
Rapport de soutenance

Charles Quentin Maxime Nathan



BITARRAYS

Par Bitarrays | Projet "Lambda"

Table des matières

1	Présentations	5
1.1	Les membres	5
1.1.1	Quentin "Scout" FISCH	5
1.1.2	Maxime "Maxmad" MADRAU	5
1.1.3	Charles "Draze" SIMON-MEUNIER	5
1.1.4	Nathan "Goruza" AVÉ	5
1.2	Le groupe	6
2	Traitement de l'image (pré-traitement)	7
2.1	Ce que nous devons faire	7
2.2	Le niveau de gris	7
2.3	Binarisation	7
2.4	Augmentation des contrastes	8
2.5	Réduction des bruits	8
3	Rotation de l'image	9
3.1	Rotation manuelle	9
3.2	Rotation automatique	10
4	Segmentation de l'image	12
4.1	Introduction	12
4.2	Segmentation de caractères	12
4.2.1	Découpage des colonnes	12
4.2.2	Découpage des paragraphes	14
4.2.3	Découpage des lignes	16
4.2.4	Découpage des mots et des caractères	16
4.2.5	Gestion du texte en italique	18
5	Réseau de neurones	21
5.1	Avancées	21
5.1.1	Fonction XOR	21
5.1.2	Reconnaissance de caractères	21
5.2	Fonctionnement	22
5.2.1	Scripting	22
5.2.2	Compilation	24
5.2.3	Fonctionnement du réseau	24
6	Correcteur orthographique	26
6.1	L'objectif	26
6.2	La ponctuation	26
6.3	La cohérence	26
6.4	La distance de Levensthein	27
7	Command parser et interface	29
7.1	L'objectif	29
7.2	Grayscale	29
7.3	Filtres	29
7.4	Segmentation	30
7.4.1	Segmentation complète	30
7.4.2	Segmentation d'une ligne	30
7.5	L'interface	31
8	Cheminement du programme	33

8.1	Les filtres et le pré-traitement	33
8.2	De la segmentation jusqu'à la reconnaissance	33
8.3	Output du programme	33
9	Notre avancement et répartition des tâches	34
9.1	Répartition des tâches à la première soutenance	34
9.2	Répartition des tâches à la deuxième soutenance	34
9.3	Résumé de ce que nous avons fait	34
9.4	Utilisation du programme	35
10	Conclusion	36

Prologue

Voici le premier rapport de soutenance final 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 sommes directement entrés 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 projet pour proposer une segmentation des caractères convenable, une IA capable de reconnaître du texte, appliquer des filtres sur nos images et implémenter un correcteur orthographique.

Nous allons présenter ici nos réalisations ainsi que nos problèmes rencontrés sur toute la durée du projet.

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 décidé 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 le premier en langage C 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 à développer des programmes simples très tôt 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.

1.1.3 Charles "Draze" SIMON-MEUNIER

Passionné d'informatique depuis mes années du collège où 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-même certains langages comme le HTML/CSS, Java et aujourd'hui je me perfectionne et corrige mes défauts grâce 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 "Goruzza" AVÉ

Après une première année de sup où j'ai pu vivre ma première expérience de travail de groupe à EPITA, je poursuis l'aventure avec plus d'expérience dans les projets de ce type et aussi un groupe différent. Pour moi, IOCR est une seconde expérience de travail où je vais devoir apprendre à maîtriser des notions par moi-même et non pas lors des cours. Cela s'inscrit dans la continuité de mon stage réalisé cet été par l'apprentissage d'un 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 avons 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 devons 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

```
1 ./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 composent l'image. Nous faisons la moyenne des valeurs rouge, vert et bleu et vérifions si 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.xel 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

```

1  // Foreach pixels
2  for (int i = 0; i < img->w; i++)
3  {
4      for (int j = 0; j < img->h; j++)
5      {
6          // Sum of each neighbors pixels
7          Uint32 sum = 0;
8          short count = 0;
9          for (int k = -1; k < 2; k++)
10         {
11             for (int l = -1; l < 2; l++)
12             {
13                 if (i + k > -1 &&
14                     i + k < img->w &&
15                     j + l > -1 &&
16                     j + l < img->h)
17                 {
18                     pixel = getpixel(img, i + k, j + l);
19                     SDL_GetRGB(pixel, img->format, &r, &g, &b);
20                     sum += r;
21                     count += 1;
22                 }
23             }
24         }
25         // Average of pixels values to get the new current pixel value
26         sum /= count;
27         pixel = SDL_MapRGB(img->format, sum, sum, sum);
28         putpixel(imgCopy, i, j, pixel);
29     }
30 }
```


3 Rotation de l'image

3.1 Rotation manuelle

Nous avons implémenté une rotation manuelle de l'image, c'est-à-dire qu'une rotation de l'image est effectuée pour un angle donné en paramètre. Ainsi, nous pourrions créer un input sur l'interface afin que l'utilisateur puisse dire lui-même de combien de degrés l'image doit être tournée pour être droite lors de la détection du texte. Cependant, nous avons choisi de ne pas implémenter ce fonctionnement directement, mais de créer une fonction qui détecte automatiquement l'angle de l'image. Nous reviendrons sur ce choix et cette fonction dans la partie suivante.

Pour ce qui est de la fonction qui retourne une image de façon manuelle, voici comment celle-ci fonctionne :

