



Rennes
Rapport de soutenance Final

OCR Sudoku Solver

Auteurs :
Hugo Vion
Alexandre Girard
Paul Roost
Jules Péchin

MERCREDI 8 DÉCEMBRE 2021
Promotion 2025

Sommaire

1	Introduction	4
2	Répartition des charges et mise en place	5
3	Le pré-traitement	6
3.1	Greyscale	6
3.2	Binarization	7
3.2.1	La methode Otsu	8
3.3	Rotation de l'image	10
3.4	Réduction de bruit	11
3.5	Découpage de l'image	12
4	La détection de grille	13
4.1	La transformée de Hough	13
4.2	Détection des lignes	14
4.3	Déduction de la grille	15
4.4	Découpage de l'image	16
5	Le réseau de neurones	17
5.1	Introduction	17
5.2	La porte logique XOR	17
5.3	Implémentation	19
5.4	Fonctionnement du réseau de neurones	19
5.4.1	Propagation avant	19
5.4.2	La fonction sigmoïde	20
5.4.3	Rétro propagation	21
5.5	Reconnaître des chiffres	24
5.6	Enregistrement et extraction des données	26
5.7	Entraînement	27

6	Le post-traitement	28
6.1	Résolution de la grille	28
6.2	Reconstruction de la grille	31
7	L'interface graphique	32
8	Prérequis et installation	35
8.1	Prérequis	35
8.2	Installation	35
9	Avancements et retards	36
10	Conclusion	36

1 Introduction

La reconnaissance optique de caractère (de son abréviation anglaise OCR pour Optical Character Recognition), est un système permettant de détecter un texte, mais plus spécifiquement des caractères, à partir d'une image standard. Ce projet doit également reconnaître et résoudre une grille de Sudoku.

Ce projet est composé de six parties : les éventuels traitements d'images, des opérations sur ces images, la détection de la grille, avec extraction de chacune de ses cases, la reconnaissance des chiffres dans cette grille par un réseau de neurones et enfin un post-traitement.

L'objectif du projet est de pouvoir coder un programme en langage C qui aura une utilisation facilitée grâce à une interface graphique et permettant de résoudre son Sudoku grâce à une simple photo. Ce projet est réalisé dans le cadre de nos études en deuxième année du cycle préparatoire à EPITA.

Dans ce rapport de soutenance, on vous présentera tout d'abord la répartition des tâches ainsi que les avancements sur ce projet, pour ensuite également détailler chaque composante de notre projet, et enfin nos avancements et retards dans le projet.

2 Répartition des charges et mise en place

Dans le tableau ci-dessous, vous pourrez voir la répartition des charges de travail, qui étaient à effectuer pour ce projet, au sein de notre groupe :

Tâches	Responsable	Suppléant(s)
Pré-traitement	Paul	Tous
Binarisation Otsu	Paul	Tous
Rotation	Paul	Aucun
Détection de grille	Jules	Aucun
Détection des contours	Jules	Aucun
Segmentation	Jules	Aucun
Réseau de neurones	Hugo	Aucun
Implémentation	Hugo	Aucun
Processus d'entraînement	Hugo	Aucun
XOR	Hugo	Aucun
Sauvegarde Poids/Biais	Hugo	Aucun
Base de données pour l'entraînement	Hugo	Paul
Post-traitement	Tous	Tous
Résolution de la grille	Alexandre	Hugo
Reconstruction de la grille	Paul	Aucun
Interface	Alexandre	Hugo & Paul
Design	Hugo	Alexandre
Code et liaison	Alexandre	Paul & Hugo

3 Le pré-traitement

Toutes les images utilisées pour représenter le comportement des fonctions présentées ci-dessous utiliserons cette image :

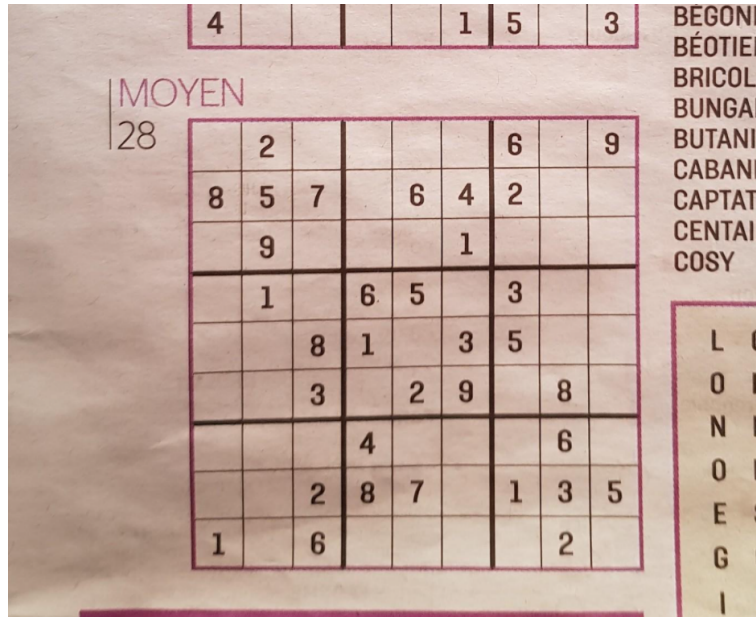


Figure 1: Grille de Sudoku non modifiée

3.1 Greyscale

Une image contient des pixels. Chaque pixel est la combinaison de 3 couleurs : le rouge, le vert et le bleu. Chaque couleur forme un octet donc la valeur décimale est comprise entre 0 et 255. Afin d'obtenir une image en nuances de gris, il faut récupérer la valeur de chaque pixel puis appliquer une formule : $(0,3 \times R + 0,59 \times V + 0,11 \times B)$. Avec ce résultat on va créer un nouveau pixel dont chaque composante R,V et B sera égal à la formule. Une fois le pixel créé, l'ancien pixel sera remplacé par le nouveau. Cette fonction est très utile pour ensuite faire

la binarisation car on n'a plus besoin de faire la moyenne de R, G et B il suffit de regarder l'un des trois.

