



Rapport de projet

Nicolas Bureau-Maury, Jean-Jules Sun, Nathan Simonin, Rémi Tribouillard

12 Décembre 2022

Table des matières

1	Introduction	2
1.1	Membres du groupe	2
2	Résumé des tâches	3
2.1	Répartition des tâches	3
2.2	Chargement d'une image	4
2.3	Suppression des couleurs et prétraitement	4
2.4	Détection de la grille	15
2.5	Rotation de l'image	22
2.6	Découpage de l'image	25
2.7	Réseau de neurones : Détection de chiffre	27
2.8	Réseau de neurones (new) : Détection de chiffre	29
2.9	Résolution du sudoku	31
2.10	Affichage du résultat sur l'image	32
2.11	Interface graphique	34
3	Conclusion	38

1 Introduction

L'équipe **Norenface** est heureuse de vous présenter son projet de **Sudoku Solver** !

L'ambition de ce projet était d'allier nos compétences et connaissances en programmation, analyse d'image, recherche... pour créer sudoku solver qui pourra résoudre vos sudokus peu importe les environnements.

1.1 Membres du groupe

Les membres de ce groupe font partie de l'équipe "Norenface" créée lors du semestre 3 de l'année 2022. Ces quatre étudiants font leurs études à l'école des ingénieurs en intelligence informatique EPITA.

Norenface comprend donc quatre membres qui sont :

- Nathan Simonin : Il connaît plusieurs langages, Python avec lequel il a commencé en terminale, Caml, C et C qu'il a découvert cette année et html/css qui serviront pour le site web. Sa motivation fait qu'il peut faire des nuits blanches sans limite afin de régler tout bug. "Finalement, le sommeil à quoi ça sert ?"
- Rémi Tribouillard : Il a commencé la programmation au lycée, en prenant NSI en première et terminale. Il adore les maths, c'est pour ça il va dans une association de maths une fois par semaine. "Penser ça tue le cerveau."

- Jean-Jules Sun : Comme les autres membres de ce projet, la programmation fait partie de sa vie depuis son entrée à l'EPITA. Désormais, avec son sweat à capuche réservé aux programmeurs de haut niveau, il est capable de programmer avec différents langages. "Genshin c'est la vie, mais la programmation est plus que la vie."
- Nicolas Bureau-Maury : Passionné par les nouvelles technologies et gadgets en tout genre. Il commence à toucher à la programmation dès son plus jeune âge et est captivé. Tout au long des multiples projets qu'il mène depuis presque 3 ans, il a acquis ce qui lui permettra d'apprendre encore d'avantage pour réaliser ce super projet, et le mener au bout avec l'équipe Norenface. "La programmation, c'est comme une drogue!"

2 Résumé des tâches

2.1 Répartition des tâches

	Réponsable :		Suppléant	
	Nathan	Rémi	Jean-Jules	Nicolas
Rotation de l'image				
Suppression des couleurs				
Prétraitement				
Détection de la grille				
Découpage de la grille				
Résolution et sauvegarde du sudoku				
Réseau de neurones				
Jeu d'image				
Affichage du sudoku				
Interface				

2.2 Chargement d'une image

Pour le chargement de l'image, nous avons tout simplement repris un de nos TP sur SDL2 où il fallait ouvrir des images, ajuster la fenêtre... Ce code est donc dans la plupart de nos fichiers pour vous montrer nos résultats lors de la première soutenance. Ce code va donc marcher de la façon à ouvrir une fenêtre, afficher l'image et si la fenêtre est trop petite pour l'image, alors la fenêtre s'ajuste automatiquement.

2.3 Suppression des couleurs et prétraitement

La suppression des couleurs et le prétraitement de l'image va nous permettre d'analyser une image en toute sérénité. Le prétraitement va enlever tout ce qui est des taches sur l'image, des ombres ou même d'imperfection.

Grayscale

Pour l'analyse de l'image, passer d'une image en RGB en mode gris est primordiale afin d'avoir un meilleur traitement. Et les astres se sont liés ! Effectivement, nous avons fait un TP de programmation sur SDL2 que nous utilisons pour ce projet. Un exercice de ce TP était de faire passer une image en RGB en mode gris ! C'est pour cela que nous avons réutilisé cette fonction dans ce projet.

Mais comment passer de RGB en mode gris sur une image ? C'est très simple, il vous suffit de parcourir chaque pixel de l'image afin de récupérer ses valeurs R, G et B et lui appliquer une petite formule magique.

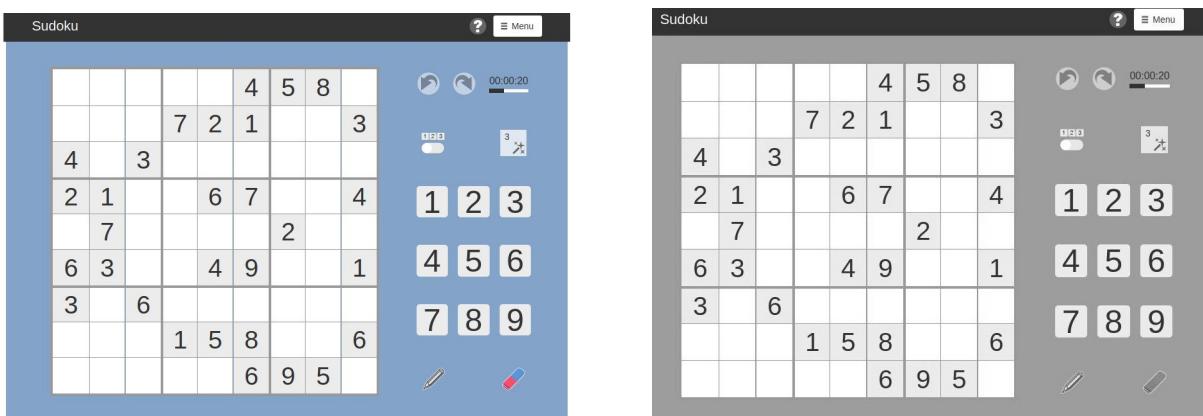
Après avoir récupéré une valeur grâce à cette formule, il suffit de l'appliquer aux valeurs R, G, B du pixel. Après ça, les valeurs RGB du pixel seront les mêmes. Et nous passerons donc au mode gris.

```
Uint32 pixel_to_grayscale(Uint32 pixel_color, SDL_PixelFormat* format)
{
    Uint8 r, g, b;
    SDL_GetRGB(pixel_color, format, &r, &g, &b);

    Uint8 average = 0.3*r + 0.59*g + 0.11*b;

    return SDL_MapRGB(format, average, average, average);
}
```

Le résultat de toute cette manipulation nous permet d'avoir les résultats suivant :



	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		

BÉGONIA
BÉOTIEN
BRICOLA
BUNGAL
BUTANIE
CABANE
CAPTATI
CENTAIN
COSY

L O
O D
N P
O R
E S
G Q
I E

	4			1	5	3	
MOYEN							
28	2			6	4	2	
	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		

BÉGONIA
BÉOTIEN
BRICOLA
BUNGAL
BUTANIE
CABANE
CAPTATI
CENTAIN
COSY

L O
O D
N P
O R
E S
G Q
I E

Flou

En effet, dans le flou basique, nous allons analyser les pixels un par un et faire la moyenne des valeurs de tous ses voisins afin d'obtenir un nouveau pixel.

Mais ce flou ne corrige pas certaines parties de l'image.

