

EPITA



OCR Solveur de sudoku

Rapport de projet

Owen Boisgontier, Mathieu Even, Axel Lelong & Jules Raitière-Delsupexhe
SPE R2 Epita 2026

08 Décembre 2022

Table des matières

1	Introduction	3
2	Image	3
2.1	Chargement d'image	3
2.2	Prétraitement	3
2.2.1	Filtre Grayscale	4
2.2.2	Filtre de contraste	5
2.2.3	Normalisation des éclairages	6
2.2.4	Filtre médian	7
2.2.5	Filtre moyen	9
2.2.6	Filtre de seuil adaptatif	10
2.2.7	Filtre de lissage	11
2.3	Détection de la grille et des cases	12
2.3.1	Filtre de Sobel	12
2.3.2	Détection des lignes	13
2.3.3	Simplification des lignes	15
2.3.4	Rotation de l'image	15
2.3.5	Détection de la grille	16
2.3.6	Correction de la perspective et recadrage	18
2.3.7	Découpage des cases	20
2.3.8	Redimension des cases	20
2.3.9	Mise au propre des images	21
3	Réseau de neurones	23
3.1	Réseau XOR	23
3.2	Sauvegarde et chargement des poids	25
3.3	Réseau de neurones final	26
4	Résolution	27
4.1	Algorithme de résolution	27
5	Application	28
5.1	Interface Utilisateur	28
6	Planning	31

7 Conclusion	32
7.1 Ressenti	32
7.1.1 Owen	32
7.1.2 Axel	32
7.1.3 Jules	32
7.1.4 Mathieu	33
7.2 Conclusion	33

1 Introduction

Pour rappel, le projet que nous réalisons est un OCR (Optical Character Recognition) capable de reconnaître une grille de sudoku peu importe l'image, de détecter les chiffres présents dedans, de résoudre cette grille et de la renvoyer complétée.

Ainsi, cela fait environ 4 mois que notre projet est lancé, et grâce à un investissement régulier de l'ensemble des membres de l'équipe, celui-ci est enfin terminé, et nous allons vous expliquer comment il fonctionne.

2 Image

2.1 Chargement d'image

Avant de pouvoir modifier des images, il faut d'abord être capable de les charger et de les mettre dans le bon format afin de les traiter correctement. Pour cela, nous avons utilisé la bibliothèque *SDL2* disponible sur les ordinateurs de l'école. Dans un premier temps, l'image doit être convertie en surface par la fonction de *SDL* : *IMGLOAD*. Ensuite, nous devons changer le format de cette surface en format pixels à l'aide de la fonction *SDLConvertSurfaceFormat*. De cette nouvelle surface est alors accessible le tableau de pixels de l'image que l'on peut modifier. C'est à partir de celui-ci que la transformation de l'image va avoir lieu et le pré-traitement peut commencer.

2.2 Prétraitement

Le prétraitement est une étape très importante. Sans cette partie, il serait impossible de reconnaître les caractères et la grille dans les images. L'objectif de cette section est de préparer l'image pour la rendre lisible et que la détection des lignes et des caractères fonctionne. Cette section consiste principalement à réduire le bruit de l'image sans perdre d'information importante.

Durant l'application des différents filtres, nous illustrerons nos propos avec l'image suivante :

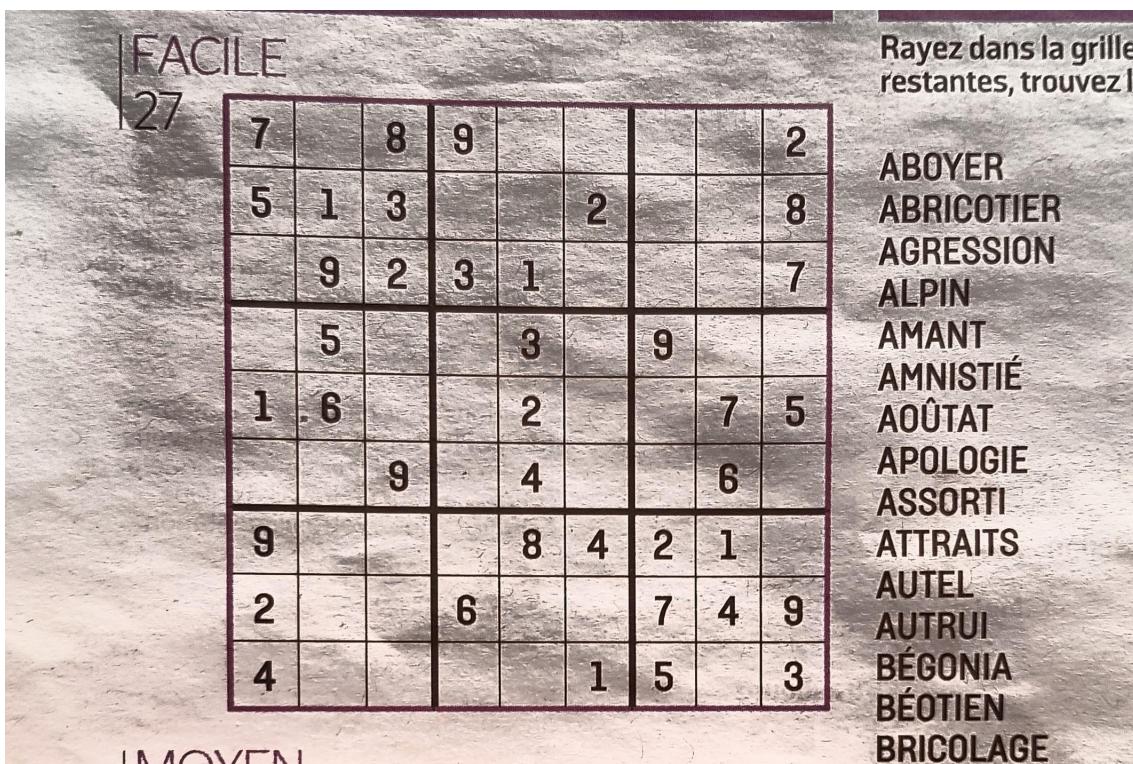


FIGURE 1 – Image support

2.2.1 Filtre Grayscale

Pour commencer, nous passons l'image qui est en couleur en nuances de gris. Pour ce faire, nous appliquons sur chaque pixel la formule suivante :

$$p_{(x,y)} = p_{(x,y)r} * 0.3 + p_{(x,y)g} * 0.59 + p_{(x,y)b} * 0.11$$

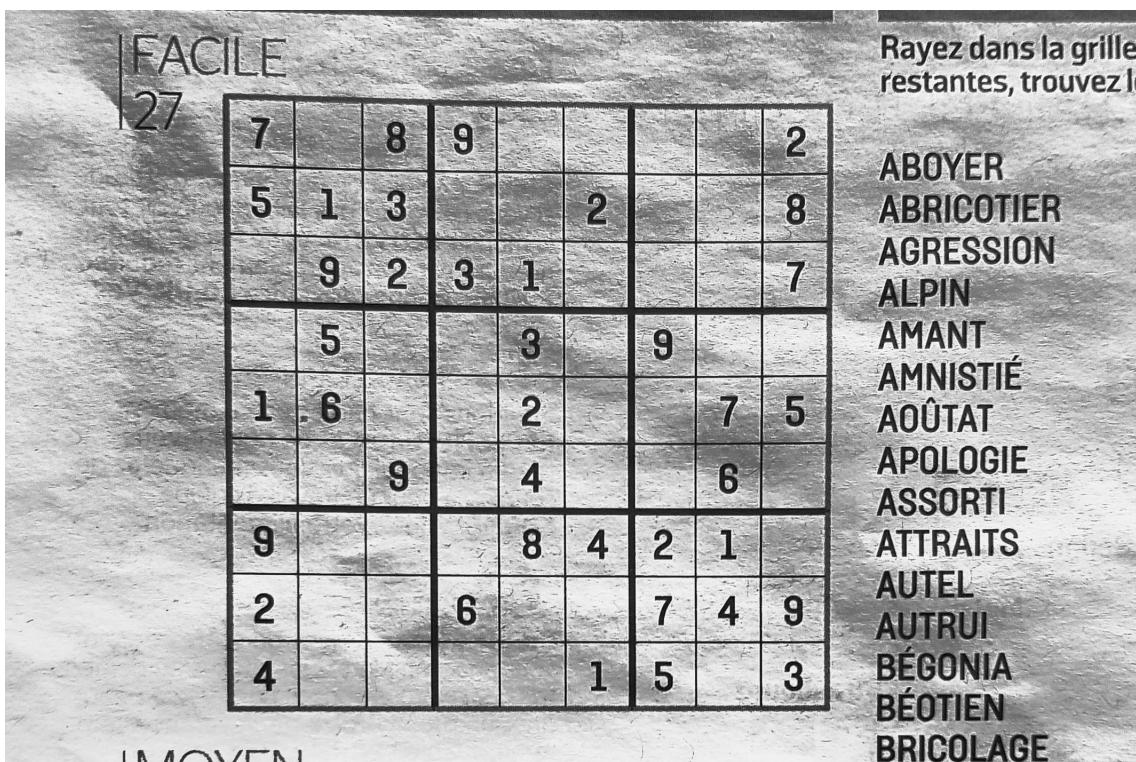


FIGURE 2 – Image après l’application du filtre grayscale

2.2.2 Filtre de contraste

Puis, nous appliquons un filtre d’augmentation des contrastes, qui va permettre de bien distinguer les écritures des éléments parasites. L’application de ce filtre nécessite un facteur, qui dans notre cas est fixé à 10. L’algorithme est le suivant :

Pour chaque pixel dans l’image

- On itère avec un index de 0 jusqu’au facteur
- Si la valeur du pixel appartient à l’intervalle $[index \times (255/facteur), (index + 1) \times (255/facteur)]$, alors nous modifions cette dernière par la borne supérieure de l’intervalle.
- Pour finir, on inverse la couleur du pixel pour le prochain filtre ($p_{(x,y)} = 255 - p_{(x,y)}$)

