



EPITA

PROJET OCR S3

Rapport de soutenance 1



Samuel LAMBERT
Mathéo CRESPEL
Augustin CLAUDE
Lucas BESNARD

Septembre 2022 - Décembre 2022

Table des matières

1	Introduction	2
2	Les membres du groupe	2
3	La répartition des tâches	2
4	L'état d'avancement du projet	3
5	Aspects techniques	4
5.1	Chargement de l'image	4
5.2	Prétraitement de l'image	5
5.2.1	Rotation manuelle	5
5.2.2	Niveau de gris	5
5.2.3	Normalisation	5
5.2.4	Réduction de bruits	6
5.2.5	Filtre de Canny	7
5.3	Détection de la grille	10
5.3.1	Détection des lignes	10
5.3.2	Détection des rectangles	12
5.3.3	Détection de la grille principale	13
5.3.4	Correction de la perspective et rotation automatique	13
5.3.5	Découpage en sous-image	15
5.4	Réseau de neurones	15
5.4.1	XOR	15
5.4.2	Sauvegarde des poids	18
5.5	Solveur du sudoku	18
5.6	Interface utilisateur	18
5.6.1	Bibliothèque GTK	19
5.6.2	Glade	19
5.6.3	Description des composants de l'UI	19
6	Avenir du projet	21
7	Conclusion	21

1 Introduction

Le groupe « Grid'okuCr » est fier de vous présenter ce premier rapport de projet. Ce dernier détaille le découpage des tâches, la répartition de celles-ci au sein du groupe, l'avancement du projet, les problèmes rencontrés, une description détaillée des aspects techniques de notre projet et enfin les prévisions pour la dernière soutenance.

2 Les membres du groupe

Notre groupe est composé de 4 personnes : Samuel LAMBERT, Lucas BESNARD, Mathéo CRESPEL et Augustin CLAUDE.

3 La répartition des tâches

Nous avons décidé de découper le projet en 4 grandes sous-tâches :

- Prétraitement des images
- Détection de la grille
- Réseau de neurones
- Interface Utilisateur / solver / sauvegarde du résultat.

Le contenu de chacune de ces grandes tâches est spécifié dans le tableau 1 ci-dessous. Chacun des membres du groupe s'est vu attribuer une de ces 4 grandes tâches comme travail à faire pour le projet tout en restant disponible si jamais un autre membre à besoin d'aide sur sa tâche attribuée.

Membres	Tâches
Lucas	<u>Prétraitement :</u> Chargement de l'image Niveau de gris Normalisation Réduction de bruit Filtre de Canny Redressement automatique de l'image
Augustin	<u>Détection de la grille :</u> Détection des lignes Détection de la grille en entier Détection des cases Découpage de la grille
Samuel	<u>Réseau de neurones :</u> Réseau XOR Récupération des chiffres dans les cases Reconnaissance de caractères Reconstruction de la grille (sous format texte compréhensible par le solver)
Mathéo	<u>UI / Solver / Sauvegarde du résultat :</u> Interface utilisateur simple répondant aux critères Solver de sudoku rapide Inscription des caractères dans les cases vides Reconstruction image avec sudoku résolu

TABLE 1 – Répartition des tâches

4 L'état d'avancement du projet

Pour cette première soutenance, nous sommes capables de :

1. Charger une image et appliquer un prétraitement sur cette dernière, la rendant

« lisible » pour notre réseau de neurones (partie 5.2). Ce prétraitement consiste à :

- Transformer l'image en niveau de gris
 - Normaliser l'image
 - Appliquer un filtre flou
 - Redresser l'image manuellement
 - Appliquer un filtre de Canny
2. Détecter les lignes ainsi que les carrés présents dans l'image. (partie 5.3)
 3. À partir des 4 angles de la grille, corriger la perspective de l'image pour obtenir une image contenant uniquement la grille.
 4. Extraire les 81 sous images contenant chacune les chiffres du sudoku.
 5. D'obtenir un réseau de neurones capable d'apprendre la fonction OU EXCLUSIF. (partie 5.4)
 6. Résoudre une grille de sudoku en utilisant la technique du « back tracking ». (partie 5.5)
 7. Disposer d'une interface graphique permettant de : charger une image à résoudre, la redresser manuellement (par rapport à un angle donné) et sauvegarder le résultat final dans un format d'image standard. L'UI comporte aussi un bouton permettant de lancer le programme principal lorsque celui-ci sera terminé.

5 Aspects techniques

5.1 Chargement de l'image

Pour commencer l'OCR, nous devons être capable de charger une image sous forme de matrice de pixels accessible dans notre code C. Pour se faire, nous utilisons la librairie SDL2. Elle est très complète et permet d'accéder très facilement aux dimensions (largeur et hauteur) de l'image ainsi qu'à une matrice de pixels sous forme RGB. En effet, la valeur de chaque pixel se décompose en trois valeurs entre 0 et 255 : l'intensité de rouge, de vert et de bleu. Pour faciliter la manipulation des données de l'image dans la suite du programme, nous avons décidé de créer deux structures C.

```
typedef struct Pixel
{
    unsigned char r, g, b;
} Pixel;
```

FIGURE 1 – Structure d'un pixel

```
typedef struct Image
{
    unsigned int width;
    unsigned int height;
    struct Pixel **matrix;
    char *path;
} Image;
```

FIGURE 2 – Structure d'une image

Ainsi, nous utilisons la librairie SDL2 uniquement lors du chargement et de l'enregistrement des images. En dehors de ces deux étapes, nous utilisons nos structures.

