

# Travailler avec des données textuelles

## Chapitre 2 : Tokenisation, BPE et Embeddings

Master - Traitement Automatique du Langage Naturel Avancé

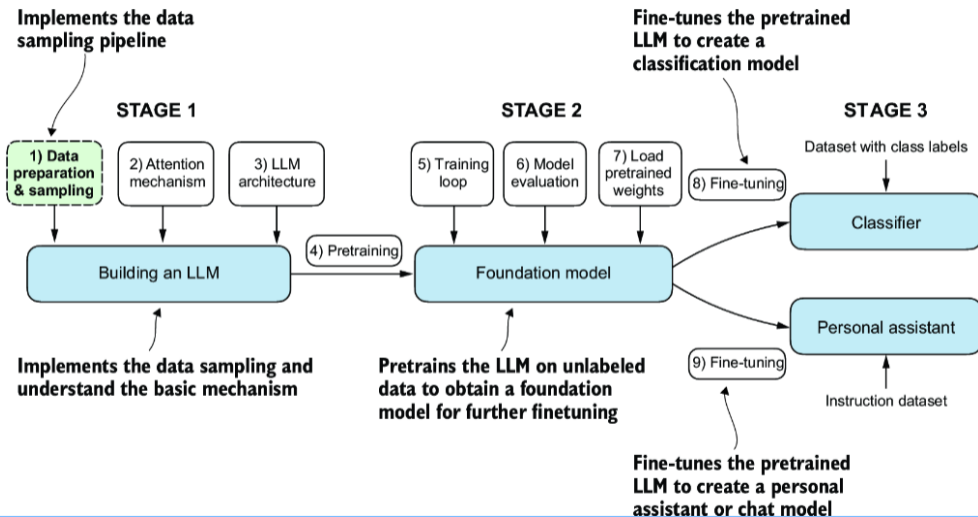
Fréjus A. A. Laleye

4 février 2026

# Plan du cours

- 1 Introduction
- 2 Comprendre les word embeddings
- 3 Tokenisation du texte
- 4 Byte Pair Encoding (BPE)
- 5 Échantillonnage et DataLoader
- 6 Création des token embeddings
- 7 Encodage des positions

# Le pipeline de traitement des LLMs



Les trois grandes étapes

# Objectifs du chapitre

À la fin de ce chapitre, vous serez capables de :

- Préparer le texte d'entrée pour l'entraînement des LLMs
- Diviser le texte en tokens de mots et de sous-mots individuels
- Comprendre le Byte Pair Encoding (BPE) comme méthode avancée de tokenisation
- Échantillonner des exemples d'entraînement avec une approche par fenêtre glissante
- Convertir les tokens en vecteurs d'embedding
- Ajouter des encodages positionnels

## Durée

Environ 3 heures avec alternance théorie/pratique

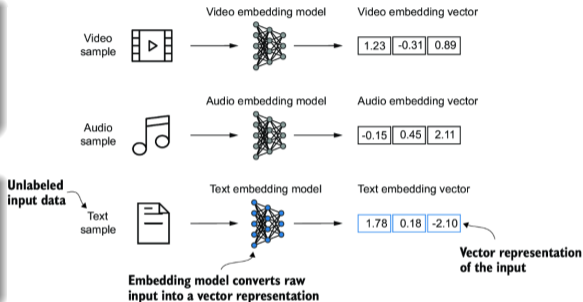
# Qu'est-ce qu'un embedding ?

## Définition

Un **embedding** est une correspondance entre des objets discrets (mots, images, documents) et des points dans un espace vectoriel continu.

## Pourquoi ?

Les réseaux de neurones ne peuvent pas traiter directement du texte catégoriel. Ils ont besoin de vecteurs de nombres réels.



## Important

Différents formats de données nécessitent des modèles d'embedding distincts (texte  $\neq$  audio  $\neq$  vidéo).

