CES Data Science 2019

Rapport de projet personnel

Génération automatique de réponses à l'aide d'un réseau de neurones récurrent

Nabil Boukraa
22 janvier 2020

Table des matières

1. Objectifs

Le projet présenté dans ce rapport a pour but de créer un modèle capable de rechercher la réponse à une question (en anglais) sur une base de données d'articles publiés sur la plate-forme anglophone de Wikipédia. Pour cela, nous nous appuierons d'une part sur un fichier *json* comprenant une base de couples question-réponse, ainsi que d'une base question-paragraphes[1] pour ces mêmes questions. Nous entraînerons séquentiellement deux réseaux neuronaux récurrents de type LSTM – dont l'implémentation est disponible dans la libraire *torch.nn* – sur la base de données SQuAD. SQuAD, pour *Stanford Question Answering Dataset*, est une base de données contenant des dizaines de milliers de couples question-réponse et question-paragraphes[2] générés par des humains sur la base d'articles publiés sur Wikipedia.

Nous combinerons deux approches afin de résoudre le problème posé :

1. Etant donné une question posée, nous utiliserons ***Document Retriever***, un moteur de recherche afin d'isoler un petit nombre d'articles dans la base de données d'articles Wikipédia, ce moteur de recherche étant basé sur la fréquence relative *TF-IDF* des mots et bi-grammes dans les articles ;

2. Un premier modèle d'apprentissage, entraîné sur les jeux de données question-paragraphes et question-réponse, et dénommé ***Document Selector***, identifiera le paragraphe parmi tous ces articles qui contiendra le plus probablement la réponse : la sortie de ce premier réseau, qui sera logiquement un *softmax*, constituera la réponse longue à la questions posée ;

3. Puis nous utiliserons le modèle d'apprentissage entraîné sur le fichier question-réponse, et dénommé ***Document Reader***, afin d'identifier, au sein des articles extraits à l'étape 1, la chaîne de caractère correspondant le plus vraisemblablement à la réponse souhaitée. Cette chaîne de caractères sera la réponse courte à la questions posée.

Pour comprendre le fonctionnement de notre modèle, et l'intérêt de sa segmentation, il nous semble important de faire les deux observations suivantes :

1. Les deux modèles *Document Selector* et *Document Reader* exploitent tous les deux les deux jeux de données question-réponse et question-paragraphes, la différence étant que :

    a. *Document Selector* s'intéresse à la présence ou non d'une réponse dans un des paragraphes de l'article, là où
    b. *Document Reader* s'intéresse aux positions les plus probables du début et de la fin de la réponse courte dans le paragraphe
    c. Comme nous l'expliquons plus bas, le rôle du *Document Selector* est de *débruiter* la sortie du *Document Reader* en lui donnant en entrée un ensemble plus pertinent de paragraphes à examiner.

2. Bien que ce soit généralement le cas, il n'est pas nécessaire que la réponse courte se trouve dans la réponse longue. En effet, comme nous le verrons plus loin, la pertinence de la réponse longue trouvée est pondérée par la pertinence de la position de début et de fin de la réponse courte que l'algorithme a pu dégager.

---

[1] Dans ce rapport, les expressions "paragraphe" et "document" seront utilisées de façon interchangeable pour désigner les chaînes de caractères commençant ou terminant par un saut de ligne dans les articles Wikipédia.
[2] La – ou les – réponse à une question posée se trouve toujours contenu dans au moins un des paragraphes associés à cette question.

L'intérêt de procéder de la sorte peut paraître clair au lecteur à l'aide de l'illustration suivante. Intéressons-nous par exemple à la question « *When was Charles de Gaulle born ?* ». Le premier paragraphe de l'article concernant Charles de Gaulle est certainement intéressant pour répondre à cette question mais elle le sera certainement beaucoup moins si la question est « *Who was the leader of the French resistance during German occupation ?* », bien que dans les deux cas, l'article fournisse bien la réponse à la question posée.

