



EPITA

PROJET OCR S3

Rapport de soutenance final



Samuel LAMBERT
Mathéo CRESPEL
Augustin CLAUDE
Lucas BESNARD

Septembre 2022 - Décembre 2022

Table des matières

1	Introduction	3
2	Les membres du groupe	3
3	La répartition des tâches	3
4	L'état d'avancement du projet	4
5	Aspects techniques	6
5.1	Chargement de l'image	6
5.2	Prétraitement de l'image	7
5.2.1	Rotation manuelle	7
5.2.2	Niveau de gris	7
5.2.3	Normalisation	7
5.2.4	Réduction de bruits	8
5.2.5	Filtre de Canny	10
5.3	Détection de la grille	13
5.3.1	Détection des lignes	13
5.3.2	Rotation automatique	15
5.3.3	Détection des rectangles	16
5.3.4	Détection de la grille principale	16
5.3.5	Correction de la perspective	17
5.4	Extraction et traitement des 81 cases	18
5.4.1	Récupération	18
5.4.2	Traitement	19
5.4.3	Détourage des chiffres	20
5.4.4	Redimensionnement	21
5.5	Réseau de neurones	22
5.5.1	XOR	22
5.5.2	Sauvegarde des poids	24
5.5.3	Le réseau de reconnaissance de caractères	24
5.6	Solveur du sudoku	26
5.7	Interface utilisateur	27
5.7.1	Bibliothèque GTK	27
5.7.2	Glade	28
5.7.3	Description des composants de l'UI	28
5.8	Generation de grille	33
6	Améliorations possibles	35

7 Conclusion	36
---------------------	-----------

1 Introduction

Le groupe « Grid'okuCr » est fier de vous présenter ce rapport final 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 améliorations possibles pour l'avenir du projet.

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 dernière soutenance, nous sommes capables de :

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

utilisable pour notre la détection de grille (partie 5.2). Ce prétraitement consiste à :

- Transformer l'image en niveau de gris
 - Normaliser l'image
 - Appliquer un filtre flou
 - Appliquer un filtre de Canny
2. Déetecter les lignes, les carrés puis la grille exacte du sudoku présente 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. Appliquer un traitement sur chacune des 81 cases pour améliorer la détection de chiffres et les redimensionner en 28*28 pixels.
 6. D'obtenir un réseau de neurones capable de reconnaître les chiffres. (partie 5.5)
 7. Résoudre une grille de sudoku en utilisant la technique du « back tracking ». (partie 5.6)
 8. Afficher la grille résolue dans une image en mettant en rouge les chiffres ajoutés par rapport à l'image originale.
 9. Disposer d'une interface graphique complète permettant de :
 - Charger une image
 - La tourner manuellement
 - Effectuer l'ensemble des étapes de la résolution pas à pas ou d'un seul coup
 - Revenir sur une étape précédente
 - Sauvegarder l'image à chacune des étapes
 - Corriger la détection de chiffres s'elle n'a pas été bonne
 10. Disposer d'un programme en version console permettant de résoudre un sudoku ou d'utiliser le réseau de neurones grâce aux options suivantes :

```

Usage : ./gridoku-ocr <image_path> [OPTIONS] or ./gridoku-ocr nn [OPTIONS]

Options ocr :
    -o <file_path> : save the result into <file_path>
    -v : print details of process
    -s <folder> : save all intermediate image into <folder>
    -w <file_path> : specify the weights file for neural network

Options nn :
    train <folder> : train the neural network with images in <folder>
    test <image_path> : predict the number in image
    -w <file_path> : specify the weights file (for testing or training)

gridoku-ocr: Wrong arguments.

```

FIGURE 1 – Options de la version console de notre OCR

5 Aspects techniques

5.1 Chargement de l'image

Pour commencer l'OCR, nous avons dû 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 2 – Structure d'un pixel

```

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

```

FIGURE 3 – 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 Prétraitemet 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 .

$$\begin{aligned}x_1 &= (x - x_0) * \cos(\theta) - (y - y_0) * \sin(\theta) + x_0 \\y_1 &= (x - x_0) * \sin(\theta) + (y - y_0) * \cos(\theta) + y_0\end{aligned}$$

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

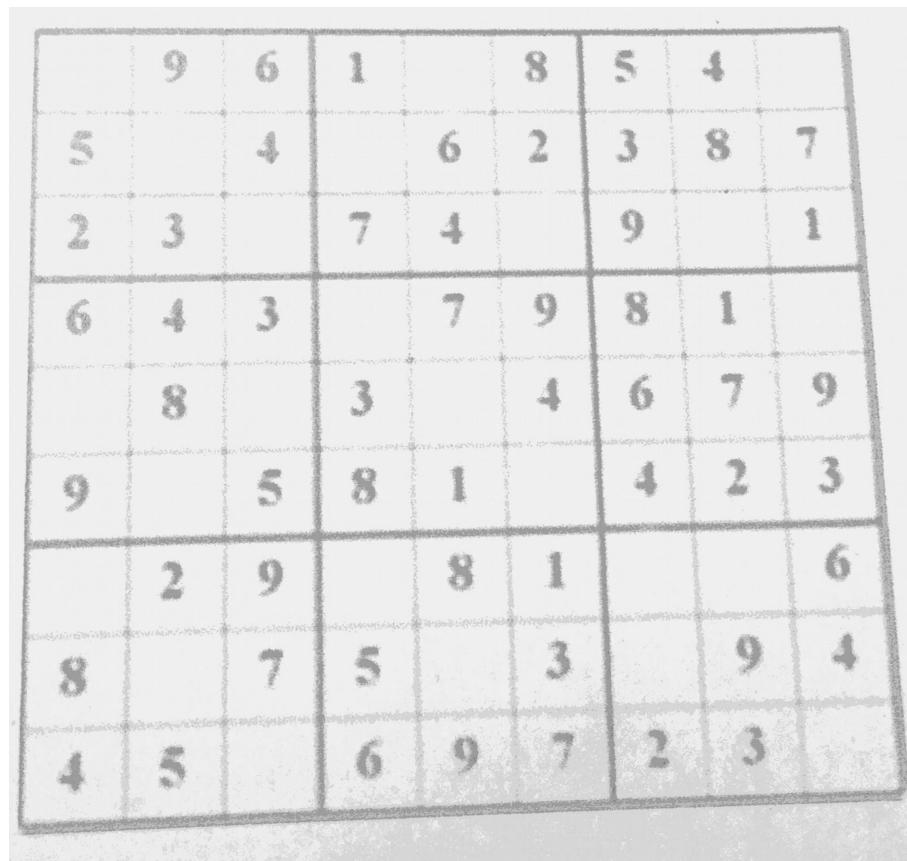


FIGURE 4 – Mise en niveau de gris et normalisation

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 erosion). 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 inexistant 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 :

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

On obtient donc le calcul suivant pour chacun des pixels :

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

Après avoir appliqué ce flou, nous utilisons deux opérations morphologiques : la dilatation et l'érosion. Ces deux processus permettent respectivement d'agrandir et de réduire les zones les plus clairs. 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 ses pixels voisins

