Université Ibn Tofail
Faculté des Sciences

جامعة ابن طفيل
ⵜⴰⵙⴷⴰⵡⵉⵜ ⵉⴱⵏ ⵟⴼⵉⵍ
Ibn Tofaïl University

IA générative & LLMs

Polycopié de Cours

Pr. Tarik HOUICHIME

Année 2025/2026

Chapitre 1 : IA, De l'Analyse à la Création

1 L'IA Traditionnelle : Spécifier "Ce que c'est"

La plupart des modèles d'IA que vous avez étudiés jusqu'à présent (tels que les classifieurs d'images ou les détecteurs de spam) suivent le paradigme **discriminatif**. Dans cette approche, le rôle du modèle est d'apprendre à distinguer, à classer ou à étiqueter des données existantes.

- **Le contrôle est la classification :** Le code apprend les frontières qui séparent différentes catégories de données. L'objectif est de prendre une décision sur une entrée.
- **L'état est une prédiction :** Le programme analyse une entrée (un email, une image) et produit une sortie simple (une étiquette : "spam" / "non-spam", "chat" / "chien").
- **Le modèle apprend à juger :** C'est au modèle de trouver les caractéristiques (les *features*) qui lui permettent de discriminer au mieux les données.

En somme, l'IA discriminative consiste à apprendre à un ordinateur **comment analyser et étiqueter le monde.**

2 L'IA Générative : Décrire "Comment créer"

L'IA générative est une forme d'IA qui adopte une approche radicalement différente. Au lieu de classer des données, elle apprend la structure sous-jacente de ces données pour en **générer de nouvelles instances**, originales et plausibles.

- **Le contrôle est la création :** L'objectif du code est de produire une nouvelle sortie (un texte, une image) qui ressemble aux données sur lesquelles il a été entraîné.
- **L'état est une distribution de probabilité :** Le programme apprend la probabilité que chaque élément (mot, pixel) apparaisse dans un certain contexte. Il génère ensuite en "échantillonnant" dans cette distribution.
- **Le modèle apprend l'essence des choses :** Il ne cherche pas les frontières entre les concepts, mais la structure interne de chaque concept.

En somme, l'IA générative consiste à apprendre à un ordinateur **comment créer des artefacts qui appartiennent à notre monde.**

3 Qu'est-ce qu'un "Modèle" ? L'idée de base

Que nous parlions d'IA discriminative ou générative, le terme **modèle** est central. Un modèle est une représentation mathématique simplifiée d'un processus du monde réel. Son but est de "prédire" une sortie (y) à partir d'une entrée (x). Pour ce faire, il utilise des **paramètres** (souvent appelés **poids**), qui sont des nombres ajustables que le modèle apprend pendant l'apprentissage.

3.1 Analogie : La Régression Linéaire

La régression linéaire est le modèle le plus simple pour illustrer cette idée de **paramètrage**. Imaginons que nous voulions prédire le prix d'une maison (y) en fonction de sa surface (x). La relation peut être modélisée par une simple droite :

$$y = w \cdot x + b$$

Ici :

- x est l'entrée (la surface).
- y est la sortie prédite (le prix).
- w (le poids) et b (le biais) sont les **paramètres** du modèle. Ce sont ces valeurs que le modèle doit "apprendre" à partir des données pour que la droite s'ajuste au mieux aux exemples réels.

Un *Large Language Model* (LLM) est conceptuellement identique, mais à une échelle inimaginable. Au lieu de deux paramètres (w et b), il en possède des milliards. Et au lieu de prédire un prix, son objectif est de prédire... **le mot suivant**.

4 L'Objectif Final d'un LLM : Prédire le Mot Suivant

La tâche fondamentale d'un LLM est, étant donné une séquence de mots, de **prédire le mot le plus probable qui suit**. C'est tout. La magie de la génération de texte, de la traduction ou du résumé émerge de **la répétition de cette simple tâche**.

4.1 Pourquoi était-ce si difficile avant ?

Les modèles précédents (comme les Réseaux de Neurones Récurrents, RNN) luttaient avec un problème majeur : la **mémoire à court terme**.

- **Gestion du contexte limité :** Un RNN lisait les mots séquentiellement. Pour prédire le mot N, il se basait principalement sur le mot N-1. L'influence des mots plus anciens s'estompait très rapidement. Il était donc difficile de maintenir une cohérence sur de longs paragraphes.
- **Le "goulot d'étranglement" de l'information :** Toute l'information pertinente sur le passé devait être compressée dans un "état caché" de taille fixe. C'est comme essayer de résumer un livre entier en une seule phrase avant d'écrire la suivante.

4.2 Pourquoi les LLM sont-ils si efficaces ?

Les LLM modernes sont basés sur une architecture appelée **Transformer**. Son innovation clé est le **mécanisme d'attention**.

- **Accès direct au contexte :** Au lieu de lire séquentiellement, le mécanisme d'attention permet au modèle, pour chaque nouveau mot à générer, de "regarder" tous les mots précédents simultanément.
- **Pondération de l'importance :** Le modèle apprend à donner plus de "poids" ou d'"attention" aux mots du contexte qui sont les plus pertinents pour la prédiction actuelle. Si le texte parle de "Paris" et doit prédire le mot suivant après "la Tour", il portera une grande attention au mot "Paris", même s'il est loin dans la phrase.

Ce mécanisme a brisé le goulot d'étranglement de la mémoire et a permis aux modèles de gérer des contextes beaucoup plus longs et complexes, menant à la révolution de l'IA générative.

5 L'Architecture Simplifiée d'un LLM

Un LLM basé sur l'architecture Transformer se compose de plusieurs couches successives, dont les principales sont :

1. **Couche d'Embedding :** Les mots ne sont pas traités comme du texte, mais sont transformés en vecteurs de nombres (des "embeddings"). Ces vecteurs représentent la signification sémantique des mots.
2. **Encodage Positionnel :** Le modèle d'attention, par nature, ne comprend pas l'ordre des mots. On ajoute donc une information de position à chaque embedding pour que le modèle sache quel mot vient avant quel autre.
3. **Blocs Transformer (la pile de Décodeurs) :** C'est le cœur du modèle. Un LLM est une succession de dizaines de ces blocs. Chaque bloc est composé de deux sous-couches principales :
   - **Le Mécanisme d'Attention Multi-têtes :** C'est la couche qui examine l'importance des mots précédents pour prédire le suivant. "Multi-têtes" signifie qu'il le fait sous plusieurs angles différents en parallèle, apprenant différents types de relations entre les mots.
   - **Un Réseau de Neurones Feed-Forward :** Après la couche d'attention, cette couche traite l'information pour l'enrichir et la préparer pour le bloc suivant.
4. **Couche de Sortie :** Après être passée à travers tous les blocs, la représentation vectorielle finale est envoyée à une couche de sortie. Cette couche calcule une probabilité pour chaque mot possible dans le vocabulaire du modèle. Le mot avec la plus haute probabilité est (généralement) choisi comme le mot suivant.

Ce processus est ensuite répété : le nouveau mot généré est ajouté à la séquence d'entrée, et le modèle prédit le mot suivant, et ainsi de suite.

Le schéma 1 représente l'architecture Transformer “encodeur-décodeur”. À gauche, l'encodeur reçoit les entrées sous forme d'« input embeddings » additionnés à un encodage positionnel. Chaque bloc (répété N fois) applique une attention multi-têtes, puis une connexion résiduelle avec normalisation

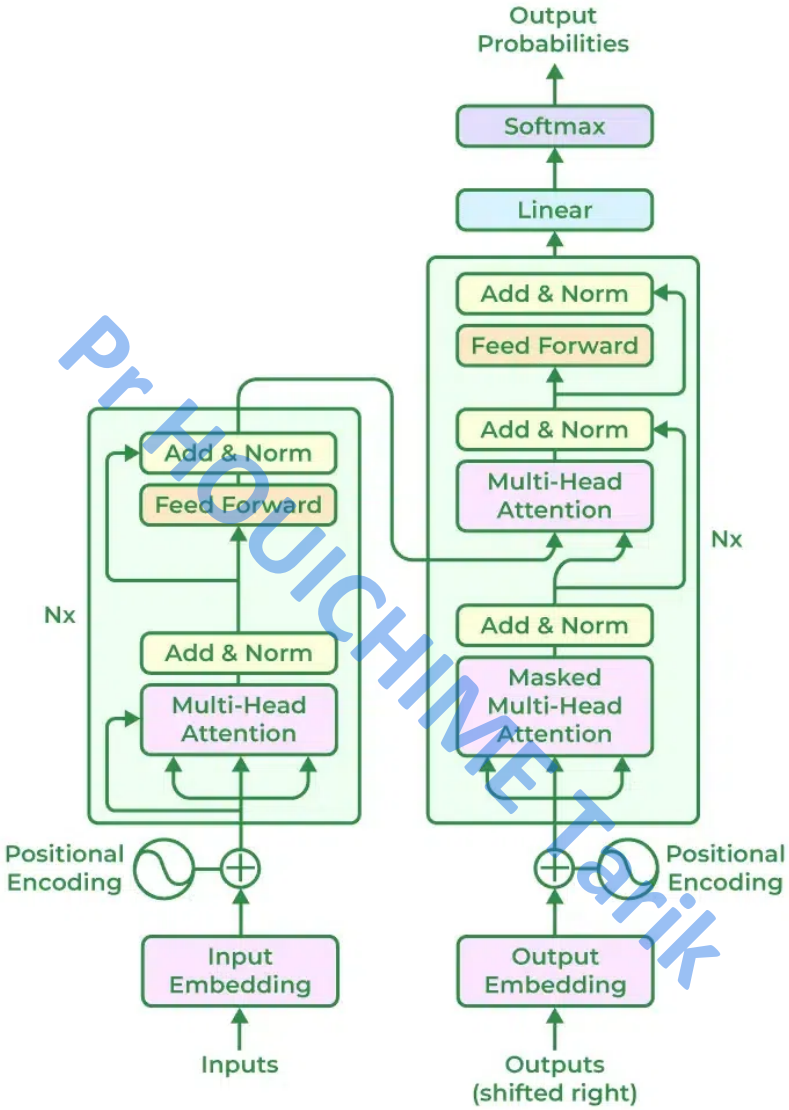

FIGURE 1 – Schéma de l'architecture Transformer encodeur–décodeur : à gauche, l'encodeur applique N blocs « Multi-Head Attention → Add & Norm → Feed Forward → Add & Norm » sur des embeddings d'entrée enrichis d'un encodage positionnel ; à droite, le décodeur traite des sorties décalées via une attention multi-têtes masquée, une attention croisée sur l'encodeur puis un réseau feed-forward, avant projection linéaire et softmax pour obtenir les probabilités de sortie.

