Question Answering Project with Deep Learning

Charles de la Roche Saint André
CentraleSupélec
charles.de-la-roche-saint-andre@student.ecp.fr

Axel Compain
CentraleSupélec
axel.compain@student.ecp.fr

Adrien Thomas
CentraleSupélec
adrien.thomas@student.ecp.fr

Abstract

Tandis que les moteurs de recherche fondent leur recherche sur l'utilisation de mots clefs, les systèmes de "Question Answering" explorent de nouvelles méthodes de recherche d'informations exploitant des requêtes formulées à l'aide du langage naturel. Le système informatique concerné utilise alors des techniques de traitement automatique des langues afin d'analyser la question et de rechercher une réponse adéquate au sein des documents auxquels il a accès

Introduction

Nous nous sommes donc proposés de créer un modèle de Question Answering qui reproduirait l'état de l'art en terme de réponse à une question portant sur un contexte donné. L'objectif de notre projet a été d'implémenter et d'entraîner un réseau de neurones qui prendrait en entrée un contexte de taille limitée et une question portant sur ce dernier et qui renverrait l'extrait du contexte répondant à cette question.Language

1. Question Answering

Un des projets phares en lien avec notre problématique est celui démocratisé par le SQuAD (Stanford Question Answering Dataset) qui consiste à entraîner un algorithme afin qu'il puisse répondre à une question portant sur un texte en renvoyant le passage du texte répondant à la question. Le dataset mis à disposition est ainsi constitué d'un ensemble de textes, de questions et des réponses à ces questions.

Nous nous sommes intéressés à des projets déjà existants et face à leur complexité, nous avons décidé de les subdiviser en sous-problèmes abordables. Nous avons rapidement identifié cinq axes principaux de travail:

- le preprocessing : cette étape permet de traiter nos données textuelles, de les tokeniser, d'en éliminer la ponctuation, et d'associer à chaque mot un index unique
- l'embedding layer : cette étape primordiale consistant à modéliser les mots en des vecteurs
- l'encoder layer : cette étape permet de vectoriser le texte et la question de façon à en garder le sens
- l'attention layer : cette étape permet d'identifier la séquence du texte pertinente pour la question donnée
- l'output layer : cette étape permet de déterminer les positions de début et de fin de la réponse dans le texte associé

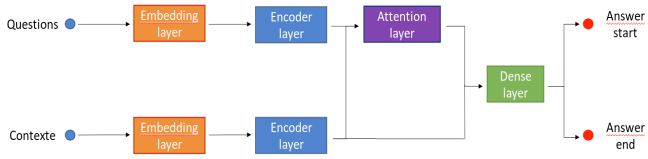

Figure 1: Architecture de notre modèle.

1.1. Preprocessing

Un des défis clés était de gérer le nombre de paramètres à entraîner pour notre modèle. La taille maximal des contextes et la taille maximale des questions sont des variables qui jouent grandement sur le nombre de paramètres. En analysant l'histogramme de la longueur des contextes de notre dataset, à partir d'une taille maximale de contexte initiale de 675 mots, nous avons décidé de ne garder que les contextes avec moins de 150 mots, ce qui nous permet de garder 78% de notre dataset. Cela permet de diminuer par un facteur 4 la taille maximale des contextes. Dans un second temps, la taille du vocabulaire de notre dataset était également une variable avec un impact sur le nombre de paramètres de notre modèle. Pour diminuer cette variable, nous avons donc choisi de remplacer tous les mots qui ne sont pas dans notre vocabulaire d'embedding par le même mot identifiable dans la phase d'embedding sous le pseudonyme "unknown".

Pour finir le nettoyage des données, nous avons supprimer à l'aide d'opération sur les expressions

régulières la ponctuation. De plus, nous avons décidé de transformer chaque chiffre en un seul et même mot identifiable dans la phase d'embedding sous le pseudonyme "digit". En effet, l'information utile est de savoir que nous avons affaire à un chiffre et non de savoir précisément quel chiffre ou nombre. Cela contribue en plus à diminuer la taille du vocabulaire.

1.2. Embedding Layer

Nos données initiales étant ici sous forme de texte, il est nécessaire de représenter ces données sous forme de vecteur. Nous avons donc mis en place une couche d'embedding afin de faciliter notamment l'analyse sémantique des mots.

