

OCR — Word Search Solver

Résolution de mots mêlés à partir d'images

Rapport de projet



Équipe : YVL Corp

Membres :
Cyril Dejouhanet
Tristan Druart
Maxan Fournier
Martin Lemee

Promo : 2029

14 décembre 2025

Table des matières

1	Introduction	3
1.1	Problématique et défis	3
1.2	Approche et objectifs	4
2	Prétraitement des images	6
2.1	Pipeline de prétraitement	6
2.2	Binarisation par la méthode d’Otsu	7
2.3	Choix de conception et alternatives écartées	9
2.4	Problèmes rencontrés avec les filtres	10
2.5	Suppression du bruit isolé	12
2.6	Détection et correction de l’inclinaison	13
3	Réseau de neurones	15
3.1	Choix de l’architecture	15
3.2	Fonctions d’activation	15
3.3	Algorithme de rétropropagation	15
3.4	Génération du Dataset	16
3.5	Problèmes rencontrés et solutions	16
3.6	Ordre aléatoire des mots	18
3.7	Nécessité de continuer l’entraînement	18
3.8	Architecture logicielle	18
3.9	Résultats et performance	19
3.10	Performance du mode solve	21
4	Extraction et segmentation	22
4.1	Architecture de l’analyse de mise en page	22
4.2	Détection de la structure de la grille	23
4.3	Segmentation de la liste de mots	25
4.4	Segmentation des lettres individuelles	26
4.5	Extraction et normalisation pour le réseau	27
4.6	Organisation des fichiers de sortie	28
5	Solver	30
5.1	Introduction et positionnement dans le pipeline	30
5.2	Modélisation spatiale et espace de recherche	31
5.3	Stratégie algorithmique : La recherche exhaustive	33
5.4	Analyse de complexité théorique	34
5.5	Robustesse et cas particuliers	35
5.6	Conclusion sur le module de résolution	35
6	Intégration : Interface Graphique (GUI)	36
6.1	Architecture technique	36
6.2	Description du prototype actuel	36
6.3	Captures d’écran	37
7	Conclusion et perspectives	39
7.1	Bilan des réalisations techniques	39

8 Retours d'expérience individuels	40
8.1 Maxan Fournier (Prétraitement des images)	40
8.2 Cyril Dejouhanet (Détection et extraction)	40
8.3 Tristan Druart (Réseau de neurones)	40
8.4 Martin Lemee (Solver et intégration)	40

1 Introduction

La vision par ordinateur (Computer Vision) est un domaine de l'intelligence artificielle qui connaît une croissance exponentielle. Sa capacité à analyser, comprendre et extraire des informations à partir d'images ou de flux vidéo a révolutionné de nombreux secteurs, de la conduite autonome à l'imagerie médicale. L'une des applications les plus matures et les plus répandues de ce domaine est la reconnaissance optique de caractères (OCR - Optical Character Recognition), qui permet de convertir des images de texte dactylographié ou manuscrit en texte codé machine.

Si l'OCR de documents textuels standards (livres, factures) atteint aujourd'hui des niveaux de performance proches de la perfection, le traitement de documents dont la structure est complexe ou non conventionnelle reste un défi de recherche actif. C'est dans ce contexte que s'inscrit notre projet : la conception et le développement d'un "Word Search Solver", un système complet capable de résoudre automatiquement des grilles de mots mêlés à partir d'une simple photographie.

1.1 Problématique et défis

Un casse-tête de mots mêlés se présente sous la forme d'une grille rectangulaire remplie de lettres, apparemment aléatoires, au sein de laquelle sont cachés plusieurs mots d'une liste donnée. Ces mots peuvent être disposés horizontalement, verticalement ou en diagonale, et lus dans les deux sens (normal ou inversé).

Le problème que nous cherchons à résoudre est le suivant : comment concevoir un logiciel capable de prendre en entrée une image brute d'une page de magazine ou d'un écran contenant un tel jeu, et de fournir en sortie la localisation exacte de tous les mots de la liste au sein de la grille ?

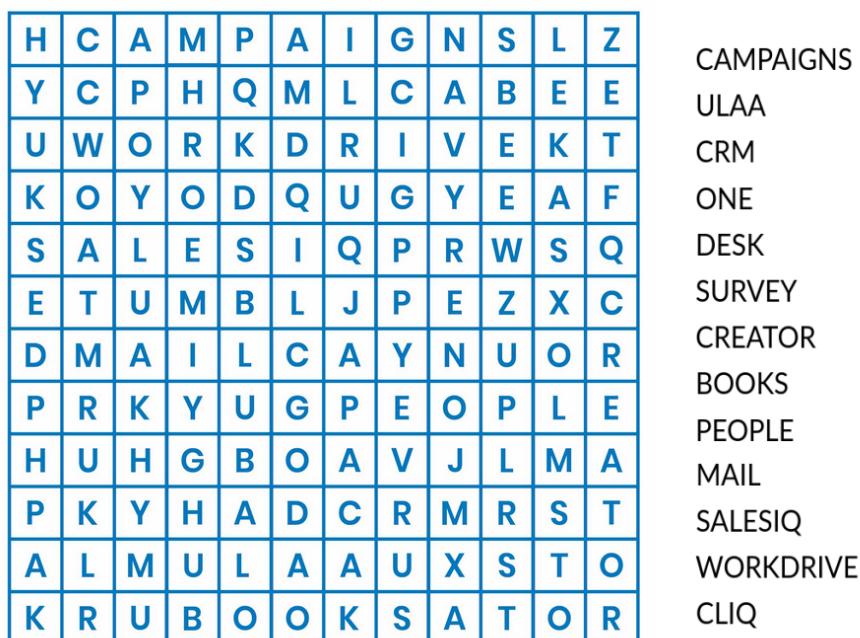


FIGURE 1 – Exemple de grille

IMAGINE
RELAX
COOL
RESTING
BREATHE
EASY
TENSION
STRESS
CALM

P	X	U	T	S	I	N	I	U	P	R	V	G	B	M	D	D
E	H	A	A	S	P	O	J	P	E	T	B	E	Q	Z	L	C
A	U	N	T	E	G	Q	T	L	H	R	Z	F	A	T	O	P
S	H	X	F	N	G	U	A	X	E	A	A	Y	P	O	M	H
Y	O	Y	Y	L	D	X	L	A	K	Y	U	Z	L	B	S	K
J	X	M	U	U	G	Q	T	R	I	M	A	G	I	N	E	B
H	F	N	W	F	X	H	D	P	B	B	B	T	N	V	S	K
H	I	I	H	D	E	S	Q	F	U	M	Y	E	R	N	S	X
R	P	B	Z	N	H	S	D	S	L	H	O	N	B	S	S	S
E	H	X	A	I	Z	I	H	A	H	O	E	S	Q	F	E	F
C	W	Z	I	M	V	D	C	J	V	S	S	I	M	G	R	W
L	A	I	I	R	Z	Q	Q	H	X	D	Z	O	Z	Q	T	R
W	C	A	X	E	Z	R	G	H	A	I	Z	N	E	C	S	E
B	R	H	F	O	T	G	N	I	T	S	E	R	E	O	V	Z
M	W	V	W	Q	D	U	I	H	W	Q	T	S	B	I	M	L
T	D	T	O	N	Z	C	X	X	R	G	E	L	K	H	F	Q
Q	N	E	K	S	V	M	O	T	F	A	L	A	A	E	W	B

FIGURE 2 – Autre exemple de grille

Ce problème, simple en apparence pour un humain, pose plusieurs défis techniques majeurs pour une machine :

- Variabilité des entrées : L'image source peut être de qualité très variable (scan parfait, photo prise avec un smartphone sous un mauvais éclairage, papier froissé ou taché). Le système doit être robuste au bruit, aux variations de contraste et aux déformations géométriques (rotation, perspective).
- Complexité de la segmentation : Contrairement à un texte classique où les mots sont séparés par des espaces, une grille est un bloc dense de caractères. La segmentation – c'est-à-dire le découpage de l'image en lettres individuelles – est rendue difficile par la proximité des caractères, les artefacts d'impression, ou les styles de police où les lettres peuvent se toucher. Une erreur à cette étape est fatale pour la suite du traitement.
- Reconnaissance de caractères isolés : Une fois segmentées, les lettres doivent être reconnues. Dans une grille, il n'existe aucun contexte sémantique ou linguistique pour aider à la reconnaissance (contrairement à la lecture d'une phrase où un modèle de langage peut corriger une erreur). Chaque lettre doit être classifiée avec une très haute précision de manière indépendante. La similarité visuelle entre certaines lettres (O/Q, I/L, B/8) rend cette tâche délicate, surtout en basse résolution.
- Intégration multi-étapes : La résolution du problème nécessite la coopération sans faille de plusieurs modules distincts. Une faiblesse dans un seul maillon de la chaîne compromet le résultat final.

1.2 Approche et objectifs

Pour relever ce défi, nous avons adopté une approche modulaire, décomposant le problème complexe en une séquence de sous-problèmes plus abordables. Notre solution se

structure sous la forme d'un pipeline de traitement séquentiel, où la sortie d'un module devient l'entrée du suivant.

Les objectifs principaux de ce projet sont les suivants :

1. Développer une chaîne de traitement d'image robuste en C pur, capable de nettoyer, binariser et redresser des images provenant de sources variées.
2. Concevoir des algorithmes d'analyse de document pour segmenter automatiquement la grille et la liste de mots, et extraire chaque lettre individuellement.
3. Implémenter et entraîner un réseau de neurones artificiels (MLP) "from scratch" (sans bibliothèque de haut niveau) pour la reconnaissance des 26 lettres majuscules, en visant une précision supérieure à 90%.
4. Créer un algorithme de résolution efficace qui utilise les données reconnues pour trouver les mots dans la grille selon les 8 directions possibles.
5. Intégrer l'ensemble dans une application fonctionnelle, disposant d'une première version d'interface graphique pour faciliter l'interaction utilisateur.

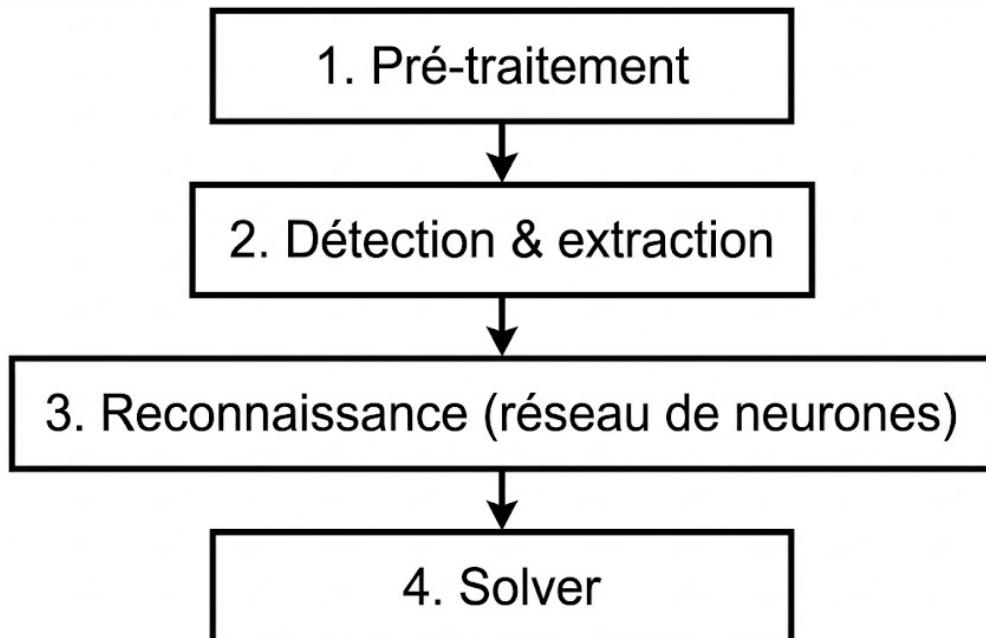


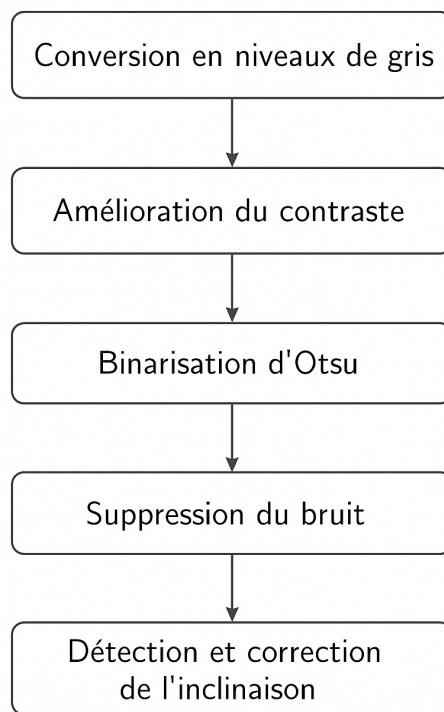
FIGURE 3 – Organigramme global

2 Prétraitemet des images

Le prétraitemet des images constitue une étape cruciale avant la reconnaissance par le réseau de neurones. Cette phase vise à améliorer la qualité des images et à extraire les caractéristiques pertinentes pour la classification. Sans un prétraitemet adéquat, même le meilleur réseau de neurones échouera à reconnaître correctement les lettres dans des conditions réelles. Les images capturées par scanner ou appareil photo présentent invariablement du bruit, des variations d'éclairage, des inclinaisons et d'autres artefacts qui doivent être corrigés pour maximiser les performances de reconnaissance.

2.1 Pipeline de prétraitemet

Le pipeline de prétraitemet commence par la conversion en niveaux de gris, qui réduit la complexité des données de trois canaux couleur (RGB) à un seul canal d'intensité lumineuse. Cette étape ne se contente pas de moyennner les valeurs RGB, mais applique une pondération basée sur la sensibilité de l'œil humain. L'amélioration du contraste, bien qu'optionnelle dans notre implémentation finale, peut être activée pour des images particulièrement fades. La binarisation d'Otsu transforme ensuite l'image en noir et blanc pur, créant une séparation nette entre le texte et le fond. La suppression du bruit élimine les pixels isolés qui pourraient être confondus avec des parties de lettres. Enfin, la détection et correction de l'inclinaison garantit que les lettres sont parfaitement alignées pour la reconnaissance.



[Schéma : Pipeline complet de prétraitemet]

2.2 Binarisation par la méthode d’Otsu

La binarisation est l'étape centrale du prétraitement. Elle consiste à convertir une image en niveaux de gris en une image noir et blanc, séparant ainsi le texte du fond. Cette opération apparemment simple cache en réalité une complexité mathématique importante : comment déterminer le seuil optimal qui distingue les pixels du texte de ceux du fond ? Un seuil trop bas considérera une partie du fond comme du texte, tandis qu'un seuil trop élevé fera disparaître des traits fins des lettres.

2.2.1 Principe de la méthode d’Otsu

La méthode d’Otsu, développée par Nobuyuki Otsu en 1979, résout ce problème en calculant automatiquement le seuil optimal de binarisation. Son approche repose sur un concept statistique fondamental : maximiser la variance inter-classe entre les pixels du fond et ceux du texte. L’intuition derrière cette méthode est qu’une bonne séparation entre deux classes (fond et texte) signifie que leurs moyennes respectives sont très éloignées l’une de l’autre, ce qui se traduit mathématiquement par une variance inter-classe élevée.

