

O.C.R.
Rapport de projet

— **Groupe SoyCent** —

Nicolas FROGER
Corentin PAPE
Enzo BOURICHE
Thomas BERLIOZ

Décembre 2019

Table des matières

1	Introduction	4
2	Présentation du groupe	5
3	Répartition des tâches	7
4	Explications des fonctionnalités	8
4.1	Bibliothèque de matrices	8
4.1.1	Structure et initialisation d'une matrice	8
4.1.2	Opérations sur et entre les matrices	8
4.1.3	Fonctions de calculs sur les matrices	9
4.1.4	Fonctions annexes sur les matrices	9
4.2	Traitement de l'image d'entrée	10
4.2.1	Du fichier...	10
4.2.2	...à la matrice	10
4.2.3	Recherche sur la rotation et l'épuration du texte	12
4.2.4	Résultat final	17
4.3	Segmentation du document	18
4.3.1	Principe de la segmentation	18
4.3.2	Calcul des espaces entre les mots	19
4.3.3	Stocker le résultat de la segmentation	20
4.3.4	Généralisation des matrices de caractère	21
4.4	Réseau de neurones	22
4.4.1	Architecture du réseau de neurones - Cas du XOR	22
4.4.2	Architecture générale du réseau de neurones	25
4.4.3	Initialisation du réseau de neurones	26

4.4.4	Calcul des valeurs d'activation	28
4.4.5	Fonction d'entraînement : la descente de gradient	29
4.4.6	Mise à jour du réseau : taux d'apprentissage adaptatif	30
4.4.7	Paramétrage du réseau	32
4.4.8	Entraînement d'un réseau déjà entraîné	33
4.4.9	Réseau de neurones - Conclusion	34
4.5	Interface utilisateur	35
4.6	Chaînage des éléments du projet	37
5	Conclusion	38

1 Introduction

Notre troisième semestre à EPITA a été marqué par le projet d'informatique en groupe ayant pour but de reconnaître des caractères sur une image. Celui-ci est un projet d'ampleur importante car il mobilise un certain nombre de nouvelles compétences ainsi que de nouvelles connaissances dans différents domaines tels que le traitement d'images ou la structure d'un réseau de neurones.

La fin du semestre marque également la fin de ce projet après la première soutenance ayant eu lieu en octobre.

Ce rapport de projet présente les différentes fonctionnalités implémentées dans le projet ainsi que les explications de leurs fonctionnements. Les fonctionnalités n'ayant pas abouti à un résultat satisfaisant seront également abordées.

2 Présentation du groupe

Le groupe s'est formé très rapidement lorsque les classes ont été dévoilées en début d'année. En effet, nous nous connaissions déjà depuis l'année dernière pour trois d'entre nous. Le fait de partager la même classe a influencé ce choix car nous avons jugé qu'il serait plus judicieux en termes d'horaires de rester ensemble.

THOMAS : Au départ plus attiré par un avenir professionnel dans les mathématiques que dans l'informatique, ce domaine m'a néanmoins toujours fasciné pour cette sensation de liberté que l'on a de créer, de concevoir à partir de seuls savoir et imagination, notamment dans l'univers du jeu vidéo au départ. C'est lorsque j'ai commencé à m'intéresser à la cybersécurité que j'ai pris conscience de ce que je voulais vraiment faire toute ma vie. Depuis, j'essaie d'en apprendre un peu plus chaque jour sur le monde immense de l'informatique, de progresser toujours pour atteindre ce rêve d'avoir pour métier ma passion. C'est ainsi que je trouve particulièrement intéressant d'en savoir autant que possible sur les réseaux de neurones, dont l'utilisation est en hausse constante dans de nombreux domaines.

CORENTIN : J'ai découvert la programmation assez jeune, grâce à plusieurs membres de ma famille qui s'y intéressaient. Je m'y suis donc toujours intéressé, prenant de plus en plus de temps sur mon temps libre pour des projets personnels, le plus souvent des jeux vidéos. Entendre les adultes parler de l'informatique comme un domaine et des métiers d'avenir m'a conforté dans mon idée de l'étudier. Les projets de programmation sont en général plus un plaisir qu'un travail, et celui-ci ne fait pas exception. Il permet de découvrir et de jouer un maximum avec toutes les nouvelles notions qu'apporte la programmation en C en travaillant sur quelque chose de concret.

ENZO : Je n'ai pas toujours été passionné par la programmation mais c'est en voyant le potentiel de celui-ci dans l'univers du modding et des plugins des jeux-vidéos que je m'y suis attaché avec le temps. De ce fait, j'ai passé beaucoup de temps au lycée à programmer en vue de réaliser des mods ou des plugins pour certains jeux, notamment en Java. Les TP de l'année précédente furent un réel plaisir à travailler pour moi, chacun des sujets proposés par nos ACDC étaient intéressants, d'autant plus qu'ils me permettaient souvent d'aller un peu plus loin, ayant déjà eu un peu d'expérience avec les langages de haut niveau type C#. Cette année semble enfin proposer des notions dont je n'ai jamais vraiment entendu parler auparavant. Le langage C par exemple, avec lequel je n'avais que très peu d'expérience, commence à lever un peu le voile sur toutes les facilités que nous permettaient les langages haut niveau que l'on a vu jusqu'à présent.

NICOLAS : (chef de groupe) : J'ai toujours été passionné d'informatique. J'éprouve en général beaucoup de plaisir lorsque je travaille sur des TP ou sur des projets, que ce soit dans le cadre de l'école ou non. Le projet de SUP

s'était très bien passé l'année dernière et j'étais très satisfait du résultat final. Cependant, dès le début de cette année, je ressentais beaucoup d'appréhension vis à vis de ce nouveau projet. En effet, contrairement au précédent, je me sentais beaucoup moins capable de réussir en voyant les notions requises afin de réaliser de projet. Parmi ces notions, il y a notamment le langage C que je n'avais jamais essayé auparavant. Le sujet du projet me faisait peur, en effet, un logiciel de reconnaissance de caractères dans une image me semblait infaisable à mon niveau. Cependant, j'ai beaucoup apprécié travailler sur ce projet et je suis fier du résultat. J'ai l'impression d'avoir beaucoup appris et je suis satisfait du travail effectué.

3 Répartition des tâches

Afin d'atteindre les objectifs demandés, il a été nécessaire d'organiser une répartition des tâches au sein du groupe. Nous avons pensé qu'assigner deux membres à la réalisation du réseau de neurones était justifié en vue de la difficulté de la tâche.

Voici la répartition détaillée des tâches :

Tâche	Nicolas	Corentin	Enzo	Thomas
Bibliothèque de matrices			•	•
Traitement d'image		•		
Binarisation de l'image		•		
Segmentation du document	•			
Réseau de neurones			•	•
Interface	•			
Châinage des éléments du programme	•			

4 Explications des fonctionnalités

La section suivante détaille les fonctionnalités du programme qui sont utilisées tout le long du processus ainsi que leurs différentes implémentations.

4.1 Bibliothèque de matrices

Participants au développement :

- Enzo BOURICHE
- Thomas BERLIOZ

4.1.1 Structure et initialisation d'une matrice

Comme plusieurs des sections de ce document l'exprimeront, que ce soit dans le réseau de neurones ou dans le traitement des images, il existe un besoin quasi-constant d'une implémentation de matrices et d'opérations sur celles-ci. Une bibliothèque permettant de manipuler les matrices a donc été réalisée au plus tôt dans la chronologie du projet, puis complétée au fur et à mesure de celui-ci.

La base de cette implémentation est un *struct*, une structure contenant les données d'une matrice : liste (de *double*), nombre de colonnes, nombre de lignes, nombre total d'éléments (longueur de la liste). Effectivement, comme cela a été conseillé dans le document informatif du projet, nous avons fait le choix d'implémenter la matrice sous forme de liste unidimensionnelle. Nous avons aussi décidé de la stocker sous forme de *struct* car cela permet à chaque matrice de toujours avoir avec elle ses propriétés (sa longueur, etc...) afin de permettre des prototypes de fonctions plus légers, et une manipulation plus simple de celle-ci en général.

Il en découle ainsi les fonctions d'initialisation de base tel que *init_matrix* permettant de générer des matrices de valeurs aléatoires suivant une distribution de Gauss, ou de 0 dans le cas contraire. *init_ones*, qui a le même fonctionnement que *init_matrix*, mais qui renvoie une matrice remplie de 1 cette fois-ci. Enfin, *onebyone_matrix* permet la création d'une matrice d'un élément, à partir d'une unique valeur *double*.

4.1.2 Opérations sur et entre les matrices

Les opérations sur les matrices se font avec *transpose* qui renvoie la transposée d'une matrice, *copy_matrix* qui renvoie la copie d'une matrice, ainsi que *mult_matrix* qui permet de multiplier chaque terme d'une matrice par un

double. La bibliothèque se devait aussi naturellement d'être munie d'opérations entre matrices. Ainsi ont pu voir le jour les fonctions *add_matrix*, *sub_matrix* et *multiply_matrix*, dont les noms sont assez explicites. De plus, *hada_product* est une multiplication de matrices terme à terme dont le réseau de neurones a besoin dans certaines formules.

Ensuite, *add_listofmatrices*, *sub_listofmatrices* et *mult_listofmatrices* sont des fonctions qui effectuent leurs opérations respectives entre les matrices d'une liste de matrices, et elles sont nécessaires lors de la mise à jour du réseau de neurones.

4.1.3 Fonctions de calculs sur les matrices

Comme du côté du réseau de neurones, l'application de la fonction sigmoïde se faisait souvent sur des matrices, nous avons décidé qu'elle serait dans la bibliothèque. Ainsi nous avons *sigmoid* et *d_sigmoid* qui calculent respectivement $\sigma(x)$ et $\sigma'(x)$ sur des *double*. De plus, comme nous avons testé le réseau de neurones en remplaçant la sigmoïde, d'autres fonctions avec le même rôle ont été implémentées. Ainsi, nous avons ajouté *relu* (où $\text{ReLU}(x)=x$ si $x>0$, 0 sinon), *d_relu* sa dérivée, et *d_tanh* la dérivée de la tangente hyperbolique utilisée depuis la bibliothèque *math.h*.

