Rapport projet 3 TLNL : Modèle de langage neuronal

Leader : Joey SKAF Follower : Emma MENDIZABAL

Pour le 5 novembre 2023

Les modèles de langage neuronal (MLN) sont au confluant de plusieurs domaines de l'intelligence artificielle. Ils se basent sur le modèle de *Markov*, où la probabilité d'un mot dépend de son contexte "proche", sur les embeddings d'un mot, i.e. sa représentation sous la forme d'un vecteur qui encode le contexte dans lequel "vit" le mot, et les réseaux de neurones denses du Deep Learning. Ce rapport cherche à rendre compte de notre travail sur ce modèle. Nous nous attacherons, tout d'abord, à introduire nos différentes approches sur l'implémentations d'un tel modèle, puis nous nous pencherons plus précisément sur les différents étapes de la conception de ce modèle et leurs résultats pour enfin conclure et proposer des perspectives d'amélioration.

1 Introduction

Un modèle langage a pour but de calculer la probabilité d'une séquence de mots (phrases, textes, corpus etc.), dans le but de réaliser des tâches automatisées de langage comme la génération de texte ou encore la traduction, en sélectionnant la séquence de mots "la plus probable" à retourner. Si l'on note $P(m_i^n)$, la probabilité d'une séquence de n mots, alors

$$P(m_i^n) = \prod_{i=1}^{n} P(m_i|m_1^{i-1})$$

On se penche donc sur le calcul des $P(m_i|m_1^{i-1})$. L'hypothèse de *Markov* permet de simplifier cette quantité : on fait l'approximation qu'un mot m_i ne dépend que de son voisinage proche. Soit k, la longueur du voisinage antérieur du mot considéré, il vient que :

$$P(m_i|m_1^{i-1}) \approx P(m_i|m_{i-1}, ..., m_{i-k})$$

Notons, pour le reste du rapport, V, le vocabulaire du texte considéré, i.e. l'ensemble des mots distincts d'un texte.

Représenter un mot est une questions cruciale dans l'implémentation du modèle de langage et le calcul des probabilités ci-dessus. Si un mot peut être considérer de prime abord, par une chaîne de caractère ou son nombre d'occurrences dans un texte, on peut aussi vouloir le représenter sous forme d'un vecteur, que l'on appellera dans la suite de ce papier de *plongements*. L'idée est de palier le problème d'un encodage brut d'un mot qui consisterait pour tout

le vocabulaire d'un corpus, de mettre un 1 lorsque le mot cible et un mot du vocabulaire apparaissent dans une même fenêtre de contexte. On choisit de garder cette idée d'encodage du contexte d'un mot, en compressant la dimension du vecteur à un espace plus petit de dimension d.

Un mot est donc représenté par un *plongement* qui est un vecteur "compressé" qui encode le contexte dans lequel vit un mot pour un corpus donné, et par extension introduit une notion de sémantique dans la représentation du mot.

Le calcul de tels vecteurs peut se faire à l'aide de l'algorithme *Skip-gram with Negative Sampling*. Nous ne rentrerons pas dans le détail de cet algorithme dans ce projet.

Soit E, la matrice des *plongements* des mots du vocabulaire de notre corpus. Il vient alors que $[\mathbf{E}]_{i,.}$ est le plongement du i-ème mot de notre vocabulaire. Soit x_i le *one-hot* vecteur ligne du i-ème mot de notre vocabulaire. Il s'ensuit que : $x\mathbf{E} = [\mathbf{E}]_{i,.}$.

2 Modèle Initial

Pour calculer la probabilité $P(m_i|m_1^{i-1})$, on utilise un réseau de neurones dense, à une couche cachée $\mathbf{h}$. La couche d'entrée est ici $\mathbf{x}$ et celle de sortie est $\mathbf{y}$.

Soit m_t un mot cible, x_t son *one-hot* vecteur, $x_{t-i}, ..., x_{t-k}$, les *k-one-hot* vecteur antérieur du mot cible dans l'hypothèse de *Markov*, pour i allant de 1 à k. On donne comme entrée $\mathbf{x}$ à notre réseau de neurones

$$\mathbf{x} = [x_{t-i}\mathbf{E}; ..., x_{t-k}\mathbf{E}]$$

la concaténation des vecteurs $x_{t-i}\mathbf{E}$.

