Plonger au cœur des “LLM” sans faire de maths

Ce document :

- introduit des termes clés de l'**Intelligence Artificielle Générative** (IAG) ;
- se concentre principalement sur les **Grands Modèles de Langage** (GML) ou, en anglais, **Large Language Model** (LLM) ;
- présente des concepts importants à l’intérieur de la “boîte noire” des LLM (1) ;
- se limite à la génération de texte (sans aborder la génération d’images par exemple).

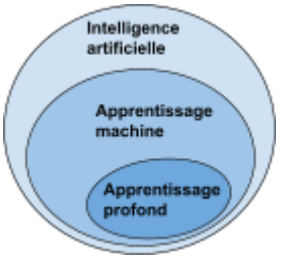
Sommaire

Préambule

Dans le contexte de l'intelligence artificielle (IA), un **modèle** est une **représentation mathématique** qui, pour effectuer une tâche spécifique, apprend à partir de données.
L'IA est la science et l'ingénierie de la fabrication de machines intelligentes, en particulier de programmes informatiques intelligents. C’est un concept qui englobe deux sous-ensembles :

L’apprentissage machine, aussi appelé apprentissage automatique (***machine learning***), permet une fois le modèle entraîné :

- d’utiliser les connaissances acquises à partir de données, **sans nécessiter de programmation** explicite supplémentaire ;
- d’utiliser des modèles pour analyser de nouvelles données **structurées** ou **semi-structurées** lorsque les données fournies ressemblent suffisamment aux données d'entraînement ;
- de produire des résultats sous forme de **prédictions** ou de **classifications** basées sur son apprentissage.



L’apprentissage profond (***deep learning***) est un sous-ensemble de l’apprentissage machine qui permet de traiter des données **non structurées** (comme du texte) et qui, pour ce faire, utilise des **réseaux de neurones multicouches**, utilisés dans les LLM, et que l’on va expliquer plus loin**.**

(1) Dans ce document, la traduction des termes s’appuie sur le site Datafranca. Le choix a été fait de conserver certains termes en anglais lorsqu’ils sont largement usités dans la littérature française.

Les grands modèles de langage

Dans le cas des **LLM**, l'apprentissage profond permet d'apprendre à traiter et à générer du langage naturel en analysant et en étudiant de très grandes quantités de textes. Il permet de capturer les relations entre les données avec lesquelles il s'est entraîné afin de **prédire** ou de **générer** du **nouveau contenu** en réponse à un texte fourni en entrée par l'utilisateur. On parle d'**IA générative**.

Ces modèles utilisent des **réseaux de neurones**. Cette technique repose sur **l'analyse probabiliste** des modèles linguistiques, où le système identifie les relations **statistiques** entre les mots et leurs contextes d'utilisation. En parallèle, cette technique intègre des caractéristiques **structurelles et sémantiques du langage humain**. Cela permet aux modèles de générer des réponses adaptées aux **nuances contextuelles** et aux **intentions sous-jacentes.**

Dans les systèmes d'IAG, il est important de distinguer deux phases :

- **La phase d'entraînement** durant laquelle **les paramètres** du modèle mathématique sont initialisés. Ces paramètres, internes au système, sont des valeurs numériques qui sont ajustées tout au long de cette phase, par le système lui-même, grâce à un volume important de données. Nous verrons plus tard que ces paramètres **ne bougent plus** une fois l'entraînement terminé.
- La phase **d'exploitation**, également appelée phase **d'inférence**, est la phase durant laquelle l'utilisateur transmet un texte au LLM qui génère un texte en retour.

Si aujourd'hui les LLM sont perçus comme une boîte noire, il est possible d'en illustrer macroscopiquement leur fonctionnement en 3 étapes (cf. figure 1). L'objectif de ce document est d'éclairer ces grandes étapes de traitement de la **phase d'inférence** entre le **prompt** fourni **en entrée** du LLM par l'utilisateur et le **texte généré** en sortie.

Nous expliquerons également comment les informations **initialisées lors de l'entraînement** (comme le vocabulaire, les token embeddings et les paramètres) sont **utilisées durant la phase d'inférence**.

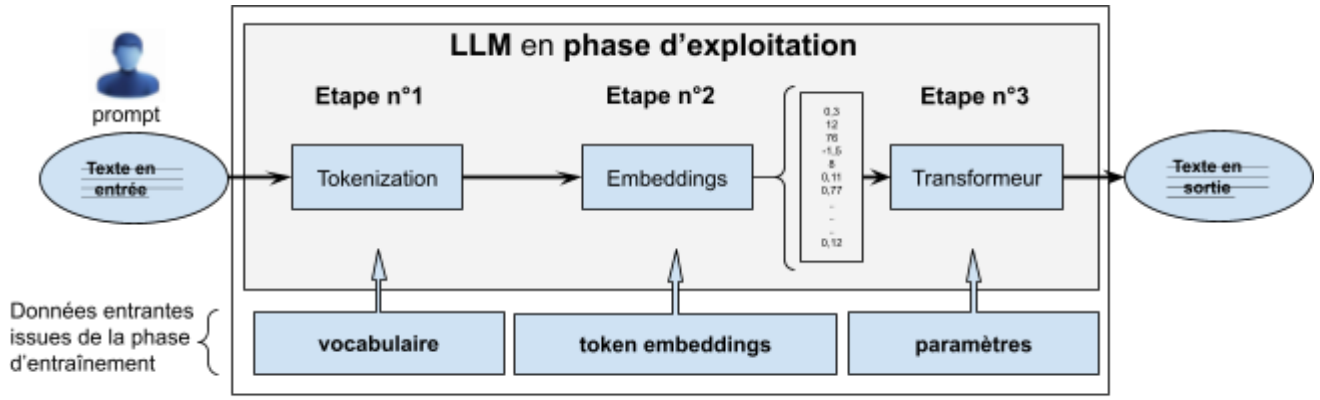


Figure 1 : **Boite noire du LLM**

Etape n°1 : Découpage en segments textuels : la tokenization

La première étape du LLM que nous allons décrire est la **tokenization**. Cette étape permet de découper le texte (en anglais “**prompt**”) fourni par l’utilisateur de telle sorte que le système puisse en établir des caractéristiques **structurelles et sémantiques**. Dans la **phase d’exploitation**, afin de pouvoir analyser le contenu du texte saisi par l’utilisateur, un **tokenizer** le découpe **en morceaux**, appelés **tokens**. Cette étape de découpage s’appelle la **segmentation**, ou encore la **tokenization** en anglais. Dans l’absolu, un token peut être un **ensemble de mots**, un **mot**, un **morceau de mot**, ou même un **simple caractère**, d’où sa traduction française : **segment textuel**.

Un **identifiant numérique** est associé à chaque token. A l’issue du découpage du prompt, le tokenizer fournira, en sortie, une liste d’identifiants qui sera la donnée d’entrée de l’étape n°2.

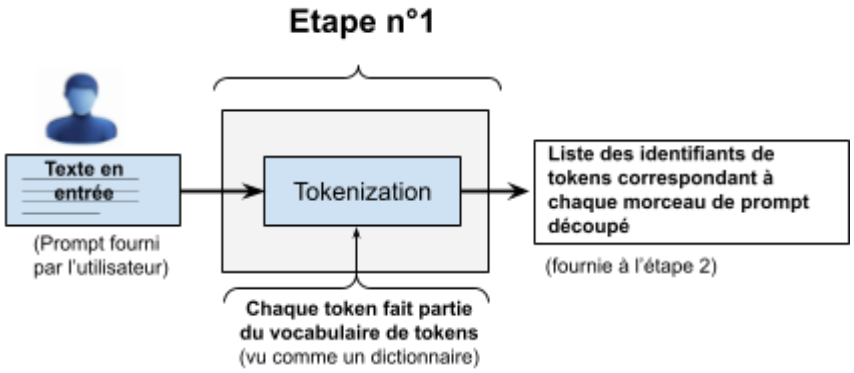


Figure 2 : **La tokenization**

Pour illustrer la **segmentation** en **token** d’un **texte en entrée**, il existe un site web (cf. figure 3) pour vous permettre de voir par vous même le résultat de cette segmentation. Ce site web permet d’illustrer les différents tokenizers utilisés par plusieurs dizaines de modèles **en phase d’exploitation** : https://tiktokenizer.vercel.app (testez le, c’est très visuel !)

