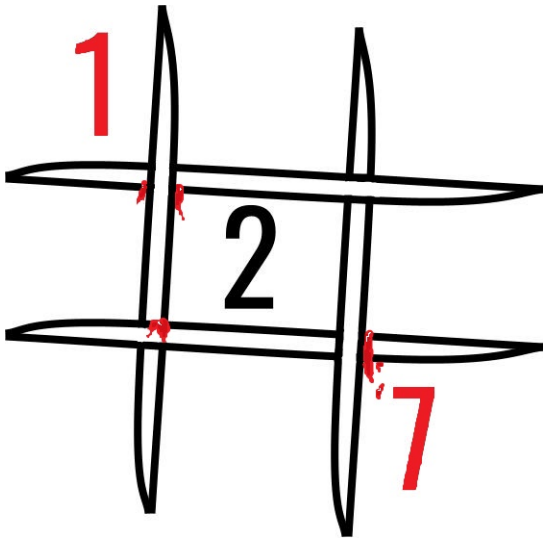
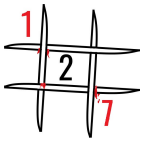
Rapport OCR

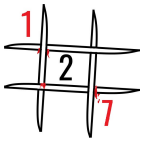


Sudoku Slayers

Mars 2021



Sommaire

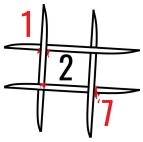


1 Introduction

Le but de ce projet est de créer un solveur de sudoku par OCR (Optical Character Recognition). C'est à dire, un programme qui prends en entrée une image de sudoku et ressort une nouvelle image du sudoku résolut. Pour ce faire, il faut passer par plusieurs étapes :

Tâches	assignée à	supervisée par
Chargement d'une image et suppression des couleurs	Mathieu Gr	Mathieu C
Détection de la grille et de la position des cases	Mathieu Gr	X
Découpage de l'image pour chaques cases	Mathieu C	Mathieu Gr
Implémentation de l'algorithme de résolution d'un sudoku	Mathieu C	Emeric
Une preuve du concept de notre réseau de neurones	Emeric	X
Sauvegarde et chargement des poids du réseau de neurones	Emeric	X
Jeu d'images pour l'apprentissage	Petre	Emeric
Sauvegarde des résultats	Mathieu C	X

Dans ce rapport, nous allons vous expliquer chacune de ces étapes et ce que nous avons depuis le lancement du projet.



2 Avancées

2.1 Emeric - Réseau de neuronnes

Pour ce projet, j'ai décidé de travailler sur le réseau de neurones. J'ai tous d'abord cherché le fonctionnement d'un réseau de neurone, et voici ce que j'ai trouvé.

2.1.1 Structure

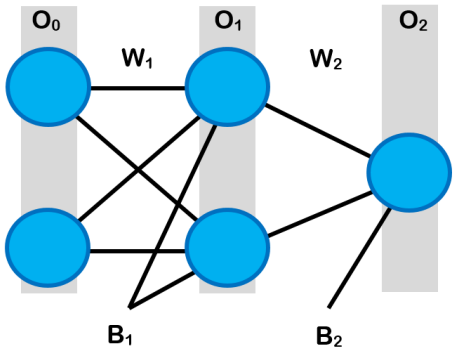
Un réseau de neuronnes est composé de couches et de neurones. Ces couches sont assimilées à une liste de neuronnes et se séparent en 3 catégories :

- Entrée : correspondant aux données que l'on transmet au réseau.
- Couches Cachées : correspondant aux étapes permettant de nous approcher d'un résultat.
- Sortie : correspondant aux résultats possibles donnés par le réseau en sortie.

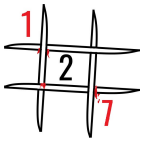
Les neuronnes, quand à eux, possèdent 4 éléments :

- Valeur : litéralement la valeur associé au neuronne.
- Valeur d'activation : lorsque la valeur du neuronne est supérieure à un certain seuil, elle devient la valeur d'activation.
- Liste de poids : les poids correspondent à l'influence de ce neuronne sur les neuronnes de la prochaine couche.
- Biais : permet de différencier les valeurs de chaque neurone de la couche suivante.

Voici un schéma permettant de visualiser le réseau de neurone. Ici, on peut voir 3 couches (O_0,O_1,O_2) possédant des neurones (en bleu), qui sont reliés entre les couches par des poids (W). On peut voir que les valeurs de chaque neurone de la couche suivante dépendent de la valeur de ceux de la couche précédente, de leurs poids et du biais (B).



