

# SUDOCODE

*OCR - Sudo-code* : Rapport de soutenance.

SOUTENANCE 2.

OUFFA Rawane - LOPES Christina - HAMDOUN Zaky - DHEHIBI Dima

---

EPITA - 66 Rue Guy Môquet, 94800 Villejuif

*S3 - B1*

Villejuif, France, Lundi 12 Décembre 2022.

”Les détails font la perfection, et la perfection n’est pas un détail.”  
Léonard de Vinci.

## Table des matières

<b>1</b>	<b>Avant-Propos.</b>	<b>4</b>
1.1	Présentation. . . . .	4
1.2	Sujet. . . . .	4
1.3	Composition du groupe. . . . .	4
1.4	Contact. . . . .	5
<b>2</b>	<b>Nature.</b>	<b>7</b>
2.1	Nature du projet. . . . .	7
<b>3</b>	<b>Exécution du programme principal (<i>main</i>).</b>	<b>8</b>
3.1	Makefile. . . . .	8
3.2	Main. . . . .	8
<b>4</b>	<b>Tâches.</b>	<b>10</b>
4.1	Chargement et sauvegarde des images. . . . .	10
4.2	Mise en nuance de gris (grayscale). . . . .	11
4.3	Réduction des bruits (flou gaussien et baguette magique). . . . .	12
4.4	Augmentation de l'éclairage contrasté (gamma). . . . .	14
4.5	Binarisation de l'image (mise en noir et blanc). . . . .	15
4.6	Rotation. . . . .	16
4.6.1	Rotation manuelle. . . . .	16
4.6.2	Rotation automatique. . . . .	17
4.7	Détection et extraction de la grille. . . . .	18
4.8	Détection et extraction des chiffres. . . . .	21
4.9	Réseau de neurones. . . . .	22
4.10	Construction de la grille. . . . .	25
4.11	Chargement de la grille de sudoku. . . . .	25
4.12	Résolution de la grille de sudoku. . . . .	26
4.13	Réseau de neurones complet et fonctionnel . . . . .	27
4.13.1	Jeu d'images pour l'apprentissage . . . . .	27
4.13.2	Apprentissage . . . . .	28
4.13.3	Sauvegarde et chargement des poids du réseau de neurones . . . . .	28
4.14	Interface graphique . . . . .	29
<b>5</b>	<b>Conclusion.</b>	<b>36</b>

## 1 Avant-Propos.

### 1.1 Présentation.

Dans le cadre du projet du troisième semestre à l'EPITA, nous allons mettre en oeuvre les connaissances acquises lors des TDs et TPs, ainsi que nos capacités à rechercher de nouvelles informations et méthodes. Ce projet se voit être temporairement divisé en deux dates clés : elles sont celles de la première et deuxième soutenance.

Ce document est un rapport de soutenance. Il a pour but de mettre en exergue l'avancement du projet entre la formation du groupe ainsi que la première soutenance. Il met également en avant les prochaines étapes à suivre et la composition du groupe.

### 1.2 Sujet.

L'objectif principal de ce projet est de former le groupe à l'utilisation de divers outils informatiques. Principalement, le groupe devra mettre en place un réseau de neurones, une interface, et tout un système de traitement d'images.

Par ailleurs, ce projet entraîne également les compétences organisationnelles des membres du groupe, afin de perfectionner l'utilisation de git, et, plus généralement, permettre une amélioration des aptitudes à travailler en groupe.

### 1.3 Composition du groupe.

Le groupe est composé de quatre personnes. Le nombre d'individus présents dans le groupe est un atout : ce dernier garantit la diversité des opinions vis-à-vis des fonctionnalités et caractéristiques du groupe, mais également la possibilité d'effectuer une division des tâches efficace en fonction des spécialités de chacun. Ci-dessous la liste de ceux composant le groupe Sudo-Code :

*Chef de groupe : Mr. HAMDOUN Zaky.*

- Passionné par l'informatique et la géopolitique, j'ai rejoint l'EPITA avec plus tard l'envie de devenir ingénieur en informatique spécialisé en cybersécurité, ou en big data et intelligence artificielle. Cependant, mon penchant pour la géopolitique et les métiers de la défense font également que je suis intéressé par l'Open Source Intelligence (OSINT) et les renseignements, qui se résument très souvent à du traitement d'image, comme c'est le cas dans ce projet. J'aime également les mathématiques et la philosophie.

*LOPES Christina.*

- De nature curieuse, je suis autant intéressée par les sciences dures que les sciences humaines, avec une attirance particulière pour les concepts complexes au premier abord. C'est d'ailleurs ce qui alimente mon intérêt pour l'informatique, qui est l'union de plusieurs

domaines scientifiques bien distincts comme l'électronique, l'algorithmique, l'architecture des données sans oublier les concepts mathématiques et les notions mises en avant dans la formation à l'EPITA. De plus l'envie de comprendre et d'en apprendre davantage sur ces notions, est ma principale source de motivation.

*DHEHIBI Dima.*

- Attentive et passionnée par les nouvelles technologies, j'ai le goût pour la découverte de nouvelles formes de communications et de résolutions de problématiques variées. C'est ce qui fait notamment mon intérêt pour la vocation d'ingénieure informatique, au sein de laquelle j'ai la certitude de pouvoir m'épanouir dans le futur. Les dernières innovations dans le domaine de la physique quantique m'inspirent autant dans leur formulation que leur développement. Il me tarde de voir la résolution de sudokus sous forme quantique.

*OUFFA Rawane.*

- Attirée depuis très jeune par la Science sous toutes ses formes, j'ai d'abord commencé par me passionner pour l'astrophysique, autant pour le côté historique que technique, ne cherchant par là qu'à satisfaire ma soif de savoir. Cela m'a amené à remettre en question le monde dans lequel j'évoluais et interagissais. Par la suite, j'ai entrepris de considérer plus sérieusement l'informatique pour son potentiel "créer quelque chose à partir de rien". Appréciant énormément la lecture de livres de fiction, j'ai vu la capacité de l'imagination à se déployer dans l'informatique. Par ailleurs, les découvertes et innovations informatiques font que l'on n'a jamais fini d'apprendre !

#### 1.4 Contact.

- Mr. HAMDOUN Zaky :
  - Login : zaky.hamdoun
  - Adresse e-mail : zaky.hamdoun@epita.fr
  - Numéro de téléphone : +33 6 69 69 69 59
- Mme OUFFA Rawane :
  - Login : rawane.ouffa
  - Adresse e-mail : rawane.ouffa@epita.fr
  - Numéro de téléphone : +33 6 58 74 44 52
- Mme DHEHIBI Dima :
  - Login : dima.dhehibi
  - Adresse e-mail : dima.dhehibi@epita.fr
  - Numéro de téléphone : +33 6 63 76 10 64
- Mme LOPES Christina :
  - Login : christina
  - Adresse e-mail : christina.lopes@epita.fr
  - Numéro de téléphone : +33 6 63 52 29 98

La communication entre les membres du groupe se fait principalement via un **serveur Discord dédié**. Un contact peut également être établi avec eux par l'intermédiaire de ce

réseau social. À noter qu'une communication présentielle est tout de même privilégiée au sein de groupe, cependant Discord est un très bon outil permettant de s'adapter simplement aux disponibilités de chacun.

## 2 Nature.

### 2.1 Nature du projet.

***OCR - Sudo-code*** est une **interface de reconnaissance optique de caractères et de résolution de sudoku** programmée en langage C. Le projet contient également une partie de prétraitement d'image. À partir d'une photo de sudoku, le rendu final doit pouvoir extraire la grille, la résoudre, ainsi que la reconstruire de manière automatique, ou plus ou moins manuelle. La reconnaissance des chiffres doit se faire grâce à un réseau de neurones capable de s'entraîner sur un ensemble fini d'exemples. Tout de même, le projet contient des contraintes spécifiques à respecter. Dans un premier temps, il y a toute une série de contraintes d'écritures, qui permettent la création d'un code compréhensible et plaisant visuellement. Il y a ensuite des contraintes d'organisation entre autres constituées des règles sur le dépôt git ainsi que les dates de rendus. Enfin, le projet doit se limiter à l'utilisation de bibliothèques extérieures C spécifiques : SDL et GTK.

### 3 Exécution du programme principal (*main*).

#### 3.1 Makefile.

Pour chaque partie du projet, un fichier *Makefile* est présent afin de générer les exécutables des différents programmes de manière individuelle. Cela permet notamment un meilleur débogage, ainsi que la possibilité de travailler à plusieurs de manière organisée.

Par ailleurs, les fichiers Makefile des programmes générant des images possèdent tous l'instruction *clean\_images* permettant de ne supprimer que les images. L'instruction *clean* quant à elle, supprime tous les fichiers générés (images, résultats, et exécutables). L'instruction *all* est également présente.

```
$ make program
$ make clean_images
$ make clean
$ make all
$ make clean
```

FIGURE 1 : fonctionnement de la commande *make*.

#### 3.2 Main.

Un fichier *main.c* est présent dans la racine du projet afin de pouvoir le générer. Là où précédemment, le fichier *main* permettait d'exécuter chaque instruction séparément, il est désormais impossible d'exécuter le programme de cette manière. En effet, la programme *main.c* est désormais uniquement relié à l'interface, constituant la partie centrale du projet. Cette dernière, une fois lancée, permet de gérer toutes les autres situations.

```
$ make all                # Commande make.
$ ./main                  # Lancement de la fonction.
$ make clean              # Reset l' tat avant compilation.
# Peuvent galemment     tre utilis s :
$ make                    # Commande make.
$ ./main                  # Lancement de la fonction.
$ make clean              # Reset l' tat avant compliation.
```

FIGURE 2 : fonctionnement de la commande *./main*.

```
$ make all
$ ./main "img" gray      # Mise en nuance de gris.
$ ./main "img" gauss     # Flou gaussien.
```



```

$ ./main "img" gamma           # Augmentation du gamma.
$ ./main "img" bin             # Binarisation.
$ ./main "img" rotation angle  # Rotation avec angle.
$ ./main "img" auto-rotation   # Rotation automatique.
$ ./main "img" all-no-rotate   # Tout sauf rotation.
$ ./main "img" all-man-rotate angle # Tout, rotation manuelle.
$ ./main "img" all-auto-rotate # Tout, rotation automatique.
$ ./main "img" grid_extract _  # Extraction de la grille.*
$ make clean

```

*FIGURE 2.bis : fonctionnement de la commande ./main.*

## 4 Tâches.

### 4.1 Chargement et sauvegarde des images.

Avant même de commencer à modifier les images, il est nécessaire de les charger ainsi que de pouvoir sauvegarder les modifications de ces dernières. Étant donné que le chargement et la sauvegarde d'image constituent une sous-étape qui sera présente dans une majorité des autres tâches, il était obligatoire de les séparer afin de pouvoir permettre une utilisation des fonctions dédiées partout dans le programme. C'est notamment la raison pour laquelle nous avons décidé de créer un dossier *utils* dans lequel on trouve le fichier *auxiliary.c* contenant une vingtaine de fonctions auxiliaires, utilisables dans tous les fichiers du projet par l'intermédiaire de la directive d'inclusion :

```
#include " ../utils/auxiliary.h"
```

FIGURE 3 : directive d'inclusion d'*auxiliary.h*.

