

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 travail fastidieux. La plupart du temps, seule une copie numérisée des documents est réalisée, alors qu'une réelle digitalisation nécessiterait de taper manuellement tous les textes dans leur intégralité. La masse de travail 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'essor 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 requièrent 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

1	Moteurs OCR existants	4
1.1	ocropy	4
1.1.1	Présentation d'ocropy	4
1.1.2	Principe et méthodologie	4
1.1.3	Le réseau utilisé	5
1.1.4	Premiers essais	5
1.1.5	Modification des paramètres d'entraînement	7
1.2	Tesseract	10
1.2.1	Présentation de Tesseract	10
2	Création d'une solution OCR	11
2.1	Principe du réseau de neurone appliqué à l'OCR	11

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. **Ocropy** utilise des réseaux de neurones, notamment des réseaux de neurones récurrents (*LSTM*) pour détecter les caractères.

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.

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 entraîne le réseau de neurones sur le set de données d'entraînement ;
7. On valide sur l'ensemble 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 Le réseau utilisé

Comme mentionné précédemment, **ocropy** utilise les réseaux de neurones pour reconnaître les caractères. Nous nous sommes donc approprié le code source afin de comprendre brièvement le fonctionnement. Remarquons déjà que **ocropy** n'utilise pas de bibliothèque de *Deep Learning* auxiliaire et que les réseaux ont été ré-implémentés manuellement. (cf fichier `lstm.py`). Le code est basé sur une classe mère *Network* pour laquelle on définit les méthodes usuelles. Ensuite, chaque type de réseau est défini dans une classe fille (*Softmax*, *MLP*, *LSTM*). A ceci sont ajoutées les classes permettant de combiner les réseaux (*Stacked*, *Parallel*, *Reversed*).

Le réseau utilisé est un *LSTM bidimensionnel*, représenté figure 1.1.

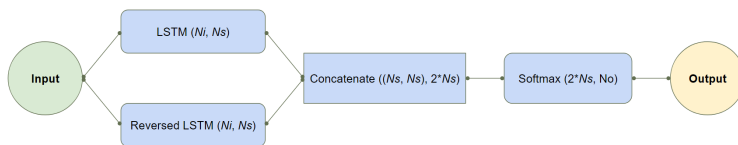


FIGURE 1.1 – Le réseau utilisé par **ocropy**

L'entrée est un *batch* d'images dont la largeur est variable mais la hauteur est fixée à N_i . N_s est le nombre de cellules d'états dans le *LSTM*, et également la taille de la sortie de ce réseau. Enfin, N_o est le nombre de classes.

L'entrée est envoyée dans un *LSTM* qui parcourt l'image de gauche à droite et un autre "renversé" qui la parcourt de droite à gauche. Les deux sorties sont ensuite concaténées pour être envoyées dans un layer *Softmax* pour obtenir les classes.

La fonction de perte utilisée est la *CTC-loss* (*Connectionist Temporal Classification*) qui est une fonction de perte classiquement utilisée pour les réseaux récurrents. Nous reviendrons dessus plus tard.

1.1.4 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'Otinél* écrit en anglo-normand au XIII^{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.

Commandes basiques

La commande naïve à exécuter pour entraîner un modèle est :

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

On peut également visualiser les étapes avec la commande :

```
ocropus-rtrain -o <ModelName> -d 1 <training-dir>/**/*.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 \u1ebdferr v g\u0131\u017fent l\u0131 larun'
ALN: u'e l g\u0363nt \u1ebdferr v g\u0131~ent l\u0131 larun'
OUT: u'e lgc ern o g\u0131~ent l\u0131 larun'
```

FIGURE 1.2 – 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 les lignes suivantes :

- TRU est la solution recherchée
- ALN est une version alignée de TRU, adaptée à la *CTC-loss*
- OUT est le résultat trouvé par le réseau de neurones

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-dir>/**/*.bin.png
```

`ocropus-errs <test-dir>/**/*.gt.txt` permet d’évaluer la précision du modèle.

`ocropus-econf <test-dir>/**/*.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.

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.

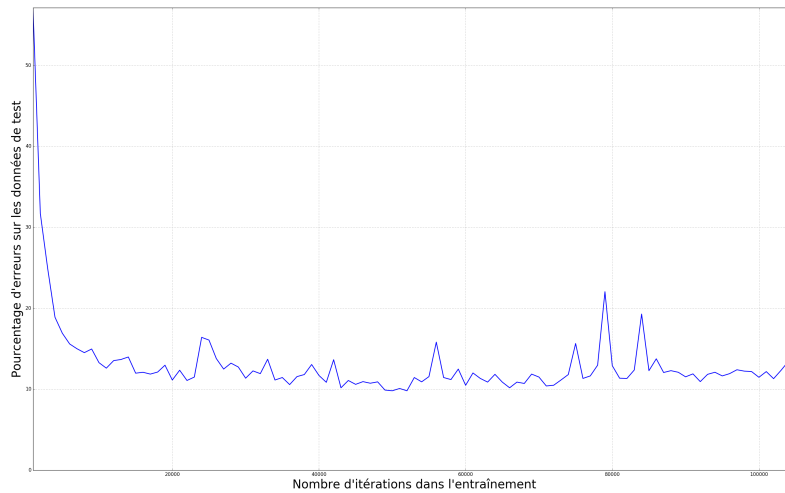


FIGURE 1.3 – Évolution de l'erreur sur les données comportant des caractères spéciaux.

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`

1.1.5 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. 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} .

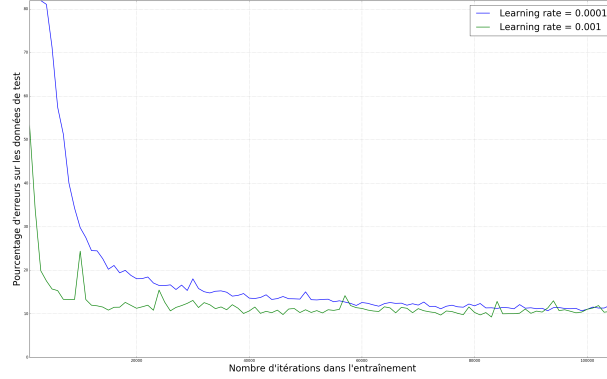


FIGURE 1.4 – Évolution de l'erreur pour deux vitesses d'apprentissage différentes.

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.

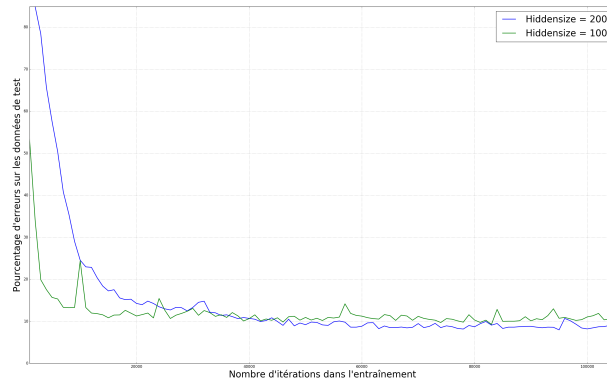


FIGURE 1.5 – Évolution de l'erreur pour deux tailles de réseau différentes.

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.

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

Entraînement de Tesseract

C'est en voulant entraîner 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 entraîner des modèles soi-même.

Tesseract est donc un puissant outil pour la reconnaissance de caractères dans des langues ayant déjà été entraînées, 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