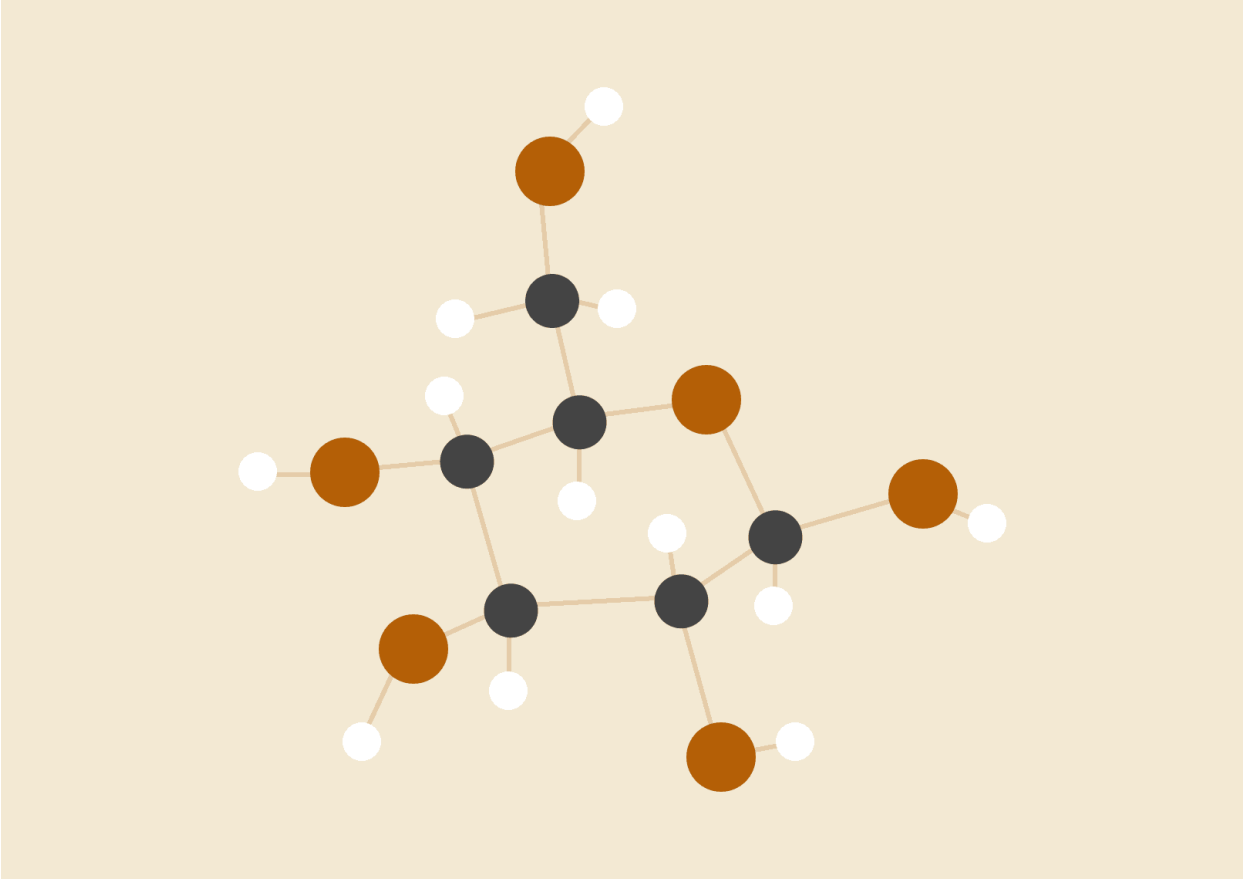
UNIVERSITÉ DE PARIS - LINGUISTIQUE INFORMATIQUE

MODÈLE <<WORD2VEC>> POUR LA CONSTRUCTION DE VECTEURS DE MOTS

réalisé par
Wei HU
Zong-you KE
Vo Tuan Anh AN
encadré par
Mme Marie-Hélène CANDITO



Paris, le 17 mai 2021

MASTER 1 LINGUISTIQUE INFORMATIQUE

PROMOTION 2020-2022

RÉSUMÉ

Le projet a pour objectif d'utiliser le modèle Word2Vec pour la construction de vecteurs de mots. Pour ce faire, nous cherchons à construire le modèle Skip-Gram avec échantillonnage négatif (en anglais, *Skip-Gram with negative sampling*), en appliquant la librairie pytorch dans l'optimisation de l'apprentissage. Le modèle est entraîné sur un corpus annoté de grande taille, libre de droits issu du journal ***L'Est républicain***.

MOTS-CLÉS: *Word2Vec, Skip-Gram, échantillonnage négatif, pytorch*

REMERCIEMENTS

Au terme de ce projet, qui a été pour nous une expérience très enrichissante et qui nous a aussi permis d'apprendre beaucoup tant sur le plan scientifique que sur le plan personnel, nous souhaitons remercier tous ceux qui ont contribué à sa réalisation. Nous tenons à remercier Madame Marie Candito de nous avoir encadrés. Elle a toujours fait preuve d'une grande patience à notre égard. Elle était toujours disponible pour répondre aux questions. Nous tenons également à remercier Madame Bingzhi LI qui nous a fait l'honneur d'accepter d'évaluer ce travail. Nos remerciements vont aussi à l'Université de Paris, en particulier aux professeurs. Nous admirons beaucoup leur compétence et leur passion pour le TAL. Enfin, nous voudrions exprimer nos remerciements aux familles qui nous ont soutenus et nous ont encouragés pendant la réalisation de ce projet.

INTRODUCTION

Le Word2Vec est l'outil permettant de représenter les mots comme des vecteurs à valeur réelle. Deux fameux modèles sont CBOW (en anglais, *Continuous Bag-Of-Words*) et Skip-Gram. Dans ce projet, nous nous intéressons particulièrement au Skip-Gram avec l'échantillonnage négatif.

Grâce au traitement des corpus, le Word2Vec permet de simplifier le traitement du contenu du texte en opérations vectorielles dans l'espace vectoriel K-dimensionnel. La similitude des vecteurs de mots dans l'espace vectoriel peut être utilisée pour exprimer la similarité sémantique du texte. Par conséquent, les vecteurs de mots générés par le Word2Vec sont utilisés pour de nombreuses tâches liées au TAL (en anglais, NLP), telles que le regroupement, la recherche de synonymes et l'analyse d'une partie de la parole.

Un exemple simple, pour déterminer la partie du discours d'un mot, qu'il s'agisse d'un verbe ou d'un nom. En utilisant l'idée de l'apprentissage automatique, nous avons une série d'échantillons (x, y), où x est le mot et y est leur catégorie grammaticale. Si nous voulons construire une application de f (x) -> y, mais ici, le modèle mathématique f (tel que le réseau de neurones, SVM) n'accepte qu'une entrée numérique, les mots en NLP sont cependant des phrases humains, qui sont abstraits pour les machines et sont sous forme symbolique. Il faut donc convertir les énoncés naturels aux formes numériques ou bien intégrer des mots dans un espace mathématique. Cette méthode de plongement est appelée *word embedding*, et le Word2Vec est une sorte d'incorporation de mots.

Le Skip-Gram est le modèle utilisé dans notre travail, dont le principe consiste à utiliser le mot central pour prédire les mots contexte, tandis que le modèle CBOW est à l'inverse. Par rapport au modèle CBOW, il y a plus d'exécutions lors de la prédiction parce que chaque mot du skip-gram est prédit une fois en utilisant les mots contextuels lorsqu'il est utilisé comme un mot central. Cela correspond à K fois de plus que la méthode de CBOW, son temps d'apprentissage est donc plus long que celui de CBOW. C'est une des raisons pour laquelle le temps de notre apprentissage a pris beaucoup de temps en amont de l'apprentissage du modèle. Mais cette caractéristique est plus précise lorsque la quantité est petite, ce qui nous renvoie un résultat relativement précis.

Le modèle Word2Vec est initialement proposé par Google en 2013, il est utilisé pour le prolongement lexical et la reconstruction du contexte linguistique des mots en vecteur. Le Word2Vec distingue des modèles traditionnels comme SOW, BOW, etc. qui permettent de capturer l'information de contextes d'un mot. Son utilisation varie dans différents domaines, comme le système de recommandation, l'analyse des sentiments, la

construction de dictionnaire et la recherche, etc.

Sur la base des travaux réalisés par les auteurs et les chercheurs précédents, un cadre de modèle de langage plus compact a été utilisé pour générer des vecteurs de mots, à savoir le Word2Vec dans l'article intitulé «Distributed Representations of Sentences and Documents». D'ailleurs, avec la méthode proposée dans l'article «Efficient estimation of word representations in vector space», nous avons des théories claires et essentielles à l'apprentissage de deux astuces principales et les plus utilisées de Word2vec: la méthode hiérarchique *softmax* et l'échantillonnage négatif. Ces deux articles sont des travaux de pionnier du Word2Vec présentant l'architecture essentielle du Word2Vec. Pourtant, ils ne sont pas complets à la dérivation théorique, aux détails de la réalisation comme le softmax hiérarchique et l'échantillonnage négatif. Pour avoir plus de détails en formules mathématiques de la dérivation mathématique du modèle, le travail de Yoav Goldberg est recommandé («word2vec Explained-Deriving Mikolov et al.'s Negative-Sampling Word-Embedding Method»).

Dans le projet, nous cherchons à construire le modèle word2vec skip-gram avec échantillonnage négatif en utilisant la librairie pytorch à implémenter la classification du corpus dédiés et la représentation vectorielle des données. Lors du prétraitement du corpus, il faut implémenter à transformer les corpus en représentation vectorielle et matricielle, à faire une classification négative et positive du corpus et à réduire le procédé du calcul par subsampling et l'échantillonnage négatif. En suite par rapport à l'apprentissage, nous testons le modèle skip-gram et calculons la loss, dans cette partie, pour laisser le modèle d'apprentissage marcher plus vite, nous fixons la taille de l'embedding. Par la suite, il faut optimiser le modèle autant que possible et faire afficher des résultats formels. Pendant l'implémentation des codes, nous avons bien noté les dépenses du temps de chaque étape importante, et noté des "training time" de différente taille des données (dans l'annexe).

