

O.C.R

Arthur Oldrati | Terence Degroote | Kerian Allaire | Benoit Burg

24 Octobre 2021

Rapport de Soutenance 2

Table des matières

1	Introduction	4
1.1	Présentation de l'équipe	4
1.1.1	Arthur OLDRATI	4
1.1.2	Kerian ALLAIRE	4
1.1.3	Terence DEGROOTE	5
1.1.4	Benoit BURG	5
1.2	Répartition des tâches	6
2	Pré-traitement de l'image	6
2.1	Les filtres	6
2.1.1	Filtre nuance de blanc et noirs	7
2.1.2	Flou de Gauss	8
2.1.3	Flou Médian	10
2.2	La Binarisation et mise en valeur des contrastes	10
2.2.1	Méthode d'Otsu	10
2.2.2	Méthode d'otsu adaptative	10
2.2.3	Résultat en noir et blanc	13
2.3	Rotation	14
2.3.1	rotation manuelle	14
2.3.2	rotation automatique	14
2.4	Filtre de Canny	15
2.4.1	Principe	15
2.4.2	Pré-requis	15
2.4.3	Gradient d'intensité	16
2.4.4	Détection des maximum locaux	16
2.4.5	Nettoyage de l'image	17
3	Reconnaissance de la grille	18
3.1	La transformé de Hough	18
3.2	Détection de la grille	20
3.3	Formater nos cases	20
4	Réseau de neurones	22
4.1	Variable	22
4.2	Fonctionnement du réseau	23
4.2.1	Résolution	23

4.2.2	Entraînement	24
4.3	Sauvegarde des valeurs et test du réseau	26
5	Résolution	27
5.1	Sauvegarde et traitement des fichiers	27
5.2	Algorithme de résolution : Backtracking	28
6	Image de retour	29
7	Exécutable	30
8	Suivi du projet	39
9	Conclusion	40

1 Introduction

L'OCR est le projet du troisième semestres consistant à résoudre un sudoku via la lecture d'une photo de celui-ci. Pour cela il est nécessaire de traiter l'image pour la rendre exploitable, puis de reconnaître le sudoku sur celle-ci en utilisant des algorithmes tel que *Hough Line* ou encore *Canny*. En enfin de reconnaître les nombres allant de 1 à 9 grâce à un réseau de neurones, pour ensuite pouvoir résoudre la grille et l'afficher.

1.1 Présentation de l'équipe

1.1.1 Arthur OLDRATI

Bonjour ! Je suis dans le F.C HUMBLE pour ce projet qu'est l'OCR, puisque c'est un trait qui me définit parfaitement.

Ce projet est l'occasion pour moi d'améliorer ma maîtrise du C en me confrontant à divers Segfault :)

De plus les projets sont une source de motivation pour moi aussi j'affirme que ce projet sera une réussite

L'OCR en lui même me plaît beaucoup car en effet j'ai intégré l'EPITA en étant intéressé par l'IA, ce projet me permet donc de me rendre compte de ce que cela peut être.

Je vais cependant pour cette première soutenance m'atteler au traitement d'image et notamment la reconnaissance de la grille et des cases de notre sudoku.

1.1.2 Kerian ALLAIRE

Je suis Kerian Allaire, étudiant en deuxième année d'Epita. j'ai beaucoup apprécié les différents projets de l'année passée, que cela soit l'Afit, qui m'a permis de comprendre ce qu'était qu'un projet, de part le principe de deadline, de rendu intermédiaire, et nous a également appris à avoir une certaine régularité dans notre travail. Le second projet m'a également beaucoup plu. En effet, celui-ci nous a laissé bien plus de liberté, nous permettant ainsi de nous diriger dans des directions intéressantes que nous n'aurions peut-être pas découvert avec un projet plus dirigé. Ces projets nous ont également appris à chercher par nous-même, à découvrir de la documentation sur

des concepts scientifiques et informatiques que nous n'avions pas forcément vu dans le cadre d'Epita. L'OCR m'intéresse beaucoup étant donné que je suis intéressé par le concept d'IA, et le traitement d'image m'était assez nébuleux mais me semblait pertinent. J'ai donc hâte de continuer ce projet et d'enrichir mes connaissances dans les différents domaines cités plus haut.

1.1.3 Terence DEGROOTE

Pour ma part, je suis le benjamin d'une fratrie de trois garçons. J'ai pratiqué beaucoup de sport durant ma jeunesse(notamment le rugby pendant 7 ans). Je suis venu à EPITA après avoir choisi l'option ISN(Informatique et Science du Numérique) au lycée Alain. Je me suis alors découvert un plaisir dans l'informatique, j'ai aussi été influencé par mon grand frère qui lui aussi est actuellement à l'Epita(en 4ème année). Je souhaite à travers ce projet continuer ma formation dans les sciences de l'ingénieur en faisant un projet totalement nouveau pour moi qui n'ait jusqu'à présent jamais programmé en C. J'ai toujours été intrigué par la reconnaissance de texte et je suis donc très intéressé dans le projet de ce semestre qui je l'espère va m'apprendre les bases d'un OCR.

1.1.4 Benoit BURG

Je suis Benoit Burg, étudiant en B2 cette année et en E2 l'année dernière. L'OCR est un projet qui me passionne beaucoup notamment la partie réseau de neurones puisque je suis très intéressé par tout ce qui rapproche de l'IA. Je suis très motivé et j'espère bien réussir ce projet et faire encore mieux que le projet de S2 ou bien celui du S1. J'essaye également d'apprendre à me servir de *VIM* correctement pour améliorer mon efficacité... même si pour l'instant, les résultats sont loin d'être ceux attendus !

1.2 Répartition des tâches

Les tâches pour chacun des membres seront repartie de la façon suivante :

	Arthur	Terence	Kerian	Benoit
Chargement d'une image et suppression des couleurs		X	X	
Rotation manuelle de l'image		X		X
Rotation automatique de l'image			X	
Découpage de l'image	X		X	
Lecture des cases de Sudoku	X			
Détection de la grille et position des cases	X		X	
Réseau de neurone				X
Solver				X
Exécutable				X

TABLE 1 – Répartition du travail

2 Pré-traitement de l'image

2.1 Les filtres

Afin de détecter des éléments dans un image, il est nécessaire que celle-ci soit la plus propre possible, c'est-à-dire que le bruit et tout élément qui pourrait entraver la détection de la grille de sudoku soit supprimer, ou le cas échéant rendu insignifiant. Pour cela, nous utilisons des filtres. Un filtre est une fonction mathématique qui, une fois appliquer aux valeurs RGB de chaque pixel de l'image, va donner à ces derniers de nouvelles valeurs RGB. Pour appliquer tous les filtres dont nous avons besoin, nous commençons par utiliser un filtre de nuance de gris, un **grayscale**.

2.1.1 Filtre nuance de blanc et noirs

Le filtre pour mettre en nuances de gris une image en couleur est possible de différentes façon, la première que l'on risque probablement sélectionner et de récupérer les 3 composantes de couleurs de chaque pixel et de créer sa nouvelle couleur en gris d'intensité égale à la moyenne des 3 composantes rouges bleues et vertes. Une seconde et d'applique la formule de $0.3*r + 0.59*g + 0.11*b$. Ce filtre grayscale est moins intéressant pour l'OCR puisqu'il embellit la photo mais n'est pas représentatif de la grille de sudoku notamment si elle est dans les teintes bleu comme c'est le cas de la photo numéro 3.

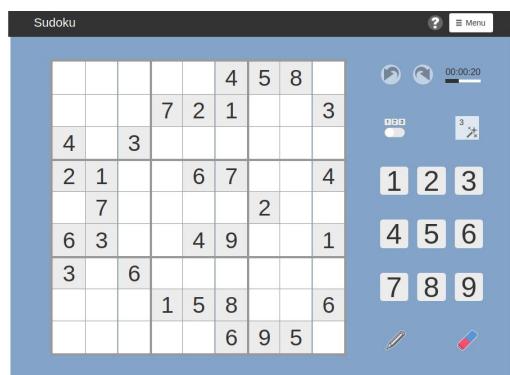


FIGURE 1 – sans filtre gris

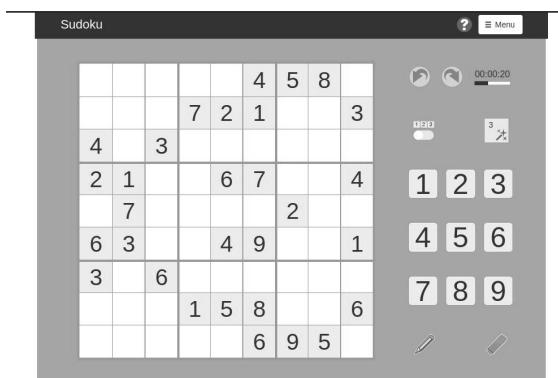


FIGURE 2 – avec filtre gris

Une fois l'image en nuance de gris, nous pouvons appliquer différents filtres qui vont nous permettre d'éliminer les pixels seuls et le bruit ; en bref, les éléments qui peuvent poser problèmes pour la binarisation, étape que nous verrons juste ensuite. Pour ces filtres, nous passons un kernel, qui est une matrice plus ou moins grande, sur la totalité de l'image. C'est-à-dire que pour plaçons le centre du kernel sur le pixel que nous étudions actuellement, et nous associons de différentes manières les valeurs contenues dans le kernel et celle se trouvant à leur place "équivalente" dans la matrice de l'image. A l'aide de l'association des valeurs, nous calculons des nouvelles valeurs RGB pour chaque pixel, ce qui a pour effet de modifier l'aspect de l'image.

