

# **SudOCRu**

## Rapport Final (S3)

Sofiane Meftah (Chef de projet)

Kévin Jamet

Mehdi El Alaoui El Aoufoussi

Hamza Jamai

Décembre 2022

# Table des matières

<b>1 Présentation du groupe</b>	<b>3</b>
1.1 Sofiane . . . . .	3
1.2 Kévin . . . . .	3
1.3 Mehdi . . . . .	4
1.4 Hamza . . . . .	4
<b>2 Organisation générale</b>	<b>5</b>
2.1 Répartition des tâches . . . . .	5
2.2 Etat d'avancement . . . . .	5
<b>3 Prétraitement</b>	<b>6</b>
3.1 Niveaux de gris . . . . .	6
3.2 Filtre médian . . . . .	7
3.3 Filtre Bilatéral . . . . .	7
3.4 Binarisation . . . . .	8
3.4.1 Methode d'Otsu . . . . .	8
3.4.2 Seuillage adaptatif . . . . .	9
3.4.3 Dilatation morphologique . . . . .	9
3.5 Détection des contours . . . . .	9
3.5.1 Intensité du Gradient . . . . .	10
3.5.2 Réduction des bords . . . . .	10
<b>4 Détection de la grille</b>	<b>12</b>
4.1 Transformée de Hough . . . . .	12
4.1.1 Hough Space . . . . .	12
4.1.2 Extraction des lignes . . . . .	13
4.1.3 Répéter les lignes parallèles . . . . .	13
4.2 Extraction de la grille . . . . .	14
4.3 Correction de perspective . . . . .	14
4.4 Découpage des cases . . . . .	16
4.4.1 Suppression du bruit . . . . .	17
<b>5 Résolution des Sudokus</b>	<b>19</b>
5.1 Backtracking . . . . .	19
5.2 Structure et Traitement . . . . .	20
5.3 Optimisation : Bit Masks . . . . .	22

5.4	Rotation Manuelle . . . . .	24
<b>6</b>	<b>UI</b>	<b>25</b>
6.1	Glade . . . . .	25
6.1.1	Elements utilisés dans Glade . . . . .	26
6.1.2	Catégories et options utilisées . . . . .	27
6.2	L'application et le design . . . . .	29
6.2.1	Fenêtre principale et design général . . . . .	29
6.2.2	Traitement de l'image . . . . .	29
6.2.3	Threshold et traitement . . . . .	30
6.2.4	Cropping . . . . .	30
6.2.5	Réseau de neurones . . . . .	32
6.2.6	Prévisualisation du résultat et export . . . . .	33
6.2.7	CSS et Gtk . . . . .	34
6.3	Intégration . . . . .	36
6.3.1	Intégration des différentes parties . . . . .	36
<b>7</b>	<b>Réseaux de neurones</b>	<b>37</b>
7.1	Principe du réseau de neurones . . . . .	37
7.2	Reconnaissance de chiffres . . . . .	38
7.3	Notre implémentation du réseau de neurones . . . . .	38
<b>8</b>	<b>Conclusion</b>	<b>41</b>

# 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 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.

### Aperçu du prétraitement

- Niveaux de gris
- Filtre Median (5x5)
- Filtre Bilatéral (11x11,  $\sigma = 55$ )
- Binairization : Seuillage adaptatif (15x15,  $\delta = 5.5\%$ )
- Dilatation morphologique (3x3)

### 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  $L$ . Parallèlement 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 arrondit 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.

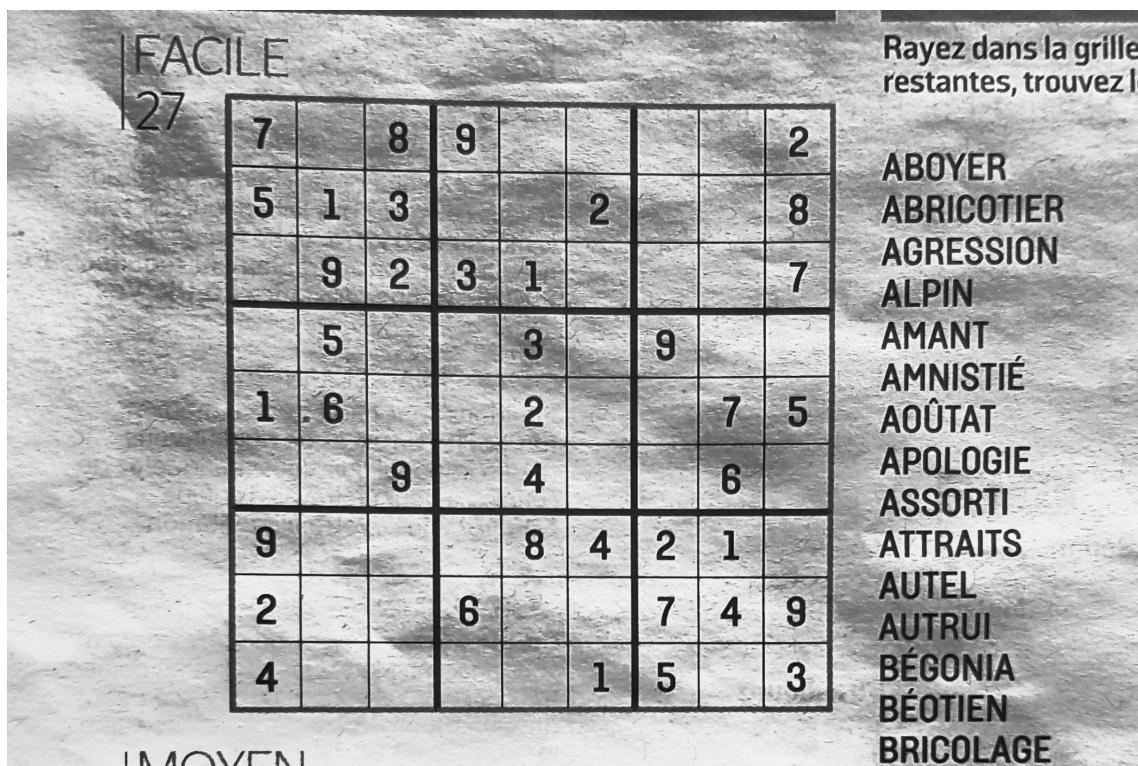


FIGURE 3.1 – Image n°04 après la mise en niveaux de gris et filtre médian

## 3.3 Filtre Bilatéral

Pour adoucir le bruit et les possibles tâches présentes dans l'image, nous avons opté pour le filtre Bilatéral. Celui-ci est similaire au filtre gaussian mais permet de limiter la perte des bords. En effet, nous devons à tout prix préserver les bords puisque c'est grâce à eux qu'on va pouvoir extraire la grille. Le filtre Bilatéral est constitué d'une convolution de l'image d'entrée avec une fenêtre (Kernel) un peu spéciale : il y a une composante pour la différence d'intensité dans la fenêtre et une autre pour la différence de position (exactement comme dans le flou gaussian). Chaque pixel de l'image se calcule avec la formule suivante (celle-ci correspond à une convolution particulière) :

$$I_{x,y} = I_{x,y} \times \sum_{i=0,j=0}^{15} I_{i,j} \exp \left( -\frac{(i-x)^2 + (j-y)^2}{2\sigma_d^2} - \frac{|I_{i,j} - I_{x,y}|^2}{2\sigma_r^2} \right)$$

Un avantage du filtre Bilatéral par rapport au flou gaussian est les deux seuils  $\sigma$  pour déterminer le rayon d'effet spacial mais aussi en fonction de l'intensité : plus il y a de pixels sombres à proximité (un bord), moins le flou gaussian aura de l'effet et donc le bord sera préservé.

