Étiqueteur POS avec réseaux récurrents

PSTAL - TP 1 - Carlos Ramisch

1 Introduction

Tous les travaux pratiques de PSTAL consisteront à *développer* et *évaluer* un système de **prédiction de structures linguistiques** à partir du texte. Ces structures peuvent être très diverses, allant du rôle grammatical de chaque mot (p.ex. nom, verbe, adjectif) aux entités nommées contenues dans les phrases (*Carlos*$_{[PERS]}$ *accompagne les étudiant.e.s de Centrale Marseille*$_{[ORG]}$ *à Luminy*$_{[LOC]}$). Chaque TP se concentrera sur un seul type de structure à prédire, par exemple, pour ce premier TP, nous nous concentrons sur les parties du discours.

2 L'étiquetage en parties du discours

Dans cette tâche, chaque mot reçoit une étiquette de **partie du discours** (POS, de l'anglais *part of speech*). Ces étiquettes indiquent le rôle grammatical du mot dans la phrase, par exemple : le mot *été* est un nom (étiquette `NOUN`) dans *l'été arrive.* [1] Il s'agit d'une tâche d'**étiquetage de séquences** car, pour chaque phrase $w_1, w_2 \dots w_n$ en entrée, il faut prédire une séquence d'étiquettes $t_1, t_2 \dots t_n$ de même longueur :

	w_1	w_2	w_3	w_4	w_5	w_6
$x =$	*L'*	*été*	*arrive*	*à*	*Marseille*	.
	⇓	⇓	⇓	⇓	⇓	⇓
$y =$	DET	NOUN	VERB	ADP	PROPN	PUNCT
	t_1	t_2	t_3	t_4	t_5	t_6

Les défis de cette tâche sont : (1) certains mots sont ambigus (p.ex. *été* est un verbe dans *j'ai été informée*) ; (2) le contexte, c.-à-d. les mots d'avant et d'après, joue un rôle crucial pour désambiguïser les mots ayant plusieurs POS possibles ; et (3) le système doit être capable de prédire des POS pour des mots non observés dans le corpus d'entraînement, que l'on appelle souvent "OOV" pour *out-of-vocabulary* (p.ex. la plupart des noms propres). Nous nous concentrerons sur les deux premiers défis, le dernier est proposé en extension (Sec. 10).

3 Corpus Sequoia et format CoNLL-U+

Pour ce TP et les suivants, nous allons utiliser le corpus *Sequoia*, contenant 3 099 phrases en français annotées avec plusieurs niveaux d'informations. [2] Ouvrez le fichier fourni `sequoia-ud.parseme.frsemcor.simple.small` pour lire les explications ci-dessous avec le corpus sous les yeux. Les fichiers contiennent du texte encodé en UTF-8, vous pouvez les ouvrir avec n'importe quel éditeur de textes. Voici un exemple extrait du corpus :

```
# sent_id = annodis.er_00007
# text = Amélioration de la sécurité
1 Amélioration amélioration NOUN _ Gender=Fem|Number=Sing 0 root _ _ * Act *
2 de de ADP _ _ 4 case _ _ * * *
3 la le DET _ Definite=Def|Gender=Fem|Number=Sing|PronType=Art 4 det _ _ * * *
4 sécurité sécurité NOUN _ Gender=Fem|Number=Sing 1 nmod _ _ * State *
```

Le corpus est segmenté en phrases et chaque phrase est segmentée en mots. Chaque ligne du fichier contient un mot, avec des lignes blanches pour séparer les phrases entre elles. Les phrases se lisent donc à la verticale ! Chaque mot contient 13 colonnes séparées par des tabulations (TAB ou `\t`), avec un type d'information linguistique par colonne. Les noms des colonnes sont donnés dans la toute première ligne du fichier. Vous pouvez utiliser un tableur (p.ex. Excel, Libreoffice Calc) pour visualiser les colonnes alignées, comme ci-dessous :

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	ID	FORM	LEMMA	UPOS	XPOS	FEATS	HEAD	DEPREL	DEPS	MISC	PARSEME:MWE	FRSEMCOR:NOUN	PARSEME:NE
2	1	Amélioration	amélioration	NOUN	_	Gender=Fem|Number=Sing	0	root	_	_	*	Act	*
3	2	de	de	ADP	_	_	4	case	_	_	*	*	*
4	3	la	le	DET	_	Definite=Def|Gender=Fem|N	4	det	_	_	*	*	*
5	4	sécurité	sécurité	NOUN	_	Gender=Fem|Number=Sing	1	nmod	_	_	*	State	*

1. Nous utiliserons les étiquettes Universal Dependencies (UD), décrites ici : https://universaldependencies.org/u/pos/.
2. Les détails se trouvent dans le fichier `sequoia/README.md` fourni, voir Sec. 4.

Les meta-informations (lignes commençant par un `#`) peuvent être ignorées. L'entrée de votre système sera toujours la deuxième colonne : FORM. Il s'agit du mot tel qu'il apparaît dans le corpus, sans aucune annotation. [3] Le système doit prédire les autres colonnes ; p.ex le TP d'aujourd'hui permettra de faire un système qui prédit la colonne numéro 4 (UPOS). Vous trouverez la description du format CoNLL-U (10 premières colonnes) sur le site de Universal Dependencies. [4] Les colonnes 11, 12 et 13 seront détaillées ultérieurement.

