

EPITA



OCR Solveur de sudoku

Rapport de projet

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

10 Novembre 2022

Table des matières

1	Introduction	2
2	Image	2
2.1	Chargement d'image	2
2.1.1	Chargement de l'image	2
2.2	Prétraitement	2
2.2.1	Filtre Grayscale	3
2.2.2	Filtre de contraste	4
2.2.3	Normalisation des éclairages	5
2.2.4	Filtre médian	6
2.2.5	Filtre moyen	8
2.2.6	Filtre de seuil adaptatif	9
2.2.7	Filtre de lissage	10
2.3	Détection de la grille et des cases	11
2.3.1	Filtre de Sobel	11
2.3.2	Détection des lignes	12
2.3.3	Simplification des lignes	14
2.3.4	Rotation de l'image	14
3	Réseau de neurones	15
3.1	Réseau XOR	15
3.2	Sauvegarde et chargement des poids	17
4	Résolution	18
4.1	Algorithme de résolution	18
5	Planning	18
6	Conclusion	19

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 2 mois que notre projet est lancé et grâce à un investissement régulier de l'ensemble des membres de l'équipe, celui-ci avance comme nous le souhaitons.

2 Image

2.1 Chargement d'image

2.1.1 Chargement de l'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 :

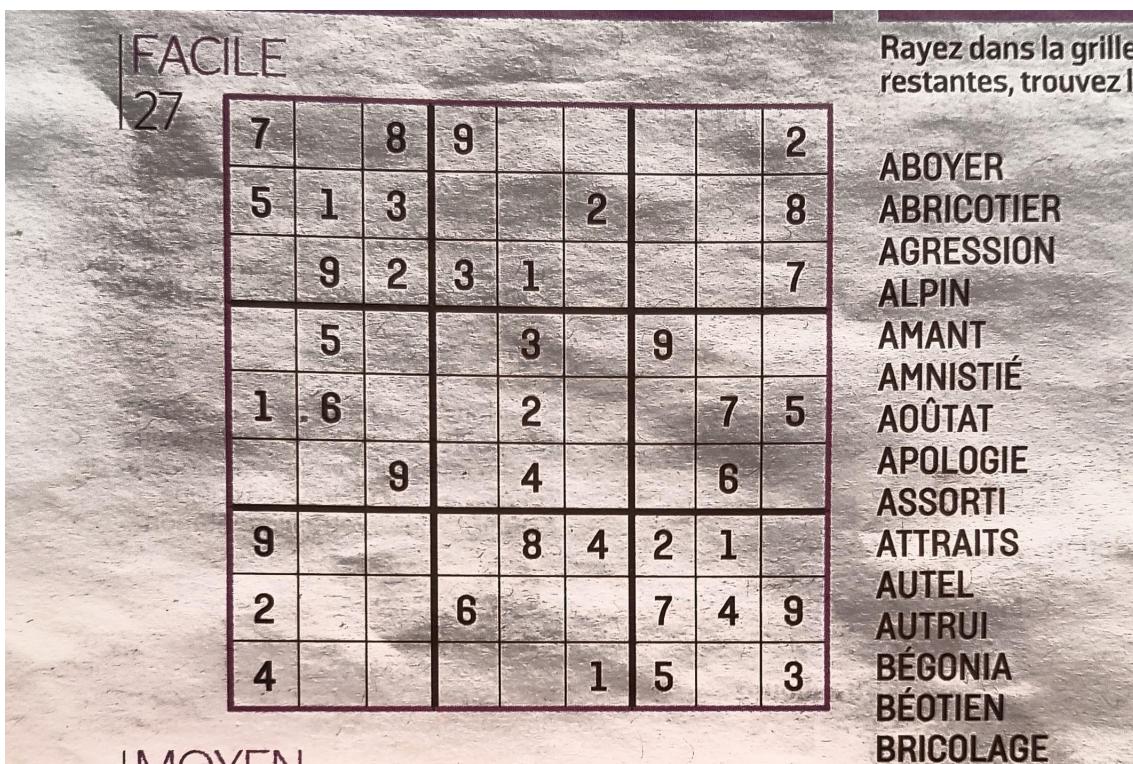


FIGURE 1 – Image support

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 :

