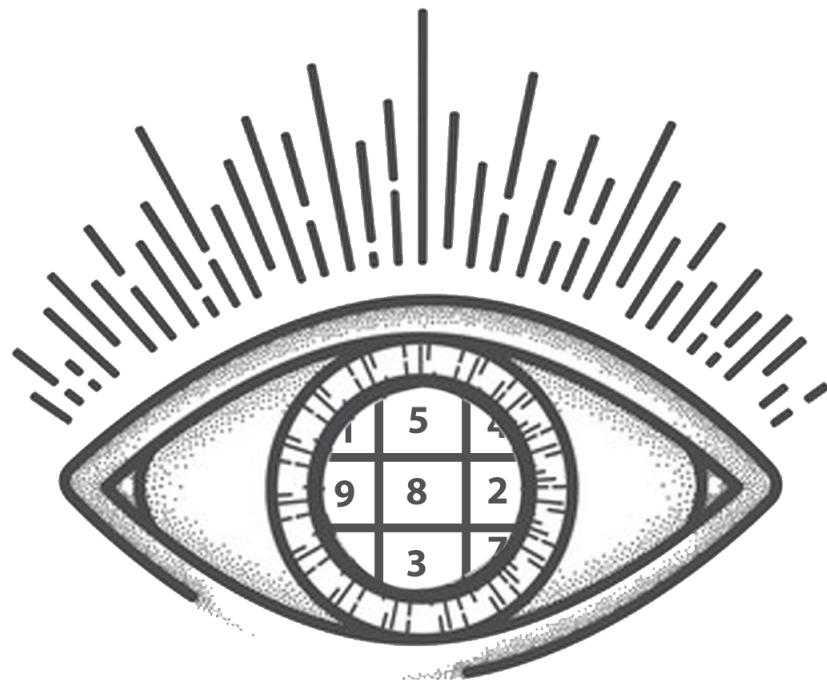


# Rapport soutenance Finale OCR

SUDO.C.R

Giovanni Le Bail | Kristen Quesnel | Tudual Lefeuvre | Aloïs Colléaux-Le Chêne

Octobre 2021



**SUDO.C.R**

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Qu'est ce qu'un OCR ? . . . . .	4
1.2	Qu'est ce que SUDO.C.R ? . . . . .	4
<b>2</b>	<b>Pré-traitement de l'image</b>	<b>5</b>
2.1	Règles générales du traitement d'image . . . . .	5
2.2	Mise en niveau de gris . . . . .	5
2.3	Réduction de bruit . . . . .	5
2.4	Flou linéaire . . . . .	6
2.4.1	Flou gaussien . . . . .	6
2.4.2	Filtre de Kuwahara . . . . .	6
2.5	Binarisation . . . . .	7
2.6	Détection des lignes . . . . .	8
2.7	Filtre de Canny . . . . .	9
2.7.1	Filtre de Sobel . . . . .	10
2.7.2	Suppression des non-maxima . . . . .	12
2.7.3	Double seuillage . . . . .	13
2.7.4	Hystérésis . . . . .	14
2.8	Opérations morphologiques . . . . .	15
2.9	Remplissage par diffusion (alias Pot de peinture) . . . . .	15
2.10	Détection du plus grand blob . . . . .	16
2.11	Détection des coins de la grille . . . . .	17
2.12	Rotation d'image . . . . .	17
2.13	Transformation de perspective . . . . .	20
2.14	Découpage de la grille . . . . .	23
<b>3</b>	<b>Interface Graphique</b>	<b>24</b>
3.1	Création des graphismes . . . . .	24
3.1.1	Choix de la librairie . . . . .	24
3.1.2	Glade . . . . .	24
3.2	Le code . . . . .	25
3.2.1	Définitions des éléments graphiques . . . . .	26
3.2.2	Le fonctionnement . . . . .	26
3.2.3	Chargement d'une Image . . . . .	26
3.3	Interface utilisateur - Utilisation . . . . .	27
3.4	Identité graphique . . . . .	28
<b>4</b>	<b>Réseau de neurones</b>	<b>29</b>
4.1	Le principe de fonctionnement . . . . .	29
4.2	Le principe d'entraînement du réseau de neurones . . . . .	30
4.2.1	Base de données . . . . .	32
<b>5</b>	<b>Le solveur de sudoku</b>	<b>34</b>
5.1	Le principe de fonctionnement . . . . .	34
5.2	Benchmarking . . . . .	34

---

<b>6 Mode d'emploi</b>	<b>34</b>
<b>7 Conclusion</b>	<b>36</b>
7.1 Réalisation aux soutenances . . . . .	36
7.1.1 1ère soutenance . . . . .	36
7.1.2 2ème soutenance . . . . .	37
7.2 Problèmes rencontrés . . . . .	37
<b>8 Bibliographie</b>	<b>37</b>

# 1 Introduction

Sécu Studio, se lance dans un nouveau projet, SUDO.C.R, "SUDO" faisant référence au sudoku, ainsi que la commande "sudo" sur Linux, "O.C.R" référence à "OCR" qui est une référence à notre projet. Pour ce nouveau projet pour le Sécu Studio, est toujours composé de ses 4 membres : Aloïs Colleaux–Le Chene, Tudual Lefeuvre, Kristen Quesnel et Giovanni Le Bail.

Dans le cadre de notre projet de 3e semestre, nous avions pour consigne de développer un programme, en utilisant le langage de programmation "C", qui est capable de reconnaître un sudoku sur une photographie et de retourner à l'utilisateur le sudoku résolu. Ce projet se déroule sur 4 mois durant lesquels nous devons repartir la grande charge de travail afin de finir ce projet dans les temps.

Nous avons décidé de découper ce projet en 4 parties :

- Interface graphique
- Prétraitement d'image
- Reconnaissance de caractères
- Solveur de Sudoku

## 1.1 Qu'est ce qu'un OCR ?

"OCR" qui vient de Optical Caractère Recognition, ou Reconnaissance optique de caractère en français, "désigne les procédés informatiques pour la traduction d'images de textes imprimés ou dactylographiés en fichiers de texte."<sup>1</sup>, il s'agit d'un logiciel permettant de reconnaître du texte sur une photographie, que ce soit des lettres ou des chiffres. Dans notre cas, nous devons seulement reconnaître les chiffres de 1 à 9.

## 1.2 Qu'est ce que SUDO.C.R ?

Pour notre projet, nous devons donc développer un logiciel composé de 4 parties différentes, tout en gardant une compatibilité entre nos différentes parties afin de permettre un échange d'informations fluide entre ces dernières.

Dans un premier temps, il conviendra d'appliquer différents filtres (rotations, niveau de gris, flous, binarisation...) sur l'image choisie par l'utilisateur grâce à l'interface graphique. Ces filtres auront pour objectif de normalisé les entrées utilisateurs pour les transmettre au réseau de neurones.

Le réseau de neurones, composé de 3 couches (une couche d'entrée, cachée, de sortie) a pour objectif d'être capable, suite à un entraînement, de donner une probabilité sur le chiffre qui lui a été donné en entré.

Une fois que le réseau de neurones à donner une estimation des chiffres présents sur la grille de sudoku, une grille de sudoku au format texte est créé, c'est cette grille qui sera envoyé à la partie du solveur de sudoku

Le solveur de sudoku aura pour objectif, via un algorithme de "backtracking", de résoudre rapidement le sudoku.

---

1. [https://fr.wikipedia.org/wiki/Reconnaissance\\_optique\\_de\\_caract%C3%A8res](https://fr.wikipedia.org/wiki/Reconnaissance_optique_de_caract%C3%A8res)

Une fois le sudoku résolu, notre programme reconstitue une grille de sudoku avec la solution trouvé par nos algorithmes, en vert.

Afin de permettre une utilisation ergonomique de notre logiciel, nous avons développé une interface graphique permettant à l'utilisateur d'appliquer les différents filtres à la main, où de choisir d'appliquer les filtres automatiquement.

## 2 Pré-traitement de l'image

### 2.1 Règles générales du traitement d'image

De manière générale, un algorithme de traitement d'image s'applique sur chaque pixel d'une image. Cela signifie que cet algorithme va passer de façon itérative sur tous les pixels de l'image et appliquer le même algorithme, qui va modifier les valeurs Rouge, Vert et Bleu (aussi appelées valeur RGB) de ce pixel. Dans les prochaines sections, lorsque nous parlerons d'un pixel, nous ferons référence à un pixel générique, qui sera appliqué à l'algorithme de traitement.

Il faut d'ailleurs noter que les pixels modifiés ne doivent pas interférer avec les pixels qui n'ont pas encore été modifiés par l'algorithme, ainsi pour certains algorithmes, il sera nécessaire de stocker les nouveaux pixels avant de pouvoir réellement modifier l'image. On utilisera ensuite la notation " $P$ " pour noter le pixel d'entrée de l'algorithme, et " $P'$ " le pixel de sortie de l'algorithme.

### 2.2 Mise en niveau de gris

La mise en niveau de gris d'une image est l'une des premières étapes à appliquer lors d'un traitement d'image. Elle consiste à "supprimer" les couleurs d'une image en faisant la moyenne des valeurs RGB pour chaque composante du pixel.

Ainsi, pour chaque pixel  $P$ , si l'on pose  $R, G, B$  (resp.  $R', G', B'$ ) les composantes RGB de  $P$  (resp.  $P'$ ), on a :

$$R' = G' = B' = \frac{1}{3}R + \frac{1}{3}G + \frac{1}{3}B$$

Il existe aussi une formule alternative représentant une vision plus réaliste pour l'œil humain où les coefficients ne sont pas tous égaux, mais le choix de l'une ou l'autre technique n'est pas important pour notre utilisation étant donné qu'il y a aura une étape de binarisation plus tard dans le processus.

L'étape de mise en niveau de gris est importante, car elle permet pour les algorithmes suivant de ne prendre en compte qu'une seule composante de couleur du pixel plutôt que les 3, simplifiant le développement des algorithmes.

### 2.3 Réduction de bruit

Le bruit d'une image signifie une fluctuation parasite et aléatoire de lumière ou de couleurs dans une image. Ce bruit est souvent présent lors de prise de photos et peut causer de nombreux problèmes pour plus tard. Cependant, il est très difficile de réduire le bruit entièrement, mais il est possible de le réduire un peu, au moins de façon à ce qu'il

soit moins impactant. Il existe plusieurs façons de réduire le bruit, et nous avons considéré trois d'entre elles pour notre programme.

## 2.4 Flou linéaire

Pour appliquer le flou linéaire, nous avons utilisé une technique répandue dans le traitement d'image : les matrices de convolution.

Une matrice de convolution est une matrice carrée, de taille impaire et souvent symétrique, avec des poids. Cette matrice sera centrée en un pixel et la nouvelle valeur de ce pixel sera la moyenne des pixels autour, multipliée par leur poids respectif. Cette nouvelle valeur est donc la nouvelle valeur du pixel. En suivant ce raisonnement, une approche naïve serait de faire la moyenne des pixels aux alentours. Cette approche se nomme le flou linéaire, elle a cependant un inconvénient non négligeable, les pixels éloignés du centre, sont considérées comme aussi important que les pixels plus centraux.

Ce problème est ainsi réglé par la technique suivante, le flou gaussien.

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

FIGURE 1 – Matrice linéaire de paramètre  $n = 3$

### 2.4.1 Flou gaussien

Le flou gaussien répartit les poids en fonction de leurs distance par rapport au pixel. Ainsi les poids plus aux centres de la matrice seront plus importants, tandis que ceux plus proches des bords et des coins seront bien moindres. L'avantage principal est que ce filtre préserve les bords des images.

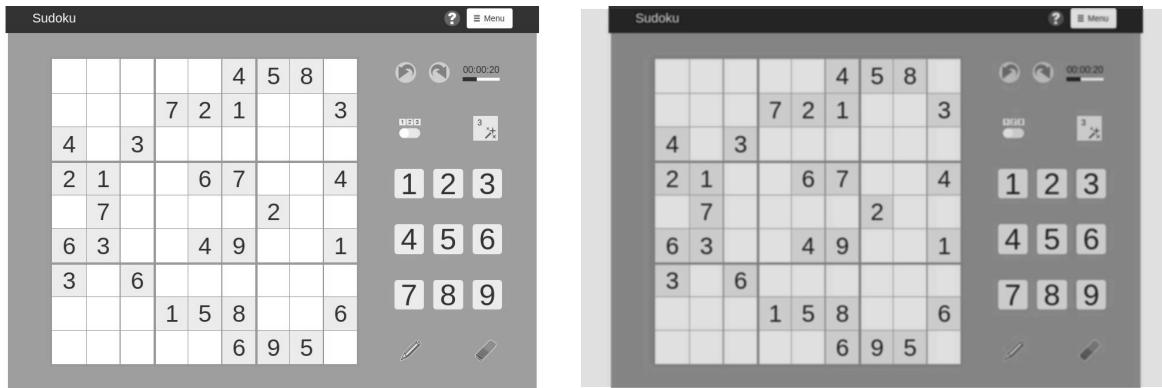


FIGURE 2 – Avant/Après application du flou gaussien

### 2.4.2 Filtre de Kuwahara

Bien que le flou gaussien est relativement efficace pour préserver les bords, le filtre de Kuwahara permet de lisser l'image, en préservant encore plus les bords. Supposons une

fenêtre carrée de taille  $2a + 1$  centrée autour d'un point  $(x, y)$  de l'image, et  $I(x, y)$ . Ce carré peut être divisé en quatre zones carrées plus petites  $Q_{i=1..4}$  tel que

$$Q_i(x, y) = \begin{cases} [x, x+a] \times [y, y+a] & \text{si } i = 1 \\ [x-a, x] \times [y, y+a] & \text{si } i = 2 \\ [x-a, x] \times [y-a, y] & \text{si } i = 3 \\ [x, x+a] \times [y-a, y] & \text{si } i = 4 \end{cases}$$

où  $\times$  est le produit cartésien entre deux ensembles.