Le feedforward du réseau neuronal donne un $\mathbf{y}$ tel que :

$$\mathbf{y} = softmax(Relu(\mathbf{x}\mathbf{W} + \mathbf{b}_1)\mathbf{U} + \mathbf{b}_2)$$

Avec $\mathbf{W}, \mathbf{b}_1, \mathbf{U}$ et $\mathbf{b}_2$ les paramètres d'apprentissage de notre réseau.

$\mathbf{y}$ est alors une distribution de probabilité sur tous les mots du vocabulaire, tel que $[\mathbf{y}]_i$ est la probabilité du i-ème mot de notre vocabulaire d'être à la suite des k mots du contexte considéré.

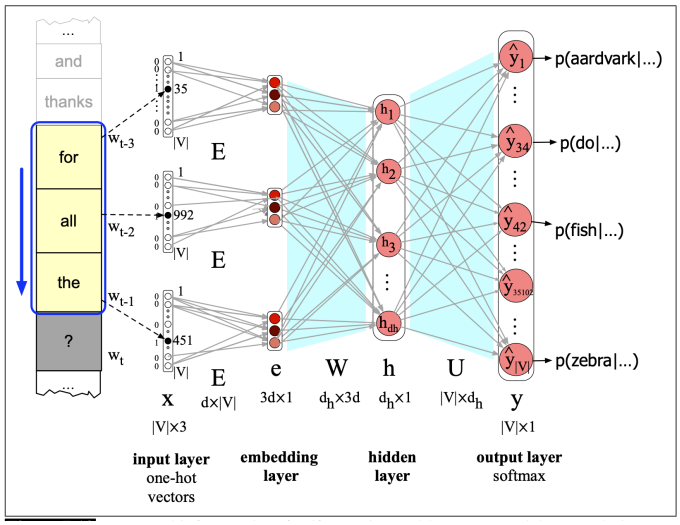
On mesure alors l'erreur entre la distribution prédite et réelle par une Loss Cross Entropy, on minimise notre loss par la méthode d'optimisation du gradient d'Adam et on rétropropage l'erreur pour mettre à jour nos paramètres.

On définit pour la suite du projet les hyperparamètres suivants :

— $\mathbf{d}$: la dimension des *plongements*
— $\mathbf{d}_h$: la dimension de la couche cachée $\mathbf{h}$
— $\mathbf{k}$: le nombre de mot du contexte du mot cible
— $|\mathbf{V}|$: le cardinal de l'ensemble du vocabulaire du texte
— η : le taux d'apprentissage initial
— $\mathbf{e}$: le nombre d'epoch pour minimiser notre loss

Nous utiliserons les bibliothèques de python pour réaliser ce MLN, notamment *pytorch*. Enfin, l'apprentissage se fera par mini-batch, pour profiter du parallélisme des calculs qu'offre *pytorch*. La taille de ces batchs, noté $\mathbf{b}$, sera d'ailleurs sujette à expérimentation pour la fixer.

FIGURE 1 – Graphique synthétisant notre modèle de langage neuronal. Lien : https ://web.stanford.edu/ jurafsky/slp3/7.pdf



Une fois le modèle entraîné sur un texte *train*, nous testons ses performances à travers le calcul de perplexité d'un texte *test*, et nous pouvons aussi générer du texte à partir du début du texte *test*. Pour cette dernière tâche, nous décidons d'afficher : le mot attendu avec sa probabilité, le mot le plus probable et le mot prédit avec sa probabilité. D'ailleurs, nous prédisons un mot en échantillonnant sur la distribution prédite, que l'on décide d'adoucir ou d'accentuer en divisant par une "température" θ. Pour finir, l'hyperparamètre **l** permet de selectionner les t l plus grandes valeurs de la distribution pour plus ou moins accélérer le processus de génération de texte.

2.0.1 Résultats

Malgré des difficultés dans l'implémentation de la partie ci-dessus, nous réussissons à entraîner notre MLN. On ne construit pas E de manière claire, mais on récupère les *plongements* des mots à travers un objet python V qui a pour méthode de les récupérer d'un fichier où ils sont stockés. Le reste est codé explicitement grâce à la bibliothèque *pytorch*. Afin de choisir les hyperparamètres les plus optimaux pour l'apprentissage, nous les faisons varier une à une en fixant les autres et nous comparons à chaque la courbe d'évolution de la fonction de perte. Ci-dessous FIGURE 2, les graphiques synthétisant ces expérimentations. Il en résulte qu'une bonne première initialisation de nos paramètres est de prendre $\mathbf{d}_h = 200$ et $\eta = 0.00005$. Pour la taille des mini-batchs, un bon compromis entre le temps d'exécution et un bon apprentissage est de prendre $\mathbf{b} = 8$. Enfin, quelques tests encore sur le nombre de mot de contexte à prendre ou encore le nombre d'*epoch* surlequel entraîné notre MLN (FIGURE 3) permet de conclure sur initialiser $\mathbf{k} = 3$ et $\mathbf{e} = 50$.