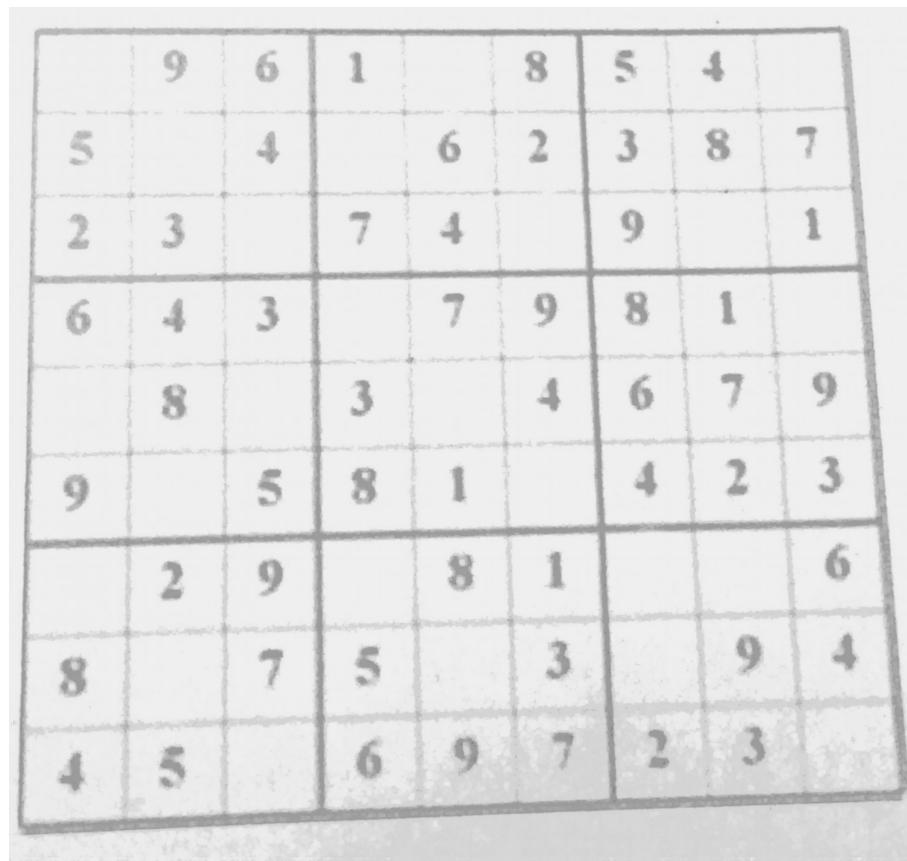


FIGURE 5 – Flou et opérations morphologiques

5.2.5 Filtre de Canny

La dernière étape de notre prétraitement de l'image est l'application d'une partie filtre de Canny. L'utilisation de celui-ci n'était pas encore acté lors de la première soutenance car il faisait apparaître de nombreuses doubles lignes ce qui perturbait la détection de la grille. Cependant, nous avons réussi à améliorer la détection de lignes afin de conserver uniquement les droites les plus importantes. De ce fait, nous avons décidé de garder le filtre de Canny qui est la dernière étape du prétraitement.

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, 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 6 – Matrice horizontale G_x

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

FIGURE 7 – 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 :

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

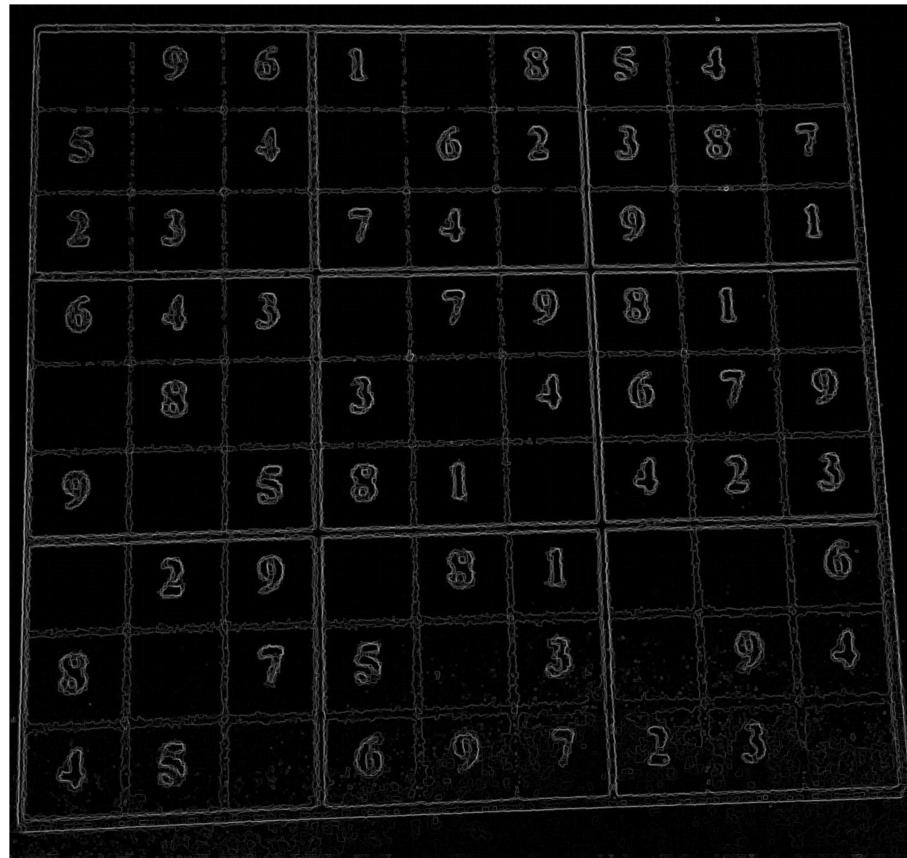


FIGURE 8 – Filtre de Sobel

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'Ötsu qui permet de déterminer une valeur de seuil automatiquement à partir de l'histogramme de l'image. La valeur trouvé par la méthode d'Ötsu devient notre seuil haut et le seuil bas correspond à la moitié du seuil haut.

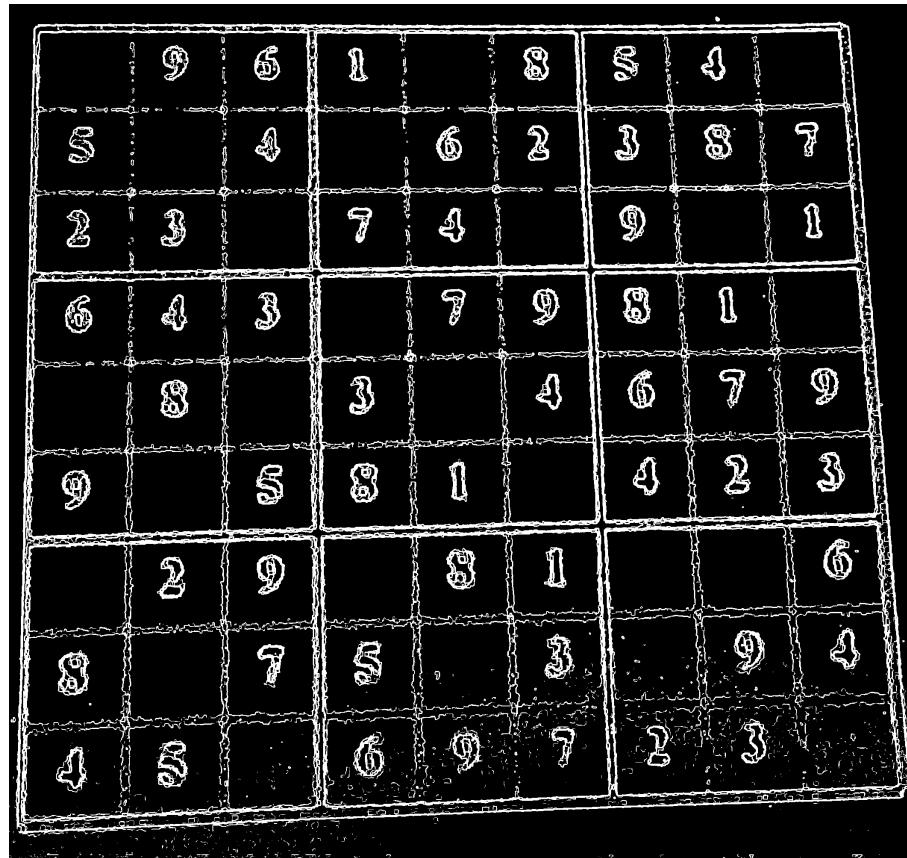


FIGURE 9 – Image finale

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 créer 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.

