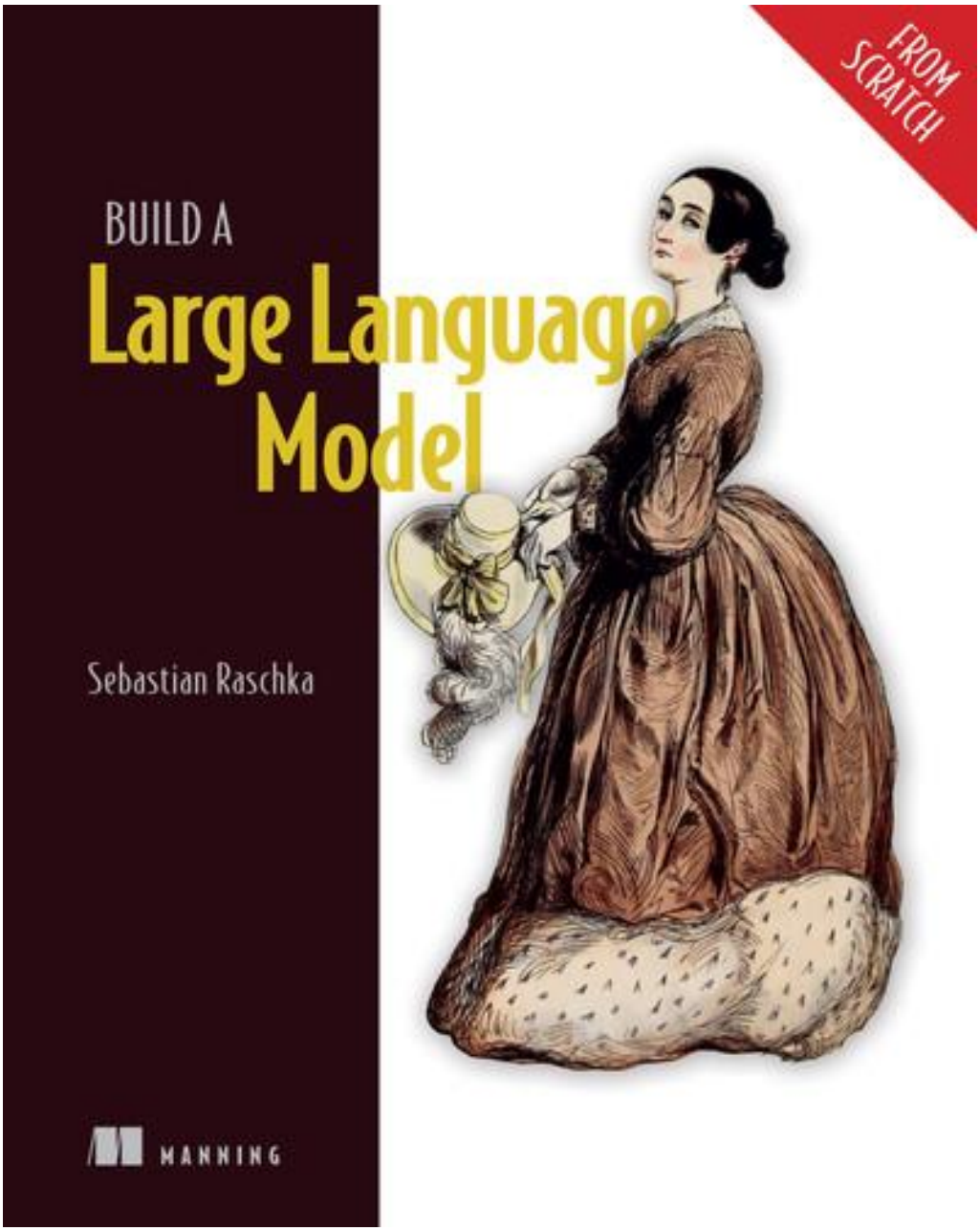
Lecture individuelle 1

Build an large language model (from scratch)



[1]

[1] https://learning.oreilly.com/covers/urn:orm:book:9781633437166/400w/

Table des matières

1. Introduction

Dans le contexte actuel où l'intelligence artificielle et le machine learning jouent un rôle prépondérant dans la transformation numérique, la compréhension des modèles de langage devient essentielle pour les professionnels de l'informatique. Le livre *"Build an LLM from Scratch"* de Sebastian Raschka s'inscrit dans cette dynamique en proposant une approche pratique et détaillée de la construction de modèles de langage à grande échelle.

Cet ouvrage, basé sur des principes fondamentaux du machine learning et des architectures de modèles modernes, vise à rendre accessible le processus complexe de création d'un modèle de langage de type GPT. À travers des explications claires et des exemples concrets, l'auteur guide le lecteur dans les différentes étapes, de la normalisation des données à l'implémentation des mécanismes d'attention, tout en abordant les défis liés à l'entraînement et à l'évaluation des modèles.

Ce livre s'inscrit parfaitement dans le cadre de l'acquisition des compétences pour ce 5ème semestre, notamment la compétence B4 « Connaître les impacts et les apports du Machine Learning et de l'intelligence artificielle sur le système d'information de l'entreprise », ainsi que la compétence M7 « Connaître les principaux concepts mathématiques… et savoir les appliquer dans un cas d'utilisation du Machine Learning ».

2. Compréhension des LLMS

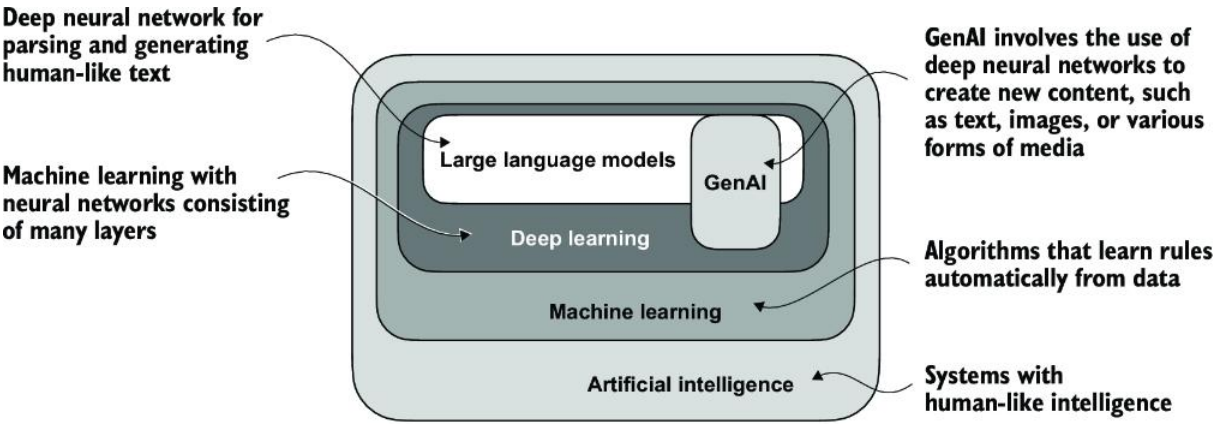
2.1. Définition d'un LLM

Un **LLM** (Large Language Model ou grand modèle de langage) est un réseau de neurones conçu pour comprendre, générer et répondre à des textes de manière similaire à un humain. Ces modèles sont des réseaux neuronaux profonds entraînés sur d'énormes quantités de données textuelles, couvrant parfois une grande partie des textes disponibles publiquement sur Internet.

Le terme "grand" dans "grand modèle de langage" fait référence à la fois à la taille du modèle en termes de paramètres et à l'immense ensemble de données sur lequel il est entraîné. Ces modèles comptent souvent des dizaines, voire des centaines de milliards de **paramètres**, qui sont les poids ajustables du réseau optimisés lors de l'entraînement pour prédire le mot suivant dans une séquence.

Les LLMs utilisent une architecture appelée le **transformers**, qui leur permet de prêter attention à différentes parties de l'entrée de manière sélective lorsqu'ils font des prédictions, les rendant particulièrement aptes à gérer les subtilités et les complexités du langage humain.

Étant donné que les LLMs sont capables de générer du texte, ils sont souvent considérés comme une forme d'**intelligence artificielle générative**, souvent abrégée en **GenAI** (Generative AI).



