfd/tfdimage/tfdimage.html (Valide à la date du 21/05/2024)

[15] "*Documentation* `Kivy`", https://kivy.org/doc/stable/ (Valide à la date du 30/05/2024)

Annexes

A.1 Organigramme



A.2 Noyaux

A.2.1 Repoussage

Ce noyau produit un effet de relief à partir de l'image source. GIMP utilise le noyau suivant :

$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 1 & -1 \\ 0 & -1 & -2 \end{bmatrix}$$

On peut découper ce noyau en 2 parties :

$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 1 & -1 \\ 0 & -1 & -2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}$$

Il s'agit donc du noyau identité avec un ajout d'un certain type de noyau de détection des bords. Les modifications apportées par ce filtre sont calculées avec $f * \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}$. Grâce aux coefficients 2 et -2, les bords sont accentués. En effet, de la distance est créée entre les objets et les creux qui sont souvent détectables, comme par exemple les bords. Il est intéressant de noter qu'on obtient un résultat similaire mais beaucoup moins prononcé avec la matrice $\begin{bmatrix} -2 & -1 & 0 \\ -1 & 5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$, on peut donc faire l'analogie entre le noyau de rehaussement des bords et de détection des bords abordé précédemment.

A.2.2 Flou Gaussien

Le but du filtre de Gauss est de "lisser" l'image afin de se débarrasser d'éventuelles valeurs de pixel aberrantes qui pourraient compromettre la détection de bords. En effet, les algorithmes de détection de bords se basent sur la détection de changements locaux importants des valeurs des pixels. Retirer les valeurs aberrantes permet donc de limiter l'apparition de "faux positifs" et donc d'éviter de détecter des bords qui ne le sont pas.

Plusieurs autres filtres permettent de faire la même chose : le filtre moyenne ou le filtre médian. Ces filtres changent la valeur d'un pixel en fonction de la moyenne ou de la médiane des 9 ou 25 pixels autour de celui que l'on modifie. Ils compromettent donc grandement la qualité de l'image, elle nous apparaît plus floue. Le filtre de Gauss se base sur le même principe. Cependant, la méthode de calcul pondère l'influence de chaque pixel différemment en fonction de la distance qui les sépare du pixel central. Plus le pixel est proche du pixel central, plus la pondération sera élevée. Les valeurs de pondération sont déterminées selon une courbe gaussienne, d'où vient le nom du filtre. Il existe donc plusieurs types de filtres de Gauss en fonction de l'écart type choisi pour la courbe de Gauss.

Voici la formule mathématique d'une courbe gaussienne en 2 dimensions :

$$\frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

avec σ l'écart type de la gaussienne. Pour obtenir un noyau de Gauss pour la convolution matricielle, on discrétise l'espace, les x et y représentant les coordonnées des cases du noyau. En implémentant le problème en **Python**, on peut obtenir le code présenté dans l'annexe suivante. Ce code permet d'obtenir n'importe quelle dimension de noyau de Gauss en fonction d'un écart type prédéfini.

A.3 Code noyau de Gauss

```
from math import exp, pi, floor

def g(x, y, sig):
    return exp(- ((x ** 2) + (y ** 2)) / (2 * (sig ** 2)))/(2 * pi * (sig ** 2))

def matGauss(n, sig):
    assert n % 2 == 1
    return [[g(x-floor(n/2), y-floor(n/2), sig) for x in range(n)] for y in range(n)]

def sumM(mat):
    return sum([sum(mat[i]) for i in range(len(mat))])

if __name__ == "__main__":
    m=matGauss(5, 1)
    sum=sumM(m)
    print(m,sum)
```

La fonction g est la fonction de Gauss définie préalablement sans les constantes. En effet, nous avons affaire à des poids, des coefficients. Comme un traitement de perte est déjà mis en place dans le programme principal, les constantes n'auront pas d'effet.

La fonction matGauss calcule ensuite le noyau recherché en s'adaptant aux dimensions et en vérifiant que la dimension choisie est bien impaire.

La fonction sumM calcule la somme de tous les coefficients calculés dans le noyau pour éventuellement effectuer plus tard un calcul d'erreur pour se rendre compte des approximations faites lors de la discrétisation. En effet, l'intégrale d'une gaussienne est égale à 1 donc la somme, qui peut être considérée comme une intégrale discrétisée, doit approcher 1.

A.4 Méthode du gradient pour l'entrainement des réseaux de neurones

Nous avons donc une fonction C qui calcule le coût :

$$C_x(w,b) = \| \hat{a}(x_{w,b}) - a(x_{w,b})\|$$

avec $\hat{a}(x_{w,b})$ l'activation désirée et $a(x_{w,b})$ l'activation obtenue avec les paramètres w et b fixés. Et pour l'entièreté des données d'entraînement, on a la moyenne de chaque coût :

$$C(w,b) = \frac{1}{n}\sum_x C_x(w,b)$$

avec n le nombre de données d'entraînement. On veut trouver les paramètres (poids w et biais b) qui la minimisent. On considère un réseau de neurones avec essentiellement deux paramètres : un pour le poids noté w_i et un pour le biais noté b_j. Dans ce cas, on a une fonction bidimensionnelle. Notre but est d'arriver au coût le plus faible, donc au minimum de la fonction coût et de trouver les valeurs de w_i et b_j associées. Pour cela, on mesure le changement du coût lors de la modification des paramètres w_i et b_j. Pour évaluer le changement de la fonction coût, on a la formule des séries de Taylor :

$$f(x+\Delta x, y+\Delta y) = f(x,y) + \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y$$

