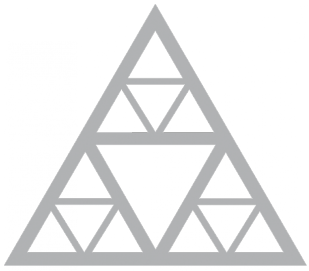

École des Ponts

ParisTech

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

Introduction

Dans leurs recherches, les historiens et chercheurs en littérature se retrouvent régulièrement confrontés à des parchemins ou des manuscrits dans des langues anciennes. Pour améliorer la qualité et la rapidité de leurs recherches, mais aussi pour sauvegarder ce patrimoine, des bases de données de connaissances et de ressources numérisées sont indispensables. La digitalisation des écrits est cependant un travaille fastidieux. La plupart du temps, seule une copie numérisée des documents est réalisée, alors qu'une réelle digitalisation nécéssiterait de taper manuellement tous les textes dans leur intégralité. La masse de travaille évidente qu'un tel processus implique rebute même si les gains seraient conséquents.

C'est ici qu'intervient la montée récente du Deep Learning. Depuis quelques années, l'essort qu'ont connu ces techniques d'apprentissage a été justifié par d'incroyables résultats, notamment en vision artificielle. La reconnaissance optique de caractères (OCR), réalisé pour la première fois par Gustav Tauschek en 1929 mais qui nécessitait que l'écriture soit très spéciale, a trouvé dans ces nouvelles méthodes un élan nouveau. De nombreux outils d'OCR ont vu le jour, cependant ceux-ci requierent souvent des connaissances préalables en machine learning et en informatique.

L'objectif de ce projet est d'étudier la faisabilité **d'outils pour la digitalisation de documents anciens qui seraient à la portée de tous**.

Nous commencerons d'abord par utiliser des moteurs de reconnaissance existants, à savoir `ocropy` et `Tesseract` afin d'avoir des résultats de référence. Ensuite nous discuterons de l'implémentation de nos propres systèmes de reconnaissance de caractères. Enfin nous présenterons notre solution logicielle, conçue pour être le plus simple pour des utilisateurs voulant des résultats consistants rapidement.

Table des matières

Moteurs OCR existants

1.1 ocropy

1.1.1 Présentation d'ocropy

GitHub du projet ocropy : https ://github.com/tmbdev/ocropy

OCRopus est une collection d'outils pour l'analyse de document, comprenant notamment des scripts d'édition des valeurs de vérité (*ground truth*), d'évaluation de taux d'erreurs et de matrices de confusion, etc. **ocropy** est un projet d'outils de ligne de commande, programmé en Python, permettant de réaliser ces opérations en ligne de commande.

Pour obtenir une version digitale du document, on utilise des **modèles** de prédiction, déjà existant ou à créer soi-même. Pour les manuscrits anciens, il faut la plupart du temps créer son modèle. C'est ce qu'on appelle **l'entraînement**.

1.1.2 Principe et méthodologie

La méthodologie pour créer un modèle est la suivante :

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 extrait les lignes ;
4. On annote à la main un maximum de lignes ;
5. On sépare les données annotées en deux : une partie des données servira pour la création et l'amélioration du modèle (set de données d'**entraînement**), et l'autre partie permettra de vérifier si le modèle fonctionne bien (set de données de **validation**) ;
6. On créer le modèle (c'est-à-dire que l'on entraîne le réseau de neurones sur le set de données d'entraînement) ;
7. On valide sur le set de données de validation en étudiant le taux d'erreur et les confusions faites par le modèle.

Le modèle ainsi obtenu permet de faire des **prédictions**, c'est-à-dire de calculer pour une image donnée la chaîne de caractères qu'il considère la plus probablement écrite.

1.1.3 Premiers essais

Nous avons dans un premier temps téléchargé et installé ocropy. Le but fut d'abord de comprendre comment l'utiliser.

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.

