Devoir Industrialisation

M2 – Linguistique Informatique

LeCloirec Hafida
Vallois Benjamin
Ahamadi Halima
Fromont Léonard

Table des matières

Objectif du projet :

Dans ce dossier, nous allons détailler les étapes que nous avons suivies pour réaliser la tâche de segmentation d'offre d'emploi en un schéma précis : Titre / Description de l'entreprise / Description du poste / Description du profil recherché.

Introduction:

Que ce soit dans les réseaux d'entreprise ou sur le web, les données RH en version électronique sont en constante augmentation. Elles nécessitent des outils informatiques et de TAL pour les traiter. Nous allons nous pencher sur la gestion des offres d'emploi et plus précisément sur le traitement automatique de leur segmentation en sous-parties spécifiques. Nous utilisons des approches sémantiques et d'apprentissage pour les segmenter. L'idée est d'analyser automatiquement les caractéristiques dispositionnelles des offres et de les segmenter en conséquence. Nous nous sommes inspirés des méthodes de tagging pour la segmentation. Chaque phrase possédant un tag, nous déterminons la frontière par le changement de tag entre deux phrases à la suite.

Description et prétraitement du corpus:

Nous disposons d'un corpus de train de 50 000 offres et d'un fichier de test de 10 offres

Ces offres sont dans un fichier csv et contiennent des balises HTML. Nous décidons de ne pas pré-traiter ces balises et de laisser notre système d'apprentissage décider de leur importance.

Une lecture classique du fichier csv (via par exemple la fonction read_csv de la bibliothèque panda) ne permet pas d'obtenir les offres au complet. En effet, certaines lignes sont de formats différents.

Nous avons donc opté pour une lecture ligne à ligne afin d'être sûrs de récupérer toutes les offres.

Afin d'extraire les informations utiles, nous pré-traitons les données en les splittant pour obtenir une sorte de fichier gold pouvant permettre ainsi un apprentissage supervisé.

Ainsi, sur :

- ➔ le fichier de train, nous splittons chaque offre en 5 segments [id_offre, job_title, company_Desc, job_Desc, profile_Desc]
- ➔ le fichier de test, nous splittons chaque offre en 2 segments [id_offre, "texte de l'offre"]

Puis, nous découpons les offres en phrases taguées sur les points, les balises
 et les fermetures de balises.
Nous obtenons ainsi une liste de tuples (id_offre, liste de phrases taguées) de la forme suivante: [(id_offre, [(phrase, tag), …])] qui nous permettra d'alimenter notre système d'apprentissage.

Analyse des offres:

Les offres d'emploi se composent de blocs similaires. Ainsi, une offre se compose de quatre sections:

1. job_title: Titre de l'emploi
2. company_Desc: bref résumé de l'entreprise qui recrute
3. job_Desc : courte description de l'emploi ;
4. profile_Desc : qualifications et connaissances exigées pour le poste. Les contacts sont généralement inclus dans cette partie.

Dans une offre, il peut y avoir plusieurs segments (ou phrases) de même type qui sont consécutifs. Ainsi, nous pouvons trouver la segmentation suivante: (job_title, company_desc, job_desc, job_desc, job_desc, job_desc, profile_desc, profile_desc, profile_desc), mais non (job_title, company_desc, job_desc,company_desc,profile_desc…). Les différentes parties se succèdent dans un ordre précis. Ce qui facilite la segmentation, dès qu'un tag change, c'est une nouvelle partie.
Dans notre programme, nous coderons les sections de la façon suivante:
0 -> job_title, 1 -> company_Desc, 2 -> job_Desc, 3 ->profile_Desc.

