



## Machine Translation Mini-Project

Réalisé par : Malick Diaw - Vanda Martins Barbosa

Supervisé par : Pr. Abdelhak Mahmoudi

Tout d'abord, nous tenons à remercier notre professeur **Pr. Abdelhak Mahmoudi** de nous avoir permis d'apprendre plus et de pratiquer avec un projet. Avec nos recherches faites pour réaliser ce projet et avec nos erreurs lors de sa réalisation, nous avons pu apprendre et comprendre beaucoup plus sur le Traitement du Langage Naturel et le deep learning.

Ceci est un rapport illustrant notre projet du module **Natural Language Processing (NLP)**, semestre 2 du **Master Ingénierie de Données et Développement Logiciel (IDDL)** portant sur la réalisation d'une **Traduction Automatique (Machine Translation)**.

Ce projet fait objet de traduire un texte d'un langage source en un autre texte d'un langage de destination.

Dans ce projet, on a réalisé deux modèles de traduction, l'un pour une traduction de Français en Portugais et l'autre de l'anglais en Français. Ces deux modèles sont illustrés et expliqués en détail étapes par étapes dans un **jupyter NoteBook** pour chacun.

Pour sa réalisation, nous avons construit un réseau de neurones profonds en utilisant le mécanisme de **Transformer** « *Attention is all you need* » pour former un modèle de Machine Translation.

Les différentes étapes fondamentales sont :

- Le chargement des librairies et du dataset pour l'entraînement
- Le pre-processing, nettoyage et tokenization
- La création des Classes et Fonctions pour construire un modèle de Transformer
- La construction et l'entraînement du modèle
- Et enfin la prédiction pour visualiser les résultats.

### ✓ Chargement des libraries et du dataset

## Loading the libraries

```
In [1]: import os
import time
import re
from unicodedata import normalize
import string

import pandas as pd
import numpy as np
import pickle
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow import keras

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import GRU
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Embedding
from tensorflow.keras.layers import RepeatVector
from tensorflow.keras.layers import TimeDistributed
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras import layers

import tensorflow_datasets as tfds

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

import sklearn
from sklearn.model_selection import train_test_split
```

Ici on charge les données avec un nombre de lignes souhaité utilisant le DataFrame **pandas**, puis on mélange les données selon les lignes utilisant **sklearn** et ensuite on sauvegarde les données utilisées dans un fichier. Enfin, on récupère les **Features** dans une variable « *input\_data* » et les **Targets** dans une autre variable « *target\_data* ».

## Loading the dataset

The data can be found in `data_eng_fr/fra_eng.txt`. The data will be loaded with a desired number of lines per pandas DataFrame and will be shuffled again. Then we get the `input` and `target` columns in two variables to apply the preprocessing functions.

```
In [5]: # Load the dataset: sentence input and sentence output
data_file = pd.read_csv(train_filenamepath,
                       sep="\t",
                       header=None,
                       names=[INPUT_COLUMN, TARGET_COLUMN],
                       usecols=[0,1],
                       nrows=NUM_SAMPLES
                      )

print(data_file)
data_file = sklearn.utils.shuffle(data_file) # shuffle the dataset

# Save the dataset selected as csv and txt file
data_file.to_csv(DATA_PATH+'/data_file.csv', header=None, index=False)
data_file.to_csv(DATA_PATH+'/data_file.txt', sep="\t", header=None, index=False)

# load the input and target data
input_data = data_file[INPUT_COLUMN].values
target_data = data_file[TARGET_COLUMN].values

print('\nNumber of sentences: ', len(input_data))
print(input_data[:5])
print(target_data[:5])
```

## ✓ Pre-processing

Cette fonction aide à nettoyer les données, en enlevant d'abord les accents puis les rendre en minuscule, garde quelques ponctuations (?) . !) en les décollant du mot et

remplace les ponctuations (' et -) en un espace. Ensuite supprime les chiffres et réduit les espaces en une seule.

