

O.C.R

Arthur Oldrati | Terence Degroote | Kerian Allaire | Benoit Burg

24 Octobre 2021

Rapport de Soutenance 1

Table des matières

1	Introduction	4
1.1	Présentation de l'équipe	4
1.1.1	Arthur OLDRATI	4
1.1.2	Kerian ALLAIRE	4
1.1.3	Terence DEGROOTE	5
1.1.4	Benoit BURG	5
1.2	Répartition des tâches	6
2	Pré-traitement de l'image	6
2.1	Filtre noir et blanc	6
2.2	Binarisation et méthode d'Otsu	8
2.3	Rotation	9
2.4	Filtre de Canny	11
2.4.1	Principe	11
2.4.2	Pré-requis	11
2.4.3	Gradient d'intensité	11
2.4.4	détection des maximum locaux	12
2.4.5	nettoyage de l'image	12
3	Reconnaissance de la grille	13
3.1	La transformé de Hough	13
3.2	Détection de la grille	14
4	Réseau de neurone	15
4.1	Initialisation du réseau de Neurone	15
4.2	Fonctionnement du réseau	15
4.2.1	Résolution	15
4.2.2	Entraînement	16
4.3	Sauvegarde des valeurs et test du réseau	17
4.3.1	Manuel	18
5	Résolution	22
5.1	Sauvegarde et traitement des fichiers	22
5.2	Algorithme de résolution	23
6	Suivi du projet	25

7 Conclusion

25

1 Introduction

L'OCR est le projet du troisième semestre consistant à résoudre un sudoku via la lecture d'une photo de celui-ci. Pour cela il est nécessaire de traiter l'image pour la rendre exploitable, puis de reconnaître le sudoku sur celle-ci en utilisant des algorithmes tels que *Hough Line* ou encore *Canny*. En effet de reconnaître les nombres allant de 1 à 9 grâce à un réseau de neurones, pour ensuite pouvoir résoudre la grille et l'afficher.

1.1 Présentation de l'équipe

1.1.1 Arthur OLDRATI

Bonjour ! Je suis dans le F.C HUMBLE pour ce projet qu'est l'OCR, puisque c'est un trait qui me définit parfaitement.

Ce projet est l'occasion pour moi d'améliorer ma maîtrise du C en me confrontant à divers Segfault :)

De plus les projets sont une source de motivation pour moi aussi j'affirme que ce projet sera une réussite

L'OCR en lui-même me plaît beaucoup car en effet j'ai intégré l'EPITA en étant intéressé par l'IA, ce projet me permet donc de me rendre compte de ce que cela peut être.

Je vais cependant pour cette première soutenance m'atteler au traitement d'image et notamment la reconnaissance de la grille et des cases de notre sudoku.

1.1.2 Kerian ALLAIRE

Je suis Kerian Allaire, étudiant en deuxième année d'Epita. j'ai beaucoup apprécié les différents projets de l'année passée, que cela soit l'Afit, qui m'a permis de comprendre ce qu'était qu'un projet, de part le principe de deadline, de rendu intermédiaire, et nous a également appris à avoir une certaine régularité dans notre travail. Le second projet m'a également beaucoup plu. En effet, celui-ci nous a laissé bien plus de liberté, nous permettant ainsi de nous diriger dans des directions intéressantes que nous n'aurions peut-être pas découvert avec un projet plus dirigé. Ces projets nous ont également appris à chercher par nous-même, à découvrir de la documentation sur

des concepts scientifiques et informatiques que nous n'avions pas forcément vu dans le cadre d'Epita. L'OCR m'intéresse beaucoup étant donné que je suis intéressé par le concept d'IA, et le traitement d'image m'était assez nébuleux mais me semblait pertinent. J'ai donc hâte de continuer ce projet et d'enrichir mes connaissances dans les différents domaines cités plus haut.

1.1.3 Terence DEGROOTE

Pour ma part, je suis le benjamin d'une fratrie de trois garçons. J'ai pratiqué beaucoup de sport durant ma jeunesse(notamment le rugby pendant 7 ans). Je suis venu à EPITA après avoir choisi l'option ISN(Informatique et Science du Numérique) au lycée Alain. Je me suis alors découvert un plaisir dans l'informatique, j'ai aussi été influencé par mon grand frère qui lui aussi est actuellement à l'Epita(en 4ème année). Je souhaite à travers ce projet continuer ma formation dans les sciences de l'ingénieur en faisant un projet totalement nouveau pour moi qui n'ait jusqu'à présent jamais programmé en C. J'ai toujours été intrigué par la reconnaissance de texte et je suis donc très intéressé dans le projet de ce semestre qui je l'espère va m'apprendre les bases d'un OCR.

1.1.4 Benoit BURG

Je suis Benoit Burg, étudiant en B2 cette année et en E2 l'année dernière. L'OCR est un projet qui me passionne beaucoup notamment la partie réseau de neurones puisque je suis très intéressé par tout ce qui rapproche de l'IA. Je suis très motivé et j'espère bien réussir ce projet et faire encore mieux que le projet de S2 ou bien celui du S1. J'essaye également d'apprendre à me servir de *VIM* correctement pour améliorer mon efficacité... même si pour l'instant, les résultats sont loin d'être ceux attendus !