C'est pour cela que le filtre de Gauss est meilleur. Il suit un peu le cheminement du flou basique mais en multipliant tous les voisins de chaque pixel dans un rayon de 1 par une matrice de Kernel comme vous pouvez le voir ci-dessous :

245	24	24	23	87			
47	89	44	87	23			
5	31	46	15	123	X		
145	145	12	12	42			
145	145	54	156	156			

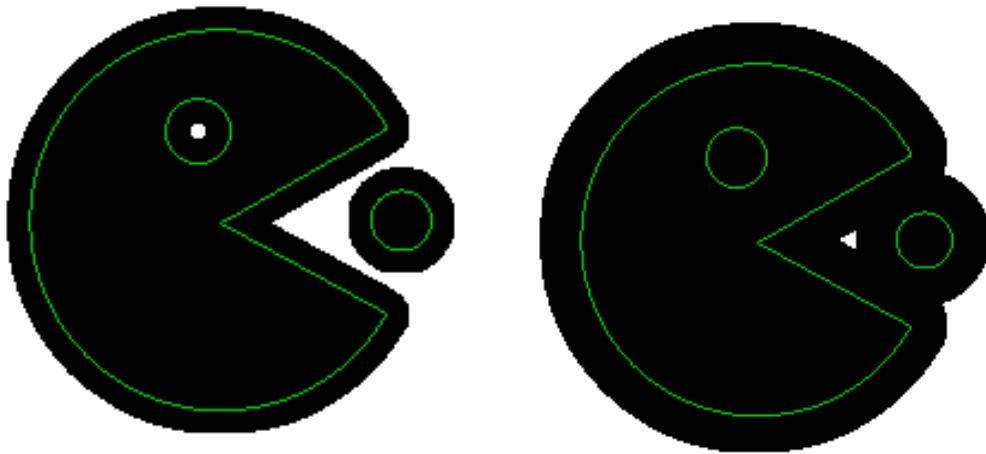
1	2	1		
2	4	2		
1	2	1		

X 1 / 16

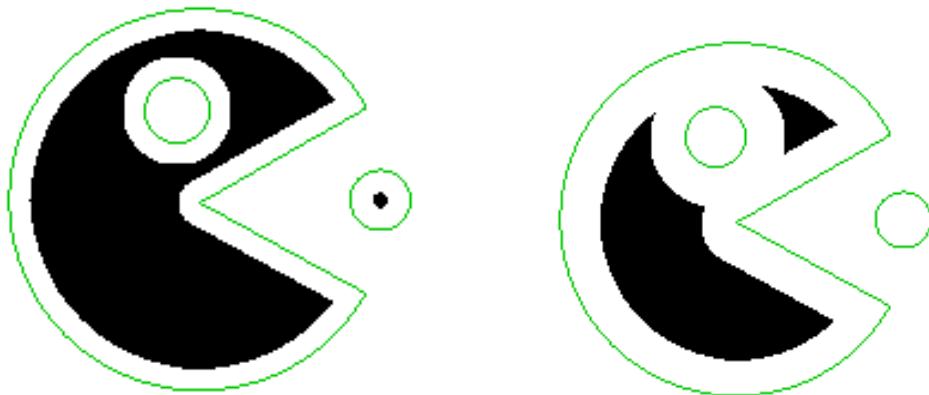
Dilatation et érosion

La dilatation et l'érosion sont 2 traitements qui combiné vont nous permettre de corriger des imperfections de l'image et en plus de faire ressortir les lignes noires ce qui va aider la suite du traitement.

La dilatation consiste donc à récupérer la valeur minimum entre les 8 voisins d'un pixel actuel avec un certain multiplicateur obtenu grâce à une matrice bien défini. Puis à lui appliquer cette valeur. Ceci va permettre de "dilater" l'image, c'est-à-dire, grossir toutes les valeurs de noir de l'image et donc obtenir des images de la sorte :



L'érosion va marcher de la même façon, sauf qu'ici, on fait le chemin inverse, on multiplier par une matrice les voisins du pixel, puis nous allons prendre le maximum de ses voisins et nous allons lui appliquer cette valeur. Voici encore une fois un petit exemple de ce que nous obtenons :



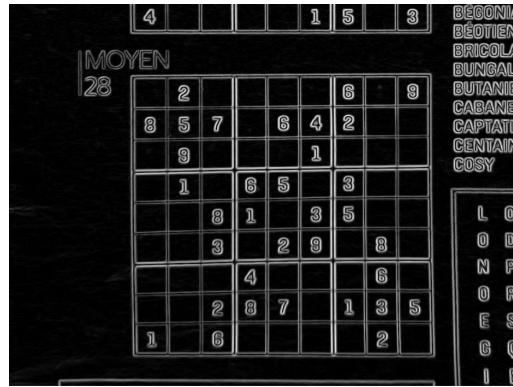
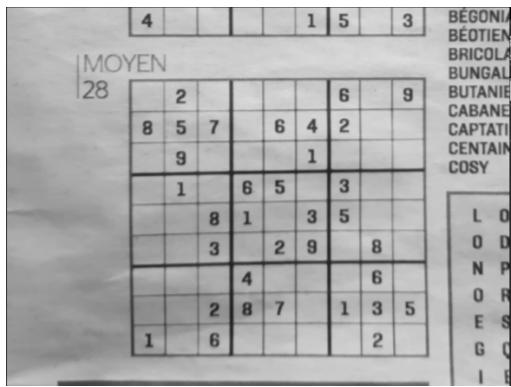
Ce traitement peut paraître insignifiant sur nos images, mais sans cela, de nombreuses petite tâche blanche peuvent apparaître lors de traitement future ce qui engendrera une difficulté au niveau de la binarisation.

Sobel

Plus tôt, nous avons parlé de Sobel, et bien nous utilisons cet algorithme dans le prétraitement de l'image ! Celui-ci nous permet de faire une manipulation sur notre image qui va nous aider grandement pour la prochaine et dernière étape du prétraitement, la binarisation. Il consiste à faire ressortir les lignes horizontal et vertical présente sur l'image en faisant un calcul encore et toujours sur les voisins d'un pixel, et d'une multiplication d'une matrice sur eux ! Voici les matrices qui vont être utilisées :

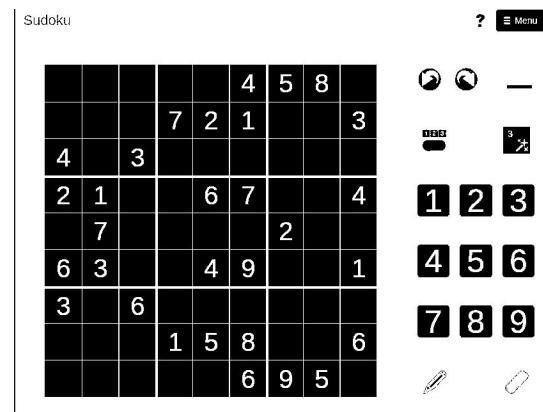
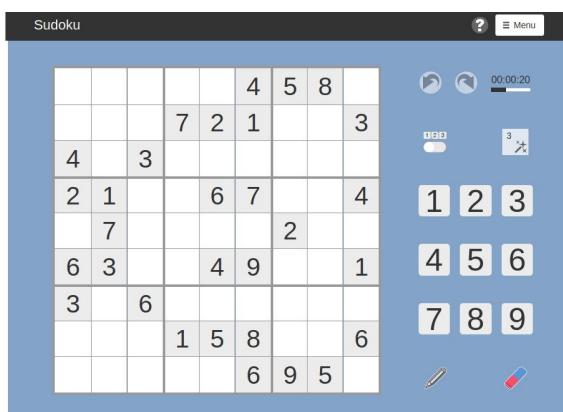
Gx			Gy		
-1	0	1	1	2	1
-2	0	2	0	0	0
-1	0	1	-1	-2	-1

Nous allons donc créer 2 listes de pixels, une pour la ligne horizontale et une pour la ligne verticale. Dans les 2 cas, nous allons, sur un pixel faire la somme de tous ses voisins multipliés par leur valeur respective dans les matrices, puis nous faisons la moyenne des voisins et nous appliquons cette moyenne à notre pixel actuel (avec une valeur max de 255). Ensuite, nous réunissons nos 2 nouvelles listes de pixel en faisant un petit calcul sur chacun des pixels étant : la racine de la somme des carrés des 2 valeurs du pixel dans nos listes. Nous obtenons avec ça une image qui va donc retirer la plupart des ombres de l'image comme ci-dessous :



Binarisation

La binarisation est la dernière et la plus importante des parties pour faire l'analyse de l'image. Binariser une image signifie qu'elle va être transformée en pixel soit noir ou blanc. Les actions comme le flou permettent justement d'améliorer cette transformation pour avoir un très bon résultat pour l'analyse. Plusieurs méthodes sont possibles afin d'exécuter cette action. Mais nous nous sommes tourné vers la méthode d'Otsu qui consiste à donner un seuil lors du passage sur chaque pixel de l'image. Enfin, si ce seuil est dépassé par le pixel, alors il deviendra blanc, sinon il deviendra noir.

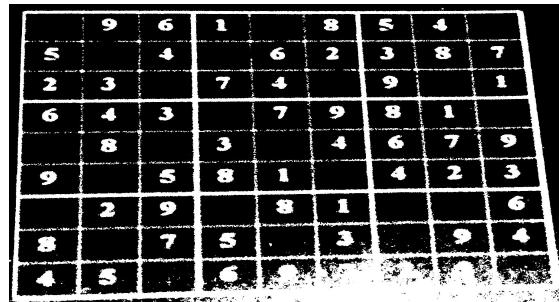
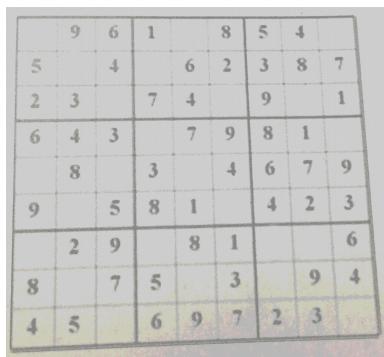


Seulement, sur les images de sudoku que les utilisateurs pourront nous donner, il peut y avoir des ombres par exemple, et les images en contenant vont avoir une analyse assez compliquée, car elles vont aussi ressortir après avoir appliqué la binarisation.

