# Rapport de soutenance

Charles Quentin Maxime Nathan


Par Bitarrays | Projet "Lambda"

# Table des matières

# *Prologue*

Voici le premier rapport de soutenance pour notre projet d'OCR à réaliser en groupe de 4 étudiants. L"objectif est de répondre pleinement au cahier des charges imposé par l'école et obtenir un résultat satisfaisant dans la reconnaissance des caractères.

Nous n'avons pas eu de phase de réflexion sur le projet pour cette année et avons directement dans le vif du sujet et la réalisation des tâches obligatoires pour respecter scrupuleusement les deadlines du projet et ne pas prendre de retard. Il nous fallait nous familiariser avec le langage C et comprendre les attentes du projet.

Nous allons apprendre à communiquer et à s'adapter aux différents caractères des membres de notre groupe. Nous avons, par expérience, déjà compris qu'un projet de groupe est un réel défi au niveau humain mais ce projet s'annonce encore plus complexe. Nous sommes tous très motivés par la création de l'OCR.

Ce fut un gros investissement pour nous 4 durant ce premier mois pour proposer une segmentation des caractères convenable ainsi que le début de la réalisation de plusieurs éléments qui rendront notre OCR fonctionnel pour la date de la deuxième soutenance.

Nous allons présenter ici nos réalisation ainsi que nos problèmes rencontrés jusqu'à aujourd'hui.

# 1 Présentations

## 1.1 Les membres

### 1.1.1 Quentin "Scout" FISCH

J'ai vraiment découvert ma passion pour l'informatique par le biais de TP effectués en Sciences de L'ingénieur en Première et Terminale. J'ai donc décider d'étudier car je suis très intéressé en ce qui concerne la robotique et le développement de technologies dans les systèmes embarqués. Ce projet sera pour moi mon premier jeu à créer de A à Z et m'apprendra comment gérer un tel projet en respectant des deadlines et diverses contraintes. Je suis ainsi très motivé à sortir le meilleur projet possible et développer mes compétences en informatique.

### 1.1.2 Maxime "Maxmad" MADRAU

Passionné d'informatique depuis toujours, j'ai commencé a développer des programmes simples très tot dans mon enfance. J'ai par la suite appris des langages comme le Python, le Lua, le Javascript, le Swift, et les langages web comme le HTML et le CSS. Mes domaines de prédilection sont les applications mobiles, les bases de données et le client/serveur. La conception de jeux vidéo a pour moi commencé en année de Terminale, pour le Bac d'ISN, ou on a du développer un jeu en Python avec la bibliothèque Pygame. Ce projet m'a permis de découvrir le développement de projets dans le respect de contraintes de temps, de ressources ainsi que de licences et de droits.

### 1.1.3 Charles "Draze" SIMON-MEUNIER

Passionné d'informatique depuis mes années du collège ou j'ai commencé à développer des serveurs sur Minecraft, EPITA est pour moi une révélation. Une école avec une formation permettant d'atteindre le statut d'ingénieur ainsi que des compétences en informatique plus que confortables est pour moi le cursus idéal. J'ai appris par moi meme certains langages comme le HTML/CSS, Java et aujourd'hui je me perfectionne et corrige mes défauts grace à l'école. Ce projet devrait m'apporter à titre personnel beaucoup d'expérience, autant au niveau humain que dans les compétences informatiques.

### 1.1.4 Nathan "Goruza" AVÉ

Après une première année de sup ou jai pu vivre ma première expérience de travail de groupe à EPITA, je poursuis laventure avec plus dexpérience dans les projets de ce type et aussi un groupe différent. Pour moi, lOCR est une seconde expérience de travail où je vais devoir apprendre à maitriser des notions par moi-même et non pas lors des cours. Cela sinscrit dans la continuité de mon stage réalisé cet été par lapprentissage dun nouveau langage et la similarité de mes collègues pour ce projet ayant fait ce stage avec Quentin et Charles. Préparez-vous ce projet sera légendaire !

## 1.2 Le groupe

Nous avons créé notre groupe en reprenant les membres avec qui nous avions travaillé l'année dernière car l'entente était très bonne tout comme l'efficacité au sein du groupe. De plus, nous avons déjà quelques projets de groupe à notre actif ce qui permet de connaître les points forts et faibles de chacun. Malheureusement nous n'étions que 3 et il fallait trouver un dernier membre pour former un groupe de 4 personnes. Nathan était le bon candidat car nous nous connaissons bien et le travail pourrait alors être efficace.

Comme nous avons l'habitude de travailler ensemble, nous savons comment chacun fonctionne et comment chacun réalise ses tâches. Maxime est plus solitaire mais très acharné et réussi donc toujours à réaliser ce qu'il doit faire, Nathan sait innover et trouver les idées pour apporter des améliorations sur des réalisations tandis que Quentin et Charles arrivent à travailler efficacement en binôme plutôt que seuls.

Avec ses données, chacun a pu trouver dans ce projet sa place et fournir un travail qui permet de répondre aux attentes de cette première soutenance. Tout ce que nous avons développé n'est pas parfait mais les idées sont là et ne peuvent qu'être améliorées par la suite.

Ainsi l'entente au sein du groupe est très bonne et devrait permettre de continuer dans ce projet motivant pour révéler le meilleur de chacun et pouvoir réaliser des bonus et rendre notre projet utilisable par la suite.

Notre groupe de projet pour le même nom que l'année dernière (Bitarrays) car nous avons continué de le faire vivre après la fin du projet de S2. Nous avons décidé de donner le nom Lambda à notre projet de cette année.

