Résumé d'article
Neural Text Generation from Structured Data with Application to the Biography Domain

KHOULADI Salma **LU Xiaohua**

1 Introduction

L'article [1] présente un modèle de neurones pour la génération de texte à partir de concept qui génère la première phrase biographique à partir des informations du 'infobox' en utilisant une large dataset contenant des biographies provenant de Wikipédia.

Keywords: Génération de texte, Modélisation du Langage Naturel (NLM),

2 Résumé de l'article

2.1 Travaux associés

Les systèmes de génération traditionnels reposaient sur des règles et des spécifications manuelles. La génération est divisée en trois parties : **Planification du contenu**, définit les parties des champs d'entrée qui doivent être sélectionnées, **Planification des phrases**, détermine les champs sélectionnés qui doivent être traités dans chaque phrase de sortie et **Réalisation**, génère ces phrases. Des approches ultérieures ont combiné deux ou plusieurs de ces parties. L'approche présentée s'inspire des modèles linguistiques utilisés pour le sous-titrage des images, la traduction automatique, la modélisation des conversations et des dialogues. Le modèle discuté utilise un réseau de neurones de type encodeur-décodeur, des unités LSTM et un mécanisme d'attention qui réduit la scalabilité.

2.2 Modèle du langage pour la génération de phrases avec contraintes

Ils prévoient d'utiliser et de construire un plus grand jeux de données que le jeux de données de météo en extrayant toutes les biographies sur Wikipedia [2]. Les approches basées sur les templates ne conviennent plus car il y a 400 000 mots et 700 000 biographies dans le nouvel ensemble de données et qu'ils ne puissent pas définir en détail toutes les règles! Ainsi, ils utilisent le **modèle de langage conditionné par tableau** pour contraindre la génération de la première phrase de l'introduction des articles Wikipédia.

2.2.1 Modèle de langage conditionné par tableau

L'action de copie (Copy action) est inspirée du fait que les phrases qui expriment des faits d'un tableau donné copient souvent des mots du tableau. Ils peuvent donc utiliser d'informations de tableau telles que: Q (Tous les tokens de tableau) lors du calcul du score pour des mots de sortie ω_t.

Pour utiliser les informations d'occurrence du mot dans le tableau, ils utilisent un descripteur de l'occurrence de mot sous forme de triplet qui inclut la position du mot compté à la fois depuis le début (+) et la fin du champ (-): $z_w = (f_i, p_i^+, p_i^-)_{i=1}^m$.

Ces descripteurs de table $z_{ct} = z_{w_{t(n1)}}, \dots, z_{w_{t1}}$ sont aussi appelés comme variables de conditionnement locales puisqu'elles décrivent les informations des relations de contexte local (mot précédent) avec la table.

g_f, g_ω représentent des informations de tous les tokens et champs de tableau, tels que les noms d'équipes, les noms de ligue. Ils peuvent aider le modèle à donner une meilleure prédiction.

Ils ont ainsi un nouveau modèle NLM conditionné: P($\omega_t|c_t, z_{ct}, g_f, g_\omega$), avec c_t le contexte à l'instant t et ω_t les candidats de sortie, $\omega \in W \ \ (words \cup CopyAction)$.

2.3 Cadre du modèle

Les entrées du NN sont définis par $x = \{c_t, z_{ct}, g_f, g_\omega\}$ et embeddings de x: $\psi(x) = \{\psi(c_t), \psi(z_{ct}), \psi(g_f), \psi(g_\omega)\}$. Ensuite, une **transformation linéaire** est appliquée sur les entrées obtenir alors les représentations cachées h(x). Troisièmement, ils utilisent h(x) pour obtenir **2 scores**: le score lié aux mots de sortie possibles (mots du vocabulaire) et le score lié aux descripteurs de tableau. Chaque mot $w \in W \cup Q$ reçoit ainsi un score final en additionnant les 2 scores. Enfin, la fonction **softmax** sera utilisée pour obtenir la probabilité des mots. Ce modèle de langage neuronal est entraîné pour **minimiser la log-vraisemblance négative** d'une phrase d'apprentissage S avec une descente de gradient stochastique.

2.4 Expérimentations

Toutes les biographies sont extraites de Wikipédia en utilisant la dataset WikiBio qui contient plus de 700k biographies, répartis en ensembles de train (80%), de validation (10%) et de test (10%). De chaque biographie, ils ont extrait l'Infobox et la section d'introduction de l'article de wikipedia.