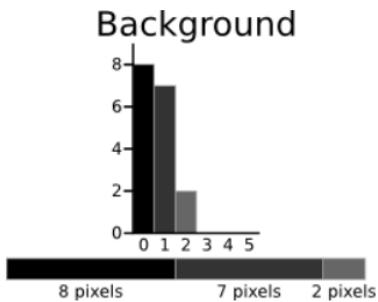
- Tout d'abord, le seuil fixe. Il consiste à donner directement une valeur comprise entre 0 et 255 (généralement fixées à 127). Seulement, cette technique n'est pas valable pour toute sorte d'image. Par exemple, une image trop blanche deviendra toute noir, car le seuil est trop haut peu importe si des pixels ressortent plus que d'autre.

Ensuite, il y a le seuil global. Celui-ci consiste à faire une moyenne de la valeur de chaque pixel de l'image ce qui donne un plutôt bon résultat de la binarization sur la plupart des images.

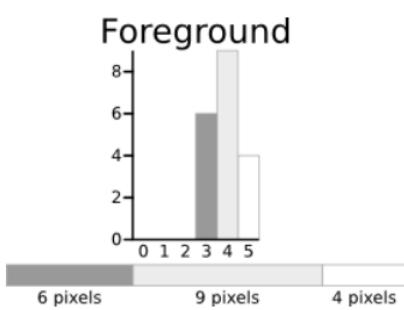
- Enfin, il y a le seuil local. Dans cette dernière méthode, on va changer de seuil pour chaque pixel en prenant la moyenne des pixels voisins dans un rayon défini (généralement 1 ou 2). Mais cette technique fait apparaître de nombreuses imperfection.



C'est pour cela que nous avons finalement découvert une meilleure version de Otsu qui utilise un histogramme contenant toutes les valeurs de nos pixels. Il y a des calculs de variance, et bien d'autre qui permette d'avoir le meilleur seuil sur chaque image. Voici une petite démo des calculs qui sont utilisé :



$$\begin{aligned} \text{Weight } W_b &= \frac{8 + 7 + 2}{36} = 0.4722 \\ \text{Mean } \mu_b &= \frac{(0 \times 8) + (1 \times 7) + (2 \times 2)}{17} = 0.6471 \\ \text{Variance } \sigma_b^2 &= \frac{((0 - 0.6471)^2 \times 8) + ((1 - 0.6471)^2 \times 7) + ((2 - 0.6471)^2 \times 2)}{17} \\ &= \frac{(0.4187 \times 8) + (0.1246 \times 7) + (1.8304 \times 2)}{17} \\ &= 0.4637 \end{aligned}$$



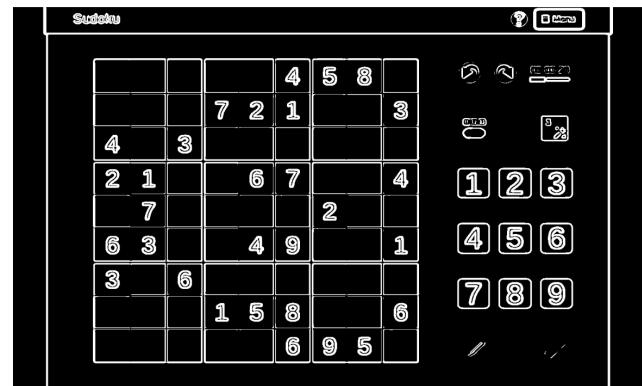
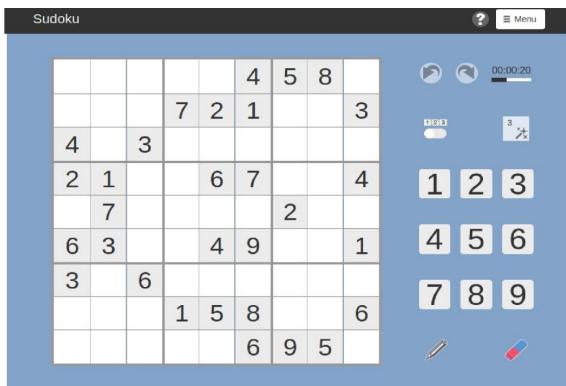
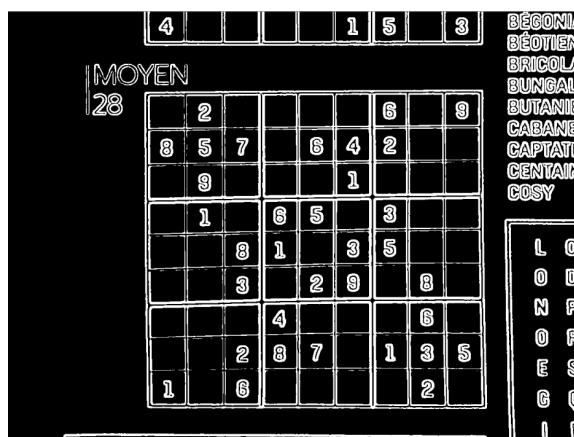
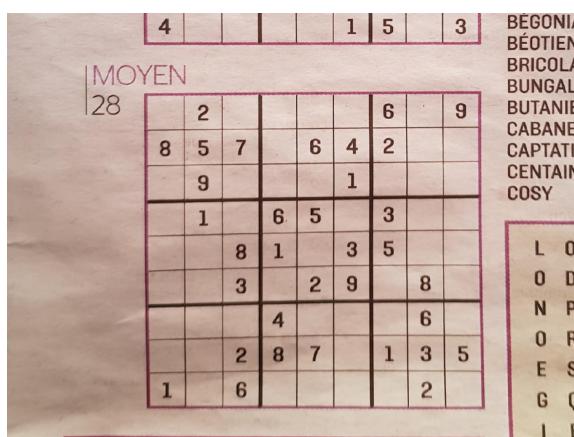
$$\begin{aligned} \text{Weight } W_f &= \frac{6 + 9 + 4}{36} = 0.5278 \\ \text{Mean } \mu_f &= \frac{(3 \times 6) + (4 \times 9) + (5 \times 4)}{19} = 3.8947 \\ \text{Variance } \sigma_f^2 &= \frac{((3 - 3.8947)^2 \times 6) + ((4 - 3.8947)^2 \times 9) + ((5 - 3.8947)^2 \times 4)}{19} \\ &= \frac{(4.8033 \times 6) + (0.0997 \times 9) + (4.8864 \times 4)}{19} \\ &= 0.5152 \end{aligned}$$

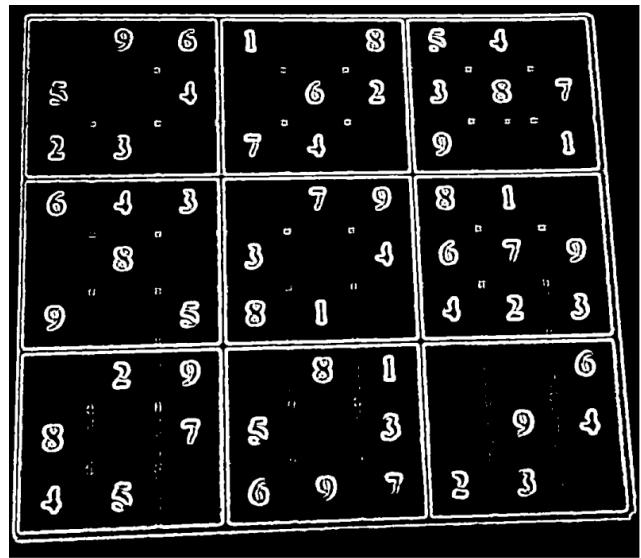
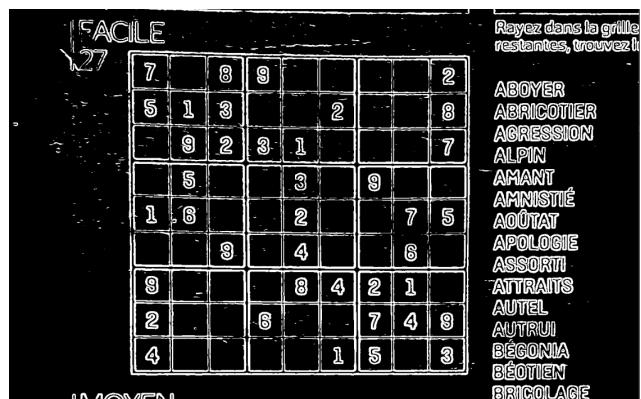
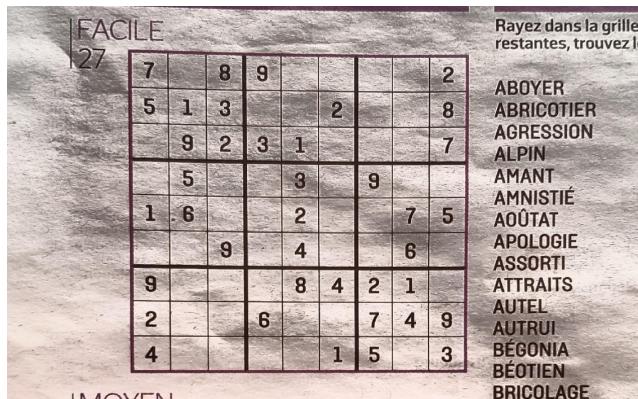
Pour résumer, nous essayons de garder la plus petite variance de l'image entre le background et le foreground pour trouver le meilleur seuil. Ceci permet une détection parfaite de la grille en retirant tout ce qui est superflu dans l'image.