Ci-dessous un récapitulatif du contenu du corpus de train d'offres d'emploi:

Nombre total d'offres d'emploi	50 000
Nombre total de segments	807 670
Nombre de segments étiquetés job_title	50 118 (~6%)
Nombre de segments étiquetés company_desc	116 918 (~14%)
Nombre de segments étiquetés job_desc	407 828 (~51%)
Nombre de segments étiquetés profile_desc	232 806 (~29%)

Pré-traitement linguistique (Split)

Nous segmentons d'abord chaque offre en paragraphes ou phrases.
Pour cela nous avons testé 2 méthodes:

La première méthode consiste à spliter nos offres entières sur les points ou balises de <p> ou
 mais cela donnait un résultat assez approximatif car les balises ne correspondaient pas forcément à une séparation de partie. Nous avons alors corrigé ce split en reconstituant certaines parties isolées (par exemple des
 ou des espaces isolés), via un nettoyage. Nous avons aussi attribué pour chaque partie de notre offre une classe : {0, 1, 2 ,3} pour job, desc entreprise, desc poste, desc profil, en fonction de leur position dans l'offre.

La seconde méthode consiste en l'utilisation du module nltk.split pour splitter en phrase. Pour cela, nous avons dû remplacer les codes html par leur équivalent en caractère, car le split par phrase ne fonctionnait pas tout à fait avec ceux-ci. Cependant nous perdions certains espaces ainsi que quelques caractères spéciaux de l'offre originale.

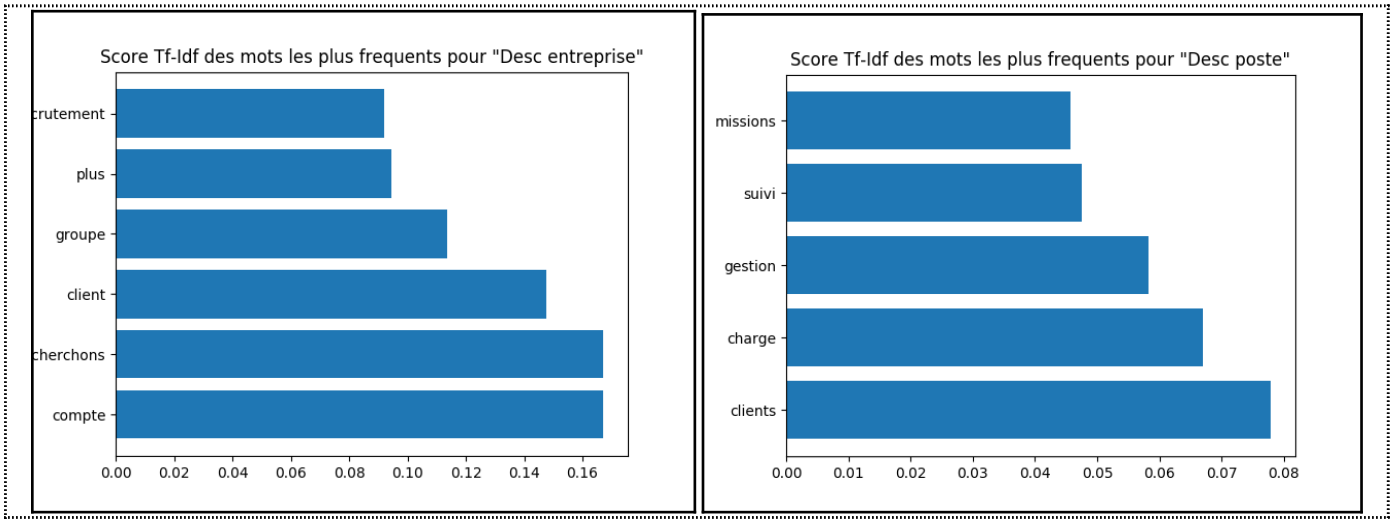
Nous avons au final opté pour la première méthode, avec quelques améliorations, c'est-à-dire splitting sur les balises de fin uniquement </"nom de balise"> ou
 ou ". " avec espace derrière, car cela nous permettait de reconstituer l'offre telle quelle.