1.2 Répartition des tâches

Les tâches pour chacun des membres seront répartie de la façon suivante :

	Arthur	Terence	Kerian	Benoit
Chargement d'une image et suppression des couleurs		X		
Rotation manuelle de l'image		X		X
Découpage de l'image	X		X	
Détection de la grille et position des cases	X		X	
Réseau de neurone				X
Solver				X

TABLE 1 – Répartition du travail

2 Pré-traitement de l'image

2.1 Filtre noir et blanc

Le filtre pour mettre en nuances de gris une image en couleur est possible de différentes façon, la première que l'on risque probablement sélectionner et de récupérer les 3 composantes de couleurs de chaque pixel et de créer sa nouvelle couleur en gris d'intensité égale a la moyenne des 3 composantes rouges bleues et vertes. Une seconde et d'applique la formule de $0.3*r + 0.59*g + 0.11*b$. Ce filtre grayscale est moins intéressant pour l'OCR puisqu'il embellit la photo mais n'est pas représentatif de la grille de sudoku notamment si elle est dans les teintes bleu comme c'est le cas de la photo numéro 3.

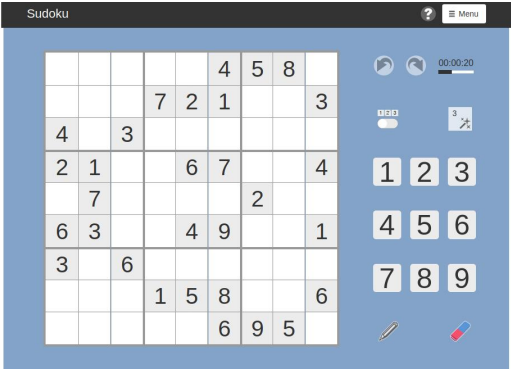


FIGURE 1 – sans filtre gris

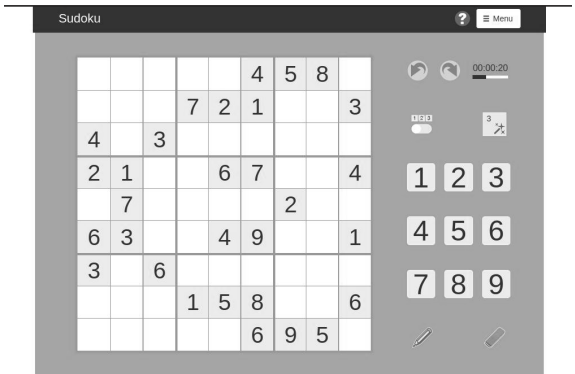


FIGURE 2 – avec filtre gris

2.2 Binarisation et méthode d'Otsu

Nous avons ensuite cherché un moyen pour rendre notre image en noir et blanc c'est à dire soit des pixel d'intensité 255 pour les 3 composantes (affichant blanc) soit 0 pour les noirs. Pour se faire il nous fallait établir un seuil à partir duquel on bascule le pixel en blanc ou en noir. Nous avons alors trouvé la méthode d'Otsu permettant de calculer le seuil optimal en fonction de chaque photo. Nous avons donc calculer la variance de chaque seuil (0 à t avec $t < 256$) et la variance complémentaire (t à 255). Nous sélectionnons ensuite le seuil pour lequel la variance inter classe est la plus petite. Cela permet de rester précis que l'image soit plus sombre ou plus claire.

