Mathieu `RustyCrowbar' Corre
Thibaud `zehir' Michaud
Valentin `toogy' Iovene
Pierre `Grimpow' Gorjux

11 décembre 2013

Table des matières

Introduction

Ce document présente le compte-rendu de notre projet, KiBiOCR, dans le cadre du curses de l'EPITA. Le nom du projet, "KiBiOCR" parle de lui-même puisque c'est un logiciel d'OCR (Optical Character Recognition). Le but est donc de reconnaître des caractères sur une image.

Notre projet est séparé en 5 parties bien distinctes, auxquelles nous avons donné des noms pour pouvoir les designer plus facilement.

— Lenna : c'est la partie de notre programme qui s'occupe du pre-processing. C'est-à dire que son but est de modifier l'image donnée par l'utilisateur pour la rendre utilisable par les autres parties du programme ; d'où son nom, "pre"-processing. Lenna intervient au début du process.

— Freddy : il intervient après Lenna pour découper (d'où son nom) notre image en petit-morceaux (paragraphes, lignes, mots, caractères).

— Anna : c'est la partie du programme qui s'occupe d'identifier les caractères. Anna prend une image en input et retourne un caratères correspondant à l'image.

— *(Le petit)* Robert : c'est la partie du programme qui s'occupe de la correction syntaxique de la langue du document.

— Guy : c'est l'interface entre l'homme et la machine. Nous avons décidé de la faire en HTML/CSS pour avoir un rendu plus clair et pour pouvoir faire tourner notre programme sur un serveur pour que n'importe qui puisse tester notre OCR n'importe quand.

Nous allons expliquer comment chaque brique de notre programme a été réalisé, les contraintes techniques que nous avons rencontrées pour cha-

cune, et comment nous y avons remédier/comment nous avons essayé d'y remédier.

1 | Lenna -- *Pré-traitement*

Avant d'essayer de détecter les paragraphes, les lignes ainsi que les caractères dans l'image, et de reconnaître les caractères, il faut "nettoyer" l'image. Dans ce chapitre nous allons décrire différents algorithmes que nous avons implémentés dans ce but.

1.1 Binarisation

C'est le premier algorithme que nous avons écrit parce qu'il est simple, essentiel et c'est un bon point de départ pour se familiariser avec la bibliothèque et ses fonctions de base. Nous avons d'abord implémenté la méthode la plus naïve : calculer la moyenne des composantes RGB de chaque pixel pour avoir sa luminosité, et appliquer un seuil. Tout pixel dont la luminosité est en dessous de 127 est considéré comme blanc et les autres comme noir. Evidemment, ce ne fut pas notre algorithme définitif.

Nous aurions pu l'améliorer en prenant comme seuil la luminosité moyenne des pixels, cependant cela n'aurais pas été beaucoup mieux. Nous avons plutot décidé de rechercher un meilleur algorithme, et avons décidé que la méthode d'Otsu était la plus adaptée à notre cas. De plus cet algorithme est relativement simple à impémenter. La principe de la méthode d'Otsu est de trouver le seuil idéal pour l'algorithme décrit précedemment. Le seui idéal est défini comme étant celui qui minimise la variance entre les deux classes formées par ce seuil. Il faut donc essayer chaque seuil possible et calculer à chaque fois la variance de chaque classe. Heureusement une petite astuce

mathématique nous permet de réduire de façon significative la complexité de cet algorithme en maximisant la variance inter-classe au lieu de minimiser la variance intra-classe.

1.2 Réduction du bruit

Nous avons éssayé deux méthodes pour réduire le bruit dans l'image : le flou gaussien et le filtre médian. Nous avons été insatisfait des deux car les lettres étaient moins lisible après le filtre. Pour corriger ça, nous avons réduit le nombre de pixel voisins pris en compte dans le filtre median : pour chaque pixel, nous prenons le pixel median parmi le pixel lui même et ses quatre voisins immédiats.

1.3 Rotation

1.4 Algorithme de base

Il est très difficile voir impossible (ou du moins contre-productif) de détecter des zones de textes, caractères, mots dans une image si cette image n'est pas droite. C'est pourquoi il est nécessaire, au cours de l'étape de prétraitement *pre-processing*, de roter l'image fournie par l'utilisateur afin qu'elle soit bien adaptée aux étapes de segmentation et d'identification des caractères.

