RÉPUBLIQUE TOGOLAISE
Ministère de l'Économie Numérique et de la Transformation Digitale
Togo Data Lab

Masterclass organisée par :
Togo Data Lab
Fondements Mathématiques des Transformers et des LLMs

UNIVERSITE DE KARA

Module 2 : Transformer : architecture et principes mathématiques

Présentée par : Tiebekabe Pagdame
Enseignant-chercheur - Université de Kara

Dates : 15-16 juillet 2025

RÉPUBLIQUE TOGOLAISE
Ministère de l'Économie Numérique et de la Transformation Digitale
Togo Data Lab

RÉPUBLIQUE TOGOLAISE
Ministère de l'Économie Numérique et de la Transformation Digitale

atd AGENCE TOGO DIGITAL

Bienvenue à la Masterclass

Objectifs : comprendre comment fonctionne un modèle Transformer avec ses fondements mathématiques.

- Problème du traitement séquentiel et limites des RNN/LSTM
- Mécanisme d'attention : attention scalaire, multi-tête
- Encodage positionnel (sinusoidal encoding)
- Architecture générale du Transformer (encoder/décodeur, normalisation, résidus)
- Complexité temporelle et spatiale
- Base mathématique du self-attention : matrices de requêtes (Q), clés (K) et valeurs (V)

Public cible

- Étudiants en Mathématiques/Informatique et Science des Données
- Étudiants à la Faculté des Sciences et de la Santé
- Chercheurs en NLP
- Professionnels du secteur

Sommaire

Traitement séquentiel des données

- Les données textuelles (phrases, documents) sont naturellement **ordonnées** : le sens dépend de l'ordre des mots.
- Exemples :
  - "Le patient souffre de fièvre" $\neq$ "Fièvre souffre le patient".
- **Objectif** du traitement séquentiel : modéliser les **dépendances contextuelles** entre les éléments d'une séquence.
- Cela implique de tenir compte :
  - des relations à **court terme** (mot précédent),
  - mais aussi à **long terme** (information du début de la phrase).
- **Défi principal** : comment capturer efficacement les relations entre mots **éloignés** dans la séquence ?

Approches traditionnelles :

- **RNN** : Réseaux de neurones récurrents (Recurrent Neural Networks)
- **LSTM** : Mémoire à long et court terme (Long Short-Term Memory)

Traitement séquentiel des données

- Les données textuelles (phrases, documents) sont naturellement **ordonnées** : le sens dépend de l'ordre des mots.
- Exemples :
  - "Le patient souffre de fièvre" $\neq$ "Fièvre souffre le patient".
- **Objectif** du traitement séquentiel : modéliser les **dépendances contextuelles** entre les éléments d'une séquence.
- Cela implique de tenir compte :
  - des relations à **court terme** (mot précédent),
  - mais aussi à **long terme** (information du début de la phrase).
- **Défi principal** : comment capturer efficacement les relations entre mots **éloignés** dans la séquence ?

Approches traditionnelles :

- **RNN** : Réseaux de neurones récurrents (Recurrent Neural Networks)
- **LSTM** : Mémoire à long et court terme (Long Short-Term Memory)

RNN : Réseaux Neuronaux Récurrents

Principe de fonctionnement

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b)$$

- À chaque pas de temps t :
  - x_t est le vecteur d'entrée (mot encodé),
  - h_{t-1} est l'état caché précédent,
  - h_t est l'état caché courant,
  - f est une fonction d'activation non-linéaire (ex. $\tanh$, ReLU).
- Le même réseau (mêmes paramètres) est **réutilisé à chaque étape** : c'est ce qu'on appelle le **partage des poids**.
- La sortie à l'étape t dépend de l'entrée actuelle **et** de l'historique passé à travers h_{t-1}.
- Permet une forme de **mémoire dynamique** de la séquence.

Limites des RNN classiques :

- Traitement strictement séquentiel $\Rightarrow$ **pas de parallélisme** possible lors de l'entraînement.
- Les **gradients** peuvent :
  - **Exploser** : instabilités numériques (nécessite du *clipping*),
  - **S'évanouir** : perte d'information pour les longues dépendances.
- Difficulté à capturer des dépendances à **long terme**.

RNN : Réseaux Neuronaux Récurrents

Principe de fonctionnement

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b)$$

- À chaque pas de temps t :
  - x_t est le vecteur d'entrée (mot encodé),
  - h_{t-1} est l'état caché précédent,
  - h_t est l'état caché courant,
  - f est une fonction d'activation non-linéaire (ex. $\tanh$, ReLU).
- Le même réseau (mêmes paramètres) est **réutilisé à chaque étape** : c'est ce qu'on appelle le **partage des poids**.
- La sortie à l'étape t dépend de l'entrée actuelle **et** de l'historique passé à travers h_{t-1}.
- Permet une forme de **mémoire dynamique** de la séquence.

Limites des RNN classiques :

- Traitement strictement séquentiel $\Rightarrow$ **pas de parallélisme** possible lors de l'entraînement.
- Les **gradients** peuvent :
  - **Exploser** : instabilités numériques (nécessite du *clipping*),
  - **S'évanouir** : perte d'information pour les longues dépendances.
- Difficulté à capturer des dépendances à **long terme**.

LSTM : Mémoire à long terme (Long Short-Term Memory)

- Le LSTM est une variante des RNN introduite pour surmonter le **problème des gradients qui disparaissent**.
- Il introduit une **mémoire cellulaire** c_t contrôlée par des **portes** qui régulent le flux d'information.
- Trois portes principales :
  - **Porte d'oubli** f_t : décide ce qu'on oublie de la mémoire précédente.
  - **Porte d'entrée** i_t : contrôle ce qu'on ajoute à la mémoire.
  - **Porte de sortie** o_t : détermine la sortie finale.

Formulation mathématique complète

$$
\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) && \text{(Porte d'oubli)} \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) && \text{(Porte d'entrée)} \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) && \text{(Porte de sortie)} \\
\tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) && \text{(Mémoire candidate)} \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t && \text{(Nouvel état de mémoire)} \\
h_t &= o_t \odot \tanh(c_t) && \text{(Nouvel état caché)}
\end{aligned}
$$

Remarques :

- σ : fonction sigmoïde, sortie entre 0 et 1 (porte).
- $\tanh$: activation hyperbolique, sortie entre -1 et 1.
- $\odot$: produit élément par élément (Hadamard).

LSTM : Mémoire à long terme (Long Short-Term Memory)

- Le LSTM est une variante des RNN introduite pour surmonter le **problème des gradients qui disparaissent**.
- Il introduit une **mémoire cellulaire** c_t contrôlée par des **portes** qui régulent le flux d'information.
- Trois portes principales :
  - **Porte d'oubli** f_t : décide ce qu'on oublie de la mémoire précédente.
  - **Porte d'entrée** i_t : contrôle ce qu'on ajoute à la mémoire.
  - **Porte de sortie** o_t : détermine la sortie finale.

Formulation mathématique complète

$$
\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) && \text{(Porte d'oubli)} \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) && \text{(Porte d'entrée)} \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) && \text{(Porte de sortie)} \\
\tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) && \text{(Mémoire candidate)} \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t && \text{(Nouvel état de mémoire)} \\
h_t &= o_t \odot \tanh(c_t) && \text{(Nouvel état caché)}
\end{aligned}
$$

Remarques :

- σ : fonction sigmoïde, sortie entre 0 et 1 (porte).
- $\tanh$: activation hyperbolique, sortie entre -1 et 1.
- $\odot$: produit élément par élément (Hadamard).

