VOX MACHINA :
ANNA

Rapport de Soutenance 2

Ruben Gervais
Ilyann Gwinner
Nathan Hirth
Josselin Priet

Table des matières

1 Introduction

1.1 Présentation des membres

— Ruben Gervais : Après un Bac général en mathématiques et physique-chimie, je porte beaucoup d'intérêt sur les sciences et en particulier l'informatique. J'ai appris à programmer en python et java sans projet précis en objectif. Je suis dirigé par une grande curiosité et une envie de dépassement de soi. C'est pourquoi ce projet est pour moi une chance et un défi à relever. En apportant mes qualités au groupe, j'espère l'aider à se dépasser et à obtenir des résultats dont nous serons fiers.

— Ilyann Gwinner : Je me suis passionné pour l'informatique en fin 5ème quand un professeur de physique-chimie a parlé d'un langage nommé python. Depuis ce jour, j'ai appris de nombreux langages. J'ai également découvert la cybersécurité qui devint assez rapidement mon centre d'intérêt principal. Depuis quelques années, avec l'explosion de l'IA, j'ai commencé à énormément m'y intéresser. Aujourd'hui, je m'intéresse à beaucoup de domaine de l'informatique comme le hardware, le développement ou encore l'ordinateur quantique, mais la cybersécurité et l'IA reste ce que je préfère. C'est pour cela que je suis heureux de faire ce projet qui me permet de m'en apprendre plus sur l'IA et j'espère comprendre correctement son fonctionnement avec la fin du projet.

— Nathan Hirth : J'ai commencé à m'intéresser grandement l'informatique, il y a un peu plus de six ans. Durant ces quelques année, j'ai pu découvrir différent domaine de l'informatique tel que la création de jeux vidéo, la création de site web, la cybersécurité... Mais je n'ai jamais essayé de faire de l'IA en profondeur malgré le fait que c'est l'un des domaines qui m'intéresse le plus avec celui de la cybersécurité. J'ai réalisé quelque IA, mais avec l'aide de bibliothèque déjà faite, c'est pour cela que pour ce projet, j'ai souhaité me pencher sur la réalisation de l'IA pour la reconnaissance de caractères. Comprendre comment fonctionne l'IA, quelles sont les fonctions qu'elle utilise pour obtenir de tels résultats, c'est ce que j'aimerais atteindre à la fin de ce projet.

— Josselin Priet : Intéressé à l'origine par le journalisme, je me suis très vite redirigé vers l'informatique durant la période du lycée. J'ai ainsi pu peaufiner mes connaissances et découvrir davantage ce monde, ce qui m'a permis d'engranger de l'expérience. Ce projet est donc pour moi une nouvelle manière d'apprendre dans un nouveau langage. Malgré cela, j'espère aussi pouvoir utiliser les outils et astuces que j'ai pu développer depuis le lycée, en particulier en algorithmie.

1.2 Tâches

Tâche	Avancement prévu	Avancement réel	Responsable
Preprocessing			/
GrayScale	100%	100%	Ruben
Binarisation	100%	100%	Ruben
Rotation Manuelle	100%	100%	Josselin
Rotation Automatique	100%	95%	Josselin
Cropping	100%	80%	Ruben
AI			/
NXOR	100%	100%	Ilyann et Nathan
ANNA	100%	95%	Ilyann et Nathan
Other			/
Solver	100%	100%	Nathan
UI	100%	100%	Ilyann

On peut constater que les objectifs fixés pour cette dernière soutenance ont été globalement bien atteints, ce qui reflète une avancée significative par rapport aux étapes précédentes du projet. La plupart des fonctionnalités prévues ont été implémentées avec succès, et les résultats obtenus dans plusieurs domaines sont encourageants. Cependant, il reste un certain retard concernant le cropping, un problème déjà évoqué lors de la première soutenance et qui n'a pas encore été totalement résolu.

Plus précisément, si le cropping fonctionne correctement sur les cas les plus simples et les premiers niveaux de complexité, des dysfonctionnements apparaissent lorsque la difficulté augmente. Dans ces cas, certaines lettres ne sont pas détectées de manière optimale, ce qui entraîne des erreurs dans le traitement. De plus, des éléments indésirables, qui ne sont ni des mots ni des lettres, sont parfois identifiés par erreur, ce qui complique davantage le processus. Ces limitations impactent directement l'efficacité de la segmentation et peuvent avoir des répercussions sur les étapes suivantes, comme la reconnaissance des caractères.

Ce retard peut s'expliquer par plusieurs facteurs, notamment une gestion du temps qui aurait pu être mieux optimisée. La complexité technique du cropping semble avoir été sous-estimée, ce qui a conduit à un manque de temps pour affiner et corriger cette partie. Par ailleurs, une répartition des tâches légèrement déséquilibrée au sein de l'équipe pourrait également avoir contribué à ce retard. En concentrant davantage d'efforts sur d'autres fonctionnalités, le développement du cropping a probablement été relégué au second plan, ce qui a limité les avancées possibles dans ce domaine.

2 Preprocessing

Cette section contient tout ce qui se passe avant l'AI. Pour le préprocessing, l'image passera par une étape de grayscale, de binarisation, de rotation et enfin de cropping.

2.1 GrayScale

Le passage d'une image en niveaux de gris, également appelé grayscale, constitue une étape fondamentale et incontournable du processus de prétraitement des données (preprocessing). Cette étape consiste à convertir une image initialement en couleurs, composée des trois canaux de base (rouge, vert et bleu, souvent abrégés en RGB), en une image en niveaux de gris où chaque pixel est représenté par une seule intensité lumineuse. Cela simplifie considérablement le traitement des images, car elle réduit la complexité des calculs tout en conservant les informations visuelles essentielles.

Pour réaliser cette conversion, nous avons utilisé une formule standard qui attribue des pondérations spécifiques aux trois canaux de couleur en fonction de leur contribution perçue à la luminosité globale de l'image. La formule que nous avons appliquée est la suivante :

$$GrayScale = 0.3 * R + 0.59 * G + 0.11 * B$$

Ici, représente l'intensité du rouge, G celle du vert et B celle du bleu pour chaque pixel. Ces coefficients reflètent la sensibilité de l'œil humain, qui perçoit plus fortement le vert, modérément le rouge, et moins le bleu.

L'intérêt principal de cette transformation réside dans la simplification qu'elle apporte. En passant de trois valeurs par pixel à une seule, elle permet de réduire la quantité de données à manipuler, ce qui est particulièrement avantageux pour les étapes ultérieures de traitement et d'analyse. Par exemple, dans le cadre de notre projet, cela facilite non seulement la binarisation, mais aussi d'autres étapes comme la rotation et le découpage (cropping).

De plus, en supprimant les informations de couleur, cette étape met l'accent sur les formes et les motifs présents dans l'image, rendant ainsi les caractéristiques visuelles plus cohérentes et universelles, indépendamment des variations chromatiques. Cela s'avère crucial dans des tâches comme la reconnaissance de caractères, où la couleur des lettres ou du fond est généralement une distraction inutile pour les algorithmes.

Ainsi, le grayscale constitue une phase préparatoire essentielle pour garantir une meilleure efficacité des étapes suivantes tout en maintenant une représentation fidèle et exploitable des données visuelles.

Voici un exemple de ce que donne le grayscale :

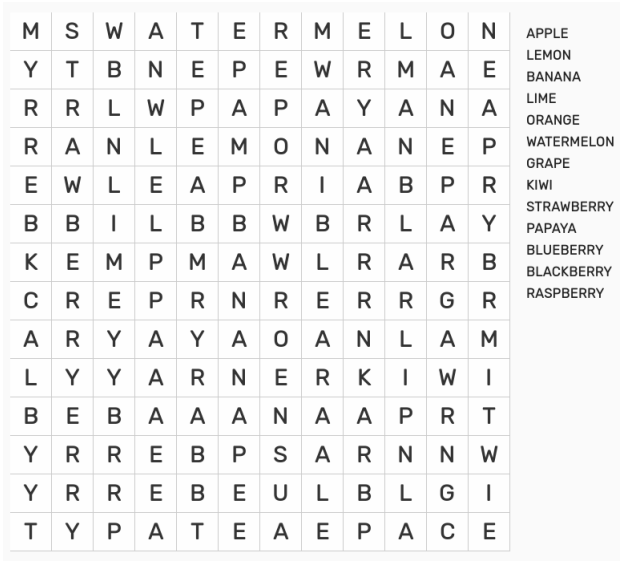

FIGURE 1 – Application du grayscale

2.2 Binarisation

La transformation d'une image en noir et blanc, également appelée binarisation, est une étape clé dans le processus de prétraitement, qui suit naturellement la conversion en niveaux de gris. Cette étape vise à simplifier davantage l'image en éliminant toutes les nuances de gris, réduisant ainsi les données à deux états possibles : noir (valeur 0) et blanc (valeur 255). Cela permet de concentrer l'analyse sur les éléments essentiels de l'image tout en supprimant les détails superflus, tels que les variations subtiles de luminosité ou de texture.

our réaliser cette binarisation, nous avons utilisé l'algorithme d'Otsu, une méthode statistique robuste et largement reconnue. Cet algorithme a pour objectif de séparer les pixels d'une image en deux classes distinctes : le premier plan, correspondant généralement aux objets d'intérêt, et l'arrière-plan. Il le fait en calculant un seuil optimal qui maximise la distinction entre ces deux classes, ce qui permet d'obtenir une segmentation claire et précise.

L'algorithme d'Otsu se déroule en plusieurs étapes :

- Dans un premier temps, on analyse la distribution des intensités de gris dans l'image. Cette étape permet de déterminer la fréquence de chaque niveau de gris, qui sert de base au calcul du seuil optimal.

- Une fois l'histogramme établi, l'algorithme calcule le seuil qui minimise la variance intra-classe (la dispersion des pixels à l'intérieur de chaque classe) et maximise la variance inter-classe (la distinction entre les classes). La formule utilisée est la suivante :

$$\sigma_\omega^2(t) = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t)$$

$$\sigma_b^2(t) = \sigma^2 - \sigma_\omega^2(t) = \omega_1(t)\omega_2(t)\left[\mu_1(t) - \mu_2(t)\right]^2$$

Où les poids ω_i représentent la probabilité qu'un pixel appartienne à la classe i, chaque classe étant définie par un seuil t. Et où les σ_i^2 sont les variances de ces classes.

- Une fois le seuil calculé, chaque pixel de l'image est évalué individuellement. Si la valeur de son intensité est supérieure au seuil, le pixel est défini comme blanc (valeur 255). Dans le cas contraire, il est défini comme noir (valeur 0). Ce processus transforme l'image en une représentation binaire prête à être analysée par les étapes suivantes.

Voici un exemple d'image après avoir appliqué la binarisation :

FIGURE 2 – Application de la binarisation

2.3 Rotation

2.3.1 Rotation Manuelle

L'objectif est de faire tourner une image en spécifiant un angle en degrés et de l'afficher ensuite grâce à SDL.

Fonctionnement :

La rotation manuelle d'une image constitue une étape fondamentale pour garantir une orientation correcte et adaptée à l'analyse ultérieure. En utilisant les fonctionnalités de base de la bibliothèque SDL associées aux outils mathématiques standards, nous avons pu implémenter un programme capable d'effectuer des rotations d'images de manière fluide et précise. Cette approche repose sur la conversion des angles exprimés en degrés en radians, une opération essentielle pour effectuer des calculs trigonométriques nécessaires à la rotation.