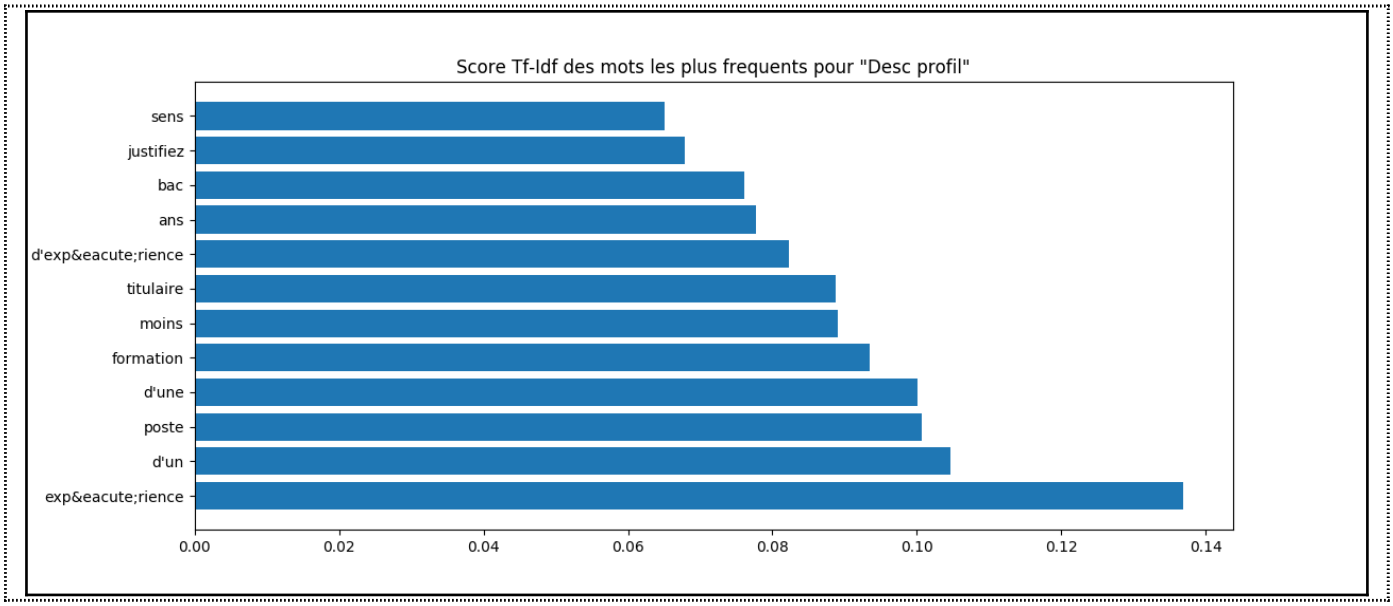
Puis nous procédons à une étape de tokenisation en mot de chaque phrase auxquels nous appliquons des traitements additionnels.
En effet, nous cherchons à conserver les mots discriminants, c'est-à-dire des mots utiles pour la segmentation.
Nous sommes passés d'une représentation bag-of-words à une représentation Tf-Idf pour avoir une meilleure information tout en réduisant la taille du vocabulaire.
Nous tokenisons sur les espaces, et nous ne conservons que les mots ayant un score df (document frequency) de plus de 500, c'est-à-dire qui apparaissent dans au minimum 500 parties d'offres. Ainsi, nous nous débarrassons des mots apparaissant dans trop peu de documents et d'un lexique de 131 150 mots, nous passons à un lexique de 1 613 mots.
Nous obtenons donc une matrice lexique*tf-idf-score à laquelle nous appliquons divers processus afin d'en réduire la dimensionnalité:

- uniformisation de la casse
- filtrage: suppression des stopwords français fournis par la librairie nltk, et de la ponctuation.

Cette diminution de la dimension de l'espace permet d'avoir des temps de calculs plus rapides.
Lorsque l'on regarde en détail les mots discriminants par section d'offre, nous constatons que pour l'intitulé de l'offre, le mot "h/f" ressort clairement.

