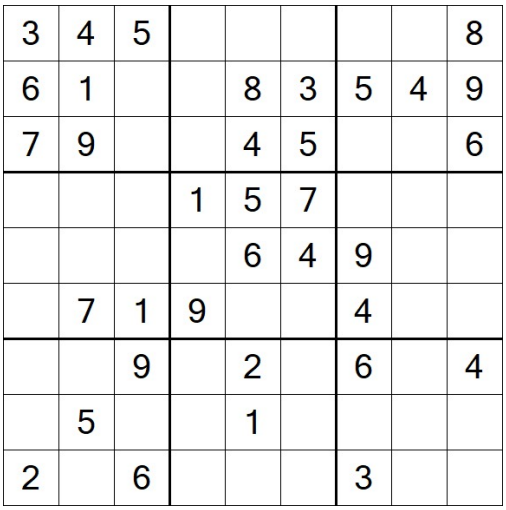
Rapport de soutenance intermédiaire
OCR Sudoku Solver



Au pneu AI

Médéric LAVIROTTE
Ivan HUARD
Sacha WIZEN

Contents

1 Introduction

1.1 Le projet

Pour ce semestre à Epita, nous avons formé une équipe nommée Au pneu AI composé de trois codeurs dévoués, pour réaliser le projet OCR Sudoku Solver.

L'objectif de ce projet est de réaliser un logiciel de type OCR (Optical Character Recognition) qui résout une grille de sudoku. Notre application prendra donc en entrée une image représentant une grille de sudoku et affichera en sortie la grille résolue. Dans sa version définitive, notre application devra proposer une interface graphique permettant de charger une image dans un format standard, de la visualiser, de corriger certains de ses défauts, et enfin d'afficher la grille complètement remplie et résolue. Nous allons mettre en place un aspect d'apprentissage qui permettra d'entraîner notre réseau de neurones, puis de sauvegarder et de recharger le résultat de cet apprentissage. Ce document est le fruit de nos travaux et aura pour but de vous présenter l'étendue du projet créé.
Bonne lecture.

1.2 L'équipe

Médéric LAVIROTTE :
Je m'appelle Médéric LAVIROTTE, j'ai 19 ans, et je suis passionné par l'informatique. Ma vie a changé depuis que j'ai découvert Vim. Coder sur Vim devient tout un coup une experience spirituelle, c'est sublime. J'adore également le C; je trouve que c'est un tres beau langage, notamment grâce au contrôle total sur la gestion de la mémoire. Le projet d'OCR est très intéressant notamment grace à sa diversité. En effet on étudie beaucoup de sujets variés comme le réseau de neurones, la résolution d'un sudoku ou encore les traitements d'images. Je suis donc très motivé et je donnerais mon maximum pour la réussite de ce projet.

Ivan HUARD :
C'est moi qui ai choisi le nom de l'équipe et je ne comprends vraiment pas pourquoi personne ne l'aime. Je trouve ça pourtant très original et drôle. Sinon je suis motivé pour ce projet.

Sacha WIZEN :
Je suis nul en math. Voila comme ca c'est dit. Et on m'a quand même donné la partie ou il y a le plus de maths : la rotation et la détection de la grille. Ca c'est vraiment pas malin. Je commence déjà à avoir mal à la tête rien qu'en lisant la documentation sur wikipedia. Ça va être long. Et sinon, je ne comprends toujours pas le jeu de mots sur le nom de l'équipe. Enfin je suis comme d'habitude très motivé étant donné que coder c'est la vie.

1.3 La répartition des charges

	Mederic	Ivan	Sacha
Prétraitement		***	*
Rotation			***
Reconnaissance de la grille			***
Découpage de la grille	***		
Réseau de neurones	*	***	
Resolution du sudoku	***		

