

Rapport de Projet OCR n°2

Sudoku Solver

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

Membres du groupe 9 :

Charlotte Buat
Camille Nguyen (*Cheffe de projet*)
Florine Kieraga
Sami Carret

Sommaire

1	Introduction	3
2	Répartition des tâches accomplies et état d'avancement du projet	4
2.1	Répartition des tâches accomplies	4
2.2	Etat d'avancement du projet	5
3	Aspects techniques	6
3.1	Chargement de l'image	6
3.2	Suppression des couleurs	6
3.2.1	Niveau de gris	6
3.2.2	Noir et blanc	7
3.3	Prétraitement	10
3.3.1	Flou Gaussien et dilatation	10
3.3.2	Inversion noir et blanc	10
3.3.3	Filtre de Sobel	11
3.3.4	Rotation automatique	12
3.4	Détection de la grille et des cases	13
3.4.1	Détection et isolation de la plus grande composante connexe .	13
3.4.2	Détection des lignes à l'aide de la transformée de Hough . . .	15
3.4.3	Isolation de la grille	16
3.4.4	Isolation des cases	17
3.5	Réseau de neurones	18
3.5.1	Forward propagation	18
3.5.2	Back propagation	19
3.5.3	Détection de caractères	19
3.5.4	Exportation des images	21
3.6	Résolution du sudoku	23
3.6.1	Backtracking	23
3.6.2	Format du fichier	24
3.7	Interface graphique	25
3.7.1	Drawing Area (Grille)	25
4	Ressenti des membres du groupe	27
4.1	Charlotte Buat	27
4.2	Camille Nguyen	27

4.3	Florine Kieraga	27
4.4	Sami Carret	28
5	Conclusion	29
5.1	Bibliographie	30

1 Introduction

Dans ce rapport, nous détaillerons les différentes tâches que nous avons accomplies jusqu'à la soutenance finale de projet OCR (*Optical Character Recognition*). Pour rappel, le projet du semestre 3 est un projet qui a pour objectif de réaliser un logiciel de type OCR capable de résoudre automatiquement une grille de sudoku. Celui-ci prend une image représentant une grille de sudoku en entrée et affiche en sortie la grille résolue. Pour ce faire, il est nécessaire de prendre en compte les différentes contraintes imposées par la photo ayant été sélectionnée (par exemple, une photo floue, penchée, en couleur... rendra le processus de prétraitement d'image plus long et fastidieux). Par ailleurs, ce logiciel est écrit en langage C et utilise les bibliothèques présentes sur la machine de l'école. Il doit compiler sur NixOS.

Dans un premier temps, nous présenterons la répartition des tâches réalisées par l'ensemble des membres afin de concevoir le projet OCR. Dans un second temps, nous développerons l'aspect technique du projet avec les différents traitements appliqués à l'image, la détection de celle-ci ainsi que le réseau de neurones et l'algorithme de résolution du sudoku.

2 Répartition des tâches accomplies et état d'avancement du projet

2.1 Répartition des tâches accomplies

Ce tableau présente les tâches accomplies par les membres du groupe entre la formation de notre groupe et la soutenance finale du projet. Les X représentent la ou les personne(s) ayant accompli cette tâche. Le s representent la ou les personnes(s) ayant aidé à faire cette tâche.

	Camille	Charlotte	Florine	Sami
Chargement d'image	X			
Suppression des couleurs			X	X
Prétraitement			X	X
Rotation Manuelle				X
Rotation Automatique			X	
Détection de la grille			X	
Détection des cases de la grille			X	
Récupération des chiffres		X		X
Présence de chiffre dans les cases		X		X
Réseau de neurones	X	X		X
Résolution de la grille		X		
Affichage de la grille résolue (UI)	X		s	
Sauvegarde de la grille résolue		X		

2.2 Etat d'avancement du projet

Ce tableau présente les tâches accomplies par les membres du groupe pour la dernière soutenance. Les T représentant l'état terminé de la tâche.

	Soutenance finale
Chargement d'image	T
Suppression des couleurs	T
Prétraitement	T
Détection de la grille	T
Détection des cases de la grille	T
Récupération des chiffres	T
Présence de chiffre dans les cases	T
Réseau de neurones	T
Reconstruction et résolution de la grille	T
Affichage de la grille résolue (UI)	T
Sauvegarde de la grille résolue	T

3 Aspects techniques

3.1 Chargement de l'image

Afin de charger l'image qui sera utilisée pour résoudre un sudoku, nous avons créé une fonction capable de convertir n'importe quel format d'image (png, jpg, jpeg...) en fichier bmp.

Pour ce faire, nous avons utilisé la librairie SDL.h que nous utiliserons tout au long de ce projet afin de pouvoir travailler sur l'image. Pour pouvoir modifier une image il faut utiliser des SDL_Surface. Grâce au SDL_Surface, nous pouvons accéder à la taille de l'image mais aussi à chaque pixel de celle-ci pour les lire ou les modifier.

3.2 Suppression des couleurs

3.2.1 Niveau de gris

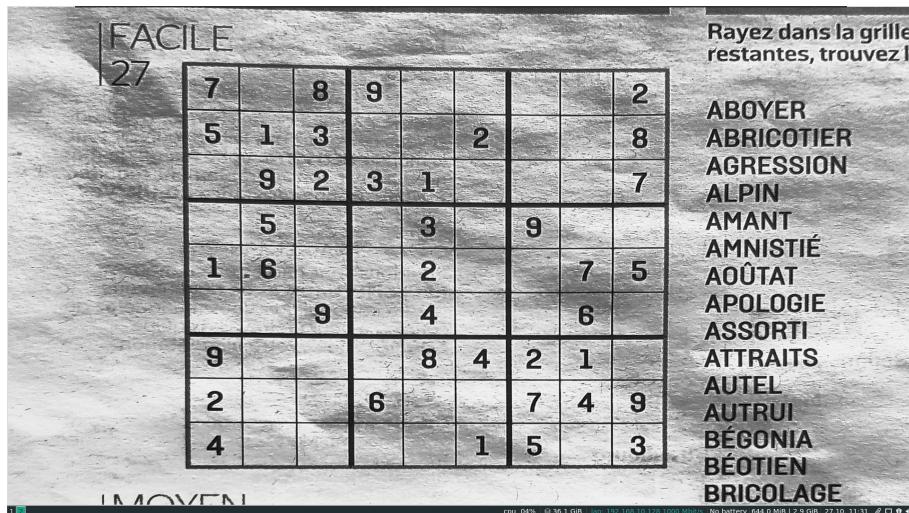
Afin de pouvoir travailler sur une image, il est nécessaire de supprimer les couleurs qui la composent. En effet celles-ci rendent le travail de détection de la grille plus compliqué.

Pour supprimer les couleurs de notre image, nous avons tout d'abord transformé celle-ci en niveaux de gris. Pour créer un pixel gris, il suffit d'utiliser la même valeur pour chacune de ses composants RGB (Red, Green, Blue) (en français : Rouge, Vert, Bleu). La conversion d'un pixel de couleur à un pixel gris est donc la moyenne de ses trois composantes RGB. Une fois cette moyenne obtenue, il suffit de changer la valeur des composantes RGB par la valeur de la moyenne.