```
1
2  SDL_Surface *rotation(SDL_Surface *img, double teta, int x0, int y0) {
3      SDL_Surface *imgCopy = SDL_CreateRGBSurface(0, img->w, img->h, 32, 0, 0, 0, 0);
4
5      // Full white image
6      pixel = SDL_MapRGB(img->format, 255, 255, 255);
7      for (int i = 0; i < img->w; i++) {
8          for (int j = 0; j < img->h; j++) {
9              putpixel(imgCopy, i, j, pixel);
10         }
11     }
12
13     teta = toradians(teta);
14
15     for (int x = 0; x < img->w; x++) {
16         for (int y = 0; y < img->h; y++) {
17             int x2 = round(cos(teta) * (double)(x - x0) + sin(teta) * (double)(y - y0));
18             int y2 = round(cos(teta) * (double)(y - y0) - sin(teta) * (double)(x - x0));
19
20             x2 += x0;
21             y2 += y0;
22             if (x2 > -1 && x2 < img->w && y2 > -1 && y2 < img->h) {
23                 putpixel(imgCopy, x2, y2, getpixel(img, x, y));
24             }
25         }
26     }
27     return imgCopy;
28 }
```

Cette fonction est plutôt simple dans sa manière de fonctionner. Tout d'abord, on crée une nouvelle image toute blanche de la même taille que l'image passée en paramètre. C'est le but de la première boucle de la fonction. Ensuite, on redéfinit l'angle θ donné en paramètre en radian, ce qui permet ensuite d'utiliser les fonctions $\sin()$ et $\cos()$ de la librairie *Math.h*.

Pour ce qui est de la rotation en elle-même de l'image, on parcourt tous les pixels de l'image grâce aux deux boucles *for* imbriquées. On calcule les nouvelles coordonnées du pixel grâce à la formule suivante :

$$\begin{cases} x_2 = \cos(\theta) * (x - x_0) + \sin(\theta) * (y - y_0) \\ y_2 = \cos(\theta) * (y - y_0) + \sin(\theta) * (x - x_0) \end{cases}$$

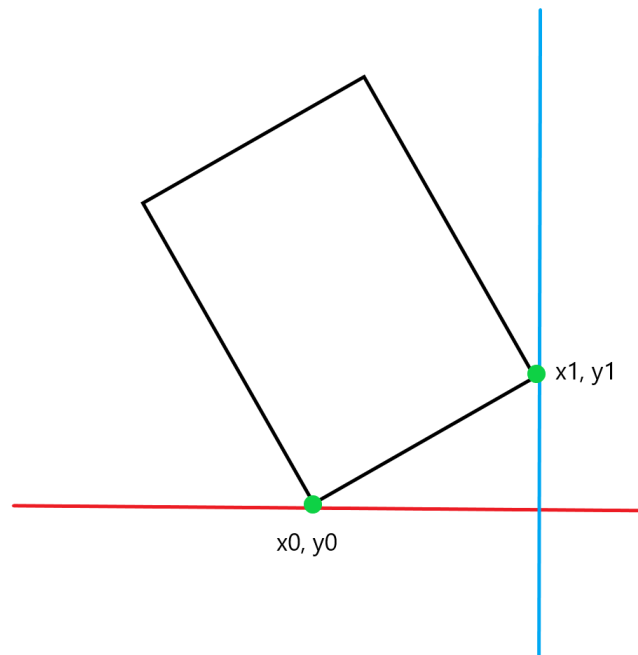
Après avoir calculé les nouvelles coordonnées, on ajoute x_0 à x_2 et y_0 à y_2 pour centrer la rotation. Finalement, il suffit d'ajouter le pixel sur lequel on est placé aux nouvelles coordonnées sur la nouvelle image. Le résultat est le bon, mais on obtient des pixels manquants sur l'image :



Nous avons essayé plusieurs algorithmes pour corriger ce problème, mais nous n'avons pas eu de résultats concluant qui supprimaient totalement ce problème. C'est le défaut d'effectuer une rotation de l'image avant de traiter le texte.

3.2 Rotation automatique

Nous avons souhaité implémenter une rotation automatique de l'image, c'est-à-dire que l'image donnée en entrée du programme est automatiquement testée pour savoir si elle doit être redressée ou non. Si c'est le cas, on utilise un algorithme qui se base sur l'utilisation du théorème de Pythagore. Voici un schéma de la situation :



Tout d'abord, on recherche le couple de coordonnées (x_0, y_0) , qui correspond simplement au pixel le plus bas de l'image qui n'est pas blanc :

```
1
2  coords leftpivot;
3  for (int i = img->h - 1; i > -1; i--)
4  {
5      unsigned char fullWhite = 1;
6      for (int j = 0; j < img->w; j++)
7      {
8          Uint32 pixel = getpixel(img, j, i);
9          SDL_GetRGB(pixel, img->format, &r, &g, &b);
10         if (r < 250 || g < 250 || b < 250)
11         {
12             fullWhite = 0;
13             leftpivot.x = j;
14             leftpivot.y = i;
15             break;
16         }
17     }
18 }
```

Ensuite, on effectue une recherche similaire pour trouver le point le plus à droite de l'image, le couple de coordonnées (x_1, y_1) :