Nous avons donc vu la structure d'un réseau de neurone, il est temps de s'intéresser au parcours de ce dernier.



2.1.2 Parcours

Pour que le réseau nous donne un résultat, il faut que les neurones des couches cachées et de la sortie soit actualiser. C'est la propagation. Dans cette propagation, chaque valeur des neurones de la couche suivante est égale à :

$$v = \sum (w_i * va_i) + b \tag{1}$$

Avec w_i les poids, v valeur, va_i les valeurs d'activation et b les biais.

Ainsi, les poids des neurones influent sur la valeur des neurone de la couche suivante et sur le résultat. Pour le couche de sortie on va venir appliqué une fonction d'activation pour avoir une sortie égale à 0 ou 1. Voici la fonction Sigmoïd :

$$S(x) = 1/(1 + exp(-x)) \tag{2}$$

Cependant, le parcours d'un réseau de neurone doit s'effectuer dans les deux sens. En effet, la première fois que l'on appelle le réseau de neurone, le résultat en sortie a de grandes chances d'être mauvais. L'explication est très simple, le réseau n'a pas appris à donner le bon résultat. Ainsi, pour que le réseau sache donner le bon résultat, on doit le remonter et changer tous les poids et biais. C'est ce qu'on appelle la rétropropagation. Pour effectuer cette rétropropagtion, on utilise les equations suivantes :

$$\text{delta}_{\text{output layer}} = dz^i = (\text{actv}^i(x) - y(x)) * S'(z)$$

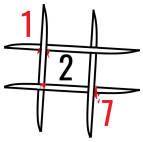
$$\text{delta}_{\text{hidden layer}} = dz^i = (\text{out\_weights}^{i+1} * dz^{i+1}) * R'(z)$$

$$dw^i_{jk} = dz_j^i * \text{actv}_k^{i-1}$$

$$dbias^i = dz_j^i$$

Ici, dz^i est la nouvelle valeur du neurone de sortie qui va permettre de remonter le réseau et changer les poids et les biais. La fonction R' est une fonction qui met la valeur a zero si elle est inférieur à un certain seuil.

Pour entrainer le réseau de neurone, on va répéter le parcours de ce dernier jusqu'à ce que les poids et les biais permette d'avoirla bonne réponse en sortie à chaque fois.

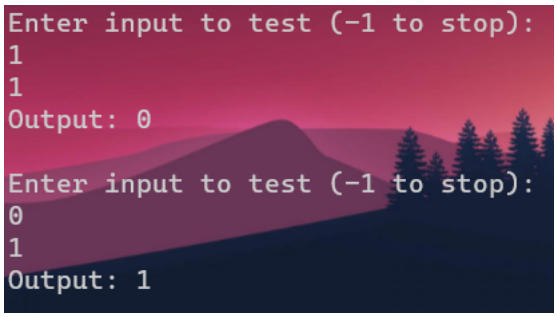


2.1.3 Pratique

Maintenans que nous avons vu la théorie, je vais vous expliquer ce que j'ai mis en pratique.

Pour cette première soutenance, j'ai créer un réseau de neurones que applique la fonction XOR. Pour cela, j'ai créer deux struct, neuron et layer, ausquels j'ai donné chaque composant que l'on a vu dans la partie Structure. Ensuite, j'ai créer une liste de 3 layer avec respectivement 2, 2 et 1 neurones et j'ai également initialisé chaque poids et biais par un nombre aléatoire entre 0 et 1. Pour l'entrainer, j'ai parcourus le réseau 20000 fois pour les 4 entrées suivante : (0,0), (0,1), (1,0) et (1,1). J'ai lancé la rétropropagation à chaque que les sorties ne correspondaient pas aux sorties suivante pour les entrées respectives : 0, 1, 1 et 0.

