# Rapport de projet

ECE Paris – Deep Learning Challenge



TEIGER Max
HALBEHER Clément
MAUNICK Mathis

Pr : Côme Arvis

# Nos différentes approches

## Extraction de features

Notre première approche a consisté à réaliser un modèle sans utiliser de vecteurs d'enchevêtrements (fournis par le projet GloVe, ou encore word2vec ). Nous avons ainsi utilisé le Tokenizer fournis par la librairie Keras, afin de représenter nos phrases en une suite de nombres, chacun représentant un mot.

Nous avons de plus extrait nous-même des features depuis les deux phrases du dataset original, comme la longueur de chaque phrase, le mot le moins utilisé dans le dataset présent dans chaque phrase la différence de longueur de chaque phrase, moyenne des vecteurs de chaque phrase ou encore une features contenant les deux vecteurs multipliés.

Nous nous sommes rapidement rendu compte que ces nouvelles caractéristiques que nous avions définies ne seraient pas concluantes nous avons cependant nous avons décider de construire un premier modèle en guise « d'échauffement ».

Ce modèle était très simple et était constitué uniquement de Dense et de Dropout layers (Keras).

Nous avons choisi comme outils de mesure la Categorical Cross Entropy pour la fonction de perte et l'optimiseur Adam.

Ce modèle n'était pas performant, nous avons obtenu avec celui-ci un résultat proche d'une fonction aléatoire. Nous avons donc décidé de passer à une autre méthode pour aborder ce problème.

## Model with GloVe : Global Vectors for Word Representation

Notre deuxième approche a consisté à utiliser les vecteurs d'enchevêtrement fournis par le projet GloVe ( Jeffrey Pennington, Richard Socher, Christopher D. Manning ). Le projet GloVe met à disposition des vecteurs d'enchevêtrement pré-entrainés sur des volumes de données conséquents, qui peuvent ainsi venir nourrir notre réseau de neurones. Ils proposent de nombreux formats de vecteurs, obtenus de façon différente et de plusieurs dimensions. Pour des raisons d'efficacité de calcul et de praticité, nous avons choisi de nous servir des vecteurs de dimension 50 obtenus depuis Wikipedia et Gigaword (400.000 mots).

La première étape, après avoir récupéré et avoir lu avec Python le fichier contenant les vecteurs, est de transformer nos données. Il a donc fallu transformer chaque phrase du dataset initial en un ensemble de vecteurs d'intégration. Nous nous sommes servi du snippet de code qui suit :

```
def sentence2sequence(sentence, wordmap, visualize=False):
    """

    Turns an input sentence into an (n,d) matrix,
        where n is the number of tokens in the sentence
        and d is the number of dimensions each word vector has.

    """
    tokens = sentence.lower().split(" ")
    rows = []
    words = []
    #Greedy search for tokens
    for token in tokens:
        i = len(token)
        while len(token) > 0 and i > 0:
            word = token[:i]
            if word in wordmap:
                rows.append(wordmap[word])
                words.append(word)
                token = token[i:]
                i = len(token)
            else:
                i = i-1

    if visualize: return rows, words
    else: return rows
```

Nous avons ensuite « normalisé » ces données, pour cela il a fallu fixer une taille maximale pour chaque phrase, que nous avons fixée à 80 mots. Nous n'avons pas choisi ce nombre au hasard, après avoir observé nos données, nous nous sommes rendus compte que les phrases tenaient en moyenne une vingtaine de mots, et que le très peu de phrases de plus de 80 mots étaient souvent dénouées de sens.

Une fois les données transformées, nous nous sommes inspirés de l'architecture du « Asynchronous Deep Interaction Network » (ADIN - Di Liang, Fubao Zhang, Qi Zhang, Xuanjing Huang) sans toute fois le reproduire, pour des raisons de complexité et de temps disponible.