Le résultat est le suivant :

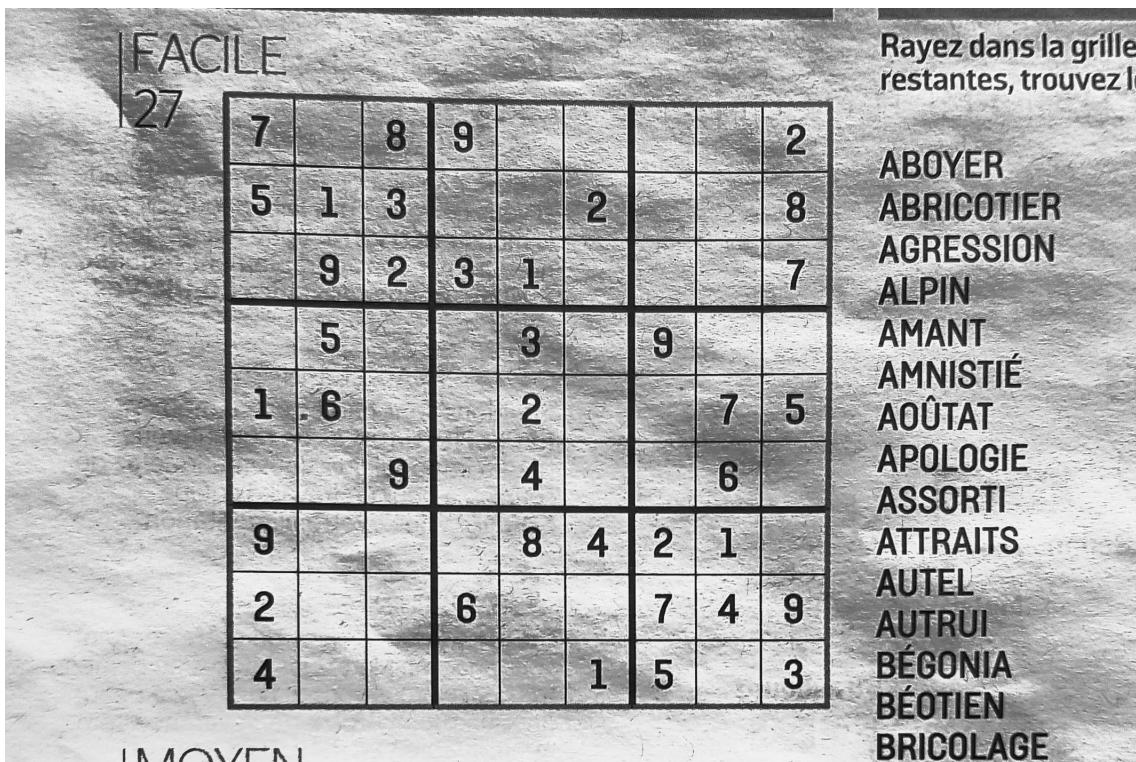


FIGURE 3 – Image après l’application du filtre de contraste

2.2.3 Normalisation des éclairages

Ensuite, nous normalisons l’éclairage pour créer une cohérence entre les différents niveaux d’éclairage avant d’appliquer les prochains filtres. Il suffit d’appliquer sur chaque pixel la formule suivante :

$$p_{(x,y)} = 255 - p_{(x,y)} * (255/m)$$

'm' étant la valeur maximale de l'image

On obtient le résultat suivant :

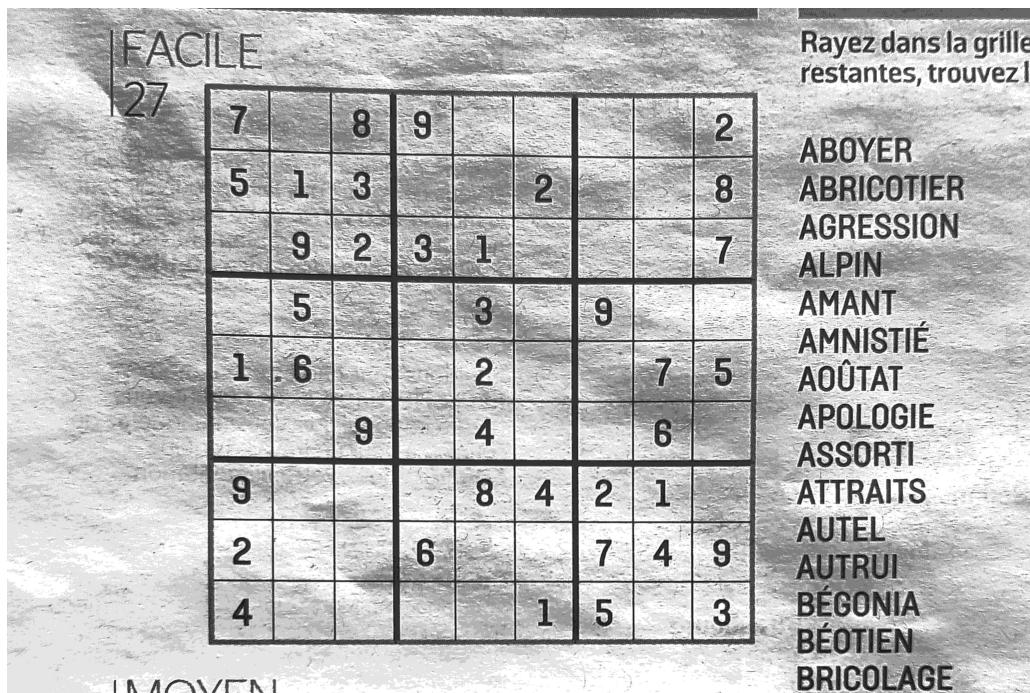


FIGURE 4 – Image après la normalisation des éclairages

2.2.4 Filtre médian

Le filtre médian permet d'éliminer les valeurs aberrantes de l'image. Ce filtre efface donc les bruits impulsionnels. L'application de ce filtre se résume en ces étapes :

- On récupère une matrice 3x3 qui correspond au pixel et ses 8 voisins et on met cette matrice dans un tableau de taille 9.
- On trie cette matrice dans l'ordre croissant
- On récupère la valeur médiane des voisins, cette à dire à la place 4 dans le tableau.

20	20	1	20	10	20	10	13	12
20	0	0	0	0	0	0	20	20
20	0	90	90	90	90	90	20	20
20	0	90	0	90	90	90	20	20
10	0	90	90	90	90	90	10	10
10	0	90	90	90	90	90	10	10
10	0	90	90	90	90	90	90	10
20	0	0	0	0	0	0	20	20
20	20	10	20	10	20	10	13	13

FIGURE 5 – Matrice 3x3 d'un voisinage de pixel (en rouge)

Pour la matrice ci-dessus, la valeur 0 sera remplacé par 90.

0	90	90	90	90	90	90	90	90
---	----	----	----	----	----	----	----	----

FIGURE 6 – Tableau de voisinage d'un pixel rangé dans l'ordre croissant



Lorsqu'on cherche les voisins au bord de l'image, il va nous manquer certains pixels. Il existe plusieurs technique pour pallier ce manque. Nous avons choisi celle du "Zéro Padding", cette méthode consiste à admettre que les pixels manquant sont des pixels noirs (et donc on place la valeur zéro dans le tableau des voisins à la place de ces pixels).

Le résultat est le suivant :

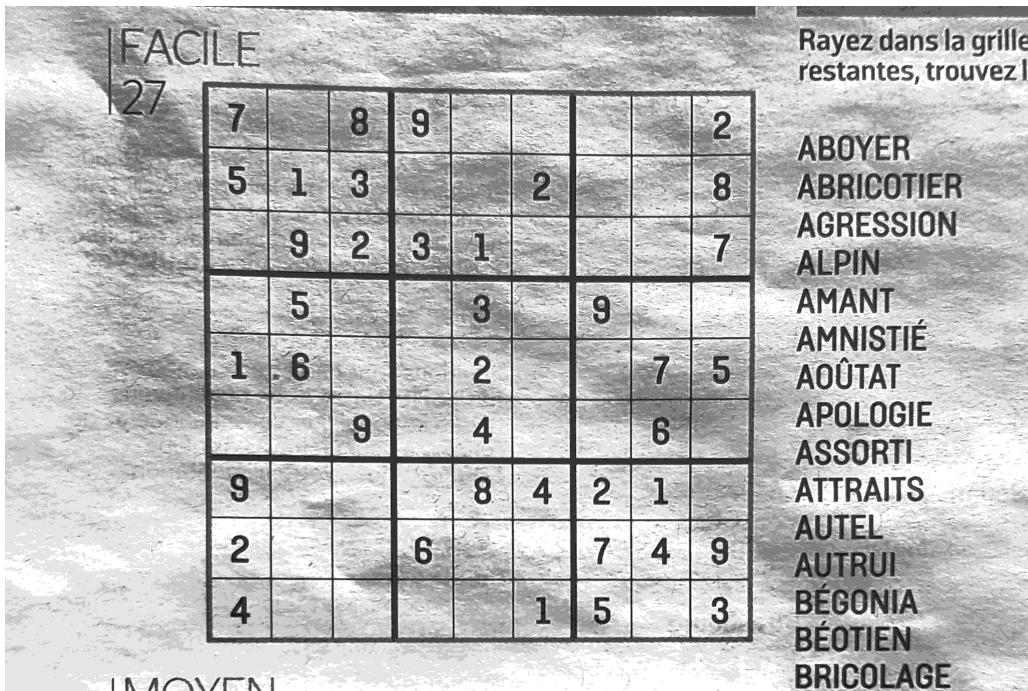


FIGURE 7 – Image après l’application du filtre médian

2.2.5 Filtre moyen

Le filtre moyen, comme son nom l’indique, calcule la moyenne des pixels situés dans le voisinage de chaque pixel. Ce filtre permet de réduire le bruit dans l’image, c’est-à-dire tous les pixels isolés, ce qui rend les zones homogènes plus lisses. Par contre, les contours sont fortement dégradés, et les structures trop fines peuvent devenir moins visibles. En reprenant la matrice de voisinage de la figure ??, on multiplie la matrice des voisins par un masque de convolution h_0 . Ainsi, on a :

$$p(x, y) = h_0 * \begin{bmatrix} 90 & 90 & 90 \\ 90 & 0 & 90 \\ 90 & 90 & 90 \end{bmatrix}$$

$$h_0 = 1/9 * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Pour obtenir des résultats plus propres, nous avons utilisé le Flou de Gauss. Pour cela, nous avons changé le masque de convolution h_0 en utilisant les coefficients du

triangle de Pascal :

$$h_{pascal} = 1/16 * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Ce qui donne le résultat suivant :