Par la suite, nous avons abandonné ces données au profit de données annotées nous ayant été mises à disposition par l'ENC. Parmi ces données, nous avons commencé par travailler sur un manuscrit de la *Chanson d'Otinel* écrit en anglo-normand au XIIIème 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.

Commandes basiques

La commande naïve à exécuter pour entraîner un modèle est :

```
ocropus-rtrain -o <ModelName> train/*/*.bin.png
```

On peut également visualiser les étapes avec la commande :

```
ocropus-rtrain -o <ModelName> -d 1 train/*/*.bin.png
```

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

```
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 1.1 – Output d'une itération de l'entraînement

Sur la première ligne se trouve d'abord le numéro de l'itération, 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`

— `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 <ModelName> test/*/*.bin.png
```

`ocropus-errs test/*/*.gt.txt` permet d'évaluer la précision du modèle.

`ocropus-econf test/*/*.gt.txt` 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.

Entraînement avec les caractères spéciaux

Afin d'intégrer les caractères spéciaux à la résolution, la méthode est de les ajouter à la liste des caractères, stockée dans `char.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
```

Ceci nous a permis d'obtenir une première courbe d'erreur satisfaisante. Le tracé est réalisé en analysant les modèles sauvegardés toutes les 1000 itérations, et en inscrivant les résultats dans des documents `.txt`. Un *parsing* avec python permet de récupérer les informations utiles de ceux-ci. Le temps nécessaire pour effectuer 1000 itérations est d'environ 11 minutes sur l'ordinateur que nous avons utilisé pour notre installation `ocropy`. C'est un portable MSI doté d'une NVIDIA GeForce GTX 1060, le GPU étant le plus important pour entraîner un réseau de neurones.

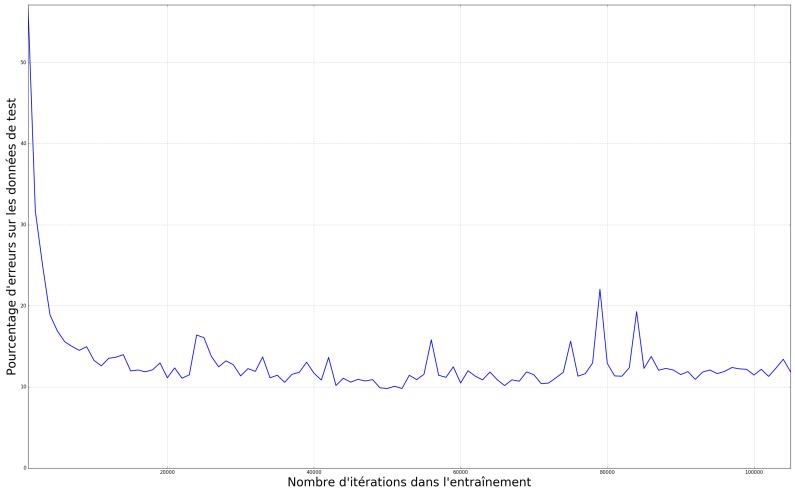

Figure 1.2 – Évolution de l'erreur sur les données comportant des caractères spéciaux.

Le meilleur modèle est obtenu au bout de 50000 itérations, et atteint une erreur de 9.811%. Ces résultats sont corrects aux vues de la complexité des données, mais nous pensons pouvoir faire mieux avec un peu de pré-traitement et en modifiant certains paramètres.

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

Table 1.1 – Output de la commande `ocropus-errs`

1.1.4 Amélioration des modèles

Premières normalisation

Nous avons donc opté pour une nouvelle méthode, consistant à repérer manuellement les caractères utilisés avec les fichiers textes contenant l'information de référence. On utilise ajoute à la commande d'entraînement la commande suivante :