Il est notable de remarquer qu'il y aura des pixels présents dans plusieurs zones à la fois. La moyenne  $m_i(x, y)$  et l'écart-type  $\sigma_i(x, y)$  des quatre régions sont calculés et sont utilisés pour déterminer la valeur du pixel central. La valeur du filtre de Kuwahara  $\Phi(x, y)$  pour chaque point  $(x, y)$  est donnée par  $\Phi(x, y) = m_i(x, y)$  où  $i = \arg \min_j \sigma_j(x, y)$

En d'autre termes, le pixel prendra la valeur de la moyenne de la zone la plus homogène.

a	a	a/b	b	b
a	a	a/b	b	b
a/c	a/c	a/b/ c/d	b/d	b/d
c	c	c/d	d	d
c	c	c/d	d	d

FIGURE 3 – Fenêtre utilisée par un filtre de Kuwahara

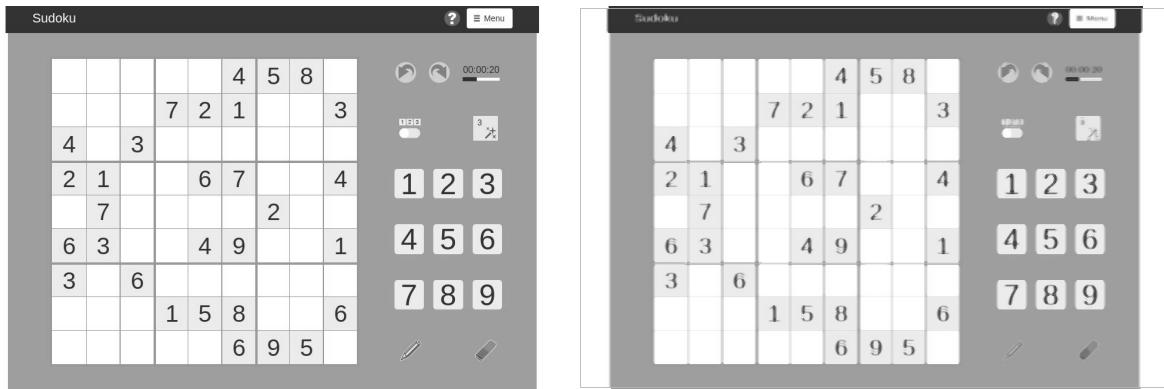


FIGURE 4 – Avant/Après application du filtre de Kuwahara

## 2.5 Binarisation

L'étape d'après la réduction de bruit est la binarisation. Cela signifie passer d'une image en nuance de gris à une image en noir et blanc. Cette étape est très utile car elle permet de

réduire encore plus l'information, et simplifie grandement les algorithmes d'après. Pour ce faire, nous avons utilisé une méthode de seuillage. Un seuillage, comme son nom l'indique, consiste à définir une certaine valeur  $s$  en tant que seuil. Tout pixel dépassant ce seuil deviendra un pixel blanc, tandis que tous les autres pixels deviendront noir. La méthode la plus basique consiste à définir un  $s$  constant pour toute l'image, il s'agit du *seuillage global* mais cette technique basique est peu efficace pour des images ayant des variances de luminosité. C'est pour cette raison que nous avons choisi la technique du *seuillage adaptatif*.

Le seuillage adaptatif fait varier le seuil  $s$  tel que  $s$  est égal à la moyenne des valeurs des pixels aux alentours, moins une constante  $C$ . Ainsi, les pixels plus sombres que leurs voisins seront mis en arrière-plan tandis que les pixels plus clairs ou de même intensité seront au premier plan.

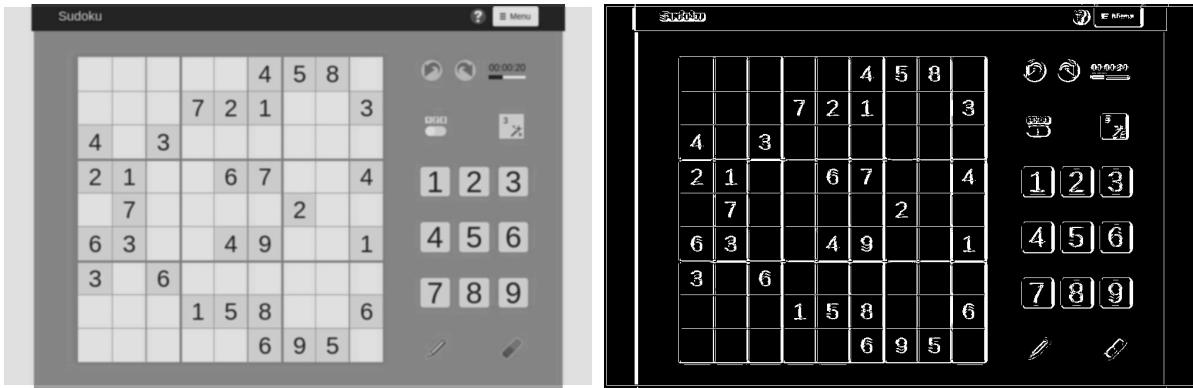


FIGURE 5 – Avant/Après application du seuillage adaptatif

## 2.6 Détection des lignes

La détection des lignes est une étape cruciale pour la suite du traitement d'image. Elle permet de détecter le plus gros polygone à quatre côtés, qui sera notre grille de sudoku (à noter qu'il ne s'agira pas forcément d'un carré bien orienté, mais peut-être d'une grille penchée, elle ne sera pas strictement carrée). Pour détecter les lignes, nous utilisons la technique de la *transformée de Hough*.

Pour rappel, il est possible d'écrire une droite sous forme paramétrique :

$$y = ax + b$$

, et chaque point de coordonnées blanc a des coordonnées  $(x, y)$ . Le principe de la transformée est sous forme de vote, chaque point blanc a donc certains  $a$  et  $b$  qui valident l'équation  $y = ax + b$ , ce point-là va donc voter pour chacun des ses points. Et ainsi de suite pour chaque points blancs. Il est ensuite possible de récupérer les couples  $(a, b)$  ayant récupéré le plus de votes. Ce seront les lignes de notre image.

La seule différence avec l'application pratique de l'algorithme de Hough est qu'il ne s'agit pas de couple  $(a, b)$ , tel que  $y = ax + b$  mais de couple  $(\rho, \theta)$ , tel que  $\rho = x \cos(\theta) + y \sin(\theta)$ . En effet, ce changement est nécessaire car il est impossible de représenter des droites horizontales avec le couple  $(a, b)$ , cela aurait demandé  $a = +\infty$  et c'est impossible.

Ci-dessous, un exemple de l'espace des couples  $(\rho, \theta)$ . Plus un point est blanc, plus il a reçu de vote.

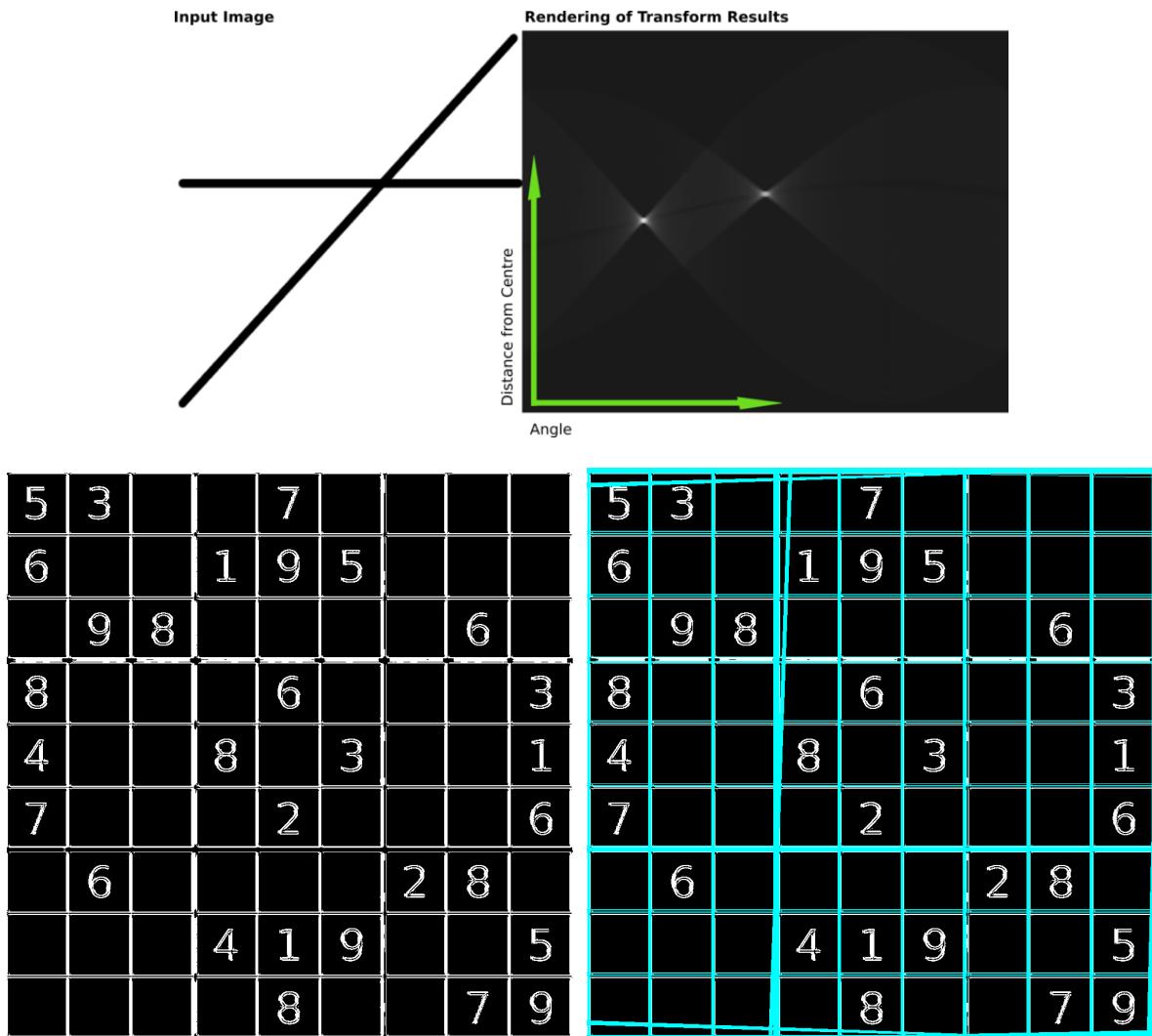


FIGURE 6 – Avant/Après application de la transformée de Hough

## 2.7 Filtre de Canny

Le filtre de Canny est une méthode avancée de détection de grille. Ce filtre pourra remplacer le seuillage adaptatif afin de simplifier la détection des lignes de la transformée de Hough. Ce filtre est une amélioration d'un autre filtre (Filtre de Sobel), en appliquant d'autres étapes après pour minimiser le bruit et mettre en évidence les lignes. Ces étapes sont :

- Filtre de Sobel
- Suppression des non-maxima
- Double seuillage
- Hystérésis

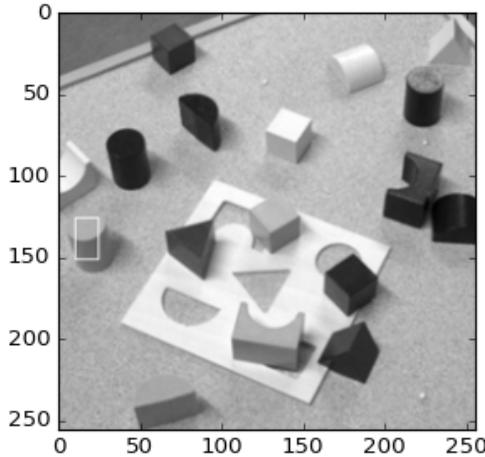


FIGURE 7 – Image de départ utilisée pour la démonstration du filtre

### 2.7.1 Filtre de Sobel

Si l'on considère une image comme un filtre continu à 2 dimensions, il s'agit de calculer le gradient

$$\overrightarrow{\text{grad}} v = \frac{\delta v}{\delta x} \vec{e}_x + \frac{\delta v}{\delta y} \vec{e}_y$$

Ainsi, les bords de l'image sont ainsi les endroits  $(x, y)$  où la norme du gradient est la plus élevée. De plus le vecteur gradient sera perpendiculaire au bord.