En revanche, le calcul de la perplexité du texte *test*, donne des résultats plutôt décevant : elle est de 350.00 pour 99.96% des mots du texte. Le reste ayant une probabilité de 0, les compter donnerait une perplexité infinie. A titre de

comparaison, la perplexité du texte avec un modèle *trigram* donne une perplexité de 557.27 pour tous les mots du texte *test*. Pour finir, la génération est tout aussi limitée, les mots qui s'enchaînent n'ont pas voire peu de lien, et les mots attendus ont des probabilités faibles (FIGURE 3).

Une façon d'améliorer notre système pourrait être de rajouter des paramètres d'apprentissage : la surparamétrisation est une spécificité du Deep Learning qui, contrairement à ce qui est prévu par la théorie du Machine Learning, permet de meilleurs résultats en généralisation.

Un moyen d'augmenter ce nombre de paramètres est d'apprendre **E**, la matrice des *plongements*. On se propose donc de l'inclure dans notre réseau de neurones, ce qui sera l'objet des pistes explorées. Nous nous proposons tout d'abord d'implémenter l'inclusion de **E** dans l'apprentissage de notre MLN, puis de comparer ce **E** obtenu avec la matrice des *plongements* construite par la méthode SGNS, que l'on notera $\mathbf{E}_{SGNS}$ sur une tâche de similarité (savoir dire que 2 mots proches sont similaires face à deux mots indépendants sémantiquement parlant), de comparer les mots proches au sens de la matrice **E** obtenue avec ceux de la matrice $\mathbf{E}_{SGNS}$.

2.1 Piste à creuser 1 : Apprentissage de E dans le MLN

2.1.1 Description

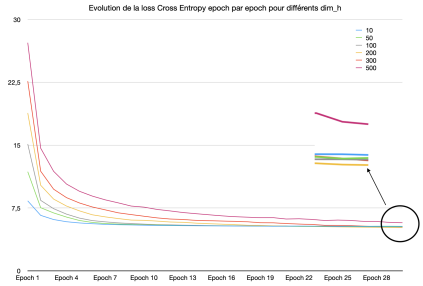
Cette première piste consiste en l'apprentissage de **E** dans notre MLN. On décide dans un premier temps d'initialiser cette dernière matrice de manière aléatoire, puis le reste du programme d'apprentissage change peu : le forward inclut désormais **E** de manière explicite. On cherche en fait à savoir si procéder de cette manière permet un gain dans les résultats de perplexité et de génération.

2.1.2 Mise en œuvre

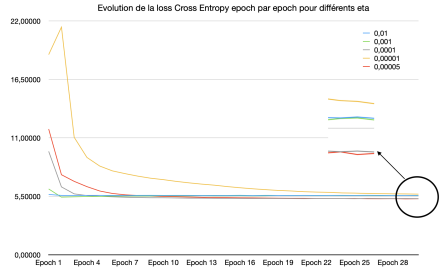
Comme dit précédemment, l'ajout au modèle de **E** se fait par ajout d'une ligne de concaténation des lignes de E explicite pour le forward ensuite dans le réseau de neurone. On rajoute ensuite aussi E dans notre *optimizer* (ce que j'ai oublié de faire pendant un moment, j'obtenais des résultats pour un E initialisé de manière aléatoire. J'en parlerai plus en détails dans la seconde piste à creuser). Aussi, j'ai décidé de modifier mes hyperparamètres pour accélérer l'entraînement tout en ayant essayé de rester proche de ce que j'utilisais pour un *forward* sans **E**.

