AMAL - TP 6
Réseaux récurrents : Séquence à séquence (seq2seq)

Nicolas Baskiotis - Benjamin Piwowarski - Laure Soulier

2020-2021

Introduction (brève, cf cours)

Dans ce troisième et dernier TP sur les RNNs, nous allons étudier deux tâches de type "séquence vers séquence" (Seq2Seq) :

— Étiquetage de chaque élément de la séquence (le nombre d'éléments en entrée et en sortie sont les mêmes)

— Génération d'une séquence à partir d'un état latent. Cet état latent représente la donnée en entrée, et dépend de la tâche : représentation de la phrase (traduction/question réponse), représentation d'une image (légende d'image), etc.

Par rapport aux TPs précédents, trois nouveautés :

1. Utilisation des RNN définis par torch qui prennent en entrée une `PackedSequence` et renvoient en sortie des structures similaires (il est possible de les utiliser également avec des batchs standards).
2. Utilisation d'un encodage de taille variable (*wordpiece*)
3. Utilisation de *curriculum learning* pour l'apprentissage

1 Étiquetage

Dans cet exercice, nous allons nous intéresser à la tâche d'analyse syntaxique (Part-Of-Speech) qui consiste à associer à chaque mot une nature/catégorie grammaticale. Nous utiliserons le jeu de données GSD (cf squelette du code pour le télécharger). Chaque exemple est une phrase, déjà segmentée en tokens, pour lesquels nous nous intéresserons au mot brut (`form`) et au tag (`upostag`).

Le code suivant vous est donné :

— la classe `Vocabulary` qui permet de gérer un vocabulaire (correspondance entre un token et son index) ; elle sera utilisée à la fois pour indexer les mots et pour indexer les tags ;

— la classe `TagDataset` qui permet de construire un dataset où chaque item est un couple (token, tag), les tokens étant les mots d'une phrase et les `tags` les catégories associées ;
— la fonction de collage `collate_fn` du `DataLoader` qui utilise la fonction `pad_sequence` pour former des séquences de même taille (cf ci-dessous).

Le padding tel que vu au TP précédent peut être fait avec la fonction `pad_sequence` de `Pytorch`. Le paramètre `ignore_index` des fonctions de coût permet d'indiquer l'index du caractère de padding afin de ne pas prendre en compte le paddin dans le calcul du coût (comme le masque de la séance dernière). Cependant, compléter ainsi les séquences a un coût computationnel qui peut être très grand car beaucoup d'opérations inutiles vont être effectuées (lors du feeding du réseau et également lors du calcul de la loss). `Pytorch` permet de limiter cet effet de bord en utilisant des séquences dites "packées" : les séquences sont mises à plat et toutes allignées sur la même dimension, la longueur réelle des séquences est stockée conjointement, et seules les opérations nécessaires seront effectuées. La fonction `pack_sequence` appliquée sur une liste de tenseurs permet d'obtenir une `PackedSequence`, un tuple qui contient d'un côté le batch et de l'autre la longueur de chaque élément du batch. Pour obtenir ce tenseur aplatit, on peut utiliser également la fonction `pack_padded_sequence` sur un batch déjà paddé. L'opération inverse est `pad_packed_sequence` qui permet d'obtenir un tenseur paddé à partir d'un tenseur aplatit.

Question 1

Implémentez un modèle `seq2seq` pour le tagging et la boucle d'apprentissage en utilisant le module LSTM de `Pytorch` et le padding de séquences (le packing est à faire en bonus).
Pour tenir compte du problème des mots OOV pendant l'apprentissage, on peut remplacer au hasard dans les exemples d'apprentissage des mots par des mots OOV (en utilisant un token spécial - `[OOV]`). Implémentez et comparer les résultats.
Pour visualiser les résultats, afficher pour une phrase donnée en entrée la séquence de tags associés.

2 Traduction

Pour la tâche de traduction, nous allons utiliser deux RNNs :
— un encodeur qui est en charge de produire un état caché après avoir lu la séquence à traduire
— un décodeur, qui à partir de l'état caché, va engendré la phrase traduite.
En plus du token *EOS (End of Sequence)*, vous aurez besoin d'un token spécial *SOS (Start*

of Sequence) qui sera le premier token donné en entrée au décodeur (en plus de l'état caché) à partir duquel la phrase est traduite.

Il y a deux manières d'entraîner le décodeur :

— le mode contraint (ou *teacher forcing*) où la phrase cible est passée au décodeur et à chaque pas de temps c'est un mot de la phrase cible qui est considéré : la génération est guidée et permet de corriger précisement chaque état latent produit ;

— le mode non contraint où la phrase cible n'est pas considérée lors de la génération itérative de la traduction : c'est le mot correspondant à la probabilité maximale du décodage de l'état latent du pas précédent qui est introduit à chaque pas de temps (ou un tirage aléatoire dans cette distribution) ; ce mode consiste à générer comme si vous étiez en inférence puis à corriger une fois toute la phrase engendrée.

