Reconnaissance Optique de Caractères (ROC)

Rapport de Soutenance n°2

Auteurs :

Édouard Baudouïn
Adem Çelik
Bashir Toufir

EPITA

Réalisé le : 13/12/2025

Table des matières

Table des matières

1 Introduction

Dans ce rapport, nous présentons le projet mené au **campus EPITA de Strasbourg**, dans le cadre du **cycle préparatoire**. L'objectif est de concevoir, en **langage C**, une interface capable de **résoudre automatiquement des grilles de mots mêlés** à partir d'une image fournie par l'utilisateur, avec des niveaux de difficulté différents.

Le système proposé se décompose en plusieurs étapes complémentaires. Tout d'abord, un algorithme détecte et **reconnaît la grille** dans l'image (redressement, normalisation et correction des perspectives), puis procède à la **segmentation** des cases pour isoler chaque caractère. Chaque lettre découpée est ensuite **classifiée par un réseau de neurones** entraîné à cet effet, de manière à reconstruire la grille au format texte. Sur cette base, un algorithme "solver" parcourt la matrice de caractères pour **trouver les mots cibles** dans toutes les directions usuelles (horizontale, verticale et diagonale, dans les deux sens). Afin de rendre l'outil accessible et agréable à utiliser, nous développons une **interface graphique** permettant de charger une image, d'afficher la grille reconnue et d'exporter le résultat. L'ensemble du projet met l'accent sur la **robustesse des traitements d'image**, la **précision de la reconnaissance de caractères** et la **réactivité de l'interface**.

Les contributions de l'équipe (*Édouard Baudouïn*, *Adem Çelik*, *Bashir Toufir*) s'organisent autour des trois axes suivants :

- **Prétraitements et détection de la grille** : binarisation, filtrage, détection des lignes, correction géométrique.
- **Segmentation et OCR** : découpe des cases et **réseau de neurones** pour la classification des lettres.
- **Résolution et interface** : recherche de mots dans la matrice, visualisation des résultats et ergonomie de l'application.

Pour la première soutenance, plusieurs éléments devaient être développés et présentés afin de démontrer la progression du projet. Ces tâches constituaient les premières étapes de la chaîne de traitement globale du programme, allant du chargement d'une image jusqu'à la preuve de concept du réseau de neurones.

Tout d'abord, il s'agissait d'assurer le **chargement d'une image** et la **supression des couleurs**, en convertissant les images sources en niveaux de gris, puis en noir et blanc. Cette étape de *binarisation* est essentielle pour simplifier les traitements ultérieurs, notamment la détection des contours et la segmentation des lettres.

Une autre fonctionnalité attendue concernait la **rotation manuelle de l'image**, permettant de corriger les défauts d'orientation sur les images d'entrée. Cette manipulation offre la possibilité d'aligner correctement la grille avant de procéder aux opérations de détection.

Vient ensuite la phase de **détection**, qui constitue le cœur du prétraitement. Elle comprend la localisation :
— de la **grille** principale contenant les lettres ;
— de la **liste de mots** à rechercher ;
— des **lettres présentes dans la grille** ;
— des **mots de la liste** et des lettres qui les composent.
Une fois ces zones identifiées, il est nécessaire d'effectuer le **découpage de l'image** afin d'extraire chaque caractère sous la forme d'une image individuelle. Ces images serviront de base pour l'entraînement et la reconnaissance des lettres par le réseau de neurones.

Parallèlement, une première version de l'algorithme de résolution a été développée sous la forme d'un programme en ligne de commande nommé `solver`. Ce dernier lit un fichier représentant la grille et un mot à chercher, puis retourne les coordonnées des lettres correspondantes si le mot est trouvé. Cet outil constitue le noyau logique de la recherche de mots dans la grille.

Enfin, une **preuve de concept du réseau de neurones** a été réalisée. Le but était d'implémenter un mini réseau capable d'apprendre la fonction logique XOR ($A.B + A'B$), où A et B sont deux variables booléennes. Cette étape a permis de valider la compréhension des mécanismes d'apprentissage supervisé, de rétropropagation et d'ajustement des poids internes du modèle, avant d'appliquer ces principes à la reconnaissance de caractères dans les images de grilles.

Pour la soutenance finale, il nous est demandé de présenter un programme complet, homogène et fonctionnel, capable d'exécuter l'ensemble de la chaîne de traitement définie dans le cahier des charges du projet OCR. Contrairement à la première soutenance, qui portait principalement sur des preuves de concept et des modules indépendants, cette seconde soutenance exige un système intégré démontrant clairement l'interopérabilité de toutes les composantes développées.

Le programme doit être en mesure d'appliquer l'ensemble des opérations de prétraitement nécessaires pour améliorer la qualité des images d'entrée, notamment :

— **redressement manuel** via un angle fourni par l'utilisateur ;
— **redressement automatique** de la perspective ou de la rotation ;
— **réduction du bruit** (filtrage, suppression des artefacts) ;
— **amélioration des contrastes** pour faciliter les phases de segmentation et d'OCR.

Ces étapes sont indispensables pour traiter les images des niveaux 2 et 3, qui comportent des imperfections telles que du bruit, une luminosité incorrecte ou une rotation aléatoire.

Contrairement à la première soutenance qui n'exigeait qu'une preuve de concept (exemple : apprentissage du XOR), la version finale doit intégrer :

— une **phase d'apprentissage supervisé** utilisant un jeu d'images de lettres correctement découpées ;
— un **modèle capable de reconnaître les caractères** présents dans la grille et dans la liste de mots ;
— un système de **sauvegarde et de rechargement des poids** du modèle entraîné.

Le réseau doit être suffisamment robuste pour traiter des caractères issus d'images bruitées, légèrement déformées ou mal contrastées.

À partir des caractères reconnus individuellement, le programme doit :
— reconstruire la **matrice de la grille** sous forme d'un tableau de caractères ;
— extraire et reconstruire la **liste complète des mots** à rechercher ;
— garantir un **ordre cohérent et fidèle** aux positions observées dans l'image.

Ces reconstructions sont essentielles pour permettre à l'algorithme de résolution d'opérer correctement.

Le solver doit être intégré au logiciel principal. À partir de la grille reconstruite, il doit :
— rechercher l'ensemble des mots dans les huit directions possibles ;
— renvoyer les coordonnées de début et de fin de chaque mot trouvé ;
— gérer correctement le cas des mots introuvables.

Le solver, initialement développé en ligne de commande, doit démontrer sa capacité à fonctionner au sein de l'interface finale.

L'application doit fournir un rendu visuel clair comprenant :
— la grille d'origine ou reconstruite ;
— les mots trouvés **encerclés** ou mis en évidence selon un format lisible ;
— une **image exportable** du résultat final.

La qualité de l'affichage fait explicitement partie de l'évaluation finale.

Enfin, un des éléments majeurs de la soutenance finale est la présence d'une interface graphique permettant :

— de charger une image depuis un fichier ;
— de visualiser l'image prétraitée, la grille identifiée et les lettres reconnues ;
— de lancer la résolution complète de la grille ;
— d'afficher et sauvegarder le résultat

La démonstration doit montrer une chaîne de traitement fluide, depuis le chargement de l'image jusqu'à l'affichage du résultat final.

Pour synthétiser, un projet est considéré comme conforme aux exigences de la soutenance finale s'il prend en charge :

1. le **prétraitement automatique et manuel** des images ;
2. la **détection complète** de la grille, des cases et de la liste de mots ;
3. la **segmentation** des lettres ;
4. la **reconnaissance** des caractères par un réseau de neurones entraîné ;
5. la **reconstruction** de la grille et de la liste de mots ;
6. la **résolution** automatique des mots cachés ;
7. l'**affichage graphique** du processus et des résultats ;
8. la **sauvegarde** de la grille résolue.

Cet ensemble constitue la base indispensable pour valider le projet lors de la soutenance finale.

2 Présentations personnelles

2.1 Édouard BAUDOUÏN

Je m'appelle **Édouard Baudouïn**, étudiant en **deuxième année à l'EPITA** sur le campus de **Strasbourg**. Passionné par les **nouvelles technologies** et plus particulièrement par l'**intelligence artificielle**, j'envisage de m'orienter vers cette **majeure** lors de mon cycle ingénieur. L'**IA** est pour moi un domaine en pleine expansion, et j'aimerais plus tard travailler dans une **structure souveraine** comme **Mistral AI**.