5.2 Prétraitement de l'image

5.2.1 Rotation manuelle

L'utilisateur a la possibilité de tourner l'image manuellement afin de redresser une photo prise de travers. Après avoir fait une copie de l'image originale, la nouvelle valeur d'un pixel correspond à la valeur du pixel de l'image originale à la position x_1, y_1 .

$$x_1 = (x - x_0) * \cos(\theta) - (y - y_0) * \sin(\theta) + x_0$$

$$y_1 = (x - x_0) * \sin(\theta) + (y - y_0) * \cos(\theta) + y_0$$

x_0 et y_0 : coordonnées du centre de l'image

θ : angle de rotation en radians

5.2.2 Niveau de gris

Tout d'abord, nous transformons l'image en niveau de gris. Ainsi, la formule suivante est appliquée sur l'ensemble des pixels p :

$$p = p_r * 0.3 + p_g * 0.59 + p_b * 0.11$$

5.2.3 Normalisation

La variété d'images pouvant être traitées doit être la plus grande possible. Dans cette phase de normalisation, nous allons modifier la gamme des valeurs d'intensité des pixels pour qu'elle soit maximale : entre 0 et 255. Ainsi, la formule suivante est appliquée sur l'ensemble des pixels p :

$$p = (p - \min) * (255 / (\max - \min))$$

min : intensité minimale de l'image

max : intensité maximale de l'image

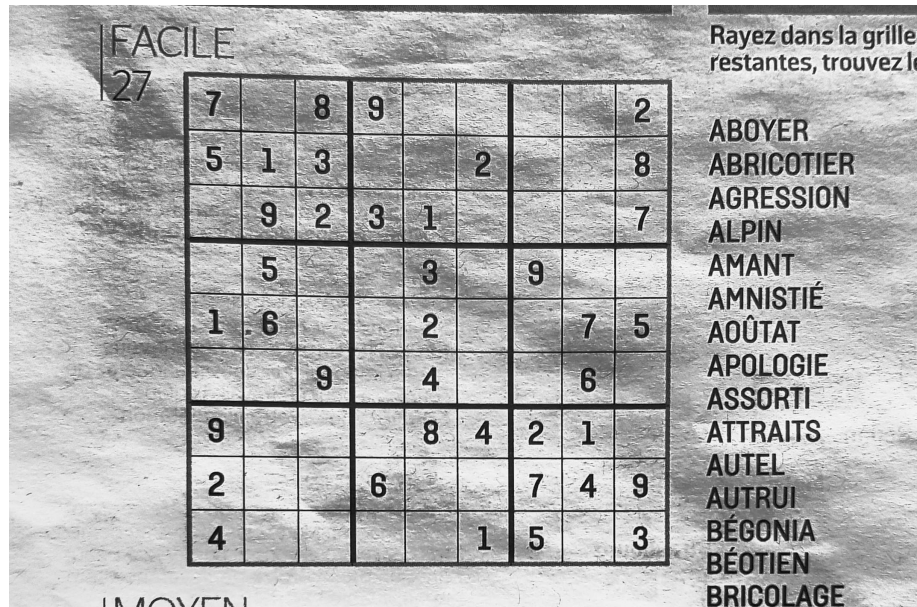


FIGURE 3 – Mise en niveau de gris et normalisation

5.2.4 Réduction de bruits

Toutes les images contiennent du bruit à différents niveaux. On appelle « bruit » de l'image la fluctuation indésirable de la couleur ou de la luminance qui obscurcit les détails de celle-ci et en dégrade la qualité. Celui-ci peut apparaître à cause de la scène que vous photographiez ou de votre capteur photo.

Nous utilisons successivement deux méthodes afin de réduire celui-ci : un flou gaussien puis deux opérations morphologiques (dilatation et érosion). Ces méthodes sont appliquées en fonction de la taille de l'image d'entrée.

Tout d'abord, nous appliquons le flou gaussien grâce à la convolution. Pour ce faire, nous avons besoin d'une matrice de convolution, celle-ci se calcule grâce à la fonction gaussienne. Afin d'éviter son calcul à chaque traitement d'image, nous avons décidé d'utiliser une matrice de convolution de 3*3 écrite "en dur" que nous appliquons une ou plusieurs fois en fonction de la taille de l'image. Ainsi, pour chaque pixel de l'image, il faut récupérer ses 8 voisins (s'il s'agit d'un pixel situé sur le bord de l'image, nous remplaçons ses voisins inexistants par des pixels noirs). Les 8 voisins et le pixel actuel forment une matrice de 3*3 que nous pouvons multiplier par la matrice de convolution. Attention, ce n'est pas un produit de matrice mais bien un produit de matrice de

convolution (lui-même lié à une forme de convolution mathématique) qui est utilisée dont voici la formule générale :

$$\begin{bmatrix} x_{11} & x_{12} & \cdot & \cdot & x_{1n} \\ x_{21} & x_{22} & \cdot & \cdot & x_{2n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ x_{m1} & x_{m2} & \cdot & \cdot & x_{mn} \end{bmatrix} * \begin{bmatrix} y_{11} & y_{12} & \cdot & \cdot & y_{1n} \\ y_{21} & y_{22} & \cdot & \cdot & y_{2n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ y_{m1} & y_{m2} & \cdot & \cdot & y_{mn} \end{bmatrix} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} x_{m-i,n-j} y_{1+i,1+j}$$

On obtient donc le calcul suivant pour chacun des pixels :

$$p_{x,y} = \begin{bmatrix} p_{x-1,y-1} & p_{x,y-1} & p_{x-1,y+1} \\ p_{x,y-1} & p_{x,y} & p_{x,y+1} \\ p_{x-1,y+1} & p_{x,y+1} & p_{x+1,y+1} \end{bmatrix} * \begin{bmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{bmatrix}$$

Après avoir appliqué ce flou, on utilise deux opérations morphologiques : la dilatation et l'érosion. Ces deux processus permettent respectivement d'agrandir et de réduire les zones les plus claires. Cependant, en utilisant les deux opérations à la suite, cela permet de réduire le bruit de l'image car les pixels et leurs voisins auront des intensités proches, évitant ainsi des pics ou des creux d'intensité. Voici l'algorithme utilisé pour la dilatation, celui de l'érosion étant identique à l'exception du fait que l'on garde le maximum des pixels voisins.