Le degré d'implication : *

2 Prétraitement

2.1 Filtre Sobel

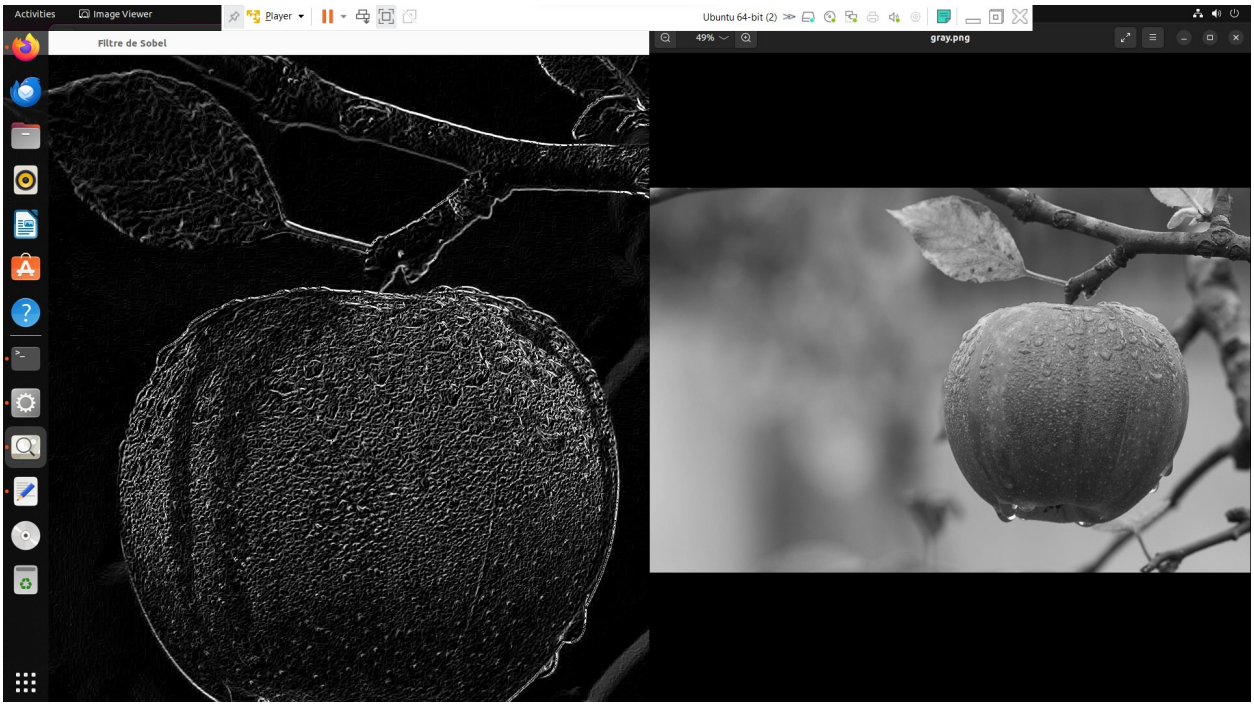
Le principe du filtre de Sobel est le suivant : détecter les bords des éléments dans une image en analysant les variations d'intensité des pixels. Afin de simplifier sa tâche, on converti d'abord l'image en niveaux de gris, en utilisant le code développé (grayscale) lors d'un TP. On applique un filtre sur chaque pixeld de l'image.

```
Uint8 r, g, b;
SDL_GetRGB(pixel_color, format, &r, &g, &b);
float average = 0.3*r + 0.59*g + 0.11*b;
Uint32 color = SDL_MapRGB(format, average, average, average);
return color;
```

Cette conversion en gris permet de se débarrasser des trois canaux de couleurs et de travailler directement avec l'intensité de l'image. Ensuite, on a créé une nouvelle "SDL_surface" (une nouvelle image) en recréant chaque pixel en fonction de ses pixels environnants. Pour déterminer s'il y avait une variation d'intensité, on compare le pixel en question avec les pixels voisins. Pour ce faire, il a utilisé des noyaux, qui sont en réalité des matrices pré-remplies.

```
int sobel_kernel_x[3][3] = {
        { -1, 0, 1 },
        { -2, 0, 2 },
        { -1, 0, 1 }
};

int sobel_kernel_y[3][3] = {
        { -1, -2, -1 },
        { 0, 0, 0 },
        { 1, 2, 1 }
};
```