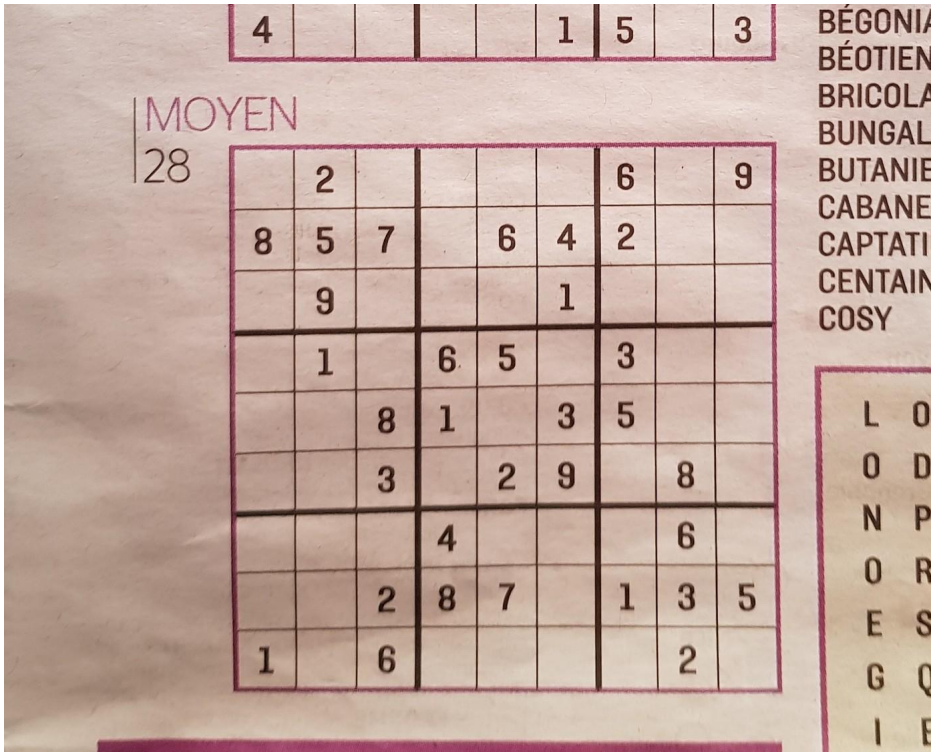
Après s'être entrainé, je demande à l'utilisateur les entrées et le réseau donne le bon résultat en sortie. Voici ce que l'on voit dans le shell :



2.2 Mathieu Gras - Traitement de l'image

De mon coté je me suis occupé du traitement de l'image afin de faciliter la reconnaissance des caractères dans le sudoku malgré que ce processus ne soit pas fini il y a des concepts intéressants que j'ai découvert. Tout d'abord il y a la suppression des couleurs de l'image qui passe d'abord par la convertion de l'image en échelle de gris. C'est à dire modifier les couleurs de tous les pixels pour qu'ils soient tous gris mais plus ou moins sombre.

