Embeddings de mots ou de phrases

Cours « Recherche d'Information et Graphes de Données »

Nicolas Delestre

INSA
ROUEN NORMANDIE

Aides de mes amis imaginaires

- Ce cours a été rédigé en LaTeX sous Emacs avec le mode Copilot
- J'ai très souvent échangé avec ChatGPT (version 5.2) et Gemini (version 3) pour présenter les concepts de ce cours plus clairement

INSA ROUEN NORMANDIE

Plan

1. Rappel
2. Embeddings statiques de mots
3. Embedding dynamiques de mots
4. Embedding de phrases
5. Conclusion

INSA ROUEN NORMANDIE

Embedding de documents (1/2)

TF-IDF, LSA

- **Sac de mots** : on ignore l'ordre, on compte les occurrences
- **TF-IDF** : pondération qui favorise les termes *discriminants* :

$$\text{tfidf}(t, d) = \text{tf}(t, d) \cdot \log\left(\frac{N}{\text{df}(t)}\right)$$

- **LSA** : réduction de dimension via SVD sur la matrice terme–document (projection dans un espace latent de dimension $k \ll |V|$)

INSA ROUEN NORMANDIE

Embedding de documents (2/2)

Avantages, inconvénients

- ✓ Simple, interprétable (TF-IDF), efficace pour la recherche d'information classique
- ✓ Fonctionne sans apprentissage supervisé
- ✗ Très grande dimension (TF-IDF), *sparses*
- ✗ Peu de sémantique fine : synonymes / paraphrases mal capturés
- ✗ Pas de notion de contexte (polysémie, ordre des mots)

INSA ROUEN NORMANDIE

Vecteurs *one-hot*

Principe

- Vocabulaire V de taille $|V|$
- Un mot w_i est représenté par un vecteur $x \in \{0,1\}^{|V|}$ tel que $x_i = 1$ et $x_j = 0$ pour $j \neq i$
- Deux mots distincts sont *orthogonaux* : $x_i^\top x_j = 0$

Avantage, inconvénient

✓ Représentation exacte (pas de collisions), très simple

✗ Dimension énorme ($|V|$), mémoire/temps

✗ Aucune notion de similarité sémantique (tout est à distance égale)

✗ Généralisation difficile : mots rares / nouveaux mots

Word2Vec (2013) 1 / 4

Principe

- Apprentissage *non supervisé* d'un vecteur dense $v_m \in \mathbb{R}^d$ par mot
- Idée : deux mots partageant les même contexte doivent avoir des embeddings similaires
  - Contexte : mots dans une fenêtre de taille k autour du mot cible
  - Mesure de similarité courante : **cosinus**

$$\cos(u, v) = \frac{u^\top v}{\|u\|\|v\|}$$

- Deux réseaux de neurones simples avec une seule couche cachée (taille entre 100 et 300) :
  1. CBOW : prédire le mot cible à partir du contexte
  2. Skip-Gram : prédire le contexte à partir du mot cible
- Afin de réduire le vocabulaire, le corpus d'apprentissage est nettoyé (élimination des mots vides, racinisation ou lemmatisation, regroupement de mots formant un seul mot)
- Les mots sont présentés aux réseaux sous forme de vecteurs *one-hot*

Word2Vec (2013) 2 / 4

CBOW

- Entrée : contexte $C = \{m_{i-k}, \ldots, m_{i-1}, m_{i+1}, \ldots, m_{i+k}\}$
- Sortie : mot cible m_i
- Architecture :
  - Couche d'entrée : vecteurs *one-hot* des mots du contexte
  - Couche cachée : moyenne des embeddings des mots du contexte
  - Couche de sortie : softmax sur le vocabulaire pour prédire m_i
- Fonction de perte : cross-entropy entre la prédiction et le mot cible