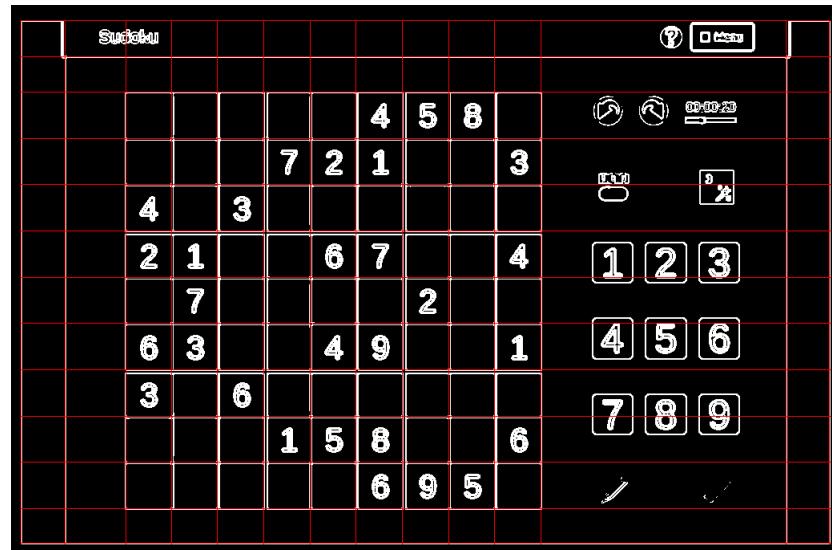


FIGURE 10 – Grille de sudoku après application de la transformée de Hough

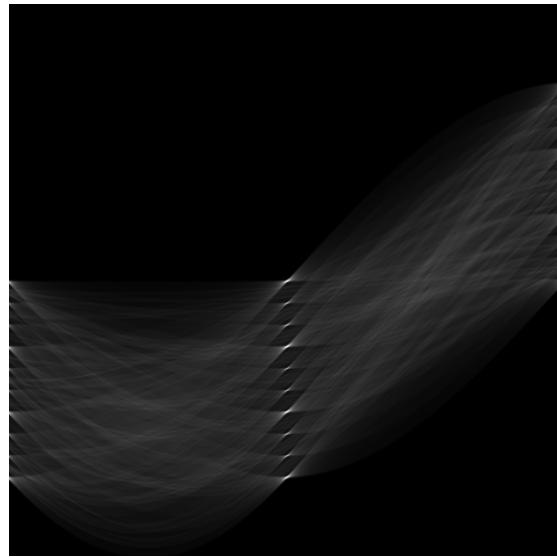


FIGURE 11 – Visualisation de l'accumulateur

5.3.2 Rotation automatique

Une fois les lignes récupérées, nous devons procéder à une rotation automatique si nécessaire.

Pour cela, nous regardons la liste des θ de toutes les lignes puis nous récupérons l'angle le plus commun. Nous appliquons ensuite une rotation par rapport à l'angle trouvé si celui-ci représente un changement de plus de 10° .

Si la rotation n'a pas été effectuée, alors le programme continue sans modifier l'image.

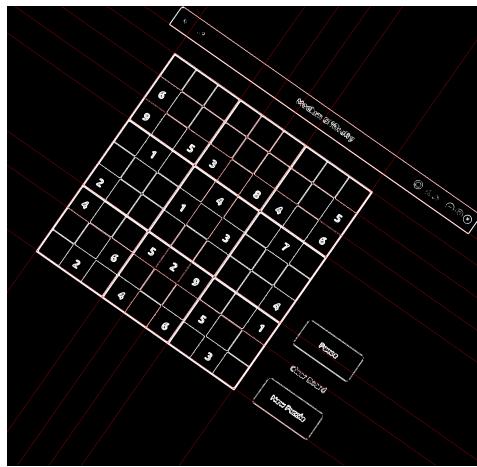


FIGURE 12 – Image 5 avant auto-rotation

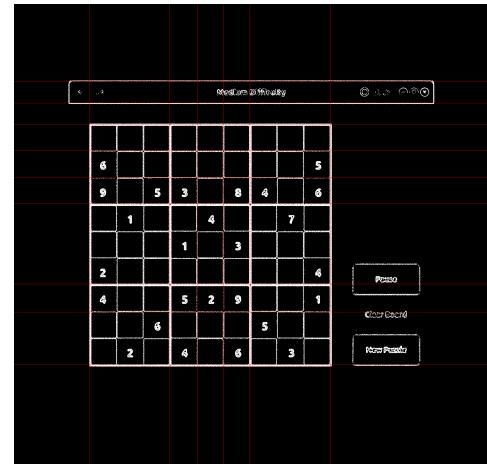


FIGURE 13 – Image 5 après auto-rotation

5.3.3 Détection des rectangles

Après avoir effectué ou non la rotation automatique sur notre 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.

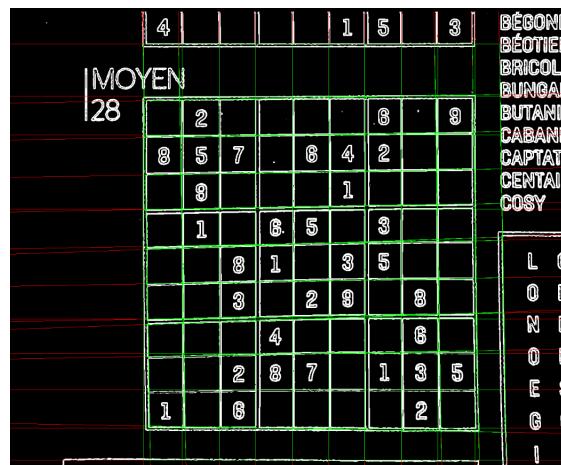


FIGURE 14 – Grille de sudoku avec en vert les carrés détectés

5.3.4 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 utilisant 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.