Ensuite, évidemment, *sigmoid_matrix* et *d_sigmoid_matrix* qui appliquent leurs fonctions respectives sur chacun des éléments d'une matrice, ainsi que *relu_matrix*, *d_relu_matrix*, *tanh_matrix* et *d_tanh_matrix*.

4.1.4 Fonctions annexes sur les matrices

Afin de pouvoir suivre le comportement du réseau de neurones, nous devons afficher les matrices qui le constituaient. Ainsi *print_matrix* affiche une matrice et *print_matrices* affiche une liste de matrices. De plus, nous devons afficher la lettre que le réseau analysait, donc nous avons ajouté *print_letter*.

A noter que chacune de ces fonctions renvoient un nouveau pointeur vers une nouvelle matrice, elles ne modifient pas les matrices passées en argument. En conséquences, il a fallu implémenter de nouvelles fonctions pour libérer la mémoire allouée après chaque calcul. *free_matrices* libère une liste de matrice entière puis la liste elle-même et *free_matrices_range* libère une liste de matrices entre deux indices puis la liste elle-même.

4.2 Traitement de l'image d'entrée

Participants au développement :
— Corentin PAPE

Le premier chaînon du programme est le chargement de l'image et les pré-traitements pour la rendre exploitable. L'objectif est de pouvoir accepter en entrée des images scannées par l'utilisateur, ou même des photos de texte. Nous discuterons en fin de partie sur les résultats obtenus par rapport à ces attentes.

4.2.1 Du fichier...

La première étape est évidemment l'ouverture du fichier contenant l'image. Pour nous assurer un maximum de formats de fichier compatibles, nous avons utilisé les bibliothèques SDL et SDL Image en version 1.2. Ce choix était aussi guidé par notre volonté de rester en terrain connu, les ASM ayant présenté cette bibliothèque lors d'un TP. Ainsi nous avons repris les mêmes fonctions que lors de ce TP pour le chargement de l'image et l'extraction des pixels de l'image. Parmi ces fonctions, certaines ont servi à trouver les bugs et à visualiser le comportement des algorithmes, avec l'affichage des différentes étapes des traitements.

Ainsi, cette première étape charge le fichier, le convertit en nuances de gris grâce à des opérations sur chaque pixel et renvoie un tableau contenant ces nuances de gris.

4.2.2 ...à la matrice

Dans la deuxième étape, il s'agit de transformer l'image en nuances de gris en image en noir et blanc. Pour cela, nous faisons deux traitements pour conserver uniquement les éléments qui nous intéressent, à savoir le texte, et donc à exclure toute forme de bruit numérique potentiel.

Cela se fait en deux temps : l'augmentation du contraste de l'image pour que le texte ressorte sur le fond et l'élimination du bruit numérique, qu'il ne faut évidemment pas faire ressortir lors du premier traitement.

Bien que des algorithmes existent et soient efficaces pour ces problèmes, notamment l'algorithme d'Otsu pour l'augmentation du contraste, nous avons décidé de faire nos recherches sans utiliser vraiment Internet. Nous avons donc beaucoup expérimenté avant d'obtenir un résultat satisfaisant.

L'idée de calculer la luminosité des pixels voisins nous est tout de suite venue à l'esprit quand nous cherchions à augmenter le contraste de l'image. Cepen-

dant, programmer la fonction qui va suivre nous a demandé beaucoup de temps, et nous n'avons pas pensé immédiatement à utiliser la luminosité moyenne. Ces données sont utilisées dans la fonction qui suit, à savoir le traitement à proprement parler de l'image.

bw_denoise : L'image, maintenant contenue dans le tableau *grayTab*, se voit modifiée pixel par pixel pour obtenir un résultat en noir et blanc, avec le plus de texte original restitué. C'est la partie qui nous a pris le plus de temps pour la binarisation de l'image. Voici les différentes expériences menées pour arriver au résultat retenu.

Nous avons commencé par appliquer le même traitement à tous les pixels d'une nuance de gris supérieure à 122, c'est-à-dire à la moitié entre complètement blanc (codé 255) et parfaitement noir (codé 0). En effet, lors de la binarisation, les pixels plus lumineux que 122 étaient classés comme « blancs » dans la matrice de sortie, et les autres, « noirs ». Ce traitement vise donc à « permettre » certains pixels pas assez sombres d'être reconnus comme noirs. A ce moment, nous n'utilisions pas encore la moyenne précédemment présentée. Les pixels au-delà de ce seuil était rendus plus sombres si leurs voisins étaient très clairs. Cela se résume à la formule :

$$nuanceGrise = nuanceGrise - \frac{nuanceTotaleDesVoisins}{8}$$

Les résultats obtenus étaient très mitigés, car cela créait des regroupements au niveau des lettres et renforçait les pixels isolés, comme le « grain » d'une photo. Nous avons alors pensé à différencier le cas de ces pixels isolés et du texte trop clair. Le premier essai ne fut pas très concluant, car nous n'arrivions pas à trouver une formule différenciant les pixels sur les diagonales ou les extrémités des lettres. Ceux-ci ne se distinguent malheureusement pas des pixels isolés indésirables avec les données à notre disposition. Pour mieux visualiser ces traitements, nous avons utilisé certaines formules.

La première, appliquée aux pixels d'une nuance de gris plus grande que 122, vise à les rendre plus sombre s'ils ont des voisins très clair. C'est en fait la même que précédemment. Cependant, contrairement à la précédente expérience, une deuxième formule est appliquée aux pixels ayant une luminosité supérieure à 80, pour exclure les pixels très foncés du traitement. Pour lutter contre ces petits pixels isolés, nous calculons la moyenne entre la nuance de gris du pixel et celle de ses voisins, augmentant ainsi potentiellement la luminosité du dit-pixel. On a ainsi

$$nuanceGrise = \frac{(nuanceGrise \times 4 + nuanceTotaleDesVoisins)}{8}$$

Cette fois-ci, les résultats étaient exploitables. Les pixels isolés étaient souvent éliminés et les mots écrits dans des couleurs plus claires ressortaient un peu mieux. Néanmoins, cela n'était pas encore suffisamment satisfaisant dans des conditions de contrastes difficiles, notamment pour les images plus sombres, alors nous avons continué d'expérimenter.

Au lieu de faire deux traitements successifs, l'un pouvant remettre en cause le traitement effectué par l'autre, nous avons décidé de nous concentrer sur le renforcement du contraste entre le texte et le fond, quitte à laisser de côté la réduction du bruit, des « artefacts » de l'image.

A la place des deux traitements, nous faisons un unique traitement appliqué sur des pixels sélectionnés plus minutieusement. Pour ce faire, nous avons utilisé trois conditions, déterminant le type de pixel que nous voulions, ou pas. La première garde seulement les pixels ayant une luminosité supérieure à 122, donc les pixels trop lumineux pour être considérés noirs. La deuxième vise à ne pas assombrir les pixels trop lumineux pour éviter de créer du « bruit ». Enfin, la troisième condition permet d'exclure les pixels un peu sombres situés aux bords de pixels sombres, qu'on peut espérer faire partie d'une lettre. Elle permet de dessiner plus précisément ces caractères. Enfin, si toutes ces conditions sont respectées, la formule suivante est appliquée au pixel :

$$\text{nuanceGrise} = \text{nuanceGrise} - \text{différenceDeLuminosité}$$

avec

$$\text{différenceDeLuminosité} = \left| \text{nuanceGrise} - \frac{\text{nuanceTotaleDesVoisins}}{4} \right| \times 2$$

Ainsi, nous obtenons un algorithme offrant un traitement ne gâchant pas la qualité de l'image, du moins pas la précision des caractères, tout en amplifiant dans la plupart des cas les contrastes. Cependant, il subsistait un cas qui ne fonctionnait pas assez bien. En effet, lorsque nous avons une image avec un fond sombre, utiliser comme valeur de référence 255 pour le fond dans nos calculs est très imprécis et surtout faux. C'est pourquoi nous avons introduit une variable moyenne des nuances de gris des pixels de l'image.

Ainsi, en utilisant cette moyenne à la place de 255 comme valeur de référence à chaque fois, nous avons conclu nos expériences par l'algorithme actuellement utilisé dans notre programme.

4.2.3 Recherche sur la rotation et l'épuration du texte

Les explications qui vont suivre ont pour sujet des algorithmes ayant présenté un certain travail de recherche mais qui n'ayant pas fonctionné à temps, soit pour cause de bugs, soit pour cause d'une efficacité peu convaincante.

Pour commencer, les résultats obtenus par les algorithmes d'augmentation de contraste et d'élimination du bruit n'étaient toujours pas suffisants, par conséquent nous avons voulu les améliorer avant la soutenance finale.

Nous avons débuté par rendre itératifs, c'est-à-dire répéter plusieurs fois, nos deux traitements déjà existants. Leur impact individuel est ainsi réduit mais ils se répètent des centaines de fois. Cela permet de recalculer les variables comme la luminosité moyenne de l'image et les informations sur les voisins de chaque pixels à chaque itération. Il en résulte un traitement très lourd et long sur l'image, produisant un effet inattendu.

En effet, l'image obtenue en passant par ce traitement est complètement inexploitable en l'état, les pixels avoisinant les lettres s'étant noircis. On peut toujours voir clairement là où était le texte, et par ailleurs nous ne voyions que le texte, toute trace de bruit numérique ayant disparu. Sans le vouloir, nous avons créé un algorithme mettant en valeur les zones comportant du texte, et excluant tout pixel indésirable, à l'exception de traits trop gros ou tout autre rassemblement important de pixels.