Pour les autres sections, ci-dessous des graphiques représentant les mots discriminants et leur score tf-idf:



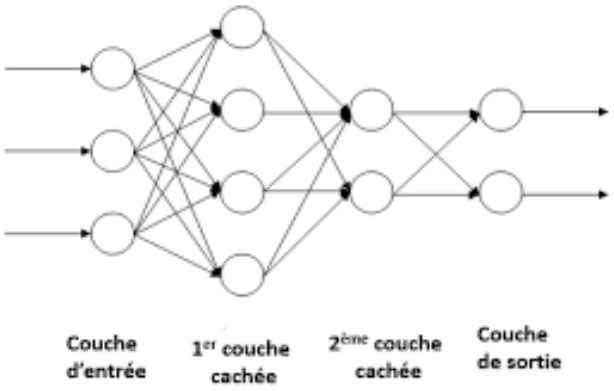


Méthode d’apprentissage

Nous avons choisi de réaliser un réseau de neurones pour réaliser une tâche de tagging. Avec pour but de classer chaque phrase dans une catégorie correspondant à une section d’offre. Nous testons deux méthodes: MLP (multi-layer perceptron) et LSTM (Long short-term memory).

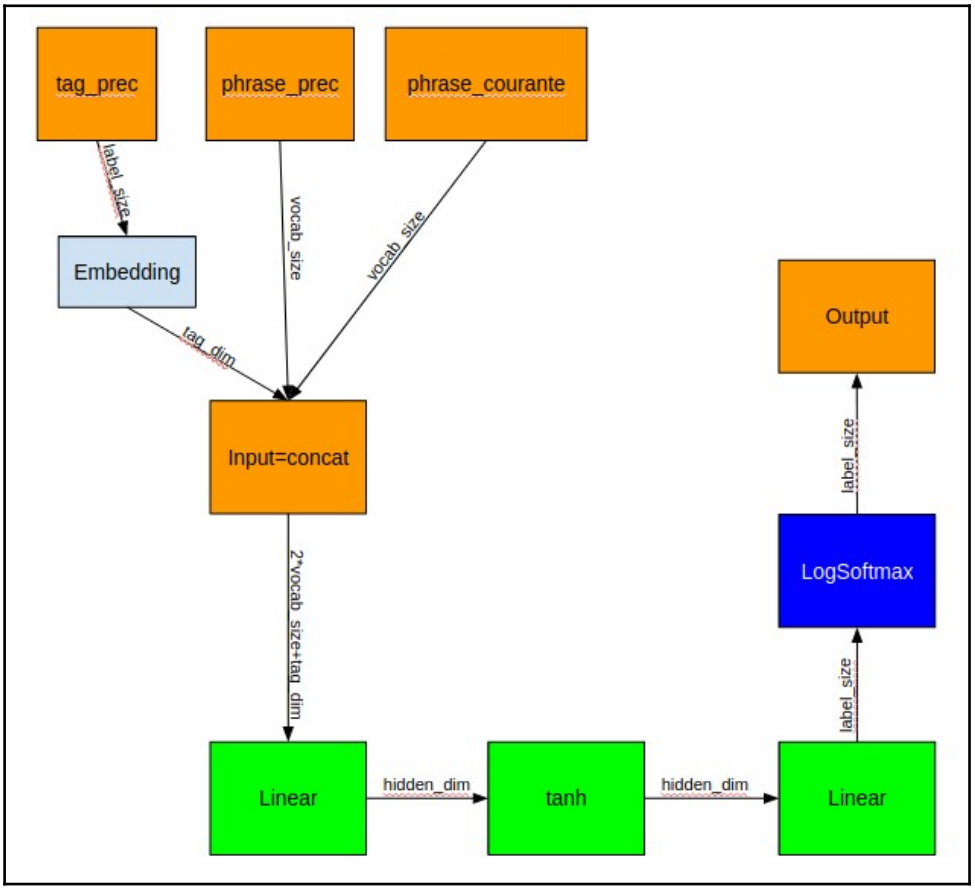
MLP: explication du réseau

Le perceptron multicouche (multilayer perceptron MLP), inventé en 1957 par Frank Rosenblatt, est un type de réseau neuronal artificiel organisé en plusieurs couches au sein desquelles une information circule de la couche d'entrée vers la couche de sortie uniquement ; c'est un réseau à propagation directe. Chaque couche est constituée d'un nombre variable de neurones, les neurones de la dernière couche (dite « de sortie ») étant les sorties du système global.