Image d'origine



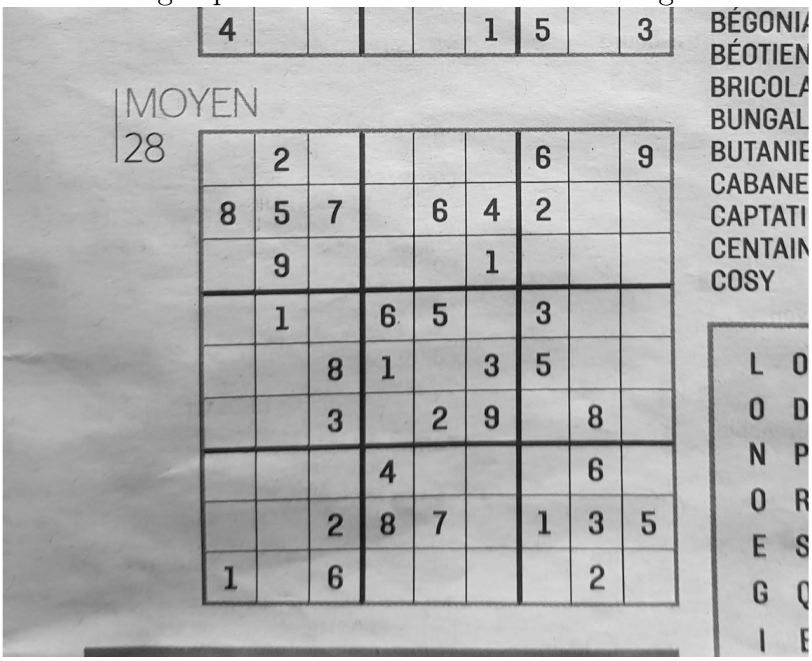
MOYEN
28

4					1	5		3
	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

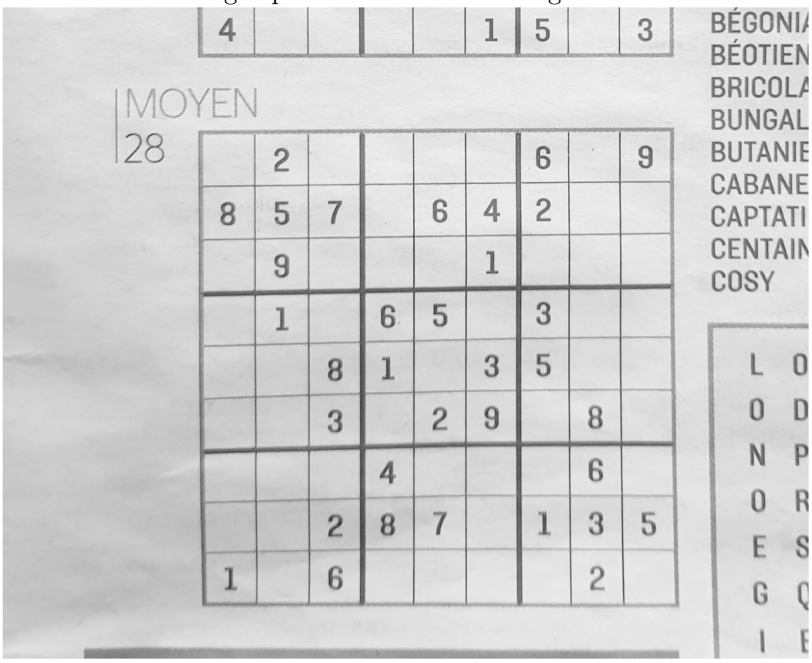
L O
O D
N P
O R
E S
G
I

Image après modification de l'échelle de gris



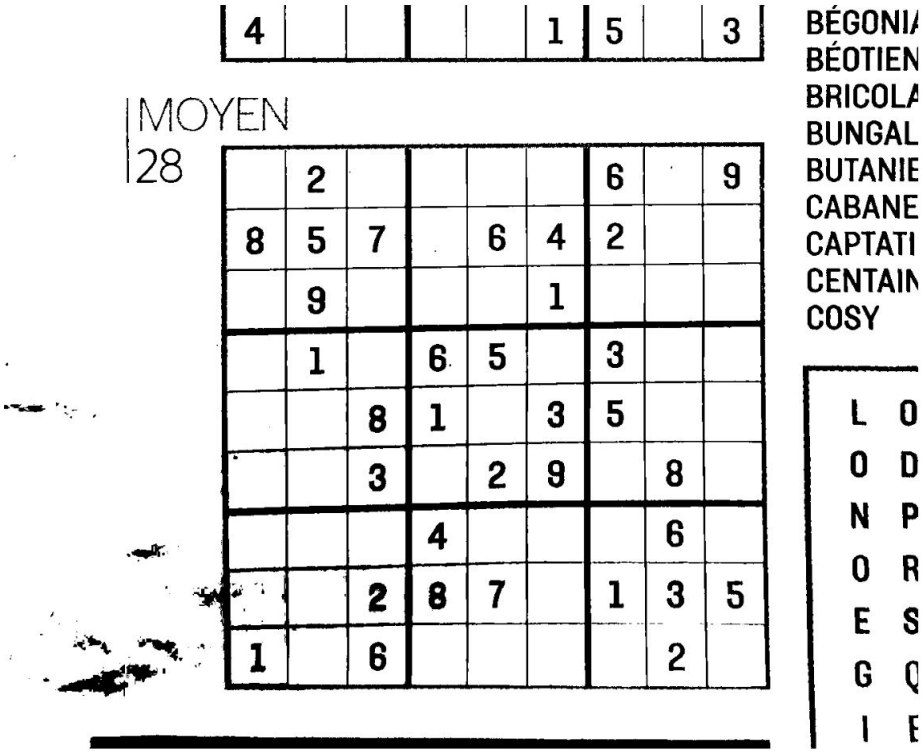
L'étape d'après est d'augmenter le gamma de l'image vers le blanc. Cette astuce permet à uniformiser tous les pixels de l'image vers le blanc pour éliminer un peu de bruit de l'image.

