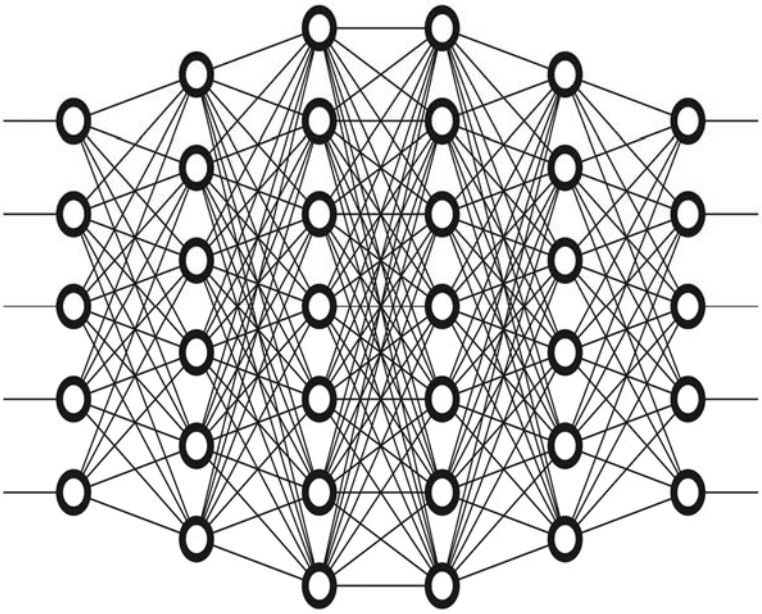




LLM : Large langage Model

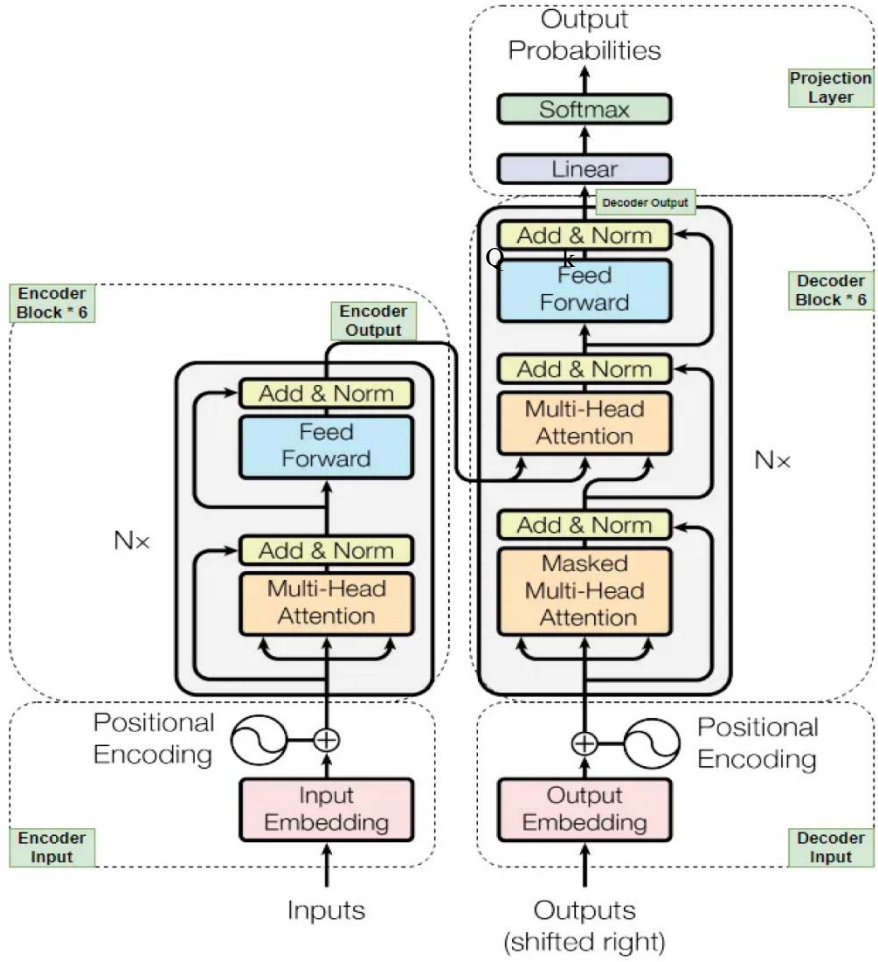


1 Présentation

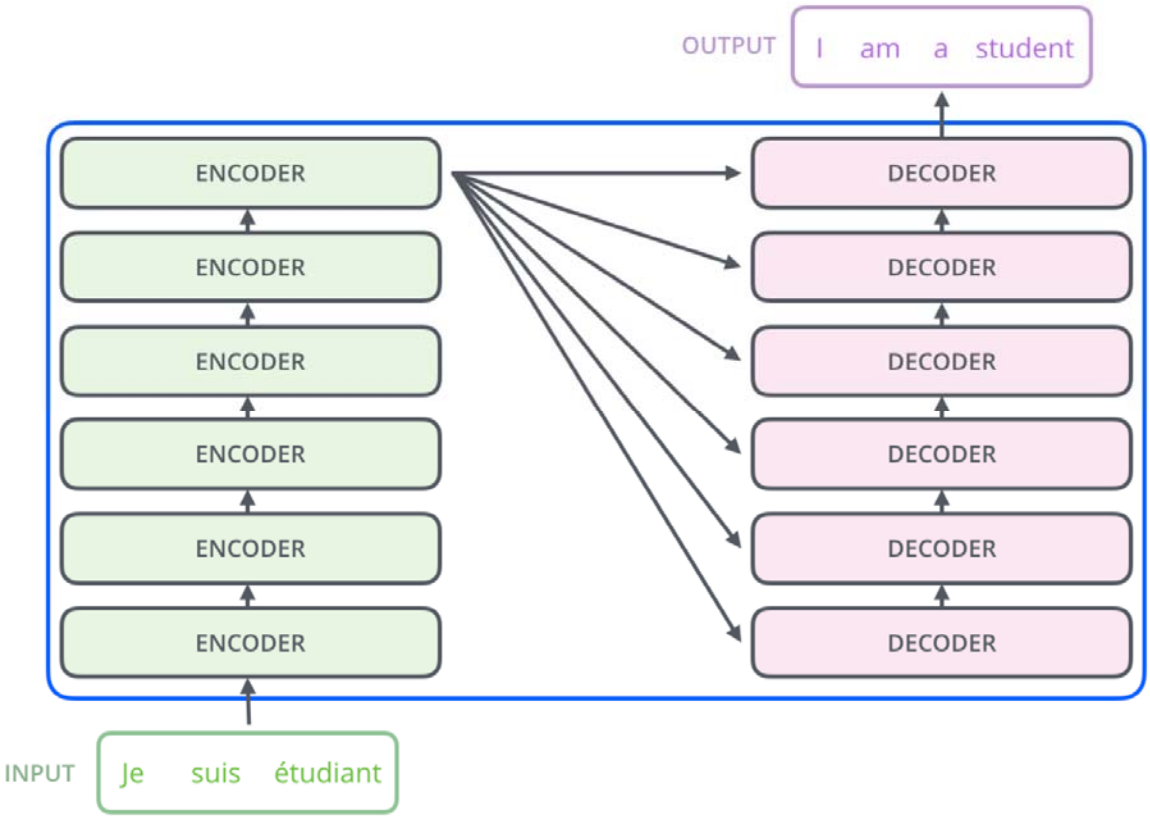
Les mécanismes d'attention ont révolutionné le domaine du traitement du langage naturel (**NLP natural langage processing)** ces dernières années, entraînant des améliorations significatives dans des tâches telles que la traduction automatique, la compréhension du langage et la génération de texte. Au coeur de cette innovation se trouve l'architecture transformer, qui est devenue le modèle de pointe pour de nombreuses tâches NLP.
L'architecture du transformer est basée sur le concept d'attention, qui permet au modèle de se concentrer sélectivement sur différentes parties de la séquence d'entrée lors du calcul de la sortie. Ceci est particulièrement utile pour les tâches NLP, où la séquence d'entrée peut être très longue et complexe, et les architectures récurrentes ou convolutives traditionnelles peuvent avoir du mal à capturer les informations contextuelles nécessaires.
les grands modèles de langages **LLM (large langage model)** sont à la base de **l'IA générative** les plus populaires tels que ChatGPT, Gemini, MetaAI, Mistral AI, etc.

2 L'architecture transformer

Au cœur même de chaque **LLM**, il y a une architecture appelée **Transformer**. Nous allons donc d'abord découvrir cette architecture de transformateur basée sur le célèbre article « L'attention est tout ce dont vous avez besoin »



Commençons en regardant le modèle comme une simple boîte noire. Pour une application de traduction à la machine, il prendrait à l'entrée une phrase dans une langue, et il sortirait sa traduction dans une autre langue.
Le composant d'encodage est une pile de six d'encodeurs Nx=6 voir schéma précédent . Le composant de décodage est une pile de six décodeurs . **Nx=96 pour GPT-3**
Les encodeurs sont tous identiques en structure (mais ils ne partagent pas les mêmes poids).