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

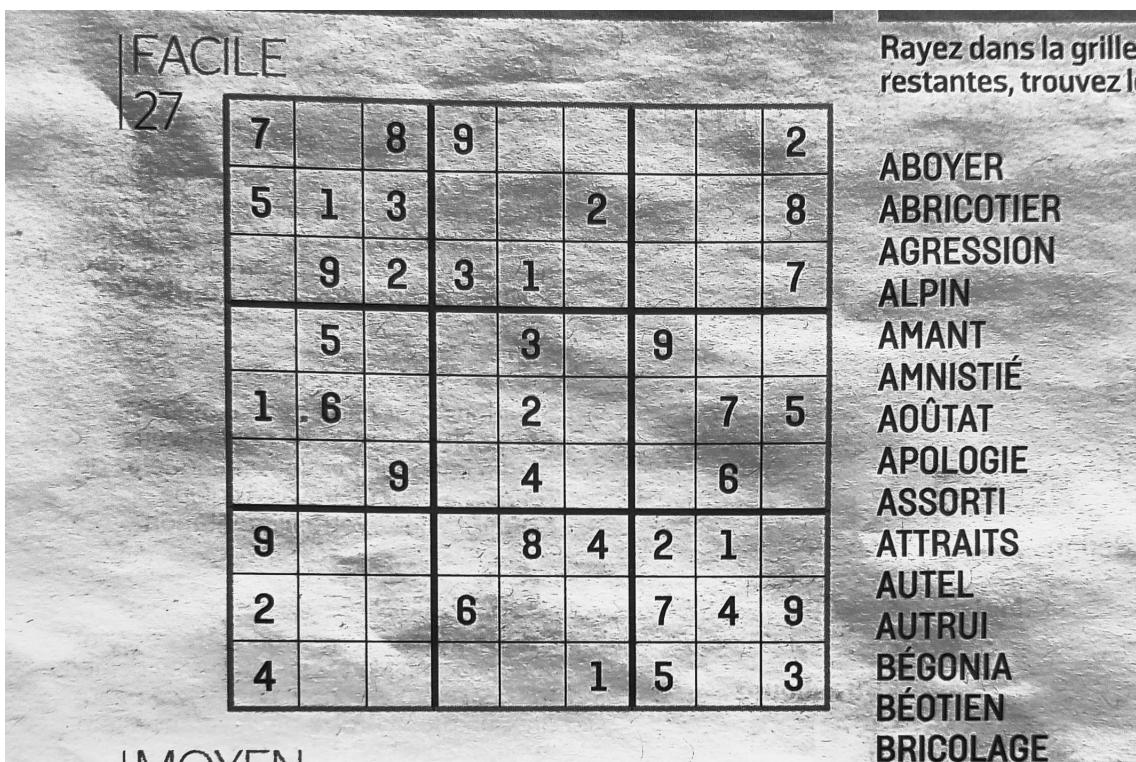


FIGURE 2 – Image après l’application du filtre grayscale

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 :

