

Rapport projet TLNL

Leader : Adrien ZABBAN
Follower : Yanis LABEYRIE

05 novembre 2023

1 Introduction

Le but de ce projet est de coder un modèle de langage basé sur un réseau de neurones multicouche. Ce modèle de langage devra permettre de prédire le mot suivant à partir d'un contexte, qui est un ensemble de mots précédant le mot à prédire.

Pour faire cela, on utilise des embeddings. Le principe est d'associer à chaque mot un vecteur dans un espace latent de telle sorte que deux mots similaires (en termes de sens) ont des vecteurs proches (avec une distance euclidienne), et deux mots totalement différents sont très loin. Cela permet de projeter les mots dans un espace latent pour pouvoir travailler plus efficacement sur ces mots. On note e la dimension de cet espace.

Dans un premier temps, on utilisera les embeddings des mots appris par l'algorithme Word2Vec [1] (modèle nommé *FROZEN*). Par la suite, nous tenterons d'améliorer le modèle en lui permettant d'apprendre ses propres embeddings à partir d'embeddings aléatoires (modèle nommé *SCRATCH*), ou directement les embeddings appris de Word2Vec (modèle nommé *ADAPT*).

2 Modèle *FROZEN*

Nous avons implémenté le perceptron multicouche comme modèle de langage de base. Celui-ci est composé d'une couche d'entrée qui prenant un vecteur de dimension $k \times e$ représentant les embeddings de k mots concaténés. Une couche de neurone cachée dont nous avons choisi de faire varier le nombre de neurones noté h , suivie d'une fonction ReLU [2]. Puis une deuxième couche de neurones suivie d'un softmax retournant un vecteur de taille V , le nombre de mots appris. La Figure 1 représente ce modèle.

2.1 Formalisme mathématiques

En notant, $W \in \mathcal{M}_{k \times e, h_1}(\mathbb{R})$, et $U \in \mathcal{M}_{h, V}(\mathbb{R})$ les matrices de poids, $(b_1, b_2) \in \mathbb{R}^h \times \mathbb{R}^V$ les biais, $X \in \mathbb{R}^{k \times e}$ le vecteur d'entrée, l'équation 1 donne la fonction de sortie $F(X) \in \mathbb{R}^V$ du modèle.

$$F(x) = \text{softmax}(U(\text{ReLU}(WX + b_1)) + b_2) \quad (1)$$

Après plusieurs essais, nous avons choisi de prendre : $k = 3$, $e = 100$, $h_1 = 256$. Et dans nos données d'entraînement, on avait un vocabulaire contenant :

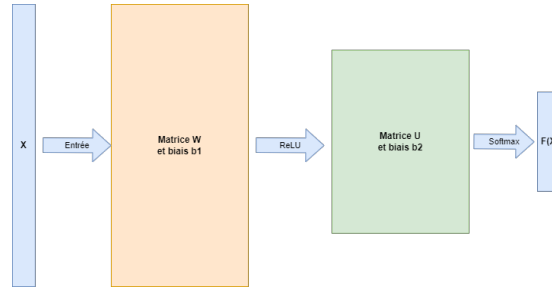


FIGURE 1 – Modèle : *FROZEN*

métriques	accuracy	top k	perplexité	f_1 score
modèle <i>FROZEN</i>	0.19	0.34	144	9.6e-4

TABLE 1 – Meilleures métriques de validation obtenues en fin d'apprentissage du modèle *FROZEN*.

$V = 2908$ mots distincts. Avec ces hyperparamètres, nous avons dans ce modèle 824412 paramètres apprenables.

Pour entraîner ce modèle, nous avons comparé notre sortie au mot à prédire dans le format One-hot encoding¹, et nous appliquons le calcul de la fonction de coût : cross-entropy. On a utilisé Adam [3] pour optimiser les paramètres avec un learning rate de 0.01 pour les 5 premières époques et 0.001 pour les suivantes.

2.2 Métriques

Nous avons par ailleurs décidé d'évaluer ce réseau à l'aide de plusieurs métriques comme l'accuracy, la perplexité, le f_1 -score et la métrique "top k "² (avec $k = 5$), voir [4].

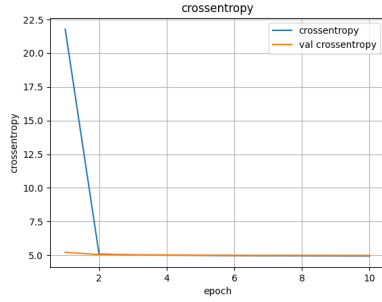
2.3 Résultats

Sur la Figure 2, on voit les courbes d'apprentissage. Les valeurs des métriques sur les données d'entraînement sont en bleue, sur les données de validations sont en orange.