Chaque encodeur comprend deux sous-couches l'auto attention et le réseau de neurone

2.1 Principe de l'encodeur

L'encodeur traite les données d'entrée, comme une phrase, en les analysant simultanément pour créer une représentation numérique riche en contexte. Il est structuré en couches, utilisant des mécanismes d'attention pour se concentrer sur la compréhension, et non sur la génération de texte, et produit des vecteurs (embending) qui capturent la sémantique de l'entrée.

Le travail de l'encodeur consiste à traiter les données d'entrée (comme une phrase dans une langue pour la traduction, ou toute séquence de données pour d'autres tâches).

Il lit et comprend l'ensemble de la séquence d'entrée en une seule fois (grâce au mécanisme d'attention) et en crée une représentation. C'est comme créer une carte détaillée ou un résumé de l'entrée.

Il n'a pas à se soucier de générer un nouveau texte ; il se concentre uniquement sur l'analyse.

L'encodeur est généralement composé d'un empilement de couches identiques qui incluent des **réseaux d'attention et de rétroaction multi-têtes.**

La sortie de l'encodeur est un ensemble de vecteurs qui encapsulent les informations de la séquence d'entrée. Ces vecteurs portent un contexte pour chaque mot, en tenant compte de l'ensemble de la séquence.

Les jetons et vecteurs de plongement ou embending

Le texte est donc notre jeu de données (corpus) à entrainer . La taille du vocabulaire dans ce texte détermine le nombre total de mots uniques dans notre jeu de données. Il peut être calculé à l'aide de la formule ci-dessous, où N est le nombre total de mots dans notre jeu de données.

$$vocab_{size} = count(set(N))$$

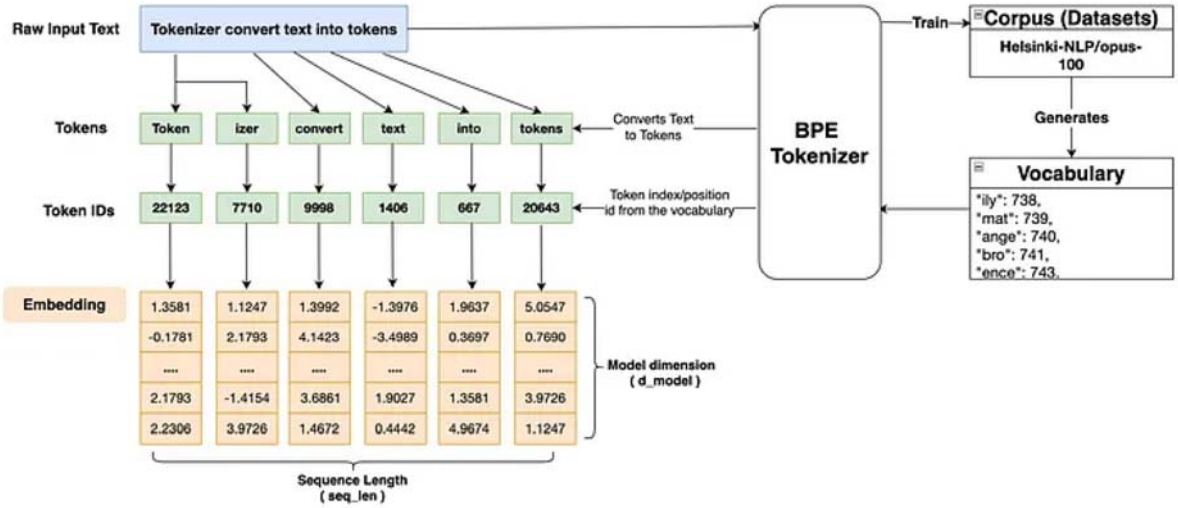
Après avoir obtenu N, nous effectuons une opération définie pour supprimer les doublons, puis nous pouvons compter les mots uniques pour déterminer la taille du vocabulaire.

Le modèle de transformateur ne traite pas le texte brut, il ne traite que les nombres compris dans les vecteurs de plongement ou embending.

Par conséquent, nous devrons faire quelque chose pour convertir le texte brut en nombres (vecteur d'embending ou plongement). Pour cela, nous pouvons utiliser par exemple un tokenizer populaire appelé BPE tokenizer qui est un tokenizer de sous-mot utilisé dans des modèles comme GPT3.

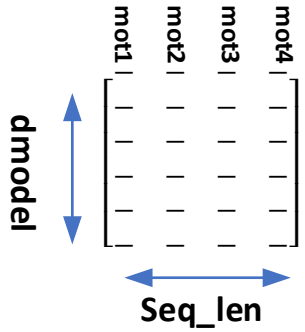
Le vocabulaire est une collection de jetons uniques à partir des données du corpus.

Le générateur de jetons BPE prend un texte brut, le mappe avec les jetons dans le vocabulaire et renvoie un jeton pour chaque mot dans le texte brut d'entrée. Les jetons peuvent être un seul mot ou un sous-mot.



Le générateur de jetons renvoie ensuite l'index unique ou l'ID de position du jeton dans le vocabulaire qui sera ensuite utilisé pour créer des intégrations comme indiqué dans le flux ci-dessus.
la séquence d'ID de jeton générée à partir des générateurs de jetons sera introduite dans la couche d'intégration. La couche d'intégration mappe l'identifiant du jeton au vocabulaire et génère un vecteur d'intégration de dimension **d_model**=512 pour chaque jeton.
Maintenant, nous devons attribuer un numéro unique à chaque mot unique.
Le vecteur d'intégration embending ou plongement peut capturer la signification sémantique du jeton en fonction du jeu de données d'entraînement sur lequel il a été entraîné. Chaque valeur de dimension à l'intérieur du vecteur d'intégration représente une sorte de caractéristique liée au jeton.
Par exemple, si le jeton est Chien, une valeur de dimension représenterait les yeux, la bouche, la jambe, la taille, etc. Si nous dessinons un vecteur dans un espace à n dimensions, les objets d'apparence similaire tels que les chiens et les chats seraient situés à proximité les uns des autres, et les objets d'apparence non similaire tels que les écoles, les vecteurs d'intégration à domicile seraient situés beaucoup plus loin.
Pour des raisons de simplicité dans se tutorial les vecteurs de plongement seront limités à une dimension de dvector = 6 et la séquence de mot à Seq_len=4. Pour GPT-3 Seq_len=128 000 et dvector = 122298

Vecteur d'embending ou plongement



Tous les embending et tokens sont traités en même temps en les empilant pour former une matrice de plongement seq_len par dmodel
un inconvénient est que lors du traitement de nombreuses séquences de jetons en parallèle, la **position des jetons dans une phrase ne sera pas dans l'ordre**. Cela pourrait potentiellement entraîner des significations ou des contextes de phrases différents selon l'endroit où les jetons sont positionnés. Par conséquent, pour résoudre ce problème, l'article d'attention implémente la méthode d'encodage positionnel.

2.2 L'encodage positionnel

Calcul de l'incorporation positionnelle

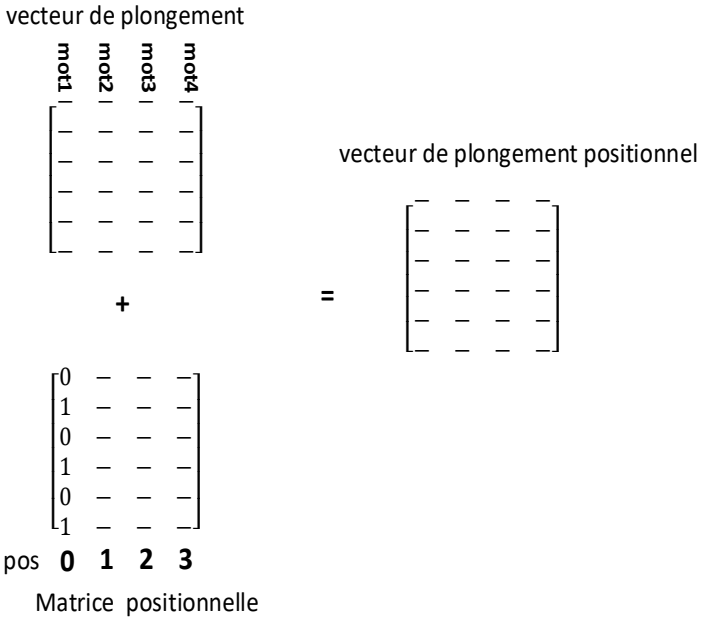
Le modèle doit connaître l'ordre des mots . Cette étape revient à étiqueter chaque mot avec une petite note indiquant où il se situe dans la ligne.
Maintenant, nous devons trouver des plongements positionnels pour notre entrée. Il existe deux formules pour le plongement positionnel en fonction de la position de la i^{ieme} valeur de ce vecteur de plongement pour chaque mot.
Pour la position paire

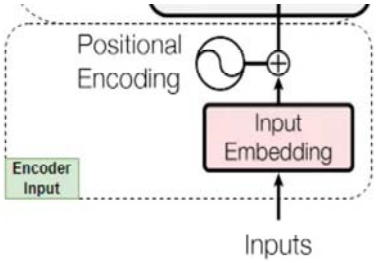
$$PE_{(pos,2i)} = \sin\left(\frac{pos}{1000^{\frac{2\times i}{dmodel}}}\right)$$

Pour la position impaire

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{1000^{\frac{2\times i}{dmodel}}}\right)$$

Intégration des plongements positionnels





2.3 L'attention de soi-même (Q K V)

L'un des avantages de l'architecture du transformateur est qu'elle peut traiter n'importe quel nombre de séquences d'entrée **en parallèle**, ce qui réduit beaucoup le temps d'entraînement et rend également la prédiction beaucoup plus rapide.
pourquoi avez-vous besoin d'attention personnelle ? Répondons à cette question avec un exemple simple ci-dessous.

