Département IMI
Février 2018 - Mai 2018

Projet de Deep Learning

Reconnaissance de Caractères dans des Manuscrits

Guillaume Desforges et Théo Viel,
Encadrés par Mathieu Aubry, Imagine, LIGM, École des Ponts

Contents

Part I
Introduction

L'objectif de ce projet est de tester différentes méthodes de reconnaissance de texte pour des manuscrits anciens écrits en vieux français. Ces manuscrits sont souvent très difficiles à lire, même pour des initiés. Il s'agit donc d'automatiser une tâche déjà dure pour l'homme.

Le problème de reconnaissance de caractères (OCR) est une application classique des réseaux de neurones. C'est donc ce genre de technique que nous explorerons.

Nous commencerons par utiliser des moteurs de reconnaissance existants, à savoir `ocropy` et `Tesseract`. La deuxième partie du projet consistera soit à essayer d'améliorer le fonctionnement de l'un d'eux, soit à implémenter notre propre réseau de neurones.

Part II
Base théoriques et état de l'art

Insert ~~meme~~ relevant info here.

Part III
Reconnaissance avec ocropy

A Avant-propos

A.1 Présentation

GitHub du projet ocropy

> OCRopus is a collection of document analysis programs, not a turn-key OCR system. In order to apply it to your documents, you may need to do some image preprocessing, and possibly also train new models.
>
> In addition to the recognition scripts themselves, there are a number of scripts for ground truth editing and correction, measuring error rates, determining confusion matrices, etc. OCRopus commands will generally print a stack trace along with an error message; this is not generally indicative of a problem (in a future release, we'll suppress the stack trace by default since it seems to confuse too many users).

A.2 Principe général

Le principe d'entraînement, adapté aux manuscrits, est le suivant :

1. On récupère un ensemble de photos de pages d'un livre ;
2. On les binarise, c'est-à-dire que l'on met tous les pixels soit à 1 soit à 0 ;
3. On en extrait les lignes ;
4. On annote un maximum données ;
5. On sélectionne une partie des données annotées pour l'entraînement (90% des données annotées par exemple), et on garde le reste pour la validation ;
6. On entraîne le réseau de neurones sur les données d'entraînement ;
7. On valide sur les données de validation en regardant l'erreur.

B Premiers essais

B.1 Préparation des données

Afin de nous familiariser avec cet outil, nous avons commencé par binariser et extraire des lignes d'un manuscrit pris sur le site `http://www.e-codices.unifr.ch` nous-même. Ce premier test nous a permis de bien comprendre les premiers enjeux. En effet, arrivés à l'étape d'annotation des données, nous avons été confrontés à la difficulté que représente la traduction de ces manuscrits. Le processus est fastidieux puisque les mots mais aussi les caractères ont changé depuis.

Par la suite, nous avons abandonné ces données au profit de données annotées nous ayant été mises à disposition par l'ENC.

Nous avons ensuite travaillé sur un manuscrit de la *Chanson d'Otinel* écrit en anglo-normand au XIII$^{\text{ème}}$ siècle.

Il est d'une longueur de 46 pages, d'environ 30 lignes chacune. Nous avons pris environ 1300 lignes pour l'entraînement et 100 pour la validation.

B.2 Commandes basiques

La commande naïve à exécuter pour entraîner un modèle est :
`ocropus-rtrain -o <Model Name> <Training Images>`
On peut également visualiser les étapes avec la commande:
`ocropus-rtrain -o <Model Name> -d 1 <Training Images>`

A chaque itération, on a un output de la forme suivante :

Sur la première ligne se trouve d'abord le numéro de l'itération, une idée de l'erreur sur la ligne, puis quelques propriétés de l'image étudiée. L'information se contenant dans le lignes suivantes:
- `TRU` est la solution recherchée
- `ALN` est une version alignée de `TRU`

```
4004 26.45 (493, 48) train/0033/010016.bin.png
   TRU: u'e l g\u0363nt \u1ebdfern v g\u0131\u017fent l\u0131 larun'
   ALN: u'e l g\u0363nt \u1ebdfern v g\u0131~ent l\u0131 larun'
   OUT: u'e lgc ern o g\u0131~ent l\u0131 larun'
```

Figure 1: Output d'une itération de l'entraînement

- `OUT` est le résultat trouvé par le réseau de neurones
Un réseau de neurones idéal aurait les 3 lignes identiques.

Afin de lancer la validation d'un modèle et d'avoir le pourcentage d'erreur, on utilise les commandes suivantes :
- `ocropus-rpred -m <Model Name> <Training Images>`
- `ocropus-errs <Testing Truth>` permet d'évaluer la précision du modèle.
- `ocropus-econf <Testing Truth>` donne les erreurs les plus fréquentes. On peut ajouter `-C2` pour une contextualisation des erreurs.

Sur ce premier entraînement sans préparations, on obtient une erreur de l'ordre de 27% au bout de 12 000 itérations ce qui est extrêmement mauvais. En effet, l'ensemble des caractères sortant de l'ordinaire ne sont pas reconnus.