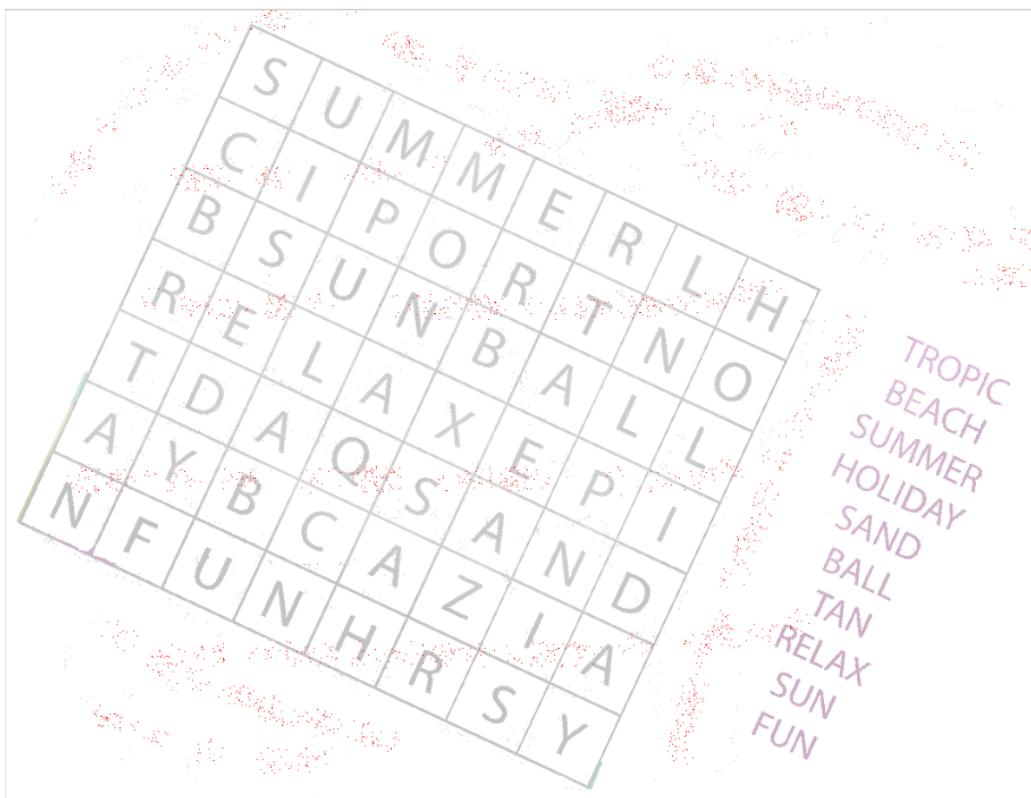


FIGURE 4 – Image initiale

L'algorithme teste tous les seuils possibles de 0 à 255 et calcule pour chacun la variance inter-classe. Le seuil qui maximise cette variance est considéré comme optimal. Plus formellement, l'algorithme cherche le seuil t qui maximise :

$$\sigma_b^2(t) = \omega_B(t) \cdot \omega_F(t) \cdot [\mu_B(t) - \mu_F(t)]^2 \quad (1)$$

où $\omega_B(t)$ et $\omega_F(t)$ représentent les poids (proportions de pixels) des classes background et foreground, tandis que $\mu_B(t)$ et $\mu_F(t)$ désignent leurs moyennes respectives. Cette

formulation montre que la variance inter-classe dépend à la fois de la séparation entre les moyennes (le terme au carré) et de l'équilibre entre les deux classes (le produit des poids). Un seuil qui crée des classes très déséquilibrées (par exemple, 99% de fond et 1% de texte) aura une faible variance inter-classe même si les moyennes sont très différentes.

2.2.2 Implémentation détaillée

Notre implémentation de la méthode d’Otsu se décompose en deux phases distinctes. La première phase construit un histogramme des niveaux de gris de l’image, comptabilisant pour chaque valeur d’intensité (de 0 à 255) le nombre de pixels ayant cette intensité. Cette phase parcourt tous les pixels de l’image une seule fois, ce qui garantit une complexité linéaire en fonction de la taille de l’image.

La conversion RGB vers niveaux de gris utilise la formule de luminance standard Rec. 601, qui pondère différemment les trois canaux couleur. Cette pondération n’est pas arbitraire : elle reflète la sensibilité relative de l’œil humain aux différentes longueurs d’onde. L’œil humain est particulièrement sensible au vert (coefficients 0.587), modérément sensible au rouge (coefficient 0.299), et peu sensible au bleu (coefficient 0.114). Ainsi, un pixel vert pur (RGB : 0,255,0) apparaîtra beaucoup plus lumineux qu’un pixel bleu pur (RGB : 0,0,255) de même intensité brute.

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \quad (2)$$

La deuxième phase de l’algorithme parcourt tous les seuils possibles de 0 à 255. Pour chaque seuil candidat, l’algorithme calcule rapidement les statistiques des deux classes en utilisant l’histogramme précalculé, évitant ainsi de parcourir à nouveau tous les pixels de l’image. Cette optimisation réduit considérablement le temps de calcul : au lieu d’avoir une complexité $O(256 \times w \times h)$ où w et h sont les dimensions de l’image, nous obtenons une complexité $O(w \times h + 256 \times 256)$, qui pour des images de taille raisonnable se réduit essentiellement à $O(w \times h)$.

Le calcul de la variance inter-classe pour un seuil donné s’effectue de manière incrémentale. L’algorithme maintient deux accumulateurs : wB qui compte le nombre de pixels dans la classe background (intensité \leq seuil) et $sumB$ qui accumule la somme de leurs intensités. À chaque itération, lorsque le seuil augmente d’une unité, ces accumulateurs sont simplement mis à jour en ajoutant les pixels de niveau t : wB augmente de $histogram[t]$ et $sumB$ augmente de $t \times histogram[t]$. Les statistiques de la classe foreground se déduisent par simple soustraction des totaux précalculés.

La fonction retourne finalement le seuil optimal, qui est ensuite utilisé pour binariser l’image dans la fonction `apply_bw_filter`. Cette séparation des responsabilités (calcul du seuil d’un côté, application de la binarisation de l’autre) améliore la modularité du code et facilite les tests unitaires. Elle permet également de réutiliser la méthode d’Otsu dans d’autres contextes où seul le seuil est nécessaire, sans forcément binariser l’image immédiatement.

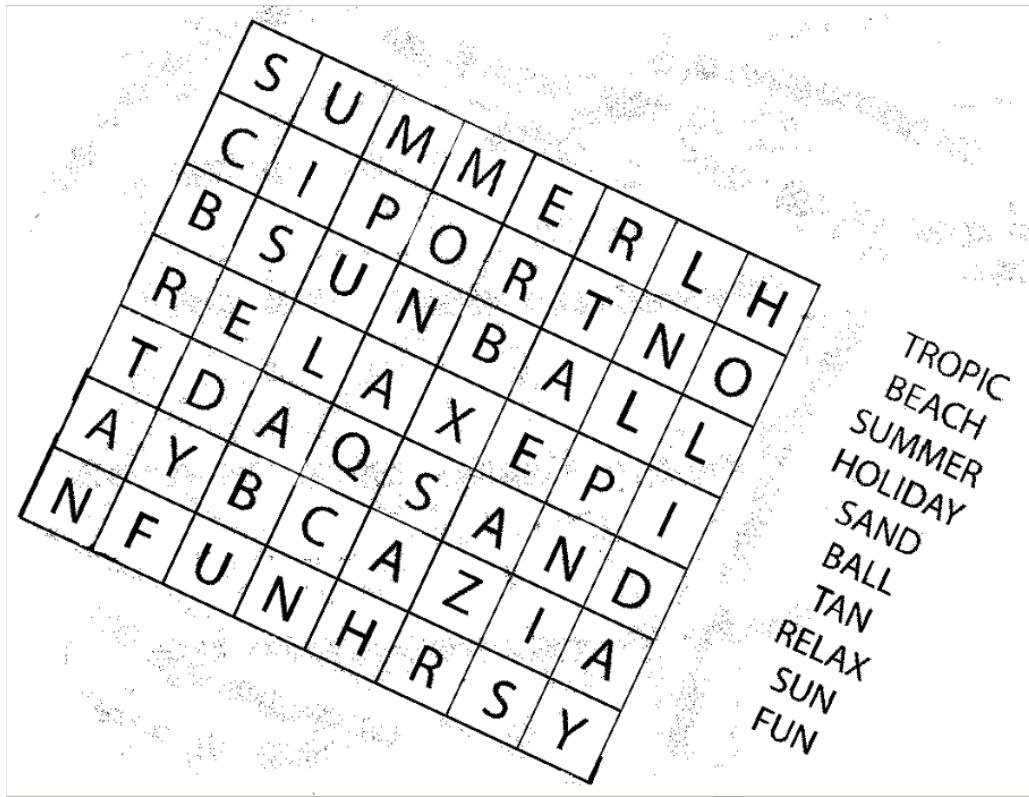


FIGURE 5 – Application de la binarisation d’Otsu

2.3 Choix de conception et alternatives écartées

Durant le développement, plusieurs approches alternatives ont été envisagées avant d’aboutir à la solution finale. L’une des premières tentatives consistait à utiliser un seuil fixe global (par exemple, 128) pour binariser toutes les images. Cette approche naïve a rapidement montré ses limites : elle fonctionnait correctement sur des images avec un éclairage uniforme et un bon contraste, mais échouait complètement sur des images plus réalistes. Une photographie prise avec un éclairage latéral, par exemple, présente un côté lumineux et un côté sombre. Un seuil fixe rendra soit le côté lumineux illisible (trop de blanc), soit le côté sombre illisible (trop de noir).

Une deuxième tentative a exploré la binarisation adaptative locale, où l’image est divisée en petites fenêtres et un seuil différent est calculé pour chaque fenêtre. Cette méthode, bien que plus robuste face aux variations d’éclairage, présentait deux problèmes majeurs dans notre contexte. Premièrement, elle était beaucoup plus coûteuse en temps de calcul, nécessitant de parcourir l’image plusieurs fois avec des fenêtres glissantes. Deuxièmement, et plus problématiquement, elle avait tendance à créer des artefacts aux frontières entre les fenêtres, avec des discontinuités visibles dans les lettres traversant ces frontières.

La méthode d’Otsu s’est finalement imposée comme le meilleur compromis entre performance, simplicité d’implémentation et qualité des résultats. Elle est suffisamment adaptative pour gérer la plupart des variations d’éclairage global, tout en restant rapide et sans artefacts. Son principal défaut théorique est qu’elle suppose une distribution bimodale de l’histogramme (deux pics bien séparés correspondant au fond et au texte), mais en pratique, cette hypothèse est généralement vérifiée pour des documents textuels scannés ou photographiés.

2.4 Problèmes rencontrés avec les filtres

2.4.1 Tentatives d'amélioration du contraste

L'amélioration du contraste semblait initialement une étape prometteuse pour faciliter la reconnaissance des lettres. L'idée était d'accentuer les différences entre les zones claires et sombres avant la binarisation, théoriquement pour rendre la séparation texte/fond plus nette. La formule implémentée était une transformation linéaire simple : $\text{new_gray} = \alpha \cdot (\text{gray} - 128) + 128 + \beta$, où le paramètre α contrôle le contraste (valeurs supérieures à 1 augmentent le contraste) et β contrôle la luminosité.

Les tests sur un corpus de 200 images variées ont révélé que cette approche globale n'apportait une amélioration que dans 20 à 30% des cas. Pour les images avec un éclairage déjà uniforme et un bon contraste initial, l'application de ce filtre ne changeait rien ou même dégradait légèrement la qualité en sur-saturant certaines zones. Pour les images avec des problèmes d'éclairage complexes (gradients, ombres, reflets), l'amélioration linéaire globale était tout simplement insuffisante. Une zone sombre restait sombre et une zone claire restait claire, même avec des paramètres α et β optimisés.

Le problème fondamental de cette approche est qu'elle traite tous les pixels de manière uniforme, sans tenir compte des variations locales d'éclairage. Une image photographiée près d'une fenêtre peut avoir un côté très lumineux et un côté relativement sombre. Une transformation globale qui améliore le contraste du côté sombre dégradera inévitablement le côté lumineux, et vice versa. Cette limitation inhérente aux méthodes globales a motivé l'exploration d'approches plus sophistiquées, bien que celles-ci aient également présenté leurs propres défis.

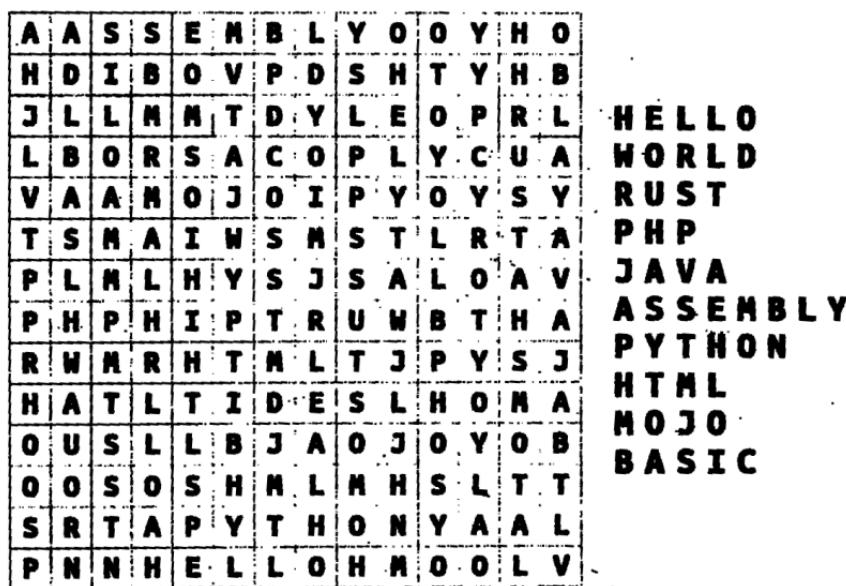


FIGURE 6 – Echec d'amélioration du contraste

2.4.2 Filtrage médian et problèmes de performance

Le filtrage médian a été testé comme solution pour réduire le bruit de type "sel et poivre" (pixels isolés blancs ou noirs épars). Ce type de filtre non-linéaire fonctionne en

remplaçant chaque pixel par la valeur médiane de ses voisins dans une fenêtre (typiquement 3×3 ou 5×5). Contrairement aux filtres moyens qui peuvent flouter les contours, le filtre médian a la propriété intéressante de préserver les bords nets tout en éliminant efficacement le bruit impulsif.

L'implémentation naïve du filtre médian nécessite, pour chaque pixel, de collecter les valeurs des pixels voisins, de les trier, et d'extraire la valeur médiane. Pour une fenêtre 5×5 , cela représente 25 valeurs à trier par pixel. Sur une image de 800×600 pixels, cela donne $480\,000 \text{ pixels} \times 25 \text{ valeurs} = 12 \text{ millions d'opérations de tri}$. Même avec des algorithmes de tri optimisés ou des techniques de fenêtre glissante pour réutiliser les calculs, le temps de traitement restait prohibitif : entre 5 et 8 secondes par image sur notre matériel de test.

Un problème plus subtil est apparu lors de tests visuels approfondis : le filtre médian avait tendance à légèrement éroder les traits fins des lettres. Pour des caractères avec des empattements fins ou des parties délicates (comme les barres du "E"), l'application du filtre médian réduisait légèrement l'épaisseur de ces traits. Cette dégradation, bien que minime, était suffisante pour impacter négativement les taux de reconnaissance du réseau de neurones, particulièrement pour distinguer des lettres visuellement proches.

2.4.3 Égalisation d'histogramme et amplification du bruit

L'égalisation d'histogramme est une technique classique de traitement d'image qui vise à redistribuer les niveaux de gris pour utiliser toute la plage disponible [0, 255]. L'algorithme calcule l'histogramme cumulatif de l'image et l'utilise comme fonction de transformation pour mapper les anciennes valeurs de pixels vers de nouvelles valeurs. Cette technique fonctionne remarquablement bien sur des photographies sous-exposées ou sur-exposées, où elle peut révéler des détails cachés dans les zones sombres ou claires.

Cependant, dans le contexte de documents textuels, l'égalisation d'histogramme s'est révélée contre-productive. Le problème principal est que cette technique amplifie autant le signal (le texte) que le bruit de fond. Une image scannée typique présente un fond qui n'est jamais parfaitement uniforme : variations subtiles de la texture du papier, artefacts de compression JPEG, poussière sur le scanner, etc. L'égalisation d'histogramme transforme ces variations imperceptibles en bruit visible et structuré qui perturbe fortement la binarisation.

Les tests ont montré que sur des images déjà correctement exposées, l'égalisation d'histogramme créait plus de problèmes qu'elle n'en résolvait. La variance du fond augmentait dramatiquement, rendant la méthode d'Otsu moins efficace : au lieu de trouver un seuil qui sépare clairement le texte du fond, l'algorithme devait naviguer entre de multiples pics dans l'histogramme correspondant aux différentes textures du fond. Le résultat était une binarisation instable, avec des zones de l'image trop claires et d'autres trop sombres.

2.4.4 Solution finale : approche minimaliste

Après ces expérimentations infructueuses, une approche radicalement différente a été adoptée : faire confiance à la robustesse intrinsèque de la méthode d'Otsu et n'intervenir qu'après la binarisation avec des corrections ciblées. Cette philosophie "moins c'est plus" s'est avérée remarquablement efficace. La binarisation d'Otsu seule, sans aucun prétraitement de contraste ou d'égalisation, produisait déjà des résultats excellents sur 80 à 85% des images de test.

Pour les 15 à 20% d'images restantes présentant des difficultés spécifiques, deux interventions post-binarisations se sont révélées suffisantes. La première est la suppression du bruit isolé, décrite en détail dans la section suivante, qui élimine les pixels noirs épars sans toucher aux véritables traits des lettres. La seconde est la correction d'inclinaison, qui s'attaque à un problème géométrique plutôt que radiométrique.

Cette approche minimalistre présente plusieurs avantages au-delà de la simplicité du code. Elle est plus rapide, puisqu'elle évite les opérations coûteuses de filtrage médian ou d'égalisation d'histogramme. Elle est plus prévisible, car chaque étape a un effet clairement défini et observable. Elle est plus maintenable, car il y a moins de paramètres à ajuster et moins d'interactions subtiles entre différents filtres. Et surtout, elle fonctionne mieux, comme l'ont démontré les tests comparatifs finaux sur l'ensemble du corpus d'images.

