

SudOCRu

Rapport de Projet (S3)

Sofiane Meftah (Chef de projet)

Kévin Jamet

Mehdi El Alaoui El Aoufoussi

Hamza Jamai

Novembre 2022

Table des matières

1 Présentation du groupe	2
1.1 Sofiane	2
1.2 Kévin	2
1.3 Mehdi	3
1.4 Hamza	3
2 Organisation générale	4
2.1 Répartition des tâches	4
2.2 Etat d'avancement	4
3 Prétraitement	5
3.1 Niveaux de gris	5
3.2 Filtre médian	5
3.3 Binairisation	6
3.4 Détection des contours	7
3.4.1 Intensité du Gradient	7
3.4.2 Réduction des bords	8
4 Détection de la grille	9
4.1 Transformée de Hough	9
4.1.1 Hough Space	9
4.1.2 Extraction des lignes	10
4.1.3 Réperer les lignes parallèles	10
4.2 Extraction de la grille	11
4.3 Découpage des cases	11
5 Résolution des Sudokus	13
5.1 Backtracking	13
5.2 Structure et Traitement	14
5.3 Optimisation : Bit Masks	15
5.4 Rotation Manuelle	16
6 Réseaux de neurones	17
6.1 Principe du réseau de neurones	17
6.2 Reconnaissance de chiffres	18
7 Conclusion	19

Chapitre 1

Présentation du groupe

1.1 Sofiane

Je m'appelle Sofiane Meftah et sans grande surprise... je suis passionné d'informatique (et d'aviation). Cela fait assez longtemps que je touche à de l'informatique mais pas seulement. À mes débuts, au collège, je m'intéressais beaucoup à l'électronique, je possédais une carte Arduino et avec mes petits composants je réalisais des petits circuits. J'avais même investi dans un petit écran TFT ! Mais mon aventure s'est de plus en plus dirigée vers la programmation, c'est alors que je me suis intéressé à comment programmer des serveurs Minecraft qui étaient très populaires à l'époque. C'est grâce à cela que j'ai entamé l'apprentissage de mon premier langage : Java. Plus tard, au début du lycée, j'ai appris les langages du web et à me servir simplement des bases de données, puis j'ai appris le Python avec la NSI. À la fin du lycée et avec le confinement j'ai décidé de créer mon premier gros projet : un site de partage d'animes (série d'animation japonaise) avec une application Android. Je précise que j'avais décidé de commencer ce projet sans aucune connaissance dans ces domaines ! En effet, j'aime bien me lancer des défis car c'est ce qui va me forcer à m'améliorer et apprendre davantage.

1.2 Kévin

Bonjour, je m'appelle Kévin JAMET. Tout comme mon chef de projet, je suis un passionné d'informatique, mais aussi de sciences et un peu de philosophie. Dès le collège, mon esprit curieux m'a permis de découvrir plein de domaines de l'informatique et de faire plein de projets personnels dont des projets de designs et de maintenance, de montage et d'autres projets de groupe comme CubeTools, un explorateur de fichier. J'essaie toujours de me surpasser et de découvrir de nouvelles choses, ce projet est alors l'occasion parfaite pour continuer sur cette lancée ! Etant passionné de design et d'IA, j'ai décidé de me charger de ces parties en plus du Solver pour m'améliorer davantage en algorithmie.

1.3 Mehdi

Je m'appelle Mehdi El Alaoui El Aoufoussi. J'ai commencé à utiliser un ordinateur de manière autonome depuis que je suis assez jeune. Vers la moitié du collège, j'ai découvert la programmation et j'ai commencé à m'y intéresser. J'ai tout d'abord commencé avec le Java, principalement pour créer des mods et plugins pour Minecraft. Je suis principalement resté sur ce langage de programmation, mais en m'écartant du contexte de Minecraft jusqu'à mon arrivée à Epita. J'ai bien évidemment touché à quelques autres langages tel que le C++, C#, HTML/CSS, Python ou encore PostgreSQL. En somme j'ai commencé avec l'orienté objet. Ce qui est nouveau pour moi à travers ce projet, c'est le fait de le programmer en C, qui n'est pas un langage orienté objet. Il m'a fallu donc utiliser une nouvelle syntaxe et surtout penser différemment afin de concevoir mon code et mes algorithmes. Dans ce projet, j'ai été chargé de développer l'OCR afin de reconnaître des chiffres à partir d'une image.

1.4 Hamza

Je suis un grand fan de nouvelles technologies, la preuve je n'ai jamais raté une seule conférence d'Apple (oui je sais ça fait un peu snob). Le monde des nouvelles technologies est un monde sans fin où la technique et l'imagination sont reines, c'est dans ce monde où j'ai grandi, où j'ai évolué ou je me suis forgé. Lorsque j'ai pris connaissance du projet de S3 j'étais excité et motivé à travailler dessus tout, j'ai ressenti la même sensation que pour le projet de S2, créer un programme de A à Z est un rêve qui devient enfin réalité, j'aurais l'occasion et la chance de vivre cette incroyable expérience. Cette dernière, pour ma part est un avant-goût du monde professionnel qui m'aidera à m'améliorer et à développer de nouvelles compétences essentielles au métier d'ingénieur en informatique. Dans ce projet, j'ai pris en charge le traitement d'image, un domaine excitant tout à fait nouveau pour moi. Au début j'étais assez désorienté sur le processus global et les filtres à utiliser, mais après des recherches approfondies, j'ai eu les idées beaucoup plus claires.

Chapitre 2

Organisation générale

2.1 Répartition des tâches

La répartition des tâches est plutôt simple : chacun travaille sur la partie qui l'intéresse le plus et sur quoi il a envie de travailler.

