# Introduction

Le but de ce projet était de mettre en place différents modèles de traitement naturel du langage et de les comparer. Ces réseaux NPL permettent de faire de la détection de sentiments dans des phrases. Dans ce projet, nous avons implémenté chacun une manière de faire. Nous allons expliquer ci-dessous nos deux méthodologies.

# Fonctionnalité

## Transformation des données

### Méthode 1 :

Nous avions à notre disposition trois fichiers : un pour le train, test et la validation. Pour chacun, nous avons dû uniformiser les phrases. En commençant par retirer les stop words, puis en ajoutant le caractère de bourrage <PAD> aux phrases les plus petites et en coupant les plus longues. Par la suite, nous avons construit le vocabulaire en concaténant tous les sets de données. À la suite de cela, nous avons transformé chacun des ensembles en ID selon la place des mots dans le vocabulaire et refait du bourrage pour être sûr que nos tensors avaient la même taille. Enfin, pour construire les dataloader, nous avons indexé chaque sentiment et créé des tensors comprenant à la fois les textes et les labels des sets.

Pour faire ce que nous avons décrit, nous avons mis en place plusieurs fonctions disponibles dans le fichier Data.py :

- Remove_StopWord : comme son nom l'indique, cela permet de retirer les stop words en utilisant la fonction *remove_stopwords* de la bibliothèque *parsing.preprocessing*
- PAD_words : cette fonction va permettre le bourrage ou la réduction des phrases.
- Load_file : une simple fonction pour lire les fichiers et séparer le texte des sentiments en lisant les fichiers phrase par phrase.

Concernant l'encodage one hot, nous le faisons pendant les différentes phases d'entrainement. Pour cela, dans l'itération sur le dataloader, nous avons une seconde boucle qui permet de parcourir les mots de chaque phrase pour récupérer son indice et l'encoder. Enfin, nous donnons cela à notre modèle dans la même boucle pour que notre modèle s'applique à chaque mot.

### Méthode 2 :

Dans la deuxième méthode de préparation des données, nous suivons plusieurs étapes pour transformer notre jeu de données textuel en une forme utilisable par un modèle.

#### Création du Vocabulaire :

On commence par générer un vocabulaire à partir des textes d'entraînement. Ce vocabulaire est une liste de tous les mots uniques présents dans le jeu de données d'entraînement. Il servira à encoder les phrases en vecteurs basés sur les mots du vocabulaire.

### Nettoyage des Données : Suppression des Mots Vides

Les mots vides sont retirés des textes d'entraînement, de validation et de test. Les mots vides sont des mots fréquents (comme "the", "and", "of" en anglais) qui n'apportent pas beaucoup de sens et peuvent introduire du bruit dans l'analyse.

### Padding des Textes :

Chaque phrase est d'abord découpée en mots individuels, puis on y ajoute du padding (<PAD>) pour que toutes les phrases aient la même longueur. Cela permet d'aligner les phrases pour que le modèle puisse traiter les données par lots. Après le padding, les listes de mots sont recomposées en chaînes de caractères.

### Encodage des Sentiments :

Les sentiments associés à chaque phrase sont transformés en valeurs numériques. Un dictionnaire est créé pour mapper chaque sentiment unique à un entier (par exemple, "positif" devient 0, "négatif" devient 1). Ainsi, chaque sentiment est représenté par un nombre, ce qui facilite le traitement par le modèle.

### Définition de la Classe DatasetN :

Cette classe organise les données dans une structure compatible avec PyTorch, facilitant l'accès aux phrases, aux sentiments et au vocabulaire. Les phrases sont encodées en utilisant le vocabulaire pour créer des représentations one-hot. Chaque mot est représenté par un vecteur indiquant sa position dans le vocabulaire, ce qui rend les données plus faciles à manipuler pendant l'entraînement.

### Création des Jeux de Données :

Trois jeux de données sont créés : train_dataset pour l'entraînement, val_dataset pour la validation et test_dataset pour les tests. Chacun est structuré à l'aide de la classe DatasetN, qui organise les phrases, les sentiments et le vocabulaire de manière uniforme.

### Création des DataLoaders :

Les DataLoaders chargent les jeux de données en petits lots pendant l'entraînement, la validation et les tests. Cela rend le processus d'entraînement plus efficace et gère automatiquement le traitement par lots .

## Entrainement du modèle

Par la suite, nous définissons notre modèle (RNN, LSTM ou GRU), la fonction d'optimisation et de perte, nous expliciterons ceux-ci plus tard dans la partie Résultats. Cela nous permet ensuite de commencer l'entrainement du modèle défini.

