PROJET OCR - RAPPORT DE SOUTENANCE 1

Tom HOUSSIN

Luca MACHEDA

Wassim ALOUINI

Sommaire

Objectifs du projet

L'objectif de ce projet est de concevoir un programme capable de résoudre automatiquement une grille de mots cachés à partir d'une image. L'idée est de partir d'une simple photo contenant une grille et une liste de mots, et de créer un système capable de traiter cette image, d'en extraire les informations nécessaires, de reconnaître les lettres, puis de trouver la position exacte des mots dans la grille grâce à un algorithme de recherche.

Pour y arriver, plusieurs grandes étapes ont été mises en place. Tout d'abord, le programme procède au chargement et à la conversion de l'image, accompagnés d'une suppression des couleurs afin de simplifier le traitement. Ensuite, une rotation manuelle de l'image est prévue pour permettre un bon alignement de la grille si celle-ci est inclinée. Le système procède ensuite à la détection automatique des différentes zones de l'image : la grille, la liste de mots, ainsi que les lettres présentes dans chacune d'elles. Une fois ces zones identifiées, l'image est découpée afin d'isoler chaque lettre et de la sauvegarder individuellement.

Une autre étape importante consiste à implémenter un algorithme de résolution, dont le rôle est de parcourir la grille et de localiser les mots à trouver. Enfin, un mini réseau de neurones a été créé, servant de démonstration pour la partie apprentissage automatique.

Répartition et organisation du travail

Dès le début du projet, l'équipe a choisi de répartir les tâches selon les compétences et les envies de chacun afin d'avancer efficacement et en parallèle. Une personne s'est consacrée au développement du solver, c'est-à-dire la partie algorithmique du projet, dont l'objectif était de concevoir et de coder un programme capable de parcourir la grille dans toutes les directions possibles pour repérer les mots à trouver.

Une autre personne a pris en charge la partie réseau de neurones, en se concentrant sur la création d'un petit modèle capable d'apprendre la fonction logique A.B + A.B. Ce travail servira par la suite de base pour la reconnaissance automatique des lettres. Enfin, une troisième personne s'est occupée du traitement d'image, en gérant le chargement des fichiers, la suppression des couleurs, la rotation manuelle, la détection des zones importantes (grille, mots, lettres) et le découpage des caractères.

Cette organisation du travail a permis à l'équipe d'avancer de manière structurée et efficace, sans interférer sur les tâches des autres membres. Les différentes parties du projet ont été développées séparément mais dans le but de pouvoir être intégrées ensemble à la suite, assurant ainsi une cohérence globale du système.

Luca Macheda:

Pour ce premier rendu, mon rôle a principalement été de réfléchir et de concevoir l'algorithme de résolution de la grille de mots cachés. J'ai travaillé sur la partie du programme qui permet de rechercher un mot dans une grille de lettres, en testant différentes approches pour rendre cette recherche efficace. J'ai d'abord participé à la mise en place d'un algorithme simple qui parcourt la grille dans toutes les directions possibles afin de vérifier si le mot recherché y est présent. Ensuite, j'ai commencé à explorer des pistes d'optimisation, notamment avec une méthode utilisant une lettre pivot pour limiter le nombre de comparaisons nécessaires. Cette partie a demandé pas mal de réflexion et de tests pour comprendre les cas particuliers et améliorer la fiabilité du code. En parallèle, j'ai aussi participé à la mise en place du fonctionnement général du programme, notamment la lecture du fichier contenant la grille et l'organisation des données pour qu'elles puissent être traitées correctement par l'algorithme.

Tom Houssin :

