Large Language Models

Introduction

Les modèles de langage de grande taille (LLM) représentent une avancée majeure dans le domaine de l'intelligence artificielle, avec des applications variées allant de la génération de texte à l'analyse sémantique et à la traduction automatique. En s'appuyant sur des architectures de réseaux neuronaux, notamment les modèles Transformers, les LLM permettent de traiter, comprendre et générer du langage naturel de manière performante et cohérente.

Nous explorerons les principes fondamentaux qui sous-tendent le fonctionnement des LLM, en passant par les bases de leur entraînement massif sur des corpus diversifiés, les mécanismes d'attention introduits par les Transformers, ainsi que par les phases de fine-tuning. En étudiant en profondeur les modèles comme GPT, nous examinerons également les architectures qui permettent aux LLM de capter les complexités et les nuances du langage humain.

Nous verrons finalement comment ces modèles peuvent être utilisé dans des architectures plus globales comme les RAG (Retrieval Augmented Generation).

Contents

1 Introduction Générale aux LLM

1.1 Origine historique

Les Large Language Models (LLM) résultent des avancées significatives du Machine Learning et du traitement du langage naturel (NLP) au cours de la dernière décennie. Des modèles comme BERT (Bidirectional Encoder Representations from Transformers) de Google et GPT (Generative Pre-trained Transformer) d'OpenAI ont été pionniers dans ce domaine, établissant les bases des LLM modernes.

Développé par Google en 2018, BERT a été un des modèles pionniers. Ce dernier a introduit une approche bidirectionnelle pour comprendre le contexte des mots dans une phrase. Contrairement aux modèles précédents qui lisaient le texte séquentiellement de gauche à droite ou de droite à gauche, BERT considère l'ensemble du contexte entourant chaque mot simultanément. Cette capacité a permis à BERT d'atteindre des performances inégalées dans diverses tâches NLP comme la classification de texte, la réponse à des questions, et plus encore.

Par la suite, OpenAI a développé GPT qui deviendra par la suite une référence en matières de LLM. Le modèle GPT-2, publié en 2019, a démontré une capacité impressionnante à générer du texte de manière cohérente et créative en se basant sur un petit texte d'amorce. GPT fonctionne en deux étapes : le pré-entraînement sur un énorme corpus de texte pour apprendre la structure du langage et le fine-tuning sur des tâches spécifiques pour affiner ses performances. GPT-3, sorti en 2020, a continué à repousser les limites en termes de taille et de capacité, avec 175 milliards de paramètres, offrant des résultats encore plus impressionnants dans la génération de texte et les applications conversationnelles.

Les LLM continuent de croître en taille et en sophistication, chaque nouvelle itération repoussant les limites de ce qui est possible en traitement du langage naturel. Les modèles les plus récents intègrent des techniques avancées pour améliorer l'efficacité de l'entraînement et réduire les biais présents dans les données d'entraînement. Ces modèles ouvrent de nouvelles perspectives pour l'intelligence artificielle, notamment dans des domaines tels que la traduction automatique, la génération de contenu créatif, les chatbots intelligents, l'assistance virtuelle, et bien d'autres applications pratiques.

1.2 Fonctionnement général

Pour comprendre comment fonctionnent les LLM, il faut d'abord saisir leur principe de base. Un LLM est, en quelque sorte, un "prédicteur de mots". En d'autres termes, son objectif principal est de deviner le mot qui vient après une séquence de mots donnée.

Lorsque nous entrons une phrase, le modèle analyse chaque mot et tente de prédire le mot suivant en fonction de ce qu'il a appris pendant son entraînement. Cet entraînement repose sur des milliards de mots, issus de divers textes, articles, livres, forums en ligne, et bien d'autres sources. Le modèle apprend à identifier des schémas dans le langage, c'est-à-dire des relations entre les mots, les phrases et les structures grammaticales.

Par exemple, si on lui donne la phrase "Le chat chasse le...", le modèle va anticiper que le mot suivant pourrait être "souris" ou un autre mot en lien avec le contexte. Grâce aux énormes bases de données utilisées pour l'entraînement, le modèle a rencontré cette situation (ou des situations similaires) des millions de fois, ce qui lui permet de faire des prédictions cohérentes et logiques.

1.3 Entraînement et Fine Tuning

L'entraînement des modèles LLM est une étape très importante qui permet au modèle d'apprendre à générer du texte de manière cohérente et pertinente. Durant cette phase, les différentes parties du LLM

vont intéragir de manière à apprendre les représentations linguistiques. On peut résumer l'entraînement d'un modèle LLM en 3 grandes étapes :

1. **Propagation Avant (Forward Pass) :** Lors de chaque étape d'entraînement, une séquence de tokens est passée à travers le modèle afin de prédire une séquence de tokens de sortie. On va utiliser les tokens de sortie pour évaluer les performances du modèle et calculer une fonction de perte.

2. **Rétropropagation (Backward Pass) :** La perte calculée à partir des prédictions est utilisée pour ajuster les paramètres du modèle. La rétropropagation des gradients est effectuée à travers toutes les couches du modèle, mettant à jour les poids pour minimiser la fonction de perte. Les gradients sont calculés en utilisant la dérivée de la fonction de perte par rapport aux paramètres du modèle.

3. **Mise à Jour des Paramètres :** Les paramètres du modèle sont mis à jour en utilisant les gradients calculés et l'algorithme d'optimisation choisi (par exemple, Adam). Cette mise à jour se fait de manière itérative pour chaque lot de données d'entraînement.

Plus précisément, l'entraînement d'un LLM se déroule en deux phases principales : le **pré-entraînement** et le **fine-tuning**.

1.3.1 Pré-entraînement

Cette phase consiste à entraîner le modèle sur un vaste corpus de texte pour apprendre les structures linguistiques et les relations contextuelles.

- **Objectif de Pré-formation :** Le modèle est pré-entraîné pour prédire le token suivant dans une séquence donnée. Cette tâche est connue sous le nom de modélisation du langage auto-régressive. L'objectif est de maximiser la probabilité de la séquence de tokens dans le corpus d'entraînement.

$$L = -\sum_{i=1}^{N} \log P(t_i | t_1, t_2, \ldots, t_{i-1})$$

  où t_i est le i-ème token dans la séquence et N est la longueur de la séquence.

