Rapport de Projet
Extraction de Relations

Matthieu RÉ et Yaohui WANG

January 31, 2017

Abstract

Nous présentons ici le projet que nous proposons dans le cadre de notre UE de TAL de l'ENSIIE. Nous y implémentons et y testons la solution proposée dans l'article de Mm. Zeng et ses collaborateurs, *Relation Classification via Convolutional Deep Neural Network*[1]

Mots-clefs Extraction de relations, Traitement Automatique de la Langue

1 Introduction

Lors d'un travail passé, nous avons traité des problématiques d'extraction de relations : principalement la détection et la classification de relation dans un document. Nous avons également dressé un état de l'art du domaine qui tend de plus en plus à privilégier l'utilisation de réseaux de neurones pour résoudre le problème.

En effet, des techniques plus anciennes consistaient en la récolte d'un maximum d'information autour d'une phrase et des mots par lesquels on souhaite identifier une relation. Ces informations sont issues d'un long travail de pré-analyse, utilisant de nombreux outils de TAL (POS-tagging, recherche d'hyperonymes, etc.), et qui nécessitent de nombreuses ressources tant matérielles que temporelles.

L'utilisation de réseaux de neurones, qui constitue l'état de l'art actuel dans le domaine, tend à s'émanciper de ces coûteux calculs de caractéristiques, pour qu'ils soient remplacés par un travail d'apprentissage via des réseaux de neurones profonds et convolutifs. Dans ce papier et dans le projet que nous présentons ici, nous nous intéressons plus précisément à la solution apportée par l'article *Relation Classification via Convolutionnal Deep Neural Network*[1].

La présentation de notre projet se fera en trois parties : après avoir discuté de la solution et plus précisément du réseau de neurones proposé par l'article, nous parlerons de l'implémentation que nous avons utilisé pour reproduire les résultats de celui-ci et mettre en place notre application, avant d'en présenter les résultats et de conclure.

2 Présentation du CDNN et de ses justifications

Pour classifier les relations de chaque phrase, nous devons construire des *features* représentant ces phrases à partir des données brutes. Il y a deux types de *features* : les *lexical features* et les *sentence features*. Le processus peut se décomposer en 3 étapes, que nous allons expliquer.

2.1 Vectorisation des mots

Les réseaux de neurones travaillant avec des nombres, et aussi il nous est nécessaire de transformer les mots et les phrases en vecteurs conservant ou apportant l'information que ces phrases détiennent. Pour faire cela, l'article utilise *Word2Vec*[1]. On fixe la dimension de ces vecteurs de représentation à 50 ; on pourrait penser que cette taille arbitraire peut influencer sur les résultats, mais c'est la taille proposée par l'article.

2.2 *Lexical features*

Les *features* lexicales ont 5 composantes :

1. Le vecteur correspondant au premier token
2. Le vecteur correspondant au second token
3. Les vecteurs des mots voisins (gauche et droite) du token 1
4. Les vecteurs des mots voisins (gauche et droite) du token 2
5. Des hyperonymes des tokens, issus de WordNet

Une fois ces 5 composantes obtenues, on les concatène pour former un unique vecteur et on utilise ce vecteur pour représenter la phrase entière. Sachant que nous avons 8000 phrases dans les données d'entraînement, la dimension de ces *features* lexicales atteint 8000 * 300.

2.3 *Sentence features*

Les *sentence features* sont plus difficiles à appréhender que les *lexical features*, étant donné que les auteurs de l'article utilisent dors et déjà un réseau de neurones profond convolutionnel pour les obtenir automatiquement. Le procédé est lui décomposé en 3 parties.

Obtention de la matrice de la phrase Tout d'abord nous utilisons une fenêtre sur chaque éléments de la phrase pour capter le contexte voisin de ces mots. On déplace cette fenêtre du début à la fin de la phrase ; la matrice obtenue représente ainsi la phrase, et chaque colonne représente un mot. On fixe la fenêtre à une largeur de 3.

Prenons la phrase suivante comme exemple :

There is an elephant in the car.