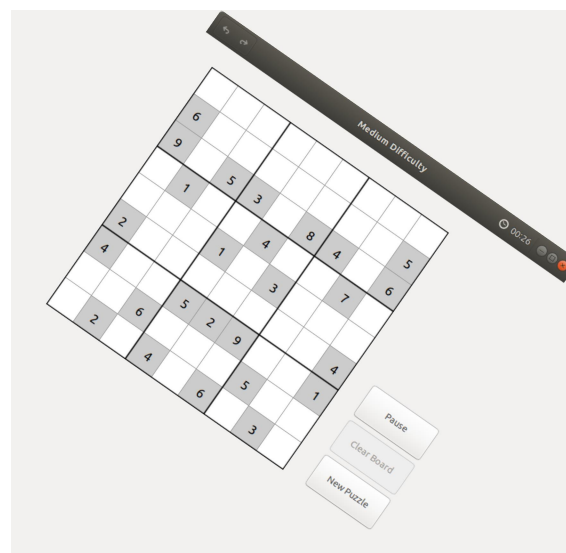


FIGURE 3 – Sans filtre

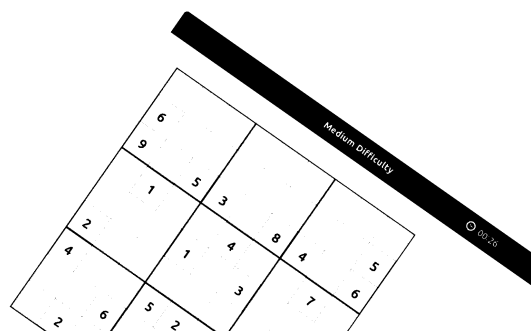
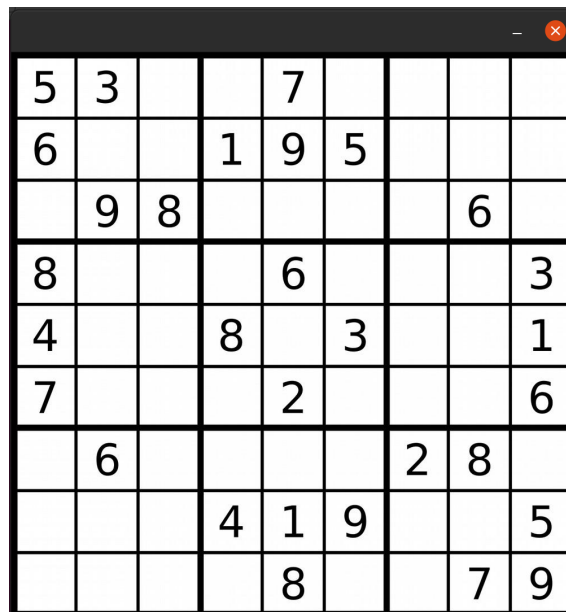


FIGURE 4 – avec binarisation

2.3 Rotation

La rotation d'une image se fait manuellement en appelant le programme `./rotation` et en passant en paramètre un entier. Cela effectue une rotation sur l'image donnée dans la fonction. Voici un exemple avec une rotation de 45° :



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	

FIGURE 5 – Sans rotation

Cette fonction utilise la fonction *rotozoom* de la librairie de *SLD-gfx*, et permet donc d'effectuer très facilement une rotation et un zoom.

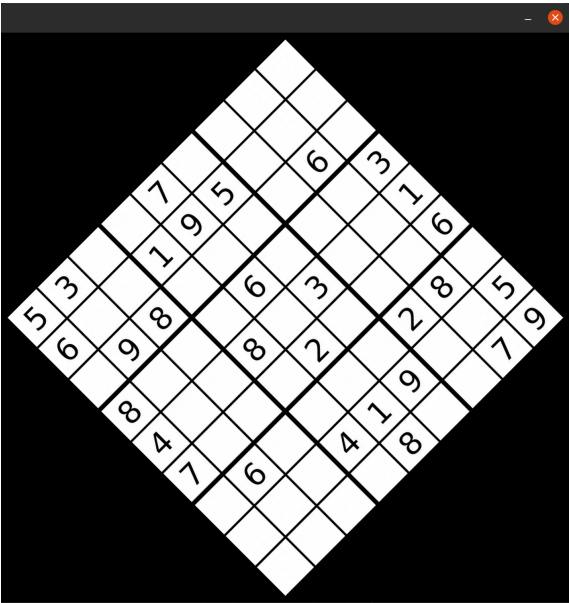


FIGURE 6 – Avec rotation de 45°

2.4 Filtre de Canny

2.4.1 Principe

Le filtre de Canny est un filtre qui doit remplir les critères suivant, énoncés par John Canny en 1986 :

- **une bonne détection** : un faible taux d'erreur dans la signalisation des contours
- **une bonne localisation** : une minimisation des distances entre les contours détectés et les contours réels
- **une réponse claire** : une seule réponse par contour et pas de faux positifs

2.4.2 Pré-requis

Afin de faire fonctionner le filtre de Canny, il faut pour cela avoir une image remplissant plusieurs conditions. La première est que cette image a été modifiée afin de n'avoir plus que des nuances de gris. Cela nous permet d'avoir les valeurs respectives r, g et b identiques dans chaque pixel. Cela facilite largement les calculs, et permet de calculer notamment le gradient. La seconde condition est que l'image doit être nettoyé. Pour cela, nous devons passer des filtres tel que le filtre Gaussien, afin d'éliminer le bruit parasite qui pourrait nuire à la détection des contours. En effet, comme nous allons pouvoir le voir, des pixels isolés peuvent créer des contours au milieu des cases qui risquent alors de fausser la détection des lignes.

2.4.3 Gradient d'intensité

Une fois le pré-traitement effectué, nous pouvons alors appliquer un filtre afin de déterminer le gradient en tout pixel de l'image. Nous nous servons ainsi du filtre de Sobel, qui permet de permet, à l'aide d'une matrice de dimension 3X3 : $\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$ de calculer le gradient horizontal (sur l'axe x), et une seconde $\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$ pour calculer le gradient vertical (sur l'axe y). Cela permet d'effectuer un premier filtrage de l'image. Nous calculons alors une norme du gradient en tout point afin de combiner ces deux gradients. Nous en profitons également pour déterminer l'orientation du gradient en tout point. Nous en avons alors fini avec les calculs de gradients.