Tout se passe comme si en quelque sorte, un intervenant humain se rendait sur la page Wikipédia consacrée au général de Gaulle, lisait *en diagonale* les paragraphes de cet article[3], puis qu'il se penchait plus en détail sur les paragraphes qu'il aurait identifié comme étant les plus pertinents – c'est-à-dire les plus susceptibles de contenir la réponse à la question posée.

Cela permet (i) de réduire les chances d'extraire des passages par erreur, et (ii) (bien que cela ne se vérifie pas sur mon ordinateur) d'accélérer le temps de calcul, en appliquant le *Document Reader* à un nombre plus restreint de paragraphes. L'idée de « débruiter » la lecture des paragraphes en enrichissant le modèle d'une phase de pré-filtrage est introduite dans l'article « ***Denoising Distantly Supervised Open-Domain Question Answering*** ». Cette publication datant de 2018 a très largement servi de référence à mon projet et elle est disponible sur le site de l'université de Tsinghua[4]. Le code python l'accompagnant est également disponible en libre accès[5].

En dépit de nombreuses difficultés rencontrées lors de l'installation de PyTorch, que j'admets avoir très largement sous-estimées au moment de choisir le sujet de mon projet personnel[6], j'ai pris l'initiative de mettre à jour le code sur la base de PyTorch 1.4.0., en suivant les recommandations préconisées sur les forums spécialisés, et de repenser l'arborescence du code sur python, en incluant un *pipeline* me permettant d'évaluer la méthode sur des cas concrets. Par ailleurs, j'ai également adapté le code afin qu'il fonctionne par défaut avec spaCy – le code d'origine s'appuyant sur la librairie *CoreNLP*.

S'agissant d'un projet très ambitieux, qui demande du temps et de l'engagement pour être appréhendé dans sa globalité, **notre objectif sera de nous approprier une partie du code, d'installer la méthode et dans la mesure du possible, d'y apporter une touche personnelle.**

---

[3] Nous verrons plus loin qu'en réalité, le sélectionneur de paragraphes se contente de scanner les 30 premiers paragraphes, contre 50 dans le code d'origine publié par les équipes de recherche de *Facebook*, et cela pour des questions de limites de mémoire sur la carte graphique lors de l'apprentissage.

[4] https://nlp.csai.tsinghua.edu.cn/~lzy/publications/acl2018_qa.pdf

[5] https://github.com/thunlp/OpenQA

[6] En l'occurrence, l'acquisition d'une carte graphique compatible – nommément la GeForce RTX 2070 avec mémoire tampon de 8G – s'est avérée nécessaire, de même que l'installation d'un système d'exploitation Linux.

2. La génération automatique de réponse

La génération automatique de réponse ou *Open Question Answering* en anglais est une discipline alliant sciences des données et traitement du langage naturel, et visant à répondre à une question posée en langage naturel sur un sujet non spécifié – par opposition au *Closed-Domain Question Answering* qui porte sur un sujet défini et qui peut s'appuyer en pratique sur une base de connaissance structurée, une base de données relationnelle, une ontologie, etc. pour les requêter.

Nos travaux reposent sur trois jeux de données : (1) les données collectées sur le site anglophone de Wikipedia[7] que nous utiliserons comme base de connaissance ; et (2) le jeu de données SQuAD qui sera notre principale source pour entraîner et évaluer *Document Selector* puis *Document Reader*. Nous disposons également d'autres bases de données de même nature que SQuAD, notamment Quaqar-T et SearchQA, mais celles-ci ont été générées automatiquement sur la base de sources autres que Wikipedia.

Il est important de noter qu'au moment d'évaluer la méthode, les couples question-paragraphes de la base SQuAD ne sont plus disponibles. Il est demandé au modèle de répondre à une question en se basant sur la base Wikipedia dans son intégralité.