2.1.3 Résultats

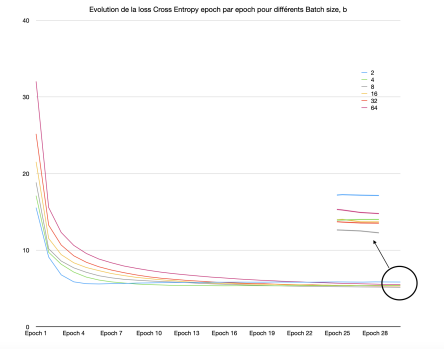
On observe FIGURE 4, l'évolution de la fonction de perte pour différentes configurations. Premièrement, on se rend compte que l'on apprend "mieux" en incluant **E** dans l'entraînement, la fonction de perte est mieux optimisée qu'en figeant la matrice des *plongements* à une initialisation *SGNS*. En revanche, les perplexités que l'on calcule sur le fichier *test* reste à désirer, voire performe moins bien (TABLE 1). La génération n'est pas meilleure. On ne valide donc pas notre hypothèse de meilleure performance. Pour comprendre ces piètres résultats, la piste 2 apporte quelques éclaircissements.



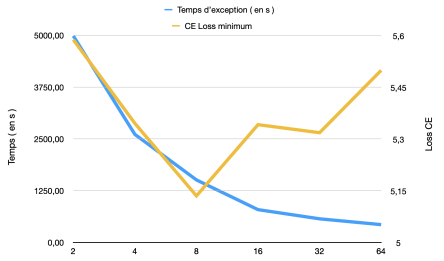
(a) Hyperparamètres fixés choisis : $\mathbf{k} = 3$, $\mathbf{d} = 100$, $\mathbf{b} = 8$, $\mathbf{e} = 30$, $\eta = 0.00005$.



(b) Hyperparamètres fixés choisis : $\mathbf{k} = 3$, $\mathbf{d} = 100$, $\mathbf{b} = 8$, $\mathbf{e} = 30$, $\mathbf{d}_h = 50$.

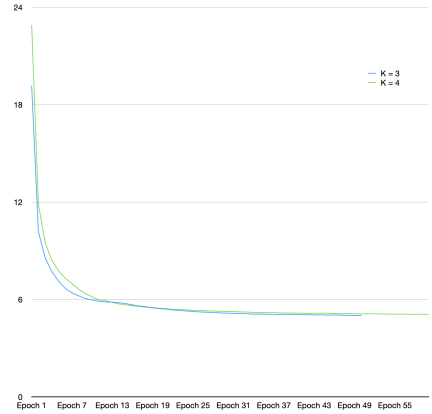


(c) Hyperparamètres fixés choisis : $\mathbf{k} = 3$, $\mathbf{d} = 100$, $\eta = 0.00005$, $\mathbf{e} = 30$, $\mathbf{d}_h = 200$.

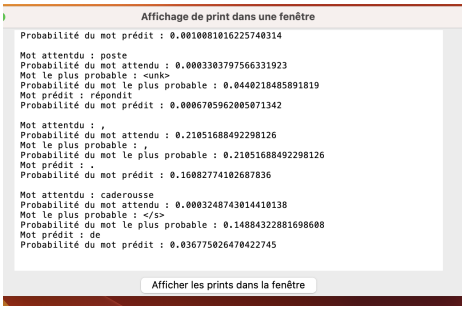


(d) Hyperparamètres fixés choisis : $\mathbf{k} = 3$, $\mathbf{d} = 100$, $\eta = 0.00005$, $\mathbf{e} = 30$, $\mathbf{d}_h = 200$.

FIGURE 2



(a) Hyperparamètres fixés choisis : $\mathbf{b} = 8$, $\mathbf{d} = 100$, $\eta = 0.00005$, $\mathbf{e} = 60$, $\mathbf{d}_h = 200$. Plus que le petit gain que l'on observe avec $\mathbf{k} = 3$, les temps d'exécutions différent aussi : 2512.92 s pour ce premier hyperparamètre contre 2936.20 s pour $\mathbf{k} = 4$.



