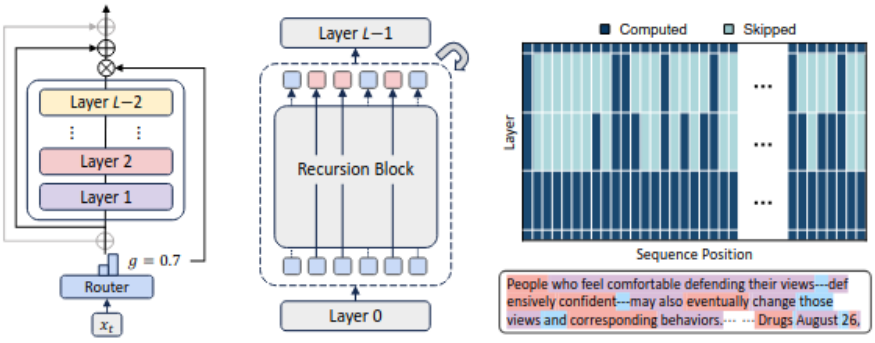
ANALYSE DÉTAILLÉE DE MIXTURE-OF-RECURSIONS (MOR)

Mokira
Ingénieur Machine Learning
(+229) 019 798 5109
dr.mokira@gmail.com

September 8, 2025

ABSTRACT

Quand on augmente la taille des modèles de langage (plus de couches, plus de paramètres, plus de données), ils deviennent plus performant et intelligents. Mais en contrepartie, leur entraînement coûte énormément (des semaines de GPU), et leur utilisation (en inférence) demande beaucoup de mémoire et de puissance de calcul. Jusqu'ici, les approches d'efficacité se concentraient soit sur le partage de paramètres, soit sur le calcul adaptatif, sans parvenir à combiner les deux. Mixture-of-Recursions (MoR) propose une nouvelle façon de combiner ces deux approches au sein d'un Transformer récursif. D'un côté, une même pile de couches est réutilisée à travers plusieurs étapes de récursion, ce qui réduit le nombre de paramètres; de l'autre, des routeurs légers permettent d'adapter dynamiquement la profondeur de calcul à chaque token. Cela qui réduit le coût mémoire et accélère les calculs. Ainsi, le calcul du score d'attention – ayant une complexité quadratique – se limite uniquement aux tokens encore actifs à une étape donnée, et le cache mémoire (*KV cache*) ne conserve que leurs clés-valeurs, ce qui améliore fortement l'efficacité d'accès. En somme, MoR ouvre une voie prometteuse vers la qualité des grands modèles, mais sans leurs énormes coûts de calcul.



Contents

1 Introduction

Les chercheurs ont découvert que lorsqu'on augmente énormément la taille des Transformers — jusqu'à des centaines de milliards de paramètres — ces modèles deviennent capables de choses remarquables : ils savent généraliser à partir de très peu d'exemples (few-shot learning) et montrent des capacités de raisonnement impressionnantes (comme l'ont démontré GPT-3, PaLM, LLaMA, Gemini, etc.).

Mais il y a un problème majeur : plus le modèle est grand, plus il devient lourd en mémoire et demande « plus de patates », je veux dire : plus de puissance de calcul. Cela signifie que son entraînement comme son utilisation deviennent extrêmement coûteux, au point que seuls les très grands centres de données (hyperscale datacenters) — ceux que possèdent Google, OpenAI ou Meta — peuvent réellement se le permettre. Pour la majorité des chercheurs, entreprises ou institutions, ces modèles sont donc inaccessibles.

C'est pour cela que la communauté a commencé à explorer d'autres approaches plus efficaces. Jusqu'ici, deux grandes pistes se dégagent : partage de paramètres et le calcul adaptatif.

- Supposons qu'on souhaite effectuer dix (10) tâches, au lieu d'acheter dix (10) machines différentes pour les faire, on pourrait acheter une seule machine configurable pour tous les traiter.
- Supposons que tu lis un livre, lorsque tu tombes sur des passages ou phrases faciles à comprendre, tu les passes rapidement sans perdre de temps, mais lorsque tu tombes sur des parties difficiles et compliqués, tu ralentis et réfléchis davantage pour mieux les appréhender.

