Analyse morphologique avec apprentissage multi-tâches

PSTAL - TP 3 - Carlos Ramisch

L'objectif de ce TP est de développer et évaluer un système d'analyse morphologique. Par exemple, le système doit prédire que le mot *étudiantes* est décliné au féminin (`Gender=Fem`) et pluriel (`Number=Plur`). Le système sera basé sur un classifieur neuronal avec `pytorch` qui prendra en entrée une suite de mots et prédira les étiquettes de chacun de ces mots. En français, ce sont souvent les suffixes qui aident à prédire les traits morphologiques. Ainsi, nous lirons les phrases caractère à caractère, et non pas mot à mot. De plus, il faudra prédire non pas une seule étiquette mais plusieurs paires clé-valeur (voire aucune). Notre système sera donc plus complexe que celui développé pour la prédiction des POS au TP1.

1 L'analyse morphologique

Les traits morphologiques (dorénavant TM) sont des paires `clé=valeur` où la clé indique le type de trait (genre, nombre, temps,...) Ils sont présents dans la colonne `feats` du corpus (6ème colonne), avec chaque paire séparée de la suivante par une barre verticale. P.ex. `Gender=Masc|Number=Sing` représente deux traits, le genre (masculin) et le nombre (singulier). Dans l'exemple ci-dessous, les TM sont affichées sur plusieurs lignes :

	La	*gare*	*routière*	*attend*	*toujours*	*ses*	*illuminations*
POS=	DET	NOUN	ADJ	VERB	ADV	DET	NOUN
TM=	Definite=Def Gender=Fem Number=Sing PronType=Art	Gender=Fem Number=Sing	Gender=Fem Number=Sing	Mood=Ind Number=Sing Person=3 Tense=Pres VerbForm=Fin	_	Number=Plur Poss=Yes	Gender=Fem Number=Plur

Les TM varient selon les langues. Par exemple, en français le TM `Case` est absent, alors qu'il est fréquent dans les langues à cas comme l'allemand, le polonais, et le hongrois. Plus important pour nous, les TM varient au sein d'une même langue selon la POS du mot : en français, les noms prennent les TM `Number` et `Gender`, alors que les verbes ont aussi `Number` mais, au lieu de `Gender`, prennent les TM `Mood`, `Person`, `Tense` et `VerbForm`.

Chaque clé (type de TM) possède un ensemble fini de valeurs possibles.[1] Notez que les TM sont des **ensembles** : une clé ne peut pas apparaître deux fois pour un même mot. L'ensemble vide est dénoté par l'underscore (p.ex. *toujours* ci-dessus). La bibliothèque `conllu` renvoie `None` pour les ensembles de TM vides, et un dictionnaire Python sinon. Par convention, les TM sont ordonnés par ordre alphabétique : cela rend immédiate la comparaison de deux ensembles de TM par simple comparaison de chaîne de caractères.

2 Préparation des données

Entrées : Les TM sont liés à la forme des mots, notamment à la présence de certains suffixes en français (*-s* → pluriel, *-ons* → 1ère personne du pluriel,...). Nous n'extrairons pas les suffixes explicitement : ils seront représentés implicitement à l'aide d'un **réseau récurrent sur les caractères.**[2] Il faut donc transformer les phrases en séquences de caractères, pour ensuite les encoder avec un vocabulaire spécifique.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
chars=	<pad>	L	a	<esp>	g	a	r	e	<esp>	r	o	u	t	i	è	r	e	<esp>	a	t	t	e	n	d	...
in_enc=	0	3	4	2	5	4	6	7	2	6	8	9	10	11	12	6	7	2	4	10	10	7	13	14	...
ends=	2	7	16	23	32	36	50																		

1. La documentation complète des traits se trouve sur le site Universal Dependencies pour les traits morphologiques
2. Cette idée est inspiré du modèle de langage FLAIR (Akbik et al. 2018)

Dans cet exemple, les caractères sont placés dans un vecteur `chars`, où un caractère special `<esp>` indique les frontières entre les mots. Notez que la première position est remplie par un caractère spécial `<pad>` : cela sera expliqué plus tard. Le vecteur `in_enc` correspond aux mêmes caractères encodés sous la forme d'entiers : un vocabulaire des caractères doit garder les correspondances, p.ex. {`<pad>`→0, `<unk>`→1, `<esp>`→2, `L`→3, `a`→4, ...}. Pensez à réserver un indice spécial pour le caractère OOV `<unk>` (rencontré en inférence mais absent du `.train`). C'est le même encodage que lors du TP1, mais cette fois-ci sur les caractères au lieu des mots.

Le troisième vecteur `ends` nous permettra d'obtenir une représentation par mot pour la prédiction (voir § 3). Il indique les positions des fins de mots. Par exemple, le dernier caractère du premier mot *La* se trouve à la position 2 de `in_enc`. Le deuxième mot *gare* se termine à la position 7, etc. Les deux vecteurs `in_enc` et `ends` sont les entrées du classifieur (paramètres de *forward*).

Sorties : Comme pour le TP1, les sorties sont encodées à l'aide d'un vocabulaire. L'indice 0 est réservé au padding, comme d'habitude. Pour commencer, nous nous concentrerons sur un seul type de TM : `Number`. Les mots ayant ce trait (p.ex. noms, verbes) ont sa valeur encodée sous la forme d'indice. Les mots n'ayant pas ce trait (p.ex. adverbes) se voient attribuer une valeur spéciale `<N/A>`, comme illustré ci-dessous :

	La	*gare*	*routière*	*attend*	*toujours*	*ses*	*illuminations*
`Number` =	`Sing`	`Sing`	`Sing`	`Sing`	`<N/A>`	`Plur`	`Plur`
`out_enc` =	2	2	2	2	1	3	3

Padding : Pour créer des tenseurs, il faut homogénéiser la longueur des phrases. Comme dans le TP1, coupez les phrases trop longues et ajoutez du padding pour celles trop courtes. Nous avons deux types d'entrées : établissez un nombre maximal de caractères (p.ex. `max_c`=200) pour `in_enc`, et un nombre maximal de mots (p.ex. `max_w`=20) pour `ends` et `out_enc`. Les phrases dépassant ces seuils seront coupées (crop). Pour le vecteur `ends`, le padding est constitué de zéros, comme pour les deux autres. Nottez que `Util.dataloader` peut prendre plusieurs tenseurs en entrée (`in_enc` et `ends`), il n'est pas nécessaire d'avoir des entrées et sorties uniques.

3 Le modèle RNN d'étiquetage

Le classifieur est très proche de celui du TP1 : une classe `nn.Module` avec les fonctions `forward` et `__init__` :

- `nn.Embeddings` : matrice de dimensions $|V_c| \times d_c$ prenant en entrée des entiers représentant les caractères (un batch dans `in_enc`), et donnant en sortie un vecteur de dimension d_c par caractère.[3] Si l'entrée est de dimension $B \times$ `max_c` (où B est la taille du *batch*), la sortie est de dimension $B \times$ `max_c` $\times d_c$.
- `nn.GRU` : pour chaque embedding de dimension d_c, la couche récurrente génère un vecteur caché de dimension d_h. Indiquer `batch_first=True`, `bias=False`, et `bidirectional=False`. La sortie `rnn_out` encode des informations contextuelles sur le voisinage des caractères (implicitement, des suffixes).
- `gather` : La dimension 1 de `rnn_out` ($B \times$ `max_c` $\times d_h$) est incompatible avec celle des sorties $B \times$ `max_w`. Le vecteur `ends` nous aidera à sélectionner uniquement les états cachés du RNN correspondant à des fins de mots. Pour cela, utilisez la fonction `rnn_out.gather` pour sélectionner les positions de `ends` dans la dimension 1 et obtenir un tenseur de dimension ($B \times$ `max_w` $\times d_h$).[4, 5]
- `nn.Linear` : la couche de décision est de dimension $d_h \times |V_t|$, où V_t est le vocabulaire du TM `Number` dans cette première version. N'oubliez pas le *dropout* et le softmax implicite (intégré à la *loss*).

4 Entraînement du modèle

Dans cette première version, les fonctions `fit` et `perf` peuvent être directement copiées du TP1. La seule particularité est que le `DataLoader` renvoie ici 2 entrées (`in_enc` et `ends`) et 1 sortie. Comme pour le TP1, utilisez la `nn.CrossEntropyLoss` comme loss et `optim.Adam` comme optimiseur. Parcourrez les `epochs`, puis les batches. Pour chaque batch, mettez les gradients à zéro, passez les 2 entrées dans le modèle, calculez la loss, rétro-propagez et actualisez les paramètres. Copiez aussi la fonction `perf` qui donne la valeur de la loss et de l'*accuracy* (masquer le padding) sur le `.dev`. Sauvegardez le modèle, les hyper-paramètres, et les vocabulaires des caractères et des TM. Référez vous à la section "Entraînement du modèle" du TP1 pour plus de détails.

3. La valeur de d_c peut être petite, p.ex. entre 50 et 100, car le vocabulaire des caractères est plus petit que celui des mots.
4. En pratique, ajoutez une dimension à `ends` et propagez l'indice sur celle-ci : `ends.unsqueeze(2).expand(-1,-1,`d_h`)`
5. Comme `in_enc` commence par `<pad>`, `gather` utilisera l'état du RNN correspondant à `<pad>` pour padder le résultat.

5 Prédiction

Comme pour le TP1, chargez un modèle sauvegardé avec `load_state_dict`, puis lisez le corpus `.dev` phrase par phrase.[6] Encodez la phrase sous la forme d'une paire de tenseurs `in_enc` et `ends`, passez-les dans le modèle pour obtenir $\hat{y}$, puis prédire $\hat{t}$ comme l'*argmax* de $\hat{y}$. La fonction `rev_vocab` dans `conllulib.py` permet d'obtenir les valeurs des TM à partir des indices. Pensez à effacer tous les autres TM présents, et à mettre `feats=None` quand l'étiquette prédite est `<N/A>` avant d'imprimer les phrases avec `serialize`.

6 Prédiction de plusieurs TM : modèle multi-tâches

Une fois votre classifieur pour le TM `Number` prêt et testé, il est temps d'effectuer la prédiction pour plusieurs traits en même temps. Nous adopterons la technique d'**apprentissage joint**, aussi appelé apprentissage **multi-tâches**. Votre classifieur aura autant de couches de décision `nn.Linear` que de types de traits (une matrice pour `Gender`, une pour `Tense`, etc.), mais les embeddings de caractères et le RNN seront uniques, partagés par toutes les couches de décision, comme dans le modèle illustré en Figure 1.

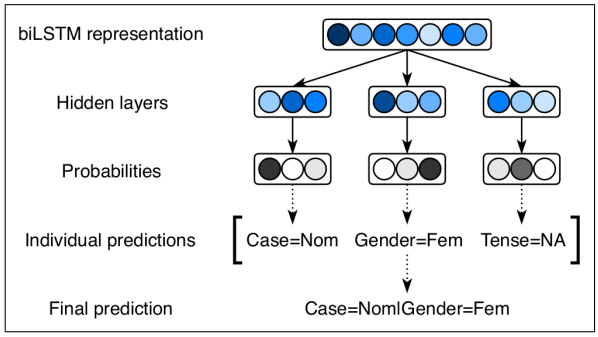

FIGURE 1 – Rybak & Wróblewska (2018)

Pour préparer les données, il faut un vocabulaire par type de TM (14 au total, pour *sequoia*). Vous pouvez stocker ces vocabulaires dans un dictionnaire indexé par le nom du TM. Les sorties seront encodées et transformées en tenseurs (crop/pad) comme pour `Number`. Le `DataLoader` aura désormais 2 entrées et 14 sorties ! Lors de l'initialisation du classifieur, stockez les 14 couches linéaires dans un `torch.nn.ParameterDict` pour les inclure dans les paramètres. Ces couches sont de dimension $d_h \times |V_t|$ avec où V_t est le vocabulaire du trait t.Lors du `forward`, ces 14 couches prennent le même `rnn_out` en entrée, et renvoient un tuple contenant 14 tenseurs. La loss sera la somme (`torch.sum`) des losses sur les 14 sorties, et le gradient les prendra toutes en même temps.

7 Évaluation

Le script d'évaluation `accuracy.py` calculera l'*accuracy* pour les ensembles complets de traits prédits correctement. Cependant, cette évaluation est très stricte, car un seul trait mal prédit suffit pour que l'ensemble soit considéré faux. Ainsi, en indiquant `--tagcolumn feats` vous aurez aussi les métriques de précision `P`, rappel `R` et F-score par TM, ainsi que la micro- et macro-moyenne. Utilisez ces scores pour évaluer vos systèmes.

8 Travail à effectuer

Vous devez créer deux programmes : `train_morph.py` pour l'entraînement du modèle (Sections 4 et 6), et `predict_morph.py` pour la prédiction des TM (Section 5). Comme suggérez ci-dessus, faites d'abord le système complet pour le TM `Number` avant de l'étendre aux autres TM avec un modèle multi-tâches.

9 Extensions

Convolution 1D sur les caractères Le modèle de Rybak & Wróblewska (2018) propose un CNN plutôt qu'un RNN sur les caractères. Créez des représentations des mots avec des CNN sur les caractères, suivi de max-pooling sur la longueur du mot. L'entrée du classifieur sera un seul tenseur à 3 dimensions : $B \times$ `max_w` $\times$ `avg_w` où `avg_w` est la longueur moyenne des mots, avec du padding/crop dans les 2 dernières dimensions. Faites varier le nombre de filtres, la taille des kernels, la dilatation. Comparez le nombre de paramètres, le temps d'apprentissage et la performance (P/R/F) par rapport au modèle RNN du sujet.

Embeddings de mots, préfixes et suffixes Utilisez comme point de départ un modèle fondé sur les mots, comme celui du TP1. Ajoutez à ce modèle des embeddings représentant des préfixes et suffixes de taille $1, 2 \ldots k$ que vous combinez aux embeddings de mots, à la manière de fasttext. Faites une étude d'ablation, c'est-à-dire, comparez le modèle uniquement avec les mots, avec mots + suffixes de longueur 1, de longueur 2, etc. Pour chaque modèle, analysez le nombre de paramètres, le temps d'entraînement et la performance (P/R/F).

6. Il n'y a pas de traitement par batch, donc pas de longueur maximale (crop) ni de padding.