RAPPORT MINI PROJET NLP

Réalisé par : lina Mahrouch & Riad Elajaje

I.INTRODUCTION

Le but de ce mini projet est de réaliser un pipeline de traduction automatique de bout en bout qui accepte le texte anglais en entrée et renvoie la traduction française.

Ce projet prend n'importe quel texte anglais et le convertit en séquences d'entiers basés sur un vocabulaire français et anglais suffisamment grand et le transmet à un modèle qui renvoie une distribution de probabilité sur les traductions possibles avec une précision supérieure à 97%.

II.MOTS CLES

1-Machine learning
2-Traduction
3-Pipeline
4-Dataset
5-Conversion
6-VWT
7-Python
8-Numpy
9-Tensorflow
10-Keras.

III.Technologies utilisées.

1-Python : nous avons utilisé la version 3.7 de python. (le projet nécessite python version3+ et les librairies suivantes installées).
2-Numpy : rapides et polyvalents, nous avons utilisés Numpy pour les concepts de vectorisation, d'indexation et de diffusion.
3-Tensorflow : Libraire open source pour ai.
4-Keras : qui fournit des api pour ai, nous l'avons utilisé dans notre projet, en utilisant la (sequence to sequence Learning).
Dataset : Nous avons utilisé WMT comme dataset, pour ses riches dataset utilisés pour la traduction.

IV. Traduction.

1-Alors comment se passe la traduction dans notre projet ?

Après avoir tokenisé le texte et effectué tous les pré-traitements, nous le transmettons à une couche d'intégration de mots, puis à 2 LSTM bidirectionnels avec 256 unités, puis à une couche TimeDistributed avec une fonction d'activation softmax pour produire une distribution de probabilité.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 15, 256)           50944
_________________________________________________________________
bidirectional_1 (Bidirection (None, 512)               1050624
_________________________________________________________________
repeat_vector_1 (RepeatVecto (None, 21, 512)           0
_________________________________________________________________
bidirectional_2 (Bidirection (None, 21, 512)           1574912
_________________________________________________________________
time_distributed_1 (TimeDist (None, 21, 344)           176472
=================================================================
Total params: 2,852,952
Trainable params: 2,852,952
Non-trainable params: 0
_________________________________________________________________
None
```

1-Sequence to sequence learning ?

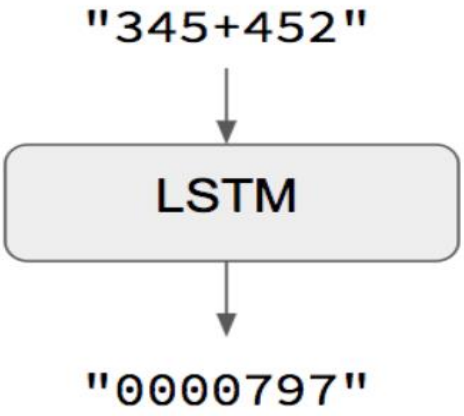
L'apprentissage séquence à séquence (Seq2Seq) consiste à entraîner des modèles pour convertir des séquences d'un domaine (par exemple, des phrases en anglais) en séquences dans un autre domaine (par exemple, les mêmes phrases traduites en français).

