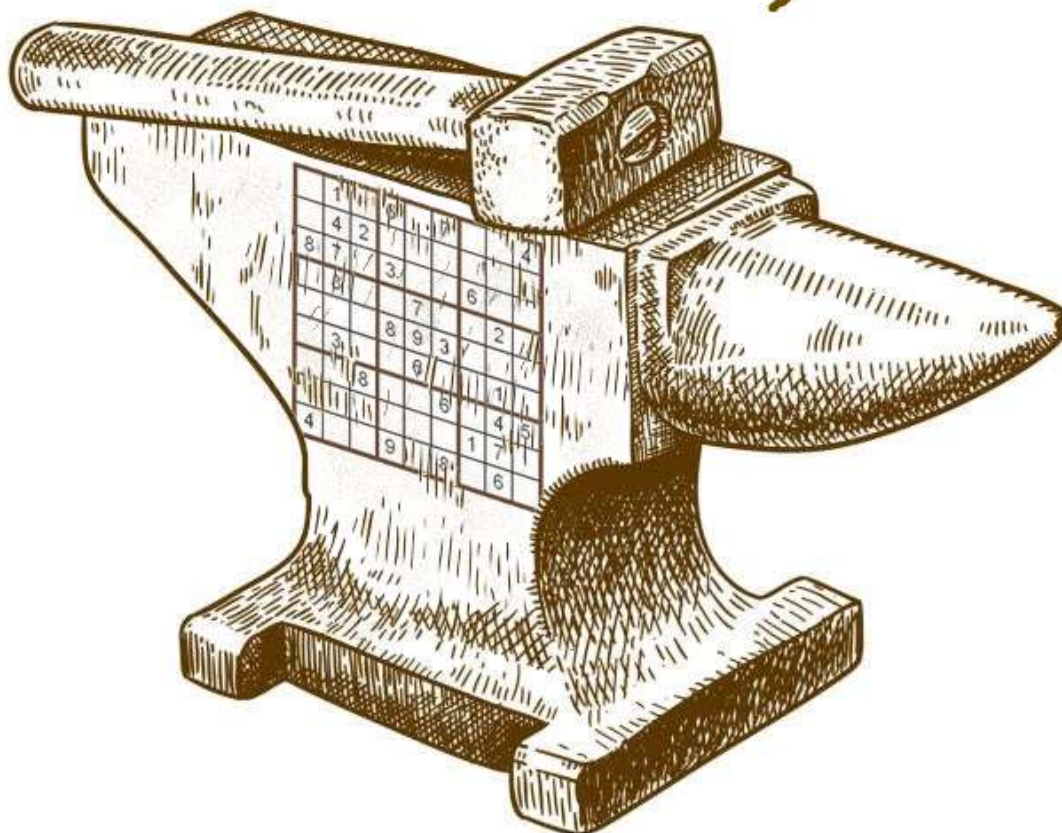


OCR - Soutenance 2

Décembre 2023

# *Sudoku Forgers*



Victor Gardère, Thomas MORIN, Brian PERRET, Ronan Leboucher

## Table des matières

<b>1</b>	<b>Groupe</b>	<b>4</b>
1.1	Présentation du groupe . . . . .	4
1.2	Répartition des charges . . . . .	4
<b>2</b>	<b>Prétraitement</b>	<b>4</b>
2.1	Les filtres appliqués . . . . .	4
2.1.1	Nuance de gris . . . . .	4
2.1.2	Contraste . . . . .	5
2.1.3	Normalisation de l'éclairage . . . . .	6
2.1.4	Médian . . . . .	6
2.1.5	Flou Gaussien . . . . .	7
2.2	Binarisation . . . . .	8
2.2.1	Méthode de Otsu . . . . .	8
2.2.2	Seuil adaptatif . . . . .	10
2.3	Un autre filtre . . . . .	11
2.3.1	Morphologie . . . . .	11
2.4	Avant, Après . . . . .	13
2.5	Rotation de l'image . . . . .	13
2.6	Homographie . . . . .	14
2.7	Interpolation bilinéaire . . . . .	16
2.8	Détection de la grille . . . . .	17
2.9	Détection des coins de la grille . . . . .	19
2.10	Vérification du Blob . . . . .	20
2.11	Détection des cases de la grille . . . . .	20
2.12	Extraction des chiffres présents dans les cases . . . . .	20
<b>3</b>	<b>Solveur de sudoku</b>	<b>21</b>
3.1	Validité de la grille . . . . .	21
3.2	Résolution de la grille . . . . .	21
3.3	Affichage de la grille résolue . . . . .	21
3.4	Fichier du solver . . . . .	22
<b>4</b>	<b>Réseau de neurones</b>	<b>23</b>
4.1	Jeu de donnée . . . . .	23
4.2	Configuration de réseau . . . . .	24
4.3	Propagation avant . . . . .	24
4.4	Apprentissage du réseau . . . . .	25
4.5	Paramètres d'apprentissages du réseau de neurones . . . . .	27
4.5.1	Taux d'apprentissage . . . . .	27
4.5.2	Mini-batch . . . . .	27
4.5.3	Epoch . . . . .	28
4.6	Performance du réseau de neurones . . . . .	29
4.7	Sauvegarde du réseau de neurones . . . . .	30
4.8	Amélioration du réseau . . . . .	31

---

<b>5</b>	<b>Interface graphique</b>	<b>31</b>
5.1	Chargement d'une image . . . . .	32
5.2	Rotation manuelle . . . . .	34
5.3	Analyse et traitement de l'image . . . . .	35
5.4	Modification des chiffres reconnus . . . . .	36
5.5	Grille résolue . . . . .	37
5.6	Paramètres . . . . .	38
<b>6</b>	<b>Site web</b>	<b>39</b>
<b>7</b>	<b>Conclusion du projet</b>	<b>40</b>
7.1	Objectifs précédents . . . . .	40
7.2	Conclusion générale . . . . .	40

# 1 Groupe

## 1.1 Présentation du groupe

En charge du précieux projet que représente cet OCR, notre groupe nommé Sudoku Forgers est composée de : Brian Perret (chef de groupe), Thomas Morin, Ronan Leboucher et Victor Gardère.

## 1.2 Répartition des charges

<b>Charges</b>	<b>Membres</b>	<b>Victor</b>	<b>Thomas</b>	<b>Brian</b>	<b>Ronan</b>
<b>Traitement de l'image</b>					
Binarisation				X	
Détection de la grille	X				X
Rotation manuelle			X		
Rotation automatique			X		
Détection des cases	X	X			
Correction de la perspective			X		
Nettoyage des cases	X				
<b>Reconnaissance des chiffres</b>					
Dataset				X	
Réseau de neurones				X	
<b>Solver de sudoku</b>					
Solver				X	
Reconstruction de la grille				X	
<b>Autres</b>					
Site internet	X				
GUI (backend)				X	
GUI (frontend)			X	X	X
Architecture				X	
Analyse syntaxique				X	

# 2 Prétraitement

## 2.1 Les filtres appliqués

### 2.1.1 Nuance de gris

La première opération que l'on fait sur notre image originelle est le passage en nuance gris. Cette opération permet ainsi de se passer des informations de couleurs qui, dans le cadre de la reconnaissance de chiffre, ne sont pas quelque chose d'utile. En outre, cela permet de grandement réduire la complexité des futures opérations.

Il existe de nombreuses formules, s'adaptant à divers problèmes, tels que :

$$Gris = (0,299 \times Rouge) + (0,587 \times Vert) + (0,114 \times Bleu)$$

Or, cette formule est mise en relation avec la vision humaine et donc n'est pas la plus efficace pour garder un maximum d'informations utiles pour le traitement d'image.

C'est pour cela qu'on utilise plutôt celle-ci :

$$Gris = \left( \frac{1}{3} \times Rouge \right) + \left( \frac{1}{3} \times Vert \right) + \left( \frac{1}{3} \times Bleu \right)$$

Cela nous donne ainsi les résultats suivants :

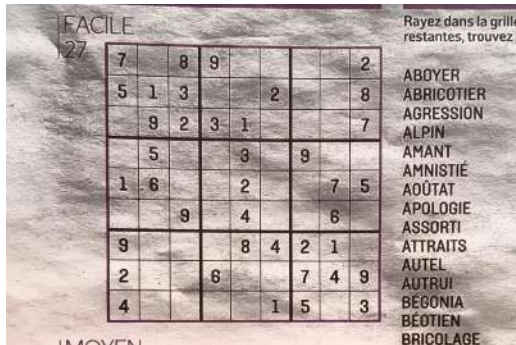


FIGURE 1 – Image originelle

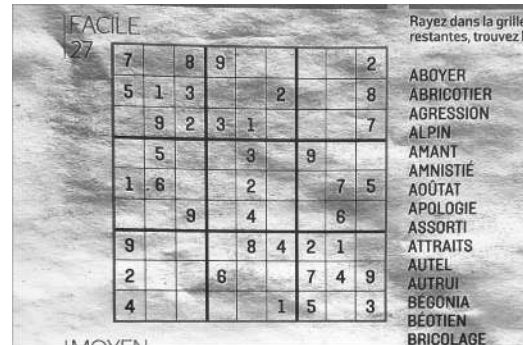


FIGURE 2 – Nuance de gris

### 2.1.2 Contraste

La prochaine étape du processus est l'augmentation des contrastes de l'image. Cela a pour objectif d'augmenter la différence de gris sur l'image et ainsi mieux faire ressortir les traits.

Cela nous donne ainsi les résultats suivants :

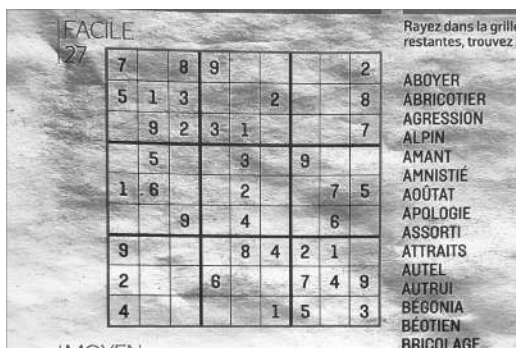


FIGURE 3 – Nuance de gris

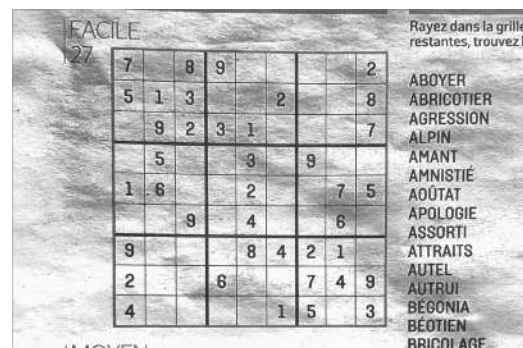


FIGURE 4 – Augmentation du contraste

### 2.1.3 Normalisation de l'éclairage

A la suite de cela, nous procédons à la normalisation de l'éclairage de l'image. Cette correction permet de supprimer les variations d'éclairage entre deux endroits de l'image. Cela permet ainsi de rendre les caractères plus distincts les uns des autres.