SDL s'avère être une bibliothèque puissante et intuitive pour manipuler les éléments graphiques. Elle propose une série d'outils qui simplifient la gestion des transformations, comme la rotation, le redimensionnement ou la translation d'images. Grâce à ces fonctionnalités, notre programme peut appliquer une rotation en spécifiant un angle exact, offrant une grande précision et flexibilité.

Le fonctionnement repose sur un processus relativement simple, mais efficace :

- La rotation d'une image nécessite de travailler avec des radians, car les fonctions trigonométriques utilisées pour manipuler les coordonnées (comme le sinus et le cosinus) opèrent dans ce format. La conversion est effectuée à l'aide de la formule suivante :

$$radians = degres \times \frac{\pi}{180}$$

- Une fois l'angle converti, le programme utilise les fonctions de SDL pour redessiner chaque pixel de l'image en fonction de ses nouvelles coordonnées. Ces coordonnées sont calculées à partir des formules de rotation :

$$x' = xcos(\theta) - ysin(\theta)$$
$$y' = xcos(\theta) + ysin(\theta)$$

Où θ est l'angle de rotation en radians, et (x', y') représentent les nouvelles coordonnées du pixel après rotation.

- Lors de la rotation, certains pixels peuvent se retrouver en dehors des limites initiales de l'image. SDL gère automatiquement ces cas en ajustant les dimensions de l'espace d'affichage pour inclure tous les pixels de l'image transformée.

L'utilisation de SDL pour la rotation manuelle apporte une grande simplicité et une robustesse à notre programme. Elle permet non seulement d'effectuer des rotations précises, mais aussi de visualiser instantanément les résultats, ce qui est essentiel dans le cadre d'un processus

interactif ou d'une interface utilisateur.

Cette implémentation constitue une base solide pour des transformations plus complexes, comme la rotation automatique, qui détecte et ajuste automatiquement l'orientation des images sans intervention humaine. Grâce aux outils proposés par SDL, la rotation manuelle est devenue une opération à la fois performante et accessible, rendant le programme facile à intégrer dans une chaîne de traitement d'images.

2.3.2 Rotation Automatique

Le deuxième objectif, plus complexe, est de faire tourner l'image automatiquement, sans spécifier d'angle. Le programme doit ainsi détecter si l'image n'est pas droite et appliquer la rotation nécessaire. La fonction de rotation manuelle fonctionne parfaitement, tout comme la rotation automatique. Cependant, dans certaines conditions (par exemple, un angle trop élevé), des problèmes peuvent apparaître. Malgré cela, pour les objectifs du projet, la rotation automatique s'avère être très performante.

Fonctionnement :

La détection de l'angle de rotation automatique repose sur la méthode de la transformée de Hough, qui permet de détecter les lignes présentes dans l'image. Avant d'entrer dans cette étape, l'image a déjà été prétraitée par Ruben.

Cette étape simplifie le traitement en réduisant l'image à des valeurs de luminosité plus simples à analyser.

Une fois ce prétraitement effectué, le programme parcourt chaque pixel de l'image pour identifier les pixels blancs. Pour chaque pixel identifié, il calcule la transformée de Hough en testant divers angles (θ) et distances (r) possibles. L'espace de Hough est ensuite utilisé pour accumuler des votes associés à chaque combinaison de r et θ, représentant ainsi les différentes lignes potentielles dans l'image.

Une fois la transformée de Hough calculée, le programme analyse l'espace de Hough pour identifier l'angle dominant, celui qui a recueilli le plus grand nombre de votes. Cet angle sera ensuite utilisé pour déterminer l'orientation de l'image. Afin de corriger certaines inversions possibles (par exemple, une détection erronée où l'image serait à l'envers ou sur le côté), des ajustements sont appliqués à l'angle détecté. L'angle final, après correction, est ensuite retourné, permettant ainsi d'effectuer une rotation pour redresser correctement l'image.

Détail de la Transformée de Hough :

La transformée de Hough est une technique qui permet de détecter des formes géométriques dans une image, notamment les lignes. Au lieu de travailler directement dans l'espace de l'image, la transformée de Hough transforme chaque point en une courbe dans un espace paramétrique constitué de deux paramètres : r et θ.

- r est la distance entre l'origine et la ligne, mesurée perpendiculairement à la ligne.

- θ est l'angle entre l'axe x et la normale à la ligne.

La formule associée à la paramétrisation d'une ligne est la suivante :

$$r = xcos(\theta) + ysin(\theta)$$

Chaque pixel de l'image contribue à l'augmentation du compteur dans l'espace de Hough en fonction de sa position (x,y) et de l'angle θ. La ligne la plus représentée (c'est-à-dire celle avec le plus grand nombre de votes dans l'espace de Hough) correspondra à l'orientation dominante de l'image.

Échantillons et tests :

Le programme a été testé sur un grand nombre d'images, ce qui a permis d'identifier quelques problèmes mineurs. Toutefois, sur l'échantillon testé, la rotation automatique atteint actuellement environ 90% de réussite. Si l'on exclut les images ne respectant pas les consignes du projet, comme celles contenant des angles trop marqués, le taux de réussite monte à 99%.

Exemple :

Voici un exemple d'image avant l'application de la fonction pour la rotation automatique :

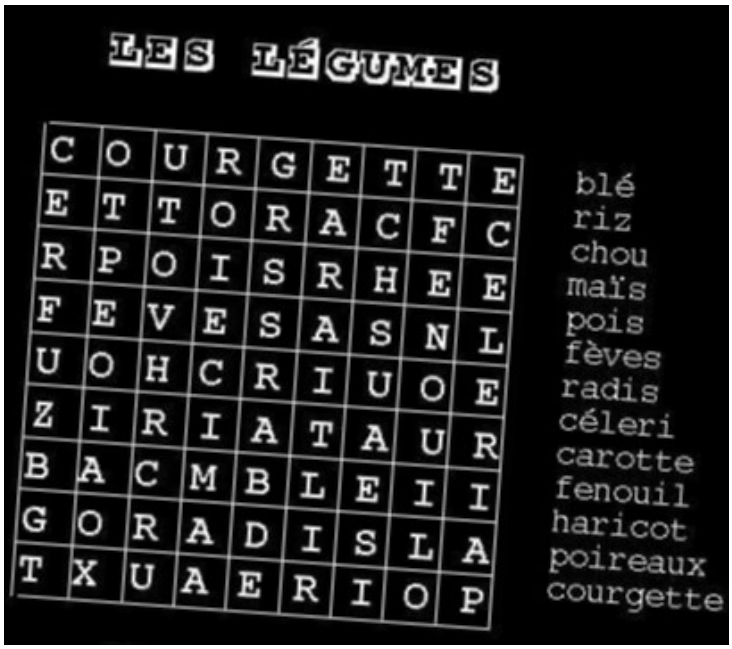

FIGURE 3 – Image avant la rotation automatique

Et voici maintenant la même image après l'application de la rotation automatique :

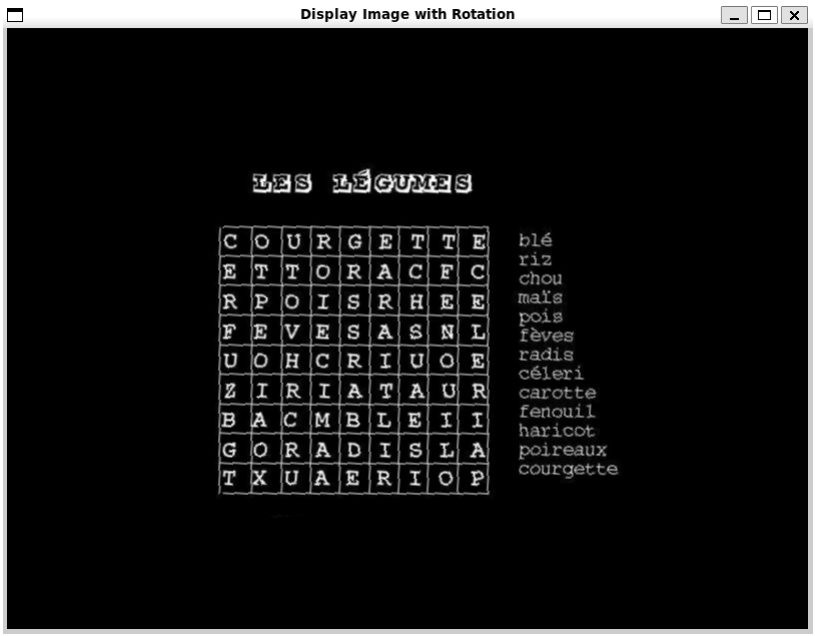

FIGURE 4 – Image avant la rotation automatique

2.4 Cropping

L'une des étapes essentielles de notre traitement consiste à distinguer les mots présents dans la liste des lettres composant la grille. Cette dis-

tinction est fondamentale pour le bon fonctionnement de notre système, car elle permet de segmenter et de classer correctement les éléments textuels. Pour ce faire, nous utilisons un algorithme basé sur la détection des composantes connectées. Une fois les composantes identifiées, nous les classifions comme étant soit des mots issus de la liste, soit des lettres isolées dans la grille, en nous appuyant sur leur rapport largeur/hauteur

Détection des Composantes Connectées :

L'algorithme employé pour détecter les composantes connectées repose sur la méthode dite du "flood-fill" (ou remplissage par diffusion). Cette technique analyse l'image pour identifier des zones contiguës de pixels connectés ayant la même couleur, qui, dans notre cas, correspond généralement aux pixels blancs représentant les lettres ou les mots. Chaque zone détectée est ensuite étiquetée avec un identifiant unique, facilitant ainsi le traitement et l'analyse ultérieurs. Cette étape est cruciale, car elle constitue la base de la segmentation et garantit que chaque élément textuel est traité de manière indépendante.

Calcul des Boîtes Englobantes :

Pour chaque composante connectée identifiée, une boîte englobante (bounding box) est calculée. Cela consiste à déterminer les coordonnées minimales et maximales des pixels en x et y pour délimiter la région occupée par chaque composante. Une fois ces limites définies, la largeur et la hauteur de la boîte sont calculées, fournissant des dimensions précises pour chaque élément détecté. Ces boîtes englobantes permettent non seulement de visualiser les composantes, mais elles servent également d'entrée pour les étapes de classification.

Séparation des Mots et des Lettres Basée sur le Rapport d'Aspect (Aspect Ratio) :

Une fois les boîtes englobantes obtenues, nous procédons à leur classification en calculant leur rapport largeur/hauteur (aspect ratio). Cette métrique est déterminante, car elle permet de différencier les mots des lettres uniques en fonction de leurs proportions géométriques. Une règle empirique est appliquée :

- Si le rapport largeur/hauteur est supérieur à 2, il est probable qu'il s'agisse d'un mot de la liste.

- Si le rapport est proche de 1, cela indique qu'il s'agit vraisemblablement d'une lettre isolée de la grille.

Cette classification est essentielle pour éviter toute ambiguïté entre

les deux types d'éléments et garantir une séparation claire et exploitable. Elle permet également de limiter les erreurs de détection en s'appuyant sur des critères géométriques simples mais robustes.

Voici un exemple illustrant le résultat obtenu après la séparation des composantes connectées en mots et lettres. Ces étapes permettent de transformer une image brute en une structure organisée, prête à être utilisée dans les phases ultérieures, telles que la reconnaissance de caractères ou la recherche de mots dans la grille :

FIGURE 5 – Séparation des Mots et des Lettres

Extraction et Enregistrement des Composantes :

Pour les lettres de la grille :

Chaque lettre détectée dans la grille est également extraite individuellement pour garantir une représentation claire et uniforme. Après extraction, chaque lettre est redimensionnée à une taille standardisée, telle que 16x16 pixels, ce qui assure une cohérence dans les données et facilite leur traitement dans les modèles d'apprentissage automatique ou de reconnaissance optique de caractères (OCR). Une fois redimensionnée, chaque lettre est enregistrée dans un fichier CSV où ses pixels sont représentés sous forme de lignes et de colonnes, reflétant fidèlement sa structure visuelle.