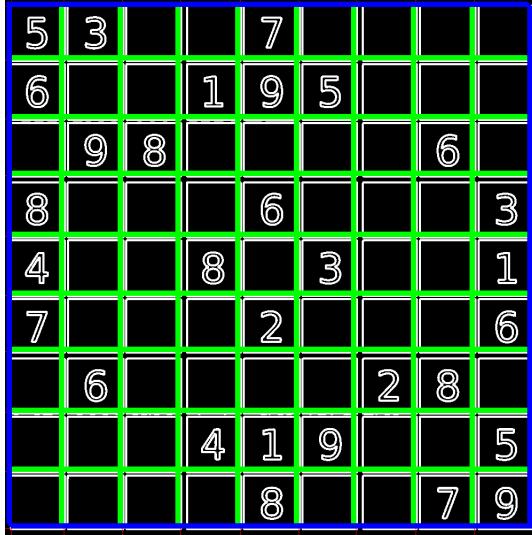


FIGURE 15 – Grille de sudoku avec en bleu la grille principale détectée

5.3.5 Correction de la perspective

Une fois les 4 angles de la grille repérés, on procède à une correction de perspective. Celle-ci permet d'avoir la grille qui occupe toute l'image. 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 :

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

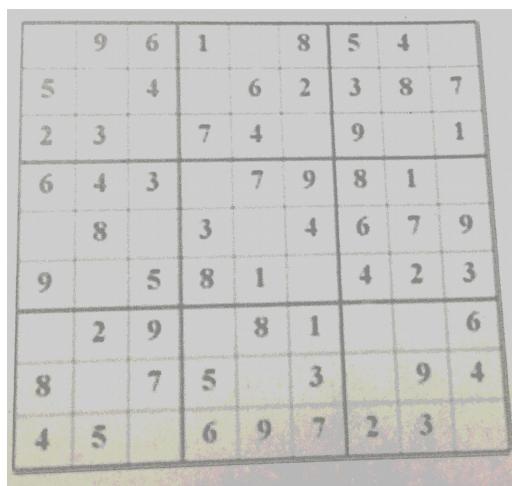


FIGURE 16 – Image originale



FIGURE 17 – Image après correction de perspective

5.4 Extraction et traitement des 81 cases

5.4.1 Récupération

Grâce à la correction de perspective, l'image complète est carrée et contient exactement 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.

5.4.2 Traitement

Dans la suite de l'OCR, nous utilisons un réseau de neurones qui permet de reconnaître un chiffre à partir d'une image. Cependant, pour maximiser les chances qu'il reconnaisse correctement les chiffres, il est essentiel de lui envoyer des images de qualités. Nous avons tout d'abord simplement essayé de lui envoyer les sous-images extraites du filtre de Canny après la correction de perspective. Malheureusement, les résultats n'ont pas du tout été à la hauteur de nos espérances : le réseau de neurones avait du mal à reconnaître les chiffres car les sous-images contenaient encore les bords de la grille et les chiffres étaient "creux" dû au filtre de Canny alors que le réseau est uniquement entraîné à reconnaître des chiffres "pleins".

Nous avons alors décidé d'utiliser une méthode différente. Nous récupérons l'image obtenue juste avant le filtre de Canny (nous l'appellerons *image_before_canny*), c'est à dire celle qui a passé les différentes étapes de réduction de bruits. Après avoir détecté les 4 angles de la grille avec la transformée de Hough, nous appliquons la correction de perspective sur *image_before_canny*. On obtient ainsi les 81 sous-images suivantes :

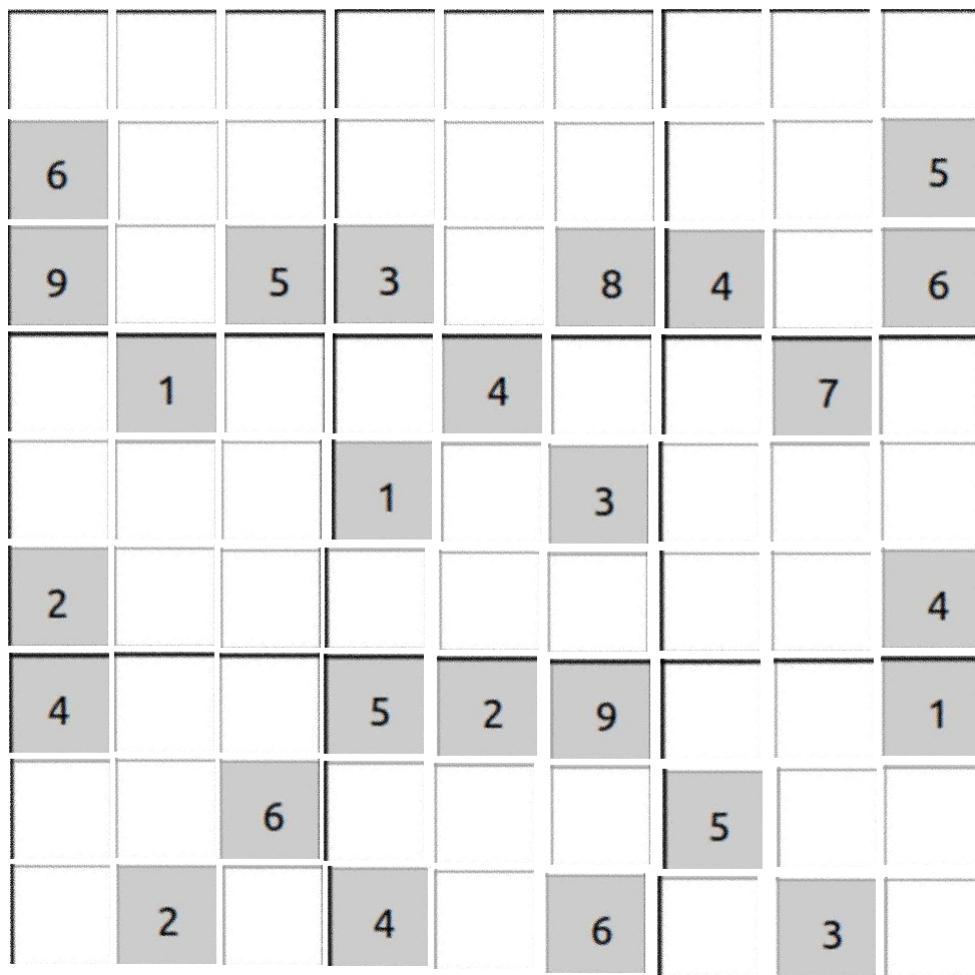


FIGURE 18 – 81 cases avant le traitement

Ensuite, nous binarisons ces sous-images grâce à l'algorithme d'Ötsu. Celui-ci apportait de mauvais résultats lorsque qu'il était appliqué à une image entière à cause des variations d'intensités. Mais, il est ici très efficace puisque ce sont des petites images avec peu de différences de luminosité.

5.4.3 Détourage des chiffres

Nous avons maintenant les 81 cases constituées uniquement de noir et de blanc mais il reste un problème : ces sous-images contiennent toujours les bords de la grille et les chiffres n'occupent pas l'entièreté de l'image.

Nous allons donc rogner chacune des images pour qu'elles ne contiennent plus que les chiffres. Pour ce faire, nous utilisons un algorithme de « Blob Detection ».

Pour chacune des 81 sous-images binarisées :

Nous commençons par explorer un petit carré au centre de l'image, puis, tant qu'il y a des pixels blancs sur les contours de ce carré, nous l'agrandissons afin de délimiter exactement le chiffre.

5.4.4 Redimensionnement

Le réseau de neurones permettant la reconnaissance des chiffres prend un certain nombre d'entrée fixe. Ainsi, chaque case doit être redimensionné en 28*28 pixels. Ainsi, chaque pixel de la nouvelle image de 28*28 :

$$p_{x,y} = p'_{new_x, new_y}$$

avec p' : un pixel de l'image original
 $new_x = \frac{x}{28/img_width}$ et $new_y = \frac{y}{28/img_height}$

Voici donc le résultat final du traitement après détection de la grille, ces images seront ensuite données au réseau de neurones.