Limites des RNN et LSTM

- **1. Dépendances longues :** même les LSTM ont des difficultés à apprendre des relations entre mots très éloignés (ex. sujet/verbe à longue distance).
- **2. Calcul séquentiel :** les états h_t dépendent de h_{t-1}, ce qui empêche un entraînement parallèle $\Rightarrow$ *bottleneck* en vitesse.
- **3. Problème de gradient :**
  - Les LSTM atténuent mais ne résolvent pas complètement le **vanishing gradient problem**.
  - Cela rend l'optimisation difficile sur de très longues séquences.
- **4. Coût computationnel élevé :**
  - Chaque étape nécessite plusieurs opérations de multiplication matricielle.
  - Temps d'apprentissage et inférence augmentent fortement avec la taille des séquences.

Conséquence : Malgré leurs avancées, les LSTM restent limités pour des tâches comme :

- Résumé de documents longs.
- Traduction de paragraphes complexes.
- Compréhension du contexte global d'un texte médical.

Ces limites ont motivé le développement de nouvelles architectures comme les Transformers.

Limites des RNN et LSTM

- **1. Dépendances longues :** même les LSTM ont des difficultés à apprendre des relations entre mots très éloignés (ex. sujet/verbe à longue distance).
- **2. Calcul séquentiel :** les états h_t dépendent de h_{t-1}, ce qui empêche un entraînement parallèle $\Rightarrow$ *bottleneck* en vitesse.
- **3. Problème de gradient :**
  - Les LSTM atténuent mais ne résolvent pas complètement le **vanishing gradient problem**.
  - Cela rend l'optimisation difficile sur de très longues séquences.
- **4. Coût computationnel élevé :**
  - Chaque étape nécessite plusieurs opérations de multiplication matricielle.
  - Temps d'apprentissage et inférence augmentent fortement avec la taille des séquences.

Conséquence : Malgré leurs avancées, les LSTM restent limités pour des tâches comme :

- Résumé de documents longs.
- Traduction de paragraphes complexes.
- Compréhension du contexte global d'un texte médical.

Ces limites ont motivé le développement de nouvelles architectures comme les Transformers.

Illustration du problème de dépendance longue

La dépendance entre x_1 et x_n est difficilement apprise.

Pourquoi aller au-delà des RNN/LSTM ?

- Les modèles RNN et LSTM ont marqué une avancée majeure pour le traitement des données séquentielles :
  - Gestion implicite du temps via l'état caché h_t.
  - Capacité à mémoriser des motifs fréquents dans les séquences.
- Toutefois, ces architectures présentent des **limites structurelles majeures** :
  1. **Faible parallélisation :** la nature séquentielle impose que le calcul de h_t dépende de h_{t-1}, ce qui empêche tout parallélisme :
  $$h_t = f(h_{t-1}, x_t)$$
  2. **Dépendances longues difficiles à modéliser :**
     - Même les LSTM peinent à capturer des corrélations sémantiques à longue distance.
     - Le gradient se propage difficilement à travers des centaines d'étapes.
  3. **Temps d'entraînement long :** dû à la structure récursive et au manque d'optimisation par blocs.
- **Solution émergente :** l'architecture **Transformer** (Vaswani et al., 2017) :
  - **Entièrement parallèle**, sans récursivité.
  - Basée sur le **mécanisme d'attention** qui permet à chaque mot de *considérer tous les autres mots* dans la séquence.
  - Surpasse les LSTM dans presque toutes les tâches de NLP (traduction, résumé, Q/R, etc.).

Pourquoi aller au-delà des RNN/LSTM ?

- Les modèles RNN et LSTM ont marqué une avancée majeure pour le traitement des données séquentielles :
  - Gestion implicite du temps via l'état caché h_t.
  - Capacité à mémoriser des motifs fréquents dans les séquences.
- Toutefois, ces architectures présentent des **limites structurelles majeures** :
  1. **Faible parallélisation :** la nature séquentielle impose que le calcul de h_t dépende de h_{t-1}, ce qui empêche tout parallélisme :
  $$h_t = f(h_{t-1}, x_t)$$
  2. **Dépendances longues difficiles à modéliser :**
     - Même les LSTM peinent à capturer des corrélations sémantiques à longue distance.
     - Le gradient se propage difficilement à travers des centaines d'étapes.
  3. **Temps d'entraînement long :** dû à la structure récursive et au manque d'optimisation par blocs.
- **Solution émergente :** l'architecture **Transformer** (Vaswani et al., 2017) :
  - **Entièrement parallèle**, sans récursivité.
  - Basée sur le **mécanisme d'attention** qui permet à chaque mot de *considérer tous les autres mots* dans la séquence.
  - Surpasse les LSTM dans presque toutes les tâches de NLP (traduction, résumé, Q/R, etc.).

Pourquoi aller au-delà des RNN/LSTM ?

- Les modèles RNN et LSTM ont marqué une avancée majeure pour le traitement des données séquentielles :
  - Gestion implicite du temps via l'état caché h_t.
  - Capacité à mémoriser des motifs fréquents dans les séquences.
- Toutefois, ces architectures présentent des **limites structurelles majeures** :
  1. **Faible parallélisation :** la nature séquentielle impose que le calcul de h_t dépende de h_{t-1}, ce qui empêche tout parallélisme :
  $$h_t = f(h_{t-1}, x_t)$$
  2. **Dépendances longues difficiles à modéliser :**
     - Même les LSTM peinent à capturer des corrélations sémantiques à longue distance.
     - Le gradient se propage difficilement à travers des centaines d'étapes.
  3. **Temps d'entraînement long :** dû à la structure récursive et au manque d'optimisation par blocs.
- **Solution émergente :** l'architecture **Transformer** (Vaswani et al., 2017) :
  - **Entièrement parallèle**, sans récursivité.
  - Basée sur le **mécanisme d'attention** qui permet à chaque mot de *considérer tous les autres mots* dans la séquence.
  - Surpasse les LSTM dans presque toutes les tâches de NLP (traduction, résumé, Q/R, etc.).

Pourquoi le mécanisme d'attention ?

- Les RNN/LSTM traitent les séquences de manière **incrémentale** : à chaque pas de temps t, l'information est condensée dans un vecteur h_t.
- **Problème :** cette représentation perd de l'information globale, surtout lorsque les dépendances sont lointaines.

Intuition du mécanisme d'attention :

- À chaque étape, *plutôt que de ne regarder que le mot précédent*, le modèle **pèse l'importance de chaque mot de la séquence entière**.
- Chaque sortie est donc une **combinaison pondérée** de toutes les représentations d'entrée.

Motivation formelle :

- Permettre une dépendance directe entre deux positions (i, j) indépendamment de leur distance :

$$\text{Output}_i = \sum_{j=1}^{n} \alpha_{ij} \cdot x_j$$

- α_{ij} : poids d'attention (score d'importance de x_j pour la position i), calculé via une fonction d'alignement.

Avantages clés :

- Captation explicite des dépendances longues.
- Traitement **massivement parallèle**.
- Meilleure interprétabilité (visualisation des poids α_{ij}).

Pourquoi le mécanisme d'attention ?

- Les RNN/LSTM traitent les séquences de manière **incrémentale** : à chaque pas de temps t, l'information est condensée dans un vecteur h_t.
- **Problème :** cette représentation perd de l'information globale, surtout lorsque les dépendances sont lointaines.

Intuition du mécanisme d'attention :

- À chaque étape, *plutôt que de ne regarder que le mot précédent*, le modèle **pèse l'importance de chaque mot de la séquence entière**.
- Chaque sortie est donc une **combinaison pondérée** de toutes les représentations d'entrée.

Motivation formelle :

- Permettre une dépendance directe entre deux positions (i, j) indépendamment de leur distance :

$$\text{Output}_i = \sum_{j=1}^{n} \alpha_{ij} \cdot x_j$$

- α_{ij} : poids d'attention (score d'importance de x_j pour la position i), calculé via une fonction d'alignement.

Avantages clés :

- Captation explicite des dépendances longues.
- Traitement **massivement parallèle**.
- Meilleure interprétabilité (visualisation des poids α_{ij}).