Pour calculer la moyenne des composantes RGB, nous avons utilisé la formule de pondération de la recommandation 709 :

$$\text{Moyenne} = 0.3 * \text{Rouge} + 0.59 * \text{Vert} + 0.11 * \text{Bleu}$$

Pour transformer l'image en niveau de gris, nous avons donc parcouru chaque pixel de l'image et calculé la moyenne des composantes RGB puis changé la valeur des composantes RGB du pixel par la valeur de la moyenne.

*Image après conversion en niveau de gris*

3.2.2 Noir et blanc

L'intégration d'un algorithme permettant à une image en nuances de gris d'être passée en noir et blanc est essentielle pour permettre à notre OCR de détecter la grille, les cases et par la suite reconnaître les caractères. C'est un concept qui paraît simple mais qui peut devenir très périlleux à mettre en place.

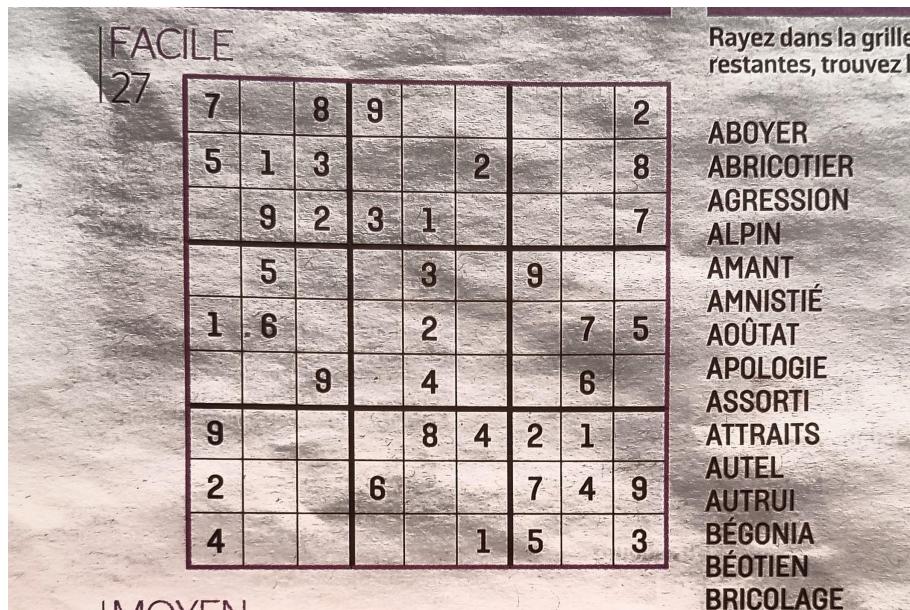
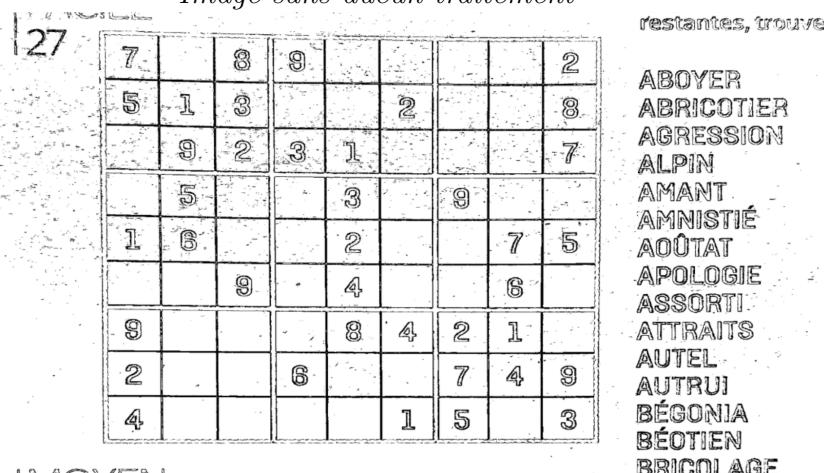
Pour transformer une image de sudoku qui possède déjà un fond homogène ou des contrastes élevés, la tâche est assez simple, il suffit d'appliquer un seuil (nombre pour lequel chaque pixel ayant une valeur de nuance de gris supérieure au seuil devient blanc sinon il devient noir) global qui séparera écriture et fond pour permettre le traitement de l'image dans les parties suivantes. Malheureusement cette méthode a des résultats très limités pour les images ayant un fond moins homogène ou des contrastes bas ce qui est le cas pour certaines images que nous avons à traiter.

Pour pallier à ce problème, nous nous sommes rabattu sur un algorithme de seuillage local qui va trouver un seuil sur de petites fenêtres (après de nombreux test nous avons trouvé optimal d'utiliser des fenêtres qui font 1/32 de la largeur de l'image) à travers l'image et leur appliquer le traitement expliqué plus tôt. Il existe de multiples méthodes de seuillage local plus ou moins efficaces.

Dans un premier temps nous avons utilisé la méthode des "local minima and maxima" qui va effectuer une opération utilisant le pixel le plus clair et le plus foncé de chaque fenêtre, la formule utilisée dans cet algorithme est très simple mais nous

avions des résultats très limités en particulier sur l'image 4 et l'image 6 qui renvoient une image intégralement blanche.

Ensuite nous avons utilisé la méthode de Sauvola qui utilise une formule mathématique plus compliquée utilisant les écart-types locaux et la variance. Les résultats avec cette méthode étaient plus satisfaisants mais l'image 4 gardait une certaine quantité de bruit et l'image 6 apparaissait toujours intégralement blanche.

*Image sans aucun traitement**Image après suppression des couleurs et application de la méthode de Sauvola*

Après de nombreuses recherches nous sommes ensuite tombés sur la méthode de Wellner qui utilise le principe d'image intégrale (l'image intégrale est une matrice dont chaque cellule a pour valeur la somme des pixels du rectangle supérieur gauche aux coordonées de cette cellule sur l'image que nous voulons traiter) et va prendre les pixels qui ont une valeur supérieure à la moyenne de chaque cellule de l'image intégrale. Cette méthode nous a permis d'avoir des résultats très concluants sur chacune des grilles peu importe le niveau de bruit ou le contraste de cette image.