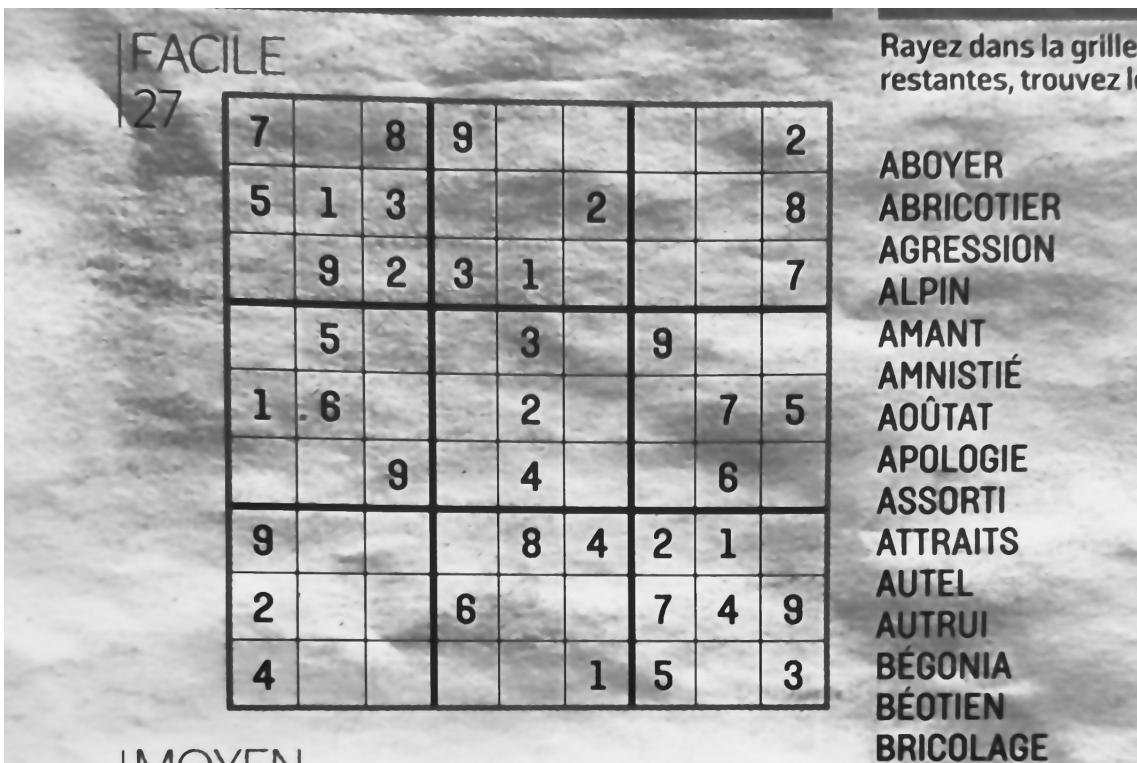


FIGURE 3.2 – Image n°04 après filtre médian et filtre Bilatéral

## 3.4 Binarisation

### 3.4.1 Méthode d’Otsu

La dernière étape avant la détection des contours est la binarisation : transformer notre image avec 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),  $\omega_0$  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.

### 3.4.2 Seuillage adaptatif

Cependant nous avons remarqué que la binarisation d'Otsu fonctionne correctement uniquement sur certaines images très particulières. Nous avons donc implémenté une autre méthode de binarisation : Il s'agit du seuillage adaptatif. Celui-ci donne de biens meilleurs résultats sur des images dont l'éclairage n'est pas équitable. Le principe du seuillage adaptatif est de calculer pour chaque pixel un seuil adapté selon une zone d'affluence. Dans notre cas nous avons choisi 15 pixels (15 par 15 centré sur le pixel). Ce seuil correspond à une moyenne des intensités de tous les voisins. Cependant, la moyenne n'est pas un seuil adéquat, en effet, certains pixels peuvent être légèrement en dessous (plus gris) tout en faisant partie de la bordure de la grille.

C'est pour cela qu'on va réduire d'un certain pourcentage cette moyenne tel que : Tous les pixels voisins qui ont une intensité supérieure ou égale à 90% de la moyenne de la zone d'affluence va être considéré comme sombre (blanc dans notre image).

### 3.4.3 Dilatation morphologique

Un gros désavantage du seuillage adaptatif est la formation de tranchées à l'intérieur des bords : seuls les bords extérieurs qui ont un fort contraste avec les pixels voisins seront gardés. Les pixels qui sont entièrement entourés de bords seront malheureusement considérés comme clairs vu que la moyenne de la zone d'affluence sera bien plus basse (à cause des bords). Pour pallier ce problème, nous avons opté pour la dilatation morphologique qui consiste à grossir les bordures en prenant l'intensité maximale dans une certaine fenêtre. Plusieurs fenêtres peuvent être possibles (pour des effets différents comme pour  $F_1$ ), nous avons opté pour un rayon de 3 avec la fenêtre  $F_2$ .

$$F_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{et} \quad F_2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

## 3.5 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.

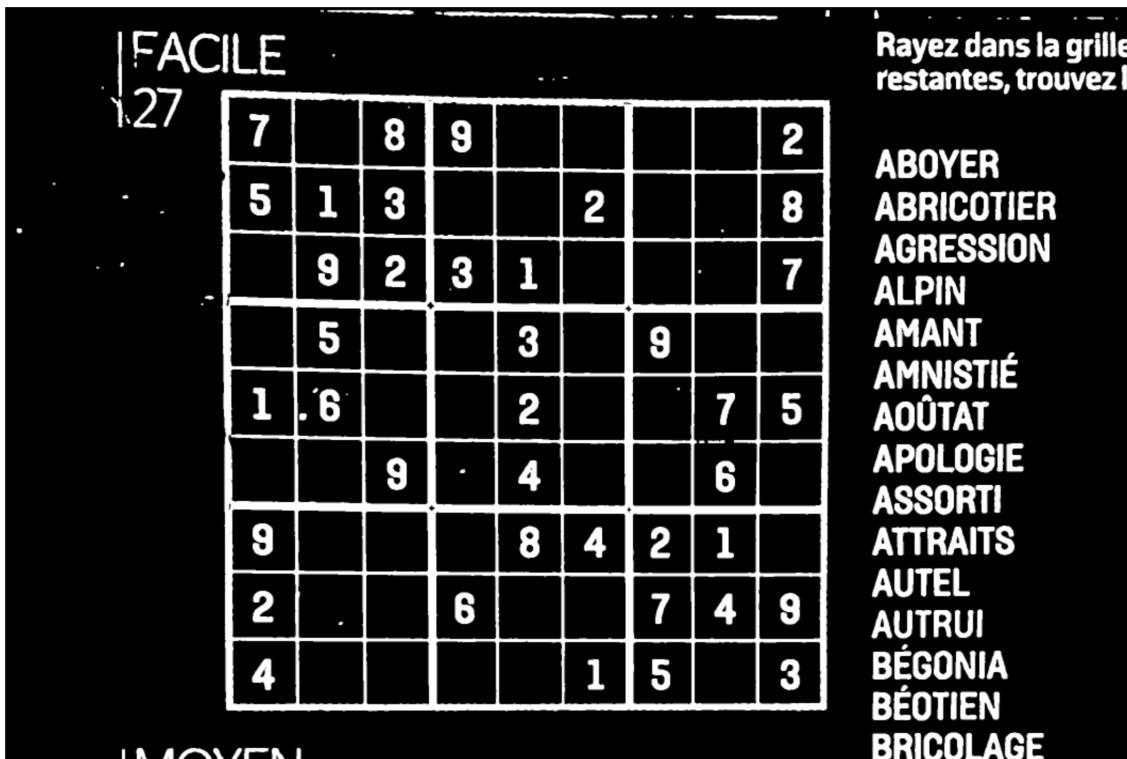


FIGURE 3.3 – Prétraitement complet sur l'image n°04

### 3.5.1 Intensité du Gradient

Dans un premier temps, nous allons générer deux nouvelles images : une première image qui va contenir la "dérivée verticale" de l'image binarisé et la seconde la "dérivée horizontale". On parle ici de dérivée 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 :

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

### 3.5.2 Réduction des bords

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

- Pour chaque bord détecté, on récupère l'angle  $\theta$  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 élevée.

- S'il en existe un, alors il faut mettre 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.

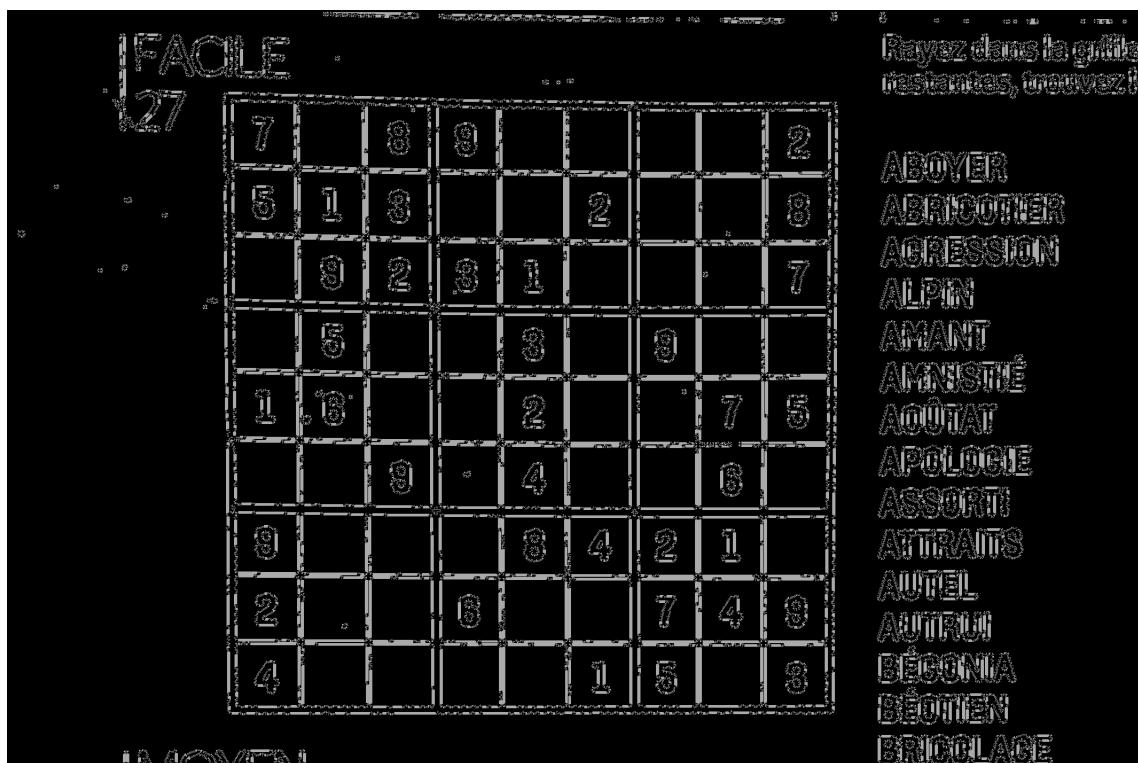


FIGURE 3.4 – Résultat de la détection des contours sur l'image n°04

# 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 parcourt 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  $\theta$  (axe x) et  $\rho$  (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  $\theta$  de  $-90^\circ$  à  $90^\circ$  avec comme pas  $\Delta\theta$ .
  - On calcule le  $\rho$  correspondant à  $\theta$  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  $\theta$  et de  $\rho$ . 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  $8^\circ$ , 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  $\delta s$  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 parallèles 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 peuvent être considérés comme carrés.

