ENSAI – Traitement Automatique du Langage

[TP INFO] Développement d'approches neuronales

Ce TP sur 4 séances a pour but de vous familiariser avec la mise en œuvre des techniques d'apprentissage neuronal en traitement automatique des langues. Il est organisé en deux parties, *grosso modo* de 2 séances chacune.

La première partie s'inscrit à la suite du TP 2 sur la classification en polarité et vous guide dans la conception de nouveaux classifieurs fondés sur des réseaux de neurones. Au-delà de la réalisation de nouveaux classifieurs et de la comparaison de différentes architectures (que l'on pourra comparer aussi aux trois développés lors des séances précédentes, *i.e.*, k-nn et word net), ce TP aborde la notion de préparation des données langagières pour l'apprentissage neuronal. La seconde partie s'intéresse à la détection d'entités nommées vue comme une tâche d'étiquetage de séquences : à partir d'un modèle simple donné, vous êtes invités à mobiliser les connaissances du cours et l'expérience acquise dans la première partie du TP pour définir le meilleur modèles possible.

Pour développer des approches neuronales, le TP s'appuie sur le *framework* `tensorflow.keras`[1] et, dans une moindre mesure, sur la librairie Huggingface pour les modèles transformeurs pré-entrainés[2]. Il existe bien entendu d'autres *frameworks*, notamment `pyTorch`, qui diffèrent peu dans leurs principes fondamentaux de `tf.keras` malgré une syntaxe différente.

Dans le framework `tf.keras`, les étapes de la création d'un modèle sont de manière classique les suivantes :

1. préparation des données sous la forme d'un objet `tf.data.Dataset` qui permet facilement les opérations telles que le mélange aléatoire des données, la sélection de certains données, la création de *batch*, ou encore la normalisation de la longueur pour les séquences (*padding*).
2. définition de l'architecture du modèle neuronal en combinant des `layers` préprogrammés dans `tf.keras`, les principaux en NLP étant[3] : `Embedding` pour l'apprentissage des plongements de mots, `Dense` pour une cellule de type perceptron, `LSTM` pour un réseau récurrent de type LSTM, `BiDirectionnal` pour des réseaux récurrents bi-directionnel, `Attention` et `MultiHeadAttention` pour les mécanismes d'attention.
3. définition du modèle (aka *compilation*) en spécifiant les paramètres d'apprentissage comme la fonction de coût, l'algorithme d'optimisation des paramètres, *etc.*
4. estimation des paramètres du modèles à partir de données d'apprentissage et de données d'évaluation
5. utilisation du modèle pour faire des prédictions sur les données de test

Pour plus de détails sur ces différentes étapes et notions, vous pouvez vous référez au tutoriel `https://www.tutorialspoint.com/keras/keras_deep_learning.htm`.

Partie 1. Classification de documents

Dans cette partie, on utilisera `spaCy` comme lors des séances précédentes pour l'analyse des textes, notamment la *tokenisation* et l'étiquetage morpho-syntaxique. On utilisera le résultat de l'analyse par `spaCy` pour mettre en œuvre trois approches : (a) un perceptron multi-couche prenant en entrée une représentation pré-calculée de taille fixe d'un document ; (b) un réseau récurrent de type LSTM pour apprendre la représentation d'un document (y compris *embedding* des *tokens*) ; (c) un modèle

[1] *Cf.* `https://www.tensorflow.org/api_docs/python/tf/keras`
[2] *Cf.* `https://huggingface.co/transformers`
[3] *Cf.* `https://www.tensorflow.org/api_docs/python/tf/keras/layers` pour une liste des `layers` existants

d'auto-attention bi-directionnel pré-entrainé comme modèle de langage masqué (de type BERT) et adapté à la tâche de classification.