- **Corpus d'Entraînement :** Le modèle est pré-entraîné sur un large corpus de texte provenant de diverses sources comme des livres, des articles, des sites web, etc. Ce corpus est non étiqueté et très diversifié pour que le modèle puisse capturer une vaste gamme de structures et de styles linguistiques. L'objectif est de maximiser la variété du contenu pour que le modèle soit à l'aise et arrive à comprendre tout type de langage et de style d'expression. Pour prendre un exemple, GPT4 s'est entraîné sur 1 petabytes de données, ce qui correspond à 1 000 000 de gigaoctets, soit 227 000 fois le contenu de wikipedia.

- **Optimisation :** L'optimisation est réalisée en utilisant des méthodes comme Adam ou AdamW, qui sont des variantes de la descente de gradient stochastique adaptées pour les réseaux de neurones profonds.

- **Régularisation :** Des techniques de régularisation comme le dropout ou encore des couches de normalisation sont utilisées pour prévenir le surapprentissage et améliorer la généralisation du modèle.

1.3.2 Fine-Tuning

Une fois le pré-entraînement fait, le modèle subit une phase de fine-tuning pour l'adapter à des tâches spécifiques avec des données étiquetées. Cette phase permet d'ajuster les paramètres du modèle pour qu'il excelle dans des applications particulières.

- **Objectifs :** Pendant le fine-tuning, le modèle est entraîné sur un ensemble de données spécifique à une tâche avec une fonction de perte adaptée à cette tâche. Par exemple, pour la génération de texte conditionnée, la fonction de perte pourrait être ajustée pour maximiser la probabilité des séquences de sortie conditionnées par des entrées spécifiques.

$$L = -\sum_{i=1}^{N} \log P(t_i | t_1, t_2, \ldots, t_{i-1}, c)$$

  où c représente le contexte ou la condition spécifique de la tâche.

- **Données d'Entraînement :** Les données utilisées pour le fine-tuning sont spécifiques à la tâche et souvent plus petites que celles utilisées pour le pré-entraînement. Ces données sont étiquetées et organisées en paires entrée-sortie pertinentes pour la tâche ciblée.

- **Ajustement des Hyperparamètres :** Les hyperparamètres du modèle, comme le taux d'apprentissage, le nombre d'epochs, et la taille des batchs, sont ajustés pour optimiser les performances sur la tâche spécifique. Une recherche d'hyperparamètres peut être effectuée pour trouver les meilleures configurations.

- **Régularisation et Techniques Avancées :** Des techniques de régularisation avancées, telles que l'early stopping, peuvent être employées pour éviter le surapprentissage.

1.4 Enjeux et Limitations

Les LLM ont ainsi révolutionné le traitement du langage naturel, offrant des capacités impressionnantes en termes de génération de texte et de compréhension contextuelle. Cependant, malgré leurs nombreux avantages, ils présentent également certaines limitations significatives.

- **Hallucination et Génération d'Informations Incorrectes** : L'un des problèmes majeurs des LLM est leur tendance à halluciner, c'est-à-dire à générer des informations incorrectes ou inventées. Les modèles peuvent produire des réponses qui semblent plausibles mais qui ne sont pas factuellement exactes. Cela est particulièrement problématique dans les applications nécessitant une précision élevée.

- **Dépendance aux Données d'Entraînement** : Les LLM sont fortement dépendants des données sur lesquelles ils ont été entraînés. Si les données d'entraînement contiennent des biais, des stéréotypes ou des informations obsolètes, le modèle risque de reproduire ces biais dans ses réponses. Par conséquent, les LLM peuvent parfois fournir des réponses discriminatoires, non éthiques ou encore sexistes.

- **Coût Computationnel et Ressources Nécessaires** : L'entraînement et le déploiement de LLM nécessitent des ressources computationnelles importantes. L'entraînement d'un modèle peut coûter des millions de dollars en ressources de calcul et consommer une quantité d'énergie considérable. Pour prendre un exemple, le modèle GPT4 d'OpenAI a coûté 60 millions dollars simplement pour l'entraînement. De plus, l'utilisation de ces modèles en production peut également nécessiter des infrastructures puissantes pour assurer des temps de réponses rapides et une disponibilité élevée.

2 Introduction aux modèles Transformer

Avant d'expliquer plus en profondeur le fonctionnement des LLM modernes, il est important de se pencher sur un élément central de leur architecture : les modèles Transformers. C'est en effet grâce aux Transformers que les LLM ont atteint des niveaux de performance inégalés, rendant possible la génération de texte de manière cohérente et fluide.

2.1 Définition et Structure

Un Transformer est un type de modèle de réseau neuronal introduit en 2017 dans l'article "Attention Is All You Need" par Vaswani et ses co-auteurs. Contrairement aux modèles séquentiels traditionnels comme les réseaux de neurones récurrents (RNN) ou les réseaux LSTM, qui traitent le texte mot par mot ou étape par étape, les Transformers sont capables d'analyser une séquence de mots dans son ensemble, en portant une attention particulière aux relations entre tous les mots de la séquence.

Le principe clé derrière les Transformers est ce qu'on appelle le mécanisme d'attention. Ce mécanisme permet au modèle de se concentrer sur des mots spécifiques dans une phrase, en fonction du contexte, et d'ignorer ceux qui sont moins pertinents pour une tâche donnée. Par exemple, dans la phrase "Le chat noir saute par-dessus le mur", le Transformer peut déterminer que "chat" et "saute" sont fortement liés, car le chat est le sujet de l'action, tandis que "noir" est un adjectif descriptif.

2.1.1 Encodage et Décodage

L'architecture d'un Transformer se divise généralement en deux parties : l'encodeur et le décodeur.