```
"the cat sat on the mat" -> [Seq2Seq model] -> "le chat etait assis sur le tapis"
```

Cela peut être utilisé pour la traduction automatique ou pour une réponse à une question gratuite (générer une réponse en langage naturel à partir d'une question en langage naturel) - en général, il est applicable à chaque fois que vous avez besoin de générer du texte.

Déroulement :

Lorsque les séquences d'entrée et les séquences de sortie ont la même longueur, vous pouvez implémenter de tels modèles simplement avec une couche Keras LSTM ou GRU (ou une pile de celles-ci). C'est le cas dans cet exemple de script qui montre comment apprendre à un RNN à apprendre à additionner des nombres, encodés sous forme de chaînes de caractères :



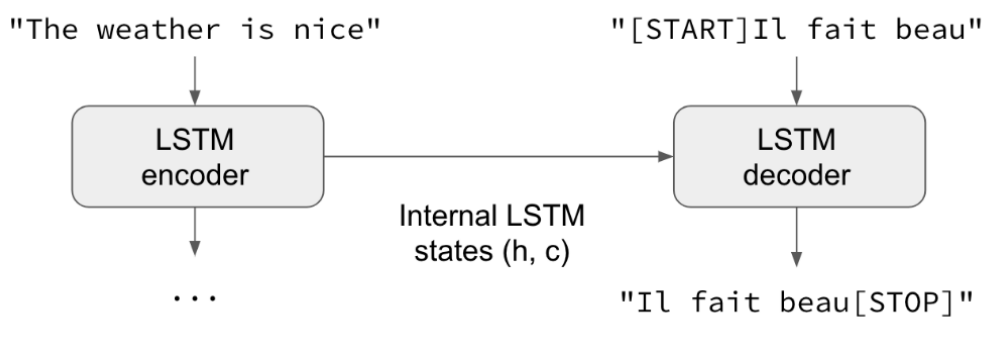
Une mise en garde de cette approche est qu'elle suppose qu'il est possible de générer la cible[...t] à partir de l'entrée[...t]. Cela fonctionne dans certains cas (par exemple, l'ajout de chaînes de chiffres) mais ne fonctionne pas dans la plupart des cas d'utilisation. Dans le cas général, des informations sur l'ensemble de la séquence d'entrée sont nécessaires pour commencer à générer la séquence cible.

2-Le cas général : séquence-à-séquence canonique

Dans le cas général, les séquences d'entrée et les séquences de sortie ont des longueurs différentes (par exemple, la traduction automatique) et l'intégralité de la séquence d'entrée est requise pour commencer à prédire la cible. Cela nécessite une configuration plus avancée, ce à quoi les gens se réfèrent généralement lorsqu'ils mentionnent des "modèles séquence à séquence" sans autre contexte. Voilà comment cela fonctionne :

Une couche RNN (ou un empilement de celle-ci) joue le rôle d'« encodeur » : elle traite la séquence d'entrée et renvoie son propre état interne. A noter que nous écartons les sorties de l'encodeur RNN, ne récupérant que l'état. Cet état servira de "contexte", ou "conditionnement", du décodeur à l'étape suivante.

Une autre couche RNN (ou pile de celle-ci) joue le rôle de « décodeur » : elle est entraînée à prédire les prochains caractères de la séquence cible, étant donné les caractères précédents de la séquence cible. Concrètement, il est entraîné à transformer les séquences cibles en séquences identiques mais décalées d'un pas de temps dans le futur, un processus d'entraînement appelé « forçage enseignant » dans ce contexte. Il est important de noter que le codeur utilise comme état initial les vecteurs d'état du codeur, c'est ainsi que le décodeur obtient des informations sur ce qu'il est censé générer. En effet, le décodeur apprend à générer des cibles[t+1...] à partir de cibles[...t], conditionnées à la séquence d'entrée.



En mode inférence, c'est-à-dire lorsque nous voulons décoder des séquences d'entrée inconnues, nous passons par un processus légèrement différent :

1) Encoder la séquence d'entrée en vecteurs d'état.

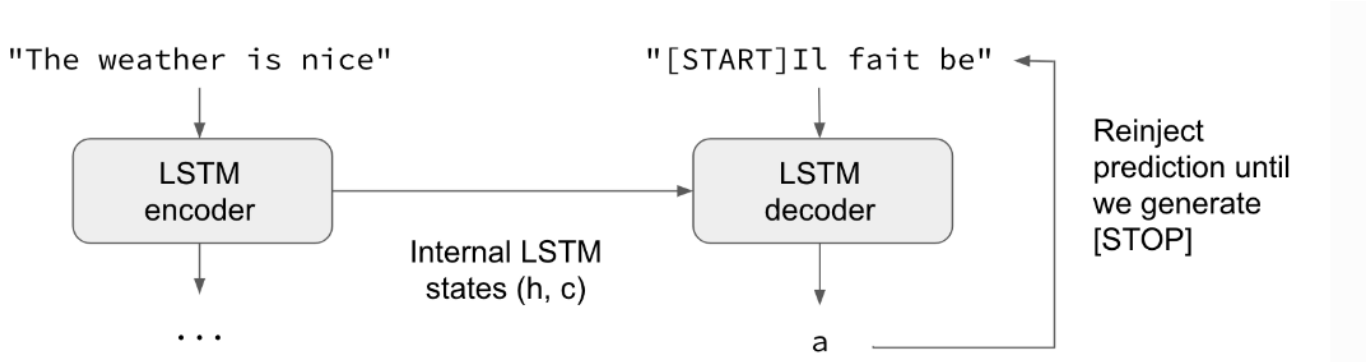
2) Commencez avec une séquence cible de taille 1 (juste le caractère de début de séquence).

3) Fournissez les vecteurs d'état et la séquence cible à 1 caractère au décodeur pour produire des prédictions pour le caractère suivant.

4) Échantillonnez le caractère suivant en utilisant ces prédictions (nous utilisons simplement argmax).

5) Ajouter le caractère échantillonné à la séquence cible

6) Répétez jusqu'à ce que nous générions le caractère de fin de séquence ou que nous atteignions la limite de caractères.



Le même procédé peut également être utilisé pour entraîner un réseau Seq2Seq sans « teaching forcing », c'est-à-dire en réinjectant les prédictions du décodeur dans le décodeur.

3-Explication du code:

Voici un résumé de notre processus :

1) Transformez les phrases en 3 tableaux Numpy, encoder_input_data, decoder_input_data, decoder_target_data :encoder_input_data est un tableau 3D de forme (num_pairs, max_english_sentence_length, num_english_characters) contenant une vectorisation one-hot des phrases anglaises.
decoder_input_data est un tableau 3D de forme (num_pairs, max_french_sentence_length, num_french_characters) contenant une vectorisation one-hot des phrases françaises.
decoder_target_data est identique à decoder_input_data mais décalé d'un pas de temps.
decoder_target_data[:, t, :] sera le même que decoder_input_data[:, t + 1, :].

2) Former un modèle Seq2Seq de base basé sur LSTM pour prédire decoder_target_data en fonction de encoder_input_data et decoder_input_data. Notre modèle utilise le forçage de l'enseignant.

3) Décoder quelques phrases pour vérifier que le modèle fonctionne (c'est-à-dire transformer les échantillons de encoder_input_data en échantillons correspondants de decoder_target_data).

Étant donné que le processus d'apprentissage et le processus d'inférence (phrases de décodage) sont assez différents, nous utilisons des modèles différents pour les deux, bien qu'ils exploitent tous les mêmes couches internes.

C'est notre modèle de formation. Il exploite trois fonctionnalités clés des RNN Keras : L'argument du constructeur return_state, configurant une couche RNN pour renvoyer une liste où la première entrée correspond aux sorties et les entrées suivantes aux états RNN internes. Ceci est utilisé pour récupérer les états de l'encodeur.

L'argument d'appel inital_state, spécifiant le ou les états initiaux d'un RNN. Ceci est utilisé pour transmettre les états du codeur au décodeur en tant qu'états initiaux.

L'argument du constructeur return_sequences, configurant un RNN pour qu'il renvoie sa séquence complète de sorties (au lieu de simplement la dernière sortie, dont le comportement par défaut). Ceci est utilisé dans le décodeur.