(Add & Norm), suivie d'un réseau feed-forward et d'une seconde connexion résiduelle/normalisation. Le résultat de la pile d'encodeurs condense le contexte de la séquence d'entrée.

À droite, le décodeur prend les « output embeddings » décalés d'un pas vers la droite, additionnés à leur encodage positionnel. Chaque bloc (répété N fois) commence par une attention multi-têtes masquée pour empêcher de « voir » les mots futurs, puis réalise une attention multi-têtes croisée sur la sortie de l'encodeur, avant un feed-forward, chaque sous-couche étant entourée de Add & Norm. La sortie finale passe par une couche linéaire puis une softmax pour produire les probabilités sur le vocabulaire.

6 Exercices d'Application

Exercice 1 : Paradigmes de l'IA

Pour chacune des tâches d'IA suivantes, déterminez si elle relève principalement du paradigme **discriminatif** ou **génératif**. Justifiez votre réponse en une phrase en vous basant sur les concepts du cours (analyse vs. création, classification vs. production).

1. Un système qui examine une radiographie et détermine si elle présente des signes de pneumonie ou non.
2. Un programme qui compose une sonate pour piano dans le style de Mozart.
3. Une application qui prend une photo de votre salon et suggère une nouvelle disposition des meubles en générant une nouvelle image.
4. Un algorithme qui analyse les transactions par carte de crédit en temps réel pour bloquer celles qui sont potentiellement frauduleuses.
5. Un outil qui lit un article de recherche de 20 pages et en écrit un résumé de 200 mots.
6. Un logiciel dans une usine qui inspecte des pièces sur une chaîne de montage et les écarte si elles présentent des défauts.

Exercice 2 : Le Rôle du Contexte

Considérez la phrase suivante :

"Le chef prépare le plat avec soin, car il sait que le critique gastronomique le plus redouté du pays est assis dans la salle, attendant son..."

L'objectif d'un LLM est de prédire le mot suivant.

1. Quel serait, selon vous, le mot le plus probable à prédire pour un LLM moderne ? Proposez 2 ou 3 alternatives plausibles.
2. Expliquez pourquoi un ancien modèle de type RNN aurait eu beaucoup de difficultés à faire une prédiction pertinente pour cette phrase. Quel concept clé, vu dans le chapitre, lui fait défaut ?
3. Comment l'architecture **Transformer** résout-elle ce problème ? Décrivez en une phrase le rôle du mécanisme d'attention dans ce contexte précis.
4. Si l'on remplaçait "critique gastronomique" par "inspecteur sanitaire" dans la phrase, comment cela changerait-il la prédiction du mot final ? Qu'est-ce que cela nous apprend sur l'importance du contexte pour un LLM ?

Chapitre 2 : Anatomie d'un Modèle Génératif

7 Le Vocabulaire : Les Atomes du Langage

Un LLM ne lit pas le texte comme un humain. Sa première tâche est de décomposer une phrase en unités qu'il peut comprendre. Ces unités, appelées **tokens**, constituent son **vocabulaire**.

Un token n'est pas nécessairement un mot. Il peut être un mot entier, un groupe de lettres, un seul caractère ou un signe de ponctuation. Cette technique, appelée *Byte Pair Encoding* (BPE), permet au modèle de gérer des mots rares ou inconnus en les décomposant en sous-unités qu'il a déjà vues.

Pour un modèle comme **GPT-3**, le vocabulaire est fixe et contient environ **50 257 tokens**. C'est le dictionnaire de tous les "atomes de langage" que le modèle peut reconnaître et générer. Chaque token est associé à un identifiant unique (un entier).

8 Le Principe de l'Embedding : Traduire les Mots en Vecteurs

Un ordinateur ne peut pas manipuler directement des tokens textuels. Pour qu'un modèle puisse effectuer des calculs, chaque token doit être transformé en une représentation numérique : un **vecteur**. Ce processus de transformation est appelé **embedding**.

L'objectif de l'embedding n'est pas seulement de numériser les mots, mais de capturer leur **signification sémantique**. Dans l'espace mathématique créé par ces vecteurs, des mots sémantiquement proches (comme "chien" et "chat") auront des vecteurs proches, tandis que des mots éloignés (comme "chien" et "philosophie") auront des vecteurs très distants.

8.1 Les Étapes de Création d'un Embedding

Le processus se déroule en deux temps :

1. **La Tokenisation :** Le texte d'entrée est segmenté en une séquence de tokens à partir du vocabulaire.
2. **La Vectorisation :** Chaque token est associé à un vecteur dense, c'est-à-dire un long tableau de nombres à virgule flottante.

Dans le contexte de GPT-3 (modèle 'davinci'), chaque token est représenté par un vecteur de **12 288 dimensions**. C'est un espace difficile à imaginer, mais c'est cette haute dimensionnalité qui permet de capturer les nuances subtiles du langage.

8.2 La Matrice d'Embedding

On peut représenter l'embedding comme une immense table de consultation, appelée **matrice d'embedding**. C'est la première couche d'un LLM.

— Le nombre de **lignes** de cette matrice est égal à la taille du vocabulaire (environ 50 257).

— Le nombre de **colonnes** est égal à la dimension de l'embedding (12 288).

La dimension de cette matrice est donc : (taille du vocabulaire$\times$dimension de l'embedding), soit (50257×12288).

Lorsqu'un token (par exemple, le token no. 42) entre dans le modèle, ce dernier va simplement chercher la 42ème ligne de cette matrice pour obtenir le vecteur qui le représente. Les valeurs de ces vecteurs sont les **poids** ou **paramètres** que le modèle apprend **durant son entraînement**.

9 Interlude Mathématique : Les Espaces Vectoriels

Pour comprendre l'intuition géométrique des embeddings, un bref rappel sur les espaces vectoriels est utile. Un espace vectoriel peut être vu comme un "terrain de jeu" pour les vecteurs. C'est un ensemble où deux opérations sont définies :

1. **L'addition de vecteurs :** $\vec{u} + \vec{v} = \vec{w}$ (se déplacer d'un point à un autre).

2. **La multiplication par un scalaire :** $a \cdot \vec{u}$ (étirer ou contracter un vecteur).

L'espace vectoriel utilisé par les LLM est $\mathbb{R}^n$, où n est la dimension de l'embedding (par exemple, $n = 12288$). Dans cet espace, les notions de **distance** et de **direction** (ou angle) ont un sens. C'est cette propriété qui permet de représenter la sémantique de manière géométrique.

10 L'Interprétation Géométrique des Embeddings

La magie des embeddings appris par un LLM est que les relations sémantiques entre les mots se traduisent par des relations géométriques entre leurs vecteurs.

La relation la plus célèbre est l'analogie :

$$\overrightarrow{\text{Roi}} - \overrightarrow{\text{Homme}} + \overrightarrow{\text{Femme}} \approx \overrightarrow{\text{Reine}}$$

Cela signifie que le vecteur qui représente le "concept de royauté masculine" (de "Homme" à "Roi") est presque le même que celui qui représente le "concept de royauté féminine" (de "Femme" à "Reine").

10.1 Exemple Simplifié en 3D

Imaginons un espace sémantique simplifié à 3 dimensions :
- Axe 1 : **Genre** (de -1 pour Féminin à +1 pour Masculin)
- Axe 2 : **Royauté** (de 0 pour non-royal à +1 pour royal)
- Axe 3 : **Âge** (de -1 pour Jeune à +1 pour Adulte)

Après entraînement, les vecteurs de certains mots pourraient ressembler à ceci :
- $\overrightarrow{\text{Roi}} = [0.9, 0.95, 0.8]$ (très masculin, très royal, adulte)
- $\overrightarrow{\text{Reine}} = [-0.9, 0.92, 0.75]$ (très féminin, très royal, adulte)
- $\overrightarrow{\text{Prince}} = [0.85, 0.8, -0.9]$ (très masculin, assez royal, jeune)
- $\overrightarrow{\text{Homme}} = [0.98, 0.05, 0.7]$ (très masculin, non-royal, adulte)

Dans cet espace, **la direction et la distance entre les points ont une signification intuitive**.

11 Simulation de Similarité après Apprentissage

Pour mesurer la "proximité" sémantique entre deux mots, on ne calcule pas leur distance euclidienne mais plutôt l'angle entre leurs vecteurs. Cette mesure s'appelle la **similarité cosinus**. Elle varie de -1 (vecteurs opposés) à +1 (vecteurs identiques). Une valeur proche de 1 indique une forte similarité.