Pour calculer une approximation du gradient, il est possible d'appliquer un opérateur de différentiation. Soit  $V_{i,j}$  la valeur du pixel  $(i, j)$ . Une approximation de la dérivée par rapport à  $x$  peut être obtenue par la différence

$$V_{i+1,j} - V_{i-1,j}$$

En terme de filtrage, cela peut se traduire par le masque de convolution

$$(-1 \quad 0 \quad 1)$$

Il est cependant préférable d'appliquer un masque carré 3 x 3. Pour cela, l'opérateur optimal est l'opérateur de Sobel

$$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

De même, nous avons l'opérateur de Sobel pour l'axe y :

$$S_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

Ainsi, nous obtenons deux images  $G_x$  et  $G_y$ , en faisant une convolution de l'image avec les masques  $S_x$  et  $S_y$ .

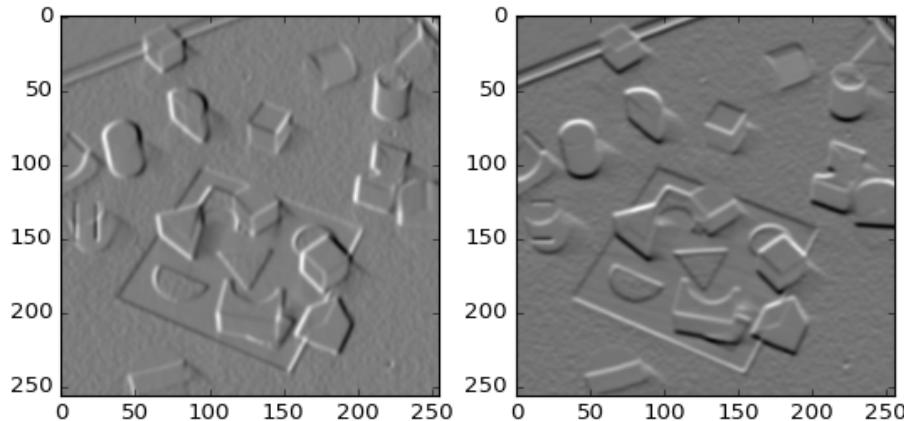


FIGURE 8 – Représentation des convolutions des masques  $S_x$  et  $S_y$

La seconde étape consiste à calculer le gradient, et l'angle par rapport à la direction x :

$$G = \|\overrightarrow{\text{grad}} v\|$$

$$\theta = (\vec{e}_x, \overrightarrow{\text{grad}} v)$$

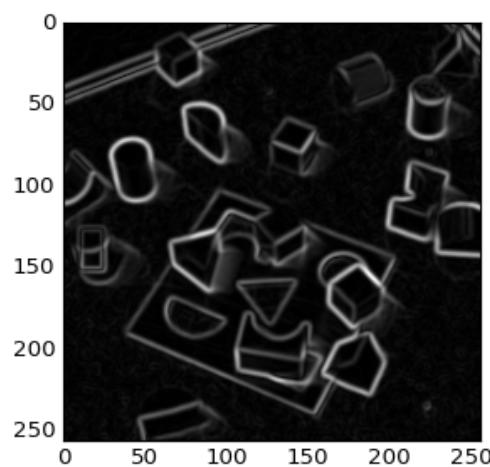
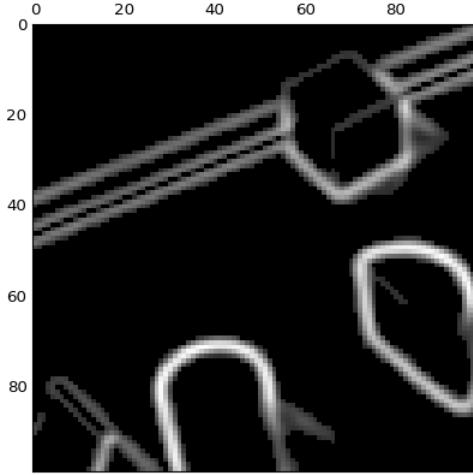


FIGURE 9 – Résultat de l'application du filtre de Sobel  $G$

### 2.7.2 Suppression des non-maxima

Observons un détail de l'image.



On remarque les bords sont épais de plusieurs pixels. Ici les bords sont épais d'environ 3 pixels mais ils peuvent être beaucoup importants si l'image a subi un flou important ou si du bruit est trop présent. Ainsi, nous voulons conserver uniquement le pixel du bord correspond où le gradient est maximal.

La figure suivante représente un point  $(i, j)$  entouré de ses 8 voisins. Le bord en ce point, représenté par le trait plein, est perpendiculaire au gradient. On a aussi représenté les 2 points  $M_1$  et  $M_2$  situé de part et d'autre du bord.

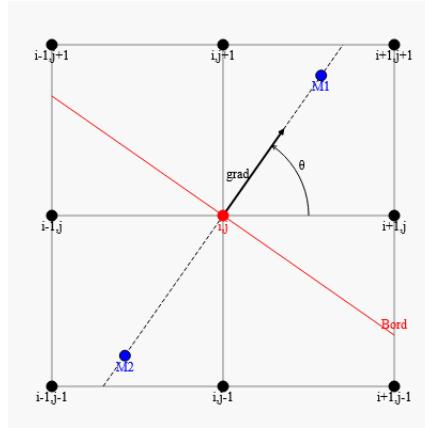


FIGURE 10 – Représentation d'un pixel bord avec deux autre pixels bords

En rapprochant les points  $M_1$  et  $M_2$  des pixels voisins par interpolation linéaire, il est possible de comparer les gradients. Si le gradient en  $(i, j)$  est supérieur aux gradients en  $M_1$  et  $M_2$ , on le conserve. Sinon, on l'élimine. On procède ainsi pour tout les pixels bords de l'image, ainsi les bords ne seront large d'un pixel uniquement

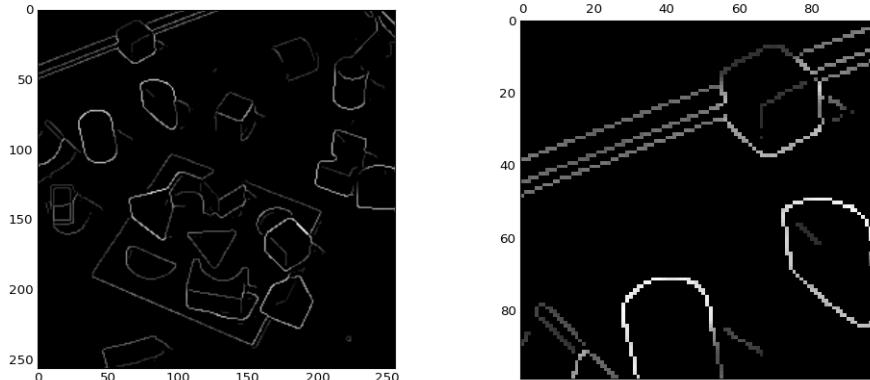


FIGURE 11 – Résultat de la suppression des non-maxima

### 2.7.3 Double seuillage

Le double seuillage a comme objectif de catégoriser tous les pixels en 3 catégories :

- Les pixels forts sont ceux ayant une intensité assez forte pour être certain que ce pixel fait partie des bords
- Les pixels faible n'ont pas une intensité assez élevée pour être certain qu'ils sont un bord, ni assez faible pour être certains qu'ils n'en sont pas
- Les autre pixels qui ont une intensité assez faible pour être considéré comme non-bord avec certitude

Ainsi nous aurons deux seuils. Le seuil haut, et le seuil bas. En comparant chacun des pixels à ces deux seuils, nous allons ensuite pouvoir les catégoriser dans l'une des trois catégories susmentionnées. Si l'intensité du pixel est supérieure au seuil haut, le pixel est fort. Si le pixel est entre le seuil haut et le seuil bas, c'est un pixel faible et sera traité durant l'étape d'hystérésis. Sinon, l'intensité du pixel est inférieure au seuil bas, c'est un pixel autre.

Dans la figure suivante, nous pouvons voir les deux types de bords, en blanc les pixels forts, en gris, les pixels faible.

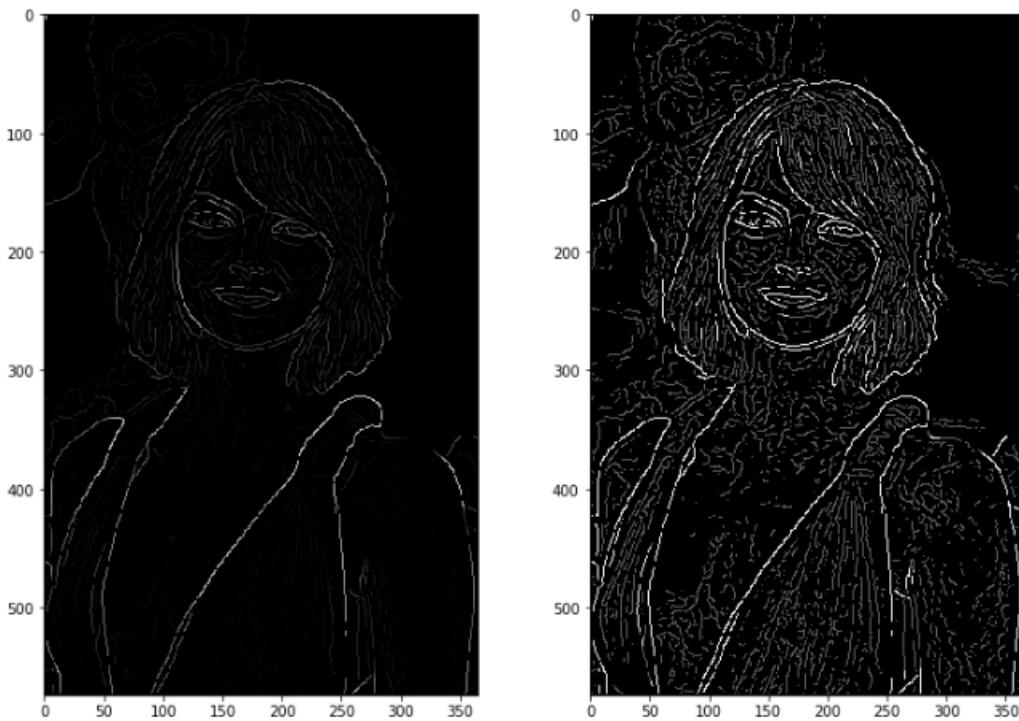
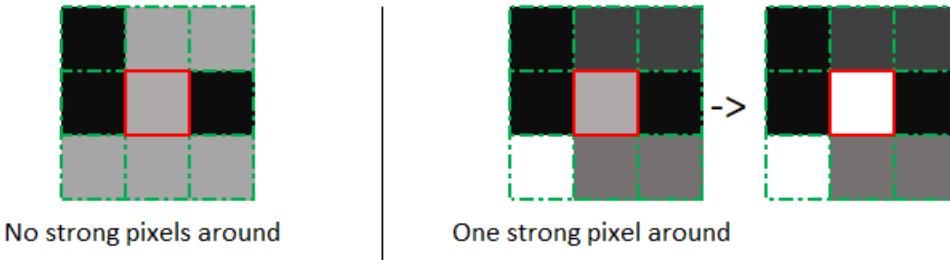


FIGURE 12 – Résultat du double seuillage.

#### 2.7.4 Hystérésis

L'hystérésis consiste à convertir chaque pixel faible en pixel fort si et seulement si il entouré d'au moins un pixel fort. Sinon, il est éliminé.



Ainsi, chaque pixel faible sera converti en pixel fort, ou éliminé. On se retrouve avec une image binarisée, et avec des bords large d'un pixel uniquement.