Deux fonctions permettant pour la première de charger une image sous forme de surface SDL, et pour la deuxième de sauvegarder une surface en .jpeg grâce à une fonction présente nativement dans SDL ont donc été définies.

Pour le bien du rapport, une image sera utilisée afin de mettre en avant les différentes étapes du traitement de l'image ainsi que de la détection des grilles. Sachant qu'il est nécessaire de montrer également la rotation, l'image choisie pour l'exemple sera l'image numéro cinq.

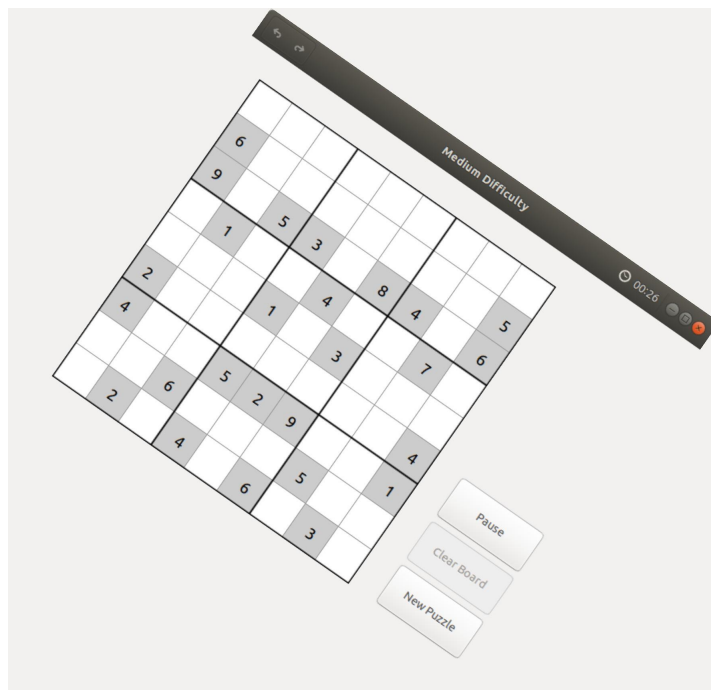


FIGURE 4 : image numéro 5 - après chargement (original).

## 4.2 Mise en nuance de gris (grayscale).

La deuxième étape correspond à la première étape du prétraitement. Cette dernière est la mise en nuance de gris de l'image. En effet, les couleurs ne sont pas nécessaires pour la reconnaissance des caractères, il est donc beaucoup plus utile de transformer l'image en noir et blanc afin de pouvoir traiter les pixels en fonction de leur couleur. Cependant, pour atteindre un noir et blanc complet, il faut d'abord passer par des nuances de gris.

Pour convertir une image infographique couleur en niveau de gris il faut remplacer, pour chaque pixel les trois valeurs représentant les niveaux de rouge, de vert et de bleu, par une seule valeur représentant la luminosité. Pour cela, l'utilisation d'une formule mathématique calculant la moyenne des composantes rouges, vertes et bleues de chaque pixel est nécessaire. Celle-ci fut définie de la manière suivante :

$$gray = 0.4 * red + 0.35 * green + 0.25 * blue$$

FIGURE 5 : formule de calcul du niveau de gris.

Cette transformation permet alors, lorsqu'appliquée sur tous les pixels de l'image, d'obtenir la version visuellement nivelée en gris suivante :

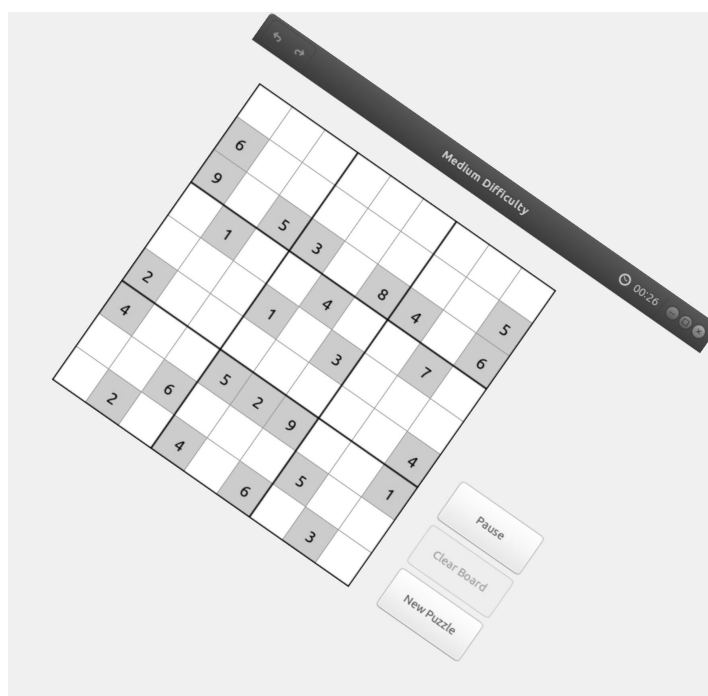


FIGURE 6 : image numéro 5 en niveau de gris.

L'image est ensuite sauvegardée sous le nom de "gray\_image.jpeg". C'est l'image qui sera utilisée pour la prochaine étape. Une nouvelle sauvegarde se fait d'ailleurs pour chaque étape du prétraitement afin d'examiner les résultats en cas d'erreur. Par ailleurs, à chaque étape, c'est le résultat de l'étape précédente qui est utilisé, il est donc possible de préciser la nouvelle adresse à modifier de manière arbitraire. Afin de simplifier les différentes transitions, l'image est tout le temps sauvegardée dans le dossier racine. Enfin, le message suivant apparaît à la fin de la mise en nuance de gris :

Grayscale: **done!**

FIGURE 7 : message affiché après la mise en nuance de gris.

### 4.3 Réduction des bruits (flou gaussien et baguette magique).

Afin de simplifier la reconnaissance des pixels pour la mise en noir et blanc de l'image, une réduction des bruits des pixels est nécessaire. On appelle bruit numérique toute fluctuation parasite ou dégradation que subit l'image dès l'instant de son acquisition jusqu'à son enregistrement. En l'occurrence, la qualité des images contenant les grilles de sudoku n'est pas connue en avance, il est donc nécessaire de faire le mieux possible pour réduire les bruits, dans le cas où il y en aurait.

Pour cela, le flou gaussien constitue une solution fonctionnelle et simple à implémenter à l'aide des fonctions déjà présentes dans les fichiers du projet. Il utilise notamment une moyenne qui est calculée par rapport aux pixels alentours de celui qui est analysé. Dès lors, le pixel concerné peut voir ses valeurs RGB modifiées de la manière suivante :

$$(R, G, B) = \left( \frac{\sum_{n=0}^n r}{n}, \frac{\sum_{n=0}^n g}{n}, \frac{\sum_{n=0}^n b}{n} \right)$$

FIGURE 8 : formule de calcul du flou gaussien.

Avec  $n$  correspondant au nombre de pixel choisi dans les paramètres de la fonction, le rayon du carré autour du pixel concerné. En l'occurrence,  $n = 1$ . Pour rappel, cette étape est effectuée après la mise en niveau de gris. Cette transformation permet alors, lorsqu'appliquée sur tous les pixels de l'image, d'obtenir la version gaussienne de cette dernière :

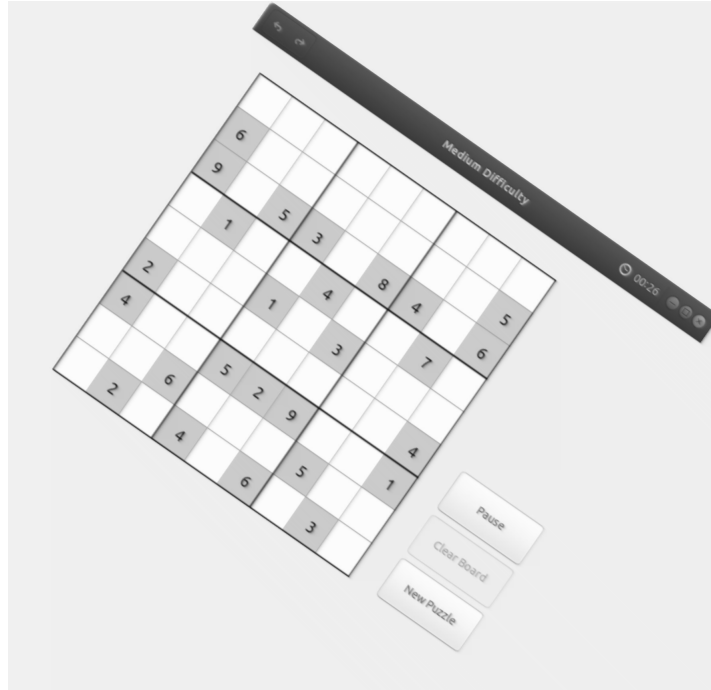


FIGURE 9 : image numéro 5 avec flou gaussien.