Tous cela pour vous dire que jusqu'ici, il n'y a que deux grandes familles d'approches qui existent pour rendre les Transformers plus efficaces : la réutilisation de mêmes couches plusieurs fois (layer tying) pour économiser des paramètres, et l'arrêt plus tôt des calculs sur les tokens simples (early exiting) pour économiser du temps. Mais aucun modèle ne réussit vraiment à combiner ces deux stratégies dans une seule architecture. Les Recursive Transformers, qui réutilisent naturellement un même bloc de couches en boucle, semblaient prometteurs. Pourtant, en pratique, rendre cette récursion adaptative s'est révélé compliqué, si bien que la plupart des approches reviennent encore à une récursion fixe, inefficace pour un calcul réellement dynamique au niveau des tokens.

Pour résoudre ce problème qui embêche la combinaison des deux méthodes d'optimisation dans un seul modèle , MoR introduit Un petit module appelé **router** décide, pour chaque token, combien de fois il doit passer par la pile de couches, de tel sorte que, les tokens faciles se retrouvent avec peu de recursion et les tokens difficiles se retrouvent avec plus de recursion.

Dans un modèle transformer, chaque passage d'un token dans un bloc de Transformer génère et stocke des paires clé–valeur (KV) utilisées par le mécanisme d'attention. Mais, dans MoR, comme les tokens n'ont pas tous la même profondeur de calcul, alors on stocke uniquement les KV en fonction à la profondeur de chaque chaque token. Cela réduit donc le trafic mémoire (moins d'écritures/lectures inutiles) et augmente le débit (vitesse d'inférence) du modèle.

Le Mixture-of-Recursions (MoR) permet aux modèles de faire du raisonnement « silencieux » à l'intérieur même de leur espace latent, en appliquant plusieurs fois le même bloc de paramètres.

Contrairement aux méthodes qui réfléchissent sur un prompt augmenté (prompt enrichi) avant de générer, MoR fait ce raisonnement directement pendant la génération de chaque token. Grâce à son mécanisme de routage, il ajuste dynamiquement la profondeur de réflexion selon la difficulté du token, combinant ainsi compacité du modèle et calcul adaptatif efficace. Pour illustrer ce paragraphe, imaginons un écrivain qui rédige un text, Dans d'autres approches, comme le prompt augmenté, il prend des notes détaillées avant de commencer à écrire, puis rédige sans revenir dessus. Avec MoR, l'écrivain réfléchit en temps réel à chaque mot, et il peut décider de prendre plus de temps pour certains mots complexes (réflexion profonde) et moins de temps pour les mots simples.

2 Connaissances de Base Nécessaires

Pour comprendre ce papier, il faut se familiariser avec quelques concepts clés des modèles de type Transformers.

2.1 Architecture Transformer de base (Vaswani et al., 2017)

Un transformer est un modèle de réseau de neurones qui utilise des **mécanismes d'attention** pour traiter des séquences de données (comme du texte). Il est composé de couches (layers) empilées les unes sur les autres. Chaque couche a une **auto-attention** et un réseau **feed-forward**.

Imaginez un Transformer comme une usine avec plusieurs étages (couches/layers). Chaque étage traite l'information et la passe à l'étage suivant. Dans un modèle traditionnel comme GPT :

Entrée : "Le chat mange" $\rightarrow$ Couche $1 \rightarrow$ Couche $2 \rightarrow \ldots \rightarrow$ Couche $N \rightarrow$ Sortie : "le poisson"

À chaque étage, chaque mot (token) regarde tous les autres mots pour mieux se comprendre soi-même (c'est l'auto-attention), puis une petite usine interne (le feed-forward) pour calculer sa représentation.

2.2 Cache KV (Key-Value Cache)

Pour générer du texte de manière autoregressive (un mot à la fois), on doit éviter de recalculer les représentations des mots passés à chaque nouvelle étape. Le cache KV stocke les "clés" et "valeurs" de tous les mots déjà générés, ce qui rend le processus beaucoup plus rapide.

De façon analogique, lorsque vous lisez un livre, vous n'oubliez pas les chapitres précédents. Votre cerveau garde en cache les informations importantes. Le cache KV fait la même chose pour le modèle.

2.3 Partage de Paramètres (Parameter Sharing)

Au lieu d'avoir des poids uniques pour chaque couche, on utilise le même jeu de poids pour plusieurs couches. Cela réduit énormément la taille du modèle.

Exemple : Au lieu d'avoir 24 stations de travail différentes dans notre usine, on n'en a que 8, mais on fait passer la phrase 3 fois par les mêmes 8 stations. L'usine est plus petite (moins de paramètres) mais le travail est tout aussi profond (3 passages = 24 traitements effectifs).

2.4 Calcul Adaptatif (Adaptive Computation)

L'idée que tous les mots ne méritent pas le même effort de calcul. Les mots de structure ("le", "la", "et") sont simples, tandis que les mots de fond ("quantique", "philosophie") sont complexes. Le calcul adaptatif permet au modèle de "réfléchir" plus longtemps aux mots difficiles.