2.2. Application des LLM

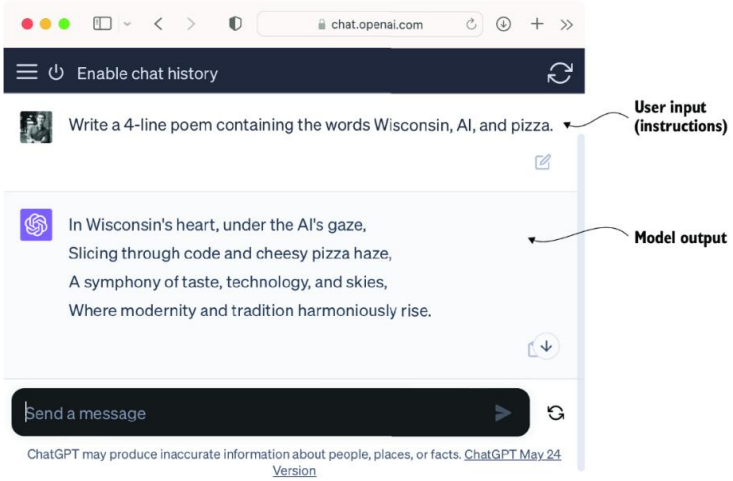
En raison de leurs capacités avancées à analyser et comprendre des données textuelles non structurées, les LLM (modèles de langage de grande taille) ont une large gamme d'applications

dans divers domaines. Aujourd'hui, ils sont utilisés pour la traduction automatique, la génération de textes nouveaux, l'analyse des sentiments, la synthèse de texte, et bien d'autres tâches. Récemment, les LLM ont été employés pour la création de contenu, tels que l'écriture de fictions, d'articles, et même de code informatique.

Les LLM alimentent également des chatbots sophistiqués et des assistants virtuels, comme ChatGPT d'OpenAI ou Gemini de Google (anciennement Bard), qui peuvent répondre aux questions des utilisateurs et compléter les moteurs de recherche traditionnels comme Google Search ou Microsoft Bing. De plus, ils peuvent être utilisés pour la récupération efficace de connaissances à partir de vastes volumes de textes dans des domaines spécialisés comme la médecine ou le droit. Cela inclut le tri de documents, la synthèse de passages longs et la réponse à des questions techniques.

En résumé, les LLM sont inestimables pour automatiser presque toute tâche impliquant l'analyse et la génération de texte. Leurs applications sont pratiquement infinies, et à mesure que nous continuons à innover et à explorer de nouvelles façons d'utiliser ces modèles, il est clair que les LLM ont le potentiel de redéfinir notre relation avec la technologie, la rendant plus conversationnelle, intuitive et accessible.

Dans cette lecture individuelle, l'accent sera mis sur la compréhension du fonctionnement des LLM depuis le début, avec la programmation d'un LLM capable de générer des textes. Vous apprendrez également les techniques permettant aux LLM d'effectuer des requêtes, allant de la réponse à des questions à la synthèse de textes, en passant par la traduction dans différentes langues, et bien plus encore. En d'autres termes, vous découvrirez comment des assistants LLM complexes tels que ChatGPT fonctionnent en construisant un modèle étape par étape.



2.3. Pourquoi construire son propre LLM ?

Pourquoi devrions-nous construire nos propres LLM ? Programmer un LLM depuis le début est un excellent exercice pour comprendre ses mécanismes et ses limitations. Cela nous permet également d'acquérir les connaissances nécessaires pour préformer ou affiner des architectures LLM open source existante en fonction de nos propres ensembles de données ou tâches spécifiques.

La plupart des LLM actuels sont implémentés à l'aide de la bibliothèque de deep learning PyTorch, que nous utiliserons également.

Des recherches ont montré que les LLM construits sur mesure, adaptés à des tâches ou domaines spécifiques, peuvent surpasser les LLM à usage général, comme ceux fournis par ChatGPT, conçus pour une large gamme d'applications. Des exemples incluent BloombergGPT, spécialisé en finance, et des LLM adaptés aux questions médicales.

L'utilisation de LLM sur mesure présente plusieurs avantages, notamment en matière de confidentialité des données. Par exemple, les entreprises peuvent préférer ne pas partager de données sensibles avec des fournisseurs tiers comme OpenAI en raison de préoccupations liées à la confidentialité. De plus, développer des LLM plus petits permet un déploiement direct sur des appareils clients, tels que des ordinateurs portables et des smartphones, ce que des entreprises comme Apple explore actuellement. Cette mise en œuvre locale peut réduire considérablement la latence et diminuer les coûts liés aux serveurs. En outre, les LLM personnalisés offrent aux développeurs une autonomie totale, leur permettant de contrôler les mises à jour et les modifications du modèle selon les besoins.

2.4. Processus de création d'un LLM

Le processus général de création d'un LLM comprend la préformation et le fine-tuning. La "pré" dans "préformation" fait référence à la phase initiale où un modèle, comme un LLM, est entraîné sur un large ensemble de données diverses pour développer une compréhension générale du langage. Ce modèle préformé sert ensuite de ressource fondamentale qui peut être affiné par le fine-tuning, un processus où le modèle est spécifiquement entraîné sur un ensemble de données plus restreint, adapté à des tâches ou domaines particuliers.

La **première** étape de la création d'un LLM consiste à l'entraîner sur un vaste corpus de données textuelles, souvent appelé texte brut. Ici, "brut" désigne le fait que ces données sont simplement du texte régulier sans aucune information d'étiquetage (bien que des filtrages puissent être appliqués, comme le retrait de caractères de formatage ou de documents dans des langues inconnues).

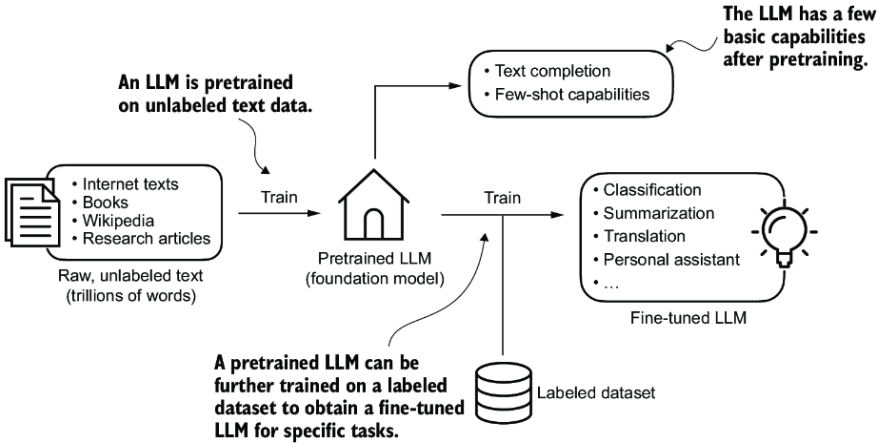
Il est à noter que, contrairement aux modèles d'apprentissage automatique traditionnels, qui nécessitent généralement des informations d'étiquetage, ce n'est pas le cas lors de la phase de préformation des LLM. À ce stade, les LLM utilisent l'apprentissage auto-supervisé, où le modèle génère ses propres étiquettes à partir des données d'entrée.

Cette première étape de formation d'un LLM est également connue sous le nom de préformation, créant un LLM préformé initial, souvent appelé modèle de base ou modèle fondamental. Un exemple typique de ce type de modèle est le modèle GPT-3 (précurseur du modèle original proposé dans ChatGPT), capable de compléter des textes, c'est-à-dire de terminer une phrase à moitié écrite fournie par un utilisateur. Il dispose également de capacités limitées en few-shot learning, ce qui signifie qu'il peut apprendre à effectuer de nouvelles tâches en se basant sur seulement quelques exemples, sans nécessiter une vaste quantité de données d'entraînement.

Après avoir obtenu un LLM préformé à partir de l'entraînement sur de grands ensembles de données textuelles, où le LLM est formé pour prédire le mot suivant dans le texte, nous pouvons affiner le LLM sur des données étiquetées, ce qu'on appelle le fine-tuning.

Les deux catégories de fine-tuning les plus populaires sont le fine-tuning par instruction et le fine-tuning par classification. Dans le fine-tuning par instruction, l'ensemble de données étiqueté se compose de paires d'instructions et de réponses, comme une requête pour

traduire un texte accompagné du texte correctement traduit. Dans le fine-tuning par classification, l'ensemble de données étiqueté comprend des textes associés à des étiquettes de classe, par exemple des courriels associés aux étiquettes "spam" et "non spam".

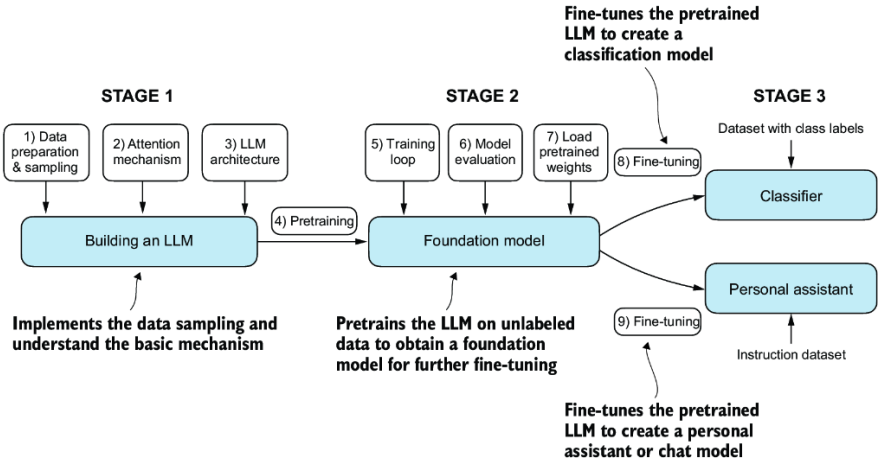


L'architecture Transformers

Les grands ensembles de données d'entraînement pour des modèles populaires comme GPT et BERT représentent des corpus textuels divers et complets englobant des milliards de mots, couvrant une vaste gamme de sujets et de langues naturelles et informatiques. Par exemple, le tableau ci-dessous résume l'ensemble de données utilisé pour la préformation de GPT-3, qui a servi de modèle de base pour la première version de ChatGPT. L'idée principale est que l'échelle et la diversité de cet ensemble de données d'entraînement permettent à ces modèles de bien performer sur diverses tâches, y compris la syntaxe, la sémantique et le contexte du langage, et même certaines nécessitant des connaissances générales.