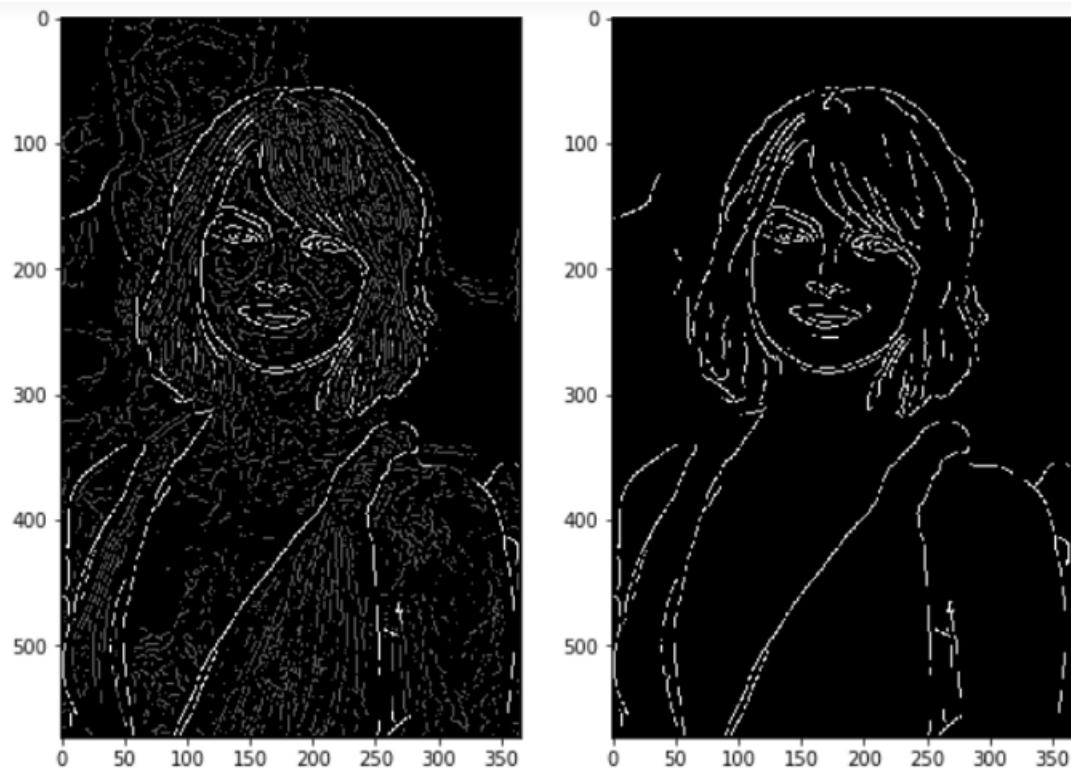


FIGURE 13 – Résultat de l'hystérésis. Résultat final du filtre de Canny

## 2.8 Opérations morphologiques

On distingue 2 types d'opérations morphologiques de base : l'érosion et la dilatation ; C'est deux techniques ce complètent. L'érosion est un processus qui permet de séparer des objets qui sont collés, il supprimer les aspérités. La Dilatation est un processus qui permet de réparer des traits interrompus, La Dilatation permet de combler les trous. La combinaison des ces opérations dans des ordres différents permet d'améliorer l'image.



FIGURE 14 – Érosion et dilatation

## 2.9 Remplissage par diffusion (alias Pot de peinture)

L'algorithme de remplissage par diffusion permet de remplir une zone contiguë de pixels de même pixels. L'algorithme utilisé est relativement simple et peut être exprimé dans le pseudo-code suivant :

```
remplissage(pixel, couleurSource, couleurDestination)
```

```

début
    si couleur(pixel) == couleurSource
    alors
        couleur(pixel) = couleurDestination
        remplissage4(pixel au nord,    couleurSource, couleurDestination)
        remplissage4(pixel au sud,    couleurSource, couleurDestination)
        remplissage4(pixel à l'est,   couleurSource, couleurDestination)
        remplissage4(pixel à l'ouest,  couleurSource, couleurDestination)
    fin si
fin

```

## Optimisation

Pour éviter les erreurs de dépassement de pile due à la récursion, il est judicieux de déplacer la récursion dans une structure de données annexe telle qu'une file. Ce qui est ce que nous avons fais ici. La fonction devient donc itérative et peux s'exprimer ainsi :

```

remplissage(pixel, couleurSource, couleurDestination)
début
    Soit q, une file
    enfiler pixel
    tant que q n'est pas vide
        On défile q dans le pixel n
        si couleur(pixel) == couleurSource
        alors
            couleur(pixel) = couleurDestination
            enfiler le pixel au nord dans q
            enfiler le pixel au sud dans q
            enfiler le pixel à l'est dans q
            enfiler le pixel à l'ouest dans q
        fin si
    fin tant que
fin

```

Cet algorithme, en plus d'être utilisé pour l'outil "pot de peinture" dans des programmes de dessins ou autre, permet de compter le nombre de pixels dans une zone continue (aussi appelé blob).

### 2.10 Détection du plus grand blob

En utilisant le remplissage par diffusion, il est possible de marquer tous les blobs blancs et d'enregistrer leurs tailles. Ainsi on peut enregistrer le plus grand blob, que l'on suppose être la grille, et éliminer tout les autres blobs restant.

On se retrouve ainsi avec un seul blob restant. Ce blob est la grille.

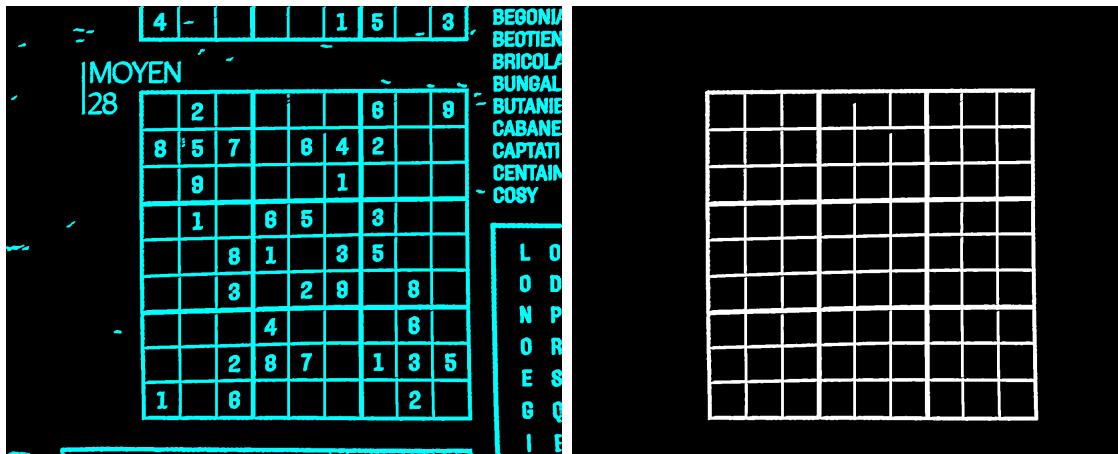


FIGURE 15 – Représentation des blobs blancs, et résultat de la détection du plus grand blob

## 2.11 Détection des coins de la grille

Pour l'orientation automatique et la transformation homomorphe, il est nécessaire de détecter les coins de la grille. À noter qu'une détection exacte n'est pas nécessaire, une approximation est suffisante.

Nous allons utiliser la détection du plus grand blob pour détecter les coins. Il est possible de détecter les coins en utilisant la propriété suivante :

Soit  $ul$ ,  $ur$ ,  $ll$  et  $lr$  respectivement les coins supérieur gauche, supérieur droit, inférieur gauche et inférieur droit de la grille.

- $ul$  est le point  $(x, y)$  où la somme  $x + y$  est la plus faible
- $lr$  est le point  $(x, y)$  où la somme  $x + y$  est la plus forte
- $ll$  est le point  $(x, y)$  où la différence  $x - y$  est la plus faible
- $ur$  est le point  $(x, y)$  où la différence  $x - y$  est la plus forte

Il est possible de comprendre cette propriété en remarquant que :

- Le coin en haut à gauche aux coordonnées  $(0, 0)$  a pour somme  $0 + 0$
- Le coin en bas à droite aux coordonnées  $(w, h)$  a la somme  $w + h$  maximale
- Le coin en bas à gauche aux coordonnées  $(0, h)$  a la différence  $0 - h$  la plus faible
- Le coin en haut à droite aux coordonnées  $(w, 0)$  a la différence  $w - 0$  la plus forte

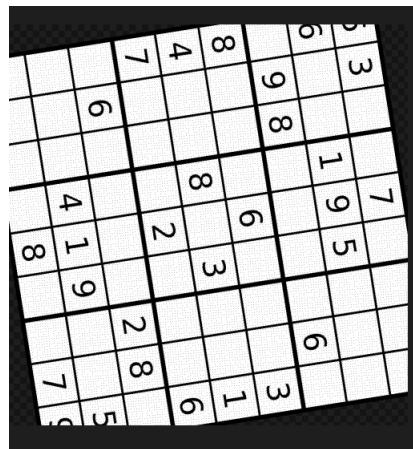
Tout les autres points ont des coordonnées entre ces extrêmes, et ces propriétés restent vraies (approximativement) au fur et à mesure que l'on se rapproche des 4 coins de la grille.

## 2.12 Rotation d'image

Il existe plusieurs méthodes de rotation d'image. Nous avons opté pour la plus simple et efficace. Nous avons tout d'abord implémenté une méthode simple qui consiste à faire la rotation avec une seule matrice :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Le problème étant que nous avions des pertes de qualités de l'image, car certains pixels n'existaient plus. Ils étaient hors de l'image, ce qui est dû à une simplification et donnait donc une imprécision de la formule.

FIGURE 16 – Rotation de  $45^\circ$ 

Nous avons utilisé le shearing qui consiste à prendre chaque pixel de l'image puis à appliquer 3 multiplications successives par des matrices dite de shearing. Le shearing est une transformation qui va consister à décaler chaque point de l'image en fonction d'un certain angle.

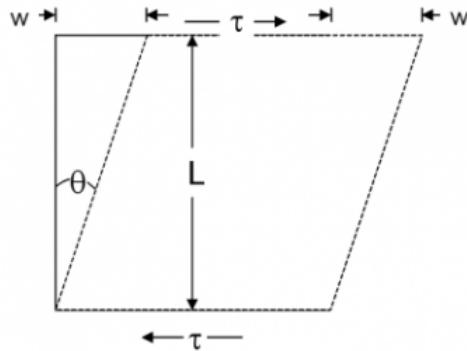


FIGURE 17 – Illustration du shearing

Notre rotation fait donc 3 shearing consécutif pour avoir une perte de qualité minimum

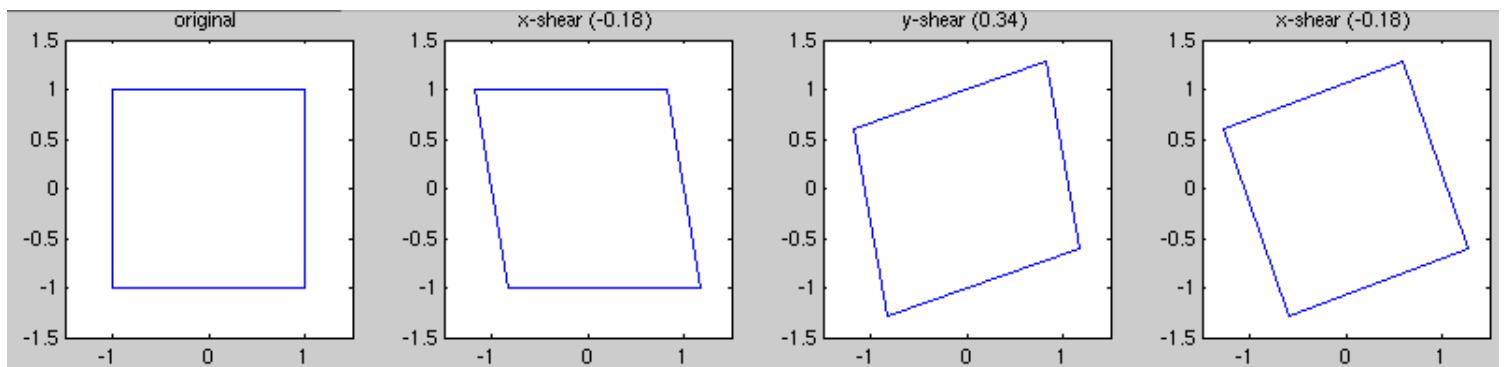


FIGURE 18 – Rotation par shearing

Nous avons donc le calcul suivant avec  $\theta$  est notre angle en radian :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & -\tan(\theta/2) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \sin(\theta) & 1 \end{pmatrix} \begin{pmatrix} 1 & -\tan(\theta/2) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Après avoir calculé nos nouvelles coordonnées, il nous faut leurs ajouter un décalage pour pouvoir replacer bien l'image en son centre.



La perte de qualité après rotation est minime cependant, il y a un peu de crénelage (c'est-à-dire qu'il y a des sortes d'escalier) sur l'image. Mais cela n'empêche en rien la détection de ligne. Ce qui est bien mieux que la première fonction de rotation où nous avions de la perte qui pouvait occasionner une gêne dans la détection des lignes.

## 2.13 Transformation de perspective

La transformation perspective ou homographic transform en anglais, va nous permettre de "normaliser" notre grille, cet à dire la remettre aux bonnes dimensions, la "redresser" , si l'angle de la photo originale n'est pas correct, et la tourner en même temps.