Figure 1: Un exemple plutôt utopique

Le vecteur représentant le premier mot "There" est alors [*DEBUT*, There, is]. Ici, le token de début et de fin de la phrase est le même, et est différent des autres. On les fixe à 0.

Une fois ceci fait on calcule la distance relative entre chaque mot. Les mots marqués sont *elephant* et *car*, la distance relative au premier mot "There" est donc $d_{1,There} = 0 - 3 = -3$ et $d_{2,There} = 0 - 6 = -6$. $[d_{1,There}, d_{2,There}]$ sera concaténé au vecteur du mot "There", et ainsi de suite pour chaque mot.

[1] https://www.tensorflow.org/tutorials/word2vec/

Pour que les matrices aient la même taille pour toutes les phrases, on fixe cette taille à la longueur de la plus grande des phrases. Cela posera problème dans notre implémentation si l'utilisateur souhaite trouver une phrase de longueur supérieure à cette phrase, qui contient 97 tokens. Les vecteurs ne correspondant à cette complétion pour atteindre une matrice de taille 97 sont remplis de token *PADDING*, de valeur 0.

Construction du CDNN Intéressons-nous maintenant à la structure du réseau de neurones :

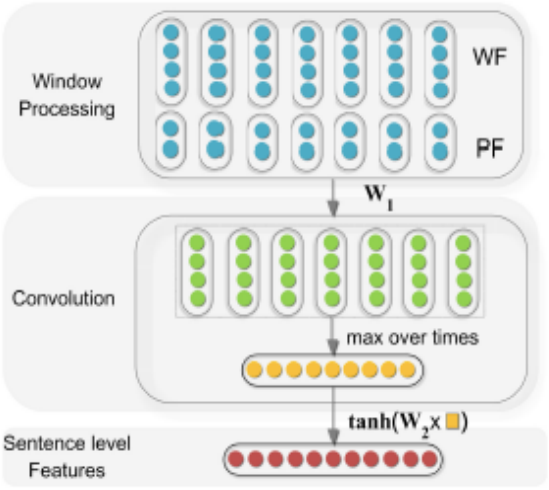

Figure 2: Architecture du CDNN avant calcul des *sentence level features*

Voici la structure du CDNN. WF represente le vecteur de chaque mot dans la phrase, et PF représente les features des distances relatives des mots. On peut voir sur la figure que l'on utilise deux couches cachées pour générer les *sentence level features*. L'entrée en est la concaténation des WF et des PF. La sortie est calculée par le réseau de neurones et reste plutôt abstraite.

1. Première couche cachée L'entrée de la première couche cachée est formée par l'ensemble des matrices de phrases. La fonction d'activation est un fonction linéaire. D'où l'équation :

$$F_1 = W_1 * X$$

Après cela, on utilise un opérateur de *pooling* pour récupérer les valeurs maximales de chaque ligne, formant ainsi en sortie un vecteur.

2. Seconde couche cachée L'entrée de cette couche est la sortie de la première. Sa fonction d'activation est tanh. L'équation la représentant est :

$$F_2 = \tanh(W_2 * X)$$

3. Couche de sortie La sortie de cette seconde couche cachée donne une *feature* vectorielle que le réseau génère de lui-même ; et celle-ci est concaténée aux *lexical feature* de chaque mot. Cette dernière couche a un softmax pour fonction d'activation, le but étant de faire ressortir la classe de relation la plus probable. La dernière équation est donc :

$$F_3 = \text{softmax}(W_3 * X)$$

Intéressons-nous maintenant au vif du sujet, c'est à dire l'implémentation de tout ceci pour notre application.

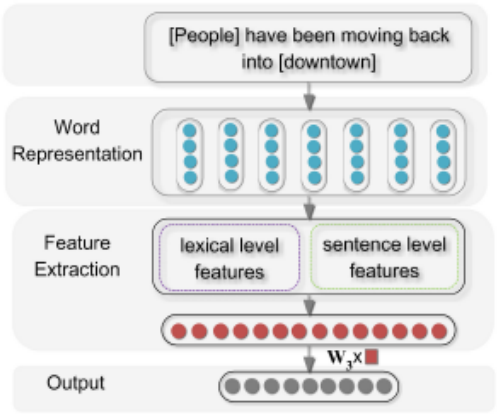

Figure 3: Architecture globale du CDNN

3 Présentation de l'implémentation

Notre principale motivation à choisir cette méthode vient du fait qu'elle n'utilise presque pas de ressources extérieures pour construire les caractéristiques des phrases. Cette économie de ressource a un contre-coup sur la complexité du réseaux de neurones, bien sûr.