Image après la modification du gamma

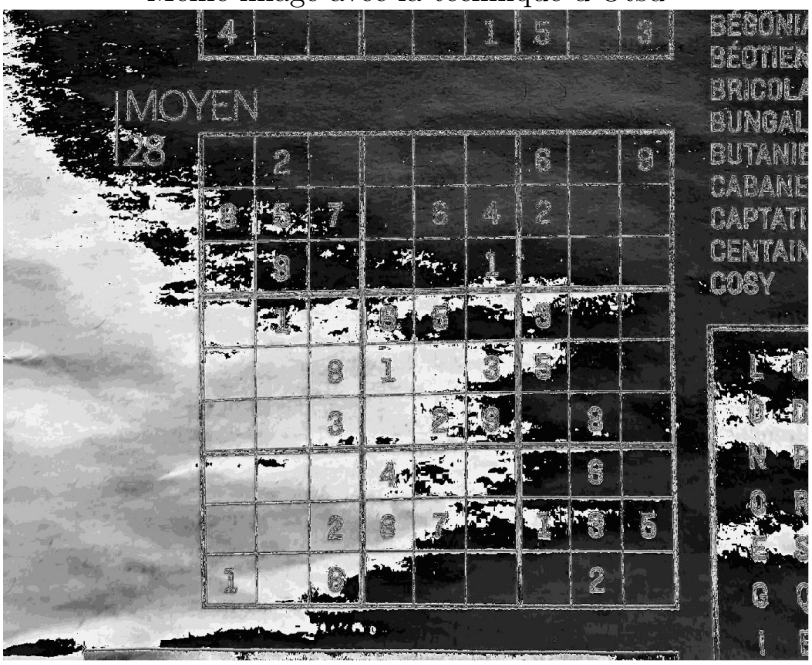


Ensuite vient la modification du contraste de l'image pour faire ressortir la grille et les numéros de chaques cases. Pour cela il y a plusieurs méthodes plus ou moins complexes. La méthode de binarisation la plus simple est de fixer un seuil à partir duquel les pixels seront considérés comme noir ou blanc. Or il existe des méthodes pour déterminer automatiquement le seuil pour chaques images celle avec laquelle j'ai expérimenté le plus est la méthode d'Otsu.

Image après l'augmentation du contraste avec la méthode basique



Même image avec la technique d'Otsu



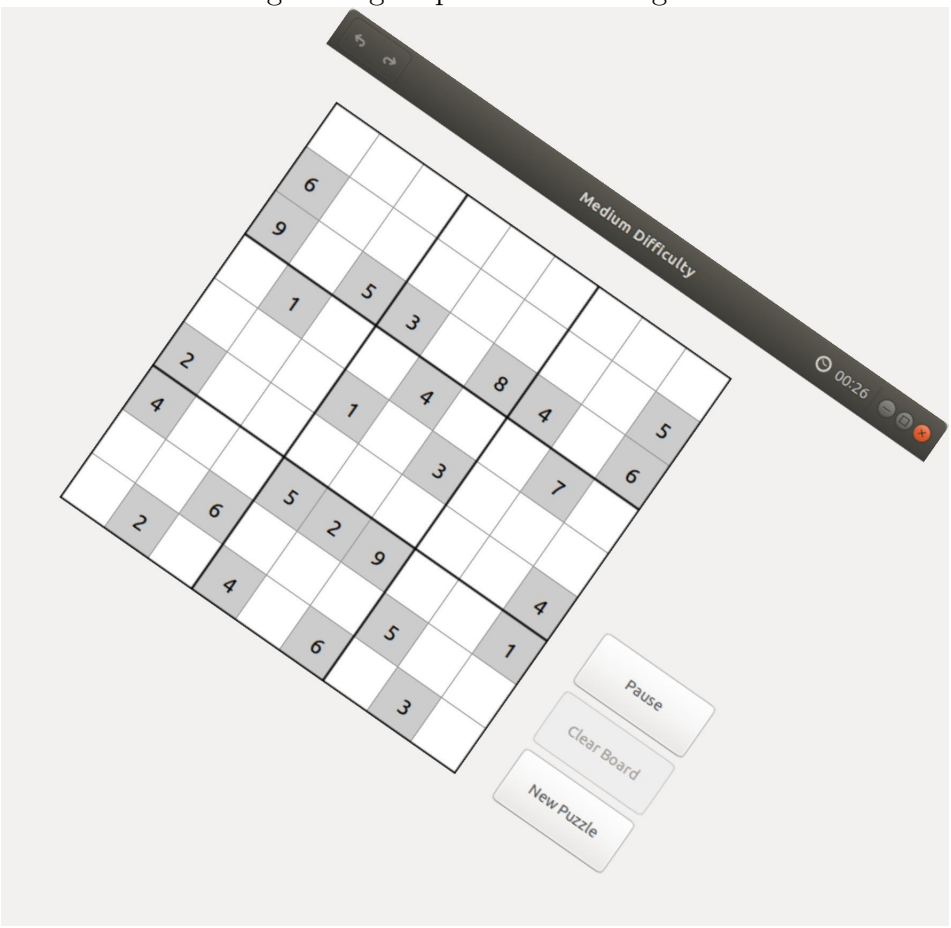
A noter que notre fonction otsu a remplacé les pixels blancs par des noirs en vu de futurs tests sur l'image mais elle ne sera pas utilisée ici car elle rajoute du bruit à l'image d'où le fait qu'on utilise la méthode plus "basique" pour l'instant du moins. Concernant la détection de la grille je n'ai malheuresement pas réussi à mettre en place l'algorithme cela vient en partie du fait qu'on était que 3 sur ce projet donc on ne pouvait pas être plusieurs à plein temps sur une partie même si on s'est quand même entraidés.