Cette nouvelle représentation a ceci de particulier que les mots apparaissant dans des contextes similaires possèdent des vecteurs correspondants qui sont relativement proches. L'hypothèse principale de ces méthodes étant de prendre en compte le "contexte" dans lequel le mot a été trouvé, c'est à dire les mots avec lesquels il est souvent utilisé. On appelle cette hypothèse distributional hypothesis. Ce contexte permet de créer un espace qui rapproche des mots qui ne se sont pas forcément trouvés à côté les uns des autres dans un corpus.

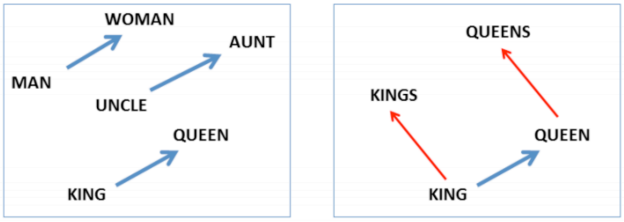

Figure 2: Représention de la « distributional hypothesis ».

Deux méthodes connues sont utilisées pour entraîner l'embedding de mot:

- GloVe developpé en Open-Source par Stanford utilise des algorithmes d'apprentissage non supervisé.
- Word2Vec développé par des équipes de Google et qui utilise des réseaux de neurones artificiels à deux couches entraînés. Cette méthode est accessible à travers la bibliothèque Gensim de Python.

Une troisième méthode, plus lente, serait également de développer un modèle spécifique à notre dataset.

Il existe également deux possibilités d'entrainement :

- Dans le développement de la couche d'embedding, nous pouvons soit apprendre à notre dataset un embedding en le "fittant" à un modèle d'embedding GloVe ou Word2Vec. L'embedding serait alors adapté à notre vocabulaire. L'avantage d'entraîner son propre embedding est donc d'avoir un plongement spécifique à notre corpus et donc plus performant concernant la problématique que l'on veut traiter.
- Une deuxième méthode serait également, d'utiliser un vocabulaire gigantesque qui aurait déjà été entraîné sur un modèle GloVe ou Word2Vec et d'en extraire uniquement le vocabulaire qui nous intéresse. Les chercheurs à l'origine de la méthode GloVe proposent sur leur site Web une série de mots-clés pré-formés. Nous utiliserons un paquet formé sur un ensemble de données d'un milliard de mots avec un vocabulaire de 400 000 mots. Il existe différentes tailles de vecteur d'incorporation, notamment 50, 100, 200 et 300 dimensions. Pour une dimension de 50, en regardant à l'intérieur du fichier, nous pouvons voir un "token" (mot) suivi des poids (50 nombres entre -1 et 1) sur chaque ligne. Il reste deux options principales pour l'utilisation d'embeddings pré-formés.

Nous avons donc opté pour une méthode intermédiaire qui a été évoquée dans le preprocessing.
Nous avons mis en place la matrice d'embedding avec les mots de notre vocabulaire qui était dans le vocabulaire GloVe. Pour les chiffres (les "digits"), nous les avons indexés selon le même index en avant dernière position de la matrice. Pour tous les autres mots non présents dans le vocabulaire GloVe (les "unknown"), nous les avons indexés selon un dernier index différent en dernière position de la matrice.

Pour ces deux derniers indices, nous les avons initialisés aléatoirement selon la loi normale que suivait chacun des coefficients la matrice d'embedding de notre vocabulaire.
Ainsi chaque mot de notre dataset est transformé en un indice qui est ensuite lui même changé à travers la matrice d'embedding en un vecteur de dimension 50.

1.3. Encoder Layer

La problématique est ici la suivante : étant donnée une séquence de vecteurs représentant des mots, comment saisir le contexte et donc le sens de la phrase? Les algorithmes classiques de Deep Learning tels que les Réseaux Denses ne permettent pas de répondre à cette problématique.

