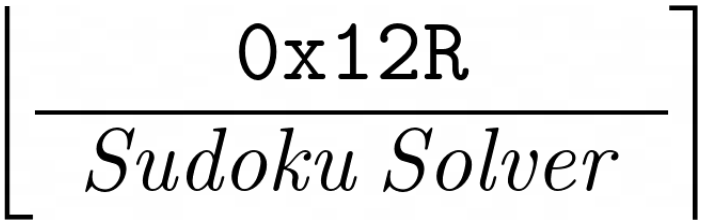
EPITA Rennes

Rapport de soutenance Projet OCR

0x12R



0x12R Team

Paolo Wattebled
Xavier de Place
Arthur Guelennoc
Mathieu Pastre

10 novembre 2022

Table des matières

Introduction

Ce document est le premier rapport de notre projet OCR. Nous allons vous détailler comment nous sommes arrivés à notre état actuel, et les problèmes rencontrés. Nous vous souhaitons une bonne lecture !

Équipe

Voici les membres de notre équipe, la Team 0x12R :

Arthur Guelennoc

Pour ce projet, je me suis chargé de mettre en place le réseau de neurones. J'ai toujours été intéressé par le domaine du *Deep Learning*, et j'avais eu l'occasion d'y mener quelques projets personnels. Cependant, je n'ai jamais réalisé un réseau de neurones sans l'aide des libraires extensives de `Python` avec `TensorFlow` ou `Keras`. Ainsi, ce fût une expérience originale qui m'as permis d'avoir une toute nouvelle perspective sur ce domaine d'intérêt. En utilisant le `C`, j'ai compris le fonctionnement des calculs qui me paraissaient assez abstraits lorsque j'utilisais des architectures pré-fabriquées sans me poser de questions.

Mathieu Pastre

Élève en SPÉ à l'EPITA, je devais m'occuper du traitement d'image. Mon objectif était de faire la rotation de l'image, manuellement et automatiquement, ainsi que de modifier sa taille. Le langage `C` m'a beaucoup impressionné au début de l'année mais je commence à m'y faire petit à petit. Il m'a été difficile de m'investir autant que mes camarades pour ce début de projet. Je compte cependant faire au mieux pour la suite.

Paolo Wattebled

Élève en SPÉ à l'EPITA, je me suis chargé au cours de ce projet du traitement d'image ainsi que la détection de la grille et sa segmentation. Ce projet, m'a apporté énormément de compétences et connaissances en programmation `C`. Malgré la difficulté de prise en main d'un tel langage, je me suis accommodé et appris à l'apprécier malgré son exigence.

Xavier de Place

Je suis en SPÉ R1 à EPITA Rennes cette année. Pour ce projet, je me suis occupé de l'interface graphique et j'étais en support pour le traitement d'image et pour la rédaction de ce rapport. Je n'avais jamais fait de `C` avant ce projet. Je suis content d'avoir pu apprendre ce langage. Il est bien différent de `Python`, mais je trouve que c'est un langage très puissant. J'ai appris beaucoup de choses sur les interfaces, notamment ce qu'est `GTK` et l'utilisation du `CSS` même en dehors des pages web.

Projet

Le concept de notre projet est simple. Nous créons un OCR, un *Optical Character Recognition*. C'est un programme qui prend une image en entrée, et qui en extrait le texte. Nous le faisons spécialement pour les grilles de Sudoku, il contient donc un *solveur* de Sudoku.

Répartition des tâches

Voici un tableau récapitulatif de la répartition des tâches au sein de l'équipe.

	Arthur	Mathieu	Paolo	Xavier
Traitement de l'image				
Chargement de l'image				R
Suppression des couleurs			R	
Prétraitement de l'image		R		
Détection de la grille			R	
Réseau de Neurones				
XOR	R			
Réseau de neurones	R			
Autres				
Interface Graphique				R
Résolveur			R	

1 Traitement de l'image

Le traitement d'image est un axe de travail majeur dans un projet de reconnaissance de caractères. Il fallait donc mettre en place des algorithmes efficaces et optimisés dans le but de détecter la grille du Sudoku ainsi que les chiffres à l'intérieur. Nous vous présenterons ce que nous avons réalisé pour cette première soutenance.

Pour illustrer les différentes étapes, nous allons montrer les différentes étapes sur l'image ci-dessous au fur et à mesure de ce rapport.