Il va à la banque pour déposer de l'argent
Il fait une prise de sang pour la banque de sang

Dans les phrases 1 et 2, le mot « banque » a clairement deux significations différentes. Cependant, la valeur d'intégration du mot « banque » est la même dans les deux phrases. Ce n'est pas la bonne chose. Nous voulons que la valeur de plongement soit modifiée en **fonction du contexte de la phrase**. Par conséquent, nous avons besoin d'un mécanisme où la valeur de plongement peut changer dynamiquement pour donner le sens contextuel basé sur le sens global de la phrase. Le mécanisme d'auto-attention peut mettre à jour dynamiquement la valeur de plongement qui peut représenter le sens contextuel en fonction de la phrase.
Nous devrons donc trouver un moyen pour le mécanisme d'auto-attention d'apprendre ces multiples relations dans une phrase à la fois. C'est donc là **que l'auto-attention multi-têtes (l'attention multi-têtes peut être utilisée de manière interchangeable)** entre en jeu. Dans l'attention multi-têtes, les plongements à tête unique vont se diviser en plusieurs têtes afin que chaque tête examine différents aspects des phrases et apprenne en conséquence. C'est ce que nous voulons.

Dans un premier temps pour des raisons de simplicité prenant une seule tête pour le mécanisme d'auto-attention et faisons 3 copies de l'entrée de l'encodeur. Donnons à chacun d'eux un nom **Q, K et V.** Que sont les vecteurs « Q requête », « K clé » et « V valeur » ? Ce sont des abstractions utiles pour calculer et penser à l'attention.

Q (Requêtes)

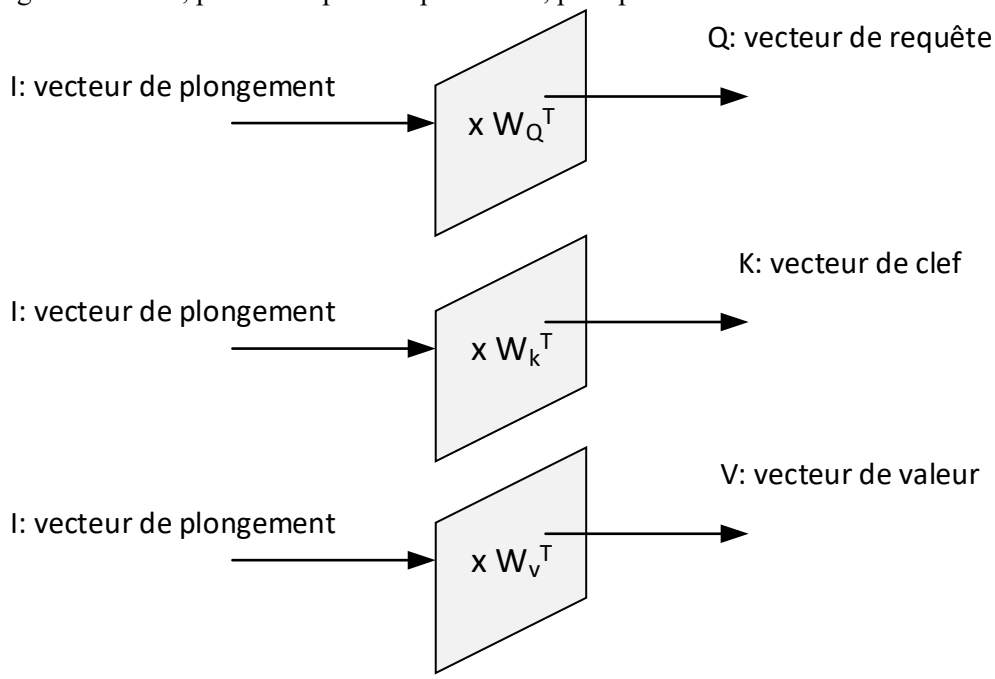
: Matrice contenant les vecteurs de requête. Ceux-ci représentent l'ensemble des éléments sur lesquels vous souhaitez attirer l'attention. Dans le contexte du traitement d'une phrase, une requête est généralement associée au mot sur lequel vous vous concentrez. Le modèle utilise la requête pour rechercher des informations pertinentes dans la séquence.

K (clés)

: Matrice contenant les vecteurs clés. Les clés sont associées à des valeurs et sont utilisées pour récupérer des informations. Chaque clé est associée à une valeur de manière à ce que le modèle puisse utiliser la similarité entre une requête et une clé pour déterminer le degré d'attention à accorder à la valeur correspondante.

V (Valeurs)

: Matrice contenant les vecteurs valeur. Les valeurs contiennent les informations réelles que le modèle souhaite récupérer. Une fois que le modèle a déterminé quelles clés (et donc valeurs) sont les plus pertinentes pour une requête donnée, il agrège ces valeurs, pondérées par leur pertinence, pour produire la sortie.



Chacun d'entre eux **Q, K, V** n'est rien d'autre qu'une copie de l'entrée de l'encodeur. Forme d'entrée de l'encodeur : (seq_len, d_model), **seq_len** : longueur maximale de la séquence, **dvector** : la dimension vectorielle d'intégration dans ce cas est 512 (pour notre exemple 6).

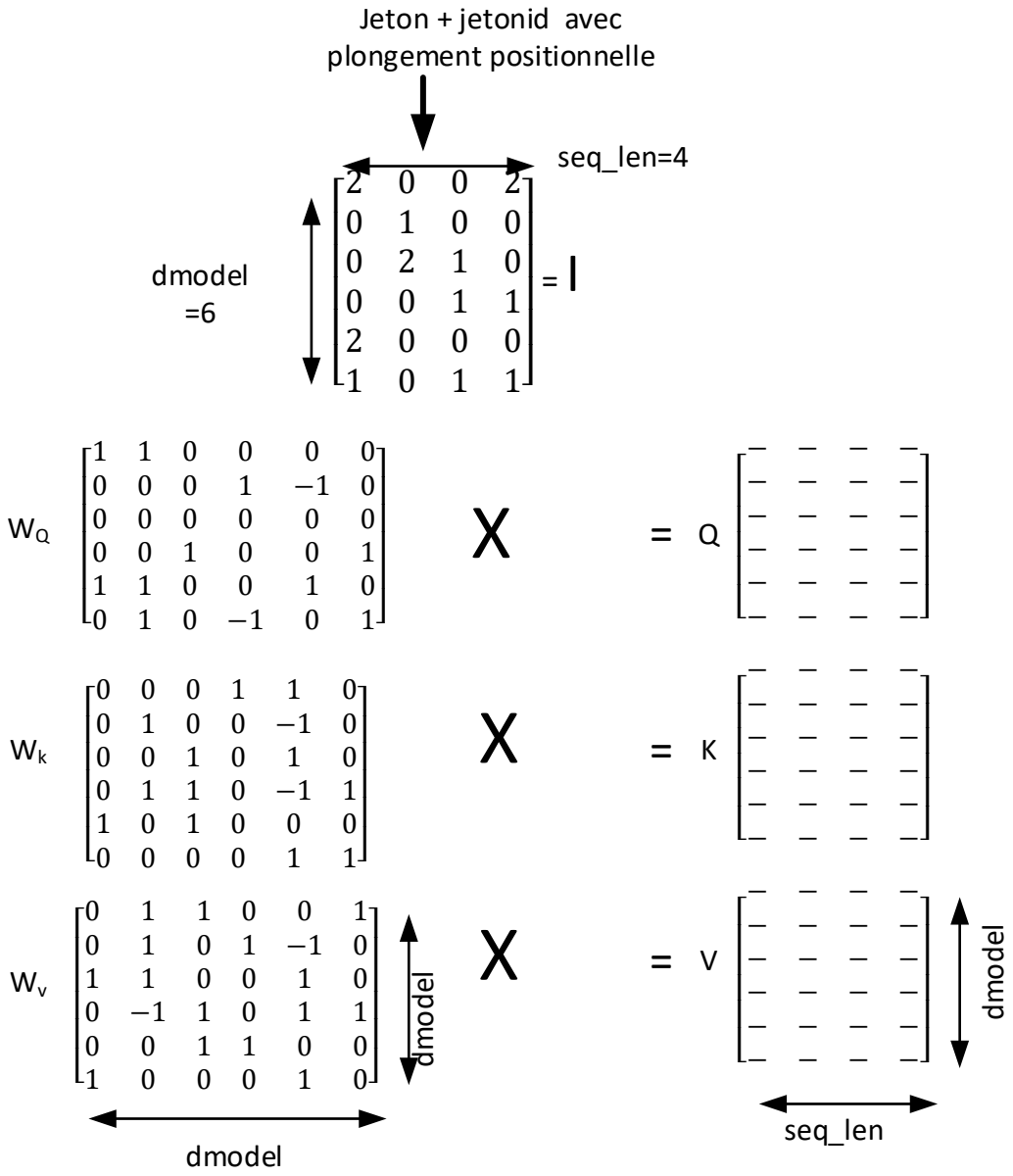
L'intuition est donc qu'un mot devrait avoir des comportements (et donc des Embeddings) différents pour chacun de ses rôles. Cela est obtenu via l'introduction de trois nouvelles matrices (**Wq**, **Wk** et **Wv**) qui représentent respectivement les rôles de requête, de clé et de valeur. D'ailleurs, c'est précisément ces matrices-là qui seront mises à jour lors de la rétropropagation des erreurs.