# L'approche Word2Vec

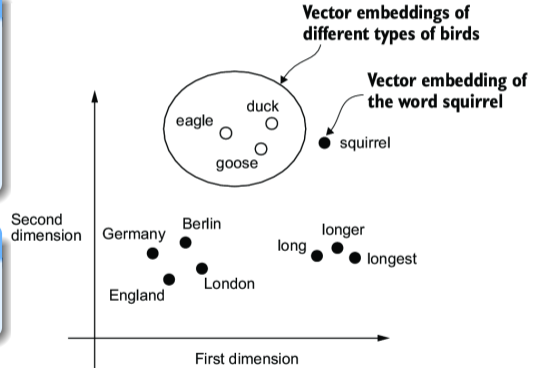
## Principe

Word2Vec entraîne un réseau de neurones pour générer des word embeddings en prédisant :

- Le contexte étant donné le mot (Skip-gram)
- Le mot étant donné le contexte (CBOW)

## Idée clé

Les mots qui apparaissent dans des contextes similaires ont tendance à avoir des significations similaires.



# Propriétés remarquables des embeddings

## Relations sémantiques

Les embeddings capturent des relations sémantiques :

- $\text{roi} - \text{homme} + \text{femme} \approx \text{reine}$
- $\text{Paris} - \text{France} + \text{Italie} \approx \text{Rome}$

## Similarité

La distance (cosinus, euclidienne) entre vecteurs reflète la similarité sémantique :

$$\text{sim}(v_1, v_2) = \frac{v_1 \cdot v_2}{\|v_1\| \|v_2\|}$$

## Exemple

chat et chien auront des vecteurs proches, tandis que chat et voiture seront éloignés.

# Embeddings dans les LLMs

## Word2Vec

- Embeddings **statiques**
- Pré-entraînés séparément
- Un seul vecteur par mot
- Dimension : 100-300

## LLMs (GPT, BERT)

- Embeddings **contextuels**
- Appris pendant l'entraînement
- Vecteur dépend du contexte
- Dimension : 768-12,288

## Exemple : "banque"

- Word2Vec : un seul vecteur pour "banque" (institution ou siège)
- LLM : vecteur différent selon le contexte
  - "Je vais à la **banque**" (institution financière)
  - "Je m'assois sur la **banque**" (siège)

# Visualisation et dimensionnalité

## Défi de la visualisation

Les embeddings de haute dimension sont impossibles à visualiser directement (notre perception est limitée à 3D).

## Tailles d'embeddings dans GPT

Modèle	Paramètres	Dimension
GPT-2 Small	117M	768
GPT-2 Medium	345M	1,024
GPT-2 Large	774M	1,280
GPT-2 XL	1.5B	1,600
GPT-3	175B	12,288

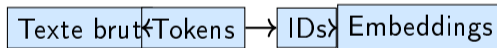
## Compromis

Plus de dimensions = plus de capacité expressive, mais aussi plus de calculs et de mémoire.

# Transition : Du texte aux embeddings

## Pipeline de préparation

- 1 **Tokenisation** : Diviser le texte en tokens
- 2 **Conversion en IDs** : Mapper chaque token à un ID unique
- 3 **Création des embeddings** : Convertir les IDs en vecteurs
- 4 **Encodages positionnels** : Ajouter l'information de position



## Focus

Nous allons maintenant plonger dans chacune de ces étapes en détail.

# Tokenisation : Première approche

## Approche naïve : Division sur les espaces

```
text = "Hello, world. Is this-- a test?"
result = text.split(" ")
# ['Hello,', 'world.', 'Is', 'this--', 'a', 'test?']
```

## Problèmes

- Ponctuation collée aux mots : "Hello," vs "Hello"
- Vocabulaire explosif : "hello", "hello,", "hello.", "hello!", ...
- Perte de généralisation

## Solution : Expressions régulières

```
import re
text = "Hello, world. Is this-- a test?"
result = re.split(r'([,.;?!"()\' ]|--|\s)', text)
result = [item.strip() for item in result if item.strip()]
# ['Hello', ',', 'world', '.', 'Is', 'this', '--', 'a', 'test', '?']
```

# Application : "The Verdict" d'Edith Wharton

## Chargement du texte

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

print("Total de caracteres:", len(raw_text))    # 20,479
print("Premiers 99 caracteres:")
print(raw_text[:99])
# I HAD always thought Jack Gisburn rather a cheap genius
# --though a good fellow enough--so it was no
```

## Tokenisation

