INF8460 – Traitement automatique de la langue naturelle - Automne 2024

TP2: Plongements de mots

Identification de l'équipe:

Groupe de laboratoire: G03_B2

Equipe numéro : 02

Membres:

- Axel Morin (2306406) (33.3% de contribution, Tout le TP)
- Étienne Chevalier (2021242) (33.3% de contribution, Tout le TP)
- Kaywan Sanjari (1989922) (33.3% de contribution, Tout le TP)

L'inscription "Tout le TP" signifie que chaque étudiant a réalisé l'ensemble du TP de son côté et que les résultats ont été rassemblés en dernière instance. De la même manière les analyses ont été discutées et écrites ensemble au moment de la mise en commun.

- nature de la contribution: Décrivez brièvement ce qui a été fait par chaque membre de l'équipe. Tous les membres sont censés contribuer au développement. Bien que chaque membre puisse effectuer différentes tâches, vous devez vous efforcer d'obtenir une répartition égale du travail. Soyez précis ! N'indiquez pas seulement : travail réparti équitablement

Objectif du TP

L'objectif de ce TP est d'entraîner un modèle de plongements lexicaux qui intègre des notions de synonymie et d'antonymie en utilisant des réseaux de neurones.

Contrairement aux plongements vus en cours comme GloVe ou Word2Vec, qui positionnent souvent les mots à proximité équivalente de leurs synonymes et antonymes, nous chercherons à faire distinguer à nos modèles les synonymes et antonymes, en rapprochant les mots de leurs synonymes et en les éloignant de leurs antonymes.

Jeux de données

Paires de synonymes et antonymes (entraînement) : Les fichiers `train_synonyms.txt` et `train_antonymms.txt` contiennent les paires de synonymes et d'antonymes qui serviront à l'entraînement de nos modèles.

- train_synonyms: ~640k paires de synonymes
- train_antonyms: ~12k paires d'antonymes

SimLex-999 (test) : Le fichier `simlex_english.txt` contient 1000 paires de mots et leur similarité entre 0 et 10. Des antonymes auront une similarité de 0 et des mots proches auront une similarité plus élevée. Par exemples :

- *nice* & *cruel* -> 0
- *violent* & *angry* -> 5.9
- *essential* & *necessary* -> 9.8

Développement du TP

Le TP suivra les étapes suivantes:

- Partie 1 : Familiarisation avec GloVe, modèle de plongements de mots pré-entraîné
- Partie 2 : Évaluation de GloVe sur SimLex-999
- Partie 3 : Mise en place de la méthode d'entraînement
- Partie 4 : Entraînement de zéro (baseline)
- Partie 5 : Entraînement utilisant GloVe pré-entraîné et conclusion

Le TP est noté sur 89 points.

Librairies autorisées

- numpy
- pandas
- torch
- matplotlib

Si vous voulez utiliser une autre librairie, veuillez demander à votre chargé de lab.

Imports

Les imports effectués dans la cellule suivante devraient être suffisants pour faire tout ce TP.

In [1]:
```
from tqdm import tqdm
import gc

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
```

1. GloVe (10 Points)

Dans ce TP, nous allons utiliser le modèle pré-entraîné GloVe qui crée des plongements lexicaux de mots.

1.1 Chargement du modèle

La cellule suivante permet charger le modèle GloVe pré-entraîné. Le chargement du modèle peut prendre quelques minutes.

In [2]:
```
# # Décommenter pour télécharger les GloVe embeddings à partir de https://nlp.stanfora
# !wget http://nlp.stanford.edu/data/glove.42B.300d.zip -P /content
# !unzip /content/glove.42B.300d.zip -d /content
```

In [3]:
```
pretrained_model_path = 'glove.42B.300d.txt'

data_root = 'data'

train_synonyms_path = f"{data_root}/train_synonyms.txt"
train_antonyms_path = f"{data_root}/train_antonyms.txt"

eval_simlex = f'{data_root}/simlex_english.txt'
```

In [4]:
```
def load_word_vectors(filepath, vocab=None):
    """
    Télécharge le modèle pré-entraîné de plongements de mots en pytorch
    """
    word_to_index = {}
    embeddings = []
    index = 0

    with open(filepath, 'r', encoding='utf-8') as f:
        for line in f:
            split_line = line.split()
            word = split_line[0]
            if vocab is None or word in vocab:
                embedding = np.array(split_line[1:], dtype=np.float32)
                word_to_index[word] = index
                embeddings.append(embedding)
                index += 1

    embeddings = np.stack(embeddings)
    embeddings = torch.from_numpy(embeddings)
    return word_to_index, embeddings
```

In [5]:
```
word_to_index, embeddings = load_word_vectors(pretrained_model_path)
```

1.2 Implémentez la fonction `cosine_similarity` avec pytorch et sans utiliser `torch.nn.CosineSimilarity` (2 points)

In [6]:
```
def cosine_similarity(a, b):
    """
    Calcule la matrice de similarité cosinus entre deux matrices

    Args :
        a   : torch.Tensor, shape=(n, d)
```

```
        b   : torch.Tensor, shape=(m, d)

    Returns:
    torch.Tensor, shape=(n, m)
    """

    #Matrix multplication
    num = torch.mm(a, b.t())

    #Calculates the norm for each row of the matrix
    a_norm = torch.linalg.norm(a, dim=1, keepdim=True)
    b_norm = torch.linalg.norm(b, dim=1, keepdim=True)

    denom = a_norm*b_norm.t()


    return num / denom
```

In [7]:
```
#Example of cosine similarity
A = torch.Tensor([[0,1,0,5],
                  [1,2,5,6]])
B = torch.Tensor([[-1,0,-5,0],
                  [9,7,2,5],
                  [4,7,0,9]])
cosine_similarity(A,B)
#tensor([[ 0.0000,  0.4977,  0.8440],
#        [-0.6276,  0.6150,  0.7335]])
```

Out[7]:
```
tensor([[ 0.0000,  0.4977,  0.8440],
        [-0.6276,  0.6150,  0.7335]])
```

1.3 Complétez la fonction `n_closest_vect` qui retourne les n mots les plus proches d'un mot donné. (5 points)

`n_closest_vect` prendra en entrée la matrice des plongements `embeddings`, le dictionnaire de correspondance entre les mots et les indices `word_to_index`, le plongement d'un mot `word` et le nombre `n` de mots attendus. La fonction devra retourner la liste des mots dont les plongements sont les plus proches du vecteur de référence et leur similarité cosinus.

C'est-à-dire les n mots avec lesquels le mot a la plus grande similarité cosinus. Utilisez la fonction `cosine_similarity` que vous venez d'implémenter.

In [8]:
```
def n_closest_vect(embeddings, word_to_index, wordEmbedding, n=5):
    """
    Trouve les n mots les plus proches du vecteur donné et leur similarité

    Args :
        embeddings      : torch.Tensor, shape=(vocab_size, embedding_dim)
        Matrice de plongement de tous les mots

        word_to_index   : dict
        Dictionnaire qui relie un mot à son index dans le vocabulaire

        wordEmbedding   : torch.Tensor, shape=(embedding_dim,)
        Plongement du mot dont on cherche les n mots les plus proches

        n               : int, number of closest words to return
```

```
        Nombre de mots à retourner

    Returns:
    Liste de tuple contenant les n mots les plus similaires avec leur coefficient
    de similarité
    """

    #Evaluation de la matrice de similarité
    cosine_matrix = cosine_similarity(embeddings, wordEmbedding.unsqueeze(0))
    cosine_matrix.squeeze_()

    # Extraction des n plus grandes valeurs
    top_n_values = torch.topk(cosine_matrix, n)

    # Construction du dictionnaire de sortie
    top_n_words = {}

    i = 0
    word_keys = list(word_to_index.keys())
    for indice in top_n_values.indices:
      word_value = word_keys[indice]
      top_n_words[word_value] = top_n_values.values[i].item()
      i += 1

    return top_n_words
```