Avant de démarrer le TP, chargez les modules dont vous aurez besoin :

```
import numpy as np
import json
import spacy
import tensorflow as tf
import matplotlib.pyplot as plt

# si besoin
!wget http://people.irisa.fr/Guillaume.Gravier/teaching/ENSAI/data/imdb-trn.json
!wget http://people.irisa.fr/Guillaume.Gravier/teaching/ENSAI/data/imdb-tst.json

# si besoin -- la première fois ou en cas de réinitialisation du notebook
!python -m spacy download en_core_web_md
```

À faire :

1. Chargez les données d'apprentissage et de test depuis les fichiers `imdb-*.json` et lancez la *tokenisation* et l'étiquetage morphosyntaxiques avec `spaCy` et le modèle `en_core_web_md`. Nous n'utiliserons que la *tokenisation* et l'étiquetage morpho-syntaxique. Mémorisez le résultat de cette analyse, par exemple en ajoutant une colonne aux données chargées.

Perceptron multicouche avec une représentation pré-calculée

Dans cette approche, on va utiliser `spaCy` pour obtenir une représentation de chaque document sous forme d'un vecteur de taille fixe comme pour l'approche par k-nn dans le TP 2 et un réseau de neurone de type perceptron multicouche pour la classification à partir de la représentation vectorielle. L'objectif de cette première partie est de se familiariser avec la préparation des données et la définition d'un modèle dans `tf.keras`.

Préparation des données

Pour chaque document, nous allons définir un vecteur de type *embedding* moyen des lemmes ou des *tokens* de dimension `dim` et un vecteur de dimension 2 pour encoder la classe du document (*e.g.*, `np.array([0, 1])` pour les exemples positifs). À partir de cette représentation, nous pouvons créer un objet `tf.data.Dataset` à partir des données en mémoire avec les instructions suivantes :

```
nsamples = 25000 # number of documents
dim = 300 # embedding dimension

data = np.empty([nsamples, dim], dtype='float32')
label = np.zeros([nsamples, 2], dtype='int32')

# [TODO] fill in data et label with the appropriate data

ds = tf.data.Dataset.from_tensor_slices((data, label))

# visualize the dataset and its elements
print(ds)
print(ds.element_spec)
print(ds_trn.cardinality().numpy())
for item in ds.take(1):
  print(item)
```

À faire :

2. Reprendre la fonction du TP 2 qui calcule la moyenne des plongements des *tokens* pour créer une fonction qui prend en entrée un échantillon (label, résultat de l'analyse spacy) et qui retourne la représentation du document sous la forme d'un *embedding* moyen des lemmes des verbes et adjectifs ainsi que le vecteur à deux dimension encodant l'étiquette.

3. En utilisant cette fonction et en vous inspirant des instructions ci-dessus, créez un `Dataset` pour les données d'apprentissage et un autre pour les données de test.

Pour la phase d'apprentissage, on doit encore faire deux sous-ensembles, l'un pour l'estimation des paramètres et l'autre pour la validation croisée, et créer les *batches*. Il faut évidemment mélanger les données au préalable de manière à ne pas avoir que des exemples positifs ou négatifs au sein d'un *batch*. Les instructions suivantes illustrent les principales opérations de manipulation d'un `Dataset` :

```
# manipulate data set
ds10 = ds.take(10) # takes 10 first samples
ds10_remainder = ds.skip(10) # skip 10 first samples
ds10 = ds.skip(10).take(10) # skip first 10 and take 10 next

# shuffle data, possibly each iteration through ds
ds_shuffled = ds.shuffle(ds_trn.cardinality().numpy(), reshuffle_each_iteration=False)

# batch data into batches of 128 samples, with access optimization
ds_batched = ds.batch(64).prefetch(64).cache()
```

À faire :

4. À partir du `Dataset` pour les données d'apprentissage et en vous appuyant sur les différentes opérations de manipulation d'un `Dataset` ci-dessus, créez les deux ensembles en prenant 80 % des données pour l'estimation des paramètres et 20 % pour la validation[4]. Une bonne pratique consiste à ne pas présenter les échantillons dans le même ordre à chaque itération.