Pour la détection de la grille je comptais utiliser un algorithme de détection de ligne qui renvoie des struct avec deux attributs : l'angle de la ligne et sa longueur. Avec ces deux attributs j'aurais pu déterminer l'angle de rotation de l'image pour que la grille soit droite et soit reconnaissable par l'ocr, la distance aurait servie pour la détection de la grille en elle même car elle est composée de 10 lignes et 10 colonnes de longueurs et angles égaux (excépté pour les colonnes mais elles sont perpendiculaires aux lignes donc facilement identifiables).

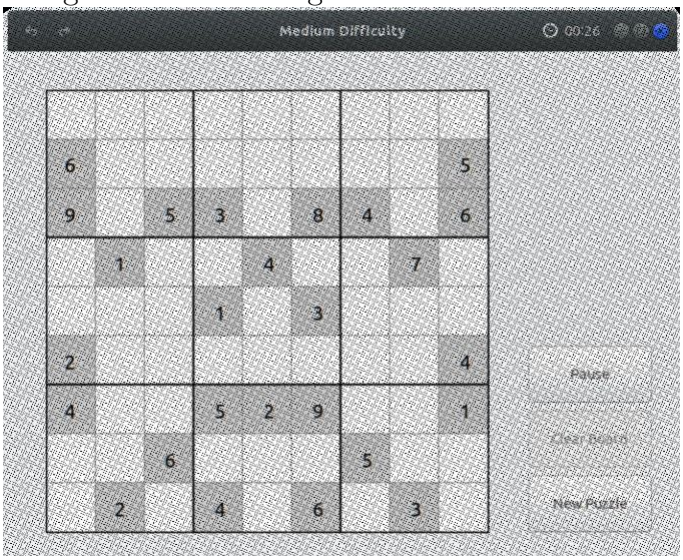
Cependant j'ai déjà fait des fonctions qui me permettront une fois la position de la grille identifiée de terminer le traitement de l'image.

Tout d'abord la rotation de l'image (pour l'instant uniquement donnée par l'utilisateur) :

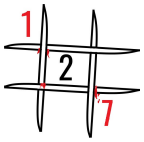
Image d'origine penchée à 30 degrés



Même image rotatée à 30 degrés avec ma fonction de rotation



Ici des paternes apparaissent qui ont l'air aléatoires mais qui sont due au fait que pour supporter des angles différents de 90,180 et 270 il a fallu utiliser des fonctions trigonométriques donc il y a des arrondis inévitables qui semblent être la cause de l'appartion de ces paternes néanmoins comme ils sont discrets ils ne devraient pas géner la reconnaissance des caractères.



Ensuite j'ai fait une fonction qui avec les coordonnés de la grille permet d'avoir une image centrée sur la grille mais elle ne sera opérationnelle seulement quand j'aurais accès aux coordonnées de la grille.

