TAL

Transition Based Parsing – Arc Eager avec Keras

Blouin Baptiste – Bousquet Jérémie

Abstract : ce rapport présente une implémentation de l'algorithme Arc Eager [AE] en Keras, pour différentes variantes dans les définitions des features (*part of speech* ou parties du discours, morphologie, lemmes, …). Des vecteurs de mots pré-entraînés sont utilisés pour les entrées du réseau de neurone correspondant aux lemmes. Un réseau de neurone de type PMC (perceptron multicouche) est spécifiquement entraîné pour chaque jeu de features, et pour quatre langues (français, anglais, japonais, néerlandais). Des résultats calculés selon la métrique LAS sont proposés pour les jeux de tests définis pour chaque langue. Les données UD (universal dependencies) sont utilisées.

Le système a également été entraîné et testé en ajoutant les couples de mots (et non les lemmes) en entrée. D'autres possibilités sont finalement discutées.

Méthode

Arc-eager

Pour mettre en place notre parser nous avons utilisé arc-eager. Ce parser nous sera utile afin de créer nos exemples d'apprentissage, mais aussi pour créer notre fichier conllu de test.

Une configuration pour notre parser ressemble à c = (B , S , A) où B représente le buffer, S la stack , et A l'ensemble des arcs. Pour passer d'une configuration à une autre il faut appliquer une des quatre transitions possibles, citées ci-dessous, sur le dernier élément du stack et le premier élément du buffer.

- *LEFT_ARC* : Ajoute à A un arcs $w_b \rightarrow w_s$ avec le label l_s , où wb est le premier mot du buffer, w_s le dernier mot du stack et l_s le label du dernier mot du stack. Puis on retire w_s du stack.
- *RIGHT_ARC* : Ajoute à A un arcs $w_s \rightarrow w_b$ avec le label l_b , et retire w_b du buffer et l'ajoute au stack.
- *SHIFT* : Retire w_b du buffer et l'ajoute au stack.
- *REDUCE* : Retire w_s du stack.

Afin de déterminer quelle transition doit être effectuée pour passer d'une configuration à une autre nous avons utilisé deux oracles, un pour l'apprentissage et un pour le test. Ces deux oracles seront décrits par la suite.

Pour rajouter le ROOT nous avons pris comme configuration initiale S = [ROOT] , B=[w_1, … , w_n] et A vide , où w_i représente un le mot (structuré en fonction des besoins des features) à la position i dans la phrase.

Lors de la partie d'apprentissage, nous avons pris comme heuristique de générer un *SHIFT* lorsque la stack est vide et que le premier mot du buffer est ROOT, afin d'être cohérent avec notre configuration initiale. La configuration finale de notre parser est telle que le buffer et la stack sont vides. Lors de la partie d'apprentissage, lorsque le buffer est vide nous avons pris comme heuristique de générer un *REDUCE* pour chaque mot restant dans la stack.

Afin de générer nos données d'apprentissages, pour chaque configuration nous utilisons w_s et w_b reformatés par rapport aux features et à la transition résultante de l'oracle.

Feature 1	**Feature 2**	**Feature 3**
S.0.POS	S.0.POS	S.0.POS
B.0.POS	S.0.LEMMA	S.0.LEMMA
DIST	S.0.MORPHO	S.0.MORPHO
	S.-1.POS	S.-1.POS
	B.0.POS	B.0.POS
	B.0.LEMMA	B.0.LEMMA
	B.0.MORPHO	B.0.MORPHO
	B.-1.POS	B.-2.POS
	B.1.POS	B.-1.POS
	DIST	B.1.POS
		DIST

Pour générer nos données dans le cas des deux heuristiques, nous avons rajouté un nouveau mot inexistant dans notre vocabulaire afin de remplacer les valeurs inexistantes dans ces deux cas.

Du fait que nous ne traitons pas le problème des phrases non projectives, nous avons rajouté une valeur booléenne à la fin de nos features afin de différencier les données provenant d'une phrase projective ou non.

Lors de la partie de test, notre parser est utilisé pour écrire dans le fichier de test les arcs créés grâce à celui-ci.

Oracle pour l'entrainement

Durant la partie d'apprentissage nous avons utilisé comme oracle le fichier .conllu . Pour rappel, l'oracle permet de prédire une transition en fonction d'une configuration. Ayant une vision en continue sur les « HEAD » du fichier, il est alors possible prédire :

- *RIGHT* si le HEAD de w_b (noté HEAD(w_b) par la suite) est égal à position de w_s dans la phrase.
- *LEFT* si HEAD(w_s) est égal à la position de w_b dans la phrase.
- *REDUCE* si tous les mots ayant la position de w_s dans la phrase comme HEAD ont déjà étaient traités.
- *SHIFT* dans tous les autres cas.