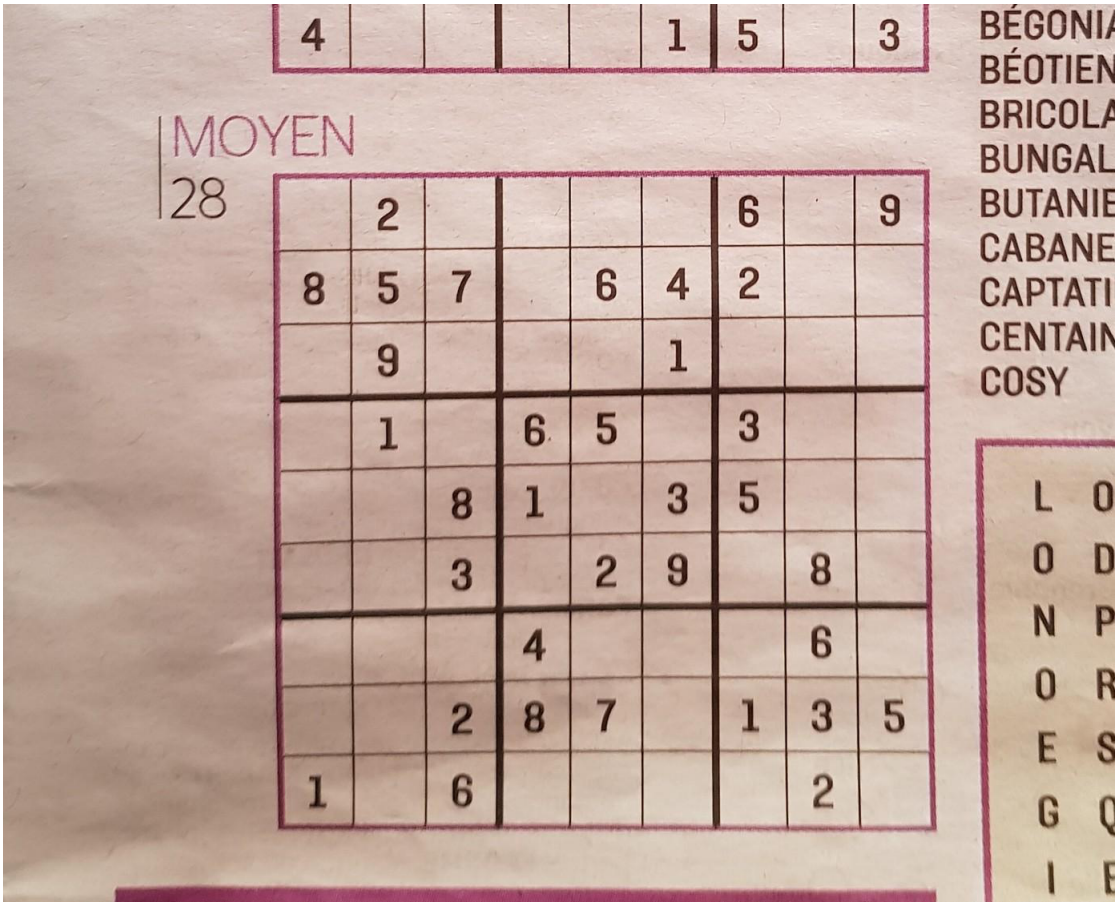


FIGURE 1 – Image originale

1.1 Pré-traitements

1.1.1 Importation d'image

Lors de notre réflexion sur l'architecture globale du projet, nous avons décidé d'utiliser nos propres structures d'images au lieu d'utiliser la librairie disponible SDL2. Nous avons fait ce choix parce qu'utiliser SDL2 demande de créer en amont une architecture très peu modulable. De plus, SDL2 est beaucoup moins optimisée et prend plus de temps à exécuter des opérations sur des images. Donc pour ce qui est du traitement d'image, nous utilisons notre propre structure. Mais pour importer/exporter des images, nous utilisons SDL2. Voici la structure que nous utilisons :

```
1                typedef struct Pixel {
2                    unsigned int r, g, b;
3                } Pixel;
4
5                typedef struct Image {
6                    unsigned int width;
7                    unsigned int height
8                    Pixel **pixels;
9                    char *path;
10               } Image
11
```

1.1.2 Redimensionnement d'image

Après l'importation et la création de notre structure d'image, nous réduisons les dimensions pour devoir traiter moins de données tout en ayant les mêmes résultats qu'avec la taille de base. Pour ce faire, nous utilisons l'*Interpolation du plus proche voisin.*

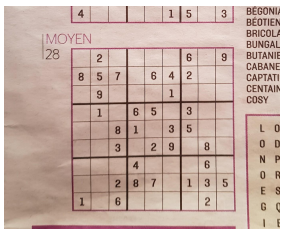


FIGURE 2 – Image Redimensionnée

Pour la suite des illustrations, nous allons garder la version non redimensionnée de l'image, pour mieux voir les changements.

1.1.3 Filtre de gris

Nous réalisons un filtre de gris sur chaque pixel de l'image pour réaliser les algorithmes de traitement d'image. Cela nous permet de normaliser les couleurs, et de nous enlever un paramètre compliqué à gérer. La formule appliquée est :

$$p_{(x,y)} = p_{(x,y)}.r * 0.3 + p_{(x,y)}.g * 0.59 + p_{(x,y)}.b * 0.11$$

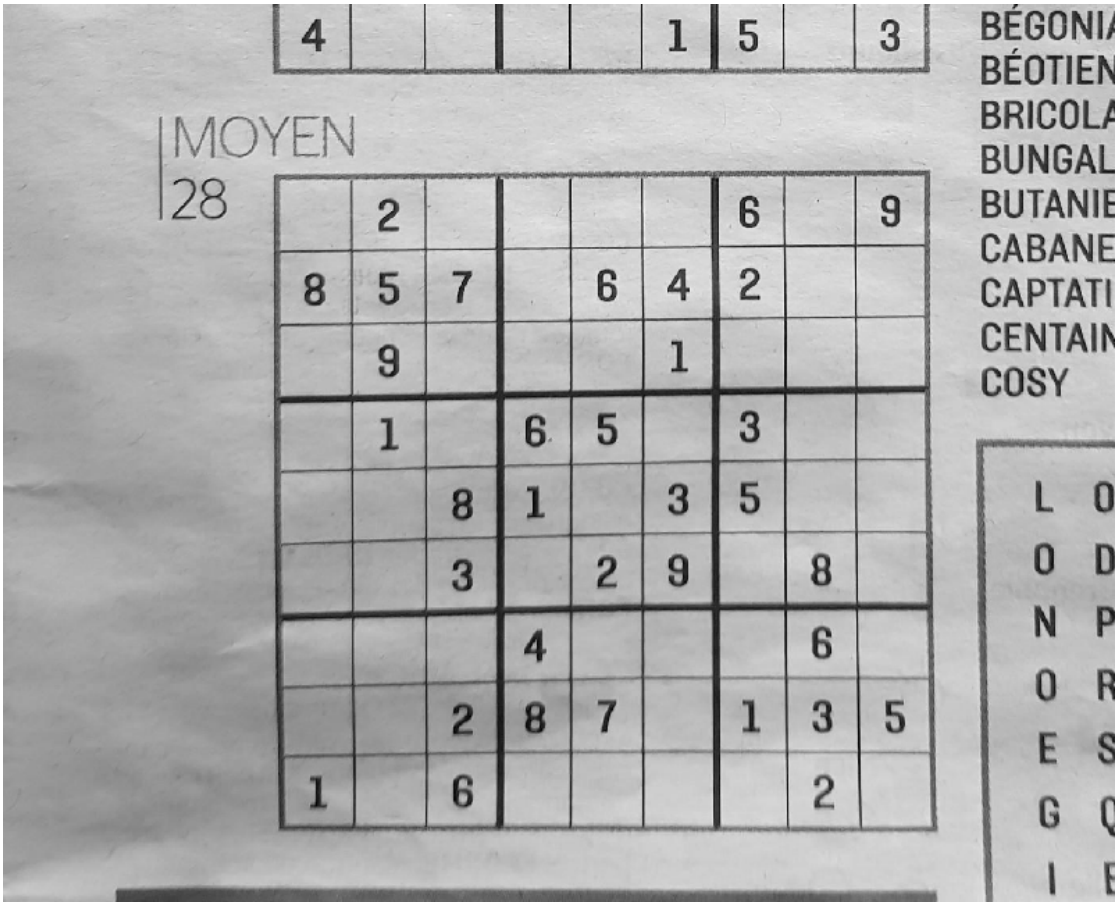


FIGURE 3 – Filtre de gris

1.1.4 Augmentation du contraste

Après avoir réalisé le filtre de gris et que toutes les valeurs R,G,B d'un pixel sont les mêmes, nous augmentons le contraste de notre image pour réduire certains bruits parasites. La logique de l'algorithme pour chaque pixel de notre image est la suivante :

$i = 0 \rightarrow F$

si $p_{(x,y)} \in [i * (255/F), (i+1) * (255/F)]$

$p_{(x,y)} = (i+1) * (255/F)$

Ici, F est le facteur qui permet de faire un contraste plus ou moins élevé. Pour notre projet, il est mit à la valeur de $F = 12$.

