suite. La solution retenue à été d'avoir un tableau de positions des indices de début et de fin pour chaque caractères. Ainsi, lorsque le début d'un caractère est détecté, sa position est enregistrée dans un tableau, de même pour la fin d'une détection. Ainsi, le problème de perte de précision n'en n'est plus un.

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 :

Dans ce même dossier se trouvent des dossiers pour les caractères de chaque mots contenus qui contiennent les images en 32x32 pixels avec le caractère positionné en haut à gauche de l'image. Voici donc l'architecture finale de notre projet :

Cette architecture était une première version pour ce projet, mais nous avons finalement modifier cette dernière, notamment après avoir modifier notre segmentation des caractères pour quelque chose de plus précis. Nous avons au final très peu d'image qui sont sauvegardées après l'exécution du projet, seulement les colonnes et les paragraphes et quelques images d'analyse.

Cependant, nous sauvegardons temporairement les images telles que les lignes, les mots et les caractères découpés. En effet, lorsque le projet s'exécute, des dossiers temporaires sont créés et contiennent quelques informations utiles à certaines fonctions, notamment pour récupérer les caractères à envoyer au réseau de neurones. Ces dossiers sont supprimés une fois la compilation terminée.

#### 4.2.5 Gestion du texte en italique

Nous avons constaté que la gestion d'un texte en italique était plus complexe et demandait une approche différente que celle d'un texte classique. Nous avons donc réfléchit à quelles étaient les meilleures options pour pouvoir découper les caractères même s'ils sont en italique.

Nous nous sommes tout de suite concentrés sur le fait de ne pas dessiner sur l'image pour ensuite découper les caractères à l'aide de ces marques, comme nous avions pu le faire pour les autres types de segmentation. Nous avons donc repris le système de pointeurs, implémenté pour la segmentation classique, qui stocke le début et la fin d'un caractère dans une tableau, nous permettant ensuite de découper et enregistrer le caractère dans une nouvelle petite image. Cette nouvelle image sera ensuite envoyée dans le réseau de neurones pour reconnaître le caractère écrit.

Pour ce qui est de détecter le début et la fin d'un caractère en italique, nous avons tout simplement repris le système codé pour une écriture classique et nous avons fait en sorte qu'il soit incliné. C'est-à-dire que nous vérifions que l'image est toute blanche en hauteur sur un pixel défini, tout ceci en fonction de ce qui est recherché (début ou fin de caractère). Nous ne testons cependant plus vraiment si l'image est toute blanche en hauteur, mais plutôt si elle est toute blanche suivant un angle d'inclinaison donné, qui correspond à l'angle d'inclinaison d'un texte en italique classique. Voici l'algorithme :

```
        if (!fullWhite && firstCut)
        {
            // Saving beginning index
            // (exclusive position: where we will need to cut)
            allPositions[currentIndex] = j - 1;
            currentIndex ++;

            // Drawing lines for test
            int put_j = j != 0 ? j - 1 : 0;
            beginingCharPixel = j;
            int jj = j;
            for (int i = 0; i < img -> h && jj > -1; i++)
            {
                pixel = SDL_MapRGB(img_copy -> format, 0, 0, 255);
                putpixel(img_copy, jj, i, pixel);

                // Updating indexes for italic segmentation (mod = 5)
                if (i % mod == mod-1){
                    jj--;
                }
            }
            firstCut = 0;
        }

        if(fullWhite && !firstCut)
        {
            // Saving ending index
            // (exclusive position: where we will need to cut)
            allPositions[currentIndex] = j + 1;

            // Drawing lines for test
            endingCharPixel = j - 1;
            int actualCharLength = abs(endingCharPixel - beginingCharPixel + 1);
            int jj = j;
            for (int i = 0; i < img -> h && jj > -1; i++)
            {
                pixel = SDL_MapRGB(img_copy -> format, 0, 0, 255);
                putpixel(img_copy, jj, i, pixel);

                // Updating indexes for italic segmentation (mod = 5)
                if (i % mod == mod-1){
                    jj--;
                }
            }
            firstCut = 1;
        }
```

On remarque que cet extrait de code gère le cas où ledébut d'un caractère est trouvé (première boucle), ainsi que le cas où la fin d'un caractère est trouvée (deuxième boucle). Leur fonctionnement est plutôt similaire et nous allons décortiquer la première boucle.

Tout d'abord, on sauvegarde la position du caractère dans notre tableau 'allPositions' qui les stocke. Ensuite, on dessine sur l'image pour voir comment la segmentation s'est déroulée. Dessiner sur l'image est juste un indicateur pour voir comment le programme fonctionne et permet de comprendre comment on détecte que l'image est toute blanche sur la partie souhaitée. La façon dont se passe la detection inclinée est la suivante : on fait une boucle sur toute la hauteur de l'image, puis tous les 5 itérations, on décale le pixel de recherche de 1 vers la gauche, ce qui permet au final d'avoir une détection en biais. Voici le résulat d'une segmentation sur un texte en italique :

$IMAGE_{C}ARACTERES_{I}TALIQUE$

Nous avons également appliqué se processus de détection inclinée pour la segmentation des mots. L'exact même algorithme est appliqué, et il permet d'être plus précis sur la détection des mots. Voici le résultat obtenu :

$IMAGE_{M}OTS_{I}TALIQUE$

# 5 Réseau de neurones

## 5.1 Avancées

### 5.1.1 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'entraînements.

| | 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 "nouveau" 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).