Dataset name	Dataset description	Number of tokens	Proportion in training data
CommonCrawl (filtered)	Web crawl data	410 billion	60%
WebText2	Web crawl data	19 billion	22%
Books1	Internet-based book corpus	12 billion	8%
Books2	Internet-based book corpus	55 billion	8%
Wikipedia	High-quality text	3 billion	3%

Maintenant que nous avons posé les bases pour comprendre les LLM, passons à la programmation d'un LLM à partir de zéro. Nous utiliserons l'idée fondamentale derrière GPT comme modèle de référence et aborderons cela en trois étapes comme illustré dans l'image ci-dessous.



Étape 1 : Prétraitement des données

Dans la première étape, nous apprendrons les étapes fondamentales de prétraitement des données et coderons le mécanisme d'attention, qui est au cœur de chaque LLM.

Dans cette lecture individuelle, seule l'étape 1 va être traitée, car elle est assez conséquente.

Cette étape permet déjà la génération de texte.

3. Traitement des données textuels

3.1. Comprendre l'embedding des mots

Les modèles de réseaux de neurones profonds, y compris les LLM, ne peuvent pas traiter directement le texte brut. Étant donné que le texte est catégorique, il n'est pas compatible avec les opérations mathématiques utilisées pour implémenter et entraîner des réseaux de neurones. Par conséquent, nous devons trouver un moyen de représenter les mots sous forme de vecteurs à valeurs continues.

Le concept de conversion de données en un format vectoriel est souvent désigné sous le terme d'embedding. En utilisant une couche spécifique de réseau de neurones ou un autre modèle de réseau de neurones pré-entraîné, nous pouvons intégrer différents types de données — par exemple, vidéo, audio et texte. Cependant, il est important de noter que différents formats de données nécessitent des modèles d'embedding distincts. Par exemple, un modèle d'embedding conçu pour le texte ne serait pas adapté pour intégrer des données audios ou vidéo. Au cœur de ce concept, un embedding est une représentation des objets discrets, tels que des mots, des images ou même des documents entiers, en points d'un espace vectoriel continu. L'objectif principal des embeddings est de convertir des données non numériques en un format que les réseaux de neurones peuvent traiter.

Bien que les embeddings de mots soient la forme la plus courante d'intégration de texte, il existe également des embeddings pour des phrases, des paragraphes ou des documents entiers. Les embeddings de phrases ou de paragraphes sont des choix populaires pour la génération augmentée par récupération. La génération augmentée par récupération combine la génération (comme la production de texte) avec la récupération (comme la recherche dans une base de connaissances externe) pour extraire des informations pertinentes lors de la génération de texte. Étant donné que notre objectif est de former des LLM de type GPT, qui apprennent à générer du texte un mot à la fois, nous nous concentrerons sur les embeddings de mots.

Plusieurs algorithmes et cadres ont été développés pour générer des embeddings de mots. L'un des premiers et des plus populaires est l'approche Word2Vec. Word2Vec entraîne une architecture de réseau de neurones pour générer des embeddings de mots en prédisant le contexte d'un mot donné le mot cible, ou vice versa. L'idée principale derrière Word2Vec est

que les mots apparaissant dans des contextes similaires ont tendance à avoir des significations similaires. Par conséquent, lorsqu'ils sont projetés dans des embeddings de mots en deux dimensions à des fins de visualisation, des termes similaires sont regroupés. Les embeddings de mots peuvent avoir des dimensions variées, allant d'une à des milliers. Une dimensionnalité plus élevée peut capturer des relations plus nuancées, mais au prix de l'efficacité computationnelle.

Bien que nous puissions utiliser des modèles pré-entraînés tels que Word2Vec pour générer des embeddings pour des modèles d'apprentissage automatique, les LLM produisent généralement leurs propres embeddings qui font partie de la couche d'entrée et sont mis à jour pendant l'entraînement. L'avantage d'optimiser les embeddings dans le cadre de l'entraînement du LLM, plutôt que d'utiliser Word2Vec, est que les embeddings sont optimisés pour la tâche et les données spécifiques en question. Nous implémenterons de telles couches d'embedding plus loin dans ce chapitre.

3.2. Concept de la tokenization

La tokenization est le processus de division d'un texte en unités plus petites, appelées "tokens". Ces tokens peuvent être des mots, des caractères ou des sous-unités, et ils sont essentiels pour que les modèles de langage, comme les LLM, puissent traiter et comprendre le texte.

Byte Pair Encoding (BPE)

Le **Byte Pair Encoding (BPE)** est une méthode de tokenization plus sophistiquée, utilisée pour former des LLM comme GPT-2, GPT-3 et le modèle original de ChatGPT. Contrairement à des méthodes de tokenization simples, BPE permet de gérer des mots inconnus de manière efficace.

Fonctionnement du BPE :

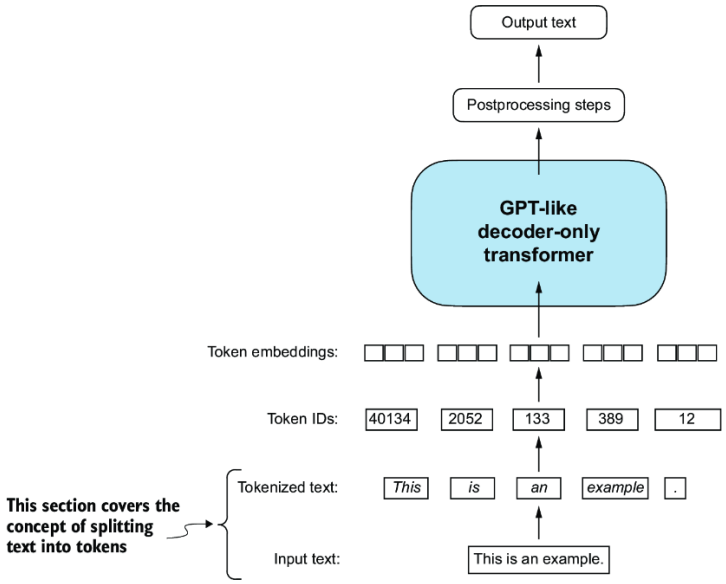
- **Division des mots** : Lorsque le tokenizer BPE rencontre un mot qui n'est pas dans son vocabulaire prédéfini, il le décompose en unités plus petites appelées sous-mots ou caractères individuels. Cela permet de représenter des mots inconnus sans avoir recours à des tokens spéciaux comme <|unk|>.

- **Vocabulaire** : Le tokenizer BPE utilisé pour former les modèles comme GPT-2 et GPT-3 possède une taille de vocabulaire de 50 257 mots, où chaque mot ou sous-mot est associé à un identifiant unique.

- **Utilisation de la bibliothèque tiktoken** : Pour faciliter l'implémentation du BPE, la bibliothèque Python open-source tiktoken est utilisée. Elle est simple à installer avec la commande pip install tiktoken. Une fois installée, le tokenizer BPE peut être instancié avec le code tokenizer = tiktoken.get_encoding("gpt2").