Le changement de la fonction est donc :

$$\Delta f(x,y) \approx \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y$$

Dans notre cas, on a :

$$\Delta C \approx \frac{\partial C}{\partial w_i}\Delta w_i + \frac{\partial C}{\partial b_j}\Delta b_j$$

On peut écrire cela sous forme vectorielle :

$$\Delta C = \begin{bmatrix} \frac{\partial C}{\partial w_i} & \frac{\partial C}{\partial b_j} \end{bmatrix} \begin{bmatrix} \Delta w_i \\ \Delta b_j \end{bmatrix}$$

Ainsi, on obtient :

$$\Delta C = \nabla C \, \Delta v$$

avec le gradient $\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_i} & \frac{\partial C}{\partial b_j} \end{bmatrix}$ et $\Delta v = \begin{bmatrix} \Delta w_i \\ \Delta b_j \end{bmatrix}$.
Le gradient appliqué à la fonction coût indique dans quelle direction elle augmente le plus. Or, le but est qu'elle diminue, il faut donc de changer les variables dans la direction opposée à cette dernière. On a ainsi la relation :

$$\Delta v = -\nabla C$$

Avec les formules précédentes, on obtient donc finalement :

$$\Delta C = \|\nabla C\|^2$$

On peut introduire un nouveau paramètre η :

$$\Delta C = -\eta\|\nabla C\|^2 \text{ et } \Delta v = -\eta \nabla C$$

η correspond au taux d'apprentissage. Il permet de savoir à quelle vitesse s'effectue la descente. Avec cela, on sait comment il faut modifier les paramètres w_i et b_j en w_i' et b_j' :

$$\begin{cases} {w_i}' = w_i - \eta \frac{\partial C}{\partial w_i} \\ {b_j}' = b_j - \eta \frac{\partial C}{\partial b_j} \end{cases}$$

Par conséquent, pour chaque itération, on a les paramètres w_i et b_j et il suffit de calculer la dérivée de la fonction coût par rapport à chaque paramètre et d'appliquer les formules du système ci-dessus afin d'obtenir les valeurs des nouveaux paramètres.
On peut maintenant s'intéresser à l'effet du paramètre η. Si on prend un taux d'apprentissage faible, on va se déplacer vers le minimum lentement. En revanche, si on prend un taux d'apprentissage élevé, on va se déplacer vers le minimum trop rapidement et le dépasser voire même trouver un minimum local qui n'est pas celui recherché. Il est donc important de choisir un taux d'apprentissage optimal qui permet de se déplacer vers le minimum rapidement.

Nous venons de voir le cas où nous n'avons que deux paramètres : w_i et b_j. Cependant, il est possible d'avoir des millions de paramètres. La descente du gradient peut s'étendre à ces cas et les formules précédentes peuvent être utilisées. Cependant, il faut souligner que les calculs augmentent avec la dimension. Dans le cas d'un réseau de neurones multi-couches général, on a donc comme précédemment :

$$\Delta v = -\eta \nabla C$$

avec $\Delta v = \begin{bmatrix} \Delta w_{jk}^{(l)} \\ \dots \\ \Delta b_j^{(l)} \\ \dots \end{bmatrix}$ et $\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_{jk}^{(l)}} \\ \dots \\ \frac{\partial C}{\partial b_j^{(l)}} \\ \dots \end{bmatrix}$ où j est le neurone, k l'entrée et l la couche. Le changement de paramètre est donc :

$$\begin{cases} {w_{jk}^{(l)}}' = w_{jk}^{(l)} - \eta \frac{\partial C}{\partial w_{jk}^{(l)}} \\ {b_j^{(l)}}' = b_j^{(l)} - \eta \frac{\partial C}{\partial b_j^{(l)}} \end{cases}$$