Pour tous les pixels de l'image :

- On récupère les x pixels voisins (x dépend de la taille de l'image)
- La valeur du pixel devient le minimum de ces pixels voisins

5.2.5 Filtre de Canny

La dernière étape de notre prétraitement de l'image est l'application d'une partie filtre de Canny. Néanmoins, l'utilisation de celui-ci pourra être amené à changer car il produit de nombreuses doubles lignes ce qui perturbent la détection de la grille.

Le filtre de Canny est utilisé pour la détection de contours (dans notre cas, les lignes de la grille et les chiffres). Les 4 étapes du filtre sont les suivantes : flou, filtre de Sobel, suppression des non-maxima et seuillage des contours.

Le flou et plus généralement la suppression du bruit dans l'image est effectué précédemment comme expliqué dans la section 5.2.4.

Nous appliquons ensuite le filtre de Sobel qui utilise la convolution dont le principe est expliqué dans la section 5.2.4. Le filtre calcule le gradient de l'intensité de chaque pixel. Le gradient représente la direction des contours de l'image. Afin de le calculer,

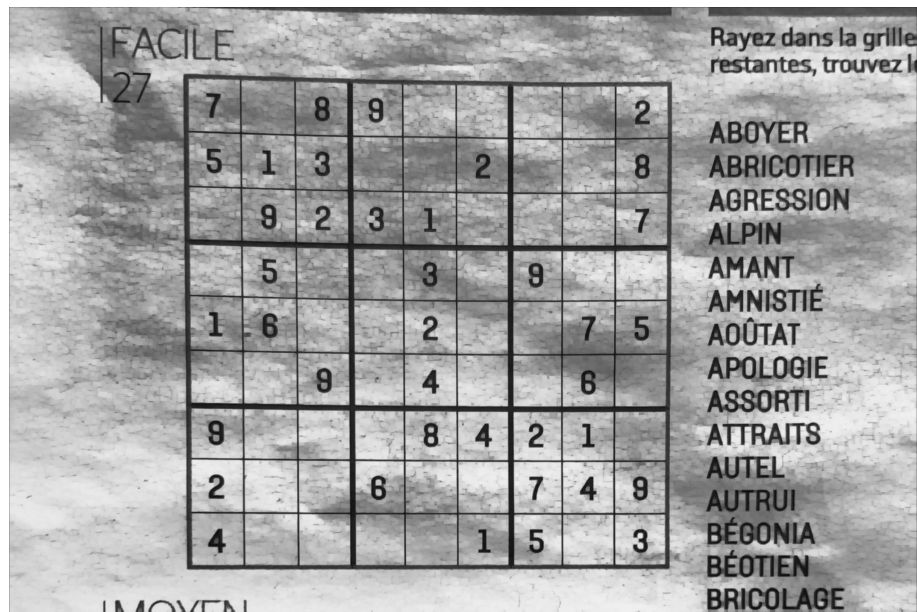


FIGURE 4 – Flou et opérations morphologiques

on utilise deux matrices de convolution : la première pour calculer des approximations des dérivées horizontale et la deuxième des approximations des dérivées verticale.

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

FIGURE 5 – Matrice horizontale G_x

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

FIGURE 6 – Matrice verticale G_y

On peut ainsi déduire la norme du gradient qui sera la nouvelle valeur d'intensité du pixel :

$$p = \sqrt{G_x^2 + G_y^2}$$

Et la direction du gradient qui servira au seuillage des contours :

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

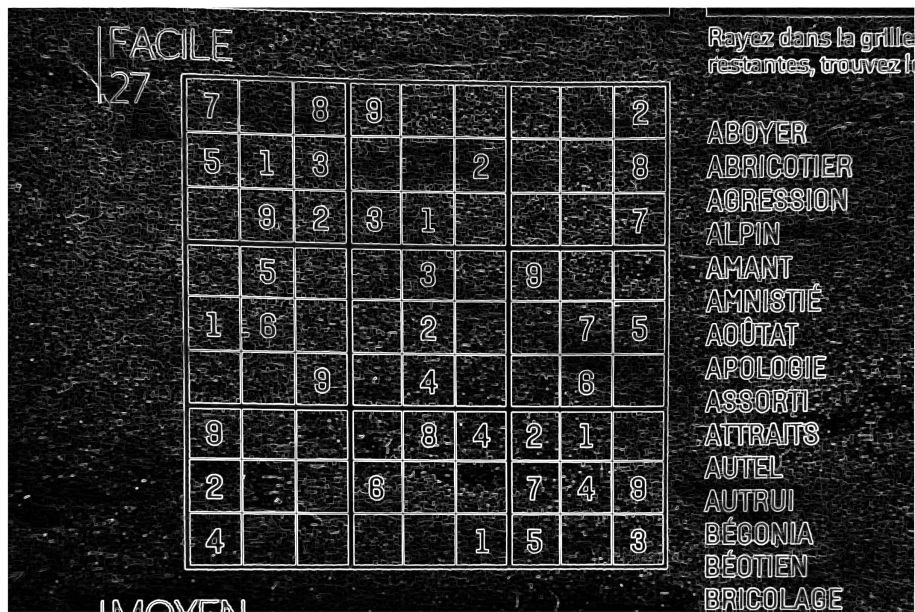


FIGURE 7 – Filtre de Sobel

L'étape suivante est la suppression des non-maxima mais nous ne l'utilisons pas car cette étape rend les bordures trop fines et non détectable pour la suite de l'OCR.