In [9]:
```
# Exemple

print(n_closest_vect(embeddings, word_to_index, embeddings[word_to_index['morning']]))
```

```
{'morning': 0.9999999403953552, 'afternoon': 0.8665473461151123, 'evening': 0.7880070
805549622, 'yesterday': 0.7614548206329346, 'sunday': 0.7548925876617432}
```

Sortie attendue :

```
[('morning', 1.0), ('afternoon', 0.8665473461151123), ('evening',
0.7880070209503174), ('yesterday', 0.7614548206329346), ('sunday',
0.7548925876617432)]
```

1.4 Quelle est la similarité cosinus entre 'fast' et 'slow' ? Entre 'fast' et 'rapid' ? Commentez les résultats et expliquez leur origine. (3 points)

In [10]:
```
fastEmbedding = torch.reshape(embeddings[word_to_index['fast']],(1,-1))
slowEmbedding = torch.reshape(embeddings[word_to_index['slow']],(1,-1))
rapidEmbedding = torch.reshape(embeddings[word_to_index['rapid']],(1,-1))

fastSlowRapidSimilarity = cosine_similarity(fastEmbedding, torch.cat((slowEmbedding,ra

print(f'{"":^10} |{"slow":^10} |{"rapid":^10}')
fastSlowStr = f'{float(fastSlowRapidSimilarity[0,0]):.5f}'
fastRapidStr = f'{float(fastSlowRapidSimilarity[0,1]):.5f}'
print(f'{"fast":^10} |{fastSlowStr:^10} |{fastRapidStr:^10}')
```

```
           |   slow     |  rapid
   fast    | 0.71137    | 0.64453
```

Comme le plongement se base sur le contexte, *slow* et *rapid* sont similaires à *fast* car on peut retrouver ces mots dans des contextes similaires (*slow* est même plus similaire à *fast* que *rapide*

ne l'est). Ce calcul montre que les antonymes sont mal traités puisque deux antonymes (*fast* et *slow*) sont autant, voire plus, similaires que deux synonymes (*fast* et *rapid*). Cette remarque permet d'introduire la probablématique du TP qui vise à palier cette incohérence.

2. Évaluation (12 Points)

Données

Les cellules qui suivent permettent de télécharger les données et de se restreindre au vocabulaire qui nous sera utile, pour éviter de charger des plongements inutiles.

Comme décrit dans l'introduction, nous avons 3 fichiers de données:

- Des paires de synonymes pour l'entraînement (`train_synonyms.txt`)
- Des paires d'antonymes pour l'entraînement (`train_antonymms.txt`)
- Des paires de mots avec leur similarité pour l'évaluation (`simlex_english.txt`)

In [11]:
```
def load_data(filepath):
    """
    Télécharge les paires de synonymes et antonymes
    """
    data = []
    with open(filepath, 'r', encoding='utf-8') as f:
        for line in f:
            pair = line.strip().split()
            assert pair[0].startswith('en_') and pair[0].startswith('en_')
            data.append((pair[0][3:], pair[1][3:]))
    return data

def data_to_tensor(data, word_to_index):
    indices = [word_to_index[word] for pair in data for word in pair if word in word_t
    return torch.tensor(indices).view(-1, 2)
```

In [12]:
```
# Données d'entraînement
train_synonyms = load_data(train_synonyms_path)
train_antonyms = load_data(train_antonyms_path)

# Données d'évaluation
evaluation_simlex = pd.read_csv(eval_simlex, sep='\t') # pd dataframe with columns 'wo

# On se restreint au vocabulaire qu'on va utiliser pour éviter de charger des embeddin
vocab = set([word for pair in train_synonyms + train_antonyms for word in pair])
eval_vocab = set(evaluation_simlex['word 1']).union(set(evaluation_simlex['word 2']))
vocab.update(eval_vocab)

glove_word_to_index, glove_embeddings = load_word_vectors(pretrained_model_path, vocab
```

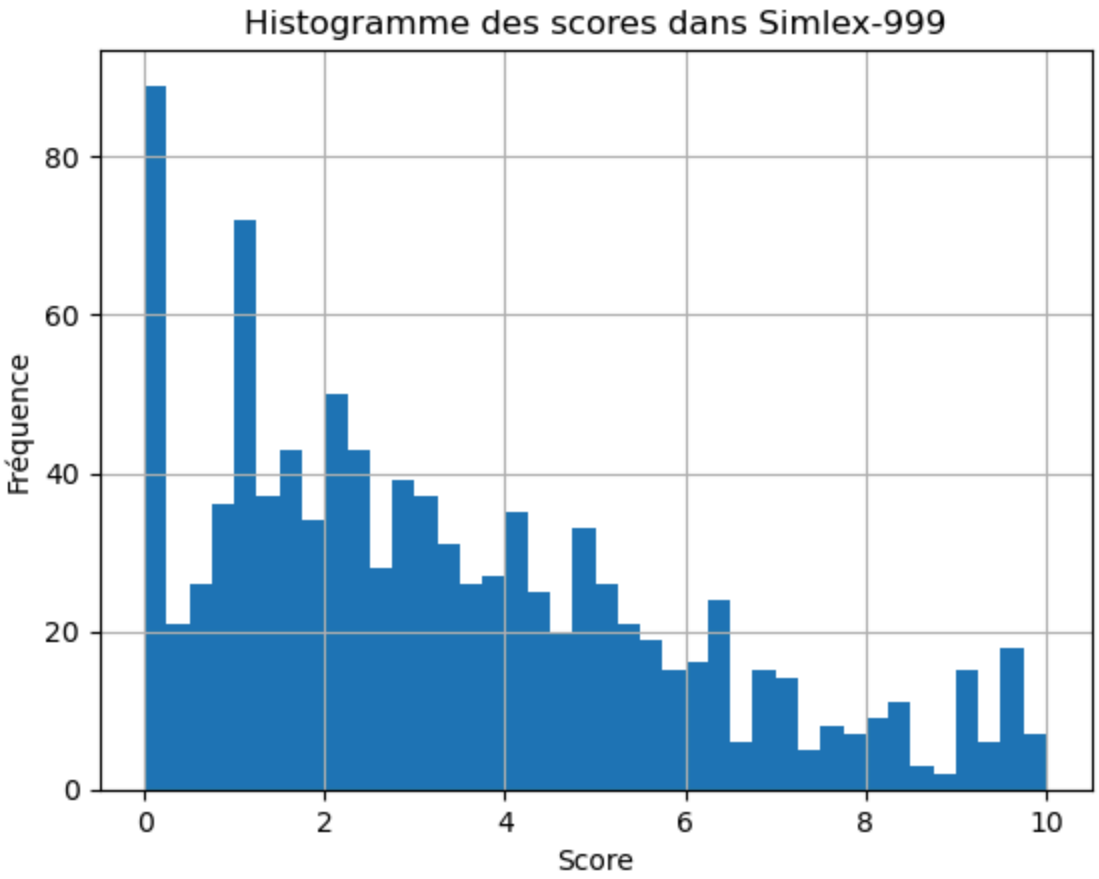
2.1 Observation du jeu de données SimLex-999. (2 points)

Affichez l'histogramme de fréquence des scores de similarité dans le jeu de données SimLex-999. Utilisez `bins=40` . Votre axe des x doit représenter le score de similarité et votre axe des y doit représenter la fréquence. Comment interpréter un score de 0?

In [13]:
```
plt.figure()
evaluation_simlex.hist('score',bins = 40,density = False)
plt.title("Histogramme des scores dans Simlex-999")
plt.ylabel("Fréquence")
plt.xlabel("Score")
```

Out[13]:
```
Text(0.5, 0, 'Score')
```

```
<Figure size 640x480 with 0 Axes>
```



Des scores de 0 dans le jeu de données SimLex-999 représentent une paire d'antonymes. Par exemple, le score de la paire (*old*,*new*) est de 0.

