Rapport de soutenance no.1

A Pouception Creation

Hugo Baptista
Romain Doulaud
Camille Chapelle
Félix Coste

2022 - 2023

Contents

1 Introduction

Nous allons vous présenter l'avancement sur notre projet dans le cadre de la première soutenance. Avant de plonger dans le vif du sujet, voici le tableau de répartition des tâches avec leur avancement.

1.1 Répartition des tâches

Table 1: Répartition des tâches

	Hugo	Romain	Camille	Felix
Binarisation de l'image				X
Découpe de la grille		X		
Réseau de neurones	X			
Résolution du Sudoku			X	
UI			X	

Dans un premier temps, nous verrons l'ensemble des modules qui ont été avancés, en commençant par le pré-traitement de l'image, puis par la découpe de la grille ainsi que la reconnaissance des chiffres et pour finir nous parlerons de la résolution de la grille et l'UI. A chaque fois, nous noterons l'avancement, les problèmes rencontrés et nos solutions.

Table 2: Planification de l'avancement

	Soutenance 1	Soutenance finale
Binarisation	80%	100%
Découpe de la grille	80%	100%
Réseau de neurones	50%	100%
Résolution du Sudoku	100%	100%
UI	20%	100%

Pour ce qui est des prédictions de l'avancement des tâches, nous les avons bien respectées pour la plupart. Seule la partie UI n'a pas encore été mise en application cependant nous avons déjà réfléchi à la manière dont nous allons l'implementer ainsi qu'au design de l'interface.

2 Le Prétraitement de l'image

La première étape du projet est d'effectuer un prétraitement sur l'image afin que celle-ci soit traitable pour les étapes futures.

2.1 Méthode de binarisation

La binarisation de l'image consiste à supprimer les couleurs afin de ne garder que le noir et le blanc dans une image.

Tout d'abord, la première étape est de passer l'image en grayscale, c'est-à-dire uniquement en tons de gris. Pour ce faire, nous avons utilisé le code du tp 6 d'Informatique pratique, nous apprenant à nous servir de SDL et à grayscale une image. Cette méthode de grayscale consiste à récupérer la composante rgb de chaque pixel séparément en Uint8, et d'y appliquer la formule : $0.3 * r + 0.59 * g + 0.11 * b$ (r représentant la composante rouge, g la verte et b la bleue). Cette formule nous renvoyait alors un Uint32, servant de nouvelle composante rgb au pixel. On l'appliquait alors à l'image et le grayscale était créé.

Une fois le grayscale réalisé, il fallait transformer l'image en pixels noirs et blancs uniquement.

Afin de réaliser une binarisation, il nous faut une valeur seuil. Ce seuil (threshold) sera le point de passage entre un pixel blanc et un pixel noir sur l'image, la fameuse limite cherchée afin que l'image puisse rester exploitable.

Nous avons d'abord essayé de réaliser une binarisation avec la valeur moyenne des pixels, la valeur maximum étant 255,puis nous divisions par 2 afin de récupérer le premier seuil. Cette méthode s'est toutefois avérée inutile, car l'image restait complètement inexploitable.

La deuxième méthode essayée fut d'implémenter cette moyenne de couleur des pixels de manière locale. Nous divisions alors l'image en 5 sur la longueur et 5 sur la largeur et faisions la moyenne du niveau de gris du carré. Puis nous colorions chaque pixel dont la valeur de gris est inférieure à ce seuil en noir, et les autres en blanc. Faute de Segfault récurrents et de grandes difficultés à implémenter cette méthode dû au grand nombre de pixels dans les images, il a été décidé que cette méthode sera optimisée et réutilisée par la suite, afin d'obtenir un meilleur résultat sur l'algorithme d'Otsu.

Enfin, après de nombreuses recherches,nous avons trouvé 2 algorithmes de binarisation d'image : la méthode de Sauvola et celle d'Otsu. Nous avons décidé d'implémenter l'algorithme d'Otsu car cette méthode semblait plus facile.