Enfin, le message suivant apparaît à la fin du flou gaussien :

Gaussian : **done!**

*FIGURE 10 : message affiché après le flou gaussien.*

Cette étape n'est d'ailleurs pas l'unique étape nécessitant une réduction des bruits. Effectivement, avant la prédiction du réseau de neurone, il faut absolument supprimer les bouts de grilles présents dans les images des chiffres extraits. Ces derniers peuvent, dans le cas où ils sont trop nombreux, fausser les résultats du réseau de neurone, et ainsi produire une grille insolvable.

Ci-dessous, un chiffre nécessitant une réduction des bruits par baguette magique.



*FIGURE 11 : chiffre extrait.*

L'algorithme de la baguette magique, en anglais : Magic Wand ou Flood Fill, est un algorithme récursif très simple à implémenter au sein du traitement d'image. En effet, son pseudo-code se présente tel que :

```
PROCEDURE baguette_magique(x, y, couleur_fill)

    couleur = avoir_couleur(x, y)
    si couleur est gal couleur_fill:
        Repeindre le pixel aux coordonn es (x, y)
        baguette_magique(x - 1, y, couleur_fill)
        baguette_magique(x + 1, y, couleur_fill)
        baguette_magique(x, y - 1, couleur_fill)
        baguette_magique(x, y + 1, couleur_fill)
```

*FIGURE 12 : création de l'accumulateur de Hough.*

Avec cet algorithme, qu'on exécute sur tous les pixels formant les contours (d'une épaisseur de 4) du carré, on peut très facilement supprimer une majorité des bruits parasites, et ainsi avoir une image modifiée étant celle-ci :

# 6

*FIGURE 13 : chiffre extrait.*

## 4.4 Augmentation de l'éclairage contrasté (gamma).

Pour certaines images, notamment les images prises avec un téléphone, il est fréquent d'observer la présence de l'ombre du photographe. Celle-ci assombrit certains pixels, et il est donc nécessaire d'intégrer une fonction permettant la correction de la luminosité de ces pixels. Une augmentation du gamma de l'image est donc obligatoire.

En photographie, le gamma, ou facteur de contraste, caractérise le contraste de l'émulsion. Plus sa valeur absolue est élevée, plus l'émulsion est contrastée et inversement. Une émulsion négative a un gamma positif puisqu'elle s'opacifie avec la lumière. À l'inverse une émulsion inversible (diapositive) a un gamma négatif. L'augmentation du gamma se fait par l'intermédiaire d'une fonction  $g(x)$  définie en l'occurrence par :

$$g(x) = 255 * \left(\frac{x}{255}\right)^n$$

*FIGURE 14 : formule de l'augmentation du gamma.*

Avec :

- $x$  : valeur de la couleur du pixel en 32 bits non signés.
- $n$  : coefficient du gamma, en l'occurrence 0,623.

Cette étape est effectuée après le flou gaussien. La transformation permet alors, lorsqu'appliquée sur tous les pixels de l'image, d'obtenir la version contrastée de cette dernière :

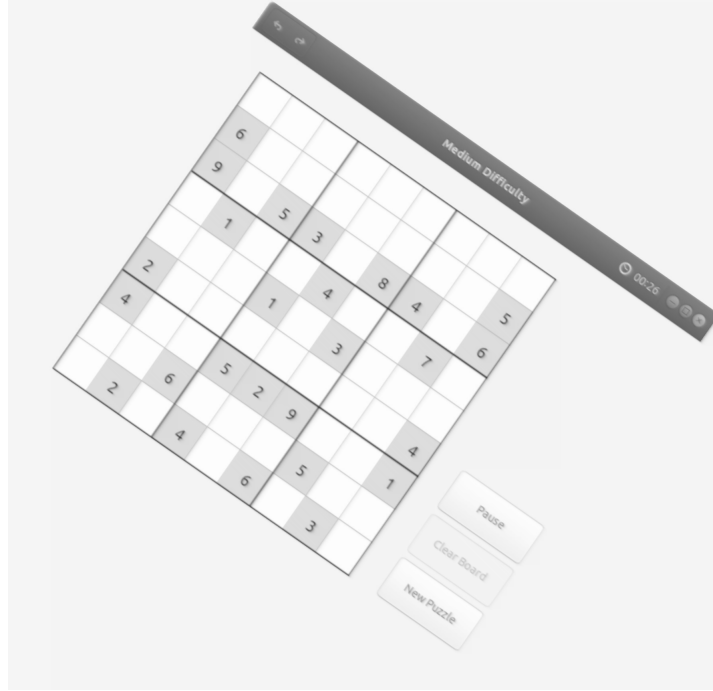


FIGURE 15 : image numéro 5 avec augmentation du gamma.

Enfin, le message suivant apparaît à la fin de l'augmentation du contraste :

Gamma: **done!**

FIGURE 16 : message affiché après l'augmentation du contraste.

#### 4.5 Binarisation de l'image (mise en noir et blanc).

Le principe de binarisation consiste à passer d'une image grayscale ( c'est à dire avec plusieurs niveaux de gris différents) à une image aux couleurs totalement binaire c'est à dire dans laquelle chaque pixel est soit noir soit blanc. Pour faire cela, le principe est identique à l'étape de grayscale qui crée une surface (à partir d'une image colorée) avec différents tons de gris.

Dans un premier temps nous calculons la valeur du pixel traité en fonctions de ces coefficients RGB associés, puis nous vérifions si le résultat obtenu est supérieur à un certain seuil. Si oui, alors le pixel passe au noir, sinon, le pixel sera blanc.

La valeur de seuil n'est pas fixe. Elle dépend de l'image traitée. Le calcul consiste à prendre les valeurs des pixels voisins à notre pixel traités afin de calculer la valeur de  $t$  tel que la variance reste identique le plus souvent, puis nous renvoyons cette valeur comme étant le seuil. Deux méthodes nous permettent de calculer cela :

$$\sigma^2(t) = \omega_{bg}(t)\sigma_{bg}^2(t) + \omega_{fg}(t)\sigma_{fg}^2(t)$$

FIGURE 17 : formule probabilistique de la variance.

$$\sigma^2(t) = \frac{\sum (x_i - \bar{x})^2}{N - 1}$$

FIGURE 18 : formule par somme de la variance.

Il est ensuite possible de calculer la variance pour différentes valeurs de  $t$ . La variance sera alors seuillée étant donnée que pour un certain  $t$ ,  $\sigma^2(t)$  threshold, celle-ci ne changera plus. Le  $t$  deviendra alors le seuil à utiliser dans le cadre de la binarisation. Cependant, il est important de noter que la binarisation d'Otsu ne fonctionne pas avec toutes les images. Nous avons donc pris le choix d'ajouter une méthode de binarisation par moyenne, qui n'a malheureusement pas pu être implémentée pour la première soutenance, mais qui le sera pour la deuxième soutenance. L'image suivante est alors obtenue :

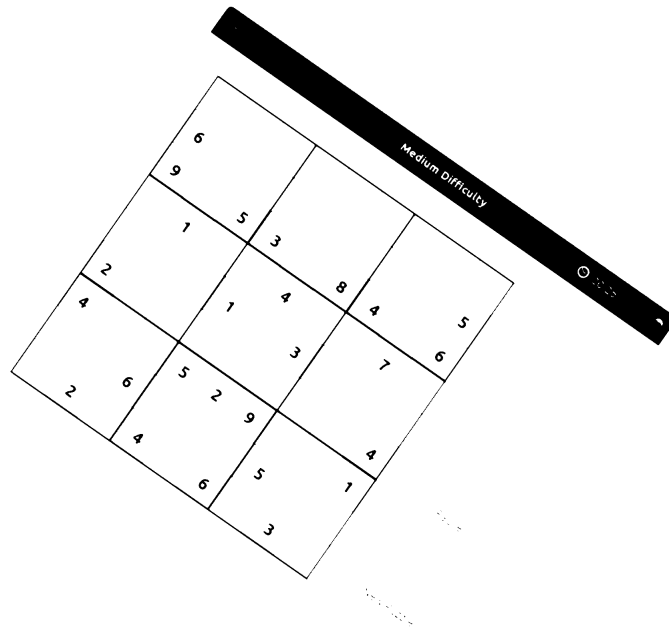


FIGURE 19 : image numéro 5 binarisée.

Enfin, le message suivant apparaît à la fin de l'augmentation du contraste :

Binarization : **done!**

FIGURE 20 : message affiché après binarisation.

## 4.6 Rotation.

### 4.6.1 Rotation manuelle.

Dans le cas où l'image aurait subi une rotation avant d'être insérée en paramètre, il est nécessaire de pouvoir la faire tourner d'un certain angle à l'aide d'une fonction de rotation centrale. La partie rotation manuelle s'occupe donc d'effectuer cette opération géométrique de manière arbitraire, avec un angle qui doit être entré lors de l'appel de la fonction.



Il est ensuite possible de calculer les nouvelles dimensions de l'image, étant donné qu'elles risquent d'être modifiées lors du processus, on a donc :

$$l_d = \text{ceil}(l_s * |\cos(\theta)| + h_s * |\sin(\theta)|)$$

$$h_d = \text{ceil}(l_s * |\sin(\theta)| + h_s * |\cos(\theta)|)$$

*FIGURE 21 : formules de calcul des nouvelles dimensions de l'image.*

Avec :

- l : longueur de l'image (d pour destination et s pour source).
- h : largeur (hauteur) de l'image (d pour destination et s pour source).
- $\theta$  : angle de rotation en radian (une fonction de conversion est présente dans le fichier des fonctions auxiliaires).
- ceil : arrondi à l'entier supérieur.

On peut ensuite calculer pour chaque pixel  $p(x, y)$  son équivalent  $r(x, y)$  correspondant à ses coordonnées dans la matrice de rotation avec les formules suivantes :

$$r_x = \text{ceil}(\cos(\theta) * (x - m_{xd}) + \sin(\theta) * (y - m_{yd}) + m_x)$$

$$r_y = \text{ceil}(-\sin(\theta) * (x - m_{xd}) + \cos(\theta) * (y - m_{yd}) + m_y)$$

*FIGURE 22 : formules de calcul des nouvelles coordonnées d'un pixel  $p(x, y)$ .*

Avec :

- r : nouvelles positions (x et y).
- m : milieu (x pour la longueur, y pour la hauteur, d si destination).
- $\theta$  : angle de rotation en radian (une fonction de conversion est présente dans le fichier des fonctions auxiliaires).
- ceil : arrondi à l'entier supérieur.