Le processus de nommage des lettres de la grille suit un format rigoureux, tel que grid_{x_index}_{y_index}.png. Ce format inclut les coordonnées de la lettre dans la grille, spécifiées par ses indices de ligne (x) et de colonne (y), ce qui permet de localiser chaque lettre avec précision dans le contexte de la grille. Cette méthode garantit une organisation

claire des données et facilite leur utilisation dans les algorithmes de recherche ou de reconstruction de texte.

De plus, le stockage dans des fichiers CSV permet une manipulation et une analyse efficace des lettres, tout en réduisant l'espace disque requis par rapport aux formats d'image classiques. Ces fichiers peuvent être utilisés comme entrée directe pour les modèles ou pour des analyses statistiques sur les données, offrant une grande flexibilité dans les traitements ultérieurs. Grâce à cette structuration soignée, les mots et les lettres extraits sont non seulement prêts pour les étapes suivantes, mais également facilement traçables et manipulables pour diverses applications.

Une fois ces étapes réalisées, nous obtenons une séparation claire entre les mots de la liste et les lettres de la grille. Ce processus garantit une préparation optimale des données pour la reconnaissance de texte ou d'autres analyses.

Ce pipeline méthodique assure une robustesse dans l'extraction et le traitement des composants textuels, facilitant ainsi les étapes ultérieures et minimisant les erreurs potentielles.

3 AI

Cette section se divise en deux sous-parties. La première est une AI qui a pour but de reconnaitre la porte logique NXOR ($A.B + \overline{A}.\overline{B}$). Ce modèle permet de vérifier si l'architecture du réseau créé pour la deuxième partie est capable d'apprendre sur un système simple. La seconde est une AI capable de reconnaitre les lettres de l'alphabet minuscule et majuscule. Nous avons name cette deuxième IA ANNA pour Advanced Neural Network Analyser.

3.1 NXOR

Pour réaliser l'IA qui reconnait la porte NXOR, nous avons utilisé un modèle simple de réseau de neurone. Cette IA contient deux neurones d'entrer, trois neurones dans la couche cachée et un neurone de sortie. Il contient donc trois couches de neurones.

Afin de fonctionner, ces différents neurones sont reliés par des synapses. Ces synapses relient les neurones d'entrer aux neurones de la couche cachée et les neurones de la couche cachée au neurone de sortie. Chaque synapse contient deux valeurs pour pouvoir faire le lien entre les différentes couches, la première valeur est le poids de la synapse et la seconde est le biais.

Grâce à trois fonctions permettant de calculer la sortie, et de calculer et de mettre en place l'erreur et la correction du modèle, l'IA à partir de poids et biais aléatoire peut apprendre à reconnaitre la porte NXOR.

- Forward Propagation : Cette fonction permet de calculer la valeur de la sortie. Elle multiplie les valeurs de chaque neurone de chaque couche avec les poids des synapses correspondant et en y ajoutant les biais pour donner les valeurs des neurones de la couche suivante. Une fois calculé, on passe le résultat de chaque neurone dans une fonction d'activation qui permet de garder la valeur entre 0 et 1, ici la fonction sigmoïde dont la formule est :

$$Sigmode = \frac{1}{1 + e^{-x}}$$

À la fin, la valeur du neurone de sortie nous donne la probabilité que le résultat du NXOR soit 1. Si cette probabilité inférieure à 0.5 alors la valeur de sortie est 0 sinon c'est 1.

- Back Propagation : La rétropropagation est une fonction qui permet de calculer les erreurs du réseau de neurone afin de l'ajuster les résultats de l'IA. Cette fonction utilise la dériver de la fonction d'activation, donc pour la sigmoïde, nous avons :

$$SigmodePrime = x * (1 - x)$$

Une fois cette fonction appliquée sur les différentes couches du réseau, nous obtenons les valeurs pour l'ajustement des différents poids et biais de l'IA.

- Update : Cette fonction permet de mettre à jour les poids et les biais des différentes synapses. Nous utilisons la régulation L2 afin d'optimiser la rapidité d'apprentissage du réseau. Cette fonction va aussi utiliser une variable qu'on appelle learning rate. Cette variable permet de contrôle de la vitesse de convergence, de stabilité de l'apprentissage et d'équilibre entre exploration et convergence. Plus elle est petite, plus le modèle convergera, mais le temps d'apprentissage sera plus élevé.

Pour vérifier et suivre l'apprentissage de l'IA, nous utilisons une loss function du nom de Log Loss dont la formule est :

$$LogLoss = -\frac{1}{N}\sum_{i=1}^{N}(y_i * log(p_i) + (1 - y_i) * log(1 - p_i))$$

Avec N le nombre d'observations, yi le résultat binaire réel (0 ou 1) pour la i-ième observation et pi la probabilité prédite que la i-ième observation.
Cette fonction permet d'avoir la valeur de l'écart entre les résultats du

modèle et donc sa performance. Si l'IA donne un résultat de 1 à 60% et un 1 à 90%, la fonction Log Loss donnera un résultat plus petit pour le deuxième cas.

Afin d'optimiser l'apprentissage du réseau, nous initialisons les poids et les biais aléatoirement grâce à l'initialisation de Xavier qui est faite pour la fonction d'activation sigmoïde.

Une fois le programme lancer, le modèle arrive à 100% de réussite en environ 300 boucle d'entraînement (chaque boucle comprendre les trois fonctions vu précédemment).

3.2 ANNA

Contrairement au NXOR le réseau de neurone artificiel ANNA que nous utilisons pour reconnaitre des lettres est bien plus complexe dans sa réalisation. Afin d'arriver au résultat actuel, nous avons expérimenté de nombreuses architectures de réseau de neurone différent.

Notre programme pour la reconnaissance de caractère peu être découpé en 4 phase distincte :

- Le chargement des images, des poids et des paramètres du réseau,

- La forward propagation,

- La back propagation,

- Et l'update et la sauvegarde des poids et des paramètres.

3.2.1 Dataset

Entrainer un modèle d'IA tel qu'ANNA demande un certain nombre de données qui doit être judicieusement choisi afin d'être le plus efficace possible. Dans notre cas, nous avons récupéré les nombreuses polices d'écriture de Google Fonts. Puis grâce à celle-ci, nous avons pu générer des images en blanc sûr un fond noir de la taille que nous voulions. Grâce à cette opération, nous avons récupéré plus de 300000 images au format 16*16 pixels.

Mais après, nous avons dû vérifier que les images générées par notre programme donnaient des images visibles et reconnaissables. Cela nous a pris beaucoup de temps et même si on a enlevé la plupart des images incorrecte, il doit sûrement en rester certain que ne sont pas très visibles même si ce n'est qu'un infime partie. Il nous restait à la fin 5808 polices d'écriture différentes.

Voici un exemple d'image en 16*16 qu'on obtient à la fin :

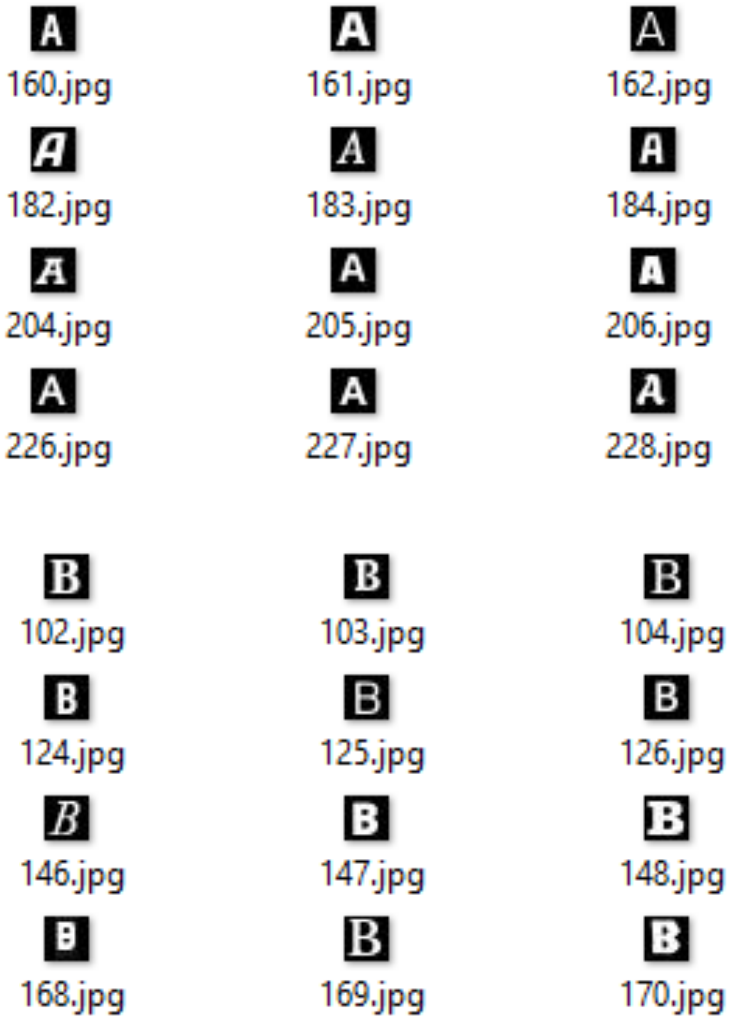

FIGURE 6 – Exemple d'image des lettres A et B en 16*16

On peut voir sur ces images que chaque image porte un numéro correspondant au numéro que nous avons attribués à chaque police d'écriture de notre dataset.

Nous avons structuré notre jeu de données de manière méthodique pour faciliter son utilisation lors de l'entraînement et du test de notre IA. Chaque image du dataset est associée à un numéro unique correspondant à la police d'écriture utilisée pour la générer. Ce numéro permet de retracer facilement l'origine de chaque image et d'assurer un tri rigoureux des données.

Le jeu de données est organisé dans un système de dossiers bien structuré. Chaque lettre de l'alphabet, en majuscule ou en minuscule,

possède son propre dossier, ce qui donne un total de 52 dossiers. À l'intérieur de ces dossiers, nous avons stocké toutes les images générées pour chaque lettre en utilisant les 5808 polices d'écriture récupérées. Cela représente un total impressionnant de 302016 images, avec 5808 images par lettre. Cette organisation garantit une accessibilité rapide et une manipulation facile des données lors des phases de prétraitement et d'entraînement.

À partir de cette structure, nous avons généré des fichiers CSV pour l'entraînement et le test. Les fichiers CSV ont été choisis pour leur légèreté et leur simplicité de manipulation. Pour chaque image, un programme dédié extrait les pixels, leur applique une conversion en niveaux de gris (grayscale), puis enregistre ces pixels sur une seule ligne dans le fichier CSV, les séparant par des virgules. En parallèle, nous générons un fichier CSV contenant les réponses attendues. Chaque réponse est codée sous forme d'un vecteur de 52 valeurs (une par lettre), où la valeur correspondant à la lettre cible est 1, et les autres sont 0. Par exemple, pour une image représentant un "A", le fichier de réponse contiendra "1" suivi de 51 "0", séparés par des virgules.