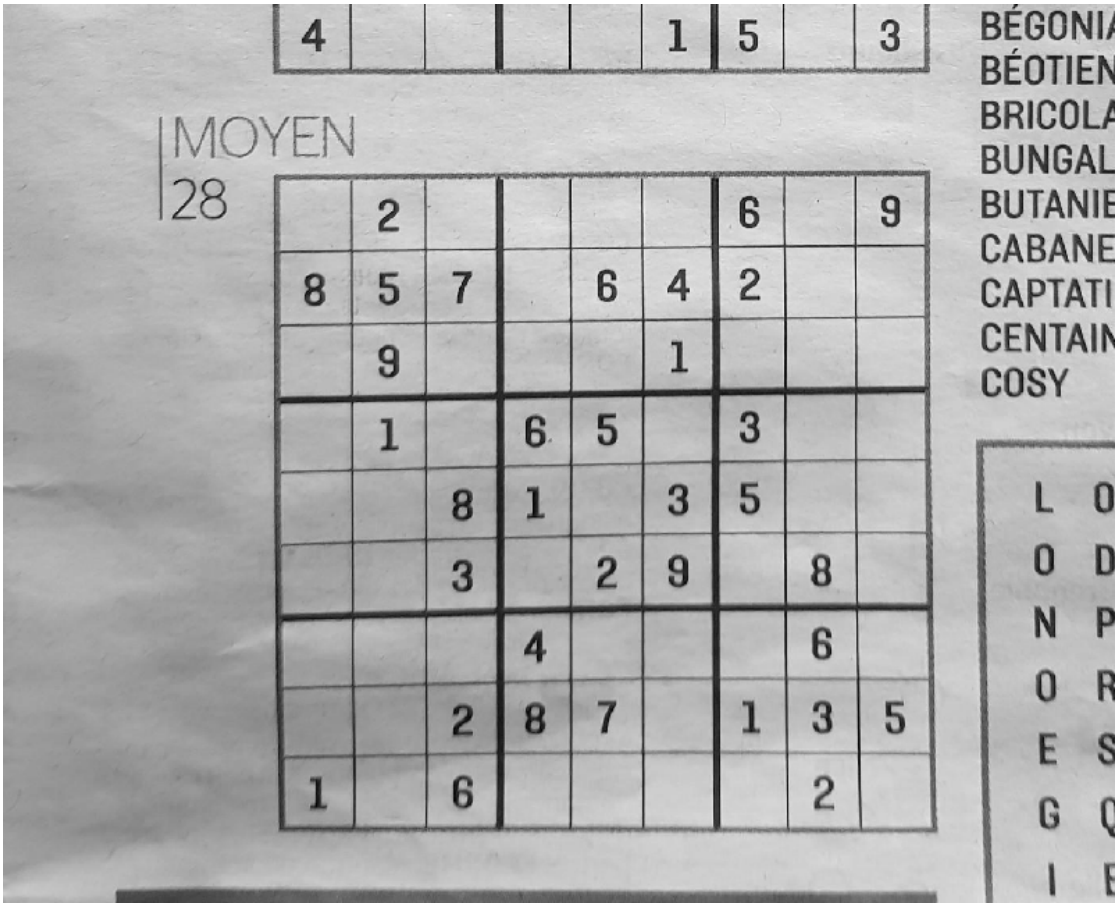


FIGURE 4 – Augmentation du contraste

1.1.5 Normalisation des éclairages

Cette technique nous a permi de réduire encore davantage certains bruits sur notre image. Ici, m est le pixel qui a la plus grande valeur.

$$p_{(x,y)} = 255 - p_{(x,y)}.r * (255/m)$$

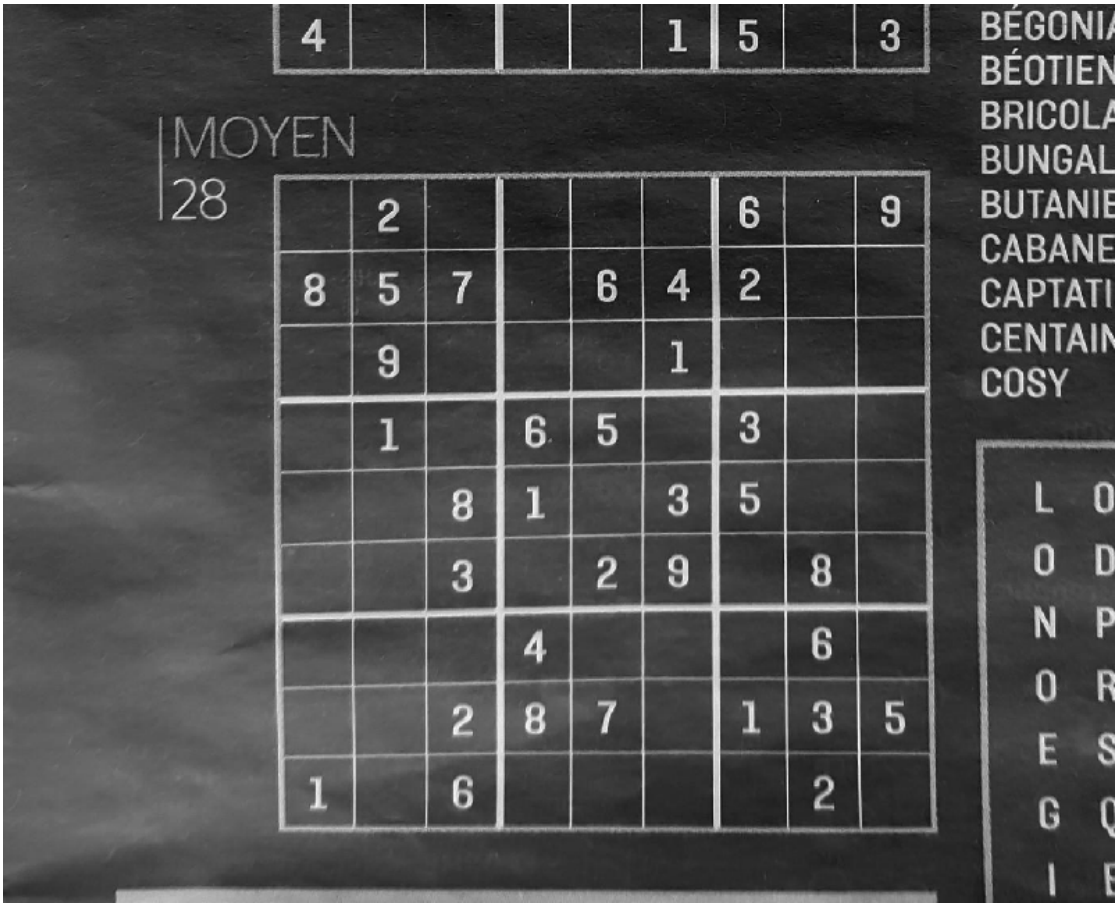


FIGURE 5 – Ajustement des éclairages

1.1.6 Flou Gaussien

Finalement, lors du pré-traitement d'image, nous effectuons un flou gaussien pour réduite les bruits et lisser l'image. L'intensité du flou peut être ajusté selon une variable `size` s. La logique de cet algorithme est de multiplier un pixel par une matrice $G(x, y)$ telle que :

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Avec $\sigma = \frac{s}{2}$ si $\frac{s}{2} > 1$ sinon 1
Enfin,

$$p_{(x,y)} = \sum_{x'=-s}^{s} \sum_{y'=-s}^{s} G(x + x', y + y')$$

FIGURE 6 – Application du flou gaussien

1.2 Binarisation et détection des contours

Afin de détecter la grille, nous devons faire ressortir les lignes principales de notre image. Nous avons testé plusieurs algorithmes différents tels que le ? ? ? ? ? ?