Ensuite, nous allons effectuer une multiplication matricielle de Q avec le poids **Wq**, K avec le poids **Wk** et V avec le poids **Wv.** Chaque matrice de poids a la forme de (d_model, d_vector). Le nouveau vecteur de plongement de requête, de clé et de valeur résultant a la forme (**seq_len, d_vector**). Les paramètres de poids seront initialisés de manière aléatoire par le modèle et seront ensuite mis à jour lorsque le modèle commencera à s'entraîner. Pourquoi avons-nous besoin de la multiplication de la matrice pondérale **Wq, Wk, Wv** en premier lieu ? car ce sont des paramètres apprenables qui sont nécessaires pour que les vecteurs d'intégration de requête, de clé et de valeur donnent une meilleure représentation.

Ces vecteurs sont des représentations multidimensionnelles capturant divers aspects de la signification et du rôle syntaxique du mot.

Les dimensions des plongements de mots (par exemple, 512 dimensions) sont appelées « caractéristiques ».

Chaque caractéristique correspond à un aspect spécifique de la représentation du mot. Il peut s'agir de rôles syntaxiques, de significations sémantiques, d'utilisation contextuelle, etc.



Chaque vecteur d'intégration de requête Q effectuera l'opération de produit scalaire avec la transposition du vecteur d'intégration clé K de lui-même. Ce produit scalaire donne un score d'attention. Le score d'attention indique à quel point le jeton donné est similaire à tous les autres jetons de la séquence d'entrée donnée. Plus le score est élevé, plus la similitude est grande.

L'idée ici est de calculer **le produit scalaire entre chaque paire de vecteurs de requête et de clé**. Le calcul du produit scalaire nous donne une estimation du **score d'appariement entre chaque paire** « clé-requête », en utilisant l'idée de similarité cosinusoïdale entre les deux vecteurs. C'est la partie « produit scalaire » de l'attention sur le produit scalaire. le produit scalaire entre chaque paire de vecteurs de requête et de clé est chargé d'extraire la pertinence de chaque élément de la séquence d'entrée en calculant les scores d'attention et la fonction **Softmax le complète en attribuant une probabilité aux scores d'attention**.

Cosinus-similarité

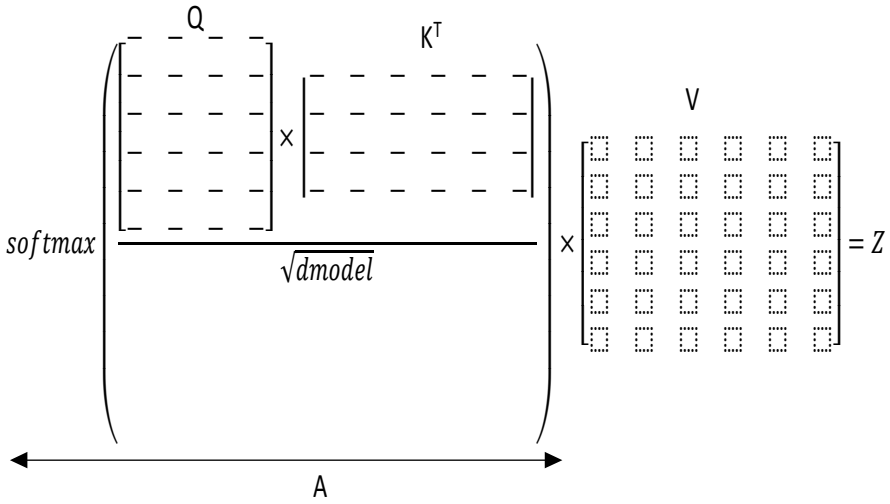
La similitude cosinus est le cosinus de l'angle entre les vecteurs ; c'est-à-dire qu'il s'agit du produit scalaire des vecteurs divisé par le produit de leurs longueurs. Il mesure approximativement si deux vecteurs pointent dans la même direction, ce qui implique que

les deux vecteurs sont similaires.

Rappelez-vous cos(0°) = 1, cos(90°) = 0 , cos(180°) =-1

- Si le produit scalaire entre les deux vecteurs est d'environ 1, cela implique que nous regardons un angle presque nul entre les deux vecteurs, ce qui signifie qu'ils sont très proches l'un de l'autre.
- Si le produit scalaire entre les deux vecteurs est d'environ 0, cela implique que nous regardons des vecteurs orthogonaux l'un à l'autre et peu similaires.
- Si le produit scalaire entre les deux vecteurs est d'environ -1, cela implique que nous regardons un angle de presque 180° entre les deux vecteurs, ce qui signifie qu'ils sont opposés.

Le score d'attention sera ensuite divisé par la racine carrée de la dmodel qui est nécessaire pour normaliser la valeur du score dans la matrice. Mais pourquoi faut-il diviser par dmodel pour normaliser, cela pourrait être n'importe quel autre nombre. La raison principale est qu'à mesure que **la dimension vectorielle d'intégration augmente, la variance totale dans la matrice d'attention augmente proportionnellement.** C'est pourquoi diviser par dmodel équilibrera l'augmentation de la variance. Si nous ne divisons pas par dmodel, pour un score d'attention plus élevé.
les scores d'attention A à partir des vecteurs clé K et requête Q

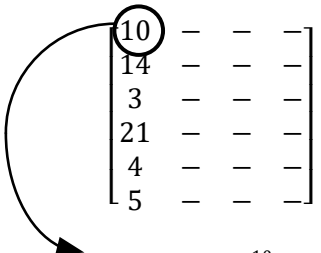


La fonction softmax donnera une valeur de probabilité très élevée et de même pour toute valeur de score d'attention faible, la fonction softmax donnera une valeur de probabilité très faible. Cela finira par amener les modèles à se concentrer uniquement sur l'apprentissage des caractéristiques avec ces valeurs de probabilité et à ignorer les caractéristiques avec des valeurs de probabilité plus faibles, ce qui conduira à la disparition du gradient. Par conséquent, la normalisation de la matrice du score d'attention est absolument nécessaire.

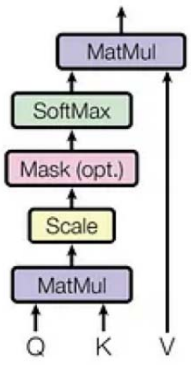
La fonction Softmax

$$S(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{dvector} e^{x_j}}$$

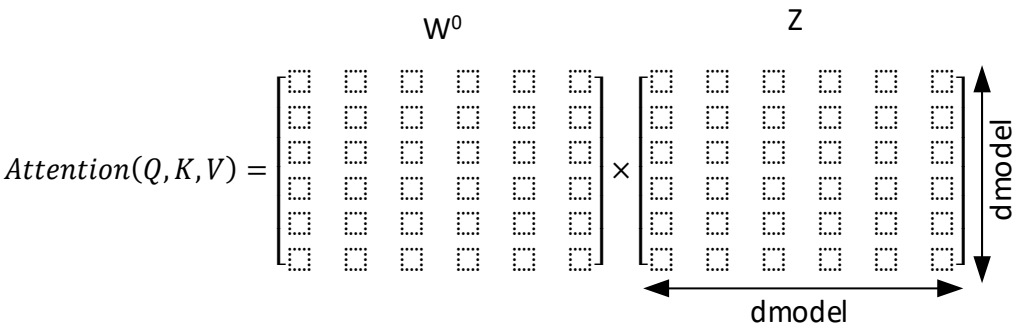
J indice de la colonne de la matrice
i indice d'une valeur de la matrice



$$Softmax(10) = \frac{e^{10}}{e^{10} + e^{14} + e^{3} + e^{21} + e^{4} + e^{5}}$$



Enfin, nous multiplions les vecteurs de valeur (V) par la matrice de poids de l'attention (A). Ces vecteurs de valeur sont importants car ils contiennent les informations associées à chaque mot de la séquence.
Et le résultat de la multiplication finale par W_0 dans cette étape sont les caractéristiques pondérées de l'attention Z qui sont la solution ultime du mécanisme d'auto-attention.
Ces caractéristiques pondérées en fonction de l'attention contiennent essentiellement une représentation pondérée des caractéristiques attribuant des pondérations plus élevées aux caractéristiques plus pertinentes selon le contexte.