Durant la partie de test nous avons utilisé comme oracle notre réseau entraîné pour prédire une transition en fonction d'une configuration.

Oracle en inférence

Un réseau de neurone de type PMC est utilisé pour prédire une relation, étant donnée une configuration. Cet « oracle » est utilisé en inférence lors de l'analyse des phrases des jeux de données de test. La phase d'apprentissage se déroule sur des jeux de données d'apprentissage et de validation, générés par Arc Eager avec un oracle ayant la connaissance des relations attendues.

Couche d'entrée

Les éléments d'une configuration dépendent du jeu de features choisi (f1, f2 ou f3), et donc le réseau construit est adapté à ce jeu de features.

Deux types de représentations différents sont utilisés pour encoder ces features pour le réseau (toutes les entrées étant discrètes dans notre cas) :

1. Lorsque le nombre de valeurs possibles pour une feature est raisonnable (ensemble $\{0,\ldots,n-1\}$, elle est encodée sous forme de vecteur de dimension n ayant toutes ses composantes à zéro sauf la composante i. Cet index i correspond à l'index de la valeur dans un vocabulaire construit au préalable à partir des valeurs possibles. Ce format est « one-hot encoding ».

Les types de features suivants sont représentés sous cette forme : POS, morphologie, distance, et les relations à prédire (LABEL).

2. Lorsque cet ensemble (vocabulaire) est trop grand, on utilise des plongements ou embeddings. Chaque symbole est ramené à un vecteur de dimension $d \ll n$. Nous utilisons des embeddings déjà appris, pour des vocabulaires de mots conséquents, produits par facebookresearch [FBE].

Les plongements sont utilisés pour la représentation des mots (qu'ils soient lemmes ou formes).

Concrètement, suivant le jeu de features :

- toutes les features de type « one-hot » sont concaténées, et passées à une entrée du réseau de taille adéquate

- une entrée spécifique de taille 1 est définie pour chaque feature utilisant les plongements

Pour f1, nous avons donc seulement une entrée avec un vecteur one-hot des features concaténées, pour f2 nous avons 3 entrées, 2 pour les lemmes, et 1 pour le vecteur de features, et de même pour f3.

Embeddings

Chaque feature représentée dans le réseau par un embedding, est associée à une couche spécifique de type « Embedding ». L'entier fourni en entrée est considéré comme un index et converti en « one-hot » , et l'embedding du mot correspondant est activé. Les poids de cette couche sont donc une matrice de dimensions $|vocabulaire| x d$.

Couches denses

Le perceptron multicouche en lui-même est représenté par deux couches de neurones totalement connectées. Suivant les features, les différentes couches d'entrée de ces couches denses (c'est à dire, les entrées pour les features discrètes, et les sorties des couches de plongement) sont concaténées au préalable.

Sortie

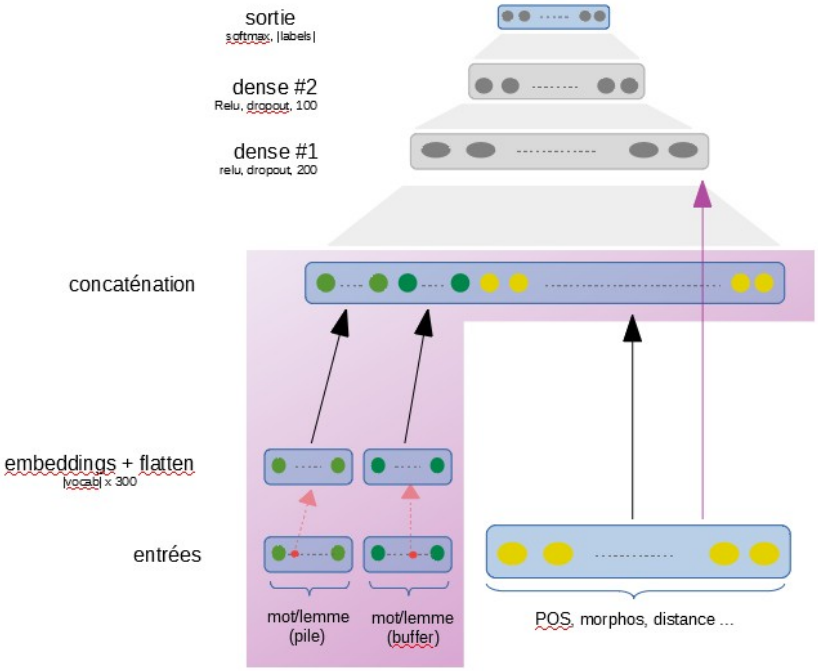
La dimension de sortie de la dernière couche dense est également celle d'un vecteur parcimonieux et interprétée comme un encodage « one-hot », basé sur le vocabulaire de relations possibles. Les valeurs possibles pour chaque neurone sont donc entre 0 et 1, et on entraîne le réseau à indiquer (par le neurone ayant l'activation la plus proche de 1) le type de relation à prédire.