The following function will be used to apply data cleansing.

```
In [6]: # clean a list of lines
def clean_preprocess_text(lines):
    clean_pair = list()
    for line in lines:
        # normalize unicode characters
        line = normalize('NFD', line).encode('ascii', 'ignore')
        line = line.decode('UTF-8')
        # convert to lowercase
        line = line.lower()
        # put space before and after punctuation
        line = re.sub(r"([?.!])", r" \1 ", line)
        # replace ' and - with a space
        line = re.sub(r"[-]", r" ", line)
        # remove digits
        remove_digits = str.maketrans('', '', string.digits)
        line = line.translate(remove_digits)
        # reduces spaces into one
        line = re.sub(r" +", r" ", line)
        # remove space at start and end of line
        line = line.strip()
        # add the cleaned line in a new table which will contain all the new sentences
        clean_pair.append(line)

    return np.array(clean_pair)
```

Apply cleansing on input and target data

Pour qu'un réseau de neurones puisse prédire sur des données textuelles, il doit d'abord être transformé en données qu'il peut comprendre. Les données textuelles telles que "student" sont une séquence d'encodages de caractères ASCII. Étant donné qu'un réseau de neurones est une série d'opérations de multiplication et d'addition, les données d'entrée doivent être des nombres.

```
In [8]: # fit a tokenizer
def create_tokenizer(lines):
    to_exclude = '#$&(*+,-/:;<=>@\[\]\\]^`{|}-\t\\n'
    to_tokenize = '.!?"
    tokenizer = tf.keras.preprocessing.text.Tokenizer(filters=to_exclude)
    tokenizer.fit_on_texts(lines)
    return tokenizer

# max sentence length
def max_length(lines):
    # add 2 to max length because sos_token(start sent) and eos_token(end sent) will be added in the sentence
    return max(len(line.split()) for line in lines) + 2

# encode the corpus and add paddind to max length
def encode_sequences(corpus, tokenizer, max_length, sos_token, eos_token):
    # encode the corpus
    sentences = tokenizer.texts_to_sequences(corpus)

    # add START and END token to the corpus
    sentences = [sos_token + sentence + eos_token for sentence in sentences]

    # Pad the sentences
    sentences = tf.keras.preprocessing.sequence.pad_sequences(sentences,
                                                               value=0,
                                                               padding='post',
                                                               maxlen=max_length
                                                               )

    return sentences
```

Voyons ensuite comment préparer les données pour notre modèle. C'est très simple et les étapes sont les suivantes :

- Créer du vocabulaire à partir du corpus en utilisant la tokenisation des mots. La fonction `create_tokenizer()` crée le tokenizer en utilisant **Keras** pour obtenir le token pour chaque mot du corpus, en acceptant une certaine ponctuation ('.!?'').
- Calculez la longueur maximale des séquences d'entrée et de sortie. fonction `max_length()`
- Convertir du texte brut en une séquence d'entiers. `encode_sequences()`
- Remplissage de phrases : on remplit des zéros à la fin des séquences pour que toutes les séquences aient la même longueur. Sinon, nous ne pourrons pas entraîner le modèle par Batch.

## ✓ Modèles

Pour ce projet, on a présenté *deux types* de modèles, l'un c'est un encoder-decoder construit en utilisant les **Réseaux de Neurones Récurrents (RNN, Recurrent Neural Network)** et l'autre un modèle **Transformer encoder-decoder avec Attention**.

Mais nous adopterons au modèle Transformer qui sera utilisé dans l'application de traduction automatique, car le Transformer est une architecture de modèle évitant la récurrence et reposant entièrement sur un mécanisme d'attention pour dessiner des dépendances globales entre l'entrée et la sortie. Le Transformer permet beaucoup plus de parallélisation et se repose entièrement sur l'auto-attention pour calculer les représentations de son entrée et de sa sortie sans utiliser de RNN alignés sur la séquence ou de convolution.

### Model 1: Encoder-Decoder RNN

Let's create an encoder-decoder model in the cell below.

```
In [14]: # define NMT model
def Encoder_decoder_model_RNN(src_vocab, tar_vocab, src_length, tar_length, n_units):
    model = keras.models.Sequential()
    model.add(keras.layers.Embedding(src_vocab, n_units, input_length=src_length, mask_zero=True))
    model.add(keras.layers.LSTM(n_units))
    model.add(keras.layers.RepeatVector(tar_length))
    model.add(keras.layers.LSTM(n_units, return_sequences=True))
    model.add(keras.layers.TimeDistributed(keras.layers.Dense(tar_vocab, activation='softmax')))

    model.compile(loss='sparse_categorical_crossentropy',
                  optimizer='adam')
    return model

# Train the neural network
enc_dec_rnn_model = Encoder_decoder_model_RNN(input_vocab_size,
                                                output_vocab_size,
                                                LIMITE,
                                                LIMITE,
                                                256)
enc_dec_rnn_model.summary()
print()

enc_dec_rnn_model.fit(X_train,
                      Y_train,
                      batch_size=64,
                      epochs=50
                      )

Epoch 42/50
10000/10000 [=====] - 82s 8ms/sample - loss: 1.6934
```

