Reconnaissance Optique de Caractères (ROC)

Rapport de Soutenance n°1

Auteurs :

Édouard Baudouïn
Adem Çelik
Bashir Toufir

EPITA

Réalisé le : 29 octobre 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 cette 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.

Ce premier rapport de soutenance présente le cadre et les objectifs du projet, la répartition des tâches, ainsi que l'état d'avancement sur chaque composant. Nous détaillons enfin les difficultés rencontrées, les pistes d'amélioration identifiées et les jalons prévus pour la suite.

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

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%	100%
Détection des mots et découpage des lettres	100%	25%
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.

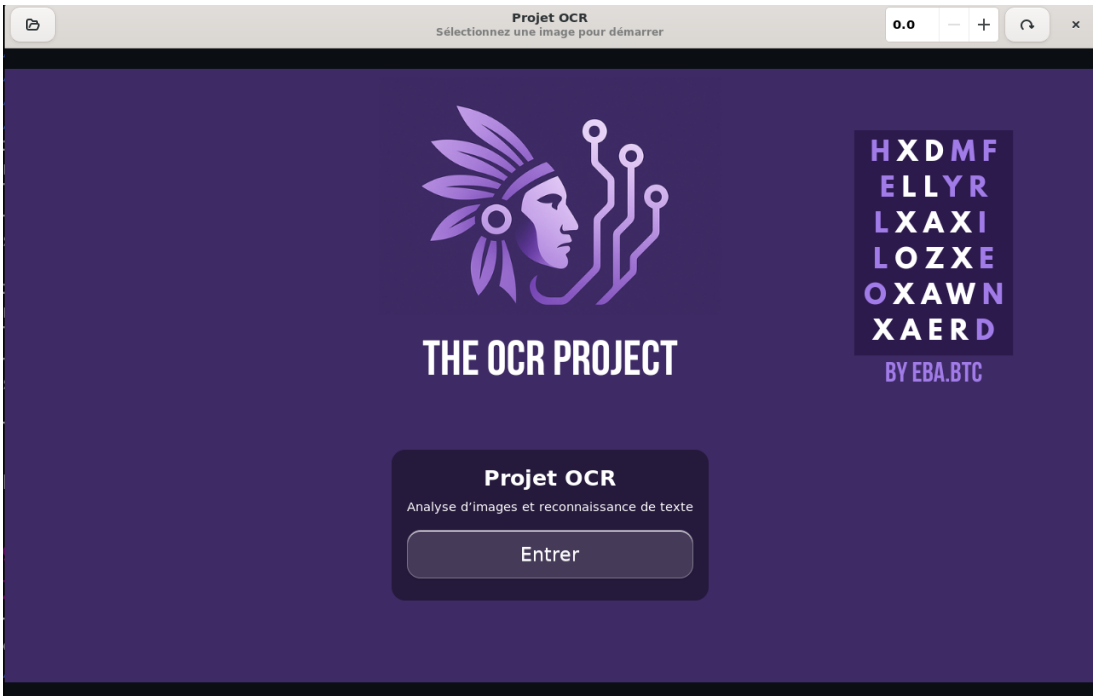

FIGURE 2 – Aperçu de l'interface du projet

4.2.2 Adem

Rappel des tâches :

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

Le programme que j'ai développé devait être capable de détecter les lignes horizontales et verticales d'une grille, puis d'extraire chaque cellule sous forme d'image séparée. Le plus gros défi a été de faire en sorte que le traitement fonctionne même avec des images imparfaites, floues ou légèrement inclinées.

Structure du projet :

J'ai organisé le projet de façon modulaire, avec un fichier dédié à chaque partie du traitement. Cette structure m'a permis de garder un code clair et facile à maintenir, sans tout mettre dans une seule grosse fonction. Le point d'entrée se trouve dans `main.c` : il charge l'image grâce à la bibliothèque STB Image, puis appelle la fonction `"splitgrid"`, qui gère toute la logique du programme.

Détection des bandes :

Dans `grid_bands.c`, j'ai écrit le code pour repérer les lignes et colonnes de la grille. L'idée est simple : je parcours chaque ligne et chaque colonne de l'image et je compte le nombre de pixels sombres. Si une ligne contient beaucoup de pixels noirs, c'est probablement un trait de la grille.

Au départ, j'utilisais un seuil fixe pour décider si une ligne était une bande ou non, mais ça ne marchait pas bien sur les images trop claires ou mal contrastées. J'ai donc ajouté un seuil adaptatif, basé sur la moyenne et le maximum des valeurs détectées. Ainsi, le programme s'ajuste automatiquement à la luminosité de chaque image. Cette partie m'a donné pas mal de fil à retordre : sur certaines images, les lignes étaient à peine visibles. J'ai donc mis en place un système de recalcul automatique du seuil, ce qui a rendu la détection beaucoup plus robuste.

Extraction des lettres :

Une fois les lignes horizontales et verticales trouvées, le fichier `grid_cells.c` s'occupe d'extraire les cellules. Chaque paire de bandes délimite une case : j'extrais donc cette portion d'image, je coupe légèrement les bords pour éviter les traits noirs, puis je sauvegarde le résultat au format PNG.