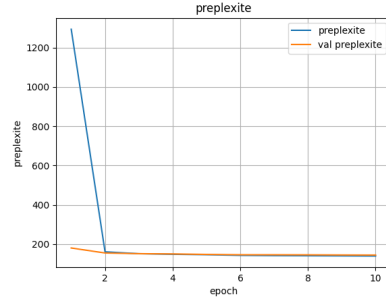
On constate d'après les courbes que le modèle apprend très vite (les courbes atteignent un plateau en quasiment 2 epochs). Ceci est dû au fait que les embeddings du modèle sont déjà appris. On observe par ailleurs que le modèle n'overfit pas car il n'y a pas de décalage important entre les valeurs de métrique d'entraînement et de validation à la fin de l'entraînement. Le modèle se stabilise à la fin de l'entraînement avec les métriques de validation présenté dans la Table 1.

1. Le format One-hot encoding est un encodage des indices en un vecteur d'une taille du nombre d'indice possible tel que ce vecteur possède des 0 partout sauf à l'indice en question qui possède un 1.

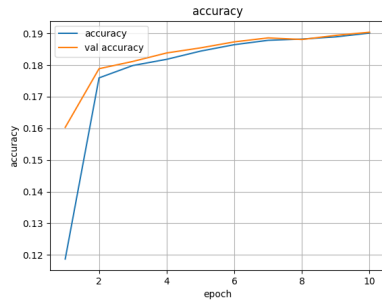
2. La métrique "top- k " évalue un modèle en comptant combien de prédictions correctes il fait parmi les k premières prédictions les plus probables. Attention ici k n'est pas le nombre de mots dans le contexte!



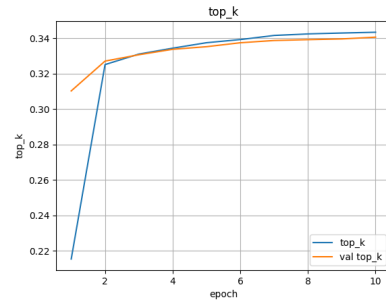
(a) Cross Entropy Loss (Lower is Better)



(b) perplexité (Lower is Better)



(c) Accuracy (Higher is Better)



(d) top k, avec k=5 (Higher is Better)

FIGURE 2 – Entraînement du modèle *FROZEN*

3 Apprentissage automatique des embeddings par le modèle de langage : modèle *SCRATCH*

3.1 Description

Cette première piste que nous avons exploré consiste à se dire que notre modèle d'embeddings pré-entraîné n'est pas forcément pertinent pour la tâche de prédiction du mot suivant d'une phrase. L'idée est donc de considérer que la matrice d'embeddings peut-être apprise par le modèle de langage et que la matrice apprise sera plus pertinente qu'une matrice apprise séparément pour résoudre la tâche. On va donc considérer que les paramètres de la matrice sont des paramètres apprenables du modèle de langage et donc étendre jusqu'à eux l'algorithme de rétro-propagation du gradient.

3.2 Mise en œuvre

Il a fallu adapter un peu notre code car avant pour avoir l'embedding d'un mot i , on utilisait $E[i]$, où $E \in \mathcal{M}_{V,e}(\mathbb{R})$ est la matrice d'embedding. Cependant, cette opération n'est pas dérivable, donc il a fallu transformer le mot i en un vecteur $x \in \mathbb{R}^V$ one-hot, et faire $x \times E$ pour obtenir l'embedding du mots i , qui cette fois-ci, est une opération dérivable. Une fois cette adaptation faite, on a rajouté cette opération dans notre modèle et nous obtenons donc un nouveau modèle illustré par la Figure 3, et donné par l'équation 2.

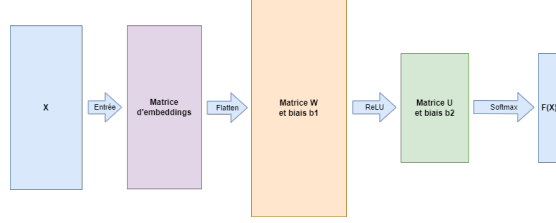


FIGURE 3 – Modèle : *SCRATCH* avec l'apprentissage des embeddings

modèle	accuracy	top k	perplexité	f_1 score
<i>FROZEN</i>	0.19	0.34	144	9.6e-4
<i>SCRATCH</i>	0.195	0.333	280.37	1.27e-3

TABLE 2 – Meilleures métriques de validation obtenues en fin d'apprentissage des modèles *FROZEN* et *SCRATCH*.

$$F(x) = \text{softmax}(U(\text{ReLU}(W\tilde{X} + b_1)) + b_2) \quad (2)$$

où $\tilde{X} = \text{flatten}(XE)$ et $X \in \mathcal{M}_{k,V}(\mathbb{R})$.