```
1
2  coords rightpivot;
3  for (int i = img->w - 1; i > -1; i--)
4  {
5      unsigned char fullWhite = 1;
6      for (int j = 0; j < img->h; j++)
7      {
8          Uint32 pixel = getpixel(img, i, j);
9          SDL_GetRGB(pixel, img->format, &r, &g, &b);
10         if (r < 250 || g < 250 || b < 250)
11         {
12             fullWhite = 0;
13             rightpivot.x = i;
14             rightpivot.y = j;
15             break;
16         }
17     }
18 }
```

Lorsque ces deux couples sont définis, on peut simplement appliquer le théorème de Pythagore et trouver l'angle de rotation. Voici la formule :

$$\theta = \arctan\left(\frac{y_1 - y_0}{x_1 - x_0}\right)$$

Après avoir déterminé l'angle de rotation, on appelle tout simplement la fonction de rotation manuelle, qui va se charger du reste du travail.

4 Segmentation de l'image

4.1 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.

4.2 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.

4.2.1 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.

L'EPITA consacre deux grandes semaines de conférences technologiques à ses étudiants. Se déroulant traditionnellement au printemps et à l'automne, elles leur permettent de rencontrer de nombreux professionnels et de se familiariser avec les dernières tendances touchant les grandes entreprises de l'innovation.

Le traditionnel forum annuel se tient au cœur du campus parisien. Rendez-vous incontournable pour nos étudiants, l'EPITA y reçoit en moyenne 70 entreprises qui proposent plusieurs milliers d'offres de stages et de premières embauches.

Chaque année, l'EPITA lance un appel à projets ouvert aux grandes entreprises, startups, administrations publiques et associations souhaitant innover ou développer de nouveaux services ou concepts. Les PFEE EPITA sont des projets réalisés par les étudiants dans le cadre d'un mécénat au service d'entreprises ou de l'État.

La conférence TIC et Géopolitique, organisée depuis 2009, réunit experts et praticiens autour des grandes problématiques économiques, sociales et politiques liées aux nouvelles technologies, à leur évolution et à leur impact. Pour l'édition 2018, l'EPITA a l'honneur de recevoir Cédric Villani, médaillé Fields.

député de l'Essonne et vice-Président de l'Office parlementaire d'évaluation des choix scientifiques et technologiques, ainsi que Laurence Devillers, professeure à l'Université Paris-Sorbonne, chercheuse au Laboratoire d'informatique pour la mécanique et les sciences de l'ingénieur (Limsi) du CNRS et auteure, autour de l'Intelligence Artificielle.

Prologin est un concours d'informatique ouvert à tous les étudiants âgés de moins de 21 ans, organisé par l'association éponyme. Son but est de faire découvrir le monde de la programmation et de l'algorithmique aux jeunes et de les confronter à des problèmes classiques et des challenges excitants.

Deux jours et une nuit de festivités dédiés à la japanimation et la culture japonaise : une fois par an, le campus de Paris accueille la convention Epitanime. Concours de cosplay, fanzines, karaoké, jeux de rôle, conférences... un programme à ne manquer sous aucun prétexte !

C'est un moment attendu par tous nos étudiants ! La remise des titres d'ingénieur, organisée dans un lieu prestigieux, réunit les élèves fraîchement diplômés, leurs parents et amis, l'équipe pédagogique de l'école, ainsi que le corps professoral et l'équipe dirigeante de l'ONIS Education Group. Tenues de soirée, Champagne et petits fours, une cérémonie inoubliable pour conclure plusieurs années d'études intenses !

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, nous avons 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             firstCut = 0;
25         }
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 constante 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 vert les lignes tracées par la programme pour délimiter les colonnes, qui peuvent maintenant être coupées :

<p>L'EPITA consacre deux grandes semaines de conférences technologiques à ses étudiants. Se déroulant traditionnellement au printemps et à l'automne, elles leur permettent de rencontrer de nombreux professionnels et de se familiariser avec les dernières tendances touchant les grandes entreprises de l'innovation.</p>	<p>député de l'Essonne et vice-Président de l'Office parlementaire d'évaluation des choix scientifiques et technologiques, ainsi que Laurence Devillers, professeure à l'Université Paris-Sorbonne, chercheuse au Laboratoire d'informatique pour la mécanique et les sciences de l'ingénieur (Limsi) du CNRS et auteure, autour de l'Intelligence Artificielle.</p>
<p>Le traditionnel forum annuel se tient au cœur du campus parisien. Rendez-vous incontournable pour nos étudiants, l'EPITA y reçoit en moyenne 70 entreprises qui proposent plusieurs milliers d'offres de stages et de premières embauches.</p>	<p>Prologin est un concours d'informatique ouvert à tous les étudiants âgés de moins de 21 ans, organisé par l'association éponyme. Son but est de faire découvrir le monde de la programmation et de l'algorithmique aux jeunes et de les confronter à des problèmes classiques et des challenges excitants.</p>
<p>Chaque année, l'EPITA lance un appel à projets ouvert aux grandes entreprises, startups, administrations publiques et associations souhaitant innover ou développer de nouveaux services ou concepts. Les PFEE EPITA sont des projets réalisés par les étudiants dans le cadre d'un mécénat au service d'entreprises ou de l'État.</p>	<p>Deux jours et une nuit de festivités dédiés à la japanimation et la culture japonaise : une fois par an, le campus de Paris accueille la convention Epitanime. Concours de cosplay, fanzines, karaoké, jeux de rôle, conférences... un programme à ne manquer sous aucun prétexte !</p>
<p>La conférence TIC et Géopolitique, organisée depuis 2009, réunit experts et praticiens autour des grandes problématiques économiques, sociales et politiques liées aux nouvelles technologies, à leur évolution et à leur impact. Pour l'édition 2018, l'EPITA a l'honneur de recevoir Cédric Villani, médaillé Fields.</p>	<p>C'est un moment attendu par tous nos étudiants ! La remise des titres d'ingénieur, organisée dans un lieu prestigieux, réunit les élèves fraîchement diplômés, leurs parents et amis, l'équipe pédagogique de l'école, ainsi que le corps professoral et l'équipe dirigeante de IONIS Education Group. Tenues de soirée, Champagne et petits fours, une cérémonie inoubliable pour conclure plusieurs années d'études intenses !</p>

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.

4.2.2 Découpage des paragraphes

Cette partie est la suite logique du découpage des colonnes. En effet, en regardant l'image 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.

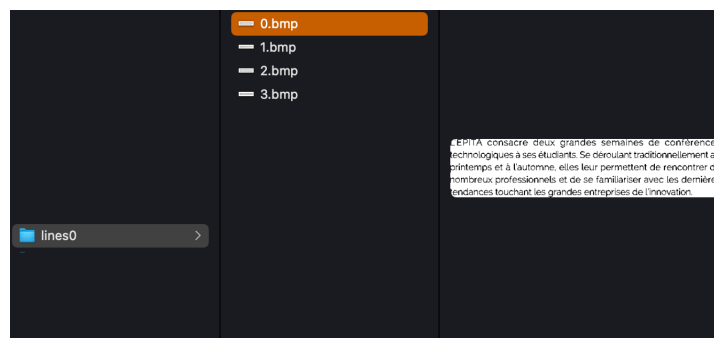
Ici, le changement est principalement basé sur le fait de regarder si la largeur de l'image est complètement blanche, contrairement à la hauteur pour les colonnes. De plus, on détermine ici si un paragraphe se termine en fonction de la "hauteur" de la dernière ligne/du dernier paragraphe trouvé. Voici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