Définition du modèle et apprentissage

Nous avons maintenant nos données prêtes sous forme de `Datasets` et pouvons définir l'architecture d'un premier modèle qui prend en entrée l'*embedding* moyen, projette cette représentation sur une couche cachée avec une activation de type *rectified linear unit* (ReLU) et réalise la prédiction de classe à partir de la couche cachée avec une activation de type *softmax*.

Définition du modèle

La première étape est donc de définir l'architecture de ce modèle en déclarant un modèle séquentiel et en ajoutant progressivement les couches de projection. On utilisera ici une couche de type `Input` pour spécifier la dimension de l'entrée, et deux couches de type `Dense` pour les projections intermédiaires et finales respectivement. Pour chaque couche dense, `tf.keras` permet de spécifier le type d'activation et éventuellement une normalisation des poids, *e.g.* de type L1[5]. Pour éviter le sur-apprentissage, il est également important de spécifier une couche de `Dropout` qui permet de ne réestimer qu'une partie des poids à chaque itération. De manière générique, la définition d'un modèle de ce type se fait par les instructions suivantes :

[4] En pratique, on utilise souvent la fonction `sklearn.model_selection.train_test_split()` de `scikit learn` pour séparer les listes avant de créer les `Dataset` de `tensorflow`

[5] *Cf*. `https://www.tutorialspoint.com/keras/keras_layers.htm` pour plus de précisions sur `tf.Layers`

```
idim = 300 # input dimension
hdim = 100 # hidden dimension

# create an empty model and stack layers one after another
mlp = tf.keras.Sequential() # create an empy model
mlp.add(tf.keras.Input(shape=(idim,))) # empty dimension for batch size
mlp.add(tf.keras.layers.Dense(hdim, activation="relu", name="hidden1"))
mlp.add(tf.keras.layers.Dropout(0.5)) # drop 50% of the weights
mlp.add(tf.keras.layers.Dense(2, activation="softmax", name="output"))

print(mlp.inputs, mlp.layers, mlp.outputs, mlp.output_shape)
```

Une fois l'architecture de votre modèle définie, vous pouvez le construire en spécifiant la fonction de coût à utiliser et la méthode d'optimisation des paramètres. Nous avons choisis ici de représenter la sortie idéale sous la forme d'un encodage *1-hot* avec une valeur de 1 dans la dimension correspondant à la classe à prédire : la fonction de coût adaptée à ce type de représentation est l'entropie croisée entre la distribution de probabilité (sur l'ensemble des classes) prédite sur la couche de sortie et la distribution idéale *1-hot*. Dans `tf.keras`, cette fonction de coût est désignée comme `'categorical_crossentropy'`[6].

Dans notre cas, les instructions suivantes permettent de construire le modèle en spécifiant la métrique qui sera utilisée pour mesurer sa qualité, ici la précision sur la prédiction de la polarité, et de vérifier l'architecture définie :

```
mlp.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
mlp.summary()
```

À faire :

5. Définissez un premier modèle `mlp1` à partir de cet exemple et vérifiez que l'architecture correspond bien à celle que vous attendiez. Combien de paramètre libre ce modèle a-t-il ?

Estimation des paramètres

Une fois le modèle défini, l'estimation des paramètres se fait très simplement en utilisant le `Dataset` préparé pour l'estimation des paramètres et en vérifiant le comportement du modèle sur les données de validation comme suit :

```
# ds_estim is the estimation dataset, ds_valid the validation one
training = mlp.fit(ds_estim, epochs=20, validation_data=ds_valid)
```

La variable `training` retournée par l'apprentissage regroupe les informations sur la manière dont l'apprentissage s'est déroulé au travers de l'objet `training.history` qui regroupe les valeurs de la fonction de coût et de la précision au cours des itérations (*epochs*) pour les données d'estimation des paramètres et celles de validation. L'exemple ci-dessous montre comment récupérer ces valeurs, typiquement pour les afficher et s'assurer que l'apprentissage a bien convergé.

```
print(training.history.keys())

estim_acc = trainin.history['accuracy']
valid_acc = training.history['val_accuracy']
estim_loss = training.history['loss']
valid_loss = training.history['val_loss']
```

