Première Soutenance OCR

Grégoire Suissa Malo Legendre-Lemaire Anthony Caron
Romain Dubois

10 November 2022

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Table des matières

1 Introduction

1.1 Présentation du groupe

1.1.1 Composition du groupe

Notre groupe OCR est composé de 4 personnes, Romain Dubois, Malo Legendre, Anthony Caron ainsi que Grégoire Suissa.

1.1.2 Répartition des tâches

Pour ce qui est de la répartition des tâches Romain et Malo se sont focalisés sur le traitement de l'image. D'un autre côté, Grégoire et Anthony ont travaillé sur la partie concernant le réseau de neurones.

1.2 Objectif fixés

1.2.1 Objectif obligatoire

Pour la première soutenance, nous avons terminé tous les objectifs :
— Chargement d'une image et suppression des couleurs
— Rotation manuelle de l'image
— Détection de la grille et de la position des cases
— Découpage de l'image
— Implémentation de l'algorithme de résolution d'un sudoku
— Preuve du fonctionnement du réseau de neurone, avec un mini réseau capable d'apprendre la fonction OU EXCLUSIF

1.2.2 Objectif personnel

En plus de ces objectifs, nous voulions terminer l'ensemble du projet OCR avec un réseau de neurones ainsi que le traitement de l'image entièrement fonctionnel mais pas forcément optimisé.

2 Les systèmes utilitaires

2.1 Le 'Makefile' - ou système de compilation

Pour ce projet ce qui n'est traditionelement qu'un simple makefile d'une dixaine de ligne est ici un système de compilation complexe permettant une agilité de développement plus que désirable.

2.1.1 Libraries et executables

Un des majeurs problème avec la façon dont d'autres groupes se répartissent le travail est que leurs projet tout entier est compilé en un seul et même exécutable, ce qui fais qu'il est impossible de pouvoir efficacement travailler parallèlement sur des parties différentes tout en pouvant les tester. La solution a ce problème pourrait d'utiliser un sous-dossier par sous-partie logique du projet, et de pour chaque sous-partie utiliser une compilation différente avec un Makefile différent pour chacune. Mais en faisant ainsi, le problème n'est que déplacé. Le problème existant durant le développent va maintenant poser un problème quand chaque sous parties devront être réunies pour former le projet sous sa forme final.
C'est en raison des problèmes citées précédemment que la solution que nous avons décidé d'utiliser est de séparer notre projet en plusieurs sous parties toutes compilée grâce au même Makefile, et donc avec les mêmes paramètres sur les différentes étapes de compilations. Chaques sous parties sont donc maintenant appelées des *targets* (anglais pour cible). Une *target* est donc au minimum définie d'un dossier dans lequel se trouve le code source, ainsi que comment cette *target* devrait être compilée.
C'est pour cela qu'il existe 2 type principaux de *targets* possible :

— Les librairies statique
— Les exécutables

La seul différence entre les deux étant qu'un exécutable seras compilé en un fichier binaire exécutable alors qu'une librairie statique est compilé en fichier de librairie statique (fichier .a n'étant qu'un fichier d'archive contenant tout les fichier objets de la librarie).

2.1.2 Les dépendances

S'il est totalement possible de n'utiliser que des exécutables pour coder l'ocr en entier on y perd tout l'intérêt de ce système ! Un moyen de pouvoir utiliser les libraries a partir des executable ou même entre libraries est donc recquis.
Nous allons donc définir un autre paramètre pour chaques *targets*, les dépendances (ou *depedencies* en anglais), une liste d'autres *target* recquise pour pouvoir compiler une *target* précise.
Afin d'illustrer ce principe, imaginons une *target* de type librairie appelé "matrice". Ainsi qu'une seconde *target* de type librairie appelé "neural-network". Précisons maintenant que "neural-network" a comme dépendance la librairie "matrice". Le système de compilation vas donc maintenant toujours compiler la librairie "matrice" avant de compiler "neural-network". On peut donc avoir un réseau complexe de *targets* dépendantes entre elles.