1.4.1 Implémentation naive

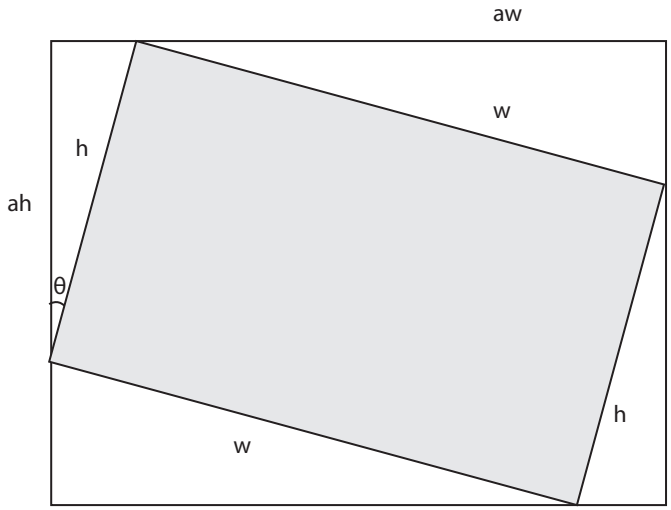
Une implémentation naive de la rotation serait l'algorithme suivant. C'est celui que nous utilisions à la première soutenance mais qui posait quelques soucis.

— Input

  — Image de base
  — Angle de rotation θ

— Output
  — Image rotée

1. Soit (w, h) les dimensions de l'image de base. Calculer les dimensions (nw, nh) de l'image après qu'une rotation d'angle θ soit appliquée sur l'image de base. Visuellement, les dimensions de l'image de base et de l'image rotée sont liées par une simple relation trigonométrique.



2. Créer une nouvelle image de dimensions (nw, nh) remplie de blanc.
3. Pour chaque pixel de la nouvelle image, matérialisé par ses coordonnées (x, y) dans l'image rotée, calculer ses anciennes coordonnées (x', y') dans l'image de base. Pour ce faire, on a la relation :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

4. Arrondir ces nouvelles coordonnées et donner aux pixel (x, y) de la nouvelle image la couleur du pixel (x', y') de l'image de base. Renvoyer l'image.

Cependant, cet algorithme déforme légèrement l'image. Ici, il s'agit de texte et cette petite déformation réduit énormément sa lisibilité ; pour l'homme comme pour la machine. Il s'agit donc d'être plus intelligent.

1.4.2 Interpolation Bilinéaire

Interpolation mathématique

En mathématiques, une interpolation consiste à se servir de données déjà existante pour approximer des données qui nous manquent.

Une interpolation peut se faire de plusieurs façons ; plus ou moins précises. Bien sûr, qui dit plus de précision dit plus de temps de calcul.

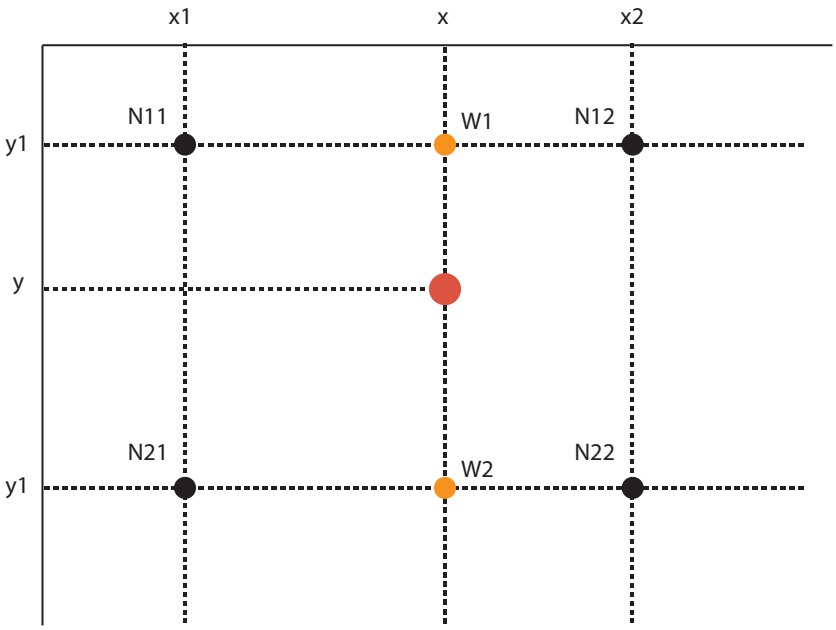
Interpolation bilinéaire appliquée à la rotation

Ici, il s'agit de se servir des pixels de notre image de base (nos données) pour déterminer les pixels de notre nouvelle image (les données que nous allons interpoler).

Plusieurs algorithmes d'interpolation existent :

— Nearest-neighbor Interpolation : le plus simple, il consiste à faire la moyenne des composantes (r, g, b) des pixels voisins de notre pixel (x', y'). C'est le plus rapide.

— Bilinear Interpolation : un peu plus complexe, il s'agit ici de faire deux interpolations linéaires : une entre les deux pixels au dessus de (x', y') et une autre entre les deux pixels en dessous de (x', y') (les valeurs x' et y' n'étant évidemment pas approximées mais bien située entre les entiers de l'arrondi supérieure et de l'arrondi inférieur des coordonnées).