Pour les expériences, seule la première phrase de la section d'introduction est générée. L'évaluation se fait en utilisant la méthode **Perplexité** et trois paramètres pour la qualité de la génération: **BLEU-4**, **ROUGE-4** (F-mesure), **NIST4**.

2.5 Résultats

Le modèle de base est le modèle linguistique Kneser-Ney (KN) (5-gram). Le modèle formé est un modèle linguistique de 11-gram. Sans l'utilisation des actions de copie, le modèle fonctionne mal par rapport au modèle KN. En ajoutant une addition locale (Local additioning), la mesure de la perplexité s'améliore légèrement. Les expériences avec des actions de copie et des conditionnements (locaux et globaux) ont les meilleurs scores pour toutes les mesures par rapport aux autres expériences et au modèle de base. L'utilisation de Beam Search pour le décodage des phrases, permet d'explorer un plus grand ensemble de phrases par rapport à la simple recherche gourmande. En comparant différents réglages de Beam, la meilleure validation BLEU peut être obtenue avec une taille de Beam K = 5, et ne prend que 200 ms par phrase.

3 Nos travaux

Notre tâche consiste à étudier le code et à tester progressivement des réglages de certains paramètres afin d'étudier leurs effets sur le résultat et d'aboutir à de meilleures performances.

3.1 Adaptation de code

Cette première partie consiste à adapter le code source à notre environnement (python 3.6 et plus, Tensorflow 1.x sur Google Colab). Le code source et l'ensemble des données Wikipedia sont partagés par Remi Lebret [1] sur GitHub. Le code est écrit en python 2 et utilise l'outil Tensorflow. Nous avons (1) **converti le code du python 2 en python 3**, (2) **configuré les chemins globaux** et (3) **géré les exceptions** en ajoutant des blocs 'Try Except' pour éviter les erreurs provenant des données anormales.

3.2 Configuration de l'environnement des expériences

3.2.1 Périmètre des données

Vue la largeur de la base de données utilisée, on a effectué des expériences en utilisant des différentes tailles de données. Pour bien conduire nos expériences, on utilise 10% des données pour tester les paramètres du modèle de langage, soit 7 mille exemples pour l'entraînement, 9 mille pour la validation et 9 mille pour le test. Ainsi que 1% des données pour tester les paramètres des expériences.

3.2.2 Métriques d'évaluation

La métrique utilisée dans le code fournit est **l'Entropie Croisée** *H(W)*. Dans son article, l'auteur évalue les modèles avec différentes métriques. On a choisi de développer deux de ces métriques. La première est **la Perplexité** qui est la métrique standard pour les modèles de langue. La perplexité est définit comme l'inverse normalisé de la probabilité de l'ensemble de test ou comme l'exponentielle de l'entropie croisée et est interprétée comme le nombre moyen des mots qui peuvent être encodés en utilisant *H(W)* mots, c'est à dire, le modèle va être confus de choisir entre *H(W)* mots. Alors, on cherche à minimiser la perplexité.

La deuxième métrique est **BLEU** (Bilingual Evaluation Understudy Score) qui calcule le nombre des n-gram identiques dans le texte généré et le texte de référence. L'ordre des mots n'est pas pris en compte. Elle retourne une valeur comprise entre 0 et 1 avec 1 si les deux textes sont identiques.

3.2.3 GPU vs CPU

Nous avons mené des expériences pour tester l'effet de l'utilisation du GPU et CPU sur le temps d'entraînement sur Colab. Les résultats ci-dessous montrent que l'entraînement en utilisant le GPU est 10 fois plus rapide que par le CPU. Par conséquent, nous utilisons le GPU pour des tests suivants.