La formule est : $\text{similarité}(\vec{A}, \vec{B}) = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \|\vec{B}\|}$

11.1 Exemple de calcul

Supposons que notre modèle ait appris les vecteurs (simplifiés en 3D) suivants :
- $\overrightarrow{\text{chat}} = [2.5, 3.1, -1.2]$
- $\overrightarrow{\text{chien}} = [2.1, 2.8, -0.9]$
- $\overrightarrow{\text{voiture}} = [-0.5, -1.5, 3.5]$

1. Similarité entre "chat" et "chien" :

$$\text{sim}(\overrightarrow{\text{chat}}, \overrightarrow{\text{chien}}) = \frac{(2.5 \cdot 2.1) + (3.1 \cdot 2.8) + (-1.2 \cdot -0.9)}{\sqrt{2.5^2 + 3.1^2 + (-1.2)^2} \cdot \sqrt{2.1^2 + 2.8^2 + (-0.9)^2}} = \frac{15.01}{4.15 \cdot 3.61} \approx \mathbf{1.00}$$

Une similarité cosinus très proche de 1, ce qui confirme que les concepts sont sémantiquement très proches.

2. Similarité entre "chat" et "voiture" :

$$\text{sim}(\overrightarrow{\text{chat}}, \overrightarrow{\text{voiture}}) = \frac{(2.5 \cdot -0.5) + (3.1 \cdot -1.5) + (-1.2 \cdot 3.5)}{\sqrt{2.5^2 + 3.1^2 + (-1.2)^2} \cdot \sqrt{(-0.5)^2 + (-1.5)^2 + 3.5^2}} = \frac{-10.1}{4.15 \cdot 3.84} \approx \mathbf{-0.63}$$

Une similarité négative, indiquant que les concepts sont perçus comme très différents, voire opposés, dans cet espace sémantique.

12 Exercices d'Application

Exercice 1 : Dimensions et Paramètres

La première version de GPT-2 avait une dimension d'embedding de 768 et un vocabulaire de 50 257 tokens.

1. Quelle est la dimension de sa matrice d'embedding ?
2. Combien de paramètres (poids) cette matrice contient-elle à elle seule ?
3. Comparez ce nombre au nombre de paramètres de la matrice d'embedding de GPT-3.

Exercice 2 : Interprétation Géométrique

En reprenant notre espace sémantique simplifié en 3D (Genre, Royauté, Âge), imaginez et écrivez les coordonnées approximatives que pourraient avoir les vecteurs suivants :

1. $\overrightarrow{\text{Princesse}}$
2. $\overrightarrow{\text{Femme}}$
3. $\overrightarrow{\text{Garçon}}$

Justifiez brièvement vos choix pour chaque coordonnée.

Exercice 3 : L'Anatomie d'une Phrase

Imaginons un LLM "jouet" avec un vocabulaire très simple de 10 tokens et une dimension d'embedding de 8.

Son vocabulaire est : `{1:"le", 2:"chat", 3:"mange", 4:"la", 5:"souris", 6:".", 7:"et", 8:"dort", 9:"sur", 10:"tapis"}`

On lui donne la phrase suivante à traiter : **le chat mange la souris .**

1. **Tokenisation :** Quelle est la séquence d'identifiants (IDs) qui représente cette phrase ?
2. **Matrice d'Embedding :** Quelle est la dimension de la matrice d'embedding complète de ce modèle jouet ? Combien de paramètres (poids) contient-elle au total ?
3. **Représentation de la phrase :** Une fois les embeddings récupérés, la phrase est représentée par une nouvelle matrice. Combien de lignes et de colonnes cette matrice possède-t-elle ?
4. **Mot inconnu :** Que se passerait-il si on donnait au modèle la phrase "le **chien** mange" ? Comment le processus de tokenisation gère-t-il généralement ce problème dans les vrais LLM (comme GPT-3) ?

Exercice 4 : Le Voisinage Sémantique

Après son entraînement, un modèle a appris les relations sémantiques entre différents concepts. On mesure la similarité cosinus entre le vecteur du mot **voiture** et ceux d'autres mots. Voici les scores obtenus :

— $\text{sim}(\overrightarrow{\text{voiture}}, \overrightarrow{\text{camion}}) = 0.89$
— $\text{sim}(\overrightarrow{\text{voiture}}, \overrightarrow{\text{route}}) = 0.71$
— $\text{sim}(\overrightarrow{\text{voiture}}, \overrightarrow{\text{fleur}}) = -0.34$
— $\text{sim}(\overrightarrow{\text{voiture}}, \overrightarrow{\text{moteur}}) = 0.81$
— $\text{sim}(\overrightarrow{\text{voiture}}, \overrightarrow{\text{planète}}) = -0.95$

1. Quel mot est considéré comme sémantiquement le plus proche de "voiture" par le modèle ?
2. Quel mot est considéré comme le plus éloigné (ou le plus opposé) ?
3. La similarité entre "voiture" et "route" est plus faible qu'entre "voiture" et "moteur". Proposez une explication à ce résultat du point de vue du modèle. (Indice : pensez aux types de relations : "est un", "se déplace sur", "fait partie de").
4. Sans faire de calcul, où situeriez-vous approximativement la similarité cosinus entre $\overrightarrow{\text{voiture}}$ et $\overrightarrow{\text{vélo}}$? Justifiez votre estimation.

Chapitre 3 : Le Mécanisme d'Attention

13 L'Objectif : Permettre aux Mots de se "Parler"

Dans le chapitre précédent, nous avons vu que chaque mot (token) est transformé en un vecteur, ou "embedding". Cependant, cet embedding initial ignore le contexte. Le mot "robot" a le même embedding de base dans "un petit robot" que dans "le robot de cuisine".

Le **mécanisme d'attention** est le processus qui permet de raffiner ces embeddings. Son but est de permettre à chaque mot de "regarder" les autres mots de la phrase pour mieux comprendre son propre rôle et sa signification dans ce contexte spécifique.

Pour illustrer ce mécanisme, nous suivrons pas à pas le traitement de la phrase :

Un petit robot bleu curieux traverse le couloir silencieux .

14 Prérequis : Dimensions et Matrices de Départ

Avant les calculs, posons le cadre.

— L : La longueur de la séquence, ici $L = 10$ tokens.

— d_{model} : La dimension de l'embedding de chaque token à l'entrée. C'est la "richesse" de la représentation initiale du mot. Pour un grand modèle comme **GPT-3**, $d_{\text{model}} = 12288$.

— d_k : La dimension des vecteurs que nous allons créer. Elle est souvent plus petite que d_{model}. Dans GPT-3, $d_k = 128$.

Pour notre exemple, nous utiliserons une dimension très faible de $\mathbf{d_k = 3}$. L'entrée du mécanisme est une matrice $X \in \mathbb{R}^{L \times d_{\text{model}}}$ qui empile les embeddings de nos 10 tokens.

15 Étape 1 : Création des Requêtes, Clés et Valeurs (Q, K, V)

Pour que les mots puissent interagir, le modèle ne peut pas utiliser directement les embeddings de départ. Il doit d'abord les transformer pour leur assigner des rôles spécialisés. C'est l'étape de **projection**.

L'ensemble des embeddings de notre phrase forme une matrice d'entrée X. Le modèle apprend trois matrices de poids distinctes, W_Q, W_K et W_V, qui vont agir comme des "projecteurs". Chaque embedding sera multiplié par ces trois matrices pour créer trois nouveaux vecteurs : une **Requête**, une **Clé** et une **Valeur**.

$$Q = X \cdot W_Q \quad | \quad K = X \cdot W_K \quad | \quad V = X \cdot W_V$$

Analogie. Pensez à un acteur (l'embedding) qui doit jouer trois scènes différentes. Pour chaque scène, il met un costume différent (les matrices W) pour adopter un rôle spécifique (Q, K, ou V).

— **La Requête (Query, Q)** : C'est le mot qui cherche de l'information. La matrice W_Q apprend à transformer l'embedding en une question pertinente : "Quels mots sont importants pour moi ?".

— **La Clé (Key, K)** : C'est "l'étiquette" que chaque mot présente. La matrice W_K apprend à transformer l'embedding en une "annonce" : "Voici le type d'information que je présente".

— **La Valeur (Value, V)** : C'est le "contenu" réel du mot. La matrice W_V apprend à extraire de l'embedding l'information la plus utile à transmettre **si le mot (la clé(s)) est jugé pertinent**.

Pour notre exemple, nous allons directement utiliser des vecteurs Q, K et V simplifiés (avec une dimension de 3 au lieu des 128 de GPT-3) pour illustrer les calculs qui suivent.

16 Étape 2 : Calcul des Scores d'Affinité Bruts

Pourquoi ? Maintenant que chaque mot a une Requête et une Clé, nous pouvons mesurer leur compatibilité. Cela nous permettra de savoir à quel point chaque mot doit prêter attention à chaque autre mot.

Comment ? Cette affinité est mesurée par un **produit scalaire** entre le vecteur Requête du mot i (Q_i) et le vecteur Clé du mot j (K_j). On divise le résultat par $\sqrt{d_k}$ (la racine de la dimension des vecteurs) pour stabiliser les calculs.

$$S_{ij} = \frac{Q_i \cdot K_j}{\sqrt{d_k}}$$

Le résultat est une matrice S de taille $(L \times L)$ contenant tous les scores d'affinité bruts.