Tâches	Sofiane	Kévin	Mehdi	Hamza
Prétraitement des images				Responsable
Détection de la grille	Responsable			Suppléant
Extraction des cases	Responsable		Suppléant	
Réseaux de neurones		Suppléant	Responsable	
Résolution des Sudokus		Responsable		
Application et Site Web	Suppléant	Responsable		

2.2 Etat d'avancement

Tâches	1ère Soutenance	Soutenance Finale
Prétraitement	80%	100%
Détection de la grille	90%	100%
Extraction des cases	100%	100%
Réseau de neurones	50%	100%
Résolution des Sudokus	100%	100%
Application	10%	100%
Site Web	0%	100%

Chapitre 3

Prétraitement

Le but du prétraitement de est de mettre en valeur la grille de Sudoku pour ensuite extraire les cases. Pour cela, nous allons dans un premier temps supprimer les couleurs et augmenter le contraste. Dans un deuxième temps, nous utiliserons des filtres qui permettent de réduire le bruit et les parasites dans l'image pour pouvoir éclaircir les écritures. Enfin, on binairisera l'image et on appliquera une détection des contours.

3.1 Niveaux de gris

La première étape est de convertir l'image d'entrée en une image en niveaux de gris. Nous n'avons pas vraiment besoin des couleurs pour la détection de la grille et cela nous permet d'alléger le traitement de l'image. Pour cela, on va utiliser la formule de luminosité pour traduire chaque composant en niveau de gris de manière naturel : le rouge et le bleu n'ont pas le même effet sur la luminosité. Pour chaque pixel, on va alors calculer sa valeur de luminosité :

$$L = 0.299 * R + 0.587 * G + 0.114 * B$$

Pour recomposer l'image, on va alors assigner la valeur des 3 composantes à la nouvelle valeur L . Parallélairement à la conversion en niveaux de gris, nous allons déterminer la minimale et maximale pour pouvoir normaliser les valeurs de tous les pixels. La valeur finale des pixels après l'équilibrage des niveaux est :

$$L = 255 * ((L - min) / (max - min))$$

3.2 Filtre médian

La deuxième étape est le filtre médian qui permet de réduire le bruit tout en gardant les contours nets. Nous avons opté pour celui ci en faveur du flou gaussian (à cette étape dans le traitement) car le flou gaussian ou filtre moyen a l'effet inverse du filtre médian : il arrondi les contours tout en réduisant le bruit. Ce n'est pas ce qu'on veut, on doit à tout prix préserver les contours les plus nets possibles pour la détection des contours.

Le principe du filtre médian est de parcourir l'image avec une fenêtre (kernel) de 3 par 3 pixels autour du pixel actuel. La nouvelle valeur du pixel sera alors la médiane de la liste (triée en ordre croissant) des valeurs des pixels voisins.

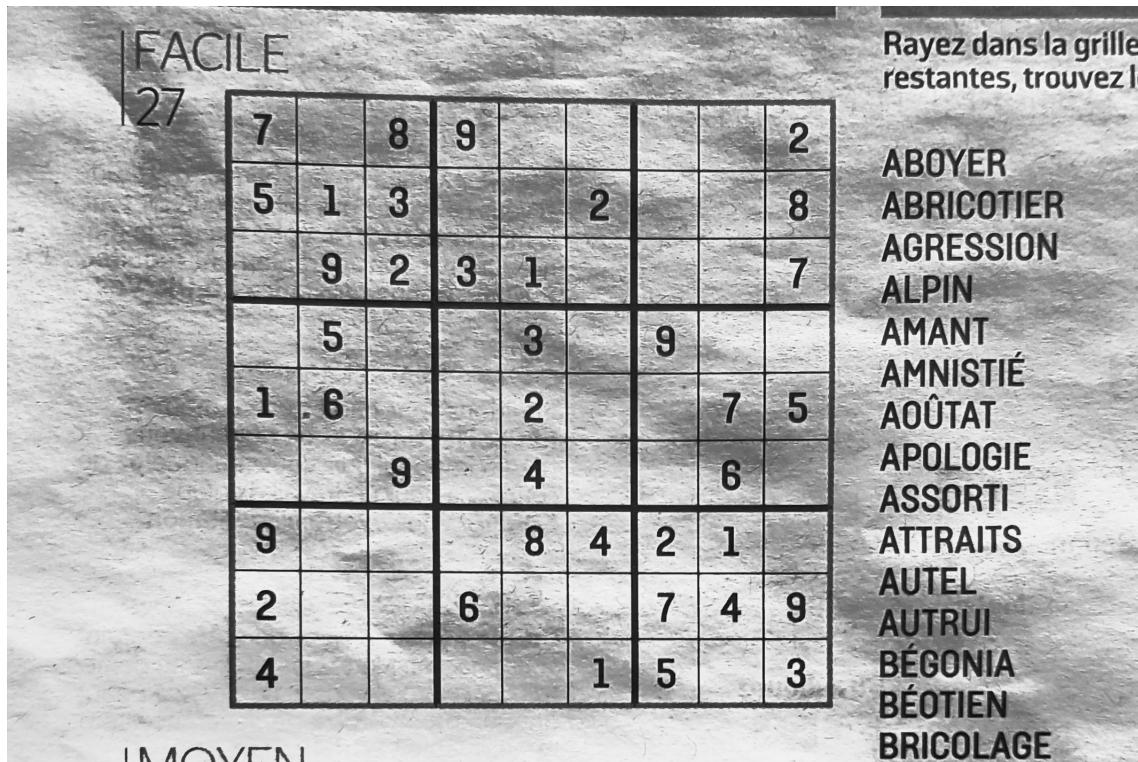


FIGURE 3.1 – Image n°04 après la mise en niveaux de gris et filtre médian

3.3 Binairisation

La dernière étape avant la détection des contours est la binairisation : transformer notre image en des 0 et des 1... littéralement. Tous les pixels vont être séparés en deux catégories : considéré comme sombre ou considéré comme clair. Pour cela, nous avons utilisé la méthode d’Otsu. Celle-ci consiste à, depuis un histogramme des niveaux de gris à cette étape, déterminer la valeur du seuil qui minimise la variance intra-classe (qui va diviser les niveaux de gris en deux classes) tel que :

$$\sigma_w^2(t) = \omega_0(t)\sigma_0^2(t) + \omega_1(t)\sigma_1^2(t)$$

Où t représente l'axe x de l'histogramme (l'axe y étant le nombre de pixels ayant pour valeur t), ω_0 est la somme des niveaux de gris de 0 jusqu'à un certain t et $\omega_1 = total - \omega_0$ (la somme de t à la fin). Une fois le seuil t déterminé, tous les pixels ayant leur valeur en dessous du seuil seront considérés comme sombres et ceux au dessus comme claires.