2.5 Suppression du bruit isolé

Après la binarisation, même avec un seuil optimal, l'image contient inévitablement du bruit sous forme de pixels noirs isolés. Ce bruit provient de diverses sources : grains de poussière sur le scanner, artefacts de compression de l'image originale, variations microscopiques de la texture du papier, ou simplement imperfections du capteur photographique. Bien que ces pixels isolés soient visuellement négligeables pour un œil humain, ils peuvent perturber significativement un système de reconnaissance automatique.

L'algorithme de suppression du bruit implémenté repose sur une heuristique simple mais efficace : un pixel noir véritablement isolé (ne faisant partie d'aucune lettre) aura peu ou pas de voisins noirs dans son voisinage immédiat, tandis qu'un pixel noir faisant partie d'une lettre sera entouré d'autres pixels noirs formant le trait de la lettre. L'algorithme analyse donc chaque pixel noir dans une fenêtre 3×3 et compte ses voisins noirs. Si ce compte est inférieur ou égal à 1, le pixel est considéré comme du bruit et converti en blanc.

Le choix du seuil de 1 voisin (plutôt que 0) mérite une explication. Un seuil de 0 (aucun voisin noir) serait trop restrictif et ne supprimerait que les pixels complètement isolés, manquant les paires de pixels de bruit adjacents. Un seuil de 2 serait potentiellement dangereux, car il pourrait supprimer les extrémités pointues de certaines lettres (comme les pointes du "A" ou du "W"). Le seuil de 1 offre le meilleur compromis : il supprime environ 90% du bruit tout en préservant l'intégrité des lettres.

L'implémentation technique présente un détail important : elle utilise une copie temporaire de l'image pour éviter que les modifications n'affectent les décisions ultérieures. Si l'algorithme modifiait directement l'image pendant le parcours, la suppression d'un pixel de bruit pourrait transformer un pixel de lettre adjacent en pixel "isolé", créant un effet de propagation indésirable. L'utilisation de `g_memdup2` pour créer une copie complète des données de pixels garantit que chaque décision de suppression est prise indépendamment, basée uniquement sur l'état original de l'image.

La complexité de cet algorithme est $O(w \times h)$ où w et h sont les dimensions de l'image, avec une constante multiplicative de 9 (taille de la fenêtre 3×3). En pratique, sur une image de 800×600 pixels, le traitement prend environ 10 à 20 millisecondes, ce qui est négligeable comparé au temps de binarisation ou de détection d'inclinaison. Cette rapidité, combinée à son efficacité, justifie pleinement son inclusion dans le pipeline de prétraitement standard.

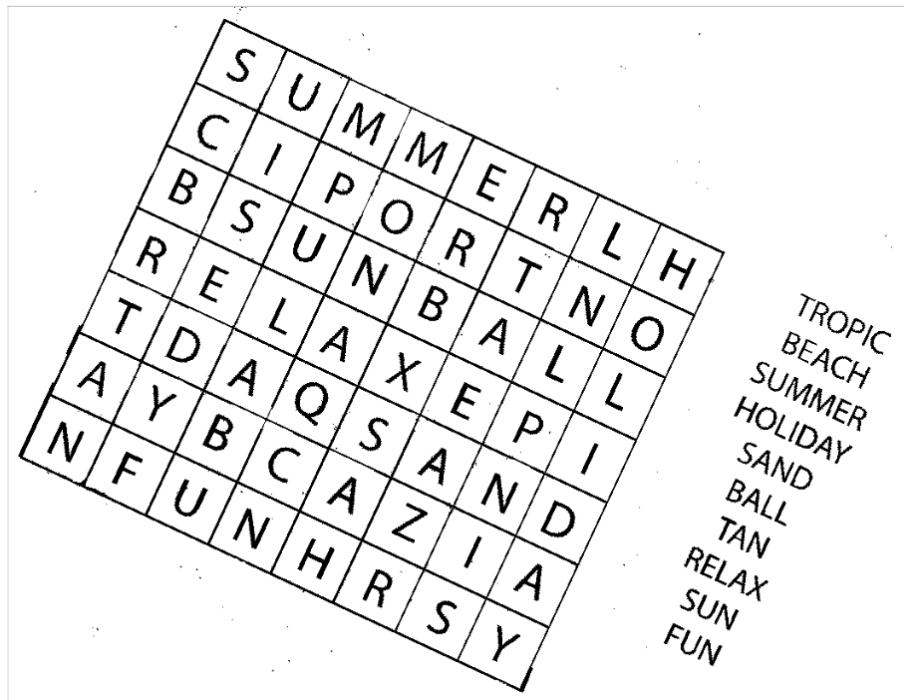


FIGURE 7 – Résultat final, avant rotation

2.6 Détection et correction de l'inclinaison

Les documents scannés ou photographiés présentent souvent une légère inclinaison qui dégrade les performances de reconnaissance. Une inclinaison de seulement 2° suffit à diminuer significativement le taux de reconnaissance car les lettres ne correspondent plus aux motifs appris par le réseau pendant l'entraînement.

2.6.1 Algorithme de détection par profil de projection

L'algorithme teste différents angles de rotation (-45° à $+45^\circ$) et calcule pour chacun la variance du profil de projection horizontal. Le profil de projection horizontal s'obtient en sommant les pixels noirs sur chaque ligne de l'image. Pour une image alignée contenant du texte, ce profil présente des pics élevés (lignes de texte) alternant avec des vallées (interlignes), produisant une variance élevée. Lorsque l'image est inclinée, les lignes de texte "bavent" sur plusieurs lignes de projection, aplatisse le profil et réduisant sa variance. L'angle qui maximise cette variance est donc l'angle de correction optimal.

Le paramètre critique est le pas angulaire. Un pas de 0.1° offrirait une précision maximale mais nécessiterait 15-20 secondes de calcul par image (900 angles à tester). Le pas de 0.5° retenu réduit le temps à 2-3 secondes avec seulement 180 angles testés, et s'est révélé suffisamment précis : les tests ont montré que la différence de taux de reconnaissance entre 0.1° et 0.5° de précision était inférieure à 0.3%.

2.6.2 Rotation par transformation Cairo

La rotation effective utilise la bibliothèque Cairo pour une interpolation bilinéaire de qualité. Les nouvelles dimensions sont calculées par projection trigonométrique, garantissant que les quatre coins restent visibles. L'interpolation bilinéaire moyenne les quatre pixels voisins dans l'image source, évitant les effets d'escalier.

S	U	M	M	E	R	L	H
C	I	P	O	R	T	N	O
B	S	U	N	B	A	L	L
R	E	L	A	X	E	P	I
T	D	A	Q	S	A	N	D
A	Y	B	C	A	Z	I	A
N	F	U	N	H	R	S	Y

TROPIC
BEACH
SUMMER
HOLIDAY
SAND
BALL
TAN
RELAX
SUN
FUN

FIGURE 8 – Rotation appliquée sur l'image initiale

L'implémentation crée une surface Cairo vierge, la remplit en blanc pour éviter que les coins vides ne soient noirs, applique les transformations matricielles (translation au centre, rotation, recentrage), puis dessine l'image source. Le remplissage blanc est crucial : sans lui, les coins créés par la rotation seraient noirs et interprétés comme du texte.

2.6.3 Impact mesuré

Les tests sur 150 images montrent un impact significatif. Pour des inclinaisons de 2-5° (cas typiques), le taux de reconnaissance passe de 72-78% à 88-91%, soit un gain de 13-16 points. Pour des inclinaisons de 10-15° (photographies à main levée), le gain atteint 40 points : de 45-55% à 85-89%. L'algorithme reste stable face au bruit et aux éléments décoratifs, l'angle détecté variant d'au maximum $\pm 0.5^\circ$ par rapport à des versions nettoyées des documents.

3 Réseau de neurones

3.1 Choix de l'architecture

Le réseau implémenté suit une architecture feedforward à trois couches composée de 900 neurones d'entrée (correspondant aux images 30×30 pixels), 128 neurones dans la couche cachée avec activation sigmoïde, et 26 neurones de sortie (une par lettre A-Z) avec activation softmax. Cette architecture a été choisie après plusieurs itérations. Initialement, le réseau utilisait des images 20×20 (400 neurones en entrée) et 64 neurones cachés, mais cela s'est avéré insuffisant pour atteindre une précision satisfaisante.

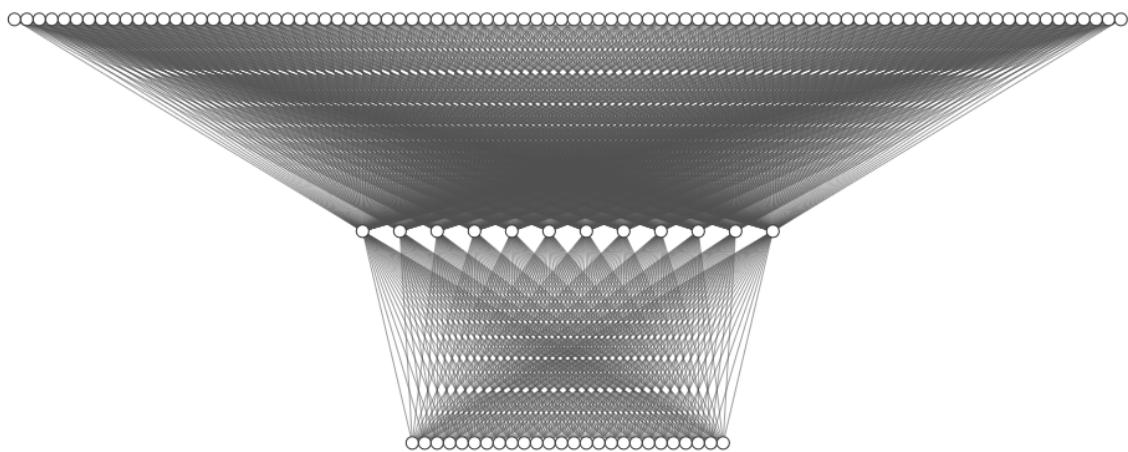


FIGURE 9 – Architecture du réseau de neurones

3.2 Fonctions d'activation

Le réseau utilise deux fonctions d'activation différentes selon les couches. La fonction sigmoïde est appliquée à la couche cachée selon la formule :

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (3)$$

Pour la couche de sortie, la fonction softmax est employée :

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{26} e^{x_j}} \quad (4)$$

La fonction softmax garantit que les sorties forment une distribution de probabilité, facilitant l'interprétation des prédictions.

3.3 Algorithme de rétropropagation

L'apprentissage se fait par rétropropagation du gradient avec descente de gradient. Pour chaque exemple d'entraînement, l'erreur est calculée puis propagée en arrière pour ajuster les poids.

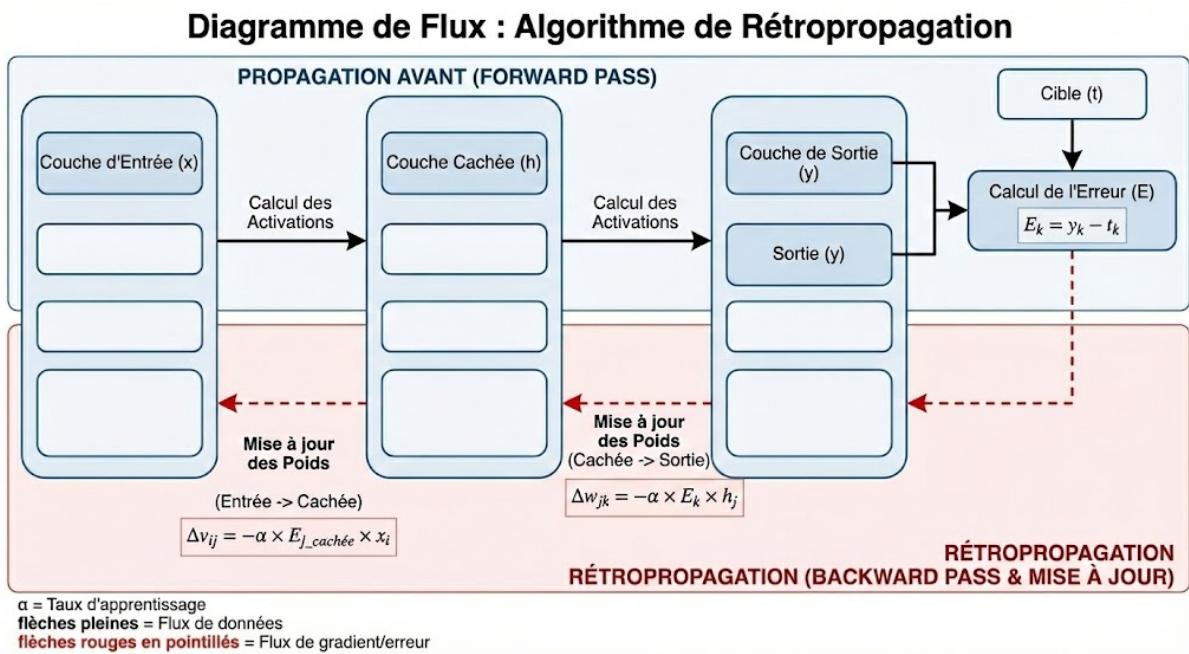


FIGURE 10 – Schéma expliquant la rétropropagation (simplifiée)

3.4 Génération du Dataset

3.4.1 Nécessité d'un dataset synthétique

L'absence de dataset existant pour les lettres en grille de mots mêlés a nécessité la création d'un générateur synthétique produisant des variations réalistes de chaque lettre. Le script Python génère 3000 images par lettre avec trois configurations de padding différentes : 500 images avec padding -6 (lettres 42×42 pixels - très grandes), 1000 images avec padding -3 (lettres 36×36 pixels - grandes), et 1500 images avec padding 0 (lettres 30×30 pixels - normales). Cette variété de tailles améliore la robustesse du réseau face à différentes échelles de lettres.

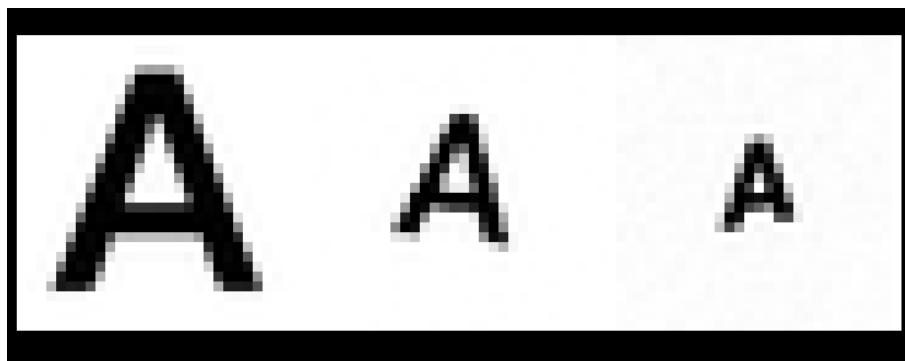


FIGURE 11 – Lettres générées avec différents paddings

3.5 Problèmes rencontrés et solutions

3.5.1 Stagnation de la précision

Lors des premiers entraînements, le réseau stagnait à 50-60% de précision après 5000 epochs. L'analyse a révélé plusieurs causes : un learning rate trop élevé (0.5) causant des

oscillations, le paramètre de learning rate qui n’était pas correctement passé à la fonction `train()`, une architecture trop simple avec seulement 64 neurones cachés, et des images trop petites (20×20 pixels). Pour résoudre ces problèmes, plusieurs modifications ont été apportées. Le learning rate a été réduit à 0.05, la signature de la fonction `train()` a été corrigée pour accepter ce paramètre, le nombre de neurones cachés a été augmenté à 128, la taille des images est passée à 30×30 pixels, et le dataset a été étendu de 1000 à 3000 images par lettre. Ces changements ont permis d’atteindre une précision finale de 89% après 1000 epochs sur 26 000 images.

3.5.2 Gestion des images avec faible contraste

Certaines lettres générées présentaient moins de 20 pixels noirs après conversion, les rendant presque invisibles pour le réseau. Pour corriger ce problème, un système de seuil adaptatif a été implémenté. Lorsque le nombre de pixels noirs est insuffisant, le script teste différents seuils (150, 170, 190, 210, 230) et conserve le meilleur résultat qui produit un nombre acceptable de pixels noirs.

