Royaume du Maroc
Haut-Commissariat au Plan (HCP)
Ecole des Sciences de l'Information (ESI)

Rapport de Projet Système Expert :

Le Développement d'un Chatbot basé sur un Réseau des Neurones Récurrent : Modèle Seq2Seq

Amine Salim

Groupe A

47

Table des matières :

Introduction :

Pour se familiariser avec le *Deep Learning* et les *Réseau des Neurones Récurrents,* nous donnons les définitions suivantes :

Le *deep Learning* s'appuie sur un réseau de neurones artificiels s'inspirant du cerveau humain. Ce réseau est composé de dizaines voire de centaines de « couches » de neurones, chacune recevant et interprétant les informations de la couche précédente. Le système apprendra par exemple à reconnaître les lettres avant de s'attaquer aux mots dans un texte, ou détermine s'il y a un visage sur une photo avant de découvrir de quelle personne il s'agit, à chaque étape, les « mauvaises » réponses sont éliminées et renvoyées vers les niveaux en amont pour ajuster le modèle mathématique. Au fur et à mesure, le programme réorganise les informations en blocs plus complexes. Lorsque ce modèle est par la suite appliqué à d'autres cas, il est normalement capable de reconnaître un chat sans que personne ne lui ait jamais indiqué qu'il n'a jamais appris le concept de chat. Les données de départ sont essentielles : plus le système accumule d'expériences différentes, plus il sera performant.

Les réseaux récurrents (ou RNN pour Recurrent Neural Networks) sont des réseaux de neurones dans lesquels l'information peut se propager dans les deux sens, y compris des couches profondes aux premières couches. En cela, ils sont plus proches du vrai fonctionnement du système nerveux, qui n'est pas à sens unique. Ces réseaux possèdent des connexions récurrentes au sens où elles conservent des informations en mémoire : ils peuvent prendre en compte à un instant t un certain nombre d'états passés. Pour cette raison, les RNNs sont particulièrement adaptés aux applications faisant intervenir le contexte, et plus particulièrement au traitement des séquences temporelles comme l'apprentissage et la génération de signaux, c'est à dire quand les données forment une suite et ne sont pas indépendantes les unes des autres. Néanmoins, pour les applications faisant intervenir de longs écarts temporels (typiquement la classification de séquences vidéo), cette « mémoire à court-terme » n'est pas suffisante. En effet, les RNNs « classiques » (réseaux de neurones récurrents simples ou Vanilla RNNs) ne sont capables de mémoriser que le passé dit proche, et commencent à « oublier » au bout d'une cinquantaine d'itérations environ. Ce transfert d'information à double sens rend leur entrainement beaucoup

plus compliqué, et ce n'est que récemment que des méthodes efficaces ont été mises au point comme les LSTM (Long Short Term Memory). Ces réseaux à large « mémoire court-terme » ont notamment révolutionné la reconnaissance de la voix par les machines (Speech Recognition) ou la compréhension et la génération de texte (Natural Langage Processing). D'un point de vue théorique, les RNNs ont un potentiel bien plus grand que les réseaux de neurones classiques : des recherches ont montré qu'ils sont « Turing-complet » (ou Turing-complete), c'est à dire qu'ils permettent théoriquement de simuler n'importe quel algorithme.

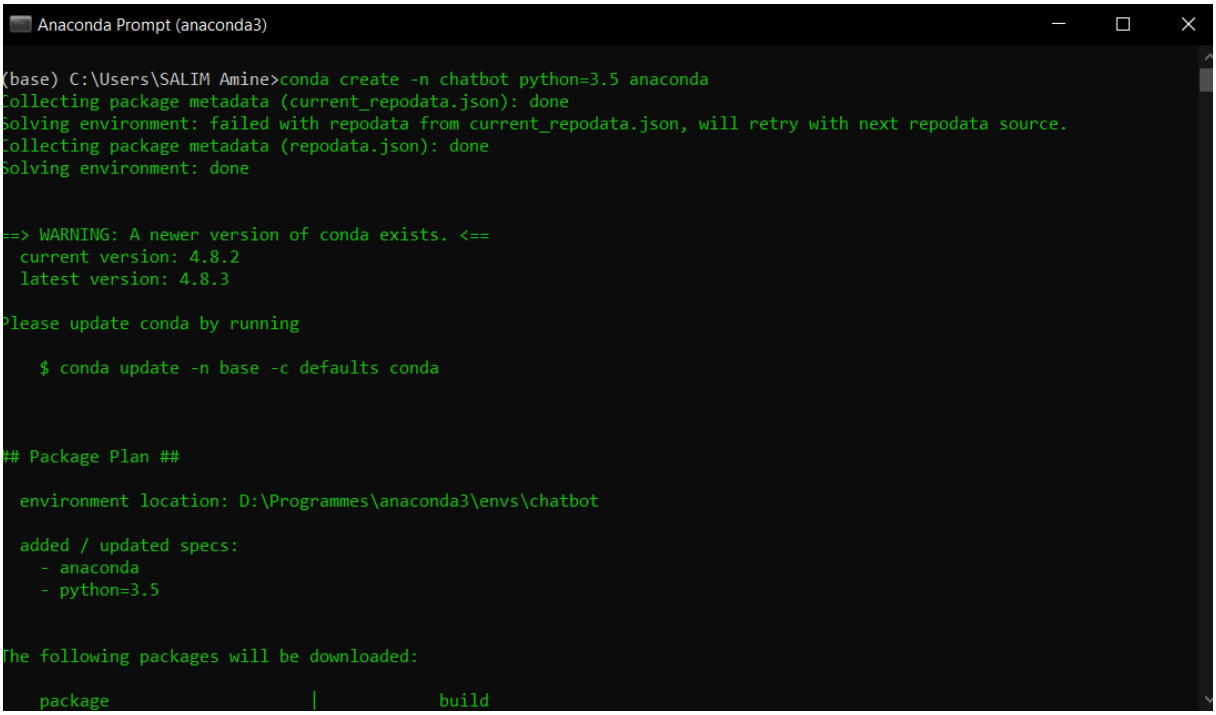
Dans le modèle *Seq2Seq,* l'encodeur traite chaque élément (*token*) de la séquence d'entrée, il compile les informations qu'il capture dans un vecteur (appelé le vecteur de contexte ou d'état caché). Après avoir traité toute la séquence d'entrée, le codeur envoie le vecteur de contexte au décodeur, qui commence à produire la séquence de sortie élément par élément. Le principe général de ce modèle est le suivant :