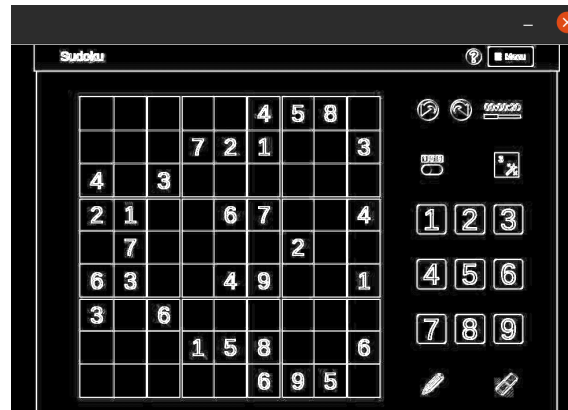


FIGURE 7 – Image après combinaison des gradients

2.4.4 détection des maximum locaux

Nous avons alors une image qui est passé par plusieurs filtres, ce qui remplit la première condition de Canny : un faible taux d’erreurs dans la détection des contours. Nous devons alors minimiser les distances entre les contours détectés et les contours réels.

Afin d’affiner notre image, nous allons déterminer quels sont les gradients dont la norme est un maximum local. En effet, les points qui n’appartiennent pas aux maximums sont des imperfections, et rendrons alors l’écriture des bords imprécis. Le local dont il est ici question est déterminé grâce à l’orientation du gradient : en fonction de l’angle de celui-ci, nous allons comparer notre norme avec la norme de deux des 8 pixels voisins de notre pixel (ayant la forme d’un carré, notre pixel initial étant le centre) par exemple, si l’angle est entre 0 et 22.5, nous prenons le point de droite et celui de gauche (point étant sur la même ligne, même de colonne différente). Nous comparons les normes : s’il est inférieur, nous mettons sa norme à 0 (pixel noir), sa norme étant la valeur de ses composantes RGB. Nous effectuons ces opérations pour l’ensemble des pixels.

2.4.5 nettoyage de l’image

Après ces divers filtres et traitements, nous pouvons observer l’apparition de bruit sur l’image. Afin de nettoyer celui-ci, nous utilisons un algorithme de seuil. Il marche de la façon suivante :

- Si la valeur du r de RGB est supérieur au seuil maximal, alors le point est fort, il devient blanc.

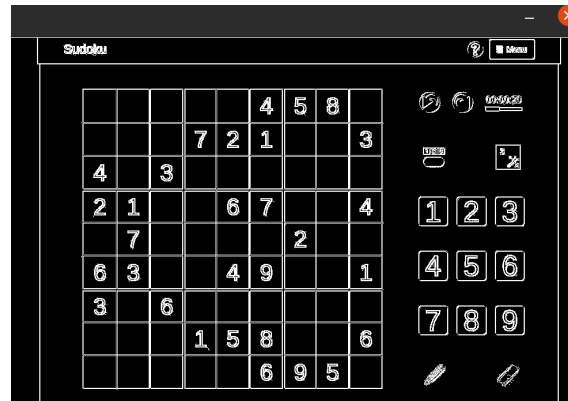


FIGURE 8 – Traitement de l'image terminé

- Si elle est inférieure à un seuil minimal, alors le pixel est faible, il devient noir
- Si elle est ni inférieure au seuil minimal, ni supérieure maximal, alors nous vérifions que l'un de ses voisins est fort. Si c'est le cas, alors il est également fort, et il devient blanc. Le cas échéant, il est faible, et devient noir. Ainsi, l'image est nettoyée, et le filtre de Canny est alors totalement appliqué.

3 Reconnaissance de la grille

Une des plus grande problématique du traitement d'image est en effet la détection de la grille. Pour répondre à cette problématique nous avons choisis une approche se basant sur la transformé de Hough qui une fois implémenté nous permet de repérer toutes les droites de notre image, cela va constituer la première étape de notre détection.

3.1 La transformé de Hough

Le fonctionnement de la transformé de Hough est le suivant :// Dans un premier temps, nous devons calculer l'équation polaire de la forme $ro = x \cdot \cos(\text{teta}) + y \cdot \sin(\text{teta})$, avec teta un angle. Nous allons alors calculer ro pour l'ensemble des angles de 0 à 180 degrés. Nous ajoutons un 1 dans un malloc, un tableau utilisant les qualités des pointeurs, de lignes d'index teta et de colonnes d'index ro. Cela nous permet d'avoir la densité des droites, c'est à dire le nombre de point par lesquels passent chaque droite. Après cela, nous parcourons notre

malloc afin de ne récupérer que les droites ayant une certaine densité avant de les stocker dans une liste de tuple de composants ro, teta et densité, ces deux structures structure ayant été créées pour notre OCR. Ce sont les lignes présentes sur l'image. Une fois cela fait, nous déterminons deux points appartenant à chaque droite sélectionnées qui appartiennent à l'image afin de tracer celle-ci sous forme de pixel vert sur notre image.

Bon, avoir des droites c'est bien mais si nous ne savons pas quoi en faire ce n'est pas très utile. Avant de passer à la deuxième étape une rotation de l'image grille sera potentiellement nécessaire, cela tombe bien Hough nous permet de trouver l'angle de rotation. Une fois les préparations terminées nous pouvons implémenter la suite de notre algorithme.