Les RNN ont cette notion de persistance de l'information du fait qu'ils bouclent sur leur sortie. Les réseaux «Long Short Term Memory» ou LSTM sont des éléments essentiels de ce type de réseau de neurones récurrents qui fonctionnent beaucoup mieux que la version standard pour de nombreuses tâches. Nous gardons

toutefois à l'esprit qu'il existe un type plus récent de réseaux, les Gated Recurrent Unit ou GRU qui peuvent s'avérer plus efficace (malheureusement pas dans notre cas, cf. la partie implémentation du modèle).

1.4. Attention Layer

Dans les problèmes classiques de Question Answering, pour que la machine soit capable de répondre à une question à partir d'un contexte donné, il est nécessaire de construire un modèle complexe prenant en compte les interactions fortes entre le contexte et la question.

A la sortie de l'encoder layer, nous avons un vecteur encodé pour la question et un vecteur encodé pour le contexte. Jusqu'à présent, la question et le contexte ont été traité de manière totalement dissociée. Pour déterminer la réponse, nous devons considérer ces deux vecteurs ensemble. C'est typiquement le rôle des mécanismes d'attention qui sont implémentés dans la couche d'attention.

La couche d'attention est l'élément clef de notre modèle puisque c'est elle qui va nous permettre de déterminer quelle partie du contexte permet de répondre à la question posée. En effet, par un ensemble d'opérations de calcul matriciel, cette couche va évaluer l'importance de chaque mot du contexte par rapport à la question posée et nous permettre d'identifier les éléments de réponse. Nous avons décidé d'implémenter un modèle de mécanismes d'attention assez classique et basique permettant de mettre en place une attention unidirectionnelle : the Dot Product Attention.

Tout d'abord, nous commençons par multiplier chacun vecteur encoded_question par le vecteur encoded_context lui correspondant. Ce produit nous donne un vecteur nommé attention_scores. Ensuite, nous appliquons une fonction softmax à ce vecteur obtenu, ce qui nous permet d'obtenir un vecteur appelé attention_distribution. Pour finir, nous multiplions ce vecteur en sortie à nouveau avec le vecteur question encodé initial et nous obtenons notre vecteur attention_output.

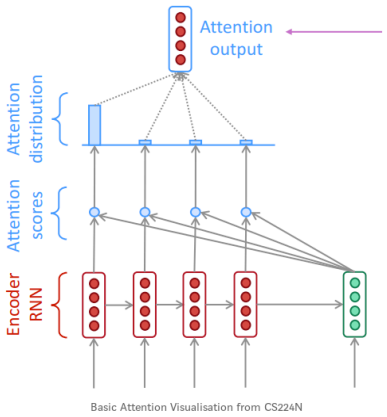

Figure 3: Représention de l'Attention Layer.

1.5. Output Layer

Dans la dernière couche de notre réseau, nous commençons par concaténer notre contexte encodé et notre vecteur attention_output. Ce vecteur concaténé appelé merged devient l'entrée d'un réseau dense. La sortie de ce réseau dense passe alors dans deux couches d'activation différentes appliquant chacune un softmax à nos vecteurs. Ces deux couches nous permettent de déterminer les positions de début et de fin de la réponse qui sont caractérisées dans notre modèle par des vecteurs

Nous avons choisi comme fonction de perte la somme de la categorical cross-entropy loss pour la position de début de la réponse et celle pour la position de fin. Chaque contexte étant paddé sur une taille de 150, nous avons estimé que notre problème revenait à prédire la classe parmi les 150 la plus adaptée pour chaque position et donc que la "categorial cross-entropy" convenait parfaitement à notre modèle.

Nous avons choisi l'optimiseur Adam pour minimiser cette fonction d'erreur car c'est celui qui présente les meilleurs résultats en général.

Afin d'évaluer nos prédictions sur les données d'entraînement et sur les données de validation, nous avons choisi la mesure categorical accuracy pour les mêmes raisons qui nous ont poussés à choisir la categorical cross-entropy fonction d'erreur.

2. Implémentation et optimization du modèle

Après avoir analysé et pris en main les différentes couches, nous avons pu implémenter le modèle dans son ensemble.

