



Rapport de soutenance 1

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

9 Novembre 2022

Table des matières

1	Introduction	2
1.1	Membres du groupe	2
2	Avancement des tâches	3
2.1	Répartition des tâches	3
2.2	Chargement d'une image	4
2.3	Rotation manuelle de l'image	4
2.4	Suppression des couleurs	6
2.5	Prétraitement	10
2.6	Détection de la grille	11
2.7	Découpage de l'image	13
2.8	Résolution du sudoku	14
2.9	Chargement et sauvegarde du sudoku	15
2.10	Réseau de neurones : OU EXCLUSIF	16
2.11	Réseau de neurones : calculs et entraînement	18
3	Conclusion	19

1 Introduction

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

L'ambition de ce projet est 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 Avancement des tâches

2.1 Répartition des tâches

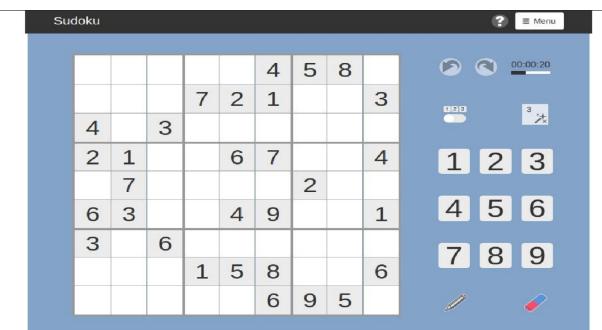
	Réponsable :		Suplémentaire :	
	Nathan	Rémi	Jean-Jules	Nicolas
Rotation de l'image				
Suppression des couleurs				
Prétraitement				
Détection de la grille				
Découpage de la grille				
Résolution d'un sudoku				
Chargement et sauvegarde du sudoku				
Réseau de neurones				
Jeu d'image				

2.2 Chargement d'une image

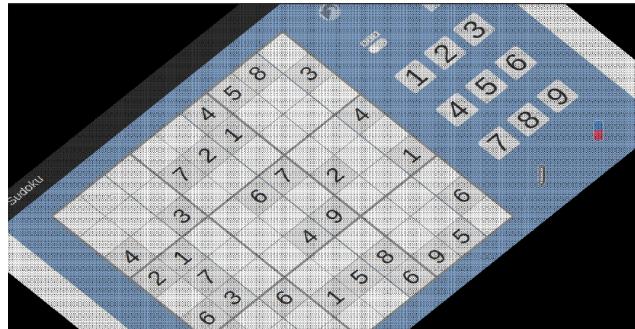
Pour le chargement de l'image, nous avons tout simplement repris un de nos TP sur SDL2 où il fallait ouvrir des images, ajuster la fenêtre... Ce code est donc dans la plupart de nos fichiers pour vous montrer nos résultats lors de la première soutenance. Ce code va donc marcher de la façon à ouvrir une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

2.3 Rotation manuelle de l'image

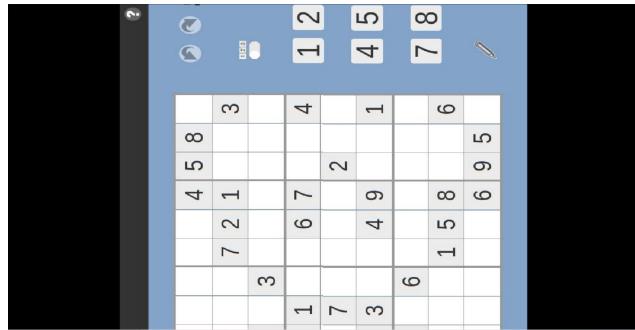
La rotation est une étape importante du processus de reconnaissance d'image en effet si nous ne pouvons retourné une image comment différentier un 6 d'un 9. De plus si nous ne pouvons faire des rotations manuelle alors nous pourrions recadrer l'image. Pour les rotations d'images nous avons choisi de 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 :