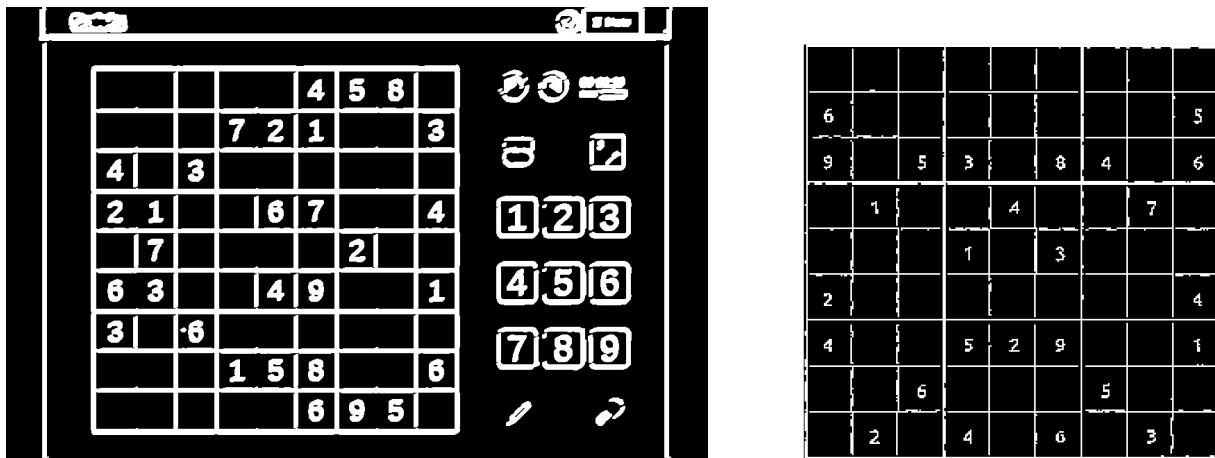


FIGURE 19 – Illustration de la transphormée homographique

Pour appliquer cette transformation, nous devons d'abord trouver les coefficients de la matrice de transformation.

En effet pour chaque transformation, nous avons besoin d'une matrice de taille  $3 \times 3$  ainsi pour appliquer une transformation linéaire que ce soit une rotation ou une réduction d'échelle.

Soient  $(x, y)$  les coordonnées d'un coin de l'image d'arrivée,  $(x', y')$  les coordonnées d'un coin l'image de départ et  $a, b, c, d, e, f, g, h$  nos coefficient de matrice, nous avons le calcul suivant :

$$\begin{pmatrix} x' \\ y' \\ w \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Où  $w = gx + hy + 1$ , le problème étant qu'on ne veut pas le garder, car notre image est en 2 dimensions alors nous allons réécrire notre matrice pour faire apparaître un 1 à la place de  $w$  dans notre matrice de fin.

En remaniant l'équation, nous avons cette expression :

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \frac{\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}}{(g \ h \ 1) \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}}$$

On peut donc déduire les coordonnées de  $x'$  et  $y'$  :

$$x' = \frac{ax + by + c}{gx + hy + 1}$$

$$y' = \frac{dx + ey + f}{gx + hy + 1}$$

Maintenant la prochaine étape est de déterminer nos 8 inconnues  $a, b, c, d, e, f, g, h$ . Pour ce faire, nous allons modifier l'expression de  $x'$  et  $y'$  :

On commence par multiplier par le dénominateur de chaque côté pour retirer la fraction :

$$x'(gx + hy + 1) = ax + by + c$$

$$y'(gx + hy + 1) = dx + ey + f$$

On va distribuer  $x'$  et  $y'$  :

$$x'gx + x'hy + x' = ax + by + c$$

$$y'gx + y'hy + y' = dx + ey + f$$

On isole  $x'$  et  $y'$  ce qui nous donne :

$$x' = ax + by + c - x'gx - x'hy$$

$$y' = dx + ey + f - y'gx - y'hy$$

On remarque que cela ressemble à un produit d'une matrice par un vecteur si on ajoute tous les termes :

$$x' = ax + by + c + 0d + 0e + 0f - x'gx - x'hy$$

$$y' = 0a + 0b + 0c + xd + ye + f - y'gx - y'hy$$

Nous avons donc ces matrices :

$$\begin{pmatrix} x & y & 1 & 0 & 0 & 0 & -x'x & -x'y \\ 0 & 0 & 0 & x & y & 1 & -y'x & -y'y \\ & & & & & \vdots & & \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ \vdots \end{pmatrix}$$

Ici, nous pouvons utiliser une unique matrice pour calculer les 4 coins de notre image en même temps. Ainsi, pour calculer les coefficients, nous pouvons utiliser cette propriété :

Soit  $B$  La matrice contenant les coordonnée de nos coins,  $\lambda$  la matrice contenant nos coefficient et  $A$  l'autre matrice.

Nous avons :

$$\begin{aligned} A\lambda &= B \\ A^T A\lambda &= A^T B \\ \lambda &= (A^T A)^{-1} A^T B \end{aligned}$$

Une fois nos coefficients calculés, il ne nous reste qu'à appliquer la matrice à chaque pixel de l'image.

## 2.14 Découpage de la grille

Après la transformation homographique, la grille est bien orientée et bien découpée. Il est donc possible de découper l'image en  $9 \times 9 = 81$  carrés en partitionnant l'image. Ces carrés contiendront, ou non, les chiffres. Il est donc nécessaire de détecter si ces cases sont vides, ou si elles ont un chiffre.

Pour cela, nous supprimons la grille de sudoku avant la transformation homographique (en utilisant la fonction de remplissage par diffusion). Elle n'est plus nécessaire maintenant que nous avons les coins de la grille.

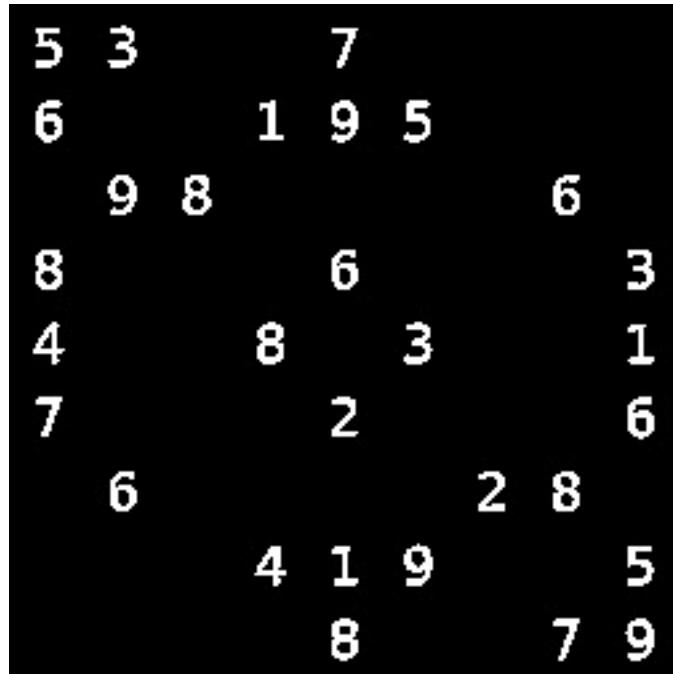
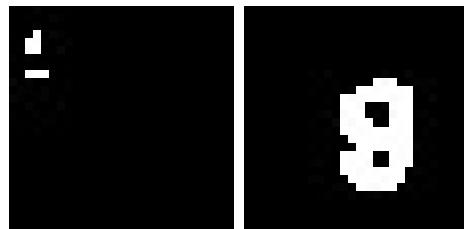


FIGURE 20 – Véritable résultat de la transformation homographique

Ainsi, une fois les carrés découpés, nous aurons deux cas, avec plus ou moins de bruit dans l'image.



Pour éviter le bruit, nous commençons par garder uniquement le plus grand blob blanc. Ce sera ainsi, soit le chiffre, soit du bruit aléatoire. Pour différencier ces deux cas, nous pouvons faire un remplissage par diffusion en couleur source noir à partir du centre de l'image. On suppose que le chiffre, s'il existe, est au centre de la case. Ainsi, si le blob de couleur noire au centre est assez petit ou n'existe pas (pixel de couleur blanc au centre), la case contient un chiffre. Si le blob noir est dépassé une certaine taille (posée arbitrairement à 20 pixels), on considère qu'il s'agit d'une case vide.

Notons que cette méthode de choix est probablement plus stricte que nécessaire, nous avons fait le choix de préférer des faux négatifs, plutôt que des faux positifs qui aurait eu des conséquences plus grave dans la résolution du sudoku (possibilité d'avoir une grille de sudoku insolvable).

## 3 Interface Graphique

### 3.1 Création des graphismes

Afin de permettre une utilisation à tous et pour tous, nous avons dû réaliser une interface graphique permettant d'utiliser notre programme autrement que via la console.

#### 3.1.1 Choix de la librairie

Comme le langage de programmation imposé était le C, et que pour réaliser une interface graphique dans ce langage nous avons besoin de *librairies*, nous devions faire un choix sur quelle librairie choisir.

Une librairie est un ensemble de fonctions de programmation, permettant d'effectuer un groupe de tâches précises. Par exemple, en C, il existe la librairie *stdio*, qui est une librairie utilisée dans tous les programmes permettant en autre d'afficher du texte, ou de récupérer une entrée utilisateur.

Nous devions donc trouver une librairie permettant de créer facilement une interface graphique. Après plusieurs recherches, nous sommes tombés sur 2 candidats intéressants. De plus, il fallait que ces librairies soient présentes sur les ordinateurs de l'école, ce qui est le cas.

La première librairie, étant *SDL* pour *Simple DirectMedia Layer*, cette librairie sortie en 1998, et continuellement mise à jour, est une bonne librairie graphique qui a permis depuis longtemps de créer différents programmes comme des jeux vidéo, des logiciels de traitement d'images...

La seconde librairie, étant *GTK* qui est plus précisément un ensemble de librairie, sorti aussi en 1998. GTK a été notamment utilisé dans le projet *GNOOME*, qui permet une interface graphique à un grand nombre de distribution Linux.

Nous avons choisi GTK, particulièrement la version GTK3+, puisque nous avons aussi utilisé le logiciel Glade, qui ne permet pas d'utiliser la SDL. De plus, GTK est bien plus facile à utiliser que SDL pour créer une interface graphique, bien que SDL permette plus de personnalisation.

#### 3.1.2 Glade

Pour nous aider dans notre tâche, le logiciel Glade nous sera grandement utile, ce logiciel permet de créer des objets (fenêtres, boutons, textes...) de manière graphique ainsi que de choisir leurs propriétés (taille, position, couleur...).

Une fois que nous avons créé les différents éléments de notre interface graphique, nous devons créer les "signaux", lorsque l'on clique sur un bouton par exemple. Pour ce faire, Glade permet d'associer un nom de fonction à une action de l'utilisateur sur un élément, comme dans cet exemple :

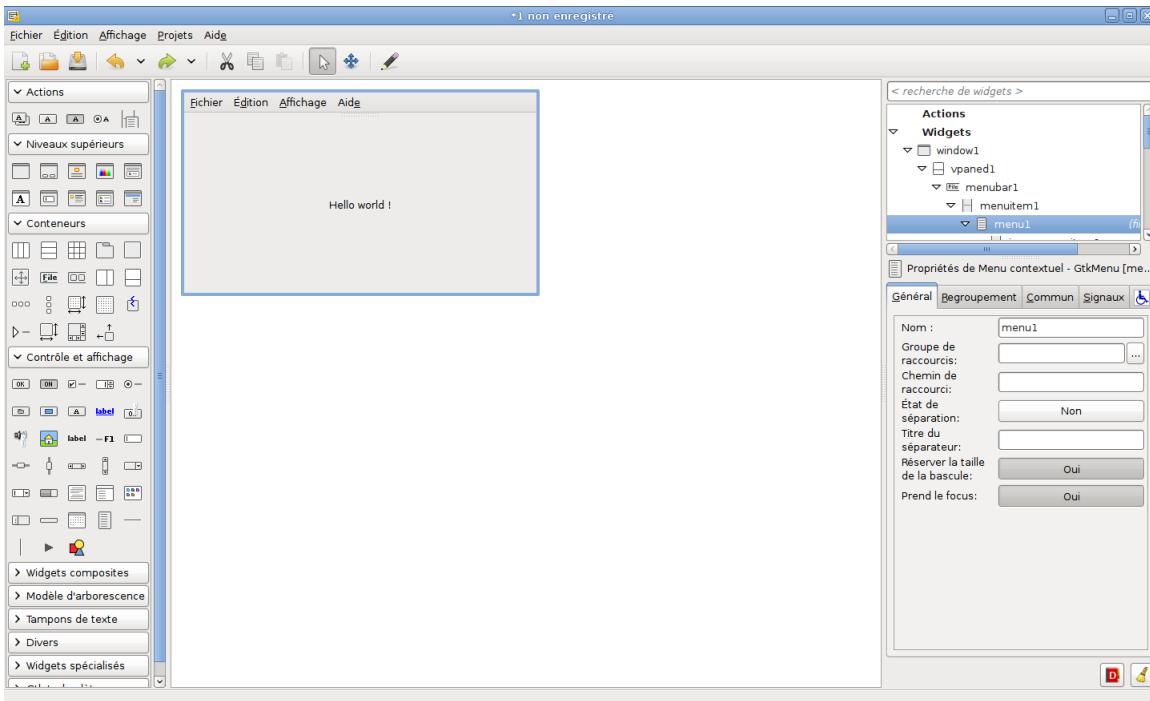


FIGURE 21 – Interface de Glade

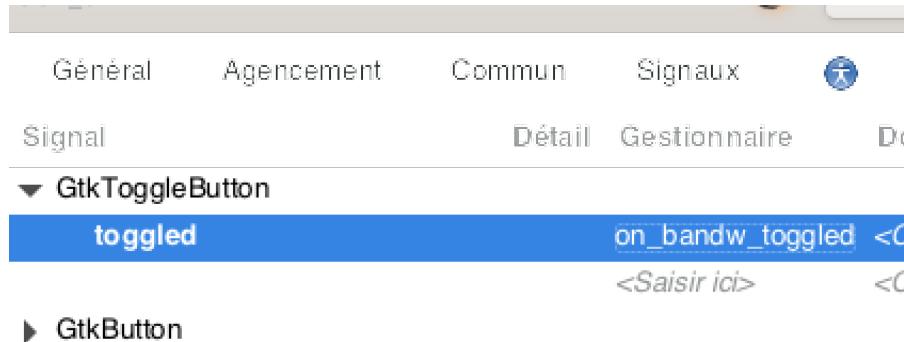


FIGURE 22 – Interface "signaux" de Glade

Lorsque l'utilisateur va cocher le bouton "noir et blanc" la fonction "on\_bandw\_toggled" sera exécuter.