Ces noyaux ont aidé à détecter les changements d'intensité dans un bloc de l'image. Si la somme des pixels environnants dépassait un certain seuil, cela signifiait qu'il y avait un changement d'intensité et donc un contour, et la fonction "dessine" le pixel en blanc. Sinon, on laisse en noir. Le resultat de la transformation de l'image en gris par la fonction Sobel :



2.2 Rotation

Afin d'implémenter une rotation qui satisfait tous nos critères, nous sommes passés par plusieurs étapes, la première était d'effectuer des rotations simples 90 – 180 – 270 (les multiples de 90). Pour implémenter ces opérations nous effectuions des calculs sur les coordonnées dépendant de l'angle afin de récupérer les pixels sur l'image d'origine et les placer à leur position à la suite de la rotation sur l'image de destination. Ces calculs individuellement était assez élémentaire, mais les effectuer tous en même temps demande un certain temps d'optimisation.
Toutefois, pour effectuer des rotations plus complexes nous avons décidé de changer d'approche. En effet, si nous effectuons une rotation avec un angle qui n'est pas un multiple de 90 alors l'image n'est plus alignée avec les axes du repère, cela veut dire que certaines parties de l'image peuvent déborder de celle-ci.

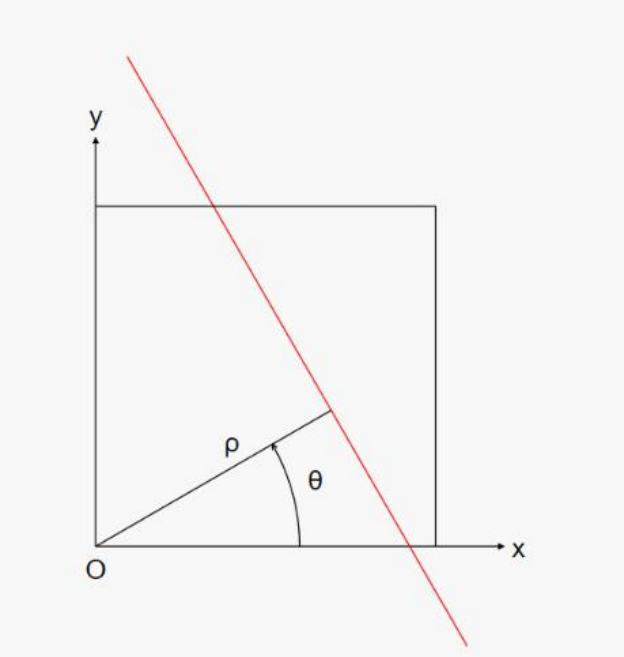
Afin d'éviter ces problèmes nous effectuons un calcul des dimensions de l'image de destination, ainsi si l'angle n'est pas un multiple de 90 alors l'image de destinations sera plus grande que celle d'origine.Une fois cela effectué, nous effectuons ensuite une rotation autour de son centre qui ne change pas selon l'angle de rotation. Un des éléments importants des rotations d'un angle dif-

férent d'un multiple de 90 était de parcourir l'image de destination est non pas l'image de départ car comme la taille de l'image de destination est amené à changer cela peut créer des problèmes d'indices si on ne fait pas attention.