Nous vous conseillons la bibliothèque https://pypi.org/project/conllu/. Son installation et fonctionnement sont faciles à prendre en main. La fonction `parse_incr` est particulièrement utile pour récupérer des `TokenList`, où chaque élément est un dictionnaire représentant le mot avec ses annotations. Voici un exemple :

```
from conllu import parse_incr
for sent in parse_incr(open("sequoia-ud.parseme.frsemcor.simple.small", encoding='UTF-8')):
  print(" ".join(tok["upos"] for tok in sent))
```

4 Données et code fournis

Des données et du code sont fournis dans https://gitlab.lis-lab.fr/carlos.ramisch/pstal-etu. Il s'agit d'un dépôt `git` qu'il convient d'actualiser souvent (`git pull`) pour avoir la dernière version des fichiers. Dans le dossier `lib/` vous trouverez le script d'évaluation `accuracy.py`. Son fonctionnement est expliqué dans `./accuracy.py --help`. Le module `conllulib.py` regroupe des fonctions qui peuvent vous être utiles.

Le dossier `sequoia/` contient 5 fichiers *CoNLL-U+* nommés `sequoia-ud.parseme.frsemcor.simple`. Les fichiers suffixés `.train`, `.dev` et `.test` contiennent les données d'entraînement, de validation/développement, et de test. Rappel : vos expériences doivent être réalisées sur le corpus `.dev`, vous ne reporterez les résultats sur le `.test` qu'à la toute fin. Le fichier `.full` contient l'union des trois fichiers précédents : il ne doit jamais être utilisé tel quel. Finalement, le fichier `.small` est un extrait du corpus `.dev` qui peut être pratique pour développer et déboguer votre système sans perdre trop de temps à chaque exécution.

5 Préparation des données

Le modèle requiert des tenseurs en entrée/sortie. Chaque élément du tenseur est un indice représentant un mot/étiquette dans le vocabulaire. Il faut donc convertir les entrées (mots) et sorties (étiquettes) en suites d'entiers, et construire en parallèle le vocabulaire des mots V_w et des étiquettes V_t. N'oubliez pas de garder un indice à part pour le padding (p.ex. `PAD_ID=0`) dans V_w et V_t, et un indice pour les OOV dans V_w (p.ex. `UNK_ID=1`). L'encodage est nécessaire pour `.train` et pour `.dev`, et les mots du `.dev` absents de V_w sont encodés `UNK_ID`. La structure `defaultdict` est pratique pour encoder vocabulaire, comme illustré dans `conllulib`.

Des *batches* regrouperont des phrases de longueur variable. Il faut donc (a) tronquer les phrases dépassant la longueur maximale L et (b) ajouter du padding pour les phrases plus courtes que L. Les phrases tronquées, paddées, et transformées en `LongTensor` sont transmises à la fonction `Util.dataloader` dans `conllulib`.

6 Le modèle RNN d'étiquetage

Le modèle d'étiquetage est une classe héritant de `nn.Module` contenant les éléments suivants :

— `nn.Embeddings` : matrice de dimensions $|V_w| \times d_e$ prenant en entrée des entiers représentant les mots, et donnant en sortie un vecteur de dimension d_e par mot. Contrairement à `keras`, vous ne transformerez pas les entiers en *one-hot*, cela est automatique. Le *broadcast* est aussi automatique : si votre entrée est un `LongTensor` de dimension $B \times L$ (où B est la taille du *batch* et L la longueur des phrases), la sortie sera de dimension $B \times L \times d_e$, avec un vecteur d_e-dimensionnel par entrée. Il est important d'indiquer l'indice du padding (`padding_idx=PAD_ID`) pour que ce vecteur reste nul pendant l'apprentissage.

— `nn.GRU` : pour chaque embedding de dimension d_e, la couche récurrente génère un vecteur cachée de dimension d_h. Contrairement à `keras`, la longueur L de la suite n'est pas un paramètre de la couche : comme le graphe de calcul est dynamique, la couche admet des séquences de longueur variable. Nous devons indiquer à cette couche que `batch_first=True`. De plus, nous ne voulons pas de biais (`bias=False`) car celui-ci interférerait sur le padding dans un réseau bidirectionnel.

— `nn.Linear` : la couche de décision est une matrice de dimension $d_h \times |V_t|$. L'activation softmax est intégrée dans la *loss* et n'est pas appelée explicitement. N'oubliez pas d'ajouter un peu de *dropout*.