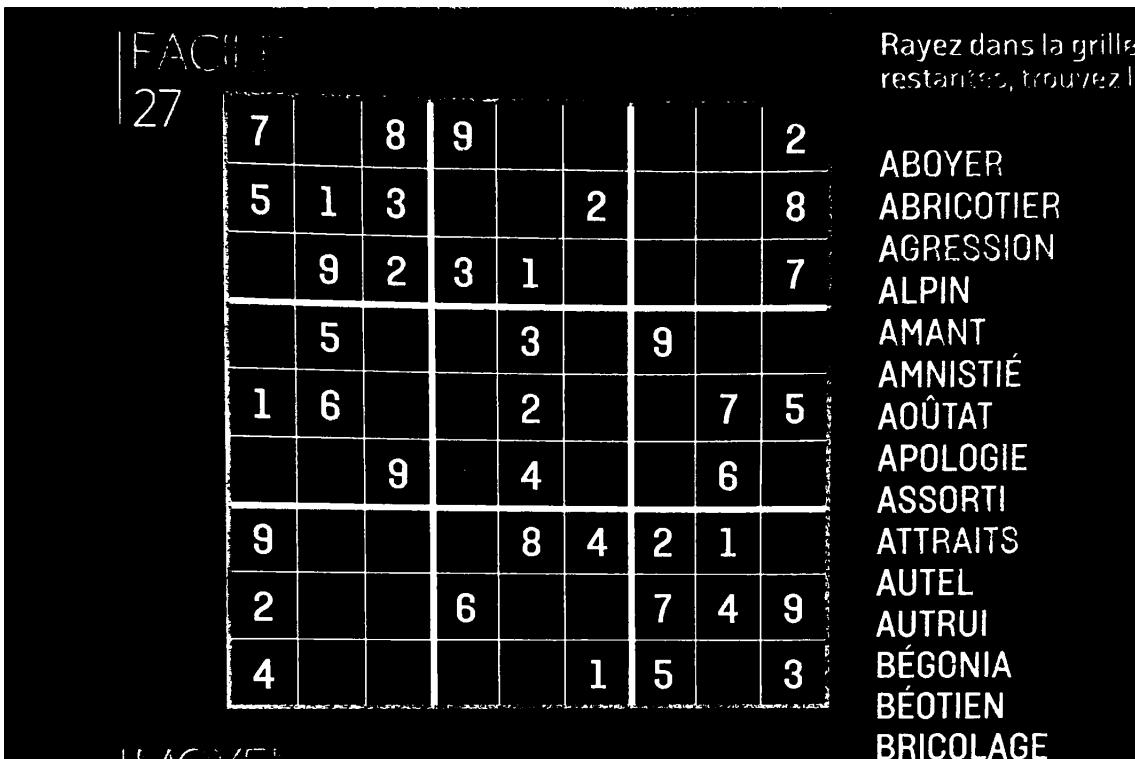


FIGURE 3.2 – Application de la méthode d’Otsu sur l’image n°04

3.4 Détection des contours

L’étape tant attendue... la détection des contours. Pour cela nous avons utilisé le détecteur de bords Canny (Canny Edge Detection). Celui-ci permet d’extraire les contours des formes pour pouvoir ensuite appliquer la détection de ligne dessus. Afin d’avoir de meilleurs résultats nous allons légèrement flouter l’image à l’aide d’un flou gaussian de rayon 3.

3.4.1 Intensité du Gradient

Dans un premier temps, nous allons générer deux nouvelles images : une première image qui va contenir la "dérivé verticale" de l’image binaraisé et la seconde la "dérivé horizontale". On parle ici de dérivé pour désigner les changements brusques de valeurs (sombre à clair par exemple). On va générer ces deux images parallèlement en déplaçant les fenêtres suivantes (* est l’opérateur convolution) :

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{et} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Par la suite, on va calculer l’intensité du gradient et sa direction avec :

$$I_x = G_x * I, \quad I_y = G_y * I, \quad G = \sqrt{I_x^2 + I_y^2}, \quad \theta = \arctan(I_y/I_x)$$

3.4.2 Réduction des bords

Une fois les bords extraits, nous devons les adoucir les bords. Nous allons utiliser l'algorithme de suppression non-maximale :

- Pour chaque bord détecté, on récupère l'angle θ associé à son gradient
- On vérifie dans la direction de l'angle (par exemple si $\theta = \frac{\pi}{2}$, on vérifie en haut et en bas du pixel) s'il n'existe pas un pixel avec une intensité de gradient plus élevé.
- Si il en existe un, alors met la valeur d'intensité du gradient du pixel actuel à 0

Après cette étape, nous appliquons un double seuillage pour réduire encore une fois le nombre de bord détecté. Ce double seuillage va générer trois catégories : intensité nulle, intensité faible et intensité forte selon la valeur de l'intensité du gradient. L'intérêt du double seuillage se situe à la prochaine étape : on va appliquer l'algorithme "hysteresis" qui va permettre de reconnecter certains bords qui ont pu être séparés. Le principe est relativement simple : pour chaque pixel ayant une intensité de gradient faible, le pixel devient une intensité forte s'il possède un pixel voisin d'intensité forte.