2.1. Base de connaissance

Un des principaux défis de cette méthode est que notre unique source d'information est Wikipedia, à savoir une base de connaissance conçue pour des humains. D'autres bases de connaissances existent, qui permettent d'archiver de l'information sois forme structurée dans le but de faciliter son utilisation ultérieure par un ordinateur. Or contrairement à Wikipedia, ces bases de connaissances ne bénéficient pas nécessairement des enrichissements et mises à jour constants apportés par une très large communauté d'utilisateurs.

Un autre avantage de traiter ce problème dans sa forme la plus générique est que la méthode de résolution peut s'appliquer à un vaste éventail de corpus : articles de presses, livres, etc. Il est également important de noter que la méthode testée ici ne tient pas compte des références explicites pouvant exister entre articles et ne s'intéresse qu'au texte brut.

2.2. L'encodage syntaxique

Afin de donner en entrée des réseaux récurrents des données numériques, les mots des questions, des réponses et des documents ont tous été convertis en leur vecteur *GloVe*, diminutif pour « ***Global Vectors for Word Representation*** ». Nous avons utilisé la représentation *glove.840B.300d* contenant 2.2 millions de mots projetés dans un espace vectoriel à 300 dimensions.

Le choix de cette représentation va au-delà de l'approche *TF-IDF* qui repose sur les caractères, puisqu'elle traduit également le sens des mots employés. Les mots anglais « *car* » et « *vehicle* » ont certes des acceptions différentes mais sont très proches du point de vue du sens. En revanche, ils seraient traités très différemment selon la métrique *TF-IDF*.

Même si cela n'est pas dit explicitement dans la publication, il me semble que le *Document Selector* bénéficie de ce principe au moment d'affiner les premiers résultats obtenus à l'aide du moteur de recherche. Je pense également que l'usage des encodages syntaxiques permet de traiter un problème aussi complexe sans qu'il soit nécessaire d'avoir recours à une base de connaissances.

[7] https://dumps.wikimedia.org/enwiki/latest/

3. Description du modèle

3.1. Document Retriever

Comme expliqué précédemment, *Document Retriever* réalise une présélection des articles à l'aide d'une simple pondération *TF-IDF*. Les questions et les articles sont comparés sur la base de leur représentation vectorielle dans cet espace. L'ordre des mots n'est pas pris en compte – si ce n'est indirectement au travers de l'utilisation de bi-grammes. Afin d'accélérer la recherche et de sauver de la mémoire, une méthode de *hashage* fait correspondre les bi-grammes avec 2^{24} classes ou *bins* en anglais.

Cette phase est supposée fonctionner correctement et nous faisons le choix de nous concentrer sur les phases suivantes : *Document Selector* et *Document Reader*.

3.2. Document Selector

Soit un couple question-réponse $(\boldsymbol{q}, \boldsymbol{a})$ et soient $\{\boldsymbol{q_1}, \dots, \boldsymbol{q_L}\}$ l'ensemble des *tokens* composants la question *q*. Nous disposons d'un ensemble $\boldsymbol{P}$ de paragraphes $\boldsymbol{p}$, eux-mêmes décomposables en mots[8] : $\boldsymbol{p_{i,j}}$ désignera le *j*-ème mot du *i*-ème paragraphe, de même que $\boldsymbol{p_j}$ désignera le *j*-ème mot du paragraphe *p*.

Les mots – ou *tokens* – ne sont pas utilisés tels quels. Ce sont en réalité leurs encodages – c'est-à-dire leur représentation vectorielle dans la base *GloVe* ou *word embeddings* – qui sont utilisés en entrée des réseaux. Cette traduction est largement simplifiée par l'utilisation de la librairie *torch.nn.Embeddings*.

Nous supposons que les paragraphes présentés sont pertinents pour répondre à *q*, soit parce qu'ils sont fournis avec la question dans la base SQuAD, soit parce qu'ils auront été extraits à l'aide du *Docuement Retriever*.