La méthode d'Otsu consiste à trouver le seuil idéal afin de séparer binariser l'image. Il faut donc commencer par récuperer les valeurs de chaque pixel d'une image grayscale, puis de construire un histogramme à partir de ces valeurs. A partir de cet histogramme, il faut trouver la valeur idéale du seuil. Cette valeur peut être trouvée lorsque la variance de l'histogramme calculée à partir de cette formule $\sigma(t) = \omega1(t)\omega2(t)(\mu1(t) - \mu2(t))$ est maximale. $\omega1$ correspond à l'espérance de toutes les valeurs, c'est-à-dire la somme de chaque valeur (de 0 à 255) multipliée par son nombre d'occurences dans l'histogramme. $\omega2$ la somme de toutes les espérances dans les valeurs de 0 à t. $\mu1$ correspond quant à elle à la valeur moyenne du cluster 1 (des valeurs de 0 à t), c'est à dire $\omega1/n1$ où n1 correspond à la somme de toutes les valeurs de l'histogramme de 0 à t. $\mu2$ correspond donc à la valeur moyenne du cluster 2 (des valeurs de t à 255), c'es-à-dire $(\omega1 - \omega2)/n2$ où n2 correspond à la somme des valeurs de l'histogramme allant de t à 255.

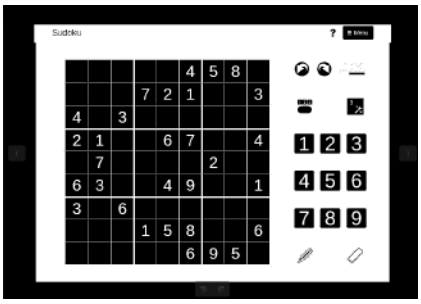

Figure 1: Résultat de la binarisation de l'image 3

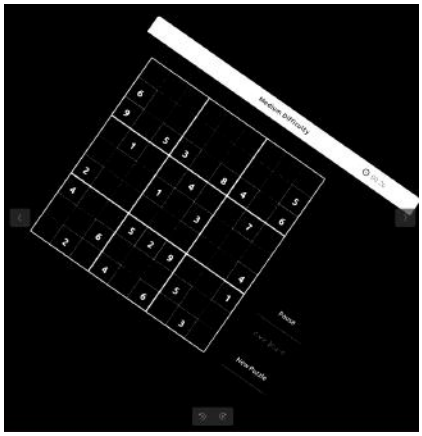

Figure 2: Résultat de la binarisation sur l'image 5

Pour chaque valeur de t allant de 0 à 255, nous effectuons donc le calcul de cette variance, et la valeur maximale est choisie.

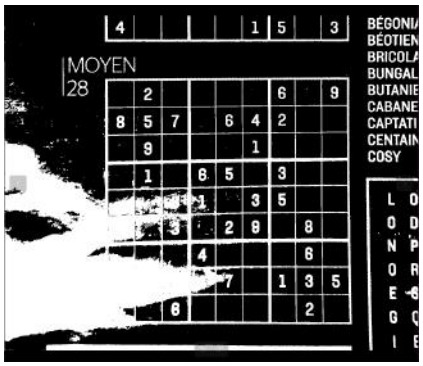

Figure 3: Résultat de la binarisation sur l'image 2, avec quelques fioritures

2.2 Ce qui doit être amélioré

Comme nous pouvons le voir sur la figure 3, l'algorithme d'Otsu a un peu de mal à binariser l'image lorsqu'elle est couverte de bruit ou autre. Ainsi, pour la prochaine soutenance, nous prévoyons d'améliorer cet algorithme afin de l'exécuter en local sur l'image, comme ce qui avait été tenté au début du projet. De plus, si tout ceci ne suffit pas, un filtre Gaussien sera implémenté afin de réduire drastiquement le bruit sur l'image. Nous pourrons également envisager un downscale de l'image, afin de simplifier les calculs.

3 La Découpe de la grille

3.1 Détéction des lignes

Afin de détécter la grille du sudoku, nous avons utilisé la méthode du transfomé de HOUGH.

Cette méthode utilise le fait qu'une droite peut être représentée de plusieurs façons différentes: la première étant la répresentation la plus commune en coordonnées cartesiennes: y=ax+b. La seconde étant en coordonnées polaires: $\rho = x * cos(\theta) + y * sin(\theta)$. Grâce à cela, nous pouvons ainsi créer deux répères géometriques pour répresenter la même chose. Le premier est un répère cartésien avec les conventions habituelles: x en abscisse et y en ordonnée. Tandis que le second est un repère en coordonées polaires avec en abscisse: un angle theta (θ), et en ordonnée: une distance rho (ρ).

Ces deux repères sont très liés l'un l'autre. En effet, un point dans le répère cartésien réprésente une courbe sinusoïdale dans le répère polaire. Réciproquemenent, un point dans le polaire répresente une droite dans le cartésien.