Implémentation

Réseau de neurones

Le réseau de neurones est un classifieur PMC construit grâce à la librairie Keras[K], combinée au backend Tensorflow[TF].

Le réseau est schématisé ci-contre. La partie violette représente une architecture lorsque des couches d'embeddings sont nécessaires (cas de f2 et f3). Cette partie est absente pour f1, et les différentes features concaténées sont directement connectées à la première couche dense.



Les trapézoïdes gris clair représentent les couches entièrement connectées.

Un modèle enregistre toutes les informations nécessaires pour construire le réseau de neurones approprié (langue, plongements ou pas, dropout, dimensions des couches, vocabulaires…) de façon dynamique. Le programme principal boucle sur les langues, puis les jeux de feature, construit une architecture de réseau de neurone adaptée, puis appelle l'algorithme arc eager en se passant en paramètre. Arc Eager est alors exécuté mais en utilisant le réseau de neurones comme oracle (fonction predict du modèle Keras). Une autre fonction adaptée à l'architecture du réseau, est chargée de l'encodage d'une donnée d'entrée de test.

Pour optimiser les performances, certaines informations produites par des traitements longs sont stockées en cache sur disque (les plongements alignés sur les vocabulaires, le réseau lui-même au format de Keras, etc). La maintenance d'un statut par langue / jeu de features, permet de n'exécuter que les combinaisons qui ne l'ont pas encore été.

Plusieurs étapes sont nécessaires pour préparer les données d'entrée du réseau :

- les jeux de données sont générés à partir des fichiers .conllu (train, dev, test) en appliquant Arc Eager avec l'oracle basé sur le fichier, ainsi qu'un vocabulaire pour chaque élément (POS, LEMME, FORM, …), exceptée la distance (un entier entre 0 et 7)
- on obtient des matrices de dimension n x p (n étant le nombre d'arcs, p comptant une dimension par mot, lemme, et feature, défini dans le jeu de features, plus une colonne indiquant la projectivité ou non de la phrase)
- pour l'entrainement
    - les phrases non projectives sont retirées des jeux de données train et dev – nous avons considéré que les données partielles et fausses produites par Arc Eager pour les phrases non projectives, risquaient de dégrader les performances de l'oracle d'inférence.
    - les vocabulaires (train et dev) sont consolidés, seul le vocabulaire « train » étant utilisé en apprentissage (chaque élément de dev manquant dans train est ajouté)
    - si nécessaire un « mot inconnu » (« <UNK> ») est ajouté dans les vocabulaires
    - les indices des données de test sont également alignés sur le vocabulaire train (mais sans ajout de mot), donc alignés sur les vocabulaires connus du réseau de neurones

Note : a ce stade, ces jeux de données dev et test sont utilisés uniquement pour évaluer les capacités du réseau de neurone, avant son utilisation en inférence – jamais en phase d'inférence.

    - Les plongements sont chargés en mémoire, et une matrice de plongement est générée :

- pour chaque mot du vocabulaire concerné, si ce mot existe dans les embeddings, le vecteur est ajouté au même indice dans la matrice
- si ce mot n'existe pas dans les embeddings, à cet indice est ajouté un vecteur aléatoirement généré dans la matrice
- les mots des embeddings, absents du vocabulaire, sont ajoutés au vocabulaire, jusqu'à la limite définie si elle existe $|vocab|_{max}$
- On attribue le vecteur de « l'apax » (le mot le moins fréquent) du catalogue d'embeddings, à notre « mot inconnu ». Avec les plongements fasttext, c'est le dernier mot du fichier (par exemple « contradicat » pour le français)

Note sur les plongements / embeddings : des embeddings pré-entrainés de dimension 300 et comprenant parfois plus de 1 000 000 de mots ont été utilisés. Si l'on considère un vocabulaire de 100 000 mots, cela représente 30 000 000 de paramètres par couche d'Embedding. Etant donné la quantité limitée de mémoire sur nos machines, nous avons pris en compte une limite pour la taille de vocabulaire d'une entrée de couche de plongement. Lorsque les indices du vocabulaire, sont alignés sur les mots des embeddings pré-entrainés, les mots non présents à l'origine dans le vocabulaire mais présents dans les embeddings sont ajoutés au vocabulaire, mais seulement dans la limite fixée. On ne retire en revanche jamais de mot du vocabulaire d'origine qui correspond au jeu de données d'apprentissage et validation.