1. Chaque *token* en entrée (ici des mots ou ponctuation) est transformé en vecteur *d'embedding* (*word vector*) de taille identique (dimension de l'espace des *embeddings,* généralement de 200 ou 300) qui via l'encoder (RNN du backbone) va être transformé à son tour en vecteur d'état (ou vecteur de contexte).
2. Une fois que l'ensemble des *tokens* de la séquence sont ainsi transformés, un vecteur d'état de l'ensemble de la séquence est généré (state *vector*, *context vector*, *hidden vector* ou *hidden state*) ; il contient donc toutes les informations décrivant la séquence (sa taille est généralement de 256, 512 ou 1024).
3. Il est alors mis en entrée d'un décodeur (un autre RNN) qui va générer au fur et à mesure les *tokens* de la séquence de sortie.

1. Etapes Préliminaire :

Afin de réaliser notre Chatbot, nous avons besoin d'installer *Anaconda Navigator,* ensuite on crée un environnement virtuel avec *Python 3.5* en exécutant la ligne de commande suivante sous *Anaconda Prompt* :

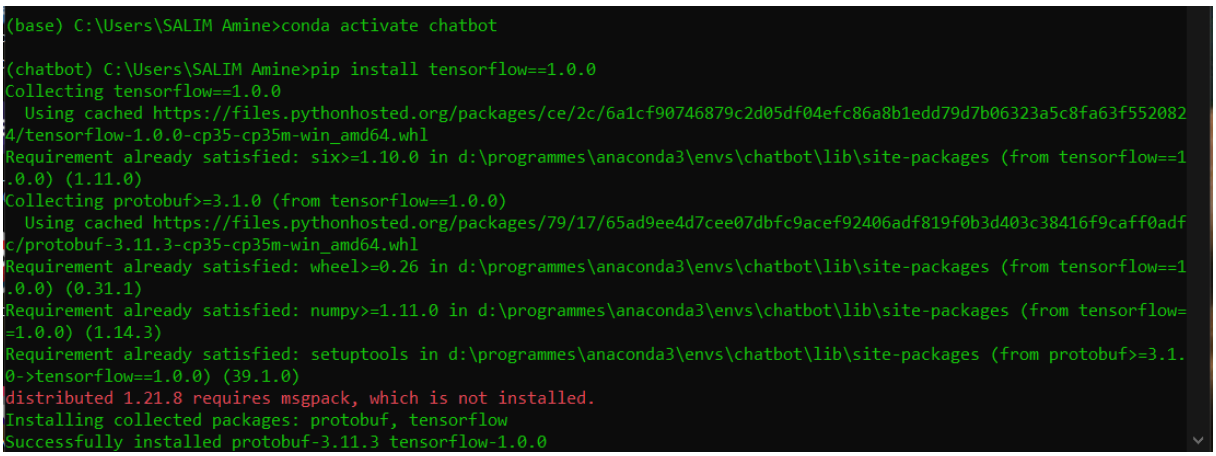
conda create -n chatbot python=3.5 anaconda



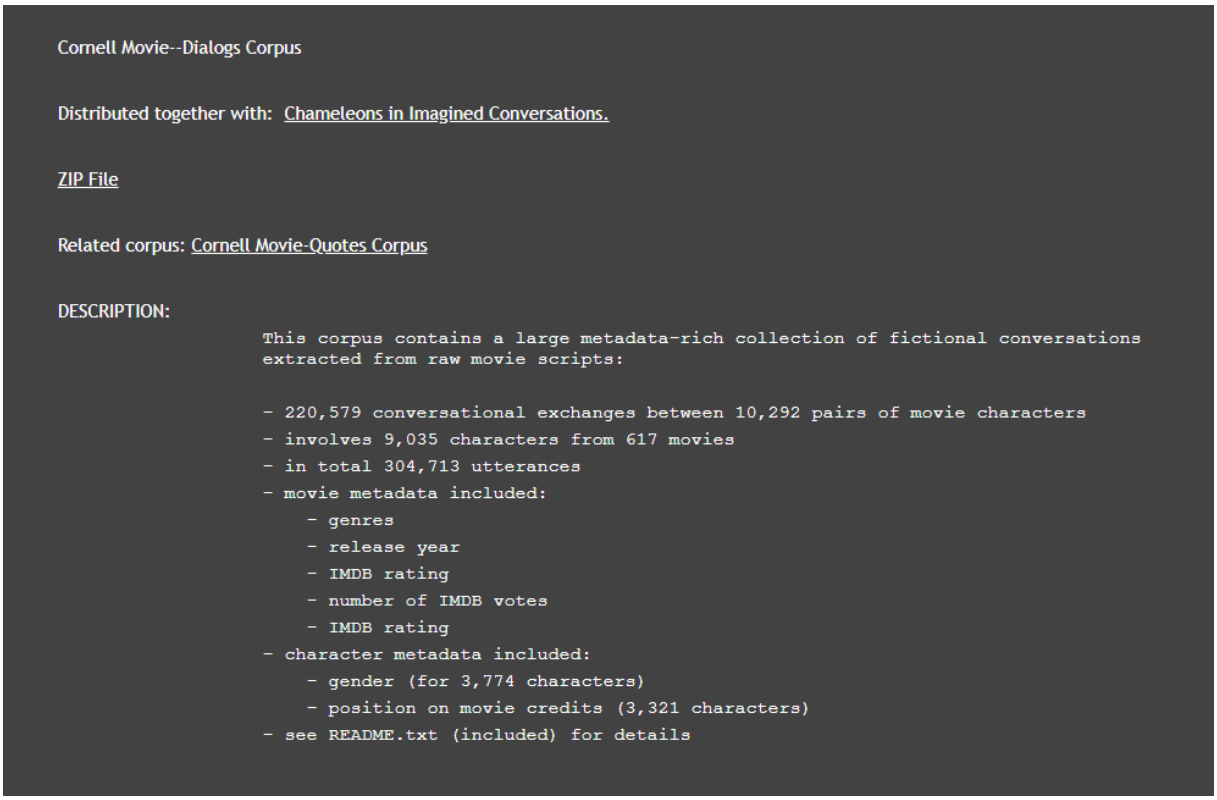
Toujours sous *Anaconda Promt,* nous activons l'environnement, et nous installons *tensorflow 1.0,* suivant les commandes ci-dessous :