2.4 Suppression des couleurs

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.

```

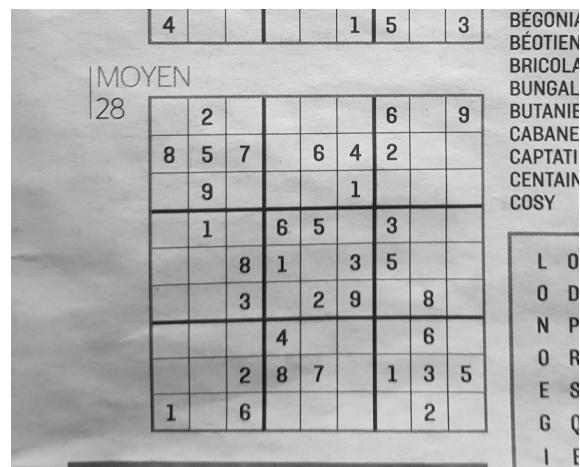
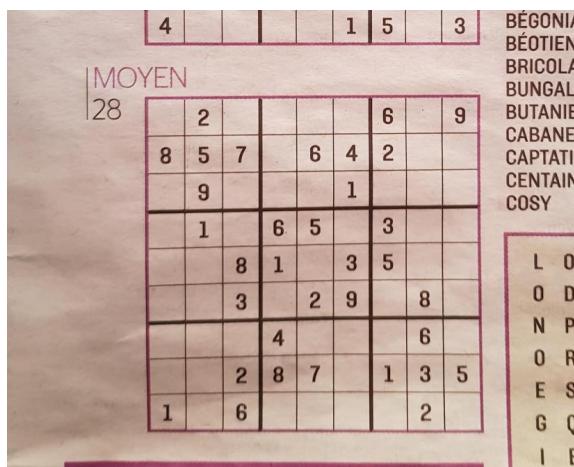
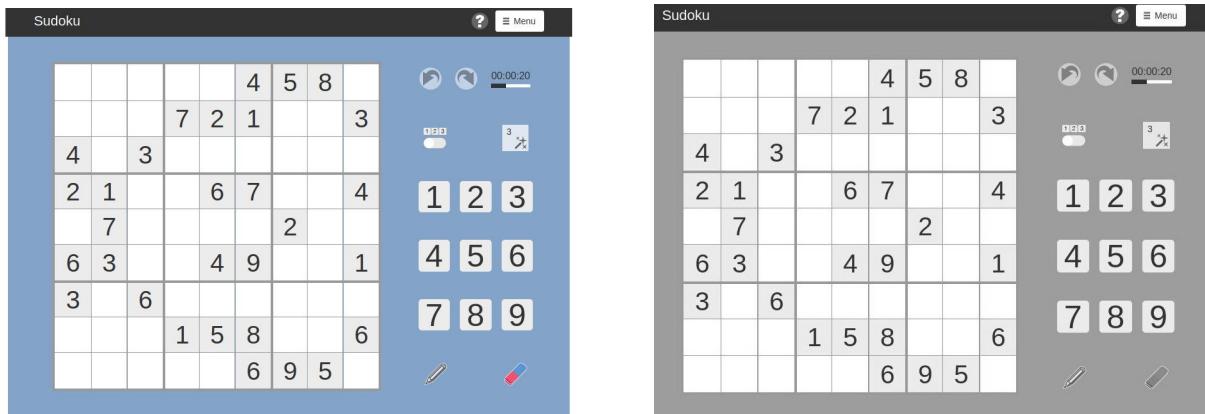
Uint32 pixel_to_grayscale(Uint32 pixel_color, SDL_PixelFormat* format)
{
    Uint8 r, g, b;
    SDL_GetRGB(pixel_color, format, &r, &g, &b);

    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 :



Binarization

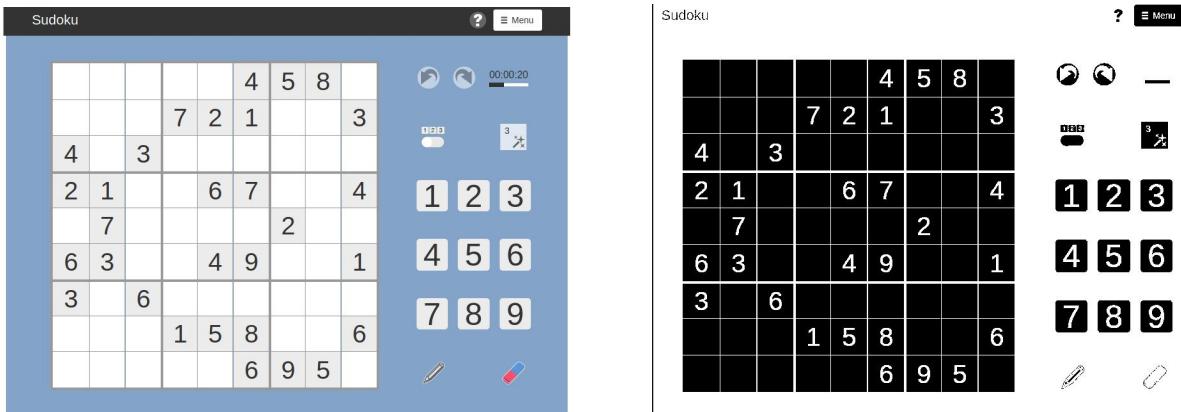
La binarization est la dernière et la plus importante des parties pour faire l'analyse de l'image. Binarizer une image signifie qu'elle va être transformée en pixel soit noir ou blanc. Les actions comme le flou permettent justement d'améliorer cette transformation pour avoir un très bon résultat pour l'analyse. Plusieurs méthodes sont possible 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. Cette méthode répertorie 3 façons de définir le seuil.

- 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.

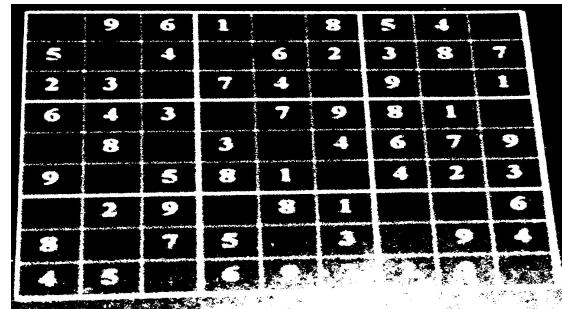
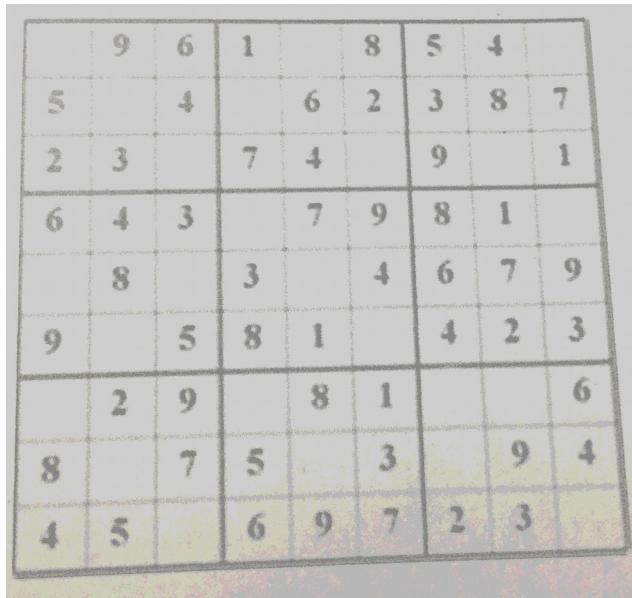
- Enfin, il y a le seuil local. Dans cette dernière méthode, on va changer de seuil pour chaque pixel en prenant la moyenne des pixels voisins dans un rayon défini (généralement 1 ou 2). Mais cette technique fait apparaître de nombreuses imperfection.

C'est pour cela que nous avons choisi de garder la méthode du seuil global d'Otsu pour la binarization !



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 binarization.

Après des heures à chercher comment améliorer cette partie, nous avons découvert d'autres moyens comme le filtre de Sobel ou encore la technique tu "closing" qui pouvait subvenir à nos besoins. Seulement par un manque de temps et une difficulté de recherche pour programmer ces 2 méthodes, nous ne pouvons pas vous présenter de meilleurs résultats.



2.5 Prétraitement

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.

De nombreux algorithmes permettent cela, mais nous allons plus nous intéresser à ce qui permet de réduire certaines imperfections comme des chiffres qui ne sont pas complet. La technique qui permet cela est le **flou gaussien** qui va flouter l'image avec quelque différence que pour un flou basique.