2.1.3 Utilisation concrète

Pour reprendre notre exemple de la partie précédente, nous pouvons définir les *targets* recquise en ajoutant dans le Makefile :

```
# La target 'neural-network', le type etant librarie par defaut.
# Le chemin du dossier contenant le code source de cette libraries et aussi trouve par
    defaut comme etant 'libs/neural-network'.
$(guile (define-target "neural-network" \
    # Une liste de dependances, ici uniquement matrices
    `(deps "matrices") \
))

# La target 'matrices'
$(guile (define-target "matrices"))
```

Pour utiliser cette librairie nous pouvons ajouter un exécutable ayant comme dépendance ces deux librairies :

```
# Ici le chemin du dossier contenant le code source de cet executable est 'executables/ocr'
$(guile (define-target "ocr" \
    `(target-type "executable") \
```

```
    '(deps "matrices" "neural-network") \
))
```

Avec ces deux targets dans le Makefile, il est maintenant possible de compiler la *target* 'ocr' de cette manière :

```
make ocr
```

Cela aura pour effet de compiler les deux librairies ainsi que l'exécutable ocr. Le fichier binaire d'ocr peux être trouvé positionné au chemin 'build/release/ocr.out'.

2.1.4 Autres plus encode

Actuellement ce système possède déjà de nombreuses autres fonctionnalités, pouvoir compiler et exécuter des tests pour chaque *target* séparément, pouvoir facilement compiler avec des profiles de compilations différents, génération automatique de documentation avec *doxygen* et le formattage automatique du code avec *clang-format*.

2.2 Les librairies matrices & vec

Les libraries matrices et vec sont deux libraries permettant l'utilisation simplifier de structure de données "générique" autant sur la taille que sur le type. Les deux utilise un principe similaire.

2.2.1 Les pointeurs papillons

Les librairies matrices et vec utilisent un principe similaire. Les pointeurs papillons tireent leurs noms de la symétrie des ailes d'un papillon, en effet un pointeur papillon n'est pas uniquement un pointeur vers le début d'une structure comme généralement le cas, en plus d'une liste dans le cas d'un vec ou une matrice ce trouvant a droite du pointeur, une structure de metadata se trouve aussi a gauche du pointeur.
En pratique cela veux dire qu'il est possible de sauvegarder la taille d'une liste sans modifier le type qu'on aurait utilisé en C classique.

2.2.2 La librairie matrices

Pouvoir créer et manipuler des matrices a son importance dans de nombreuses parties de cet OCR, il est donc vital de pouvoir les manipuler facilement.
La librairie matrices permet la création de matrices de n'importe quelle taille et de n'importe quel type. Grâce aux pointeurs papillon les fonctions d'opérations arithmétique n'ont comme arguments uniquement des matrices, leurs taille étant trouvable a partir du pointeur.

2.2.3 La librairie vec

La librairie vec met a disposition une structure de type "LIFO" (Last In First Out) implémenté grâce a une liste statique avec une mémoire re-dimensionné au besoin. La complexité de toutes les opérations algorthmique sur une structure "LIFO" est constante en moyenne car quand la structure n'a plus de place disponible elle est recopiée dans un espace mémoire plus grand. De la même manière de la librairies matrices, la taille d'un vec peut être retrouvé grâce au pointeur seulement.

3 Parties principales

3.1 Résolution du Sudoku

3.1.1 Sudoku Solver

Pour le Solver de sudoku, nous avons utilisé un algorithme de back-tracking qui est un algorithme de brute force, c'est-à-dire qui essaye des combinaisons dans la grille, et reviens en arrière si elles ne fonctionnent pas. Pour cela, on explore complètement une branche de solution possible avant de passer à une autre branche. Cet algorithme visite donc les cellules vides du sudoku, et place une valeur aléatoire, il va ensuite regarder si cette valeur n'est pas déjà présente dans la ligne, dans sa colonne, et enfin dans son carré de 3 par 3. Si cette valeur est déjà présente dans une de ces trois étapes, alors il recommence avec une autre valeur et ainsi de suite. Si aucune valeur ne fonctionne, alors il revient d'une cellule en arrière et change la valeur de la cellule, et on répète cette opération jusqu'à ce que la 81eme case soit remplie.

3.1.2 Difficultés rencontrés

3.2 Traitement de l'image

Dans un principe pédagogique, et au final de simplicité, la bibliothèque SDL n'est jamais utilisée dans ce projet. Ce qui veux dire que tout les traitements d'image, du chargement du fichier jusqu'à la gestion de différent formats est implémenté par le projet.

3.2.1 La librairie Image

Entre le moment où l'image est chargée pour la première fois en mémoire et quand le traitement de l'image est fini, l'image vas passer par de nombreuses représentations différentes, RGB, Grayscale binaire ou même sous forme matriciel. Mais il serait préférable d'avoir une seule et même représentation d'une Image pour que chaque algorithme que nous utilisons sur les images n'ai pas a ce soucier de quel type d'image a été donné en paramètre. Sauf que pour certains type d'image, il existe des représentations dont les avantages sont tel qu'il est préférable de les utiliser au lieu d'un format "universel" qui pourrait être i.
La librairie Image ne définie donc pas un format d'image fix qui serait le même pour toutes les images, mais plutôt un moyen d'avoir une image de n'importe quel format et d'utiliser des fonctions d'operations sur des images qui sont agnostique du format actuel de l'image. Elle définie donc un enum "PixelFormat" qui contient tout les types supportées par la librairies, ainsi que des fonction du type "img_get_pixel_X" ou X est un format, ces fonctions vont faire une conversion automatique du format de l'image au format demandé.