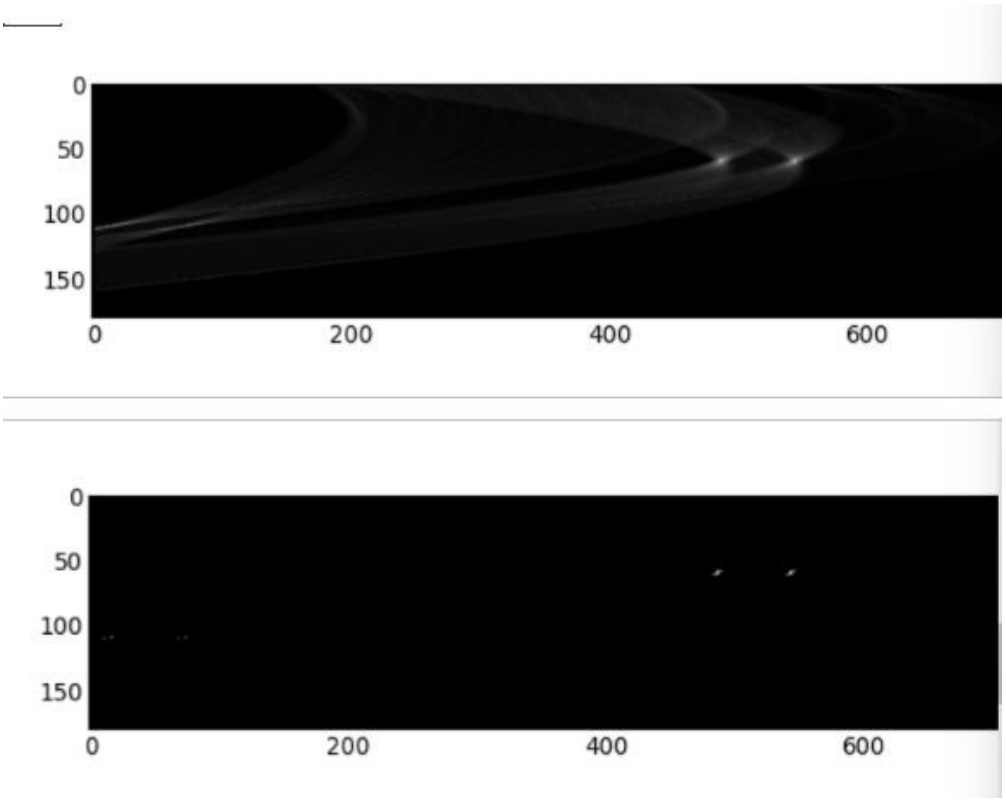
3 Détection

3.1 Détection de la grille

Nous avons implémenté un algorithme nous permettant de déterminer les lignes de la grille du sudoku, afin de faire cela nous a utilisé l'algorithme de la transformation de hough qui nous permet de détecter de manière approximative les droites de l'image. L'image à traiter doit être binaire (noir et blanc) afin que cette fonction nous permette de créer une matrice qui correspond à un domaine rectangulaire dans laquelle chaque valeur représente le nombre de fois ou une droite a été considéré comme une ligne potentielle du sudoku. Plus précisément chaque pixel rencontré nous permet d'incrémenter d'une unité la probabilité qu'une droite existe. Cet algorithme met en application les propriétés mathématiques qui composent une droite, en effet une droite dans un plan peux être mise sous la forme : $ax + by + 1 = 0$, et pour chaque pixel de l'image une droite est tracée dans l'accumulateur. Cependant il est impossible dans notre cas de prendre en compte toutes les valeurs de coefficients (petite, moyenne, grande) c'est pour cela que la transformation de hough utilise les paramètres : $p = \cos(\rho) + \sin(\rho)$ qui correspondent aux coordonnées polaires.



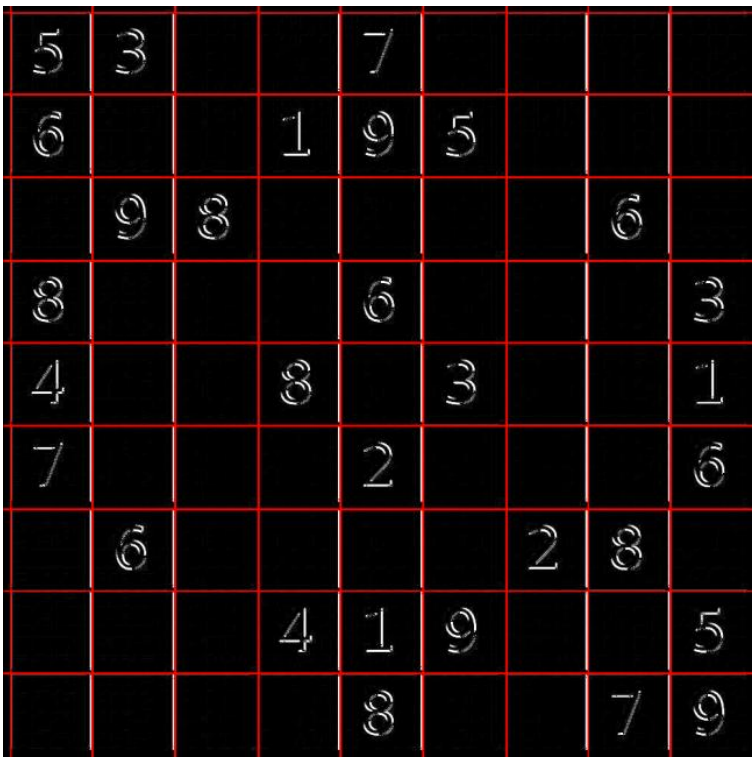
Pour chaque pixel de l'image nous avons donc incrémenter dans la matrice qui est considéré comme un accumulateur les points de la courbe allant de 0 à 180 degrés. Une fois que nous avons traité tous les pixels composant l'image il faut récupérer la valeur maximal contenu dans l'accumulateur, grâce à lui nous allons pouvoir déterminer un seuil à partir duquel les valeurs supérieurs à celui-ci seront considérées comme des droites de l'image.