# 2 Traitement de l'image (pré-traitement)

## 2.1 Ce que nous devions faire

Pour traiter les images de la meilleure des façons, il fallait penser à réaliser un pré traitement sur ces dernières pour que notre segmentation soit la meilleure possible. Il fallait réaliser un niveau de gris, une binarisation et appliquer des filtres permettant de réduire le bruit sur l'image. Toute cette partie est réalisée dans le fichier filters.c, contenu dans **"/src/ImageTreatment"**

## 2.2 Le niveau de gris

Rien de compliqué pour réaliser cette fonction, il suffit de parcourir entièrement l'image et d'appliquer pour chacun des pixels une nouvelle valeur utilisant les valeurs de rouge, vert et bleu de chaque pixel. Les coefficients appliqués aux valeurs rouge, vert et bleu sont respectivement 0.3, 0.59, 0.11. Cela permet de créer un nouveau pixel avec cette nouvelle valeur pour sa valeur rouge, vert et bleu (pour obtenir un niveau de gris).

Pour utiliser cette fonctionnalité dans notre programme, il suffit d'exécuter

```
./Lambda grayscale "image_path" "destination_path"
```

Et voici le résultat obtenu une fois le programme exécuté

## 2.3 Binarisation

Pour binariser l'image (c'est à dire ne laisser que des pixels blancs ou noirs), nous avons appliqué une condition simple sur chacun des pixels qui compose l'image. Nous faisons la moyenne des valeurs rouge, vert et bleu et vérifions sur cette dernière est supérieure ou non à 127. Si elle est supérieure, nous mettons un pixel blanc et un noir sinon. Il n'existe pas de commande pour appliquer ce filtre dans notre exécutable.

Ce n'est pas ici la fonction la plus compliquée de notre code mais ce sont des éléments essentiels pour la suite.

## 2.4 Augmentation des contrastes

Sur certaines images, les contrastes ne sont pas assez prononcés et il faut alors les augmenter. Pour cela, nous avons créé une fonction utilisant un histogramme normalisé de tous les pixels d'une image en niveaux de gris. L'histogramme est représenté par une liste de taille 256.

Comme l'image traitée est en niveau de gris, nous construisons l'histogramme en parcourant chaque pixel et en ajoutant la valeur rouge du pixel par exemple (comme les valeurs de rouge, bleu et vert sont les mêmes).

Ensuite, l'objectif est de trouver une nouvelle valeur pour chaque pixel de l'image. Pour cela, nous repassons à nouveau dans chaque pixel et utilisons la formule :

$$\sum_{n=0}^{k} \frac{255}{nbPixels} * histogram[n]$$

qui correspond à la somme du nombre de pixels de l'image dont la valeur est inférieure ou égale au pixel traité. Nous multiplions ensuite cette donnée par $\frac{255}{nbPixels}$ pour normaliser et obtenir la nouvelle valeur de gris du pixel.

## 2.5 Réduction des bruits

Nous n'appliquons ce filtre que sur des images en niveaux de gris.

Pour réduire les bruits sur l'image, nous suivons un algorithme qui s'applique sur chaque pixel. Nous faisons la somme des valeurs des pixels voisins (les 8 pixels qui entourent le pixel actuel s'ils existent). La valeur du nouveau pixel est la moyenne des valeurs des pixels voisins.

Voici la fonction qui est utilisée dans notre code

```
// Foreach pixels
  for (int i = 0; i < img->w; i++)
  {
      for (int j = 0; j < img->h; j++)
      {
          // Sum of each neighbors pixels
          Uint32 sum = 0;
          short count = 0;
          for (int k = -1; k < 2; k++)
          {
              for (int l = -1; l < 2; l++)
              {
                  if (i + k > -1 &&
                      i + k < img->w &&
                      j + l > - 1 &&
                      j + l < img->h)
                  {
                      pixel = getpixel(img, i + k, j + l);
                      SDL_GetRGB(pixel, img->format, &r, &g, &b);
                      sum += r;
                      count += 1;
                  }
              }
          }
          // Average of pixels values to get the new current pixel value
          sum /= count;
          pixel = SDL_MapRGB(img->format, sum, sum, sum);
          putpixel(imgCopy, i, j, pixel);
      }
  }
```

# 3 Segmentation de l'image

## 3.1 Introduction

Ce premier mois de projet nous a permis de commencer à implémenter des fonctionnalités primordiales pour la réalisation de notre projet final. En effet nous avons réalisé la segmentation des caractères, à reconnaître par la suite.
Ce processus est complexe et crucial pour pouvoir fournir au programme de reconnaissance des caractères bien découpés et formalisés comme le dataset utilisé pour l'entraîner.

## 3.2 Segmentation de caractères

Comme dit précédemment, la segmentation du texte initial en caractères reconnaissables par le programme est une des parties sur laquelle nous nous sommes le plus penché durant ce premier mois.

### 3.2.1 Découpage des colonnes

Si la détection des caractères est primordiale, elle n'est que le point final du chaîne d'opérations sur l'image initiale. La première étape du processus de segmentation est le découpage des colonnes. Par exemple, l'image présentée ci-dessous comporte deux colonnes, qu'il faut séparer pour pouvoir continuer le processus de segmentation.

*IMAGEparagraph.png*

La détection des colonnes est un processus assez simple, qui tient compte de la "logique" d'organisation d'un texte. En effet, en tant qu'humains, nous distinguons deux colonnes comme des "colonnes" et non une suite de mots par l'espace situé entre les deux blocs qui est anormalement grand par rapport aux espaces présents entre les mots du texte.