3 Le Problème posé

Les modèles de langage (LLMs) comme GPT-4 sont très puissants mais ont deux énormes défauts :

- Ils ont des milliards de paramètres, ce qui les rend très coûteux à entraîner et à stocker.
- Le mécanisme d'attention standard a une complexité quadratique $O(n^2)$. Pour générer de longues séquences, le calcul devient extrêmement lent et gourmand en énergie.

Les solutions existantes s'attaquent souvent à un seul de ces problèmes à la fois.

- *ALBERT* réduit la taille du modèle mais applique le même nombre de calculs à tous les tokens (inefficient).
- *Mixture-of-Depths* réduit le calcul pour les tokens "faciles" mais n'utilise pas de partage de paramètres, donc la taille du modèle reste grande.

Le problème fondamental est donc : Comment créer un modèle qui est à la fois petit (grâce au partage de paramètres) et rapide à l'inférence (grâce au calcul adaptatif) sans sacrifier la performance ? C'est ce problème que MoR résout.

4 La Solution : Mixture-of-Recursions (MoR)

MoR est une architecture unifiée qui combine intelligemment le partage de paramètres et le calcul adaptatif.

4.1 Concept de Base

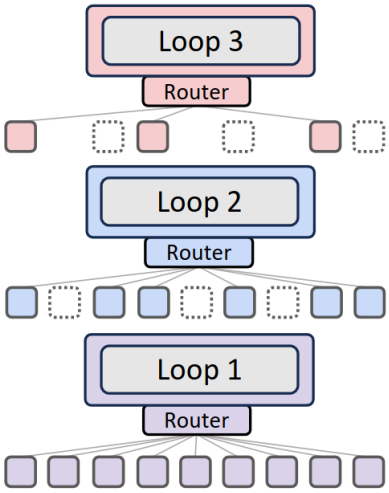
Au lieu d'empiler 24 couches différentes, MoR a un petit bloc de N couches partagées (par exemple, 8 couches). Ce bloc va être appliqué de manière répétée (récursive), jusqu'à N_r fois (par exemple, 3 fois). Le traitement effectif total est de $8 \times 3 = 24$ couches, mais on stocke que 8 couches de paramètres. Ce qui nous permet d'économiser de la mémoire stockage.

À la fin de chaque application du bloc récursif, un routeur léger décide pour chaque token s'il doit "sortir" (il a fini de réfléchir) ou s'il doit passer une nouvelle fois dans le bloc récursif (il a besoin de plus de réflexion). C'est le *Routage Adaptatif par Token* (voir figure 1). Cela permet d'économiser en calcul sur des mots simples à comprendre.

4.2 Expert choice routing

Imaginez qu'on place un filtre à chaque sortie d'une couche.

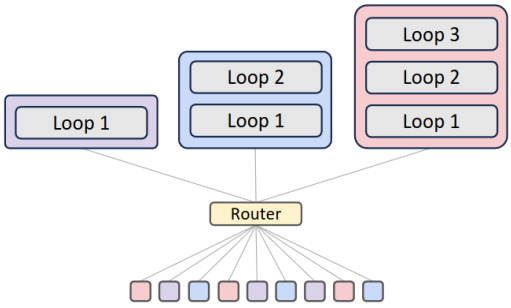
1. À la fin de la première étape de calcul, un "manager" (le routeur) évalue tous les coureurs et dit : "Seuls les 70% qui ont le plus d'énergie continuent !". Les autres (les mots simples comme `"le"`, `"la"`) quittent la piste.
2. Les tokens restants font une deuxième tour. À la fin, le manager regarde à nouveau et dit : "Maintenant, seuls les 40% meilleurs continuent !".
3. Le processus se répète, en gardant de moins en moins de tokens à chaque fois.



Chaque "étape de récursion" agit comme un expert qui choisit les tokens qu'il veut traiter. Ce qui permet un contrôle très précis du budget calcul à chaque étape. On sait exactement combien de tokens sont actifs.

4.3 Token choice routing

Le token lui-même, via sa représentation initiale, "choisit" en quelque sorte combien de calcul il va recevoir. Dès le début, un routeur regarde la représentation de chaque token et en fonction de cela il assigne une "profondeur de récursion" (nombre de tour).



Prenons par exemple la phrase suivante : `"Le chat philosophique mange la nourriture délicate."`, pour chaque mot, en fonction de leur représentation, le routeur va décider immédiatement du nombre de tours qu'il fera.

