MODÈLE 'WORD2VEC' POUR LA CONSTRUCTION DE VECTEURS DE MOTS

Projet final de master 1 linguistique-informatique Université de Paris-cité

Eléonora Khachaturova Armand Garrigou Léo Rongieras

Table of contents

List of Figures

List of Tables

1 Introduction

Le modèle Word2Vec a marqué une étape significative dans le domaine du traitement du langage naturel, en révolutionnant la manière d'obtenir des représentations la signification des mots. Depuis son introduction par Mikolov et al. (2013), Word2Vec est devenu l'un des modèles les plus influents et largement adopté pour la représentation des mots dans les tâches de traitement automatique du langage.

L'objectif de ce rapport est de présenter en détail notre implémentation du modèle Word2Vec et de discuter des résultats obtenus lors de nos expérimentations. Nous aborderons en premier lieu les fondements théoriques du modèle.

Le modèle Word2Vec se base sur l'hypothèse distributionnelle, selon laquelle les mots ayant des contextes similaires ont tendance à partager des significations similaires. En exploitant de vastes corpus de textes, Word2Vec apprend des représentations vectorielles denses pour chaque mot, capturant ainsi les relations sémantiques et syntaxiques entre les mots. Ces représentations vectorielles, souvent appelées embeddings, ont été utilisées avec succès dans de nombreuses applications telles que la traduction automatique, la recherche d'informations et la classification de documents.

Dans notre projet, nous avons souhaité développer notre propre implémentation du modèle Word2Vec, afin de mieux comprendre son fonctionnement interne. Cette approche "from scratch" nous a permis d'explorer en profondeur les mécanismes de Word2Vec, depuis la création des fenêtres contextuelles jusqu'à l'entraînement du réseau neuronal.

Nous avons défini les objectifs suivants pour notre projet : (1) implémenter l'algorithme Word2Vec en utilisant le modèle CBOW, (2) entraîner notre modèle sur un corpus de texte de grande envergure, et (3) évaluer la qualité des embeddings appris, en utilisant différentes mesures de similarité sémantique et en effectuant des tâches d'analogie de mots.

L'intégralité de notre code est retrouvable sur notre github

2 Approche théorique

Le modèle Word2Vec propose deux architectures principales : Skip-gram et CBOW (Continuous Bag-of-Words). Dans notre implémentation, nous avons choisi de nous concentrer sur l'architecture CBOW. Contrairement au modèle Skip-gram qui prédit les mots environnants à partir d'un mot central, CBOW utilise le contexte environnant pour prédire le mot central. Cette approche nous permet d'apprendre des embeddings de mots en exploitant les relations contextuelles.

Par exemple en supposant une taille de fenetre de deux : pour la phrase "le chat dort les souris chantent", si notre mot central est "dort" notre contexte sera défini par [le, chat, les, souris]. Comme le montre le schéma suivant :

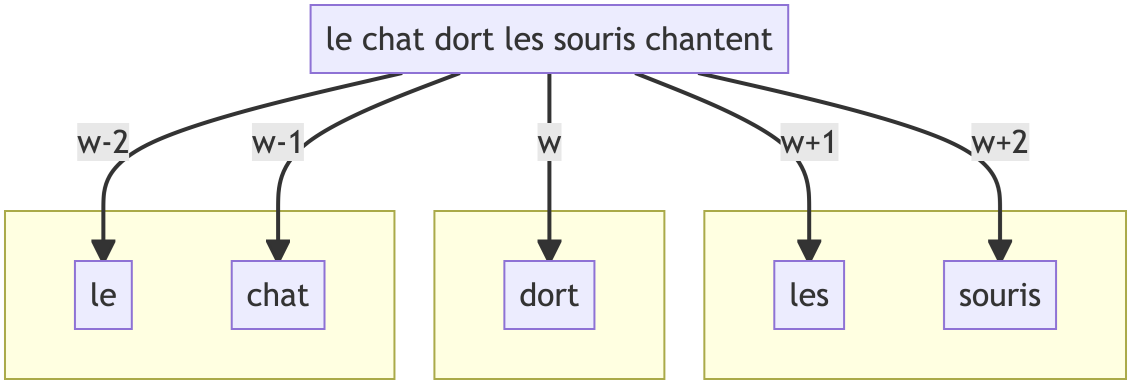


Schéma de la décomposition mot-cible/contexte

L'idée qu'un mot est déterminé par son contexte est parfaitement explicitée dans les modèles de langue comme les modèles n-grams, on l'on assume que la probabilité de la présence d'un mot dans une phrase est déterminée par les mots qui précèdent, soit pour un modèle bi-gram :

$$P(w_1, w_2, ..., w_n) = \prod_{i=2}^{n} P(w_i|w_{i-1})$$

.

L'intérêt de l'architecture CBOW est qu'elle permet d'exploiter aussi les mots qui apparaissent *après* notre mot cible. Comme le montre notre schéma, contrairement à un modele n-grams traditionnel, le contexte *global* est pris en compte comme information. Un des apports majeur

du modèle word2vec est donc se passage à la *bi-directionalité*, prendre le contexte de gauche et de droite.

Nous allons donc voir comment CBOW peut apprendre ces probabilités.

Notre modèle prend en entré un vecteur contexte x et retourne un vecteur mot y qui correspond au mot au centre de notre contexte. On définit deux matrices $\mathcal{A}^{(c)} \in \mathbb{R}^{|V| \times n}$ et $\mathcal{A}^{(w)} \in \mathbb{R}^{n \times |V|}$ On note :

- w_i le mot en position i du vocabulaire V
- $\mathcal{A}^{(c)} \in \mathbb{R}^{|V| \times n}$ la matrice des mots en entré où a_i correspond à la i-ème ligne de $\mathcal{A}^{(c)}$ c'est à dire le vecteur qui représente le mot en entré w_i
- $\mathcal{A}^{(w)} \in \mathbb{R}^{n \times |V|}$ la matrice en sortie où y_i correspond à la i-ème colone de $\mathcal{A}^{(w)}$ c'est à dire le vecteur qui représente le mot en sortie w_i
- n correspond à la dimension arbitraire des embeddings

Les étapes du fonctionnement du modèle peuvent être décrites de telle sorte:

1. On crée un vecteur d'entré composé des indices i des mots $\in V$ en contexte avec une fenètre de taille N soit
   $v^c = (x^{c-N}, ..., x^{c-1}, x^{c+1}, ..., x^{c+N})$
2. On obtient nos embeddings pour ce contexte. Soit $\mathcal{A}^{(c)} v^c$, on obtient une matrice de taille $N \times n$ ou chaque ligne i correspond à l'embeddings du mot en contexte. Chaque vecteur dense est de dimension n
3. On veut récupérer la somme des vecteurs appartenant au contexte C, c'est à dire la somme des éléments **par colone** de notre matrices $\mathcal{A}^{(c)} v^c$ soit
   $$\hat{x} = \sum_{i=1}^{N \times 2} a_i$$
4. Notre vecteur score z est obtenu par
   $$z = \mathcal{A}^{(w)} \sum_{i=1}^{N \times 2} a_i = \mathcal{A}^{(w)} \hat{x}$$
5. On retourne ce score transformé en log probabilité soit $\hat{y} = \text{log\_softmax}(z)$

On peut donc remarquer que après l'application du $softmax$ on obtient un vecteur $\hat{y}$ de la taille du vocabulaire V, où chaque $\widehat{y_i}$ correspond à la *log-probabilité* que $\widehat{y_i}$ soit le mot cible du contexte en entrée.