Nous avons appliqué des corpus de différentes tailles de 50 lignes à 10000 lignes pour entraîner notre modèle et optimiser le résultat. La structure intérieure de chaque corpus qui se constitue par des textes taggés est identique. A l'égard de chaque ligne de type string du corpus, elle est classée par un virgule, se termine par le symbole à la ligne et se consiste en mots en minuscule sauf la 1e lettre de la ligne et ses taggings en majuscule, de façon "mot/TAG" comme "quatre/DET".

Le rapport du projet se déroule comme suit:

- Section 1 pour l'introduction: Dans cette section, nous présentons la tâche, le but du projet et l'état de l'art du modèle word2vec.
- Section 2 pour le cadre théorique: Dans cette section, nous résumons simplement

le Word2Vec et la construction des vecteurs de mots. Nous étudions des concepts fondamentaux du modèle skip-gram avec l'échantillonnage négatif et récapitulons des astuces pour simplifier la tâche.

- Section 3 pour l'implémentation: Cette section est dédiée à implémenter et à concrétiser notre projet. Nous détaillons les étapes de l'algorithme du skip-gram avec l'échantillonnage négatif et l'architecture intégrale de codes à présenter nos idées.
- Section 4 : Nous faisons un résumé du projet.

CADRE THÉORIQUE

- ## Du comptage à la tâche de prédiction de mot

Jusqu'en 2013, les modèles traditionnels pour les tâches étaient des modèles basés sur le comptage. Ils impliquent principalement le calcul d'une matrice de cooccurrence pour capturer les relations sémantiques entre les mots.

La modélisation linguistique basée sur le comptage est facile à comprendre - les mots liés sont observés (comptés) ensemble plus souvent que les mots non liés. De nombreuses tentatives ont été faites pour améliorer les performances du modèles à l'état de l'art, en utilisant SVD, fenêtre en rampe et factorisation matricielle non négative, mais ils n'ont pas bien fonctionné en capturant des relations complexes entre les mots.

- ## Word2Vec Skip-Gram

L'un des modèles de langage basés sur la prédiction introduits par Mikolov (2013) est Skip-Gram:

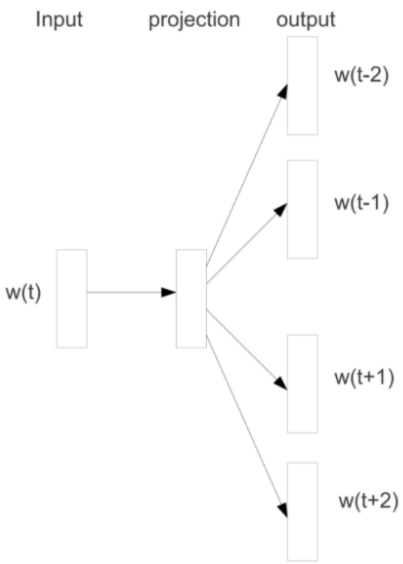


Figure 1: L'architecture du modèle Skip-Gram (Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

La figure 2 est un diagramme présenté dans l'article original du Word2Vec. Il décrit essentiellement que le modèle utilise un réseau de neurones d'une couche cachée (de projection) pour prédire correctement les mots de contexte w(t-2), w(t-1), w(t+1),

w(t+2) d'un mot d'entrée w(t). En d'autres termes, le modèle tente de maximiser la probabilité d'observer les quatres mots de contexte ensemble, étant donné un mot central. Mathématiquement , il peut être noté:

$$\underset{\theta}{\operatorname{argmax}}\ log\, p(w_1, w_2, \ldots, w_C | w_{center};\, \theta)$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

où *C* est la taille de la fenêtre et *w* est un vecteur de mot (qui peut être un contexte ou un mot central). Rappelons qu'en statistique, la probabilité de A étant donné B est exprimée par P(A|B). Ensuite le log naturel est pris sur l'équation (1) pour simplifier la prise de dérivées.

$$\underset{\theta}{\operatorname{argmax}}\ log\, p(w_1, w_2, \ldots, w_C | w_{center};\, \theta)$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

- Pourquoi prédire les mots de contexte?

Une question est pourquoi prédire les mots de contexte? Il faut comprendre que le but du modèle Skip-Gram n'est pas de prédire les mots de contextes, mais d'apprendre une représentation vectorielle intelligente des mots. Il se trouve que la prédiction des mots de contexte aboutit inévitablement à de bonnes représentations vectorielles des mots, en raison de la structure du réseau de neurones de Skip-Gram. Le réseau de neurones consiste, par essence, à optimiser simplement les matrices de poids θpour prédire correctement la sortie. Dans le Word2Vec Skip-Gram, les matrices de poids sont effectivement les représentations vectorielles des mots. Par conséquent, l'optimisation de la matrice de poids est égale à de bonnes représentations vectorielles des mots.

- La dérivation de la fonction de coût

Le modèle Skip-Gram cherche à optimiser la matrice de poids des mots en prédisant correctement les mots de contexte, étant donné un mot central. En d'autres termes, le modèle veut maximiser la probabilité de prédire correctement tous les mots de contexte en même temps, étant donné un mot central. Maximiser la probabilité de prédire les mots de contextes conduit à optimiser la matrice de poids (θ) qui représente le mieux les mots dans un espace vectoriel. θ est une concaténation de matrices de poids

d'entrée et de sortie - [*Winput Woutput*], comme décrit ci-dessous. Il est passé dans la fonction de coût (J) en tant que variable et optimisé.

Dans le Skip-Gram, la fonction softmax est utilisée pour la classification des mots de contenu. Le softmax dans le Skip-Gram a l'équation suivante:

$$p(w_{context}|w_{center};\theta) = \frac{exp(W_{output_{(context)}} \cdot h)}{\sum_{i=1}^{V} exp(W_{output_{(i)}} \cdot h)}$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

$Woutput_{(context)}$ est un vecteur de ligne pour un mot de contexte de la matrice d'embeddings de sortie, et *h* est le vecteur de mot de la couche cachée (projection) pour un mot central. La fonction Softmax est ensuite connectée à l'équation (2) pour produire une nouvelle fonction objective qui maximise la probabilité d'observer tous les mots de contexte *C*, étant donné un mot central:

$$\underset{\theta}{\text{argmax}}\ log \prod_{c=1}^{C} \frac{exp(W_{output_{(c)}} \cdot h)}{\sum_{i=1}^{V} exp(W_{output_{(i)}} \cdot h)}$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

Cependant, en apprentissage automatique, la convention est de minimiser la fonction de coût, pas de la maximiser. Pour respecter la convention, nous ajoutons un négatif à l'équation (4). Cela peut être fait parce que minimiser une log-vraisemblance positive. Par conséquent, la fonction de coût que nous voulons minimiser devient:

$$J(\theta; w^{(t)}) = -log \prod_{c=1}^{C} \frac{exp(W_{output_{(c)}} \cdot h)}{\sum_{i=1}^{V} exp(W_{output_{(i)}} \cdot h)}$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

où *c* est l'indice du mot de contexte autour du mot central (w_t). *t* est l'indice du mot central dans un corpus de taille *T*. En utilisant la propriété de *log*, il peut être changé en :

$$J(\theta; w^{(t)}) = -\sum_{c=1}^{C} log \frac{exp(W_{output_{(c)}} \cdot h)}{\sum_{i=1}^{V} exp(W_{output_{(i)}} \cdot h)}$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

Prendre le log de la fonction *softmax* nous permet de simplifier l'expression en des formes plus simples car nous pouvons diviser la fraction en addition du numérateur et du dénominateur:

$$J(\theta; w^{(t)}) = -\sum_{c=1}^{C}(W_{output_{(c)}} \cdot h) + C \cdot log \sum_{i=1}^{V} exp(W_{output_{(i)}} \cdot h)$$

$$J(\theta; w^{(t)}) = -\sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j} \mid w_t; \theta)$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

Les deux équations ci-dessus sont équivalentes. Elles supposent tous deux une descente de gradient stochastique, ce qui signifie que pour chaque échantillon d'apprentissage (mot central) $w_{(t)}$ dans le corpus de taille *T*, une mise à jour est effectuée sur la matrice de poids (θ). La fonction de coût montre l'équation de *descente de gradient par lots*, ce qui signifie qu'une seule mise à jour est effectuée pour tous les échantillons d'apprentissage.

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-c \leq j \leq c, j \neq 0} \log p(w_{t+j} \mid w_t; \theta)$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

Cependant, dans Word2Vec, la descente de gradient par lots n'est presque jamais utilisée en raison de son coût de calcul élevé. On a déclaré qu'on utilisait la descente de gradient stochastique pour l'entraînement.

- L'architecture du réseau de neurones de Skip-Gram

  Comment le réseau de neurones est-il utilisé pour minimiser la fonction de coût

décrite pour l'équation? Il faut examiner la structure du modèle Skip-Gram pour mieux comprendre leur corrélation.

Supposons que l'ensemble du corpus soit composé de la citation de *La lumière du monde* (2001), "Les bébés sont mes maîtres à penser, ils ne sont jamais tristes.", de Christian Bobin. Il y a 12 mots (T=12) et 11 mots uniques (V=11). Nous utiliserons une fenêtre=1 et supposerons que "maîtres" est le mot central, créant ainsi "mes" et "à" mots de contexte. La fenêtre est un hyper-paramètre qui peut être rectifié empiriquement. Il a une plage de [1,10]. Un réseau neuronal ayant une couche cachée de 3 neurones va être construit. Cela signifie que la matrice de poids d'entrée W_{Input} aura une taille de 11 X 3 et que la matrice de poids de sortie (W_{Output}) aura une taille de 3 X 11. La taille de la couche cachée est aussi un hyper-paramètre qui peut être réglé empiriquement. En pratique, un modèle Word2Vec typique a 200-600 neurones.