activate chatbot

pip install tensorflow==1.0



Puis nous importons notre base de données nommée Cornell movie corpus, disponible sur internet, qui contient des milliers de conversations entre les acteurs de plus de 600 films.



https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html

Après la création de l'environnement et le téléchargement de la base données nous ouvrons *Spyder*, et nous commençons le *Data Preprocessing*.

2. Le Data Preprocessing :

Le *Data Preprocessing* est l'étape de préparation de la base de données afin de faciliter l'apprentissage au Chatbot, et ce que nous voulons former dans cette étape est une base de données contenant deux colonnes, une pour les *Inputs* et l'autre pour les *Outputs*.

2.1. L'importation des librairies :

Les librairies *Numpy* et *Tensorflow* nous servirons leurs fonctions et leurs modules tout au long de notre développement et surtout dans l'étape du *Training* du Chatbot.

La librairie *re* nous permettra de nettoyer le texte contenu dans la base de données pour le simplifier et le clarifier.

La librairie *Time* nous donnera les durées de chaque *Training* du Chatbot.

```
In [1]: import numpy as np
   ...: import tensorflow as tf
   ...: import re
   ...: import time
```

2.2. L'importation de la base de données :

Nous importons la base données *movie_lines* sous le nom *lines*, et pour éviter le problème d'encodage, en d'autres termes pour importer la base de données sans que le texte soit modifié, on précise l'encodage *utf-8*, en outre, nous ignorons les erreurs que le système peut générer même si nous perdrons quelques lignes de la base données mais cela n'affectera pas la construction de notre Chatbot.

Le *Read* permet de lire la base de données, et nous séparons le contenu par le saut de ligne en utilisant *split* par \n.

Ensuite, nous utilisons la même syntaxe pour importer la deuxième base *movie_conversations*.

lines	list	304714	['L1045 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ They do not!', 'L ...
conversations	list	83098	['u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L194', 'L195', 'L196', 'L197']', ...

2.3. La création d'un dictionnaire qui relie chaque ligne à un identifiant :

On nomme ce dictionnaire *id2line*, ensuite nous retirons les caractères de séparation utilisés dans la base de données ' +++$+++ ', puis nous ne prenons que les lignes ayant une longueur égale à 5.

Maintenant nous créons le dictionnaire en prenant l'identifiant qui est le premier élément (index 0) de la ligne, et la valeur qui est le dernier élément (index 4).

id2line	dict	304713	{'L626372':'The Joe Boys in 1980...!', 'L345155':'Go to hell.', 'L1563 ...

2.4. La création d'une liste contenante toutes les conversations :

Nous avons déjà une liste des conversations mais celle-ci contient les *metadata*, mais nous ne devons garder que ce dont nous avons besoin pour le *training* du Chatbot.

Nous prenons les identifiant (L…), et nous retirons les caractères non nécessaires, en plus, la dernière colonne d'index -1 de la base de données est vide, elle sera enlevée, et le signe '_' avant la variable pour préciser qu'il s'agit d'une variable temporaire.

Nous prendre le dernier élément de la ligne, pour cela nous séparons les éléments de chaque ligne par (' +++$+++ '), puis nous le dernier élément de la ligne, qui est une liste maintenant, par l'index [-1], ensuite nous enlevons les caractères « ' » (remplacé par rien), « [» (le premier élément), «] » (le dernier élément) et les espaces sont également remplacés par rien.

Finalement, nous ajoutons la liste que nous venons de former, dont les éléments sont séparés par «, », à la liste globale des conversations par append.

conversations_ids	list	83097	[['L194', 'L195', 'L196', 'L197'], ['L198', 'L199'], ['L200', 'L201', ...

2.5. La séparation des questions (inputs) et des réponses (targets) :

Nous créons deux listes, la première pour les questions et la deuxième pour les réponses, mais elles doivent avoir la même taille, parce que chaque réponse d'index i devrai être la réponse à la question du même index i, pour cela les deux listes doivent être alignées.