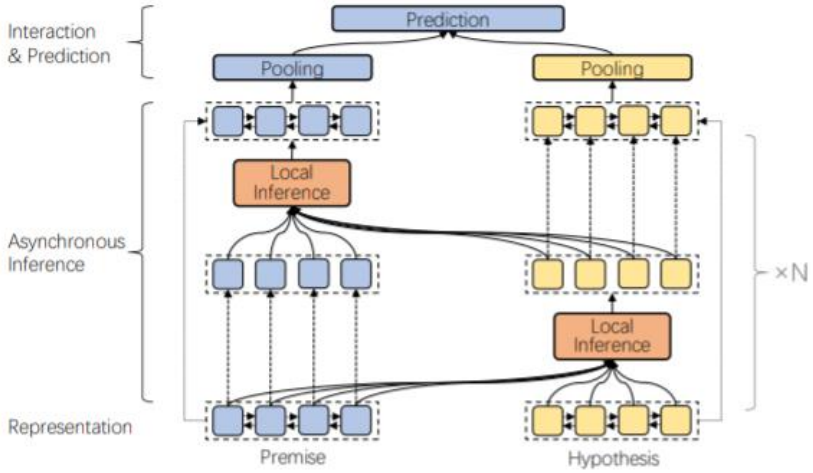


*Schéma Général ADIN 1*

Nous avons donc repris se schéma en enlevant les cellules « Local Inférence ». Nous nous sommes servis d'une première couche Long-Short Term Memory Bidirectionnelle (BiLSTM) suivie d'une couche LSTM emboitée sur une couche Dense elle-même suivie de la couche de sortie.

Après avoir essayé ce modèle sur un échantillon des inputs proposées (environ 5000 lignes utilisées), nous nous sommes rendu compte que le modèle posait un problème d'over-fitting très rapidement. Nous avons donc décidé d'ajouter des couches dropout afin de d'amoindrir ce problème.

Nous obtenons ainsi un modèle de 965, 571 paramètres entrainables.

Voici le code du modèle :

```
inputs1 = Input(shape=(80,50))
inputs2 = Input(shape=(80,50))

lstm_layer_1 = Bidirectional(LSTM(256, return_sequences=True))
x1 = lstm_layer_1(inputs1)
x2 = lstm_layer_1(inputs2)

dropout_layer = Dropout(0.2)
x1 = dropout_layer(x1)
x2 = dropout_layer(x2)

lstm_layer_2 = LSTM(128)
x1 = lstm_layer_2(x1)
x2 = lstm_layer_2(x2)

x1 = dropout_layer(x1)
x2 = dropout_layer(x2)

dense = Dense(64, activation='relu')
x1 = dense(x1)
x2 = dense(x2)

x1 = dropout_layer(x1)
x2 = dropout_layer(x2)

x = Concatenate(axis=-1)([x1,x2])
predictions = Dense(3, activation='softmax')(x)

model_1 = Model(inputs=[inputs1, inputs2], outputs=predictions)
model_1.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['acc'])

model_1.summary()
```