On a alors :

$$W_1 = \frac{x_2 - x}{x_2 - x_1} N_{11} + \frac{x - x_1}{x_2 - x_1} N_{21}$$

$$\text{et } W_2 = \frac{x_2 - x}{x_2 - x_1} N_{12} + \frac{x - x_1}{x_2 - x_1} N_{22}$$

Les nouvelles composantes du pixel peuvent alors être calculée via la formule

$$R = \frac{y_2 - y}{y_2 - y_1} W_1 + \frac{y - y_1}{y_2 - y_1} W_2 \textit{ (cas de la composante rouge du pixel)}$$

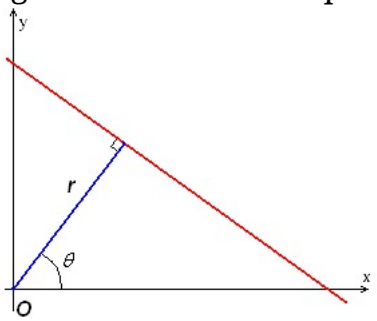
— Bicubic Interpolation : plus longue de part son temps de calcul, c'est le même principle que pour l'interpolatian bicubique sauf que 8 pixels sont étudiés au lieu de 4. Elle donne des résultats plus esthétiques qui ne sont pas vraiment utiles dans notre cas.

1.4.3 Detection de l'angle

Transformée de Hough

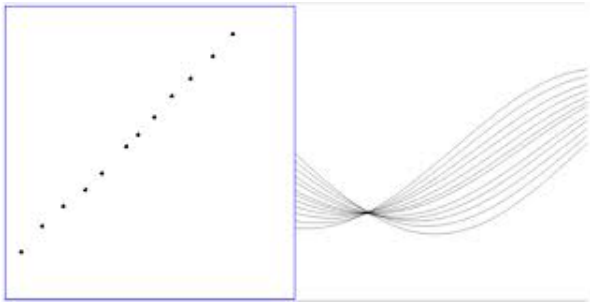
La transformée de Hough est une méthode générale pour trouver des lignes et des ellipses dans une image. Elle est souvent utilisée car une fois le principe mathématique compris, elle est simple à implémenter, donne de bons résultats et à une complexité assez faible par rapport, par exemple, à la transformée de Fourier. Nous n'avons besoin que du cas le plus simple de l'algorithme qui consiste à trouver les lignes dans une image binarisée.

La transformée de Hough se base sur une représentation très particulière des droites du plan : plutot que deux les représenter avec les deux paramètres usuels a et b dans l'équation $y = ax + b$, une droite est décrite comme un couple (r, θ) où r est la distance de la ligne à l'origine et θ est l'angle de cette distance par rapport à l'abscisse.



En quoi cette représentation est-elle utile ? Cela vient du fait que, étant donné un point de coordonnées (x, y) et l'angle θ de la droite passant par ce point, il est facile d'exprimer la distance r entre la ligne et l'origine. L'équation est la suivante :
$r = x \cos\theta + y \sin\theta$
Donc pour un point de coordonnées (x, y) on peut tracer le graphe de r en fonction de θ. Il est facile de deviner d'après l'équation que ce graphe sera une sinusoïde. Cette sinusoïde représente l'ensemble des droites passant par ce point. Si nous stockons ce graphe dans un accumulateur, et ajoutons dans cet accumulateur la sinusoïde correspondant à tous les autres points, on obtient un graphe tableau dans lequel la valeur la plus grande représentera les coordonnées de la droite (dans l'espace de Hough) passant par le

plus de points possible.



Plus de pré-traitement

Il y'a encore un problème à régler avant de pouvoir appliquer la transformée de Hough à notre image scannée : Une ligne de texte n'apparait pas comme une ligne droite. Nous avons essayé d'appliquer directement à une image binarisée, mais la plupart du temps il trouvait ou bien la diagonale de l'image, ou bien une reliure particulièrement visible. Après quelques recherches, nous avons trouvé une solution : on détecte tout d'abord les blocs de pixels noirs connexes, qui ne sont pas toujours mais très souvent des caractères, et on les remplaces par leur point centrale. L'utilité est triple : rendre plus apparentes (du point de vue de la transformée de Hough) les lignes de texte, réduire considérablement la complexité de l'algorithme et reduire l'importance des gros blocs de pixels comme les tâches ou les reliures. Voici un exemple de sortie :