```

1  unsigned char fullWhite = 1;
2  unsigned char firstCut = 1;
3  int endText = -1; //Gets the first pixel (height wise) with full white width
4  int beginingText = -1; //Gets the first pixel without full white width after several full white
5  int lastLineHeight = -1; //Stores the number of pixels from the last text line
6
7  for (int i = 0; i < img -> w; i++)
8  {
9      // Returns whether img's height is full of white pixels or not
10     fullWhite = fullWhite(img, i);
11
12     if (!fullWhite && firstCut)
13     {
14         // Begins column cut
15         beginingText = i;
16         if (endText == -1 || (lastLineHeight <= abs(endText - beginingText))){
17             for (int k = 0; k < img -> w; k++){
18                 {
19                     pixel = SDL_MapRGB(img_copy -> format, 0, 255, 0);
20                     putpixel(img_copy, k, beginingText, pixel);
21                 }
22             }
23             firstCut = 0;
24         }
25
26         if(fullWhite && !firstCut) {
27
28             // Ends column cut
29             endText = i-1;
30             lastLineHeight = endText - beginingText;
31             int ii = i;
32             do
33             {
34                 fullWhite = fullWhite(img, ii);
35                 ii++;
36             } while (fullWhite);
37
38             if (ii - endText >= lastLineWidth || ii >= img -> h) {
39                 // Space is too big, draw line to separate
40                 for (int k = 0; k < img -> w; k++){
41                     {
42                         pixel = SDL_MapRGB(img_copy -> format, 0, 255, 0);
43                         putpixel(img_copy, k, i, pixel);
44                     }
45                 }
46                 firstCut = 1;
47             }
48         }

```

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 :



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

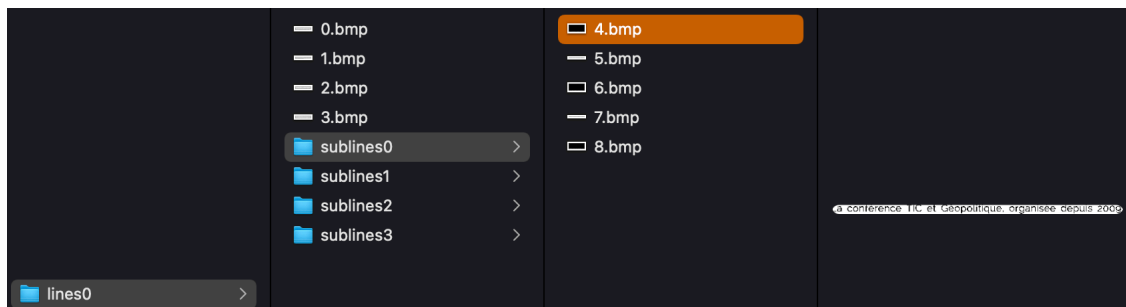
4.2.3 Découpage des lignes

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 de considération qui change par rapport au découpage des paragraphes. Ici, on considère un changement si l'écart détecté 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 :



4.2.4 Découpage des mots et des caractères

Première Version :

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 $0.5 * (image \rightarrow h)$, 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 pour le découpage des caractères pour la première version :