Sur le plan technique, j'ai pu développer mes compétences en **C** cette année, tout en consolidant mes acquis en **Python**, un langage que je maîtrise particulièrement bien. L'année précédente, j'ai eu l'occasion d'apprendre le **C#** et de mettre ces connaissances en pratique lors de la conception d'un **jeu vidéo en groupe** intitulé *Narcops Mexico*. Ce projet, première vraie expérience de travail collectif, m'a permis d'apprendre à **m'organiser efficacement** et à collaborer de manière structurée.

En parallèle de mes études, je produis de la **musique** sur le logiciel **FL Studio**, une activité qui me permet d'exprimer ma créativité et d'entretenir un **sens du rythme et de la précision** que je retrouve également dans le code.

J'apprécie particulièrement les **travaux de groupe**, qui favorisent la **communication**, le **partage de compétences** et la **progression collective**. En tant que **chef de projet** sur ce travail, j'ai à cœur de **coordonner efficacement l'équipe**, d'assurer une **répartition claire des tâches** et de garantir la **cohérence du projet** dans sa globalité.

2.2 Bashir TOUFIR

Je m'appelle **Bashir Toufir**, étudiant en **deuxième année à l'EPITA** sur le campus de **Strasbourg**. Né d'une famille où l'**informatique occupe une place centrale**, avec un père **ingénieur en informatique**, j'ai grandi dans un environnement qui a nourri ma **curiosité** et ma **passion** pour ce domaine. Depuis mon plus jeune âge, l'univers de la **technologie** a toujours éveillé en moi un certain intérêt, allant de la simple découverte des **jeux vidéo** à l'intérêt pour leur **conception** et leur **développement**.

Très vite, je me suis rendu compte que je ne voulais pas être qu'un simple utilisateur de ces technologies, mais que je souhaitais en devenir un **créateur**. Cette **passion pour l'informatique et le développement** s'est amplifiée au fil des années, me poussant à explorer différents **langages de programmation**. Chaque nouveau projet ou expérience dans ce domaine a renforcé ma **détermination** à poursuivre cette voie.

À **18 ans**, intégrer une **école d'ingénieurs en informatique** comme l'**EPITA** représente pour moi la fin d'un rêve et le début d'un parcours professionnel. Mon objectif est clair : acquérir les **compétences techniques et créatives** nécessaires pour me spécialiser dans le **développement de jeux vidéo**. En rejoignant l'EPITA, je me vois non loin d'atteindre mon plus grand rêve : devenir **ingénieur en informatique**.

Dans ce projet, je compte apporter ma **rigueur**, mon **esprit analytique** et ma **capacité à résoudre des problèmes** grâce à mes compétences en programmation développées au fil de mes expériences. Mon **expérience** et ma **passion** me permettront d'aider le groupe à **surmonter les défis techniques** et à proposer des **solutions réalisables et innovantes**.

2.3 Adem CELIK

Je m'appelle **Adem Celik** et je suis actuellement en **deuxième année à l'EPITA**, sur le campus de **Strasbourg**. Depuis toujours, je suis passionné par les **technologies de pointe**, en particulier par tout ce qui touche à l'**intelligence artificielle**, à l'**aéronautique** et aux **systèmes embarqués**. Ces domaines représentent pour moi le cœur de l'**innovation moderne** : ils allient **rigueur**, **performance** et **créativité**, trois valeurs auxquelles je m'identifie pleinement.

Au fil de mon parcours, j'ai pu développer de solides compétences en **langage C**, tout en perfectionnant ma maîtrise du **Python**, que j'utilise pour des **projets personnels** et des **prototypes techniques**. J'ai également eu l'occasion de travailler en **C#** sur un **projet de groupe** nommé *AMALGAM*, un **jeu vidéo** que nous avons conçu et développé collectivement. Ce projet m'a permis de découvrir la **réalité du travail d'équipe**, la **gestion du temps** et l'**importance d'une organisation claire** pour mener un projet à terme.

En dehors de mes études, je m'intéresse particulièrement au monde de la **finance** et du **trading**, notamment à l'**analyse graphique des marchés** et à l'évolution des **actifs financiers** comme le **XAU/USD**. J'aime observer les **tendances**, comprendre les **mouvements économiques** et les décisions qui influencent les marchés.

En parallèle, je travaille dans une **entreprise** où je gère à la fois le **site internet** et les **réseaux sociaux**. Cette expérience professionnelle me permet de mettre en pratique mes **compétences technologiques** tout en développant un vrai sens de la **polyvalence** et de la **responsabilité**.

3 Répartition des tâches

Dans le tableau ci-dessous, vous pourrez donc voir la répartition des charges de travail effectuées pour cette première soutenance, au sein de notre groupe :

Tâche	**Édouard**	**Bashir**	**Adem**
Algorithme solver	$\oplus$		
Noir et blanc (binarisation)		$\oplus$	
Réseau neuronal A-B	$+$	$\oplus$	
Interface (chargement image et rotation)	$\oplus$		$+$
Détection de grilles et de lettres			$\oplus$
Découpage des lettres et des mots		$+$	$\oplus$

Légende :

$\oplus$: activité principale
$+$: activité secondaire

Ces tâches ont été distribuées lors d'une réunion pour fixer les objectifs de chacun. Cependant, elles sont uniquement présente de manière à séparer le travail, lorsqu'un membre du groupe éprouvait des difficultés à réaliser sa tâche, le reste du groupe venait apporter du soutien afin de réaliser la tâche.

4 Avancement du projet : première soutenance

4.1 Avancement général

Dans l'ensemble, le projet avance de manière efficace. Nous avons tous réalisé nos tâches avec envie, dans les temps que nous nous étions donnés.

Les difficultées principales que nous avons rencontrés étaient vastes, entre la prise en main réelle du langage C, mais aussi avec les choix que nous avons du faire et les restrictions de bibliothèques.

Vous pourrez observer ci-dessous un tableau regroupant l'avancement des tâches pour la soutenance 1 et 2 :

Tâche	**Avancement prévu**	**Avancement réel**
Chargement de l'image	100%	100%
Rotation manuelle	100%	100%
Rotation automatique	0%	0%
Interface	50%	50%
Réseau neuronal XOR	100%	100%
Détection et découpage de grille	100%	75%
Réseau de neurones final	0%	0%
Algorithme de résolution de grille	100%	100%
Binarisation des images	100%	100%
Relation entre les algorithmes et les fonctionnalités	0%	0%
Tâches pour la soutenance 1	100%	100%
Tâches pour la soutenance 2	0%	0%

Table 1 – Tableau d'avancement général du projet

4.2 Avancements individuels

4.2.1 Édouard

Recherche de mots dans une grille (`solver.c` et `main.c`)

Pour cette partie, j'ai créé un petit programme capable de **chercher des mots dans une grille de lettres**. L'idée est simple : le programme lit une grille depuis un fichier texte, puis parcourt toutes les positions possibles pour trouver chaque mot de la liste. Le mot peut être lu dans n'importe quelle direction (horizontalement, verticalement ou en diagonale).

J'ai d'abord écrit une fonction `load_grid()` qui charge la grille depuis un fichier texte. Cette fonction lit chaque ligne, enlève les espaces et les retours à la ligne, puis vérifie que toutes les lignes ont la même longueur pour éviter les erreurs. Si tout est correct, elle stocke la grille dans un tableau à une dimension. J'ai aussi ajouté des vérifications pour détecter une grille vide ou mal formatée.

Ensuite, la fonction `search_word()` s'occupe de la recherche elle-même. Pour chaque case de la grille, le programme essaie de lire le mot dans les huit directions possibles : haut, bas, gauche, droite et les quatre diagonales. Pour éviter de sortir de la grille, je vérifie que la fin du mot reste bien dans les limites avant de tester une direction. La fonction `match_at()` compare ensuite chaque lettre une par une (en ignorant la différence entre majuscules et minuscules). Si toutes les lettres correspondent, la position de départ et de fin du mot est enregistrée.

Difficultés rencontrées :

Au début, j'ai eu du mal à **charger correctement la grille**. Certaines lignes contenaient des espaces ou des retours à la ligne cachés, ce qui cassait la lecture. J'ai donc ajouté un passage pour "nettoyer" chaque ligne et supprimer tous les caractères inutiles. J'ai aussi dû faire attention à la mémoire, car j'utilisais des allocations dynamiques avec `malloc` et `realloc`, et il fallait penser à tout libérer à la fin.