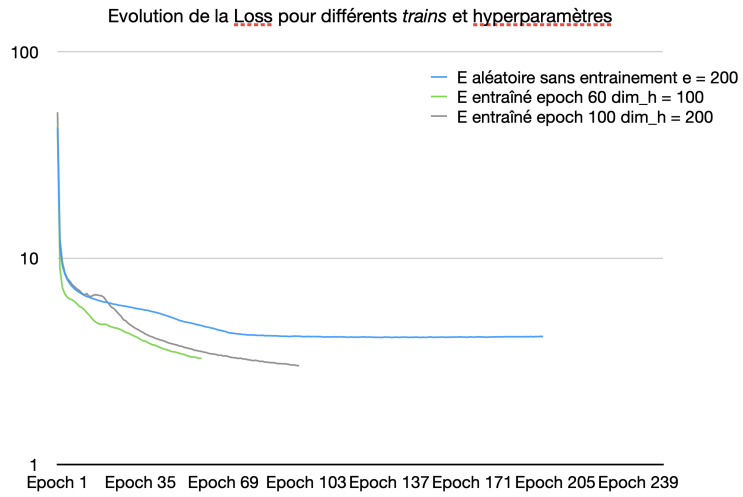
(b) Capture d'écran de la fenêtre d'affichage de la génération de texte de notre modèle. On observe que les probabilités de prédiction des bons mots sont assez faibles.

FIGURE 3

Perplexité pour différents Modèles de Langage			
Trigram	MLN **E** figé	MLN **E** aléatoire figé	**E** aléatoire appris
557.27	350.00 pour 99.96% des mots du texte	681.98	1975.01 pour 99.97 % du texte

TABLE 1 – Hyperparamètres : pour la 2ème colonne $\mathbf{k} = 3$, $\mathbf{d} = 100$, $\mathbf{b} = 8$, $\mathbf{e} = 50$, $\mathbf{d}_h = 200$, $\eta = 0.00005$; pour la 3ème et 4ème colonne $\mathbf{k} = 3$, $\mathbf{d} = 100$, $\mathbf{b} = 8$, $\mathbf{e} = 200$, $\mathbf{d}_h = 100$, $\eta = 0.0001$.

FIGURE 4 – Hyperparamètres fixés choisis : $\mathbf{k} = 3$, $\mathbf{d} = 100$, $\eta = 0.0001$. Échelle logarithmique en ordonnée. Le minimum de la fonction de perte atteint est respectivement de gauche à droite : 3.27, 3.01, 4.23 .



2.2 Piste à creuser 2 : Comparaison du E obtenu par le MLN et celui obtenu par l'algorithme SGNS

2.2.1 Description

On s'intéresse à comparer les deux matrices de *plongements* à travers, dans un premier temps, une tâche de similarité de mots que l'on explicitera dans la partie suivante : fait-on mieux en apprenant **E** avec notre MLN ou par l'algorithme SGNS ? Dans un deuxième temps, en prenant un mot du vocabulaire, nous évaluerons la similarité des contextes donnés par ces deux matrices pour ce même mot et on se pose donc la question : est-ce que les contextes s'intersectent de manière conséquente ? Enfin, en projetant en 2 dimensions grâce à la méthode t-SNE, on peut essayer de visualiser ces deux matrices et comparer ces projections.

2.2.2 Mise en œuvre

Pour rappel, le *plongement* d'un mot encode la sémantique d'un mot, le contexte dans lequel le mot vit pour un corpus donné. Deux mots "vivant" dans un même contexte ont de grande chance d'être similaires. Par exemple, pour un corpus à la thématique animalière par exemple, on s'attend à ce que *animaux* et *mammifères* soient plus proches entre eux qu'avec le mot *table*. On mesure la similarité grâce au cosinus similarité entre deux vecteurs. Soient $\mathbf{v}_1$ et $\mathbf{v}_2$ deux vecteurs, le cosinus similarité de deux vecteurs s'écrient :

$$\cos(\mathbf{v}_1, \mathbf{v}_2) = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\|\|\mathbf{v}_2\|}$$

Plus cette quantité est proche de 1, plus les vecteurs sont dits similaires. On

peut donc comparer nos deux matrices de *plongement* avec un fichier contenant deux mots similaires et un troisième hors contexte et on compte pour chaque matrice le nombre de fois ou la similarité entre les deux premiers mots est supérieure à celle entre le premier et le 3ème mot.

Pour la deuxième comparaison, on se propose de l'implémenter de la manière suivante. Soit m, un mot du vocabulaire, v_m, son *plongement*, on calcule $dist(v_m, [\mathbf{E}]_{i,.})$, pour tout i dans $1, ..., |\mathbf{V}|$, et on récupère les mots correspondant aux K plus petites valeurs, où K est choisi arbitrairement (ici on prend $K = 20$). On définit ainsi les mots de plus proche voisinage de m au sens de $\mathbf{E}$ et $\mathbf{E}_{SGNS}$. On peut faire ainsi l'intersection des deux ensembles et obtenir le nombre de mot voisins communs pour chaque mot du vocabulaire.