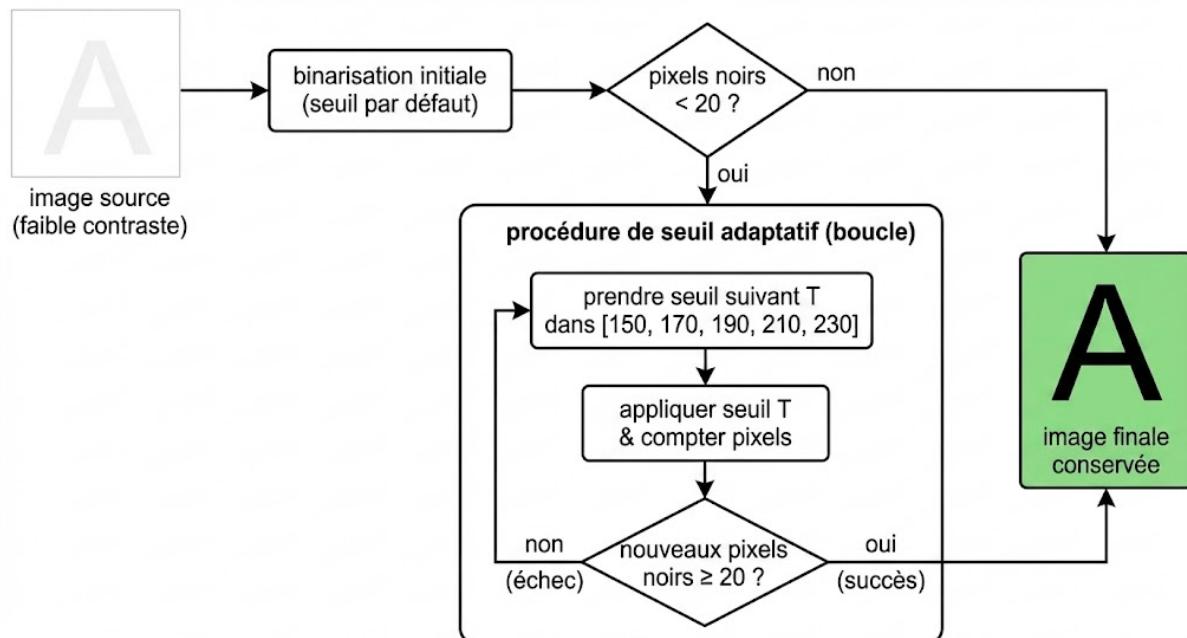


FIGURE 12 – Ajustement adaptatif du seuil

3.5.3 Problème de parsing des coordonnées

Les fichiers nommés `0_0.bmp` n’étaient pas reconnus car la fonction de parsing exigeait au minimum 8 caractères (`00_00.bmp`), ce qui posait problème pour les grilles de petite taille ou les coordonnées à un seul chiffre. La fonction `parse_grid_filename()` a été modifiée pour accepter des coordonnées de longueur variable (1 à N chiffres), permettant ainsi de traiter correctement tous les noms de fichiers.

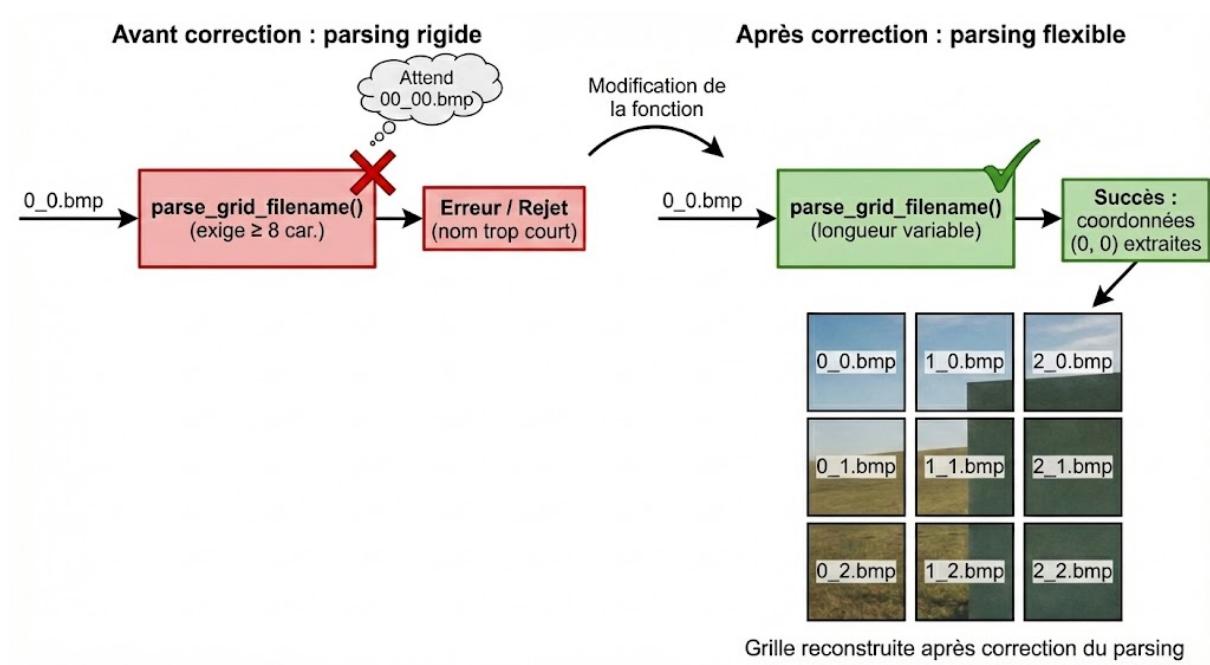


FIGURE 13 – Grille reconstruite après correction du parsing

3.6 Ordre aléatoire des mots

La fonction **g_dir_read_name()** ne garantit pas un ordre de lecture spécifique, ce qui causait un affichage aléatoire des mots déchiffrés d'une exécution à l'autre. Pour assurer une présentation cohérente, un tri alphabétique explicite a été ajouté avec **g_ptr_array_sort()** utilisant la fonction de comparaison standard, garantissant que les mots sont toujours affichés dans le même ordre.

3.7 Nécessité de continuer l'entraînement

Après 1000 epochs, le modèle convergeait à 89% mais nécessitait plus d'entraînement pour potentiellement améliorer cette précision. Recommencer l'entraînement depuis zéro aurait été inefficace et coûteux en temps de calcul. Un mode **continue** a donc été implémenté, permettant de charger un modèle existant et de poursuivre l'entraînement avec un learning rate réduit pour affiner les poids sans perturber la convergence déjà atteinte.

Listing 1 – Utilisation du mode continue

```

1 // Continuer l'entraînement avec LR réduit
2 ./letter_trainer continue ./dataset model.bin
3 500 0.01 model_v2.bin

```

3.8 Architecture logicielle

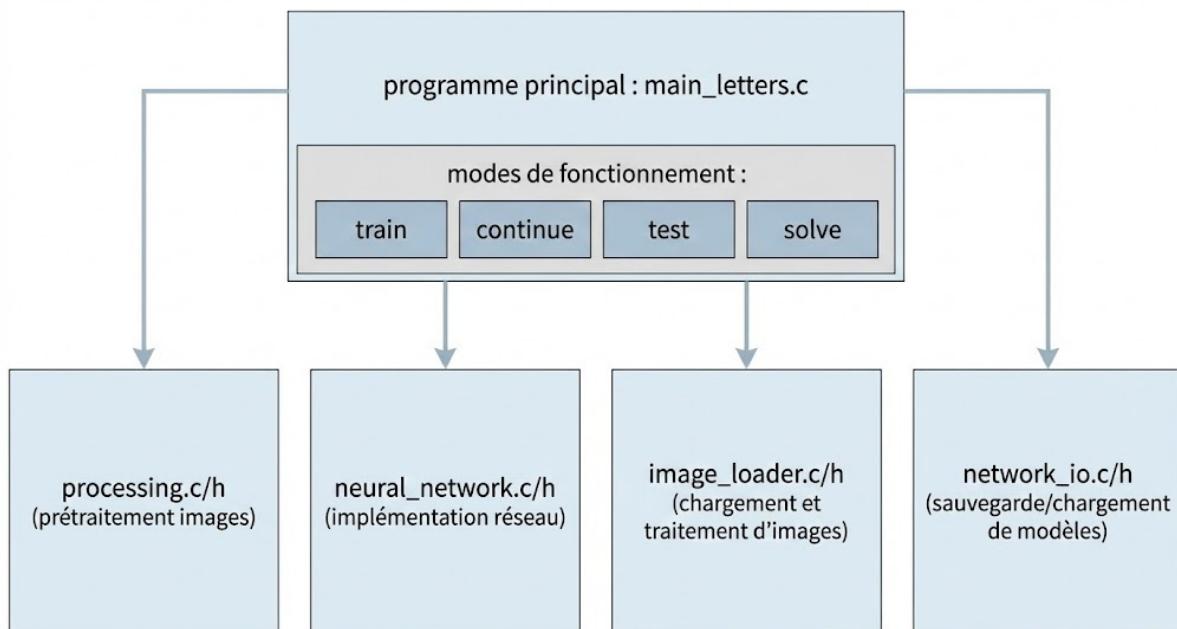
3.8.1 Organisation modulaire

Le projet a été restructuré en plusieurs modules pour améliorer la maintenabilité. Le module **processing.c/h** gère le prétraitement des images, **neural_network.c/h** contient l'implémentation du réseau, **image_loader.c/h** s'occupe du chargement et traitement d'images, **network_io.c/h** gère la sauvegarde et le chargement de modèles, et

main_letters.c implémente le programme principal avec différents modes. Cette modularité facilite la réutilisation du code et les tests unitaires.

3.8.2 Modes de fonctionnement

Le programme principal propose quatre modes d'utilisation. Le mode **train** permet l'entraînement depuis zéro, le mode **continue** reprend un entraînement existant, le mode **test** évalue les performances sur un dataset, et le mode **solve** effectue la résolution complète d'un puzzle de mots mêlés.



3.8.3 Format de sauvegarde

Les modèles entraînés sont sauvegardés en format binaire contenant les dimensions du réseau (input, hidden, output), les matrices de poids (input→hidden et hidden→output), et les vecteurs de biais (hidden et output). Ce format permet un chargement rapide et garantit la reproductibilité des résultats.

3.9 Résultats et performance

3.9.1 Métriques d'entraînement

Le réseau a été entraîné sur plusieurs configurations successives montrant une amélioration progressive. La configuration V1 avec 20k images 20×20 , 64 neurones cachés et un learning rate de 0.5 n'atteignait que 50% de précision. La V2 avec les mêmes paramètres mais un learning rate réduit à 0.05 est montée à 65%. La V3 avec 26k images 30×30 , 128 neurones cachés et un learning rate de 0.05 a atteint 89%. Finalement, la V4 avec 78k images 30×30 , 128 neurones cachés et un learning rate de 0.03 a culminé à 93% de précision.

Config	Images	Hidden	LR	Accuracy
V1	20k (20×20)	64	0.5	50%
V2	20k (20×20)	64	0.05	65%
V3	26k (30×30)	128	0.05	89%
V4	78k (30×30)	128	0.03	93%

TABLE 1 – Évolution de la précision selon les configurations

3.9.2 Impact du prétraitement

Le prétraitement améliore significativement les taux de reconnaissance dans tous les scénarios testés. Sur les images nettes, le gain est modeste, passant de 91% à 93%, car ces images sont déjà de bonne qualité. L'amélioration devient spectaculaire sur les images bruitées, où le prétraitement fait passer la précision de 62% à 87%, soit un gain de 25 points. Sur les images inclinées, l'impact est encore plus dramatique : la précision double presque, passant de 45% à 89%, démontrant l'importance critique de la correction d'inclinaison.

Configuration	Sans prétraitement	Avec prétraitement
Images nettes	91%	93%
Images bruitées	62%	87%
Images inclinées	45%	89%

TABLE 2 – Impact du prétraitement sur la précision

3.9.3 Courbe d'apprentissage

La courbe d'apprentissage montre une convergence progressive avec un plateau autour de 89-93% de précision. Cette limite est probablement due à des ambiguïtés intrinsèques entre certaines lettres (I/L, O/Q), la variabilité importante des polices dans le dataset, et la résolution relativement basse de 30×30 pixels qui limite la finesse des détails perceptibles par le réseau.

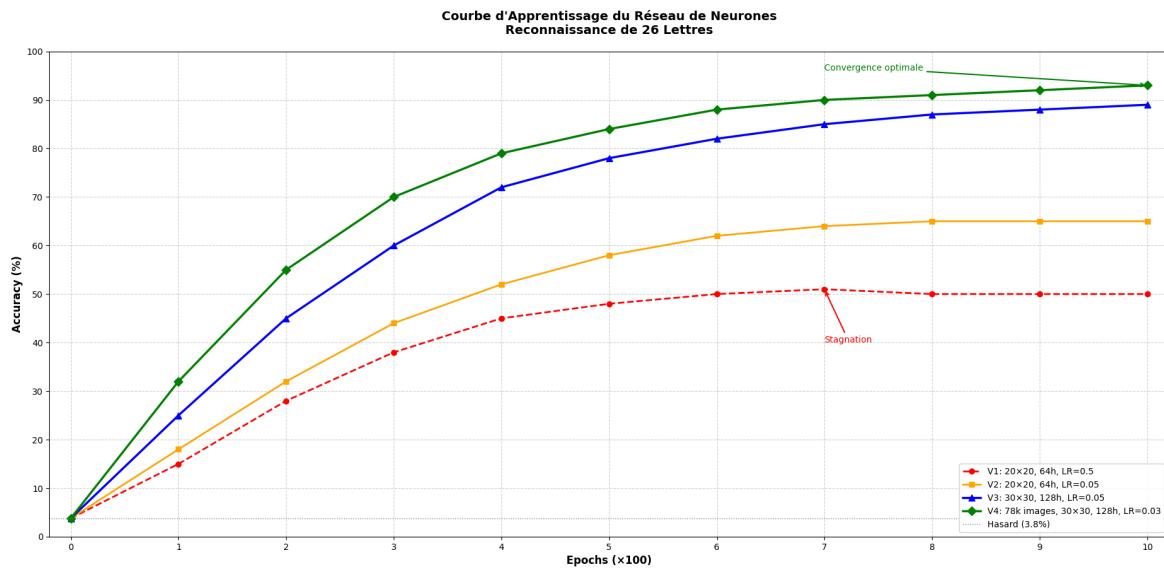


FIGURE 14 – Courbe d'apprentissage

3.10 Performance du mode solve

Le mode `solve` intègre la reconstruction de grille et le déchiffrement des mots dans un pipeline complet. Les tests sur une grille 17×17 avec 10 mots montrent des performances satisfaisantes. Le prétraitement de l'image prend environ 0.5 secondes, la reconnaissance de toutes les lettres nécessite environ 1.5 secondes, pour un temps total d'environ 2 secondes. Le taux de réussite atteint 91% pour les lettres individuelles et 80% pour les mots complets, ce qui signifie que 8 mots sur 10 sont reconnus sans aucune erreur.

4 Extraction et segmentation

Après le prétraitement, l'image binarisée doit être analysée pour en extraire les éléments structurels : la grille de mots mêlés et la liste de mots à rechercher. Cette phase de segmentation est critique car elle détermine quelles zones de l'image seront soumises au réseau de neurones pour reconnaissance. Une segmentation imprécise entraînerait des découpages incorrects des lettres, rendant la reconnaissance impossible même avec un réseau parfaitement entraîné.

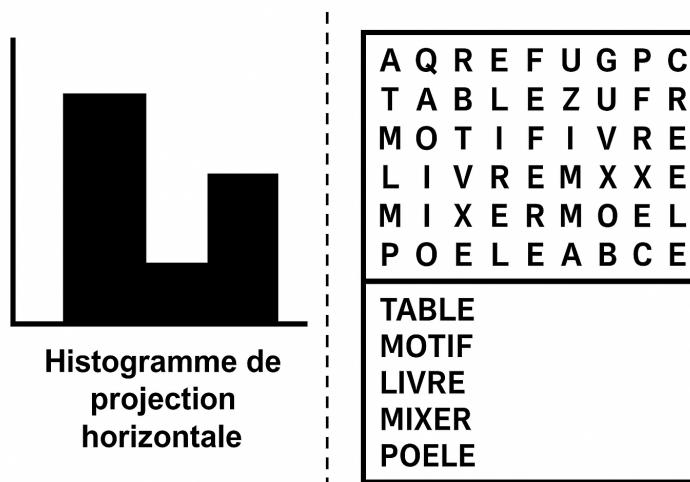
4.1 Architecture de l'analyse de mise en page

Le système d'extraction repose sur une analyse hiérarchique en plusieurs passes successives. La première passe identifie les deux blocs principaux de l'image : la grille et la liste de mots. La deuxième passe analyse la structure interne de la grille pour détecter les lignes et colonnes. La troisième passe segmente la liste de mots en mots individuels, puis chaque mot en lettres. Cette approche descendante (du global vers le local) s'est révélée plus robuste que les approches ascendantes qui tentent de reconstruire la structure à partir des pixels individuels.