Nous avons donc pu en faire un "filtre", où chaque pixel en nuance de gris, s'il est noir sur ce filtre et plus sombre que la moyenne sur l'image en nuances de gris, est alors considéré comme faisant partie d'une lettre, et est sinon "éliminé", c'est-à-dire rendu blanc dans la matrice de sortie.

Les résultats de notre algorithme se sont donc grandement améliorés, malgré un traitement durant parfois une à deux secondes par image. Cela a été d'entrée de jeu le désavantage de ce traitement, surtout quand un OCR est censé traiter beaucoup de données en même temps, notamment à cause du réseau de neurones. Néanmoins, nous avons continué nos recherches.

Une dernière approche intuitive pour s'améliorer était de calculer la luminosité moyenne locale de plusieurs zones de l'image au lieu de faire cette moyenne sur toute l'image. Concrètement, l'image est divisée en n parties, avec un n fixe dans un premier temps, dont on calcule la luminosité moyenne locale. Ensuite, à chaque fois qu'une valeur de luminosité moyenne était demandée dans le code, la moyenne locale était utilisée à la place.

La touche finale est l'estimation du nombre de parties requises pour le meilleur résultat possible, en fonction de la taille et du nombre de lettres. Plus les lettres sont grandes, moins il est nécessaire de diviser l'image. Pour cela, nous calculons le rapport entre le nombre de pixels à la bordure des lettres par la hauteur de l'image, grâce aux informations sur les voisins de chaque pixel.

Le résultat après tant d'effort fut très satisfaisant, malgré un temps d'exécution important. Malheureusement, le programme s'arrête dans de très rares cas lors du pré-traitement, à cause d'une SEGFAULT, erreur lors de l'écriture dans la mémoire, d'origine encore inconnue, même après l'utilisation d'outils

comme gdb pour essayer de la trouver. Faute de temps, le bug n'a pas pu être corrigé avant le rendu final et l'ancien algorithme a été utilisé. Cependant, cette recherche de l'algorithme le plus performant a été très instructive. Elle a permis d'acquérir des connaissances sur le langage C et de l'expérience sur des méthodes de recherche sur un sujet non documenté (volontairement dans le cas présent).

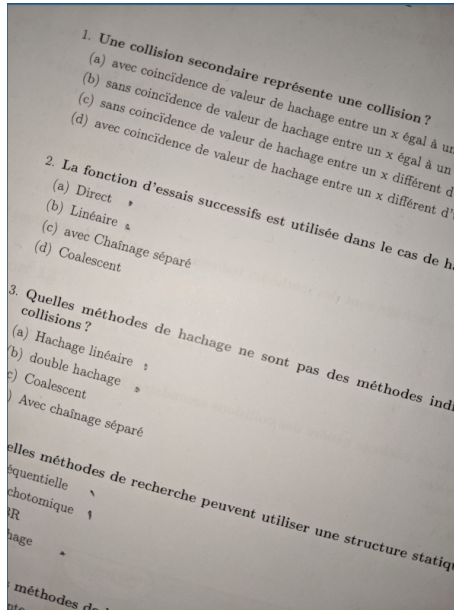


FIGURE 1 – Image avant tout traitement

Le second type d'images que nous souhaitons pouvoir accepter en entrée est l'ensemble des images avec un angle quelconque, par exemple une photo de texte ou un scan légèrement raté. Pour cela, il nous fallait un algorithme de détection d'angle et un algorithme de correction d'angle. Cette fois-ci, nous avons fait des recherches sur Internet et nous sommes tombés sur plusieurs sites présentant des solutions très intéressantes. Nous nous sommes évidemment concentrés en premier sur la détection d'angle.

Deux méthodes se démarquent du lot :

- L'utilisation d'une fonction déterminant le rectangle d'aire minimum englobant tous les pixels noirs de l'image (MinAreaRect), l'angle de ce rectangle étant égal à l'angle du texte.
- L'estimation de l'angle avec la position de certains des pixels les plus à gauche de l'image.

La première demande un haut niveau mathématique et repose surtout sur un principe très complexe. Elle est assez rebutante aux premiers abords, et ayant trouvé la deuxième méthode, nous nous sommes plutôt penchés sur cette dernière. Cependant, aux termes de nos recherches sur MinAreaRect, nous avons trouvé une implémentation en C++ sur le site GeometricTools de D. Eberly.

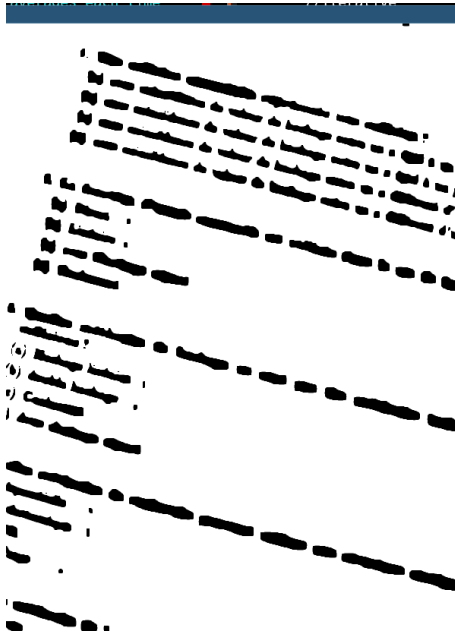


FIGURE 2 – Filtre donnant la position du texte

1. Une collision secondaire représente une collision ?
 (a) avec coïncidence de valeur de hachage entre un x égal à un y
 (b) sans coïncidence de valeur de hachage entre un x égal à un y
 (c) sans coïncidence de valeur de hachage entre un x différent d'un
 (d) avec coïncidence de valeur de hachage entre un x différent d'un

2. La fonction d'essais successifs est utilisée dans le cas de hachage
 (a) Direct
 (b) Linéaire
 (c) avec chaînage séparé
 (d) Coalescent

3. Quelles méthodes de hachage ne sont pas des méthodes indirectes ?
 (a) Hachage linéaire
 (b) double hachage
 (c) Coalescent
 Avec chaînage séparé

4. Les méthodes de recherche peuvent utiliser une structure statique
 (a) arborescente
 (b) arborescente
 (c) arborescente
 (d) arborescente

5. Les méthodes de recherche peuvent utiliser une structure statique
 (a) arborescente
 (b) arborescente
 (c) arborescente
 (d) arborescente

FIGURE 3 – Image inclinée issue d'une photo (également résultat du pré-traitement "lourd" après application du filtre précédent)

Pour faire simple, la deuxième méthode se résume à prendre une partie des pixels de l'image, ceux les plus à gauche, et de calculer la moyenne de leurs coordonnées, notée A_x et A_y . Ensuite, la moyenne des coordonnées des 10%

Une collision secondaire représente une collision ?
 (a) avec coïncidence de valeur de hachage entre un x égal à un y
 (b) sans coïncidence de valeur de hachage entre un x égal à un y
 (c) sans coïncidence de valeur de hachage entre un x différent d'un y
 (d) avec coïncidence de valeur de hachage entre un x différent d'un y

La fonction d'essais successifs est utilisée dans le cas de hachage
 (a) Direct
 (b) Linéaire
 (c) avec Chaining séparé
 (d) Coalescent

Quelles méthodes de hachage ne sont pas des méthodes induisant des collisions ?
 (a) Hachage linéaire
 (b) double hachage
 (c) Coalescent
 (d) Avec chaining séparé

Les méthodes de recherche peuvent utiliser une structure de données
 séquentielle
 chronologique
 B+
 hachage

Les méthodes de recherche peuvent utiliser une structure de données
 séquentielle
 chronologique
 B+
 hachage

FIGURE 4 – Après correction d'angle

pixels les plus à gauche parmi ces pixels est calculée, notée B_x et B_y . Il suffit enfin de calculer l'angle orienté $(AO; AB)$, avec O un point de même abscisse que A et la même ordonnée que B . Cet angle est en fait l'angle du texte.

Sur le principe, cela paraît efficace et très simple, et c'est ce qui nous a attiré. En effet, algorithmiquement, le défi le plus complexe fut de coder une fonction de tri, à savoir un "quicksort" un peu modifié pour prendre deux listes en paramètres.

En pratique, les résultats sont bons si la qualité de l'image est bonne mais des comportements étranges surviennent si le texte n'est pas rassemblé en un rectangle parfait, comme constaté sur l'image 6).

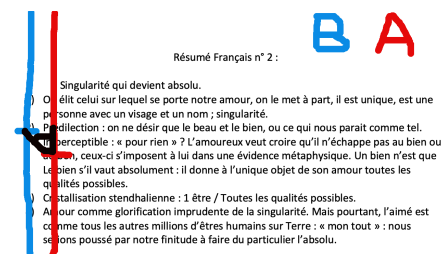


FIGURE 5 – Représentation des points A et B ainsi que leur angle

Résumé Français...

- Singularité qui devient absolu.
- 1) On élit celui sur lequel se porte notre amour, on le met à part, il est unique, s...
- personne avec un visage et un nom ; singularité.
- 2) Prédilection : on ne désire que le beau et le bien, ou ce qui nous paraît comme tel.
- imperceptible : « pour rien » ? L' amoureux veut croire qu'il n'échappe pas au bien ou
- au bon, ceux-ci s'imposent à lui dans une évidence métaphysique. Un bien n'est que
- Le bien s'il vaut absolument : il donne à l'unique objet de son amour toutes les
- qualités possibles.
- 3) Cristallisation stendhalienne : 1 être / Toutes les qualités possibles.
- 4) Amour comme glorification imprudente de la singularité
- comme tous les autres millions d'êtres h...
- serions poussé par notre fin...