Enfin, pour implémenter la projection t-SNE, on utilise la bibliothèque du python *sklearn*, on convertit nos *torch tensors* en tableaux numpy pour utiliser la méthode *TSNE()*. On rappelle que la méthode TSNE est une méthode de projection non-supervisée qui consiste à estimer la distance entre deux points dans un espace plus petit, en calculant une distribution de probabilité. La "grandeur" que l'on souhaite considéré du voisinage d'un point se fixe par un choix d'un proxy d'écart-type σ, que l'on calcule en fait à travers une *perplexité* voulue et fixée en amont. En d'autres termes, du fait de cette non supervision, il faut visualiser cette projection pour plusieurs *perplexités* pour essayer d'interpréter quelque chose. C'est ce qu'on se propose de faire ci-dessous, de projeter les deux matrices de *plongements* en deux dimension pour plusieurs *perplexités*, et observer le graphique ainsi obtenue.

2.2.3 Résultats

Comme j'ai pu le mentionner précédemment, mes premiers résultats sont faussés car je n'avais pas écrit dans l'*optimizer* que $\mathbf{E}$ devait être appris. Après avoir corrigé cela, les résultats sur la similarité était identique à un $\mathbf{E}$ initialisée aléatoirement puis figée à l'entraînement (FIGURE 5). En revanche, le nombre de voisinages communs est légèrement meilleur pour un $\mathbf{E}$ appris (FIGURE 6). Lorsque l'on affiche le *plongement* du mot *</s>* pour ces différents $\mathbf{E}$, on se rend compte en réalité du problème : les valeurs ne sont pas du tout à la même échelle, comme si $\mathbf{E}$ n'était pas mise à jour à chaque itération, comme si $\mathbf{E}$ en réalité était initialisée aléatoirement mais non appris dans tous les cas (voir fichier *"embbeding </s>.txt"*).

Cette remarque est encore plus flagrante lorsque l'on décide d'apprendre $\mathbf{E}$ en l'initialisant avec $\mathbf{E}_{SGNS}$ (piste 3, que j'ai décidé de tester pour comparer avec mes résultats). Avec une perplexité assez médiocre (2536.60 pour 99.71%) et une génération pas plus performante, on remarque une même performance sur la tâche de similarité, un plus grand nombre de voisinages communs, mais surtout un *plongement* du mot *</s>* identique (FIGURE 5 et 6).

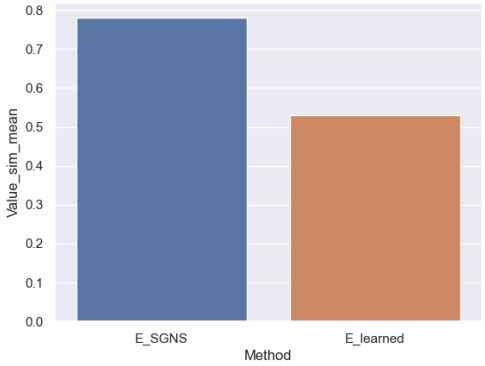
Nous supposons donc que l'on observe une évanescence du gradient : le réseau est tellement profond que le gradient, au cours de sa *backpropagation*, tend vers 0, pour ne plus avoir d'effet de mise à jour sur les premiers paramètres du réseau neuronal. Pour résoudre ce problème, on pourrait sûrement ajouter, de manière classique en *Deep Learning*, un couche de *Batch Normalization* ou encore rajouter des couches résiduelles : introduire les premiers paramètres directement dans les couches supérieures pour qu'ils puissent bénéficier de la descente de gradient. Une première façon de le faire serait, à notre sens d'ajouter avant le *softmax*,

un vecteur qui moyenne sur les colonnes de la matrice de *plongements* $\mathbf{E}$, pour obtenir un vecteur de même dimension que $\mathbf{y}$. Cette opération "moyenne" se ferait en multipliant par un vecteur β, que l'on apprendrait aussi au cours de l'entraînement. Cela est motivé par le fait que l'on veut que $\mathbf{E}$ puisse bénéficier de la descente de gradient, une manière de l'introduire dans la couche suivante est de le mettre à la dimension de $\mathbf{y}$ de taille $|\mathbf{V}|$.