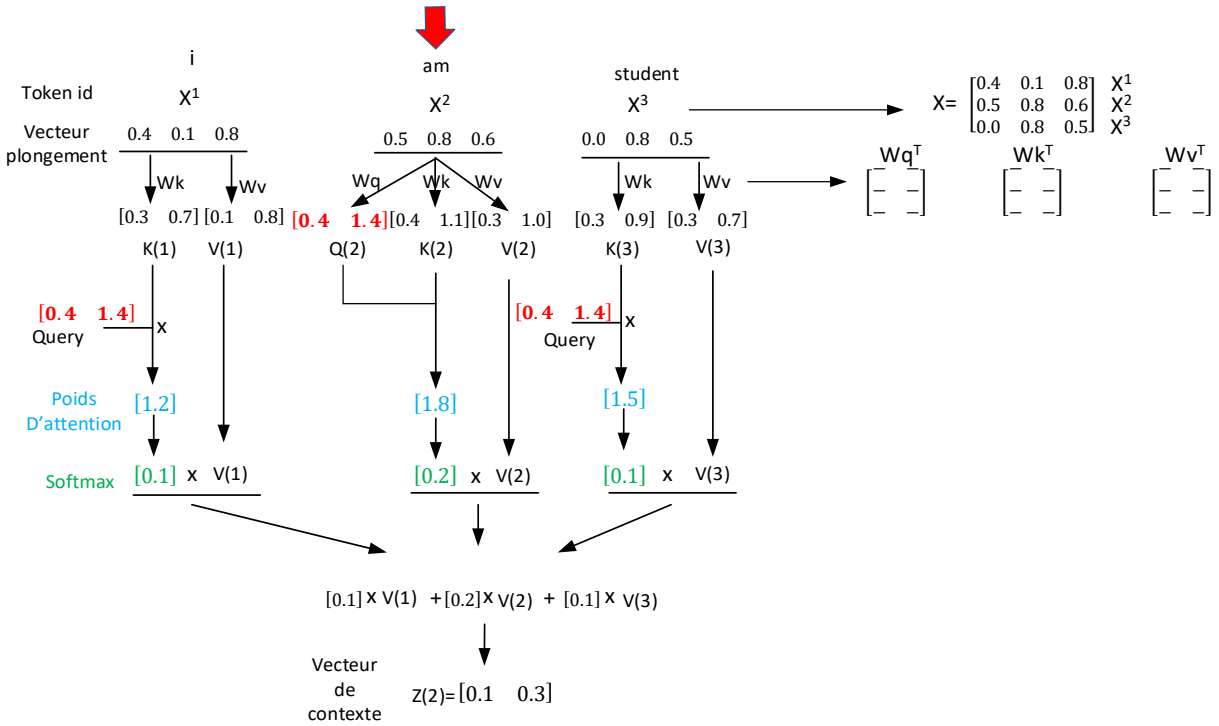


.

Maintenant que ces informations sont disponibles, nous passons à l'étape suivante de l'architecture du transformateur où la couche d'anticipation traite davantage ces informations

Reprenons le modèle de l'attention de soi

Soit la phrase i am student constitué de trois tokens et d'un vecteur de plongement de dimension 3. On souhaite obtenir le vecteur de contexte Z pour la requête position 2 soit le am (flèche rouge)



Pour chaque position de mot dans l'entrée, nous allons prendre l'encodage q et l' encodage K et calculer la similitude.
Nous utilisons ce qu'on appelle un produit scalaire, également appelé similarité cosinusoïdale.
On obtient la matrice d'attention suivante après la fonction softmax
La matrice de poids d'attention contient les scores d'attention pour chaque paire de mot d'entrée

Matrice des poids
D'attention

Query \ key	i	am	student
i	0.15	0.2	0.1
am	0.1	0.2	0.1
student	0.4	0.25	0.3

En pratique, les scores ne seront pas parfaits et le résultat Z sera un peu de tous les encodages mélangés (par exemple 15 % du mot un , 10% pour le mot trois, 20 % pour le mot deux). Mais cela illustre à quel point l'attention de soi est un mélange et un échange.

Matrice des Poids d'attention × Wv = Vecteur de contexte Z

$$\begin{bmatrix} \mathbf{0.15} & \mathbf{0.2} & \mathbf{0.1} \\ \mathbf{0.1} & \mathbf{0.2} & \mathbf{0.1} \\ \mathbf{0.4} & \mathbf{0.25} & \mathbf{0.3} \end{bmatrix} \times \begin{bmatrix} \mathbf{0.1} & \mathbf{0.8} & \mathbf{0.3} \\ \mathbf{0.3} & \mathbf{1} & \mathbf{0.1} \\ \mathbf{0.3} & \mathbf{0.2} & \mathbf{0.15} \end{bmatrix} \begin{matrix} \text{mot1} \\ \text{mot2} \\ \text{mot3} \end{matrix} = \begin{bmatrix} \mathbf{0,105} & \mathbf{0.34} & \mathbf{0.08} \\ \mathbf{0.1} & \mathbf{0.3} & \mathbf{0.065} \\ \mathbf{0.205} & \mathbf{0,63} & \mathbf{0.19} \end{bmatrix}$$

Un transformateur apprend quels mots d'une séquence d'entrée sont liés, puis crée un nouvel encodage pour chaque position de la séquence d'entrée qui est une fusion de tous les mots liés.

le modèle de l'attention de soi avec masque

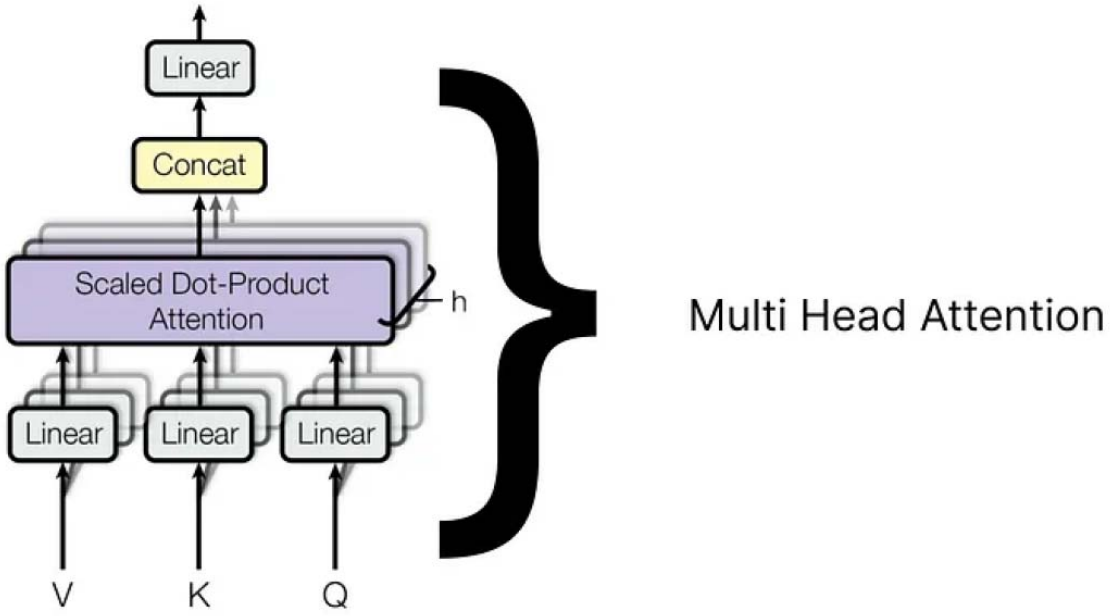
Dans cette section, nous modifions le mécanisme standard d'auto-attention pour créer un mécanisme d'attention causale, essentiel pour développer un LLM . L'attention causale, également connue sous le nom d'attention masquée, est une forme spécialisée de attention personnelle. Il restreint un modèle à ne prendre en compte que les entrées précédentes et actuelles dans une séquence lors du traitement d'un jeton donné. Ceci contraste avec le mécanisme d'auto-attention standard, qui permet d'accéder à l'intégralité de l'entrée séquence à la fois. Par conséquent, lors du calcul des scores d'attention, l'attention causale Le mécanisme garantit que le modèle ne prend en compte que les jetons qui se produisent à ou avant le jeton actuel dans la séquence.
Dans l'attention causale, nous masquons les poids d'attention au-dessus de la diagonale de telle sorte que pour une entrée donnée, le LLM ne peut pas accéder aux futurs jetons lors du calcul des vecteurs de contexte à l'aide de la matrice des poids d'attention. Par exemple, pour le mot « am » en deuxième ligne, on ne garde que le poids d'attention pour les mots avant ("i") et dans la position actuelle ("am").