Pour cette opération, nous utilisons la formule ci-dessous :

$$P_{n_i} = \frac{Pixel_i}{P_{max} \times 255}$$

$$P \in [0;255];$$

$P_{max}$  : Valeur max de l'image

$P_n$  : Pixel normalisé

$P_i$  : Pixel à l'index i

Cela nous donne ainsi les résultats suivants :

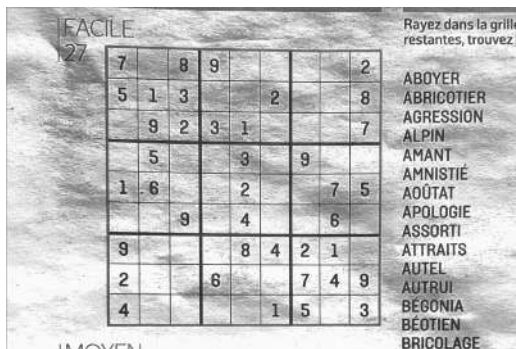


FIGURE 5 – Augmentation du contraste



FIGURE 6 – Normalisation de l'éclairage

### 2.1.4 Médian

Le filtre médian, est un filtre linéaire qui permet de rendre l'image moins bruitée tout en préservant les contours. Pour cela, il atténue les variations soudaines de luminosité à des endroits très localisés.

Voici un exemple de fonction du filtre médian : Soit la matrice des pixels voisins du pixel de valeur n (ici 111)

$$\begin{pmatrix} 5 & 6 & 7 \\ 6 & 111 & 8 \\ 7 & 8 & 9 \end{pmatrix}$$

Ensuite, on trie par ordre croissant les valeurs dans un tableau, tel que :

$$(5 \ 6 \ 6 \ 7 \ 7 \ 8 \ 8 \ 9 \ 111)$$

On prend la valeur médiane de ce tableau, ici 7 Et ainsi, cela nous donne :

$$\begin{pmatrix} 5 & 6 & 7 \\ 6 & 7 & 8 \\ 7 & 8 & 9 \end{pmatrix}$$

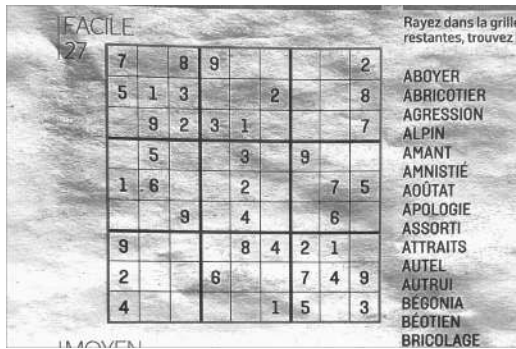


FIGURE 7 – Normalisation de l'éclairage



FIGURE 8 – Filtre médian

### 2.1.5 Flou Gaussien

L'étape suivante est de flouter très légèrement l'image pour enlever les minuscules détails de l'image qui ne nous intéressent pas. Grâce à cela, ce filtre permet de réduire le bruit et de lisser l'image.

Son principe de fonctionnement est le suivant :



$$I_{\text{flou}}(x, y) = \sum_i \sum_j I(x - i, y - j) \cdot G(i, j)$$

$I_{\text{flou}}(x, y)$  : est la valeur du pixel dans l'image floutée à la position.  $(x, y)$

$I(x - i, y - j)$  : est la valeur du pixel dans l'image d'origine à la position.  $(x - i, y - j)$

$G(i, j)$  : est la valeur du noyau Gaussien à la position.  $(i, j)$

Et la formule du noyau Gaussien :

$$G(i, j) = \frac{1}{2\pi\sigma^2} e^{-(i^2+j^2)/(2\sigma^2)}$$

$i$  et  $j$  : parcourent les coordonnées du noyau Gaussien.

$\sigma$  : est l'écart-type du noyau Gaussien, qui contrôle la quantité de flou.

Un écart-type plus grand entraîne un flou plus important.

## 2.2 Binarisation

L'étape de la binarisation, est l'étape la plus importante parmi toutes ces opérations. En effet, c'est elle qui va dire si le pixel devient un pixel blanc ou un pixel noir.

Pour cela, nous utilisons un seuil tel que :

$$\begin{cases} \text{Valeur du pixel} < \text{seuil} & \Rightarrow \text{Blanc} \\ \text{Valeur du pixel} \geq \text{seuil} & \Rightarrow \text{Noir} \end{cases}$$

Ainsi, tout l'enjeu, est de trouver le meilleur seuil pour ne garder que les pixels utiles.

### 2.2.1 Méthode de Otsu

Le premier type d'algorithme, est une méthode de seuillage dite "global". Cet algorithme classe les pixels parmi deux types :

- Les pixels au premier-plan.
- Les pixels à l'arrière-plan.

C'est-à-dire qu'elle ne va effectuer qu'une seule mesure globale de l'image pour déterminer le seuil optimal de l'image et ainsi déterminer dans quelle classe se situe un pixel.

Pour déterminer le seuil optimal, on commence par faire l'histogramme des valeurs des pixels de l'image, tel que :

Ensuite, il suffit de trouver le meilleur emplacement entre deux pics de l'image, qui séparera au mieux l'image en deux classes.

La formule pour trouver le seuil optimal est :

$$\sigma_w^2(t) = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t)$$



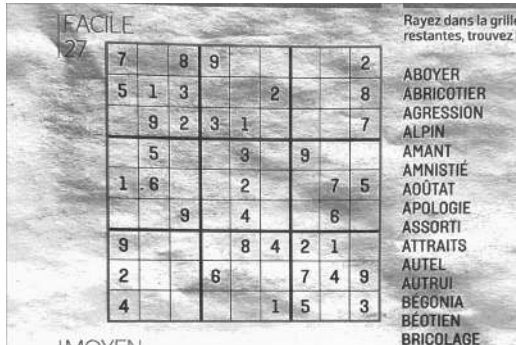


FIGURE 9 – Filtre médian

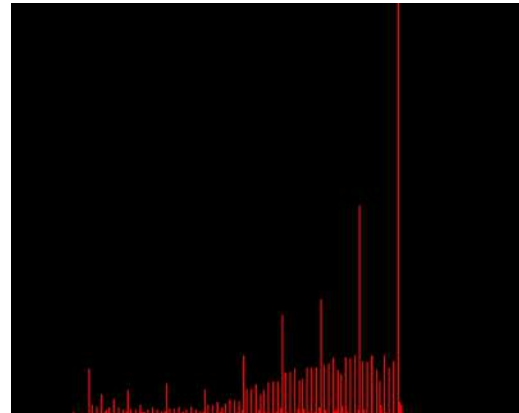


FIGURE 10 – Histogramme de l'image

$\sigma_w^2(t)$  : La variance intraclasse à un seuil.  $t$ .

$\omega_1(t)$  : Le poids de la classe du premier-plan.  $t$ .

$\sigma_1^2(t)$  : La variance de la classe du premier-plan.  $t$ .

$\omega_2(t)$  : Le poids de la classe de l'arrière-plan.  $t$ .

$\sigma_2^2(t)$  : La variance de la classe de l'arrière-plan.  $t$ .

Voici le résultat après l'utilisation de la méthode d'Otsu :

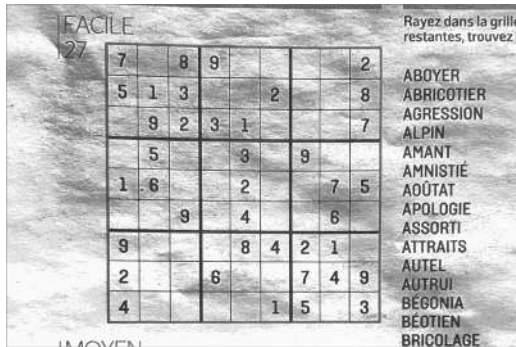


FIGURE 11 – Filtre médian

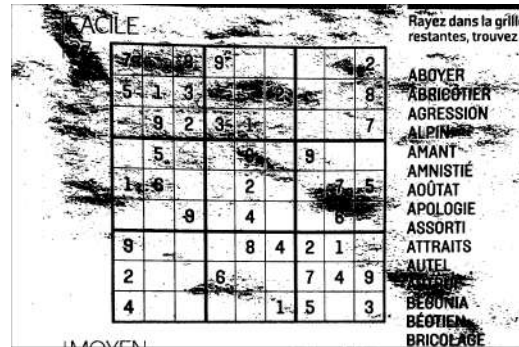


FIGURE 12 – Méthode d'Otsu

Comme on peut le voir, cette méthode est très simple, ce qui pose un problème. En effet, celle-ci perd de son efficacité dès qu'il y a des zones de l'image plus foncées que d'autres. (ombre, vignettage, etc...)

### 2.2.2 Seuil adaptatif

En revanche, les méthodes de binarisation via l'utilisation d'un seuil adaptatif règle ce problème. En effet, au lieu d'effectuer un seul seuil, nous divisons l'image en multiples zones. A la suite de cela, il nous suffit de calculer le seuil pour chacune de ces zones.

Pour calculer la taille d'une zone, nous calculons le niveau de bruit dans l'image. Plus une image est bruitée, plus il faut de nombreuses zones différentes, et inversement.

Pour calculer le niveau de bruit d'une image, il suffit de calculer la moyenne des valeurs dans une fenêtre, et de compter le nombre de pixels qui se trouvent bien au-dessus de cette moyenne.

La taille de la fenêtre et le seuil pour considérer qu'il s'agit d'un pixel parasite sont déterminés à l'avance avec des tests.

Cette méthode étant très performante, nous n'avons pas besoin d'effectuer une méthode de détermination du seuil très poussée telle que la méthode d'Otsu, qui si est répétée de multiples fois, peu causer un fort coût de calcul.

C'est pour cela que nous utilisons simplement les valeurs moyennes dans chacune des zones pour déterminer les seuils respectifs.