Pour cela, nous avons trois phases : l'entrainement, la validation et enfin le test. Cela correspond à ce que nous faisons classiquement pour l'entrainement d'un réseau de neurones.

Pendant la validation, nous avons également fait de l'early stopping pour ne pas surentrainer notre réseau et optimiser le temps d'entrainement.

# Modèles

## RNN

Notre RNN ou réseau de neurones récurrents est constitué de plusieurs couches linéaires et d'une activation Relue et softmax. Voici les différentes couches définies dans l'initiation du modèle :

- I2e : input to embedding, elle prend en entrée le vecteur de la taille *input_size* et produit un vecteur d'embedding de taille *emb_size*. Ces deux métriques sont définies en paramètre de l'initialisation du réseau.
- Concat2h : concatenation to hidden, permet de faire une concaténation des embeddings et de l'état de la couche cachée. Cette couche génère un vecteur de taille *hidden_size* et permet de calculer une couche cachée à partir de la combinaison de l'entrée actuelle et de la couche cachée de l'itération précédente.
- H2o, hiden to output, comme son nom l'indique, cette couche permet de donner la couche cachée à l'output du modèle de taille *output_size* correspondant au nombre de sentiments présents dans nos textes.
- LogSoftmax : il s'agit de la fonction d'activation appliquée à la sortie.

Vient ensuite la fonction de forward. Dans celle-ci, nous commençons par donner l'entrée en vecteur d'embedding à l'aide de la couche i2e. Ensuite, nous mettons à jour la couche cachée en combinant l'embedded d'input avec la couche cachée précédente via la couche concat2h. Cette combinaison est passée à travers une couche Relue. Enfin, la couche cachée est donnée à h2o et à la fonction de softmax.

## LSTM

Comme pour le RNN, ce modèle Long Short-Term Memory est constitué de plusieurs couches. Cependant, son architecture particulière permet de capturer des dépendances à long terme et permet de réguler le flux d'informations à travers le réseau.

- Embelling_layer : permet de transformer les données d'entrée en vecteur de taille *emb_size*
- Forget_layer : cette couche détermine quelles informations de la couche d'embelling devraient être oubliées. Elle prend les données d'entrée actuelles et l'état caché précédent. La sortie est une valeur entre 0 et 1 grâce à la fonction sigmoid qui est appliquée aux couches.
- Input_layer : cela correspond à l'input gate de LSTM, elle calcule la nouvelle information qui est ajoutée à l'état de la cellule. Pour cela, on combine la couche cachée et celle d'entrée en leur appliquant la fonction tanh.
- Output_layer : permet de déterminer la sortie du modèle grâce à une sigmoid.

En plus de cela, nous devons mettre à jour l'état de la cellule en fonction de la couche d'oubli et de l'input gate. Ensuite, nous mettons à jour la couche cachée avec l'output et la cellule calculée précédemment.

Enfin, nous recalculons l'output grâce à la couche linéaire h2end (hidden to end) et en appliquant une fonction softmax.

# Résultats

Dans cette partie, nous allons expliciter notre meilleur résultat pour chaque modèle selon la 1ere méthode. Pour chacun, la fonction d'optimisation utilisée est Adam et CrossEntropyLoss pour la fonction de perte.

| | Eta | Batch size | Embeling size | Hidden size | Epoch | Accuracy |
|---|---|---|---|---|---|---|
| RNN | 0.001 | 64 | 64 | 128 | 10 | 86.64% |
| LSTM | 0.0008 | 32 | 64 | 128 | 10 | 88.26% |

## Analyse comparative:

### Learning Rate (Eta):

LSTM utilise un taux d'apprentissage légèrement plus bas (0.0008) par rapport au RNN (0.001). Cela peut suggérer que le modèle LSTM a besoin d'un apprentissage plus lent et plus stable pour converger.

### Batch Size :

Le RNN utilise un batch size plus élevé (64) par rapport au LSTM (32). Un plus grand batch size peut accélérer l'entraînement, mais cela peut aussi augmenter la variance des gradients. Le LSTM, avec un batch size plus petite, semble mieux se comporter dans ce cas précis, offrant une meilleure précision.

### Embedding Size et Hidden Size:

Les deux modèles ont les mêmes valeurs pour l'embedding size (64) et hidden size (128), ce qui les rend comparables à cet égard. Ces paramètres semblent bien adaptés aux deux architectures.

### Accuracy :

LSTM obtient une meilleure précision de 88.26% contre 86.64% pour le RNN.