$$L(y, \hat{y}) = -\sum_{j=1}^{|V|} y_j \log(\hat{y}_j)$$

Word2Vec (2013) 3 / 4

Skip-Gram

- Entrée : mot cible m_i
- Sortie : contexte $C = \{m_{i-k}, \ldots, m_{i-1}, m_{i+1}, \ldots, m_{i+k}\}$
- Architecture :
  - Couche d'entrée : vecteur *one-hot* du mot cible
  - Couche cachée : embedding du mot cible
  - Couche de sortie : softmax sur le vocabulaire pour prédire les mots du contexte
- Fonction de perte : somme des cross-entropies entre les prédictions et les mots du contexte

$$L(y, \hat{y}) = -\sum_{j=1}^{|C|} \sum_{l=1}^{|V|} y_{j,l} \log(\hat{y}_{j,l})$$

$$v_{roi} - v_{homme} + v_{femme} \approx v_{reine}, v_{France} - v_{Paris} + v_{Berlin} \approx v_{Allemagne}$$

Word2Vec (2013) 4 / 4

Exemple en python

```python
from gensim.models import KeyedVectors
import numpy as np

print("Chargement du modèle Word2Vec (French
      CoNLL17 corpus)...")
model = KeyedVectors.load_word2vec_format(
    "model.bin",
    binary=True
)

print("Vocabulaire :", len(model.key_to_index))
print("Dimension :", model.vector_size)

print("roi - homme + femme ≈ ?")
for w, s in model.most_similar(positive=["roi", "
     femme"],
                               negative=["homme"],
                               topn=5
                               ):
    print(f"{w:10s} {s:.3f}")
```

```python
print("paris - france + allemagne ≈ ?")

v_paris = model["paris"]
v_france = model["france"]
v_allemagne = model["allemagne"]

combo = v_paris - v_france + v_allemagne

for w, s in model.most_similar([combo], topn=5):
    print(f"{w:10s} {s:.3f}")
```

D'autres systèmes d'embeddings statiques

GloVe (2014)

- Fondé sur la matrice de co-occurrence globale des mots dans le corpus d'apprentissage :
  - X_{ij} = nombre de fois que les mots i et j co-occurrent dans une fenêtre de contexte
  - Les embeddings w doivent satisfaire $w_i^T w_j \approx log(X_{ij})$

FastText (2016)

- Extension de word2vec qui utilise des sous-mots (n-grammes de caractères) pour mieux gérer les mots rares ou inconnus

INSA ROUEN NORMANDIE

En résumé

Avantages, inconvénients

- ✓ Vecteurs denses de faible dimension (100-300)
- ✓ Petits réseaux de neurones (une seule couche cachée)
- ✓ Capturent des similarités sémantiques entre mots
- ✗ Embeddings fixes, pas de prise en compte du contexte pour représenter un ou plusieurs mots successifs :
  - « **Il a ouvert** la porte » vs « **Il a ouvert** un compte bancaire »
  - « Il a ouvert **la porte** » vs « Il **la porte** »
  - « La **petite reine** observait la cour avec gravité, bien consciente du rôle important qui l'attendait malgré son jeune âge » vs « Depuis l'enfance, elle rêvait de parcourir le monde sur sa **petite reine**, libre au gré des routes et des paysages »

Mécanisme d'attention « All you need is attention » (2017) 1 / 8

Principe d'une tête d'attention :

- Ajouter à l'embedding statique e_m d'un mot m :
  - un vecteur de position e_p

$$e_{m,p} = e_m + e_p$$

  - un vecteur contextuel e_c qui dépend des autres mots du contexte

$$e_{m,p,c} = e_{m,p} + e_c$$

- Ainsi un même mot pourra avoir un embedding différent en fonction du contexte

Comment calculer e_c ?