Nous avons créé un total de 90 fichiers CSV pour l'entraînement et 1 pour le test. En comptant les fichiers de réponse, cela porte le total à 180 fichiers pour l'entraînement et 2 pour le test. Chaque fichier d'entraînement contient 2600 images, soit un total de 234000 images d'entraînement. De son côté, le fichier de test contient 68016 images. Cette répartition garantit une uniformité dans les données : chaque fichier contient le même nombre d'instances pour chaque lettre, ce qui facilite leur utilisation en tant que batches d'entraînement. Par exemple, dans un fichier d'entraînement, il y a exactement 50 lignes pour chaque lettre.

Ce dataset structuré en deux parties distinctes permet d'évaluer le modèle sur deux fronts. Les fichiers d'entraînement permettent de peaufiner les performances de l'IA en apprenant sur des données déjà vues, tandis que le fichier de test sert à mesurer la capacité du modèle à généraliser en détectant des lettres sur des images qu'il n'a jamais vues auparavant. Cette distinction entre données d'entraînement et de test est essentielle pour valider la robustesse et l'efficacité du modèle.

3.2.2 Chargement des données

Notre IA de reconnaissance de caractère possède trois modes différents : la réinitialisation, l'entraînement et la prédiction. Mis à part la réinitialisation, les modes ont besoin de charger les paramètre du réseau de neurone avant de pouvoir être utilisable.

Il y a différent paramètre à charger : les poids et les biais du réseau,

mais aussi les hyperparamètre. Nous verrons plus tard à quoi sert ces différents paramètres. Afin de les charger, nous avons une fonction qui regarde dans plusieurs fichiers CSV puis extraire les donner de ces fichiers. Cette opération est faite à chaque fois qu'on lance le programme. Cela permet de soit reprendre l'entrainement avec les paramètres précédent ou donner une nouvelle image au réseau pour qu'il nous dise quelle lettre c'est afin de faire une prédiction.

Au moment du chargement des images, nous les chargeons batch par batch afin d'améliorer la rapidité d'apprentissage de l'IA et nous mélange les batchs ainsi que les images à l'intérieur aléatoirement afin que l'IA n'apprenne pas l'ordre des réponses par cœur.

Une fois les données chargées, elles sont mises dans un struct qui contient toutes les variables important utiliser dans notre IA afin d'être facilement accessible et utilisable.

Toutes les donner ou tableau utilisé dans les calculés de l'IA que nous verrons après et qui n'ont pas été initialisés grâce au fichier CSV sont initialisés au moment du chargement des données. Étant donné que notre réseau de neurone demande d'allouer beaucoup de mémoire, nous utilisons les fonctions malloc() et calloc() pour allouer cette mémoire dans la heap.

3.2.3 Forward Propagation

Une fois les données correctement charger, nous calculons la sortie du modèle à l'aide de la propagation avant. Comme nous l'avons vu précédemment, nos données d'entraînement sont des images en 16*16, notre réseau de neurone prend donc en entrer 256 valeurs, il a donc 256 neurones d'entrer. Étant donné que nous cherchons à reconnaitre les lettres de l'alphabet, notre réseau de neurone sans forcément une distinction entre les minuscules et les majuscules, car en sortie, on les met tous en majuscule, nous avons au début mis 26 neurones de sortie. Mais suite à plusieurs tests, nous sommes arrivés à la conclusion que mettre 52 neurones de sortie était mieux pour l'apprentissage.

Les calculs effectués dans la propagation avant sont assez simples. Pour calculer la couche suivante, nous avons besoin des valeurs de la couche d'avant, par exemple pour la première couche caché, nous utilisons les pixels des images que nous avons prises en entrer. Nous avons aussi besoin des poids et des biais que nous avons chargés au début du programme. Voici la formule permettant de calculer les neurones de la couche suivante :

$$N_i = \sum_{j=0}^{n} N_j * w_{ij} + b_j$$

Avec Ni la valeur du neurone i dans la couche actuelle. Nj la valeur du neurone j dans la couche précédente. wij le poids entre le neurone j de la couche précédente et le neurone i de la couche actuelle. bi le biais ajouté au neurone i. n le nombre total de neurones dans la couche précédente.

Suite à ce calcul, nous passons le résultat obtenu dans une fonction d'activation qui contrairement au Poc n'est pas sigmoïde, mais ReLU dont voici la formule :

$$ReLU = max(0, x)$$

Nous utilisons la fonction ReLU seulement pour les couches cachée, car pour calculer la dernière couche, nous utilisons la fonction softmax dont voici la formule :

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_i}}$$

Avec zi représentant la sortie du neurone i de la dernière couche, k étant le nombre total de neurones dans cette couche. La fonction softmax transforme ces sorties en probabilités, permettant d'interpréter les sorties du réseau comme des scores de classification, où la somme des probabilités est toujours égale à 1.

La fonction softmax nous a permis des passés d'un d'un entraînement au début qui commence aux alentours de 10% de réussite à un entraînement qui comment à 40% de réussite.

Une fois ces calculs effectuée, nous obtenons le résultat qu'à donner notre IA. La valeur de chacun des 52 neurones de sortie correspond à la probabilité que ce soit la lettre représentée par l'indice du neurone de sortie. Par exemple, si le premier neurone de sortie contient la valeur 0.7, cela veut dire qu'il y a 70% de chance que la lettre soit un A d'après l'IA.

Si nous laissons notre fonction de propagation avant comme ça, cela pourrait poser des problèmes de surapprentissage dans l'entraînement du modèle. C'est pour cela que nous avons ajouté un dropout à notre fonction. Ce dropout permet de désactiver certains neurones afin que le modèle ne se focalise pas trop sur les donner d'entraînement et qu'il puisse donc reconnaitre des lettres qu'il n'a jamais vues.

3.2.4 Back Propagation

Quand nous avons fini de calculer la sortie et si nous sommes dans le mode d'entraînement, nous allons utiliser la fonction de rétro propagation. Cette fonction permet de calculer la dériver des poids et des

biais de notre réseau. Pour ce faire, nous allons utiliser la dériver de la fonction d'activation ReLU qui est :

$$ReLUPrime(x > 0) = 1$$

$$ReLUPrime(x <= 0) = 0$$

Le fait de calculer les dérivés des poids et des biais va nous permettre de mettre à jour les poids et les biais de notre IA.

C'est dans cette fonction-là que les fichiers contenant les résultats des images vont nous être utile. Ces résultats nous permettent de calculer l'erreur de notre réseau de neurone en faisant la différence entre la sortie que l'IA nous a donnée et ce qu'on attendait.

3.2.5 Update

La mise à jour des poids et des biais est très importante pour que notre réseau de neurone puisse apprendre correctement. Avec les dérivées que nous avons calculées plus tôt, il existe plusieurs façons d'optimiser la mise à jour des poids. Au début, nous avions juste mis à jour les poids avec la descente de gradient et ajouter la régulation L2.

La régulation L2 est une méthode utilisée pour réduire le surapprentissage dans les réseaux de neurones. Elle agit en ajoutant une pénalité basée sur la somme des carrés des poids au coût total de la fonction de perte. Cela encourage le réseau à apprendre des poids plus petits, ce qui a tendance à simplifier le modèle et à améliorer sa capacité de généralisation. La fonction de perte avec régulation L2 est donnée par :

$$L_{total} = L_{original} + \frac{\lambda}{2} \sum_i w_i^2$$

Avec L total, la perte totale avec régulation. L original, la perte calculée sans régulation. Lambda, le facteur de régularisation. Et wi, les poids individuels du réseau.

Lors de la mise à jour des poids avec la descente de gradient, la dérivée partielle de la régulation L2 par rapport aux poids est ajoutée, ce qui donne :

$$w_i = w_i - \eta(\frac{\partial L_{original}}{\partial w_i} + \lambda w_i)$$

Avec eta, le learning rate.

Après avoir mis en place la descente de gradient ainsi que la régulation L2 notre modèle était bloqué en dessous des 50% de réussite sur le dataset d'entrainement comme on peut le voir sur ce graphique.

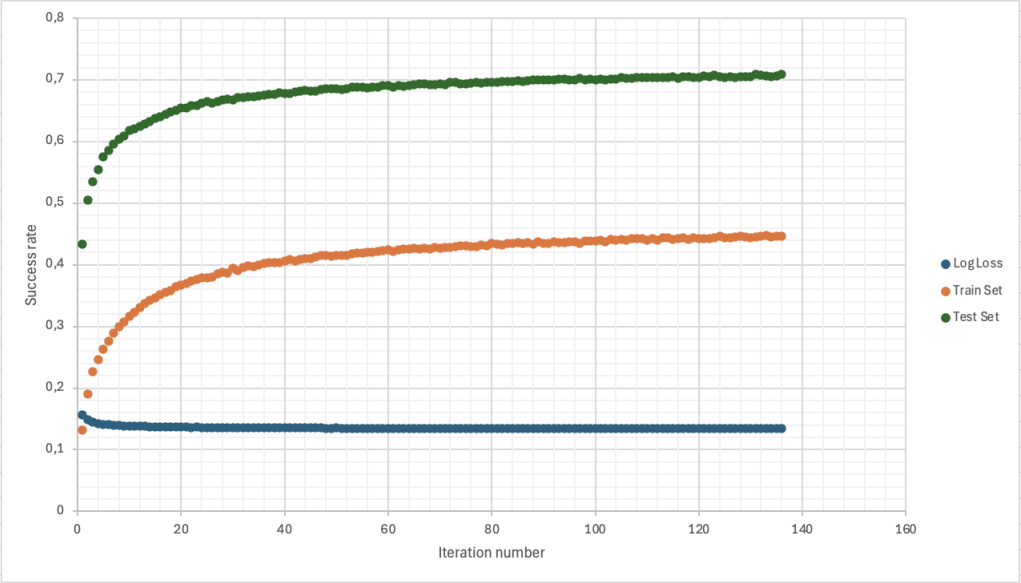

FIGURE 7 – Statistique du modèle avec la descente de gradient et la régulation L2

Sur ce graphe, nous pouvons voir trois courbes qui sont en fonction du nombre d'itérations de l'IA. Celle qui nous intéresse ici est celle en orange qui représente le train set. On peut voir quel augment beaucoup au début, mais arrivent à un plateau à partir des 45%.

Pour palier a ce problème, nous avons ajouté un nouvel optimiseur à notre modèle. Cet optimiseur se nomme Adam. L'optimiseur Adam (pour "Adaptive Moment Estimation") est un des algorithmes d'optimisation les plus populaires en apprentissage profond. Il combine les avantages de deux autres méthodes, AdaGrad et RMSProp. Il permet d'ajuster le learning rate vu précédemment pour chaque paramètre du modèle (les poids et les biais). Cela permet d'augmenter considérablement l'efficacité de l'entrainement du modèle. Cet optimiseur nous a permis de passer de 45% de réussite à plus de 60% de réussite comme le montre ce nouveau graphe.

