# AMAL - TP 5
# Réseaux récurrents
# LTSM, GRU, et autres cellules à mémoire

Nicolas Baskiotis - Benjamin Piwowarski - Laure Soulier

2020-2021

Dans ce TP, nous continuons l'étude des RNNs, et leur application au traitement du langage naturel, en étudiant deux problèmes :

1. Comment traiter et générer des séquences de taille variable ?
2. Comment prendre en compte des dépendances à plus long terme (*Vanishing/exploding* gradients)

Nous reprenons le but de l'exercice 3 du TP 4 (génération de séquences) ainsi que le jeu de données (discours pré-électoraux de Trump). Vous copierez donc le code du TP 3 que vous modifierez petit à petit pour répondre aux différentes questions.

## 1 Génération de séquences de taille variable

Contrairement au TP précédent, nous considérons ici que chaque séquence est une phrase (qui peut se terminer par un point, un point d'interrogation ou un point d'exclamation). Nous ne supposons donc plus que les séquences ont une taille fixe. Il faut dès lors :

— employer un symbole spécial qui marque la fin d'un séquence (EOS). Lors de l'apprentissage, il faut ajouter ce token à chaque séquence pour apprendre à le prédire ;

— padder chaque séquence, i.e. ajouter un caractère nul (BLANK) autant de fois qu'il est nécessaire afin que les séquences d'un même batch aient toutes la même longueur ; ce caractère devra être ignoré lors de l'apprentissage (cf question 2).

**Question 1**

Dans `textloader.py`, implémenter une clase `TextDataset` tel que :

— chaque exemple soit une phrase (utiliser le `"."` comme délimiteur[a]) ; la taille du dataset est le nombre de phrases dans le corpus ;

— la phrase renvoyée soit sous forme d'une séquence d'entiers, chaque entier codant

pour un caractère (utiliser pour cela `string2code`).
Définir ensuite une fonction `collate_fn` qui prépare un tenseur batch (de taille `longueur x taille du batch`) à partir d'une liste d'exemples du `TextDataset` : elle doit ajouter le code du symbole EOS à la fin de chaque exemple et padder les séquences avec le code du caractère nul.
Cette fonction peut être passée en argument d'un DataLoader pour créer les batchs adéquats.
Exécuter `generate.py` pour vérifier que tout fonctionne bien.

*a.* Penser à utiliser la fonction `split` pour découper une chaîne de caractère et la fonction `strip()` pour enlever les espaces si nécessaire

De plus, il faut prendre soin de ne pas inclure le padding lorsqu'on calcule le critère de maximum de vraisemblance : pour cela, la solution la plus courante [1] est d'utiliser un masque binaire (0 lorsque le caractère est nul, 1 sinon) qui est multiplié avec les log-probabilités avant de les additionner (le paramètre `reduce="none"` dans une fonction de coût - en particulier pour la cross entropie - permet d'obtenir le coût pour chaque élément plutôt que la moyenne).

**Question 2**

Créer votre fonction de coût `maskedCrossEntropy(output, target, padcar)` qui permet de calculer ce coût sans prendre en compte les caracètres nuls en fonction de la sortie obtenue `output`, la sortie désirée `target` et le code de caractère de padding `padcar`. Vous ferez attention à n'utiliser aucune boucle pour ce calcul.

La dernière modification est au niveau de la génération.

**Question 3**

Dans `generate.py`, implémenter une fonction de génération de façon à générer des séquences jusqu'à rencontrer le caractère EOS (penser à prévoir une taille maximum tout de même !). Vous pouvez prévoir de faire de la génération aléatoire dans la distribution obtenue ou déterministe en choisissant le caractère le plus probable à chaque pas de temps.

Lors du TP4, vous avez utilisé un encodage one-hot pour chaque caractère, suivi d'un module linéaire. Le module `nn.Embedding` de Torch permet de combiner ces deux étapes

1. Jusqu'à l'apparition des `packedsequence` que vous n'utiliserez pas dans ce TP.

pour éviter la création (non-nécessaire et coûteuse) de vecteurs one-hot.