Exemple. Voici la matrice des scores bruts pour notre phrase :

Requête (Q) \ Clé (K)	Un	petit	robot	bleu	curieux	traverse	le	couloir	silencieux	.
Un	0.300	1.000	1.600	0.600	0.200	0.300	0.100	0.800	0.200	0.100
petit	0.800	0.300	2.600	1.000	0.600	0.300	0.100	0.400	0.100	0.100
robot	0.500	2.000	0.400	1.800	1.800	1.200	0.100	0.200	0.100	0.200
bleu	0.300	1.200	2.400	0.300	0.900	0.300	0.100	0.300	0.100	0.100
curieux	0.300	1.200	2.400	0.900	0.300	0.300	0.100	0.300	0.100	0.100
traverse	0.200	0.300	1.800	0.200	0.200	0.400	0.900	1.600	0.700	0.200
le	0.200	0.200	0.300	0.100	0.100	0.600	0.400	2.600	1.400	0.100
couloir	0.200	0.200	0.600	0.200	0.100	1.400	1.600	0.400	2.200	0.200
silencieux	0.200	0.200	0.300	0.100	0.100	0.600	1.000	2.800	0.400	0.200
.	0.100	0.100	0.400	0.100	0.100	0.600	0.700	2.000	1.600	0.500

17 Étape 3 : Normalisation des Scores (Softmax)

Pourquoi ? Les scores bruts sont relatifs. Pour les utiliser comme des "pourcentages d'attention", il faut les normaliser pour que leur somme par ligne soit égale à 1.

Comment ? On applique une fonction **softmax** sur chaque ligne de la matrice des scores. Un score brut élevé deviendra un poids d'attention élevé (proche de 1), un score faible deviendra un poid proche de 0.

$$\alpha_{ij} = \text{softmax}(S_i)_j = \frac{e^{S_{ij}}}{\sum_{k=1}^{L} e^{S_{ik}}}$$

Le résultat, α, est la **matrice d'attention**.

Exemple.

Requête (Q) \ Clé (K)	Un	petit	robot	bleu	curieux	traverse	le	couloir	silencieux	.
Un	0.071	0.143	**0.260**	0.096	0.064	0.071	0.058	0.117	0.064	0.058
petit	0.080	0.049	**0.485**	0.098	0.066	0.049	0.040	0.054	0.040	0.040
robot	0.054	**0.241**	0.049	**0.198**	**0.198**	0.108	0.036	0.040	0.036	0.040
bleu	0.053	0.130	**0.432**	0.053	0.096	0.053	0.043	0.053	0.043	0.043
curieux	0.053	0.130	**0.432**	0.096	0.053	0.053	0.043	0.053	0.043	0.043
traverse	0.053	0.058	**0.261**	0.053	0.053	0.064	0.106	0.213	0.087	0.053
le	0.044	0.044	0.048	0.040	0.040	0.065	0.053	0.482	0.145	0.040
couloir	0.045	0.045	0.067	0.045	0.040	0.148	0.181	0.055	0.330	0.045
silencieux	0.041	0.041	0.045	0.037	0.037	0.061	0.092	0.554	0.050	0.041
.	0.047	0.047	0.063	0.047	0.047	0.077	0.085	0.311	0.209	0.069

Lecture. La ligne "robot" nous indique qu'il accorde 24.1% de son attention à "petit", 19.8% à "bleu" et 19.8% à "curieux". Le modèle a bien compris les relations sémantiques.

18 Étape 4 : Le Masque Causal (Spécificité des Décodeurs)

Pourquoi ? Pour un modèle génératif comme GPT (un "décodeur"), il est crucial de ne pas "tricher" en regardant les mots futurs lors de la génération. Pour prédire le mot qui suit "robot", le modèle ne doit avoir accès qu'à "Un, petit, robot".

Comment ? On applique un "masque" qui met les scores d'attention vers le futur à $-\infty$ *avant* la softmax. Ainsi, leurs poids deviennent nuls.

Exemple.

Requête (Q) \ Clé (K)	Un	petit	robot	bleu	curieux	traverse	le	couloir	silencieux	.
Un	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
petit	0.622	0.378	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
robot	0.157	**0.702**	0.142	0.000	0.000	0.000	0.000	0.000	0.000	0.000
bleu	0.079	0.195	**0.647**	0.079	0.000	0.000	0.000	0.000	0.000	0.000
curieux	0.069	0.170	**0.565**	0.126	0.069	0.000	0.000	0.000	0.000	0.000
traverse	0.097	0.108	**0.482**	0.097	0.097	0.119	0.000	0.000	0.000	0.000
le	0.131	0.131	0.145	0.119	0.119	0.196	0.160	0.000	0.000	0.000
couloir	0.071	0.071	0.107	0.071	0.065	0.237	0.290	0.087	0.000	0.000
silencieux	0.043	0.043	0.047	0.039	0.039	0.064	0.095	0.577	0.052	0.000
.	0.047	0.047	0.063	0.047	0.047	0.077	0.085	0.311	0.209	0.069

Lecture. Avec le masque, "robot" ne peut plus voir "bleu" ou "curieux". Son attention est redistribuée sur les mots passés, principalement "petit" (70.2%). 1324

19 Exemple Concret Détaillé : de l'Embedding au Score d'Affinité

Pour bien comprendre le flux complet, nous allons décomposer le calcul pour obtenir un seul score d'affinité : celui de la Requête de "petit" pour la Clé de "robot".

1. Les Embeddings de Départ (Hypothétiques). Imaginons que nos mots sortent de la couche d'embedding avec des vecteurs de dimension $d_{\text{model}} = 4$.

— Embedding de "petit", $x_{\text{petit}} = [1.0, 0.5, 0.0, 0.0]$

— Embedding de "robot", $x_{\text{robot}} = [0.0, 0.0, 1.0, 0.8]$

2. Les Matrices de Projection (Apprises par le modèle). Le modèle a appris des matrices W_Q et W_K de dimension $(d_{\text{model}} \times d_k)$, soit (4×3). Elles servent à "projeter" nos embeddings dans l'espace plus petit des Requêtes et des Clés.

$$W_Q = \begin{bmatrix} 1.5 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \qquad W_K = \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 2.0 & 0.2 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}$$

(Note : Ces matrices sont simplifiées à l'extrême pour l'exemple. En réalité, elles sont denses et pleines de valeurs non nulles).

3. Calcul de la Requête pour "petit" (q_{petit}). On multiplie l'embedding de "petit" par la matrice W_Q.

$$q_{\text{petit}} = x_{\text{petit}} \cdot W_Q = [1.0, 0.5, 0.0, 0.0] \cdot \begin{bmatrix} 1.5 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}$$

$$q_{\text{petit}} = [(1.0 \cdot 1.5 + 0.5 \cdot 0 + \dots), (1.0 \cdot 0 + \dots), \dots] = [1.5, 0.0, 0.0]$$

Nous retrouvons bien le vecteur Q_{petit} de notre exemple.

4. Calcul de la Clé pour "robot" (k_{robot}). On multiplie l'embedding de "robot" par la matrice W_K.

$$k_{\text{robot}} = x_{\text{robot}} \cdot W_K = [0.0, 0.0, 1.0, 0.8] \cdot \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 2.0 & 0.2 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}$$

$$k_{\text{robot}} = [(0 \cdot 0 + \dots + 1.0 \cdot 2.0 + \dots), \dots, \dots] = [2.0, 0.2, 0.0]$$

Nous retrouvons bien le vecteur K_{robot} de notre exemple.

5. Calcul du Score d'Affinité Final. Maintenant que nous avons la Requête et la Clé, nous pouvons calculer leur affinité comme avant.

— $Q_{\text{petit}} = [1.5, 0, 0]$
— $K_{\text{robot}} = [2.0, 0.2, 0]$
— La dimension est $d_k = 3$, donc $\sqrt{d_k} \approx 1.732$.

Le score d'affinité de "petit" pour "robot" est :

$$s_{\text{petit,robot}} = \frac{(1.5 \times 2.0) + (0 \times 0.2) + (0 \times 0)}{1.732} = \frac{3.0}{1.732} \approx \mathbf{1.732}$$

Ce score, qui indique une forte relation, est ensuite utilisé pour construire la matrice complète des affinités brutes, comme vu précédemment.

Requête (Q) \ Clé (K)	Un	petit	robot	bleu	curieux	traverse	le	couloir	silencieux	.
Un	0.300	1.000	1.600	0.600	0.200	0.300	0.100	0.800	0.200	0.100
petit	0.800	0.300	**1.732**	1.000	0.600	0.300	0.100	0.400	0.100	0.100

20 Étape 5 : Agrégation des Valeurs ($O = \alpha V$)

Nous avons nos poids d'attention. Il est temps de les utiliser pour créer les nouveaux embeddings de mots, enrichis par le contexte.

Le calcul. Le nouvel embedding de sortie pour le mot i (O_i) est une **somme pondérée** de tous les vecteurs **Valeur** (V_j) de la phrase. Les poids de cette somme sont les scores d'attention α_{ij} que nous venons de calculer.

$$O_i = \sum_{j=1}^{L} \alpha_{ij} \cdot V_j$$

Exemple concret pour "robot". On reprend la ligne "robot" de la matrice d'attention (sans masque) et on multiplie chaque poids par le vecteur Valeur (V_j) correspondant.

$$O_{\text{robot}} = \quad 0.054 \cdot V_{\text{Un}} + \mathbf{0.241} \cdot V_{\text{petit}} + 0.049 \cdot V_{\text{robot}} + \mathbf{0.198} \cdot V_{\text{bleu}} + \dots$$

En additionnant toutes les contributions, on obtient le vecteur de sortie final pour "robot" :

$$\boxed{O_{\text{robot}} \approx [0.643, 0.247, 0.012, 0.136]}$$

Ce nouveau vecteur n'est plus l'embedding de base de "robot". C'est une représentation contextuelle, **qui a "absorbé"** les informations des autres mots. Ce vecteur enrichi est ensuite passé à la couche suivante du Transformer.