Enfin, c'est le seuillage des contours qui va permettre de binariser notre image, c'est à dire de n'avoir que des pixels blancs ou noirs. Cette binarisation utilise un seuil haut et un seuil bas. Si un pixel a une valeur supérieure au seuil haut, il devient blanc. À l'inverse si un pixel a une valeur inférieure au seuil bas, il devient noir. Si un pixel se situe entre le seuil haut et le seuil, on va utiliser la direction du gradient calculé lors du filtre de Sobel : si il y a un pixel blanc dans cette direction, le pixel devient lui aussi blanc sinon il devient noir. La valeur de ces seuils dépend de chaque image, nous utilisons donc l'algorithme d'Otsu qui permet de déterminer une valeur de seuil automatiquement à partir de l'histogramme de l'image. La valeur trouvé par la méthode d'Otsu devient notre seuil haut et le seuil bas correspond à la moitié du seuil haut.

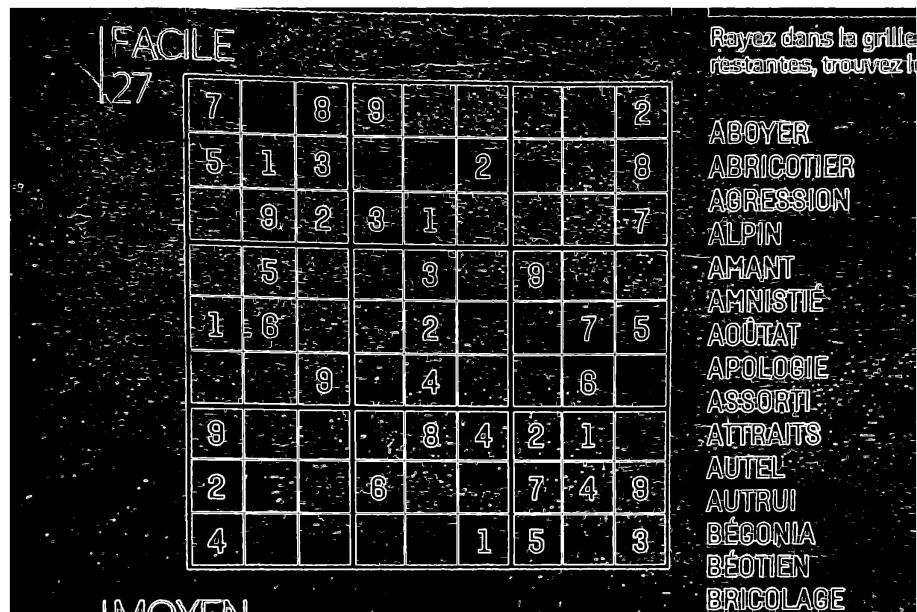


FIGURE 8 – Image finale

5.3 Détection de la grille

5.3.1 Détection des lignes

Tout d'abord, afin de détecter la grille, il a fallu détecter les lignes. Pour cela, nous avons utilisé un principe nommé la transformée de Hough. Cette transformée nous permet de détecter des formes sur une image, mais dans notre cas, son application la plus simple, permettant de détecter des lignes droites, nous suffit. Son principe est le suivant :

- On itère sur chaque pixel blanc de l'image.
- On crée une matrice en 2 dimensions nommée "accumulateur" de taille $\rho * \theta$ avec $\rho = 2 * \text{diag}$ où diag correspond à la diagonale de l'image, et $\theta = 180$.
- On convertit ensuite les coordonnées du pixel sur lequel on itère en coordonnées polaires en utilisant la formule suivante :

$$\rho = x * \cos(\theta) + y * \sin(\theta) \quad (1)$$

- Pour chaque θ allant de 0 à 180, on calcule le ρ correspondant à l'aide de l'équation ci-dessus.

- On ajoute 1 aux coordonnées (ρ, θ) de l'accumulateur, créant ainsi une courbe de 1 dans l'accumulateur pour chacun des pixels blancs de l'image.

- On récupère tous les maximums locaux de l'accumulateur au dessus d'un certain seuil. Autrement dit, on récupère tous les points d'intersections des courbes dans

l'accumulateur au dessus d'un certain seuil.

- Ce seuil est calculé en récupérant le maximum global de l'accumulateur puis en le multipliant par une constante entre 0 et 1. Elle se situe en général autour de 0.5.

- Chaque couple (ρ, θ) récupéré correspond à une ligne en coordonnées cartésiennes.

- Enfin, on dessine toutes ces lignes en testant si chaque pixel (x, y) de l'image fait parti de l'équation de droite (1) en remplaçant avec les valeurs (ρ, θ) trouvées.