L'autre difficulté venait de la **gestion des directions**. Au début, je faisais des erreurs d'indices et je sortais parfois de la grille sans m'en rendre compte, ce qui faisait planter le programme. J'ai réglé ça en ajoutant des vérifications pour ne pas dépasser les bords avant de lancer la comparaison. J'ai aussi dû faire attention à ne pas confondre les coordonnées `x` et `y` dans les boucles.

Une autre petite galère a été la **lecture du fichier de mots**. Certains mots contenaient des espaces ou des retours à la ligne, donc j'ai ajouté un nettoyage similaire à celui de la grille pour éviter les erreurs.

Résultat final :

Une fois toutes ces étapes terminées, le programme lit correctement la grille et le fichier de mots. Pour chaque mot, il affiche s'il a été trouvé ou non, et dans le cas positif, il montre les coordonnées de départ et de fin. Ce projet m'a vraiment aidé à mieux comprendre la gestion des fichiers, des pointeurs, et surtout les parcours de tableaux en C.

```
crbz@LAPTOP-AOUG0FIB:/mnt/c/Users/miste/Documents/GitHub/s1OCR/solver$ ./solver_test
Grille 5x6 chargée.
HELLO : trouvé de (0,4) à (4,4)
GRID : trouvé de (1,2) à (4,2)
SOLVE : trouvé de (0,3) à (4,3)
SEARCH : trouvé de (0,1) à (5,1)
BASHIRADEM : non trouvé
```

FIGURE 1 – Sortie après un test du script solver

Interface et rotation d'image

Pour l'interface du projet, j'ai choisi **GTK** plutôt que SDL. La raison est simple : avec GTK, je peux **utiliser du CSS** (boutons, fonds, labels), et c'est directement **disponible sur le NixOS de l'école**. Au final, ça me donne une interface plus propre et moderne, sans devoir tout recoder à la main.

Pourquoi GTK (et pas SDL) ?

— **CSS natif** : j'applique un style via un `GtkCssProvider` (classes comme `.accent-btn`, `.hero-enter`, `.home-dim`) pour avoir des boutons arrondis, des panneaux semi-transparents, etc.
— **Intégration facile** : GTK est **déjà dispo sur le NixOS** de l'école, donc l'utiliser était une option pratique pour éviter SDL.
— **Widgets prêts à l'emploi** : fenêtres, `GtkHeaderBar`, `GtkFileChooser`, `GtkImage`, `GtkStack`, `GtkScrolledWindow`... J'ai pu me concentrer sur la logique au lieu du "plomberie UI".

Structure de l'interface

— **Header bar** (`build_header_bar`) : un bouton *Ouvrir* (icône `document-open`), un `GtkSpinButton` pour choisir l'angle (0–360°, pas de 1, avec 1 décimale), et un bouton *Rotation*.
— **Deux pages** dans un `GtkStack` :
  — **Home** (`build_home_page`) avec un fond (image si dispo, sinon icône), un panneau stylé (`.home-dim`) et un bouton *Entrer* (`.hero-enter`).
  — **Main** (`build_main_page`) avec un `GtkScrolledWindow` qui contient l'image et un label d'état (*Aucune image chargée.*).
— **Ouverture d'image** (`on_open_image`) : `GtkFileChooserNative`, filtre PNG/JPEG, puis `gtk_image_set_from_file`. Je garde le chemin en `g_object_set_data(...,"current-filepath",...)` et j'affiche un message dans la barre d'état.

Rotation d'image

— **Conversion Pixbuf → RGBA** (`pixbuf_to_rgba`) : je récupère les pixels de `GdkPixbuf` et je les copie dans un buffer `unsigned char` en **RGBA**, en forçant l'alpha à 255 si l'image n'en a pas.
— **Calcul de la taille de destination** : avant de tourner, je calcule la taille **minimale** qui contient toute l'image *sans la couper*. J'utilise cos et sin de l'angle (en radians) pour déterminer la largeur/hauteur nécessaires (`rotate_rgba_nn_padauto`).
— **Rotation (nearest-neighbor)** : pour chaque pixel de sortie, je fais un *mapping inverse* vers la source (rotation inverse autour du centre). Je prends le pixel le plus proche (NN) pour rester simple et rapide. Les zones qui tombent en dehors restent transparentes (buffer initialisé à 0).
— **Cas angle nul** : si l'angle est 0°, je me contente de **centrer-coller** l'image source dans le buffer de destination, sans faire de trigonométrie inutile.
— **Affichage et sauvegarde** : je reconstruis un `GdkPixbuf` à partir du buffer RGBA et je l'affiche dans le `GtkImage`. Je **vide le dossier** `out/` (fonction `clear_directory_contents`), puis j'enregistre l'image tournée en `PNG` sous `out/<nom>_rotX.Y.png` (l'angle est inclus dans le nom).

Petits détails pratiques

— **Centrage de la vue** : après chaque rotation, je recentre les barres de scroll (`center_scroller`) pour voir l'image au milieu.
— **Gestion des messages** : j'ai une fonction `update_status_label` pour afficher ce qui se passe (*Image chargée*, *Rotation appliquée*, *Échec sauvegarde*, etc.).
— **Nettoyage** : j'utilise des `g_free`, `g_object_unref` et `free` au bon moment pour éviter les fuites mémoire.

Difficultés rencontrées

— **Chemins et formats** : certaines images avaient de l'alpha ou des dimensions étranges. La conversion Pixbuf→RGBA m'a évité pas mal de surprises.
— **Indices et géométrie** : j'ai eu quelques erreurs de débordement au début (x/y inversés, offsets de stride, et conversions `double` → `int`). Le *mapping inverse* a réglé les trous visuels.
— **Taille de sortie** : si on ne calcule pas bien la taille cible, l'image est coupée après rotation. J'ai donc ajouté le calcul des dimensions « *pad-auto* » pour tout contenir, puis un centrage.
— **UX** : sans CSS, l'appli faisait "outil brut". Avec GTK + CSS, j'ai pu avoir un header propre, des boutons lisibles, et une page d'accueil un peu plus sympa (`overlay` + fond).

Bilan

Au final, GTK m'a fait gagner du temps et m'a permis d'avoir une appli qui **fait le job** et qui **a l'air propre**. La rotation n'est pas parfaite comme du bilinéaire, mais elle est **simple, rapide** et suffisante pour mon usage.

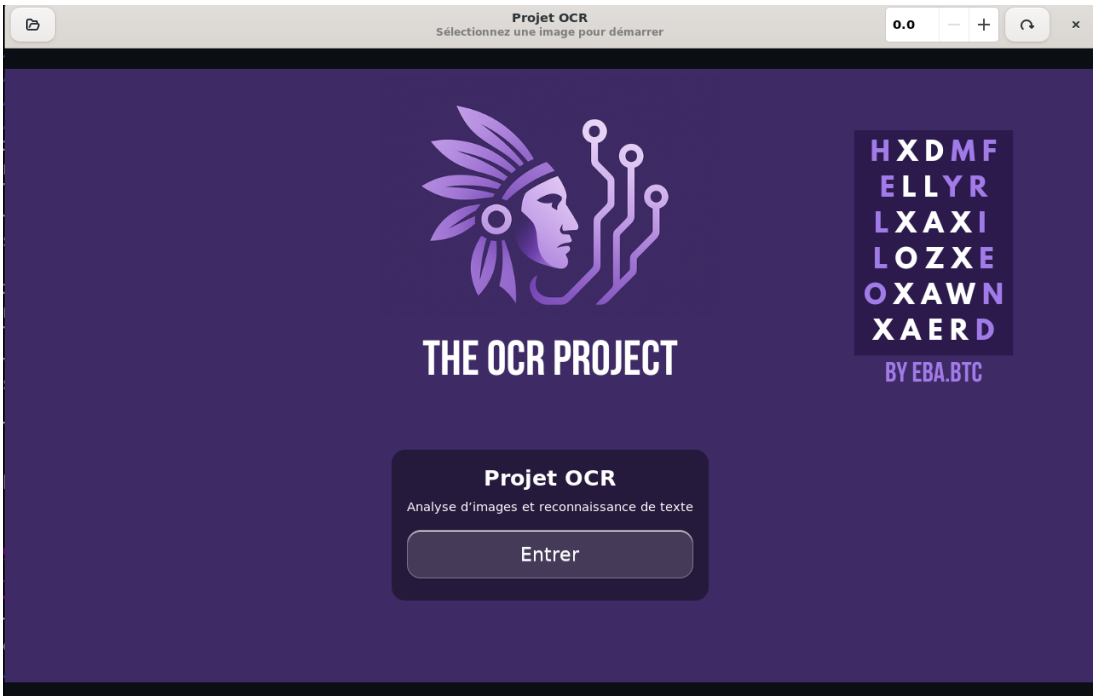

FIGURE 2 – Aperçu de l'interface du projet

4.2.2 Adem

Rappel des tâches :

Détection de grilles et de lettres / Découpage des lettres et des mots

Mon objectif :

Mon objectif était de développer un programme capable de **prendre en entrée une image contenant une grille**, de **détecter automatiquement les lignes horizontales et verticales**, puis d'**extraire chaque case** pour les sauvegarder individuellement sous forme d'images.