3.2 Détection de la grille

Afin de détecter la grille, nous allons nous servir des droites précédemment tracées. Pour cela, nous allons aller de droite en droite, et les stocker dans une liste de tuple ayant pour composant le x (ou le y) de la droite précédente, le x (ou le y) de la droite suivante, et la distance entre les deux (x et y dépend si nous cherchons les droites horizontales ou verticales). Nous allons parcourir notre liste et supprimer toutes les valeurs qui ont une distance inférieure à un certain seuil (un centième du plus grand côté de l'image), ce sont des entre-deux-lignes trop petit pour être un sudoku, et du bruit créé par Canny, étant donné que ce sont des contours, si une ligne était trop large, cela a pu créer deux lignes au lieu d'une avec canny et donc Hough. Une fois cela effectué, nous créons une copie de cette liste. Nous allons ensuite trier cette copie afin d'avoir la valeur médiane de la distance.

Nous allons ensuite parcourir la liste initiale. Nous allons vérifier pour une case si sa distance est égale à sa médiane avec une marge d'erreur d'un centième du côté le plus long de l'image. Une fois cela effectué, nous vérifions que c'est le cas pour huit de ses voisins, étant donné qu'un sudoku possède neuf cases de larges et de haut. Sinon, nous avançons simplement jusqu'à la prochaine case à tester. Si elle possède bel et bien huit voisins, alors nous récupérons ceux-ci et nous le stockons. Nous réitérons l'entièreté de l'opération pour les droites verticales, étant donné que nous commençons avec les droites hori-

zontales. Nous avons ainsi les différents carrés composants le sudoku, nous les traçons donc pour des raisons esthétique.

Il ne reste plus qu'à découper des carré de côté égaux à la plus petite distance de chaque case, avant de les dimensionner de façon à avoir des images de 28 par 28 pour notre réseau de neurone, et de sauvegarder les images sous forme de fichier.

4 Réseau de neurone

4.1 Initialisation du réseau de Neurone

L'objectif de ce premier réseau neurone est d'apprendre l'opérateur *XOR* qui est l'opérateur OU EXCLUSIF définit ainsi :

- $0 \text{ XOR } 0 \Rightarrow 0$
- $1 \text{ XOR } 0 \Rightarrow 1$
- $0 \text{ XOR } 1 \Rightarrow 1$
- $1 \text{ XOR } 1 \Rightarrow 0$

Un réseau de neurone étant toujours composé d'une couche de noeuds d'entrées, d'une couche de noeuds "cachés" et d'une couche de sortie ; le notre va posséder très exactement deux noeuds d'entrées, quatre noeuds "cachés" et bien sur un seul noeud de sorti. Les noeuds d'entrées prendront comme valeur bien évidemment soit **0** soit **1** et le noeud de sorti étant censé prendre comme valeur le résultat du XOR des deux entrées. Lors de l'entraînement notre réseau de neurone va itérer ses entraînements pendant près de 10.000 générations. Cette valeur est totalement arbitraire et nous l'avons choisis car nous avons remarqué c'est environ à ce moment là que l'erreur s'approche d'environ 0.05.

4.2 Fonctionnement du réseau

4.2.1 Résolution

Que ce soit pour les entraînements comme pour les tests, la résolution est une étape indispensable. En réalité cette étape est plus simple qu'il n'y parait. Nous allons nous balader de couche en couche en déterminant les valeurs des neurones à partir des précédents.

Voici le fonctionnement en détail :

- Première boucle d'opérations durant laquelle nous allons déterminer les valeurs des neurones de la couche "cachée" à partir de ceux d'entrée. Voici le raisonnement effectué pour déterminer la valeur d'un neurone : nous multiplions chaque valeur des neurones d'entrée par les poids associés entre cette entrée et le neurone. Nous les additionnons tous, puis on ajoute à cela le biais du neurone. Néanmoins nous souhaitons trouver une valeur entre 0 et 1. Pour cela nous allons utiliser la fonction sigmoïde définie ci-dessous.

- Seconde boucle d'opérations. Pour celle-ci nous allons effectuer le même raisonnement que la précédente mais en utilisant bien évidemment les valeurs des neurones "cachés" en entrée et la sortie en arrivée. Encore une fois, grâce à la fonction sigmoïde le résultat obtenu se trouve entre 0 et 1. Pour savoir si le neurone de sortie doit afficher 1 ou 0, nous avons simplement pris la valeur de 0.5 comme valeur **seuil** : Si la sortie est strictement supérieure nous le prendrons comme un 1 et à l'inverse, si la sortie est strictement inférieure ou égale au seuil nous considérerons le résultat comme un 0.