2.1. Apprentissage du modèle et overfitting

Dans un premier temps, nous avons eu du mal à obtenir une convergence. Nous avons pu identifier que le problème venait de la taille de notre modèle (en terme du nombre de paramètres à entraîner). Malgré notre choix de ne pas entraîner notre matrice d'embedding (plus de 8M de paramètres), notre modèle comportait encore plus de 3 millions de paramètres à entraîner. Nous avons donc décidé de nous séparer de la partie du dataset d'entraînement dont le taille de contexte était la plus grande (cf. partie PREPROCESSING). Grâce à cette astuce, nous avons réduit notre modèle à 500k paramètres à entraîner.

Avec cette première version du modèle, en utilisant un réseau LSTM en guise d'encoder, nous avons réussi pour la première fois à le faire converger et en particulier à faire en sorte qu'il réussisse à apprendre notre dataset. Nous avons lancé cette opération d'entraînement sur 69 719 samples et l'avons testé sur les 3 353 samples restants.

Nous avons rapidement réalisé que dans la configuration actuelle, le modèle était surtout en train de sur-apprendre sur notre set d'entraînement (overfit) puisque la fonction d'erreur diminuait significativement après chaque epoch (passage complet du dataset) tandis que la fonction d'erreur sur le set de validation continuait de croître.

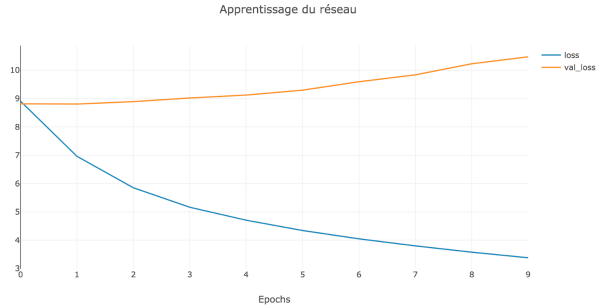

Figure 4: Courbe d'apprentissage du réseau en overfittant

Le fait que le modèle converge était en soi une première réussite. Nous avons donc décidé de mettre en place des mesures pour éviter ce phénomène et procédé à la régularisation du réseau.

2.2. Régularisation et masking

Afin d'éviter l'overfitting, nous avons décidé d'ajouter du dropout à notre réseau. Nous nous sommes inspirés des différents modèles que nous avions pu observé et avons décidé de placer trois couches de dropout : une après chaque RNN et une après l'attention layer. Nous avons de plus réglé le taux de dropout (taux de connexions aléatoirement annulées) à 0.5 comme cela se fait conventionnellement.

Nous avons de plus observé qu'il pouvait être judicieux d'ajouter un masque à notre réseau : les contextes et les questions ayant été paddés, il y a un risque que les couches RNN donnent de l'importance à ces éléments de padding qui pourraient faire perdre de l'information essentielle à l'interprétation de la phrase. Nous avons donc utilisé un masque qui permet de s'assurer que les encoders ne prennent pas en compte ces éléments. Nous rajoutons simplement après les RNN une couche de NonMasking que nous avons nous même créée afin de ne plus renvoyer le masque aux couches suivantes (les opérations faites lors de l'attention layer ne sont pas compatibles avec un masque).

Grâce à ces deux améliorations (principalement au dropout), nous avons obtenu des résultats de généralisation bien plus satisfaisants que précédemment sur la série d'entraînements suivante.

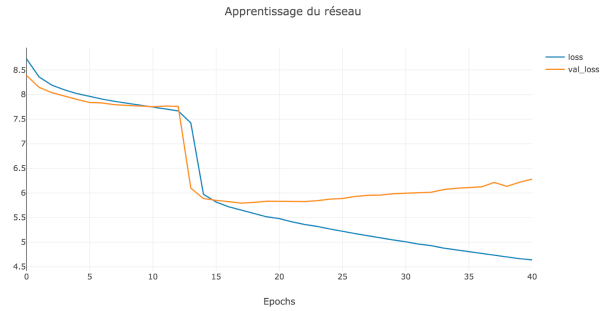

Figure 5: Courbe d'apprentissage du réseau en se généralisant.

Nous avons ainsi pu valider notre architecture. Vous pourrez trouver le graphe complet ci-dessous. Le modèle a toutefois une categorical accuracy faible (autour de 5% de réponse justement prédite sur le test set). Nous avons donc entrepris la phase d'optimisation.