3.3 Résultats

On remarque, sur la Figure 4, tout d'abord que l'apprentissage à la fois du modèle et des embeddings est plus long car l'ensemble (modèle plus matrice d'embedding) possède un plus grand nombre de paramètres (1115212 contre 824412 pour le modèle *FROZEN*) qui nécessitent donc plus d'epochs que *FROZEN* pour être actualisés.

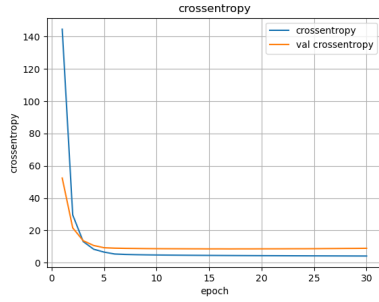
On obtient en termes de métriques de validation, des résultats semblables au modèle précédent. Cela s'explique par le fait que, bien qu'on apprenne par rétro-propagation les embeddings et que donc on s'attend à une meilleure qualité de résultat, cela est compensé par le fait que l'apprentissage se fait à partir de 0 et on commence donc l'apprentissage avec des résultats aléatoires.

A la fin de l'apprentissage, nous obtenons les métriques de validation suivantes, comme on peut le voir dans la Table 2.

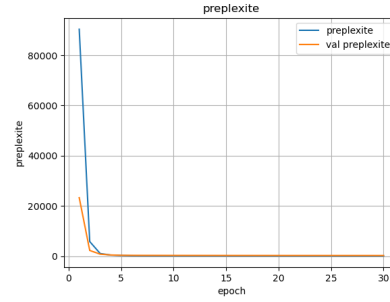
4 Apprentissage des embeddings à partir de l'algorithme Word2Vec : modèle *ADAPT*.

4.1 Description

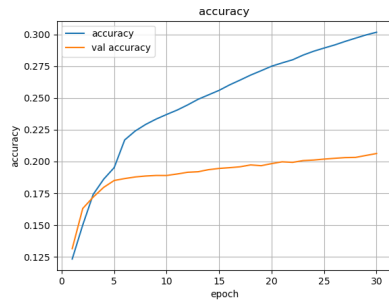
Dans cette partie, nous allons tenter d'améliorer la performance et surtout la vitesse d'apprentissage. Au lieu, d'initialiser la matrice d'embedding aléatoirement comme dans la partie précédente, nous allons initialiser cette matrice à partir des embeddings déjà appris avec l'algorithme Word2Vec. En procédant comme cela, nous allons pouvoir continuer d'apprendre ces embeddings pour coller au mieux à notre tâche de prédiction.



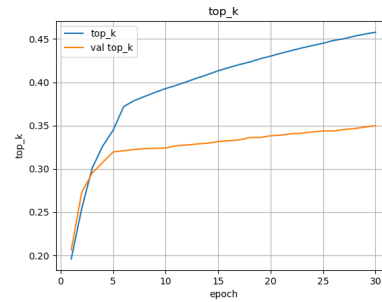
(a) Cross Entropy Loss (Lower is Better)



(b) perplexité (Lower is Better)



(c) Accuracy (Higher is Better)



(d) top k, avec k=5 (Higher is Better)

FIGURE 4 – Entraînement du modèle *SCRATCH*

4.2 Mise en œuvre

La mise en œuvre de cette partie n'a pas été très compliquée à implémenter car le modèle est le même que celui de *SCRATCH*, voir Figure 3. Il nous a donc suffi de récupérer la matrice d'embedding apprise et de la copier dans le modèle lors de son initialisation.

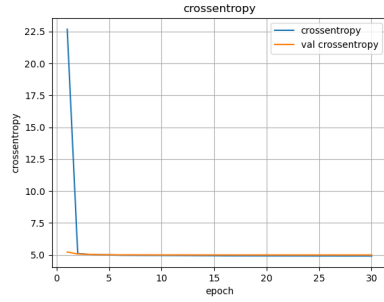
4.3 Résultats

On constate immédiatement, avec la Figure 5, que l'apprentissage est bien plus rapide dans le cas où on initialise les embeddings avec Word2Vec par rapport au cas où on les initialise aléatoirement. Cela s'explique par le fait que contrairement au modèle précédent, le modèle a déjà des embeddings valides aux début de l'apprentissage qu'il peut par la suite perfectionner.