Nous nous retrouvons finalement avec traitement de l'image parfaite pour la détection de la grille de sudoku, voici des exemples qui montre bien la différence entre l'image de base et à la fin du traitement :

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		

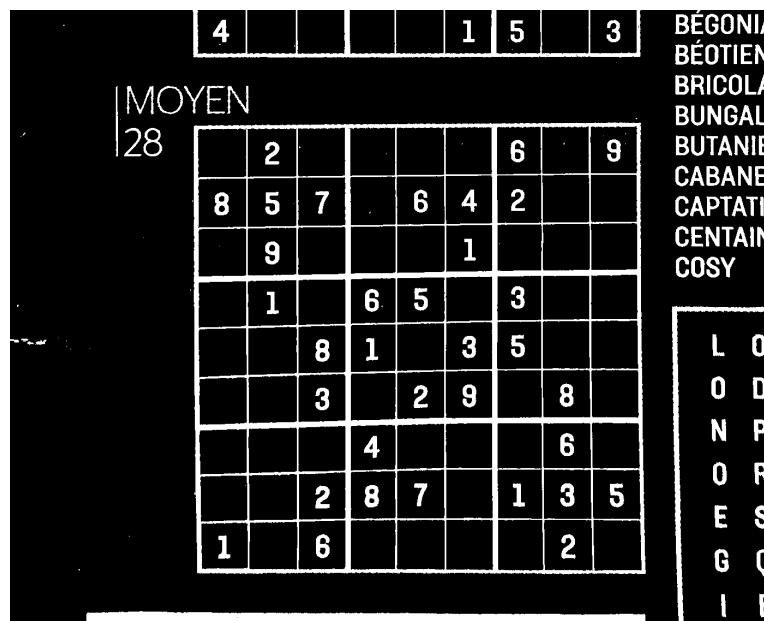
5	3			7				
6			1	9	5			
	9	8				6		
8			6					3
4		8	3			1		
7		2				6		
	6			2	8			
		4	1	9				5
			8		7	9		





2.4 Détection de la grille

La détection d'une grille de Sudoku est un processus qui prend en paramètre l'image en question après le binarisation et retourne un pointeur qui contient la taille de la grille, la taille des cases et les coordonnées des cases. Pour réaliser cette partie, plusieurs solutions ont été pensées et testées.



Tout d'abord, on a pensé à détecter la grille en cherchant la position de la première ligne droite qui est constituée par les pixels blancs. Puis de trouver les coordonnées des lignes et des colonnes de la grille pour avoir toutes les intersections possibles. Cependant, après avoir testé avec des exemples, nous nous sommes rendu compte que la grille du Sudoku n'est pas forcément le seul élément de l'image.

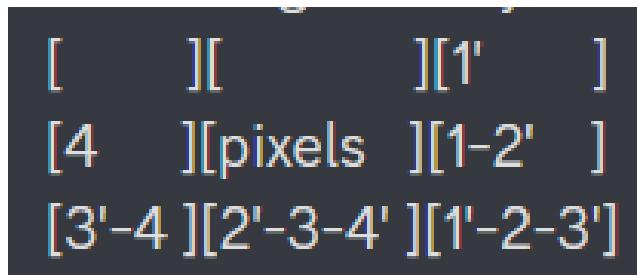
Ainsi, pour résoudre le problème, une autre solution a été développée. Afin de détecter la grille, les positions de tous les pixels blancs de l'image ont été enregistrés. Et suite à de multiples processus d'élimination, seuls les pixels qui se trouvent en haut à gauche des carrés formés par les pixels blancs ont été conservés. De plus, comme ces carrés sont situés dans la grille du Sudoku, les indices des pixels en question sont les coordonnées des carrés. Cette partie retourne donc les informations nécessaires pour le découpage de l'image.

	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		

	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		

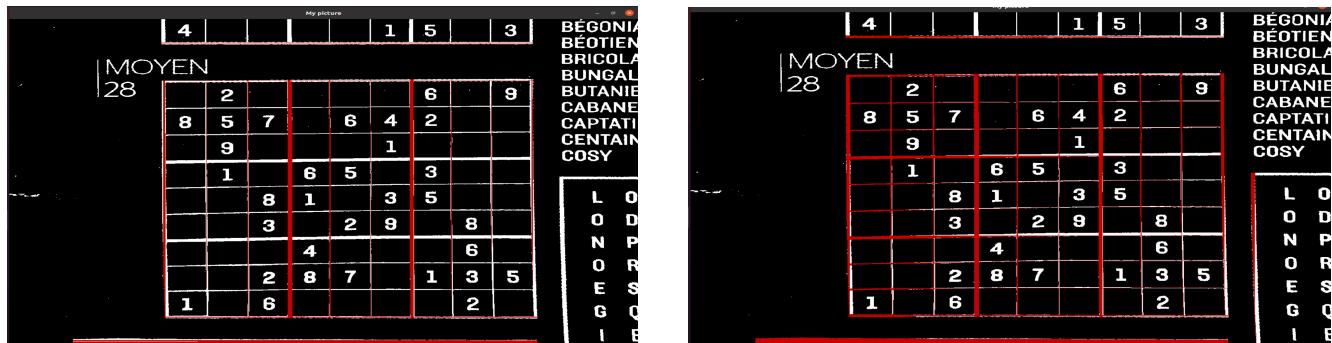
Après un certain temps, nous nous sommes rendu compte que si nous faisions comme ça nous n'allions pas pouvoir gérer tous les cas, car on ne peut trouver que des lignes verticales ou horizontales.

Pour résoudre ce problème nous avons mis au point un nouvel algorithme qui se base sur le précédent. Il commence à détecter les lignes d'une certaine longueur et les colore en rouge, la différence par rapport à notre algorithme d'avant, c'est qu'il est récursif et qui peut accepter des lignes qui ne sont pas parfaitement droites. Le procédé est très simple, nous sommes sur un pixel et nous regardons si le pixel qui est à coté de nous est blanc, si c'est le cas j'y vais, sinon je regarde si les deux voisins communs à ces deux point sont blancs si c'est le cas on se répète dessus. Cependant si nous lançons la récursion sur ses 9 voisins cela seraient trop coûteux en calcul. Comme nous parcourons l'image de haut en bas et de droite à gauche, nous pouvons avoir 4 angles différents, l'image ci-dessous permet de mieux illustré ce propos, sachant que les ' après un nombre désigne les pixels choisis si le pixel de l'angle n'est pas blanc.