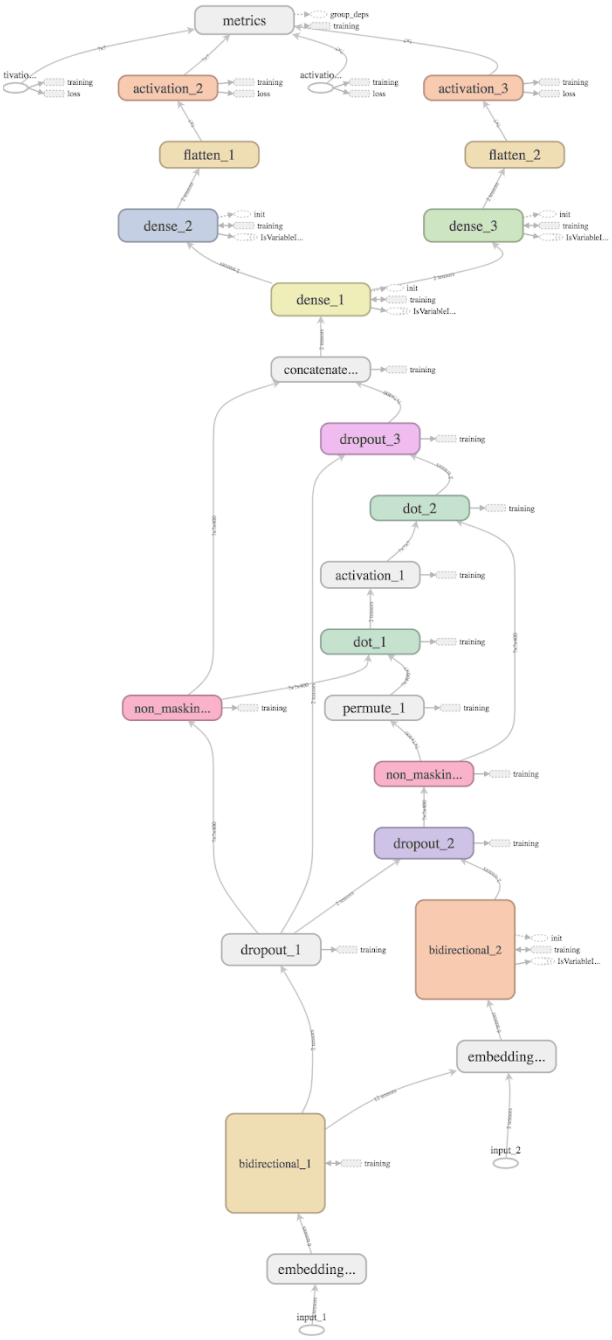

Figure 6: Courbe d'apprentissage du réseau avec un LSTM Bidirectionnel.

2.3. Optimisation et choix de la couche d'encoding

Nous avons rapidement identifié qu'un facteur d'influence important de notre modèle était la qualité de la couche RNN. Nous avons donc décidé de tester deux types de couches recommandées pour notre cas d'usage, LSTM et GRU, et de les comparer.

Nous avons de plus tenu à comparer leur efficacité à celle d'une même couche bidirectionnelle. Cette couche permet d'envoyer à la couche suivante des informations d'états passés (parcours de la phrase dans le sens conventionnel) et futures (parcours de la phrase dans le sens inverse) et ainsi d'augmenter la quantité d'informations d'entrée disponible sur le réseau. Plus simplement, le réseau lit le texte dans les deux sens et en tire une meilleure compréhension.
Nous avons donc entraîné notre réseau avec un encoder LSTM bidirectionnel et avons obtenu des résultats probants.

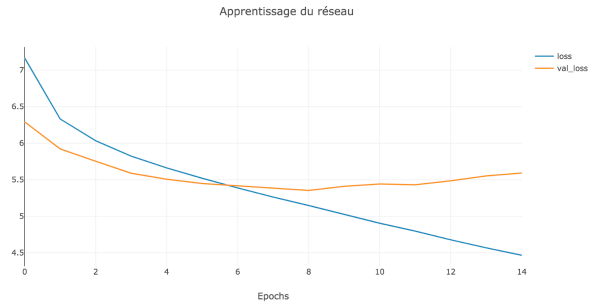

Figure 6: Courbe d'apprentissage du réseau avec un LSTM Bidirectionnel.