On peut le vérifier algébriquement : en effet notre score est obtenu par le scalaire entre la matrice $\mathcal{A}^{(w)} \in \mathbb{R}^{n \times |V|}$ et le vecteur $\hat{x}$ de taille n on a donc :

$$\begin{bmatrix} y_{1,1} & y_{2,1} & \cdots & y_{|V|,1} \\ y_{1,2} & \cdots & \cdots & \cdots \\ y_{1,3} & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ y_{1,n} & \cdots & \cdots & y_{|V|,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} \widehat{y_1} & \widehat{y_2} & \widehat{y_3} & . & . & . & \widehat{y_{|V|}} \end{bmatrix}$$

On obtient donc un vecteur $\hat{y}$ de taille $|V|$

On peut résumer l'ensemble avec le schéma suivant[1] :

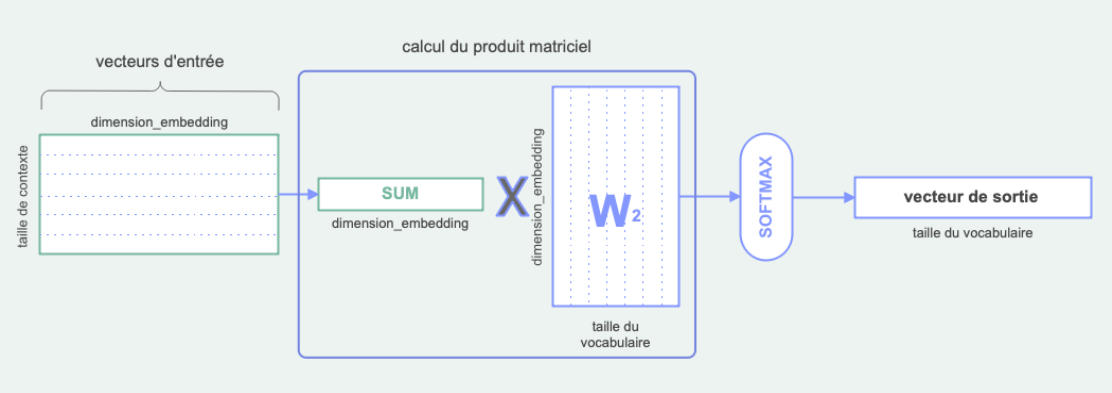


Figure 2.1: Architecture de modèle CBOW

en dernier lieu, le *softmax* prend en entré le vecteur de score z de taille $|V|$ et renvoie un vecteur de même dimension, où chaques composants i est défini comme :

$$(\text{softmax}(z))_i = \frac{e^{z_i}}{\sum_{j=1}^{|V|} e^{z_i}}$$

[1] On peut noter qu'il n'y pas dans l'architecture CBOW de fonction d'activation, la seule couche cachée du réseau correspond justement à la somme des embeddings des mots en contexte que nous avons décrit, le nombre de neurones correspond simplement à la dimension de nos embeddings.

3 Méthodologie et implémentation

Nous implémentons notre modele avec les caractéristiques suivantes :

- utilisations de batch
- pas de negative sampling[1]

En choisissant une implémentation en batch, on assure une optimisation de computation. Si l'on regarde la définition du *softmax* on peut se rendre compte que ce calcul est en fait computationnellement lourd, il parcourt l'ensemble du vocabulaire **pour tout mot**. Ce calcul est donc particulierement lourd. En utilisant les batchs, on se permet de passer quelques exemples à chaques itération, réduisant la lourdeur de ce calcul. De fait, l'utilisation d'une telle technique va nous permettre d'entrainer notre modèle bien plus rapidement, et sur de plus gros corpus de texte.

Le negative sampling est une technique introduite par Mikolov et son équipe. Bien qu'il s'agisse là aussi d'une technique d'optimisation computationnelle, nous ne l'implémenterons pas dans ce projet.

Notre implémtations repose sur l'utilisation du module `pytorch`. Ce module propose un ensemble de classes et de fonctions permettant d'optimiser nos taches et nos modèles bien plus facilemment. Nous verrons dans les parties suivantes comment nous avons utilisé ce module.

3.1 Données d'entrainement

Nous disposons de trois corpus d'entrainement. Deux sont de langue anglaise et directement issus de texte Wikipédia. Nous avons utilisé deux datasets `Wikitext2` et `Wikitext103`[2] qui est une collection de plus de 100 millions de tokens.

Pour le corpus en langue française, nous disposons d'une partie du corpus *frcow*. Il s'agit d'extraction de texte issue du web. La particularité de ce corpus et qu'il a été pré-traité avec un travail de lemmatisation[3].

[1] Initialement, Mikolov et al. (2013) avait introduit le negative sampling à partir de l'architecture Skip-gram
[2] Les datasets sont par exemple disponibles sur Hugging Face
[3] Nous remercions le professeur Olivier Bonami pour nous avoir transmis une partie de ce corpus comprenant son travail de lemmatisation

L'intéret d'avoir sous la main deux corpus de bonnes tailles est de pouvoir comparer l'apprentissage de notre modèle sur deux langues différentes et de deux types différents, un corpus plain text (`Wikitext103`) et un corpus lemmatisé (`frcow`) pour les besoins d'une étude morphologique.

Comme toutes taches de NLP, nous devons dans un premier temps préparer nos données à être passées dans le modèle, c'est que nous allons décrire dans la partie suivante.

3.2 Prétraitement des données

A partir des données nous avons besoins d'extraire :

- le vocabulaire : un `set` des mots présents dans notre corpus
- un map des mots par leurs indices dans le corpus : un `dictionnaire word2idx`
- une liste des mots triés par leurs indices : cette liste `idx2word` permet à partir d'un indice de récupérer le mot correspondant
- pour chaque mot parcouru sur notre corpus, on doit récupérer son contexte : un `tuple ([context], target)`

Pour récupérer le vocabulaire, `word2idx` et `idx2word` nous utilisons un module particulier de `pytorch` : `torchtext`. Nous pourrions récuperer ces informations avec des fonctions écrites "à la main", ce que nous avons fait lors des premiers tests de notre projet, cependant, si l'utilisation et la documentation de torchtext est parfois peu explicite et intuitive, ce module permet d'optimiser grandement la phase de prétraitement.