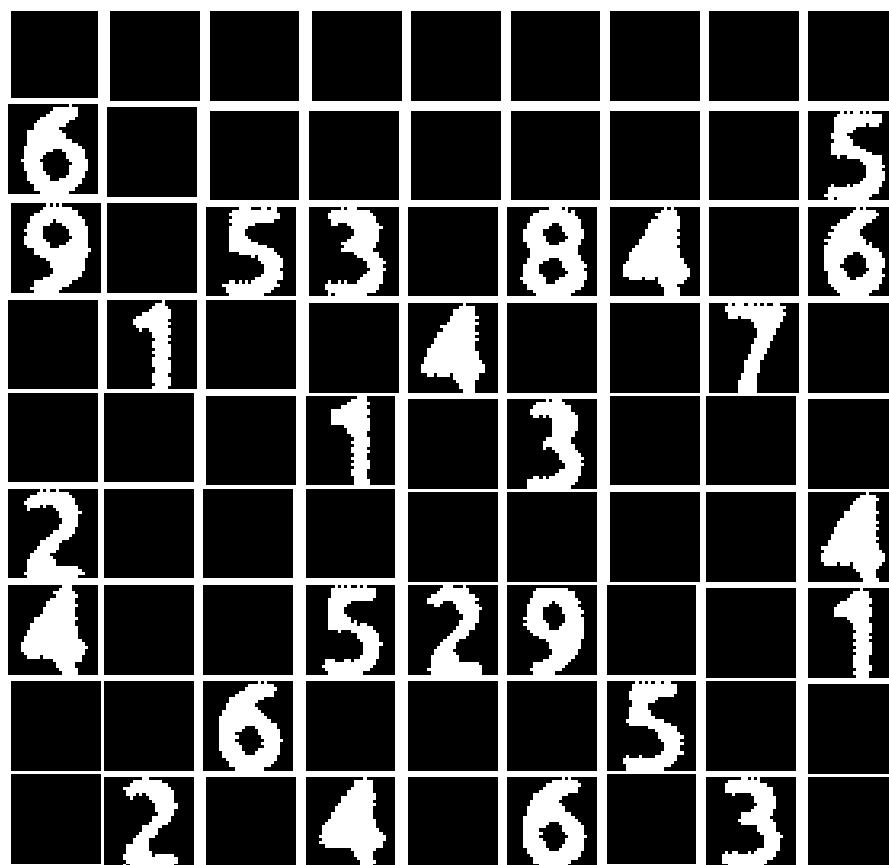


FIGURE 19 – 81 cases après le traitement et le redimensionnement

5.5 Réseau de neurones

5.5.1 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 20 – 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 21 – Structure d'un neurone de la seconde couche

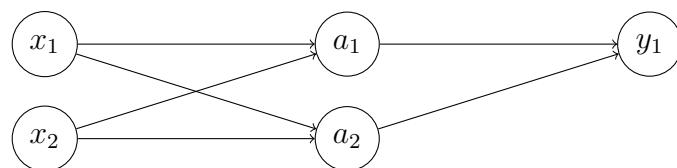


FIGURE 22 – Modèle du premier réseau

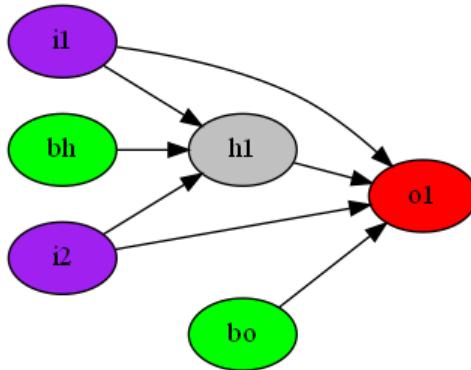


FIGURE 23 – Représentation du réseau de neurones

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.

5.5.2 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.

5.5.3 Le réseau de reconnaissance de caractères

Suite à la première soutenance, nous avons réussi à faire fonctionner le xor-network avec le premier modèle 22 mais avec les structures du second ce qui permet un meilleur contrôle des unités. Ce modèle a l'avantage d'être assez facilement adaptable à un grand nombre de neurones. Il a donc été relativement simple d'avoir un réseau avec 784 entrées (1 par pixel, la grille étant découpée en images de 28x28 pixels). Il a néanmoins fallu adapter le fait que l'on trouve le neurone avec la plus haute valeur dans la couche de sortie qui nous donne le chiffre sur l'image. Aussi, lors de la première implémentation, lors du calcul des gradients d'erreur de la couche de sortie, nous avions pris en compte le fait qu'il n'y avait qu'un neurone de sortie.

Aussi nous avons dû augmenter le nombre de neurones cachés, cependant un nombre trop élevé de neurones cachés aurait trop ralenti notre réseau car pour chaque neurone

de la couche caché en plus il y a $784+10$ (neurones d'entrée+neurones de sortie) calcul en plus. Nous sommes donc arrivés à la conclusion après plusieurs essai que 15 neurones dans la couche cachée nous permettait d'avoir un temps d'exécution pas trop élevé et une précision satisfaisante. On arrive donc à cette structure :

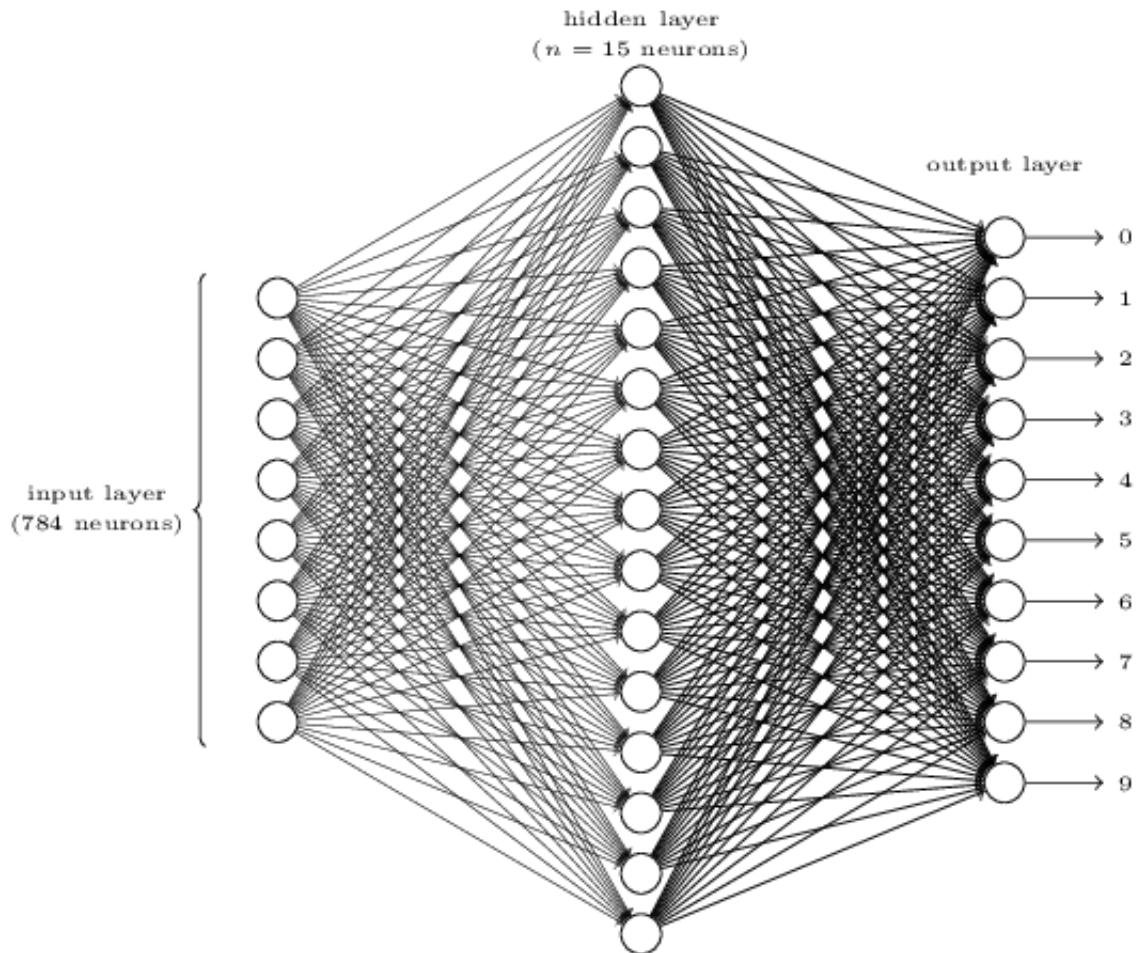


FIGURE 24 – Le réseau final

Ensuite il a fallut générer un jeu de données pour entraîner le réseau, pour cela nous avons utilisé un script python qui écrit chaque chiffre de 1 à 9 (pas de zéro) dans différentes polices d'écriture et place l'image dans un dossier correspondant au chiffre.