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}.$$

FIGURE 9 – Fonction Sigmoïde

4.2.2 Entraînement

Pour s'entraîner, le réseau de neurone possède une série des 4 tests du XOR. A chaque génération cette série est complètement mélangée pour éviter que notre réseau de neurone ne s'entraîne uniquement que sur un schéma de répétition. A chaque génération, l'entraînement va simplement résoudre le test qui lui est donné. Mais c'est ensuite que

les choses changent. En effet, nous allons faire ce que l'on appelle la **"BackPropagation"**. C'est à dire que nous allons parcourir notre réseau de neurone et modifier très légèrement toutes les valeurs. La grande question est comment modifier ses valeurs pour se rapprocher du bon résultat ? Pour cela nous allons calculer le gradient de chaque valeur de sortie. Pour calculer ce gradient, nous allons avoir besoins de calculer la différence entre le résultat sorti et celui attendu puis la dérivé de la fonction sigmoïde avec en paramètre le résultat sorti. Nous allons ensuite multiplier les deux. Pour finir, nous allons multiplier le résultat par ce que l'on appelle le *"Learning Rate"*, qui permet de d'amplifier et surtout de s'assurer d'une certaine modification à chaque essai. Une fois cette expression mise en place, il suffit de modifier chaque biais et chaque poids en ajoutant à leur valeur actuelle le gradient. Bien évidemment la valeur de ce gradient est différente pour chaque couche de neurone et même chaque neurone puisqu'il s'appuie sur la valeur de ceux-ci.

Cette formule représente en quelque sorte la modification nécessaire pour que l'erreur soit minimal en fonction de chaque biais et poids. Pour représenter cela, il est possible de s'imaginer notre fonction variant pour chaque poids et biais comme une courbe en 3 dimensions. Notre point de départ étant suggéré comme une balle. Notre objectif est de faire rouler cette balle vers le point le plus bas de cette courbe. Ce point étant le minimum, cela signifie qu'une fois atteint, notre réseau s'approche de l'erreur minimal. Cependant, nous n'appliquons pas les lois de la gravité pour faire descendre notre balle, mais plutôt la formule exprimé ci-dessus.

4.3 Sauvegarde des valeurs et test du réseau

Une fois que nous avons obtenu nos valeurs, nous ne souhaitons pas les perdre. Nous allons donc les sauvegarder dans un fichier dont le nom sera donné en paramètre. Le format de sauvegarde est simple, le voici :

- Poids des neurones cachés.
- Poids des neurones de sorties (ici un seul)
- Biais des neurones cachés.
- Biais des neurones de sorties

```
tebion@tebion-Creator-15-A10SF:~/OCR/ocr_sudoku/Scripts/OCR$ cat test.test
1.160183
6.054511
3.334346
0.692432
1.002711
5.951059
3.476393
0.566947
-2.952015
8.571478
-7.103246
-2.242919
-1.461482
-2.513302
-5.233067
-0.169033
-1.856379
```

FIGURE 10 – Fichier de sauvegarde du réseau de neurone

Une fois que nous avons sauvegardé ces valeurs, nous pouvons tester notre réseau de neurone. Pour cela il suffit d'appeler le programme `./xor` en passant en paramètre le nom du fichier. Pour plus de détail voici le manuel concernant le réseau de neurone trouvable dans le réseau de neurone accompagné de photos pour illustrer.

4.3.1 Manuel

Voici l'appel du programme ainsi que les arguments possibles :

```
./xor file.name [0-1][0-1]
```

- Pour lancer un entraînement, il suffit d'appeler le programme `./xor` suivi du nom du fichier sous lequel vous souhaitez enregistrer vos valeurs. Une fois ceci fait vous verrez un nouveau fichier avec dedans les valeurs de votre réseau de neurone, sous le format décrit au dessus.

De plus sur la console, vous verrez apparaître le résultat et la progression de l'entraînement. Dans un premier nous affichons toutes les valeurs de départ de notre réseau de neurone. Ensuite arrive une large partie avec les résultats des entraînements toutes les 1000 générations ainsi que la première. Ceux-ci sont très simple à interpréter : En premier les valeurs de l'entraînement, ensuite le résultat compris entre 1 et 0 ainsi que son interprétation, puis pour finir le résultat attendu. Si celui-ci est faux alors la ligne s'affichera en rouge, en vers sinon.

```

tebion@tebion-Creator-15-A10SF:~/OCR/Ocr_sudoku/Scripts/OCR$ ./xor test.test
===== Values =====

=> Weights :

Hidden Weights 0-->0 = 0.910000
Hidden Weights 0-->1 = 0.430000
Hidden Weights 0-->2 = 0.770000
Hidden Weights 0-->3 = 0.010000
Hidden Weights 1-->0 = 0.410000
Hidden Weights 1-->1 = 0.710000
Hidden Weights 1-->2 = 0.530000
Hidden Weights 1-->3 = 0.520000

Output Weights 0-->0 = 0.530000
Output Weights 1-->0 = 0.750000
Output Weights 2-->0 = 0.760000
Output Weights 3-->0 = 0.540000

=> Bias :

Hidden Bias [0] = 0.170000
Hidden Bias [1] = 0.740000
Hidden Bias [2] = 0.150000
Hidden Bias [3] = 0.140000

Output Bias [0] = 0.620000
=====