2.1.2 Flou de Gauss

Le flou de Gauss est un filtre qui floute l'image. Cela permet de réduire le bruit d'une image en la floutant, mais aussi et surtout de supprimer une partie importante des pixels isolés. Le kernel a pour effet de récupérer les valeurs dans un diamètre de la taille du kernel, de les multiplier par les valeurs du kernel, avant de faire une moyenne de celle-ci. Le kernel en question possède une matrice exponentielle, c'est à dire que plus l'on s'approche du centre, plus les coefficients contenus dans le kernel sont grands. Cela permet de donner beaucoup plus d'importance aux points les plus proches du centre dans la moyenne. Cela à pour intérêts d'avoir une meilleure conservation des contours, ce qui est important pour la détection du sudoku. De plus, il existe des kernels de différentes tailles pour appliquer le flou de Gauss, amenant des flous plus ou moins précis. On peut ainsi retrouver des boutons permettant d'appliquer un flou avec un kernel de taille 3, de taille 5 et de taille 7. Une fois cela effectué, notre image est débarrassée d'une partie du bruit.

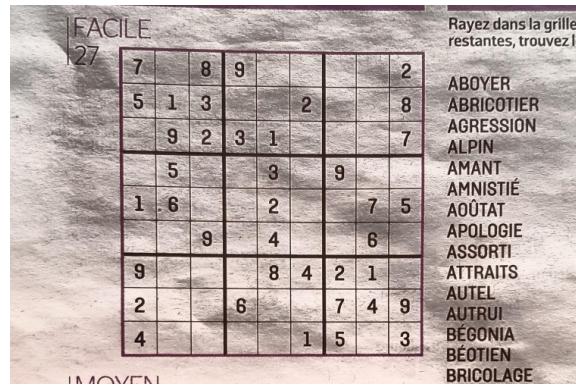


FIGURE 3 – Sans filtre



FIGURE 4 – Avec 4 flou de gauss de rayon 3

2.1.3 Flou Médian

Un deuxième flou que nous utilisons dans notre traitement d'images est le flou median. Celui-ci est particulièrement efficace pour effacer le bruit "sel" et "poivre" de l'image. Couplé à des flous de gauss, il est particulièrement efficace, et permet ainsi une très bonne détection des lignes. Son fonctionnement diffère du flou de Gauss : le kernel, au lieu de faire une moyenne des valeurs des pixels avec des coefficients, va simplement trouver les valeurs RGB médiane des pixels entourant celui qui nous intéresse (ainsi que celui-ci), avant de l'assigner au pixel que nous traitons. Un désavantage conséquent de cette méthode est la disparition des lignes trop fines ou trop clair, c'est pourquoi nous le couplons avec le flou de Gauss qui va lui élargir les lignes, permettant leur conservation pendant la suppression du bruit.

2.2 La Binarisation et mise en valeur des contrastes

Une fois les filtres appliqués, nous allons chercher à mettre en valeur les contrastes de l'image. Pour cela, nous binarisons l'image à l'aide d'algorithme permettant une détection plus intelligente qu'une simple comparaison avec une valeur minimale.

2.2.1 Méthode d'Otsu

Nous avons ensuite chercher un moyen pour rendre notre image en noir et blanc c'est à dire soit des pixels d'intensité 255 pour les 3 composantes (affichant blanc) soit 0 pour les noirs. Pour se faire il nous fallait établir un seuil à partir duquel on bascule le pixel en blanc ou en noir. Nous avons alors trouvé la méthode d'Otsu permettant de calculé le seuil optimal en fonction de chaque photo. Nous avons donc calculer la variance de chaque seuil ($0 \leq t < 256$) et la variance complémentaire ($t \geq 255$). Nous sélectionnons ensuite le seuil pour lequel la variance inter classe est la plus petite. Cela permet de rester précis que l'image soit plus sombre ou plus clair.

2.2.2 Méthode d'otsu adaptative

Après de nombreux test sur différentes images, nous nous rendons compte que les résultats d'Otsu ne sont pas parfait nous nous sommes

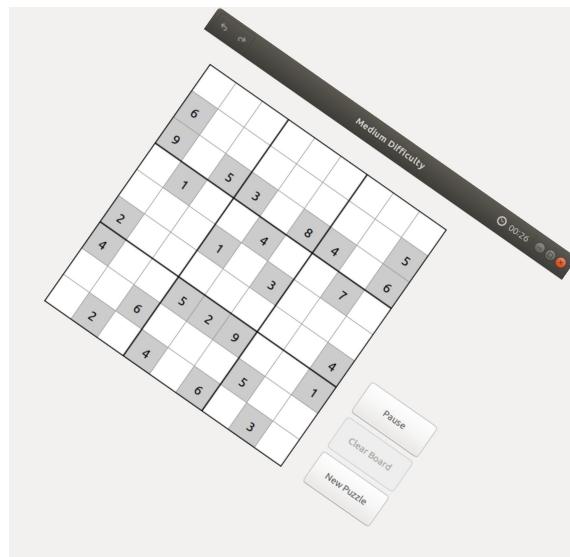


FIGURE 5 – Sans filtre

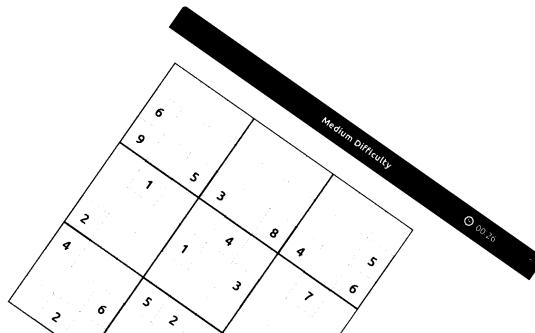


FIGURE 6 – Avec binarisation

donc pencher sur une méthode d'otsu plus dynamique qui permet d'avoir un seuil différents en fonction de l'endroit de l'image et du seuil de nettetés définis en argument (un seuil plus bas va afficher plus de bruit , parfait pour les images propres, un seuil plus haut est tout de même nécessaire lorsque la photo a une faible intensité. Ce seuil est calculé grâce a un seul parcours de l'image, de ce fait les données sont stockées dans une listes et déstocker au fur et à mesure de l'avancement sur l'image. Permettant ainsi une complexité très correctes de l'ordre de nombre de pixel opérations. Les résultats observés ci-dessous sont très correcte mais nécessite tout de même un 2ème argument pour le seuil de netteté.

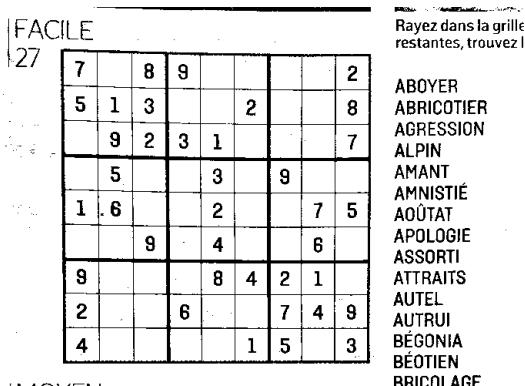


FIGURE 7 – Avec filtre adaptatif

2.2.3 Résultat en noir et blanc

Après avoir utilisé la méthode d’Otsu classique on applique la fonction de banalisation qui permet de vérifier pour chaque pixel sont appartenance a sa classe. Pour la méthode adaptative on change le pixel directement dans la boucle en même temps que le calcul du seuil pour chaque pixel.

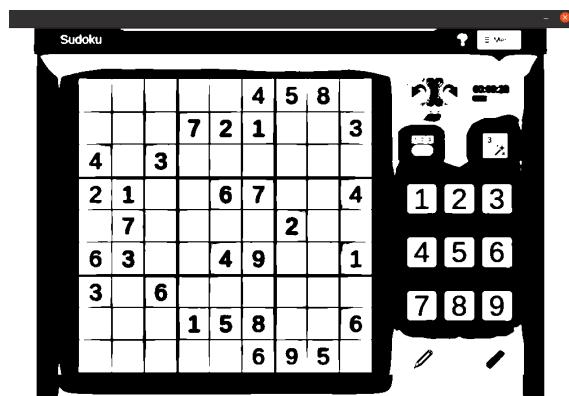


FIGURE 8 – résultat final

	4				1	5	3	

MOYEN

28

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

BEGONIA
BÉOTIEN
BRICOLA
BUNGAL
BUTANIE
CABANE
CAPTATI
CENTAIN
COSY

L O
O D
N P
O R
E S
C C

FIGURE 9 – résultat final

2.3 Rotation

2.3.1 rotation manuelle

La rotation manuelle d'une image s'effectue à l'aide d'une pression du bouton rotation après la sélection d'un angle via la fenêtre se situant à côté de celui-ci. Voici un exemple d'une rotation de 45°.