3.2.2 Chargement de fichiers images

Le format BitMap est probablement un voir le format d'image le plus simple a décoder existant a ce jour. C'est pour cette raison que le BMP a été choisi comme seul format supporté par notre OCR pour l'instant, d'autres formats seront rajoutés au fur et à mesure.

3.3 Les algorithmes de traitement d'image

3.3.1 Une base : La Convolution

La convolution est une opération mathématique qui a su trouver son utilité dans le milieu du traitement d'image. Elle consiste a faire pour chaque pixels sur une image, une addition pondérée sur tout les pixels autour. Voir la Fig 1. Une convolution

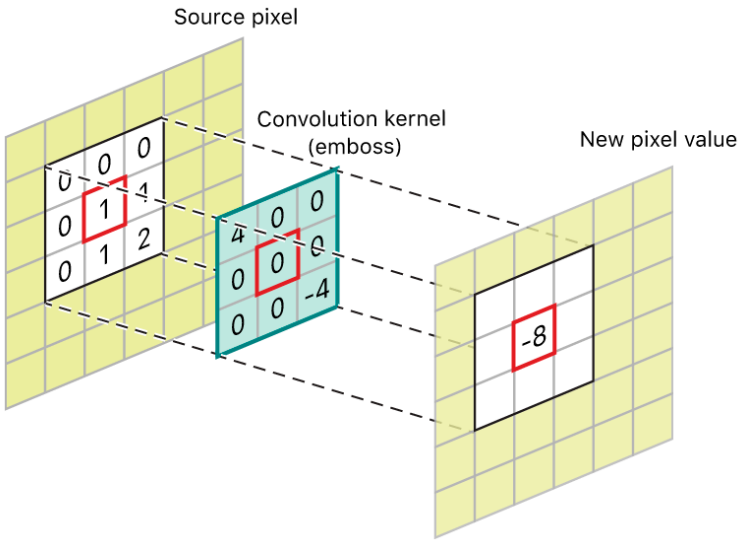

FIGURE 1 – Schéma sur la convolution d'un kernel sur une image

sur une image prend une matrice carrée de taille impaire, la convolution utilisant le centre de la matrice comme pixel d'origine.

3.3.2 Le box blur

Le box blur est un type de flou qui correspond a la convolution d'un kernel de taille n par n dont chaque valeur est mis a 1, c'est donc équivalent a faire une moyenne de n^2 pixel autour de chaque pixel. Si on compare ce flou a un flou plus standard comme un gaussian blur, le résultat peut facilement être considéré comme moche. Voir Fig 2. C'est par ce que l'intérêt du box blur n'est pas dans sa beauté mais plus dans sa complexité de calcul. En effet il possible de calculer un box blur en complexité constante par rapport a la taille du box blur. Il est donc aussi rapide de calculer un

box blur de 10x10 que un box blur de 160x160.

3.3.3 Le Flou Gaussien

Le flou gaussien est un flou basé sur une convolution suivant une loi gaussienne ou normal centré sur chaque pixel selon la formule suivante :

$$h_{(x,y)} = \frac{1}{2\pi\sigma^2} * e^{-(x^2+y^2)/(2\sigma^2)}$$

où x et y sont les coordonnées centrés sur le pixel et σ est la variance de la distribution standard à générer, plus σ est grand, plus l'image seras floue.
On peux constater que plus les coordonnées sont loin de l'origine, plus la valeur est petite, on peut même dire que la valeur réduit exponentielement. $h_{(x,y)}$ vas enfaite être tellement proche de 0 pour un x et y même relativement petit que nous allons pouvoir faire une approximation d'un flou gaussien en utilisant un kernel assez grand pour que toutes les valeurs ne se trouvant pas dedans serait de toutes façon égales a zéro.

Optimisation avec un box blur Grâce au box blur il est possible d'optimiser le flou gaussien encore plus. En effet un box blur peut être executés en temps constant, et il s'avère que lorsque que l'on applique plusieurs fois de suite un box blur a une même image, on obtient un résultat qui vas s'approcher d'un véritable flou gaussien. L'avantage ici est que l'image converge très rapidement vers un flou gaussien, surtout si la taille de chaque box blur est optimisé pour. On vas donc pouvoir exécuter un flou gaussien en temps constant.

