

Intelligence Bio-Inspirée
Classifieur de textes à partir d'un RNN (Pytorch)

1 Rendu

Rendu par trinômes obligatoirement :

- (1) Déposer le code dans la colonne dédiée de Tomuss par trinômes
- (2) Déposer votre compte rendu dans la colonne Tomuss dédiée (4 à 8 pages au maximum) :
 - lister les fonctionnalités développées
 - pour votre meilleur hyperparamétrage : courbes d'apprentissage (train + validation), matrice de confusion (test), visualisation(s) de l'encodage des mots
 - analyser vos résultats

2 Préparation des données

2.1 Chargement du dataset

Le dataset (prétraité) disponible sur Moodle est divisé en trois jeux :

- train : utilisé pour l'apprentissage
- validation : utilisé pour mesurer la capacité de généralisation de votre réseau, et choisir l'itération propice à l'arrêt de l'apprentissage
- test : utilisé pour vérifier la généralisation des performances mesurées. C'est la performance du jeu de test qui doit être au final reportée lors d'une évaluation impartiale.

Formatage des jeux d'apprentissages ligne par ligne : <texte de longueur variable>
<séparateur> <sentiment>

Développer une fonction `load_file(file)` en utilisant la fonction native `open(...)`, qui retourne la liste des textes et la liste associée des émotions (en format "texte").

2.2 Préparation des données par encodage one-hot

- Créer une correspondance : mot -> id_mot
Prévoir un "mot" dédié au padding (mot de rembourrage pour avoir des phrases de même taille).
- Créer un encodage one-hot de vos mots de dimension : (vocab_size)
- Le réseau de neurone prend en entrée de façon récurrente un vecteur : (batch_size, vocab_size). Soit par exemple pour un batch_size = 10, sentence_size = 20 : les dix premiers mots, les dix seconds mots, ..., les dix vingtièmes mots
- Le réseau est alimenté par batchs. Préparer une classe torch.utils.data.Dataset qui permettra de générer le codage onehot d'une phrase (fonction __getitem__(self, idx)).

3 RNN

3.1 Architecture du réseau récurrent : *class RNN(nn.Module)*

Vous pouvez utiliser et adapter le code associé à figure 1 :

- Cette architecture n'est pas assez puissante pour un codage one-hot par mot.
Ajouter une couche d'embedding de taille emb_size : self.i2e = nn.Linear(input_size, emb_size), c'est donc la couche d'embedding qui alimente désormais la couche combined
- le code proposé prévoit un batch (première dimension du tensor en entrée) de taille 1 dans la fonction init_hidden(..).
- Tester votre réseau, en l'appelant avec un mot unique (batch_size = 1, sans récurrence)
- Tester avec la récurrence (ajout de chaque mot), puis avec un batch plus grand. Utiliser un Dataloader pour cela.

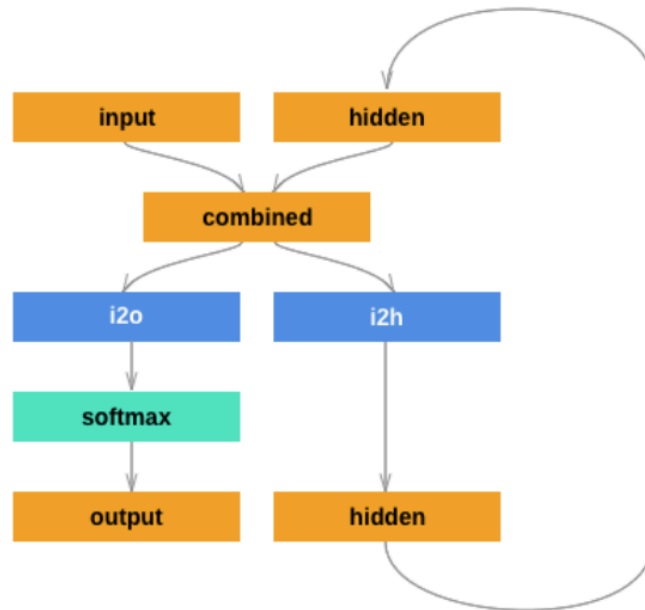


FIGURE 1 – Architecture pour RNN présentée ici : https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial

3.2 Apprentissage du réseau

- *développer votre boucle d'entraînement par epoch/batch*
- analyser les performances, paramétrer votre réseau

Les performances devraient dépasser les 50% d'accuracy (et même plus ...).

4 Optimisations diverses

- *limiter les effets du déséquilibre des classes (loss adaptée)*
- *étudier la distribution des mots et filtrer également les mots rares (TF-IDF)*

5 Approfondissement

Afficher la représentation des mots obtenue par la matrice d'embeddings (PCA ou t-SNE). Analyser les résultats.

Utiliser cette fois le dataset pour réaliser un apprentissage auto-supervisé (contexte -> mot). Afficher et analyser.