Cette opération se répète ensuite pour chacun des pixels de l'image, mais n'est exécutée que si les valeurs ne dépassent pas la matrice de rotation. Cette vérification est faite à l'aide de la condition suivante :

```

if (0 <= rotate_x
    && rotate_x < src_width
    && 0 <= rotate_y
    && rotate_y < src_height)

```

*FIGURE 23 : condition de vérification de la matrice de rotation.*

#### 4.6.2 Rotation automatique.

Il y a deux étapes majeures pour la rotation automatique : dans un premier temps, il faut calculer l'angle de rotation. Puis, dans un second temps, il faut effectuer la rotation avec l'angle trouvé. L'étape la plus complexe est sans aucun doute la détection de l'angle de rotation. Pour cela, une transformée de Hough a été effectuée. Celle-ci se décrit par l'algorithme suivant :

```

accumulateur = tableau la taille de l'image (2D).
pour chaque x dans longueur:
pour chaque y dans largeur:
si p(x, y) constitue une bordure (couleur):
    pour theta allant de 0 -> 180:
        rho = ceil(x * cos(theta) + y * sin(theta))
        accumulateur[theta][rho]++

```

FIGURE 24 : création de l'accumulateur de Hough.

Il ne suffit alors que de calculer l'index de l'angle le plus présent dans l'accumulateur, et d'effectuer une rotation de l'image par un angle  $\theta_r = 360 - 90 - \theta = 450 - \theta$  qui correspond à l'angle opposé par trigonométrie. En appliquant cette rotation sur notre image, le résultat suivant est alors obtenu :

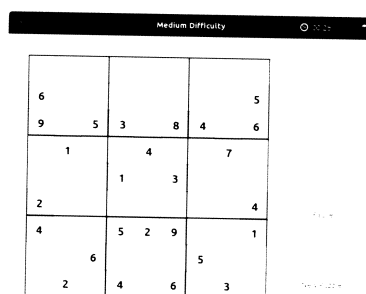


FIGURE 25 : image numéro 5 après rotation automatique.

Le message suivant apparaît au début de la rotation automatique :

Auto-Rotation: starting rotation with angle .

FIGURE 26 : message affiché au début de la rotation automatique.

#### 4.7 Détection et extraction de la grille.

Tout le processus de prétraitement terminé, il faut ensuite extraire la grille de sudoku de l'image. C'est-à-dire créer une nouvelle image ne contenant que cette dernière, afin par la suite d'obtenir une grille pouvant être traitée de manière efficace.

Il existe une multitude de manières d'effectuer cette extraction. En l'occurrence ici, il s'agit d'une extraction arbitraire : la détection commence par la recherche des abscisses

des quatres plus grandes lignes verticales de l'image. Le minimum de ces abscisse correspond conséquemment à la ligne la plus à gauche, qui devrait correspondre à une des extrémités du sudoku.

Il faut ensuite retrouver les valeurs du haut et du bas de la ligne, afin de pouvoir en calculer la hauteur, pour, ensuite, extraire le carré sachant que la grille de sudoku est un carré. On peut donc retrouver les étapes suivantes :

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

FIGURE 27 : grille extraite.

Il est ensuite possible d'extraire chacune des rangées de la grille par une simple division par 9.

1	4	7
---	---	---

FIGURE 28 : rangée extraite.

Cependant, cette méthode possède quelques points faibles, et elle est notamment mise à défaut dans le cas où il y a une plus grande ligne verticale à gauche qui ne constitue pas un côté de la grille de sudoku. C'est la raison pour laquelle un système de détection de grille par recherche de la plus grande composante connexe sera effectué afin de pouvoir prendre en compte tous les différents cas.

Toutes les rangées sont sauvegardées sous le format "RANG.jpeg"

Les messages de l'extraction de la grille sont d'ailleurs les suivants :

```
Grid Detection: sudoku edges found at: 1211  915  619  1519.
Grid Detection: found probable edge at x = 619
Grid Row Extraction: height found.
Grid Row Extraction: interval set to: 102
```

FIGURE 29 : messages affichés lors de l'extraction de la grille.

Comme expliqué plus tôt, ce modèle peut poser des problèmes : dans le cas où il y aurait une autre longue ligne verticale à gauche de la grille de sudoku, il est tout à fait possible que la détection précédente considère cette dernière comme étant le côté gauche de la grille de sudoku.

C'est la raison pour laquelle il a été nécessaire d'implémenter une détection de grille se basant sur la recherche de composantes connexes noires. Pour ce faire, l'algorithme utilisé est assez simple. Un carré est tracé au centre de l'image, avec une taille de côté étant proportionnelle à l'image, correspondant à une case un peu plus grande que celles du sudoku. Ces mesures sont effectuées de sorte à ce que les bords du premier carré possèdent tous des intersections avec au moins une des lignes du sudoku.

Il suffit ensuite uniquement d'augmenter la taille du carré, jusqu'à atteindre le premier carré de pixel blanc ininterrompu, le schéma de ce processus est le suivant :

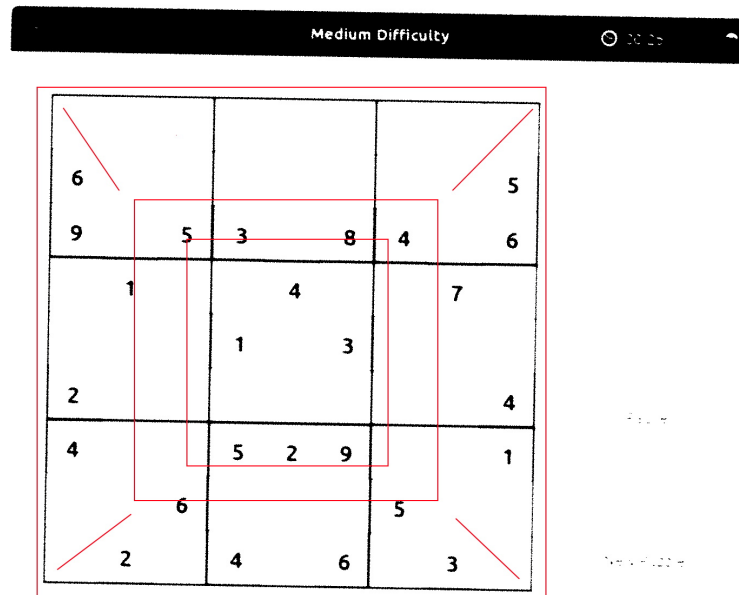


FIGURE 30 : image numéro 5 après rotation automatique.

#### 4.8 Détection et extraction des chiffres.

Après avoir extrait les rangées, il est très facile d'obtenir les chiffres individuels. Il suffit uniquement de passer à travers chacune d'entre-elles et à découper avec un interval égal au neuvième de la longueur de l'image recadrée (étant donné qu'il n'y a que 9 chiffres maximum par rangée).



FIGURE 31 : chiffre extrait.

Tous les chiffres sont stockés dans le dossier *numbers/* par ordre d'apparition, afin ensuite de pouvoir les manipuler par *Raw Major Order*. Le dossier ressemble alors à ceci :

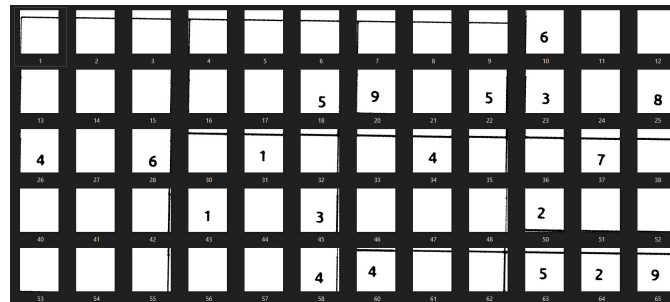


FIGURE 32 : dossier contenant les chiffres extraits.

#### 4.9 Réseau de neurones.

L'objectif pour cette première soutenance était de réaliser un réseau de neurones simple, capable d'apprendre la fonction XOR. C'est à dire :

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 33 : Table XOR.

Comme nous pouvons le voir, pour avoir un réseau de neurones capable d'apprendre la fonction XOR, il nous faut deux neurones en Input constituant notre *Input Layer* ainsi qu'un neurone de sortie (Output) constituant notre *Output Layer*.

En ce qui concerne notre *Hidden Layer*, le nombre de neurones est plus arbitraire. Nous avons trouvé cela judicieux d'en choisir deux, étant donné que le problème présenté est assez simple, et surtout pour ne pas surcharger notre réseau de long calcul (pas forcément pertinent dans notre cas de figure) qui le ferait perdre en efficacité.

Notre réseau ressemble donc au schéma suivant :

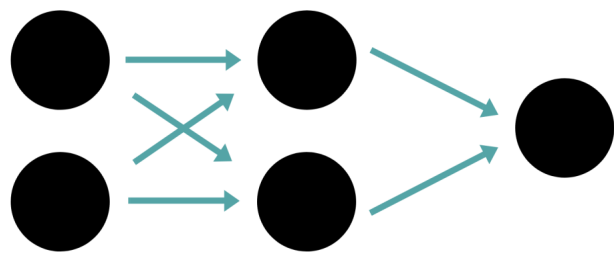


FIGURE 34 : Schéma du réseau de neurones utilisé.

Comment marche un réseau de neurones ?

Le neurone calcule la somme de ses entrées  $x$  pondérées par les poids  $w$ , avant d'appliquer la fonction d'activation  $\phi$ . Le résultat de cette somme sera ensuite passée à une fonction d'activation qui définira le résultat en Output obtenu. Autrement dit :

$$a_j = \phi(b_j + \sum (w_{ij}x_i))$$

Maintenant que nous avons expliqué le principe, nous allons nous attarder sur l'implémentation de notre réseau.