Nous avons choisi, comme input pour ce modèle, un vecteur qui correspond a la concaténation de la représentation du label de la phrase précédente, ainsi que celle de cette dernière, plus celle de la phrase en cours d’analyse.

Ce vecteur est ensuite passé dans un réseaux à plusieurs couches représenté ci-dessous:



Ici label_size=4, vocab_size=1631, tag_dim=1000, hidden_dim=100.

La fonction de perte choisie pour notre réseau de neurones est une NLLLoss(), fonction plus adaptée à une classification multiclasse.

La catégorisation de segments sans considérer leur position dans la phrase peut-être source d'erreurs. En effet, même si nous prenons en compte la phrase précédente et le label prédit précédent, nous ne gardons que la position relative du segment. Ainsi, nous constatons que le MLP produit globalement une bonne classification des segments individuels, mais les segments d'une même offre d'emploi sont rarement tous correctement étiquetés.

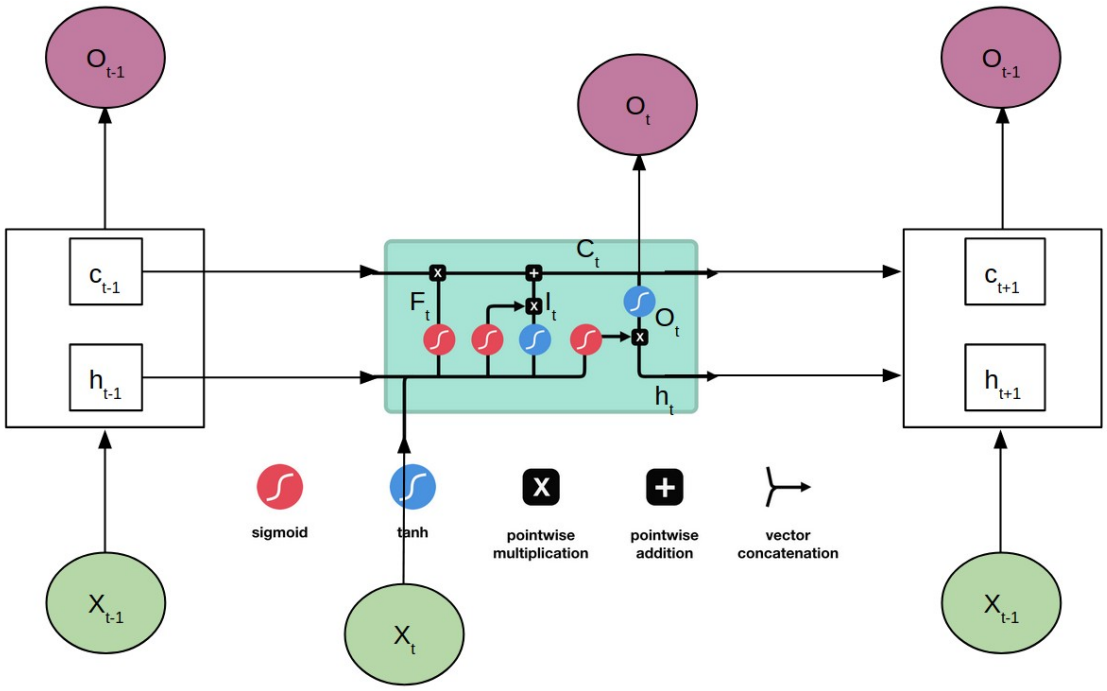
LSTM: explication du réseau

Les "Long Short Term Memory" ou réseaux de mémoire à long et court terme, sont un type spécial de réseaux de neurones récurrents, capable d'apprendre les dépendances à long terme. Ils ont été introduits par Hochreiter & Schmidhuber (1997). Ils fonctionnent extrêmement bien pour une grande variété de problèmes, et sont maintenant largement utilisés.