On retrouve aussi un score de 0 entre des paires de mots qui n'ont aucune lien évident entre eux comme la paire de termes (*ankle*, *window*)

2.2 Corrélation de Spearman

La corrélation de Spearman est une mesure de la relation monotone entre deux variables x et y. Elle est comprise entre -1 et 1. Plutôt que de comparer les valeurs brutes des variables, elle compare leurs rangs. Elle est calculée comme suit :

$$\rho(x, y) = 1 - \frac{6 \sum_{i=1}^{n} (r_{x_i} - r_{y_i})^2}{n(n^2 - 1)}$$

où r_{x_i} est le rang de la i-ème valeur de la variable x, r_{y_i} est le rang de la i-ème valeur de la variable y, n est le nombre total de paires d'observations (x, y).

Les rangs sont attribués en ordonnant les valeurs de chaque variable du plus petit au plus grand. La plus petite valeur reçoit le rang 1, la suivante le rang 2, et ainsi de suite.

Expliquez pourquoi on utilise la corrélation de Spearman plutôt que la corrélation entre les valeurs des variables. (3 points)

À la question 1.4, on a observé que les antonymes et synoynmes possédaient la même valeur de similarité. En revanche, dans le jeu de données SIMLEX-999, les antonymes avaient un score très faible (i.e. un rang très bas) et les synonymes avaient un score élevé (i.e. un rang supérieur). Si on ne s'intéresse pas à la valeur en tant que telle, on cherche la relation d'ordre qui dit que la similarité d'une paire d'antonymes et inférieure à celle d'une paire de synonymes, ce qui vient à dire que la comparaison de rangs fait plus de sens que de comparer les valeurs pour évaluer la similarité entre une paire de termes.

2.3 Implémentation de la corrélation de Spearman (2 points)

Complétez la fonction suivante pour calculer la corrélation de Spearman entre deux listes de valeurs.

In [14]:
```
def spearman_rank_correlation(x, y):
    """
    Calcule la corrélation de Spearman entre deux listes de valeurs.

    Args:
        x   : list of float
        y   : list of float

    Returns:
    La corrélation de Spearman entre les deux listes (float).
    """
    N = len(x)
    x = np.array(x)
    y = np.array(y)
    x_y = x-y
    factor = 6 / (N * ( (N ** 2) - 1))
    return 1 - factor * np.dot(x_y,x_y)
```

In [15]:
```
#Example of Spearman rank correlation
spearman_rank_correlation([1,3,4,2],[2,1,4,3])
```

Out[15]:
```
0.3999999999999999
```

2.4 Évaluation du modèle GloVe (2 points)

Retrouvez les plongements du modèle GloVe de tous les mots du jeu de données SimLex-999, puis calculez la similarité cosinus entre les paires.

Calculez ensuite la corrélation de Spearman entre les scores de simlex et les similarités cosinus obtenues et affichez-la.

In [16]:
```
similarityBetweenPairs = pd.DataFrame({"word 1" : list(evaluation_simlex["word 1"]),
                                       "word 2" : list(evaluation_simlex["word 2"])})
```

In [17]:
```
def getRank(array):
    order = array.argsort()
    return order.argsort()
```

In [18]:
```
# Calculate similarity between pairs
score = []
for i in range(len(similarityBetweenPairs)):
    score.append(float(cosine_similarity(
        torch.reshape(glove_embeddings[glove_word_to_index[similarityBetweenPairs['wor
        torch.reshape(glove_embeddings[glove_word_to_index[similarityBetweenPairs['wor
    )))
similarityBetweenPairs['score'] = pd.Series(score)

# Evaluate the rank (1 = lowest score, N = highest score)
similarityBetweenPairs['rank'] = getRank(similarityBetweenPairs['score'])
evaluation_simlex['rank'] = getRank(evaluation_simlex['score'])
```

In [19]:
```
# Evaluate Spearman correlation
print('Spearman correlation : ',spearman_rank_correlation(similarityBetweenPairs['rank
                          evaluation_simlex['rank']))
```

```
Spearman correlation :  0.29230790710550225
```

Résultat attendu: 0.29

2.5 Interprétation du résultat (3 points)

Qu'est ce que ce nombre représente et que peut-on en conclure sur la qualité des plongements GloVe (2 conclusions) ?

1. Étant donné deux listes de données rangées en ordre croissant, la correlation de Spearman permet de savoir si les données sont rangées de la même manière. Dans notre cas, connaître la correlation de Spearman du score de similarité de notre modèle glove permet de savoir si les scores représentent bien à quel point deux mots sont synonymes. Dans le meilleur des cas, un score de 1 signifierait qu'il existerait une tendance monotone croissante, donc que nos scores reflètent bien le caractère synonyme d'une paire de mots. Ici, la correlation retourne une valeur de .29 ce qui signifie qu'il est difficile de discerner une tendance monotone croissante.

2. Cette faible valeur réaffirme le point énoncé précédemment sur le fait que les plongements séparent très mal les antonymes des synonymes.

3. Description de la méthode contrastive (33 Points)

Nous allons maintenant implémenter une méthode contrastive de plongements de mots. Elle vise à améliorer les plongements lexicaux de mots en tenant compte des synonymes et antonymes.

Notre modèle se basera simplement sur une matrice de plongements de mots, qui associe à chaque mot un vecteur de plongement.

L'idée est d'entraîner ce modèle à rapprocher les plongements de synonymes et d'éloigner ceux d'antonymes.

La cellule suivante définit le modèle et ses attributs.

In [20]:
```
class ContrastiveWordEmbeddingModel(nn.Module):
    def __init__(self, embeddings, device='cpu', margin_plus=0.6, margin_minus=0., reg
        super(ContrastiveWordEmbeddingModel, self).__init__()

        self.device = device

        # Hyperparamètres pour les fonctions de coût
        self.margin_plus = margin_plus
        self.margin_minus = margin_minus
        self.regularization = regularization

        # Initialisation des plongements de mots
        self.embeddings = nn.Embedding.from_pretrained(embeddings.detach().clone(), fr
        self.original_embeddings = nn.Embedding.from_pretrained(embeddings.detach().cl
```

3.1 Création des négatifs

Pendant l'entraînement, au lieu de traiter tout le jeu d'entraînement d'un coup, nous allons avoir des lots (batchs) de paires de synonymes B_S et d'antonymes B_A.

Dans un lot de synonymes, on définit le négatif d'un mot comme le mot du lot le plus proche qui n'est pas dans la même paire. Intuitivement, c'est le mot que le modèle devrait confondre le plus avec le synonyme. Similairement, dans un lot d'antonymes, on définit le négatif d'un mot comme le mot du lot le plus éloigné qui n'est pas dans la même paire.

On répète ce processus pour chaque mot de chaque paire de synonymes et d'antonymes.

Attention, un mot peut apparaître plusieurs fois dans un lot avec des synonymes ou antonymes différents, et il ne peut être le négatif d'aucun de ses synonymes, ou antonymes.

3.1.1 Exemple pour illustrer l'implémentation

Prenons un exemple avec un lot B_S de synonymes de taille 3. On veut construire le lot de négatifs T_S

B_S:

- (arbre, plante)
- (voiture, véhicule)
- (arbre, buisson)

On a 5 mots uniques dans le lot: arbre, plante, voiture, véhicule, buisson. Supposons que la matrice de similarité cosinus soit la suivante :

	arbre	plante	voiture	véhicule	buisson
arbre	1	0.8	0.1	0.2	0.9

	arbre	plante	voiture	véhicule	buisson
plante	0.8	1	0.3	0.4	0.7
voiture	0.1	0.3	1	0.9	0.2
véhicule	0.2	0.4	0.9	1	0.3
buisson	0.9	0.7	0.2	0.3	1

On commence par calculer les voisins de chaque mot du lot B_S. Le voisin d'un mot m est défini comme tout mot qui apparait dans au moins une paire avec m dans B_S. Un mot est aussi considéré comme son propre voisin.