```

1  for (int j = 0; j < img -> w; j++)
2  {
3      fullWhite = fullWhiteHeight(img, j);
4      if (!fullWhite && firstCut)
5      {
6          beginingCharPixel = j;
7
8          /* Draw line for cut (first version usage) */
9
10         firstCut = 0;
11     }
12
13     if(fullWhite && !firstCut)
14     {
15         endingCharPixel = j-1;
16         int actualCharLength = abs(endingCharPixel - beginingCharPixel);
17         firstCut = 1;
18
19         /* Draw line for cut */
20
21         // Bloc is too long
22         if (actualCharLength > 2*averageCharLength)
23         {
24             int times = round(actualCharLength/averageCharLength);
25             int middle = actualCharLength/2 + beginingCharPixel;
26             for (char k = 1; k <= times; k++){
27
28                 /* Draw line at beginingCharPixel + k*actualCharLength/times for cut */
29
30             }
31             nbChars += times;
32             averageCharLength = (averageCharLength*(nbChars-times) + actualCharLength)/nbChars;
33         }
34         else {
35             nbChars += 1;
36             averageCharLength = (averageCharLength*(nbChars-1) + actualCharLength)/nbChars;
37         }
38     }
39 }

```

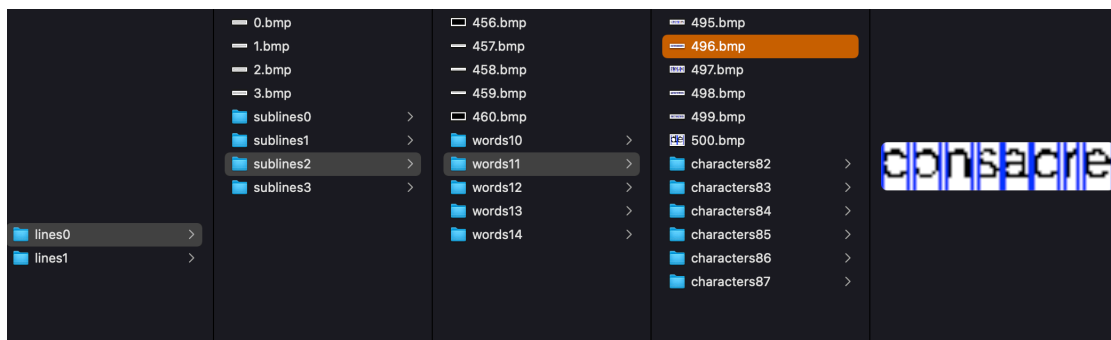
L'utilisation du dessin sur l'image de ligne de couleurs n'était au final pas la solution la plus précise, car il était possible que des lignes empiètent sur des pixels importants pour pouvoir reconnaître des caractères ensuite. Nous avons donc établi une autre version plus précise de la segmentation des caractères.

Version finale :

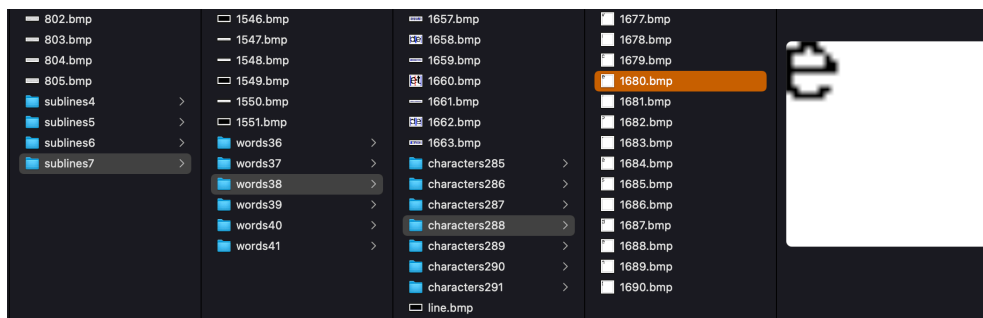
Nous avons réfléchi à une solution permettant de stocker facilement les informations de début et de fin des caractères dans l'image, afin de pouvoir les réutiliser dans d'autres fonctions par la

suite. La solution retenue a 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 vérifier si l'espace trouvé est supérieur à 22% de la hauteur de l'image. Cet algorithme fonctionne très bien et 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 réfléchi 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 avons 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 :

```

1  if (!fullWhite && firstCut)
2  {
3      // Saving beginning index
4      // (exclusive position: where we will need to cut)
5      allPositions[currentIndex] = j - 1;
6      currentIndex ++;
7
8      // Drawing lines for test
9      int put_j = j != 0 ? j - 1 : 0;
10     beginingCharPixel = j;
11     int jj = j;
12     for (int i = 0; i < img -> h && jj > -1; i++)
13     {
14         pixel = SDL_MapRGB(img_copy -> format, 0, 0, 255);
15         putpixel(img_copy, jj, i, pixel);
16
17         // Updating indexes for italic segmentation (mod = 5)
18         if (i % mod == mod-1){
19             jj--;
20         }
21     }
22     firstCut = 0;
23 }
24
25 if(fullWhite && !firstCut)
26 {
27     // Saving ending index
28     // (exclusive position: where we will need to cut)
29     allPositions[currentIndex] = j + 1;
30
31     // Drawing lines for test
32     endingCharPixel = j - 1;
33     int actualCharLength = abs(endingCharPixel - beginingCharPixel + 1);
34     int jj = j;
35     for (int i = 0; i < img -> h && jj > -1; i++)
36     {
37         pixel = SDL_MapRGB(img_copy -> format, 0, 0, 255);
38         putpixel(img_copy, jj, i, pixel);
39
40         // Updating indexes for italic segmentation (mod = 5)
41         if (i % mod == mod-1){
42             jj--;
43         }
44     }
45     firstCut = 1;
46 }

```

On remarque que cet extrait de code gère le cas où le début 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 détection 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 résultat d'une segmentation sur un texte en italique :



La soutenance se passe super bien

Nous avons également appliqué ce 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 :



La soutenance se passe super bien

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 32×32 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 écrit 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 sûrement 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 écrit 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 Sigmoidale 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 prédictions 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 :

```
1 make lib
2 make analyze
3 make learning
4
5 make all
6 make clean
```

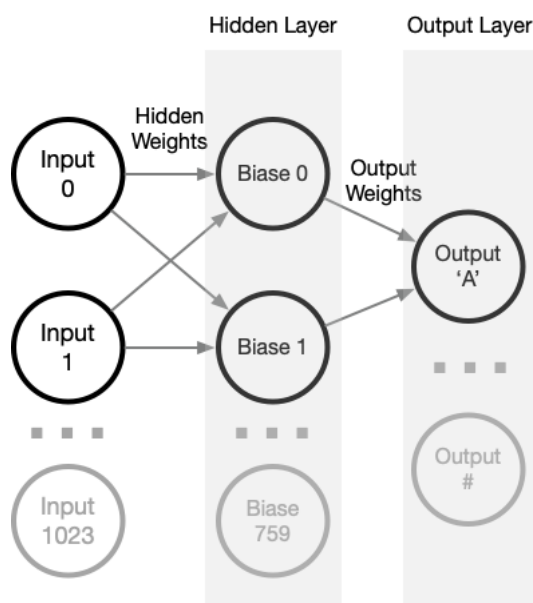
Effectuer un make pour compiler un des programmes va créer l'exécutable correspondant dans un dossier bin. Compiler la lib va créer un dossier lib contenant des fichiers .o. Le fichier principal LambdaNeuralNetwork.o est le lien vers tous les autres .o.

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

5.2.3 Fonctionnement du réseau

L'apprentissage du réseau utilise un algorithme du gradient 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 l'intermédiaire d'un poids caché (hidden weight) et du biais du neurone caché. De même, chaque neurone caché influera sur chaque neurone de sortie par l'intermédiaire d'un poids de sortie (output weight) et du biais du neurone de sortie. À chaque itération de l'apprentissage, une valeur est calculée pour chaque neurone caché :

$$hiddenLayer_j = \sigma(hiddenLayerBias_j + \sum_i input_i \times hiddenWeight_{i \rightarrow j})$$

(où $hiddenLayerBias_j$ est le biais du neurone, $hiddenWeight_{i \rightarrow j}$ le poids de la transition de l'entrée i vers le neurone)

Idem pour les valeurs de la couche de sortie :

$$outputLayer_k = \sigma(outputLayerBias_k + \sum_i hiddenLayer_j \times outputWeight_{j \rightarrow k})$$

(où $outputLayerBias_k$ est le biais du neurone, $outputWeight_{j \rightarrow k}$ le poids de la transition du neurone caché j 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 IOCR et avoir une vérification supplémentaire de ce qui est renvoyé par le réseau de neurones.

6.2 La ponctuation

Pour vérifier la ponctuation d'un texte nous avons opté pour un ensemble de fonctions 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 d'interrogation et celui d'exclamation. Ces fonctions ont pour vocation d'être utilisées avec une main qui leur enverrait l'index des points dans un texte.