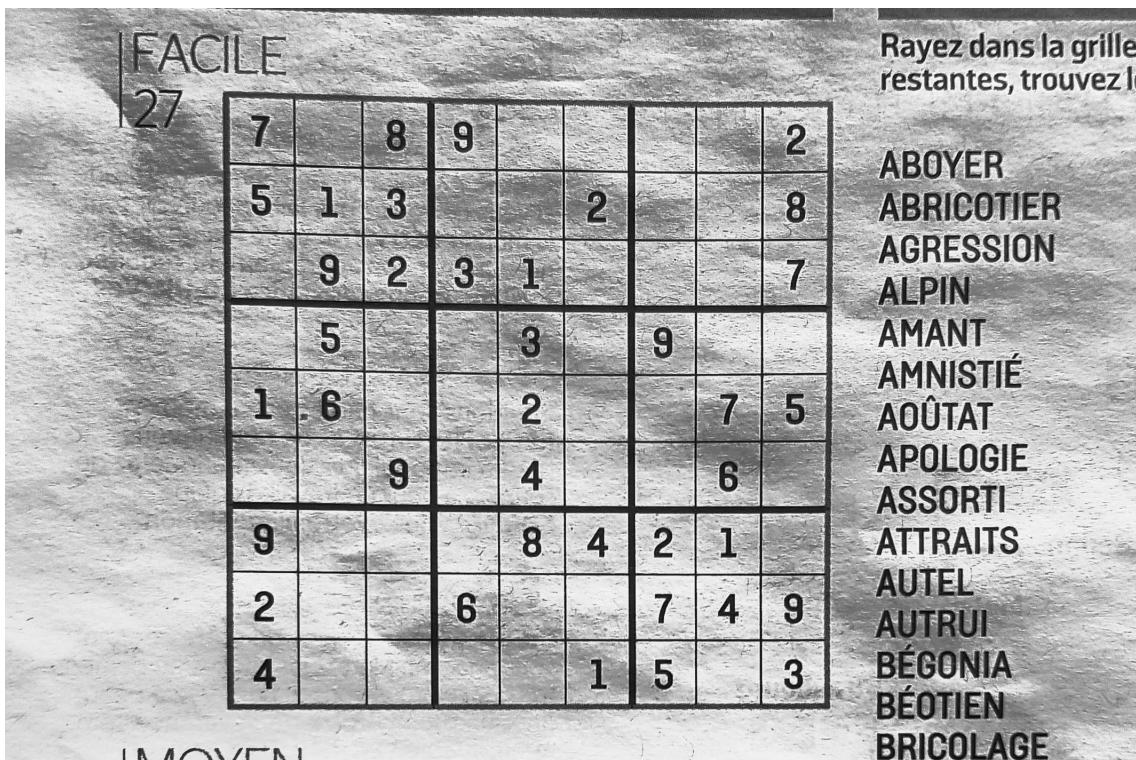


FIGURE 3 – Image après l’application du filtre de contraste

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 :

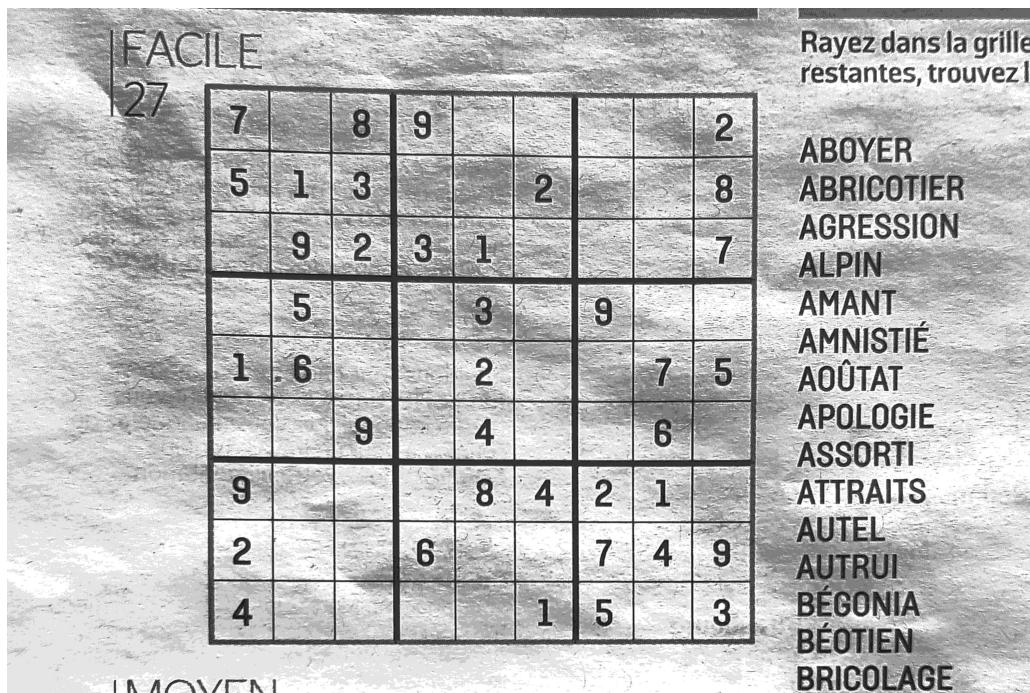


FIGURE 4 – Image après la normalisation des éclairages

2.2.4 Filtre médian

Le filtre médian permet d'éliminer les valeurs aberrantes de l'image. Ce filtre efface donc les bruits impulsifs. 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 :