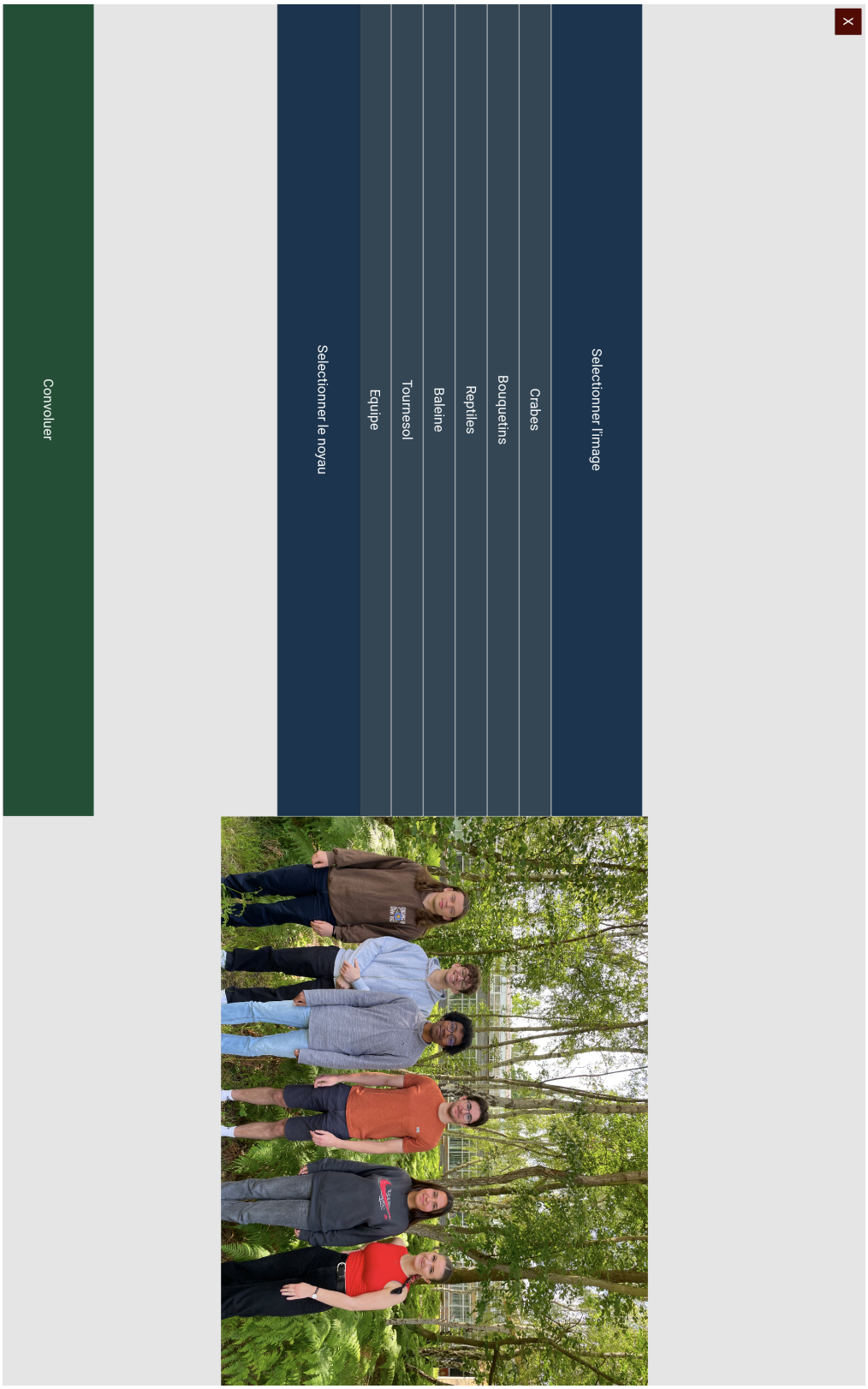
Pour calculer les gradients, on utilise la méthode des différences finies :

$$\frac{\partial C}{\partial w_{jk}^{(l)}} \approx \frac{C(w + \Delta w, b) - C(w, b)}{\Delta w_{jk}^{(l)}} \quad \text{et} \quad \frac{\partial C}{\partial b_j^{(l)}} \approx \frac{C(w, b + \Delta b) - C(w, b)}{\Delta b_j^{(l)}}$$

avec $\Delta w = \begin{bmatrix} 0 \\ \dots \\ \Delta w_{jk}^{(l)} \\ \dots \\ 0 \end{bmatrix}$ et $\Delta b = \begin{bmatrix} 0 \\ \dots \\ \Delta b_j^{(l)} \\ \dots \\ 0 \end{bmatrix}$

Le problème de la descente du gradient est qu'il faut répéter les calculs de différence finie autant de fois qu'il y a de paramètres. Or ce nombre peut être très grand. L'entraînement devient alors difficile et les coûts des calculs élevés. Afin de rendre les calculs de la descente du gradient plus efficaces, on peut utiliser la rétropropagation du gradient. Il s'agit d'un algorithme d'optimisation qui permet de simplifier les calculs. En effet, grâce à la règle de la chaîne, les expressions des dérivées partielles de la fonction coût en fonction des variables ne sont plus que sommes de produits entre les valeurs d'activation et les poids. Il faut calculer les valeurs d'activation de tout le réseau de neurones à partir des poids et des biais fixés. Puis, comme l'indique son nom, la rétropropagation du gradient consiste à calculer les dérivées partielles de la dernière couche afin de calculer progressivement celles des couches précédentes.

A.5 Interface Kivy



A.6 Code FFT

```
import numpy as np
def fft(x):
    N = len(x)
    if N==1:
        return x
    pair = x[0::2]
    impair = x[1::2]
    tfd_pair = fft(pair)
    tfd_impair = fft(impair)
    tfd = np.zeros(N,dtype=np.complex64)
    W = np.exp(-1j*2*np.pi/N)
    N2 = N//2
    for n in range(N2):
        tfd[n] = tfd_pair[n]+tfd_impair[n]*W**n
    for n in range(N2,N):
        tfd[n] = tfd_pair[n-N2]+tfd_impair[n-N2]*W**n
    return tfd
```