L'entraînement

Un fois le jeu de données récupéré, nous pouvons entraîner le réseau simplement en lui indiquant le chemin du dossier contenant les images des chiffres. Ce dossier doit

absolument contenir 10 sous-dossiers de 0 à 9, ainsi, il va automatiquement récupérer chacune des images dans les sous-dossiers pour s'entraîner avec celle-ci.

Ensuite, pour chaque chiffre, nous le faisons passer dans le réseau puis nous appliquons la correction des poids et nous passons au chiffre suivant. Nous faisons cela pour toutes les images jusqu'à obtenir 90% de précision ou après avoir fait 10 passages sur chaque image. Pour des soucis de précisions, nous avons décidé de prendre un learnrate de 0,1. En plus d'arriver à reconnaître des chiffres écrit par ordinateur, nous avons essayé d'entraîner le réseau sur des chiffres manuscrits et cela a fonctionné.

L'utilisation

Une fois que la découpe des images a été faite, le réseau les reçoit et les transforme en un tableau qui sera envoyé au solver.

5.6 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 la 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 antérieur 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. Dans ce dernier cas, notre solver renvoie simplement une grille NULL, indiquant aux autres fonctions utilisant le solver que le sudoku n'est pas résoluble.

Pour conclure, 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.

5.7 Interface utilisateur

L'interface graphique a été réalisée dans le but qu'elle produise le moins d'erreurs possible. 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. Le deuxième but de cette UI est de fournir la capacité de nous permettre de passer d'une étape à une autre, de la manière la plus fluide possible, que cela soit dans un sens ou dans l'autre. Elle doit nous permettre de charger n'importe quelle image, n'importe quand et à n'importe quelle étape. Finalement, elle doit prendre en compte le plus grand nombre de "corner cases" et afficher le plus de messages d'erreur en évitant tout segfault ou autre signaux.

5.7.1 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 GtkWidget). Notre Interface est donc constituée de six principaux types de widgets :

- GtkDrawingArea : Ce widget permet de dessiner sur une surface. Nous l'utilisons pour dessiner l'ensemble des images 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.
- GtkLabel : Ce widget permet d'afficher du texte simple, en gras, souligne ou bien italique.
- GtkEntry : Ce widget permet à un utilisateur de saisir du texte comme, par exemple, des coordonnées pour placer des chiffres dans une grille de sudoku.
- GdkPixbuf : Ce widget provient de la bibliothèque Gdk. Il permet de sauvegarder une image sous forme de buffer de pixels utilisable parGtk comme un widget GtkImage.

5.7.2 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. En effet, il aurait fallu créer chaque widget à la main, un par un, dans nos fichiers sources C. Pour nous faciliter les choses, 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.

5.7.3 Description des composants de l'UI

L'UI est composée de trois grandes parties :

1. Partie principale (affichage images et contrôles sous-étapes)

Cette partie est le cœur de l'UI. Elle permet d'afficher chacune des étapes de notre projet ainsi que l'ensemble des contrôles disponibles pour chacune d'entre elles. Cette partie prend plusieurs formes en s'adaptant aux options disponibles pour chaque étape.

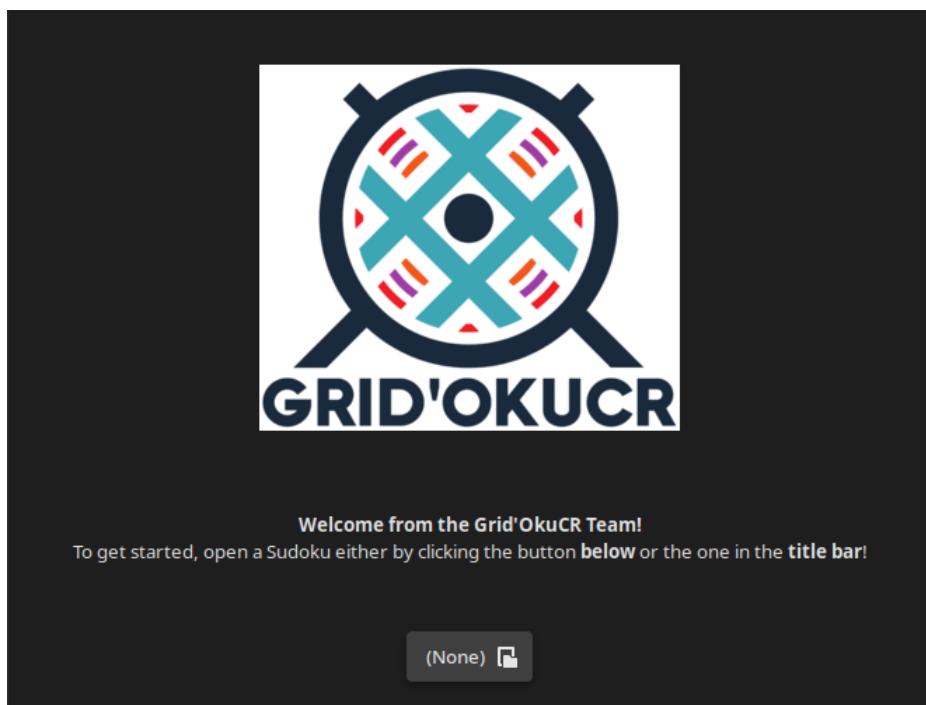


FIGURE 25 – Accueil

Voici l'ensemble des différentes formes que peut prendre la partie principale :

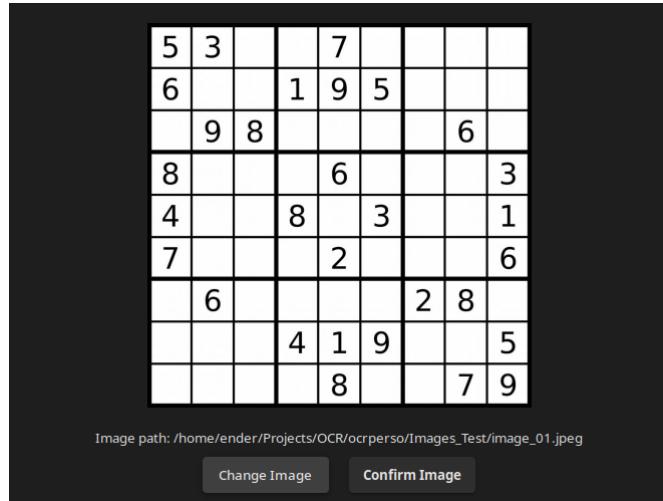


FIGURE 26 – Chargement d'image

Cette forme comporte deux boutons : Un pour confirmer l'image affichée et l'autre pour la changer.