```
1 char* dot_check (char* text_to_check, unsigned long start){
2     unsigned int i = start;
3     char *dest = calloc(4, sizeof(char));
4     dest[0] = '.';
5     dest[1] = ' ';
6
7     if (text_to_check[i+1] != ' '){
8         i--;
9     }
10    if (text_to_check[i+2] >= 'A' && text_to_check[i+2] <= 'Z')
11    {
12        dest[2] = text_to_check[i+2];
13    }
14    else if (text_to_check[i+2] >= 'a' && text_to_check[i+2] <= 'z')
15    {
16        dest[2] = text_to_check[i+2] - 32;
17    }
18    else
19    {
20        /* Make a call for the dictionary function */
21    }
22
23    return dest;
```

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 s'il y a plus de lettres majuscules ou minuscules dans le mot et change le tout. Cette fonction nous apporte une sécurité supplémentaire.

```
1 char* capital_check (char* word){
2     size_t count_capital = 0;
3     size_t len = strlen(word);
4     size_t half = len / 2;
5     size_t i = 0;
6     char *dest = calloc(len + 1, sizeof(char));
7
8     while (i < len && count_capital < half){
9         if (word[i] >= 'A' && word[i] <= 'Z'){
10             count_capital++;
11         }
12         i++;
13     }
```

```

14  if (i < len){
15      for (size_t y = 0; y < len; y++)
16      {
17          if (word[y] >= 'a' && word[y] <= 'z'){
18              dest[y] = word[y] - 32;
19          }
20          else
21          {
22              dest[y] = word[y];
23          }
24      }
25  }
26  else
27  {
28      for (size_t y = 0; y < len; y++)
29      {
30          if (word[y] >= 'A' && word[y] <= 'Z'){
31              dest[y] = word[y] + 32;
32          }
33          else
34          {
35              dest[y] = word[y];
36          }
37      }
38  }
39
40  return dest;

```

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 l'utilisation 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 d'apporter des solutions de correction viable sans l'utilisation des habitudes de l'utilisateur.

		H	Y	U	N	D	A	I
	0	1	2	3	4	5	6	7
H	1	0	1	2	3	4	5	6
O	2	1	1	2	3	4	5	6
N	3	2	2	2	2	3	4	5
D	4	3	3	3	3	2	3	4
A	5	4	4	4	4	3	2	3

Pour cette deuxième soutenance nous devons améliorer le temps d'exécution de la distance de Levenshtein. Cependant, nous nous sommes rendu compte que la version présentée à la première soutenance avait encore des bugs importants qui étaient passés inaperçus. De ce fait, le temps annoncé à la dernière soutenance n'avait plus autant d'importance que prévu. Nos plans ont donc changé et sont passés de l'optimisation à la correction puis à l'optimisation.

Par exemple, un mot comme *bonjour* ne répondait pas correctement à notre algorithme et renvoyait une distance erronée avec le mot *bonjour*. En effet, les mots dont il existe un préfixe dans la langue française faussaient le résultat. À l'heure de cette soutenance ce problème est réglé.

De plus, des mots comme *anticonstitutionnellement* (pour prendre un exemple long) avaient tendance à avoir un résultat non pertinent en fonction des différentes erreurs possibles dans le mot. Ce problème est à présent corrigé.

Après ces corrections, le correcteur orthographique effectuait en 0.7 secondes. Nous avions donc perdu du temps d'exécution, ce qui était l'opposé de l'objectif pour cette soutenance.

Nous avons ensuite essayé de l'optimiser par la technique prévue à la première soutenance, cependant, les résultats ne furent pas ceux escomptés et nous ne gagnâmes que 0.2 secondes dans le processus. Nous revenions alors à notre point de départ. Nous avons alors abandonné cette idée pour pouvoir nous concentrer sur l'optimisation du calcul de la distance de Levenshtein.

La meilleure nouvelle version de notre distance était plus rapide pour les mots étant dans les premiers de l'ordre alphabétique mais plus longue pour les derniers (On oscillait entre 0.3 et 0.8). Cette version n'étant pas viable pour un correcteur orthographique car nous avons autant de chance de détecter un mot de la fin de l'alphabet que du début, nous nous sommes décidés à garder l'ancienne version et à tenter de trouver de nouvelles optimisations. (Optimisation de code ou bien réduction du dictionnaire).

À la suite de ces optimisations, nous avons obtenu une distance de Levenshtein avec un temps d'exécution de 0.25 secondes ce qui n'est malheureusement toujours pas assez bas pour pouvoir l'utiliser en condition réelle mais qui nous a permis d'en apprendre plus sur les différentes techniques d'optimisation de code en C.

7 Command parser et interface

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
14         - 2);
15     return 1;
16 }
```