A cet effet, nous utilisons l'algorithme de Canny. Il se décompose en plusieurs parties :

1. Flou Gaussien (réalisé précédemment)
2. Gradient d'intensité (Filtre de Sobel)
3. Direction des contours
4. Suppression des non-maxima
5. Seuillage des contours (Hystérésis)

1.2.1 Gradient d'intensité

L'étape de calcul du gradient détecte l'intensité et la direction des bords en calculant le gradient de l'image à l'aide d'opérateurs de détection des bords. Les bords correspondent à un changement d'intensité des pixels. Pour la détecter, le plus simple est d'appliquer des filtres qui mettent en évidence ce changement d'intensité dans les deux directions : horizontale (`x`) et verticale (`y`).

Lorsque l'image est lissée, les dérivées $\mathbf{G}_x$ et $\mathbf{G}_y$ par rapport à `x` et `y` sont calculées. Cela peut être mis en œuvre en convoluant un pixel $p_{(x,y)}$ avec les noyaux de Sobel $\mathbf{K}_x$ et $\mathbf{K}_y$, respectivement :

$$\mathbf{K}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad \text{et} \quad \mathbf{K}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Ainsi, la valeur du gradient et de l'angle est, où $\mathbf{G}_x$ et $\mathbf{G}_y$ sont les résultat de la convolution avec les masques $\mathbf{K}_y$ et $\mathbf{K}_y$:

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$
$$\theta = \arctan\left(\frac{\mathrm{G}_y}{\mathrm{G}_x}\right)$$

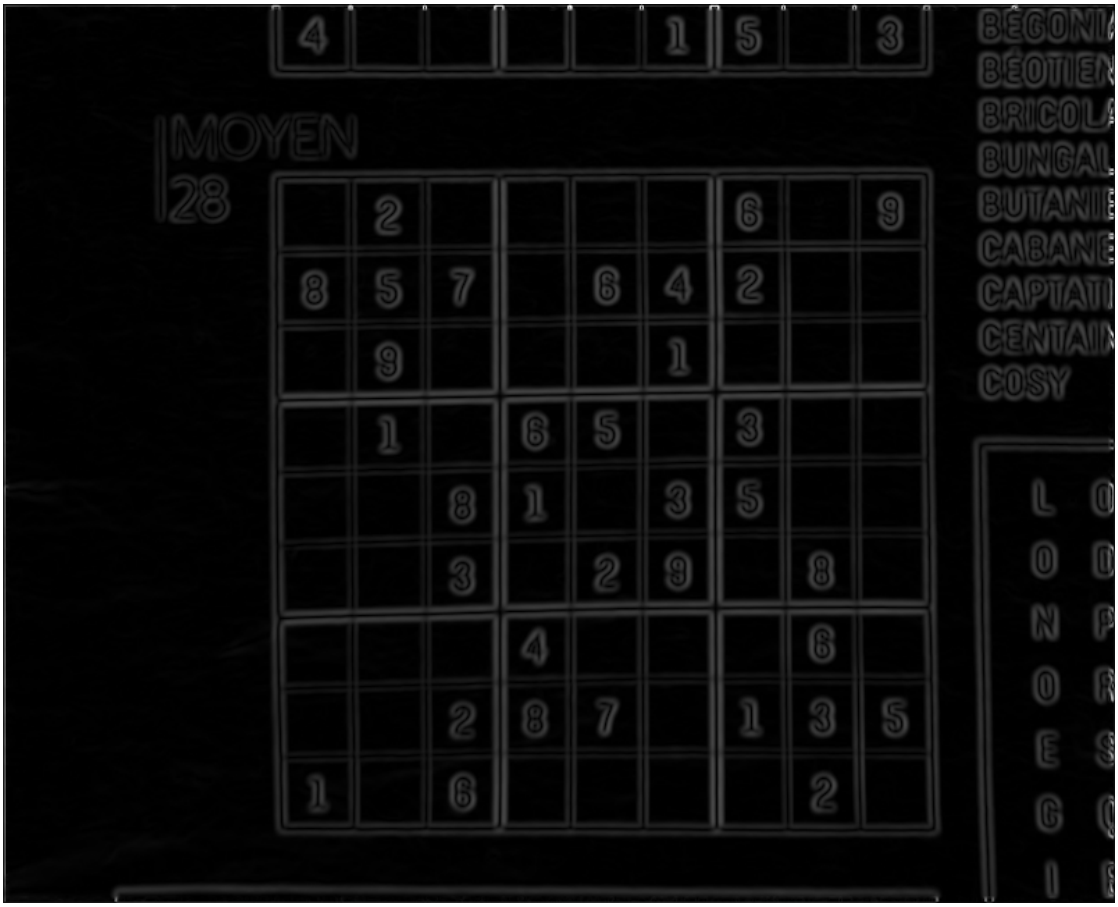


FIGURE 7 – Application du filtre Sobel

1.2.2 Suppression des non-maxima

Pour ce faire, considérons 2 points $P1$ et $P2$. En rapprochant les points $P1$ et $P2$ des pixels voisins par interpolation linéaire, il est possible de comparer les gradients. Si le gradient en (i, j) est supérieur aux gradients en $P1$ et $P2$, on le conserve. Sinon, on l'élimine. On procède ainsi pour tous les pixels bords de l'image, ainsi les bords ne seront large uniquement que d'un pixel.