La première étape est d'initialiser nos poids et nos biais de manière aléatoire (avec des valeurs comprises entre 0 et 1). Nos deux neurones constituant notre *Inputs Layers*, se verront attribuer un poids reliant cette couche à l'*Hidden Layer*. Ce dernier, se verra également attribuer des poids le reliant à l'*Outputs Layers*. Et pour finir, notre neurone en sortie se verra attribuer un biais.

La deuxième étape est celle de l'entraînement.

Pour cette étape, nous faisons le calcul décrit précédemment, auquel nous appliquons la fonction d'activation Sigmoidale parfaite qui est de la forme suivante :

$$f(x) = \frac{1}{1 + \exp(-x)}$$

Cette fonction d'activation permet de voir l'évolution de notre réseau de neurones avec des résultats de plus en plus proches des valeurs attendues (contrairement à la fonction Marche Heaviside qui renvoie un résultat binaire, 0 ou 1).

Une fois avoir appliqué la fonction, nous pouvons effectuer la dernière étape, celle de **Backpropagation**.

Cette étape consiste à rééquilibrer les poids et les biais en fonction du taux d'erreur. Ce

taux d'erreur se calcule en faisant la différence entre le résultat attendu et le résultat en Output. Cette différence sera multipliée par l'application de la dérivée sur chaque valeur des neurones constituant l'*Hidden Layer*.



Pour rappel, la dérivée de la fonction Sigmoidale s'écrit sous la forme :

$$f'(x) = x * (1 - x)$$

Pour cette étape, notre réseau effectue 4 cycles de 50 000 entraînements. Ces chiffres sont totalement arbitraires mais nous permettent d'avoir des résultats très satisfaisants. Cependant, ils seront amenés à être modifiés au cours de ce projet afin de réduire le temps de la phase d'entraînement de notre réseau.

Les résultats obtenus pour 4 cycles de 10 000 entraînements sont déjà suffisamment intéressants pour notre projet.

#### 4.10 Construction de la grille.

Une fois le réseau lancé et entraîné, il va générer les prédictions des images des chiffres de la grille que nous stockerons dans un tableau à 1 dimension. Cela nous permet par la suite, de créer un fichier texte dans le format spécifié afin qu'il puisse être utilisé correctement lors de la résolution. Il écrira le chiffre correspondant si possible et un 0 le cas échéant.

#### 4.11 Chargement de la grille de sudoku.

Le programme se chargeant de la construction de la grille sera appelé afin de pouvoir avoir un fichier texte valide à fournir à notre programme solveur. Il lira un à un les caractères du fichier et tentera une conversion "from int to string" lorsqu'il tombera sur un chiffre. S'il lit un '.' alors il considérera cela comme un 0. Tout autre caractère sera ignoré (sauts de ligne et espaces). Finalement le programme enregistrera le fichier sous le nom "**grid\_00**".

```
$ ls
grid_00 solver
$ cat grid_00
... ..4 58.
... 721 ..3
4.3 ... ..

21. .67 ..4
.7. ... 2..
63. .49 ..1

3.6 ... ...
... 158 ..6
... ..6 95.
```

FIGURE 35 : Fichier texte contenant la grille à résoudre.

Le principe est simple, il s'agit d'une lecture de fichier. La méthode sera appliquée identiquement à chaque fichier étant donné que ceux-ci sont censés avoir une mise en forme similaire. La fonction d'import va lire un caractère à la fois dans le fichier et stocker la valeur numérique de celui-ci dans un tableau 2D quand cela est possible, c'est-à-dire lorsqu'elle lit un chiffre. Notons que les '.' correspondent à des cases vides (étiquetés 0 dans le tableau) et les sauts à la ligne témoignent d'un changement de ligne dans la grille du sudoku. Les lignes vides et les espaces entre les cases ne sont, quant à eux, présents que par pur souci de lecture et de visibilité pour le programmeur.

Ainsi à la fin, la fonction assigne chaque ligne du fichier à un tableau de dimension 1 dans le tableau final du sudoku.

#### 4.12 Résolution de la grille de sudoku.

Pour la résolution du sudoku, nous avons décidé d'opter pour la méthode du ***backtracking***. En plus d'être réursive, elle se veut complète et garantit un résultat unique pour chaque grille.

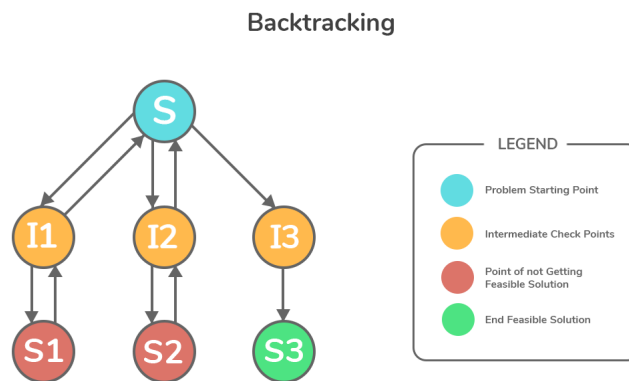


FIGURE 36 : Principe du backtracking.

Le programme solver prend en entrée un fichier, importe ce fichier dans un tableau 2D (fonction d'import) puis se charge de résoudre ce tableau. S'il existe une solution alors la grille initiale sera modifiée sinon, grâce au backtracking le programme retournera à la case de coordonnées (0,0) en ayant remis toutes les cases modifiées à 0 et le tableau demeurera donc inchangé par rapport à celui de départ.

```
$ ./solver grid_file
```

FIGURE 37 : Appel au programme solver.

Lorsque la résolution est terminée, le programme se charge d'exporter le tableau final sous forme de fichier texte. Le fichier sera alors sous la forme : **grid\_00.result** où **grid\_00**

est le fichier passé en entrée. Il aura la même mise en forme que le fichier d'import. Et ce quand bien même la grille serait insoluble dès le début. La résolution fonctionne donc parfaitement, que la grille soit résoluble ou non le programme renverra toujours une réponse.

```
$ ls
grid_00 grid_00.result solver
$ cat grid_00.result
127 634 589
589 721 643
463 985 127

218 567 394
974 813 265
635 249 871

356 492 718
792 158 436
841 376 952
```

FIGURE 38 : Fichier contenant la grille résolue (cf grille fig.31).

En plus du résultat stocké dans un fichier grid, nous avons également implementé une fonction qui permet de reconstruire la grille finale sous forme d'image. Notre image de sudoku final se veut simple de compréhension pour l'utilisateur. En effet, nous avons décidé de prendre en image de fond une grille pré-dessinée afin de coller des images de chiffres par dessus, dans chaque case.

Nous avons préalablement fourni à notre programme un dossier d'images de taille 70x70 qui representent les chiffres de 1 à 9. Dans ce dossier se trouvent 2 sets identiques pour seule différence la couleur des chiffres. Ceux en noir seront utilisés pour les éléments déjà présents dans la grille initiale, tandis que ceux en bleu représenteront les chiffres placés après résolution.

1

FIGURE 39 : Image de chiffre initial

1

FIGURE 40 : Image de chiffre final

## 4.13 Réseau de neurones complet et fonctionnel

### 4.13.1 Jeu d'images pour l'apprentissage

Dans un premier temps, le réseau de neurones avait été entraîné à partir de la base de données MNIST. La base de données MNIST pour Modified ou Mixed National Institute

of Standards and Technology, est une base de données de chiffres écrits à la main. C'est un jeu de données très utilisé en apprentissage automatique. Cependant, cette dernière est, comme indiquée, uniquement composée de chiffres manuscrits numériques. Ces derniers ne sont dès lors non représentatifs de la réalité. Effectivement, les grilles de sudoku sont pour la majorité constituées de chiffres numériques imprimés par un ordinateur. Ainsi, la base de donnée MNIST n'a pas été utilisée.

Nous avons donc, à la place, crée notre propre système de dataset. Il s'agissait ici d'un programme Python, permettant de générer un nombre défini d'images de chiffres, crée à partir des fichiers de polices de caractères .ttf. Ont été rajoutées les possibilités d'ajouter du bruit, ainsi que de changer la position du chiffre, et son angle de rotation. Dans notre situation, le réseau de neurones a été entraîné sur plus de 350 000 images, afin d'obtenir les résultats les plus précis possibles.

#### 4.13.2 Apprentissage

Pour détecter des nombres avec une fonction softmax et un réseau neuronal, il faut d'abord entraîner le réseau neuronal sur un ensemble de données de nombres étiquetés. Cela implique généralement de fournir au réseau neuronal de nombreux exemples de nombres et les étiquettes correspondantes, afin que le réseau apprenne à reconnaître les modèles dans les données.

Une fois le réseau neuronal formé, vous pouvez utiliser la fonction softmax pour effectuer des prédictions sur de nouvelles données. La fonction softmax est un choix courant pour les tâches de classification, car elle produit une distribution de probabilité sur les classes possibles. Dans ce cas, les classes possibles seraient les nombres que le réseau neuronal a été formé à reconnaître.

Pour utiliser la fonction softmax, il faut d'abord faire passer la sortie du réseau neuronal par la fonction softmax pour obtenir la distribution de probabilité prédite. On choisit ensuite le nombre ayant la probabilité prédite la plus élevée comme étiquette prédite pour les données d'entrée. La fonction softmax est définie telle que :

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

#### 4.13.3 Sauvegarde et chargement des poids du réseau de neurones

Une fois que le réseau de neurones a été entraîné, il est ensuite possible de sauvegarder tous les poids du réseau de neurones, dans un fichier présent dans le dossier "ocr neural network". Ce fichier est ensuite rappelé à chaque étape de la prédiction neuronale, on possède ainsi un array contenant les 81 chiffres prédits par le réseau de neurones.

## 4.14 Interface graphique

L'interface de notre application se veut être simple, pratique et efficace, l'objectif étant de réussir à réaliser une plateforme facile et accessible afin de convenir au plus grand nombre d'utilisateurs.

Pour réaliser cette interface nous avons utilisé Glade, un outil interactif de conception d'interface graphique GTK. Notre interface glade est directement reliée à notre fichier *main.c* permettant d'associer des fonctions à différents boutons. Pour lier les deux types de fichiers, la première étape de notre fichier *.c* est de récupérer les informations concernant notre builder (c'est à dire notre *.glade*).