- ### Propagation avant

Les matrices d'embeddings de mots (W_{input}, W_{output}) dans le Skip-Gram sont optimisées grâce à des propagations avant (forward propagation) et arrière (backward propagation). Pour chaque itération des propagations avant + arrière, le modèle apprend à réduire l'erreur de prédiction en optimisant la matrice de poids et acquiert ainsi des matrices d'embeddings de meilleure qualité qui capturent mieux les relations entre les mots.

La propagation avant comprend l'obtention de la distribution de probabilité des mots étant donné un mot central, et la propagation arrière comprend le calcul de l'erreur de prédiction et la mise à jour des matrices de poids pour minimiser l'erreur de prédiction.

- ### La couche d'entrée (x)

La couche d'entrée est un vecteur codé V-dim one-hot. Chaque élément du vecteur est 0 sauf une composante qui correspond au mot central (d'entrée). Le vecteur d'entrée est multiplié par la matrice de poids d'entrée (W_{input}) de taille V x N, et produit une couche cachée (h) de vecteur N-dim. Étant donné que la couche d'entrée est codée à chaud, la matrice de poids d'entrée se comporte comme une table de consultation pour le mot central. Supposons un epoch (iter = 1) et une descente de gradient stochastique, le vecteur d'entrée est injecté dans le réseau T fois pour chaque mot du corpus et effectue T mises à jour de la matrice de poids (θ) pour apprendre à partir de l'entraînement des exemples.

- ### La descente de gradient stochastique

L'objectif de tout modèle d'apprentissage automatique est de trouver les valeurs optimales d'une matrice de poids (θ) pour minimiser l'erreur de prédiction. Une équation de mise à jour générale pour la matrice de poids ressemble à ce qui suit:

$$\theta^{(new)} = \theta^{(old)} - \eta \cdot \nabla_{J(\theta)}$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

η est le taux d'apprentissage, $\nabla J(\theta)$ est le gradient pour la matrice de poids et J(θ) est la fonction de coût qui a des formes différentes pour chaque modèle. La fonction de coût pour le modèle Skip-Gram proposé dans l'article originale de Word2Vec a l'équation suivante:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j} \mid w_t;\theta)$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

Ici, ce qui nous donne mal à la tête est l'expression, $\frac{1}{T}\sum_{t=1}^{T}$, car T peut être supérieur à des milliards ou plus dans de nombreuses applications du TAL. Cela nous dit essentiellement que des milliards d'itérations doivent être calculées pour effectuer une seule mise à jour de la matrice de poids (θ). Afin d'atténuer cette charge de calcul, l'auteur de l'article déclare que la descente de gradient stochastique (SGD) a été utilisée pour l'optimisation des paramètres. Le SGD supprime l'expression, $\frac{1}{T}\sum_{t=1}^{T}$, de la fonction de coût et effectue la mise à jour des paramètres pour chaque exemples d'apprentissage, $w^{(t)}$:

$$J(\theta; w^{(t)}) = -\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j} \mid w_t;\theta)$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

Ensuite, la nouvelle équation de mise à jour des paramètres pour le SGD devient:

$$\theta^{(new)} = \theta^{(old)} - \eta \cdot \nabla_{J(\theta;w^{(t)})}$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

- **Matrices de poids d'entrée et de sortie (Winput, Woutput)**

L'objectif du modèle Skip-Gram est de construire les matrices des embeddings de mots (W_{input}, W_{output}) pour calculer les relations entre mots dans l'espace vectorielle. Pour cela, le modèle utilise un réseau de neurones. Il permet d'optimiser les matrices de poids pour minimiser l'erreur de prédiction (Y_{pred} - Y_{gold}).

- **La couche cachée (h)**

Le Skip-Gram utilise le réseau de neurones ayant une seule couche cachée. Il s'agit d'un vecteur de N-dim projeté par le vecteur d'entrée encodé one-hot. h est obtenu en multipliant la matrice d'embeddings d'entrée par le vecteur du mot d'entrée:

$$h = W_{input}^{T} \cdot x \in \mathbb{R}^{N}$$

(Source: Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io))

- **La couche de sortie softmax**

A la sortie, un neurone permet de calculer la probabilité d'appartenance d'un mot du vocabulaire. Une matrice $E^{(c)}$ est utilisée pour la combinaison linéaire entre la couche cachée et la sortie.

- **Échantillonnage négatif**

Pour un mot cible, au lieu de donner une distribution de probabilité pour tous les mots du vocab, on applique une classification binaire probabiliste. On prend en entrée un couple de mots et donne en sortie la probabilité qu'un mot soit un mot de contexte de l'autre mot. Pour apprendre, on a aussi besoin d'exemples négatifs pour chaque exemple positif.

En particulier, pour chaque mot cible on prend le contexte de taille C ainsi qu'un ensemble d'exemples négatifs de taille $k * C$(C et k sont des hyperparamètres). Le but est de maximiser la probabilité du contexte avec le mot cible, et de minimiser celle du mot cible avec les exemples négatifs.

Si on note n la représentation d'un exemple négatif, minimiser $f(w \cdot n)$ revient à maximiser $1 - f(w \cdot n)$.

Si on note, pour un mot cible w, M l'ensemble des mots du contexte de w, et N l'ensemble de ses exemples négatifs, alors on cherche pour ce mot:

$$argmax(M,N)\,(\prod_{c_i \in M}^{C} f(w \cdot c_i) + \prod_{n_j \in N}^{k \times C} 1 - f(w \cdot n_j))$$

On a donc besoin pour réaliser cette tâche:

- d'une représentation vectorielle pour le mot cible,
- d'une représentation vectorielle pour chaque mot du contexte,
- et d'une représentation vectorielle pour chaque mot des exemples négatifs, ceci afin de pouvoir calculer les probabilités qui viennent d'être décrites.

IMPLEMENTATION

1. Le corpus utilisé

Nous utilisons un corpus libre de droits issu du journal *L'Est républicain*. Chaque archive contient environ 40 millions de mots (26 fichiers de 100 000 phrases/lignes chacun, environ 1,6 million de mots chacun). Les fichiers sont au format une phrase par ligne, segmentée en tokens, et les mots sont taggés (on n'a pas besoin des tags). Certains mots composés ont été repérés (comme "parce que").

Pour entraîner notre modèle de manière graduelle, nous avons choisi un fichier contenant 100 000 lignes pour le découper en plusieurs sous-corpus, ces derniers ayant 50, 100, 200, 1 000, 3 000, 5 000, 10 000, 30 000, 50 000, 100 000 lignes, respectivement.

2. Le choix de l'algorithme

On implémente le modèle Skip-Gram avec l'échantillonnage négatif sur un réseau neuronal non-linéaire qui transforme notre tâche en classification binaire.

3. La description de l'algorithme

L'apprentissage se fait sur un corpus, d'où l'on construit des paires d'un mot cible et un mot contexte positif ou négatif. Chaque paire (*w, c*) est suivie par une classe binaire (0 = positif et 1 = négatif). On itère sur chaque mot du vocabulaire et passe toutes les paires autour de chaque mot dans le réseau neuronal pour exécuter la propagation avant et la rétropropagation. Ensuite on mesure les pertes d'entraînement et de validation pour déterminer une époque optimale à faire des expériences chronologiques en fonction de notre objectif. Finalement, on modifie les hyperparamètres et on observe le temps utilisé pour des différentes tailles de corpus avec de différentes combinaisons d'hyperparamètres.

Ci-dessous est notre algorithme schématisé :

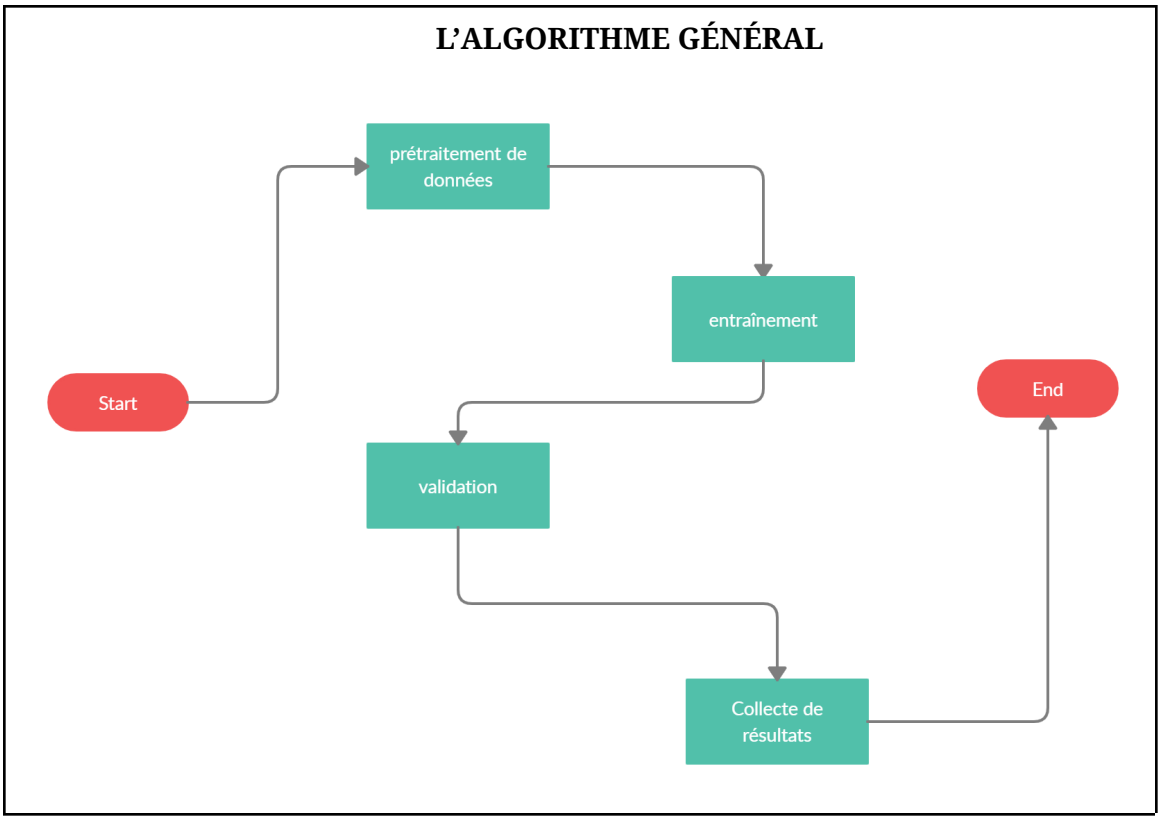


Figure 2: L'algorithme général

a. Prétraitement de données:

Notre algorithme lit chaque sous-corpus ligne par ligne et tokenise chaque phrase. Les tags et les chiffres sont exclus de l'ensemble des tokens.