## Model 2: Transformer

La réalisation des Classes et Fonctions pour construire ce modèle est bien détaillé dans le NoteBook avec explication pour chaque bloc de code.

Dans ce notebook, on a réalisé un modèle Transformer Encoder-Decoder comme illustré par la figure ci-dessous.

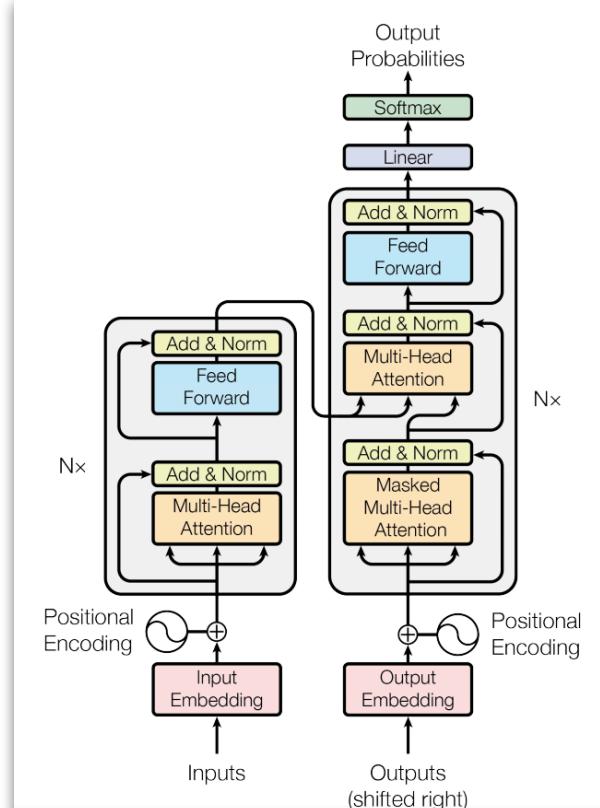


Figure 1: The Transformer - model architecture.

La méthode « `call` » de notre classe Transformer qui hérite de « `keras.model` » reçoit en entrée l'encoder, le décoder et training qui sera True ou False. Et enfin retourne un output en sortie comme indiqué par la figure ci-dessus.

```
def call(self, enc_inputs, dec_inputs, training):
    # Create the padding mask for the encoder
    enc_mask = self.create_padding_mask(enc_inputs)
    # Create the mask for the causal attention
    dec_mask_1 = tf.maximum(
        self.create_padding_mask(dec_inputs),
        self.create_look_ahead_mask(dec_inputs)
    )
    # Create the mask for the encoder-decoder attention
    dec_mask_2 = self.create_padding_mask(enc_inputs)
    # Call the encoder
    enc_outputs = self.encoder(enc_inputs, enc_mask, training)
    # Call the decoder
    dec_outputs = self.decoder(dec_inputs,
                               enc_outputs,
                               dec_mask_1,
                               dec_mask_2,
                               training)
    # Call the Linear and Softmax functions
    outputs = self.last_linear(dec_outputs)

    return outputs
```

## ✓ Entrainement du modèle

On a défini deux fonctions (*train\_step* et *valid\_step*) qui seront appelées lors de chaque étapes d'entraînement et de validation par la fonction *main\_train*.

Dans la fonction *main\_train*, on entraîne le modèle par lots (par Batch en anglais), donc nous avons créé un générateur de données par Batch en utilisant la bibliothèque tf.data et la fonction *batch\_on\_slices* sur les séquences d'entrée et de sortie.