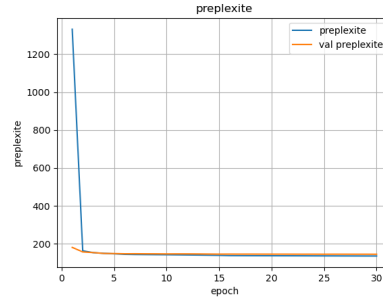
Notre hypothèse était que initialiser les embeddings avec Word2Vec puis les modifier au cours de l'apprentissage permettrait d'obtenir de meilleures performances que celles du modèle où les embeddings sont gelés pendant l'apprentissage.

5 Génération de texte des modèles

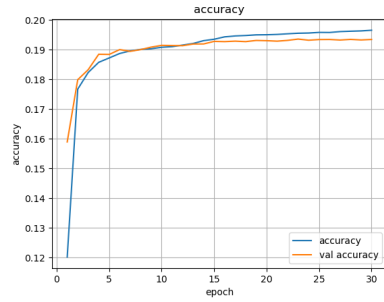
Après les entraînements des modèles, nous avons généré du texte avec les modèles à partir d'une entrée (input). Nous avons choisi arbitrairement un input



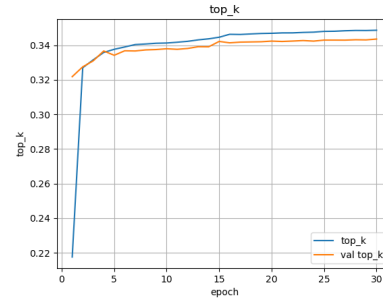
(a) Cross Entropy Loss (Lower is Better)



(b) perplexité (Lower is Better)



(c) Accuracy (Higher is Better)



(d) top k, avec k=5 (Higher is Better)

FIGURE 5 – Entraînement du modèle *ADAPT*

qui est : *le vieillard fit un* car il fait partie du corpus de données d'entraînement. Nous avons ensuite généré les 10 mots suivants grâce aux modèles. La Table 3 représente les résultats obtenues.

modèles	génération
<i>FROZEN</i>	le vieillard fit un ,,,,,,,,,,
<i>SCRATCH</i>	le vieillard fit un certains,,,,,,,,,
<i>ADAPT</i>	le vieillard fit un ,,,,,,,,,,

TABLE 3 – Résultat des générations de 10 mots des modèles à partir de : *le vieillard fit un*.

On constate que les mots suivants prédits par notre modèle sont peu pertinents et que surtout le modèle renvoie plusieurs fois le même mot à la suite, ici le mot ','. Ceci peut s'interpréter comme un phénomène de *mode collapse*, c'est à dire que le modèle présente peu de variété dans ses sorties. En effet le mot virgule étant très fréquent dans le corpus d'entraînement, il est normal qu'un modèle peu performant ne parvienne à générer que ce mot là.

Le problème vient de la façon dont on choisi le mot à générer à chaque étapes : on prend le mot m^* tel que $m^* = \operatorname{argmax}_{m \in V} f(X)_m$, où f représente le modèles choisie et $f(X)_m$ est la propabilité du mot m connaissant le contexte X . Pour tenter de régler ce problème, à chaque étape, on prend les 5 mots qui sont les plus prédits (ie les 5 mots qui maximine $f(X)_m$), et l'on choisi de prendre un

de ces 5 mots avec la loi : $P(m_i|X) = \alpha f(X)_{m_i}$, où $\alpha = 1/\sum_{i=1}^5 f(X)_{m_i}$. On rajoute aussi une *temperature* $t \in]0, 1]$ à la fonction softmax dans les équations 1 et 2, décrite par la formule 3. Cela permet de rendre la distribution $f(X)$ plus homogènes et donc de générer un mot m différent que m^* .

$$\text{softmax}(X, t)_i = \frac{e^{X_i/t}}{\sum_j e^{X_j/t}} \quad (3)$$

On a donc ensuite fait de nouvelles générations de textes avec les 3 modèles. La Table 4 montre ces générations³. On observe une plus grande variété de mots prédits, comme des points ou des symboles de fin de génération ($</s>$). On remarque aussi que les points sont toujours suivit de symboles $</s>$, ce qui est normal car la base de données est une liste de phrase et non un grand texte. Cependant on voit quand même les limites des 3 modèles sur la génération sur le peu de sens de la phrase générée.