```
-c train/*/*.gt.txt test/*/*.gt.txt
```

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*.

Analyse des premiers modèles normalisés

La figure 1.3 montre les améliorations obtenues grâce à ces premières normalisations.

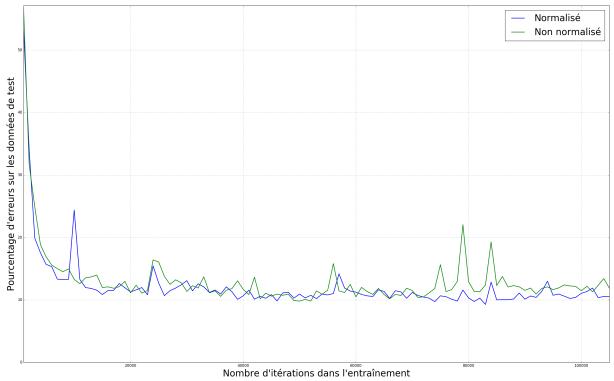

Figure 1.3 – Évolution de l'erreur sur les données normalisées.

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 1.2 résume les statistiques importantes de ce modèle.

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

Table 1.2 – Résumé de la commande `ocropus-errs`

Ce qui nous intéresse plus est contenu dans la table 1.3 et explique les défauts du réseau de neurones. 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 δ u	a_δ u
12	_	´	2	uɪ_t	uɪ´ t
10	_		2	ɪez	ɪe-
6	_	ª	1	e u la lt	e n la_lt
4	u	n	1	uɪ_ b	uí b
zsh :1 : command not found : q 4	t	_	1	aɪ_nZ	aínZ
4	ɪ	r	1	eg ler	egɪ_er
3	ɪ	u	1	O_gɪ	O gɪ
3	z	_	1	e uZ	e_uZ

Table 1.3 – Résumé de la commande `ocropus-econf`

Amélioration de la normalisation

1.1.5 Modification des paramètres d'entraînement

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. Par défaut il vaut 10^6, mais en général nous nous arrêtons vers 10^5 itérations.

`--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. Le paramètre par défaut est 1000 et est un bon compromis entre la précision des courbes et le temps passé à tester les modèles.

`--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.

Dans les sous-sections suivantes, nous travaillons avec les données obtenues avec la première normalisation.

Modification de la *learning rate*

On rappelle que la *learning rate*, vitesse d'apprentissage, ou vitesse d'entraînement, est un hyper-paramètre qui caractérise le pas dans la descente du gradient stochastique présente dans le réseau de neurone. Les courbes figure 1.4 comparent les modèles pour la valeur par défaut de la vitesse d'apprentissage, 10^{-3}, et pour une valeur de 10^{-4}.

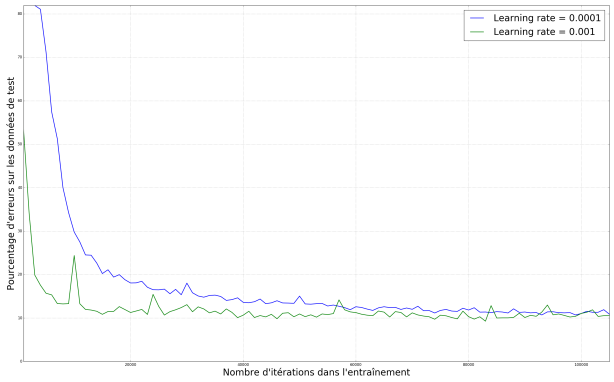

Figure 1.4 – Évolution de l'erreur pour deux vitesses d'apprentissage différentes.

On remarque que la courbe au *learning rate* plus faible est plus lente à converger, mais plus régulière. En raison de cette stabilité, le meilleur modèle, obtenu pour 133000 itérations a un taux d'erreur de 9.949% pour la *learning rate* la plus faible, contre 9.275% au bout de 83000 itérations dans l'autre cas. C'est pourquoi nous garderons une *learning rate* de 10^{-3} dans la suite.

Modification de la taille du réseau

Le paramètre sur lequel on joue ici est le paramètre définissant le nombre de *hidden state*. Ce sont les variables que le *LSTM* utilise pour garder une trace du temps. C'est la partie récurrente du réseau. L'influence de ce paramètre est présentée figure 1.5.

On remarque que les résultats sont meilleurs quand on augmente ce paramètre. La convergence est un peu plus lente en nombre d'itérations, cependant, nous sommes passé de 12 à 18 minutes de temps de calcul nécessaire pour effectuer 1000 itérations. Le meilleur modèle que nous avons obtenu a une erreur de 7.976%, atteinte pour 95000 itérations, soit plus d'un jour d'entraînement.

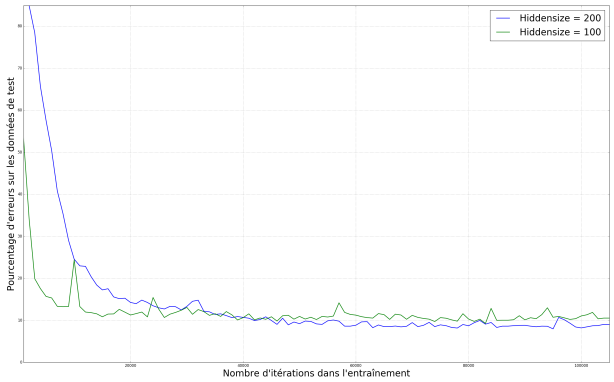

Figure 1.5 – Évolution de l'erreur pour deux tailles de réseau différentes.

1.1.6 Commandes bash utilisées