### Create the batch data generator

- Create a batch data generator: we want to train the model on batches, group of sentences, so we need to create a Dataset using the tf.data library and the function *batch\_on\_slices* on the input and output sequences.

```
In [13]: # Define a dataset for training
train_dataset = tf.data.Dataset.from_tensor_slices(
    (encoder_inputs, decoder_outputs))
train_dataset = train_dataset.shuffle(len(encoder_inputs), reshuffle_each_iteration=True).batch(
    BATCH_SIZE, drop_remainder=True)

train_dataset = train_dataset.prefetch(tf.data.experimental.AUTOTUNE)
```

```
In [14]: # Define a dataset for validation
valid_dataset = tf.data.Dataset.from_tensor_slices(
    (encoder_inputs_valid, decoder_outputs_valid))
valid_dataset = valid_dataset.shuffle(len(encoder_inputs_valid), reshuffle_each_iteration=True).batch(
    BATCH_SIZE, drop_remainder=True)

valid_dataset = valid_dataset.prefetch(tf.data.experimental.AUTOTUNE)
```

Pour chaque époque dans *main\_train*, on print la durée d'epoch et crée un checkpoint et enregistre le modèle pendant l'entraînement. Car l'entraînement peut prendre beaucoup de temps et nous pouvons restaurer le modèle pour un entraînement ou utilisation future.

```
# Checkpoint the model on every epoch
ckpt_save_path = ckpt_manager.save()
print("Saving checkpoint for epoch {} in {}".format(epoch+1, ckpt_save_path))
```

A la fin de l'entraînement *main\_train*, on print la durée total d'entraînement et retourne les tableaux de loss et d'accuracy de l'entraînement et validation pour les illustrer dans des graphes.

```
train_time = time.time() - start_train
s = int(train_time % 60)
m = int(train_time // 60) % 60
h = int(train_time // 60) // 60
print("Total training time : {}:{}:{}.".format(h, m, s))

return losses, accuracies, losses_valid, accuracies_valid
```

Avec seulement 10 epochs (époques) et 20.000 lignes de phrases sélectionnées, il nous a fallut 7 heure 7 minutes et 9 secondes pour l'entraînement du modèle et au final 8% d'accuracy. Ce qui explique qu'il nous faut de très bon matériels pour ce genre de réalisation, mais l'essentiel c'était les étapes et l'écriture des classes et fonctions pour construire le modèle Transformer.

Mais après on a de quelques bonnes résultats sur nos prédictions.

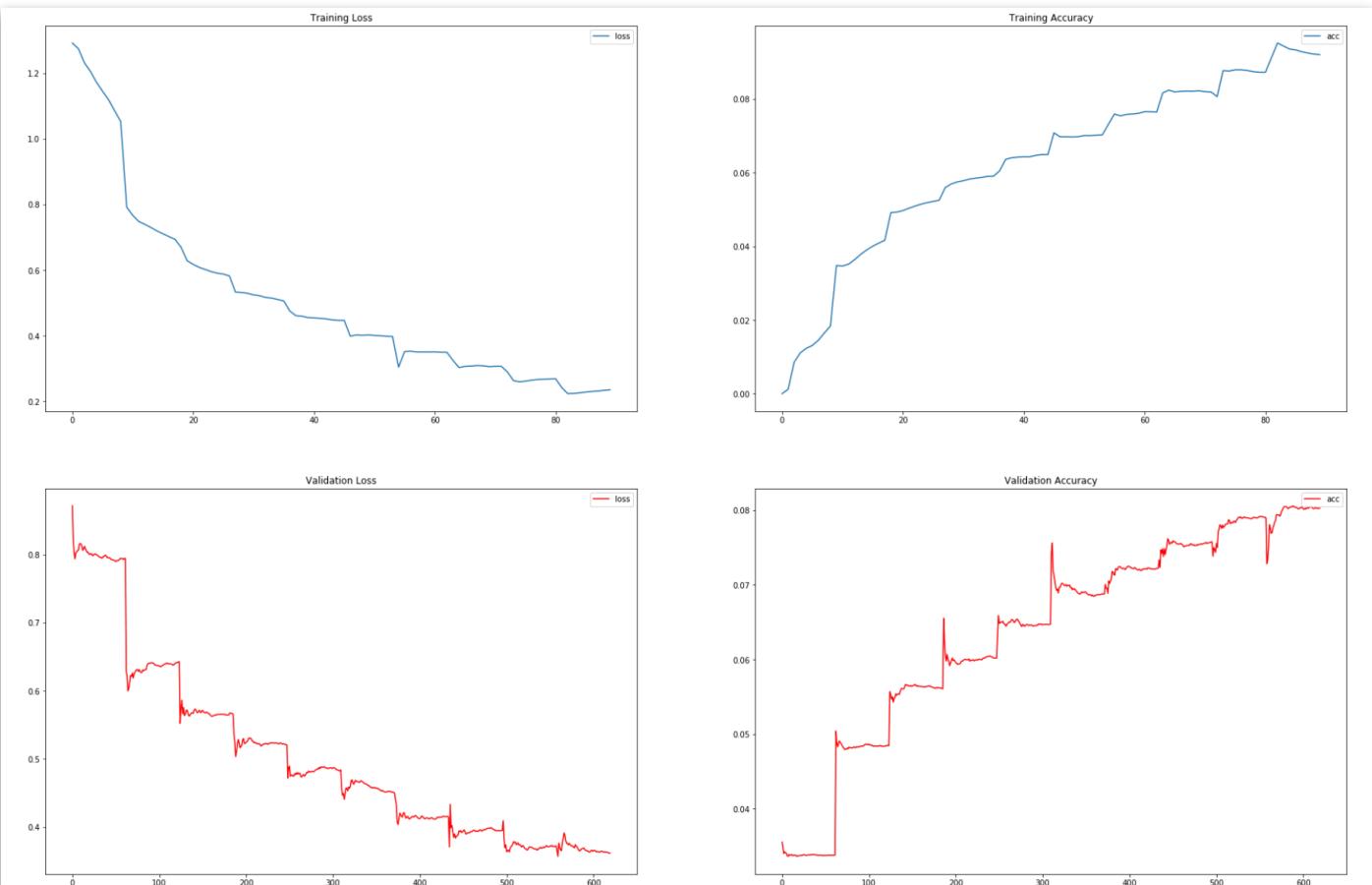
Et grâce au checkpoints, on pourra restaurer et entraîner encore le modèle plus tard. Et c'est ce qu'on a fait avec encore 10 epochs, pour obtenir de meilleurs résultats que précédents.