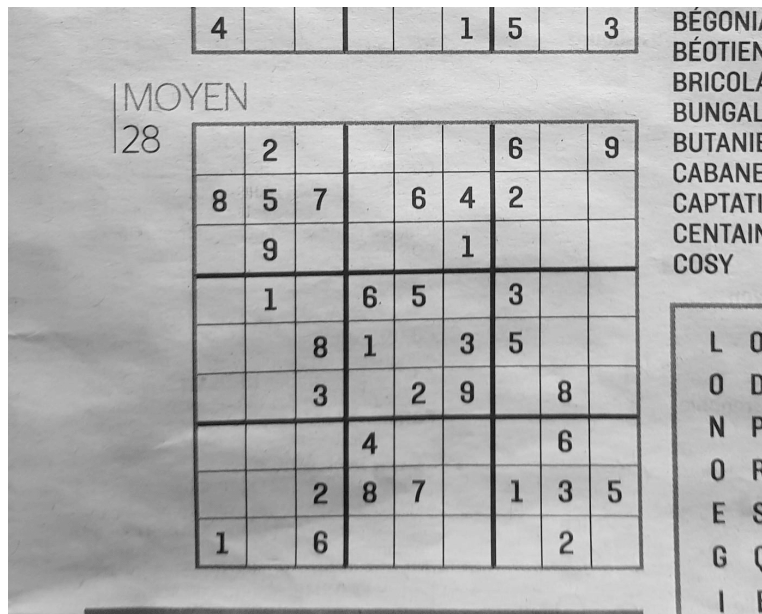


Figure 2: Nuance de gris

3.2 Binarization

Afin de faciliter la reconnaissance de caractères, on va "binariser" l'image : elle sera uniquement composée de pixels noirs et blancs. Pour ce faire, on regarde si la moyenne du R, V et B est supérieure à 127. Si c'est le cas, alors le pixel devient blanc sinon il devient noir. Lorsque l'on arrive sur des images plus complexes, cette méthode n'est plus très efficace. Alors, il faut calculer un nouveau nombre, différentes techniques sont possibles, dont la méthode d'Otsu :

3.2.1 La methode Otsu

Cela consiste à trouver le seuil parfait pour faire un noir et blanc. Pour cela, elle utilise un histogramme de couleur que l'on normalise (cela permet d'obtenir un histogramme dont toutes les valeurs sont comprises entre 0 et 1). Cette formule permet d'y parvenir : $f(x) = \frac{x-\min}{\max-\min}$. Ensuite, on calcule la variance inter-classes entre $C1$ et $C2$ avec cette formule:

$$var_{C1 \cap C2} = \frac{(Moy \times Proba_{C1}(k) - Moy_{C1}(k))^2}{Proba_{C1}(k) \times Proba_{C2}(k)}$$

où Moy est la moyenne de tous les pixels de k . Soit 2 classes de pixels $C1$ et $C2$. La classe 1 ($C1$) est définie comme étant composée des pixels ayant une valeur comprise entre 0 et k avec $k < 255$. Le reste (classe $C2$) des pixels compris entre k et 255 (inclus) font partie de la seconde classe. Le but du jeu est de trouver " k " tel qu'il sépare au mieux le fond et les objets de l'image traitée.

On trouve la variance et on garde la plus grande valeur, l'indice où se trouve la plus grande variance sera le seuil qui remplacera 127. L'inconvénient de cette méthode est qu'elle n'utilise un seuillage global. Ainsi, elle n'est pas adaptée pour les images contenant du bruit.

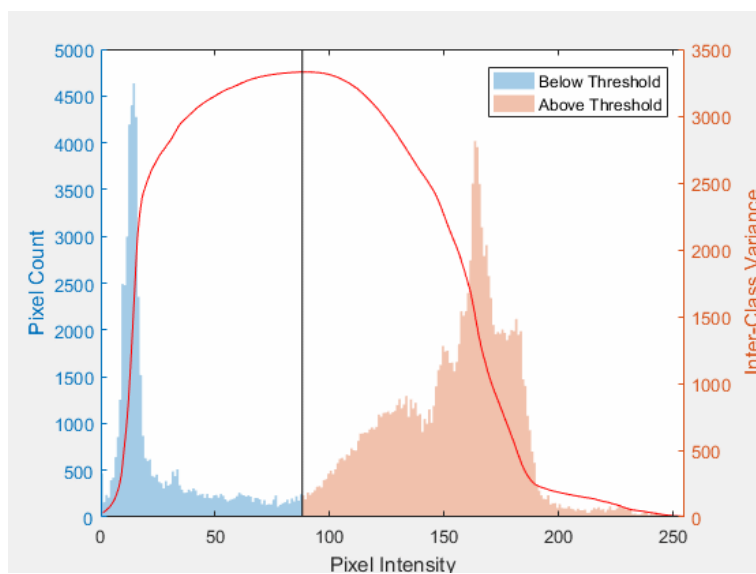


Figure 3: Méthode Otsu

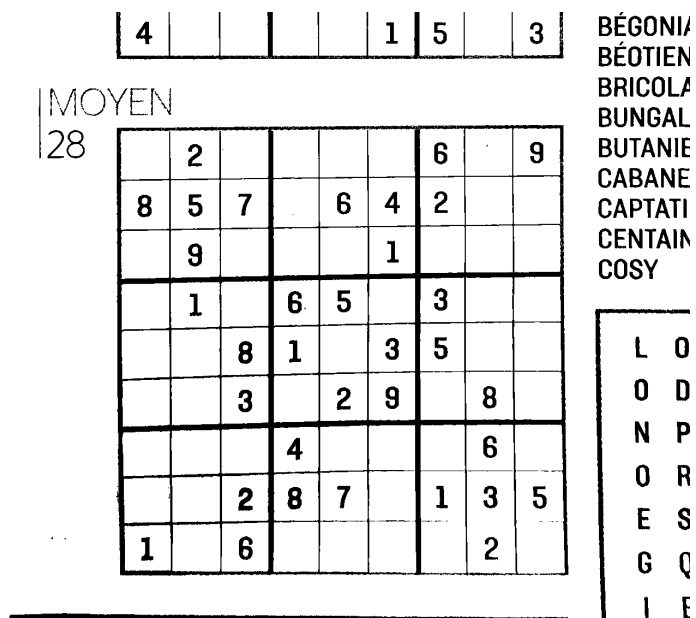


Figure 4: Noir et blanc

3.3 Rotation de l'image

Pour cette fonction, on donne un angle à la fonction, et l'image doit faire une rotation par rapport au degré demandé. Pour cela, il faut recréer une nouvelle image aux bonnes dimensions. Il faut mettre l'angle en radians puis faire le sinus et le cosinus. Ensuite, on doit parcourir la nouvelle image pour mettre le pixel de l'ancienne image au nouvel emplacement. Les calculs effectués peuvent générer des coordonnées pixels qui sont en dehors de l'ancienne image. Si c'est le cas, il est donc impossible de les récupérer ainsi on génère un pixel blanc qui le remplacera.