Matrice des poids
D'attention

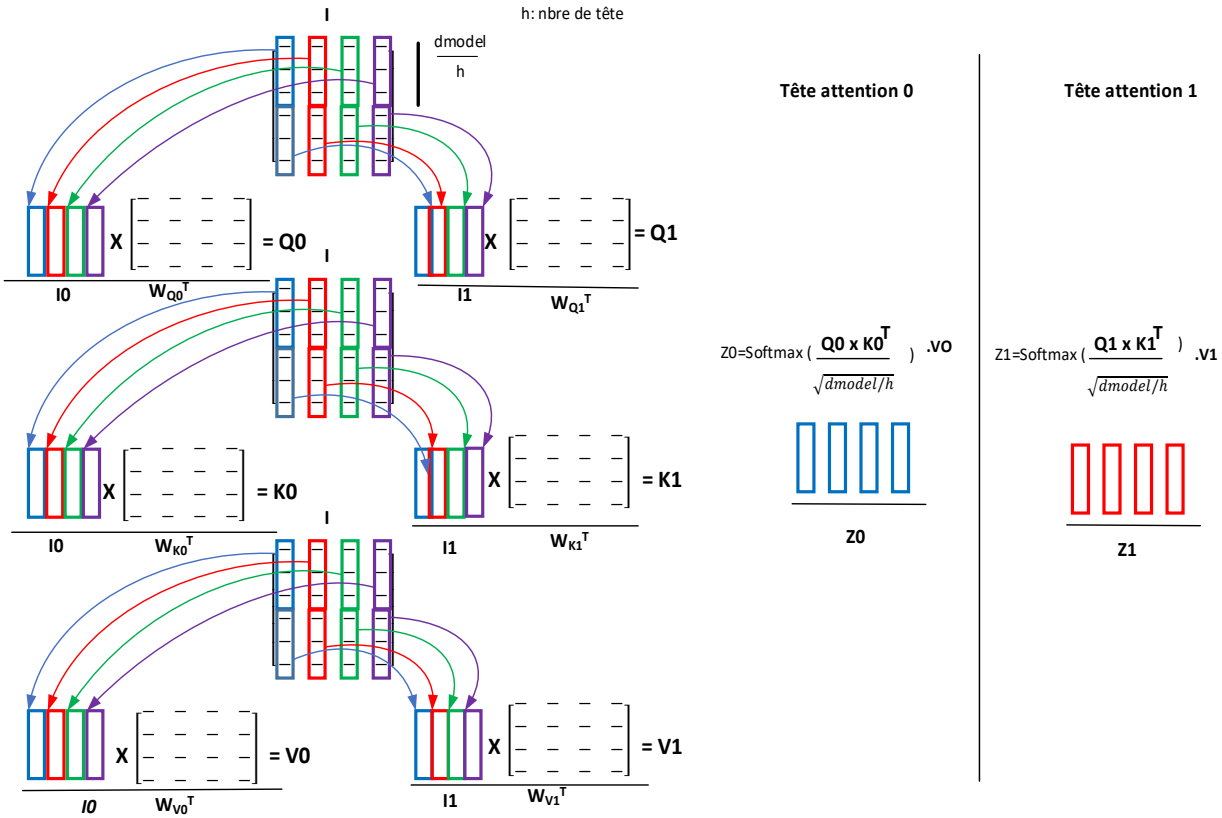
$$\begin{bmatrix} 0.15 & _ & _ \\ 0.1 & 0.2 & _ \\ 0.4 & 0.25 & 0.3 \end{bmatrix}$$

2.4 l'attention aux multiples têtes

(l'attention aux multiples têtes). Ce mécanisme améliore la performance de la couche d'attention de deux façons:
Il étend la capacité du modèle de se **concentrer sur des multiples positions.** Il donne à la couche d'attention **des multiples "sous-espaces de représentation".** Comme on va voir prochainement, quand on utilise la multi-headed attention, on a plusieurs ensembles de matrices de poids Query/Key/Value (le transformer utilise huit têtes d'attention, alors on finit avec huit ensembles pour chaque encodeur/décodeur). Chaque ensemble s'initialise de manière aléatoire. Ensuite, après l'entraînement, chaque ensemble s'utilise pour faire une projection des embeddings à l'entrée (ou des vecteurs des encodeurs/décodeurs plus bas) dans une autre sous-espace de représentation.
Selon le document d'attention, le nombre de têtes serait de h=8. Chaque nouveau vecteur d'intégration de requête, de clé et de valeur sera divisé en 8 unités plus petites de vecteur d'intégration de requête, de clé et de valeur. La nouvelle forme du vecteur d'encastrement est (seq_len, d_vector/h) ou (seq_len, dk). [dk = d_vector/h].

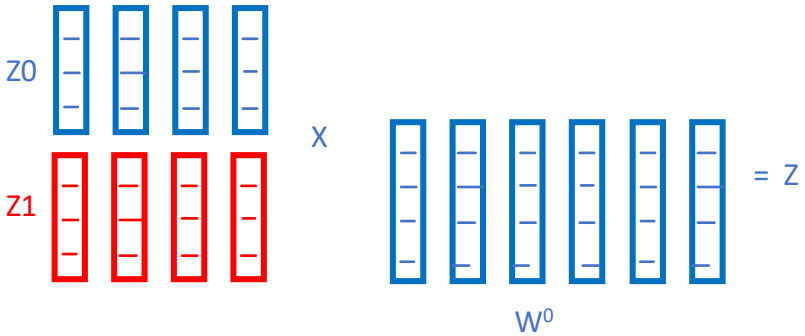


Chaque vecteur d'intégration de requête effectuera l'opération de produit scalaire avec la transposition du vecteur d'intégration clé de lui-même et de tous les autres vecteurs d'intégration de la séquence. Ce produit scalaire donne des scores d'attention. Le score d'attention indique à quel point le jeton donné est similaire à tous les autres jetons de la séquence d'entrée donnée. Plus le score est élevé, plus la similitude est grande. Prenons un exemple simple avec h=2, dmodel=6 soit 2 têtes dimension 6 pour les vecteurs



Le score d'attention sera ensuite divisé par la racine carrée de la dk qui est nécessaire pour normaliser la valeur du score dans la matrice. Mais pourquoi faut-il diviser par dk pour normaliser, cela pourrait être n'importe quel autre nombre. La raison principale est qu'à mesure que la dimension vectorielle d'intégration augmente, la variance totale dans la matrice d'attention augmente proportionnellement. **C'est pourquoi diviser par dk équilibrera l'augmentation de la variance.** Si nous ne divisons pas par dk, pour un score d'attention plus élevé, la fonction softmax donnera une valeur de probabilité très élevée et de même pour toute valeur de score d'attention faible, la fonction softmax donnera une valeur de probabilité très faible. Cela finira par amener les modèles à se concentrer uniquement sur l'apprentissage des caractéristiques avec ces valeurs de probabilité et à ignorer les caractéristiques avec des valeurs de probabilité plus faibles, ce qui conduira à la disparition du gradient. Par conséquent, **la normalisation de la matrice du score d'attention est absolument nécessaire.**

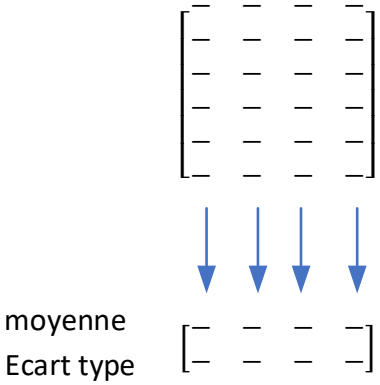
Enfin, toutes les têtes seront concaténées en une seule tête avec une nouvelle forme (seq_len, dmodel). Pour h=2 on a :

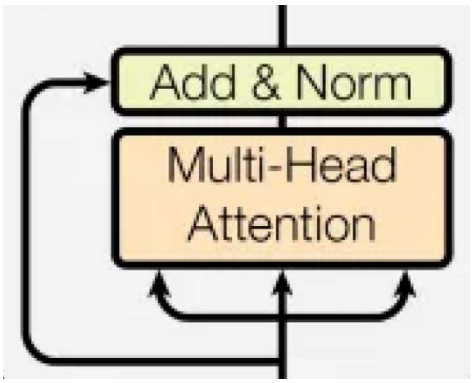


Cette nouvelle tête unique sera multipliée par la matrice de poids de sortie, W^0 (dmodel, dmodel). La sortie finale de l'attention multi-têtes Z **représente la signification contextuelle du mot ainsi que la capacité d'apprendre plusieurs aspects de la phrase d'entrée.** Pour GPT-3 le nombre de tête **est de 96 têtes**

Ajout et normalisation

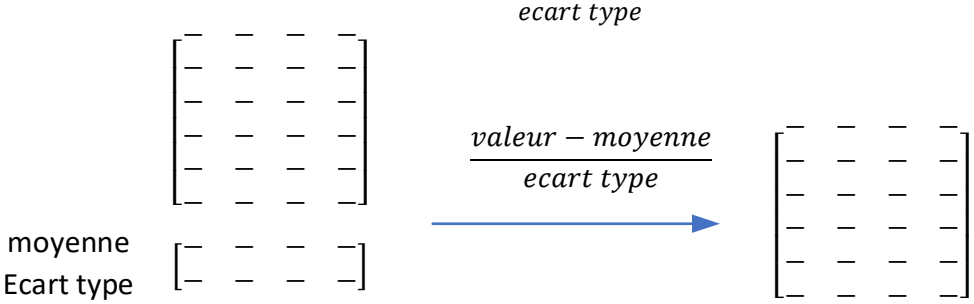
La normalisation de couche est une technique utilisée pour améliorer l'entraînement des réseaux neuronaux profonds en normalisant les entrées à travers les fonctionnalités pour chaque exemple d'entraînement. Il aide à stabiliser le processus d'apprentissage, à améliorer la convergence et à réduire la dépendance à l'égard d'une initialisation minutieuse des paramètres.
Une fois que nous obtenons la matrice résultante de l'attention multi-têtes, nous devons l'ajouter à notre matrice d'origine. Faisons-le d'abord.
Pour normaliser la matrice ci-dessus, nous devons calculer la moyenne et l'écart-type par colonne pour chaque colonne.