Nous savons que chaque élément de la liste *conversations_ids* et celui qui le suit sont respectivement la question et sa réponse.

Chaque élément de la liste *conversations_ids* et celui qui le suit sont respectivement la question et sa réponse.

D'abords, nous introduisons deux listes *questions* et *answers*, et ensuite, nous ajoutons à la liste *questions* le texte formant les questions (index i) en utilisant le dictionnaire *id2line*.

La même syntaxe pour les réponses *answers* (index i+1), mais cette fois-ci nous agissons sur la liste *answers*.

answers	list	221616	['Well, I thought we'd start with pronunciation, if that's okay with y ...

questions	list	221616	['Can we make this quick? Roxanne Korrine and Andrew Barrett are havi ...

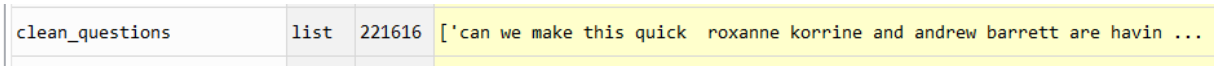
2.6. Une simplification primaire du texte :

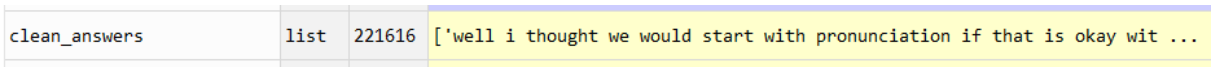
Nous introduisons une fonction *clean_text* qui transforme tout le texte en minuscule, ensuite, elle supprime les apostrophes, à titre d'exemple « *I'm* » devient « *I am* », et finalement, elle supprime les caractères spéciaux.

2.7. La simplification des questions et des answers :

Nous introduisons une nouvelle liste *clean_questions*, et ensuite nous appliquons la fonction *clean_text*.

La même syntaxe sera appliquée pour les *answers*.



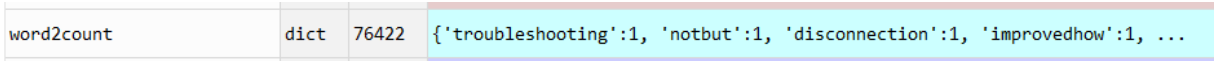


2.8. La création d'un dictionnaire qui relie chaque mot avec son nombre d'occurrence :

Ce dictionnaire a pour objectif de supprimer les mots qui apparaissent de temps en temps, plus précisément, les mots constituant moins de cinq pour cent de l'ensemble des mots de la base de données.

Le dictionnaire *word2count* reliera chaque mot avec le nombre d'occurrence qui le correspond.

On commence par *questions*, si le mot *[word]* est rencontré pour la première fois c'est-à-dire il n'appartient pas encore au dictionnaire nous l'affectons le nombre 1, sinon on l'incrémente par 1. Ensuite nous faisons exactement la même chose pour *answers*.



2.9. La création de deux dictionnaires qui relient chaque mot de questions et chaque mot de answers à un entier unique :

Dans ces dictionnaires nommés *questionswords2int* et *answerswords2int*, nous avons une condition *if* qui vérifie si le nombre d'occurrence de chaque mot est supérieur d'un seuil de 5 pour cent (*threshold_questions*, *threshold_answers*), et si

c’est le cas le mot s’ajoute au dictionnaire comme une clé, sinon il ne sera pas considéré.

questionswords2int	dict	8825	{'margaret':2213, 'papa':4373, 'bringin'':1125, 'lisa':7371, 'warm':43 ...

answerswords2int	dict	8825	{'margaret':2213, 'papa':4373, 'bringin'':1125, 'lisa':7371, 'warm':43 ...

2.10. L’ajout des indications au deux dictionnaires :

Ces indications sont utiles dans le modèle *seq2seq* que nous allons construire par la suite, et elles sont comme suit :

- **SOS** : Start of string, le début de la chaine de caractères ;
- **EOS** : End of string, la fin de la chaine de caractères ;
- **PAD** : qui est très important pour notre modèle parce que les *training data* et les *sequences* doivent avoir la même langueur, et nous allons l’introduire dans une position vide ;
- **OUT** : correspond à tous les mots qui ont été filtré par les deux dictionnaires précédents, en conséquence, tous ces mots seront remplacés par OUT.

Maintenant, pour que l’entier lié à chacune de ces indications soit différent des entiers liés aux mots, nous allons utiliser l’entier *len(questionswords2int) + 1*, ensuite nous appliquons le même processus pour *answerswords2int*.

Ensuite on crée l’inverse du dictionnaire *answerswords2int*, car nous avons besoin de la liaison inverse des entiers aux mots des *answers*, dans l’implémentation. Ce dictionnaire sera nommé *answersint2words*.

answerswords2int	dict	8825	{'margaret':2213, 'papa':4373, 'bringin'':1125, 'lisa':7371, 'warm':43 ...

2.11. L’utilisation de ‘EOS’ :

Nous agissons sur les réponses simplifiées *cleaned_answers* en utilisant la boucle *for*, nous ajoutons ‘*EOS’* à la fin de chaque *answer* et un espace avant chaque *EOS*, afin qu’elle ne soit pas confondue avec le dernier mot d’*answer*.

2.12. La traduction des questions et answers en entiers et remplacer les mots filtrés par OUT :

Nous trions les questions et les *answers* par leurs longueurs, et pour cela on doit créer deux listes contenant la traduction de chacune de questions et *answers*. Ce tri est important pour l'optimisation du *training* du Chatbot.