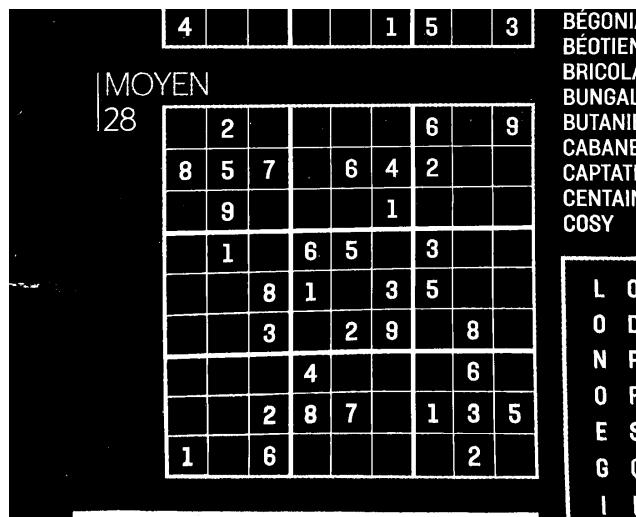
En effet, dans le flou basique, nous allons analyser les pixels un par un et faire la moyenne des valeurs de tous ses voisins afin d'obtenir un nouveau pixel. Mais ce flou ne corrige pas certaines parties de l'image.

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	1	2	1
145	145	12	12	42		2	4	2
145	145	54	156	156		1	2	1

2.6 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	

2.7 Découpage de l'image

Le découpage de l'image est la partie du projet où l'image, après avoir subi la binarisation, est rognée avec les coordonnées des cases obtenues dans la section précédente pour avoir en résultat les images de chaque case de Sudoku.



Pour ce faire, il suffit de sauvegarder les pixels de chaque case et de les transformer d'abord en une surface, puis en une image au format png.

2.8 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	0	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.

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

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	

2.9 Chargement et sauvegarde du sudoku

Pour récupérer la grille de sudoku à résoudre, on part d'un fichier contenant les caractères la composant. On a donc des chiffres de 1 à 9 pour les cases rempli, et les cases vides sont représenté par des '.'. On a donc des fichiers comme celui-ci :

... .4 58.
... 721 ..3
4.3

21. .67 ..4
.7. ... 2..
63. .49 ..1

3.6
... 158 ..6
... ..6 95.

Dans le code, on va donc convertir tous ces caractères en tableau à deux entrées. Pour cela, on lit les caractères un par un. Si on tombe sur un '.', alors on place un 0 dans le tableau, sinon on place le chiffre lu. Dès qu'on arrive à la fin d'une ligne, on incrémente la valeur de notre 1ère dimension du tableau. Puis on rempli le tableau ainsi de suite.

Et enfin pour créer un fichier, on va nommer ce fichier du même nom que le fichier avec la grille à résoudre et lui ajouter le suffixe ".result". On va remplir ce fichier à l'aide du tableau à deux entrées que nous avons complété. On va parcourir le tableau, ajouter les chiffres un par un, ajouter des espaces et des retour à la ligne là où il faut pour une meilleure lecture du fichier. Cela va donc nous donner des résultats de ce genre :