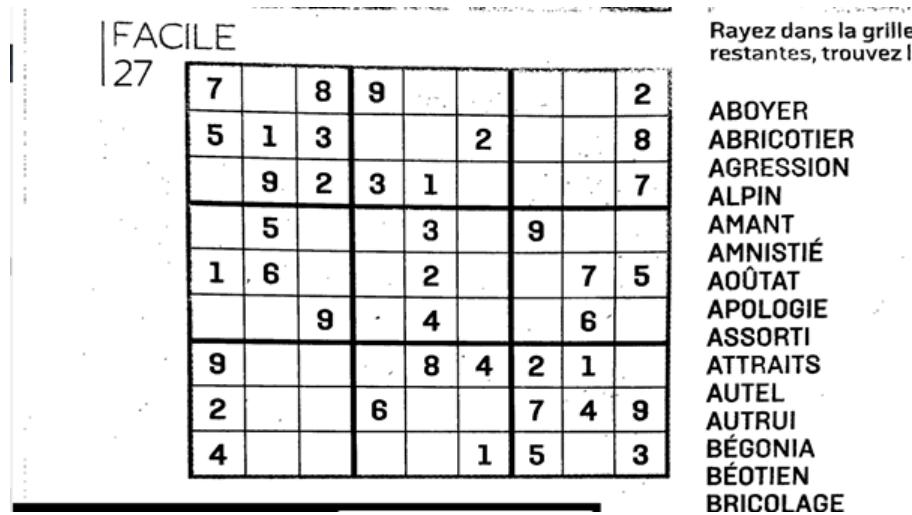


Image après suppression des couleurs et application de la méthode de Wellner

La création d'une matrice des sommes de tous les pixels de l'image est une méthode qui coûte beaucoup de mémoire, nous avons donc dû avoir recours à un calloc pour permettre à l'ordinateur de supporter cette charge en mémoire.

3.3 Prétaitemet

Après avoir binarisé l'image, nous avons appliqué différents traitements à celle-ci afin de faciliter la détection de la grille et des caractères.

3.3.1 Flou Gaussien et dilatation

Tout d'abord, nous avons commencé par réaliser deux opérations sur celle-ci:

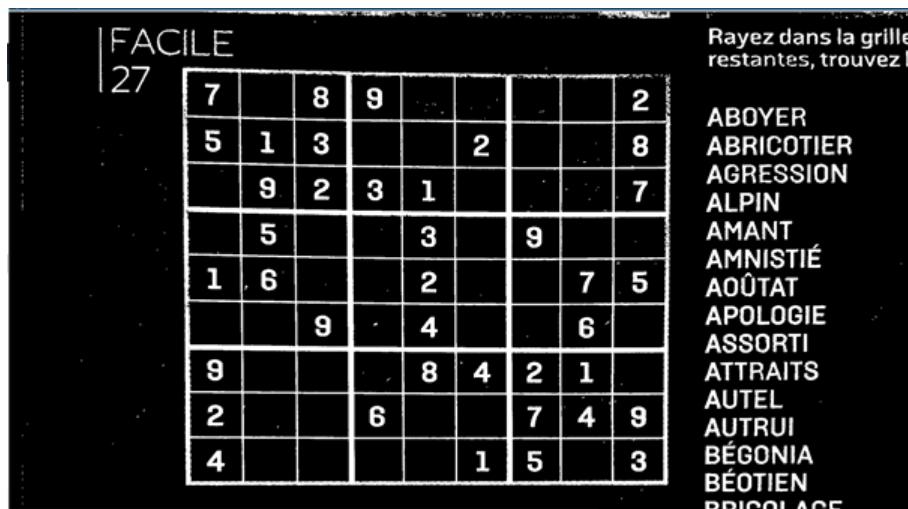
- Un flou Gaussien
- Une dilatation

Le flou Gaussien consiste à faire la moyenne des pixels entourant chaque pixel, permettant d'atténuer les fortes variations de couleur et rendant la suppression du bruit plus facile.

Pour notre dilatation, nous avons crée un algorithme qui va rendre noirs tous les pixels autour de chaque pixel noir, rendant les lignes et caractères plus visibles et faciles à détecter par les traitements suivants. Cela permet également de supprimer l'apparition de petits trous dans les lignes après l'application de la binarisation.

3.3.2 Inversion noir et blanc

Afin de faciliter les différentes détections et créer un contraste plus grand sur l'image ce qui fait ressortir la grille du sudoku, nous avons inversé les pixels noirs et blancs. Pour chaque pixel, si le pixel est blanc, nous l'avons coloré en noir et inversement. La grille de sudoku devient alors blanche sur un fond noir.



Images après inversion des pixels noirs et blancs

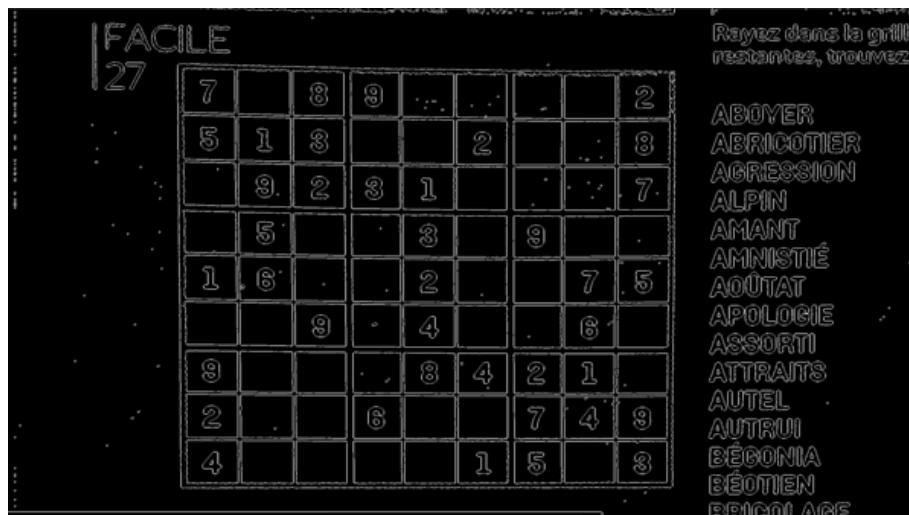
3.3.3 Filtre de Sobel

Le filtre Sobel est un filtre qui permet la détection des contours d'une image. Ce filtre multiplie un pixel et ses voisins avec deux matrices de taille 3x3 :

$$Gx = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} * P \text{ et } Gy = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * P$$

Après avoir appliqué les deux matrices, il suffit de faire faire la racine carrée de la somme des deux valeurs trouvées au carré afin d'avoir la nouvelle valeur du pixel:

$$G = \sqrt{Gx^2 + Gy^2}$$



Images après application du filtre Sobel

3.3.4 Rotation automatique

Pour pouvoir appliquer une rotation à nos images, nous avons utilisé une fonction très pratique de la librairie SDL 1.2 se nommant rotozoom, cette fonction permet de tourner et de zoomer une image.

Afin de détecter l'angle de rotation de l'image, nous avons utilisé la transformée de Hough sur l'image après application du filtre de Sobel. La transformée de Hough permet de détecter les lignes d'une image, nous expliquerons plus en détail son fonctionnement dans la partie *Détection des lignes*. Lors de la transformée de Hough, nous remplissons un accumulateur qui enregistre les angles θ et la distance ρ . Pour détecter l'orientation de la grille, nous avons donc cherché l'angle entre 90° et 135° qui a le plus d'accumulation. Une fois cet angle trouvé, il suffit de lui soustraire 90° . Cette soustraction nous donne l'angle à utiliser pour que la grille soit droite.

La fonction rotozoom créant une ligne blanche autour de l'image tournée, nous avons donc redécoupé l'image afin de ne plus avoir cette ligne. Celle-ci étant dérangeante pour la suite du traitement de l'image.