J'ai développé un réseau de neurones capable d'apprendre la fonction XNOR. Au départ, j'ai consacré beaucoup de temps à comprendre la descente de gradient, un concept mathématique essentiel pour ajuster les poids et les biais du réseau afin de minimiser l'erreur de prédiction. La notion de dérivées partielles composées et de rétropropagation m'a particulièrement posé des difficultés, notamment pour saisir comment l'erreur se propage d'une couche à l'autre. Une fois ces principes compris, j'ai pu passer à l'implémentation concrète du réseau en langage C. J'ai conçu une architecture simple avec deux neurones en entrée, deux neurones cachés et un neurone de sortie utilisant la fonction sigmoïde comme activation. L'un des défis majeurs a été la gestion manuelle de la mémoire, en allouant dynamiquement les matrices de poids et les vecteurs de biais à l'aide de structures. Pour rendre les tests reproductibles, j'ai initialisé les paramètres avec une fonction sinus. L'implémentation de la descente de gradient stochastique (SGD) a nécessité de calculer précisément les gradients à chaque étape, puis de mettre à jour les poids et les biais en conséquence. J'ai ensuite testé le réseau en ajustant les hyperparamètres : le nombre de neurones cachés, le learning rate et le nombre d'epochs. Après plusieurs essais, j'ai constaté qu'un réseau avec deux neurones cachés, un learning rate de 0,5 et environ 8000 itérations offrait la meilleure convergence. Ce projet m'a permis de mieux comprendre le fonctionnement interne d'un réseau de neurones, de consolider mes bases en mathématiques appliquées et de développer une solide maîtrise de la gestion mémoire et de la programmation bas niveau en C.

Wassim Alouini :

Dans mon travail, j'ai développé une application complète de traitement d'image en utilisant la bibliothèque SDL2, dans le but de poser les bases d'un futur moteur OCR. J'ai conçu un programme capable de charger, afficher, transformer et analyser des images à travers une interface graphique interactive. L'application repose sur une architecture modulaire : un module principal pour la boucle d'événements et les commandes, et un module de segmentation dédié à la détection de zones cohérentes de pixels (blobs) et à leur encadrement par des boîtes englobantes. J'ai implémenté plusieurs opérations clés : la conversion en niveaux de gris, la binarisation selon un seuil ajustable, la rotation d'image, et la détection automatique de zones sombres. Ces traitements sont commandés en temps réel via une fenêtre de console intégrée, permettant d'observer instantanément les effets à l'écran. J'ai également implémenté un algorithme de flood fill récursif pour détecter les blobs et calculer leurs boîtes englobantes, première étape vers la segmentation des caractères. Les résultats obtenus montrent que le système peut transformer et analyser des images efficacement, en produisant des visualisations claires des zones détectées. Malgré quelques limites notamment certaines incompatibilités lors des rotations d'image , le prototype démontre la faisabilité d'un pipeline de prétraitement complet. Ce projet m'a permis de maîtriser la gestion d'images sous SDL2, la logique événementielle, les traitements d'image fondamentaux, ainsi que la conception d'un code modulaire et extensible. Il constitue une base solide pour intégrer, à terme, des fonctionnalités avancées comme le seuillage adaptatif, le redressement automatique ou encore la classification OCR.

État d’avancement attendu et réel

Pour cette première soutenance, toutes les fonctionnalités demandées ont été réalisées. Le programme est capable de charger une image et d’en supprimer les couleurs afin d’en faciliter le traitement. La rotation manuelle fonctionne correctement et permet d’aligner la grille lorsque celle-ci n’est pas droite. La détection des zones, comprenant la grille, les mots et les lettres, a été mise en place et donne déjà des résultats satisfaisants.

Le découpage de l’image fonctionne lui aussi : chaque lettre peut être extraite et sauvegardée séparément pour être utilisée ultérieurement dans le processus de reconnaissance. Le solveur est opérationnel et peut lire une grille depuis un fichier texte tout en trouvant les mots dans toutes les directions possibles. Enfin, le réseau de neurones fonctionne correctement sur un cas simple, celui de la fonction logique A.B + A.B, démontrant ainsi le bon fonctionnement du modèle et la compréhension du concept.

En résumé, tout ce qui devait être terminé pour cette première étape l’a été. Le projet est désormais bien lancé et prêt à aborder la phase suivante, au cours de laquelle toutes les parties développées séparément seront reliées entre elles afin de former un système complet et fonctionnel.

Conclusion de cette étape

Cette première phase nous a permis de poser les bases du projet et de bien répartir les rôles au sein du groupe. Chacun a pu se concentrer sur une partie spécifique : traitement d’image, réseau de neurones ou algorithmique. Ces trois aspects devront être combinés pour que le programme final soit entièrement autonome.

L’état actuel du projet correspond à ce qui était attendu pour cette soutenance. Pour la suite, l’objectif sera d’intégrer tous les modules ensemble et d’améliorer le réseau de neurones afin qu’il puisse reconnaître les lettres extraites directement à partir des images.