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 Correction de perspective

Avant de pouvoir découper la grille, il faut corriger la perspective de l'image pour que le réseau de neurones puisse donner de bons résultats. Pour cela, nous allons utiliser nos connaissances de Maths de S3 ainsi que des ressources disponibles sur internet. En effet, nous allons utiliser les applications linéaires ainsi que les matrices d'homographies. Dans un premier temps, nous devons trouver une application linéaire qui transforme la grille dans l'image vers une grille toute droite ayant pour coordonnées  $((0, 0), (0, l), (l, l), (l, 0))$  avec  $l$  étant la longueur du plus grand côté de la grille non recadrée. Il est très important d'avoir les points de la grille retournée par l'algorithme d'extraction tel que (en haut à gauche -> en bas à gauche -> en bas à droite -> en haut à droite) pour que la grille soit toujours mise à l'endroit. On va alors résoudre un ensemble d'équation pour déterminer les coefficients de l'application linéaire qui transforme les points  $F$  en les points  $T$ , on pose :

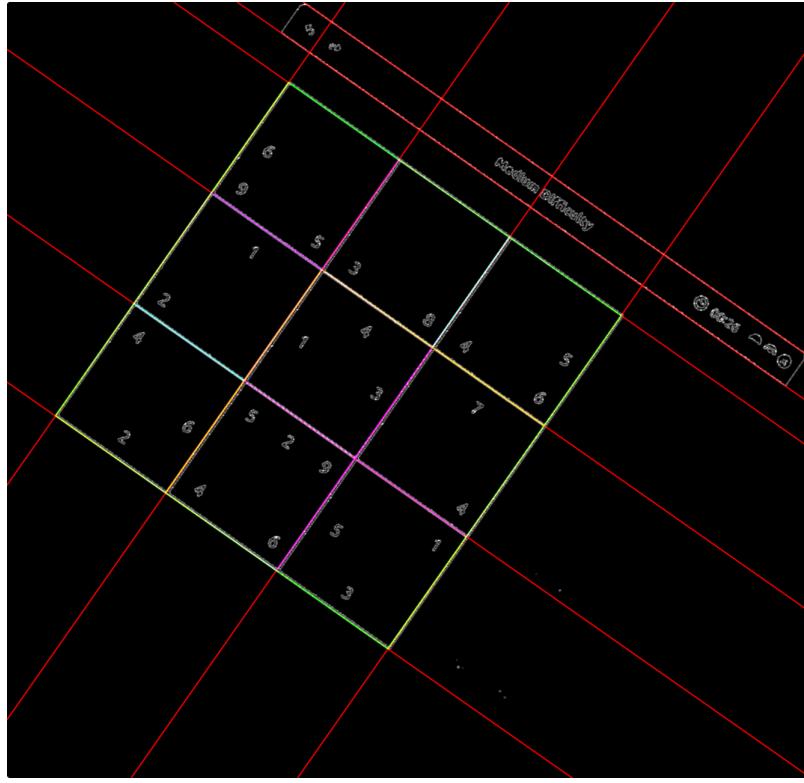


FIGURE 4.1 – Grille retenue en verte détectée depuis les lignes rouges

$$F = ((x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)), T = ((x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3), (x'_4, y'_4))$$

$$A = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1 x_1 & -x'_1 y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1 x_1 & -y'_1 y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x'_2 x_2 & -x'_2 y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -y'_2 x_2 & -y_2 y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x'_3 x_3 & -x'_3 y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -y'_3 x_3 & -y_3 y_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x'_4 x_4 & -x'_4 y_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -y'_4 x_4 & -y_4 y_4 \end{bmatrix}, \quad H = \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} \quad \text{et} \quad B = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix}$$

On a alors :  $AH = B$ . On va calculer la matrice d'homographie en utilisant la méthode des moindres carrés (least square method) pour minimiser les coefficients de  $H$  :

$$H = (A^T A)^{-1} A^T B$$

Ainsi, on détermine les coefficients de la matrice de l'application linéaire souhaitée :

$$M(f) = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix}$$

Enfin, on calcule l'image de l'image d'entrée par l'application linéaire inverse pour corriger la perspective tel que :

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix}^{-1} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

où  $(x, y)$  sont les coordonées dans l'image d'entrée (non corrigée) et  $(x', y')$  dans l'image de sortie. chaque composante d'une couleur différente).

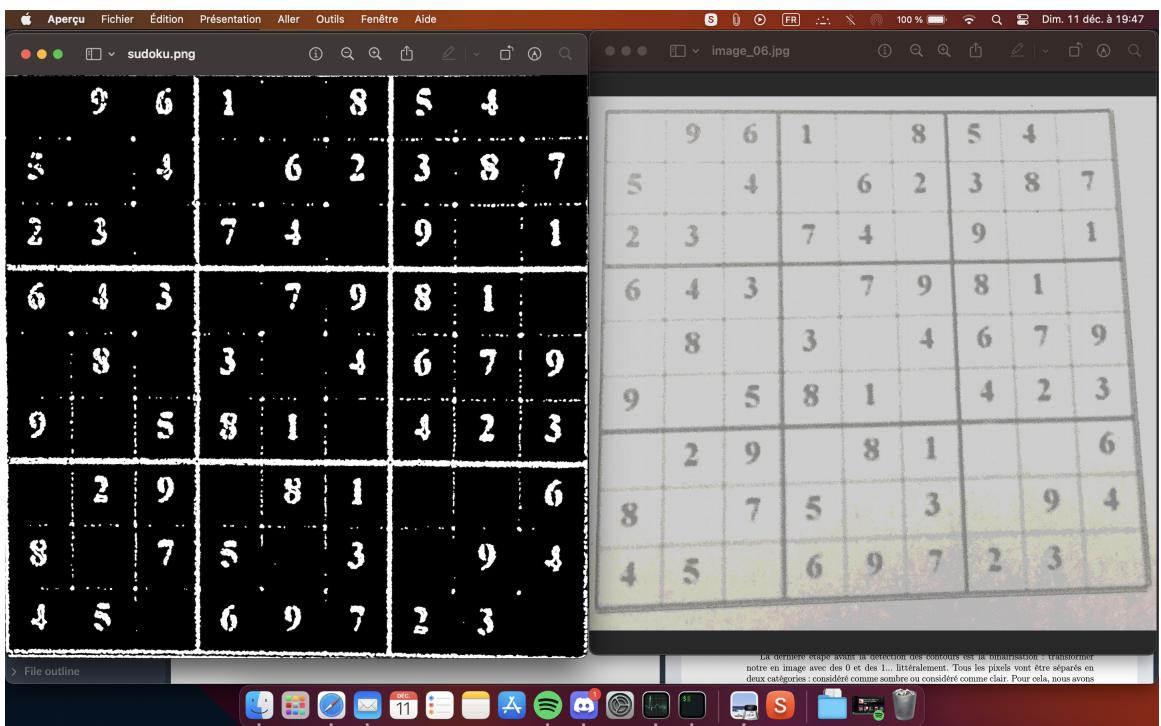


FIGURE 4.2 – Correction de la perspective de l'image n°06

## 4.4 Découpage des cases

Une fois la grille identifiée. Il faut déterminer les coordonnées des intersections des lignes qui la compose. Avec ces coordonnées, 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.

#### 4.4.1 Suppression du bruit

Après avoir découpé la grille en 9, il faut supprimer les bords restants et compresser l'image jusqu'à avoir seulement le nombre. Pour faire ça, nous avons considéré l'image comme un graphe non-orienté complet mais non connexe et nous effectuons un parcours largeur (trigonométrique) pour déterminer toutes les composantes connexes. Voici le résultat produit par notre algorithme, que nous avons baptisé l'algorithme confetti (nous avons coloré chaque composante d'une couleur différente).