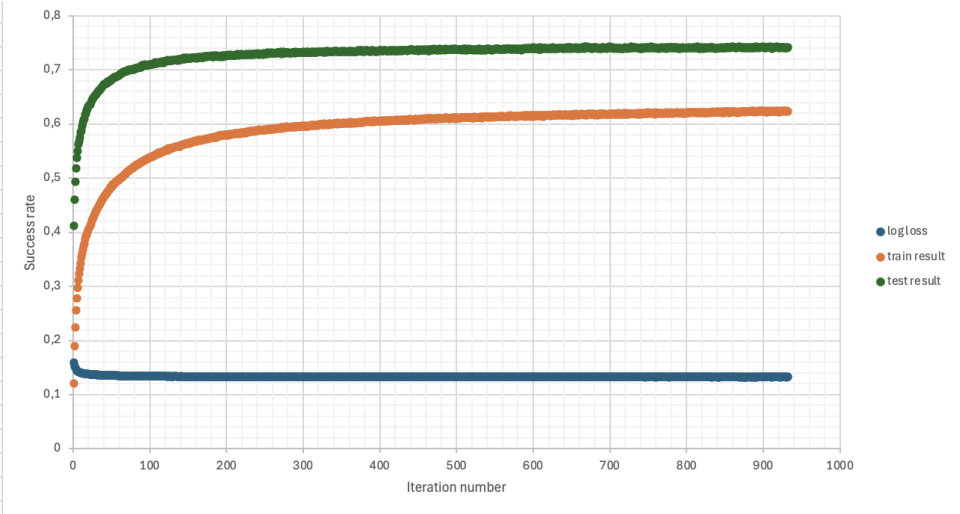

FIGURE 8 – Statistique du modèle avec l'optimiseur Adam

On peut voir sur ce graphique que les résultats de l'entrainement monte beaucoup plus rapidement au début pour se stabiliser au-dessus des 60%.

3.2.6 Sauvegarde des données

Une fois les poids mise à jour, il nous faut les sauvegarder pour pouvoir reprendre l'entraînement, mais aussi pouvoir utiliser la prédiction avec un modèle entrainer. Dans la fonction de sauvegarde de notre réseau de neurone, nous ne sauvegardons pas seulement les poids et les biais, mais aussi les hyperparametre et les statistiques de réussite à chaque époque, ce qui nous permet de créer des graphes pour analyser notre modèle.

Nous sauvegarder ces paramètres dans des fichiers CSV et dans le même format que lorsqu'on les charge dans le modèle.

Cette fonction de sauvegarder nous permet aussi de réinitialiser notre modèle. Pour ce faire, nous demandons à l'utilisateur combien de couche caché, il veut et de quelle taille il veut ses différentes couches. Cela permet de pouvoir tester plusieurs architectures de réseau différent. Au moment de la réinitialisation du modèle, nous réinitialisons les hyperparametres à leur valeur prévue par défaut. Nous réinitialisons les poids grâce à l'initialisation de He ce qui permet de commencer avec des poids qui sont bien pour la fonction d'activation ReLU ce qui optimise l'apprentissage.

3.2.7 Prédiction

Afin de pouvoir transforme les images que l'on donne à l'IA en lettre, nous avons un mode de prédiction comme mentionner plus tôt. Ce mode de prédiction utilise en entrer des fichiers CSV afin de prédire ce qu'il y a dedans.

Il y a plusieurs fichiers CSV différent :

- Un fichier pour la grille qui contient les lettres de la grille ainsi que sa taille en largeur et en longueur.

- Plusieurs fichiers comportant des lettres. Chacun de ses fichiers représente un mot à retrouver dans la grille.

Une fois les donner des CSV récupérer, le programme va les passer un par un dans la fonction de forward pour obtenir en sortie le résultat de la prédiction. Une fois cela fait, on transforme les résultats de lettre puis on crée deux fichiers. Dans le premier, on y met la grille reconstituée et dans le deuxième, on met les mots à retrouver. Ce qui permet au solver de facilement les récupérer.

3.2.8 Configuration du modèle

Après avoir appliqué toutes les fonctions et les optimisations citées précédemment et s'être assuré que notre code marchait, nous avons voulu tester plein d'architecture différente pour notre réseau. C'est pour cela qu'au lieu de rajouter des couches manuellement dans le code, en ajouter des boucles à nos fonctions, nous avons décidé de faire un réseau de neurone qui s'adapte tout seul en fonction des paramètres mis en entrer.

Une fois le code pour l'adaptation finalisé, nous arrivions au-dessus des 70% sur le train set. Pour augmenter se chiffre, il nous suffisait de relance l'entraînement en boucle avec des architectures différentes.

La première que nous ayons testée est une architecture avec 2 couche cachées de 512 et 256 neurones.

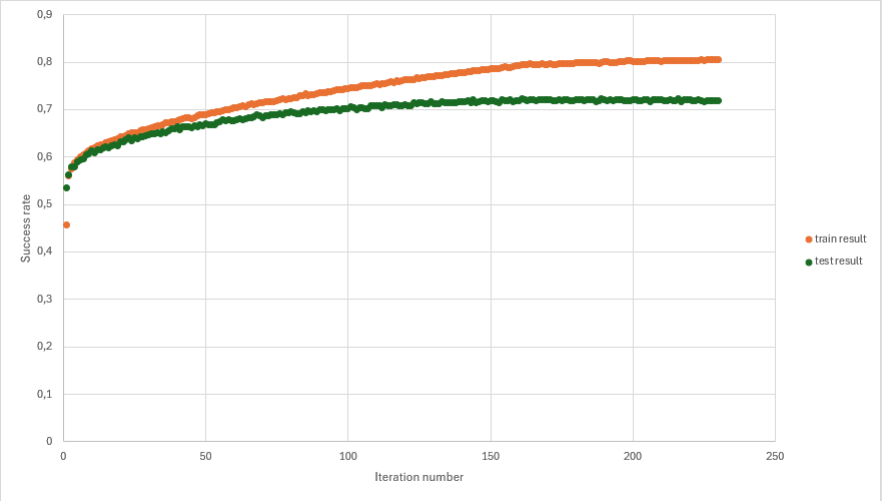

FIGURE 9 – Statistique du modèle avec 2 couche cachées de 512 et 256 neurones

Ce graphe nous permet de voir qu'avec deux couches cacher, on peut dépasser les 80% de réussite sur le train set. Et avoir des résultats sur le test qui dépasse les 70% de réussite.

Après avoir test avec deux couches de 512 et 256 neurones, nous avons testé notre modèle avec 3 couche cachées de 512, 256 et 128 neurones en pensant que nous allions aller plus haut dans les résultats, car le modèle a plus de couche.

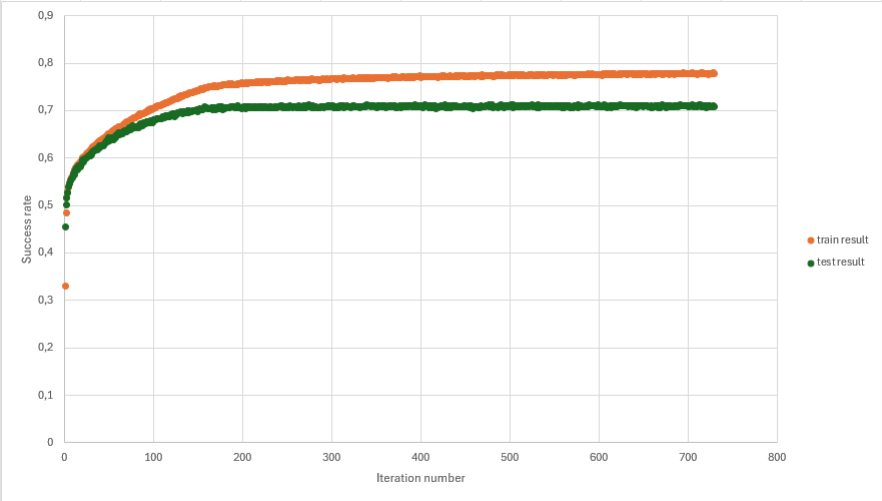

FIGURE 10 – Statistique du modèle avec 3 couche cachées de 512, 256 et 128 neurones

On peut voir sur le graphe qu'un modèle avec trois couches cachées n'est pas plus performant que celui avec deux couches cachées pour la reconnaissance de caractère. Les résultats de ce modèle sur les images d'entrainement ne dépassent pas les 80%. Dès le début de son entraine-

ment, le modèle est moins rapide. Par exemple, au bout de 100 époques, on peut voir que le modèle atteint les 70% alors que pour celui a 2 couches cachées, il atteignait les 70% au bout de 50 époques.

Maintenant que l'on sait qu'augmenter le nombre de couches caché ne sert à rien et réduit plutôt la performance du modèle pour de la reconnaissance de caractère, nous avons essayé de réduire le nombre de couches cachées. Pour ce faire, nous avons testé un modèle avec une seule couche caché de 1024 neurones.

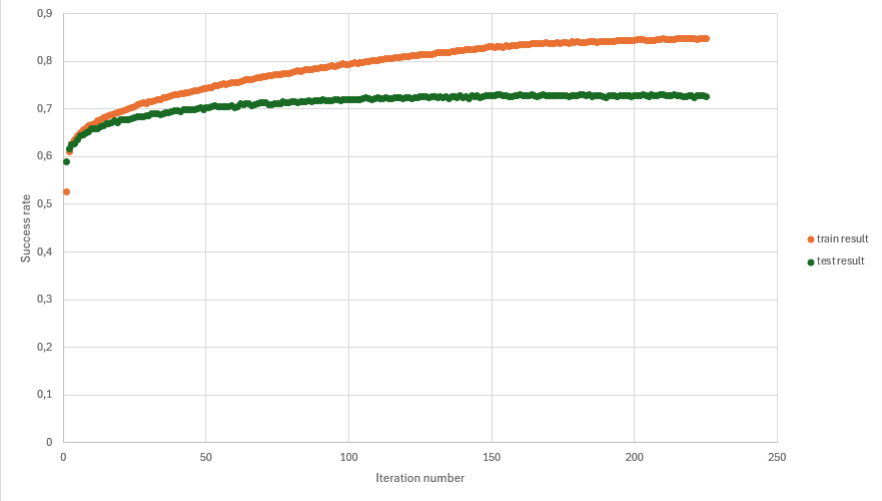

FIGURE 11 – Statistique du modèle avec une seule couche caché de 1024 neurones

On peut voir qu'avec une seule couche cachée, les résultats de l'IA dépassent largement les 80% sur les données d'entraînement. L'IA atteint les 85% de réussite qui est 5% de plus que sur des architectures avec plusieurs couches cachées. Après avoir analysé ces résultats, nous en avons conclu que pour de la reconnaissance de caractère, une seule souche cachée est le meilleur choix et donne les meilleurs résultats.

Maintenant que l'on sait que pour maximiser la réussite de notre IA, nous avons besoin d'une seule couche cachée, il faut déterminer combien de neurones, il faut sur cette couche cachée pour avoir les meilleurs résultats. Nous avons commencé par essayer avec beaucoup de neurone sur la couche cachée. Voici les statistiques d'une architecture que l'on a testée contenant 4096 neurones sur sa couche cachée.

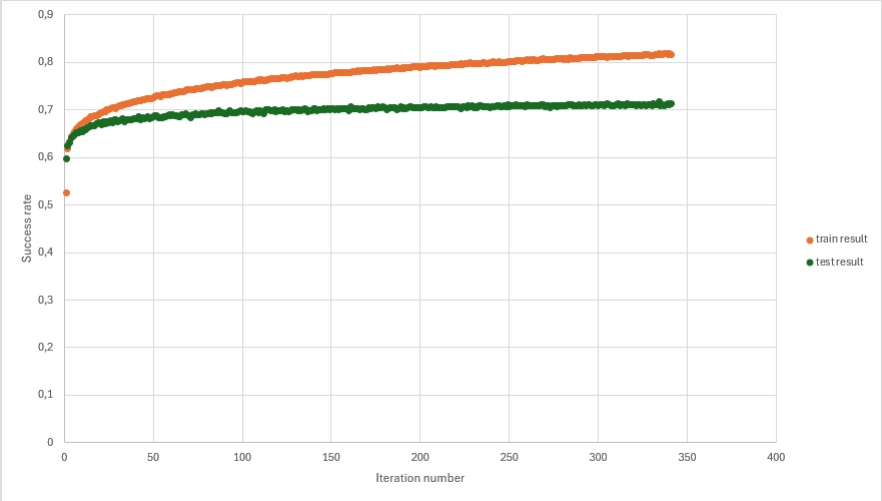

FIGURE 12 – Statistique du modèle avec une seule couche caché de 4096 neurones

On peut voir sur ce graphe que contrairement à une seule couche cacher de 1024 neurones, l'augmentation de la réussite est bien plus lente et n'arrive qu'à 82% de réussite. Cela montre qu'en plus de mettre beaucoup plus de temps à être entrainé, mettre trop de neurone dans la couche cachée affecte aussi négativement les résultats du modèle.