- voisins de arbre : arbre, plante, buisson
- voisins de plante : plante, arbre
- voisins de voiture : voiture, véhicule
- voisins de véhicule : véhicule, voiture
- voisins de buisson : buisson, arbre

Après avoir masqué les voisins, la matrice est :

	arbre	plante	voiture	véhicule	buisson
arbre	-inf	-inf	0.1	0.2	-inf
plante	-inf	-inf	0.3	0.4	0.7
voiture	0.1	0.3	-inf	-inf	0.2
véhicule	0.2	0.4	-inf	-inf	0.3
buisson	-inf	0.7	0.2	0.3	-inf

Pour calculer les négatifs, on prend le maximum de chaque ligne (donc le mot le plus similaire qui n'est pas un voisin) :

Ici,

- le négatif d'arbre est véhicule
- le négatif de plante est buisson
- le négatif de voiture est plante
- le négatif de véhicule est plante
- le négatif de buisson est plante

En reprenant le batch B_S:

- (arbre, plante)
- (voiture, véhicule)
- (arbre, buisson)

T_S sera composé de paires composées du négatif de chaque élément de B_S :

B_S -> T_S

- (arbre, plante) $\rightarrow$ (véhicule, buisson), car le négatif d'arbre est véhicule et le négatif de plante est buisson
- (voiture, véhicule) $\rightarrow$ (plante, plante), car le négatif de voiture est plante et le négatif de véhicule est plante
- (arbre, buisson) $\rightarrow$ (véhicule, plante), car le négatif d'arbre est véhicule et le négatif de buisson est plante

T_S sera donc :

- (véhicule, buisson)
- (plante, plante)
- (véhicule, plante)

3.1.2 Implémentez la fonction `prepare_neighbors` qui renvoit la liste des voisins de chaque mot dans le lot. (4 points)

Les voisins d'un mot m sont tous les mots du lot qui apparaissent dans au moins une paire avec m. Utilisez les bons indices (indice dans la matrice d'embeddings et indice dans le lot). Le résultat est une liste de liste de voisins, où `neighbors[i]` est la liste des voisins du mot `i` dans le lot.

In [21]:
```
def prepare_neighbors(index_pairs, unique_idx, index_to_idx):
    """
    Prépare les voisins pour chaque mot dans les paires de mots.

    Args :
        index_pairs     : torch.Tensor de seconde dimension 2
        Tensor contenant les indices des embeddings des mots dans le vocabulaire.
        Des indices qui sont reliés ensemble par une ligne dans ce tenseur ont
        une relation sémantique entre eux (synonymes ou antonymes).

        unique_idx      : set
        Ensemble de tous les indices qui sont mentionnés dans la liste `index_pairs`.

        index_to_idx    : dict
        Dictionnaire associant un indice mentionné dans `index_pairs` à son indice dan
        la liste qui sera retournée. Par exemple, si dans ce dictionnaire, la clé 4 es
        associée à la valeur 12, cela veut dire que les voisins du mot 4 dans le vocab
        seront retournés à l'indice 12 dans la liste de retour.

    Returns:
    Une liste où chaque élément est une liste des indices des voisins pour chaque mot
    """
    tempDict = {}
    # The strategy here is to first loop through all the pairs and create a dictionnar
    # of the words (eventually out.keys() = unique_idx) and the values are the indexes

    # Looping through the pairs
    for pair in index_pairs:
        pair0,pair1 = int(pair[0]),int(pair[1])
        # checkinng if the word has already been seen
        if pair0 not in tempDict.keys():
            # if not, create an empty lisr
            tempDict[pair0] = []
```

```
        tempDict[pair0] += [pair0,pair1]

        # Do the same for the other word
        if pair1 not in tempDict.keys():
            tempDict[pair1] = []
        tempDict[pair1] += [pair1,pair0]


    # right now we have a dictionnary linking a word's index to all the other words' i
    lenOut = max(index_to_idx.values()) + 1
    listOut = [[] for i in range(lenOut)]

    for word_idx in unique_idx:
        listOut[index_to_idx[int(word_idx)]] += [index_to_idx[el] for el in tempDict[i

    return listOut
```

In [22]:
```
# Exemple

index_pairsTEST = torch.tensor([[0, 12], [12, 31], [53, 4]])
unique_idxTEST = {0, 4, 12, 31, 53}
index_to_idxTEST = {0: 0, 4: 1, 12: 2, 31: 3, 53: 4}
print(prepare_neighbors(index_pairsTEST, unique_idxTEST, index_to_idxTEST))
```

```
[[0, 2], [1, 4], [2, 0, 2, 3], [3, 2], [4, 1]]
```

Réponse attendue

`[[0, 2], [1, 4], [2, 0, 2, 3], [3, 2], [4, 1]]`

3.1.3 Implémentez la fonction `select_negatives` qui renvoit un dictionnaire qui associe à chaque élément son négatif. (4 points)

Pour chaque élément du lot, on cherche le voisin le plus proche qui n'est pas le voisin de l'autre élément de la paire.

Utilisez un masque pour cacher, dans la matrice de similarité, les voisins.

La fonction utilise un paramètre `synonym` qui indique si on travaille sur un lot de synonymes ou d'antonymes. En cas de synonymes, on cherche le voisin le plus proche qui n'est pas un voisin de l'autre élément de la paire. En cas d'antonymes, on cherche le voisin le plus éloigné qui n'est pas un voisin de l'autre élément de la paire.

In [23]:
```
def select_negatives(indices, similarity_matrix, neighbors, synonym=True):
    """
    Sélectionne les exemples négatifs à partir de la matrice de similarité et des vois

    Args :
        indices             : torch.Tensor (vocab_size)
        Indices des mots présents dans le vocabulaire

        similarity_matrix   : torch.Tensor (vocab_size, vocab_size)
        Matrice de similarité entre tous les mots présents dans le vocabulaire.

        neighbors           : list of lists
        Liste des voisins de chaque mot. Par exemple, le premier élément de la liste
        contiendra tous les voisins du mot 0 dans le vocabulaire.
```

```
        synonym             : bool, optional (default=True)
        Indique si l'on cherche des négatifs pour les synonymes (True) ou pour les ant

    Returns:
    Dictionnaire mappant les mots avec leurs indices de négatifs {mot_index: négatif_i
    """
    vocabSize = len(indices)
    outDict = {}
    # looping through the similarity matrix rows
    for row in range(vocabSize):
        # mask the neighbors with +- inf
        similarity_matrix[row,neighbors[row]] = (-2 * synonym + 1) * np.inf # if synor
        # get the most similar words that are not neighbors
        indexSorted = np.argsort(similarity_matrix[row,:].detach().numpy())
        #return the index of the most similar word that is not a neighbor
        outDict[int(indices[row])] = int(indices[indexSorted[-1 * synonym]]) # if sync

    return outDict
```

In [24]:
```
# Exemple
indices = torch.tensor([0, 1, 2, 3, 4])
neighbors = [[0, 2], [1, 4], [2, 0, 2, 3], [3, 2], [4, 1]]

similarity_matrix = torch.tensor([
    [ 1.0000, -0.4263, -0.7167, -0.9838, -0.5823],
    [-0.4263,  1.0000, -0.1600,  0.5088, -0.3708],
    [-0.7167, -0.1600,  1.0000,  0.7247,  0.5631],
    [-0.9838,  0.5088,  0.7247,  1.0000,  0.4394],
    [-0.5823, -0.3708,  0.5631,  0.4394,  1.0000]
    ])

print(select_negatives(indices, similarity_matrix, neighbors, synonym=True))
```

```
{0: 1, 1: 3, 2: 4, 3: 1, 4: 2}
```

Réponse attendue

```
{0: 1, 1: 3, 2: 4, 3: 1, 4: 2}
```

3.1.4 Implémentez la fonction `run_negative_extraction` qui prépare les paires de synonymes et d'antonymes et appelle `prepare_neighbors` et `select_negatives`. (4 points)

Préparez les indices uniques des mots du batch, calculez la similarité des mots, et appelez `prepare_neighbors` et `select_negatives`.