[6]Dans le cas d'une classificastion binaire, il et possible de simplifier en n'ayant en sortie qu'une seule valeure, *i.e.*, 0 pour les échantillons négatifs et 1 pour les positifs, auquel cas la fonction de coût adaptée est `'binary_crossentropy'`. Nous choisissons ici un cas plus général qui peut être aisément transposé à des problèmes de classification non binaires.

À faire :

6. Lancez l'apprentissage sur les données préparées en première partie du TP. Visualisez la convergence du modèle au travers de deux graphiques montrant l'évolution sur les données d'estimation et celles de validation de (a) la fonction de coût et (b) la précision. Le modèle converge-t-il correctement ?

7. Relancez l'apprentissage plusieurs fois et regardez la précision sur les données de validation. Comment expliquez-vous les différences observées entre deux apprentissages ? Ces différences sont-elles significatives ?

Utilisation du modèles pour la classification

Dernière étape, tester notre modèle sur les données de test et diagnostiquer ce qui marche bien et ce qui ne marche pas. Une fois les paramètres du modèle appris, deux manières de l'utiliser sur les données de test : on peut donner explicitement des données au modèle et récupérer la prédiction *via* `mlp.predict()` ; ou déléguer à `tf.keras` l'ensemble de l'évaluation avec la fonction `mlp.evaluate()`. L'utilisation de ces deux fonctions est illustrée dans l'exemple suivant en supposant que `ds_test` correspond au `Dataset` créé avec les données de test[7] :

```
mlp.evaluate(ds_test)
out = mlp.predict(ds_test)
```

À faire :

8. Évaluez le modèle précédemment créé sur les données de test, de manière implicite avec `mlp.evaluate()`, puis de manière explicite en se fondant sur les prédictions effectuées avec `mlp.predict()` pour chacun des échantillons du corpus de test. Obtient-t-on la même précision ?

9. Analysez quelques exemples où la prédiction fait une erreur de classification flagrante. Peut-on trouver une explication à l'erreur faite par le classifieur ?

LSTM avec apprentissage de représentation

Dans cette deuxième partie, on souhaite mettre en pratique un réseau récurrent de type LSTM qui analyse l'ensemble du document comme une séquence, produit une représentation du document en sortie du LSTM et réalise la prédiction à partir de cette dernière représentation grâce à un réseau de type perceptron multicouche. À l'inverse de l'approche précédente, on cherche une approche de bout en bout qui optimisera conjointement le plongement des *tokens* (*word embedding*), les paramètres du LSTM qui permettent d'obtenir une représentation du document et la ou les couches de classification.

Préparation des données

Comme dans la première approche, nous allons créer trois `Dataset` pour l'estimation des paramètres, la validation et le test. Mais à l'inverse de l'approche précédentes, chaque document sera représenté par une séquence plutôt que par un vecteur de taille fixe. Chaque séquence est une séquence d'entiers qui correspondent aux indices des *tokens* dans une table listant le vocabulaire. Pour des raisons pratiques, les séquences devront avoir la même longueur : on utilisera un mot arbitraire vide (`''`), encodé par l'indice 0, pour compléter les séquences. Cette opération est connue sous le nom de (*zero*) *padding*.

Cette préparation des données se passe en deux temps : on définit tout d'abord le vocabulaire qui servira à établir la séquence d'entiers qui encode un document ; on encode ensuite chacune des

[7] Le modèle attendant des entrées en *batch*, les données de tests doivent également être organisée sous forme de *batch* sous peine de voir des erreurs. Pour faciliter le travail de predict, on peut prendre une taille de *batch* égale à 1.

séquences avec ce vocabulaire. En pratique, ces deux étapes peuvent être prises en charge par la couche `TextVectorization` de `tf.keras` : pour des raisons pédagogiques, nous préférons ne pas utiliser cette couche dans le cadre de ce TP et définir les étapes d'encodage des textes de manière explicite.