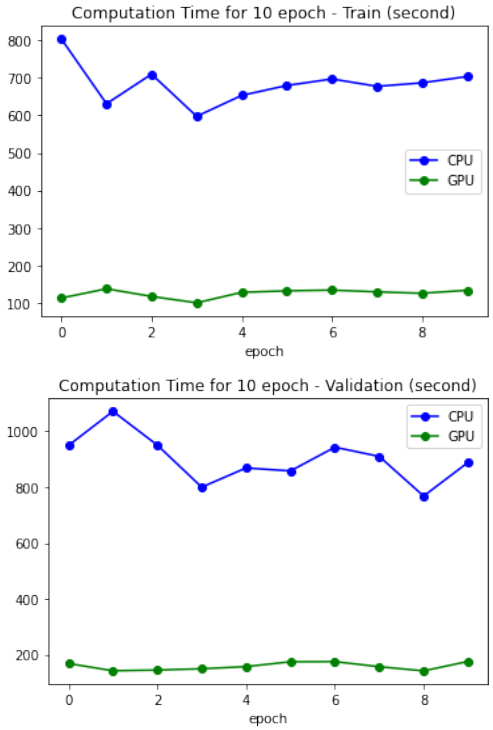

figure 3.3 - Comparaison de temps d'exécution entre CPU et GPU

3.3 Réglage des paramètres du modèle

Le modèle de langage proposé dans l'article se repose sur l'approche de Copy Attention. Dans cette partie, on teste sur différentes valeurs des paramètres du modèle Copy Attention: n (Ngram) et nhu (nombre des unités cachées) sur 1% des données (7K). L'évaluation se fait via les métriques entropie croisée, BLEU et temps d'exécution.

[1] L'auteur de Neural Text Generation from Structured Data with Application to theBiography Domain

3.3.1 Le paramètre de n-gram

Le paramètre n sert à définir n-gram du modèle de language, autrement dit la longueur du contexte à un instant donné. Dans le code source, il est défini par défaut comme 11. Nous avons ainsi testé pour **n = 10, 11 et 14** pour comparer leur performances.

Dans les figures dessous, nous trouvons que n = 10 donne toujours la perte minimale lors de l'entraînement et de la validation alors que le 14-gram nous donne le meilleure score BLEU. De plus, nous trouvons que quand n est égale à 14, le score BLEU est environ **10 fois** plus grand que les autres cas.

Compte tenu le temps d'exécution, 14-gram est le plus coûteux mais acceptable. Ainsi, nous avons choisi d'utiliser **n = 14** pour la suite des expérimentations.

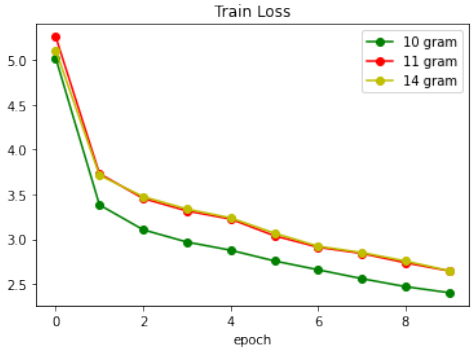

figure 3.4.1(1) - Perte d'apprentissage sur 10 époques

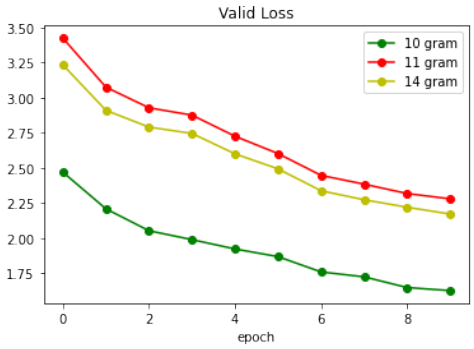

figure 3.4.1(2) - Perte de validation sur 10 époques

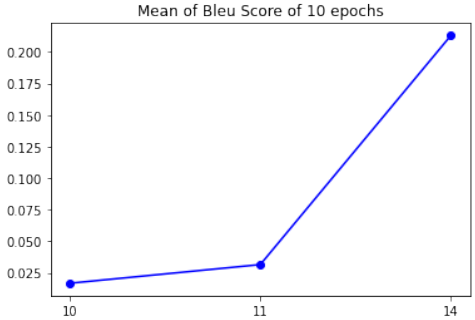

figure 3.4.1(3) - Moyenne du score bleu de 10 époques pour chaque valeur de n

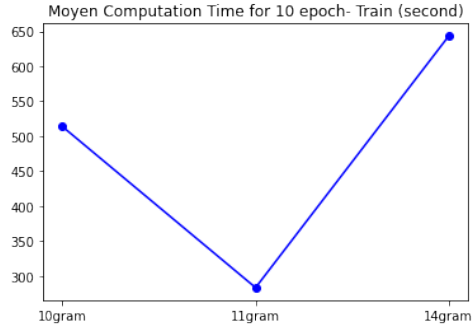

figure 3.4.1(4) - Temps d'exécution de 10 époques pour chaque valeur de n

3.3.2 Le nombre des unités cachées (nhu)

nhu signifie le nombre d'unités dans la couche cachée du réseau de neurones des actions de copie. La valeur par défaut de ce paramètre est 256.