3.3.4 Canny Edge Detector

L'algorithme de canny edge detector vas détecter et isoler les bordures de l'image qui lui est donnée, l'avantage de Canny face a d'autres algorithmes qui ont le même but est que les bordures sont réduites à des largeurs de un seul pixel ce qui vas considérablement réduire le travail des prochains algorithmes, le nombre de pixels significatif étant réduit.
L'algorithme du canny edge detector peux être séparé en plusieurs étapes :

1. Le filtre de sobel.
   Cette étape permet de calculer un gradient de l'image ainsi que des informations de direction sur le gradient. Pour voir à quoi ressemble le gradient voir la Fig 3
2. D'épaississement des lignes Cette étape est le coeur de l'algorithme, l'information de gradient et d'angle du filtre de sobel est utilisé pour détecter les dorsales maximum du gradient. Pour se faire, chaque pixel est comparé avec les 2 pixels avant et après lui même dans la direction normale au gradient. Voir la Fig 4. Afin de réduire l'impact du bruit, tous les maximums ne sont pas inclus, un *threshold* est défini au préalable, les pixels avec comme gradient une valeur en dessous du *threshold* sont ignorés et donc en noir sur l'image, et seulement les valeurs au dessus du *threshold* sont gardés.
3. Hystesis de lignes
   Grâce a cette dernière étape, d'éventuels trous dans les lignes sont reliés. Un deuxième *threshold* plus petit que le premier est utilisé pour trouver des "faux" bords, au final les "faux" bords directement connectés a des vrai bords deviennent des vrai bords, et cela récursivement jusqu'à ce que tout les bords ont étés étendus au maximum.

3.3.5 Le Flood Filling comme masque du canny edge detector

Avant de continuer sur d'autres algorithmes nous allons essayer séparer du résultat de canny, les bordures appartenant au sudoku du reste. Pour cela nous allons d'abord flouter le résultat de canny en utilisant le box blur décrit précédemment sur lequel nous appliquons un threshold global sur un valeur très basse, cela vas avoir pour effet de grossir toutes les lignes blanches crées par canny. Voyez la Fig 5 pour un exemple de résultat. Avec cette image composé de plus ou moins gros amas de pixels blanc, nous allons detecter le plus gros amas de pixels blancs connectés les uns des autres en utilisant un algorithm de "flood filling" ce qui consiste a récursivement se déplacer sur tout les pixels blancs cote à cote. Maintenant que nous avons le plus gros amas de pixels blanc, nous allons mettre tous les pixels de canny qui ne sont pas présent dans cet amas. Le résultat peut être vu sur la Fig 6.

3.3.6 Hough Transform !

Maintenant que nous avons une image ne contenant que ce qui devrait être les bords du sudoku (Voir Fig 6) nous pouvons detecter les lignes du sudoku. Pour ce faire nous avons utiliser l'algorithme nomé "Hough Transform".
Un hough transforme peut être utilisé pour détecter n'importe quel forme mathématique avec un nombre raisonnable de paramètres. Les lignes étant un exemple parfait de tel forme mathématique, nous avons tous déjà vu la représentation $ax + b = y$ d'une ligne. Mais cette représentation pose un problème, les lignes horizontal ont un paramètre a qui tend vers l'infini. La solution est d'utilisé la représentation $r = x\cos\theta + y\sin\theta$, avec r la distance entre le point le plus proche de l'origine appartenant a la ligne et l'origine de l'image ici situé au centre.
Pour utiliser hough transform nous allons pour chaque pixel de l'image d'entrée allons rajouter 1 pour chaque ligne passant par ce pixel dans l'histogram (ou hough space). Une fois l'algorithme exécuter, une suite de ligne sinusoïdal les unes sur les autres ce trouverons dans l'hough space.

3.3.7 Isolating the extreme lines

L'hough transform nous donne donc une estimation de toutes les lignes présentes dans l'image, il est dur de trouver les coins du sudoku a partir d'une dixaine de lignes, dans cette étape nous isolons donc les 4 lignes maximums.

3.3.8 Homography transform

Maintenant que nous avons nos 4 coins de l'image, nous allons devoir transformer ce quadrilatère en image carré, mais la difficultée vien qu'il faut respecter la perspective. C'est la que vient l'homographique transform qui permet de transformer en respectant la perspective.

3.4 Réseau de neurones

Nous entrons maintenant dans une partie clé de ce projet, l'intelligence artificielle qui va nous permettre de reconnaître les chiffres a travers les différents sudoku.