3.2 Détection et Découpage des cases de la grille

Une fois que nous avons effectuer la transformation de hough est que nous avons déterminer un seuil à partir duquel nous considérons que les paramètres sont des droites nous avons implémenté une fonction nous permettant de tracer les droites ainsi obtenues mais également d'essayer de déterminer les coordonnées de plusieurs points de la grille qui vont nous permettre d'extraire les chiffres qui composent le sudoku. Le tracage de la grille nous permet de verifier que la detection se fait correctement. Pour ce faire nous sommes d'abord partis sur une implémentation qui consistait à changer de couleur chaque pixel un par un pour chaque pixel composant les droites, cette méthode était plutôt efficace au début sur des images parfaitement droite cependant cette méthode permet de tracer des lignes pas seulement droites mais également courbe. Ainsi nous avons donc changé notre méthode de traçage et avons opté pour l'utilisation de la méthode inclus dans la bibliothèque SDL : Drawline. Cette méthode nous à permis de tracer des lignes plus droites qu'à l'origine cependant cette méthode nous oblige à avoir un prétraitement de l'image optimal afin que l'algorithme de Hough et notre algorithme de traçage fonctionne dans les meilleures conditions.

Le résultat du tracage des lignes sur la grille de sudoku :



Ce résultat est trouvé lorsque l'image initiale du sudoku est "bien". En effet sur certaines images, la fonction trouve des droites qui ne sont pas sur la grille du sudoku mais autour. Il faudra donc travailler ce point pour la version finale.

Une fois que la grille et toutes coordonnées de chaque droites horizontales et verticales sont trouvées on procède au découpage. Pour cela on stocke ses valeurs dans deux tableaux, une pour les coordonnées x et l'autre pour les coordonnées y. Pour connaître les positions de toutes les cases, on fait l'intersection de tous les points x et y en faisant une double boucle. Il faut également connaitre la taille de chaque case et pour cela on soustrait les positions x y qui sont voisines.

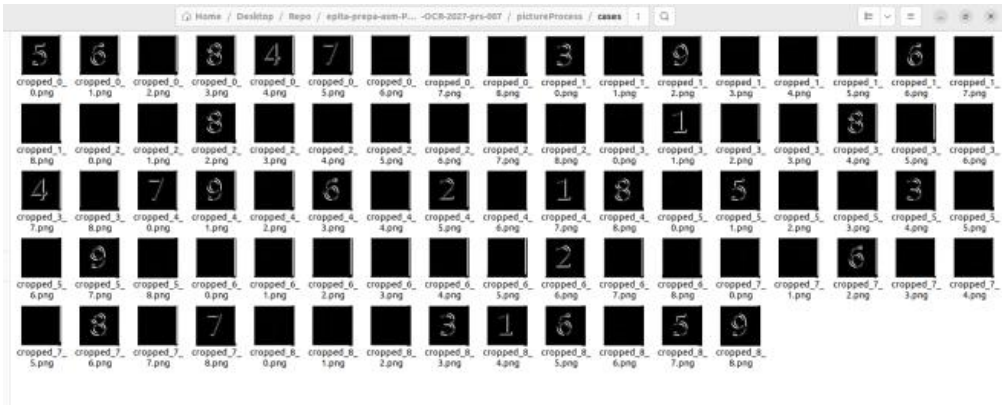
```
for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 9; j++) {
            int x = X[j];
            int y = Y[i];

    //taille de la case
            int width = X[i + 1] - X[i];
            int height = Y[i + 1] - Y[i];

    //position de la case
            SDL_Rect rect;
            rect.x = x;
            rect.y = y;
```