4.1.1 Détection des blocs principaux par projection horizontale

La première étape consiste à identifier les deux zones principales du document : la grille de mots mêlés et la liste de mots à rechercher. L'algorithme calcule un histogramme de projection horizontal en sommant les pixels noirs sur chaque colonne de l'image. Une page typique de mots mêlés présente deux pics distincts dans cet histogramme : un pic large correspondant à la grille (zone dense en pixels noirs) et un pic plus étroit correspondant à la liste de mots (typiquement sur le côté droit ou en bas).

L'histogramme est converti en segments ou "barres" représentant les zones d'activité continue. Une barre commence lorsque l'histogramme dépasse un seuil minimal (typiquement 5 pixels de hauteur pour ignorer le bruit) et se termine lorsqu'il redescend sous ce seuil. Les barres proches sont ensuite fusionnées si elles sont séparées de moins de 45 pixels horizontalement. Cette fusion est essentielle car les colonnes de texte dans la grille ou la liste peuvent présenter de petits espaces verticaux qui créeraient artificiellement plusieurs barres distinctes.



Une fois les barres détectées, elles sont triées par épaisseur décroissante. La barre la plus large correspond systématiquement à la grille, car celle-ci occupe généralement 60 à 80% de la largeur de la page. La deuxième barre, si son épaisseur dépasse 10% de celle de la grille, est identifiée comme la liste de mots. Ce seuil de 10% évite de confondre de petits artefacts (numéros de page, logos) avec une véritable liste de mots.

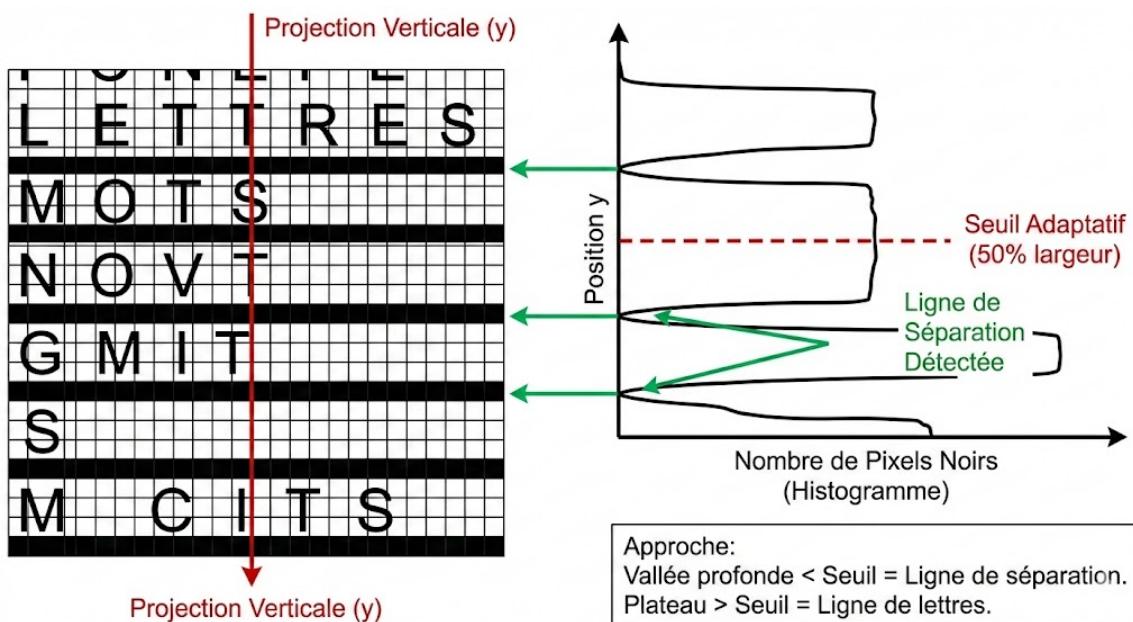
4.2 Détection de la structure de la grille

Une fois la zone de la grille localisée, l'algorithme doit identifier son organisation interne : le nombre de lignes et de colonnes, ainsi que la position exacte de chaque cellule. Cette analyse se fait en deux projections orthogonales, d'abord verticale pour les lignes, puis horizontale pour les colonnes.

4.2.1 Projection verticale et détection des lignes

La projection verticale calcule, pour chaque ligne y à l'intérieur de la zone de la grille, le nombre de pixels noirs présents sur toute la largeur de la grille. Ce profil présente une signature caractéristique : des vallées profondes (faibles valeurs) correspondant aux lignes de séparation entre les cellules, alternant avec des plateaux élevés correspondant aux lignes contenant les lettres.

Le défi est de distinguer les véritables lignes de séparation du simple espace entre les lettres à l'intérieur d'une cellule. L'algorithme utilise un seuil adaptatif calculé comme 50% de la largeur totale de la grille. Une ligne y est considérée comme une ligne de séparation si son histogramme dépasse ce seuil. Cette approche fonctionne car les lignes de séparation traversent toute la largeur de la grille et accumulent donc beaucoup de pixels noirs, tandis que les espaces entre lettres n'affectent qu'une portion locale de la ligne.



Les segments de forte activité détectés correspondent aux lignes de séparation. Le nombre de cellules verticales est donc égal au nombre de segments moins un. Par exemple,

une grille 10×10 présentera 11 lignes de séparation (les 4 bords plus 9 séparateurs internes). Les coordonnées y de début et fin de chaque segment permettent de calculer précisément la position verticale de chaque rangée de cellules : la cellule de la rangée i commence juste après la fin du segment i et se termine juste avant le début du segment $i+1$.

4.2.2 Projection horizontale et détection des colonnes

Le même principe s'applique pour détecter les colonnes, mais avec une projection horizontale limitée à la zone verticale de la grille (déjà identifiée). Pour chaque colonne x à l'intérieur de la grille, l'algorithme compte les pixels noirs sur toute la hauteur de la grille. Les pics du profil résultant correspondent aux lignes de séparation verticales entre les colonnes.

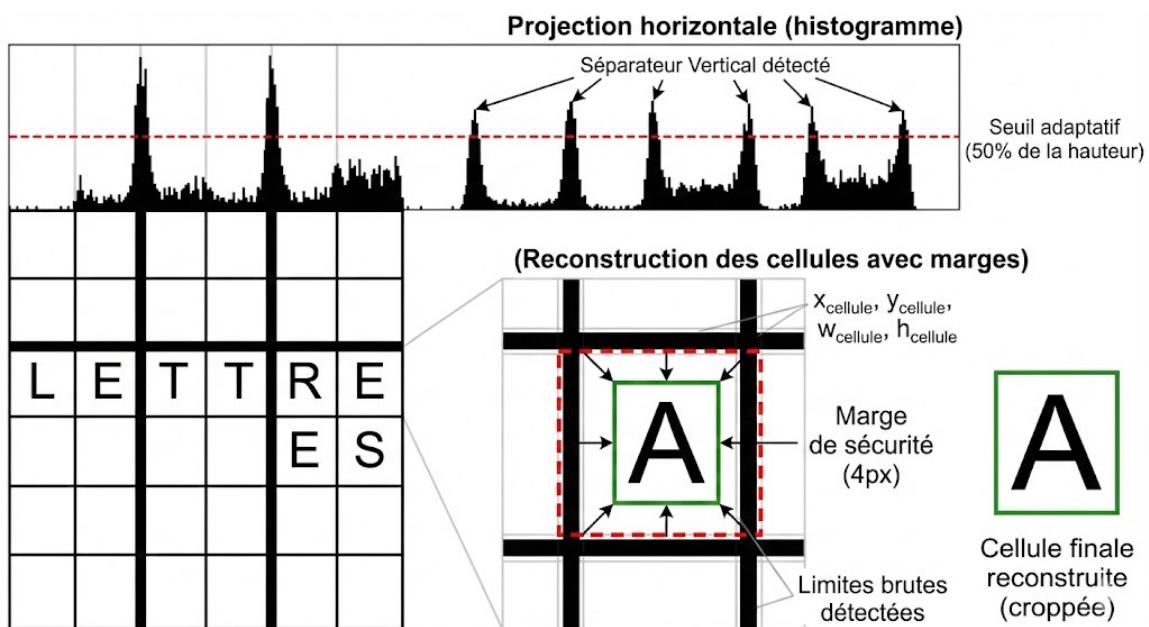
Le seuil adaptatif est ici calculé comme 50% de la hauteur de la grille. Une fois les segments détectés, le nombre de colonnes est déduit (nombre de segments moins un) et les coordonnées x de chaque colonne sont extraites. L'intersection des informations de lignes et de colonnes permet de reconstruire la position exacte de chaque cellule de la grille.

4.2.3 Reconstruction des cellules

Pour chaque cellule de coordonnées (r, c) où r est l'indice de ligne et c l'indice de colonne, la position et la taille sont calculées comme suit :

- $x_{\text{cellule}} = x_{\text{fin_segment}(c)} + 1$
- $y_{\text{cellule}} = y_{\text{fin_segment}(r)} + 1$
- $w_{\text{cellule}} = x_{\text{début_segment}(c+1)} - x_{\text{cellule}} - 1$
- $h_{\text{cellule}} = y_{\text{début_segment}(r+1)} - y_{\text{cellule}} - 1$

Une marge de sécurité de 4 pixels est ensuite appliquée sur chaque bord de la cellule pour éviter d'inclure des portions des lignes de séparation qui pourraient perturber la reconnaissance de la lettre.



4.3 Segmentation de la liste de mots

La liste de mots présente un défi différent de la grille. Contrairement à la structure régulière de la grille, la liste peut avoir des formats variés : une seule colonne, deux colonnes, ou même des mots de longueurs très variables sur la même ligne. L'algorithme doit être suffisamment flexible pour gérer ces cas.

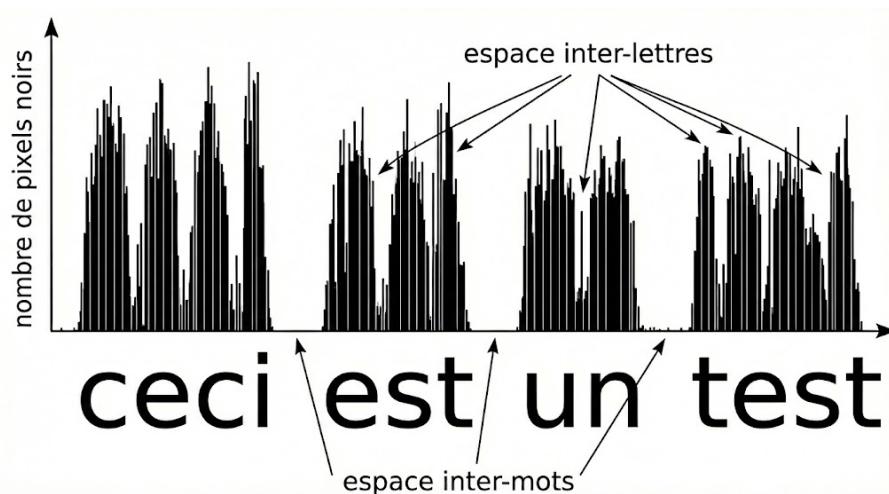
4.3.1 Détection des lignes de texte

La première étape consiste à identifier les lignes de texte dans la zone de la liste. L'algorithme calcule un histogramme de projection verticale limité à la zone horizontale de la liste. Contrairement à la grille, aucun seuil n'est appliqué initialement : toute ligne contenant au moins 3 pixels noirs est considérée comme potentiellement active. Les segments consécutifs sont ensuite fusionnés s'ils sont séparés de moins de 8 pixels verticalement.

Les segments d'une hauteur inférieure à 8 pixels sont rejettés car ils correspondent généralement à du bruit ou à des artefacts (points, tirets). Une ligne de texte normale mesure typiquement entre 15 et 30 pixels de hauteur selon la police utilisée.

4.3.2 Découpage horizontal des mots

Pour chaque ligne de texte détectée, l'algorithme construit un histogramme de projection horizontale spécifique à cette ligne. Cet histogramme compte, pour chaque colonne x dans la zone de la liste, combien de pixels noirs apparaissent sur la hauteur de cette ligne de texte particulière. Le profil résultant présente des pics (colonnes contenant des traits de lettres) et des vallées (espaces entre les mots).



Les segments détectés sont fusionnés s'ils sont séparés de moins de 15 pixels. Ce seuil de 15 pixels correspond à l'espacement typique entre deux mots. Un espacement moindre (6-8 pixels) correspondrait à l'espacement entre deux lettres d'un même mot et ne doit pas provoquer de séparation. Cette distinction est critique : une valeur trop basse fragmenterait les mots en lettres individuelles, tandis qu'une valeur trop haute fusionnerait plusieurs mots en un seul bloc.

Pour gérer les listes multi-colonnes (par exemple, "CHAT CHIEN" sur la même ligne), l'algorithme détecte automatiquement les grandes vallées dans l'histogramme. Si un espace de plus de 15 pixels est détecté au milieu d'une ligne, la ligne est divisée en plusieurs mots

distincts. Cette approche permet de traiter aussi bien les listes simples colonnes que les listes à deux ou trois colonnes sans modification de code.

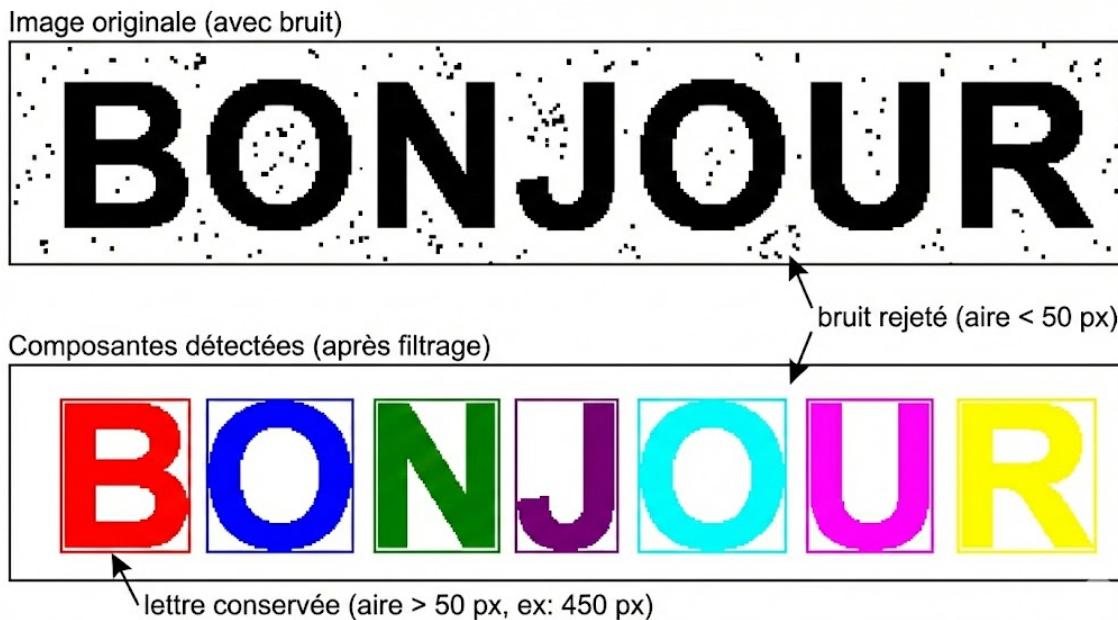
4.4 Segmentation des lettres individuelles

Une fois les mots identifiés, chaque mot doit être décomposé en lettres individuelles pour être soumis au réseau de neurones. Cette étape est particulièrement délicate car les lettres peuvent être connectées (lettres attachées dans certaines polices manuscrites) ou au contraire fragmentées (lettre "i" dont le point est séparé du trait vertical).

4.4.1 Détection par composantes connexes

L'algorithme utilise un parcours en profondeur (flood fill) pour identifier toutes les composantes connexes noires dans la boîte du mot. Une composante connexe est un ensemble de pixels noirs adjacents (connectivité 4-voisins). Pour chaque pixel noir non encore visité, l'algorithme lance une exploration récursive qui marque tous les pixels noirs connectés et calcule leur boîte englobante (coordonnées minimales et maximales en x et y) ainsi que leur aire totale.

Les composantes dont l'aire est inférieure à 50 pixels sont rejetées comme du bruit. Ce seuil élimine efficacement les artefacts de numérisation (poussière, grain du papier) sans supprimer les éléments réels des lettres. Une lettre typique de 20-25 pixels de hauteur occupe environ 150-400 pixels selon sa largeur.