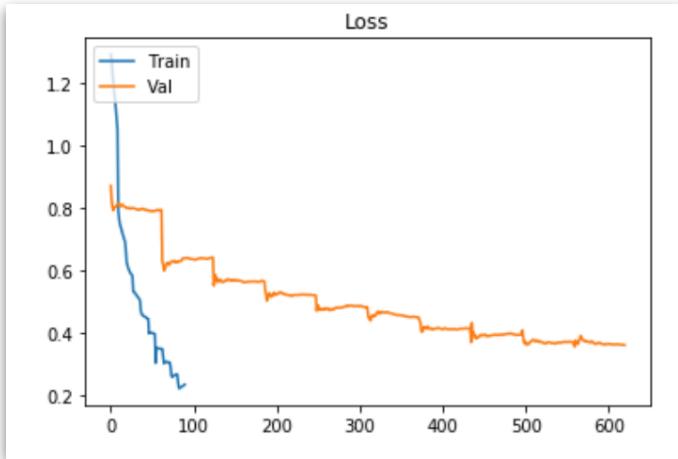
```
In [30]: # Train the model
losses, accuracies, losses_valid, accuracies_valid = main_train(train_dataset,
                                                               valid_dataset,
                                                               transformer,
                                                               EPOCHS,
                                                               30
)
Epoch 9 Batch 0 Loss 0.2002 Accuracy 0.0790
Saving checkpoint for epoch 9 in /Users/milkzo/Documents/Jupyter/mini_projet/data_eng_fr/checkpoint/ckpt-9
Epoch 9, Validation Loss 0.3719, Validation Accuracy 0.0790
Time for this epoch: 2499.1698381900787 secs

Starting epoch 10
Epoch 10 Batch 0 Loss 0.2416 Accuracy 0.0912
Epoch 10 Batch 30 Loss 0.2230 Accuracy 0.0952
Epoch 10 Batch 60 Loss 0.2235 Accuracy 0.0944
Epoch 10 Batch 90 Loss 0.2255 Accuracy 0.0936
Epoch 10 Batch 120 Loss 0.2276 Accuracy 0.0933
Epoch 10 Batch 150 Loss 0.2295 Accuracy 0.0928
Epoch 10 Batch 180 Loss 0.2309 Accuracy 0.0925
Epoch 10 Batch 210 Loss 0.2331 Accuracy 0.0922
Epoch 10 Batch 240 Loss 0.2348 Accuracy 0.0921
Saving checkpoint for epoch 10 in /Users/milkzo/Documents/Jupyter/mini_projet/data_eng_fr/checkpoint/ckpt-10
Epoch 10, Validation Loss 0.3613, Validation Accuracy 0.0803
Time for this epoch: 2640.213226556778 secs

Total training time : 7:7:9
```