FIGURE 4.3 – Algorithme confetti

Une fois les composantes fortement connexes déterminées, nous devons choisir quelles composantes garder et lesquelles supprimer. Pour réaliser cette tâche, nous déterminons la composante la plus grande ainsi que les composantes qui sont suffisamment grandes (jusqu'à 50% plus petites que la plus grande composante). Après avoir supprimé le bruit, nous devons la redimensionner en 28x28 pour qu'elle puisse être traitée par le réseau de neurones. Pour cela, nous utilisons l'algorithme de Downscaling par moyennage : on calcule la moyenne dans une fenêtre dans l'image d'entrée pour tous les pixels dans l'image de sortie. Cependant cette méthode est efficace que pour des facteurs de rétréssissement entiers, avec des valeurs flottantes, celle-ci crée des décalages non-négligeables dans l'image de sortie. C'est pour cela que dans l'application SudOCRu, le traitement est réalisé avec l'image de pleine résolution alors que celles-ci sont affichées avec une résolution inférieure pour des soucis de place sur l'écran (comme sur les slides).

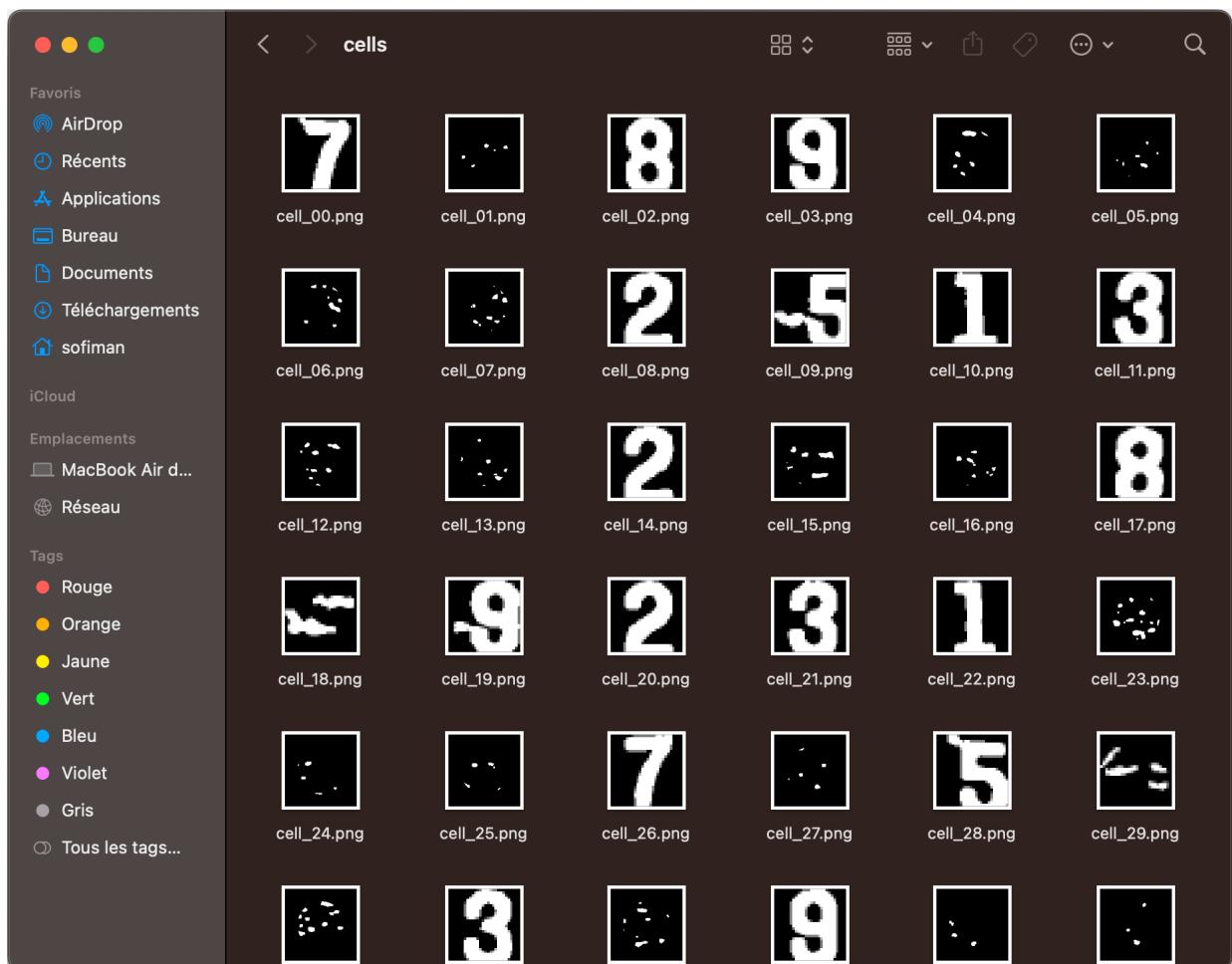


FIGURE 4.4 – Extraction des 81 cases redimensionnées nettoyées depuis la grille recadrée

# 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).

Pour le Sudoku nous avons fait des types d'erreurs en fonction des problèmes que l'algorithme peut rencontrer. En voici un tableau récapitulatif :

Types et codes d'erreurs			
Nom	Catégorie	Numéro	Description
General			
Exit_Success	General	0	Aucune erreur retournée
Operation_Succeeded	General	1	Opération exécutée avec succès
SolveError			
Sudoku_Not_Solvable	SolveError	11	Sudoku non soluble
Sudoku_Not_Solved	SolveError	12	Sudoku non résolu (donc non soluble)
ImportError			
File_Do_Not_Exists	ImportError	21	Le fichier spécifié n'existe pas
Empty_File	ImportError	22	Le fichier spécifié est vide
Sudoku_Format_Invalid	ImportError	23	La taille de la grille sauvegardée est invalide
Invalid_Character	ImportError	24	Un ou plusieurs caractères ne sont pas valides
SaveError			
Cant_Save_Sudoku	SaveError	31	Le fichier ne peut pas être sauvegardé
DestroyError			
Sudoku_Null	DestroyError	41	Le sudoku ne peut pas être détruit (inexistant)
PrintError			
Cant_Print_Null_Sudoku	PrintError	51	Le sudoku ne peut pas être affiché (inexistant)

FIGURE 5.2 – Erreur dans la grille entrée

### 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 solutions 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 possibilités 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'applique 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

## UI

### 6.1 Glade

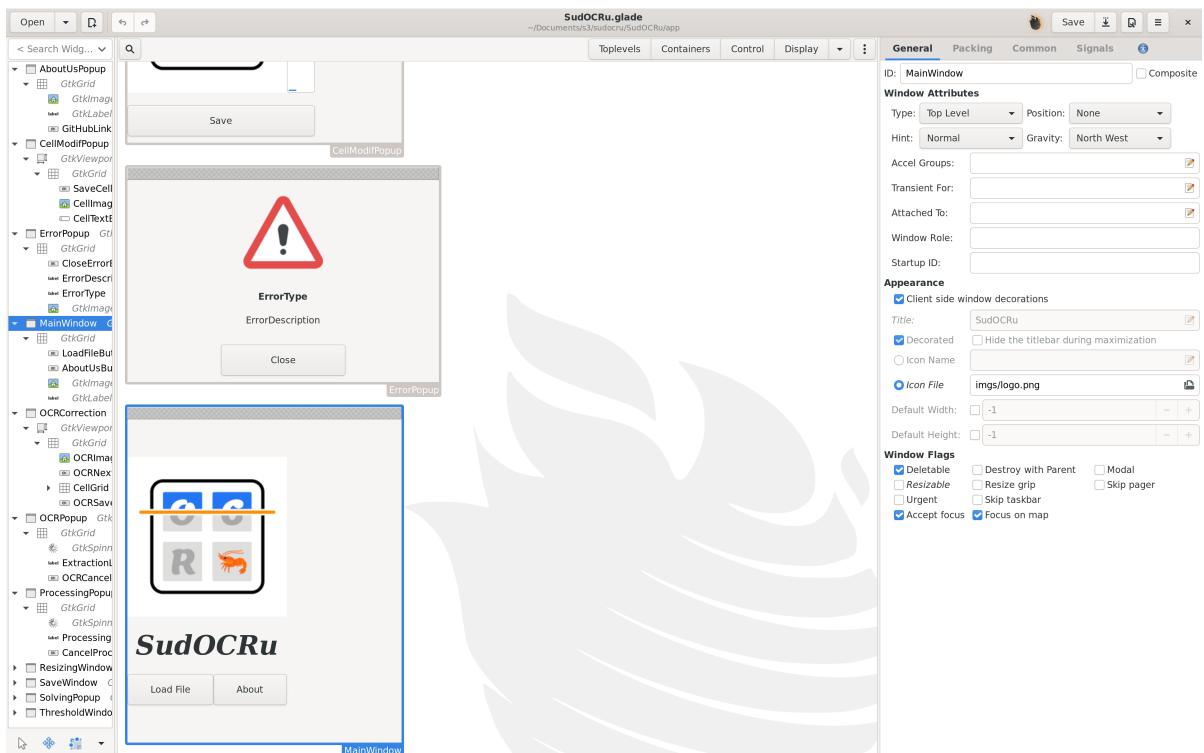


FIGURE 6.1 – Fenêtre glade de notre projet

Pour cette dernière soutenance, l'équipe a travaillé d'arrache-pied sur le design de l'application car pour qu'elle soit utilisée par un grand nombre d'utilisateurs, il faut que le design soit moderne, simple, élégant et très intuitif. C'est pour cette raison que nous avons beaucoup réfléchi sur l'application, ou plutôt sur la façon dont l'utilisateur va interagir avec notre outil de résolution de sudokus.