A.7 Code convolution naïve

```
import PIL.Image as plg
import random as rd
import time

def get_colors_pixel(img, x : int, y : int):
    width, height = img.size
    assert 0 <= x <= width - 1 and 0 <= y <= height - 1, f"Invalid values:
        x={x}, y={y}"
    pixel_color = img.getpixel((x,y))
    return (pixel_color)

def create_copy_img(path_origin : str, path_copy : str):
    img_origin = plg.open(path_origin)
    img_copy = img_origin.copy()
    img_copy.save(path_copy)

def modif_pixel_img(img_name : str, tab_coord):
    img = plg.open(img_name)
    width, height = img.size
    cap = height // 10
    c2 = cap
    for i in range(height):
        if i > c2:
            print("10%")
            c2 += cap
        for j in range(width):
            img.putpixel((j,i), (tab_coord[i][j][0], tab_coord[i][j][1],
                tab_coord[i][j][2]))
    img.save(img_name)

def create_tab_autour_img(img, x, y, dim, couleur):
    height, width = len(img), len(img[0])
    pas = (dim - 1) // 2
    tab = [[] for i in range(dim)]
    cpt = -1
    for i in range(y - pas, y + pas + 1):
        cpt += 1
        for j in range(x - pas, x + pas + 1):
            if i < 0 or i >= height or j < 0 or j >= width:
                tab[cpt].append(0)
            else :
                tab[cpt].append(img[i][j][couleur])
    return tab

def create_tab_convolue(img, noyau):
    height, width = len(img), len(img[0])
    dim = len(noyau)
    new_img = [[[] for i in range(width)] for j in range(height)]
    for i in range(height):
        for j in range(width):
            for k in range(3):
```

```
            tab_autour_px = create_tab_autour_img(img, j, i, dim, k)
            convolution = convoluer_tab(tab_autour_px, noyau)
            new_img[i][j].append(convolution)
    return new_img

def create_tab_img(img):
    width, height = img.size
    tab_img = [[] for i in range(height)]
    for i in range(height):
        for j in range(width):
            tab_img[i].append(get_colors_pixel(img, j, i))
    return (tab_img)

def create_name_new_img(img_name):
    index_dot = img_name.index('.')
    name = img_name[:index_dot]
    extension = img_name[index_dot:]
    ch = name + "_convolue" + extension
    return ch

def convoluer_tab(t1, t2):
    res = 0
    for i in range(len(t1)):
        for j in range(len(t1[0])):
            res += t1[i][j] * t2[i][j]
    return int(res)

def convolution(img_name, noyau):
    deb = time.time()
    img = plg.open(img_name)
    print("open : ", time.time() - deb, "secondes")
    width, height = img.size
    new_img_name = create_name_new_img(img_name)
    print("img name : ", time.time() - deb, "secondes")
    create_copy_img(img_name, new_img_name)
    print("copy img : ", time.time() - deb, "secondes")
    tab_img = create_tab_img(img)
    print("create tab : ", time.time() - deb, "secondes")
    tab_convolue = create_tab_convolue(tab_img, noyau)
    print("tab convolue : ", time.time() - deb, "secondes")
    modif_pixel_img(new_img_name, tab_convolue)
    print("modif pixel : ", time.time() - deb, "secondes")


noyau = [[-1, -1, -1], #Detection de contours
         [-1, 8, -1],
         [-1, -1, -1]]
convolution("ciel3.png", noyau)
```

A.8 Code convolution multiprocessing

```
import numpy as np
import imageio.v2 as imageio
import time
import concurrent.futures

def create_tab_autour(tab, h, w, l, c, dim_noyau):
    half_dim = dim_noyau // 2
    tab_autour = np.zeros((dim_noyau, dim_noyau))

    start_l = max(0, l - half_dim)
    end_l = min(h, l + half_dim + 1)
    start_c = max(0, c - half_dim)
    end_c = min(w, c + half_dim + 1)

    start_sl = max(0, half_dim - l)
    end_sl = min(dim_noyau, h - l + half_dim)
    start_sc = max(0, half_dim - c)
    end_sc = min(dim_noyau, w - c + half_dim)

    tab_autour[start_sl:end_sl, start_sc:end_sc] = tab[start_l:end_l,
        start_c:end_c]

    return tab_autour

def create_tab_convolue(img_couleur, h, w, noyau, dim_noyau):
    d2 = dim_noyau//2
    tab = np.empty([h-2, w-2])
    for i in range(1, h - 1):
        for j in range(1, w - 1):
            t_autour = img_couleur[i-d2:i+d2+1, j-d2:j+d2+1]
            tab[i-1,j-1] = np.sum(noyau * t_autour)
    return tab


def convoluer(img_name, noyau):
    img = imageio.imread(img_name)
    red = img[:,:,0]
    green = img[:,:,1]
    blue = img[:,:,2]
    h = np.shape(img)[0]
    w = np.shape(img)[1]
    dim_noyau = np.shape(noyau)[0]
    colors_convoluees = []
    with concurrent.futures.ProcessPoolExecutor() as executor:
        colors = [red, green, blue]
        results = [executor.submit(create_tab_convolue, color, h, w, noyau,
            dim_noyau) for color in colors]
        for f in concurrent.futures.as_completed(results):
            colors_convoluees.append(f.result())
    img_convolue = np.dstack((colors_convoluees[0], colors_convoluees[1],
        colors_convoluees[2]))
```

```
    img_convolue = np.clip(img_convolue, 0, 255).astype(np.uint8)
    imageio.imwrite('image_convolue' + img_name[img_name.index('.'):],
        img_convolue)

if __name__ == "__main__":
    img_name = 'ciel3.png'
    M = np.array([[-1,-1,-1],[-1,8,-1],[-1,-1,-1]])
    deb = time.time()
    convoluer(img_name, M)
    print("temps total : ", time.time() - deb)
```