Lorsque les éléments sont présents, que leurs propriétés ont été fixés, que les signaux associés par rapports aux actions utilisateur, on été défini, nous pouvons quitter Glade. Glade nous fournit un fichier avec toutes les informations au format ".glade", qui est en réalité un fichier au format ".xml".

### 3.2 Le code

Maintenant, nous devons associer toutes ces informations aux différentes parties du programme, ainsi que de créer les différentes fonctions pour les signaux, et les éléments graphiques.

### 3.2.1 Définitions des éléments graphiques

Dans un premier temps, nous allons créer un "builder" GTK à partir du fichier généré par Glade, ce builder va nous permettre par la suite de définir les différents éléments. Pour définir les différents éléments graphiques, nous avons besoin de créer pour chaque élément, un "GtkWidget". Ce qui va permettre au programme d'afficher l'interface graphique, grâce aux propriétés présentes dans le builder. Par la suite, il faut définir les fonctions correspondantes à nos différents signaux, ce qui revient à définir une fonction classique dans notre programme. Par conséquent, pour chaque signal, il y aura une fonction associé.

### 3.2.2 Le fonctionnement

Certaines actions qui peuvent être très simple pour l'utilisateur, peut être compliqué à implémenter.

Par exemple, le fait de permettre à l'utilisateur de choisir une image parmi les différents fichiers de son ordinateur.

### 3.2.3 Chargement d'une Image

Pour charger une image, nous avons créé un objet image dans Glade, puis nous avons défini l'objet dans notre code. Pour charger l'image, il suffit de charger l'image grâce à une "Surface SDL" qui appartient à la librairie "SDL\_Image", qui permet grâce à la fonction "IMG\_Load(char \*path)" de créer une "Surface SDL" depuis le chemin de l'image sur l'ordinateur.

L'un des défis a été de trouver comment permettre à l'utilisateur de choisir parmi ses fichiers une photo à choisir comme entré pour notre programme.

Pour ce faire, il faut définir un "gtk\_file\_chooser" qui est un type de variable appartenant à la librairie GTK, ce qui permet de définir une boîte de dialogue qui permet à l'utilisateur de choisir un fichier parmi ses fichiers.

Maintenant, que nous avons l'image, créer grâce au chemin du fichier, nous devons nous demander comment appliquer les différents calques sur la photo. Pour ce faire, nous utilisons des "Surface SDL" pour communiquer entre les différentes parties de notre programme.

Par conséquent, lorsque l'utilisateur coche une option, l'interface graphique va appeler la fonction qui applique le filtre, sauvegarder le résultat et l'afficher pour l'utilisateur.

Un autre problème rencontré a été à cause du filtre de rotation, l'interface graphique permet via des boutons d'effectuer une rotation horaire ou anti-horaire à l'utilisateur. Néanmoins, nous avons été confrontés à un problème, lorsque nous effectuons plusieurs rotations à la suite la photo était partiellement dégradée. Ce qui était dû au premier système d'application de filtres. Dans un premier lieu, nous effectuons les filtres à la suite sur une image temporaire, ce qui permettait d'effectuer plusieurs filtres à la suite. Ce système ne fonctionnait pas sur la rotation, puisque nous devions effectuer une rotation de X fois sur l'image de départ, et non des rotations à la suite ce qui dégradait la qualité de l'image.

Pour palier à ce problème, nous avons ajouté un accumulateur qui compte +1 pour une

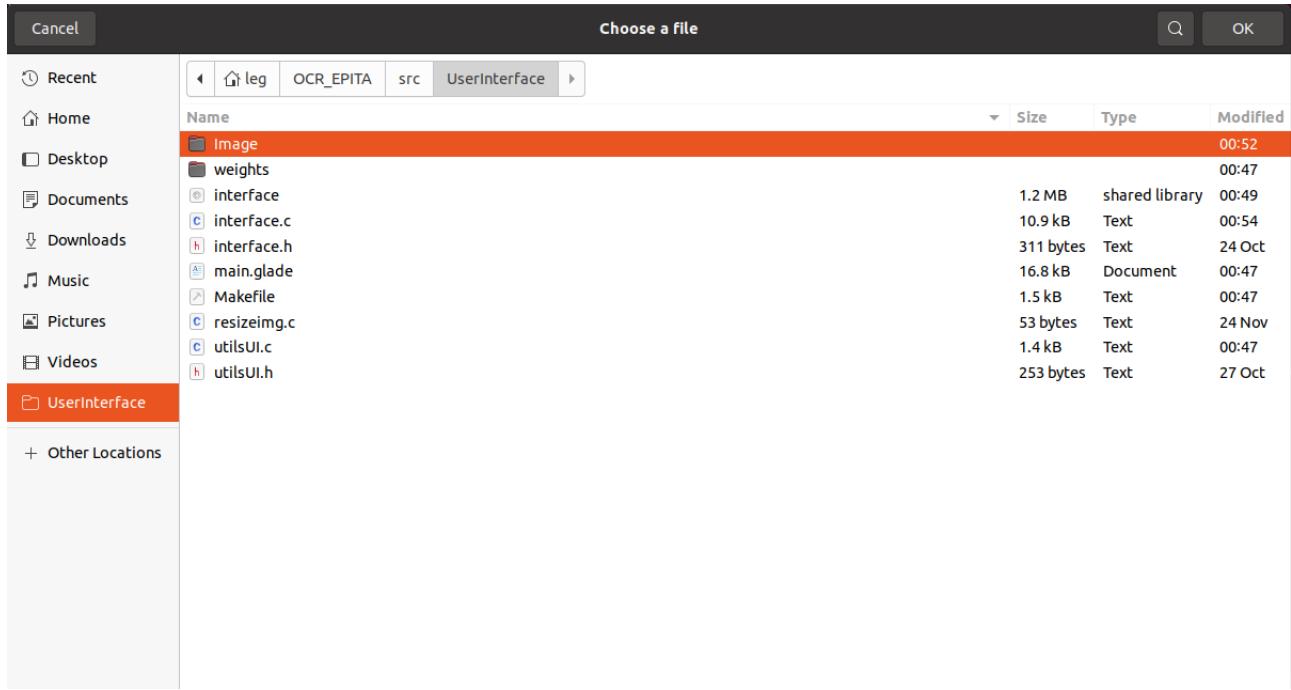


FIGURE 23 – Boite de dialogue permettant de choisir un fichier

rotation de l'image dans le sens horaire et -1 pour une rotation de l'image dans le sens anti-horaire. Ce qui permettait d'effectuer la rotation de X degrés, indiqués par : la valeur absolue de l'accumulateur \* 5), ainsi que la direction indiquée par le signe de l'accumulateur.

### 3.3 Interface utilisateur - Utilisation

Notre programme dispose désormais d'une interface graphique qui permet à l'utilisateur de choisir l'image qu'il souhaite modifier et passer en paramètre de notre programme. Pour ce faire l'utilisateur pourra trouver un bouton dans "File->Open", qui lui ouvrira une boîte de dialogue pour l'utilisateur.

De plus, grâce aux boutons à cocher à droite de l'interface, l'utilisateur peut appliquer "à la main" les différents filtres présents dans notre programme. L'utilisateur peut également modifier manuellement la rotation de l'image, grâce aux boutons sur l'interface graphique.

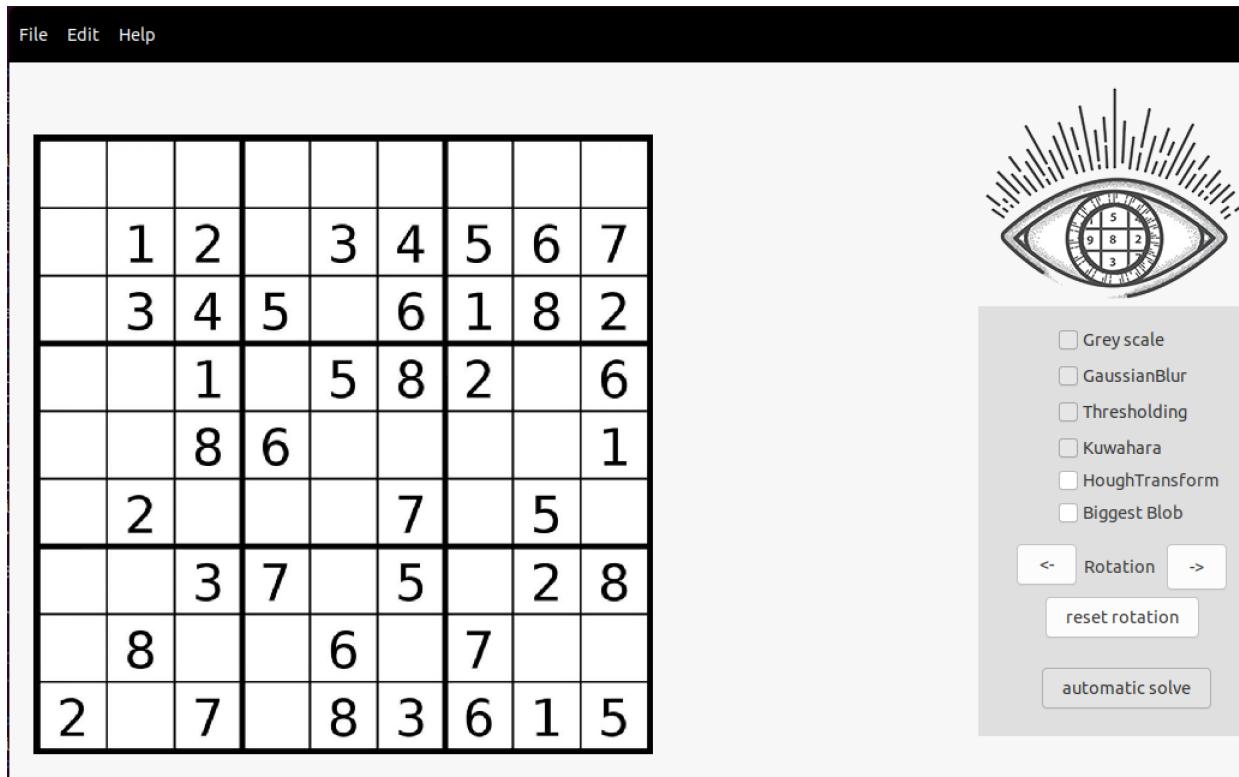
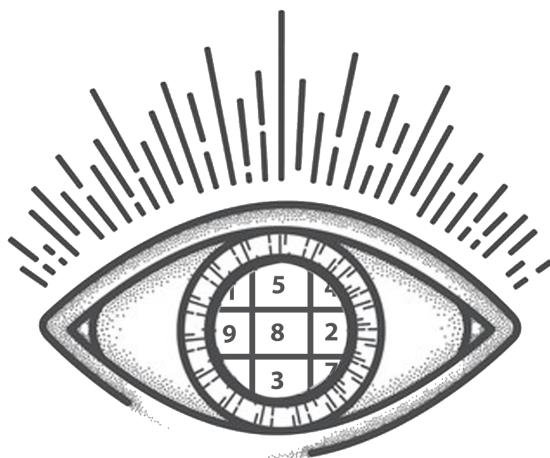


FIGURE 24 – Interface SUDO.C.R

### 3.4 Identité graphique

Pour l'identité graphique de ce projet, contrairement au projet précédent, nous avons décidé de choisir des couleurs sombres.

Notre logo combine à la fois un œil, qui représente la reconnaissance optique , ainsi qu'une grille de sudoku, rappelant le but de ce logiciel à l'utilisateur.



**SUDO.C.R**

FIGURE 25 – Logo de SUDO.C.R

## 4 Réseau de neurones

Nous avons développé un réseau de neurones artificiels.

Nous lui avons appris à faire de la reconnaissance d'images.