Ainsi nous pouvons initialiser le répère cartesien comme une liste à deux dimensions représentant les pixels de l'image à analyser. Dès lors, lorsqu'on parcourt l'image pixel par pixel afin de trouver ceux qui ont été mis en valeur lors du processus de binarisation, nous allons pouvoir tracer sa courbe

correspondante dans le repère polaire. Or gâce au transformé de Hough, toutes les courbes sinusoïdales correspondantes aux points situés sur une même ligne dans le repère cartesien vont se croiser en un point dans le repère polaire. Ce point précis de coordonnées theta et rho va donc correspondre à une des lignes du quadrillage du sudoku.

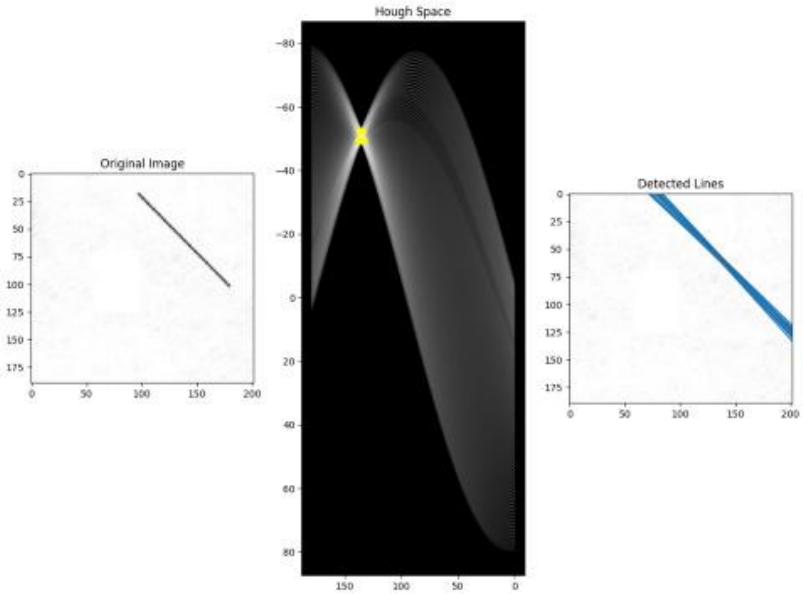

Figure 4: Transformé de Hough

Une fois l'entièreté de l'image analysée, il arrive régulièrement qu'on obtienne beaucoup de droites très similaires pour une seule et même ligne de l'image originale (voir figure 2). Nous avons donc trié toutes les droites afin d'en isoler une seule pour chaque ligne du quadrillage.

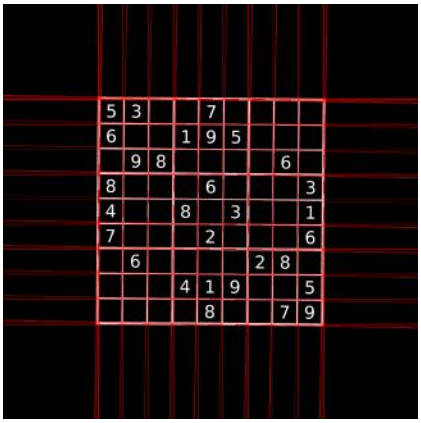

Figure 5: Toutes les droites

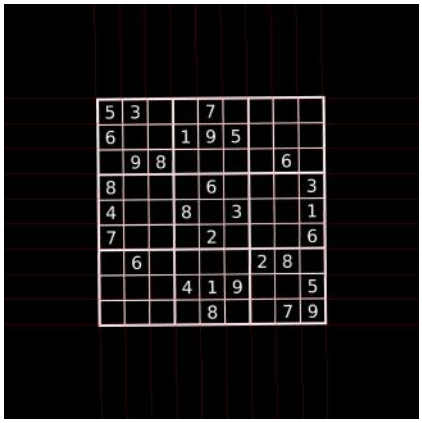

Figure 6: Isolation des droites

3.2 Rotation de l'image

Nous avons supposé qu'avant que l'image passe dans tout le processus de binarisation, l'utilisateur a redressé l'image de manière assez vulgaire. C'est à dire que la grille de sudoku a été orientée vers le haut mais n'est pas parfaitement droite (l'angle maximal avant la rotation étant de plus ou moins 45°). La fonction de rotation permet ainsi de recalibrer la grille de manière très précise. Pour ce faire, nous avons calculé la moyenne de la valeur de l'angle theta de chacune d'entre elles afin d'obtenir une valeur de rotation de l'image.