Avantages du BPE :

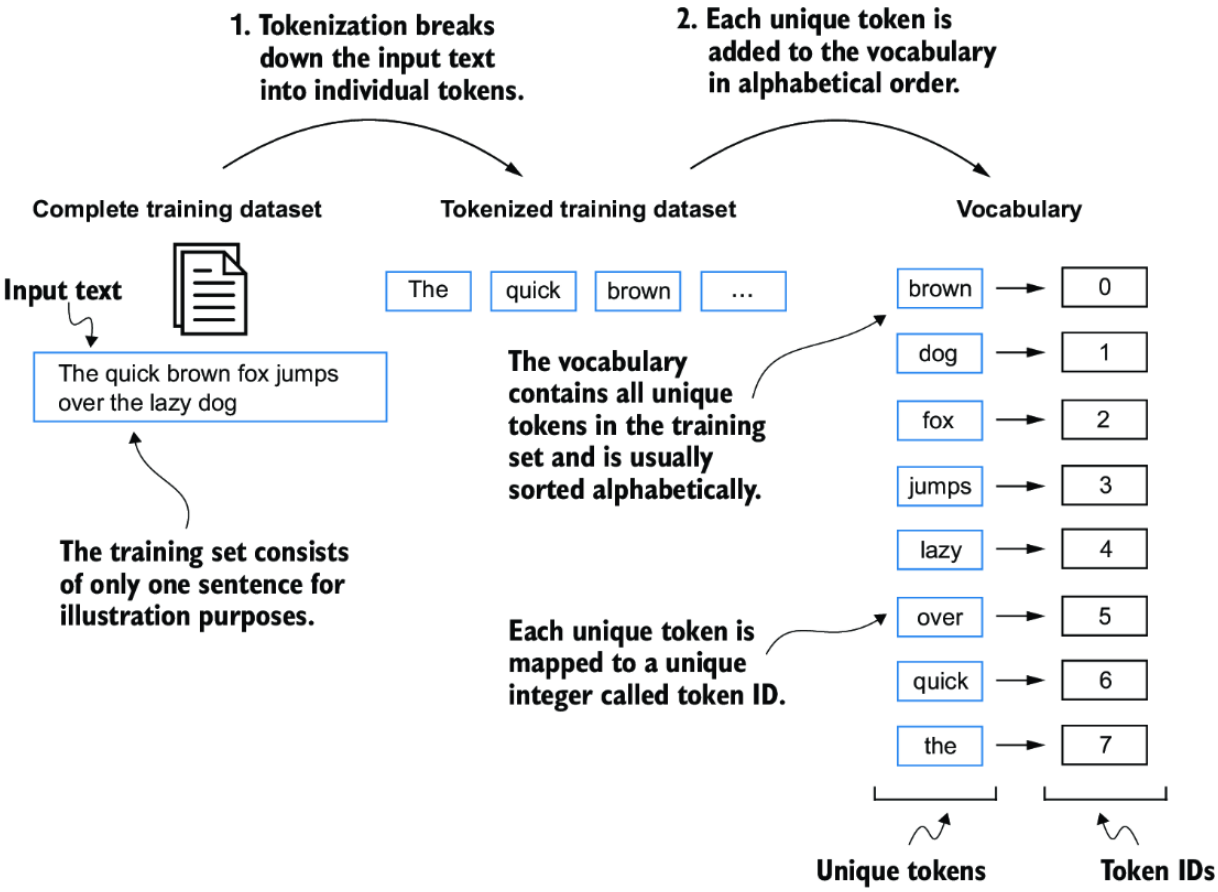
- **Flexibilité** : Grâce à sa capacité à traiter des mots inconnus en les décomposant, le BPE est particulièrement efficace pour les modèles de langage.
- **Performance** : En ayant un vocabulaire bien défini et en permettant une représentation des mots inconnus, le BPE contribue à améliorer la compréhension et la génération du langage par les LLM.



3.3. Conversion des tokens en IDs de tokens

Une fois qu'on a les tokens, il faut les convertir d'une chaîne Python en une représentation entière pour produire les IDs de tokens. Cette conversion est une étape intermédiaire avant de convertir les IDs de tokens en vecteurs d'embedding.

Pour mapper les tokens générés précédemment en IDs de tokens, nous devons construire un vocabulaire. Ce vocabulaire définit comment nous associons chaque mot unique et chaque caractère spécial à un entier unique.



3.4. Création d'embeddings de tokens

La dernière étape de la préparation du texte d'entrée pour l'entraînement des LLM consiste à convertir les IDs de tokens en vecteurs d'embeddings. Cette conversion se fait en plusieurs étapes.

Tout d'abord, les poids d'embedding doivent être initialisés avec des valeurs aléatoires. Cela sert de point de départ pour l'apprentissage du modèle. Ces poids seront ensuite ajustés pendant l'entraînement du LLM, à mesure que le modèle apprend à représenter les tokens d'une manière plus utile pour la tâche à accomplir.

Une fois les poids initialisés, la conversion des IDs en vecteurs d'embeddings peut commencer. Il est important de noter que pour que cette conversion soit efficace, il faut que chaque token soit représenté par un vecteur dans un espace continu. Cela est nécessaire, car les LLM comme GPT sont des réseaux de neurones profonds, et ces réseaux sont entraînés en utilisant un

algorithme appelé rétropropagation, qui ajuste les poids du modèle pour améliorer ses prédictions.

Pour illustrer le processus, imaginons que nous ayons un vocabulaire très simple de 6 mots, chacun étant associé à un ID unique. La dimension des vecteurs d'embedding est également définie. Par exemple, nous choisissons une dimension de 3 pour les vecteurs d'embedding. Dans ce cas, chaque token du vocabulaire sera représenté par un vecteur de taille 3. Ces vecteurs seront stockés dans une matrice où chaque ligne représente un token et chaque colonne une dimension de l'embedding.

Ensuite, pour obtenir le vecteur d'embedding correspondant à un token, on procède à une recherche dans cette matrice en utilisant l'ID du token. Par exemple, si l'ID du token est 3, on récupère la ligne correspondante dans la matrice pour obtenir le vecteur d'embedding de ce token. Cette opération est répétée pour chaque token d'entrée.

À la fin de cette étape, chaque token d'entrée est représenté par un vecteur d'embedding. Cela permet de transformer le texte en une forme que le modèle peut traiter et apprendre à partir de lui.

3.5. Encodage des positions des mots

Les embeddings de tokens constituent une bonne base d'entrée pour un LLM, mais un léger inconvénient se présente : le mécanisme d'auto-attention, qui est au cœur de ces modèles, ne prend pas en compte la position ou l'ordre des tokens dans une séquence. En effet, le processus d'embedding fonctionne de manière que le même ID de token soit systématiquement mappé à la même représentation vectorielle, peu importe où cet ID apparaît dans la séquence. Cela peut être vu comme un avantage en termes de reproductibilité, mais cela pose un problème lorsqu'il s'agit de saisir la structure séquentielle du texte.

Pour remédier à cette limitation, il est nécessaire d'ajouter des informations supplémentaires concernant la position des tokens dans la séquence. Cela permet au modèle de prendre en compte l'ordre des mots pour mieux comprendre le contexte. Il existe deux grandes catégories d'embeddings positionnels : les embeddings positionnels absolus et relatifs.

Les **embeddings positionnels absolus** sont directement associés à des positions spécifiques dans la séquence d'entrée. Chaque token recevra un embedding unique en fonction de sa position exacte dans la séquence. Par exemple, le premier token aura un embedding positionnel particulier, le deuxième un autre, et ainsi de suite. Cela permet au modèle de savoir où chaque token se situe dans la séquence.

D'autre part, les **embeddings positionnels relatifs** se concentrent non pas sur la position exacte d'un token, mais sur la **distance** relative entre les tokens. Le modèle apprend ainsi les relations entre les tokens en fonction de leur proximité plutôt qu'en fonction de leur position absolue dans la séquence. Cette approche présente un avantage majeur : elle permet au modèle de mieux généraliser à des séquences de différentes longueurs, même si certaines longueurs n'ont pas été rencontrées lors de l'entraînement.

Ces deux types d'embeddings positionnels visent à améliorer la compréhension de l'ordre et des relations entre les tokens dans une séquence. Ils aident à garantir des prédictions plus précises et un meilleur traitement du contexte. Le choix entre ces deux approches dépend souvent des spécificités de l'application et des données traitées.

Les modèles GPT d'OpenAI, par exemple, utilisent des **embeddings positionnels absolus**, mais avec une différence importante : ces embeddings sont **optimisés pendant l'entraînement**, contrairement à ceux du modèle Transformer original, qui sont fixés à l'avance. Ce processus d'optimisation fait partie intégrante de l'entraînement du modèle, ce qui permet d'ajuster les embeddings positionnels en fonction des besoins spécifiques du modèle pour des performances optimales.

4. Les mécanismes d'attention

4.1. Le problème de la modélisation de longues séquences

Avant l'arrivée des architectures avec mécanismes d'attention, comme les transformers, les réseaux neuronaux récurrents (RNNs) étaient utilisés dans les modèles de traduction, avec une structure d'encodeur-décodeur. L'encodeur lit et traite le texte source, et le décodeur génère la traduction.

Cependant, les RNNs présentent une limitation majeure : lorsqu'un texte complexe est traité, l'encodeur résume l'information dans un état caché final, mais ce dernier ne permet pas de conserver un accès aux informations plus anciennes pendant la phase de décodage. Ce manque d'accès au contexte peut entraîner une perte d'information, surtout pour des phrases longues ou des dépendances à longue distance entre les mots.

Ce problème a motivé la création de mécanismes d'attention, qui permettent de mieux capturer et utiliser les dépendances contextuelles dans les longues séquences, améliorant ainsi la traduction et d'autres tâches séquentielles.