Expérimentation

Les réseaux de neurones ont été spécifiés comme ci-dessous pour l'expérimentation :

- Couches d'entrée définies suivant la langue (les vocabulaires sont de dimensions variables) et le jeu de feature.
- Embeddings pré-entraînés pour les lemmes (f2 et f3), de dimension 300, limités à $|vocab|_{max}=50000$, 2 couches d'embedding (lemme de la pile, lemme du buffer). Note : une couche Keras « Flatten » est nécessaire à la suite des couches d'embeddings, ces couches dans Keras gèrent une séquence de mots (pour réseaux récurrents), dans notre cas on élimine la dimension supplémentaire introduite avant de transmettre les activations aux couches denses.
- Une couche Keras de type « Concatenate » permet de rassembler les entrées en une seule pour les couches denses (les colonnes sont simplement mises bout à bout)
- Les deux couches totalement connectées de 200 et 100 neurones respectivement, suivies chacune d'une fonction d'activation relu et de dropout, (0.15) et une dernière couche dense dont la taille dépend de la taille du vocabulaire de relations à prédire, avec une fonction d'activation softmax. En prédiction, l'argmax du Y prédit donne l'indice de la relation prédite dans le vocabulaire.
- Optimiseur Adam, avec une fonction de perte de type categorical_crossentropy, et une métrique d'accuracy pour surveiller l'apprentissage. Chaque réseau de neurone a appris durant 5 epochs. Le jeu de données test est passé à la fonction evaluate du modèle keras, simplement pour se faire une idée des capacités du réseau en généralisation et adapter au besoin. Ensuite les phrases de test sont analysées grâce à l'algorithme Arc Eager et à ce nouvel oracle.
- Le jeu de données dev est utilisé dans keras sous forme de validation_data lors de l'appel à la fonction fit()
- Le code source de l'expérience est disponible [SRC]

Résultats

La métrique utilisée est LAS.

Langue / Features	Allemand	Anglais	Français	Japonais
1	55.30	55.26	62.50	68.36
2	67.09	68.04	72.93	75.93
3	66.70	67.68	74.12	78.77

Taux de phrase projective dans le test

Allemand	Anglais	Français	Japonais
840 / 876 ≈ **96%**	800 / 914 ≈ **87.5%**	398 / 416 ≈ **96%**	538 / 551 ≈ **98%**

Commentaires

On constate que l'ajout de features calculées supplémentaires (et donc d'informations supplémentaires sur le contexte de la relation à prédire) permet d'obtenir de meilleurs résultats en inférence.

On ne voit pas vraiment de corrélation entre le taux de phrases projectives et les performances – sans doute la métrique LAS n'est pas la plus adaptée pour mettre en évidence cette corrélation.

La limitation du nombre de mots maximal des embeddings pour des raisons d'utilisation mémoire, a sans doute pu avoir une incidence négative sur les résultats, qu'il serait intéressant d'analyser pour aller plus loin.

Pour aller plus loin

Dans l'optique d'expérimenter, et de pouvoir éventuellement utiliser le système et son oracle pour analyser des phrases non analysées au préalable, nous avons évaluer pour le français le jeu de features suivant :

Feature 1+form

```
S.0.FORM
B.0.FORM
S.0.POS
B.0.POS
DIST
```

De la même façon que pour les lemmes, les mots ont été intégrés au réseau sous forme d'embeddings pré-entraînés. Le résultat est le suivant sur le jeu de test UD pour le français (métrique LAS) :

Langue / Features	Français
1 + formes	67.28

On obtient un résultat légèrement amélioré comparé au f1 sans les mots, en revanche il reste plus optimal d'ajouter de multiples features pour obtenir de meilleurs résultats. Mais le réel intérêt ici est que pour aller plus

loin, il serait possible de fabriquer un moteur d'inférence travaillant sur une phrase ne provenant pas d'un fichier .conllu (une phrase quelconque, entrée par un utilisateur, par exemple). Dans ce cas on pourrait estimer la partie de discours (POS) pour les mots en se basant sur le suffixe du mot. Du fait de l'utilisation de valeurs estimées pour POS, il est probable que l'ajout des mots eux-même en entrée permette de meilleurs résultats dans ce contexte.

Bibliographie

AE: Yoav Goldberg,Joakim Nivre, A Dynamic Oracle for Arc-Eager Dependency Parsing, , http://www.aclweb.org/anthology/C12-1059
FBE: Facebook Research, FaceBook Reserach - fasttext embeddings, , https://github.com/facebookresearch/fastText/blob/master/docs/crawl-vectors.md
K: , Keras, , https://keras.io/
SRC: https://github.com/jbousque/tal
TF: , Tensorflow, , https://www.tensorflow.org/