3.3 Détections des intersections

Maintenant que l'on a une seule droite correspondante à chaque ligne de la grille de sudoku, il nous faut trouver les intersections de celles-ci afin de pouvoir les découper par la suite. Pour ce faire, nous avons créer une liste à deux dimensions remplie de zéros de la taille de notre image, ayant subi ou non une rotation. Ensuite, pour chaque ligne, nous l'avons tracée dans cette liste c'est-à-dire que pour chaque point (x, y) de la droite, nous avons incrémenté de 1 la valeur à l'indice [x][y] dans la liste. Ainsi, lorsque deux droites se croisent, une valeur dans la liste aura été incrémentée deux fois ce qui nous révèlera la position exacte de l'intersection de ces deux droites.

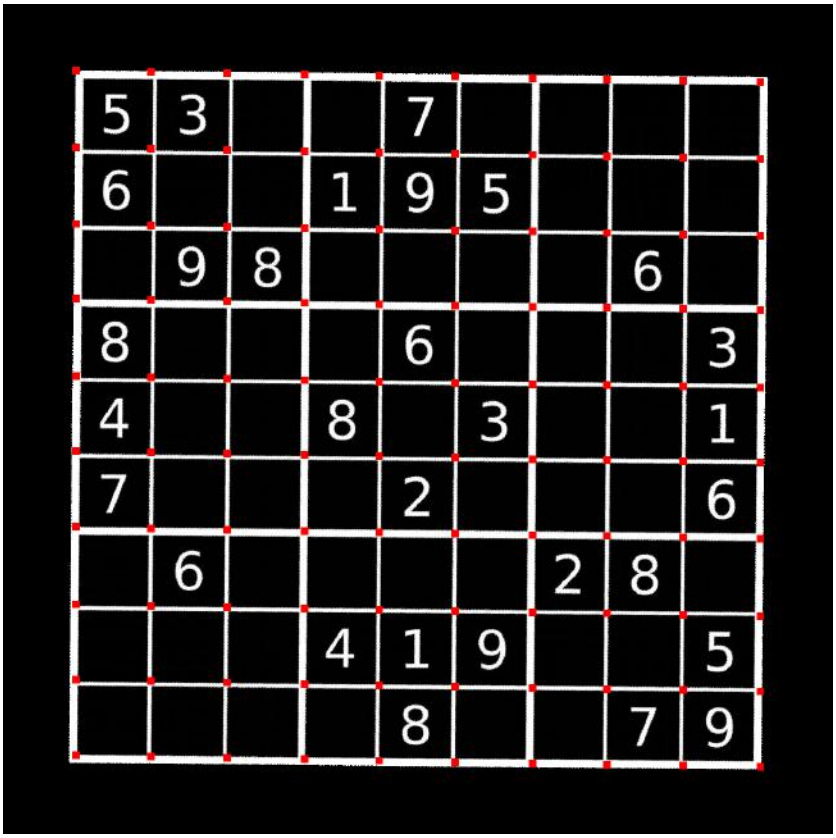

Figure 7: Intersections des droites

3.4 Découpe des cases

Dans le cas de la figure 5, toutes les intersections ont été détéctées (100 au total) et dans ce cas là, nous sommes dans la situation la plus simple pour la suite du processus. En effet, si on connait déjà la position de toutes nos intersections et qu'aucune autre ligne parasite n'a été détéctée autour du Sudoku, découper les cases est un jeu d'enfant. En effet, pour extraire une partie d'une image, il faut créer un rectangle avec comme position celle du bord supérieur gauche de la case et comme dimensions celles de la case en question. Ainsi, il suffit d'itérer à travers les 81 intersections situées en haut

à gauche d'une case afin de découper cette dernière pour l'envoyer à l'étape suivante qui n'est autre que la reconnaissance des chiffres.

NB: Pour le moment, la découpe des cases ne marche que lorsque toutes les 20 lignes sont détéctées et par conséquent, les 100 intersections aussi. La prochaine étape est donc de pouvoir trouver les intersections manquantes lorsque quelques lignes n'ont pas été reconnues avec le transformé de HOUGH.

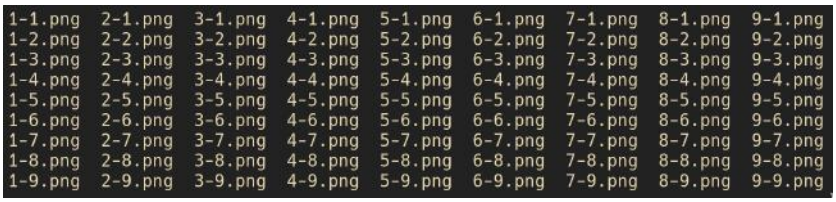

Figure 8: Toutes les cases découpées

4 La reconnaissance des chiffres

La reconnaissance de chiffres permet de convertir l'image en une grille de texte que le solver peut lire et résoudre. Notre objectif est de convertir une liste de pixels en un nombre de 1 à 9.