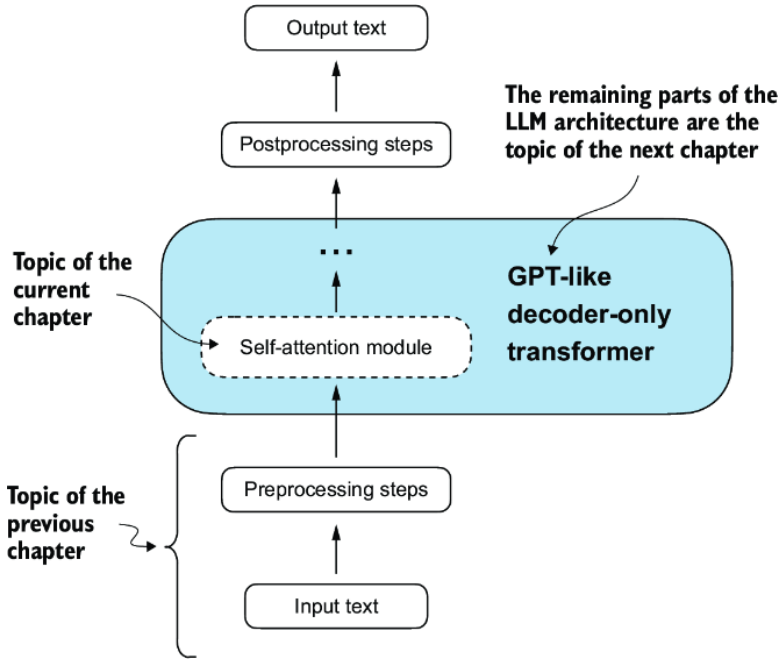
4.2. Capturer les dépendances des données avec les mécanismes d'attention

Les RNNs, bien qu'efficaces pour les courtes phrases, sont limités pour les textes longs car ils ne peuvent pas accéder directement aux mots précédents dans l'entrée. Leur principal défaut est qu'ils doivent condenser toute l'entrée dans un état caché unique avant de la transmettre au décodeur.

Pour pallier ce problème, le mécanisme d'attention Bahdanau a été développé en 2014. Ce mécanisme modifie l'architecture RNN en permettant au décodeur d'accéder de manière sélective aux différentes parties de la séquence d'entrée à chaque étape de décodage. Cela permet au modèle de prêter attention à des mots spécifiques en fonction de leur pertinence pour la génération de chaque mot de la sortie.

Trois ans plus tard, la recherche a montré que les RNNs n'étaient pas nécessaires pour le traitement du langage naturel, conduisant à la création du transformeur, une architecture utilisant le mécanisme d'attention auto-référentielle (self-attention). Ce mécanisme permet à chaque position de la séquence d'entrée de considérer la pertinence de toutes les autres

positions lorsqu'elle calcule la représentation de la séquence. Ce mécanisme d'attention est essentiel dans les LLMs modernes comme ceux de la série GPT.



4.3. Le mécanisme de self-attention sans poids

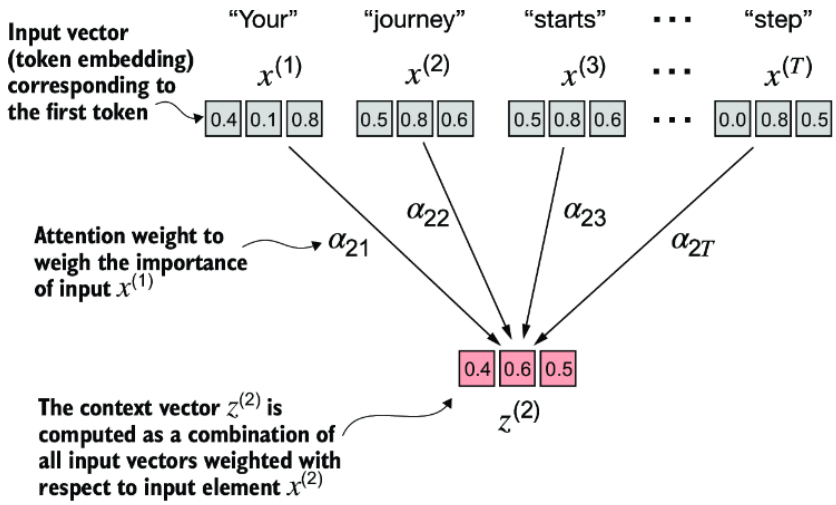
Le concept de self-attention est crucial pour permettre à un modèle de comprendre les relations entre différents éléments d'une même séquence d'entrée, comme les mots d'une phrase ou les pixels d'une image.

Nous commencerons par voir une version simplifiée du mécanisme d'attention, sans utiliser de poids entraînables, afin de mieux comprendre les principes de base. Ce mécanisme permet de calculer des vecteurs de contexte enrichis pour chaque élément de la séquence d'entrée, basés sur les relations et dépendances internes à la séquence.

Exemple de calcul des scores d'attention :

L'élément x(2) dans la séquence, correspondant à "journey", sera utilisé comme "query". Nous calculons les scores d'attention entre ce "query" et les autres éléments de la séquence (x(1) à x(T)).

Les scores d'attention sont obtenus en calculant le produit scalaire entre x(2) et chaque élément de la séquence.

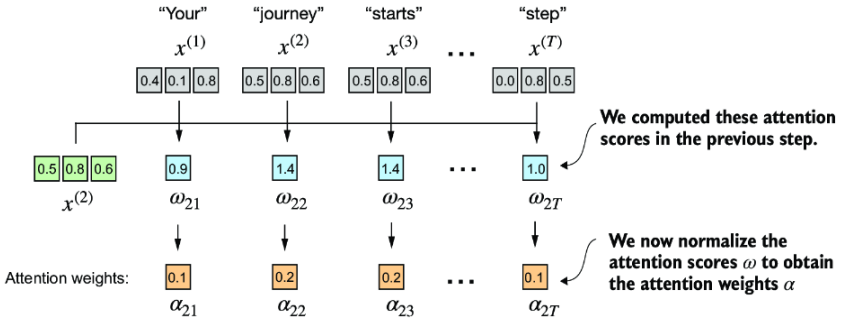


Les scores d'attention calculés mesurent la similarité entre l'élément de requête et chaque autre élément. Ce calcul est essentiel pour déterminer quelles parties de la séquence doivent être considérées lors de la création du vecteur de contexte.

4.3.1. Normalisation des scores d'attention

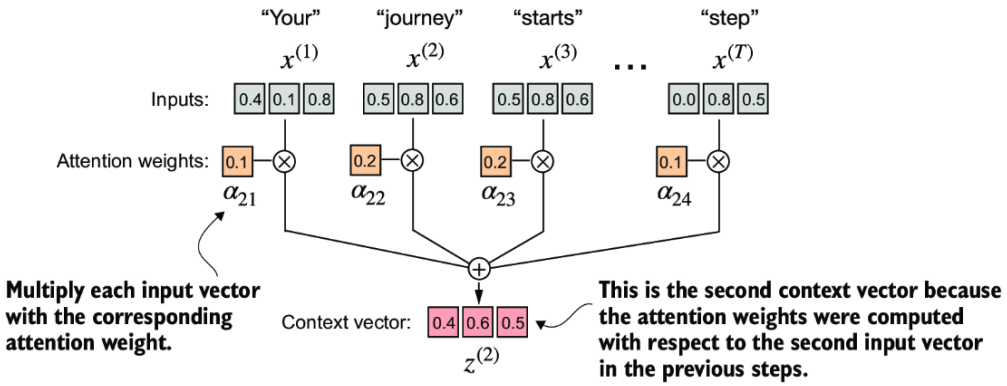
Une fois les scores d'attention calculés, ils sont normalisés pour obtenir des poids d'attention qui sommeillent à 1. Cela permet d'interpréter les poids comme des probabilités, où des poids plus élevés indiquent une plus grande importance.

La normalisation est effectuée en utilisant la fonction softmax, qui transforme les scores en poids interprétables :



4.3.2. Calcul du vecteur de contexte

Le vecteur de contexte est obtenu en effectuant une somme pondérée des vecteurs d'entrée, chaque vecteur étant multiplié par son poids d'attention correspondant. Ce vecteur enrichi contient des informations provenant de l'ensemble de la séquence, permettant au modèle de mieux comprendre les relations entre les éléments.



4.4. Le mécanisme de self-attention avec poids

Le mécanisme de self-attention avec poids fait évoluer le concept de self-attention pour permettre à l'algorithme de s'apprendre des relations entre les différents éléments d'une séquence en utilisant des poids ajustables au cours de l'entraînement. Ce mécanisme, utilisé dans des architectures comme les Transformers et les modèles GPT, introduit des matrices de poids entraînables qui modifient les vecteurs de requête, clé et valeur, permettant au modèle de s'adapter et de mieux représenter les dépendances dans les données.