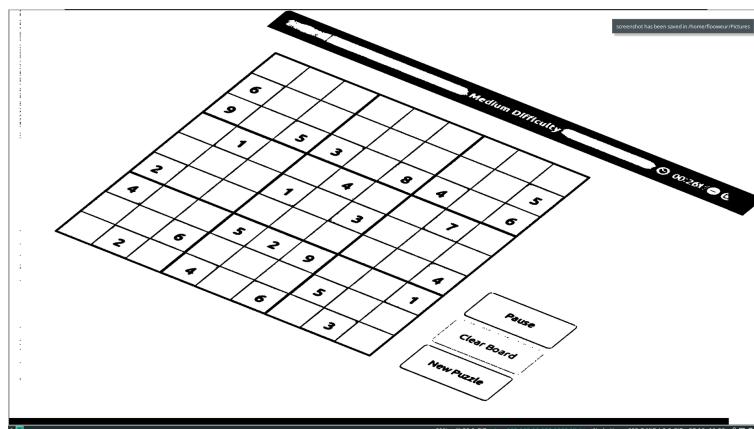
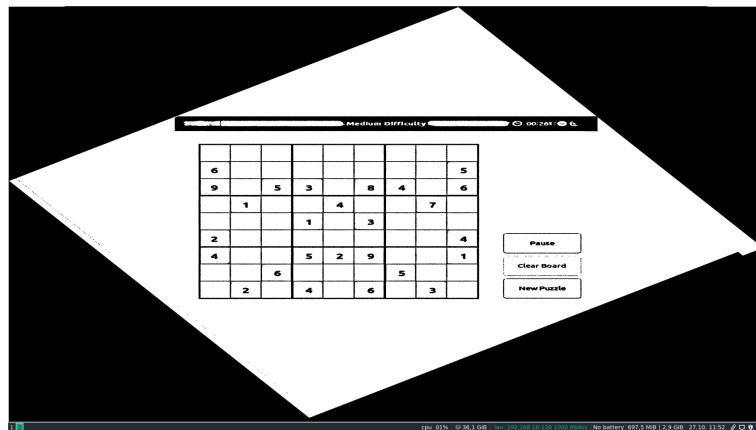


Image avant rotation

*Image après rotation de 35°*

3.4 Détection de la grille et des cases

Afin de pouvoir résoudre le sudoku sur l'image donnée par l'utilisateur, il faut que l'ordinateur détecte la grille présente sur l'image ainsi que les cases qui la composent. Pour cela, nous avons procédé à plusieurs changements sur l'image qui seront détaillés dans cette partie :

1. Détection et isolation de la plus grande composante connexe
2. Détection des lignes à l'aide de la transformée de Hough
3. Isolation de la grille
4. Isolation des cases

3.4.1 Détection et isolation de la plus grande composante connexe

L'image donnée par l'utilisateur n'étant pas seulement composée de la grille de sudoku, il est nécessaire d'isoler celle-ci pour pouvoir ensuite détecter les différentes cases qui la composent.

Pour isoler la grille, nous avons tout d'abord utilisé la méthode des composantes connexes. Cette méthode permet de regrouper les pixels de même couleur qui sont en contact dans l'image. Pour chaque groupe de pixels en contact et de même couleur, une couleur leur est attribuée. L'image de l'utilisateur ayant suivi un prétraitement, elle est maintenant composée seulement de pixels noirs (de valeur R = 0, G = 0 et B = 0) et de pixels blancs (de valeur R = 255, G = 255 et B = 255). La grille de sudoku étant représentée par les pixels de couleur blanche, les zones de composantes connexes cherchées seront donc les zones blanches. Nous avons utilisé l'algorithme

Hoshen-Kopelman qui est basé sur l'algorithme *Union-Find*.

L'algorithme *Hoshen-Kopelman* se déroule de la façon suivante :

- Pour chaque pixel de la grille, si le pixel rencontré est blanc alors nous vérifions :
 - Si le pixel n'a pas de pixels voisins blancs alors un nouveau "label" (une nouvelle couleur de cluster) lui est attribuée
 - Si le pixel a des voisins blancs alors sa nouvelle couleur est décidée par rapport à la couleur de ses voisins à l'aide de l'algorithme *Union-Find*

L'algorithme *Union-Find* consiste en :

- Find (trouver) : déterminer si deux pixels sont de même classe équivalente, si les deux pixels sont de classe équivalente alors le pixel dont nous cherchions à déterminer la couleur va prendre la couleur du pixel de classe équivalente.
- Union (unir) : réunir les pixels équivalents

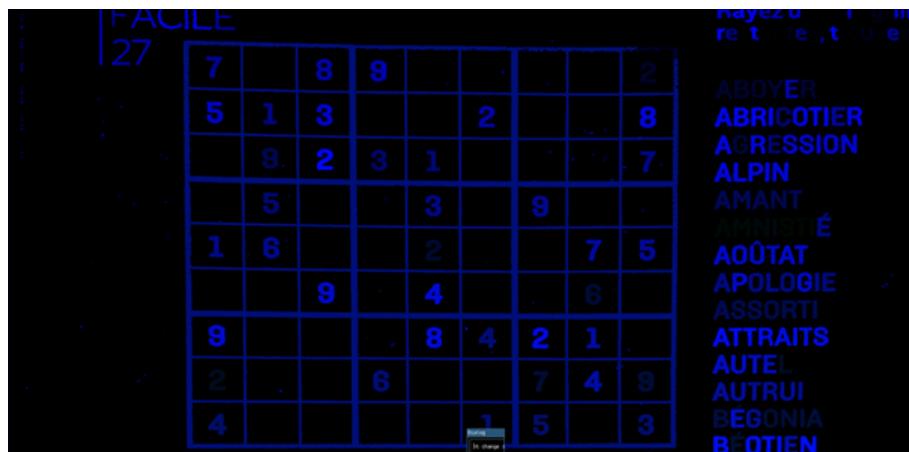


Image après détection des composantes connexes

Nous avons considéré que la grille de sudoku était la plus grande composante connexe de l'image. Nous avons donc retiré toutes les autres composantes pour ne conserver que la plus grande composante, la grille de sudoku. Celle-ci devient donc la seule composante blanche sur le fond noir de l'image.