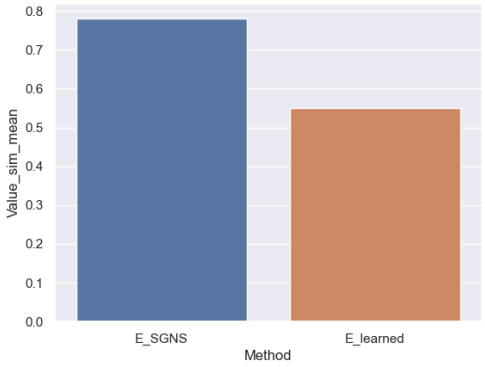
Enfin, la projection t-SNE (FIGURE 7) montre que tous les mots restent relativement proches entre eux. Il est difficile d'en dégager une tendance, ou un motif interprétable. Peut-être que le petit pique que l'on observe pour un $\mathbf{E}$ initialisé et appris pour de grandes perplexités et à mettre en lien avec le pic que l'on voit dans la FIGURE 5 pour ce même $\mathbf{E}$?

3 Conclusions et perspectives

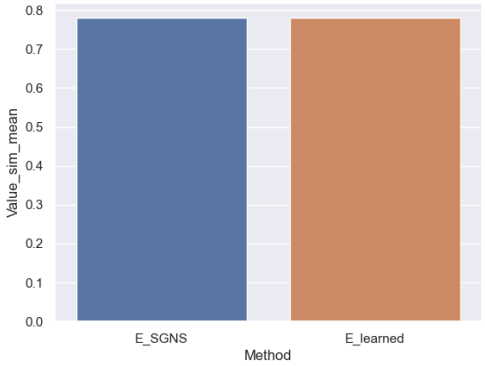
Pour conclure, le modèle de Langage Neuronal reste une approche intéressante pour apprendre un modèle de langage, mais pose plusieurs difficultés que l'on a pu observer ci-dessus. Si, dans une première approche, la matrice de *plogements* $\mathbf{E}$ était initialisée avec la méthode SGNS puis figée à l'entraînement, nous ne produisions pas de résultats assez satisfaisants, bien qu'ayant choisi des hyperparamètres cohérents, et implémenté plusieurs détails pour améliorer l'entraînement, et la génération de texte. L'idée d'augmenter le nombre de paramètres apprenantes en incluant dans l'apprentissage cette matrice $\mathbf{E}$ se trouve confronter a priori à un problème d'évanescence de gradient, ce qui implique que la manière dont est initialisée $\mathbf{E}$, aléatoirement ou à l'aide de la méthode SGNS, n'a pas grande importance (sous réserve que notre code n'est pas faux). Finalement, plusieurs perspectives s'ouvrent à nous pour améliorer ce travail : on pourait introduire une couche résiduelle pour palier à cette evanescence ou rajouter une couche de *Batch Normalization*. Quelques dernières questions peuvent se poser quant à la nature de la sortie cherchée, i.e une distribution multi-modale sur l'ensemble de notre vocabulaire, où certaines classes sont voulues "liées", similaires dans notre problème. Est-ce que notre problème est prompt à un "mode collapse" ? En effet, la génération montrait une tendance à afficher des "mots" qui reviennent souvent dans un texte : la ponctuation, les articles, les marqueurs de début et de fin de phrases et les mots inconnus. Enfin, serait-il pertinent d'ajouter à notre fonction de perte, la volonté de réduire la similarité entre des mots au sens opposés, et augmenter celle de mots très proches ? Par exemple rajouter un terme en $\frac{1}{\mathbf{sim(...)}}$ pour pénaliser lorsque des mots proches ont des similarités trop faibles ?