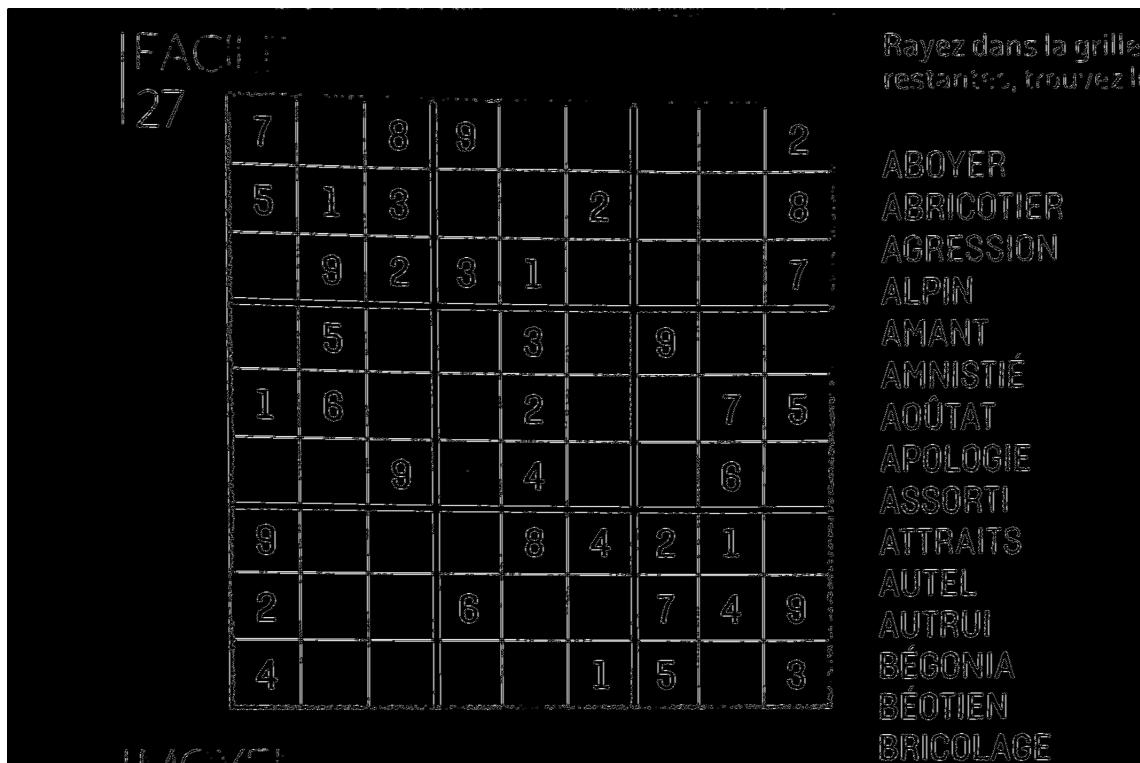


FIGURE 3.3 – Résultat de la détection des contours sur l'image n°04

Chapitre 4

Détection de la grille

Pour détecter la grille dans l'image d'entrée, il faut pouvoir identifier les bords du carré qui constitue la grille mais aussi son angle de rotation dans le cas où la grille n'est pas droite. La première étape implique l'utilisation de la transformée de Hough, celle-ci nous permet de détecter toutes les lignes présentes dans l'image issues du prétraitement. Nous allons ensuite trouver tous les carrés possibles à partir des lignes détectées. Finalement, nous allons choisir le carré qui a le plus de chance de contenir la grille voulue.

4.1 Transformée de Hough

La Transformée de Hough est un processus de transformation des points de l'image d'entrée traitée en points dans l'espace de Hough. Dans cet espace, les lignes dans l'image d'entrée vont être représentés par un point et vice-versa, dans l'espace de Hough, un point représentera une ligne dans l'image d'entrée.

4.1.1 Hough Space

L'espace de Hough est le résultat d'une recherche analytique pour chaque pixel dans l'image d'entrée. Pour chaque pixel, on parcours tous les angles possibles avec un certain pas et on incrémentera tous les points de la courbe représentative dans l'espace de Hough. On obtiendra alors un tableau à deux dimensions de taille finie avec un nombre de votes par θ (axe x) et ρ (axe y).

L'algorithme détaillé de génération de l'espace de Hough est :

- Pour chaque pixel blanc (x, y) dans l'image prétraitée :
 - On parcourt l'intervalle de θ de -90° à 90° avec comme pas $\Delta\theta$.
 - On calcule le ρ correspondant à θ actuel avec la formule :

$$\rho = \cos(\theta) * x + \sin(\theta) * y$$

- On incrémentera la valeur de l'accumulateur à la position $(x', y') = (\theta, \rho)$.

Le choix du pas $\Delta\theta$ est un compromis entre performance et précision : un plus grande nombre d'étapes donnera des résultats plus précis et donc la position des lignes détectées sera plus précise mais nécessitera plus d'itérations par pixel. Dans notre cas nous avons défini $\Delta\theta = \frac{\pi}{180}$. De plus, pour optimiser le processus de génération, nous avons opté pour le calcul d'une table contenant à l'avance les valeurs de $\cos(\theta)$ et $\sin(\theta)$, étant donné que $\Delta\theta$ est connu, juste avant le lancement de l'algorithme.

4.1.2 Extraction des lignes

L'étape suivante est d'extraire les lignes, pour cela, nous calculons d'abord le nombre maximum de votes présent dans l'accumulateur final. Puis nous définissons un seuil relatif à ce maximum, dans notre cas nous avons choisi 50%, toutes les lignes ayant 50% ou plus seront retenues. Ensuite, nous appliquons la transformation inverse pour récupérer les valeurs de θ et de ρ . Pour récupérer les coordonnées cartésiennes de la ligne dans notre image d'entrée, nous choisissons arbitrairement, deux points très éloignés, $x_1 = -10000$, $x_2 = 10000$ et nous calculons leur position y avec les formules suivantes :

$$A_x = \rho * \cos(\theta) + x_2 * \sin(\theta)$$

$$A_y = \rho * \sin(\theta) + x_1 * \cos(\theta)$$

$$B_x = \rho * \cos(\theta) - x_2 * \sin(\theta)$$

$$B_y = \rho * \sin(\theta) - x_1 * \cos(\theta)$$

Nous obtenons alors deux points qui nous permettent de déterminer leur équation plan. Cependant, avec ce processus, beaucoup de lignes parasites sont générées. Pour cela, nous allons parcourir la liste des lignes trouvées et fusionner les lignes qui sont similaires pour réduire drastiquement le nombre de "doublons". Nous avons choisi de fusionner toutes les lignes pour lesquelles $|\Delta\theta| < 5$ et $|\Delta\rho| < 15$. Toutes lignes respectant ces conditions seront fusionnées en une seule tel que :