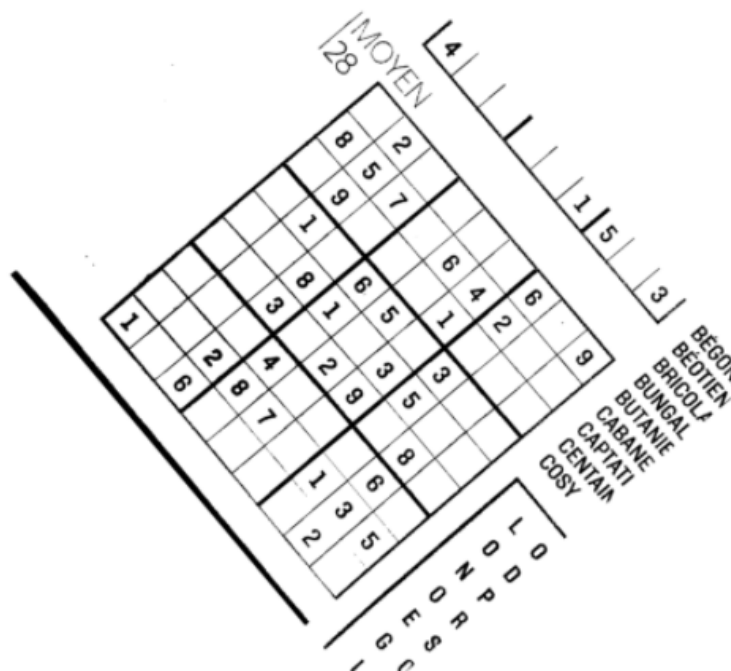


Figure 5: rotation de l'image de 50°

3.4 Réduction de bruit

Afin de réduire le bruit d'une image, on va donner à la fonction l'image que l'on veut traiter, une valeur comprise entre 0 et 7. Cette fonction consiste à enlever le "bruit" d'une image : enlever tous les pixels qui parasitent l'image. La particularité de cette fonction c'est que l'on laisse à l'utilisateur la possibilité d'activer cette fonction et de choisir la valeur "d'optimisation". Il y a également un autre avantage à cette fonction, lorsque l'on met une valeur proche de 7, cela permet de grossir les traits de l'image en créant plus de bruit.

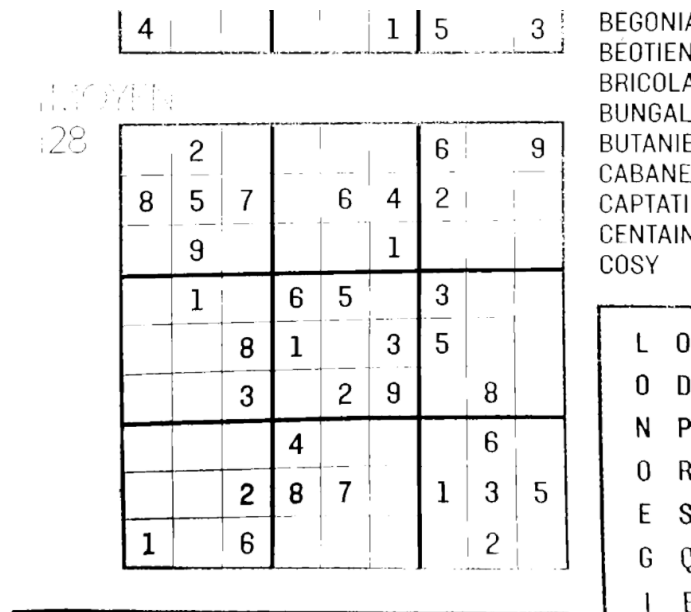


Figure 6: Noise de 0

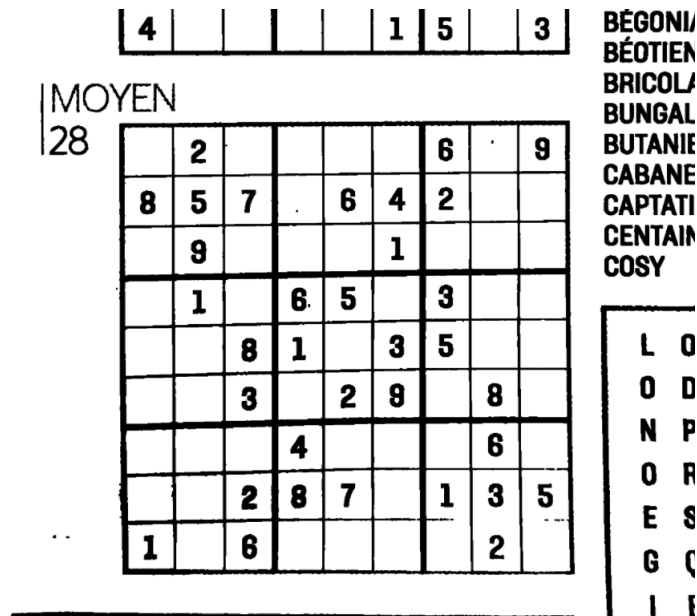


Figure 7: Noise de 7

3.5 Découpage de l'image

Pour pouvoir découper l'image afin que le réseau de neurones puisse résoudre la grille. La fonction aura besoin des coordonnées du coin en haut à gauche de l'image qu'on veut découper. Ainsi, avec la liste des 81 coordonnées de chaque case du Sudoku, on pourra découper les 81 cases pour la reconnaissance des caractères.

4 La détection de grille

La détection de la grille est une étape primordiale dans ce projet. Puisqu'elle permet de transformer n'importe quelle image de Sudoku en une quantité de petites images chacune contenant une case de la grille, que le réseau de neurones peut accepter pour les transformer en une grille de Sudoku compréhensible par un ordinateur. Il faut donc garantir que la détection, puis le découpage de la grille soient parfaits pour que le réseau de neurones puisse faire son travail. Et pour arriver à cet objectif, la solution qui nous paraissait la plus efficace était d'utiliser la transformée de Hough.

4.1 La transformée de Hough

La transformée de Hough est une méthode qui paraît simple pour la tâche qu'elle doit accomplir, mais qui reste complexe à mettre en place. L'objectif de cette méthode à l'origine, est simplement de détecter les lignes dans une image qui a été plus ou moins traitée auparavant (plus l'image a été pré-traitée plus il est facile d'implémenter l'algorithme). Mais en poussant les algorithmes et en améliorant le concept de base, il est possible de dériver la méthode pour qu'elle déduise et découpe des formes dans une image. Notre implémentations de la méthode se découpe donc en trois étapes principales : la détection des lignes au sein de l'image, la déduction de la grille en utilisant les lignes détectée et le découpage de l'image en sous-image contenant chacune soit une case contenant un chiffre soit une case vide du sudoku que nous souhaitons résoudre.

4.2 Détection des lignes

Pour détecter la grille, on utilise la base de l'algorithme de Hough. Pour cela, on commence par scanner chacun des pixels de l'image, et pour chaque pixel qui pourrait appartenir à une ligne de l'image (dans notre cas, un pixel noir) on va remplir une matrice d'accumulation (un tableau à double entrées). Une fois qu'on a fini de parcourir l'image, les valeurs les plus hautes de la matrice d'accumulation sont des lignes de l'image d'origine. La seule ombre au tableau, est que à l'intérieur d'une image on utilise les coordonnées cartésiennes pour se repérer (système classique avec deux axes), alors qu'à l'intérieur de la matrice d'accumulation qu'on utilise, on se repère en utilisant des coordonnées polaires (c'est à dire une distance par rapport à l'origine du repère et angle). Ce qui signifie que l'on doit pouvoir passer d'une base de coordonnées à une autre pour faire fonctionner l'algorithme.