- **L'encodeur :** Cette partie du modèle prend une séquence de texte en entrée et transforme chaque mot en une représentation vectorielle. L'encodeur utilise une série de couches d'attention pour créer des vecteurs qui capturent le sens de chaque mot en tenant compte de son contexte. Il aide le modèle à comprendre la bonne signification en fonction des mots voisins.
- **Le décodeur :** Il génère la séquence de sortie en se basant sur les informations capturées par l'encodeur. Pour un LLM qui génère du texte, le décodeur est responsable de produire chaque mot de la réponse, en tenant compte du contexte fourni par l'encodeur. Ainsi, il "décide" du mot à générer en fonction des relations entre les mots identifiées précédemment.

2.1.2 L'auto-attention et le multi-head attention

L'élément qui distingue vraiment le Transformer est le concept d'auto-attention, qui permet au modèle de déterminer quelles parties de la séquence sont les plus importantes pour comprendre un mot donné. Par exemple, dans une longue phrase, l'auto-attention permet de concentrer l'analyse sur les mots clés en rapport avec le mot en cours de traitement, plutôt que de traiter chaque mot comme également important.

Le multi-head attention est une extension de l'auto-attention : il permet au modèle d'appliquer plusieurs "têtes d'attention", c'est-à-dire différents points de vue, en parallèle. Chaque "tête" analyse la séquence sous un angle différent, capturant ainsi différents types de relations contextuelles. Par exemple, une tête d'attention pourrait se focaliser sur la relation sujet-verbe, tandis qu'une autre pourrait se concentrer sur les adjectifs.

En unissant les mécanismes d'encodeur et de décodeur et en appliquant l'attention, les Transformers permettent aux modèles de traiter des séquences plus rapidement et de façon plus flexible que les approches séquentielles. Grâce à cette capacité, ils peuvent gérer de longs contextes sans perdre de vue les relations importantes entre les mots.

2.2 Transformers vs Modèles Séquentiels

Pour bien comprendre pourquoi les Transformers ont été une révolution, il est utile de les comparer avec les modèles séquentiels traditionnels, comme les RNN (Réseaux de Neurones Récurrents) et les LSTM (Long Short-Term Memory). Avant l'arrivée des Transformers, ces modèles étaient l'état de l'art dans le traitement du langage naturel. Cependant, ils présentaient plusieurs limitations que les Transformers ont permis de surmonter.

2.2.1 Les modèles séquentiels : RNN et LSTM

Les RNN et LSTM sont des modèles conçus pour traiter les données séquentielles. Ils traitent les mots d'une phrase un par un, en conservant une "mémoire" du contexte de la phrase au fur et à mesure de la lecture. Par exemple, dans la phrase "Le chat noir saute par-dessus le mur", un modèle séquentiel lira chaque mot dans l'ordre et conservera progressivement l'information pour comprendre le contexte. Cette façon de traiter l'information pose plusieurs problèmes :

- Problèmes de dépendance à long terme : Plus la séquence est longue, plus il devient difficile pour le modèle de se souvenir des premiers mots, ce qui est un défi pour capturer le contexte global dans les phrases ou paragraphes longs. Les LSTM améliorent un peu ce point, mais restent limités dans leur capacité à conserver le contexte.
- Traitement lent et séquentiel : Parce qu'ils traitent chaque mot l'un après l'autre, les RNN et LSTM sont plus lents et difficiles à paralléliser sur des architectures modernes de calcul, comme les GPU. Cela rend l'entraînement de ces modèles long et coûteux en ressources.

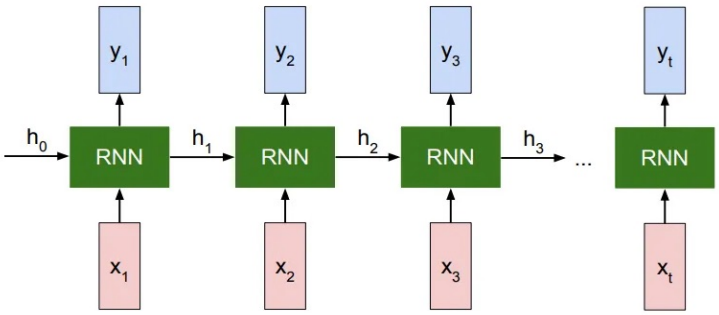

Figure 1: Architecture RNN

2.2.2 Avantage des Transformers

Les Transformers, en revanche, ne traitent pas les mots de manière séquentielle. Grâce à leur architecture d'attention, ils analysent tous les mots de la phrase simultanément, ce qui leur permet de mieux capter les relations globales dans une phrase, indépendamment de la distance entre les mots. Les 3 grands avantages sont donc :

- **Capacité de traiter les dépendances longues** : Les Transformers peuvent établir des connexions entre des mots éloignés sans "oublier" les premiers mots de la phrase. Cela les rend bien plus efficaces pour comprendre le contexte dans des phrases ou documents longs. Par exemple, dans un texte où le sujet est introduit au début et référencé à la fin, un Transformer est capable de faire ces connexions naturellement.

- **Parallélisation** : Comme les Transformers traitent toute la phrase en une seule fois, ils sont parfaitement adaptés aux architectures de calcul parallèles, comme les GPU. Cela permet d'entraîner des modèles plus grands et plus puissants dans un temps réduit, ce qui est crucial pour les LLM actuels, qui peuvent contenir des milliards de paramètres.
- **Flexibilité dans l'apprentissage des relations contextuelles** : Le mécanisme d'attention des Transformers leur permet de se concentrer sur différentes parties de la phrase selon les besoins. Cela signifie qu'ils sont capables de s'adapter à des contextes variés et de reconnaître les nuances linguistiques de manière bien plus précise que les modèles séquentiels.

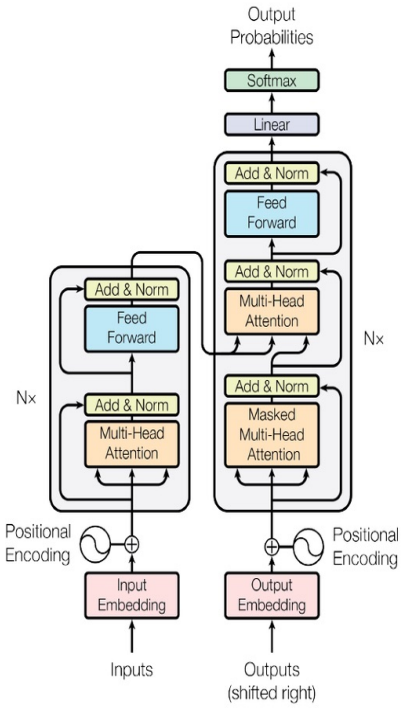

Figure 2: Architecture transformers

2.3 Transformers dans les LLM Modernes

2.3.1 GPT et la génération de texte

GPT (Generative Pre-trained Transformer) est l'un des premiers modèles à avoir popularisé l'architecture Transformer pour des tâches de génération de texte. Son fonctionnement repose uniquement sur une partie de l'architecture Transformer : le décodeur. En effet, GPT utilise plusieurs couches de décodeurs pour produire du texte en se basant sur le contexte des mots précédents dans une séquence.