De plus, plusieurs études montrent que les LSTM peuvent apprendre des séquences spécifiques comme le langage $a^n b^n c^n$, [Felix A. Gers and Jürgen Schmidhuber. 2001. *LSTM recurrent networks learn simple context-free and context-sensitive languages*], ou la gestion des parenthèses dans un modèle de langue, [Hermans, Michiel and Schrauwen, Benjamin. 2013. *Training and analysing deep recurrent neural networks*], ce qui nous a motivé à choisir ce modèle pour traiter notre problème de séquence d'une offre, qui est de type $0^n 1^m 2^p 3^q, avec n, m, p, q > 0$.

Une couche LSTM est constituée d'une cellule, d'une porte d'entrée, d'une porte de sortie et d'une porte d'oubli. La cellule se souvient des valeurs sur des intervalles de temps arbitraires et les trois portes régulent le flux d'informations entrant et sortant de la cellule. La porte d'entrée autorise ou bloque la mise à jour. Celle de sortie contrôle si l'état de cellule est communiqué en sortie de l'unité LSTM. La porte d'oubli permet la remise à zéro de l'état de la cellule.

Ces réseaux peuvent être représentés comme suit :



Où F_t, I_t et O_t représentent respectivement les portes d'oubli, d'entrée et de sortie, C_t la cellule et h_t l'état caché du LSTM.

Notre LSTM prend en input un vecteur qui correspond à la concaténation de la représentation de toutes les phrases pi, soit les phrases à la position i de chaque offre (i de 0 à length of taille de l'offre).
Ce vecteur est ensuite passé dans un réseaux à trois couches avec une couche LSTM d'entrée de dimension (taille du vocab, taille dimension cachée=100) qui fournit des états cachés. Puis il est passé dans une couche cachée de dimension (taille dimension cachée*2=200, nombre de tags=4), la dimension cachée possède un facteur 2 car nous avons opté pour un LSTM bi-directionnel. En effet, au tout début de notre implémentation, sur de petits ensembles d'entraînement, nous avons remarqué que le dernier tag dans la séquence n'apparaissait pas toujours dans dans les prédictions avec un LSTM uni-directionnel, alors qu'avec le bi-directionnel, tous les tags sont prédits.
Et enfin en sortie, une couche appliquant une fonction log_softmax qui calcule le score de chaque tag.

La fonction de perte choisie est également une NLLLoss(), pour les mêmes raisons que citées ci-dessus pour le MLP.

Mode d'évaluation

Nous optons pour deux mesures d'évaluation:

- ➔ Accuracy sentences: qui comptabilise le nombre de phrases bien taguées par rapport au nombre de phrases du gold.
- ➔ Accuracy parts: qui comptabilise le nombre de phrase bien taggées par partie, par rapport au gold.
- ➔ Order evaluation : qui vérifie qu'en sortie nous avons bien une séquence ordonnée de tags ($0^n 1^m 2^p 3^q$)

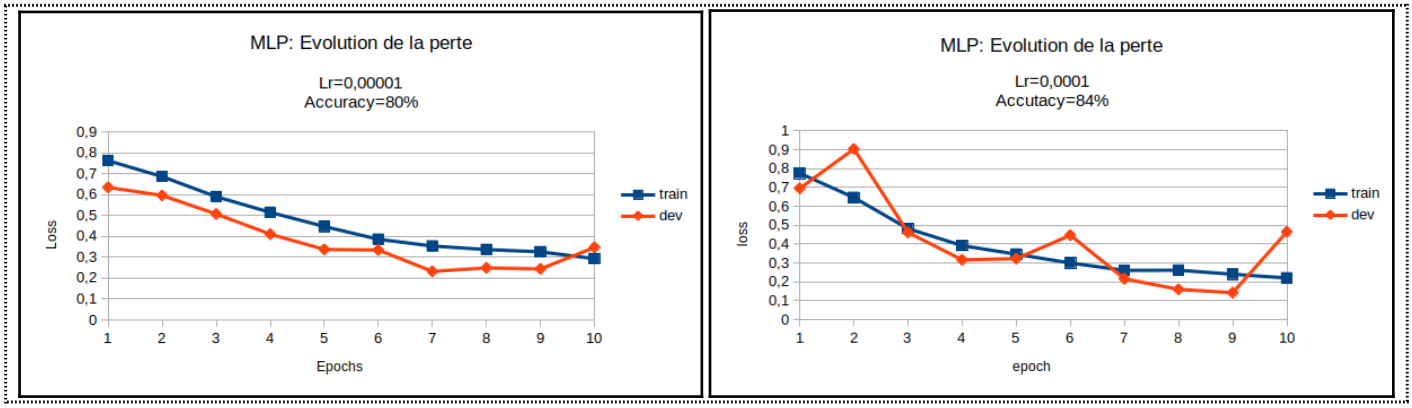
Calibrage de l'apprentissage et difficultés rencontrées