Dans nos expériences, on teste **nhu = 128** et **nhu = 512**. L'évaluation des performances du modèle des actions de copie se fait via le calcul de perte. Dans les figures ci-dessous, nhu = 512 donne une perte minimale.

Pour le score BLEU, la différence entre ces 3 paramètres est assez petite. On peut dire que le changement de nhu n'améliore pas trop la performance de génération des textes.

Quant au temps d'exécution lors de l'entraînement, quand nhu est égale à 256, le temps d'exécution est minimal. Alors, on utilise **nhu = 256** pour nos expériences.

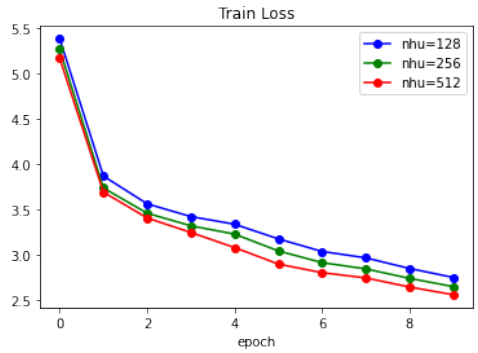

figure 3.4.2(1) - Perte d'apprentissage par époque pour différentes valeurs de nhu

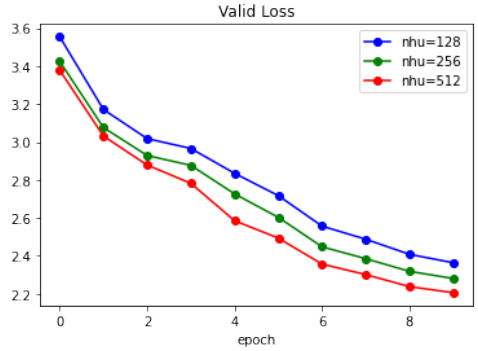

figure 3.4.2(2) - Perte de validation par époque pour différentes valeurs de nhu

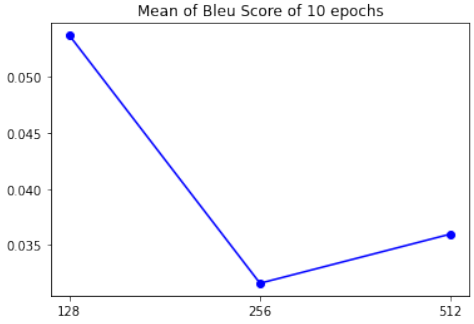

figure 3.4.2(3) - Moyenne du score bleu de 10 époques

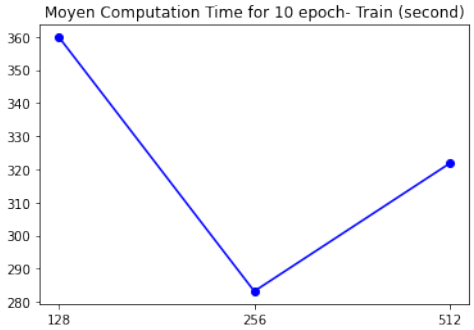

figure 3.4.2(4) - Temps d'exécution de 10 époques

3.4 Réglage des paramètres des expériences

Après avoir testé et choisi les paramètres du modèle Copy Attention, on fait des expériences sur les paramètres nombre d'époques et taille du batch afin d'étudier leur effet sur la performance.

3.4.1 Le nombre des époques

Ce paramètre définit le nombre de fois l'algorithme d'entraînement traite toutes les données du dataset. Sa valeur par défaut est 10. Nous avons implémenté le mécanisme "**Early Stopping**" pour éviter de choisir le nombre d'époque par hasard. Dans notre cas, nous notons la meilleure perte de validation et nous comparons ensuite si les 4 pertes de validation consécutives sont mieux que la meilleure perte de validation. Si oui alors on arrête l'entraînement.