Ainsi, j'ai implémenté la même logique pour la segmentation, qui détecte un espace rempli de pixels blancs anormalement grand entre deux parties ne contenant pas uniquement des pixels blancs. Voici un aperçu du code implémenté :

```
1
2     unsigned char fullWhite = 1;
3     unsigned char firstCut = 1;
4     int endText = -1; //Gets the first pixel (width wise) with full white width
5     int beginingText = -1; //Gets the first pixel without full white height after several full white
6     lastLineWidth = 0.03 * img -> w;
7
8     for (int i = 0; i < img -> w; i++)
9     {
10        // Returns whether img's height is full of white pixels or not
11        fullWhite = fullWhite(img, i);
12
13        if (!fullWhite && firstCut)
14        {
15            // Begins column cut
16            beginingText = i;
17            if (endText == -1 || lastLineWidth <= abs(endText - beginingText)){
18                //Draw line to separate column
19                for (int k = 0; k < img -> h; k++)
20                {
21                    pixel = SDL_MapRGB(img_copy -> format, 0, 255, 0);
22                    putpixel(img_copy, beginingText, k, pixel);
23                }
24            }
25            firstCut = 0;
26        }
27
28        if(fullWhite && !firstCut) {
```

```
29
30            // Ends column cut
31            endText = i-1;
32            int ii = i;
33            do
34            {
35                fullWhite = fullWhite(img, ii);
36                ii++;
37            } while (fullWhite);
38
39            if (ii - endText >= lastLineWidth || ii >= img -> w) {
40                // Space is too big, draw line to separate
41                for (int k = 0; k < img -> h; k++)
42                {
43                    pixel = SDL_MapRGB(img_copy -> format, 0, 255, 0);
44                    putpixel(img_copy, i, k, pixel);
45                }
46            }
47            firstCut = 1;
48        }
49    }
```

Le principe est le suivant : on possède une variable *fullWhite* qui indique si, pour un pixel donné (sur la largeur), tous les pixels sur la hauteur de l'image sont blancs. Cette variable est très importante, car combinée à *firstCut*, on peut savoir si la partie de l'image où l'on se situe est du texte ou non. *firstCut* permet de savoir si nous sommes en train de chercher le début d'une colonne (1 dans ce cas), ou si le début la colonne est déjà repéré et qu'on cherche donc la fin.

Ainsi le code devient plus compréhensible. En parcourant toute la largeur de l'image, on regarde à chaque fois si toute la hauteur de l'image est blanche.

Si c'est n'est pas cas et que nous cherchons le début d'une colonne, alors nous l'avons trouvé et pouvons dessiner une ligne verticale pour marquer le début de la colonne.

Si cependant nous ne cherchons plus le début mais la fin d'une colonne et qu'on se trouve sur un pixel dont la hauteur de l'image est toute blanche, alors nous évaluons les possibilités que ce soit la fin de la colonne. On note *endText* comme l'indice final temporaire du texte. On continue d'avancer dans la largeur de l'image jusqu'au prochain pixel qui n'est pas *fullWhite*. Enfin, la distance entre ce nouveau pixel et *endText* est comparée à une constant représentant 3% de la largeur de l'image. Si l'écart est supérieur à cette constante, on considère l'espace comme trop grand pour être un espace entre deux mots et on coupe la fin de la colonne au niveau de *endText*.

Le résultat de ce programme sur la photo précédente donne l'image ci-dessous, comportant en rouge les lignes tracées par la programme pour délimiter les colonnes, qui peuvent maintenant être coupées :

*IMAGE_AVEC_LES_LIGNES*

Comme vous pouvez le voir, quatre lignes sont créées : la première correspond au début de la première question, la deuxième à la fin de cette même colonne. Les deux autres représentent respectivement le début et la fin de la deuxième colonne. Bien entendu, ici, les colonnes sont juste découpées par des lignes, mais il faudrait les séparer en deux images différentes, ce qui est fait par une autre fonction de notre programme.

### 3.2.2 Découpage des paragraphes

Cette partie est la suite logique du découpage des colonnes. En effet, en regardant l'image 1 précédemment affichée, chaque colonne contient plusieurs paragraphes nettement distinguables.

Ainsi, un programme similaire est utilisé sur chaque colonne trouvée pour découper les différents paragraphes afin qu'ils soient traités plus tard.

Ici, le changement est principalement basé sur le fait de regarder si la largeur de l'image est complètement blanche, contrairement à la hauteur pour les colonnes. De plus, on détermine ici si un paragraphe se termine en fonction de la "hauteur" de la dernière ligne/du dernier paragraphe trouvé. Voici le nouveau code pour ce type de segmentation :

```

    unsigned char fullWhite = 1;
    unsigned char firstCut = 1;
    int endText = -1; //Gets the first pixel (height wise) with full white width
    int beginingText = -1; //Gets the first pixel without full white width after several full white
    int lastLineHeight = -1; //Stores the number of pixels from the last text line

    for (int i = 0; i < img -> w; i++)
    {
        // Returns whether img's height is full of white pixels or not
        fullWhite = fullWhite(img, i);

        if (!fullWhite && firstCut)
        {
            // Begins column cut
            beginingText = i;
            if (endText == -1 || (lastLineHeight <= abs(endText - beginingText)){
                for (int k = 0; k < img -> w; k++)
                {
                    pixel = SDL_MapRGB(img_copy -> format, 0, 255, 0);
                    putpixel(img_copy, k, beginingText, pixel);
                }
            }
            firstCut = 0;
        }

        if(fullWhite && !firstCut) {

            // Ends column cut
            endText = i-1;
            lastLineHeight = endText - beginingText;
            int ii = i;
            do
            {
                fullWhite = fullWhite(img, ii);
                ii++;
            } while (fullWhite);

            if (ii - endText >= lastLineWidth || ii >= img -> h) {
                // Space is too big, draw line to separate
                for (int k = 0; k < img -> w; k++)
                {
                    pixel = SDL_MapRGB(img_copy -> format, 0, 255, 0);
                    putpixel(img_copy, k, i, pixel);
                }
            }
            firstCut = 1;
        }
    }
```

Le résultat de ce nouveau programme, combiné à une fonction qui retire les lignes vertes créées, découpe et stocke les paragraphes dans de nouvelles images donne le résultat suivant :

*IMAGES_DES_PARAGRAPHES_DECOUPES*

Les paragraphes sont parfaitement découpés et près à être traités par la suite du programme jusqu'à pouvoir arriver à détecter des caractères.