En construisant ces listes, nous remplaçons tous les mots filtrés par l'indication '*OUT'*, en utilisant une condition *if* qui vérifie si le mot est filtré, et dans ce cas il sera remplacé par l'indication '*OUT'*.

Ces opérations auront effet sur les questions et les *answers* pour aboutir finalement à deux listes *questions_into_int* et *answers_into_int*.

answers_into_int	list	221616	[[3682, 5041, 4511, 4634, 7759, ...], [5067, 2219, 8824, 7089, 8824, ...

questions_into_int	list	221616	[[3879, 4634, 2603, 4573, 1021, ...], [3682, 5041, 4511, 4634, 7759, ...

2.13. Le tri des questions et des answers par la longueur des questions :

L'objectif de ce tri est de faciliter et optimiser le *training* du Chatbot.

Nous introduisons deux nouvelles listes *sorted_clean_questions* et *sorted_clean_answers*, et la variable boucle va de 1 qui est la plus petite langueur d'une question ou une *answer*, elle s'incrémente, et nous allons l'arrêter à un certain nombre parce que nous ne voulons pas considérer les questions qui sont très longues parce que celles-ci posent des problèmes pour le Chatbot au niveau de *training*.

sorted_clean_answers	list	203949	[[2219, 8705, 5889, 4863, 5041, ...], [8824, 6738, 5285, 947, 5765, ...

sorted_clean_questions	list	203949	[[4863], [4638], [1960], [355], [1594], [7644], [2786], [7644], [6142] ...

3. La Construction du Modèle Seq2Seq :

Dans cette partie nous allons utiliser la librairie *tensorflow*, pour faire appel à plusieurs fonctions, qui vont nous permettre de définir l'architecture du modèle *Seq2Seq*.

3.1. La création des placeholders pour les inputs et les targets :

Dans *tensorflow* toutes les variables sont utilisées dans des *tensors*, les *tensors* sont des matrices avancées que celles de *numpy*, et toutes les variables utilisées dans les *tensors* doivent être définie dans ce qu'on appelle *tensorflow placeholders*.

La fonction appelée *model_inputs* crée des *placeholders* pour les inputs et des *placeholders* pour les *targets*, ensuite elle ajoute un taux d'apprentissage (*learning_rate*), et d'autres paramètres hybrides.

En bref, nous créerons les *placeholders* afin que nous puissions utiliser ces variables dans le futur *training* du Chatbot.

Nous importerons la fonction *placeholder* de la librairie *tensorflow*, et nous créerons les *placeholders* pour les *inputs* en spécifiant que la taille de la matrice est égale à deux, ensuite nous appliquons le même processus pour les *targets*. Puis nous introduisons le *learning_rate,* et le *keep_prob* (expliqué dans les prochains paragraphes).

Finalement la fonction retourne les *inputs*, les *targets*, le *learning_rate* et le *keep_prob*.

3.2. Le preprocessing des targets :

Le preprocessing des *targets* est nécessaire car le décodeur ne les accepte qu'à sous un format spécifique, le décodeur prend les *targets* mais pour que ces derniers soient acceptés dans le *neural network* ils doivent être de format *twofold*. Premièrement, les targets doivent être groupés par lot de targets (*batch of targets*), Nous fixons la taille de chaque lot à 10 targets. Chaque élément dans les réponses (*answers*) dans le lot des *targets* doit commencer par l'entier affecté à '*SOS'*. Nous prenons tous les éléments de chaque *answer* sauf de dernier élément d'index -1, et ensuite nous ajoutons notre entier *SOS* par concaténation.

3.3. La création des layers de l'encodeur réseau des neurones récurrents :

Tout d'abord, les *layers* sont des composantes intermédiaires qui se situent entre les *inputs* et les *outputs*, et dans lesquelles on parle de la *long-short term memory (LSTM)* de *tensorflow*.

Dans cette phase on va inclure la classe *long-short term memory* (LTSM) de *tensorflow*.

Nous avons dans cette étape comme arguments le *rnn_inputs* pour les *model_inputs*, le *rnn_size* est le nombre des *inputs tensors* de l'encodeur que nous sommes en train est de construire, le *num_layers*, *keep_prob* pour appliquer une régularisation de taux d'abandon à notre *LSTM* afin d'améliorer l'exactitude de notre Chatbot, *sequence_lentgh* est la liste des longueurs de chaque question dans les lots (*batchs*) construis précédemment.

Pour créer la *LSTM*, on fait appel à la classe *BasicLSTMCell* de la librairie *tensorflow*, qui prend l'argument *rnn_size*. La technique de *dropout* désactive un certain pourcentage des neurones durant les itérations du *training* du Chatbot, *lstm_dropout* utilise la classe *DropoutWrapper* qui prend deux arguments *lstm*, *input_keep_prob* qui doit être égal à la *keep_prob* précédente. *Encoder_cell* fait appel à la classe *MultiRNNCell* de *tensorflow*, et pour avoir plusieurs *layers* on multiplie le *lstm_dropout* par le nombre des *layers* qu'on veut avoir dans le *layer* de notre *RNN* (*num_layers*).

La cellule d'encodeur *encoder_cell* est composée de plusieurs *lstm layers*, et nous avons appliqué la technique de *dropout* pour chacun de ces *lstm layers.*

_, *encoder_state* : le « _ » pour obtenir seulement le deuxième élément retourné par la fonction RNN bidirectionnelle dynamique de *tensorflow*, cela créer une version dynamique du RNN bidirectionnelle, cette fonction prend les *inputs* et construit deux RNN avant et arrière indépendants, mais il faut s'assurer que la taille d'input du RNN avant est exactement la même taille *d'input* de RNN arrière.