FIGURE 10 – Sans rotation

Cette fonction utilise la fonction *rotozoom* de la librairie de *SLD-gfx*, et permet donc d'effectuer très facilement une rotation et un zoom.

2.3.2 rotation automatique

Afin d'effectuer une bonne détection des grilles, et surtout d'envoyer des chiffres au réseau de neurone ayant un aspect compréhensible pour celui-ci, il est impératif que notre sudoku ne soit pas incliné. Pour cela, lors de la détection des lignes par la méthode d'Hough Line (cf 3.1), nous comptons le nombre de droite (dépassant un certain seuil) ayant chaque angle d'inclinaison. nous cherchons ensuite parmi celle ayant un angle compris 0 et 90 degré (car nous cherchons à replacer le sudoku droit, c'est à l'utilisateur à l'aide de la rotation manuelle de placer celui-ci plus ou moins droit) des droites au nombres d'au moins 10, avant de vérifier qu'il existe également au moins 10 autres droites inclinés à 90 degré par rapport à celle-ci. En effet, un sudoku est composé d'au moins 10 lignes verticales et 10 horizontales. Nous

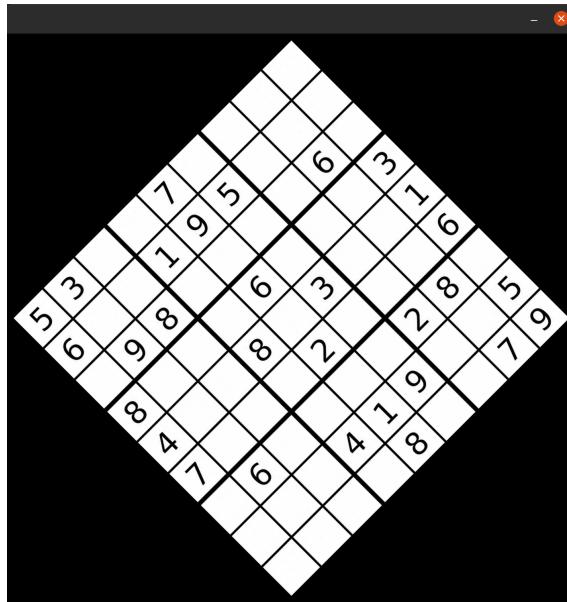


FIGURE 11 – Avec rotation de 45°

avons ainsi trouver l’angle d’inclinaisons de notre sudoku, et il ne reste plus qu’à appliquer rotozoom afin de réajuster l’orientation de l’image.

2.4 Filtre de Canny

Ce filtre prend place après la binarisation de l’image. Il permet de ne garder que les contours de l’image, et est la dernière étape avant la détection des lignes par la transformée de Hough(cf 3.1).

2.4.1 Principe

Le filtre de Canny est un filtre qui doit remplir les critères suivant, énoncés pas John Canny en 1986 :

- **une bonne détection** : un faible taux d’erreur dans la signalisation des contours
- **une bonne localisation** : une minimisation des distances entre les contours détectés et les contours réels
- **une réponse claire** : une seule réponse par contour et pas de faux positifs

2.4.2 Pré-requis

Afin de faire fonctionner le filtre de Canny, il faut pour cela avoir une image remplissant plusieurs conditions. La première est que cette

image a été modifiée afin de n'avoir plus que des nuances de gris. Cela nous permet d'avoir les valeurs respectives r, g et b identiques dans chaque pixel. Cela facilite largement les calculs, et permet de calculer notamment le gradient. La seconde condition est que l'image doit être nettoyé. Pour cela, nous devons passer des filtres tel que le filtre Gaus-sien, afin d'éliminer le bruit parasite qui pourrait nuire à la détection des contours. En effet, comme nous allons pouvoir le voir, des pixels isolés peuvent créer des contours au milieu des cases qui risquent alors de fausser la détection des lignes.

2.4.3 Gradient d'intensité

Une fois le pré-traitement effectué, nous pouvons alors appliquer un filtre afin de déterminer le gradient en tout pixel de l'image. Nous nous servons ainsi du filtre de Sobel, qui permet de permet, à l'aide d'une matrice de dimension 3X3 : (-1 0 1,-2 0 2,-1 0 1) de calculer le gradient horizontal (sur l'axe x), et une seconde (-1 -2 -1, 0 0 0, 1 2 1) pour calculer le gradient vertical (sur l'axe y). Cela permet d'effectuer un premier filtrage de l'image. Nous calculons alors une norme du gradient en tout point afin de combiner ces deux gradients. Nous en profitons également pour déterminer l'orientation du gradient en tout point. Nous en avons alors fini avec les calculs de gradients.

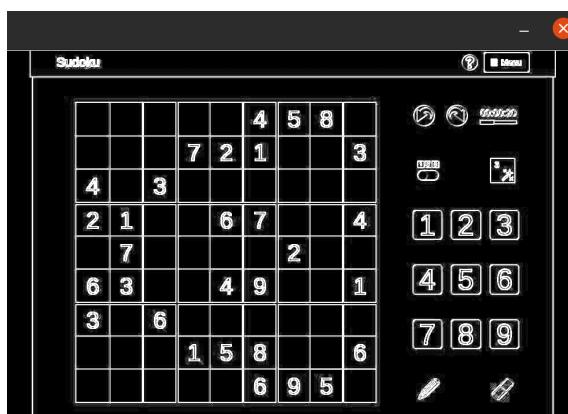


FIGURE 12 – Image après combinaison des gradients

2.4.4 Détection des maximum locaux

Nous avons alors une image qui est passé par plusieurs filtres, ce qui remplit la première condition de Canny : un faible taux d'erreurs dans

la détection des contours. Nous devons alors minimiser les distances entre les contours détectés et les contours réels.

Afin d'affiner notre image, nous allons déterminer quels sont les gradients dont la norme est un maximum local. En effet, les points qui n'appartiennent pas aux maximums sont des imperfections, et rendrons alors l'écriture des bords imprécis. Le local dont il est ici question est déterminé grâce à l'orientation du gradient : en fonction de l'angle de celui-ci, nous allons comparer notre norme avec la norme de deux des 8 pixels voisins de notre pixel (ayant la forme d'un carré, notre pixel initial étant le centre) par exemple, si l'angle est entre 0 et 22.5, nous prenons le point de droite et celui de gauche (point étant sur la même ligne, même de colonne différente). Nous comparons les normes : s'il est inférieur, nous mettons sa norme à 0 (pixel noir), sa norme étant la valeur de ses composantes RGB. Nous effectuons ces opérations pour l'ensemble des pixels.

2.4.5 Nettoyage de l'image

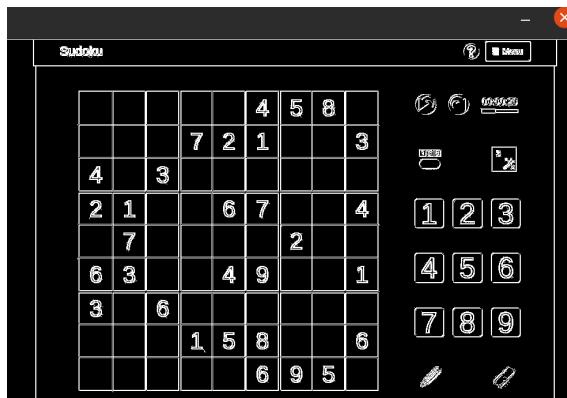


FIGURE 13 – Traitement de l'image terminé

Après ces divers filtres et traitements, nous pouvons observer l'apparition de bruit sur l'image. Afin de nettoyer celui-ci, nous utilisons un algorithme de seuil. Il marche de la façon suivante :

- Si la valeur du r de RGB est supérieur au seuil maximal, alors le point est fort, il devient blanc.
- Si elle est inférieur à un seuil minimal, alors le pixel est faible, il devient noir
- Si elle est ni inférieur au seuil minimal, ni supérieur maximal, alors nous vérifions que l'un de ses voisins est fort. Si c'est le cas,

alors il est également fort, et il devient blanc. Le cas échéant, il est faible, et devient noir. Ainsi, l'image est nettoyé, et le filtre de Canny est alors totalement appliqué.

3 Reconnaissance de la grille

Une des plus grande problématique du traitement d'image est en effet la détection de la grille. Pour répondre à cette problématique nous avons choisis une approche se basant sur la transformé de Hough qui une fois implémenté nous permet de repérer toutes les droites de notre image, cela va constituer la première étape de notre détection.