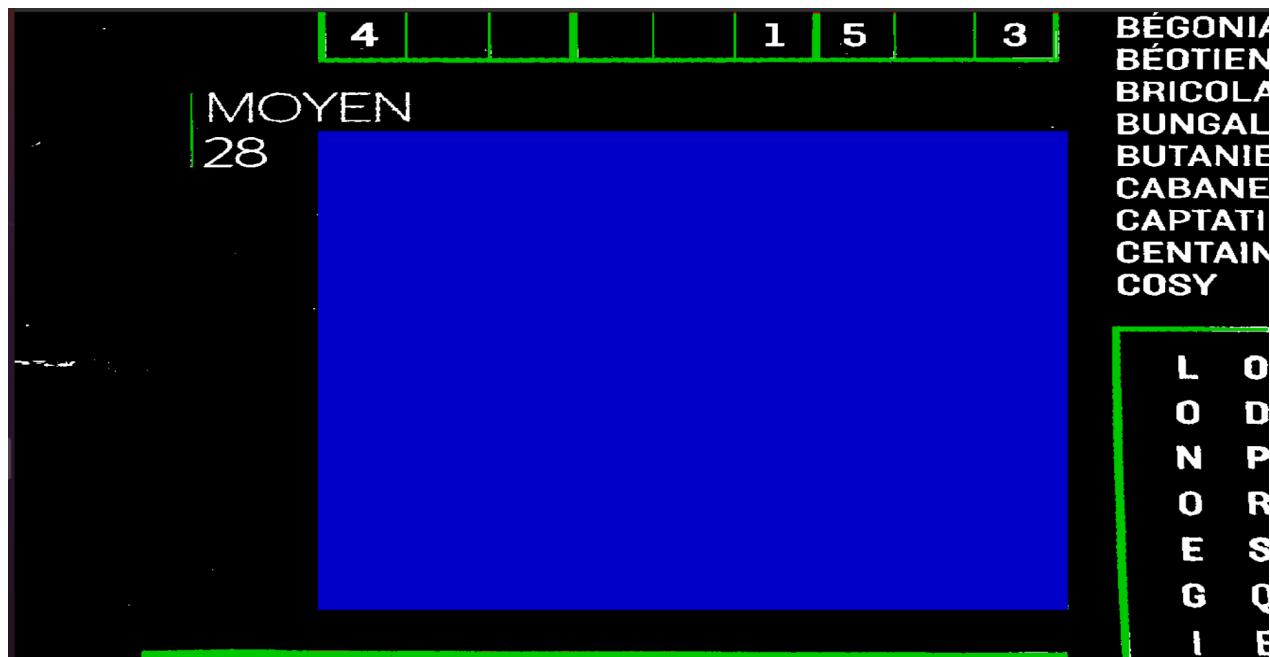


Cet algorithme possède aussi des défauts, il faut que la longueur de la ligne soit assez grande pour éviter de colorier tous les pixels blancs en rouge mais assez petite pour pouvoir s'adapter à chaque lignes peut importe son angle.

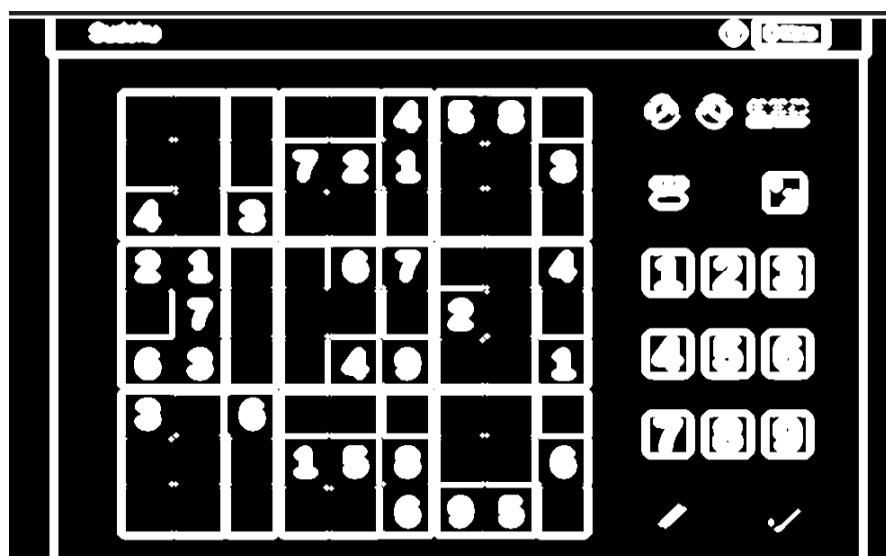
Nous pouvons voir ci-dessous deux images avec une longueur de ligne minimal différente. Sur la première nous avons une longueur de lignes minimales qui est trop grande, ce qui fait que nous détectons pas toutes les lignes.



Au moment où toutes les lignes sont coloriées en rouge, nous pouvons lancer la deuxième partie de notre algorithme, qui va chercher la plus grande surface générée par des pixels rouges, on créera un nouveau carré où se situe le sudoku, ce carré sera forcément droit par rapport à l'image. Cette plus grande surface devrait être la grille de sudoku, comme on peut le voir sur cette image qui coloris en bleu la position du sudoku.



Pourtant si nous avons une grille qui se trouve dans une autre forme, cela ne peut pas fonctionner. Si nous prenons comme exemple donné ci-dessous nous aurons une grande image bleue dans laquelle nous n'aurons pas que la grille de sudoku.



Pour résoudre ce problème, nous avons implémenter une fonction qui reconnaît si une surface est une grille. Pour ce faire, rien de plus simple, il suffit de voir si nous avons deux lignes vertical qui se trouve au tier et au deux tier du carré si ces lignes existe alors nous pouvons supposé qu'il s'agit d'une grille de sudoku.

Pour résoudre le problème de la rotation nous avions pensé utiliser les règles trigonométriques, en regardant le premier point le plus haut tout à droite et le point le plus à droite du haut de l'image nous pouvons savoir le degré de rotation à faire, puis vérifier si cette surface est bien une grille, mais il y a quelques problèmes, notre grille n'est pas parfaitement carrée, ce qui fait qu'il nous donne une rotation qui est dans le meilleur des cas justes et dans le pire des cas il peut même empêcher de bien reconnaître une grille. Nous avons donc abandonné cette idée.

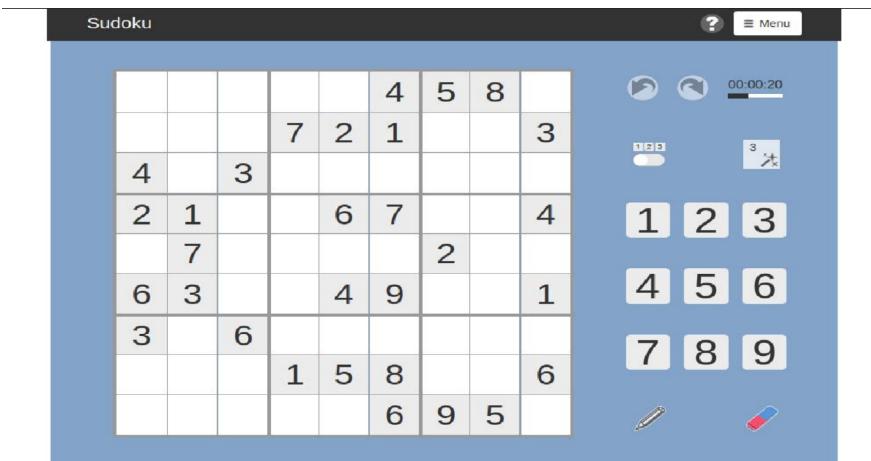
Nous avons donc pensé calculé l'aire de pixel noir se trouvant sur le dessus de la forme puis par un calcul, trouvé l'angle de rotation. Mais le problème est que la forme obtenue par sobel peut ne pas être droit, ce qui fait que nous aurions une estimation mais pas une valeur précise et qui est trop dépendante d'une image à une autre, c'est pour ça que nous n'avons pas gardé ce système.

Pour finir nous avons trouvé une dernière méthode, qui consiste à calculer l'aire de notre carré dans lequel il y a notre forme, faire une rotation sur celle-ci de telle manière à ce que l'aire de notre carré qui contient notre surface devient minimal. Si celui-ci se trouve êtres une grille de sudoku, elle sera forcément droite, ou toutes au plus avec un angle négligeable modulo 90° . Malgré ça ce procédé demande trop de calcul et de temps.

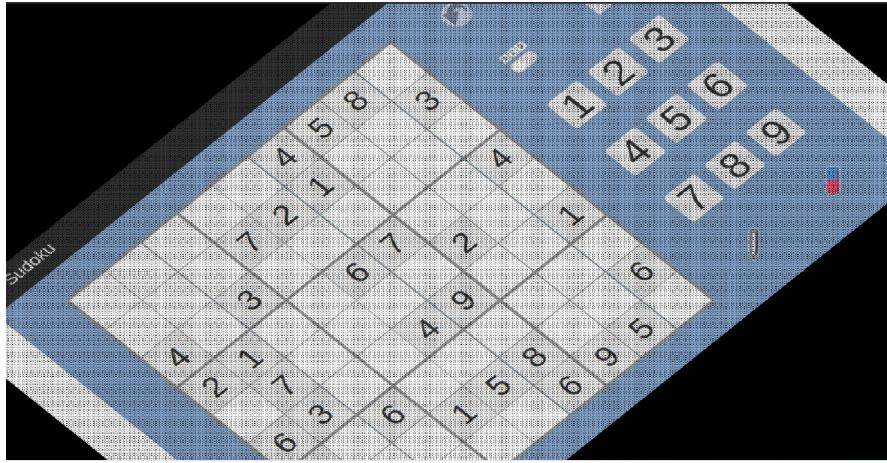
Ce qui fait que nous n'avons pas pu mettre de rotation automatique dans notre programme.