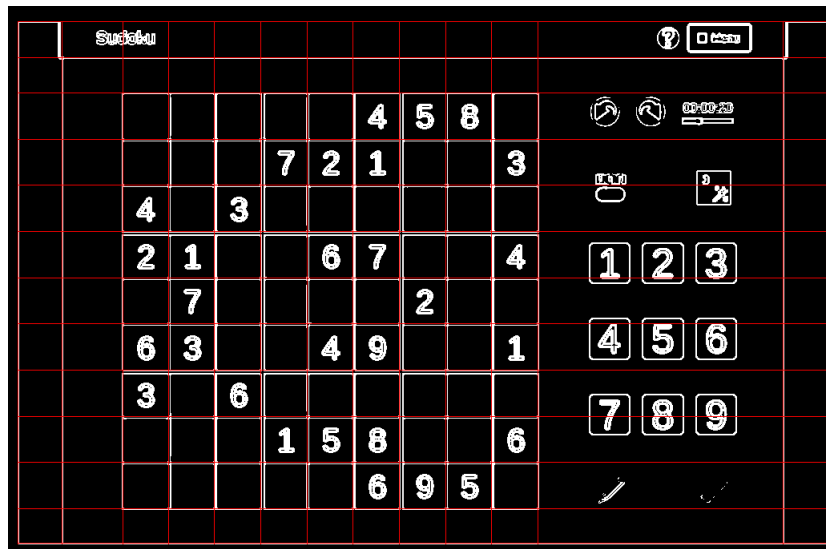


FIGURE 9 – Grille de sudoku après application de la transformée de Hough

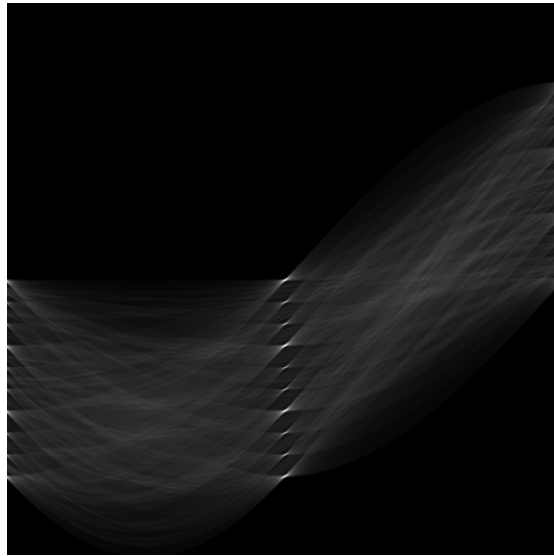


FIGURE 10 – Visualisation de l'accumulateur

5.3.2 Détection des rectangles

Après avoir récupérer les lignes de l'image, nous devons les classifier en lignes verticales et lignes horizontales. Cela nous permet de détecter tous les différents rectangles de l'image.

Pour chaque ligne horizontale, il faut itérer sur toutes les lignes verticales, puis recommencer ce processus pour un total de 4 fois correspondant aux 4 côtés d'un carré. A l'issue de ces itérations, nous avons bien dessiné tous les rectangles de l'image.

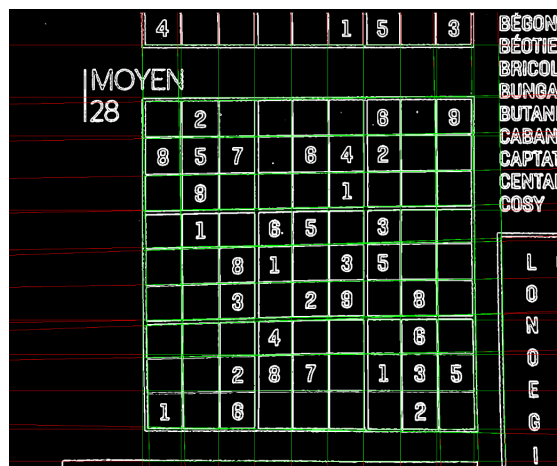
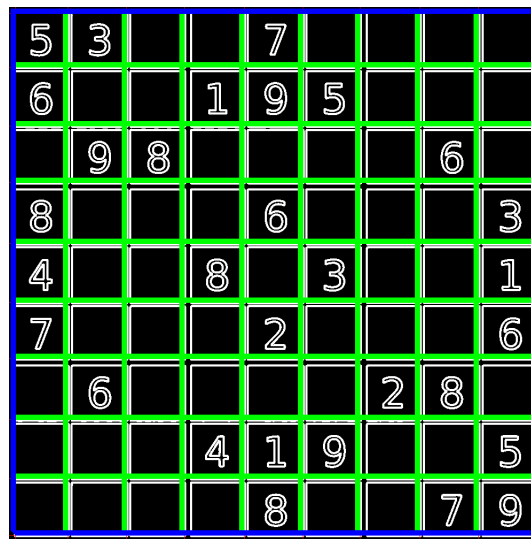


FIGURE 11 – Grille de sudoku avec en vert les carrés détectés

5.3.3 Détection de la grille principale

Ensuite, il faut récupérer la grille principale afin de récupérer ses 4 coins. Pour cela, nous récupérons chaque rectangle précédemment détectés puis nous recherchons ceux qui ressemblent le plus à un carré en comparant l'aire du côté le plus petit et l'aire du côté le plus grand. Parmi ces carrés, nous gardons le carré ayant la plus grande aire. Le résultat nous donne la grille principale de notre sudoku.

Afin de calculer ce résultat, nous utilisons une formule nous donnant un facteur pour chaque rectangle. Celle-ci dépend à la fois de sa ressemblance à un carré et de la taille de son aire. C'est le facteur le plus grand qui correspond à notre grille principale.