In [25]:
```
def run_negative_extraction(model, index_pairs, synonym=True):
    """
    Extrait les exemples négatifs pour un ensemble de paires de mots.

    Args:
        index_pairs : torch.Tensor of shape (n, 2)
                      Contient les indices des mots.

        synonym     : bool, optional (default=True)
                      Indique si l'on cherche des négatifs pour les synonymes (True) c
```

```
    Returns:
    Dictionnaire mappant les indices des mots avec leurs indices de négatifs {mot_inde
    """
    # create unix_id
    unix_id = torch.unique(index_pairs)

    #create index_to_idx
    index_to_idx = {}
    for i,word_idx in enumerate(unix_id):
        index_to_idx[int(word_idx)] = i


    # Get the neighbors
    neighbors = prepare_neighbors(index_pairs,unix_id,index_to_idx)


    # Calculate the similarity matrix
    wordEmbeddings = model.embeddings.weight[unix_id,:]
    similarity_matrix = cosine_similarity(wordEmbeddings,wordEmbeddings)

    return select_negatives(unix_id, similarity_matrix, neighbors, synonym)
```

3.2 Fonctions de coût

Pour chaque paire de synonymes (x^l, x^r) *pour x left et x right* dans le lot B_S, nous trouvons un négatif (t^l, t^r) ce qui constitue le lot T_S:

- t^l est le mot dans le lot le plus proche de x^l mais qui n'est pas x^r.
- t^r est le mot dans le lot le plus proche de x^r mais qui n'est pas x^l.

De même, pour chaque paire d'antonymes (x^l, x^r) dans le lot B_A, nous trouvons un négatif (t^l, t^r) ce qui constitue le lot T_A:

- t^l est le mot dans le lot le plus éloigné de x^l mais qui n'est pas x^r.
- t^r est le mot dans le lot le plus éloigné de x^r mais qui n'est pas x^l.

Comparer un mot à son synonyme (ou antonyme) et à son négatif permet d'entraîner le modèle sur des exemples difficiles qui forcent le modèle à apprendre des représentations plus robustes.

Il y aura trois fonctions de coût :

1. **Attraction** : Attire les synonymes plus proches les uns des autres.
2. **Répulsion** : Repousse les antonymes plus loin les uns des autres.
3. **Régularisation** : Évite que les plongements ne s'éloignent trop de ceux du modèle pré-entraîné.

Les fonctions de coût sont définies comme suit, en sommant sur `i`, les paires de synonymes et d'antonymes dans les lots B_S et B_A :

1. **Attraction** :

$$S(B_S, T_S) = \sum_{i=1}^{|B_S|} \left[\max\left(0, \delta_{syn} + x_i^l t_i^l - x_i^l x_i^r\right) + \max\left(0, \delta_{syn} + x_i^r t_i^r - x_i^l x_i^r\right)\right]$$

2. **Répulsion** :

$$A(B_A, T_A) = \sum_{i=1}^{|B_A|} \left[\max\left(0, \delta_{ant} + x_i^l x_i^r - x_i^l t_i^l\right) + \max\left(0, \delta_{ant} + x_i^r x_i^l - x_i^r t_i^r\right)\right]$$

3. **Régularisation** :

$$R(B_S, B_A) = \sum_{x_i \in V(B_S \cup B_A)} \lambda_{reg} \|\hat{x}_i - x_i\|^2$$

La fonction de coût totale est la somme de ces trois termes :

$$C(B_S, T_S, B_A, T_A) = S(B_S, T_S) + A(B_A, T_A) + R(B_S, B_A)$$

δ_{syn}, δ_{ant} et λ_{reg} sont des hyperparamètres.

Avec l'exemple précédent, prenons

> - x_i^l : voiture
> - x_i^r: véhicule
> - t_i^l: plante
>
> On veut que voiture et véhicule aient un plus grand produit scalaire que voiture et plante, donc que $\delta_{syn} + x_i^l t_i^l - x_i^l x_i^r < 0$, et donc que S soit minimisé. De même pour la deuxième partie de l'équation, symmétrique, avec le 2nd élément du couple.

3.2.1 Implémentez la fonction `synonym_cost` qui calcule la fonction de coût d'attraction (sur les paires de synonymes). (5 points)

$$S(B_S, T_S) = \sum_{i=1}^{|B_S|} \left[\max\left(0, \delta_{syn} + x_i^l t_i^l - x_i^l x_i^r\right) + \max\left(0, \delta_{syn} + x_i^r t_i^r - x_i^l x_i^r\right)\right]$$

Le membre de gauche pénalise si le mot de gauche est plus éloigné de son négatif que de son synonyme. De même, le membre de droite pénalise si le mot de gauche est plus éloigné de son négatif que de son synonyme.

In [26]:
```
def synonym_cost(model, synonym_pairs, synonym_negatives):
    """
    Calcule le coût d'attraction pour les paires de synonymes.

    synonym_pairs: liste de tuples d'indices de paires de synonymes
    synonym_negatives: dictionnaire de mots avec leurs négatifs {mot_index: négatif_in

    Returns:
    torch.Tensor, coût total pour les paires de synonymes
    """
    delta_s = model.margin_plus

    #Defining left/right of synonyms and negatives
    xl = synonym_pairs[:,0]
```

```
    xr = synonym_pairs[:,1]
    tl = torch.empty((len(xl),), dtype=torch.long)
    tr = torch.empty((len(xr),), dtype=torch.long)

    #Filling values of left and right negatives using antonym negatives dict
    for i, pair in enumerate(synonym_pairs):
        tl[i] = torch.tensor(synonym_negatives[pair[0].item()])
        tr[i] = torch.tensor(synonym_negatives[pair[1].item()])

    #Calculating the dot product of every row of the embeddings
    xl_tl = (model.embeddings(xl) * model.embeddings(tl)).sum(dim=1)
    xl_xr = (model.embeddings(xl) * model.embeddings(xr)).sum(dim=1)
    xr_tr = (model.embeddings(xr) * model.embeddings(tr)).sum(dim=1)

    #Cost of every row
    matrix = torch.relu(delta_s + xl_tl - xl_xr) + torch.relu(delta_s + xr_tr - xl_xr)

    #Adding the sum of the array
    S = torch.sum(matrix)

    return S
```

3.2.2 Implémentez la fonction `antonym_cost` qui calcule la fonction de coût de répulsion (sur les paires d'antonymes). (5 points)

$$A(B_A, T_A) = \sum_{i=1}^{|B_A|} \left[\max\left(0, \delta_{ant} + x_i^l x_i^r - x_i^l t_i^l\right) + \max\left(0, \delta_{ant} + x_i^r x_i^l - x_i^r t_i^r\right)\right]$$

Le membre de gauche pénalise si le mot de gauche est plus éloigné de son antonyme que de son négatif. De même, le membre de droite pénalise si le mot de gauche est plus éloigné de son antonyme que de son négatif.