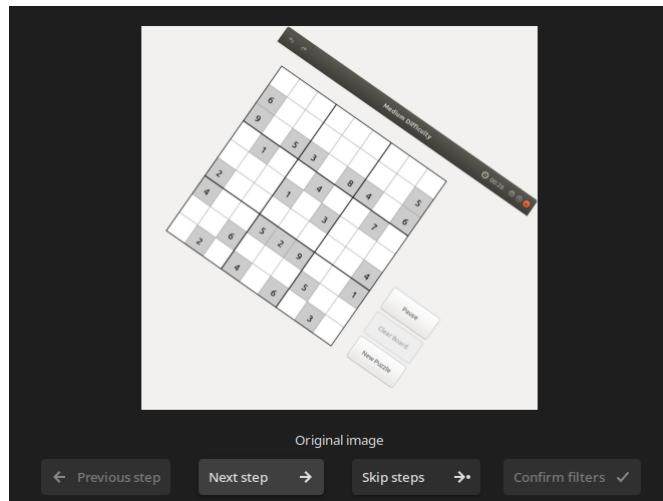


FIGURE 27 – Sous-étapes

Cette forme comporte quatre boutons : 3 pour changer les sous-étapes actuelles et un quatrième pour confirmer l'ensemble de ces étapes.

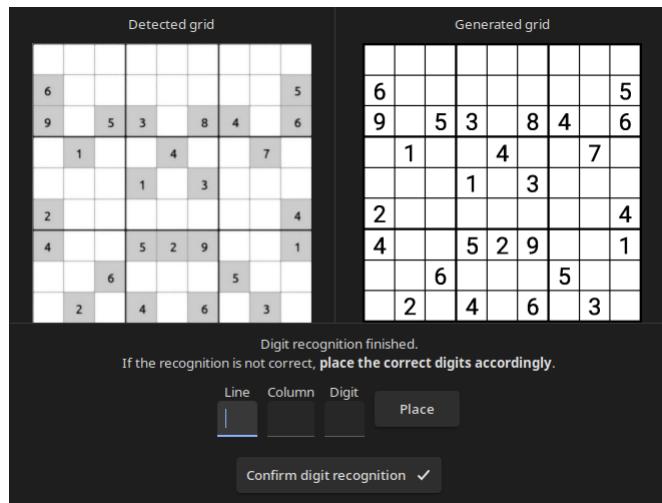


FIGURE 28 – Reconnaissance de caractere et correction

Cette forme comporte plusieurs widgets : 3 GtkEntry permettant de placer un chiffre dans la grille sur la droite et 2 boutons, un permettant de confirmer les coordonnees dans les GtkEntry et l'autre confirmant la reconnaissance de chiffres.

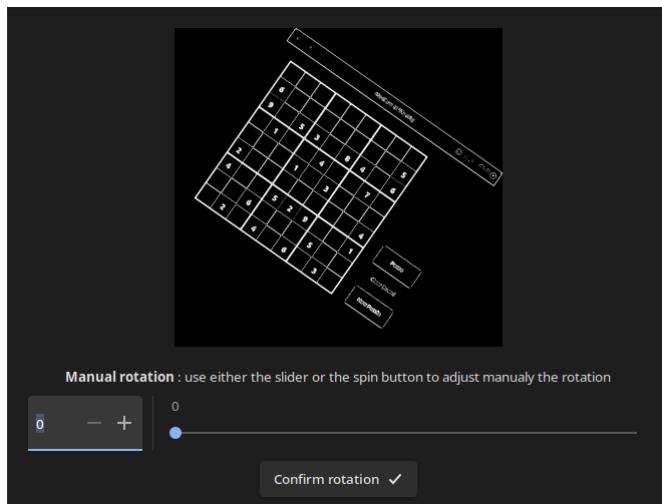


FIGURE 29 – Rotation manuelle

Cette forme comporte deux widgets : l'un permettant de specifier un angle particulier entre 0 et 360 degré soit en l'écrivant au clavier, soit en cliquant sur les '+' et '-' et l'autre permettant de tourner l'image à la volée, sans specifier d'angle.

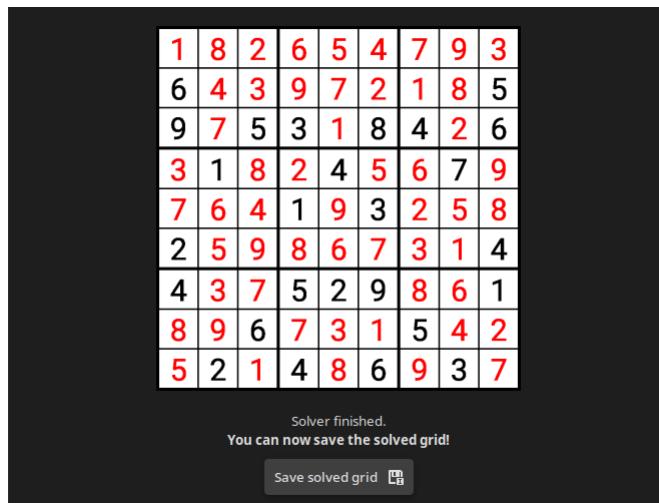


FIGURE 30 – Sauvegarde grille resolue

Cette forme comporte un seul bouton : celui-ci permet d'enregistrer le sudoku genere par l'ensemble des etapes.

2. Partie selection etapes

Cette partie regroupe l'ensemble des etapes de notre projet. Chaque etape possede un nom ecrit dans un bouton. Lorsque l'on clique sur ce bouton, la partie principale de l'UI change pour afficher les controles de ladite etape. Chaque bouton est disponible uniquement si l'utilisateur a deja termine l'etape que le bouton represente. Si jamais l'utilisateur a change d'image apres cette etape, l'ancienne image sera affichee. Il lui faudra recharger une nouvelle image s'il souhaite la changer. Il est donc possible de revenir en arriere a une etape sans relancer l'application.

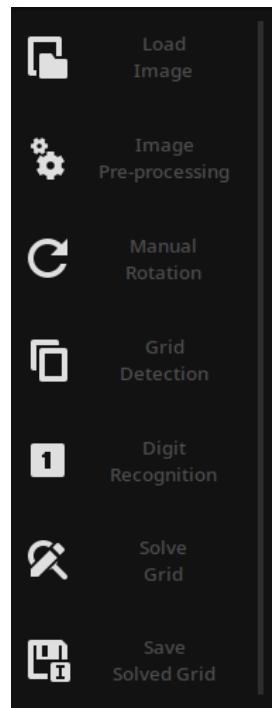


FIGURE 31 – Affichage etapes

Voici la signification de chaque bouton :

- **Load Image** : Cette etape permet de charger une image contenant un sudoku a analyser. Il est suppose que l'image ne provienne pas d'une etape posterieure a celle-ci, meme s'il est probable que la detection de grille fonctionne correctement.
- **Image Pre-Processing** : Cette etape permet d'appliquer l'ensemble des sous-etapes du pretraitemet de l'image. Ces sous-etapes peuvent etre appliquee soit une par une, soit toutes en un seul coup.
- **Manual Rotation** : Cette etape permet de tourner l'image manuellement. Meme si la detection de grille permet de tourner l'image automatiquement, il est possible qu'un utilisateur veuille l'orienter correctement lui-même.
- **Grid Detection** : Cette etape permet de detecter la grille dans l'image apres application des pre-traitements. Apres detection, elle permet l'auto-rotation de l'image ainsi que la correction de perspective.
- **Digit Recognition** : Cette etape permet de reconnaire les chiffres a l'intérieur de la grille detectee par la detection de grille. Pour les reconnaître, elle utilise notre reseau de neurones. Une fois les chiffres reconnus, elle permet de les modifier manuellement si jamais la detection s'est mal passee.