3.4. Le décodage des données du training :

La fonction *decode_training_set* prend plusieurs argument, premièrement *encoder_state* parce que notre décodeur prend l'*encoding_state* comme une partie de l'*input*, et pour cela on a besoin de *encoder_state* de la fonction précédente pour réaliser le décodage, ensuite l'argument *decoder_cell* qui est la cellule contenue

dans le décodeur RNN, puis l'argument suivant est *decoder_embedded_input* qui sert à relier les mots à des vecteur de nombres réels, le *sequence_lenght,* et le *decoding_scope* qui permettra d'envelopper (*wrap*) les variables *tensorflow, output_function* est une fonction qui retourne l'*outputs* du décodeur, *keep_prob,* et *batch_size.*

Les *attention_states* sont des matrices de dimension égale à 3 et nous les initialiserons par des 0 à l'aide de la fonction *zeroes,* cette matrice contiendra le nombre des lignes qui est le *batch_size* comme première dimension, la deuxième dimension est le nombre des colonnes qui est égale à 1 car on n'aura besoin que d'une seule colonne et la troisième dimension est *decoder_cell.output_size.*

L'étape suivante est de préparer les clés d'attention *attention_keys,* les valeurs d'attention *attention_values,* la fonction du score d'attention *attention_score_function* et la fonction de construction d'attention *attention_construct_function,* en faisant appel à un seul sous-module *tensorflow* qui est le *seq2seq*. Nous choisissons comme *attention_option* l'option *'bahdanau'.*

Nous introduirons une autre variable *attention_traning_function* dont nous utiliserons la fonction *attention_decoder_fn_train* qui décodera le training set.

L'étape finale est d'appliquer un *dropout* à notre *decoder_output* en utilisant la fonction *dropout.*

3.5. Le décodage des données de test et de validation :

Dans cette partie nous introduisons une nouvelle fonction similaire à la fonction précédente mais pour un autre type de données qui sont les données du test et les données de validation, ces donnéess ne serons pas utilisés dans le training, mais ce décodage nous sera utile pour tester notre Chatbot.

Nous allons utiliser une autre fonction de *tensorflow* qui est *attention_ decoder_fn_inference,* cette fonction donnera à notre chatbot une logique de penser et fonctionner.

3.6. La création du décodeur réseau de neurones récurrent :

Dans cette phase nous allons construire un autre réseau de neurones récurrent, elle est similaire à l'étape de création de l'encodeur *RNN,*

Les arguments, *decoder_embedded_input, decoder_embeddings_matrix, encoder_sate, num_words, sequence_lenght, rnn_size, num_layers, word2int, keep_prob, batch_size.*

Nous introduisons un nouveau concept ; les poids *weights* des neurones qui sont complétement connectés, et on va les initialiser par *trancated_normal_initializer* qui va générer une distribution normal tronquée des poids, cette fonction prend un seul argument, et on va choisir une déviation standard de 0,1 par *stddev=0,1*, ensuite les *biases* qui sont initialiser par des zéros.

La fonction *fully_connected* prend comme arguments les *inputs* qui sont la variable x, *num_words* le nombre de la totalité des mots et d'autres arguments, et nous avons la variable *test_predictions,* qui utilise la fonction *decoder_test_set.*

3.7. La construction du modèle Seq2Seq :

La construction uni l'encodeur RNN et le décodeur RNN, et cette union est le cerveau du notre Chatbot.

Les *inputs* sont les *questions* du corpus *movie Cornell* qui serons après le *training*, les questions que nous allons poser à notre Chatbot lorsqu'on lui parle, les *targets* sont les réponses, et les arguments qu'on a vus précédemment.

Dans cette phase nous utilisons les fonctions ; *embed_sequence, encoder_rnn, preprocessing_targets*.

La classe *Variables* prend les dimensions de la variable *decoder_embeddings_matrix*, et la 1ère dimension est le nombre de ligne, la 2ème est le nombre des colonnes. Cette matrice est remplie par des nombres aléatoire compris entre 0 et 1.

Dans les étapes qui suivent nous allons construire et faire le *training* du cerveau de notre Chatbot.

4. Le training du modèle Seq2Seq :

L'objectif de cette partie est de ramener le Chatbot à devenir intelligent et capable de parler avec nous.

4.1. Le Paramétrage des Hyperparamètres :

Premièrement le nombre des *epochs* ; l'*epoch* est le processus permettant d'avoir les lots des *inputs,* ensuite avant propager ces derniers dans les encodeurs pour avoir les *encoder_states,* puis avant propager ces derniers avec les *targets* dans le décodeur RNN pour avoir les *outputs_scores* et les *answers* comme prévu, puis arrière propager ces derniers par les *outputs* et les *targets* dans le réseau de neurones et mettre à jour les poids afin d'améliorer l'aptitude du Chatbot de chatter comme un humain.

Pour *epochs* on choisit 100, si l'ordinateur prend beaucoup de temps à faire le *training* du chatbot il sera mieux de prendre la valeur 50.

Pour le *batch_size* on prend 64, encore une fois si on rencontre des problèmes il vaut mieux prendre 128 comme *batch_size*.

Pour l'*encoding_embedding_size* on choisit 512, qui est le nombre des colonnes dans la *embeddings_matrix* dont chaque ligne corresponds à un *token* du corpus des *questions*. La même valeur pour le *decoding_embedding_size*.

Un des plus importants paramètres est le *learning_rate*, il ne faut pas qu'il soit très grand sinon le chatbot va apprendre rapidement par conséquence il ne va pas bien parler, et si le *learning_rate* est très petit le Chatbot va prendre des années pour apprendre à bien parler, alors on a choisi le valeur 0,01.