### 3.2.3 Découpage des lignes

Le découpage des mots est l'étape suivante que nous avons créée. En effet, les paragraphes sont des blocs de lignes qu'il faut séparer.

Cette partie ressemble très fortement, pour ne pas dire exactement, à la partie précédente. La seule différence apportée se situe au niveau de la tolérance de hauteur entre deux lignes. Pour détecter si on a bien affaire à un changement de ligne, il nous faut regarder l'espace entre la fin supposée de la ligne précédente et le début de la nouvelle. Si cet écart est trop faible, alors on ne considère pas la fin de ligne supposée comme une fin de ligne.

C'est justement le degré considération qui change par rapport au découpage des paragraphes. Ici, on considère un changement si l'écart détecter entre le début et la fin des supposées lignes est supérieur à 0.2 fois la hauteur d'une ligne, comparé à la hauteur elle-même pour la détection de paragraphes.

Dans le code, cela revient à changer les conditions des lignes 17 et 39, on remplaçant *lastLineHeight* par *lastLineHeight * 0.2*. Tout comme le découpage des paragraphes, on dessine les lignes séparatrices puis découpe les lignes pour les stocker dans un dossier dédié. Voici le résultat du découpage des lignes par notre programme :

*IMAGES_DES_LIGNES_DECOUPES*

### 3.2.4 Découpage des mots et des caractères

Nous présenter le découpage des mots et des caractères ensemble car ils sont effectués plus ou moins en même temps. En effet, le découpage des caractères reste une étape très délicate à appréhender car elle nécessite de prévoir ou doivent être découpés les caractères.

L'ordre des segmentations voudrait que les mots soient découpés en premiers, puis les caractères ensuite. Au final, c'est plus ou moins ce qui se passe : on dessine les lignes séparant chaque caractère, puis on découpe les mots de la même façon que l'on découpe les colonnes, et enfin on découpe les caractères en utilisant les lignes dessinées auparavant, pour chacun des mots.

Pourquoi faisons nous ces deux découpages en même temps ? Tout simplement pour une question de précision sur le découpage des caractères. En effet, notre découpage des caractères se fait en fonction de la largeur moyenne d'un caractère de la phrase en cours de découpage.

Pour être encore plus clair, voici ce qu'il se passe précisément. Nous définissons la largeur moyenne d'un caractère comme *0.5 * (image -> h)*, soit la moitié de la hauteur de l'image (résultat basé sur une comparaison des largeurs moyennes de caractères pour plusieurs polices, celles-ci étant centrées autour de cette valeur). Ensuite, nous parcourons l'image de gauche à droite, c'est-à-dire que nous cherchons le début d'un caractère (tous les pixels ne sont pas blancs sur la hauteur de l'image), puis nous cherchons sa fin (tous les pixels sont blancs). Cette fin n'est pas forcément la bonne, car deux caractères peuvent se chevaucher.

C'est alors qu'intervient notre variable de largeur moyenne d'un caractère. On regarde quelle largeur fait le supposé caractère sélectionné. S'il est x fois plus large que la moyenne, alors nous considérons que x caractères se chevauchent, et on coupe le bloc en x caractères différents. La moyenne est alors modifiée en fonction de la taille des caractères découpés.

Voici pourquoi nous faisons le découpage des caractères avant celui des mots, car plus la phrase contient de caractères, plus on peut être précis sur la façon de les découper.

Voici à quoi ressemble le programme final pour le découpage des caractères :

```
for (int j = 0; j < img -> w; j++)
  {
      fullWhite = fullWhiteHeight(img, j);
      if (!fullWhite && firstCut)
      {
          beginingCharPixel = j;

          /* Draw line for cut */

          firstCut = 0;
      }

      if(fullWhite && !firstCut)
      {
          endingCharPixel = j-1;
          int actualCharLength = abs(endingCharPixel - beginingCharPixel);
          firstCut = 1;

          /* Draw line for cut */

          // Bloc is too long
          if (actualCharLength > 2*averageCharLength)
          {
              int times = round(actualCharLength/actualCharLength);
              int middle = actualCharLength/2 + beginingCharPixel;
              for (char k = 1; k <= times; k++){

                  /* Draw line at beginingCharPixel + k*actualCharLength/times for cut */

              }
              nbChars += times;
              averageCharLength = (averageCharLength*(nbChars-times) + actualCharLength)/nbChars;
          }
          else {
              nbChars += 1;
              averageCharLength = (averageCharLength*(nbChars-1) + actualCharLength)/nbChars;
          }
      }
  }
```

Comme dit précédemment, le découpage des mots est effectué après avoir dessiné les lignes pour celui des caractères. Ceci n'a pas d'impact sur son efficacité, et peut même aider au découpage car les espaces blancs entre les caractères ont été remplacés ou réduits avec les nouvelles lignes dessinées.

