AYI NEDJIMI Consultants

Vecteurs en Intelligence Artificielle : Guide Complet

Catégorie : Intelligence Artificielle Lecture : 23 min Publié le : 07/12/2025 Auteur : Ayi NEDJIMI

Découvrez comment les vecteurs sont utilisés en intelligence artificielle pour représenter données textuelles, images et audio. Guide complet avec...

Définition mathématique

Un **vecteur** est une structure mathématique représentée par une séquence ordonnée de nombres réels, notée **$\mathbf{v = [v_1, v_2, ..., v_n]}$**, où n est la **dimension** du vecteur. En intelligence artificielle, les vecteurs encodent l'information sous forme numérique permettant aux algorithmes de traiter, comparer et manipuler les données. Découvrez comment les vecteurs sont utilisés en intelligence artificielle pour représenter données textuelles, images et audio. Guide complet avec... Dans un contexte où l'intelligence artificielle transforme les pratiques de cybersécurité, la maîtrise de ia vecteurs intelligence artificielle devient un avantage stratégique pour les équipes techniques. Nous abordons notamment : qu'est-ce qu'un vecteur en ia ?, les représentations vectorielles en pratique et comprendre les dimensions vectorielles. Les professionnels y trouveront des recommandations actionnables, des commandes prêtes à l'emploi et des stratégies de mise en œuvre adaptées aux environnements d'entreprise.

Formellement, un vecteur à n dimensions appartient à l'espace vectoriel $\mathbb{R}^n$. Par exemple :

- **Vecteur 2D** : v = [3.5, -2.1] ∈ $\mathbb{R}^2$
- **Vecteur 3D** : v = [1.0, 0.5, -0.8] ∈ $\mathbb{R}^3$
- **Vecteur haute dimension** : v = [0.123, -0.456, ..., 0.789] ∈ $\mathbb{R}^{768}$ (typique pour les embeddings textuels)

Chaque composante v_k du vecteur représente une caractéristique ou dimension de l'information encodée. La **magnitude** (ou norme) d'un vecteur mesure sa "longueur" : `||v|| = √(v₁² + v₂² + ... + vₙ²)`.

Exemple Python : Création de vecteurs

```
import numpy as np

# Vecteur 2D simple
vec_2d = np.array([3.5, -2.1])

# Vecteur haute dimension (768D comme OpenAI text-embedding-ada-002)
vec_embedding = np.random.randn(768)

# Calcul de la norme (magnitude)
norme = np.linalg.norm(vec_2d)  # = 4.08
print(f"Norme du vecteur : {norme:.2f}")

# Normalisation (ramener à norme = 1)
vec_normalized = vec_2d / norme
print(f"Vecteur normalisé : {vec_normalized}")  # [0.86, -0.51]
print(f"Nouvelle norme : {np.linalg.norm(vec_normalized):.2f}")  # 1.00
```

Du concept mathématique à l'application IA

En intelligence artificielle, les vecteurs transcendent leur définition mathématique pour devenir le **langage universel** permettant aux machines de représenter et manipuler n'importe quel type d'information : texte, images, sons, vidéos, comportements utilisateurs, etc.

Cette transformation s'appelle **vectorisation** ou **embedding** : convertir des données brutes en vecteurs numériques qui capturent leur **sémantique** (sens) plutôt que leur forme syntaxique.

Type de données	Avant (format brut)	Après (vecteur)	Dimension typique
Texte	"Intelligence artificielle"	[0.23, -0.45, 0.12, ..., 0.67]	384-1536
Image	chat.jpg (3MB, 1920x1080)	[0.89, 0.34, -0.12, ..., 0.56]	512-2048
Audio	voix.mp3 (30 secondes)	[0.45, -0.23, 0.78, ..., -0.11]	768-1024
Utilisateur	Historique achats, clics, pages vues	[0.12, 0.89, -0.34, ..., 0.23]	64-256

La magie des vecteurs : Deux contenus sémantiquement similaires auront des vecteurs **proches dans l'espace vectoriel**, même s'ils utilisent des mots différents. Par exemple, "voiture" et "automobile" auront des vecteurs très similaires, alors que "voiture" et "banane" seront très éloignés.

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

Pourquoi les vecteurs sont-ils essentiels en IA ?

Les vecteurs sont essentiel à l'IA moderne pour quatre raisons fondamentales :

1. Traitement mathématique uniforme
Une fois les données converties en vecteurs, on peut appliquer des **opérations mathématiques standardisées** : calcul de similarité, classification, clustering, réduction de dimension. Cela permet aux algorithmes de machine learning de fonctionner indépendamment du type de données source.

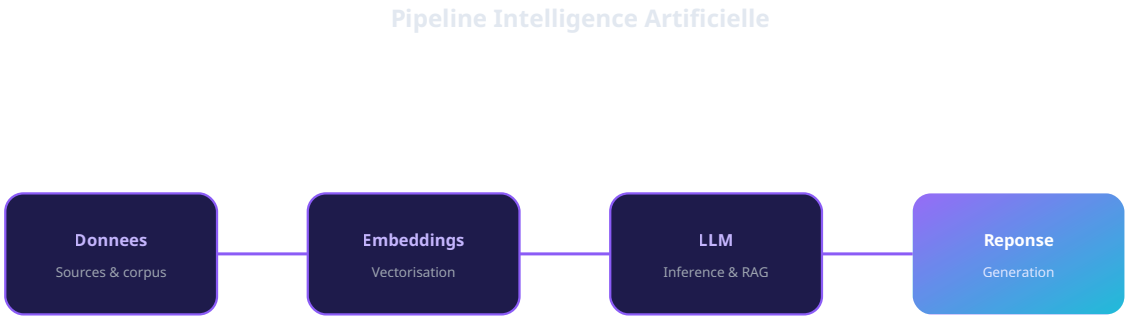
2. Capture de la sémantique
Les vecteurs modernes (embeddings) encodent le **sens contextuel** des données. Par exemple, dans Word2Vec : `vecteur("Roi") - vecteur("Homme") + vecteur("Femme") ≈ vecteur("Reine")`. Cette propriété transformateur permet aux machines de comprendre les relations conceptuelles.

3. Calcul de similarité efficace
Comparer deux vecteurs est **rapide** : un simple produit scalaire ou calcul de distance euclidienne. Cela permet des recherches sémantiques sur des millions de documents en quelques millisecondes avec des structures d'indexation appropriées (HNSW, IVF).

4. Compatibilité avec les réseaux de neurones
Les réseaux de neurones ne peuvent traiter que des **nombres**. Les vecteurs sont donc la représentation intermédiaire obligatoire entre données brutes et modèles d'IA. Chaque couche d'un réseau de neurones transforme les vecteurs d'entrée en vecteurs de sortie de plus en plus abstraits.

Impact Concret en Production

- **Recherche Google** : Transforme votre requête en vecteur et compare avec des milliards de pages indexées
- **Netflix/Spotify** : Vecteurs utilisateurs × vecteurs contenus = recommandations personnalisées
- **ChatGPT** : Chaque mot du contexte converti en vecteur (token embedding) avant traitement
- **Reconnaissance faciale** : Visage → vecteur 128D unique → comparaison instantanée



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

Notre avis d'expert
La gouvernance de l'IA est le prochain grand chantier de la cybersécurité. Les attaques par prompt injection, l'empoisonnement de données d'entraînement et l'extraction de modèles sont des menaces concrètes que nous observons de plus en plus lors de nos missions. Ne pas s'y préparer, c'est accepter un risque majeur.

Les représentations vectorielles en pratique

Représentation du texte en vecteurs

La vectorisation du texte a connu plusieurs évolutions majeures, chacune capturant de plus en plus de sémantique.

Approche 1 : One-Hot Encoding (obsolète)
Chaque mot = vecteur de taille V (vocabulaire) avec un seul 1. Exemple avec vocabulaire {chat, chien, voiture} :

- "chat" = [1, 0, 0]
- "chien" = [0, 1, 0]
- "voiture" = [0, 0, 1]

Problème : Tous les mots sont équidistants (pas de notion de similarité), dimension explosive (V = 50K-1M pour langues naturelles).

Approche 2 : TF-IDF (classique)
Représente un document par un vecteur pondéré basé sur la fréquence des mots :

- **TF (Term Frequency)** : Fréquence du mot dans le document
- **IDF (Inverse Document Frequency)** : Importance du mot dans le corpus

Usage : Moteurs de recherche classiques, classification de documents simples.

Approche 3 : Word Embeddings (Word2Vec, GloVe) - 2013-2015
Révolution : chaque mot = vecteur dense (50-300D) entraîné pour capturer le contexte. "Chien" et "chat" ont des vecteurs similaires car utilisés dans des contextes similaires.

Exemple Word2Vec

```
from gensim.models import Word2Vec

# Corpus d'entraînement
sentences = [
    ["le", "chat", "mange", "des", "croquettes"],
    ["le", "chien", "mange", "des", "croquettes"],
    ["la", "voiture", "roule", "vite"]
]

# Entraînement Word2Vec
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1)

# Obtenir le vecteur d'un mot
vec_chat = model.wv['chat']  # Array 100D

# Trouver mots similaires
similaires = model.wv.most_similar('chat', topn=3)
print(similaires)  # [('chien', 0.92), ('mange', 0.65), ...]

# Arithmétique vectorielle
vec_result = model.wv['roi'] - model.wv['homme'] + model.wv['femme']
print(model.wv.similar_by_vector(vec_result)[0])  # 'reine'
```

Approche 4 : Contextuels (BERT, GPT) - 2018-2025
Rupture : Le même mot a des vecteurs **différents selon le contexte**.

- "La **banque** du fleuve" → $vecteur_1$
- "Ma **banque** en ligne" → $vecteur_2$

Modèles actuels : OpenAI text-embedding-3 (1536D), BGE-M3 (1024D), Mistral Embed (1024D) atteignent 85-90% de précision sur benchmarks MTEB.

Exemple OpenAI Embeddings

```
from openai import OpenAI
import numpy as np

client = OpenAI(api_key="sk-...")

# Générer embeddings
textes = [
    "Le machine learning bouleverse l'IA",
    "L'apprentissage automatique transforme l'intelligence artificielle",
    "J'aime les pizzas margherita"
]

response = client.embeddings.create(
    model="text-embedding-3-small",  # 1536 dimensions
    input=textes
)

vectors = [emb.embedding for emb in response.data]

# Calcul similarité cosinus
def cosine_similarity(v1, v2):
    return np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))

print(f"Sim(texte1, texte2): {cosine_similarity(vectors[0], vectors[1]):.3f}")  # 0.92
print(f"Sim(texte1, texte3): {cosine_similarity(vectors[0], vectors[2]):.3f}")  # 0.15
```

Représentation des images

Les images sont converties en vecteurs via des **réseaux de neurones convolutifs (CNN)** qui extraient progressivement des caractéristiques de bas niveau (contours, couleurs) puis haut niveau (objets, scènes).

Pipeline de vectorisation d'image

1. **Pixels bruts** : Image 224×224×3 (RGB) = 150 528 valeurs
2. **Couches convolutives** : Extraction de features (ResNet, VGG, EfficientNet)
3. **Couche finale (embedding layer)** : Vecteur dense 512-2048D capturant le contenu sémantique

Exemple avec ResNet50 (PyTorch)

```
import torch
import torchvision.models as models
import torchvision.transforms as transforms
from PIL import Image

# Charger ResNet50 pré-entraîné
model = models.resnet50(pretrained=True)
model = torch.nn.Sequential(*list(model.children())[:-1])  # Retirer la couche
classification
model.eval()

# Preprocessing
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Convertir image en vecteur
image = Image.open('chat.jpg')
image_tensor = transform(image).unsqueeze(0)

with torch.no_grad():
    embedding = model(image_tensor)
    embedding = embedding.squeeze().numpy()  # Shape: (2048,)

print(f"Vecteur image : {embedding.shape}")  # (2048,)
print(f"Norme : {np.linalg.norm(embedding):.2f}")
```

Applications concrètes :

Cas concret
L'attaque par prompt injection sur les systèmes GPT documentée par OWASP en 2023 a révélé que des instructions malveillantes dissimulées dans des documents pouvaient détourner le comportement de chatbots d'entreprise, accédant à des données internes sensibles sans aucune authentification supplémentaire.

Vos pipelines de données d'entraînement sont-ils protégés contre l'empoisonnement ?

- **Recherche d'images similaires** : Pinterest, Google Images
- **Reconnaissance faciale** : FaceNet génère vecteurs 128D uniques par visage
- **Détection de duplicatas** : Distance < seuil → images identiques/similaires
- **Classification** : Vecteur → classifieur → catégorie (chat, chien, voiture, ...)

Attention : Modèles multimodaux

CLIP (OpenAI) et **LLaVA** encodent images ET textes dans le **même espace vectoriel**, permettant de chercher des images avec du texte : "chat qui dort" → vecteur texte → recherche vecteurs images similaires.

Représentation de l'audio

L'audio est vectorisé via des modèles spécialisés qui capturent le **contenu acoustique** (parole, musique, bruit) et le **contenu sémantique** (ce qui est dit).

Approches de vectorisation audio

1. Caractéristiques acoustiques classiques

- **MFCC** (Mel-Frequency Cepstral Coefficients) : 13-40 coefficients par trame (25ms)
- **Spectrogrammes** : Représentation temps-fréquence convertie en vecteur
- **Usage** : Classification de sons simples, détection d'événements

2. Embeddings par réseaux de neurones

- **Wav2Vec 2.0** (Meta) : Vecteurs 768-1024D capturant la sémantique de la parole
- **Whisper** (OpenAI) : Transcription + embeddings pour recherche vocale
- **MusicGen** : Vecteurs musicaux pour similarité de morceaux

Exemple avec Wav2Vec2

```
from transformers import Wav2Vec2Processor, Wav2Vec2Model
import librosa
import torch

# Charger le modèle
processor = Wav2Vec2Processor.from_pretrained("facebook/wav2vec2-base")
model = Wav2Vec2Model.from_pretrained("facebook/wav2vec2-base")

# Charger fichier audio
audio, sr = librosa.load('voix.wav', sr=16000)
inputs = processor(audio, sampling_rate=16000, return_tensors="pt")

# Générer embedding
with torch.no_grad():
    outputs = model(**inputs)
    # Moyenne des embeddings temporels pour obtenir un vecteur global
    embedding = outputs.last_hidden_state.mean(dim=1).squeeze().numpy()

print(f"Vecteur audio : {embedding.shape}")  # (768,)

# Comparaison de similarité pour détection de locuteur
audio2, _ = librosa.load('voix2.wav', sr=16000)
inputs2 = processor(audio2, sampling_rate=16000, return_tensors="pt")
with torch.no_grad():
    embedding2 = model(**inputs2).last_hidden_state.mean(dim=1).squeeze().numpy()

similarity = np.dot(embedding, embedding2) / (np.linalg.norm(embedding) *
np.linalg.norm(embedding2))
print(f"Similarité vocale : {similarity:.3f}")  # > 0.9 = même locuteur
```

Cas d'usage audio :

- **Identification de locuteur** : Banques, call centers (vérification d'identité)
- **Recherche de musique similaire** : Shazam, Spotify recommendations
- **Détection d'anomalies sonores** : Maintenance prédictive en usine

- **Classification de sons** : Alarmes, animaux, événements urbains

Représentation multi-modale

Les modèles **multi-modaux** (2021-2025) représentent plusieurs types de données (texte, image, audio, vidéo) dans un **espace vectoriel unifié**. Cette avancée majeure permet de combiner et comparer des modalités différentes.

Principe : Espace latent partagé
Au lieu d'avoir des vecteurs incomparables (vecteur_texte ≠ vecteur_image), les modèles multi-modaux apprennent à projeter toutes les modalités dans le **même espace**, où la similarité sémantique est préservée quel que soit le type de données source.

Modèle	Modalités	Dimension	Usage Principal
CLIP (OpenAI, 2021)	Texte + Image	512-768	Recherche image par texte, classification zero-shot
ImageBind (Meta, 2023)	6 modalités (texte, image, audio, vidéo, profondeur, IMU)	1024	Recherche cross-modale généralisée
LLaVA (2023)	Texte + Image	4096	Vision-Language conversation
Gemini (Google, 2024)	Texte + Image + Audio + Vidéo	Non publié	Assistant multimodal général

Cas d'usage changants

- **Recherche cross-modale** : Chercher une image avec du texte, ou du texte avec une image
- **Génération conditionnelle** : "Génère une image représentant ce morceau de musique"
- **Traduction cross-modale** : Audio → Texte → Image automatiquement
- **Détection d'incohérences** : Vérifier qu'une image correspond bien à sa description textuelle

Exemple CLIP : Recherche image par texte

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

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

# Images à chercher
images = [Image.open(f"image{i}.jpg") for i in range(1, 6)]

# Requête textuelle
textes = ["un chat qui dort", "un chien qui court", "une voiture rouge"]

# Encoder images et textes dans le même espace
inputs = processor(text=textes, images=images, return_tensors="pt", padding=True)
outputs = model(**inputs)

# Extraire embeddings
image_embeds = outputs.image_embeds  # Shape: (5, 512)
text_embeds = outputs.text_embeds    # Shape: (3, 512)

# Calcul similarité : chaque texte vs chaque image
similarity = torch.matmul(text_embeds, image_embeds.T)
print(similarity)  # Matrix (3, 5) : scores de similarité

# Pour chaque texte, trouver l'image la plus similaire
for i, texte in enumerate(textes):
    best_image_idx = similarity[i].argmax().item()
    print(f"{texte} → image{best_image_idx + 1}.jpg (score: {similarity[i,
best_image_idx]:.2f})")
```

Comprendre les dimensions vectorielles

Qu'est-ce qu'une dimension ?

En termes simples, la **dimension d'un vecteur** correspond au **nombre de composantes** (valeurs numériques) qu'il contient. Chaque dimension représente une **caractéristique ou un axe** dans l'espace vectoriel.

Géométriquement, la dimension définit l'espace dans lequel le vecteur existe :

- **1 dimension (1D)** : Une ligne droite (position sur un axe)
- **2 dimensions (2D)** : Un plan (position sur axes X et Y)
- **3 dimensions (3D)** : Un volume (position X, Y, Z dans l'espace physique)
- **n dimensions (nD)** : Espace mathématique abstrait impossible à visualiser

Interprétation conceptuelle

En IA, chaque dimension capture une **caractéristique latente** (souvent non-interprétable directement). Pour un embedding de texte 768D, on ne sait pas exactement ce que représente la dimension 234, mais l'ensemble des 768 dimensions encode le sens sémantique complet du texte.

Exemple : Vecteurs de dimensions variables

```
import numpy as np

# Vecteur 1D : température simple
temp = np.array([23.5])  # 1 dimension

# Vecteur 2D : coordonnées GPS
location = np.array([48.8566, 2.3522])  # [latitude, longitude]

# Vecteur 3D : couleur RGB
couleur = np.array([255, 128, 0])  # [Rouge, Vert, Bleu]

# Vecteur 50D : profil utilisateur (simplifié)
user_profile = np.random.randn(50)  # âge, genre, préférences, historique...

# Vecteur 768D : embedding textuel (BERT)
text_embedding = np.random.randn(768)  # Capture sémantique complète

print(f"Dimensions : {temp.shape}, {location.shape}, {couleur.shape},
{user_profile.shape}, {text_embedding.shape}")
# Output: (1,) (2,) (3,) (50,) (768,)
```

Vecteurs 2D, 3D et au-delà

Les vecteurs de basse dimension (2D-3D) sont faciles à visualiser et souvent utilisés pour des données simples, mais l'IA moderne repose sur des **vecteurs de haute dimension** (64-2048D) qui capturent des informations complexes.

Vecteurs 2D : Visualisation et cas simples
Exemples concrets : coordonnées géographiques, prix/surface immobilière, âge/salaire.

- **Avantage** : Visualisation directe sur un graphique X-Y
- **Limite** : Capture très peu d'information, rarement suffisant pour des tâches IA complexes

Vecteurs 3D : Géométrie et physique
Exemples : positions 3D dans l'espace, couleurs RGB, vitesse (vx, vy, vz).

- **Usage IA** : Robotique (positions, orientations), graphisme 3D, modèles physiques
- **Visualisation** : Possible avec outils 3D interactifs

Haute dimension (64-2048D) : Lère de l'IA moderne
Pourquoi tant de dimensions ? Les données réelles (texte, images, comportements) sont **intrinsèquement complexes** et nécessitent de nombreuses dimensions pour capturer toutes leurs nuances sémantiques.

Type de données	Dimensions typiques	Raison
Word2Vec	100-300	Vocabulaire 50K+ mots projeté dans espace réduit
BERT/GPT	768-1024	Contexte linguistique riche, relations complexes
OpenAI text-embedding-3	1536-3072	Précision maximale, capture nuances fines
Images (ResNet)	512-2048	Représentation hiérarchique : textures → objets → scènes
Utilisateurs (recommandation)	64-256	Préférences multiples, comportements variés

Trade-off dimensions

Plus de dimensions = plus d'expressivité (meilleure précision) MAIS aussi plus de coûts (mémoire, calcul, temps d'entraînement). En production, on cherche le **sweet spot** : minimum de dimensions pour atteindre la performance cible.

Haute dimensionnalité : avantages et défis

Avantages de la haute dimension

1. Expressivité accrue

Plus de dimensions permettent de capturer des **relations complexes et nuancées**. Par exemple, différencier "j'aime ce film" (positif) de "j'aime bien ce film" (neutre/tiede) nécessite des dimensions supplémentaires pour encoder l'intensité émotionnelle.

2. Séparation linéaire

En haute dimension, des données non-séparables en basse dimension deviennent **linéairement séparables**. C'est le principe du "kernel trick" en SVM : projeter dans un espace de dimension supérieure facilite la classification.

3. Réduction d'information

Un vecteur 1536D semble énorme, mais c'est une **compression massive** comparé aux données brutes : 1 page de texte (5000 caractères) → 1536 float32 (6KB). Une image 1920×1080 (6MB) → 2048 float32 (8KB).

Défis de la haute dimension

1. Coût computationnel

- **Mémoire** : 1M vecteurs 1536D float32 = 6GB RAM (vs 400MB pour 64D)
- **Calcul de distance** : O(d) où d = dimensions. 1536D = 24x plus lent que 64D
- **Indexation** : Structures HNSW/IVF plus volumineuses et coûteuses

2. Overfitting

En machine learning, trop de dimensions par rapport au nombre d'exemples d'entraînement conduit à un modèle qui mémorise plutôt que généraliser. **Règle empirique** : avoir au moins 10x plus d'exemples que de dimensions.

3. Difficulté d'interprétation

Contrairement à des features explicites ("prix", "âge"), les dimensions d'embeddings sont des **features latentes** non-interprétables. Impossible de dire "la dimension 247 représente X".

Techniques de réduction de dimension

```
from sklearn.decomposition import PCA
import numpy as np

# Dataset : 1000 vecteurs de 1536D
high_dim_vectors = np.random.randn(1000, 1536)

# PCA : réduire à 128D en préservant 95% de la variance
pca = PCA(n_components=128)
low_dim_vectors = pca.fit_transform(high_dim_vectors)

print(f"Dimensions avant : {high_dim_vectors.shape}")  # (1000, 1536)
print(f"Dimensions après : {low_dim_vectors.shape}")   # (1000, 128)
print(f"Variance expliquée : {pca.explained_variance_ratio_.sum():.2%}")  # 95%

# Réduction mémoire : 1536 → 128 = 12x moins de RAM
print(f"Réduction mémoire : {1536/128:.1f}x")
```

La malédiction de la dimension

La **"curse of dimensionality"** (malédiction de la dimension) est un phénomène contre-intuitif : en haute dimension, les vecteurs deviennent **presque tous équidistants**, rendant les notions de "proche" et "loin" moins significatives.

Phénomène mathématique
En haute dimension, le **volume de l'espace explose** exponentiellement. Dans un hypercube unité $[0,1]^n$, la quasi-totalité du volume se concentre dans les "coins" (zones éloignées du centre).

- **2D** : Aire cercle / aire carré = 78.5%
- **3D** : Volume sphère / volume cube = 52.4%
- **10D** : Hypersphère / hypercube = 0.25%
- **100D** : Hypersphère / hypercube ≈ 0% (volume quasi-nul)

Conséquence : En haute dimension, les points sont presque tous à la même distance les uns des autres (tous sur la "surface" de l'hypersphère). Les distances relatives perdent leur sens.

Impact sur l'IA

- **k-NN dégrade** : Les k plus proches voisins deviennent presque aussi éloignés que les points aléatoires
- **Densité faible** : Pour couvrir l'espace, besoin de **N ∝ exp(d)** points (explosion exponentielle)
- **Recherche approximative nécessaire** : Recherche exacte devient prohibitive

Solutions pratiques en IA

1. Réduction de dimension

- **PCA** (Principal Component Analysis) : Projection linéaire sur axes de variance maximale
- **t-SNE, UMAP** : Réduction non-linéaire pour visualisation 2D/3D
- **Autoencoders** : Réseaux de neurones pour compression intelligente

2. Indexes ANN optimisés

HNSW, IVF+PQ sont spécifiquement conçus pour fonctionner efficacement en haute dimension en exploitant des structures de données intelligentes (graphes, clustering).

3. Métriques adaptées

La **similarité cosinus** (angle entre vecteurs) est souvent plus robuste que la distance euclidienne en haute dimension, car elle ignore la magnitude et se concentre sur l'orientation.

Démonstration : Distances en haute dimension

```
import numpy as np
from scipy.spatial.distance import euclidean

def test_curse_of_dimensionality(n_dims):
    # Générer 1000 vecteurs aléatoires
    vectors = np.random.randn(1000, n_dims)
    query = np.random.randn(n_dims)

    # Calculer distances au point query
    distances = [euclidean(query, v) for v in vectors]

    avg_dist = np.mean(distances)
    std_dist = np.std(distances)

    # Ratio std/mean : mesure de dispersion relative
    # Faible ratio = tous à distance similaire (curse)
    return avg_dist, std_dist, std_dist / avg_dist

# Tester pour différentes dimensions
for dims in [2, 10, 50, 100, 500, 1000]:
    avg, std, ratio = test_curse_of_dimensionality(dims)
    print(f"{dims}D: avg={avg:.2f}, std={std:.2f}, ratio={ratio:.3f}")

# Output approximatif :
# 2D:    avg=1.42, std=0.52, ratio=0.366
# 10D:   avg=4.20, std=0.91, ratio=0.217
# 100D:  avg=13.2, std=0.95, ratio=0.072
# 1000D: avg=41.8, std=0.99, ratio=0.024
# → En haute dimension, les distances varient très peu (faible ratio)
```

Opérations vectorielles essentielles

Addition et soustraction de vecteurs

L'addition et la soustraction de vecteurs sont des opérations **élément par élément** qui permettent de combiner ou comparer des représentations vectorielles.

Addition vectorielle

Formule : $v + w = [v_1+w_1, v_2+w_2, ..., v_n+w_n]$

L'addition combine les caractéristiques de deux vecteurs. En embeddings textuels, cela permet d'**agréger du sens**.

Exemple : Addition de vecteurs

```
import numpy as np

# Vecteurs 3D simples
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])

# Addition
v_sum = v1 + v2
print(f"v1 + v2 = {v_sum}")  # [5, 7, 9]

# Exemple sémantique avec embeddings de mots (simplifié)
# En réalité, ces vecteurs ont 100-768D
vec_roi = np.array([0.5, 0.8, 0.1])    # "roi" (masculin, puissance)
vec_femme = np.array([-0.6, 0.2, 0.3]) # "femme" (féminin)
vec_homme = np.array([0.6, 0.2, 0.1])  # "homme" (masculin)

# Arithmétique sémantique : Roi - Homme + Femme ≈ Reine
vec_result = vec_roi - vec_homme + vec_femme
print(f"Roi - Homme + Femme = {vec_result}")  # Vecteur proche de "reine"

# Moyenne de vecteurs (agrégation)
documents = [np.random.randn(768) for _ in range(5)]
vec_moyen = np.mean(documents, axis=0)
print(f"Vecteur moyen de 5 documents : {vec_moyen.shape}")  # (768,)
```

Soustraction vectorielle

Formule : $v - w = [v_1-w_1, v_2-w_2, ..., v_n-w_n]$

La soustraction calcule la **différence directionnelle** entre deux vecteurs. En IA, cela permet d'extraire des **relations conceptuelles**.

Applications pratiques

- **Analogies** : "Paris est à France ce que Berlin est à ?" → vec(Berlin) - vec(Allemagne) + vec(France) ≈ vec(Paris)
- **Détection de biais** : vec(médecin) - vec(homme) vs vec(infirmière) - vec(femme)
- **Agrégation de documents** : Moyenne de paragraphes pour représenter un chapitre
- **Transfert de style** : vec(image) - vec(style1) + vec(style2)

Produit scalaire (dot product)

Le **produit scalaire** (ou produit interne) est l'opération vectorielle la plus utilisée en IA pour mesurer la **similarité** entre vecteurs.

Formule : $v \cdot w = v_1 \times w_1 + v_2 \times w_2 + ... + v_n \times w_n = \Sigma(v_i \times w_i)$

Interprétation géométrique : `v · w = ||v|| × ||w|| × cos(θ)` où θ est l'angle entre les deux vecteurs.

- **Produit élevé** : Vecteurs alignés (angle petit, similaires)
- **Produit proche de 0** : Vecteurs orthogonaux (angle 90°, non-corrélés)
- **Produit négatif** : Vecteurs opposés (angle > 90°, dissimilaires)

Exemple : Produit scalaire

```
import numpy as np

# Vecteurs 2D pour visualisation
v1 = np.array([3, 4])  # Norme = 5
v2 = np.array([4, 3])  # Norme = 5
v3 = np.array([-3, 4]) # Orthogonal à v1

# Produit scalaire
dot_v1_v2 = np.dot(v1, v2)
dot_v1_v3 = np.dot(v1, v3)

print(f"v1 · v2 = {dot_v1_v2}")  # 24 (vecteurs similaires)
print(f"v1 · v3 = {dot_v1_v3}")  # 7 (moins alignés)

# Application : Recherche dans base vectorielle
query = np.random.randn(768)  # Vecteur requête
database = np.random.randn(10000, 768)  # 10K documents

# Calculer similarité avec TOUS les documents (vectorisé)
scores = database @ query  # Multiplication matricielle optimisée
# Équivalent à : [np.dot(doc, query) for doc in database]

# Top 10 documents les plus similaires
top_10_indices = np.argsort(scores)[-10:][::-1]
print(f"Top 10 documents : {top_10_indices}")
print(f"Scores : {scores[top_10_indices]}")

# Temps de calcul : 10K vecteurs 768D en ~1-2ms avec NumPy optimisé
```

Optimisation GPU

Le produit scalaire est **hautement parallélisable**. Sur GPU, on peut calculer des millions de produits scalaires simultanément. Les bases vectorielles modernes (Qdrant, Pinecone) exploitent massivement cette propriété avec SIMD (CPU) et CUDA (GPU) pour atteindre des **centaines de milliers de QPS**.

Calcul de distance entre vecteurs

La **distance** entre vecteurs quantifie leur dissimilarité. Plusieurs métriques existent, chacune avec ses propriétés et usages spécifiques.

1. Distance euclidienne (L2)
Formule : `d_L2(v, w) = √(Σ(vᵢ - wᵢ)²)`

Distance géométrique directe ("ligne droite"). Sensible à la **magnitude** des vecteurs.

- **Valeurs** : 0 (identiques) à +∞

- **Usage** : Vision (images), embeddings non-normalisés, k-means
- **Propriété** : Satisfait inégalité triangulaire (métrique mathématique)

2. Similarité cosinus
Formule : `cos_sim(v, w) = (v · w) / (||v|| × ||w||)`

Mesure l'**angle** entre vecteurs (ignore la magnitude). Distance cosinus = 1 - cos_sim.

- **Valeurs** : -1 (opposés) à +1 (identiques), 0 = orthogonaux
- **Usage** : NLP (texte), systèmes de recommandation, embeddings normalisés
- **Propriété** : Invariant par mise à l'échelle (cos(2v, w) = cos(v, w))

3. Distance de Manhattan (L1)
Formule : `d_L1(v, w) = Σ|vᵢ - wᵢ|`

Somme des différences absolues (distance "taxi").

- **Usage** : Données clairsemées, variables catégorielles, certains cas de régularisation
- **Propriété** : Plus robuste aux outliers que L2

Comparaison des métriques

```
import numpy as np
from scipy.spatial.distance import euclidean, cosine, cityblock

# Deux vecteurs 5D
v1 = np.array([1, 2, 3, 4, 5])
v2 = np.array([2, 3, 4, 5, 6])
v3 = np.array([5, 4, 3, 2, 1])  # Inversé de v1

# Distance euclidienne
print(f"L2(v1, v2) = {euclidean(v1, v2):.3f}")    # 2.236
print(f"L2(v1, v3) = {euclidean(v1, v3):.3f}")    # 6.325

# Similarité cosinus (scipy.distance.cosine retourne 1 - cos_sim)
cos_sim_v1_v2 = 1 - cosine(v1, v2)
cos_sim_v1_v3 = 1 - cosine(v1, v3)
print(f"Cosine(v1, v2) = {cos_sim_v1_v2:.3f}")   # 0.998 (très similaires)
print(f"Cosine(v1, v3) = {cos_sim_v1_v3:.3f}")   # 0.636 (moins similaires)

# Distance Manhattan
print(f"L1(v1, v2) = {cityblock(v1, v2):.3f}")   # 5.0
print(f"L1(v1, v3) = {cityblock(v1, v3):.3f}")   # 12.0

# Cas spécial : vecteurs normalisés
v1_norm = v1 / np.linalg.norm(v1)
v2_norm = v2 / np.linalg.norm(v2)

# Pour vecteurs normalisés : distance euclidienne ∝ distance cosinus
print(f"\nVecteurs normalisés :")
print(f"L2 normé : {euclidean(v1_norm, v2_norm):.3f}")  # 0.070
print(f"Cosine : {1 - cosine(v1_norm, v2_norm):.3f}")  # 0.998
```

Métrique	Formule	Coût Calcul	Usage Principal
Euclidienne (L2)	$\sqrt{(\Sigma(v_i-w_i)^2)}$	Moyen (sqrt)	Images, k-means, SVM
Cosinus	$(v \cdot w)/(\|\|v\|\|\|\|w\|\|)$	Moyen (normalisation)	Texte, recommandation
Dot Product	$\Sigma(v_i \times w_i)$	Rapide	Vecteurs pré-normalisés
Manhattan (L1)	$\Sigma\|v_i-w_i\|$	Rapide	Données clairsemées

Normalisation vectorielle

La **normalisation** transforme un vecteur pour qu'il ait une **norme (longueur) de 1**, tout en préservant sa direction. C'est une étape cruciale en IA pour standardiser les embeddings.

Formule : `v_norm = v / ||v|| = v / √(Σvᵢ²)`

Pourquoi normaliser ?

1. Comparaison équitable

Sans normalisation, un vecteur avec de grandes valeurs dominera les calculs de distance, même s'il est sémantiquement moins pertinent. La normalisation assure que seule la **direction** (sens) compte, pas la magnitude.

2. Optimisation des calculs Pour approfondir, consultez Computer Vision en Cybersécurité : Détection et Surveillance.

Avec vecteurs normalisés, **similarité cosinus = produit scalaire**. On évite la division coûteuse, accélérant les recherches vectorielles.

`cos_sim(v, w) = (v · w) / (||v|| × ||w||)` devient simplement `v_norm · w_norm`

3. Stabilité numérique

Les réseaux de neurones génèrent parfois des vecteurs avec des valeurs extrêmes. La normalisation évite les **overflows** et instabilités numériques.

Exemple : Normalisation en pratique

```
import numpy as np

# Vecteur original
v = np.array([3.0, 4.0, 0.0])
print(f"Vecteur original : {v}")
print(f"Norme : {np.linalg.norm(v):.2f}")  # 5.00

# Normalisation
v_norm = v / np.linalg.norm(v)
print(f"Vecteur normalisé : {v_norm}")  # [0.6, 0.8, 0.0]
print(f"Nouvelle norme : {np.linalg.norm(v_norm):.2f}")  # 1.00

# Normalisation batch (efficace pour grandes matrices)
matrix = np.random.randn(10000, 768)  # 10K vecteurs 768D

# Méthode 1 : Boucle (lent)
# normalized = np.array([v / np.linalg.norm(v) for v in matrix])

# Méthode 2 : Vectorisé (rapide)
norms = np.linalg.norm(matrix, axis=1, keepdims=True)  # Shape: (10000, 1)
matrix_normalized = matrix / norms  # Broadcasting

print(f"Normes après normalisation : {np.linalg.norm(matrix_normalized, axis=1)[:5]}")
# [1. 1. 1. 1. 1.]

# Vérification : similarité cosinus = produit scalaire
v1_norm = matrix_normalized[0]
v2_norm = matrix_normalized[1]

cos_sim_manual = np.dot(v1_norm, v2_norm) / (np.linalg.norm(v1_norm) *
np.linalg.norm(v2_norm))
cos_sim_optimized = np.dot(v1_norm, v2_norm)  # Plus rapide!

print(f"Cosinus (méthode classique) : {cos_sim_manual:.4f}")
print(f"Cosinus (vecteurs normalisés) : {cos_sim_optimized:.4f}")  # Identiques
```

Attention : Vecteurs nuls

La normalisation d'un vecteur nul (toutes composantes = 0) provoque une **division par zéro**. En production, toujours vérifier : `if np.linalg.norm(v) > 1e-8: v_norm = v / np.linalg.norm(v)`

Bonnes pratiques

- **OpenAI/Cohere embeddings** : Déjà normalisés par l'API
- **Sentence-BERT** : Option `normalize_embeddings=True`
- **Bases vectorielles** : Qdrant/Pinecone supportent nativement la normalisation automatique
- **Recherche** : Normaliser requête ET documents pour cohérence

Applications des vecteurs en IA

Word embeddings et traitement du langage

Les **embeddings** ont transformé le NLP en remplaçant les représentations symboliques (one-hot) par des **vecteurs denses capturant la sémantique**.

Systèmes RAG (Retrieval-Augmented Generation)
Application la plus populaire en 2024-2025. Les systèmes RAG combinent :

1. **Vectorisation** : Documents convertis en embeddings et stockés dans une base vectorielle
2. **Recherche** : Question utilisateur → vecteur → top-k documents similaires
3. **Génération** : LLM (GPT, Claude) génère réponse avec contexte récupéré

Exemples concrets :

- **Support client** : Base de connaissances → réponses instantanées contextualisées
- **Assistants juridiques** : Recherche dans milliers de documents légaux
- **Documentation technique** : Recherche sémantique dans code + docs

Traduction automatique
Les modèles transformers (Google Translate, DeepL) encodent phrase source en vecteur **indépendant de la langue**, puis décodent vers langue cible.

Analyse de sentiment
Phrase → vecteur → classifieur → sentiment (positif/négatif/neutre). Les embeddings capturent les nuances ("excellent" vs "pas mal" vs "décevant").

Systèmes de recommandation

Les vecteurs permettent de représenter utilisateurs ET contenus dans le **même espace latent**, où la similarité prédit l'affinité.

Architecture typique
1. Vecteurs utilisateurs

- Historique d'achats/vues/clics → vecteur 64-256D
- Capture préférences implicites (genres, styles, prix...)

2. Vecteurs contenus

- Caractéristiques produit/film/musique → vecteur même dimension
- Encodé par réseau de neurones ou matrix factorization

3. Scoring

```
score(user, item) = vecteur_user · vecteur_item
```

Plus le score est élevé, plus l'utilisateur est susceptible d'apprécier le contenu.

Exemple simplifié : Recommandation de films

```
import numpy as np

# Vecteurs utilisateurs (100 users, 64 dimensions)
user_vectors = np.random.randn(100, 64)

# Vecteurs films (1000 films, 64 dimensions)
movie_vectors = np.random.randn(1000, 64)

# Recommandation pour user_id = 42
user_id = 42
user_vec = user_vectors[user_id]

# Calculer scores pour TOUS les films (vectorisé)
scores = movie_vectors @ user_vec  # Shape: (1000,)

# Top 10 recommandations
top_10_movies = np.argsort(scores)[-10:][::-1]
print(f"Films recommandés pour user {user_id} : {top_10_movies}")
print(f"Scores : {scores[top_10_movies]}")

# En production : filtrer films déjà vus, appliquer règles business
```

Cas d'usage industrie :

- **Netflix** : Vecteurs utilisateurs × séries/films (collaborative filtering)
- **Spotify** : Vecteurs chansons basés sur audio + métadonnées + écoutes
- **Amazon** : "Clients ayant acheté X ont aussi acheté Y" via similarité vectorielle
- **LinkedIn** : Recommandation d'offres d'emploi basée sur profil vectorisé

Reconnaissance d'images

Les vecteurs d'images (générés par CNN) permettent des tâches variées : classification, recherche, détection de duplicatas, reconnaissance faciale.

Classification d'images
Pipeline : Image → CNN → vecteur 512-2048D → couche dense → probabilités classes

- **ImageNet** : 1000 classes (chat, chien, voiture, ...)
- **Custom** : Radiologie (tumeur/sain), agriculture (maladie plantes), industrie (défauts produits)

Recherche d'images similaires
Utilisé par Pinterest, Google Images, e-commerce ("trouver produits similaires").

1. Toutes les images du catalogue → vecteurs via ResNet/EfficientNet
2. Stockage dans base vectorielle (Qdrant, Milvus)
3. Image requête → vecteur → recherche k-NN → images similaires

Reconnaissance faciale
Modèles spécialisés (FaceNet, ArcFace) génèrent **vecteurs 128-512D uniques par visage**.

- **Distance < seuil** : Même personne (authentification)
- **Distance > seuil** : Personnes différentes
- **Précision** : 99.5%+ sur LFW benchmark

Applications industrielles

- **E-commerce** : Recherche visuelle (photo produit → articles similaires)
- **Sécurité** : Contrôle accès biométrique, vidéo-surveillance
- **Santé** : Détection anomalies radiologiques par comparaison avec base saine
- **Automobile** : Détection piétons/panneaux (vecteurs multiples par frame)

Recherche sémantique

La recherche sémantique transcende la recherche par mots-clés en comprenant l'**intention et le sens** plutôt que les termes exacts.

Recherche textuelle vs sémantique

Critère	Recherche Textuelle (BM25)	Recherche Sémantique (Vecteurs)
Principe	Correspondance mots exacts	Similarité de sens
Requête	"voiture électrique"	"véhicule à batterie"
Résultats	Seulement docs contenant ces mots exacts	Docs sur voitures électriques même sans ces termes
Synonymes	Non gérés nativement	Automatiquement compréhens
Fautes orthographe	Problématique	Tolérées (similarité sémantique préservée)

Architecture recherche sémantique

1. **Indexation** : Documents → chunks (paragraphes) → embeddings → base vectorielle
2. **Requête** : Question → embedding (même modèle)
3. **Recherche** : k-NN dans base vectorielle → top-k chunks
4. **Reranking** (optionnel) : Modèle cross-encoder pour affiner l'ordre

Exemple : Recherche sémantique avec Sentence-BERT

```
from sentence_transformers import SentenceTransformer
import numpy as np

# Modèle d'embedding
model = SentenceTransformer('all-MiniLM-L6-v2')  # 384D, rapide

# Corpus de documents
documents = [
    "Les voitures électriques réduisent les émissions de CO2",
    "La batterie lithium-ion équipe la plupart des véhicules électriques",
    "Le moteur thermique fonctionne avec de l'essence ou du diesel",
    "Les panneaux solaires produisent de l'énergie renouvelable"
]

# Vectoriser corpus
doc_embeddings = model.encode(documents, normalize_embeddings=True)

# Requête utilisateur
query = "véhicule à batterie"  # Pas de mots exacts du corpus!
query_embedding = model.encode(query, normalize_embeddings=True)

# Recherche (produit scalaire = similarité cosinus car normalisés)
scores = doc_embeddings @ query_embedding

# Résultats triés
ranked_indices = np.argsort(scores)[::-1]

print("Résultats recherche sémantique :")
for i, idx in enumerate(ranked_indices, 1):
    print(f"{i}. [{scores[idx]:.3f}] {documents[idx]}")

# Output :
# 1. [0.672] La batterie lithium-ion équipe la plupart des véhicules électriques
# 2. [0.589] Les voitures électriques réduisent les émissions de CO2
# 3. [0.412] Le moteur thermique fonctionne avec de l'essence ou du diesel
# 4. [0.198] Les panneaux solaires produisent de l'énergie renouvelable
```

Hybridation BM25 + Sémantique : Approche optimale en production.

- **BM25** : Excellent pour requêtes factuelles (noms propres, codes, références)
- **Vectorielle** : Supérieur pour questions conceptuelles, synonymes
- **Fusion** : Combiner scores (RRF - Reciprocal Rank Fusion) pour bénéficier des deux

Vecteurs denses vs vecteurs creux

Caractéristiques des vecteurs denses

Un **vecteur dense** contient majoritairement des valeurs **non-nulles**. La plupart des embeddings modernes (BERT, OpenAI, images CNN) sont denses.

Propriétés

- **Distribution** : Valeurs réparties uniformément, peu de zéros
- **Dimension** : Typiquement 64-2048D (relativement faible vs dimensionnalité originale)
- **Représentation** : Array NumPy classique, tous éléments stockés en mémoire
- **Densité** : 90-100% valeurs non-nulles

Avantages

- **Expressivité** : Chaque dimension contribue à l'information sémantique
- **Performance** : Calculs vectorisés (SIMD, GPU) extrêmement efficaces
- **Généralisation** : Réseaux de neurones apprennent représentations compactes et robustes
- **Recherche ANN** : Indexes HNSW/IVF optimisés pour vecteurs denses

Inconvénients

- **Mémoire** : 1M vecteurs 768D float32 = 3GB RAM (toutes valeurs stockées)
- **Calcul** : Chaque dimension traitée, même si contribution faible
- **Interprétabilité** : Dimensions = features latentes non-explicites

Exemple : Vecteur dense

```
import numpy as np

# Vecteur dense typique (embedding textuel)
dense_vector = np.array([
    0.234, -0.567, 0.123, 0.891, -0.234, 0.456, 0.789, -0.123,
    0.345, -0.678, 0.901, 0.234, -0.456, 0.678, -0.890, 0.123
])  # 16D (simplifié, réel = 768D)

print(f"Dimension : {dense_vector.shape[0]}")
print(f"Valeurs non-nulles : {np.count_nonzero(dense_vector)}")
print(f"Densité : {np.count_nonzero(dense_vector) / len(dense_vector) * 100:.1f}%")
# Output: 100% (toutes valeurs significatives)

# Mémoire
print(f"Mémoire : {dense_vector.nbytes} bytes")  # 128 bytes (16 × 8 bytes float64)

# Vecteur dense réel (768D)
real_embedding = np.random.randn(768)
print(f"\nVecteur réel 768D : {real_embedding.nbytes / 1024:.2f} KB")  # ~6 KB
```

Usage typique : NLP moderne (transformers), vision (CNN), audio (Wav2Vec), systèmes multi-modaux.

Caractéristiques des vecteurs creux

Un **vecteur creux** (sparse) contient majoritairement des **zéros**. Utilisés historiquement en NLP (TF-IDF, bag-of-words) et pour certaines données structurées.

Propriétés

- **Distribution** : 90-99.9% de zéros, quelques valeurs non-nulles
- **Dimension** : Très élevée (10K-1M), taille du vocabulaire ou espace features
- **Représentation** : Formats spécialisés (CSR, COO) stockant seulement valeurs non-nulles
- **Densité** : 0.1-10% valeurs non-nulles

Avantages

- **Mémoire efficace** : Stocke seulement (index, valeur) des non-zéros
- **Interprétabilité** : Dimensions correspondent à features explicites (mots, caractéristiques)
- **Calculs optimisés** : Algorithmes spécialisés ignorent les zéros

Inconvénients

- **Haute dimensionnalité** : Malédiction de la dimension exacerbée
- **Pas de sémantique** : Mots similaires = vecteurs orthogonaux ("chat" et "félin" totalement différents)
- **Performance ANN** : Indexes moins efficaces que pour vecteurs denses

Exemple : Vecteur creux (TF-IDF)

```
import numpy as np
from scipy.sparse import csr_matrix
from sklearn.feature_extraction.text import TfidfVectorizer

# Corpus
documents = [
    "le chat mange des croquettes",
    "le chien court dans le jardin",
    "les oiseaux chantent le matin"
]

# Vectorisation TF-IDF (sparse)
vectorizer = TfidfVectorizer()
sparse_matrix = vectorizer.fit_transform(documents)

print(f"Forme : {sparse_matrix.shape}")  # (3 documents, ~15 mots uniques)
print(f"Type : {type(sparse_matrix)}")   # scipy.sparse.csr.csr_matrix
print(f"Densité : {sparse_matrix.nnz / (sparse_matrix.shape[0] * sparse_matrix.shape[1]) *
100:.1f}%")
# Output: ~30-40% (faible densité)

# Vecteur du premier document
doc1_sparse = sparse_matrix[0]
print(f"\nDocument 1 (sparse) :")
print(f"  Valeurs non-nulles : {doc1_sparse.nnz}")
print(f"  Mémoire : {doc1_sparse.data.nbytes + doc1_sparse.indices.nbytes} bytes")

# Conversion en dense pour comparaison
doc1_dense = doc1_sparse.toarray().flatten()
print(f"\nDocument 1 (dense) :")
print(f"  Valeurs non-nulles : {np.count_nonzero(doc1_dense)}")
print(f"  Mémoire : {doc1_dense.nbytes} bytes")
print(f"  Gain mémoire sparse : {doc1_dense.nbytes / (doc1_sparse.data.nbytes +
doc1_sparse.indices.nbytes):.1f}x")

# Vocabulaire
print(f"\nVocabulaire : {vectorizer.get_feature_names_out()[:10]}")
```

Usage typique : TF-IDF, bag-of-words, one-hot encoding, données catégorielles, graphes creux.

Quand utiliser quel type ?

Le choix entre vecteurs denses et creux dépend du cas d'usage, des données et des contraintes de performance.

Critère	Vecteurs Denses	Vecteurs Creux
Dimension	64-2048	10K-1M+
Densité	90-100%	0.1-10%
Mémoire (1M vecteurs)	3-12 GB	0.1-2 GB (selon densité)
Sémantique	Oui (embeddings)	Non (syntaxique)
Recherche ANN	Excellent (HNSW, IVF)	Moyen
Interprétabilité	Faible (latent)	Élevée (features explicites)

Guide de décision

Choisir DENSE si :

- Recherche sémantique / RAG / NLP moderne
- Images, audio, vidéo
- Recommandation (collaborative filtering neural)
- Besoin de comprendre synonymes, contexte
- Volume < 100M vecteurs (mémoire gérable)

Choisir CREUX si :

- Recherche par mots-clés exacte
- Données catégorielles (one-hot)
- Interprétabilité critique (médical, juridique)
- Mémoire extrêmement limitée
- Features explicites importantes (TF-IDF + règles métier)

Approche hybride (recommandée)

En production, combiner les deux types apporte le meilleur des deux mondes :

- **BM25 (creux) + Embeddings (denses)** : Recherche factuelle ET sémantique
- **SPLADE** : Modèle générant vecteurs creux avec sémantique (apprises par transformers)
- **Hybrid search** : Qdrant/Weaviate supportent recherche simultanée dense+sparse

Impact sur les performances

Le choix dense vs creux a des implications majeures sur latence, débit, mémoire et précision.

Performance de recherche

Métrique	Dense (HNSW)	Creux (Inverted Index)
Latence (1M vecteurs)	10-30ms	5-20ms
Débit (QPS)	1K-10K	5K-50K
Précision sémantique	85-95% (excellent)	60-75% (moyen)
Scalabilité (100M+)	Possible avec optimisations	Excellent

Coûts infrastructure

Vecteurs denses (1M documents, 768D)

- **RAM** : 6-12 GB (index HNSW)
- **SSD** : 3-6 GB (persistence)
- **CPU** : 4-8 cores pour 1K QPS
- **GPU** (optionnel) : 10-100x accélération pour grandes bases

Vecteurs creux (1M documents, 50K dim, 1% densité)

- **RAM** : 0.5-2 GB (inverted index)
- **SSD** : 0.2-1 GB
- **CPU** : 2-4 cores pour 5K QPS
- **GPU** : Peu bénéfique

Benchmark : Dense vs Creux

```
import time
import numpy as np
from scipy.sparse import random as sparse_random

# Setup
n_docs = 100000
dim_dense = 768
dim_sparse = 50000
sparse_density = 0.01  # 1%

# Vecteurs denses
dense_db = np.random.randn(n_docs, dim_dense).astype(np.float32)
dense_query = np.random.randn(dim_dense).astype(np.float32)

# Vecteurs creux
sparse_db = sparse_random(n_docs, dim_sparse, density=sparse_density, format='csr')
sparse_query = sparse_random(1, dim_sparse, density=sparse_density, format='csr')

# Benchmark dense
start = time.time()
scores_dense = dense_db @ dense_query
top_10_dense = np.argsort(scores_dense)[-10:][::-1]
time_dense = (time.time() - start) * 1000

# Benchmark creux
start = time.time()
scores_sparse = sparse_db @ sparse_query.T
top_10_sparse = np.argsort(scores_sparse.toarray().flatten())[-10:][::-1]
time_sparse = (time.time() - start) * 1000

print(f"Dense (768D, 100K docs) : {time_dense:.2f}ms")
print(f"Sparse (50KD, 1% density) : {time_sparse:.2f}ms")
print(f"Ratio : {time_dense / time_sparse:.2f}x")

# Mémoire
print(f"\nMémoire dense : {dense_db.nbytes / 1024**2:.1f} MB")
print(f"Mémoire creux : {(sparse_db.data.nbytes + sparse_db.indices.nbytes) / 1024**2:.1f}
MB")
```

Trade-off fondamental

Dense : Meilleure qualité sémantique, mais plus coûteux en ressources.
Creux : Plus rapide et économe, mais précision sémantique limitée.
Solution : Hybride dense+sparse pour bénéficier des deux (Qdrant, Weaviate, Elasticsearch 8+).

Outils et bibliothèques pour travailler avec les vecteurs

NumPy : la base du calcul vectoriel en Python

NumPy est la bibliothèque fondamentale pour le calcul numérique en Python. Toutes les frameworks IA (TensorFlow, PyTorch, Scikit-learn) reposent sur NumPy.

Fonctionnalités essentielles

- **Arrays n-dimensionnels** : Structure de données optimisée pour calculs vectoriels
- **Opérations vectorisées** : Calculs parallélisés (C/Fortran sous le capot)
- **Algèbre linéaire** : Module `linalg` pour matrices, normes, décompositions

• **Broadcasting** : Opérations automatiques entre arrays de tailles différentes

NumPy : Opérations vectorielles essentielles

```
import numpy as np

# Création de vecteurs
v1 = np.array([1, 2, 3, 4, 5])
v2 = np.random.randn(768)  # Vecteur aléatoire 768D
v3 = np.zeros(100)         # Vecteur de zéros
v4 = np.ones(50)           # Vecteur de uns

# Opérations arithmétiques (vectorisées)
v_sum = v1 + v1            # Addition
v_mult = v1 * 2            # Multiplication scalaire
v_power = v1 ** 2          # Puissance

# Produit scalaire
dot_product = np.dot(v1, v1)  # ou v1 @ v1
print(f"Produit scalaire : {dot_product}")  # 55

# Norme et normalisation
norm = np.linalg.norm(v1)  # √(1²+2²+3²+4²+5²) = 7.416
v1_normalized = v1 / norm

# Matrices (batch de vecteurs)
matrix = np.random.randn(1000, 768)  # 1000 vecteurs 768D

# Produits matriciels (très optimisé)
query = np.random.randn(768)
scores = matrix @ query  # Shape: (1000,) - Tous les produits scalaires

# Top-k indices
top_10 = np.argsort(scores)[-10:][::-1]

# Statistiques
mean_vec = np.mean(matrix, axis=0)  # Vecteur moyen
std_vec = np.std(matrix, axis=0)    # Écart-type par dimension

# Sauvegarde/Chargement
np.save('embeddings.npy', matrix)
loaded = np.load('embeddings.npy')

print(f"Performance : {matrix.shape[0]} vecteurs traités instantanément")
```

Bonnes pratiques NumPy

• **Éviter les boucles** : Toujours privilégier opérations vectorisées (100-1000x plus rapide)
• **Type float32** : Divise mémoire par 2 vs float64, suffisant pour IA
• **In-place ops** : `v1 += v2` au lieu de `v1 = v1 + v2` pour économiser mémoire
• **Axis parameter** : Maîtriser pour opérations sur matrices (mean, sum, std...)

TensorFlow et PyTorch

PyTorch et **TensorFlow** sont les frameworks deep learning dominants. Tous deux manipulent des vecteurs (tensors) avec accélération GPU.

PyTorch (recommandé pour recherche et prototypage)

- **API pythonique** : Proche de NumPy, courbe d'apprentissage douce
- **Dynamic computation graph** : Plus flexible pour expérimentations
- **Adoption** : Dominant en recherche IA (80%+ des papers NeurIPS/ICML)

PyTorch : Manipulation de vecteurs

```
import torch

# Création tensors (GPU si disponible)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

v1 = torch.randn(768, device=device)  # Vecteur sur GPU
matrix = torch.randn(10000, 768, device=device)  # 10K vecteurs

# Opérations (syntaxe similaire NumPy)
dot_product = torch.dot(v1, v1)
norm = torch.norm(v1)
v1_normalized = v1 / norm

# Batch operations
query = torch.randn(768, device=device)
scores = matrix @ query  # Produits scalaires parallèles GPU

# Top-k (optimisé GPU)
top_values, top_indices = torch.topk(scores, k=10)

print(f"Top 10 indices : {top_indices}")
print(f"Device : {v1.device}")

# Conversion NumPy <-> PyTorch
numpy_array = v1.cpu().numpy()  # Tensor → NumPy
torch_tensor = torch.from_numpy(numpy_array).to(device)  # NumPy → Tensor

# Embeddings avec modèle pré-entraîné
from transformers import AutoModel, AutoTokenizer

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

inputs = tokenizer("Bonjour le monde", return_tensors='pt').to(device)
with torch.no_grad():
    outputs = model(**inputs)
    embedding = outputs.last_hidden_state.mean(dim=1)  # Pooling

print(f"Embedding shape : {embedding.shape}")  # (1, 768)
```

TensorFlow (production et déploiement)

- **TensorFlow Serving** : Déploiement production optimisé
- **TensorFlow Lite** : Mobile et edge devices
- **Adoption** : Google, industries avec infrastructure TF existante

Comparaison rapide :

Critère	PyTorch	TensorFlow
Facilité d'usage	Excellent (pythonique)	Moyen (plus verbeux)
Débogage	Facile (eager execution)	Plus complexe (graph mode)
Production	TorchServe (récent)	TF Serving (mature)
Communauté recherche	Dominant	Minoritaire
Mobile	PyTorch Mobile	TF Lite (plus mature)

Scikit-learn pour le machine learning

Scikit-learn est la bibliothèque de référence pour le machine learning classique (non-deep learning) en Python.

Fonctionnalités pour vecteurs

- **Preprocessing** : Normalisation, standardisation, scaling
- **Dimensionality reduction** : PCA, t-SNE, UMAP
- **Clustering** : k-means, DBSCAN sur vecteurs
- **Classification/Régression** : SVM, Random Forest, etc.
- **Metrics** : Calculs de distances, similarités

Scikit-learn : Workflows vectoriels

```
from sklearn.preprocessing import StandardScaler, normalize
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Dataset : 1000 vecteurs 768D
vectors = np.random.randn(1000, 768)

# 1. Normalisation (vecteurs unités)
vectors_normalized = normalize(vectors, norm='l2')
print(f"Normes après normalisation : {np.linalg.norm(vectors_normalized, axis=1)[:5]}")
# [1. 1. 1. 1. 1.]

# 2. Standardisation (mean=0, std=1 par dimension)
scaler = StandardScaler()
vectors_scaled = scaler.fit_transform(vectors)
print(f"Mean par dim : {vectors_scaled.mean(axis=0)[:5]}")
print(f"Std par dim : {vectors_scaled.std(axis=0)[:5]}")

# 3. Réduction de dimension : 768D → 128D
pca = PCA(n_components=128)
vectors_reduced = pca.fit_transform(vectors)
print(f"Variance expliquée : {pca.explained_variance_ratio_.sum():.2%}")
print(f"Nouvelle shape : {vectors_reduced.shape}")  # (1000, 128)

# 4. Clustering : Trouver 10 groupes
kmeans = KMeans(n_clusters=10, random_state=42)
clusters = kmeans.fit_predict(vectors_reduced)
print(f"Clusters : {np.bincount(clusters)}")  # Nombre de vecteurs par cluster

# 5. Similarité cosinus (batch)
query = np.random.randn(1, 768)
similarities = cosine_similarity(query, vectors)[0]  # Shape: (1000,)
top_10 = np.argsort(similarities)[-10:][::-1]
print(f"Top 10 plus similaires : {top_10}")
print(f"Scores : {similarities[top_10]}")
```

Quand utiliser Scikit-learn ?

- **Preprocessing** : Toujours pour normalisation, scaling, encoding
- **Prototypage rapide** : Baselines ML avant deep learning
- **Petits datasets** : <1M exemples, Scikit-learn suffit souvent
- **Évaluation** : Metrics (accuracy, F1, confusion matrix...)
- **Pas GPU nécessaire** : Optimisations CPU suffisantes

Bibliothèques de manipulation d'embeddings

Des bibliothèques spécialisées simplifient la génération et manipulation d'embeddings pour diverses modalités.

Sentence-Transformers (NLP)
Bibliothèque de référence pour embeddings textuels sémantiques.

- **Modèles pré-entraînés** : 100+ modèles optimisés (multilangues, domaines spécifiques)
- **API simple** : 3 lignes de code pour embeddings production-ready

- **Performance** : Batch processing, GPU, quantization

Sentence-Transformers : Exemples

```
from sentence_transformers import SentenceTransformer
import numpy as np

# Charger modèle (multilangue, 384D)
model = SentenceTransformer('paraphrase-multilingual-MiniLM-L12-v2')

# Encoder phrases
textes = [
    "L'intelligence artificielle transforme le monde",
    "L'IA change de nombreux secteurs",
    "J'aime les croissants au beurre"
]

embeddings = model.encode(textes, normalize_embeddings=True)
print(f"Shape : {embeddings.shape}")  # (3, 384)

# Similarités (matrice 3x3)
similarities = embeddings @ embeddings.T
print("Matrice similarités :")
print(similarities)
# [[1.00, 0.87, 0.12],
#  [0.87, 1.00, 0.15],
#  [0.12, 0.15, 1.00]]

# Batch processing (efficace pour grandes quantités)
large_corpus = [f"Document {i}" for i in range(10000)]
embeddings_batch = model.encode(
    large_corpus,
    batch_size=32,
    show_progress_bar=True,
    convert_to_numpy=True
)

print(f"10K documents encodés : {embeddings_batch.shape}")  # (10000, 384)
```

Autres bibliothèques spécialisées
OpenAI / Cohere / Voyage AI (APIs)

- Embeddings de qualité maximale (1536-4096D)
- Pas d'infrastructure à gérer
- Coût : $0.0001-0.0004 / 1K tokens

CLIP / ImageBind (multimodal)

- Embeddings image+texte dans espace partagé
- Recherche cross-modale
- Bibliothèque `transformers` (Hugging Face)

Faiss (Meta)

- Recherche vectorielle ultra-optimisée (CPU/GPU)
- Indexes : IVF, HNSW, PQ
- Scalabilité : Milliards de vecteurs

Recommandations par cas d'usage

- **RAG multilingue** : Sentence-Transformers (paraphrase-multilingual-mpnet-base-v2)
- **Qualité maximale** : OpenAI text-embedding-3-large (3072D)
- **On-premise / privacy** : BGE, E5 (open-source, performants)
- **Images** : CLIP (OpenAI) ou ImageBind (Meta)
- **Audio** : Wav2Vec2, Whisper embeddings

Bonnes pratiques et pièges à éviter

Normalisation et preprocessing

Le preprocessing des vecteurs est **crucial** pour garantir performance et cohérence. Des vecteurs mal préparés dégradent drastiquement les résultats.

1. Normalisation (recommandée pour similarité cosinus)
Quand : Systèmes RAG, recherche sémantique, recommandation

```
import numpy as np

# TOUJOURS normaliser si utilisation similarité cosinus
vectors = np.random.randn(1000, 768)
vectors_normalized = vectors / np.linalg.norm(vectors, axis=1, keepdims=True)

# Vérification
assert np.allclose(np.linalg.norm(vectors_normalized, axis=1), 1.0)
```

2. Standardisation (mean=0, std=1)
Quand : Avant clustering, PCA, certains classifieurs

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
vectors_scaled = scaler.fit_transform(vectors)

# Sauvegarder scaler pour inference
import joblib
joblib.dump(scaler, 'scaler.pkl')
```

3. Gestion des valeurs manquantes

```
# Détecter NaN/Inf
if np.isnan(vectors).any() or np.isinf(vectors).any():
    print("ATTENTION : Valeurs invalides détectées")
    # Remplacer NaN par 0
    vectors = np.nan_to_num(vectors, nan=0.0, posinf=0.0, neginf=0.0)
```

Erreurs fréquentes

- **Normaliser documents mais pas query** : Incohérence fatale
- **Scaler différent train/prod** : Résultats erronés
- **Oublier de sauvegarder preprocessors** : Impossible de reproduire

• **Diviser par norme nulle** : Toujours vérifier norm > epsilon

Gestion de la mémoire avec des vecteurs de haute dimension

Les vecteurs haute dimension consomment rapidement la mémoire. Optimisations essentielles pour éviter les crashes et coûts excessifs.

1. Utiliser float32 au lieu de float64
Impact : Division par 2 de la mémoire, précision suffisante pour IA

```
import numpy as np

# MAUVAIS : float64 par défaut
vectors_f64 = np.random.randn(1000000, 768)  # 6 GB

# BON : Spécifier float32
vectors_f32 = np.random.randn(1000000, 768).astype(np.float32)  # 3 GB

print(f"Ratio mémoire : {vectors_f64.nbytes / vectors_f32.nbytes:.1f}x")
```

2. Batch processing pour grandes quantités

```
def process_large_dataset(texts, model, batch_size=32):
    """Traiter par batches pour éviter OOM (Out Of Memory)"""
    embeddings = []
    for i in range(0, len(texts), batch_size):
        batch = texts[i:i+batch_size]
        batch_embeddings = model.encode(batch)
        embeddings.append(batch_embeddings)
    return np.vstack(embeddings)

# Traiter 1M documents sans saturer RAM
texts = [f"Document {i}" for i in range(1000000)]
embeddings = process_large_dataset(texts, model, batch_size=128)
```

3. Memmap pour datasets trop grands pour RAM

```
# Créer fichier memmap (lecture/écriture sur disque)
memmap_vectors = np.memmap(
    'vectors.dat',
    dtype='float32',
    mode='w+',
    shape=(10000000, 768)  # 10M vecteurs, seulement 30GB disque
)

# Écriture progressive
for i in range(0, 10000000, 1000):
    memmap_vectors[i:i+1000] = generate_batch_embeddings()
    if i % 100000 == 0:
        memmap_vectors.flush()  # Persist sur disque

# Lecture : Seules les données accédées sont chargées en RAM
subset = memmap_vectors[1000:2000]  # Charge seulement 1000 vecteurs
```

4. Quantization (compression avec perte)

```
# Quantization 8-bit : float32 (4 bytes) → int8 (1 byte)
def quantize_vectors(vectors, num_bits=8):
    min_val, max_val = vectors.min(), vectors.max()
    scale = (max_val - min_val) / (2**num_bits - 1)
    quantized = ((vectors - min_val) / scale).astype(np.uint8)
    return quantized, min_val, scale

def dequantize_vectors(quantized, min_val, scale):
    return quantized.astype(np.float32) * scale + min_val

vectors = np.random.randn(100000, 768).astype(np.float32)
quant, min_v, scale = quantize_vectors(vectors)

print(f"Mémoire originale : {vectors.nbytes / 1024**2:.1f} MB")
print(f"Mémoire quantizée : {quant.nbytes / 1024**2:.1f} MB")
print(f"Compression : {vectors.nbytes / quant.nbytes:.1f}x")  # 4x
```

Coûts typiques (AWS/GCP)

- **1M vecteurs 768D float32** : 3GB RAM → ~$20/mois (instance dédiée)
- **10M vecteurs** : 30GB RAM → ~$150/mois
- **100M vecteurs** : 300GB RAM → ~$1500/mois (ou sharding)
- **Avec quantization/PQ** : Diviser coûts par 4-10x

Optimisation des calculs vectoriels

Les calculs vectoriels peuvent être accélérés de 10-1000x avec les bonnes techniques.

1. Vectorisation : Éliminer les boucles Python

```
import numpy as np
import time

matrix = np.random.randn(10000, 768).astype(np.float32)
query = np.random.randn(768).astype(np.float32)

# MAUVAIS : Boucle Python (LENT)
start = time.time()
scores_slow = [np.dot(query, vec) for vec in matrix]
time_slow = time.time() - start

# BON : Opération vectorisée
start = time.time()
scores_fast = matrix @ query
time_fast = time.time() - start

print(f"Boucle : {time_slow*1000:.1f}ms")
print(f"Vectorisé : {time_fast*1000:.1f}ms")
print(f"Accélération : {time_slow/time_fast:.0f}x")  # Typiquement 100-500x
```

2. Utilisation GPU pour grandes opérations

```
import torch

# Transférer sur GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
matrix_gpu = torch.from_numpy(matrix).to(device)
query_gpu = torch.from_numpy(query).to(device)

# Calcul GPU (10-100x plus rapide pour grandes matrices)
start = time.time()
scores_gpu = matrix_gpu @ query_gpu
torch.cuda.synchronize()  # Attendre fin calcul GPU
time_gpu = time.time() - start

print(f"CPU : {time_fast*1000:.1f}ms")
print(f"GPU : {time_gpu*1000:.1f}ms")
print(f"Accélération GPU : {time_fast/time_gpu:.1f}x")
```

3. Pré-calculs et caching

```
# Si similarité cosinus utilisée : pré-normaliser TOUT
vectors_normalized = vectors / np.linalg.norm(vectors, axis=1, keepdims=True)
query_normalized = query / np.linalg.norm(query)

# Maintenant : similarité cosinus = simple produit scalaire (plus rapide)
scores = vectors_normalized @ query_normalized

# Cache résultats fréquents
from functools import lru_cache

@lru_cache(maxsize=1000)
def get_embedding(text: str):
    return model.encode(text)

# Requêtes identiques → cache hit (instantané)
```

4. Approximations intelligentes

```
# Pour top-k : Pas besoin de trier TOUT
import heapq

# LENT : Tri complet
top_k_slow = np.argsort(scores)[-10:][::-1]

# RAPIDE : Heap partiel
top_k_fast = heapq.nlargest(10, range(len(scores)), key=lambda i: scores[i])

# Pour distances : Utiliser carré de distance euclidienne (sans sqrt)
# sqrt(sum((a-b)^2)) vs sum((a-b)^2) : Même ordre, 2x plus rapide
```

Checklist optimisation

- ✅ Utiliser float32 partout (sauf si précision critique)
- ✅ Vectoriser : remplacer boucles par opérations NumPy/PyTorch
- ✅ Pré-normaliser si cosinus (produit scalaire ensuite)
- ✅ GPU si matrices >1000x1000

- ✅ Indexes ANN (HNSW) si recherche >100K vecteurs
- ✅ Batch processing : Traiter par lots de 32-256
- ✅ Profiler : `cProfile` pour identifier bottlenecks

Erreurs courantes à éviter

1. Incohérence normalisation query/documents

Erreur fatale

```
# FAUX : Documents normalisés, query non-normalisée
docs_normalized = normalize(docs)
query_raw = model.encode(query)  # Non-normalisé
scores = docs_normalized @ query_raw  # Résultats faux!

# CORRECT : Même traitement
docs_normalized = normalize(docs)
query_normalized = normalize(model.encode(query))
scores = docs_normalized @ query_normalized
```

2. Comparer vecteurs de modèles différents

```
# FAUX : Espaces vectoriels incompatibles
vec1 = model_bert.encode(text1)      # BERT 768D
vec2 = model_openai.encode(text2)    # OpenAI 1536D
similarity = cosine_similarity(vec1, vec2)  # ERREUR ou nonsense!

# CORRECT : Même modèle pour tout
vec1 = model.encode(text1)
vec2 = model.encode(text2)
similarity = cosine_similarity(vec1, vec2)
```

3. Oublier la dimension batch

```
# FAUX : Shape incorrecte
vector = model.encode("texte")  # Shape: (768,)
matrix = database  # Shape: (10000, 768)
scores = matrix @ vector  # Fonctionne par chance

# CORRECT : Explicite
vector = model.encode("texte")  # (768,)
scores = matrix @ vector  # (10000,) - OK
# OU avec reshape pour clarté :
scores = (matrix @ vector.reshape(-1, 1)).flatten()
```

4. Ne pas gérer les cas limites

```
def safe_normalize(vector, epsilon=1e-8):
    """Normalisation sécurisée"""
    norm = np.linalg.norm(vector)
    if norm < epsilon:
        # Vecteur quasi-nul : retourner zéros ou lever exception
        return np.zeros_like(vector)
    return vector / norm

def safe_cosine_similarity(v1, v2, epsilon=1e-8):
    """Similarité cosinus robuste"""
    norm1 = np.linalg.norm(v1)
    norm2 = np.linalg.norm(v2)
    if norm1 < epsilon or norm2 < epsilon:
        return 0.0
    return np.dot(v1, v2) / (norm1 * norm2)
```

5. Ignorer la version du modèle

```
# Sauvegarder métadonnées avec vecteurs
metadata = {
    'model_name': 'sentence-transformers/all-MiniLM-L6-v2',
    'model_version': '2.2.2',
    'dimension': 384,
    'normalized': True,
    'date': '2025-01-15'
}

# Si modèle change → ré-encoder TOUT le corpus
if current_model != metadata['model_name']:
    print("ATTENTION : Modèle changé, ré-indexation nécessaire")
```

Tests de validation

```
import numpy as np

def validate_embeddings(embeddings):
    """Tests de santé sur embeddings"""
    # 1. Pas de NaN/Inf
    assert not np.isnan(embeddings).any(), "NaN détectés"
    assert not np.isinf(embeddings).any(), "Inf détectés"

    # 2. Normes cohérentes (si normalisés)
    norms = np.linalg.norm(embeddings, axis=1)
    if np.allclose(norms, 1.0, atol=1e-5):
        print("✅ Vecteurs normalisés correctement")
    else:
        print(f"⚠️ Normes variables : min={norms.min():.3f}, max={norms.max():.3f}")

    # 3. Distribution raisonnable
    mean = embeddings.mean()
    std = embeddings.std()
    print(f"Mean : {mean:.3f}, Std : {std:.3f}")
    if abs(mean) > 0.1:
        print("⚠️ Mean éloignée de 0, standardisation recommandée")

    # 4. Pas de vecteurs zéro
    zero_vectors = (norms < 1e-6).sum()
    if zero_vectors > 0:
        print(f"⚠️ {zero_vectors} vecteurs quasi-nuls détectés")

validate_embeddings(your_embeddings)
```

Sources et références : ArXiv IA · Hugging Face Papers

Questions fréquentes

Quelle est la différence entre un vecteur et un embedding ?

Un **vecteur** est une structure mathématique générale (séquence de nombres), tandis qu'un **embedding** est un vecteur spécifique qui **encode la sémantique** d'une donnée (texte, image, audio). Tous les embeddings sont des vecteurs, mais tous les vecteurs ne sont pas des embeddings. Par exemple, [48.8566, 2.3522] est un vecteur (coordonnées GPS) mais pas un embedding sémantique, alors que le vecteur 768D de BERT encodant "intelligence artificielle" est un embedding car il capture le sens du texte.

Combien de dimensions faut-il pour un vecteur efficace ?

Cela dépend du cas d'usage et de la complexité des données :

- **64-128D** : Suffisant pour features simples, recommandation basique, prototypage
- **384-512D** : Bon compromis performance/coût pour NLP (Sentence-BERT MiniLM), recherche sémantique
- **768-1024D** : Standard industrie (BERT, GPT), qualité élevée
- **1536-3072D** : Qualité maximale (OpenAI, Cohere), systèmes critiques

En pratique, **384-768D offre le meilleur ratio qualité/coût** pour la majorité des applications. Au-delà de 1536D, les gains de précision sont marginaux (1-3%) mais les coûts doublent.

Comment visualiser des vecteurs de haute dimension ?

Les vecteurs 768D ne peuvent pas être visualisés directement. On utilise des techniques de **réduction de dimension** pour projeter en 2D ou 3D :

- **PCA** (Principal Component Analysis) : Projection linéaire rapide, préserve distances globales
- **t-SNE** : Non-linéaire, excellent pour visualiser clusters, mais lent (>5 min pour 10K points)
- **UMAP** : Similaire à t-SNE mais 10-100x plus rapide, préserve mieux structure globale

Exemple avec UMAP :

```
import umap
import matplotlib.pyplot as plt

# Réduire 768D → 2D
reducer = umap.UMAP(n_components=2, random_state=42)
vectors_2d = reducer.fit_transform(vectors_768d)

# Visualisation
plt.scatter(vectors_2d[:, 0], vectors_2d[:, 1], alpha=0.5)
plt.title("Visualisation UMAP des embeddings")
plt.show()
```

Attention : Ces projections perdent de l'information. Elles sont utiles pour exploration, mais ne représentent pas fidèlement toutes les relations en haute dimension.

Les vecteurs sont-ils utilisés uniquement pour le texte ?

Non, les vecteurs sont utilisés pour **tous les types de données** en IA :

- **Texte** : Embeddings (BERT, GPT), recherche sémantique, traduction
- **Images** : Reconnaissance (ResNet, EfficientNet), recherche visuelle, génération (Stable Diffusion)
- **Audio** : Reconnaissance vocale (Whisper), identification locuteur, classification sons
- **Vidéo** : Action recognition, recherche de scènes similaires
- **Graphes** : Node embeddings (GraphSAGE, Node2Vec) pour réseaux sociaux, molécules
- **Séries temporelles** : Détection d'anomalies, prévision
- **Code source** : CodeBERT, recherche de code similaire, détection bugs
- **Utilisateurs** : Recommandation, segmentation, churn prediction

Les **modèles multimodaux** (CLIP, ImageBind, Gemini) vont encore plus loin en encodant **plusieurs types de données dans le même espace vectoriel**, permettant des interactions cross-modales (chercher images avec texte, audio avec vidéo, etc.).

Peut-on convertir n'importe quel type de données en vecteur ?

Oui, **tout peut être vectorisé**, mais la qualité de la représentation dépend de la méthode utilisée et de la disponibilité de modèles entraînés :

Données structurées :

- **Numériques** : Directement utilisables comme vecteur [age, salaire, score, ...]
- **Catégorielles** : One-hot encoding ou entity embeddings (apprises)
- **Dates** : Cycliques (sin/cos) ou timestamps normalisés

Données non-structurées :

Pour approfondir, consultez les ressources officielles : Hugging Face, arXiv et ANSSI.

- **Texte** : Transformers (BERT, GPT) - très mature
- **Images** : CNN (ResNet, ViT) - très mature
- **Audio** : Wav2Vec, Whisper - mature
- **Vidéo** : VideoMAE, TimeSformer - en développement

Données spécialisées :

- **Molécules** : ChemBERTa, MolFormer (drug discovery)
- **Protéines** : ESM, ProtBERT (biologie)
- **ADN** : DNABERT, Nucleotide Transformer
- **Graphes** : GNN (Graph Neural Networks)

Principe général : Si des modèles de deep learning peuvent apprendre sur ces données, on peut en extraire des vecteurs (couches intermédiaires). La difficulté est d'avoir suffisamment de données d'entraînement et de puissance de calcul pour créer des embeddings de qualité.

Règle pratique

Si vous pouvez décrire des **similarités** entre vos données ("ces deux X sont similaires"), alors vous pouvez probablement les vectoriser efficacement. L'objectif des vecteurs est précisément de capturer ces relations de similarité dans un espace géométrique.

> **Ressources open source associées :**
>
> - awesome-cybersecurity-tools — Liste de 100+ outils de cybersécurité

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

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