Le *learnig_rate_decay* qui est le pourcentage qu'avec lequel le *learning_rate* diminue de chaque itération de *training* à une autre, et on va choisir une valeur commune qui est 0.9 (90%).

Le *min_learning_rate* égale à 0.0001, ce paramètre est utilisé pour arrêter le training quand le *learning_rate* atteint la valeur 0.0001.

Le *keep_probability* on choisit la valeur 0,5 (50%), et c'est la valeur recommandée par Jeffrey Hinton dans son article intitulé « *A Simple Way to Prevent Neural Networks from Overfitting* ».

4.2. La définition d'une session :

Dans cette partie on va définir une session *tensorflow* dans la laquelle s'exécute tout le *tensorflow training.*

Pour ouvrir une session dans *tensorflow,* on crée un objet de la classe session interactive, et cet objet sera notre session.

Avant de créer notre objet on doit réinitialiser les graphes de *tensorflow* pour s'assurer que le graph est prêt pour le *training* et cela en utilisant la fonction *reset_default_graph*, ensuite on définit la session par la fonction *InteractiveSession.*

4.3. Le chargement des inputs du modèle Seq2Seq :

Dans cette partie nous chargeons les inputs du modèle *seq2seq,* pour faire ce chargement on utilise un outil qu'on a déjà construit dans la partie **La Construction du modèle Seq2Seq**, qui est la fonction *input_model* qui retourne les *inputs,* les *targets* et la *keep_prob*.

4.4. Le paramétrage de la longueur de la séquence :

Dans cette phase on paramètre la longueur de la séquence à une longueur maximal.

Pour créer la variable *sequence_length* on utilise la fonction *placeholder_with_default,* qui prend comme argument la longueur maximale qui est égale à 25, le deuxième argument est la forme *shape* elle ne nous concerne pas pour le moment ainsi on la donne la valeur *None,* le troisième argument est tout simplement le nom de la séquence.

4.5. Définir la forme (shape) du tensor des inputs :

Avant d'utiliser la fonction du modèle *seq2seq* pour avoir les prédictions du *training* et celles du *test* on doit avoir la forme *shape* du *tensor* des *inputs*, pour cela on utilise la fonction *ones*, qui créer un *tensor* contenant des 1, et les dimensions de ce *tensor* sont la forme *shape* qu'on cherche.

4.6. Les prédictions du training et du test :

Ce n'est pas dans cette partie qu'on va commencer le training, ce que nous allons faire ici est avoir les prédictions du *training* et du *test* lorsqu'on entre les *inputs,* les *targets,* le *learning_rate,* et la *keep_prob* dans le réseau de neurones.

On utilise la fonction du modèle *seq2seq,* et on introduit deux nouvelles variables *training_predictions* et *test_predictions* à ne pas confondre avec les variables locales de la fonction du modèle *seq2seq* qu'on a défini antérieurement.

4.7. Le Paramétrage de l'erreur de perte, de l'optimiseur, et de l'écrêtage de gradient :

L'écrêtage de gradient est une opération qui restreint le gradient dans le graphe entre une valeur minimale at autre maximale pour éviter le problème d'explosion du gradient ou sa disparition, alors ce que nous allons faire est d'applique l'écrêtage de gradient à notre optimiseur.

L'erreur de perte sera basée sur l'entropie croisé qui est la plus pertinente dans le *deep natural language processing,*

L'optimiseur qu'on va utiliser est l'optimiseur d'Adam.

4.8. Le remplissage des séquences par le token 'PAD' :

Le but de cette partie est d'attribuer la même longueur qu'a la question à la réponse, par exemple :

La question : ['Who', 'are', 'you']

La réponse : [<SOS>, 'I', 'am', 'a', 'bot', '.', <EOS>]

Deviendra ;

La question : ['Who', 'are', 'you', <PAD>, <PAD>, <PAD>, <PAD>]

La réponse : [<SOS>, 'I', 'am', 'a', 'bot', '.', <EOS>, <PAD>]

On introduit une fonction *apply_padding* qui agit sur le *batch_of_sequences,* et *word2int.*

4.9. La division de données en lots de questions et de answers :

On introduit une nouvelle fonction *split_into_batches*, qui agit sur les *questions*, les *answers*, et le *batch_size*. Cette fonction nous permet de diviser les données en jouant avec les index.

Ensuite, on applique la fonction *apply_padding.*

4.10. La division des questions et des answers en données de validation et données de training :

Cette étape va produire des données pour les *training questions*, des données pour les *training answers*, des données pour les *questions* de validation avec un pourcentage de 15% de la totalité des questions et des données pour les *answers* de validation.

4.11. Le training :

Afin de bien comprendre cette partie, imaginons un nouveau-né qui apprend à parler en écoutant son entourage, la différence ici c'est que le nouveau-né est un réseau des neurones récurrent, et l'entourage est l'ensemble des *questions* et des *answers*.

Nous prenons le *batch_index_training_loss*=100 pour vérifier la perte du *training* sur chaque chaque 100 lots, et nous prendre la moitié des lots pour vérifier la perte de validation.

Le *training_loss_error* est utilisé pour calculer la somme des pertes durant les *trainings* sur 100 lots, et on va l'initialiser par 0 et il sera incrémenté à chaque itération.

On crée une liste de validation de perte des erreurs, parce que nous allons utiliser la technique de de l'arrêt précoce ; une technique qui consiste à vérifier si nous avons atteint une perte inférieure au minimum de toutes les pertes que nous avons récupéré dans chaque itération, et toutes ces pertes serons déposé dans notre liste.