$$\rho' = (\rho_1 + \rho_2)/2, \quad \theta' = (\theta_1 + \theta_2)/2$$

$$A'_x = (A_{x_1} + A_{x_2})/2, \quad A'_y = (A_{y_1} + A_{y_2})/2$$

$$B'_x = (B_{x_1} + B_{x_2})/2, \quad B'_y = (B_{y_1} + B_{y_2})/2$$

4.1.3 Répéter les lignes parallèles

Avant de trouver tous les rectangles qui peuvent être formées par ces lignes, nous allons d'abord grouper ensembles les lignes parallèles. De cette manière, il sera beaucoup plus facile de calculer les intersections et d'éliminer les rectangles impossibles. Voici l'algorithme que nous avons programmé :

- Pour chaque ligne trouvée
- On parcourt la liste des lignes trouvées
- Si $\Delta\theta$ entre les deux lignes est inférieur à un seuil, nous avons choisi 5° , alors les lignes ont une forte chance d'être parallèles.

- Et si les deux lignes ont un nombre de votes similaires dans l'espace de Hough, alors on les considère parallèles. On notera $\alpha = (\theta_1 + \theta_2)/2$. Nous avons choisi un écart maximum de 40%.

4.2 Extraction de la grille

Après avoir trouvé et groupé les lignes parallèles, il faut extraire tous les rectangles possibles. Comme on sait qu'un rectangle est constitué de lignes parallèles deux à deux et perpendiculaires deux à deux, il faut alors trouver deux paires de lignes parallèles qui sont perpendiculaires entre elles : $\Delta\alpha = ||\alpha_1 - \alpha_2| - 90| < \delta s$. Où δs définit l'incertitude que ces deux lignes soient perpendiculaires car les lignes ne sont pas extraites sans erreur depuis l'espace de Hough.

Une fois qu'on a trouvé deux paires de lignes perpendiculaires nous devons déterminer s'il s'agit bien d'un carré. Pour cela, on va calculer le rapport des longueurs des côtés car on sait qu'un carré a tous ses cotés de même longeur tel que :

$$\text{squareness} = \left| \frac{\rho_{A1} - \rho_{A2}}{\rho_{B1} - \rho_{B2}} \right|$$

Où A est la première paire de lignes parallèle et B la deuxième. Encore une fois, il faut définir une certaine incertitude pour obtenir des résultats corrects. Nous pensons que tous les rectangles avec une différence de moins de 7.5% entre le plus et le plus court côté est acceptable.

Finalement, après avoir générée la liste des pseudo-carrés dans l'image, nous allons les trier par surface car on suppose que le sudoku représente le grand carré dans l'image. Enfin, la dernière étape est de comparer les 5 plus grands carrés de l'image avec un calcul de score : $S = \text{squareness}^2 * \text{area}$. Le carré présent dans le top 5 avec le meilleur score sera retenu et considéré comme la grille de Sudoku.

4.3 Découpage des cases

Une fois la grille identifiée. Il faut déterminer les coordonées des intersections des lignes qui la compose. Avec ces coordonées, on va pouvoir déterminer le centre de la grille et effectuer une rotation de manière automatique centrée sur la grille, si celle-ci n'est pas droite. Puis on va recadrer l'image d'entrée de manière à ce qu'on se retrouve avec une nouvelle image contenant uniquement la grille. La dernière étape est de découper celle-ci selon la composition d'une grille classique de Sudoku, c'est-à-dire, 9 cases par coté pour 81 cases au total.