Le modèle que nous avons utilisé est le perceptron multi-couches, c'est un système qui apprend par l'expérience. Son objectif est d'apprendre à classifier les données que l'on lui donne en entrée. Pour cela, chaque neurone va créer une courbe pour séparer les données. Le réseau de neurones va calculer une combinaison de pixels d'une image pour reconnaître ce qu'elle représente. Pour cela, nous avons des images en noir et blanc, ainsi les valeurs d'entrée du réseau de neurones seront des nombres entier compris entre 0 et 255. Un pixel noir est représenté par la valeur 0 et un pixel blanc par 255. Un pixel gris foncé aurait par exemple une valeur de 127.

Une manière simple de déterminer quel chiffre est donnée en entrée est d'appliquer des filtres sur l'image. Chaque neurone applique comme un filtre qui reconnaît une forme, et grâce à la combinaison des formes reconnues on peut savoir quel est le chiffre donné en entrée.

### 4.1 Le principe de fonctionnement

Un perceptron multi-couches est un ensemble de neurones organisé en plusieurs couches. On trouve une couche d'entrée, une couche cachée et une couche de sortie. La couche d'entrée et la couche cachée possèdent des matrices de poids et de biais. Les neurones entre chaque couche sont reliés avec des liens pondérés, les valeurs des liens pondérés sont appelées les poids. Les biais sont des valeurs qui permettent de décaler vers la gauche ou la droite les courbes que dessine chaque neurone. On peut résumer le fonctionnement ainsi : on entre des valeurs puis on propage ces valeurs au travers de chacune des couches. Au final, dans la dernière couche, on a 9 neurones qui représentent les chiffres de 1 à 9 et celui avec la valeur la plus proche de 1 est le nombre que le réseau de neurones à détecté.

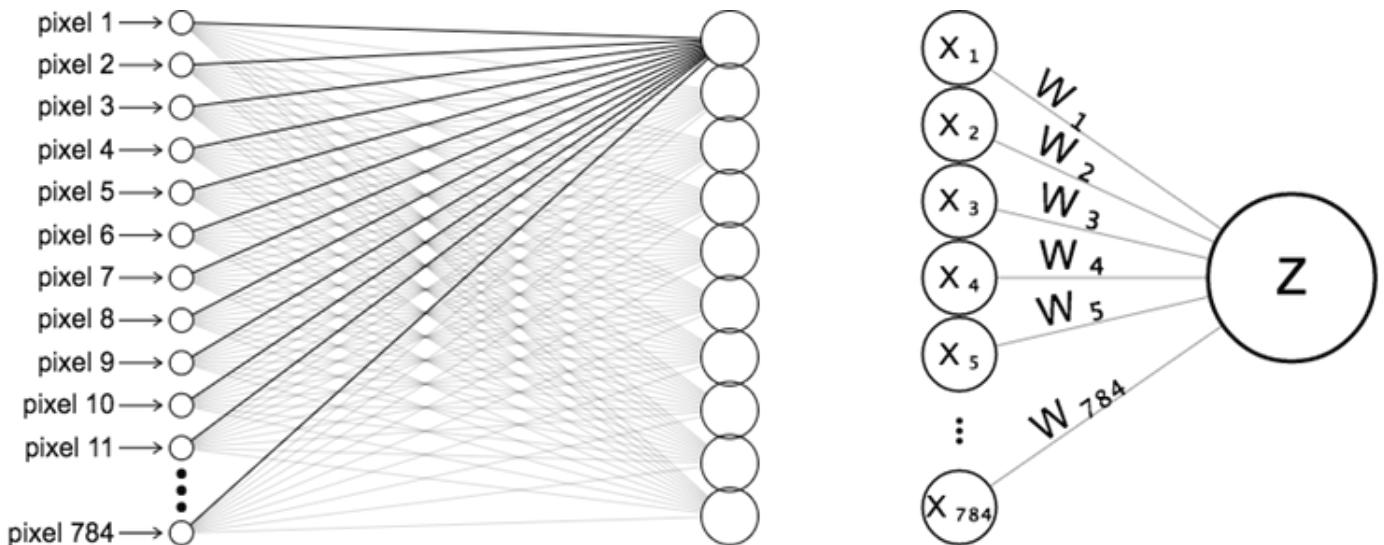


FIGURE 26 – Liaisons pondérées entre la couche d'entrée et la couche cachée

## 4.2 Le principe d'entraînement du réseau de neurones

Nous allons détailler le fonctionnement de notre réseau. Dans un premier temps, il va falloir entraîner le réseau de neurones. La procédure d'entraînement se divise en 4 étapes.

D'abord, on initialise le réseau de neurones en utilisant la loi uniforme pour les poids et on initialise les biais à 0. La loi uniforme soit la formule suivante :

Soit  $n$  le nombre de neurones de la couche à laquelle appartiennent les poids, les poids sélectionnés seront choisis aléatoirement dans l'intervalle suivant  $[-y, y]$  avec  $y = \frac{1}{\sqrt{n}}$ .

Pour initialiser les poids entre le couche d'entrée et la couche cachée, nous utilisons un autre type d'initialisation, la "Xavier/ Glortort Initialization" ainsi les poids appartiennent à l'intervalle suivant  $[-y, y]$  avec  $y = \sqrt{\frac{6}{i+o}}$ ,  $i$  est le nombre de neurones en entrée et  $o$  les nombres de neurones de la couche cachée.

Nous utilisons cette fonction, car elle marche mieux avec la fonction sigmoïde que nous verrons plus tard.

### Bien initialiser les poids est très important.

Si tous les poids sont nul alors le réseau de neurone ne fonctionnera pas, car les valeurs des neurones des couches suivantes ne dépendront plus des valeurs données en entrée.

Tous les neurones suivront le même gradient, ce qui modifiait uniquement l'échelle du vecteur de poids et pas la direction. (Le gradient est le calcul qui permet de faire converger le réseaux de neurones vers une erreur minimal). Donc cela l'empêcheras d'apprendre si les poids sont égaux à 0. L'erreur que l'on propage au travers du réseau de neurones pour corriger les poids est proportionnelle à la valeur des poids donc si les poids sont tous identiques cela ne fonctionne pas, car l'erreur propagée sera la même et tous les poids seront modifiés par la même quantité. Ici, on a donc un problème de symétrie, il faut donc les initialiser avec des valeurs aléatoires différentes.

Ensuite, on propage les valeurs de la couche d'entrée. Pour calculer les valeurs de la couche d'entrée à la couche caché on utilise la formule suivante :

$$h_i = \sigma(x_i W^h + b^h)$$

et de la couche caché à la couche de sortie :

$$y_i = SM(h_i W^y + b^y)$$

Ici  $h$  représente les valeurs des neurones de la couche caché,  $x_i$  est la valeur du pixel donné en entrée. Le poids entre la couche d'entrée et la couche caché sont la matrice  $W^h$  et ceux entre la couche cachée et de sortie  $W^y$ . Les matrices des biais sont  $b^h$  et  $b^y$ . Enfin  $SM$  (softmax) et  $\sigma$  sont des fonctions d'activation.

Les fonctions d'activation que nous utilisons comme la fonction sigmoïde permettent d'obtenir une valeur entre 0 et 1, peu importe la valeur d'entrée. Il suffit de multiplier la matrice des valeurs des neurones de la couche précédentes par la matrice des poids et d'ajouter la matrice des biais. Enfin on active le résultat avec la fonction sigmoïde ou softmax. On obtient un résultat entre 0 et 1 et si le résultat est supérieur à 0.5 alors on considère le neurones actif. Donc le résultat est le nombre associé aux neurones activé. Grâce à cette première étape, on obtient le nombre que le perceptron prédit.

La troisième étape consiste à appliquer la *backpropagation*. Dans un premier temps, on calcule l'erreur (notée  $\delta_i^y$ ) de la couche de sortie pour savoir si notre réseau est performant ou non. Puis on la propage en arrière aux autres couches. On cherche à calculer la valeur de nouveaux poids qui permettrait de réduire l'erreur grâce à l'erreur des couches précédentes. La formule de l'erreur est la soustraction des valeurs de la couche de sortie aux valeurs attendues. Ensuite, grâce à cette erreur, on peut calculer la correction des poids et des biais aussi appelés le gradient noté  $\Delta$ .

Ainsi pour le calcul du gradient d'un poids entre la couche caché et la couche sortie, on utilise la formule suivante :

$$\Delta W_i^y = \delta_i^y * h_i^T$$

<sup>2</sup>.

La formule du gradient des biais  $\Delta b^y$  est :  $\Delta b_i^y = \delta_i^y$  Ensuite, on va propager l'erreur, pour ça on calcule l'erreur de la couche caché avec la formule suivante :

$$\delta_i^h = \delta_i^{yT} W^y \odot \sigma'(h^i)$$

Pour le calcul du gradient d'un poids entre la couche d'entrée et la couche caché, on utilise la formule suivante :

$$\Delta W_i^y = \delta_i^h * x_i^T$$

La formule du gradient des biais  $\Delta b^h$  est :  $\Delta b_i^h = \delta_i^h$

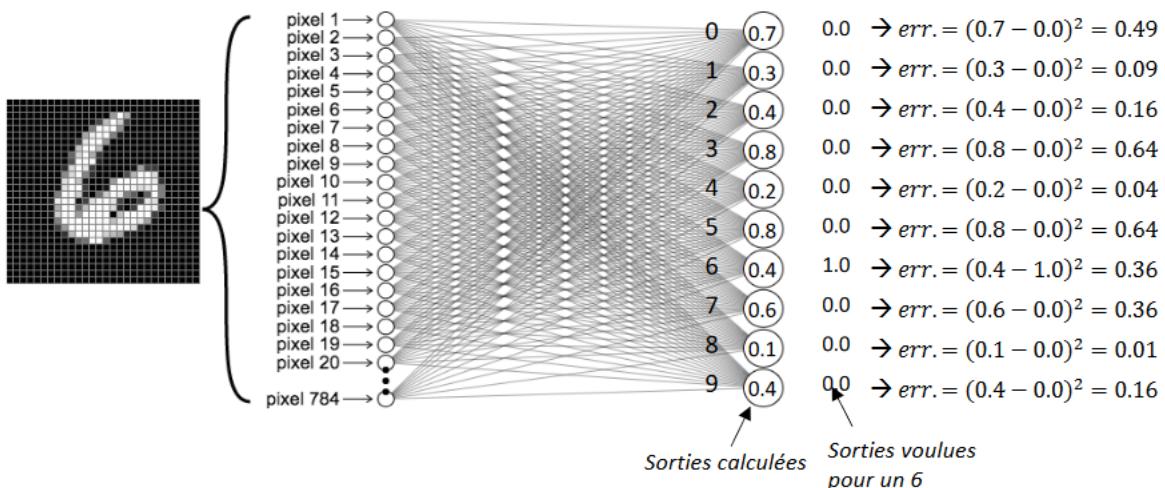


FIGURE 27 – Calcule de l'erreur

Enfin, on applique un algorithme appelé *descente de gradient*. Grâce au gradient, on peut mettre à jour les poids et les biais. Mais pour éviter que le résultat des neurones oscille, en corrigeant les poids avec des trop grandes valeurs, on ajoute une valeur un pas notée  $\alpha$  appelé le taux d'apprentissage. Il faut faire attention qu'il ne soit pas trop petit, car le réseau de neurones évoluerait très lentement ce qui demanderait d'augmenter le nombre

2. T signifie que l'on transpose la matrice.

d'itérations. De plus, nous avons utilisé une descente de gradient spécifique appelé *Stochastic Gradient Descent*. Lors de l'entraînement aux lieux de propager la valeur d'une image, appliquer la propagation de l'erreur puis une descente de gradient, on va travailler sur un échantillon d'images. Pour chaque itération, on va mélanger toutes les images de notre base de données pour éviter que le réseau de neurones apprenne en fonction de l'ordre des images. Ensuite on choisit la taille des échantillons et on fait une propagation et une rétro-propagation de l'erreur pour chaque image de l'échantillon enfin, on fait une descente de gradient. Et on recommence sur l'échantillon suivant. On obtient la formule suivante :  $w_i^{l+1} = \alpha * \frac{1}{m} * w_i^l - \Delta w_i^l$ .

Une fois le réseau de neurones entraîné, il suffit d'entrer des valeurs, de les propager et de regarder la valeur des neurones de sorties.