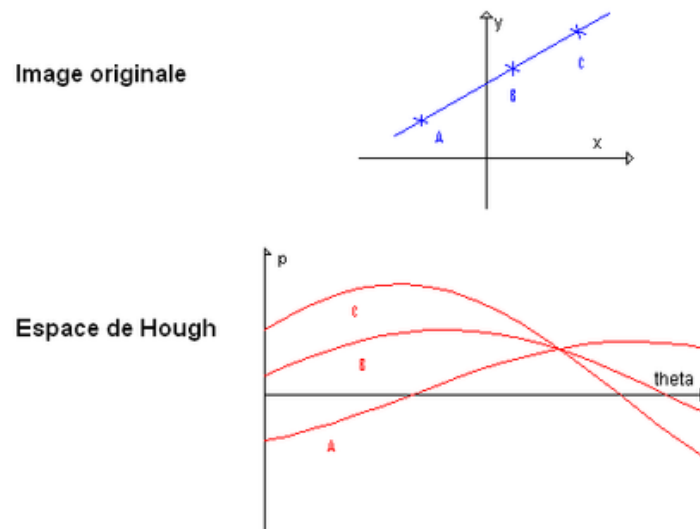


Figure 8: Une ligne dans l'image et sa représentation dans la matrice

On utilise donc la formule suivante pour passer de coordonnées cartésiennes à coordonnées polaires :

$$\begin{cases} r = \sqrt{x^2 + y^2} \\ \theta = \arctan^2(y, x) \end{cases}$$

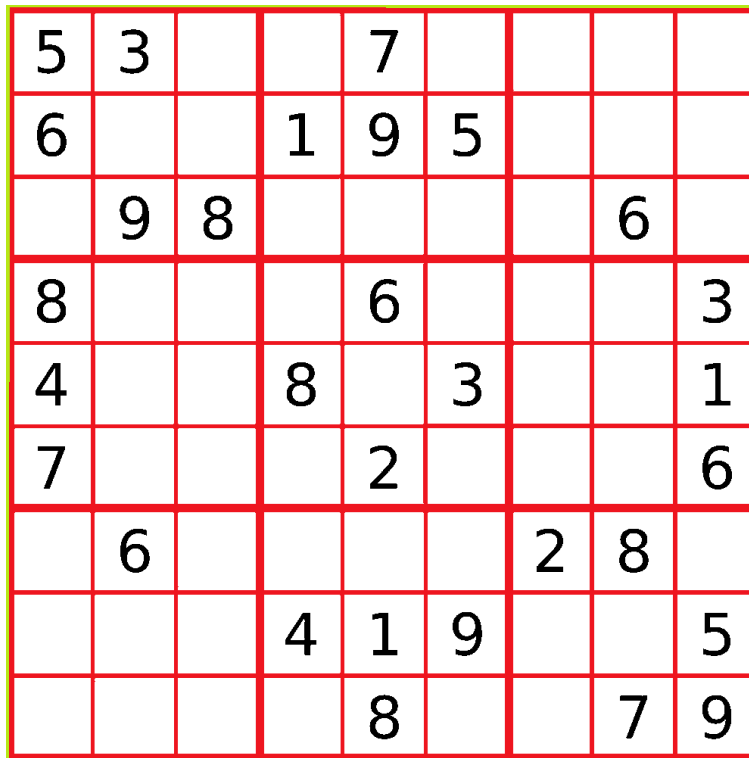
Et celle-ci pour passer de coordonnées polaires à coordonnées cartésiennes :

$$\begin{cases} x = r \cdot \cos(\theta) \\ y = r \cdot \sin(\theta) \end{cases}$$

Maintenant que l'on est capable de créer et de se repérer dans une matrice d'accumulation, on peut s'en servir pour tracer les lignes de l'image d'origine. Même si avant de tracer les lignes, on va exploiter un peu plus la matrice pour pouvoir détecter la grille de Sudoku.

4.3 Dédution de la grille

Une fois la matrice d'accumulation établie, on sait où se trouvent les lignes de l'image, mais cela ne suffit pas à détecter la grille de Sudoku. La prochaine étape est donc d'établir une liste des endroits où se croisent les lignes de l'image. Car cela nous servira directement à déduire où se trouve le plus grand carré qui entoure la grille, et donc la grille. Puisque ce carré encadre un grand nombre d'intersections, et regroupe quatre intersections alignées et les plus éloignées. Une fois que l'on sait où se trouve la grille il ne reste plus qu'à la transmettre au réseau de neurones.



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 9: Grille et lignes de l'image détectées et tracées

4.4 Découpage de l'image

Pour transmettre, la grille au réseau de neurones, il faut découper chacune des cases de la grille (ce qui résulte en 81 petites images). Pour cela, Nous utilisons la grille déduite, et en utilisant chacune des dimensions divisée par neuf, on peut retrouver la position de chacune des cases de la grille. Enfin, il reste à redimensionner les images pour qu'elle s'adapte à la taille du réseau de neurones.

5 Le réseau de neurones

5.1 Introduction

Les réseaux de neurones sont au cœur de certains grands domaines de l'informatique. On en retrouve dans la reconnaissance vocale, l'analyse d'image, dans la reconnaissance de texte et dans plein d'autres domaines.

Le modèle ressemble fortement à nos structures neuronales : on peut identifier dans notre réseau neuronal, le neurone, possédant des dendrites, les synapses, des axones par rapport au poids, biais et connexion d'un réseau de neurone artificiel. Les réseaux de neurones artificiels sont capables "d'apprendre", de mieux se paramétrer en fonction de leur utilisation.

5.2 La porte logique XOR

Avant de commencer un réseau de neurones capable de reconnaître des chiffres, nous avons fourni une preuve de concept de votre réseau de neurones. Pour cette preuve, nous avons réalisé un réseau de neurones capable d'apprendre la fonction OU EXCLUSIF. Nous avons créé un petit réseau de test pour vérifier et mettre en place tout ce que nous avons appris. Cette porte logique prend deux entrées et a une sortie. Ce réseau de neurone possède donc deux neurones dans la couche d'entrée, deux neurones sur la couche cachée et un neurone en couche de sortie : c'est un perceptron multicouche.

Table de vérité de XOR		
A	B	$R = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Figure 10: Table de vérité de l'opérateur XOR

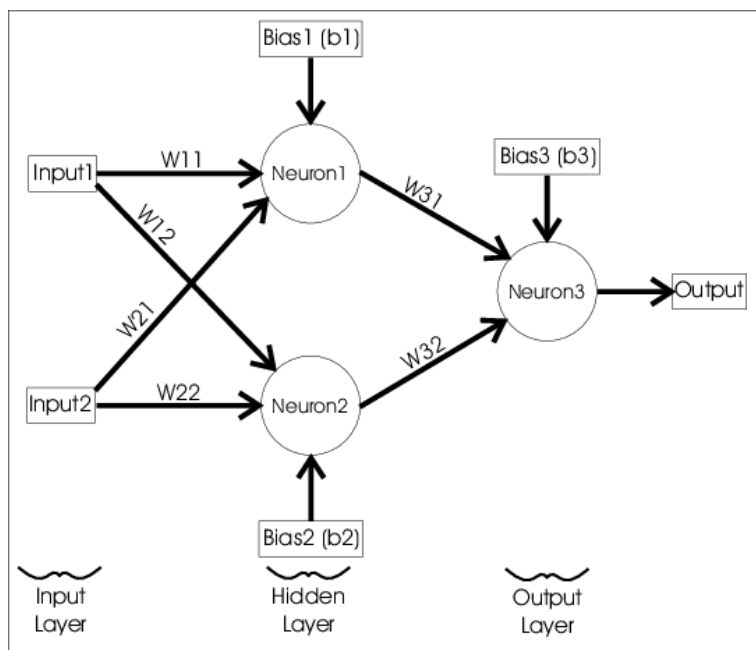


Figure 11: Architecture d'un perceptron multicouche

5.3 Implémentation

Il existe différentes manières de construire un réseau de neurone, soit avec des variables globales afin de pouvoir les retrouver dans différents fichiers, soit avec une structure, qui est un outil en C qui permet au programmeur de créer un type de variables et de lui donner un ensemble donné de données de types différent. Nous avons choisi d'utiliser les structures pour plus de clarté dans le code.