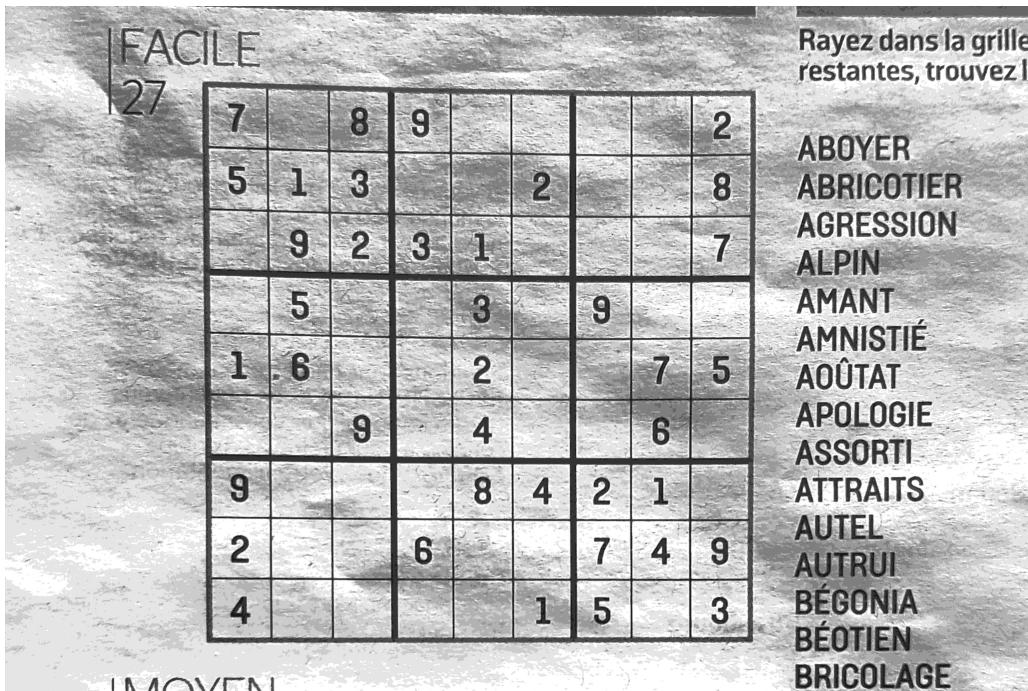


FIGURE 7 – Image après l’application du filtre médian

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 :

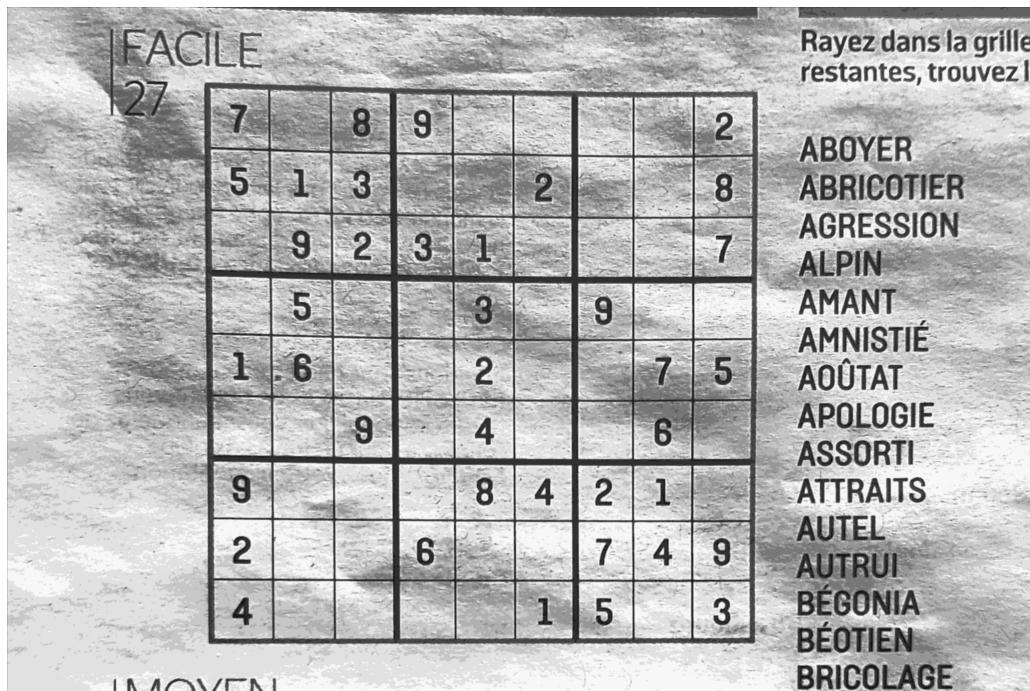


FIGURE 8 – Filtre Moyen

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 :