GPT est un exemple d'un LLM « auto-régressif », ce qui signifie qu'il prédit chaque mot successif de manière séquentielle, mais il le fait en utilisant l'attention et la capacité du Transformer à analyser l'ensemble de la séquence en parallèle, avant de générer le mot suivant. Cela lui permet de créer du texte de manière fluide, en gardant la cohérence et le contexte tout au long des phrases et des paragraphes.

2.3.2 BERT et la compréhension de texte

BERT (Bidirectional Encoder Representations from Transformers) est un autre modèle Transformer qui s'est imposé pour des tâches de compréhension de texte. Contrairement à GPT, qui utilise le décodeur

du Transformer, BERT repose uniquement sur l'encodeur, ce qui lui permet d'analyser un texte de manière bidirectionnelle. Cela signifie qu'il lit la phrase complète, en prenant en compte à la fois les mots qui précèdent et ceux qui suivent le mot analysé.

Cette approche bidirectionnelle rend BERT particulièrement efficace pour les tâches de compréhension, comme la classification de texte, l'analyse des sentiments, et la recherche de réponses dans un document. Son objectif est d'apprendre des représentations riches pour chaque mot en fonction de son contexte global, ce qui lui donne une grande précision dans la compréhension des nuances du langage.

2.3.3 Différentes architectures basées sur les Transformers

Depuis la sortie de GPT et BERT, d'autres modèles basés sur les Transformers ont été développés pour répondre à des besoins plus spécifiques et améliorer encore les performances :

- T5 (Text-To-Text Transfer Transformer) : T5 adopte une approche unifiée en traitant toutes les tâches de NLP sous la forme d'une tâche de génération de texte. Que ce soit pour une traduction, un résumé ou une classification, T5 convertit chaque tâche en un format "texte à texte", ce qui le rend extrêmement polyvalent.
- RoBERTa : Une version optimisée de BERT, qui ajuste certains aspects de l'entraînement pour améliorer les performances. RoBERTa utilise des volumes de données plus importants et des périodes d'entraînement plus longues, ce qui lui permet d'atteindre une précision accrue sur certaines tâches.
- DistilBERT : Ce modèle est une version plus légère de BERT, conçue pour fonctionner plus rapidement et consommer moins de ressources tout en maintenant une bonne performance. DistilBERT est souvent utilisé dans des environnements où la puissance de calcul est limitée.

3 Approfondissement des Transformers

3.1 Introduction du mécanisme d'attention

Les Transformers introduisent le mécanisme d'attention pour surmonter la limitation des réseaux récurrents. Plutôt que de compter uniquement sur l'état caché précédent pour effectuer les prédictions, le modèle utilise toute la séquence d'entrée pour déterminer les informations pertinentes.

Imaginons une séquence de trois mots, chacun représenté par une entrée X_1, X_2, et X_3. Pour chaque entrée, le Transformer génère trois vecteurs :

- **Query** Q : La requête, qui guide le modèle pour identifier les informations pertinentes.
- **Key** K : La clé, permettant de comparer chaque entrée avec les requêtes.
- **Value** V : La valeur, représentant l'information de chaque entrée.

Pour chaque mot, le modèle compare sa requête avec les clés de toutes les autres entrées via un produit scalaire. Plus le produit scalaire est élevé, plus la similarité entre la requête et la clé est forte, indiquant une information potentiellement pertinente.

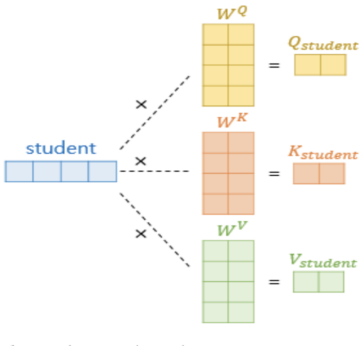

Figure 3: Calcul des vecteurs Q, K et V

3.1.1 Calcul des scores d'attention

Pour prendre un exemple, on souhaite calculer la valeur de h_2, qui correspond à l'état caché de X_2. Pour cela on va commencer par calculer les scores d'attention de chaque entrée X_i en multipliant le vecteur de Query associé à X_2, Q_2, avec l'ensemble des keys. On fait également passer le résultat de ce produit dans une fonction Softmax pour normaliser les résultats de sorte à ce que leurs sommes fassent 1. On a alors :

$$s_1^{(2)} = Softmax(Q_2 k_1)$$
$$s_2^{(2)} = Softmax(Q_2 k_2)$$
$$s_3^{(2)} = Softmax(Q_2 k_3)$$

On peut donc calculer h_2 :

$$h_2 = s_1^{(2)} v_1 + s_2^{(2)} v_2 + s_3^{(2)} v_3$$

Ainsi, chaque entrée influe plus ou moins sur h_2 en fonction de sa pertinence avec Q_2, et donc selon son rapport avec l'entrée initiale X_2.