Cette structure possède plusieurs attributs : la taille de chaque couche (entrée, cachée et sortie), les tableaux pour les différentes couches, ainsi que le taux d'erreur et des caractéristiques comme le taux d'apprentissage et une chaîne de caractères qui sera la grille reconnue. Chaque couche est représentée comme un tableau de synapse, qui est un tableau de poids. Il y a également un tableau pour les biais de chaque couche.

5.4 Fonctionnement du réseau de neurones

5.4.1 Propagation avant

Le réseau de neurones prend plusieurs entrées, les traite à travers plusieurs neurones à partir de plusieurs couches cachées et renvoie le résultat à l'aide d'une couche de sortie. Ce processus d'estimation des résultats est connu sous le nom de "Propagation vers l'avant". Ensuite, nous comparons le résultat avec la sortie réelle. La tâche consiste à rendre la sortie vers le réseau de neurones aussi proche de la sortie réelle (souhaitée). Chacun de ces neurones contribue à une erreur dans la sortie finale.

Pour propager les valeurs dans la couche suivante il faut appeler une fonction d'activation pour chaque neurone. On calcule la

valeur d'un neurone avec cette formule :

$$\lambda(a) = \sum_{i=0}^n (\omega_i \lambda_i) + b$$

où ω_i est le poids n°i, λ_i est le résultat que l'on doit obtenir pour la valeur i et b est le biais du neurone.

On appelle ensuite la fonction sigmoïde avec ce résultat en paramètre qu'on propagera dans la couche suivante.

5.4.2 La fonction sigmoïde

Un neurone applique des fonctions d'activation aux entrées et aux biais. Il existe de nombreuses fonctions d'activation différentes, mais dans le cadre de la reconnaissance de caractères, la fonction sigmoïde peut suffire. Notre fonction d'activation (nommée Sigmoïde) prend la somme des entrées comme argument et renvoie la sortie du neurone.

Elle va décider si un neurone artificiel doit être activé ou non. Il aide le réseau neuronal à apprendre dans les données et aide également à normaliser la sortie de chaque neurone.

Formule de la fonction sigmoïde appliquée à notre neurone :

$$\sigma(x) = \frac{1}{1 + e^{-\sum_{i=0}^n (\omega_i \lambda_i) + b}}$$

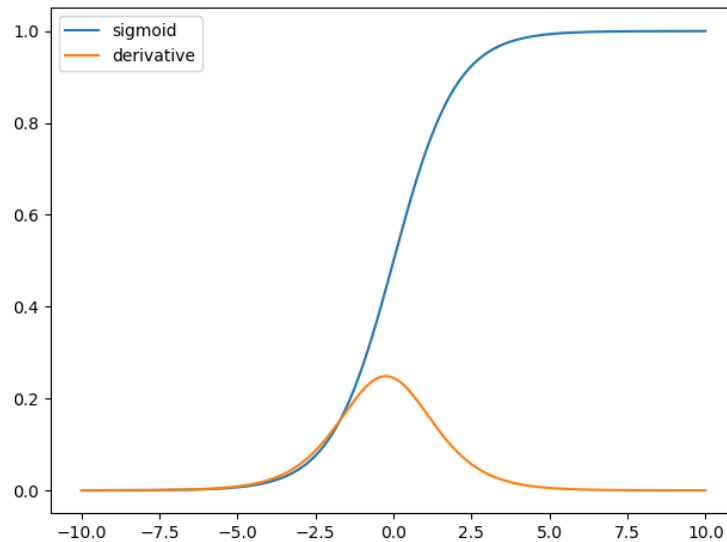


Figure 12: La fonction d'activation Sigmoide ainsi que sa dérivée

5.4.3 Rétro propagation

Le but de la rétropropagation est de calculer les dérivées partielles de la fonction de coût C par rapport à tout poids ω ou biais b dans le réseau. Une fois que nous aurons ces dérivées partielles, nous mettrons à jour les poids et les biais dans le réseau par le produit d'un certain α (aussi appelé taux d'apprentissage) et de la dérivée partielle de cette quantité par rapport à la fonction de coût. C'est l'algorithme de descente de gradient. Les dérivées partielles nous donnent la direction de la plus grande ascension. Donc, nous faisons un petit pas dans la direction opposée - la direction de la plus grande descente, c'est-à-dire la direction qui nous mènera aux minima locaux de la fonction de coût.

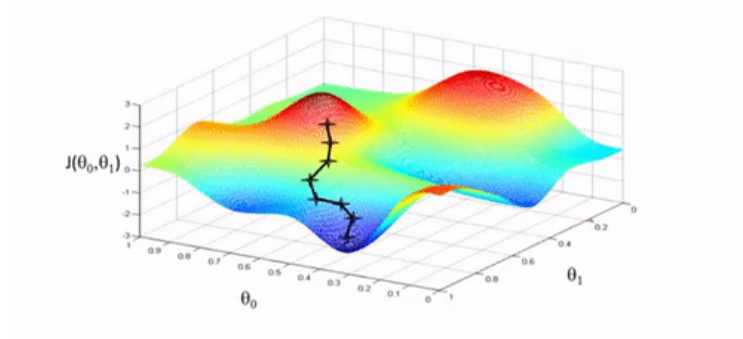


Figure 13: Visualisation de l'algorithme de descente de gradient

On peut calculer la fonction de coût pour les neurones de la couche de sortie grâce à cette formule :

$$\partial o = (g_i - o_i) * \sigma'(o_i)$$

où g_i est la sortie souhaitée du réseau neuronal et o_i est la sortie du réseau neuronal et $\sigma'(o_i)$ est la dérivée de la fonction sigmoïde.

On peut calculer la fonction de coût pour les neurones de la couche cachée grâce à cette formule :

$$\partial h = \sum_{i=0}^n (\omega_i * \Delta) * \sigma'(o_i)$$

où Δ est la fonction de coût ∂o , ω_i est le poids et $\sigma'(o_i)$ est la dérivée de la fonction sigmoïde.