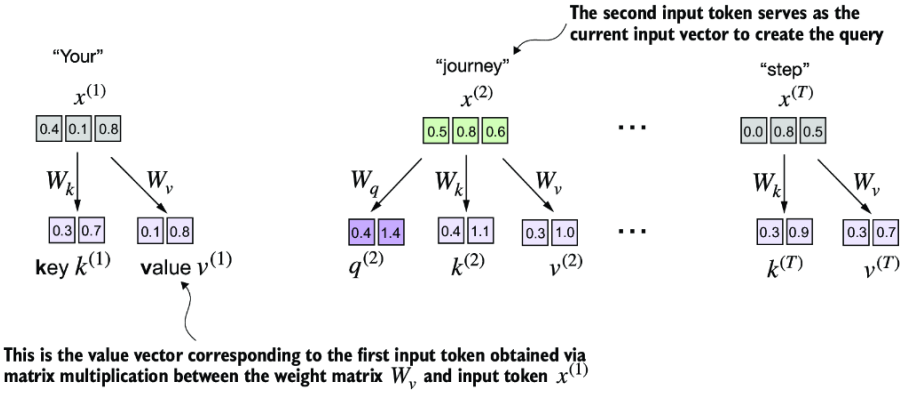
4.4.1 Matrices de poids : Requêtes, Clés et Valeurs

Le cœur du mécanisme de self-attention avec poids repose sur l'introduction de trois matrices de poids, Wq, Wk, et Wv, qui sont utilisées pour transformer les représentations d'entrée en vecteurs de requête, de clé, et de valeur, respectivement. Ces matrices sont apprises au cours de l'entraînement du modèle, ce qui permet au mécanisme d'affiner les relations entre les éléments de la séquence.

Étapes de transformation :

1. **Requête (q)** : Chaque élément d'entrée x(i)x(i)x(i) est multiplié par la matrice WqW_qWq pour obtenir le vecteur de requête q(i)q(i)q(i).
2. **Clé (k)** : Chaque élément d'entrée x(i)x(i)x(i) est multiplié par la matrice WkW_kWk pour obtenir le vecteur de clé k(i)k(i)k(i).
3. **Valeur (v)** : Chaque élément d'entrée x(i)x(i)x(i) est multiplié par la matrice WvW_vWv pour obtenir le vecteur de valeur v(i)v(i)v(i).

Les matrices sont les éléments clés que le modèle ajustera pendant l'entraînement pour apprendre à mieux contextualiser chaque élément par rapport aux autres dans la séquence.

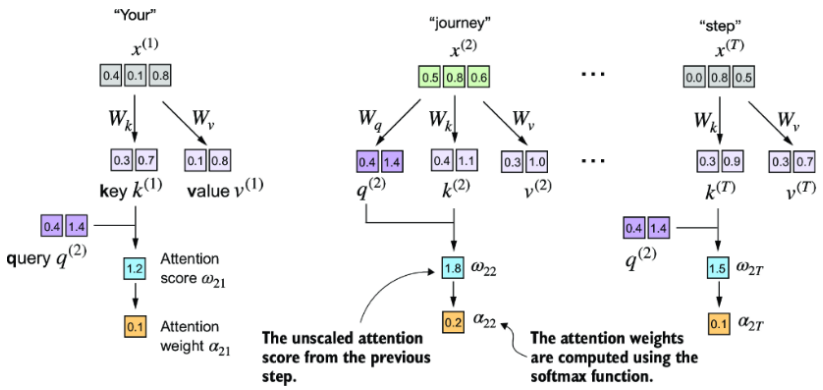


Ensuite, le processus est un peu près le même pour le calcul des scores d'attention et la normalisation.

Pourquoi les termes "requête", "clé" et "valeur" ?

Les termes requête, clé et valeur proviennent de la recherche d'information et de la gestion des bases de données, où ces concepts sont utilisés pour rechercher, indexer et récupérer des informations.

- Requête (query) : Représente l'élément actuel (par exemple, un mot ou un token) sur lequel le modèle se concentre. C'est l'élément à partir duquel le modèle recherche des relations dans la séquence.
- Clé (key) : Sert à indexer les éléments de la séquence. Chaque élément d'entrée possède une clé, et ces clés sont utilisées pour comparer avec la requête.
- Valeur (value) : Représente l'information associée à chaque élément d'entrée, qui est récupérée si l'attention portée à cet élément est élevée.



4.5. Causal attention

Dans de nombreuses tâches pour les LLMs, il est essentiel que le modèle ne tienne compte que des tokens précédant ou au même endroit que le token actuel lors de la prédiction. **Causal attention** modifie le mécanisme standard de self-attention pour **masquer les tokens futurs**, empêchant ainsi toute fuite d'information.

Principe

On masque les poids d'attention situés **au-dessus de la diagonale** de la matrice d'attention. Après le masquage, les poids restants sont **normalisés** pour que chaque ligne totalise 1. Cela garantit que seuls les tokens précédents (ou actuels) influencent le calcul du contexte d'un token donné.

Étapes d'implémentation

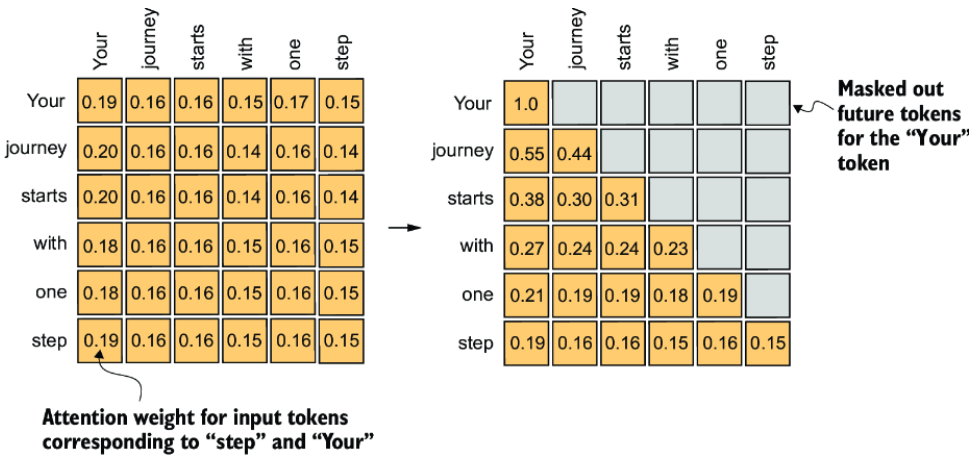
1. Calcul initial des poids d'attention :

   Les scores d'attention sont obtenus via les produits scalaires entre les requêtes et les clés, suivis d'une normalisation avec la fonction softmax.
2. Application d'un masque :

   On utilise torch.tril pour créer une matrice triangulaire inférieure, qui masque les valeurs au-dessus de la diagonale.

   On applique ensuite ce masque pour **annuler les contributions des tokens futurs**.
3. Renormalisation :

   Après masquage, les poids d'attention sont recalculés pour que chaque ligne reste une distribution de probabilités valide.



4.6. Architecture d'attention multi-têtes

L'un des principaux progrès des mécanismes d'attention est l'introduction de l'attention multi-têtes, une technique clé des modèles de type transformers. Contrairement à l'attention simple (single-head attention), où une seule projection linéaire est utilisée pour les vecteurs de requête, de clé et de valeur, l'attention multi-têtes divise le processus en plusieurs sous-espaces. Chaque tête traite les informations à partir d'une perspective différente, ce qui permet au modèle de capturer une diversité plus large de relations contextuelles au sein des données.

4.6.1. Principe de l'attention multi-têtes

Dans l'attention multi-têtes, les vecteurs d'entrée (requête, clé et valeur) sont projetés en plusieurs sous-espaces via des matrices de poids indépendantes pour chaque tête. Ces sous-espaces permettent à chaque tête de se concentrer sur des aspects spécifiques de la séquence, comme des relations à courte ou à longue distance entre les mots.

Après calcul des scores d'attention et des vecteurs de contexte pour chaque tête, les résultats sont concaténés et transformés à l'aide d'une dernière projection linéaire. Ce processus enrichit les représentations finales en combinant différentes perspectives, améliorant ainsi la capacité du modèle à comprendre des relations complexes dans les données.

4.6.2. Calcul parallèle dans les têtes d'attention

L'implémentation efficace de l'attention multi-têtes repose sur le calcul simultané des opérations pour toutes les têtes. Plutôt que de traiter chaque tête indépendamment, les vecteurs de requête, de clé et de valeur sont regroupés en un seul tenseur, ce qui permet d'effectuer les multiplications matricielles pour toutes les têtes en parallèle.

Ce traitement parallèle réduit considérablement le temps de calcul tout en exploitant pleinement les ressources de calcul, telles que les unités de traitement graphique (GPU). Par exemple, en utilisant une matrice de poids partagée mais adaptée aux têtes, les calculs des projections peuvent être réalisés simultanément, améliorant ainsi la rapidité et l'efficacité.