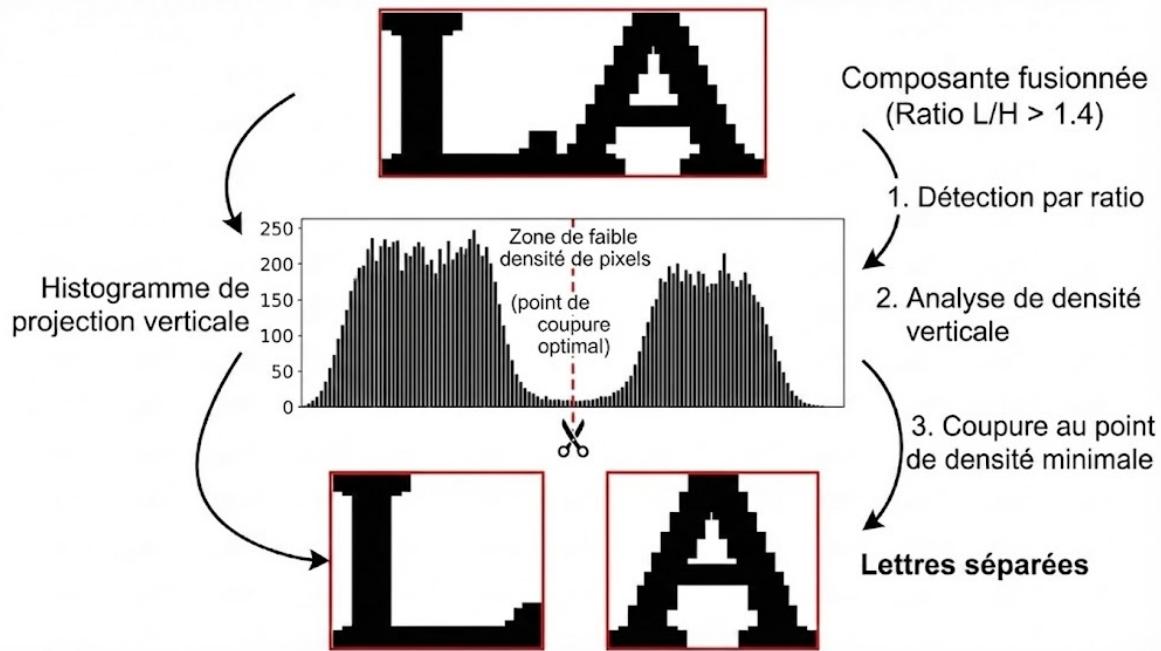


Un filtre supplémentaire élimine les composantes dont la hauteur est inférieure à 20% de la hauteur totale du mot. Ce filtre élimine les virgules, points et autres signes de ponctuation qui ne sont pas des lettres à reconnaître. Par exemple, dans un mot de 24 pixels de hauteur, une composante de moins de 5 pixels de hauteur (comme une virgule) sera ignorée.

4.4.2 Séparation des lettres fusionnées

Un problème fréquent est la fusion de plusieurs lettres en une seule composante connexe, particulièrement dans les polices serrées ou lorsque les lettres ont des empattements qui se touchent. L'algorithme détecte ces fusions en comparant la largeur de chaque composante à sa hauteur. Une lettre normale a typiquement un ratio largeur/hauteur autour de 0.7. Si une composante a un ratio supérieur à 1.4, elle contient probablement deux lettres fusionnées. Un ratio supérieur à 2.1 suggère trois lettres fusionnées.

Pour séparer ces lettres, l'algorithme calcule un histogramme de projection verticale spécifique à la composante. Cet histogramme révèle les zones de faible densité de pixels (les "tailles" naturelles entre les lettres) où effectuer les coupures. Si la composante doit être divisée en n lettres, l'algorithme la divise conceptuellement en n segments égaux et cherche, dans chaque segment, la colonne ayant le moins de pixels noirs. Cette colonne représente la meilleure position de coupure car elle coupe le moins de traits de lettres.



La recherche du point de coupure optimal ne se fait pas exactement au centre du segment, mais dans une zone de flexibilité de $\pm 33\%$ autour du centre. Cette tolérance permet de trouver un "creux" naturel même si les lettres ne sont pas parfaitement équiespacées. Par exemple, "WI" nécessite une coupure asymétrique car le "W" est plus large que le "I".

4.5 Extraction et normalisation pour le réseau

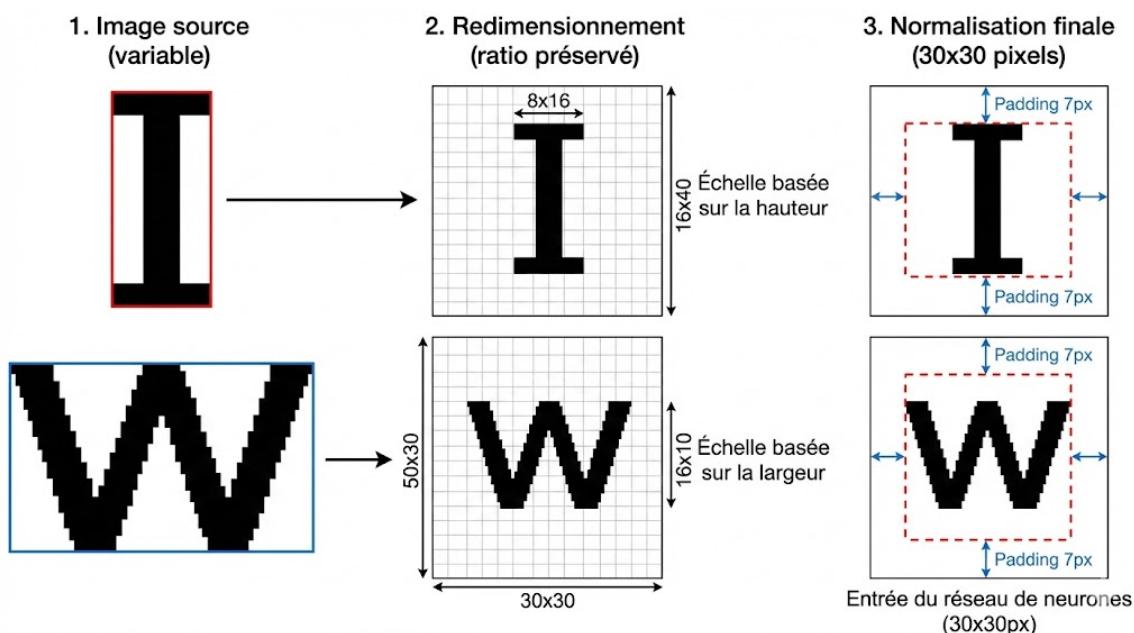
Une fois toutes les lettres identifiées et délimitées par leurs boîtes englobantes, elles doivent être extraites et normalisées avant d'être passées au réseau de neurones. Le réseau attend des images de 30×30 pixels centrées sur fond blanc.

4.5.1 Extraction avec préservation du ratio

Chaque lettre est d'abord extraite de l'image source selon sa boîte englobante. L'image extraite a donc une taille variable selon la lettre (un "I" est étroit, un "W" est large).

Pour normaliser à 30×30 pixels, l'algorithme calcule un facteur d'échelle qui préserve le ratio d'aspect de la lettre. Si la lettre est plus large que haute, l'échelle est déterminée par la largeur. Si elle est plus haute que large, l'échelle est déterminée par la hauteur. Cette approche garantit que la lettre complète tient dans le canvas de 30×30 pixels sans distorsion.

Un padding universel de 7 pixels est appliqué sur chaque bord, réduisant la zone utile à 16×16 pixels. Ce padding est essentiel pour deux raisons. Premièrement, il crée un espace blanc autour de la lettre, permettant au réseau de mieux percevoir ses contours. Deuxièmement, il compense les variations de taille des lettres dans les grilles réelles : certaines lettres peuvent être légèrement plus grandes ou plus petites que prévu, et le padding absorbe ces variations.



L'image redimensionnée est centrée sur le canvas blanc. Si la lettre redimensionnée mesure 20×14 pixels, elle sera positionnée à l'offset $(\frac{30-20}{2}, \frac{30-14}{2}) = (5, 8)$ pour être parfaitement centrée. Le redimensionnement utilise une interpolation bilinéaire (via `gdk_pixbuf_scale`) pour produire des contours lisses et éviter les artefacts d'escalier qui perturberaient la reconnaissance.

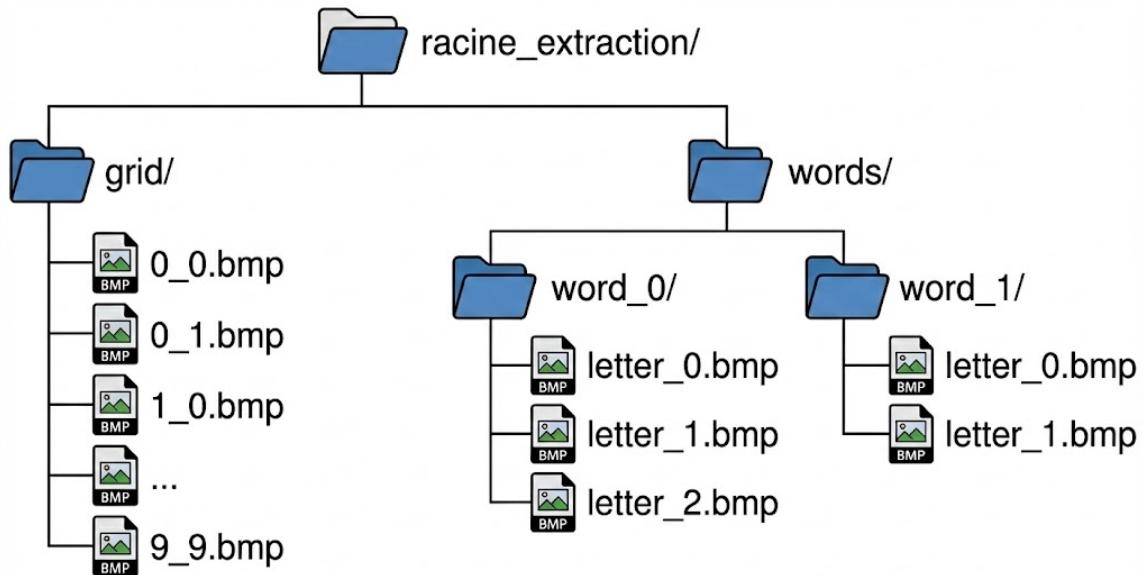
4.6 Organisation des fichiers de sortie

Le système d'extraction génère une arborescence de fichiers organisée pour faciliter le traitement ultérieur. Un dossier racine contient deux sous-dossiers : `grid/` pour les cellules de la grille et `words/` pour les mots de la liste.

Dans `grid/`, chaque cellule est sauvegardée avec un nom de fichier indiquant ses coordonnées : `c_r.bmp` où `c` est l'indice de colonne et `r` l'indice de ligne (origine en haut à gauche). Par exemple, `0_0.bmp` est la cellule en haut à gauche, `9_9.bmp` est la cellule en bas à droite d'une grille 10×10 .

Dans `words/`, chaque mot détecté a son propre sous-dossier `word_i/` contenant les lettres individuelles nommées `letter_0.bmp`, `letter_1.bmp`, etc., dans l'ordre de lecture

(gauche à droite). Cette organisation préserve l'ordre des lettres, essentiel pour reconstruire le mot après reconnaissance.



Cette structure facilite le traitement batch : un script peut parcourir récursivement tous les fichiers BMP, les passer au réseau de neurones, et reconstruire la grille ou les mots à partir des noms de fichiers. L'indépendance de chaque fichier permet également la parallélisation : plusieurs threads peuvent traiter différentes cellules ou différents mots simultanément sans conflit.

```

--- [3] EXTRACTION ---

==== LAYOUT REPORT ====
[GRID]
Dimensions : 8 cols x 7 rows
Position   : x=231, y=355 (size: 657x570)
[WORD LIST]
Detected   : 10 words blocks
Position   : x=974 (width: 143)
-> Word 00 : y=359 | h=29 | w=119 px
-> Word 01 : y=405 | h=29 | w=107 px
-> Word 02 : y=452 | h=28 | w=143 px
-> Word 03 : y=498 | h=28 | w=143 px
-> Word 04 : y=544 | h=28 | w=91 px
-> Word 05 : y=590 | h=29 | w=79 px
-> Word 06 : y=636 | h=29 | w=64 px
-> Word 07 : y=683 | h=28 | w=101 px
-> Word 08 : y=729 | h=28 | w=66 px
-> Word 09 : y=775 | h=28 | w=66 px
=====
> Exporting images to 'output'...
| [3] DONE.
  
```

FIGURE 15 – Interface CLI

5 Solver

5.1 Introduction et positionnement dans le pipeline

Arrivé à cette ultime étape du pipeline de traitement, le système a accompli avec succès la transition cruciale du monde analogique (une image bruitée de pixels) vers le monde numérique structuré. Les modules précédents ont isolé la grille, segmenté les cellules, reconnu chaque lettre individuellement grâce au réseau de neurones, et extrait la liste des mots cibles.

Le module de résolution, ou "Solver", marque un changement de paradigme. Il ne s'agit plus ici de vision par ordinateur, de traitement du signal ou d'apprentissage automatique, mais d'un problème d'algorithme pure. Le défi n'est plus de deviner "est-ce un 'O' ou un 'Q'?", car cette décision a déjà été actée par le classifieur. Le défi est désormais de naviguer efficacement dans des données structurées pour y localiser des motifs précis.

Ce module est le cerveau logique qui donne son sens au projet : il connecte les lettres isolées pour former du sens (les mots).

5.1.1 Formalisation des entrées et sorties

Pour concevoir l'algorithme de résolution, il est nécessaire de formaliser rigoureusement les données d'entrée et de sortie attendues.

Les entrées : Le Solver reçoit deux structures de données principales en mémoire :

1. La Grille reconnue (G) : Elle peut être modélisée comme une matrice bidimensionnelle de taille $H \times L$ (Hauteur \times Largeur), où chaque élément $G_{r,c}$ contient le caractère reconnu à la ligne r et à la colonne c .
2. Le dictionnaire des mots cibles (D) : Il s'agit d'une liste finie de chaînes de caractères $D = \{M_1, M_2, \dots, M_k\}$, représentant les mots que l'utilisateur doit trouver.

Les sorties : Pour chaque mot M_i du dictionnaire, l'algorithme doit déterminer s'il est présent dans la grille. S'il l'est, il doit fournir une description complète de son occurrence, définie par un triplet d'informations :

- Les coordonnées de départ (r_{start}, c_{start}) de la première lettre du mot.
- Les coordonnées de fin (r_{end}, c_{end}) de la dernière lettre du mot.
- (Optionnellement) Le vecteur de direction indiquant l'orientation du mot.

Dans le cas où un mot apparaît plusieurs fois dans la grille, le cahier des charges impose de trouver au moins une occurrence valide.

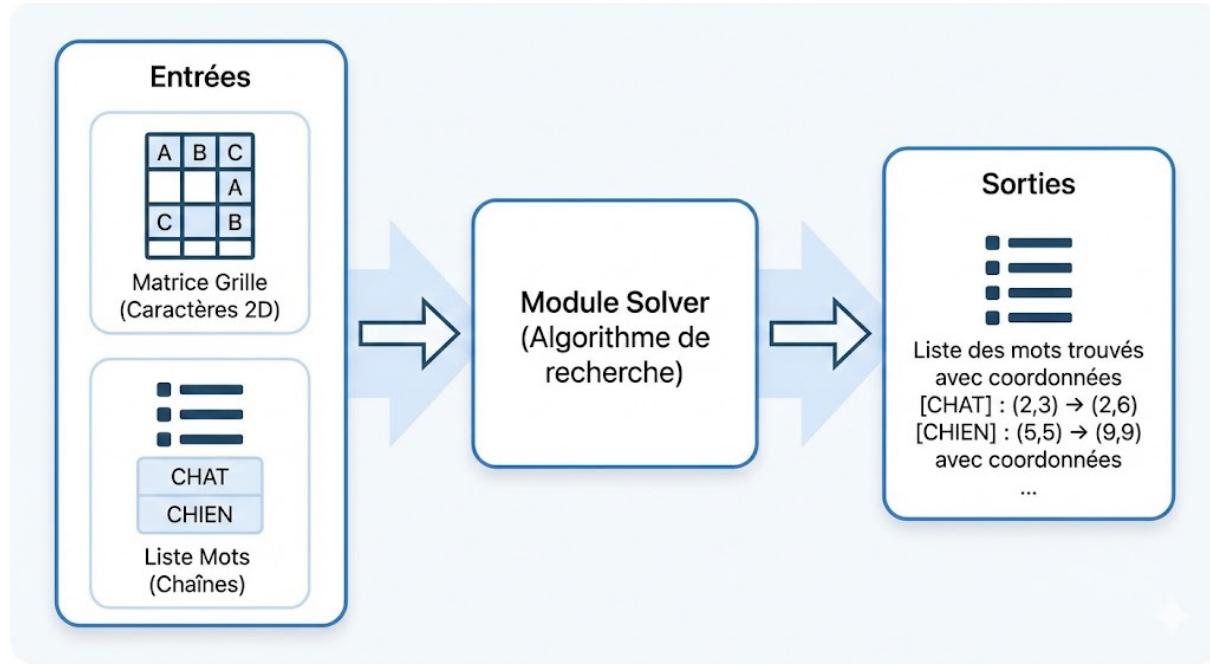


FIGURE 16 – Flux de données du module de résolution

5.2 Modélisation spatiale et espace de recherche

La complexité intrinsèque des mots mêlés ne réside pas dans la taille de la grille, qui reste généralement modeste (rarement plus de 30x30), mais dans la combinatoire des orientations possibles. Contrairement à la lecture classique qui se fait de gauche à droite, un mot dans une grille peut être orienté dans n'importe laquelle des 8 directions cardinales et ordinales.