Pourquoi le mécanisme d'attention ?

- Les RNN/LSTM traitent les séquences de manière **incrémentale** : à chaque pas de temps t, l'information est condensée dans un vecteur h_t.
- **Problème :** cette représentation perd de l'information globale, surtout lorsque les dépendances sont lointaines.

Intuition du mécanisme d'attention :

- À chaque étape, *plutôt que de ne regarder que le mot précédent*, le modèle **pèse l'importance de chaque mot de la séquence entière**.
- Chaque sortie est donc une **combinaison pondérée** de toutes les représentations d'entrée.

Motivation formelle :

- Permettre une dépendance directe entre deux positions (i, j) indépendamment de leur distance :

$$\text{Output}_i = \sum_{j=1}^{n} \alpha_{ij} \cdot x_j$$

- α_{ij} : poids d'attention (score d'importance de x_j pour la position i), calculé via une fonction d'alignement.

Avantages clés :

- Captation explicite des dépendances longues.
- Traitement **massivement parallèle**.
- Meilleure interprétabilité (visualisation des poids α_{ij}).

Pourquoi le mécanisme d'attention ?

- Les RNN/LSTM traitent les séquences de manière **incrémentale** : à chaque pas de temps t, l'information est condensée dans un vecteur h_t.
- **Problème :** cette représentation perd de l'information globale, surtout lorsque les dépendances sont lointaines.

Intuition du mécanisme d'attention :

- À chaque étape, *plutôt que de ne regarder que le mot précédent*, le modèle **pèse l'importance de chaque mot de la séquence entière**.
- Chaque sortie est donc une **combinaison pondérée** de toutes les représentations d'entrée.

Motivation formelle :

- Permettre une dépendance directe entre deux positions (i, j) indépendamment de leur distance :

$$\text{Output}_i = \sum_{j=1}^{n} \alpha_{ij} \cdot x_j$$

- α_{ij} : poids d'attention (score d'importance de x_j pour la position i), calculé via une fonction d'alignement.

Avantages clés :

- Captation explicite des dépendances longues.
- Traitement **massivement parallèle**.
- Meilleure interprétabilité (visualisation des poids α_{ij}).

Pourquoi le mécanisme d'attention ?

- Les RNN/LSTM traitent les séquences de manière **incrémentale** : à chaque pas de temps t, l'information est condensée dans un vecteur h_t.
- **Problème :** cette représentation perd de l'information globale, surtout lorsque les dépendances sont lointaines.

Intuition du mécanisme d'attention :

- À chaque étape, *plutôt que de ne regarder que le mot précédent*, le modèle **pèse l'importance de chaque mot de la séquence entière**.
- Chaque sortie est donc une **combinaison pondérée** de toutes les représentations d'entrée.

Motivation formelle :

- Permettre une dépendance directe entre deux positions (i, j) indépendamment de leur distance :

$$\text{Output}_i = \sum_{j=1}^{n} \alpha_{ij} \cdot x_j$$

- α_{ij} : poids d'attention (score d'importance de x_j pour la position i), calculé via une fonction d'alignement.

Avantages clés :

- Captation explicite des dépendances longues.
- Traitement **massivement parallèle**.
- Meilleure interprétabilité (visualisation des poids α_{ij}).

Clés, requêtes, valeurs (Q, K, V)

- Soit une séquence d'entrée représentée par la matrice $X \in \mathbb{R}^{n \times d_{\text{model}}}$, où :
  - n est la longueur de la séquence (nombre de mots ou tokens),
  - d_{model} est la dimension d'embedding de chaque mot.
- Chaque vecteur d'entrée est projeté linéairement dans trois sous-espaces vectoriels distincts :

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

avec :

  - $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$: matrices de projection apprises pendant l'entraînement.
  - $Q, K, V \in \mathbb{R}^{n \times d_k}$: matrices contenant respectivement les **requêtes** (Q), **clés** (K) et **valeurs** (V).
- Interprétation :
  - Chaque mot génère une requête : *Que suis-je en train de chercher ?*
  - Chaque mot est aussi une clé : *À quoi est-ce que je corresponds ?*
  - Chaque mot porte une valeur : *Quelle est l'information que je véhicule ?*
- Objectif : **calculer une représentation contextuelle de chaque mot** en combinant toutes les valeurs V pondérées par la similarité entre requêtes Q et clés K.

Clés, requêtes, valeurs (Q, K, V)

- Soit une séquence d'entrée représentée par la matrice $X \in \mathbb{R}^{n \times d_{\text{model}}}$, où :
  - n est la longueur de la séquence (nombre de mots ou tokens),
  - d_{model} est la dimension d'embedding de chaque mot.
- Chaque vecteur d'entrée est projeté linéairement dans trois sous-espaces vectoriels distincts :

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

avec :

- $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$: matrices de projection apprises pendant l'entraînement.
- $Q, K, V \in \mathbb{R}^{n \times d_k}$: matrices contenant respectivement les **requêtes** (Q), **clés** (K) et **valeurs** (V).

- Interprétation :
  - Chaque mot génère une requête : *Que suis-je en train de chercher ?*
  - Chaque mot est aussi une clé : *À quoi est-ce que je corresponds ?*
  - Chaque mot porte une valeur : *Quelle est l'information que je véhicule ?*
- Objectif : **calculer une représentation contextuelle de chaque mot** en combinant toutes les valeurs V pondérées par la similarité entre requêtes Q et clés K.

Clés, requêtes, valeurs (Q, K, V)

- Soit une séquence d'entrée représentée par la matrice $X \in \mathbb{R}^{n \times d_{\text{model}}}$, où :
  - n est la longueur de la séquence (nombre de mots ou tokens),
  - d_{model} est la dimension d'embedding de chaque mot.
- Chaque vecteur d'entrée est projeté linéairement dans trois sous-espaces vectoriels distincts :

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

avec :

  - $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$: matrices de projection apprises pendant l'entraînement.
  - $Q, K, V \in \mathbb{R}^{n \times d_k}$: matrices contenant respectivement les **requêtes** (Q), **clés** (K) et **valeurs** (V).
- Interprétation :
  - Chaque mot génère une requête : *Que suis-je en train de chercher ?*
  - Chaque mot est aussi une clé : *À quoi est-ce que je corresponds ?*
  - Chaque mot porte une valeur : *Quelle est l'information que je véhicule ?*
- Objectif : **calculer une représentation contextuelle de chaque mot** en combinant toutes les valeurs V pondérées par la similarité entre requêtes Q et clés K.

Clés, requêtes, valeurs (Q, K, V)

- Soit une séquence d'entrée représentée par la matrice $X \in \mathbb{R}^{n \times d_{\text{model}}}$, où :
  - n est la longueur de la séquence (nombre de mots ou tokens),
  - d_{model} est la dimension d'embedding de chaque mot.
- Chaque vecteur d'entrée est projeté linéairement dans trois sous-espaces vectoriels distincts :

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

avec :

  - $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$: matrices de projection apprises pendant l'entraînement.
  - $Q, K, V \in \mathbb{R}^{n \times d_k}$: matrices contenant respectivement les **requêtes** (Q), **clés** (K) et **valeurs** (V).
- Interprétation :
  - Chaque mot génère une requête : *Que suis-je en train de chercher ?*
  - Chaque mot est aussi une clé : *À quoi est-ce que je corresponds ?*
  - Chaque mot porte une valeur : *Quelle est l'information que je véhicule ?*
- Objectif : **calculer une représentation contextuelle de chaque mot** en combinant toutes les valeurs V pondérées par la similarité entre requêtes Q et clés K.