Il nous a par ailleurs été particulièrement difficile de reproduire ce dernier, faute tout d'abord de connaître les outils appropriés (comme Tensorflow) (nous poussant ainsi à créer notre réseau de neurones "à la main" de prime abord), et ensuite de savoir maîtriser les outils tels que Tensorflow et Keras, nous nécessitant un long temps d'adaptation et d'apprentissage.

3.1 Ressources extérieures

Données d'entraînement et de test Nous avons longuement cherché une base de données que nous pourrions utiliser afin d'expérimenter les résultats de notre réseau de neurones. Au contraire de certaines bases de références telles que celle de Stanford ou de ACE, une des seules base de données de relations complètement disponibles sur internet est celle issue de la SemEval 2010, Task8[2]. Nous avons ainsi pu la récupérer et l'utiliser.

Ces données sont composées de plus de 10000 exemples composées de phrases en anglais, où deux noms (balisés par <e1> et <e2>) sont reliés par l'une des 19 relations définies sur le dataset. Parmi elles, les 9 écrites dans la liste ci-après, leur relation symétrique, et une catégorie 'autres'.

- **Cause-Effet**(e1,e2) : relation de cause à effet entre deux mots (ex : CE(*exposure*, *cancer*))
- **Outil-Agent**(e1,e2) : agent utilisant un outil, un instrument (ex : IA(*phone*, *operator*))
- **Produit-Fabricant**(e1,e2) : produit conçu par un fabricant (ex : PP(*factory*, *suits*))
- **Contenant-Contenu**(e1,e2) : contenant contenant un contenu (oui) (ex : CC(*glass*, *water*))
- **Entité-Origine**(e1,e2) : entité provenant d'une source, géographique ou physique (ex : EO(*letter*,*foreigh*))
- **Entité-Destination**(e1,e2) : entité se dirigeant vers une destination (ex : ED(*boy*,*bed*))
- **Composant-Catégorie**(e1,e2) : composant d'une catégorie plus globale (ex : CW(*kitchen*,*house*))
- **Membre-Collection**(e1,e2) : membre d'un groupe d'objet de même sorte (ex : MC(*tree*,*forest*))
- **Message-Sujet**(e1,e2) : message dont le thème est <e2> (ex : MT(*lecture*,*semantics*))

[2]http://www.aclweb.org/anthology/S10-1006

Tous ces exemples dépendant évidemment du contexte des phrases dont ils sont issus. Près de 8000 de ces exemples serviront de données d'entraînement, et les 2000 restantes constitueront l'ensemble de test.

Word embedding Dans cette implémentation, nous avons besoin d'une représentation vectorielle des mots et des phrases ; nous utilisons celle de Omer Levy et Yoav Goldberg[3][2], disponible gratuitement[4].

3.2 Création du train et du test

Les données du dataset sont sous une forme bien plus complexe que ce dont nous avons besoin. Il y a en effet des commentaires pour chaque donnée dont nous nous passerons bien, ainsi que des données inutilisable. Le premier script `CreateTrainTest.py` nous permet de créer les fichiers épurés qui nous serviront ensuite pour le modèle. Ils sont sous la forme suivante :

<Relation> <Index de e1 dans p> <Index de e2 dans p> <Phrase p>

Où la phrase p est écrite avec tous mots et ponctuations espacés. Par exemple :

Message-Topic(e1,e2) 3 6 The most common audits were about waste and recycling .