> **Question 4**
>
> Modifier votre code pour utiliser des *embeddings* directement par l'intermédiaire du module `nn.Embeddings`, sans passer manuellement par une représentation *one-hot*.

## 2 Prise en compte de dépendences lointaines : LSTM et GRU

Les réseaux récurrents (RNN) sont un type d'architectures privilégié pour traiter les séquences en "encodant" de manière itérative une séquence (cf. TP 4), par le biais d'une fonction $f$ qui calcule l'état suivant par une étape d'encodage : $\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$.

Les RNNs ont du mal à capturer les dépendances à long terme. Ceci est dû au fait que le gradient est très instable lorsqu'on remonte à quelques pas de temps; en simplifiant à l'extrême : la contribution de l'entrée $t$ à pour l'étape $t+k$ sera alors de l'ordre de (cf [3])

$$\|\mathbf{W}^k\|^2 = \sum |\lambda_i|^{2k}$$

Dans ce TP, nous allons étudier des variantes des RNNs, et en particulier les Long-Short Term Memories/LSTMs [2] et les Gated Recurrent Units/GRU [1] (voir aussi ce post).

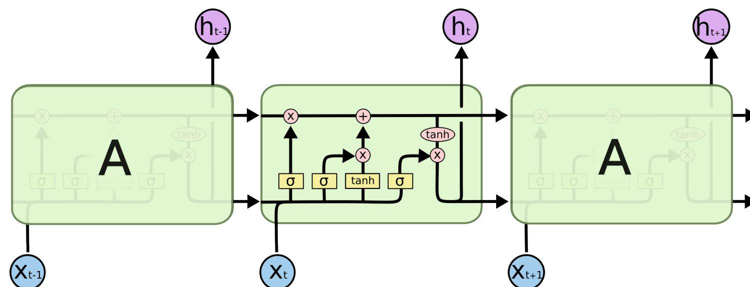


FIGURE 1 – Un LSTM (Cola'h blog)

Les LSTMs sont définis par un état externe $h_t$ (analogue à ceux des RNNs usuels) et un état interne $C_t$ qui représente la mémoire "à long terme" du réseau. À chaque pas de temps, $C_t$ est mis-à-jour en fonction de de l'état interne précédent $h_{t-1}$ et de l'entrée $x_t$ en spécifiant ce qui doit être "oublié" du passé et ce qui doit être "retenu" pour le futur. Le nouveau état externe est calculé à partir du nouveau état interne. Ce mécanisme d'écriture repose sur la notion de *porte* empruntée à la logique, qui permet de masquer une partie du signal qui a peu d'intérêt. Dans le cas des réseaux de neurones, une porte est une fonction continue (et

non discrète comme en logique), souvent une couche linéaire suivie d'une activation sigmoïde (qui produit donc une sortie entre 0 et 1).

L'évolution des états internes et externes est définie par les équations suivantes (Figure 1) qui repose sur trois portes (oubli, entrée, et sortie) :

$$\begin{aligned}
f_t &= \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) && \text{Porte (oubli)} \\
i_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) && \text{Porte (entrée)} \\
C_t &= f_t \otimes C_{t-1} + i_t \otimes \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) && \text{Mise à jour (mém. interne)} \\
o_t &= \sigma\left(W_o\, [h_{t-1}, x_t] + b_o\right) && \text{Porte (sortie)} \\
h_t &= o_t \otimes \tanh\left(C_t\right) && \text{Sortie}
\end{aligned}$$

où $[a, b]$ est une concaténation vectorielle et $\otimes$ est un produit terme à terme (Hadamard).