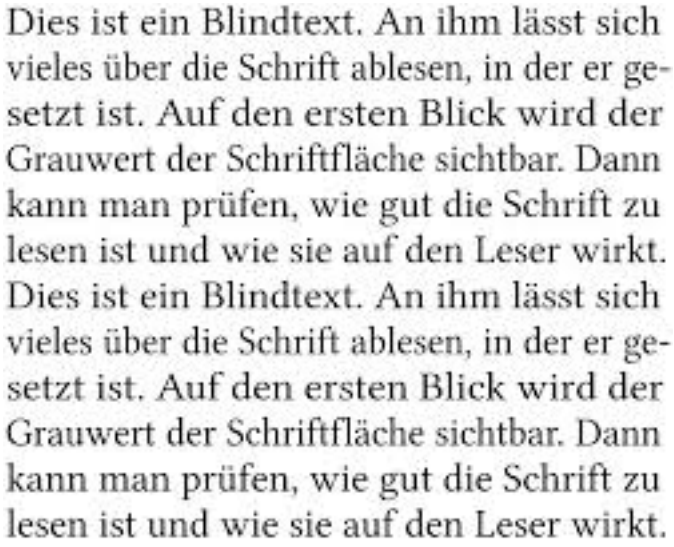


Figure 1.1 – input image

Figure 1.2 – center of each block

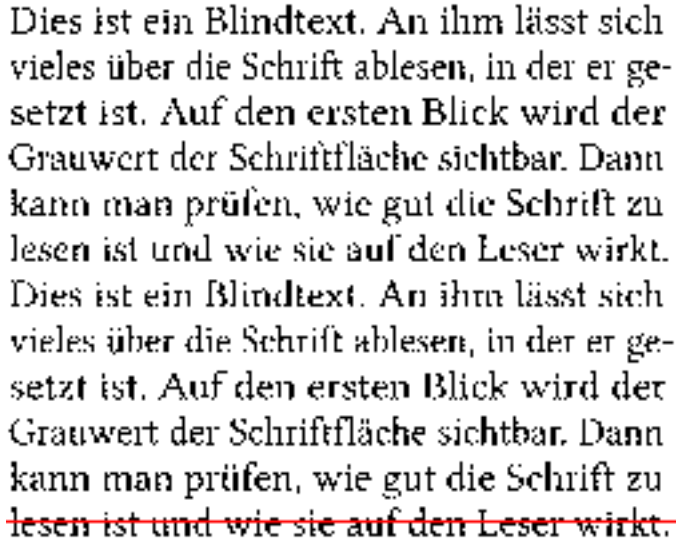


Figure 1.3 – line found by the Hough transform

2 | Freddy - *Segmentation*

Freddy est notre module de segmentation, il utilise OCSFML, et est basé sur l´algorithme RLSA (Run Lenght Smoothing Algorithm).

2.1 L'ancienne méthode, ses points faibles

Au moment de la première soutenance, nous utilisions un algorithme très simple, et moyennement efficace : nous faisions un histogramme horizontal, un tableau de taille la hauteur en pixels de l´image. Nous le remplissions avec le nombre de pixels noirs à la ligne correspondante. Cette opération nous permettait de savoir à quel moment une nouvelle ligne ou un nouveau paragraphe apparaissent.
Une fois ce découpage effectué, il nous suffisait de répéter cette opération horizontalement sur chacune des lignes repérées, afin de détecter les mots et caractères composant chacune des lignes.
Cependant, nous avons été confrontés à plusieurs problèmes, qui nous ont amenés à chercher une autre méthode plus performante, que nous verrons en détail dans la prochaine section. En effet, avec cette méthode, les caractères avaient tendance à être coupés, ou bien collés à d´autres. Il en était de même pour les lignes, qui avaient tendance à fusionner. Cette technique nécessitait également d´avoir une image sortie du prétraitement parfaite, la moindre impureté ou la moindre ligne, image ruinant totalement le résultat de l´algorithme.

2.2 Le RLSA

Pour détecter les paragraphes, lignes, mots ou les caractères, nous utilisons donc maintenant l'algorithme Run Lenght Smoothing Algorithm (RLSA) modifié par nos soins, pour couvrir au mieux nos besoins. Le principe est simple : on regarde horizontalement et verticalement quels sont les pixels noirs adjacents, c´est à dire quels sont les pixels noirs séparés de moins de n pixels noirs. Nous allons voir comment définir cette constante, qui n´est évidemment pas ´hardcoded´ dans la partie suivante. Il en résultera donc deux matrices de bouléens aux dimensions de l´image. Dans ces matrices représentant la couleur noire ou blanche des pixels de l´image, nous aurons changé tous les pixels blancs entre deux pixels noirs adjacents. Les images représentées par ces matrices ressemblent à l´image d´entrée, mais comme si l´encre avait bavée horizontalement ou verticalement.
Nous appliquons ensuite un opérateur binaire logique entre les bits des deux matrices aux mêmes coordonnées, afin de former une nouvelle matrice. Cette nouvelle matrice, toujours aux dimensions de l´image d´origine, représente l´image résultat, qui sera utilisée pour la suite des traitements. Dans cette image, les lettres, mots, lignes ou paragraphes se retrouveront rassemblés en blocs noirs, suivant le nombre de pixels adjacents que l´on aura considéré.
Cette transformation nous permet ensuite, via un autre algorithme que nous détaillerons plus tard, de récupérer les coordonées et dimensions des zones noircies.
Ainsi, les images éventuelles n´auront pas plus de valeur qu´un simple caractère, après l´application de cet algorithme, générant très peu d´erreurs lors du passage d´anna, notre réseau de neuronnes.