Attention scalaire (Scaled Dot-Product)

Formule mathématique canonique

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

- Produit $QK^\top \in \mathbb{R}^{n\times n}$:
  - Chaque élément $s_{ij} = \langle q_i, k_j \rangle$ mesure la **similarité** entre le mot i (requête) et le mot j (clé).
  - Il s'agit d'un produit scalaire entre les vecteurs q_i et k_j.
- Division par $\sqrt{d_k}$:
  - Sans ce facteur, les produits scalaires peuvent devenir très grands pour d_k élevé.
  - Cela entraîne des gradients proches de zéro ou de un après le softmax (effet de saturation).
  - La normalisation améliore la stabilité numérique et la convergence.
- Application de softmax ligne par ligne :

$$\alpha_{ij} = \frac{\exp\left(\langle q_i, k_j \rangle / \sqrt{d_k}\right)}{\sum_{j'=1}^{n} \exp\left(\langle q_i, k_{j'} \rangle / \sqrt{d_k}\right)}$$

  - Cela produit des **poids d'attention** α_{ij} pour chaque paire (i,j).
  - Chaque ligne de la matrice obtenue est une distribution de probabilité (somme à 1).
- Multiplication finale par V :

$$\text{Output}_i = \sum_{j=1}^{n} \alpha_{ij} \cdot v_j$$

  - Chaque sortie est une combinaison linéaire des valeurs v_j, pondérée par l'importance contextuelle.
  - Le modèle apprend ainsi à extraire l'information la plus pertinente pour chaque position.

Attention scalaire (Scaled Dot-Product)

Formule mathématique canonique

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

- Produit $QK^\top \in \mathbb{R}^{n\times n}$:
  - Chaque élément $s_{ij} = \langle q_i, k_j \rangle$ mesure la **similarité** entre le mot i (requête) et le mot j (clé).
  - Il s'agit d'un produit scalaire entre les vecteurs q_i et k_j.
- Division par $\sqrt{d_k}$:
  - Sans ce facteur, les produits scalaires peuvent devenir très grands pour d_k élevé.
  - Cela entraîne des gradients proches de zéro ou de un après le softmax (effet de saturation).
  - La normalisation améliore la stabilité numérique et la convergence.
- Application de softmax ligne par ligne :

$$\alpha_{ij} = \frac{\exp\left(\langle q_i, k_j \rangle / \sqrt{d_k}\right)}{\sum_{j'=1}^{n} \exp\left(\langle q_i, k_{j'} \rangle / \sqrt{d_k}\right)}$$

  - Cela produit des **poids d'attention** α_{ij} pour chaque paire (i,j).
  - Chaque ligne de la matrice obtenue est une distribution de probabilité (somme à 1).
- Multiplication finale par V :

$$\text{Output}_i = \sum_{j=1}^{n} \alpha_{ij} \cdot v_j$$

  - Chaque sortie est une combinaison linéaire des valeurs v_j, pondérée par l'importance contextuelle.
  - Le modèle apprend ainsi à extraire l'information la plus pertinente pour chaque position.

Attention scalaire (Scaled Dot-Product)

Formule mathématique canonique

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

- Produit $QK^\top \in \mathbb{R}^{n\times n}$:
  - Chaque élément $s_{ij} = \langle q_i, k_j \rangle$ mesure la **similarité** entre le mot i (requête) et le mot j (clé).
  - Il s'agit d'un produit scalaire entre les vecteurs q_i et k_j.
- Division par $\sqrt{d_k}$:
  - Sans ce facteur, les produits scalaires peuvent devenir très grands pour d_k élevé.
  - Cela entraîne des gradients proches de zéro ou de un après le softmax (effet de saturation).
  - La normalisation améliore la stabilité numérique et la convergence.
- Application de softmax ligne par ligne :

$$\alpha_{ij} = \frac{\exp\left(\langle q_i, k_j \rangle / \sqrt{d_k}\right)}{\sum_{j'=1}^{n} \exp\left(\langle q_i, k_{j'} \rangle / \sqrt{d_k}\right)}$$

  - Cela produit des **poids d'attention** α_{ij} pour chaque paire (i,j).
  - Chaque ligne de la matrice obtenue est une distribution de probabilité (somme à 1).
- Multiplication finale par V :

$$\text{Output}_i = \sum_{j=1}^{n} \alpha_{ij} \cdot v_j$$

  - Chaque sortie est une combinaison linéaire des valeurs v_j, pondérée par l'importance contextuelle.
  - Le modèle apprend ainsi à extraire l'information la plus pertinente pour chaque position.

Attention scalaire (Scaled Dot-Product)

Formule mathématique canonique

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

- Produit $QK^\top \in \mathbb{R}^{n\times n}$:
  - Chaque élément $s_{ij} = \langle q_i, k_j \rangle$ mesure la **similarité** entre le mot i (requête) et le mot j (clé).
  - Il s'agit d'un produit scalaire entre les vecteurs q_i et k_j.
- Division par $\sqrt{d_k}$:
  - Sans ce facteur, les produits scalaires peuvent devenir très grands pour d_k élevé.
  - Cela entraîne des gradients proches de zéro ou de un après le softmax (effet de saturation).
  - La normalisation améliore la stabilité numérique et la convergence.
- Application de softmax ligne par ligne :

$$\alpha_{ij} = \frac{\exp\left(\langle q_i, k_j \rangle / \sqrt{d_k}\right)}{\sum_{j'=1}^{n} \exp\left(\langle q_i, k_{j'} \rangle / \sqrt{d_k}\right)}$$

  - Cela produit des **poids d'attention** α_{ij} pour chaque paire (i,j).
  - Chaque ligne de la matrice obtenue est une distribution de probabilité (somme à 1).
- Multiplication finale par V :

$$\text{Output}_i = \sum_{j=1}^{n} \alpha_{ij} \cdot v_j$$

  - Chaque sortie est une combinaison linéaire des valeurs v_j, pondérée par l'importance contextuelle.
  - Le modèle apprend ainsi à extraire l'information la plus pertinente pour chaque position.

Attention scalaire (Scaled Dot-Product)

Formule mathématique canonique

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

- Produit $QK^\top \in \mathbb{R}^{n\times n}$:
  - Chaque élément $s_{ij} = \langle q_i, k_j \rangle$ mesure la **similarité** entre le mot i (requête) et le mot j (clé).
  - Il s'agit d'un produit scalaire entre les vecteurs q_i et k_j.
- Division par $\sqrt{d_k}$:
  - Sans ce facteur, les produits scalaires peuvent devenir très grands pour d_k élevé.
  - Cela entraîne des gradients proches de zéro ou de un après le softmax (effet de saturation).
  - La normalisation améliore la stabilité numérique et la convergence.
- Application de softmax ligne par ligne :

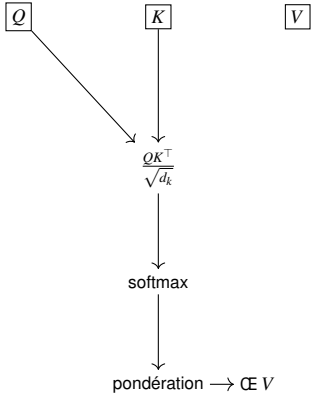
$$\alpha_{ij} = \frac{\exp\left(\langle q_i, k_j \rangle / \sqrt{d_k}\right)}{\sum_{j'=1}^{n} \exp\left(\langle q_i, k_{j'} \rangle / \sqrt{d_k}\right)}$$

  - Cela produit des **poids d'attention** α_{ij} pour chaque paire (i,j).
  - Chaque ligne de la matrice obtenue est une distribution de probabilité (somme à 1).
- Multiplication finale par V :

$$\text{Output}_i = \sum_{j=1}^{n} \alpha_{ij} \cdot v_j$$

  - Chaque sortie est une combinaison linéaire des valeurs v_j, pondérée par l'importance contextuelle.
  - Le modèle apprend ainsi à extraire l'information la plus pertinente pour chaque position.