FIGURE 6 – Résultat de la "correction" d'angle

4.2.4 Résultat final

L'algorithme final pour l'augmentation du contraste est suffisamment performante dans la plupart des situations, notamment dans la gestion des fonds sombres. Cependant, il souffre de plusieurs défauts, comme l'incapacité à exploiter un texte écrit sur une feuille à carreaux, faisant ressortir ces derniers sur le résultat final. Nous avons fait des recherches pour tenter de résoudre ces problèmes, et nous y sommes presque arrivés.

4.3 Segmentation du document

Participants au développement :
— Nicolas FROGER

Dans la chaîne finale du programme de reconnaissance des caractères d'une image, la segmentation joue un rôle très important. En effet, c'est cette partie du programme qui, à partir d'une grande image, va récupérer chaque caractère afin de les envoyer ensuite un par un au réseau de neurones pour la reconnaissance. Cette partie, essentielle, n'a pas été facile à implémenter.

4.3.1 Principe de la segmentation

Le fonctionnement de notre segmentation est le suivant. Lorsque le traitement et la binarisation de l'image sont effectués, la segmentation entre en jeu. Lorsque l'image est binarisée, la segmentation reçoit une grande matrice de 0 ou de 1 correspondant à la valeur du pixel de l'image. La première étape de la segmentation est de découper l'image en lignes. Pour cela, nous calculons la somme de chaque ligne de pixels de la matrice. Si une ligne de pixels est vide, sa somme sera donc égale à 0. À l'inverse, si une ligne contient au moins un pixel, la somme de cette ligne sera supérieure ou égale à 1. De cette manière, nous pouvons déterminer les lignes de l'image qui ne contiennent pas de texte. Nous délimitons les lignes de texte à l'aide des lignes de pixels vides. Cette première étape retourne alors un tableau de matrices, correspondant à un tableau de lignes.

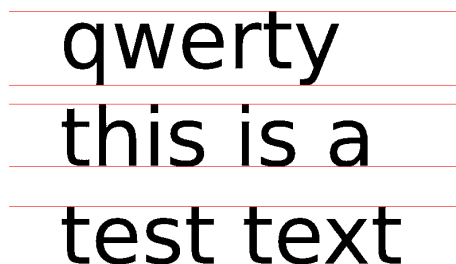
The image shows three lines of text: "qwerty", "this is a", and "test text". Each line is enclosed within a thin red rectangular border, illustrating the result of line segmentation. The text is centered and uses a clean, sans-serif font.

FIGURE 7 – Visualisation de la segmentation en lignes

La seconde partie est très similaire à la précédente. En effet, les mêmes opérations vont être effectuées, mais cette fois-ci verticalement sur chacune des matrices de lignes. Nous effectuons donc la somme des colonnes de pixels des matrices de lignes et nous déterminons les colonnes qui ne contiennent pas de pixels, c'est-à-dire celles dont la somme de pixels est nulle. Nous délimitons ainsi les différents caractères présents sur la ligne. C'est également à ce moment que nous détectons les espaces entre les mots.

4.3.2 Calcul des espaces entre les mots

Afin de détecter les espaces entre les mots, nous effectuons un parcours de toutes les lignes afin de déterminer deux valeurs d'espacement entre les différents caractères. Nous recherchons à ce moment deux valeurs : une première "minimale" correspondant au nombre moyen de colonnes de la matrice séparant deux lettres dans un mot, et une valeur "maximale" censée correspondre au nombre moyen de colonnes de la matrice séparant deux mots. Au début de l'algorithme, les deux valeurs sont égales et correspondent à la largeur du premier espace qu'il trouve entre deux caractères. L'algorithme parcourt ensuite toutes les lignes et met à jour la valeur minimale ou maximale. S'il détecte un espace inférieur à la valeur minimale, elle est alors mise à jour en calculant la moyenne entre l'ancienne valeur et l'espacement trouvé. Si l'espacement calculé est supérieur à la valeur maximale, celle-ci est alors mise à jour avec la moyenne entre l'ancienne valeur et l'espacement calculé. De cette manière, à la fin de l'exécution de l'algorithme, on obtient deux valeurs moyennes : la taille moyenne d'un espace entre deux lettres et la valeur moyenne d'un espace entre deux mots.

Lors de la segmentation verticale, nous calculons l'espacement entre chaque caractère en calculant la différence entre le numéro de colonne correspondant à la fin du caractère précédent et le numéro de colonne correspondant au caractère actuel. Nous comparons ensuite cette valeur avec les deux valeurs d'espacement calculées au préalable. Si la valeur est plus proche de la valeur minimale, nous pouvons conclure qu'il s'agit d'un simple espace entre deux caractères, et l'algorithme poursuit donc la segmentation. Sinon, si la valeur est plus proche de la valeur maximale, nous pouvons conclure qu'il s'agit d'un espace entre deux mots, et nous ajoutons alors à la liste de résultat un espace.

Cette méthode de calcul a néanmoins ses limites. En effet, dans le cas où le texte ne possède aucun espace entre les mots, l'algorithme retournera tout de même deux valeurs minimales et maximales très proches. Lors de la segmentation, des différences très basses d'espacement entre les lettres sera considéré comme un espace entre deux mots. Cependant, dans le cas général, les extraits de texte donnés au programme seront composés de mots.

À la fin de la première période de développement s'arrêtant à la première soutenance d'octobre, la segmentation n'était pas capable de déterminer les espaces entre les mots. En effet, seuls les caractères étaient délimités et les espaces n'étaient pas considérés. De plus, la structure de données que retournait la segmentation à ce moment là n'était pas adaptée. Il s'agissait d'un tableau de tableaux de pointeurs vers des matrices. Nous avions alors un triple pointeur. Or, nous avons vu en cours de programmation qu'il fallait éviter les doubles pointeurs lorsque cela était possible, mais nous avions ici des triples pointeurs, ce qui nous semblait être pire encore. De ce fait, nous avons pensé à une nouvelle structure de données.

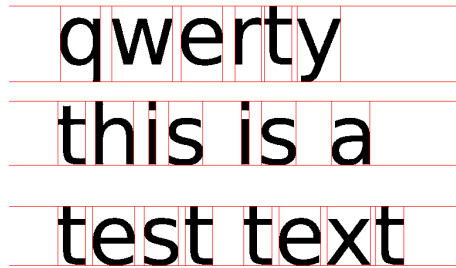


FIGURE 8 – Visualisation de la segmentation verticale

4.3.3 Stocker le résultat de la segmentation

Pour éviter les triples pointeurs, ainsi que pour améliorer les performances, nous avons développé une bibliothèque de listes chaînées à double sens. Il s'agit d'une structure qui, pour chaque élément, dispose de deux pointeurs. Le premier pointe vers l'élément précédent de la liste, et le second pointe vers l'élément suivant. De plus, nous utilisons des "sentinelles", situées en début de liste, permettant de manipuler plus facilement les pointeurs de la liste. Ce type de structure est adaptée pour des opérations aux extrémités de la liste, c'est-à-dire au début et à la fin. Dans le cadre de la segmentation et pour la suite du projet, c'est précisément ce dont nous avons besoin.

Le type de donnée que nous stockons dans cette liste est également différent de précédemment. Auparavant, nous ne faisons que stocker des matrices dans des tableaux. Lorsque nous avons développé notre bibliothèque de listes chaînées, nous avons opté pour une autre manière de stocker les caractères segmentés de l'image. En effet, avec l'ancienne méthode, il était impossible de stocker les espaces entre les mots. Pour palier à ce problème, nous avons créé une structure représentant un élément de texte. Cette structure dispose de deux éléments : le premier est un nombre, correspondant au type de l'élément de texte stocké. S'il s'agit d'un caractère, donc d'une lettre notamment, le type est 0. Le type 1 est associé aux espaces et le type 2 aux retours à la ligne. Si le type est 0, donc qu'il s'agit d'un caractère, la deuxième donnée de la structure est utilisée. Il s'agit d'un pointeur vers une matrice représentant le caractère dans l'image. Lorsque l'élément n'est pas un caractère, le pointeur vers une matrice est simplement nul.

Lorsque la segmentation s'effectue, l'élément de texte détecté est ajouté dans la liste à la fin. Cette opération est rendue très simple grâce à la sentinelle en début de liste qui dispose d'un pointeur vers le dernier élément de la liste. Après la segmentation et lors de l'envoi des caractères dans l'ordre vers le réseau de neurones, nous retirons simplement l'élément suivant la sentinelle jusqu'à

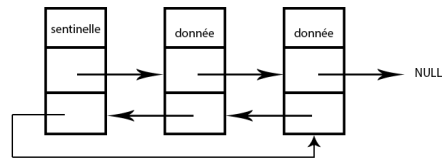


FIGURE 9 – Représentation d’une liste chaînée à double sens de deux éléments

qu’il n’y ait plus rien. Les premiers caractères ajoutés dans la liste lors de la segmentation sont les premiers à y sortir lors de la reconnaissance. De cette manière, nous pouvons dire que notre utilisation de ces listes chaînées à double sens s’assimile aux files, une structure de donnée.

4.3.4 Généralisation des matrices de caractère

Afin de répondre aux besoins du réseau de neurones, il a été nécessaire d’étudier les dimensions des matrices de caractères après la segmentation. En effet, le réseau de neurones dispose d’un nombre fixe de neurones d’entrées. Il n’est donc pas possible de lui envoyer des matrices de taille différentes. Pour palier à ce problème, nous avons pensé à fixer une taille générale pour toutes les matrices de caractère. Plusieurs étapes sont donc nécessaires pour transformer une matrice quelconque en une matrice compatible avec le réseau de neurones. La première est de la rendre carrée. Nous ajoutons donc des lignes ou des colonnes de 0 afin que la matrice représentant le caractère devienne carrée. La seconde étape est le rétrécissement de la matrice. Arbitrairement, nous avons choisi une taille de matrice de 26x26. Pour réduire la taille de la matrice, l’algorithme calcule d’abord le rapport entre la taille de la matrice et la taille désirée. Un parcours de la matrice est effectué avec un pas égal au rapport calculé précédemment. Nous obtenons alors une matrice de taille 26x26 dans laquelle nous avons le même caractère que dans la matrice calculée lors de la segmentation.