Pour changer les poids entre les couches d'entrée et cachées, on utilise cette formule :

$$\omega_{ih} = \omega_{ih} + \partial h * o_i + \alpha * \omega'$$

où α est le taux d'apprentissage, ω' est la dérivée du poids

$$\omega'_{ih} = \partial h * o_i$$

Pour changer les poids entre les couches cachées et de sortie, on utilise cette formule :

$$\omega_{ho} = \omega_{ho} + \partial h * o_h + \alpha * \omega'$$

$$\omega'_{ho} = \partial h * o_h$$

Ensuite, on pourra mettre à jour les biais de la couche l (soit cachée ou sortie) avec cette formule :

$$b_l = b_l + \partial h$$

Nous essayons de minimiser la valeur/le poids des neurones qui contribuent le plus à l'erreur et cela se produit en retournant vers les neurones du réseau neuronal et en trouvant où se trouve l'erreur. Afin de réduire ce nombre d'itérations pour minimiser l'erreur, on utilise une formule nommée "Mean squared error (MSE)" :

$$\frac{1}{2} \sum_{n=0}^{10} (g_n - o_n)^2$$

où g_n est la sortie souhaitée du réseau neuronal et o_n est la sortie du réseau neuronal.

Ce cycle d'itération de propagations vers l'avant et de rétro-propagations est généralement appelé "Epochs". Lorsque le réseau aura fait Epochs fois la propagation vers l'avant et la rétro-propagation, il nous renverra ce qu'il pense être sa solution au problème.

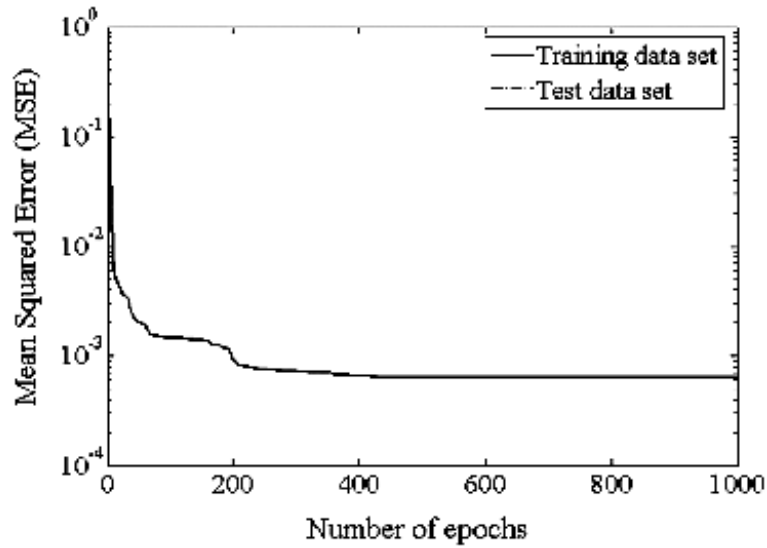


Figure 14: Graphique représentatif de la fonction MSE

5.5 Reconnaître des chiffres

Chaque image peut être traitée comme une matrice schématisant les pixels de l'image. Il y aura 1 si le pixel est noir, et 0 s'il est blanc. On dit que cette représentation est un bitmap de l'image.



Figure 15: Exemple d'un bitmap sur un 1

Chaque image que l'on envoie dans le réseau de neurone est de taille fixe : 28x28 pixels. Ainsi, notre couche d'entrée aura une taille de 784 neurones (28×28). La couche de sortie a une taille de 10, car nous avons que 10 résultats possible pour une image (la reconnaissance d'une case vide ainsi que chaque nombre de 1 à 9).

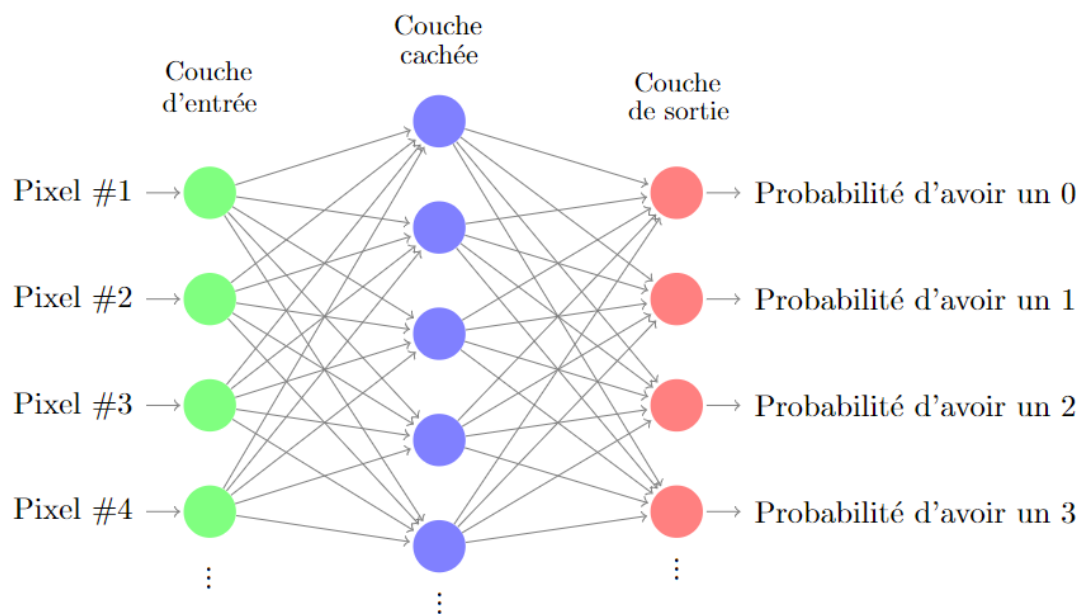


Figure 16: Représentation du réseau de neurone

5.6 Enregistrement et extraction des données

Une fois que le réseau a été testé, il peut être enregistré dans un fichier répertoriant les poids et biais de chaque neurone. Cela sera utilisé lorsqu'il faudra reconnaître un caractère : il n'y aura pas besoin de ré-entraîner le réseau de neurones pour chaque nombre qu'il doit reconnaître. La sauvegarde des données s'effectue dans 4 fichiers différents : deux fichiers pour sauvegarder les poids et biais de la couche cachée, et deux fichiers pour sauvegarder les poids et biais de la couche de sortie.

≡ biasH.b ×	≡ biasO.b ×	≡ weightHO.w ×	≡ weightIH.w ×
-0.770894	-1.327464	-1.417685	0.629725
1.208240	-2.647160	-6.054538	1.087700
4.947613	-1.545616	4.164873	0.250925
2.591711	-1.130309	1.484984	-0.237928
3.191651	-2.040550	0.525744	-0.412780
2.801198	-2.637910	-1.446154	0.098996
2.402955	-2.229476	-0.931304	-0.131326
-4.457697	-0.877113	-8.013118	-0.275921
-2.667057	-3.683831	-0.048571	-0.476698
0.418248	-2.609740	-0.064213	-0.032701
-2.386878		4.709344	-0.044804
1.162957		-3.245201	0.985956
4.895607		-1.288641	0.080728
2.167896		-0.250462	-1.215316
2.902878		-1.790606	0.786309
5.771946		-1.194406	-0.624173
4.641092		-1.266727	0.611274
2.134060		2.635436	-1.111570
-0.653139		-0.952339	1.384751
1.120276		-2.570726	0.625809
		3.344223	-0.465493
		-2.690200	0.723908
		-3.495353	0.332919
		-2.105046	-0.371103

Figure 17: Extraits des 4 fichiers sauvegardés

5.7 Entraînement

Le réseau est entraîné avec des images, contenant les caractères que l'on souhaite qu'il reconnaisse. La base de données utilisée pour l'entraînement est constituée, pour chaque chiffre, de 10 images provenant de photos de Sudoku divers et variés, et de 24 images générées grâce à un script Python où chaque nombre est écrit avec une police d'écriture différente. Cette base de données ne contient que des chiffres écrits informatiquement : aucun chiffre écrit à la main n'est présent dedans. Il y a donc 340 images dans notre base de données.