À la suite de cela, nous avons continué à chercher le nombre optimal de neurones pour cette couche cachée. Nous avons vu qu'augmenter trop ne sert à rien, nous avons donc essayé une modèle avec 2048 neurones sur la couche cachée. Voici les statistiques obtenues à la suite de nombreuses époques avec ses 2048 neurones sur la couche cachée.

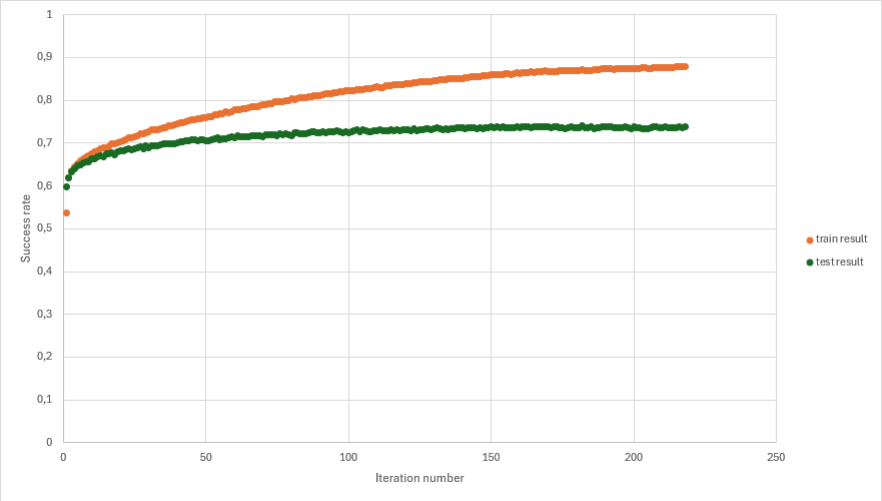

FIGURE 13 – Statistique du modèle avec une seule couche caché de 2048 neurones

On peut voir que ce modèle semble être le meilleur, il arrive presque à atteindre les 90% de réussite sur les données d'entraînement. Et si

nous l'entrainions plus longtemps, il pourrait dépasser les 90% de réussite.

Mais comme on peut le voir sur les différentes statistiques des différentes architectures, seul les donner d'entraînement change réellement d'une architecture à l'autre. Si nous regarder la courbe des données de test, on peut voir que sur aucun des graphes, elle ne dépasse les 75% de réussite. Sur tous les graphes, elle tourne autour les 73% de réussite sans jamais aller plus loin.

Nous avons fait plusieurs tests suite à cela en changeant la valeur des hyperparamètre telle que le dropout permettant d'éviter le surapprentissage, mais cela n'a eu qu'infime répercussion sur les résultats des données de test.

3.2.9 Optimisation

Effectuer les nombreux tests nécessaires pour configurer et optimiser le modèle demande une puissance de calcul considérable, ce qui rallonge significativement le temps requis pour parvenir à des résultats concluants. Pour pallier ce problème et rendre le processus d'entraînement plus efficace, nous avons exploré diverses stratégies d'optimisation, visant à réduire le temps nécessaire pour chaque itération tout en maintenant, voire améliorant, les performances de notre réseau.

L'une des solutions que nous avons mises en œuvre est l'utilisation du multithreading via la bibliothèque pthread. Cette approche nous a permis de répartir certaines parties intensives du calcul, notamment les opérations matricielles, entre plusieurs threads. Les calculs matriciels étant omniprésents dans l'entraînement de l'IA, cette parallélisation a eu un impact significatif sur le temps global d'exécution. Grâce à cette technique, nous avons pu exploiter au maximum les capacités des processeurs multicœurs, réduisant ainsi les goulets d'étranglement liés aux calculs séquentiels. Cette optimisation s'est traduite par une accélération notable du processus, rendant chaque époque d'entraînement bien plus rapide.

Par ailleurs, nous avons également optimisé le programme au niveau de la compilation en utilisant le flag -O3, une option d'optimisation avancée offerte par le compilateur. Ce flag permet au compilateur de produire un code machine plus performant en appliquant des optimisations agressives, telles que l'inlining des fonctions, l'élimination des redondances dans les boucles et l'amélioration de la gestion des registres. Ces optimisations ont considérablement réduit la latence d'exécution du programme, en maximisant son efficacité sur la machine cible.

En combinant ces deux approches, le multithreading pour paralléliser les calculs et l'optimisation du code au niveau de la compilation, nous avons réussi à presque diviser par deux le temps requis pour effectuer une époque complète d'entraînement. Cela représente un gain de temps significatif, nous permettant d'effectuer davantage de tests et d'itérations dans des délais plus raisonnables. Ces améliorations ont non seulement accéléré notre processus de développement, mais elles ont également démontré l'importance d'optimiser à la fois le matériel et le logiciel pour relever les défis liés à l'apprentissage profond. Ces méthodes d'optimisation constituent une base solide pour les développements futurs et pourraient être étendues à d'autres parties du projet nécessitant des calculs intensifs.

3.2.10 Problèmes rencontrés

Comme évoqué précédemment, l'architecture du réseau de neurones joue un rôle crucial dans l'amélioration de ses performances, en particulier sur les données d'entraînement. Une architecture bien pensée permet d'optimiser l'apprentissage et d'obtenir des résultats plus précis sur l'ensemble des données vues pendant l'entraînement. Cependant, dans le cas de notre réseau, les changements d'architecture n'ont pas eu d'impact significatif sur les performances des données de test, qui restent un indicateur clé de la capacité du modèle à généraliser. Même après avoir ajusté divers hyperparamètres, tels que le taux d'apprentissage, le nombre de neurones par couche ou encore la régularisation, le pourcentage de réussite sur les données de test n'a augmenté que très légèrement, indiquant une certaine limite de notre approche actuelle.

Malgré la mise en place de techniques visant à réduire le surapprentissage, comme l'ajout de dropout pour désactiver aléatoirement certains neurones pendant l'entraînement et l'utilisation de la régularisation L2 pour limiter la complexité des poids, notre modèle semble continuer à surapprendre sur les données d'entraînement. Ce phénomène est visible par l'écart notable entre les performances sur l'ensemble d'entraînement, qui atteignent des valeurs élevées, et celles sur les données de test, qui stagnent. Cela suggère que notre réseau mémorise trop les données d'entraînement, au lieu d'apprendre des patterns généralisables.

Face à cette problématique, nous avons exploré plusieurs pistes pour améliorer la robustesse et la capacité de généralisation du modèle. Nous avons testé différentes modifications sur notre architecture existante, mais sans grand succès. En parallèle, nous avons envisagé de recourir à des modèles d'intelligence artificielle plus avancés, comme les réseaux de neurones convolutionnels (CNN), qui sont spécifiquement conçus pour les tâches de traitement d'image. Ces modèles, en exploitant des couches

convolutionnelles et des techniques avancées de réduction de dimensionnalité, permettent de mieux capturer les caractéristiques visuelles des données. Cependant, leur intégration dans notre projet nécessitait une refonte complète de l'architecture actuelle, ce qui s'est avéré complexe à mettre en œuvre dans les délais impartis.

Ainsi, bien que des efforts significatifs aient été déployés pour améliorer notre modèle, nous avons atteint une limite avec notre approche actuelle. Cela met en évidence l'importance d'un choix architectural initial pertinent et des ajustements appropriés pour maximiser les performances tout en garantissant une bonne capacité de généralisation. Ces réflexions serviront de base pour d'éventuelles améliorations futures, notamment en adoptant des architectures spécialisées ou en explorant des approches hybrides combinant plusieurs types de réseaux.

3.2.11 CNN

Pour résoudre les problèmes liés à notre tâche de classification, nous avons décidé de mettre en place une architecture totalement différente et adaptée : un réseau de neurones convolutionnels, ou CNN (Convolutional Neural Network). Les CNN sont largement utilisés pour le traitement des images, car ils permettent d'extraire automatiquement les caractéristiques importantes des données visuelles et de les utiliser pour effectuer des prédictions.

Un CNN repose sur un enchaînement de couches spécifiques qui transforment progressivement une image brute en un ensemble de caractéristiques utilisables pour la classification. Contrairement aux réseaux de neurones classiques qui traitent les images sous forme de vecteurs, les CNN exploitent directement la structure 2D des images pour préserver les relations spatiales entre les pixels.

Étape 1 : Application des filtres (convolution) :

Un CNN commence par appliquer des filtres ou kernels à l'image d'entrée. Ces filtres permettent de détecter des motifs spécifiques, comme des bords, des textures ou des zones contrastées. L'image est "balayée" par le filtre grâce à une opération mathématique appelée convolution, qui produit une nouvelle image appelée carte de caractéristiques.

Voici un schéma illustrant une convolution avec un filtre de 3x3 :

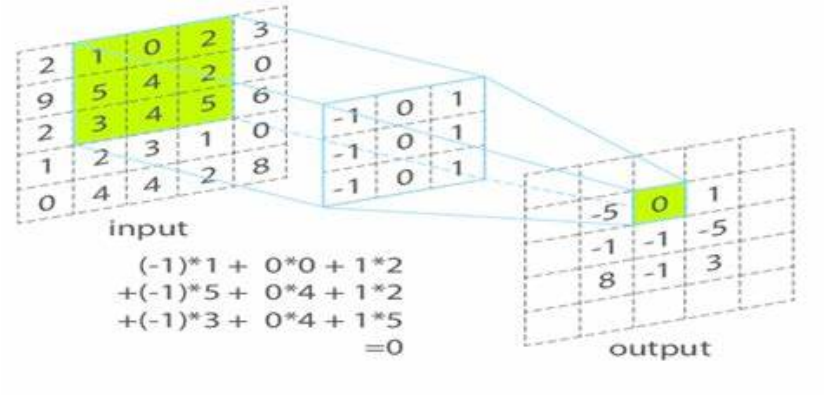

FIGURE 14 – Schéma illustrant une convolution

Les filtres sont appris par le modèle pendant l'entraînement, ce qui permet au réseau de détecter les motifs les plus pertinents pour la tâche. Par exemple, dans un premier niveau, un filtre pourrait détecter des lignes horizontales, tandis que dans des niveaux plus profonds, il pourrait détecter des formes plus complexes, comme des yeux ou des lettres.

Étape 2 : Réduction des dimensions (pooling) :

Après la convolution, le réseau applique une autre opération appelée pooling. Cette étape consiste à réduire la taille des cartes de caractéristiques tout en conservant les informations les plus importantes. Le pooling aide à diminuer le nombre de paramètres du réseau, ce qui réduit les risques de surapprentissage tout en accélérant l'entraînement.

Il existe plusieurs types de pooling, les plus courants étant :

- Max pooling : Ne conserve que la valeur maximale dans une région donnée, mettant en avant les caractéristiques les plus fortes.