Nous avons testé nos réseaux avec différents taux d'apprentissage (learning rate) et différents nombres d'epochs.

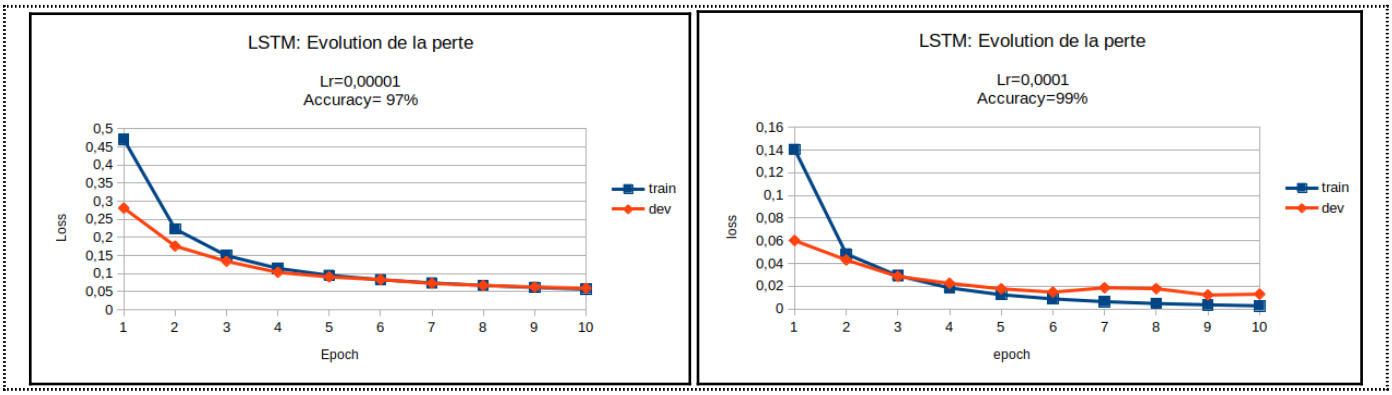
Nous les avons fait tourner sur 5/10 epochs et sur différents learning rate. Nous avons au final opté pour 5 epochs et un learning rate de 10^{-4} (avec des learning rate supérieurs, on tombe à des accuracy autour de 50%, avec des learning rate inférieurs, on n'améliore pas l'accuracy).

Ci-dessous des graphiques représentant l'évolution de la perte, en fonction du nombre d'époch et du learning rate.

MLP:



LSTM:



Nous avons entraîné nos réseaux de différentes façons.
Dans un premier temps, les temps de traitement étant trop longs, nous avons décidé d'utiliser la plateforme "Google collab" car elle dispose d'un GPU qui peut améliorer grandement les temps de traitement.

Cependant, nos réseaux, malgré l'utilisation du GPU, mettaient toujours beaucoup de temps à tourner.
Nous avons alors opté pour la création de mini-batchs, via le dataloader de pytorch, que nous lancions par paquets lors d'une epoch. Cela a amélioré les temps de traitement, mais malheureusement nos résultats s'en trouvaient impactés, car nous obtenions des accuracy inférieures à 50%.