- En utilisant les autres embeddings e_{m_i,p_i} des mots m_i du contexte
- En pondérant l'influence de chaque mot m_i sur le mot m grâce à un schéma d'attention S

Mécanisme d'attention « All you need is attention » (2017) 2 / 8

Définitions

Soit une phrase composée de mot m_i a des positions p_i. On définit :

- La requête (*Query (Q)*) : vecteur $q_{e_{m,p}}$ (dans un espace de taille réduite d_a) qui permet de questionner les autres embeddings : "qui peut avoir une influence sur moi ?"

$$q_{e_{m,p}} = W_Q e_{m,p}$$

- La clé (*Key (K)*) : vecteur k_i (dans le même espace de taille d_a) qui permet de calculer l'importance de l'influence d'un mot m_i :

$$k_i = W_K e_{m,i}$$

- Le schéma d'attention S d'un contexte est une matrice carrée (taille = nombre de mots) dont les valeurs sont les produits scalaires des $k_i q_j$ transformés en distribution de probabilité au niveau des colonnes, grâce à l'utilisation softmax sur chacune des colonnes $\Rightarrow S_{ii}$ représente l'influence du mot i sur le mot j

Mécanisme d'attention « All you need is attention » (2017) 3 / 8

Définitions (suite)

- La valeur (*Value (V)*) : vecteur v_i (toujours dans l'espace de taille d_a) qui indique quelle modification appliquer en fonction l'influence du mot i :

$$v_i = W_v e_{m,i}$$

- L'embedding d'attention e_a d'un mot m à la position p est la somme des vecteurs v_j pondérés par leurs importances calculées dans le schéma d'attention S :

$$e_a = \sum_i S_{ip} v_i$$

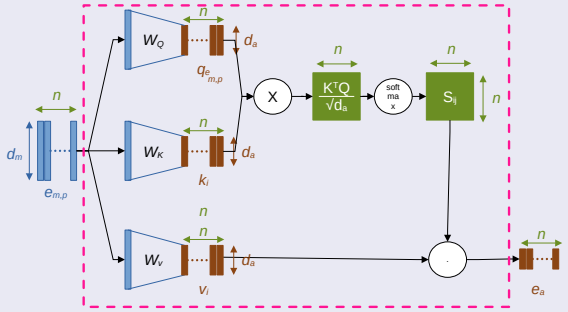
Notation dans l'article

$$Attention(Q, K, V) = softmax(\frac{K^T Q}{\sqrt{d_k}})V$$

Mécanisme d'attention « All you need is attention » (2017) 4 / 8

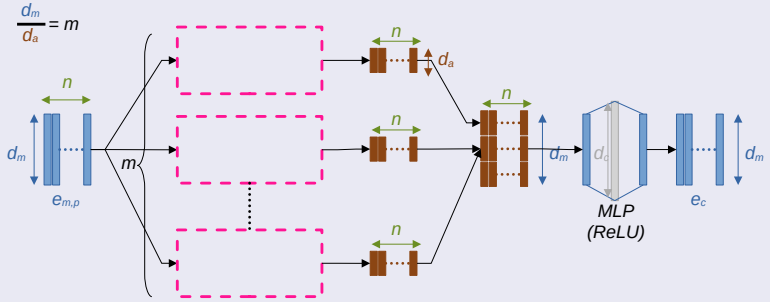
Une tête d'attention = réseaux de neurones interconnectés

- On note :
  - d_m la taille de l'espace des embeddings de mots e_m, vecteurs positionnels e_p des embeddings positionnels $e_{m,p}$
  - d_a la taille de l'espace d'attention , taille des vecteurs $q_{e_{m,p}}$, k_i et v_i



Mécanisme d'attention « All you need is attention » (2017) 5 / 8

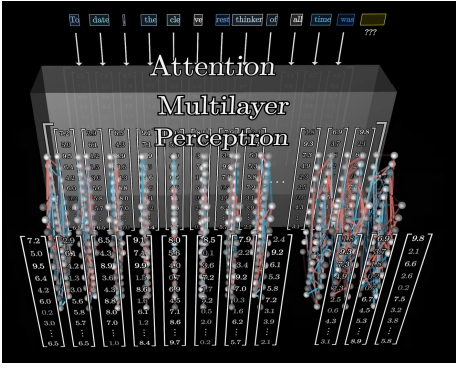
Une couche d'attention = m multi-têtes d'attention + MLP