4.4 Réseau de neurones

Participants au développement :

- Enzo BOURICHE
- Thomas BERLIOZ

Le cœur du projet est la reconnaissance de chaque caractère, afin de pouvoir restituer le texte original. Pour cela, il était obligatoire d'implémenter un réseau de neurones capable non seulement d'être performant sur un grand nombre de polices mais également optimisé pour s'entraîner efficacement.

Afin d'expliquer le fonctionnement général de l'architecture choisie pour le réseau de neurones, c'est-à-dire un réseau multi-couches, nous utiliserons d'abord le réseau capable d'effectuer un XOR (i.e. un OU exclusif), construit en premier lieu afin de nous habituer à manipuler les algorithmes, car il est plus simple de le représenter. Ensuite sera détaillé le réseau de neurones utilisé pour l'OCR.

4.4.1 Architecture du réseau de neurones - Cas du XOR

Le réseau de neurones implémenté au départ, et présenté lors de la première soutenance, pour la fonction XOR peut se représenter graphiquement sous la forme suivante :

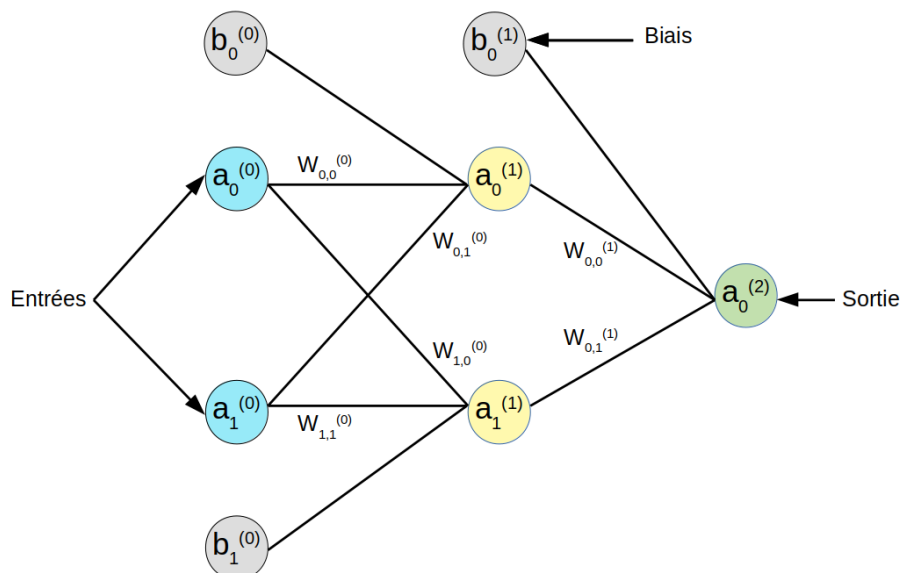


FIGURE 10 – Réseau XOR

Ici, chaque cercle représente un neurone ou un biais et chaque lien étiqueté représente un poids avec les caractéristiques suivantes : $w_{jk}^{(l)}$ est le poids reliant le neurone k de la couche l au neurone j de la couche $l + 1$. Chaque neurone a une valeur d'activation de type *double* comprise entre 0 et 1 grâce à la fonction d'activation choisie, la sigmoïde :

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

Lorsque x tendra vers des valeurs respectivement très grandes ou très petites, $\sigma(x)$ tendra vers des valeurs respectivement proches de 1 ou de 0 grâce à l'exponentielle.

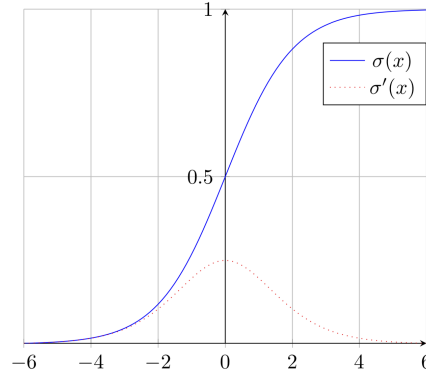


FIGURE 11 – Fonction sigmoïde

L'objectif est de calculer, tour à tour, la valeur d'activation de chaque neurone dépendant des poids et des biais qui lui sont reliés afin d'obtenir le résultat en sortie, ici directement la valeur du seul neurone de la dernière couche. Pour cela, tout neurone $a_j^{(l)}$, en tant que $j^{ième}$ neurone de la $l^{ième}$ couche se calcule de la manière suivante, où n est le nombre de neurones de la couche $l - 1$:

$$a_j^{(l)} = \sigma \left(\sum_{k=0}^n w_{jk}^{(l-1)} a_k^{(l-1)} + b_j^{(l-1)} \right) \quad (2)$$

Ainsi, voici par exemple comment se calcule l'activation du neurone $a_0^{(1)}$ d'un réseau quelconque :

$$a_0^{(1)} = \sigma \left(w_{0,0}^{(0)} a_0^{(0)} + w_{0,1}^{(0)} a_1^{(0)} + \dots + w_{0,n}^{(0)} a_n^{(0)} + b_0^{(0)} \right) \quad (3)$$

On reconnaît ainsi l'équivalent d'une opération entre matrices. C'est donc pour cela que nous avons décidé d'implémenter le calcul d'une couche de valeurs

d'activations de la manière suivante : Soit W la matrice représentant les poids de la couche l , composée de n neurones, vers la couche $l + 1$, composée de k neurones :

$$W = \begin{bmatrix} w_{0,0}^{(l)} & w_{0,1}^{(l)} & \dots & w_{0,n}^{(l)} \\ w_{1,0}^{(l)} & w_{1,1}^{(l)} & \dots & w_{1,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0}^{(l)} & w_{k,1}^{(l)} & \dots & w_{k,n}^{(l)} \end{bmatrix}$$

$a^{(l)}$ la matrice représentant les neurones de la couche l :

$$a^{(l)} = \begin{bmatrix} a_0^{(l)} \\ a_1^{(l)} \\ \vdots \\ a_n^{(l)} \end{bmatrix}$$

et $b^{(l)}$ la matrice représentant les biais qui influencent la couche $l + 1$:

$$b^{(l)} = \begin{bmatrix} b_0^{(l)} \\ b_1^{(l)} \\ \vdots \\ b_k^{(l)} \end{bmatrix}$$

On peut alors écrire :

$$a^{(l+1)} = \sigma \left(W a^{(l)} + b^{(l)} \right) \quad (4)$$

Ainsi nous avons décidé d'implémenter notre réseau de neurones de L couches sous la forme d'une liste de matrices de neurones (de taille L donc), d'une liste de matrices de poids (de taille $L - 1$) et d'une liste de matrices de biais (de taille $L - 1$). De cette manière, toute couche L , c'est-à-dire toute matrice de neurones (d'indice l dans sa liste), pourra être calculée à l'aide du produit de la matrice de poids (d'indice $l - 1$ dans sa liste), par la matrice de neurones précédente (d'indice $l - 1$ dans sa liste), à laquelle on ajoute la matrice de biais (d'indice $l - 1$ dans sa liste).

Après avoir réussi une implémentation propre avec la structure du schéma de la figure 10, nous avons réalisé que certaines fois le réseau s'entraînait mal. Ainsi, ayant constaté l'importance d'un réseau correctement paramétré, nous avons opté pour un réseau XOR avec 100% de réussite sous la forme :

$$L = \{2, 4, 1\}$$

4.4.2 Architecture générale du réseau de neurones

Lorsqu'il a fallu adapter le réseau de neurones afin qu'il soit fonctionnel pour l'OCR, la question du nombre de couches et du nombre de neurones par couche s'est posée. Comme il n'existe pas de règle générale, seule l'expérimentation nous a permis de déduire la meilleure structure pour le réseau. Ainsi nous avons conclu que le réseau optimisé était de la forme :

$$L = \{676, 300, 200, 200, 150, 100, 100, 26\}$$

Les 676 neurones de la première couche sont les 0 et les 1 qui constituent la matrice de chaque caractère passé en entrée au réseau. En effet, il s'agit du résultat de la segmentation qui sépare les caractères en matrices de 26 par 26, que nous pouvons afficher à l'aide de `print_letter` présentée dans la partie sur la bibliothèque de matrices implémentée.

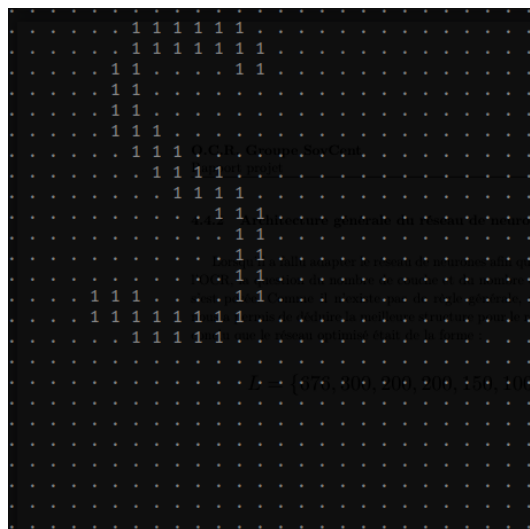


FIGURE 12 – Matrice de la lettre s

Le nombre décroissant de neurones à chaque couche est plus instinctif que réellement prouvé mais cela permet une visualisation naturelle du réseau s'affinant de 676 neurones à 26 neurones, donc nous avons décidé de garder cette forme qui a fait ses preuves du début à la fin.