Nous avons constaté que c'était à cause du fait que, le nombre de phrases par offre étant variable, le dataloader coupait les offres dans un batch en les limitant à l'offre du batch ayant la longueur la moins élevée. Nous avons alors artificiellement uniformisé les longueurs d'offres par batch en ajoutant des 0 aux vecteurs représentant les offres les moins longues. Nous obtenons une accuracy par phrases de 53%.

Nous avons alors opté pour une diminution drastique de la taille du lexique comme expliqué dans la partie pré-traitement linguistique. Cela nous a fait passer à une taille de vocabulaire d'~1600 et a permis de réduire les temps de calcul, et repasser à un apprentissage stochastique (un exemple par update).

De plus contrairement à ce que nous pensions le LSTM n'a pas bien généralisé l'ordre de la séquence . En effet, nous avons remarqué des erreurs dans l'ordre des tags. Ce qui nous a amené à écrire une fonction de réordonnancement sur nos prédictions, avec un léger impact négatif sur l'accuracy.

Résultats

Nous avons effectué une évaluation selon deux niveaux de finesse : par segment (accuracy_sentences) et par offre d'emploi entièrement reconnue (accuracy_parts).

Nous testons sur les modèles:

- ➔ MLP sans batch (car moins performants en terme de résultats) et sur 1000 exemples (car trop long) et
- ➔ LSTM sans batch et sur la totalité du corpus.

Nous obtenons les résultats suivants:

	MLP (sur 1000 exemples)	LSTM (sur tout le corpus)
Accuracy sentences	82%	95%

LSTM	Output (en %)	Input (en %)
Accuracy sentences	98	95
Accuracy parts	0: 100 1: 93 2: 100 3: 100	0: 100 1: 95 2: 94 3: 93
Accuracy sentences ordered	97	95
Accuracy parts ordered	0: 100 1: 86 2: 100 3: 100	0: 100 1: 90 2: 97 3: 93

Nous constatons que le LSTM prédit beaucoup mieux que le MLP, ce qui n'est pas étonnant car:

1. le MLP catégorise les segments sans considérer leur position dans la phrase. En effet, dans notre modèle, nous ne prenons en compte que la position relative du segment.
2. le LSTM fonctionne bien avec des séquences et prend en compte la position absolue du segment dans la phrase.

Ainsi, nous constatons que le MLP produit globalement une bonne classification des segments

individuels, mais les segments d'une même offre d'emploi sont rarement tous correctement étiquetés. En effet, le modèle arrive bien prédire les tags 0 et 1. Mais pour les catégories 2 et 3, il y a plus d'erreurs. Il prédit souvent des 3 à la place des 2. Et vice versa. Et ne prédit pas bien les frontières de catégories 2 et 3 car il peut prédire un segment 2 puis le segment suivant 3 puis revenir au segment 2. Ce qui est consistant avec la mémoire à court-terme du MLP, dès qu'on passe aux catégories, 2 et 3, le modèle perd la mémoire des catégories 0 ou 1 prédites précédemment.

Pour le MLP, il est été donc difficile de prédire une séquence complète. De plus, les temps de traitement sont conséquents et l'optimisation est difficile car il aurait fallu prendre plus de contexte autour de notre phrase.

Pour le LSTM, les prédictions sont bonnes mais les temps de traitement, bien que moins importants que le MLP, restent toujours importants. Nous avons choisi de tester à la fois input.csv et output.csv, le premier possédant 4 balises </ br> supplémentaire, et avons constaté que les résultats diffèrent un peu.

Nous optons donc au final pour le modèle LSTM.

Conclusion

Le traitement des offres d'emploi est une tâche difficile car l'information y est fortement non structurée.
Les premiers résultats obtenus par le LSTM sont très intéressants entre 95% et 98% de segments correctement étiquetés, selon le fichier en entrée choisi .
Cependant, les temps de traitement sont longs et une grosse partie du travail que l'on a fourni a été d'optimiser le code pour que l'apprentissage se déroule en un temps acceptable.
On pourrait appliquer un processus correctif (du type algorithme de Viterbi) à l'issue de l'apprentissage pour améliorer encore nos résultats en remédiant le problème d'ordre sans fonction auxiliaire.