Pour cette première soutenance, nous avons commencé par implémenter un réseau de neurones basique. Ce réseau sera dans le futur remplacé par une IA plus fonctionnelle et optimisée.

3.4.1 Fonctionement

Structure : Fonctionnement :

- Architecture :

```
typedef struct {
    size_t layer_size;
    Matrix(double) m_bias;
    Matrix(double) m_weight;
    Matrix(double) last_output;
    Matrix(double) last_output_activated;
} Layer;
```

Layer : Notre réseau de neurones est composé de 3 classes principales. -Les layers, qui contiennent des matrices, pour les weights, les biais ainsi que pour la passe arrière

```
typedef struct {
    Layer *layers_;
    size_t layers_number;
    size_t *layers_sizes;
} neural_network;
```

Neural network : Le neural network (réseau de neurone) qui contient une liste de layer.

```
typedef struct {
    Matrix(double) input;
    Matrix(double) expect;
} DataPoint;
```

Datapoint : -Les datapoint, qui correspondent aux données que nous avons envie de faire apprendre à l'IA, correspond à une liste d' inputs (entré) et une liste d'output (sortie) qu'on est censé recevoir.

- Passe avant : La passe avant fonctionne donc totalement à l'aide des matrices. Avec l'aide d'une certaine formule, après avoir multiplié et additionné les différentes matrices nous obtenons les résultats que le réseau de neurone a calculé. Evidemment, au premier lancement de celui-ci, les valeurs obtenues en retour ne sont pas convenables car l'IA n'a pas encore été entraînée.

- Passe arrière : La passe arrière est la fonctionnalité la plus importante sur une IA , elle permet de faire apprendre les neurones pour qu'ils rendent le résultat attendu. La capture d'écran au-dessus est le résultat de l'apprentissage du réseau de neurones.

3.4.2 Fonction XOR

Afin de réaliser cette fonction XOR, nous avons décidé de créer un très petit réseau de neurones, ce qui convient parfaitement à la création d'un XOR qui n'a besoin que de très peu de neurones et de calcul, au contraire de notre objectif final de reconnaître des chiffres. Le principal objectif de ce XOR était de débuter dans le monde de l'IA, en posant des bases fonctionnelles pour au final l'améliorer. Voici les résultat obtenus :

```
inputs :
-------------
array 0 :  0.000000
array 1 :  0.000000
-------------

result : 0 with probability of 0.994372


inputs :
-------------
array 0 :  0.000000
array 1 :  1.000000
-------------

result : 1 with probability of 0.992537


inputs :
-------------
array 0 :  1.000000
array 1 :  0.000000
-------------

result : 1 with probability of 0.992122


inputs :
-------------
array 0 :  1.000000
array 1 :  1.000000
-------------

result : 0 with probability of 0.991985
```

3.4.3 Amélioration

Dorénavant, notre passe-avant est entièrement finalisée. Cependant, notre passe-arrière reste à être amélioré puisqu'elle fonctionne très bien pour un nombre de neurones assez petit comme un simple XOR mais est beaucoup trop lente sur un nombre plus imporant de neurones. C'est pourquoi, nous allons essayer d'utiliser des matrices que l'on va multiplier, transposer entre elles pour accélérer les calculs.

3.4.4 Sauvegarde

Fichier xor : Bien que la sauvegarde doit supposée être pour la seconde soutenance, elle a tout de même été implémenté. Celle-ci sauvegarde un réseau de neurones à l'intérieur d'un fichier texte sous la forme suivante. Celui correspond au réseau de neurone entraîné du XOR, nous pouvons donc ensuite le charger à partir du fichier xor.txt et l'utiliser sans avoir besoin de l'entraîner de nouveau.

ocr > xor.txt

```
2
2
5
2
-4.249467
3.752989
6.441588
-0.947242
4.848421
6.773659
-2.535326
-3.516487
1.895763
4.664156
0.876839
0.835091
0.371874
0.590621
0.401435
-2.729316
-13.694467
-6.429055
-9.261607
-2.979987
-11.774236
-6.680892
-6.912822
-21.710403
-4.061437
0.952846
0.279538
```

3.4.5 Difficultés rencontrés

La principale difficulté rencontrée à bien sûr été de découvrir le langage C. En effet, celui-ci introduit la notion de gestion de mémoire que nous n'avions pas rencontré précédemment. Débuter l'allocation de mémoire en créant une intelligence artificielle est en effet peu intuitif.