Voici le résultat de la binarisation avec une méthode de seuillage adaptatif :

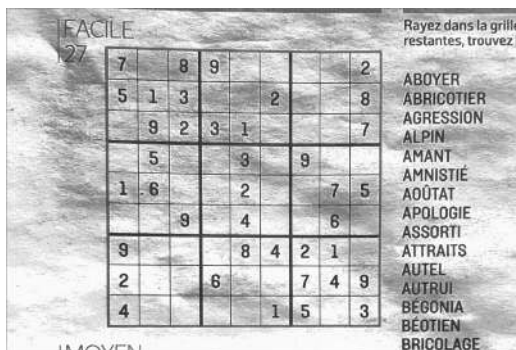


FIGURE 13 – Filtre médian

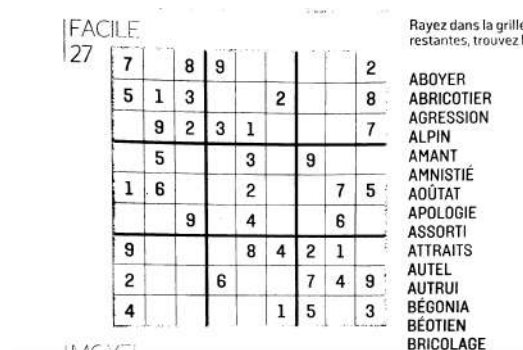


FIGURE 14 – Seuil adaptatif

## 2.3 Un autre filtre

### 2.3.1 Morphologie

Pour finir avec ce processus, nous appliquons un filtre de morphologie. Cela permet de séparer deux objets légèrement collés, ainsi que de compléter les espaces vides quand il y en a.

Cette opération se décompose en deux filtres quasi-identiques.

- Filtre de dilatation
- Filtre d'érosion

Principe de fonctionnement du filtre de dilatation :

Soit la matrice des pixels voisins de pixels de valeur  $n$  (ici 1). La valeur 0 représente un pixel de couleur blanche, et 1, un pixel de couleur noire.

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Si l'un des voisins du pixel est de couleur noire, alors le pixel devient noir, tel que :

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Voici le résultat de la dilatation :

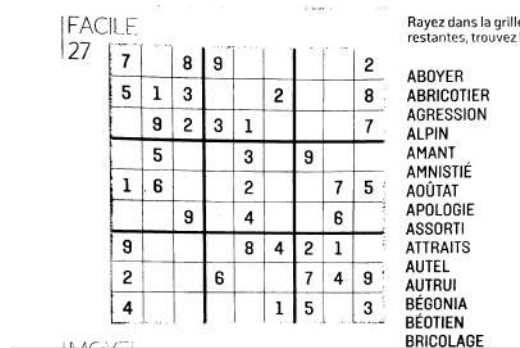


FIGURE 15 – Seuil adaptatif

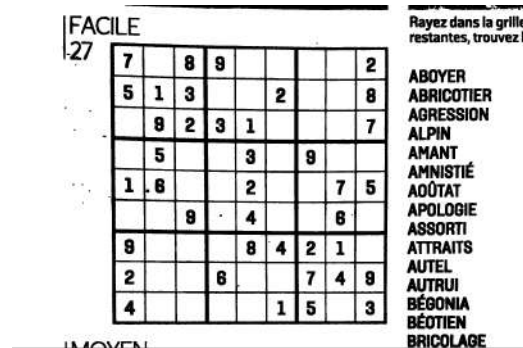


FIGURE 16 – Dilatation

Principe de fonctionnement du filtre d'érosion :

Soit la matrice des pixels voisins de pixels de valeur  $n$  (ici 1). La valeur 0 représente un pixel de couleur blanche, et 1, un pixel de couleur noire.

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Si l'un des voisins du pixel est de couleur blanc, alors le pixel devient blanc, tel que :

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Voici le résultat de l'érosion :

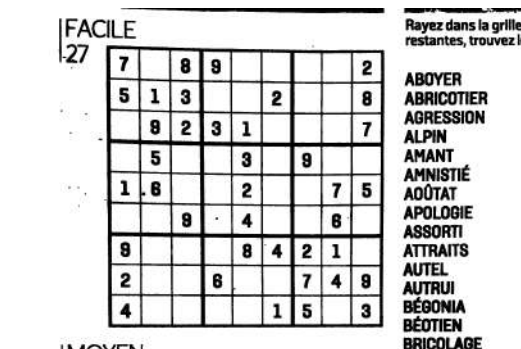


FIGURE 17 – Dilatation

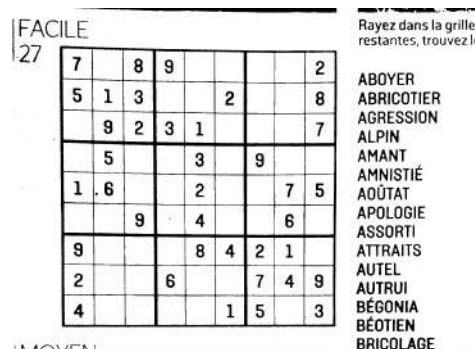


FIGURE 18 – Erosion

## 2.4 Avant, Après

Un comparatif avant/après du processus.

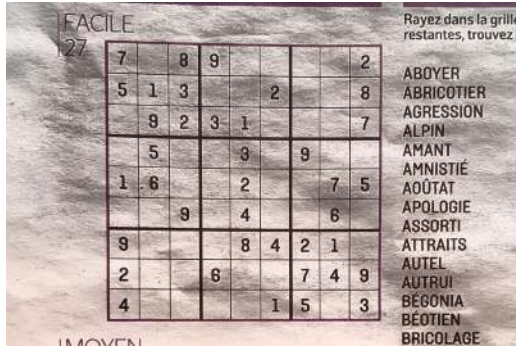


FIGURE 19 – Image original

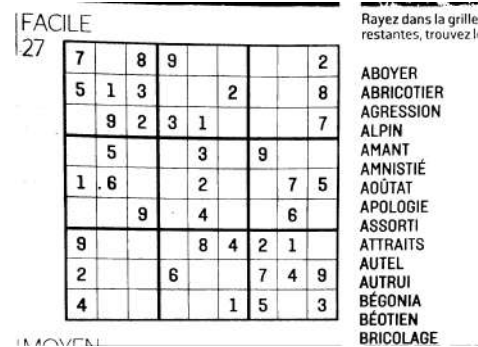


FIGURE 20 – Image final

## 2.5 Rotation de l'image

Il existe plusieurs méthodes de rotation d'images. Nous avons opté pour la méthode qui nous donnera le meilleur résultat. Comme nous l'avons pensé, une méthode plus simple nous donnerait un résultat de plus basse qualité avec des pixels qui n'existent plus à cause de la simplification et de l'imprécision du calcul de la méthode.

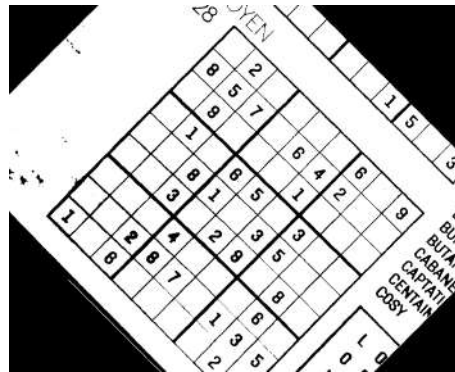


FIGURE 21 – Rotation de 45°

Nous avons donc utilisé une méthode du nom de shearing qui consiste à prendre chaque pixel de l'image puis à appliquer 3 multiplications successives par des matrices de shearing. Le shearing est une transformation qui va consister à décaler chaque point de l'image en fonction d'un certain angle.

Notre rotation fait de 3 shearing consécutifs pour avoir une perte de qualité minimum :

Nous avons donc le calcul suivant avec  $\theta$  notre angle en radian :

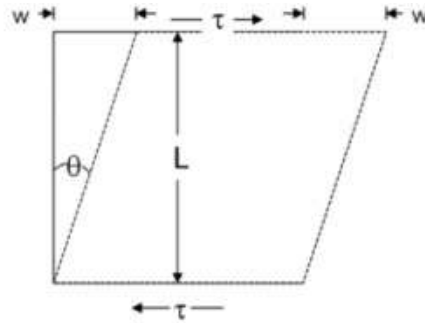


FIGURE 22 – Illustration de la rotation par shearing



FIGURE 23 – Rotation par shearing

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & \cos(\theta/2) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \sin(\theta) & 1 \end{pmatrix} \begin{pmatrix} 1 & -\cos(\theta/2) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Après avoir calculé nos nouvelles coordonnées, ils nous faut leur ajouter un décalage pour pouvoir bien replacer l'image en son centre.

La perte de qualité après rotation est minime, cependant il y a un peu de crénelage (c'est-à-dire qu'il y a des sortes d'escaliers) sur l'image. Mais cela n'empêche en rien le bon fonctionnement des fonctions qui suivront la rotation.

## 2.6 Homographie

La transformation de perspective ou homographic transform en anglais, va nous permettre de "normaliser" notre grille, c'est-à-dire la remettre aux bonnes dimensions, la "redresser", si l'angle de la photo originale n'est pas correcte, et la tourner en même temps. Pour appliquer cette transformation, nous devons d'abord trouver les coefficients de la matrice de transformation. En effet pour chaque transformation, nous avons besoin d'une matrice 3\*3 pour appliquer une transformation linéaire, que ce soit une rotation ou une réduction d'échelle. Soient  $(x, y)$  les coordonnées d'un coin



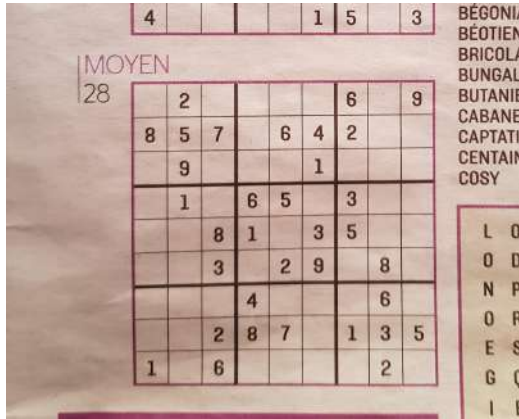


FIGURE 24 – Avant rotation

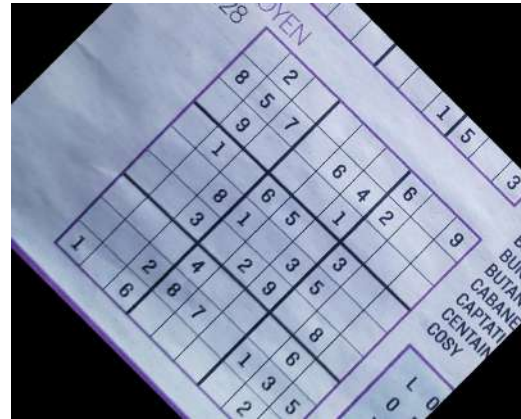


FIGURE 25 – Après rotation

de l'image d'arrivée,  $(x', y')$  les coordonnées d'un coin de l'image de départ et  $a, b, c, d, e, f, g, h$  nos coefficients de matrice, nous avons le calcul suivant :

$$\begin{pmatrix} x' \\ y' \\ w \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Où  $w = gx + hy + 1$ , le problème étant qu'on ne veut pas le garder, car notre image est en 2 dimensions alors nous allons réécrire notre matrice pour faire apparaître un 1 à la place de  $w$  dans notre matrice de fin. En remaniant notre expression, nous avons :

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \frac{\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}}{\begin{pmatrix} g & h & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}}$$

On peut donc déduire les coordonnées de  $x'$  et  $y'$  :

$$x' = \frac{ax + by + c}{gx + hy + 1}$$

$$y' = \frac{dx + ey + f}{gx + hy + 1}$$

Après simplification nous avons ces deux expressions :

$$x' = ax + by + c - x'gx - x'hy$$

$$y' = dx + ey + f - y'gx - y'hy$$



On remarque que cela ressemble à un produit d'une matrice par un vecteur si on ajoute tous les termes :

$$\begin{aligned}x' &= ax + by + c + 0d + 0e + 0f - x'gx - x'hy \\y' &= 0a + 0b + 0c + dx + ey + f - y'gx - y'hy\end{aligned}$$

Nous avons donc ces matrices :

$$\begin{pmatrix} x & y & 1 & 0 & 0 & 0 & -x'x & -x'y \\ 0 & 0 & 0 & x & y & 1 & -y'x & -y'y \\ & & & & & & \vdots & \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ \vdots \end{pmatrix}$$

Ici, nous pouvons utiliser une unique matrice pour calculer les 4 coins de notre image en même temps.