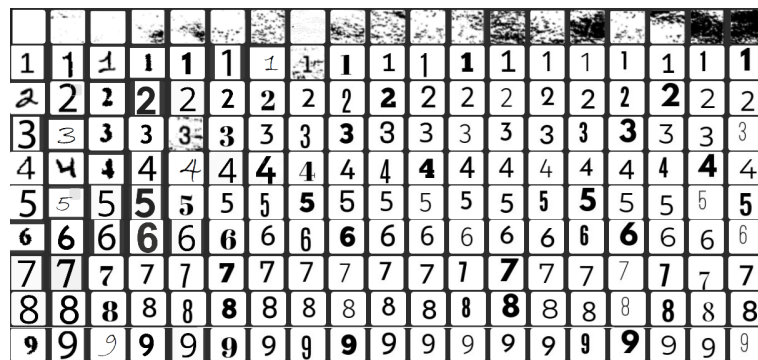


Figure 18: Petit extrait de la base de donnée

```

Number recognized: 0 | Expected 0 OK | ErrorRate: 0.000104
Number recognized: 1 | Expected 1 OK | ErrorRate: 0.000205
Number recognized: 2 | Expected 2 OK | ErrorRate: 0.000104
Number recognized: 3 | Expected 3 OK | ErrorRate: 0.000045
Number recognized: 4 | Expected 4 OK | ErrorRate: 0.000080
Number recognized: 5 | Expected 5 OK | ErrorRate: 0.000026
Number recognized: 6 | Expected 6 OK | ErrorRate: 0.000205
Number recognized: 7 | Expected 7 OK | ErrorRate: 0.000016
Number recognized: 8 | Expected 8 OK | ErrorRate: 0.000430
Number recognized: 9 | Expected 9 OK | ErrorRate: 0.000008
Epoch 2600 | MaxErrorRate = 0.000430

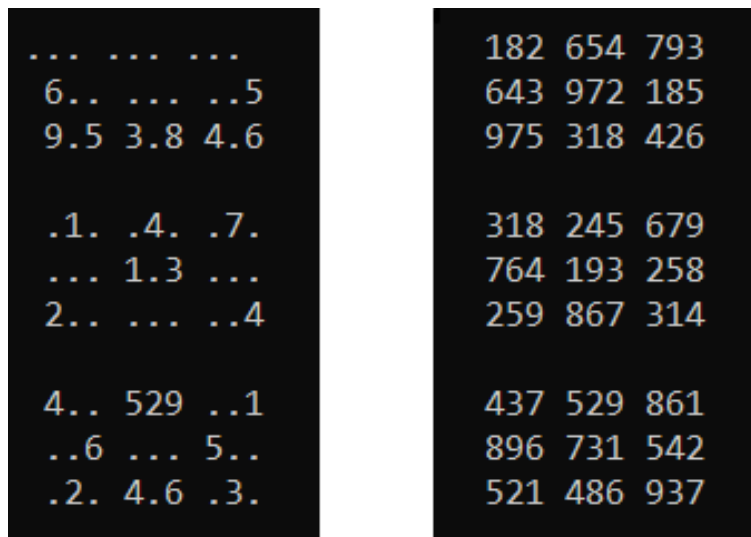
```

Figure 19: Extrait de l'affichage de l'entraînement du réseau de neurones

6 Le post-traitement

6.1 Résolution de la grille

Une fois le Sudoku récupéré, il ne reste maintenant plus qu'à en faire la résolution ainsi que de renvoyer l'image du Sudoku résolu. Pour commencer, le réseau de neurone va nous renvoyer un tableau contenant les caractères contenus dans la grille. Ensuite, on transformera ce tableau en fichier si l'utilisateur veut voir la grille sans passer par l'interface graphique.



...
6..5
9.5	3.8	4.6
.1.	.4.	.7.
...	1.3	...
2..4
4..	529	..1
..6	...	5..
.2.	4.6	.3.

182	654	793
643	972	185
975	318	426
318	245	679
764	193	258
259	867	314
437	529	861
896	731	542
521	486	937

Figure 20: Deux fichiers contenant les grilles

De plus, il faut également résoudre cette grille via un algorithme nommé "Backtracking" (Retour sur trace). Le résultat sera retranscrit en fichier texte, mais également en image.

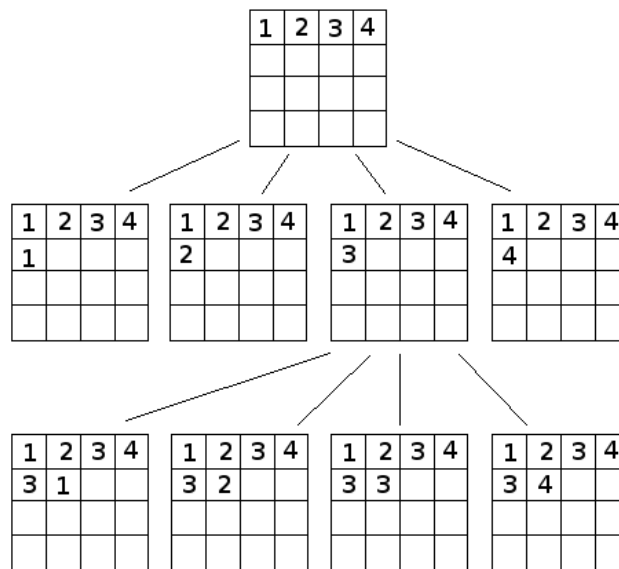


Figure 21: Représentation du Retour sur trace

Cet algorithme permet de tester systématiquement l'ensemble des affectations potentielles du problème. On va sélectionner une variable du problème, et pour chaque affectation possible de cette variable, à tester récursivement si une solution valide peut-être construite à partir de cette affectation partielle. Si aucune solution n'est trouvée, la méthode abandonne et revient sur les affectations qui auraient été faites précédemment.

Nous avons pensé à utiliser des algorithmes plus optimisés et performants, car celui-ci n'est pas des plus efficaces, car il n'est pas des plus efficaces. Un programme plus rapide serait plus avantageux pour nous.