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

FIGURE 12 – Grille de sudoku avec en bleu la grille principale détectée

5.3.4 Correction de la perspective et rotation automatique

Une fois les 4 angles de la grille repérés, on procède à une correction de perspective. Celle-ci permet d'appliquer une rotation de l'image si nécessaire. Elle permet également de diviser la grille en sous-image même si certaines lignes ou colonnes n'ont pas été repérées par la transformée de Hough. Pour pouvoir faire cette correction, nous avons besoin d'une matrice homographique H . Découvrons comment la calculer.

L'algorithme de correction de perspective nous impose de résoudre cette équation avec x_i, y_i les coordonnées de la grille dans l'image actuelle et les x'_i, y'_i les nouvelles

coordonnées pour avoir notre grille sur l'image entière :

$$PH = R \Leftrightarrow \begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x'_1 & y_1x'_1 & x'_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y'_1 & y_1y'_1 & y'_1 \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x'_2 & y_2x'_2 & x'_2 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y'_2 & y_2y'_2 & y'_2 \\ -x_3 & -y_3 & -1 & 0 & 0 & 0 & x_3x'_3 & y_3x'_3 & x'_3 \\ 0 & 0 & 0 & -x_3 & -y_3 & -1 & x_3y'_3 & y_3y'_3 & y'_3 \\ -x_4 & -y_4 & -1 & 0 & 0 & 0 & x_4x'_4 & y_4x'_4 & x'_4 \\ 0 & 0 & 0 & -x_4 & -y_4 & -1 & x_4y'_4 & y_4y'_4 & y'_4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Ensuite, nous pouvons résoudre cette équation afin de trouver les h_i :

- On inverse la matrice P
- On obtient $H = P^{-1} * R$
- On transforme H sous forme de matrice 3*3
- On inverse H

Nous connaissons donc maintenant la matrice homographique permettant le changement de perspective de notre image. Il faut désormais l'appliquer à chaque pixel p :

$$N = HC \Leftrightarrow \begin{bmatrix} n_1 \\ n_2 \\ n_3 \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} * \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}$$

Ainsi, on peut calculer les nouvelles coordonnées de notre pixel :

$$\begin{aligned} p'_x &= n_1/n_3 \\ p'_y &= n_2/n_3 \end{aligned}$$



FIGURE 13 – Image originale



FIGURE 14 – Image après correction de perspective

5.3.5 Découpage en sous-image

Grâce à la correction de perspective, l'image complète contient la grille du sudoku. Il suffit donc de diviser les dimensions de l'image par 9 pour extraire 81 sous-images représentant chacune des cases du sudoku.

5.4 Réseau de neurones

5.4.1 XOR

Dans un premier temps, nous avons implémenté un réseau capable d'apprendre le XOR ou "ou exclusif". Le principe est le suivant : on a deux entrées qui sont soit égale à 0 soit à 1 et l'on doit renvoyer 1 si la somme des deux entrées est égale à 1.

Entrée 1	Entrée 2	Résultat
0	0	0
1	1	0
1	0	1
0	1	1

TABLE 2 – Fonctionnement du xor

Pour faire cela, nous avons fait un réseau à 3 niveaux : un niveau d'entrée, un niveau caché et un niveau de sortie. Nous avons utilisé des structures pour simplifier

la manipulation des informations.

```
typedef struct unit
{
    double value;
    double bias;
    unsigned char nb_input;
    struct unit **inputlinks;
    double *inputweights;
} unit;
```

FIGURE 15 – Structure d'un neurone de la première couche

```
typedef struct NeuralNetwork
{
    unit *input[NB_INPUT];
    unit *hidden[NB_HIDDEN];
    unit *output[NB_OUTPUT];
} NeuralNetwork;
```

FIGURE 16 – Structure d'un neurone de la seconde couche

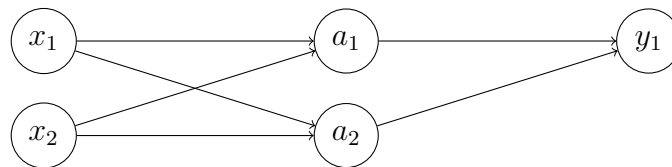


FIGURE 17 – Modèle du premier réseau

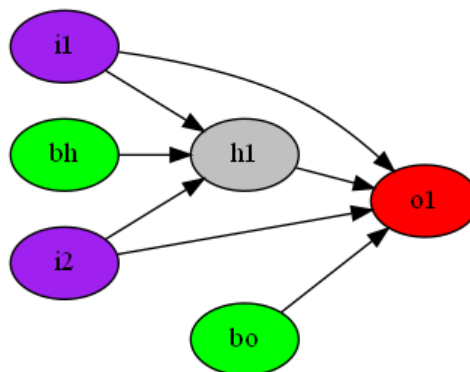


FIGURE 18 – Représentation du réseau de neurones

Samuel a d'abord essayé d'implémenter le xor avec le premier modèle puis Lucas a réussi à implémenter le xor avec le deuxième modèle.