Soit B la matrice contenant les coordonnées de nos coins,  $\lambda$  la matrice contenant nos coefficients et A l'autre matrice.

Nous avons :

$$\begin{aligned}A\lambda &= B \\ A^T A\lambda &= A^T B \\ \lambda &= (A^T A)^{-1} A^T B\end{aligned}$$

Une fois nos coefficients calculés, il ne nous reste qu'à appliquer la matrice à chaque pixel de l'image.

## 2.7 Interpolation bilinéaire

L'interpolation bilinéaire ou filtrage bilinéaire ou encore mappage de texture bilinéaire est une méthode de rééchantillonnage en vision par ordinateur qui va nous permettre de redimensionner une image avec une hauteur et une largeur donnée.

L'interpolation bilinéaire est réalisée en utilisant une interpolation linéaire d'abord dans une direction, puis à nouveau dans une autre direction. Dans notre cas, tout d'abord deux interpolations linéaire sur l'axe X puis une interpolation linéaire sur l'axe des Y.

Bien que chaque étape soit linéaire dans les valeurs échantillonnées et dans la position, l'interpolation bilinéaire dans son ensemble n'est pas linéaire mais quadratique dans l'emplacement de l'échantillon.

Nous avons utilisé cette méthode en la remaniant, c'est-à-dire que :

Pour chaque pixel de l'image finale, nous avons effectué un calcul pour nous permettre d'avoir les coordonnées d'un pixel équivalent sur l'image de départ. Or, ces coordonnées peuvent avoir des chiffres

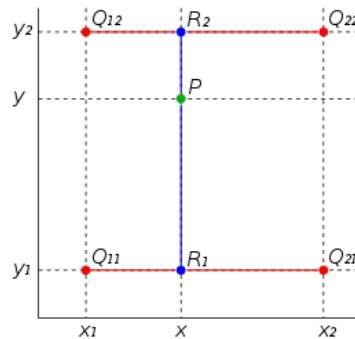


FIGURE 26 – Illustration de l'interpolation bilinéaire

Les quatre points commençant par  $Q$  sont les points existants, et le point  $P$  est le point dont on cherche la valeur par interpolation.

après la virgule donc pour cela nous regarderons les pixels avec la valeur entière de ces coordonnées ainsi que ceux avec la valeur supérieure. Ce qui nous donne quatre pixels autour de ces coordonnées.

Avec ceci, nous pouvons connaître la couleur du pixel que l'on souhaite avoir sur l'image de retour, pour cela on va appliquer un pourcentage à cette couleur pour avoir quelque chose de plus homogène en termes de couleur sur l'image. Ce pourcentage est calculé en fonction de la proximité du pixel avec un des quatre pixels calculés précédemment.

Le pourcentage est appliqué sur chaque interpolation linéaire que l'on fait, ainsi on évite des calculs trop compliqués car le pourcentage vers un pixel peut être de " $p$ " et dans l'autre sens il sera de " $1 - p$ ".

Lors d'un rétrécissement de l'image, celle-ci perd de la qualité, ce qui est attendu. Alors que dans le cas contraire, on voit apparaître une perte de qualité inattendue sur les lignes dans l'image avec des sortes de pixels qui alternent entre noir et blanc, ce qui est dû à l'approximation du calcul du pixel sur l'image de départ.

## 2.8 Détection de la grille

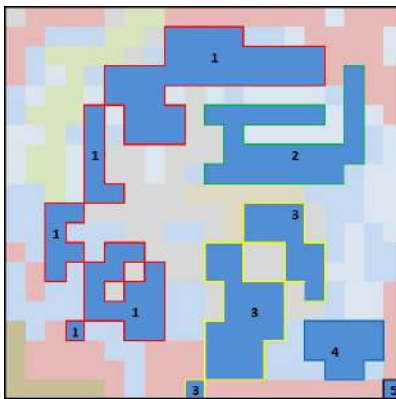
L'objectif de cette partie était de pouvoir détecter la grille de sudoku sur une image, et de l'isoler en ne laissant que la grille sur l'image, ce qui sera plus utile pour la suite du traitement de l'image. Pour arriver à détecter la grille, l'algorithme qui correspondait le plus à notre vision du projet est l'algorithme nommé "Blob Detection". Cet algorithme consiste à parcourir les groupes de pixels adjacents ayant la même couleur (les "blobs"). A chaque nouveau blob détecté, on compare sa taille avec la taille maximale enregistrée pour ainsi obtenir le plus gros blob. Sur une image sur laquelle on a appliqué de la réduction de bruit, ce blob correspond donc à la grille.

Il est important d'appliquer cet algorithme à une image binarisée (avec uniquement des pixels

noirs ou blancs) car cela optimisera son temps d'exécution et évitera de devoir créer des filtres de détection des couleurs, ce qui harmonisera les différents résultats possibles pour l'isolation de la grille sur l'image.

En plus du parcours classique de l'image, il a donc fallu implémenter un algorithme de parcours des pixels voisins efficace, tout en évitant de repasser deux fois par le même pixel, et donc pour cela, nous avons tout simplement utilisé une matrice de la même taille que notre image, avec des entiers représentant chacun un blob. Au début nous avons opté pour un parcours récursif, car plus facile à implémenter. L'algorithme fonctionnait ainsi sur la plupart des images données en exemple, sauf celles qui étaient trop grandes et qui donc entraînaient trop de récursions successives. Il a donc fallu changer l'algorithme en le passant en itératif, en utilisant une stack pour stocker les voisins successifs à partir d'un pixel.

Voilà une représentation des différents blobs qui pourraient être trouvés avec l'algorithme de détection de blobs, avec chaque blob représenté par un nombre, comme ils le seraient dans la matrice que l'on utilise.



C'est-à-dire que chaque fois qu'on appellera la fonction pour parcourir les pixels voisins, on mettra un entier aux coordonnées correspondantes dans la matrice, et on incrémentera l'entier, pour qu'à chaque blob corresponde un entier, qui servira d'identification pour celui-ci. Ainsi, on a juste à stocker l'id du blob le plus gros, et on parcourt l'image une deuxième fois en coloriant chaque pixel en blanc, sauf ceux dont l'id dans la matrice correspond à l'id du blob max. Cela permet de parcourir l'image un minimum de fois pour un résultat optimal.

Concernant la coloration des pixels non-présents dans le blob en blanc, on a dû changer plusieurs fois la manière de procéder. En effet, dans une première version de l'algorithme, nous avons implémenté une coloration du blob en bleu, ce qui, lors d'un second parcours de l'image, nous permettait de différencier le blob maximal des autres groupements de pixels noirs, et de colorier le reste des pixels en blanc. Le problème est qu'il nécessite un parcours de l'image supplémentaire, pour colorier le blob le plus gros en bleu. C'est en implémentant l'algorithme avec l'id des blobs (décrit plus haut) que nous avons pu éviter ce parcours supplémentaire et améliorer encore la détection de la grille ainsi que son isolation dans l'image.

Enfin, il est important de pouvoir conserver l'image non-modifiée par la détection de blobs pour pouvoir conserver les chiffres et leur placement pour la résolution de la grille. L'image modifiée par la détection de blobs sera utile pour d'autres étapes du traitement de l'image.

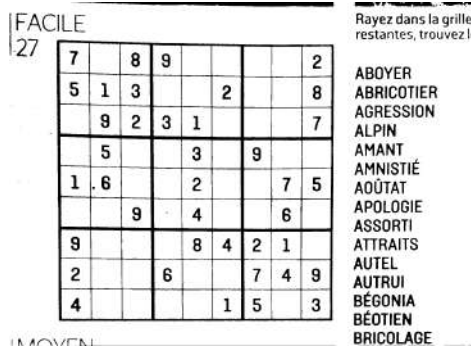


FIGURE 27 – Image non-modifiée

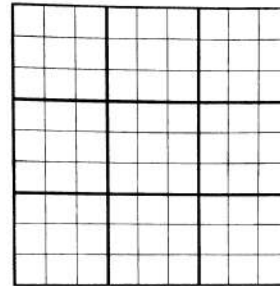
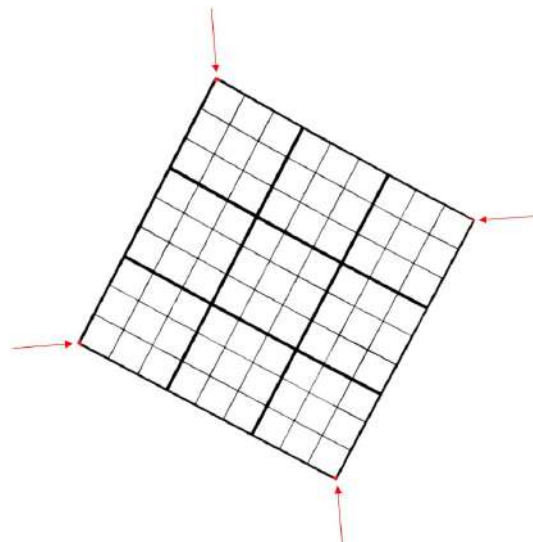


FIGURE 28 – Suppression des petits blobs

## 2.9 Détection des coins de la grille

Afin d'obtenir les coordonnées des coins de la grille de sudoku, nous allons parcourir chaque pixel de l'image binarisée, dont il ne reste que la grille de sudoku après la détection par blob. Ensuite nous allons enregistrer les coordonnées des 4 pixels noirs les plus éloignés les uns des autres jusqu'à la fin du parcours de l'image.