- `"Le"`, `"la"` —> 1 recursion (sort immédiatement après le premier passage, c'est un mot de structure);
- `"chat"`, `"mange"`, `"nourriture"` —> 2 tours (mots normaux);
- `"philosophique"`, `"délicate"` —> 3 tours (mots complexes qui nécessitent une "réflexion" profonde).

À la fin du permier passage, le routeur sélectionne `"philosophique"`, `"délicate"`, `"chat"`, `"mange"`. `"Le"` et `"la"` sortent. Ensuite, à la fin du 2ème passage, le routeur sélectionne

"philosophique" et "délicate". Les autres sortent. Au 3ème passage, seuls les deux mots complexes restants terminent leur traitement.

Chaque token parcourt le nombre de boucles qui lui a été assigné, puis s'arrête. Il n'y a pas de décision à prendre après le départ. En effet, c'est le token lui-même, via sa représentation initiale, qui "choisit" en quelque sorte combien de calcul il va recevoir.

Les mots simples ont été calculés rapidement, libérant les ressources pour que le modèle réfléchisse longuement aux mots complexes. Tout cela en réutilisant le même petit bloc de couches.

4.4 Headings: second level

$$\xi_{ij}(t) = P(x_t = i, x_{t+1} = j|y, v, w; \theta) = \frac{\alpha_i(t) a_{ij}^{w_t} \beta_j(t+1) b_j^{v_{t+1}}(y_{t+1})}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i(t) a_{ij}^{w_t} \beta_j(t+1) b_j^{v_{t+1}}(y_{t+1})} \tag{1}$$

4.4.1 Headings: third level

Paragraph

5 Examples of citations, figures, tables, references

The documentation for `natbib` may be found at

http://mirrors.ctan.org/macros/latex/contrib/natbib/natnotes.pdf

Of note is the command `\citet`, which produces citations appropriate for use in inline text. For example,

```
\citet{hasselmo} investigated\dots
```

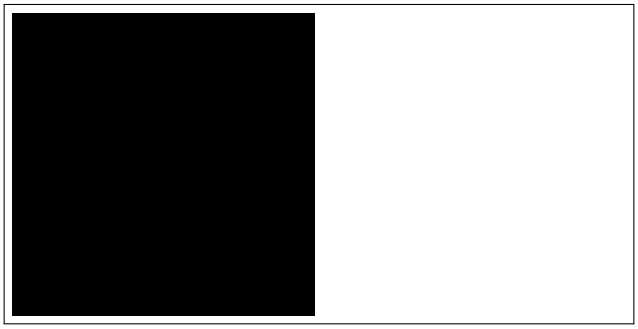
produces

Hasselmo, et al. (1995) investigated…

https://www.ctan.org/pkg/booktabs

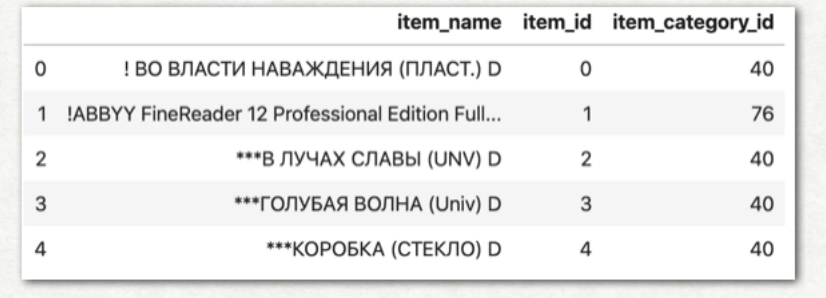
5.1 Figures

See Figure 3. Here is how you add footnotes. [1]



[1] Sample of the first footnote.

	Part	
Name	Description	Size (μm)
Dendrite	Input terminal	$\sim$100
Axon	Output terminal	$\sim$10
Soma	Cell body	up to 10^6



	item_name	item_id	item_category_id
0	! ВО ВЛАСТИ НАВАЖДЕНИЯ (ПЛАСТ.) D	0	40
1	!ABBYY FineReader 12 Professional Edition Full...	1	76
2	***В ЛУЧАХ СЛАВЫ (UNV) D	2	40
3	***ГОЛУБАЯ ВОЛНА (Univ) D	3	40
4	***КОРОБКА (СТЕКЛО) D	4	40

5.2 Tables

See awesome Table 1.

5.3 Lists

- Lorem ipsum dolor sit amet
- consectetur adipiscing elit.
- Aliquam dignissim blandit est, in dictum tortor gravida eget. In ac rutrum magna.

References