Un problème se pose assez vite quand il s'agit de créer une interface graphique sur Linux car une des seules librairies avec un minimum de fonctionnalités sans être incompréhensible est GTK. Mais pour créer une application sans retour direct de l'image, la création d'application devient très vite fastidieux voire démoralisant.

C'est pour cette raison que l'équipe SudOCRU a choisi d'utiliser Glade pour créer le UI. En effet, il propose tout ce dont l'équipe avait besoin ! Une interface pour voir visuellement à quoi ressemble l'application, de diviser plus facilement l'application, de créer des fenêtres responsives ou adaptatives en fonctions des éléments, mais surtout par sa simplicité d'utilisation !

### 6.1.1 Elements utilisés dans Glade

Pour commencer, Glade a été d'une grande aide dans la création de notre application pour sa simplicité, tout est clair :

#### Les top levels

Ce sont les éléments qui sont les plus hauts niveaux d'une application. Ils servent à créer le type de fenêtre. On peut citer par exemple les fenêtres (GtkWindow), les files chooser (GtkFileChooser) qui permettent de lancer l'explorateur de fichiers natif de l'OS (pour ouvrir ou sauvegarder des fichiers) et bien d'autres encore mais nous nous limiterons à ceux-ci car les autres types nous étaient d'aucune utilité pour notre application.

#### Les conteneurs

Les conteneurs sont en revanche une partie très importante de notre application. En effet, cette catégorie de GtkWidget sert à intégrer les deux derniers types de GtkWidget (les controls et display). Mais tous n'ont pas les mêmes propriétés, le choix de ceux-ci a été également très important.

Nous avons opté majoritairement pour des GtkViewPort et des GtkPaned. Ces premiers conteneurs nous permettent de rendre notre application responsive pour donner un aspect bien plus pro à notre application. Ces conteneurs sont d'autant plus intéressants pour leurs propriétés d'héritage (sur la taille notamment).

Dans le cas des GtkPaned, ceux-ci sont extrêmement pratiques pour rendre des structures responsives et avec des panneaux latéraux par exemple. Les GtkViewports servent à étendre les limites du conteneur afin d'épouser au mieux les formes et la taille de la fenêtre.

#### Les Grid

Les grids sont en réalité des conteneurs mais nous avons décidé de le mettre dans une autre catégorie car ses propriétés sont bien plus diverses et variées que celles décrites précédemment. En effet, elles permettent de diviser une partie du conteneur ou de l'application en un nombre défini de conteneurs avec les mêmes propriétés d'héritage de taille etc.

Ces éléments sont donc logiquement les plus utilisés par la communauté GTK mais également par l'équipe SudOCRu car notre application étant basée sur un Sudoku, il fallait utiliser ce conteneur.

## Les controls

Comme décrit précédemment, ces GtkWidget peuvent être intégrés dans les conteneurs. Cette catégorie de widgets permet de créer des actions dans le code. Comme des boutons ou des sliders, des scroll viewers. Tous présentant des actions différentes mais sont assez complètes pour répondre à toutes les problématiques.

Ici, l'équipe a majoritairement utilisé les GtkButtons, les GtkScales. Les GtkButtons sont des GtkWidget qui permettent de faire des actions comme « clicked ».

Elles permettent de passer d'une fenêtre à une autre ou de confirmer des choix notamment sur les fenêtres de traitement.

Les GtkScales sont des sliders qui permettent de modifier une valeur directement en glissant avec la souris. Ce type de GtkWidget est parfait pour modifier le threshold car cela nous permet de voir directement dans le previewer l'impact de celui-ci.

## Les displays

Les displays nous permettent notamment d'afficher les images dans le UI (après et pendant les prétraitements). Ici nous avons utilisé de manière très judicieuse les GtkImage car il n'est pas possible de mettre une SDL\_Surface directement dans le UI.

Pour cela, nous avons du résonner différemment, car nos images sont de type Image\* (un array de Int, les couleurs étant sur 8 bits chacune) et on ne peut directement afficher dans le UI un array de pixels. En réalité, c'est possible mais bien trop coûteux en ressources car il faut redessiner dans une GtkWidget chaque pixel de l'image.

Pour palier à ce problème, l'équipe SudOCRu a utilisé les Gtk\_Pixbufs qui sont des buffers pouvant stocker les pixels d'une image et pouvant ensuite être affichés sous forme d'image dans le UI. Cette fonctionnalité étant native dans GTK est bien plus adaptée pour afficher une surface en image.

### 6.1.2 Catégories et options utilisées

Et pour chaque élément de ces catégories, il y a plusieurs onglets :

#### General

Permet de modifier les paramètres généraux du GtkWidget et sont différents en fonction de l'élément sélectionné. Par exemple, dans notre utilisation, cette rubrique permet de modifier le nombre de lignes et colonnes de notre grid, de définir si un logo de chargement est visible ou non, de définir le pas dans le GtkScale et bien d'autres encore.

## Packing

Cette rubrique est notamment faite pour définir la place de l'élément sélectionné dans une grid ou même de faire en sorte que des éléments se superposent dans la grid, ou que de déterminer la largeur, la longueur de l'élément dans la grid et bien plus encore.

## Common

Cet onglet regroupe les éléments communs aux GtkWidget, c'est-à-dire les attributs qui sont communs à tous les éléments de GTK. Comme la taille, si l'élément est visible ou non...

## Signals

Les signals sont très explicites : les différents types de signaux à envoyer dans le code en C. ils permettent de faire des traitements en back-end à partir d'actions en front end.

Comme vous pouvez le constater, l'interface de Glade ainsi que sa simplicité a permis à l'équipe de créer une application simplement et ce dans un environnement de travail bien plus pratique que dans un simple éditeur de texte.

## 6.2 L'application et le design

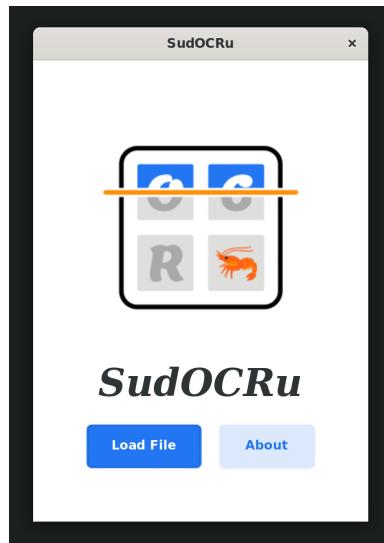


FIGURE 6.2 – Accueil de l'application

### 6.2.1 Fenêtre principale et design général

Notre outil passe d'abord par la simplicité, en effet, pas de superflu dans l'affichage, que des boutons utiles et bien placés pour ne pas perdre l'utilisateur dès l'ouverture de notre outil. Sur l'écran d'accueil : deux uniques boutons au centre pour charger un fichier (contenant le sudoku) et pour avoir des informations sur l'équipe SudOCRu.

### 6.2.2 Traitement de l'image



FIGURE 6.3 – Popup de chargement du traitement d'images

Pour le traitement de l'image, un popup avertit l'utilisateur que le traitement est en train de tourner en arrière-plan et qu'il faut patienter un petit instant.

Cette étape nous semblait nécessaire car de plus en plus d'applications de nos jours freezent ou laggent pendant le chargement d'un fichier ou quand une tâche est en train d'être effectuée en arrière-plan. Pour l'équipe SudOCRu ce popup était indispensable pour montrer à l'utilisateur que le traitement est en train de tourner et que l'application n'est juste pas en train de bugger ou de freeze.

### 6.2.3 Threshold et traitement

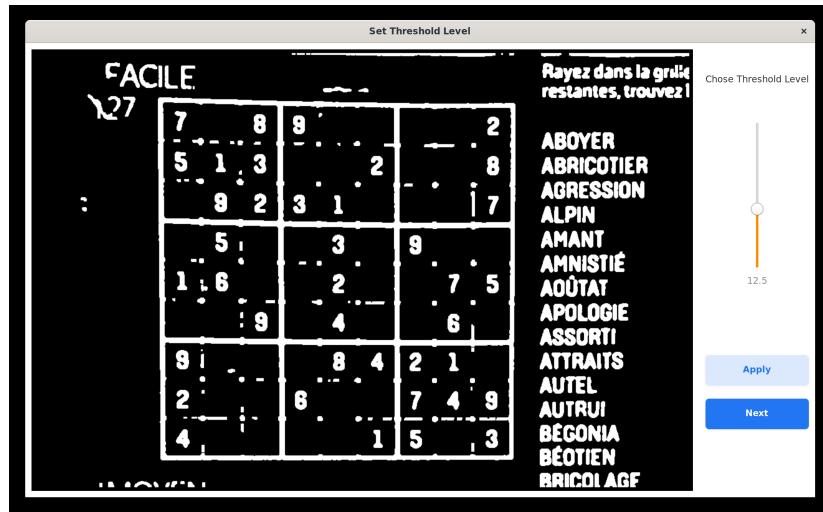


FIGURE 6.4 – Fenêtre de correction de threshold

La prochaine fenêtre nous semblait évidemment indispensable (voir ci-dessus) car elle permet de régler le niveau de bruit (threshold) de l'image à la fin du traitement pour enlever les résidus restants après application des multiples filtres appliqués précédemment. Il nous semblait essentiel d'implémenter cette fenêtre pour plusieurs raisons :

- Pour montrer à l'utilisateur que l'image a bien été traitée et s'il y a des corrections à faire sur d'éventuels résidus ou si son image a besoin d'être corrigée avant le traitement et ne pas avoir de surprises sur les étapes qui seront décrites plus bas dans la reconnaissance de caractères.
- Pour faciliter la définition du Threshold. En effet, cette tâche est assez subjective et complexe à implémenter algorithmiquement car il faut faire varier le threshold en arrière-plan et confirmer à l'utilisateur que le threshold a bien été réglé avant toute analyse par le réseau de neurones.
- Dans ce cas, nous avons donc laissé le choix à l'utilisateur de choisir son threshold et lui permettre d'avoir la mainmise sur son fichier et le traitement de celui-ci par notre réseau de neurones.