- Average pooling : Calcule la moyenne des valeurs dans une région donnée, produisant une version plus "douce" de la carte de caractéristiques.

Voici Une illustration montrant une opération de max pooling et d'average pooling sur une carte de caractéristiques.

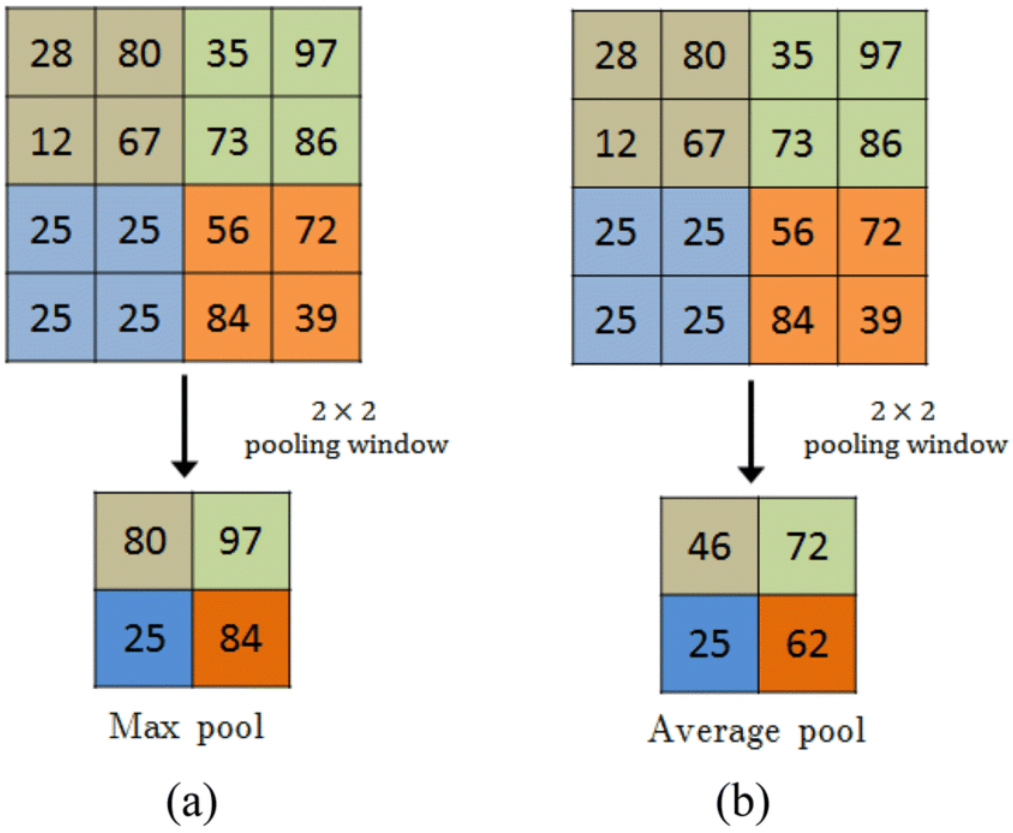

FIGURE 15 – Illustration montrant une opération de max pooling et d'average pooling sur une carte de caractéristiques

On peut voir sur cette illustration que pour la (a) on prend le maximum de chaque carré de couleur alors que pour la (b) on prend la moyenne de chaque carré de couleur.

Étape 3 : Passage au réseau de neurones classique

Une fois les couches de convolution et de pooling appliquées, l'image initiale a été transformée en un ensemble compact de caractéristiques essentielles. Ces caractéristiques sont ensuite transmises à une ou plusieurs couches entièrement connectées (comme dans un réseau de neurones classique) pour effectuer la classification.

Voici à quoi ressemble au final un CNN :

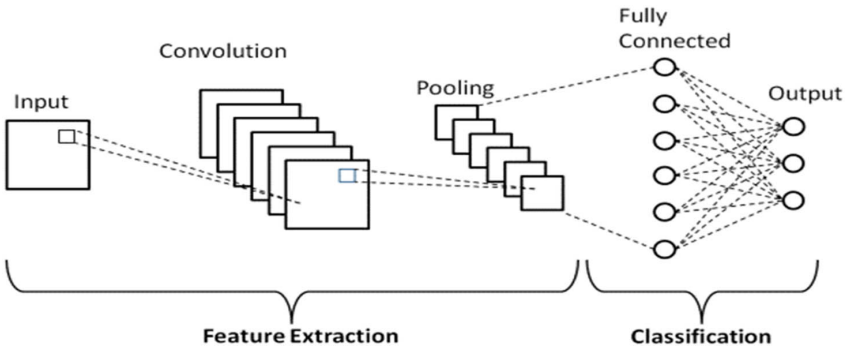

FIGURE 16 – Illustration des étapes d'un CNN

Pour le projet, nous avons essayé d'utiliser l'architecture LeNet-5, l'un des premiers CNN introduits par Yann LeCun en 1998. LeNet-5 a été conçu à l'origine pour la reconnaissance de caractères manuscrits (comme les chiffres des chèques bancaires). Il se compose de plusieurs couches de convolution, de pooling, et de couches connectées, et est connu pour sa simplicité et son efficacité sur des images de petites dimensions (par exemple, 32x32 pixels).

Malheureusement après de nombreuse heure à implémenter cette architecture, elle ne marchait pas. Nous avons revu l'entièreté de notre code sans aucun succès. Nous avons donc abandonné cette idée pour revenir à ce que nous avions déjà.

3.2.12 Conclusion

Comme vous avez pu le voir, nous avons essayé énormément d'architecture et de modification avant d'arriver à notre réseau de neurone actuelle. Même si nous n'avons pas réussi à mettre en place un CNN, nous sommes très fiers de notre réseau final, malgré le fait que ses performances sur de nouvelle image sur lequel il ne s'est pas entrainé ne dépasse pas les 75% de réussite.

Notre réseau final se compose d'une couche cachée de 2048 neurones qui comme nous l'avons vu plus tôt est l'architecture qui nous donne les meilleurs résultats. Notre réseau final comporte toutes les améliorations que nous avons vues plus tôt :

- Un mélange des données au début de l'entrainement pour éviter l'apprentissage des réponses.

- Une propagation avant utilisant la fonction d'activation ReLU pour les premières couches et la fonction softmax pour la dernière. Et avec un dropout pour éviter le surapprentissage.

- Une fonction de propagation arrière utilisant la dériver de la fonction d'activation ReLU.

- Une fonction de mise à jour des poids et des biais grâce à Adam pour améliorer la vitesse d'apprentissage du modèle et a la régulation L2 pour éviter encore une fois le surapprentissage.

- Une fonction de sauvegarde qui permet de reprendre l'entrainement et de pouvoir analyser l'apprentissage du modèle.

Avec tout ça, notre IA finale est arrivée jusqu'au 91% de réussite sur les données d'entrainement et 74% de réussite sur celle de test.

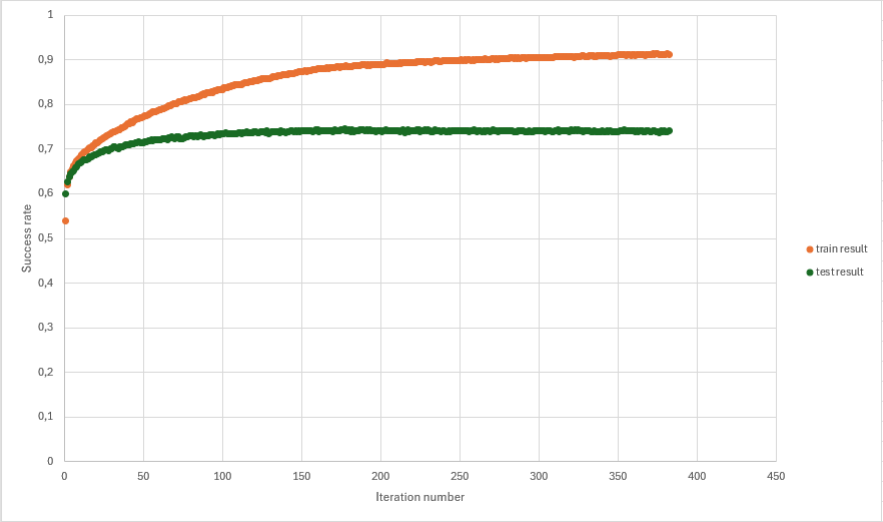

FIGURE 17 – Statistique du modèle final de notre réseau de neurone

4 Other

Cette dernière section explique toutes les fonctions utilisées après l'application de l'AI aux différentes lettres. Elle qui contient le solver ainsi que l'UI qui regroupe tous les codes.

4.1 Solver

Le solver est le programme permettant de résoudre la grille de mots cachés. Il prend en entrer un fichier contenant une grille de lettre qui correspond à la grille dans laquelle l'on doit retrouver les mots. Cette fonction prend deux paramètres : le premier est celui du fichier de la grille et le deuxième est le mot à retrouver dans la grille.

Afin de trouver le mot à chercher, on récupère les donner du fichier contenant dans la grille et on le stocke dans un tableau. Pour chaque

lettre du tableau, on regarde si elle correspond à la première lettre du mot à chercher et si c'est le cas, on prend le nombre de lettres qu'il reste dans chaque direction et on le compare à la longueur du mot pour voir s'il peut être dans cette direction. Puis pour chaque direction valide, on regarde la deuxième puis troisième lettre... et on les compare aux lettres du mot, s'il y a une différence, on arrête, sinon ont continu. Si on trouve le mot, on renvoie la position de départ ainsi que celle d'arriver, sinon on renvoie "Not Found".

Pour d'optimiser la recherche du mot, nous parcourons la grille dans les deux sens en même temps quand on cherche dans une direction en vérifiant le mot à l'envers. Cela nous permet de ne parcourir que la moitié de la grille en lettre par lettre.

4.2 UI

Comme dit plus tôt, l'UI est la partie visuelle de l'application, mais c'est également cette partie qui va appeler les différents codes un par un. L'application a visuellement été créée avec le logiciel Glade.

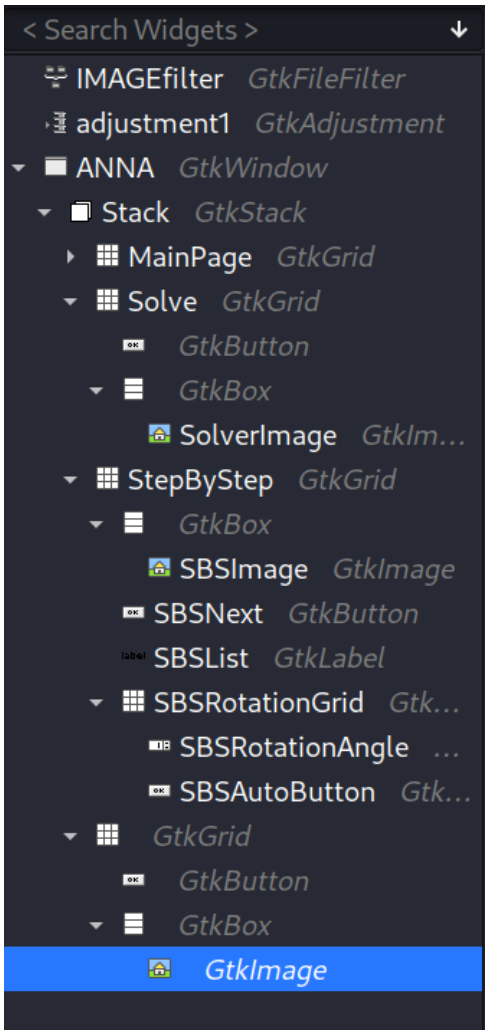

FIGURE 18 – Architecture de l'application sur Glade

Le programme est structuré en plusieurs étapes et fonctionnalités pour offrir une expérience utilisateur fluide et intuitive. Tout d'abord, l'utilisateur doit choisir une image à analyser. Ensuite, il peut décider d'activer ou non le mode pas à pas. Ce mode pas à pas est une fonctionnalité pédagogique qui permet de visualiser chaque étape intermédiaire du traitement de l'image, comme la conversion en niveaux de gris (GrayScale), la binarisation, ou encore d'autres transformations. Si ce mode n'est pas activé, le programme génère directement le résultat final sans afficher les étapes intermédiaires, offrant ainsi un accès rapide à la solution.