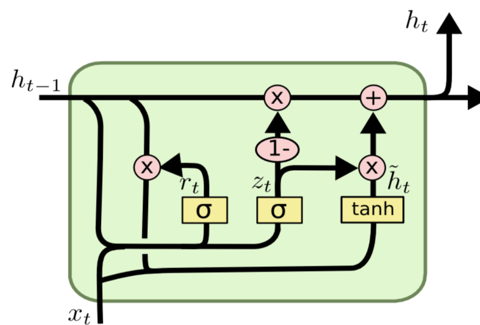


FIGURE 2 – Un GRU (Cola'h blog)

Les GRUs sont des simplifications des LSTMs (il en existe d'autres) – l'état interne et externe est le même, les portes entrée/oubli sont fusionnées (avec $i_t := 1 - f_t$) ; les équations qui définissent un GRU (Figure 2) sont :

$$\begin{aligned}
z_t &= \sigma\left(W_z \cdot [h_{t-1}, x_t]\right) \\
r_t &= \sigma\left(W_r \cdot [h_{t-1}, x_t]\right) \\
h_t &= (1 - z_t) \otimes h_{t-1} + z_t \otimes \tanh\left(W \cdot [r_t \otimes h_{t-1}, x_t]\right)
\end{aligned}$$

**Question 5**

Implémenter un GRU puis un LSTM. Notez que pour les LSTMs, il y a un état interne. Comparez les résultats en vraissemblance (loss) ainsi que qualitativement (séquences générées).

Utilisez tensorboard pour surveiller la magnitude des gradients ainsi que l'évolution des valeurs des différentes portes. Utiliser pour cela `add_histogram`. Si vous avez des problèmes de stabilité et que les magnitudes des gradients sont grandes, utilisez du "gradient clipping".

# 3 Beam-search

Lors de vos tentatives de génération, vous observerez que en prenant à chaque pas de temps l'argmax vous obtiendrez très peu souvent des phrases intelligibles (vous faîtes en fait une approximation gloutonne du maximum de vraisemblance). Dans un premier temps, vous pouvez échantillonner à chaque pas de temps dans la distribution inférée, mais le résultat ne sera pas bien meilleur. La solution usuelle consiste à utiliser un "beam search" pour approximer l'argmax sur toute la séquence engendrée : le beam search consiste à conserver à tout moment $t$ un ensemble de $K$ séquences (et leur log-probabilité associée) ; à l'étape $t+1$, on génère pour chacune des séquences $s$ les $K$ symboles les plus probables étant donnée $s$. Puis on sélectionne les $K$ séquence de taille $t+1$ les plus probables (et on ré-itère).

**Question 6**

Implémenter un beam-search pour la génération de phrase, et comparez de manière qualitative les résultats obtenus.

La qualité des séquences générées peut encore être améliorée en utilisant des techniques d'échantillonage telle que le Nucleus Sampling [4], qui consiste à définir la probabilité de génération en ne considérant que les $K$ caractères les plus probables (où $K$ est un hyperparamètre). Formellement, si $I_K(p, s)$ est un ensemble de $K$ symboles tel que

$$\forall y \in I_K(p,s), y \notin I_K(p,s) \quad p(y|s) \geq p(y'|s)$$

la probabilité

$$p_{\text{nucleus } K}(y|s_t) = \mathbb{1}\left[y \in I_K(p)\right] \frac{p(y|s_t)}{\sum_{y' \in I_K(p)} p(y'|s_t)}$$

**Question 7**

Implémentez le beam search en utilisant le Nucleus sampling.

# Références


[1] Kyunghyun Cho et al. "Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation". In : 2014.

[2] Sepp Hochreiter et Jürgen Schmidhuber. "Long Short-Term Memory". In : 2006.

[3] Sepp Hochreiter et al. "Gradient Flow in Recurrent Nets : The Difficulty of Learning Long-Term Dependencies". In : (2001).

[4] Ari Holtzman et al. “The Curious Case of Neural Text Degeneration”. In : International Conference on Learning Representations. 25 sept. 2019.