L'algorithme est conçu pour traiter un dossier complet d'images et générer automatiquement des sous-dossiers contenant les cases extraites.

Structure générale du code :

Le programme repose sur une série de fonctions indépendantes permettant de gérer chaque étape du script.

`Structure ImageSimple :`

Cette structure stocke les informations essentielles d'une image :
— sa largeur et sa hauteur,
— un pointeur vers les pixels (en niveaux de gris).
Elle permet de manipuler les images de manière légère et efficace.

`liberer_image(ImageSimple *image) :`

Libère la mémoire allouée pour les pixels chargés avec la bibliothèque `stb_image`. Cette étape est indispensable pour éviter les fuites mémoire, surtout lors du traitement d'un grand nombre d'images.

`est_extension_image(const char *nom) :`

Vérifie que le fichier possède une extension d'image valide parmi les formats pris en charge : `.png`, `.jpg`, `.jpeg`, ou `.bmp`. Cela permet directement de savoir si une image est compatible ou non à l'algorithme, et de marquer un cas d'arrêt sinon.

`creer_repertoire(const char *chemin) :`

Crée un répertoire s'il n'existe pas encore, en assurant une compatibilité entre Windows et Linux (pour nos tests, car nous avons fait le projet sur Windows et compiler via une WSL Ubuntu). Cette fonction est essentielle pour stocker les images découpées sans dépendre du système d'exploitation.

`joindre_chemin(...) :`

Assemble proprement un chemin de dossier et un nom de fichier. Elle gère automatiquement les séparateurs de chemin (/ ou \).

`nom_sans_extension(...) :`

Récupère le nom du fichier sans extension ni chemin. Cette fonction est utile pour nommer le sous-dossier de sortie correspondant à l'image d'origine (Elle récupère le nom de l'image pour en faire le nom du dossier).

`charger_image(const char *chemin, ImageSimple *out) :`

Charge une image depuis le fichier `clean_grid` et la convertit en une structure `ImageSimple`. L'image est directement lue en niveaux de gris pour simplifier les traitements de détection.

`ecrire_image(const char *chemin, const unsigned char *pixels, int L, int H) :`

Écrit une image PNG à partir d'un tableau de pixels en niveaux de gris. Chaque découpe de case est sauvegardée dans un fichier distinct portant un nom unique d'un positionnement dans une matrice, voir l'image ci-dessous.

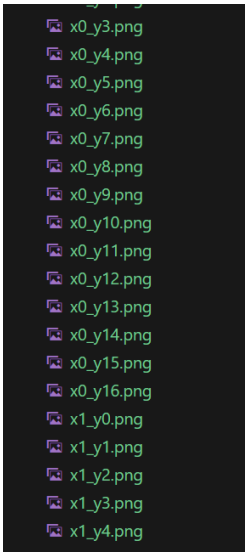

FIGURE 3 – Extrait des fichiers lettres renvoyés

`collecter_lignes(..., bool horizontal, int **lignes) :`

Analyse l'image pour repérer les positions des lignes de la grille :
— Si `horizontal` vaut `true`, la fonction parcourt l'image ligne par ligne (axe Y).
— Sinon, elle parcourt colonne par colonne (axe X).
Elle compte les pixels « noirs » (ou sombres) afin de déterminer les emplacements où se trouvent les traits de la grille. Les coordonnées des lignes détectées sont ensuite stockées dans un tableau dynamique (malloc).

`decouper_grille(const char *chemin_entree, const char *rep_sortie) :`

Fonction principale du traitement :

1. Charge l'image d'entrée.
2. Détecte les lignes horizontales et verticales grâce à `collecter_lignes`.
3. Calcule les intersections des lignes pour déterminer les limites de chaque case.
4. Extrait chaque zone rectangulaire (lettres).
5. Enregistre chaque case dans le dossier de sortie au format PNG.

.

`pour_chaque_image(const char *rep_in, cb, ctx) :`

Parcourt un dossier et applique une fonction de traitement à chaque image reconnue. Ce mécanisme de rappel (callback) permet de traiter automatiquement un ensemble d'images sans modifier la logique de traitement individuelle.

`rappel_decoupe(...) :`

Fonction adaptatrice servant de passerelle entre `pour_chaque_image` et `decouper_grille`. Elle configure le contexte et transmet les bons chemins de sortie.

`decouper_lettres_dans_repertoire(const char *rep_in, const char *rep_out) :`

Prépare les répertoires de sortie et lance le traitement complet sur l'ensemble des images du dossier d'entrée.

`main(int argc, char **argv) :`

Point d'entrée du programme. Lit les chemins d'entrée et de sortie depuis la ligne de commande (ou utilise des valeurs par défaut), puis exécute la découpe sur toutes les images détectées.

Difficultés rencontrées :

J'ai rencontré de nombreuses difficultées durant le développement :

— **Détection des lignes :** La détection des traits de grille dépend fortement du contraste et de la netteté des lignes. Si les contours sont flous ou si la grille est légèrement inclinée, les comptages de pixels deviennent moins fiables, rendant les détections incomplètes ou décalées, ce qui faussait mes résultats.

— **Gestion mémoire :** Chaque image chargée, chaque tableau de lignes détectées et chaque découpe nécessite une allocation dynamique. Il a fallu s'assurer que chaque ressource est correctement libérée pour éviter toute fuite mémoire, surtout lors du traitement de grands lots d'images, l'allocation de mémoire à été difficile à prendre en main car j'ai eu du mal à comprendre ce système depuis que l'on a commencé à apprendre le C.

— **Compatibilité multi-plateforme :** Les différences entre Windows et Linux concernent principalement la gestion des chemins et la création des dossiers. Il a fallu faire en sorte que pour la soutenance le projet puisse être compilé sur Nixos, donc nous avons du prendre des dispositions pour être sur de ne pas avoir d'erreur.

— **Découpage précis :** Lors de la séparation des cases, les coordonnées des lignes détectées doivent être utilisées avec précision. Un léger décalage peut fausser tout mes résultats.

Conclusion :

Malgré les contraintes techniques et les difficultés liées à la détection des lignes, la version actuelle du programme remplit pleinement son objectif, sauf dans le cas le plus dur avec les images de niveau 3 sans grille, j'ai déjà une idée d'algorithme qui consisterait à créer une nouvelle grille par dessus, mais par manque de temps je n'ai pas pu finaliser le script.

4.2.3 Bashir

Rappel des tâches :

Développement du réseau neuronnal pour la fonction XOR la binarisation des couleurs des images

Implémentation du réseau neuronal :

Avant de passer à la version finale en langage C, j'ai d'abord développé une première version du réseau neuronal en Python. Ce choix m'a permis de bien comprendre le fonctionnement de l'apprentissage supervisé, notamment la propagation avant, la rétropropagation et la descente de gradient. Python est plus pratique pour faire des tests rapidement, donc j'ai pu expérimenter et ajuster les calculs plus facilement.

J'ai créé un prototype dans un fichier `nn_python.py`, avec les mêmes fonctions principales que celles que j'allais ensuite coder en C : `forward`, `backward`, `bce_loss` et `train_xor`.

Une fois que tout fonctionnait correctement et que les résultats étaient cohérents, j'ai traduit le modèle en C, en reprenant exactement les mêmes formules et comportements que dans la version Python. Malgré tout, certaines choses m'ont fait avoir des difficultés, car le Python et le C sont deux langages différents, le C demande au niveau des variables plus de détails, au niveau des types principalement, ce qui m'a demandé une réfléxion supplémentaire. C'était donc une étape qui était assez difficile.