On exécute la session, et ensuite on démarre notre boucle qui se charge entièrement du *training*, nous utilisons la librairie *Time*, avec les variables *starting_time* et *ending_time* qui vont mesurer la durée du *training* sur chaque 100 lots, et l'affecter à *batch_time*, et à chaque fois qu'on acheve le *training* sur 100 lots, on affiche les

informations relatives à celui-ci, on cite le durée et l'ordre du lot par rapport à l'ensemble des lots.

La première condition *if* permet d'afficher les informations du *training* sur chaque 100 lots, et la deuxième affiche l'erreur de perte de validation et la durée de validation du lot deux fois à chaque *epoch* puisque nous avons pris la moitié des lots pour la validation, mais la *keep_prob* ici doit être égale à 1, parce que les neurones doivent être toujours présent, et on a pris la *keep_prob* antérieur égale à 50% juste pour améliorer le *training*,

Ensuite nous appliquons des diminutions à notre *learning_rate*, en le multipliant par le *learning_rate_decay*, ensuite si le *learning_rate* devient inférieur à son minimum on le réaffecter le *min_learning_rate*, que nous avons déjà paramétrer.

Puis on remplit notre liste par la dernière moyenne des pertes de validation trouvée, ensuite on vérifie si cette moyenne est inférieur de toutes le pertes de validation qu'on a trouvé, et si c'est le cas alors le Chatbot améliore son aptitude de parler et affiche « *I feel much better now, I really improved my speaking !* », par la suite on doit réinitialiser la variable *early_stopping_check* à 0, cette opération se répète à chaque fois qu'il y a une amélioration, et on enregistre notre modèle en utilisant la méthode *saver* de *tensorflow*. Mais si ce n'est pas le cas on affiche « *Oh, I feel terrible, I do not speak better, I am sorry, I need more practice* », et on incrémente le *early_stopping_check* par 1, et si ce dernier atteint 1000 on arrête la boucle et on affiche « *This is the best I can do, I cannot improve myself any longer* », et ensuite on arrête la boucle globale, et le Chatbot affiche « *Game Over Chico !* ».

A savoir aussi que cette partie du code prend des jours pour s'exécuter, pour moi j'ai laissé le *training* pendant 7 jours d'exécution afin d'avoir des poids acceptables, mais pour avoir des poids pertinents, il sera mieux de laisser le *training* s'exécuter pour 20 jours ou plus.

5. Le lancement du Chatbot :

Cette partie nous permet de chatter avec notre Chatbot en se basant sur les poids que nous avons obtenu après le training,

Premièrement, nous importons les poids que nous avons nommé *'chatbot_weights.ckpt',* en introduisant la variable *checkpoint* qui contient le chemin à ceux-ci.

Ensuite on ouvre une session avec la variable *session* en utilisant *tensorflow*. Cette session est un objet de la classe session interactive que nous avons déjà utilisé dans le *training*, puis on exécute la session.

Deuxièmement, on connecte les poids que nous venons d'importer à notre session, en créant un objet *saver* et à travers la méthode *restore*.

Troisièmement, on crée la fonction *convert_string2int* qui va convertira les *questions* qui sont maintenant des chaînes de caractères, en une liste de leurs entiers d'encodage, cette fonction prend deux arguments *question* et *word2int* qui est un dictionnaire. Avant de convertir les *questions* on les simplifie d'abord en utilisant la fonction *clean_text,* puis on convertit les *questions* en utilisant le dictionnaire et si le mot ne figure pas dans le dictionnaire la fonction retourne l'indicateur '*OUT*'.

Dernièrement, on paramètre le chat avec le Chatbot, d'une part le *while(true)* exécute le chat jusqu'à ce qu'on applique le *break* par « *See you later* », d'autre part, nous allons établir les *questions* de sorte que serons entrées dans le réseau des neurones pour avoir les réponses souhaitées. Cela étant, on applique la fonction *convert_string2int*, ensuite on applique le remplissage par l'indicateur '*PAD'* afin que les *questions* et le *answers* aient la même longueur que nous avons déjà fixer à 20. L'étape suivante est de transformer les *questions* en des lots (*batchs*), parce que les réseaux des neurones n'acceptent que les *inputs* qui sont regroupé en des lots, la variable *fake_batch* se charge de ce processus en créant une matrice à deux dimensions, les lignes sont le *batch_size* puisque chaque ligne du *batch* correspond à une question, et les colonnes sont la longueur 20. Maintenant, on prépare les *outputs*, et ceci en établissant les espaces entre les mots, remplacer le '*i*' par le '*I*' et remplacer l'indicateur '*EOS'* par *'.',* et le '*OUT'* par 'out', finalement on termine la boucle si le Chatbot rencontre dans sa réponse le '.'.

Le Chatbot est prêt pour chatter.

6. Le code source :

Voire le fichier HTML.

Afin d'exécuter le Chatbot, il faut exécuter tout le code sauf la sous-partie intitulée « *training* ».

Bibliographie :

A list of cost functions used in neural networks, alongside application [en ligne], CrossValidated. 2015. https://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications

A Neural Network in 13 lines of Python: Improving our neural network by optimizing Gradient Descent [en ligne], iamtrask. 2015. https://iamtrask.github.io/2015/07/27/python-network-part2/

Neural Networks and Deep Learning [en ligne], M. Nielsen. 2015: Determination Press. http://static.latexstudio.net/article/2018/0912/neuralnetworksanddeeplearning.pdf

Learning long-term dependencies with gradient descent is difficult [en ligne]. Y. Bengio, P. Simard, P. Frasconi. 1994 : IEEE Transactions on Neural Networks. ISSN: 1045-9227.