Etant donné que notre builder contient toutes les paramètres importantes concernant l'ID de nos boutons, de nos images et d'autres widgets, nous avons opté pour l'idée de créer une classe que nous nommons *SGlobalData*. Cette structure se définit de la manière suivante : grid file.result

```
typedef struct
{
    GtkBuilder *builder;
    gpointer user_data;
}SGlobalData;
```

FIGURE 41 : Structure associée

Afin de rendre l'expérience utilisateur la plus agréable possible, nous avons décidé d'agir sur les boutons en les faisant apparaître ou disparaître selon le besoin, et ainsi alléger le contenu de notre interface de sorte à ne pas perdre notre utilisateur.

Le principe est simple : il suffit de jouer sur la dualité hide/show sur les GtkWidget :

```
gtk_widget_show(GTK_WIDGET(gtk_builder_get_object(data->builder, "SolveButton")));
gtk_widget_hide(GTK_WIDGET(gtk_builder_get_object(data->builder, "SaveButton")));
```

FIGURE 42 : fonctions de gtk permettant de show/hide les boutons "Solve"/"Save"

La première étape du processus est de Load l'image contenant le Sudoku que l'utilisateur souhaite résoudre. Pour mettre en évidence cette étape, lors de l'ouverture de notre application, le bouton "Load" est le seul disponible au centre de notre menu principal (ainsi que notre logo).

En outre, chaque bouton a une fonctionnalité qui permet de suivre la résolution du sudoku. Lorsque l'utilisateur appuie sur le bouton "Load", il se présente à celui-ci une interface de chargement de fichier, afin qu'il lui soit permis de sélectionner l'image qu'il souhaite résoudre. Ensuite, d'autres boutons aux fonctionnalités bien distinctes apparaissent :

```
Solve
# Resolution du sudoku

Quit (quit.png)
```

```

# Quitter le logiciel SUDOCODE

Grayscale
# Application du grayscale sur l'image

Gamma
# Application du gamma sur l'image

Rotate
# Rotation automatique de l'image

Threshold
# Binarisation de l'image

Correct
# Correction de la grille si non solvable

```

De surcroît, d'autres widgets ont été ajouté afin de permettre à l'utilisateur d'accéder à de nombreuses fonctionnalités sur la résolution du sudoku, ainsi que d'autres paramètres pour la démonstration du réseaux de neurones.

```

Label : "Please enter x,y,value"
# Indication sur l'entree de la correction

Label : "Manual rotation angle input"
# Indication sur la possibilite de choisir l'angle manuellement

GtkScale Slider
# Curseur permettant de choisir l'angle manuellement

GtkEntry Entry
# Entr e pr dispos e afin de permettre la correction

```

Lorsque l'image est chargée, tous les boutons sont affichés. Si l'utilisateur clique sur "Solve", la résolution s'opère et d'autres widgets apparaissent, afin qu'une correction puisse être appliquée si le sudoku n'est pas solvable.

Voici l'apparence de l'interface, avec tous les widgets de départ :

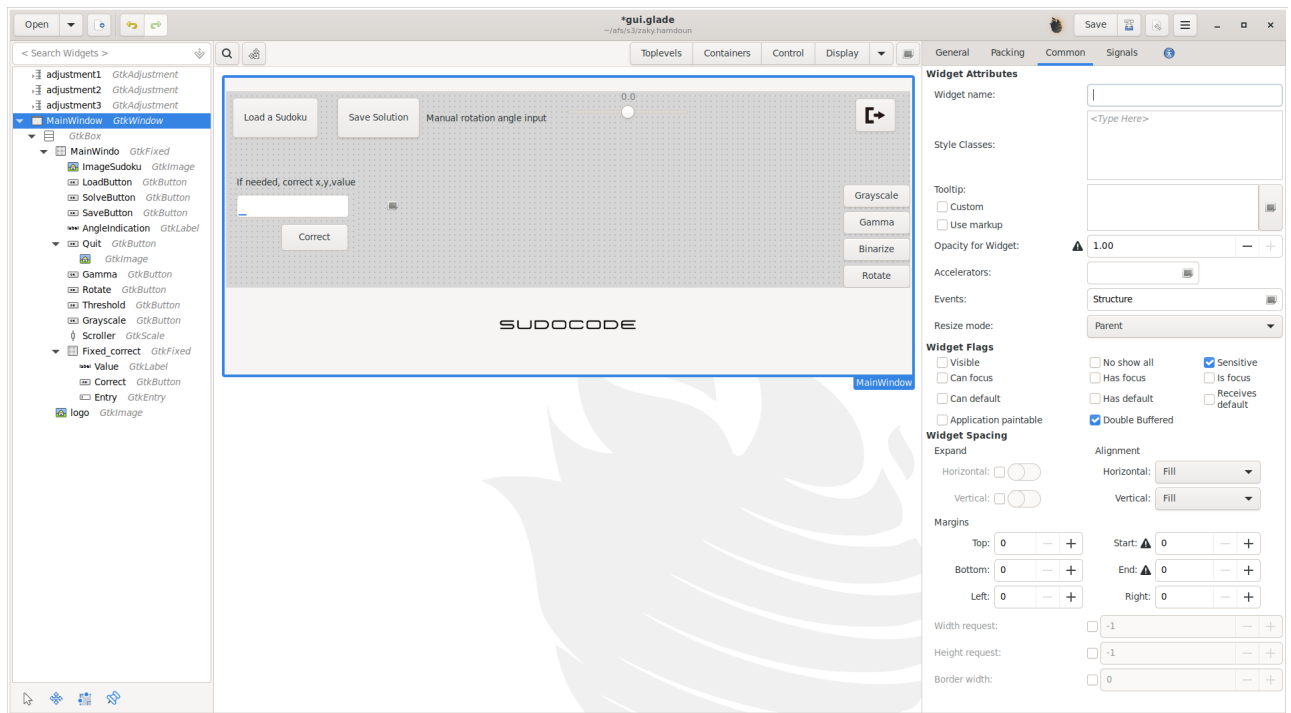


FIGURE 43 : Logiciel sur lequel l'interface est créée

Ouverture du logiciel :

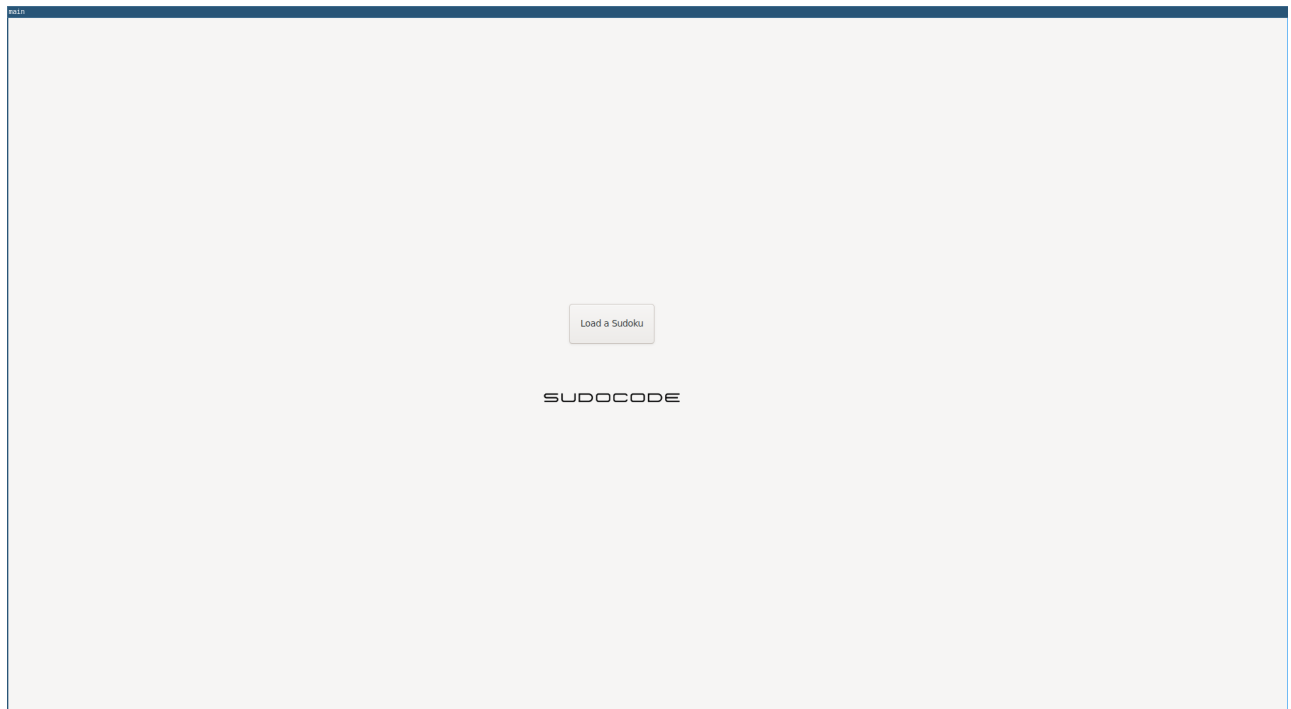


FIGURE 44 : Main menu de l'interface

- Clic sur le bouton "Load". Interface de sélection de fichiers :