```
ocropus-rpred -m test3-00005000.pyrnn test/*/*.bin.png
ocropus-errs test/*/*.gt.txt

ocropus-rtrain -o <modelname> <training-dir>/*/*.bin
ocropus-rtrain -o myModel -d 1 train/*/*.bin.png

for i in test8/*.pyrnn.gz; do
    echo "\$i" >> results.txt
    ocropus-rpred -m \$i test/*/*.bin.png
    ocropus-errs test/*/*.gt.txt >> results.txt
    echo "\$i" >> index.txt
done

sudo python setup.py install

ocropus-rtrain -o test8 train2/*/*.bin.png --load test8-00086000.pyrnn

ocropus-rtrain -c train/*/*.gt.txt test/*/*.gt.txt -o test7 train/*/*.bin.png --loa

ocropus-rpred -m test8-00050000.pyrnn test/*/*.bin.png
```

Modification de la vitesse d'apprentissage

Modification de la taille du réseau

1.2 Tesseract

1.2.1 Présentation de Tesseract

Tesseract est un outil d'OCR, développé par HP en 1985. Il est depuis devenu Open Source, et est maintenu par Google depuis 2006. Il est souvent considéré comme l'un des meilleurs moteur d'OCR.

Utilisation de Tesseract

Tesseract s'utilise comme ocropy en ligne de commande. L'utilisation la plus simple est pour lire le texte d'une image dans une langue pour un modèle existant.

```
tesseract imagename|stdin outputbase|stdout
```

Google ayant entrainé des modèles dans plus de 50 langues sur des millions de caractères pour des milliers de polices différentes, les résultats sont incroyables.

Entrainement de Tesseract

C'est en voulant entrainer Tesseract que nous avons constaté de nombreux défauts.

Dans sa récente version incluant les réseaux LSTM, créer des modèles relève du défi aussi bien technique que psychologique. La documentation à ce propos est inintelligible pour les personnes n'étant pas initiée aussi bien dans la vision artificielle qu'à l'utilisation avancée des versions précédentes de Tesseract, mais elle est en plus incomplète et indigeste. Une des raisons à cela serait probablement que l'outil est encore en version alpha, mais aussi que la plupart des personnes autour du projet considèrent que le code en lui-même constitue un élément suffisant pour pouvoir savoir entrainer des modèles soi-même.

Tesseract est donc un puissant outil pour la reconnaissance de caractères dans des langues ayant déjà été entrainées, mais dans le cadre de notre problème il ne semble pas adapté. En effet, il nous faudrait un outil adaptable à de nombreuses langues et écritures.

Création d'une solution OCR

2.1 Principe du réseau de neurone appliqué à l'OCR

Parce que c'est BUENO