Enfin, les 26 neurones de la dernière couche correspondent aux 26 caractères que le réseau est capable de reconnaître, c'est-à-dire les 26 lettres de l'alphabet en minuscule. Initialement, il devait reconnaître 54 caractères, avec l'alphabet majuscule, la virgule et le point, mais comme nous avons rencontré des problèmes lors de l'apprentissage, nous avons réduit le spectre de reconnaissance

afin qu'il fonctionne au moins sur un jeu plus réduit d'exemples. Le problème ayant été résolu très tard, nous n'avons pas eu le temps d'ajouter les majuscules, mais maintenant que le réseau est fonctionnel, il serait facile de le faire, d'autant plus que notre implémentation est assez souple pour permettre de grosses modifications en un minimum de changements.

4.4.3 Initialisation du réseau de neurones

Afin d'initialiser facilement un réseau de neurones, nous avons créé une structure *neuralnet* qui contient le nombre de couches du réseau L ; le nombre de neurones par couche; la liste de matrices des neurones avant, et après l'application de la fonction d'activation, les deux de taille L ; la liste de matrices de poids, *weights*, de taille $L - 1$ et la liste des matrices de biais, *biases*, de taille $L - 1$. C'est la fonction *init_neural_net* qui génère ces composantes, avec des neurones nuls, et, au départ, des poids et biais aléatoires compris entre -2.5 et 2.5.

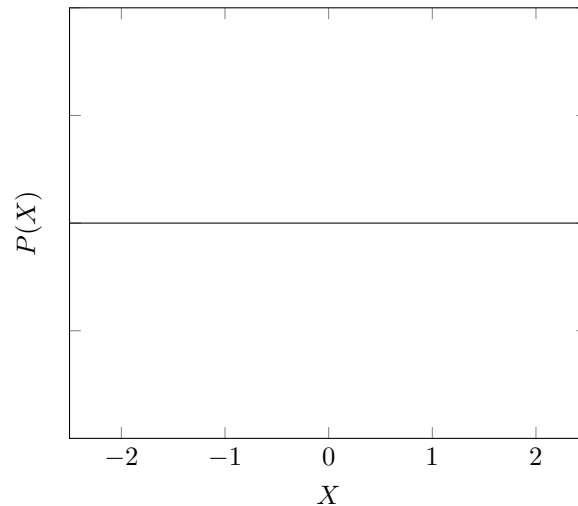
Plus tard, lorsque nous cherchions pourquoi le réseau ne s'entraînait pas correctement, nous avons trouvé un document décrivant les problèmes liés à une distribution aléatoire équiprobable des poids et des biais dans un intervalle centré en 0, ce qui était notre cas. Selon celui-ci, dans le cas d'un réseau de neurones suffisamment peu profond comme le nôtre, une initialisation aléatoire des poids et des biais pourrait être la cause d'un problème appelé le "gradient explosif", problème apparemment d'autant plus récurrent avec l'utilisation de la sigmoïde.

En effet, l'utilisation de la sigmoïde pourrait mener à une grande variation de la fonction de coût (décrite dans la partie sur l'entraînement du réseau), qui signifie elle-même un plus grand δ à chaque itération sur les couches du réseau, influençant donc plus les changements appliqués aux poids et aux biais (fonctionnement également détaillé dans la partie sur l'entraînement du réseau). Pour faire simple, les changements sont grands, et ce de plus en plus.

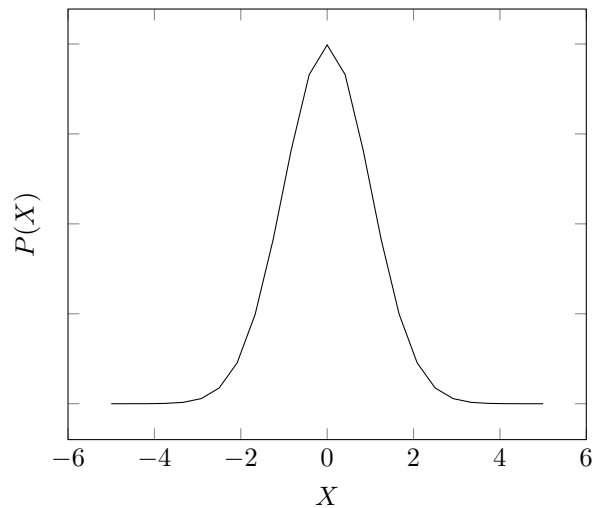
Par exemple, disons que le δ d'un poids (ce pourrait aussi être un biais) quelconque est positif en un premier temps, la fonction de coût va donc vouloir le corriger dans l'autre sens, et ce avec un δ potentiellement encore plus grand mais cette fois-ci négatif. Ainsi, le réseau va sans cesse rater son point optimal, à cause de corrections δ trop grandes, elles-mêmes causées par l'interaction entre la sigmoïde et des valeurs de poids et de biais initialement trop grandes et trop dispersées. On peut l'illustrer de la manière suivante, où $+$ est grand nombre positif, $-$ un grand nombre négatif et $w_{jk}^{(l)}$ un poids quelconque :

$$+w_{jk}^{(l)} \xrightarrow{--\delta} -w_{jk}^{(l)} \xrightarrow{++\delta} ++w_{jk}^{(l)} \xrightarrow{\dots} \dots$$

C'est la dispersion des valeurs initiales de poids et de biais qui est le point important : ce que nous voulons en réalité, ce sont des valeurs plus resserrées autour d'une moyenne. Cela s'est donc fait grâce une prise de valeurs selon une distribution de Gauss d'écart-type 1 et centrée en 0. Nous avons donc modifié la distribution initiale suivante :



pour avoir la distribution suivante :



Une telle distribution peut être retrouvée algorithmiquement en prenant deux valeurs x et y aléatoires entre 0 et 1 et en effectuant l'opération suivante :

$$gauss(x, y) = \sqrt{-2 \log(x)} \times \cos(2\pi y) \quad (5)$$

En utilisant une telle fonction sur le réseau XOR de la forme $L = \{2, 2, 1\}$ par exemple (afin de constater la différence, l'autre forme fonctionnant déjà tout le temps), nous avons remarqué un temps d'entraînement nécessaire très réduit, et un taux de réussite bien plus haut. Malheureusement, malgré cela, le réseau OCR ne semblait toujours pas vouloir apprendre les 52 symboles prévus initialement (majuscules, minuscules, point et virgule). Nous avons toujours, au final, un réseau qui, en essayant de baisser sa fonction de coût, convergait sur la reconnaissance d'un seul caractère.

4.4.4 Calcul des valeurs d'activation

Le calcul des valeurs d'activation se fait avec la fonction *feed_forward*. C'est la même que celle décrite dans le XOR, c'est-à-dire qu'en parcourant la liste des matrices de neurones de l'entrée à la sortie, nous mettons à jour, couche par couche, les valeurs d'activation.

En revanche, comme ce réseau est bien plus grand que celui de XOR, nous avons donc constaté les premiers problèmes liés à la mémoire. En effet, il est précisé dans la partie sur la bibliothèque de matrices que chaque fonction crée une nouvelle matrice, et donc alloue de la mémoire. Cela prend son importance car les matrices qui sont manipulées sont bien plus grandes. La liste de matrices de poids, par exemple, est constituée au total de 370 400 poids, qui sont des *double*. Ainsi, si nous ne libérons pas les matrices avant de les mettre à jour, ou que nous n'isolons pas chaque calcul pour libérer son résultat une fois qu'il a été utilisé, les fuites de mémoire sont colossales. Il faut donc libérer chaque couche de neurones avant la mettre à jour.

Ensuite, suite au changement d'implémentation de la fonction d'entraînement, nous avons besoin de stocker les valeurs des neurones avant l'application de la fonction d'activation. Nous avons donc également ajouté la mise à jour de la liste des matrices de neurones avant que soit appliquée cette fonction.

Enfin, à cause des problèmes rencontrés lors de l'entraînement du réseau, nous avons trouvé que ceux-ci pouvaient être liés au choix de la fonction d'activation. Ainsi, nous avons tenté de remplacer la sigmoïde par la fonction ReLU (Rectified Linear Unit), qui s'écrit :

$$ReLU(x) = \begin{cases} x & \text{si } x > 0 \\ 0 & \text{sinon} \end{cases} \quad (6)$$

mais cela n'a mené qu'à faire exploser la fonction de coût. En effet, les valeurs d'activation n'étaient plus bornées entre 0 et 1, donc lors de la comparaison avec le résultat attendu (i.e. 0 ou 1), le résultat était toujours bien supérieur à 1. En conséquences, à force d'additionner de grandes valeurs, on dépassait la taille d'un *double* et la fonction de coût n'était plus affichable. Nous avons également essayé avec la tangente hyperbolique, mais celle-ci donnait un comportement

étrange à la fonction de coût, donc nous avons gardé la sigmoïde, qui, au final, a fonctionné et permet un temps d'entraînement tout à fait correct.

4.4.5 Fonction d'entraînement : la descente de gradient

L'objectif de la fonction d'entraînement est d'apprendre au réseau sa propre fonction. Concrètement, elle sert à savoir comment modifier les poids et biais afin que la sortie corresponde à ce qui est attendu en fonction de l'entrée. Grâce aux matrices de tous les caractères générées après la segmentation, le réseau peut lui-même analyser si la sortie est correcte ou non. L'apprentissage se déroule en plusieurs étapes, et nécessite deux nouvelles listes de matrices : δW et δB . Ces deux listes de matrices, respectivement de la même taille que W et B , sont remplies au fur et à mesure de la fonction d'entraînement. Elles permettent de modifier les poids et les biais du réseau afin qu'il soit fonctionnel.

Pour comprendre la manière dont on calcule ces deux listes de matrices, il faut introduire la fonction de coût. C'est une fonction qui dépend des paramètres du réseau et qui indique à quel point le réseau est précis ou non. Ainsi, elle permet de voir comment le réseau se comporte en fonction de chacun de ses poids et biais, ce qui nous intéresse. On la notera C . Nous avons choisi la fonction quadratique de coût, qui peut s'exprimer

$$C = \frac{1}{2n} \sum_x |y - a^{(L-1)}|^2 \quad (7)$$

où L est le nombre de couche du réseau ; $a^{(L-1)}$ la matrice de neurones de la couche $L - 1$ (i.e. la sortie) après l'application de la fonction d'activation, dans notre cas et pour la suite, la sigmoïde ; x un exemple d'entraînement individuel et n le nombre total de générations.