2.2.1 La constante d´adjacence

2.2.2 Optimisations

Pour sélectionner les caractères, mots, lignes ou paragraphes, nous devons modifier la constante représentant le nombre de pixels adjacents à

fusionner. Cependant, cette modification seule ne nous permet pas de sélectionner efficacement une partie précise de l´image. En effet, les caractères sont globalement carrés, tandis que les mots et lignes sont assez horizontaux. Une seule constante ne suffira donc plus. Nous utilisons deux constante : une pour la matrice horizontale, et une autre pour la matrice verticale. Nous ne récupérons qu´une seule valeur lors de l´algorithme de la partie précédente. Nous avons cependant pu définir les deux constantes en fonction du retour de la fonction précédente à l´aide de simples multiplications. Il s´est révèlé que ces valeurs fonctionnent très bien pour toutes les images testés jusqu´ici, quelle que soit leur résolution et leur densité de pixels.
Contrairement au RLSA classique, nous n´utilisons pas uniquement l´opérateur logique ´et´, mais également le ´ou´. Il se trouve que pour la detection des lignes et des paragraphes, avec plus d´espaces entre les pixels noirs (une proportion de pixels noirs par rapport aux pixels blancs plus faible) que pour les caractères et les mots, le ´ou´ donne de bien meilleurs résultats que le ´et´. Cette méthode également s´est révèlée être très efficace quelle que soit l´image.

2.3 Bounding boxes

Une fois les ´blocs noirs´ formés, il faut récupérer leur coordonnées.

2.3.1 La mise en forme des données

Toutes les ´bounding boxes´ doivent être mises en ordre, afin d´être utilisables par anna (Nous voulons restituer le texte dans l´ordre, et non les caractères dans un ordre quelconque). En plus de l´ordre, il faut savoir à quel moment un changement de mot, ligne, ou paragraphe a lieu, afin de pouvoir le faire apparaitre sur le document final. La mise en page importe beaucoup. Il est très difficile de lire un texte sans espace entre les mots, ou un minimum de mise en page. C´est encore plus vrai si certaines erreurs apparaissent. Nous avons décidé de représenter les caractères et tous les

goupes les contenant (mots, lignes, paragraphes) sous forme de listes imbriquées : nous avons accès à la liste des paragraphes. Les paragraphes sont des listes de lignes, qui sont des listes de mots, qui sont des listes de caractères.

2.4 Line and character detection

2.4.1 Making horizontal/vertical histogram

To detect lines, we need to make histograms, containing the number of black pixels in every line/column of line of pixel of the provided image.We put everything in lists.

2.4.2 Delimit lines/characters

When the number of black pixel changes a lot, we decide to create or to end a line/to create or end a character. We put it in lists.

2.4.3 Delimit lines/characters

When the number of black pixel changes a lot, we decide to create or to end a line/to create or end a character. We put it in lists

3 | Anna - *Identification*

3.1 Proof of concept

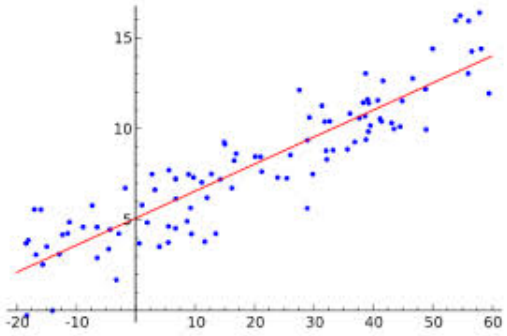
Après avoir pré-traité l'image et trouvé les paragraphes, les lignes et les caractères dans le document, il faut reconnaître ces caractères. Pour ça, nous utilisons un des algorithmes de machine learning les plus répandu : le réseau de neurone feed-forward. Nous allons ici détailler ce que nous avons appris sur cet algorithme, et comment nous l'avons implémenté.