La même technique est utilisée pour découper colonnes et mots, on regarde quelle taille fait l'espace entre la supposée fin d'un mot et le début de l'autre. Après avoir fait des recherches, un espace représente environ 25% de la hauteur des caractères. On utilise ainsi cette donnée pour vérifié si l'espace trouvé est supérieur à 22% de la hauteur de l'image. Cet algorithme fonctionne très bien est découpe brillamment les mots. En voici un exemple :

*IMAGES_DES_MOTS_DECOUPES*

# 4 Réseau de neurones

## 4.1 Avancées

### 4.1.1 Fonction XOR

La première étape pour obtenir un bon réseau de neurones était de développer un réseau capable d'apprendre la fonction XOR. Cela a pour avantage que le réseau serait rapide à entraîner, le réseau n'a que 2 entrées pour une seule sortie, et seulement 4 motifs d'entrainements.

| | entrée 1 | entrée 2 | sortie |
|---|---|---|---|
| Motif 1 | 0 | 0 | 0 |
| Motif 2 | 0 | 1 | 1 |
| Motif 3 | 1 | 0 | 1 |
| Motif 4 | 1 | 1 | 0 |

N'étant alors pas tout à fait à l'aise avec ce ńănouveauăż langage qu'est le C, j'ai d'abord cherché à faire ce réseau en Python, un langage que je maîtrise depuis plusieurs années. J'ai donc cherché de l'aide et des exemples de réseaux de neurones par rétropropagation en Python, mais la plupart des résultats utilisaient le module Numpy pour la manipulation de matrice ainsi que certains calculs, or, en C, tout cela doit être traité manuellement.

Après plusieurs heures de recherche, je suis tombé sur un très bon tutoriel qui expliquait en détail le fonctionnement d'un tel programme, avec des exemples en C++. J'ai facilement réussi à refaire le programme en Python, puis en C.

Ensuite, pour voir si un tel réseau était adaptable à d'autres algorithmes, j'ai essayé de lui faire apprendre d'autres motifs, comme une fonction qui à partir de trois entrées, renvoie 1 si au moins 2 sont à 1 (et 0 sinon).

### 4.1.2 Reconnaissance de caractères

Une seconde étape était de prévoir une structure pour notre réseau de neurones. Au final, le réseau qui apprendra à reconnaître des caractères est un réseau semblable à notre réseau XOR, avec un nombre d'entrées égal à nombre de pixel dans une image, et le nombre de sorties serait égal au nombre de caractères à reconnaître.

Les entrées sont des doubles compris entre 0 et 1, un 0 correspond à un pixel dont la couleur est celle du fond de l'image, un 1 correspond à un pixel de la couleur d'écriture du caractère.

Les images qui lui seront transmises auront une taille de $32 \times 32$ et seront collées au coin supérieur gauche. Le réseau a donc 1024 entrées.

Le nombre de sorties est égal au nombre de caractères à reconnaître. On suppose donc que la sortie ayant la plus grande valeur correspond au caractère reconnu.

Dès la réception du cahier des charges le 10 septembre, j'ai écris un programme C qui à l'aide de SDL2, lit des images et enregistre la valeur des pixels dans un tableau. Ce programme est encore utilisé aujourd'hui, mais sera surement modifié d'ici à la fin du projet pour mieux correspondre avec le traitement de l'image effectué par mes camarades.

Ensuite, il nous fallait un catalogue d'images (dataset) qui pourrait être utilisé pour l'entraînement et les tests du réseau. Les catalogues trouvés sur internet étaient très limités (taille des images ne correspondant pas, trop peu d'images,), la meilleure solution était de créer nos propres datasets.

Pour cela, j'ai écris un petit script en Python qui génère des images avec un caractère dessus (grâce à la bibliothèque PIL). La police, couleur et taille sont aléatoires afin d'obtenir un entraînement plus diversifié.

Le premier réseau devait apprendre à reconnaître les caractères numériques 0 à 9. Les résultats n'étaient au début pas très probants. Et pour cause : les poids et biais des neurones étaient mal initialisés (Ils étaient initialisés entre 0 et 1 au lieu de entre -0.5 et 0.5). Une fois ce problème corrigé, le réseau arrivait à apprendre et reconnaissait les caractères avec une précision de 67% environ. Deux problèmes se posaient alors :

— Le réseau ne montait pas au dessus de 67% de réussite

— Il avait besoin de beaucoup de neurones (plus de 1000) dans sa couche intermédiaire.

Ces problèmes m'ont pas mal pris la tête, et m'ont obligé à revoir l'ensemble du programme. Au bout de deux semaines de migraines, l'origine du problème m'est apparue : la dérivée de la fonction Sigmoïde était mal implémentée.

La dérivée de la sigmoïde s'écrit comme ceci :

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Or, les paramètres récupérés et qui passent dans cette fonction sont déjà $\sigma(x)$. Du coup, la dérivée était au final calculée comme $\sigma^2(x)(1 - \sigma^2(x))$

Une fois ce problème résolu, on a obtenu des résultats proches de 95% pour une couche intermédiaire de seulement 760 neurones.

## 4.2 Fonctionnement

### 4.2.1 Scripting

Une fois que j'étais sûr que le réseau était bien en train d'apprendre, je me suis chargé du scripting, c'est à dire de créer des structs et des fonctions qui permettent d'interagir avec le réseau plutôt que de tout avoir dans un même fichier .c.

Le réseau de neurones tourne donc autour de deux structs :

— MMImage est une struct qui contient toutes les informations caractérisant une image pour une prédiction (dimensions de l'image, valeurs des pixels, ) ou le training (caractère que c'est vraiment, )

— MMNetwork, qui contient les informations à propos d'un réseau de neurones (nombre d'entrées, sorties, neurones, les poids et les biais)

J'ai implémenté des fonctions autour de ces structs, comme la fonction LoadDataset, qui permet de charger un dataset complet en mémoire, ou LoadImage qui permet de ne charger qu'une seule image.

```
1 MMImage*  LoadDataset(int noChars, int imagesPerChar);
2 MMImage  LoadImage(char* path);
```

Concernant MMNetwork, on peut initialiser un réseau grâce à la fonction InitNetwork. Pour libérer la mémoire, on a la fonction DestroyNetwork.

```
1 MMNetwork InitNetwork(unsigned int numInputs, unsigned short numHiddenNodes, unsigned int numOutputs
      );
2 void DestroyNetwork(struct MMNetwork n);
```