```

FIGURE 11 – Valeurs initiales aléatoires du réseau de neurone.

```

===== EPOCH N°0 =====
[1] XOR [1] => 0.935598 => 1 Expected : 0
[1] XOR [0] => 0.917053 => 1 Expected : 1
[0] XOR [1] => 0.917659 => 1 Expected : 1
[0] XOR [0] => 0.890869 => 1 Expected : 0

===== EPOCH N°1000 =====
[1] XOR [0] => 0.507855 => 1 Expected : 1
[1] XOR [1] => 0.533439 => 1 Expected : 0
[0] XOR [0] => 0.477457 => 0 Expected : 0
[0] XOR [1] => 0.493170 => 0 Expected : 1

===== EPOCH N°2000 =====
[1] XOR [0] => 0.523437 => 1 Expected : 1
[0] XOR [0] => 0.450444 => 0 Expected : 0
[1] XOR [1] => 0.549551 => 1 Expected : 0
[0] XOR [1] => 0.505202 => 1 Expected : 1

===== EPOCH N°3000 =====
[1] XOR [1] => 0.574577 => 1 Expected : 0
[0] XOR [0] => 0.287747 => 0 Expected : 0
[1] XOR [0] => 0.569163 => 1 Expected : 1
[0] XOR [1] => 0.574645 => 1 Expected : 1

===== EPOCH N°4000 =====
[0] XOR [1] => 0.683493 => 1 Expected : 1
[0] XOR [0] => 0.214935 => 0 Expected : 0
[1] XOR [1] => 0.422433 => 0 Expected : 0
[1] XOR [0] => 0.684000 => 1 Expected : 1

```

FIGURE 12 – Résultats de certains test de l'entraînement.

```
=====> EPOCH N°10000 <=====
[0] XOR [0] => 0.057024 => 0 | Expected : 0
[1] XOR [0] => 0.949228 => 1 | Expected : 1
[0] XOR [1] => 0.948974 => 1 | Expected : 1
[1] XOR [1] => 0.052745 => 0 | Expected : 0

===== Values =====

=> Weights :

Hidden Weights 0-->0 = 2.828940
Hidden Weights 0-->1 = 0.842676
Hidden Weights 0-->2 = 5.978119
Hidden Weights 0-->3 = 2.281841
Hidden Weights 1-->0 = 2.664122
Hidden Weights 1-->1 = 0.981460
Hidden Weights 1-->2 = 6.004736
Hidden Weights 1-->3 = 2.442160

Output Weights 0-->0 = -5.673422
Output Weights 1-->0 = -0.951523
Output Weights 2-->0 = 8.583483
Output Weights 3-->0 = -5.017744

=> Bias :

Hidden Bias [0] = -4.304003
Hidden Bias [1] = 0.778950
Hidden Bias [2] = -2.562646
Hidden Bias [3] = -3.728265

Output Bias [0] = -2.574458
=====
```

FIGURE 13 – Valeurs finale du réseau de neurone.

- Maintenant que nous avons entraîné un réseau de neurone, nous souhaitons le tester, pour cela il suffit de rappeler `./xor` avec derrière le nom du fichier contenant les valeurs de notre réseau. Il faut ensuite rajouter 2 arguments. Ces arguments seront les valeurs d'entrées de notre réseau. Il est donc nécessaire que celles-ci soit au choix **0** ou **1**.

Le résultat de ce test se trouvera dans la console, affiché comme les tests précédemment. De plus, pour vérifier si les valeurs du réseau de neurone sont les bonnes, nous les affichons en dessous.

```
teblon@teblon-Creator-15-A10SF:~/OCR/ocr_sudoku/Scripts/OCR$ ./xor test.test 1 0
Reading .../
##### RESULT #####
| [1] XOR [0] => 0.949262 => 1 | Expected : 1
#####
===== Values =====
=> Weights :
Hidden Weights 0-->0 = 2.828940
Hidden Weights 0-->1 = 0.842676
Hidden Weights 0-->2 = 5.978119
Hidden Weights 0-->3 = 2.281841
Hidden Weights 1-->0 = 2.664122
Hidden Weights 1-->1 = 0.981460
Hidden Weights 1-->2 = 6.004736
Hidden Weights 1-->3 = 2.442160
Output Weights 0-->0 = -5.673422
Output Weights 1-->0 = -0.951523
Output Weights 2-->0 = 8.583483
Output Weights 3-->0 = -5.017744
=> Bias :
Hidden Bias [0] = -4.304003
Hidden Bias [1] = 0.778950
Hidden Bias [2] = -2.562646
Hidden Bias [3] = -3.728265
Output Bias [0] = -2.574458
=====
Done
```

FIGURE 14 – Test du réseau de neurone.

REMARQUES

Evidemment une certaine gestion d'erreur a été mise en place, voici les points importants à respecter :

- Un nom de fichier obligatoire. Si celui-ci n'existe pas il sera alors créé.
- Les entrées du XOR doivent bien sur être 1 ou 0 et uniquement ceux-ci. Sinon une erreur s'affichera.
- Bien pensé à rentrer les deux entrées et celle-ci espacé d'un espace puisqu'il s'agit de 2 arguments distincts.

Pour finir il est utile de savoir qu'il est possible de rentrer un nouveau nom de fichier pour effectuer un entraînement, puis dans la même ligne rentrer les arguments de tests. Le test s'effectuera alors à la suite de cet entraînement.