Une fois la taille et la position définie on peut découper votre image en créant d'abord une surface puis en utilisant :

```
// Créer un rectangle découpé
SDL_Surface* croppedImage = SDL_CreateRGBSurface(0, width, height, image->format->BitsPerPixel, ima
SDL_BlitSurface(image, &rect, croppedImage, NULL);
```

Enfin on enregistre le tout dans un dossier.

Le découpage de la grille une fois les coordonnées trouvées :



4 Réseau de neurones

4.1 Qu'est-ce que c'est ?

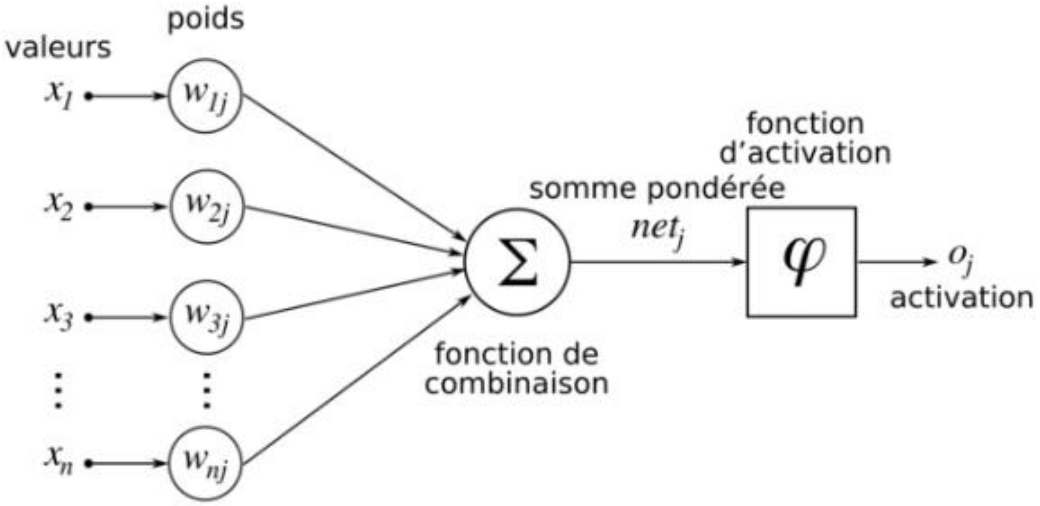
Le réseau de neurones joue un rôle crucial dans les systèmes de reconnaissance optique des caractères. En effet c'est lui qui permet d'identifier les lettres et de reconstituer le texte désiré. La mise en place de cette composante est complexe,

car le réseau fonctionne de manière semblable à un cerveau humain : il peut apprendre pour s'améliorer au fil du temps, produisant ainsi des résultats de plus en plus précis.

Les réseaux de neurones sont structurés en plusieurs couches, comprenant au minimum deux la couche d'entrée et la couche de sortie et une ou plusieurs couches cachées. Les neurones de la couche d'entree contiennent les informations à tester. Les neurones de la couche de sortie fournissent le résultat de la résolution du problème par le réseau. Les couches intermédiaires servent principalement à effectuer des calculs. Leurs valeurs ne sont exploitées que par le réseau de neurones lui-même.

Les neurones de la couche d'entrée sont connectés à tous les neurones de la première couche cachée, s'il y en a une. Les neurones de cette première couche cachée sont à leur tour connectés à tous les neurones de la couche suivante, et ainsi de suite jusqu'à atteindre la couche de sortie.

Chaque connexion entre deux neurones est associée à un poids qui est utilisé pour effectuer divers calculs. En outre, chaque neurone possède une valeur de sortie, généralement comprise entre 0 et 1, qui sert à la fois comme donnée d'entrée pour la couche d'entrée et comme résultat d'un calcul pour les autres couches. Cette valeur représente l'information transmise par chaque neurone aux neurones suivants.