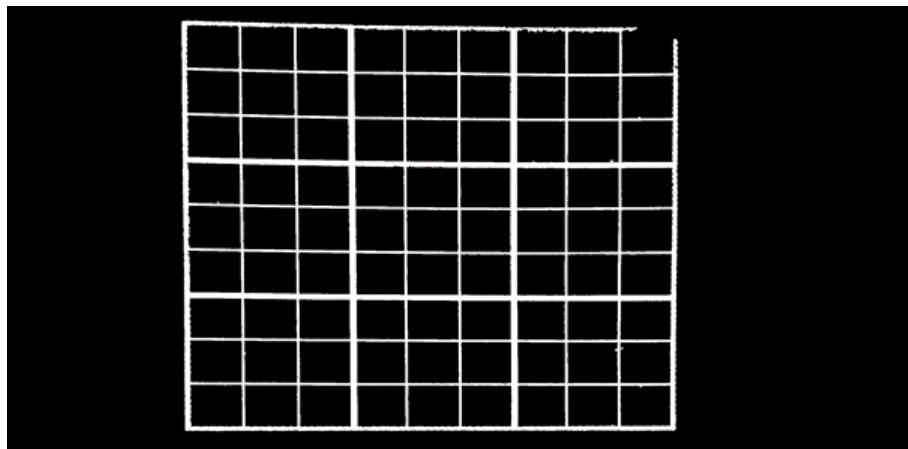


Image après isolation de la grille

3.4.2 Détection des lignes à l'aide de la transformée de Hough

Une fois la grille isolée, il est devenu plus facile de détecter les lignes horizontales et verticales qui la composent. Pour détecter ces lignes nous avons utilisé la transformée de Hough. Celle-ci consiste en l'utilisation de ρ et θ . Chaque ligne étant un vecteur de coordonnées θ et ρ :

- ρ : la distance du segment perpendiculaire à la droite d'angle θ et passant par l'origine du repère
- θ : l'angle

Chaque ligne vérifie : $\rho = x \cos(\theta) + y \sin(\theta)$

Nous avons donc créé un accumulateur qui va enregistrer et incrémenter les valeurs de θ (θ étant entre 0 et 180 dégrées) et ρ en les calculant pour chaque pixel de notre image. Nous avons ensuite tracé les lignes donc l'angle θ était de valeur 0 ou 90. 0 étant l'angle pour les lignes horizontales et 90 étant l'angle pour les lignes verticales.

Pour pouvoir exploiter ces lignes plus tard, nous les avons tracées de différentes couleurs :

- Vert représentant les lignes verticales
- Bleu représentant les lignes horizontales
- Rouge représentant les intersections entre les lignes verticales et les lignes horizontales

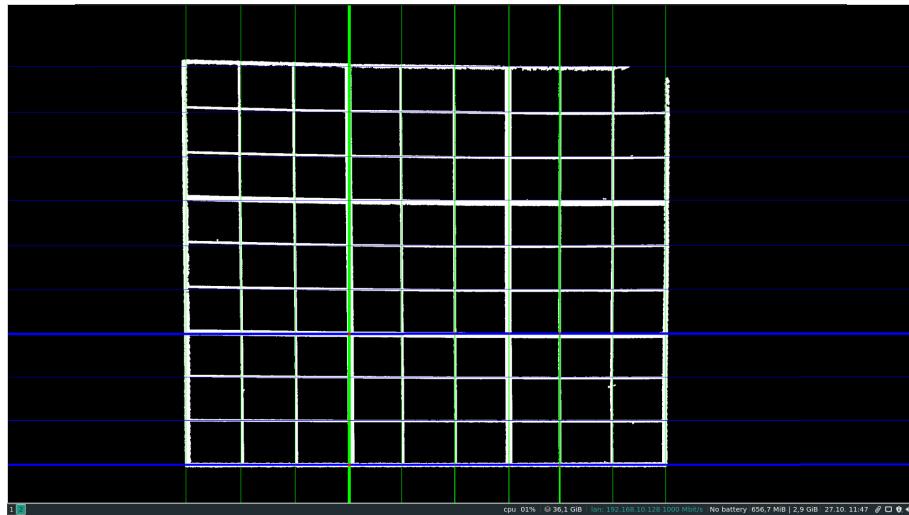


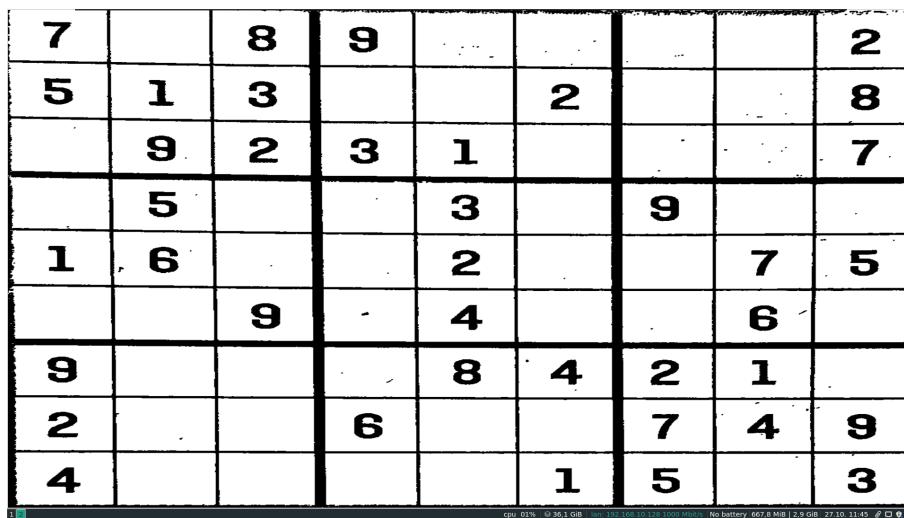
Image après détection des lignes

3.4.3 Isolation de la grille

Une fois les lignes détectées la prochaine étape fut d'isoler la grille de sudoku. Pour isoler la grille de sudoku, nous avons enregistré la position (x,y) de chaque coin de la grille grâce aux lignes déterminées précédemment. En effet, la grille étant maintenant la seule composante de notre image, toutes les lignes tracées à l'aide de la transformée de Hough représentent la grille de sudoku.

Il nous a donc suffi de partir de chaque extrémité de l'image et d'avancer jusqu'à détecter la couleur d'une ligne (vert ou bleu) ou d'une intersection (rouge). Si la couleur détectée est le rouge on peut en déduire que nous avons trouvé un coin de la grille de sudoku. Si nous avons la couleur d'une ligne alors nous continuons le long de celle-ci jusqu'à rencontrer un pixel rouge dont nous allons enregistrer la position. Étant parti de chaque coin de notre image nous avons donc récupéré les quatre coins de la grille.

A l'aide des quatre coins détectés, nous avons créé une nouvelle image de la taille de la distance entre chaque coin et l'avons remplie des pixels étant à l'intérieur des quatre coins.

*Image après détection des composantes connexes*

3.4.4 Isolation des cases

La dernière étape fut d'isoler chaque case de la grille dans des images différentes. Pour cela, nous avons seulement découpé l'image de la grille de sudoku créée précédemment en 81 images de longueurs et largeurs identiques. En effet, une grille de sudoku étant généralement un carré composé de petits carrés, ce découpage permet d'isoler correctement chaque case.