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     }
```

```

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 -
21           2);
22     return 1;
23 }

```

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",
11           argc - 2);
12     return 1;
13 }

```

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 :

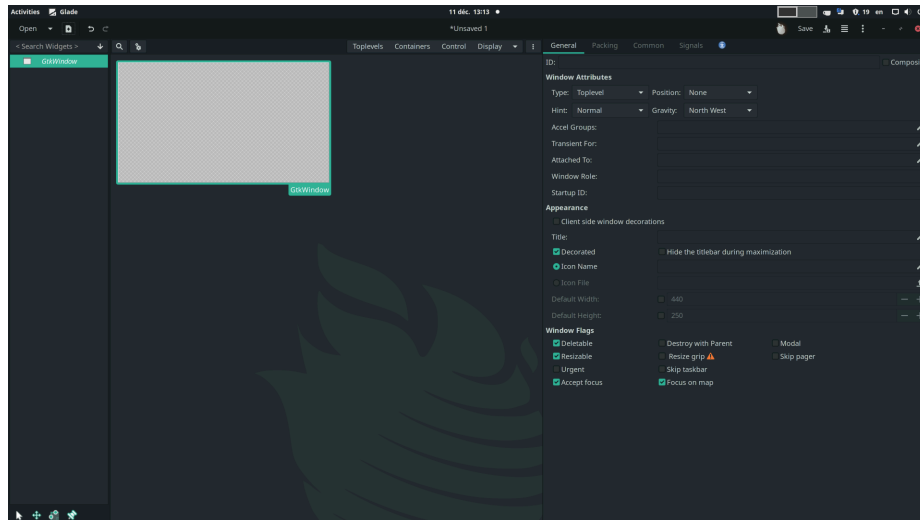
```

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",
15           argc - 2);
16     return 1;
17 }

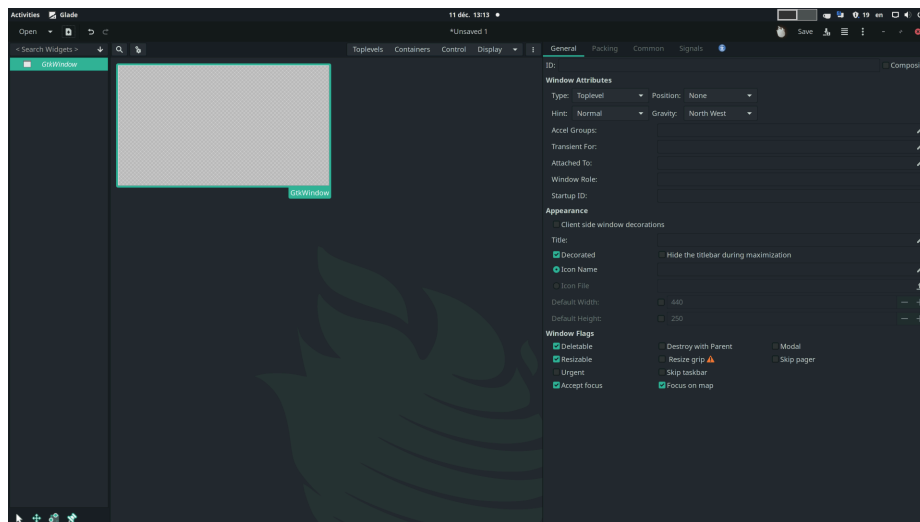
```

7.5 L'interface

Dans ce projet, il est nécessaire de créer une interface en C. Pour cela, nous avons décidé d'utiliser GTK et de construire l'interface via Glade qui permet de manipuler les éléments et les événements de façon plus simple et plus intuitive. L'interface de glade ressemble à cela :

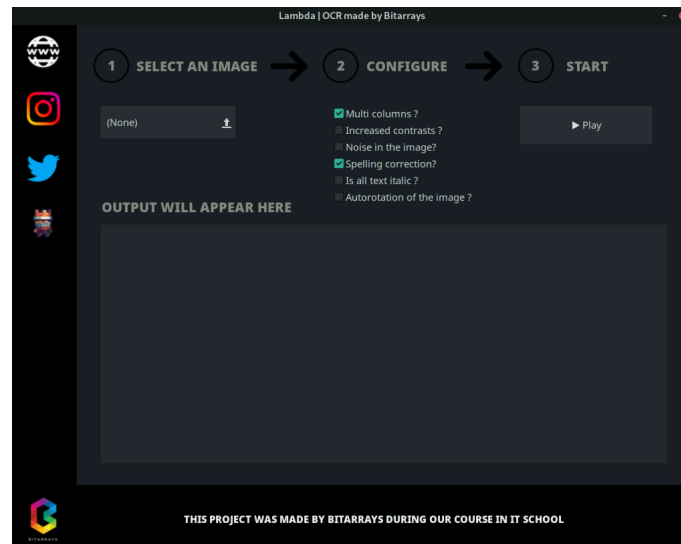


Notre interface est donc générée grâce à ce logiciel, ainsi que les événements directement. Voici notre vision de l'interface dans glade :