modèles	génération
<i>FROZEN</i>	le vieillard fit un , . $</s>$
<i>SCRATCH</i>	le vieillard fit un signe à la et . $</s>$
<i>ADAPT</i>	le vieillard fit un de , . $</s>$

TABLE 4 – Résultat des générations de 10 mots des modèles à partir de : *le vieillard fit un*, avec une température de 0.6

6 Conclusions et perspectives

6.1 Performance

Après avoir entraîné ces trois modèles et pris les poids qui minimise la loss de validation, nous avons ensuite passé ces modèles dans une base de données de teste⁴. La Table 5 présente les résultats des modèles sur différentes métriques. On voit que les 3 modèles ont une accuracy et une métrique top k similaires. Cependant, le modèle *SCRATCH* est très mauvais sur la Cross Entropy et la perplexité (par rapport aux autres modèles). Il est aussi étonnant de constater que le modèle *FROZEN* et *ADAPT* ont des résultats similaires. Cela signifie que les embeddings appris par Word2Vec correspondent à cette tâche. Dans ce cas, les embeddings obtenues avec Word2Vec ont été appris sur le même corpus que l'entraînement des modèles. Nous pensons que si les embeddings avaient été appris sur un autre corpus, le modèle *ADAPT* aurait pu se différencier par rapport au modèle *FROZEN*.

6.2 Temps d'entraînement

Il peut aussi être intéressant de regarder le temps d'entraînement des différents modèles, voir Table 6. Ces modèles ont été entraînés sur un Windows 11

3. Attention, il faut noter que cette nouvelle méthode de génération est stochastique et que par conséquent, les résultats présentés ne sont qu'une génération possible parmi l'ensemble des générations.

4. La base de données est bien sûr disjointe de celles de l'entraînement et de validation, mais toujours tirée du corpus du *Comte de Monté-Cristo* d'Alexandre Dumas.

modèles	Cross Entropy	accuracy	top k	f_1 score	perplexité
<i>FROZEN</i>	4.89	0.20	0.35	$9.42e - 4$	131
<i>SCRATCH</i>	8.05	0.20	0.35	$1.16e - 3$	247
<i>ADAPT</i>	4.90	0.20	0.35	$8.86e - 4$	130

TABLE 5 – Résultat des différents modèles sur la base de données de teste.

avec un intel i7 de 12ème génération et une carte graphique : Nvidia RTX 3050 modèle ordinateur portable. On remarque sans surprise que l’entraînement le plus rapide est celui de *FROZEN* car c’est le modèle qui a le moins de paramètres. Il est très intéressant de voir que le modèle *ADAPT* s’entraîne beaucoup plus vite que le modèle *SCRATCH*, alors que le nombre de paramètres est le même. Notre hypothèse est que les poids de la matrice d’embeddings étant déjà pertinents, peut-être que lors de la rétro-propagation certains gradients valent dès le début 0 et ne sont donc pas propagés, ce qui fait que la rétro-propagation prend moins de temps. En comparant les résultats des modèles et leurs temps d’entraînement, on peut voir qu’il n’est pas rentable d’entraîner un modèle comme celui de *SCRATCH*, si l’on a déjà une partie pré-entraîné car cela prend beaucoup de temps et que l’on des performances plus faibles.

modèles	<i>FROZEN</i>	<i>SCRATCH</i>	<i>ADAPT</i>
temps (en s)	360	1001	460
nombre de paramètres	800k	11M	11M

TABLE 6 – Comparaison du temps d’entraînement des différents modèles sur 10 epochs et nombre de paramètres apprenables des modèles.

Références

- [1] T. MIKOLOV, K. CHEN, G. CORRADO et J. DEAN, « Efficient estimation of word representations in vector space, » *arXiv preprint arXiv :1301.3781*, 2013.
- [2] A. F. AGARAP, « Deep Learning using Rectified Linear Units (ReLU), » *CoRR*, t. abs/1803.08375, 2018. arXiv : [1803.08375](https://arxiv.org/abs/1803.08375). adresse : <http://arxiv.org/abs/1803.08375>.
- [3] D. P. KINGMA et J. BA, « Adam : A method for stochastic optimization, » *arXiv preprint arXiv :1412.6980*, 2014.
- [4] L. LIU, T. G. DIETTERICH, N. LI et Z. ZHOU, « Transductive Optimization of Top k Precision, » *CoRR*, t. abs/1510.05976, 2015. arXiv : [1510.05976](https://arxiv.org/abs/1510.05976). adresse : <http://arxiv.org/abs/1510.05976>.