Ensuite, on lit chaque mot dans chaque phrase du sous-corpus. Pour chaque mot cible avec un contexte positif (c_{pos}), on construit K paires contenant chacun le même mot cible et un mot contexte négatif (c_{neg}). Avec la taille de demi-fenêtre C, on considère les C mots avant et après le mot cible comme ses mots contextes positifs qui vont être transformés en paires (w, c). Le nombre des paires construites à partir d'un mot ne dépasse donc pas $2C$ (positif) + $2C*K$ (négatif).

Quant à l'échantillonnage négatif, pour chaque exemple positif, on tire au hasard K mots comme exemples négatifs, qui ne sont pas dans l'ensemble des c_{pos} d'un mot cible, depuis le vocabulaire. On échantillonne d'abord le vocabulaire, ou l'étape du sous-échantillonnage. Pour ce faire, on lisse la distribution probabiliste de chaque mot à la puissance de 0,75 (selon les auteurs de Word2Vec) et applique l'équation proposée par les auteurs de

Word2Vec à chaque mot lissé :

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)|}}$$

où $P(w_i)$ signifie la probabilité calculée d'un mot w_i, $f(w_i)$ la fréquence relative de w_i et t un constante de seuil. Les $P(w_i)$ sont comparées avec un chiffre de 0 à 1 déterminé à chaque opération d'apprentissage, et les mots ayant une probabilité supérieure à ce chiffre sont gardés dans la population des mots contextes négatifs. De cette population, on tire au hasard K mots négatifs pour chaque paire positive.

Durant le prétraitement, les tokens en string sont transformés en integer et mis dans les tuples (w, c_{pos}/c_{neg}, *gold label*), où *gold label* égal à 1 dans une paire positive et à 0 dans une paire négative. Finalement, on sauvegarde les tuples chiffrées en fichiers .txt pour chaque sous-corpus pour ne pas devoir répéter le prétraitement à chaque opération d'apprentissage.

Pour s'adapter au design du réseau neuronal, les tuples chiffrées sont transformées en trois vecteurs de mots cibles, mots contextes et gold labels. Les exemples positifs et négatifs sont aussi divisés en un jeu d'apprentissage et un jeu de validation, avec un ratio de division à 4:1, afin de vérifier le résultat de l'entraînement.

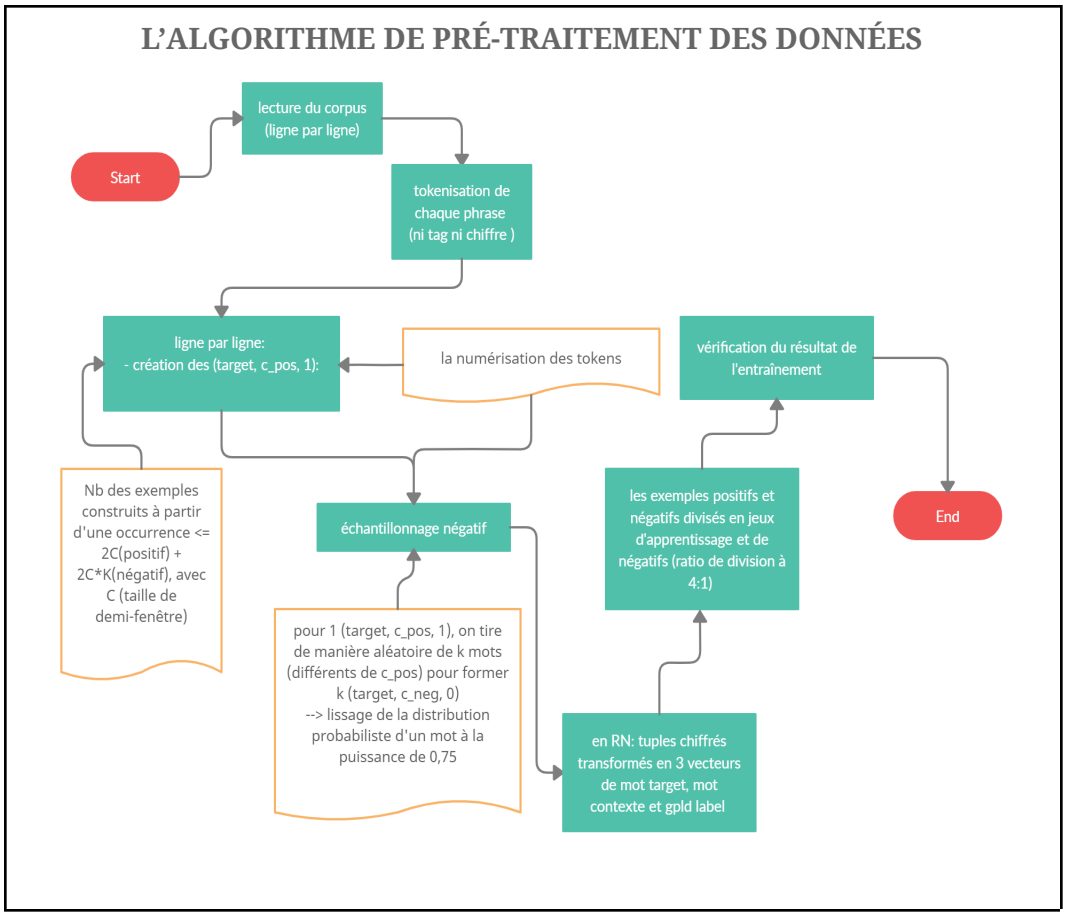


Figure 3: L'algorithme de pré-traitement des données

b. Entraînement

L'entraînement se fait en PyTorch à travers d'un réseau neuronal pour exécuter la propagation avant et la rétropropagation d'un certain nombre d'époques. Le résultat de l'entraînement est validé par la phase de validation pour déterminer une époque optimale en évitant le sur-apprentissage. Plusieurs combinaisons d'hyperparamètres sont testées pour arriver aux temps optimaux pour entraîner et valider un sous-corpus.

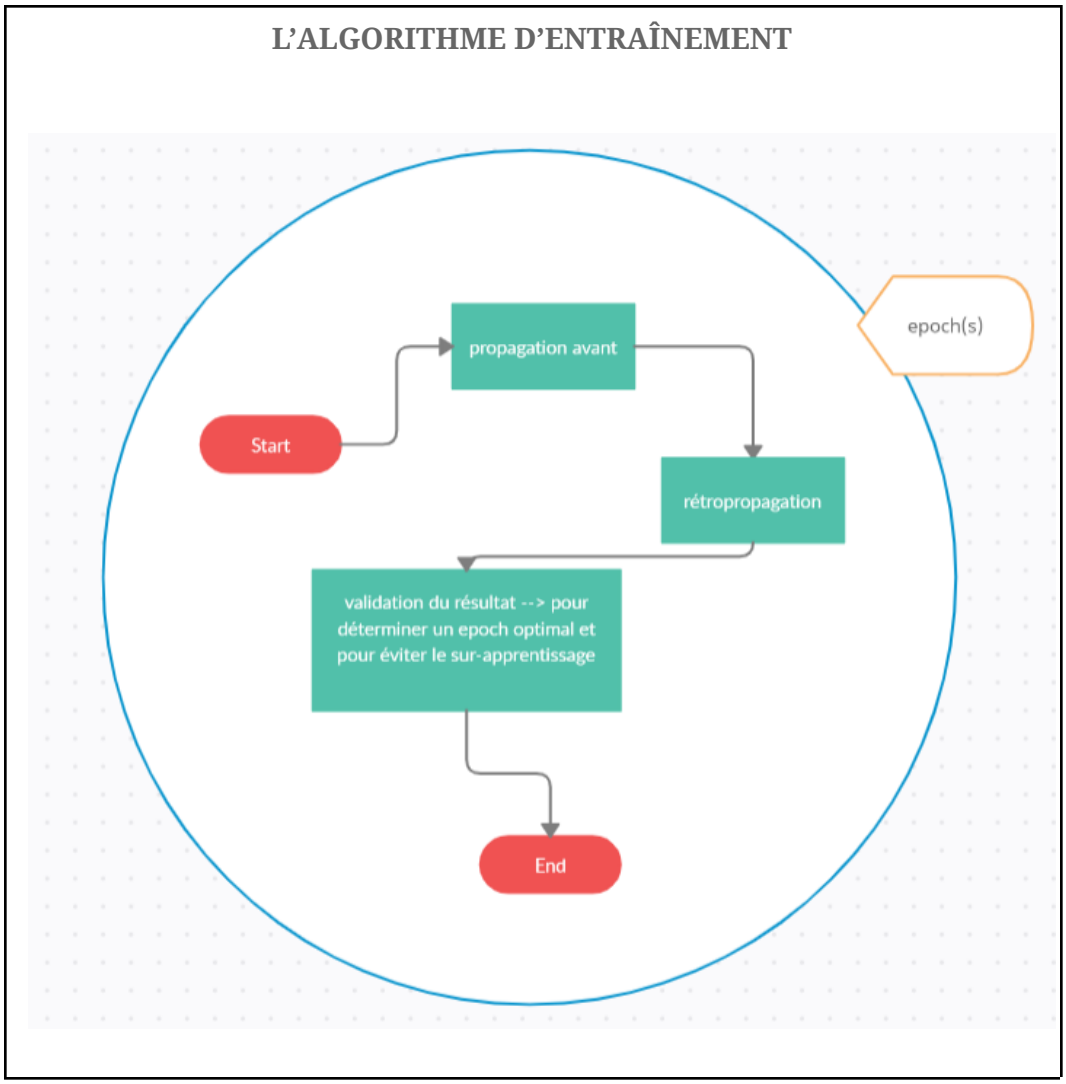


Figure 4: L'algorithme d'entraînement

4. L'architecture du réseau neuronal

L'architecture du réseau neuronal comprend une couche d'entrée, une couche d'embeddings, deux couches cachées et une couche de sortie. La couche d'entrée contient deux vecteurs one-hot d'un mot cible et d'un mot contexte qui vont être multipliés par une matrice d'embeddings de mots cibles (E^w) et par une matrice d'embeddings de mots contextes (E^c) dans la couche d'embeddings pour obtenir deux vecteurs d'embeddings. Ces deux vecteurs d'embeddings sont V^w et V^c, respectivement.