Illustration : attention scalaire



Attention multi-tête

Motivation : Une seule tête d'attention capte une seule relation de dépendance par position. L'attention multi-tête permet de capturer **simultanément** plusieurs types de relations (syntaxiques, sémantiques, etc.).

Formule générale :

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

avec :

$$\text{head}_i = \text{Attention}(QW_i^Q,\ KW_i^K,\ VW_i^V) \quad \text{pour } i = 1,\ldots,h$$

Détails mathématiques :

- h : nombre de têtes d'attention.
- $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$: matrices de projection spécifiques à la tête i.
- $W^O \in \mathbb{R}^{hd_k \times d_{\text{model}}}$: matrice de projection finale.
- Chaque $\text{head}_i \in \mathbb{R}^{n \times d_k}$ (avec n = longueur de séquence).
- La concaténation donne une matrice $\in \mathbb{R}^{n \times (hd_k)}$.

Interprétation :

- Chaque tête apprend à **attend differently** sur les autres mots : l'attention est spécifique à une projection (W_i^Q, W_i^K, W_i^V).
- Cela permet de capturer plusieurs types de dépendances contextuelles simultanément (par ex. sujet-verbe, coreference, structure logique, etc.).
- Le produit final W^O rassemble toutes les informations captées par les différentes têtes dans une même représentation vectorielle de dimension d_{model}.

Attention multi-tête

Motivation : Une seule tête d'attention capte une seule relation de dépendance par position. L'attention multi-tête permet de capturer **simultanément** plusieurs types de relations (syntaxiques, sémantiques, etc.).

Formule générale :

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

avec :

$$\text{head}_i = \text{Attention}(QW_i^Q,\ KW_i^K,\ VW_i^V) \quad \text{pour } i = 1,\ldots,h$$

Détails mathématiques :

- h : nombre de têtes d'attention.
- $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$: matrices de projection spécifiques à la tête i.
- $W^O \in \mathbb{R}^{hd_k \times d_{\text{model}}}$: matrice de projection finale.
- Chaque $\text{head}_i \in \mathbb{R}^{n \times d_k}$ (avec n = longueur de séquence).
- La concaténation donne une matrice $\in \mathbb{R}^{n \times (hd_k)}$.

Interprétation :

- Chaque tête apprend à **attend differently** sur les autres mots : l'attention est spécifique à une projection (W_i^Q, W_i^K, W_i^V).
- Cela permet de capturer plusieurs types de dépendances contextuelles simultanément (par ex. sujet-verbe, coreference, structure logique, etc.).
- Le produit final W^O rassemble toutes les informations captées par les différentes têtes dans une même représentation vectorielle de dimension d_{model}.

Attention multi-tête

Motivation : Une seule tête d'attention capte une seule relation de dépendance par position. L'attention multi-tête permet de capturer **simultanément** plusieurs types de relations (syntaxiques, sémantiques, etc.).

Formule générale :

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

avec :

$$\text{head}_i = \text{Attention}(QW_i^Q,\ KW_i^K,\ VW_i^V) \quad \text{pour } i = 1,\ldots,h$$

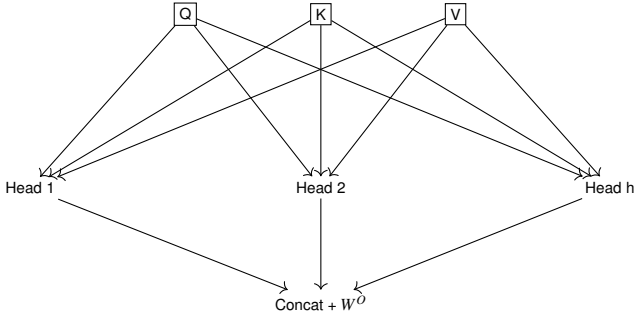
Détails mathématiques :

- h : nombre de têtes d'attention.
- $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$: matrices de projection spécifiques à la tête i.
- $W^O \in \mathbb{R}^{hd_k \times d_{\text{model}}}$: matrice de projection finale.
- Chaque $\text{head}_i \in \mathbb{R}^{n \times d_k}$ (avec n = longueur de séquence).
- La concaténation donne une matrice $\in \mathbb{R}^{n \times (hd_k)}$.

Interprétation :

- Chaque tête apprend à **attend differently** sur les autres mots : l'attention est spécifique à une projection (W_i^Q, W_i^K, W_i^V).
- Cela permet de capturer plusieurs types de dépendances contextuelles simultanément (par ex. sujet-verbe, coreference, structure logique, etc.).
- Le produit final W^O rassemble toutes les informations captées par les différentes têtes dans une même représentation vectorielle de dimension d_{model}.

Illustration : attention multi-tête



Pourquoi l'encodage positionnel ?

Problème fondamental :

- Contrairement aux RNN ou LSTM, le Transformer ne possède **aucune structure séquentielle implicite**.
- Chaque token x_t d'une séquence est traité de manière **indépendante et parallèle** à travers des couches d'attention.
- Par conséquent, **l'information sur l'ordre des tokens est perdue**.

Solution :

- Ajouter une information de position à chaque vecteur d'entrée.
- Pour chaque token d'indice pos, on modifie son embedding par :

$$x_{pos}^{\text{modifié}} = x_{pos} + \text{PE}_{pos}$$

où $\text{PE}_{pos} \in \mathbb{R}^{d_{\text{model}}}$ est un vecteur positionnel.

Deux approches principales :

- **Encodage sinusoïdal** : déterministe, fixe, utilisé dans le papier original Transformer (Vaswani et al., 2017).
- **Encodage appris** : vecteurs positionnels appris durant l'entraînement (similaires à des word embeddings).

Pourquoi l'encodage positionnel ?

Problème fondamental :

- Contrairement aux RNN ou LSTM, le Transformer ne possède **aucune structure séquentielle implicite**.
- Chaque token x_t d'une séquence est traité de manière **indépendante et parallèle** à travers des couches d'attention.
- Par conséquent, **l'information sur l'ordre des tokens est perdue**.

Solution :

- Ajouter une information de position à chaque vecteur d'entrée.
- Pour chaque token d'indice pos, on modifie son embedding par :

$$x_{pos}^{\text{modifié}} = x_{pos} + \text{PE}_{pos}$$

où $\text{PE}_{pos} \in \mathbb{R}^{d_{\text{model}}}$ est un vecteur positionnel.

Deux approches principales :

- **Encodage sinusoïdal** : déterministe, fixe, utilisé dans le papier original Transformer (Vaswani et al., 2017).
- **Encodage appris** : vecteurs positionnels appris durant l'entraînement (similaires à des word embeddings).

Pourquoi l'encodage positionnel ?

Problème fondamental :

- Contrairement aux RNN ou LSTM, le Transformer ne possède **aucune structure séquentielle implicite**.
- Chaque token x_t d'une séquence est traité de manière **indépendante et parallèle** à travers des couches d'attention.
- Par conséquent, **l'information sur l'ordre des tokens est perdue**.

Solution :

- Ajouter une information de position à chaque vecteur d'entrée.
- Pour chaque token d'indice pos, on modifie son embedding par :

$$x_{pos}^{\text{modifié}} = x_{pos} + \text{PE}_{pos}$$

où $\text{PE}_{pos} \in \mathbb{R}^{d_{\text{model}}}$ est un vecteur positionnel.

Deux approches principales :

- **Encodage sinusoïdal** : déterministe, fixe, utilisé dans le papier original Transformer (Vaswani et al., 2017).
- **Encodage appris** : vecteurs positionnels appris durant l'entraînement (similaires à des word embeddings).

Encodage sinusoïdal Définition mathématique

Idée : Définir une fonction continue injectant des informations positionnelles dans les dimensions de l'espace d'encodage.