Les fichiers sont nommés selon leur position dans la grille (par exemple : 0,0.png, 1,0.png, etc.). J'ai aussi ajouté un contrôle de cohérence : si une cellule est beaucoup plus grande ou plus petite que les autres, c'est probablement une erreur de détection. Dans ce cas, le programme passe automatiquement à une méthode alternative.

Méthode alternative : détection par composants

Quand la détection de lignes échoue, je fais appel à `grid_components.c`. Cette méthode ne cherche plus les traits, mais les zones sombres de l'image (lettres, morceaux de traits, etc.). Pour cela, j'utilise une méthode de flood-fill qui regroupe les pixels connectés. Chaque zone détectée devient un "composant". Ensuite, je trie ces composants et je les regroupe en lignes et colonnes selon leur position.

À partir de ces regroupements, je calcule des médianes pour estimer les bords de chaque cellule (haut, bas, gauche, droite). Cela permet de reconstruire une grille approximative, même quand les lignes ont complètement disparu.

C'est sans doute la partie la plus complexe du projet, mais aussi la plus intéressante : elle permet au programme de fonctionner sur des images où il n'y a quasiment plus de traits visibles.

Fonctions utilitaires :

Pour rendre le code plus propre et réutilisable, j'ai ajouté plusieurs fichiers d'utilitaires :

`grid_fs.c` gère la création et le nettoyage des dossiers de sortie ;

`grid_image.c` s'occupe de l'écriture des images via STB Image Write. Ces fichiers permettent de centraliser les tâches répétitives et de simplifier le reste du code.

Difficultés rencontrées :

Le projet n'a pas été simple. D'abord, la qualité des images variait beaucoup : certaines étaient sombres, d'autres très claires, ce qui rendait la détection difficile. J'ai dû concevoir un système de seuil dynamique pour que le programme s'adapte à chaque cas. Ensuite, la gestion de la mémoire m'a posé des soucis, surtout pendant le flood-fill, où il faut manipuler de très grands tableaux. J'ai dû être vigilant sur les allocations et libérations de mémoire pour éviter les fuites.

Mais le défi le plus intéressant a été la reconstruction de grilles sans traits visibles. J'ai dû imaginer une logique capable de deviner la structure d'une grille uniquement à partir de la position des lettres. Ça m'a demandé pas mal de tests et de réflexion, mais au final, j'ai beaucoup appris sur la manière de décomposer un problème complexe en plusieurs étapes claires et logiques.

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 3 – Structures Params et Grams.

Fonction Forward :

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

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

```
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 4 – 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 5 – 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.

```c
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 6 – 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 7 – 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.

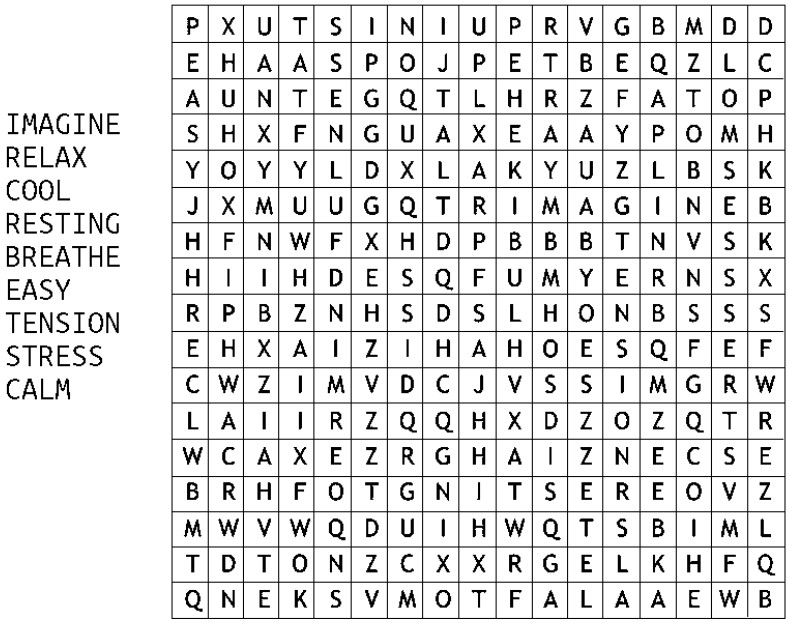

FIGURE 8 – Sortie après la binarisation de l'image "Easy"

5 Après la soutenance

Après cette soutenance, beaucoup de tâches sont à finaliser.

Déjà, nous allons devoir reprendre l'architecture du projet et lier tout nos scripts, pour l'instant ils sont tous séparés pour être présentés simplement durant la première soutenance.

Il faudra compléter les tâches restantes sur le tableau présenté au début de ce rapport également.

6 Conclusion

En conclusion, ce premier rendu nous a permis de faire un point sur notre organisation, sur les tâches de chacun et le travail restant.
Nous pensons être sur la bonne voie pour mener le projet à bien, et nous allons travailler dès l'achèvement de cette première soutenance sur la suite pour le réaliser dans les temps.