4 Conclusion

4.1 Objectifs atteints

Finalement, nous avons réussi à atteindre tous les objectifs prérequis que nous devions absolument avoir finalisé pour la première soutenance mais aussi ceux que nous voulions atteindre. Dorénavant, nous allons nous focaliser sur les bonus pour la soutenance finale.

4.2 Objectif pour la prochaine soutenance

Pour finaliser le projet, nous voulons faire un maximum de bonus pour rendre la résolution du Sudoku la plus rapide possible. Pour cela, nous chercherons à implémenter un meilleur algorithme pour améliorer les performances de la résolution. De plus, nous pourrons améliorer le traitement de l'image avec la reconnaissance de chiffres manuscrits. Mais encore, nous réaliserons probablement un site web et pourquoi pas s'attaquer à la résolution d'un hexadoku. Enfin, nous considérons que l'esthétisme joue un rôle prépondérant dans la finalisation d'un projet. C'est la raison pour laquelle nous réfléchirons à une meilleure interface durant la résolution du sudoku ainsi qu'une fois le sudoku entièrement réalisé.

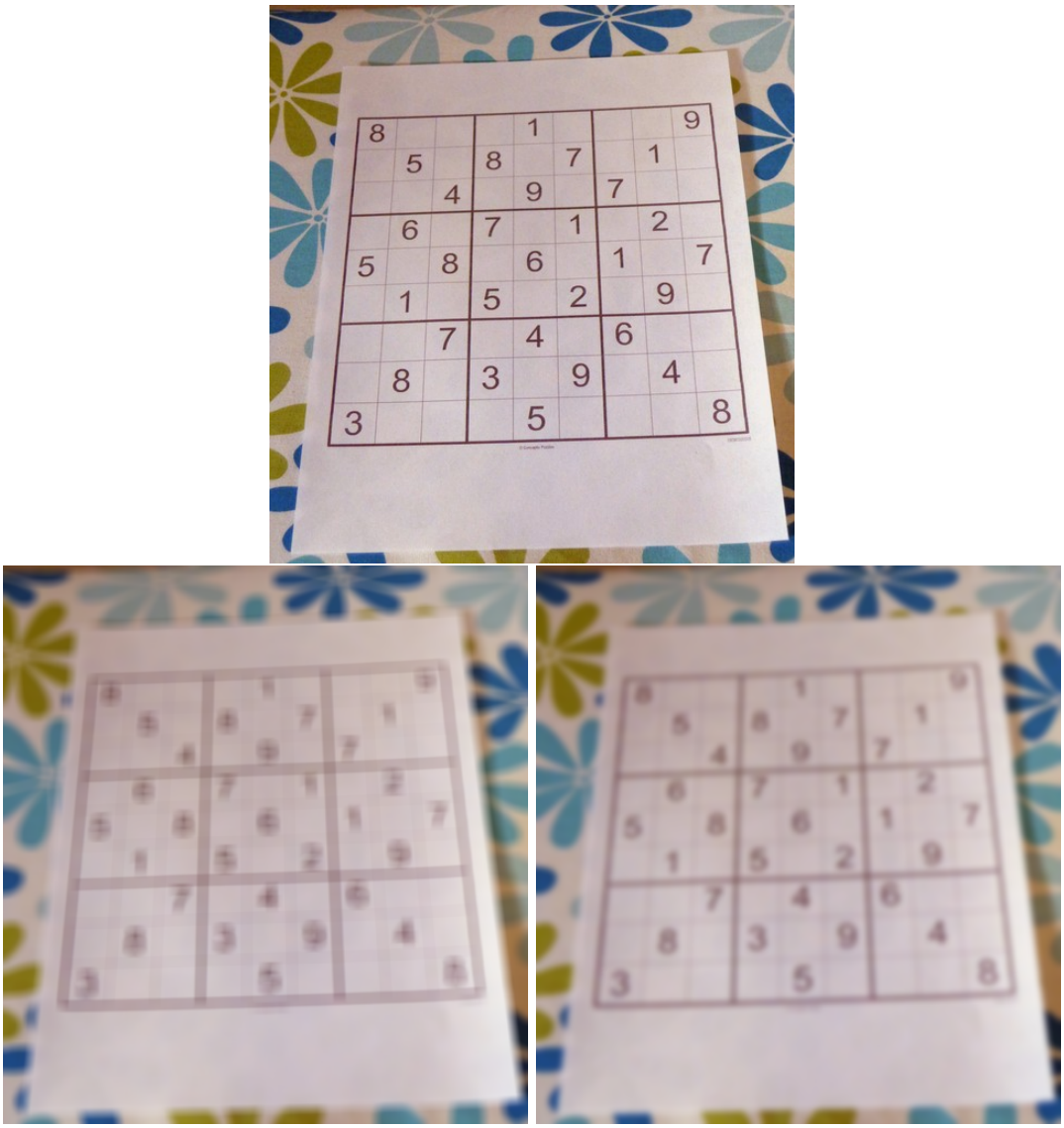

FIGURE 2 – Comparaison entre un box blur (gauche) et un gaussian blur (droite)