3.1 La transformé de Hough

Le fonctionnement de la transformé de Hough est le suivant : Dans un premier temps, nous devons calculer l'équation polaire de toutes les droites passant par tous les points blancs (ceux qui nous interesse donc après l'application de Canny) de l'image, de la forme $r\theta = x\cos(\theta) + y\sin(\theta)$, avec θ un angle. Nous allons alors calculer $r\theta$ pour l'ensemble des angles de 0 à 180 degrés. Nous ajoutons un 1 dans un accumulateur prenant la forme d'un malloc, un tableau utilisant les qualités des pointeurs, de lignes d'index θ et de colonnes d'index $r\theta$ à chaque fois que nous trouvons une combinaison entre un angle et un point. Cela nous permet d'avoir la densité des droites, c'est à dire le nombre de point par lesquels passent chaque droite.

Après cela, nous parcourons notre malloc afin de ne récupérer que les droites ayant une certaine densité, c'est-à-dire au moins 0.4 fois la plus grande densité observé, avant de les stocker dans une liste de tuple de composants $r\theta$, θ et densité, ces deux structures ayant été créées pour notre OCR. Ce sont les lignes présentes sur l'images qui sont d'une taille minimale. Une fois cela fait, nous déterminons deux points appartenant à chaque droite sélectionnée qui appartiennent à l'image afin de tracer celle-ci sous forme de pixel vert sur notre image. Pour ce faire, nous déterminons par quel bord de l'image passe notre droite, et cela nous donne un x ou un y par deux fois, et nous calculons à l'aide de l'équation polaire le paramètre manquant.

Nous vérifions à ce moment si une rotation automatique est nécessaire (cf 2.3.1), et nous l'appliquons si c'est le cas. Nous ré appliquons alors

la transformée de Hough à cette image, et nous cherchons les lignes orientées avec un angle de 0 ou de 90 degré, puisque notre sudoku est alors parfaitement droit.

Une fois les préparations terminé nous pouvons implémenter la suite de notre algorithme.

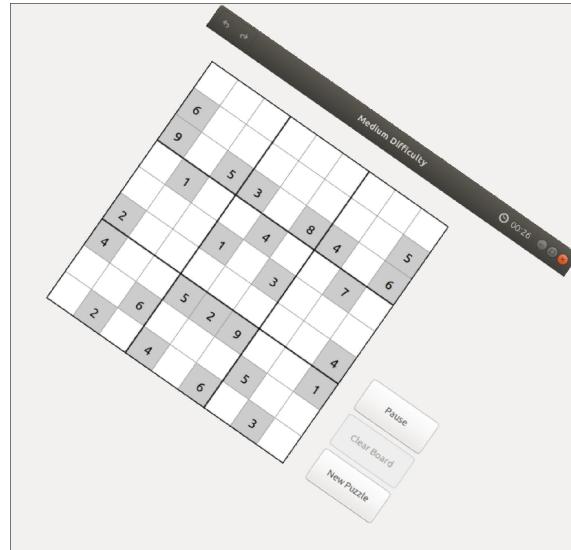


FIGURE 14 – Image sans traitement

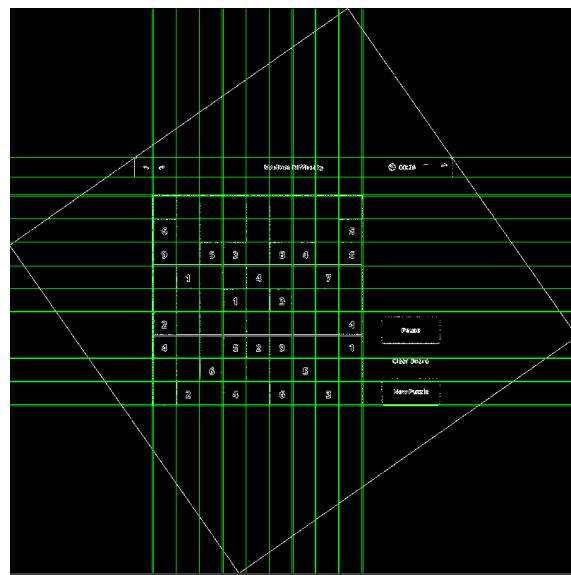


FIGURE 15 – Image après rotation automatique, détection des lignes et zoom de l'image(donc disparition sur l'affichage uniquement d'une partie des lignes vertes)

3.2 Détection de la grille

Afin de détecter la grille, nous allons nous servir des droites précédemment tracées. Pour cela, nous allons aller de droite en droite, et les stocker dans une liste de tuple ayant pour composant le x (ou le y) de la droite précédente, le x (ou le y) de la droite suivante, et la distance entre les deux (x et y dépend si nous cherchons les droites horizontales ou verticales). Nous allons parcourir notre liste et supprimer toutes les valeurs qui ont une distance inférieure à un certain seuil (un centième du plus grand côté de l'image), ce sont des entre deux-lignes trop petit pour être un sudoku, et du bruit créé par Canny, étant donné que ce sont des contours, si une ligne était trop large, cela a pu créer deux lignes au lieu d'une avec canny et donc Hough. Une fois cela effectué, nous créons une copie de cette liste. Nous allons ensuite trier cette copie afin d'avoir la valeur médiane de la distance.

Nous allons ensuite parcourir la liste initiale. Nous allons vérifier pour une case si sa distance est égale à sa médiane avec une marge d'erreur d'un centième du côté le plus long de l'image. Une fois cela effectué, nous vérifions que c'est le cas pour huit de ses voisins, étant donné qu'un sudoku possède neuf cases de larges et de haut. Sinon, nous avançons simplement jusqu'à la prochaine case à tester. Si elle possède bel et bien huit voisins, alors nous récupérons ceux-ci et nous les stockons. Nous réitérons l'entièreté de l'opération pour les droites verticales, étant donné que nous commençons avec les droites horizontales. Nous avons ainsi les différents carrés composants le sudoku, nous les traçons donc pour des raisons esthétique.

Il ne reste plus qu'à découper des carré de côté égaux à la plus petite distance de chaque case, avant de les dimensionner de façon à avoir des images de 28 par 28 pour notre réseau de neurone, et de sauvegarder les images sous forme de fichier.

3.3 Formater nos cases

Une fois nos cases bien détecté nous devons les formater pour les passer à notre réseau de neurones. En effet celui-ci n'acceptant que des images en 28x28, de manière générales les cases que nous découperons auront une dimension bien plus élevé bien plus élevé.

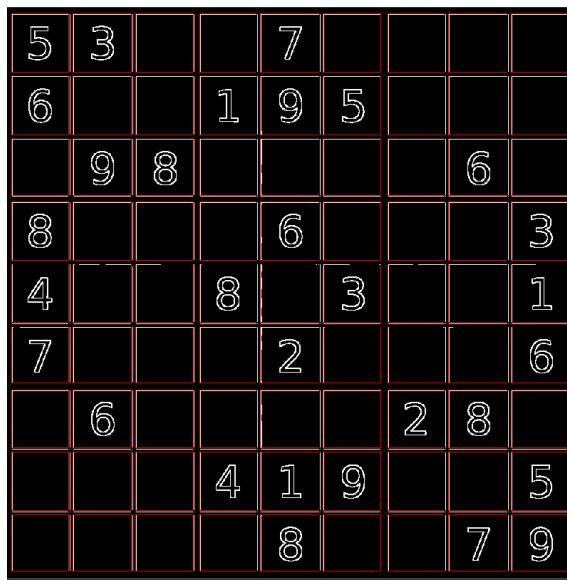


FIGURE 16 – détection de la grid du sudoku

Pour réduire la dimension de nos cases en gardant la pertinence de ce qu'elles contiennent nous allons d'abord chercher le multiple de 28 supérieur le plus proche de la dimension actuelle de la case, ensuite nous allons dimensionner naïvement notre case à ce multiple de 28.

Une fois cela fait nous allons utiliser le coefficient (nommons le "A" ici) qui nous permet de passer de 28 à notre multiple (ex : $112 = 4 \times 28$), et nous allons l'utiliser pour parcourir nos cases par carré de dimension AxA, pour chacun de ces carrés nous allons ajouter le nombre de pixels blanc et en faire une moyenne en la divisant par le nombre de pixels total, le résultat obtenu nous donnera alors la valeur d'un pixel de notre nouvelle image en 28x28.

Cependant nos cases peuvent aussi avoir des défauts due soit à la qualité de l'image d'origine soit au traitement par lequel elle a pu passer pour la détection du Sudoku, pour pallier à ce problème nous devons nettoyer nos cases de pixels blancs parasites ce que fait en partie notre algorithme de redimensionnement d'image en associant un certains poids à chaque pixel grâce à la moyenne nous permettant de filtrer les pixels blanc isolé nous appliquons aussi certaines autres règles plus arbitraires pour nous permettre de nettoyer l'image.