Ce module permet aussi de charger un dataset, dont notre dataset `Wikitext103`, ce qui est particulierement pratique car il n'est donc pas nécessaire de télécharger en amont notre dataset. Notre programme est exécutable sur la plupart des machines disposant d'une connexion internet, sans se préoccuper de la présence ou de l'emplacement du dataset.

La fonction `build_vocab_from_iterator` de `torchtext` permet d'obtenir depuis un texte un objet `vocab`. `Torchtext` optimise cette opération est permet de récupérer tout ce dont nous avons besoin (`word2idx`, `idx2word`) à partir des attributs de cette objet `vocab`.

Tout le prétraitement se fait dans le fichier `Prepro.py`. On doit retenir qu'en organisant les taches de cette manière, on se retrouve en fin de phase de prétraitrement avec un objet `preprocess` dont les attributs sont :

- un objet `Vocab` disposant d'un ensemble d'attributs dont :
  - un mapping `word2idx`
  - un mapping inverse `idx2word`

On peut obtenir ce dont on a besoin ainsi :

```
X = Preprocess(RAW_DATA, DATA_TYPE, WINDOW_SIZE, MIN_FREQ, BATCH_SIZE) ①
vocab = X.vocab
word2idx = vocab.get_stoi()
idx2word = vocab.get_itos()
```

① Les constantes passées en argument sont toutes dans un fichier `config.py`[4] qui est importé.

```
print(word2idx["king"])
```

287

- Un autre attribut de notre objet `preprocess` est simplement le `train_data`

```
train_data = X.train_data
```

et voici un exemple pour le premier batch, c'est à dire le premiere ensemble d'exemples que l'on peut trouver dans nos `train_data`:

```
batch numéro 0
Un exemple d'input: tensor([[  13,  239,  109,  ...,   15,    0,    2],
        [ 239,  109,    2,  ...,    0,    2,    0],
        [ 109,    2,  354,  ...,    2,    0,    2],
        ...,
        [ 304,   21, 2226,  ...,    0,  186,  368],
        [  21, 2226,   20,  ...,  186,  368,    0],
        [2226,   20,   21,  ...,  368,    0,   20]])

Le gold_label associé: tensor([   2, 1548,   11,  ..., 1411,    0,    2])
```

On comprend ici l'importance de l'utilisation de batch[5]. On a donc, pour nos données d'input, plusieurs contextes, en l'occurrence l'indice des mots en contexte. Ce qu'on passe en `input`[6] de notre modele est donc une *matrice* qui comprends un ensemble de contexte. Le *gold_label* est donc par la même logique devenu un vecteur dont les éléments sont les gold_label associés à chaques contextes de notre matrice.(Dans notre optimisation : le vecteur input est devenu

[4] la description et l'usage de ce fichier est donné dans Section 6

[5] Cette étape est particulièrement importante. En réalité toute notre tentative d'optimisation repose sur cette idée de calcul matriciel, il a donc été très important pour nous de vérifier comme on le fait ici que l'on utilise bel et bien un systeme matriciel.

[6] Attention : ce ne sont pas encore des matrices d'embeddings, il s'agit des indices des mots en contexte, ce sont les données telles qu'elles sont prêtes à être envoyées dans le modèle.

une matrice de plusieurs vecteurs, le mot représentant le gold_label est devenu un vecteur de plusieurs gold_label)

On a donc notre phase de pré-traitement terminée. Nous allons décrire brievement l'implémentation du modèle CBOW.

3.3 Modèle CBOW

Nous avons contruit notre modèle à partir de la classe de base `Module`[7] proposée par `pytorch`. Nous pouvons visualiser l'ensemble de notre classe :

```
class CBOWModeler(nn.Module):

  def __init__(self, vocab_size, embedding_dim):
    super(CBOWModeler, self).__init__()
    self.embeddings = nn.Embedding(vocab_size, embedding_dim)          (1)
    self.linear1 = nn.Linear(embedding_dim, vocab_size)                (2)

    initrange = 0.5 / embedding_dim
    self.embeddings.weight.data.uniform_(-initrange, initrange)

  def forward(self, input):
    '''
    calcul de la somme des contextes à laquelle
    on applique la transformation linéaire (tensor : [1, len(vocab)])

    returns: log_softmax appliqué à la transformation

    '''
    embedding = self.embeddings(input)
    embedding = torch.sum(embedding, dim=1)                            (3)

    Z_1 = self.linear1(embedding)                                      (4)
    out = F.log_softmax(Z_1, dim=1)                                    (5)

    return out
```

① On retrouve ici notre matrices $\mathcal{A}^{(c)} \in \mathbb{R}^{|V| \times n}$ ou n est la dimension des embeddings
② Ici notre matrice $\mathcal{A}^{(w)} \in \mathbb{R}^{n \times V}$
③ La sommes des contextes
④ Le vecteur score z est obtenu par $z = \mathcal{A}^{(w)} \sum_{i=1}^{N \times 2} a_i = \mathcal{A}^{(w)} \hat{x}$

[7] voir nn.Module pour la documentation officielle

⑤ On retourne ce score transformé en log probabilité soit $\hat{y} = \text{log\_softmax}(z)$

Les classes et méthodes essentielles au programme sont décrites dans la documentation. Cette documentation en `.html` sera jointe au dossier final. Comme on peut le voir, notre classe suit plus ou moins explicitement ce qui a été décrit dans la première partie. La propagation consiste donc en la somme des contextes ainsi que l'application linéaire. Cette dernière retourne un `log_softmax`, un vecteur comprenant les log_probabilités pour chaque classe, dans notre cas, pour chaque mot du vocabulaire.

Voyons donc en dernier lieu comment nous avons conçu la phase d'entrainement.

3.4 Phase d'entrainement

Tout se passe dans le fichier `trainer.py`.

Notre modèle, les hyperparametres[8], les données d'entrainement sont importées dans ce fichier python. L'ensemble est instancié. Nous utilons comme `optimizer` l'algorithme `Adam`

```
grad = optim.Adam(cbow.parameters(), lr=LEARNING_RATE)
```

Il est possible d'indiquer au programme que l'on souhaite sauvegarder le modèle toutes les époques, ou selon un certain nombre de batch parcouru. Cela nous a été particulierement pratique pour effectuer des tests de manière systématique et de "jongler" avec différents hyperparametres.

A partir de cela nous avons effectué quelques experiences à partir de l'entrainement de notre modèle. L'ensemble de ces experiences est présenté dans la partie suivante.

[8] voir Section 6 dans laquelle on décrit comment configurer les hyperparametres

4 Resultats

4.1 Calculer la qualité des embeddings

Notre objectif dans cet exercice est d'obtenir des embeddings de bonne qualité, c'est à dire représentatifs du sens des mots qu'ils encodent. Pour déterminer si nos embeddings encodent des informations sémantiques, nous avons plusieurs moyens de procéder.