```
$ cat grid_00.result
127 634 589
589 721 643
463 985 127

218 567 394
974 813 265
635 249 871

356 492 718
792 158 436
841 376 952
```

Grâce à cela, la partie sur le sudoku solver est complété. Il ne nous reste plus qu'à afficher le résultat contenu dans ces fichiers !

2.10 Réseau de neurones : OU EXCLUSIF

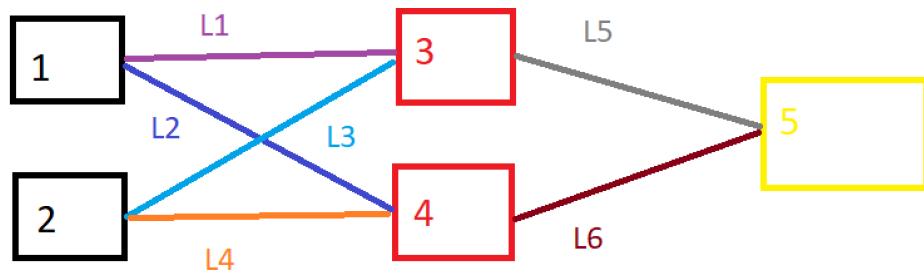
Pour faire notre réseau de neurones, j'ai crée un type sur c qui ce nomme Neuron. De cette manière nous pouvons avoir des fonctions de base pour notre réseau de neurones qui sont sûre.

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'enregistrer.

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.

2.11 Réseau de neurones : calculs et entraînement

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.

3 Conclusion

Actuellement, nous avons un très bon travail de groupe, nous nous répartissons très bien les tâches, chacun est à l'écoute des besoins des autres. De plus, nous avons tous une bonne idée de comment réaliser ce projet. Cependant, il y a des points à améliorer comme la binarization et le prétraitement qui n'est clairement pas parfait pour une analyse parfaite de l'image. Malgré cela, nous avons tout de même de bon point ce qui nous motive à continuer de manière dynamique ce projet pour réussir à le faire devenir un super résolveur de sudoku. Nous aspirons tous à cette idée et nous faisons tout ce qui est possible pour faire au mieux lors de nos différents travaux sur ce projet.

Notre travail en équipe est encore une fois très bien organisé avec une répartition qui va à tout le monde. Cela va continuer ainsi ne peut que finir sur cette lancée.