In [27]:
```
def antonym_cost(model, antonym_pairs, antonym_negatives):
    """
    Calcule le coût de répulsion pour les paires d'antonymes.

    antonym_pairs: liste de tuples d'indices de paires d'antonymes
    antonym_negatives: dictionnaire de mots avec leurs négatifs {mot_index: négatif_in

    Returns:
    torch.Tensor, coût total pour les paires d'antonymes
    """
    delta_a = model.margin_minus

    #Defining left/right of antonyms and negatives
    xl = antonym_pairs[:,0]
    xr = antonym_pairs[:,1]
    tl = torch.empty((len(xl),), dtype=torch.long)
    tr = torch.empty((len(xr),), dtype=torch.long)

    #Filling values of left and right negatives using antonym negatives dict
    for i, pair in enumerate(antonym_pairs):
        tl[i] = torch.tensor(antonym_negatives[pair[0].item()])
        tr[i] = torch.tensor(antonym_negatives[pair[1].item()])

    #Calculating the dot product of every rows of the embeddings
    xl_xr = (model.embeddings(xl) * model.embeddings(xr)).sum(dim=1)
```

```
    xl_tl = (model.embeddings(xl) * model.embeddings(tl)).sum(dim=1)
    xr_xl = (model.embeddings(xr) * model.embeddings(xl)).sum(dim=1)
    xr_tr = (model.embeddings(xr) * model.embeddings(tr)).sum(dim=1)

    #Cost of every row
    matrix = torch.relu(delta_a + xl_xr - xl_tl) + torch.relu(delta_a + xr_xl - xr_tr)

    #Adding the sum of the array
    A = torch.sum(matrix)

    return A
```

3.2.3 Implémentez la fonction `regularization_cost` qui calcule la fonction de coût de régularisation. (4 points)

$$R(B_S, B_A) = \sum_{x_i \in V(B_S \cup B_A)} \lambda_{reg} \|\hat{x}_i - x_i\|^2$$

In [28]:
```
def regularization_cost(model, synonym_pairs, antonym_pairs):
    """
    Calcule le coût de régularisation pour les paires de synonymes et antonymes.

    synonym_pairs: liste de tuples d'indices de paires de synonymes
    antonym_pairs: liste de tuples d'indices de paires d'antonymes

    Returns:
    torch.Tensor, coût total de régularisation
    """

    #Joining synonym and antonym pairs into one tensor
    indexes = torch.unique(torch.cat((synonym_pairs, antonym_pairs)))

    xhat = model.original_embeddings(indexes)
    x = model.embeddings(indexes)

    #Calculating norm of every row
    squared_norms = torch.linalg.norm(xhat-x, ord=2,  dim=1)

    #Reg cost
    R = model.regularization * torch.sum(squared_norms)

    return R
```

3.3 Mise en place

3.3.1 Implémentez la fonction `forward` qui utilise les fonctions définies plus tôt pour calculer le coût total. (4 points)

La fonction prend en entrée un lot de synonymes et un lot d'antonymes, c'est-à-dire des paires de synonymes et des paires d'antonymes.

Vous devez trouver les négatifs de tous les mots des lots au moyen de votre fonction run_negative_extraction puis calculer la fonction de coût totale.

In [29]:
```
def forward(model, synonym_pairs, antonym_pairs):
    """
```

```
    Fonction forward pour calculer le coût total.

    Args :
        synonym_pairs   :
        Liste de tuples d'indices de paires de synonymes

        antonym_pairs   :
        Liste de tuples d'indices de paires d'antonymes

    Returns:
    Tenseur contenant le coût total (attraction, répulsion et régularisation)
    """

    synonym_negatives = run_negative_extraction(model,synonym_pairs,synonym = True)

    # We can now evaluate S
    S = synonym_cost(model,synonym_pairs,synonym_negatives)

    antonym_negatives = run_negative_extraction(model,antonym_pairs,synonym = False)

    # We can now evaluate A
    A = antonym_cost(model,antonym_pairs,antonym_negatives)

    # We evaluate the regularization cost
    R = regularization_cost(model, synonym_pairs, antonym_pairs)

    return S + A + R
```

3.3.2 Évaluation (3 points)

Utilisez la fonction `spearman_rank_correlation` pour compléter la fonction d'évaluation `evaluate` qui exécute le modèle sur le jeu d'évaluation et calcule la corrélation de Spearman entre les scores prédits et réels.

Utilisez `torch.no_grad()` pour éviter de stocker les gradients.

In [30]:
```
def evaluate(model, eval_data, word_to_index):
    """
    Calcule les prédictions du modèle sur le jeu d'évaluation puis la corrélation de S

    model: modèle de plongements de mots
    eval_data: pd.DataFrame
    word_to_index: dict

    Returns:
    float, la corrélation de Spearman entre les scores prédits par le modèle et réels.
    """

    model.eval()
    device = model.device
    word_pairs = list(zip(eval_data['word 1'], eval_data['word 2']))
    eval_indices = [(word_to_index[w1], word_to_index[w2]) for w1, w2 in word_pairs if

    # Estimate the similarity of the words in eval_data
    score = []
    for pairOfIndexes in eval_indices:
        score.append(float(cosine_similarity(
            torch.reshape(model.embeddings.weight[pairOfIndexes[0],:],(1,-1)),
```

```
            torch.reshape(model.embeddings.weight[pairOfIndexes[1],:],(1,-1))
        )))

    # Estimate the ranks of words in eval_data
    rank = getRank(np.array(score))
    with torch.no_grad():
        sCorrelation = spearman_rank_correlation(
            rank,
            evaluation_simlex['rank']
        )
    return sCorrelation
```

4. Entraînement de zéro (16 Points)

Nous allons maintenant entraîner le modèle de zéro, sans utiliser les plongements GloVe pré-entraînés. Ensuite, dans la partie 5, nous entraînerons le modèle en l'initialisant avec les plongements GloVe pré-entraînés.

In [31]:
```
# Hyperparamètres, optimiseur et DataLoader
BATCH_SIZE = 64
NUM_EPOCHS = 20
LEARNING_RATE = 5e-3

device = 'cuda' if torch.cuda.is_available() else 'cpu'

torch.cuda.empty_cache()
gc.collect()

train_syn_tensor = data_to_tensor(train_synonyms, glove_word_to_index)
train_ant_tensor = data_to_tensor(train_antonyms, glove_word_to_index)

syn_data_loader = DataLoader(train_syn_tensor, batch_size=BATCH_SIZE, shuffle=True, dr
ant_data_loader = DataLoader(train_ant_tensor, batch_size=BATCH_SIZE, shuffle=True, dr
```

4.1 Complétez la cellule suivante pour créer le modèle de zéro `model_zero`, à partir d'une matrice de plongements aléatoire. (1 point)

In [32]:
```
random_seed = 0
random_generator = torch.Generator(device=device).manual_seed(random_seed)

embeddings_size = glove_embeddings.size()
random_init_embeddings = torch.randn(embeddings_size, generator=random_generator, devi

model_zero = ContrastiveWordEmbeddingModel(random_init_embeddings, device=device)

optimizer = optim.Adam(model_zero.parameters(), lr=LEARNING_RATE)
```

4.2 Entraînez le modèle sur le jeu des paires de synonymes et d'antonymes. (6 points)

N'oubliez pas que l'entraînement se fait sur les synonymes et antonymes et que l'évaluation se fait sur SimLex-999.

À défaut d'avoir un jeu de validation, on observe les résultats sur le corpus de test : SimLex-999. Ceci n'est fait qu'à titre illustratif pour voir l'évolution de l'apprentissage. Il ne faut pas faire de choix pour l'entrainement à partir des résultats sur le corpus de test.

Note : Les jeux de synonymes et d'antonymes n'ont pas la même taille. Une époque (epoch) correspond à une itération sur le jeu de données le plus petit.