3.1.1 Machine learning

Le machine learning est un champ de recherche dont le but est de trouver des algorithmes capables de s'améliorer sur une tâche particulière avec l'expérience. Souvent, ce que l'on appelle l'expérience consiste en un ensemble d'exemples entrée-sortie correspondant à ce que l'on veut apprendre. Dans notre cas, la tâche du réseau de neurones artificiel sera d'associer à une image de caractère un numéro (par exemple son code ascii), et les exemples sont des paires (image de caractère, numero du caractère).

3.1.2 La régression linéaire à une seule variable

Ramenons nous à un problème plus simple : on a des exemples de la forme (x, y) où x et y sont des nombres, et notre but est de trouver la droite correspondant le mieux à ces exemples.



La ligne sera de la forme $h_w(x) = w_1 x + w_0$. La première chose à faire est de déterminer une fonction d'erreur pour évaluer les différentes droites possibles. Il se trouve que l'erreur quadratique possède des propriétés intéressantes (comme le fait d'être toujours positive et dérivable). Cette fonction d'erreur s'écrite comme suit :
$Err(h_w) = \sum_{j=1}^{N} (y_j - h_w(x_j))^2$
où (x_j, y_j) est le j-ème exemple et $h_w(x_j)$ est l'image du point x par la droite $h_w : x \rightarrow w_1 x + w_0$. Notre but est maintenant de minimiser cette fonction. Imaginons que cette fonction a une forme de bol, et que nous commencions à un point (w_1, w_0) aléatoire de cette fonction (c'est à dire, avec une droite du plan aléatoire). Pour trouver le minimum de cette fonction, il suffit de suivre la pente vers le bas. Mathématiquement parlant, cela signifie qu'il faut prendre le point actuel, et lui soustraire le gradient de la fonction d'erreur. Ce gradient est multiplié par une constante appelée taux d'apprentissage, noté α, pour réguler la taille du pas. Un faible taux d'apprentissage signifie une convergence lente, mais avec un taux trop haut, on risque de passer par dessus le minimum et de ne pas converger. On modifie donc le point courant avec la formule :

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Err(w)$$

3.1.3 La régression linéaire à plusieurs variables

L'exstension de cet algorithme à plusieurs variables se fait assez naturellement. Notons $\mathbf{x} = (x_0, x_1, \ldots, x_n)$ r. La fonction d'estimation devient alors

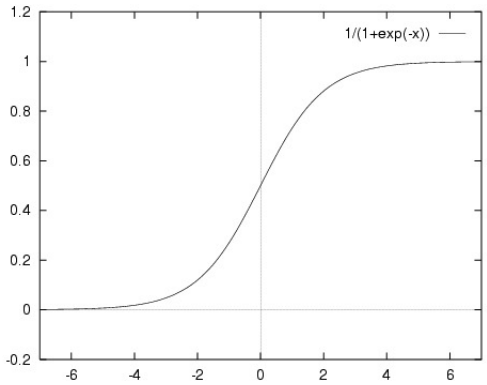
$h_w(\mathbf{x}) = w_n x_n + \cdots + w_1 x_1 + w_0 x_0$.

3.1.4 La regression logistique

Ce dont nous avons besoin pour reconnaître les caractères est un algorithme de classification. Nous n'avons besoin que de quelques modifications pour faire de l'algorithme précédent un classificateur binaire : nous gardons la fonction d'estimation $h_w(x)$, mais nous passons ensuite le résultat de cette fonction dans une autre fonction dont la sortie se trouve dans $[0; 1]$. La fonction de pas en est une : sa sortie vaut 0 si son entrée est négative, et 1 sinon. Mais cette fonction n'est pas dérivable, et nous verrons plus tard que cela pose problème. On lui préfère donc souvent la fonction logistique (aussi appelée sigmoïde)

Son équation est $g(x) = \frac{1}{1+e^{-x}}$, et sa dérivée est $g(x)(1-g(x))$. En applicant le même algorithme que pour la régression linéaire, la formule pour mettre à jour le point actuel devient :

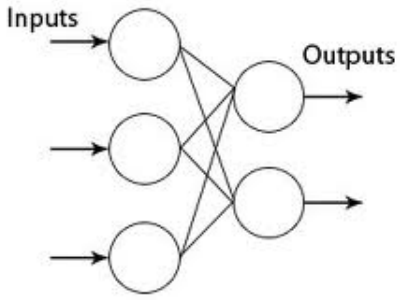
$w_i \leftarrow w_i + \alpha(y - h_w(\mathbf{x}))h_w(\mathbf{x})(1 - h_w(\mathbf{x}))x_i$



3.1.5 Perceptron

La régression logistique est la brique de base du réseau de neurones. C'est ce que nous appelons un neurone, du fait de la ressemblance avec le fonctionnement d'un neurone biologique. Maintenant complexifions un