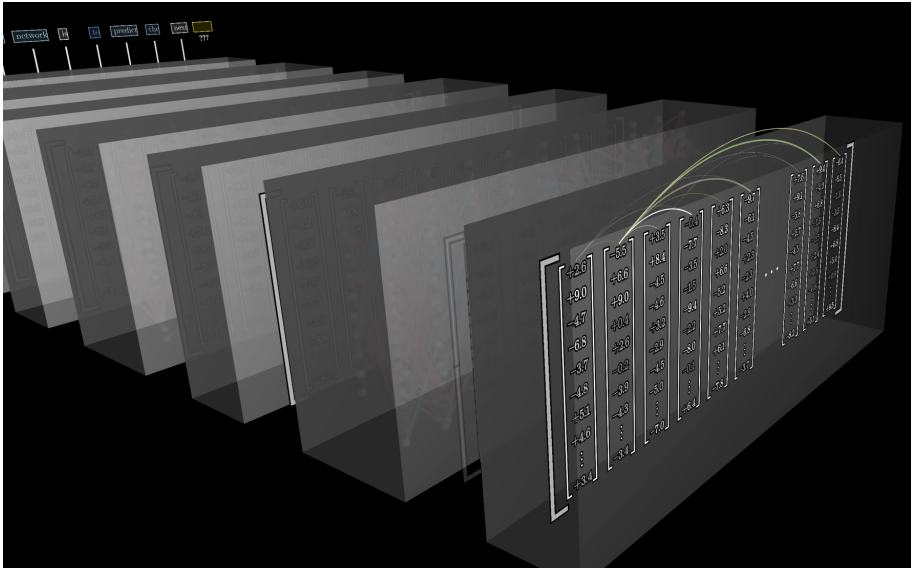
Un transformeur (encodeur)

- Empilement de L couches d'attention

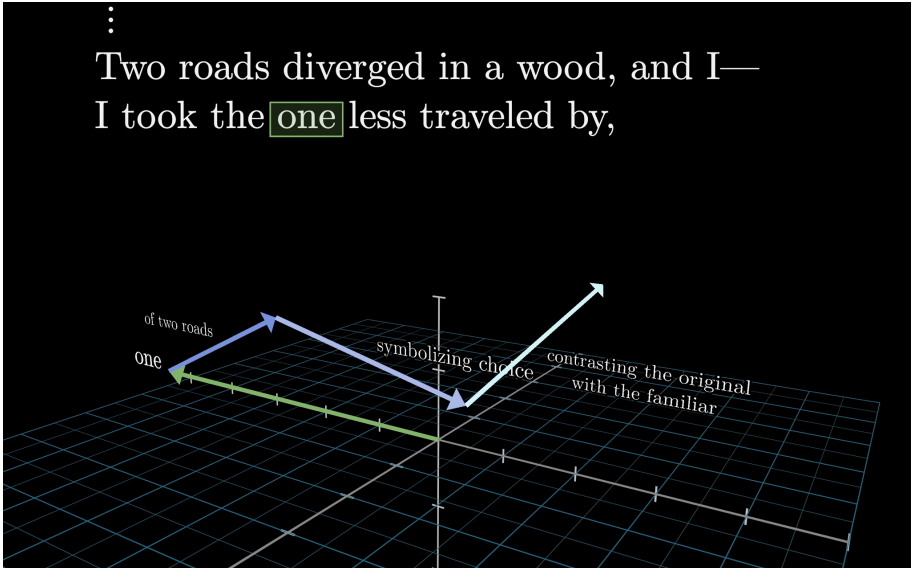
Mécanisme d'attention « All you need is attention » (2017) 6 / 8