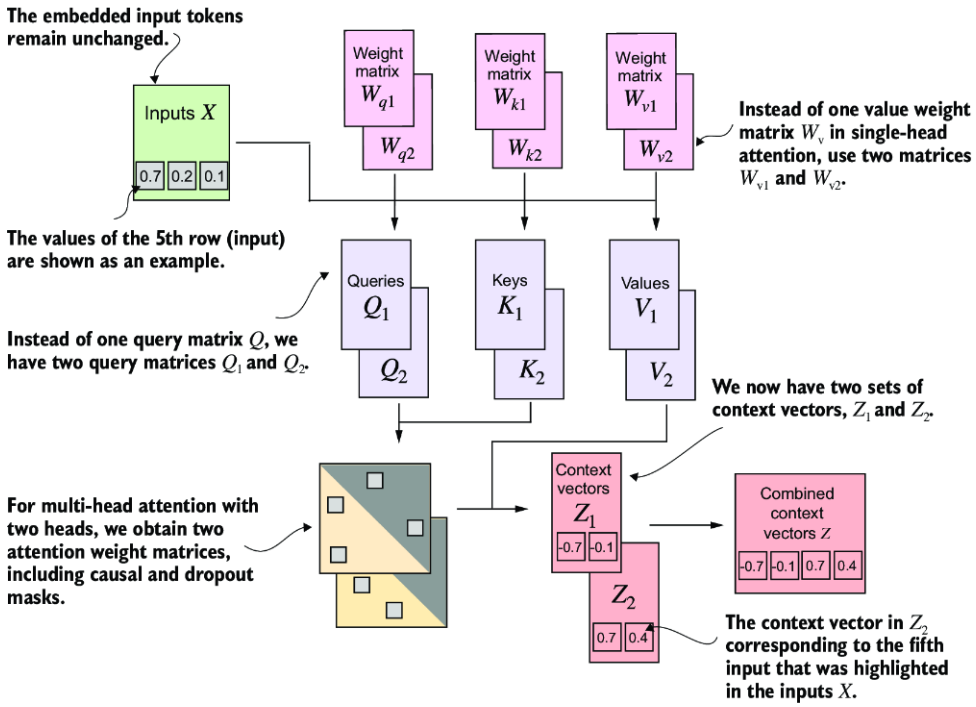
4.6.3. Avantages et limitations de l'attention multi-têtes

L'attention multi-têtes présente plusieurs avantages :

- **Diversité contextuelle** : Chaque tête offre une perspective unique sur les relations entre les éléments d'une séquence.

- **Richesse des représentations** : La concaténation des résultats des têtes permet une représentation plus riche et plus complète de la séquence.

- **Flexibilité d'adaptation** : En ajustant le nombre de têtes et les dimensions des sous-espaces, le mécanisme peut être optimisé pour différentes tâches et tailles de données.



5. L'implémentation d'un modèle GPT

Les modèles de langage de grande taille (LLM), tels que GPT (Generative Pretrained Transformer), sont des réseaux neuronaux profonds conçus pour générer du texte un mot (ou un token) à la fois. Bien que leur taille puisse être impressionnante, leur architecture est relativement simple, car elle repose sur la répétition de certains composants clés.

Un modèle GPT typique est constitué de plusieurs éléments principaux :

- **Embeddings des tokens et des positions :**
  Les tokens (mots ou parties de mots) sont d'abord transformés en vecteurs numériques appelés "embeddings". En plus des embeddings des tokens, un modèle GPT utilise des embeddings de position pour indiquer l'ordre des tokens dans la séquence. Cela permet au modèle de comprendre la relation entre les différents mots dans une phrase.
- **Blocs transformers :**
  Le cœur du modèle GPT repose sur des blocs appelés **transformers**. Chaque bloc contient une unité d'attention multi-têtes masquée. L'attention permet au modèle de se concentrer sur différentes parties de la séquence d'entrée de manière parallèle et hiérarchique. L'attention "masquée" signifie que, pendant l'entraînement, chaque token ne peut voir que les tokens précédents et non ceux qui viennent après.
- **Norme et sortie :**
  Après avoir traversé plusieurs blocs transformer, la sortie est normalisée et envoyée à une couche linéaire (une sorte de "tête" de sortie) qui génère les prédictions du modèle sous forme de logits. Ces logits seront convertis en tokens (mots) lors de la phase de génération de texte.

Dans ce chapitre, nous allons voir tous les éléments nécessaires à un modèle GPT.

5.1. Paramètres du modèle GPT

Les **paramètres** d'un modèle GPT désignent les poids et variables internes ajustés pendant l'entraînement. Ces paramètres permettent au modèle d'apprendre et de s'améliorer pour générer du texte plus cohérent et réaliste. Un modèle comme GPT-2, par exemple, contient environ 124 millions de paramètres dans sa version la plus petite.

- Le **vocabulaire** du modèle GPT se compose de tokens qui sont associés à des indices. Par exemple, GPT-2 utilise un vocabulaire de 50 257 tokens, qui représente des mots ou des morceaux de mots.
- **Longueur de contexte :** Cela indique combien de tokens le modèle peut prendre en compte simultanément. GPT-2, par exemple, peut gérer un contexte de 1024 tokens.
- **Dimensions des embeddings :** Chaque token est converti en un vecteur de taille fixe, ici 768 dimensions dans l'exemple de GPT-2. Ce vecteur représente chaque mot dans un espace continu, ce qui permet au modèle de capter des relations sémantiques entre les mots.

5.2. Normalisation des activations avec la normalisation de couche

L'entraînement des réseaux neuronaux profonds comportant de nombreuses couches peut être difficile en raison de problèmes tels que la disparition ou l'explosion des gradients. Ces problèmes rendent l'entraînement instable, ce qui complique l'ajustement efficace des poids du réseau. Par conséquent, le réseau rencontre des difficultés à apprendre les motifs sous-jacents dans les données, ce qui entrave sa capacité à effectuer des prédictions précises.

La **normalisation de couche** est une méthode permettant de surmonter ces problèmes. Son objectif est d'ajuster les activations (sorties) d'une couche du réseau de manière qu'elles aient une moyenne de 0 et une variance de 1, également connue sous le nom de **variance unitaire**. Cette opération aide à accélérer la convergence vers des poids efficaces et assure un entraînement stable.

Dans les architectures modernes de transformateurs, telles que GPT-2, la normalisation de couche est appliquée avant et après le module d'attention multi-têtes.

Procédure de la normalisation de couche

1. Création des entrées et des couches :
   On commence par créer un ensemble de données d'entrée et une couche neuronale. La couche se compose d'une **couche linéaire** suivie d'une activation non linéaire, ici ReLU, une fonction standard dans les réseaux neuronaux. Cette activation permet de garantir que les valeurs de sortie de la couche sont uniquement positives.

2. Calcul de la moyenne et de la variance :

   Une fois les sorties obtenues, nous calculons la **moyenne** et la **variance** des activations pour chaque entrée. L'objectif est de vérifier si les activations sont bien équilibrées avant d'appliquer la normalisation.

3. Application de la normalisation :

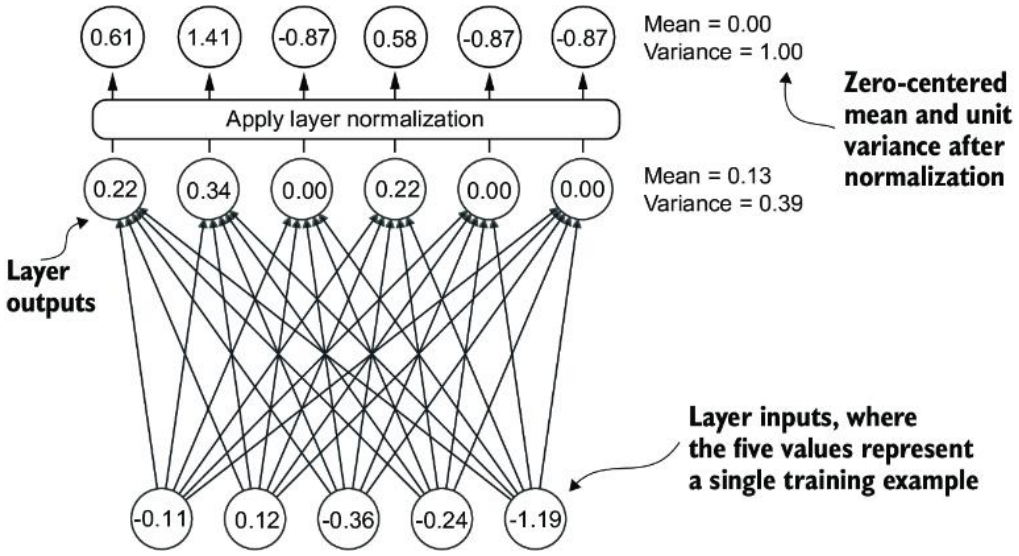
   La normalisation se fait en soustrayant la moyenne de chaque activation et en divisant par l'écart type (la racine carrée de la variance). Cette étape garantit que les activations de chaque entrée auront une moyenne de 0 et une variance de 1.

4. Vérification des résultats :

   Une fois la normalisation effectuée, il est important de vérifier les résultats en recalculant la moyenne et la variance des sorties normalisées. La normalisation devrait aboutir à une moyenne très proche de 0 et une variance de 1.