A.9 Code convolution `scipy`

```
import numpy as np
import matplotlib.pyplot as plt
import imageio.v2 as imageio
from scipy import signal
import time

deb = time.time()
A = imageio.imread('ciel3.png')
print("lecture de limage : ", time.time() - deb)
h = np.shape(A)[0]
w = np.shape(A)[1]

M = np.array([[-1,-1,-1],[-1,8,-1],[-1,-1,-1]])

deb = time.time()
B_red = signal.convolve2d(A[:,:,0], M, boundary='fill', mode='same')
B_green = signal.convolve2d(A[:,:,1], M, boundary='fill', mode='same')
B_blue = signal.convolve2d(A[:,:,2], M, boundary='fill', mode='same')
print("convolution des matrices 2d : ", time.time() - deb)

deb = time.time()
B = np.dstack((B_red, B_green, B_blue))
B = np.clip(B, 0, 255).astype(np.uint8)
print("reconstitution de limage : ", time.time() - deb)

B = np.clip(B,0,255)
B = B.astype(np.uint8)
deb = time.time()
imageio.imwrite('image_apres.png', B)
print("enregistrement : ", time.time() - deb)
```

A.10 Code convolution C

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

unsigned char* create_convoluted_array3d(unsigned char *color_array, int h
    , int w, float *kernel, int dim_ker)
{
    int result;
    unsigned char *result_array;
    int d2;
    int canaux;
    unsigned char px;
    float n_ker;

    result = 0;
    d2 = dim_ker / 2;
    canaux = 3;
    result_array = (unsigned char*)malloc(h * w * canaux * sizeof(unsigned
        char));
    if (result_array == NULL)
        exit(EXIT_FAILURE);
    for (int i = 0; i < h; i++)
    {
        for (int j = 0; j < w; j++)
        {
            for (int color = 0; color < canaux; color++)
            {
                result = 0;
                for (int k = -d2; k <= d2; k++)
                {
                    for (int l = -d2; l <= d2; l++)
                    {
                        if (!((i + k) < 0 || (j + l) < 0 || (i + k >= h) || (j +
                            l >= w)))
                        {
                            px = *(color_array + (i+k)*canaux*w + (j+l)*canaux +
                                color);
                            n_ker = *(kernel + l + d2 + (k + d2)*dim_ker);
                            result = result + (px * n_ker);
                        }
                    }
                }
                if (result < 0)
                    result = 0;
```

```
            if (result > 255)
               result = 255;
            *(result_array + i*canaux*w + j*canaux + color) = result;
         }
      }
   }
   return (result_array);
}

unsigned char* load_image(char img_name[], int *w, int *h, int *canaux)
{
   int nb_pixels;
   unsigned char *image_data;
   unsigned char *pixels;

   image_data = stbi_load(img_name, w, h, canaux, 3);
   nb_pixels = *w * *h * *canaux;
   pixels = (unsigned char*)malloc(nb_pixels * sizeof(unsigned char));
   if (pixels == NULL)
      exit(EXIT_FAILURE);
   memcpy(pixels, image_data, nb_pixels * sizeof(unsigned char));
   stbi_image_free(image_data);
   return (pixels);
}

void write_image(const char *nom_fichier, int w, int h, int canaux, unsigned
    char *array, int qualite)
{
   int i;

   i = 0;
   while (nom_fichier[i] != '.')
      i++;
   i++;

   if (nom_fichier[i] == 'j')
      stbi_write_jpg("camarche.jpg", w, h, canaux, array, qualite);
   else
      stbi_write_png("camarche.png", w, h, canaux, array, w*canaux);
}

void start_convolution(char img_name[], int code_ker)
{
   unsigned char *img;
   unsigned char *img_convoluted;
   int *w;
   int *h;
   int *canaux;
   int dim_ker;
   float num = 1.0/9.0;
   float un = 1.0/16.0;
   float de = 2.0/16.0;
   float qu = 4.0/16.0;
```

```
100     float *kernel;