Définition du vocabulaire

La première étape consiste à définir le vocabulaire qui sera utilisé, c'est-à-dire la liste des *tokens* pour lesquels nous allons définir un *embedding*, ainsi qu'un *mapping* permettant de remplacer un *token* par son identifiant (ou indice) dans le vocabulaire. Pour améliorer la couverture avec un vocabulaire de taille fixe, nous normaliserons la casse et supprimerons les *tokens* `x` correspondants à `x.is_punct`, `x.is_space`, `x.is_bracket` et `x.is_quote`.

À faire :

10. Écrire une fonction `normalize_doc()` qui prend en entrée un document tel que fourni par `spaCy` et retourne sous forme d'une liste la suite des *tokens* après normalisation de la casse et suppression des *tokens* inutiles.

11. En utilisant la normalisation ci-dessus, établir la liste des *tokens* apparaissant dans les données d'apprentissage avec le nombre d'occurrence de chacun des *tokens*.

12. Créez avec les 10 000 *tokens* les plus fréquents une table de hashage qui associe un entier unique à chaque *token*. On initialisera cette table avec deux *tokens* spécifiques : `mapping = {'': 0, '<unk>': 1}`. Le premier correspond au *token* vide utilisé pour le *padding*, le second sera utilisé pour représenter les *tokens* hors-vocabulaire, *i.e.*, qui ne figurent pas dans les 10 000 plus fréquents.

Création des `Dataset`

Avec le *mapping* que nous venons de définir, nous pouvons maintenant encoder chaque document de la base de données comme une séquence d'entiers. Toutes les séquences devront avoir la même longueur pour pouvoir représenter l'ensemble des données comme un tenseur unique : on peut choisir une longueur arbitraire et tronquer les séquences trop longues, ou prendre la longueur maximal dans les données d'apprentissage.

À faire :

13. Regardez rapidement la longueur des séquences normalisées dans les données d'apprentissage de manière à fixer la longueur maximale $L_{\max}$ qui sera utilisée pour encoder chaque document.

14. Écrire une fonction qui prend en entrée une séquence normalisée de *tokens*, le *mapping* et $L_{\max}$ et qui renvoie la liste des entiers encodant la séquence (liste de longueur $L_{\max}$).

    Tip. *On peut utiliser la fonction* `tf.keras.preprocessing.sequence.pad_sequences()` *pour assurer le* padding *des séquences.*

Pour finir, nous n'avons plus qu'à transformer l'ensemble des données (apprentissage et test) en `Dataset`. Chaque élément d'un `Dataset` correspond à une séquence d'entiers de longueur $L_{\max}$ et un vecteur *1-hot* de dimension 2 représentant la classe de l'échantillon.

À faire :

15. En adaptant ce que vous avez vu dans la première partie sur la création de `Dataset`, créez les trois `Dataset` pour cette nouvelle représentation. N'oubliez pas le mélange des échantillons et la regroupement en *batches*, indispensables pour l'apprentissage.

16. Vérifiez sur quelques échantillons des données d'apprentissage (avant mélange, c'est plus facile !) que l'encodage fonctionne correctement.

Défintion du modèle

La définition d'un modèle récurrent sur des données séquentielles telles que nous venons de les créer se fait de manière assez directe en utilisant une couche de type `Embedding` dont la sortie est utilisée par une couche de type `LSTM`, dont la sortie de la dernière cellule est à son tour utilisée par des couches de type `Dense` pour la classification , *e.g.* :

```
vocsize = 10002
dim = 64
lstm = tf.keras.Sequential()
lstm.add(tf.keras.layers.Embedding(input_dim=vocsize, output_dim=dim, mask_zero=True)
lstm.add(tf.keras.layers.LSTM(dim, dropout=0.5))
lstm.add(tf.keras.layers.Dense(32, activation="relu"))
lstm.add(tf.keras.layers.Dropout(0.5))
lstm.add(tf.keras.layers.Dense(2, activation="softmax"))
```

À faire :

17. Créez un premier modèle à partir de l'exemple ci-dessus. Vous pouvez adapter les paramètres si vous le souhaitez. Combien de paramètres votre modèle possède-t-il ?