Notre modèle cherche à maximiser la probabilité d'extraire la réponse *a* de l'ensemble *P* des paragraphes sachant la question *q*. Ce problème est décomposable en sous-problèmes plus simples, à l'aide de la formule des probabilités totales :

$$\Pr(a \mid q, P) = \sum_{p \in P} \Pr(a \mid q, p) \,.\, \Pr(p \mid q, P)$$

Au cours d'une première étape, le modèle *Document Selector* va apprendre la distribution de probabilité $\Pr(p \mid q, P)$ sur l'ensemble des paragraphes présentés. En d'autres termes, ce premier réseau renverra en sortie la distribution de probabilité que la réponse à la question *q* se trouve dans un paragraphe *p*, parmi un ensemble *P* de paragraphes donnés en entrée. Ou dit encore autrement, il renverra un indice de confiance pour chaque paragraphe, et seulement les paragraphes les plus pertinents seront soumis à l'étape suivante du *Document Reader*. Le rôle du *Document Selector* est ainsi de *débruiter* la sortie du *Document Reader* en lui donnant en entrée un ensemble plus pertinent de paragraphes à examiner.

Formellement, cela peut s'écrire sous forme matricielle, où $\boldsymbol{W^{SELECT}}$ est une matrice à apprendre.

[8] Pour simplifier, nous considèrerons que seuls les mots-clés sont présents dans le paragraphe. Cela élimine donc les autres types de *tokens*, tels que les caractères de ponctuation ou encore les apostrophes.

$$\Pr(p_i \mid q, P) = \underset{1 \le i \le M}{\text{softmax}}\left(\max_{1 \le j \le N}(p_{i,j}\, \boldsymbol{W^{SELECT}} q)\right)$$

La fonction max ci-dessus s'explique par l'hypothèse faite qu'un paragraphe est plus informatif que tous les autres pour répondre à la question posée. Si la réponse se trouvait dispersée dans plusieurs paragraphes, notre modèle serait incapable de la trouver.

Dans le cas de notre implémentation, nous avons choisi de travailler avec $M = 30$. C'est pratiquement le seul changement qu'il a fallu faire pour que la méthode s'exécute étant donné la mémoire limitée à 8G de notre carte graphique. Pour information, M vaut 50 dans la version originale du code.

3.3. Document Reader

Après cette phase d'écrémage rapide, le modèle *Document Reader* prend le relai pour apprendre la distribution de probabilité $\Pr(a \mid q, p)$ de la position de la réponse dans un paragraphe, sachant la question. Plus précisément, ce second modèle apprend les positions de début et de fin de la réponse a dans la séquence $p = \{p_1, \dots, p_N\}$, sachant la séquence $q = \{q_1, \dots, q_L\}$ en entrée du réseau. En termes plus formels, si l'on dénote par a_s (resp. a_e) $\in [1, \dots, N]$ la position de début (resp. de fin) de la réponse a dans la séquence $\{p_1, \dots, p_N\}$, cela se traduit par l'équation suivante, dans laquelle $\boldsymbol{W_s^{READ}}$ et $\boldsymbol{W_e^{READ}}$ sont deux matrices à apprendre :

$$\begin{cases} \Pr(a \mid q, p) = \max\limits_{1 \le a_s, a_e \le N} \left(P_s(a_s \mid q, p)\, P_e(a_e \mid q, p)\right) \\ P_s(j) = \underset{1 \le j \le N}{\text{softmax}}\left(p_j \boldsymbol{W_s^{READ}} q\right) \\ P_e(j) = \underset{1 \le j \le N}{\text{softmax}}\left(p_j \boldsymbol{W_e^{READ}} q\right) \end{cases}$$

La fonction max s'explique ici par le fait que la réponse peut se rencontrer plusieurs fois dans un même paragraphe.