### 6.2.4 Cropping

Malgré le traitement de qualité de l'application SudOCRu, il fallait absolument que l'utilisateur puisse corriger le cropping de la grille s'il elle s'avérait peu ou pas visible. Pour cela, nous avons introduit la fenêtre de cropping. Elle affiche l'image de départ et quatre points à drag afin de corriger le cropping, les lignes sont retracées automatiquement afin de corriger parfaitement le cropping de la grille.

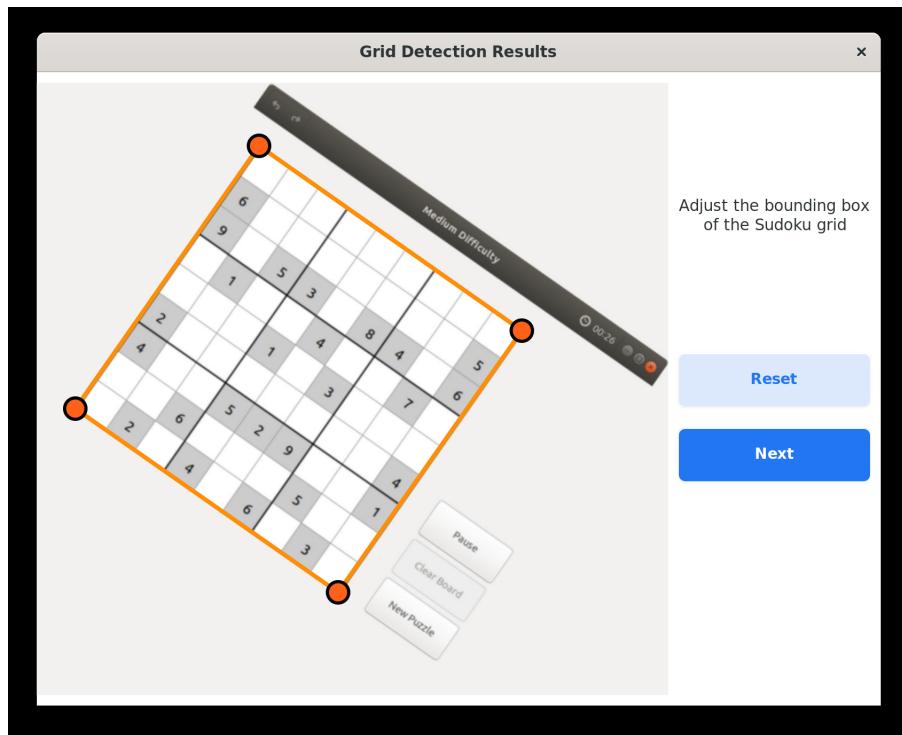


FIGURE 6.5 – Fenêtre de correction de cropping

L'utilisateur peut donc effectuer son cropping perso nnalisé ou juste vérifier et corriger que le traitement a correctement été effectué.

Le bouton reset sert à remettre la position des points aux mêmes coordonnées que le traitement automatique. cette option nous semblait indispensable car si l'utilisateur a modifié sans faire exprès le cropping, il peut toujours annuler son action.

### 6.2.5 Réseau de neurones

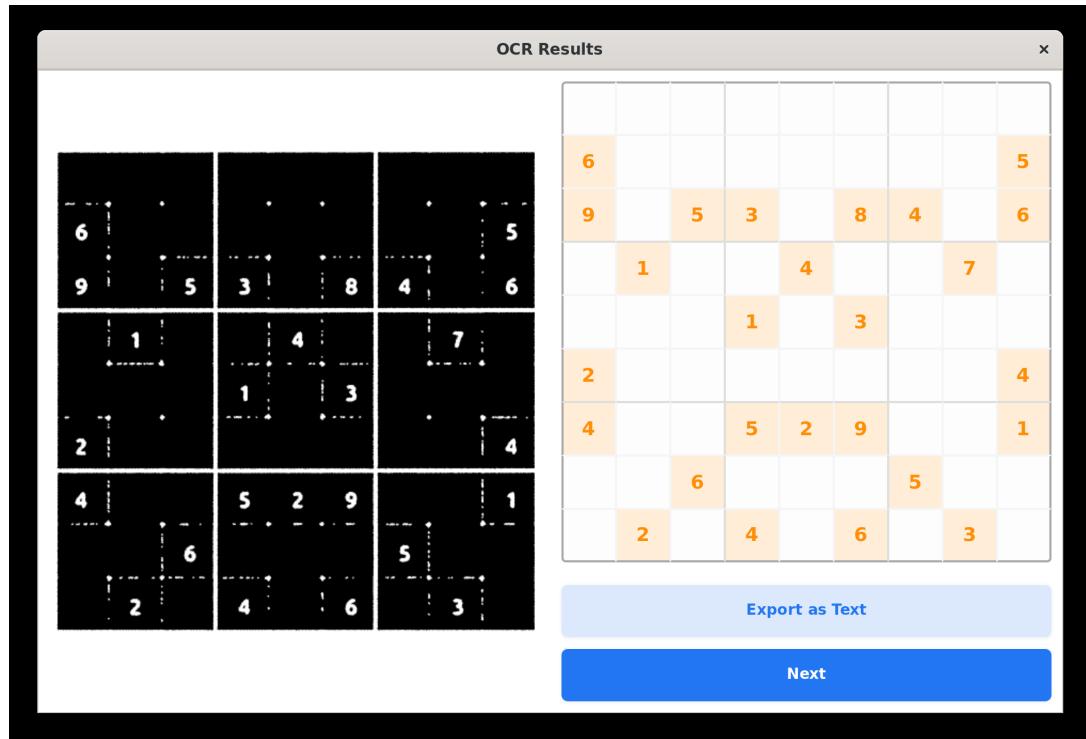


FIGURE 6.6 – Fenêtre de modification de la grille

Une fois confirmée le popup apparaît de nouveau pour prévenir l'utilisateur que le traitement est à nouveau en cours (cette fois-ci pour l'application du réseau de neurones). L'équipe tient à rappeler que toute étape peut être interrompue et ce à chaque instant. L'utilisateur n'est donc pas frustré de devoir attendre la fin du traitement ou de tout le processus pour relancer une analyse ou pour tout simplement arrêter notre application.

Lorsque le réseau de neurones a fini de traiter toutes les cases. Une nouvelle fenêtre s'ouvre. Cette fenêtre est sans aucune hésitation la plus importante de notre application. Elle permet en effet de vérifier que tous les traitements effectués auparavant ont donné les bons résultats et que le réseau de neurones a su reconnaître les nombres correctement.

Pour cela, une matrice de 9x9 est affichée à côté de la grille donnée par l'utilisateur. Cette grille représente les nombres reconnus par le réseau de neurones. Car même si le réseau de neurones est relativement performant, la technologie n'est pas infaillible !

Dans ce cas, il nous faut donc donner la possibilité à l'utilisateur de modifier les valeurs directement et simplement. Pour cela, il suffit de cliquer sur la case à modifier puis de corriger dans le champ de texte dans le popup. Cette petite fenêtre affiche la case de départ ainsi que le nombre reconnu.



FIGURE 6.7 – Popup de modification de la reconnaissance de caractères

### 6.2.6 Prévisualisation du résultat et export

Une fois la grille confirmée par l'utilisateur, la résolution du sudoku est lancée et il suffit d'attendre quelques instants pour que la solution apparaisse sur l'écran de l'utilisateur et pour qu'il puisse la sauvegarder à l'emplacement désiré et ce dans le format souhaité. En effet, nous avons laissé le choix à l'utilisateur de sauvegarder le résultat dans le format texte, c'est à dire de sauvegarder le résultat en fichier ".txt".

La deuxième possibilité est de sauvegarder le résultat dans une grille préchargée et créée par l'équipe. Le but de cette option est d'exporter le résultat dans un style propre à l'application et à montrer une solution du sudoku dans un affichage élégant et moderne et non pas dans la photo qui peut être prise de manière peu présentable et/ou dans un milieu inadapté.

La dernière option sert à sauvegarder la solution directement dans l'image de départ avec les nombres de replacés dans la grille.