— Embedding de base "robot", $x_{\text{robot}} = [0.0, 0.0, 1.0, 0.8]$
— Embedding enrichi de "robot", $x_{\text{robot}} = [0.643, 0.247, 0.012, 0.136]$

Ce mécanisme, répété sur de multiples **"têtes d'attention"** en parallèle et sur plusieurs couches, est le secret de la compréhension profonde du langage par les LLM.

Chapitre 4 : L'Attention Multi-Têtes et le Bloc Décodeur

21 L'Objectif : La Limite d'une Seule Tête d'Attention

Pourquoi ? Dans le chapitre précédent, nous avons étudié le mécanisme d'attention (une "tête" unique). Cette tête apprend à calculer des poids d'affinité en projetant les embeddings dans un espace (Q, K, V). Cependant, elle est limitée. En apprenant, cette tête va se spécialiser dans un seul type de relation. Par exemple, elle pourrait devenir très performante pour lier les adjectifs à leurs noms, mais par conséquent, être médiocre pour lier un pronom à son sujet lointain.

Une seule tête d'attention force le modèle à choisir un **"angle d'analyse"** unique pour la phrase, ce qui est un goulot d'étranglement. Pour une compréhension profonde, le modèle doit pouvoir analyser la phrase sous de multiples perspectives simultanément.

Analogie. Si une seule tête d'attention est un **expert unique** (par exemple, un grammairien qui vérifie les accords), l'attention multi-têtes est un **comité d'experts** qui regardent la même phrase en même temps. Ce comité inclut un grammairien, un sémanticien (qui comprend le sens), un expert en anaphores (qui lie "il" à "robot"), etc. Chaque expert fournit son propre rapport, et la synthèse de ces rapports donne une compréhension beaucoup plus riche.

22 Le Mécanisme : L'Attention Multi-Têtes (Multi-Head Attention)

Comment ? L'idée est d'exécuter le mécanisme d'attention du Chapitre 3, non pas une fois, mais h **fois en parallèle**. Le nombre de têtes, h, est un hyperparamètre (par ex : 12 pour GPT-2, 96 pour GPT-3).

Chaque "tête" (i, de 1 à h) apprend son propre jeu de matrices de projection : $W_Q^{(i)}, W_K^{(i)}, W_V^{(i)}$.

1. L'embedding d'entrée X (de dimension d_{model}) est donné à toutes les têtes.
2. **Tête 1** calcule son propre O_1 : $Q_1 = XW_Q^{(1)}, K_1 = XW_K^{(1)}, V_1 = XW_V^{(1)}$
   $\alpha_1 = \text{softmax}(\frac{Q_1 K_1^\top}{\sqrt{d_k}})$
   $O_1 = \alpha_1 V_1$
3. **Tête 2** calcule son propre O_2 : $Q_2 = XW_Q^{(2)}, K_2 = XW_K^{(2)}, V_2 = XW_V^{(2)}$
   ...
   $O_2 = \alpha_2 V_2$
4. ...
5. **Tête** h calcule son propre O_h.

Puisque chaque tête a des poids W différents (initialisés aléatoirement), chaque tête va se spécialiser et "regarder" différentes choses. O_1 contiendra une information contextuelle différente de O_2.

Dimensions des Têtes. Chaque tête d'attention i (de 1 à h) produit sa propre matrice de sortie contextuelle, O_i, en calculant $O_i = \alpha_i V_i$.

La dimension de la matrice des Valeurs V_i et de la matrice de sortie O_i est la même. **Elle n'est pas d_{model} dans le context de multi-têtes, mais une dimension plus petite,** d_v.

Dans l'architecture Transformer standard, on choisit $d_v = d_k = d_{\text{model}}/h$.

Par exemple, pour GPT-3 ($d_{\text{model}} = 12288$ et $h = 96$ têtes) :

— L'embedding d'entrée X a une dimension de 12288.

— Chaque tête (Tête 1, Tête 2, ... Tête 96) crée une matrice Valeur V_i de dimension $(L \times \mathbf{128})$.

— Après la multiplication $\alpha_i V_i$, la matrice de sortie individuelle O_i a aussi la dimension $(L \times \mathbf{128})$.

Ce n'est qu'à l'étape suivante (Étape 3 : Agrégation) que ces 96 petites matrices de sortie (chacune de dimension $L \times 128$) seront concaténées et re-projetées pour recréer une seule matrice finale, O, qui aura bien la dimension $L \times d_{\text{model}}$ (soit $L \times 12288$).

23 Étape 3 : Agrégation et Projection de Sortie

Pourquoi ? À la fin de l'étape 2, nous avons h vecteurs de sortie (les "rapports" de nos h experts) pour chaque mot. Nous ne pouvons pas passer h vecteurs à la couche suivante. Nous devons les agréger en un **seul vecteur unifié** qui synthétise toutes ces perspectives.

Comment ? Le processus se fait en deux temps :

1. **Concaténation :** Les h vecteurs de sortie $(O_1, O_2, \ldots, O_h)$ sont simplement "collés" bout à bout (concaténés) pour former un très long vecteur.

$$\text{Concat} = [O_1|O_2|\ldots|O_h]$$

   La dimension de ce vecteur est $(L \times (h \cdot d_k))$. Notons que $h \cdot d_k = d_{\text{model}}$.

2. **Projection de Sortie :** Ce long vecteur est multiplié par une dernière matrice de poids apprise, W_O (Matrice de Sortie), de dimension $(d_{\text{model}} \times d_{\text{model}})$. Cette projection finale mélange les informations des h têtes pour produire le vecteur de sortie final, O, qui a la bonne dimension $(L \times d_{\text{model}})$.

La formule complète de l'Attention Multi-Têtes est donc :

$$\text{MultiHead}(X) = \text{Concat}(O_1, \ldots, O_h) \cdot W_O$$

Comment la matrice W_O est-elle obtenue ? La matrice W_O n'est pas "calculée" par une formule fixe. Elle est **apprise** (ou "découverte") par le modèle pendant la phase d'entraînement, de la même manière que tous les autres poids (W_Q, W_K, W_V, etc.).

- **Le Point de Départ (Pourquoi ?) :** Au tout début de l'entraînement, la matrice W_O est initialisée avec des millions de valeurs aléatoires. Elle ne sait absolument pas comment "mélanger" les informations des différentes têtes. Son "mixage" initial est inutile.
- **Le Processus d'Apprentissage (Comment ?) :** Pendant l'entraînement, le modèle traite des milliards de phrases.
  1. Il fait une prédiction (par exemple, il prédit "chat" au lieu de "chien").
  2. Il calcule son erreur.
  3. Un algorithme (la "rétropropagation") envoie un signal de correction à *tous* les poids du modèle, y compris à chaque valeur dans la matrice W_O.
  4. Ce signal dit à W_O : "Ton 'mixage' a contribué à cette erreur. Ajuste tes poids très légèrement pour donner un peu plus d'importance à la Tête 5 et un peu moins à la Tête 2 dans ce contexte."

Analogie : Le Rapporteur Stagiaire. Reprenons notre analogie du "Rapporteur Principal" (W_O). Au début, c'est un stagiaire. Il prend les 12 rapports des experts (les O_i) et les mélange au hasard pour écrire son résumé (le vecteur O). Son résumé est mauvais. Après chaque erreur, son manager (l'algorithme d'entraînement) lui dit : "Ton résumé est mauvais, tu as ignoré l'avis crucial de l'expert en sémantique (Tête 5) !". Le stagiaire (W_O) apprend de cette erreur et ajuste sa "recette de mixage". Après avoir fait cela des milliards de fois, il devient un expert qui sait exactement comment pondérer chaque tête pour produire le résumé parfait.

24 Le Bloc Décodeur Complet : L'Architecture Résiduelle

Le mécanisme d'Attention Multi-Têtes (MHA) est le cœur, mais ce n'est pas tout le bloc. Un bloc décodeur (comme vu dans le schéma du Chapitre 1) contient deux sous-couches principales.

1. La Connexion Résiduelle (Add & Norm)

Pourquoi ? L'attention raffine l'embedding X pour produire un output O. Cependant, en se concentrant sur le contexte, le modèle pourrait "oublier" le mot de départ lui-même. De plus, empiler des couches de calcul mène à des problèmes de "disparition de gradient" (le signal d'apprentissage se perd).

Comment ? La solution est la **connexion résiduelle** ("Add"). On prend simplement l'embedding d'entrée X et on *l'additionne* au résultat O de l'attention.

Analogie. C'est comme garder une "copie de sauvegarde". On envoie l'original (X) dans le mécanisme d'attention pour obtenir un "rapport contextuel" (O), puis on attache ce rapport à l'original. Le résultat final est $X + O$.

On applique ensuite une "Normalisation de Couche" (LayerNorm) pour re-stabiliser les valeurs. La sortie de la première sous-couche est donc :

$$Y = \text{LayerNorm}(X + \text{MultiHead}(X))$$

2. Le Réseau Feed-Forward (FFN)

Pourquoi ? L'étape d'attention a permis de *collecter* l'information contextuelle. Le FFN sert à *traiter* ou "digérer" cette information. C'est l'étape de calcul ou de "réflexion" du bloc. Mais que signifie "trouver des motifs plus complexes" ? Le vecteur Y qui sort de l'étape d'attention est un **mélange** d'informations (par exemple, il contient des "traces" de "robot", "petit", "traverse" et "couloir"). C'est une collection, une somme pondérée. Le FFN, qui est un réseau de neurones à deux couches, va **transformer** cette collection. Il ne se contente pas de la transmettre ; il trouve des relations non-linéaires entre les informations collectées. C'est là que les concepts sont réellement formés.