Quant aux couches cachées, chacune d'elles contient soit V^w, soit V^c. La couche de

sortie contient le résultat du produit scalaire de V^w et V^c, qui est le score à passer dans la fonction d'activation pour déclencher la rétropropagation.

Figure 5 illustre l'architecture du réseau neuronal utilisé dans ce projet:

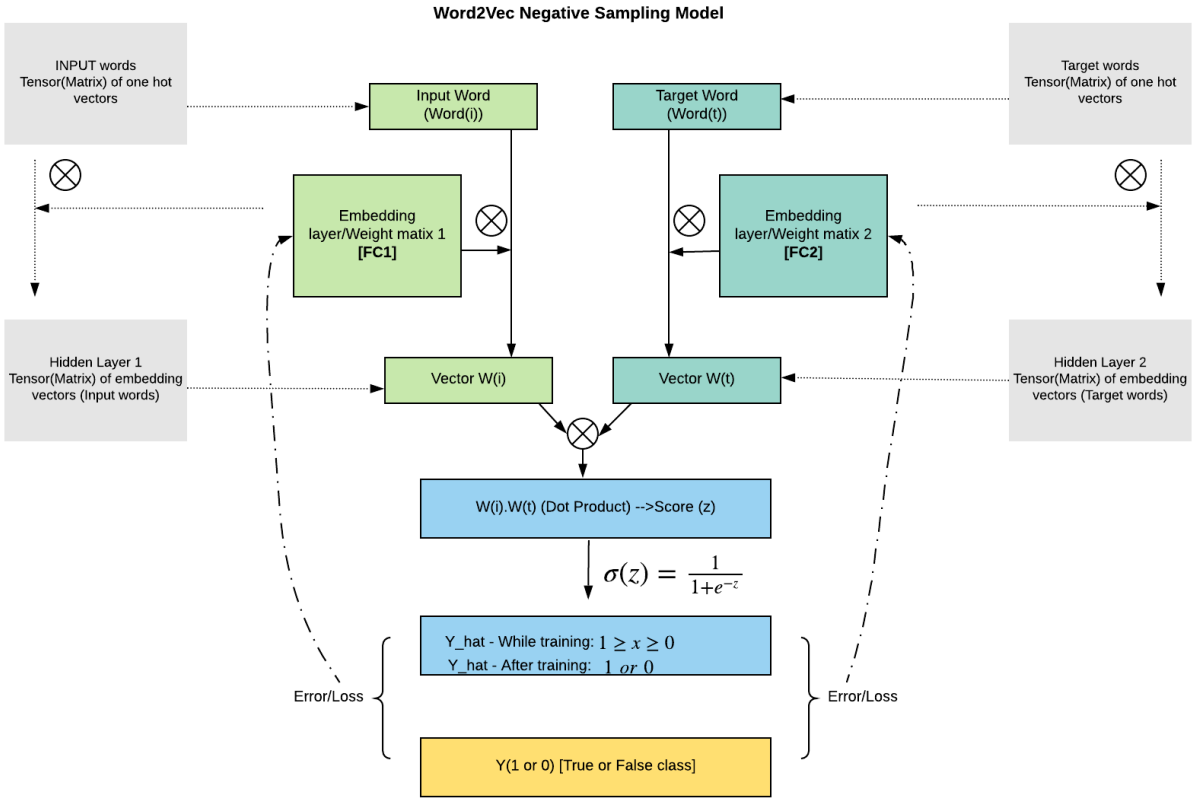


Figure 5 : l'architecture du modèle Skip-Gram avec l'échantillonnage négatif. (Source: Word2Vec -Negative Sampling made easy)

Les détails de chaque couche sont décrits comme suit :

a. Couche d'entrée

La couche d'entrée reçoit un vecteur d'un mot cible et un autre vecteur d'un mot contexte positif ou négatif. Ces deux vecteurs one-hot issus du prétraitement du corpus utilisé ont la taille V (la taille du vocabulaire). Quand les paires *(w, c)* sont passées en batch, il y aura 2**batch_size* vecteurs d'input de taille V.

b. Couche d'embeddings et couches cachées

Les deux couches cachées contiennent deux vecteurs d'embeddings V^w et V^c de taille M (*embedding dimension*), qui sont les produits de la multiplication entre les vecteurs d'input et deux matrices d'embeddings E^w et E^c de dimension V x M dans la couche d'embeddings. Quand les vecteurs d'input sont passés en batch, la quantité de V^w et V^c sera le *batch_size*.

Aucune fonction d'activation n'est appliquée dans la couche cachée.

c. Couche de sortie

La couche de sortie contient le score, qui est le résultat du produit scalaire de V^w avec V^c. Ici, on applique la fonction sigmoïde pour normaliser le score puisqu'il s'agit d'une classification binaire au lieu d'une classification multiclasse.

d. Fonction de perte

La fonction de perte est BCEWithLogitsLoss en PyTorch qui contient une fonction sigmoïde. On calcule les gradients de la fonction objectif par rapport aux matrices d'embeddings E^w et E^c. Afin d'obtenir des gradients, on calcule d'abord la perte du modèle pour chaque paire d'exemples. Dans le cas où l'on adopte la méthode stochastique, la perte avec une paire d'exemples est passée directement au calcul des gradients. Dans le cas de mini-batch, on calcule une perte pour toutes les paires dans un mini-batch et puis on passe le résultat au calcul des gradients. La taille de mini-batch est un hyperparamètre à déterminer empiriquement.

5. L'architecture du code

Nous avons produit deux versions de code : l'ancienne qui est moins structurée et moins efficace, et la récente qui est la version structurée et essentielle pour faire des expériences. Nous détaillons donc la nouvelle version ci-dessous.

Notre code est structuré en trois classes (Data, SGNS et Word2Vec) et une section *main*. Les utilités principales de ces parties sont comme suit :

- Data : prétraitement des données.
- SGNS : modèle du Skip Gram avec échantillonnage négatif
- Word2Vec : l'algorithme de l'apprentissage

- *main* : déclencheur de l'algorithme et affichage/enregistrement de résultats

Figure 6 ci-dessous illustre les composants de chaque classe et les interactions entre elles.

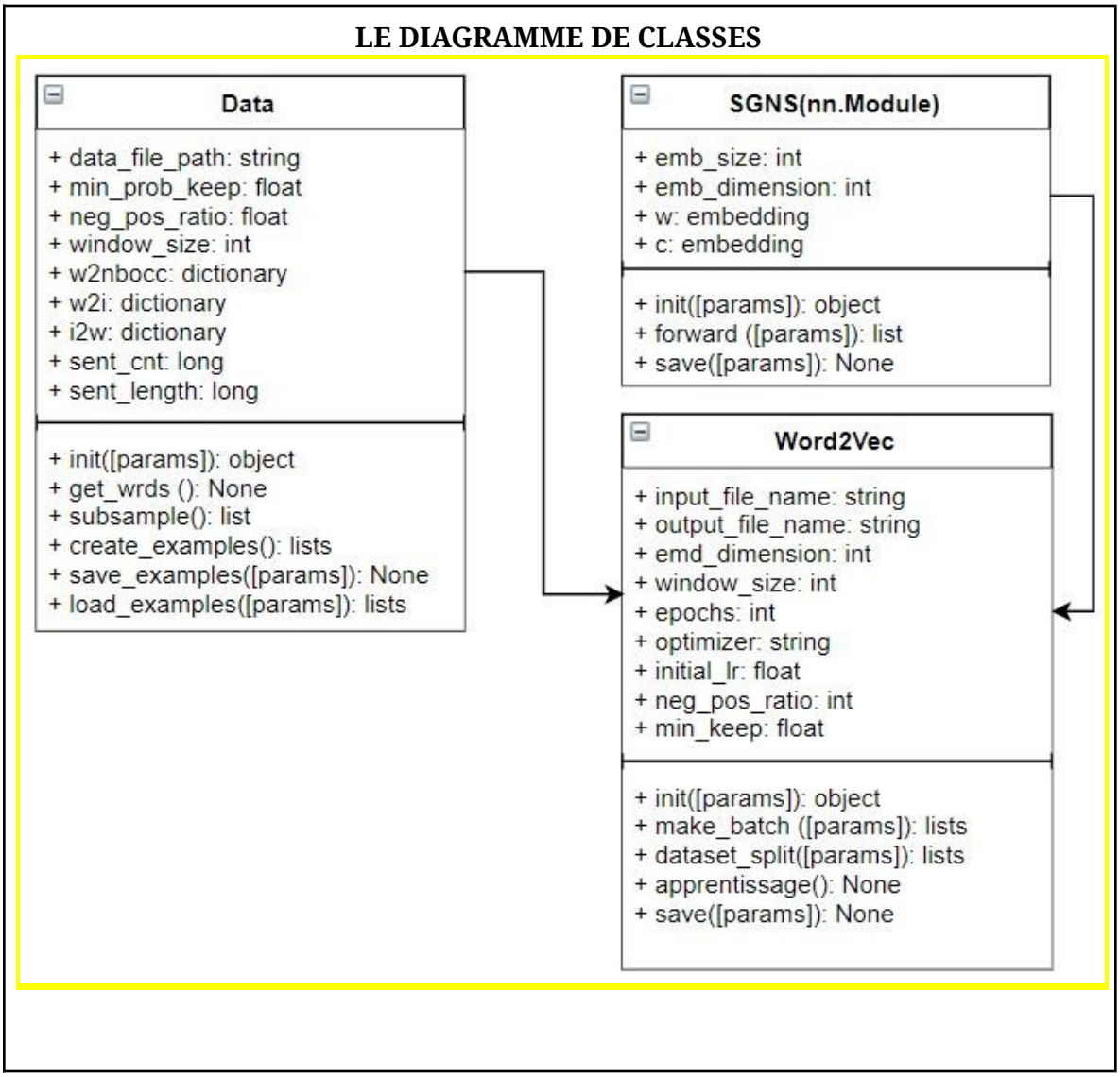


Figure 6: Le diagramme de classes

(1) Data :

Cette classe s'utilise à l'aide de trois fonctions principales : *get_wrds* (lire et tokeniser un corpus), *subsample* (sous-échantillonner le vocabulaire) *et create_examples* (créer des exemples positifs et négatifs). Deux fonctions auxiliaires *save_examples* et *load_examples* sont aussi incluses pour

sauvegarder les exemples nouvellement créés et lire ceux qui ont déjà été créés.