```
preprocessed = re.split(r'([,.;?_!()"\' ]|--|\s)', raw_text)
preprocessed = [item.strip() for item in preprocessed if item.strip()]

print("Total de tokens:", len(preprocessed))    # 4,649
```

## Observations

- Ratio : 4.4 caractères par token

# Conversion tokens → IDs

## Construction du vocabulaire

```
# Obtenir tous les tokens uniques et les trier
all_words = sorted(set(preprocessed))
vocab_size = len(all_words) # 1,130

# Créer les mappings
vocab = {token: integer for integer, token in enumerate(all_words)}
inverse_vocab = {integer: token for integer, token in enumerate(all_words)}
```

## Encodage

"Hello , world"



[234, 5, 1089]

## Décodage

[234, 5, 1089]



"Hello , world"

# Implémentation : SimpleTokenizerV1

```
import re

class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = {i: s for s, i in vocab.items()}

    def encode(self, text):
        """Convertit du texte en liste d'IDs"""
        preprocessed = re.split(r'([,.;?_!"()\' ]|--|\s)', text)
        preprocessed = [item.strip() for item in preprocessed if item.strip()]
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids):
        """Convertit une liste d'IDs en texte"""
        text = " ".join([self.int_to_str[i] for i in ids])
        text = re.sub(r'\s+([,.;?_!"()\' ])', r'\1', text)
        return text
```

## Problème

Que se passe-t-il si on rencontre un mot inconnu ? → `KeyError` !

# Tokens spéciaux

## Deux tokens essentiels

- 1 <|unk|> (Unknown) : Remplace les mots inconnus
- 2 <|endoftext|> (End of Text) : Sépare les documents

### Exemple : <|unk|>

"Hello, do you like pizza?"

↓ (si "pizza" est inconnu)

["Hello", ",", "do", "you", "like", "<|unk|>", "?"]

### Exemple : <|endoftext|>

Document 1: "The cat sleeps." <|endoftext|>

Document 2: "The dog barks." <|endoftext|>

Document 3: "The bird sings." <|endoftext|>

## Importance

# SimpleTokenizerV2 avec tokens spéciaux

```
class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = {i: s for s, i in vocab.items()}

    def encode(self, text):
        preprocessed = re.split(r'([,.;?_!"()\' ]|--|\s)', text)
        preprocessed = [item.strip() for item in preprocessed if item.strip()]

        # Remplacer les tokens inconnus par <|unk|>
        preprocessed = [
            item if item in self.str_to_int
            else "<|unk|>"
            for item in preprocessed
        ]

        ids = [self.str_to_int[s] for s in preprocessed]
        return ids
```

## Ajout au vocabulaire

```
all_tokens.extend(["<|endoftext|>", "<|unk|>"])
vocab = {token: integer for integer, token in enumerate(all_tokens)}
```

# Introduction au Byte Pair Encoding (BPE)

## Problèmes avec la tokenisation simple

- Vocabulaire limité (seulement les mots vus pendant l'entraînement)
- Mots inconnus  $\rightarrow$  `<|unk|>` (perte d'information)
- Variations morphologiques : "run", "running", "runner" sont différents
- Taille du vocabulaire peut devenir énorme

## Solution : BPE

### Décomposer les mots en sous-mots

"running"  $\rightarrow$  ["run", "ning"]

## Avantages

- Vocabulaire plus petit
- Pas de mots inconnus (au pire, décomposition en caractères)

# Algorithme BPE : Vue d'ensemble

## Phase 1 : Entraînement

- ❶ **Initialisation** : Vocabulaire = caractères individuels
- ❷ **Itération** :
  - Compter toutes les paires de tokens adjacents
  - Fusionner la paire la plus fréquente
  - Ajouter le nouveau token au vocabulaire
- ❸ **Arrêt** : Quand la taille de vocabulaire cible est atteinte

Exemple : Corpus ["low", "lower", "newest", "widest"]

Itération	Fusion
0	Vocabulaire initial : [l, o, w, e, r, n, s, t, i, d]
1	(e, s) → es
2	(es, t) → est
3	(l, o) → lo

# BPE : Formalisme mathématique

## Notations

- $C$  : corpus d'entraînement (ensemble de mots)
- $V$  : vocabulaire (ensemble de tokens)
- $V_0$  : vocabulaire initial (caractères)
- $k$  : taille de vocabulaire cible

## Initialisation

$$V_0 = \{c \mid c \text{ est un caractère dans } C\}$$

## Comptage des paires

$$\text{freq}(t_i, t_{i+1}) = \sum_{w \in C} \sum_{j=1}^{|w|-1} \mathbb{1}[w_j = t_i \wedge w_{j+1} = t_{i+1}]$$

## Fusion

# Utilisation du tokenizer tiktoken (OpenAI)

## Installation et utilisation