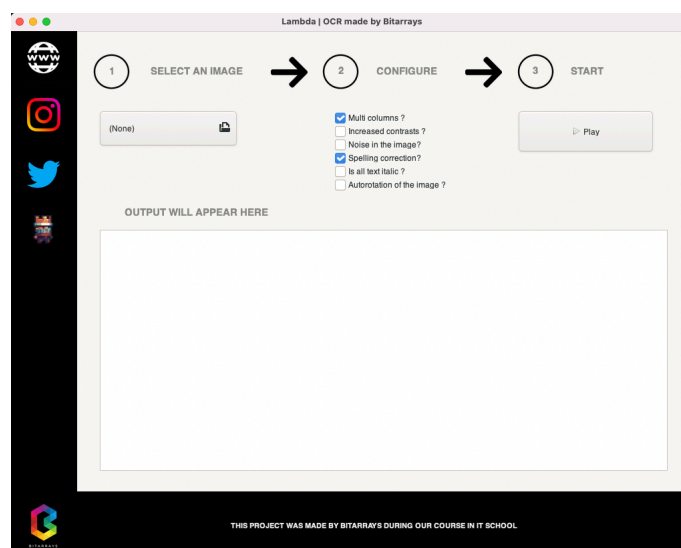
Pour utiliser les éléments de l'interface, nous utilisons la partie Signals associée à chaque éléments qui se trouvent sur l'interface. Par exemple, pour modifier le statut des options de notre programme comme la gestion du multi-colonnes, du texte en italique, etc nous avons un booléen global dans notre code en C une fonction qui permet d'inverser le statut de ce booléen. Notre interface possède un signal 'clicked' sur ces éléments qui appelle la fonction et maintient le booléen à 1 si la checkbox est cochée et 0 sinon.

Pour utiliser notre interface depuis une commande, il faut utiliser `./Lambda` sans argument. C'est alors une fenêtre qui ne peut pas être redimensionnée qui s'ouvre. Dans cette dernière, il est possible de faire charger sa propre image pour la tester à travers notre OCR, en sélectionnant les options adéquates (pas de multi-colonnes si l'image ne possède qu'une colonne, pas de rotation si l'image est droite, etc...). Le texte rendu par notre programme s'écrit dans l'encadré. Il est ensuite possible de scroller dans le texte envoyé, de le copier et de le coller ailleurs.



Cette interface répond donc bien au cahier des charges du projet. Elle permet d'utiliser toutes les fonctionnalités de notre programme et de récupérer le texte détecté sur une image entrée en paramètre.

Le petit point supplémentaire qu'il est intéressant de souligner est le fait que notre interface s'adapte au thème de l'OS (sous Linux et macOS). Sous Linux, la simple implémentation de la fenêtre comme nous l'avons fait suffit à l'adapter au thème de l'OS, sous macOS il a fallu y ajouter une feuille de style CSS pour configurer ce changement de couleur de la fenêtre. Voici un aperçu de notre interface en thème clair :



8 Cheminement du programme

8.1 Les filtres et le pré-traitement

La première chose qui est exécutée au lancement du programme est la suppression de tous les dossiers et fichiers qui peuvent se trouver dans le dossier results à la racine de notre projet. C'est dans ce dernier que les fichiers temporaires nécessaires au fonctionnement de notre programme sont stockés. Après cela, nous nous assurons de pouvoir lancer notre programme sans avoir de problèmes de fichiers/dossiers déjà existants ou non vides.

La prochaine étape est l'application des filtres et des traitements sur l'image. Dans tous les cas nous effectuons un niveau de gris puis une binarisation de l'image entrée en paramètre. Cette dernière est stockée dans results/temp.bmp. C'est avec cette image que notre programme commencera. Avant de faire ces étapes nous effectuons les options sélectionnées à partir de notre interface : c'est à dire la réduction de bruit, l'augmentation des contrastes, la rotation automatique.

8.2 De la segmentation jusqu'à la reconnaissance

yo

8.3 Output du programme

yo

9 Notre avancement et répartition des tâches

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.

9.1 Répartition des tâches à la première soutenance

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

9.2 Répartition des tâches à la deuxième soutenance

	Quentin	Maxime	Charles	Nathan
Rotation de l'image	X		X	
Segmentation italique	X			
Réseau de neurones		X		
Correcteur orthographique				X
Interface		X	X	
Architecture globale	X		X	

9.3 Résumé de ce que nous avons fait

Notre segmentation des caractères est plutôt efficace, il arrive parfois d'avoir 2 caractères qui sont détectés comme un seul mais cela est assez rare pour que la forme globale du texte reconnu soit intéressante. Nous avons ajouté une découpe des caractères en italique en reprenant notre méthode pour les caractères non italiques. Le seul problème est que nous n'avons pas pu faire une segmentation sur du texte qui est écrit en police classique puis en italique sur une autre ligne.

Notre IA possède un nombre important de caractères et symboles reconnus, et notre temps d'entraînement (bien qu'important) n'est sûrement pas encore suffisant pour qu'elle soit parfaite. Le pourcentage d'erreur est faible mais il est encore assez important pour que l'on remarque des erreurs dans les mots.

Le correcteur orthographique était une partie bien compliquée de notre programme, celui-ci est fonctionnel mais manque de méthodes différentes pour choisir le bon mot à remplacer quand celui que nous détectons est erroné. La simple utilisation de la distance de Levenshtein à ses limites mais faire un premier essai d'un correcteur orthographique était intéressant.

9.4 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 :

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

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

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

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

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

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

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

```
1 ./Lambda
2 // Ouvre l'interface, nécessite GTK et une interface graphique
```

10 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.

Ce projet dans un langage qui nous était inconnu au début de l'année nous a apporté des connaissances, mais dans le même temps c'est un projet beaucoup trop court pour assimiler tout ce qui nous est demandé. Ainsi, nous en voyons aujourd'hui les limites avec des problèmes dans notre code qui proviennent de notions que nous n'avons pas eu le temps d'approfondir et nous n'arrivons pas à expliquer la provenance de ses dernières. C'est un point un peu frustrant sur le projet car avec du temps supplémentaire et notre expérience, nous serions prêts à reprendre une grande partie de notre code pour le simplifier et limiter les problèmes que nous avons aujourd'hui.