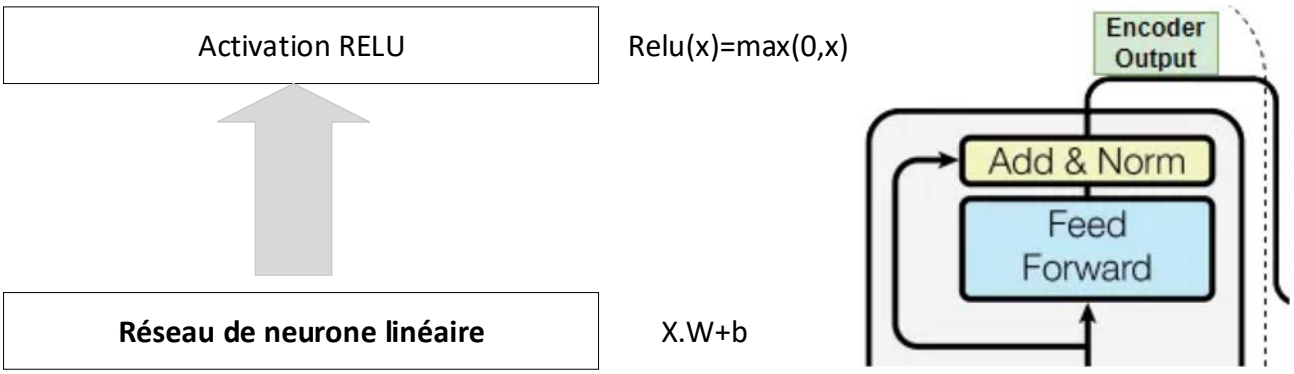
Nous soustrayons chaque valeur de la matrice par la moyenne des lignes correspondante et la divisons par l'écart-type correspondant.

$$\frac{valeur - moyenne}{ecart\ type}$$

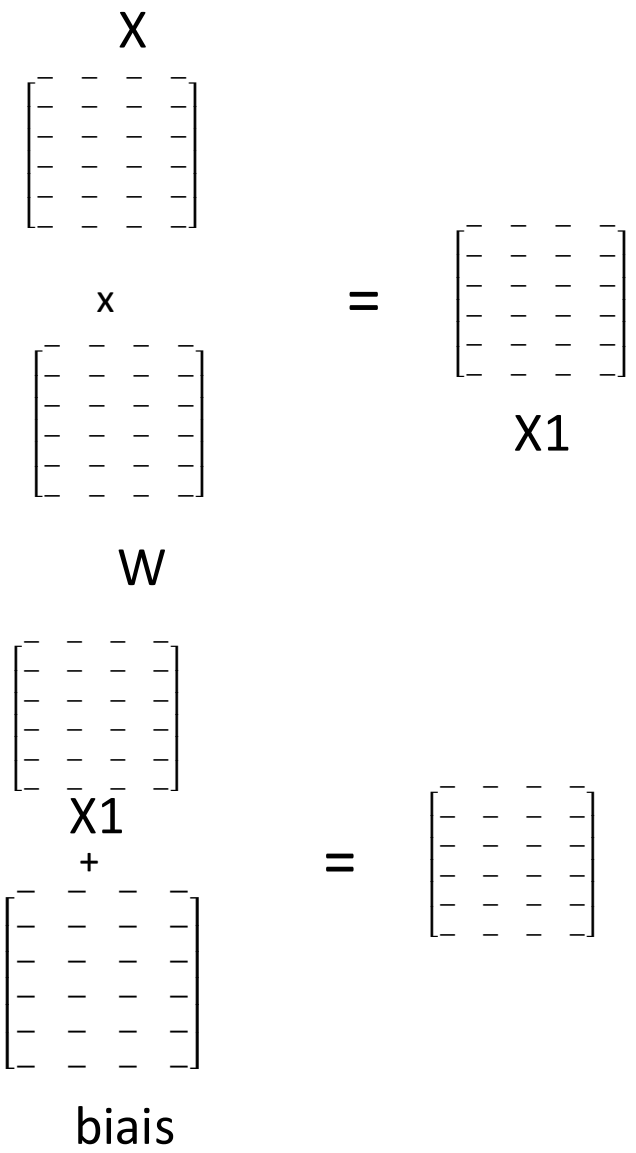


Réseau de rétroaction (Feed forward)

FeedForward est essentiellement une couche entièrement connectée, que le transformateur utilise à la fois dans l'encodeur et le décodeur. Il se compose de deux transformations linéaires avec une activation ReLU entre les deux.
Cela permet d'ajouter de la non-linéarité au modèle, ce qui lui permet d'apprendre des motifs plus complexes.
Après avoir normalisé la matrice, elle sera traitée par le biais d'un réseau feed forward. Nous utiliserons un réseau très basique qui ne contient qu'une seule couche linéaire et une seule couche de fonction d'activation ReLU dans cette exemple. Le plus souvent la couche non linéaire ReLU est prise entre deux réseaux neuronaux. Voici à quoi cela ressemble visuellement :



Tout d'abord, nous devons calculer la couche linéaire en multipliant notre dernière matrice calculée X par un ensemble aléatoire de matrices de poids W, qui sera mis à jour lorsque le transformateur commencera à apprendre, et en ajoutant la matrice résultante à une matrice de biais qui contient également des valeurs aléatoires.

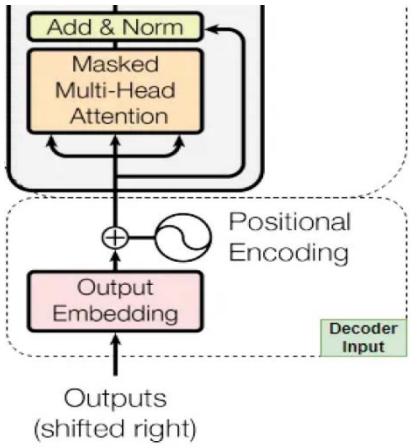


Ajout et normalisation à nouveau

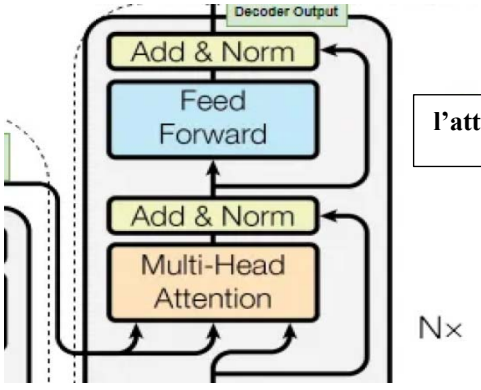
Une fois que nous obtenons la matrice résultante à partir du réseau de rétroaction, nous devons l'ajouter à la matrice obtenue à partir de l'étape d'addition et de norme précédente, puis la normaliser à l'aide de la moyenne des lignes et de l'écart type.

2.5 Décodeur:

Le décodeur construit la sortie finale étape par étape, **en utilisant à la fois les indices de l'encodeur et sa propre chaîne croissante de mots pour comprendre ce qui va suivre**. C'est comme compléter un puzzle avec des indices de l'encodeur, en s'assurant qu'il ne jette pas un coup d'œil aux pièces invisibles.
Le rôle du décodeur est de **générer la séquence de données de sortie étape par étape, une pièce à la fois.** Par exemple, dans la traduction linguistique, il générerait la phrase traduite mot par mot.
Il prend la sortie de l'encodeur et l'utilise pour faire des prédictions pour l'élément suivant de la séquence. Il commence ce processus avec une entrée initiale (comme le début d'une phrase) et construit la sortie séquentiellement.
Il existe deux types d'attention multi-têtes dans le bloc décodeur. Le premier est **l'attention multi-têtes masquées.** Il prend en compte l'entrée du décodeur sous forme de requête, de clé et de valeur et un masque de décodeur (également appelé masque causal). Le masque causal empêche le modèle d'examiner les plongements qui sont en avance dans l'ordre de séquence.
Le décodeur est conscient de ce qu'il a déjà généré, grâce à l'attention multi-têtes masquée, ce qui l'empêche de voir les futurs jetons de sortie lors des prédictions. Ceci est nécessaire car, pendant l'entraînement, le décodeur a accès à la séquence de sortie complète et nous devons l'empêcher de « tricher ». Côté gauche et droit de l'architecture du transformateur :
Les couches d'auto-attention du décodeur fonctionnent d'une manière légèrement différente de celle de l'encodeur :
Dans le décodeur, la couche d'auto-attention n'est autorisée à s'occuper que des positions antérieures dans la séquence de sortie. Cela se fait en masquant les positions futures (en les réglant sur inf) avant l'étape softmax dans le calcul de l'auto-attention.