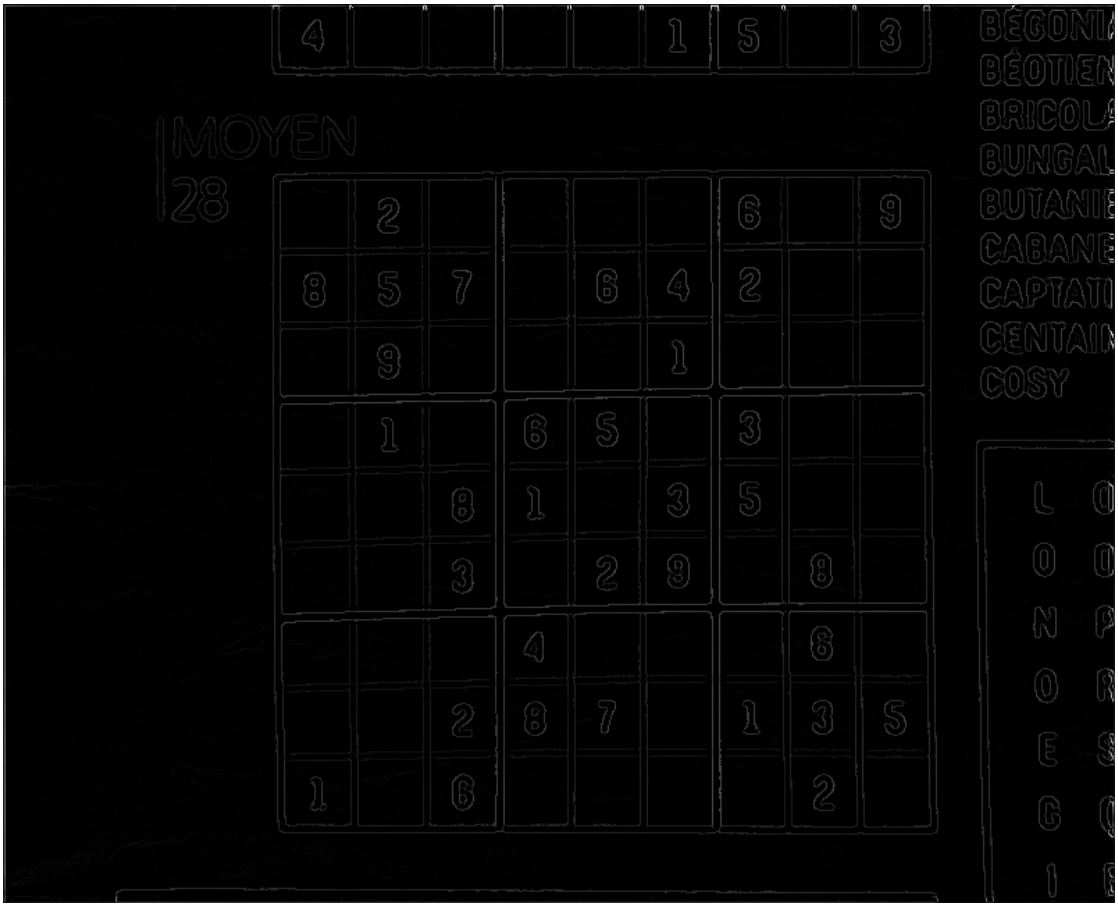


FIGURE 8 – Suppression des non-maxima

1.2.3 Seuillage des contours

La différenciation des contours sur l'image générée se fait par seuillage à hystérésis.

Cela nécessite deux seuils (un haut et un bas), qui seront comparés à l'intensité du gradient de chaque point. Le critère de décision est le suivant. Pour chaque point, si l'intensité de son gradient est :

1. Inférieur au seuil bas, le point est rejeté,
2. Supérieur au seuil haut, le point est accepté comme formant un contour,
3. Entre le seuil bas et le seuil haut, le point est accepté s'il est connecté à un point déjà accepté.

Une fois ceci réalisé, l'image obtenue est binaire avec d'une part les pixels appartenant aux contours et d'autre part, les autres.

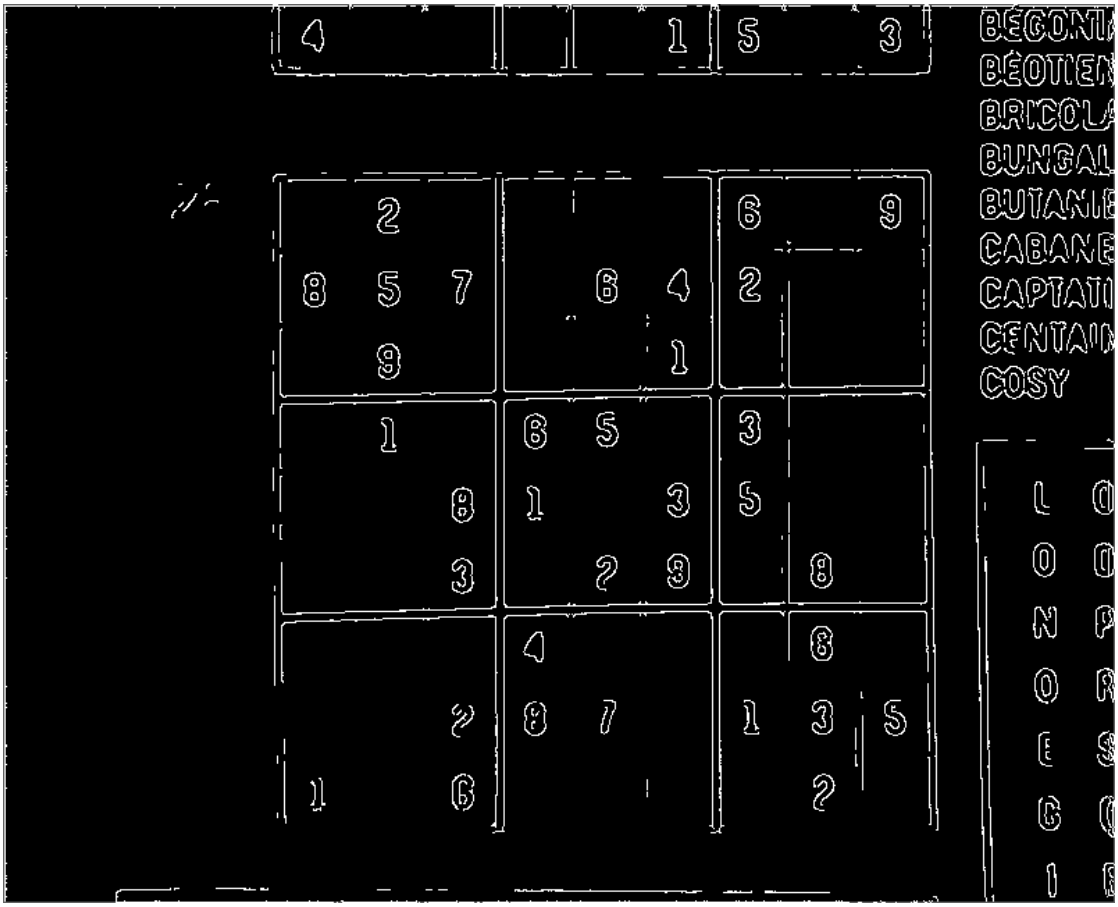


FIGURE 9 – Seuillage à hystérésis de l'image

1.3 Détection de la grille

Pour réaliser la détection de la grille, nous avons essayé 2 méthodes : *Hough Transform* et *Blob detection*. D'un côté, *Hough transform* détecte les lignes sur une image. De l'autre, *Blob detection* crée et détecte le plus gros amas de pixels blancs sur l'image. Ainsi, de part nos résultats, nous avons opté pour la seconde option, étant donné que ses résultats sont plus fiables et plus faciles à manipuler. Voici en image ce que cela donne :