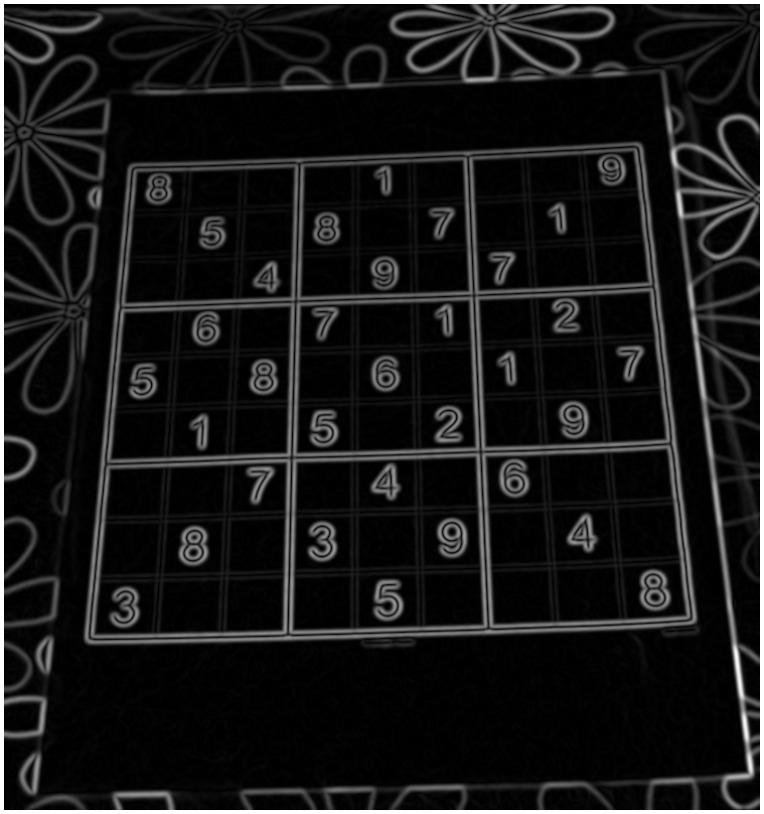

FIGURE 3 – Résultat du Sobel Operator

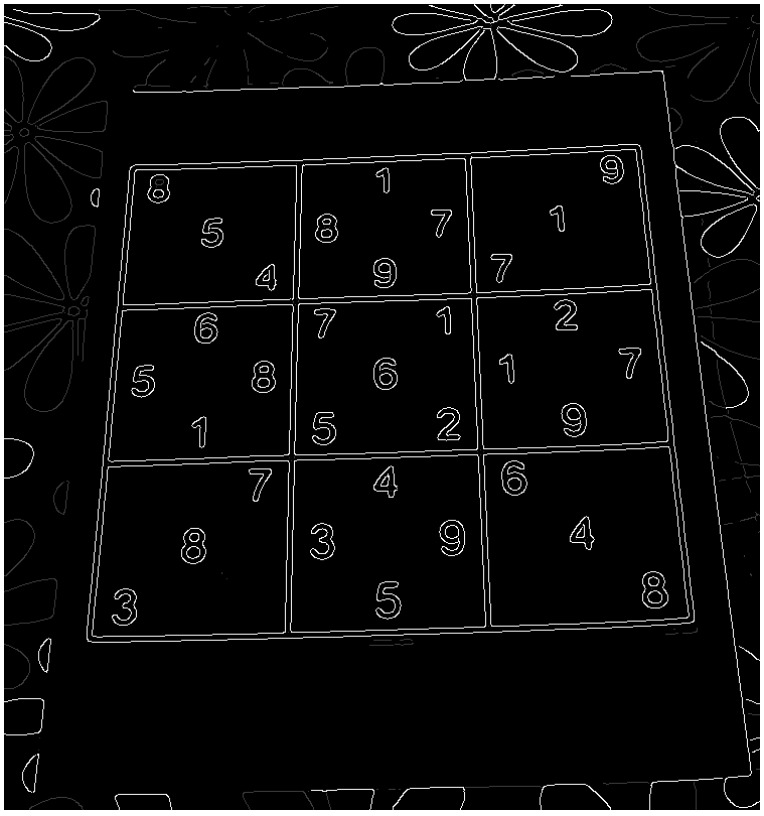

FIGURE 4 – Résultat de Canny edge detector