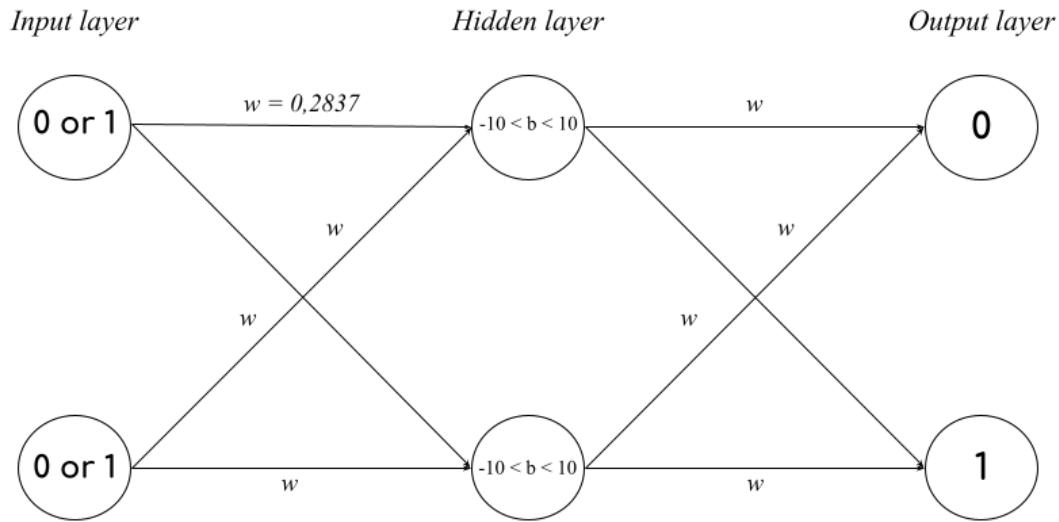
*Case on (3,0) après le découpage de la grille*

3.5 Réseau de neurones

Pour réaliser un réseau de neurones capable d'apprendre la fonction OU EXCLUSIF, nous nous sommes beaucoup aidés de documentations et de vidéos explicatives trouvées sur internet afin de comprendre comment ce mécanisme marchait. Ainsi, nous avons créé 3 couches de neurones sous forme de listes avec des tailles variant selon le nombre de neurones qu'elles contiennent :

- La couche d'entrée contenant 2 neurones (Input layer)
- La couche cachée contenant 2 neurones (Hidden layer)
- La couche de sortie contenant 2 neurones (Output layer)

La couche d'entrée possède deux éléments - deux neurones - chacun pouvant prendre les valeurs 0 ou 1. Ces derniers sont reliés aux neurones contenus dans la couche cachée avec des liaisons possédant chacun des "poids" (weights) différents, tandis que des "biais" (bias) sont affectés aux neurones de la couche cachée : ces deux valeurs permettent de faire varier "l'importance" du neurone étudié, c'est à dire son impact sur la déduction finale du programme, ce qui déterminera alors quelle valeur sera lue à la couche de sortie.



3.5.1 Forward propagation

Pour calculer les valeurs de chaque neurone de la couche cachée ainsi que ceux dans la couche output, nous utilisons une fonction d'activation sur chacun d'eux.

Nous avons opté pour la fonction sigmoïde : $f(x) = \frac{1}{1+e^{-x}}$ avec $x = \sum_{i=0}^n weight[i] * layerprecedente[i] + bias$, n étant le nombre de neurone du layer précédent.

3.5.2 Back propagation

Après que nous soyons arrivés à un résultat après un test, nous devons procéder à une back propagation. Lors de cette propagation, nous mettons à jour les valeurs de chaque biais et chaque poids des neurones contenus dans la couche hidden et la couche output jusqu'à ce que la valeur d'erreur finale se rapproche le plus possible de 0. Pour savoir par quelle valeur chaque élément devra être remplacé, nous utilisons les quatre formules élémentaires de la back propagation, trouvées sur le site [Neural networks and deep learning](#).

3.5.3 Détection de caractères

Pour la deuxième soutenance, nous devions modifier notre réseau de neurones pour qu'il puisse apprendre 9 caractères différents plutôt que d'apprendre la fonction OU EXCLUSIF.

Cela rend la tâche plus compliquée car au lieu de prendre 2 entrées, il doit prendre 784 entrées. Ce qui correspond à chaque pixel dans une image qui correspond à une case remplie ou non. De plus pour avoir des résultats plus précis, nous avons choisi d'utiliser une couche de 64 neurones cachés. Nous avons donc trois couches:

- La couche d'entrée contenant 784 neurones qui vont prendre différentes valeurs tirées d'un fichier contenant des images en MNIST.(Input layer)
- La couche cachée contenant 64 neurones qui va s'occuper de vérifier les liens entre les neurones pour nous permettre une détection plus précise. Au début nous avions décidé de ne pas utiliser de couche cachée pour un réseau de neurones plus simple mais nos résultats tournaient autour de 80% de réussite. Nous avons réussi à passer à 92% en utilisant cette couche cachée de 64 neurones(Hidden layer)
- La couche de sortie contenant 10 neurones correspondants aux 10 caractères possibles : 1, 2, 3, 4, 5 ,6, 7, 8, 9 et le caractère nul qui seront ensuite utilisés dans le résolveur pour résoudre la grille (Output layer)

Entre chaque neurone se trouve un lien appelé un bias qui au début est un nombre aléatoire entre 0 et 1 et qui au fur et à mesure de l'entraînement va changer de valeur afin de détecter plus fidèlement les caractères.

Pour pouvoir avoir des valeurs correctes dans les bias, le réseau de neurones doit subir un entraînement, pour l'entraîner nous allons lui passer un fichier MNIST de 60000 images et un fichier MNIST de 60000 résultats afin qu'il adapte ses bias en fonction de s'il devine quelle image correspond à quel résultat ou non.

Après avoir subi un entraînement, il suffit de donner une image au format MNIST au réseau de neurone et il renverra à quel entier cette image correspond.

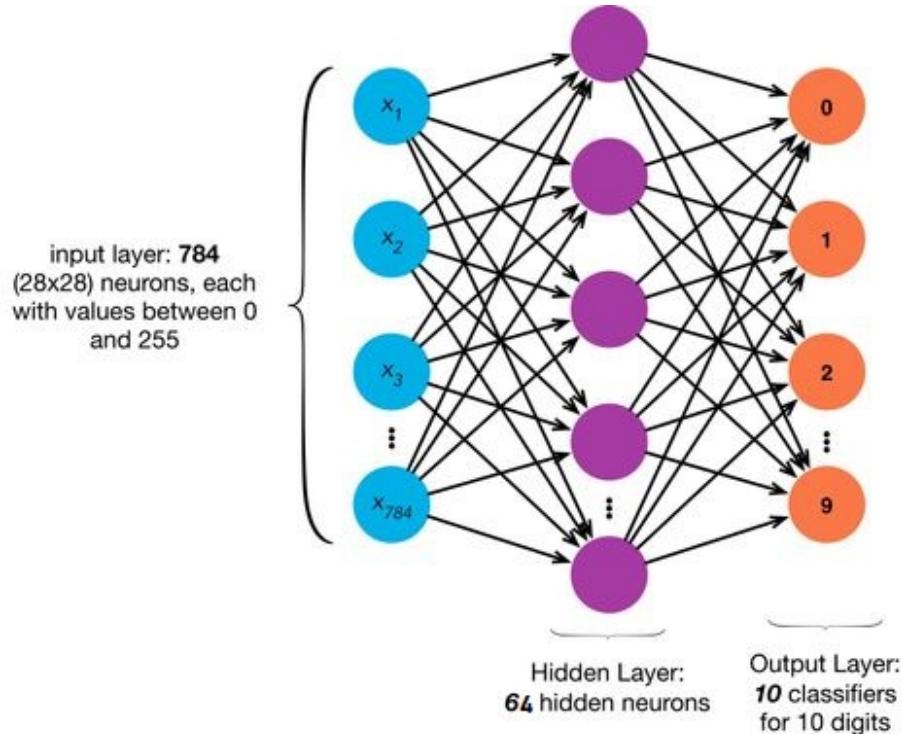


Schéma représentant notre réseau de neurones

Pour le côté technique du réseau de neurones :

- Lors de l'entraînement : Le réseau de neurones va récupérer chaque image de la base de données de 60000 images et va faire subir au réseau de neurones un "feed forward" qui correspond à faire passer l'image par chaque couche du réseau et obtenir un résultat à la fin. Ce résultat sera très probablement incorrect au début de l'entraînement car les bias entre chaque couche est une valeur aléatoire mais devrait devenir de plus en plus précis au fur et à mesure.