Tout d'abord, on donne à la fonction d'entraînement une entrée aléatoire, c'est-à-dire la matrice d'un caractère aléatoirement choisi dans le jeu d'entraînement, et son caractère associé sous la forme d'une matrice correspondant à la sortie attendue, qu'on note ici y . On calcule ensuite la matrice δ l'erreur associée à la dernière couche (i.e. la sortie) grâce à y :

$$\delta = (a^{(L-1)} - y) \odot \sigma'(z^{(L-1)}) \quad (8)$$

où \odot est le produit de matrice terme à terme et $z^{(L-1)}$ la matrice de neurones de la couche $L - 1$ (i.e. la sortie) avant l'application de la fonction d'activation.

Commence ensuite la propagation arrière de l'erreur. Elle consiste à propager l'erreur en remontant le réseau de neurones depuis la sortie, et à remplir δW

et δW au fur et à mesure. Ainsi, à chaque couche en partant de $L - 2$, on répète les mêmes étapes. D'abord, on a, où l est la couche actuelle,

$$\frac{\partial C}{\partial b^{(l)}} = \delta \quad (9)$$

dont le résultat est stocké dans δB dans la couche l . Ensuite, on a

$$\frac{\partial C}{\partial w} = \delta (a^{(l)})^T \quad (10)$$

où T est l'opérateur transposée, et dont le résultat est stocké dans δW dans la couche l . Enfin, on met à jour δ pour la prochaine boucle, ce qui permet la propagation arrière de l'erreur de couche en couche :

$$\delta = ((w^{(l)})^T \delta) \odot \sigma'(z^{(l)}) \quad (11)$$

Il est intéressant de noter que nous avons essayé d'ajouter une variable m appelée le momentum, comprise entre 0 et 1 et qui permettait, lors du calcul de la matrice l de δW , d'ajouter une proportion de la matrice $l+1$ de δW . Cela permettait d'accentuer chaque modification dans sa direction, c'est-à-dire que plus on a voulu augmenter un poids à la couche d'avant, plus on augmentera ceux qui arrivent à son origine (la propagation se faisant en arrière), et inversement. Mais la recherche de ce m fut trop longue pour garder cette solution.

Évidemment, on libère après chaque calcul matriciel la matrice utilisée pour le résultat dès lors qu'elle ne sert plus. Enfin, on met à jour la fonction de coût afin de l'afficher entre chaque génération pour garder un œil sur le comportement du réseau.

4.4.6 Mise à jour du réseau : taux d'apprentissage adaptatif

Après avoir suffisamment entraîné le réseau pour enfin arriver à une fonction de coût relativement basse, nous rencontrons un premier problème qui empêche le réseau d'atteindre le vrai minimum de la fonction de coût. Cela peut être bien illustré grâce à la courbe de fonction de coût suivante :

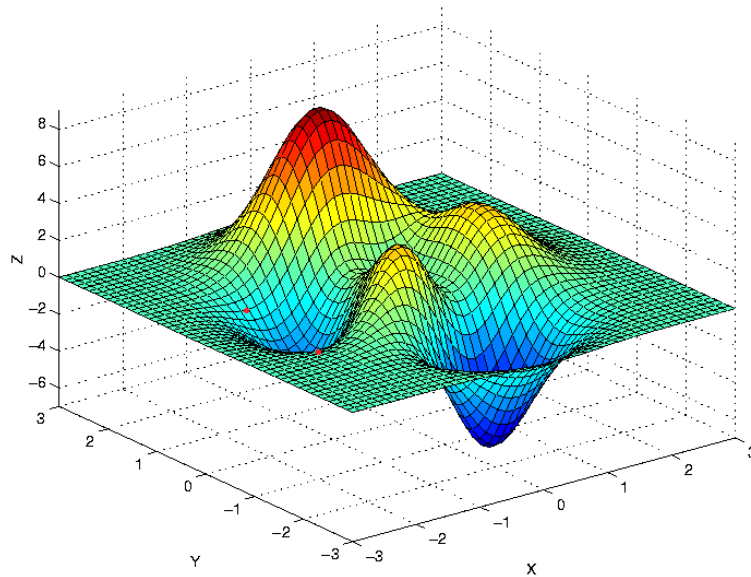


FIGURE 13 – Courbe de coût

Ici, la représentation de la fonction de coût est en 3 dimensions, ce qui équivaut à 2 neurones de sortie. En réalité, l'OCR utilise bien 26 neurones de sortie mais le principe reste le même. La descente de gradient que nous effectuons trouve d'abord le vecteur tangent à cette courbe au point de coût où nous sommes et modifie les poids et les biais selon ce vecteur. Cette modification revient à se déplacer sur la courbe : on peut s'imaginer cela comme une boule qui "roule" sur la courbe.

À chaque entraînement, la boule est "poussée" selon le vecteur trouvé lors de celui-ci. Intuitivement, nous remarquons le problème qui peut survenir lorsque la boule se rapproche du point le plus bas de la courbe. En effet, prenons un des deux points rouges de la courbe ci-dessus. L'entraînement se fait, les poids et les biais sont modifiés, donc la boule se déplace vers le creux. Mais que se passe-t-il si ce changement est trop grand ? Si la boule est "poussée" trop fort ? Naturellement, elle ratera le minimum de la fonction et remontera au deuxième point rouge.

Il faut alors implémenter un taux de d'apprentissage η qui varie selon la progression de la fonction de coût. Ainsi, la norme du vecteur qui dirige notre point rouge diminue au fur et à mesure que le point se rapproche du fond de la courbe, pour enfin s'y stabiliser : la fonction de coût ne variera plus. Autrement dit, le réseau est entraîné.

Pour implémenter cette fonctionnalité, nous fixons d'abord un scalaire η à 0.06, qui multipliera δW et δB à chaque entraînement afin de diminuer son impact sur le réseau de neurones. Ce taux d'apprentissage η doit donc varier

selon la progression du réseau. Il dépend alors du coût car il décroît lorsque le coût se rapproche d'un minimum de sa courbe.

Mais cela soulève un nouveau problème : comment savoir si le minimum atteint est bien le minimum global, que nous cherchons, et non un minimum local de la fonction de coût ? En effet, c'était le problème rencontré lorsque le réseau n'apprenait qu'un seul caractère. Le taux η devenait trop petit, et nous nous retrouvions coincés dans ce minimum car les changements faits aux poids n'étaient plus suffisamment grands pour nous permettre d'en sortir.

C'est pour cela que nous avons décidé de faire varier η selon le nombre de générations. L'opération qui s'est avérée la plus efficace pour réaliser cela fut la suivante, où t correspond à la génération actuelle et T au nombre total de générations :

$$\eta_{(t)} = e^{-\frac{t}{T}} \times \eta_{(t-1)} \quad (12)$$

Ainsi notre fonction de mise à jour des poids devient :

$$weights_{(t)} = weights_{(t-1)} - \eta_{(t)} \times \delta W_{(t)} \quad (13)$$

4.4.7 Paramétrage du réseau

Jusqu'à quelques heures avant l'heure du rendu, nous pensions nous retrouver le jour de la soutenance avec un réseau de neurones non-fonctionnel pour l'OCR, malgré toutes les améliorations que nous avons faites. Nous avons beau augmenter le jeu d'entraînement avec plus de polices différentes, modifier sa forme (i.e. le nombre de couches et le nombre de neurones par couche) ou changer son taux d'apprentissage, rien n'y faisait. Le réseau ne reconnaissait toujours qu'une seule lettre, et cela semblait logique à première vue : c'est un moyen facile pour lui de diminuer sa fonction de coût tôt dans l'entraînement.

Nous pensons maintenant que le coût se coinçait dans un minimum local. Mais nous sommes restés déterminés, car si le réseau pouvait apprendre le XOR, c'est que sa fonction d'entraînement était correcte. Il s'agissait donc simplement d'une question de paramétrage. Nous avons alors eu l'idée cette fois de le faire apprendre sur deux lettres plutôt que les 54 symboles prévus au début. En voyant que le réseau arrivait à reconnaître les deux lettres, nous avons décidé d'en rajouter petit à petit jusqu'aux 26 lettres minuscules de notre jeu de symboles.

En changeant la forme du réseau, nous pouvions remarquer quelques particularités. Un réseau profond se devait d'avoir beaucoup de neurones dans chacune

de ses couches. De même, un réseau avec peu de couches ne fonctionnait bien que si son nombre de neurones par couche était faible et, de plus, apprend très vite, que ce soit en terme de temps ou de générations nécessaires, par rapport au réseau précédent. En revanche, il reste très peu précis.

Le réseau profond, lui, apprend plus lentement, mais a plus de potentiel. En effet, si l'on l'entraîne suffisamment longtemps, il finira par faire de moins en moins de fautes. Après plusieurs essais et beaucoup de temps d'entraînement, nous avons un réseau qui reconnaissait 21 des 26 lettres minuscules. Nous nous sommes alors dit que, grâce au taux d'apprentissage adaptatif, celui-ci finirait sûrement par reconnaître toutes les lettres au fur et à mesure d'autres entraînements, voire qu'il arriverait mieux à apprendre d'autres polices.

4.4.8 Entraînement d'un réseau déjà entraîné

Le ré-entraînement d'un réseau déjà entraîné a deux objectifs primordiaux. Il permet dans un premier temps de repartir d'un réseau qui a de bonnes valeurs de poids et de biais après le ou les entraînement(s) précédent(s) afin de recorriger ses petites erreurs. Dans un second temps, il permet d'adapter intelligemment son taux d'apprentissage nous-même, en fonction des observations de la variation de sa fonction de coût. C'est ce second intérêt qui a fait vraiment fait la différence, car c'est ce paramétrage manuel entre les entraînements qui nous a permis de vraiment baisser le coût du réseau au maximum.