3.1.2 Parralélisation des calculs

On peut ainsi voir les calculs d'états cachés sont indépendants entre eux, ce qui signfie qu'on peut parraléliser leurs calculs. Pour cela on va introduire plusieurs matrices :

$$\mathbf{Q} = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix} \qquad \mathbf{K} = \begin{pmatrix} k_1 \\ k_2 \\ k_3 \end{pmatrix} \qquad \mathbf{V} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

où Q représente le vecteur des Query, K le vecteurs des Keys et V le vecteur des Value. On peut alors calculer notre matrice de scores d'attention de la façon suivante :

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\mathbf{Q}\mathbf{K}^\top)\mathbf{V}$$

On commence par calculer $\mathbf{Q}\mathbf{K}^\top$:

$$\mathbf{Q}\mathbf{K}^\top = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \end{pmatrix} \begin{pmatrix} k_1 & k_2 & k_3 \end{pmatrix} = \begin{pmatrix} q_1k_1 & q_1k_2 & q_1k_3 \\ q_2k_1 & q_2k_2 & q_2k_3 \\ q_3k_1 & q_3k_2 & q_3k_3 \end{pmatrix}$$

Ici, on va rajouter une étape de masquage. La partie supérieure à la diagonale de la matrice correspond à une comparaison entre des clés futurs par rapport à la requête. Par exemple, le terme q_1k_3 signifie qu'on cherche à connaître le rapport entre le premier mot de la séquence et le troisième. Or, comme on reçoit nos informations de manière séquentielle, la valeur de l'état caché h_1 ne peut pas dépendre de séquences futurs.

Pour résoudre ce problème on va ajouter la matrice M de telle sorte que l'on va réduire à moins l'infini les termes problématiques :

$$\begin{pmatrix} q_1k_1 & q_1k_2 & q_1k_3 \\ q_2k_1 & q_2k_2 & q_2k_3 \\ q_3k_1 & q_3k_2 & q_3k_3 \end{pmatrix} + \begin{pmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} q_1k_1 & -\infty & -\infty \\ q_2k_1 & q_2k_2 & -\infty \\ q_3k_1 & q_3k_2 & q_3k_3 \end{pmatrix}$$

Ensuite, pour obtenir la valeur de scores d'attention, on va appliquer la fonction softmax à notre résultat:

$$\begin{aligned} \text{Softmax}(\mathbf{Q}\mathbf{K}^\top + M) &= \text{Softmax}\left(\begin{pmatrix} q_1k_1 & -\infty & -\infty \\ q_2k_1 & q_2k_2 & -\infty \\ q_3k_1 & q_3k_2 & q_3k_3 \end{pmatrix} \right) \\ &= \begin{pmatrix} s_1^{(1)} & 0 & 0 \\ s_1^{(2)} & s_2^{(2)} & 0 \\ s_1^{(3)} & s_2^{(3)} & s_3^{(3)} \end{pmatrix} \end{aligned}$$

Enfin, il suffit de mutliplier la matrice de scores d'attention par le vecteur $\mathbf{V}$ de Values afin d'obtenir le vecteur $\mathbf{H}$ contenant les états cachés de chaque entrée X :

$$\text{Softmax}(\mathbf{Q}\mathbf{K}^\top + M)V = \begin{pmatrix} s_1^{(1)} & 0 & 0 \\ s_1^{(2)} & s_2^{(2)} & 0 \\ s_1^{(3)} & s_2^{(3)} & s_3^{(3)} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix}$$

Les Transformers n'intègrent pas d'information de position naturellement, contrairement aux RNN. Pour introduire un sens d'ordre, chaque entrée est donc combinée à un vecteur de position, permettant au modèle de distinguer les ordres des mots comme "le chat poursuit la souris" de "la souris poursuit le

chat".

3.1.3 Entraînement du Transformer

Les paramètres entraînables du Transformer incluent principalement trois matrices :

- W_Q pour transformer les entrées en vecteurs de requêtes,
- W_K pour les clés,
- W_V pour les valeurs.

Ces matrices sont ajustées pour optimiser la performance du modèle sur la tâche cible, en apprenant les coefficients optimaux.

3.2 Attention Head et Multi Layer

Nous allons maintenant voir comment augmenter la capacité d'un Transformer, en ajoutant davantage de paramètres pour le rendre plus puissant.

Il existe deux façons principales d'agrandir un Transformer :

- **Extension verticale** : Ajouter plus de couches.
- **Extension en profondeur** : Ajouter plusieurs têtes d'attention.

3.2.1 Extension en profondeur : Les multi-têtes d'attention

L'idée derrière l'utilisation de plusieurs têtes d'attention réside dans la spécialisation. Chaque tête va pouvoir analyser une caractéristique particulière des entrées. Pour schématiser, une tête pourrait s'attarder sur la nature des mots, une autre sur les lieux ou encore sur les personnes, ...

Pour chaque entrée, nous allons désormais utiliser plusieurs têtes d'attention. Ainsi, au lieu d'avoir une seule requête, une clé et une valeur, nous aurons y requêtes, y clés, et y valeurs, où y représente le nombre de têtes d'attention.

Chaque tête i est associée à trois matrices spécifiques, notées $W_Q^{(i)}$, $W_K^{(i)}$, et $W_V^{(i)}$, utilisées pour calculer les vecteurs $Q^{(i)}$, $K^{(i)}$, et $V^{(i)}$ pour chaque entrée. On obtient donc y états cachés par entrée. Pour obtenir l'état caché h de chaque entrée i, on va devoir concaténer les $h_i^1, ..., h_i^y$ états cachés en un état noté Z dans le schéma suivant.

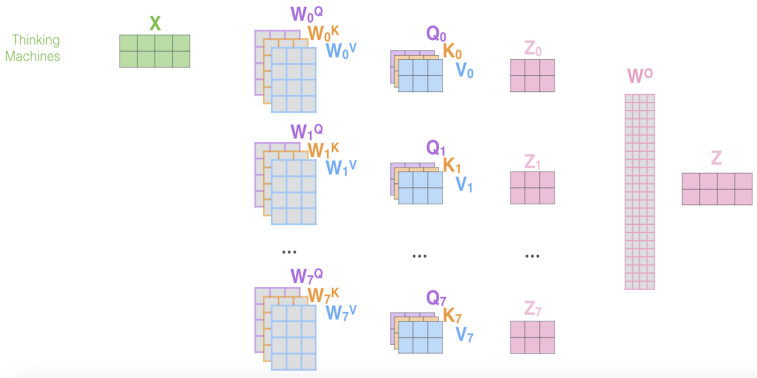

Figure 4: Têtes d'Attention Multiples

3.2.2 Extension verticale : Ajout de couches

Après avoir calculé les états cachés des différentes têtes et les avoir concaténés, on ne va plus prendre l'état caché obtenu comme état final. Le vecteur concaténé va ainsi passer dans une couche de réseau de neurones classique, où il est multiplié par une matrice de poids et appliqué à une fonction d'activation. Le résultat est noté $z_t^{(1)}$, où t est l'indice de l'entrée et 1 indique qu'il s'agit de la première couche.