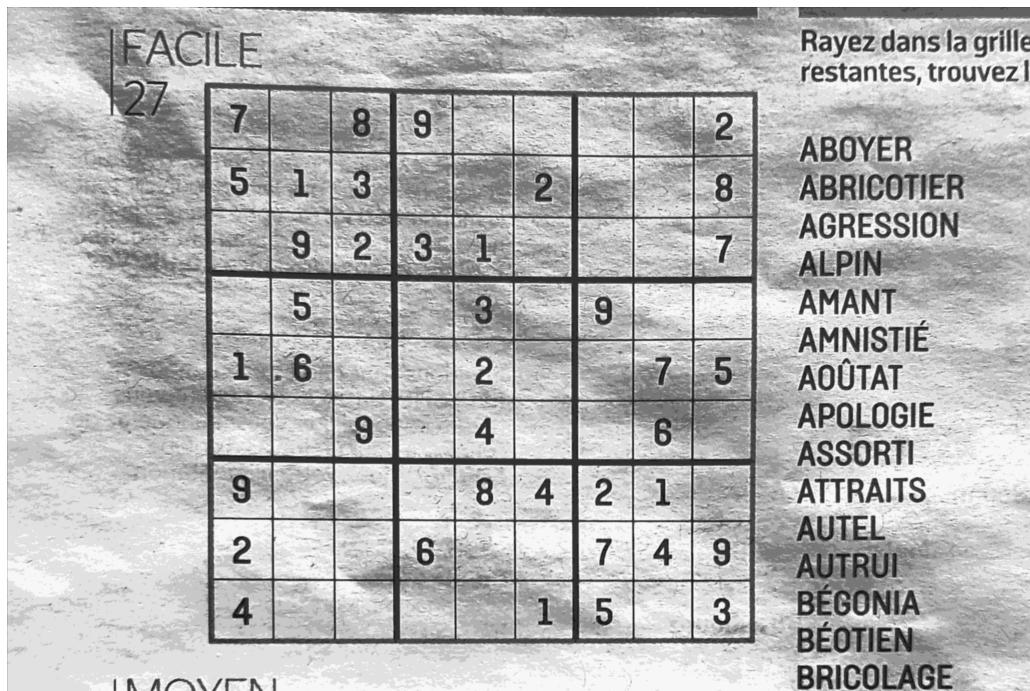


FIGURE 8 – Filtre Moyen

2.2.6 Filtre de seuil adaptatif

Le principe de seuil adaptatif est similaire à l'algorithme d'Otsu. Le principe de ce dernier est de calculer un seuil global pour l'image, et si la valeur d'un pixel est supérieure au seuil, alors le pixel devient blanc sinon noir. On va alors se retrouver avec seulement des pixels blancs et noirs, c'est la binarisation. Le but est de travailler sur des petites zones de l'image pour avoir plus de précision. L'image est donc divisée en sous-images plus petites. Ensuite, nous calculons la valeur de chaque seuil de ces images et l'appliquons à l'image globale. La taille de ces sous-images est très importante, et nous la déterminons en calculant le niveau de bruit dans l'image.

Pour calculer le niveau de bruit de l'image globale, nous appliquons l'algorithme suivant :

- Pour chaque pixel dans l'image,

- On récupère la valeur du pixel et de ses voisins dans un tableau de taille 9.
- On calcule la moyenne m des valeurs de ce tableau.
- On applique le calcul suivant, si $(1 - p_{(x,y)}/m) >$ seuil de bruit alors, on considère le pixel actuel comme faisant partie du bruit (Le seuil de bruit est fixé à 0.5)

Ensuite, en fonction du niveau de bruit dans l'image, nous appliquons le seuil adaptatif avec une taille des sous-images plus ou moins importante. Ce qui donne le résultat suivant :

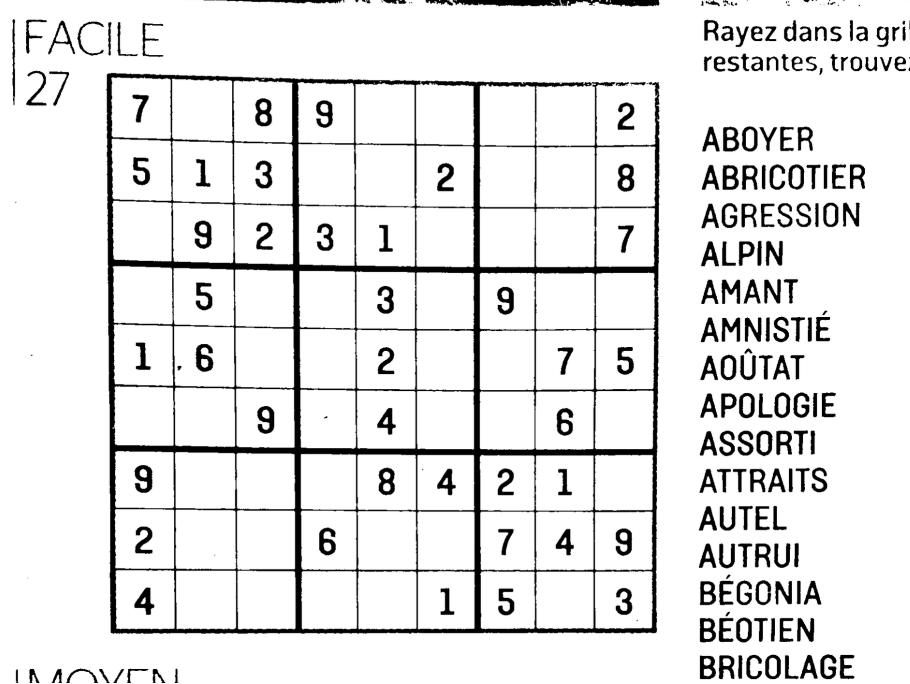


FIGURE 9 – Filtre de seuil adaptatif

2.2.7 Filtre de lissage

Le filtre de lissage, comme son nom l'indique, a pour fonction de "lisser l'image". C'est-à-dire qu'il permet d'épaissir les éléments importants de l'image : chiffres et lignes. Pour ce faire, nous utilisons l'algorithme hysteresis, qui est une des étapes du filtre de Canny, expliqué plus tard dans la détection des lignes. L'algorithme est assez simple :

Pour chaque pixel blanc de l'image : Si au moins un de ses voisins est noir, le pixel devient noir.

On obtient le résultat suivant. On peut observer que les caractères et les lignes se sont épaissies :

FACILE

27

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

MOVEMENT

Rayez dans la grille restantes, trouvez le

**ABOYER
ABRICOTIER
AGGRESSION
ALPIN
AMANT
AMNISTIÉ
AOÛTAT
APOLOGIE
ASSORTI
ATTRATS
AUTEL
AUTRUI
BÉGONIA
BÉOTIEN
BRICOLAGE**

FIGURE 10 – Filtre de lissage

2.3 Détection de la grille et des cases

2.3.1 Filtre de Sobel

Le filtre de Sobel est une fonction utilisée en traitement d’image pour la détection de contours. Il s’agit d’un des opérateurs les plus simples qui donne toutefois des résultats corrects. Le but étant de calculer le gradient de l’intensité de chaque pixel.

Du fait de sa simplicité, le filtre de Sobel peut être aisément implémenté. En effet, seulement huit points autour d’un pixel considéré sont nécessaires pour calculer le gradient. Ce calcul utilise simplement des calculs sur les entiers. De plus, les filtres horizontaux et verticaux sont séparables :

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

La matrice horizontale de Kernel.

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

La matrice verticale de Kernel.