3.3 Preprocessing

Une fois ces deux documents `train.txt` et `text.txt` créés, on cherche à créer les *embeddings* nécessaires à l'entraînement de notre modèle. Pour cela :

- On récupère tous les mots contenus dans le `train.txt` et le `text.txt` .
- On en fait le compte, et on mémorise pour chaque mot les poids issus de l'*embedding*, dans le but d'utiliser ces représentations plus tard dans l'entraînement du réseau.
- On transforme chacune des phrases et des deux mots issus des `train` et `test` pour créer des vecteurs de caractéristiques, de longueur égale à la plus longue phrase d'exemple (ici : 97 tokens), où chaque composante du vecteur est le numéro associé au token (pour le vecteur de représentation de la phrase), et qui prend en compte les relations sémantiques entre mots graĉe au calcul des distances expliqué dans la première partie.
- On sauvegarde ces deux caractéristiques (embedding et représentation des tokens) pour les ensembles d'entraînement et de test dans les fichiers `pkl/embeddings.pkl.gz` et `pkl/sem-relations.pkl.gz` .

On se lance ensuite dans la construction du réseau de neurones proprement dit !

3.4 Entraînement du réseau de neurones

Nous avons énormément **bataillé** à élaborer le réseau de neurones dans sa forme finale. C'est après de longs efforts et de longues recherches que nous avons finalement pu trouver un code qui nous a aidé à implémenter notre réseau de neurones à l'aide de Keras[5], outil que nous ne connaissions pas encore, et nous remercions ces heureux bienfaiteurs !

S'il y a beaucoup à dire sur ce que nous avons essayé de faire avant d'utiliser Keras (apprentissage "manuel" des matrices W, avec forward et backward, malgré l'obstacle de la concaténation de plusieurs features dans l'apprentissage de ces matrices W), c'est moins le cas désormais. L'outil est assez explicite et on retrouve pour chaque *layer* les fonctions d'activations, *inputs* et *outputs*, et quelques configurations supplémentaires pour les couches convolutionnelles et de concaténation des *sentence* et *lexical level* features. Egalement une couche de *dropout* pour éviter le surapprentissage, pour l'instant configurée à 25% mais, comme pour les autres paramètres, que nous pourrions optimiser plus efficacement avec de grosses puissances de calcul !

[3]https://levyomer.files.wordpress.com/2014/04/dependency-based-word-embeddings-acl-2014.pdf
[4]ici : https://levyomer.wordpress.com/2014/04/25/dependency-based-word-embeddings/
[5]https://github.com/UKPLab/deeplearning4nlp-tutorial

Nous reviendrons plus tard sur les performances de notre réseau de neurones.

3.5 Tâche principale

Notre objectif est de pouvoir appliquer ce système sur la donnée d'une phrase et de deux tokens dont on veut trouver une relation.

Ainsi, le script `main.py` est composé d'une partie qui gère les arguments de la commande, d'une partie qui pré-*process* la phrase et les deux mots, et d'une partie qui applique le CNN et rend la relation obtenue.

La première partie récupère la phrase et la position des deux mots dans la commande. Plus d'informations sont disponibles dans l'aide de l'application. Un mode *verbose* est également disponible, pour afficher les messages de Keras, et les 3 features correspondant à la phrase, et aux positions des mots.

La partie de processing utilise des mécaniques similaires au script `prepocessing.py` pour des raisons évidentes de similarité des tâches à effectuer ! On peut cependant noter que le vocabulaire sur lequel ces features sont calculées dans le script initial, se base sur les mots contenus dans `files/train.txt` et `files/test.txt` . Un ajustement est donc nécessaire.

Finalement, Keras propose une API assez similaire à celle de scikit-learn, il est donc ainsi d'obtenir la prédiction finale et de l'afficher.

4 Résultats

Pour récupérer quelques métriques de l'apprentissage de notre réseau de neurones, nous avons utilisé Tensorboard. Comme nous découvrons également cette technologie (qui d'ailleurs est toute récente), nous n'avons pu transformer de façon exploitables certains de nos graphes. Cependant, nous avons pu récupérer les données nécessaires à la création des graphes suivants (Figure).