```
import tiktoken

# Charger le tokenizer GPT-2
tokenizer = tiktoken.get_encoding("gpt2")

# Encoder du texte
text = "Hello, do you like tea? <|endoftext|> And some more text."
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>"})

print("IDs:", integers)
# [15496, 11, 466, 345, 588, 8887, 30, 50256, 843, 617, 517, 2420, 13]

# Decoder
decoded_text = tokenizer.decode(integers)
print("Decode:", decoded_text)
```

## Vocabulaire GPT-2

- **50,257 tokens** au total
- 256 tokens pour les bytes individuels (0-255)
- 50,000 tokens pour les sous-mots appris par BPE

# Échantillonnage avec fenêtre glissante

## Principe

Les LLMs sont entraînés avec une tâche de **prédiction du mot suivant**.  
Pour créer les paires (entrée, cible), on utilise une **fenêtre glissante**.

Exemple : "The cat sits on the mat"

Tokenisé : [1, 2, 3, 4, 5, 6] (IDs simplifiés)

Avec une fenêtre de taille 4 :

Entrée	Cible
[1, 2, 3, 4]	[2, 3, 4, 5]
[2, 3, 4, 5]	[3, 4, 5, 6]

## Important

La cible est l'entrée décalée d'une position vers la droite. Pour chaque token dans l'entrée, le modèle doit prédire le token suivant.

# Implémentation du DataLoader

```
def create_dataloader(text, batch_size=4, max_length=256, stride=128):
    tokenizer = tiktoken.get_encoding("gpt2")
    token_ids = tokenizer.encode(text)

    input_chunks = []
    target_chunks = []

    for i in range(0, len(token_ids) - max_length, stride):
        input_chunk = token_ids[i:i + max_length]
        target_chunk = token_ids[i + 1:i + max_length + 1]
        input_chunks.append(torch.tensor(input_chunk))
        target_chunks.append(torch.tensor(target_chunk))

    dataset = TensorDataset(
        torch.stack(input_chunks),
        torch.stack(target_chunks)
    )

    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
    return dataloader
```

## Paramètres importants

- `max_length` : Taille de la fenêtre de contexte
  - GPT-2 : 1024 tokens
  - GPT-3 : 2048 tokens
  - GPT-4 : 8192 tokens (ou plus)
- `stride` : Pas de déplacement de la fenêtre
  - Si `stride = max_length` : pas de chevauchement
  - Si `stride < max_length` : chevauchement (plus de données, mais redondance)
- `batch_size` : Nombre d'exemples par batch
  - Dépend de la mémoire GPU disponible
  - Typiquement : 8, 16, 32, 64

# Création des token embeddings

## Concept

Une **couche d'embedding** est une table de correspondance (lookup table) :

$$E \in \mathbb{R}^{|V| \times d}$$

Où :

- $|V|$  : taille du vocabulaire (ex : 50,257 pour GPT-2)
- $d$  : dimension de l'embedding (ex : 768 pour GPT-2)

Pour obtenir l'embedding d'un token avec ID  $i$  :

$$e_i = E[i, :] \in \mathbb{R}^d$$

## Implémentation PyTorch