# Conclusion

Nous avons donc proposé deux méthodes différentes pour réaliser un classifieur sur du texte et des sentiments. Puis, nous avons développé deux modèles (RNN et LSTM).

En termes de précision, le modèle LSTM surpasse le modèle RNN avec une augmentation de 1.62%. Cela pourrait être dû à la capacité du LSTM à mieux capturer les dépendances à long terme grâce à sa structure de mémoire.

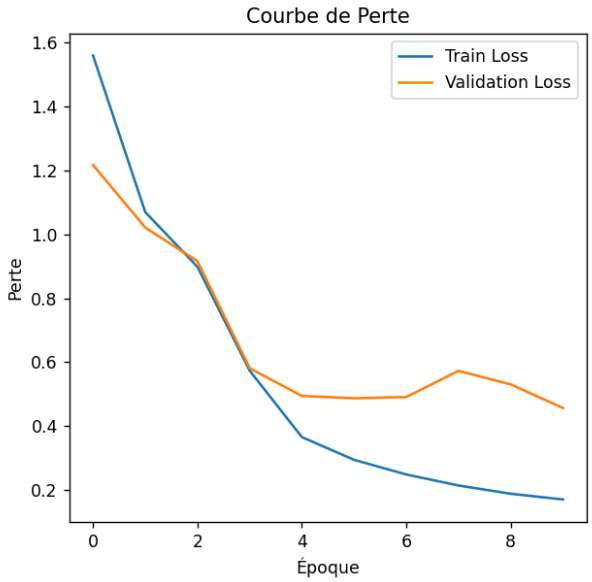
Le LSTM semble mieux adapté pour cette tâche, même avec un learning rate plus faible et un batch size plus petite.

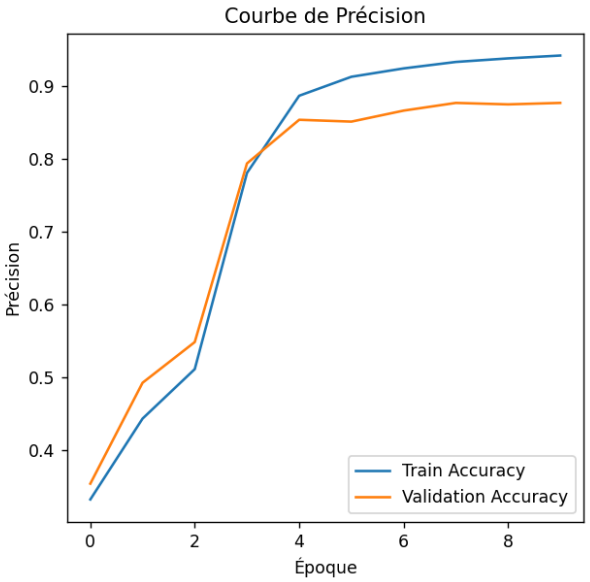
En résumé, bien que le RNN offre une précision respectable, le LSTM semble être un choix plus performant bien que légèrement.

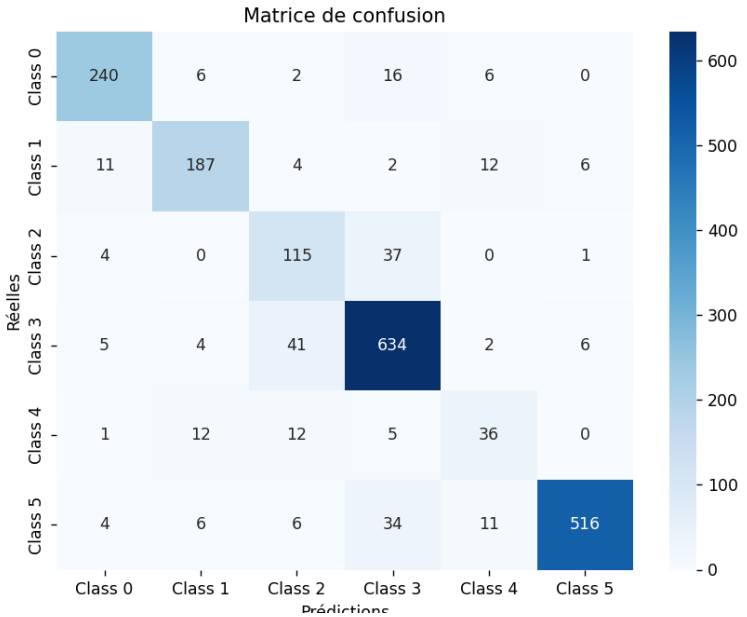
# Annexe

Vous pouvez retrouver ci-dessous les courbes de perte et de précision pour chaque modèle, ainsi que la matrice de confusion finale.