3.4. Fonction coût

Notre estimateur du maximum de vraisemblance prend la forme suivante :

$$L(\theta) = -\sum_{(q,a,P) \in \Gamma} \log\left(\Pr(a \mid q, P)\right) - \alpha\, R(P)$$

où $R(P)$ est un terme de régularisation contrôlé par le paramètre α = 1 dans le code, θ représente l'ensemble des paramètres de nos deux modèles et Γ désigne les jeux de données question-réponse et question-documents de la base SQuAD.

Dans cette implémentation de la méthode, la régularisation mesure en réalité la divergence de Kullback-Leibler[9] – aussi appelée *entropie relative* – entre la distribution de probabilité $\Pr(p_i \mid q, P)$ obtenue en sortie du *Document Selector* et la vérité-terrain, qui n'est autre que la proportion de paragraphes $p_i \in P$ qui contiennent la réponse à la question q.

[9] Pour deux distributions de probabilités discrètes P et Q, la divergence de Kullback–Leibler de P par rapport à Q est définie par $D_{KL}(P, Q) = \sum_i P(i)\, \log(\frac{P(i)}{Q(i)})$. Dans le cas présent, la vérité-terrain est une loi de probabilité uniforme sur l'ensemble des paragraphes d'un même article.

3.5. Spécificités du modèle

L'encodage des paragraphes

La présentation faite au paragraphe précédent est volontairement simpliste. En réalité, lors de la phase de *forward propagation* du réseau de neurones, les encodages ou *word embeddings* passent d'abord dans un réseau récurrent bi-directionnel à une couche, de type LSTM. La couche cachée est alors utilisée pour encoder les mots en entrée du *Document Selector* puis du *Documenr Reader*.

$$\{\hat{p}_1, \dots, \hat{p}_N\} = Hidden\ layer\ of\ StackedBRNN\big(\ \{\hat{p}_1, \dots, \hat{p}_N\}\ \big)$$

Cela a pour effet d'encoder l'information véhiculée par les mots, non pas de façon isolée, mais en tenant compte du contexte dans lequel ils sont placés.

Ce réseau récurrent est également appris lors de la phase d'apprentissage, en même temps que tous les autres paramètres du modèle.

L'encodage des questions et l'auto-attention

Cette même opération est réalisée pour encoder les séquences de mots dans les questions, à la différence que l'on ajoute à cette représentation cachée une pondération – ou opération d'auto-attention – qui permet d'encoder également l'importance des mots dans les questions. Cela se comprend assez aisément car le premier mot d'une question ouverte – *What*, *Why*, *Where*, *Who* ou *When* – dirige fortement certaines caractéristiques de la réponse attendue – par exemple, *Where*, *Who* ou *When* auront tendance à référer à des entités nommées.

$$\{\hat{q}_1, \dots, \hat{q}_L\} = SelfAttention\Big(\ Hidden\ layer\ of\ StackedBRNN\big(\ \{\hat{q}_1, \dots, \hat{q}_L\}\ \big)\ \Big)$$

Ce réseau récurrent est également appris lors de la phase d'apprentissage, en même temps que tous les autres paramètres du modèle.

Plus précisément, si l'on note $\{\bar{q}_1, \dots, \bar{q}_L\}$ la couche cachée du réseau récurrent appliqué à la séquence $\{\hat{q}_1, \dots, \hat{q}_L\}$, l'opération $SelfAttention$ s'écrit :

$$\begin{cases} \hat{q}_j = \sum_{1 \le k \le L} \alpha_{j,k}\, \bar{q}_k \\ \alpha_j = \underset{1 \le k \le N}{\text{softmax}}\big(\ \boldsymbol{w^{ATTN}} \bar{q}_k\ \big) \end{cases}$$

où $\boldsymbol{w^{ATTN}}$ est un vecteur de pondération à apprendre.