(2) SGNS :

Cette classe définit les paramètres du réseau neuronal, à savoir la couche d'entrée, la couche d'embeddings, les couches cachées, la couche de sortie, ainsi que les fonctions forward. Il existe également une fonction auxiliaire *save* pour sauvegarder la configuration du réseau.

(3) Word2Vec :

Cette classe définit la procédure de l'apprentissage, de la vectorisation au calcul de perte totale d'une époque, en passant par la propagation avant et la rétropropagation. Suite à l'apprentissage, on implémente une phase de validation pour évaluer le modèle (c'est-à-dire les matrices d'embeddings E^w et E^c) entraîné par la phase précédente pour éviter le sur-apprentissage. Cela nous permet de déterminer les époques optimales pendant les expériences sur les hyperparamètres.

On ajoute deux fonctions auxiliaires dans cette classe, à savoir *make_batch* et *dataset_split*. Respectivement, elles servent à regrouper les exemples en mini-batch et à découper les exemples en un jeu d'entraînement et un jeu de validation.

(4) Main

La section *main* permet de déclencher l'algorithme en introduisant un sous-corpus de choix. Les hyperparamètres (la dimension d'embeddings, la taille de batch, le taux d'apprentissage, etc.) peuvent être modifiés ici et puis testés en plusieurs combinaisons dans une seule opération d'apprentissage grâce au module *itertool*. Les temps utilisés pour le prétraitement et l'apprentissage (le temps au total ainsi que celui de chaque époque) sont stockés dans un DataFrame et aussi dans un fichier .csv. En outre, les graphes sur les pertes d'entraînement et de validation sont représentés ici pour observer les époques optimales.

RÉSULTATS ET ANALYSES

1. Corpus

Le tableau ci-dessous détaille les nombres de tokens des sous-corpus que nous avons utilisés pour ce projet. Pour nos expériences sur les hyperparamètres, les nombres de tokens déterminent directement la taille d'un sous-corpus à traiter, au lieu du nombre des lignes.

Taille du corpus (lignes)	Taille du corpus (tokens)
50	476
100	959
800	10 568
1 000	12 838
3 000	37 421
5 000	60 076
10 000	123 019
30 000	345 600
50 000	608 135
100 000	129 2493

Table 1: Nombres de lignes et de tokens pour chaque sous-corpus

2. Différences d'efficacité entre versions

L'ancienne version du code sans représentation vectorielle et matricielle prenait beaucoup plus de temps pour l'apprentissage que la nouvelle version. Par exemple, un sous-corpus ayant 10 000 lignes devait prendre 10 heures pour finir 5 époques d'entraînement. Inversement, la nouvelle version permet de traiter la même taille de sous-corpus en moins de 2 minutes. Nous avons opté pour la nouvelle version sans hésitation.

3. Tendances générales

Nous avons commencé nos expériences par observer les tendances générales entre chaque hyperparamètre et le temps d'apprentissage.

D'abord sur les petits sous-corpus de 50 lignes jusqu'à 1 000 lignes. Prenons

l'exemple du sous-corpus ayant 800 lignes. Dans Tables 2 et 3 ci-dessous, on peut constater que le temps d'apprentissage diminue avec l'augmentation de la taille d'un batch.

Input_file	Emb_dim	Batch_size	Min_keep	Epochs	Learning_rate	T_Preproccessing	T_Training (s) per epoch
train_800.txt	25	16	0.5	20	0.001	15.538	6.548
train_800.txt	25	32	0.5	20	0.001	13.892	2.708
train_800.txt	25	64	0.5	20	0.001	15.200	1.817
train_800.txt	50	16	0.5	20	0.001	17.745	8.726
train_800.txt	50	32	0.5	20	0.001	15.397	3.846
train_800.txt	50	64	0.5	20	0.001	15.904	2.752
train_800.txt	100	16	0.5	20	0.001	15.512	15.365
train_800.txt	100	32	0.5	20	0.001	16.947	8.240
train_800.txt	100	64	0.5	20	0.001	14.674	3.408

Input_file	Emb_dim	Batch_size	Min_keep	Epochs	Learning_rate	T_Preproccessing	T_Training (s) per epoch
train_3000.txt	25	16	0.5	20	0.001	26.894	31.450
train_3000.txt	25	32	0.5	20	0.001	26.237	16.133
train_3000.txt	25	64	0.5	20	0.001	25.621	9.098
train_3000.txt	50	16	0.5	20	0.001	25.877	57.199
train_3000.txt	50	32	0.5	20	0.001	25.478	28.443
train_3000.txt	50	64	0.5	20	0.001	26.212	15.287
train_3000.txt	100	16	0.5	20	0.001	27.148	122.811
train_3000.txt	100	32	0.5	20	0.001	40.167	61.590
train_3000.txt	100	64	0.5	20	0.001	40.538	29.292

Tables 2 & 3: Résultats obtenus avec *train_800* et *train_3000*, 1er ensemble d'hyperparamètres

On peut constater qu'entre *train_800* et *train_3000*, le nombre de tokens augmente de 10 568 à 37 421, soit 3.54 fois de croissance. Pour le but de comparer les deux sous-corpus, si on fixe tous les hyperparamètres, le temps d'apprentissage requis per époque augmente de 1.817 secondes à 9.098 secondes, soit 5 fois de croissance, quand *emb_dim = 25* et *batch_size = 64*. Quand *emb_dim = 50* et *batch_size = 64*, la croissance du temps d'apprentissage par époque est à 5.56 fois. Quand *emb_dim = 100* et *batch_size = 64*, la croissance du temps d'apprentissage par époque devient 8.61 fois. Cela montre que la relation entre la croissance du temps d'apprentissage par époque et la dimension d'embeddings n'est pas linéaire.

Input_file	Emb_dim	Batch_size	T_Training (s) per epoch
train_3000.txt	200	1024	6.063
train_3000.txt	300	1024	8.249
train_3000.txt	200	512	9.400
train_3000.txt	300	512	13.360
train_3000.txt	200	256	14.876
train_3000.txt	300	256	20.594
train_3000.txt	200	128	28.172
train_3000.txt	300	128	41.685
train_3000.txt	200	64	55.750
train_3000.txt	300	64	87.379

Table 4: Résultats obtenus avec *train_3000*, 2ème ensemble d'hyperparamètres

En se servant du sous-corpus *train_3000*, on continue à tester les valeurs supérieures de la dimension d'embeddings et de la taille d'un batch pour observer les résultats. Il se trouve que quand ***emb_dim*** = ***200*** et ***batch_size*** = ***1024***, on obtient le meilleur résultat, soit 6.063 secondes par époque.

4. Tendances spécifiques

Après avoir déterminé la meilleure combinaison de la dimension d'embeddings et de la taille d'un batch (*emb_dim = 200* et *batch_size = 1024*), nous avons continué à tester d'autres valeurs des autres hyperparamètres pour trouver des combinaisons optimales. À cette étape, on restait encore sur *train_3000* pour faire des expériences sur différents taux d'apprentissage (0.1, 0.001 et 0.00001) et différentes valeurs du seuil probabiliste du filtrage (0.5, 0.6 et 0.7). Les résultats obtenus sont comme suit :

Input_file	Emb_dim	Batch_size	Min_keep	Learning_rate	T_Training (s) per epoch
train_3000.txt	200	1024	0.7	0.001	2.813
train_3000.txt	200	1024	0.7	1.00E-05	2.822
train_3000.txt	200	1024	0.7	0.1	2.871
train_3000.txt	200	1024	0.6	1.00E-05	2.952
train_3000.txt	200	1024	0.6	0.001	2.954
train_3000.txt	200	1024	0.6	0.1	2.954
train_3000.txt	200	1024	0.5	0.001	2.958
train_3000.txt	200	1024	0.5	0.1	3.013
train_3000.txt	200	1024	0.5	1.00E-05	3.091

Table 5: Résultats obtenus avec *train_3000*, 3ème ensemble d'hyperparamètres

On peut constater que quand *min_keep = 0.7*. le temps d'apprentissage per époque est le plus petit. Cela est cohérent avec d'autres tailles des sous-corpus. Par contre, on ne peut pas constater une influence évidente du taux d'apprentissage sur le temps d'apprentissage.

Étant donné les analyses ci-dessus, nous avons choisi *emb_dim = 200*, *batch_size = 1024*, *min_keep = 0.7* et *learning_rate = 0.001*, tout en gardant le *nég_pos_ratio = 4*. Ces hyperparamètres seront utilisés pour calculer le temps qu'il faut pour apprendre sur les grands corpus.