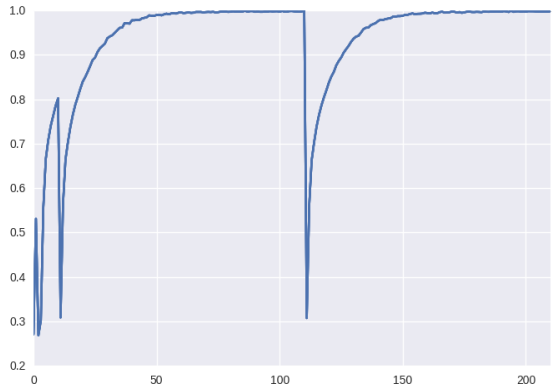

Figure 4: Accuracy

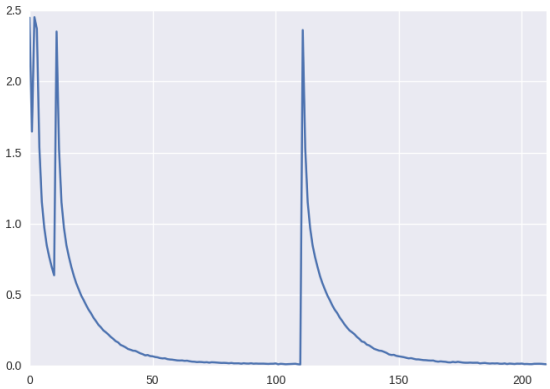

Figure 5: Loss

Figure 6: Évolution de la précision et de la loss au fil de l'apprentissage

On peut voir sur nos figures que notre réseau apprend plutôt rapidement. Le pic peut être dû à une opération de *dropout* (consistant à élaguer une partie des paramètres des matrices W) mais dont l'apprentissage s'en remet plutôt bien. La précision tendant vers 1, on voit que l'on peut améliorer le score total de notre CDNN en évitant plus le surapprentissage, par exemple en augmentant le taux de *dropout*.

La précision de notre réseau s'élève à 0.7847, et sa mesure F1 sans considérer la relation 'Autres' à 0.7603. C'est un joli nombre mais inférieur à ce que l'article annonçait. Pour l'améliorer, il nous serait probablement nécessaire d'optimiser les hyper-paramètres que nous avons utilisé (dropout, taille des couches cachées, etc.). Les résultats restent cependant satisfaisant, si tant est que l'on convienne que nous n'avons ici utilisé qu'une seule ressource extérieure (d'embedding des mots, pour prendre en compte leur similarité) en dehors des données d'apprentissage.

Le temps d'apprentissage a également été plutôt court (nous avons mis 1h pour les apprentissages de notre réseau, mais un ordinateur compatible avec GPU pourrait mettre un temps bien inférieur, sans parler d'un serveur de calcul).

5 Conclusion

Finalement ! Après de longues tribulations, nous avons pu résoudre le problème que nous nous étions posés au départ. Nous ne nous attendions pas au début du projet à avoir tant de difficulté à aboutir à quelque chose. Mais nous y sommes finalement parvenus !

La précision de notre solution n'est pas celle proposée par MM.Zeng et ses collègues. Elle est cependant améliorable et nous pourrions y tendre en optimisant mieux les paramètres de notre réseau.

L'application fait cependant le travail que nous désirions, et sa précision est presque équivalente en l'état à celle d'un SVM utilisant de nombreuses méthodes de TAL, gourmandes et coûteuses.

Nous avons particulièrement apprécié travailler avec ces *features* apprises "toutes seules" par le réseaux de neurones : si nous n'avons pour l'instant que peu de contrôle sur elles, il n'est pas compliqué de prendre conscience de leur puissance et de l'amélioration des performances que l'on pourrait observer en arrivant un peu mieux à les appréhender.

Et nous arrivons au meilleur moment pour suivre les avancés de la recherche dans ce domaine !

References

[1] Siwei Lai Guangyou Zhou Jun Zhao Daojian Zeng, Kang Liu. Relation classification via convolutional deep neural network. *Proceedings of COLING, Aug 2014, Vancouver, Canada*, COLING 2014:2335–2344, 2014.

[2] Yoav Goldberg Omer Levy. Dependency-based word embeddings. *Computer Science Department, Bar-Ilan University, Ramat-Gan, Israel*, 2014.