4 Réseau de neurones

4.1 Variable

Une fois notre réseau de neurones reconnaissant un Xor fait, il étais temps de l'adapter pour la reconnaissance de caractères. Tout d'abord il est nécessaire de trouver une base de donnée sur laquelle notre réseau pourra s'entraîner puis se tester pour vérifier son efficacité. Pour cela nous avons choisi la base de donné MNIST composé de 60.000 images de chiffre en 0 et 9 pour l'entraînement ainsi que 10.000 images pour tester notre réseau de neurone. Ces images sont en format de 28 pixel par 28 pixel, ce qui va obliger notre réseau de neurone à avoir 784 neurones d'entrées. Ces chiffres sont des chiffres manuscrits, ils sont donc pratique car très diversifiés et permettent à notre réseau de neurone de reconnaître une grande diversité de chiffre. Néanmoins, nous avons rajouté par nous mémé des images de chiffres manuscrits également en 28x28 pour aider notre réseau de neurone sur des chiffres numériques très similaires. Nous avons donc une base de donnée de 120.000 éléments d'entraînements et 20.000 de tests.

Très peu de modification ont été effectuées entre le réseau de neurones du Xor et celui-ci. Le learning rate n'a pas été modifié et le nombre de couche de neurone caché non plus, il n'y en a qu'une seule. Cependant le nombre de neurone dans chaque couche a bien sûr été modifié. Il y a donc maintenant 784 neurones d'entrées (1 pour chaque pixel de l'image), 98 neurones cachés ainsi que 8 neurones de sortie. Pourquoi 8 et pas 9 ? Pour limiter les erreurs nous avons choisi de ne pas entraîner notre réseau de neurone sur les 0 puisqu'un sudoku ne peut être rempli que de 1 à 9. Les 98 neurones cachés ont été choisi arbitrairement à la suite de nombreux tests. Cela nous semble optimal de plus 98 étant un diviseurs de 784, nous pensons que cela aide le réseau de neurone à réfléchir de manière plus symétrique. Pour éviter le sur-apprentissage, nous effectuons l'apprentissage 2,5 sur les 120.000 images. Si nous augmentons trop, le réseau de neurone va apprendre « par cœur » les résultats et risque donc de ce tromper trop souvent, il faut que celui-ci garde une certaine flexibilité. Si nous diminuons trop, notre réseau n'apprend plus assez et risque de se tromper trop souvent.

RÉSUMÉ

Rapide résumé :

- 784 neurones d'entrés pour des images de 28 par 28 pixels.
- 98 neurones cachés.
- 8 neurones de sorties pour éviter d'apprendre les "0".
- 1 seule couche de neurones cachées
- 120.000 images d'entraînement
- 20.000 images de test
- 2,5 générations pour l'apprentissage
- 0.1 taux d'apprentissage
- 90-95% de réussite lors des tests.

4.2 Fonctionnement du réseau

4.2.1 Résolution

Que ce soit pour les entraînements comme pour les tests, la résolution est une étape indispensable. En réalité cette étape est plus simple qu'il n'y paraît. Nous allons nous balader de couche en couche en déterminant les valeurs des neurones à partir des précédents.

Voici le fonctionnement en détail :

- Première boucle d'opérations durant laquelle nous allons déterminer les valeurs des neurones de la couche "cachée" à partir de ceux d'entrée. Voici le raisonnement effectué pour déterminer la valeur d'un neurone : nous multiplions chaque valeur des neurones d'entrée par les poids associés entre cette entrée et le neurone. Nous les additionnons tous, puis on ajoute à cela le biais du neurone. Néanmoins nous souhaitons trouver une valeur entre 0 et 1. Pour cela nous allons utiliser la fonction sigmoïde définie ci-dessous.

- Seconde boucle d'opérations. Pour celle-ci nous allons utiliser une fonction *softmax*. Comme pour la précédente, cette fonction permet de donner un résultat compris entre 1 et 0 pour chaque neurone de sortie. Il nous suffit ensuite d'effectuer une simple opération permettant de récupérer le neurone ayant la plus grande valeur de sortie pour connaître le résultat de notre réseau de neurone. Cette valeur de sortie est en réalité un taux de succès transformé en probabilité que

le neurone soit la bonne réponse par la fonction softmax.

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}.$$

FIGURE 17 – Fonction Sigmoïde

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

FIGURE 18 – Fonction Softmax

4.2.2 Entraînement

Pour s'entraîner, le réseau de neurone possède une série de 120.000 images. Étant donné la diversité d'images que nous n'avons pas besoins d'utiliser une fonction pour les mélanger. De plus les images donné à l'intérieur ne se suivent pas, ce qui améliore l'efficacité du réseau de neurone. A chaque génération, l'entraînement va simplement résoudre le test qui lui est donné. Mais c'est ensuite que les choses changent. En effet, nous allons faire ce que l'on appelle la "**Back-Propagation**". C'est à dire que nous allons parcourir notre réseau de neurone et modifier très légèrement toutes les valeurs. La grande question est comment modifier ses valeurs pour se rapprocher du bon résultat ? Pour cela nous allons calculer le gradient de chaque valeur de sortie. Pour calculer ce gradient, nous allons avoir besoins de calculer la différence entre le résultat sorti et celui attendu puis la dérivé de la fonction sigmoïde avec en paramètre le résultat sorti. Nous allons ensuite multiplier les deux. Pour finir, nous allons multiplier le résultat

par ce que l'on appelle le "*Learning Rate*", qui permet de d'amplifier et surtout de s'assurer d'une certaine modification à chaque essai. Une fois cette expression mise en place, il suffit de modifier chaque biais et chaque poids en ajoutant à leur valeur actuelle le gradient. Bien évidemment la valeur de ce gradient est différente pour chaque couche de neurone et même chaque neurone puisqu'il s'appuie sur la valeur de ceux-ci.

Cette formule représente en quelque sorte la modification nécessaire pour que l'erreur soit minimal en fonction de chaque biais et poids. Pour représenter cela, il est possible de s'imaginer notre fonction variant pour chaque poids et biais comme une courbe en 3 dimensions. Notre point de départ étant suggéré comme une balle. Notre objectif est de faire rouler cette balle vers le point le plus bas de cette courbe. Ce point étant le minimum, cela signifie qu'une fois atteint, notre réseau s'approche de l'erreur minimal. Cependant, nous n'appliquons pas les lois de la gravité pour faire descendre notre balle, mais plutôt la formule exprimé ci-dessus.

```
=====> EPOCH N°0 <=====

RESULT : 5 => 0.468752
|   Expected : 9
1 Res => 0.025207
2 Res => 0.121663
3 Res => 0.033527
4 Res => 0.000005
5 Res => 0.468752
6 Res => 0.000000
7 Res => 0.345528
8 Res => 0.003139
9 Res => 0.002179
Error dif = -0.025207 => 0.000000 - 0.025207
Error dif = -0.121663 => 0.000000 - 0.121663
Error dif = -0.033527 => 0.000000 - 0.033527
Error dif = -0.000005 => 0.000000 - 0.000005
Error dif = -0.468752 => 0.000000 - 0.468752
Error dif = -0.000000 => 0.000000 - 0.000000
Error dif = -0.345528 => 0.000000 - 0.345528
Error dif = -0.003139 => 0.000000 - 0.003139
=>BackProp Input : (j)9 => (expected)9
Error dif = 0.997821 => 1.000000 - 0.002179
```

FIGURE 19 – Entraînement Époque N°0

```

=====> EPOCH N°6000 <=====
RESULT : 1 => 0.991758
| Expected : 1
Previous : 9
Next : 3
1 Res => 0.991758
2 Res => 0.000706
3 Res => 0.000978
4 Res => 0.000005
5 Res => 0.002529
6 Res => 0.002899
7 Res => 0.000566
8 Res => 0.000538
9 Res => 0.000021
=>BackProp Input : (j)1 => (expected)1
Error dif = 0.008242 => 1.000000 - 0.991758
Error dif = -0.000706 => 0.000000 - 0.000706
Error dif = -0.000978 => 0.000000 - 0.000978
Error dif = -0.000005 => 0.000000 - 0.000005
Error dif = -0.002529 => 0.000000 - 0.002529
Error dif = -0.002899 => 0.000000 - 0.002899
Error dif = -0.000566 => 0.000000 - 0.000566
Error dif = -0.000538 => 0.000000 - 0.000538
Error dif = -0.000021 => 0.000000 - 0.000021

```

FIGURE 20 – Entraînement Époque N°6.000

4.3 Sauvegarde des valeurs et test du réseau

Une fois que nous avons obtenu nos valeurs, nous ne souhaitons pas les perdre. Nous allons donc les sauvegarder dans un fichier dont le nom sera donné en paramètre. Le format de sauvegarde est simple, le voici :

- Poids des neurones cachés.
- Poids des neurones de sorties
- Biais des neurones cachés.
- Biais des neurones de sorties

```

tebion@tebion-Creator-15-A10SF:~/OCR/0cr_sudoku/Scripts/OCR$ cat test.test
1.160183
6.054511
3.334346
0.692432
1.002711
5.951059
3.476393
0.566947
-2.952015
8.571478
-7.103246
-2.242919
-1.461482
-2.513302
-5.233067
-0.169033
-1.856379

```

FIGURE 21 – Fichier de sauvegarde du réseau de neurone

Une fois que nous avons sauvegardé ces valeurs, nous pouvons tester notre réseau de neurone. Pour cela il suffit d'appeler le programme en cliquant sur tester au niveau de l'application. Le résultat s'affichera en dessous ainsi que dans la console de manière plus précise.