In [33]:
```
from tqdm import tqdm
spearman_corr = evaluate(model_zero, evaluation_simlex, glove_word_to_index)
print(f'Before training, Spearman Correlation: {spearman_corr:.4f}')

lossesModel0 = []
spearman_corrListModel0 = []
for epoch in tqdm(range(NUM_EPOCHS), desc="epoch"):

    #Set model to training mode
    model_zero.train()

    #Initializing loss for a single epoch
    temp_losses = 0

    #Training loop for an epoch by joing synonyms and antonyms dataloaders
    for syn, ant in zip(syn_data_loader, ant_data_loader):
        loss = forward(model_zero, syn, ant)

        optimizer.zero_grad()
        loss.backward()

        temp_losses += loss

        optimizer.step()

    #Appending mean loss and spearman correlation
    lossesModel0.append((temp_losses/(len(syn_data_loader)+len(ant_data_loader))).item
    spearman_corrListModel0 += [evaluate(model_zero,evaluation_simlex,glove_word_to_ir


spearman_corr = evaluate(model_zero, evaluation_simlex, glove_word_to_index)
print(f'After training, Spearman Correlation: {spearman_corr:.4f}')
```

```
Before training, Spearman Correlation: -0.0272
epoch: 100%|██████████| 20/20 [11:48<00:00, 35.41s/it]
After training, Spearman Correlation: 0.1797
```

4.3 Courbes d'entraînement du modèle de zéro (4 points)

Affichez la perte moyenne sur le jeu d'entraînement et la corrélation de Spearman sur le jeu de validation à chaque époque.

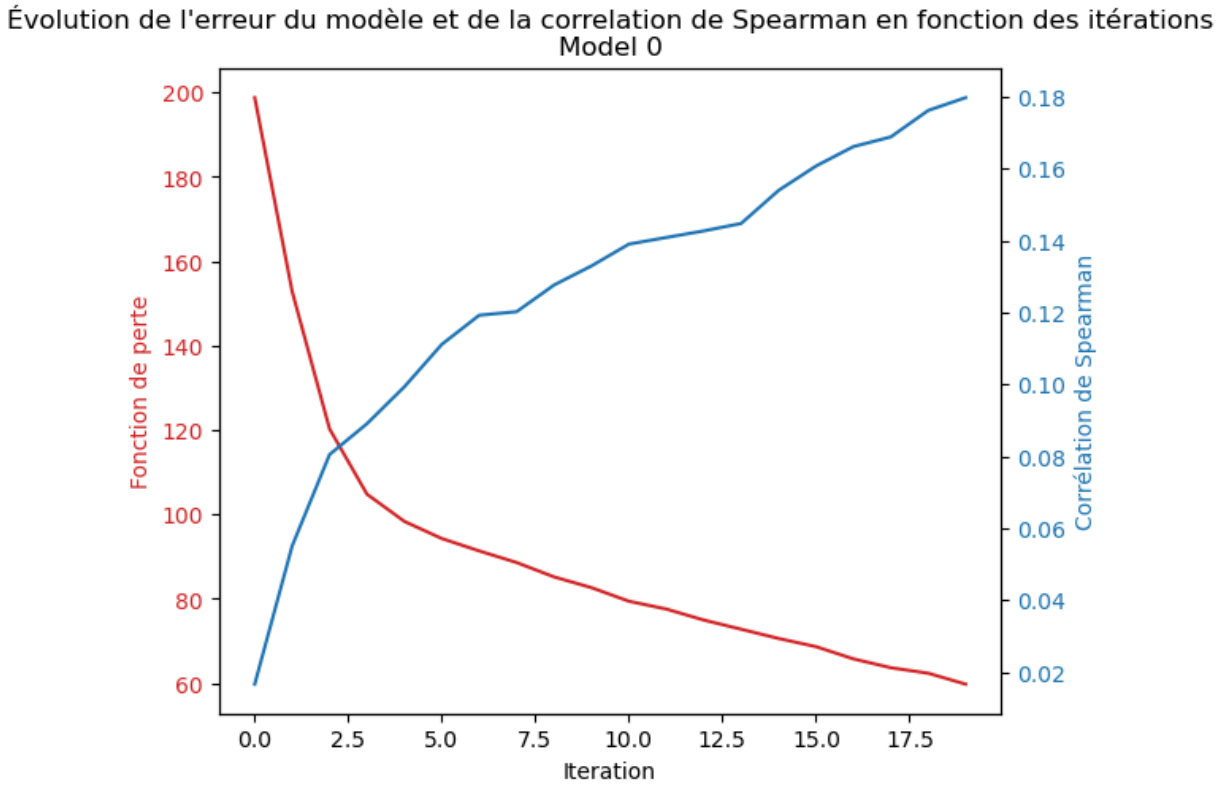
In [34]:
```
fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.plot(lossesModel0, color=color)
ax1.set_xlabel("Iteration")
ax1.set_ylabel("Fonction de perte", color = color)
ax1.tick_params(axis='y', labelcolor=color)
```

```
ax2 = ax1.twinx()  # instantiate a second Axes that shares the same x-axis

color = 'tab:blue'
ax2.plot(spearman_corrListModel0, color=color)
ax2.set_ylabel("Corrélation de Spearman",color = color)
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.title("Évolution de l'erreur du modèle et de la correlation de Spearman en fonctic
plt.show()
```



4.4 Évaluation du modèle de zéro et comparaison avec GloVe (5 points)

Comparez le modèle de zéro après l'entraînement à GloVe (résultat de la partie 2.4) en termes de corrélation de Spearman sur le jeu de validation. Quelle méthode est la plus performante ? Pourquoi ?

En comparant le modèle de zéro à celui de GloVe, il est possible de conclure que GloVe est la méthode la plus performante avec une corrélation de Spearman d'une valeur de 0.29 comparé à 0.17 pour le modèle de zéro.

- Modèle de zéro : Une des raisons, est le fait d'avoir utiliser au départ une matrice générée par des plongements aléatoires ainsi le modèle de zéro n'effectue pas de pré-entrainement. De plus, ce modèle a seulement été formé à partir de paires de synonymes et d'antonymes ce qui limite le nombre de relations sémantiques que le modèle couvre alors que le vocabulaire est beaucoup plus large.
- Glove : Ce modèle est plus performant, car il est pré-entrainé sur de grands corpus tout en capturant des relations sémantiques contextuelles à partir de cooccurrences de mots dans

des millions ou milliards de phrases. Ce large contexte permet une meilleure performance comparé au jeu de données limité du modèle de zéro. Autrement, les embeddings de Glove sont de meilleure qualité, car ils sont formés à partir de cooccurences de mots et de statistiques globales du corpus.

Ainsi, bien que le modèle de zéro a démontré une amélioration importante après l'entrainement, le pré-entrainement et la qualité des relations sémantiques formant le modèle Glove font en sorte que ce modèle est plus performant.

5. Intérêt de GloVe (18 Points)

Dans la section précédente, nous avons entraîné un modèle de zéro.

Nous allons maintenant évaluer si initialiser le modèle avec les plongements de GloVe permet d'améliorer les performances.

5.1 Initialisation avec GloVe (6 points)

Entraînez le modèle `model_fine_tuned`, mais cette fois en ititialisant directement avec les plongements du modèle pré-entraîné GloVe.

On utilisera Adam comme optimiseur.

In [35]:
```
model_fine_tuned = ContrastiveWordEmbeddingModel(glove_embeddings,device=device)
optimizer = optim.Adam(model_fine_tuned.parameters(), lr=LEARNING_RATE)
```

In [36]:
```
spearman_corr = evaluate(model_fine_tuned, evaluation_simlex, glove_word_to_index)
print(f'Before training, Spearman Correlation: {spearman_corr:.4f}')

lossesFineTuned = []
spearman_corrListFineTuned = []
for epoch in tqdm(range(NUM_EPOCHS), desc="epoch"):

    #Set model to training mode
    model_fine_tuned.train()

    #loss for a single epoch
    temp_losses = 0

    #Training loop for an epoch by joing synonyms and antonyms dataloaders
    for syn, ant in zip(syn_data_loader, ant_data_loader):

        loss = forward(model_fine_tuned, syn, ant)

        optimizer.zero_grad()

        loss.backward()
        temp_losses += loss

        optimizer.step()

    #Appending mean loss and spearman correlation
    lossesFineTuned.append((temp_losses/(len(syn_data_loader)+len(ant_data_loader))).i
```

```
    spearman_corrListFineTuned += [evaluate(model_fine_tuned,evaluation_simlex,glove_w


spearman_corr = evaluate(model_fine_tuned, evaluation_simlex, glove_word_to_index)
print(f'After training, Spearman Correlation: {spearman_corr:.4f}')
```

```
Before training, Spearman Correlation: 0.2923
epoch: 100%|██████████| 20/20 [11:50<00:00, 35.54s/it]
After training, Spearman Correlation: 0.4727
```

5.2 Courbes d'entraînement (4 points)

Affichez la perte moyenne sur le jeu d'entraînement et la corrélation de Spearman sur le jeu de validation à chaque époque.