Comme montré ci-dessus, les 4 points les plus éloignés sont forcément les coins de la grille, quelque soit l'orientation de celle-ci. Ces 4 points sont ensuite renvoyés dans une liste d'entiers.

## 2.10 Vérification du Blob

Il se pourrait que dans certain cas le blob trouvé ne soit pas la grille, et pour résoudre ce problème nous avons fait une fonction qui permet, grâce aux 4 points qui sont les extrémités du blob, de savoir si le blob est un carré (avec une petite marge d'erreur). Et si le blob n'est pas la grille, alors on enlève ce blob trouvé dans l'image de base pour ensuite réappliquer la fonction de blob qui nous permettra d'agir sur un deuxième blob maximal, différent de celui trouvé auparavant. On applique cette fonction de vérification jusqu'à ce que l'on ait le bon blob, c'est-à-dire la grille.

## 2.11 Détection des cases de la grille

Pour récupérer chaque cases de la grille, nous allons diviser la grille par 9 dans chaque axe, ce qui nous fait 81 cases. Pour chacune des 81 cases nous prenons les coins de la case et lui appliquons une homographie pour sortir une image de 28\*28 que nous mettons dans une liste. Cette liste contient également la position de chaque case dans la grille pour pouvoir reconstruire la grille de la bonne manière.

## 2.12 Extraction des chiffres présents dans les cases

Avec notre liste de cases de la grille récupérée, nous la trions pour enlever les cases où il n'y a pas de chiffres. Pour les cases restantes dans la liste, nous prenons l'image avec son plus grand blob qui est le chiffres présent sur l'image. Donc, à la fin, nous avons une liste d'images avec seulement les chiffres. Notons que les images que le réseau utilisera sont des images où le chiffres est blanc sur fond blanc et non l'inverse comme ce que nous avions dans une première version de l'algorithme. Nous avons donc dû échanger les couleurs pour que les images envoyées au réseau de neurone soient compatibles avec le mode de détection de ce dernier.

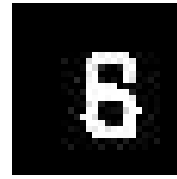
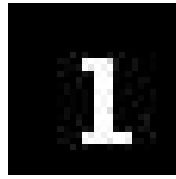
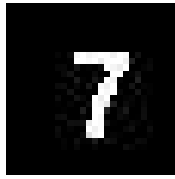


Ci-dessus sont les images découpées et renvoyées dans la fonction de nettoyage des images. Elles seront ensuite repassées dans la fonction de blob afin de garder le chiffre pour les cases avec un chiffre, ou le plus gros amas de pixels parasites dans le cas d'une case vide.

Dans le cas où les lignes de la grille seraient sur la case découpée à cause d'une distortion de l'image, les bords de l'images sont vérifiés. Si jamais une ligne de pixel noire trop longue est détectée, alors elle est effacée.

Afin de différencier les amas de pixels parasites des chiffres afin de l'indiquer au réseau de neurones, nous avons mis en place une limite de pixel.

Ensuite les couleurs de l'images sont inversées pour le réseau de neurones.



Enfin, une fois que toutes ces étapes sont passées, l'image finale est envoyée dans le réseau de neurones.

Ci-dessus les images finales.

## 3 Solveur de sudoku

### 3.1 Validité de la grille

Avant de chercher à résoudre la grille, on vérifie d'abord que la grille est valide. C'est-à-dire qu'on va chercher à vérifier si celle-ci respecte les règles du sudoku.

- Il existe une seule occurrence d'un nombre par rangée.
- Il existe une seule occurrence d'un nombre par colonne.
- Il existe une seule occurrence d'un nombre dans la sous-zone de la grille.

### 3.2 Résolution de la grille

Pour résoudre la grille de sudoku, nous utilisons un algorithme de backtracking. Celui-ci parcourt la grille à la recherche d'une case vide et y place une valeur possible. Pour chacune de ces valeurs, l'algorithme teste récursivement si une solution valide peut-être construite à partir de cette valeur. Si cela échoue, alors il passe à la valeur possible suivante, et ainsi de suite. S'il n'y a plus de valeur possible et qu'il a échoué, alors la grille est impossible. Sinon la grille est résolue.

Dans le meilleur des cas, cet algorithme est de complexité linéaire. Dans le pire des cas, cet algorithme est de complexité exponentielle.

Ainsi pour une grille de sudoku de 9x9, cela est quasiment instantané, même sur une grille difficile. En revanche, pour une grille de sudoku de 16x16, le temps de calcul d'une grille complexe peut vite se faire ressentir sur le temps d'exécution.

### 3.3 Affichage de la grille résolue

Une fois la grille résolue, il est possible de convertir les données de la grille en une image. De nombreux paramètres (police, taille, bordure, ...) permettent de choisir quel style de grille on souhaite avoir en sortie.

Voici le résultat de la conversion en image de la grille de sudoku :

5	4	3	9	2	1	8	7	6
2	1	9	6	8	7	5	4	3
8	7	6	3	5	4	2	1	9
9	8	7	4	6	5	3	2	1
3	2	1	7	9	8	6	5	4
6	5	4	1	3	2	9	8	7
7	6	5	2	4	3	1	9	8
4	3	2	8	1	9	7	6	5
1	9	8	5	7	6	4	3	2

FIGURE 29 – Grille 9x9

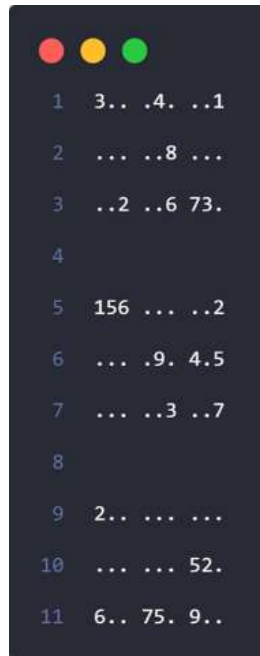
0	5	1	2	6	B	7	4	C	F	E	3	9	D	A	8
6	E	C	9	0	D	8	3	B	1	A	4	7	2	5	F
7	B	8	F	1	2	9	A	5	D	0	6	4	3	C	E
A	4	3	D	5	C	F	E	8	2	9	7	0	B	1	6
5	8	0	7	B	1	2	6	3	A	4	C	F	9	E	D
D	1	B	4	8	E	3	C	9	6	2	F	A	7	0	5
2	6	F	C	A	7	5	9	0	E	D	B	8	4	3	1
3	9	E	A	F	0	4	D	7	8	1	5	B	C	6	2
4	3	2	6	9	8	E	0	A	7	5	D	C	1	F	B
1	D	5	0	7	3	6	F	4	C	B	E	2	8	9	A
C	F	9	B	2	A	D	1	6	0	3	8	E	5	7	4
8	A	7	E	C	4	B	5	1	9	F	2	3	6	D	0
E	C	6	1	4	F	0	7	D	B	8	9	5	A	2	3
9	7	A	8	3	5	1	2	E	4	6	0	D	F	B	C
B	2	D	5	E	6	C	8	F	3	7	A	1	0	4	9
F	0	4	3	D	9	A	B	2	5	C	1	6	E	8	7

FIGURE 30 – Grille 16x16

### 3.4 Fichier du solveur

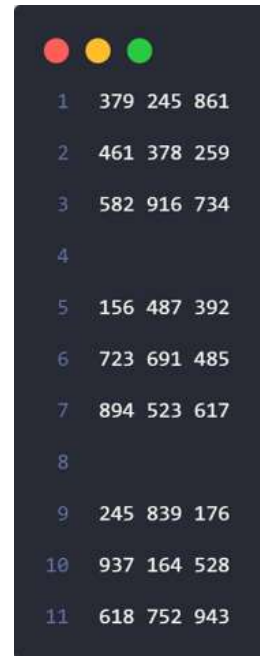
Pour pouvoir facilement résoudre une grille dans le solveur, un fichier contenant la grille d'entrée est utilisé. Une fois la grille résolue, le solveur créera automatiquement un nouveau fichier contenant la grille résolue :





1	3.. .4. ..1
2	... ..8 ...
3	..2 ..6 73.
4	
5	156 ... ..2
6	... .9. 4.5
7	... ..3 ..7
8	
9	2.. ... ..
10	... ... 52.
11	6.. 75. 9..

FIGURE 31 – Grille d'entrée



1	379 245 861
2	461 378 259
3	582 916 734
4	
5	156 487 392
6	723 691 485
7	894 523 617
8	
9	245 839 176
10	937 164 528
11	618 752 943

FIGURE 32 – Grille de sortie

## 4 Réseau de neurones

La tâche du réseau de neurones est de convertir l'image d'un chiffre en un chiffre correspondant à celui présent dans l'image.

Depuis la dernière soutenance, nous avons implementé la version 2 du réseau de neurones qui est bien plus performante et rapide.

De plus, l'implementation du modèle de réseau de neurones que nous avons mis en place est générique. C'est-à-dire que nous pouvons choisir précisément la configuration de celui-ci pour s'adapter à différentes situations.

### 4.1 Jeu de donnée

Dans le cadre de la reconnaissance de caractère, nous avons utilisé une base de donnée open source de 60 000 images de chiffres écrits à la main. Ce jeu de donnée est fourni par Mnist.

De plus, nous séparons ce dataset en deux : un dataset 50 000 images, que l'on utilisera dans le cadre de l'entraînement du réseau de neurones, ainsi qu'un dataset de 10 000 images qui nous servira à tester la capacité de reconnaissance de ce dernier.

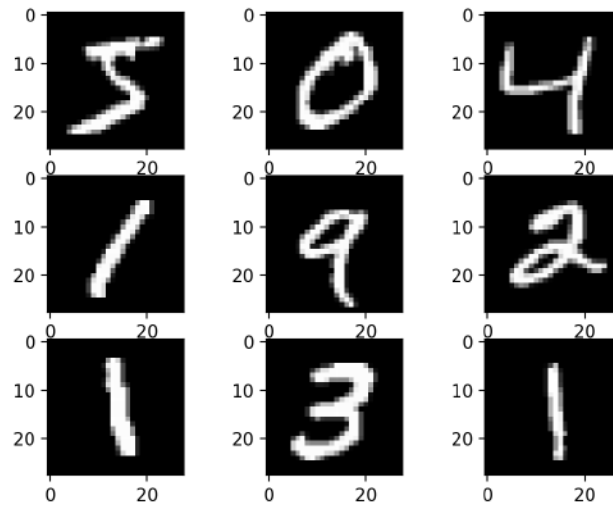


FIGURE 33 – Quelques chiffres du dataset

## 4.2 Configuration de réseau

La configuration que nous avons choisi pour ce projet est une configuration en 3 couches, détaillées plus bas.

Etant donné notre choix d'utiliser un dataset nommé Mnist contenant 60 000 images d'une résolution de 28x28, nous avons configuré notre réseau pour s'adapter à celui-ci.