Voici un exemple de test effectué sur un réseau de neurone entraîné au préalable :

```
cebion@cebion-Creator-15-A10SF:~/OCR/FC_humble/Scripts/OCR$ ./xor defaultnet
Reading .../

=====> EPOCH N°0 <=====
RESULT : 9 => 0.993812
| Expected : 9
1 Res => 0.000043
2 Res => 0.000002
3 Res => 0.000000
4 Res => 0.005888
5 Res => 0.000088
6 Res => 0.000023
7 Res => 0.000107
8 Res => 0.000039
9 Res => 0.993812

RESULT : 4 => 0.998953
| Expected : 4
Previous : 9
Next : 1
1 Res => 0.000000
2 Res => 0.000036
3 Res => 0.000000
4 Res => 0.998953
5 Res => 0.000004
6 Res => 0.000007
7 Res => 0.000000
8 Res => 0.000000
9 Res => 0.000000

=====> EPOCH N°1000 <=====
RESULT : 1 => 1.000000
| Expected : 1
Previous : 1
Next : 0
1 Res => 1.000000
2 Res => 0.000000
3 Res => 0.000000
4 Res => 0.000000
5 Res => 0.000000
6 Res => 0.000000
7 Res => 0.000000
8 Res => 0.000000
9 Res => 0.000000
```

FIGURE 22 – Début d'un test

5 Résolution

5.1 Sauvegarde et traitement des fichiers

Comme demandé, le programme intitulé "*solver*" prend en paramètre le nom du fichier contenant la grille de départ puis le résout. Le format demandé pour la grille de départ est celui ci :

Les cases vides sont remplacées par des points tandis que les cases pleines sont représentées par leur chiffre respectif. Les colonnes sont représenté par un espace entre deux chiffres ou points et les lignes sont représenté par un saut de ligne. Ainsi il est facile de repérer dans quel carré nous nous trouvons. Le format de la réponse est exactement du même type. Néanmoins, comme la réponse se doit d'être complète, il n'y a plus de point représentant le 0. En voici un exemple :

De plus il est possible d'ouvrir ses réponses sous n'importe quel éditeur ou bien de les afficher dans le terminal via la commande "*cat*" puis le nom du fichier. Pour en finir avec la sauvegarde des résultats, le

```

=====> EPOCH N°19000 <=====
RESULT : 8 => 0.720004
|   Expected : 8
Previous : 1
Next : 6
1 Res => 0.155823
2 Res => 0.006716
3 Res => 0.011446
4 Res => 0.007846
5 Res => 0.047737
6 Res => 0.022542
7 Res => 0.000724
8 Res => 0.720004
9 Res => 0.033162
RESULT : 6 => 0.994705
|   Expected : 6
Previous : 8
Next : 3
1 Res => 0.000034
2 Res => 0.002387
3 Res => 0.000000
4 Res => 0.002475
5 Res => 0.000019
6 Res => 0.994705
7 Res => 0.000002
8 Res => 0.000002
9 Res => 0.000376
Number Success : 17869.000000

=====> RESULT <=====
93.948475/100 SUCESS RATE
=====> RESULT <=====
Done

```

FIGURE 23 – Calcul d'un taux de réussite

fichier comprenant la réponse au sudoku se nommera toujours comme le nom du fichier source en rajoutant *.result* en suffixe.

5.2 Algorithme de résolution : Backtracking

Une fois la grille créée à partir d'un fichier, l'algorithme permettant de résoudre la grille se met en marche. La première étape est de vérifier la validité de la grille. En effet, si la grille donnée est fausse notre programme sera capable de renvoyer un message d'erreur. Une grille est fausse si elle contient par exemple deux fois le même chiffre dans une colonne, dans une ligne ou dans un carré. Arrive ensuite l'algorithme de résolution. Celui-ci part depuis la case en haut à gauche, puis parcourt la totalité de la grille en appliquant ce principe :

- Si la case est déjà prise je passe à la suivante
- Si la case est vide j'essaye de placer un nombre en commençant par **1**. Je vérifie que celui-ci ne se trouve pas déjà dans la colonne, dans la ligne ou bien dans le carré. Sinon je passe au chiffre suivant.
- Je passe ensuite à la case suivante et réitère le principe. Si il est impossible de placer un chiffre entre **1** et **9** alors je reviens à la

```
tebion@tebion-Creator-15-A10SF:~/OCR/0cr_sudoku/Scripts/Solver$ tree -L 1
.
├── grid_00
└── solver
    └── solver.c

0 directories, 3 files
tebion@tebion-Creator-15-A10SF:~/OCR/0cr_sudoku/Scripts/Solver$ cat grid_00
8.. ...
..3 6..
.7. .9. 2..
.
.5. ..7 ...
.... .45 7..
... 1.. .3.

..1 ... .68
..8 5.. .1.
.9. ... 4..


```

FIGURE 24 – grid-00

```
tebion@tebion-Creator-15-A10SF:~/OCR/arthur.oldrati/Scripts/Solver$ tree
.
├── grid_00
├── grid_00.result
└── solver.c

0 directories, 4 files
tebion@tebion-Creator-15-A10SF:~/OCR/arthur.oldrati/Scripts/Solver$ cat grid_00.result
812 753 649
943 682 175
675 491 283

154 237 896
369 845 721
287 169 534

521 974 368
438 526 917
796 318 452


```

FIGURE 25 – grid-00.result

case précédente place le chiffre suivant.

- Et ainsi de suite jusqu'à arriver à la fin du sudoku !

Une fois arrivé ici, nous rentrons le résultat dans le fichier de sauvegarde en respectant le format demandé. Le programme "*Solver*" s'arrête ici.

6 Image de retour

Afin de présenter les résultats à l'utilisateur sous une autre forme que celle d'un fichier test, nous initialisons une nouvelle image composée d'une grille vide. A l'aide du fichier renvoyé par la reconnaissance de caractère de notre OCR, nous ajoutons des chiffres en noir sur l'image.

Une fois cela effectué, nous parsons le fichier renvoyé par le solver, qui est donc le sudoku résolu. Cela nous permet d'ajouter dans notre grille

des chiffres rouges s'il n'existe pas encore de chiffre à cet emplacement dans le fichier renvoyé par l'OCR.

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

FIGURE 26 – exemple de sudoku renvoyé

7 Exécutable

Pour créer une interface utilisateur, nous avons décidé d'utiliser le programme *gtk-3.0* avec l'application *glade* permettant la construction de ceux-ci plus facile. Cette interface sera notre portail vers tout les programmes et toutes les étapes pour la résolution du sudoku. Cette interface se divise en 3 parties différentes que nous allons regarder en détail :

- La première partie située sur la gauche est le menu et permet à l'utilisateur de contrôler l'application de A à Z. Tout d'abord dans la partie supérieure se trouve une entrée de texte ainsi qu'un bouton permettant de valider l'entrée. Celle-ci doit être le nom de l'image situé dans le fichier ressource, il n'est pas nécessaire d'indiquer le chemin