## ❖ Affichage des graphes de loss et accuracy :





Sur cette figure ci-contre, on remarque bien que notre modèle a besoin encore plus d'entraînement car le loss d'entraînement ne cesse pas de diminuer et n'est pas encore stable.

## ❖ Résultats des prédictions (1er train) tester sur les données de la validation :

```
In [19]: for i in range(10):
    print("Review:", seq2text(tokenizer_inputs, encoder_inputs_valid[i]))
    print("Original summary:", seq2text(tokenizer_outputs, decoder_outputs_valid[i]))
    print("Predicted summary:", predict_from_seq(encoder_inputs_valid[i], output_max_length))
    print("\n")

Review: who invented the telephone ?
Original summary: qui inventa le telephone ?
Predicted summary: qui le telephone a regarde le telephone ?

Review: good job !
Original summary: beau travail !
Predicted summary: bonne travail !

Review: they made fun of my accent .
Original summary: ils se sont moqués de mon accent .
Predicted summary: ils sont vraiment amusant de mon affaires .

Review: i ate a slice of ham .
Original summary: j ai mange une tranche de jambon .
Predicted summary: j ai mange une pierre .

Review: i could not afford to buy a bicycle .
Original summary: je n aurais pas les moyens de m acheter un velo .
Predicted summary: je n aurais pas pu m acheter un velo .

Review: what country were you born in ?
Original summary: dans quel pays etes vous nes ?
Predicted summary: quel pays etait tu née ?

Review: i give up .
Original summary: j abandonne .
Predicted summary: je me donne moi .
```

## ❖ Résultats des prédictions (2eme train) tester sur les données de la validation :

```
In [38]: for i in range(10):
    print("Review:", seq2text(tokenizer_inputs, encoder_inputs_valid[i]))
    print("Original summary:", seq2text(tokenizer_outputs, decoder_outputs_valid[i]))
    print("Predicted summary:", predict_from_seq(encoder_inputs_valid[i], output_max_length))
    print("\n")

Review: this is a very rare specimen .
Original summary: c est un exemplaire tres rare .
Predicted summary: c est un appareil pauvre .

Review: tom pulled out his gun .
Original summary: tom sortit son arme .
Predicted summary: tom sortit son arme

Review: tom isn't good looking .
Original summary: tom n'est pas beau .
Predicted summary: tom n'est pas bon .

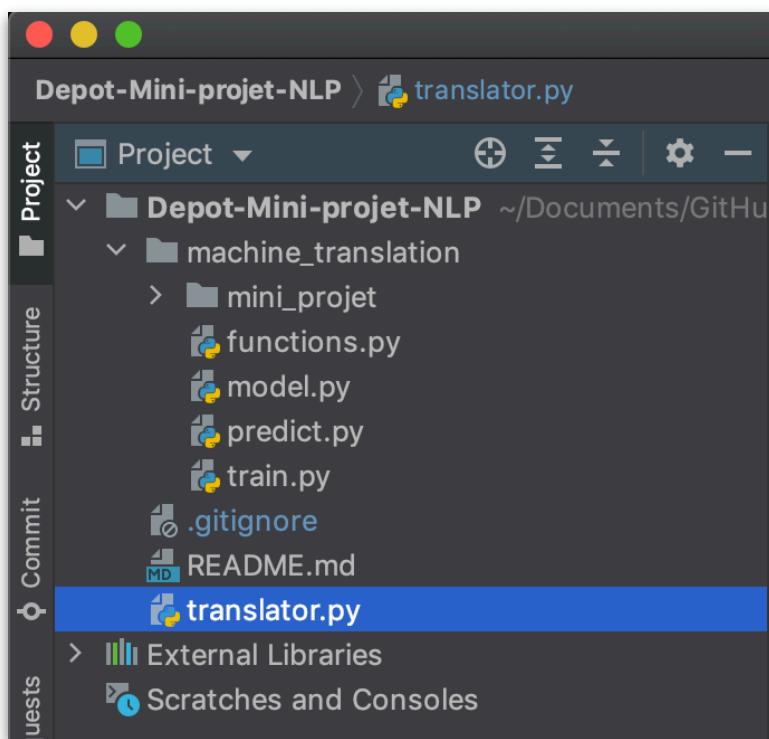
Review: that's not what i heard .
Original summary: ce n'est pas ce que j'ai entendu .
Predicted summary: ce n'est pas ce que j'ai entendu .

Review: i almost forgot .
Original summary: j'ai presque oublié .
Predicted summary: j'ai presque oublié .

Review: he has been studying for two hours .
Original summary: il a étudié pendant deux heures .
Predicted summary: il a fait deux heures deux heures .

Review: wait !
Original summary: attends !
Predicted summary: attendez .
```