Traitement d’image complet : Wassim Alouini

L’objectif de cette preuve de concept est d’implémenter un ensemble d’opérations graphiques sur des images chargement, affichage, rotation, conversion en niveaux de gris, binarisation, et détection de zones en utilisant la bibliothèque SDL2. Ces traitements visuels préparent les images à l’étape suivante du pipeline OCR, où les algorithmes de segmentation et de classification pourront être appliqués.

Le code repose principalement sur deux modules: main.c, qui gère l’initialisation, l’affichage et les commandes interactives de traitement d’image. bounds.c, qui met en œuvre les fonctions d’analyse de blobs et de délimitation de zones (bounding boxes). L’ensemble forme une application graphique interactive capable de manipuler des images et d’en extraire les structures élémentaires (zones sombres, blocs de texte, etc.) dans un environnement SDL.

Le POC a été conçu pour atteindre plusieurs objectifs techniques et fonctionnels : Vérifier la compatibilité du flux graphique SDL avec le format et les opérations d’image nécessaires à un futur moteur OCR. Tester les transformations géométriques (rotation, redimensionnement, translation). Implémenter des traitements d’image classiques : niveaux de gris, binarisation et détection de blobs. Mettre en place une interface de commande simple via une fenêtre de console intégrée (cmd_window). Évaluer la robustesse du code dans le cadre d’une boucle d’événements SDL complète.

L’application suit une architecture modulaire reposant sur plusieurs composants bien séparés : Module principal (main.c) : responsable de la boucle de rendu, de la gestion des événements, et du déclenchement des opérations de traitement d’image. Module de segmentation (bounds.c) : chargé d’identifier les zones cohérentes de pixels (blobs) et d’en extraire les boîtes englobantes (bounding boxes). Modules auxiliaires : image_loader.h, window_manager.h et cmd_window.h (non détaillés ici) assurent respectivement le chargement des images, la création des fenêtres SDL et la gestion des entrées utilisateurs. Cette modularité permet une évolution fluide vers un projet plus complet, où d’autres traitements (filtrage, détection de lignes, OCR neuronal) pourront être ajoutés.

Pour ce qui est du chargement et initialisation, l'exécution débute par la conversion d’une image d’entrée (input.png) en un format compatible BMP (output.bmp) à l’aide d’un appel au logiciel externe ImageMagick. Une fenêtre SDL est ensuite initialisée, avec un rendu matériel (renderer) et une surface principale (master_surface) qui contient les pixels manipulables. Puis, l’image est chargée dans cette surface et affichée via une texture SDL et la taille de la fenêtre est automatiquement ajustée en fonction de la rotation potentielle de l’image, de façon à toujours contenir la totalité de la texture après transformation.

Boucle d’événements et interface utilisateur : L’application repose sur une boucle principale qui intercepte les événements SDL. L’utilisateur peut saisir différentes commandes via la fenêtre de commande intégrée (cmd_window), chacune correspondant à une opération de traitement d’image : rotate [angle] : fait pivoter l’image selon un angle donné. grayscale : convertit l’image en niveaux de gris. binarize [seuil] : transforme l’image en noir et blanc

selon un seuil de luminosité. boxes : déclenche la détection et le tracé de boîtes englobantes autour des régions sombres (blobs). Chaque opération est immédiatement visualisée dans la fenêtre SDL, permettant à l'utilisateur de suivre les transformations en temps réel.

Opérations d'image: La fonction apply_grayscale() parcourt chaque pixel de l'image et calcule une valeur moyenne pondérée selon la formule standard de luminance : gray = 0.299R + 0.587G + 0.114B. Ce traitement prépare l'image à la binarisation en supprimant la couleur et en conservant uniquement l'intensité lumineuse. La fonction binarize() transforme chaque pixel en noir ou blanc selon un seuil fixé. Cette étape est essentielle pour les systèmes OCR, qui nécessitent une séparation nette entre le texte (noir) et le fond (blanc). Ici, un seuil trop bas produit une image très sombre, tandis qu'un seuil trop haut peut effacer les détails. Ce paramètre est donc ajustable dynamiquement par commande. La fonction rotate_and_render() calcule les nouvelles dimensions de la fenêtre en fonction de l'angle choisi et applique une transformation géométrique à la texture. Le code gère le recalcul du bounding box maximal pour éviter toute découpe de l'image. La surface principale est ensuite actualisée grâce à apply_rotation_to_surface().