Détail de la configuration utilisée :

- La première couche est composée de 784 neurones, chaque neurone correspondant à un pixel distinct de l'image
- La seconde est quant à elle composée de 128 neurones
- La dernière couche, composée de 10 neurones, correspond respectivement aux chiffres 0 à 9.

Le choix de dix neurones pour la couche de sortie s'explique par de meilleurs résultats obtenus en encodant le chiffre en base 10, au lieu d'une base 2.

## 4.3 Propagation avant

La propagation avant consiste à propager l'entrée initiale du réseau à travers toutes les couches dites "cachées" jusqu'à la dernière couche.

Pour effectuer cette opération nous utilisons la formule suivante :

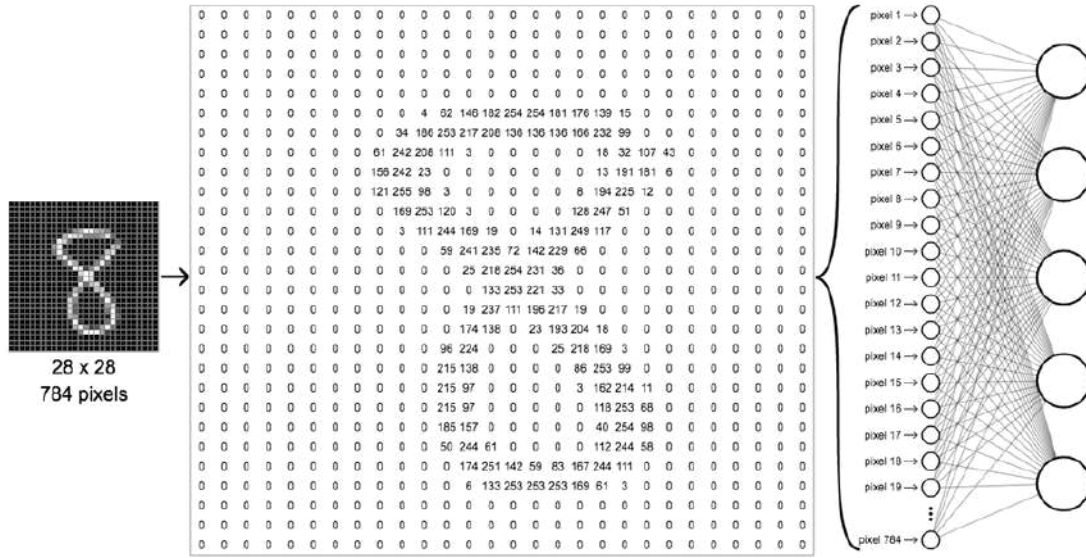


FIGURE 34 – Schéma de l’attribution des valeurs de la première couche

$$N_i^L = \sigma(N_i^{L-1} \times W_i^L + B_i^L)$$

$\sigma$  : La fonction d’activation sigmoid

$N_i^L$  : La valeur du neurone à l’indice  $i$  de la couche  $L$

$N_i^{L-1}$  : La valeur du neurone à l’indice  $i$  de la couche précédant la couche  $L$

$W_i^L$  : Est la matrice des poids du neurone  $i$  de la couche  $L$

$B_i^L$  : Est le biais du neurone  $i$  de la couche  $L$

Pour la dernière couche, nous faisons quasiment la même chose, mais nous utilisons la fonction d’activation softmax, c’est-à-dire :

$$N_i^L = \text{Softmax}(N_i^{L-1} \times W_i^L + B_i^L)$$

Grâce à cette étape, il nous est possible de connaître les valeurs de sortie du réseau et ainsi de connaître le chiffre reconnu par le réseau de neurones.

#### 4.4 Apprentissage du réseau

Pour que le réseau retourne les valeurs que l’on souhaite pour notre application. Nous devons procéder à une étape d’apprentissage.

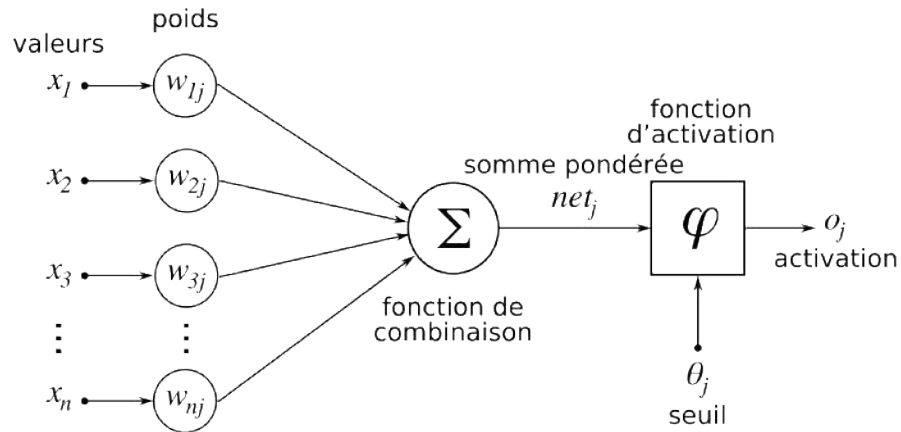


FIGURE 35 – Illustration de la propagation en avant sur un neurone

L'apprentissage consiste concrètement à calculer l'erreur de la couche de sortie puis de la propager en arrière pour calculer l'erreur des couches précédentes.

Pour calculer l'erreur de la couche de sortie

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \times \sigma'(z_j^L)$$

Pour calculer l'erreur d'une couche précédente :

$$\delta_j^L = \left( \left( W_{jk}^{L+1} \right) \delta_j^{L+1} \right) \times \sigma'(z_j^L)$$

Une fois que nous avons calculé l'erreur de chaque neurone, nous pouvons procéder aux calculs des nouveaux poids et biais.

Pour calculer le nouveau poid à partir de l'erreur calculée :

$$W_{jk}^L - = \alpha \times (a_k^{L-1} \times \delta_j^L)$$

Et pour calculer le nouveau biais :

$$b_j^L - = \alpha \times \delta_j^L$$

Avec :

$\delta_j^L$  : Erreur calculée pour le neurone  $j$  de la couche  $L$

$C$  : La fonction de coût (ici, le coût quadratique)

$a_j^L$  : La valeur attendue du neurone à l'indice  $j$  de la couche  $L$

$\sigma'$  : La dérivée de la fonction sigmoïd

$z_j^L$  : La valeur calculée à partir de la somme des poids, biais et de la valeur de sortie de la couche précédente

$W_{jk}^L$  : Poids du neurone  $j$  à la couche  $L$

$\alpha$  : Le taux d'apprentissage

$b_j^L$  : Le biais du neurone à l'indice  $j$  de la couche  $L$

Ainsi, en effectuant une propagation arrière de l'erreur, nous pouvons ajuster légèrement\* le biais et les poids de chaque neurone. Cela a pour effet de faire converger notre réseau de neurones vers le résultat attendu.

\*En fonction du taux d'apprentissage que l'on a indiqué.

## 4.5 Paramètres d'apprentissages du réseau de neurones

Lorsque l'on parle d'apprentissage de réseau de neurones, on parle aussi de ses paramètres. Taux d'apprentissage, mini-batch ou bien nombre d'époque sont les paramètres principaux de l'entraînement d'un réseau de neurones.

### 4.5.1 Taux d'apprentissage

Le taux d'apprentissage est un facteur qui s'applique au moment de la mise à jour des poids et des biais, noté  $\alpha$ .

Le choix du taux d'apprentissage est crucial dans le processus d'entraînement d'un réseau de neurones, car il influence directement la convergence de l'algorithme d'optimisation. Le taux d'apprentissage, détermine la taille des pas effectués lors de la mise à jour des poids du réseau à chaque itération. Un taux d'apprentissage inapproprié peut entraîner des conséquences significatives, telles que la convergence lente, la divergence de l'entraînement, voire l'échec complet de la convergence. Un taux trop élevé peut provoquer des oscillations ou des dépassements de la solution optimale, tandis qu'un taux trop bas peut entraîner une convergence extrêmement lente et, dans certains cas, la convergence vers un minimum local au lieu du minimum global.

### 4.5.2 Mini-batch

Le mini-batch est une partie du dataset d'entraînement. Celui-ci détermine le nombre d'exemples d'entraînement utilisés pour mettre à jour les poids du réseau à chaque itération. De plus un mini-batch adéquat impacte directement la stabilité et la rapidité de convergence du modèle. Des mini-batches trop petits peuvent introduire une variabilité excessive, tandis que des mini-batches trop grands peuvent ralentir le processus d'entraînement.

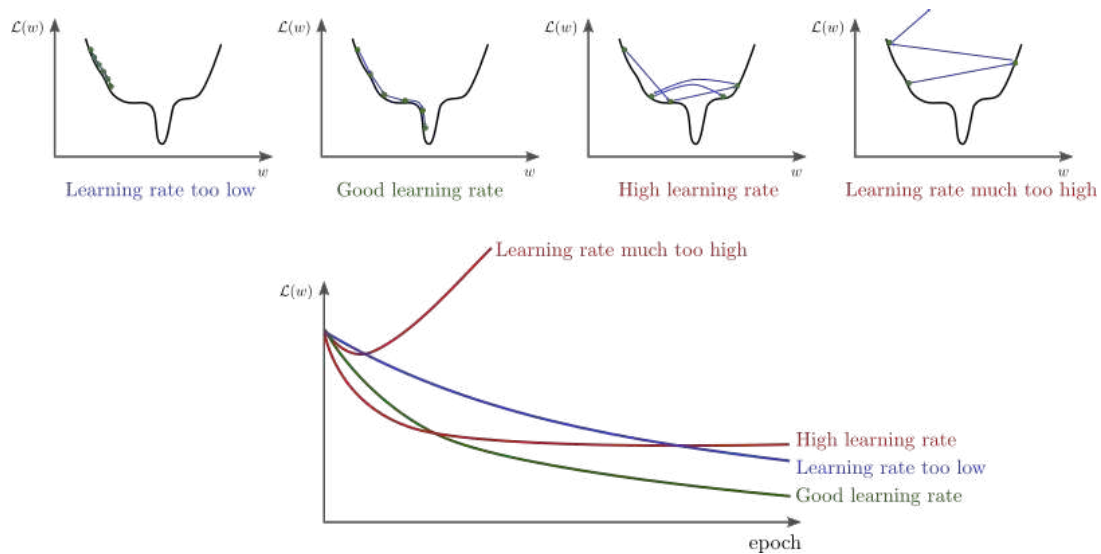


FIGURE 36 – Schéma sur l'importance du taux d'apprentissage

En outre, les mini-batch permettent aussi l'implementation de l'apprentissage sur plusieurs threads, permettant ainsi de raccourcir le temps de l'apprentissage du réseau de neurones.