18. Lancez l'estimation des paramètres de votre modèle : on limitera le nombre d'itération à 5, l'estimation des paramètres sur ce type de modèle étant assez chronophage. Comme précédemment, vérifiez la convergence de votre modèle.

19. Évaluez ces performances sur les données de test. Pourquoi l'avantage d'un réseau récurrent est-il limité pour cette tâche ?

Adaptation à la tâche d'un modèle BERT

Dans cette troisième partie, on souhaite mettre en pratique une approche de l'état de l'art qui consiste à adapter un modèle de type BERT à la tâche de classification. Pour cela, une couche de classification (dénommée *tête de classification* dans la terminologie consacrée) est ajoutée : elle prend en entrée le plongement du *token* [CLS] après contextualisation au travers des couches d'auto-attention de BERT et le projette au travers d'une couche `Dense` pour prédire la sortie souhaitée. Il est bien sûr possible de définir soit même l'architecture du modèle de classification en définissant un modèle qui intègre BERT comme une couche[8]. Nous choisissons ici une option différente et plus rapide en utilisant directement un modèle BERT muni d'une tête de classification tel que l'on peut le récupérer *via* la librairie `transformers` de Huggingface qui fait référence en la matière.

Il est à noter que l'adaptation des modèles de type BERT est assez lourde en calcul : il est important d'utiliser si possible un GPU pour l'accélération. Sur colab, il suffit de sélectionner `Change runtime type` dans le menu `Runtime` et de choisir GPU comme accélérateur matériel. Si vous êtes sur votre propre machine, vous pouvez vérifier à l'aide des commandes suivantes si l'utilisation d'un GPU est possible[9].

```
import tensorflow as tf
tf.config.list_physical_devices('GPU') # donne la liste des GPU disponibles
!nvidia-smi # commande nvidia pour voir les caractéristiques du/des GPU

# pour vérifier si votre GPU est bien utilisé
tf.debugging.set_log_device_placement(True) # très, très verbeux !!
```

[8] *Cf.* `https://www.tensorflow.org/text/tutorials/classify_text_with_bert`
[9] Plus de précisions sur `https://www.tensorflow.org/guide/gpu` si vous le souhaitez.

Préparation des données

Le modèle BERT que nous utilisons a été entrainé avec une *tokenisation* particulière en fragments de mots : il est évidemment important d'utiliser la même. La préparation des données sera donc assez différente des deux méthodes précédentes en appelant directement le *tokenizer* associé au modèle `DistilBert` que nous allons utiliser de la manière suivante :

```
from transformers import DistilBertTokenizerFast
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')

encodings = tokenizer(texts, truncation=True, padding=True, is_split_into_words=True)

print(encodings[0]) # pour voir les attributs issus de la tokenisation
print(texts[0]) # liste des tokens pour le 1er texte
print(encodings[0].tokens) # liste après tokenisation
tokenizer.decode(encodings[0].ids) # reconstruction du texte (+ padding)

ds = tf.data.Dataset.from_tensor_slices((dict(encodings),labels))
```

Ici, la variable `texts` est une lise des textes à encoder, chaque texte de la liste étant lui-même une liste de *tokens*/mots. En sortie de `tokenizer`, chaque texte est représenté par plusieurs attributs, notamment `ids` qui contient les identifiants des *tokens* et `attention_mask` qui indique les *tokens* sur lesquels calculer l'auto-attention. La dernière ligne permet de transformer la sortie de `tokenizer` en `tf.Dataset` en y ajoutant les labels à prédire. Comme dans les questions précédentes, on utilisera une représentation *1-hot* du label.