Détection de blobs et encadrement : La commande boxes appelle le cœur du module bounds.c, dont le rôle est d'analyser la surface pixel par pixel pour identifier des zones connectées de pixels noirs (blobs). Cette détection est réalisée par un algorithme de flood fill récursif, inspiré des méthodes de segmentation d'images binaires. Chaque blob détecté donne lieu à la création d'une structure Box représentant sa boîte englobante minimale. Ces boîtes sont ensuite tracées en rouge sur l'image grâce à la fonction draw_boxes(). Cette fonctionnalité constitue une première approche de la segmentation des caractères dans le cadre d'un OCR.

Analyse algorithmique : La fonction flood_fill() implémente un parcours récursif des pixels adjacents. Lorsqu'un pixel noir est trouvé, il est marqué comme visité et ajouté au blob courant, puis la recherche se propage à ses voisins immédiats. Ce processus permet d'extraire de manière robuste toutes les zones de texte ou de dessin présentes sur une image binarisée. Pour chaque blob identifié, compute_blob_boxes() calcule le rectangle minimal couvrant l'ensemble de ses coordonnées. Ces boîtes servent à repérer la position et la taille des caractères, mais aussi à distinguer différentes zones (titres, colonnes, tableaux). Le code introduit également des fonctions d'analyse plus avancées, telles que differentiate_grid_list_and_words() : qui sépare les zones en grilles, listes et lignes de texte selon leur distribution spatiale. sort_boxes() et boxcmp() : qui ordonnent les boîtes selon leurs coordonnées, facilitant la reconstruction du texte. Ces traitements préparent les données pour une phase ultérieure de classification OCR.

Résultats observés : Le POC démontre avec succès la faisabilité d'un prétraitement complet d'images dans un environnement SDL. L'interface graphique permet de visualiser chaque étape. Les fonctions de rotation et de binarisation sont stables et précises. La détection de blobs par flood fill donne des résultats cohérents sur des images simples en noir et blanc. Les boîtes englobantes sont correctement tracées, offrant une première segmentation visuelle des

caractères. Ces résultats constituent une base solide pour le développement d'un module OCR complet basé sur ces fondations.

Perspectives d'évolution : Les prochaines étapes du projet consisteront à : Intégrer un filtrage adaptatif (par exemple le seuillage d'Otsu ou le filtrage gaussien). Ajouter un redressement automatique des images par détection de lignes de texte. Étendre l'interface graphique pour inclure un système de logs et de comparaison avant/après. Optimiser le traitement des images en parallèle (multi-threading via SDL2 ou OpenMP).

Conclusion :Ce POC constitue une étape clé dans la conception d'un système OCR performant. Il démontre la capacité à manipuler, transformer et analyser des images au sein d'une interface graphique interactive basée sur SDL2. L'architecture modulaire, la clarté du code et la pertinence des traitements mis en œuvre assurent une base robuste pour la suite du développement.

Nota Bene : Certaines parties du code sont encore en cours de développement et peuvent présenter des comportements instables. En particulier, la rotation de l'image entraîne actuellement des incompatibilités avec certaines opérations ultérieures telles que la détection de zones (boxes), provoquant des erreurs de segmentation ou des incohérences de rendu. Ces limitations sont connues et identifiées : elles seront corrigées dans les itérations suivantes du projet, notamment par une refonte du pipeline de gestion de surfaces après rotation et un meilleur contrôle des formats de texture.

Développement d’un réseau de neurones pour la fonction XNOR :

Tom HOUSSIN

Compréhension et documentation mathématique de la descente de gradient :

Au début de ce projet, l’un des aspects les plus complexes à appréhender a été la descente de gradient. Cette méthode est essentielle pour entraîner un réseau de neurones, car elle permet de modifier les poids et les biais progressivement afin de réduire l’erreur de prédiction. L’aspect le plus difficile à saisir initialement était le principe de la chaîne de dérivées partielles composées. Notamment pour les couches cachées dans un réseau à plusieurs couches ou l’erreur à la sortie influence la correction des poids dans la couche cachée. La notion de dérive composée a été source de confusion, car il faut propager l’erreur en tenant compte de la dépendance des couches. Une fois comprise, elle permet de faire tendre la fonction de coût vers zéro en modifiant progressivement tous les paramètres du réseau.