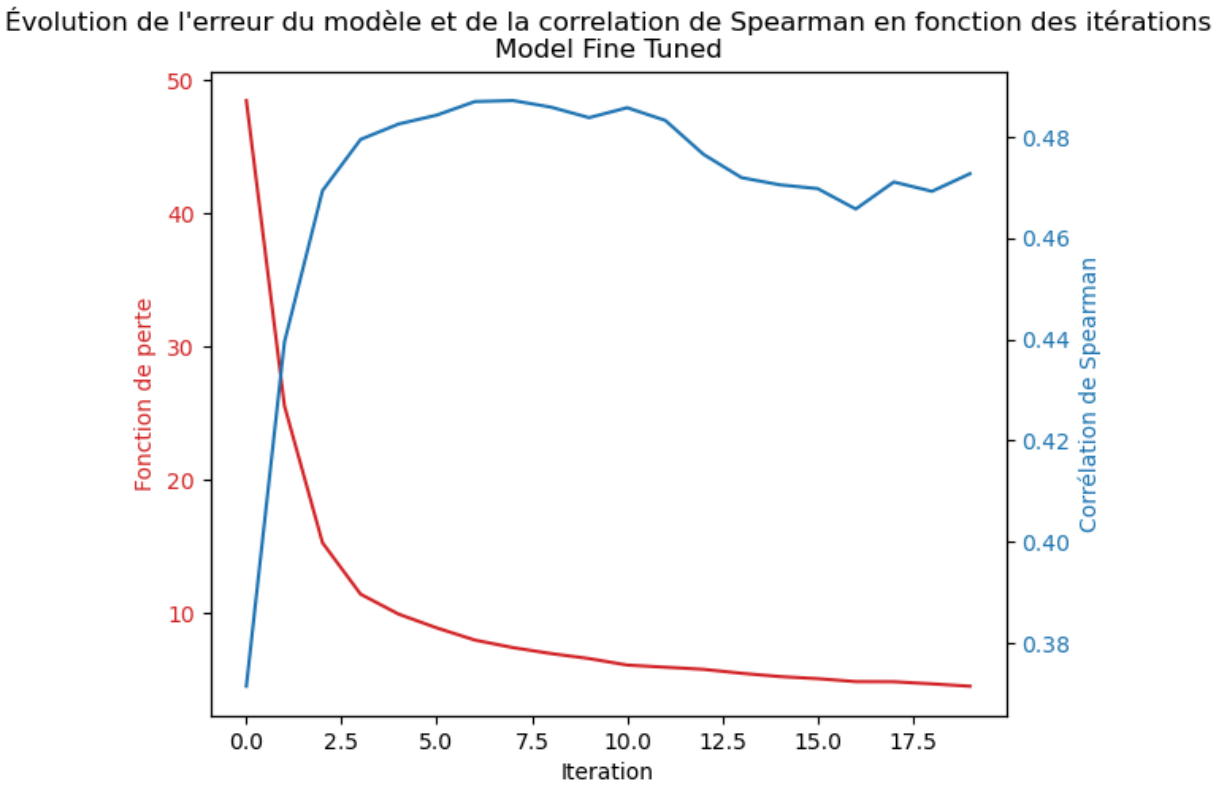
In [37]:
```
fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.plot(lossesFineTuned, color=color)
ax1.set_xlabel("Iteration")
ax1.set_ylabel("Fonction de perte", color = color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second Axes that shares the same x-axis

color = 'tab:blue'
ax2.plot(spearman_corrListFineTuned, color=color)
ax2.set_ylabel("Corrélation de Spearman",color = color)
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.title("Évolution de l'erreur du modèle et de la correlation de Spearman en fonctic
plt.show()
```



5.3 Vérification sur un exemple (3 points)

Avec le modèle `model_fine_tuned`, calculez la similarité cosinus entre 'fast' et 'slow' et entre 'fast' et 'rapid'. Commentez les résultats en les comparant avec ceux de la partie 1.4.

In [38]:
```
fastEmbedding = torch.reshape(model_fine_tuned.embeddings.weight[word_to_index['fast']
slowEmbedding = torch.reshape(model_fine_tuned.embeddings.weight[word_to_index['slow']
rapidEmbedding = torch.reshape(model_fine_tuned.embeddings.weight[word_to_index['rapid

fastSlowRapidSimilarity = cosine_similarity(fastEmbedding, torch.cat((slowEmbedding,ra

print(f'{"":^10} |{"slow":^10} |{"rapid":^10}')
fastSlowStr = f'{float(fastSlowRapidSimilarity[0,0]):.5f}'
fastRapidStr = f'{float(fastSlowRapidSimilarity[0,1]):.5f}'
print(f'{"fast":^10} |{fastSlowStr:^10} |{fastRapidStr:^10}')
```

```
           |   slow     |  rapid
   fast    | 0.14882    | 0.41810
```

5.4 Analyse, comparaison, conclusion (5 points)

Comparez les performances des trois modèles (GloVe, zéro, fine-tuned). Quelle méthode est la plus performante ? Pourquoi ?

Le meilleur modèle correspond au fine-tuned puisqu'on part d'un plongement qui est déja cohérent sur la similarité des synonymes et est entraîné pour capturer des relations sémantiques contextuelles. L'entrainement permet de régler l'incohérence d'une haute similarité pour une paire d'antonymes. Le modèle zéro est le moins performant sachant qu'il part d'un plongement aléatoire, l'entrainement réalisé n'est pas suffisant pour proposer des performances correctes. Au mieu on peut espérer que le modèle 0 puisse faire la différence entre des synonymes et des

antonymes puisque c'est ce à quoi il est entraîné, cependant il ne pourra pas capturer de relations sémantiques contextuelles puisqu'il n'y est pas entraîné.

La fonction de perte utilisé permet de discriminer les antonymes vis à vis des synonymes puisque la similarité entre les termes *fast* et *slow* est passé de 0.71 à 0.14 tandis que la similarité entre *fast* et *rapide* est passé de 0.64 à 0.41. De ce fait, on remarque que même si les similarités ont globalement diminué, les rangs de *fast*|*rapid* et *fast*|*slow* ont été inversés, donc la corrélation de Spearman s'est vu augmentée.

Notre entraînement a permis de discriminer les antonymes, cette tâche est réussie. Cependant, on peut voir que la similarité *fast*/*rapid* de 0.41 est relativement basse. Il serait intéressant de vérifier les performances de notre modèle dans d'autres applications pour s'assurer que le plongement reste utilisable.

Livrables

Vous devez remettre votre notebook sur Moodle et Gradescope en ipynb et pdf. Pour Gradescope vous devez associer les numéros de questions avec vos réponses dans le pdf grâce à l'outil que fournit Gradescope.

Évaluation

Votre TP sera évalué selon les critères suivants :

1. Exécution correcte du code et obtention des sorties attendues
2. Réponses correctes aux questions d'analyse
3. Qualité du code (noms significatifs, structure, performance, gestion d’exception, etc.)
4. Commentaires clairs et informatifs