Cette étape m'a permis de vérifier la fiabilité du programme tout en respectant les contraintes techniques de la SAÉ, c'est-à-dire une exécution native sans dépendances externes.

Choix de l'algorithme : perceptron multicouche (MLP)

Le modèle que j'ai choisi est un **perceptron multicouche (MLP)** comportant une seule couche cachée. C'est une structure simple, mais largement suffisante pour reconnaître des motifs binarisés.

Avantages du MLP :

— Facile à implémenter entièrement en langage C, sans utiliser de bibliothèques externes ;
— Compatible avec le test du **XOR**, ce qui permet de valider la capacité du réseau à apprendre des relations non linéaires ;
— Bien adapté aux données binaires (valeurs 0 ou 255).

.

Structures de données :

Le réseau neuronal repose sur deux structures principales, Params et Grads, qui contiennent respectivement les poids et biais du modèle ainsi que leurs gradients calculés pendant la rétropropagation. Les poids sont initialisés dans [-0.5, 0.5] via `rand_uniform()`, et les biais à zéro.

```c
typedef struct {
    double W1[2][2];
    double b1[2];
    double W2[1][2];
    double b2[1];
} Params;

typedef struct {
    double dW1[2][2];
    double db1[2];
    double dW2[1][2];
    double db2[1];
} Grads;
```

FIGURE 4 – Structures Params et Grams.

Fonction Forward :

La fonction `forward()` applique les équations suivantes :

$$z_1 = W_1 \cdot x + b_1$$
$$a_1 = \sigma(z_1)$$
$$z_2 = W_2 \cdot a_1 + b_2$$
$$\hat{y} = \sigma(z_2)$$

```
void forward(const double x[2], const Params* p, double a1[2], double* y_hat)
{
    double z1_0 = p->W1[0][0]*x[0] + p->W1[0][1]*x[1] + p->b1[0];
    double z1_1 = p->W1[1][0]*x[0] + p->W1[1][1]*x[1] + p->b1[1];
    a1[0] = sigmoid(z1_0);
    a1[1] = sigmoid(z1_1);
    double z2 = p->W2[0][0]*a1[0] + p->W2[0][1]*a1[1] + p->b2[0];
    *y_hat = sigmoid(z2);
}
```

FIGURE 5 – Fonction Forward

Nous avons également utilisé une **fonction sigmoïde** comme fonction d'activation dans notre réseau neuronal. Cette fonction est définie par la formule suivante :

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

La sigmoïde transforme n'importe quelle valeur réelle en une valeur comprise entre **0 et 1**. Cela permet de normaliser la sortie du neurone et de l'interpréter comme une **probabilité**.

Dans notre cas, ce comportement est particulièrement utile, car le réseau doit reconnaître des motifs binarisés (0 ou 1). Grâce à la sigmoïde, les sorties proches de 0 indiquent une non-activation du neurone, tandis que celles proches de 1 indiquent une activation forte.

Fonction de coût :

La **Binary Cross Entropy** mesure la différence entre la sortie prédite et la sortie réelle.

$$L = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

Cette fonction punit fortement les erreurs de prédiction et fournit la dérivée $(\hat{y} - y)$ utilisée lors de la **rétropropagation**.

```c
void bce_loss(double y_hat, double y, double* loss, double* dL_dyhat)
{
    double eps = 1e-12;
    double p = fmin(fmax(y_hat, eps), 1.0 - eps);
    *loss = -(y * log(p) + (1 - y) * log(1 - p));
    *dL_dyhat = y_hat - y;
}
```

FIGURE 6 – Fonction de coût

Rétropropagation :

Les **gradients des poids** sont calculés puis appliqués grâce à la **descente de gradient**, selon la règle suivante :

$$W = W - \eta \times \frac{\partial L}{\partial W}$$

où :
— W représente les poids du réseau,
— η est le taux d'apprentissage (*learning rate*),
— et $\frac{\partial L}{\partial W}$ est le gradient de la fonction de perte par rapport aux poids.
Cette étape est essentielle pour **ajuster les poids** et **minimiser la perte** au fil des "EPOCH" d'apprentissage.

Entraînement sur XOR :

Le réseau a été entraîné sur le jeu logique **XOR**, avec les données suivantes :

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Ce petit jeu de données sert à vérifier que le réseau est capable d'apprendre des relations non linéaires. Le test du XOR est souvent utilisé parce qu'un modèle linéaire ne peut pas le résoudre, donc c'est un bon moyen de voir si le réseau apprend vraiment.

```
void train_xor(int epochs, double lr, Params* p)
{
    double X[4][2] = { {0,0}, {0,1}, {1,0}, {1,1} };
    double Y[4] = { 0, 1, 1, 0 };
    int order[4] = {0,1,2,3};
    Grads g;

    for (int e = 0; e < epochs; e++) {
        shuffle4(order);
        for (int k = 0; k < 4; k++) {
            int i = order[k];
            double a1[2], y_hat, loss, dL_dyhat;
            forward(X[i], p, a1, &y_hat);
            bce_loss(y_hat, Y[i], &loss, &dL_dyhat);
            backward(X[i], a1, y_hat, p, dL_dyhat, &g);
            apply_grads(p, &g, lr);
        }
    }
}
```

FIGURE 7 – Fonction de coût

```
Combien d'epochs pour entraîner ? (ex: 5000) : 5000

Résultats (sorties réelles entre 0 et 1) :
[0, 0] -> 0.026078645935443012
[0, 1] -> 0.9718842221731657
[1, 0] -> 0.9717247958393046
[1, 1] -> 0.0352978583115826
PS C:\Users\miste\Documents\GitHub\s1OCR>
```

FIGURE 8 – Sortie du script en Python pour la fonction XOR

Binarisation des images

Pour traiter les images avant de les passer au réseau, j'ai créé un module appelé `binary.c`. Ce programme prend une image couleur et la convertit en noir et blanc selon sa luminosité. J'ai utilisé la formule suivante pour calculer la luminance de chaque pixel :

$$Y = 0.299R + 0.587G + 0.114B$$

Cette formule donne une valeur entre 0 et 255 selon la clarté du pixel. Si la valeur est supérieure à un certain seuil, le pixel devient blanc (255), sinon noir (0). Cette étape sert à **simplifier le travail du réseau** en supprimant les couleurs inutiles pour ne garder qu'une image nette et facile à analyser.

Pour déterminer le seuil automatiquement, j'ai ajouté une méthode appelée **méthode d'Otsu**. Elle calcule le seuil optimal en fonction de la distribution des niveaux de gris dans l'image, ce qui permet d'obtenir un résultat correct même quand la luminosité change d'une image à l'autre. Cependant, j'ai aussi prévu une option pour définir manuellement le seuil depuis la ligne de commande, ce qui m'a aidé à tester différents cas.

Difficultés rencontrées :

La partie la plus compliquée au début a été la gestion de la mémoire et le chargement des images avec **STB Image**. Je devais bien penser à libérer chaque zone mémoire pour éviter les fuites. J'ai aussi eu quelques problèmes avec la transparence (canal alpha) : certaines images avaient un fond transparent, ce qui rendait la conversion bizarre. Pour régler ça, j'ai décidé d'appliquer la couleur du pixel sur un fond blanc avant de calculer la luminance.

L'autre difficulté venait du calcul du seuil. Quand j'utilisais une valeur fixe, certaines images ressortaient complètement noires ou blanches, donc la méthode d'Otsu m'a vraiment simplifié la vie. Enfin, j'ai dû gérer la compatibilité entre Windows et Linux pour la création du dossier `out/`, en utilisant une petite macro qui s'adapte selon le système.

Au final, ce module m'a permis de bien comprendre comment fonctionne la **binarisation d'image**, et surtout comment transformer une image couleur en données exploitables pour un réseau neuronal.

Conclusion sur la binairisation :

Finalement, nous avons un script parfait pour rendre des images en noir et blanc, on pourra l'utiliser pour la suite du projet. Pour l'instant, ce script est séparé des autres pour le présenter à la soutenance, mais sera lié pour la soutenance finale.