En chaque pixel de l'image d'origine, les approximations des gradients horizontaux et verticaux peuvent être combinées comme suit :

$$G = \sqrt{Gx^2 + Gy^2}$$

L'image G résultante fait donc ressortir les contours de l'image :

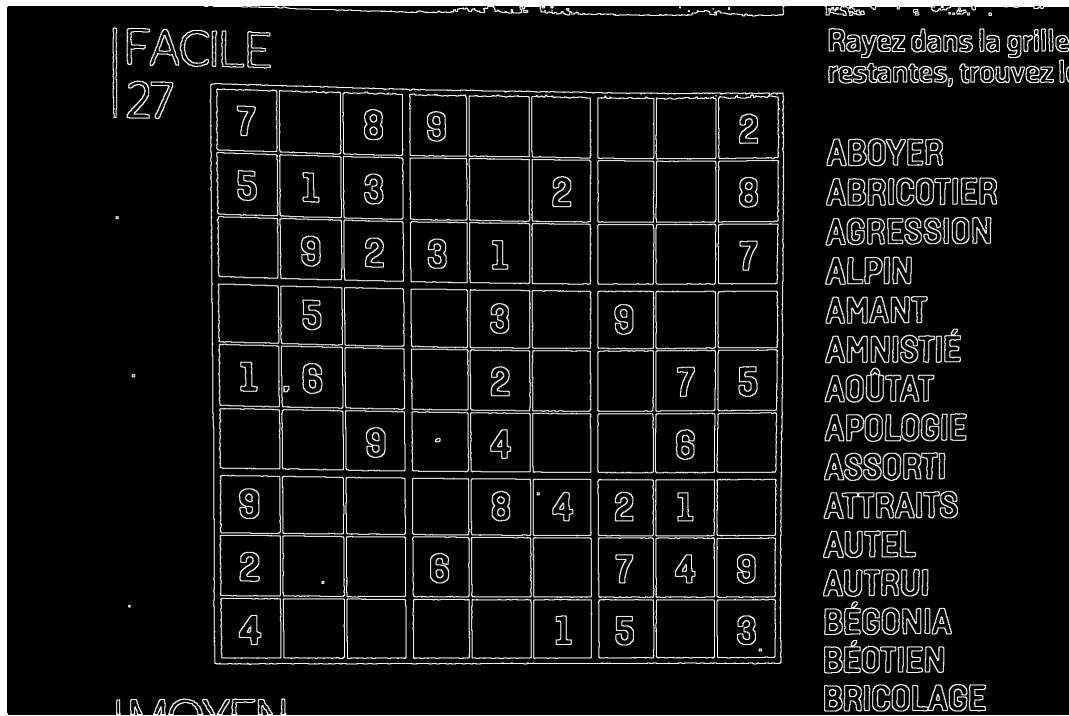


FIGURE 11 – Image après application du filtre de Sobel

2.3.2 Détection des lignes

La transformée de Hough est une méthode qui consiste à parcourir tous les pixels d'une image et de renvoyer toutes les lignes qui la composent. Nous utilisons donc cet algorithme pour détecter la grille du sudoku.

Nous commençons par définir la diagonale de l'image ($diag$), un angle θ (allant de 0° à 180°) et un ρ (allant de $-diag$ à $diag$). À partir de ces valeurs, on initialise une matrice de taille $(180 * diag)$ remplie de zéro. Cette matrice va nous servir

d'accumulateur. Ensuite, pour chaque pixel blanc, on calcule ρ pour chaque valeur de θ :

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

Puis on incrémente la valeur de accumulateur[ρ][θ]. Pour trouver les lignes, on parcourt notre accumulateur et pour chaque point de coordonnées (ρ, θ) au-dessus du seuil défini au préalable : On récupère les coordonnées polaires du pixel et on en déduit les coordonnées cartésiennes (x, y) grâce à la formule suivante.

$$x_0 = \rho * \cos(\theta) \quad y_0 = \rho * \sin(\theta)$$

Ces coordonnées nous permettent de déterminer les coordonnées de début et de fin de la ligne grâce aux formules suivantes :

$$x_{debut} = x_0 + (diag * \cos(\theta))$$

$$y_{debut} = y_0 + (diag * \sin(\theta))$$

$$x_{fin} = x_0 - (diag * \cos(\theta))$$

$$y_{fin} = y_0 - (diag * \sin(\theta))$$

Nous stockons ces coordonnées pour la suite et nous pouvons tracer les lignes. Ce qui nous donne le résultat suivant :



FIGURE 12 – Détection des lignes

2.3.3 Simplification des lignes

Souvent, la transformée de Hough renvoie des lignes parfois très proches, ces lignes doivent être simplifiées. Pour ce faire, nous appliquons l'algorithme suivant :

Pour les lignes L_1 et L_2 :

Si $|(x_1, y_1)_{debut} - (x_2, y_2)_{debut}| < \text{seuil de tolérance}$
et $|(x_1, y_1)_{fin} - (x_2, y_2)_{fin}| < \text{seuil de tolérance}$

Alors, on fait la moyenne des lignes :

$$\begin{aligned}x_{dbut} &= (x_{0_{debut}} + x_{1_{debut}})/2 \\y_{dbut} &= (y_{0_{debut}} + y_{1_{debut}})/2 \\x_{fin} &= (x_{0_{fin}} + x_{1_{fin}})/2 \\y_{fin} &= (y_{0_{fin}} + y_{1_{fin}})/2\end{aligned}$$

2.3.4 Rotation de l'image

Une image n'est pas toujours droite, il est alors essentiel de pouvoir la tourner. En effet, il est beaucoup plus simple de travailler sur une image dans le bon sens. Pour effectuer cette rotation, nous avons appliqué l'algorithme suivant :

- Faire la copie de l'image
- Pour chaque pixel :
 - Calculer la position du nouveau pixel dans l'image
 - $x_{new} = mid_h + (x_{old} - mid_h) * \cos(\text{angle}) - (y_{old} - mid_w) * \sin(\text{angle})$
 - $y_{new} = mid_w + (y_{old} - mid_w) * \cos(\text{angle}) + (x_{old} - mid_h) * \sin(\text{angle})$
 - Avec angle, l'angle de rotation en degré, mid_h , la hauteur du tableau de pixels/2 et mid_w , la largeur du tableau de pixels/2
- Si les nouvelles coordonnées appartiennent à l'image alors le pixel prend la valeur du nouveau pixel
- Sinon le pixel devient noir

Une rotation peut alors ressembler à ça :

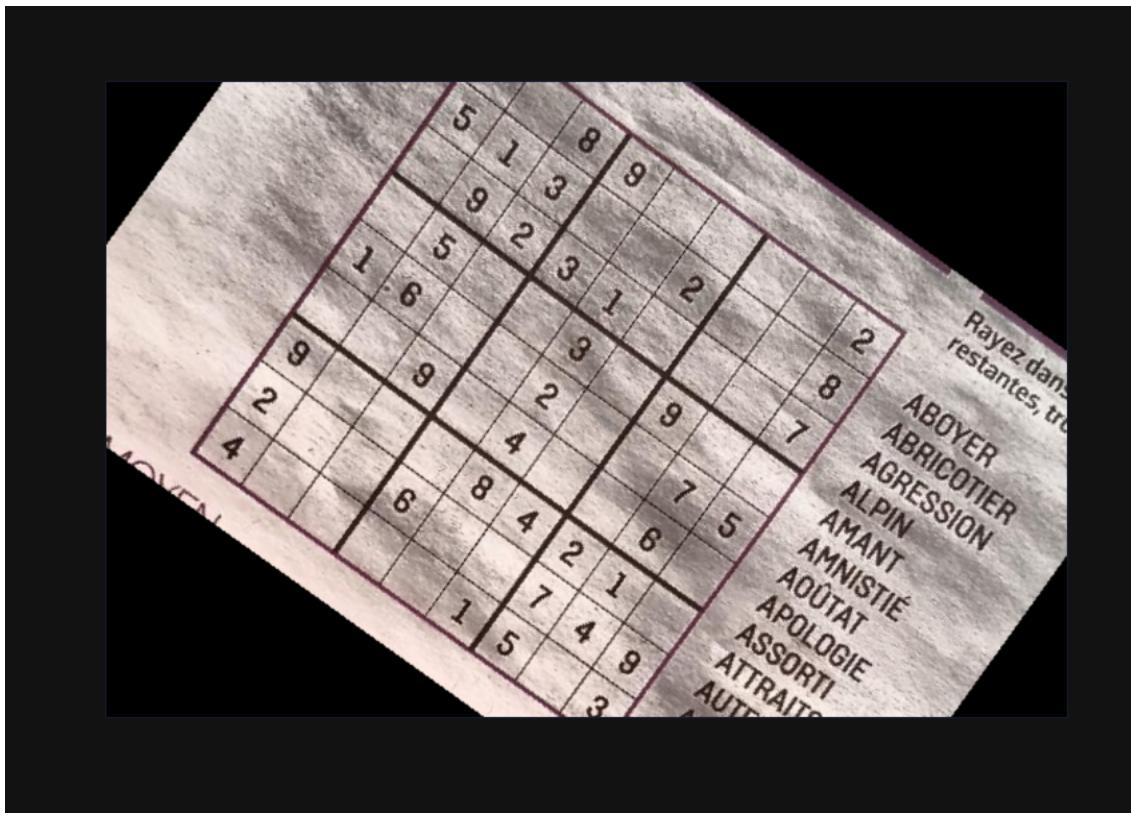


FIGURE 13 – Rotation de 30° sur l'image₁

Après avoir effectué la rotation manuelle, le but est de la rendre automatique grâce à la transformée de hough. Comme dit plus tôt, l'image n'est pas forcément toujours droite. Ainsi lors de la détection des lignes par la transformée de hough, nous récupérons l'angle prédominant dans les lignes simplifiées. Cet angle va nous permettre de tourner l'image et par conséquent la mettre dans le bon sens. Cependant pour détecter le carré de la grille du sudoku, nous devons aussi tourner les lignes simplifiées à l'aide des mêmes formules afin qu'elle soient en adéquation avec l'image tournée.

2.3.5 Détection de la grille

Une fois correctement orientée, nous pouvons détecter les carrés présents dans l'image, de manière à obtenir celui de la grille. Pour cela, nous allons détecter tous les quadrilatères qui se rapprochent d'un carré dans l'image. Ainsi si il y a un petit peu de perspective ou de distorsion nous pourrons quand même obtenir le carré de la grille et corriger cela par la suite.

Pour détecter les carrés de l'image nous appliquons l'algorithme suivant :

- On prend une ligne parmi nos lignes simplifiées et on cherche toutes ses intersections.
- On relance le processus sur les lignes avec lesquelles il y a une intersection.
- Pour obtenir des carrés, on relance le processus 4 fois
- Si au bout des 4 itérations la ligne l à une intersection avec la ligne de départ (alors c'est un quadrilatère)
 - Si leur (plus grand cote - plus petit côté) > 40
 - Alors le quadrilatère est un carré
 - Enregistrer le carré
 - Si le périmètre du carré est plus grand que le carré maximal trouvé alors il devient le carré maximal

Ainsi l'algorithme nous donne le plus grand carré qui correspond à la grille du sudoku. Cette grille peut être déformée, c'est pour cela qu'il faut corriger la perspective.

