Introduction aux Grands Modèles de Langage (LLMs)

Fondements théoriques, architecture Transformer et pratique

Dr. Ahmed NAIT CHABANE
Enseignant-chercheur, CESI LINEACT
anaitchabane@cesi.fr

ENSTA Bretagne

28 Novembre 2025

Objectifs pédagogiques

À l'issue du module, vous serez capables de :

- Expliquer le principe d'un **modèle supervisé** en deep learning.
- Discuter des **limites** des RNN / LSTM pour le NLP.
- Décrire l'architecture **Transformer** et le mécanisme d'**attention**.
- Manipuler un **LLM open-source** avec Hugging Face sous Jupyter.
- Mettre en œuvre un **prototype** simple autour d'un LLM.

Évaluation :

- Quiz final (QCM)
- Mini-projet Jupyter en binôme (rapport court + notebook)

Organisation générale du module

Jour 1 — Fondamentaux et démonstrations

- Contexte, définitions, objectifs du module.
- Rappel de l'apprentissage supervisé, fonction de perte et gradient.
- RNN / LSTM : principe, modélisation séquentielle et limites.
- Tokens, embeddings, représentation du texte.
- Notion d'attention et introduction au Transformer.
- Démos Jupyter : modèles simples + visualisation de scores d'attention.

Jour 2 — Pratique LLMs et mini-projet

- Pipeline LLM : pré-entraînement, fine-tuning, RLHF (vue d'ensemble).
- Atelier Hugging Face : génération, prompts, décodage.
- Esquisse de fine-tuning léger (LoRA / PEFT).
- Quiz final + lancement du mini-projet en binôme.

Plan détaillé du cours

Qu'est-ce que le Machine Learning ?

Définition selon Yann LeCun

Pour Yann LeCun, pionnier du Deep Learning, le Machine Learning vise à :

- apprendre des **représentations utiles** du monde ;
- optimiser une fonction permettant de **prédire**, **classifier** ou **agir** ;
- réduire la nécessité de règles explicites programmées par l'humain.

Citation clé (LeCun)

"If intelligence is the ability to predict,
then learning is the ability to get better at predicting."

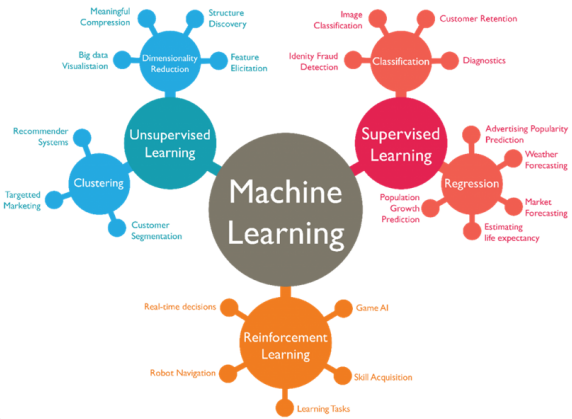
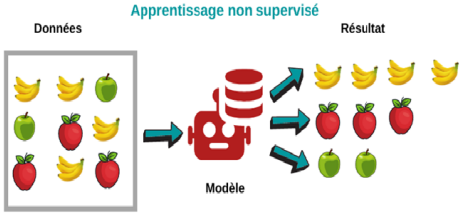
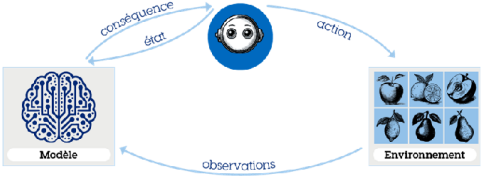


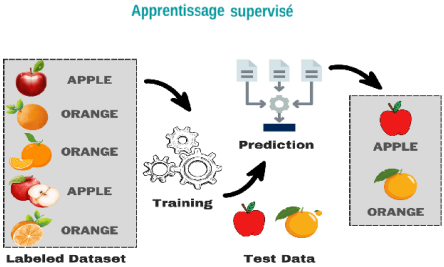
Illustration générale des familles du Machine Learning.

Panorama des familles d'apprentissage automatique





Apprentissage par renforcement



Supervisé, non supervisé et renforcement

Apprentissage supervisé

- Données annotées : couples (x, y).
- Objectif : apprendre une fonction $f : \mathcal{X} \to \mathcal{Y}$ qui **prédit** y à partir de x.
- Exemples : régression, classification (SVM, réseaux de neurones, arbres de décision, etc.).

Apprentissage non supervisé

- Données **non annotées** : seulement x.
- Objectif : découvrir une **structure** latente dans les données.
- Exemples : clustering (k-means), réduction de dimension (PCA), modèles de mélange.

Apprentissage par renforcement

- Un agent interagit avec un environnement et reçoit des **récompenses**.
- Objectif : apprendre une politique optimale $\pi^\star$ maximisant la récompense cumulée.

Modèle supervisé : rappel fonctionnel

- Données annotées : $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$.
- Modèle paramétré : $f_\theta : x \mapsto \hat{y}$.

$$\theta^\star = \arg\min_\theta \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\big(f_\theta(x^{(i)}), y^{(i)}\big).$$

- Optimisation : descente de gradient / Adam, etc.
- Notion de **généralisation** : performance hors échantillon.

Démonstration Jupyter

Ouvrir le notebook : `Mini-MLP.ipynb`

Réseaux séquentiels (RNN) : idée générale

Objectif : traiter une séquence $(x_1, \ldots, x_T)$ en maintenant un **état caché** qui résume le passé.

$$h_t = f(h_{t-1}, x_t), \quad t = 1, \ldots, T$$

- x_t : entrée au temps t (token, vecteur d'embedding, mesure, etc.).
- h_t : **état caché** contenant une information sur le passé $x_1, \ldots, x_t$.
- Sortie possible à chaque pas : $\hat{y}_t = g(h_t)$ (prédiction séquentielle).
- En **apprentissage supervisé**, les paramètres sont appris à partir de couples (séquence, cible) en minimisant une perte (entropie croisée, MSE, etc.).
- Applications : texte (NLP), séries temporelles, audio, signaux.

Idée clé : la séquence est transformée en une suite d'états cachés qui « transportent » la mémoire d'un pas de temps au suivant.

RNN simple : formulation mathématique

Formulation standard d'un RNN :

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$$

$$\hat{y}_t = g(W_y h_t + c)$$

- W_h : matrice de transition (mémoire de l'état caché).
- W_x : matrice appliquée à l'entrée x_t.
- W_y : matrice de projection vers l'espace de sortie.
- $f = \tanh$ ou ReLU : non-linéarité.

Interprétation supervisée

À chaque pas de temps, le RNN combine l'entrée courante x_t avec l'état précédent h_{t-1} pour produire un nouvel état h_t et une sortie $\hat{y}_t$. Les paramètres (W_h, W_x, W_y, b, c) sont appris en minimisant une perte supervisée sur un ensemble de séquences étiquetées.

RNN déroulé dans le temps

- Un même **bloc de neurones** est réutilisé à chaque pas de temps.
- Les paramètres (W_h, W_x, b) sont **partagés** pour tous les t.
- Le réseau peut en principe modéliser des dépendances longues, mais en pratique la mémoire reste limitée.

Démonstration Jupyter

Ouvrir le notebook

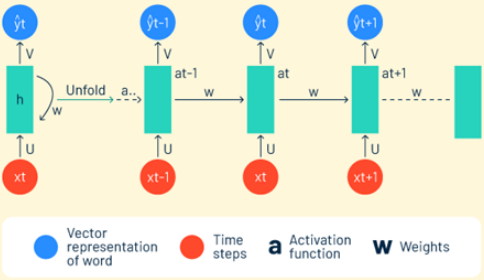


Schéma d'un RNN « déroulé » sur plusieurs pas de temps.

LSTM : motivation et principe

Problème des RNN simples :

- **Gradients qui s'annulent ou explosent** sur les longues séquences.
- Difficulté à maintenir une mémoire fidèle sur des dizaines / centaines de pas.

Idée des LSTM (Long Short-Term Memory) :

- Introduire une **cellule mémoire** c_t séparée de l'état caché h_t.
- Utiliser des **portes** (gates) pour contrôler :
  - ce qu'on **oublie** du passé (forget gate),
  - ce qu'on **ajoute** comme nouvelle information (input gate),
  - ce qu'on **expose** à la sortie (output gate).
- Mieux gérer les **dépendances à long terme**.

LSTM : architecture de la cellule

- Trois portes principales : **forget**, **input**, **output**.
- Voie presque linéaire pour la mémoire c_t (limite l'oubli excessif).
- Permet de modéliser des dépendances plus longues qu'un RNN simple.

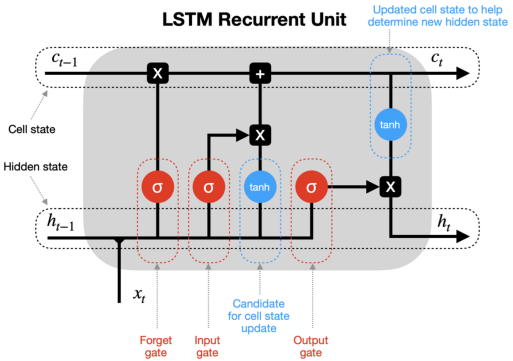


Schéma d'une cellule LSTM (portes + mémoire).

LSTM : équations (vue simplifiée)

Pour chaque pas de temps t, on calcule :

$$
\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) && \text{(forget gate)} \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) && \text{(input gate)} \\
\tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) && \text{(nouvelle info)} \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t && \text{(mise à jour mémoire)} \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) && \text{(output gate)} \\
h_t &= o_t \odot \tanh(c_t) && \text{(état caché exposé)}
\end{aligned}
$$

- σ : sigmoïde (valeurs entre 0 et 1, rôle de « filtre »).
- $\odot$: produit élément par élément.
- c_t : mémoire interne à long terme ; h_t : état caché (sortie).

Démonstration Jupyter

Ouvrir le notebook

GRU : Gated Recurrent Unit

Motivation : simplifier la structure LSTM tout en gardant une bonne capacité de mémoire à long terme.

- Fusionne certaines portes de LSTM en **deux portes** :
  - **reset gate** : part de l'information passée utilisée pour calculer la nouvelle candidate ;
  - **update gate** : compromis entre conserver h_{t-1} et intégrer la nouvelle information.
- Pas de cellule mémoire séparée (c_t), seulement l'état caché h_t.

GRU : équations (vue simplifiée)

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \qquad \text{(update gate)}$$
$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \qquad \text{(reset gate)}$$
$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

- Moins de paramètres qu'un LSTM, entraînement souvent plus simple.
- Performances comparables à LSTM sur de nombreuses tâches supervisées séquentielles.

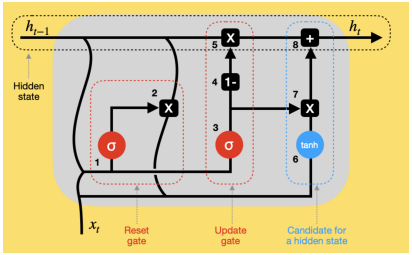


Schéma d'une cellule GRU (update & reset gates).

Synthèse RNN / LSTM / GRU

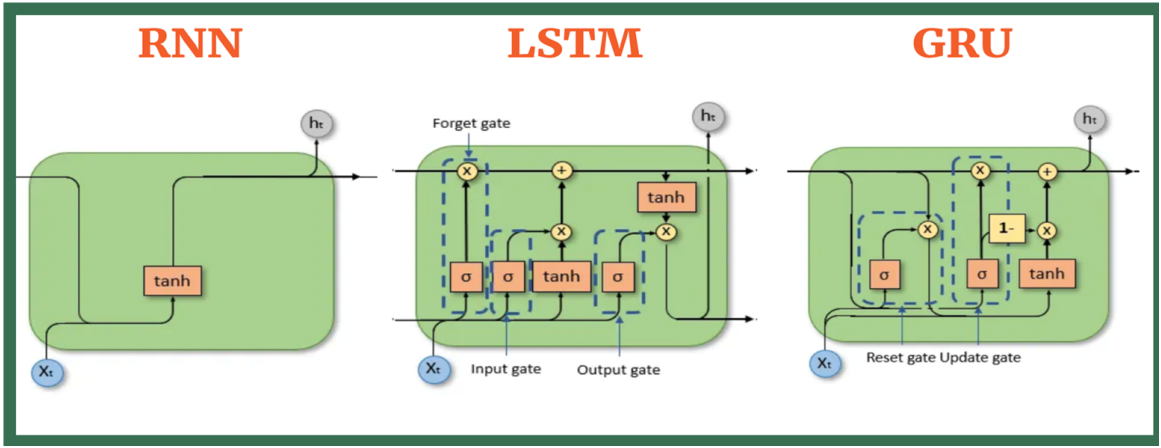


Figure – Architectures : RNN, LSTM, GRU pour l'apprentissage supervisé sur séquences.

Pourquoi les RNN/LSTM ne passent pas à l'échelle ? (1/2)

- T : longueur de la séquence (nombre de tokens)
- d : dimension interne (taille de l'état caché ou embedding)

Architecture	**Complexité**	**Conséquence**
RNN simple	$\mathcal{O}(T \cdot d^2)$	traitement strictement séquentiel
LSTM / GRU	$\mathcal{O}(T \cdot d^2)$	4–6 matrices par pas $\Rightarrow$ plus lent
Transformer (Self-attention)	$\mathcal{O}(T^2 \cdot d)$	parallélisable intégralement
Transformer (FlashAttention)	$\mathcal{O}(T \cdot d)$	optimisation GPU moderne

Implication pratique

Les LLMs modernes (GPT-3, LLaMA, Mistral...) nécessitent des contextes longs (4k–128k tokens) et des milliards de paramètres, ce qui est **incompatible avec la nature séquentielle** des RNN/LSTM/GRU.

Pourquoi les RNN/LSTM ne passent pas à l'échelle des LLMs ? (2/2)

Pourquoi cette complexité pose problème ?

- **Séquentialité RNN** : l'état h_t dépend de $h_{t-1} \Rightarrow$ **impossible de paralléliser sur GPU**. Exemple : pour $T = 4\,000$ tokens, chaque token doit être traité **dans l'ordre**.
- **Coût réel LSTM** : 4 projections linéaires $\Rightarrow$ coût $\approx 4\times$ un RNN.
- **Transformers** : calcul matriciel large $\Rightarrow$ 4 000 tokens traités en **un seul batch GPU**.

Illustration chiffrée (d = 2048, T = 4096) :

- LSTM : $\approx 4 \times Td^2 \approx 4 \times 4096 \times 2048^2 \approx 68 \times 10^9$ opérations.
- Self-attention GPU : $\approx T^2 d \approx 4096^2 \times 2048 \approx 34 \times 10^9$ opérations, **mais parallélisés**.

Conclusion : même avec plus d'opérations, le Transformer est **100–500× plus rapide** en pratique grâce au parallélisme massif.

Pourquoi s'intéresser aux LLMs ?

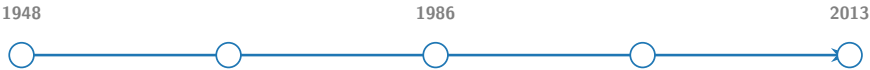
- Forte croissance des usages : chatbots, assistants de code, agents, résumés automatiques.
- Généralisation dans l'industrie, l'éducation et la recherche.
- Importance de comprendre les mécanismes plutôt que d'utiliser en boîte noire.

Évolution des modèles de représentation des mots (1948–2013)

Modèle n-gramme
Prédit le prochain élément dans une séquence à partir des $n-1$ éléments précédents.

Modèle de langage probabiliste neuronal
Apprend une représentation distribuée des mots pour le modèle de langage.

Word2vec
Méthode simple et efficace pour apprendre des représentations vectorielles continues des mots, utilisée dans de nombreux modèles de TAL.



Hypothèse distributionnelle
Un mot est caractérisé par la compagnie qu'il tient.

Sac de mots
Représente une phrase ou un document comme un ensemble non ordonné de mots.

Représentation distribuée
Représente chaque élément par un motif d'activation réparti sur plusieurs dimensions continues.

Démonstration Jupyter

Exemple Simple N-Gram

Des modèles pré-entraînés aux grands modèles de langage (2018–aujourd'hui)

Modèles de langage pré-entraînés
Représentations contextuelles des mots
(pré-entraînement massif puis affinage).
Corpora très larges et architectures profondes.

Modèles de très grande échelle
Milliards de paramètres (par ex. GPT-3, T5, etc.).
Pré-entraînement auto-supervisé et forte capacité
de transfert vers de nombreuses tâches de TAL.

Grands modèles de langage et modèles de fondation
Assistants conversationnels, raisonnement plus avancé,
instruction tuning, RLHF, interaction multi-modalité
(texte, image, audio) et intégration dans des agents
et outils pour des applications industrielles.

Démonstration Jupyter

Premier LLM (GPT-2)

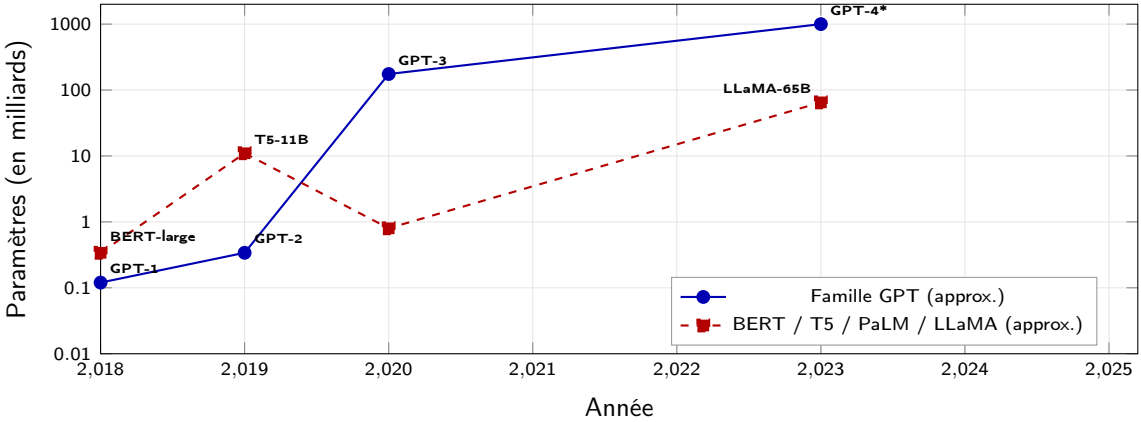
Modèle de langage : définition formelle

Un **modèle de langage** estime la probabilité d'une séquence de tokens $(x_1, \ldots, x_T)$:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t}).$$

- x_t : token (mot, sous-mot, caractère...).
- Objectif : assigner une probabilité élevée aux phrases « plausibles ».
- **LLM** : modèle de langage avec un très grand nombre de paramètres (ordre du milliard et au-delà).

Explosion de la taille des modèles de langage (2018–2025)



Message à retenir : en moins de 7 ans, on est passé de modèles à quelques centaines de millions de paramètres à des LLMs dépassant le **mille milliards** de paramètres (ordre de grandeur), ce qui change complètement :

- la **capacité de représentation** (mémoire des faits, raisonnement...),
- les besoins en **données**, **calcul** et **énergie**.

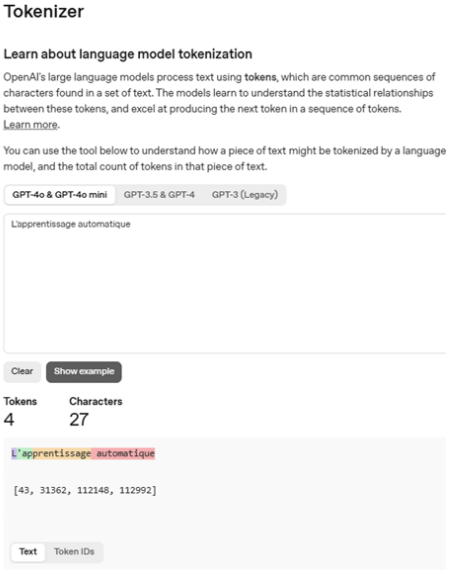
Tokenisation : pourquoi et comment découper le texte

Objectif : convertir un texte brut en une séquence de symboles (**tokens**) traitables par un modèle neuronal.

- Les modèles n'opèrent pas sur du texte → uniquement sur des entiers.
- Mapping nécessaire :

$$\text{Texte} \to (x_1, x_2, \ldots, x_T) \in \{1, \ldots, V\}$$

- Pourquoi pas un vocabulaire mot-à-mot ?
  - vocabulaire explosif ($> 10^6$ formes),
  - difficulté avec les mots rares / néologismes,
  - impossibilité multilingue.



Méthodes modernes de tokenisation (subwords)

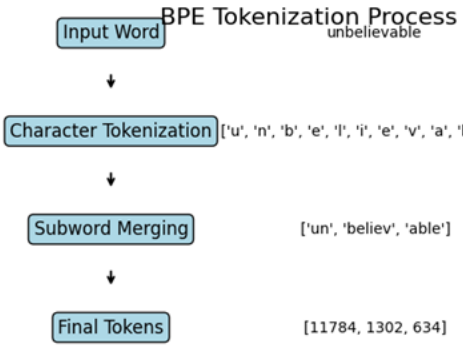
Idée clé : utiliser des unités mixtes (mots + sous-mots) pour équilibrer vocabulaire et expressivité.

- **BPE (Byte-Pair Encoding) — GPT-2/3**
- **WordPiece — BERT**
- **SentencePiece — LLaMA, T5**

Principe commun : apprendre les segments les plus fréquents.

Exemple :

"internationalisation" $\rightarrow$ [`inter`, `nation`, `alisation`]



Découpage visuel d'un mot en sous-unités (BPE)

Algorithme BPE : principe et déroulement étape par étape

Byte-Pair Encoding (BPE) apprend des sous-unités fréquentes pour réduire la taille du vocabulaire tout en restant robuste aux mots rares.

Étapes de l'algorithme :

1. **Initialisation** On représente chaque mot comme une séquence de **caractères**, ex. : `unbelievable` → [u, n, b, e, l, i, e, v, a, b, l, e].
2. **Comptage des paires** On compte toutes les paires adjacentes de symboles : (u,n), (n,b), (b,e), (e,l), (l,i), ...
3. **Fusion de la paire la plus fréquente** Exemple : si (e, l) est la plus fréquente → `e l` devient `el`. On met à jour toutes les occurrences dans le corpus.
4. **Itération** Répéter les étapes #2–3 jusqu'à atteindre une taille de vocabulaire V imposée (ex. GPT-2 : 50 000 unités).
5. **Tokenisation d'un nouveau mot** On applique les fusions apprises, du plus long sous-mot possible vers le plus court.

Exemple final : `unbelievable` → [`un`, `believ`, `able`]

$$[\texttt{un}] \to 11784, \quad [\texttt{believ}] \to 1302, \quad [\texttt{able}] \to 634$$

Démonstration Jupyter : tokenisation BPE réelle

Objectif : montrer la tokenisation d'un mot réel avec un tokenizer pré-entraîné.

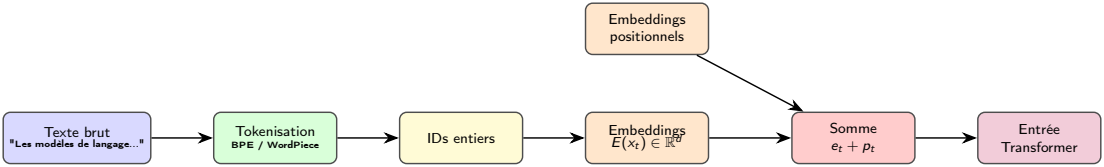
```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")

text = "internationalisation"
tokens = tokenizer.tokenize(text)
ids = tokenizer.encode(text)

print("Tokens :", tokens)
print("IDs    :", ids)
```

Faire varier la phrase « Les modèles de langage internationaux » et observer les tokens.

Démonstration Jupyter

Tokenisation BPE

Pipeline : du texte brut au Transformer



Pipeline : transformation du texte en tenseurs traitables par un Transformer.

Embeddings : vecteurs continus pour représenter les tokens

Objectif : remplacer les IDs entiers par des vecteurs réels optimisables.

$$e_t = E(x_t) \in \mathbb{R}^d$$

- E : matrice d'embedding de taille $V \times d$.
- x_t : ID du token.
- e_t : vecteur dense appris lors de l'entraînement.

Pourquoi des vecteurs ?

- calcul matriciel efficace sur GPU,
- capture de la sémantique,
- généralisation par similarité (mots proches dans l'espace vectoriel).

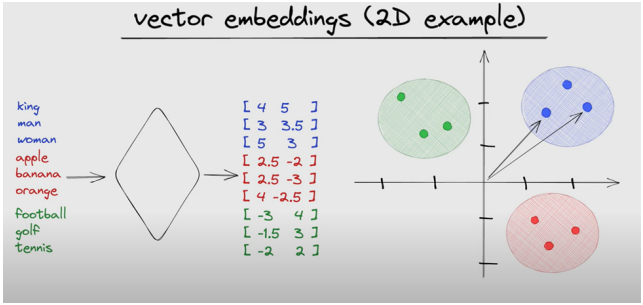


Illustration : tokens BPE projetés dans un espace continu.

Démonstration Jupyter : similarité entre embeddings

Objectif : illustrer que certains mots sont plus proches que d'autres dans l'espace d'embedding.

Démonstration Jupyter

Similarité entre Embeddings

Intuition de l'attention

Problème fondamental : lorsqu'un modèle lit une séquence, il doit déterminer **quels tokens du contexte sont utiles** pour comprendre ou prédire le token courant.

Idée clé : tous les tokens n'ont pas la même importance.

- À chaque position t, le modèle calcule **l'importance relative** de chaque autre token j.
- Cette importance est un **poids d'attention** (probabilité entre 0 et 1).
- Le modèle construit ensuite une représentation enrichie du token t en combinant les autres tokens selon ces poids.
- La self-attention permet au modèle de regarder **globalement toute la séquence en un seul passage** (contrairement aux RNN).

Avant les équations, voyons un exemple concret.

Exemple simplifié : comment un token regarde les autres

Phrase : *"Paris is the capital"*

Chaque ligne = token **Query** (celui qui regarde). Chaque colonne = token **Key** (celui qui est regardé). Chaque case = **poids d'attention** après softmax.

Query ↓ / Key →	**Paris**	**is**	**the**	**capital**
Paris	0.60	0.20	0.10	0.10
is	0.25	0.50	0.15	0.10
the	0.15	0.20	0.40	0.25
capital	0.10	0.10	0.30	0.50

Table – *

Exemple fictif : l'attention montre quelles relations la phrase encode.

Observation : "capital" porte beaucoup d'attention à "the" (relation grammaticale), alors que "Paris" s'auto-attend fortement (0.60).

Matrices Q, K, V : version essentielle

Séquence encodée :

$$X \in \mathbb{R}^{T \times d} \quad \text{(T = longueur, d = dimension des embeddings)}$$

Projections linéaires :

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

Dimensions :

$$W^Q, W^K, W^V \in \mathbb{R}^{d \times d_k}, \quad Q, K, V \in \mathbb{R}^{T \times d_k}$$

Rôles :

- W^Q, W^K, W^V : matrices **apprises** (poids du modèle).
- Q = **Queries** : ce que chaque token cherche.
- K = **Keys** : ce que chaque token offre comme indice.
- V = **Values** : l'information transmise.

Self-attention : formulation mathématique

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

Interprétation des étapes :

- $QK^\top$: scores de similarité entre tokens (réels, positifs ou négatifs).
- $\sqrt{d_k}$: facteur de normalisation (évite des valeurs trop grandes).
- **softmax** : transforme chaque ligne en **probabilités**

$$\text{softmax}(s_j) = \frac{e^{s_j}}{\sum_i e^{s_i}}$$

- Résultat : chaque ligne contient des **poids** qui somment à 1.
- Multiplication par V : combinaison pondérée des informations.

Mini-calcul d'attention sur 3 tokens

Exemple jouet : 3 tokens dans une phrase

$$\text{tokens} = [t_1, t_2, t_3]$$

On choisit des vecteurs Q, K, V très simples (ici dimension $d_k = 2$) :

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

Étapes :

1. Scores bruts : $S = \dfrac{QK^\top}{\sqrt{d_k}} \in \mathbb{R}^{3\times 3}$.
2. Poids d'attention : $A = \mathrm{softmax}$ *ligne par ligne* sur S.
3. Sortie : $\mathrm{Attention}(Q, K, V) = AV$ (combinaison pondérée des V).

Chaque ligne de A = distribution d'attention d'un token vers les 3 tokens.

Démonstration Jupyter : mini-calcul d'attention

Objectif : calculer une self-attention sur 3 tokens, à la main.

Démonstration Jupyter

Mini-calcul d'attention

Commenter les matrices : chaque ligne = distribution d'attention d'un token.

Mini-calcul d'attention : lecture des matrices

Scores bruts (similaires à $QK^\top/\sqrt{d_k}$) :

$$S = \begin{bmatrix} 0.707 & 0.707 & 0.000 \\ 0.000 & 0.707 & 0.707 \\ 0.707 & 1.414 & 0.707 \end{bmatrix}$$

Poids d'attention (après softmax ligne par ligne) :

$$A = \begin{bmatrix} 0.401 & 0.401 & 0.198 \\ 0.198 & 0.401 & 0.401 \\ 0.248 & 0.503 & 0.248 \end{bmatrix}$$

Chaque ligne de A est une **distribution d'attention** pour un token : par ex. la ligne 3 montre que t_3 regarde surtout t_2 (0.503).

Pourquoi plusieurs têtes d'attention ?

Limite : une seule tête d'attention regarde la séquence **dans un seul espace de projection**.
Solution : le Transformer utilise plusieurs têtes (multi-head) pour capturer **plusieurs types de relations** en parallèle.

- Une tête = un **point de vue** sur la séquence (dans un espace de dimension d_k).
- Problème : une seule tête doit tout capturer (syntaxe, sémantique, co-références...).
- Solution : utiliser H têtes différentes, chacune avec ses poids W_h^Q, W_h^K, W_h^V.
- Chaque tête peut se spécialiser sur un type de relation.

Multi-head attention et bloc Transformer

Idée : une seule tête d'attention ne regarde la séquence que dans **un seul espace**. La **multi-head attention** utilise plusieurs projections en parallèle.

- Pour chaque tête h :

$$\text{head}_h = \text{Attention}(QW_h^Q,\ KW_h^K,\ VW_h^V)$$

- Chaque tête apprend à capturer des **relations différentes** (syntaxe, co-références, dépendances longues, etc.).
- On concatène ensuite les têtes :

$$\text{MHA}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)W^O$$

- Un bloc Transformer typique :
  - Multi-head attention,
  - Connexion résiduelle + normalisation,
  - Réseau feed-forward positionnel,
  - Nouvelle normalisation.

Du bloc Transformer au modèle de langage

Récapitulatif :

- On dispose maintenant d'un bloc Transformer :
  - embeddings + position,
  - multi-head attention,
  - feed-forward + résidus + normalisation.
- En empilant plusieurs blocs, on obtient un **encodeur/décodeur profond**.
- Pour un modèle de type GPT, on ajoute à la sortie :
  - une projection linéaire vers le vocabulaire ($\mathbb{R}^d \to \mathbb{R}^V$),
  - un softmax pour obtenir $P_\theta(x_t \mid x_{<t})$.

Question suivante : comment entraîne-t-on ce modèle ? $\to$ *objectif de langage causal, pré-entraînement, RLHF.*

Objectif de langage causal (modèles type GPT)

Pour un modèle auto-régressif (type GPT), construit à partir de blocs Transformer, on cherche à minimiser la perte :

$$\mathcal{L}_{\mathrm{LM}}(\theta) = -\sum_{t=1}^{T} \log P_\theta(x_t \mid x_{<t}).$$

- On maximise la probabilité de la séquence observée (entropie croisée).
- En pratique : moyenne sur un **batch** de séquences et sur toutes les positions.
- Le **pré-entraînement** utilise de très grands corpus non annotés (web, livres, code, etc.).

Pipeline LLM : du pré-entraînement à l'alignement

Objectif global : partir d'un modèle brut et le rendre utile et aligné.

1. **Pré-entraînement (Language Modeling)**
   - Tâche : prédire le prochain token x_t à partir de $x_{<t}$.
   - Données : corpus massif non annoté (web, livres, code...).
   - Résultat : modèle généraliste, mais non aligné sur l'usage humain.
2. **Instruction tuning (Supervised Fine-Tuning)**
   - Tâche : répondre à des **instructions** données par l'utilisateur.
   - Données : paires (instruction, réponse) annotées.
   - Résultat : modèle plus adapté aux formats "assistant".
3. **RLHF (Reinforcement Learning from Human Feedback)**
   - Tâche : privilégier les réponses **préférées par les humains**.
   - Données : comparaisons de réponses, modèle de récompense, RL.
   - Résultat : modèle **aligné** (plus utile, moins toxique).

Zoom sur RLHF : aligner le modèle sur l'humain

Étapes principales :

- **1. Génération de réponses candidates**
  - On demande au LLM (déjà pré-entraîné + instruction-tuned) de produire plusieurs réponses pour un même prompt.
- **2. Recueil de préférences humaines**
  - Des annotateurs humains comparent les réponses (A vs B, C...).
  - On entraîne un **modèle de récompense** à prédire ces préférences.
- **3. Apprentissage par renforcement (ex. PPO)**
  - Le LLM est ajusté pour **maximiser** la récompense prédite.
  - Cela pénalise les réponses jugées mauvaises ou dangereuses.

Idée clé : RLHF corrige le comportement du modèle au-delà du simple "next token prediction".

Ce qu'il faut retenir (1/3) : Machine Learning et séquences

Machine Learning (rappel) :

- On apprend une fonction f_θ à partir de données annotées (x, y).
- L'objectif est de **généraliser** sur de nouvelles données.

RNN / LSTM / GRU :

- Adaptés aux **données séquentielles** (texte, séries temporelles, audio).
- Mémorisent le passé via un **état caché** mis à jour pas à pas.
- LSTM/GRU améliorent les RNN simples (gestion des dépendances longues).
- Mais restent **séquentiels** $\Rightarrow$ difficiles à paralléliser, limités pour les LLMs.

Ce qu'il faut retenir (2/3) : Tokens, embeddings, attention

Tokens et représentation du texte :

- Le modèle ne voit pas du texte mais une **séquence d'entiers** (tokens).
- La tokenisation par **subwords** (BPE, WordPiece, SentencePiece) gère mieux les mots rares et le multilingue.
- Les **embeddings** projettent les tokens dans un espace continu où la proximité a un sens sémantique.

Attention et Transformer :

- La **self-attention** permet à chaque token de regarder toute la séquence.
- Les matrices Q, K, V codent **ce qu'on cherche, ce qu'on offre, ce qu'on transmet**.
- La **multi-head attention** capture plusieurs types de relations en parallèle.
- Architecture Transformer = **parallélisable** et scalable $\Rightarrow$ base des LLMs modernes.

Ce qu'il faut retenir (3/3) : Pipeline LLM et pratique

Modèle de langage (LLM) :

- Apprend à modéliser $P(x_t \mid x_{<t})$ sur des corpus massifs.
- La taille des modèles a explosé (de millions à $\sim 10^{12}$ paramètres).

Pipeline d'entraînement :

- **Pré-entraînement** : apprentissage auto-supervisé sur du texte brut.
- **Instruction tuning** : ajustement supervisé sur des paires (instruction, réponse).
- **RLHF** : alignement sur les **préférences humaines** via un modèle de récompense.

Pour vous, à l'issue du module :

- Comprendre les briques clés derrière les LLMs (tokenisation, Transformer, attention).
- Être capable de **manipuler un LLM open-source** sous Jupyter avec Hugging Face.
- Concevoir un **mini-projet** LLM reproductible.

TP1 : Découvrir un modèle pré-entraîné

Objectif : prendre en main un pipeline de génération Hugging Face.

Travail demandé :

- Installer `transformers`.
- Créer un pipeline de génération :

```
from transformers import pipeline
gen = pipeline("text-generation", model="gpt2")
```

- Tester au moins 3 prompts :
  - question ouverte,
  - phrase à compléter,
  - instruction brève.
- Faire varier :
  - `max_length`,
  - `temperature`,
  - `do_sample`.

À rendre : outputs générés + commentaire personnel (3–4 lignes).

TP2 : Stratégies de génération

Objectif : observer l'effet des techniques de décodage.

Travail demandé :

- À partir d'un même prompt :
  - Greedy decoding (sans sampling),
  - Sampling classique (temperature = 0.7),
  - Top-k (k=10, 50, 100),
  - Top-p (p=0.9, 0.95).

```
print(gen(prompt, max_length=40, do_sample=False))
print(gen(prompt, do_sample=True, temperature=0.7))
print(gen(prompt, do_sample=True, top_k=50))
print(gen(prompt, do_sample=True, top_p=0.9))
```

À rendre :

- Tableau comparatif.
- Analyse personnelle : quel mode est le plus cohérent, le plus créatif ?

TP3 : Prompt Engineering

Objectif : comprendre l'impact du prompt sur le comportement du LLM.

Travail demandé :

- Choisir un sujet (résumé, explication, classification simple...).
- Construire trois types de prompts :
  - **Zero-shot** : simple instruction.
  - **Few-shot** : 2 exemples (Q→R).
  - **Prompt structuré** (liste, tableau, JSON, pseudo-code).
- Exécuter le modèle pour les trois cas.

À rendre :

- Les 3 prompts,
- Les 3 réponses du LLM,
- Analyse personnelle (5 lignes max) : quel prompt est le plus efficace ?

Projet final en binôme : Mini-projet LLM

Objectif : développer un petit cas d'usage reproductible avec un LLM.

Sujet au choix :

- résumé automatique,
- extraction d'information,
- classification simple,
- assistant pédagogique,
- génération guidée (liste, JSON, pseudo-code).

Livrables :

- **Notebook Jupyter** : code exécuté + analyses.
- **Rapport PDF (2 pages)** : objectif, méthodo, résultats, limites.

Installation requise pour les notebooks Jupyter

Pour exécuter tous les notebooks du module, vous devez installer :

1) Python 3.10 ou 3.11

- Recommandé : **Anaconda** ou **Miniconda**
- Alternative : Python officiel + `venv`

2) Bibliothèques Python utilisées dans les TPs :

```
pip install jupyter transformers datasets torch
pip install matplotlib numpy pandas
pip install accelerate sentencepiece
pip install huggingface_hub
```

Détails :

- `transformers` : manipulation des modèles Hugging Face (pipelines, tokenizers, MHA...)
- `datasets` : chargement de jeux de données (optionnel mais utile)
- `torch` : backend deep learning des modèles (CPU/GPU)
- `accelerate` : optimisation et accélération (optionnel)
- `sentencepiece` : tokenizers utilisés par LLaMA/T5

Merci pour votre attention !

Dr. Ahmed NAIT CHABANE

Enseignant-chercheur, CESI LINEACT

Contact : anaitchabane@cesi.fr

Slides et notebooks Jupyter seront disponibles sur :

dépôt GitHub:

https://github.com/anaitchabane/Cours-ENSTA-Bretagne

Expérience 1 — Courbes d'apprentissage d'un MLP

Objectif : étudier empiriquement la **généralisation** d'un modèle supervisé.
Travail demandé :

- Générer un dataset 2D simple (ex. 2 classes gaussiennes) avec `sklearn.datasets.make_moons` ou `make_classification`.
- Entraîner un **MLP** (PyTorch ou `sklearn.neural_network.MLPClassifier`).
- Faire varier la **taille du dataset d'entraînement** (ex. 100, 500, 1 000 points).
- Pour chaque taille, mesurer :
  - la perte / accuracy sur **train**,
  - la perte / accuracy sur **validation**.
- Tracer les **courbes d'apprentissage** (erreur train vs val en fonction du nombre d'exemples).

Analyse attendue : discussion sur **sous-apprentissage** / **surapprentissage** et rôle de la quantité de données.

Expérience 2 — Effet de la régularisation

Objectif : quantifier l'impact de la **régularisation** sur un modèle supervisé.
Travail demandé :

- Reprendre le MLP de l'Expérience 1.
- Tester 3 réglages :
  1. sans régularisation (L2=0, pas de dropout),
  2. L2 modérée (ex. `weight_decay = 1e-4`),
  3. fort dropout (ex. `p = 0.5`).
- Pour chaque réglage :
  - tracer les courbes de perte train/val en fonction des epochs,
  - tracer la frontière de décision finale.

Analyse attendue : relier visuellement la **capacité du modèle**, la régularisation et le **comportement de la frontière de décision**.

Expérience 3 — Gradient qui s'annule dans un RNN

Objectif : illustrer expérimentalement le **vanishing gradient**.
Travail demandé :

- Construire un **RNN simple** en PyTorch pour une tâche jouet : prédire le dernier élément d'une séquence binaire.
- Faire varier la **longueur de séquence** T (ex. 10, 20, 50, 100).
- Pendant l'entraînement, enregistrer la **norme du gradient** $\|\nabla_\theta \mathcal{L}\|$ en fonction de T.
- Tracer :
  - la norme moyenne du gradient vs T,
  - l'accuracy vs T.

Analyse attendue : montrer que pour des séquences longues, le RNN a du mal à apprendre $\Rightarrow$ justification empirique des LSTM/Transformers.

Expérience 4 — Comparer n-gramme vs GPT-2 (perplexité)

Objectif : comparer un **modèle n-gramme classique** et un **LLM** sur une petite tâche de modélisation de langage.

Travail demandé :

- Construire un mini-corpus (quelques dizaines de phrases en français ou anglais).
- Implémenter un **modèle trigramme** :
  - estimation des probabilités $P(w_t \mid w_{t-2}, w_{t-1})$,
  - lissage simple (add-k, par ex. $k = 1$).
- Utiliser GPT-2 (pipeline `"text-generation"` ou `AutoModelForCausalLM`) pour calculer la **log-vraisemblance** des mêmes phrases.
- Calculer et comparer la **perplexité** :

$$\mathrm{PPL} = \exp\left(-\frac{1}{N}\sum_{t=1}^{N}\log P(w_t)\right)$$

Analyse attendue : discussion sur la différence de perplexité et les **limitations structurelles** des n-grammes.

Expérience 5 — Distribution de la longueur en tokens

Objectif : caractériser statistiquement la tokenisation BPE sur un corpus.
Travail demandé :

- Choisir un texte (ou plusieurs pages web, articles scientifiques, etc.).
- Utiliser `AutoTokenizer.from_pretrained("gpt2")` ou un tokenizer français.
- Pour chaque phrase :
  - mesurer la longueur en **mots**,
  - mesurer la longueur en **tokens BPE**.
- Tracer :
  - un histogramme des longueurs de phrases (en mots),
  - un histogramme des longueurs de phrases (en tokens),
  - un nuage de points (mots vs tokens).

Analyse attendue : implication pour la **fenêtre de contexte** des LLMs (combien de phrases typiques dans 4k / 8k tokens ?).

Expérience 6 — Visualiser des têtes d'attention

Objectif : interpréter **empiriquement** les matrices d'attention d'un Transformer.
Travail demandé :

- Charger un petit modèle (ex. `"bert-base-multilingual-cased"` ou `"distilbert-base-uncased"`).
- Activer `output_attentions=True`.
- Choisir quelques phrases :
  - avec co-référence (« Le médecin a vu sa patiente. Il lui a parlé longtemps. »),
  - avec structure syntaxique claire.
- Pour 1–2 couches et 1–2 têtes :
  - tracer les **heatmaps** d'attention (token $\rightarrow$ token),
  - commenter les patterns (auto-attention sur le token, attention sur le verbe, etc.).

Analyse attendue : relier qualitativement les têtes à des **rôles linguistiques** (syntaxe, co-référence, proximité locale...).

Expérience 7 — Coût de la self-attention vs RNN

Objectif : mesurer empiriquement la **complexité temporelle**.
Travail demandé :

- Implémenter :
  - un bloc **RNN** simple,
  - un bloc **self-attention** (simple, sans multi-head).
- Générer des séquences aléatoires de dimension d et longueur T (ex. $T = 32, 64, 128, 256, 512, 1\,024$).
- Pour chaque T :
  - mesurer le **temps moyen d'inférence** sur 100 passes pour le RNN,
  - idem pour la self-attention (sur GPU si disponible).
- Tracer temps (ms) vs T en échelle log-log.

Analyse attendue : comparer empirique vs théorie : RNN en $\mathcal{O}(T)$, self-attention en $\mathcal{O}(T^2)$ mais hautement parallélisable.

Expérience 8 — LLM en zero-shot vs classifieur supervisé

Objectif : comparer une approche **LLM prompté** à un **classifieur classique**.
Travail demandé :

- Choisir une petite tâche de **classification de texte** : ex. sentiment (positif / négatif) ou sujet (tech / sport / politique).
- Approche 1 : **LLM zero-shot**
  - Construire un prompt du type : « Texte : ...
    Sentiment (positif ou négatif) : »
  - Interroger un LLM open-source (ex. `"gpt2"` ou autre).
  - Extraire la classe prédite (par parsing simple).
- Approche 2 : **classifieur supervisé**
  - Construire des embeddings (TF-IDF ou embeddings de phrase).
  - Entraîner une régression logistique ou SVM.
- Comparer les accuracy sur un petit set de test.

Analyse attendue : avantages / limites du **zero-shot** par rapport au **supervisé classique** sur petit dataset.

Expérience 9 — Effet de la température sur l'entropie

Objectif : relier **température de génération** et **incertitude** du modèle.
Travail demandé :

- Choisir un LLM (ex. GPT-2) et un prompt fixe (ex. début d'histoire).
- Pour chaque température $T \in \{0.1, 0.5, 1.0, 1.5\}$:
  - récupérer, pour un token donné, la distribution de probabilité $p(\cdot)$ sur le vocabulaire,
  - calculer l'**entropie** :

$$H(p) = -\sum_i p_i \log p_i$$

  - générer plusieurs suites de texte et les comparer qualitativement.

Analyse attendue : montrer que T contrôle **directement** l'entropie (diversité) des sorties, et donc la créativité / cohérence du modèle.

Expérience 10 — Petit benchmark de prompts

Objectif : quantifier l'impact du **prompt engineering**.
Travail demandé :

- Choisir une tâche simple (ex. classification, résumé court, explication d'un concept).
- Construire un **jeu de 20 entrées** (20 phrases / textes).
- Définir au moins 3 prompts :
  1. **Zero-shot** simple,
  2. **Few-shot** (2–3 exemples),
  3. **Format structuré** (liste, puces, JSON, etc.).
- Évaluer la qualité des réponses (grille simple : 0 = incorrect, 1 = partiel, 2 = correct).

Analyse attendue : calculer une **moyenne de score** par prompt et discuter des **bonnes pratiques de formulation**.