À faire :

20. À partir de la tokenisation `spaCy`, créer la liste des textes au format ci-dessus pour l'apprentissage, la validation et le test. On reprendra les principes de sélection des *tokens* du modèle LSTM (normalisation de la casse, suppression de `is_punct`, `is_space`, *etc.*)

21. Créer les `Dataset` correspondant aux données d'apprentissage, de validation et de test à partir des sorties de `tokenizer`. On s'attachera à identifier les différents éléments retournés par `tokenizer` et à comprendre leur rôle.

Chargement du modèle et apprentissage

Il nous reste maintenant à charger le modèle pré-entrainé au format `tensorflow` depuis la librairie `transformers` de Huggingface et à le rendre opérationnel pour l'apprentissage à l'aide des instructions ci-dessous.

```
from transformers import TFDistilBertForSequenceClassification
bert = TFDistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased')

optimizer = tf.keras.optimizers.Adam(learning_rate=5e-5)
model.compile(optimizer=optimizer, loss='binary_crossentropy')
```

Notez au passage le taux d'apprentissage spécifié pour la méthode d'optimisation par descente stochastique du gradient *Adam*. Nous utilisons ici la même méthode d'optimisation que pour les deux modèles précédents mais avec un pas d'apprentissage beaucoup plus petit (la valeur par défaut dans `tf.keras.optimizers.Adam` est de 0.001. Les modèles transformeurs comme BERT ou sa version simplifié DistilBert[10] sont en effet connu pour nécessiter un pas d'apprentissage plus petit que pour les modèles plus classiques.

[10] *Cf.* `https://huggingface.co/docs/transformers/model_doc/distilbert` pour plus de précisions sur DistilBert

À faire :

20. Charger le modèle et regarder sa structure ainsi que le nombre de paramètres.
21. Lancer l'adaptation du modèle sur une ou deux itérations (selon le temps de calcul) et vérifier sa convergence comme précédemment
22. Évaluer enfin la performance du modèle sur les données de test

Pour aller plus loin

Vous avez maintenant acquis les bases de la mise en œuvre d'approches neuronales pour la classification de documents. Vous pouvez exploiter ces connaissances pour concevoir de nouveaux modèles pour résoudre cette tâche et expérimenter de nouvelles architectures : on pourra par exemple regarder l'impact de normalisations par lot (*batch normalization*), de régularisations, les approches bi-directionnelles, *etc.*

Pour l'approche par représentation sous forme de vecteur de chacun des documents, on peut explorer d'autres manière de représenter les documents, *e.g.* avec des approches de type `doc2vec` ou LDA telles que proposées dans `gensim` – on arrive alors à combiner `spaCy`, `gensim` et `tf.keras`. Peut-on exploiter le lexique sémantique *SentiWordNet* dans ces approches ? On peut aussi apprendre directement les plongements des *tokens* pour la tâche visée plutôt que d'utiliser des plongements génériques de type *word2vec* ou *GloVe*, dès lors qu'on a suffisamment de données d'apprentissage. Dans le cadre d'une approche par MLP comme celle que nous venons de voir, cela peut se faire en utilisant en entrée du réseau une couche de type `Embedding` suivie d'une couche `GlobalAveragePooling1D` avant d'entrer dans les couches de classification. Dans ce cas, les plongements sont appris en même temps que le reste du réseau.

Pour les réseaux récurrents, peut-on étendre l'approche pour tenir compte des étiquettes morphosyntaxiques issues de l'analyse linguistique par `spaCy` ? Une piste possible pour accélérer l'approche par réseau récurrent consiste à filtrer les mots très fréquents des séquences ou, dans le cas de la dernière méthode, à se limiter aux adjectifs et verbes comme nous l'avions fait au TP2. Pour l'amélioration, de nombreuses variantes sont envisageables, notamment avec des approches bi-directionnelles.