- Il peut apprendre que lorsque le contexte contient "pas" ET "très" ET "bon" (collectés par l'attention), le "motif complexe" qu'il doit créer est "NÉGATION PUISSANTE".
- Il peut apprendre que "critique gastronomique" + "attendant son..." implique un "MOTIF DE JUGEMENT IMPORTANT", ce qui change la signification des mots qui suivront.

Analogie 1 : Le Chef Cuisinier. Si l'Attention Multi-Têtes est un commis de cuisine qui *rassemble* tous les ingrédients bruts sur le plan de travail (le vecteur Y contient : [tomate, basilic, fromage]), le FFN est le **Chef Exécutif**. Le Chef ne se contente pas de regarder les ingrédients ; il les *transforme*. Il les coupe, les cuit et les combine pour créer un "motif complexe" : un plat fini, par exemple une "sauce savoureuse". Le FFN prend les informations brutes collectées et les "cuit" pour en faire une idée ou une abstraction de plus haut niveau.

Analogie 2 : Le Département de Synthèse. Imaginez que l'Attention Multi-Têtes est un groupe de 96 analystes (les têtes) qui déposent chacun une note de 128 mots sur votre bureau. Votre vecteur d'entrée Y est juste ces 96 notes agrafées ensemble. Vous (le FFN) lisez toutes ces notes, vous "réfléchissez", et vous écrivez un seul mémo unifié qui dit : "L'analyse de tous ces fragments suggère que notre concurrent va lancer un nouveau produit". Vous avez détecté un motif complexe (une intention) que_aucun_ des analystes n'avait écrit individuellement. En résumé, si l'Attention **collecte** le contexte, le FFN **pense** à ce contexte pour en déduire de nouvelles idées.

Comment ? Le vecteur Y (sorti de la première sous-couche) passe à travers un simple réseau de neurones à deux couches, appliqué indépendamment à chaque token.

Le FFN, qui est un réseau de neurones à deux couches, effectue une opération en deux temps :

1. **Une couche d'expansion (la "réflexion") :** Le vecteur Y (de dimension d_{model}) est d'abord projeté dans un espace beaucoup plus grand, typiquement 4 fois sa taille ($4 \times d_{\text{model}}$).
2. **Une couche de contraction (la "synthèse") :** Ce grand vecteur intermédiaire est ensuite re-projeté pour revenir à sa taille d'origine (d_{model}).

Pourquoi faire cela ? Cette expansion n'est pas un calcul inutile ; c'est l'espace où le modèle "réfléchit".

Analogie : La Salle de Brainstorming ;)

- Le vecteur d'entrée (d_{model}) est une **idée compacte**.
- **L'expansion (Étape 1)**, c'est comme "entrer dans une grande salle de brainstorming". En projetant l'idée dans un espace 4 fois plus grand, le modèle a plus de "place" pour déballer tous les concepts, les relations et les implications cachés dans cette idée. Il peut y trouver des motifs non-linéaires qu'il ne pouvait pas "voir" dans l'espace d'origine, plus restreint.
- **La contraction (Étape 2)**, c'est comme "sortir de la salle". Le modèle rassemble toutes les nouvelles connexions qu'il a trouvées et les synthétise en un nouveau vecteur, plus riche et mieux compris, de la taille d_{model} originale.

Donc, la phrase peut être comprise comme : "...un réseau de neurones à deux couches : une première couche qui projette le vecteur dans un **espace de 'réflexion' de plus grande dimension**, et une seconde qui **synthétise cette réflexion** pour la ramener à la dimension d'origine.

3. Assemblage Final Le FFN est lui-même suivi d'une connexion résiduelle. Le bloc décodeur complet suit donc ce schéma pour chaque token :

1. $X_{\text{in}} \rightarrow$ **Multi-Head Attention** (masquée) $\rightarrow O$
2. $Y = \text{LayerNorm}(X_{\text{in}} + O)$ *(1ère connexion résiduelle)*
3. $Y \rightarrow$ **Feed-Forward Network** $\rightarrow F$
4. $X_{\text{out}} = \text{LayerNorm}(Y + F)$ *(2ème connexion résiduelle)*

Ce X_{out} est la sortie finale du premier bloc décodeur. Il est ensuite passé comme **entrée** au bloc décodeur suivant (puisqu'un LLM est une pile de N de ces blocs).

25 Le Paradoxe du Parallélisme : Entraînement vs. Génération

Le Transformer est célèbre pour son architecture parallèle, qui le rend bien plus rapide à entraîner que les anciens modèles (RNN). Cependant, nous avons aussi vu que la génération de mots est un processus séquentiel, et que les blocs sont empilés séquentiellement.

Comment ces deux idées peuvent-elles être vraies en même temps ? La réponse est qu'il existe **deux types de flux** : un flux "horizontal" (sur les mots) et un flux "vertical" (sur les blocs). Le parallélisme ne s'applique pas partout.

1. Le Parallélisme (Pendant l'Entraînement) : Horizontal $\leftrightarrow$

Pourquoi ? Pendant l'entraînement, le modèle n'a pas besoin de "générer" une phrase. Il a déjà la phrase complète (par exemple, `[Le, chat, dort]`). Son seul objectif est d'analyser cette phrase d'un coup pour apprendre. C'est là que le parallélisme intervient.

Comment ? Le calcul **sur les mots de la phrase** se fait en parallèle.

- **Attention Multi-Têtes :** Comme nous l'avons vu, les 12 ou 96 têtes travaillent toutes en même temps (parallélisme des têtes). De plus, le calcul d'attention pour le mot "dort" (qui regarde "Le" et "chat") n'a pas besoin d'attendre que le calcul pour "chat" soit terminé et la compréhension de la phrase par une autre tête. Tout est un grand produit matriciel.
- **Réseau Feed-Forward (FFN) :** C'est le point clé. Le FFN est appliqué **indépendamment à chaque mot**. Le calcul FFN pour "Le", le calcul FFN pour "chat" et le calcul FFN pour "dort" n'ont aucune dépendance entre eux. Un GPU peut donc exécuter ces 3 opérations (ou 1000, pour une phrase plus longue) exactement en même temps.

Analogie : La Salle d'Examen. Imaginez l'entraînement sur la phrase `[Le, chat, dort]`.

- Le **Bloc 1** est la "Salle d'Examen 1".
- Les mots `[Le, chat, dort]` sont **3 étudiants**.
- Le FFN est l'épreuve : le professeur (le FFN) donne l'épreuve aux 3 étudiants **en même temps**. Les 3 étudiants travaillent **en parallèle**.

2. Le Séquentiel (Partie 1) : Vertical $\uparrow$

Pourquoi ? L'architecture du Transformer est une **pile de blocs** (par exemple, 96 blocs). Chaque bloc raffine le travail du bloc précédent.

Comment ? La sortie du Bloc 1 est l'entrée du Bloc 2. Le Bloc 2 ne peut donc pas commencer son calcul tant que le Bloc 1 n'a pas *totalement* terminé son travail sur *tous* les mots.

Analogie (suite) :
- Les 3 étudiants (mots) doivent tous finir leur épreuve dans la "Salle 1" (Bloc 1).
- **PUIS** (étape séquentielle), ils se déplacent tous ensemble vers la "Salle 2" (Bloc 2) pour commencer l'épreuve suivante.

Le flux *entre les blocs* ($1 \rightarrow 2 \rightarrow ... \rightarrow 96$) est donc **séquentiel**.

3. Le Séquentiel (Partie 2) : Pendant la Génération →

Pourquoi ? Lors de la génération (inférence), le modèle est "auto-régressif" : il doit générer ses propres mots un par un. Il ne connaît pas la phrase à l'avance.

Comment ? Pour prédire le 5ème mot, le modèle doit obligatoirement connaître les 4 mots qu'il vient de générer.

Analogie : L'Écrivain. Ici, le modèle n'est plus un étudiant, c'est un **écrivain**. Pour écrire "Le chat dort", il doit :

1. Écrire "Le".
2. **PUIS** (étape séquentielle) analyser "Le" pour décider d'écrire "chat".
3. **PUIS** (étape séquentielle) analyser "Le chat" pour décider d'écrire "dort".

26 Exercices d'Application

Exercice 1 : Dimensions et Paramètres

Un modèle de type GPT-2 a une dimension d'embedding $d_{\text{model}} = 768$ et utilise $h = 12$ têtes d'attention.

1. Quelle est la dimension d_k (la taille du vecteur de sortie) pour chaque tête individuelle ?
2. Après l'étape de concaténation (avant la projection W_O), quelle est la dimension du vecteur agrégé ?
3. Quelle est la taille de la matrice de projection finale W_O ?

Exercice 2 : Le Rôle des Connexions Résiduelles

1. En utilisant l'analogie de la "copie de sauvegarde", expliquez pourquoi il est utile de faire $X + \text{Attention}(X)$.
2. Que se passerait-il si le mécanisme d'attention, pour une raison quelconque, échouait et produisait un vecteur nul ($O = 0$) ? Le signal du mot original serait-il perdu ? Pourquoi ?

Exercice 3 : Suivi des Dimensions

Imaginons un modèle plus petit où $d_{\text{model}} = 512$ et nous avons $h = 8$ têtes d'attention. La phrase d'entrée a $L = 100$ mots. En suivant le schéma :

1. Quelle est la dimension de la matrice d'entrée X ?
2. Quelle est la dimension des matrices Q, K, et V après leur passage dans les premières couches "Linear" (en supposant $d_k = d_v = d_{\text{model}}/h$) ?
3. Quelle est la dimension de la sortie d'une **seule** "Scaled Dot-Product Attention" (avant la concaténation) ?
4. Quelle est la dimension de la matrice après l'étape "Concat" ?
5. La couche "Linear" finale (notre W_O) prend l'entrée de l'étape 4 et produit la "Multi-Head Output". Quelle doit être la dimension de cette matrice W_O pour que cela fonctionne ?