- **Solve Grid** : Cette etape permet de resoudre le sudoku genere a partir de la reconnaissance de chiffres. Elle regenere un sudoku avec les chiffres resolus en rouge.
 - **Save Solved Grid** : Cette etape permet de sauvegarder le sudoku resolu sous forme d'une image en ".png".
3. Partie sauvegarde, ouverture d'image et redemarrage application

Cette partie, situee en haut a gauche de l'UI, est destinee a la sauvegarde et au chargement d'image pendant une etape. Elle permet donc de sauvegarder chaque image generee par l'ensemble des etapes ou bien de charger une image correspondant a une certaine etape en particulier. De plus, elle permet de redemarrer l'application. Il est donc possible de charger une autre image avec un nouveau sudoku sans redemarrer l'application.



FIGURE 32 – Affichage sauvegarde, chargement et redemarrage

Voici la signification de chaque bouton :

- **Bouton reload** : Ce bouton permet de relancer l'application avec l'image originelle. Autrement dit, il permet de revenir a l'accueil.
- **Bouton chargement image** : Ce bouton permet, lorsqu'il est accessible, de remplacer l'image de l'etape actuelle par une quelconque autre image. Le deroulement des etapes ne sera pas modifie, seul le resultat pourra etre hasardeux en fonction du pretraitement de l'image chargee. Ce bouton n'est pas accessible pendant les sous-etapes, evitant une consommation de memoire trop importante.
- **Save As** : Ce bouton permet, lorsqu'il est accessible, de sauvegarder l'image actuelle sous un format ".png". Le deroulement des etapes ne sera pas modifie et les etapes manquantes seront toujours disponibles. Ce bouton n'est accessible uniquement lorsque l'UI affiche une image (autrement dit, uniquement lorsque l'UI contient une image a enregistrer).

5.8 Génération de grille

Que ce soit pour la derniere etape de notre projet ou bien pour la correction de la reconnaissance de chiffres, il nous faut generer une grille de sudoku propre, suffisamment grande et detaillee mais aussi lisible et comprehensible. Pour cela, nous utilisons 10 images :

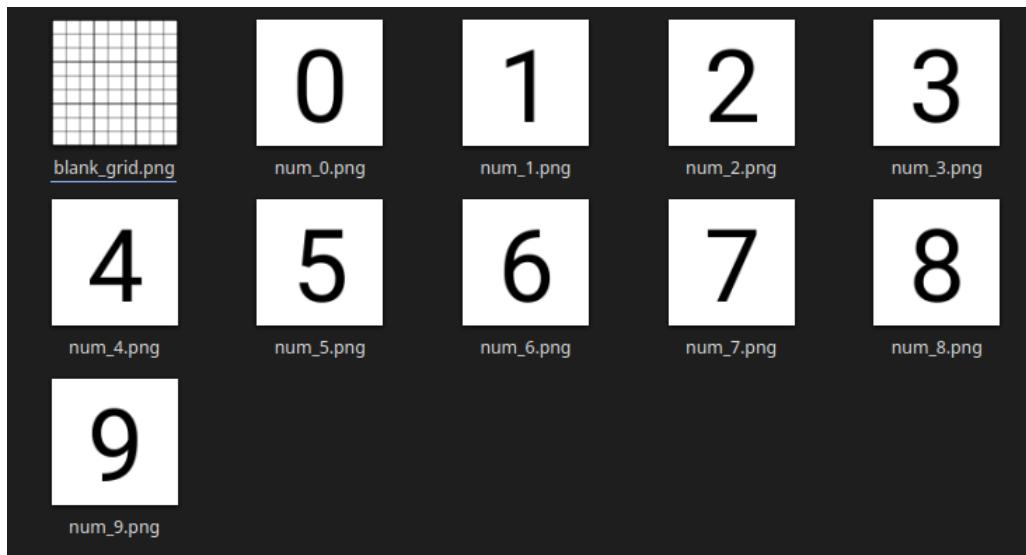


FIGURE 33 – Assets pour la generation de grille

La premiere consiste en une grille de sudoku 900x900pixels completement vide. Cette grille a ete cree avec un script python ecrit par nos soins. Les 9 autres sont des images 100x100pixels contenant chacune un numero entre 0 et 9. Ces numeros sont destines a etre places dans la grille de sudoku vide.

Nous nous generons, grace a ces images, ces deux grilles :

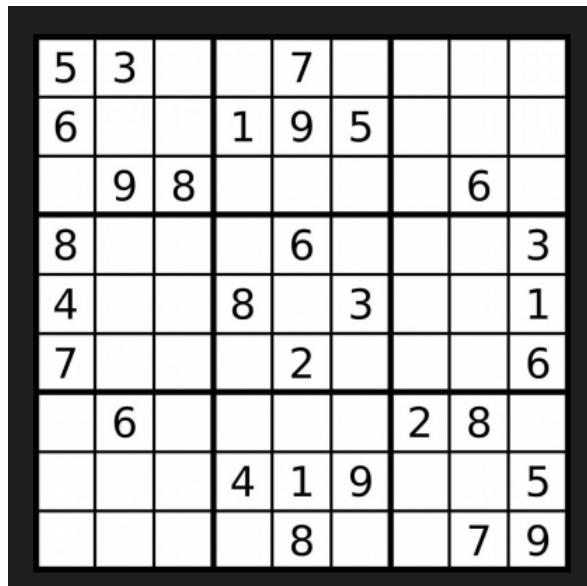


FIGURE 34 – Grille partiellement vide



FIGURE 35 – Grille resolue

La premiere grille possede certains trous : elle est destinee a etre utilisee par l'etape post reconnaissance de chiffres, pour pouvoir corriger cette reconnaissance.

La deuxieme grille possede certains chiffres en rouge : ce sont les chiffres ajoutes par le solver. La couleur rouge permet de mieux les distinguer.

Pour generer ces grilles, nous copions simplement les images contenant les chiffres entre 0 et 9 au coordonnees indiquees par le fichier ecrit par le sovler. Si un chiffre n'est pas present dans le fichier originel, soit il sera ecrit en rouge lors de l'etape finale, soit il ne sera pas ecrit lors de l'etape reconnaissance de chiffres.

6 Améliorations possibles

Nous arrivons à faire l'ensemble des étapes demandés dans le cahier des charges de l'OCR. Néanmoins, il y a toujours des améliorations possibles.

Par exemple, pouvoir reconnaître des chiffres manuscrits (en utilisant un dataset comme celui du MNIST contenant plus de 50000 images contenant les chiffres de 0 à 9 écrit par des milliers de personnes). Être capable de résoudre une grille de 16*16 ou encore développer différents moyens de communications pour notre projet comme un site web ou bien des réseaux sociaux.

De plus, les performances du réseau de neurones peuvent encore être amélioré notamment en utilisant une structure de données sous forme de matrices ou bien en utilisant la fonction d'activation Softmax.

En outre, la détection de grille est parfois perturbé par des lignes externes au Sudoku. Ainsi, un algorithme de Blob Detection par exemple pourrait permettre d'éviter

ce genre de désagrément.

7 Conclusion

Nous sommes très satisfait de notre projet. En effet, nous arrivons à résoudre un large éventail d'images dans un temps réduit et avec une interface utilisateur agréable.