(a) **E** initialisé aléatoirement et figé

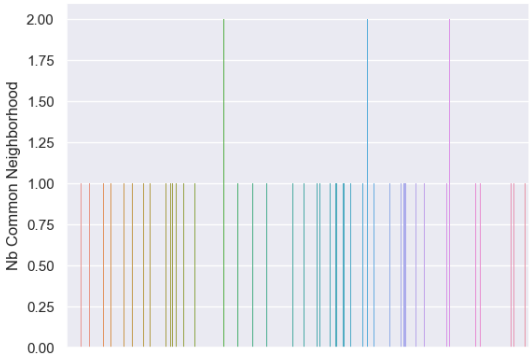


(b) **E** initialisé aléatoirement et appris

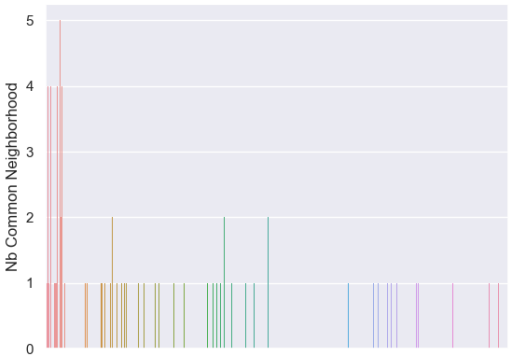


(c) **E** initialisé avec SGNS et appris

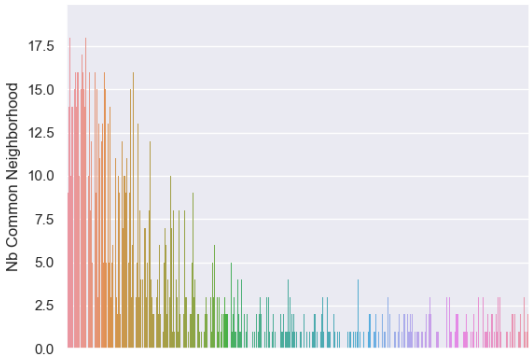
FIGURE 5 – Résultats pour la tâche de similarité pour différents **E**



(a) **E** initialisé aléatoirement et figé

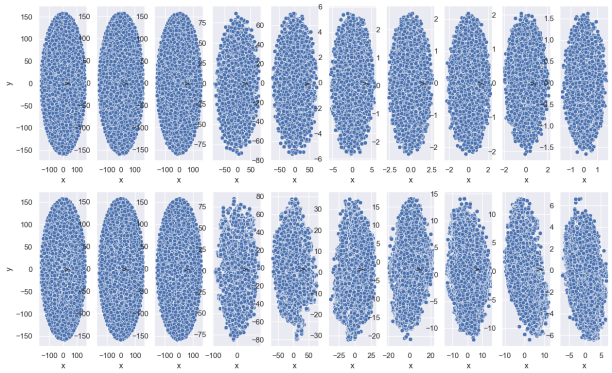


(b) **E** initialisé aléatoirement et appris

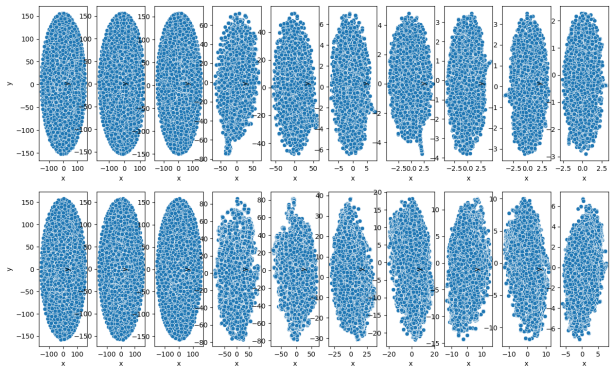


(c) **E** initialisé avec SGNS et appris

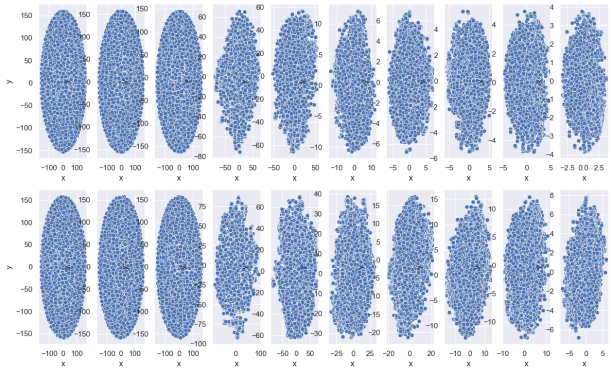
FIGURE 6 – Résultats pour le nombre de voisinage commun pour chaque mot du vocabulaire **E**



(a) **E** initialisé aléatoirement et figé



(b) **E** initialisé aléatoirement et appris



(c) **E** initialisé avec SGNS et appris

FIGURE 7 – Projection TSNE (en première ligne le **E** appris, en deuxième ligne, le **E** obtenu avec SGNS). Les perplexités utilisées sont : 0.001, 0.01, 0.1,5,10,50,100,150,200 et 500