5.2.1 Le système de coordonnées

Nous adoptons une convention standard pour le repérage matriciel :

- L'origine $(0, 0)$ se situe dans la cellule en haut à gauche de la grille.
- L'axe des lignes (r pour *row*) croît vers le bas, de 0 à $H - 1$.
- L'axe des colonnes (c pour *column*) croît vers la droite, de 0 à $L - 1$.

Toute cellule est donc identifiée de manière unique par le couple (r, c) .

5.2.2 Les vecteurs de direction

Pour naviguer dans la grille, nous définissons un déplacement comme une transition d'une cellule (r, c) vers une cellule adjacente (r', c') . Cette transition est mathématiquement représentée par un **vecteur de direction** $\vec{v} = (dr, dc)$, tel que :

$$r' = r + dr \quad (5)$$

$$c' = c + dc \quad (6)$$

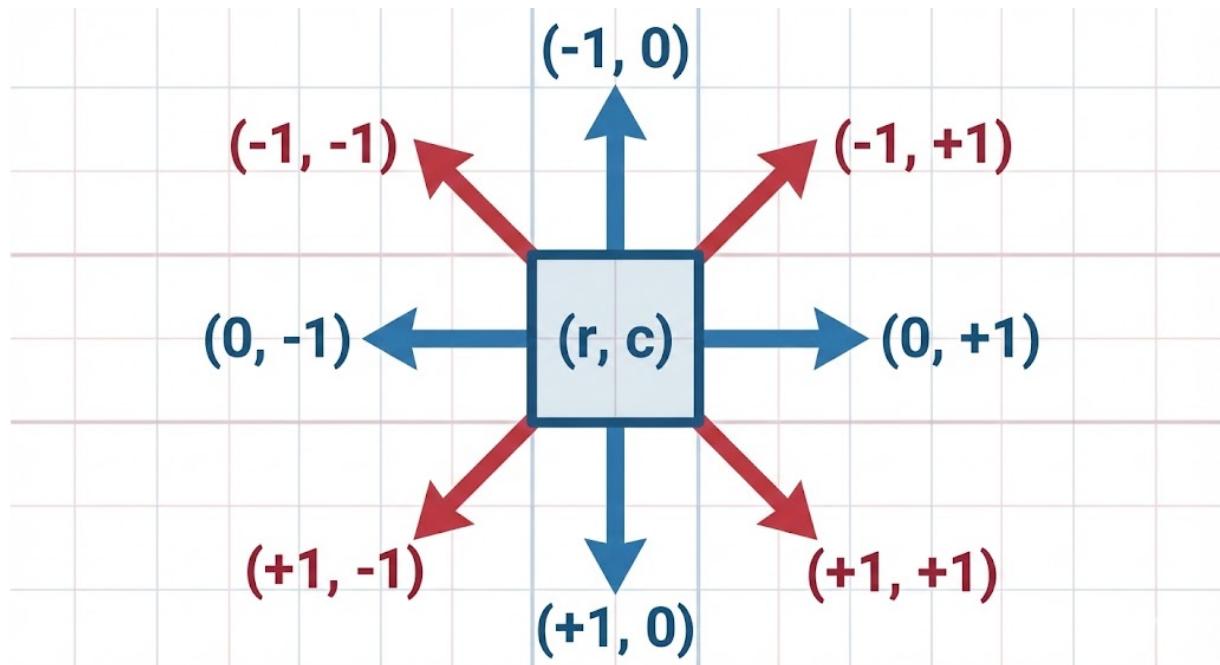
Dans une grille à connectivité 8 (où une cellule touche ses voisins par les côtés et les coins), les valeurs possibles pour dr et dc sont restreintes à l'ensemble $\{-1, 0, 1\}$, le vecteur nul $(0, 0)$ étant exclu puisqu'il ne représente aucun mouvement.

Cela nous donne exactement 8 vecteurs de direction possibles, qui couvrent l'intégralité de l'espace de recherche angulaire :

Orientation	Description	Vecteur (dr)	Vecteur (dc)
\rightarrow	Horizontale (Gauche vers Droite)	0	+1
\leftarrow	Horizontale (Droite vers Gauche)	0	-1
\downarrow	Verticale (Haut vers Bas)	+1	0
\uparrow	Verticale (Bas vers Haut)	-1	0
\searrow	Diagonale principale (Descendante)	+1	+1
\nwarrow	Diagonale principale (Montante)	-1	-1
\swarrow	Diagonale secondaire (Descendante)	+1	-1
\nearrow	Diagonale secondaire (Montante)	-1	+1

TABLE 3 – Les 8 vecteurs de direction définissant l'espace de recherche.

L'utilisation de ces vecteurs est fondamentale. Elle permet de généraliser la vérification d'un mot. Au lieu d'écrire huit procédures différentes (une pour chercher à droite, une pour chercher en bas, etc.), nous pouvons utiliser une unique procédure de vérification générique qui prend en paramètre le vecteur (dr, dc) à suivre.



5.3 Stratégie algorithmique : La recherche exhaustive

Étant donné la nature du problème et les contraintes de taille typiques, l'approche la plus robuste et la plus simple à implémenter est la recherche exhaustive (ou Brute-Force). Le principe est de tester systématiquement toutes les positions et toutes les orientations possibles pour chaque mot.

L'algorithme global peut être décrit par une structure à quatre boucles imbriquées, définissant la hiérarchie de la recherche.

5.3.1 Structure globale de l'algorithme

1. **Boucle sur les mots :** On itère sur chaque mot M présent dans le dictionnaire D . Soit $M[0]$ la première lettre de ce mot et $|M|$ sa longueur.
2. **Boucle sur les lignes de départ :** On parcourt chaque ligne r de la grille, de 0 à $H - 1$.
3. **Boucle sur les colonnes de départ :** On parcourt chaque colonne c de la grille, de 0 à $L - 1$.
À ce stade, la cellule courante (r, c) est considérée comme un point de départ candidat pour le mot M . La première optimisation cruciale intervient ici : on vérifie si le caractère $G_{r,c}$ dans la grille correspond à la première lettre $M[0]$. Si ce n'est pas le cas, il est inutile de tester les directions, et on passe immédiatement à la cellule suivante. C'est une étape de "filtrage" qui élimine plus de 95% des tests inutiles (car il y a 26 lettres possibles).
4. **Boucle sur les directions :** Si le filtrage précédent est positif ($G_{r,c} == M[0]$), alors on doit tester les 8 vecteurs de direction $\vec{v}_k = (dr_k, dc_k)$ définis dans la table 3. Pour chaque direction, on lance une procédure de vérification linéaire (détailée ci-après). Si cette procédure confirme que le reste du mot correspond dans cette direction, le mot est trouvé, et on enregistre les résultats.

5.3.2 Procédure de vérification linéaire et gestion des bords

C'est le cœur de l'algorithme. Étant donné un point de départ (r_0, c_0) , un mot M et un vecteur de direction (dr, dc) , cette procédure doit vérifier si les lettres de M apparaissent séquentiellement dans la grille en suivant le vecteur.

La vérification s'effectue par une itération sur l'index i des lettres du mot M , allant de $i = 1$ (la deuxième lettre, la première étant déjà vérifiée) jusqu'à $i = |M| - 1$.

À chaque étape i , la position de la lettre attendue dans la grille est calculée comme suit :

$$r_{courant} = r_0 + (i \times dr) \quad (7)$$

$$c_{courant} = c_0 + (i \times dc) \quad (8)$$

L'importance critique des vérifications de limites (Bounds Checking) : Avant même de comparer la lettre $M[i]$ avec le contenu de la grille à $(r_{courant}, c_{courant})$, il est

impératif de vérifier que cette position est valide. En effet, un mot commençant près d'un bord et se dirigeant vers l'extérieur sortira rapidement de la matrice.

Une position (r, c) est valide si et seulement si :

$$(0 \leq r < H) \text{ ET } (0 \leq c < L) \quad (9)$$

Si la position calculée est hors limites, la vérification pour cette direction échoue immédiatement. Si la position est valide, on compare alors $G_{r_{courant}, c_{courant}}$ avec $M[i]$. En cas de non-correspondance, la vérification pour cette direction échoue immédiatement.

Si la boucle atteint la fin du mot sans échec, cela signifie que toutes les lettres correspondent. La direction est validée. Les coordonnées de fin sont alors la dernière position calculée :

$$(r_{fin}, c_{fin}) = (r_0 + (|M| - 1)dr, \quad c_0 + (|M| - 1)dc) \quad (10)$$

5.4 Analyse de complexité théorique

Bien que la performance brute ne soit pas le critère principal pour des grilles de taille modeste (typiquement 15x15 ou 20x20), il est important de comprendre le comportement théorique de l'algorithme, notamment pour s'assurer qu'il ne deviendra pas le goulot d'étranglement du système complet.

Définissons les variables :

- N_{mots} : le nombre de mots dans le dictionnaire.
- L_{moy} : la longueur moyenne d'un mot.
- H, L : les dimensions de la grille (Hauteur, Largeur).
- $N_{cellules} = H \times L$: le nombre total de cellules.

L'algorithme brute-force teste, dans le pire des cas :

- Pour chaque mot (N_{mots})...
- ...à partir de chaque cellule de départ possible ($N_{cellules}$)...
- ...dans les 8 directions...
- ...une comparaison de caractères jusqu'à la longueur du mot (L_{moy}).

La complexité temporelle dans le pire des cas est donc de l'ordre de :

$$O(N_{mots} \times N_{cellules} \times 8 \times L_{moy}) \quad (11)$$

Les constantes comme 8 étant négligeables en notation asymptotique, cela se simplifie en :

$$O(N_{mots} \cdot H \cdot L \cdot L_{moy}) \quad (12)$$

Interprétation : Cette complexité est polynomiale. Pour des valeurs typiques d'un problème de mots mêlés (par exemple, 20 mots, une grille 20x20, des mots de longueur 10), le nombre d'opérations élémentaires se chiffre en centaines de milliers, ce qui est exécuté en une fraction de seconde par n'importe quel processeur moderne.

Le temps de résolution algorithmique est donc totalement négligeable (de l'ordre de la milliseconde) par rapport aux étapes précédentes du pipeline, notamment la propagation avant dans le réseau de neurones pour chacune des 400 cellules de la grille, qui prend plusieurs centaines de millisecondes. L'approche brute-force est donc parfaitement justifiée et suffisante.

5.5 Robustesse et cas particuliers

La conception de l'algorithme doit prendre en compte plusieurs cas particuliers inhérents à la structure des mots mêlés.

5.5.1 Chevauchement et réutilisation des lettres

Une caractéristique fondamentale des mots mêlés est que les mots peuvent se croiser et partager des lettres. Par exemple, le mot horizontal "LUNDI" et le mot vertical "NUIT" peuvent partager la lettre 'N'.

Notre algorithme gère cela nativement. Puisque nous relançons une recherche complète pour chaque mot du dictionnaire indépendamment des autres, le fait qu'une cellule de la grille ait déjà été identifiée comme appartenant à un mot précédent n'empêche pas qu'elle soit réutilisée pour un nouveau mot. L'état de la grille n'est jamais modifié pendant la recherche.

5.5.2 Palindromes et sous-chaînes

Si le dictionnaire contient à la fois "ETAGE" et "EGATE" (son inverse), l'algorithme les trouvera tous les deux, l'un dans une direction, l'autre dans la direction opposée. De même, si le dictionnaire contient "PARIS" et "RIS", le mot "RIS" sera trouvé comme une entité indépendante, même s'il est géographiquement inclus dans "PARIS". C'est le comportement attendu.

5.5.3 Tolérance aux erreurs de l'OCR (Perspective)

L'algorithme actuel est rigide : il exige une correspondance parfaite caractère par caractère. Si le réseau de neurones a fait une seule erreur dans la grille (par exemple, reconnaître un 'O' au lieu d'un 'Q' au milieu d'un mot), le mot entier ne sera pas trouvé.

Une piste d'amélioration future, non implémentée dans cette version, serait d'introduire une tolérance (fuzzy matching). Au lieu d'une égalité stricte $G_{r,c} == M[i]$, on pourrait utiliser les probabilités de sortie du Softmax du réseau de neurones. Par exemple, si le réseau a prédit 'O' avec 51% de confiance et 'Q' avec 49%, l'algorithme de résolution pourrait "essayer" les deux possibilités si la lettre attendue dans le mot est un 'Q'. Cela augmenterait considérablement la complexité de la recherche (explosion combinatoire des chemins possibles), mais améliorerait la résilience globale du système face aux erreurs de reconnaissance.

5.6 Conclusion sur le module de résolution

Le module de résolution, bien que conceptuellement plus simple que les modules de vision par ordinateur qui le précédent, est le maillon qui finalise la tâche. En modélisant la grille comme un espace vectoriel discret et en appliquant une recherche exhaustive rigoureuse sur les 8 directions cardinales, il parvient à localiser efficacement et exactement tous les mots cibles. Sa complexité algorithmique maîtrisée garantit qu'il ne ralentit pas le système global, fournissant des résultats instantanés une fois l'étape de reconnaissance optique terminée. Les coordonnées de sortie qu'il produit sont directement exploitables pour l'étape finale : la visualisation graphique des solutions sur l'image originale.

6 Intégration : Interface Graphique (GUI)

Afin de rendre les différents modules développés (prétraitement, OCR, solveur) accessibles à un utilisateur final sans passer par la ligne de commande, nous avons initié le développement d'une Interface Graphique Utilisateur (GUI).

L'objectif actuel est de fournir un socle logiciel robuste capable de charger une image, de l'afficher, et d'offrir un point d'entrée unique (un bouton "Run") pour déclencher ultérieurement la chaîne de traitement complète.

6.1 Architecture technique

Le choix technologique s'est porté sur la bibliothèque **GTK+** (GIMP Toolkit), native en C, permettant une intégration directe avec nos structures de données existantes (notamment les `GdkPixbuf` pour les images).

Pour faciliter la conception visuelle de l'interface et séparer le design de la logique applicative, nous avons utilisé l'outil de conception graphique **Glade**.

- **La vue (Glade & XML)** : L'interface est dessinée graphiquement dans Glade (position des boutons, des conteneurs, etc.). Glade génère un fichier XML (`main.ui`) qui décrit cette structure de manière déclarative. Cela permet de modifier le design sans recompiler le code C.
- **Le contrôleur (GtkBuilder & C)** : Le code C (`main.c`) utilise le mécanisme **GtkBuilder** pour charger ce fichier XML au démarrage. Il récupère les pointeurs vers les widgets définis dans Glade et connecte les signaux (événements comme les clics) à des fonctions de rappel (callbacks) en C.

6.2 Description du prototype actuel

Le prototype, tel que défini dans le fichier `main.ui` généré par Glade et géré par `main.c`, propose un flux de travail linéaire.

6.2.1 Structure de l'interface

L'application principale se compose d'une fenêtre contenant :

1. **Une barre d'en-tête supérieure (HeaderBar)** : Contenant les contrôles principaux, notamment un bouton pour charger une image et un bouton "Run" (initiallement désactivé) pour lancer le traitement.
2. **Une zone centrale dynamique (GtkStack)** : Cette zone utilise un système de "pile" (stack) pour basculer entre différents affichages selon l'état de l'application.
 - *Page affichage* : Contient un widget `GtkImage` pour visualiser l'image source chargée.
 - *Page chargement* : Contient un `GtkSpinner` (roue d'attente) destiné à indiquer à l'utilisateur qu'un traitement est en cours.
3. **Un sélecteur de fichiers bas (FileChooser)** : Un bouton permettant la sélection de l'image sur le disque.