De manière similaire, de nombreux modèles de type BERT sont disponibles et peuvent être adaptés à la tâche de détection de polarité. Vous pouvez vous référer au répertoire de `https://huggingface.co/models` pour chercher d'autres modèles et les adapter en suivant la même procédure que celle que nous venons de voir.

Partie 2. Étiquetage de séquences

Dans cette seconde partie, on s'intéresse à la tâche de détection des entités nommées comme un problème d'étiquetage. L'idée est de développer le meilleur système possible à partir de ce que vous avez vu en cours et de ce que vous avez appris sur la mise en œuvre de modèles neuronaux sous `tf.keras` dans la première partie.

On utilisera pour cette partie les données de la compétition CONLL 2003 qui consiste en trois jeux de données étiquetées avec un modèle BIO, respectivement pour l'apprentissage, la validation et le test. Vous pouvez récupérer à l'aide des instructions suivantes :

```
!wget http://people.irisa.fr/Guillaume.Gravier/teaching/ENSAI/data/conll03-trn.json
!wget http://people.irisa.fr/Guillaume.Gravier/teaching/ENSAI/data/conll03-val.json
!wget http://people.irisa.fr/Guillaume.Gravier/teaching/ENSAI/data/conll03-tst.json
```

À faire :

23. Récupérez les données et regardez le format et la signification des différentes colonnes.

24. Écrire une fonction qui lit un fichier et stock en mémoire les tokens et les différentes étiquettes associées aux tokens.

25. Transformez les données de manière à pouvoir les utiliser ensuite avec `tf.keras` dans un modèle séquentiel de type réseau récurrent. Pour les séquences de *tokens*, on suivra une procédure similaire à celle utilisée dans la première partie du TP pour le modèle LSTM. Pour chacun des tokens d'une séquence, l'étiquette associée sera représentée sous la forme d'un vecteur *1 hot* : pour une séquence de longueur L, les étiquettes seront donc représentées sous la forme d'une matrice $L \times N$ où N est le nombre d'étiquette possible[11].

Pour démarrer, nous allons partir d'un modèle simple définit comme suit :

```
# maxlen : longueur maximale d'une séquence
# emdn_dim : dimension des word embeddings
# lstm_dim : dimension des vecteurs dans le LSTM

ner1 = tf.keras.Sequential()
ner1.add(tf.keras.layers.InputLayer(input_shape=(maxlen, )))
ner1.add(tf.keras.layers.Embedding(input_dim=ntokens, output_dim=embed_dim, mask_zero=True))
ner1.add(tf.keras.layers.LSTM(lstm_dim, return_sequences=True))
ner1.add(tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(ntags)))
ner1.add(tf.keras.layers.Activation('softmax'))
```

À faire :

26. Créez le modèle `ner1` et lancez l'estimation des paramètres à partir des données d'apprentissage et de validation que vous aurez préparées.

27. Décodez avec ce modèle les données de test de manière à récupérer les étiquettes prédites pour chacun des séquences.

28. Pourquoi ne peut-on pas utiliser directement la fonction `evaluate` ?

29. Écrivez une fonction qui calcul pour chacune des étiquettes `B-*` et `I-*` ($* \in \{$`LOC`, `PER`, `ORG`, `MISC`$\}$) le rappel et la précision ainsi que le rappel et la précision globale pour l'ensemble de ces étiquettes. Quel score obtient-on ?

À vous de jouer ! Vous avez maintenant un premier modèle pour la détection d'entités nommées : à vous de l'améliorer en optimisant la configuration et en vous appuyant sur les techniques vues en cours pour l'étiquetage de séquence. On pourra avantageusement consulter la documentation des `layers` proposés dans `tf.keras`.

[11] Par convention, on associe l'étiquette `'O'` aux *tokens* ajoutés artificiellement pour le *padding* des séquences.