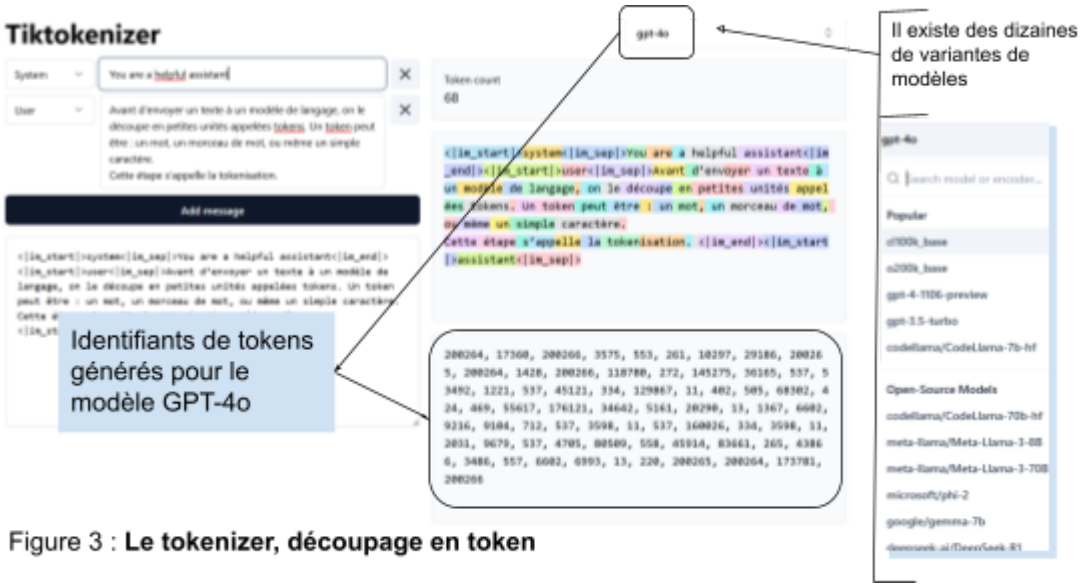


Figure 3 : **Le tokenizer, découpage en token**

Après avoir illustré ce qu'il se passe dans la **phase d'exploitation,** nous allons voir que ces tokens font partie d'un ensemble de tokens, appelé le **vocabulaire**. On peut voir cela comme un dictionnaire de tokens généré durant la **phase d'entraînement** du tokenizer grâce à une grosse quantité de documents.
Dans cette phase d'entraînement du tokenizer, il existe plusieurs **méthodes de découpage**. Une méthode connue pour cette tâche est le **Byte Pair Encoding** (BPE) utilisée par le modèle GPT, ou encore la méthode **SentencePiece** utilisée par les modèles Mistral et LLama. Ces méthodes diffèrent principalement dans leur approche pour segmenter le texte en tokens, notamment en termes de granularité de la segmentation, de traitement des espaces et des caractères spéciaux. Pour aller plus loin, voir l'article *Neural Machine Translation of Rare Words*.
Quelle que soit la méthode, ce qu'il faut retenir, c'est que le **vocabulaire généré** à l'issue de l'entraînement du tokenizer est **spécifique à ce tokenizer**. Le vocabulaire obtenu est composé d'un ensemble de tokens avec chacun **un identifiant associé**. Les **tokens sont uniques** et grâce à des méthodes comme BPE, ils peuvent refléter les **particularités linguistiques** et la **distribution des mots** dans le corpus d'entraînement.
Au début de l'entraînement, on considère un ensemble de tokens contenant initialement les **caractères de base** (lettres, chiffres, ponctuations, etc.). Au fil de l'entraînement, l'algorithme identifie des **associations de tokens** qui apparaissent fréquemment dans les textes utilisés lors de l'entraînement. Par exemple, le système remarque que les lettres "q" et "u" sont souvent juxtaposées dans des mots comme "question", "qualité", etc. Le système décide alors de fusionner ces deux tokens pour créer un **nouveau token unique** "qu". Dans cet exemple, plutôt que d'avoir deux tokens séparés "q" et "u", nous avons maintenant un nouveau token "qu" qui est ajouté à notre vocabulaire, et ainsi de suite.
Ce n'est qu'à l'issue de la phase d'entraînement que **l'intégralité des tokens disponibles** dans le **vocabulaire** est établie avec un identifiant unique pour chaque token. Cet entraînement n'est réalisé qu'une seule fois afin de produire le vocabulaire du tokenizer. La méthode est la même quelles que soient la ou les langues traitées par le tokenizer.
À noter également qu'une fois la phase d'entraînement du tokenizer terminée, le **nombre final de tokens** dans le dictionnaire **ne change plus**. Cette quantité de tokens devient une **constante du modèle**.

A noter : le vocabulaire du tokenizer du modèle **Llama 2** est de **32 000 tokens**.
Pour le modèle **Llama 3**, il y a **250 000 tokens** dans le vocabulaire.

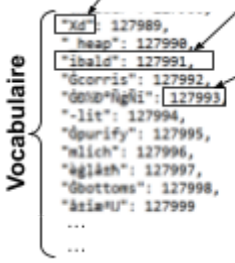
Au-delà des tokens de type texte standards, il existe **différents types de tokens** dits "**spéciaux**". Ces tokens spéciaux ont des rôles spécifiques, notamment dans le **contrôle** et le guidage des modèles de langage :

- Les **tokens de structuration** qui par exemple délimitent des sections. Exemple : <|im_start|>, <|im_end|> (pour GPT), [INST], [/INST] (pour Mistral).
- Les **tokens outils** qui par exemple listent les capacités disponibles compréhensibles dans la suite du traitement. Exemple : <|calculator_available|>, <|web_search_enabled|>

Comme chaque modèle utilise son propre tokenizer, la représentation des **tokens spéciaux** n'est **pas la même** d'un tokenizer à l'autre.

Pour résumer :

- Un **texte** est découpé en **segment textuel.**
- Un **segment textuel** est associé à un **identifiant**.
- L'ensemble des **tokens** constitue le **vocabulaire** du modèle.



Voici un extrait du **fichier tokenizer** au format **JSON** du modèle **Deepseek R1** contenant dans son dictionnaire 128 000 tokens.

On peut trouver ces fichiers de vocabulaire sur le site **Huggingface pour chaque modèle** en source ouverte. Cette plateforme propose des **outils**, des **modèles pré-entraînés** et **des jeux de données**, facilitant le **développement** et le **partage** d'applications d'IA.

huggingface.co/deepseek-ai/DeepSeek-R1/raw/8a58a132790c9935686eb97f042afa8013451c9f/tokenizer.json

Etape n°2 : Représentation vectorielle : le token embedding

Dans l'étape n°1, nous avons vu que le **tokenizer découpait le prompt en tokens**, et générait en sortie une liste d'**identifiants numériques**.
Dans l'étape n°2, via un mécanisme que nous allons voir juste après, chaque identifiant numérique va être mis en correspondance avec un **token embedding** (que nous allons expliquer).
Dans l'étape n°3, la liste des token embeddings obtenue à l'étape n°2 est fournie en entrée de la partie "transformer" du LLM.

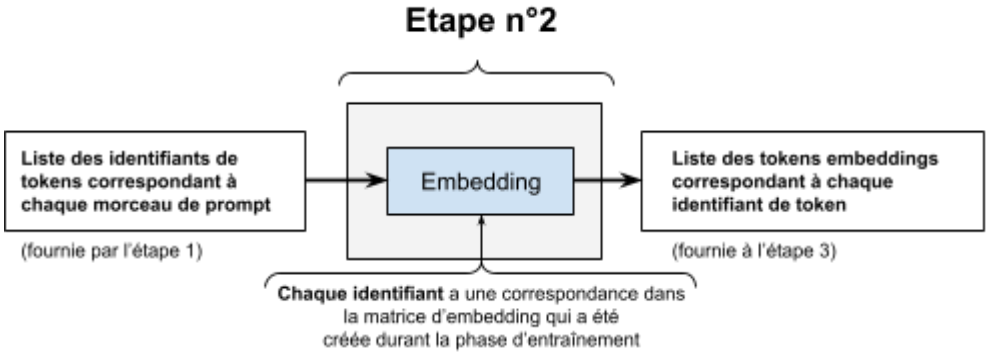


Figure 4 : **Phase d'exploitation de l'embedding**

Un **token embedding** est un **vecteur,** objet mathématique qui contient un ensemble de valeurs numériques plus facilement traitable et manipulable par un ordinateur. Retenons simplement qu'un **token embedding** permet de capturer, grâce aux valeurs numériques qu'il contient, **les nuances sémantiques et contextuelles** pour chaque token - on parle aussi de **représentation vectorielle dense**.

En phase d'exploitation, la fonction **Embedding** ci-dessus permet de faire correspondre, à chaque identifiant numérique, un **token embedding** associé. Cette association se fait grâce à une table de correspondance, appelée **matrice d'embeddings** (cf. figure 5), qui contient autant de **token embeddings** que de segments textuels connus dans le vocabulaire du LLM. Cette matrice est créée durant la phase d'entraînement.

Il y a **V** token embeddings dans cette table. Chaque token embedding possède **E** valeurs numériques.

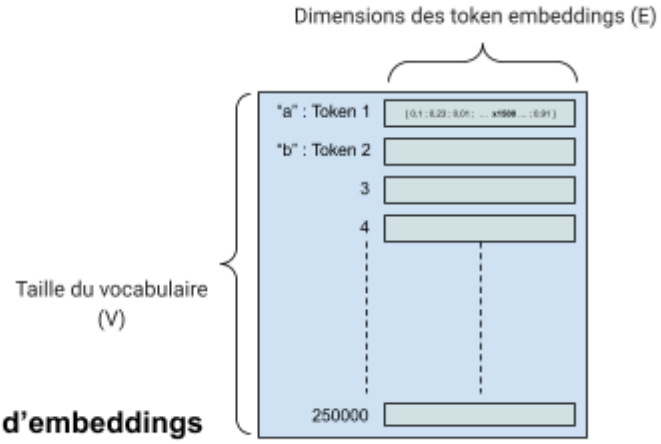


Figure 5 : **Matrice d'embeddings**

Note :

- **Token embedding** est aussi appelé **plongement sémantique.**
- **Matrice d'embeddings** est aussi appelée **matrice des plongements.**

Chaque valeur numérique correspond à une **dimension** du token embedding. Dit autrement, chaque **token embedding** possède **"E" dimensions** (cf. figure 5).

Pour simplifier et illustrer ces concepts, gardons en tête que les valeurs numériques, pour un token embedding donné, participent à représenter numériquement des notions sémantiques et conceptuelles telles que :

- Sens et contexte : certaines valeurs peuvent représenter le sens général du mot. Par exemple, pour le mot "*chien*", nous aurons un nombre élevé dans les dimensions participant à la notion "*animal*".
- Catégories grammaticales : certaines dimensions peuvent correspondre à des caractéristiques grammaticales, comme le fait d'être un nom ou un verbe.
- Niveau de langage : certaines dimensions pourraient refléter si le mot est formel, familier, ou argotique.
- Relations temporelles : d'autres dimensions pourraient capturer des notions de temps, comme le passé, le présent, ou le futur.

Les token embeddings peuvent contenir **des dizaines, des centaines, voire des milliers de dimensions**, offrant une représentation **très riche** d'un token donné.

> Les valeurs numériques de chaque token embedding sont ajustées **durant la phase d'entraînement** du module "embedding".
>
> Dans la phase d'apprentissage, la signification de chaque dimension du vecteur n'est **pas déterminée** à l'avance et n'est explicitement définie nulle part.
>
> Les exemples donnés permettent simplement d'illustrer un concept très abstrait. Une fois l'entraînement des embeddings terminé, **les valeurs numériques** dans chaque token embedding **ne changent plus** quelle que soit la quantité de texte fournie au LLM en phase d'exploitation.

Entre l'étape n°2 et n°3, une opération est réalisée pour apporter une notion de positionnement des tokens **entre eux**. Cette opération mathématique est faite entre le token embedding et un autre vecteur appelé "**Positional Encoding**". Nous n'allons pas expliquer dans ce document le détail de cette opération. Il s'agit avant tout de vous donner une idée des différentes étapes nécessaires entre le prompt et la génération de texte par le LLM.

La figure 6 ci-dessous illustre les différentes étapes entre le **texte** transmis **en entrée** et les **données numériques exploitables par le** transformeur.

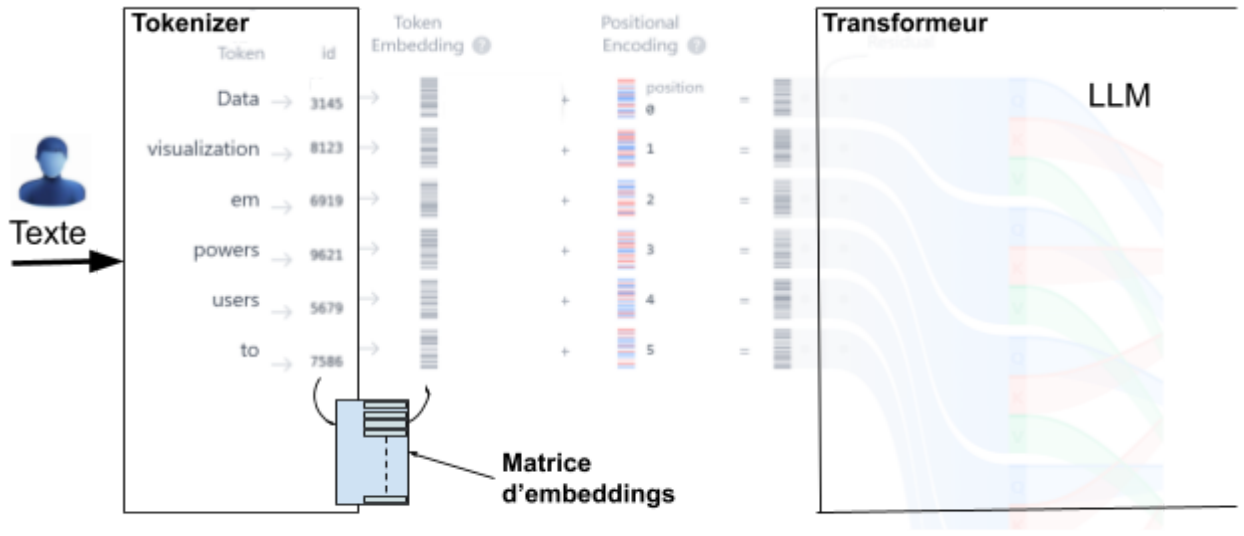


Figure 6 : **token embedding + positional encoding** transmis au **LLM**

A retenir :
Un **token embedding** est un vecteur fixe qui permet de traduire un segment textuel en valeurs numériques pouvant être comprises et traitées par l'IA et dont les valeurs ne bougent plus après l'entraînement du modèle.

Le terme plus générique d'**embedding** fait aussi référence à un vecteur, mais qui est utilisé comme objet mathématique tout au long du traitement numérique dans le transformeur.

A noter également, que lorsque l'on parle de "**nombre de tokens**", il peut y avoir 2 interprétations possibles suivant le contexte :

- Il peut s'agir de la fenêtre contextuelle du modèle.
  Dans ce cas, c'est le **nombre maximum de tokens** qu'il est possible de présenter simultanément **en entrée** du modèle, dans la phase d'exploitation.

- Il peut s'agir aussi de la taille du dictionnaire du tokenizer utilisé par le modèle.
  Dans ce cas, c'est le **nombre total de tokens** que le modèle a **appris à reconnaître** durant la phase d'entraînement et que le tokenizer utilise pour traiter le texte fourni en entrée durant la phase d'exploitation.

Etape 3 : Les transformeurs

Neurone artificiel

Afin de mieux comprendre la terminologie associée à un **réseau de neurones**, décrivons le fonctionnement d'**un seul neurone** qui, dans l'exemple présenté ci-après, prend trois données en entrée. Le neurone ne produit qu'une seule sortie :

- Trois données numériques en entrée représentent par exemple les "caractéristiques" d'un token.
- Chacune de ces entrées est associée à un poids (valeur numérique). Chaque poids détermine l'importance relative de la donnée d'entrée dans le calcul de la sortie.
- Le neurone calcule ensuite une somme de ses entrées pondérées avec les poids.
- Après quoi cette somme pondérée est passée à travers une fonction appelée **fonction d'activation** qui détermine la valeur numérique de **sortie du neurone**. Il existe plusieurs types de fonctions d'activation (sigmoïde, ReLU, tanh, etc.). Mais nous n'irons pas plus loin dans la vulgarisation d'un neurone.

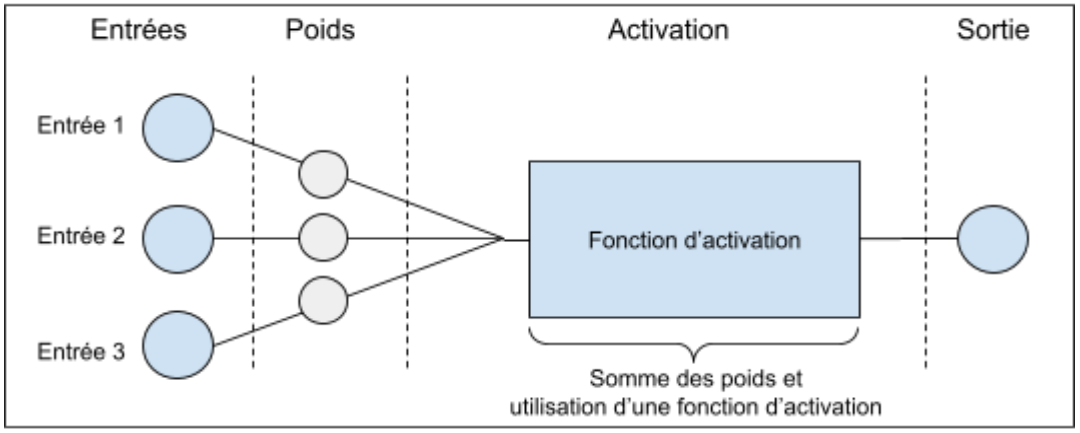


Figure 7 : **Un neurone simplifié**

Contrairement aux algorithmes traditionnels qui permettent de résoudre un problème spécifique en utilisant une séquence d'instructions prédéfinies, les **réseaux de neurones** sont capables "**d'apprendre**" à partir **de données**. L'apprentissage peut être vu comme un processus qui ajuste les **poids** des neurones pour **minimiser l'erreur** entre les **sorties prédites** et les **sorties réelles.** Durant la phase d'apprentissage, les neurones **ajustent leurs paramètres internes** en fonction des données qui leur sont présentées, permettant ainsi d'améliorer leur performance grâce à un volume de données d'apprentissage important.

La flexibilité et la **capacité de généralisation** sont également des atouts majeurs des réseaux de neurones. Une fois entraînés, ils peuvent être appliqués à de nouvelles données, inconnues lors de la phase d'apprentissage, pour faire des prédictions ou des classifications. Cette **capacité à généraliser** est importante pour leur utilisation dans des applications réelles où les conditions et les données peuvent varier.

Contrairement aux algorithmes traditionnels dont le fonctionnement est généralement transparent, explicable mathématiquement et facile à comprendre, les réseaux de neurones sont souvent considérés comme des "boîtes noires" en raison de la complexité des représentations linguistiques qu'ils apprennent.

Les transformeurs.

Avant l'avènement des transformeurs, les modèles de traitement de langage reposaient souvent sur des types de réseaux neuronaux qui présentaient plusieurs inconvénients. Ces modèles traitaient les données d'entrée de manière séquentielle, ce qui pouvait entraîner entre autres des limitations de capacité à prendre en compte les dépendances dans les séquences de texte fournies au modèle.

Proposés pour la première fois en 2017, les types de réseaux neuronaux appelés **transformeurs**, utilisés à l'origine pour **la traduction de texte,** ont permis de combler quelques lacunes. Ils étaient composés à cette époque de deux ensembles de traitements réalisés sur des entrées : **l'encodeur** et **le décodeur** (cf. figure 9).

- **L'encodeur** transforme la liste de tokens en une représentation condensée contenant l'information utile, permettant ainsi de "comprendre" le texte en profondeur.
- **Le décodeur** génère ensuite le texte en sortie, token par token, en utilisant les tokens précédemment générés et la représentation encodée du texte d'entrée.

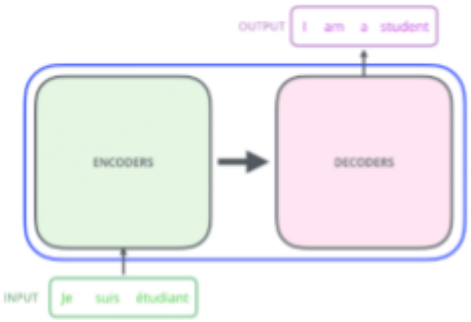


Figure 9 : **L'encodeur et le décodeur**

Source : https://meritis.fr/blog/les-transformers-le-modele-derriere-la-puissance-de-chatgpt

Cependant, **les LLM actuels ont évolué** et utilisent désormais **une approche plus directe sans utiliser le bloc encodeur**.
Nous allons maintenant détailler cette **étape n°3** et illustrer les fonctions utilisées par **le transformeur** dans la chaîne de traitement. L'idée n'est pas de décrire toutes les fonctions et toutes les couches d'un LLM dans le détail, mais plutôt de laisser entrevoir la quantité de traitements constituant le cœur d'un LLM.

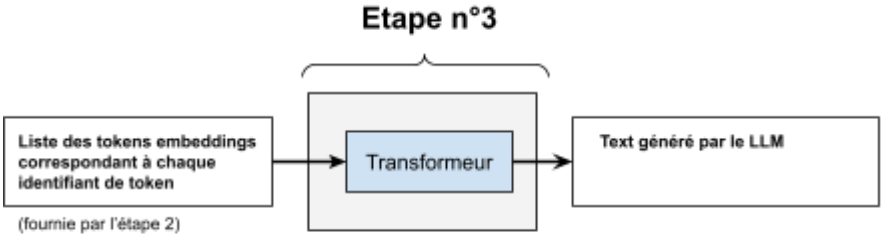


Figure 10 : **Transformeur dans la chaîne de traitement**

Afin de percevoir les opérations traitées par un **transformeur**, le schéma ci-dessous montre les principaux **blocs fonctionnels** d'un bloc transformeur de **modèle GPT** (Generative Pre-trained Transformer), en illustrant au passage la complexité des étapes de traitement de la donnée.

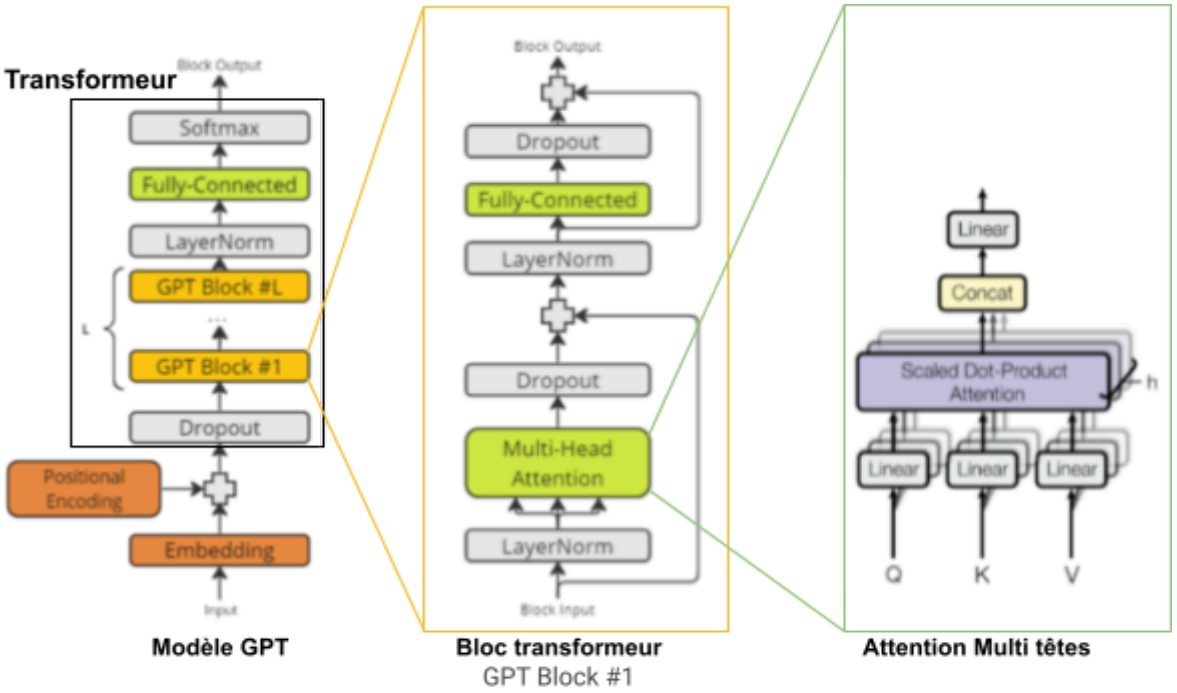


Figure 11 : **Vue éclatée d'un transformeur**

Source : https://www.yadavsaurabh.com/building-a-transformer-llm-with-code-fundamental-transformer-gpt/

Dans la vue éclatée du modèle GPT proposé en exemple, le transformeur représenté dans la partie gauche (cf. figure 11) est composé de **plusieurs blocs** de traitement de données, appelés "**GPT Block**" ou **bloc transformeur**. Les données en sortie d'un bloc servent d'entrée au bloc suivant. Chaque bloc transformeur est à son tour composé de plusieurs composants, présentés au centre de la figure 11 (entouré en orange).

Parmi les blocs importants du "**Bloc transformeur**", vous pouvez remarquer le composant "**Muti-head Attention**" (ou "Attention multi-têtes") et également le bloc "**Fully-Connected**" (ou "Réseau de neurones à propagation avant").

Ces deux blocs ont été choisis de par leur intérêt dans la chaîne de traitement. Leur fonctionnement interne s'avérant complexe, il est vulgarisé dans la section suivante. Il n'est pas nécessaire de détailler tous les autres blocs de cette boîte noire pour se faire une idée globale et éclairée de l'ensemble du système.

Pour résumer :
Les LLM reçoivent en entrée des tokens embeddings associés aux segments textuels provenant du prompt découpé en morceaux. Ces tokens embeddings sont traités par une série de transformeurs, chaînés les uns derrière les autres. Chaque transformeur utilise, entre autres, un bloc appelé "**attention multi-têtes**", et utilise séquentiellement, dans la chaîne de traitement, un bloc "**réseau de neurones à propagation avant**".

L'auto-attention multitêtes (**multi-head auto attention**), est une composante essentielle des transformeurs. C'est un ensemble de fonctions intégrées **à l'intérieur d'un réseau de neurones**. Elle a révolutionné la façon de traiter le langage naturel. Ce mécanisme complexe exploite des principes de traitement parallèle et une représentation contextuelle (grâce aux embeddings) permettant d'analyser les séquences d'entrée (comme les phrases ou les textes).

Dans le modèle GPT, chaque bloc transformeur contient un module **d'auto-attention multitêtes**, elle-même composée de **plusieurs têtes** exécutées **en parallèle**. La figure 12 illustre les opérations matricielles réalisées par une tête d'auto-attention.

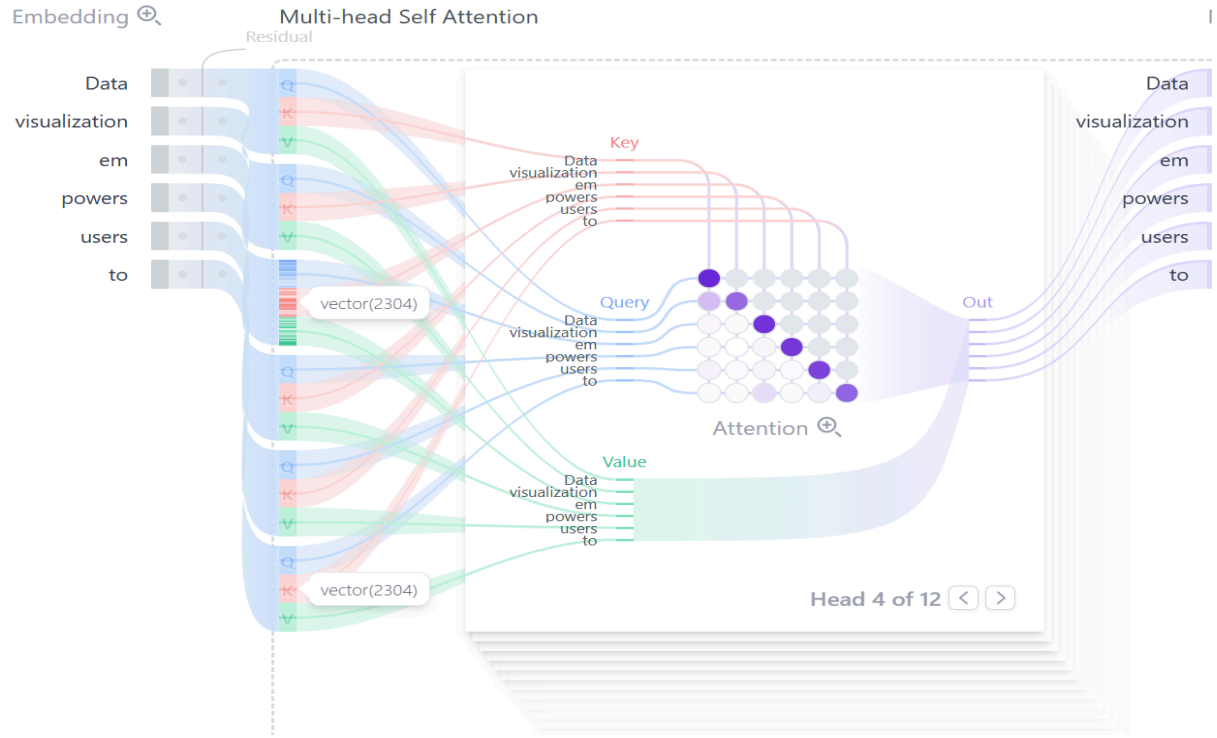


Figure 12 : **L'auto attention multitêtes**

Source : https://poloclub.github.io/transformer-explainer/ (**Consultez ce site très visuel**)

Voir également : https://www.3blue1brown.com/topics/neural-networks **(chapitres 5, 6 & 7)** Ces vidéos sont vraiment très visuelles que vous apprécierez après avoir lu ce document.

Avant d'illustrer un peu plus en détail les transformations matricielles successives utilisées dans la figure 13, voici une description très sommaire des 3 blocs représentés :

- **Dot product** : cela correspond à un produit scalaire entre deux vecteurs qui est utilisé pour calculer les scores d'attention de la matrice d'attention.
- **Scaling mask** : c'est un masque qui est utilisé pour mettre à l'échelle les poids de pondération.
- **Softmax** : c'est une fonction de normalisation qui convertit les scores d'attention en des poids de pondération qui somment à 1.

Techniquement, **l'auto-attention multitêtes** fonctionne en décomposant chaque représentation vectorielle d'entrée (embedding) en trois vecteurs distincts :
requête (Q), **clé (K)** et **valeur (V)**. Ces vecteurs sont ensuite utilisés pour calculer des **scores d'attention** entre les différentes **paires de tokens** de la séquence d'entrée du transformeur.
Le processus implique une série de transformations matricielles successives, incluant des produits scalaires entre les vecteurs Q et K, suivis d'une normalisation (softmax) pour obtenir **les poids d'attention**, qui sont ensuite utilisés pour pondérer les vecteurs V.

Cette opération est répétée sur plusieurs têtes d'attention en parallèle, chacune pouvant **capturer différents aspects ou relations dans les données**. Les résultats issus de ces différentes têtes sont ensuite combinés, généralement par concaténation suivie d'une transformation linéaire, pour former une représentation contextuelle riche et nuancée de la séquence d'entrée.

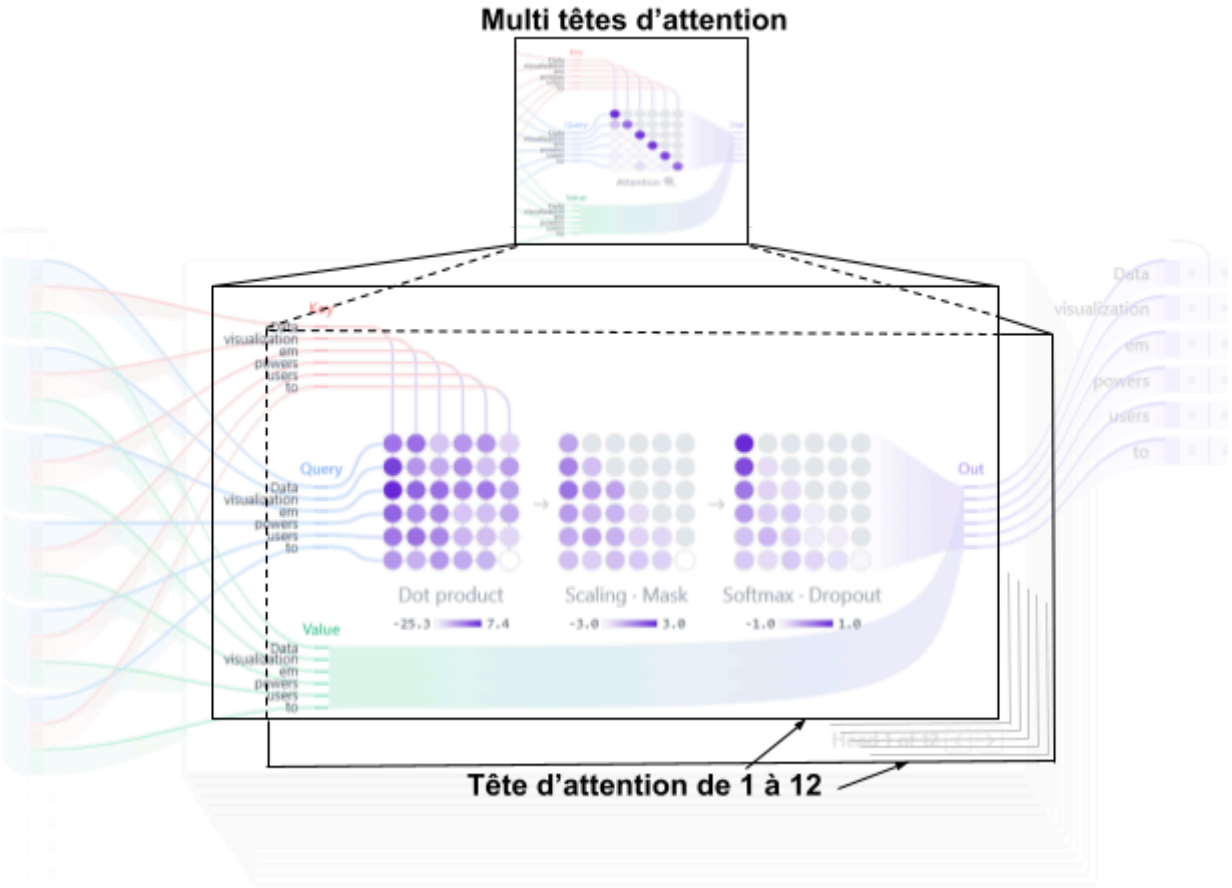


Figure 13 : **Multi têtes d'attention**

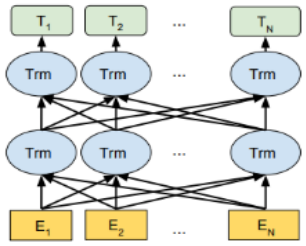
A noter que le nombre de têtes peut varier selon la configuration du modèle. Cette architecture contribue à la capacité du modèle à générer des textes de manière cohérente et contextuellement appropriée, en exploitant les relations complexes entre les éléments de la séquence d'entrée. A noter que ces 12 têtes sont spécialisées lors de la phase d'entraînement.

La possibilité d'effectuer la plupart des calculs en parallèle rend d'autant plus pertinente **l'utilisation de processeur graphique** (GPU) pour les calculs **réalisés en parallèle**.

Aparté sur l'**auto-attention** et **taille de la fenêtre contextuelle**.

Pour chaque token, le modèle doit évaluer son interaction **avec tous les autres tokens** présentés au LLM. Avec l'auto-attention, chaque token ajouté (dans la limite de la fenêtre contextuelle) augmente de manière importante les calculs nécessaires. Cela explique en partie pourquoi la **taille des fenêtres contextuelles** était très limitée il y a encore quelques années.
Cependant, l'accroissement de la puissance de calcul et l'amélioration des modèles actuels permettent aujourd'hui d'utiliser des fenêtres contextuelles allant jusqu'à **128 000 tokens** (fin 2024). A titre comparatif, en 2020, elles dépassaient difficilement les **2000 tokens**. Ce bond de performance explique en partie la récente popularité de ces modèles.

Réseau de neurones à propagation avant



Le deuxième composant important dans un transformeur est le **réseau de neurones à propagation avant,** en anglais **feedforward neural network (FFN)**. Après que les embeddings aient été traités par le bloc d'auto-attention multitêtes, ils sont ensuite passés à travers un réseau de neurones à propagation avant. Ce processus peut être vu comme une étape de transformation supplémentaire qui affine la représentation de chaque token dans la séquence. Le FFN est essentiellement composé de plusieurs couches de neurones :

1. **Couche d'entrée** : Les embeddings issus de l'auto-attention multitêtes sont introduits dans le FFN.
2. **Couches cachées** : Ces couches intermédiaires effectuent des transformations non linéaires sur les données. Typiquement, un FFN dans un transformeur utilise des couches cachées avec une fonction d'activation non linéaire, comme ReLU (Rectified Linear Unit) ou GELU (Gaussian Error Linear Unit).
3. **Couche de sortie** : La dernière couche produit la représentation finale des tokens après avoir été traitée par le FFN.

En appliquant des transformations non linéaires, le FFN peut modifier les représentations des tokens pour mieux capturer les nuances sémantiques et contextuelles.

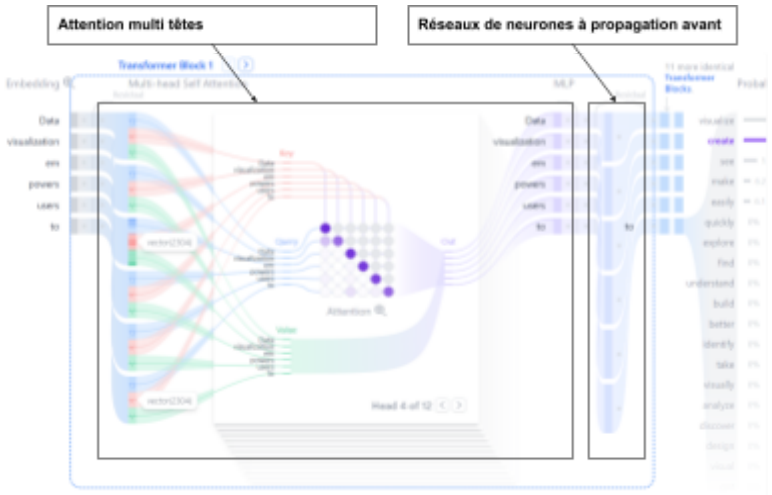


Figure 15 : **Attention multi têtes** suivies de **réseaux à propagation avant**

Mise en série des blocs

Les blocs transformeurs sont **cascadés en série**, formant une **chaîne de traitement profonde**. Chaque bloc affine progressivement la représentation, permettant au modèle de capturer des nuances linguistiques de plus en plus complexes.

Figure 16 : **Cascade en série de transformeurs**

A noter que le fait qu'il y ait 12 blocs transformeurs n'a pas de rapport avec le fait qu'il y ait 12 têtes d'attention.

Couche de sortie

Après avoir traversé les différentes étapes des blocs transformeurs, le modèle de langage produit un vecteur qui représente les probabilités **de chaque token** du vocabulaire reconnu. Cependant, cette matrice n'est pas encore du texte. Pour générer du texte cohérent et contextuellement approprié, le modèle utilise un processus appelé **génération auto-régressive**. C'est un mécanisme qui permet au modèle de générer du texte de manière séquentielle, en utilisant ses propres prédictions précédentes comme entrée pour générer les prédictions suivantes.

Le modèle prédit un token en fonction de l'entrée initiale. Il utilise ensuite les **tokens précédents** comme entrée pour prédire le **token suivant** et continue à générer des tokens de manière séquentielle, en utilisant les tokens précédents comme entrée.

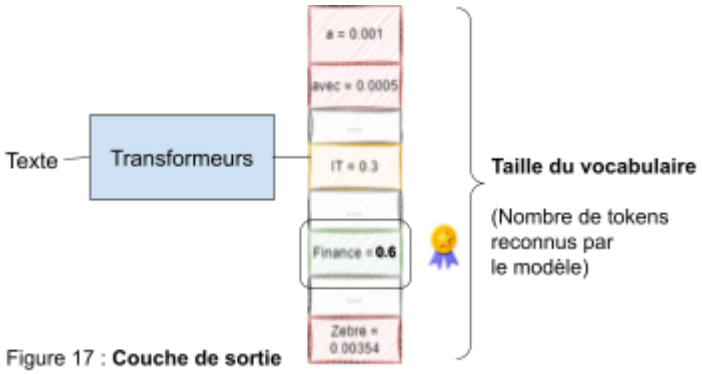


Figure 17 : **Couche de sortie**

Cependant, le modèle **n'a pas une mémoire infinie**, il ne peut donc pas utiliser tous les tokens précédents. La limite de token utilisée par le modèle est **la fenêtre contextuelle du modèle**. Les tokens qui ne sont pas dans cette fenêtre ne sont pas pris en compte.

Source : https://meritis.fr/blog/les-transformers-le-modele-derriere-la-puissance-de-chatgpt

La génération de texte s'arrête lorsque le **token spécial** "End of Sequence" (exemple <|endoftext|>) est généré par le LLM ou lorsque l'on **dépasse une limite de mots fixés** en hyperparamètre du modèle. A noter que ce token spécial est généré par le modèle lorsqu'il considère avoir terminé de répondre à la question ou d'accomplir la tâche demandée. Il est important de noter que ce **token spécifique** fait partie du vocabulaire du modèle et joue un rôle important dans le contrôle de la longueur et de la structure des réponses générées. Lorsque le modèle produit ce token, cela signale la **fin de la séquence de texte générée**.

Les limites des modèles LLM

Les LLM, c'est bien … Mais …
Comme pour tous les systèmes il y a des limites !

Pour ne citer que le modèle sur lequel nous nous sommes concentrés, les transformeurs sont des modèles qui ont permis de repousser les frontières de la compréhension du langage naturel par les ordinateurs, notamment grâce au mécanisme "d'attention".

Néanmoins, ces modèles présentent encore de nombreuses limitations, risques et impacts qui doivent être adressées (**hallucination**, **biais**, **consommation énergétique,** etc.).
Il serait intéressant de traiter l'ensemble de ces aspects dans un unique document, mais il serait au final indigeste. Nous verrons cela (peut-être) dans un prochain document.

Allez … juste un mot sur **les hallucinations :**

> Les hallucinations d'un LLM se réfèrent à la génération d'une réponse **factuellement incorrecte** ou **trompeuse**, présentée de **manière convaincante** comme une **information véridique**. Cela arrive lorsque le modèle produit des affirmations qui **semblent plausibles** mais qui ne reposent que sur des données résultant de sa méthode de prédiction **basée sur des probabilités** apprises à partir de vastes ensembles de données textuelles.
> Pour aller plus loin : https://datascientest.com/ai-hallucinations-tout-savoir

Des chiffres pour réaliser l'impact d'un LLM - juillet 2024

Source : https://www.youtube.com/watch?v=9vM4p9NN0Ts (Stanford lecture, 55'42")

Modèle : **LLaMA 3** (405 milliards de paramètres)
Entraîné sur : **15,6T tokens** (soit 15 600 000 000 000 000 tokens)
Nombre de tokens par paramètre durant l'entraînement : **~ 40** (explication dans l'annexe)
Nombre d'opérations pour s'entraîner : **3,8e25** (38 suivi de 24 zéros)
GPU : **16 000 cartes H100** (400 TFLOPS chaque) (Terra floating opérations per second)
Temps d'entraînement : **~70 days** - plus de **26 000 000 d'heures** GPU intensives
Pour équivalent, avec un coût de location d'une carte GPU : $2 / heure : **52 Million de $**

Défis et Enjeux de l'IA (Infrastructure, Puissance, Souveraineté) :
https://www.dailymotion.com/video/x96gebk

Annexes

Pour éviter de trop charger ce document de termes tout au long de l'explication, voici plusieurs définitions qui viennent compléter les notions présentées.

Poids

Les poids sont **les coefficients** qui sont appliqués **aux connexions entre les neurones** dans un réseau de neurones. Ils déterminent l'importance de chaque caractéristique d'entrée pour la prédiction finale. Les poids sont ajustés pendant l'entraînement du modèle en utilisant une méthode d'optimisation, telle que la descente de gradient, pour minimiser la fonction de coût.

Fonction d'activation

Une fonction d'activation dans un LLM est un composant mathématique essentiel qui introduit la non-linéarité dans le réseau neuronal. Elle détermine si un neurone artificiel doit être activé et l'intensité de cette activation, en transformant le signal d'entrée en une sortie utilisable par les neurones suivants.

La fonction d'activation :

- Transforme la somme pondérée des entrées d'un neurone en une sortie
- Permet au modèle de capturer des relations complexes et non linéaires entre les données.
- Imite le potentiel d'activation des neurones biologiques, décidant si l'information doit être transmise ou non.

Les fonctions d'activation courantes incluent la sigmoïde, la tangente hyperbolique (tanh) et la ReLU (Rectified Linear Unit). Chacune des ces fonctions a ses propres caractéristiques et applications spécifiques, permettant au réseau neuronal d'apprendre et de modéliser efficacement des patterns complexes dans les données.

https://cursa.app/fr/page/concepts-des-neurones-et-fonctions-d-activation

Biais

Les biais sont **des poids particuliers** qui sont ajoutés avant le passage d'information à la **fonction d'activation** dans un réseau de neurones. Ces valeurs sont déterminées pendant l'entraînement du modèle, en utilisant une méthode d'optimisation.

Paramètres et **hyperparamètres**

- Les paramètres sont les variables internes qui sont modifiées lors de la phase d'apprentissage du modèle (valeurs des poids, biais, etc.).
- Les hyperparamètres sont les variables configurables propre à l'algorithme utilisé (taux d'apprentissage, nombre de couches de neurones, etc.).

Tokens par paramètre (unité que l'on emploie parfois pour caractériser un modèle)

Règle empirique concernant le rapport entre le nombre de tokens dans les données d'entraînement et le nombre de paramètres dans un modèle de langage. Lorsqu'on entraîne un modèle de langage, il est important d'avoir suffisamment de données pour ajuster les paramètres de manière efficace. Un modèle entraîné avec trop de paramètres par rapport à la quantité de données d'entraînement risque de sur-apprendre (cf. surapprentissage) c'est-à-dire que le réseau de neurones va "mémoriser" les données d'entraînement plutôt que d'apprendre des représentations générales.

Exemple : Une proportion de 40 tokens par paramètre suggère que pour chaque paramètre du modèle, il faudrait environ 40 tokens dans les données d'entraînement. Cela permettrait au modèle d'avoir suffisamment d'exemples pour ajuster ses paramètres de manière robuste.
A noter que cette règle n'est pas universelle et peut varier en fonction de la complexité du modèle, de la tâche spécifique, de la qualité des données d'entraînement, etc. Certains modèles peuvent nécessiter plus ou moins de données par paramètre en fonction de leur architecture et de leur objectif.

Température
La température est un **hyperparamètre** qui contrôle le degré aléatoire de la sortie du modèle. Voici ses caractéristiques principales :

- Une température élevée (> 1.0) produit des résultats plus imprévisibles et créatifs.
- Une température basse (< 1.0) génère une sortie plus commune et conservatrice.
- À une température de 0, le modèle choisit toujours le token le plus probable, résultant en un texte plus prévisible.

L'ajustement de la température permet de trouver un équilibre entre la créativité et la cohérence du texte généré.

Top K
Le Top K est un autre **hyperparamètre** important dans la génération de texte par les LLM. Voici ses principales caractéristiques :

- Il limite le nombre de tokens les plus probables parmi lesquels le modèle peut choisir à chaque étape de la génération.
- Par exemple, si K = 10, le modèle ne considérera que les 10 tokens les plus probables pour chaque prédiction.
- Cela aide à contrôler la diversité et la qualité du texte généré en excluant les options moins probables.

Le Top K est souvent utilisé en conjonction avec d'autres techniques comme le Top p (nucleus sampling) pour affiner davantage le processus de génération.

Le taux d'apprentissage
Détermine la vitesse à laquelle le modèle ajuste les poids et les biais pendant l'entraînement. Un taux élevé peut causer des oscillations dans la convergence de l'apprentissage.À l'inverse, si le taux d’apprentissage est trop petit, les ajustements sont mineurs, et le modèle converge lentement vers la solution optimale. Cela peut rendre l'entraînement très long, nécessitant un grand nombre d'itérations.

Le surapprentissage
C’est un phénomène qui se produit lorsqu'un modèle de réseau neuronal apprend trop bien les détails et le bruit des données d'entraînement, au point de perdre sa capacité à généraliser à de nouvelles données ou des données inconnues.

La fonction de coût
Cette fonction est aussi appelée fonction d'erreur ou fonction de perte. Son rôle principal est de quantifier la différence entre les prédictions effectuées par le modèle et les valeurs réelles

observées dans les données d'entraînement. L'objectif fondamental de l'entraînement d'un modèle est de minimiser la valeur de cette fonction, indiquant ainsi que le modèle est capable de faire des prédictions aussi précises que possible.

Phase d'entraînement
Étape d'entraînement d'un modèle de langage où les paramètres internes (les poids, les biais, etc.) sont ajustés à partir de vastes données pour permettre au modèle d'apprendre et de comprendre le langage.

Phase d'inférence
C'est la phase durant l'utilisateur exploite le modèle entraîné. Le modèle génère des réponses textuelles en fonction des prompts utilisateurs, en utilisant les paramètres fixés lors de la phase d'entraînement.

Quelques autres exemples de réseaux de neurones profonds :
Dans le domaine de l'apprentissage profond, les architectures de réseaux de neurones sont diverses et répondent à des besoins spécifiques en fonction des types de données traitées et des tâches à accomplir. Il existe d'autres types de réseau de neurones que ceux que nous avons vus dans ce document pour vulgariser un LLM. Chacun ont leurs propres caractéristiques et domaines d'application.

- **Réseaux de neurones convolutifs** (Convolutional Neural Networks - CNN) : contiennent cinq types de couches : entrée, convolution, mise en commun, connexion complète et sortie. Chaque couche a une fonction spécifique, comme la synthèse, la connexion ou l'activation. Les réseaux neuronaux convolutifs ont popularisé la classification d'images et la détection d'objets. Cependant, les CNN ont également été appliqués à d'autres domaines, tels que le traitement du langage naturel et les prévisions.

- **Réseaux de neurones récurrents** (Recursive Neural Networks - RNN) : adaptés aux séquences de données. Contrairement aux réseaux neuronaux traditionnels, toutes les entrées d'un réseau neuronal récurrent ne sont pas indépendantes les unes des autres, et la sortie de chaque élément dépend des calculs des éléments qui le précèdent. Les RNN sont utilisés dans les applications de prévision et de séries temporelles, l'analyse des sentiments et d'autres applications textuelles.

- **Réseaux neuronaux auto-encodeurs** :
  Un auto encodeur est une catégorie de modèles génératifs qui crée des représentations pertinentes (embeddings) en compressant puis reconstruisant les données. (pour aller plus loin)

Neurone biologique

Pour comprendre l'analogie entre neurone artificiel et **neurone biologique** voici quelques notions intéressantes :

Neurone biologique détaillé

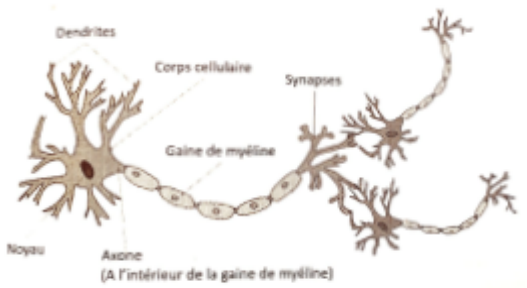
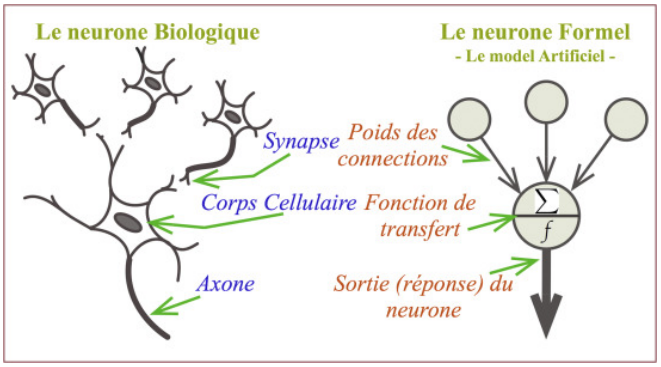


Figure 8 : **Représentation d'un neurone biologique**

Neurone simplifié en correspondance d'un neurone artificiel



Source à gauche : **Intelligence artificielle vulgarisée**, par Aurélien Vannieuwenhuyze, Edition ENI.
Source à droite : https://www.sciencedirect.com/science/article/abs/pii/S0242649819300203

Rappelons-nous quelques instants nos cours de biologie du lycée sur le fonctionnement de notre cerveau. Notre cerveau est composé de **86 à 100 milliards de neurones** dont le rôle est d'acheminer et de traiter des messages dans notre organisme. Certains neurones ont un rôle dédié aux perceptions des sensations et aux mouvements, alors que d'autres sont responsables des fonctions automatiques de notre corps (digestion, respiration...).

Biologiquement, un neurone est une cellule, composée :

- d'un corps cellulaire (également appelé "péricaryon") ;
- d'un noyau ;
- de plusieurs ramifications appelées dendrites ayant pour fonction d'être les **points d'entrée** de l'information dans le neurone ;
- d'un chemin de **sortie** de l'information appelé axone, pouvant atteindre une longueur d'un mètre ;
- d'une gaine de myéline protégeant l'axone ;
- des terminaisons axonales également appelées **synapses** connectées aux autres neurones.

La communication entre neurones s'opère par l'échange de messages sous forme de variation de tension électrique. Un neurone peut recevoir **plusieurs messages** de la part **d'autres neurones** auxquels il est connecté. En s'inspirant du fonctionnement du cerveau humain, un **réseau de neurones artificiels** tente de modéliser la structure et le fonctionnement des neurones et de leurs connexions. Selon Yann LeCun, un réseau de neurones artificiels *"C'est un abus de langage que de parler de neurones ! De la même façon qu'on parle d'aile pour un avion inspirée d'une aile d'un oiseau, le neurone artificiel est un modèle extrêmement simplifié de la réalité biologique."*