Les variables explicatives du modèle ou *features*

En plus de l'encodage fourni par *GloVe*, nous ajoutons à notre modèle les éléments suivants :

1. Trois variables binaires indiquant si un des mots du paragraphe se trouve également dans la question, soit sous sa forme originale, soit sous sa forme minuscule, soit enfin sous sa forme canonique.
2. Trois étiquettes syntaxiques permettant de caractériser les mots du paragraphe dans leur contexte : NER pour *named entity recognition*, POS pour *part-of-speech* et TF pour *term frequency*.

4. Conclusions

Les choix dans la construction de réseaux neuronaux m'ont toujours paru arbitraires et c'est la raison pour laquelle j'ai choisi de m'intéresser à ce type de modèles afin de mieux appréhender ces choix sur un cas d'application précis. Le cas des réseaux récurrents est d'autant plus intéressant qu'il est un candidat naturel dans le cadre du traitement du langage naturel, qui est aussi un centre d'intérêt que je nourris.

4.1. Principaux apprentissages

Il faudra bien plus qu'un projet de quelques semaines pour se forger une intuition solide sur ces questions, mais quoiqu'il en soit, ce petit défi personnel aura été l'occasion de :

1. Faire une première prise en main de PyTorch, qui offre une opportunité – assez formidable – de réaliser une descente de gradient dans un graphe computationnel sans avoir à se soucier des détails pratiques ;
2. Travailler avec des encodages syntaxiques, ou *word embeddings* en anglais, du projet GloVe[10] ;
3. Explorer un cas concret de fouille de textes à grande échelle à l'aide de la mesure de similitude *TF-IDF* ;
4. Extraire des informations syntaxiques avec spaCy – parsage textuel, mise sous forme canonique, étiquetage grammatical et reconnaissance d'entités nommées[11] – sachant que la mise en oeuvre originale utilisait le package *CoreNLP* de Stanford.

Un aspect négatif de ce choix de sujet se dégage clairement néanmoins, à savoir l'effet « empilement des boîtes noires » décrites dans le paragraphe précédent. Sachant que le modèle se compose de plusieurs milliers de lignes de code, certaines idées d'apport personnel se sont avérées trop lourdes à intégrer dans le temps imparti[12]. Il est évident qu'un sujet moins ambitieux se serait mieux prêté à l'interprétation et à l'analyse des résultats.

4.2. Possibiltés de pistes à explorer

Au cours des prochaines semaines, j'entends poursuivre les axes suivants, par ordre de priorité :

- Utilisation d'encodages (*word embeddings*) construits à partir de la base Wikipedia[13]
- Utilisation de transformeurs à la place des réseaux de type LSTM, en me basant dans un premier temps sur la librairie *torch.nn.Transofomer* de PyTorch[14]
- Utilisation du modèle BERT, supposément à la pointe en termes de performance sur les sujets classiques de traitement du langage naturel.

4.3. Sources

Pour m'auto-former sur le sujet, et avec l'aide des mes tuteurs, je me suis appuyé sur les publications suivantes :

---

[10] https://nlp.stanford.edu/projects/glove/
[11] En anglais : *tokenization*, *lemmatization*, *POS tagging* et *NER*
[12] Cela étant dit, je suis beaucoup plus à l'aise pour continuer à explorer le sujet dans un futur proche.
[13] https://github.com/idio/wiki2vec/
[14] https://pytorch.org/tutorials/beginner/transformer_tutorial.html

Yankai Lin, Haozhe Ji, Zhiyuan Liu, Maosong Sun, 2018 : *« Deniosing Distantly Supervised Open-Domain Question Answering »*

Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes, 2017 : « *Reading wikipedia to answer open-domain questions »*

Jeffrey Pennington, Richard Socher, Christopher D. Manning, 2014 : *« GloVe : Global Vectors for Word Representation »*

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin, 2017 : *« Attention is all you need »*

Jacob Devlin Ming-Wei Chang Kenton Lee Kristina Toutanova 2019 : *« BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding »*