AYI NEDJIMI Consultants

Embeddings vs Tokens : Guide Pratique Cybersecurite

Catégorie : Intelligence Artificielle   Lecture : 18 min   Publié le : 07/12/2025   Auteur : Ayi NEDJIMI

Comprenez la différence fondamentale entre embeddings et tokens en NLP. Rôles, utilisations, transformations et impact sur les modèles de langage.

Résumé exécutif

Tokens et **embeddings** sont deux concepts fondamentaux mais distincts en NLP moderne. Les **tokens** sont des identifiants discrets (entiers) obtenus après découpage du texte via tokenisation (BPE, WordPiece). Les **embeddings** sont des vecteurs denses de nombres réels (768-12288 dimensions) qui capturent la sémantique des tokens.

Pipeline : Texte → Tokens (IDs) → Embeddings (vecteurs) → Transformers → Représentations contextuelles. Les tokens servent d'index pour récupérer les embeddings dans une matrice de lookup. Les embeddings sont ensuite enrichis par les couches Transformer pour devenir contextuels (un même token a différents embeddings selon son contexte). Comprendre cette distinction est crucial pour : optimiser les coûts API (facturation au token), dimensionner les systèmes RAG (stockage des embeddings), et choisir le bon modèle selon ses contraintes.

Qu'est-ce qu'un token ?

Un **token** est l'unité atomique de traitement du texte dans un modèle de langage. Il s'agit d'une représentation discrète et symbolique d'un fragment de texte, obtenu après la phase de **tokenisation**.

Définition formelle

Un token est un **identifiant unique (ID entier)** associé à un élément du vocabulaire d'un modèle. Il peut représenter un caractère, un mot, une partie de mot (sous-mot), ou un symbole spécial. Par exemple, le mot "intelligence" peut être tokenisé en : `["int", "elli", "gence"]` avec les IDs correspondants `[524, 8765, 15436]`.

Caractéristiques clés d'un token :

- **Nature discrète** : Un token est un symbole distinct, non continu
- **Appartenance à un vocabulaire fini** : GPT-3 utilise ~50 257 tokens, GPT-4 ~100 000 tokens
- **Pas de sémantique intrinsèque** : Le token ID 524 n'a aucune signification avant d'être converti en embedding
- **Unité de facturation** : Les APIs LLM (OpenAI, Anthropic) facturent au nombre de tokens traités

© Ayi NEDJIMI Consultants — ayinedjimi-consultants.fr

Qu'est-ce qu'un embedding ?

Un **embedding** (ou représentation vectorielle) est une transformation d'un token en un **vecteur dense de nombres réels** de dimension fixe, généralement entre 128 et 12 288 dimensions selon le modèle.

Définition mathématique

Un embedding est une fonction `E : V → ℝ^d` qui projette un token du vocabulaire `V` vers un espace vectoriel de dimension `d`. Par exemple, le token "intelligence" (ID 524) devient un vecteur `[0.012, -0.543, 0.891, ..., 0.234]` de 768 dimensions dans BERT-base.

Propriétés fondamentales des embeddings :

- **Nature continue** : Chaque dimension est un nombre réel (float32/float16)
- **Capture sémantique** : Les tokens similaires ont des embeddings proches dans l'espace vectoriel
- **Appris par le modèle** : Les valeurs sont optimisées durant l'entraînement pour minimiser la loss
- **Contextuels (LLMs modernes)** : L'embedding d'un mot change selon son contexte ("banque de données" vs "banque financière")

La relation entre les deux

Les tokens et embeddings forment un pipeline séquentiel dans tout modèle de langage moderne :

Texte brut → [Tokenisation] → **Tokens (IDs)** → [Lookup Table] → **Embeddings (vecteurs)** → [Transformers] → **Représentations contextuelles**

Exemple concret avec la phrase "ChatGPT transforme l'IA" :

```
# Étape 1 : Tokenisation (avec tiktoken pour GPT-4)
import tiktoken
enc = tiktoken.encoding_for_model("gpt-4")
tokens = enc.encode("ChatGPT change l'IA")
print(tokens)  # [13158, 38, 2898, 96265, 326, 10485, 8, 5987]

# Étape 2 : Conversion en embeddings (simplifié)
# Dans le modèle, chaque token ID est converti via une matrice d'embeddings
embedding_matrix = model.get_input_embeddings()  # Shape: [vocab_size, hidden_dim]
embeddings = embedding_matrix(tokens)  # Shape: [8, 768] pour BERT-base
print(embeddings[0][:5])  # [-0.234, 0.891, -0.012, 0.543, 0.234]
```

La **matrice d'embeddings** (ou *lookup table*) est une simple table de correspondance : chaque ligne correspond à un token du vocabulaire, et contient son vecteur d'embedding.

Pourquoi cette confusion ?

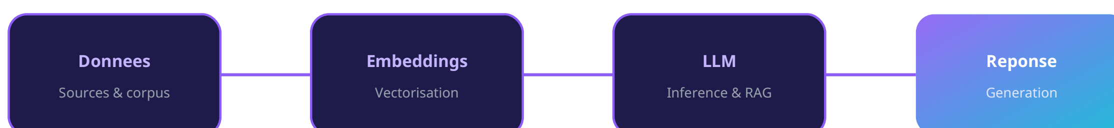
La confusion entre tokens et embeddings provient de plusieurs facteurs :

- **Terminologie ambiguë** : On entend souvent "encoder un texte en tokens" alors qu'on parle en fait d'embeddings
- **APIs simplifiées** : Les bibliothèques comme Hugging Face encapsulent la conversion token → embedding de façon transparente
- **Usage interchangeable erroné** : Dans le langage courant, "tokeniser" est parfois utilisé pour désigner l'ensemble du processus jusqu'à l'embedding
- **Abstractions masquées** : Rares sont les praticiens qui manipulent directement les token IDs ; on travaille généralement avec les embeddings

Erreur courante

Faux : "J'ai tokenisé mon texte et je l'ai envoyé au modèle."
Correct : "J'ai tokenisé mon texte (obtention des IDs), puis le modèle a converti ces tokens en embeddings avant de les traiter dans les couches Transformer."

Pipeline Intelligence Artificielle



Architecture IA - Du traitement des donnees a la generation de reponses

Comment garantir que vos modèles de machine learning ne deviennent pas des vecteurs d'attaque ?

Le processus de tokenisation

Tokenisation au niveau caractère

La tokenisation par caractères découpe le texte en unités individuelles de caractères. Chaque lettre, chiffre ou symbole devient un token.

Exemple :

```
Texte : "IA"
Tokens : ["I", "A"]
Vocabulaire : ~100-500 tokens (a-z, A-Z, 0-9, ponctuation, espaces)
```

Avantages :

- Vocabulaire minimal (pas de tokens OOV - Out Of Vocabulary)
- Capable de traiter n'importe quelle langue ou symbole

Inconvénients majeurs :

- Séquences très longues (limite de contexte atteinte rapidement)
- Perte de l'information morphologique ("chat" et "chats" partagent peu de structure)
- Performances médiocres : modèles anciens comme CharRNN (2015) sont obsolètes

Utilisation actuelle : Très rare. Uniquement pour des tâches de bas niveau (OCR, correction orthographique caractère par caractère).

Tokenisation au niveau mot

La tokenisation par mots sépare le texte en mots complets, généralement en utilisant les espaces et la ponctuation comme délimiteurs.

Exemple :

```
Texte : "L'intelligence artificielle transforme les entreprises."
Tokens : ["L'", "intelligence", "artificielle", "transforme", "les", "entreprises", "."]
Vocabulaire : 50 000 à 1 000 000 de mots
```

Avantages :

- Sémantique préservée (chaque mot = une unité de sens)
- Séquences plus courtes qu'au niveau caractère

Inconvénients critiques :

- **Vocabulaire explosif** : 1M+ mots en français si on inclut formes fléchies, néologismes, noms propres
- **Tokens OOV (Out-Of-Vocabulary)** : Incapacité à traiter les mots rares, fautes de frappe, langues mixtes
- **Mémoire prohibitive** : Matrice d'embeddings de 1M tokens × 768 dimensions = 3GB en float32

Obsolescence

Les modèles pré-2018 (Word2Vec, GloVe) utilisaient la tokenisation par mots. Cette approche n'est plus compétitive depuis l'apparition des tokeniseurs par sous-mots (BPE, WordPiece).

Tokenisation par sous-mots (BPE, WordPiece)

La tokenisation par sous-mots (subword tokenization) représente le **standard actuel** pour tous les LLMs modernes (GPT, BERT, T5, LLaMA). Elle divise les mots en fragments fréquents, optimisant le compromis entre taille de vocabulaire et expressivité. Pour approfondir, consultez Intégration d'Agents IA avec les API Externes.

Notre avis d'expert
L'IA responsable n'est pas un luxe — c'est une nécessité opérationnelle. Nos audits révèlent que 70% des déploiements IA en entreprise manquent de mécanismes de détection des biais et de garde-fous contre les injections de prompt. Il est temps d'intégrer la sécurité dès la conception des pipelines ML.

Byte-Pair Encoding (BPE)
BPE est un algorithme de compression adapté au NLP. Il fusionne itérativement les paires de caractères/sous-mots les plus fréquentes dans le corpus d'entraînement.

Algorithme simplifié :

```
# Étape 1 : Initialiser avec caractères
vocab = {"a", "b", "c", ..., "z", " "}

# Étape 2 : Fusionner les paires les plus fréquentes N fois
for _ in range(50000):  # GPT-3 utilise ~50K merges
    pair = find_most_frequent_pair(corpus)
    vocab.add(pair)  # Ex: ("th", "e") → "the"

# Étape 3 : Tokeniser nouveau texte en appliquant les merges
text = "intelligence"
tokens = bpe_tokenize(text, vocab)  # ["int", "elli", "gence"]
```

Exemple avec GPT-4 (tiktoken) :

```
import tiktoken
enc = tiktoken.encoding_for_model("gpt-4")

print(enc.encode("intelligence"))    # [396, 8677, 7761]
print(enc.encode("anticonstitutionnellement"))  # [519, 1965, 17453, 28324, 479]
print(enc.encode("🤖"))               # [9468, 97, 230]  # Émoji = plusieurs tokens
```

WordPiece (utilisé par BERT)
Variante de BPE développée par Google, utilisée dans BERT et ses dérivés. Différence clé : utilise une métrique probabiliste (likelihood) au lieu de la simple fréquence.

```
from transformers import BertTokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

print(tokenizer.tokenize("intelligence"))  # ['intelligence']  # Mot connu
print(tokenizer.tokenize("superintelligence"))  # ['super', '##int', '##ellig', '##ence']
# Note: ## indique la continuation d'un mot
```

Pourquoi BPE/WordPiece dominent ?

- **Vocabulaire optimal** : 30K-100K tokens (vs 1M+ pour les mots, 100 pour caractères)

- **Zéro OOV** : Tout texte peut être tokenisé en décomposant jusqu'aux caractères si nécessaire
- **Multilingue efficace** : Partage de sous-mots entre langues ("internationalization" en anglais ≈ "internationalisation" en français)
- **Robustesse** : Gère fautes de frappe, néologismes, noms propres

SentencePiece et tokenisation moderne

SentencePiece (Google, 2018) est une bibliothèque de tokenisation indépendante de la langue, utilisée par LLaMA, T5, ALBERT, et de nombreux modèles multilingues.

Innovations clés :

- **Traitement du texte brut** : Pas de pré-tokenisation (pas d'hypothèses sur les espaces ou la ponctuation)
- **Réversible** : Permet de retrouver exactement le texte original (important pour les langues sans espaces comme le chinois)
- **Supporte BPE et Unigram LM** : Deux algorithmes selon les besoins

```
import sentencepiece as spm

# Entraîner un tokenizer SentencePiece
spm.SentencePieceTrainer.train(
    input='corpus.txt',
    model_prefix='tokenizer',
    vocab_size=32000,
    character_coverage=0.9995,  # Couverture des caractères Unicode
    model_type='bpe'  # ou 'unigram'
)

# Utilisation
sp = spm.SentencePieceProcessor(model_file='tokenizer.model')
print(sp.encode("L'IA transforme le monde", out_type=str))
# ['▁L', "'", 'IA', '▁transform', 'e', '▁le', '▁monde']
# Note: ▁ représente un espace
```

LLaMA tokenizer

LLaMA (Meta) utilise SentencePiece avec un vocabulaire de 32 000 tokens. Particularité : chaque chiffre est un token individuel (0-9), permettant un meilleur raisonnement arithmétique.

Vocabulaire et tokens spéciaux

Tous les tokenizers incluent des **tokens spéciaux** pour encoder la structure et les limites des séquences.

Token	Rôle	Exemple d'utilisation
`[PAD]`	Padding pour uniformiser la longueur des batchs	Séquences de longueurs variables dans un batch
`[UNK]`	Token inconnu (rare avec BPE)	Caractères Unicode non couverts
`[CLS]`	Classification (BERT)	Début de séquence, embedding utilisé pour classification
`[SEP]`	Séparateur entre phrases (BERT)	Séparer question/contexte dans QA
`<|endoftext|>`	Fin de document (GPT)	Séparation entre documents dans le corpus
`<|im_start|>`, `<|im_end|>`	Marqueurs de rôle (ChatGPT)	Délimiter messages user/assistant/system

Exemple avec BERT :

Avez-vous évalué les risques d'injection de prompt sur vos systèmes d'IA en production ?

De tokens à embeddings : la transformation

Couche d'embedding dans les réseaux de neurones

La **couche d'embedding** (Embedding Layer) est la première couche de tout modèle de langage. C'est une simple **matrice de poids apprenables** qui convertit les tokens discrets en vecteurs denses.

Architecture mathématique :

```
# Pseudo-code PyTorch
class EmbeddingLayer(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super().__init__()
        # Matrice d'embeddings : shape (vocab_size, embedding_dim)
        self.weight = nn.Parameter(torch.randn(vocab_size, embedding_dim))

    def forward(self, token_ids):
        # token_ids : shape (batch_size, seq_len)
        # Lookup simple via indexation
        embeddings = self.weight[token_ids]  # Shape: (batch_size, seq_len, embedding_dim)
        return embeddings
```

Exemple concret avec GPT-2 :

```
from transformers import GPT2Model
import torch

model = GPT2Model.from_pretrained('gpt2')  # 124M paramètres
embedding_layer = model.wte  # Word Token Embeddings

print(f"Vocabulaire : {embedding_layer.num_embeddings} tokens")  # 50257
print(f"Dimension : {embedding_layer.embedding_dim}")  # 768
print(f"Taille matrice : {embedding_layer.num_embeddings * embedding_layer.embedding_dim *
4 / 1e6:.1f} MB")  # ~154 MB

# Lookup d'un token
token_id = torch.tensor([[15496]])  # Token "intelligence"
embedding = embedding_layer(token_id)
print(embedding.shape)  # torch.Size([1, 1, 768])
print(embedding[0, 0, :5])  # tensor([-0.0234,  0.0891, -0.0012,  0.0543,  0.0234],
grad_fn=)
```

Pourquoi une matrice et pas un modèle complexe ?

La couche d'embedding est intentionnellement simple (lookup table). La complexité sémantique est **apprise** durant l'entraînement via la backpropagation. Les valeurs initiales (aléatoires) sont progressivement optimisées pour minimiser la loss du modèle sur des milliards de tokens.

Token IDs vers vecteurs denses

Le passage de tokens (entiers discrets) à embeddings (vecteurs denses) est une opération de **projection** dans un espace continu à haute dimension.

Transformation mathématique :

Token ID (scalaire discret) ∈ {0, 1, ..., V-1}
↓ Indexation dans matrice d'embeddings
Embedding (vecteur dense) ∈ ℝ^d

Formellement : `E[token_id] = [e₁, e₂, ..., e_d]` où `eᵢ ∈ ℝ`

Exemple de transformation avec dimensions réduites (pédagogique) :

```
import numpy as np

# Vocabulaire simplifié : 5 tokens, embeddings de dimension 3
vocab = ["[PAD]", "chat", "mange", "souris", "."]
embedding_matrix = np.array([
    [0.0, 0.0, 0.0],      # [PAD] : vecteur nul
    [0.8, 0.2, -0.1],     # "chat" : animal domestique
    [-0.3, 0.9, 0.5],     # "mange" : action
    [0.7, 0.1, -0.2],     # "souris" : animal (proche de "chat")
    [0.0, 0.0, 1.0]       # "." : ponctuation
])

# Phrase : "chat mange souris ."
token_ids = [1, 2, 3, 4]
embeddings = embedding_matrix[token_ids]

print("Token IDs:", token_ids)
print("Embeddings:")
print(embeddings)
# [[0.8  0.2 -0.1]
#  [-0.3  0.9  0.5]
#  [0.7  0.1 -0.2]
#  [0.0  0.0  1.0]]

# Similarité cosinus entre "chat" et "souris"
from sklearn.metrics.pairwise import cosine_similarity
sim = cosine_similarity([embeddings[0]], [embeddings[2]])[0][0]
print(f"Similarité chat-souris : {sim:.3f}")  # ~0.95 (très similaires)
```

Apprentissage des embeddings

Les embeddings ne sont pas prédéfinis : ils sont **appris automatiquement** durant l'entraînement du modèle en minimisant une fonction de perte (loss) sur une tâche spécifique.

Processus d'apprentissage (entraînement supervisé)

1. **Initialisation** : Valeurs aléatoires (distribution normale centrée réduite)
2. **Forward pass** : Tokens → Embeddings → Transformers → Prédiction
3. **Calcul de la loss** : Comparaison prédiction vs vérité terrain (cross-entropy pour LLMs)
4. **Backpropagation** : Calcul des gradients ∂Loss/∂Embeddings
5. **Mise à jour** : Ajustement des valeurs d'embeddings via optimiseur (Adam, AdamW)
6. **Répétition** : Des milliards d'itérations sur des téraoctets de texte

Exemple simplifié d'entraînement :

```
import torch
import torch.nn as nn

# Modèle minimal
class SimpleLM(nn.Module):
    def __init__(self, vocab_size=1000, embedding_dim=128, hidden_dim=256):
        super().__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, vocab_size)

    def forward(self, token_ids):
        embeds = self.embeddings(token_ids)  # Lookup
        lstm_out, _ = self.lstm(embeds)
        logits = self.fc(lstm_out)
        return logits

model = SimpleLM()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
loss_fn = nn.CrossEntropyLoss()

# Boucle d'entraînement (simplifié)
for epoch in range(100):
    for batch in dataloader:
        token_ids, targets = batch

        # Forward
        logits = model(token_ids)
        loss = loss_fn(logits.view(-1, vocab_size), targets.view(-1))

        # Backward
        optimizer.zero_grad()
        loss.backward()  # Les gradients se propagent jusqu'aux embeddings
        optimizer.step()  # Mise à jour des poids, y compris embeddings

        print(f"Loss: {loss.item():.4f}")
```

Résultat de l'apprentissage

Après entraînement, les embeddings capturent automatiquement la sémantique : les tokens "roi" et "reine" auront des vecteurs proches, "Paris" et "France" aussi. Ces relations émergent naturellement de la tâche de prédiction, sans supervision explicite sur la similarité.

Embeddings contextuels vs statiques

Une distinction majeure dans l'évolution du NLP : **embeddings statiques** (2013-2018) vs **embeddings contextuels** (2018-aujourd'hui).

Embeddings statiques (Word2Vec, GloVe, FastText)
Chaque mot a **un seul vecteur fixe**, indépendamment du contexte.

```
# Word2Vec (2013)
from gensim.models import Word2Vec

model = Word2Vec.load("word2vec-google-news-300.bin")
print(model.wv["banque"])  # Toujours le même vecteur de 300 dimensions
# [-0.123, 0.456, ..., -0.789]

# Problème : impossible de distinguer les sens
print(model.wv.similarity("banque", "argent"))  # Banque financière ?
print(model.wv.similarity("banque", "données"))  # Banque de données ?
```

Embeddings contextuels (BERT, GPT, RoBERTa)

Chaque token a **un vecteur différent selon le contexte** de la phrase.

```
from transformers import BertModel, BertTokenizer
import torch

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Phrase 1 : Sens financier
text1 = "Je vais à la banque retirer de l'argent."
inputs1 = tokenizer(text1, return_tensors="pt")
outputs1 = model(**inputs1)
embedding_banque_1 = outputs1.last_hidden_state[0, 5, :]  # Index du token "banque"

# Phrase 2 : Sens base de données
text2 = "Cette banque de données contient des millions d'images."
inputs2 = tokenizer(text2, return_tensors="pt")
outputs2 = model(**inputs2)
embedding_banque_2 = outputs2.last_hidden_state[0, 2, :]

# Comparaison
from torch.nn.functional import cosine_similarity
sim = cosine_similarity(embedding_banque_1, embedding_banque_2, dim=0)
print(f"Similarité contextes différents : {sim.item():.3f}")  # ~0.6-0.7 (modérée)

# Même mot, même contexte
text3 = "Je vais à la banque déposer de l'argent."
inputs3 = tokenizer(text3, return_tensors="pt")
outputs3 = model(**inputs3)
embedding_banque_3 = outputs3.last_hidden_state[0, 5, :]
sim2 = cosine_similarity(embedding_banque_1, embedding_banque_3, dim=0)
print(f"Similarité contextes similaires : {sim2.item():.3f}")  # ~0.95+ (très élevée)
```

Critère	Embeddings statiques (Word2Vec)	Embeddings contextuels (BERT/GPT)
Unicité	1 vecteur par mot	∞ vecteurs (dépend du contexte)
Polysémie	Non géré ("avocat" = fruit+métier)	Géré parfaitement
Performance	Lookup O(1), léger	Inference Transformer $O(n^2)$, lourd
Précision NLP	60-70% sur benchmarks	85-95% (SOTA)
Usage actuel	Obsolète (sauf contraintes extrêmes)	Standard industriel

Visualisation du pipeline complet

Voici le pipeline complet d'un modèle Transformer moderne (ex: BERT) du texte brut à la prédiction :

Implémentation PyTorch simplifiée du pipeline complet : Pour approfondir, consultez Orchestration d'Agents IA : Patterns et Anti-Patterns.

```
from transformers import BertTokenizer, BertModel
import torch

# Chargement
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Texte
text = "L'intelligence artificielle transforme le monde"

# Étape 1: Tokenisation
encoded = tokenizer(text, return_tensors='pt', padding=True, truncation=True)
print("Token IDs:", encoded['input_ids'])
print("Attention Mask:", encoded['attention_mask'])

# Étape 2-4: Forward pass (embeddings + transformers)
with torch.no_grad():
    outputs = model(**encoded)

# Étape 5: Extraction des représentations
last_hidden_state = outputs.last_hidden_state  # [batch, seq_len, 768]
cls_embedding = last_hidden_state[:, 0, :]  # Embedding [CLS] pour classification

print(f"Shape last hidden state: {last_hidden_state.shape}")
print(f"CLS embedding (pour classification): {cls_embedding.shape}")
print(f"Premiers 5 valeurs CLS: {cls_embedding[0, :5]}")
```

Différences clés et tableau comparatif

Nature des représentations

La différence fondamentale réside dans la nature mathématique des objets :

TOKEN

- • **Type :** Entier discret
- • **Exemple :** 15496 (ID du token "intelligence")
- • **Espace :** Fini {0, 1, ..., V-1}
- • **Opérations :** Égalité, lookup
- • **Sémantique :** Aucune (symbole arbitraire)

EMBEDDING

- • **Type :** Vecteur de réels (float)
- • **Exemple :** [-0.234, 0.891, ..., 0.543] (768 dim)
- • **Espace :** Continu $\mathbb{R}^d$
- • **Opérations :** Distance, similarité, algèbre vectorielle
- • **Sémantique :** Capture le sens (tokens similaires = vecteurs proches)

Dimensionnalité

Les dimensions caractéristiques varient considérablement entre les modèles :

Modèle	Taille Vocabulaire (tokens)	Dimension Embeddings	Taille Matrice
BERT-base	30 522	768	~94 MB
GPT-2	50 257	768	~154 MB
GPT-3	50 257	12 288	~2.5 GB
GPT-4	~100 000	Inconnu (~20 000 estimé)	~8 GB estimé
LLaMA 2 (70B)	32 000	8 192	~1 GB
T5-11B	32 128	1 024	~131 MB

Relation entre taille vocabulaire et dimension :

- **Vocabulaire plus large** (↑) = moins de tokens par phrase, mais matrice d'embeddings plus lourde
- **Dimension plus élevée** (↑) = plus de capacité représentationnelle, mais coût computationnel accru
- **Compromis optimal** : 30K-100K tokens × 768-8192 dimensions pour les modèles modernes

Rôle dans le pipeline NLP

Tokens et embeddings interviennent à des stades différents du traitement :

Pipeline de traitement

1. **1. Pré-traitement** : Nettoyage du texte brut (lowercasing, normalisation Unicode)
2. **2. Tokenisation** : Texte → Tokens (IDs) via BPE/WordPiece
3. **3. Embedding Lookup** : Tokens → Embeddings (vecteurs) via matrice
4. **4. Traitement neural** : Embeddings traversés par les couches Transformer
5. **5. Post-traitement** : Conversion des logits en tokens (détokenisation pour génération)

Qui manipule quoi ?

- **Développeur/Utilisateur** : Manipule principalement le texte brut et les tokens (comptage, pricing)
- **Tokenizer** : Convertit texte ↔ tokens (encode/decode)
- **Modèle** : Convertit tokens → embeddings, puis traite les embeddings
- **Base vectorielle (RAG)** : Stocke et recherche sur les embeddings finaux (après Transformer)

Interprétabilité

Tokens et embeddings diffèrent radicalement en termes de lisibilité humaine :

Tokens : Interprétables

- ✔ **Décodage direct** : ID 15496 → "intelligence"
- ✔ **Debugging facile** : On peut lire la liste des tokens
- ✔ **Comptage explicite** : "Cette phrase fait 8 tokens"

- ✔ **Transparence** : Permet d'auditer le découpage

Embeddings : Opaques

- ✖ **Pas de sens direct** : [-0.234, 0.891, ...] = ?
- ✖ **Debugging complexe** : Nécessite visualisation (t-SNE, PCA)
- ✔ **Relations mesurables** : Similarité cosinus entre vecteurs
- ✔ **Algèbre sémantique** : roi - homme + femme ≈ reine

Exemple pratique de débugging :

```
from transformers import GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
text = "L'IA bouleverse le monde"

# Débogage au niveau tokens (FACILE)
tokens = tokenizer.tokenize(text)
print("Tokens:", tokens)  # ['L', "'", 'IA', 'Ġr', 'é', 'volution', 'ne', 'Ġle', 'Ġmonde']
print("Nombre:", len(tokens))  # 9 tokens
print("Budget:", len(tokens) * 0.00002, "$")  # 0.00018$ (pricing GPT-4)

# Débogage au niveau embeddings (DIFFICILE)
encoded = tokenizer(text, return_tensors='pt')
embeddings = model.transformer.wte(encoded['input_ids'])
print("Embeddings shape:", embeddings.shape)  # [1, 9, 768]
print("Premier embedding:", embeddings[0, 0, :5])  # Pas directement interprétable
# tensor([-0.0234,  0.0891, -0.0012,  0.0543,  0.0234]) <- Que signifie ce vecteur ?
```

Tableau récapitulatif détaillé

Critère	Token	Embedding
Type mathématique	Entier discret	Vecteur de réels (float32/16)
Exemple	15496	[-0.234, 0.891, ..., 0.543]
Dimension	Scalaire (1 valeur)	Vecteur (128-12288 dimensions)
Espace	Fini : {0, ..., vocab_size-1}	Continu : $\mathbb{R}^d$
Sémantique	Aucune (identifiant arbitraire)	Capture le sens (similarité géométrique)
Opérations	Égalité, lookup, comptage	Distance, similarité, addition, multiplication
Taille mémoire	2-4 bytes (int16/32)	512-49152 bytes (128-12288 × 4 bytes)
Interprétabilité	Facile (décodage direct)	Difficile (visualisation nécessaire)
Rôle dans le modèle	Indexation de la matrice d'embeddings	Input des couches Transformer
Conversion	Texte → Token (tokenisation)	Token → Embedding (lookup), puis Embedding → Embedding contextuel (Transformer)
Utilisation en production	Comptage (pricing API), limite de contexte	Recherche vectorielle (RAG), clustering, similarité
Facturation API	✔ Unité de facturation (ex: $0.03/1K tokens pour GPT-4)	✖ Pas facturé directement
Stockage base vectorielle	✖ Pas stocké (trop peu informatif)	✔ Stocké pour recherche de similarité
Contexte dépendant	✖ Statique ("banque" = toujours ID 5432)	✔ Dynamique avec Transformers ("banque" a différents embeddings selon contexte)

⚠️ Piège fréquent

Ne confondez pas :
- **Token embeddings** (couche 0, lookup statique de la matrice) : [vocab_size, hidden_dim]
- **Contextualized embeddings** (sortie des Transformers) : [seq_len, hidden_dim]
Les deux sont des "embeddings" mais servent à des fins différentes. Pour le RAG, on utilise les embeddings contextuels finaux.

Rôle dans les modèles de langage

Architecture BERT

BERT (Bidirectional Encoder Representations from Transformers, Google 2018) utilise une architecture encoder-only avec trois types d'embeddings combinés.

Les trois embeddings de BERT :

```
# Structure d'entrée BERT
final_embedding = token_embedding + position_embedding + segment_embedding

# 1. Token Embeddings : représentation du mot
# 2. Position Embeddings : position dans la séquence (0, 1, 2, ..., 511)
# 3. Segment Embeddings : appartenance à la phrase A ou B (pour NSP task)
```

Vocabulaire BERT-base-uncased

- **Taille vocabulaire :** 30 522 tokens
- **Dimension embeddings :** 768
- **Tokenizer :** WordPiece
- **Contexte max :** 512 tokens
- **Tokens spéciaux :** [CLS], [SEP], [PAD], [UNK], [MASK]

Exemple concret avec BERT :

Comparaison GPT-2 vs GPT-3 vs GPT-4

Modèle	Vocabulaire	Dim. Embeddings	Contexte max
GPT-2	50 257	768 (small) - 1600 (XL)	1 024 tokens
GPT-3	50 257	12 288	2 048 tokens
GPT-4	~100 000	Non divulgué (~20K estimé)	8 192 / 32 768 / 128 000 tokens

Exemple avec GPT-2 :

```
from transformers import GPT2LMHeadModel, GPT2Tokenizer
import torch

tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
model = GPT2LMHeadModel.from_pretrained('gpt2')

prompt = "L'intelligence artificielle est"
encoded = tokenizer(prompt, return_tensors='pt')

print("Token IDs:", encoded['input_ids'])
print("Tokens:", tokenizer.convert_ids_to_tokens(encoded['input_ids'][0]))
# ["L'", 'int', 'ellig', 'ence', 'Ġartificielle', 'Ġest']

# Génération auto-régressive
with torch.no_grad():
    outputs = model.generate(
        encoded['input_ids'],
        max_length=50,
        num_return_sequences=1,
        temperature=0.7,
        do_sample=True
    )

generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
print("\nTexte généré:", generated_text)

# Comptage tokens pour pricing
total_tokens = len(outputs[0])
print(f"\nTokens utilisés: {total_tokens}")
print(f"Coût estimé (GPT-4 pricing): ${total_tokens * 0.00003:.6f}")
```

Différence clé BERT vs GPT

- **BERT** : Voit tout le contexte (bidirectionnel). Idéal pour : classification, NER, QA
- **GPT** : Voit seulement le contexte gauche (unidirectionnel). Idéal pour : génération, complétion
- **Usage embeddings** : BERT [CLS] pour similarité, GPT dernier token pour continuer la génération

Modèles d'embeddings spécialisés (Sentence-BERT)

Les modèles comme **Sentence-BERT (SBERT)** sont optimisés spécifiquement pour produire des embeddings de phrases de haute qualité, utilisés massivement dans les systèmes RAG.

Problème avec BERT vanilla

BERT n'est pas directement utilisable pour comparer des phrases :

- Comparer 10 000 phrases entre elles = 10 000² = 100M de paires à passer dans BERT → **impraticable**
- L'embedding [CLS] n'est pas optimisé pour la similarité sémantique inter-phrases

Solution : Sentence-BERT (2019)

SBERT fine-tune BERT avec un objectif de contrastive learning (triplet loss) pour que les embeddings [CLS] capturent directement la sémantique de la phrase complète.

```python
from sentence_transformers import SentenceTransformer, util
import numpy as np

# Chargement modèle optimisé pour embeddings
model = SentenceTransformer('all-MiniLM-L6-v2')  # 384 dimensions, rapide

# Corpus de documents
documents = [
    "L'intelligence artificielle transforme les entreprises.",
    "Le machine learning améliore la précision des prédictions.",
    "Paris est la capitale de la France.",
    "Les réseaux de neurones profonds nécessitent beaucoup de données."
]

# Génération embeddings (1 forward pass par phrase)
doc_embeddings = model.encode(documents, convert_to_tensor=True)
print(f"Shape embeddings: {doc_embeddings.shape}")  # [4, 384]

# Requête utilisateur
query = "Comment l'IA impacte les organisations ?"
query_embedding = model.encode(query, convert_to_tensor=True)

# Recherche de similarité (produit scalaire / cosine)
similarities = util.cos_sim(query_embedding, doc_embeddings)[0]
print("\nSimilarités:")
for i, (doc, score) in enumerate(zip(documents, similarities)):
    print(f"{i+1}. [{score:.3f}] {doc}")

# Résultat :
# 1. [0.687] L'intelligence artificielle transforme les entreprises.  <- Pertinent
# 2. [0.512] Le machine learning améliore la précision des prédictions.
# 3. [0.089] Paris est la capitale de la France.  <- Non pertinent
# 4. [0.423] Les réseaux de neurones profonds nécessitent beaucoup de données.
```

Modèles d'embeddings recommandés (2025)

- **all-MiniLM-L6-v2** : 384 dim, rapide, bon compromis (usage général)
- **bge-large-en-v1.5** : 1024 dim, SOTA sur MTEB benchmark
- **text-embedding-3-small/large** (OpenAI) : API payante, excellentes performances
- **multilingual-e5-large** : Multilingue (100+ langues), 1024 dim

Position embeddings et segment embeddings

Au-delà des token embeddings, les Transformers utilisent d'autres types d'embeddings pour encoder la structure.

Position Embeddings
Les Transformers n'ont **aucune notion intrinsèque de l'ordre** des tokens (contrairement aux RNN). Les position embeddings injectent cette information.

Deux approches : Pour approfondir, consultez Fuzzing Assisté par IA : Découverte de Vulnérabilités.

1. **Sinusoïdales (Transformer original)** : Fonctions périodiques fixes

```
PE(pos, 2i) = sin(pos / 10000^(2i/d_model))
PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))
```

2. **Apprises (BERT, GPT)** : Matrice de poids apprenables [max_seq_len, hidden_dim]

```
# BERT : position embeddings appris pour positions 0-511
position_embeddings = nn.Embedding(512, 768)  # [512, 768]
```

Segment Embeddings (BERT)
Permettent de distinguer deux phrases dans une paire (utile pour Next Sentence Prediction).

```
from transformers import BertTokenizer, BertModel
import torch

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Paire de phrases
sentence_a = "Paris est une ville."
sentence_b = "Elle est en France."

encoded = tokenizer(
    sentence_a,
    sentence_b,
    return_tensors='pt',
    add_special_tokens=True
)

print("Tokens:", tokenizer.convert_ids_to_tokens(encoded['input_ids'][0]))
# ['[CLS]', 'paris', 'est', 'une', 'ville', '.', '[SEP]', 'elle', 'est', 'en', 'france',
'.', '[SEP]']

print("\nToken type IDs (segment embeddings):")
print(encoded['token_type_ids'][0])
# tensor([0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])
#          ^^^^^^^^^^^^^^^ Sentence A  ^^^^^^^^^^^ Sentence B

# Forward pass combinant les 3 embeddings
with torch.no_grad():
    outputs = model(**encoded)
    final_embeddings = outputs.last_hidden_state

print(f"\nFinal embeddings shape: {final_embeddings.shape}")  # [1, 13, 768]
print("Composition : token_emb + position_emb + segment_emb")
```

Récapitulatif des embeddings dans les Transformers

Type	Rôle	Shape
Token Embeddings	Capture le sens lexical	[vocab_size, hidden_dim]
Position Embeddings	Encode la position dans la séquence	[max_seq_len, hidden_dim]
Segment Embeddings	Distingue phrases A/B (BERT)	[2, hidden_dim]

Exemples pratiques en Python

Tokenisation avec Hugging Face Transformers

Hugging Face Transformers offre une API unifiée pour travailler avec tous les tokenizers modernes.

```
from transformers import AutoTokenizer

# Chargement automatique du tokenizer adapté au modèle
tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')

text = "L'intelligence artificielle transforme les entreprises en 2025."

# Méthode 1 : Tokenisation simple (liste de strings)
tokens = tokenizer.tokenize(text)
print("Tokens:", tokens)
# ['l', "'", 'intelligence', 'artific', '##ielle', 'revolution', '##ne', 'les',
'entreprises', 'en', '2025', '.']

# Méthode 2 : Encodage complet (avec tokens spéciaux et padding)
encoded = tokenizer(
    text,
    add_special_tokens=True,  # Ajoute [CLS] et [SEP]
    padding='max_length',     # Pad jusqu'à 512 tokens
    truncation=True,          # Tronque si > 512
    max_length=20,
    return_tensors='pt'       # Retourne PyTorch tensors
)

print("\nToken IDs:", encoded['input_ids'][0][:15])  # Premiers 15 IDs
print("Attention Mask:", encoded['attention_mask'][0][:15])
# Attention mask : 1 = token réel, 0 = padding

# Décodage (token IDs -> texte)
decoded = tokenizer.decode(encoded['input_ids'][0], skip_special_tokens=True)
print("\nTexte décodé:", decoded)

# Inspection détaillée
print("\nMapping Token <-> ID:")
for token, id in zip(tokens[:5], tokenizer.convert_tokens_to_ids(tokens[:5])):
    print(f"{token:20s} -> ID {id}")
```

Extraction des embeddings

Une fois les tokens obtenus, extrayons les embeddings à différents niveaux du modèle.

```
from transformers import AutoModel, AutoTokenizer
import torch

tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
model = AutoModel.from_pretrained('bert-base-uncased')

text = "Les embeddings capturent la sémantique."
encoded = tokenizer(text, return_tensors='pt')

with torch.no_grad():
    outputs = model(**encoded, output_hidden_states=True)

# 1. Token embeddings (couche 0, avant Transformers)
token_embeddings_layer0 = model.embeddings.word_embeddings(encoded['input_ids'])
print(f"Token embeddings (layer 0): {token_embeddings_layer0.shape}")  # [1, seq_len, 768]

# 2. Embeddings après chaque couche Transformer
all_hidden_states = outputs.hidden_states  # Tuple de 13 tensors (input + 12 layers)
print(f"\nNombre de couches: {len(all_hidden_states)}")
for i, hidden_state in enumerate(all_hidden_states):
    print(f"Layer {i}: {hidden_state.shape}")

# 3. Embedding final (sortie dernière couche)
final_embeddings = outputs.last_hidden_state  # [1, seq_len, 768]
print(f"\nFinal embeddings: {final_embeddings.shape}")

# 4. Extraction embedding [CLS] pour classification/similarité
cls_embedding = final_embeddings[:, 0, :]  # Premier token = [CLS]
print(f"CLS embedding: {cls_embedding.shape}")  # [1, 768]
print(f"Premières valeurs: {cls_embedding[0, :10]}")

# 5. Moyenne des embeddings (sentence embedding alternatif)
mean_embedding = final_embeddings.mean(dim=1)  # Moyenne sur seq_len
print(f"\nMean pooling embedding: {mean_embedding.shape}")  # [1, 768]

# 6. Extraction embeddings de tokens spécifiques
tokens = tokenizer.convert_ids_to_tokens(encoded['input_ids'][0])
print(f"\nToken 'semantique' (index 5):")
print(f"Embedding: {final_embeddings[0, 5, :5]}")  # Premières 5 dims
```

Comparaison tokens vs embeddings sur un exemple concret

Visualisons la différence entre tokens et embeddings sur un cas pratique de recherche de similarité.

```python
from transformers import AutoTokenizer, AutoModel
from sklearn.metrics.pairwise import cosine_similarity
import torch
import numpy as np

tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
model = AutoModel.from_pretrained('bert-base-uncased')

# Trois phrases à comparer
sentences = [
    "Le chat mange une souris.",
    "Un félin dévore un rongeur.",  # Sémantiquement similaire
    "La voiture roule sur l'autoroute."  # Sémantiquement différent
]

# APPROCHE 1 : Comparaison au niveau TOKENS (naïve, incorrecte)
print("=== COMPARAISON TOKENS (INCORRECTE) ===")
for i, sent in enumerate(sentences):
    tokens = tokenizer.tokenize(sent)
    print(f"Phrase {i+1}: {tokens}")

# Overlap de tokens entre phrase 1 et 2
tokens1 = set(tokenizer.tokenize(sentences[0]))
tokens2 = set(tokenizer.tokenize(sentences[1]))
overlap = len(tokens1.intersection(tokens2)) / len(tokens1.union(tokens2))
print(f"\nJaccard similarity (tokens) phrase 1-2: {overlap:.3f}")
# Résultat : très faible (~0.1) alors que phrases sont similaires sémantiquement

# APPROCHE 2 : Comparaison au niveau EMBEDDINGS (correcte)
print("\n=== COMPARAISON EMBEDDINGS (CORRECTE) ===")

def get_sentence_embedding(text):
    encoded = tokenizer(text, return_tensors='pt', padding=True, truncation=True)
    with torch.no_grad():
        outputs = model(**encoded)
    # Moyenne des embeddings (mean pooling)
    return outputs.last_hidden_state.mean(dim=1).numpy()

embeddings = [get_sentence_embedding(sent) for sent in sentences]

# Calcul similarités cosinus
sim_1_2 = cosine_similarity(embeddings[0], embeddings[1])[0][0]
sim_1_3 = cosine_similarity(embeddings[0], embeddings[2])[0][0]
sim_2_3 = cosine_similarity(embeddings[1], embeddings[2])[0][0]

print(f"Cosine similarity (embeddings):")
print(f"  Phrase 1 - Phrase 2: {sim_1_2:.3f}  # HAUTE (sémantiquement proches)")
print(f"  Phrase 1 - Phrase 3: {sim_1_3:.3f}  # BASSE (sémantiquement différentes)")
print(f"  Phrase 2 - Phrase 3: {sim_2_3:.3f}  # BASSE")

# Résultats typiques :
# Phrase 1 - Phrase 2: 0.85-0.92 (très similaires malgré vocabulaire différent)
# Phrase 1 - Phrase 3: 0.45-0.60 (peu similaires)

print("\n✅ Les embeddings capturent correctement la similarité sémantique.")
print("❌ Les tokens seuls échouent (ne voient que le vocabulaire exact).")
```

Visualisation avec t-SNE

Pour comprendre intuitivement les embeddings, utilisons t-SNE pour les projeter en 2D.

```
from transformers import AutoTokenizer, AutoModel
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import torch
import numpy as np

tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
model = AutoModel.from_pretrained('bert-base-uncased')

# Corpus de mots à visualiser
words = [
    # Groupe 1 : Animaux
    "chat", "chien", "lion", "tigre", "souris",
    # Groupe 2 : Véhicules
    "voiture", "camion", "moto", "bus", "train",
    # Groupe 3 : Technologies
    "ordinateur", "smartphone", "tablette", "serveur", "logiciel",
    # Groupe 4 : Pays
    "France", "Allemagne", "Italie", "Espagne", "Portugal"
]

# Extraction embeddings pour chaque mot
embeddings = []
for word in words:
    encoded = tokenizer(word, return_tensors='pt')
    with torch.no_grad():
        output = model(**encoded)
    # Embedding du token principal (ignorer [CLS] et [SEP])
    word_embedding = output.last_hidden_state[0, 1, :].numpy()
    embeddings.append(word_embedding)

embeddings = np.array(embeddings)  # Shape: [20, 768]

print(f"Embeddings shape: {embeddings.shape}")

# Réduction de dimension : 768D -> 2D
tsne = TSNE(n_components=2, random_state=42, perplexity=5)
embeddings_2d = tsne.fit_transform(embeddings)

print(f"Embeddings 2D shape: {embeddings_2d.shape}")

# Visualisation
plt.figure(figsize=(12, 8))
colors = ['red']*5 + ['blue']*5 + ['green']*5 + ['orange']*5

for i, (word, color) in enumerate(zip(words, colors)):
    x, y = embeddings_2d[i]
    plt.scatter(x, y, c=color, s=200, alpha=0.6)
    plt.annotate(word, (x, y), fontsize=12, ha='center')

plt.title("Visualisation t-SNE des embeddings BERT (768D -> 2D)", fontsize=14)
plt.xlabel("Dimension 1")
plt.ylabel("Dimension 2")

# Légende
from matplotlib.patches import Patch
legend_elements = [
    Patch(facecolor='red', label='Animaux'),
    Patch(facecolor='blue', label='Véhicules'),
    Patch(facecolor='green', label='Technologies'),
    Patch(facecolor='orange', label='Pays')
]
```

```
plt.legend(handles=legend_elements, loc='best')

plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('embeddings_tsne.png', dpi=300)
print("\n✅ Visualisation sauvegardée : embeddings_tsne.png")
print("\nObservation : Les mots sémantiquement similaires forment des clusters.")
print("Les embeddings de dimension 768 encodent cette structure sémantique.")
```

Interprétation du graphique

Dans la projection 2D, vous observerez :
- **Clusters sémantiques** : Les animaux sont groupés ensemble, les véhicules aussi, etc.
- **Distances relatives** : "chat" est plus proche de "chien" que de "voiture"
- **Continuité de l'espace** : Pas de frontières nettes, transitions progressives

Ceci est impossible avec des tokens (IDs discrets) : le token ID 5432 n'a aucune relation géométrique avec 5433.

Limitations et considérations

Taille de vocabulaire et tokens inconnus (OOV)

La gestion des tokens hors vocabulaire (Out-Of-Vocabulary) est un défi majeur résolu différemment selon l'approche.

Problème OOV classique (Word2Vec, GloVe)

Avec tokenisation par mots complets :
- Vocabulaire typ. : 50K-100K mots les plus fréquents
- Mots rares, fautes, néologismes → token [UNK] (Unknown)
- Perte totale d'information : "superintelligence" = [UNK] = "zxqwerty" = [UNK]
- Impact : 2-5% des tokens en production sont [UNK]

Solution : Tokenisation par sous-mots (BPE, WordPiece)

Avec BPE/WordPiece :
- Vocabulaire optimal : 30K-50K sous-mots
- **Zéro OOV** : Tout mot peut être décomposé jusqu'aux caractères
- "superintelligence" → ["super", "##int", "##ellig", "##ence"]
- Conservation partielle du sens même pour mots inconnus

```
from transformers import BertTokenizer, GPT2Tokenizer

# Comparaison tokenizers
bert_tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
gpt2_tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

# Mots complexes / néologismes
words = [
    "anticonstitutionnellement",
    "cryptomonnaie",
    "ChatGPT",
    "zxqwerty"  # Mot totalement inventé
]

print("Tokenisation de mots rares/inconnus:\n")
for word in words:
    bert_tokens = bert_tokenizer.tokenize(word)
    gpt2_tokens = gpt2_tokenizer.tokenize(word)

    print(f"Mot: {word}")
    print(f"  BERT:  {bert_tokens} ({len(bert_tokens)} tokens)")
    print(f"  GPT-2: {gpt2_tokens} ({len(gpt2_tokens)} tokens)")
    print()

# Résultat typique :
# Mot: anticonstitutionnellement
#   BERT:  ['anti', '##con', '##stit', '##ution', '##nelle', '##ment'] (6 tokens)
#   GPT-2: ['Ġanti', 'const', 'itut', 'ion', 'nelle', 'ment'] (6 tokens)

# Même "zxqwerty" est tokenisé en caractères : ['z', '##x', '##q', '##w', '##erty']
```

Impact sur les embeddings :

- Avec tokenisation par mots : Tous les mots OOV ont le **même embedding [UNK]** → perte d'information catastrophique
- Avec BPE : Chaque sous-mot a son propre embedding → représentation compositionnelle préservée

Dimensionnalité des embeddings

Le choix de la dimension des embeddings est un compromis entre expressivité et efficacité computationnelle.

Dimension	Avantages	Inconvénients	Cas d'usage
128-384	Très rapide, léger, faible latence	Capacité représentationnelle limitée	Modèles mobiles, edge computing, prototypage
768	Standard industrie, bon compromis	-	BERT-base, DistilBERT, usage général
1024-1536	Haute précision, SOTA benchmarks	Plus lourd (+30% latence vs 768)	BERT-large, RoBERTa, embeddings OpenAI
4096-12288	Capacité maximale, modèles géants	Coût prohibitif, nécessite GPU puissant	GPT-3, GPT-4, recherche académique

Impact sur les performances :

```
# Comparaison mémoire et latence
import numpy as np

vocab_size = 50000
sequence_length = 512
batch_size = 32

for dim in [128, 384, 768, 1536, 4096]:
    # Mémoire matrice d'embeddings
    embedding_matrix_mb = (vocab_size * dim * 4) / (1024**2)  # float32

    # Mémoire batch en forward pass
    batch_mb = (batch_size * sequence_length * dim * 4) / (1024**2)

    # Latence relative (approximative)
    latency_factor = dim / 768  # Normalisé à 768

    print(f"\nDimension: {dim}")
    print(f"  Matrice embeddings: {embedding_matrix_mb:.1f} MB")
    print(f"  Batch mémoire: {batch_mb:.1f} MB")
    print(f"  Latence relative: {latency_factor:.2f}x")

# Output:
# Dimension: 128  -> 24.4 MB, latency 0.17x
# Dimension: 768  -> 146.5 MB, latency 1.00x (baseline)
# Dimension: 1536 -> 293.0 MB, latency 2.00x
# Dimension: 4096 -> 781.2 MB, latency 5.33x
```

Recommandation pratique

Pour RAG / Recherche vectorielle :
- Prototypage : 384 dim (all-MiniLM-L6-v2)
- Production standard : 768-1024 dim (bge-base, e5-base)
- Haute précision : 1536 dim (text-embedding-3-large)

Trade-off : +100% dimension = +30-50% latence, +5-10% recall

Coût computationnel

Tokens et embeddings ont des coûts très différents en termes de calcul et de stockage.

Coût de stockage

```
# Comparaison stockage pour 1M de documents
import numpy as np

num_documents = 1_000_000
avg_tokens_per_doc = 500

# Stockage TOKENS (IDs)
token_storage_mb = (num_documents * avg_tokens_per_doc * 2) / (1024**2)  # int16
print(f"Stockage tokens (1M docs): {token_storage_mb:.1f} MB (~{token_storage_mb/1024:.2f}
GB)")

# Stockage EMBEDDINGS (vecteurs 768D)
embedding_dim = 768
embedding_storage_gb = (num_documents * embedding_dim * 4) / (1024**3)  # float32
print(f"Stockage embeddings (1M docs): {embedding_storage_gb:.2f} GB")

print(f"\nRatio: Embeddings = {embedding_storage_gb / (token_storage_mb/1024):.0f}x plus
lourd")

# Output:
# Stockage tokens: 953.7 MB (~0.93 GB)
# Stockage embeddings: 2.86 GB
# Ratio: Embeddings = 3x plus lourd
```

Coût d'inférence (latence)

Opération	Complexité	Latence typique
Tokenisation (texte → tokens)	O(n) n=longueur texte	0.1-1 ms (CPU)
Embedding lookup (tokens → embeddings)	O(1) par token	0.01-0.1 ms (GPU)
Transformer forward (embeddings → contextuels)	$O(n^2)$ pour self-attention	10-100 ms (GPU, seq_len=512)
Recherche vectorielle (ANN avec HNSW)	O(log n) n=taille base	5-50 ms (1M-10M vecteurs)

Coût API (pricing réel) :

```
# Pricing OpenAI (Janvier 2025)
import tiktoken

enc = tiktoken.encoding_for_model("gpt-4")

text = """L'intelligence artificielle transforme radicalement le paysage économique
mondial.
Les entreprises investissent massivement dans les LLMs pour automatiser leurs
processus."""

tokens = enc.encode(text)
num_tokens = len(tokens)

# Pricing GPT-4 Turbo (exemple)
input_price_per_1k = 0.01  # $0.01 / 1K tokens
output_price_per_1k = 0.03  # $0.03 / 1K tokens

input_cost = (num_tokens / 1000) * input_price_per_1k
print(f"Texte: {len(text)} caractères, {num_tokens} tokens")
print(f"Coût input: ${input_cost:.6f}")

# Si génération de 200 tokens en output
output_tokens = 200
output_cost = (output_tokens / 1000) * output_price_per_1k
total_cost = input_cost + output_cost

print(f"Coût output (200 tokens): ${output_cost:.6f}")
print(f"Coût total: ${total_cost:.6f}")

# À l'échelle :
monthly_requests = 1_000_000
monthly_cost = total_cost * monthly_requests
print(f"\nCoût mensuel (1M requêtes): ${monthly_cost:,.2f}")
```

Multilinguisme

Les tokens et embeddings multilingues présentent des défis spécifiques liés à la diversité des langues.

Problème : Inefficacité de tokenisation multilingue

Les tokenizers entraînés sur corpus anglophones (GPT-2, GPT-3 initiaux) sont **très inefficaces** pour d'autres langues :

```
import tiktoken

enc_gpt4 = tiktoken.encoding_for_model("gpt-4")

# Comparaison anglais vs autres langues
texts = {
    "English": "Artificial intelligence transforms businesses.",
    "French": "L'intelligence artificielle transforme les entreprises.",
    "German": "Künstliche Intelligenz transformiert Unternehmen.",
    "Japanese": "人工知能はビジネスを変える。",
    "Arabic": "يحول الذكاء الاصطناعي الأعمال التجارية."
}

print("Efficacité de tokenisation par langue:\n")
for lang, text in texts.items():
    tokens = enc_gpt4.encode(text)
    chars = len(text)
    ratio = chars / len(tokens)

    print(f"{lang:12s}: {len(tokens):3d} tokens, {chars:3d} chars, ratio: {ratio:.2f}
chars/token")

# Output typique :
# English     :   7 tokens,  48 chars, ratio: 6.86 chars/token  <- Efficace
# French      :  10 tokens,  57 chars, ratio: 5.70 chars/token
# German      :  11 tokens,  53 chars, ratio: 4.82 chars/token
# Japanese    :  22 tokens,  18 chars, ratio: 0.82 chars/token  <- Très inefficace
# Arabic      :  18 tokens,  44 chars, ratio: 2.44 chars/token

print("\n⚠️ Le japonais nécessite 3x plus de tokens que l'anglais pour le même contenu !")
print("Impact : Coût API 3x supérieur, limite de contexte atteinte plus vite.")
```

Solutions multilingues

- **XLM-RoBERTa** : Tokenizer et modèle entraînés sur 100 langues, vocabulaire de 250K tokens
- **mBERT** (Multilingual BERT) : 104 langues, vocabulaire partagé de 110K tokens
- **LLaMA 2** : Tokenizer SentencePiece avec meilleure couverture non-anglophone
- **GPT-4** : Amélioration significative du tokenizer multilingue vs GPT-3

```
from transformers import AutoTokenizer

# Tokenizer multilingue optimisé
tokenizer_multi = AutoTokenizer.from_pretrained('xlm-roberta-base')

text_fr = "L'intelligence artificielle transforme les entreprises."
text_ja = "人工知能はビジネスを変える。"

tokens_fr = tokenizer_multi.tokenize(text_fr)
tokens_ja = tokenizer_multi.tokenize(text_ja)

print(f"Français: {len(tokens_fr)} tokens - {tokens_fr}")
print(f"Japonais: {len(tokens_ja)} tokens - {tokens_ja}")

# XLM-RoBERTa est beaucoup plus efficace pour les langues non-latines
print(f"\nRatio amélioration: {22 / len(tokens_ja):.1f}x moins de tokens vs GPT-4")
```

Evolution et tendances futures

Tokenisation sans vocabulaire fixe

Les recherches actuelles explorent des alternatives à la tokenisation traditionnelle avec vocabulaire fixe.

ByT5 (Byte-level T5, Google 2021)
Modèle qui opère directement sur les **bytes UTF-8** (vocabulaire de 256 tokens seulement) au lieu de sous-mots.

- **Avantages** : Zéro OOV, parfaitement multilingue, robuste aux fautes
- **Inconvénients** : Séquences 3-4x plus longues, coût computationnel accru
- **Statut** : Recherche active, pas encore adopté en production massive

Charformer (Google 2021)
Architecture hybride qui tokenise au niveau caractère puis utilise des blocs de "gradient-based subword tokenization" apprises dynamiquement.

Embeddings adaptatifs

Les embeddings adaptatifs ajustent dynamiquement la capacité représentationnelle selon la fréquence des tokens.

Principe Pour approfondir, consultez Prompt Engineering Avancé : Chain-of-Thought et Techniques.

Observation : Les tokens fréquents ("le", "de", "est") nécessitent moins de capacité que les tokens rares ("anticonstitutionnellement"). Les embeddings adaptatifs allouent plus de dimensions aux tokens rares.

Implémentation (Adaptive Input Representations, Baevski & Auli 2019) :

```
# Concept simplifié
# Tokens fréquents (0-10K)     : 128 dimensions
# Tokens moyens (10K-30K)       : 256 dimensions
# Tokens rares (30K-50K)        : 512 dimensions

# Projection finale vers dimension uniforme (768) pour les Transformers
# Économie : 30-50% de réduction paramètres embeddings
```

Utilisé dans : Transformer-XL, certaines variantes de BERT optimisées

Modèles multimodaux

Les modèles multimodaux (CLIP, GPT-4V, Gemini) unifient tokens textuels et embeddings visuels dans un espace partagé.

CLIP (OpenAI, 2021)
CLIP apprend un espace d'embeddings commun pour texte et images via contrastive learning.

```
from transformers import CLIPProcessor, CLIPModel
from PIL import Image
import requests

model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")

# Image
url = ""
image = Image.open(requests.get(url, stream=True).raw)

# Textes candidats
texts = ["un chat", "un chien", "une voiture"]

# Encodage
inputs = processor(text=texts, images=image, return_tensors="pt", padding=True)
outputs = model(**inputs)

# Similarité image-texte
logits_per_image = outputs.logits_per_image
probs = logits_per_image.softmax(dim=1)

for text, prob in zip(texts, probs[0]):
    print(f"{text:20s}: {prob.item():.3f}")

# Output:
# un chat             : 0.952  <- Match correct
# un chien            : 0.045
# une voiture         : 0.003
```

Architecture :

- **Texte** : Tokenisation BPE → Token embeddings → Transformer text encoder → Embedding 512D
- **Image** : Patches visuels → Vision Transformer → Embedding 512D
- **Alignement** : Les deux embeddings sont projetés dans le même espace sémantique

GPT-4 Vision, Gemini
Génération suivante : tokens textuels et "tokens visuels" (patches d'images) traités conjointement dans le même Transformer.

Vers des représentations universelles

L'objectif ultime : des embeddings universels capturant tout type de données (texte, image, audio, vidéo, code) dans un espace sémantique unifié.

Tendances 2025

- **ImageBind (Meta, 2023)** : 6 modalités alignées (image, texte, audio, profondeur, thermique, IMU)
- **Universal Speech Model (USM, Google)** : 300+ langues, embeddings audio universels
- **Embeddings de code** : CodeBERT, GraphCodeBERT pour recherche sémantique de code
- **Embeddings 3D** : PointBERT pour nuages de points, applications robotique/AR

Vision futur (2025-2030) :

- Embeddings zero-shot pour nouvelles modalités (haptic, olfactif ?)
- Compression extrême : Matryoshka embeddings (dimensions emboîtées, pertes minimales)
- Embeddings dynamiques : Dimension adaptée automatiquement selon complexité de la requête
- Fin de la tokenisation ? : Modèles opérant directement sur bytes/pixels bruts

Sources et références : ArXiv IA · Hugging Face Papers

Questions fréquentes

Un token peut-il avoir plusieurs embeddings ?

Oui, et c'est fondamental dans les modèles modernes.

- **Embedding statique (couche 0)** : Chaque token a UN seul embedding initial dans la matrice [vocab_size, hidden_dim]
- **Embeddings contextuels (après Transformers)** : Le même token aura des embeddings **différents** selon son contexte. Exemple : "banque" dans "banque de données" vs "banque financière" aura des embeddings finaux très distincts après passage dans BERT.

C'est la force des Transformers : capturer la polysémie et le contexte.

Pourquoi les modèles ont-ils des limites de tokens ?

Pour des raisons de complexité computationnelle et mémoire.

- **Self-attention = $O(n^2)$** : Doubler le contexte (512 → 1024 tokens) **quadruple** le coût de calcul et mémoire
- **Mémoire GPU** : Un batch de 8 séquences de 2048 tokens avec GPT-3 nécessite ~40-80 GB VRAM
- **Latence utilisateur** : Contextes longs = inférence plus lente (100K tokens = plusieurs secondes même sur A100)

Solutions en 2025 : Flash Attention, sparse attention, modèles linéaires (Mamba, RWKV) réduisent à $O(n)$, permettant 100K-1M+ tokens (Claude 3, Gemini 1.5).

Peut-on créer nos propres embeddings ?

Oui, trois approches possibles :

1. **From scratch (déconseillé)** : Entraîner un modèle complet (BERT, GPT) sur votre corpus. Coût : $100K-$1M+, nécessite GPU cluster. Rarement justifié sauf domaines ultra-spécialisés (médical, légal).
2. **Fine-tuning (recommandé)** : Adapter un modèle pré-entraîné (Sentence-BERT) sur vos données avec contrastive learning. Coût : $100-$5K, faisable sur 1-4 GPUs. Gains : +5-15% recall sur votre domaine.

3. **Domain-specific tokenizer** : Entraîner un tokenizer BPE sur votre corpus technique, puis fine-tuner. Utile si vocabulaire très spécialisé (code, chimie).

Outils : Sentence-Transformers, Hugging Face Trainer, OpenAI fine-tuning API.

Les embeddings sont-ils toujours meilleurs que les tokens one-hot ?

Oui, en pratique toujours pour le NLP moderne.

Encodage one-hot (ancien) : Vecteur sparse de taille [vocab_size] avec un seul 1. Exemple : token ID 5432 → [0, 0, ..., 1, ..., 0] (50K dimensions).

Problèmes one-hot :

- ❌ Aucune similarité capturée : "roi" et "reine" sont orthogonaux (similarité = 0)
- ❌ Mémoire prohibitive : 50K dimensions vs 768 pour embeddings
- ❌ Curse of dimensionality : Performances désastreuses avec haute dimension

Embeddings denses : Capturent relations sémantiques, compacts, performants.

Seul cas one-hot viable : Classification simple avec vocabulaire minuscule (<100 classes) et sans besoin de sémantique.

Comment gérer les tokens rares ou spécialisés ?

Plusieurs stratégies selon le cas :

1. Tokens techniques/métier (ex: "SIEM", "Kubernetes", "pgvector")

- **Avec BPE** : Décomposés en sous-mots, sens partiellement préservé
- **Fine-tuning** : Adapter le modèle sur corpus technique pour améliorer embeddings
- **Tokenizer custom** : Ajouter tokens spéciaux au vocabulaire (via `tokenizer.add_tokens()`)

2. Noms propres, entités

- Les modèles récents (GPT-4, Claude) gèrent bien via BPE
- Pour métier : Utiliser NER + entity linking vers base de connaissances

3. Code, formules mathématiques

- Modèles spécialisés : CodeBERT, CodeT5, StarCoder pour code
- LaTeX-aware tokenizers pour math

4. Multilinguisme

- Utiliser XLM-RoBERTa, mBERT ou modèles multilingues natifs
- Éviter GPT-3 (inefficace hors anglais), préférer GPT-4 ou LLaMA 2

```python
# Exemple : Ajouter tokens spécialisés
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')

# Tokens métier à ajouter
special_tokens = ["[RAG]", "[EMBEDDING]", "[HNSW]", "pgvector"]
num_added = tokenizer.add_tokens(special_tokens)

print(f"Tokens ajoutés: {num_added}")
print(f"Nouveau vocabulaire: {len(tokenizer)} tokens")

# Redimensionner la matrice d'embeddings du modèle
model.resize_token_embeddings(len(tokenizer))

# Fine-tuner pour apprendre les nouveaux embeddings
# (nécessite corpus avec ces tokens)
```

Ressources open source associées :

- CUDAEmbeddings — Serveur d'embeddings GPU (Python)
- llm-finetuning-fr — Dataset fine-tuning LLM (HuggingFace)

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr

© 2025 — Reproduction interdite sans autorisation.