Le mode non contraint est plus difficile que le mode contraint : une erreur lors de la prédiction d'un mot va perturber énormément la génération et la backpropagation ne sera pas très efficace sur la suite de la séquence. Cependant, il permet de mieux généraliser et d'éviter de l'apprentissage par coeur. Intuitivement, il faudrait commencer l'entraînement en utilisant le mode contraint pour bien initialiser le décodeur puis basculer au fur et à mesure vers le mode non contraint. Ce genre de procédé s'appelle Curriculum learning.

Question 2

Dans `tp6-traduction.py`, implémentez l'encodeur-décodeur. Utilisez dans les deux cas des GRUs et les architectures suivantes :

— encodeur : un embedding du vocabulaire d'origine puis un GRU

— décodeur : un embedding du vocabulaire de destination suivi d'un fonction ReLU, puis un GRU suivi d'un réseau linéaire pour le décodage de l'état latent (et un softmax pour terminer)

Dans le décodeur, vous aurez besoin d'une méthode `generate(hidden,lenseq=None)` qui à partir d'un état caché `hidden` (et du token SOS en entrée) produit une séquence jusqu'à ce que la longueur `lenseq` soit atteinte ou jusqu'à ce que le token EOS soit engendré.

Implémentez la boucle d'apprentissage en utilisant une stratégie simple de Curriculum learning qui consiste à tirer au hasard uniformément entre le mode contraint et le mode non contraint pour chaque mini-batch [a]. Vous pouvez passer lors de la génération la longueur attendue des phrases cibles dans le cas du mode non contraint.

Entrainez votre modèle en gardant un jeu de test pour vérifier que vous ne sur/sous-apprenez pas.

Utilisez (en test) les méthodes de générations du TP précédent pour visualiser les traductions proposées.

a. Dans l'article original, le choix entre les deux modes se fait selon une probabilité décroissante pour le mode contraint et ce choix est fait à chaque pas de temps, non pas pour le minibatch entier. Vous pouvez en bonus améliorer la stratégie et comparer les résultats.

Le pré-traitement des textes repose sur une étape de segmentation où le texte est découpé en unités linguistiques. Pendant longtemps le niveau choisi était le mot (= chaîne alphanumérique entourée d'espace) ; depuis quelques années, des alternatives ont été (ré)explorées avec les nouveaux modèles neuronaux.

Une des segmentations les plus efficaces à l'heure actuelle est le découpage en n-grammes variables (*subword units*) popularisé par le Byte-Pair Encoding (BPE) en 2016. Ces segmentations ont l'avantage d'avoir un vocabulaire de taille fixe qui couvre au mieux le jeu de données, et permet d'éviter le problème des mots inconnus.

Par exemple, "You shoulda got David Carr of Third Day to do it" sera segmenté en `"_You", "_should", "a", "_got", "_D", "av", "id", "_C", "ar", "r", "_of", "_Th", "ir", "d", "_Day", "_to", "_do", "_it"` où les séquences fréquentes (ex. You, should) sont extraites directement alors que des séquences moins fréquentes (ex. David, Carr) sont segmentées en plusieurs parties.

Vous utilisez la librairie sentencepiece (n'oubliez pas de mettre à jour les modules python si vous ne travaillez pas sur une machine de l'université). Les tokens Unknown <unk>, BOS (<s>) and EOS (</s>) sont prédéfinis, mais vous pouvez en ajouter d'autres avec `user_defined_symbols` :

```
import sentencepiece as spm
spm.SentencePieceTrainer.train(
  input='FILEPATH',
  model_prefix='MODEL_OUTPUT_PATH',
  vocab_size=1000, user_defined_symbols=[]
)
```

Pour segmenter une phrase, vous pourrez utiliser les instructions suivantes :

```
import sentencepiece as spm
s = spm.SentencePieceProcessor(model_file='MODELPATH.model')

# Renvoie les identifiants des tokens
ids = s.encode('New York', out_type=int)

# Renvoie la chaine correspondante
sp.decode(ids)

# Renvoie les tokens
s.encode('New York', out_type=str)
```

Question 3

Créez le modèle de segmentation pour le jeu de données puis changez le `Dataset` pour qu'il utilise une segmentation variable. Une fois la segmentation en place, attaquez-vous au problème de l'apprentissage d'un modèle de traduction. Vous pouvez comparer les résultats avec ou sans segmentation.