Définition formelle :

$$\mathrm{PE}_{(pos,\,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \qquad \mathrm{PE}_{(pos,\,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

Notation :

- pos : position du token dans la séquence (entier ≥ 0).
- i : indice de la dimension du vecteur d'encodage ($0 \leq i < d_{\text{model}}/2$).
- d_{model} : dimension des vecteurs de représentation (ex. 512 ou 768).
- $\mathrm{PE}_{pos} \in \mathbb{R}^{d_{\text{model}}}$: vecteur positionnel à injecter à l'embedding.

Pourquoi sinus et cosinus ?

- Ces fonctions sont **périodiques** : permettent une généralisation naturelle des relations de positions.
- Les fréquences des sinusoïdes décroissent exponentiellement encodage **multi-échelle** : certaines dimensions captent des motifs locaux, d'autres des motifs globaux.
- Les décalages relatifs peuvent être appris facilement :

$$\mathrm{PE}_{pos+k} \cdot \mathrm{PE}_{pos} \approx \text{fonction de } k$$

Propriété utile : la distance entre deux positions dépend uniquement de leur différence (translation invariance partielle).

Encodage sinusoïdal Définition mathématique

Idée : Définir une fonction continue injectant des informations positionnelles dans les dimensions de l'espace d'encodage.

Définition formelle :

$$\mathrm{PE}_{(pos,\,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \qquad \mathrm{PE}_{(pos,\,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

Notation :

- pos : position du token dans la séquence (entier ≥ 0).
- i : indice de la dimension du vecteur d'encodage ($0 \leq i < d_{\text{model}}/2$).
- d_{model} : dimension des vecteurs de représentation (ex. 512 ou 768).
- $\mathrm{PE}_{pos} \in \mathbb{R}^{d_{\text{model}}}$: vecteur positionnel à injecter à l'embedding.

Pourquoi sinus et cosinus ?

- Ces fonctions sont **périodiques** : permettent une généralisation naturelle des relations de positions.
- Les fréquences des sinusoïdes décroissent exponentiellement encodage **multi-échelle** : certaines dimensions captent des motifs locaux, d'autres des motifs globaux.
- Les décalages relatifs peuvent être appris facilement :

$$\mathrm{PE}_{pos+k} \cdot \mathrm{PE}_{pos} \approx \text{fonction de } k$$

Propriété utile : la distance entre deux positions dépend uniquement de leur différence (translation invariance partielle).

Encodage sinusoïdal Définition mathématique

Idée : Définir une fonction continue injectant des informations positionnelles dans les dimensions de l'espace d'encodage.

Définition formelle :

$$\mathrm{PE}_{(pos,\,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \qquad \mathrm{PE}_{(pos,\,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

Notation :

- pos : position du token dans la séquence (entier ≥ 0).
- i : indice de la dimension du vecteur d'encodage ($0 \leq i < d_{\text{model}}/2$).
- d_{model} : dimension des vecteurs de représentation (ex. 512 ou 768).
- $\mathrm{PE}_{pos} \in \mathbb{R}^{d_{\text{model}}}$: vecteur positionnel à injecter à l'embedding.

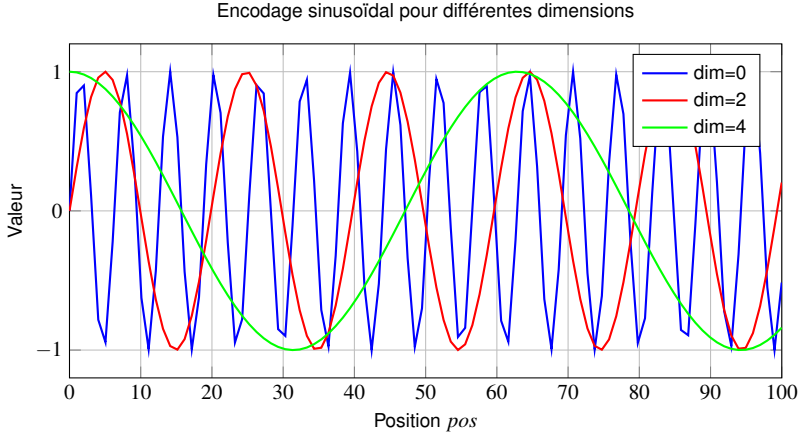
Pourquoi sinus et cosinus ?

- Ces fonctions sont **périodiques** : permettent une généralisation naturelle des relations de positions.
- Les fréquences des sinusoïdes décroissent exponentiellement encodage **multi-échelle** : certaines dimensions captent des motifs locaux, d'autres des motifs globaux.
- Les décalages relatifs peuvent être appris facilement :

$$\mathrm{PE}_{pos+k} \cdot \mathrm{PE}_{pos} \approx \text{fonction de } k$$

Propriété utile : la distance entre deux positions dépend uniquement de leur différence (translation invariance partielle).

Visualisation : positions encodées par sinus/cosinus



Propriétés de l'encodage sinusoïdal

- **Généralisable** à des séquences plus longues que celles vues en entraînement.
- **Différentiable et périodique** : bien adapté aux modèles neuronaux.
- **Codage relatif** possible :

$$\text{PE}_{pos+k} \text{ peut être exprimé comme fonction linéaire de } \text{PE}_{pos}$$

- **Aucune mémoire apprise** : pas de paramètres pas d'overfitting.

Ajout de l'encodage aux embeddings

Objectif : Injecter de l'information sur l'ordre des tokens dans une architecture qui, par construction, est insensible à la position (comme le Transformer).

Formalisme :

- Soit une séquence de n tokens : $[w_1, w_2, \ldots, w_n]$.
- Chaque token w_{pos} (avec $pos \in \{0, 1, \ldots, n-1\}$) est transformé en un vecteur d'embedding $x_{pos} \in \mathbb{R}^{d_{\text{model}}}$ via une matrice d'embedding E :

$$x_{pos} = E(w_{pos})$$

- À chaque position pos, on associe un vecteur positionnel $\text{PE}_{pos} \in \mathbb{R}^{d_{\text{model}}}$.

Ajout de l'information de position :

$$z_{pos} = x_{pos} + \text{PE}_{pos}$$

où :

- $z_{pos} \in \mathbb{R}^{d_{\text{model}}}$ est le vecteur final injecté dans le modèle.
- L'opération est une **somme composante à composante** (élément-wise addition).

Propriétés importantes :

- L'ajout est **structurellement simple** mais **puissant**, car il permet d'enrichir les représentations lexicales avec une information de séquentialité.
- Ce mécanisme est **compatible avec l'attention** :
  - L'attention opère uniquement sur les vecteurs z_{pos}.
  - Les relations de position sont ainsi prises en compte indirectement dans le calcul des poids d'attention.
- Le modèle peut apprendre à interpréter la position selon le contexte, sans que la mécanique d'attention ait à changer.

Ajout de l'encodage aux embeddings

Objectif : Injecter de l'information sur l'ordre des tokens dans une architecture qui, par construction, est insensible à la position (comme le Transformer).

Formalisme :

- Soit une séquence de n tokens : $[w_1, w_2, \ldots, w_n]$.
- Chaque token w_{pos} (avec $pos \in \{0, 1, \ldots, n-1\}$) est transformé en un vecteur d'embedding $x_{pos} \in \mathbb{R}^{d_{\text{model}}}$ via une matrice d'embedding E :

$$x_{pos} = E(w_{pos})$$

- À chaque position pos, on associe un vecteur positionnel $\text{PE}_{pos} \in \mathbb{R}^{d_{\text{model}}}$.

Ajout de l'information de position :

$$z_{pos} = x_{pos} + \text{PE}_{pos}$$

où :

- $z_{pos} \in \mathbb{R}^{d_{\text{model}}}$ est le vecteur final injecté dans le modèle.
- L'opération est une **somme composante à composante** (élément-wise addition).

Propriétés importantes :

- L'ajout est **structurellement simple** mais **puissant**, car il permet d'enrichir les représentations lexicales avec une information de séquentialité.
- Ce mécanisme est **compatible avec l'attention** :
  - L'attention opère uniquement sur les vecteurs z_{pos}.
  - Les relations de position sont ainsi prises en compte indirectement dans le calcul des poids d'attention.
- Le modèle peut apprendre à interpréter la position selon le contexte, sans que la mécanique d'attention ait à changer.

Ajout de l'encodage aux embeddings

Objectif : Injecter de l'information sur l'ordre des tokens dans une architecture qui, par construction, est insensible à la position (comme le Transformer).

Formalisme :

- Soit une séquence de n tokens : $[w_1, w_2, \ldots, w_n]$.
- Chaque token w_{pos} (avec $pos \in \{0, 1, \ldots, n-1\}$) est transformé en un vecteur d'embedding $x_{pos} \in \mathbb{R}^{d_{\text{model}}}$ via une matrice d'embedding E :

$$x_{pos} = E(w_{pos})$$

- À chaque position pos, on associe un vecteur positionnel $\text{PE}_{pos} \in \mathbb{R}^{d_{\text{model}}}$.

Ajout de l'information de position :

$$z_{pos} = x_{pos} + \text{PE}_{pos}$$

où :

- $z_{pos} \in \mathbb{R}^{d_{\text{model}}}$ est le vecteur final injecté dans le modèle.
- L'opération est une **somme composante à composante** (élément-wise addition).

Propriétés importantes :

- L'ajout est **structurellement simple** mais **puissant**, car il permet d'enrichir les représentations lexicales avec une information de séquentialité.
- Ce mécanisme est **compatible avec l'attention** :
  - L'attention opère uniquement sur les vecteurs z_{pos}.
  - Les relations de position sont ainsi prises en compte indirectement dans le calcul des poids d'attention.
- Le modèle peut apprendre à interpréter la position selon le contexte, sans que la mécanique d'attention ait à changer.

Structure globale du Transformer

Le Transformer est un modèle encodeur-décodeur fondé sur l'attention.

Deux grandes composantes

- **Encodeur (Encoder)** :
  - Transforme la séquence source $[x_1, x_2, \ldots, x_n]$ en représentations contextuelles $[z_1, z_2, \ldots, z_n]$.
  - Opère de manière parallèle sur tous les tokens.
- **Décodeur (Decoder)** :
  - Génère la séquence cible $[y_1, y_2, \ldots, y_m]$ de manière autoregressive.
  - À chaque étape t, produit y_t en fonction de $y_1, \ldots, y_{t-1}$ et des représentations de l'encodeur.

Empilement modulaire

- Chaque composant (encodeur ou décodeur) est constitué d'un empilement de N blocs identiques ($N = 6$ dans [Vaswani et al., 2017]).
- Chaque bloc applique des opérations différentiables en séquence :
  - Attention (avec multi-head),
  - Normalisation de couche (LayerNorm),
  - Connexion résiduelle (Residual connection),
  - Réseaux feed-forward (FFN).

Architecture de l'encodeur (Encoder)

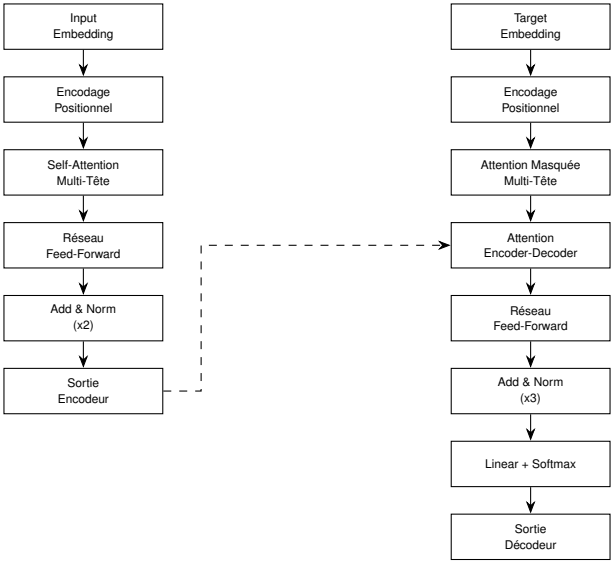
Objectif : Encoder chaque token source avec une représentation tenant compte de tout le contexte source.

Structure d'un bloc d'encodeur (répété N fois) :

1. **Multi-head self-attention** :
   - Calcul de :
   $$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$
   - Ici $Q = K = V = X$, l'entrée du bloc (self-attention).
2. **Add & Layer Normalization** :
   - Ajout de la connexion résiduelle : $X + \text{Attention}(X)$
   - Normalisation sur chaque dimension d'embedding :
   $$\text{LayerNorm}(x) = \frac{x-\mu}{\sigma}\cdot\gamma+\beta$$
   - μ, σ : moyenne et écart-type sur les dimensions.
   - γ, β : paramètres appris.
3. **Feed Forward Neural Network (FFN)** :
   - Deux couches linéaires séparées par une non-linéarité (ReLU ou GELU) :
   $$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$
4. **Add & Layer Normalization (encore)** :
   - Résidu + normalisation : $x + \text{FFN}(x)$

Entrée : $X = [x_1 + \text{PE}_1, \ldots, x_n + \text{PE}_n]$ **Sortie** : Représentation contextuelle $Z = [z_1, \ldots, z_n]$

Architecture complète du Transformer



(Source : Vaswani et al., 2017)

Explication de l'architecture du Transformer

- **Deux parties principales** :
  - **Encodeur** : traite la séquence source pour produire une représentation contextuelle.
  - **Décodeur** : génère la séquence cible mot par mot, en tenant compte du contexte source.
- **Encodeur** :
  - Entrée : Embeddings des tokens + encodage positionnel.
  - Chaque bloc applique :
    - *Multi-head self-attention* : capture les dépendances entre tokens.
    - *Feed-forward network* : transformation non-linéaire identique pour chaque position.
    - *Add & Norm* : normalisation avec connexions résiduelles.
  - Sortie : représentation vectorielle enrichie de chaque token.
- **Décodeur** :
  - Entrée : embeddings des mots déjà générés + encodage positionnel.
  - Blocs contenant :
    - *Masked self-attention* : empêche de voir les mots futurs.
    - *Encoder-decoder attention* : intègre le contexte de la source.
    - *Feed-forward network + Add & Norm.*
  - Sortie : passe par une couche linéaire + softmax pour prédire le mot suivant.
- **Entraînement** : le décodeur prédit chaque mot à partir des précédents, avec supervision.

Connexions résiduelles (Residual Connections)

- Chaque sous-couche est encadrée par une connexion résiduelle :

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

- Avantages :
  - Favorise la propagation du gradient (évite vanishing).
  - Stabilise l'apprentissage profond.
- Introduites par He et al. (ResNet).

Résumé

- Le Transformer repose sur une architecture modulaire (encodeur/décodeur).
- Chaque bloc contient attention, normalisation, résidu, FFN.
- Connexions résiduelles et normalisation sont essentielles pour l'optimisation.

Clé de son efficacité : traitement parallèle + structure hiérarchique.

Pourquoi étudier la complexité ?

- Les modèles Transformer sont performants mais coûteux.
- Comprendre la complexité permet :
  - d'identifier les limites pratiques ;
  - de motiver les améliorations architecturales ;
  - d'évaluer l'adéquation à des cas réels.