## Modèle avec une couche Embedding

Nous avons ensuite décider d'implémenter un modèle en créant nous même notre Embedding. Pour cela, il a fallu transformer autrement nos données. Nous nous sommes donc servis du Tokenizer inclut dans keras. Celui-ci, en lui donnant une liste de mots donne un identifiant unique a chacun d'entre eux. Cet identifiant est un nombre entier, cela permettra au modèle d'effectuer ses calculs, ce qu'il ne pourrait pas faire avec les phrases brutes. Chacune de ces phrases est ensuite envoyé dans un embedding layer, lui-même relié à un layer LSTM puis à un layer Dense avant la couche de sortie.

```
inputs1 = Input(shape=(80,))
inputs2 = Input(shape=(80,))

embedding_layer = Embedding(vocab_size, 128, input_length=max_length)
emb_out1 = embedding_layer(inputs1)
emb_out2 = embedding_layer(inputs2)

lstm_layer = LSTM(128)
x1 = lstm_layer(emb_out1)
x2 = lstm_layer(emb_out2)

dropout_layer = Dropout(0.3)
x1 = dropout_layer(x1)
x2 = dropout_layer(x2)

dense = Dense(64, activation='relu')
x1 = dense(x1)
x2 = dense(x2)

dropout_layer = Dropout(0.2)
x1 = dropout_layer(x1)
x2 = dropout_layer(x2)

x = Concatenate(axis=-1)([x1,x2])
predictions = Dense(3, activation='softmax')(x)

model_3 = Model(inputs=[inputs1, inputs2], outputs=predictions)
model_3.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

Ce modèle comporte ainsi 6,540,227 paramètres entrainables.

## Modèle RoBERTa

BERT (Bidirectional Encoder Representations from Transformers) est une technique révolutionnaire de pré-entraînement auto-surveillé qui apprend à prédire les sections de texte intentionnellement cachées (masquées). Il est essentiel de noter que les représentations apprises par BERT se sont bien généralisées aux tâches en aval, et lorsque BERT a été publié pour la première fois en 2018, il a obtenu des résultats de pointe sur de nombreux ensembles de données de référence du PNA.

RoBERTa, développé par l'équipe AI de Facebook, s'appuie sur la stratégie de masquage linguistique de BERT et modifie les hyperparamètres clés du modèle, notamment en supprimant l'objectif de pré-entraînement de la phrase suivante de BERT et en augmentant considérablement le nombre de mini-lots et le taux de formation. RoBERTa a également été formé sur un ordre de grandeur plus de données que BERTa, pour une durée plus longue. Ceci permet aux représentations RoBERTa de généraliser encore mieux les tâches en aval par rapport à BERTa.

## Résultats obtenus

Avec ces modèles, nous avons obtenu des résultats corrects. La principale difficulté résidait dans le temps nécessaire à leur entrainement.

Le premier modèle prend 6370 secondes par époque pour entrainer ses 965.571 paramètres. Nous ne réalisons donc qu'une seule période et arrivons à une accuracy de 0.45989, soit environ 46%, sur le jeu de données de validation.

Pour l'améliorer, nous aurions pu utiliser des vecteurs d'embedding GloVe de taille plus importante, ceux-ci donnant ainsi plus d'information sur l'embedding de chaque mot. Utiliser des layers supplémentaires n'aurait pas forcément amélioré la qualité du modèle, celui-ci ayant déjà tendance à overfitter sur le training set.

Le deuxième modèle prend environ 4000 secondes pour entrainer ses presque 7.000.000 de paramètres. Nous obtenons cependant des résultats très corrects avec ce modèle, puisque nous sommes montés à plus de 55% de précision sur le set de validation, ainsi que sur le dataset de test soumis sur la plateforme Kaggle.

Ce modèle peut toujours être amélioré en augmentant la dimension de la couche embedding, ainsi que la taille du vocabulaire utilisé, seulement, pour des raisons d'efficacité de calcul, il était plus simple pour nous de limiter ces paramètres à des valeurs raisonnables.

Sans surprise, avec le modèle RoBERTa les résultats sont bien meilleurs, nous avons obtenus une précision de plus de 90%.

## Principaux problèmes rencontrés

Formatter les données brutes en données exploitable a été un vrai problème, surtout quand il a été question de réaliser ces transformations sur l'ensemble des données. En effet, il a fallu adapter nos méthodes afin que la RAM soit libérée plus facilement et ce pour toutes les tâches impliquant un aussi gros volume de données.

Il fallait également penser à créer un modèle pertinent, c'est-à-dire limiter les paramètres inutiles et toutes les opérations allant avec. Cela toujours dans le but d'accélérer les calculs, ce projet ayant été réalisé dans un temps limité.

A cela s'ajoutait des problèmes de développement classique, problèmes de dépendances ainsi que prise en main de l'outil « Google Colab ».

En effet celui-ci avait un problème récurrent : l'environnement de développement proposé par Google Colab se déconnectait parfois sans raison apparente, ou au bout d'un certain temps d'inactivité même lorsque l'exécution n'est pas terminée.

Nous avons eu plusieurs fois cette mauvaise surprise, en pensant retrouver un modèle entrainé, l'exécution s'était juste arrêtée et nous devions recommencer depuis le début.

Nous avons cependant trouvé une solution à ce problème en ajoutant un code dans la console du navigateur :

```
function ClickConnect(){
console.log("Working");
document.querySelector("colab-toolbar-button#connect").click()
}
setInterval(ClickConnect,60000)
```

Ce code empêche l'environnement de se déconnecter en cliquant régulièrement sur la page automatiquement.

## Sources

Asynchronous Deep Interaction Network for Natural Language Inference:
https://www.aclweb.org/anthology/D19-1271.pdf

Keras documentation :

https://keras.io/layers/core/

Medium articles:

https://medium.com/@shivamrawat_756/how-to-prevent-google-colab-from-disconnecting-717b88a128c0

NLP Progress :

http://nlpprogress.com/english/natural_language_inference.html

Roberta :

https://github.com/pytorch/fairseq/blob/master/examples/roberta/README.md

https://pytorch.org/hub/pytorch_fairseq_roberta/