Après avoir subit ce "feed forward" le réseau de neurones va comparer son résultat avec le résultat attendu qui est inscrit dans une autre base de données

et si le résultat est incorrect il va subir une "back propagation" qui va changer les poids de chaque bias en utilisant une formule mathématique Sigmoïde afin de l'aider à détecter le caractère présent dans chaque image de manière plus précise.

- Après que le réseau de neurones ait été entraîné, il est apte à reconnaître un caractère avec une exactitude de 92% dans notre cas. Pour lui faire reconnaître un caractère il suffit de les "feed forward" à partir d'un fichier contenant chaque case de notre sudoku préalablement transformée en MNIST.

3.5.4 Exportation des images

Le format des fichiers de la base de données MNIST est un format bien particulier qui permet d'avoir toutes les informations d'une image tout en étant beaucoup moins lourd et en étant beaucoup plus facile et rapide à lire pour le réseaux de neurones.

Nous avons donc décidé de rassembler toutes nos images obtenues après le découpage de l'image, c'est à dire toutes les cases de la grille, dans un fichier de ce même format.

Chaque fichier de la base MNIST commence toujours par un "nombre magique". Nous avons mis quelques instants à comprendre ce qu'il représentait et nous pensions juste au début que c'était un nombre au hasard. Après avoir lu entièrement la page d'informations sur cette base de données, nous nous sommes rendu compte que ce nombre avait tout de même une certaine importance. En effet, celui-ci permet d'indiquer au programme qui lira le fichier, le type de données du fichier mais aussi le nombre de dimensions des données du fichier.

Il faut ensuite écrire le nombre de données présentes dans le fichier : qui sera dans notre cas 81 car il y a 81 cases pour chaque grille de sudoku.

Enfin, juste avant d'entrer les données, il faut donner la taille de de la grille. C'est à dire que, en fonction de la dimension qu'on a donnée dans le nombre magique, nous devons écrire un certain nombre de tailles. Ici chaque cases étant en 2D, on va écrire 2 nombres qui seront 28 et 28, l'image étant un carré. Le nombre 28 à lui été défini arbitrairement, mais il définira ensuite la taille de chaque image du fichier.

Pour écrire les données, les images, nous avons tout d'abord du redimensionner les images, puisque elles sont au départ beaucoup trop grandes et mettraient donc trop de temps à être parcourues. Nous en avons donc dimensionné les images en 28 par 28 comme dit précédemment.

Ensuite, il a fallu parcourir chaque pixel de chaque image puis écrire leurs correspondances dans le fichier. Une image étant en noir et blanc grâce au traitement précédent, un pixel noir était représenté par un 255 et un pixel blanc par un 0.

Chiffre 9 dans la base de donnée MNIST

3.6 Résolution du sudoku

Pour la résolution de la grille, nous partons d'un fichier suivant un certain format contenant toutes les informations sur la grille de sudoku récupérée. Puis nous résolvons la grille et enregistrons cette grille dans un nouveau fichier en suivant le même format. Nous avons utilisé la méthode de backtracking pour construire notre fonction.

3.6.1 Backtracking

Pour ce faire, nous avons utilisé l'algorithme de *Backtracking* suivant :

1. Récupérer les informations du fichier sous la forme d'un tableau en initialisant les cases vides avec des 0.
2. Utiliser une fonction récursive sur le tableau. Cette fonction va tester, pour chaque élément égal à 0 du tableau, tous les nombres de 1 à 9 en testant le premier nombre sur la première case et en testant si la grille est toujours valable. Puis en mettant le premier nombre sur la deuxième case et en testant si la grille est toujours valable etc. Dès que l'on arrive sur une grille non valable (plusieurs fois le même chiffre dans une grande case, une colonne ou une ligne), on termine la récursion et on revient donc dans la boucle de récursion précédente qui change la valeur de la case actuelle pour le chiffre d'après. Le programme s'arrête lorsque la grille est complètement remplie.
3. Réenregistrer le tableau obtenu dans un nouveau fichier final.

Après la première soutenance, ayant pris en compte les conseils de nos professeurs, nous avons décidé d'ajouter quelque chose à notre programme. Cette chose est le fait que le code détecte avant d'aller plus loin et de commencer toute la récursion pour le backtracking si la grille est valide. En effet, une grille est valide seulement si chaque lignes, chaque colonnes et chaque carré de 9x9 ne comporte un nombre qu'une seul et unique fois. Dans le cas où la grille n'est pas valide, le programme renvoie une erreur et dit à l'utilisateur que sa grille n'est pas valide.

En testant d'autre grille, je me suis aussi rendu compte que certaine grille, bien que valable, ne pouvait pas être résolue. C'est à dire qu'il peut y avoir deux chiffres égaux qui forcent un autre chiffre à être à un certains endroits mais un autre chiffre qui conflicterait avec ce choix précédent. Dans ces cas là, tous le backtracking sera tout de même effectué, mais le code remarquera que la grille est resté la même qu'au départ et renverra aussi une erreur en disant à l'utilisateur que la grille n'est pas solvable.

3.6.2 Format du fichier

Le format du fichier d'entrée et de celui de sortie est un format très simple, il comporte 11 lignes et 11 colonnes. Chaque lignes/colonnes comportent des chiffres ou des points sauf la 4ème et la 8ème ligne/colonne qui permettent de délimiter chaque carré et de rendre le format plus facile à lire pour un humain.

Chaque caractère a la place (x, y) représente le caractère dans la case de coordonnée (x, y) de la grille d'entrée. Les chiffres sont représentées par eux même tandis que les cases vides deviennent des points.

```
... .4 58.  
... 721 ..3  
4.3 ... ...  
  
21. .67 ..4  
.7. ... 2..  
63. .49 ..1  
  
3.6 ... ...  
... 158 ..6  
... ..6 95.
```

Format du fichier contenant une grille de sudoku

3.7 Interface graphique

Notre interface graphique est composée de deux zones : la zone de boutons et la zone de dessin.