Pour structurer les différentes pages de notre application, nous avons intégré une GtkStack, un widget idéal pour organiser et basculer entre différentes vues, comme la page de lancement, l'interface du mode pas à pas ou encore la section dédiée à la sauvegarde des résultats. Cette architecture modulaire garantit une navigation simple et logique à tra-

vers l'application.

En complément, pour organiser les widgets dans chaque page, nous avons utilisé des GtkGrid, qui offrent une flexibilité optimale pour aligner et agencer les éléments. De plus, pour améliorer la présentation et ajouter du style à nos interfaces, nous avons eu recours au markup, un langage inspiré du HTML, permettant d'appliquer des mises en forme avancées, comme du texte stylisé ou des éléments accentués.

Un autre défi que nous avons adressé est la gestion de la taille des images. Afin d'assurer une expérience visuelle cohérente et d'éviter des problèmes liés à des images trop grandes, nous avons mis en place une contrainte : l'image affichée dans l'application est automatiquement redimensionnée si elle dépasse 500 pixels en largeur ou en hauteur. Cela garantit que toutes les images s'adaptent harmonieusement à l'interface, sans compromettre leur lisibilité ni leur qualité.

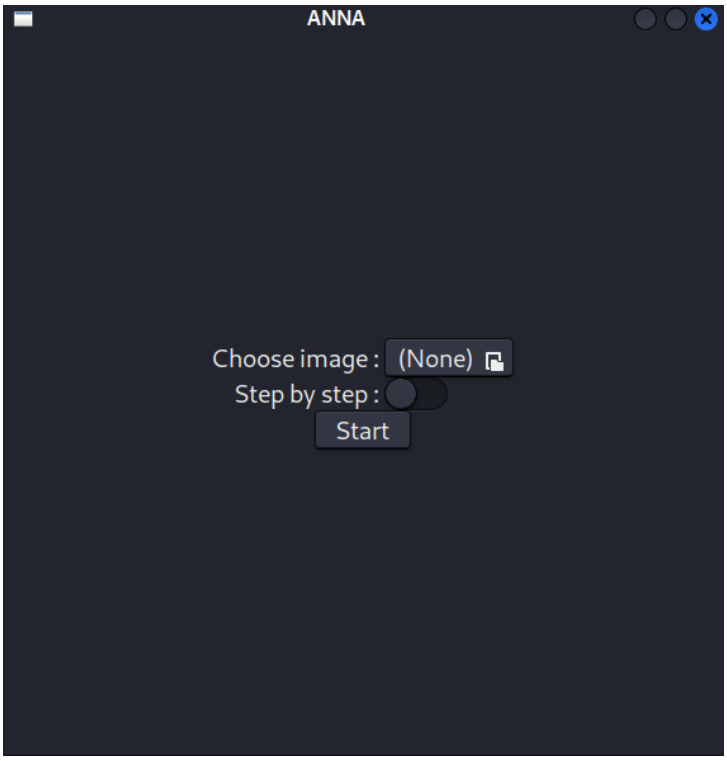

FIGURE 19 – Page de lancement

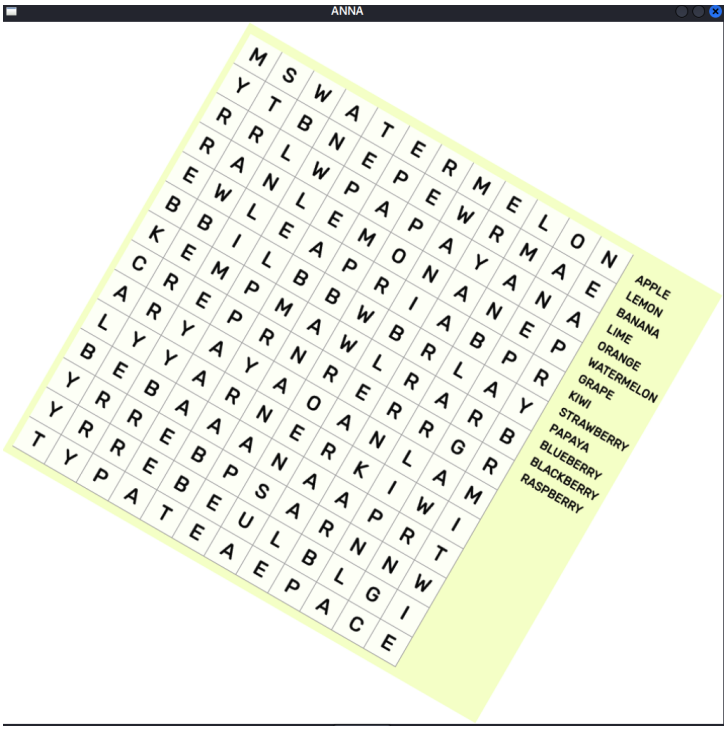

FIGURE 20 – Mode classique

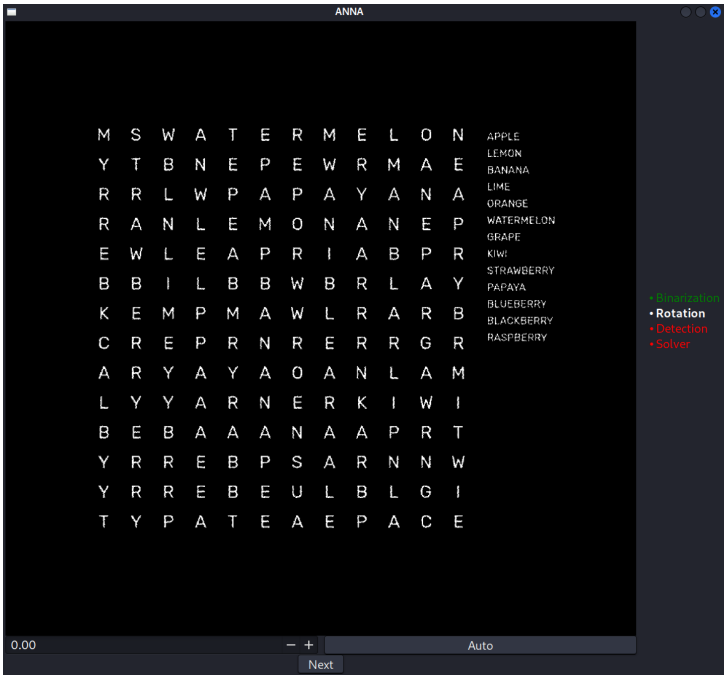

FIGURE 21 – Mode pas à pas Pendant la rotation

Ensuite, nous avons exploité les fonctionnalités offertes par SDL pour dessiner un trait rouge semi-transparent afin de surligner les mots identifiés dans la grille. Cette méthode permet non seulement de mettre

en évidence les mots trouvés de manière visuellement claire, mais aussi de conserver une certaine transparence pour ne pas masquer complètement les lettres sous le surlignage. Par exemple, comme illustré dans l'image ci-dessous, nous avons utilisé cette technique pour surligner le mot "water", en rendant immédiatement visible son emplacement tout en laissant le reste de la grille lisible. Cette approche améliore l'expérience utilisateur et facilite la validation des mots identifiés.

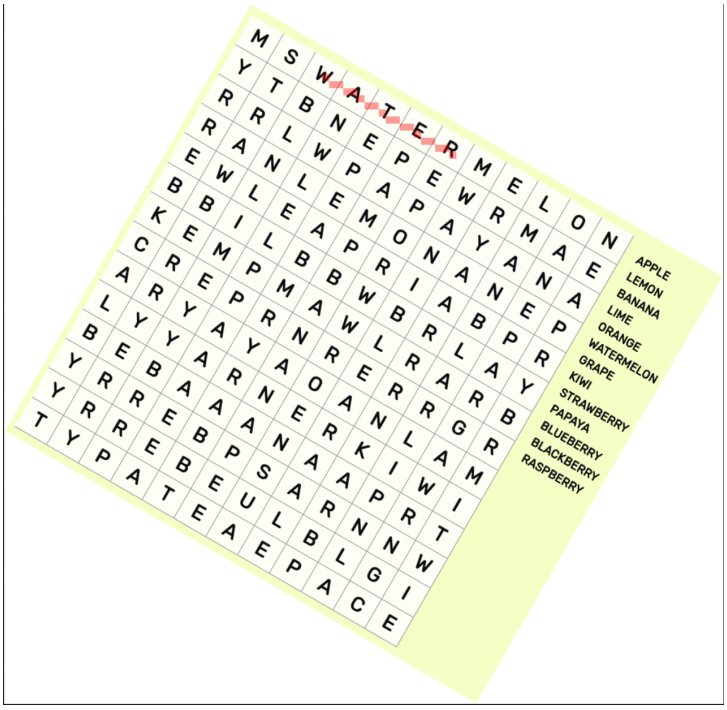

FIGURE 22 – Exemple de dessin sur le mot "water"

5 Conclusion

Pour conclure, nous sommes globalement très satisfaits des progrès réalisés dans le cadre de cette soutenance finale. Bien que le projet ait présenté plusieurs défis, notamment lors de l'intégration de certaines optimisations avancées et de l'implémentation d'un réseau de neurones convolutif (CNN), nous avons su surmonter ces obstacles grâce à une approche méthodique et collaborative, ce qui nous a permis de livrer un projet abouti et fonctionnel.

Nous sommes particulièrement fiers des performances obtenues par notre réseau de neurones. Après de nombreuses itérations, tests et ajustements minutieux, il atteint un taux de réussite remarquable de 90% sur les données d'entraînement et 74% sur les données de test. Ces résultats témoignent de nos efforts continus pour affiner la structure du réseau et optimiser les hyperparamètres tels que le dropout et la régula-

risation L2, deux éléments clés qui ont permis d'améliorer la robustesse et la généralisation de notre modèle.

Cependant, malgré ces succès, certains aspects du projet ont été impactés par le temps investi dans les optimisations de l'IA. Par exemple, l'intégration de certaines fonctionnalités secondaires a pris du retard, et le cropping (processus de détection et de séparation des lettres et des mots) bien que significativement amélioré, nécessite encore des ajustements pour gérer efficacement tous les cas complexes. Cela constitue une piste importante pour d'éventuelles améliorations futures.

Cette expérience nous a également permis de tirer de précieuses leçons, notamment sur la gestion du temps et la répartition des tâches au sein de l'équipe. Nous sommes fiers du chemin parcouru et des solutions que nous avons mises en place pour surmonter les imprévus. Avec un peu plus de temps, nous aurions sans doute atteint tous les objectifs initiaux que nous nous étions fixés pour ce projet.

En somme, ce projet a été une opportunité enrichissante de mettre en pratique nos connaissances, de repousser nos limites techniques, et de développer des compétences en gestion de projet, autant d'acquis qui nous seront précieux pour nos futurs travaux.