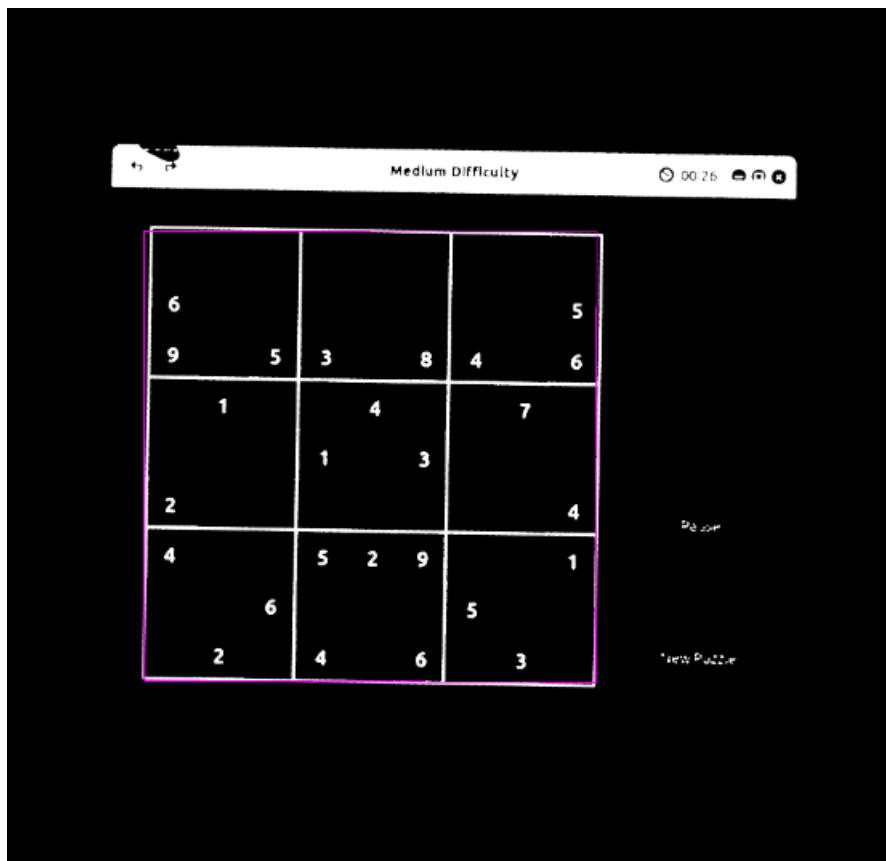


FIGURE 14 – Carré de la grille de Sudoku

2.3.6 Correction de la perspective et recadrage

Une fois le carré détecté, il faut en faire une image avec la grille en plein écran de manière à correctement la segmenter. Pour cela, nous appliquons un algorithme pour à la fois corriger la distorsion et recadrer correctement l'image. Pour se faire voici comment nous procédons :

- En paramètre nous avons les quatre coins de la grille $[(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)]$ et en sortie désirée les quatre coins de l'image $[(x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3), (x'_4, y'_4)]$
- Ceci est la matrice $2 * 9$ qui représente un point

$$P_i = \begin{bmatrix} -x_i & -y_i & -1 & 0 & 0 & 0 & x_i x'_i & y_i x'_i & x'_i \\ 0 & 0 & 0 & -x_i & -y_i & -1 & x_i y'_i & y_i y'_i & y'_i \end{bmatrix}$$

- Nous pouvons donc écrire la matrice P représentant notre problème

$$P = \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1 x'_1 & y_1 x'_1 & x'_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1 y'_1 & y_1 y'_1 & y'_1 \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2 x'_2 & y_2 x'_2 & x'_2 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2 y'_2 & y_2 y'_2 & y'_2 \\ -x_3 & -y_3 & -1 & 0 & 0 & 0 & x_3 x'_3 & y_3 x'_3 & x'_3 \\ 0 & 0 & 0 & -x_3 & -y_3 & -1 & x_3 y'_3 & y_3 y'_3 & y'_3 \\ -x_4 & -y_4 & -1 & 0 & 0 & 0 & x_4 x'_4 & y_4 x'_4 & x'_4 \\ 0 & 0 & 0 & -x_4 & -y_4 & -1 & x_4 y'_4 & y_4 y'_4 & y'_4 \end{bmatrix}$$

- Ici, notre but est de trouver la matrice $3 * 3$ homo-graphique H associée tel que $PH = 0$. En considérant $h_9 = 1$, on obtient :

$$PH = \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1 x'_1 & y_1 x'_1 & x'_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1 y'_1 & y_1 y'_1 & y'_1 \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2 x'_2 & y_2 x'_2 & x'_2 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2 y'_2 & y_2 y'_2 & y'_2 \\ -x_3 & -y_3 & -1 & 0 & 0 & 0 & x_3 x'_3 & y_3 x'_3 & x'_3 \\ 0 & 0 & 0 & -x_3 & -y_3 & -1 & x_3 y'_3 & y_3 y'_3 & y'_3 \\ -x_4 & -y_4 & -1 & 0 & 0 & 0 & x_4 x'_4 & y_4 x'_4 & x'_4 \\ 0 & 0 & 0 & -x_4 & -y_4 & -1 & x_4 y'_4 & y_4 y'_4 & y'_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

- Ensuite, nous pouvons résoudre le système suivant de manière à trouver la matrice homographique de notre problème et ainsi pourvoir calculer la position des nouvelles coordonnées de chaque pixels On appellera $[0, 0, 0,$

$[0, 0, 0, 0, 0, 1]$ R pour la suite.

- Pour se faire nous procémons de la sorte :
- On inverse la matrice P
- On obtient $H = P^{-1} * R$
- On inverse la matrice obtenu de manière à obtenir la matrice homographique H

$$H = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1 & \beta_2 & \beta_3 \\ \gamma_1 & \gamma_2 & \gamma_3 \end{bmatrix}$$

- Ensuite pour chaque pixel :

- On obtient la matrice P'_i de la sorte :

$$P'_i = \begin{bmatrix} \alpha' \\ \beta' \\ \gamma' \end{bmatrix} = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1 & \beta_2 & \beta_3 \\ \gamma_1 & \gamma_2 & \gamma_3 \end{bmatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

- Ainsi on peut calculer les nouvelles coordonnées de P_i :

$$x_i = \alpha'_i / \gamma'_i$$

$$y_i = \beta'_i / \gamma'_i$$

Cet algorithme donne le résultat suivant :

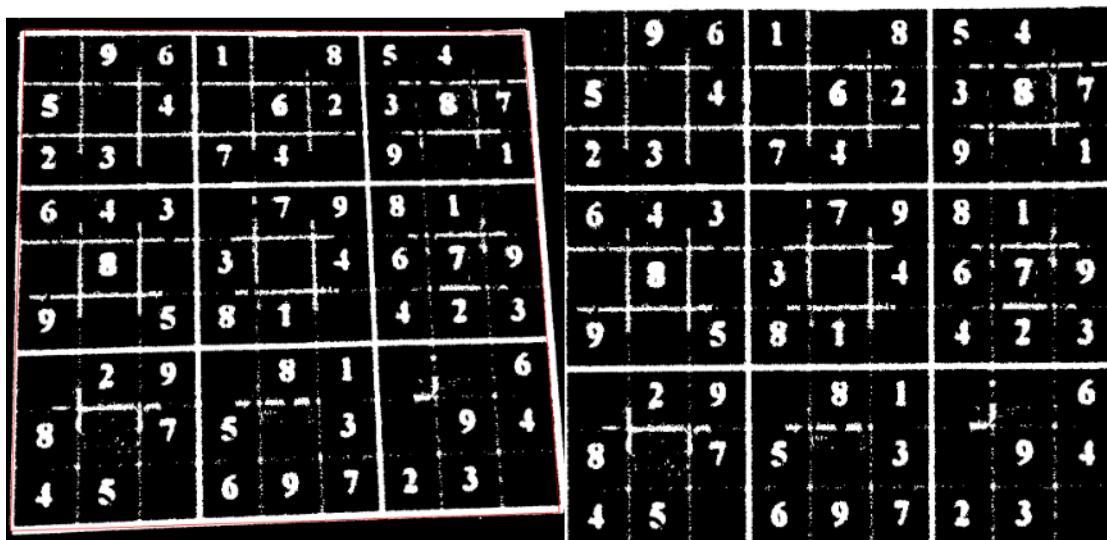


FIGURE 15 – Image avant et après recadrage et correction de la perspective

2.3.7 Découpage des cases

Pour pouvoir obtenir tous les chiffres présents sur l'image il faut d'abord découper l'image en 81 petites images qui seront par la suite envoyées au réseau de neurones. Pour se faire nous divisons par 9 la hauteur et la largeur de l'image puis pour chaque image de cette taille dans l'image de départ nous l'a stockons dans un tableau. Ainsi nous avons récupéré un tableau des 81 blocs désirés que nous allons pouvoir redimensionner en 28 pixels par 28 pixels pour le réseau de neurones.