4.2 Initialisation

On initialise la structure qui représentent les poids et biais, en créant des listes dynamiques, et les remplir de valeurs aléatoires (entre -1 et 1). Ainsi puisque les listes dynamique et l'instance de réseau sont créées avec des "malloc" on doit implémenter dans notre code une fonction "destroyNetwork" afin de libérer la mémoire des listes, puis de l'instance (l'ordre étant crucial).

```
typedef struct
{
    double* weightsInputHidden;
    double* weightsHiddenOutput;
    double* biasesHidden;
    double* biasesOutput;
    double* hiddenLayerValues;
    double* outputLayerValues;
    int* sizes;

}Network;
```

4.3 L'Apprentissage

Un réseau ne peut donner de bonnes réponses étant donné que ses valeurs initiales sont aléatoires. Le but de l'apprentissage est donc de permettre au réseau de neurones de s'améliorer et de se rapprocher des résultats optimaux.

Ainsi, un processus bien particulier entre en jeu. Tout d'abord on doit essayer de calculer à quel point le réseau est loin de son objectif, on va donc lui donner des données, le laisser les traiter puis regarder les résultats. D'ici on va calculer la marge d'erreur. Une fois la marge d'erreur acquise, le code va se charger d'ajuster les poids et biais de chaque neurones en fonction de leurs impact sur l'erreur. Cette ajustement ce fait par deux etapes Calcul en avant et Retro-propagation:
Calcul en avant calcule les valeurs des neurones de la couche cachée et de sortie en utilisant les poids, les biais, et la fonction d'activation sigmoïde. Cela permet de produire les sorties du réseau en fonction des entrées fournies.
La rétropropagation ajuste les poids et les biais du réseau pour minimiser l'erreur entre les sorties réelles et attendues. Cela se fait en calculant les deltas (contributions à l'erreur) pour chaque neurone, puis en mettant à jour les poids et les biais en fonction de ces deltas. Cette mise à jour itérative des paramètres du réseau est au cœur de l'apprentissage des réseaux de neurones. Au fur et à mesure des corrections, le réseau de neurones va s'améliorer et réduire son er-

reur vis-à-vis de l'output "désirée". C'est à ce moment là seulement que l'on va pouvoir tester notre réseau et voir comment il a réussi à apprendre la fonction en regardant ses output en fonction des inputs données.

4.4 XOR

Pour la fonction xor notre modèle d'entraînement est le suivant: double allInput[4][2] = 0, 0 | 0, 1 | 1, 0 | 1, 1 | double wantedOutput [4] = 0, 1, 1, 0 ;

Et on teste pour les 4 possibilités d'input avant l'entraînement et après l'entraînement:

```
navi@navi-virtual-machine:~/Desktop/Commun$ ./XORNN
-------------------- AVANT ENTRAINEMENT --------------------
Input: 0.000000 0.000000, Output: 0.089432
Input: 0.000000 1.000000, Output: 0.060031
Input: 1.000000 0.000000, Output: 0.064020
Input: 1.000000 1.000000, Output: 0.046022
-------------------- APRES ENTRAINEMENT --------------------
Input: 0.000000 0.000000, Output: 0.016455
Input: 0.000000 1.000000, Output: 0.978026
Input: 1.000000 0.000000, Output: 0.978140
Input: 1.000000 1.000000, Output: 0.025320
```

5 Résolution de la grille

5.1 Résolution

Pour la résolution de la grille, on avons utilisé un algorithme qui teste un par un les cases. Pour ce faire, nous avons conçu une première fonction chargée de déterminer si une case peut être remplie par le chiffre "n". On rappelle les règles du sudoku : un chiffre ne peut être placé que s'il n'apparaît pas déjà dans la même ligne, la même colonne et le carré de 3x3 correspondant.