Passons au fonctionnement du réseau de neurones :

Pour calculer la valeur d'un neurone k de la couche cachée ou de la couche de sortie il faut d'abord calculer la somme des produits des neurones de la couche précédente et du poids de la liaison correspondante. Il faut ensuite ajouter le biais du neurone k. Notons S cette somme, on a donc

$$S = \sum_{i=1}^n w_i x_i$$

avec w_i le poids de la liaison entre le neurone i et le neurone k , n le nombre de neurones de la couche précédente et θ_k le biais associé au neurone k . On doit ensuite passer la valeur obtenue dans une fonction d'activation (une fonction sigmoïde dans notre cas) donnée par l'équation $f(x) = \frac{1}{1+e^{-x}}$. Pour résumer, la valeur du neurone k sera donc $\frac{1}{1+e^{-(S+\theta_k)}}$. Une fois cela appliqué à la couche cachée et au neurone de sortie, il faut corriger les poids. Pour cela dans un premier temps, il faut :

- Calculer la différence entre le résultat obtenu et celui attendu, nous noterons cette différence e et o la valeur obtenue.
- Calculer le gradient d'erreur de la couche de sortie. La formule pour l'obtenir est $g = o(1 - o) * e$
- On peut ainsi changer le biais du neurone de sortie et des poids des liaisons entre la couche cachée et la couche de sortie. Grâce à la formule :

$$w_{ik} = w_{ik} + g * \alpha * o$$

α est le taux d'apprentissage, plus il est élevé plus le réseau apprendra rapidement mais s'il est trop élevé, le réseau ne convergera pas vers la bonne réponse.

- Il faut ensuite changer le poids des liens entre la couche d'entrée et la couche cachée. Pour cela nous calculons le gradient d'erreur de chaque neurone de la couche cachée grâce à la formule suivante :

$$\theta_j = o(1 - o) \sum_k g_k w_{jk}$$

La somme est la somme des produits entre le poids de la liaison entre le neurone de la couche cachée j et du neurone de sortie k et le gradient du neurone de sortie k .

- Pour changer le poids entre un neurone d'entrée et un neurone de la couche cachée il faut ajouter au poids le calcul suivant :

$$\alpha \times \theta_j \times o_i$$

Puis il faut répéter ce processus sur chacun des exemples à tour de rôle jusqu'à ce que le neurone de sortie soit assez proche de la bonne réponse à chaque fois.

5.4.2 Sauvegarde des poids

Pour le réseau de neurones servant à détecter les caractères, l'utilisateur aura la possibilité d'entraîner le réseau mais ne sera pas obligé. Nous devons donc être capable de sauvegarder les poids reliant chacun des neurones lors de notre phase d'apprentissage en amont puis de les charger sur les bonnes connexions au lancement du programme pour l'utilisateur. Pour ce faire, nous sauvegardons l'ensemble de ces valeurs dans un fichier en les séparant d'un retour à la ligne. Ensuite, pour recharger les valeurs, il suffit de lire ligne par ligne, dans le même ordre que la sauvegarde, et d'attribuer les bonnes valeurs aux bonnes connexions.

5.5 Solveur du sudoku

Notre solver consiste en un programme en ligne de commandes, prenant en entrée un sudoku sous le format texte spécifié dans le cahier des charges et écrivant en sortie dans un fichier « .result » le sudoku résolu dans le même format.

Ce solver utilise l'algorithme de « Backtracking ». Cet algorithme consiste à lire un sudoku case par case, en commençant par la case en haut à gauche et en finissant par case en bas à droite. Pour chacune de ces cases, l'algorithme vérifie si elle est vide. Si c'est le cas alors ce dernier va essayer d'écrire à la place tous les chiffres entre 1 et 9. Si un de ces chiffres n'existe pas dans, à la fois, la colonne, la ligne et le carré où se trouve la case, alors un autre appel récursif sera effectué en simulant la case vide comme case possédant un chiffre. Si aucun des chiffres entre 1 et 9 est valide alors l'algorithme considérera que le dernier chiffre placé dans une case vide antérieure à celle-ci est incorrect.

Une fois arrivé à la case en bas à droite, deux cas sont possibles : le sudoku est résolu ou bien le sudoku n'est pas résoluble. Notre solver ne traite pas du cas où le sudoku n'est pas résoluble.

Pour résumer, notre solver est constitué de deux fonctions principales : une fonction vérifiant si un sudoku est correct (pas deux fois le même chiffre dans chacun des carrés, colonnes et lignes) et une fonction implémentant le back tracking.

5.6 Interface utilisateur

L'interface graphique a été réalisée dans le but qu'elle produise le moins d'erreurs possible (aucune au jour de la première soutenance). Étant une partie nous permettant de regrouper l'ensemble de nos tâches, il serait dommage qu'une erreur venant de cette UI nous perturbe dans le bon déroulement de ce projet. C'est pour cette raison

que l'entièreté de cette partie est compilée avec les flags -Wall, -Wextra, -Werror et -pedantic-errors.

5.6.1 Bibliothèque GTK

L'UI utilise la bibliothèque GTK à la version 3.24. GTK est un ensemble de bibliothèques logicielles, c'est-à-dire un ensemble de fonctions permettant de réaliser des interfaces graphiques.

GTK possède certaines normes pouvant afficher des messages d'erreurs ainsi que des messages d'avertissement. L'UI a été codée dans le but d'éviter ce genre de message, toujours dans le même optique de produire le moins d'erreur possible.

Une interface, selon GTK, est un ensemble de widgets (par exemple les boutons sont des GtkButton, étant eux-mêmes des GtkWidgets). Notre Interface est donc constituée de trois principaux types de widgets :

- GtkImage : Ce widget permet d'afficher une image dans notre interface graphique.
- GtkButton : Ce widget permet d'afficher un bouton et de lier des fonctions à ce dernier, les lançant ensuite lors d'un appui.
- GtkFileChooser : Ce widget permet d'afficher un gestionnaire de fichiers, rendant possible le chargement et la sauvegarde d'image par l'UI.