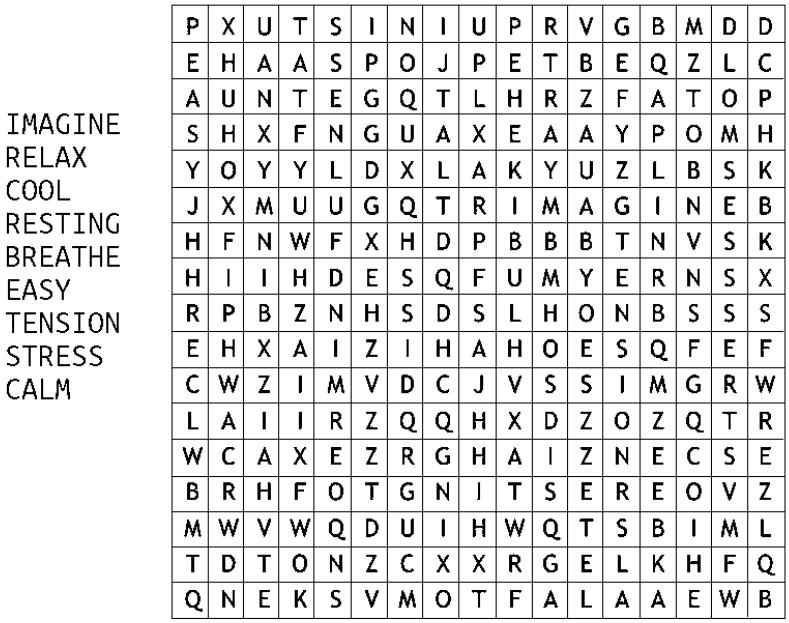

FIGURE 9 – Sortie après la binarisation de l'image "Easy"

5 Avancement du projet : soutenance finale

5.1 Avancement général

Pour la soutenance finale, nous avons rencontré des difficultées bien plus récurrentes que lors de la première présentation, sur tout les aspects.

Nous avons eu des difficultés sur toutes les parties du projet, que ce soit la détection de grille, il fallait qu'elle marche parfaitement pour la suite. Pour le réseau neuronal, la compréhension de chaque lettre était également difficile, et le réseau nécéssite un bon entraînement pour réussir. La difficultée suivante a été d'attendre ces résultats pour tout mettre en lien afin de rendre le projet final.

Vous pourrez observer ci-dessous un tableau regroupant l'avancement des tâches pour la soutenance 1 et 2 :

Tâche	**Avancement prévu**	**Avancement réel**
Chargement de l'image	100%	100%
Rotation manuelle	100%	100%
Rotation automatique	100%	100%
Interface	100%	100%
Réseau neuronal XOR	100%	100%
Détection et découpage de grille	100%	100%
Réseau de neurones final	100%	75%
Algorithme de résolution de grille	100%	100%
Binarisation des images	100%	100%
Relation entre les algorithmes et les fonctionnalités	100%	100%
Tâches pour la soutenance 1	100%	100%
Tâches pour la soutenance 2	100%	95%

Table 2 – Tableau d'avancement général du projet

5.2 Avancements individuels

5.2.1 Édouard

Mise en relation des algorithmes dans l'interface finale et rotation automatique (`interface.c`)

Dans cette partie du projet, j'ai implémenté un module complet de **rotation automatique** ainsi que la mise en lien de l'ensemble des scripts au sein d'une interface graphique construite avec GTK. L'objectif était de permettre à l'utilisateur de charger une image, puis d'enchaîner automatiquement les différentes étapes de traitement : rotation, binarisation, extraction et enfin résolution. Chaque fonction développée dans `ocr_window.c` est directement reliée aux boutons de l'interface et s'exécute en mettant à jour l'état interne et l'affichage.

Rotation automatique

La rotation automatique repose sur un algorithme analysant l'orientation de la grille présente dans l'image. La procédure se déroule en plusieurs étapes :

- conversion de l'image affichée en un tableau RGBA ;
- génération d'une version en niveaux de gris, puis binarisation grâce au seuil d'Otsu ;
- projection verticale et horizontale de l'image pour plusieurs angles ;
- sélection de l'angle maximisant la variance des projections, signe d'un alignement optimal ;
- rotation réelle de l'image via `rotate_rgba_nn_padauto()`, qui applique une transformation pixel par pixel avec centrage automatique ;
- sauvegarde de l'image corrigée dans `out/working.png`.

La fonction centrale, `rotate_image()`, remplace l'affichage courant par l'image tournée, recalcule ses dimensions, recentre la zone de visualisation et met à jour le statut utilisateur.

Mise en relation de l'ensemble des scripts

L'interface gère l'intégralité du pipeline de traitement. Chaque bouton appelle une fonction précise :

— **Ouverture d'image :** `on_open_image()` charge l'image choisie, la copie dans le dossier `out`, l'affiche et réinitialise l'historique d'opérations.

— **Rotation automatique :** `on_auto_rotate_clicked()` recharge l'image d'origine, détecte l'angle, applique la rotation et enregistre l'image corrigée.

— **Binarisation :** `on_binarize_clicked()` convertit l'image en niveaux de gris, calcule le seuil optimal puis produit et affiche une version binaire.

— **Extraction et résolution :** les boutons correspondants appellent les scripts externes pour traiter la grille et résoudre la grille.

Afin d'assurer la cohérence entre les opérations, des métadonnées sont stockées directement dans l'objet image via `g_object_set_data()`. Cela permet notamment de savoir :

— quel est le fichier actuellement affiché ;
— quelle a été la dernière opération effectuée ;
— si une binarisation doit suivre automatiquement une rotation.

Ainsi, l'ensemble des modules communique de manière fluide au sein de l'interface.

Difficultés rencontrées

Plusieurs obstacles se sont présentés durant l'implémentation.

Gestion de la rotation et des indices. Au début, la rotation générait parfois des dépassements de mémoire ou des images incorrectement centrées. Il a fallu recalculer précisément :

— la taille de l'image après rotation ;
— la position du centre source et destination ;
— les conversions entre coordonnées réelles et indices du tableau.

Détection d'angle instable. Les premières versions du détecteur retournaient des angles imprécis. L'amélioration est venue de :

— la combinaison des projections verticales et horizontales ;
— la réduction de la zone analysée pour éviter le bruit sur les bords ;
— un affinage de l'angle autour de la meilleure estimation.

Mise à jour de l'image dans GTK. GTK conservant des références internes sur les `GdkPixbuf`, chaque rotation devait impérativement :

— régénérer un nouveau `GdkPixbuf` ;
— sauvegarder dans le fichier `working.png` ;
— mettre à jour les métadonnées liées à l'image.

Enchaînement automatique des traitements. Pour éviter des séquences incohérentes telles que Auto → Binariser → Auto, j'ai introduit des marqueurs internes tels que `"last-op"` et `"auto-then-binarize"`.

Résultat final

L'application est désormais capable de :

— charger une image ;
— détecter et corriger automatiquement son orientation ;
— binariser l'image tournée ;
— préparer l'extraction et la résolution ;
— afficher un statut clair pour chaque opération.

Cette partie du projet m'a permis d'approfondir la manipulation d'images en C, les transformations géométriques, la gestion de la mémoire, ainsi que l'intégration d'algorithmes complexes dans une interface utilisateur ergonomique. J'ai eu énormément de difficultées à implémenter toutes ces fonctionnalités, car j'ai du attendre la création et finalisation de toutes les fonctionnalités pour les implémenter. J'ai donc passer ce temps à aider les autres membre du groupe à réaliser leur tâches.

Bilan

Au final, GTK m'a fait gagner du temps et m'a permis d'avoir une appli qui **fait le job** et qui **a l'air propre**. La rotation n'est pas parfaite comme du bilinéaire, mais elle est **simple, rapide** et suffisante pour mon usage. Ensuite, GTK à été utile pour implémenter les boutons et le reste des fonctionnalités.