#### 4.5.3 Epoch

Les epochs sont le nombre de fois que l'on parcourt l'ensemble complet du dataset. Chaque epoch consiste en une passe avant et une passe arrière à travers l'ensemble des données. Le choix du nombre d'epochs est un aspect crucial de l'entraînement du modèle, et il est important de trouver un équilibre.

En effet, il existe un risque associé à un nombre excessif d'epochs, le surapprentissage (ou overfitting). Le surapprentissage se produit lorsque le modèle apprend non seulement les tendances générales des données d'entraînement, mais aussi le bruit et les détails spécifiques à ces données. Il peut même arriver que le réseau apprenne l'ordre des images, d'où l'importance de mélanger à chaque epoch les mini-batches.

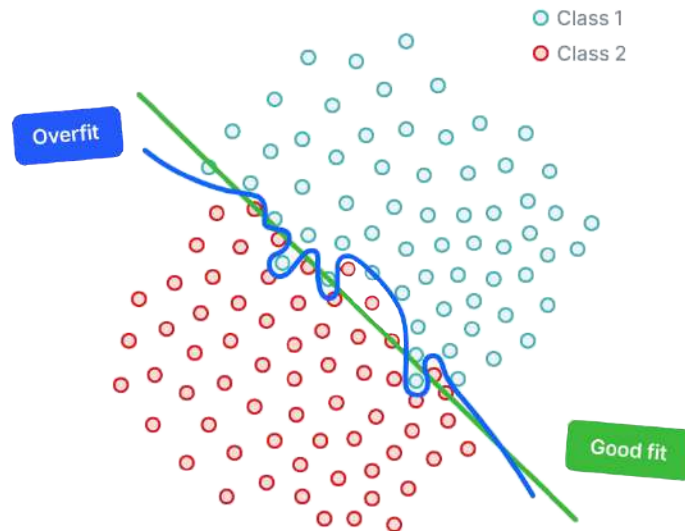


FIGURE 37 – Illustration représentant le surapprentissage

#### 4.6 Performance du réseau de neurones

Grâce à la version 2 de notre réseau, nous arrivons à des performances satisfaisantes. Sur un réseau composé d'une couche cachée avec 50 neurones, nous arrivons à obtenir 98.8% de chiffres reconnus sur un dataset de 50 000 images et en 30 epochs.

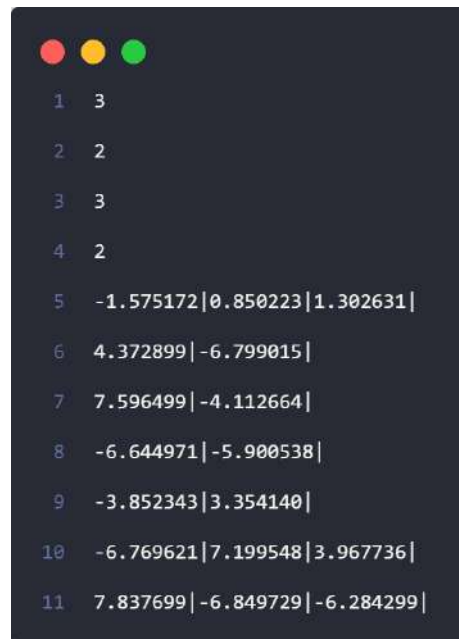


```
1 #####
2
3   Training data spec :
4
5 #####
6
7   Number of epoch : 30
8   Batch size : 50000
9   Learning rate : 0.10
10
11   Save training : Off
12
13   No neural network specification ⚠
14   Load default neural network specification.
15 #####
16
17   Neural network parameters :
18
19 #####
20
21   Input layer :
22     Number of neurons : 784
23
24   Hidden layer :
25     Number of layers : 1
26     Number of neurons per layer : 128
27
28   Output layer :
29     Number of neurons : 10
30
31 #####
32
33   Starting digit training. :
34
35 #####
36
37 Epoch: 6 | Progress: [=====>] 23.33% | Accuracy : 96.48%
```

FIGURE 38 – Console de l'apprentissage des chiffres

## 4.7 Sauvegarde du réseau de neurones

Pour sauvegarder l'état d'un réseau de neurones après un entraînement, nous stockons la configuration du réseau (nombre de couches, nombre de neurones par couches), ainsi que les poids et biais de chaque neurone. Cet enregistrement s'effectue dans un fichier .txt qui nous permet d'utiliser un réseau entraîné ultérieurement.



```
1 3
2 2
3 3
4 2
5 -1.575172|0.850223|1.302631|
6 4.372899|-6.799015|
7 7.596499|-4.112664|
8 -6.644971|-5.900538|
9 -3.852343|3.354140|
10 -6.769621|7.199548|3.967736|
11 7.837699|-6.849729|-6.284299|
```

FIGURE 39 – Vue d'un fichier de sauvegarde d'un réseau ayant été entraîné par la porte xor

## 4.8 Amélioration du réseau

Bien que notre réseau fonctionne correctement, celui-ci a quelques défauts de conception qui le ralentissent, comme la structure de donnée utilisée par exemple. Actuellement, notre réseau fonctionne avec un ensemble de structs (Layer, Neuron, ...). L'optimal serait de mettre toutes les données dans des matrices et de faire les opérations directement dessus, comme il existe des opérations bien plus optimisées sur les matrices pour obtenir les mêmes résultats.

Cela a pour conséquence de rendre plus lent son apprentissage.

Une autre optimisation possible serait d'implémenter l'apprentissage en multi-thread. Cela permettrait de paralléliser les calculs durant l'entraînement du réseau, et ainsi rendre l'apprentissage encore plus rapide.

## 5 Interface graphique

Pour que notre projet soit accessible au grand public, nous avons créé une interface graphique permettant de faire tout le déroulement du processus de résolution d'une grille de sudoku à partir d'une image.

Cette interface graphique est désignée autour d'une barre de progression, qui indique les étapes à suivre pour le bon déroulement du processus.

## 5.1 Chargement d'une image

Lorsque nous ouvrons l'interface graphique, nous arrivons sur une page qui nous demande de charger l'image de la grille que l'on souhaite utiliser.

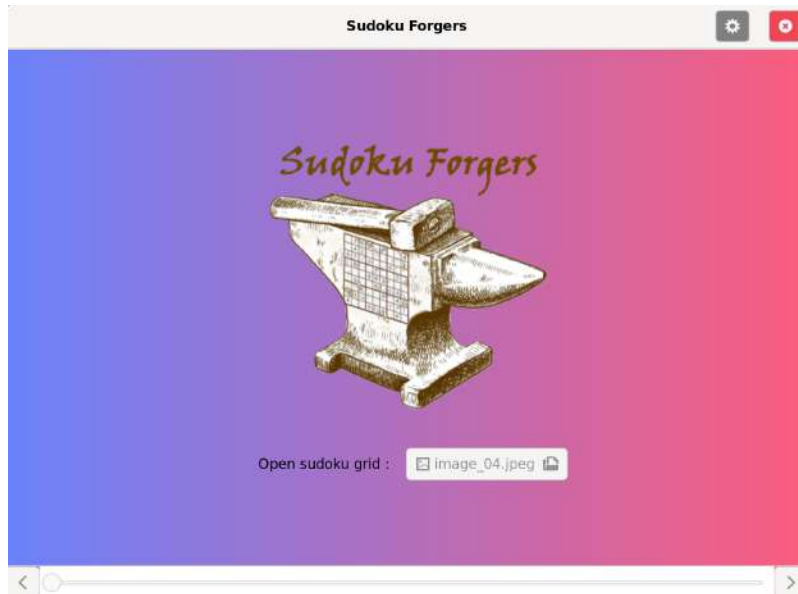


FIGURE 40 – Vue de la page d'accueil de l'application



FIGURE 41 – Fenêtre de sélection de la grille

## 5.2 Rotation manuelle

Une fois que l'utilisateur a chargé une image et cliqué sur la flèche de droite pour passer à l'étape suivante.

Celui-ci a le choix de faire la rotation de son image pour la mettre dans un sens normal de lecture.

Une fois que l'utilisateur estime que l'image est assez droite, il peut passer à l'étape suivante. L'étape suivante procède donc à l'analyse et au traitement de l'image.

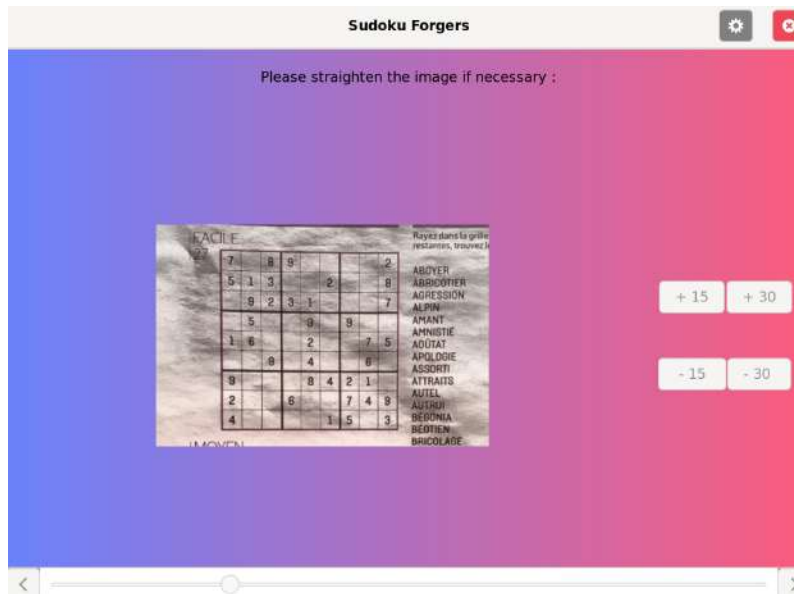


FIGURE 42 – Vue sur la page de la rotation manuelle

### 5.3 Analyse et traitement de l'image

L'utilisateur a la possibilité de voir les différents traitements appliqués à l'image, il lui suffit d'appuyer sur le bouton correspondant au traitement associé.