### 5.1.2 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.

## 5.2 Fonctionnement

### 5.2.1 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.

### 5.2.2 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
```

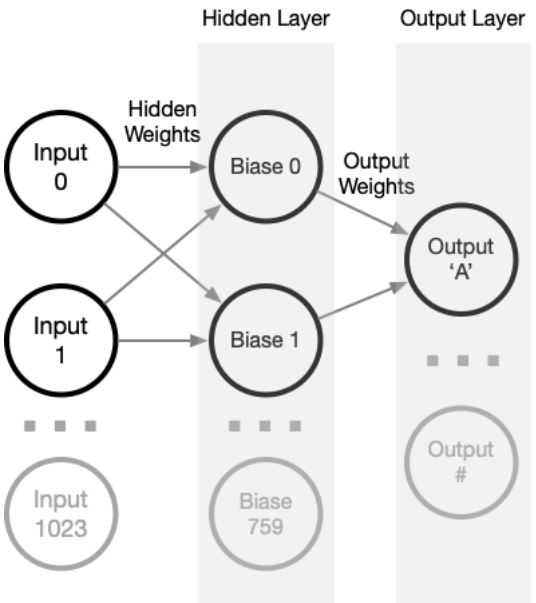
Effectuer un make pour compiler un des programme va créer l'exécutable correspondant dans un dossier bin. Compiler la lib va créer un dossier lib contenant des fichiers .o. Le fichier principal LambdaNeuralNetwork.o est le lien vers tous les autres .o.

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

### 5.2.3 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.)

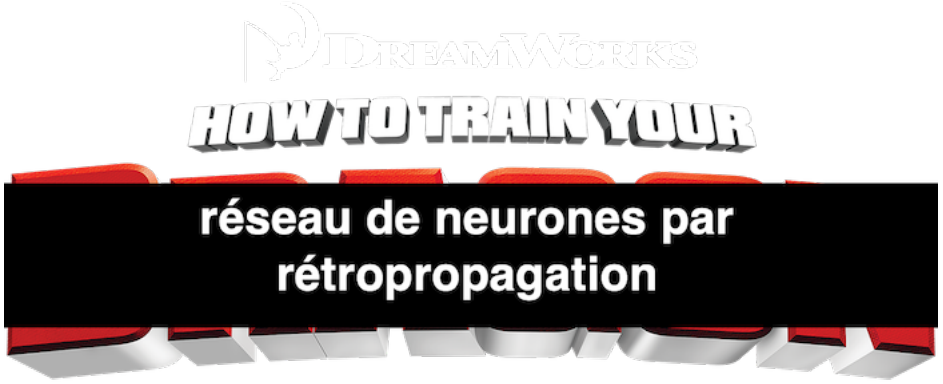


Chaque entrée influera sur chaque neurone caché par lintermédiaire dun poids caché (hidden weight) et du biais du neurone caché. De même, chaque neurone caché influera sur chaque neurone de sortie par lintermédiaire dun poids de sortie (output weight) et du biais du neurone de sortie. À chaque itération 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)



# 6 Correcteur orthographique

## 6.1 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.

## 6.2 La ponctuation

Pour vérifier la ponctuation dun texte nous avons 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 s'il 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 d'être utilisé avec une main qui leur enverrait lindex des points dans un texte.

## 6.3 La cohérence

Cette fonction a vu le jour pour palier un problème du réseau de neurones. En effet, il pouvait arriver que l'IA 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.

## 6.4 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 l'heure 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 l'utilisation d'une IA et dapporter des solutions de correction viable sans lutilisation des habitudes de lutilisateur.

# 7 Command parser

## 7.1 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.

## 7.2 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.

## 7.3 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    }
```

## 7.4 Segmentation

### 7.4.1 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  }
```

### 7.4.2 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    }
```

# 8 Notre avancement et répartition des tâches

## 8.1 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 | |

## 8.2 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 ajouterai 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.

## 8.3 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 column "image_path"
// Applique la segmentation partielle en colonnes de l'image et créer l'architecture resultColumn
    /*.bmp
```

```
./Lambda paragraph "image_path"
// Applique la segmentation partielle en paragraphes de l'image et créer l'architecture
    resultParagraph/*.bmp
```

```
./Lambda line "image_path"
// Applique la segmentation partielle en lignes de l'image et créer l'architecture resultLine/*.
    bmp
```

```
./Lambda word "image_path"
// Applique la segmentation partielle en mots de l'image et créer l'architecture resultWord/*.bmp
```

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

# 9 Conclusion

Ce projet est très motivant pour nous tous. Il nous permet d'expérimenter et de réfléchir à des problématiques intéressantes que nous prenons comme des défis.

Nous avons fortement aimé le sujet ainsi que le cahier des charges de ce projet, et nous nous sommes ainsi entièrement investis dans sa réalisation. Nous sommes toujours autant motivés par ce projet, malgré les hauts et les bas qui, pour les derniers, peuvent être vraiment complexes à solutionner.

Nous vous avons ici présenté ce que nous avons pu réaliser en un mois dans le respect des consignes proposées par le cahier des charges.