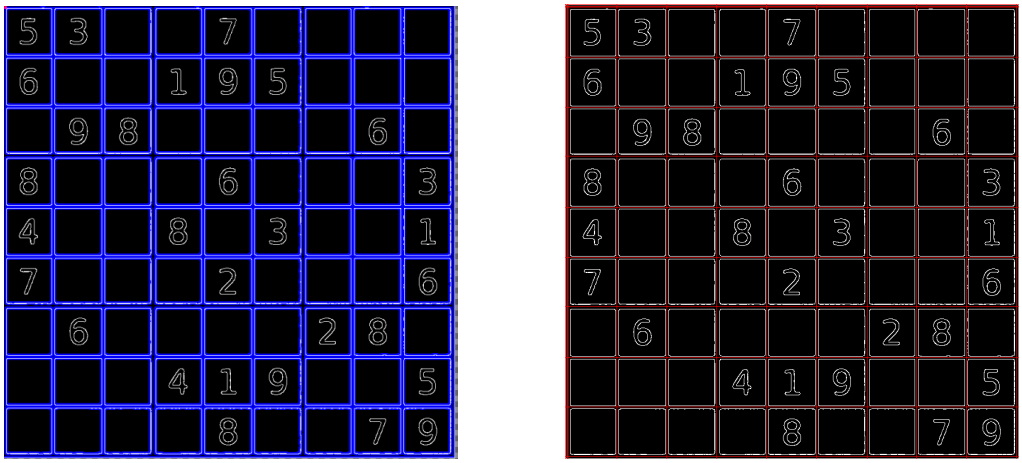


FIGURE 10 – Blob detection et Hough transform

1.3.1 Rotation d'Image Manuelle

Pour appliquer correctement le réseau de neurones, il a fallu s'assurer que l'image soit bien orientée. Pour cela, nous avons fait un premier prototype de rotation. Celui-ci prenait comme valeur l'image que nous voulions traiter ainsi que l'angle à manipuler. Il a ensuite fallu lire les valeurs de chaque pixel de notre image source et faire une rotation centrale. Nous appliquions ensuite la formules suivantes pour trouver x' et y' les deux nouvelles coordonnées du point en question :

$$x' = (x - x_O) * cos(\alpha) + (y - y_O) * sin(\alpha) + x_O$$
$$y' = (x - xO) * cos(\alpha) + (y - y_[O]) * sin(\alpha) + x_O$$

Ici, les points x_0 et y_0 sont les coordonnées du point central de l'image, que nous retrouvons en fonction de la taille de celle-ci. Et x et y les coordonnées de notre pixel actuel. Nous récupérons ensuite les valeurs RGB du pixel (x,y) pour les mettre dans le pixel (x',y').

1.3.2 Transformation Homographique

Le but de la transformation homographique est de faire une rotation de l'image tout en modifiant sa taille. Cette fonction lit en entrée les coordonnées de quatre points, ce seront celles des coins du sudoku. Nous nous sommes servi pour cela de la théorie homographique qui se base sur cette équation :
Soient $(h, a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2) \in R^9$

$$\begin{bmatrix} h \times x' \\ h \times y' \\ h \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \implies \begin{bmatrix} h \times x' \\ h \times y' \\ h \end{bmatrix} = \begin{bmatrix} a_1 \times x + a_2 \times y + a_3 \\ b_1 \times x + b_2 \times y + b_3 \\ c_1 \times x + c_2 \times y + 1 \end{bmatrix}$$

$$\implies \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \frac{a_1 \times x + a_2 \times y + a_3}{c_1 \times x + c_2 \times y + 1} \\ \frac{b_1 \times x + b_2 \times y + b_3}{c_1 \times x + c_2 \times y + 1} \end{bmatrix}$$

$$\implies \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -x \times x' & -y \times x' \\ 0 & 0 & 0 & x & y & 1 & -x \times y' & -y \times y' \end{bmatrix} \times \vec{d}$$

où $\vec{d} = \begin{bmatrix} a_1 & a_2 & a_3 & b_1 & b_2 & b_3 & c_1 & c_2 \end{bmatrix}$

Ces calculs permettent de trouver la coordonnée logique que devrait avoir chaque pixel de coordonnée (x,y). Faisons de même pour nos 3 autres pixels, le tout dans une seule matrice :

$$\vec{e} = \mathbf{K} \times \vec{d} \implies \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1 \times x'_1 & -y_1 \times x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1 \times y'_1 & -y_1 \times y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2 \times x'_2 & -y_2 \times x'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2 \times y'_2 & -y_2 \times y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3 \times x'_3 & -y_3 \times x'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3 \times y'_3 & -y_3 \times y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4 \times x'_4 & -y_4 \times x'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4 \times y'_4 & -y_4 \times y'_4 \end{bmatrix} \times \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ b_1 \\ b_2 \\ b_3 \\ c_1 \\ c_2 \end{bmatrix}$$

Pour trouver les valeurs de d il nous suffit de résoudre l'équation suivante :

$$\vec{d} = \mathbf{K}^{-1} \times \vec{e}$$

Nous pouvons alors garder ces constantes pour trouver les nouvelles coordonnées de chaque pixels de l'image. Les pixels qui débordent de la taille demandée en paramètre ne sont pas pris en compte.

2 Réseau de Neurones

2.1 XOR

Avant de commencer à travailler sur le réseau de neurones final, il fallait réaliser une porte logique XOR en guise de preuve de concept. Ceci fût assez simple à mettre en place. Composé de 3 couches, le réseau passe l'information initiale à la couche cachée puis, ce dernier, à la couche finale.

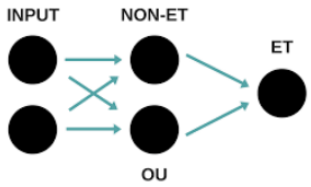


FIGURE 11 – Concept du réseau de neurones XOR

A chaque passage, les valeurs des neurones impliqués sont modifiées par les poids. Nous utilisons la même fonction d'activation, `sigmoid`, pour normaliser les valeurs de la couche cachée entre 0 et 1, ainsi que celle de la couche finale. Enfin, avec notre valeur finale obtenue, nous appliquons la rétropropagation afin d'obtenir un résultat qui se rapproche un peu plus de la valeur désirée, en ajustant les poids.

Toutefois, notre mise en oeuvre laissais à désirer. Avec un réseau de neurones aussi simple, nous nous ne sommes pas posés la question d'utiliser des structures, nous appuyant plutôt sur des tableaux. Étant donné que les valeurs d'entrée et la structure étaient assez rudimentaires, notre réseau mettait très peu de temps à compiler et nous affichais le résultat désiré dans 100% des cas.

```
antithetical@Chunchunmaru:~/EPITA/OCR/neural_network/xor$ ./a.out
Output    Desired output
0.000000  0.000000
1.000000  1.000000
1.000000  1.000000
0.000000  0.000000

Input layer weights
0.972751
9.011277
0.972751
9.011372

Hidden layer weights
-61.300918
49.214821
```

FIGURE 12 – Sortie du XOR

2.2 Digit Recognition

Le réseau de neurones que nous utilisons pour identifier correctement les chiffres sur le sudoku a donc suivi le même principe. L'implémentation, cependant, fut totalement différente. En utilisant des structures pour mettre en place le réseau, les couches et les neurones, nous pouvions facilement manipuler nos paramètres d'entrée, nommés "hyperparamètres".