Nous avons remarqué que les pixels qui constituent un nombre sur une image varient en fonction de la police d'écriture, du placement du pixels sur l'image et de la qualité de l'image, il est donc impossible d'écrire un algorithme qui compare juste les pixels. Pour pallier ce problème, nous faisons appel à un réseau de neuronnes qui utilise des couches intermédiaires de calculs en créant des relations entre des combinaisons de pixels.

4.1 Les matrices

Les calculs qui prennent place dans un réseau de neuronnes sont majoritairement matriciels. Alors nous avons réaliser une librairie en C pour gérer tous les calculs matriciels. Ceci nous a permis de développer rapidement des tests sans avoir à penser à la structure des données (car gérée par la librairie matrix).

4.2 Preuve de concept

Avant de se lancer directement dans la reconnaissance de caractère, nous avons écrit notre premier réseau de neuronnes pour qu'il puisse résoudre le problème du XOR. En effet, il existe beaucoup de problèmes de classification qui sont solvables linéairement. Cependant, le cas du XOR est un cas où on ne peut pas tracer une seule droite pour séparer les 0 des 1 (cf. image)

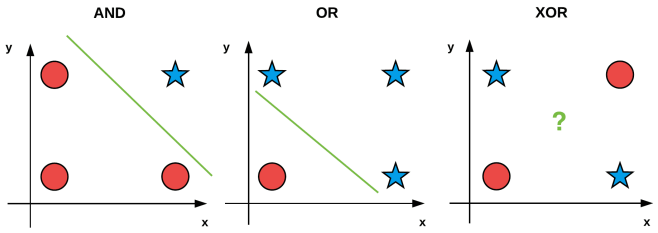

Figure 9: AND et OR linéairement séparable, XOR problème non linéaire

4.3 La classification

Pour le XOR, nous n'utilisons qu'un neuronne de sortie, car le problème est binaire. Cependant, pour notre problème initial, qui est de convertir des pixels en 1 chiffre, nous utilisons exactement 10 neuronnes de sortie. Parmis ces 10 neuronnes, 9 d'entre-eux permettent de classifier le nombre de 1 à 9.

Le dixième neuronne sert pour les cases vides. En effet, lors de la découpe des cases, nous ne faisons pas la distinction d'une case vide ou remplie, alors c'est au réseau de lever l'indétermination.

Finalement, le réseau de neuronnes sans fonction d'activation sur la dernière couche, peut nous donner des valeurs qui ne sont pas comprises entre 0 et 1. Or nous souhaitons que le réseau nous donne un pourcentage indiquant quel chiffre il est le plus confiant d'identifier. Pour cela, nous utilisons la fonction softmax qui produit en sortie des probabilités par neuronnes.

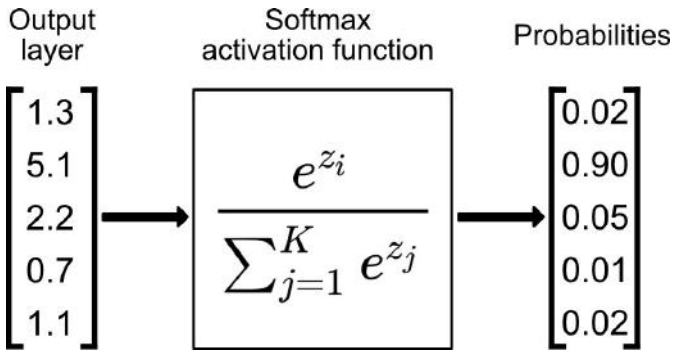

Figure 10: Exemple de Softmax

4.4 Les problèmes rencontrés

Nous avons rencontrés multiples problèmes lors de la conception de notre premier réseau de neuronnes. La phase d'apprentissage de ce dernier fût la partie la plus complexe du développement. En effet, malgré le fait que le sujet soit très documenté, chacun a sa méthode d'implémentation de l'apprentissage. Nous avons alors choisi pour le XOR d'utiliser une seule couche intermédiaire constituée de 4 neuronnes (car c'est beaucoup plus rapide à entrainer), mais aussi d'utiliser la fonction d'activation sigmoid partout car ReLU et Softmax ne donnaient pas de bon résultat. Finalement, nous avons choisi d'entraîner le réseau sur l'entièreté du jeu de données et de lui faire apprendre sur la moyenne de ses erreurs, car nous avons remarqué

qu'entrainer le réseau à chaque élément ne lui permettait pas de généraliser et donc il n'apprenait pas, ou mal.