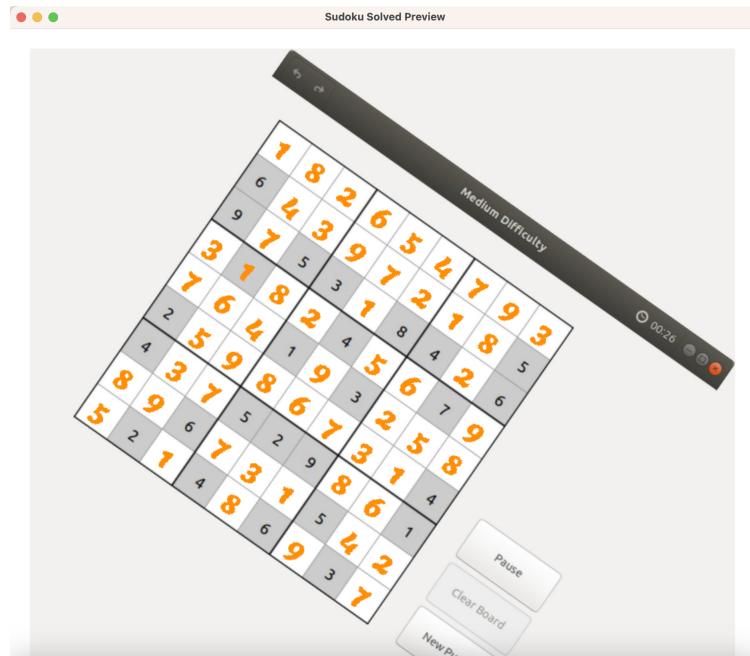


FIGURE 6.8 – Export de la grille 5 avec reconstruction de la grille

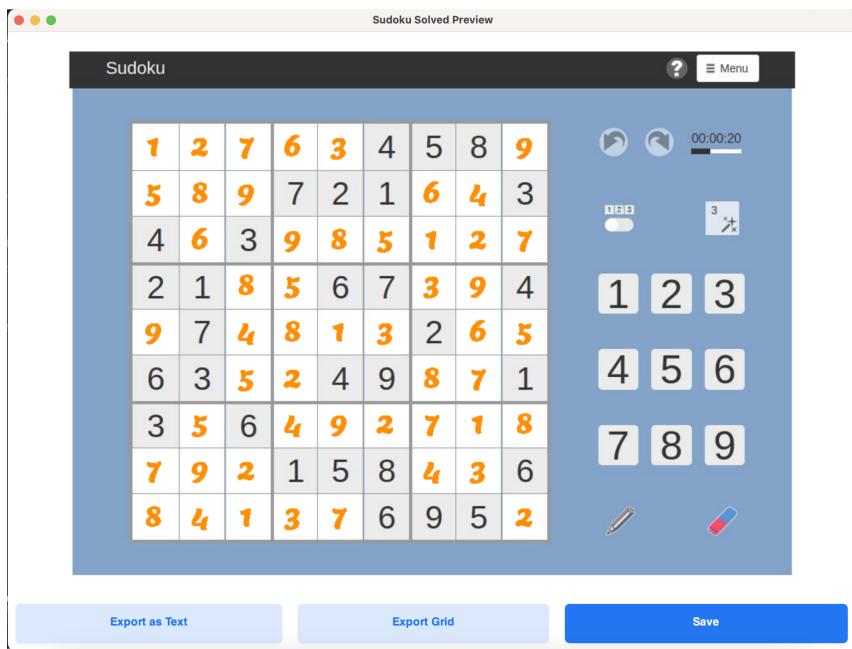


FIGURE 6.9 – Sauvegarde de la grille 3 avec reconstruction de la grille

L’application a donc été pensée pour être non seulement une version graphique mais surtout pour être un véritable outil et surtout avec la possibilité d’avoir la mainmise sur absolument toutes les étapes de la résolution d’une grille de sudoku. L’équipe s’est focalisé sur l’expérience utilisateur et l’ergonomie plutôt que sur les fonctionnalités et sur un design époustouflant.

### 6.2.7 CSS et Gtk

A la grande surprise de l’équipe SudOCRu, Gtk supporte l’intégration de classes de style en CSS, tout comme le HTML, ce qui est vraiment bénéfique pour notre application car le design est très peu satisfaisant par défaut dans GTK comme montré ci-dessous (comparé à l’image 6.2 de la rubrique : L’application et le design).



FIGURE 6.10 – UI sans utilisation du CSS

Pour commencer, il nous fallait diviser les éléments du UI en classes. Pour ce faire, nous avons cherché à faire une maquette de l’application sur Figma, ou du moins les différents types de boutons dont nous avions besoin ainsi que la couleur de ceux-ci.

## La maquette et les couleurs

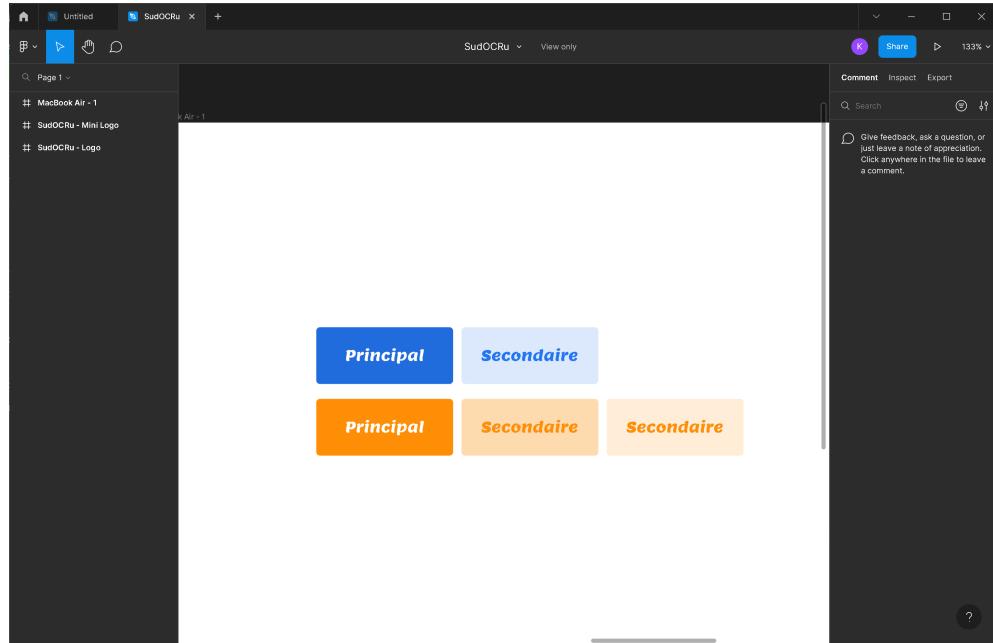


FIGURE 6.11 – Maquette des couleurs et boutons

Pour commencer, nous avons créée deux boutons sur Figma avec des arrondis de 6, des boutons primaires, secondaires et les couleurs des boutons respectifs.

Cette partie était clairement la plus pointilleuse car il fallait mettre des couleurs qui "matchaient" bien ensemble et ne pas faire de contrastes avec les autres couleurs ou éléments du UI. L'équipe a opté pour les couleurs ci-dessus et en "hover", les couleurs deviennent plus sombre simplement.

## Design final

De nombreuses versions se sont succédées mais celles-ci sont les plus modernes, élégantes et moins vives car les autres étaient trop peu aesthetics pour convenir aux goûts de chacun. Dans le code, seulement quelques boutons sont de type : ClassicButton (les boutons secondaires) et les autres sont de type : MainButton.

Les MainButton sont les boutons en bleu foncé et signifient que cette action permet d'accéder à la prochaine étape du cheminement. Elle permet en d'autres termes de confirmer l'étape en cours et d'accéder à la prochaine.

Les MainButton permettent quant à eux d'indiquer que le bouton réalise una action sur l'étape en cours ou d'accéder à d'autres paramètres par exemple.

Cette disposition et ce design sont relativement logiques car elle est en accord avec les design actuels, comme avec les applications natives d'Apple sur IOS par exemple.

## 6.3 Intégration

Pour la partie intégration, l'équipe a eu du mal à commencer et à voir comment rendre la partie back-end visible en front-end. En effet, les membres ont vraiment très bien anticipé la partie affichage, c'est à dire le front-end, ce que l'utilisateur voit. Des les premières semaines, nous nous sommes familiarisés avec l'outil afin de comprendre son fonctionnement ainsi que l'intégration avec le code en C.

La partie la plus difficile était le début du UI avec Glade, car il fallait comprendre comment faire pour lier les fonctions avec des callbacks en back-end. après quelques recherches et quelques applications pratiques, nous étions en mesure de faire l'implémentation. En parallèle, le reste de l'équipe avançait le UI et le réseau neuronal afin d'être dans les temps.

Tout d'abord, après le UI terminé (la partie affichage), il fallait réfléchir à donner des IDs aux éléments que l'on souhaite modifier en back-end.

On a donc utilisé des noms clairs comme OCRCorrection pour la fenêtre qui permet de modifier ce que l'OCR a reconnu, et pour les cells : Cell0, Cell1, Cell2... elles mêmes contenues dans la CellGrid. Comme ceci, en back-end, tout était accessible et logique, pas besoin de revenir dans le ".glade" pour se rappeler du nom de chacunes des variables ou des éléments.