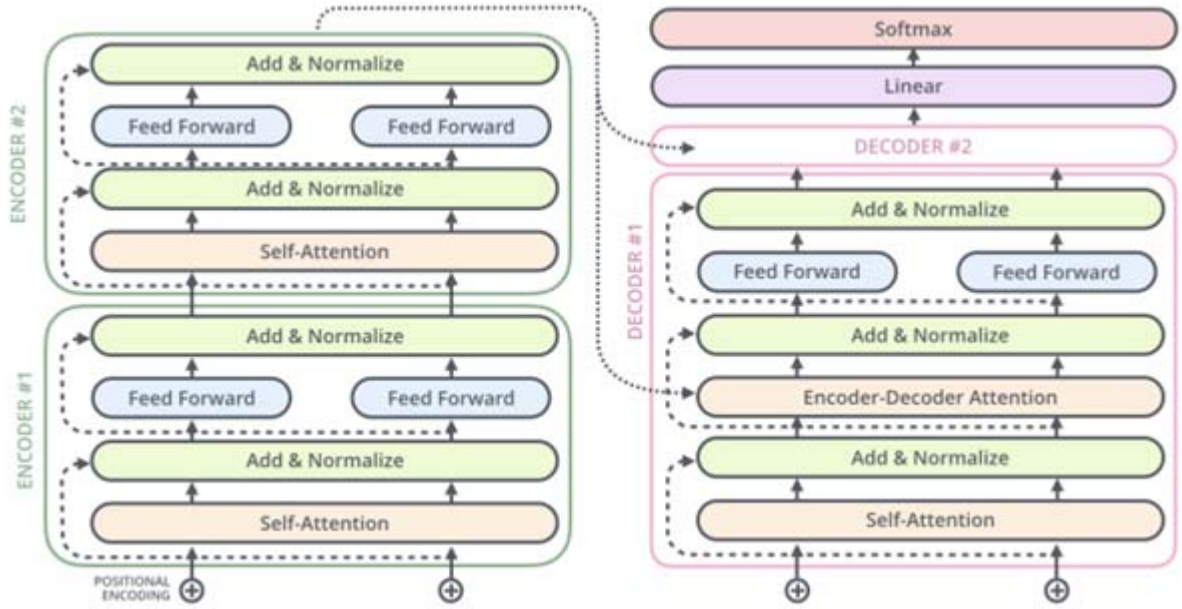


Mais le texte d'entrée prédit doit suivre un retour à la ligne standard de jetons qui permettent au transformateur de savoir où commencer et où finir.
Entrée de l'encodeur → Je suis étudiant
Entrée du décodeur -→ <start> I am a student <end>
La deuxième types d'attention multi-têtes dans le bloc décodeur a également des couches similaires à l'encodeur, mais avec une couche d'attention supplémentaire qui se concentre sur les sorties de l'encodeur. **C'est ce qu'on appelle souvent « l'attention encodeur-décodeur ».**

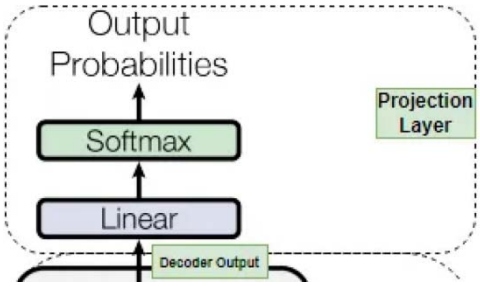


l'attention encodeur-décodeur

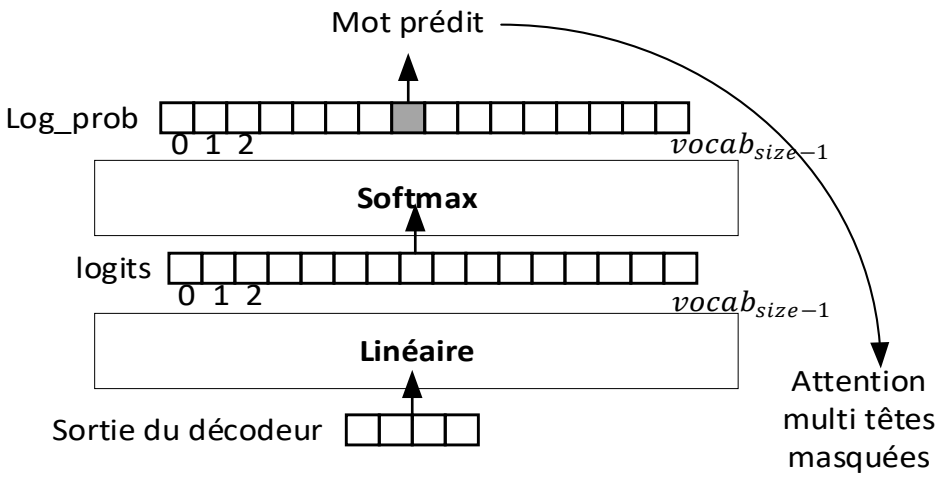
Pour deux encodeurs et deux décodeurs intimement liées ensemble



Lors de l'entraînement, il y a deux entrées dans le décodeur (**l'attention encodeur-décodeur)**. L'une provient de l'encodeur, **où la matrice de sortie de la dernière couche d'addition et de norme sert de clé et de valeur pour la deuxième couche** d'attention multi-têtes dans la partie décodeur. Elle crée sa matrice de Requêtes à partir de la couche inférieure, et prend la matrice Clés et Valeurs de la sortie de la pile d'encodeurs.
Tandis que la matrice de valeur provient du décodeur après la première étape d'ajout et de norme. La deuxième entrée du décodeur est le texte prédit. Si vous vous souvenez, notre entrée à l'encodeur est donc l'entrée au décodeur est le texte prédit
Comme nous le savons déjà, ces plongements sont remplis de valeurs aléatoires, qui seront mises à jour ultérieurement au cours du processus d'apprentissage.
La dernière matrice résultante du décodeur doit être aplatie afin de la faire correspondre à une couche linéaire pour trouver la probabilité prédite de chaque mot unique dans notre jeu de données (corpus).

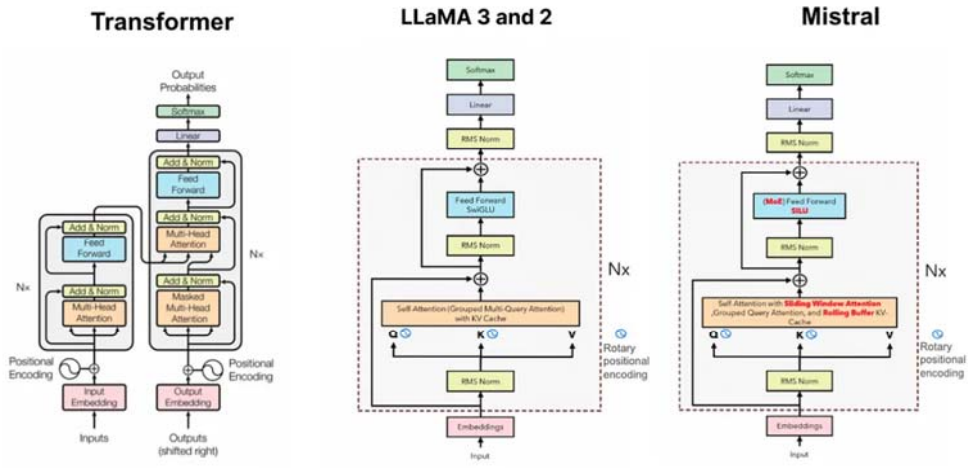


Cette couche aplatie passera à travers une couche linéaire pour calculer les **logits** (scores) de chaque mot unique dans notre jeu de données.



Ce mot prédit , sera traité comme le mot d'entrée du décodeur **l'attention multi-têtes masquées**, et ce processus se poursuit jusqu'à ce que le jeton soit prédit.

Les structures de LLM les plus utilisés



Feature	Plain Transformers	LLaMA 3 and 2	Mistral
Architecture	Encoder-Decoder	Single Layer	Single Layer + Enhancements
Positional Encoding	Standard	Rotary	Rotary + Additional Innovations
Normalization	LayerNorm	RMS Norm	RMS Norm + Enhanced Mechanisms
Attention Mechanism	Self-Attention	Grouped Multi-Query Attention	Sliding Window Attention
Memory Efficiency	N/A	KV Cache	Rolling Buffer KV Cache
Activation in Feedforward	RELU	SwiGLU	SwiGLU + Mixture of Experts

De Rajesh Kavadiki