Chapitre 5 : L'Algorithme d'Apprentissage

27 L'Objectif : Comment le Modèle Apprend ?

Pourquoi ? Jusqu'à présent, nous avons assemblé une architecture incroyablement complexe (la pile de Blocs Décodeurs) remplie de millions de poids (Paramètres), comme les matrices W_Q, W_K, W_V, W_O et celles des FFN. Au début, tous ces poids sont **aléatoires**. Le modèle est un "cerveau" puissant mais vide. Il produit des prédictions qui n'ont aucun sens.

L'objectif de l'apprentissage est d'ajuster ces millions de poids, petit à petit, pour que les prédictions du modèle **ressemblent le plus possible à la réalité** (le texte écrit par des humains sur lequel nous l'entraînons).

Comment ? L'apprentissage est un cycle en quatre étapes, répété des milliards de fois :

1. **La Prédiction (Forward Pass)** : Le modèle fait une prédiction.
2. **Le Calcul de l'Erreur (Loss Function)** : On compare la prédiction à la réalité pour calculer un "score d'erreur".
3. **La Correction (Backward Pass)** : On détermine comment chaque poids dans le modèle a contribué à cette erreur.
4. **La Mise à Jour (Optimizer)** : On ajuste légèrement chaque poids dans la bonne direction pour réduire l'erreur la prochaine fois.

Nous allons détailler ce cycle.

28 Étape 1 : La Prédiction (Forward Pass)

Rappel de l'architecture. Après que les embeddings aient traversé la pile de N Blocs Décodeurs (Chapitre 4), nous obtenons un vecteur de sortie final, X_{final}, pour chaque mot.

Comment le modèle prédit-il un mot ? Pour transformer ce vecteur final en une prédiction, deux couches (vues au Chapitre 1) entrent en jeu :

1. **La Couche Linéaire Finale :** Le vecteur X_{final} (dimension d_{model}) est multiplié par une immense matrice de poids (souvent liée à la matrice d'embedding) pour le projeter dans l'espace du vocabulaire. On obtient un vecteur de "scores" (appelés **logits**), de dimension V (taille du vocabulaire, ex : 50 257).
2. **La Couche Softmax :** Ces scores sont transformés en une distribution de probabilité (des pourcentages).

Exemple. Le modèle a reçu "Le chat". Il fait passer ces mots dans toute la pile de blocs. Le vecteur final X_{final} pour le mot "chat" est envoyé aux couches de sortie.

Le résultat de la Softmax est un vecteur de 50 257 probabilités :

— $P(\text{"dort"}) = 0.35$ (35%)
— $P(\text{"mange"}) = 0.22$ (22%)
— $P(\text{"bleu"}) = 0.0001$ (0.01%)
— ...

C'est la **prédiction** du modèle.

29 Étape 2 : Le Calcul de l'Erreur (La Fonction de Perte)

Pourquoi ? Le modèle a prédit 35% pour "dort" et 22% for "mange". Nous devons lui dire quelle était la bonne réponse et à quel point il était "loin" de la vérité.

Comment ? Dans notre texte d'entraînement, le mot qui suivait "Le chat" était **"dort"**. C'est notre **vérité terrain** (Ground Truth).

Nous comparons la prédiction du modèle (le vecteur de 50k probabilités) à la vérité (le mot "dort") en utilisant une fonction de coût, la **Cross-Entropy Loss**.

La "vérité" est un vecteur où le bon mot a 100% de probabilité et tous les autres 0% :

— $Vérité(\text{"dort"}) = 1.0$ (100%)
— $Vérité(\text{"mange"}) = 0.0$ (0%)
— $Vérité(\text{"bleu"}) = 0.0$ (0%)
— ...

La "Loss" (l'erreur) est simplement $-\log(P(\text{mot correct}))$.

$$\text{Erreur} = -\log(0.35) = 1.05$$

Analogie : La "Surprise" du Modèle. La Loss est une mesure de "surprise".

— Le modèle a donné 35% au bon mot. Il n'est "pas très surpris". Son erreur est faible (1.05).
— Si le modèle avait prédit $P(\text{"dort"}) = 0.001$, son erreur aurait été $-\log(0.001) = 6.9$. Il aurait été "très surpris", et son score d'erreur aurait été très élevé.

L'objectif de l'entraînement est de minimiser cette "surprise", c'est-à-dire de minimiser ce score d'erreur (la Loss).

30 Étape 3 : La Correction (Backward Pass / Rétropropagation)

Pourquoi ? Nous avons un seul chiffre, l'Erreur (1.05). Ce chiffre nous dit "le modèle s'est trompé", mais il ne nous dit pas *qui* est responsable. Est-ce W_O du Bloc 48 ? W_Q du Bloc 1 ?

Comment ? C'est la magie du **gradient**. L'algorithme de "rétropropagation" va "casser" cette erreur et distribuer la responsabilité à **chaque paramètre** (poids) du modèle.

Le signal d'erreur (1.05) "coule" à l'envers dans le réseau, de la sortie vers l'entrée. Il traverse les FFN, les W_O, les Têtes d'Attention... jusqu'aux embeddings du Bloc 1. Chaque poids sur son chemin reçoit une information (son "gradient") qui lui dit :

"Pour réduire l'erreur finale de 1.05, toi, ce poids spécifique, tu aurais dû être un peu plus grand (gradient positif) ou un peu plus petit (gradient négatif)."

Analogie : L'Armée de Stagiaires. Imaginez que les 175 milliards de poids du modèle sont une armée de stagiaires.

1. Ils font une prédiction (Forward Pass).
2. Le "Grand Superviseur" (la Loss Function) crie "ERREUR DE 1.05 !".
3. Ce cri (le gradient) se propage à l'envers dans toute l'armée.
4. Chaque stagiaire (chaque poids) entend une instruction minuscule : "Toi, w_{123}, tu dois augmenter de +0.0004 !". "Toi, w_{124}, tu dois diminuer de -0.0001 !".

À la fin de cette étape, chaque paramètre du modèle sait exactement comment il doit changer pour corriger l'erreur.

31 Étape 4 : La Mise à Jour (L'Optimiseur)

Pourquoi ? Tous les stagiaires (poids) veulent changer. Mais s'ils changent tous d'un coup, ce sera le chaos. Il faut un manager pour orchestrer cette mise à jour.

Comment ? C'est le rôle de l'**Optimiseur** (par exemple, "Adam" ou "SGD"). Il prend tous les gradients (les "suggestions de changement") et les applique, mais de manière contrôlée, grâce au **Taux d'Apprentissage (Learning Rate)**.

Le Learning Rate est un chiffre très petit (ex : 0.0001).

Analogie : Le Manager Prudent. L'Optimiseur est le Manager qui dit à l'armée de stagiaires : *"J'ai entendu vos suggestions de changement. C'est bien. Mais nous allons être prudents. Ne faites que 0.01% (le Learning Rate) de la modification que vous avez suggérée."*

Le poids w_{123} ne change donc pas de $+0.0004$, mais de $0.0004 \times 0.0001 = +0.00000004$.

C'est un changement minuscule, mais en répétant ce cycle (Prédiction → Erreur → Correction → Mise à Jour) des milliards de fois, sur des téraoctets de texte, les poids convergent lentement vers des valeurs qui permettent au modèle de prédire le mot suivant avec une précision incroyable.

32 Zoom sur l'Étape 3 : La Rétropropagation (Backward Pass)

Pourquoi ? Dans l'étape 2, nous avons obtenu un seul chiffre, l'Erreur (Loss), qui nous dit *à quel point* le modèle s'est trompé. Mais pour corriger le modèle, nous avons besoin de savoir *qui* est responsable de l'erreur, et *comment* le corriger.

La rétropropagation est l'algorithme qui répond à cette question. Son but est de calculer le **gradient** de l'erreur par rapport à chaque poids (paramètre) du réseau.

Analogie : Le Manager et le Stagiaire. Imaginez un seul stagiaire (un poids, w) qui fait un calcul.

- Le Manager (la fonction d'erreur) voit la sortie et crie : "Erreur de 1.05 !".
- Le stagiaire w entend cela et demande : "Que dois-je faire ? Augmenter ou diminuer ma valeur, et de combien ?".
- La rétropropagation est le calcul mathématique qui permet au manager de répondre : "Ton action a contribué à l'erreur de telle manière. Pour la corriger, tu dois changer de X." Ce "X" est le gradient.

Exemple : Le Réseau le Plus Simple (Un Neurone)

Pour voir le calcul, oublions les 175 milliards de paramètres de GPT-3 et imaginons un "modèle" avec **un seul poids**, w.

Le Schéma de Calcul (Le Graphe). Notre modèle prend une entrée x et doit prédire une sortie y.

$$\texttt{(Entrée x)} \rightarrow [\textbf{Poids } w] \rightarrow \texttt{(Prédiction } pred\texttt{)} \rightarrow [\textbf{Erreur } E] \leftarrow \texttt{(Vérité } y\texttt{)}$$

Nous avons deux étapes de calcul (le "Forward Pass") :

1. La prédiction : $pred = x \cdot w$
2. L'erreur (Loss) : Nous utiliserons l'Erreur Quadratique, plus simple à dériver : $E = (pred - y)^2$

Objectif Mathématique. Nous voulons trouver la "responsabilité" du poids w dans l'erreur E. C'est la dérivée de E par rapport à w, notée :

$$\frac{\partial E}{\partial w}$$

Cette valeur (le gradient) nous dira exactement comment un petit changement de w affecte l'erreur E.