## ✓ L'application réalisé pour la traduction automatique :



Le dossier **Depot-Mini-projet-NLP** est le projet de l'application de traduction, il contient à sa racine un fichier python nommé « **translator.py** » qui est le fichier à exécuter. Ce fichier crée l'interface de l'application avec ses composants champs de texte, bouton, etc... et fait appel, lorsqu'on clique sur le bouton **Traduire**, à la fonction **translate()** qui est dans le fichier **predict.py** qui créera le modèle correspondant et renvoie le résultat prédit.

Le dossier **mini\_projet** contient les données et modèles correspondants à chaque traduction et les fichiers jupyter NoteBook.

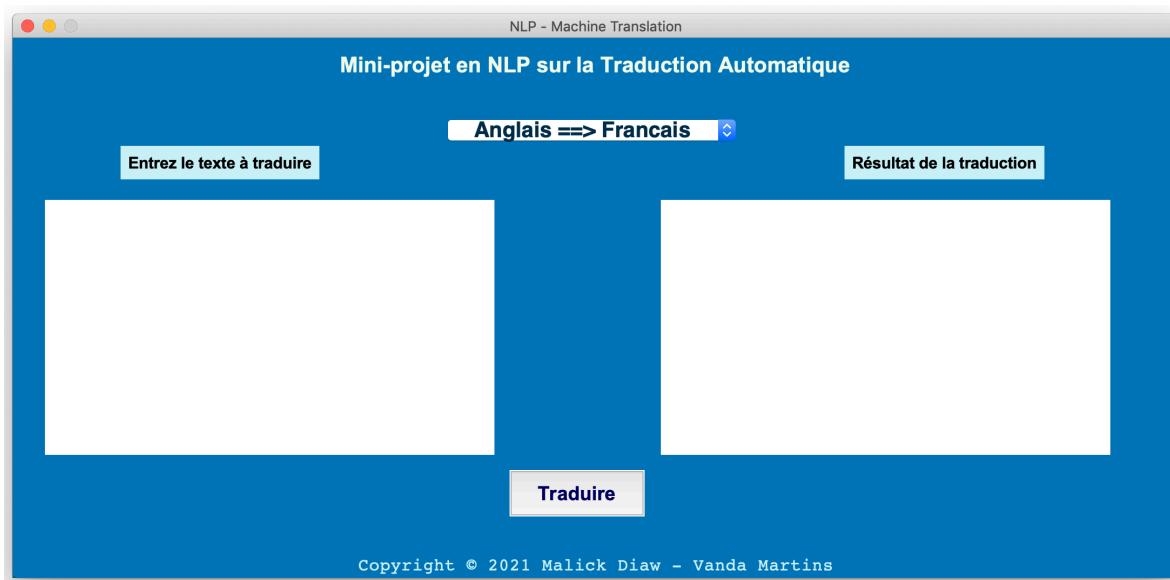
Le fichier **model.py** contient les classes et fonctions pour construire un modèle Transformer.

Le fichier **predict.py** contient la fonction qui charge les paramètres du modèle et le crée, et des fonctions pour encoder et décoder le texte pour ensuite retourner le résultat prédict.

Le fichier **functions.py** contient quelques à utiliser par les autres fichiers.

Le fichier **train.py** contient l'ensemble du code de l'importation de librairies à la prédiction des résultats.

## ❖ Interface de l'application :



## ❖ Quelques traductions :