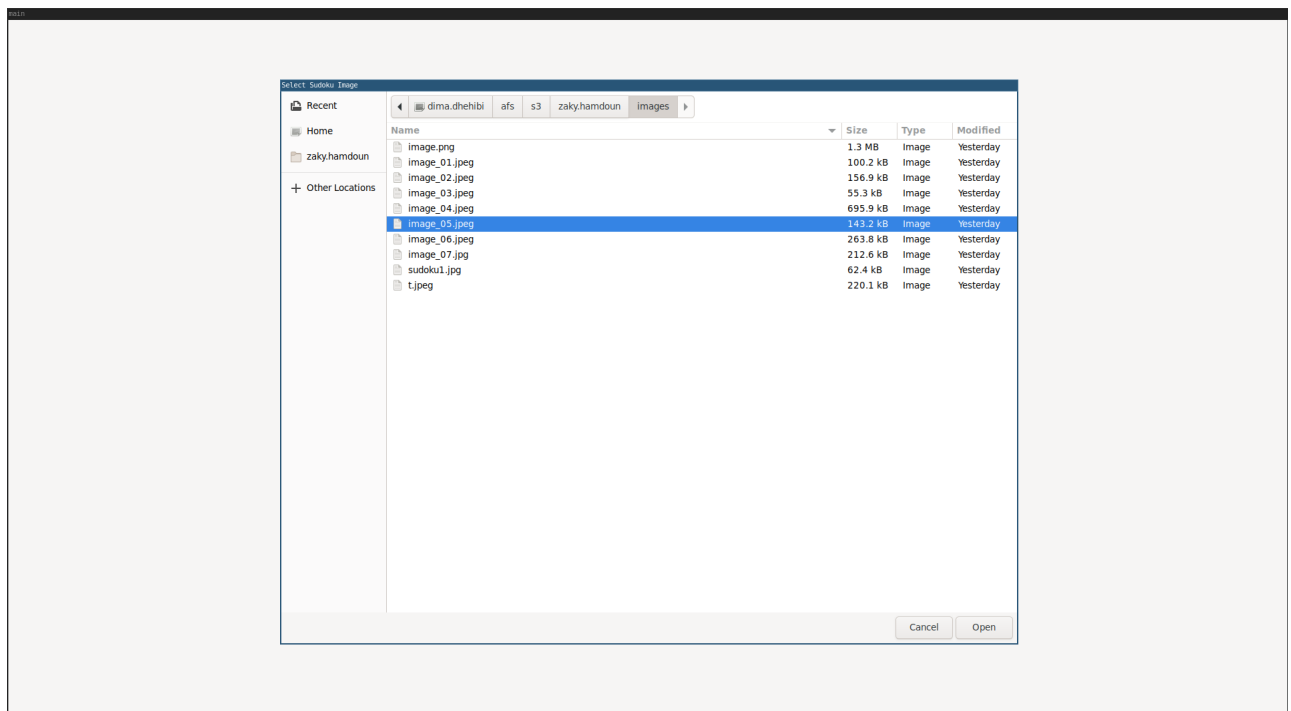


FIGURE 45 : Sélection d'un fichier .png

Placement de l'image sur le container du widget GtkImage :

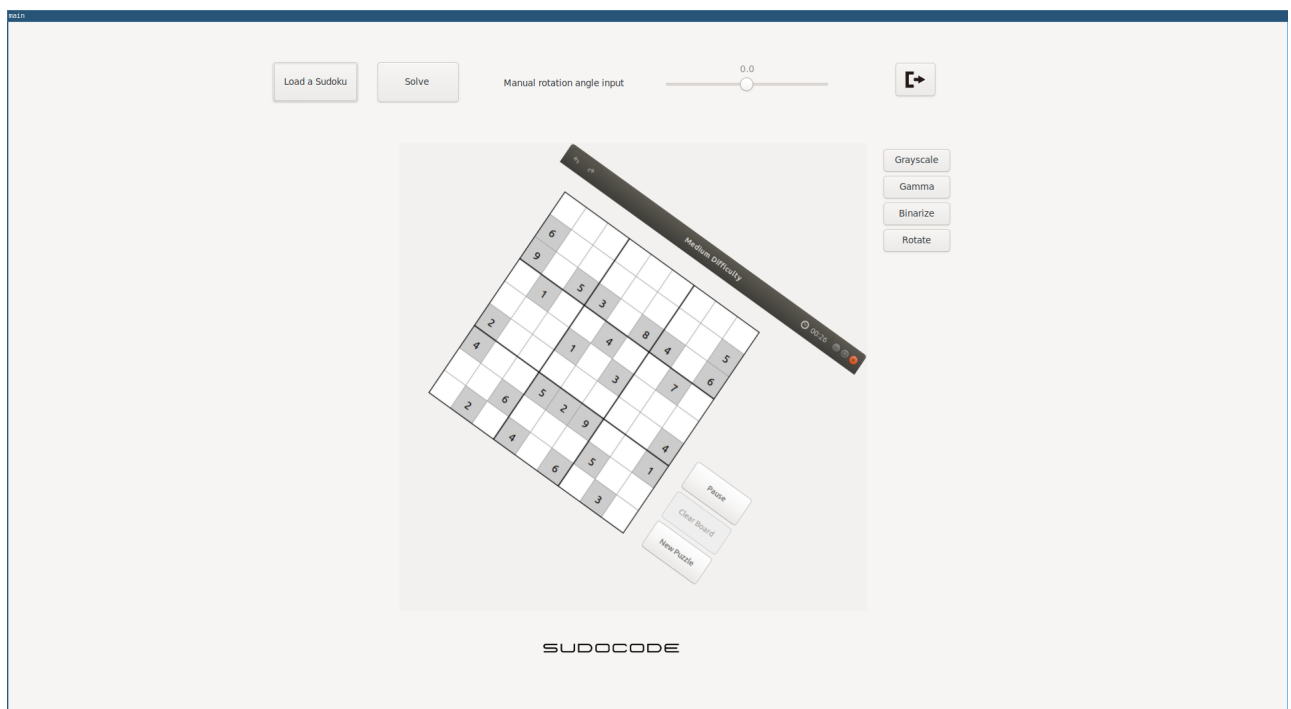


FIGURE 46 : Chargement de l'image

On peut suivre tous les changements de solve qui vont être effectués sur l'image afin de résoudre le sudoku importé.



- Clic sur le bouton "Grayscale"

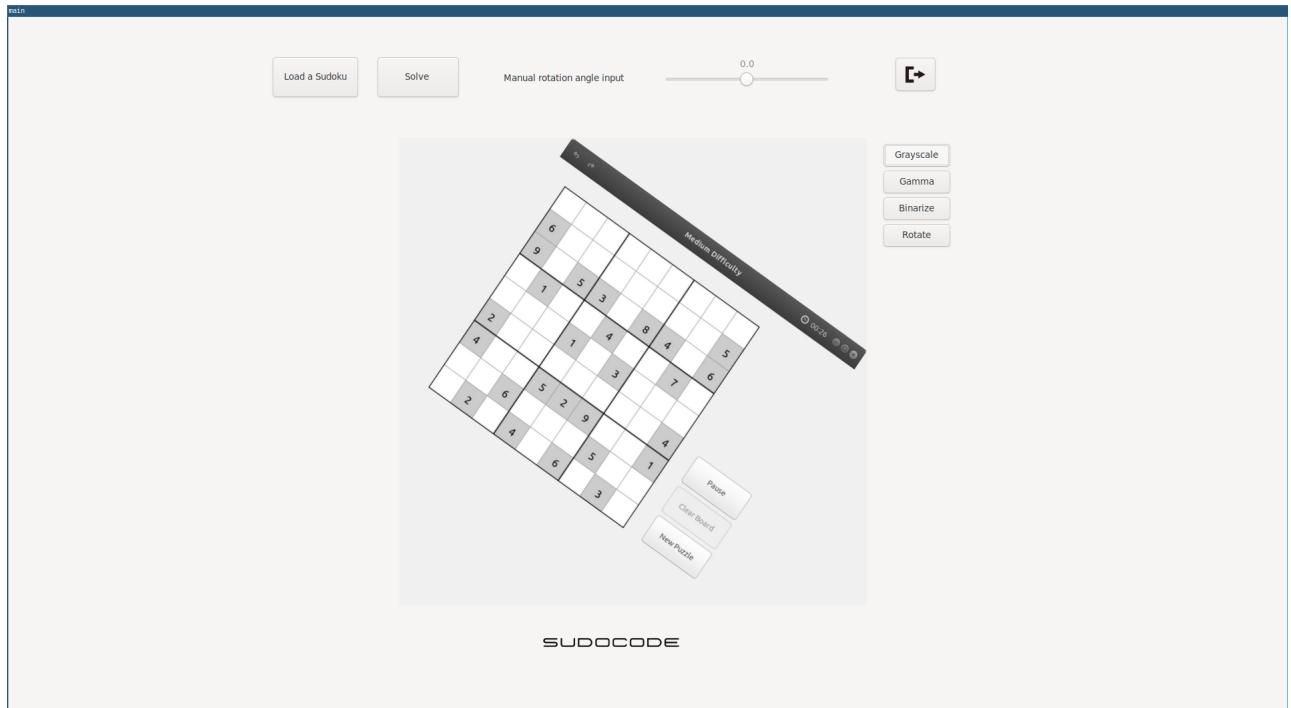


FIGURE 47 : Application du grayscale

- Clic sur le bouton "Gamma"