```
import torch.nn as nn

vocab_size = 50257 # GPT-2
embedding_dim = 768 # GPT-2
```

# Initialisation et apprentissage des embeddings

## Initialisation

Au début de l'entraînement, les embeddings sont initialisés aléatoirement :

$$E_{ij} \sim \mathcal{N}(0, \sigma^2)$$

Typiquement,  $\sigma = 0.02$  ou  $\sigma = 1/\sqrt{d}$ .

## Apprentissage

Pendant l'entraînement, ces valeurs sont mises à jour par rétropropagation pour capturer le sens des mots.

## Différence avec Word2Vec

	Word2Vec	LLM Embeddings
Pré-entraînés	Oui	Non
Fixes	Oui	Non (appris)
Contextuels	Non	Oui (après Transformer)

# Encodage des positions

## Problème

Le mécanisme d'attention est **invariant à la permutation**. Il ne peut pas distinguer :

"Le chat mange la souris"  
de  
"La souris mange le chat"

## Solution : Encodages positionnels

Ajouter un vecteur de position à chaque embedding de token :

$$x_i = e_i + p_i$$

Où :

- $e_i$  : embedding du token à la position  $i$
- $p_i$  : encodage de la position  $i$
- $x_i$  : embedding final (token + position)

# Deux approches pour les encodages positionnels

## 1. Encodages sinusoïdaux

### Formules :

$$PE_{(pos, 2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$PE_{(pos, 2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

### Avantages :

- Pas de paramètres à apprendre
- Généralise à des longueurs non vues
- Propriétés mathématiques élégantes

**Utilisé dans :** Transformer original, BERT

## 2. Embeddings apprenables

### Principe :

$$P \in \mathbb{R}^{L \times d}$$

Où  $L$  est la longueur max de séquence.

### Avantages :

- Plus flexible
- Optimisé pour la tâche
- Simple à implémenter

### Inconvénients :

- $L \times d$  paramètres supplémentaires
- Ne généralise pas au-delà de  $L$

**Utilisé dans :** GPT-2, GPT-3, GPT-4

# Implémentation des positional embeddings

## Approche apprenable (GPT)

```
import torch.nn as nn

max_length = 1024 # GPT-2
embedding_dim = 768

# Créer la couche d'embedding positionnel
pos_embedding_layer = nn.Embedding(max_length, embedding_dim)

# Utilisation
seq_length = 4
positions = torch.arange(seq_length) # [0, 1, 2, 3]
pos_embeddings = pos_embedding_layer(positions) # [4, 768]
```

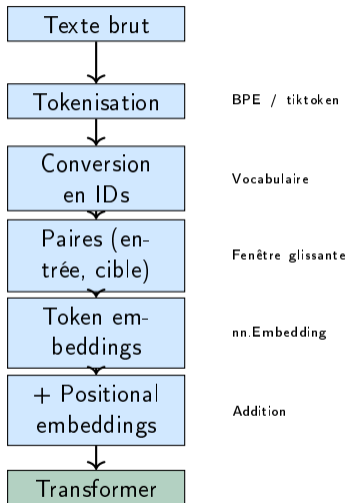
## Combinaison avec token embeddings

```
# Token embeddings
token_embeddings = token_embedding_layer(token_ids) # [batch, seq, 768]

# Positional embeddings
pos_embeddings = pos_embedding_layer(positions) # [seq, 768]

# Combiner (addition)
input_embeddings = token_embeddings + pos_embeddings # [batch, seq, 768]
```

# Pipeline complet de préparation des données



Forme finale

# Conclusion et prochaines étapes

## Ce que nous avons appris

- 1 **Word embeddings** : Représentation vectorielle des mots
- 2 **Tokenisation** : Division du texte en tokens (BPE)
- 3 **Tokens spéciaux** : `<|unk|>`, `<|endoftext|>`
- 4 **Fenêtre glissante** : Création des paires d'entraînement
- 5 **Token embeddings** : Conversion IDs  $\rightarrow$  vecteurs
- 6 **Positional embeddings** : Ajout de l'information de position

## Compétences acquises

- Préparer des données textuelles pour un LLM
- Utiliser tiktoken (tokenizer de GPT)
- Implémenter un DataLoader PyTorch
- Créer des embeddings avec PyTorch

# Questions ?

Merci de votre attention !