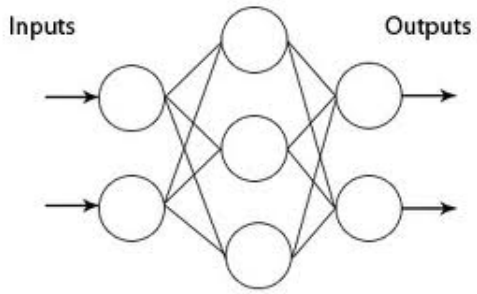
peu le problème : nous devons ranger les entrées parmi plusieurs classes possibles. Supposons par exemple que les entrées puissent appartenir à trois classes possibles.
La solution est de passer l'entrée à trois neurones différents. Cela produira un vecteur de sortie plutot qu'une simple valeur, et idéalement, ce vecteur convergera vers quelque chose comme (1, 0, 0), (0, 1, 0) ou (0, 0, 1), ces trois vecteurs représentant évidemment les trois classes possibles. Cet algorithme est appelé un réseau de neurones feed-formard à une couche, ou perceptron.



3.1.6 Réseau de neurones feed-forward multi-layer

Propagation

Un réseau de neurone feed-forward multi-couches est un perceptron dont le vecteur de sortie sera l'entrée d'un autre percptron, et ansi de suite jusqu'à un perceptron de sortie. Chaque perceptron est appelée une ``couche'' du réseau de neurone. Les couches qui ne sont ni la couche de sortie ni la couche d'entrée sont appelées couches cachées. Le gros avantage d'un réseau de neurones multi-couches par rapport au perceptron est qu'il peu potentiellement calculer n'importe que fonction continue avec seulement une seule couche cachée et avec le bon nombre de neurones dans chaque couche. C'est le résultat énoncé par le théorème universel d'approximation.



Backpropagation

Pour l'algorithme d'apprentissage, la différence majeure est la fonction dérivée de la fonction d'erreur. Il n'est pas nécessaire ici d'expliciter cette dérivée, mais il est à noter que le calcul de la dérivé des poids d'une couche l dépendra de calculs fait à la couche l+1. Ainsi la mise à jour des poids doit se faire de la dernière couche à la première, d'où le nom de ``backpropagation''.

3.1.7 Optimisations

The only optimization we implemented at this point is the momentum. In the gradient descent process, we can consider that the current point has some inertia by adding a certain percentage of the gradient of the last iteration in the update rule. There are two purposes to this : converging faster, and avoiding local minimum.

Deux optimisations ont étés implémentées pour augmenter la vitesse de convergence du réseau de neurones et éviter les minimums locaux :

— Le moment : appelons Δ_n le nombre que l'on ajoute à un poids w_i, j à une étape n de l'apprentissage. A l'étape n+1, on ajoutera en plus du gradient de la fonction d'erreur un certain pourcentage de Δ_n. Cela permet à la fois d'éviter les minimums locaux, car le point possède une sorte d'``inertie'', et à la fois de converger plus vite.

— Taux d'apprentissage non constant : une technique courante pour augmenter la précision de la convergence est de modifier le taux d'apprentissage au cours du temps. Il est grand au départ pour chercher

un minimum global avec peu de précision, puis il diminue pour cherger plus précisément ce minimum.

3.2 Apprentissage

L'algorithme décrit dans la section précedente nous a permis de montrer un réseau de neurone apprenant une porte logique quelconque lors de la première soutenance. Pour la deuxième soutenance, la tâche était nettement plus difficile : nous devions être capable de reconnaître une grande quantité de caractères différents et potentiellement de mauvaise qualité.
Un problème courant en machine learning, et auquel nous avons étés confrontés, est l'overfitting. On parle d'overfitting quand la fonction que l'on essaye d'apprendre devient trop spécifique aux exemples qu'on lui donne, et qu'elle est incapable de généraliser à d'autres entrées. Etant limités par le temps, nous avons optés pour la solution qui semblait la plus simple à implémenter : entrainer plusieurs réseaux de neurones et classifier en fonction de la moyenne de leur sorties. Nous avons donc essayé avec de plus en plus de réseaux de neurones, jusqu'à ce que les différences deviennent imperceptibles. Nous utilisons donc 20 réseaux de neurones au total.
Lors des premiers essais, nous avons aussi eu des problèmes pour trouver la part de responsabilité entre la segmentation et le réseau de neurones. De petites imperfections dans l'un et dans l'autre amène vite à des résultats illisibles.
Au jour du rendu, le résultat final est encore loin d'être parfait, de nombreux caractères sont confondus (9 pour g, c pour e,...) ou n'ont rien à faire là, mais nous avons au moins réussi à produire quelque chose malgré les limitations de temps.