        switch(code_ker)
        {
            case 1: {
105             float ker_tmp[] = {num,num,num,num,num,num,num,num,num}; //Flou
                dim_ker = (sizeof(ker_tmp) / sizeof(ker_tmp[0]));
                kernel = (float *)malloc(sizeof(ker_tmp));
                if (kernel == NULL)
                    exit(EXIT_FAILURE);
110             memcpy(kernel, ker_tmp, sizeof(ker_tmp));
                break;}
            case 2:{
                float ker_tmp[] = {0,-1,0,-1,5,-1,0,-1,0}; //Up contraste
                dim_ker = (sizeof(ker_tmp) / sizeof(ker_tmp[0]));
115             kernel = (float *)malloc(sizeof(ker_tmp));
                if (kernel == NULL)
                    exit(EXIT_FAILURE);
                memcpy(kernel, ker_tmp, sizeof(ker_tmp));
                break;}
120         case 3:{
                float ker_tmp[] = {-1,-1,-1,-1,9,-1,-1,-1,-1}; //Accentuation
                dim_ker = (sizeof(ker_tmp) / sizeof(ker_tmp[0]));
                kernel = (float *)malloc(sizeof(ker_tmp));
                if (kernel == NULL)
125                 exit(EXIT_FAILURE);
                memcpy(kernel, ker_tmp, sizeof(ker_tmp));
                break;}
            case 4:{
                float ker_tmp[] = {-1,0,1,-2,0,2,-1,0,1}; //contours hor
130             dim_ker = (sizeof(ker_tmp) / sizeof(ker_tmp[0]));
                kernel = (float *)malloc(sizeof(ker_tmp));
                if (kernel == NULL)
                    exit(EXIT_FAILURE);
                memcpy(kernel, ker_tmp, sizeof(ker_tmp));
135             break;}
            case 5:{
                float ker_tmp[] = {-1,-2,-1,0,0,0,1,2,1}; //contours ver
                dim_ker = (sizeof(ker_tmp) / sizeof(ker_tmp[0]));
                kernel = (float *)malloc(sizeof(ker_tmp));
140             if (kernel == NULL)
                    exit(EXIT_FAILURE);
                memcpy(kernel, ker_tmp, sizeof(ker_tmp));
                break;}
            case 6:{
145             float ker_tmp[] = {un, de, un, de, qu, de, un, de, un}; //flou
                    gaussien
                dim_ker = (sizeof(ker_tmp) / sizeof(ker_tmp[0]));
                kernel = (float *)malloc(sizeof(ker_tmp));
                if (kernel == NULL)
                    exit(EXIT_FAILURE);
150             memcpy(kernel, ker_tmp, sizeof(ker_tmp));
                break;}
```

```
        case 7:{
            float ker_tmp[] = {-2,-1,0,-1,1,1,0,1,2}; //emboss
            dim_ker = (sizeof(ker_tmp) / sizeof(ker_tmp[0]));
            kernel = (float *)malloc(sizeof(ker_tmp));
            if (kernel == NULL)
                exit(EXIT_FAILURE);
            memcpy(kernel, ker_tmp, sizeof(ker_tmp));
            break;}
        case 8:{
            float ker_tmp[] = {0,-1,0,-1,5,-1,0,-1,0}; //sharpen
            dim_ker = (sizeof(ker_tmp) / sizeof(ker_tmp[0]));
            kernel = (float *)malloc(sizeof(ker_tmp));
            if (kernel == NULL)
                exit(EXIT_FAILURE);
            memcpy(kernel, ker_tmp, sizeof(ker_tmp));
            break;}
        case 9:{
            float ker_tmp[] = {1,0,-1,0,0,0,-1,0,1}; // edge
            dim_ker = (sizeof(ker_tmp) / sizeof(ker_tmp[0]));
            kernel = (float *)malloc(sizeof(ker_tmp));
            if (kernel == NULL)
                exit(EXIT_FAILURE);
            memcpy(kernel, ker_tmp, sizeof(ker_tmp));
            break;}
        case 0:{
            float ker_tmp[] = {0,1,0,1,-4,1,0,1,0}; //laplace
            dim_ker = (sizeof(ker_tmp) / sizeof(ker_tmp[0]));
            kernel = (float *)malloc(sizeof(ker_tmp));
            if (kernel == NULL)
                exit(EXIT_FAILURE);
            memcpy(kernel, ker_tmp, sizeof(ker_tmp));
            break;}
        default:{
            float ker_tmp[] = {-1,-1,-1,-1,8,-1,-1,-1,-1}; //contours
            dim_ker = (sizeof(ker_tmp) / sizeof(ker_tmp[0]));
            kernel = (float *)malloc(sizeof(ker_tmp));
            if (kernel == NULL)
                exit(EXIT_FAILURE);
            memcpy(kernel, ker_tmp, sizeof(ker_tmp));}

    }

    if (dim_ker == 9)
        dim_ker = 3;
    if (dim_ker == 25)
        dim_ker = 5;

    w = (int*)malloc(sizeof(int));
    h = (int*)malloc(sizeof(int));
    canaux = (int*)malloc(sizeof(int));
    if (w == NULL || h == NULL || canaux == NULL)
        exit(EXIT_FAILURE);
```

```
    img = load_image(img_name, w, h, canaux);
    img_convoluted = create_convoluted_array3d(img, *h, *w, kernel, dim_ker
        );
    write_image(img_name, *w, *h, *canaux, img_convoluted, 100);

    free(kernel);
    free(img);
    free(w);
    free(h);
    free(canaux);
    free(img_convoluted);
}

void test(char *a)
{
    printf("%c\n",a[1]);
    printf("%s\n", a);
}

int main(int argc, char **argv){
    clock_t debut, fin;
    double temps_execution;
    srand(time(NULL));

    debut = clock();
    if (argc == 3)
        start_convolution(argv[1], atoi(argv[2]));
    else
        printf("Pas le bon nombre d'argument\n");
    fin = clock();
    temps_execution = ((double)(fin - debut)) / CLOCKS_PER_SEC;
    printf("Temps d'execution : %.3f secondes\n", temps_execution);
}
```

Image originale

Accentuation des contours

Flou gaussien

Détection des bords

A.11 Code interface Tkinter