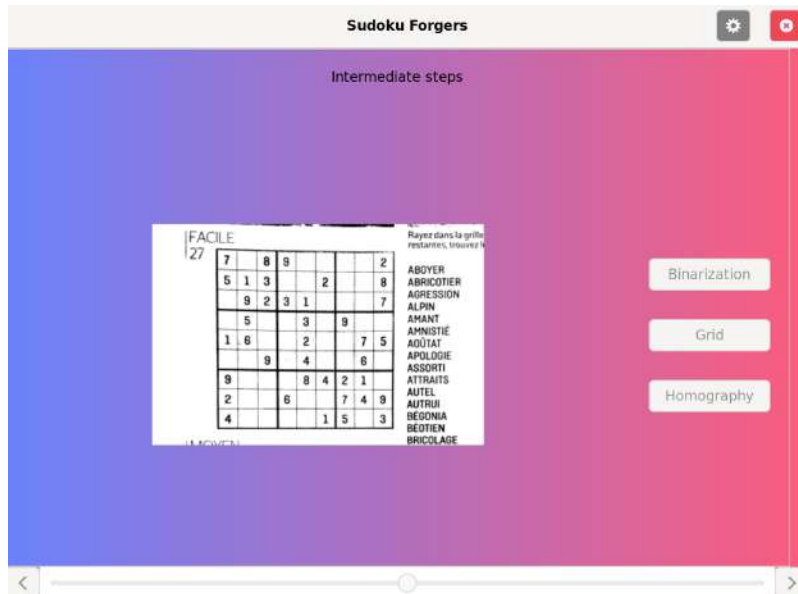


FIGURE 43 – Vue sur la page des traitements intermediaires

## 5.4 Modification des chiffres reconnus

Lorsque l'utilisateur se trouve sur cette page, il a le choix de modifier les chiffres que le réseau de neurones a reconnus en cas d'erreur. Pour cela, il lui suffit de cliquer sur la case de la grille avec une erreur (cela peut être une case vide) puis d'entrer sur son clavier le chiffre voulu.

Une fois les corrections apportées (si nécessaire), l'utilisateur peut passer à l'étape suivante.

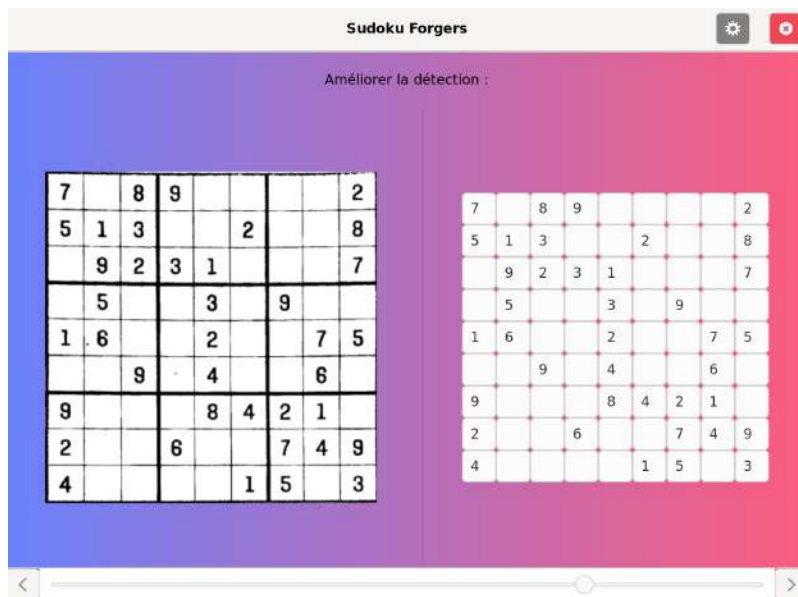


FIGURE 44 – Vue sur la page d'edition de la grille



## 5.5 Grille résolue

Sur cette dernière page, l'utilisateur a la possibilité d'enregistrer la grille de sudoku résolue qui a été générée. L'utilisateur peut aussi, charger une nouvelle image, ou bien consulter les étapes clés du traitement de l'image.

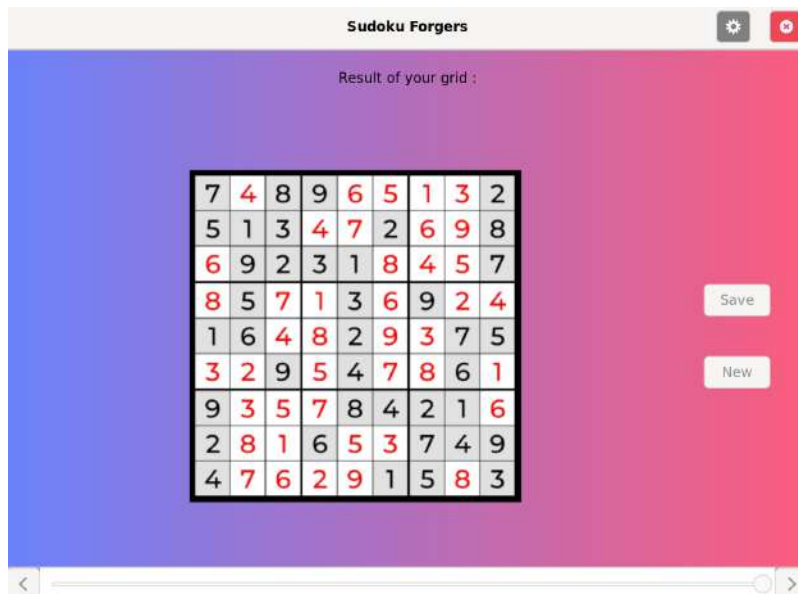


FIGURE 45 – Vue sur la page de la grille résolue



FIGURE 46 – Fenêtre de sauvergarde de la grille résolue

## 5.6 Paramètres

La page des paramètres est activable depuis le bouton engrenage, situé en haut à droite de l'application.

Dans ce menu, l'utilisateur peut modifier les paramètres principaux du réseau de neurones, ainsi que d'y réaliser l'entraînement de ce dernier.

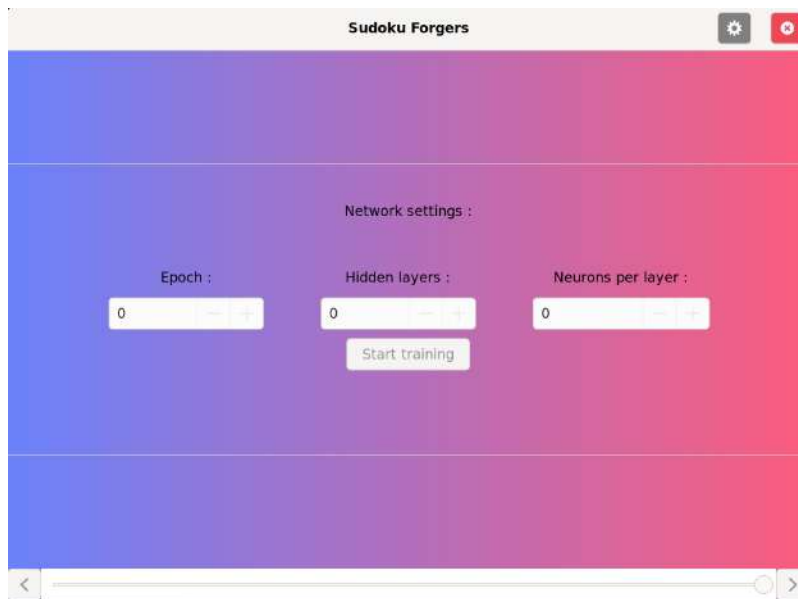


FIGURE 47 – Fenêtre des paramètres

## 6 Site web

Nous avons mis en place un site web pour permettre la diffusion de notre application via le bouton de téléchargement, mais aussi pour présenter notre projet ainsi que son avancement.

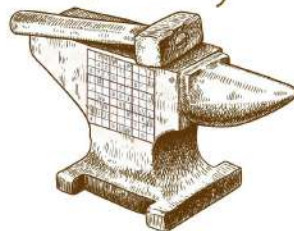


### How does it work :

Our Sudoku Solver is an application that lets you upload an image of your unsolved sudoku grid from your phone, and automatically suggests a solution.

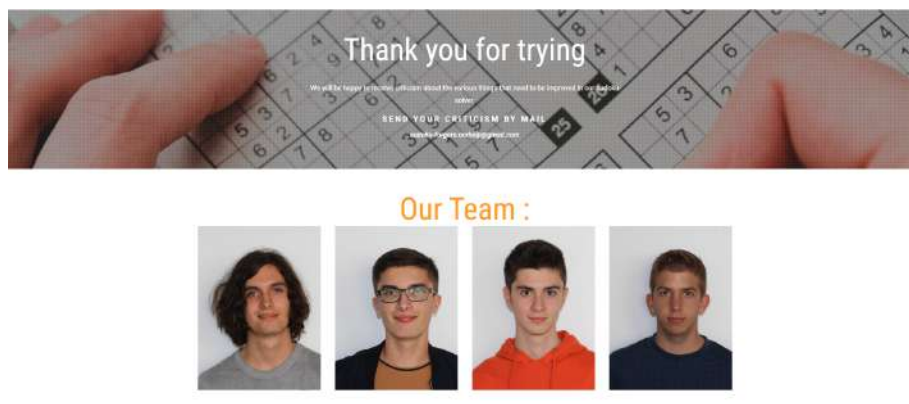
Take a picture of your sudoku and just wait for our application to solve it and give you a clear and fast solution.

### Sudoku Forgers



#### - About Us

We, Sudoku Forgers, are dedicated to making your project success from the start. Our determination and our passion for our work are our main strengths. We will do our best to make your project a success and we will be there for you every step of the way.



Notre site est hébergé sur Github via une page Github, ce qui évite des frais d'hébergement annexe.

## 7 Conclusion du projet

### 7.1 Objectifs précédents

Les objectifs de la dernière soutenance étaient les suivants :

- Création de l'interface graphique
- Mise en place de la seconde version du réseau de neurones
- Amélioration de l'efficacité de certaines fonctions

Ces objectifs ont tous été remplis, et nous avons même relevé le niveau d'exigence que nous avions envers le projet, notamment dans l'interface graphique où nous avons ajouté des gifs et des sons pour rendre l'application plus personnalisée (avec le logo du groupe, etc) et plus agréable à utiliser. Cette exigence se traduit aussi dans l'amélioration de certaines parties de ce projet pour augmenter l'efficacité de leur algorithme, comme la detection de grille ou encore le réseau de neurone. Nous avons également retravaillé la rotation pour que celle-ci engendre le moins de perte possible de l'image d'origine.

### 7.2 Conclusion générale

Pour conclure ce projet, nous avons produit une application permettant de résoudre une grille de sudoku à partir d'une simple image de celle-ci.

Ce projet très intéressant nous a permis de mieux comprendre comment de nombreux outils du quotidien utilisant l'ocr ou bien les réseaux de neurones fonctionnent.

Il a permis à chaque membre du groupe d'explorer des domaines que nous n'avions pas forcément explorés auparavant, notamment le traitement de l'image, ou encore le réseau de neurones. Concernant le traitement de l'image, nous avons eu un tp à ce sujet en début d'année mais l'ocr nécessitait des notions qui n'étaient pas forcément traitées dans ce tp. Ce fut donc un projet très

enrichissant pour tous les membres du groupe, et nous a beaucoup apporté, autant en compétences techniques qu'en cohésion de groupe.