B.3 Entraînement avec les caractères spéciaux

B.3.1 Principe

Afin d'intégrer les caractères spéciaux à la résolution, la méthode est de les ajouter à la liste des caractères, stockée dans `chars.py`. C'est ce que nous avons fait, mais cela n'a rien changé. Nous avons compris par la suite que nos modifications effectuées étaient correctes, mais qu'il faut relancer la commande d'installation pour les prendre en compte :
`sudo python setup.py install`

B.3.2 Résultats

Statistics of the model	
errors	329
missing	35
total	3547
err	9.275 %
errnomiss	8.289 %

Table 1: Résumé de la commande `ocropus-errs`

C Amélioration des résultats par modification des données

C.1 Première normalisation

Nous avons opté pour une nouvelle méthode, consistant à repérer manuellement les caractères utilisés avec les fichiers textes contenant l'information de référence. On ajoute à la commande d'entraînement la ligne suivante :
`-c <Training Truth> <Testing Truth>`
Ceci indiquant qu'on va chercher les symboles dans les *ground truth* disponibles. Les trois commandes décrites précédemment nous permettent de remarquer qu'au bout de 5000 itérations, l'erreur est descendue à 20% seulement. La matrice de confusion nous a permis de voir que le caractère ſ n'était pas reconnu. Ceci étant du à une normalisation de l'unicode effectuée par la l'appel `-c`.
Pour résoudre ce problème nous avons implémenté un script `Python` remplaçant les caractères de nos données par leur normalisation *NFC*, et ainsi assurer la symétrie entre les données pouvant être apprises et la solution. Cette normalisation est utilisée par `ocropy` pour traiter les données lors de l'appel de la fonction `normalize-text` du module `common.py`. Cependant, la normalisation du caractère ſ étant le *s*, déjà présent dans les données, nous l'avons remplacé par un *Z*.

C.2 Analyse des premiers modèles normalisés

`Ocropy` nous permet de sauvegarder nos modèles à un nombre d'itération voulue, pour ce premier test nous les avons sauvegardé tous les 1000 itérations. Ensuite, un script `bash` utilisant les commandes de validations permet d'évaluer tous les modèles, et d'écrire les output de ces commandes dans un fichier `.txt`. Nous avons réalisé un script `Python` afin de réaliser le *post-processing* nécessaire au tracé des courbes d'erreurs.

La meilleure précision est obtenue pour 83000 itérations, on s'attarde donc un peu plus sur ce modèle particulier pour comprendre les défauts de l'apprentissage. La table 2 résume les statistiques importantes de ce modèle.

Statistics of the model	
errors	329
missing	35
total	3547
err	9.275 %
errnomiss	8.289 %

Table 2: Résumé de la commande `ocropus-errs`

Ce qui nous intéresse plus est contenu dans la table 3 et explique les défauts du réseau de neuronnes.

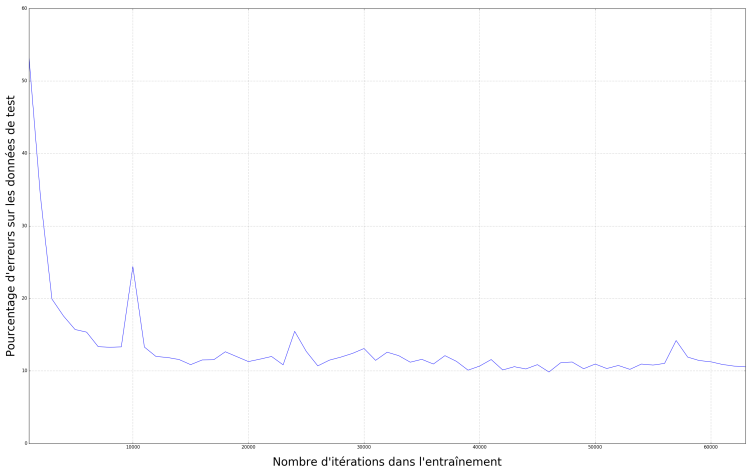

Figure 2: Évolution de l'erreur sur les données normalisées.

On distingue 3 types d'erreurs:

1. Les erreurs sur les espaces;
2. Les erreurs sur les caractères superscripts;
3. Les erreurs sur les lettres semblables.

Le premier peut sans doute être éliminé avec un peu de traitement des données, le second également mais causerait sans doute de la perte de sens. Le troisième est plus caractéristique de l'écriture puisque les lettres *u, n, m, i et l* se ressemblent souvent.

Most frequent confusions					
Without context			With context		
#	output	truth	#	ouput	truth
28		_	2	a δu	a_δu
12	_	´	2	uı$_t$	uı´ t
10	_		2	ıez	ıe-
6	_	a	1	e u la lt	e n la_lt
4	u	n	1	uı_ b	uí b
4	ı	_	1	Ztı	Ztr
4	t	_	1	aı_nZ	aínZ
4	ı	r	1	eg ler	egı_er
3	ı	u	1	O_gı	O gı
3	z	_	1	e uZ	e_uZ

Table 3: Résumé de la commande `ocropus-econf`

C.3 Amélioration de la normalisation

D Modification des paramètres d'entraînement

D.1 Commandes utiles

Pour modifier l'apprentissage, on peut ajouter une des lignes suivantes à la fin de la commande pour lancer l'entraînement, afin d'en modifier les propriétés.
`--ntrain <Number>` permet de modifier le nombre d'itérations réalisées dans l'entraînement. Plus ce nombre est grand, meilleur sera le résultat du réseau.
`--savefreq <Number>` permet de dire le nombre d'itérations entre chaque sauvegarde du modèle. Ceci nous permettra dans la suite de tracer les courbes d'erreur.
`--lrate <Number>` permet de modifier la vitesse d'apprentissage.
(*learning rate*). Un de nos premiers objectifs sera de trouver une bonne valeur de ce paramètre.
`--hiddensize <Number>` permet de modifier un des paramètres de la taille des matrices de notre réseau de neurones. Nous nous intéresserons également à son optimisation.

D.2 Modification de la vitesse d'apprentissage

D.3 Modification de la taille du réseau