5 Résolution

5.1 Sauvegarde et traitement des fichiers

Comme demandé, le programme intitulé "*solver*" prend en paramètre le nom du fichier contenant la grille de départ puis le résout. Le format demandé pour la grille de départ est celui ci :

```
tebion@tebion-Creator-15-A10SF:~/OCR/Ocr_sudoku/Scripts/Solver$ tree -L 1
.
├── grid_00
├── solver
└── solver.c

0 directories, 3 files
tebion@tebion-Creator-15-A10SF:~/OCR/Ocr_sudoku/Scripts/Solver$ cat grid_00
8.. ... ..
..3 6.. ...
.7. .9. 2..

.5. ..7 ...
... .45 7..
... 1.. .3.

..1 ... .68
..8 5.. .1.
.9. ... 4..
```

FIGURE 15 – grid-00

Les cases vides sont remplacées par des points tandis que les cases pleines sont représentées par leur chiffre respectif. Les colonnes sont représenté par un espace entre deux chiffres ou points et les lignes sont représenté par un saut de ligne. Ainsi il est facile de repérer dans quel

carré nous nous trouvons. Le format de la réponse est exactement du même type. Néanmoins, comme la réponse se doit d'être complète, il n'y a plus de point représentant le 0. En voici un exemple :

```
tebion@tebion-Creator-15-A10SF:~/OCR/Ocr_sudoku/Scripts/Solver$ tree
.
├── grid_00
├── grid_00.res
├── solver
└── solver.c

0 directories, 4 files
tebion@tebion-Creator-15-A10SF:~/OCR/Ocr_sudoku/Scripts/Solver$ cat grid_00.res
812 753 649
943 682 175
675 491 283

154 237 896
369 845 721
287 169 534

521 974 368
438 526 917
796 318 452
tebion@tebion-Creator-15-A10SF:~/OCR/Ocr_sudoku/Scripts/Solver$
```

FIGURE 16 – grid-00.res

De plus il est possible d'ouvrir ses réponses sous n'importe quel éditeur ou bien de les afficher dans le terminal via la commande "*cat*" puis le nom du fichier. Pour en finir avec la sauvegarde des résultats, le fichier comprenant la réponse au sudoku se nommera toujours comme le nom du fichier source en rajoutant *.res* en suffixe.

5.2 Algorithme de résolution

Une fois la grille créée à partir d'un fichier, l'algorithme permettant de résoudre la grille se met en marche. La première étape est de vérifier la validité de la grille. En effet, si la grille donnée est fausse notre programme sera capable de renvoyer un message d'erreur. Une grille est fausse si elle contient par exemple deux fois le même chiffre dans une colonne, dans une ligne ou dans un carré. Arrive ensuite l'algorithme de résolution. Celui-ci part depuis la case en haut à gauche, puis parcourt la totalité de la grille en appliquant ce principe :

- Si la case est déjà prise je passe à la suivante
- Si la case est vide j'essaye de placer un nombre en commençant par **1**. Je vérifie que celui-ci ne se trouve pas déjà dans la colonne, dans la ligne ou bien dans le carré. Sinon je passe au chiffre suivant.

- Je passe ensuite à la case suivante et réitère le principe. Si il est impossible de placer un chiffre entre **1** et **9** alors je reviens à la case précédente place le chiffre suivant.
- Et ainsi de suite jusqu'à arriver à la fin du sudoku !

Une fois arrivé ici, nous rentrons le résultat dans le fichier de sauvegarde en respectant le format demandé. Le programme "*Solver*" s'arrête ici.

6 Suivi du projet

Si nous faisons un point, nous pensons être dans les temps. Certaines fonctionnalités comme le filtre traitant le flou et les impuretés ou bien un programme permettant de rentrer facilement un sudoku nous manque encore. Néanmoins celle-ci ne sont pas requises pour ce premier rendu.

Ce qui nous fait défaut pour ce premier rendu est la découpe des cases, en effet nous sommes capable de détecter la totalité de la grille ainsi que les positions des cases. Cependant, il nous manque encore la fonction permettant de découper l'image grâce à ces coordonnées.

Nous sommes très content d'avoir réussi un pré-traitement de l'image permettant son utilisation pour l'algorithme de **Hough Line** ainsi que celui de **Canny**. Ces deux algorithmes étant indispensable à la reconnaissance de la grille sur une image quelconque. De plus le réseau de neurone est intégralement opérationnel avec une sauvegarde et une lecture des valeurs des poids et des biais. Il nous faudra maintenant l'adapter pour qu'il fonctionne avec la lecture d'image de 28x28 pixels. Pour conclure cette section, nous pensons avoir réussi à faire tout ce qui était nécessaire pour ce premier rendu et nous espérons continuer sur la même lancée pour le rendu final.

7 Conclusion

L'OCR est un projet qui nous tient vraiment à coeur, nous pensons avoir bien réussi la première partie de ce projet et nous avons hâte de continuer. La cohésion de groupe est bonne et nous prenons plaisir à travailler et coopérer ensemble. Nous avons bon espoir de finir le projet à temps et de pouvoir le tester sur une large gamme d'image et de sudoku.