Mécanisme d'attention « All you need is attention » (2017) 7 / 8



Mécanisme d'attention « All you need is attention » (2017) 8 / 8



BERT (2018) 1 / 2

Un système d'embedding dynamique

- Ne travaille pas sur des mots mais sur des tokens
- Architectures :
  - Phase d'inférences : Tokens → Embeddings statiques → Têtes d'attention → Embeddings dynamiques
  - Phase d'apprentissage : Tokens → Embeddings statiques → Têtes d'attention → Embeddings dynamique → Tête MLM (*Masked Language Modeling*) → *Loss* :
    - Le principe est de montrer des suites de tokens (issus des phrases) en masquant un token et de demander au modèle de le prédire
    - Les têtes d'attention calculent des embedding dynamiques $e_{m,p,c}$ des tokens
    - La tête MLM calcule la probabilité de chaque token du vocabulaire d'être le token masqué à partir de $e_{m,p,c}$: $p(token|e_{m,p,c}) = softmax(W_{vocab}e_{m,p,c} + b)$
    - La *loss* vaut : $L = -\log(p(token_{masqué}|e_{m,p,c}))$

ROUEN NORMANDIE

BERT (2018) 2 / 2

Paramètres des deux modèles initiaux

Caractéristique	BERT-base	BERT-large
Nombre de couches (L)	12	24
Dimension des embeddings (d_m)	768	1024
Nombre de têtes d'attention	12	16
Dimension par tête (d_a)	64	64
Dimension des MLP (d_c)	3072	4096
Nombre de paramètres	$\sim$110 M	$\sim$340 M
Longueur maximale de séquence	512	512
Vocabulaire	$\sim$30 000	$\sim$30 000

Table – Caractéristiques principales de BERT-base et BERT-large

Des modèles pour chaque langue

Paramètres des deux modèles initiaux

Caractéristique	BERT-base	CamemBERT	FlauBERT
Langue	Anglais	Français	Français
Architecture	BERT-base	BERT-base	BERT-base
Nombre de couches (L)	12	12	12
Dimension des embeddings	768	768	768
Têtes d'attention	12	12	12
Dimension par tête	64	64	64
Dimension des MLP	3072	3072	3072
Vocabulaire	WordPiece	SentencePiece BPE	SentencePiece BPE
Taille du vocabulaire	∼30 000	∼32 000	∼50 000
Corpus d'entraînement	Books+Wiki (EN)	OSCAR (FR)	Wiki+Books+Web (FR)
Nombre de paramètres	∼110 M	∼110 M	∼138 M

Table – Comparaison entre BERT-base, CamemBERT et FlauBERT

Intuitions ayant amené à cette architecture

Les limites des RNN, LSTM, GRU

- La séquence n'a pas besoin d'être traitée pas à pas pour modéliser des dépendances longues
- Si on pouvait comparer directement tous les mots entre eux, sans passer par un état caché séquentiel...

Attention comme mécanisme principal

- Les relations linguistiques ne sont pas locales dans le temps, mais globales dans la phrase
- La position est une information, pas un mécanisme

Multi-têtes par couche et concaténation

- Chaque tête « voit » la séquence différemment
- Concaténation = préserver l'information

Avantage, inconvénients

- ✓ Embeddings de mots dynamiques, sensibles au contexte
- ✓ Capturent la polysémie et les relations sémantiques complexes
- ✗ Très coûteux en temps et en mémoire (nombre de paramètres, complexité de l'attention $O(n^2)$)

CamemBERT en Python 1 / 7

Bibliothèques utilisées

- torch : pour les tenseurs et les opérations de base
- transformers : pour charger le modèle CamemBERT pré-entraîné et son tokenizer

import et constante

```
2 import torch
3 import torch.nn.functional as F
4 from transformers import AutoTokenizer, AutoModel
5
6 NOM_MODELE = "camembert-base"
```

main

```
113 def main():
114     print(f"Chargement de {NOM_MODELE}...")
115     tokeniseur = AutoTokenizer.from_pretrained(NOM_MODELE, use_fast=True)
116     modele = AutoModel.from_pretrained(NOM_MODELE)
117     modele.eval()
```