Implémentation en C :

Après avoir compris les fondements mathématiques, l’étape suivante a été de définir l’architecture du réseau pour apprendre la fonction XNOR. Le réseau implémenté comporte 2 neurones en entrée, correspondant aux deux variables de la fonction XNOR, 2 neurones dans la couche cachée et 1 neurone de sortie, qui produit l’output final. La fonction d’activation choisie a été la fonction sigmoïde. La mise en œuvre du réseau en C a été une difficulté car contrairement au python qui est généralement utilisé pour ce genre de code qui gère l’allocation mémoire pour nous. Il a fallu gérer et allouer dynamiquement la mémoire pour les matrices des poids et les vecteurs de biais.

Chaque poids et biais est stocké dans une structure dynamique. Des matrices pour les poids w_0_1 pour les poids entre la couche 0 (input) et la couche 1 (hidden). w_1_2 pour les poids entre la couche 1 et la couche 2 (output). Des tableaux pour les biais b_1 et b_2. Toutes ces allocations sont centralisées dans un structure NeuralNetwork qui représente le réseau de neurones avec des malloc. L’initialisation des poids et des biais utilise la fonction sinus pour générer des valeurs dans l’intervalle $[-1,1]$. Chaque poids ou biais est calculé en fonction de ses indices.Chaque paramètre reçoit donc une valeur différente. Cette méthode simplifie le débogage car les résultats sont déterministes.

La complexité principale de l'implémentation de ce réseau réside dans le calcul correct des gradients et dans la mise à jour dynamique des matrices assurée dans la fonction SGD qui implémente la descente de gradient stochastique. Pour cela, il a fallu se référer à la documentation mathématique.

Tout d'abord pour chaque input data possible, l'erreur totale du réseau est calculée Cette erreur est calculée avec la fonction d’erreur quadratique dont la dérivée est égale à la

différence entre le label et la prédiction du réseau. Pour chaque neurone de sortie, cette loss est multipliée par la dérivée de la fonction d'activation (sigmoïde dans notre cas) afin de déterminer le gradient, c'est-à-dire combien il faut ajuster ce neurone pour réduire l'erreur.

Ensuite, pour chaque neurone caché, l'erreur est rétro propagée en multipliant par la dérivée de la fonction d'activation de chaque neurone caché pour calculer les gradients des neurones cachés, qui indiquent comment chaque poids et biais de la couche cachée doit être modifié.

Les poids entre la couche cachée et la sortie sont ensuite mis à jour en fonction du gradient de sortie, de la valeur du neurone caché correspondant et du learning rate. De même, les poids entre l'entrée et la couche cachée sont ajustés selon les gradients des neurones cachés et les valeurs des entrées. Les biais des deux couches sont également aux gradients des neurones correspondants.

Ce processus est répété pour chaque échantillon du jeu de données, ce qui constitue une itération complète de l'aglo de descente de gradient stochastique. L'ensemble de ces mises à jour permet de réduire progressivement la fonction de coût, en faisant tendre l'erreur totale vers zéro.

3 Tests et ajustement des hyperparamètres

Après l'implémentation, le réseau a été testé de manière répétée pour apprendre correctement la fonction XNOR en modifiant les hyperparamètres. D'abord la taille de la couche cachée, testé avec 1 à 3 neurones. 2 neurones se sont avérés suffisants pour apprendre XNOR, tandis qu'un seul neurone échouait à converger. Ensuite le learning rate (taux d'apprentissage). Trop faible il ralentit la convergence, trop élevé il provoque une divergence. Un Learning Rate de 0,5 a été le juste milieu. Enfin le nombre d'itérations (epochs). Testé entre 1000 et 100000 epochs.Convergence rapide observée après environ 5000 à 8000 epochs.

L'implémentation de ce réseau "simple"(ou pas) a été une bonne introduction pratique à la rétropropagation et à la descente de gradient. Les principales difficultés ont été la compréhension initiale des dérivées partielles et de la chaîne de rétropropagation, la gestion dynamique des poids et biais en mémoire en C et l'implémentation de l'algorithme de la descente de gradient stochastique dans la fonction SGD.