Maintenant la preuve de concept réalisée avec le XOR, nous pensons utiliser plutôt ReLU comme fonction d'activation pour les couches intermédiaires ainsi que SoftMax sur la couche finale. Ce ne sont pas des choix définifs car nous planifions également d'implémenter un algorithme de normalisation, permettant de conserver les valeurs des poids basses. En effet sans normalisation, il y a un risque que la valeur des poids dépasse la limite des float en C, ce qui provoque des NaN qui corrompt l'entièreté du réseau.

5 La résolution du Sudoku

Pour résoudre le sudoku, nous avons utilisé une méthode connue qui s'appelle le backtracking, déjà vu l'année dernière lors d'un des TP de programmation, où on devait faire un solveur de sudoku. Le principe est très simple: il faut que notre algorithme arrive à placer les nombres de 1 à 9 sur la ligne, la colonne et le carré de 3 par 3. Pour ça, en utilisant le backtracking, il faut que sur chaque case vide, après avoir vérifié qu'on pouvait bien y placer un nombre, on y place un nombre puis on rappelle récursivement la fonction jusqu'à une des deux possibilités arrive : soit on ne trouve pas de solution et on repart en arrière, soit on a trouvé une solution au sudoku et on renvoie la grille résolue. Pour nous faciliter les choses, l'algorithme a été découpé en 3 partie : une qui récupère le ficher de la grille et qui le transforme en array d'integers, une autre qui résout la grille et une dernière qui enregistre la grille dans un fichier .result.

La partie qui a posée le plus de problèmes n'était pas le solver mais comment transformer le fichier en array, à cause de la manipulation d'arrays. Au début tout allait bien, on crée un pointeur sur le fichier et on enregistre chaque ligne dans un array en deux dimensions. Avant de faire le solveur, l'idée de renvoyer un array de charactères paraissait être le meilleure, car il ne fallait pas changer de type, mais après nous avons dû changer et renvoyer un array de nombres. L'étape d'après est d'enlever les espaces et de mettre les nombres dans un array final que nous envoyons au solver, seulement nous avons dû faire face au choix de la dimension de l'array qui n'était pas la

bonne au début et qui renvoyait un stack smashing. Le problème a été réglé en testant plusieurs valeurs jusqu'à comprendre lesquelles fonctionnaient et pourquoi. Seulement, par habitude, on considérait qu'une ligne finissait par un charactère nul sauf que l'algorithme renvoyait des valeurs qui n'avaient rien à voir avec la grille, il a fallu beaucoup d'incompréhension et de temps pour réaliser que c'était un retour à la ligne.

Pour la partie du solveur, nous nous sommes renseignés sur internet pour faire un algorithme optimisé basé sur le backtracking, ainsi que le tp de l'année dernière. Comme expliqué précédemment, le but est de tester sur une case vide, ici les points ont étés changés en 0, les valeurs de 1 à 9, et pour chaque test, rappeler récursivement la fonction sur la grille avec la valeur qu'on veut tester. Si à la fin aucune solution a été trouvée, on passe à la valeur suivante.

Pour finir cette partie, il fallait enregistrer la grille solution dans un fichier, sans oublier de remettre des espaces entre les blocs de 3.

Dans le dossier git, on peut y trouver des grilles de sudoku pour tester le solveur, certaines sont faciles et d'autres sont difficiles.

Figure 11: Résultat final

6 L'interface Utilisateur (UI)

La partie de création de l'interface n'a pas été commencé mais les recherches ont débutées et l'UI sera fait avec la librairie GTK.

7 Conclusion

Pour conclure, nous sommes dans les temps par rapport au cahier des charges. Notre organisation du git nous permet de ne pas gaspiller du temps sur des conflits, chaque dossier possède une branche et lors de la mise en commun, tout est merge et assemblé sur master. Chaque partie est conçue comme une librairie, pour pouvoir être importée et utilisée pour l'application finale, mais aussi conçue pour fonctionner en comme un utilitaire dans le terminal. Ainsi, la conception du logiciel de reconnaissance et de résolution de Sudoku sera beaucoup plus rapide.

Pouce-ment 👍