### 6.3.1 Intégration des différentes parties

Pour lier le code, nous avons opté pour la méthode : "Diviser pour régner". En effet, nous avons divisé les codes des différentes fenêtres en plusieurs fichiers distincts afin de garder des fichiers compacts et moins verbeux.

Tout d'abord, l'équipe a créé un fichier qui se nomme "utils" qui permet d'écrire dans le terminal l'état acutel du traitement, de la résolution de grille ou du réseau de neurones.

Le fichier windows.h permet quant à lui de relier tous les projets jusqu'ici séparés en 1 seul fichier. Y est présent : le projet de traitement (application de filtres propres à l'image d'entrée), celui de division de l'image et de détection de grilles, l'OCR, le SudokuSolver ainsi que les projets d'affichage de l'image, rendu... Ce fichier est donc essentiel pour lier le code de chacun de ces projets au UI. Il suffira d'appeler les fonctions de chacun des projets et de les mettre en commun pour faire la liaison.

Enfin, pour importer tout le code réalisé par notre équipe, nous avons utilisé des librairies statiques. En effet, elles sont composées d'un fichier archive (.a) qui sont des collections de fichiers objets (.o) générés par le compilateur (avant l'étape de linking). Les fichiers .o contiennent tous le code exécutable de nos fonctions, en effet, les fichiers headers ne contiennent pas de code. On ajoute alors dans chaque Makefile des sous-parties (ocr, solver, ipp) une règle pour générer ce fichier de librairie qu'on va ensuite copier dans le dossier principal qui contient l'application. Une fois les librairies statiques présentes, la compilation peut poursuivre pour générer l'exécutable principal.

# Chapitre 7

## 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.

### 7.1 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.

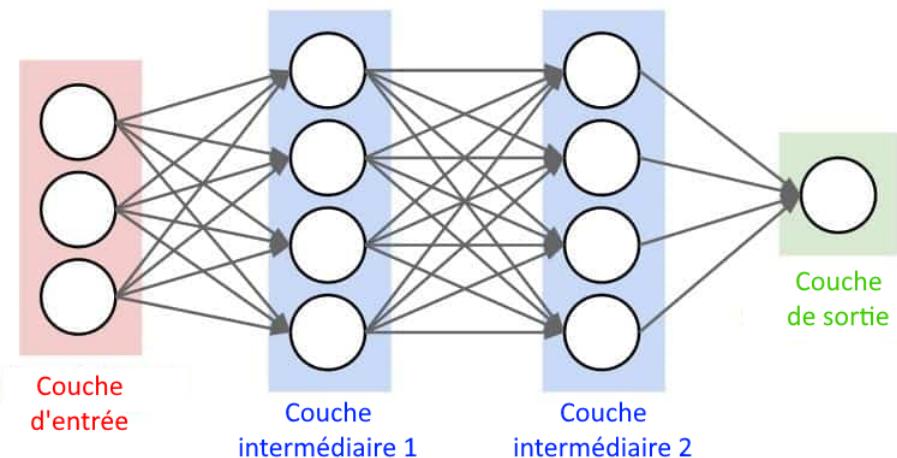


FIGURE 7.1 – Structure d'un réseau de neurone pour représenter le OU exclusif

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  $\sigma$  une fonction d'activation. On a :

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

## 7.2 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).

## 7.3 Notre implémentation du réseau de neurones

Pour pouvoir reconnaître les chiffres dans des conditions les plus optimales possible, il a fallu nous mettre d'accord sur des conventions à respecter. Tout d'abord, toute les images en entrée doivent faire une taille de 28x28 pixels. Cette taille n'a pas été choisie au hasard. En effet le MNIST (Mixed National Institute of Standards and Technology) utilise notamment cette taille. Autre convention décidée de notre part, est que les chiffres doivent être de taille maximale sur l'image, c'est-à-dire qu'il doit toucher au maximum

les bords, en particulier les bords supérieur et inférieurs de l'image.

Maintenant que ces bords ont été fixés, il a fallu choisir la taille du réseau de neurones. Nous avons opté pour un réseau en 4 couches : La première couche possède 784 entrées, qui correspond au nombre total de pixels contenu dans une image de taille 28x28, deux couches intermédiaires de chacune 50 neurones, puis finalement une couche de sortie de 10 neurones, qui chacune correspond à un chiffre. Par exemple le 4ème neurone de sortie correspond à la probabilité à ce que tout le calcul effectué par le réseau de neurones sur une image soit un 4.

Malheureusement, il n'a pas été possible d'utiliser la banque de donnée MNIST, qui contient des images de chiffres manuscrites, donc nous avons opté pour des chiffres tapés sur ordinateur. La condition nécessaire à un réseau de neurones qui puisse reconnaître un maximum de chiffres est de pouvoir l'entrainer avec beaucoup de données. Créer des centaines voire des milliers d'images contenant chacun un chiffre, avec une police et des variations en termes de rotation, de taille et de positionnement différents est très fastidieux. Pour pouvoir créer cette banque de donnée personnalisée de manière efficace et rapide, un programme en python a été créé.

Ce programme python possède trois fonctionnalités : La première est la génération d'images contenant des chiffres, la deuxième fonctionnalité est la conversion de ces différentes images en deux fichier binaires possédant le même format utilisé que le MNIST. Ces deux fichiers sont écrits d'une manière organisée. Le premier fichier contiendra les pixels de tout les images écrit de manière linéaire. Le deuxième fichier contiendra les résultats attendus des images écrits précédemment dans le même ordre de traitement.

#### **TRAINING SET LABEL FILE (train-labels-idx1-ubyte):**

```
[offset] [type]      [value]      [description]
0000  32 bit integer 0x000000801(2049) magic number (MSB first)
0004  32 bit integer 60000        number of items
0008  unsigned byte ??           label
0009  unsigned byte ??           label
.....
xxxx  unsigned byte ??           label

The labels values are 0 to 9.
```

#### **TRAINING SET IMAGE FILE (train-images-idx3-ubyte):**

```
[offset] [type]      [value]      [description]
0000  32 bit integer 0x000000803(2051) magic number
0004  32 bit integer 60000        number of images
0008  32 bit integer 28          number of rows
0012  32 bit integer 28          number of columns
0016  unsigned byte ??          pixel
0017  unsigned byte ??          pixel
.....
xxxx  unsigned byte ??          pixel
```

Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (white), 255 means foreground (black).

FIGURE 7.2 – Format des fichiers dans la base de donnee du MNIST

La troisième fonctionnalité est le test de plusieurs images sur une interface. Le programme va lire toutes les images contenues dans un dossier et les lires un par un. Dans l'interface sera affiché l'image lue et le résultat du réseau de neurones.

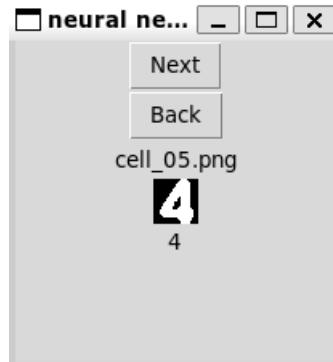


FIGURE 7.3 – Fenêtre de vérification 1

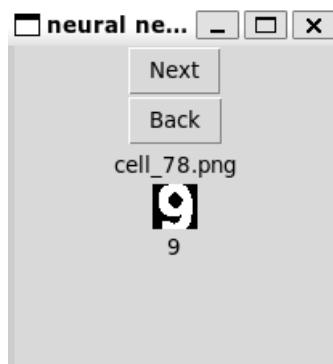


FIGURE 7.4 – Fenêtre de vérification 2

Pour avoir un maximum d'images contenant des chiffres très variés, la première fonctionnalité de ce programme va être détaillée. Tout d'abord, à partir d'un dossier contenant plusieurs fichiers de polices, des images contenant pour chacun des chiffres compris entre 0 et 9 seront exportés. Chaque chiffre écrit avec chaque police sera exporté avec des variations différentes. Ces dites variations sont de légères rotations droite/gauche et de légères translations haut/bas/droite/gauche.



FIGURE 7.5 – Samples de test

# Chapitre 8

## Conclusion

En conclusion, l'équipe SudOCRU est globalament satisfaite du projet. La totalité du travail qui a été demandé a été remis et ce avec beaucoup de sérieux. Les résultats sont plus que convenables et sauront couvrir la plupart des situations et répondre à la plupart des problématiques posées par les utilisateurs. En effet, comme décrit précédemment, notre application répond parfaitement aux demandes des utilisateurs. Tout d'abord, un design moderne, élégant et très simple d'utilisation. Un traitement efficace et modifiable si des erreurs de traitements ont été rencontrées, il en va de même pour la reconnaissance de caractères. Tout est donc très simple d'utilisation, pour ne pas perdre et frustrer l'utilisateur sans pour autant laisser de côté la correction manuelle qui peut être faite à n'importe quel moment de la chaîne de traitement ou le choix du type d'export de fichiers. L'équipe SudOCRU est par conséquent, très fière de présenter ce projet avec beaucoup de sérieux et de fierté comme un projet fini et utilisable par le plus grand nombre.