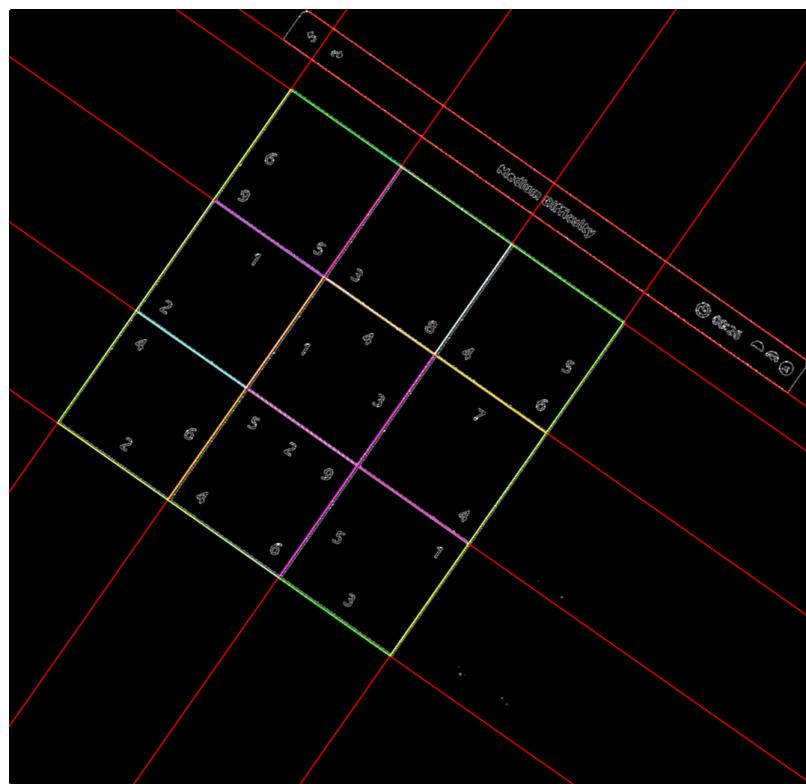


FIGURE 4.1 – Grille retenue en verte détectée depuis les lignes rouges

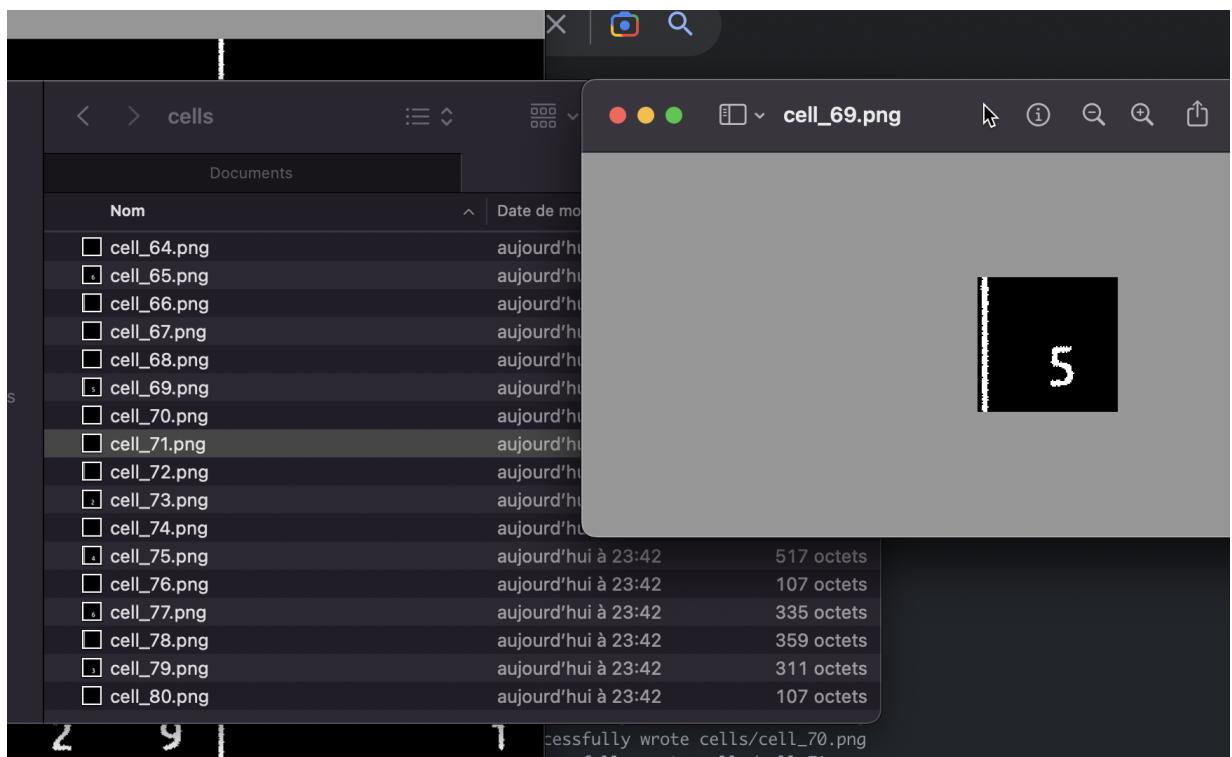


FIGURE 4.2 – Extraction des 81 cases depuis la grille droite et recadrée

Chapitre 5

Résolution des Sudokus

Pour le Sudoku Solver, c'est Kévin qui s'est chargé de cette tâche. Le Sudoku solver doit répondre à des critères bien précis : récupérer un fichier (sorti du réseau de neurones, réalisé par Mehdi) où se trouve le sudoku, puis l'importer dans le code en faisant toutes les vérifications de prétraitement, le transformer en structure de Sudoku interprétable par le code C, puis essayer de le résoudre (si le Sudoku est résoluble). Enfin, si la solution a été trouvée, le sauvegarder dans un fichier (personnalisable ou par défaut).

5.1 Backtracking

L'algorithme que nous allons utiliser pour résoudre les Sudokus est le Backtracking, une forme de bruteforce adapté pour être plus rapide pour les problèmes ayant des contraintes facilement identifiables. Dans notre cas, le Sudoku possède des règles pouvant être traduites très facilement en algorithme.

Nous allons donc dans un premier temps, déterminer toutes les possibilités de jeu pour chaque case initialement vide. C'est-à-dire, de partir de toutes les possibilités jouables à un indice précis de la grille, on regarde dans la ligne, la colonne et le carré quels nombres sont déjà présents et les retirer de la liste des nombres jouables. Puis une fois que les possibilités sont déterminées, on pose une à une les possibilités et on s'appelle récursivement sur la prochaine case, si l'appel récursif est négatif (la grille n'a pas pu être résolue), on essaie avec la possibilité suivante s'il y en a. Si la prochaine case est la dernière case du Sudoku alors on s'arrête et on vérifie que le Sudoku est résolu.

L'algorithme complet est :

- Pour chaque case initiallement vide
- On détermine tous les nombres jouables sur cette case
 - On détermine la prochaine case à remplir
 - Si elle est en dehors de la grille alors on s'arrête et on vérifie qu'elle est résolue
 - Sinon on s'appelle récursivement sur la prochaine case.
 - Si l'appel récursif retourne faux, alors on essaye le nombre jouable suivant
 - Sinon (il retourne vrai), on retourne immédiatement vrai : la grille a été résolue !

5.2 Structure et Traitement

Pour la structure donc, le Sudoku est composé d'un grille (représentée par un array), de la taille d'un côté (c'est-à-dire si le Sudoku est un HexaSudoku (12 cases) ou non, 9 cases dans ce cas), de la taille totale du Sudoku (soit la taille/len de l'array), et le nombre de carrés (correspondant aux carrés de la board, utilisés par le fonction calculant les possibilités décrite précédemment).

Il ne manque plus que la partie traitement d'un sudoku (sauvegardé dans un fichier), pour cela nous avons préféré opter pour une version CLI pour la version 1.0 de notre Solver. Pour cela, il fallait traiter de manière intelligente les fichiers.