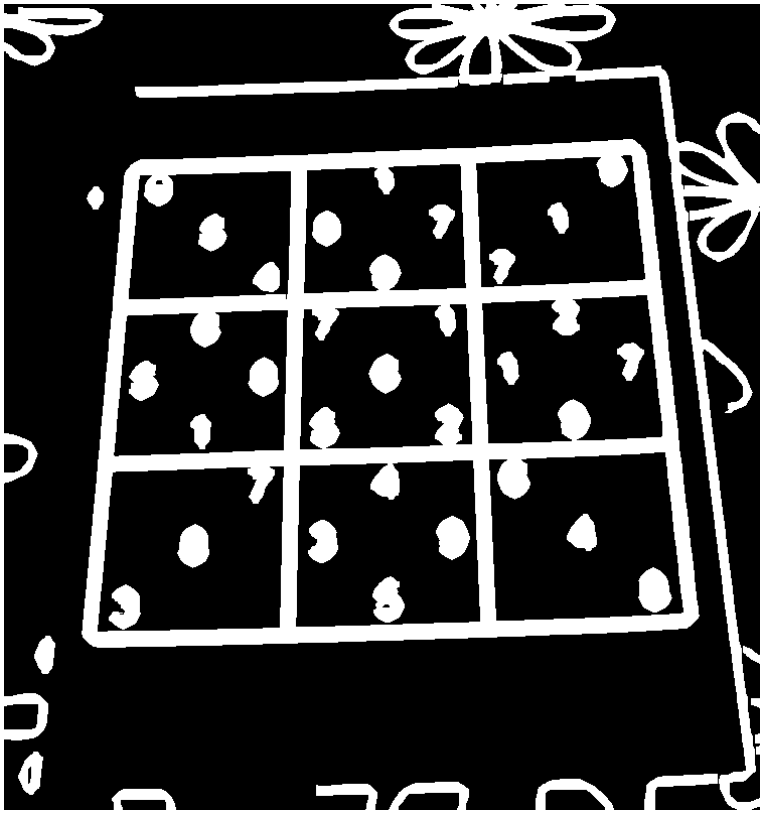

FIGURE 5 – Résultat de canny flouté et thresholdé

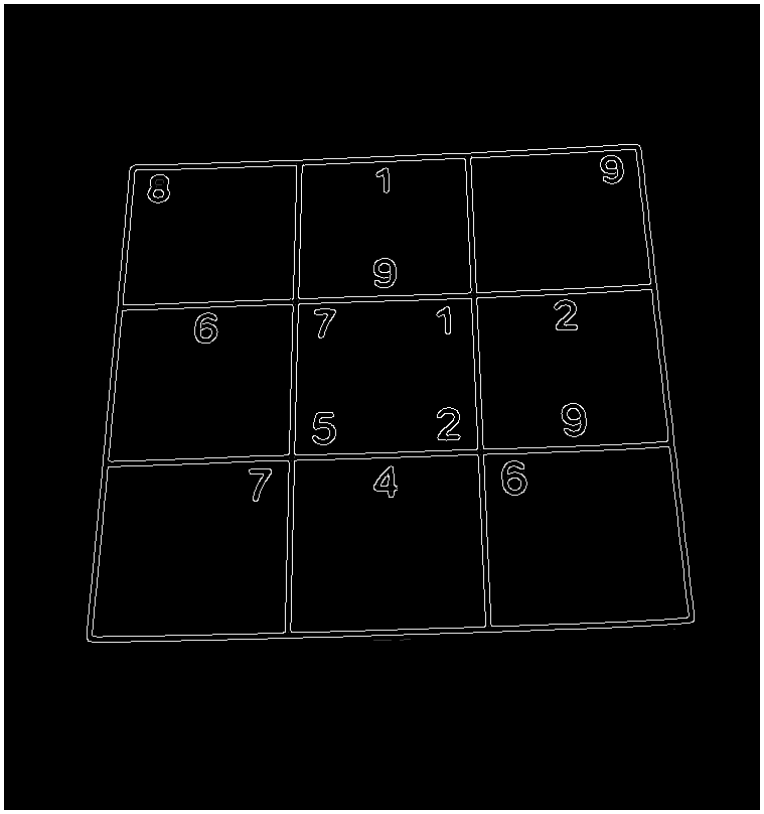

FIGURE 6 – Canny après avoir appliqué le mask