L'avantage de MMNetwork, c'est qu'il peut être enregistré et chargé dans des fichiers. Cela sert notamment à sauvegarder l'état d'un réseau entraîné pour pouvoir le récupérer par la suite pour faire des prédictions.

```
1 void SaveNetwork(MMNetwork n, char* path);
2 MMNetwork LoadNetwork(char* path);
```

À partir de là, les bases sont posées pour effectuer n'importe quel type d'opérations avec les réseaux. En effet, une simple prédiction peut se faire avec la fonction Predict, qui à partir d'un MMImage et d'un MMNetwork, renvoie les valeurs de la couche de sortie du réseau, qui peut être analysée par la fonction OutputChar.

```
1 double* Predict(MMNetwork network, const MMImage* image);
2 char OutputChar(double* outputLayer);
```

**Exemple de programme :**

Analysons maintenant le programme suivant, qui permet de prédire le caractère présent sur une image :

```
1  #include <stdio.h>
2  #include "LambdaNeuralNetwork.h"
3
4  int main(int argc, const char* argv[]) {
5
6    MMImage img = LoadImage("/Users/maxime/Documents/OCR/W.bmp");
7    MMNetwork n = LoadNetwork("/Users/maxime/Documents/OCR/IA");
8
9    double* output = Predict(n, &img);
10   char character = OutputChar(output);
11
12   printf("Recognized as a %d\n", character);
13   printf("\n");
14
15   return 0;
16 }
```

— Ligne 2 : On inclut le header principal, qui définit toutes les fonctions et structs utilisées pour la prédiction et l'utilisation des réseaux de neurones du projet.

— Ligne 6 : On charge une image à partir de son chemin d'accès dans une variable img

— Ligne 7 : On charge le réseau précédemment entraîné à partir du chemin où il a été enregistré dans une variable n

— Ligne 9 : On fait une prédiction sur l'image img par le réseau n. Les valeurs de la couche de sortie sont enregistrées dans une variable output.

— Ligne 10 : À partir de la couche de sortie output, on récupère le caractère correspondant. C'est ce caractère qui est écrit sur l'image.

Juste ces 4 lignes permettent de faire une prédiction sur le caractère écrit sur une image.

### 4.2.2 Compilation

Le réseau de neurones a été entièrement programmé sur l'IDE Xcode, et a été organisé en 3 projets :

— La library "LambdaNeuralNetwork" qui contient les fonctions et les structs

— Un programme "Learning" qui s'occupe de l'entraînement du réseau

— Un programme "Analyze" qui va effectuer des prédiction sur un testing-set afin d'obtenir un taux de réussite sur un réseau après entraînement

Un Makefile permet de compiler chacun de ces projets :

```
make lib
make analyze
make learning

make all
make clean
```

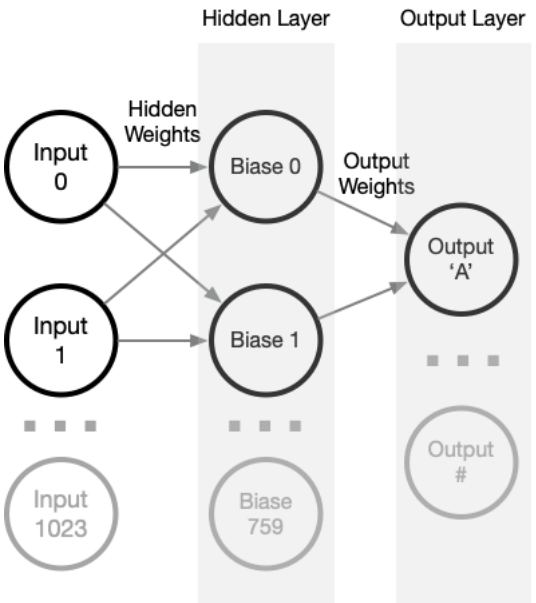
Effectuer un make pour compiler un des programme va créer l'exécutable correspondant dans un dossier bin. Compiler la lib va créer un dossier lib contenant des fichiers .o. Le fichier principal LambdaNeuralNetwork.o est le lien vers tous les autres .o.

Importer le fichier LambdaNeuralNetwork.o et include le fichier "include/LambdaNeuralNetwork.h" suffit pour pouvoir utiliser toutes les fonction et les structs.

### 4.2.3 Fonctionnement du réseau

L'apprentissage du réseau utilise un algorithme du gardient stochastique (Stochastic Gradient Descent), une "variante" de la descente du gradient qui a pour particularité de l'utiliser qu'un seul élément (en l'occurrence une image) choisi aléatoirement du training-set à la fois pour modifier les poids. (Un algorithme standard de descente du gradient utilisera tous les éléments en même temps).

Voici un schéma simplifié de notre réseau. (Le réseau comporte 1024 entrées, 760 neurones cachés, et 78 sorties. Pour des raisons de clarté, je n'ai représenté que certains d'entre eux.)