Le "Backward Pass" : La Règle de la Chaîne

Pourquoi ? L'erreur E ne dépend pas *directement* de w. E dépend de $pred$, qui lui-même dépend de w.

$$E \leftarrow pred \leftarrow w$$

Pour trouver comment w affecte E, nous devons "remonter la chaîne" en multipliant les dérivées partielles. C'est la **règle de la chaîne**.

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial pred} \cdot \frac{\partial pred}{\partial w}$$

Comment (Le Calcul) ? Décomposons le problème en deux parties simples.

Partie 1 : La responsabilité de la Prédiction. Comment l'erreur E change-t-elle quand $pred$ change ?

— Fonction : $E = (pred - y)^2$

— Dérivée : $\frac{\partial E}{\partial pred} = 2 \cdot (pred - y)$

(C'est la dérivée d'une fonction u^2, qui est $2u \cdot u'$).

Partie 2 : La responsabilité du Poids. Comment la prédiction $pred$ change-t-elle quand w change ?

— Fonction : $pred = x \cdot w$

— Dérivée : $\frac{\partial pred}{\partial w} = x$

(La dérivée de $a \cdot w$ par rapport à w est juste a. Ici, $a = x$).

Partie 3 : Le Résultat Final (Le Gradient). Maintenant, assemblons les morceaux avec la règle de la chaîne :

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial pred} \cdot \frac{\partial pred}{\partial w}$$

$$\boxed{\frac{\partial E}{\partial w} = 2 \cdot (pred - y) \cdot x}$$

Conclusion : L'Ordre de Correction

Nous avons notre gradient ! C'est l'ordre de correction pour le poids w. Pour mettre à jour le poids (Étape 4 : l'Optimiseur), on utilise cette valeur :

$$w_{\text{nouveau}} = w_{\text{ancien}} - (\text{learning\_rate} \cdot \frac{\partial E}{\partial w})$$

Le signe "moins" est important : si le gradient est positif (ce qui signifie qu'augmenter w augmente l'erreur), nous devons *diminuer* w pour réduire l'erreur.

C'est *exactement* ce processus, répété des milliards de fois sur des millions de chemins différents (via la règle de la chaîne), qui permet de distribuer la responsabilité de l'erreur à tous les poids du Transformer.

33 Zoom sur l'Étape 3 : Pourquoi le Signe "Moins" ? (La Descente de Gradient)

Pourquoi ? Dans l'étape 3, nous avons calculé le gradient, $\frac{\partial E}{\partial w}$. Ce gradient est le "signal de responsabilité". Notre formule de mise à jour (l'Optimiseur) est :

$$w_{\text{nouveau}} = w_{\text{ancien}} - (\text{learning\_rate} \cdot \frac{\partial E}{\partial w})$$

Pourquoi ce signe "moins" ? Pourquoi ne pas simplement *ajouter* le gradient ?

Explication Mathématique. Le gradient ($\frac{\partial E}{\partial w}$) est un vecteur (ou un nombre) qui, par définition, nous indique la direction de la **plus forte montée**. C'est la direction qu'il faut prendre pour *augmenter* l'erreur E le plus rapidement possible.

Notre objectif est exactement l'inverse : nous voulons **minimiser** l'erreur. Nous devons donc aller dans la direction **exactement opposée** au gradient.

Le signe "moins" ('-') est l'opérateur mathématique qui nous fait "faire demi-tour" et prendre la direction de la **descente** de gradient.

Analogie : Le Randonneur dans le Brouillard. Imaginez que votre modèle est un randonneur perdu dans un épais brouillard, au milieu d'une montagne.

— **La Montagne :** C'est la "surface de l'erreur" (la fonction de Loss).

— **L'Altitude (E) :** C'est votre score d'erreur actuel.

— **Votre Position (w) :** C'est la valeur actuelle de votre poids.

— **Votre Objectif :** Atteindre la vallée, l'altitude la plus basse possible (Erreur minimale = 0).

Dans le brouillard, vous ne voyez rien. La seule information que vous avez est la **pente** du sol sous vos pieds.

1. Vous tâtez le terrain. La "pente" que vous sentez, c'est le **gradient** ($\frac{\partial E}{\partial w}$).

2. Le gradient vous indique la direction pour **monter** la montagne le plus vite (la plus forte pente vers le haut).
3. Mais vous voulez **descendre** !
4. **L'Action Logique :** Vous identifiez la direction qui monte (le gradient), et vous faites demi-tour pour aller dans la direction opposée.

C'est exactement ce que fait l'équation :
— w_{ancien} : Votre position actuelle.
— $\frac{\partial E}{\partial w}$: La direction pour monter (le gradient).
— $-\frac{\partial E}{\partial w}$: La direction pour descendre (l'opposé du gradient).
— $w_{\text{nouveau}} = w_{\text{ancien}} - \dots$: Faire un petit pas dans la direction qui descend.

Exemple : Un Réseau de Deux Neurones (La Chaîne)

Pourquoi ? L'exemple à un neurone nous a montré comment un poids est mis à jour. Mais la puissance de la rétropropagation vient de sa capacité à "remonter" une erreur à travers *plusieurs* couches. L'erreur doit se propager en arrière, de neurone en neurone, pour que chaque poids apprenne sa part de responsabilité.

Le Schéma de Calcul (Le Graphe). Imaginons un "modèle" avec deux poids en série, w_1 et w_2. L'information x est d'abord traitée par w_1 pour créer une sortie "cachée" h, qui est ensuite traitée par w_2 pour créer la prédiction finale $pred$.

$$\texttt{(Entrée } x\texttt{)} \to [\textbf{Poids } w_1] \to \texttt{(Caché } h\texttt{)} \to [\textbf{Poids } w_2] \to \texttt{(Prédiction } pred\texttt{)} \to [\textbf{Erreur } E] \leftarrow \texttt{(Vérité } y\texttt{)}$$

Nous avons trois étapes de calcul (le "Forward Pass") :
1. Calcul du neurone 1 : $h = x \cdot w_1$
2. Calcul du neurone 2 : $pred = h \cdot w_2$
3. Calcul de l'erreur (Loss) : $E = (pred - y)^2$

Objectif Mathématique. Nous voulons trouver la "responsabilité" de *chaque* poids dans l'erreur E. Nous avons donc deux gradients à calculer :
1. $\frac{\partial E}{\partial w_2}$ (la responsabilité de w_2)
2. $\frac{\partial E}{\partial w_1}$ (la responsabilité de w_1)

Le "Backward Pass" : La Règle de la Chaîne Propagée

Analyse. Le calcul pour w_2 est simple (c'est le même que notre exemple à 1 neurone). Mais pour w_1, l'erreur E doit "remonter" à travers w_2 *puis* à travers h avant d'atteindre w_1.

La chaîne de dépendance pour w_1 est :

$$E \leftarrow pred \leftarrow h \leftarrow w_1$$

—

Calcul 1 : Gradient pour w_2 (le poids de sortie)

C'est l'étape la plus proche de l'erreur.

$$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial pred} \cdot \frac{\partial pred}{\partial w_2}$$

— $\frac{\partial E}{\partial pred} = 2 \cdot (pred - y)$
— $\frac{\partial pred}{\partial w_2} = h \quad$ (car $pred = h \cdot w_2$)

$$\boxed{\frac{\partial E}{\partial w_2} = 2 \cdot (pred - y) \cdot h}$$

—

Calcul 2 : Gradient pour w_1 (le poids caché)

Ici, nous devons propager l'erreur plus loin.

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial pred} \cdot \frac{\partial pred}{\partial h} \cdot \frac{\partial h}{\partial w_1}$$

Décomposons cette longue chaîne :
— $\frac{\partial E}{\partial pred} = 2 \cdot (pred - y)$ (On l'a déjà)
— $\frac{\partial pred}{\partial h} = w_2$ (car $pred = h \cdot w_2$, la dérivée par rapport à h)
— $\frac{\partial h}{\partial w_1} = x$ (car $h = x \cdot w_1$, la dérivée par rapport à w_1)
Assemblons les morceaux :

$$\boxed{\frac{\partial E}{\partial w_1} = [2 \cdot (pred - y)] \cdot [w_2] \cdot [x]}$$

Conclusion et Interprétation

Analogie : Le Flux d'Erreur. Regardons ce que nous avons trouvé. Le calcul de la rétropropagation se fait en "distribuant" l'erreur de droite à gauche.

1. On calcule d'abord l'erreur de base : $\frac{\partial E}{\partial pred} = 2 \cdot (pred - y)$. C'est notre "flux d'erreur" initial.
2. Pour w_2, on prend ce flux d'erreur et on le multiplie par l'entrée de w_2 (qui était h).
3. Pour w_1, c'est différent. Le flux d'erreur $\frac{\partial E}{\partial pred}$ doit d'abord "passer à travers" le neurone 2 pour savoir quelle était la responsabilité de h. Le "flux d'erreur arrivant à h" est $\frac{\partial E}{\partial h} = \frac{\partial E}{\partial pred} \cdot \frac{\partial pred}{\partial h} = [2 \cdot (pred - y)] \cdot w_2$.
4. On voit que le gradient de w_1 est ce "flux d'erreur arrivant à h" multiplié par l'entrée de w_1 (qui était x).

C'est ça, la **rétropropagation** : le signal d'erreur de base $(2 \cdot (pred - y))$ est **multiplié (ou "pondéré") par le poids** w_2 avant d'être "envoyé" en arrière pour corriger le poids w_1. L'erreur "coule" à l'envers, et chaque poids sur son chemin agit comme un multiplicateur pour le flux qui le traverse.