CamemBERT en Python 2 / 7

main (suite et fin)

```
119     # --- Exemples de polysémie ---
120     phrase_animal = "La souris a mangé du fromage dans la cuisine."
121     phrase_animal_syn = "Le mulot a mangé du fromage dans la cuisine."
122     phrase_info = "J'ai cliqué avec la souris sans fil sur mon ordinateur."
123     phrase_info_syn = "J'ai cliqué avec le trackpad sur mon ordinateur."
124
125     print("\n--- Embeddings contextualisés (CamemBERT) ---")
126     embedding_souris_animal = embedding_mot_contextualise(phrase_animal, "souris", tokeniseur, modele)
127     embedding_mulot = embedding_mot_contextualise(phrase_animal_syn, "mulot", tokeniseur, modele)
128     embedding_souris_info = embedding_mot_contextualise(phrase_info, "souris", tokeniseur, modele)
129     embedding_trackpad = embedding_mot_contextualise(phrase_info_syn, "trackpad", tokeniseur, modele)
130
131     print("\nSimilarités cosinus (proche de 1 = très similaire)")
132     print(f"souris (animal) vs mulot    : {similarite_cosinus(embedding_souris_animal, embedding_mulot):.3f}")
133     print(f"souris (info)   vs trackpad : {similarite_cosinus(embedding_souris_info, embedding_trackpad):.3f}")
134     # Comparaisons croisées
135     print(f"souris (animal) vs trackpad : {similarite_cosinus(embedding_souris_animal, embedding_trackpad):.3f}
            ")
136     print(f"souris (info)   vs mulot    : {similarite_cosinus(embedding_souris_info, embedding_mulot):.3f}")
137     # Similarité entre les deux sens du même mot
138     print(f"souris (animal) vs souris (info): {similarite_cosinus(embedding_souris_animal,
            embedding_souris_info):.3f}")
```

CamemBERT en Python 3 / 7

embedding_mot_contextualise

```python
@torch.no_grad() # Pas de calcul de gradients nécessaire pour l'inférence
def embedding_mot_contextualise(texte: str,
                                mot: str,
                                tokeniseur,
                                modele,
                                occurrence: int = 0) -> torch.Tensor:
    """
    Calcule l'embedding contextualisé d'un mot dans un texte.
    :param texte: Le texte contenant le mot
    :param mot: Le mot cible
    :param tokeniseur: Le tokenizer à utiliser
    :param modele: Le modèle de langage à utiliser
    :param occurrence: L'occurrence du mot à utiliser (0 pour la première)
    :return: Embedding contextualisé du mot
    """
    debut, fin = debut_fin_occurrence_mot(texte, mot, occurrence)
    identifiants_tokens, offsets = tokenizer_avec_offsets(texte, tokeniseur)
    sortie = modele(input_ids=identifiants_tokens)
    etats_caches = sortie.last_hidden_state[0]
    indices = indices_sous_tokens(offsets, debut, fin)
    return embedding_moyen_tokens(etats_caches, indices)
```

CamemBERT en Python 4 / 7

embedding_moyen_tokens

```
76 def embedding_moyen_tokens(etats_caches: torch.Tensor, indices: list[int]) -> torch.Tensor:
77     """
78     Moyenne des embeddings des tokens sélectionnés
79     et normalisation.
80     :param etats_caches: Tensor de shape (seq_len, hidden_size)
81     :param indices: Liste des indices de tokens à utiliser
82     :return: Embedding moyen normalisé
83     """
84     embedding = etats_caches[indices].mean(dim=0)
85     embedding = F.normalize(embedding, dim=0)
86     return embedding
```