5. Tests sur les grands corpus

Nous avons appliqué les hyperparamètres ci-dessus sur les grands sous-corpus ayant 10 000, 30 000, 50 000 et 100 000 lignes. Voici leur résultat pour l'apprentissage :

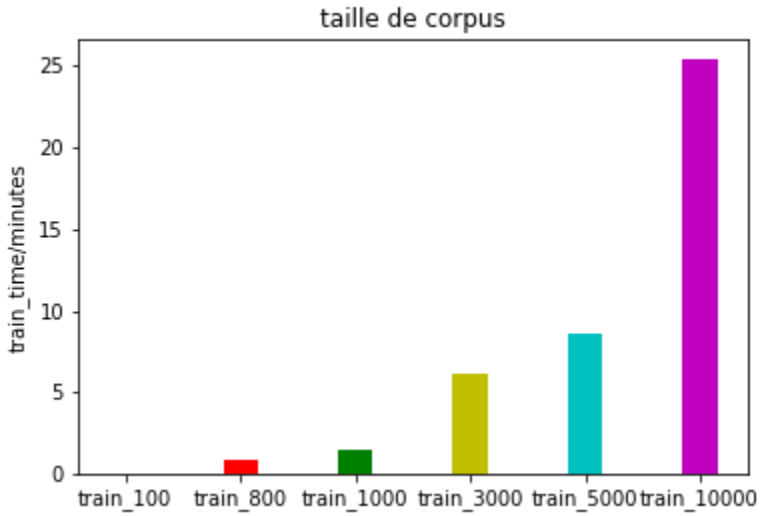


Figure 7: Application des hyperparamètres optimaux sur les sous-corpus différents

On peut constater que le temps requis pour finir une époque est meilleur que l'ancien code qui demande des heures pour finir 5 époques. Faute de temps, nous n'avons pas pu faire tourner beaucoup d'époques sur ces sous-corpus, mais il est déjà évident que l'efficacité s'est améliorée de manière importante.

DISCUSSION

Dans cette section, nous allons discuter de quelques aspects cruciaux liés à l'implémentation et l'exécution de ce projet.

a. Le design du pipeline

Le pipeline joue un rôle essentiel dans ce projet pour permettre d'obtenir plus de résultats en moins de temps. Toutefois, nous avons continué à l'optimiser du début jusqu'à la fin avec des découvertes de nouveaux obstacles. L'obstacle principal, c'est que le prétraitement des sous-corpus de grande taille (>5000 lignes) prend en général plus de 30 secondes à finir, et nous ne pouvons pas entrer directement dans l'étape d'apprentissage sans avoir prétraité un sous-corpus. Vers la fin du projet, nous avons créé des fonctions auxiliaires qui permettent d'enregistrer et de charger les exemples positifs et négatifs en tuples chiffrées. Cette méthode a permis d'économiser le temps pour produire les exemples de manière importante. Pour un sous-corpus ayant 100 000 lignes, par exemple, le temps du prétraitement s'est réduit de 55 minutes à moins de deux minutes.

Par contre, un des inconvénients est que les fichiers produits peuvent être gigantesques jusqu'à des centaines d'MB. Cela n'est pas pratique pour les importer sur Google Colab vu la forte chance que le téléchargement de grands textes soit incomplet. Il est devenu plus convenable d'avoir des copies locales sur le PC et y faire tourner l'apprentissage malgré l'indisponibilité de GPU.

b. Le choix des bonnes opérations vectorielles en Pytorch

Nous avons implémenté les représentations vectorielle et matricielle dans le prétraitement et l'entraînement du code, ce qui a contribué à accélérer l'apprentissage.

Concernant le prétraitement, nous avons choisi de construire un pool de candidats de mots contextes négatifs en se servant du type primitif *set* qui traite tout le vocabulaire en une seule opération. Après, nous créons une liste d'indices aléatoirement en fonction du ratio négatif contre positif en une seule opération vectorielle pour éviter d'accéder au pool à chaque création d'un exemple négatif.

Pour l'entraînement, nous avons choisi de représenter chaque entrée d'exemple par le biais de *np.array* et *torch.tensor*, qui sont tous deux formes vectorielles. Chaque exemple est restructuré en trois vecteurs nommés *targets*, *contexts* et *labels*, puis regroupé en mini-batch pour déclencher la propagation avant.

Dans l'ancienne version, la représentation vectorielle de *targets* et *contexts* a permis une réduction importante du temps d'apprentissage, de plus de 24 heures avec les listes (type primitif en Python) à 6,4 heures pour un sous-corpus ayant 5000 lignes. En outre, l'abandon de la représentation one-hot vecteur pour chaque mot cible et mot contexte nous a évité de dépasser la limite de la RAM sur Google Colab.

c. Le choix des hyperparamètres

Le temps requis par l'entraînement dépend de la taille des jeux de données (jeu d'entraînement et de validation), de combien de données l'algorithme peut traiter à chaque unité de temps, et du nombre d'époques au total.

En ce qui concerne la taille des jeux de données, les hyperparamètres pertinentes sont comme suit : **la taille du corpus, le ratio d'exemples négatifs contre les exemples positifs, la taille de fenêtre,** et **la dimension d'embeddings.** Le temps d'apprentissage requis est généralement basé sur la taille de données traitées. En effet, plus grande est la taille de ces dernières, plus de temps l'apprentissage a besoin pour terminer. En chronométrant les étapes au sein de l'entraînement, on peut constater que l'étape d'optimisation *optimizer.step* et l'étape de la rétropropagation *loss.backward* consommaient le plus de temps, car elles traitent chaque valeur stockée dans les représentation vectorielle ou matricielle. Si l'on souhaite réduire le temps d'apprentissage sur les grands corpus, on peut d'abord baisser le ratio d'exemples négatifs contre les exemples positifs, ou bien la taille de fenêtre pour qu'il y ait moins de données à traiter. Mais si les deux hyperparamètres sont fixés à des valeurs optimales, on peut modifier la dimension d'embeddings puisque plus grande elle est, plus de valeurs stockées il y a dans les représentations vectorielles et matricielles.

Concernant la quantité de données traitée par l'algorithme à chaque unité de temps, il y a un hyperparamètre pertinent : **la taille d'un batch**. En effet, on peut constater que le temps d'apprentissage se réduit avec l'augmentation de la taille d'un batch. Ce phénomène peut s'expliquer par la capacité de l'algorithme à exécuter une opération vectorielle sur multiples données en batch en une durée similaire. Autrement dit, des données de taille 10 et de taille 100 000 peuvent produire peu de différence sur le temps requis dans une opération vectorielle. Par conséquent, plus un batch est grand, plus de taille de données l'algorithme peut traiter à chaque unité de temps. Si l'on veut réduire davantage de temps

d'apprentissage, l'augmentation de la taille d'un batch est primordiale.

Quant au **nombre d'époques**, c'est un facteur qui détermine le temps d'apprentissage de manière importante. Dans notre code, le nombre d'époques est *la somme d'époques pour l'entraînement et la validation* fois *le nombre des combinaisons des hyperparamètres*.

Pour les nombres d'époques optimaux, nous avons trouvé que c'est avec de 15 à 30 époques que nous pouvons observer de meilleurs résultats. La méthode de détermination consiste à observer la courbe des pertes du jeu d'entraînement et celle du jeu de validation sur un même graphe. Le moment où la courbe du jeu d'entraînement stagne alors que celle du jeu de validation augmente représente le nombre d'époques théoriquement idéal. Toutefois, ces nombres d'époques sont assez grands pour ralentir l'apprentissage. Quand ils appliquent aux grands sous-corpus, il est convenable d'appliquer d'autres hyperparamètres qui servent à accélérer l'apprentissage, comme, par exemple, une grande taille de batch. Sinon, l'apprentissage sur les grands corpus pourra durer des heures.

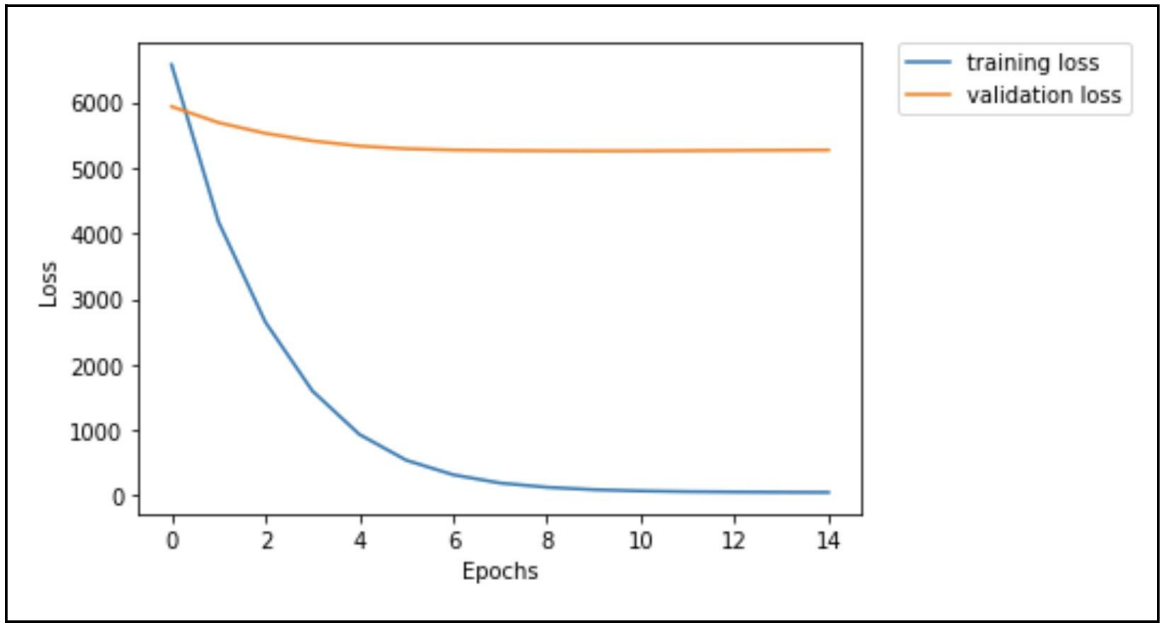


Figure 8: Courbes exemplaires des pertes pour l'entraînement et la validation

À noter que les nombres d'époques optimaux dépendent aussi du **taux d'apprentissage**. Plus le taux d'apprentissage est grand, plus vite l'apprentissage tourne mais moins facile pour obtenir la convergence. Au contraire, plus le taux d'apprentissage est petit, moins vite l'apprentissage tourne mais plus facile pour

obtenir la convergence. Les deux situations peuvent faire augmenter le nombre d'époques qu'il faut pour une convergence, le temps d'apprentissage. Il s'agit donc de bien tester pour trouver un taux idéal qui ne ralentit pas l'apprentissage de manière considérable. Dans notre projet, on a trouvé que le taux le plus performant est de 0.001.