3. Vous remarquerez que la tokénisation a séparé les contractions, p.ex. `aux`→`à les`, `du`→`de le`, `duquel`→`de lequel` ...
4. https://universaldependencies.org/format

7 Entraînement du modèle

Si dans `keras` il suffit d'appeler `model.fit`, dans `torch` il faut écrire sa propre fonction `fit(model, data)`. Heureusement, elle est similaire d'un projet à l'autre ; on peut copier puis adapter. La fonction `fit` doit :

1. Initialiser la fonction de perte (`nn.CrossEntropyLoss`) et l'optimiseur (p.ex. `optim.Adam`)
2. Dans la boucle extérieure, répéter un certain nombre d'`epochs`
3. Dans la boucle intérieure, parcourir les *batches* (x, y) du `DataLoader` et, pour chaque *batch* :
   (a) Mettre à zéro tous les gradients de tous les paramètres du modèle (`zero_grad`)
   (b) Passer x dans le modèle pour obtenir la prédiction $\hat{y}$, puis calculer la `loss` en fonction de y. [5]
   (c) Rétro-propager les gradients (`backward`), et mettre les paramètres à jour (`optimizer.step`)

Ces étapes constituent le coeur de l'apprentissage. Mais pour s'assurer que tout se passe bien, il convient d'afficher, à la fin de chaque `epoch`, la loss cumulée sur le `.train`, et la loss et sur le `.dev`. Écrivez une fonction `perf` qui parcourt les *batches* (x, y) du `DataLoader` `dev`, prédit les scores des étiquettes $\hat{y}$, accumule les valeurs de la cross-entropie, comme pour l'entraînement, puis les affiche.

L'*accuracy* (exactitude) sur le `.dev` peut aussi être calculée dans `perf`. Pour cela, transformez les logits $\hat{y}$ en indices d'étiquettes prédites $\hat{t}$ (*argmax*). Mais attention : nous voulons ignorer le padding, il faut donc le masquer. Le code `mask = (y != PAD_ID)` permet d'obtenir un tenseur de masquage, appliqué ensuite à la comparaison de $\hat{t}$ avec y, par exemple : `(t_hat == y) * mask`. Pensez à mettre le modèle en mode `model.train` au début des `epoch`, et de le mettre en mode `model.eval` lors du calcul des performances sur le `.dev`.

Une fois le modèle entraîné, il ne faut surtout pas le jeter (comme on fait souvent dans les *jupyter notebook*) ! Sauvegardez le modèle `model.state_dict` dans un fichier `.pt`. Il faut également sauvegarder les vocabulaires V_w et V_t, sans quoi le modèle sera inutilisable. Notez que `torch.save` accepte des dictionnaires quelconques, vous pouvez tout sauvegarder ensemble : les paramètres du modèle et les vocabulaires. [6] Profitez pour sauvegarder aussi les *hyper-paramètres* nécessaires à la prédiction dans ce même fichier (d_e, d_h, `PAD_ID`...).

8 Prédiction

L'étiqueteur prend en entrée un corpus `.dev` et un modèle entraîné. Le modèle est d'abord instancié (Sec. 6), puis initialisé avec `load_state_dict`. [7] Chaque phrase du `.dev` doit être transformée en entiers à l'aide de V_w, passée dans le modèle pour obtenir $\hat{y}$, puis $\hat{t}$ (*argmax* de $\hat{y}$). La fonction `rev_vocab` dans `conllulib` permet de convertir les indices en étiquettes POS, à placer dans le champ `upos` des mots avant d'imprimer la phrase avec `serialize`. La prédiction se fait par phrase et non pas par *batch* : il n'y a donc pas de troncage ni de padding.

9 Travail à effectuer

Vous devez écrire deux scripts différents : `train_postag.py` pour l'entraînement du modèle (Sec. 7), et `predict_postag.py` pour la prédiction des POS (Sec. 8). Le code du modèle RNN (Sec. 6) doit être partagé par les deux scripts : vous pouvez le mettre dans un module/fichier à part et l'importer, par exemple. L'évaluation des prédictions sera effectuée par le script fourni `lib/accuracy.py`.

10 Extensions

Gestion des OOV Dans ce TP, les OOV sont tous représentés par `UNK_ID`. On peut faire mieux avec (a) des embeddings statiques pré-entraînés non apprenables, (b) des embeddings de préfixes et suffixes de mots à la `fasttext`, ou (c) des modèles sous-lexicaux tels que les RNN ou les CNN sur les caractères à la Ma & Hovy (2016). Implémentez et évaluez (option `--train` de `accuracy`) une de ces solutions pour représenter les OOV.

Performance vs. nb. de paramètres Le nombre total de paramètres (`Util.count_params`) dépend essentiellement de d_e et d_h. On peut ajouter des paramètres : utiliser des LSTM à la place des GRU, rendre le RNN bidirectionnel, empiler des couches récurrentes (`num_layers` et `bidirectional`). Étudiez l'évolution (a) du temps d'entraînement et (b) de l'*accuracy* sur le `.dev` en fonction de la capacité et de l'architecture du RNN. Pour des résultats plus stables, implémentez le *early stopping* et fixez le *seed* aléatoire (`Util.init_seed`).

5. Pensez à transposer les 2 dernières dimensions de $\hat{y}$ pour la rendre compatible avec le format attendu par la loss.
6. Avant la sauvegarde, transformez V_w et V_t en `dict` Python standard, sinon vous aurez une erreur.
7. Spécifiez `weights_only=False` pour éviter le warning de sécurité.