INSA ROUEN NORMANDIE

CamemBERT en Python 5 / 7

indices_sous_tokens

```python
def indices_sous_tokens(offsets: list[tuple[int, int]],
                        debut: int,
                        fin: int) -> list[int]:
    """
    Retourne les indices des tokens dont les offsets
    recouvrent l'intervalle [debut, fin)].
    :param offsets: Liste de couples (debut, fin) par token
    :param debut: Indice de début du mot cible
    :param fin: Indice de fin du mot cible
    :return: Liste des indices de tokens recouvrant le mot cible
    """
    indices = []

    for i, (a, b) in enumerate(offsets):
        if a == 0 and b == 0:
            continue  # tokens spéciaux
        if max(a, debut) < min(b, fin):
            indices.append(i)

    if not indices:
        raise RuntimeError(
            f"Aucun sous-token trouvé pour l'intervalle [{debut}, {fin})"
        )

    return indices
```

CamemBERT en Python 6 / 7

tokenizer_avec_offsets

```python
def tokenizer_avec_offsets(texte: str, tokeniseur) -> tuple[torch.Tensor, list[tuple[int, int]]]:
    """
    Tokenise le texte et retourne un :
    - identifiants_tokens
    - offsets (liste de couples (debut, fin) par token)
    :param texte: Le texte à tokeniser
    :param tokeniseur: Le tokenizer à utiliser
    :return: (identifiants_tokens, offsets)
    """
    encodage = tokeniseur(texte,
                          return_tensors="pt",
                          return_offsets_mapping=True,
                          add_special_tokens=True)

    identifiants_tokens = encodage["input_ids"]
    offsets = encodage["offset_mapping"][0].tolist()

    return identifiants_tokens, offsets
```

INSA ROUEN NORMANDIE

CamemBERT en Python 7 / 7

debut_fin_occurence_mot

```
29 def debut_fin_occurrence_mot(texte: str, mot: str, occurrence: int = 0) -> tuple[int, int]:
30     """
31     Retourne (debut, fin) correspondant à la n-ième occurrence de 'mot'
32     dans 'texte' (au niveau des caractères).
33     :param texte: Le texte dans lequel chercher
34     :param mot: Le mot à trouver
35     :param occurrence: L'occurrence à trouver (0 pour la première)
36     :return: Un tuple (debut, fin) des indices de caractères
37     """
38     texte_min = texte.lower()
39     mot_min = mot.lower()
40 
41     debut = -1
42     position = 0
43     for _ in range(occurrence + 1):
44         debut = texte_min.find(mot_min, position)
45         if debut == -1:
46             raise ValueError(f"Mot '{mot}' introuvable dans : {texte}")
47         position = debut + len(mot_min)
48 
49     fin = debut + len(mot_min)
50     return debut, fin
```

Sentence-BERT, SBERT (2019) 1 / 2

Principe

- BERT est conçu pour produire des embeddings de mots, pas de phrases
- SBERT utilise une architecture de type Siamese Network pour produire des embeddings de phrases à partir de BERT
- L'idée est d'entraîner un modèle à prédire si deux phrases sont similaires ou non, en utilisant une tâche de classification binaire

Architecture

- On reprend l'architecture de BERT en ajoutant à la fin une couche de pooling (par exemple, moyenne ou max pooling) pour obtenir un vecteur de taille fixe représentant la phrase

Sentence-BERT, SBERT (2019) 2 / 2

Apprentissage

- On utilise des paires de phrases p_1 et p_2 étiquetées comme similaires ou non
- On présente ces deux phrases pour obtenir leurs embeddings e_{p_1} et e_{p_2}
- La loss utilise la similarité cosinus entre les deux embeddings :

$$L = (cos(e_{p_1}, e_{p_2}) - y)^2$$

  - $y = 1$ si les phrases sont similaires, $y = 0$ sinon

INSA ROUEN NORMANDIE

Conclusion

Nous avons vu dans ce cours...

- Les embeddings de mots statiques (word2vec, GloVe, FastText) et leurs limites
- Le mécanisme d'attention et les transformeurs (BERT)
- Comment utiliser un modèle pré-entraîné pour obtenir des embeddings de mots contextualisés

INSA ROUEN NORMANDIE