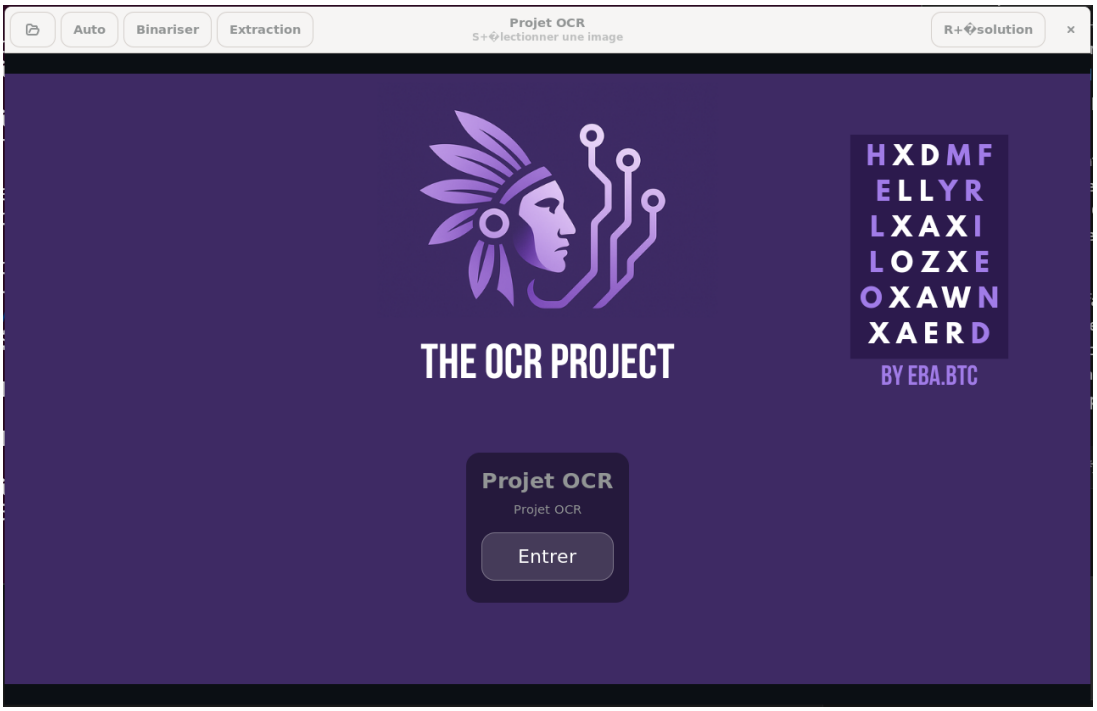

FIGURE 10 – Aperçu de l'interface du projet

5.2.2 Adem

Rappel des tâches :

Détection de grilles et de lettres / Découpage des lettres et des mots

sectionsectionTraitement Image

Dans le cadre de ce projet, nous avons conçu et mis en œuvre une chaîne de traitement d'images complète dont l'objectif est d'extraire automatiquement des lettres, puis des mots, à partir d'images contenant une grille de caractères. Cette chaîne permet de transformer une image brute en un ensemble de données visuelles structurées, prêtes à être exploitées par un système d'analyse ou un modèle de reconnaissance automatique. L'ensemble du processus s'inscrit dans une démarche modulaire afin de garantir robustesse, clarté et extensibilité du programme.

Chargement et préparation des images

La première étape fondamentale consiste à charger chaque image tout en la convertissant en niveaux de gris. Ce choix méthodologique n'est pas anodin : la représentation en monochrome réduit considérablement la complexité des traitements ultérieurs et met en évidence les contrastes entre les caractères (généralement sombres) et le fond (plus clair). Cette simplification permet d'améliorer la fiabilité des étapes de détection sans altérer l'information pertinente.

Une fois l'image convertie, le programme parcourt l'ensemble des pixels pour identifier les zones présentant des caractéristiques potentiellement textuelles. Cette base fournit un support solide aux traitements de segmentation.

Détection des composantes connexes

Afin d'isoler les régions contenant des lettres ou fragments de lettres, nous utilisons un algorithme de *recherche de composantes connexes*. Cette approche, couramment employée en traitement d'images, permet de regrouper en une même entité tous les pixels adjacents dont l'intensité indique qu'ils appartiennent à un élément significatif (par opposition au fond clair).

Pour chaque composante détectée, le programme calcule une **boîte englobante** (*bounding box*), définie par sa hauteur, sa largeur et sa position dans l'image. Cette boîte constitue une approximation fidèle de la zone occupée par l'objet visuel correspondant.

Filtrage des composantes non pertinentes

L'extraction brute contient inévitablement du bruit : artefacts, petits points isolés, irrégularités ou traits indésirables. Pour éliminer ces éléments perturbateurs, nous appliquons une stratégie statistique reposant sur les dimensions des boîtes détectées.

À partir de l'ensemble des composantes, le programme calcule les valeurs médianes de plusieurs indicateurs :

— hauteur,
— largeur,
— surface.

Ces valeurs servent de référence pour distinguer les objets pertinents (lettres, fragments significatifs) des bruits visuels. Toute composante dont les dimensions s'écartent trop des valeurs attendues est automatiquement écartée. Cette étape s'avère cruciale pour éviter que des éléments parasites ne perturbent la reconstruction des mots.

Organisation spatiale et segmentation en mots

Après filtrage, l'ensemble des boîtes valides doit être organisé pour reconstruire fidèlement le texte présent dans l'image. La première étape consiste en un **tri vertical**, permettant de regrouper correctement les caractères selon les lignes de texte. Une fois les lignes identifiées, un **tri horizontal** est appliqué à l'intérieur de chacune d'elles pour restituer l'ordre naturel de lecture.

La segmentation en mots repose ensuite sur l'analyse des espacements horizontaux entre boîtes consécutives. Lorsque la distance dépasse un seuil déterminé empiriquement ou statistiquement, une séparation de mots est identifiée. Ce procédé permet de reconstituer la structure textuelle d'origine de manière efficace et cohérente.

Extraction, recadrage et normalisation des lettres

Chaque mot détecté fait ensuite l'objet d'un traitement individuel. Les lettres qui le composent sont extraites une à une à partir de leurs boîtes englobantes. Le programme effectue alors un recadrage précis visant à isoler uniquement la zone utile du caractère.

Pour garantir une exploitation optimale par les algorithmes de reconnaissance ou les réseaux de neurones, les images de lettres sont soumises à une **phase de normalisation**, incluant :

— le recentrage du caractère dans l'image,
— l'ajout éventuel de marges pour uniformiser la présentation,
— le redimensionnement vers un format standard.

Cette normalisation assure l'homogénéité du jeu de données produit et facilite les étapes d'apprentissage ou de classification futures.

Organisation des résultats

Les résultats sont sauvegardés dans une arborescence claire, pensée pour une utilisation automatisée. Chaque mot détecté est associé à un dossier dédié contenant :

— les images PNG de chaque lettre,

— une numérotation correspondant à leur ordre de lecture.

De la même manière, les lettres extraites directement de la grille initiale sont stockées avec une convention de nommage indiquant leur position dans la matrice. Cette organisation garantit une traçabilité exemplaire et simplifie les traitements ultérieurs.

Architecture logicielle modulaire

Un point clé de ce projet réside dans la séparation nette entre les différents modules du programme. Chaque étape du traitement — détection des lignes, segmentation, extraction, filtrage, normalisation — repose sur une fonction spécialisée. Cette architecture modulaire procure plusieurs avantages majeurs :

— amélioration de la lisibilité du code,

— maintenance facilitée,

— possibilité de remplacer ou d'améliorer une étape sans modifier le reste du système,

— extensibilité pour ajouter de nouvelles fonctionnalités.

Cette approche garantit la pérennité du projet ainsi que sa capacité à évoluer en fonction des besoins.

Conclusion

En résumé, le pipeline mis en place permet une extraction automatisée, structurée et fiable d'éléments textuels à partir d'images de grilles. Le traitement combinant analyse en niveaux de gris, détection de composantes connexes, filtrage statistique, segmentation intelligente et normalisation aboutit à un système robuste, modulaire et adaptable. Cette base solide constitue un atout essentiel pour les développements futurs, notamment l'intégration d'algorithmes de reconnaissance ou l'automatisation complète de la résolution de grilles.

5.2.3 Bashir

Rappel des tâches :

Mon objectif dans cette partie était de mettre en place le **réseau neuronal chargé de reconnaître les lettres** à partir des images binarisées, puis de l'intégrer au reste de la chaîne OCR. Ce réseau ne s'occupe pas de l'apprentissage (qui a été fait en amont), mais de la **prédiction** : il charge les poids depuis un fichier, prépare les images, et retourne la lettre la plus probable pour chaque image.

Structure du modèle en C :

Le réseau neuronal est représenté par une structure `NNOCRModel` qui regroupe toutes les informations nécessaires au modèle :

- `input_dim` : taille du vecteur d'entrée (nombre de pixels de la tuile normalisée) ;
- `hidden_dim` : nombre de neurones dans la couche cachée ;
- `output_dim` : nombre de classes (ici 26 lettres de A à Z) ;
- `W1`, `b1` : poids et biais de la couche cachée ;
- `W2`, `b2` : poids et biais de la couche de sortie.