Nous allons nous baser sur la comparaison de nos vecteurs entre eux, sur des taches particulières. Nous pouvons dans un premier temps nous contenter de visualiser la distribution de nos embeddings dans l'espace vectoriel. Nos embeddings étant de taille 200 nous devons d'abord passer par un algorithme de réduction de dimensions afin de ne garder que les 2 dimensions les plus importantes de notre espace vectoriel. Nous obtenons donc des embeddings de dimension 2 et nous pouvons les visualiser. Nous donnons une représentation en deux dimensions pour le corpus anglais, donné en Figure 4.3 et français donné en Figure 4.4.

Un autre élément présent dans l'article publié par Mikolov et son équipe est la possibilté de rendre compte des relations sémantiques conjoites entre des paires de mots. Ainsi on peut remarquer sur la figure suivante que "Paris" est à "France" ce que "Berlin" est à "Germany" :

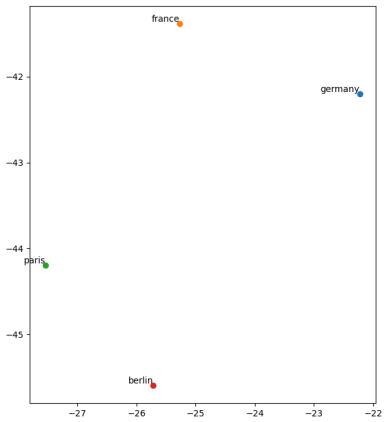


Figure 4.1: Paris/France, Berlin/Germany

plus largement on peut donc retrouver ce genre de relation sémantique graphiquement :