5.6.2 Glade

Même s'il est plutôt facile de coder une interface à la main avec GTK grâce aux widgets, notre UI finale, à cause de nos besoins, représentait un travail trop fastidieux. Nous avons donc utilisé Glade. Glade est un outil RAD (Rapid Application Development) qui permet de développer rapidement et facilement des interfaces utilisateur pour le toolkit GTK. Ce logiciel nous a permis d'obtenir un fichier au format XML, facile à importer dans notre code C grâce à GtkBuilder. Il ne nous restait plus qu'à lier nos fonctions aux différents composants de l'UI.

5.6.3 Description des composants de l'UI

L'UI est composée de deux grandes parties :

1. Partie affichage images

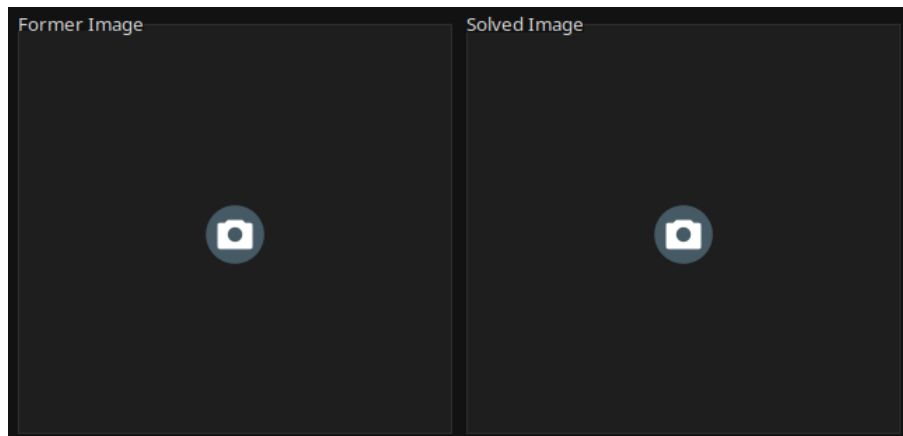


FIGURE 19 – Affichage image

- Sur la gauche s’affiche l’image du sudoku chargé et prêt à être résolu par le programme.
- Sur la droite s’affiche l’image du sudoku résolu, reconstruit et prêt à être sauvegardé.

2. Partie contrôles

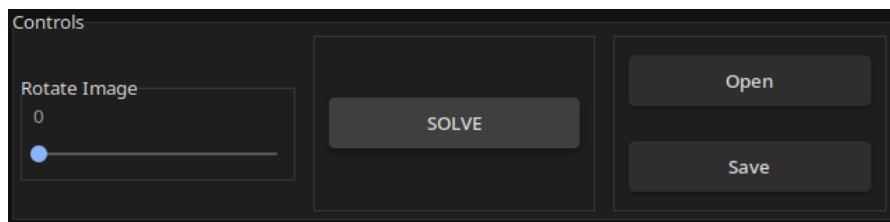


FIGURE 20 – Affichage contrôles

- **Rotate image** : comme son nom l’indique, ce contrôle permet de redresser l’image chargée par rapport à un angle. Cet angle est donné par le numéro au-dessus du slider. Le redressement est fait en temps réel, dès l’instant qu’un utilisateur bouge le slider. L’image est mise à jour et affichée dans la partie gauche de l’interface.
- **SOLVE** : ce bouton permet de lancer le programme principal du projet, se terminant au final par l’affichage de la grille résolue dans la partie droite de l’UI.
- **Open** : ce bouton permet de charger une grille à résoudre. Pour cela, il ouvre un gestionnaire de fichiers permettant de sélectionner l’image dans laquelle la grille à résoudre se trouve.

- **Save** : ce bouton permet de sauvegarder la grille résolue. Pour cela, il ouvre un autre gestionnaire de fichiers permettant d'indiquer où sauvegarder l'image.

6 Avenir du projet

Le prétraitement de l'image est fonctionnel mais pose des problèmes lors de la détection de la grille. Augustin et Lucas vont donc travailler plus en cohésion afin d'obtenir une image propre et parfaitement interprétable pour la transformée de Hough. De plus, le seuil utilisée dans la méthode de Hough (qui est aujourd'hui modifié manuellement) devra être déterminée automatiquement afin de s'adapter à toutes les images ou les images devront être plus uniforme en sortie du prétraitement afin d'avoir un seuil fixe.

Nous devons améliorer la détection de la grille principale car elle est actuellement fonctionnelle uniquement sur 2 images différentes. Sur les autres images, nous détectons bel et bien un carré mais il est trop grand par rapport à la grille qu'il devrait encadrer.

Aujourd'hui, nous arrivons, plus ou moins manuellement, à extraire chacune des cellules d'une grille de sudoku. Cependant, les images résultantes sont de dimensions différentes et peuvent comporter une partie de la grille sur les bords des sous-images. Ces sous-images devront donc être "nettoyées" pour améliorer l'efficacité de la reconnaissance de caractères.

L'utilisation du réseau de neurones doit être étendu à la détection de chiffres. Pour son apprentissage, nous réaliserons un script permettant de télécharger des milliers de chiffres de différentes polices afin d'avoir un éventail de données très large et d'optimiser les chances de reconnaissance.

Enfin, l'interface utilisateur va s'étoffer au fur et à mesure du développement des différentes fonctionnalités.

7 Conclusion

Nous sommes satisfait du développement du projet et nous espérons présenter un OCR complet, fonctionnel et répondant aux éléments cités dans la section 6.