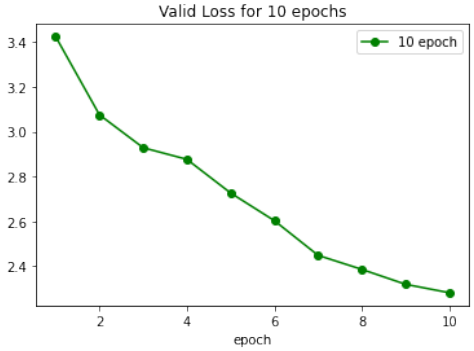

figure 3.4.1(1) - Perte de validation pour 10 époques(par défaut)

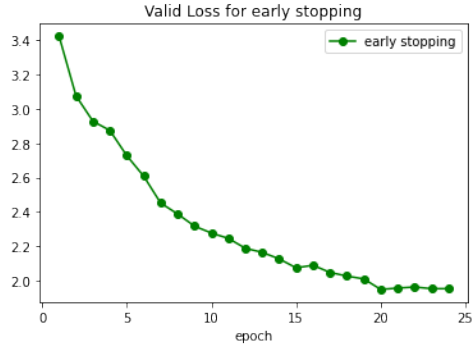

figure 3.4.2(2) - Perte de validation pour early stopping

3.4.2 La taille du batch

Ce paramètre définit le nombre d'exemples à traiter avant de mettre à jour les paramètres du modèle. La valeur par défaut de ce paramètre est 32. On étudie l'effet des tailles de batch 64 et 128. Selon les figures suivantes, on constate qu'on obtient des pertes minimales et un bleu maximal en batch de taille **128**.

Pour le score BLEU, la différence entre ces 3 paramètres n'est pas considérable. On peut dire que le changement de nhu n'améliore pas trop la performance de génération des textes.

Ainsi nous allons prendre **batch size = 32** ayant le temps d'exécution moins coûteux pour nos expériences.

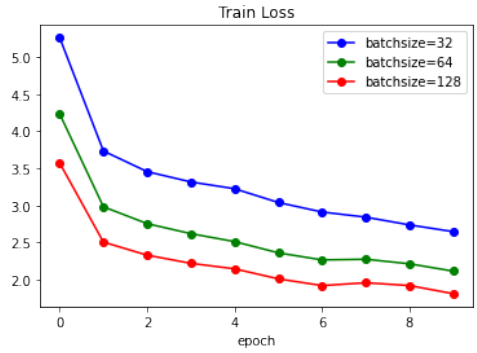

figure 3.5.2(1) - Perte d'apprentissage à chaque époque

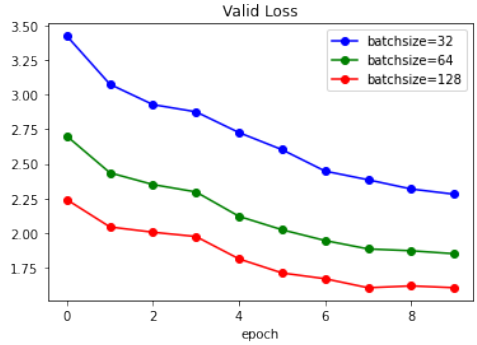

figure 3.5.2(2) - Perte de validation à chaque époque

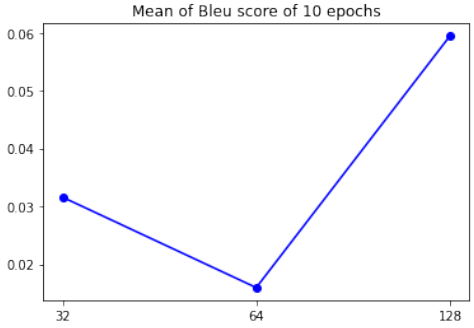

figure 3.5.2(3) - Moyenne du score bleu des 10 époques

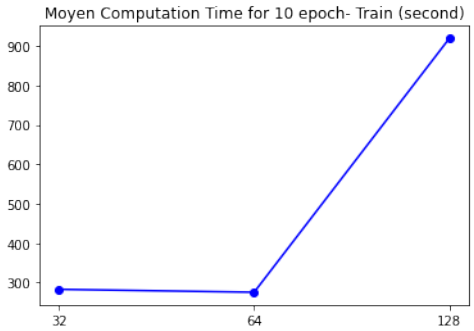

figure 3.5.2(4) - Temps d'exécution des 10 époques

3.4.3 Optimisateur

Nous voulons utiliser des optimisateurs différents lors de l'entraînement de modèle. L'optimisateur par défaut est le SGD(Stochastic Gradient Descent), et nous voulons aussi tester les optimisateurs adaptatifs tels que le RMSprop, Adagrad et Adam. Selon les figures suivantes, on constate que pour les pertes d'entraînement que les 4 optimisateurs nous donnent à la fin (lors de 10ème époque) sont presque pareilles. Puis SGD et Adam donnent les pertes de validation les plus petites lors de 10ème époque.