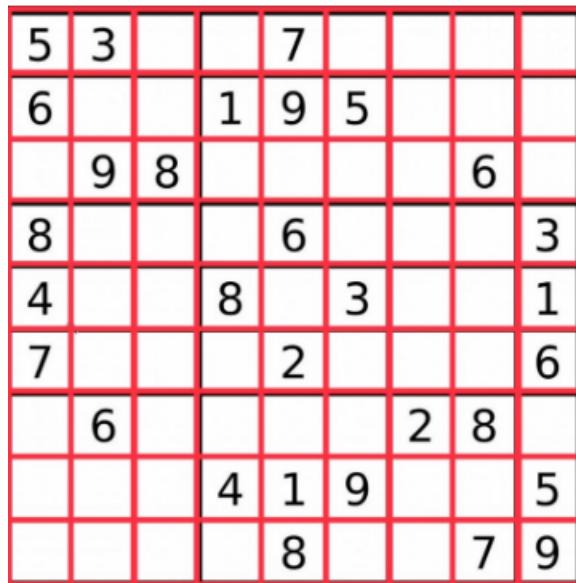


FIGURE 16 – Découpage de la grille en 81 blocs

2.3.8 Redimension des cases

Pour les besoins de notre réseau de neurones, l'image que nous faisons analyser par notre réseau de neurones doit être un cube de 28 pixels. Pour cela nous avons utilisé l'algorithme de l'interpolation bilinéaire.

L'interpolation bilinéaire est une fonction quadratique qui peut se mettre sous la forme :

$$f(x, y) = ax + by + cxy + d$$

La valeur $f(x, y)$ est la valeur interpolée au point de coordonnées (x, y) et a, b, c, d sont des constantes déterminées à partir des quatre voisins $(x_1, y_1), (x_1, y_2), (x_2, y_1), (x_2, y_2)$ du point (x, y) dont on cherche la valeur. Connaissant les valeurs en ces points, on

peut écrire un système de 4 équations à 4 inconnues :

$$\begin{cases} f(x_1, y_1) = ax_1 + by_1 + cx_1y_1 + d \\ f(x_2, y_1) = ax_2 + by_1 + cx_2y_1 + d \\ f(x_1, y_2) = ax_1 + by_2 + cx_1y_2 + d \\ f(x_2, y_2) = ax_2 + by_2 + cx_2y_2 + d \end{cases}$$

L'interpolation bilinéaire peut s'interpréter comme une succession de deux interpolations linéaires, une dans chaque direction.

Pour résumer, nous prenons la moyenne des pixels autour d'un pixel afin d'en refaire un. De cette manière on peut réduire le nombre de pixels pour arriver à 28 pixels, ce qui est parfait pour notre réseau de neurones.

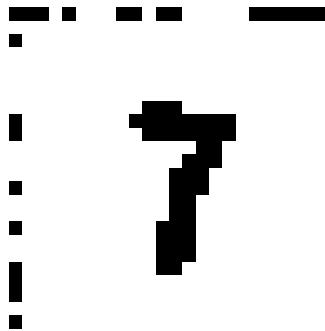


FIGURE 17 – un bloc redimensionné en 28x28 pixel

2.3.9 Mise au propre des images

Une fois les images redimensionnées, il faut les épurer pour que l'image ne contiennent plus que le chiffre. En effet, après le découpage et la redimension des cases, il reste souvent des éléments parasites comme vous pouvez le remarquer ci-dessous :

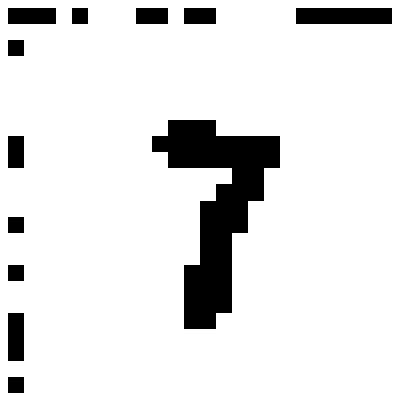


FIGURE 18 – Chiffre avec éléments parasites

Pour supprimer ces éléments indésirables nous utilisons deux méthodes.

- Pour chaque ligne, si le nombre de pixels noirs présents sur la ligne dépasse un certain seuil, alors on passe toute la ligne en pixels blanc.
- Ensuite, on applique le même raisonnement sur les colonnes.
- Cette première méthode nous permet de nous débarrasser des lignes qui délimitait chaque case du sudoku.
- Puis, pour le bruit restant, nous considérons tout les pixels noirs comme les sommets d'un graphe non-orienté. Les arrêtes du graphe existent si ces deux sommets sont deux pixels voisins. On parcours ensuite toutes les composantes connexes de l'image en comptant le degré de chaque graphe. Puis on garde la composante connexe ayant le degré le plus élevé (théoriquement toujours le chiffre), toutes les sommets des autres composantes sont remplacés par des pixels blancs.

A la fin de l'application de nos deux méthodes, on se retrouve avec seulement le chiffre dans la case :

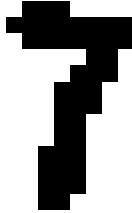


FIGURE 19 – Chiffre sans éléments parasites



Pour les cases qui ne contiennent pas de chiffres, on désire n'avoir aucun pixel noir dans l'image. Après l'application des méthodes présentées ci-dessus, on vérifie le degré du graphe que nous avons gardé : si le degré est inférieur à un certain seuil (donc que le graphe ne représente pas un chiffre), on supprime aussi ce graphe, ce qui nous donne une image toute blanche.

3 Réseau de neurones

3.1 Réseau XOR

Avant de réaliser le réseau de neurones final pour notre sudoku, il faut commencer par plus simple : un réseau de neurones qui reproduit le comportement d'une porte XOR.

Ainsi, ce réseau est composé de trois couches :

- Une couche d'entrée comprenant 2 neurones pour les valeurs qu'on donne au départ (0 ou 1)
- Une couche cachée comprenant 2 neurones, chacun reliés aux neurones de la couche d'entrée
- Une couche de sortie comprenant 1 neurone relié aux couches cachées, qui va renvoyer la décision finale du réseau (0 ou 1 pour le XOR)

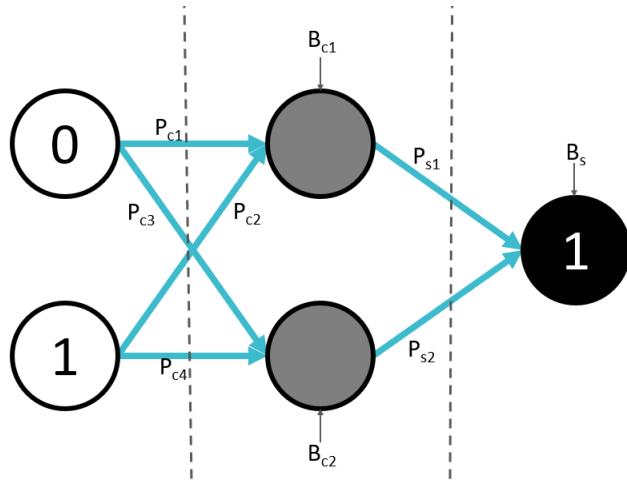


FIGURE 20 – Schéma simplifié des couches du réseau de neurone XOR

Cependant, pour chaque neurone de la couche cachée et le neurone de sortie, il y a des paramètres qui permettent d'influencer la décision afin de donner le bon résultat. On appelle ces paramètres, les poids et les biais. Cependant, pour trouver les bonnes valeurs, il faut pouvoir entraîner notre réseau.

Pour cela, on va utiliser deux principes qui fonctionnent ensemble :

● La propagation vers l'avant

Ce principe permet d'abord de partir des neurones d'entrée pour calculer un résultat en passant par tous les neurones. Pour cela, on applique à chaque neurone caché la formule suivante :

$$N_n = \text{sigmoide}(\text{Biais}_n + \sum_{i=0}^n \text{Poids}_i * \text{Input}_i), \text{ sigmoide}(x) = \frac{1}{1 + e^{-x}}$$

La fonction sigmoïde introduite ici permet d'obtenir un résultat appartenant à $[0;1]$. Ainsi, le résultat donné peut être interprété plus facilement, notamment pour le XOR dont les résultats sont 0 ou 1. On peut donc en déduire le résultat final selon s'il est proche de 1 ou de 0. De plus, la dérivée de cette fonction qui va nous servir dans la prochaine étape est simple, ce qui évite un ralentissement de notre calcul.

● La propagation vers l'arrière

Cette deuxième partie du réseau est la plus importante. Elle permet au réseau "d'apprendre". En effet, on va minimiser les erreurs de poids afin d'obtenir les résultats souhaités.

Pour re-calculer les poids entre la couche cachée et la couche de sortie, on utilise la formule suivante :

$$P_{sortie_i} = N_{cache_i} + \Delta_s * va$$

Avec $\Delta_s = (V_{obtenue} - V_{attendue}) * (\text{sigmoide})'(N_{sortie})$, et va la vitesse d'apprentissage.

Ensuite, pour re-calculer les poids au départ de la couche cachée, on utilise :

$$P_{cache_i} = Input_i * \Delta_c * va$$

Avec $\Delta_c = \sum \Delta_o * P_i * (\text{sigmoide})'(N_{cache_i})$, et va la vitesse d'apprentissage.

En même temps, il faut actualiser les biais :

$$B_{sortie_i} = \Delta_s * va$$

$$B_{cache_i} = \Delta_c * va$$

Ainsi, nos poids et biais sont actualisés et on peut recommencer le processus sur un certain nombre d'itérations qu'on choisit. Plus le nombre d'itérations sera grand, plus le réseau sera entraîné, donc aura de meilleurs résultats.

3.2 Sauvegarde et chargement des poids

Une fois que le réseaux de neurones apprend, il faut pouvoir **sauvegarder** son état pour pouvoir le réutiliser plus tard (et donc **récupérer** le même état) sans passer une nouvelle fois par une phase d'apprentissage.

Un dossier spécifique à cette partie est donc créé contenant quatre fichiers :

- - **WH** : qui contiendra les poids de la couche cachée. (Weight Hidden)
- - **BH** : qui contiendra les biais de la couche cachée. (Bias Hidden)
- - **WO** : qui contiendra les poids de la couche de sortie. (Weight Output)
- - **BO** : qui contiendra les biais de la couche de sortie. (Bias Output)

Ainsi, les poids et biais du réseaux sont toujours enregistrés avant de finir le programme.