Par ailleurs, le nombre des combinaisons des hyperparamètres détermine le temps d'apprentissage aussi. Il va de soi que ces deux chiffres sont en corrélation positive, puisque le nombre d'époques qui va tourner grandit avec le nombre des combinaisons.

Finalement, en ce qui concerne **le seuil probabiliste du filtrage**, nous n'avons pas constaté d'impacts importants liés aux changements de cet hyperparamètre.

En résumé, afin de réduire le temps d'apprentissage pour traiter les grands corpus et étant donné les nombres d'époques optimaux, il est utile de **diminuer la dimension d'embeddings**, d'**agrandir la taille d'un batch**, de **chercher un taux d'apprentissage peu ralentissant**, et de **réduire le nombre de combinaisons des hyperparamètres**.

d. Autres difficultés et solutions

(1) Environnements CPU/GPU:

Idéalement, l'algorithme s'exécute dans un environnement GPU. Vu qu'aucun membre ne dispose de GPU sur le PC, nous avons travaillé sur Google Colab. Pourtant, Google Colab a des inconvénients signifiants qui gênent l'exécution du code :

a) Un plafond pour la capacité de la RAM de 12,69 GB pour un service gratuit ;
b) Une limite GPU après 12 heures d'utilisation consécutives pour un service gratuit ;

Quant au premier inconvénient, notre code de la première version faisait planter Google Colab car nous atteignons facilement la limite de la RAM, surtout pendant l'apprentissage. Cela s'est résolu avec la nouvelle version du code qui gère mieux les représentations vectorielle et matricielle des entrées, mais il faut tout de même éviter de tester trop de combinaisons des

hyperparamètres en une opération d'apprentissage.

Pour combattre le deuxième inconvénient, nous avons créé des sous-corpus pour réduire le temps du prétraitement et de l'apprentissage. Nous avons également exécuté le code sur Google Colab en utilisant différents comptes, malgré le risque d'être interdit d'utiliser le GPU sur tous les comptes concernés.

À la fin, nous avons travaillé sur le PC individuel de chacun en parallèle avec Google Colab pour accélérer les expériences.

(2) Choix d'optimiseur

Durant les expériences, nous avons trouvé que l'optimiseur SGD produisait les pertes croissantes pour l'entraînement et pour la validation, ce qui est un résultat indésirable en vertu de la fiabilité du modèle. Face à cette difficulté, nous avons remplacé le SGD par Adam. Avec Adam, nous avons réussi à obtenir des pertes décroissantes pour l'entraînement et rendre notre modèle plus fiable.

(3) Quantité d'hyperparamètres

Il est essentiel de tester plusieurs combinaisons pour déterminer lesquelles donnent des résultats satisfaisants. Une difficulté est que le nombre de combinaisons peut être trop grand en termes de temps d'attente, de capacité CPU/RAM et d'efforts pour la modification manuelle. Pour faciliter cette tâche, nous avons implémenté le module *itertool* pour prendre plusieurs valeurs pour un hyperparamètre, et ainsi pour tous les hyperparamètres testés. Toutes les combinaisons peuvent donc être testées en une opération d'apprentissage et les résultats imprimés en un fichier .csv. Le seul enjeu de cette solution consiste à ne pas mettre trop de combinaisons en une seule opération, sinon l'apprentissage prendra des heures pour terminer.

e. Dimensions à implémenter pour l'avenir

En raison des contraintes de temps, il nous reste encore des pistes à explorer sur le sujet. Voici quelques-unes :

a. Accélération du prétraitement

Les fonctions auxiliaires que nous implémentons maintenant seront moins utiles si elles reçoivent de nouveaux corpus de grande taille. Il faudrait optimiser le prétraitement avec d'autres manières pour que cette étape s'exécute plus rapidement. Une piste éventuelle serait d'enregistrer tous les candidats de mots contextes négatifs en fichiers pour pouvoir sauter le prétraitement à chaque apprentissage sur un même corpus.

b. Analyses statistiques pour identifier les meilleures combinaisons

Nous avons observé les résultats de manière simple sans avoir appliqué les outils statistiques. De meilleures combinaisons pourraient être identifiées à l'aide des analyses statistiques avancées.

c. Evaluation sur le modèle en utilisant la similarité cosinus

Nous nous sommes concentrés sur l'efficacité de l'algorithme pour les grands corpus. Il nous reste à bien vérifier la performance du modèle à travers de la similarité cosinus sur des paires de mots choisis, ou bien d'un système de prédiction des mots contextes à partir des mots cibles pour calculer l'accuracy.

d. SGD

Les qualités et les inconvénients de l'optimiseur SGD peuvent être explorés plus profondément pour voir s'il pourrait produire des résultats aussi satisfaisants, voire plus, qu'Adam.

CONCLUSION

Le modèle Skip-Gram avec échantillonnage négatif est une version qui marche beaucoup plus efficacement que le modèle Skip-Gram tout seul. Néanmoins, on constate que quand la quantité de données à traiter devient très grande, ce modèle coûte cher aussi en termes chronologiques. Il est donc judicieux d'appliquer les représentations vectorielle et matricielle pour accélérer le prétraitement et l'entraînement. Par le biais des opérations vectorielles sur les données organisées en batch, le temps d'apprentissage

se réduit de façon considérable.

À travers ce projet préliminaire, nous avons pu identifier les hyperparamètres qui permettent de profiter des opérations vectorielles afin de réduire le temps requis. Cette liste des hyperparamètres comprend la taille du corpus, le ratio d'exemples négatifs contre les exemples positifs, la dimension d'embeddings, la taille d'un batch et le nombre d'époques. Nous avons trouvé que quand *emb_dim = 200*, *batch_size = 1024*, *min_keep = 0.7* et *learning_rate = 0.001*, *neg_pos_ratio = 4*, les grands corpus ayant plus de 10 000 lignes peuvent être traités en moins de temps, et l'idéal c'est de tourner l'apprentissage sur la GPU. Néanmoins, vu que la GPU n'est pas toujours disponible sur la machine qu'on utilise, il est essentiel d'optimiser les hyperparamètres ci-dessus, ainsi que de choisir des méthodes efficaces comme, par exemple, un type d'optimiseur qui marche bien, ou les analyses statistiques et les systèmes d'évaluation qui aident à déterminer .

Faute de temps, nous n'avons pas pu inclure toutes ces considérations dans notre projet. Ce qui n'est pas encore étudié restera des pistes à découvrir dans le futur.

ANNEXE I MANUEL D'UTILISATEUR

Ce manuel d'utilisateur a pour but d'expliquer l'utilisation du programme SGNS (Skip-Gram avec échantillonnage négatif) en ligne de commande.

Objectif

Le programme permet à l'utilisateur de faire dérouler l'apprentissage sur un corpus de choix, ainsi de saisir les hyperparamètres de son choix.

Lecture du fichier

- Mettez le code X.py et le corpus à utiliser dans le même dossier.
- Ouvrez le Terminal et allez dans le dossier ci-dessus.
- Saisir la commande python X.py nom_du_fichier.
- Notez qu'il y a un espace entre les deux segments de commande, et que le fichier devrait être en format .txt.

Manipulation des hyperparamètres

- Pour utiliser les hyperparamètres par défaut, répondez non ('N') et puis oui ('Y').
- Pour saisir vos propres hyperparamètres, répondez oui ('Y') et :

1. Saisir une ou plusieurs valeurs pour chaque hyperparamètres. Mettez bien un espace entre deux valeurs et assurez que les valeurs ne dépassent pas les marges recommandées dans les parenthèses.
2. Vérifier les hyperparamètres que vous avez saisis avec l'affichage des résultats.(3) Pour les modifier, répondez oui ('Y'). Sinon, répondez non ('N') et déclenchez l'apprentissage.

- Pour terminer l'apprentissage, répondez oui ('Q') aux questions concernées.

_Enregistrement des résultats

Chaque apprentissage terminé produit un fichier result.csv qui enregistre les valeurs et les résultats concernés.

Deux fichiers textuels neg_nom_du_corpus.txt et pos_nom_du_corpus.txt sont créés dans le même dossier du programme au cours de l'apprentissage sur un nouveau corpus.

S'il s'agit d'un corpus déjà appris qui doit être appris, mettez les deux fichiers en .txt dans le même dossier du programme pour gagner du temps.

Pré-requis:

Matériel: Le modèle Word2Vec SGNS peut fonctionner sur le Google Colab et sur un ordinateur avec un minimum de 4 Go de mémoires vives. Notez que l'apprentissage sur un grand corpus exige davantage de mémoires vives. Il sera idéal de l'exécuter sur le GPU.

Dépendances logicielles: L'exécution du Word2Vec nécessite Pytorch pour gérer l'optimisation des paramètres du modèles et les calculs sur les tenseurs. En outre, cela exige l'installation d'un certain nombre de bibliothèques Python, notamment, tqdm, NumPy et Pandas. Par souci de compatibilité des fonctionnalités et des bibliothèques, il est recommandé d'installer Python 3.7 ou des versions plus récentes.

Auteurs
Wei HU, Zong-You KE & Vo Tuan Anh AN

ANNEXE II STATISTIQUES DES RÉSULTATS ET LEUR REPRÉSENTATION GRAPHIQUE (cf. Stats & Graphs)

RÉFÉRENCES

1. Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean (2013) Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781
2. Thomas Mikolov et al (2013): Distributed Representations of Words and Phrases and their Compositionality. NIPS'13 Proceedings of the 26th International Conference on Neural Information Processing Systems
3. Dan Jurafsky, James H. Martin (2020) Speech and language processing
4. Demystifying Neural Network in Skip-Gram Language Modeling | Pythonic Excursions (aegis4048.github.io)
5. Word2Vec -Negative Sampling made easy

SOMMAIRE