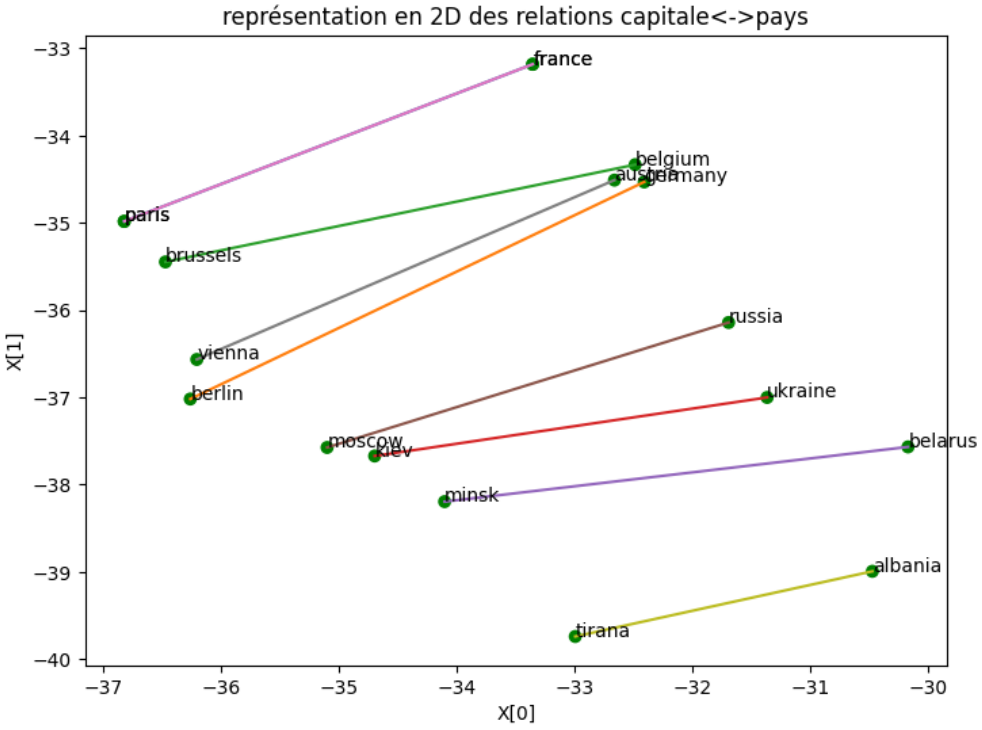


Figure 4.2: relation semantique

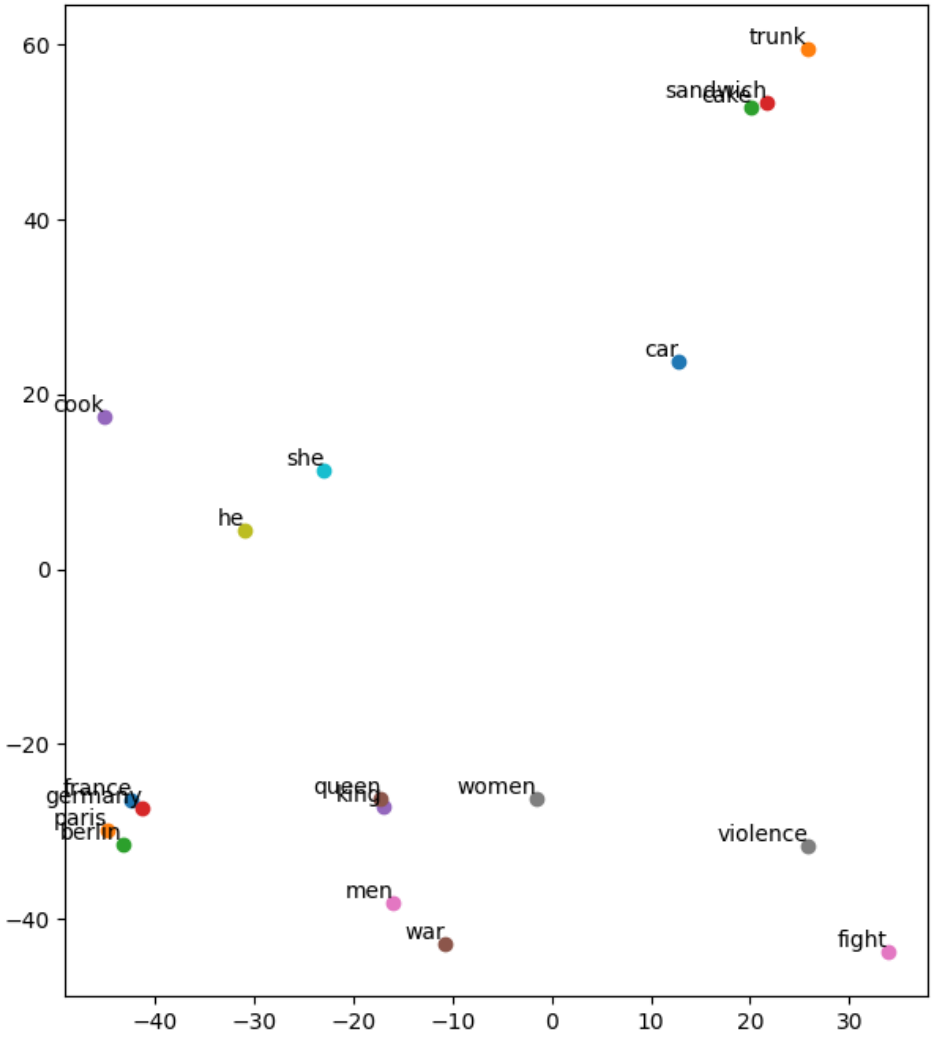


Figure 4.3: visualisation des embeddings en 2 dimensions *WikiText103*

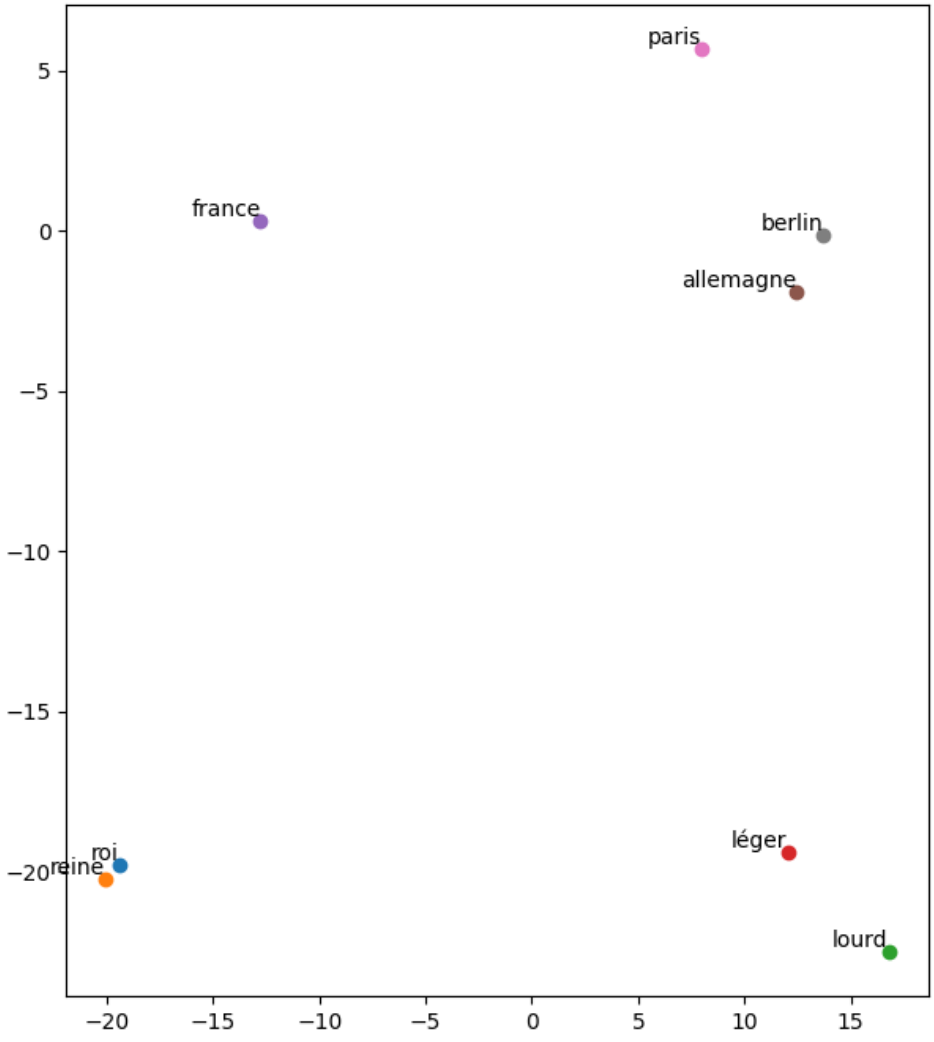


Figure 4.4: visualisation des embeddings en 2 dimensions *frcow*

Afin de confirmer ces résultats, nous pouvons mettre de coté la réduction de dimensions, qui implique forcément une perte d'information. Nous cherchons à utiliser une métrique fiable et constante pour calculer la proximité entre vecteurs. Traditionellement on utilise la distance euclidienne ou la similarité cosinus.

La distance euclidienne est une mesure de la distance entre deux points dans un espace vectoriel à plusieurs dimensions. La distance euclidienne entre deux points A et B est calculée en prenant la racine carrée de la somme des carrés des différences entre les coordonnées correspondantes des points. Mathématiquement, la formule de la distance euclidienne est la suivante :

$$||AB|| = \sqrt{(A_1 - B_1)^2 + (A_1 - B_2)^2 + ... + (A_n - B_n)^2}$$

Le calcul de similarité cosinus mesure l'angle entre deux vecteurs dans l'espace vectoriel. La similarité cosinus est une mesure de similarité normalisée qui varie entre -1 et 1. Une valeur de similarité cosinus proche de 1 indique une similarité élevée entre les vecteurs, tandis qu'une valeur proche de -1 indique des vecteurs opposés. Il est basé sur la formule mathématique suivante :

$$cos(A, B) = \frac{(A \bullet B)}{(||A|| \times ||B||)}$$

Ces deux métriques sont pertinentes pour notre tache, elles présentent toutes deux des avantages et des inconvénients. La distance euclidienne permet de calculer les vecteurs les plus proches (littéralement) d'un embedding en particulier. Toutefois, cette distance n'est pas entièrement bornée: elle peut être égale à 0 si deux vecteurs se superposent, mais n'a pas de limite naturelle supérieure. Se problème est résolu par la similarité cosinus qui est naturellement bornée entre -1 et 1. Le seul problème de cette métrique est que deux vecteurs peuvent avoir une similarité cosinus de 1 et pourtant ne pas avoir une distance euclidienne égale à 0. Traditionellement on préfère utiliser la similarité cosinus.

Par exemple :

Table 4.1: comparaison de similarité entre Paris/France et Paris/ham

	cosine	euclidien
france/paris	0.481537	3.64182
france/ham	-0.0108503	6.22445

On peut remarquer que selon nos mesures "France" est plus proche de "Paris" que de "ham". Plus proche littéralement comme l'indique la mesure de distance euclidienne mais aussi plus similaire selon notre mesure de cosinus, qui est plus proche de 1. On peut retrouver en Section 7 Annexe un tableau comparatif entre cosinus et distance euclidienne exercés sur une liste d'analogies

La tache sur laquelle nous allons pouvoir vraiment évaluer la qualité de nos embeddings sont les analogies. Elles sont utilisées dans l'article original qui introduit le modèle Word2Vec. Nos vecteurs sont représentables dans un espace euclidien et en respectent les règles:

- La distance entre deux points est toujours positive.
- La distance entre deux points est nulle si et seulement si les points sont identiques.
- La distance entre deux points est symétrique, c'est-à-dire que la distance entre A et B est la même que la distance entre B et A.
- La distance entre deux points obéit à l'inégalité triangulaire. Cela signifie que la distance entre deux points A et C est toujours plus courte que la somme des distances entre A et B, et entre B et C.