Nous avons de plus tenté l'entraînement avec un GRU bidirectionnel mais les performances de prédictions étaient faiblement inférieures à celles obtenues à l'aide du LSTM. Notre modèle dans sa forme finale nous a ainsi permis d'obtenir un taux de prédiction de 30% sur l'indice de début de réponse dans le contexte et de 33% sur l'indice de fin de réponse. On obtient ainsi en combiné 21% de réponses justes sur notre dataset test (3 553 questions et contextes).

Couche d'encoding	Accuracy p_start	Accuracy p_end	Accuracy croisée
GRU	*20%*	*23%*	*13%*
GRU Bidirectionnel	*27%*	*30%*	*19%*
LSTM	*23%*	*27%*	*15%*
LSTM Bidirectionnel	*30%*	*33%*	*21%*

Figure 7: Résultats d'accuracy des différents types d'encoder.

3. Environnement matériel et logiciel

- Dataset d'entraînement SQUAD
- Keras / TensorFlow pour le Deep Learning
- Bibliothèque NLTK de Python pour le NLP
- Glove pour le Word Embedding
- Utilisation du Jupyter Notebook pour les travaux d'exploration

4. Conclusion

Un des plus gros apports de ce projet de Question Answering fut de développer nos connaissances en Deep Learning et de les mettre en application dans un cadre concret. Ainsi, nous avons pu expérimenter sur notre propre dataset de nombreuses techniques de preprocessing existantes. Nous avons amélioré de façon significative nos connaissances en TensorFlow et en Keras en nous documentant à de nombreuses reprises en ligne. En parallèle de notre progression pratique, nous avons pu nous familiariser avec la théorie des réseaux de neurones, en particulier les récurrents et les multi-perceptrons. Enfin, dans le but d'améliorer nos performances, nous avons eu l'occasion d'expérimenter des techniques de masking et de dropout afin d'éviter l'overfitting et d'améliorer l'apprentissage de notre réseau.

Pour améliorer les performances de notre architecture actuelle, nous avons identifié plusieurs pistes : régulariser nos couches, jouer sur le learning rate (le faire décroitre avec l'entrainement) et sur les autres hyperparamètres ou encore changer la taille de notre embedding.

Une autre solution pour améliorer notre modèle pourrait être de revoir son architecture et particulièrement le mécanisme d'attention. En effet, nos recherches sur l'état de l'art nous ont permis d'isoler un mécanisme d'attention plus complexe développé sur une publication de 2017, "Bi-directionnal Attention Flow For Machine Comprehension", qui permettrait de grandement améliorer la performance du réseau.

References

Embedding layer

[1] https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras/
[2] https://towardsdatascience.com/deep-learning-4-embedding-layers-f9a02d55ac12
[3] https://nlp.stanford.edu/projects/glove/
[4] https://github.com/keras-team/keras/issues/3110
[5] https://towardsdatascience.com/word-embedding-with-word2vec-and-fasttext-a209c1d3e12c
[6] https://machinelearningmastery.com/what-are-word-embeddings/

Encoder layer

[7] https://cs224d.stanford.edu/reports/StrohMathur.pdf
[8] http://smerity.com/articles/2015/keras_qa.html
[9] https://github.com/keras-team/keras/blob/master/examples/babi_rnn.py
[10] http://colah.github.io/posts/2015-08-Understanding-LSTMs/
[11] http://karpathy.github.io/2015/05/21/rnn-effectiveness/1PoU

Attention layer

[12] https://www.lebigdata.fr/deep-learning-definition
[13] https://www.futura-sciences.com/tech/definitions/intelligence-artificielle-deep-learning-17262/
[14] https://medium.com/octavian-ai/which-optimizer-and-learning-rate-should-i-use-for-deep-learning-5acb418f9b2
[15] "Bi-directionnal Attention Flow For Machine Comprehension" : https://arxiv.org/pdf/1611.01603.pdf
[16] https://fr.wikipedia.org/wiki/Fonction_d%27activation
[17] https://www.youtube.com/watch?v=aircAruvnKk&t=718s