Cependant, certains détails ont effectivement changé : nous utilisons dorénavant la fonction d'activation `reLU`, ainsi que la fonction `sigmoid`, seulement pour la dernière couche de neurones. La fonction `reLU` nous a permis d'avoir des meilleurs performances, ce qui fut essentiel, étant donné l'envergure du réseau.

La performance ainsi que les résultats du réseau varient ; cela en fonction du nombres de couches cachées, du nombre de neurones par couche cachée mais aussi en fonction du taux d'apprentissage. Ce dernier influence la capacité d'apprentissage du réseau : un taux d'apprentissage trop élevé ou trop bas s'avère néfaste pour le taux de réussite du réseau. Ainsi, ce paramètre doit être minutieusement choisi. En effet, l'efficacité du taux d'apprentissage dépend également des autres hyperparamètres.

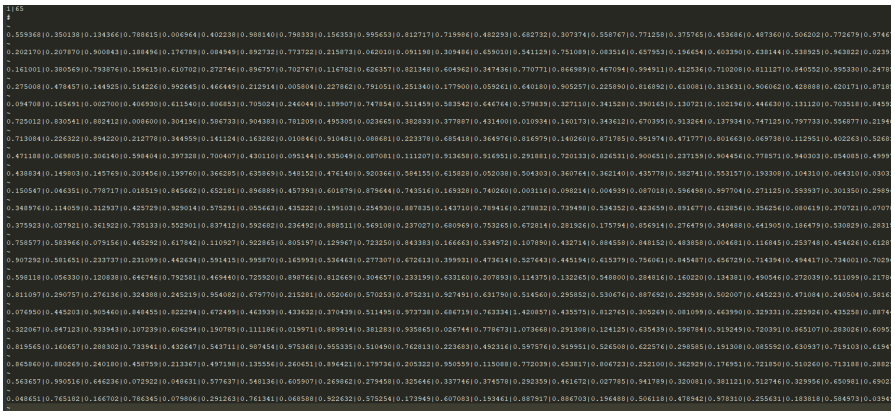
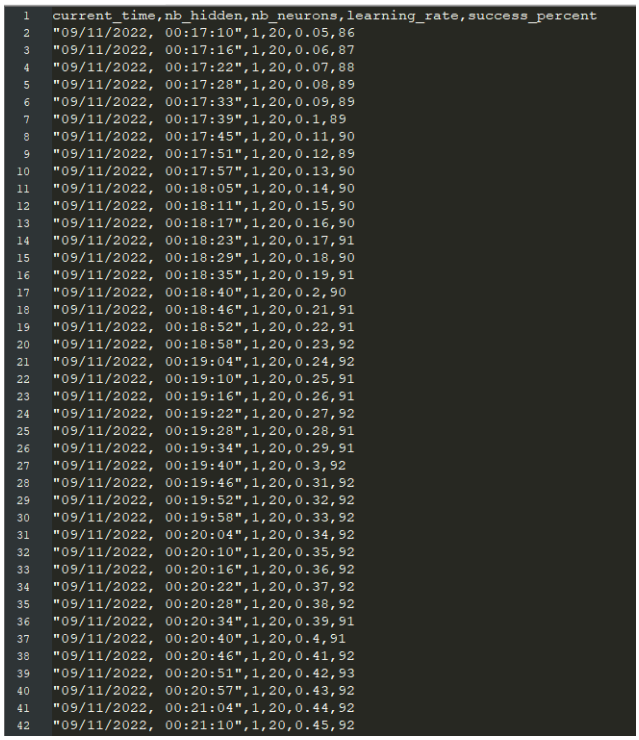


FIGURE 13 – Fichier poids

A cette fin, nous avons créé un fichier `Python` qui compile le réseau de neurones, tout en ajustant les hyperparamètres, afin d'obtenir le meilleur résultat possible. Cela nécessitait donc de sauvegarder les hyperparamètres ainsi que les poids, qui régissent la valeur de sortie du réseau, dans un fichier texte. Ceci fut implémenté, ainsi que les outils nécessaires pour charger ces données d'un ficher dans un nouveau réseau de neurones. De sorte, le fichier `Python` pouvait donc sauvegarder la meilleure itération du réseau de neurones pour que nous puissions le tester et comparer a posteriori.



```
current_time,nb_hidden,nb_neurons,learning_rate,success_percent
"09/11/2022, 00:17:10",1,20,0.05,86
"09/11/2022, 00:17:16",1,20,0.06,87
"09/11/2022, 00:17:22",1,20,0.07,88
"09/11/2022, 00:17:28",1,20,0.08,89
"09/11/2022, 00:17:33",1,20,0.09,89
"09/11/2022, 00:17:39",1,20,0.1,89
"09/11/2022, 00:17:45",1,20,0.11,90
"09/11/2022, 00:17:51",1,20,0.12,89
"09/11/2022, 00:17:57",1,20,0.13,90
"09/11/2022, 00:18:05",1,20,0.14,90
"09/11/2022, 00:18:11",1,20,0.15,90
"09/11/2022, 00:18:17",1,20,0.16,90
"09/11/2022, 00:18:23",1,20,0.17,91
"09/11/2022, 00:18:29",1,20,0.18,90
"09/11/2022, 00:18:35",1,20,0.19,91
"09/11/2022, 00:18:40",1,20,0.2,90
"09/11/2022, 00:18:46",1,20,0.21,91
"09/11/2022, 00:18:52",1,20,0.22,91
"09/11/2022, 00:18:58",1,20,0.23,92
"09/11/2022, 00:19:04",1,20,0.24,92
"09/11/2022, 00:19:10",1,20,0.25,91
"09/11/2022, 00:19:16",1,20,0.26,91
"09/11/2022, 00:19:22",1,20,0.27,92
"09/11/2022, 00:19:28",1,20,0.28,91
"09/11/2022, 00:19:34",1,20,0.29,91
"09/11/2022, 00:19:40",1,20,0.3,92
"09/11/2022, 00:19:46",1,20,0.31,92
"09/11/2022, 00:19:52",1,20,0.32,92
"09/11/2022, 00:19:58",1,20,0.33,92
"09/11/2022, 00:20:04",1,20,0.34,92
"09/11/2022, 00:20:10",1,20,0.35,92
"09/11/2022, 00:20:16",1,20,0.36,92
"09/11/2022, 00:20:22",1,20,0.37,92
"09/11/2022, 00:20:28",1,20,0.38,92
"09/11/2022, 00:20:34",1,20,0.39,91
"09/11/2022, 00:20:40",1,20,0.4,91
"09/11/2022, 00:20:46",1,20,0.41,92
"09/11/2022, 00:20:51",1,20,0.42,93
"09/11/2022, 00:20:57",1,20,0.43,92
"09/11/2022, 00:21:04",1,20,0.44,92
"09/11/2022, 00:21:10",1,20,0.45,92
```

FIGURE 14 – Fichier CSV

2.2.1 Les bases de données

Pour le moment, nous avons utilisé la base de donnée du *MNIST*, regroupant plus de 60,000 images de chiffres (en format 28x28) manuscrites. Toutefois, dans les semaines à venir, nous allons nous pencher sur d'autres bases de données afin de comparer leurs performances.

3 Résolution du sudoku

3.1 Importation

En ce qui concerne la résolution du Sudoku, nous devons respecter un format de fichier pour simplifier la lecture et l'écriture de la grille. Ainsi, nous lisons le fichier et créons une matrice d'entier de 9 par 9 pour l'envoyer dans notre algorithme de résolution, le backtracking ou retour sur trace.

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

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

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

FIGURE 15 – Format à respecter

3.2 Backtracking

Le backtracking est un algorithme récursif qui parcourt toutes les valeurs possible des cases jusqu'à trouver si le Sudoku est résolu. Les cas d'arrêts sont soit nous trouvons que le sudoku est résolu. Soit qu'il présente une erreur. Dans ce cas, nous retournons un appel en arrière. Cette image permet d'illustrer les appels récursif réalisé par cet algorithme jusqu'à trouver une solution au Sudoku.

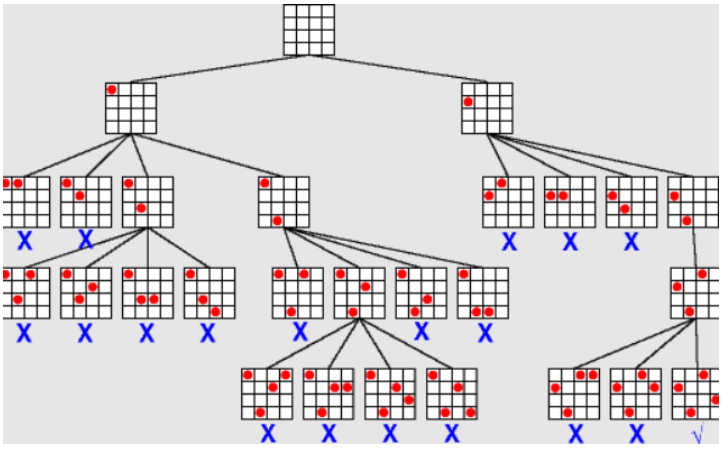
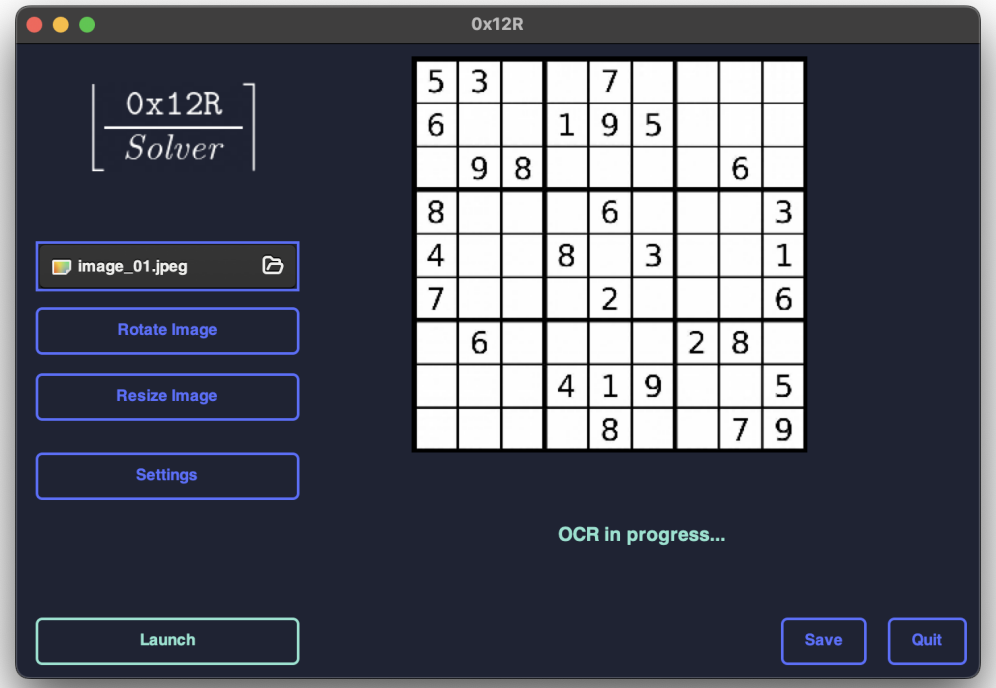


FIGURE 16 – Illustration du backtracking

4 Interface Graphique

Pour créer l'interface graphique, nous avons choisi le package GTK3. Il est gratuit et il est la base de beaucoup d'applications Linux se basant sur l'environnement de bureau *GNOME* comme GTK3. Cela implique qu'il soit déjà installé sur la plus part des systèmes Linux, même s'ils ne sont pas sous environnement *GNOME*. Nous avons utilisé le logiciel *Glade* pour générer un fichier de configuration de notre interface. Nous avons ensuite créé une feuille de style *CSS*, qui contient les couleurs et toutes les caractéristiques visuelles de notre interface. Nous nous retrouvons alors dans une configuration type web. D'un côté, notre fichier `.xml` généré par *Glade*, contenant chaque élément de notre interface de façon sérialisée, et de l'autre notre fichier `.css` constitué de propriétés graphiques pour chacun des éléments. Avec ces deux fichiers, nous avons codé une sorte de "moteur graphique" qui permet de parser notre fichier de configuration, d'en afficher les différents éléments, puis d'y appliquer le style défini. Pour notre design, nous avons décidé de faire quelque chose de simple et d'intuitif. Nous avons réduit le nombre de boutons au strict minimum, et ajouté une zone qui permet d'afficher l'état d'avancement du programme.

Voici une capture d'écran de notre interface :



Conclusion

En conclusion, nous sommes très contents du travail effectué pour cette première soutenance. Nous avons pu mettre en place le traitement quasiment complet de l'image, un réseau de neurones capable de reconnaître des chiffres manuscrits. Nous avons enfin pu créer une interface graphique qui permet d'utiliser notre OCR. Nous allons finir notre projet de notre côté et nous serons ravis de vous présenter notre travail lors de la soutenance finale !

Références

Packages

- `maths.h`
- SDL2 (`https://www.libsdl.org/`)
- SDL2_Image (`https://www.libsdl.org/`)
- GTK3 (`https://www.gtk.org/`)

Outils

- *Vim* (`https://www.vim.org/`)
- *gcc* (`https://gcc.gnu.org/`)
- *Visual Studio Code* (`https://code.visualstudio.com/`)
- *Glade* (`https://glade.gnome.org/`)

Documentation

- `stackoverflow.com`
- `https://magpi.raspberrypi.com/books/c-gui-programming`
- `https://youtube.com/channel/UC5oHS9h-prWeBrBNzXm8rYA`