En effet, après beaucoup de tests, nous avons réalisé qu'il y avait potentiellement deux moyens d'expliquer les ralentissements de la baisse de sa fonction de coût : le réseau est coincé dans un minimum local, ou il n'arrive pas à atteindre le minimum qu'il vise à cause d'un taux d'apprentissage trop haut.

Malheureusement, il est bien difficile de différencier ces deux cas car ils se traduisent tout deux par les mêmes observations. Nous avons donc décidé de remonter un peu le taux d'apprentissage η dans les deux cas, car si le réseau est coincé dans un minimum local, il y aura immédiatement une baisse de sa fonction de coût, et, sinon, celui-ci va simplement osciller plus longtemps autour du minimum qu'il vise, et cela ne lui fera que prendre un peu plus de temps pour descendre la pente.

Ainsi, l'arbre de décision est le suivant. Si la fonction de coût baisse de manière constante, nous repartons du même taux d'apprentissage que celui sur lequel l'entraînement précédent s'est arrêté. Si la fonction de coût semble stagner, voire parfois remonter un peu, on recommence l'entraînement avec un taux d'apprentissage un peu plus haut, et si cela ne fonctionne pas, on recommence avec un taux d'apprentissage un peu plus bas.

4.4.9 Réseau de neurones - Conclusion

Il est facile d'affirmer que le réseau de neurones est la partie qui a posé le plus de difficultés. C'est une structure qui nécessite du temps et beaucoup de patience pour être correctement conceptualisée mentalement. Analyser son fonctionnement en affichant les variations des valeurs de ses poids, biais et neurones des centaines de fois fut laborieux et pénible. Le moindre détail a son importance lorsqu'il s'agit de programmer ces algorithmes car la manière dont les matrices sont liées entre elles rend la moindre erreur catastrophique comme elle se répercute sur toutes les étapes du processus.

Néanmoins, nous n'aurions pu espérer un meilleur moyen de progresser en C, surtout dans la compréhension pointue du fonctionnement des pointeurs et de la mémoire que demandait cette partie de l'OCR. De plus, la satisfaction est immense après avoir passé tant de temps sur un même programme, surtout en sachant que jusqu'à quelques heures avant le rendu final, nous pensions ne jamais réussir. C'est une structure très intéressante, et nous avons conscience que nous avons à peine effleuré tout le potentiel d'un réseau de neurones, ce qui rend la chose encore plus fascinante.

4.5 Interface utilisateur

Participants au développement :
— Nicolas FROGER

À la fin de la première soutenance, le logiciel existait uniquement sous la forme d'un exécutable à utiliser en ligne de commandes. Ce type de programme nécessite une certaine connaissance du logiciel et de ses options de lancement pour pouvoir effectuer les différentes tâches disponibles. Afin de palier à ce problème, nous avons développé une interface utilisateur graphique simple à l'aide de la bibliothèque GTK. Une partie de l'interface a été réalisée à l'aide du logiciel *Glade* permettant de concevoir le squelette de l'interface.

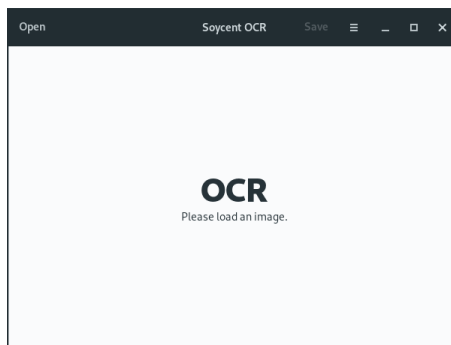


FIGURE 14 – L'interface utilisateur

L'interface se veut très simple, et dispose donc de peu d'éléments. Les différentes actions disponibles sont présentes sur la barre supérieure sous forme de boutons. La reconnaissance de caractères se fait à l'aide du bouton *Ouvrir*. Ce bouton ouvre une boîte de dialogue permettant de choisir l'image parmi les fichiers de l'utilisateur. Lorsque celui-ci charge l'image, le programme va lancer la reconnaissance de caractères. Lorsque celle-ci est terminée, le texte reconnu est affiché. L'utilisateur peut sélectionner le texte à l'aide de sa souris s'il désire par exemple le copier dans le presse-papier de son ordinateur afin de l'utiliser de façon rapide dans un autre programme. Il est également possible de sauvegarder le texte reconnu à l'aide du bouton *Sauvegarder*. En appuyant sur ce bouton, une boîte de dialogue s'ouvre permettant à l'utilisateur de choisir l'emplacement et le nom du fichier dans lequel il veut sauvegarder le texte. L'utilisateur peut relancer la reconnaissance avec d'autres images autant de fois qu'il veut sans problème.

Un menu est présent à droite du bouton *Sauvegarder*. Ce menu contient deux boutons. Le premier bouton permet d'ouvrir la fenêtre servant à entraîner le réseau de neurones utilisé dans la reconnaissance des caractères dans une image. L'utilisateur est guidé pas à pas durant l'entraînement. L'utilisateur choisit un dossier de son ordinateur contenant les images servant à l'entraînement.

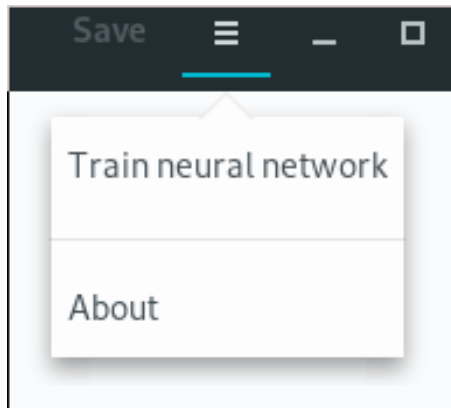


FIGURE 15 – Menu de l'interface

Lorsqu'il clique sur *Suivant*, l'entraînement se lance sur les images du dossier. Lorsque l'entraînement est terminé, l'utilisateur peut fermer la fenêtre et cliquer sur le bouton *Ouvrir* afin de lancer la reconnaissance de texte sur l'image de son choix.

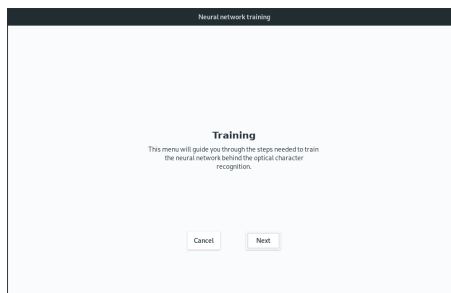


FIGURE 16 – Assistant pour l'apprentissage du réseau de neurones

Enfin, un deuxième choix est disponible dans le menu, permettant d'ouvrir la fenêtre *À propos*. Cette fenêtre présente le logiciel et contient les noms des différents membres du groupe ayant participé au projet.

La bibliothèque GTK ne nous était pas entièrement inconnue car un TP de programmation abordait ce sujet pendant le semestre. Cependant, seuls quelques aspects de la réalisation d'interface étaient abordés. Il a donc fallu beaucoup de recherches, dans la documentation de la bibliothèque principalement, afin de créer l'interface.

La particularité du développement dans le cadre d'une interface est l'utilisation d'*événements* afin de réagir aux interactions de l'utilisateur avec les éléments d'interface. Chaque élément de l'interface, ici majoritairement des boutons, sont connectés avec des fonctions qui effectuent les opérations.

Le problème le plus important rencontré durant la réalisation de l'interface était le lancement de tâches à partir de l'interface. Lorsque l'utilisateur lance la reconnaissance de caractères ou l'entraînement du réseau de neurones, l'interface se bloque durant toute la durée de l'opération. Cela est dû au fait qu'il s'agit d'opérations bloquantes. Pour palier à ce problème, il aurait fallu utiliser des *threads* afin de lancer l'opération coûteuse en temps en arrière plan. Cependant, par manque de temps ainsi que par difficulté à trouver une solution efficace à mettre en œuvre, le problème subsiste dans la version finale du projet.

Dans le programme final, un deuxième exécutable est disponible permettant de lancer le programme sans interface graphique.

4.6 Chaînage des éléments du projet

Participants au développement :
— Nicolas FROGER

Lorsque tous les éléments sont développés de chaque côté par des personnes différentes, il est nécessaire de les lier entre eux. Il s'agit ici d'une étape très importante sans laquelle il serait impossible d'utiliser le programme. Dans cette partie, la communication de groupe est très importante afin d'utiliser correctement le code des autres. La chaîne pour la partie reconnaissance des caractères sur un texte est la suivante :

Interface graphique → Chargement image → Segmentation → Envoi des caractères un par un au réseau de neurones → Affichage du résultat

La chaîne pour l'apprentissage du réseau de neurones est la suivante :

Interface graphique → Chargement du dossier contenant les images d'entraînement → Segmentation des images → Préparation des résultats de l'apprentissage → Entraînement du réseau de neurones

5 Conclusion

Ce projet fût très intéressant, de part sa difficulté ainsi que par les notions qu'il aborde. La transition entre le projet précédent et celui-ci est très marquée pour ceux présents l'année dernière. En effet, il s'agit ici de programmer dans un langage, qui nous était inconnu, des algorithmes complexes, en particulier le réseau de neurones, qui demandent beaucoup de recherche et surtout beaucoup d'essais. Malgré les difficultés rencontrées lors de la réalisation du projet, nous avons énormément appris et sommes satisfaits du résultat, bien qu'il soit en dessous de nos premières attentes, que nous considérons maintenant comme trop élevées face à la difficulté du projet. La cohésion de groupe a été très forte durant ce projet, ce qui nous a mené à avoir un travail de qualité sans le moindre conflit.