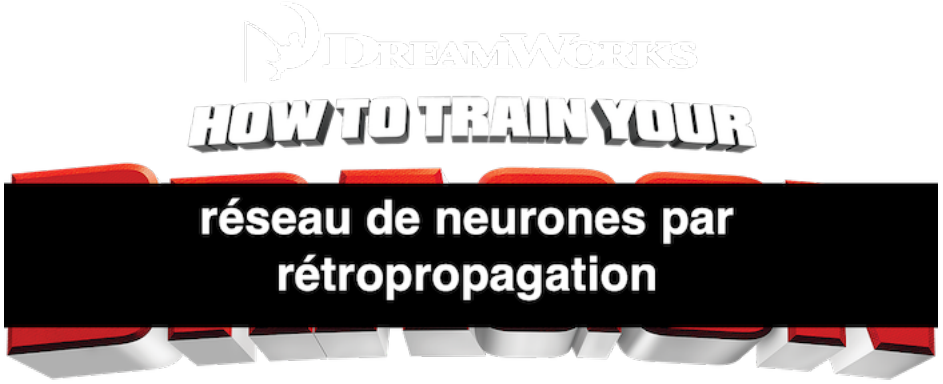


Chaque entrée influera sur chaque neurone caché par lintermédiaire dun poids caché (hidden weight) et du biais du neurone caché. De même, chaque neurone caché influera sur chaque neurone de sortie par lintermédiaire dun poids de sortie (output weight) et du biais du neurone de sortie. À chaque itération de lapprentissage, une valeur est calculée pour chaque neurone caché :

$hiddenLayer_j = \sigma(hiddenLayerBias_j + \sum_i input_i \times hiddenWeight_{i \to j})$
(où $hiddenLayerBias_j$ est le biais du neurone, $hiddenWeight_{i \to j}$ le poids de la transition de lentrée i vers le neurone)

Idem pour les valeurs de la couche de sortie :

$outputLayer_k = \sigma(outputLayerBias_k + \sum_i hiddenLayer_j \times outputWeight_{j \to k})$
(où $outputLayerBias_k$ est le biais du neurone, $outputWeight_{j \to k}$ le poids de la transition du neurone caché i vers le neurone de sortie)



# 5 Correcteur orthographique

## 5.1 L'objectif

Notre but ici était de pouvoir corriger les fautes dans un texte scanné par lOCR et avoir une vérification supplémentaire de ce qua renvoyé le réseau de neurones.

## 5.2 La ponctuation

Pour vérifier la ponctuation dun texte jai opté pour un ensemble de fonction vérifiant chacune des règles particulières de la ponctuation de la langue française. Par exemple, le code ci-dessous vérifie sil y a bien un espace après le point et une majuscule.

Il existe deux fonctions similaires respectivement pour les règles du point dinterrogation et celui dexclamation. Ces fonctions ont pour vocation dêtre utilisé avec un main qui leur enverrait lindex des points dans un texte.

## 5.3 La cohérence

Cette fonction a vu le jour pour palier un problème du réseau de neurones. En effet, il pouvait arriver que lIA confonde une majuscule avec une minuscule. La fonction ci-dessous vérifie alors sil y a plus de lettre majuscule ou minuscule dans le mot et change le tout. Cette fonction nous apporte une sécurité supplémentaire.

## 5.4 La distance de Levensthein

Pour concevoir un correcteur orthographique assez rapide pour être utilisé, nous avons dû faire de nombreuses recherches qui nous ont conduit vers la matrice de Levenshtein. Cette dernière est la méthode de calcul la plus rapide que nous avons pu implémenter et elle compare nimporte quel mot avec tous les autres du dictionnaire français (336 524 mots au total) en 0.5 seconde. A lheure actuelle, le correcteur orthographique est utilisable pour corriger 1 mot à la fois. Cependant, le temps actuel bien que déjà très bon nest pas suffisamment faible pour pouvoir être utilisé à une plus grande échelle. Nous avons donc pour objectif pour la prochaine soutenance daméliorer ce temps notamment par lutilisation dune table de hachage ou bien par la comparaison de triplet de caractères et pour pouvoir améliorer nos chances de réussite de correction dun mot dans le même temps. La plus grande difficulté pour nous aura été de trouver un moyen de contourner lutilisation dune IA et dapporter des solutions de correction viable sans lutilisation des habitudes de lutilisateur.

# 6 Command parser

## 6.1 L'objectif

Notre but ici était de permettre l'utilisation de notre programme via des lignes de commandes dans un premier temps avant de réaliser l'interface graphique.

Nous avons donc ajouté 3 commandes qui permettent d'exécuter des fonctionnalités clés telles que nos filtres, notre niveau de gris ainsi que la segmentation d'une image quelconque.

## 6.2 Grayscale

C'est une fonction qui a besoin de 2 paramètres, le premier est le chemin de l'image d'origine et le deuxième le chemin auquel l'image en niveau de gris sera sauvegardée. Si les deux paramètres ne sont pas entrés ou bien que l'image entrée en paramètre ne peut être chargée, le programme s'arrête et renvoie un message d'erreur.

```
1
2   if (argc == 4) {
3     SDL_Surface *imgDefault;
4     imgDefault = SDL_LoadBMP(argv[2]);
5     if (!imgDefault) {
6       printf("Error: unable to find bmp file at %s\n", argv[2]);
7       return 1;
8     }
9     imgDefault = grayscale(imgDefault, 1, argv[3]);
10    return 0;
11  }
12  else {
13    printf("Lambda: Grayscale take exactly 2 parameters but was called with %i parameter(s)\n", argc
          - 2);
14    return 1;
15  }
```

Ici, l'objectif est de couvrir tous les cas d'utilisation et d'éviter les crash de notre programme. Ainsi la fonction n'est appelée uniquement lorsque tous les paramètres sont entrés et la fonction grayscale renvoi un booléen indiquant qu'une erreur s'est produite pendant l'exécution et ainsi permet au parser de prévenir l'utilisateur.

## 6.3 Filtres

Il fallait ajouter la possibilité d'utiliser les filtres de notre programme. Pour cela, nous avons créé la fonction filters à notre programme. Celle ci applique une augmentation des contrastes de façon systématique et la reduction de bruits si le 3ème paramètre est vrai.