L'implémentation finale permet au réseau d'apprendre la fonction XNOR de manière fiable, et les tests sur différents hyperparamètres ont illustré l'importance de la taille de la couche cachée, du learning rate et du nombre d'epochs. L'expérience a renforcé la compréhension théorique et pratique des réseaux de neurones simples et de leur entraînement par descente de gradient.

Algorithme du Solver : Luca Macheda

Dans le cadre de ce projet, nous avons développé plusieurs algorithmes permettant de rechercher un mot dans une grille de lettres.

Le principe général est simple : à partir d'un fichier contenant la grille, le programme doit déterminer si un mot donné est présent dans n'importe quelle direction, que ce soit horizontale, verticale ou diagonale, et aussi bien à l'endroit qu'à l'envers. L'objectif n'était pas seulement d'obtenir un résultat correct, mais aussi de réfléchir à la meilleure façon de parcourir la grille pour que la recherche soit efficace et rapide. Nous avons donc testé différents algorithme, comparé leurs performances et cherché à comprendre leurs avantages et leurs limites.

Notre première idée a été de parcourir la grille case par case.Pour chaque lettre rencontrée, on vérifie si elle correspond à la première lettre du mot recherché. Si c'est le cas, on essaie de continuer dans les huit directions possibles (haut, bas, gauche, droite et les quatre diagonales).

Avant de commencer à comparer les lettres dans une direction donnée, nous avons ajouté une vérification : on regarde d'abord si le mot peut "rentrer" dans cette direction, c'est-à-dire si en suivant cette trajectoire, on ne sort pas des limites de la grille. Cette simple condition permet d'éviter beaucoup d'erreurs et de gagner un peu de temps d'exécution.L'algorithme est donc simple à comprendre et à maintenir. Il fonctionne pour tous les cas, quelle que soit la position du mot. Son principal inconvénient est qu'il effectue beaucoup de vérifications, même inutiles, notamment lorsque la grille est grande et que la lettre recherchée apparaît souvent.

Pour essayer d'améliorer les performances, nous avons ensuite conçu un second algorithme, basé sur une idée différente : au lieu de toujours commencer par la première lettre du mot, nous choisissons comme point de départ la lettre la moins fréquente du mot dans la grille.

Concrètement, le programme commence par construire un histogramme des lettres présentes dans la grille. Ensuite, il parcourt le mot à rechercher et repère celle de ses lettres qui apparaît le moins souvent. Cette lettre devient la lettre pivot. L'idée est qu'en partant de cette lettre rare, il y aura moins de positions de départ possibles, et donc moins de directions à tester. Une fois la lettre pivot identifiée, le programme cherche toutes ses occurrences dans la grille. Pour chacune d'elles, il calcule combien de lettres il y a avant et après cette lettre dans le mot. En fonction de la position du pivot, l'algorithme décide si l'on doit vérifier les 8 directions (si

le pivot est proche d'un bord du mot), ou seulement 4 directions (si le mot peut être symétrique autour du pivot).

Avant de se lancer dans une direction, l'algorithme vérifie toujours que le mot peut rentrer entièrement dans cette direction, comme dans la première version. Si c'est possible, il vérifie ensuite les lettres avant et après le pivot, pour voir si elles correspondent bien à celles du mot à rechercher.Cette méthode est intéressante sur le plan théorique, car elle réduit le nombre de points de départ et peut donc potentiellement accélérer la recherche. Même si cette approche semblait plus intelligente et optimisée, sa mise en œuvre s'est révélée compliquée et source d'erreurs.

Le principe de départ était simple : choisir une lettre pivot (celle la plus rare dans le mot), la localiser dans la grille, puis vérifier autour d'elle si le mot pouvait être lu dans différentes directions.Cependant, plusieurs difficultés sont apparues.

D'abord, la gestion des bords de la grille posait souvent problème. Lorsque la lettre pivot se trouve trop proche d'un bord, certaines directions deviennent impossibles à explorer, et il faut ajouter beaucoup de conditions pour éviter de sortir des limites du tableau. Cela rend le code plus long, plus difficile à suivre et augmente les risques d'erreurs.

Ensuite, la logique même des vérifications autour du pivot complique tout. L'algorithme doit vérifier les lettres situées avant et après la lettre pivot, puis refaire le même processus dans le sens inverse pour ne pas rater les mots écrits à l'envers. En pratique, cela revient à dupliquer une partie du code et à multiplier les indices à contrôler, ce qui rend le programme sensible à la moindre erreur.