L'ajout d'une couche de réseau de neurones avec une fonction d'activation permet d'introduire de la non-linéarité, rendant le modèle capable d'apprendre des relations plus complexes; là où le mécanisme d'attention, étant constitué de multiplications de matrices et de produits scalaires, est entièrement linéaire.

Chaque couche du Transformer est ainsi constituée d'un bloc d'attention (avec plusieurs têtes) suivi d'une couche de réseau de neurones. Il est donc possible de cumuler ces couches afin de rendre le modèle plus complexe, avec plus de paramètres à entraîner, et donc plus puissant. L'état caché devient donc la sortie de la toute dernière couche $z_t^{(n)}$, avec n le nombre de couches.

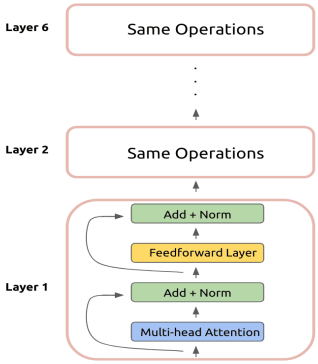

Figure 5: Schéma Architecture Multi Couches

3.3 Encodeur et Décodeur

Après avoir étudié les bases des Transformers et les méthodes pour les rendre plus puissants, nous allons maintenant explorer comment utiliser cette architecture pour diverses tâches, en particulier dans le domaine du NLP.

Deux architectures de Transformer sont utilisées selon les tâches :

- **L'encodeur** : Utilisé pour des tâches où la séquence entière est accessible avant la prédiction (ex. classification). Dans ce cas, l'attention n'est pas masquée, permettant au modèle de prendre en compte toutes les entrées.
- **Le décodeur** : Adapté pour des tâches séquentielles comme la génération de texte, où les entrées sont reçues progressivement. L'attention est masquée, restreignant le modèle aux entrées passées.

3.3.1 Génération de texte avec un décodeur

L'architecture de Transformer que nous avons vue jusqu'à présent, avec une attention masquée, convient parfaitement pour la génération de texte. Pour chaque mot généré, l'état caché associé, noté h_i, est utilisé pour prédire le mot suivant. Ce processus est itératif :

- On calcule h_1 pour obtenir le premier mot, qui devient l'entrée suivante.
- On continue ainsi avec h_2 pour obtenir le mot suivant, et ainsi de suite.

Pour chaque mot, on applique une couche Softmax sur l'état caché pour obtenir une distribution de probabilité sur tous les mots du vocabulaire. La génération est qualifiée d'**auto-régressive**, car chaque mot dépend des mots générés précédemment.

Pour l'entraînement d'un décodeur, on constitue une base de données constituée de phrases représentatives du style souhaité. Par exemple, pour générer des poèmes, on entraînera le modèle sur une base de données de poèmes.

On cherche à optimiser la probabilité de chaque mot du texte en maximisant la probabilité du mot cible pour chaque séquence d'entrée. Cette optimisation se fait via l'entropie croisée, en minimisant la différence entre la sortie du modèle et la séquence cible. Les gradients sont ensuite calculés avec la rétropropagation, et le modèle apprend à générer des phrases qui ressemblent aux exemples d'entraînement.

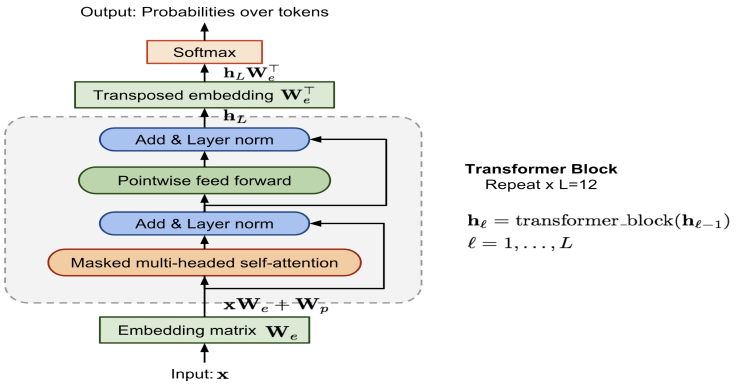

Figure 6: Architecture Décodeur Transformers

3.3.2 Classification avec un encodeur

Contrairement au décodeur, l'encodeur ne possède pas de masquage lors du calcul de l'attention. Ainsi, il peut être très utile pour des tâches comme la classification car il va permettre de capturer l'ensemble des relations entre tous les mots.

Pour faire de la classification de texte, on a besoin d'une sortie : le texte est positif ou négatif. Pour calculer cette sortie, on va utiliser l'ensemble des états cachés h_i calculés et les regrouper. Pour faire ce regroupement on peut simplement faire une moyenne, on obtiendra alors $\bar{h}$, puis faire passer ce vecteur dans une fonction d'activation Sigmoïde pour récupérer une probabilité.

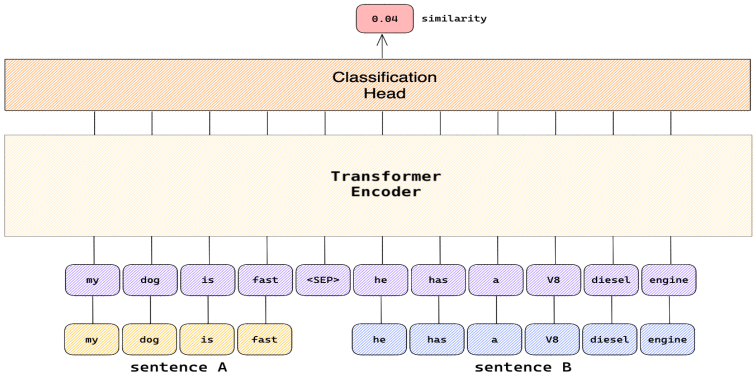

Figure 7: Classification avec Encodeur

3.3.3 Combinaison Encodeur/Décodeur

Il est possible de combiner un encodeur et un décodeur pour les tâches de type séquence à séquence, comme la traduction automatique :

- L'encodeur analyse la séquence d'entrée (ex. phrase à traduire), calcule les états cachés, puis les transmet au décodeur.
- Le décodeur génère la séquence de sortie (ex. phrase traduite) de manière auto-régressive en utilisant les états cachés de l'encodeur.

Dans le décodeur, chaque couche comprend deux mécanismes d'attention :

1. Une attention masquée, prenant en compte les états précédents du décodeur.
2. Une attention croisée, utilisant les clés et valeurs calculées par l'encodeur.

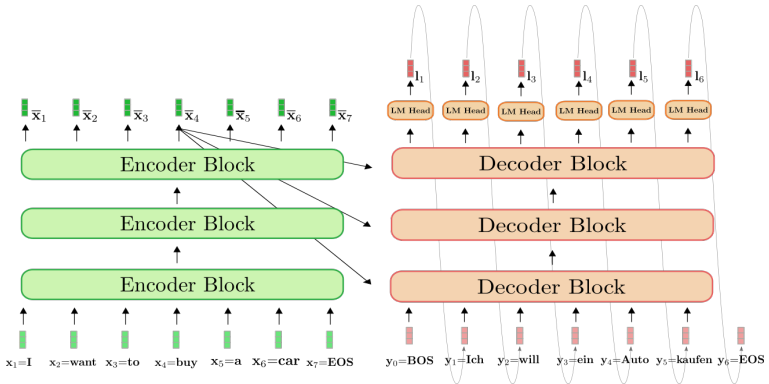

Figure 8: Exemple : Traduction avec Encodeur/Décodeur

4 Architecture et Fonctionnement des modèles GPT

4.1 Architecture

On va s'intéresser dans cette partie au modèle GPT d'OpenAI. Comme on vient de le voir, ce modèle est basé sur l'architecture Transformer.Il se compose de plusieurs types de réseaux de neurones dont :

4.1.1 Décodeur Transformer

Contrairement à certains modèles qui utilisent une architecture encoder-decoder complète, GPT utilise uniquement le décodeur de l'architecture Transformer. Ce décodeur est composé de plusieurs couches empilées identiques. Chaque couche du décodeur comprend :

1. **Masked Multi-Head Self-Attention :** Ce mécanisme permet au modèle de prêter attention à différentes parties du texte en parallèle. En d'autres termes, il aide le modèle à identifier et à se concentrer sur les mots importants dans le contexte de la séquence de texte. Le masquage empêche les positions futures d'influencer les positions actuelles, ce qui signifie que chaque prédiction ne dépend que des mots précédents dans la séquence.
2. **Add & Norm :** Après le mécanisme d'attention, chaque sous-couche (attention et feed-forward) est suivie d'une connexion résiduelle et d'une normalisation de la couche. La connexion résiduelle aide à éviter la dégradation des gradients dans les réseaux profonds, facilitant ainsi l'apprentissage. La normalisation de la couche stabilise et accélère l'entraînement en maintenant les activations des neurones dans des gammes raisonnables. Si l'entrée de la sous-couche est x et la sortie de la sous-couche est $\text{SubLayer}(x)$, alors la sortie après "Add & Norm" est :

$$\text{LayerNorm}(x + \text{SubLayer}(x))$$

3. **Réseau Feed-Forward Positionnel :** Ce composant applique une transformation non-linéaire à chaque position du texte de manière indépendante. Il aide le modèle à capturer des relations complexes dans les données. Composé de deux couches linéaires avec une activation ReLU au milieu, il augmente la capacité de modélisation du Transformeur.

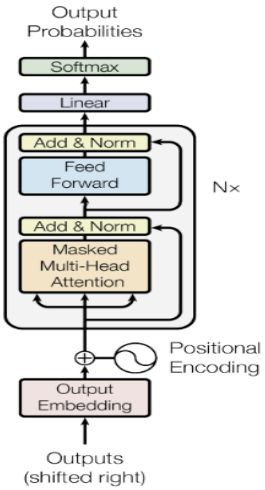

Figure 9: Architecture Modèle GPT

4.1.2 Embeddings

Les embeddings sont des représentations vectorielles des tokens (mots ou morceaux de mots) utilisées pour capturer les caractéristiques sémantiques des mots dans un espace de dimension fixe. Dans GPT, il y a deux types d'embeddings :

- **Embeddings de Tokens :** Chaque mot ou token du texte d'entrée est transformé en un vecteur dense de dimension fixe. Ces vecteurs permettent de représenter des mots ayant des significations similaires avec des représentations proches dans l'espace vectoriel.

  Supposons que la dimension de l'espace vectoriel soit d_{model}. Chaque token t_i est mappé à un vecteur d'embedding $e(t_i)$ de dimension d_{model}. Les embeddings de tokens sont appris pendant l'entraînement du modèle, ce qui signifie que les vecteurs sont ajustés pour minimiser la fonction de perte du modèle.

- **Embeddings de Position :** Pour inclure des informations de position, des embeddings de position sont ajoutés aux embeddings de tokens. Cela permet au modèle de prendre en compte l'ordre des mots dans une phrase, ce qui est essentiel pour capturer le contexte linguistique.

  Les embeddings de position p_{pos} sont des vecteurs de dimension d_{model} qui sont ajoutés aux embeddings de tokens. La formule pour calculer les embeddings de position est la suivante :

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$
$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

  où pos est la position dans la séquence et i est la dimension de l'embedding. Cette formulation permet de créer des vecteurs positionnels uniques pour chaque position dans la séquence, capturant ainsi les informations de position relative.

4.2 Fonctionnement

Le modèle GPT suit ainsi une architecture systématique et structurée pour transformer un texte d'entrée en une séquence générée de manière cohérente. Le processus peut être résumé de la manière suivante :

- **Input Text :** Le texte d'entrée est d'abord tokenisé, c'est-à-dire divisé en une série de tokens (mots ou morceaux de mots) utilisables par le modèle. Cette étape de tokenisation convertit le texte brut en une séquence de tokens $(t_1, t_2, \ldots, t_n)$.