Pour les récupérer, un argument **-load** (charger) peut-être utiliser afin d'initialiser les poids aux valeurs dans les fichiers plutôt qu'aux valeurs initialement aléatoires.

Cette capacité à récupérer les données nous permettra donc d'obtenir un résultat sans passer par un apprentissage (donc un calcul simple et rapide suffira). Cela nous permettra également d'entraîner notre réseaux de neurones à plusieurs reprises (ce qui sera nécessaire avec les images, car plusieurs types d'écriture sont possibles).

3.3 Réseau de neurones final

Le réseau de neurones XOR est donc réalisé et fonctionnel. Ainsi, cette partie nous a permis de réellement comprendre de quoi était constitué un réseau de neurones, de comprendre un fonctionnement particulier, afin de construire le réseau de neurones final.

Pour ce dernier, on va garder le même principe que le XOR, pour la structure globale. C'est-à-dire qu'il y a une couche d'entrée, une couche cachée et une couche de sortie. Cependant, c'est le nombre de neurones (ou d'entrées) qui va être modifié par rapport à avant :

- La couche d'entrée contient 784 entrées. En effet, le réseau prend en paramètre une liste des images des cases du sudoku, découpées précédemment, que l'on transforme en liste de tableaux de pixels. Les images faisant une taille de 28x28px (784px) cela nous ramène à nos donc 784 entrées. De plus, grâce au pré-traitement, les pixels étant seulement noirs ou blancs, cela revient à avoir en entrée seulement des 0 (noir) ou des 1 (blanc).
- La couche cachée ne prend maintenant plus 2 neurones cachés, mais 81. Ce nombre a été décidé à la suite de plusieurs essais de notre réseau. En effet, il semble être le meilleur pour un rapport temps/efficacité intéressant.
- La couche de sortie contient maintenant 10 neurones de sortie. Ces neurones correspondent aux 10 possibilités de résultats : 0 (case vide), 1, 2, 3, 4, 5, 6, 7, 8 ou 9. Il a été décidé de réaliser notre sortie de cette façon car la fonction que l'on utilise (sigmoïde) retourne des valeurs comprises entre 0 et 1, nous permettant de les interpréter comme des pourcentages. Ainsi, le réseau nous retourne une valeur dans chacun des neurones de sortie mais seulement 1 neurone renverra une valeur très proche de 1. Le numéro de ce neurone nous donne le chiffre reconnu.

Visuellement, cela nous donne :

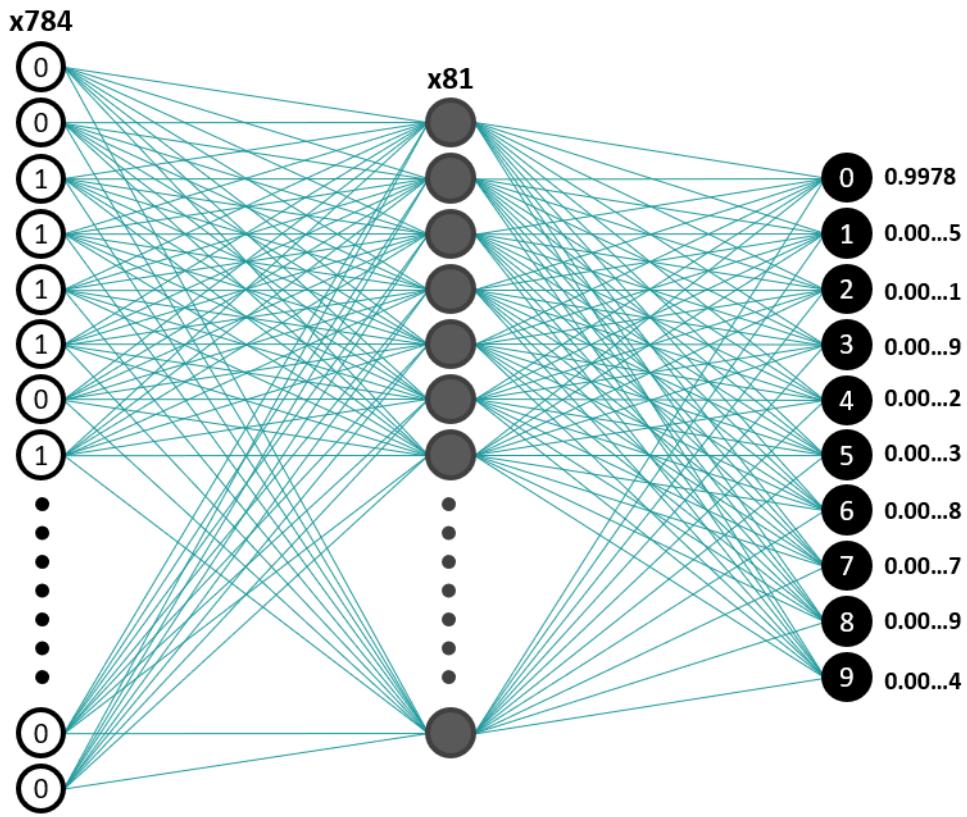


FIGURE 21 – Schéma simplifié des couches du réseau de neurone XOR

Grâce à cette structure, notre réseau de neurones peut apprendre efficacement. Pour ce faire, on ré-utilise la même méthode que pour le réseau XOR, décrite précédemment.

4 Résolution

4.1 Algorithme de résolution

Pour résoudre le sudoku, nous utilisons l'algorithme naïf, qui testera en quelques sorte toutes les valeurs possibles jusqu'à obtenir les bonnes valeurs pour le sudoku. L'algorithme est le suivant :

Pour chaque case du sudoku :

- On met le chiffre 1 dans la case
- Si le sudoku est toujours valide, on passe la case suivante
- Si le tableau n'est pas valide, on essaye avec 2, puis 3 si c'est toujours le cas, etc.

- Si aucune valeur ne correspond, on retourne à la case précédente, où on testera les autres valeurs possibles



Le sudoku étant une matrice 9x9, l'exécution de la résolution est quasi-instantanée. Il n'est donc pas nécessaire pour cet algorithme d'être trop optimisé.

5 Application

5.1 Interface Utilisateur

Le projet étant de former une application, nous avons dû en créer une. Il nous a fallu, pour cela, nous intéresser au monde de l'interface utilisateur qui est le côté visible de l'application.

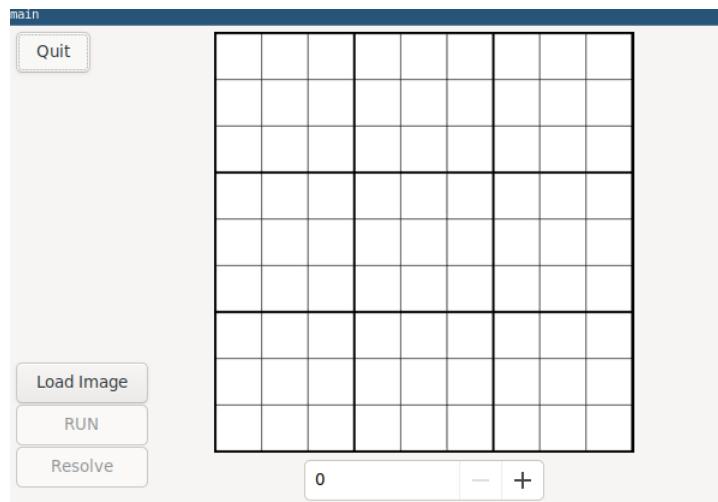


FIGURE 22 – L'application à son ouverture

Une interface utilisateur se doit être simple d'utilisation et de prise en main mais également lisible et compréhensible. C'est pour cela que nous n'avons qu'un nombre limité de boutons présent.

- le bouton **QUIT** (cf. Quitter) qui nous permet de quitter l'application.
- le bouton **LOAD** (cf. Charger) pour pouvoir choisir un sudoku. Cela ouvrira une nouvelle fenêtre permettant d'effectuer cette action.
- le bouton **RUN** (cf. Lancement) qui lancera le traitement de l'image ainsi que la reconnaissance de chiffres.
- le bouton **RESOLVE** (cf. Résoudre) pour résoudre le sudoku et afficher le résultat.



Certaines choses sont faites pour éviter des erreurs d'exécutions. Par exemple : si le fichier choisi n'est pas une image. Aucune action ne sera effectuée. Empêchant ainsi l'application de planter lors de l'analyse de l'image.