2.5 Rotation de l'image

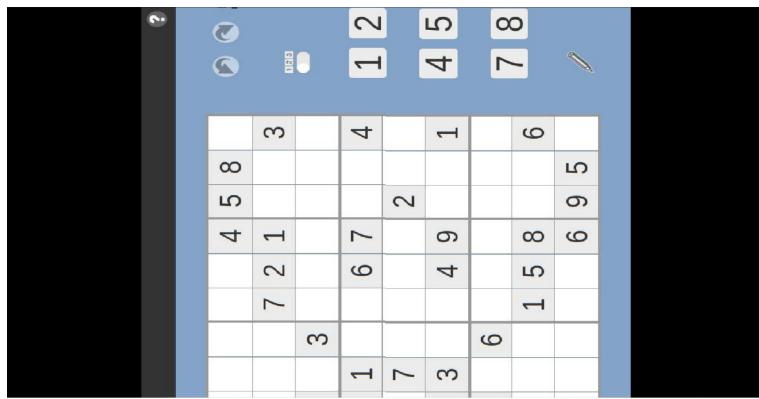
La rotation est une étape importante du processus de reconnaissance d'image en effet si nous ne pouvons retourner une image comment différentier un 6 d'un 9. De plus si nous ne pouvons faire des rotations manuelle alors nous pourrions recadrer l'image. Pour les rotations d'images nous avons choisi de faire une rotation discret cela nous permet de choisir un angle bien précis avec un centre de rotation qui peut être choisi. Cette rotation n'est pas non plus sans inconvénient, elle peut créer des pixels vides dans une image si l'angle n'est pas un multiple de 90, c'est d'eux à l'utilisation du sinus et cosinus dans la fonction. cette image est l'image originelle :



Cette image est celle obtenue après une rotation de 45° :



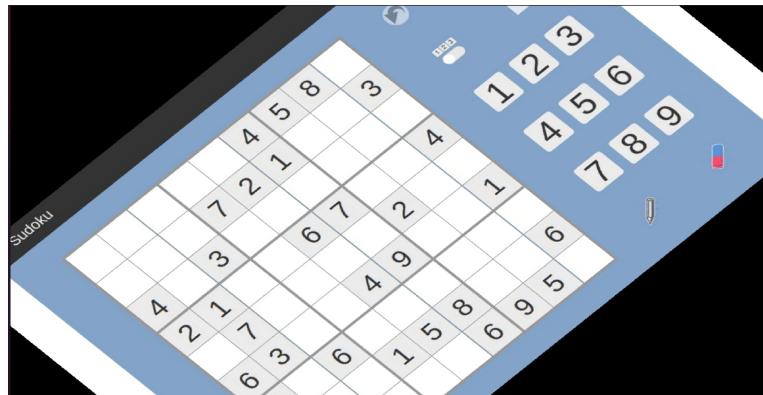
Cette image est celle obtenue après une rotation de 90° :



Nous pouvons remarquer que l'image dont la rotation est 90° n'a pas de perte de pixel, nous avons 2 bandes noirs sur le coté qui est dû à la dimension de l'image rectangulaire, nous faisons une rotation à partir du centre par défaut. Pourtant le fait de faire une rotation de 45° ne devrait pas faire disparaître des pixels, c'est une perte d'information non voulus et très problématique. Pour résoudre ce problème j'ai fait deux fonctions, la première vas coupé un pixel en un carré de taille voulus de pixel. Nous avons plus qu'à faire la rotation. Pour finir nous allons regrouper les carrée de pixels en faisant leur moyenne tout en ignorant les pixels noirs. Cette technique est coûteuse en mémoire et en calcul,

mais elle permet de ne plus avoir de perte d'information sur l'image.

Avec un angle de 45° cela donne cette image :



2.6 Découpage de l'image

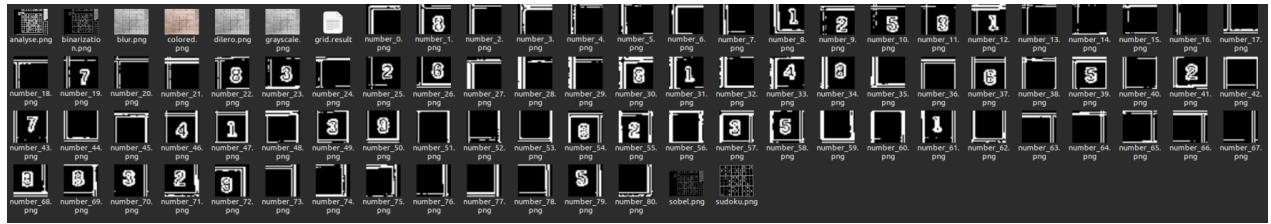
Cette partie du projet doit permettre de donner au réseau de neurones une image sur laquelle il peut déterminer le chiffre qui est contenu à l'intérieur de celle-ci.

Le résultat souhaité est un pointeur de type `SDL_Surface*` qui contient l'image de chaque case de Sudoku avec un parcours de grille de haut en bas et de gauche à droite. Pour ce faire, nous avons d'abord besoin des coordonnées de la grille obtenues dans la partie précédente. En effet, à partir de ces coordonnées, on peut calculer la longueur et la largeur des cases.

Nous créons donc en tout 81 surfaces qui constituent à elles toute la grille de sudoku. Pour remplir ces surfaces, nous découpons l'image en $9*9$ qui vont nous permettre de sauvegarder les bons pixels de l'image de base.



Ces images ont été tout d'abord sauvegardées pour faire apprendre au réseau de neurones les chiffres et maintenant, elles sont toutes sauvegardées dans un dossier afin d'avoir une petite illustration de cette partie.



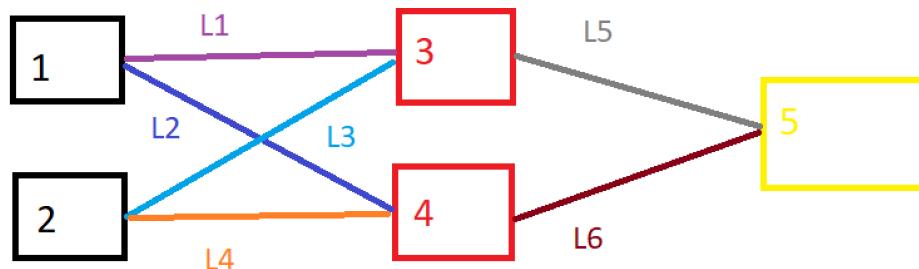
2.7 Réseau de neurones : Détection de chiffre

Pour faire notre réseau de neurones, j'ai crée un type sur c qui ce nomme Neuron. De cette manière nous pouvons avoir des fonctions de base pour notre réseau de neurones qui sont sûre.

Pour faire cette structure j'ai une liste d'entier naturel, il contient le nombre de neurones sur chaque couche plus les entrées et les sorties. Cette liste doit donc avoir une longueur minimum de trois, le premier élément est le nombre d'entrée de notre réseaux de neurones, les suivants sont les différentes couches et le dernier chiffre est le nombre de sortie.

Nous avons aussi dans cette structure la longueur de la liste en question.

Pour finir nous avons un pointeur qui pointe sur une mémoire qui est allouer statiquement, cette mémoire possède toutes les valeurs des neurones ainsi que la valeur de leurs liens. Pour ordonnée cette allocation j'ai décidé de mettre tous les neurones d'une couche puis leurs liens. Pour mieux comprendre, nous l'avons illustré avec une petite image.



```
liste={2,2,1} longueur de la liste = 3  
mémoire allouer = 1,2,L1,L2,L3,L4,3,4,L5,L6,5
```

Maintenant que nous avons notre réseaux de neurones, il faut pouvoir l'enre-

gistrer.

Pour ce faire rien de plus simple, on met dans un document le nombre d'élément dans la liste sur la première ligne, sur la deuxième ligne nous avons la liste où chaque éléments sont séparer par un espace.