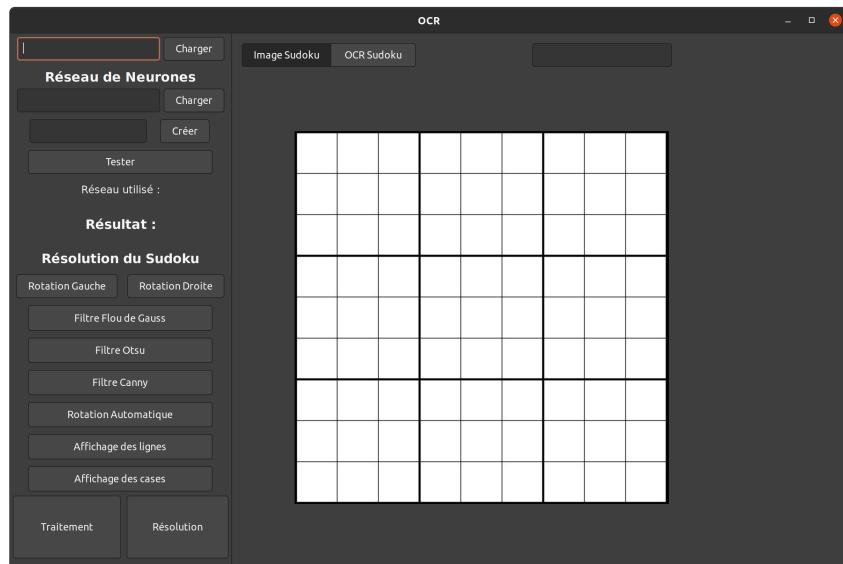


FIGURE 27 – Application à vide

celui-ci étant automatique. Une fois l'image choisie, celle-ci s'affiche au milieu de l'application. Cependant il est toujours possible de changer d'image si l'utilisateur s'est trompé. Juste en dessous se trouve la partie concernant le réseau de neurone. Tout d'abord une entrée de texte pour permettre à l'utilisateur de rentrer le nom du réseau de neurone qu'il souhaite utiliser. Il appuie sur le bouton de sauvegarde pour sauvegarder le réseau de neurone choisis. Encore une fois pour limiter les erreurs, ces fichiers sont situé par défaut dans le fichier *obj* et se terminent en **.netOCR**. En dessous se trouve un espace de création de réseau de neurone. Il suffit de rentrer dans cette zone de texte le nom que l'on souhaite lui donner pour que le programme d'apprentissage se lance automatiquement. Le détail de cet apprentissage peut-être vu et lu dans le terminal en cours. Cette opération peut prendre plusieurs secondes, une fois fini un fichier sera créé se plaçant automatiquement dans le dossier *obj* et portant le nom que l'utilisateur lui a donné ainsi que le suffixe *.netOCR*. Juste après avoir entraîner un réseau de neurones ou après en avoir sélectionner un, il est possible de tester celui-ci en cliquant sur le dernier des trois boutons, le résultat en pourcentage de réussite s'affichera ensuite juste en dessous.

Et en dernier ce trouve les 14 boutons qui permettent à l'utilisateur d'appliquer les différents filtres sur l'image. Pour commencer la première ligne est composé d'une entrée de valeur qui peut être compris entre 180 et -180. Une fois ceci fait, l'utilisateur peut appuyer sur le

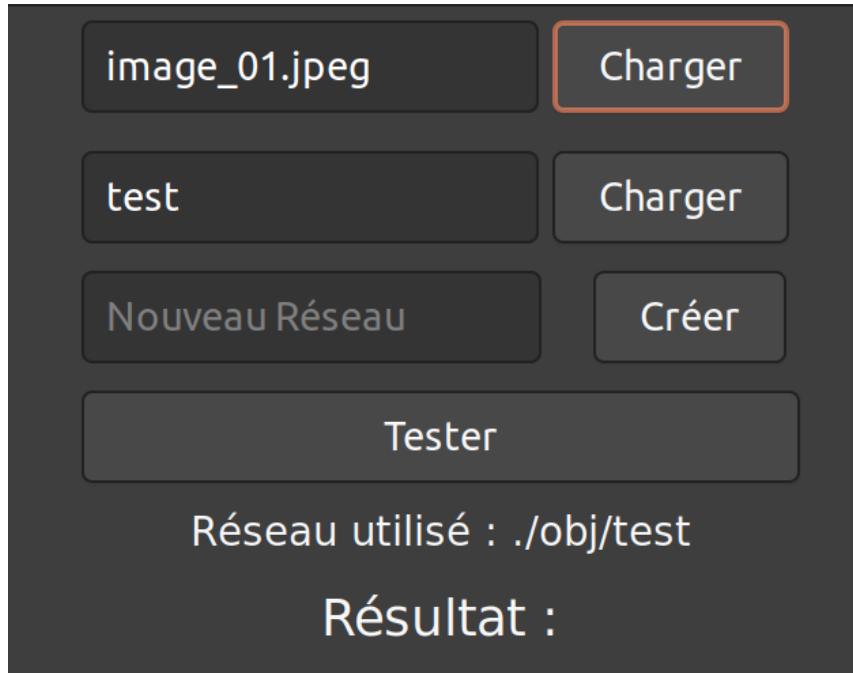


FIGURE 28 – Application à vide

bouton rotation pour l'appliquer. Il est important de savoir que si la valeur est positive alors la rotation s'effectuera sur la gauche tandis que si la valeur est négative, la rotation s'effectuera alors sur la droite. Les suivants permettent d'appliquer les flou de Gauss. Lors de l'appui de l'un d'eux cela appliquera un flou de Gauss plus ou moins fort. Il est important de savoir que les flou s'accumulent sur l'image et la sauvegarde de plus il n'est pas possible de revenir en arrière. Ensuite l'utilisateur peut appliquer le filtre médian, même si les modifications sont minimes elles seront importantes pour la suite de la résolution. Ensuite même principe que pour la rotation, l'utilisateur peut s'il souhaite appliquer une valeur pour le filtre otsu. Cependant dans ce cas l'image n'est pas sauvegarder il est donc possible de revenir en arrière afin de trouver la valeur la plus optimale pour le traitement de l'image, car en effet la valeur elle est sauvegarder pour la suite du traitement. On applique ensuite le filtre de Canny qui va transformer l'image en blanc et noir, ici l'image n'est pas non plus sauvegarder et n'est utile que pour l'utilisateur pour vérifier la validité des valeurs précédentes. Une fois ces traitements fait il est possible d'effectuer la rotation automatique en appuyant sur le bouton suivant. Puis en dernier ce trouve l'affichage des lignes et l'affichage des cases, ici encore l'image n'est pas retenue pour la suite et permet simplement à l'utili-

sateur de voir l'intégralité du traitement. Cependant il est encore une fois important de savoir que certaines lignes ou cases peuvent ne pas paraître visible mais sont pour autant détecter par la suite. Cela est du au zoom nécessaire pour afficher l'intégrité de l'image dans l'application. Cela permet à l'utilisateur de vérifier si ceux-ci sont corrects et si le traitement a fonctionné, en effet le message d'erreur suivant apparaîtra si la grille n'est pas détecté.



FIGURE 29 – Erreur N°1

Viennent ensuite les deux derniers boutons essentiels, le premier permettant d'appliquer le réseau de neurone sur l'image traité intégralement, et le second permettant de résoudre la grille donné en résultat par le précédent. Lorsque l'utilisateur appui sur le premier cela va générer un fichier ayant pour nom l'entrée rentrée par l'utilisateur dans le coin droit de l'application. Ce fichier est le fichier contenant la grille reconnue par le réseau de neurone. Lorsque l'utilisateur appuiera sur le second bouton notre programme essayera de résoudre le sudoku. Il y a alors plusieurs possibilité. Si plusieurs chiffres identiques se trouvent dans la même colonne, ligne ou même grande case alors le message

d'erreur suivant apparaitra :



FIGURE 30 – Erreur N°2

Si jamais la grille est impossible, c'est à dire qu'il n'existe aucune solution possible, alors notre programme ne générera pas d'erreur mais le sudoku renvoyé sera rempli de 0 dans les cases insolvable et l'image du sudoku généré ne sera pas complète. Mais si le sudoku reconnut est correct et alors solvable, notre programme générera une image recomposé d'un sudoku avec en noir les chiffres présents dans la grille et en rouge les chiffres rajouté par le programme lors de la résolution. En voici un exemple :

- Si nous regardons l'autre partie de l'écran, nous pouvons voir qu'il y a deux onglets : **Image Sudoku** et **OCR Sudoku**. Le premier onglet est l'aperçu photo du Sudoku, celle-ci évoluant au fur et à mesure que l'utilisateur appuie sur les boutons pour appliquer les différents filtres et observer les évolutions de la reconnaissance de la grille. L'aperçu sera toujours au maximum de 800 par 800 pixels. Pour rappel le zoom appliqué ici peut être trompeur, en effet lors de la détection des lignes ou détection des cases, certains trait peuvent être



FIGURE 31 – Application à vide

dissimulé. Le second onglet intitulé **OCR Sudoku** est un éditeur de texte qui une fois la reconnaissance effectué va reproduire en suivant le format demandé le sudoku trouvé. Cependant le principal intérêt de cet éditeur est qu'il permet de corriger les éventuels erreurs commis par le réseau de neurone lors de la reconnaissance des caractères dans la grille. Ainsi l'utilisateur peut remplacer les éventuels erreurs, puis cliquer sur Sauvegarder pour écrire de nouveau le fichier avant d'appuyer sur le bouton **Résoudre** pour finaliser la procédure et générer le fichier du résultat. A ce moment un fichier contenant la grille en **.result** sera créé et une image reconstitué sera également sauvegarder avec les numéros rajoutés en rouge. Concernant son fonctionnement il est important de savoir que l'utilisateur doit avoir correctement corrigé le fichier. En effet si celui-ci ne respecte pas le format demandé dans le cahier des charges alors cela générera une erreur dans lors de la résolution, il s'agit de celle vu précédemment. Voici quelques photos pour illustrer nos propos :