Nous avons créé une fonction (possible) qui vérifie si cette règle est respectée avec ce chiffre “n”. Cette fonction parcourt la ligne, la colonne et le petit carré de 3 fois 3 et vérifie la présence du chiffre “n”. Si ce chiffre est trouvé, alors cette case n'est pas valide avec ce chiffre “n” la fonction renvoie 0 (false). Sinon la case est compatible avec ce chiffre et la fonction renvoie 1 (true). Pour la résolution du sudoku nous avons utilisé une matrice 9 fois 9 pour stocker les valeurs de la grille. Les cases vides dans le sudoku auront comme valeur 0.

```
int possible(short n, size_t x, size_t y){
    for (size_t val = 0; val < 9; val+=1){
        //test sur la ligne
        if (sudoku[val][y] == n){
            return 0;
        }
        //test sur la colonne
        if (sudoku[x][val] == n){
            return 0;
        }
    }
    for (size_t Y = 0; Y < 3; Y += 1){
//calcul de la position de Ro et Co les premières valeurs
//dans le petit carré trois*trois dans laquelle
        //la valeur n se trouve
        size_t Ro = 6;
        size_t Co = 6;
        if(x < 3)
            Ro = 0;
        else
            if(x < 6)
                Ro = 3;
        if(y < 3)
            Co = 0;
        else
            if(y < 6)
                Co = 3;

        for(size_t X = 0; X < 3; X+=1){
            //test sur le carre de trois
            if (sudoku[Ro+Y][Co+X] == n){
                return 0;
            }
        }
    }
    return 1;
}
```

Ensuite nous parcourons le sudoku case par case, si la case est vide (c'est-à-dire que sa valeur est égale à 0) alors la fonction test les valeurs de 1 à 9 en faisant appel à la fonction "possible". Si la fonction “possible” renvoie 1 alors on place ce chiffre dans la case. Si la fonction renvoie "faux" pour toutes les valeurs de 1 à 9, cela signifie que nous avons fait un mauvais choix précédemment. La fonction revient à la case précédente pour tester d'autres possibilités. Ce processus de retour en arrière s'effectue de manière récursive. On répète ces étapes jusqu'à ce que la grille soit résolue.

```
int solve(){
    for(size_t i = 0; i < 9; i++){       //parcour toutes les cases du sudoku
    for(size_t j = 0; j < 9; j++){

        if(sudoku[i][j] == 0){
            for(short num = 1; num <= 9; num++){
                if (possible(num,i,j)){       //test de la case avec le chiffre num
                    sudoku[i][j] = num;
                    if(solve()){
                        return 1; //appel recursif, si le sudoku est resolu
                        //alors le num est bon sinon le num n'est pas bon
                    }
                    sudoku[i][j] = 0; //la valeur ne marche pas donc
                    // on reniatilise a zero et on continue avec les autres nums
                }
            }
            return 0;
        }

    }
    }
    return 1;
}
```

5.2 Sauvegarde de la grille résolue

Pour sauvegarder la grille résolue, nous avons créé un nouveau fichier, parcouru la matrice résolue, et écrit les valeurs dans ce fichier à l'aide de la fonction "fprintf".

Nous devons maintenant aborder la création de l'affichage de la grille sous forme d'image. Pour cela, nous envisageons de prendre des images individuelles des chiffres de 1 à 9. Ensuite, nous prévoyons de les juxtaposer pour constituer la grille du sudoku, en ajoutant des lignes droites pour délimiter chaque case.

6 Conclusion

6.1 Tableau d'avancement

	Intermediaire	Final
Prétraitement	70%	100%
Rotation	100%	100%
Reconnaissance de la grille	60%	100%
Découpage de la grille	95%	100%
Réseau de neurones	20%	100%
Resolution du sudoku	95%	100%

6.2 Conclusion

Ce document présente les réalisations atteintes au cours de notre soutenance intermédiaire pour le projet de résolution de Sudoku en langage C. Grâce à ce projet nous avons acquis de nouvelles connaissances intéressantes d'un point de vue algorithmique. Nous avons constaté avec satisfaction les avancées réalisées jusqu'à présent. Nous allons donc continuer pour finaliser ce projet. Cette phase nous a permis de mieux appréhender l'ampleur et la complexité du projet. Et nous sommes désormais sur la bonne voie pour accomplir notre projet avec succès.