Enfin, même si le pivot réduit le nombre de positions à tester, l'algorithme doit reconstruire l'histogramme à chaque recherche de mot et tester chaque mot indépendamment, ce qui le rend moins efficace que prévu lorsqu'on a plusieurs mots à chercher.

En résumé, cette méthode avec pivot est plus "intelligente" sur le principe, mais dans la pratique elle s'avère plus complexe, plus sensible aux erreurs, et pas réellement plus rapide qu'une recherche exhaustive.

Après plusieurs essais, mesures de temps et corrections, nous avons pu comparer les performances réelles des différentes approches. La méthode simple, celle où l'on teste chaque case de la grille dans toutes les directions possibles, s'est finalement avérée plus rapide et plus stable que la méthode à lettre pivot. Même si elle effectue davantage de comparaisons, son code est linéaire, clair et ne comporte pas de cas particuliers.La méthode avec pivot, quant à elle, semblait prometteuse, mais en pratique, son coût de préparation (calcul de l'histogramme, gestion des directions et vérifications complexes) la rendait moins efficace pour un usage répété.

Conclusion

Ce projet nous a permis d'aborder de manière concrète la conception d'un système complet capable de résoudre automatiquement une grille de mots cachés à partir d'une image. À travers les différentes étapes du développement, nous avons pu explorer plusieurs domaines complémentaires : le traitement d'image, l'apprentissage automatique et la recherche algorithmique. Chacun de ces volets a contribué à la construction d'un pipeline cohérent, allant de l'analyse visuelle des images jusqu'à la détection des mots dans la grille.

Sur le plan technique, nous avons acquis une solide compréhension du fonctionnement des bibliothèques graphiques telles que SDL2, de la logique événementielle et de la manipulation des pixels. Cette expérience nous a permis de réaliser un prétraitement d'image complet, comprenant la rotation, la conversion en niveaux de gris, la binarisation et la détection de zones cohérentes de pixels. Ces étapes sont essentielles pour la préparation des données destinées à un futur moteur OCR. Parallèlement, la conception et l'entraînement d'un réseau de neurones pour la fonction XNOR ont renforcé nos compétences en mathématiques appliquées et en programmation bas niveau, notamment dans la gestion dynamique de la mémoire et la mise en œuvre de la descente de gradient. Enfin, l'algorithme du solver a permis de développer une réflexion approfondie sur la recherche et l'optimisation, en comparant différentes approches pour identifier les mots dans la grille de manière efficace et robuste.

Au-delà des compétences techniques, ce projet a également été une expérience de travail en équipe enrichissante. La répartition claire des tâches et la complémentarité des domaines d'expertise de chacun ont permis une progression fluide et structurée. Chaque membre a pu approfondir une partie spécifique tout en gardant à l'esprit l'intégration finale des modules dans un système unique et cohérent.

En termes de résultats, les objectifs fixés pour cette première étape ont été atteints. Le système est capable de charger et de traiter des images, de détecter des zones pertinentes, de segmenter les caractères et de rechercher les mots dans une grille donnée. Le réseau de neurones fonctionne correctement sur le cas test de la fonction XNOR, et les différents modules communiquent déjà de manière cohérente. Ces réussites constituent une base solide pour la suite du développement.

Pour les prochaines étapes, nous visons à améliorer la robustesse du pipeline global en intégrant un OCR complet basé sur des réseaux de neurones entraînés à reconnaître les lettres extraites automatiquement. Nous prévoyons également d'optimiser les performances du traitement d'image, d'ajouter un filtrage adaptatif et de perfectionner l'interface graphique. En conclusion, cette première phase représente une avancée majeure vers la réalisation d'un outil intelligent et autonome, capable de résoudre des grilles de mots cachés à partir d'images réelles.

Bibliographie :

documentation mathématique : https://fr.wikipedia.org/wiki/Algorithme_du_gradient
https://www.charlesbordet.com/fr/gradient-descent/#

SGD : http://neuralnetworksanddeeplearning.com/

Reseau de neurones en C :
https://filipposcaramuzza.dev/2024/06/09/implementing-a-neural-network-in-pure-c/

Traitement de l'image : https://www.geeksforgeeks.org/dsa/flood-fill-algorithm/