Un pointeur global `g_model` contient l'instance du réseau, et les dimensions des tuiles (`g_tile_w`, `g_tile_h`) sont déduites automatiquement à partir de `input_dim`.

Les poids sont chargés depuis un fichier texte grâce à la fonction `load_weights()`, appelée via `nn_init()`. Cette fonction lit d'abord l'en-tête :

```
input_dim hidden_dim output_dim
```

puis charge successivement `W1`, `b1`, `W2` et `b2` dans des tableaux dynamiques.

Capture d'écran :

- Une capture du bloc de code qui définit la structure `NNOCRModel` et la fonction `load_weights()` (vue globale du modèle et du chargement des poids).

```c
static int load_weights(const char *weights_path) {
    FILE *f = fopen(weights_path, "r");
    if (!f) {
        fprintf(stderr, "Cannot open weights: %s\n", weights_path);
        return 0;
    }

    if (fscanf(f, "%d %d %d", &g_model.input_dim, &g_model.hidden_dim, &g_model.output_dim) != 3) {
        fprintf(stderr, "Invalid weights header\n");
        fclose(f);
        return 0;
    }

    size_t w1_sz = (size_t)g_model.input_dim * (size_t)g_model.hidden_dim;
    size_t w2_sz = (size_t)g_model.hidden_dim * (size_t)g_model.output_dim;
    g_model.W1 = (float *)malloc(sizeof(float) * w1_sz);
    g_model.b1 = (float *)malloc(sizeof(float) * (size_t)g_model.hidden_dim);
    g_model.W2 = (float *)malloc(sizeof(float) * w2_sz);
    g_model.b2 = (float *)malloc(sizeof(float) * (size_t)g_model.output_dim);
```

FIGURE 11 – Structure du modèle et chargement des poids en C.

Prétraitement des images de lettres :

Avant de passer une image au réseau, je dois la transformer en un **vecteur de flottants** de taille fixe. C'est le rôle de la fonction `load_image_vector()` :

1. Chargement du fichier PNG en niveaux de gris avec `stb_image` ;
2. Recherche de la **boîte englobante** des pixels « sombres » (la lettre) :
   — on parcourt l'image pour repérer les pixels dont la valeur est inférieure à un seuil (par exemple 200) ;
   — on détermine le rectangle minimal contenant la lettre (x_0, y_0, x_1, y_1).
3. Redimensionnement de cette boîte englobante pour qu'elle tienne dans une tuile de taille `g_tile_w` × `g_tile_h`, avec une petite marge ;
4. Centrage de la lettre dans la tuile et remplissage du reste avec la valeur de fond (1.0 pour le blanc) ;
5. Inversion éventuelle si l'image est majoritairement sombre (gestion automatique du cas « lettre blanche sur fond noir »).

Le résultat est un vecteur $\mathbf{x} \in R^{\texttt{input\_dim}}$ où chaque case représente un pixel normalisé entre 0 et 1 (0 = lettre, 1 = fond).

Captures d'écran intéressantes :

— Le code de `load_image_vector()`, montrant la détection de la boîte englobante et le centrage de la lettre.

Propagation avant (inférence) :

Une fois le vecteur d'entrée construit, la fonction `predict_letter_from_vec()` applique une **propagation avant** classique d'un perceptron multicouche (MLP) à une couche cachée. Les calculs sont les suivants :

$$z_1 = W_1 \cdot x + b_1$$
$$a_1 = \sigma(z_1)$$
$$z_2 = W_2 \cdot a_1 + b_2$$
$$\hat{y} = \sigma(z_2)$$

où :

— x est le vecteur d'entrée (image de lettre vectorisée) ;
— W_1, b_1 sont les paramètres de la couche cachée ;
— W_2, b_2 sont les paramètres de la couche de sortie ;
— σ est la fonction sigmoïde appliquée à chaque neurone.

La fonction sigmoïde est définie par :

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

Pour chaque neurone de sortie, on obtient une « probabilité » associée à une lettre. Le code recherche ensuite l'indice k correspondant à la plus grande valeur $\hat{y}_k$, puis le convertit en caractère :

$$\texttt{caractère} = \texttt{'A'} + k$$

Si l'indice sort de la plage $[0, 25]$, le programme renvoie simplement `'?'` pour indiquer une lettre inconnue.

Intégration dans la reconnaissance de la grille :

Le réseau n'est pas utilisé seul : il est intégré dans une **pipeline OCR complète** qui parcourt des dossiers d'images déjà découpées. La fonction `nn_process_grid()` suit les étapes suivantes :

1. Recherche du dossier contenant les lettres de la grille (`find_grid_directory()`) ;
2. Scan de toutes les images de lettres (`scan_letter_images()`) en récupérant leurs coordonnées (ligne, colonne) à partir du nom de fichier (`parse_letter_indices()`) ;
3. Pour chaque case :
   — chargement de l'image de lettre ;
   — vectorisation via `load_image_vector()` ;
   — prédiction avec `predict_letter_from_vec()` ;
   — stockage du caractère dans une matrice `grid[row][col]`.
4. Écriture de la grille finale dans un fichier texte `grille.txt` ;
5. Reconnaissance des mots dans des sous-dossiers numérotés (`collect_word_dirs()` puis `recognize_word_from_dir()`), et écriture dans `mots.txt`.

Le programme affiche enfin un résumé en console, par exemple :

```
Processed 256 images into 16 x 16 grid -> grille.txt / mots.txt
```

Captures d'écran :

— Vue de la fonction `nn_process_grid()` avec les différentes étapes (scan, prédiction, écriture de fichiers).
— Structure des dossiers sur le disque : `grid_letters/`, `weights.txt`, `grille.txt`, `mots.txt`.
— Sortie terminal montrant le message de fin avec le nombre d'images traitées et la taille de la grille.

FIGURE 12 – Chaîne complète : de la lettre découpée jusqu'aux fichiers `grille.txt` et `mots.txt`.

Difficultés rencontrées :

Plusieurs points ont demandé une attention particulière :

— **Gestion de la mémoire :** toutes les allocations (`malloc`/`realloc`) pour les poids, les vecteurs d'entrée, les images et la grille de caractères doivent être libérées correctement (`free_model()`, `free_words()`, etc.) pour éviter les fuites de mémoire.

— **Prétraitement robuste :** il a fallu gérer les cas où l'image contient très peu de pixels sombres, ou où la lettre est collée au bord. Pour cela, j'ai ajouté des valeurs par défaut pour la boîte englobante et des vérifications supplémentaires.

— **Convention de nommage des fichiers :** toute la logique de reconstruction de la grille repose sur des noms de fichiers cohérents (`row_col.png`, `x%d_y%d.png`, indices numériques pour les mots). Une petite erreur de nommage peut faire disparaître des lettres lors du scan.

— **Correspondance entre dimensions :** `input_dim` doit toujours être compatible avec `g_tile_w` et `g_tile_h`. J'ai donc ajouté une fonction `infer_tile_dims()` qui calcule automatiquement la largeur et la hauteur à partir de l'entrée.

Conclusion sur le réseau OCR :

Grâce à ce module, le projet dispose d'un **réseau neuronal entièrement fonctionnel en C** pour la reconnaissance de lettres. Le modèle charge ses poids depuis un simple fichier texte, prétraite les images, reconstruit une grille complète et écrit les mots reconnus dans des fichiers texte. Cette partie m'a permis de consolider :

— la compréhension de la propagation avant d'un MLP,
— la gestion fine de la mémoire en C,
— l'intégration d'un réseau neuronal dans un pipeline OCR complet, sans dépendances lourdes.

6 Résultat Final

Le résultat final du projet est complet, nous sommes certes un peu déçu de la manière dont la surchage de travail s'est accumulé au fil du temps, en effet ce projet est très long à réaliser, pose également un bon nombre de contraintes et de problèmes au fil de sa réalisation.

Positivement, nous rendons ce projet avec le sourire.

7 Conclusion

En conclusion de ce projet, nous avons appris énormément de choses. Les difficultées que nous avons rencontrés ont été très nombreuses, nous avons également appris réellement à travailler sous pression d'une dead-line, en effet le travail fut intensif surtout au niveau de la fin du temps, en raison d'une surcharge de travail externe.

Nous ressortons de ce projet plus grands et avec un avis positif, et au fond nous sommes également fier de présenter un produit (presque) terminé à 100 pourcent le jour du rendu.