5. Encapsulation dans un module :

   Une fois que la procédure de normalisation a été définie, nous l'encapsulons dans un **module PyTorch** réutilisable, qui pourra être intégré dans le modèle GPT lors de l'entraînement.



5.3. Implémentation d'un réseau de neurones feed-forward avec des activations GELU

La fonction d'activation ReLU est couramment utilisée en apprentissage profond en raison de sa simplicité et de son efficacité dans diverses architectures de réseaux de neurones. Cependant, dans les LLMs, des fonctions d'activation plus complexes, comme GELU (Gaussian Error Linear Unit) et SwiGLU (Swish-Gated Linear Unit), sont utilisées, car elles offrent de meilleures performances dans des modèles profonds.

La fonction GELU est définie par :

$$GELU(x) \approx 0.5 \cdot x \cdot \left(1 + tanh\left[\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right]\right)$$

Pour des raisons de calcul, une approximation de cette fonction est souvent utilisée, comme cela a été le cas dans GPT-2.

GELU vs ReLU

La fonction ReLU est une fonction linéaire par morceaux qui produit directement la valeur d'entrée si elle est positive, et zéro sinon. En revanche, la GELU est une fonction non linéaire lisse qui est plus fluide et permet une meilleure optimisation des paramètres du modèle pendant l'entraînement. Contrairement à ReLU, qui est égale à zéro pour les valeurs négatives, GELU permet une petite sortie non nulle même pour les entrées négatives. Cela permet aux neurones recevant des entrées négatives de contribuer au processus d'apprentissage, bien que dans une moindre mesure que pour les valeurs positives.

Impact sur l'optimisation et l'apprentissage

La douceur de la GELU peut améliorer les propriétés d'optimisation, en permettant des ajustements plus nuancés des paramètres du modèle pendant l'entraînement. En revanche, ReLU, avec son angle aigu à zéro, peut parfois compliquer l'optimisation, surtout dans les réseaux très profonds ou ayant des architectures complexes.

Application dans un réseau de neurones feed-forward

Le réseau de neurones feed-forward, qui sera utilisé dans le bloc du transformateur des LLMs, se compose de deux couches linéaires et d'une activation GELU. Bien que les dimensions

d'entrée et de sortie de ce module soient les mêmes, le modèle agrandit d'abord la dimension de l'embedding via la première couche linéaire, suit avec une activation GELU non linéaire, puis réduit la dimension de retour à sa taille d'origine avec une deuxième transformation linéaire. Cette architecture permet d'explorer un espace de représentation plus riche tout en maintenant la dimension d'entrée et de sortie inchangée.

5.4. Ajout de connexions par raccourci

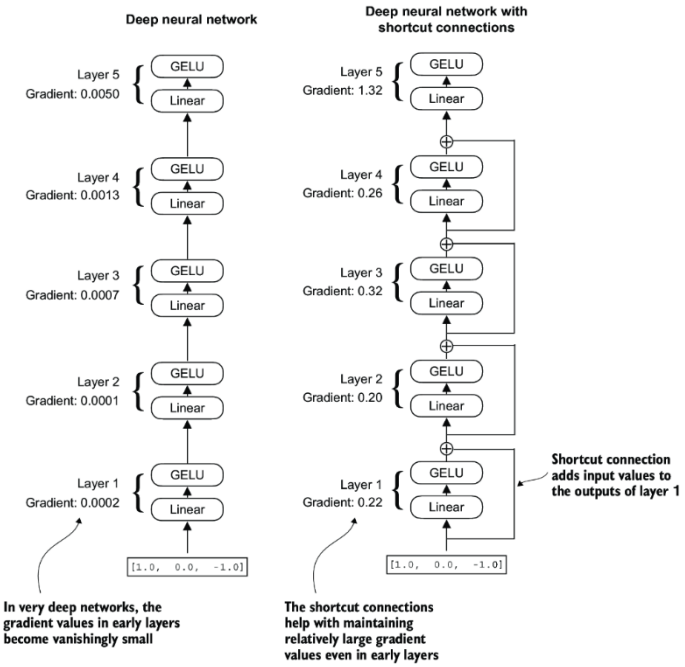
Les connexions par raccourci, aussi appelées connexions résiduelles, sont essentielles pour améliorer l'entraînement des réseaux de neurones profonds, en particulier pour contrer le problème du gradient qui disparaît. Ce phénomène se produit lorsque les gradients, utilisés pour mettre à jour les poids durant l'entraînement, deviennent de plus en plus petits à mesure qu'ils se propagent à travers les couches du réseau. Cela rend difficile l'apprentissage des premières couches du modèle, limitant ainsi l'efficacité de l'entraînement.

Concept des connexions par raccourci

Les connexions par raccourci consistent à ajouter l'entrée d'une couche à sa sortie, créant ainsi un chemin alternatif qui contourne certaines couches. Ces connexions jouent un rôle crucial dans la préservation du flux des gradients durant la rétropropagation. En ajoutant l'entrée d'une couche à sa sortie, ces connexions permettent aux gradients de passer plus facilement à travers les couches profondes, facilitant ainsi l'entraînement des réseaux très profonds.

Application dans un réseau de neurones profond

Voici un exemple de l'ajout de connexions par raccourci à un réseau de neurones constitué de cinq couches. Chaque couche est une transformation linéaire suivie d'une activation GELU. L'ajout des connexions par raccourci est conditionné par l'option use_shortcut. Si cette option est activée, la sortie de chaque couche est additionnée à son entrée, à condition que les dimensions de l'entrée et de la sortie correspondent. Cela crée un chemin plus court pour les gradients lors de la rétropropagation.



5.5. Connexion de l'attention multi-têtes et les couches linéaires

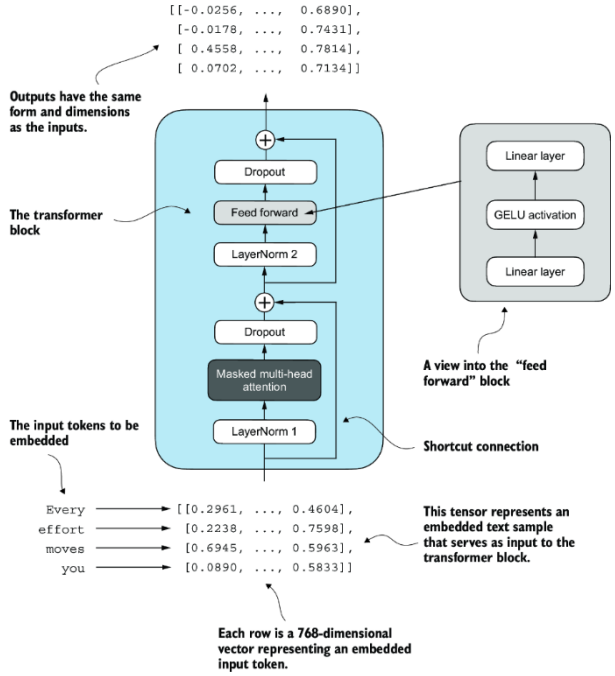
La prochaine étape est de connecter l'attention multi-tête et les couches linéaires dans un bloc de transformer. Ce bloc est un élément fondamental dans les architectures de GPT et autres modèles de type LLM. Il combine plusieurs concepts que nous avons déjà vus, tels que l'attention multi-tête, la normalisation de couche, le dropout, les réseaux feed-forward et les activations GELU.

Lorsque nous traitons une séquence d'entrée dans un bloc de transformer, chaque élément de la séquence, comme un mot ou un token, est représenté par un vecteur de taille fixe. Les opérations dans le bloc de transformer, notamment l'attention multi-tête et les réseaux feed-forward, transforment ces vecteurs sans altérer leur dimensionnalité.

L'objectif est que l'attention multi-tête analyse les relations entre les éléments de la séquence d'entrée. Ensuite, le réseau feed-forward modifie les données individuellement à chaque position de la séquence. Cette combinaison permet au modèle de mieux comprendre l'entrée dans son ensemble tout en traitant chaque élément de manière indépendante, ce qui améliore la capacité du modèle à gérer des patterns complexes.

Enfin, ce bloc transforme les vecteurs d'entrée de manière qu'ils intègrent le contexte global de la séquence, tout en maintenant les dimensions d'origine. Ainsi, le modèle peut traiter

efficacement des tâches de transformation de séquences tout en maintenant une relation directe entre les éléments d'entrée et leurs représentations contextuelles dans la sortie.



5.6. Codage du modèle GPT

La dernière étape consiste à assembler tous les composants du modèle : les couches d'intégration des tokens et des positions, les blocs de transformateurs, ainsi que les couches de normalisation et de sortie. Cette combinaison permet de former l'architecture complète du modèle GPT, capable de traiter les entrées et de prédire la suite d'une séquence.

À ce stade de l'architecture, le modèle ne génère pas correctement, car il n'a pas encore été entraîné. Dans cette lecture individuelle, nous avons simplement vu l'implémentation de l'architecture GPT et initialisé une instance de modèle avec des poids aléatoires.