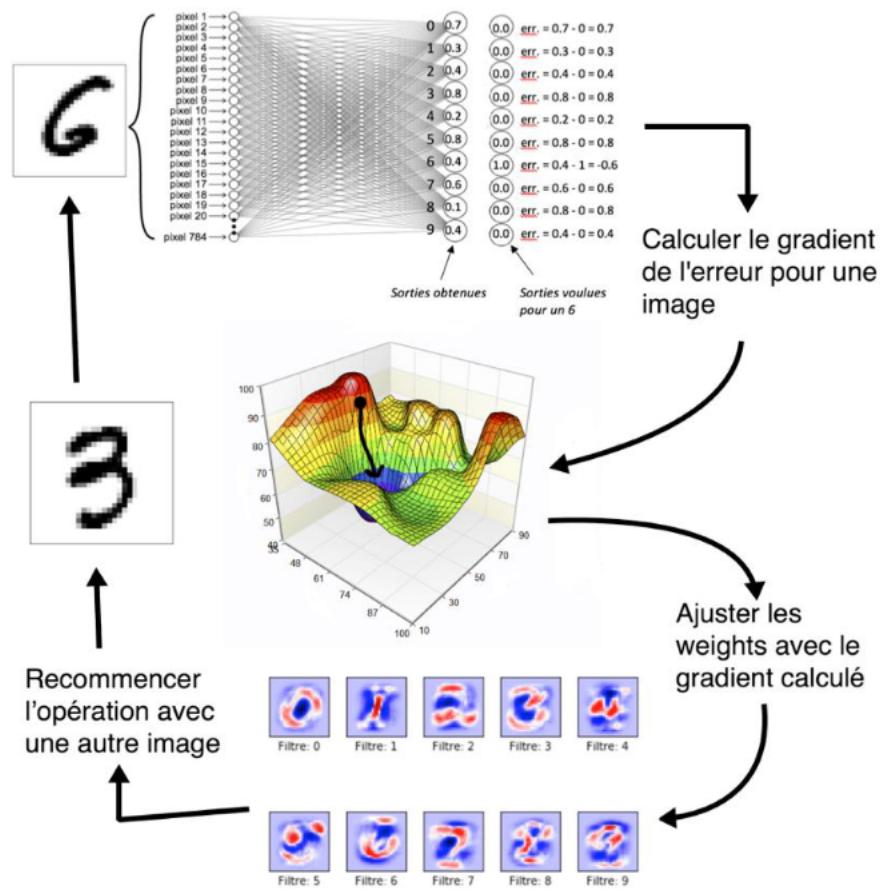


FIGURE 28 – Fonctionnement du réseau de neurones

#### 4.2.1 Base de données

Afin de permettre un meilleur entraînement de notre réseau de neurones, il a été convenu de réaliser une base de données de chiffres de 1 à 9, dans différentes polices. Grâce à un programme réalisé en python, nous avons pu réaliser rapidement une base de données de chiffres de 1 à 9 dans plus de 1000 polices de textes différentes.

Utiliser de nombreuses polices de texte différentes, permet à notre réseau de neurones d'avoir une meilleure adaptation face à des polices qu'il ne connaît pas.

Pour faciliter l'utilisation de notre base de données par le réseau de neurones, nous avons décidé de créer des images de 28 \* 28 pixels, en noir et blanc, ce qui se rapproche de ce que reçoit le réseau de neurones après le pré-traitement de l'entrée utilisateur.

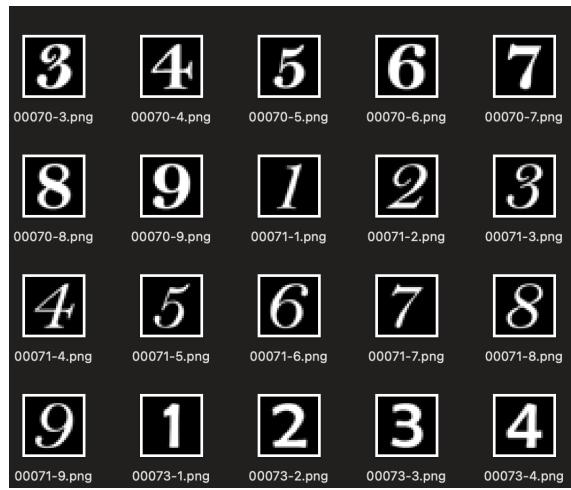


FIGURE 29 – Échantillon de la Base de donnée

## 5 Le solveur de sudoku

Une fois, tous les chiffres reconnus, nous allons résoudre le sudoku. Lors de la première soutenance, ce solveur était juste un programme à part désormais, il est intégré dans le projet.

### 5.1 Le principe de fonctionnement

Notre solveur utilise un algorithme de "backtracking", il s'agit d'un algorithme basique qui va consister à tester chaque possibilité possible (c'est-à-dire qu'avant de supposer un nombre, il va regarder s'il n'est pas déjà présent dans la carrée la ligne ou la colonne) et il va continuer de faire des suppositions jusqu'à ce qu'il soit arrivé à un endroit où aucun choix n'est possible ou que la grille soit complétée. S'il est dans le cas de figure ou aucun choix n'est possible, il va alors revenir en arrière et remplacer le dernier chiffre supposé par un autre. C'est pour cela que ça s'appelle un algorithme de backtracking, c'est, car il va à chaque fois revenir en arrière pour tester d'autres solutions jusqu'à la résolution complète du sudoku.

### 5.2 Benchmarking

Dans cette partie, nous allons parler des performances de notre solveur. Les avantages de cet algorithme sont :

- La garantie de trouver une solution (s'il est faisable).
- Le temps passé à résoudre la grille n'est presque pas influencé par le niveau de difficulté de la grille.
- L'algorithme est plus simple en termes d'implémentation que d'autres solutions comme le DPLL (Pour l'algorithme de Davis–Putnam–Logemann–Loveland) car c'est la solution la plus intuitive face à ce genre de problème.

Le principal désavantage est qu'il existe des grilles faites spécialement face à ce genre d'algorithme comme celle-ci :

Sur cette grille en particulier, notre algorithme va prendre environ 2 secondes contre quelques centième de seconde sur une grille normale. Cependant, il existe d'autres solutions comme la recherche stochastique qui va :

- 1 Remplir toutes les cases vides par des nombres aléatoires
- 2 Calculer le nombre d'erreurs
- 3 Mélanger les nombres qui ont été mis dans le sodoku dans la première étape jusqu'à ce qu'il y ait un nombre d'erreur égal à 0

Ils seront bien plus rapides sur ce genre de grille dû à leurs différences de fonctionnements.

## 6 Mode d'emploi

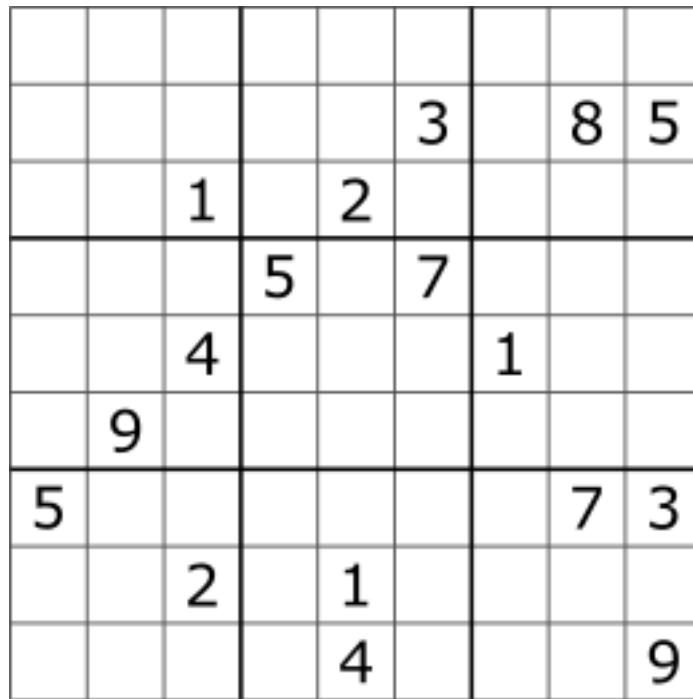


FIGURE 30 – Grille conçue contre les algorithme de backtracking

**Installation**

```
git clone https://github.com/Helyosis/OCR_EPITA.git
cd OCR_EPITA/src
make
```

**Usage****How to compile**

```
git clone https://github.com/Helyosis/OCR_EPITA
cd OCR_EPITA/src/ && make
cd UserInterface/ && make
```

FIGURE 31 – Compilation et installation de notre programme

**How to use**

Commande line:

```
./main 3.14.15 help:
[ Image mode specific options ]
-i file: Specify the input file (required)
-o file: Specify the output file (default: out.bmp)
--show: Show the image being processed, one step at a time
[ Train mode specific options ]
-n nb: Specify the number of iterations to train the neural net with (default is 100 000)
-o file: Specify the output file to save the neural network
--batch-size s: Specify the numbers of elements in a minibatch size (default 100)
--nb-images n: Specify the number of image to train with. (default 8228)
--learning-rate n / --step-size n: Specify the step size (default 0.25)
[ General options ]
-v: Increase the verbose level (default 0), can be used up to 3 times
--mode mode: Specify the mode to use. Can be one of IMAGE/TRAIN/GUI (default is GUI)
-h / --help: Show this help and quit
```

Graphical Interface:

```
cd OCR_EPITA/src/UserInterface
./interface
```

Then select your sudoku grid and click on the "solve" button and your grid will be solved automatically.

FIGURE 32 – Mode d’emploi de notre programme

## 7 Conclusion

En conclusion de ce projet "SUDO.C.R", nous avions comme objectif de réaliser un programme capable de reconnaître une grille de sudoku, puis de le résoudre. Nous avons, au bout de ces 3 mois de projet, réalisé un programme "SUDO.C.R", qui effectue un pré-traitement des images de l'utilisateur afin de normaliser les entrées de l'utilisateur. Puis grâce à un OCR, une intelligence artificielle ayant pour objectif de reconnaître des caractères, nous trouvons les chiffres déjà présents dans le sudoku. Puis, lorsque les caractères sont reconnus, le résultat est envoyé à un sous-programme le solveur de sudoku, qui permet grâce à un algorithme de backtracking de résoudre rapidement le sudoku.

Ce projet a aussi pour objectif de réaliser un travail en groupe, puisque nous avions à réaliser un programme très complet en peu de temps. Par conséquent, afin de réaliser ce projet dans les temps, nous avions du répartir les différentes tâches entre les 4 membres du groupe de projet.

### 7.1 Réalisation aux soutenances

#### 7.1.1 1ère soutenance

Lors de la première soutenance, nous avions réalisé un réseau de neurones capable d'apprendre le XOR, une majorité du pré-traitement de l'image avec l'application de filtres comme : le niveau de gris, le noir et blanc, le flou gaussien. Nous avions aussi réalisé une partie de l'interface graphique, qui ne fonctionnait malheureusement pas sur les ordinateurs de l'école, à cause d'une partie de la librairie "GTK" non-présente sur les ordinateurs de l'école. Nous avions aussi un problème avec le filtre permettant la détection des lignes du

sudoku, ce qui ne nous permettait pas de bien découper le sudoku afin de bien savoir où sont positionné les caractères reconnus.

### 7.1.2 2ème soutenance

Durant la seconde partie de ce projet, nous avons finalisé ce projet et répondu aux attentes que nous nous étions fixées lors du début de ce projet.

Nous avons amélioré et ajouté différents filtres afin d'améliorer le pré-traitement de l'image, nous avons aussi implémenté un réseau de neurones capable de s'entraîner et de reconnaître des chiffres.

L'interface graphique a, elle aussi, été améliorée permettant une pleine utilisation de notre projet via une interface graphique facile à utiliser.

## 7.2 Problèmes rencontrés

Premièrement, nous avons rencontré plusieurs problèmes lors de la confection de l'interface graphique, majoritairement dus à des problèmes de compatibilité entre nos ordinateurs personnels et les ordinateurs de l'école. Ce qui nous a obligés à travailler en grande partie sur les ordinateurs de l'école et non sur nos ordinateurs personnels. Ensuite, nous avons rencontré des problèmes aux niveaux de réseaux de neurones, il était assez difficile de comprendre pourquoi il ne convergeait pas vers une solution satisfaisante, les problèmes étant variés comme une mauvaise initialisation des poids. Nous sommes repartis 3 fois de zéro avec des modèles différent avant d'avoir un résultat convenable.

## 8 Bibliographie

- Manuel de Référence de GTK+. <https://gtk.developpez.com/doc/fr/gtk/index.html>. Accessed 8 Dec. 2021.
- Glade Reference Manual. <http://www.fifi.org/doc/glade-common/help/glade/C/user-guide/index.html>.
- <https://cedric.cnam.fr/vertigo/Cours/ml2/docs/coursDeep1.pdf>
- Travaux Pratiques - Perceptron Multi-Couche — Cnam – UE RCP209. <https://cedric.cnam.fr/vertigo/Cours/ml2/tpDeepLearning2.html>.
- Numerical Images. <https://kaggle.com/pintowar/numerical-images>.
- <https://www.apprendre-en-ligne.net/TM/neurones.pdf>
- <https://www.adventuresinmachinelearning.com/wp-content/uploads/2020/02/A-beginners-introduction-to-neural-networks-V3.pdf>
- Implementing a Neural Network in C. <https://www.cs.bham.ac.uk/~jxb/NN/nn.html>.
- Backpropagation Calculus | Chapter 4, Deep Learning. <https://www.youtube.com/watch?v=tIeHLnjs5U8>
- <https://neuralnetworksanddeeplearning.com/chap1.html>
- <https://neuralnetworksanddeeplearning.com/chap2.html>
- Genetic Algorithms and Machine Learning for Programmers, France Buontempo
- <https://helios.mi.parisdescartes.fr/bouzy/Doc/AA1/ReseauxDeNeurones1.pdf>