Tout d'abord, lire la première ligne et regarder la taille du sudoku sauvegardé. Et créer un array qui correspond à cette taille. Puis, parcourir les lignes du fichier en insérant les valeurs dans le array. Enfin, il suffit de terminer l'importation en créant la structure à partir des éléments utilisés dans cette fonction (la taille et l'array). Cette méthode est simple et très rapide d'exécution, malgré les tests de préventions lors de la lecture du fichier. Pour la sauvegarde, même chose, il suffit de faire le chemin inverse, c'est-à-dire créer le fichier, et écrire dans celui-ci les valeurs dans le board.

CLI et Gestion d'erreurs

Le CLI permet de lier toutes ces fonctions et de faire les dernières vérifications au préalable avant le traitement entier du Sudoku. Celle-ci est très optimisée et utilise les méthodes les moins coûteuses en mémoire et en calculs. Il suffit de borner le nombre d'arguments afin de renvoyer des erreurs dès le début du traitement si le format n'est pas valide, ou de regarder si par exemple, le '-' est bien précisé dans le choix des options.

Pour le manuel rien de plus simple, il suffit d'appeler le programme sans argument : toutes les commandes seront affichés ainsi que leurs descriptions.

```

→ solver git:(kevin/solver) ✘ ./solver .../examples/solver/grid_00
Group check failed: Invalid cell at index 18, cell 4 is already placed (state: 8)
-----
| 4 |   |   |   |   | 4 | 5 | 8 |   |
-----
|   |   |   | 7 | 2 | 1 |   |   | 3 |
-----
| 4 |   | 3 |   |   |   |   |   |   |
-----
| 2 | 1 |   |   | 6 | 7 |   |   | 4 |
-----
|   | 7 |   |   |   | 2 |   |   |
-----
| 6 | 3 |   |   | 4 | 9 |   |   | 1 |
-----
| 3 |   | 6 |   |   |   |   |   |   |
-----
|   |   |   | 1 | 5 | 8 |   |   | 6 |
-----
|   |   |   |   | 6 | 9 | 5 |   |
-----
solver: The board is not a valid sudoku
→ solver git:(kevin/solver) ✘

```

FIGURE 5.1 – Erreur dans la grille entrée

Les erreurs sont donc claires et explicites, il est simple de voir où l'utilisateur a fait une erreur (que ce soit dans le CLI, le format du fichier, sudoku non soluble importé, fichier non existant...) ou si une erreur interne a été déclarée (avec le nom de la fonction, le problème rencontré et le code erreur).

5.3 Optimisation : Bit Masks

Un bit mask est tout simplement un nombre dont on va uniquement utiliser sa représentation binaire pour représenter un état. Dans notre cas, le bit à la position n représente si le nombre n est présent dans la ligne, la colonne ou le groupe de 3 par 3. De cette manière, il nous suffit de passer cet entier par nos fonctions et donc éviter de copier une liste qui contiendrait la liste des nombres jouables. Par exemple, si les possibilités sont : 1, 2, 4 et 8, le bitmask (sous forme de short, 16 bits) vaudra : 010001011. Alors qu'avec le array (tableau de short), les solution seraient : 1, 2, 4, 8 ce qui est bien plus lourd en mémoire (ici, 4*16 bits), 9x16 (144) bits dans le pire des cas.

Pour résumer, les bit masks permettent d'encoder toutes les possibiltés de jeu d'une case. Pour les utiliser, nous devons implémenter plusieurs fonctions pour encoder et décoder les bit masks : Verify, Set et Clear.

- Verify permet notamment de regarder si le nombre désigné peut être joué, c'est-à-dire, si le bit est à 1 dans le bit mask. Dans ce cas, il retournera un booléen

(int).

- Set est une fonction permettant de mettre le bit du Bitmask correspondant au nombre entré en paramètre à 1, ce qui correspond à un OU logique. Autrement dit, de mettre dans les possibilités le nombre indiqué. (ex : si 5 est jouable avec le bit mask précédent on aurait : 010001011 -> 010011011).
- Clear à l'inverse permet de retirer le bit correspondant au nombre qui n'est pas/- plus jouable. C'est un NAND logique. (Même exemple que précédemment mais à l'inverse).

Ces fonctions sont possibles grâce au BitShifting. En effet, il suffit de décaler n fois (n étant le nombre à ajouter ou supprimer au/du bit mask) et d'appliquer les opérations logiques précédemment sur le bit mask, ce qui est donc très peu coûteux en terme de calculs en plus d'être peu coûteux en mémoire !

Toutes les fonctions décrites précédemment ont été créées afin de diminuer les temps de calcul et donc d'avoir une résolution de Sudoku bien plus rapide que les autres solutions. Algorithmiquement parlant, le Solver de SudOCRu est donc très performant. Mais cette optimisation se fait aussi ressentir sur le traitement d'un Sudoku !

5.4 Rotation Manuelle

Pour la rotation, nous avons cherché à aller à l'essentiel, c'est-à-dire de récupérer une image dans un fichier, de la tourner d'un angle entré en paramètre et de la sauvegarder dans un fichier.

Pour cela, nous avons d'abord récupéré la surface de l'image (tous les pixels de l'image sous forme RGB) et de créer une autre surface admettant les mêmes dimensions que celle importée, c'est celle qui contiendra l'image tournée et qui sera sauvegardée dans le fichier de sortie. Ensuite, nous parcourons toute la première surface (celle importée) afin d'effectuer la rotation. L'usage de la trigonométrie était donc indispensable. Pour effectuer la rotation, il faut prendre le pixel concerné et y récupérer les couleurs R, G et B afin de créer le nouveau pixel qui sera implanté dans la surface finale. Il faut ensuite utiliser les formules de trigonométrie pour trouver l'emplacement du pixel après rotation et y importer le pixel.

$$\begin{aligned}x' &= (x - C_x) * \cos(\theta) - (y - C_y) * \sin(\theta) + C_x \\y' &= (x - C_x) * \sin(\theta) + (y - C_y) * \cos(\theta) + C_y\end{aligned}$$

Où C est le centre de l'image. Il faut évidemment être très rigoureux dans le choix des types de variables ou dans les bornes de l'image qu'il ne faut pas dépasser auquel cas une image déformée et/ou bruitée et/ou incorrecte sera sauvegardée, dans le pire des cas, avoir des erreurs de dépassements. Pour la partie rotation donc, de simples opérations de trigonométrie et de rigueur permettent d'effectuer cette tâche. Il reste donc qu'à implémenter l'interaction avec le UI afin de rendre la rotation plus « user-friendly ».