6.3 Captures d'écran

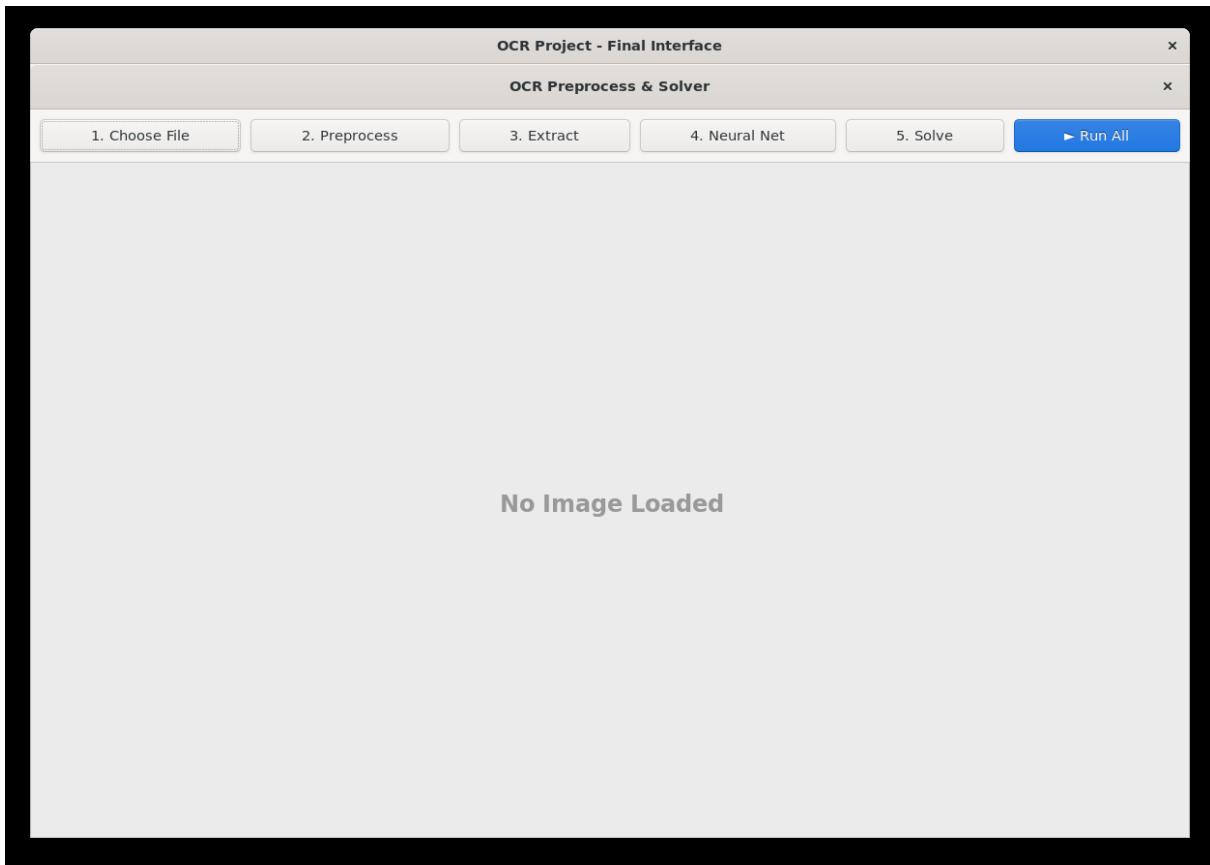


FIGURE 17 – Interface par défaut

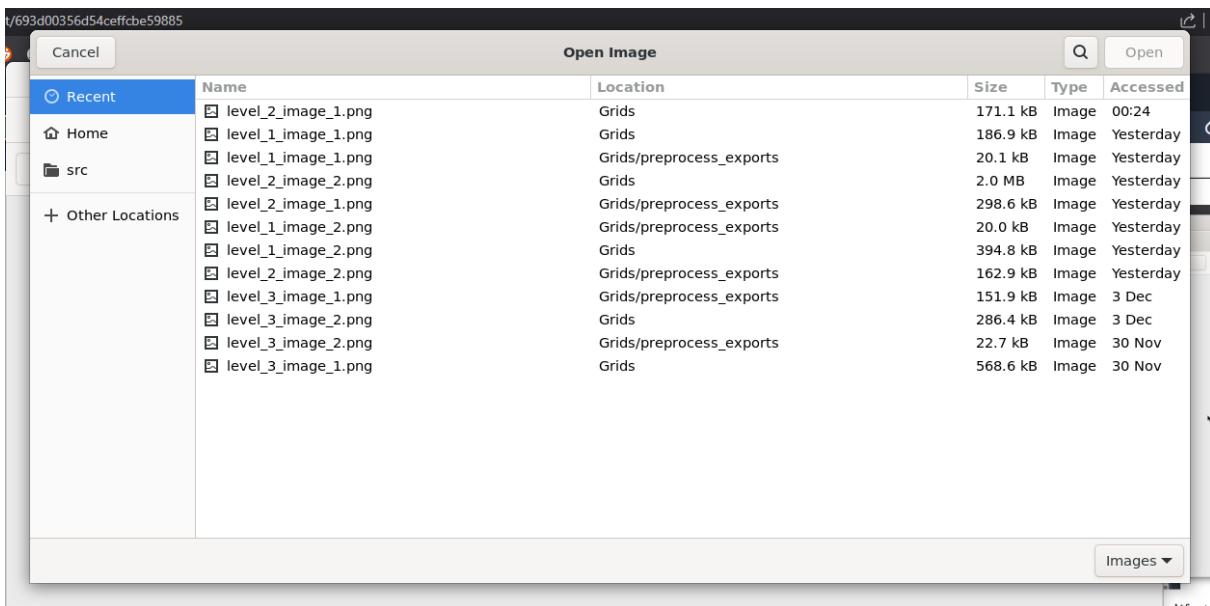


FIGURE 18 – Sélectionneur de fichiers

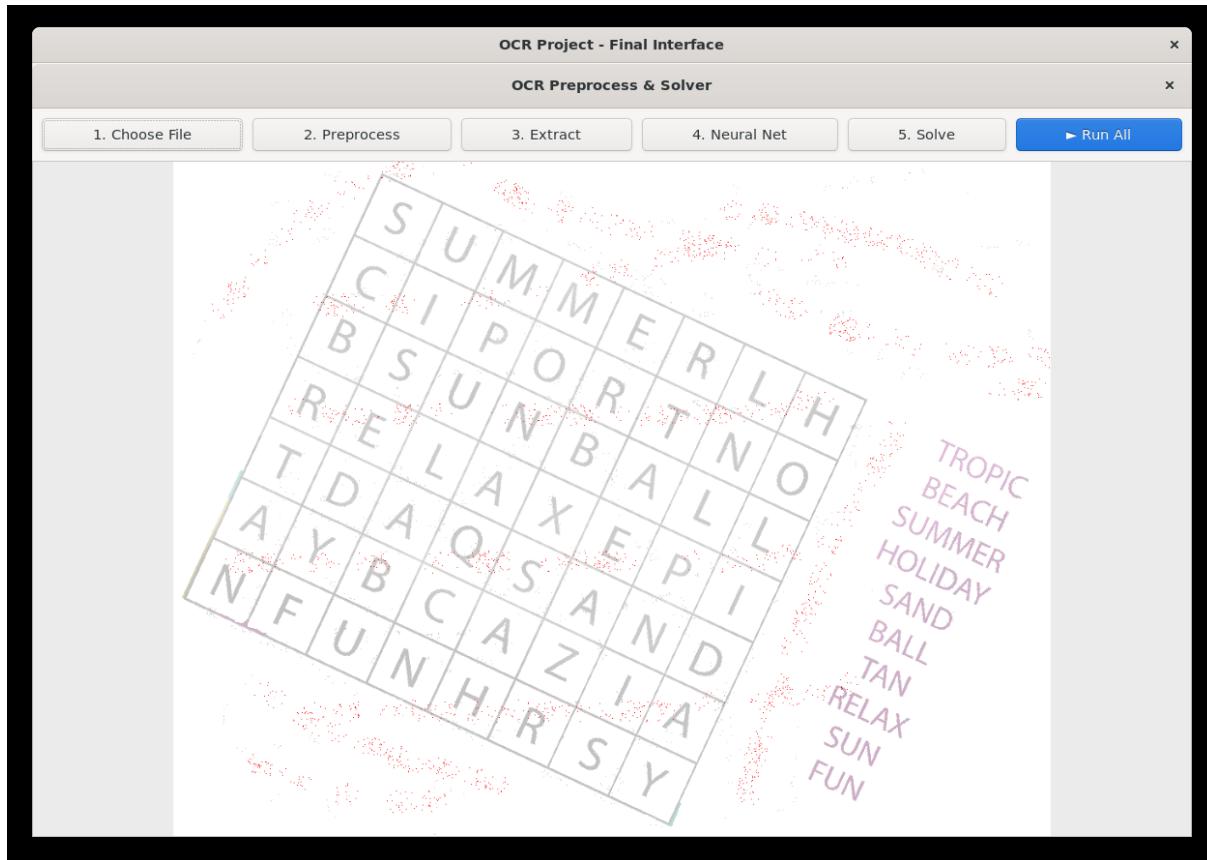


FIGURE 19 – Image sélectionnée chargée

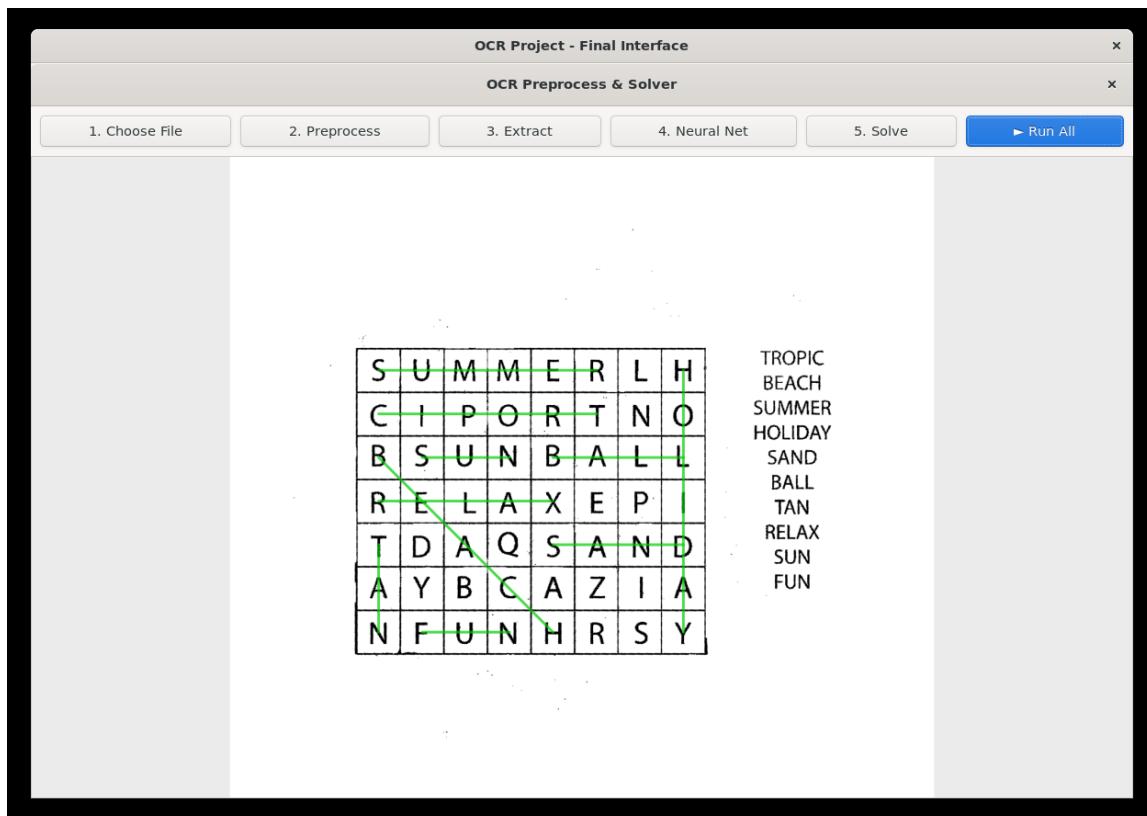


FIGURE 20 – Grille résolue

7 Conclusion et perspectives

Le projet "Word Search Solver" a constitué un défi technique complet et enrichissant, nous permettant d'explorer la chaîne intégrale d'un système de vision par ordinateur, depuis le traitement de l'image brute jusqu'à l'algorithmique de résolution. L'objectif ambitieux de transformer une simple photographie d'une grille de mots mêlés en une liste de mots résolus a nécessité l'intégration de domaines variés de l'informatique : traitement du signal 2D, analyse structurelle de documents, apprentissage automatique et algorithmique pure.

7.1 Bilan des réalisations techniques

Chaque maillon de la chaîne a été implémenté et testé, démontrant la viabilité de l'approche modulaire choisie :

- **Prétraitement robuste** : Le module de prétraitement, utilisant la binarisation d'Otsu et une correction d'inclinaison par projection, est capable de nettoyer efficacement des images issues de sources variées, compensant des conditions d'éclairage et d'orientation imparfaites.
- **Segmentation complexe** : L'algorithme d'analyse de mise en page, basé sur une approche hiérarchique descendante et des histogrammes de projection, réussit à déconstruire la structure du document pour isoler la grille, les mots de la liste et, *in fine*, chaque lettre individuelle, même dans des cas difficiles de fusion.
- **Reconnaissance optique par réseau de neurones** : Le Perceptron Multicouche (MLP) développé "from scratch" en C a atteint une précision supérieure à 90% sur notre dataset synthétique, validant son efficacité pour la classification des 26 lettres majuscules. Les mécanismes de sauvegarde/chargement et le mode "continue" ont facilité son entraînement.
- **Algorithme de résolution** : Le solver, utilisant une approche brute-force optimisée sur les 8 directions cardinales, fournit une solution rapide et exacte à partir des données reconnues.
- **Intégration graphique** : Le prototype d'interface GTK a posé les bases d'une application utilisateur finale, permettant déjà le chargement et la visualisation des images.

En conclusion, ce projet a été une expérience formatrice inestimable. Il nous a confrontés à la réalité du développement logiciel où la théorie doit s'adapter aux contraintes pratiques, où chaque module dépend de la fiabilité du précédent, et où le travail d'équipe et la communication sont aussi importants que la maîtrise technique. Le résultat final est un système fonctionnel qui, au-delà de la résolution de mots mêlés, constitue une plateforme solide et extensible pour d'autres applications d'analyse de documents.

8 Retours d'expérience individuels

8.1 Maxan Fournier (Prétraitement des images)

"Le prétraitement a été un défi intéressant car il est le premier maillon de la chaîne : si l'image n'est pas propre, tout le reste échoue. La difficulté majeure a été de trouver des algorithmes suffisamment généralistes pour gérer la grande variété d'images possibles (scans, photos, mauvais éclairage, papier froissé) sans tomber dans le piège d'une sur-optimisation pour un seul cas particulier.

La binarisation d'Otsu s'est révélée être un choix excellent, bien plus robuste que les seuils fixes que j'avais initialement envisagés. Mais c'est la correction de l'inclinaison (deskewing) qui a été la partie la plus satisfaisante à implémenter : voir l'algorithme trouver automatiquement l'angle de rotation optimal a été un vrai moment de réussite."

8.2 Cyril Dejouhanet (Détection et extraction)

"Ma partie, la segmentation, était sans doute l'étape la plus 'ingrate' mais cruciale : faire le pont entre une image binaire et des lettres individuelles. Le plus grand défi a été de gérer l'irrégularité. Sur le papier, une grille est parfaite, mais dans une image scannée, les lignes ne sont pas droites, les espacements varient, et le bruit peut connecter des lettres qui devraient être séparées.

L'approche hiérarchique (trouver la grille, puis les lignes, puis les cellules) s'est avérée bien plus stable que d'essayer de trouver des lettres directement. Le point le plus technique a été la séparation des lettres fusionnées (kerning). C'est une partie où l'on passe beaucoup de temps à ajuster des seuils pour trouver le bon équilibre, mais c'est très gratifiant quand l'algorithme parvient à découper proprement un mot complexe."

8.3 Tristan Druart (Réseau de neurones)

"Implémenter un réseau de neurones complet en C pur, sans utiliser de bibliothèques de haut niveau comme TensorFlow ou PyTorch, a été une expérience extrêmement formatrice pour comprendre les mathématiques sous-jacentes. La rétropropagation du gradient, les fonctions d'activation, la gestion des poids... coder tout cela m'a permis de démystifier le 'deep learning'.

Le principal obstacle a été l'entraînement. Au début, le réseau ne convergeait pas, ou restait bloqué sur des performances médiocres. J'ai dû expérimenter longuement avec beaucoup de paramètres, mais voir la courbe de précision grimper finalement au-dessus de 90% après des heures de calcul a été une grande satisfaction."

8.4 Martin Lemee (Solver et intégration)

"Arrivant en bout de chaîne, mon rôle était de donner du sens à tout le travail précédent. La partie algorithmique du solver était intéressante, notamment pour gérer efficacement la recherche dans les 8 directions et s'assurer que la complexité restait maîtrisée. C'est de l'algorithmique pure, ce qui changeait des problématiques de traitement d'image des autres modules.

Cependant, le plus gros défi a été l'intégration et le début de l'interface graphique avec GTK. C'est là que l'on se rend compte de l'importance de définir des interfaces claires entre les modules."