Pour finir nous avons la valeur de chaque éléments dans la mémoire statique qui se retrouve les un sous les autres, ils sont juste séparé par des saut de lignes.

Pour arriver à entraîner un réseau, il faut faire des calculs entre les biais et les neurones, puis selon le résultat obtenu, on modifie les biais de sorte que le réseau donne un meilleur résultat la fois prochaine. Pour faire cela, nous avons le fichier propagation.c qui va contenir toutes les méthodes nécessaires pour entraîner un réseau.

Tout d'abord, nous avons la méthode propagation qui va, à partir des entrées, calculer la valeur de chaque noeud jusqu'au(x) dernier(s). Puis, avec la méthode retropropagation, on compare le résultat obtenu à partir du réseau avec le résultat attendu. On crée un delta égal à la valeur attendue à laquelle on soustrait la valeur obtenue. En répétant cette opération pondérée par les biais entre chaque neurone, on obtient un delta par neurone. Ensuite, on utilise ce delta pour modifier la valeur des biais en leur ajoutant delta pondéré par le taux d'apprentissage. Pour information, le taux d'apprentissage permet de contrôler la vitesse d'évolution des biais dans le réseau de neurones pendant l'apprentissage.

Après un grand nombre d'appels consécutifs de ces deux méthodes, la sortie du réseau de neurones donne un résultat de plus en plus approchant du résultat attendu.

2.8 Réseau de neurones (new) : Détection de chiffre

Dans la partie précédente, se trouve l'ancienne version de notre réseau de neurones. Cependant, le principe général du réseau de neurones reste le même. C'est pourquoi ne ne reviendront pas dessus en détail.

Nous avons décidé de changer pour une autre implémentation pour permettre d'ajouter plus de fonctionnalités. En effet, cette nouvelle implémentation est mieux structurée, ce qui permet d'avoir des programmes que l'on peut améliorer plus facilement. Par exemple, un réseau "Network" est composé de couches "Layer", qui elles mêmes sont composées de neurones "Neuron" ou de poids "Weight" pour les liens entre les neurones. De même, un "Neuron" comporte un champ pour stocker le biais "bias". Un autre avantage de cette nouvelle implémentation est son optimisation. Le "Neuron" est aussi composé de tableaux d'indices qui renvoient directement aux poids les reliant entre eux. Grâce à cela, tous les calculs pour obtenir les indices sont faits à la création du "Network", ce qui améliore grandement les performances.

Par exemple, on a pu constater que lors de l'entraînement du réseau, 1000 itérations se font en moins de 1 seconde ! C'est largement suffisant pour nous permettre de faire plusieurs essais lors du développement du projet, et également pour lancer un très grand nombre d'itérations d'apprentissage.

Après avoir entraîné le réseau, il est sauvegardé dans un fichier avec l'extension `.network` dans le dossier `Network/`. La fonction `load_network` permet

donc de charger le réseau déjà entraîné. Cela signifie donc que lors de l'exécution du projet, le programme charge le réseau déjà entraîné sur nos machines et obtient un résultat fiable très rapidement. Notre réseau est capable de reconnaître les chiffres dans les images découpées dans la grille importée depuis l'interface.

2.9 Résolution du sudoku

Pour la résolution d'un sudoku, comme nous avions déjà fait un TP sur les sudokus en S2, il suffirait de reprendre le code pour le modifier afin qu'il fasse simplement la résolution du sudoku. Voici une petite explication du code :

On part donc d'un tableau à double entrée qui est rempli par des chiffres de 1 à 9 (des cases déjà remplies) et de 0 (case pas encore remplie). Donc dans notre algorithme, on va partir de la première case et chercher la prochaine case contenant un 0 afin de la remplir.

5	3	1	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			

Puis on rempli cette case par un premier chiffre 1 si cela est possible donc si le placement de ce chiffre respecte les règles du sudoku. Sinon on passe au 2, puis au 3 sinon, ... Jusqu'à 9. On passe enfin sur la prochaine case qui n'est pas remplie et ainsi de suite.... Enfin, si une case n'a aucune valeur qui peut y être insérée, alors on va incrémenter de 1 (max 9) la case vide remplie avant celle-ci.

5	3	1	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			

5	3	1	1	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			

5	3	1	2	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			

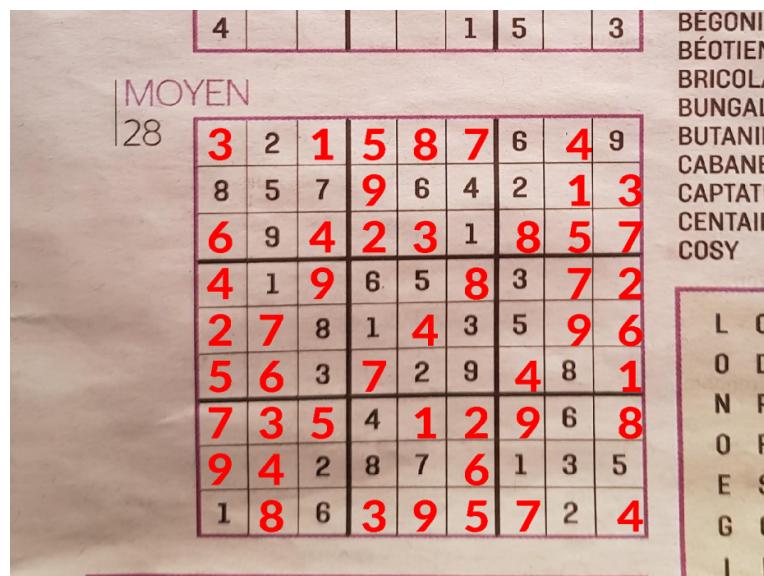
Enfin, dès qu'il n'y a plus de case vide, cela veut dire que la grille est complété et donc résolu !

2.10 Affichage du résultat sur l'image

Pour faire l'affichage du résultat de la grille sur l'image de base, nous avions d'abord eu l'idée de récupérer les coordonnées de chaque cases grâce à la détection de la grille. Mais comme celle ci nous donne les informations des 4 coins de la grille, alors nous allons simplement faire en sorte que un texte puisse être écrit dans cette zone. Après quelque recherche, il se trouve que la librairie `SDL_ttf` contient une fonction qui permet d'écrire des caractères à partir d'une police sauvegardé sous forme d'un fichier trouvable sur internet.

Les chiffres seront donc écrit un par un lors du parcours du résultat de la résolution de notre grille. Nous divisons en plus la grille en 9 puis nous donnons les coordonnées, la longueur et la largeur de chaque case de la grille à la fonction d'écriture de `SDL`.

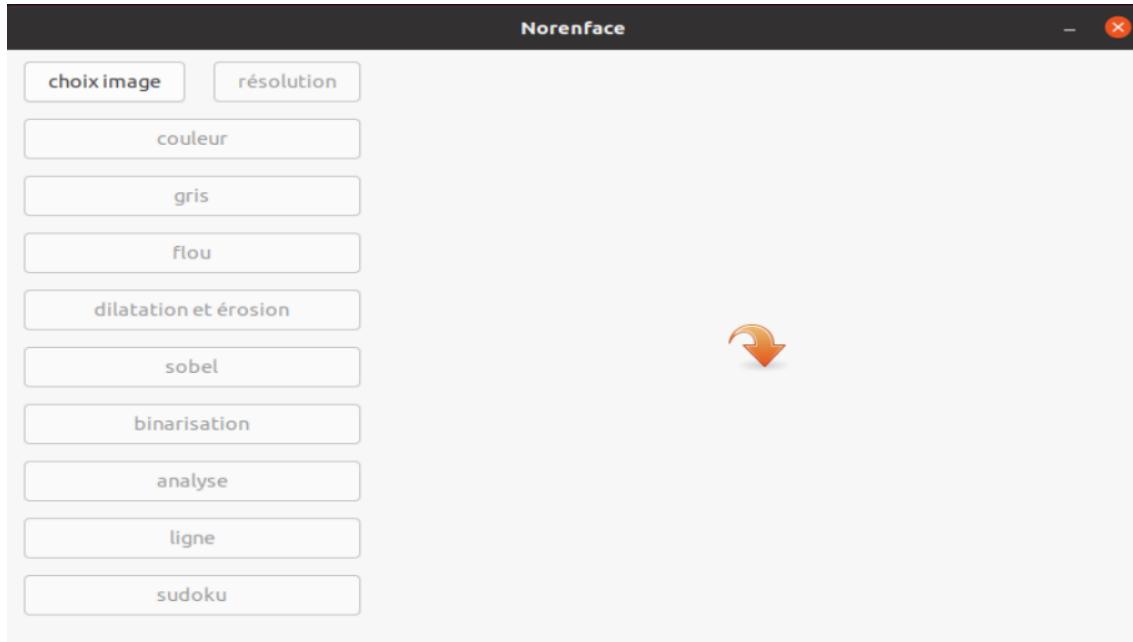
En plus de cela, il ne fallait pas que nous affichions les chiffres déjà présent sur la grille. Pour cela, nous faisons une copie profonde de notre grille de base, nous la résolvons et nous regardons sur la grille de base les chiffres qui sont vide, donc ceux représenté par des 0. Donc avant d'afficher le chiffre de la grille résolu, nous regardons si dans la grille, celui-ci était rempli ou non. Voici un exemple du résultat que cela nous propose :