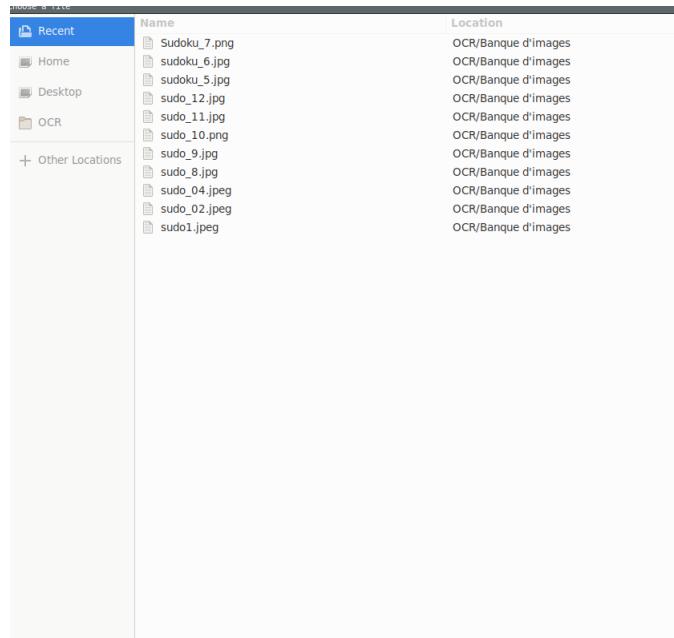


FIGURE 23 – Choix du sudoku à résoudre

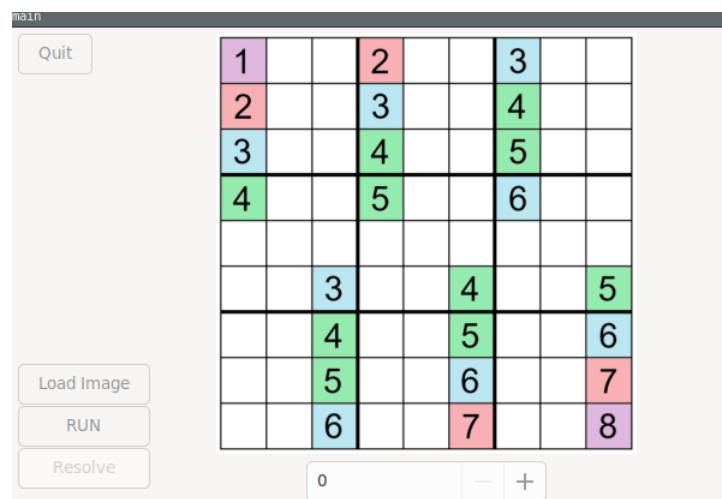


FIGURE 24 – Le sudoku est affiché à l'écran

Tous les calculs se font "en arrière plan", l'utilisateur ne les voit pas. Lors du traitement de l'image, les différents filtres sont appliqués un à un cependant, seul

le résultat final est visible à l'écran. un fichier texte servant à la résolution est également créé. C'est dans ce fichier que les chiffres reconnus seront placés et que le sudoku final sera stocké.

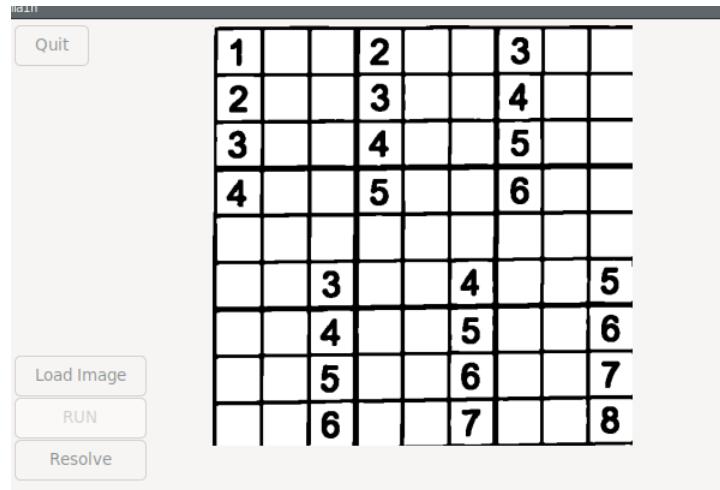


FIGURE 25 – Le sudoku traité

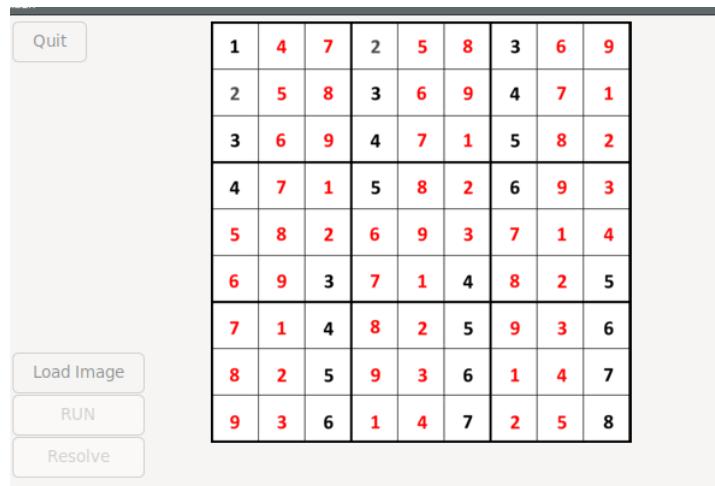


FIGURE 26 – Le sudoku résolu

A noter qu'il existe également **une zone de saisie** située en bas de l'écran dans laquelle l'utilisateur peut entrer un chiffre pour tourner l'image d'un certain nombre de degrés. Il est également possible d'appuyer sur les boutons **PLUS** (resp .**MOINS**) pour augmenter (resp. diminuer) cette rotation degrés par degrés. Une fois la rotation désirée choisie, il suffit de choisir **Save Rotation** (cf. Sauvergarder rotation) pour pouvoir appliquer la résolution sur la nouvelle image !

6 Planning

(Etat actuel) Retard● Terminée●

(Prévisions) Avancée○ Terminée●

Taches	1ère Soutenance	2ème Soutenance
Image		
Chargement d'une image	●	●
Suppression des couleurs	●	●
Prétraitement	●	●
Detection de la grille	●	●
Détection des cases	●	●
Réseau de neurones		
Réseau de neurone XOR	●	●
Sauvegarde et chargement des poids	●	●
Récupération des chiffres dans les cases		●
Reconnaissance des caractères		●
Résolution		
Algorithme de résolution	●	●
Reconstruction de la grille		●
Affichage de la grille résolue		●
Sauvegarde de la grille résolue		●
Interface graphique		●

7 Conclusion

7.1 Ressenti

7.1.1 Owen

Le projet de l'OCR, complètement différent du projet du semestre 2, a été tout aussi enrichissant que ce dernier. En effet, il m'a permis d'acquérir des connaissances dont je n'avais pas la moindre idée de l'existence. Ainsi, de nature curieuse, cela m'a beaucoup intéressé, surtout la partie sur laquelle j'ai travaillé principalement : le réseau de neurone. En effet, l'intelligence artificielle est un sujet qui me passionne, ça a donc été un moyen d'approcher en partie ce côté de l'informatique.

La différence avec le projet jeu vidéo à également la dominance de l'utilisation des mathématiques. Cependant, malgré mes difficultés dans cette matière, j'ai réussi à comprendre ce que je faisais et ce que je devais faire. Ceci a notamment été rendu possible grâce à l'aide de mes coéquipiers. En effet, l'esprit d'entre-aide était présent du début à la fin et a été un des moteurs pour que le projet final aboutisse.

7.1.2 Axel

Un projet captivant et bien différent du projet jeu-vidéo de première année mais tout aussi concret. En effet, la reconnaissance de caractère par intelligence artificielle m'intéressait beaucoup. C'était l'occasion de voir comment ce processus pouvait fonctionner.

La construction du réseaux de neurones s'est effectuée par, dans un premier temps, beaucoup de recherches sur le sujet. Les fonctions mathématiques d'activation ou encore le fonctionnement et l'impact de chaque paramètres sur l'apprentissage et la restitution de résultat.

Ce sont des connaissances que nous avons réussi à appliquer pour mener à bien ce projet. De plus, le travail constant de chacun ainsi qu'une bonne répartition des tâches nous a permis de bien avancer.

Et comme pour chaque projet, le voir prendre vie, fonctionnel à la fin est toujours source d'une grande satisfaction pour le groupe.

7.1.3 Jules

La création de cet OCR a été pour moi un grand apprentissage. Effectivement, je suis tout d'abord fier d'avoir participer au bon déroulement de ce projet.

L'ambiance de notre groupe était déterminante et propice au travail. En effet, l'entraide était efficace et permettait de régler plus facilement nos problèmes. L'or-

ganisation de notre groupe était bonne et la répartition des taches équilibrail bien le temps de travail de chacun.

Durant ce projet, j'ai aussi développé des connaissances personnelles. Outre la programmation, j'ai acquis de nouvelles compétences : comme le traitement de l'image par exemple. Je suis vraiment content d'avoir découvert et beaucoup appris dans ce domaine qui ajoute une corde à mon arc du savoir-faire.

Ce travail m'a aussi appris à confectionner un cahier des charges et rédiger des rapports, ce qui me servira plus tard.

7.1.4 Mathieu

Ce projet m'a beaucoup apporté que ça soit en terme d'organisation ou d'esprit d'équipe. Il m'a obligé à avoir une certaine régularité dans mon travail avec des objectifs à remplir en un temps donné.

Au début du projet, je me demandais comment allons-nous réussir à concrétiser le projet jusqu'au bout. Puis au fur et à mesure, à force de s'informer et d'effectuer de nombreuses recherches qui ont demandées beaucoup d'énergie, je me suis rendu compte qu'avec du travail et de la passion on pouvait accomplir beaucoup de choses qui nous paraissent impossibles. Ce projet m'a permis d'acquérir beaucoup de connaissance et un autre point de vue sur les problèmes à résoudre.

Cependant, au cours de cette réalisation, chaque avancée entraînait de nouveaux problèmes qui nous ralentissaient dans notre progression et nous prenait beaucoup de temps. À ces moments-là, la motivation n'était plus la même, mais leur résolution était des moments de joie particuliers.

Avant la première soutenance, je trouvais qu'on avait toujours quelque chose d'intéressant à ajouter et que chacun arrivait à apporter sa pierre à l'édifice. Depuis la première soutenance, travailler demande de plus en plus de motivation et les avancées se remarquent de moins en moins.

Ces dernières semaines ont été dures en termes de travail, en effet le projet demandais énormément de temps et la date de fin se rapprochais à grand pas. De mon côté, arriver à finir l'OCR comme on l'espérait me semblait compliqué, mais finalement je suis fier de ce qu'on a produit malgré quelques points à améliorer.

7.2 Conclusion

Enfin, c'est avec fierté et honneur que nous annonçons que nous avons accompli notre objectif final malgré les difficultés rencontrées durant ce périple. L'OCR appliquée aux sudokus classique est donc terminé et fonctionne selon le cahier des charges défini en amont.