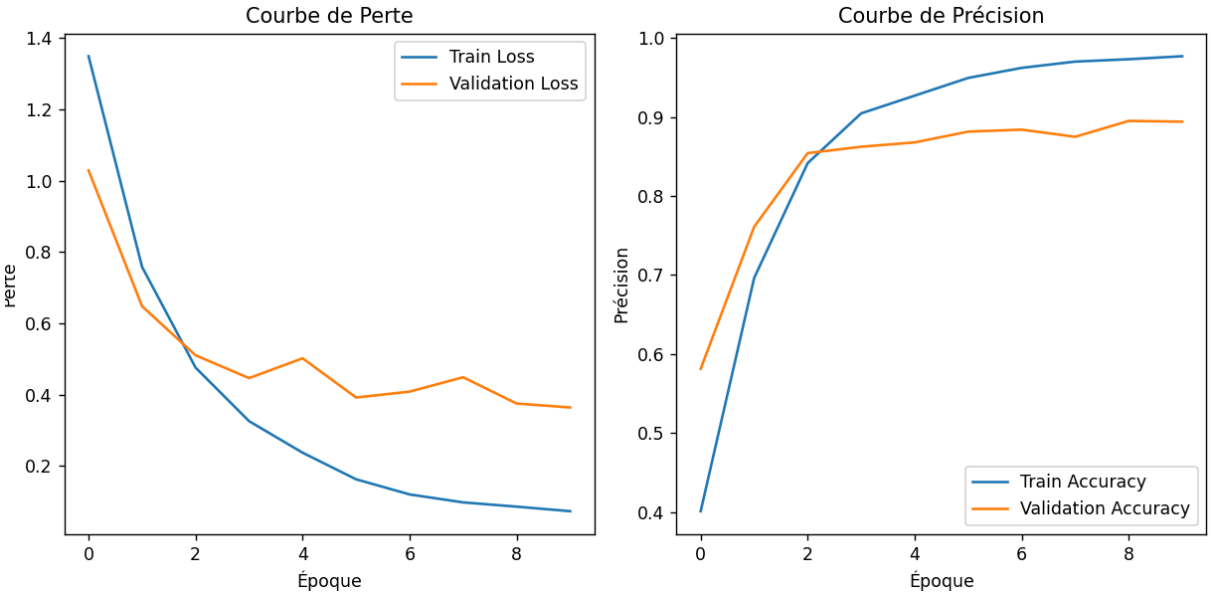
## RNN

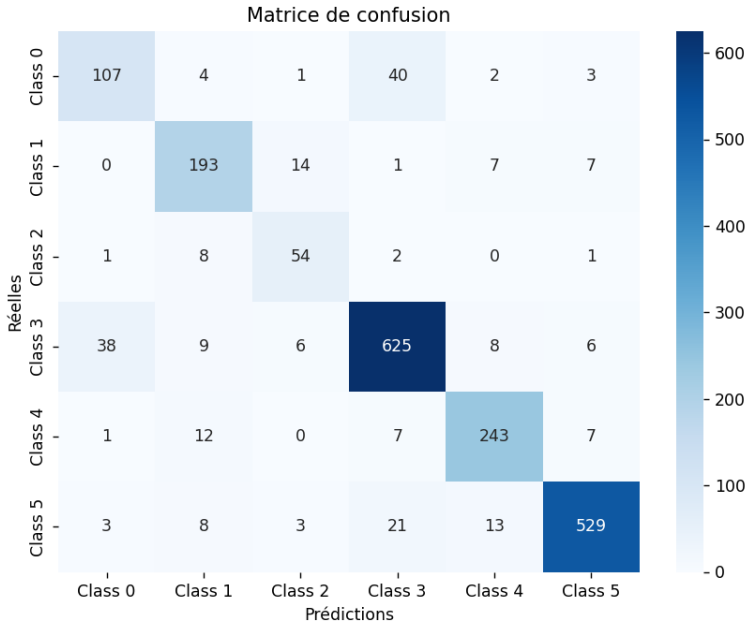


# LSTM


Courbe de Perte
Train Loss
Validation Loss
Perte
Époque
Courbe de Précision
Train Accuracy
Validation Accuracy
Précision
Époque


Matrice de confusion
Class 0
Class 1
Class 2
Class 3
Class 4
Class 5
Réelles
Prédictions
107 4 1 40 2 3
0 193 14 1 7 7
1 8 54 2 0 1
38 9 6 625 8 6
1 12 0 7 243 7
3 8 3 21 13 529
600
500
400
300
200
100
0