```
from tkinter import *
from tkinter import filedialog as fd
from PIL import Image, ImageTk, ImageFilter
import os
import time

def afficher_image(nom_image):
    global chemin_image # Ajout pour conserver le chemin de l'image
        actuelle
    chemin_image = nom_image
    image = Image.open(nom_image)
    image.thumbnail((400, 400))
    photo = ImageTk.PhotoImage(image)
    label_image.configure(image=photo)
    label_image.image = photo

def demande_nom():
    nom_image = fd.askopenfilename()
    afficher_image(nom_image)

def appliquer_filtres():
    start = time.time()
    if chemin_image: # Verifier si une image est deja chargee
            image = Image.open(chemin_image)
            if CheckVar1.get() == '1': # Flou
                image = image.filter(ImageFilter.BLUR)
            if CheckVar2.get() == '1': # Amelioration de la nettete
                image = image.filter(ImageFilter.SHARPEN)
            if CheckVar3.get() == '1': # Detection de contours
                image = image.filter(ImageFilter.FIND_EDGES)

            filtre_destination = f"{os.path.splitext(chemin_image)[0]}
                _convolueFPIL.png"
            image.save(filtre_destination)
            afficher_image(filtre_destination)
    else:
        print("Pas d'image chargee")
    temps_mis=(time.time()-start)
    print(temps_mis,"secondes")

fenetre = Tk()
fenetre.geometry('800x600')
fenetre.title('Convolution de matrices')
fenetre['bg'] = 'teal'

titre = Label(fenetre, text="Projet Scientifique Encadre : Convolution de
    matrices (v2)", font=("Helvetica", 18, "bold underline"), fg="white",
    bg="green")
titre.pack(pady=20)
```

```
label_nom = Label(fenetre, text="Choisissez l'image que vous souhaitez
    afficher :", font=("Arial", 12), fg="white", bg="teal")
label_nom.pack()

bouton_choix_image = Button(fenetre, text="Choisir une image", bg='white'
    , fg='black', command=demande_nom)
bouton_choix_image.pack(pady=10)

label_image = Label(fenetre)
label_image.pack(pady=20)

label_choix = Label(fenetre, text="Selectionnez les filtres a appliquer :
    ", font=("Arial", 12), fg="white", bg="teal")
label_choix.pack(pady=10)

CheckVar1 = StringVar()
CheckVar2 = StringVar()
CheckVar3 = StringVar()
C1 = Checkbutton(fenetre, text="Flou", variable=CheckVar1, onvalue='1',
    offvalue='0')
C2 = Checkbutton(fenetre, text="Amelioration de la nettete", variable=
    CheckVar2, onvalue='1', offvalue='0')
C3 = Checkbutton(fenetre, text="Detection de contours", variable=
    CheckVar3, onvalue='1', offvalue='0')
C1.pack(pady=5)
C2.pack(pady=5)
C3.pack(pady=5)

bouton_convoluer = Button(fenetre, text="Appliquer le/les filtre(s)", bg=
    'white', fg='black', command=appliquer_filtres)
bouton_convoluer.pack(pady=10)

chemin_image = None # Initialisation de la variable globale pour stocker
    le chemin de l'image
fenetre.mainloop()
```

A.12 Code interface `Kivy`

```
from kivy.app import App
from kivy.uix.widget import Widget
from kivy.uix.dropdown import DropDown
from kivy.uix.button import Button
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.image import Image
from kivy.uix.label import Label
from kivy.graphics import Color, Rectangle
from convo_multiprocessing import convoluer
from Stockage_Noyaux_Images import Liste_Des_Noyaux, Liste_Des_Images
import os
from kivy.core.window import Window

Window.fullscreen = 'auto'

class MainWidget(FloatLayout):

    def __init__(self, **kwargs):
        super(MainWidget, self).__init__(**kwargs)
        self.nom_image = None
        self.nom_noyau = None
        self.image_widget = None

        # Bouton de fermeture
        self.close_button = Button(text="X", size_hint=(None, None), size
            =(60, 60), pos=(10, Window.height - 50), background_color=(1, 0,
             0, 1))
        self.close_button.bind(on_release=self.close_application)
        self.add_widget(self.close_button)
        self.bind(size=self._reposition_close_button, pos=self.
            _reposition_close_button)

        # Couleur du fond
        with self.canvas.before:
            Color(0.9, 0.9, 0.9, 1) # Code pour gris clair RGBA
            self.rect = Rectangle(size=self.size, pos=self.pos)
            self.bind(size=self._update_rect, pos=self._update_rect)

        main_layout = BoxLayout(orientation='horizontal')

        # Creation de la liste deroulante pour l'image (depuis fichier de
            stockage)
        dropdownImage = DropDown()
        for clefs in Liste_Des_Images.keys():
            btn = Button(text='%s' % clefs, size_hint_y=None, height=70,
                background_color=(0.53, 0.81, 0.98, 1))
            btn.bind(on_release=lambda btn: dropdownImage.select(btn.text))
            dropdownImage.add_widget(btn)
        mainbuttonImage = Button(text="Selectionner l'image", size_hint=(1,
             None), height=200, background_color=(0.2, 0.6, 0.9, 1))
```