De la même façon que les autres commandes, il ne faut pas faire crasher le programme de façon involontaire. C'est pour cela que nous avons fait en sorte de sortir des exceptions et d'arrêter le programme dès qu'une erreur peut se produire. Voici le code pour cette commande :

```
1   if (argc == 4 || argc == 5) {
2       SDL_Surface *imgDefault;
3       imgDefault = SDL_LoadBMP(argv[2]);
4       if (!imgDefault) {
5         printf("Error: unable to find bmp file at %s\n", argv[2]);
6         return 1;
7       }
8       if (argc == 5 && strcmp(argv[4], "true") == 0)
9       {
10        imgDefault = contrast(imgDefault);
```

```
11        printf("Lambda: Noise reduction ended successfully\n");
12      }
13
14      imgDefault = contrast(imgDefault);
15      printf("Lambda: Contrast ended successfully\n");
16      SDL_SaveBMP(imgDefault, argv[3]);
17      return 0;
18    }
19    else {
20      printf("Lambda: Filters take 1 or 2 paramaters but was called with %i parameter(s)\n", argc -
            2);
21      return 1;
22    }
```

## 6.4 Segmentation

### 6.4.1 Segmentation complète

La fonction de segmentation s'exécute avec simplement un paramètre : le chemin de l'image qu'il faut segmenter. Pour le reste, tout est géré via une architecture de dossiers et fichiers précise, l'utilisateur n'a pour l'instant pas le choix de ce dernier.

```
1   if (argc == 3) {
2     if (!fullSegmentation(argv[2])) {
3       printf("Lambda: Error during segmentation execution\n");
4       return 1;
5     }
6     printf("Lambda: Segmentation ended successfully\n");
7     return 0;
8   }
9   else {
10    printf("Lambda: Segmentation take exactly 1 paramater but was called with %i parameter(s)\n",
          argc - 2);
11    return 1;
12  }
```

### 6.4.2 Segmentation d'une ligne

Notre programme gérant le multi-colonnes, nous avons dans l'idée d'ajouter une option pour que l'utilisateur indique si oui ou non l'image entrée est sur une colonne ou sur plusieurs. Voici la fonction qui gère ce cas :

```
1   if (argc == 3) {
2       SDL_Surface *img;
3       img = SDL_LoadBMP(argv[2]);
4       img = cutCharacters(img, "exampleChars/");
5       if (!img) {
6         printf("Lambda: Error during segmentation execution\n");
7         return 1;
8       }
9       removeLinesForCharacters(img, "chars/");
10      printf("Lambda: Segmentation ended successfully\n");
11      return 0;
12    }
13    else {
14      printf("Lambda: Characters take exactly 1 paramater but was called with %i parameter(s)\n",
            argc - 2);
15      return 1;
16    }
```

# 7 Notre avancement et répartition des tâches

## 7.1 Répartition des tâches à la première soutenance

Nous vous avons présenté les fonctionnalités de notre projet sans dire qui était en charge de ces dernières. Ce tableau est le récapitulatif global de nos tâches respectives pour la première soutenance, le X signifie que la personne a participé à cette tâche. L'objectif ici est de clarifier nos rôles, nous formons un seul groupe qui progresse et se donne des objectifs en fonction des possibilités et envies de chacun. Le nombre de tâches réalisées n'a pas de rapport avec l'investissement dans le projet car la complexité est différente.

| | Quentin | Maxime | Charles | Nathan |
|---|---|---|---|---|
| Pré-traitement | X | | X | |
| Segmentation de l'image | X | | X | |
| Réseau de neurones | | X | | |
| Correcteur orthographique | | | | X |
| Command parser | | | X | |

## 7.2 Nos objectifs et envies

La deuxième soutenance sera la dernière pour ce projet, il faut donc qu'il soit terminé d'ici là. Nous devons donc mettre tout en oeuvre pour lier la segmentation en caractères de notre image source avec la reconnaissance de ces derniers pour permettre à l'utilisateur de récupérer son texte.

Il nous faut aussi commencer l'interface graphique pour donner vie à notre projet et lui faire dépasser le stade des lignes de commandes. Nous avons déjà commencé à utiliser pour GTK et Glade pour créer cette dernière et elle sera terminée pour la deuxième soutenance. L'objectif est créer une interface en adéquation avec le style graphique que nous avons mis au point pour le groupe.

La finalisation du correcteur orthographique est une étape importante pour le projet. Elle ajoutera une fonctionnalité utile pour l'utilisateur. Cela nous apportera beaucoup dans la compréhension de l'implémentation d'un correcteur orthographique et sa réalisation et nous fera réfléchir à l'optimisation de notre programme pour ne pas perdre trop de temps sur la correction.

L'IA doit encore être entrainée pour avoir un taux d'erreur le plus bas possible et reconnaître au mieux les caractères de l'image. Il faut penser à la reconnaissance des accents, virgules, points, ...

Nous devons aussi penser à améliorer notre segmentation des caractères pour éviter les problèmes auxquels nous sommes confrontés aujourd'hui et peut être créer un réseau de neurones qui pourrait apprendre à reconnaître des caractères sur une ligne grâce si nous avons le temps.

## 7.3 Utilisation du programme

Il faut d'abord générer notre exécutable en entrant la commande make dans le dossier parent de notre projet. Le nom de l'exécutable est Lambda il faut donc utiliser ./Lambda pour commencer à utiliser notre programme. Voici la liste des fonctions qui sont utilisables :

```
./Lambda grayscale "image_path" "destination"
// Appliquer le filtre de niveaux de gris sur l'image et l'enregistrer dans la destination
```

```
./Lambda filters "image_path" "destination" true
// Applique l'augmentation de contrastes sur l'image et la sauvegarde dans la destination
// Dernier argument optionnel: permet d'appliquer la reduction de bruits
```

```
./Lambda segmentation "image_path"
// Applique la segmentation complète et génère l'architecture de dossiers/fichiers
```

```
./Lambda characters "image_path"
// Applique la segmentation partielle et génère l'architecture de dossiers/fichiers
```

# 8 Conclusion