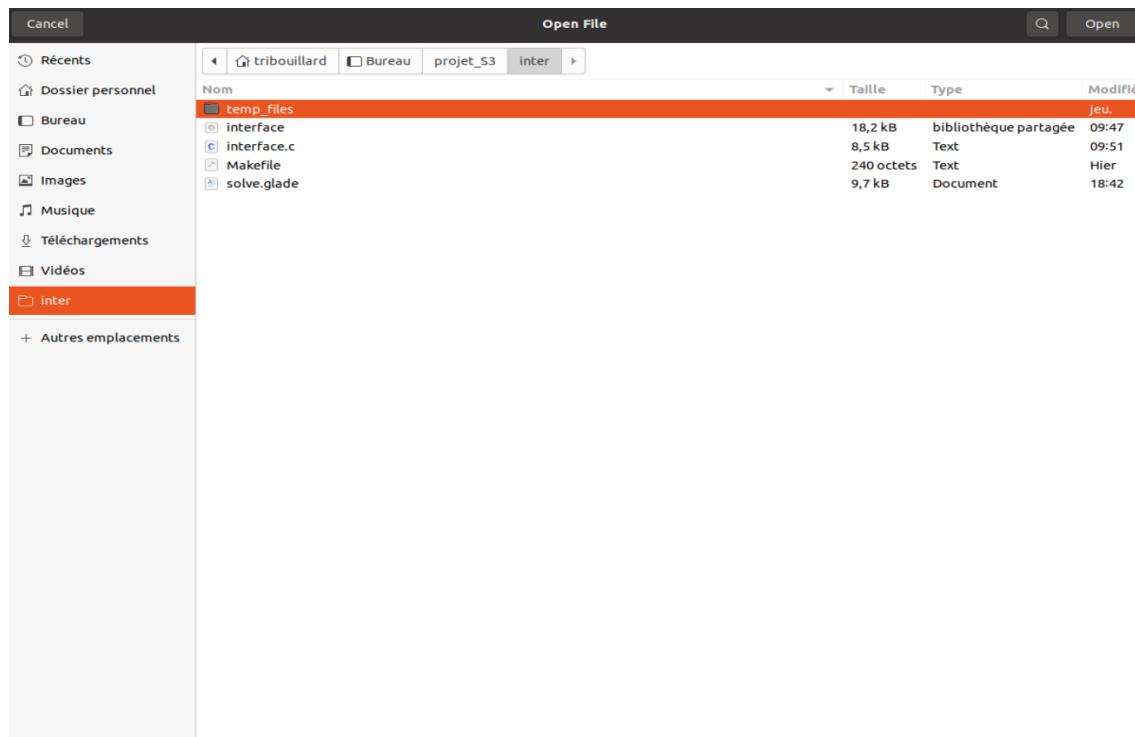
Comme vous le voyez, la grille est bien rempli, certain chiffre sorte un peu des cases mais c'est tout a fait compréhensible, et les chiffres sont d'une couleur différente que celle de base des chiffres pour avoir une meilleur lisibilité.

2.11 Interface graphique

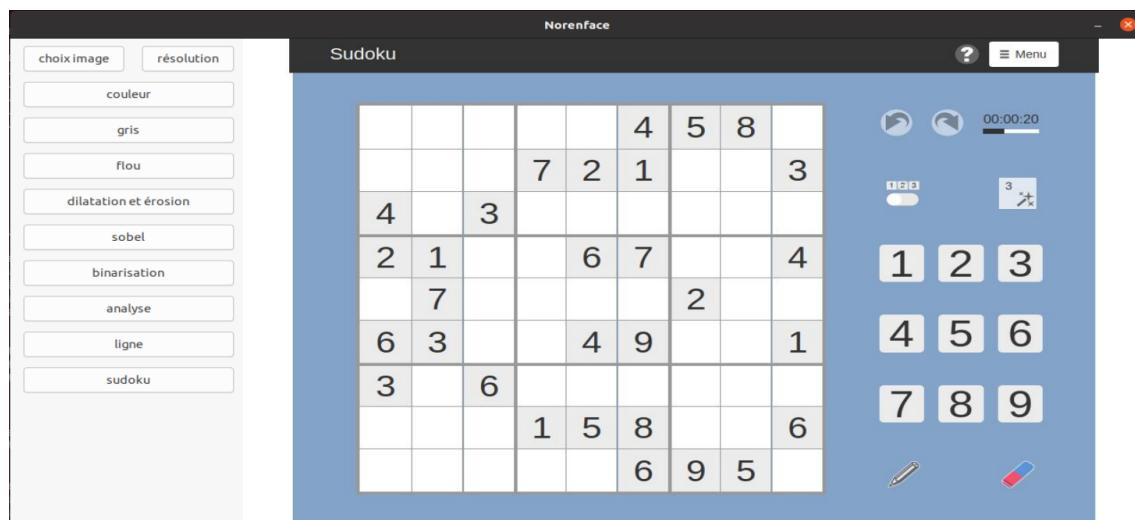
L'interface graphique est un élément qui permet à toutes les personnes qui le souhait de pouvoir utiliser notre programme de résolution de sudoku facilement. Il a été implémenté avec GTK.



Sur cette interface, il y a 11 boutons dont 10 qui sont désactivés car inutiles pour le moment, Nous pouvons aussi voir une flèche aller vers le bas à droite, ce sera l'endroit où nos images sera affichée. Cet affichage est de taille fixe mais celui-ci évoluera en fonction de la taille de l'image affichée. Il faut donc cliquer sur le bouton "Choix l'image", une nouvelle fenêtre comme celle-ci apparaîtra.



Il ne vous reste plus qu'à choisir l'image de votre sudoku et de le valider. Une fois cette étape faite, les 10 autres boutons deviennent actives. Au final on obtient cette fenêtre, l'image à droite est l'image que vous avez sélectionné.



Le fonctionnements des boutons sont simples et ne change pas trop, Il affiche une image qui est enregistrée dans un fichier temporaire. Seul le bouton "Choix de l'image" est un peut plus dure à implémenter car celui-ci doit appeler la fenêtre ci-dessus.

Je vais préciser l'utilité des différents boutons :

- Résolution sert à afficher l'image avec la résolution du sudoku.
- Couleur sert à afficher l'image sans modification.
- Gris sert à afficher l'image mais en gris.
- Flou sert à afficher l'image en flou.
- Dilatation et érosion permet d'afficher l'image après une dilatation et une érosion.
- Sobel permet de nous montré une image qui fut filtré par le filtre de Sobel.
- Binarisation permet de montré une image en noir ou blanc.
- Analyse montre l'image qui permet la détection des lignes.
- Détection permet de montré les lignes qui sont détectés par le programme, ils seront colorier en rouge.
- Sudoku sert à afficher la grille qui est reconnue par le programme.

Bien sûr tous ces boutons ne sont pas tous utile pour une personne qui veut juste avoir sa solution, mais dans l'équipe Norenface nous avons pensé que si nous montrions les étapes de notre programme cela ne pouvait être que plus bénéfique sans pour autant gêner l'utilisation.

3 Conclusion

Pour ce projet, nous avons eu un très bon travail de groupe, nous nous sommes très bien répartis les tâches, chacun était à l'écoute des besoins des autres. De plus, nous avons tous eu une bonne idée de comment réaliser ce projet. Nous avions tous le même point de vue ce qui nous a motivé à continuer de manière dynamique ce projet pour réussir à le faire devenir un super outil de solveur de sudoku. Nous aspirions tous à cette idée et nous faisions tout ce qui était possible pour faire au mieux lors de nos différents travaux sur ce projet.

Grâce à ce projet, nous avons appris un grand nombre de notions comme le traitement d'image, les calculs possibles dessus, de nombreux algorithmes et le plus important, comment faire un réseau de neurones qui est une notion très importante dans notre société actuelle.

C'était une très bonne expériences à faire en groupe.