6.2 Reconstruction de la grille

Pour la reconstruction de la grille nous avons décidé de ne pas réutiliser la grille que l'a personne nous donne mais de recréer la nôtre. Pour reconstruire le Sudoku, il faut qu'il soit rempli sous le format d'un tableau contenant chaque, chiffre et également le Sudoku avant qu'il soit résolu. Dans un premier temps la fonction va parcourir le tableau de caractères et regarder le nombre actuel afin de l'insérer dans l'image résultat. Une fois que le Sudoku résultat est rempli, on cherche quels étaient les chiffres présents avant la résolution afin de les colorier en rouge.

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

Figure 22: Reconstruction de la grille résolue

7 L'interface graphique

Concernant la mise en place de l'interface graphique, nous avons utilisé le logiciel Glade qui permet de faire des designs d'interface graphique très minimalistes. Cela permet également de faciliter la création de l'interface puisque Glade génère le code automatiquement selon ce que l'on ajoute.

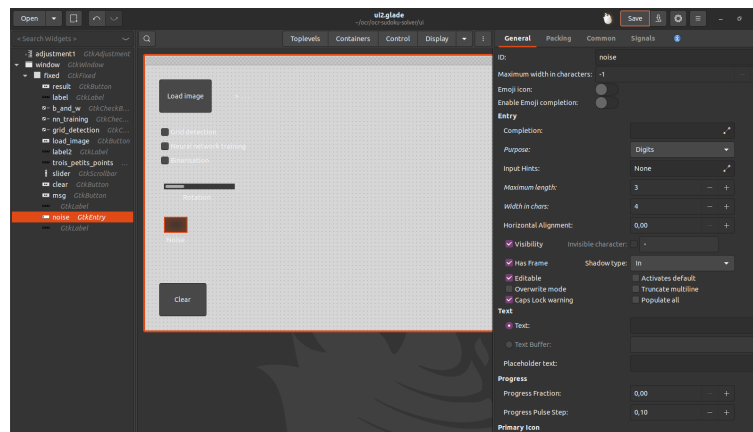


Figure 23: Glade

GTK est une librairie graphique qui permet de construire une interface graphique.

L'interface est composée de plusieurs boutons simples, boutons cochables ainsi que d'un slider et une image de fond. Le premier bouton nommé "Load image" sert à ouvrir une image via un explorateur de fichier. À partir de l'image choisie, tous nos programmes seront exécutés avec celle-ci. Ce bouton va afficher l'image choisie et le chemin pour accéder au fichier. Ce chemin apparaîtra au dessus de l'image.

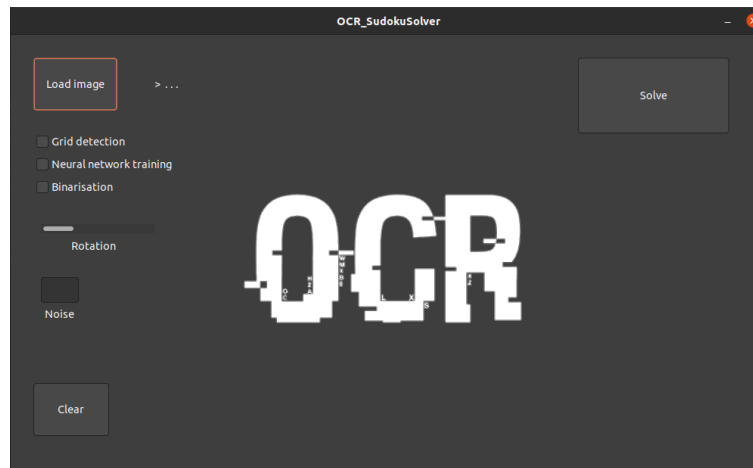


Figure 24: Interface Graphique

Ensuite, plusieurs boutons cochables vont nous servir à afficher plusieurs fonctions. L'un va nous servir à afficher une "binarisation" de l'image : elle va être transformée en image constitué uniquement de noir et de blanc et affichée à coté de l'image donnée à l'origine.

Le second bouton sert à afficher dans le terminal un aperçu de l'entraînement de notre réseau de neurones.

Le slider sert à faire une rotation manuelle de l'image. Toutes ces options donnent un aperçu du travail de notre programme, qui sera lancé grâce au bouton Solve dans le coin supérieur droite. On peut également réduire ou augmenter le bruit de l'image via un champ texte où l'on peut rentrer une valeur. Cette valeur correspond au niveau de bruit appliqué, c'est à dire que les grains ou les parasites de l'image vont être plus ou moins retirés. Les parasites de l'image sont tous les pixels qui peuvent rendre une image moins propre.

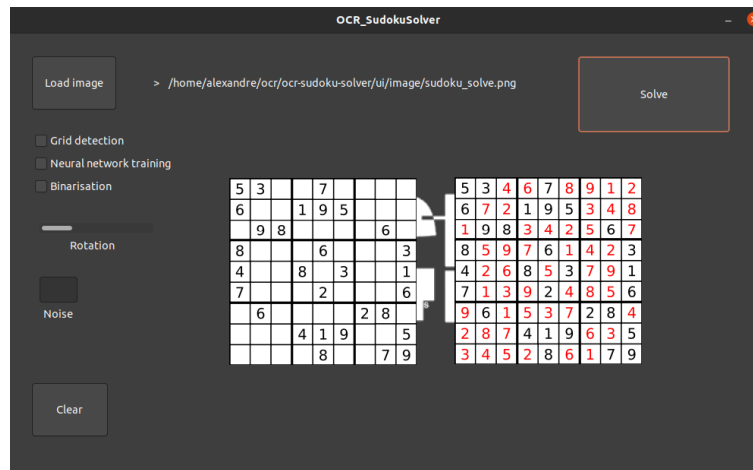


Figure 25: Interface Graphique

Le bouton "Clear", tout en bas de la fenêtre à gauche, permet d'effacer les images affichées. Nous avons donc mis plusieurs fonctionnalités de manière à facilement se rendre compte de chaque étape du projet.

Pour ajouter une touche d'originalité et nous avons placé le logo de notre projet au centre de notre interface, ce qui la rend beaucoup plus distinctive et attirante.

8 Prérequis et installation

8.1 Prérequis

Notre programme requiert plusieurs libraires afin de pouvoir le faire fonctionner :

- SDL 1.2¹
- GTK.3²
- SDL_Image³

8.2 Installation

¹<https://www.libsdl.org/download-1.2.php>

²<https://developer.gnome.org/gtk3/stable/>

³https://www.libsdl.org/projects/SDL_image/

9 Avancements et retards

Actuellement, notre programme est quasiment fini : nous avons une interface graphique présentant toutes nos fonctionnalités (Traitement de l'image, reconnaissance des chiffres, résolution du Sudoku et reconstruction de la grille lorsque qu'elle est résolue). Seul bémol à ce jour : la détection de la grille, ainsi que la récupération des chiffres présents dans les cases ne sont pas assez fiables.

10 Conclusion

Notre projet s'est plutôt bien passé, le résultat nous satisfait dans l'ensemble, même si certaines choses auraient pu être améliorées. Ce projet nous a beaucoup plu, et nous a permis de d'apprendre beaucoup de choses sur les réseaux de neurones et sur certaines méthodes de traitement d'image.