Ainsi on peut appliquer correctement les opérations mathématiques simples comme l'addition et la soustraction entre vecteurs. Mikolov et son équipe montre que le modèle Word2Vec permet d'appliquer l'addition entre vecteurs pour combiner les sens de deux mots, et inversement avec la soustraction. Nous arrivons donc au fameux example de leur article:

ROI - HOMME + FEMME = REINE

Si nous vérifions cette égalité avec notre modèle:

Table 4.2: mots les plus proches de (King-men+women)

word	cosine-sim
monarch	0.49
kings	0.41
queen	0.41
nobility	0.4
prince	0.4

On peut voir que le mot "queen" apparait en troisième position des mots les plus similaires au vecteur résultant du calcul algébrique. Ici aussi nous pouvons rendre compte de genre de relation graphiquement, par exemple :

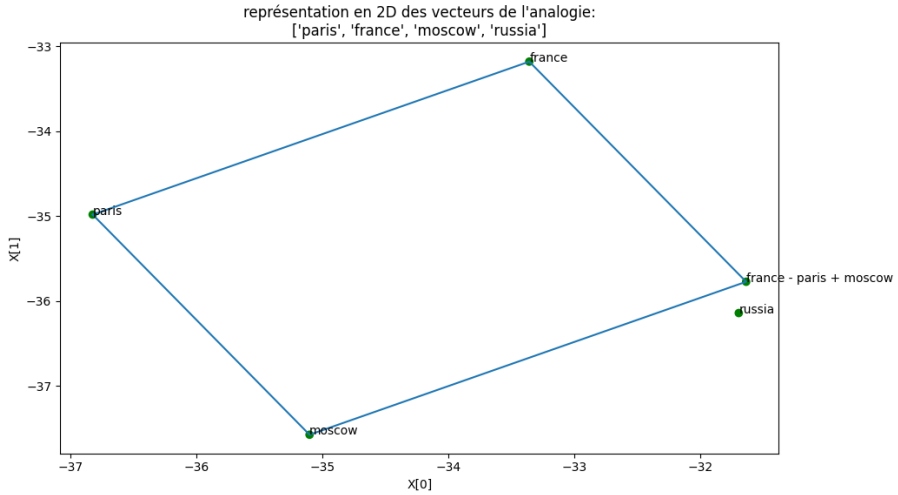


Figure 4.5: relation d'analogie

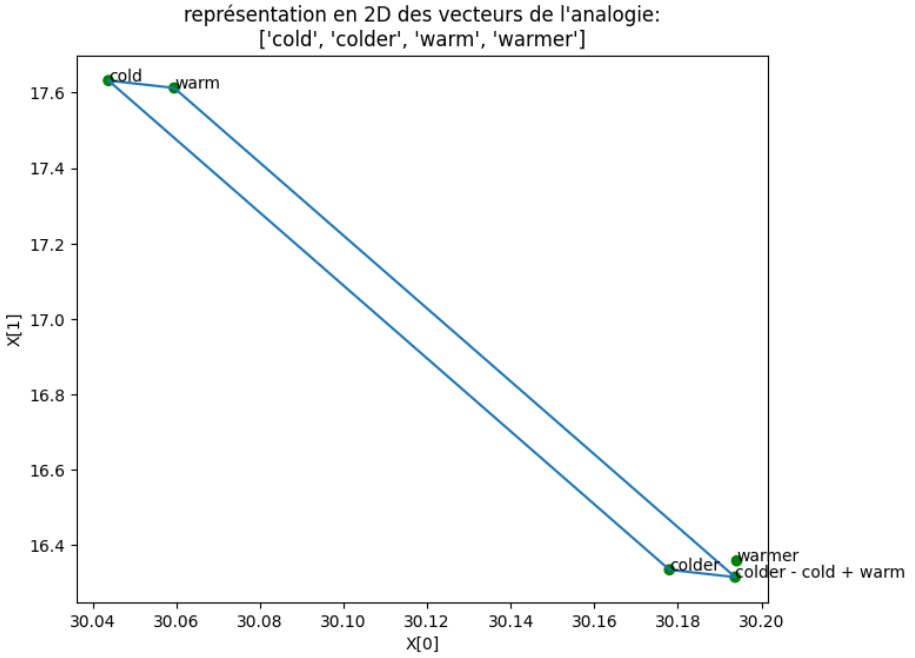


Figure 4.6: relation d'analogie

4.2 Analogies

Nous allons à présent vérifier que notre modèle performe des résultats similaires sur une liste d'analogies créée par le groupe de travail de Mikolov. Nous considerons qu'une analogie est validée si le vecteur attendu en sortie est dans les 5 embeddings les plus proches du vecteur résultant de la soustraction et de l'addition. L'ensemble des resultats est résumé par les graphes suivants.

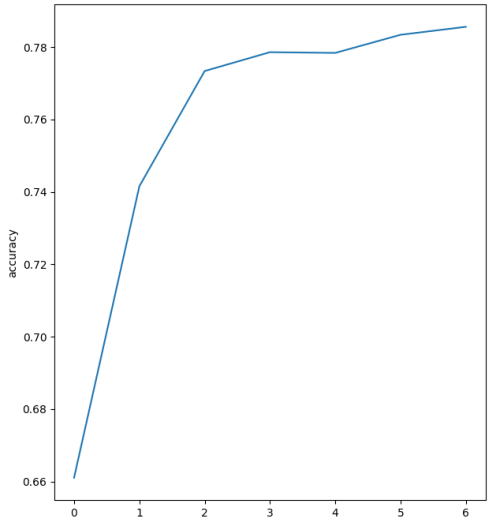
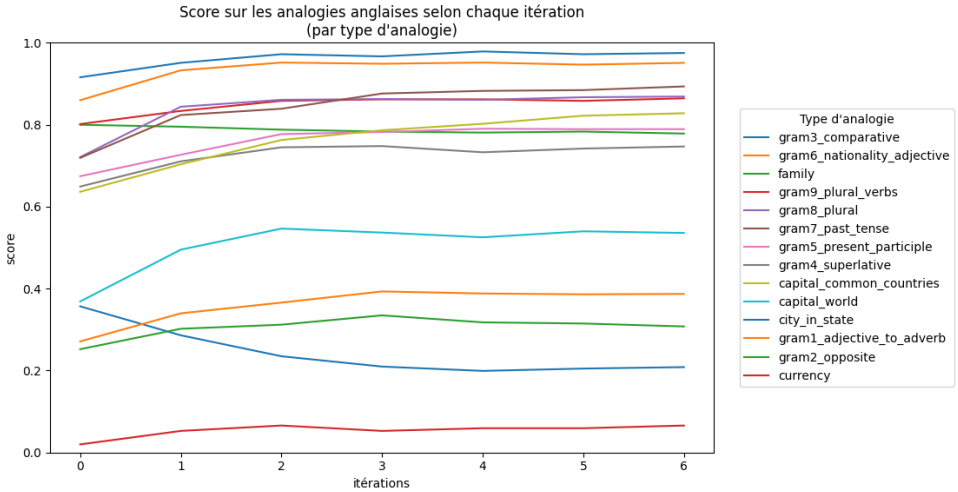


Figure 4.7: accuracy en fonction du nombre d'epoch sur une liste d'analogie



Les résultats sont bons, sachant que la random baseline de cet exercice serait de $5 \times \frac{1}{|V|}$, si nous considerons 5 examples par analogie. Nous pouvons essayer de comparer les performances de notre modèle selon le nombre d'itérations et aussi par rapport à FastText, qui fournit des embeddings entrainés sur la même architecture Word2Vec que la nôtre.