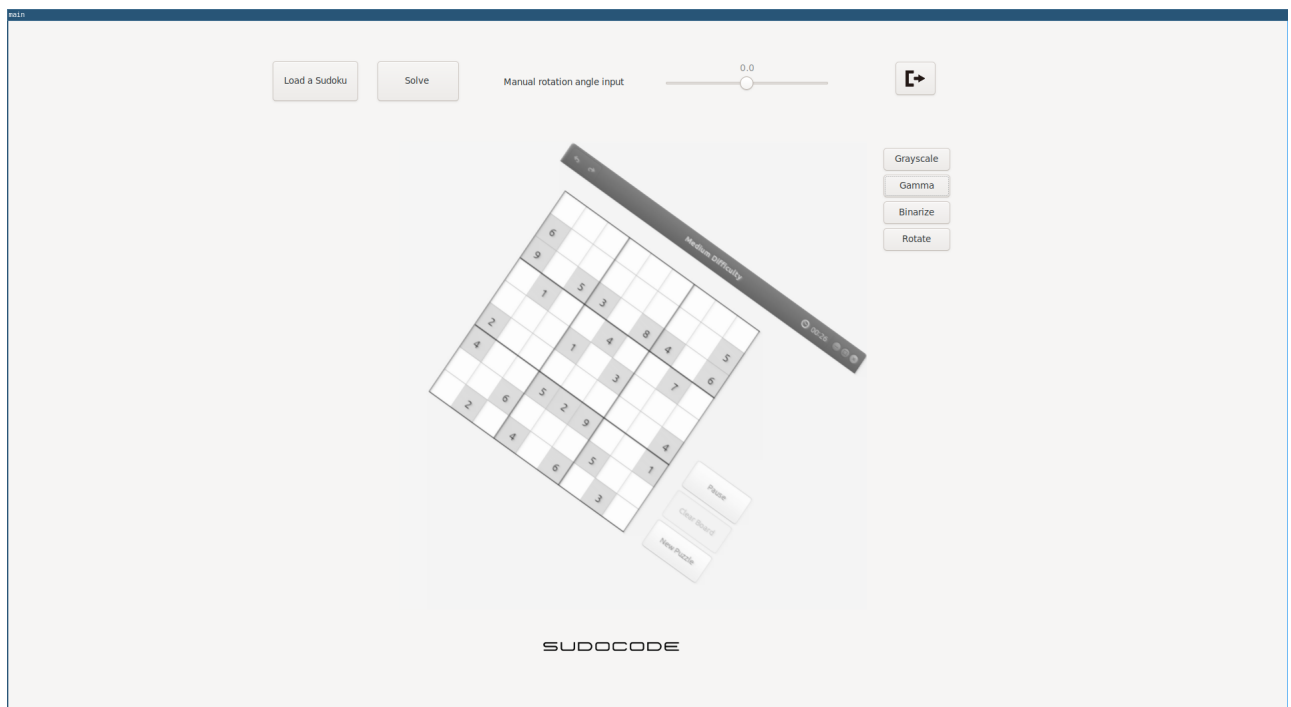
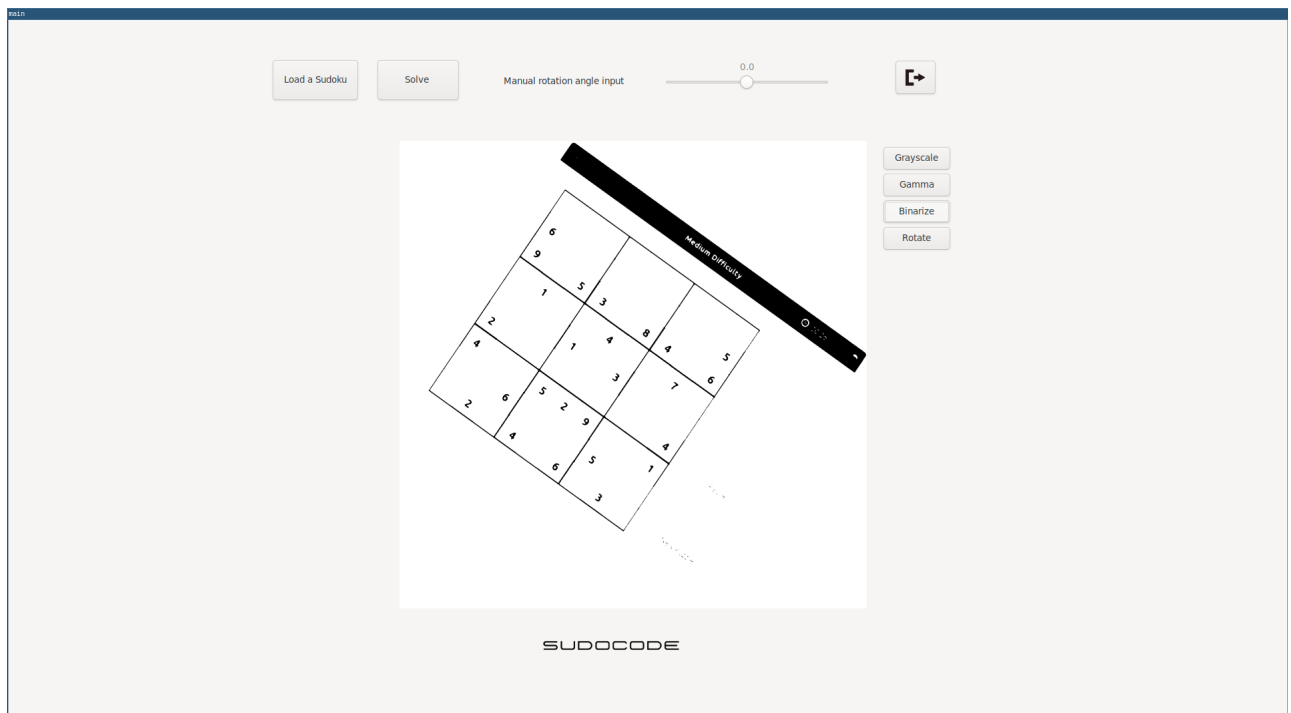


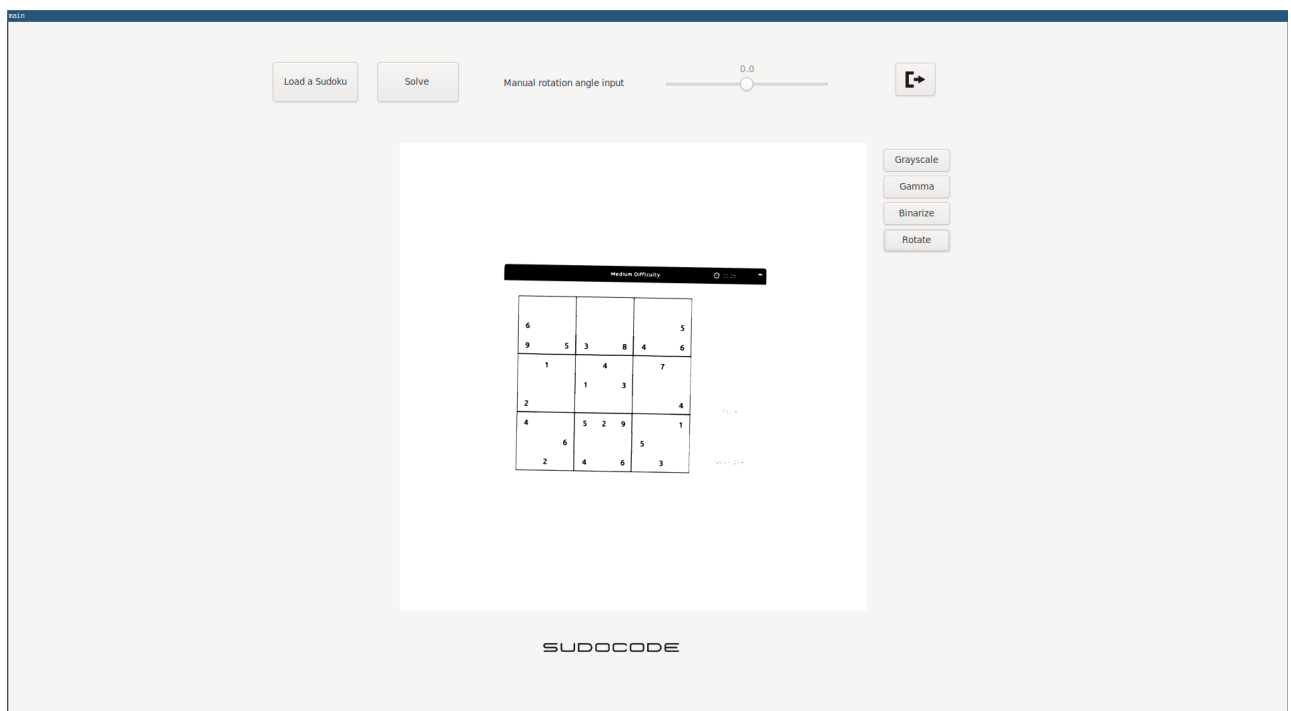
FIGURE 48 : Application du gamma

- Clic sur le bouton "Binarize"



### *Application de la binarisation*

- Clic sur le bouton "Rotate"



*FIGURE 49 : Application de la rotation*

- Clic sur le bouton "Solve". Il y a enfin l'apparition du champ de correction.

Il nous donc bien en effet possible de corriger les valeurs des cases calculées par le réseau de neurone, si celle-ci ne sont pas justes, en l'occurrence, si elles produisent un

sudoku irrésolvable, afin de palier aux erreurs issues réseau de neurones.

Pour ce faire, nous avons mis à disposition une entrée à l'aide d'un GtkEntry nous avons connecté celle-ci aux fonctions associée à la production d'une case à partir de la valeur entrée.

Cette entrée n'est bien évidemment à utiliser uniquement si le sudoku rendu n'est pas résolvable. Une fois cette étape de correction réalisée, il est enfin possible de voir le résultat de la résolution du sudoku après avoir appuyé sur le bouton "Solve".

Les nombres affichant la résolution complète du sudoku d'affichent en bleu, et permettent ainsi une meilleure lisibilité. Une entrée permet à l'utilisateur de produire ces corrections.

## 5 Conclusion.

Pour finir sur ce projet, nous avons donc réalisé une application permettant de résoudre un sudoku. Cette application permet à l'utilisateur de pouvoir résoudre son un sudoku en l'intégrant directement à notre plateforme sous format image.png. L'interface de notre application se veut être simple d'utilisation afin de convenir au plus grand nombre d'utilisateurs. Notre projet permet également de voir les différentes étapes de prétraitement avant de le lancer notre programme.

Pour le moment, nous avons donc un main ainsi qu'un Makefile permettant d'effectuer les opérations de chargement et de sauvegarde des images tests, leur mise en nuance de gris, ainsi que leur prétraitement, en d'autres termes, le redressement automatique et manuel de l'image, l'élimination des bruits parasites par la méthode gaussienne ainsi que le renforcement des contrastes.

De plus, nous avons également mis en application les processus de détection de la grille et des cases. Enfin, l'algorithme de résolution de la grille et la sauvegarde de celle-ci, résolue, sont des étapes que l'on peut considérer accomplies.

Les réalisations futures considérées pour le bon déroulement du projet seront la binarisation et l'extraction de la grille plus optimales. Finalement, les tâches auxquels nous nous attèlerons sont un prétraitement ainsi qu'un réseau de neurones complets et fonctionnels, comportant l'apprentissage et la reconnaissance des chiffres de la grille. Il est donc nécessaire de finaliser la reconstruction de la grille, l'affichage de celle-ci et puis la sauvegarde du résultat obtenu sous forme d'image. Ces opérations devront être accessibles depuis une interface graphique permettant d'utiliser tous ces éléments.