```
from keras.models import Model
from keras.layers import Input, LSTM, Dense

# Define an input sequence and process it.
encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
# We discard `encoder_outputs` and only keep the states.
encoder_states = [state_h, state_c]

# Set up the decoder, using `encoder_states` as initial state.
decoder_inputs = Input(shape=(None, num_decoder_tokens))
# We set up our decoder to return full output sequences,
# and to return internal states as well. We don't use the
# return states in the training model, but we will use them in inference.
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
                                     initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

# Define the model that will turn
# `encoder_input_data` & `decoder_input_data` into `decoder_target_data`
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
```

Nous entraînons notre modèle sur deux lignes, tout en surveillant la perte sur un ensemble retenu de 20 % des échantillons.

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
          batch_size=batch_size,
          epochs=epochs,
          validation_split=0.2)
```

Après environ une heure sur un processeur MacBook, nous sommes prêts pour l'inférence. Pour décoder une phrase de test, nous allons à plusieurs reprises :

1) Encoder la phrase d'entrée et récupérer l'état initial du décodeur

2) Exécutez une étape du décodeur avec cet état initial et un jeton "début de séquence" comme cible. La sortie sera le prochain caractère cible.

3) Ajoutez le caractère cible prédit et répétez.

****

Voici notre configuration d'inférence :

```
encoder_model = Model(encoder_inputs, encoder_states)

decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_outputs, state_h, state_c = decoder_lstm(
    decoder_inputs, initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model = Model(
    [decoder_inputs] + decoder_states_inputs,
    [decoder_outputs] + decoder_states)
```

```
def decode_sequence(input_seq):
    # Encode the input as state vectors.
    states_value = encoder_model.predict(input_seq)

    # Generate empty target sequence of length 1.
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    # Populate the first character of target sequence with the start character.
    target_seq[0, 0, target_token_index['\t']] = 1.

    # Sampling loop for a batch of sequences
    # (to simplify, here we assume a batch of size 1).
    stop_condition = False
    decoded_sentence = ''
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict(
            [target_seq] + states_value)

        # Sample a token
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_char = reverse_target_char_index[sampled_token_index]
        decoded_sentence += sampled_char

        # Exit condition: either hit max length
        # or find stop character.
        if (sampled_char == '\n' or
           len(decoded_sentence) > max_decoder_seq_length):
            stop_condition = True

        # Update the target sequence (of length 1).
        target_seq = np.zeros((1, 1, num_decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.

        # Update states
        states_value = [h, c]

    return decoded_sentence
```

```python
def model_final(input_shape, output_sequence_length, english_vocab_size, french_vocab_size):
    """
    Build and train a model that incorporates embedding, encoder-decoder, and bidirectional RNN on x and y
    :param input_shape: Tuple of input shape
    :param output_sequence_length: Length of output sequence
    :param english_vocab_size: Number of unique English words in the dataset
    :param french_vocab_size: Number of unique French words in the dataset
    :return: Keras model built, but not trained
    """

    learning_rate=5e-3

    model=Sequential()
    model.add(Embedding(english_vocab_size,256,
                        input_length=input_shape[1]))

    model.add(Bidirectional(LSTM(256),))

    model.add(RepeatVector(output_sequence_length))

    model.add(Bidirectional(LSTM(256,return_sequences=True)))

    model.add(TimeDistributed(Dense(french_vocab_size,
                                activation='softmax')))

    model.compile(loss=sparse_categorical_crossentropy,
                  optimizer=Adam(learning_rate),
                  metrics=['accuracy'])

    print(model.summary())
    return model
```

Nous obtenons de bons résultats à la fin :