- **Token Embeddings :** Chaque token dans la séquence est converti en un vecteur dense de dimension fixe. Mathématiquement, pour chaque token t_i, nous obtenons un vecteur $e(t_i)$.

- **Position Embeddings :** Pour inclure des informations sur l'ordre des tokens, des embeddings de position sont ajoutés aux embeddings de tokens.

- **Stack of Decoder Layers :** La séquence enrichie (embeddings de tokens + embeddings de position) est ensuite passée à travers une pile de couches de décodeur. Ce processus est répété pour un certain nombre de couches (typiquement 12, 24 ou 36 couches dans les modèles GPT-3).

- **Output Layer :** Enfin, la sortie de la dernière couche de décodeur est projetée dans l'espace des tokens pour prédire le token suivant dans la séquence. Cette prédiction est généralement

réalisée à l'aide d'une couche linéaire suivie d'une fonction softmax pour obtenir une distribution de probabilité sur le vocabulaire :

$$P(t_{n+1}|t_1, t_2, \ldots, t_n) = \text{softmax}(W_o h_n + b_o)$$

où h_n est la représentation finale du dernier token, W_o est la matrice de poids de sortie, et b_o est le biais de sortie.

5 RAG : Retrieval Augmented Generation

Les systèmes de génération augmentée par récupération d'informations (RAG) sont une évolution des LLM. Un RAG combine la capacité de génération d'un LLM avec un mécanisme de récupération d'informations pour améliorer la qualité et la précision des réponses du modèle. Ce mécanisme permet au modèle de rechercher et d'utiliser des informations spécifiques stockées dans une vaste base de données lors de la génération de réponses. Cela confère au modèle une capacité d'adaptation et d'exactitude supérieure dans des domaines spécifiques. De plus, cela permet de s'assurer (notamment via le prompts engineering) de la fiabilité des informations qu'il va utiliser pour générer sa réponse

5.1 Fonctionnement et Architecture

Le fonctionnement d'un RAG peut être divisé en plusieurs étapes clés :

- **Génération de la base de données :** Pour créer la base de données sur laquelle le RAG va s'appuyer, on transforme l'ensemble des documents par le biais d'embeddings. On représente l'ensemble des paragraphes sous forme de vecteurs, ce qui permet de capturer des relations sémantiques entre les mots et de les rendre compréhensibles par la machine.
- **Récupération d'informations :** Lorsqu'une requête est reçue, le RAG commence par identifier les segments de texte pertinents dans sa base de données. Le choix des segments à récupérer est souvent basé sur la similarité sémantique avec la requête. Cela est généralement réalisé en utilisant des techniques de recherche de similarité cosinus entre l'embedding de la requête et les embeddings des documents.

  $$\text{similarité}(q, d) = \frac{q \cdot d}{\|q\|\|d\|}$$

  où q est l'embedding de la requête et d est l'embedding d'un document. Les documents avec les scores de similarité les plus élevés sont récupérés. On retrouve également des scores de similarité via des distances euclidiennes.
- **Intégration des informations :** Les informations récupérées sont ensuite intégrées dans le processus de génération de texte. Le modèle utilise ces données pour informer sa réponse, assurant que la génération est non seulement pertinente mais aussi précise et basée sur des faits vérifiables.
- **Génération de réponse :** Avec les informations intégrées, le modèle génère une réponse qui reflète non seulement une compréhension contextuelle profonde mais est également enrichie par les données spécifiques récupérées. Cela permet de produire des réponses qui sont à la fois informatives et particulièrement adaptées à la requête.

5.2 Avantages

L'utilisation de RAG offre plusieurs avantages significatifs :

- **Amélioration de l'exactitude :** En s'appuyant sur des informations précises récupérées, les RAG peuvent fournir des réponses plus exactes que les LLM standards, surtout dans des domaines nécessitant une expertise spécifique. Cela permet de réduire les problèmes d'hallucination, où le modèle génère des informations incorrectes ou inventées.
- **Flexibilité et adaptation :** Les RAG peuvent s'adapter à de nouvelles informations et domaines rapidement, simplement en mettant à jour leur base de données de récupération, sans nécessité de

réentraînement complet du modèle. Cela permet une mise à jour et une maintenance plus efficaces, assurant que le modèle reste pertinent et précis à mesure que de nouvelles informations deviennent disponibles.

- **Efficacité contextuelle :** En intégrant des informations contextuelles spécifiques provenant de la récupération de documents, les RAG peuvent générer des réponses qui sont non seulement correctes mais aussi riches en contenu et en détail. Cela est particulièrement utile pour des applications telles que les outils de recherche spécialisée.
- **Réduction des coûts d'entraînement :** Puisque les RAG peuvent être mis à jour simplement en ajoutant ou en modifiant les documents dans leur base de données de récupération, ils réduisent les besoins en réentraînement complet du modèle de base. Cela conduit à une réduction des coûts computationnels et de temps associés à l'entraînement de grands modèles de langage.

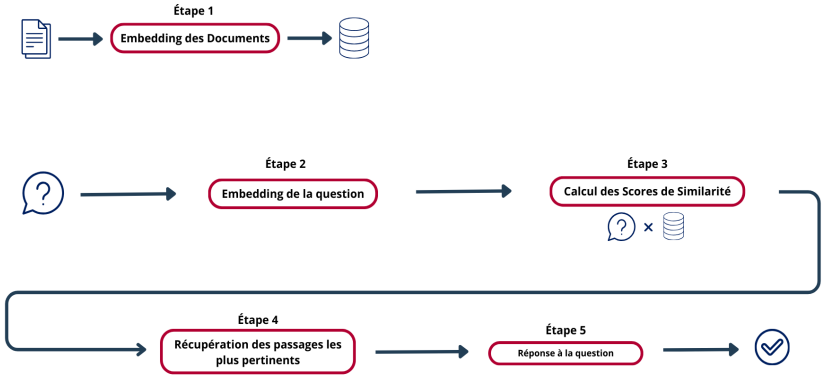

Figure 10: Schéma Architecture RAG