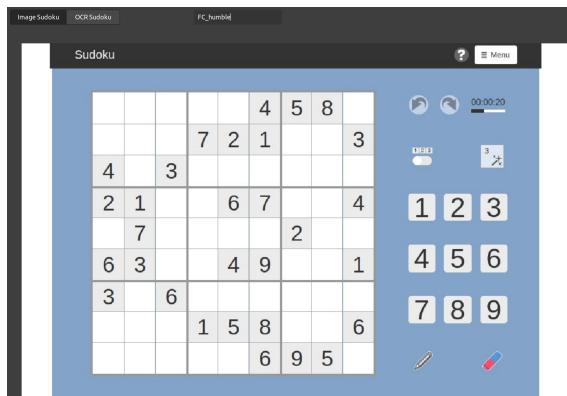


FIGURE 32 – Aperçu d'une image

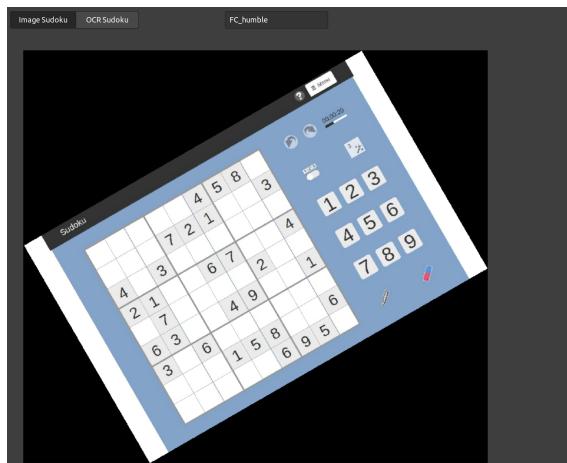


FIGURE 33 – Rotation d'une image de 30°

Ici la rotation n'est pas nécessaire il s'agit simplement d'un cas d'exemple

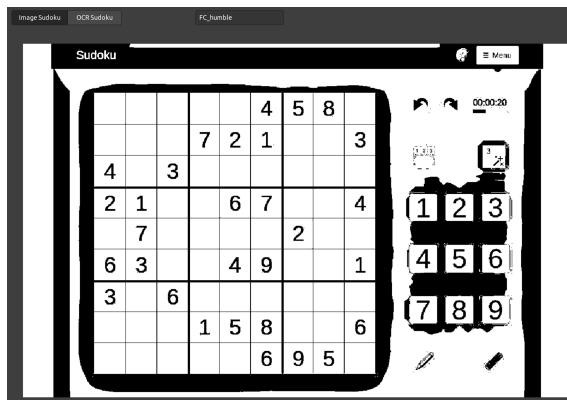


FIGURE 34 – Application d’Otsu

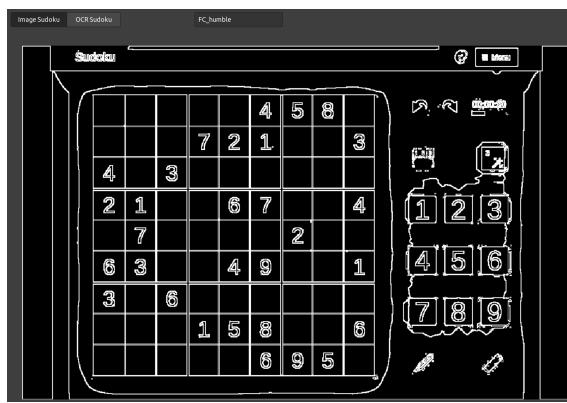


FIGURE 35 – Application de Canny

Il est aussi possible d’appliquer d’autres filtres en fonction de l’image donné. Ici celle que nous avons choisi n’a pas besoins de ces filtres.

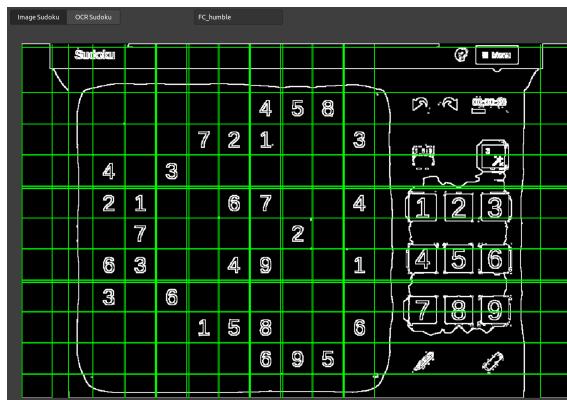


FIGURE 36 – Affichage des lignes

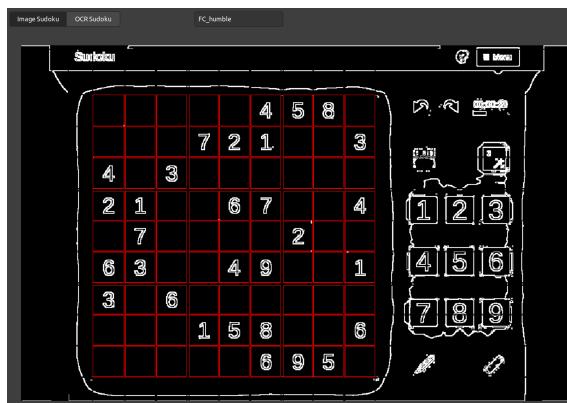


FIGURE 37 – Affichage des Cases

L'affichage des lignes et des cases nous permettent de voir que le pré-traitement de l'image a bel et bien été effectué. Si cela n'était pas le cas nous aurions soit vu un problème soit nous aurions eu une erreur.

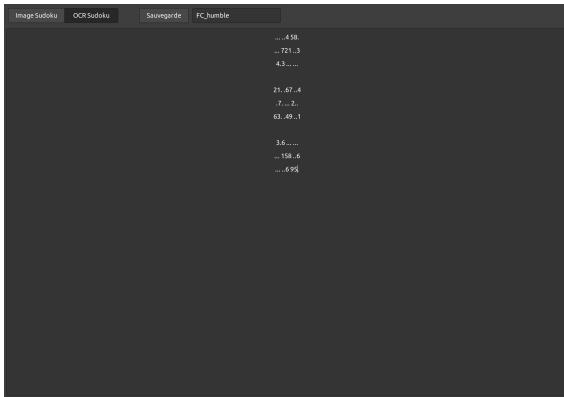


FIGURE 38 – Reconnaissance du réseau de neurone

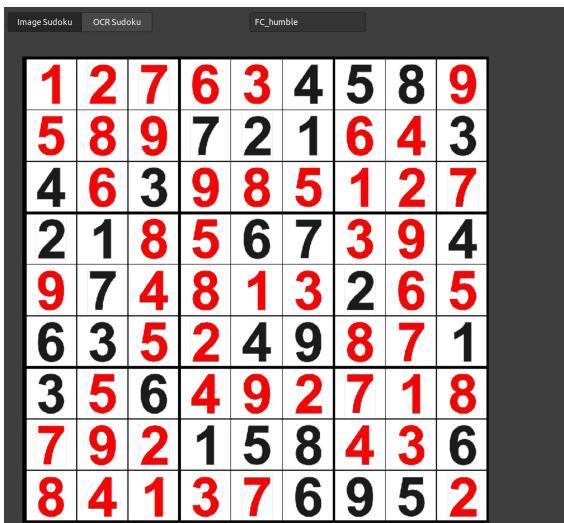


FIGURE 39 – Résultat final

Après quelques petites modifications sur la réponse du réseau de neurone, nous obtenons la réponse à notre sudoku !

8 Suivi du projet

Au global, nous pensons avoir réussi à remplir les conditions initiales. En effet, notre programme est bel et bien composé d'un executable codé via glade et gtk. Notre programme est également capable reconnaître un sudoku si les valeurs rentrée par l'utilisateur pour le flou de Gauss et pour le filtre d'Otsu sont les bonnes. De plus nous estimons que notre programme est suffisamment optimisé. En effet, il ne dépasse les 5 secondes lorsqu'il s'agit de le résoudre. De l'autre côté notre réseau de neurone est capable de lire des caractères manuscrit avec environ plus de 90% de réussite. Cela est du au fait que nous

avons beaucoup cherché les valeurs optimales pour son bon fonctionnement. Lors de la résolution du sudoku, celui-ci peut rencontré quelque difficulté à correctement lire les caractères typographié du sudoku. En effet cela est dû à une certaine différence entre les deux. Nous estimons que la reconnaissance de caractère typographié peut s'évaluer à 70% de réussite environ. Après plusieurs relecture nous pensons avoir respecté le cahier des charges dans son intégrité.

9 Conclusion

L'OCR est un projet qui nous tient vraiment à cœur, nous pensons avoir bien réussi la première partie de ce projet et nous avons hâte de continuer. La cohésion de groupe est bonne et nous avons pris plaisir à travailler et coopérer ensemble. Nous sommes entièrement satisfait du résultat de notre OCR et sommes très fier d'avoir obtenu un résultat qui dépasse nos premières attentes !