4 | Robert - *Spell checking*

Robert is a module, which can detect the language used in a given text, and eventually indicate to the user which word is wrong, and give a selection of possible corrections.

4.1 Generation of Dictionnaries

A dictionnary is a file which contains a lot of words of a specific language we can find in a text.
We didn't create them ; it's too long and we can easily find some complete dictionnaries on Internet (150 000 words in English, 700 000 words in French, for example).
But actually, searching a word in a file is'nt a really good idea and may take a long time. That's why before doing anything else, we have to generate a data structure which will store the whole words of the dictionnary file. This one is a hashtable, because the time taken to search a common word is constant. So, when loading Robert, OCaml will generate the whole dictionnaries needed by creating hashtables and put them in a list.

But there is still a problem : the time taked to create a hashtable from a dictionnary is around 1 second. If we want to accept many differents languages, the loading of the module would be too long. The solution is the Marshall module : it let us serialize an object into a bytes array, and save it into files. The opposite procedure is also possible. So OCaml has juste to read the byte array and deserialize it to load a hashtable. This technic need more memory but is really faster (around 1000% faster)

4.2 Detecting language in a given Text

The algorithm used is pretty simple : it will analyse the first X words (X is an empirical number, set as 200). For each language, it will count the number of words recognised ; The final language is the one which has the mosts of right matches.

4.3 Detection of wrong words

It's quite the same thing. The algorithm make a list, and starts to read the whole text. Each time a word is wrong, it is added to the list.

4.4 Selection of possible corrections for a wrong word

Generally, when a word is wrong, it's because the user writted it thinking about a right word, with the same pronunciation.
That's why the Soundex algorithm has been used in our project. It transforms a given string into its phonetic equivalent. So, we created a new type of dictionnary : a "phonetic hashtable" : we can find the word by searching the phonetic string. So, if we're searching a phonetic string we can find the whole words which have the same phonetic string. We have a first list with a lot of possible corrections.
But we won't give for examples 42 possibilies of correction to a wrong word : the user would be lost ; that's why we'll give to the user only 5 possibilities. The possibilities will be sorted using another algorithm, called Levenshtein. Levenshtein calculates the number of modifications we have to make in order to change a word A into a word B. So, we'll only have the 5 words which are the closest to the wrong word.

5 | Guy - *Graphical User Interface*

5.1 A quick overview of Guy

Guy is our GUI. It uses LablGTK and OCSFML, and will probably be using html to render the text output. It has several roles that we will describe right now. It is today composed of one window, to choose the file. It will be, for the second soutenance, composed of two or three more windows, to show the flow of the processing and the time taken by every main algorithm that we are using.

5.2 A user-friendly interface

5.2.1 Helping the user to choose the image to process

Guy has a built-in file browser, which is very intuitive, and easy to use, to select the file you want to use. Once the file has been selected, it's path will be displayed in an editable text box. However, the user can directly type in the provided text box the path of his desired file. Of course, it is still possible to use our program via the command-lines.

5.2.2 Displaying the usefull informations

We will display a miniature image of the image (assuming that it really is an image) that the user choosed.

5.2.3 Showing the final result to the user

Guy will be charged to show to the user the final work of our OCR. Indeed, we will present the image the user gave us, and the text we could detect, with the eventual orthograph corrections using Robert. It needs to be well organised, so that in case of a long document, the user could find where everything in the output text is located in his image, and vice-versa.

5.3 Guy is the foreman

One of the role of our GUI is to make every module work one after the other, calling them one after the other, with the right parameters. It will be possible to include one or several progress bars to inform the user of the flow of the processing.

6 Website

6.1 Python

This year, we decided to switch from PHP to Python. We think that Python is a stronger and more relevant language and we wanted to learn its syntax and its specifics. So, here we are.

Plus, if we want to put our OCR online so that people can use it via a web interface, it would me so much easier to do it with Python than with PHP. Python can easily call other programs and execute commands on the server in a very secure way. It is more complicated with PHP.

6.2 Django

There is a very nice Python web framework out there on the web that is called Django. Working with Django is quick : it is made for rapid development and for perfectionists (we are).

6.3 Nginx

We wanted to use something else than a basic Apache server so we decided to go with Nginx, a quick, clean and easy to configure web server.

6.4 Github

Our code is hosted on the now very famous website (and git server) GitHub. For the moment, it is located in a private repository (for obvious reasons) but we plan to go opensource once the last soutenance will be passed !

Conclusion

Our project is moving forward very well. Our team is organized, we are all working with passion and seriousness. We often schedule meetings and establish task lists so that everyone is aware of what has to be done and can pick up something he is interested in.

Some very tough tasks are waiting for us and we hope we will be able to show up at the final with a fresh and working piece of art.