En regardant le score BLEU, Adam a la meilleure performance de générer des textes mais son temps d'exécution est ainsi trop long. En prenant compte de réalité, nous allons choisir **Adagrad** au lieu d'Adam pour avoir une performance moyenne.

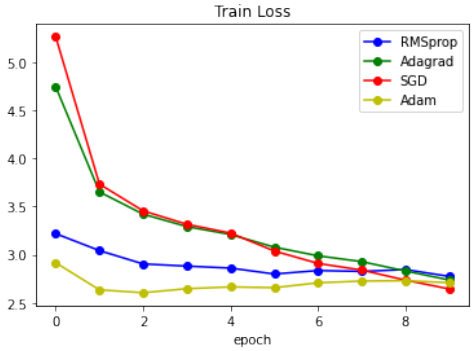

figure 3.5.3(1) - Perte d'apprentissage à chaque époque

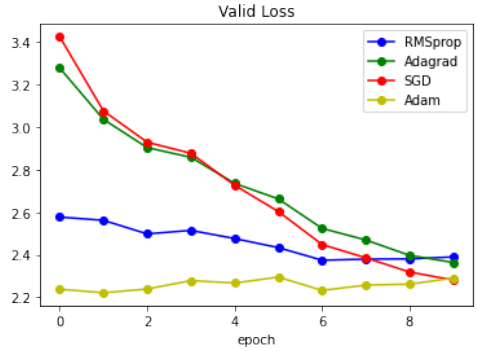

figure 3.5.3(2) - Perte de validation à chaque époque

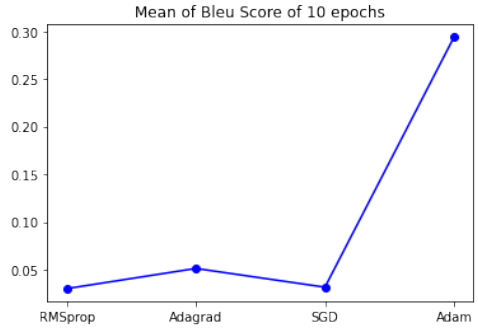

figure 3.5.3(3) - Moyenne du score bleu des 10 époques

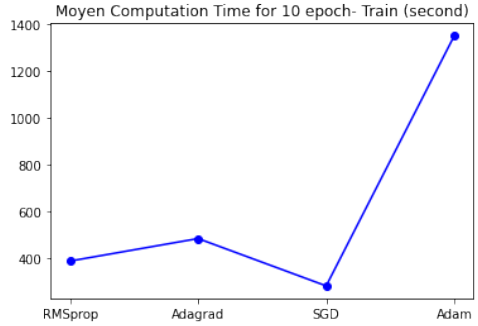

figure 3.5.3(4) - Temps d'exécution des 10 époques

3.5 Test après la modification des paramètres

Afin de bien analyser l'effet des choix des paramètres, on teste sur la moitié de la dataset. Nous avons d'abord effectué un premier test pour évaluer la performance du modèle avec des paramètres par défaut. Nous présentons dans le tableau suivant les phrases générées à la fin de la cinquième et dixième époques comparées à la phrase de référence. La note moyen de BLEU pendant 10 époques et 0.4239.

epoch 5
lenny randle, born february 12, 1949 is a former professional football player .
epoch 10
born february , 1949 , better known as lenny randle born february 12 , 1949 in long beach , california is a former major league baseball player .
Vérité terrain
lenny randle (born February 12, 1949) is a former Major League Baseball player.