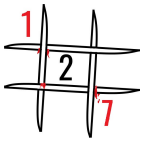
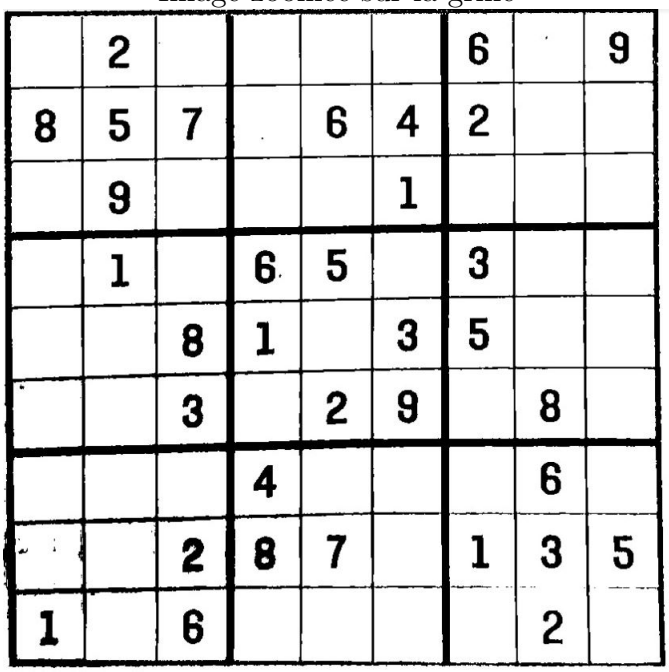


Image zoomée sur la grille



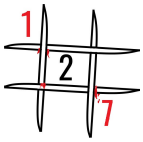
2.3 Mathieu Campan - Solver et sauvegardes

2.3.1 Découpage de l'image

A partir de la détection de la grille, Mathieu Gras effectue un Zoom (fonction qu'il a d'ailleurs rédigé) qui prend donc en considération uniquement la grille. Celle-ci se verra découpée en 81 parties égales contenant chacune soit une case vide, soit une case composé d'un chiffre différent de 0. Elles seront enregistrés en format .bmp dans un dossier img/. Pour celà, on a eu recours à une fonction SDL-save implémentée dans une bibliothèque. Chaque case se verra attribuer un nom tel que "cell00.bmp" pour la case tout en haut à gauche. J'ai d'ailleurs utiliser la fonction strcat pour la création de path dans cette partie et aussi dans la sauvegarde du fichier du sudoku résolu.

2.3.2 Résolution du sudoku

Pour la partie algorithmique qui sert à résoudre notre sudoku, de multiples fonctions rédigées dans le même dossier ont été utilisées,



ainsi que de simples bibliothèques donnant accès aux fonctions printf et strcat.

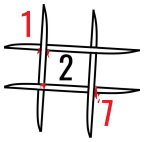
Tout d'abord, à partir du fichier contenant les valeurs connues du sudoku, notre algorithme va initialiser deux listes : une contenant les probabilités de valeurs qu'une case puisse contenir sous la forme d'un long non signé (une seule probabilité s'y trouve si la valeur est connue). La seconde, nommée new-known-values en abrégé, contient la position des valeurs connues dans la première liste, pour nous créer un ordre de résolution. A chaque nouvelle valeur connue, une série de fonctions s'effectuent.

En effet, l'algorithme passe un très grand nombre de fois par la fonction updatesudoku qui a pour but d'appeler elle même 3 fonctions : update-row, update-col, update-block. Ces dernières permettent d'effectuer la fonction del-proba sur chaque cases concernée par la nouvelle case dont la valeur est désormais connue.

Ainsi, les fonctions delproba et deleteproba permettent de réduire le nombre de possibilités sur une case donnée du sudoku en fonction d'une valeur donnée. Si cela permet d'obtenir une nouvelle valeur définitive, alors celle-ci est directement ajoutée aux new-know-values.

2.3.3 Sauvegarde des résultats

Lors de l'utilisation de l'algorithme de résolution, on s'assure d'ouvrir le fichier contenant les valeurs du sudoku, et d'accéder à un nouveau fichier dans lequel sauvegardé le résultat de notre résolution de sudoku grâce à la commande fopen. Comme demandé, la commande "./solver grid-00" qui nous crée ainsi un nouveau fichier nommé grid-00.result



2.4 Petre Finta - Préparation au training

J'ai commencé le travail assez tard et de ce fait la partie que je devait faire pour cette soutenance à été prise en charge par les autres membres du groupe alors j'ai commencé à préparer les images d'entrainement pour le réseau de neurones. Ce sont de simples images qui contiennent des chiffres de 1 à 9 en différentes polices d'écriture. Afin de dire au réseau de neurones à quoi ressemble chaque chiffre sous ses différentes formes écrites.

En voila quelques exemples :