Catégorie	Accuracy	
	Notre modèle	FastText
family	0,779	0,957
capital_common_countries	0,828	0,968
capital_world	0,536	0,905
city_in_state	0,208	0,942
currency	0,066	0,618
gram1_adjective_to_adverb	0,387	0,923
gram2_opposite	0,308	0,897
gram3_comparative	0,975	0,998
gram4_superlative	0,747	1,000
gram5_present_participle	0,789	0,998
gram6_nationality_adjective	0,951	0,938
gram7_past_tense	0,894	0,926
gram8_plural	0,869	1,000
gram9_plural_verbs	0,865	0,996

Figure 4.8: tableau comparatif d'accuracy entre notre modèle et FastText sur une liste d'analogies

5 Limitation et piste d'amélioration

Comme vu précedemment, notre implémentation de word2vec atteint déjà de bonnes performances. Il existe toutefois de nombreuses pistes à explorer pour améliorer la qualité de nos embeddings.

Nous pourrions explorer plusieurs types de pré-traitement de notre corpus avant de lancer la vectorisation. Nous avons par exemple décidé de passer notre corpus en minuscule, pour réduire le bruit lié aux tokens en début de phrase, mais cela se fait au détriment d'une perte d'information sur les noms propres, nottament: "Bordeaux" devient "bordeaux" et les sens se melangent. Toutefois nous avons pensé que cette opération nous coute moins qu'elle nous rapporte. D'autres techniques de nettoyage du corpus comme la lemmatisation, la suppression des caractères spéciaux ou des nombres pourraient être prises en compte. Le cout de pré-traitement est particulièrement long maintenant que nous sommes capables de faire marcher notre modèle sur de grandes quantités de données.

Nous avons fait de nombreux tests sur les hyperparamètres afin de trouver ceux qui puissent allier bonne qualité d'embeddings et coût de l'entrainement (en termes de temps et de ressources). Nous avons nottament fait varier plusieurs hyperparamètres comme la taille de fenêtre, la dimensionnalité des vecteurs, le nombre d'itérations et le taux d'apprentissage. Toutefois nous n'avons pas gardé de trace de manière systématique des performance de notre modèle selon ces différents paramêtres. Il faudrait à l'avenir procéder de façon plus stricte, en passant par exemple par une grid search pour trouver la combinaison de paramêtres optimale, ou bien random search qui permettrait de trouver une combinaison suffisamment bonne en moins de temps.

Aussi, un axe d'amélioration serait d'augmenter la quantité de données, et de les diversifier. Notre modèle est déjà entrainé sur un nombre important d'exemples (le corpus `WikiText103` fait 100 millions de tokens), toutefois nous savons que plus les données sont conséquentes et variées et plus notre modèle pourra représenter fidèlement les mots de son vocabulaire. De manière générale, l'augmentation de la taille du corpus est souvent la première solution avancée. Toutefois, cela a un cout que nous ne jugions plus nécessaires vu l'état d'avancement du projet. Aussi, les données ajoutées au corpus d'entrainement doivent être de qualité: la diversité dans les exemples est primordiale car sinon les embeddings ne captureront pas totallement le sens des mots qu'ils encodent. Par exemple l'embedding de "robe" ne pourra pas encoder le sens de "couleur" d'un vin si il n'existe pas dans notre corpus d'entrainement de données parlant de vin. Cette problématique est présente dans le domaine de la désambiguisation lexicale, que plusieurs de nos camarades ont expérimenté cette année en projet de TAL.

Une autre problématique est la gestion des déséquilibres lexicaux. Les mots fréquents ont mécaniquement une meilleure qualité de représentation puisqu'ils sont présents dans une quantité plus grande et plus de contextes diversifiés. A l'inverse, les mots rares auront une qualité de représentation amoindrie. Afin de contrer cela, nous pourrions ne considérer qu'une quantité bornée d'examples par mot de vocabulaire, afin de permettre une couverture plus grande du corpus d'entrainement et ainsi augmenter le nombre d'exemples rencontrés pour les mots rares. Nous pourrions aussi implémenter le négative sampling, qui permet d'augmenter la quantité d'exemples par mot en associant un mot avec des mots aléatoires du vocabulaire et en soustrayant leurs sens.

Une des lacunes de notre implémentation vient aussi de la métrique d'évaluation. Elle se repose principalement sur la résolution d'analogies. Cette méthode est choisie car elle permet de juger de la qualité de plusieurs embeddings en même temps, tout en montrant qu'une addition/soustraction de vecteurs est similaire à une addition/soustraction de sens. Toutefois, cette métrique est biaisée, nottament car le corpus de ces analogies comporte de nombreux cas de mots rares. Nous ne considérons pas les analogies où l'embedding d'un mot n'est pas encodé par notre modèle, mais il subsite de nombreux exemples de mots mal encodés qui ne valident pas les analogies. C'est notamment le cas du sous corpus des analogies sur les monnaies par exemple, ou les mots sont assez fréquents pour être encodés mais pas assez pour avoir une représentation fidèle de leurs sens. Une autre proposition de métrique pourrait être de se baser sur un modèle dont on sait que les embeddings sont de bonne qualité. En effet, nous pourrions proposer une métrique qui calculerait l'embedding le plus proche (distance ou similarité) d'un mot x dans notre modèle et de déterminer si il est aussi présent dans les n embeddings les plus proches d'un modèle tiers. Nous pourrions par exemple nous baser sur les embeddings fournis par fastext, dont l'utilisation est simple et rapide.

Enfin, nous pourrions initialiser nos vecteurs non pas de manière aléatoire mais en récupérant directement ceux fournis par un modèle tiers. Nous pourrions ensuite fine-tuner les embeddings sur notre corpus d'entrainement. Cette technique nous permettrait de spécialiser nos vecteurs sur un corpus spécifique, ce qui pourrait être utile si nous avons besoin de vecteurs spécialisé sur un domaine en particulier.

6 Usage

> **Note**
>
> Nous avons décidé d'utiliser un fichier de configuration qui est importé dans chaque fichier plutot que d'utiliser un systeme en ligne de commande. Il y a beaucoup de variables que l'on peut configurer et nous avons trouvé cela plus facile d'utilisation et plus pratique.