Chapitre 6

Réseaux de neurones

Après avoir isolé un chiffre dans une image, il faut pouvoir le reconnaître. Plusieurs solutions étaient disponibles afin de résoudre ce problème. La solution choisie est le réseau de neurones, plus précisément, l'OCR (Reconnaissance optique de caractères) ou dans notre cas les caractères seront des chiffres.

6.1 Principe du réseau de neurones

Un réseau de neurones peut être vu comme une succession de fonctions appliquées les uns après les autres. Ils sont organisés en plusieurs couches. Cette structure peut être divisée en trois parties : une couche de neurones d'entrée, une/des couches de neurones intermédiaires, puis finalement une couche de neurones de sortie.

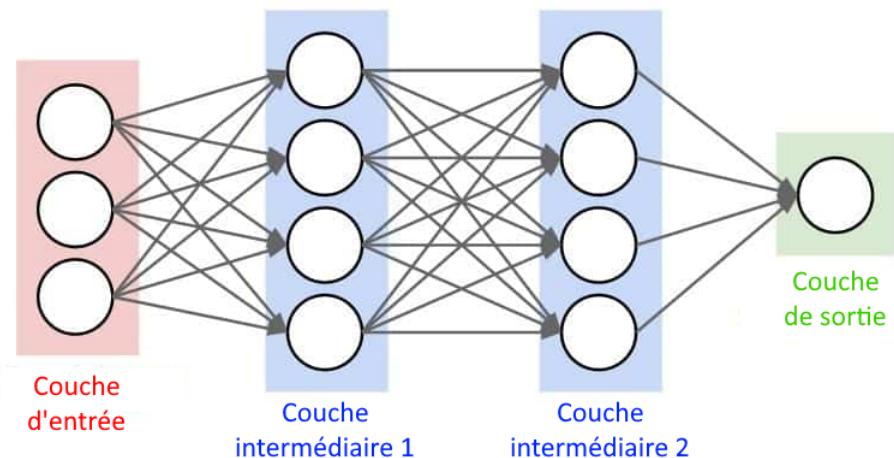


FIGURE 6.1 – Structure d'un réseau de neurone pour représenter le OU exclusif

Un neurone peut être vu comme une fonction. Chaque neurone est relié par des branches vers tous les neurones qui lui précèdent et tous les neurones qui suivent. Chaque branche possède une donnée qui lui est propre qui s'appelle "poids". Chaque groupe de branches qui appartient à la même transition de couche de neurones possèdent en commun une même donnée appelée "biais".

Chacune de ces données va être utilisée afin de calculer la valeur de chaque neurone de la couche suivante. Afin d'homogénéiser les valeurs de chaque neurone, une fonction supplémentaire va être utilisée, appelée fonction d'activation. Il en existe plusieurs applicables dans ce cas d'utilisation comme la fonction sigmoïde, la fonction tangente hyperbolique ou encore la fonction ReLU (Rectified Linear Unit). La fonction sigmoïde a finalement été utilisée. Sa notation est la suivante :

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

La formule mathématique pour calculer la valeur de chaque neurone en fonction du biais, des poids et des valeurs des neurones précédents est la suivante :

Soit $a_{n,m}$ le neurone m de la nième couche à calculer, $a_{n-1,j}$ l'ensemble des neurones de la couche précédente allant de 0 à j , $w_{i,j}$ le poids de la branche allant du neurone i au neurone j , b le biais et σ une fonction d'activation. On a :

$$a_{n,j} = \sigma \left(\sum_{k=0}^j \omega_{k,m} * a_{n-1,k} + b_{n-1} \right)$$

6.2 Reconnaissance de chiffres

Si nous voulons par exemple reconnaître un chiffre dans une image de taille 28x28 pixels, il nous faudra obligatoirement avoir 784 neurones dans la couche d'entrée, c'est-à-dire un neurone par pixel, et dix neurones dans la couche de sortie, un pour chaque chiffre. Une ou plusieurs couches de neurones intermédiaires peuvent être ajoutés afin d'affiner le résultat, mais cela impactera la puissance de calcul nécessaire.

Le problème rencontré est qu'on ne peut pas dès le premier coup deviner l'ensemble des valeurs des poids et des biais de chaque branche afin d'avoir les meilleurs résultats possibles. C'est pourquoi il faudra utiliser une méthode d'apprentissage appelée Back-propagation, ou Rétropropagation du gradient. Le principe est d'initialiser des valeurs aléatoires de poids et de biais et de tester le réseau de neurones avec des données et un résultat attendu. Après le calcul, chaque neurone de sortie sera comparé avec ce qui est attendu, puis tout poids et biais qui ont impacté la valeur de sortie vont être plus ou moins modifiés en fonction du rapprochement ou non de la valeur attendue. Ce processus est répété un certain nombre de fois (souvent très grand).

Chapitre 7

Conclusion

Pour résumer, de nombreuses tâches ont été réalisées par l'équipe SudOCRu. Pour cette soutenance, la totalité du travail qui a été demandé a été remis et ce avec beaucoup de sérieux et d'autres tâches ont été bien avancées en parallèle pour ne pas perdre de temps pour la soutenance finale. En effet, le solver est terminé à 100%, la détection de la grille ainsi que le découpage des cases est terminé également ainsi que la rotation manuelle. En revanche, quelques éléments restent à réaliser ou corriger pour la soutenance finale : la reconnaissance de caractère à terminer, l'application et le filtrage à parfaire.

L'équipe SudOCRu est très fier de présenter ce projet avec beaucoup de sérieux et d'anticipation et va se surpasser encore davantage pour le rendu final.