Notations utilisées

- n : longueur de la séquence d'entrée
- d : dimension de l'espace des représentations
- h : nombre de têtes d'attention
- b : taille du batch

On s'intéressera à :

- **Temps de calcul (temporel)** : nombre d'opérations à effectuer
- **Mémoire requise (spatial)** : taille mémoire pour stocker les intermédiaires

Complexité de l'attention standard

Produit de matrices clés-valeurs :

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

- $Q, K, V \in \mathbb{R}^{n \times d}$
- Produit QK^T coûte $O(n^2 d)$
- Puis multiplication par V : $O(n^2 d)$

Donc :

$$\text{Complexité temporelle et spatiale} = O(n^2 d)$$

Comparaison avec RNN/LSTM

RNN / LSTM

Complexité temporelle $= O(nd^2)$, séquentielle (pas parallélisable)

Mais mémoire plus faible : $O(nd)$

Transformer

Complexité temporelle $= O(n^2 d)$, hautement parallélisable

Mémoire : $O(n^2)$ pour les matrices d'attention

Conclusion : plus rapide pour traitement parallèle, mais consomme beaucoup de mémoire pour grandes séquences.

Comparaison avec RNN/LSTM

RNN / LSTM

Complexité temporelle $= O(nd^2)$, séquentielle (pas parallélisable)

Mais mémoire plus faible : $O(nd)$

Transformer

Complexité temporelle $= O(n^2 d)$, hautement parallélisable

Mémoire : $O(n^2)$ pour les matrices d'attention

Conclusion : plus rapide pour traitement parallèle, mais consomme beaucoup de mémoire pour grandes séquences.

Comparaison avec RNN/LSTM

RNN / LSTM

$$\text{Complexité temporelle} = O(nd^2), \quad \text{séquentielle (pas parallélisable)}$$

Mais mémoire plus faible : $O(nd)$

Transformer

$$\text{Complexité temporelle} = O(n^2 d), \quad \text{hautement parallélisable}$$

Mémoire : $O(n^2)$ pour les matrices d'attention

Conclusion : plus rapide pour traitement parallèle, mais consomme beaucoup de mémoire pour grandes séquences.

Limitations pratiques

- Les Transformers standards deviennent inefficaces lorsque n est grand (> 2048).
- Les modèles LLM nécessitent souvent du padding et du masking, ce qui augmente encore le coût.
- Nécessité de réduire $O(n^2)$ à $O(n \log n)$ ou $O(n)$.

Problème : le goulot d'étranglement est la self-attention $\Rightarrow$ optimisation cruciale.

Limitations pratiques

- Les Transformers standards deviennent inefficaces lorsque n est grand (> 2048).
- Les modèles LLM nécessitent souvent du padding et du masking, ce qui augmente encore le coût.
- Nécessité de réduire $O(n^2)$ à $O(n \log n)$ ou $O(n)$.

Problème : le goulot d'étranglement est la self-attention $\Rightarrow$ optimisation cruciale.

Approches pour réduire la complexité

- **Linformer** : approximation basse-rang de la matrice d'attention. $\mathcal{O}(n)$
- **Performer** : kernelized attention. $\mathcal{O}(n)$
- **Longformer, BigBird** : attention locale + globales. $\mathcal{O}(n)$
- **Reformer** : hashing + réversibilité. $\mathcal{O}(n \log n)$

Objectif : rendre les Transformers utilisables sur des séquences longues (texte, génomique, vidéo...).

Résumé

- Transformer : complexité quadratique $O(n^2 d)$ en temps et en espace.
- Avantage : parallélisme. Inconvénient : coût mémoire.
- Des variantes modernes réduisent cette complexité tout en gardant la performance.

Intuition du Self-Attention

- Le self-attention permet à chaque élément d'une séquence de **s'auto-contexualiser** :

Quel est le poids de chaque mot dans mon contexte ?

- On calcule une combinaison pondérée de tous les vecteurs de la séquence.
- Ce poids est obtenu par la similarité entre une **requête** q et toutes les **clés** k_i.

Formulation matricielle

Tenseur d'entrée

On considère une séquence d'entrée représentée par une matrice :

$$X \in \mathbb{R}^{n \times d_{\text{model}}}$$

où :

- n est la longueur de la séquence (nombre de tokens),
- d_{model} est la dimension de chaque vecteur de mot (embedding + encodage positionnel).

Chaque ligne $x_i \in \mathbb{R}^{d_{\text{model}}}$ correspond à un token.

Projections linéaires

À partir de X, on calcule les triplets Q, K, V utilisés dans le mécanisme d'attention :

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

avec :

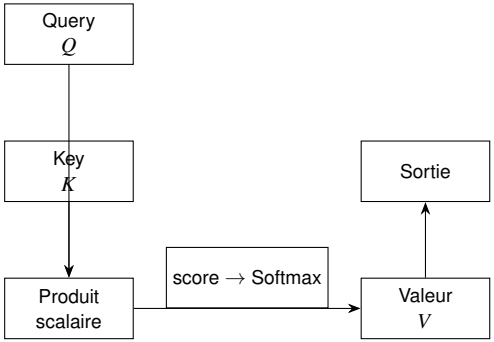
- $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$: matrices de poids apprises,
- $Q, K, V \in \mathbb{R}^{n \times d_k}$: matrices des requêtes, clés et valeurs.

Interprétation

Chaque token x_i est projeté en un triplet (q_i, k_i, v_i) :

- q_i mesure "ce que l'on cherche",
- k_j mesure "ce que chaque token peut offrir",

Schéma d'attention scalaire simple



Score d'attention entre mots

Score de compatibilité

$$\text{score}(q_i, k_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$$

- Produit scalaire pour mesurer la similarité.
- Division par $\sqrt{d_k}$ pour stabiliser les gradients (variance).
- Ce score est ensuite normalisé par softmax.

Formule du Self-Attention

Formule vectorielle

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

- $Q \in \mathbb{R}^{n\times d_k}$, $K \in \mathbb{R}^{n\times d_k}$, $V \in \mathbb{R}^{n\times d_v}$
- $QK^T \in \mathbb{R}^{n\times n}$: scores entre chaque paire de positions.
- Softmax agit ligne par ligne pour pondérer les valeurs V.

Exemple simple de self-attention

Soit une séquence de 3 vecteurs d'entrée x_1, x_2, x_3.

1. On projette chaque x_i en q_i, k_i, v_i
2. On calcule $q_1 \cdot k_j$ pour $j = 1, 2, 3$
3. On applique softmax : $[\alpha_1, \alpha_2, \alpha_3]$
4. On effectue : $z_1 = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$

$$z_i = \sum_{j=1}^{n} \text{softmax}_j(q_i \cdot k_j) \cdot v_j$$

Exemple simple de self-attention

Soit une séquence de 3 vecteurs d'entrée x_1, x_2, x_3.

1. On projette chaque x_i en q_i, k_i, v_i
2. On calcule $q_1 \cdot k_j$ pour $j = 1, 2, 3$
3. On applique softmax : $[\alpha_1, \alpha_2, \alpha_3]$
4. On effectue : $z_1 = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$

$$z_i = \sum_{j=1}^{n} \text{softmax}_j(q_i \cdot k_j) \cdot v_j$$

Remarques importantes

- Les matrices de projection $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ sont des **paramètres entraînables**, c'est-à-dire optimisés par descente de gradient pendant l'apprentissage du modèle.
- Le mécanisme d'attention est **invariant à la permutation des positions** dans la séquence :

$$\text{Attention}(X) = \text{Attention}(PX) \quad \text{pour toute permutation } P$$

  Conséquence : on introduit un **encodage positionnel** pour injecter l'information d'ordre dans la séquence.
- Chaque sortie z_i (ligne i de la sortie de l'attention) est une **combinaison linéaire pondérée** des vecteurs de valeur v_j, où les poids sont donnés par la similarité entre la requête q_i et chaque clé k_j :

$$z_i = \sum_{j=1}^{n} \alpha_{ij} v_j \quad \text{où } \alpha_{ij} = \text{softmax}\left(\frac{q_i \cdot k_j^\top}{\sqrt{d_k}}\right)$$

  Cela permet à chaque position de capturer un **contexte global** de la séquence.