3.7.1 Drawing Area (Grille)

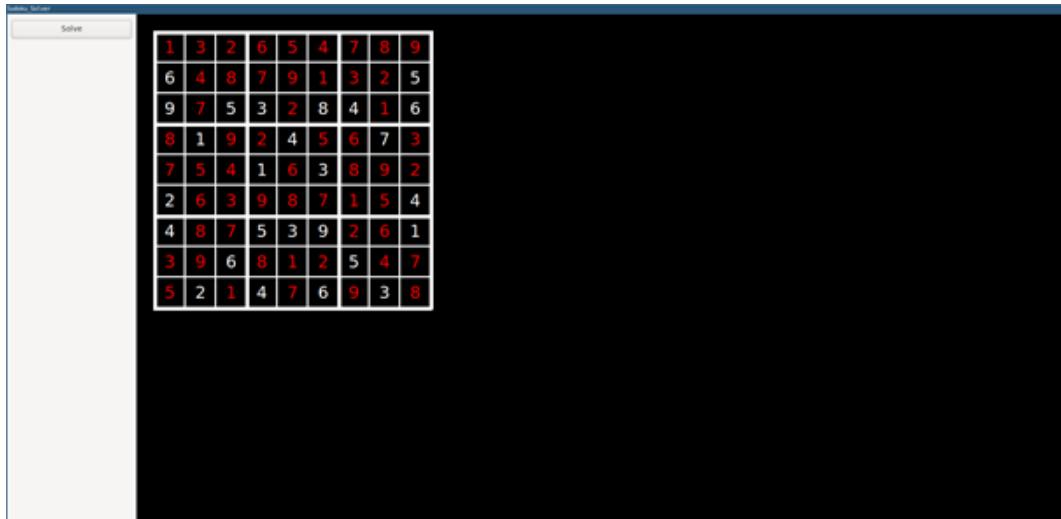
Afin d'afficher la grille dans l'interface graphique, nous avons utilisé la bibliothèque Cairo. Celle-ci nous permet de tracer des lignes horizontales et verticales, de la taille et de la couleur que nous souhaitons. Ainsi, nous avons pu dessiner la grille, celle-ci de forme carrée et composée au total de 81 cases.

Ensuite, il fallait afficher les chiffres dans leurs cases respectives. Pour faire cela, nous parcourons le fichier que notre fonction solveur crée, qui est un fichier .result. Ce fichier est composé de caractères, de 9 lignes et 9 colonnes, et contient soit des chiffres entre 0 et 9, soit un " ", qui correspond alors à une case vide pour une grille de sudoku pas encore résolue.

De plus, il serait intéressant que l'utilisateur puisse voir la différence entre la grille résolue et le résultat que l'ordinateur a créé : ainsi, les étapes de l'interface graphique se font comme suit :

- Tout d'abord, l'utilisateur indique au programme quelle image de sudoku il voudrait faire traiter.
- Ensuite, l'image passe dans nos fonctions de traitement d'image.
- Après que l'image ait été traitée, elle passe dans le réseau de neurones, là où chaque chiffre de la grille de sudoku sera récupéré et traduit en valeur numérique traitable.
- L'étape d'après est de résoudre le sudoku. Pour cela, les données du sudoku entrent dans la fonction de solveur.
- Enfin, les données de la grille résolue entrent dans l'interface graphique, où chaque chiffre sera dessinée dans la zone de dessin de la fenêtre Gtk. Une autre couleur sera utilisée pour les chiffres qui ont été déduits par notre programme (qui n'étaient donc pas sur l'image de base).

Nous choisissons une couleur (noir) pour la couleur des chiffres, leur taille et leur police, puis les plaçons un à un dans la Drawing Area. Et voilà, nous obtenons la grille de sudoku résolue dans notre interface graphique !



Interface graphique avec la grille résolue

4 Ressenti des membres du groupe

4.1 Charlotte Buat

Ce projet a été très enrichissant pour moi. En effet, il nous a permis de découvrir plusieurs nouvelles facettes de l'informatique tels que le traitement d'une image ou encore la création et le fonctionnement d'un réseau de neurones, le sujet sur lequel je me suis le plus renseigné. Malgré les difficultés parfois j'ai beaucoup aimé le concept des réseaux de neurones.

J'adore toujours autant la programmation, malgré que le travail en groupe à été parfois un peu compliqué.

4.2 Camille Nguyen

J'ai trouvé le projet OCR très intéressant. J'ai pu travailler sur le réseau de neurones de la première soutenance, celui du XOR, sur le chargement d'image et sur l'interface graphique. Le plus amusant pour moi était l'interface graphique; pouvoir voir les choses que je voulais dessiner sur la zone de dessin Gtk était très satisfaisant. J'ai eu du mal à trouver comment afficher un texte ou un chiffre dans la zone de dessin, ayant presque abandonné cette idée pour la place dessiner des chiffres traits par traits comme sur une horloge digitale, mais grâce à l'aide de mes coéquipiers, j'ai finalement trouvé comment faire à l'aide de Cairo.

Pour résumer, ce fut un projet complexe et assez stressant mais je suis très reconnaissante de mon groupe pour en être arrivés jusqu'ici.

4.3 Florine Kieraga

Ce projet fut pour moi l'occasion de découvrir ce qu'était l'OCR, notamment le traitement d'une image afin de pouvoir détecté une grille de sudoku. J'ai apprécié le fait qu'il fallait chercher par nous même les différents algorithme présent dans un OCR.

Le plus compliqué pour moi fut la détection de la grille. En effet, j'ai du essayer plusieurs méthodes différentes avant de trouver une méthode qui fonctionnait sur toutes les images. En plus de trouver la bonne méthode, une autre difficulté fut d'implanter l'algorithme en C.

4.4 Sami Carret

Pour ma part j'ai trouvé ce projet très intéressant et il m'a beaucoup apporté sur plusieurs domaines différent. Je me suis découvert une passion pour le traitement d'image ainsi que pour l'intelligence artificielle et j'ai hâte de retravailler sur un projet traitant d'une de ces domaines.

Bien que la charge de travail fût assez élevée j'ai pu arriver à bout de tout ce que je voulais faire et je suis très content du résultat. Je pense que ma plus grande difficulté a été de trouver une bonne méthode de binarisation polyvalente qui ne prenait pas trop de temps d'exécution et rendait un résultat propre. Le travail de groupe a été un peu difficile en particulier à cause des problèmes de git mais au final nous avons réussi à trouver un mode de fonctionnement qui marchait bien.

5 Conclusion

Pour conclure, nous avons réalisé toutes les tâches demandées pour cette dernière soutenance. Notre OCR est donc maintenant capable de prendre un fichier rentré par l'utilisateur, lui appliquer un traitement (suppression des couleurs, rotation, suppression du bruit etc..), détecter la grille et les cases du sudoku, reconnaître les chiffres présents dans les cases et résoudre le sudoku.

Ce projet fut pour nous l'occasion d'en apprendre plus sur le traitement d'une image ainsi que le réseau de neurones qui composent un OCR. Mais aussi, il nous a appris la gestion du temps, en effet, le temps entre les différentes soutenances étant très court, il nous a fallu nous organiser pour arriver à réaliser toutes les tâches demandées. Pour cela nous avons organisé des réunions hebdomadaires afin de garder un oeil sur l'avancée de chaque membre du groupe.

5.1 Bibliographie

- * Algorithme Hoshen–Kopelman
- * Transformée de Hough
- * Composantes connexes
- * Filtre de Sobel
- * Neural networks and deep learning (Utilisé pour le backpropagation)
- * MNIST dataset