L'ensemble des hyperparametres peut être modifié dans le fichier `config.py`. Liste des variables :

- `RAW_DATA` : `WikiText2`, `WikiText103` ou `frcow`

> **Warning**
>
> Wikitext103 et WikiText2 sont téléchargé par le programme, mais `frcow` doit être récupéré manuellement. Si l'on souhaite utiliser ce corpus on doit indiquer le `path` dans la variable `DATA_PATH`

- `DATA_PATH` : `path` vers le corpus `frcow`, neccessaire si `RAW_DATA = frcow`
- `DISPLAY_LOSS` : `boolean` affiche ou non la `loss` pendant la phase d'entrainement
- `EPOCH` : `int` nombre d'époch
- `BATCH_SIZE` : `int` taille du batch
- `MIN_FREQ` : `int` nombre d'occurrence minimum pour qu'un mot soit ajouté dans le vocabulaire
- `WINDOW_SIZE` : `int` taille de la fenètre contextuelle
- `EMBEDDING_DIM` : `int` dimension des embeddings
- `LEARNING_RATE` : `float` valeur du learning rate pour `ADAM`
- `TEST_WORDS` : `list[str]` liste de mot qui feront l'objet d'un `knn` lors de la phase d'entrainement
- `DISPLAY_N_BATCH` : `int` la fonction `knn` sera affichée tout les `n` batch

- `SAVE_N_EPOCH` : `int` le modèle sera sauvegardé toutes les `n` epoch

  Le reste des variables n'est pas à modifier.

Pour lancer l'entrainement il suffit de se placer dans le dossier courant, depuis un terminal :

```
python3 trainer.py
```

7 Annexe

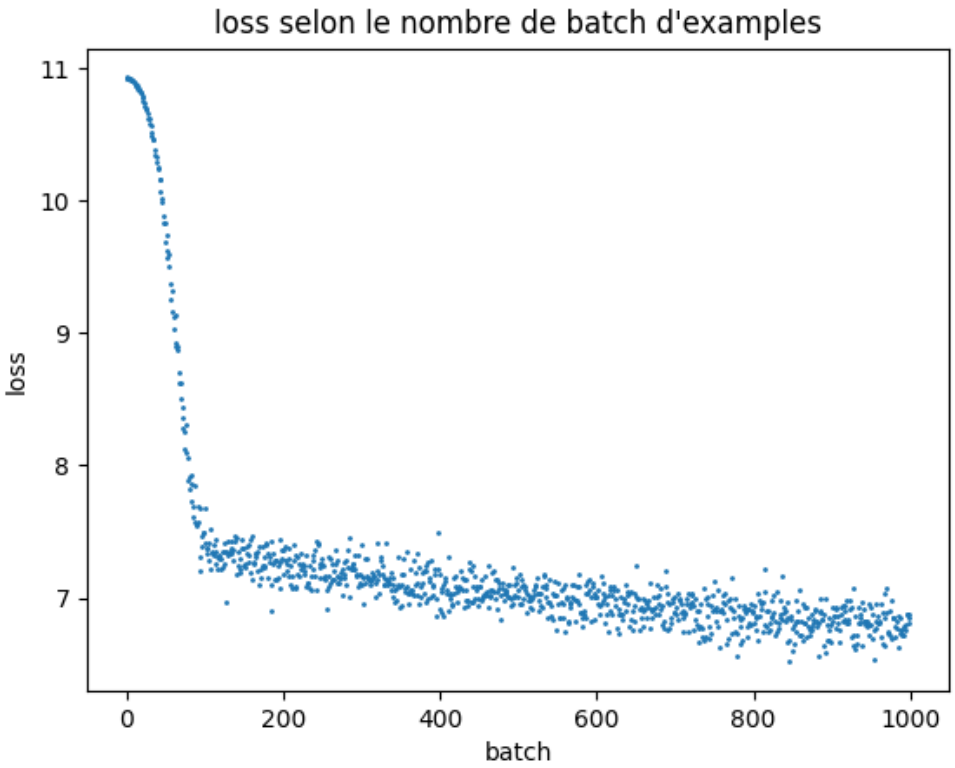


Figure 7.1: loss vs #batch *frcow*

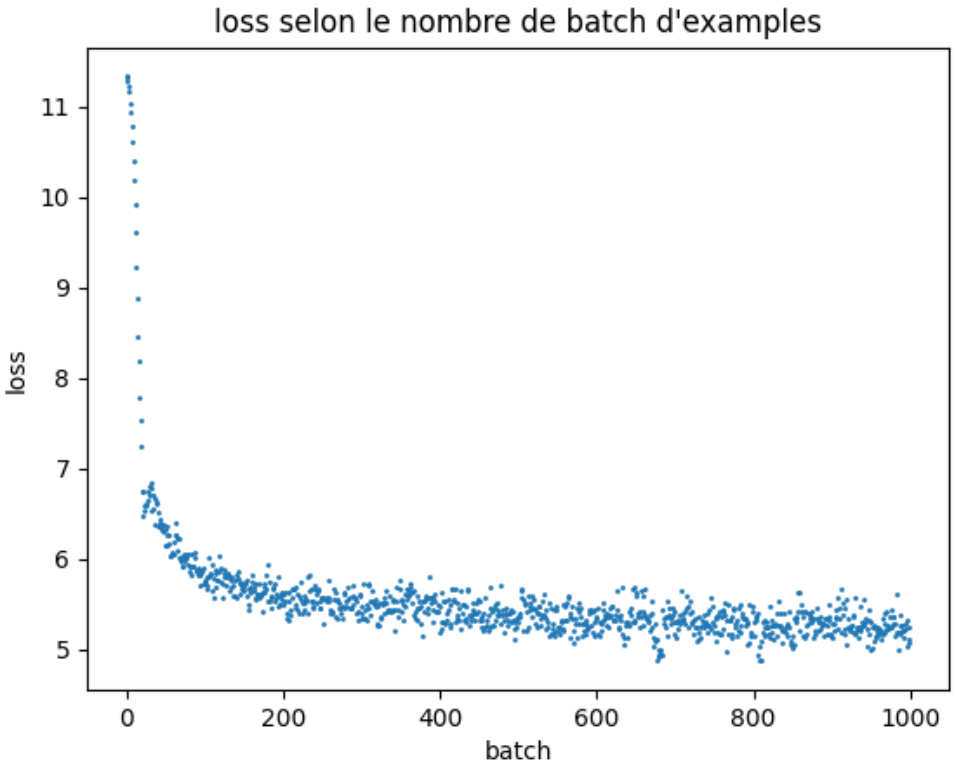


Figure 7.2: loss vs #batch Wikitext103

Catégorie	Accuracy de notre modèle	
	Similarité Cosinus	Distance Euclidienne
family	0,779	0,726
capital_common_countries	0,828	0,848
capital_world	0,536	0,516
city_in_state	0,208	0,209
currency	0,066	0,039
gram1_adjective_to_adverb	0,387	0,436
gram2_opposite	0,308	0,238
gram3_comparative	0,975	0,882
gram4_superlative	0,747	0,525
gram5_present_participle	0,789	0,768
gram6_nationality_adjective	0,951	0,953
gram7_past_tense	0,894	0,876
gram8_plural	0,869	0,835
gram9_plural_verbs	0,865	0,780

Figure 7.3: cosinus vs distance euclidienne

References

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Efficient Estimation of Word Representations in Vector Space." https://arxiv.org/abs/1301.3781.