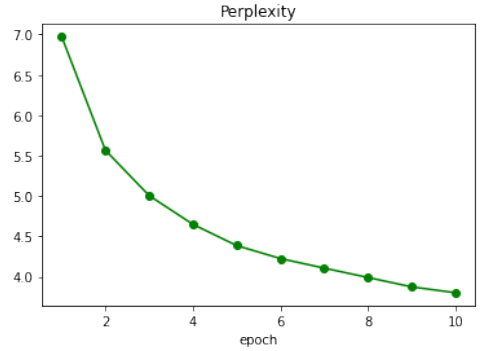

figure 3.5 - Perplexité lors de 10 époques.

Puis nous avons effectué un deuxième test avec des paramètres optimaux. Nous présentons ci-dessous les phrases générée à partir de la phrase de référence. (Pour le moment, nous avons pas encore obtenu les résultats et nous allons les montrer lors de soutenance.)

Paramètres	**n**	**nhu**	**epoch**	**Batch**	**Optimisateur**
Originaux	11	256	10	32	SGD
Changés	14	256	25	32	Adagrad

4 Conclusion

Le modèle linguistique peut générer des phrases en introduisant les actions de copie à partir du tableau et en conditionnant les champs et les mots du tableau.

Lors de ce projet, nous avons rencontré beaucoup de difficultés. Premièrement, le code source est écrit en python 2 et utilise l'outil Tensorflow. Ainsi, nous avons corrigé les écritures et les expressions non acceptées sous python 3.6 et traité des données anormales. Puis comme nous n'avons pas de GPU chez nous, nous utilisons le Google Colab Pro pour faire les expérimentations, car avec Colab Pro nous pouvons utiliser le GPU sans limitation temporelle et la taille du RAM est plus élevée (35 GB). Néanmoins, le Colab n'est pas tout le temps stable en tant qu'un outil IDE(Intergrated Development Environment) donc le temps d'entraînement est parfois trop long. Lors des expérimentations, nous avons quelquefois obtenu des mauvais résultats en testant les paramètres. Pour avoir de bons résultats à la fin avec un temps d'entraînement raisonnable, nous avons choisi d'autres valeurs pour faire les tests. De plus, pour enrichir nos analyses et s'assurer du bon choix du paramètre, nous avons implémenté et intégré dans le code les métriques Perplexité et BLEU.

5 Perspectives

Afin d'améliorer davantage la performance du modèle, nous avons pensé à utiliser d'autres word-embeddings pré-entraînés tels que GloVe[2][3], fastText[3][4] et word2vec[4]. Nous trouvons que ces word embeddings n'ont pas les mêmes structures que ceux de Remi Lebret[5]. De plus nous trouvons que les word embeddings utilisés dans ce modèle sont générés par un outil "***HPCA***", créé par Remi Lebret sous langage C, sur le corpus WikiProject Biography (WikiBio). Par conséquent, si on veut intégrer d'autres word embeddings, il nous faut absolument utiliser ***HPCA*** et une base de données qui ressemble à WikiBio. Malheureusement, nous n'avons pas assez de temps pour générer des nouveaux word embeddings lors de ce projet.

Pour pouvoir prouver la généralisation de modèle, nous utilisons souvent plusieurs bases de données et comparons leurs résultats. Comme Remi Lebret a créé cette base de données sur WikiBio en intégrant en plus les informations de tableaux (infobox), cette structure de base de données est vraiment inédite. Nous ne trouvons aucun jeux de données qui a la même structure sur Internet. Par ailleurs, nous n'avons non plus trouvé les scripts qui peuvent traiter les bases de données dans la structure que nous voulons. Nous essayerons de construire une telle base de données en écrivant nos scripts de traitement dans le futur travail.

References

[1] R. Lebret, D. Grangier, and M. Auli, "Neural Text Generation from Structured Data with Application to the Biography Domain ," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016.

[2] R. Lebret, D. Grangier, and M. Auli, "WikiBio (Wikipedia Biography Dataset) ," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016.

[3] C. D. M. Jeffrey Pennington, Richard Socher, "GloVe: Global Vectors for Word Representation ,"

[4] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *arXiv preprint arXiv:1607.04606*, 2016.

[2] Wikipedia 2014 + Gigaword 5 (6B tokens, 400K vocab, uncased, 50d, 100d, 200d, 300d vectors, 822 MB download)

[3] 1 million word vectors trained on Wikipedia 2017, UMBC webbase corpus and statmt.org news dataset (16B tokens).

[4] Pre-trained vectors trained on part of Google News dataset (about 100 billion words).

[5] L'auteur de Neural Text Generation from Structured Data with Application to the Biography Domain