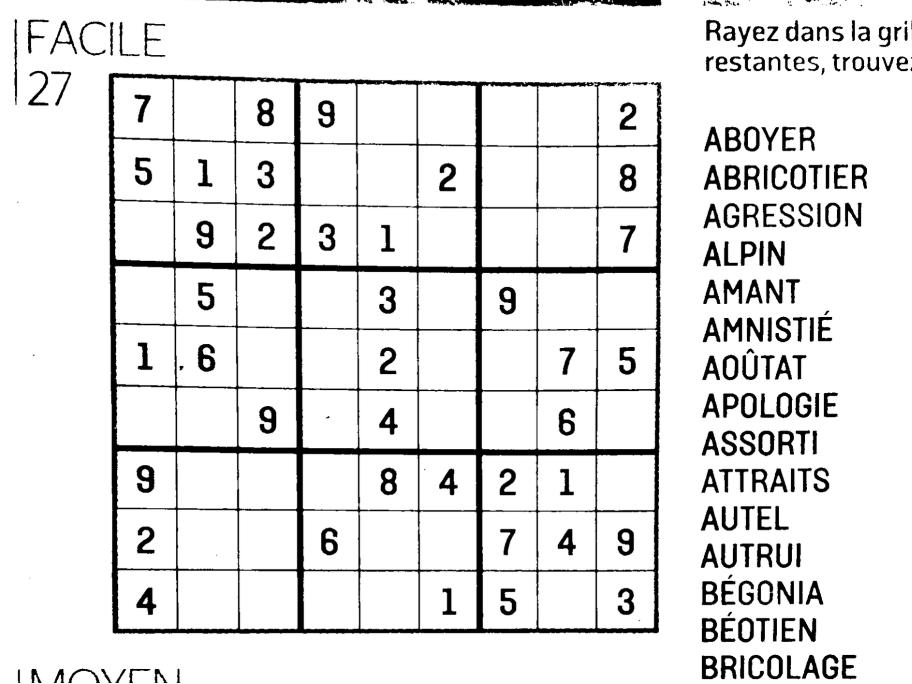


FIGURE 9 – Filtre de seuil adaptatif

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 :

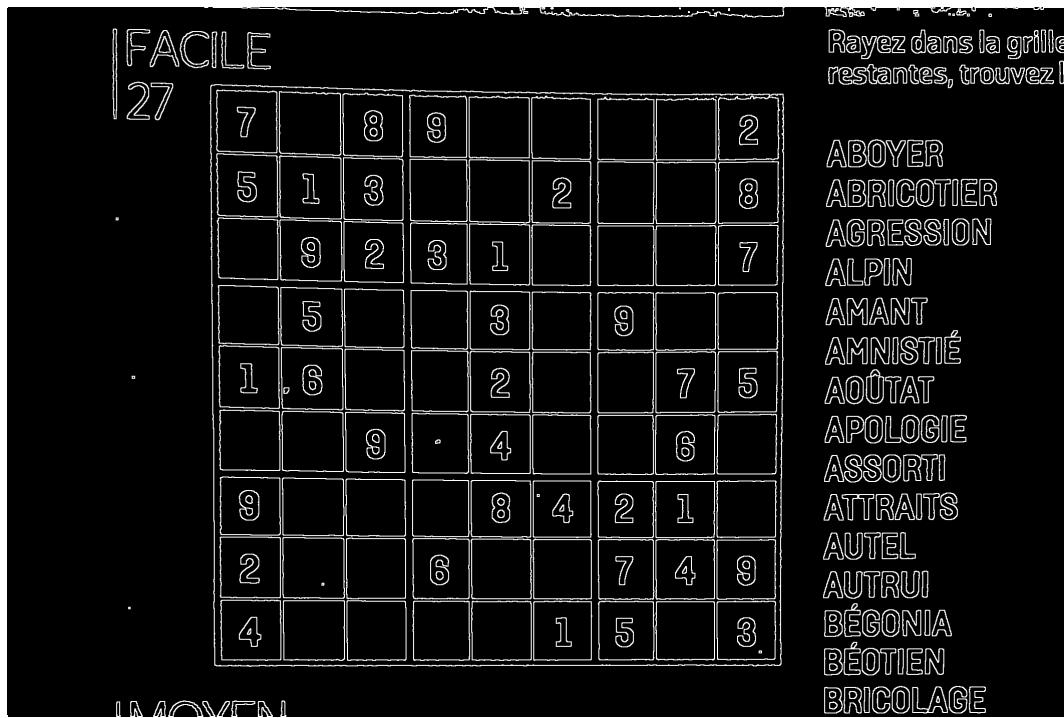


FIGURE 11 – Image après application du filtre de Sobel

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 :