```
        mainbuttonImage.bind(on_release=dropdownImage.open)
        dropdownImage.bind(on_select=self.load_image)

        # Creation de la liste deroulante du noyau (depuis fichier de
            stockage)
        dropdownNoyau = DropDown()
        for clefs in Liste_Des_Noyaux.keys():
            btn = Button(text='%s' % clefs, size_hint_y=None, height=70,
                background_color=(0.53, 0.81, 0.98, 1))
            btn.bind(on_release=lambda btn: dropdownNoyau.select(btn.text))
            dropdownNoyau.add_widget(btn)
        mainbuttonNoyau = Button(text="Selectionner le noyau", size_hint
            =(1, None), height=200, background_color=(0.2, 0.6, 0.9, 1))
        mainbuttonNoyau.bind(on_release=dropdownNoyau.open)
        dropdownNoyau.bind(on_select=self.Selection_noyau)

        # Creation du bouton de convolution
        mainbuttonConvolution = Button(text="Convoluer", size_hint=(1, None
            ), height=200, background_color=(0.2, 0.9, 0.6, 1))
        mainbuttonConvolution.bind(on_release=self.convoluer_afficher)

        # Layout pour les boutons (partie gauche de l'ecran)
        layout_buttons = BoxLayout(orientation='vertical', spacing=400,
            size_hint=(1, 1))
        layout_buttons.add_widget(mainbuttonImage)
        layout_buttons.add_widget(mainbuttonNoyau)
        layout_buttons.add_widget(mainbuttonConvolution)

        # Layout pour l'image (partie droite de l'ecran)
        self.layout_image = BoxLayout(size_hint=(0.7, 1))
        self.image_container = BoxLayout(orientation='vertical')
        self.layout_image.add_widget(self.image_container)

        # Ajout des widgets au layout principal
        main_layout.add_widget(layout_buttons)
        main_layout.add_widget(self.layout_image)

        # Ajout du layout principal au FloatLayout
        self.add_widget(main_layout)

    def convoluer_afficher(self, instance):
        if self.nom_image and self.nom_noyau:
            convoluer(self.nom_image, self.nom_noyau)
            image_convoluee_source = "image_convolue_%s_%s.jpeg" % (self.
                nom_noyau, self.nom_image)
            if os.path.exists(image_convoluee_source):
                if self.image_widget:
                    self.image_container.remove_widget(self.image_widget) #
                        Enlever l'image precedente
                self.image_widget = Image(source=image_convoluee_source)
                self.image_container.add_widget(self.image_widget)
        else:
            print("Image ou noyau non selectionne(s).")
```

```
    def load_image(self, instance, x):
        self.nom_image = x
        if self.image_widget:
            self.image_container.remove_widget(self.image_widget) # Enlever
                l'image precedente
        image = Image(source=Liste_Des_Images[x])
        self.image_container.add_widget(image)
        self.image_widget = image

    def Selection_noyau(self, instance, x):
        self.nom_noyau = x

    def _update_rect(self, instance, value):
        self.rect.pos = self.pos
        self.rect.size = self.size

    def _reposition_close_button(self, instance, value):
        self.close_button.pos = (10, self.height - self.close_button.height
            - 10)

    def close_application(self, instance):
        App.get_running_app().stop()

class MainApp(App):

    def build(self):
        return MainWidget()

if __name__ == "__main__":
    MainApp().run()
```

A.13 Code stockage noyaux et images

```
import numpy as np


Liste_Des_Noyaux = {
    'Flou': np.array([
        [1/9, 1/9, 1/9],
        [1/9, 1/9, 1/9],
        [1/9, 1/9, 1/9]
    ])  , # Normalisation du noyau

    'Augmentation du contraste': np.array([
        [0, -1, 0],
        [-1, 5, -1],
        [0, -1, 0]
    ]),

    'Accentuation': np.array([
        [-1, -1, -1],
        [-1, 9, -1],
        [-1, -1, -1]
    ]),

    'Detection des contours horizontaux (Sobel)': np.array([
        [-1, 0, 1],
        [-2, 0, 2],
        [-1, 0, 1]
    ]),

    'Detection des contours verticaux (Sobel)': np.array([
        [-1, -2, -1],
        [0, 0, 0],
        [1, 2, 1]
    ]),

    'Flou gaussien': np.array([
        [1, 2, 1],
        [2, 4, 2],
        [1, 2, 1]
    ]) / 12, # Normalisation du noyau

    'Emboss': np.array([
        [-2, -1, 0],
        [-1, 1, 1],
        [0, 1, 2]
    ]),

    'Repoussage (Sharpen)': np.array([
        [0, -1, 0],
        [-1, 5, -1],
        [0, -1, 0]
    ]),
```

```
    'Bordure (Edge Detection)': np.array([
        [1, 0, -1],
        [0, 0, 0],
        [-1, 0, 1]
    ]),

    'Laplace': np.array([
        [0, 1, 0],
        [1, -4, 1],
        [0, 1, 0]
    ])
}
Liste_Des_Images = {'Crabes':'Image_1.jpeg','Bouquetins':'Image_2.jpeg','
    Reptiles':'Image_3.jpeg'
                ,'Baleine':'Image_4.jpeg','Tournesol':'Image_5.webp','
                    Equipe':'Image_6.jpg'}
```