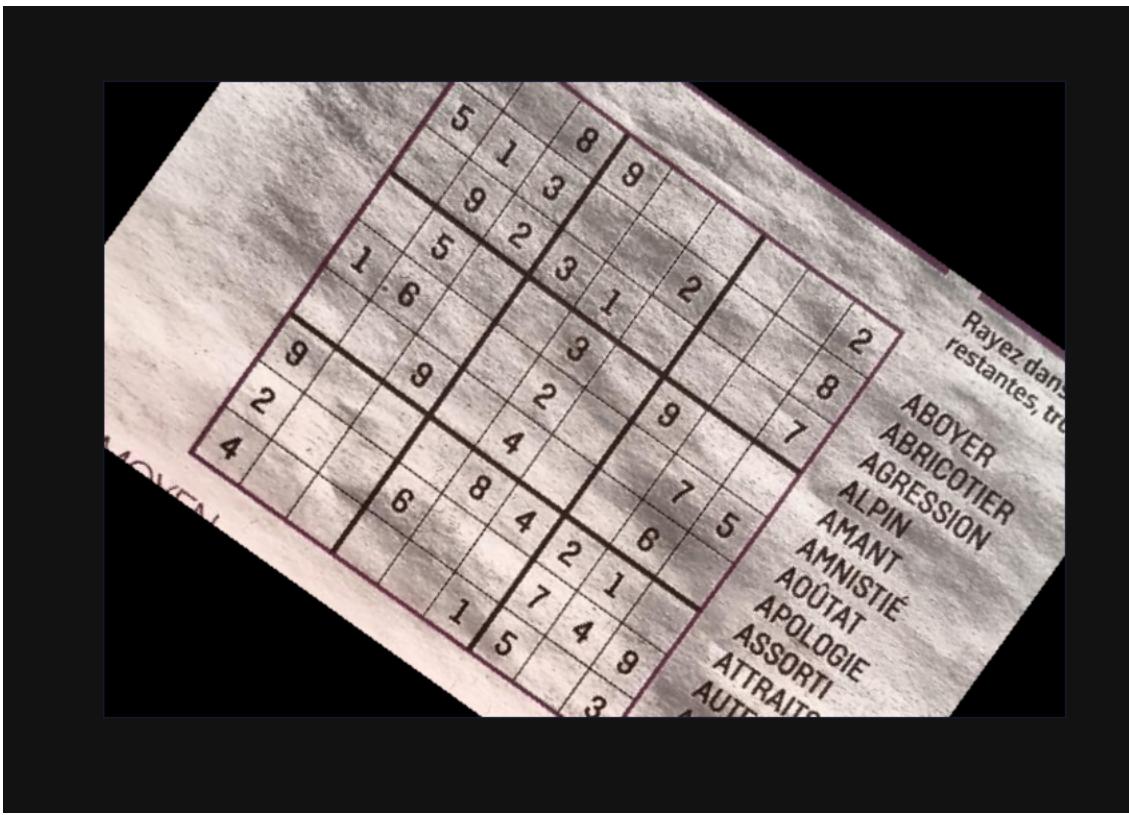


FIGURE 13 – Rotation de 30° sur l'image₁

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)

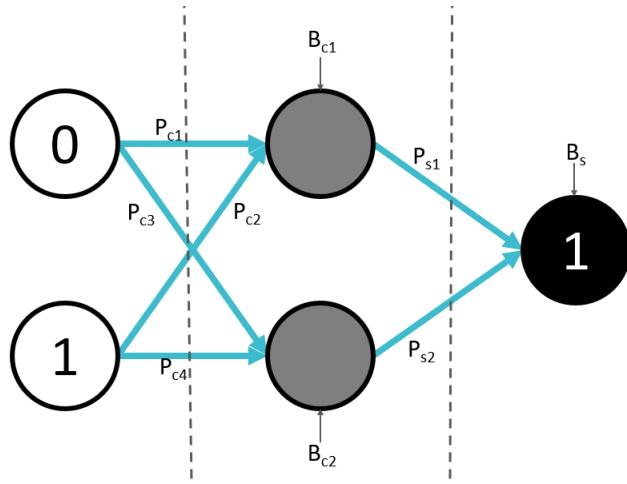


FIGURE 14 – Schéma simplifié des couches du réseau de neurone XOR

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).

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 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		●

6 Conclusion

Ce premier rendu du projet était une occasion pour nous quatre d'apprendre de nouvelles choses !

Tout d'abord la manipulation d'image qui se révèle être une partie importante et nécessaire au projet.

Puis le système de réseaux de neurones permettant à de simples fonctions mathématiques de donner des résultats cohérents avec un contexte (une fois entraîné).

Une organisation ainsi qu'une bonne répartition des tâches étaient nécessaires afin de réaliser un début de projet correct.